Supplementary Material: Learning to Simulate Complex Physics with Graph Networks

A. Supplementary GNS Model Details

Update mechanism. Our GNS implementation here uses semi-implicit Euler integration to update the next state based on the predicted accelerations:

$$\dot{\mathbf{p}}^{t_{k+1}} = \dot{\mathbf{p}}^{t_k} + \Delta t \cdot \ddot{\mathbf{p}}^{t_k}$$
$$\mathbf{p}^{t_{k+1}} = \mathbf{p}^{t_k} + \Delta t \cdot \dot{\mathbf{p}}^{t_{k+1}}$$

where we assume $\Delta t = 1$ for simplicity. We use this in contrast to forward Euler ($\mathbf{p}^{t_{k+1}} = \mathbf{p}^{t_k} + \Delta t \cdot \dot{\mathbf{p}}^{t_k}$) so the acceleration $\ddot{\mathbf{p}}^{t_k}$ predicted by the model can directly influence $\mathbf{p}^{t_{k+1}}$.

Optimizing parameters of learnable simulator. Learning a simulator s_θ, can in general be expressed as optimizing its parameters θ over some objective function,

$$\theta^* \leftarrow \arg_\theta \min \mathbb{E}_{\mathbb{P}(\mathbf{X}^{t_{0:K}})} L(\mathbf{X}^{t_{1:K}}, \tilde{\mathbf{X}}^{t_{1:K}}_{s_\theta, X^{t_0}}) \,.$$

$\mathbb{P}(\mathbf{X}^{t_{0:K}})$ represents a distribution over state trajectories, starting from the initial conditions X^{t_0}, over K timesteps. The $\tilde{\mathbf{X}}^{t_{1:K}}_{s_\theta, X^{t_0}}$ indicates the simulated rollout generated by s_θ given X^{t_0}. The objective function L, considers the whole trajectory generated by s_θ. In this work, we specifically train our GNS model on a one-step loss function, $L_{\text{1-step}}$, with

$$\theta^*_{\text{1-step}} \leftarrow \arg_\theta \min \mathbb{E}_{\mathbb{P}(\mathbf{X}^{t_{k:k+1}})} L_{\text{1-step}}(X^{t_{k+1}}, s_\theta(X^{t_k})) \,.$$

This imposes a stronger inductive bias that physical dynamics are Markovian, and should operate the same at any time during a trajectory.

In fact, we note that optimizing for whole trajectories may not actually not be ideal, as it can allow the simulator to learn biases which may not be hold generally. In particular, an L which considers the whole trajectory means θ^* does not necessarily equal the $\theta^*_{\text{1-step}}$ that would optimize $L_{\text{1-step}}$. This is because optimizing a capacity-limited simulator model for whole trajectories might benefit from producing greater one-step errors at certain times, in order to allow for better overall performance in the long term. For example, imagine simulating an undamped pendulum system, where the initial velocity of the bob is always zero. The physics dictate that in the future, whenever the bob returns to its initial position, it must always have zero velocity. If s_θ cannot learn to approximate this system exactly, and makes mistakes on intermediate timesteps, this means that when the bob returns to its initial position it might not have zero velocity. Such errors could accumulate over time, and causes large loss under an L which considers whole trajectories. The training process could overcome this by selecting θ^* which, for example, subtly encodes the initial position in the small decimal places of its predictions, which the simulator could then exploit by snapping the bob back to zero velocity when it reaches that initial position. The resulting s_{θ^*} may be more accurate over long trajectories, but not generalize as well to situations where the initial velocity is not zero. This corresponds to using the predictions, in part, as a sort of memory buffer, analogous to a recurrent neural network.

Of course, a simulator with a memory mechanism can potentially offer advantages, such as being better able to recognize and respect certain symmetries, e.g., conservation of energy and momentum. An interesting area for future work is exploring different approaches for training learnable simulators, and allowing them to store information over rollout timesteps, especially as a function for how the predictions will be used, which may favor different trade-offs between accuracy over time, what aspects of the predictions are most important to get right, generalization, etc.

B. Supplementary Experimental Methods

B.1. Physical Domains

Domain	Simulator (Dim.)	Max. # particles (approx)	Trajectory length	# trajectories (Train/ Validation/ Test)	Δt [ms]	Connectivity radius	Max. # edges (approx)
WATER-3D	SPH (3D)	13k	800	1000/100/100	5	0.035	230k
SAND-3D	MPM (3D)	20k	350	1000/100/100	2.5	0.025	320k
GOOP-3D	MPM (3D)	14k	300	1000/100/100	2.5	0.025	230k
WATER-3D-S	SPH (3D)	5.8k	800	1000/100/100	5	0.045	100k
BOXBATH	PBD (3D)	1k	150	2700/150/150	16.7	0.08	17k
WATER	MPM (2D)	1.9k	1000	1000/30/30	2.5	0.015	27k
SAND	MPM (2D)	2k	320	1000/30/30	2.5	0.015	21k
GOOP	MPM (2D)	1.9k	400	1000/30/30	2.5	0.015	19k
MULTIMATERIAL	MPM (2D)	2k	1000	1000/100/100	2.5	0.015	25k
FLUIDSHAKE	MPM (2D)	1.3k	2000	1000/100/100	2.5	0.015	20k
FLUIDSHAKE-BOX	MPM (2D)	1.5k	1500	1000/100/100	2.5	0.015	19k
WATERDROP	MPM (2D)	1k	1000	1000/30/30	2.5	0.015	12k
WATERDROP-XL	MPM (2D)	7.1k	1000	1000/100/100	2.5	0.01	210k
WATERRAMPS	MPM (2D)	2.3k	600	1000/100/100	2.5	0.015	26k
SANDRAMPS	MPM (2D)	3.3k	400	1000/100/100	2.5	0.015	32k
RANDOMFLOOR	MPM (2D)	3.4k	600	1000/100/100	2.5	0.015	44k
CONTINUOUS	MPM (2D)	4.3k	400	1000/100/100	2.5	0.015	47k

B.2. Implementation Details

Input and output representations. We define the input "velocity" as average velocity between the current and previous timestep, which is calculated from the difference in position, $\dot{\mathbf{p}}^{t_k} \equiv \mathbf{p}^{t_k} - \mathbf{p}^{t_{k-1}}$ (omitting constant Δt for simplicity). Similarly, we define "acceleration" as average acceleration between the next and current timestep, $\ddot{\mathbf{p}}^{t_k} \equiv \dot{\mathbf{p}}^{t_{k+1}} - \dot{\mathbf{p}}^{t_k}$. Accelerations are thus calculated as, $\ddot{\mathbf{p}}^{t_k} = \mathbf{p}^{t_{k+1}} - 2\mathbf{p}^{t_k} + \mathbf{p}^{t_{k-1}}$.

We express the material type (water, sand, goop, rigid, boundary particle) as a particle feature, $\mathbf{a}_i$, represented with a learned embedding vector of size 16. For datasets with fixed flat orthogonal walls, instead of adding boundary particles, we add a feature to each node indicating the vector distance to each wall. Crucially, to maintain spatial translation invariance, we clip this distance to the connectivity radius R, achieving a similar effect to that of the boundary particles. In FLUIDSHAKE, particle positions were provided in the coordinate frame of the container, and the container position, velocity and acceleration were provided as 6 global features. In CONTINUOUS a single global scalar was used to indicate the friction angle of the material.

Building the graph. We construct the graph by, for each particle, finding all neighboring particles within the connectivity radius. We use a standard k-d tree algorithm for this search. The connectivity radius was chosen, such that the number of neighbors in roughly in the range of $10 - 20$. We however did not find it necessary to fine-tune this parameter: All 2D scenes of the same resolution share $R = 0.015$, only the high-res 2D and 3D scenes, which had substantially different particle densities, required choosing a different radius. Note that for these datasets, the radius was simply chosen once based on particle neighborhood size adn total number of edges, and was not fine-tuned as a hyperparameter.

Neural network parametrizations. We also trained models where we replaced the deep encoder and decoder MLPs by simple linear layers without activations, and observed similar performance.

B.3. Training

Noise injection. Because our models take as input a sequence of states (positions and velocities), we draw independent samples $\sim \mathcal{N}(0, \sigma_v = 0.0003)$, for each input state, particle and spatial dimension, before each training step. We accumulate

them across time as a random walk, and use this to perturb the stack of input velocities. Based on the updated velocities, we then adjust the position features, such that $\dot{\mathbf{p}}^{t_k} \equiv \mathbf{p}^{t_k} - \mathbf{p}^{t_{k-1}}$ is maintained, for consistency. We also experimented with other types of noise accumulation, as detailed in Section C.

Another way to address differences in training and test input distributions is to, during training, provide the model with its own predictions by rolling out short sequences. Ummenhofer et al. (2020), for example, train with two-step predictions. However computing additional model predictions are more expensive, and in our experience may not generalize to longer sequences as well as noise injection.

Normalization. To normalize our inputs and targets, we compute the dataset statistics during training. Instead of using moving averages, which could shift in cycles during training, we instead build exact mean and variance for all of the input and target particle features up seen up to the current training step l, by accumulating the sum, the sum of the squares and the total particle count. The statistics are computed after noise is applied to the inputs.

Loss function and optimization procedures. We load the training trajectories sequentially, and use them to generate input and target pairs (from a 1000-step long trajectory we generate 995 pairs, as we condition on 5 past states), and sample input-target pairs from a shuffle buffer of size 10k. The acceleration targets are computed from the sequence of positions before adding noise to the input, and then adjusted by removing the input velocity noise accumulated at the final step of the random walk. By doing so, the model learns to predict an output acceleration that, after integrating with the Euler integrator ($\Delta t = 1$), yields a next-step velocity that matches the next-step ground-truth velocity, hence correcting for the noise in input velocity. This is equivalent to defining the loss on next-step ground-truth velocity. Rigid obstacles, such as the ramps in WATER-RAMPS, are represented as *boundary particles*. Those particles are treated identical to regular particles, but they are masked out of the loss.

Due to normalization of predictions and targets, our prediction loss is normalized, too. This allows us to choose a scale-free learning rate, across all datasets. To optimize the loss, we use the Adam optimizer (Kingma & Ba, 2014) (a form of stochastic gradient descent) with a nominal mini-batch size of 2 examples, averaging the loss for all particles in the batch. We performed a maximum of 20M gradient update steps, with an exponentially decaying learning rate, $\alpha(j)$, where on the j-th gradient update step, $\alpha(j) = \alpha_{\text{final}} + (\alpha_{\text{start}} - \alpha_{\text{final}}) \cdot 0.1^{(j \cdot 5 \cdot 10^6)}$, with $\alpha_{\text{start}} = 10^{-4}$ and $\alpha_{\text{final}} = 10^{-6}$. While models can train in significantly less steps, we avoid aggressive learning rates to reduce variance across datasets and make comparisons across settings more fair.

We train our models using second generation TPUs and V100 GPUs interchangeably. For our datasets, we found that training time per example with a single TPU core or a single V100 GPU was about the same. TPUs allowed for faster training through fast batch parallelism (each of the two training examples in the batch runs on a separate TPU core in the same TPU chip). Furthermore, since TPU cores require fixed size tensors, instead of just padding each training example up to a maximum number of nodes/edges, we set the fixed size to correspond to the largest graph in the dataset, and, at each step, build a larger minibatch using multiple training examples whenever they would fit within the set fixed size, before adding the padding. This yielded an effective batch size between 1 (large examples) and 3 examples (small examples) per device (2 to 6 examples per batch when batch parallelism is taken into account) and is equivalent to setting a mini batch size in terms of total number of particles per batch. For easier comparison, we also replicated this procedure on the GPU training.

Additionally, for our largest systems (¿ 100k edges) we also used model parallelism (a single training example distributed over multiple TPU cores)(Kumar et al., 2019), over a total of 16 TPU cores per example (16 TPU chips in total, given the batch parallelism of 2 and that each TPU chip has two TPU cores).

B.4. Distributional Evaluation Metrics

An MSE metric of zero indicates that the model perfectly predicts where each particle has traveled. However, if the model's predicted positions for particles A and B exactly match true positions of particles B and A, respectively, the MSE could be high, even though the predicted and true distributions of particles match. So we also explored two metrics that are invariant under particle permutations, by measuring differences between the distributions of particles: optimal transport (OT) (Villani, 2003) using 2D or 3D Wasserstein distance and approximated by the Sinkhorn Algorithm (Cuturi, 2013), and maximum mean discrepancy (MMD) (Gretton et al., 2012) with a Gaussian kernel bandwidth of $\sigma = 0.1$. These distributional metrics may be more appropriate when the goal is to predict what regions of the space will be occupied by the simulated material, or when the particles are sampled from some continuous representation of the state and there are no "true" particles to compare predictions to. We will analyze some of our results using those metrics in Section C.

C. Supplementary Results

C.1. Architectural Choices with Minor Impact on Performance

We provided additional ablation scans on the GOOP dataset in Figure C.1, which show that the model is robust to typical hyperparameter changes.

Number of input steps C. (Figure C.2a,b) We observe a significant improvement from conditioning the model on just the previous velocity ($C = 1$) to the two most recent velocities ($C = 2$), but find similar performance for larger values of C. All our models use $C = 5$. Note that because we approximate the velocity as the finite difference of the position, the model requires the most recent $C + 1$ positions to compute the last C velocities.

Latent and MLP layer sizes. (Figure C.2c,d) The performance does not change much as function of the latent and MLP hidden sizes, for sizes of 64 or larger.

Number of MLP hidden layers. (Figure C.2e,f) Except for the case of zero hidden layers (linear layer), the performance does not change much as a function of the MLP depth.

Use MLP encoder & decoder. (Figure C.2g,h) We replaced the MLPs used in the encoder and decoder by linear layers (single matrix multiplication followed by a bias), and observed no significant changes in performance.

Use LayerNorm. (Figure C.2i,j) In small datasets, we typically observe slightly better performance when LayerNorm (Ba et al., 2016) is not used, however, enabling it provides additional training stability for the larger datasets.

Include self-edges. (Figure C.2k,l) The model performs similarly regardless of whether self-edges are included in the message passing process or not.

Use global latent. (Figure C.2m,n) We enable the global mechanisms of the GNs. This includes both, explicit input global features (instead of appending them to the nodes) and a global latent state that updates after every message passing step using aggregated information from all edges and nodes. We do not find significant differences when doing this, however we speculate that generalization performance for systems with more particles than used during training would be affected.

Use edges latent. (Figure C.2o,p) We disabled the updates to the latent state of the edges that is performed at each edge message passing iteration, but found no significant differences in performance.

Add gravity acceleration. (Figure C.2q,r) We attempted adding gravity accelerations to the model outputs in the update procedure, so the model would not need to learn to predict a bias acceleration due to gravity, but found no significant performance differences.

C.2. Noise-Related Training Parameters

We provide some variations related to how we add noise to the input data on the GOOP dataset in Figure C.2.

Noise type. (Figure C.2a,e) We experimented with 4 different modes for adding noise to the inputs. *only_last* adds noise only to the velocity of the most recent state in the input sequence of states. *correlated* draws a single per-particle and per-dimension set of noise samples, and applies the same noise to the velocities of all input states in the sequence. *uncorrelated* draws independent noise for the velocity of each input state. *random_walk* draws noise for each input state, adding it to the noise of the previous state in sequence as in a random random walk, as an attempt to simulate accumulation of error in a rollout. In all cases the input states positions are adjusted to maintain $\dot{\mathbf{p}}^{t_k} \equiv \mathbf{p}^{t_k} - \mathbf{p}^{t_{k-1}}$. To facilitate the comparison, the variance of the generated noise is adjusted so the variance of the velocity noise at the last step is constant. We found the best rollout performance for *random_walk* noise type.

Noise Std. (Figure C.2b,f) This is described in the main text, included here for completeness.

Reconnect graph after noise. (Figure C.2c,g) We found that the performance did not change regardless of whether we recalculated the connectivity of the graph after applying noise to the positions or not.

Fraction of position noise to correct. (Figure C.2d,h) In the process of corrupting the input position and velocity features with noise, we adjust the target accelerations such as the acceleration predicted by the model would compensate for the noise in input velocity (0%). For this ablation we modify the target accelerations such as the model learns to predict accelerations that would instead correct the noise in the input position (100%). Note that this happens implicitly when the loss is defined directly on next-step ground-truth position, regardless of whether the inputs are perturbed with noise (Sanchez-Gonzalez

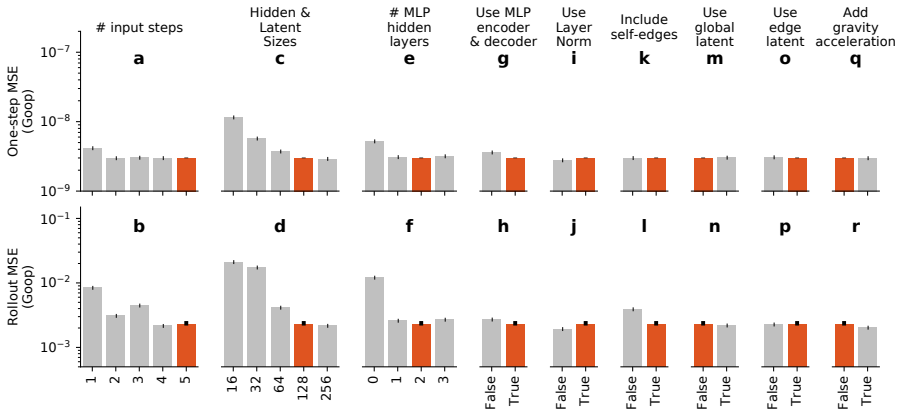

Figure C.1. Additional ablations (grey) on the GOOP dataset compared to our default model (red). Error bars display lower and higher quartiles, and are shown for the default parameters. The same vertical limits from Figure 4 are reused for easier qualitative scale comparison.

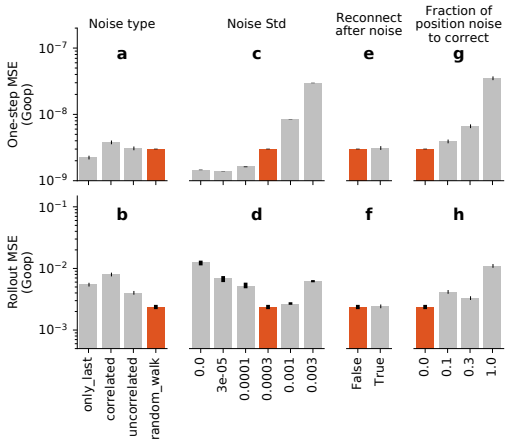

Figure C.2. Noise-related training variations (grey) on the GOOP dataset compared to our default model (red). Error bars display lower and higher quartiles, and are shown for the default parameters. The same vertical limits from Figure 4 are reused for easier qualitative scale comparison.

et al., 2018) or the inputs have noise due to model error after a rollout (Ummenhofer et al., 2020). We also investigate two intermediary points (10% and 30%), which implies training the model to correct a mix of the position and the velocity noise. Note that since the model uses a simple Euler update to compute output position from output velocity, it is impossible to exactly correct for both position and velocity noise; we have to choose one or the other (or a compromise in between). Figure C.2d,h show that asking the model to correct for the position noise instead of the velocity noise leads to worse performance.

C.3. Distributional Evaluation Metrics

Generally we find that MSE and the distributional metrics lead to generally similar conclusions in our analyses (see Figure C.3), though we notice that differences in the distributional metrics' values for qualitatively "good" and "bad" rollouts can be more prominent, and match more closely with our subjective visual impressions. Figure C.4 shows the rollout errors as a function of the key architectural choices from Figure 4 using these distributional metrics.

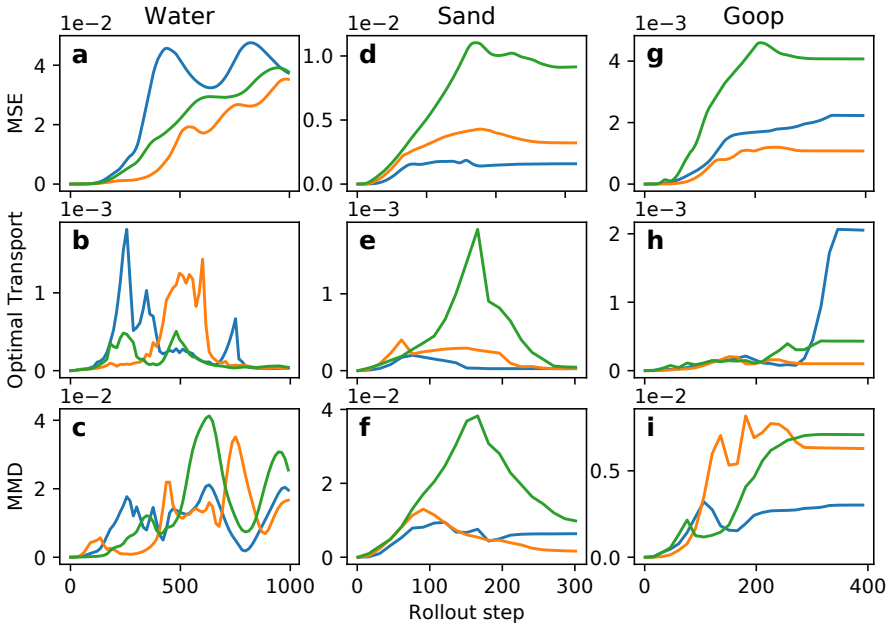

Figure C.3. Rollout error of one of our models as a function of time for water, sand and goop, using MSE, Optimal transport and MMD as metrics. Figures show curves for 3 trajectories in the evaluation datasets (colored curves). MSE tends to grow as function of time due to the chaotic character of the systems. Optimal transport and MMD, which are invariant to particle permutations, tend to decrease for WATER and SAND towards the end of the rollout as they equilibrate towards a single consistent group of particles, due to the lower friction/viscosity. For GOOP, which can remain in separate clusters in a chaotic manner, the Optimal Transport error can still be very high.

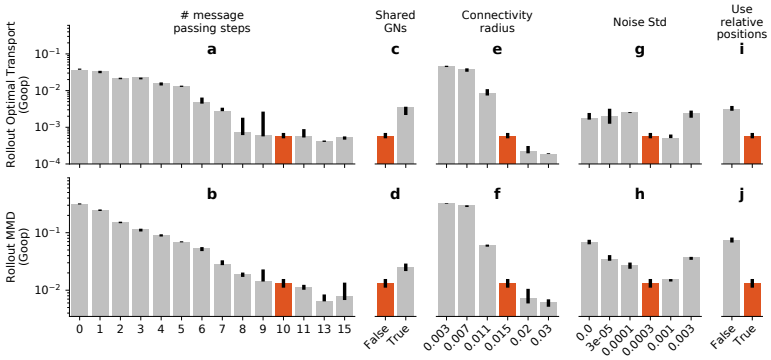

Figure C.4. Effect of different ablations on Optimal Transport (top) and Maximum Mean Discrepancy (bottom) rollout errors. Bars show the median seed performance averaged across the entire test dataset. Error bars display lower and higher quartiles, and are shown for the default parameters.

C.4. Quantitative Results on all Datasets

Domain	Mean Squared Error		Optimal Transport		Maximum Mean Discrepancy ($\sigma = 0.1$)	
	One-step $\times 10^{-9}$	Rollout $\times 10^{-3}$	One-step $\times 10^{-9}$	Rollout $\times 10^{-3}$	One-step $\times 10^{-9}$	Rollout $\times 10^{-3}$
WATER-3D	8.66	10.1	26.5	0.165	7.32	0.368
SAND-3D	1.42	0.554	4.29	0.138	11.9	2.67
GOOP-3D	1.32	0.618	4.05	0.208	22.4	5.13
WATER-3D-S	9.66	9.52	29.9	0.222	6.9	0.192
BOXBATH	54.5	4.2	–	–	–	–
WATER	2.82	17.4	6.19	0.468	10.6	7.66
SAND	6.23	2.37	11.8	0.193	32.6	6.79
GOOP	2.91	1.89	6.14	0.419	20.3	7.76
MULTIMATERIAL	1.81	16.9	–	–	–	–
FLUIDSHAKE	2.1	20.1	4.13	0.591	12.1	9.84
FLUIDSHAKE-BOX	1.33	4.86	–	–	–	–
WATERDROP	1.52	7.01	3.31	0.273	11.1	5.99
WATERDROP-XL	1.23	14.9	3.03	0.209	11.6	4.34
WATERRAMPS	4.91	11.6	10.3	0.507	14.3	7.68
SANDRAMPS	2.77	2.07	6.13	0.187	15.7	4.81
RANDOMFLOOR	2.77	6.72	5.8	0.276	14	3.53
CONTINUOUS	2.06	1.06	4.63	0.0709	23.1	3.57

C.5. Quantitative Generalization Results on CONTINUOUS Domain

See Figure C.5.

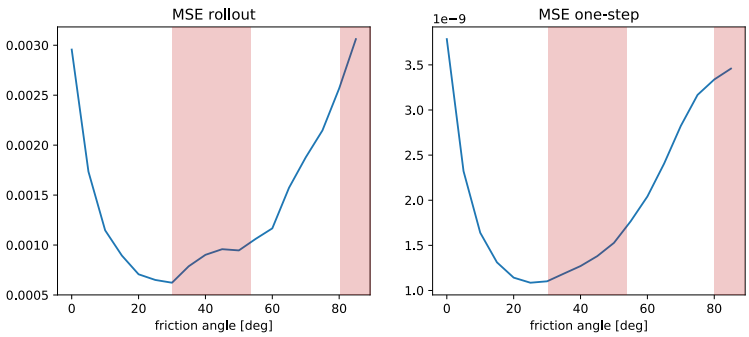

Figure C.5. Rollout and one-step error as a function of the friction angle in the CONTINUOUS domain. Regions highlighted in red correspond to values of the friction angle not observed during training. Our results show that a model trained in the $[0°, 30°]$ and $[55°, 80°]$ ranges can produce good predictions across all friction angles (see also videos), with only marginally higher errors in the $[30°, 55°]$ range that was not seen during training. Note that the dynamics at very low and very high friction angles are simply more complicated and harder to learn, hence the higher error.

C.6. Inference Times

The following table compares the performance of our learned GNS model (evaluated on a single V100 GPU) to the performance of the simulator used to generate the data (run on a 6-core workstation CPU) for each of the datasets. The learned GNS model has inference times comparable to that of the ground truth simulator used to generate the data. Note that

these results are purely informative and calculated post-hoc, as the main goal of this work was not to improve simulation time. We expect better performance could be achieved by making predictions for longer time steps, or using optimized implementations for neighborhood graph calculation (which we evaluated out-of-graph on the CPU using KDTrees).

Domain	**Simulator (Dim.)**	**Mean # particles per graph (approx)**	**Mean # edges per graph (approx)**	**Simulator time per step [s]**	**Learned GNS time per step including neighborhood computation [s] (relative to simulator)**
WATER-3D	SPH (3D)	7.8k	110k	0.104	0.358 (345%)
SAND-3D	MPM (3D)	9.8k	140k	0.221	0.336 (152%)
GOOP-3D	MPM (3D)	7.8k	120k	0.199	0.247 (124%)
WATER-3D-S	SPH (3D)	3.8k	55k	0.053	0.0683 (129%)
BOXBATH	PBD (3D)	1k	15k	–	0.0475 (–)
WATER	MPM (2D)	1.1k	12k	0.037	0.0579 (156%)
SAND	MPM (2D)	1.2k	11k	0.045	0.048 (107%)
GOOP	MPM (2D)	1k	9.2k	0.04	0.0301 (75.1%)
MULTIMATERIAL	MPM (2D)	1.6k	16k	0.049	0.0595 (121%)
FLUIDSHAKE	MPM (2D)	1.3k	13k	0.039	0.0257 (65.8%)
FLUIDSHAKE-BOX	MPM (2D)	1.4k	13k	0.048	0.133 (277%)
WATERDROP	MPM (2D)	0.6k	4.8k	0.05	0.0256 (51.3%)
WATERDROP-XL	MPM (2D)	4.3k	83k	0.166	0.192 (116%)
WATERRAMPS	MPM (2D)	1.5k	13k	0.071	0.0506 (71.3%)
SANDRAMPS	MPM (2D)	2.3k	21k	0.077	0.0691 (89.8%)
RANDOMFLOOR	MPM (2D)	2.3k	24k	0.076	0.0634 (83.4%)
CONTINUOUS	MPM (2D)	2.4k	22k	0.072	0.0919 (128%)

The table below shows the inference time for batches padded to a maximum size with pre-computed neighborhoods (which is equivalent to inference on a single large graph). Comparing to the previous table, we indeed observe that most of the computation time was spent on neighborhood computation rather than on graph neural network inference.

Domain	**# particles in batch**	**# edges in batch**	**Learned GNS time per step without neighborhood computation [s] (relative to learned GNS in previous table)**[8]
WATER-3D	14k	245k	0.071 (19.8%)
SAND-3D	19k	320k	0.086 (25.6%)
GOOP-3D	15k	230k	0.109 (44.2%)
WATER-3D-S	6k	120k	0.04 (58.6%)
BOXBATH	1k	18k	0.017 (35.8%)
WATER	2k	31k	0.025 (43.2%)
SAND	2k	21k	0.018 (37.5%)
GOOP	2k	21k	0.019 (63.2%)
MULTIMATERIAL	2k	27k	0.018 (30.3%)
FLUIDSHAKE	1.4k	23k	0.017 (66.2%)
FLUIDSHAKE-BOX	1.5k	20k	0.019 (14.3%)
WATERDROP	2k	18k	0.023 (89.7%)
WATERDROP-XL	8k	300k	0.057 (29.7%)
WATERRAMPS	2.5k	28k	0.017 (33.6%)
SANDRAMPS	3.5k	35k	0.023 (33.3%)
RANDOMFLOOR	3.5k	46k	0.023 (36.3%)
CONTINUOUS	5k	50k	0.033 (35.9%)

[8] Note that these batches are equivalent to running a single graph larger than those in the previous table, so "(relative to learned GNS in previous table)" only provides an upper bound on the fraction of time spent on graph neural network inference.

C.7. Example Failure Cases

In this video, we show two of the failure cases we sometimes observe with the GNS model. In the BOXBATH domain we found that our model could accurately predict the motion of a rigid block, and maintain its shape, without requiring explicit mechanisms to enforce solidity constraints or providing the rest shape to the network. However, we did observe limits to this capability in a harder version of BOXBATH, which we called FLUIDSHAKE-BOX, where the container is vigorously shaken side to side, over a rollout of 1500 timesteps. Towards the end of the trajectory, we observe that the solid block starts to deform. We speculate the reason for this is that GNS has to keep track of the block's original shape, which can be difficult to achieve over long trajectories given an input of only 5 initial frames.

In the second example, a *bad* seed of our model trained on the GOOP domain predicts a blob of goop stuck to the wall instead of falling down. We note that in the training data, the blobs do sometimes stick to the wall, though it tends to be closer to the floor and with different velocities. We speculate that the intricacies of static friction and adhesion may be hard to learn—to learn this behaviour more robustly, the model may need more exposure to fall versus sticking phenomena.

D. Supplementary Baseline Comparisons

D.1. Continuous Convolution (CConv)

Recently Ummenhofer et al. (2020) presented Continuous Convolution (CConv) as a method for particle-based fluid simulation. We show that CConv can also be understood in our framework, and compare CConv to our approach on several tasks.

Interpretation.

While Ummenhofer et al. (2020) state that "Unlike previous approaches, we do not build an explicit graph structure to connect the particles but use spatial convolutions as the main differentiable operation that relates particles to their neighbors.", we find we can express CConv (which itself is a generalization of CNNs) as a GN (Battaglia et al., 2018) with a specific type of edge update function.

CConv relates to CNNs (with stride of 1) and GNs in two ways. First, in CNNs, CConv, and GNs, each element (e.g., pixel, feature vector, particle) is updated as a function of its neighbors. In CNNs the neighborhood is fixed and defined by the kernel's dimensions, while in CConv and GNs the neighborhood varies and is defined by connected edges (in CConv the edges connect to nearest neighbors).

Second, CNNs, CConv, and GNs all apply a function to element i's neighbors, $j \in \mathcal{N}(i)$, pool the results from within the neighborhood, and update element i's representation. In a CNN, this is computed as, $f'_i = \sigma\left(\mathbf{b} + \sum_{j \in \mathcal{N}(i)} W(\boldsymbol{\tau}_{i,j}) f_j\right)$, where $W(\boldsymbol{\tau}_{i,j})$ is a matrix whose parameters depend on the displacement between the grid coordinates of i and j, $\boldsymbol{\tau}_{i,j} = \mathbf{x}_j - \mathbf{x}_i$ (and $\mathbf{b}$ is a bias vector, and σ is a non-linear activation). Because there are a finite set of $\boldsymbol{\tau}_{i,j}$ values, one for each coordinate in the kernel's grid, there are a finite set of $W(\boldsymbol{\tau}_{i,j})$ parameterizations.

CConv uses a similar formula, except the particles' continuous coordinates mean a choice must be made about how to parameterize $W(\boldsymbol{\tau}_{i,j})$. Like in CNNs, CConv uses a finite set of distinct weight matrices, $\hat{W}(\hat{\boldsymbol{\tau}}_{i,j})$, associated with the discrete coordinates, $\hat{\boldsymbol{\tau}}_{i,j}$, on the kernel's grid. For the continuous input $\boldsymbol{\tau}_{i,j}$, the nearest $\hat{W}(\boldsymbol{\tau}_{i,j})$ are interpolated by the fractional component of $\boldsymbol{\tau}_{i,j}$. In 1D this would be linear interpolation, $W(\boldsymbol{\tau}_{i,j}) = (1-d)\,\hat{W}(\lfloor \boldsymbol{\tau}_{i,j} \rfloor) + d\,\hat{W}(\lceil \boldsymbol{\tau}_{i,j} \rceil))$, where $d = \boldsymbol{\tau}_{i,j} - \lfloor \boldsymbol{\tau}_{i,j} \rfloor$. In 3D, this is trilinear interpolation.

A GN can implement CNN and CConv computations by representing $\boldsymbol{\tau}_{i,j}$ using edge attributes, $\mathbf{e}_{i,j}$, and an edge update function which uses independent parameters for each $\boldsymbol{\tau}_{i,j}$, i.e., $\mathbf{e}'_{i,j} = \phi^e(\mathbf{e}_{i,j}, \mathbf{v}_i, \mathbf{v}_j) = \phi^e_{\boldsymbol{\tau}_{i,j}}(\mathbf{v}_j)$. Beyond their displacement-specific edge update function, CNNs and CConv are very similar to how graph convolutional networks (GCN) (Kipf & Welling, 2016) work. The full CConv update as described in Ummenhofer et al. (2020) is, $f'_i = \frac{1}{\psi(\mathbf{x}_i)} \sum_{j \in \mathcal{N}(\mathbf{x}_i, R)} a\left(\mathbf{x}_j, \mathbf{x}_i\right) f_j \;\; g\left(\Lambda\left(\mathbf{x}_j - \mathbf{x}_i\right)\right)$. In particular, it indexes into the weight matrices via a polar-to-Cartesian coordinate transform, Λ, to induce a more radially homogeneous parameterization. It also uses a weighted sum over the particles in a neighborhood, where the weights, $a(\mathbf{x}_j, \mathbf{x}_i)$, are proportional to the distance between particles. And it includes a normalization, $\psi(\mathbf{x}_i)$, for neighborhood size, they set it to 1.

Performance comparisons.

We implemented the CConv model, loss and training procedure as described by Ummenhofer et al. (2020). For simplicity, we only tested the CConv model on datasets with flat walls, rather than those with irregular geometry. This way we could omit the initial convolution with the boundary particles and instead give the fluid particles additional simple features indicating the vector distance to each wall, clipped by the radius of connectivity, as in our model. This has the same spatial constraints as CConv with boundary particles in the wall, and should be as or more informative than boundary particles for square containers. Also, for environments with multiple materials, we appended a particle type learned embedding to the input node features.

To be consistent with Ummenhofer et al. (2020), we used their batch size of 16, learning rate decay of 10^{-3} to 10^{-5} for 50k iterations, and connectivity radius of 4.5x the particle radius. We were able to replicate their results on PBD/FLeX and SPH simulator datasets similar to the datasets presented in their paper. To allow a fair comparison when evaluating on our MPM datasets, we performed additional hyperparameter sweeps over connectivity particle radius, learning rate, and number of training iterations using our GOOP dataset. We used the best-fitting parameters on all datasets, analogous to how we selected hyperparameters for our GNS model.

We also implemented variations of CConv which used noise to corrupt the inputs during training (instead of using 2-step loss), as we did with GNS. We found that the noise improved CConv's rollout performance on most datasets. In our comparisons, we always report performance for the best-performing CConv variant.

Our qualitative results show CConv can learn to simulate sand reasonably well. But it struggled to accurately simulate solids with more complex fine grained dynamics. In the BOXBATH domain, CConv simulated the fluid well, but struggled to keep the box's shape intact. In the GOOP domain, CConv struggled to keep pieces of goop together and handle the rest state, while in MULTIMATERIAL it exhibited local "explosions", where regions of particles suddenly burst outward (see video).

Generally CConv's performance is strongest for simulating water-like fluids, which is primarily what it was applied to in the original paper. However it still did not match our GNS model, and for other materials, and interactions between different materials, it was clearly not as strong. This is not particularly surprising, given that our GNS is a more general model, and our neural network implementation has higher capacity on several axes, e.g., more message-passing steps, pairwise interaction functions, more flexible function approximators (MLPs with multiple internal layers versus single linear/non-linear layers in CConv).

D.2. DPI

We trained our model on the Li et al. (2018)'s BOXBATH dataset, and directly compared the qualitative behavior to the authors' demonstration video in this comparison. To provide a fair comparison to DPI, we show a model conditioned on just the previous velocity (C=1) in the above comparison video[9]. While DPI requires a specialized hierarchical mechanism and forced all box particles to preserve their relative displacements with each other, our GNS model faithfully represents the the ground truth trajectories of both water and solid particles without any special treatment. The particles making up the box and water are simply marked as a two different materials in the input features, similar to our other experiments with sand, water and goop. We also found that our model seems to also be more accurate when predicting the fluid particles over the long rollout, and it is able to perfectly reproduce the layering effect for fluid particles at the bottom of the box that exists in the ground truth data.

[9]We also ran experiments with C=5 and did not find any meaningful difference in performance. The results in Table 1 and the corresponding example video are run with C=5 for consistency with our other experiments