Efficient Continuous Spatio-Temporal Simulation with Graph Spline Networks

Chuanbo Hua [*1] **Federico Berto** [*1]
Stefano Massaroli [2] **Michael Poli** [3] **Jinkyoo Park** [1]

Abstract

Complex simulation of physical systems is an invaluable tool for a large number of fields, including engineering and scientific computing. To overcome the computational requirements of high–accuracy solvers, learned graph neural network simulators have recently been introduced. However, these methods often require a large number of nodes and edges, which can hinder their performance. Moreover, they cannot evaluate continuous solutions in space and time due to their inherently discretized structure. In this paper, we propose GRAPHSPLINENETS, a method based on graph neural networks and *orthogonal spline collocation* (OSC) to accelerate learned simulations of physical systems by interpolating solutions of graph neural networks. First, we employ an encoder-decoder message passing graph neural network to map the location and value of nodes from the physical domain to hidden space and learn to predict future values. Then, to realize fully continuous simulations over the domain without dense sampling of nodes, we post–process predictions with OSC. This strategy allows us to produce a solution at any location in space and time without explicit prior knowledge of underlying differential equations and with a lower computational burden compared to learned graph simulators evaluating more space–time locations. We evaluate the performance of our approach in heat equation, dam breaking, and flag simulations with different graph neural network baselines. Our method shows is consistently Pareto efficient in terms of simulation accuracy and inference time, i.e. $3\times$ speedup with 10% less error on flag simulation.

[1]KAIST, Daejeon, South Korea [2]The University of Tokyo, Tokyo, Japan [3]Stanford University, San Francisco, California, USA. Correspondence to: Jinkyoo Park <jinkyoo.park@kaist.ac.kr>.

2nd AI4Science Workshop at the 39^{th} International Conference on Machine Learning (ICML), 2022.

1. Introduction

Simulations of *partial differential equations* (PDEs) describing physical processes are an invaluable tool for an increasing number of disciplines. As a result, the scientific machine learning community has been focused on crafting computationally inexpensive yet accurate simulation methods to expand the range of applicability of dynamical system simulators. Traditional simulation methods (Houska et al., 2012), such as the first principle model solver and the generalized *Gauss-Newton methods*, can be costly in calculations: in particular, complex physical simulations need substantial computational resources to be performed. In recent years, PDEs simulators have been widely used in a variety of applied problems such as game physics engines (Lewin, 2021), *Virtual Reality* (VR), (Höll et al., 2018) and the *metaverse* (Taheri et al., 2021). Thus, the development of accurate and fast simulators becomes fundamental to the deployment of such new technologies.

Previous research has shown successes in applying deep learning for simulating a variety of PDEs (Raissi et al., 2019). Graph-based simulation methods (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021) have used graph neural networks as a natural representation of the simulation underlying discretized dynamics; their discretization is often a fundamental part of making simulations viable. These paradigms have proven successful in learning generalizable particle and mesh–based simulators thanks to the ability of graphs in capturing local and global phenomena while retaining properties such as spatial equivariance and translational invariance (Bronstein et al., 2021).

Despite their advantages, a significant downside of mesh-based graph models is their inherent structural discretization, making it hard to achieve physical space or time–space continuous simulations. The encoder–decoder strategy has hidden layers representing physical states out of mesh points, enabling a space-continuous representation (Alet et al., 2019). In addition, *message passing neural network* (MPNN) of derivatives, combined with numerical accumulation methods, work for time-continuous prediction (Iakovlev et al., 2020). However, these continuous approaches require more mesh points for higher–accuracy simulations, which yield a heavy computational burden.

On the other hand, collocation methods have considerable benefits in terms of the computational complexity of the number of collocation points and the ability to yield continuous results, such as the smoothest spline collocation, modified spline collocation (Fairweather & Meade, 2020), and the *Cubic Spline Orthogonal Spline Collocation* (OSC) methods for PDEs (Bialecki & Fairweather, 2001).

In this paper, we introduce GRAPHSPLINENETS: by leveraging the OSC method, we can readily obtain space–time continuous, free form simulations by starting from an intrinsically discrete GNN without the need for explicit prior knowledge of the problem to be solved. We utilize the method as an efficient post–processing scheme that can be applied to several different GNN simulators. GRAPHSPLINENETS allow us to reduce the number of space and time sample points of the *ad–hoc* trained underlying graph module that leads to Pareto–efficient simulations in terms of solution accuracy and inference speed. We demonstrate the method on the heat equation PDE benchmark, on a dam–breaking particle–based meshless simulation, and on a mesh–based cloth simulation.

2. Related Work

We identify related works for this paper in the area of numerical methods and deep learning for PDE simulation and categorize them into three main areas, namely: deep learning for physical simulations, graph neural network simulators, and the relationship between collocation methods and deep learning.

Deep Learning for Physical Simulations Solving differential equations with deep neural networks has been an active research area to solve the issues of traditional PDEs solvers, which often suffer from unsustainable computational requirements and scalability issues that can hinder real–time applications (see Appendix C.1 for further insights). Deep neural networks have been shown to be a viable alternative to numerical methods to solve the issues of scalability and inference time requirements. *Physics Informed Neural Networks* (PINNs) (Raissi et al., 2019), which aid in both the solution and discovery of PDEs by using ad–hoc deep architectures and loss functions enforcing boundary conditions, have received considerable attention due to their flexibility in tackling a wide range of data–driven solutions and discovery of PDE, even though they have been shown suffer from unstable training and convergence issues (Wang et al., 2022). The problem of *solving* PDEs, which we deal with in this paper, has also been explored with convolutional neural networks (Guo et al., 2016; Bhatnagar et al., 2019) which naturally incorporate the inductive bias of spatial invariance. Another active area of research concerns the use of *neural operators* (Lu et al., 2019; Li et al., 2020a; Kovachki et al., 2021) which map between infinite–dimensional function spaces. Several software libraries that have been developed to deal with numerical methods for deep learning efficiently include Poli et al. (2020); Chen et al. (2020); Lu et al. (2021).

Graph Neural Network Simulators The use of *graph neural networks* (GNNs) to address the simulation of a system with a finite number of sample points has been investigated to address issues from other deep learning paradigms for simulation. GNNs extend other models as convolutional neural networks to irregular grids and also capture physical principles deriving from geometric deep learning such as spatial equivariance and permutation (Bronstein et al., 2021) while constraining interactions to local neighborhoods (see Appendix C.2 for further intuitions on the success of GNNs for simulation). Alet et al. (2019) introduces the *Graph Element Networks* architecture to model continuous underlying physical processes with no a–priori graph structure by modeling adaptively sampled points in a graph. Sanchez-Gonzalez et al. (2020) develops with the *Graph-based Neural Simulator* (GNS) paradigm a model that learns a system dynamic update by creating graph edges on neighbor finite particles and performing message passing: this is shown to faithfully contain rollout errors and generalize well to unseen conditions. Pfaff et al. (2021) extends the mesh–free GNS to mesh–based simulations: the resulting model can capture mesh–space interactions by having edges corresponding to the ones of the mesh and obtains cheaper simulations than the baseline numerical solvers. Other related works include applications to control (Sanchez-Gonzalez et al., 2018), the extension of neural operators to graphs (Li et al., 2020c;b), and hybrid approaches with graph networks with traditional fluid simulation solvers (de Avila Belbute-Peres et al., 2020).

Collocation Methods and Deep Learning While continuous–time graph models that have been previously explored (Poli et al., 2019; Xhonneux et al., 2020) can theoretically capture a system's time evolution, space–continuous graph models still suffer from the problem of the inherent graph discretization. For this reason, representing the space and time–continuous nature of simulations has been mainly dealt by using interpolation methods such as linear interpolation in Alet et al. (2019). However, these methods may not be suitable for simulation due to their lack of *differentiability* that may be necessary for instance in control problems (Liang et al., 2019) as well as falling short of realistic simulation which is often continuously differentiable (i.d. $\mathcal{C}^1$ class). Unlike previous approaches, we employ the *Orthogonal Spline Collocation* (OSC) (Bialecki & Fairweather, 2001) method to efficiently obtain $\mathcal{C}^1$ class solutions to differential equations given few partition points and obtain both space and time continuous simulation based on an un-

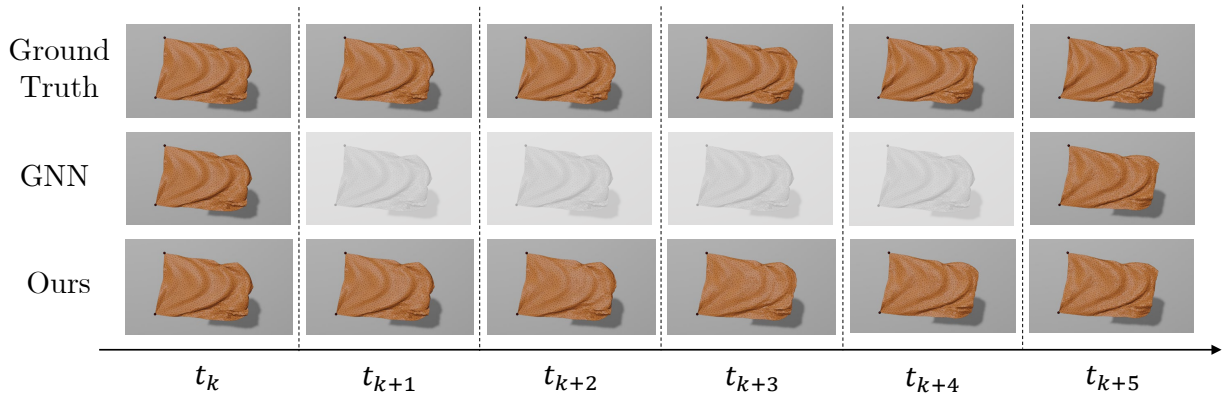

Figure 1. Rollout comparison between the ground truth and our method. GNN is the baseline model (MeshGraphNet) of our GRAPHSPLINENETS, which cannot directly predict intermediate states. By applying the time–oriented OSC method, we obtain realistic time–continuous simulations while at the same time speeding up the learned simulator by not needing additional graph neural network evaluations in time.

derlying GNN simulator. Other deep collocation methods include Guo et al. (2019), which shows an ad–hoc collocation method for the bending analysis of the Kirchoff plate which cannot be easily tackled with mesh–based methods since it requires $\mathcal{C}^1$ continuity. Brink et al. (2021) introduces a deep–learning model based on feed–forward networks and collocation method to approximate a variety of strong–form PDEs. Unlike the paradigms mentioned above that rely on deep learning to obtain collocation weights, we employ the OSC method to obtain the weights which have theoretical guarantees on convergence. Moreover, the synergy of OSC with GNNs enables our module to tackle diverse problems without the need of crafting over–engineered schemes while efficiently balancing between solution accuracy and inference time.

3. Methodology

3.1. Problem Set

We consider a continuous dynamic PDE system with state $u(\mathbf{x}, t) \in \mathbb{R}$ that evolves over time $t \in \mathbb{R}_+$ and bounded domain $\mathbf{x} \in \Omega \subset \mathbb{R}^D$

$$\begin{cases} \mathcal{L}(u) = f(\mathbf{x}, t), (\mathbf{x}, t) \in \Omega \times \mathbb{R}_+ \\ \mathcal{B}(u) = g(\mathbf{x}, t), (\mathbf{x}, t) \in \partial\Omega \times \mathbb{R}_+ \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}), \mathbf{x} \in \Omega \end{cases} \tag{1}$$

where $\mathcal{B}(\cdot)$ is the boundary condition and $u_0(\cdot)$ is the initial condition. We denote $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N\} \in \mathcal{X}$ as the set of physical space locations of sample point. $\mathbf{Y}^t = \{y_0^t, y_1^t, \cdots, y_N^t, \} \in \mathcal{Y}$ is the state of sample points at time t, i.e. $y_i^t = u(\mathbf{x}_i, t)$.

A simulator $\mathcal{S} : \mathcal{Y} \to \mathcal{Y}$ maps the current state of sample points to a future state with fixed timestep. We denote the prediction trajectory from t_0 to t_K as $\{\mathbf{Y}^{t_0}, \hat{\mathbf{Y}}^{t_1}, \cdots, \hat{\mathbf{Y}}^{t_K}\}$. A simulation model $\mathcal{M}(\cdot; \theta) : \mathcal{Y} \to \mathcal{Y}$ with learnable parameter θ takes an input $\mathbf{Y}^{t_k}$ and predicts the next timestep state $\hat{\mathbf{Y}}^{t_{k+1}} = \mathcal{M}(\mathbf{Y}^{t_k}; \theta)$. The gap between prediction states and ground truth can be used as the loss function $\mathcal{L} = \|\hat{\mathbf{Y}}^{t_{k+i}} - \mathbf{Y}^{t_{k+i}}\|^2$.

In the rest of this paper, we will follow the notation convention in which the superscript denotes the time–space index and the subscript denotes the space index.

3.2. Graph OSC Network Architecture

The overall architecture of GRAPHSPLINENETS is shown in Figure 2. Given the initial state of the domain, we firstly employ graph neural networks to obtain discrete predictions. Then, we apply the time–oriented collocation method and space–oriented collocation method on these discrete predictions to get simulation functions, generating time and space–continuous simulations. We describe our model in three parts: graph neural network, time–oriented orthogonal collocation, and space–oriented orthogonal collocation.

Graph neural network structure We employ an Encoder–Processor–Decoder structure to predict sample point values at the next timestep.

The encoder represents sample points in the input space as a node in latent graph space, where an adjacency matrix is created to describe the connection. Symmetrically, the decoder maps the updated graph to output space, representing hidden features to physical space's values for each sample point. A message passing neural network is applied to update node features in hidden graph layers dynamically. The weights of the encoder, decoder, and message passing layers constitute the learning parameters of the graph neural networks, that can be trained by minimizing the end–to–end loss between the simulation results in physical space and the ground truth.

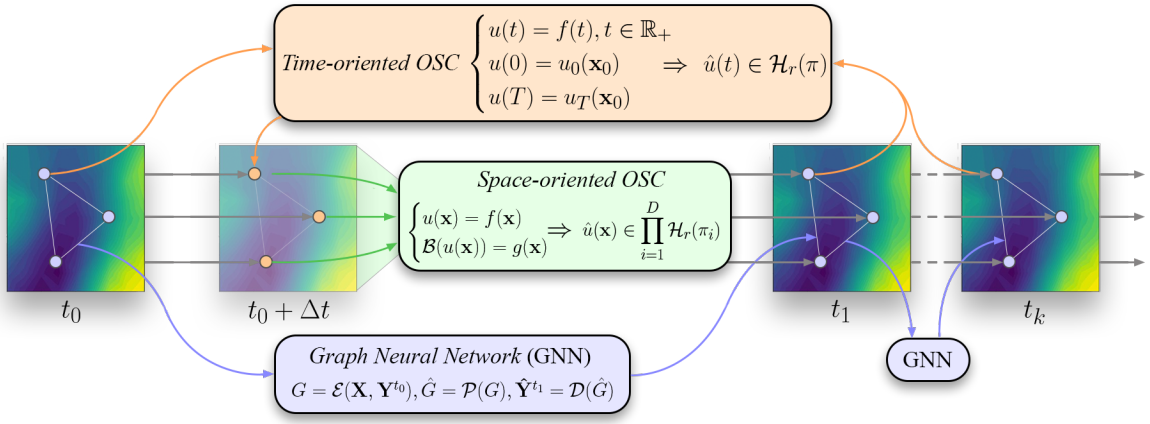

Figure 2. The overall scheme of GRAPHSPLINENETS can be split into three parts. **Graph Neural Network** takes the initial states to make rollout predictions of the value of sample points at several subsequent stages. This process is discrete and the time gap between input and prediction is fixed. **Time–oriented OSC** takes one sample point's states at all prediction steps as input and yields as output a simulation function of this sample point's value over time. By substituting time $t_k + \Delta t$ we can obtain the value of the sample point at this time frame. **Space–oriented OSC** takes the sample points' value at one–time frame as input and yields as output a simulation function of values in the domain. By substituting the location we can get the value of any point within the domain at this time frame. Here $\mathcal{H}_r(\pi_i)$ denotes all C^1 polynomials under order r.

Encoder $\mathcal{E}(\mathbf{X}, \mathbf{Y}^t) : \mathcal{X} \times \mathcal{Y} \to \mathcal{G}$ encodes observed sample points in physical space to a graph $G = (V, E) \in \mathcal{G}$ as a hidden layer with vertices $V = \{\mathbf{v}_i\}_{i=0}^N$ connected by undirected edges $E = \{\mathbf{e}_{ij}\}$, following the strategy of nearest neighbors. Vertical and edge features are assigned during the encoder process, including position, physical value, vertices distances. Details of feature are shown in Section 4.1.

Processor $\mathcal{P}(G) : \mathcal{G} \to \mathcal{G}$ firstly generates messages for every edges based on features of connected vertices, i.e. $\mathbf{m}_{ij} = \phi(\mathbf{v}_i, \mathbf{v}_j, \mathbf{e}_{ij})$. Then, it updates vertices and edges by aggregating messages $\hat{\mathbf{v}}_i = \gamma(\mathbf{v}_i, \bigoplus_{j \in \mathcal{N}(i)} \mathbf{m}_{i,j}), \hat{\mathbf{e}}_{ij} = \psi(\mathbf{e}_{ij}, \mathbf{m}_{ij})$ where $\mathcal{N}(\cdot)$ are connected neighbors, γ and ψ are implemented using *multi–layer perceptrons* (MLP) with a residual connection. Message passing can consist of multiple steps, depending on graph complexity and physical system scale. The final output of the processor is an updated graph $\hat{G}$, which can be considered as the predicted graph at the next timestep. We set the notation of graph at time t_k to G^{t_k}, then the output of processor is $\hat{G}^{t_{k+1}} = \mathcal{P}(G^{t_k})$.

Decoder $\mathcal{D}(G) : \mathcal{G} \to \mathcal{Y}$ extracts updated values for sample points as prediction at the next timestep, i.e. $\hat{\mathbf{Y}}^{t_{k+1}} = \mathcal{D}(\hat{G}^{t_{k+1}})$.

Time-oriented OSC One sample point's value changes from 0 to T over time, following an *Ordinary Differential Equation* (ODE)

$$\begin{cases} u(t) = f(t), t \in [0, T] \\ u(0) = u_0(\mathbf{x}_0) \\ u(T) = u_T(\mathbf{x}_0) \end{cases} \tag{2}$$

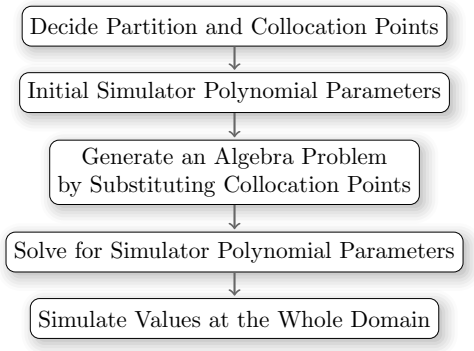

Figure 3. The overall process of applying the OSC method. We firstly choose a rule to generate partition and collocation points, where the rule can be isometric distribution, *Fundamental Solution Method* (Katsurada & Okamoto, 1996) or *Gaussian–Legendre quadrature rule* (De Boor & Swartz, 1973). Then, we define the simulator to be a series of C^1 continuous polynomials. To define the simulator's parameters, we generate an algebra problem by substituting values of collocation points to get the equations. Finally, we solve the algebra problem to get the parameters and then use the simulator to obtain values over the whole domain.

The primary technique for using the OSC approach is given in Figure 3. For time–oriented OSC we consider an isometric split of the temporal domain with N partitions $\pi : 0 = t_0 < t_1 < \cdots < t_N = T$. We aim to find one polynomial under order r on each partition and make these N polynomials C^1 continuous. These polynomials have the degree of freedom $N(r-1)$. To decide those parameters, we select $r-1$ collocation points in each partition to decide those parameters. Note that these collocation points can be

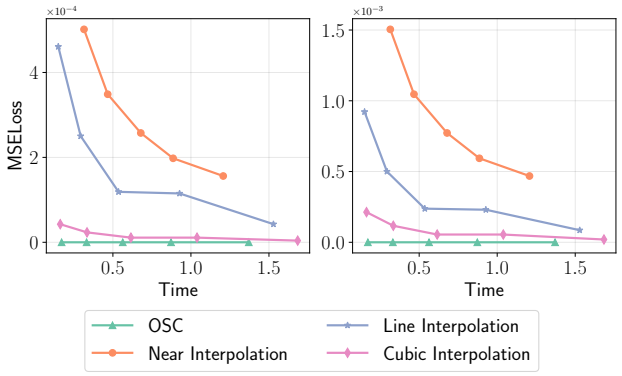

Figure 4. Running time and MSE error of interpolation methods and OSC method on **[Left]** 1-D non-linear PDEs and **[Right]** 2-D non-linear PDEs, changing the number of collocation points from 10 to 20.

isometric or non-isometric: we leave them isometric for the sake of graph neural network prediction.

Then, given an initial value of $t_0 = 0$, the graph neural network predicts in a rollout fashion a series of values at these collocation points. By substituting locations and values of collocation points to polynomials, we can transfer this ODE problem to algebraic equations. Notice that the coefficient matrix of this algebraic equation is *almost block diagonal* (ABD) (De Boor & De Boor, 1978). This kind of system is commonly easy and cheap to be solved (Amodio et al., 2000).

More details about deriving the degree of freedom, visualization of the ABD coefficient matrix, and detailed technique of applying the time–oriented OSC are shown in Appendix A.1.

Space-oriented OSC In one time frame, the state of the domain Ω can be described by

$$\begin{cases} u_x(\mathbf{x}) = f_x(\mathbf{x}), \mathbf{x} \in \Omega \\ \mathcal{B}(u_x(\mathbf{x})) = g_x(\mathbf{x}), \mathbf{x} \in \partial\Omega \end{cases} \tag{3}$$

For simplicity and without loss of generality, we consider the unit domain $[0, 1]^D = \Omega$. Similarly with the time-oriented OSC split strategy, for each dimension, we split the domain into N partitions $\pi_i : 0 = p_0^i < p_1^i < \cdots < p_N^i = 1, i = 1, \cdots, D$. Here the partitions can be isometric or non–isometric. Our target is to find one polynomial under order r on each partition for every dimension and make these $N \times D$ polynomials C^1 continuous in the domain. For example, we can choose the piecewise Hermite cubics as the base. The simulation result is the linear combination of each dimension's basis, with the degree of freedom $N^D(r-1)^D$. For each dimension, we select $r-1$ collocation points in each partition via the Gauss—Legendre quadrature rule[1](De Boor & Swartz, 1973). Thus, we obtain in total $N^D(r-1)^D$ collocation points in the space.

We can get prediction values at collocation points via a graph neural network and the time-oriented OSC. Then, we substitute locations and values to polynomials to transfer the original problem to an algebraic equation. Note that the coefficient matrix of this algebraic equation is also ABD. By solving this algebra equation, we can get the simulation result. Mode details about applying space–oriented OSC are shown in Appendix A.2. With the help of a graph neural network, time–oriented OSC, and space-oriented OSC, simulators can cover all the spatio-temporal domains. To make the model more robust to noisy inputs, we corrupt the input features of the graph with random-walk noise, so the training distribution is similar to the distribution created during rollouts.

3.3. Model Training

After deciding the time–oriented OSC time step length and space–oriented OSC collocation points' locations, we train the graph neural network by supervising on the per-node output features produced by the decoder using a mean square error loss between predictions and their corresponding ground truth.

4. Experiments

We evaluate GRAPHSPLINENETS on three dynamical systems: heat equation, mesh–free compressible fluids, and deformable mesh–based cloth simulation.

4.1. Experimental Domains

Heat Equation These datasets are generated by FEniCS Logg et al. (2012). We set the space domain to $\Omega = [0, 1] \times [0, 1]$ and time domain to $[0, 1]$ with a total of 500 time steps. Each time step corresponds to $\Delta t = 0.1\ s$. At each time step, the dataset has a fixed mesh with 49 nodes and 248 edges. This dataset is split into batches of $500 : 100 : 100$ of train, validation, and test set. We initialize these datasets by fixing four boundaries to 0 and setting one or multiple locations within the domain with the initial temperature.

Dam Breaking Simulation This dataset is generated by Taichi Hu et al. (2019) and simulates a process of 2–dimensional, particle–based water flow with an initialized state of an enclosed rectangular dam. At $t_0 = 0$, one dam wall crashes and lets the water flow down. This dataset has 3200 sample points and 1000 time steps, where each time step corresponds to $\Delta t = 0.01\ s$. This dataset is split into batches of $800 : 100 : 100$ for train, validation and test set.

[1]Widely used quadrature rule that can keep simulation results A-stable (Iserles, 2009)

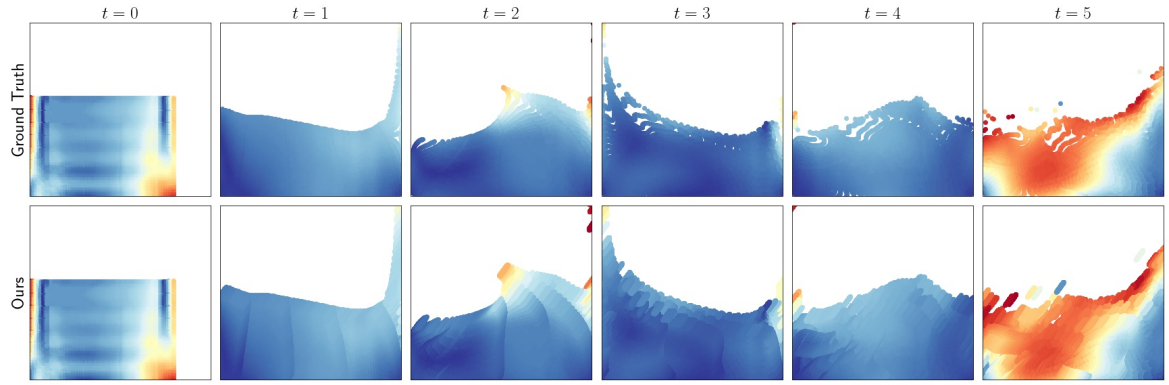

Figure 5. Comparing dam breaking simulation results (down) with the ground truth (up) at $t = 1, 2, \cdots, 5$ time frames. The colors changing from blue to red show the increase of particle position moving gradient. GRAPHSPLINENETS takes the initial state as the input and predicts 40 rollouts, where one rollout maps to 15 frames of ground truth ($\Delta t = 0.15\ s$). Then, the time-oriented OSC is applied to simulate values between rollout steps, including the 5 frames shown here.

Cloth Simulation We utilize the cloth simulation dataset with no adaptive remeshing[2] in Pfaff et al. (2021). We choose this experimental domain to demonstrate how GRAPHSPLINENETS scale to high–dimensional regimes and deal with chaotic systems such as cloth simulation. The dataset is generated by the cloth simulator ArcSim (Narain et al., 2012) with different initialized positions and rotations of a fixed–mesh flag composed of 1579 points. The trajectory in time is simulated in the presence of constant wind and fixed flag handles for a total of 400 timesteps: each timestep corresponds to $\Delta t = 0.02\ s$. The dataset is split into batches of 1000:100:100 of train, validation, and test trajectories.

4.2. Model Training

Heat Equation In the heat equation simulation, the number of nodes and connections does not change along the process and we hence keep the graph structure fixed. The input of the graph model is the state value, i.e. the temperature, of each mesh node at the initial time frame and its output is the mesh state at the subsequent step. We train a message passing neural network model by minimizing the squared difference between the target next state and the model prediction (further details are available in Appendix B.2).

Dam Breaking Simulation The particle–based dam breaking dataset cannot be represented by a fixed mesh due to the chaotic nature of moving particles. We create the graph neural network *iteratively* online as in Pfaff et al. (2021) by searching for the nearest neighbors of each particle in a connectivity radius. Since the number of particles is fixed, the time-oriented OSC can be applied to each particle. The graph neural network takes as input the current positions, node types, and their relative positions as well as a history of past states to infer the state update and is thus trained on the next state target update, i.e. the accelerations (more details are available in Appendix B.3).

[2]Dataset available at: https://github.com/deepmind/deepmind-research/tree/master/meshgraphnets

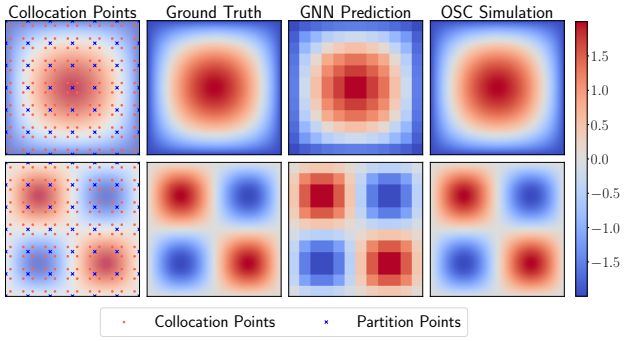

Figure 6. Heat Equation Simulation with central initialization (up) and four corners with dynamically changing boundary condition (down). GNN has discrete prediction result at each step, while with space-oriented OSC, our GRAPHSPLINENETS can have a continuous prediction in the domain.

Cloth Simulation We implement the mesh–based graph simulator in Pfaff et al. (2021) as a baseline model to efficiently deal with the flag mesh–based simulation and produce stable rollouts. Similarly to the dam breaking simulation, we encode the absolute and relative positions and velocities of each node and their types as well as the positions in the mesh–space which yields more stable rollouts. The model is optimized to predict the dynamics state update values.

4.3. Results

OSC and Interpolation Methods We show the effectiveness of orthogonality by testing spline collocation methods with different basis (orthogonal and non orthogonal) on the 2D heat equation simulation and the results are shown in Figure 7 (Left). With an orthogonal basis, the collocation method can get more accurate results with 40% reduction in running time. Moreover, since creating the OSC coefficient matrix involves independent operations, we can use multiprocessing to efficiently create it. Multiprocessing also helps for solving the generated algebraic equations with almost block diagonal matrices and we further our contributions by implementing the algorithm on GPU to increase its efficiency. We show the results of parallelization and GPU implementation in Figure 7.

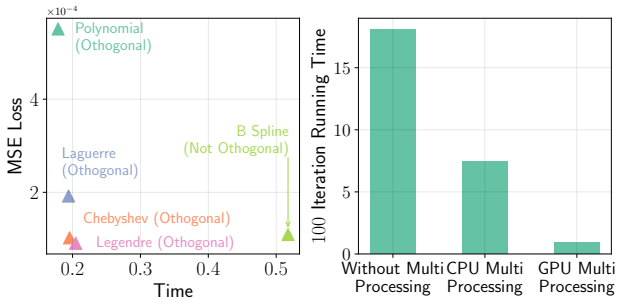

Figure 7. **[Left]** Comparison of spline collocation method with different basis; 1D with 10 collocation points. **[Right]** CPU and GPU multiprocessing comparison of running times; 2D domain with (8×8) collocation points. By efficiently parallelizing OSC, we can speed up its calculations by more than $10\times$.

We also compare the OSC with several widely used interpolation methods: linear, bilinear, 1–D cubic, and 2–D cubic interpolation methods. Note that OSC methods can have customized orders, e.g. the same with cubic interpolation (order 3) or higher (order > 4). Higher–order polynomials can better describe non–linear problems. These methods are applied to four linear and non–linear problems. Results of running time and accuracy comparisons are shown in Figure 4. OSC methods have the lowest error while incurring low computational requirements among the compared methods. The number of sample points affects the performance: however, it is demonstrated that OSC has lower calculation complexity $O(n^2 \log n)$ than the cubic interpolation method $O(n^3/4)$, where n is the number of sample points (Toraichi et al., 1987). Further insights are shown in Appendix B.1.

Heat Equation A visualization of simulation results for the centrally initialized heat equation is shown in Figure 6. GEN has discrete prediction results, while our approach can produce continuous simulations with errors lower than 10^{-7} at any point within the domain. Finally, we discuss the accuracy and running time with rollout steps for two approaches on heat equation by time region and space region.

With the help of time–oriented OSC, GRAPHSPLINENETS have a nearly 90% running time reduction while keeping similar or lower *Mean Squared Error* (MSE) loss compared to GEN as shown in Table 1. Moreover, Figure 9 shows the comparison of the two models' performance with the increase of collocation points and rollout steps. With the increase of collocation point number, our model can capture more information than GEN. Meanwhile, since the GRAPHSPLINENETS can afford a longer graph neural network prediction step, it tends to be more stable for long rollout steps, which is shown in Figure 9.

Dam Breaking We train GRAPHSPLINENETS with the time step $\Delta t = 0.15\ s$. Results comparing our model result with the ground truth are shown in Figure 5. Compared to the baseline GNN, which is trained with the time step $\Delta t = 0.03\ s$, our approach takes 47% less time to execute while the difference of average prediction mean square error with GNN is lower than 10^{-3}. More details about the result are shown in Appendix B.3.

Flag Simulation We train the baseline purely GNN model predicting the state update at each time step of $\Delta t = 0.02\ s$ and two models predicting the update each 5 and 10 simulation time steps, respectively, corresponding to 0.1 and 0.2 seconds. Figure 8 shows rollout errors and our module speedup, while Figure 1 illustrates sample time propagation and comparison with the collocation method. Our method produces stable rollouts even with larger time steps compared to the baseline model while recovering the intermediate time–steps and providing speedups in the range of $1.8\times$ to more than $7\times$ compared to the baseline model.

5. Conclusion

We introduce GRAPHSPLINENETS, a novel method that can be integrated as a post–processing scheme into several learnable GNN modules for improving the solution of a variety of physical processes in terms of both accuracy and inference time. Our approach integrates the theory of Orthogonal Spline Collocation methods to achieve space and time continuous simulations without heavy burdens on the computational side. We demonstrate how GRAPHSPLINENETS are robust in predicting complex, high–dimensional processes characterized by several different PDE processes, such as the ones arising directly from differential equations, particle–based fluid dynamics, and mesh–based deformable cloth simulation. This work represents a significant step forward in learnable simulation approaches and offers key advantages for obtaining solutions to complex systems in science and engineering.

A limitation of our current approach is that collocation is in-

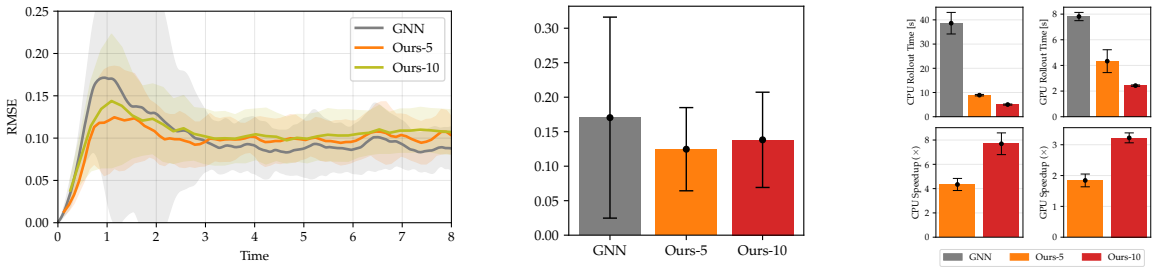

Figure 8. Flag simulation results. **[Left]**: root mean squared error (RMSE) propagation: our modules predict state updates every 5 ($\Delta t = 0.1\ s$ - Ours–5) and 10 ($\Delta t = 0.2\ s$, Ours–10) steps with time–oriented collocation can produce stable rollouts and perform competitively with the baseline model in the long run. **[Middle]** 1–second predictions: around this rollout time, decoherence takes effect due to the chaotic nature of the flag simulation: our method collects lower error compared to the baseline GNN. **[Right]**: CPU and GPU rollout time comparisons accounting for graph inference and the OSC method demonstrate noticeable speedups in terms of solution inference time.

Table 1. Inference times and rollout errors of GRAPHSPLINENETS and GEN.

	Running Time						MSE Loss					
Model	(6 × 6)	(8 × 8)	(10 × 10)	(12 × 12)	(14 × 14)	(16 × 16)	(6 × 6)	(8 × 8)	(10 × 10)	(12 × 12)	(14 × 14)	(16 × 16)
GEN	3.70×10^{-2}	5.81×10^{-2}	6.55×10^{-2}	6.71×10^{-2}	7.00×10^{-2}	7.35×10^{-2}	$\mathbf{2.01 \times 10^{-4}}$	$\mathbf{1.28 \times 10^{-4}}$	8.29×10^{-5}	7.14×10^{-5}	2.14×10^{-6}	1.32×10^{-6}
Ours	$\mathbf{6.59 \times 10^{-4}}$	$\mathbf{1.99 \times 10^{-3}}$	$\mathbf{2.04 \times 10^{-3}}$	$\mathbf{5.88 \times 10^{-3}}$	$\mathbf{6.77 \times 10^{-3}}$	$\mathbf{8.52 \times 10^{-3}}$	5.02×10^{-4}	2.78×10^{-4}	$\mathbf{1.02 \times 10^{-5}}$	$\mathbf{2.13 \times 10^{-5}}$	$\mathbf{1.22 \times 10^{-6}}$	$\mathbf{9.88 \times 10^{-7}}$

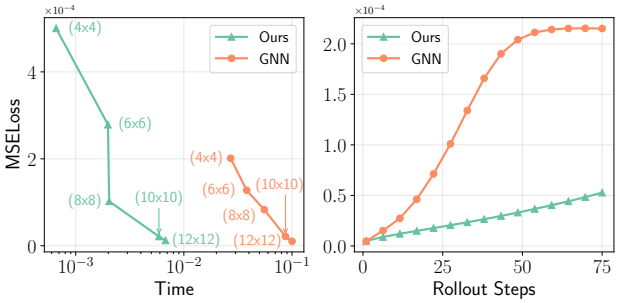

Figure 9. Comparison between our GRAPHSPLINENETS and GEN on the **time** region. **[Left]** comparison of error and time of two models with the number of time–oriented OSC collocation point changing from 4×4 to 12×12. **[Right]** error propagation with rollout steps with 10 (time) and 10×10 (space) collocation points shows how GRAPHSPLINENETS contains the error better during rollout.

troduced as post-processing to obtain continuous solutions; we plan to explore further the possibility of embedding a learnable spline collocation step within the model. Moreover, our work does not consider evolving collocation points, where the location of collocation points is adapted during the rollout steps to allow for a more efficient (in the number of nodes) inference of the solution.

References

Alet, F., Jeewajee, A. K., Villalonga, M. B., Rodriguez, A., Lozano-Perez, T., and Kaelbling, L. Graph element networks: adaptive, structured computation and memory. In *International Conference on Machine Learning*, pp. 212–222. PMLR, 2019.

Amodio, P., Cash, J., Roussos, G., Wright, R., Fairweather, G., Gladwell, I., Kraut, G., and Paprzycki, M. Almost block diagonal linear systems: sequential and parallel solution techniques, and applications. *Numerical linear algebra with applications*, 7(5):275–317, 2000.

Baraff, D. and Witkin, A. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 43–54, 1998.

Bardenhagen, S. G. and Kober, E. M. The generalized interpolation material point method. *Computer Modeling in Engineering and Sciences*, 5(6):477–496, 2004.

Bhatnagar, S., Afshar, Y., Pan, S., Duraisamy, K., and Kaushik, S. Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics*, 64(2):525–545, 2019.

Bialecki, B. and Fairweather, G. Orthogonal spline collocation methods for partial differential equations. *Journal of Computational and Applied Mathematics*, 128(1):55–82, 2001. ISSN 0377-0427. doi: https://doi.org/10.1016/S0377-0427(00)00509-4. URL https://www.sciencedirect.com/science/article/pii/S0377042700005094. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.

Brink, A. R., Najera-Flores, D. A., and Martinez, C. The neural network collocation method for solving partial differential equations. *Neural Computing and Applications*, 33(11):5591–5608, 2021.

Bronstein, M. M., Bruna, J., Cohen, T., and Veličković, P. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

Chen, F., Sondak, D., Protopapas, P., Mattheakis, M., Liu, S., Agarwal, D., and Di Giovanni, M. Neurodiffeq: A python package for solving differential equations with neural networks. *Journal of Open Source Software*, 5(46): 1931, 2020.

de Avila Belbute-Peres, F., Economon, T. D., and Zico Kolter, J. Combining differentiable pde solvers and graph neural networks for fluid flow prediction. *arXiv e-prints*, pp. arXiv–2007, 2020.

De Boor, C. and De Boor, C. *A practical guide to splines*, volume 27. springer-verlag New York, 1978.

De Boor, C. and Swartz, B. Collocation at gaussian points. *SIAM Journal on numerical analysis*, 10(4):582–606, 1973.

Eymard, R., Gallouët, T., and Herbin, R. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

Fairweather, G. and Meade, D. A survey of spline collocation methods for the numerical solution of differential equations. In *Mathematics for large scale computing*, pp. 297–341. CRC Press, 2020.

Guo, H., Zhuang, X., and Rabczuk, T. A deep collocation method for the bending analysis of kirchhoff plate. *Computers, Materials Continua*, 59(2):433–456, 2019. ISSN 1546-2226. doi: 10.32604/cmc.2019.06660. URL http://dx.doi.org/10.32604/cmc.2019.06660.

Guo, X., Li, W., and Iorio, F. Convolutional neural networks for steady flow approximation. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 481–490, 2016.

Höll, M., Oberweger, M., Arth, C., and Lepetit, V. Efficient physics-based implementation for realistic hand-object interaction in virtual reality. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 175–182. IEEE, 2018.

Houska, B., Logist, F., Diehl, M., and Van Impe, J. A tutorial on numerical methods for state and parameter estimation in nonlinear dynamic systems. *Identification for Automotive Systems*, pp. 67–88, 2012.

Hu, Y., Li, T.-M., Anderson, L., Ragan-Kelley, J., and Durand, F. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):1–16, 2019.

Iakovlev, V., Heinonen, M., and Lähdesmäki, H. Learning continuous-time pdes from sparse data with graph neural networks. *arXiv preprint arXiv:2006.08956*, 2020.

Iserles, A. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.

Katsurada, M. and Okamoto, H. The collocation points of the fundamental solution method for the potential problem. *Computers & Mathematics with Applications*, 31(1): 123–137, 1996.

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

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., and Anandkumar, A. Neural operator: Learning maps between function spaces. *arXiv preprint arXiv:2108.08481*, 2021.

Lewin, C. Swish: Neural network cloth simulation on madden nfl 21. pp. 1–2, 07 2021. doi: 10.1145/3450623.3464665.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., and Anandkumar, A. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020a.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., and Anandkumar, A. Multipole graph neural operator for parametric partial differential equations. *arXiv preprint arXiv:2006.09535*, 2020b.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., and Anandkumar, A. Neural operator: Graph kernel network for partial differential equations. *arXiv preprint arXiv:2003.03485*, 2020c.

Liang, J., Lin, M., and Koltun, V. Differentiable cloth simulation for inverse problems. 2019.

Logg, A., Mardal, K.-A., and Wells, G. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

Lu, L., Jin, P., and Karniadakis, G. E. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.

Lu, L., Meng, X., Mao, Z., and Karniadakis, G. E. Deepxde: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

Monaghan, J. J. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574, 1992.

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

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., and Battaglia, P. W. Learning mesh-based simulation with graph networks. *International Conference on Learning Representations*, 2021.

Poli, M., Massaroli, S., Park, J., Yamashita, A., Asama, H., and Park, J. Graph neural ordinary differential equations. *arXiv preprint arXiv:1911.07532*, 2019.

Poli, M., Massaroli, S., Yamashita, A., Asama, H., and Park, J. Torchdyn: A neural differential equations library. *arXiv preprint arXiv:2009.09346*, 2020.

Raissi, M., Perdikaris, P., and Karniadakis, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

Rapaport, D. C. and Rapaport, D. C. R. *The art of molecular dynamics simulation*. Cambridge university press, 2004.

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., Godwin, J., Pfaff, T., Ying, R., Leskovec, J., and Battaglia, P. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pp. 8459–8468. PMLR, 2020.

Taheri, O., Choutas, V., Black, M. J., and Tzionas, D. Goal: Generating 4d whole-body motion for hand-object grasping. *arXiv preprint arXiv:2112.11454*, 2021.

Thuerey, N., Holl, P., Mueller, M., Schnell, P., Trost, F., and Um, K. Physics-based deep learning. *arXiv preprint arXiv:2109.05237*, 2021.

Toraichi, K., Katagishi, K., Sekita, I., and Mori, R. Computational complexity of spline interpolation. *International journal of systems science*, 18(5):945–954, 1987.

Toshev, A., Paehler, L., Panizza, A., and Adams, N. A. On the relationships between graph neural networks for the simulation of physical systems and classical numerical methods. In *ICML 2022 2nd AI for Science Workshop*, 2022.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

Wang, S., Yu, X., and Perdikaris, P. When and why pinns fail to train: A neural tangent kernel perspective. *Journal of Computational Physics*, 449:110768, 2022.

Xhonneux, L.-P., Qu, M., and Tang, J. Continuous graph neural 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. 10432–10441. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/xhonneux20a.html.

Zienkiewicz, O. C., Taylor, R. L., Nithiarasu, P., and Zhu, J. *The finite element method*, volume 3. McGraw-hill London, 1977.

Supplementary Material: Efficient Continuous Spatio-Temporal Simulation with Graph Spline Networks

A. Supplementary Material on Orthogonal Spline Collocation

We further illustrate the OSC method by providing numerical examples in this section.

A.1. 1-D OSC Example

For simplicity and without loss of generality, we consider the function domain as the unit domain $[0, 1]$ and we set $N = 3, r = 2$, which means that we use a three-order three-piece function to simulate the 1-D ODE problem as shown in Equation. 2. We firstly choose the partition points as $x_i, i = 0, \cdots, 3, x_0 = 0, x_3 = 1$. The number of partition points is $N + 1 = 4$. Then, based on Gauss-Legendre quadrature rule, we choose collocation points. The number of collocation points within one partition is $r - 1 = 1$, so we have in total $N \times (r - 1) = 3$ collocation points $\xi_i, i = 0 \cdots, 3$.

After getting partition points and collocation points, we construct the simulator. Here we have three partitions; in each partition, we assign a 2-nd order polynomial

$$a_{0,0} + a_{0,1}x + a_{0,2}x^2, x \in [x_0, x_1] \tag{S1a}$$

$$a_{1,0} + a_{1,1}x + a_{1,2}x^2, x \in [x_1, x_2] \tag{S1b}$$

$$a_{2,0} + a_{2,1}x + a_{2,2}x^2, x \in [x_2, x_3] \tag{S1c}$$

Notice that these three polynomials should be C^1 continuous at the connecting points, i.e. partition points within the domain. For example, Equation S1a and Equation S1b should be continuous at x_1, then we can get two equations

$$\begin{cases} a_{0,0} + a_{0,1}x_1 + a_{0,2}x_1^2 & = a_{1,0} + a_{1,1}x_1 + a_{1,2}x_1^2 \\ 0 + a_{0,1} + 2a_{0,2}x_1 & = 0 + a_{1,1} + 2a_{1,2}x_1 \end{cases} \tag{S2}$$

For the boundary condition

$$\hat{u}(x) = \begin{cases} b_1, x = x_0 \\ b_2, x = x_3 \end{cases} \tag{S3}$$

we can also get two equations

$$\begin{cases} a_{0,0} + 0 + 0 & = b_1 \\ a_{1,0} + a_{1,1} + a_{1,2} & = b_2 \end{cases} \tag{S4}$$

We sum up the equations we got so far. Firstly, our undefined polynomials have $N \times (r + 1) = 9$ parameters. The C^1 continuous condition will create $(N - 1) \times 2 = 4$ equations and the boundary condition will create 2 equations. Then, we obtain $N \times (r - 1)$ collocation points. For each collocation point, we substitute it to polynomials to get an equation. For example, if the ODE in Equation 2 is

$$\hat{u}(x) + \hat{u}'(x) = f(x), x \in [0, 1] \tag{S5}$$

By substituting collocation point ξ_0 into the equation, we can obtain

$$\begin{aligned} & \hat{u}(\xi_0) + \hat{u}'(\xi_0) = f(\xi_0) \\ \Longrightarrow\ & a_{0,0} + a_{0,1}\xi_0 + a_{0,2}\xi_0^2 + a_{0,1} + 2a_{0,2}\xi_0 = f(\xi_0) \\ \Longrightarrow\ & a_{0,0} + a_{0,1}(\xi_0 + 1) + a_{0,2}(\xi_0^2 + 2\xi_0) = f(\xi_0) \end{aligned} \tag{S6}$$

Now we can see that the number of equations is the same as the degree of freedom of polynomials

$$\overbrace{(r+1)\times N}^{\text{Parameters}} = \overbrace{2}^{\text{Boundary}} + \overbrace{(N-1)\times 2}^{C^1\text{Continuous}} + \overbrace{N\times(r-1)}^{\text{Collocation}} \tag{S7}$$

In this example, generated equations are constructed as an algebra problem $\mathbf{Aa} = \mathbf{f}$ where the weight matrix is an *almost block diagonal* (ABD) matrix as shown in Figure S1.

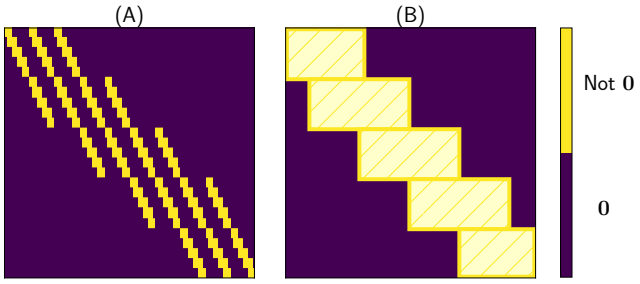

Figure S1. Visualization of an almost block diagonal matrix. Such matrices can be cheaply solved by exploiting their properties.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & \xi_0+1 & \xi_0^2+2\xi_0 & 0 & 0 & 0 \\ 1 & x_1 & x_1^2 & -1 & -x_1 & -x_1^2 \\ 0 & 1 & 2x_1 & 0 & -1 & -2x_1 \\ 0 & 0 & 0 & 1 & \xi_1+1 & \xi_1^2+2\xi_1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}, \tag{S8a}$$

$$\mathbf{a} = \begin{bmatrix} a_{0,0} \\ a_{0,1} \\ a_{0,2} \\ a_{1,0} \\ a_{1,1} \\ a_{1,2} \end{bmatrix}, \mathbf{f} = \begin{bmatrix} b_1 \\ f(\xi_0) \\ 0 \\ 0 \\ f(\xi_1) \\ b_2 \end{bmatrix}. \tag{S8b}$$

A.2. 2–D OSC Example

For simplicity and without loss of generality, we consider the function domain as unit domain $[0,1]\times[0,1]$ and we set $N_x = N_y = 2, r = 3$. The selection of partition points and collocation points is similar to the 1-D OSC method; we have $N^2\times(r-1)^2 = 16$ collocation points in total. For simplicity, we note the partition points in two dimensions to be the same, i.e., $x_i, i = 0,1,2$. Unlike the 1–D OSC method, we choose Hermite bases to describe the simulator, which is kept C^1 continuous. For instance, the base function at point x_1 would be

$$\begin{aligned} H_1(x) &= f_1(x) + g_1(x) \\ f_1(x) &= \begin{cases} \frac{(x-x_0)(x_1-x)^2}{(x_1-x_0)^2}, x\in(x_0,x_1] \\ \frac{(x-x_2)(x-x_1)^2}{(x_2-x_1)^2}, x\in(x_1,x_2] \end{cases} \\ g_1(x) &= \begin{cases} +\frac{[(x_1-x_0)+2(x_1-x_0)](x-x_0)^2}{(x_1-x_0)^3}, x\in(x_0,x_1] \\ +\frac{[(x_2-x_1)+2(x-x_1)](x_2-x)^2}{(x_2-x_1)^3}, x\in(x_1,x_2] \end{cases} \end{aligned} \tag{S9}$$

We separately assign parameters to basis functions, i.e. $H_1(x) = a_{1,i}f_1(x) + b_{1,i}g_1(x)$ for x variable in the $[x_0,x_1]\times[y_{i-1},y_i]$ partition. Then, the polynomial in a partition is the multiple combination of base functions in two dimensions. For example, the polynomial in the partition $[x_0,x_1]\times[y_0,y_1]$ is

$$\begin{aligned} &[a^x_{0,1}f_0(x) + b^x_{0,1}g_0(x) + a^x_{1,1}f_1(x) + b^x_{1,1}g_1(x)] \\ \times&[a^y_{0,1}f_0(y) + b^y_{0,1}g_0(y) + a^y_{1,1}f_1(y) + b^y_{1,1}g_1(y)] \end{aligned} \tag{S10}$$

Now we consider the degree of freedom of these polynomials. By definition, we have $2n(r-1)(n+1) = 24$ parameter. Considering the boundary conditions, we have $24 - 4 \times N = 16$ parameters. The number is equal with collocation points $N^2 \times (r-1)^2$, which means that we can get an algebra equation by substituting collocation points. Solving this equation, we can obtain the simulator parameters.

We can similarly multiply basis functions and set parameters to the simulation result for the higher dimension OSC method. And then select partition points and collocation points by the same strategy with the 2–D OSC method. Then we can generate and solve the algebra equation similarly to what we did for the 1–D case.

A.3. Simple Numerical Example

We set $N = 3, r = 3$ to simulate the problem

$$\begin{cases} u + u' = \sin(2\pi x) + 2\pi\cos(2\pi x) \\ u(0) = 0 \\ u(1) = 0 \end{cases} \tag{S11}$$

we can get a simulation solution

$$\hat{u}(x) = \begin{cases} 6.2x - 0.4x^2 - 31.4x^3, x \in [0, 1/3) \\ 1.5 + 1.6x - 13.8x^2 + 9x^3, x \in [1/3, 2/3) \\ 28.5 - 100x + 108.5x^2 - 37x^3, x \in [2/3, 1] \end{cases} \tag{S12}$$

A visualization of this simulation results is shown in Figure S2.

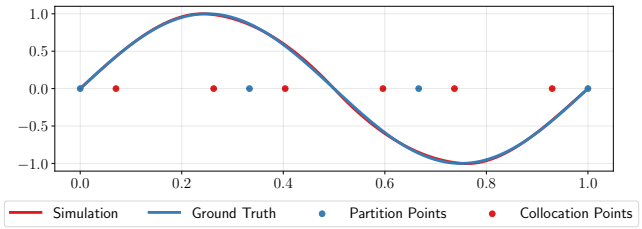

Figure S2. 1-D OSC example problem simulation result visualization, with the mean square loss around 5×10^{-4} compared with the real solution $u(x) = \sin(2\pi x)$.

B. Supplementary Experimental Results

B.1. OSC

We compared the OSC with linear, bilinear, 1–D cubic, and 2–D cubic interpolation methods on four types of problems: 1–D linear, 1–D non-linear, 2–D linear, and 2–D non-linear problems. In these experiments, we test different simulator orders of the OSC method. For example, we set the order of the simulator to 4 for 1–D linear problem and 2 for 2–D linear problem. When the order of the simulator matches the polynomial order of the real solution, OSC can directly find the real solution. For non-linear problems, increasing the order of the simulator would be an ideal way to get a lower error. For example, we set the order of the simulator to 4 for 1–D non-linear problem and 5 for 2–D non-linear problem. Thanks to the efficient calculation of OSC, even though we use higher–order polynomials to simulate, we still employ a lower running time to obtain results compared to other methods.

B.2. Heat Equation

We construct the graph by encoding the sample points' location and value to three features. The connection of nodes is provided by the FEniCS dataset generator, which follows the rule of unit mesh connections. These connections do not change during training. To construct the neural network, we employ 3 layers of message passing neural network; each layer contains 3 MLPs. Before training, we apply as normalization an independent noise $\tilde{N}(0, 10^{-4})$ to each node's value. During

Table 1. Error of OSC and four interpolation methods on different PDE problems: $u(x) = x^4 - 2x^3 + 1.16x^2 - 0.16x$ (1-D linear), $u(x) = \sin(3\pi x)$ (1-D non-linear), $u(x, y) = x^2y^2 - x^2y - xy^2 + xy$ (2-D linear), $u(x, y) = \sin(3\pi x)\sin(3\pi y)$ (2-D non-linear).

MODEL	1-D LINEAR	1-D NON-LINEAR	2-D LINEAR	2-D NON-LINEAR
NEAREST INTERPOLATION	2.3670×10^{-6}	1.7558×10^{-2}	1.9882×10^{-3}	3.8695×10^{-2}
LINEAR INTERPOLATION	1.8928×10^{-7}	8.7731×10^{-4}	3.4317×10^{-4}	1.1934×10^{-2}
QUADRATIC INTERPOLATION	2.6748×10^{-10}	2.8827×10^{-6}	-	-
CUBIC INTERPOLATION	3.5232×10^{-12}	2.2654×10^{-7}	2.9117×10^{-4}	4.5441×10^{-3}
OSC	$\mathbf{3.4153\times10^{-31}}$	$\mathbf{4.1948\times10^{-8}}$	$\mathbf{1.7239\times10^{-32}}$	$\mathbf{3.4462\times10^{-5}}$

training, we load the training trajectories (single frame) randomly to generate target frames nodes value ($\Delta t = 0.01$ for GNN and $\Delta t = 0.05$ for GRAPHSPLINENETS). The loss is the mean square error between the target frames' node value and the ground-truth one. We use the Adam optimizer (Kingma & Ba, 2014) to optimize the loss with a batch size of 2.

We test the performance of GRAPHSPLINENETS and GNN on one heat equation which is centrally initialized, and the four boundaries are fixed to 0 (Figure S3). The ground truth is generated with the resolution 256×256. GRAPHSPLINENETS is trained on 8×8 collocation points with a time step $\Delta t = 0.5\ s$. After we get the prediction values of collocation points, we apply the space–oriented OSC to get simulation polynomials in the domain. Then we can get the same resolution predictions with ground truth. Meanwhile, we train a GNN model on all the ground truth nodes with a time step $\Delta t = 0.1\ s$. We can see from the table that GRAPHSPLINENETS uses less than 10% of GNN's running time while the loss gap is lower than 5×10^{-4}.

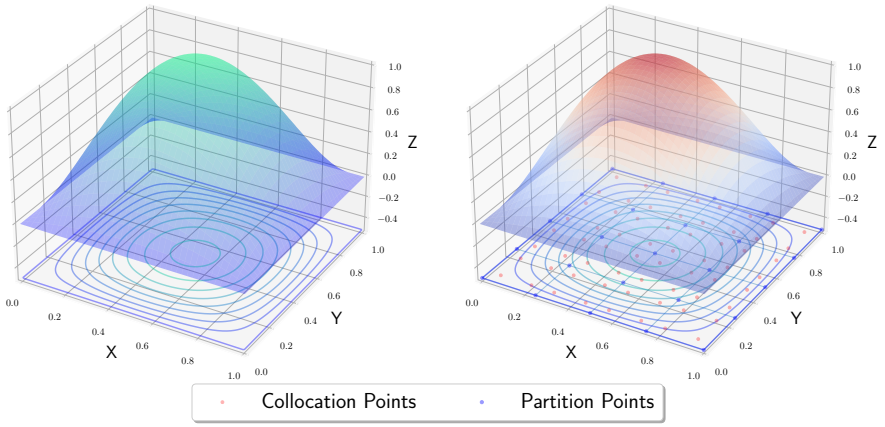

Figure S3. Heat equation with central initialization and zero boundary condition. **[Left]** ground truth solution of this problem. **[Right]** GRAPHSPLINENETS simulation results, with 16 partitions and (8×8) collocation points.

We also test GRAPHSPLINENETS on different heat equations. Figure S4shows one heat equation with zero initialization and dynamically changing boundary conditions. We choose 16 sample points within the domain and get the series of values along time, which is shown in Figure S5. In this experiment, we fix the order of time–oriented OSC to 4. We can see that our model yields accurate results for different locations' sample points. 1–D cubic interpolation method works on less dynamic points. However, for more complex points, the cubic interpolation method collects a considerable amount of error (Figure S5 $(0.9, 0.9)$, while our approach can still converge close to the ground truth.

We then discuss the influence of collocation point numbers. By increasing the number of space–oriented OSC collocation points, we can see the error of GRAPHSPLINENETS gets close results to GNN (Figure S6 left) while keeping the running time shorter. In the case of (14×14) collocation points, GRAPHSPLINENETS has a loss gap lower than 3×10^{-7} with GNN but takes only 43% time of GNN. We also illustrate the error curve of our strategy based on the number of collocation points to demonstrate its strong convergence capabilities.

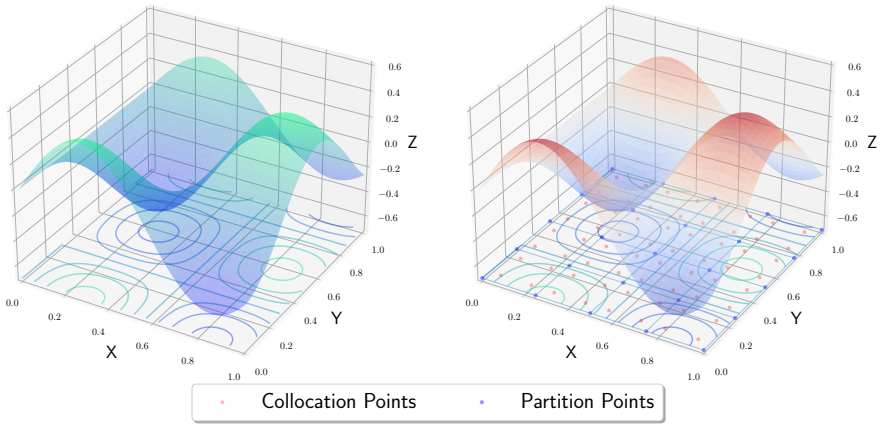

Figure S4. Heat equation with zero initialization, one boundary $(0,0)-(0,1)$ is fixed to 0, two boundaries $(0,0)-(0,1), (0,1)-(1,1)$ are not restricted, and one boundary $(1,0)-(1,1)$ is dynamic by $\sin(t)$.

Table 2. Inference times and rollout errors of our method and GNS on the Dam Breaking dataset with different collocation points for 50 rollout steps. GRAPHSPLINENETS perform competitively or even outperform the baseline model while cutting down considerably the running time.

Model	Running Time			MSELoss		
	15	21	27	15	21	27
GNS	3.44×10^{-1}	4.71×10^{-1}	6.32×10^{-1}	$\mathbf{2.98 \times 10^{-3}}$	$\mathbf{2.28 \times 10^{-3}}$	1.98×10^{-3}
Ours	$\mathbf{4.02 \times 10^{-2}}$	$\mathbf{6.55 \times 10^{-2}}$	$\mathbf{2.91 \times 10^{-1}}$	1.08×10^{-2}	7.78×10^{-3}	$\mathbf{1.62 \times 10^{-3}}$

B.3. Dam Breaking

We define the input *velocity* as average velocity between the current and previous timesteps, calculated for three dimensions $\mathbf{p}_x^{t_k} = (\mathbf{v}_x^{t_k} - \mathbf{v}_x^{t_{k-1}})/\Delta t$. This property is encoded to the node feature. For each node, we find all neighbors within a connectivity radius to connect. We decide the connectivity radius by setting a maximum connectivity k. For each node, we calculate the average distance of its k nearest neighbors, then keep this distance during training. We create the graph structure by using 3 message passing neural network layers with each one containing 3 MLPs.

Before training, we apply independent noise $\mathcal{N}(0, 10^{-3})$ to every node's vector. We only apply one normalization before training. During the training, the graph takes several previous trajectories as input, and the output is be the velocity at this time step. Then, we accumulate this output with time to get the position of this node at the next time frame. We calculate the loss by comparing the position and velocity of each node with the ground truth. We apply stochastic gradient descent via Adam to optimize the loss. Results are shown in Table 2.

B.4. Flag Simulation

We retain most of the experimental settings of the FLAGSIMPLE experiment in (Pfaff et al., 2021). More specifically, we utilize the same latent vector representations of size 128 at each node and edge and the same number of message passing steps (15). We utilize a batch size of 2 instead of 1 as in (Pfaff et al., 2021), which we found to be less brittle to training for fewer epochs. We trained the model for 1M training steps with the Adam optimizer subject to an exponential learning rate decay from 10^{-4} to 10^{-6}. We empirically found the choice of training noise to be essential for successful model training. In particular, we used the same training noise of 1e-3 with the noise correction parameter γ set to 0.1 in the one–step baseline GNN. For models predicting multiple steps, we experimentally found multiplying the step size ratio by the initial training noise to be successful. For instance, supposing that the baseline model predicts with a step update of 0.02 s, the model with step size 5 (i.e. $\Delta t = 0.1$ s) is trained with noise of scale 1e-3 $\times$ 5 = 5e-3. In Figure S7 we show differences between linear

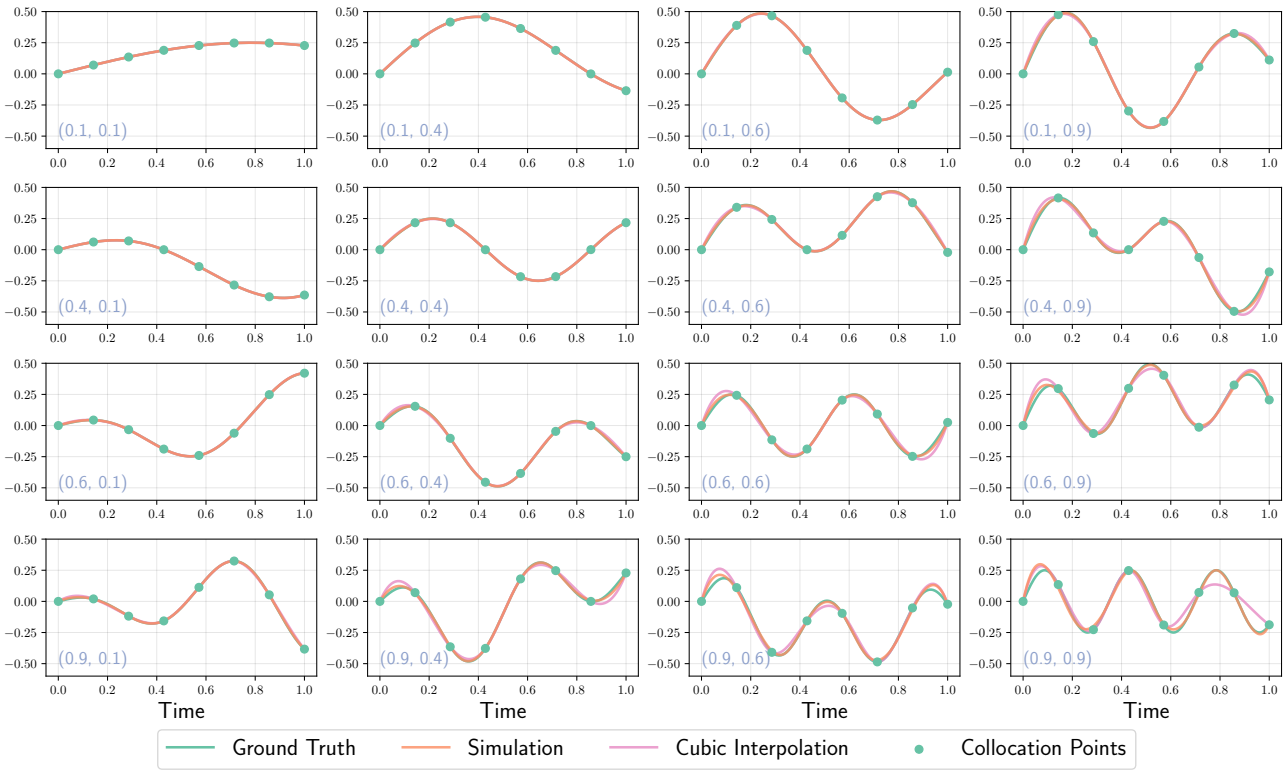

Figure S5. Value changing for 16 sample points along time of the heat equation in Figure S4, including ground truth, 1–D cubic interpolation results and our simulations. Location of sample point is marked in the left bottom corner.

interpolation and OSC for a sample point in the flag simulation.

B.5. Hardware and Software

Experiments were carried out on a machine equipped with an INTEL CORE I9 7900X CPU with 20 threads and a NVIDIA RTX 2080 TI graphic card with 11 GB of VRAM. Software–wise, we used FEniCS (Logg et al., 2012) for Finite Element simulations for the heat equation experiments; Taichi (Hu et al., 2019) for the dam breaking simulation, while the flag dataset was obtained with ArcSim (Narain et al., 2012). We implemented a parallelizable routine for the OSC method using the `multiprocessing` libraries in Python and PyTorch for GPU parallelization (Paszke et al., 2019). The GRAPHSPLINENETS code was written in Python, and PyTorch was used for deep learning while the Deep Graph Library (DGL) (Wang et al., 2020) for graph neural networks.

C. Supplementary Material on Physical Simulations and Deep Learning

C.1. Classical Simulators

Classical numerical methods for solving PDEs can be broadly divided into *mesh–based* and *mesh–free* approaches. Mesh-based methods notably include the Finite Element Method (FEM) (Zienkiewicz et al., 1977), Finite Volume Method, (Eymard et al., 2000) and deformable materials simulators (e.g., cloth) (Baraff & Witkin, 1998; Narain et al., 2012). Their mesh–free counterparts include Molecular Dynamics, (MD) (Rapaport & Rapaport, 2004), the Material Point Method (Bardenhagen & Kober, 2004) for finite particles, and the Smoothed Particle Hydrodynamics (Monaghan, 1992) which specializes in fluid simulation. The choice between mesh–free and mesh–based methods is usually dictated by computational efficiency. For example, while turbulent flow simulations may be represented in a particle-by-particle fashion, describing flows at sample points through a mesh may be desirable, thus noticeably reducing inference times by making meshing more coarse where details do not considerably hinder the simulation. Similarly, domains with dynamically changing volumes, such as fluid simulation, can benefit more from mesh–free particle simulators rather than forcing an unnatural

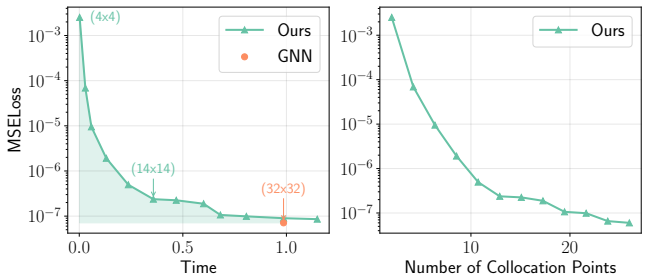

Figure S6. Comparison between GRAPHSPLINENETS and GNN in the **space** region. **[Left]** figure compares the error and time of two models with the number of space–oriented OSC collocation points of GRAPHSPLINENETS setting from 2×2 to 26×26, while the GNN model has the number of space collocation point 32×32. Note that the error of GRAPHSPLINENETS is calculated by firstly applying space–oriented OSC to collocate values at GNN's collocation points and then comparing them with the ground truth. **[Right]** shows the error decreasing with the number of space–oriented OSC collocation points increasing.

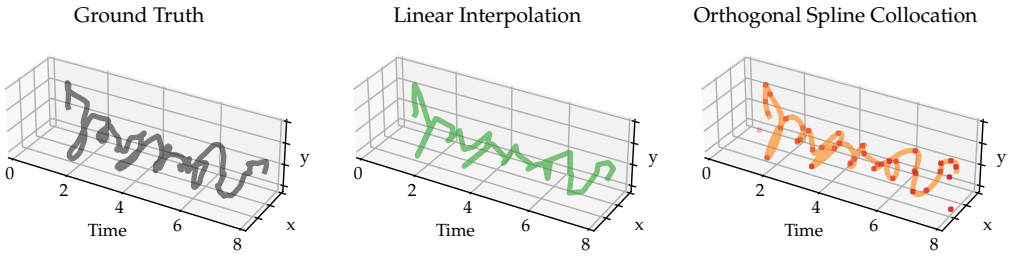

Figure S7. Trajectory in time and space of a single node from the flag simulation experiment. While linear interpolation fails to capture complex behaviors, the OSC method can model intermediate steps with guarantees of solution convergence and lower computational overheads compared to other methods such as cubic interpolation.

mesh descriptor. Major downsides of classical numerical simulators are that they often require extensive knowledge about the simulated domains as well as taking considerable computational resources and time to infer solutions: deep learning has proven to be a valid and faster alternative to existing numerical solvers (Thuerey et al., 2021).

C.2. Further Intuition on Graph Networks Simulators and OSC

Graph neural networks have intrinsically desirable properties for physical simulation. Since they can be used on unstructured grids, they are able to represent complex systems with changing connectivity. Moreover, the graph structurally attains equivariance, while translation invariance can also be easily achieved by considering relativity in nodes' positions rather than their absolute values. Like Molecular Dynamics, our module can construct edges based on each particle neighborhood to capture message passing locality (Toshev et al., 2022). Moreover, similarly to how FEM works, we can consider fixed neighbor connections in mesh space, thus constraining our optimization to a specific geometry. However, unlike the classical FEM and MeshGraphNets, GRAPHSPLINENETS can infer the space continuously without the need to refine the mesh structure. Moreover, given collocation points in time, our model can also obtain time–continuous simulation without the need to constrain the rollouts to an iterative solver.