Physics-Embedded Neural Networks: Graph Neural PDE Solvers with Mixed Boundary Conditions

Masanobu Horie
RICOS Co. Ltd.
University of Tsukuba
horie@ricos.co.jp

Naoto Mitsume
University of Tsukuba
mitsume@kz.tsukuba.ac.jp

Abstract

Graph neural network (GNN) is a promising approach to learning and predicting physical phenomena described in boundary value problems, such as partial differential equations (PDEs) with boundary conditions. However, existing models inadequately treat boundary conditions essential for the reliable prediction of such problems. In addition, because of the locally connected nature of GNNs, it is difficult to accurately predict the state after a long time, where interaction between vertices tends to be global. We present our approach termed physics-embedded neural networks that considers boundary conditions and predicts the state after a long time using an implicit method. It is built based on an $\mathrm{E}(n)$-equivariant GNN, resulting in high generalization performance on various shapes. We demonstrate that our model learns flow phenomena in complex shapes and outperforms a well-optimized classical solver and a state-of-the-art machine learning model in speed-accuracy trade-off. Therefore, our model can be a useful standard for realizing reliable, fast, and accurate GNN-based PDE solvers. The code is available at https://github.com/yellowshippo/penn-neurips2022.

1 Introduction

Partial differential equations (PDEs) are of interest to many scientists because of their application in various fields such as mathematics, physics, and engineering. Numerical analysis is used to solve PDEs because most PDE problems in real life cannot be solved analytically. For example, predicting fluid behavior in complex shapes is an essential topic because it is helpful for product design, disaster reduction, weather forecasting, and many others; however, it is a difficult problem and takes time to solve using classical solvers. Machine learning is a promising approach to predicting such phenomena because it can utilize data similar to the state to be predicted, while classical solvers cannot.

However, the main challenge in dealing with complex phenomena such as fluids is to guarantee generalization performance because possible states in complex systems can be huge and may not be covered using a purely data-driven approach. Therefore, we must apply appropriate inductive biases to machine learning models. Many approaches successfully introduced various inductive biases such as local connectedness using graph neural networks (GNNs) and symmetry under coordinate transformations using equivariance.

While these methods have made great progress in solving PDEs using machine learning, there is still room for improvement. First, there is need for an efficient and provable way to respect boundary conditions like Dirichlet and Neumann, i.e., mixed boundary conditions. Rigorous fulfillment of Dirichlet boundary conditions is indispensable because they are hard constraints and different Dirichlet conditions correspond to different problems users would like to solve. Second, there is need to reinforce the treatment of global interaction to predict the state after a long time, where interactions

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

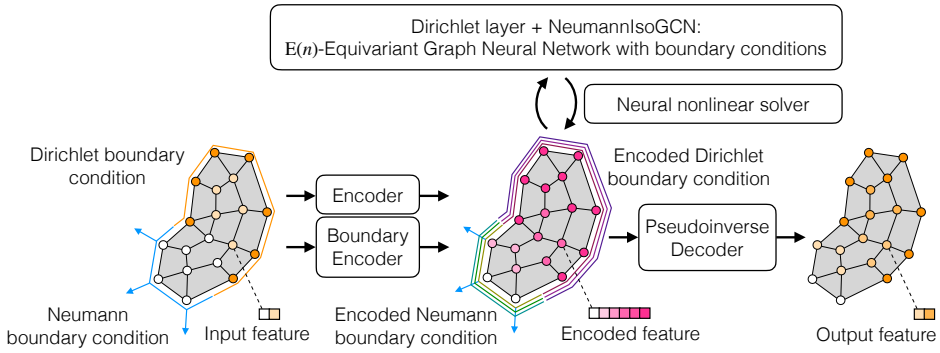

Figure 1: Overview of the proposed method. On decoding input features, we apply boundary encoders to boundary conditions. Thereafter, we apply a nonlinear solver consisting of an $\mathrm{E}(n)$-equivariant graph neural network in the encoded space. Here, we apply encoded boundary conditions for each iteration of the nonlinear solver. After the solver stops, we apply the pseudoinverse decoder to satisfy Dirichlet boundary conditions.

tend to be global. GNNs have excellent generalization properties because of their locally-connected nature; however, they may miss global interaction due to their localness.

We propose physics-embedded neural networks (PENNs), a machine learning framework to address these issues by embedding physics in the models. We build our model based on IsoGCN (Horie et al., 2021), a lightweight $\mathrm{E}(n)$-equivariant GNN to reflect physical symmetry and realize fast prediction. Furthermore, we construct a method to consider mixed boundary conditions. Finally, we reconsider a way to stack GNNs based on a nonlinear solver, which naturally introduces the global pooling to GNNs as the global interaction with high interpretability. In experiments, we demonstrate that our treatment of Neumann boundary conditions improves the predictive performance of the model, and our method can fulfill Dirichlet boundary conditions with no error. Our method also achieves state-of-the-art performance compared to a classical, well-optimized numerical solver and a baseline machine learning model in speed-accuracy trade-off. Figure 1 shows the overview of the proposed model. Our main contributions are summarized as follows:

- We construct models to satisfy mixed boundary conditions: the *boundary encoder*, *Dirichlet layer*, *pseudoinverse decoder*, and *NeumannIsoGCN* (NIsoGCN). The considered models show provable fulfillment of boundary conditions, while existing models cannot.
- We propose *neural nonlinear solvers*, which realize global connections to stably predict the state after a long time.
- We demonstrate that the proposed model shows state-of-the-art performance in speed-accuracy trade-off, and all the proposed components are compatible with $\mathrm{E}(n)$-equivariance.

2 Background and related work

In this section, we review the foundation of PDEs to clarify the problems we solve and introduce related works where machine learning models are used to solve PDEs.

2.1 Partial differential equations (PDEs) with boundary conditions

A general form of the d-dimensional temporal PDEs that we consider can be expressed as follows:

$$\frac{\partial \boldsymbol{u}}{\partial t}(t, \boldsymbol{x}) = \mathcal{D}(\boldsymbol{u})(t, \boldsymbol{x}) \qquad (t, \boldsymbol{x}) \in (0, T) \times \Omega, \tag{1}$$

$$\boldsymbol{u}(t = 0, \boldsymbol{x}) = \hat{\boldsymbol{u}}_0(\boldsymbol{x}) \qquad \boldsymbol{x} \in \Omega, \tag{2}$$

$$\boldsymbol{u}(t, \boldsymbol{x}) = \hat{\boldsymbol{u}}(t, \boldsymbol{x}) \qquad (t, \boldsymbol{x}) \in (0, T) \times \partial\Omega_{\mathrm{Dirichlet}}, \tag{3}$$

$$\hat{\boldsymbol{f}}(\nabla \boldsymbol{u}(t, \boldsymbol{x}), \boldsymbol{n}(\boldsymbol{x})) = \boldsymbol{0} \qquad (t, \boldsymbol{x}) \in (0, T) \times \partial\Omega_{\mathrm{Neumann}}, \tag{4}$$

where Ω is the domain, $\partial\Omega$ is the boundary of Ω, and $\partial\Omega_{\text{Dirichlet}}$ and $\partial\Omega_{\text{Neumann}}$ are boundaries with Dirichlet and Neumann (mixed) boundary conditions. $\hat{\cdot}$ is a known function, and $\mathcal{D}$ is a known nonlinear differential operator, which can be nonlinear and contain spatial differential operators (see equation 18 for an example of $\mathcal{D}$). $\boldsymbol{n}(\boldsymbol{x})$ is the normal vector at $\boldsymbol{x} \in \partial\Omega$. Equation 3 is called the Dirichlet boundary condition, where the value on $\partial\Omega_{\text{Dirichlet}}$ is set as a constraint. Equation 4 corresponds to the Neumann boundary condition, where the value of the derivative $\boldsymbol{u}$ in the direction of $\boldsymbol{n}$ is set on $\partial\Omega_{\text{Neumann}}$ rather than the value of $\boldsymbol{u}$. When $\boldsymbol{u} : (0, T) \times \Omega \to \mathbb{R}^f$ satisfies Equations 1 – 4, it is called the solution of the (initial-) boundary value problem.

2.1.1 Discretization

PDEs are defined in a continuous space to make differentials meaningful. Discretization can be applied in space and time so that computers can solve PDEs easily. In numerical analysis of complex-shaped domains, we commonly use meshes (discretized data of shapes), which can be regarded as graphs. We denote the position of the ith vertex as $\boldsymbol{x}_i$ and the value of a function f at the $\boldsymbol{x}_i$ as f_i.[1]

The simplest method to discretize time is the explicit Euler method formulated as:

$$\boldsymbol{u}(t + \Delta t, \boldsymbol{x}_i) = \boldsymbol{u}(t, \boldsymbol{x}_i) + \mathcal{D}(\boldsymbol{u})(t, \boldsymbol{x}_i)\Delta t, \tag{5}$$

which updates $\boldsymbol{u}(t, \boldsymbol{x}_i)$ with a small increment $\mathcal{D}(\boldsymbol{u})(t, \boldsymbol{x}_i)\Delta t$. Another way to have time discretization is the implicit Euler method formulated as:

$$\boldsymbol{u}(t + \Delta t, \boldsymbol{x}_i) = \boldsymbol{u}(t, \boldsymbol{x}_i) + \mathcal{D}(\boldsymbol{u})(t + \Delta t, \boldsymbol{x}_i)\Delta t, \tag{6}$$

which solves Equation 6 rather than simply updating variables to ensure the original PDE is satisfied numerically. The equation can be viewed as a nonlinear optimization problem by formulating it as:

$$\boldsymbol{R}(\boldsymbol{v}) := \boldsymbol{v} - \boldsymbol{u}(t, \cdot) - \mathcal{D}(\boldsymbol{v})\Delta t, \tag{7}$$

$$\text{Solve}_{\boldsymbol{v}}\ \boldsymbol{R}(\boldsymbol{v})(\boldsymbol{x}_i) = \boldsymbol{0},\ \forall i, \tag{8}$$

where $\boldsymbol{R}(\boldsymbol{v})$ is the residual vector of the discretized PDE. The solution of Equation 8 corresponds to $\boldsymbol{u}(t + \Delta t, \boldsymbol{x})$. By letting $\nabla\phi = \boldsymbol{R}$ for an appropriate ϕ, solving Equation 8 corresponds to optimizing ϕ in an $(f \times n)$-dimensional space, where n is the number of vertices in the considered mesh. A simple way to solve such an optimization problem is to apply gradient descent formulated as:

$$\boldsymbol{v}^{(0)} := \boldsymbol{u}(t, \cdot), \quad \boldsymbol{v}^{(i+1)} := \boldsymbol{v}^{(i)} - \alpha^{(i)}\boldsymbol{R}(\boldsymbol{v}^{(i)}), \tag{9}$$

where $\alpha^{(i)} \in \mathbb{R}$ is determined using line search. However, due to the high computational cost of the search, α can be fixed to a small value, which corresponds to the explicit Euler method with the time step size $\alpha\Delta t$. Barzilai & Borwein (1988) suggested another simple yet effective way to determine the step size using a two-point approximation to the secant equation underlying quasi-Newton methods.

2.2 Neural PDE solvers

We review machine learning models used to solve PDEs called neural PDE solvers, typically formulated as $\boldsymbol{u}(t_{n+1}, \boldsymbol{x}_i) \approx \mathcal{F}_{\text{NN}}(\boldsymbol{u})(t_n, \boldsymbol{x}_i)$ for $(t_n, \boldsymbol{x}_i) \in \{t_0, t_1, \dots\} \times \Omega$, where $\mathcal{F}_{\text{NN}}$ is a machine learning model.

2.2.1 Physics-informed neural networks (PINNs)

Raissi et al. (2019) made a pioneering work combining PDE information and neural networks, called PINNs, by adding loss to monitor how much the output satisfies the equations. PINNs can be used to solve forward and inverse problems and extract physical states from measurements (Pang et al., 2019; Mao et al., 2020; Cai et al., 2021). However, PINNs' outputs should be functions of space because PINNs rely on automatic differentiation to obtain loss regarding PDEs. This design constraint significantly limits the model's generalization ability because the solution of a PDE could be entirely different when the shape of the domain or boundary condition changes. Besides, the loss reflecting PDEs helps models learn physics at training time; however, prediction by PINN models can be out of physics because of lacking PDE information inside the model. Therefore, these methods are not applicable in building models that are generalizable over shape and boundary condition variations. As seen in Section 3, our model contains PDE information inside and does not take absolute positions of vertices, thus resulting in high generalizability (See Figure 3).

[1] Strictly speaking, components of the PDE e.g. $\mathcal{D}$ and Ω can be different before and after discretization. However, we use the same notation regardless of discretization to keep the notation simple.

2.2.2 Graph neural network based PDE solvers

As discussed in Section 2.1.1, one can regard a mesh as a graph. GNNs can take any graphs as inputs (Gori et al., 2005; Scarselli et al., 2008; Kipf & Welling, 2017; Gilmer et al., 2017), having the possibility to generalize over various graphs, i.e., meshes. Therefore, GNNs are strong candidates for learning mesh-structured numerical analysis data, as seen in Alet et al. (2019); Chang & Cheng (2020); Pfaff et al. (2021). Brandstetter et al. (2022) advanced these works for efficient and stable prediction. Their method could also consider boundary conditions by feeding them to the models as inputs. Here, one could expect the model to learn to satisfy boundary conditions approximately, while there is no guarantee to fulfill hard constraints such as Dirichlet conditions. In contrast, our model ensures the satisfaction of boundary conditions. Besides, most GNNs use local connections with a fixed number of message passings, which lacks consideration of global interaction. We suggest an effective way to incorporate a global connection with GNN through the neural nonlinear solver.

2.2.3 Equivariant models

In addition to GNNs, another essential concept to help machine learning models generalize is equivariance. Equivariance is characterized by using group action as $f(g \cdot x) = g \cdot f(x)$ for $f : X \to Y$ and $g \in G$ acting on X and Y. In particular, $\mathrm{E}(n)$-equivariance is essential to predict the solutions of physical PDEs because it describes rigid body motion, i.e., translation, rotation, and reflection. Ling et al. (2016) and Wang et al. (2021) introduced equivariance to a simple neural network and CNN to predict flow phenomena. Both works showed that equivariance improved predictive and generalization performance compared to models without equivariance. Horie et al. (2021) proposed $\mathrm{E}(n)$-equivariant GNNs based on GCNs (Kipf & Welling, 2017), called IsoGCNs. A form of their model is formulated as:

$$[\nabla\psi]_i \approx [\mathrm{IsoGCN}_{0\to1}(\psi)]_i := \left[\sum_{l\in\mathcal{N}_i} \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|}\right]^{-1} \sum_{j\in\mathcal{N}_i} \frac{\psi_j - \psi_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \boldsymbol{W}, \tag{10}$$

where $\mathcal{N}_i$ is the neighborhood of the ith vertex, $\otimes$ is the tensor product operator, and $\boldsymbol{W}$ is a trainable matrix acting on feature index. Here, we denote $\mathrm{IsoGCN}_{0\to1}$ an IsoGCN layer that converts the input scalar (rank-0 tensor) field ψ to the output vector (rank-1 tensor) field. This layer corresponds to the gradient operator, which helps learn PDEs because spatial derivatives such as gradient play an essential role in PDEs. They applied the model to the heat equation problem, showing high predictive performance and fast prediction, while boundary condition treatment was out of their scope.

3 Proposed method

We present our model architecture. We adopt an encode-process-decode architecture, proposed by Battaglia et al. (2018), which has been applied successfully in various previous works, e.g., Horie et al. (2021); Brandstetter et al. (2022). Our key concept is to encode input features, including information on boundary conditions, apply a GNN-based nonlinear solver loop reflecting boundary conditions in the encoded space, then decode carefully to satisfy boundary conditions in the output space.

3.1 Dirichlet boundary model

As demonstrated theoretically and experimentally in literature (Hornik, 1991; Cybenko, 1992; Nakkiran et al., 2021), the expressive power of neural networks comes from encoding in a higher-dimensional space, where the corresponding boundary conditions are not trivial. However, if there are no boundary condition treatments in layers inside the processor, which resides in the encoded space, the trajectory of the solution can be far from the one with boundary conditions. Therefore, boundary condition treatments in an encoded space are essential for obtaining reliable neural PDE solvers that fulfill boundary conditions.

To ensure the same encoded space between variables and boundary conditions, we use the same encoder for variables and the corresponding Dirichlet boundary conditions, which we term the *boundary encoder*, as follows:

$$\boldsymbol{h}_i = \boldsymbol{f}_{\mathrm{encode}}(\boldsymbol{u}_i) \text{ in } \Omega, \quad \hat{\boldsymbol{h}}_i = \boldsymbol{f}_{\mathrm{encode}}(\hat{\boldsymbol{u}}_i) \text{ on } \partial\Omega_{\mathrm{Dirichlet}} \tag{11}$$

One can easily apply Dirichlet boundary conditions in the aforementioned encoded space using the *Dirichlet layer* defined as:

$$\mathrm{DirichletLayer}(\boldsymbol{h}_i) = \begin{cases} \boldsymbol{h}_i, & \boldsymbol{x}_i \notin \partial\Omega_{\mathrm{Dirichlet}} \\ \hat{\boldsymbol{h}}_i, & \boldsymbol{x}_i \in \partial\Omega_{\mathrm{Dirichlet}} \end{cases} \tag{12}$$

This process is necessary to return to the state respecting the boundary conditions after some operations in the processor, which might disrespect the conditions.

After the processor layers, we decode the hidden features using functions satisfying:

$$\boldsymbol{f}_{\mathrm{decode}} \circ \boldsymbol{f}_{\mathrm{encode}}(\hat{\boldsymbol{u}}_i) = \hat{\boldsymbol{u}}_i \text{ on } \partial\Omega_{\mathrm{Dirichlet}} \tag{13}$$

This condition ensures that the encoded boundary conditions correspond to the ones in the original physical space. Demanding that Equation 13 holds for arbitrary $\hat{\boldsymbol{u}}$; we obtain $\boldsymbol{f}_{\mathrm{decode}} \circ \boldsymbol{f}_{\mathrm{encode}} = \mathrm{Id}_{\boldsymbol{u}}$, resulting in $\boldsymbol{f}_{\mathrm{decode}} = \boldsymbol{f}^{+}_{\mathrm{encode}}$, which we call the *pseudoinverse decoder*. It is pseudoinverse because $\boldsymbol{f}_{\mathrm{encode}}$, in particular encoding in a higher-dimensional space, may not be invertible. Therefore, we construct $\boldsymbol{f}^{+}_{\mathrm{encode}}$ using pseudoinverse matrices. For more details, see Appendix A.1.

3.2 Neumann boundary model

Matsunaga et al. (2020) proposed a wall boundary model to deal with Neumann boundary conditions for the least squares moving particle semi-implicit (LSMPS) method (Tamai & Koshizuka, 2014), a framework to solve PDEs using particles. The LSMPS method is the origin of the IsoGCN's gradient operator, so one can imagine that the wall boundary model may introduce a sophisticated treatment of Neumann boundary conditions into IsoGCN. We modified the wall boundary model to adapt to the situation where the vertices are on the Neumann boundary, which differs from the situation of particle simulations (see Appendix A.2 for more details). Our formulation of IsoGCN with Neumann boundary conditions, which is termed *NeumannIsoGCN* (NIsoGCN), is expressed as:

$$\mathrm{NIsoGCN}_{0\to 1}(\psi) := \boldsymbol{M}_i{}^{-1} \left[\sum_{j \in \mathcal{N}_i} \frac{\psi_j - \psi_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} + w_i \boldsymbol{n}_i \hat{g}_i \right] \boldsymbol{W} \tag{14}$$

$$\boldsymbol{M}_i := \sum_{l \in \mathcal{N}_i} \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} + w_i \boldsymbol{n}_i \otimes \boldsymbol{n}_i, \tag{15}$$

where $\hat{g}_i$ is the value of the Neumann boundary condition at $\boldsymbol{x}_i$, $\boldsymbol{W}$ is a trainable matrix, and $w_i > 0$ is an untrainable parameter to control the strength of the Neumann constraint. As $w_i \to \infty$, the model strictly satisfies the given Neumann condition in the direction $\boldsymbol{n}_i$, while the directional derivatives in the direction of $(\boldsymbol{x}_j - \boldsymbol{x}_i)$ tend to be relatively neglected. Thus, we keep the value of w_i moderate to consider derivatives in both $\boldsymbol{n}$ and $\boldsymbol{x}$ directions. In particular, we set $w_i = 10.0$, assuming that around ten vertices may virtually exist "outside" the boundary on a flat surface in a 3D space.

NIsoGCN is a straightforward generalization of the original IsoGCN by letting $\boldsymbol{n}_i = \boldsymbol{0}$ when $\boldsymbol{x}_i \notin \partial\Omega_{\mathrm{Neumann}}$. This model can also be generalized to vectors or higher rank tensors, similarly to the original IsoGCN's construction (see Appendix A.2). Therefore, NIsoGCN can express any spatial differential operator, constituting $\mathcal{D}$ in PDEs.

3.3 Neural nonlinear solver

As reviewed in Section 2.1, one can regard solving PDEs as optimization. Here, we adopt the Barzilai–Borwein method (Barzilai & Borwein, 1988) to solve Equation 8 in the encoded space. In our case, the step size $\alpha^{(i)}$ of gradient descent is approximated as:

$$\alpha^{(i)} \approx \alpha^{(i)}_{\mathrm{BB}} := \frac{\left\langle \boldsymbol{h}^{(i)} - \boldsymbol{h}^{(i-1)}, \boldsymbol{R}(\boldsymbol{h}^{(i)}) - \boldsymbol{R}(\boldsymbol{h}^{(i-1)}) \right\rangle_{\Omega}}{\left\langle \boldsymbol{R}(\boldsymbol{h}^{(i)}) - \boldsymbol{R}(\boldsymbol{h}^{(i-1)}), \boldsymbol{R}(\boldsymbol{h}^{(i)}) - \boldsymbol{R}(\boldsymbol{h}^{(i-1)}) \right\rangle_{\Omega}}, \tag{16}$$

wherer $\boldsymbol{R}(\boldsymbol{h})$ is the residual vector in the encoded space and $\langle \boldsymbol{f}, \boldsymbol{g} \rangle_{\Omega} := \sum_{\boldsymbol{x}_i \in \Omega} \boldsymbol{f}(\boldsymbol{x}_i) \cdot \boldsymbol{g}(\boldsymbol{x}_i)$ denotes the inner product over the mesh. Because the inner product is taken all over the mesh (graph), computing $\alpha^{(i)}_{\mathrm{BB}}$ corresponds to global pooling. With that view, one can find similarities between Equation 9 and deep sets (Zaheer et al., 2017), which is a successful method to learn point cloud data and has a strong background regarding permutation equivariance. For more details, see Appendix A.3.

Table 1: MSE loss ($\pm$ the standard error of the mean) on test dataset of gradient prediction. $\hat{g}_{\mathrm{Neumann}}$ is the loss computed only on the boundary where the Neuman condition is set.

Method	$\nabla\phi(\times 10^{-3})$	$\hat{g}_{\mathrm{Neumann}}(\times 10^{-3})$
Original IsoGCN	192.72 ± 1.69	1390.95 ± 7.93
NIsoGCN (Ours)	6.70 ± 0.15	3.52 ± 0.02

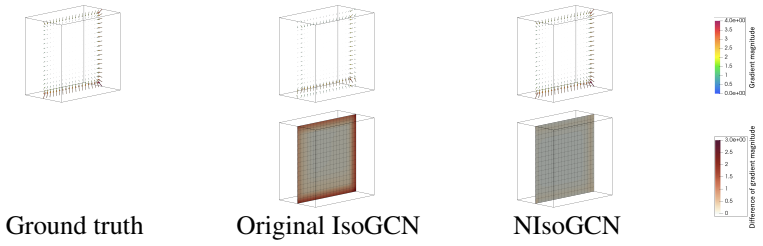

Figure 2: Gradient field (top) and the magnitude of error between the predicted gradient and the ground truth (bottom) of a test data sample, sliced on the center of the mesh.

Our aim is to use Equation 16, approximating the nonlinear differential operator $\mathcal{D}$ in Equation 7 with NIsoGCN. By doing this, we expect the processor to consider both local and global information, which may have an advantage over simply stacking GNNs corresponding to the explicit method as discussed in Section 2.1.1. Combinations of solvers and neural networks are already suggested in, e.g., NeuralODE (Chen et al., 2018). The novelty of our study is the extension of existing methods for solving PDEs with spatial structure and the incorporation of global pooling into the solver, enabling us to capture global interaction, which we refer to as the *neural nonlinear solver*. Finally, the update from the state at the ith iteration $\boldsymbol{h}^{(i)}$ to the $(i+1)$th in the neural nonlinear solver is expressed as:

$$\boldsymbol{h}^{(i+1)} = \mathrm{DirichletLayer}\left(\boldsymbol{h}^{(i)} - \alpha_{\mathrm{BB}}^{(i)}\left[\boldsymbol{h}^{(i)} - \boldsymbol{h}^{(0)} - \mathcal{D}_{\mathrm{NIsoGCN}}(\boldsymbol{h}^{(i)})\Delta t\right]\right), \tag{17}$$

where $\boldsymbol{h}^{(0)}$ is the encoded $\boldsymbol{u}(t,\cdot)$ reflecting Equation 9 and $\mathcal{D}_{\mathrm{NIsoGCN}}$ is an $\mathrm{E}(n)$-equivariant GNN reflecting the structure of $\mathcal{D}$ using differential operators provided by NIsoGCN. Here, Equation 17 enforces hidden features to satisfy the encoded PDE, including boundary conditions, motivating us to call our model *physics-embedded neural networks* because it embeds physics (PDEs) in the model rather than in the loss.

4 Experiments

Using numerical experiments, we demonstrate the proposed model's validity, expressibility, and computational efficiency. We use two types of datasets: 1) the gradient dataset to verify the correctness of NIsoGCN and 2) the incompressible flow dataset to demonstrate the speed and accuracy of the model. The implementation of our model is based on the original IsoGCN's code.[2] Our implementation is available online.[3] The details of the experiments including ablation study and another simple one can be found in Appendix B, C, and D.

4.1 Gradient dataset

As done in Horie et al. (2021), we conducted experiments to predict the gradient field from a given scalar field to verify the expressive power of NIsoGCN. We generated cuboid-shaped meshes randomly with 10 to 20 cells in the X, Y, and Z directions. We then generated random scalar fields over these meshes using polynomials of degree 10 and computed their gradient fields analytically. Our training, validation, and test datasets consisted of 100 samples. Table 1 and Figure 2 show that the proposed NIsoGCN improves gradient prediction, especially near the boundary, showing that our model successfully considers Neumann boundary conditions.

[2] `https://github.com/yellowshippo/isogcn-iclr2021`, Apache License 2.0.
[3] `https://github.com/yellowshippo/penn-neurips2022`, Apache License 2.0.

Table 2: MSE loss ($\pm$ the standard error of the mean) on test dataset of incompressible flow. If "Trans." is "Yes," it means evaluation is done on randomly rotated and transformed test dataset. $\hat{\cdot}_{\text{Dirichlet}}$ is the loss computed only on the boundary where the Dirichlet condition is set for each $\boldsymbol{u}$ and p. MP-PDE's results are based on the time window size equaling 40 as it showed the best performance in the tested MP-PDEs. For complete results, see Table 3.

Method	Trans.	$\boldsymbol{u}$ ($\times 10^{-4}$)	p ($\times 10^{-3}$)	$\hat{\boldsymbol{u}}_{\text{Dirichlet}}$ ($\times 10^{-4}$)	$\hat{p}_{\text{Dirichlet}}$ ($\times 10^{-3}$)
MP-PDE TW = 20	No	$\mathbf{1.30} \pm 0.01$	1.32 ± 0.01	0.45 ± 0.01	0.28 ± 0.02
	Yes	1953.62 ± 7.62	281.86 ± 0.78	924.73 ± 6.14	202.97 ± 3.81
PENN (Ours)	No	4.36 ± 0.03	$\mathbf{1.17} \pm 0.01$	$\mathbf{0.00} \pm 0.00$	$\mathbf{0.00} \pm 0.00$
	Yes	$\mathbf{4.36} \pm 0.03$	$\mathbf{1.17} \pm 0.01$	$\mathbf{0.00} \pm 0.00$	$\mathbf{0.00} \pm 0.00$

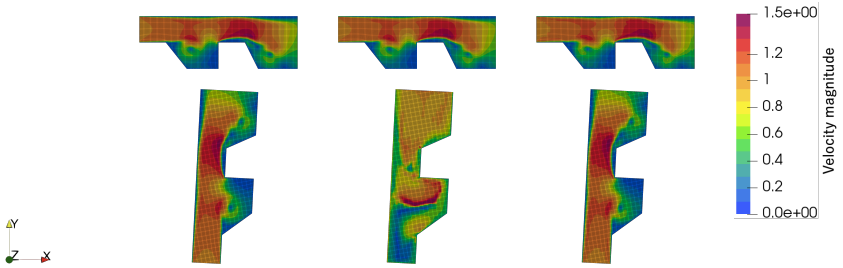

Figure 3: Comparison of the velocity field without (top row) and with (bottom row) random rotation and translation. PENN prediction is consistent under rotation and translation due to the $\mathrm{E}(n)$-equivariance nature of the model, while MP-PDE's predictive performance degrades under transformations.

4.2 Incompressible flow dataset

We tested the expressive power our model by learning incompressible flow in complex shapes. The corresponding nonlinear differential operator is denoted as:

$$\mathcal{D}_{\text{NS}}(\boldsymbol{u}) := -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \frac{1}{\text{Re}} \nabla \cdot \nabla \boldsymbol{u} - \nabla p, \tag{18}$$

with the incompressible condition $\nabla \cdot \boldsymbol{u} = 0$, where, in the present case, $\boldsymbol{u}$ is the flow velocity field, p is the pressure field, and Re is the Reynolds number.

4.2.1 Data

To generate the dataset, we first generated pseudo-2D shapes, with one cell in the Z direction, by changing design parameters, starting from three template shapes. Thereafter, we performed numerical analysis using a classical solver, OpenFOAM,[4] with $\Delta t = 10^{-3}$, and the initial conditions were the solutions of potential flow, which can be computed quickly and stably using the classical solver. The Reynolds number Re was around 10^3. The linear solvers used were generalized geometric-algebraic multi-grid for p and the smooth solver with the Gauss–Siedel smoother for $\boldsymbol{u}$. Template shapes, design parameters, and boundary conditions used can be found in Appendix C.2.

To confirm the expressive power of the proposed model, we used coarse input meshes for machine learning models. We generated these coarse meshes by setting cell sizes roughly four times larger than the original numerical analysis. We obtained ground truth variables using interpolation. The task was to predict flow velocity and pressure fields at $t = 4.0$ using information available before numerical analysis, e.g., initial conditions and the geometries of the meshes. Training, validation, and test datasets consisted of 203, 25, and 25 samples, respectively. We generated the dataset by randomly rotating and translating test samples to monitor the generalization ability of machine learning models.

[4] `https://www.openfoam.com/`

4.2.2 Machine learning models

We constructed the PENN model corresponding to the incompressible Navier–Stokes equation. In particular, we adopted the fractional step method, where the pressure field was also obtained as a PDE solution along with the velocity field. We encoded each feature in a 4, 8, or 16-dimensional space. After features were encoded, we applied a neural nonlinear solver containing NeumanIsoGCNs and Dirichlet layers, reflecting the fractional step method (See Equations 51 and 52). Inside the nonlinear solver's loop, we had a subloop that solved the Poisson equation for pressure, which also reflected the considered PDE (See Equation 50). We looped the solver for pressure five times and four or eight times for velocity. After these loops stopped, we decoded the hidden features to obtain predictions for velocity and pressure, using the corresponding pseudoinverse decoders.

For the state-of-the-art baseline model, we selected MP-PDE (Brandstetter et al., 2022) as it also provides a way to deal with boundary conditions. We used the authors' code[5] with minimum modification to adapt to the task. We tested various time window sizes such as 2, 4, 10, and 20, where one step corresponds to time step size $\Delta t = 0.1$. With changes in time window size, we changed the number of hops considered in one operation of the GNN of the baseline to have almost the same number of hops visible from the model when predicting the state at $t = 4.0$. The numbers of hidden features, 32, 64, and 128, were tested. All models were trained for up to 24 hours using one GPU (NVIDIA A100 for NVLink 40GiB HBM2).

4.2.3 Results

Table 2 and Figure 3 show the comparison between MP-PDE and PENN. The predictive performances of both models are at almost the same level when evaluated on the original test dataset. The results show the great expressive power of the MP-PDE model because we kept most settings at default as much as possible and applied no task-specific tuning. However, when evaluating them on the transformed dataset, the predictive performance of MP-PDE significantly degrades. Nevertheless, PENN shows the same loss value up to the numerical error, confirming our proposed components are compatible with $\mathrm{E}(n)$-equivariance. In addition, PENN exhibits no error on the Dirichlet boundaries, showing that our treatment of Dirichlet boundary conditions is rigorous.

Figure 4 shows the speed-accuracy trade-off for OpenFOAM, MP-PDE, and PENN. We varied mesh cell size, the time step size, linear sover settings for OpenFOAM to have different computation speeds and accuracy. The proposed model achieved the best performance in speed-accuracy trade-off between all the tested methods under fair comparison conditions.

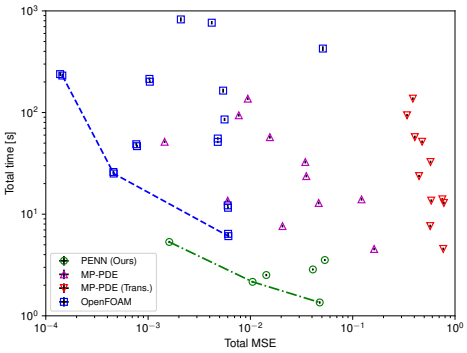

Figure 4: Comparison of computation time and total MSE loss ($\boldsymbol{u}$ and p) on the test dataset (with and without transformation) between OpenFOAM, MP-PDE, and PENN. The error bar represents the standard error of the mean. All computation was done using one core of Intel Xeon CPU E5-2695 v2@2.40GHz. Data used to plot this figure are shown in Tables 4, 5, and 6.

5 Conclusion

We have presented an $\mathrm{E}(n)$-equivariant, GNN-based neural PDE solver, PENN, which can fulfill boundary conditions required for reliable predictions. The model has superiority in embedding the information of PDEs (physics) in the model and speed-accuracy trade-off. Therefore, our model can be a useful standard for realizing reliable, fast, and accurate GNN-based PDE solvers. Although the property of our model is preferable, it also limits the applicable domain of the model because we need to be familiar with the concrete form of the PDE of interest to construct the effective PENN model. Therefore, combining PINNs and PENNs could be the next direction of the research community.

[5] `https://github.com/brandstetter-johannes/MP-Neural-PDE-Solvers`

References

Ferran Alet, Adarsh Keshav Jeewajee, Maria Bauza Villalonga, Alberto Rodriguez, Tomas Lozano-Perez, and Leslie Kaelbling. Graph element networks: adaptive, structured computation and memory. In *ICML*, 2019.

Jonathan Barzilai and Jonathan M Borwein. Two-point step size gradient methods. *IMA journal of numerical analysis*, 8(1):141–148, 1988.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Johannes Brandstetter, Daniel E. Worrall, and Max Welling. Message passing neural PDE solvers. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=vSix3HPYKSU`.

Shengze Cai, Zhicheng Wang, Frederik Fuest, Young Jin Jeon, Callum Gray, and George Em Karniadakis. Flow over an espresso cup: inferring 3-d velocity and pressure fields from tomographic background oriented schlieren via physics-informed neural networks. *Journal of Fluid Mechanics*, 915, 2021.

Kai-Hung Chang and Chin-Yi Cheng. Learning to simulate and design for structural engineering. *arXiv preprint arXiv:2003.09103*, 2020.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control. Signals Syst.*, 5(4): 455, 1992.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272. JMLR. org, 2017.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pp. 729–734. IEEE, 2005.

Deguang Han, Keri Kornelson, Eric Weber, and David Larson. *Frames for undergraduates*, volume 40. American Mathematical Soc., 2007.

Masanobu Horie, Naoki Morita, Toshiaki Hishinuma, Yu Ihara, and Naoto Mitsume. Isometric transformation invariant and equivariant graph convolutional networks. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=FX0vR39SJ5q`.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

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

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL `https://openreview.net/forum?id=SJU4ayYgl`.

Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

Zhiping Mao, Ameya D Jagtap, and George Em Karniadakis. Physics-informed neural networks for high-speed flows. *Computer Methods in Applied Mechanics and Engineering*, 360:112789, 2020.

Takuya Matsunaga, Axel Södersten, Kazuya Shibata, and Seiichi Koshizuka. Improved treatment of wall boundary conditions for a particle method with consistent spatial discretization. *Computer Methods in Applied Mechanics and Engineering*, 358:112624, 2020.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124003, 2021.

Guofei Pang, Lu Lu, and George Em Karniadakis. fpinns: Fractional physics-informed neural networks. *SIAM Journal on Scientific Computing*, 41(4):A2603–A2626, 2019.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=roNqYL0_XP.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. 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.

Siamak Ravanbakhsh, Jeff Schneider, and Barnabás Póczos. Equivariance through parameter-sharing. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 2892–2901. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/ravanbakhsh17a.html.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Tasuku Tamai and Seiichi Koshizuka. Least squares moving particle semi-implicit method. *Computational Particle Mechanics*, 1(3):277–305, 2014.

Rui Wang, Robin Walters, and Rose Yu. Incorporating symmetry into deep dynamics models for improved generalization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=wta_8Hx2KD.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.

A Details of the proposed method

A.1 Construction of pseudoinverse decoder

We can construct the pseudoinverse decoders for a wide range of neural network architectures. For instance, the pseudoinverse decoder for an multilayer perceptron (MLP) with one hidden layer $f(x) = \sigma_2\left(W_2\sigma_1(W_1x+b_1)+b_2\right)$ can be constructed as:

$$f^+(h) = W_1^+\sigma_1^{-1}\left(W_2^+\sigma_2^{-1}(h)-b_2\right)-b_1, \tag{19}$$

where W^+ is the pseudoinverse matrix of W and σ is an invertible activation function whose $\mathrm{Dom}(\sigma) = \mathrm{Im}(\sigma) = \mathbb{R}$. We chose LeakyReLU

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & (x \geq 0) \\ ax & (x < 0), \end{cases} \tag{20}$$

where set $a = 0.5$ because an extreme value of a (e.g., 0.01) could lead to an extreme value of gradient for the inverse function. In addition, one may choose activation functions whose $\mathrm{Im}(\sigma) \neq \mathbb{R}$, such as $\tanh$. However, in that case, we must ensure that the input value to the pseudoinverse decoder is in $\mathrm{Im}(\sigma)$ (in case of $\tanh$, it is $(-1,1)$); otherwise, the computation would be invalid.

Besides, similar to the Dirichlet encoder and pseudoinverse decoder, we could define the specific encoder and decoder for the Neumann boundary condition. However, this is not included in the contributions of our work because it does not improve the performance of our model, which may be because the Neumann boundary condition is a soft constraint in contrast to the Dirichlet one and expressive power seems more important than that inductive bias.

A.2 Derivation of NIsoGCN

Matsunaga et al. (2020) derived a gradient model that can treat the Neumann boundary condition with an arbitrary convergence rate with regard to spatial resolution. Here, we derive our gradient model, i.e., NIsoGCN, in a different way to simplify the discussion because we only need the first-order approximation for fast computation.

Before deriving NIsoGCN, we review introductory linear algebra using simple normation. Using a unit basis $\{\boldsymbol{e}_i \in \mathbb{R}^d : \|\boldsymbol{e}_i\| = 1\}_{i=1}^d$, one can decompose a vector $\boldsymbol{v} \in \mathbb{R}^d$ using:

$$\boldsymbol{v} = \sum_i (\boldsymbol{v}\cdot\boldsymbol{e}_i)\boldsymbol{e}_i. \tag{21}$$

Now, consider replacing the basis $\{\boldsymbol{e}_i \in \mathbb{R}^d\}_{i=1}^d$ with a set of vectors $\boldsymbol{B} = \{\boldsymbol{b}_i \in \mathbb{R}^d\}_{i=1}^{d'}$, called a *frame*, that spans the space but is not necessarily independent (thus, $d' \geq d$). Using the frame, one can assume $\boldsymbol{v}$ is decomposed as:

$$\boldsymbol{v} = \sum_i (\boldsymbol{v}\cdot\boldsymbol{b}_i)\boldsymbol{A}\boldsymbol{b}_i, \tag{22}$$

where $\boldsymbol{A}$ is a matrix that corrects the "overcount" that may occur using the frame (for instance, consider expanding $(1,0)^\top$ with the frame $\{(1,0)^\top, (-1,0)^\top, (0,1)^\top\}$). A set $\{\boldsymbol{A}\boldsymbol{b}_i\}_{i=0}^{d'}$ is called a *dual frame* for $\boldsymbol{B}$. We can find the concrete form of $\boldsymbol{A}$ considering:

$$\boldsymbol{v} = \boldsymbol{A}\sum_i (\boldsymbol{v}\cdot\boldsymbol{b}_i)\boldsymbol{b}_i \tag{23}$$

$$= \boldsymbol{A}\sum_i (\boldsymbol{b}_i\otimes\boldsymbol{b}_i)\boldsymbol{v}. \tag{24}$$

Requiring that Equation 24 holds for any $\boldsymbol{v} \in \mathbb{R}^d$, one can conclude $\boldsymbol{A} = \sum_i (\boldsymbol{b}_i\otimes\boldsymbol{b}_i)^{-1}$. Finally, we obtain

$$\boldsymbol{v} = [\boldsymbol{b}_i\otimes\boldsymbol{b}_i]^{-1}\sum_i (\boldsymbol{v}\cdot\boldsymbol{b}_i)\boldsymbol{b}_i \tag{25}$$

For more details on frames, see, e.g., Han et al. (2007).

Then, we can derive NIsoGCN at the ith vertex on the Neumann boundary, by letting

$$\boldsymbol{B} = \left\{\frac{\boldsymbol{x}_{j_1}-\boldsymbol{x}_i}{\|\boldsymbol{x}_{j_1}-\boldsymbol{x}_i\|}, \frac{\boldsymbol{x}_{j_2}-\boldsymbol{x}_i}{\|\boldsymbol{x}_{j_2}-\boldsymbol{x}_i\|}, \ldots, \frac{\boldsymbol{x}_{j_m}-\boldsymbol{x}_i}{\|\boldsymbol{x}_{j_m}-\boldsymbol{x}_i\|}, \sqrt{w_i}\boldsymbol{n}_i\right\}, \tag{26}$$

where $\{j_1, j_2, \ldots, j_m\}$ are indices of neighboring vertices to the ith vertex. In addition, we assume the approximated gradient of a scalar field ψ at the ith vertex, $\langle\nabla\psi\rangle_i$, satisfies the following conditions:

$$\langle\nabla\psi\rangle_i \cdot \frac{\boldsymbol{x}_{j_k}-\boldsymbol{x}_i}{\|\boldsymbol{x}_{j_k}-\boldsymbol{x}_i\|} = \frac{\psi_{j_k}-\psi_i}{\|\boldsymbol{x}_{j_k}-\boldsymbol{x}_i\|}, \qquad (k = 1, \ldots, m), \tag{27}$$

$$\langle\nabla\psi\rangle_i \cdot \boldsymbol{n} = \hat{g}_i. \tag{28}$$

Equation 27 is a natural assumption because we expect the directional derivative in the direction of $(\boldsymbol{x}_{j_k} - \boldsymbol{x}_i)/\|\boldsymbol{x}_{j_k} - \boldsymbol{x}_i\|$ should correspond to the slope of ψ in the same direction. Equation 28 is the Neumann boundary condition, which we want to satisfy. Finally, by substituting Equations 26, 27, and 28, we obtain NIsoGCN, i.e., Equation 14.

To apply NIsoGCN to $\boldsymbol{t}$, the rank k tensors ($k \geq 1$), one can recursively define the operation as:

$$\mathrm{NIsoGCN}_{k\to k+1}(\boldsymbol{t}) := \begin{pmatrix} \mathrm{NIsoGCN}_{k-1\to k}(\boldsymbol{t}_1) \\ \mathrm{NIsoGCN}_{k-1\to k}(\boldsymbol{t}_2) \\ \mathrm{NIsoGCN}_{k-1\to k}(\boldsymbol{t}_3) \end{pmatrix}, \tag{29}$$

where $\boldsymbol{t}_i$ is the ith component of $\boldsymbol{t}$, resulting in the rank $(k-1)$ tensor. In case of the rank 1 tensor $\boldsymbol{v}$, it can be formulated as:

$$\mathrm{NIsoGCN}_{1\to 2}(\boldsymbol{v}) := \begin{pmatrix} \mathrm{NIsoGCN}_{0\to 1}(v_1) \\ \mathrm{NIsoGCN}_{0\to 1}(v_2) \\ \mathrm{NIsoGCN}_{0\to 1}(v_3) \end{pmatrix} \approx \begin{pmatrix} \partial v_1/\partial x & \partial v_1/\partial y & \partial v_1/\partial z \\ \partial v_2/\partial x & \partial v_2/\partial y & \partial v_2/\partial z \\ \partial v_3/\partial x & \partial v_3/\partial y & \partial v_3/\partial z \end{pmatrix} = \nabla \boldsymbol{v}. \tag{30}$$

Please note that each component v_i has multiple features in the encoded space, e.g., 16 or 64, resulting in $\mathrm{NIsoGCN}_{1\to 2}(\boldsymbol{v})$ represents multiple rank 2 tensors for each vertex (see Figure 1 of Horie et al. (2021)).

As discussed in Horie et al. (2021), IsoGCNs (NIsoGCNs) correspond to spatial differential operators as:

- $\mathrm{NIsoGCN}_{0\to 1}(\psi)$: Gradient $\nabla\psi$ (rank 0 tensor to rank 1 tensor)
- $\mathrm{NIsoGCN}_{1\to 0}(\boldsymbol{v})$: Divergence $\nabla \cdot \boldsymbol{v}$ (rank 1 tensor to rank 0 tensor)
- $\mathrm{NIsoGCN}_{0\to 1\to 0}(\psi) := \mathrm{NIsoGCN}_{1\to 0} \circ \mathrm{NIsoGCN}_{0\to 1}(\psi)$: Laplacian $\nabla \cdot \nabla\psi$ (rank 0 tensor to rank 1 tensor to rank 0 tensor)
- $\mathrm{NIsoGCN}_{1\to 2}(\boldsymbol{v})$: Jacobian $\nabla\boldsymbol{v}$ (rank 1 tensor to rank 2 tensor)
- $\mathrm{NIsoGCN}_{0\to 1\to 2}(\psi) := \mathrm{NIsoGCN}_{l\to 2} \circ \mathrm{NIsoGCN}_{0\to 1}(\psi)$: Hessian $\nabla\nabla\psi$ (rank 0 tensor to rank 1 tensor to rank 2 tensor)

Because NIsoGCN contains a learnable weight matrix (see Equation 14), the component learns to predict the derivative of the corresponding tensor rank in an encoded space. This feature of NIsoGCNs enables us to construct machine learning models corresponding to PDE in the encoded space.

A.3 Derivation of the step size in the Barzilai–Borwein method

We derive Equation 16 by applying the Barzilai–Borwein method to our case. We start with Equation 8, which corresponds to a nonlinear problem:

$$\boldsymbol{R}(\boldsymbol{v}) := \boldsymbol{v} - \boldsymbol{u}(t,\cdot) - \mathcal{D}(\boldsymbol{v})\Delta t, \tag{31}$$

$$\mathrm{Solve}_{\boldsymbol{v}}\ \boldsymbol{R}(\boldsymbol{v})(\boldsymbol{x}_i) = \boldsymbol{0},\ \forall i, \tag{32}$$

We consider solving it by applying the linear iterative method using the Taylor expansion, assuming the update $\Delta\boldsymbol{v}^{(i)} := \boldsymbol{v}^{(i+1)} - \boldsymbol{v}^{(i)}$ is small enough. The iterative method is expressed as:

$$\boldsymbol{v}^{(0)} = \boldsymbol{u}(t,\cdot), \tag{33}$$

$$\boldsymbol{v}^{(i+1)} = \boldsymbol{v}^{(i)} + \Delta\boldsymbol{v}^{(i)}, \tag{34}$$

$$\boldsymbol{R}(\boldsymbol{v}^{(i)} + \Delta\boldsymbol{v}^{(i)}) \approx \boldsymbol{R}(\boldsymbol{v}^{(i)}) + \nabla_{\boldsymbol{v}}\boldsymbol{R}(\boldsymbol{v}^{(i)})\Delta\boldsymbol{v}^{(i)} = \boldsymbol{0}, \tag{35}$$

where $\nabla_{\boldsymbol{v}}\boldsymbol{R}(\boldsymbol{v}^{(i)})$ denotes the Jacobian matrix with the shape of $n \times n$ (n roughly corresponds to the number of vertices of the mesh). To optain update, we may solve Equation 35 as:

$$\Delta\boldsymbol{v}^{(i)} = \left[\nabla_{\boldsymbol{v}}\boldsymbol{R}(\boldsymbol{v}^{(i)})\right]^{-1}\boldsymbol{R}(\boldsymbol{v}^{(i)}), \tag{36}$$

corresponding to the Newton–Raphson method. However, it may take enormous computation resources because $\nabla_{\boldsymbol{v}}\boldsymbol{R}(\boldsymbol{v}^{(i)})$ is usually a huge matrix. Instead, we can approximate:

$$\left[\nabla_{\boldsymbol{v}}\boldsymbol{R}(\boldsymbol{v}^{(i)})\right]^{-1} \approx \alpha^{(i)}, \tag{37}$$

which corresponds to gradient descent:

$$\Delta\boldsymbol{v}^{(i)} \approx \alpha^{(i)}\boldsymbol{R}(\boldsymbol{v}^{(i)}). \tag{38}$$

Substituting Equation 37 into Equation 35, we obtain:

$$\boldsymbol{R}(\boldsymbol{v}^{(i+1)}) \approx \boldsymbol{R}(\boldsymbol{v}^{(i)}) + \frac{1}{\alpha^{(i)}}\Delta\boldsymbol{v}^{(i)}. \tag{39}$$

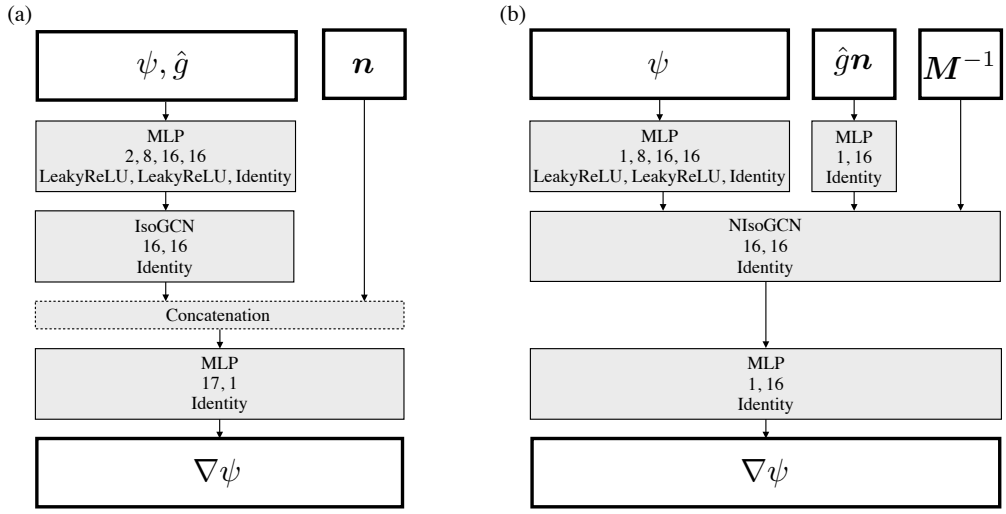

Figure 5: Architecture used for (a) original IsoGCN and (b) NIsoGCN training. In each cell, we put the number of units in each layer along with the activation functions used.

We want to find a good $\alpha^{(i)}$ satisfying Equation 39 the best. Thus, we obtain $\alpha_{\mathrm{BB}}^{(i)}$ through:

$$\alpha_{\mathrm{BB}}^{(i)} = \arg\min_{\alpha} \mathcal{L}(\alpha), \tag{40}$$

$$\mathbb{R} \ni \mathcal{L}(\alpha) := \frac{1}{2} \left\| \Delta \boldsymbol{v}^{(i)} - \alpha \Delta \boldsymbol{R}^{(i)} \right\|^2, \text{ where } \Delta \boldsymbol{R}^{(i)} = \boldsymbol{R}(\boldsymbol{v}^{(i+1)}) - \boldsymbol{R}(\boldsymbol{v}^{(i)}). \tag{41}$$

Because of the convexity of the problem, it is enough to find alpha satisfying:

$$\left. \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\alpha} \right|_{\alpha_{\mathrm{BB}}^{(i)}} = \left\langle \Delta \boldsymbol{v}^{(i)} - \alpha_{\mathrm{BB}}^{(i)} \Delta \boldsymbol{R}^{(i)}, -\Delta \boldsymbol{R}^{(i)} \right\rangle = 0, \tag{42}$$

where $< \cdot, \cdot >$ denotes the inner product in the corresponding space. Using the linearity of the inner product, we obtain:

$$\left\langle \Delta \boldsymbol{v}^{(i)} - \alpha_{\mathrm{BB}}^{(i)} \Delta \boldsymbol{R}^{(i)}, -\Delta \boldsymbol{R}^{(i)} \right\rangle = 0, \tag{43}$$

$$-\left\langle \Delta \boldsymbol{v}^{(i)}, \Delta \boldsymbol{R}^{(i)} \right\rangle + \alpha_{\mathrm{BB}}^{(i)} \left\langle \Delta \boldsymbol{R}^{(i)}, \Delta \boldsymbol{R}^{(i)} \right\rangle = 0, \tag{44}$$

$$\alpha_{\mathrm{BB}}^{(i)} = \frac{\left\langle \Delta \boldsymbol{v}^{(i)}, \Delta \boldsymbol{R}^{(i)} \right\rangle}{\left\langle \Delta \boldsymbol{R}^{(i)}, \Delta \boldsymbol{R}^{(i)} \right\rangle}. \tag{45}$$

Equation 45 is equivalent to Equation 16.

As seen from the derivation, $\alpha_{\mathrm{BB}}^{(i)}$ is determined to satisfy Equation 39 as much as possible for all vertices and all feature components. That means $\alpha_{\mathrm{BB}}^{(i)}$ has global information because it considers all vertices, making the global interaction possible. In addition, $\alpha_{\mathrm{BB}}^{(i)}$ is equivariant because it is scalar, which does not depend on coordinate. Therefore, $\alpha_{\mathrm{BB}}^{(i)}$ is suitable for realizing efficient PDE solvers with $\mathrm{E}(n)$-equivariance.

B Experiment details: gradient dataset

Figure 5 shows the architectures we used for the gradient dataset. The dataset is uploaded online.[6] We followed the instruction of Horie et al. (2021) (in particular, Appendix D.1 of their paper) to make the features and models equivariant. To facilitate a fair comparison, we made input information for both models equivalent, except for $\boldsymbol{M}^{-1}$ in Equation Equation 15, which is a part of our novelty. For both models, we used Adam (Kingma & Ba, 2014) as an optimizer with the default setting. Training for both models took around ten minutes using one GPU (NVIDIA A100 for NVLink 40GiB HBM2). Figure 5 shows model architectures used for the experiment.

[6] `https://drive.google.com/uc?export=download&id=113nzfpwRZSsDjZeNU9kgOj9X7QBSC49Y`

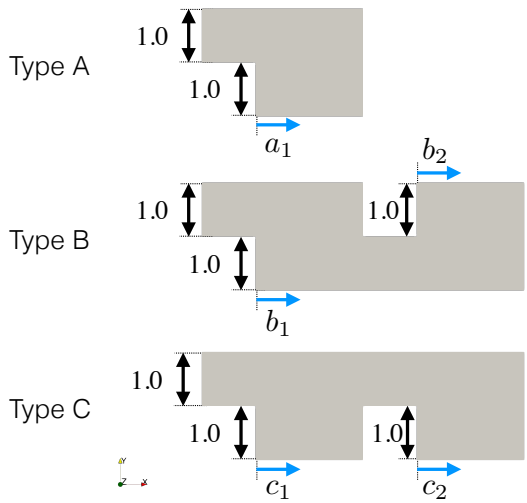

Figure 6: Three template shapes used to generate the dataset. a_1, b_1, b_2, c_1, and c_2 are the design parameters.

C Experiment details: incompressible flow dataset

C.1 Governing equation

The incompressible Navier–Stokes equations, the governing equations of incompressible flow, are expressed as:

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \frac{1}{\mathrm{Re}} \nabla \cdot \nabla \boldsymbol{u} - \nabla p \qquad (t, \boldsymbol{x}) \in (0, T) \times \Omega, \tag{46}$$

$$\boldsymbol{u} = \hat{\boldsymbol{u}} \qquad (t, \boldsymbol{x}) \in \partial\Omega^{(\boldsymbol{u})}_{\mathrm{Dirichlet}}, \tag{47}$$

$$\left[\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right] \boldsymbol{n} = \boldsymbol{0} \qquad (t, \boldsymbol{x}) \in \partial\Omega^{(\boldsymbol{u})}_{\mathrm{Neumann}}. \tag{48}$$

We also consider the following incompressible condition:

$$\nabla \cdot \boldsymbol{u} = 0 \qquad (t, \boldsymbol{x}) \in (0, T) \times \Omega, \tag{49}$$

which may be problematic when solving these equations numerically. Therefore, it is common to divide the equations into two: one to obtain pressure and one to compute velocity. There are many methods to make such a division; for instance, the fractional step method derives the Poisson equation for pressure as follows:

$$\nabla \cdot \nabla p(t + \Delta t, \boldsymbol{x}) = \frac{1}{\Delta t} (\nabla \cdot \tilde{\boldsymbol{u}})(t, \boldsymbol{x}), \tag{50}$$

where

$$\tilde{\boldsymbol{u}} = \boldsymbol{u} - \Delta t \left(\boldsymbol{u} \cdot \nabla \boldsymbol{u} - \frac{1}{\mathrm{Re}} \nabla \cdot \nabla \boldsymbol{u} \right) \tag{51}$$

is called the intermediate velocity. Once we solve the equation, we can compute the time evolution of velocity as follows:

$$\boldsymbol{u}(t + \Delta t, \boldsymbol{x}) = \tilde{\boldsymbol{u}}(t, \boldsymbol{x}) - \Delta t \nabla p(t + \Delta t, \boldsymbol{x}). \tag{52}$$

Because the fractional step method requires solving the Poisson equation for pressure, we also need the boundary conditions for pressure as well:

$$p = 0 \qquad (t, \boldsymbol{x}) \in \partial\Omega^{(p)}_{\mathrm{Dirichlet}}, \tag{53}$$

$$\nabla p \cdot \boldsymbol{n} = 0 \qquad (t, \boldsymbol{x}) \in \partial\Omega^{(p)}_{\mathrm{Neumann}}. \tag{54}$$

Our machine learning task is also based on the same assumption: motivating pressure prediction in addition to velocity with boundary conditions of both.

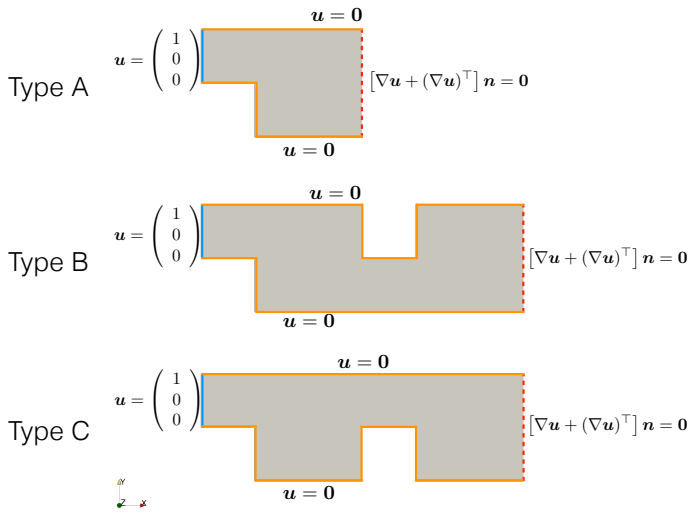

Figure 7: Boundary conditions of $\boldsymbol{u}$ used to generate the dataset. The continuous lines and dotted lines correspond to Dirichlet and Neumann boundaries.

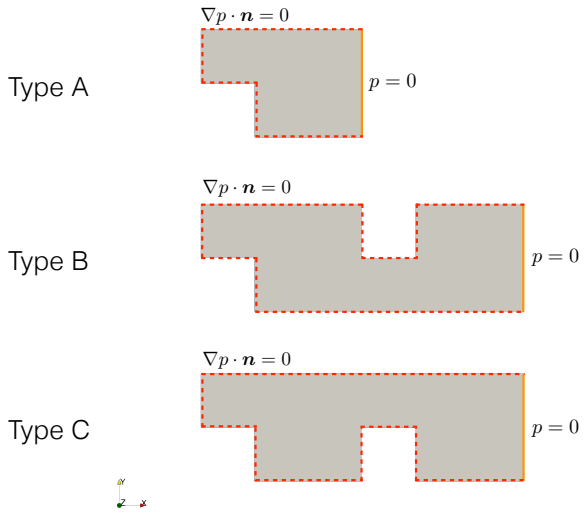

Figure 8: Boundary conditions of p used to generate the dataset. The continuous lines and dotted lines correspond to Dirichlet and Neumann boundaries.

C.2 Dataset

We generated numerical analysis results using various shapes of the computational domain, starting from three template shapes and changing their design parameters as shown in Figure 6. For each design parameter, we varied from 0 to 1.0 with a step size of 0.1, yielding 11 shapes for type A and 121 shapes for type B and C. The boundary conditions were set as shown in Figures 7 and 8. These design and boundary conditions were chosen to have the characteristic length of 1.0 and flow speed of 1.0. The viscosity was set to 10^{-3}, resulting in Reynolds number $\mathrm{Re} \sim 10^3$. The linear solvers used were generalized geometric-algebraic multi-grid for p and the smooth solver with the Gauss–Siedel smoother for $\boldsymbol{u}$. Numerical analysis to generate each sample took up to one hour using CPU one core (Intel Xeon CPU E5-2695 v2@2.40GHz). The dataset is uploaded online.[7]

[7] `https://drive.google.com/uc?export=download&id=1x10-rsRp1XckS-0iXpFCM9l6f2xM0v0I`, `https://drive.google.com/uc?export=download&id=1Q8cC5Lh3LYsesdt6LveFPzBLoLQVuliT`, `https://drive.google.com/uc?export=download&id=15eH9jvUWHvGrWldB0BcwuqRtqR8UnzlS`, `https://drive.google.com/uc?export=download&id=1GdntVtfG6wgU_8FQMXsCmpylhylWCo8B`, and `https://drive.google.com/uc?export=download&id=1aIykQ2Yyp-591GMhIX5dHkt0d6hevtYB`.

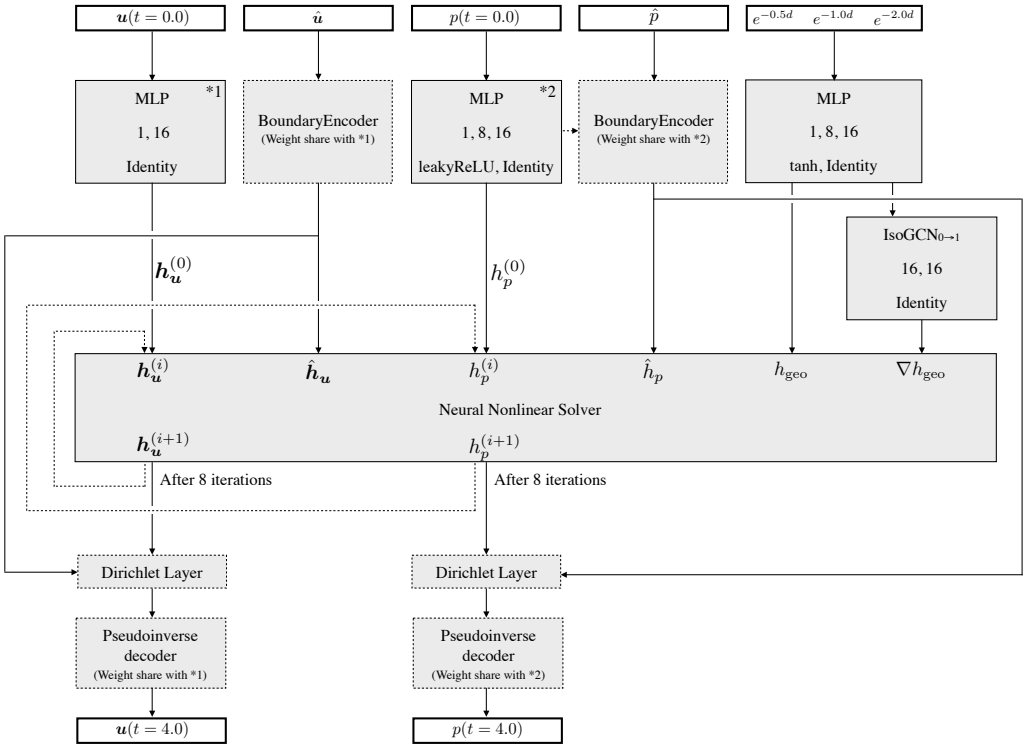

Figure 9: The overview of the PENN architecture for the incompressible flow dataset. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. Arrows with dotted lines correspond to the loop. In each cell, we put the number of units in each layer along with the activation functions used.

C.3 Model architectures

The input features of the model are:

- $\boldsymbol{u}(t=0.0)$: The initial velocity field, the solulsion of potential flow
- $\hat{\boldsymbol{u}}$: The Dirichlet boundary condition for velocity
- $p(t=0.0)$: The initial pressure field
- $\hat{p}$: The Dirichlet boundary condition for pressure
- $e^{-0.5d}, e^{-1.0d}, e^{-2.0d}$: Features computed from d, the distance from the wall boundary condition

and the output features are:

- $\boldsymbol{u}(t=4.0)$: The velocity field at $t=4.0$
- $p(t=4.0)$: The pressure field at $t=4.0$

The strategy to construct PENN for the incompressible flow dataset is the following:

- Consider the encoded version of the governing equation
- Apply the neural nonlinear solver containing the Dirichlet layer and the NIsoGCN to the encoded equation
- Decode the hidden feature using the pseudoinverse decoder.

Reflecting the fractional step method, we build PENN using spatial differential operators provided by NIsoGCN. We use a simple linear encoder for the velocity and the associated Dirichlet boundary conditions. For pressure and its Dirichlet constraint, we use a simple MLP with one hidden layer. We encode each feature in a 16-dimensional

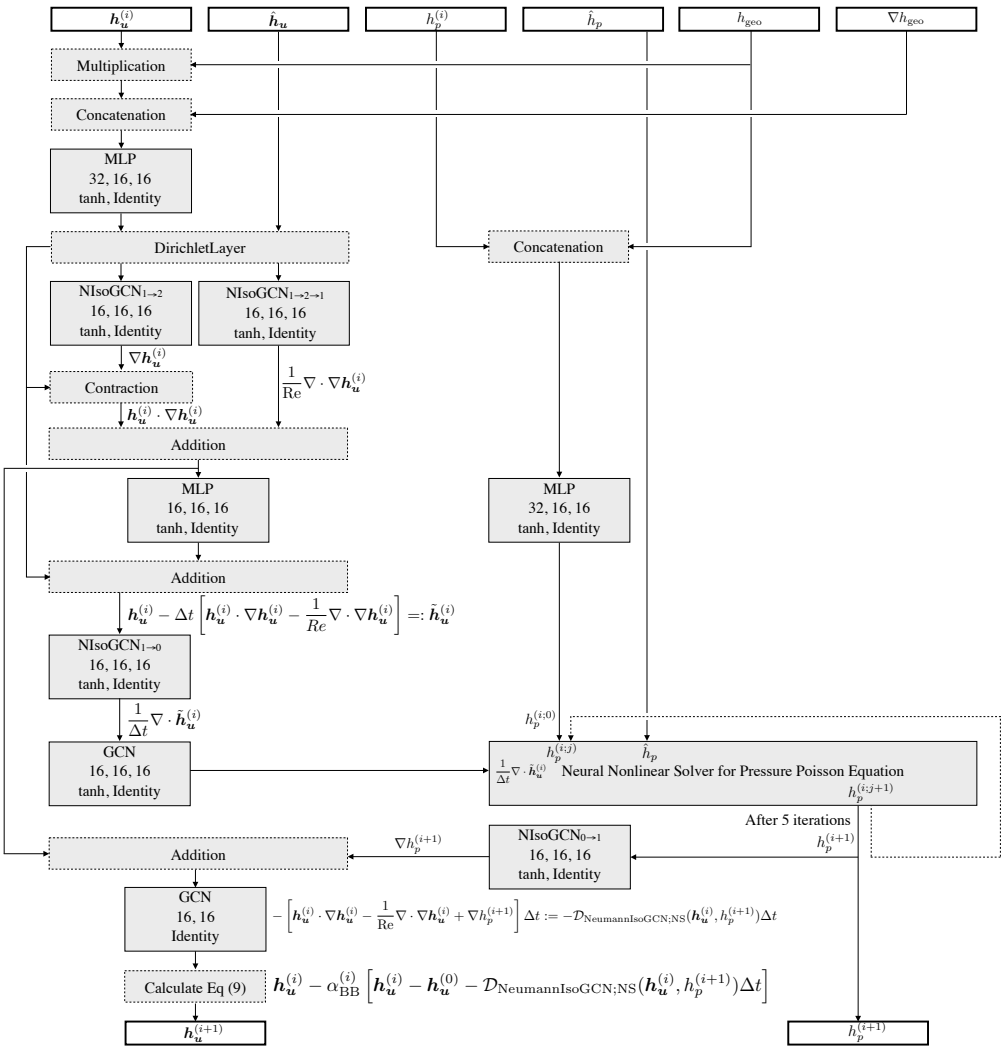

Figure 10: The neural nonlinear solver for velocity. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. Arrows with dotted lines correspond to the loop. In each cell, we put the number of units in each layer along with the activation functions used.

space. After features are encoded, we apply a neural nonlinear solver containing NeumanIsoGCNs and Dirichlet layers, reflecting the fractional step method (Equations 51 and 52).

The encoded equations are expressed as:

$$[\mathrm{NIsoGCN}_{1\to 0} \circ \mathrm{NIsoGCN}_{0\to 1}\,(h_p)](t+\Delta t, \boldsymbol{x}) = \frac{1}{\Delta t}\left[\mathrm{NIsoGCN}_{1\to 0}\left(\tilde{\boldsymbol{h}}_{\boldsymbol{u}}\right)\right](t, \boldsymbol{x}), \tag{55}$$

$$\tilde{\boldsymbol{h}}_{\boldsymbol{u}} := \boldsymbol{h}_{\boldsymbol{u}} - \Delta t\left[\boldsymbol{h}_{\boldsymbol{u}} \cdot \mathrm{NIsoGCN}_{1\to 2}\,(\boldsymbol{h}_{\boldsymbol{u}}) - \frac{1}{\mathrm{Re}}\mathrm{NIsoGCN}_{2\to 1} \circ \mathrm{NIsoGCN}_{1\to 2}\,(\boldsymbol{h}_{\boldsymbol{u}})\right], \tag{56}$$

$$\boldsymbol{h}_{\boldsymbol{u}}(t+\Delta t, \boldsymbol{x}) = \tilde{\boldsymbol{h}}_{\boldsymbol{u}}(t, \boldsymbol{x}) - \Delta t\,\mathrm{NIsoGCN}_{0\to 1}\,(h_p)\,(t+\Delta t, \boldsymbol{x}), \tag{57}$$

where $\boldsymbol{h}_{\boldsymbol{u}}$ is the encoded $\boldsymbol{u}$ and h_p is the encoded p. Note that these equations correspond to Equations 50, 51, and 52, by regarding IsoGCNs as spatial derivative operators. The corresponding neural nonlinear solvers are

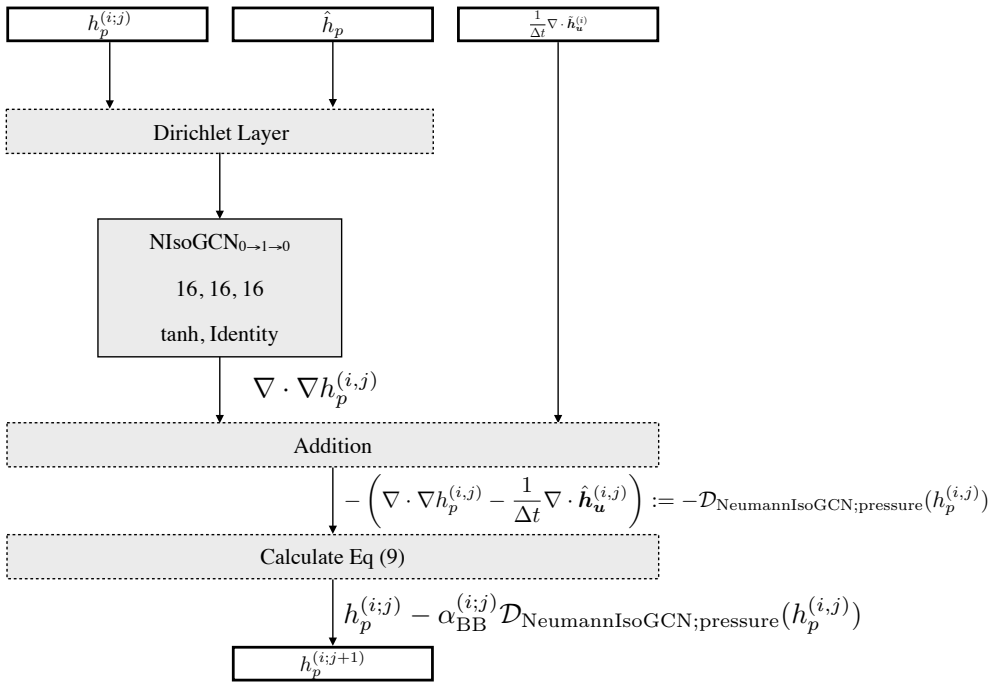

Figure 11: The neural nonlinear solver for pressure. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. In each cell, we put the number of units in each layer along with the activation functions used.

expressed as:

$$\boldsymbol{h}_{\boldsymbol{u}}^{(i+1)} = \boldsymbol{h}_{\boldsymbol{u}}^{(i)} - \alpha_{\mathrm{BB}}^{(i)}\left[\boldsymbol{h}_{\boldsymbol{u}}^{(i)} - \boldsymbol{h}_{\boldsymbol{u}}^{(0)} - \mathcal{D}_{\mathrm{NIsoGCN;NS}}\left(\boldsymbol{h}_{\boldsymbol{u}}^{(i)}, h_p^{(i+1)}\right)\Delta t\right], \tag{58}$$

$$\begin{aligned}&\mathcal{D}_{\mathrm{NIsoGCN;NS}}\left(\boldsymbol{h}_{\boldsymbol{u}}^{(i)}, h_p^{(i+1)}\right)\\ &:= \left[\boldsymbol{h}_{\boldsymbol{u}}^{(i)} \cdot \mathrm{NIsoGCN}_{1\to2}\left(\boldsymbol{h}_{\boldsymbol{u}}^{(i)}\right) - \frac{1}{\mathrm{Re}}\mathrm{NIsoGCN}_{2\to1} \circ \mathrm{NIsoGCN}_{1\to2}\left(\boldsymbol{h}_{\boldsymbol{u}}^{(i)}\right) + \mathrm{NIsoGCN}\left(h_p^{(i+1)}\right)\right],\end{aligned} \tag{59}$$

for $\boldsymbol{h}_{\boldsymbol{u}}$ and

$$h_p^{(i;j+1)} = h_p^{(i;j)} - \alpha_{\mathrm{BB}}^{(i;j)}\mathcal{D}_{\mathrm{NIsoGCN;pressure}}(h_p^{(i;j)}), \tag{60}$$

$$\mathcal{D}_{\mathrm{NIsoGCN;pressure}}\left(h_p^{(i;j)}\right) := \left(\mathrm{NIsoGCN}_{1\to0} \circ \mathrm{NIsoGCN}_{0\to1}\left(h_p^{(;,j)}\right) - \frac{1}{\Delta t}\mathrm{NIsoGCN}_{1\to0}\left(\hat{\boldsymbol{h}}_{\boldsymbol{u}}^{(i)}\right)\right), \tag{61}$$

for h_p, where $\boldsymbol{h}_{\boldsymbol{u}}^{(0)} = \boldsymbol{h}_{\boldsymbol{u}}(t,\cdot)$, $h_p^{(0)} = h_p(t,\cdot)$, and $h_p^{(i;0)} = h_p^{(i)}$. For notation regarding NIsoGCNs, please see Appendix A.2. Figures 9, 10, and 11 present the PENN model architecture used for the incompressible flow dataset.

As seen in Figure 10, we have a subloop that solves the Poisson equation for pressure in the nonlinear solver's loop for velocity. We looped the solver for pressure five times and eight times for velocity. After these loops stopped, we decoded the hidden features to obtain predictions for velocity and pressure, using the corresponding pseudoinverse decoders.

C.4 Implementation details

As discussed in Horie et al. (2021), nonlinearity can be applied to the scalar but cannot be applied to the tensors with a rank equal to or greater than one. For such a tensor, nonlinearity can be applied to its norm as:

$$\mathrm{MLP}_{\mathrm{tensor}}(\boldsymbol{v}) := \mathrm{MLP}(\|\boldsymbol{v}\|)\boldsymbol{v}. \tag{62}$$

This strategy to apply nonlinearity is used not only in the MLP blocks but also NIsoGCN blocks. To facilitate the smoothness of pressure and velocity fields, we apply GCN layers corresponding to numerical viscosity in the standard numerical analysis method. Here, please note that the PENN model consists of components that accept arbitrary input lengths, e.g., pointwise MLPs, deep sets, and NIsoGCNs. Thanks to the model's flexibility, we can apply the same model to arbitrary meshes similar to other GNNs.

C.5 Training details

Because the neural nonlinear solver applies the same layers many times during the loop, the model behaved somehow similar to recurrent neural networks during training, which could cause instability. To avoid such unwanted behavior, we simply retried training by reducing the learning rate of the Adam optimizer by a factor of 0.5. We found our way of training useful compared to using the learning rate schedule because sometimes the loss value of PENN can be extremely high, resulting in difficulty to reach convergence with a lower learning rate after such an explosion. Therefore, we applied early stopping and restarted training using a lower learning rate from the epoch with the best validation loss. Our initial learning rate was 5.0×10^{-4}, and we restarted the training twice, which was done automatically, within the 24-hour training period of PENN. For the ablation study, we used the same setting for all models. For PENN and ablation models, we used Adam (Kingma & Ba, 2014) as an optimizer. For MP-PDE solvers, we used the default setting written in the paper and the code.

C.6 Result details

Table 3 presents the detailed results of the comparison between MP-PDE and PENN. Interestingly, the performance of MP-PDE gets better as the time window size increases. Therefore, our future direction may be to incorporate MP-PDE's temporal bundling and pushforward trick into PENN to enable us to predict the state after a far longer time than we do in the present work.

Tables 4 and 5 show the speed and accuracy of the machine learning models tested. PENN models show excellent performance with a lot smaller number of parameters compared to MP-PDE models. It is achieved due to efficient parameter sharing in the proposed model, e.g., the same weights are used repeatedly in the neural nonlinear encoder. Also, as pointed out in Ravanbakhsh et al. (2017), there is a strong connection between parameter sharing and equivariance. PENN has equivariance in, e.g., permutation, time translation, and $\mathrm{E}(n)$ through parameter sharing, which is in line with them.

Table 6 presents the speed and accuracy with various settings of OpenFOAM to seek a speed-accuracy tradeoff. We tested three configurations of linear solvers:

- Generalized geometric-algebraic multi-grid (GAMG) for p and the smooth solver for $\boldsymbol{u}$
- Generalized geometric-algebraic multi-grid (GAMG) for both p and $\boldsymbol{u}$
- The smooth solver for p and $\boldsymbol{u}$

In addition, we tested different resolutions for space and time by changing:

- The number of divisions per unit length: 22.5, 45.0, 90.0
- Time step size: 0.001, 0.005, 0.010, 0.050

Ground truth is computed using the number of divisions per unit length of 90.0 and time step size of 0.001; thus, this combination is eliminated from the comparison because the MSE error is underestimated (in particular, zero).

C.7 Ablation study details

To validate the effectiveness of our model through an ablation study on the following settings:

(A) Without encoded boundary: In the nonlinear loop, we decode features to apply boundary conditions to fulfill Dirichlet conditions in the original physical space

(B) Without boundary condition in the neural nonlinear solver: We removed the Dirichlet layer in the nonlinear loop. Instead, we added the Dirichlet layer after the (non-pseudoinverse) decoder.

(C) Without neural nonlinear solver: We removed the nonlinear solver from the model and used the explicit time-stepping instead

(D) Without boundary condition input: We removed the boundary condition from input features

(E) Without Dirichlet layer: We removed the Dirichlet layer. Instead, we let the model learn to satisfy boundary conditions during training.

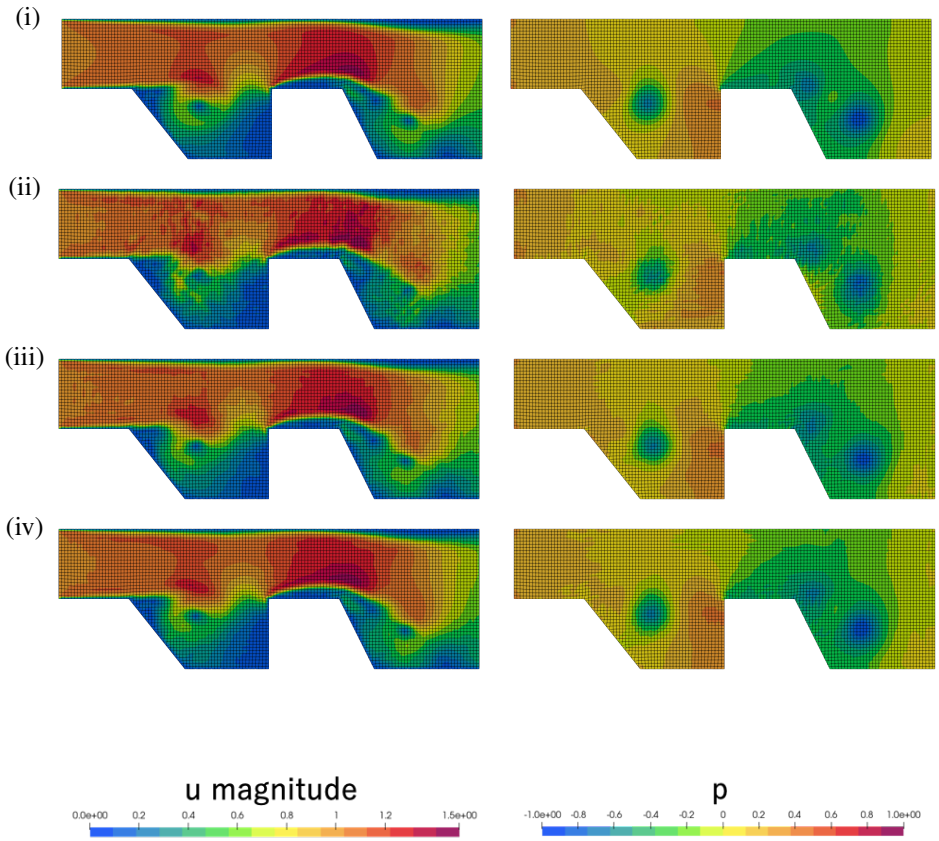

Figure 12: Visual comparison of the ablation study of (i) ground truth, (ii) the model without the neural nonlinear solver (Model (C)), (iii) the model without pseudoinverse decoder with Dirichlet layer after decoding (Model (G)), and (iv) PENN. It can be observed that PENN improves the prediction smoothness, especially for the velocity field.

(F) Without pseudoinverse decoder: We removed the pseudoinverse decoder and used simple MLPs for decoders.

(G) Without pseudoinverse decoder with Dirichlet boundary layer after decoding: Same as above, but with Dirichlet layer after decoding.

We again put the results of the ablation study in Table 7, which is already presented in Table **??**, for the convenience of the readers.

Comparison with Model (A) shows that the nonlinear loop in the encoded space is inevitable for machine learning. This result is quite convincing because if the loop is made in the original space, the advantage of the expressive power of the neural networks cannot be leveraged. Comparison with Model (C) confirms that the concept of the solver is effective compared to simply stacking GNNs, corresponding to the explicit method.

If the boundary condition input is excluded (Model (D)), the performance degrades in line with Brandstetter et al. (2022). That model also has an error on the Dirichlet boundaries. Model (E) shows a similar result, improving performance using the information of the boundary conditions. If the pseudoinverse decoder is excluded (Model (F)), the output may not satisfy the Dirichlet boundary conditions as well. Besides, the decoder has more effect than expected because PENN is better than Model (G). Both models satisfy the Dirichlet boundary condition, while PENN has significant improvement. This may be because the pseudoinverse decoder facilitates the spatial continuity of the outputs in addition to the fulfillment of the Dirichlet boundary condition. In other words, using a simple decoder and the Dirichlet layer after that may cause spatial discontinuity of outputs. Visual comparison of part of the ablation study is shown in Figure 12.

Table 3: MSE loss ($\pm$ the standard error of the mean) on test dataset of incompressible flow. If "Trans." is "Yes", it means evaluation on randomly rotated and transformed test dataset. n denotes the number of hidden features, r denotes the number of iterations in the neural nonlinear solver used in PENN models, and TW denotes the time window size used in MP-PDE models.

Method	Trans.	$\boldsymbol{u}$ ($\times 10^{-4}$)	p ($\times 10^{-3}$)	$\hat{\boldsymbol{u}}_{\text{Dirichlet}}$ ($\times 10^{-4}$)	$\hat{p}_{\text{Dirichlet}}$ ($\times 10^{-3}$)
PENN	No	4.36 ± 0.03	1.17 ± 0.01	0.00 ± 0.00	0.00 ± 0.00
$n = 16, r = 8$	Yes	4.36 ± 0.03	1.17 ± 0.01	0.00 ± 0.00	0.00 ± 0.00
PENN	No	29.09 ± 0.17	11.35 ± 0.04	0.00 ± 0.00	0.00 ± 0.00
$n = 16, r = 4$	Yes	29.09 ± 0.17	11.35 ± 0.04	0.00 ± 0.00	0.00 ± 0.00
PENN	No	177.42 ± 0.93	35.70 ± 0.12	0.00 ± 0.00	0.00 ± 0.00
$n = 8, r = 8$	Yes	177.42 ± 0.93	35.70 ± 0.12	0.00 ± 0.00	0.00 ± 0.00
PENN	No	26.82 ± 0.16	7.86 ± 0.03	0.00 ± 0.00	0.00 ± 0.00
$n = 8, r = 4$	Yes	26.82 ± 0.16	7.86 ± 0.03	0.00 ± 0.00	0.00 ± 0.00
PENN	No	92.80 ± 0.52	31.47 ± 0.13	0.00 ± 0.00	0.00 ± 0.00
$n = 4, r = 8$	Yes	92.80 ± 0.52	31.47 ± 0.13	0.00 ± 0.00	0.00 ± 0.00
PENN	No	120.35 ± 0.65	35.53 ± 0.12	0.00 ± 0.00	0.00 ± 0.00
$n = 4, r = 4$	Yes	120.35 ± 0.65	35.53 ± 0.12	0.00 ± 0.00	0.00 ± 0.00
MP-PDE	No	1.30 ± 0.01	1.32 ± 0.01	0.45 ± 0.01	0.28 ± 0.02
$n = 128, \text{TW} = 20$	Yes	1953.62 ± 7.62	281.86 ± 0.78	924.73 ± 6.14	202.97 ± 3.81
MP-PDE	No	12.08 ± 0.11	6.49 ± 0.03	1.36 ± 0.01	2.57 ± 0.05
$n = 128, \text{TW} = 10$	Yes	1468.12 ± 5.75	192.97 ± 0.57	767.17 ± 4.36	51.87 ± 1.07
MP-PDE	No	32.07 ± 0.33	6.22 ± 0.05	0.85 ± 0.01	0.92 ± 0.03
$n = 128, \text{TW} = 4$	Yes	2068.99 ± 8.30	180.54 ± 0.57	284.72 ± 1.69	59.21 ± 1.32
MP-PDE	No	58.88 ± 0.60	9.62 ± 0.07	1.02 ± 0.02	2.83 ± 0.10
$n = 128, \text{TW} = 2$	Yes	1853.27 ± 7.89	219.59 ± 0.53	965.90 ± 28.61	358.53 ± 2.13
MP-PDE	No	6.09 ± 0.05	5.39 ± 0.03	1.65 ± 0.02	2.16 ± 0.08
$n = 64, \text{TW} = 20$	Yes	1969.34 ± 7.50	388.54 ± 1.12	720.35 ± 5.15	218.06 ± 8.01
MP-PDE	No	38.54 ± 0.32	31.33 ± 0.09	2.04 ± 0.02	5.87 ± 0.09
$n = 64, \text{TW} = 10$	Yes	2738.84 ± 9.37	171.32 ± 0.60	417.57 ± 2.49	28.34 ± 0.92
MP-PDE	No	125.09 ± 1.11	21.93 ± 0.09	2.27 ± 0.03	5.92 ± 0.16
$n = 64, \text{TW} = 2$	Yes	1402.01 ± 6.03	435.75 ± 2.41	384.30 ± 4.13	57.26 ± 1.90
MP-PDE	No	32.46 ± 0.24	17.40 ± 0.07	5.92 ± 0.05	5.94 ± 0.17
$n = 32, \text{TW} = 20$	Yes	2201.16 ± 7.59	351.66 ± 0.82	429.30 ± 3.27	562.16 ± 11.62
MP-PDE	No	115.30 ± 1.01	34.97 ± 0.15	10.26 ± 0.09	6.84 ± 0.14
$n = 32, \text{TW} = 10$	Yes	2824.76 ± 8.60	496.33 ± 1.33	2276.11 ± 10.57	488.50 ± 5.01
MP-PDE	No	272.73 ± 2.07	94.27 ± 0.45	11.50 ± 0.12	35.76 ± 0.29
$n = 32, \text{TW} = 4$	Yes	1973.35 ± 8.29	554.69 ± 4.26	647.31 ± 7.40	157.85 ± 8.41
MP-PDE	No	794.90 ± 4.68	82.61 ± 0.40	50.23 ± 0.91	31.41 ± 1.88
$n = 32, \text{TW} = 2$	Yes	3240.69 ± 21.91	443.10 ± 2.56	2885.30 ± 41.17	562.08 ± 19.28

Table 4: MSE loss ($\pm$ the standard error of the mean) of PENN models on test dataset of incompressible flow.

# hidden feature	# iteration in the neural nonlinear solver	# parameter	Total MSE ($\times 10^{-3}$)	Total time [s]
16	8	8,432	1.61 ± 0.01	5.33 ± 0.13
16	4	8,432	14.26 ± 0.03	2.52 ± 0.06
8	8	2,100	53.44 ± 0.11	3.54 ± 0.08
8	4	2,100	10.54 ± 0.03	2.16 ± 0.04
4	8	596	40.75 ± 0.10	2.86 ± 0.06
4	4	596	47.57 ± 0.10	1.35 ± 0.04

Table 5: MSE loss ($\pm$ the standard error of the mean) of MP-PDE models on test dataset of incompressible flow.

# hidden feature	Time window size	# parameter	Total MSE ($\times 10^{-3}$)	Total MSE (Trans.) ($\times 10^{-3}$)	Total time [s]
128	20	709,316	1.45 ± 0.01	477.23 ± 0.77	51.61 ± 1.41
128	10	673,484	7.70 ± 0.02	339.78 ± 0.57	94.01 ± 2.66
128	4	651,972	9.43 ± 0.04	387.44 ± 0.71	137.32 ± 3.91
128	2	644,548	15.51 ± 0.07	404.92 ± 0.67	57.28 ± 1.91
64	20	204,004	6.00 ± 0.02	585.48 ± 0.95	13.62 ± 0.38
64	10	185,356	35.19 ± 0.07	445.20 ± 0.79	23.73 ± 0.67
64	2	174,740	34.44 ± 0.10	575.95 ± 1.76	32.61 ± 1.02
32	20	63,964	20.64 ± 0.05	571.77 ± 0.79	7.64 ± 0.24
32	10	55,348	46.50 ± 0.13	778.80 ± 1.12	12.93 ± 0.39
32	4	49,948	121.55 ± 0.35	752.03 ± 3.07	13.99 ± 0.41
32	2	47,924	162.10 ± 0.44	767.17 ± 2.38	4.55 ± 0.13

Table 6: MSE loss ($\pm$ the standard error of the mean) of OpenFOAM computations on test dataset of incompressible flow.

Solver for $\boldsymbol{u}$	Solver for p	# division per unit length	Δt	Total MSE ($\times 10^{-3}$)	Total time [s]
GAMG	Smooth	22.5	0.050	Divergent	Divergent
GAMG	Smooth	22.5	0.010	6.09 ± 0.02	6.08 ± 0.17
GAMG	Smooth	22.5	0.005	6.04 ± 0.02	11.57 ± 0.32
GAMG	Smooth	22.5	0.001	4.80 ± 0.02	51.43 ± 1.39
GAMG	Smooth	45.0	0.050	Divergent	Divergent
GAMG	Smooth	45.0	0.010	0.46 ± 0.00	25.12 ± 0.81
GAMG	Smooth	45.0	0.005	0.78 ± 0.00	46.71 ± 1.53
GAMG	Smooth	45.0	0.001	1.04 ± 0.00	201.11 ± 6.29
GAMG	Smooth	90.0	0.050	Divergent	Divergent
GAMG	Smooth	90.0	0.010	Divergent	Divergent
GAMG	Smooth	90.0	0.005	0.15 ± 0.00	231.18 ± 10.38
GAMG	GAMG	22.5	0.050	Divergent	Divergent
GAMG	GAMG	22.5	0.010	6.05 ± 0.02	6.41 ± 0.18
GAMG	GAMG	22.5	0.005	6.00 ± 0.02	12.21 ± 0.34
GAMG	GAMG	22.5	0.001	4.80 ± 0.02	55.51 ± 1.52
GAMG	GAMG	45.0	0.050	Divergent	Divergent
GAMG	GAMG	45.0	0.010	0.46 ± 0.00	26.00 ± 0.85
GAMG	GAMG	45.0	0.005	0.77 ± 0.00	48.78 ± 1.57
GAMG	GAMG	45.0	0.001	1.03 ± 0.00	214.29 ± 6.62
GAMG	GAMG	90.0	0.050	Divergent	Divergent
GAMG	GAMG	90.0	0.010	Divergent	Divergent
GAMG	GAMG	90.0	0.005	0.14 ± 0.00	238.94 ± 10.70
Smooth	Smooth	22.5	0.050	Divergent	Divergent
Smooth	Smooth	22.5	0.010	5.59 ± 0.02	85.50 ± 3.05
Smooth	Smooth	22.5	0.005	5.41 ± 0.02	164.36 ± 7.57
Smooth	Smooth	22.5	0.001	4.19 ± 0.02	765.50 ± 29.65
Smooth	Smooth	45.0	0.050	Divergent	Divergent
Smooth	Smooth	45.0	0.010	51.10 ± 0.05	426.07 ± 22.51
Smooth	Smooth	45.0	0.005	2.09 ± 0.00	824.71 ± 39.90
Smooth	Smooth	45.0	0.001	1.12 ± 0.00	3960.88 ± 151.93
Smooth	Smooth	90.0	0.050	Divergent	Divergent
Smooth	Smooth	90.0	0.010	Divergent	Divergent
Smooth	Smooth	90.0	0.005	4493.78 ± 1.88	3566.05 ± 183.75

Table 7: Ablation study on 2D incompressible flow dataset. The value represents MSE loss ($\pm$ standard error of the mean) on the test dataset. "Divergent" means the implicit solver does not converge and the loss gets extreme value ($\sim 10^{14}$). This presents the same results as Table **??**.

Method	$\boldsymbol{u}$ ($\times 10^{-4}$)	p ($\times 10^{-3}$)	$\hat{\boldsymbol{u}}_{\text{Dirichlet}}$ ($\times 10^{-4}$)	$\hat{p}_{\text{Dirichlet}}$ ($\times 10^{-3}$)
(A) Without encoded boundary	Divergent	Divergent	Divergent	Divergent
(B) Without boundary condition in the neural nonlinear solver	65.10 ± 0.38	21.70 ± 0.09	0.00 ± 0.00	0.00 ± 0.00
(C) Without neural nonlinear solver	31.03 ± 0.19	9.81 ± 0.04	$\mathbf{0.00} \pm 0.00$	$\mathbf{0.00} \pm 0.00$
(D) Without boundary condition input	20.08 ± 0.21	3.61 ± 0.02	59.60 ± 0.89	1.43 ± 0.05
(E) Without Dirichlet layer	8.22 ± 0.07	1.41 ± 0.01	18.20 ± 0.28	0.38 ± 0.01
(F) Without pseudoinverse decoder	8.91 ± 0.06	2.36 ± 0.02	1.97 ± 0.06	$\mathbf{0.00} \pm 0.00$
(G) Without pseudoinverse decoder with Dirichlet layer after decoding	6.65 ± 0.05	1.71 ± 0.01	$\mathbf{0.00} \pm 0.00$	$\mathbf{0.00} \pm 0.00$
PENN	$\mathbf{4.36} \pm 0.03$	$\mathbf{1.17} \pm 0.01$	$\mathbf{0.00} \pm 0.00$	$\mathbf{0.00} \pm 0.00$

D Experiment details: advection-diffusion dataset

To test the generalization ability of PENNs regarding PDE's parameters and time series, we run an experiment with the advection-diffusion dataset. The governing equation regarding the temperature field T used for the experiment is expressed as:

$$\frac{\partial T}{\partial t} = -c \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \nabla T + D \nabla \cdot \nabla T \qquad (t, \boldsymbol{x}) \in (0,1) \times \Omega, \tag{63}$$

$$T(t=0, \boldsymbol{x}) = 0 \qquad \boldsymbol{x} \in \Omega, \tag{64}$$

$$T = \hat{T} \qquad (t, \boldsymbol{x}) \in \partial\Omega_{\text{Dirichlet}}, \tag{65}$$

$$\nabla T \cdot \boldsymbol{n} = 0 \qquad (t, \boldsymbol{x}) \in \partial\Omega_{\text{Neumann}}, \tag{66}$$

where $c \in \mathbb{R}$ is the magnitude of a known velocity field, and $D \in \mathbb{R}$ is the diffusion coefficient. We set $\Omega = \{\boldsymbol{x} \in \mathbb{R}^3 \mid 0 < x_1 < 1 \wedge 0 < x_2 < 1 \wedge 0 < x_3 < 0.01\}$, $\partial\Omega_{\text{Dirichlet}} = \{\boldsymbol{x} \in \partial\Omega \mid x_1 = 0\}$ and $\partial\Omega_{\text{Neumann}} = \partial\Omega \setminus \partial\Omega_{\text{Dirichlet}}$.

D.1 Dataset

We varied c and D from 0.0 to 1.0, eliminating the condition $c = D = 0.0$ because nothing drives the phenomena, and and varied $\hat{T}$ from 0.1 to 1.0. Like the incompressible flow dataset, we generated fine meshes, ran computation with OpenFOAM, and interpolated the obtained temperature fields onto coarser meshes. We split the generated data into training, validation, and test dataset containing 960, 120, and 120 samples. The dataset is uploaded online.[8]

D.2 Model architecture

The strategy to construct PENN for the advection-diffusion dataset is consistent with one for the incompressible flow dataset (see Appendix C.3). The input features of the model are:

- $T(t = 0.0)$: The initial temperature field
- $\hat{T}$: The Dirichlet boundary condition for the temperature field
- $(c, 0, 0)^\top$: The velocity field
- c: The magnitude of the velocity
- D: The diffusion coefficient
- $e^{-0.5d}, e^{-1.0d}, e^{-2.0d}$: Features computed from d, the distance from the Dirichlet boundary

and the output features are:

- $T(t = 0.25)$: The temperature field at $t = 0.25$
- $T(t = 0.50)$: The temperature field at $t = 0.50$
- $T(t = 0.75)$: The temperature field at $t = 0.75$
- $T(t = 1.00)$: The temperature field at $t = 1.00$

The encoded governing equation is expressed as:

$$h_T(t + \Delta t, \boldsymbol{x}) = h_T(t, \boldsymbol{x}) + \mathcal{D}_{\text{NIsoGCN;A-D}}\left(h_T\right)(t + \Delta t, \boldsymbol{x}) \tag{67}$$

$$\mathcal{D}_{\text{NIsoGCN;A-D}}\left(h_T\right) := -\boldsymbol{h_c} \cdot \text{NIsoGCN}_{0\to1}(h_T) + h_D \ \text{NIsoGCN}_{0\to1\to0}(h_T) \tag{68}$$

The corresponding neural nonlinear solver is:

$$h_T^{(i+1)} = h_T^{(i)} - \alpha_{\text{BB}}^{(i)} \left[h_T^{(i)} - h_T^{(0)} - \mathcal{D}_{\text{NIsoGCN;A-D}}(h_T^{(i)}) \Delta t \right], \tag{69}$$

Because the task is to predict time series data, we adopt autoregressive architecture for the nonlinear neural solver, i.e., input the output of the solver of the previous step (which is in the encoded space) to predict the encoded feature of the next step (see Figure 13). Figures 14 and 15 present the detailed architecture of the PENN model for the advection-diffusion dataset experiment.

To confirm the PENN's effectiveness, we ran the ablation study similar to that in the incompressible flow dataset. The training is performed for up to ten hours using the Adam optimizer for each setting.

[8] `https://drive.google.com/uc?export=download&id=1bQ6RKcSbHsg2D0gikYRelCBu6bN88Lk2`

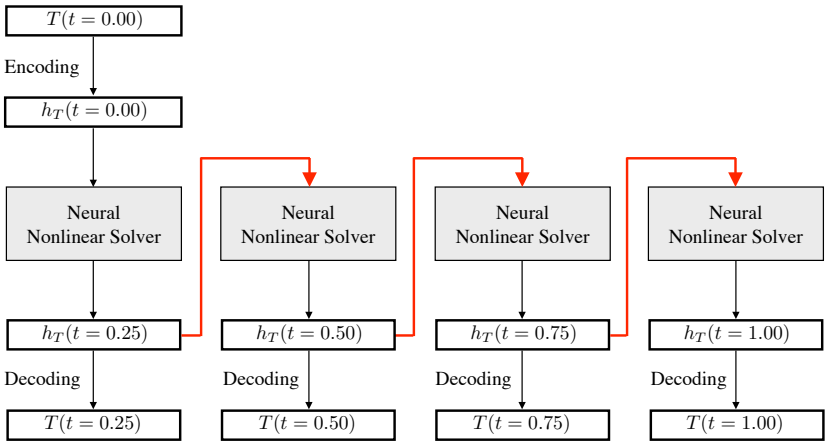

Figure 13: The concept of the neural nonlinear solver for time series data with autoregressive architecture. The solver's output is fed to the same solver to obtain the state at the next time step (bold red arrow). Please note that this architecture can be applied to arbitrary time series lengths.

Table 8: MSE loss ($\pm$ the standard error of the mean) on test dataset of the advection-diffusion dataset.

Method	T ($\times 10^{-4}$)	$\hat{T}_{\text{Dirichlet}}$ ($\times 10^{-4}$)
(A) Without encoded boundary	54.191 ± 6.36	0.0000 ± 0.0000
(B) Without boundary condition in the neural nonlinear solver	390.828 ± 24.58	0.0000 ± 0.0000
(C) Without neural nonlinear solver	6.630 ± 1.21	0.0000 ± 0.0000
(D) Without boundary condition input	465.492 ± 26.47	868.7009 ± 15.5447
(E) Without Dirichlet layer	2.860 ± 2.46	1.1703 ± 0.0328
(F) Without pseudoinverse decoder	44.947 ± 6.00	9.7130 ± 0.1201
(G) Without pseudoinverse decoder with Dirichlet layer after decoding	4.907 ± 4.87	0.0000 ± 0.0000
PENN	$\mathbf{1.795} \pm 1.33$	0.0000 ± 0.0000

D.3 Results

Table 8 presents the results of the ablation study. As well as the incompressible flow dataset, we found that the PENN model with all the proposed components achieved the best performance. Because the boundary condition applied is relatively simple compared to the incompressible flow dataset, the configuration without the Dirichlet layer (Model (E)) showed the second best performance; however, the fulfillment of the Dirichlet condition of that model is not rigorous.

Figures 16, 17, and 18 show the visual comparison of the prediction with the PENN model against the ground truth. As seen in the figures, one can see that our model is capable of predicting time series under various boundary conditions and PDE parameters, e.g., pure advection (Figure 16), pure diffusion (Figure 17), and mixed advection and diffusion (Figure 18).

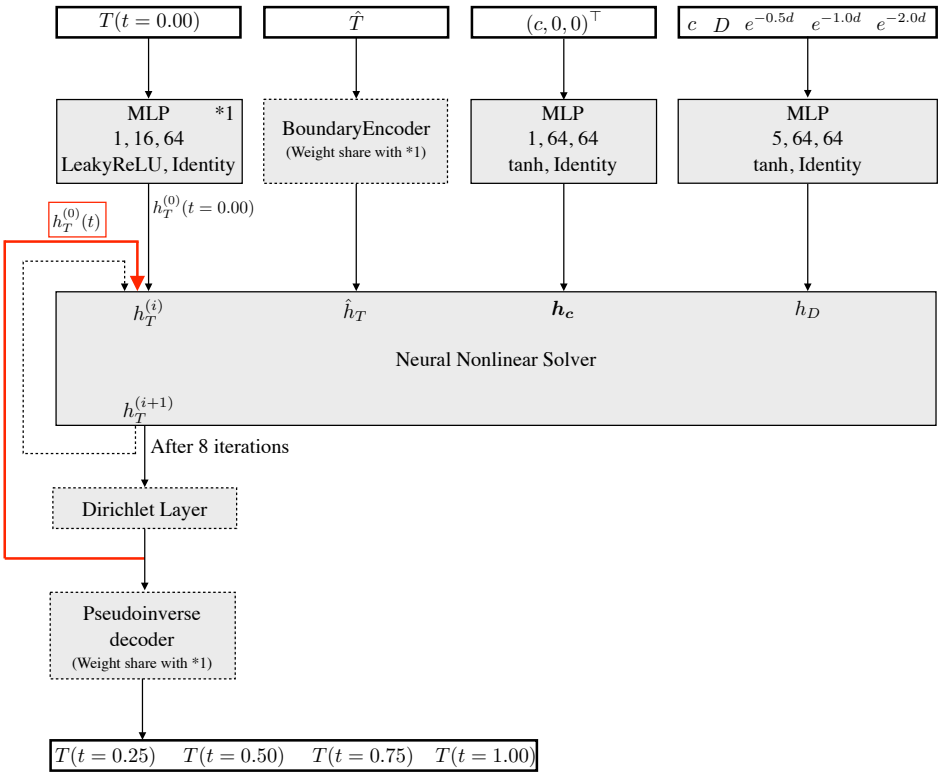

Figure 14: The overview of the PENN architecture for the advection-diffusion dataset. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. Arrows with dotted lines correspond to the loop. In each cell, we put the number of units in each layer along with the activation functions used. The bold red arrow corresponds to the one in Figure 13.

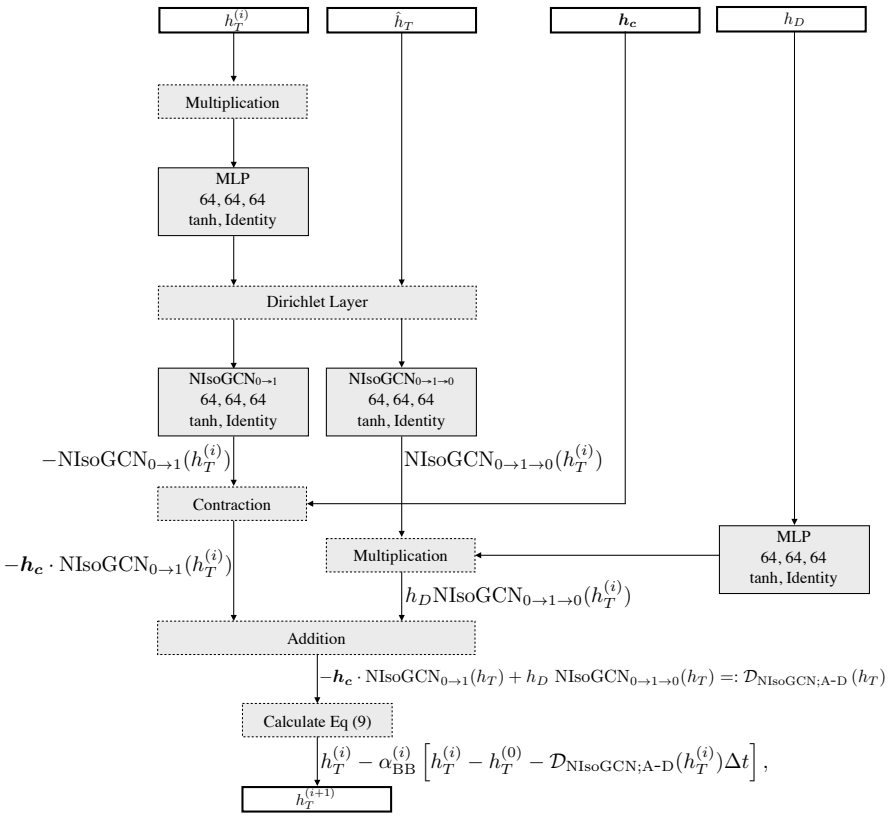

Figure 15: The overview of the PENN architecture for the advection-diffusion dataset. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. In each cell, we put the number of units in each layer along with the activation functions used.

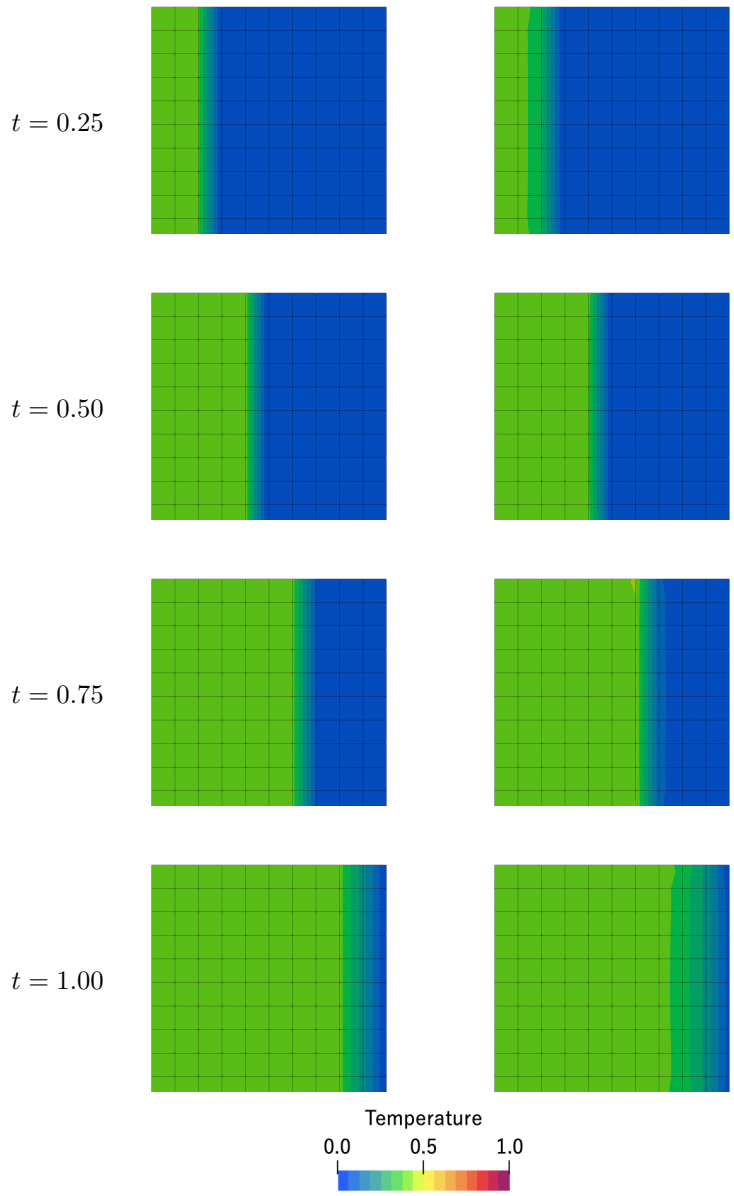

Figure 16: Visual comparison on a test sample between (left) ground truth obtained from OpenFOAM computation with fine spatial-temporal resolution and (right) prediction by PENN. Here, $c = 0.9$, $D = 0.0$, and $\hat{T} = 0.4$.

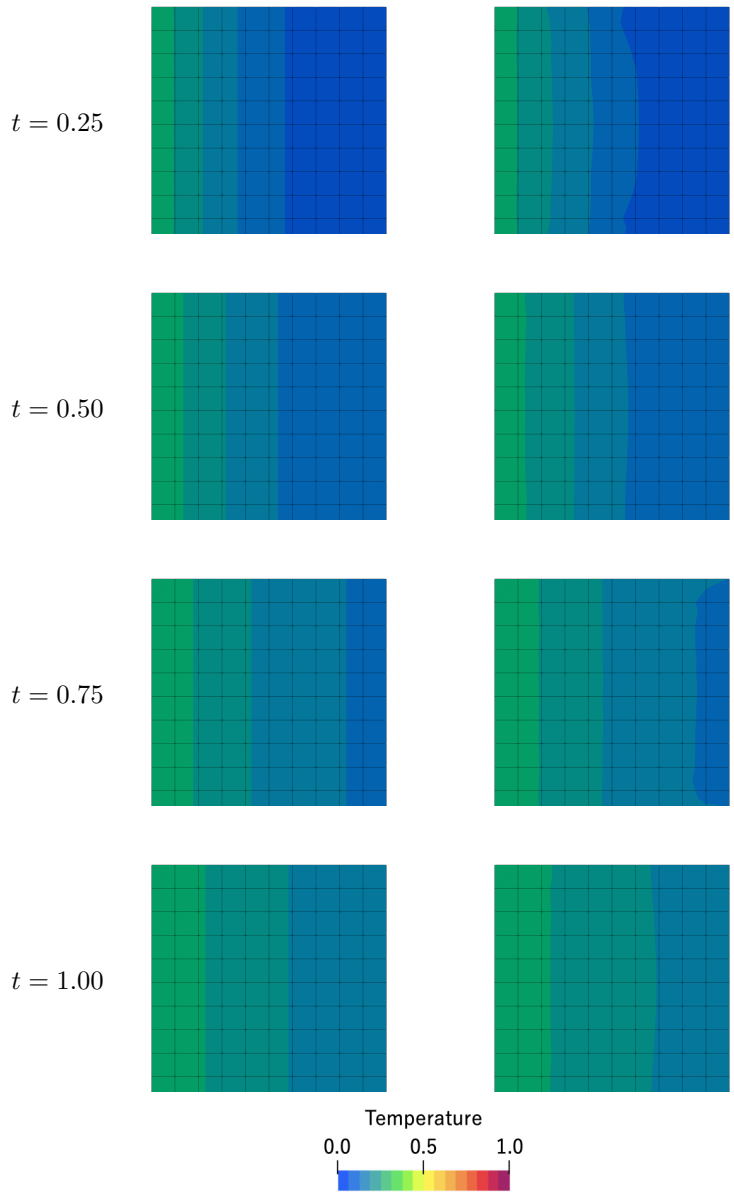

Figure 17: Visual comparison on a test sample between (left) ground truth obtained from OpenFOAM computation with fine spatial-temporal resolution and (right) prediction by PENN. Here, $c = 0.0$, $D = 0.4$, and $\hat{T} = 0.3$.

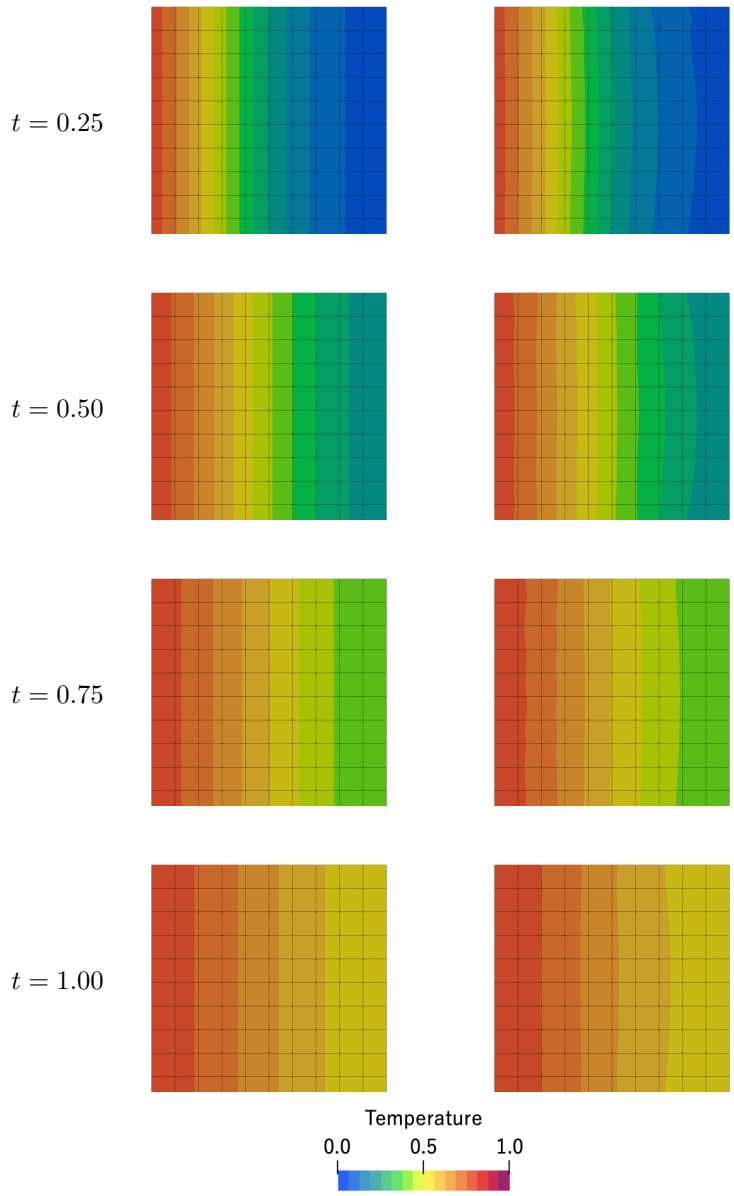

Figure 18: Visual comparison on a test sample between (left) ground truth obtained from OpenFOAM computation with fine spatial-temporal resolution and (right) prediction by PENN. Here, $c = 0.6$, $D = 0.3$, and $\hat{T} = 0.8$.