Trick or treat? Evaluating stability strategies in graph network-based simulators

Omer Rochman Sharabi
Universty of Liège
o.rochman@uliege.be

Gilles Louppe
Universty of Liège

Abstract

Particle-based simulators are ubiquitous in science and engineering, but some are expensive both in terms of time and compute. In simulations where local interactions play a major role, graph network-based simulators (GNS) show promise to address these issues due to their ability to model local interactions and relationships through the graph structure. However, their autoregressive nature makes them susceptible to distribution shifts. Numerous strategies, or tricks, have been proposed to address this issue. In this work, we evaluate three of them: adding a random walk to the input, taking the loss of a sequence, and the pushforward trick. We find that these tricks fail to address the underlying problem, even when the dynamics are relatively simple.

1 Introduction

Particle-based simulators offer flexibility to model a large class of problems, such as molecular dynamics [1], planetary orbits, or the behavior of groups of animals, such as flocks of birds [2] or human crowds [3]. Interestingly, particle-based simulators are not limited to problems where particles are physically present. For instance, in smoothed-particle hydrodynamics (SPH), a fluid is discretized as a set of particles, and their movement models the evolution of the fluid [4, 5]. A notable example of SPH is the simulation of the formation of the Moon through a collision between Earth and Theia [6]. Unfortunately, particle-based simulators are often expensive to run, which has prompted a recent surge of interest in surrogating numerical simulators with neural network counterparts. Within the domain of particle-based simulators, graph network-based simulators (GNS) have been used in [7–10], among others, and continuous convolutions in [11, 12].

GNS are trained to predict the next N states of a system given its current state, where N is usually much smaller than the total simulation length due to memory and compute limits. During inference, however, the simulator has to predict the next $L \gg N$ steps, using iteratively its output as input. Because of that, errors accumulate over time, gradually shifting the input distribution, which, in turn, generates more errors and more shift. Various solutions have been proposed to overcome this instability. For example, Sanchez-Gonzalez et al. [7] inject structured noise into the training inputs to simulate the noisy rollout, Lam et al. [10] introduces a training schedule which carefully increases N during training, Prantl et al. [12] unroll the simulator for M steps before using those steps as the input to predict the next N steps, and Brandstetter et al. [9] use a denoising based approach inspired by the zero-stability criteria from numerical methods.

The contribution of this work, in the context of particle-based simulators, is to compare approaches proposed to improve the stability of long unrolls. While those approaches may work in the context in which they were proposed (we verify this for the random walk trick [7]), we find that they are no silver bullet and fail to stabilize unrolls of a simulation undergoing distribution shift. Code and data will be made available at `https://github.com/OmerRochman/stability-strategies`.

Machine Learning and the Physical Sciences Workshop, NeurIPS 2023.

2 Graph network-based simulators

Let $x_t, a_t \in \mathbb{R}^{C \times D}$ be the state of the particle system at time t, that is, a list of C vectors in $\mathbb{R}^D$, each corresponding to the position and acceleration of a particle. The acceleration is computed from x_t using finite difference. Further, let S be the true simulator and define a trajectory as a set of states $x_{0:L} = \{x_0, x_1, ..., x_L\} \in \mathbb{R}^{L \times C \times D}$ created by applying S iteratively, $x_t = (S \circ ... \circ S)(x_0) = S^t(x_0)$. Here the exponent denotes function composition. Let $\hat{S}$ be a learned simulator and $\hat{x}_{0:L}$ a trajectory generated from x_0 by $\hat{S}$. Our goal is for $\hat{S}$ to approximate S as closely as possible.

An encode-process-decode GNS consists of three components. The encoder maps the nodes and edges of the input graph into latent space, the processor performs message passing between neighbors in latent space, and the decoder maps node features from latent space back into real space. The model takes as input a history of the last n states $x_{t-n:t} = \{x_{t-n}, ..., x_t\}$ and predicts the acceleration a_t. The acceleration is integrated using an Euler step $x_{t+1} = x_t + v_{t+1}$, where $v_{t+1} = v_t + a_t$. Using this notation, $\hat{S}(x_t) = x_t + v_t + \text{GNS}(x_{t-n:t})$. A critical aspect of the model is normalization. The input to the model is normalized to have zero mean and unit variance. During training the output of the model, assumed to be normalized, is compared against the normalized target acceleration. During inference, the output of the model is unnormalized and then integrated. We denote by a'_t the target normalized acceleration.

When L is large, backpropagating through $\mathcal{L}(x_{0:L}, \hat{x}_{0:L}) = \sum_{i=1}^{L}(x_i - \hat{S}^i(x_0))^2$ is difficult, as this requires all intermediate computations of the forward pass. A simple alternative is to use the **one-step** mean squared error $\frac{1}{L}\sum_t(a'_t - \text{GNS}(x_{t-n:t}))^2$. This loss is easy to backpropagate through, but it is unstable because it is blind to the full trajectory. The one-step loss can decrease while the trajectory loss increases, as the model has no feedback signal for the full trajectory. Nonetheless, we use it as a baseline and compare it against three strategies proposed in the literature: adding noise to the input data during training, unrolling for M steps without gradients, and using those steps as the input for the next N steps, for which we do have gradients, and the pushforward-trick. For ease of comparison, all the losses are given compactly in Table 1 in the appendix.

Random walk noise [7]. Add noise to the input of the model to simulate sampling from the noisy distribution produced by the model itself, as opposed to sampling from the distribution of the ground truth, as during the unroll the model will receive noisy inputs. That is, $\mathcal{L}(x_{0:L}, \hat{x}_{0:L}) = \frac{1}{L}\sum_t(a'_t - \text{GNS}(\tilde{x}_{t-n:t}))^2$, with $\tilde{x}_{t-n:t}$ is a perturbed version of $x_{t-n:t}$. The perturbation is obtained by drawing normal noise from $\mathcal{N}(0, \sigma)$ for each timestep, accumulating it as a random walk and perturbing the velocities with it. Then the perturbed velocities are used to compute the perturbed positions.

Preprocessing loss [11, 12]. Unroll the model for M steps and then use those steps as input to generate the next N steps. Compute the loss on those N steps and backpropagate. It is obtained by unrolling the model for M steps to get $\hat{x}_{t+M} = \hat{S}^M(x_t)$, and then computing $\mathcal{L}(x_{0:L}, \hat{x}_{0:L}) = \frac{1}{L}\sum_t \ell(x_{t+M:t+M+N}, \hat{x}_{t+M:t+M+N})$ where $\ell(x_{t+M:t+M+N}, \hat{x}_{t+M:t+M+N}) = \sum_{i=0}^{N}(a'_{t+M+i} - \text{GNS}(\hat{x}_{t+M+i-n:t+M+i}))^2$. This trick is meant to teach the model two things: how to do long unrolls, hence the sequence of length N, and how to deal with noise, hence the pre-unroll of length M. This strategy is similar to the random walk approach in that the goal is to teach the model how to deal with the noisy data generated by itself, the difference is that instead of sampling normal noise and adding it to the input, the noise is the real noise induced by the model. When $M = 0$ and $N = 1$ it reduces to the one-step loss.

Pushforward trick [9]. Sum the one-step loss and preprocessing loss with $M = 1$ and $N = 1$. $\mathcal{L}(x_{0:L}, \hat{x}_{0:L}) = \frac{1}{L}\left[\sum_t(a'_t - \text{GNS}(x_{t-n:t})^2 + (a'_{t+1} - \text{GNS}(\text{StopGrad}(\hat{x}_{t-n+1:t+1})))^2\right]$. StopGrad is the identity function in the forward pass but stops the gradients from flowing through its argument in the backward pass. As in the preprocessing loss, the input to the second term in the equation is perturbed by the noise induced by the model itself.

3 Benchmarks

We tested those tricks on three different datasets: WaterRamps (Fig. 1), an SPH dataset adapted from [7], WaterSmall, another SPH dataset generated using [13], and Flock, a dataset mimicking the flocking behavior of a group of birds (Fig. 3) adapted from the JaxMD example gallery [14]. For

WaterRamps we used the statistics provided by Sanchez-Gonzalez et al. [7] and for the other two datasets we compute them by accumulating the sum and sum of squares of the particle velocities and accelerations, and the total number of particles. We modified WaterRamps by removing some of the particles from the outer part of the walls, reducing their number from about 6k to about 2k. WaterSmall is similar to WaterRamps, but has fewer particles, both in the wall and the water, and was generated by a different simulator, leading to slightly different properties. Flock has periodic boundaries and 200 birds. It models flocking behavior following [2], where each bird follows simple local rules, such as flying in the average direction the neighboring birds are flying, and flying towards the center of mass of the neighborhood. Details of the GNS architecture can be found in the appendix.

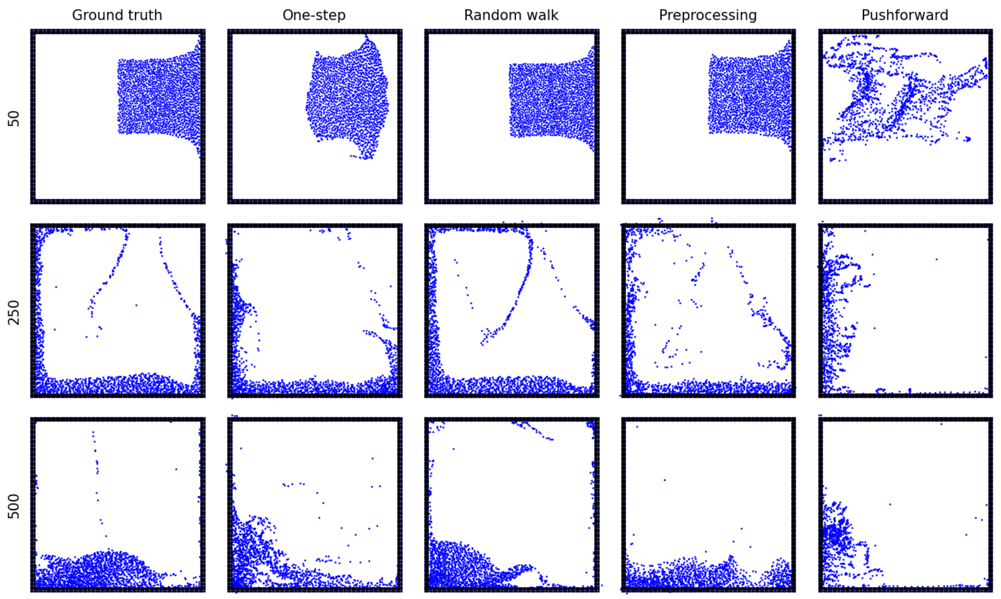

Figure 1: Qualitative comparison of WaterRamps unrolls for different stability tricks. Random walk performs the best, as that approach was developed using this dataset. We believe that by fine-tuning the architecture and hyperparameters, the preprocessing trick can be made to compete.

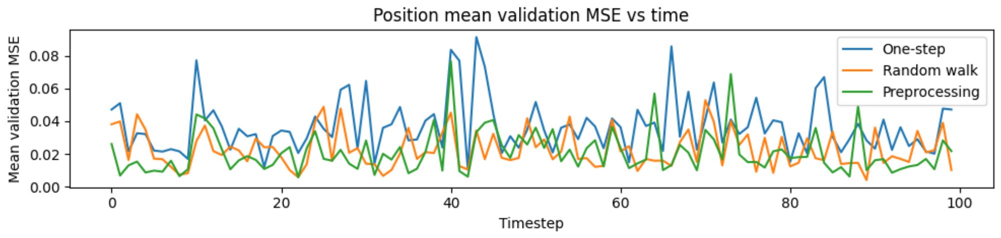

Figure 2: MSE per timesteps averaged over all validation simulations. While the error is similar for all tricks, the qualitative quality of the trajectories is not. Pushforward is omitted as it did not converge.

4 Discussion

While all the methods perform relatively well on WaterRamps (Fig. 1) and WaterSmall (in the appendix 5), it can be surmised from Fig. 3 that all tricks fail to improve the long-term stability of the unrolls in Flocks. This might seem surprising as one might think that WaterRamps is more complex than Flock, however, the latter has a key difficulty. All simulations reach an equilibrium towards the end, but while the equilibrium in the water simulations has no movements, with all the particles resting at the bottom, equilibrium in Flock is achieved in the distributional sense. The birds still move, but their distribution remains constant. The failure of all the tested approaches to deal with this appears to confirm that the problem of making GNS stable remains open. Further, rare events such as particle collisions or unlikely states are hard to learn. In some cases, such as in Flock, these rare events are pivotal to the distribution shift that happens as the system moves toward equilibrium.

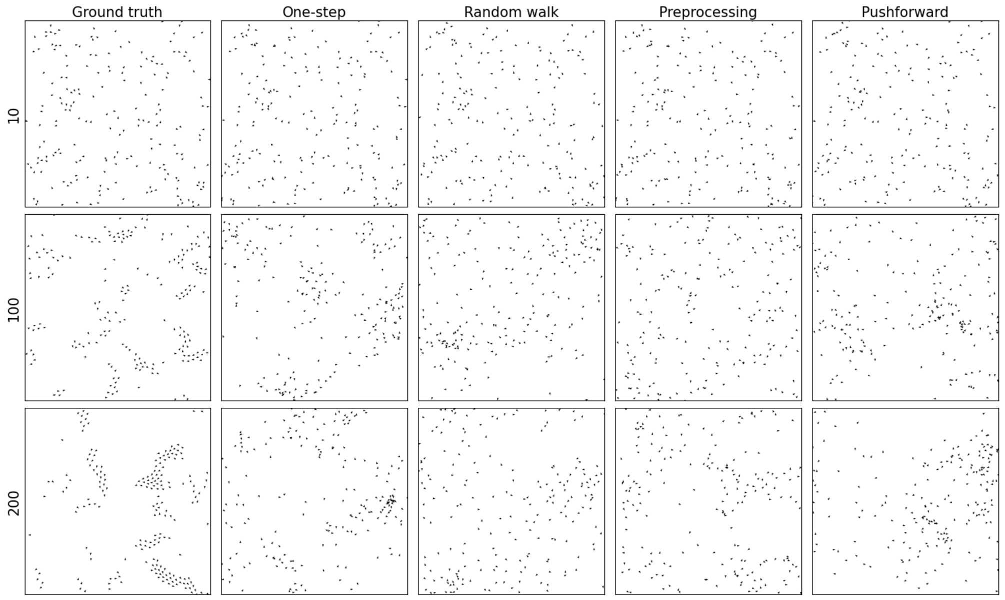

Figure 3: Qualitative comparison of Flock unrolls for different stability tricks. Each bird is described by its position x, y and heading angle θ, i.e. a 3D object. The initial state is sampled randomly and as time advanced structure starts to emerge. All the tricks tested failed to capture this phenomenon.

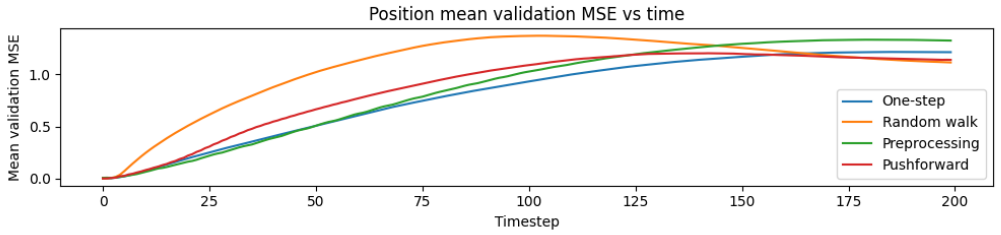

Figure 4: MSE per timesteps averaged over all validation simulations. Due to the periodic boundaries there is a maximum error. The error is very low at the start, but as collisions (rare events) occur they change the dynamics and shift the input distribution. The models' failure to capture those events leads to a rapid increase in error.

Thus, failure to accurately model these events exacerbates the inherent instability of the unroll and may demand specialized attention.

We believe one reason for this is that the model is trained on one task and then tested on a different but similar task. The model is expected to approximate the (autoregressive) solution operator that maps an initial state x_0 to a trajectory $\hat{x}_{0:L}$ such that some distance measure $d(x_{0:L}, \hat{x}_{0:L})$ is minimized. The training goal is, in contrast, defined individually for pairs $(x_t, \hat{x}_t)$ so it does not take into account the whole trajectory but individual transitions instead. Further, we ask the model to learn things that are not strictly included in the loss, such as physical plausibility. The functional form of the loss might also be inadequate, as the MSE loss assumes Gaussian output errors. Consider the case of a particle moving next to a wall. A Gaussian distribution can send that particle into the wall, which is unphysical. We also note a much-overlooked issue, training time. It is surprising to have days of training time when the actual simulation takes minutes per simulation and the dynamics can be written analytically.

In conclusion, we find that existing approaches to ensure unroll stability in GNS fail to address distribution shift and rare events. Stronger learning strategies are required to address these issues. Learning objectives that are closer to what we naturally expect from learned simulators are needed. Moreover, stronger inductive biases could be introduced into the model architecture to adapt it to the different nature of this problem and make learning faster.

5 Acknowledgements

This work was partially supported by ServicePublic de Wallonie Recherche under grant n° 2010235 – ARIAC by DIGITALWALLONIA4.AI.

References

[1] Donald G. Truhlar. “Ab Initio Molecular Dynamics: Basic Theory and Advanced Methods”. In: *Physics Today* 63.3 (Mar. 2010), pp. 54–56. ISSN: 0031-9228.

[2] Craig W. Reynolds. “Flocks, Herds and Schools: A Distributed Behavioral Model”. In: *SIGGRAPH Comput. Graph.* 21.4 (1987), pp. 25–34. ISSN: 0097-8930.

[3] Wouter van Toll et al. “SPH crowds: Agent-based crowd simulation up to extreme densities using fluid dynamics”. In: *Computers Graphics* 98 (2021), pp. 306–321. ISSN: 0097-8493.

[4] Daniel J. Price. “Smoothed particle hydrodynamics and magnetohydrodynamics”. In: *Journal of Computational Physics* 231.3 (2012). Special Issue: Computational Plasma Physics, pp. 759–794.

[5] R. A. Gingold and J. J. Monaghan. “Smoothed particle hydrodynamics: theory and application to non-spherical stars”. In: *Monthly Notices of the Royal Astronomical Society* 181.3 (Dec. 1977), pp. 375–389. ISSN: 0035-8711.

[6] J. A. Kegerreis et al. “Immediate Origin of the Moon as a Post-impact Satellite”. In: *The Astrophysical Journal Letters* 937.2 (2022), p. L40.

[7] Alvaro Sanchez-Gonzalez et al. *Learning to Simulate Complex Physics with Graph Networks*. 2020. arXiv: `2002.09405 [cs.LG]`.

[8] Sungyong Seo and Yan Liu. *Differentiable Physics-informed Graph Networks*. 2019.

[9] Johannes Brandstetter, Daniel Worrall, and Max Welling. *Message Passing Neural PDE Solvers*. 2023. arXiv: `2202.03376 [cs.LG]`.

[10] Remi Lam et al. *GraphCast: Learning skillful medium-range global weather forecasting*. 2023. arXiv: `2212.12794 [cs.LG]`.

[11] Benjamin Ummenhofer et al. *Lagrangian Fluid Simulation with Continuous Convolutions*. 2022.

[12] Lukas Prantl et al. “Guaranteed Conservation of Momentum for Learning Particle-based Fluid Dynamics”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 6901–6913.

[13] Jan Bender et al. *SPlisHSPlasH Library*. URL: `https://github.com/InteractiveComputerGraphics/SPlisHSPlasH`.

[14] Samuel S. Schoenholz and Ekin D. Cubuk. “JAX M.D. A Framework for Differentiable Physics”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020.

[15] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: `http://github.com/google/jax`.

[16] Jonathan Heek et al. *Flax: A neural network library and ecosystem for JAX*. Version 0.7.4. 2023. URL: `http://github.com/google/flax`.

[17] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: `1711.05101 [cs.LG]`.

A Appendix

Implementation details. For the GNS, we used a Jax [15] + Flax [16] reimplementation of the encode-process-decode architecture from [7], with minor differences. Instead of using a KDTree and a connectivity radius, the graph is built using nearest neighbours. We used the 10 nearest neighbors, as we found that the number of particles within the interaction radius in WaterRamps rarely exceeds 10. The node attributes are the history of velocities (computed as $x_t - x_{t-1}$) and, in the case of WaterRamps and WaterSmall, the distance to the boundaries normalized by the connectivity radius, with distances greater than the radius clipped to $\{1, -1\}$. The edge attributes are the relative vector displacement from a particle to its neighbor and the norm of said vector, both normalized and clipped as before. We used 10 message passing steps, multilayer perceptrons (MLPs) with two layers and 128 units for the encoder, decoder and processor, and latent size of 128. To conserve memory and accelerate training, we used only one processor network for all the message passing steps. We tested that against using 10 different processors and found the speedup worth the slight loss in accuracy. For WaterRamps and WaterSmall we used 0.0015 as the connectivity radius, while in Flocks it was 0.005. The noise parameter for the random walk was $\sigma = 6.7 \times 10^{-4}$ for all tests.

Learning rate and batch size. We used the AdamW [17] optimizer with a linear learning schedule, starting at a constant 10^{-4} for WaterRamps and WaterSmall, and 10^{-5} for Flock for 100k steps, then reducing linearly to 10^{-6} over the next 100k steps, and then constant 10^{-6}. Depending on the dataset we used different batch sizes. For WaterRamps and WaterSmall we used a batch size of 4, except for the preprocessing strategy where we used a batch size of 2. For Flock, we used a batch size of 8 in general, and 2 for the preprocessing strategy. We trained with two GPUs, using data parallelism and fitting half the batch in each GPU. We used either two RTX A5000 or two Quadro RTX 6000, both with 24GB of VRAM.

Losses. We write all the losses here for ease of comparison:

Table 1: Compact comparison of the different losses $\mathcal{L}(x_{0:L}, \hat{x}_{0:L})$

One-step loss	$\frac{1}{L}\sum_t (a'_t - \text{GNS}(x_{t-n:t}))^2$
Random walk loss	$\frac{1}{L}\sum_t (a'_t - \text{GNS}(\tilde{x}_{t-n:t}))^2$
Preprocessing loss	$\frac{1}{L}\sum_t \ell(x_{t+M:t+M+N}, \hat{x}_{t+M:t+M+N})$
Pushforward loss	$\frac{1}{L}\left[\sum_t (a'_t - \text{GNS}(x_{t-n:t})^2 + (a'_{t+1} - \text{GNS}(\text{StopGrad}(\hat{x}_{t-n+1:t+1})))^2\right]$

WaterSmall. We present the results for WaterSmall in Fig. 5

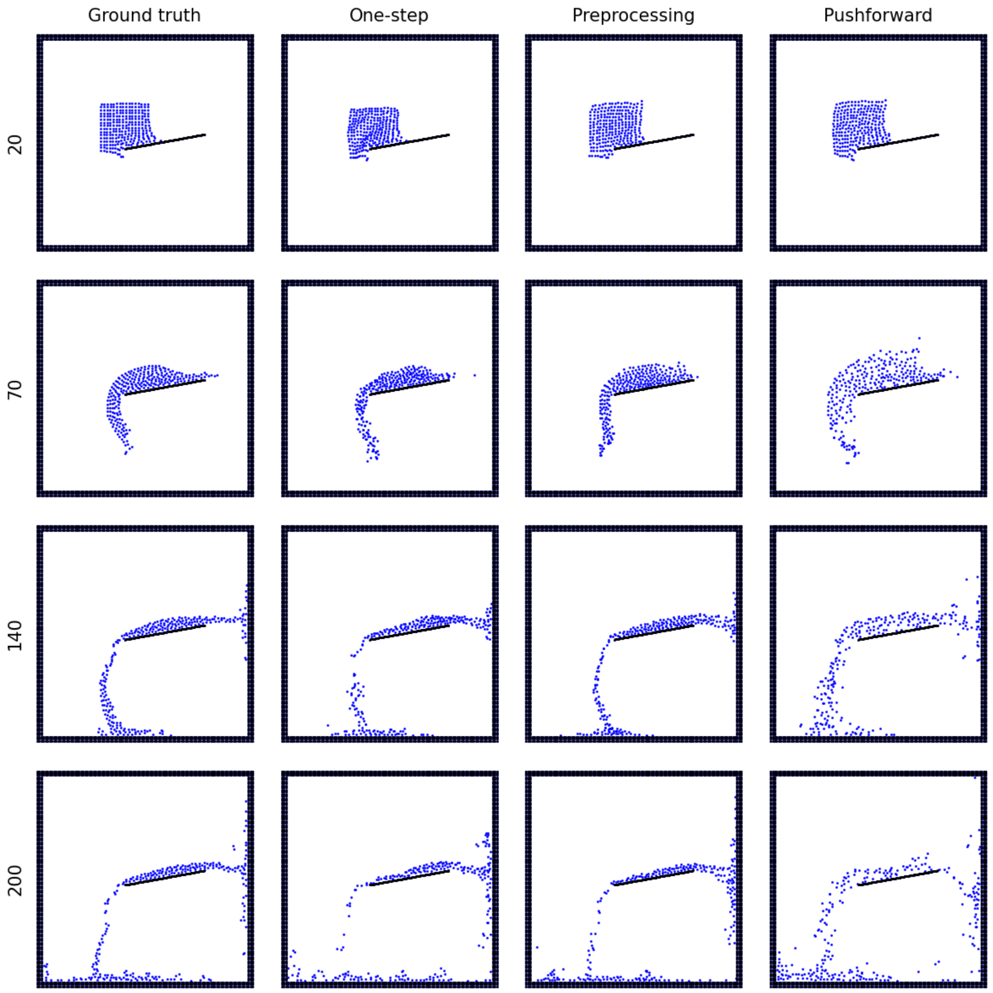

Figure 5: Qualitative comparison of WaterSmall unrolls for different stability tricks. We removed the random walk trick as it did not converge because we did not tune the noise parameter. The need to do that is a strong drawback of that method. The initial state of this simulator is very structured, with the particles arranged on a grid. It can be seen that the models struggle to maintain that structure.

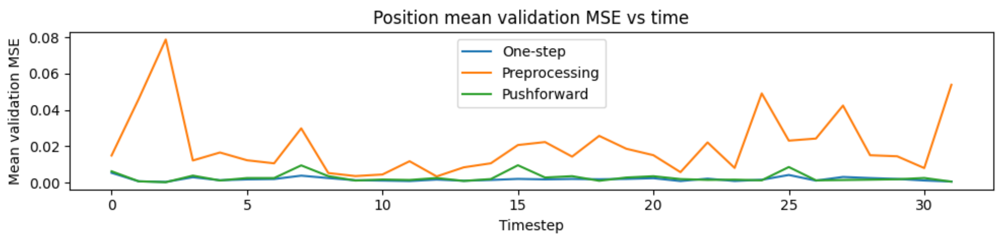

Figure 6: MSE per timesteps averaged over all validation simulations. While the preprocessing trick has the highest error, its trajectories are more stable than Pushforward, which has a lower error, and comparable to the baseline, which has the lowest error.