

---

# Constraint-based Graph Network Simulator

---

Yulia Rubanova<sup>\*1</sup> Alvaro Sanchez-Gonzalez<sup>\*1</sup> Tobias Pfaff<sup>1</sup> Peter Battaglia<sup>1</sup>

## Abstract

In the area of physical simulations, nearly all neural-network-based methods directly predict future states from the input states. However, many traditional simulation engines instead model the constraints of the system and select the state which satisfies them. Here we present a framework for constraint-based learned simulation, where a scalar constraint function is implemented as a graph neural network, and future predictions are computed by solving the optimization problem defined by the learned constraint. Our model achieves comparable or better accuracy to top learned simulators on a variety of challenging physical domains, and offers several unique advantages. We can improve the simulation accuracy on a larger system by applying more solver iterations at test time. We also can incorporate novel hand-designed constraints at test time and simulate new dynamics which were not present in the training data. Our constraint-based framework shows how key techniques from traditional simulation and numerical methods can be leveraged as inductive biases in machine learning simulators.

## 1. Introduction

Consider a bowling ball colliding with a bowling pin. You might explain this event through a pair of forces being generated: one causes the pin to move, the other one causes the ball to careen away. This approach is analogous to physical simulators that apply an explicit forward model to calculate a future state directly from the current one, i.e. by numerically integrating equations of motion. An alternative, but equally valid, way to explain the collision is in terms of constraint satisfaction: the ball and pin cannot occupy the same location at the same time, and their combined energies and momenta must be conserved. The post-collision

---

<sup>\*</sup>Equal contribution <sup>1</sup>DeepMind, London, UK. Correspondence to: Yulia Rubanova <rubanova@deepmind.com>, Alvaro Sanchez-Gonzalez <alvarosg@deepmind.com>.

trajectories are the only way the future can unfold without violating these constraints. This approach is analogous to the constraint-based simulators that generate a prediction by searching for a future state that respects all the constraints.

Both families of simulators—those based on explicit, forward functions versus those which define the dynamics implicitly, via constraints—are widely used in physics, engineering, and graphics. In principle they can model the same types of dynamics. In practice these simulators strike different trade-offs that determine which one is preferred in different domains. Explicit methods are popular for large systems with (mostly) independent local effects where space and time derivatives are relatively smooth. By contrast, implicit approaches are often preferred for systems with strong interactions, such rigid and stiff dynamics, and more accurate solutions can often be found by using more solver iterations or more sophisticated solvers. In machine learning, so far almost all methods for learned simulation have focused on explicit forward models (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021), with few exceptions (Yang et al., 2020).

Here we present a framework for learning to simulate complex dynamics via constraint satisfaction. Our “Constraint-based Graph Network Simulator” (C-GNS) defines a single learned constraint function that indicates whether a future state is consistent with the current and previous states. Conditioning on the previous states allows our method to capture the time dynamics within its learned constraint function. We implement the constraint function as a Graph Neural Network (GNN, Bronstein et al. (2017); Battaglia et al. (2018)), which can model systems with rich compositional structure, such as multiple bodies and complex meshes. To predict the future state, we use a gradient-based solver that iteratively refines a proposed state to minimize the learned constraint. We train the model end-to-end by backpropagating the loss gradients through the solver. Crucially, our model is trained directly on observed trajectory data and does not require knowledge of the true underlying constraints which govern the system dynamics. The learned constraint function only needs to yield the same solution as the true constraints, while their objective landscapes may differ (e.g., the learned constraint function may be convex, as in Figure 1(c), while the true constraint may not be).

There are several potential advantages to constraint-based

learned simulation. First, the constraint function is decoupled from procedure for satisfying it. The user can choose different solvers or invest different amounts of computation to improve the solution. Another potential advantage is that the constraint function’s semantics allows to incorporate additional constraints into the overall constraint function. Finally, we speculate that neural-network-based constraint functions may be simpler than the neural networks that directly predict the solution that satisfy all the physical constraints. By rough analogy, for NP-complete problems, evaluating solutions is (thought to be) cheaper than finding solutions.

We tested C-GNS on several physical simulation domains: ropes, bouncing balls and irregular rigids (MuJoCo engine, Todorov et al. (2012)) and splashing fluids (Flex engine, Macklin et al. (2014a)). We found that C-GNS produced more accurate rollouts than the state-of-the-art Graph Net Simulator (Sanchez-Gonzalez et al., 2020) with a comparable number of parameters, and than Neural Projections (Yang et al., 2020).

We demonstrate several unique features of our model. First, the constraint function is decoupled from the procedure for satisfying it. The user can choose different solvers or invest different amounts of computation to improve the solution. We show that C-GNS can use additional solver iterations at test time to improve its predictive accuracy, striking desired speed-accuracy trade-offs. Second, our model allows to incorporate new, hand-designed constraints at test time and satisfy them jointly alongside its learned constraints. These properties have not been reported previously for explicit forward models or Neural Projections by Yang et al. (2020).

## 2. Background and Related Work

Constraint solvers are central to many physics simulators. Most rigid-body and game engines use constraints to model joints, collision and contact (Baraff, 1994). Position-based (Müller et al., 2007) and projective dynamics (Bouaziz et al., 2014) are popular methods that express the time dynamics purely as constraints and can simulate a wide range of physical systems such as rigids, soft-bodies, fluids and cloth (Macklin et al., 2014a; Thomaszewski et al., 2009).

There is a rapid growth of machine learning methods for accelerating scientific simulation of complex systems, such as turbulence (Stachenfeld et al., 2021; Kochkov et al., 2021) and aerodynamics (Thuerey et al., 2020; Zhang et al., 2018). Particularly, a learned simulator based on graph neural networks is a popular approach for modelling a wide range of systems, from articulated dynamics (Sanchez-Gonzalez et al., 2018) to particle-based physics (Mrowca et al., 2018; Li et al., 2019; Sanchez-Gonzalez et al., 2020) and mesh-based continuum systems (Pfaff et al., 2021; De Avila

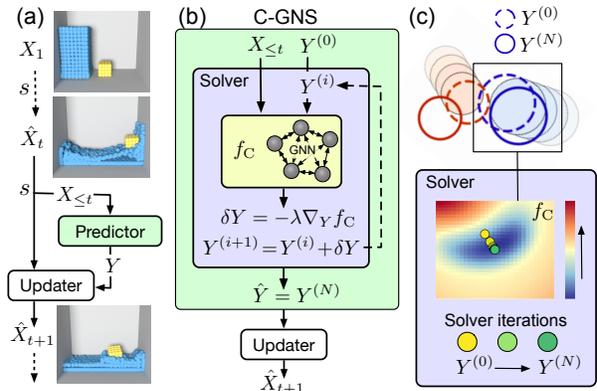


Figure 1. (a) **Learned simulator schematics.** A simulator  $s$  maps  $X_{\leq t}$  to a future state  $\hat{X}_{t+1}$ . The PREDICTOR takes  $X_{\leq t}$  and returns  $\hat{Y}$  which represents information about the system’s temporal evolution. An UPDATER uses  $\hat{Y}$  to update  $X_t$  to  $\hat{X}_{t+1}$ . (b) **Constraint-based Graph Network simulator (C-GNS).** The PREDICTOR iteratively solves for a  $\hat{Y}$  to satisfy a constraint function  $f_C$  using  $\nabla_Y f_C$ . (c) **Constraint optimization on two colliding balls.** The heatmap color shows the value of learned constraint  $f_C$  as we vary the update  $Y$  for the blue ball. The colored points on the heatmap show the iterations of the solver as it minimizes the constraint  $f_C$  (from  $Y^{(0)}$  to  $Y^{(N)}$ ), indicating that the blue ball should bounce downwards to resolve the collision. The learned  $f_C$  has a “funnel” shape around the correct next state of the ball.

Belbute-Peres et al., 2020). Combining learning algorithms with principles from physics and numerical methods can improve sample complexity, computational efficiency, and generalization (Wu et al., 2018; Karniadakis et al., 2021; Chen et al., 2018; Rubanova et al., 2019). Imposing Hamiltonian (Greydanus et al., 2019; Sanchez-Gonzalez et al., 2019; Chen et al., 2019) and Lagrangian (Lutter et al., 2019; Cranmer et al., 2020; Finzi et al., 2020) mechanics in learned simulators offers unique speed/accuracy tradeoffs and can preserve symmetries more effectively. Zhong et al. (2021) further introduce an explicit module to handle contacts inside a Lagrangian or Hamiltonian system. Our work aims to learn these inductive biases from data. Moreover, our model can be potentially combined with the inductive biases from some of these ideas to learn the remaining constraints.

Outside of the scope of physical simulations, recent methods were proposed for learning implicit functions and solving them in the forward pass (see “Deep Implicit Layers” survey by Duvenaud et al. (2020)). These methods also use a solver to produce a prediction from the model. The implicit models can play games (Amos & Kolter, 2017; Wang et al., 2019), optimize power flow (Donti et al., 2021), support robotic planning (Loula et al., 2020), and perform combinatorial optimization (Bartunov et al., 2020). A notable example of the implicit models are Deep Equilibrium Models (DEQs) (Bai et al., 2019; 2020) that look for a zero-point

of the learned function. DEQs use the implicit differentiation technique to avoid backpropagation through the solver and reduce the cost of computing the gradients. DEQs were not designed for physical simulations and search for the equilibrium (zero) point *in the latent space*. By contrast, our model searches for the equilibrium (minimum) over the physical state directly, allowing us to a) combine the learned function with other physical constraints and b) relate the solver iterations to improving the simulation accuracy.

Despite the popularity of traditional constraint-based simulators, only a single work that projects positional variables on a learned constraint manifold has been reported (Yang et al., 2020). See Section 4.4 for a detailed comparison between our model and Yang et al. (2020).

### 3. Model Framework

#### 3.1. Simulation Basics

A physical trajectory, measured at discrete time intervals, is a sequence of states,  $(X_1, \dots, X_T)$ , where  $X_t$  may contain properties of elements of the system such as the positions, instantaneous velocities, masses, etc. A physical simulator  $s$  is a function that maps current and/or previous state(s), which we term the *context*  $X_{\leq t}$ <sup>1</sup>, to a predicted future state  $\hat{X}_{t+1} = s(X_{\leq t})$ <sup>2</sup> (see Figure 1a). A simulated physical trajectory termed a *rollout*  $(X_t, \hat{X}_{t+1}, \hat{X}_{t+2}, \dots)$ , can be generated by repeatedly applying  $s$  to its own predicted state,  $\hat{X}_{t+1} = s(X_{\leq t})$ .

Simulators are often comprised of a PREDICTOR and an UPDATER mechanism. The PREDICTOR maps the context  $X_{\leq t}$  to an update value  $\hat{Y}$  that represents information about the system’s temporal evolution at the current time (e.g., new positions, velocities or accelerations). Then the UPDATER mechanism uses  $\hat{Y}$  to update the current state to the next state:  $\hat{X}_{t+1} = \text{UPDATER}(X_{\leq t}, \hat{Y})$ , e.g. updating current positions and velocities represented by  $X_t$  with new velocities and accelerations represented by  $\hat{Y}$ .

#### 3.2. Explicit Simulators

Across science, engineering, and graphics, a popular class of simulators (Todorov et al., 2012; Monaghan, 2005; Mir-tich & Canny, 1995; Witkin et al., 1990) are defined *explicitly*: the state update  $\hat{Y}$  is predicted directly from  $X_{\leq t}$  using an explicit forward function  $\hat{Y} = f_D(X_{\leq t})$ . Among

<sup>1</sup>Despite that physics is Markovian, we use  $X_{\leq t}$  as input because our framework can also apply to non-Markovian dynamic processes. Providing previous states can also be helpful when there are hidden properties of the system which are only identifiable over a sequence of observed states, for example when a state does not contain instantaneous velocities, such as in our environments.

<sup>2</sup>We loosely use the hat notation (e.g.  $\hat{X}$ ) for the quantities that are predicted by the model.

the learned simulators, the forward function  $f_D$  is typically implemented using a graph neural network (GNN) that allows simulators to scale well to large graphs of 1000s of nodes and support generalization to systems with different shapes and sizes (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021; Battaglia et al., 2016). We call the explicit GNN-based model **Forward GNN**.

#### 3.3. Constraint-Based Implicit Simulators

In this paper we explore the learned simulators based on *implicit* formulations of the dynamics. Instead of predicting the desired state directly, our implicit simulator uses a differentiable constraint function  $c = f_C(X_{\leq t}, Y)$ , where  $c$  is a scalar that quantifies how well a proposed state update  $Y$  agrees with  $X_{\leq t}$ . A future prediction is generated in two stages: 1) apply a solver (gradient descent or a zero-finding algorithm) to find a  $\hat{Y}$  that satisfies the constraint function, and 2) use the value  $\hat{Y}$  in the UPDATER to update  $X_t$  to  $\hat{X}_{t+1}$ . Our constraint function  $f_C$  is defined as a trainable neural network with a non-negative scalar output. It represents an approximation for all the physical constraints in the system, including the time dynamics.

As illustrated in Figure 1(b), we formulate our constraint-solving procedure via an iterative method that starts with an initial proposal  $Y^{(0)}$ . On the  $i$ -th iteration, the solver uses the gradient of  $f_C$  w.r.t.  $Y$  to compute a change to the proposal,  $\delta Y = -\lambda \nabla_Y f_C(X_{\leq t}, Y)|_{Y=Y^{(i)}}$ . Then,  $\delta Y$  is used to revise the proposal:  $Y^{(i+1)} = Y^{(i)} + \delta Y$ . This process repeats for  $N$  steps, and the final proposal value is considered to be the PREDICTOR’s output  $\hat{Y} = Y^{(N)}$ .

We define the solution as the minimum of the constraint function  $\hat{Y} = \text{argmin}_Y f_C(X_{\leq t}, Y)$ . We use gradient descent with the fixed step size  $\lambda$  to find the solution  $\hat{Y}$ . We refer to our Constraint-based Graph Network Simulator with gradient descent solver as **C-GNS-GD**.

During training we use a fixed number of GD iterations (5). At test time, the computational budget of the forward pass can be varied via the number of solver iterations  $N$ , as we further explore in Section 5.4.

This general formulation of constraint-based learned simulation can be trained by backpropagating loss gradients through the solver iterations<sup>3</sup>. Specifically, one step of gradient descent (GD) is  $Y^{(i+1)} = Y^{(i)} - \lambda \nabla_Y f_C(X_{\leq X}, Y)$ , which is a differentiable operation itself. Consider  $\theta$  to be the network parameters of  $f_C$ . Backpropagation through one GD iteration is simply  $\frac{d}{d\theta} Y^{(i+1)} = \frac{d}{d\theta} Y^{(i)} - \lambda \frac{d}{d\theta} \nabla_Y f_C(X_{\leq X}, Y)$

<sup>3</sup>Implicit differentiation at the solution point, mentioned in Section 2, is applicable as well but we did not explore this direction.

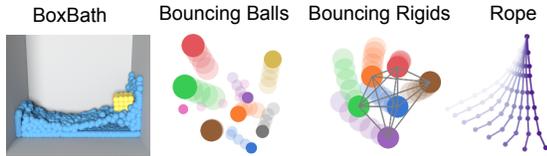


Figure 2. Renderings of the physical environments.

Videos of the model rollouts are available at:

[sites.google.com/view/constraint-based-simulator](https://sites.google.com/view/constraint-based-simulator).

### 3.4. Explicit Iterative Simulators

As a hybrid between forward and constraint-based simulators, we also introduce **Iterative GNN** model. Similarly to C-GNS-GD, this model iteratively refines the proposed state update, but at each iteration  $\delta Y$  is predicted directly by a graph neural network  $\delta Y = f_{DI}(X_{\leq t}, \hat{Y})$ , instead of being computed through a gradient. We use this hybrid model to separately study the effect of pure iterations versus iterative constraint-based optimization (Section 5.4 and Figure B.7).

## 4. Experiments

### 4.1. Experimental Task Domains

We tested our approach on a variety of physical environments, shown in Figure 2. We generated the data for our ROPE, BOUNCING BALLS and BOUNCING RIGIDS datasets using the MuJoCo physics simulator (Todorov et al., 2012). We also tested our model on BOXBATH dataset with 1024 particles from (Li et al., 2019) to explore the scaling capabilities of the model. These environments involve a diverse set of physical constraints: ‘hard’ constraints on preserving the rigid shapes and resolving collisions, and ‘soft’ constraints on fluid movement, handling gravity and preserving the momentum. The simulations consist of 150 time steps for BOXBATH and 160 time steps for other datasets.

**Representing the physical system** Our experimental domains consist of interacting point-like elements: sized objects, fluid particles or mesh vertices. The datasets contain the positions of each element:  $P_t = (p_t^j)_{j=1\dots J}$ , where  $J$  is the number of elements, and  $p_t^j$  is the  $j$ -th element’s position at time  $t$ . Note that our datasets do not contain the instantaneous velocities. Instead, the velocity information can be estimated by changes in the position, as described below. Additionally, we represent the static properties of the physical elements (masses, material types, etc.) as  $Z$  to keep it distinct from the dynamic state information.

### 4.2. Implementation of the C-GNS Model

In the implementation of the C-GNS, we follow the graph network architecture and input graph construction from GNS (Sanchez-Gonzalez et al., 2020), as described below.

GNS was designed specifically to learn physical simulations with local interactions that do not depend on the absolute particle position. In comparison to GNS, we use the graph network inside the compute constraint function over the next state, rather than to directly predict of the next state. Specifically, we (1) replace the GNS decoder to output a scalar value instead of a graph (2) use gradient descent solver to find  $Y$  that minimises the scalar output. (3) add the future state  $Y$  to the network input.

In our implementation, the state  $X_t$  consists of the positions  $P_t$  and the static information  $Z$ . The input context is a sequence of the most recent positions and the static properties:  $X_{\leq t} := (Z, P_{t-3}, P_{t-2}, P_{t-1}, P_t)$ .

To represent the dynamics, we set the update  $Y$  to be the change in position over time  $V_{t+1} = (v_{t+1}^j)_{j=1\dots J} \equiv (p_{t+1}^j - p_t^j)_{j=1\dots J}$ , which we informally call “velocity”, estimated as a backward difference. The update mechanism  $\hat{X}_{t+1} = \text{UPDATER}(X_{\leq t}, \hat{Y})$  simply becomes  $\hat{P}_{t+1} = P_t + \hat{V}_{t+1}$ , where  $\hat{V}_{t+1}$  is the output of a PREDICTOR. For BOXBATH, we set the update  $Y$  to the acceleration, for the sake of consistency with Sanchez-Gonzalez et al. (2020) (see details in Supplementary Section A.2.3).

**Constructing the GNN Input Graph** We represent the context  $X_{\leq t}$  and a proposed update  $Y^{(i)}$  as a graph where the nodes correspond to different elements, such as objects or particles, and the edges correspond to the possible pairwise interactions between them. The function  $f_C$  takes the input graph containing  $X_{\leq t}$  and  $Y^{(i)}$  and outputs a scalar value  $c$ .

To construct the graph features from the context  $X_{\leq t}$ , we enforce translation-invariance and do not explicitly provide absolute positions  $P_t$  as the input to the network. The features for the node  $j$ ,  $[z^j, v_{t-2}^j, v_{t-1}^j, v_t^j]$ , include a sequence of three most recent position changes (i.e. velocities)  $v_t^j = p_t^j - p_{t-1}^j$  and static properties  $z^j$ . To construct the edge feature from nodes  $j$  to  $k$ , we provide the relative displacement vector between the nodes’ positions,  $e_t^{jk} = p_t^k - p_t^j$ . Finally, to represent  $Y^{(i)}$ , we concatenate the proposed update for node  $j$  from the  $i$ -th solver iteration  $y^{j,(i)}$  (velocity or acceleration) to the node features.

**GNN-based Constraint Function** We implement the function  $f_C$  as a graph network (GNN). First, we encode nodes and edges of the input graph using MLPs. Then, we process the graph with a GNN consisting of a sequence of message-passing (MP) layers. We use node and edge embeddings but we do not compute and update a global vector as part of the message passing process. Finally, we perform a pooling operation that summarizes the node representations of the graph into a single scalar value. To do so, we compute the scalar values  $c^j$  for each node by running an MLP decoder on the node outputs of the GNN. We square the values  $c^j$  to

make them non-negative. Finally, we obtain a single scalar constraint  $c$  for the entire graph by averaging the per-node values  $c = f_C(X_{\leq t}, \hat{Y}) = \frac{1}{J} \sum_{j=1}^J (c^j)^2$ .

**Optimizing the Constraint** We initialize  $Y^{(0)}$  to the most recent velocity  $V_t$ , as we expect it to be a good prior for the future velocity in inertial systems. In BOXBATH, where  $Y$  represents the acceleration, we initialize  $Y^{(0)}$  to a zero vector. In any case, the model has access to the history of states and can make arbitrary corrections to  $Y^{(0)}$ . We use auto-differentiation in JAX to compute the constraint gradient  $\nabla_Y f_C$ . For the gradient descent solver, we use a fixed step size  $\lambda = 0.001$ . We used  $N = 5$  iterations during training.

### 4.3. Training and Evaluation

We compute the  $L_2$  loss between the predicted update  $\hat{Y}$  from the last solver iteration and the corresponding ground-truth update, averaged over all nodes. Note that it is straightforward to compute the ground-truth update from the dataset. For example, if  $Y$  represents the velocity, the update is simply the difference between the future and current positions.

To further incentivize to convergence to the ground-truth, we experimented with an additional loss between each intermediate update  $Y^{(i)}$  and the ground-truth with exponential decay weights (details in Section A.3). We use the additional loss only in the generalization experiments in Section 5.4, labeled with  $\alpha = 0.25$ . For other experiments, the additional loss had little effect on the MSE error (Figure 7).

We train the model on one-step prediction task using standard backpropagation with the Adam optimizer. At test time, we evaluate 1-step and rollout errors between predicted and ground truth trajectories. The rollout is computed by iteratively applying the model on the previous predictions, starting from the initial time step sequence.

### 4.4. Neural Projections and Related Ablations

The only related work involving learned constraint-based simulation that we are aware of is **Neural Projections (NP, Yang et al. (2020))**. While it inspired this work, Neural Projections has key differences from our approach, and is fundamentally limited in ways that make it insufficient as a general-purpose learned simulator.

Neural Projections operates directly on the absolute positions of the particles. First, the model uses an Euler step to propose a future position of the particles based on the previous position, estimated velocity and known external forces. Then the model refines the proposal by iteratively projecting it onto a learned constraint manifold, implemented as a multilayer perceptron (MLP). In our framework, this would be equivalent to (1) setting the optimized update  $Y$  to be the

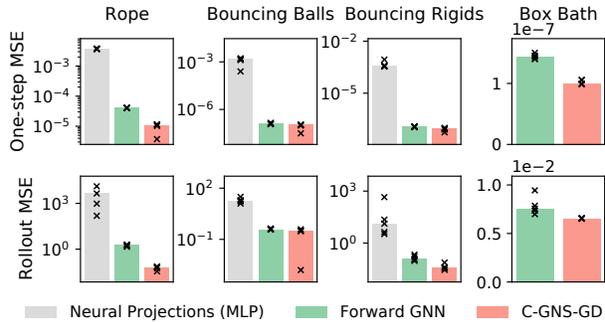


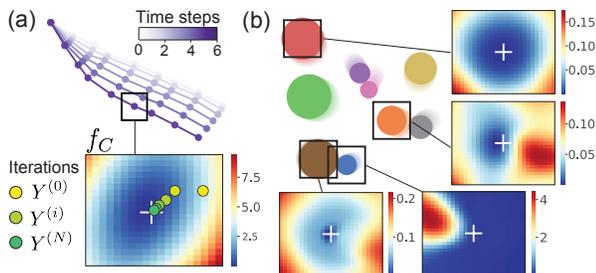
Figure 3. **Comparison to the existing baselines.** Top row: 1-step test MSE on node positions. Bottom row: full-rollout test MSE (160-step). The bar height represents the median MSEs over random seeds. The black crosses show the MSE metric for each random seed. We found Neural Projections could not effectively scale to BOXBATH (see Section 4.4). All the plots except BOXBATH are on log scale.

future positions of the system  $Y := P_{t+1}$  (the UPDATER becomes the identity function), (2) making the constraint function depend on the update only:  $f_C(Y)$  and (3) initializing  $Y^{(0)}$  to the output of the Euler step.

Crucially, the Neural Projections constraint function only measures how much the *current* set of proposed positions of the particles violates the learned constraints without context about past states  $X_{\leq t}$ . Thus, the model cannot correctly resolve scenarios such as the elastic collisions, as the constraint function does not have access to the dynamics (i.e. how the proposed state relates to the previous state, illustrated in Supplementary Figure A.1). This weakness renders Neural Projections insufficient for general-purpose physical simulations. Our model, on the contrary, does not have this issue, because the constraint function is always conditioned on the past context  $f_C(X_{\leq t}, Y)$ , which allows to model time dynamics as a part of the learned constraint. To study the effect of this difference, we provide an ablation to our model **C-GNS-GD- $f_C(Y)$**  that uses only the positional information of the future state and does not have access to the context  $X_{\leq t}$  (details in the Supplementary Section A.3)

Next, Neural Projections defines the constraint solution as  $f_C(Y) = 0$ , and uses the zero-finding “Fast Projection” algorithm (FP, Goldenthal et al. (2007)) to find a solution. In contrast, our model defines the solution as a minimization problem, i.e.,  $\hat{Y} = \operatorname{argmin}_Y f_C(X_{\leq t}, Y)$ , solved by a gradient descent. To explore these choices, we also tested an FP-based version of our model: **C-GNS-FP**.

Finally, Neural Projections uses an MLP as a constraint function that takes the concatenated features for all of the particles and outputs a constraint value. Compared to GNNs, MLP-based simulators have been shown to be sub-optimal to model particle systems (Battaglia et al., 2016; Sanchez-



**Figure 4. Visualization of the learned constraint.** The heatmaps show the values of the learned constraint  $f_C$  as a function of the update  $Y$  for one of the nodes, keeping other nodes fixed.

**(a) An example from Rope simulation.** The learned constraint function has a “funnel” shape, where the minimum coincides with the only valid next state of the simulation (ground-truth, shown as the white cross). The colored points represent the iterations of the gradient descent solver as it minimizes the learned constraint: from the initial  $Y^{(0)}$  (yellow) to final  $Y^{(5)}$  (green).

**(b) An example from Bouncing Balls.** The red ball is far from other balls, and its constraint  $f_C$  represents a smooth funnel centered around the ground-truth (white cross). For orange and blue balls, the constraint  $f_C$  has high values in the areas occupied by another ball, indicating that the current ball cannot overlap with it.

Gonzalez et al., 2018). Neural Projection paper (Yang et al., 2020) includes a heuristic parameter-sharing scheme to allow variable number of particles, but it requires manually grouping subsets of the state. It is not clear how this heuristic would scale to large systems with dynamically changing interactions. We also created ablated versions of our model that uses an MLP-based constraint function instead of a GNN: **C-MLP-GD** and **C-MLP-FP**.

## 5. Results

### 5.1. Comparison to Existing Baselines

Our experimental results show that the performance of C-GNS-GD is generally better than the existing baselines on the datasets we tested.<sup>4</sup> Figure 3 demonstrates that C-GNS-GD has the lowest 1-step and rollout MSE across all datasets, compared to Neural Projections (Yang et al., 2020) and Forward GNN (Pfaff et al., 2021) with a comparable number of parameters<sup>5</sup>. We provide the corresponding numerical results with error bars in the Supplementary Table B.1. Qualitatively, we observed that for Forward GNN, the box in BOXBATH “melts” over time, as the forward model cannot preserve its rigid shape (see Videos). By contrast, the comparable C-GNS-GD effectively maintains the rigid shape of

<sup>4</sup>Videos of the model rollouts are available at [sites.google.com/view/constraint-based-simulator](https://sites.google.com/view/constraint-based-simulator)

<sup>5</sup>To make Forward GNN comparable to C-GNS-GD, we use the same number of MP layers in both models (2 MP layers for ROPE, 1 MP for other datasets)

the cube. We further explore the comparison to a larger Forward GNN with up to 5x more parameters in Sections 5.4 and 5.7. These results suggest that constraint-based learned simulators are a competitive alternative to explicit forward simulators.

### 5.2. Interpreting the Learned Constraints

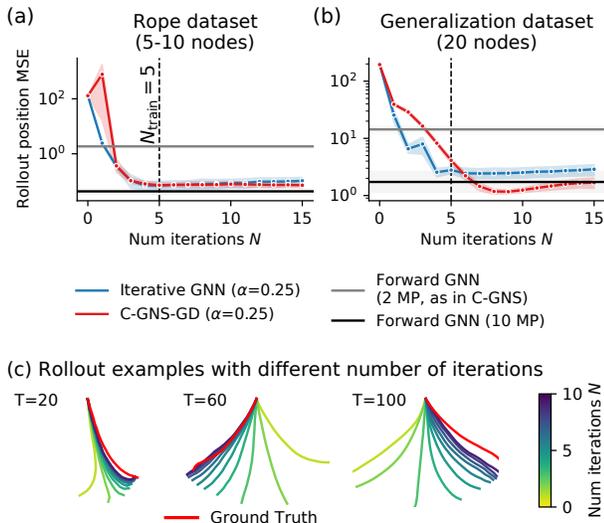
To better understand the learned constraint  $f_C$  in the C-GNS-GD, we visualized how the output of  $f_C$  changes as a function of  $Y$ . We varied the proposed update  $Y$  (velocity in 2D space) for a particular node while holding the updates  $Y$  for other nodes fixed. Figure 4(a) shows the learned constraint for a node from the ROPE dataset. The network learns a “funnel” shape of the constraint that is easy to minimize with gradient descent. The constraint has a single minimum that is near the ground-truth point (the white cross). It is expected, as there is only one valid next state of the system. The sequence of points represents the proposed updates  $Y^{(i)}$  from the solver, demonstrating that the solver reaches the ground-truth in five iterations, as expected. Note that we did not enforce the “funnel” shape of the constraint.

Figure 4(b) shows the learned constraint  $f_C$  for several nodes in BOUNCING BALLS. For the red ball, which is far from other balls, the constraint has a “funnel” shape, similarly to the ROPE example. For the balls that have another object nearby, the constraint value is high in the area occupied by that object, indicating that overlapping with another object would result in an invalid physical state.

### 5.3. Constraint Convergence to the Minimum Point

A natural question is: does the gradient descent (GD) solver converge to the minimum? Supplementary Figure B.2 demonstrates that the constraint value in C-GNS-GD approaches a constant in five or more solver iterations. It is expected, as the model was trained in conjunction with the GD solver with five iterations. Using a loss on the intermediate updates with  $\alpha=0.25$  further improves the convergence. We found that this loss is crucial for the generalization (Section 5.4), but does not affect the MSE otherwise (Figure 7).

We also investigate whether other gradient-based solvers are able to optimize the constraint function learned by C-GNS-GD ( $\alpha=0.25$ ) model at test time (Figure B.3). Other solvers, such as quasi-Newton BFGS method, find the solution with a similar constraint value and a similar MSE error to the GD solver. This finding suggests that it is sufficient to train the C-GNS model with GD solver with a fixed number of iterations in order to learn a well-behaved constraint function with the minimum near the ground-truth.



**Figure 5. Generalization to more solver iterations and larger ROPE systems at test time.** (a) Test rollout MSE for ropes with the same lengths as those during training (5-10 nodes) (b) Test rollout MSE for larger ropes (20 nodes). The x-axis indicate the number of solver iterations at test time. Vertical dashed line marks 5 iterations used at training. The y-axis represents MSE values. The horizontal black and grey lines show the performance of the Forward GNN models, which do not have an equivalent of iterations. (c) Example of the generalization rollouts from C-GNS-GD with a different number of solver iterations used at test time at every time step. The rope examples shown at time points  $T = \{20, 60, 100\}$  of the rollout. These rollouts are from a single C-GNS-GD model trained with 5 iterations.

#### 5.4. Generalizing to Larger Systems with More Iterations

A unique feature of our C-GNS-GD model is that the number of solver iterations can be increased to potentially improve the quality of the model’s predictions. We explore this property on the rope simulation in two settings: on the same dataset used during the training and on a generalization to the ropes with twice as many nodes. We compare the generalization of C-GNS-GD model to the Iterative GNN (Section 3.4) which also iteratively refines the solution, but does not use the constraint gradients. For this section, we use an additional loss ( $\alpha = 0.25$ ) on intermediate updates  $Y^{(i)}$  to further incentivize the convergence to the ground-truth point.

**ROPE dataset** We pre-train the C-GNS-GD and Iterative GNN models on the ROPE with  $N_{\text{train}} = 5$  iterations. Then we investigate how the test error changes as we vary the number of solver iterations at test time in  $N_{\text{test}} \in [0, 15]$ . Figure 5(a) shows that for C-GNS-GD the rollout MSE on the ROPE decreases from iteration 1 to 5 and then remains relatively constant: error decreases by 1.2% from iteration

5 to 15, while Iterative GNN the error increases by 47%.

**Generalization to a Larger Rope** We test if the models can generalize to a rope with twice as many nodes (20 nodes versus 5-10 during training) (Figure 5b). Crucially, for C-GNS-GD ( $\alpha = 0.25$ , red), increasing the solver iterations systematically improves the rollout accuracy on the generalization task (3-fold decrease on the rollout error between iterations 5 and 9 in Figure 5(b), 26.2% decrease on 1-step error in Figure B.4(c)). We emphasize that this experiment was performed on a new dataset and with more solver iterations on each of 160 steps of the rollout – none of these conditions were observed at training time.

Note also that this result is achieved with a shallow C-GNS-GD model with 2 message-passing layers that spans 1/10 of the rope length on generalization task. By contrast, the performance of the Iterative GNN (blue) with the same number of MP layers stays the same for  $N_{\text{test}} > 4$  in Figure 5(b). It demonstrates that C-GNS-GD can leverage extra computational resources at test time, because of the inductive bias that the solver should (approximately) converge to a solution.

In Figure 5a-b we also compared C-GNS-GD to the Forward GNN with the same number of parameters (2 MP, grey line). We find that the C-GNS-GD has an order of magnitude better performance, on both the ROPE and the generalization dataset. Next, we compared C-GNS-GD to a deeper Forward GNN with 10 MP (black line). Even though the state-of-the-art Forward GNN (10 MP) is slightly better on the ROPE dataset (Figure 5a), C-GNS-GD achieves about 50% lower error when generalizing to the larger system by leveraging additional optimization iterations (Figure 5b).

Supplementary Figure B.4 demonstrates the similar analysis for the models *without* the additional loss on the intermediate states. We find that without the additional loss, both C-GNS and Iterative GNN models tend overfit to the number of solver iterations used at training, although, Iterative GNN overfits more.

In this section we showed that by increasing the number of iterations  $N$  at test time, C-GNS-GD can achieve more accurate solutions without re-training or fine-tuning the model. To our knowledge this is the first demonstration of leveraging additional resources to improve generalization to a larger system in the domain of learned physical simulations.

#### 5.5. Incorporating Novel Constraints at Test Time

A unique advantage of the constraint-based model is that we can incorporate additional, hand-designed constraints at test time without any fine-tuning on the model. To do so, we simply take a weighted sum of the hand-designed constraints and the learned constraint  $f_C$  and run the forward

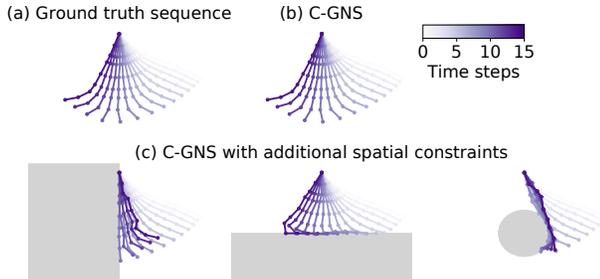


Figure 6. Adding hand-designed constraints. (a) The ground truth sequence of rope simulation 14 time steps. (b) The rollout from C-GNS-GD trained on the ROPE dataset, without added constraints. (c) The C-GNS-GD’s rollout, with wall, floor, and disk-shaped obstacles, imposed at test time via hand-designed constraint functions.

evaluation to find the solution of the joint constraint.

We designed three constraint functions for the ROPE dataset that represent the “forbidden” regions of the space: a vertical wall, a horizontal floor, and a disk-shaped region (Figure 6). The hand-designed constraints are non-negative and increase quadratically as the rope nodes enter the “forbidden” region. Figure 6 shows that the model resolves the collisions between the rope and the obstacle. This behavior is new: there are no examples of the rope interacting with other objects in the training data. Note that satisfying the additional constraint may require to slightly violate the learned constraint, which is trained on the ropes moving solely under gravity. In some rollouts, the model finds the solution where the rope links change in length to avoid the obstacle. To prevent this, we add a second hand-designed constraint to preserve the lengths of the rope links (see Videos).

More broadly, this is a powerful example of how constraint-based models can generalize to behaviors outside of their training data, and solve both for the learned dynamics and arbitrary desired constraints.

### 5.6. Examining Differences From Neural Projections

We demonstrate that our model’s key differences from Neural Projections (Yang et al., 2020) provide substantial improvements in performance. We provide ablations of our C-GNS-GD for each of these differences (conditioning on past states, using GNNs, using gradient descent), as summarized in Table 1. Figure 7 shows that C-GNS-GD has several orders of magnitude lower rollout error compared to Neural Projection. Each of our ablations towards Neural Projection had higher error than C-GNS-GD. Parameterising the constraint with a GNN instead of an MLP yields the largest improvement on all datasets, particularly on ROPE.

We found that FP-based models were difficult to train. Notice that the FP models (C-GNS-FP and C-MLP-FP) suffer

Model variant	$f_C$	Solution specification	Use $X_{\leq t}$ ?
Neural Projections	MLP	$f_C = 0$	$f_C(Y)$
C-MLP-FP	MLP	$f_C = 0$	$f_C(X_{\leq t}, Y)$
C-MLP-GD	MLP	argmin $f_C$	$f_C(X_{\leq t}, Y)$
C-GNS-FP	GNN	$f_C = 0$	$f_C(X_{\leq t}, Y)$
C-GNS-GD $f_C(Y)$	GNN	argmin $f_C$	$f_C(Y)$
<b>C-GNS-GD</b>	GNN	argmin $f_C$	$f_C(X_{\leq t}, Y)$

Table 1. Model ablations. The “Model variant” column lists the names of Neural Projections, our model, and the ablated models. The “ $f_C$ ” column indicates whether the constraint function was an MLP or GNN. The “Solution specification” column indicates how the solution was defined, i.e., as the zero point or minimum of the constraint function. The “Use  $X_{\leq t}$ ?” column indicates whether or not the constraint function operated over the previous states.

from instability across seeds. We speculate that the FP algorithm makes the training challenging because the step size  $\lambda$  is proportional to  $f_C$ . This may cause poor zero-finding early in training when the  $f_C$  is not yet informative. Additionally, we find that C-GNS-FP algorithm becomes unstable in the areas with shallow constraint gradients, perhaps because its  $\lambda$  depends on the inverse of the gradient’s norm.

We report comparisons between C-GNS-GD and Iterative GNN in Supplementary Figures B.6 and B.5. The Iterative GNN has higher 1-step error than C-GNS-GD on all our datasets and is competitive in terms of the rollout error.

### 5.7. Further Comparison to Forward GNN

We explored how varying the number of MP layers and solver iterations  $N$  at training time influenced the performance of C-GNS-GD compared to Forward GNN in our ROPE dataset (Supplementary Figure B.7). Even when training with one iteration of the solver, C-GNS-GD outperforms the Forward GNN with the same number of MP layers. The performance further improves if we train C-GNS-GD with more iterations (from 1 to 5), while using exactly the same number of model parameters. In comparison to deeper Forward GNN (right-most facet), C-GNS-GD with 4 MP layers and 5 iterations has similar 1-step and full rollout MSE to a Forward GNN with 10 MP layers, demonstrating that C-GNS-GD generally requires 2.5 times fewer MP layers than Forward GNN to achieve comparable performance.

## 6. Discussion

We presented a general-purpose framework for constraint-based learned simulation, where a learned constraint function implicitly represents the time dynamics of the physical system and future predictions are generated via a constraint solver. We implemented our constraint function as a graph network and used gradient descent as the constraint solver. Our results showed that our C-GNS has competitive or bet-

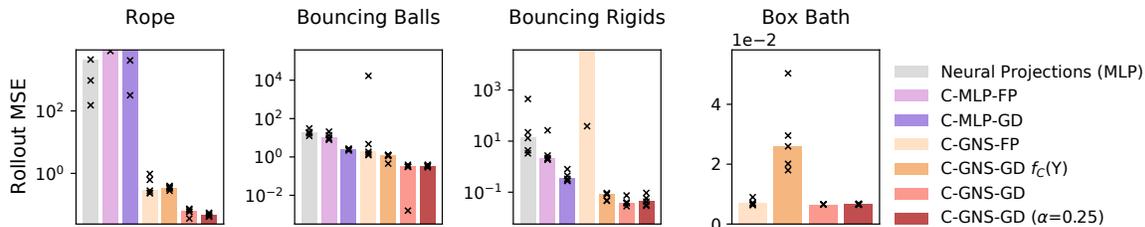


Figure 7. **Ablations towards Neural Projections model.** Y axis shows full-rollout test MSE on the node positions. The bars represent the median MSE over random seeds. The black crosses show the MSE metric for each random seed. All plots except BOXBATH are on log scale. We crop the Y axis if it exceeds the median MSE of C-GNS-GD by 5 fold. The results are not shown for MLP-based models on BOXBATH, as these models could not effectively scale to a large system (see Section 4.4). See Supplement for 1-step MSE errors.

ter performance compared to previous learned simulators in a variety of challenging physical simulation problems. We demonstrated unique abilities of C-GNS to generalize to novel, hand-designed constraints and improve the simulation accuracy on larger systems at test time by increasing solver iterations. These properties have not been previously demonstrated in the space of learned physical simulations.

Implicit constraint-based models have stronger inductive biases, compared to explicit forward simulators, offering trade-offs between expressivity, adaptive computation and allowing to incorporate manual constraint terms. One inductive bias is parameter-sharing: the gradient  $\nabla_Y f_C$  in C-GNS effectively ties the parameters across  $N$  solver iterations. In principle, a deep forward simulator can be more expressive than C-GNS: each layer in the unshared forward model could take values of the shared parameters of C-GNS. In practice, C-GNS requires 2.5 times fewer MP layers to achieve a comparable test performance to the forward simulator with 10 MP layers (Section 5.7). Another inductive bias is that C-GNS searches for a solution that converges to the fixed point. This property makes it easy and natural to incorporate novel hand-designed constraints at test time, and generalize to more solver iterations and larger systems.

One key area for further improvements in constraint-based models is the runtime. Our model applies the gradient descent solver in the forward pass, requiring  $2N$ -times longer computation time ( $N$  is the number of solver iterations) compared to the forward model with the same number of parameters. In comparison to the Iterative GNN, C-GNS requires approximately 2-times longer computation time. In both cases, the 2 multiplier is due to the gradient computation  $\nabla_Y f_C(X_{\leq X}, Y)$  via vector-Jacobian product (VJP) that is approximately 2 times more expensive than the function call itself ( $f_C(X_{\leq X}, Y)$ ) on the forward and backward pass.

Similar iterative models, such as Deep Equilibrium models (DEQ, Bai et al. (2019)), suffer from similar issues. Different techniques may help reduce the runtime and the memory

cost: using more efficient, adaptive solvers or alternative ways to compute the gradient of the solution (e.g., implicit differentiation (Liao et al., 2018), as used in DEQs).

One area where constraint-based simulation may be especially effective is in systems with hard constraints that require finding the equilibrium state of many local constraints and might leverage the adaptive computation. Domains with global constraints might also benefit from using constraint-based simulators, as it is easier to compute the constraint value than to directly propose the state that satisfies it.

Overall, the performance, generality and unique advantages of constraint-based learned simulation make it an important new direction of machine learning methods for complex simulation problems in science and engineering.

## References

- Amos, B. and Kolter, J. Z. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pp. 136–145. PMLR, 2017.
- Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization, 2016.
- Bai, S., Kolter, J. Z., and Koltun, V. Deep equilibrium models. *arXiv preprint arXiv:1909.01377*, 2019.
- Bai, S., Koltun, V., and Kolter, J. Z. Multiscale deep equilibrium models. *arXiv preprint arXiv:2006.08656*, 2020.
- Baraff, D. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 23–34, 1994.
- Bartunov, S., Nair, V., Battaglia, P., and Lillicrap, T. Continuous latent search for combinatorial optimization. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

- Battaglia, P., Pascanu, R., Lai, M., Rezende, D. J., and Kavukcuoglu, K. Interaction networks for learning about objects, relations and physics. *ArXiv*, abs/1612.00222, 2016.
- Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.
- Bouaziz, S., Martin, S., Liu, T., Kavan, L., and Pauly, M. Projective dynamics: Fusing constraint projections for fast simulation. *ACM transactions on graphics (TOG)*, 33(4):1–11, 2014.
- Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.
- Chen, R. T., Rubanova, Y., Bettencourt, J., and Duvenaud, D. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018.
- Chen, Z., Zhang, J., Arjovsky, M., and Bottou, L. Symplectic recurrent neural networks. *arXiv preprint arXiv:1909.13334*, 2019.
- Cranmer, M., Greydanus, S., Hoyer, S., Battaglia, P., Spergel, D., and Ho, S. Lagrangian neural networks. *arXiv preprint arXiv:2003.04630*, 2020.
- De Avila Belbute-Peres, F., Economou, T., and Kolter, Z. Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 2402–2411. PMLR, 13–18 Jul 2020.
- Donti, P. L., Rolnick, D., and Kolter, J. Z. Dc3: A learning method for optimization with hard constraints. *arXiv preprint arXiv:2104.12225*, 2021.
- Duvenaud, D., Kolter, Z., and Johnson, M. Deep implicit layers - neural odes, deep equilibrium models, and beyond, 2020. URL <http://implicit-layers-tutorial.org/>.
- Finzi, M., Wang, K. A., and Wilson, A. G. Simplifying hamiltonian and lagrangian neural networks via explicit constraints. *arXiv preprint arXiv:2010.13581*, 2020.
- Goldenthal, R., Harmon, D., Fattal, R., Bercovier, M., and Grinspun, E. Efficient simulation of inextensible cloth. In *ACM SIGGRAPH 2007 papers*, pp. 49–es. ACM New York, NY, USA, 2007.
- Greydanus, S., Dzamba, M., and Yosinski, J. Hamiltonian neural networks. *Advances in Neural Information Processing Systems*, 32:15379–15389, 2019.
- Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., and Yang, L. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.
- Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021.
- Li, Y., Wu, J., Tedrake, R., Tenenbaum, J. B., and Torralba, A. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. In *ICLR*, 2019.
- Liao, R., Xiong, Y., Fetaya, E., Zhang, L., Yoon, K., Pitkow, X., Urtasun, R., and Zemel, R. S. Reviving and improving recurrent back-propagation. In *ICML*, 2018.
- Loula, J., Allen, K., Silver, T., and Tenenbaum, J. Learning constraint-based planning models from demonstrations. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5410–5416. IEEE, 2020.
- Lutter, M., Ritter, C., and Peters, J. Deep lagrangian networks: Using physics as model prior for deep learning. *arXiv preprint arXiv:1907.04490*, 2019.
- Macklin, M., Müller, M., Chentanez, N., and Kim, T.-Y. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12, 2014a.
- Macklin, M., Müller, M., Chentanez, N., and Kim, T.-Y. Unified particle physics for real-time applications. *ACM Trans. Graph.*, 33(4), jul 2014b. ISSN 0730-0301. doi: 10.1145/2601097.2601152. URL <https://doi.org/10.1145/2601097.2601152>.
- Mirtich, B. and Canny, J. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D ’95, pp. 181–ff., New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917367. doi: 10.1145/199404.199436. URL <https://doi.org/10.1145/199404.199436>.
- Monaghan, J. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68:1703, 07 2005. doi: 10.1088/0034-4885/68/8/R01.
- Mrowca, D., Zhuang, C., Wang, E., Haber, N., Fei-Fei, L., Tenenbaum, J. B., and Yamins, D. L. Flexible neural representation for physics prediction. *arXiv preprint arXiv:1806.08047*, 2018.

- Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.
- Nocedal, J. and Wright, S. J. *Numerical Optimization*. Springer, New York, NY, USA, 2e edition, 2006.
- Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., and Battaglia, P. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021. URL [https://openreview.net/forum?id=roNqYL0\\_XP](https://openreview.net/forum?id=roNqYL0_XP).
- Rubanova, Y., Chen, R. T., and Duvenaud, D. Latent odes for irregularly-sampled time series. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pp. 5320–5330, 2019.
- Sanchez-Gonzalez, A., Heess, N., Springenberg, J. T., Merel, J., Riedmiller, M., Hadsell, R., and Battaglia, P. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pp. 4470–4479. PMLR, 2018.
- Sanchez-Gonzalez, A., Bapst, V., Cranmer, K., and Battaglia, P. Hamiltonian graph networks with ode integrators. *arXiv preprint arXiv:1909.12790*, 2019.
- Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J., and Battaglia, P. Learning to simulate complex physics with graph networks. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 8459–8468. PMLR, 13–18 Jul 2020. URL <https://proceedings.mlr.press/v119/sanchez-gonzalez20a.html>.
- Stachenfeld, K., Fielding, D. B., Kochkov, D., Cranmer, M., Pfaff, T., Godwin, J., Cui, C., Ho, S., Battaglia, P., and Sanchez-Gonzalez, A. Learned coarse models for efficient turbulence simulation. *arXiv preprint arXiv:2112.15275*, 2021.
- Thomaszewski, B., Pabst, S., and Strasser, W. Continuum-based strain limiting. In *Computer Graphics Forum*, volume 28, pp. 569–576. Wiley Online Library, 2009.
- Thuerey, N., Weissenow, K., Prantl, L., and Hu, X. Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows. *AIAA Journal*, 58(1):25–36, 2020.
- Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012.
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.
- Wang, P.-W., Donti, P., Wilder, B., and Kolter, Z. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pp. 6545–6554. PMLR, 2019.
- Witkin, A., Gleicher, M., and Welch, W. Interactive dynamics. *SIGGRAPH Comput. Graph.*, 24(2):11–21, feb 1990. ISSN 0097-8930. doi: 10.1145/91394.91400. URL <https://doi.org/10.1145/91394.91400>.
- Wu, J.-L., Xiao, H., and Paterson, E. Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework. *Physical Review Fluids*, 3(7):074602, 2018.
- Yang, S., He, X., and Zhu, B. Learning physical constraints with neural projections. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 5178–5189. Curran Associates, Inc., 2020. URL <https://proceedings.neurips.cc/paper/2020/file/37bc5e7fb6931a50b3464ec66179085f-Paper.pdf>.
- Zhang, Y., Sung, W. J., and Mavris, D. N. Application of convolutional neural network to predict airfoil lift coefficient. In *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, pp. 1903, 2018.
- Zhong, Y. D., Dey, B., and Chakraborty, A. Extending lagrangian and hamiltonian neural networks with differentiable contact models. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 21910–21922. Curran Associates, Inc., 2021. URL <https://proceedings.neurips.cc/paper/2021/file/b7a8486459730bea9569414ef76cf03f-Paper.pdf>.

## Supplementary Material

### A. Implementation

#### A.1. The datasets

We generate the ROPE, BOUNCING BALLS and BOUNCING RIGIDS datasets using the MuJoCo physics simulator, with a timestep of 0.001, and recording every 30th time step for our datasets. Our MuJoCo datasets contain 8000/100/100 train/validation/test trajectories of 160 time points each. We show examples of the rollouts for each environment in Supplementary Figure B.1 and Videos.

**ROPE** The rope is a mass-spring system, where the masses are represented by nodes, and the springs are represented by edges. We randomly sample the number of masses from the discrete interval [5, 10], and the rest length of the springs from the interval [0.6, 1.1]. The springs have effectively infinite stiffness, and thus maintain their rest lengths during the simulation. The rope is fixed in space at one end, and the rest moves under the force of gravity in 2D space.

**BOUNCING BALLS** The bouncing balls are a 2D particle system confined to a square box, where interactions between the balls, and between the balls and walls, are simulated as rigid collisions. The number of balls is randomly sampled from the discrete interval [5, 10], and the radii of each ball from the interval [0.11, 0.3]. The size of the box is fixed to 5x5 in MuJoCo coordinates.

**BOUNCING RIGIDS** The bouncing rigids are similar to BOUNCING BALLS, except all the balls are connected to each other with rigid bars. We randomly sample the number of balls from the discrete interval [3, 6].

**BOXBATH** This dataset is from (Li et al., 2019), and simulates 3D fluid particle dynamics within a box, with a rigid cube comprised of particles floating on the surface of the fluid, using the NVIDIA FleX physics engine (Macklin et al., 2014b). Each simulation contains 960 fluid particles and 64 particles representing the cube. The dataset contains 2700/10/100 training/validation/test trajectories with 150 time steps each.

#### A.2. Constructing the input graph

We construct the input graph such that the representation is translation invariant, motivated by the idea that the laws of physics do not change based on position in space. To do so, we never provide absolute positions of the nodes in the input to the GNN. Instead, we use velocities (position differences across time) as node features and pairwise position differences between the nodes as edge features, as described below and in the main text. In preliminary work, we found that providing absolute positions to the models causes poorer generalization, especially in larger environments, such as a longer rope.

In BOUNCING RIGIDS and BOUNCING BALLS we use a fully-connected graph. In ROPE we add edges between nodes that are adjacent within the rope. In BOXBATH we add edges between particles that are within a radius of 0.08 from within each other, and then recompute these edges at every step of a rollout according to the updated positions (as in Sanchez-Gonzalez et al. (2020)).

##### A.2.1. NODE FEATURES

To construct each input node feature, we use the concatenation of the three most recent velocities (position differences) of the node, concatenated with the static parameters (context  $X_{\leq t}$ ):  $[z^j, v_{t-2}^j, v_{t-1}^j, v_t^j]$ . See Figure B.10(a-b) for experiments with different number of time points. We use five most recent velocities for BOXBATH to match the paradigm in (Sanchez-Gonzalez et al., 2020). For the constraint-based models, e.g. C-GNS-GD, we also concatenate the optimized update  $Y$  to the node features, as  $Y$  represents velocity or acceleration for each node.

We provide an additional one-hot node feature indicating the node type (e.g. rigid, fluid, fixed). For BOUNCING BALLS and BOUNCING RIGIDS, we provide the radius of the object as an additional node feature. Note, because the edges’ relative positional displacement and distance features are computed between centers of the nodes, the model must factor in the object size feature to determine whether a collision is happening.

**Handling walls** To handle the walls, we include the Euclidean distance between the center of the node to each of wall as additional node features, treating the wall as a plain, similarly to (Sanchez-Gonzalez et al., 2020).

We clip the distance to the wall at a fixed maximum value so that this feature cannot be exploited by the network to infer the absolute position within the box. For BOUNCING BALLS and BOUNCING RIGIDS we clip the distance at 2.0, and for BOXBATH at 0.08. For the constraint-based and iterative models, we update the distances to the walls after every step of constraint optimization or iteration, respectively.

### A.2.2. EDGE FEATURES

To construct each input edge feature, we use the concatenated displacement vectors  $e_t^{jk} = p_t^k - p_t^j$  between the most recent positions at time point  $t$  for the nodes  $j$  and  $k$  connected by the edge. Through the ablation studies, we found that it is sufficient to provide the displacements between the nodes only for the most recent time point  $t$ .

For BOXBATH, we also provide the vector norm of the relative distances (not just the vector itself) as an additional edge feature to match (Sanchez-Gonzalez et al., 2020).

Note that we do not provide the “rest shapes” (the ground-truth distances of the nodes) for the rigid structures or the rope. When generating a rollout, the model only observes the pairwise displacements/distances between nodes predicted in the previous steps. This makes the rollout prediction more challenging, as the rigid shape might gradually drift from true “rest shape” during the rollout, and there is no way to recover the original shape.

### A.2.3. FURTHER DETAILS

**Parameterizing the update  $Y$**  For ROPE, BOUNCING BALLS, BOUNCING RIGIDS we use the velocity (defined as difference between the positions at adjacent time points) as the update  $Y$ . At the first iterations,  $Y^{(0)}$  is initialized to the previous velocity  $V_t$ . For BOXBATH, we use normalized acceleration of the particle as the update  $Y$  to better match the approach in (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021). The acceleration is estimated as a backward difference:  $\hat{A}_{t+1} = V_{t+1} - V_t = P_{t+1} - 2P_t + P_{t-1}$ . In this case, the UPDATER takes the form  $\hat{P}_{t+1} = P_t + V_t + \hat{A}_{t+1} = 2P_t - P_{t-1} + \hat{A}_{t+1}$ , where  $\hat{A}_{t+1}$  is produced by the PREDICTOR. The update  $Y^{(0)}$  is initialized to a zero vector on the first iteration. In both cases, the proposed  $Y$  is concatenated as an extra node feature at each optimization iteration.

**Normalization** For BOXBATH we found it was important to normalize inputs and targets to zero-mean unit-variance (as in Sanchez-Gonzalez et al. (2020)). In the other datasets, the scale of the features was already close to zero-mean unit-variance, except for the input/target velocities in BOUNCING BALLS and BOUNCING RIGIDS, so we scaled them by a factor of 100.

**Noise** To stabilize rollouts in BOXBATH, we added noise to the input sequences in the same manner, and with the same magnitude, as in (Li et al., 2019; Sanchez-Gonzalez et al., 2020).

**Fixed particle** The ROPE dataset contains a “fixed” node that does not change its position throughout the simulation. Generally, any simulator (engineered or learned) needs an indication that a particle is fixed. Otherwise, it is ambiguous if a particle with zero velocity is temporarily motionless or permanently fixed. Many physical engines and all relevant baselines (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021; Yang et al., 2020) explicitly prevent the update for fixed particles. We chose to follow this practice.

For gradient-based constraint models, we prevent the update for the fixed nodes by using *stop\_gradient*, similarly to (Yang et al., 2020). Note that this technique does not require network’s access to the absolute positions of the particles. For other models, we override fixed particles positions to remain static during a rollout for all models. We also mask out fixed particles from the loss computation.

An alternative way to handle fixed particles would be to provide the model the “fixed” label as an input feature, and expect the model to learn to predict zero velocity for fixed particles. We ran the experiment and confirmed that our model, as is, is able to learn this behavior.

## A.3. Model Implementation

**Computing the constraint gradients** To compute the gradients of the constraint scalars for the batch of graphs, we use the vector-Jacobian product (VJP) function using JAX. VJP does not explicitly construct a Jacobian, and its asymptotic computational cost is the same as the forward evaluation of the constraint function.

**Constraint function** To construct  $f_C$ , we first encode the nodes and edges of the graph using MLP encoders. Then, we process the graph using a GNN model from (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021). The GNN model has residual connections on each message-passing layer and does not use global updates. Next, we decode the node outputs of the graph network using an MLP decoder with a scalar output to compute the per-node values  $\{c^j | j = 1 \dots J\}$ . Finally, to obtain the scalar constraint value for the entire graph, we use the mean aggregation for the per-node values  $c = f_C(X_{\leq t}, \hat{Y}) = \frac{1}{J} \sum_{j=1}^J (c^j)^2$ . For gradient descent solver, we take a square of per-node outputs before aggregating them. For fast projections, we simply take the sum of per-node outputs.

We use a fixed learning rate of 0.001 for gradient descent-based constraint solvers. We did not find the model to be very sensitive to this value of the learning rate. We speculate this is because the model can indirectly control the learning rate by learning an arbitrary scaling factor for the constraint function. We use five iterations of the solver for both gradient descent and fast projection solvers during the training.

**Loss** In most experiments we used the MSE loss between the output of the last iteration  $\hat{Y}^{(N)}$  and the corresponding ground-truth state updates on node positions. In section 5.4 we used the additional MSE loss between intermediate states  $\hat{Y}^{(i)}$  and the ground-truth point, with exponentially decaying weights  $\mathcal{L}_{interm} = \sum_{i=1}^N \alpha^{N-i} \text{MSE}(\hat{Y}^{(i)}, T)$ , where  $T$  is the ground-truth,  $N$  is the number of solver iterations,  $\alpha$  is a parameter in  $(0, 1]$ . The goal of the weighted loss is to encourage the solver to reach the solution in fewer iterations. Earlier iterations have a smaller weight and are penalized less for being farther from the ground-truth. The loss on the last iteration is the same as our standard MSE loss between the last iteration and the ground-truth, as the weight on  $N$ -th iterations is  $\alpha^{N-N} = 1$ .

The results on the MSE error and constraint values and gradients with different choices of  $\alpha$  are provided on Supplementary Figure B.9. We used  $\alpha = 0.25$  for both C-GNS-GD and Iterative GNN in Section 5.4.

**Fast Projections** Fast Projection (FP) algorithm (Goldenthal et al., 2007) is a zero-finding algorithm, for constraint functions whose solutions are defined as,  $f_C(X_{\leq t}, Y) = 0$ . FP uses an adaptive step

$$\lambda = - \frac{f_C(X_{\leq t}, Y^{(i)})}{\|\nabla_Y f_C(X_{\leq t}, Y)|_{Y=Y^{(i)}}\|^2}.$$

Then FP updates the proposed state analogous to our C-GNS-GD model,

$$\begin{aligned} \delta Y &= -\lambda \nabla_Y f_C(X_{\leq t}, Y)|_{Y=Y^{(i)}} \\ Y^{(i+1)} &= \delta Y + Y^{(i)} \end{aligned}$$

For our experiments with Fast Projection, we use  $N = 5$  iterations during training, same as for the gradient descent solver.

**Forward GNN** For the Forward GNN, we use the Graph Network Simulator (GNS) model (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021).

The PREDICTOR takes only the context  $X_{\leq t}$  and directly outputs the update  $\hat{Y}$ . For Forward GNN, the update  $Y$  is set to the acceleration  $A_{t+1} = V_{t+1} - V_t = P_{t+1} - 2P_t + P_{t-1}$ , for consistency with the previous work. Then the update rule in the UPDATER becomes  $\hat{P}_{t+1} = P_t + V_t + \hat{A}_{t+1} = 2P_t - P_{t-1} + \hat{A}_{t+1}$ .

The graph with the context  $X_{\leq t}$  is built similarly to the one for C-GNS-GD. After the graph is processed by the graph network, the model uses a per-node MLP decoder to output the update values for each node  $(\hat{y}^j)_{j=1 \dots J}$

**Iterative GNN** In the Iterative GNN, the function  $f_{DI}$  takes both the context  $X_{\leq t}$  and the proposed update  $Y^{(i)}$  and outputs a change to the proposed update  $\delta Y$ . Then, the model computes the update variable for the next iteration as  $Y^{(i+1)} = Y^{(i)} + \delta Y$ . The PREDICTOR outputs the update variable from the last iteration  $Y^{(N)}$ .

The input to the  $f_{DI}$  at each iteration  $i$  is constructed the same way as in C-GNS. We take the graph representing the context  $X_{\leq t}$  and concatenate the proposed update  $Y^{(i)}$  to each node vector. We use a GNS model with a per-node decoder from (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021) to process the input graph and output  $\delta Y$  for each node.

We set meaning of the update  $Y$  similarly to the C-GNS:  $Y$  represents the future velocity  $V_{t+1}$  on ROPE, BOUNCING BALLS and BOUNCING RIGIDS; and acceleration  $A_{t+1}$  for BOXBATH. The corresponding UPDATER is also the same as in C-GNS

(see Section 4.2). For the first iteration, we initialize  $Y^{(0)}$  to the most recent velocity  $V_t$ , or to a zero vector if  $Y$  represents the acceleration (BOXBATH).

**C-GNS-GD- $f_C(Y)$**  This model is similar to C-GNS-GD, except the past states  $X_{\leq t}$  are not provided as part of the input. For this ablation, the model directly optimizes the positional information of the future state, similarly to (Yang et al., 2020), rather than velocity or acceleration. Thus, the update  $Y$  is set to the positions  $P_{t+1}$ , as in the Neural Projections model. The corresponding UPDATER becomes simply an identity function.

To construct the input graph, we use only the positional information of the proposed future state  $P_{t+1}$  and static properties  $Z$ . Thus, the node features do not include any information about the past states, nor the proposed approximate future velocity  $V_{t+1} = P_{t+1} - P_t$ . For the edge features, we use the relative displacement vector between the positions of the nodes  $j$  and  $k$  for the future state  $e_{t+1}^{jk} = p_{t+1}^k - p_{t+1}^j$ .

**Neural Projections, C-MLP-FP and C-MLP-GD** For the models with MLP-based constraint function we use a similar setup to (Yang et al., 2020). We concatenate the features for each node into a single vector and run an MLP to produce a scalar constraint output. For Neural Projections, we include only absolute positions of the nodes into the input. For C-MLP-FP and C-MLP-GD we additionally use the context of the past states, including absolute positions, velocities and distances to the walls for each node. MLP-based models are not provided with explicit pairwise position displacements between the particles, which for the GNN-based models would be in the edge features of the graph. Therefore, we include absolute positions as input to the MLP-based models instead. Next, MLP-based models take a fixed-sized inputs by construction and cannot handle the scenes with variable number of nodes without additional state segmentation schemes. We adapted the MLP-based models to handle scenes with variable number of nodes by padding with zeros up to the maximum state size.

We did not report results for MLP-based models on BOXBATH: MLP models take the concatenated input of node embeddings in order, and on the datasets with 1024 nodes like BOXBATH the model is likely to overfit to the specific ordering of the particles and would be unlikely to yield competitive performance. Additionally, the input vector optimized by the solver would have  $1024 \times 32$  elements resulting in very small gradients for each element.

**Hand-designed constraints** We make the hand-designed constraints to be non-negative and such that the minimum of the constraint results in the desired behavior. We parameterize these constraints with a  $c_{hand}^j = (\text{ReLU}(-D(p_j)))^2$ , where  $D(p_j)$  is the signed distance between the position of the node  $j$  and the boundary of the obstacle. If  $D(p_j) > 0$ , the node is outside of the obstacle, and the constraint is zero. If  $D(p_j) < 0$ , the node overlaps with an obstacle, and the constraint for this node becomes positive. We compute the total constraint for the entire graph as the average of per-node values:  $c_{hand} = \frac{1}{J} \sum_{j=1}^J c_{hand}^j$ . We optimize the weighted sum of the learned and hand-designed constraints:  $c_{hand} + Wc$ , where  $W$  is a constant weight, selected for each hand-designed constraint separately. Note that the hand-designed constraints and the constraints learned by the network can have different scales, and we use the weight  $W$  to bring the two constraints roughly on the same scale. We ran a grid search to find the appropriate weight  $W$ .

In some cases, optimizing the joint constraint resulted in an unexpected behavior: the rope shrinks or expands to avoid the obstacle instead of moving around it or stopping at the obstacle. This effect can be explained as follows. Recall that in the ROPE dataset used for training, the ropes do not collide with any obstacles, and the only valid next step is to continue moving under the force of gravity. When we optimize the the hand-designed obstacle constraints together with the learned constraint, the solution would inevitably violate the learned constraint, under which the rope would continue moving under gravity. In some cases, the model chooses to violate the learned constraint by changing the relative distances between the rope nodes, instead of changing the dynamics.

To incentivize the model to preserve the node distances, we add the second hand-designed constraint on the relative distances between the nodes. First, we compute the norm of the distances between each pair of adjacent nodes at the current time point and at the first time point of the simulation. Then we set the constraint to the squared difference between the distance norms at the two time steps, and average for all pairs of adjacent nodes. This constraint is minimized when the distances between each pair of nodes remain the same. See Videos for the rollouts with obstacle constraint only and the rollouts with both obstacle and distance constraints.

#### A.4. Complexity analysis

Here we provide a rough complexity analysis of the models mentioned in this paper. For the C-GNS and Iterative GNN, we assume that we backpropagate through all the iterations of the solver, although the techniques like implicit differentiation or backpropping through only the last few iterations of the solver can reduce the runtime.

The running times at inference and training are:

- Forward GNN:  $c \times M$
- Iterative GNN:  $c \times M \times N$
- C-GNS:  $c \times M \times N \times b$

where  $M$  is the number of message passing steps,  $N$  is the number of solver iterations,  $c$  is a common constant for all three models, the  $b$  is the factor between [1,2).  $b$  accounts for the extra time it takes to compute the the gradient of the constraint function with respect to the inputs via vector-Jacobian product ( VJP). The VJP computation takes roughly 2x longer than the evaluation of the function itself. The multiplier 2 comes from the fact that VJP needs to evaluate the function and compute the gradient for each of the weights.

#### A.5. Algorithm

---

##### Algorithm 1 Constraint-based graph network simulator

---

#  $X_{\leq t} := (P_{t-3}, P_{t-2}, P_{t-1}, P_t)$ : a sequence of the most recent positions (the static parameters are omitted in this algorithm)

**def** CONSTRAINT( $X_{\leq t}, Y$ )  
 $G = \text{CONSTRUCTGRAPH}(X_{\leq t}, Y)$  ▷ see section 4.2 on construction of the input graph  
 $G' = \text{GRAPHNETWORK}(G)$   
 $\{c_j\}_{j=1..J} = \{\text{MLP}(n_j)\}_{n_j \in G'}$  ▷ compute a scalar value for each node  $n_j$  of graph  $G'$   
▷  $J$  is the number of nodes in the graph  $G'$   
▷ Compute a single scalar value for a graph  
**Return:**  $\frac{1}{J} \sum_{j=1}^J (c_j)^2$

**def** GRADIENTDESCENTSOLVE( $X_{\leq t}, Y^{(0)}$ )  
 $\lambda = 0.001$   
**for** k in  $1 \rightarrow N$ : **do**  
 $\delta Y = -\lambda \nabla_Y \text{CONSTRAINT}(X_{\leq t}, Y)|_{Y=Y^{(k)}}$   
 $Y^{(k+1)} = Y^{(k)} + \delta Y$   
**Return:**  $Y^{(N)}$

**def** CONSTRAINTGRAPHNETSIMULATOR( $X_{\leq t}$ )  
 $P_{t-3}, P_{t-2}, P_{t-1}, P_t = X_{\leq t}$  ▷ Unpack the parameters of the history  
 $Y^{(0)} = P_t - P_{t-1}$  ▷ Initialize the proposed update to the most recent velocity.  
 $\hat{Y} = \text{GRADIENTDESCENTSOLVE}(X_{\leq t}, Y^{(0)})$   
 $\hat{X}_{t+1} = \text{UPDATER}(X_{\leq t}, \hat{Y})$   
**Return:**  $\hat{X}_{t+1}$

---

#### A.6. Hyperparameters

**Choice of the number of message-passing layers** We chose the smallest possible number of message passing (MP) steps that would allow the C-GNS family of models to solve the task with 5 optimization iterations (2 MP for ROPE, 1MP steps for all other datasets). We then compared this model to a Forward GNN (GNS) with the same architecture and number of parameters for the main result. A full comparison of Forward GNN and C-GNS across multiple values of message passing steps and optimization iterations on ROPE is available in Figure B.7.

For all GNNs, we used a residual connection for the nodes and edges on each message-passing layer. GNNs have only node and edge updates and do not use global updates.

**Choice of the activation function** We noticed that the choice of the activation function affected the Fast Projection method more than Gradient Descent. For each dataset, we chose the activation function for which Fast Projection algorithm (C-GNS-FP) was more stable (more random seeds converged). Then we used the same activation function for other models, including Gradient Descent (C-CNS-GD), as the choice of the activation function had little impact on the performance for other models.

**Rope** For GNN-based models, we used 2 message-passing steps. We use the latent size of 32 for nodes and edges. The MLPs for processing nodes and edges, as well as node encoder and decoder MLPs, have 3 hidden layers with 256 hidden units each. We used `softplus` activation and a LayerNorm (Ba et al., 2016).

**Bouncing Balls** For GNN-based models, we used 1 message-passing step. We use the latent size of 32 for nodes and edges. The MLPs for processing nodes and edges, as well as node encoder and decoder MLPs, have 3 hidden layers with 256 hidden units each. We use `softplus` activation and LayerNorm after every MLP, except the final decoder.

**Bouncing Rigids** For GNN-based models, we used 1 message-passing step. We use the latent size of 32 for nodes and edges. The MLPs for processing nodes and edges, as well as node encoder and decoder MLPs, have 3 hidden layers with 256 hidden units each. We use `tanh` activation and LayerNorm after every MLP, except the final decoder.

**Box Bath** For GNN-based models, we used 1 message-passing step. All other hyperparameters are as in (Sanchez-Gonzalez et al., 2020). The GNNs’ node and edge function MLPs each had 2 hidden layers with 128 hidden units, and hidden node and edge latent sizes of 128 each. We use `softplus` activation and LayerNorm after every MLP, except the final decoder.

**Models with MLP constraints** For MLP-based models (Neural Projecitons, C-MLP-FP and C-MLP-GD), we used an MLP with 5 hidden layers and 256 hidden units, following the architecture in (Yang et al., 2020). We used `softplus` activation with no LayerNorm.

**Training** We train the models for 1M steps on ROPE, BOUNCING BALLS and BOUNCING RIGIDS. We used the Adam optimizer with an initial learning rate of 0.0001, and a decay factor of 0.7 applied with a schedule at steps (1e5, 2e5, 4e5, 8e5). We use a batch size of 64. We trained for 2.5M steps for the experiments studying the number of solver iterations. On BOX BATH we trained for 2.5M steps with a batch size of 2, and a learning rate starting at 0.001 and decaying continuously at a rate of 0.1 every 1M steps, as in Sanchez-Gonzalez et al. (2020).

### A.7. Limitations of “Neural Projections” (Yang et al., 2020)

The Neural Projections (NP) by (Yang et al., 2020) is another approach involving learned constraint-based simulation. However, it has several fundamental limitations that make it insufficient as a general-purpose learned simulator. Here we elaborate on these limitations, briefly described in the main text.

**Limitation of static constraints** The learned constraint in NP model depends only on the proposed future positions (i.e. the “static” state). Thus, NP cannot represent constraints that depend on two or more states across time by construction. For example, NP’s constraint function on its own cannot model the time dynamics of a single particle with constant velocity, which is presumably why NP applies an Euler step to initialize the first proposal that is passed to the constraint solver. NP is also in principle not well suited to model elastic collisions properly, as illustrated in Figure A.1. In the top scenario, the Euler step proposes the ball moves past the thin wall. Since NP’s constraint function does not regard this as a constraint violation, the ball will continue moving ahead as if the wall does not exist. In the bottom scenario, the Euler step places the ball within the wall, and because the nearest constraint-satisfying position for the ball is at the edge of the wall, the approach would, in principle, only correct the position of the ball until the ball stops overlapping the wall as the solution, leaving the ball right next to the wall, regardless of the ball’s initial position and velocity before the wall collision.

More generally, NP cannot enforce constraints or symmetries defined over time, such as energy preservation: once the Euler step breaks energy preservation, the proposed future state does not contain enough information about the energy of the

previous state to be able to identify a constraint violation and resolve it in a way that is consistent with the true dynamics.

Neural Projection also incorporates external forces, such as gravity, by directly updating the velocities before the Euler step, which is probably necessary because, again, NP’s constraint function cannot enforce external effects which involve time (e.g., force and acceleration relate to the second *time* derivative of the position). This is a strong assumption: it means NP must be provided with such temporal effects explicitly, along with the appropriate hard-coded update mechanism, outside of the learnable part of the architecture. Similar to these examples, there are many other types of dynamics that cannot be expressed as constraint satisfaction over the predicted state from an initial Euler proposal, so overall NP cannot be considered a general-purpose learned simulator.

By contrast, because our approach’s constraint function takes both the proposed future state and history as input, i.e.,  $f_C(X_{\leq t}, Y)$ , our method can, in principle, capture any time dynamics which explicit forward simulators can.

**Hard-coded Euler step** NP relies on an Euler step to generate the initial proposed future state for the solver. Given that forward Euler is a relatively inaccurate integrator, when the Euler proposal is not accurate, the constraint function’s lack of access to the previous state makes it difficult for NP to recover.

In our approach, the initial proposal to the solver is less important, because the constraint function can capture all aspects of the dynamics. For this reason, we simply initialized the proposal to the most recent velocity given as input (or zero acceleration for BOXBATH). However it is possible to generate initial proposals accounting for external forces or a more sophisticated dynamics prediction mechanism (e.g., an explicit forward simulator).

**MLP network and the hard-coded grouping technique** NP uses an MLP network as the constraint function, and serializes and concatenates all input features into a vector before passing to the MLP. Generally this approach is not scalable to even moderately large systems (e.g., the  $\geq 1000$  nodes in BOXBATH) or systems which vary greatly in size (thus requiring significant padding), for similar reasons that serializing images and passing them to an MLP is inferior to CNN- and Transformer-based methods.

(Yang et al., 2020) do present a scheme for grouping subsets of the input state and passing them to a shared constraint MLP, which is likely intended to overcome the above weakness, however this is more of a heuristic that takes a partial step toward a more mainstream, full-fledged sharing approach, such as our GNN constraint function.

**Lack of translation and permutation equivariance.** One of the fundamental properties for modeling physical dynamics is translation equivariance, as the laws of physics do not change based on position in space. Similarly, bodies in a physical system are equivariant to permutations: re-indexing them does not affect the dynamics. (Yang et al., 2020)’s implementation of NP lack inductive biases for translation and permutation equivariance, which may lead to poor sample complexity of learning and overfitting.

**Zero-finding Fast Projections algorithm** NP uses the Fast Projection (FP) algorithm (Goldenthal et al., 2007) to find zero points in its constraint function. In practice we found that using FP, i.e., in our C-MLP-FP and C-GNS-FP model variants, to train less stably across seeds, and harder to train with deeper networks (see the variance of random seeds in Figure 3, and trends in Figure B.8). We speculate that because FP’s step size is proportional to ratio of the constraint function’s value over the squared norm of its gradient, if, early in training, the learned constraint value is large and/or the constraint gradient norm is small, the FP algorithm may take large steps which contribute to the unstable training.

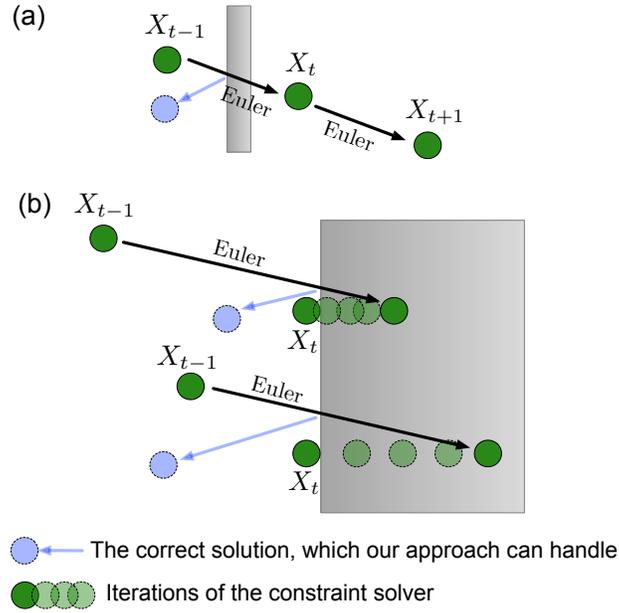


Figure A.1. Failure cases of Neural Projections (Yang et al., 2020)

**(a) Collision with a thin wall.** The Euler step in NP would propose that the ball moves through the wall. Because NP's constraint  $f_C(\text{Euler}(X_{t-1}))$  depends only on the state proposed by the Euler step, it cannot determine that there was a collision between time points  $t - 1$  and  $t$ . The ball will remain on the other side of the wall and will continue moving forward in later time steps.

**(b) Collision with a thick wall.** The Euler step in NP would propose that the ball moves into the wall, which should violate the learned constraint. The constraint-solver in NP would then move the ball to the nearest point where the constraint is not violated – the position where the ball touches the wall. As the constraint operates only on the position of the ball, but not on the previous positions or velocities, the ball would always be predicted as just touching the wall, rather than bouncing off the wall.

## B. Supplementary plots and tables

### B.1. Examples of model rollouts from C-GNS-GD models

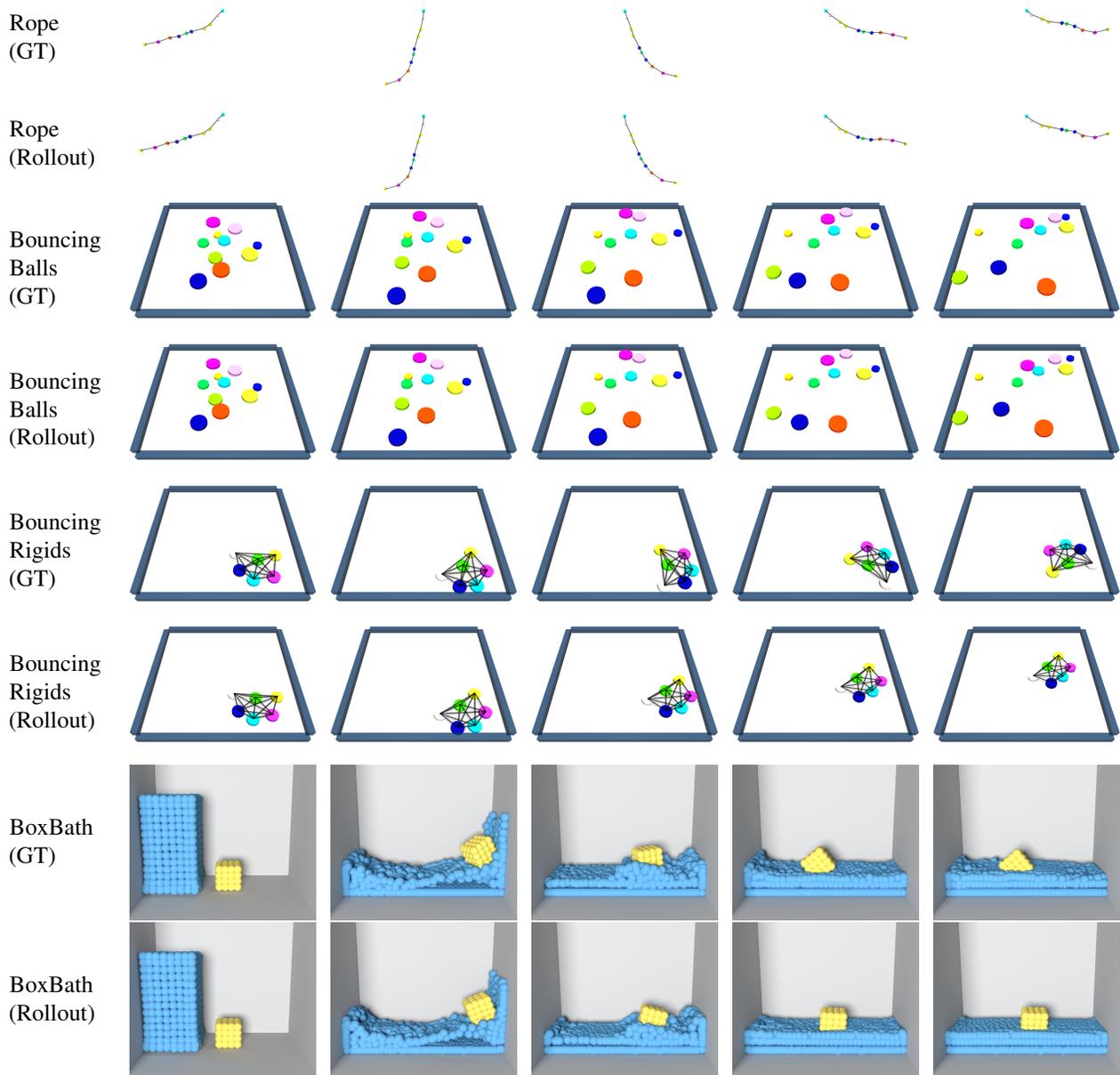


Figure B.1. Examples of the rollouts from C-GNS-GD models for our simulation environments.

### B.2. Investigating the convergence properties of the C-GNS-GD model

In this section we investigate the convergence properties of the constraint function learned by C-GNS-GD on the examples from the ROPE dataset, with and without the additional loss on intermediate states (see details in Section A.3). We train the models with 5 solver iterations and run the solver with up to 15 iterations at test time. Figure B.2 demonstrates that in C-GNS-GD with both versions of the loss, the constraint value converges to the constant value, and the gradient norm converges to zero. The model with the loss on intermediate states with  $\alpha = 0.25$  converges in fewer iterations.

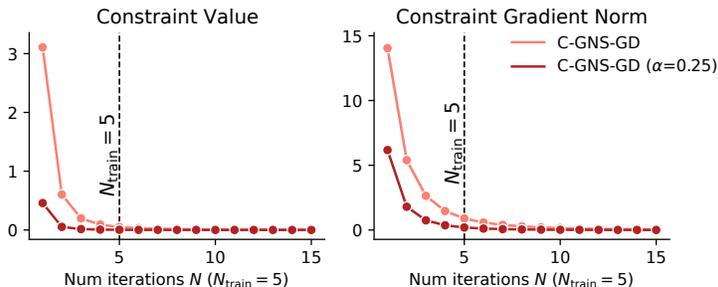


Figure B.2. The constraint value and gradient norm across the number of solver iterations on the example from the ROPE dataset. The results are shown for two models: C-GNS-GD with the MSE loss on the last iteration only (pale red) and C-GNS-GD ( $\alpha=0.25$ ) model with the additional decaying loss on intermediate states (dark red).

### B.3. Using other gradient-based solvers at test time

In Figure B.3 we further investigate whether the constraint function from the C-GNS-GD ( $\alpha=0.25$ ) model can be optimized using a different gradient-based solver at test time. We used one of the test examples in the ROPE dataset as the initial state for all the solvers we tested. We used solvers from SciPy (Virtanen et al., 2020), including Conjugate Gradient algorithm (CG, first order, Nocedal & Wright (2006)), the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS, second order method, Nocedal & Wright (2006)) and the Newton Conjugate Gradient algorithm (Newton-CG, second order method, Nocedal & Wright (2006)). We used default SciPy settings for these solvers.

The model converges when using CG (blue), BFGS (green) and Newton-CG (purple) solvers, reaching a similar mean squared error as gradient descent (dark red), indicating that the learned constraint function is robust to the choice of optimization procedure. Note that Newton-CG solver also requires computing the Hessian of the constraint function, which we also obtained via auto-differentiation of the learned constraint. The convergence of Newton-CG suggests that the second order gradients of the learned constraint function are also well-behaved, even though they were never computed during training.

Next, we varied the learning rate ( $lr$ ) of the gradient descent solver from the 0.001 value used during the training. As expected, halving the learning rate of gradient descent ( $lr=0.0005$ , orange) results in slower convergence, taking 15 iterations to reach a similar mean square error to  $GD(lr=0.001$ , dark red). On the other hand, with doubled learning rate ( $lr=0.002$ , black) the solver does not converge. We speculate that the model with  $lr=0.001$  learned a constraint function that is sufficiently steep to converge to the minimum as quickly as possible with a learning rate of 0.001, and larger values of the learning rate may be detrimental. This is possible because the model is free to learn any scale for the constraint (and its gradients), which is equivalent to re-scaling the training learning rate. We informally tested this hypothesis by verifying that the model performance is not very sensitive to the training learning rate (although very large values make training more unstable).

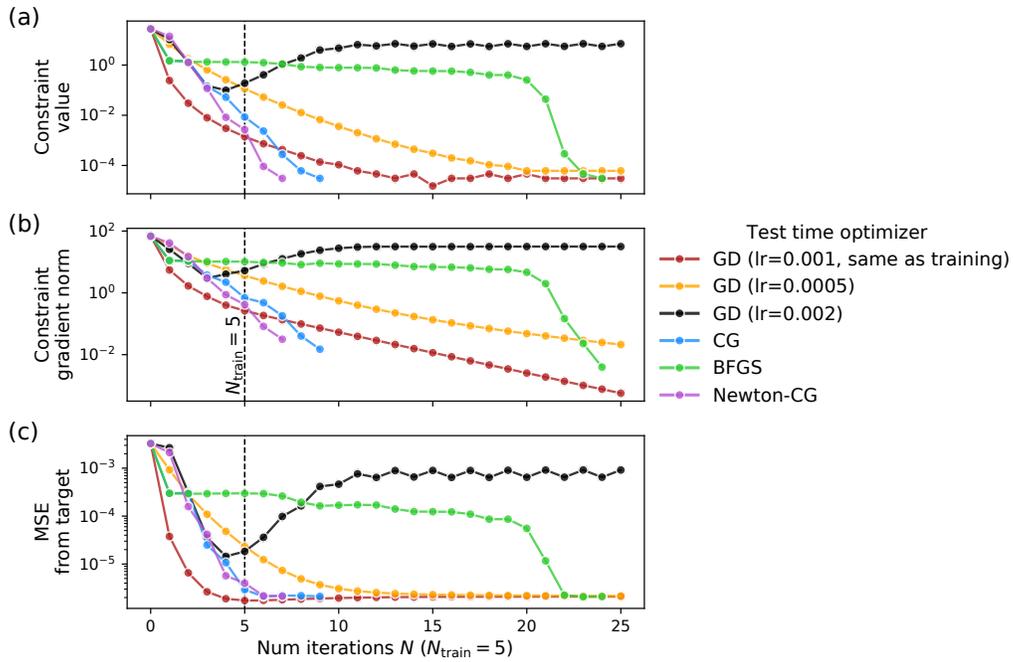
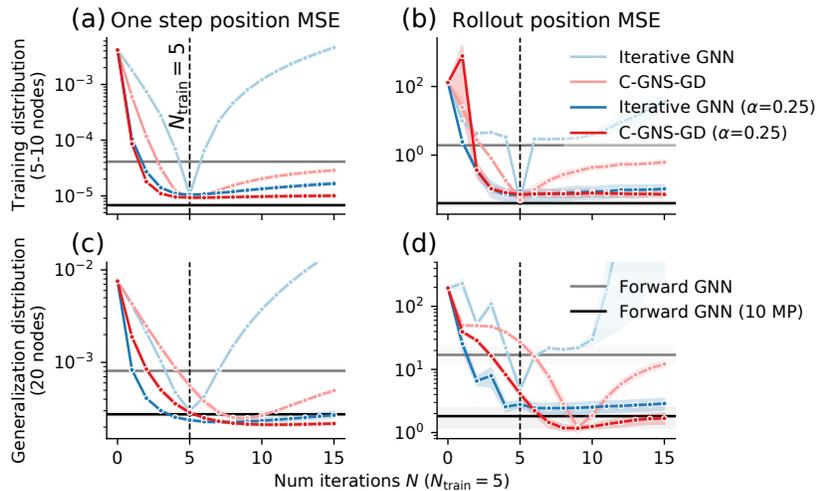


Figure B.3. **Generalization to different solvers at test time.** We run different solvers to minimize the constraint function learned by C-GNS-GD ( $\alpha=0.25$ ) model with gradient descent solver with the learning rate 0.001. We start with the initial state from one of the examples in the ROPE dataset. We ran the solvers until convergence or up to the maximum of 25 iterations. The figure show (a) the constraint value, shifted to have a minimum at zero (b), constraint gradient norm, and (c) mean squared error between the state at the current iteration and the ground-truth. These plots are on log scale.

#### B.4. Generalization to more solver iterations and larger ROPE systems at test time.

Supplementary Figure B.4 demonstrates the similar analysis for the models *without* the additional loss on the intermediate states. For the C-GNS and Iterative GNN (Supplementary Figure B.4(a-b), pale red and blue lines) the one-step and rollout errors decrease up to iteration 5 and then increase again. It indicates that, *without* the additional loss, both models overfit to the number of solver iterations used at training. Note that the Iterative GNN model has a sharp drop in the one-step and rollout errors at iteration 5 (up to one order-of-magnitude) compared to iterations 4 and 6, while the error of the C-GNS changes more smoothly across iterations. In comparison to the models *with* additional loss on the intermediate states  $Y^{(i)}$  that we discussed earlier, it is clear that both models benefit from the additional loss when generalizing to more than five solver iterations.

Note that Supplementary Figure B.4(b)-(d) is the extended version of the Figure 5(a)-(b). The figure in the main text shows only the Rollout position MSE for the Iterative GNN ( $\alpha = 0.25$ ), the C-GNS ( $\alpha = 0.25$ ), Forward GNN and Forward GNN (10 MP).



**Figure B.4. Generalization to more solver iterations and larger ROPE systems at test time.** (a-b) Test rollout MSE for ropes with the same lengths as those during training (5-10 nodes) (c-d) Test rollout MSE for larger ropes (20 nodes) than during training. The x-axis indicate the number of solver iterations used at test time. The model was trained with 5 solver iterations. The y-axis represents MSE values. The horizontal black and grey lines show the performance of the Forward GNN models, which do not have the option to vary the number of iterations at test time. The results are shown for the models with the standard MSE loss on the last iteration (pale red and blue), as well as the models with exponentially decaying loss over all iterations (bright red and blue, see details in Section A.3).

**B.5. Ablations to existing baselines**

Figure B.5 demonstrates the comparison of C-GNS-GD to Forward GNN and Iterative GNN. Adding iterative refinement of the state, but computing the update directly (Iterative GNN) improves 1-step error on ROPE and BOXBATH, but suffers from higher variance across seeds on BOUNCING BALLS and BOUNCING RIGIDS. Computing the update via constraint gradient (C-GNS-GD, ours) further improves 1-step error and improves the model stability. Both Iterative GNN and C-GNS-GD outperform Forward GNN on the full rollout.

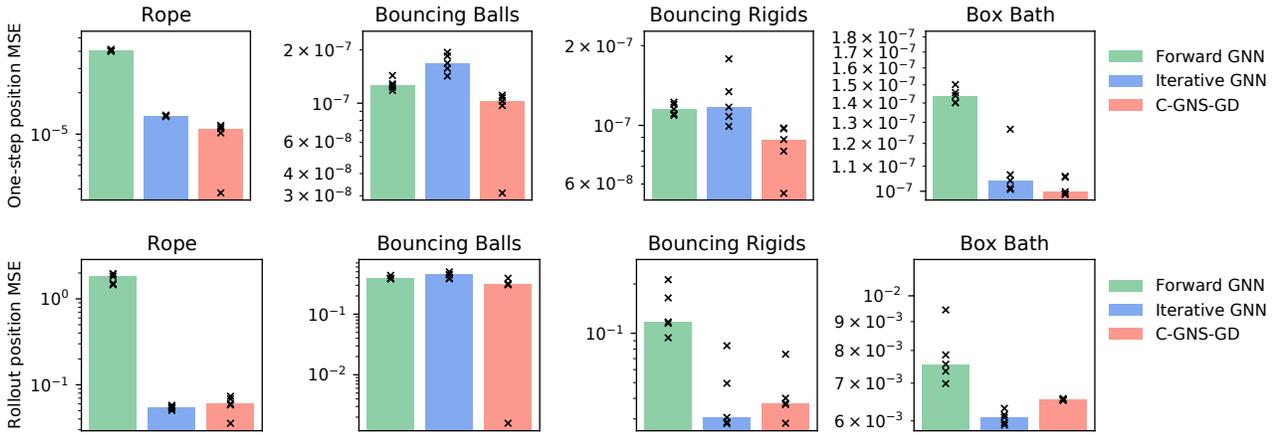


Figure B.5. Comparison to Forward GNN and Iterative GNN. Top row: one-step test MSE error on node positions. Bottom row: full 160-step rollout MSE. The bar height represents the median MSEs over random seeds. The black cross marks show the MSE metric for each random seed. The black arrows indicates that a random seeds exceeds the upper y limit of the figure.

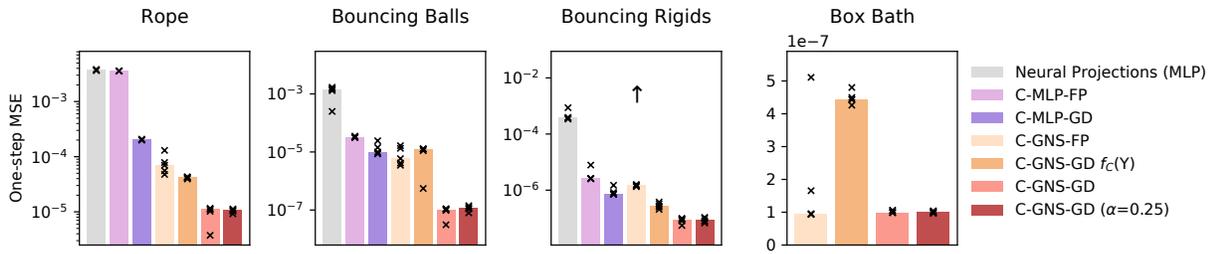


Figure B.6. Ablations to Neural Projections. One-step test MSE error on node positions. See Figure 7 for full rollout MSE. The bar height represents the median MSEs over random seeds. The black cross marks show the MSE metric for each random seed. The black arrows indicates that a random seeds exceeds the upper y limit of the figure. The upper y is set to  $1e5 \times$  the median MSE of C-GNS-GD.

Constraint-based Graph Network Simulator

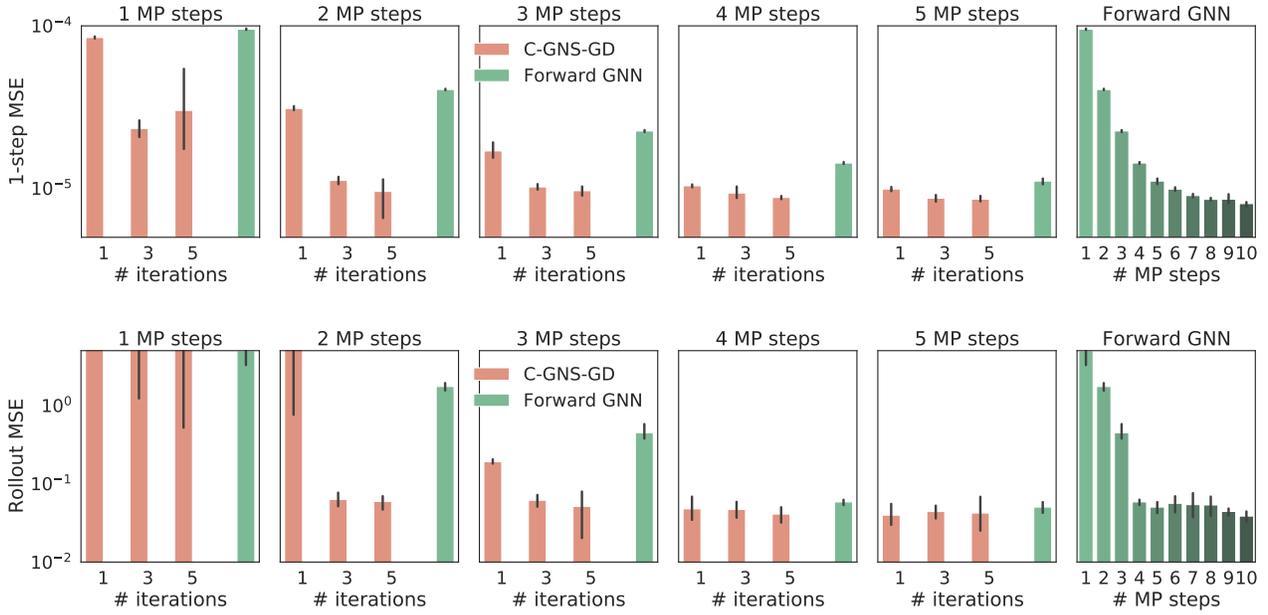


Figure B.7. Test MSE error on ROPE as a function of message-passing (MP) steps and solver iterations. Top row: 1-step test MSE error. Bottom row: full rollout test MSE error. The left five subplots shows performance of C-GNS-GD for different numbers of message-passing steps and different number of solver iterations during training. The green bars show the Forward GNN (it does not use solver iterations). The rightmost subplot shows the Forward GNN with 1 to 10 MP steps.

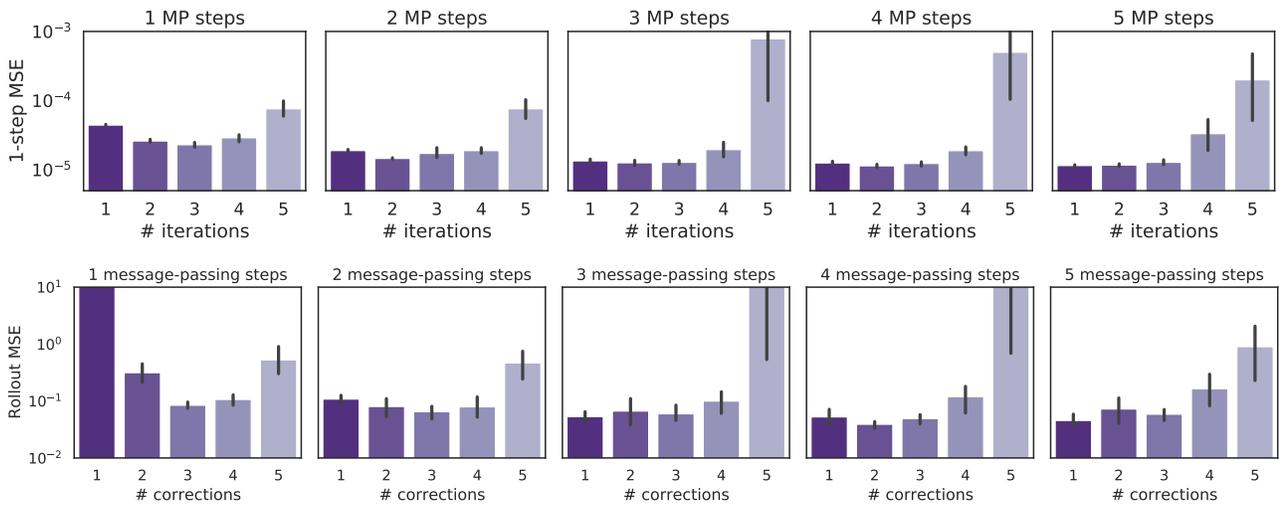


Figure B.8. Test 1-step MSE and Full Rollout MSE of the C-GNS with Neural Projection with different number of message-passing layers and number of constraint solver iterations on the Rope dataset. Top row: position MSE on the full rollout.

Table B.1. **Median performance of the models on different datasets.** The standard deviation from the median is shown over 5 random seeds. We do not show the results for the models where the median value, or standard deviation is more than 1000 times larger than the best model in each column. Note that the tables use different scales to demonstrate the errors on 1-step error, 10-step rollouts and full rollouts. Results are not shown for MLP models on BOXBATH. We omit the results for the models where the median error is more than 4 orders of magnitude larger than the median error of the C-GNS-GD model.

Model	One-step position MSE			
	Rope (1e-5)	Bouncing Balls (1e-6)	Bouncing Rigids (1e-7)	Box Bath (1e-7)
Neural Projections	370.525 ± 6.120	1434.830 ± 547.491	–	–
ConstraintMLP-FP	358.292 ± 0.532	31.984 ± 1.617	26.009 ± 23.440	–
ConstraintMLP-GD	20.422 ± 0.271	9.965 ± 6.745	7.313 ± 3.443	–
C-GNS-FP	6.983 ± 2.965	5.896 ± 5.930	–	0.954 ± 1.885
C-GNS-GD $f_C(Y)$	4.198 ± 0.137	12.241 ± 5.264	2.697 ± 0.613	4.443 ± 0.182
Forward GNN	4.062 ± 0.050	0.126 ± 0.009	1.152 ± 0.051	1.440 ± 0.038
Iterative GNN	1.355 ± 0.012	0.168 ± 0.019	1.172 ± 0.298	1.042 ± 0.103
<b>C-GNS-GD</b>	<b>1.097 ± 0.326</b>	<b>0.103 ± 0.032</b>	<b>0.884 ± 0.163</b>	<b>0.998 ± 0.038</b>
C-GNS-GD ( $\alpha=0.25$ )	<b>1.054 ± 0.070</b>	<b>0.113 ± 0.020</b>	<b>0.819 ± 0.154</b>	<b>1.000 ± 0.026</b>

Model	Rollout position MSE (10 steps)			
	Rope (1e-4)	Bouncing Balls (1e-5)	Bouncing Rigids (1e-5)	Box Bath (1e-5)
Neural Projections	9798.580 ± 176.505	–	–	–
ConstraintMLP-FP	9558.814 ± 9.186	704.608 ± 16.157	48.944 ± 392.478	–
ConstraintMLP-GD	41.000 ± 5.062	756.406 ± 177.790	6.933 ± 3.332	–
C-GNS-FP	44.577 ± 23.233	661.722 ± 372.490	–	0.200 ± 0.321
C-GNS-GD $f_C(Y)$	18.399 ± 1.090	615.769 ± 298.710	0.927 ± 0.279	43.099 ± 14.758
Forward GNN	275.851 ± 73.352	2.853 ± 0.277	12.168 ± 3.459	0.386 ± 0.061
Iterative GNN	2.506 ± 1.454	2.260 ± 0.429	<b>2.380 ± 0.257</b>	<b>0.174 ± 0.020</b>
<b>C-GNS-GD</b>	2.222 ± 0.590	<b>0.613 ± 0.333</b>	<b>2.482 ± 0.384</b>	0.288 ± 0.055
C-GNS-GD ( $\alpha=0.25$ )	<b>1.770 ± 0.443</b>	0.836 ± 0.402	<b>2.108 ± 0.943</b>	0.240 ± 0.036

Model	Rollout position MSE			
	Rope (1e-1)	Bouncing Balls	Bouncing Rigids (1e-1)	Box Bath (1e-2)
Neural Projections	–	18.881 ± 6.131	–	–
ConstraintMLP-FP	–	9.777 ± 5.367	21.828 ± 109.082	–
ConstraintMLP-GD	–	2.425 ± 0.190	3.332 ± 2.164	–
C-GNS-FP	2.804 ± 3.432	–	–	0.689 ± 0.099
C-GNS-GD $f_C(Y)$	3.427 ± 0.400	1.221 ± 0.350	0.818 ± 0.229	2.596 ± 1.189
Forward GNN	18.305 ± 2.340	0.389 ± 0.022	1.174 ± 0.486	0.756 ± 0.089
Iterative GNN	0.546 ± 0.026	0.445 ± 0.044	<b>0.306 ± 0.253</b>	<b>0.609 ± 0.015</b>
<b>C-GNS-GD</b>	0.602 ± 0.131	<b>0.308 ± 0.142</b>	0.374 ± 0.171	0.654 ± 0.002
C-GNS-GD ( $\alpha=0.25$ )	<b>0.460 ± 0.053</b>	<b>0.335 ± 0.037</b>	0.429 ± 0.249	0.660 ± 0.011

## Constraint-based Graph Network Simulator

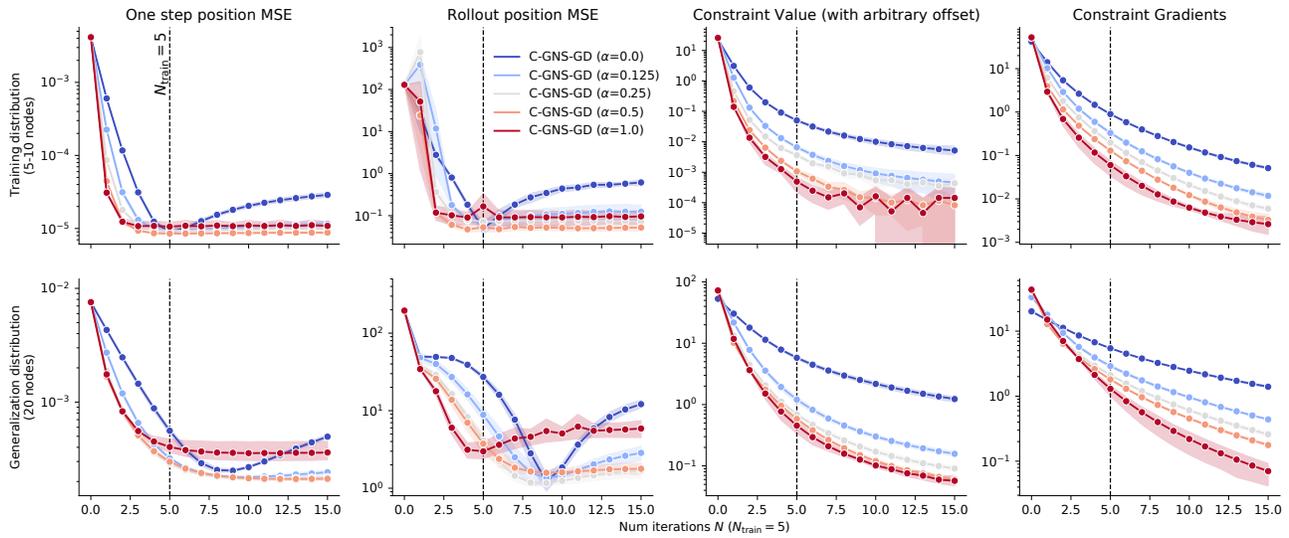
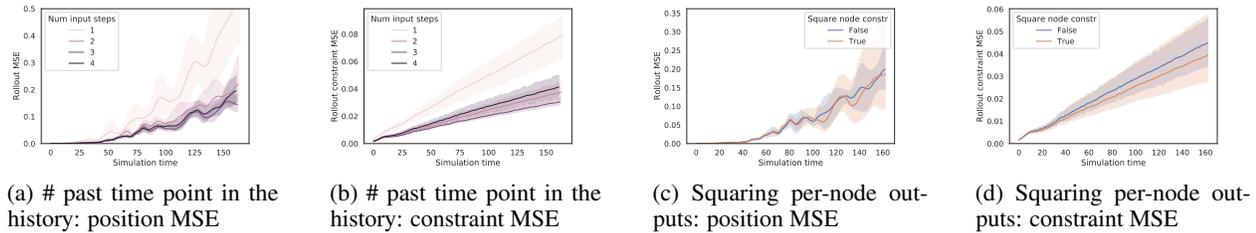


Figure B.9. Using more solver iterations at test time  $N$  as function of  $\alpha$  on the ROPE dataset. Here we loosely label C-GNS ( $\alpha = 0.0$ ) to be the model with the loss on the last iteration only. Increasing  $\alpha$  leads to faster convergence of the solver. With  $\alpha = 1$  the model has worse one-step and rollout MSE on the generalization task. Therefore, we chose to use  $\alpha = 0.25$  for our generalization experiments.



(a) # past time point in the history: position MSE

(b) # past time point in the history: constraint MSE

(c) Squaring per-node outputs: position MSE

(d) Squaring per-node outputs: constraint MSE

Figure B.10. Ablations of the modelling choices on the ROPE dataset.