E(n)-Equivariant Graph Neural Networks Emulating Mesh-Discretized Physics

March 2023

Masanobu Horie

E(n)-Equivariant Graph Neural Networks Emulating Mesh-Discretized Physics

Graduate School of Science and Technology

Degree Programs in Systems and Information Engineering

University of Tsukuba

March 2023

Masanobu Horie

Acknowledgment

First and foremost, I would like to thank my supervisor Asst. Prof. Naoto Mitsume for all his support. I tremendously appreciate his inspiring advice for research, countless fruitful discussions, and enjoyable time. In addition to technical matters, he has shown me the importance of telling a clear story, thinking about the relationship between my and others' research, and having social connections with other researchers.

I am grateful to Prof. Daigoro Isobe, Assoc. Prof. Tetsuya Matsuda, Assoc. Prof. Mayuko Nishio, and Assoc. Prof. Mitsuteru Asai for reviewing my research and providing insightful feedback as my dissertation committee.

I would like to thank my collaborators, Asst. Prof. Naoki Morita, Dr. Toshiaki Hishinuma, and Mr. Yu Ihara. I learned a lot of technical things thanks to their support.

This work was supported by JSPS KAKENHI (Grant Number 19H01098), JSPS Grant-in-Aid for Scientific Research (B) (Grant Number 22H03601), JST PRESTO (Grant Number JPMJPR21O9), and NEDO (Grant Number JPNP14012). I gratefully acknowledge their support.

RICOS Co. Ltd. also supports the work in terms of computational resources and provides me an opportunity to pursue this Ph.D. project.

Finally, I would like to thank my family, my father Jun-ichi, mother Naoko, sister Yuriko, and wife Suzuka, for their continuous support. My wife, Suzuka, has been encouraging me at any time. She made me excellent food like tempura, karaage, and ozouni when I was happy or unhappy with my research. It is awesome to share my life with you.

Contents

List of Figures

List of Tables

Nomenclature

General

$\mathbb{R}$	Set of real numbers
$\mathbb{Z}$	Set of integers
$\mathbb{Z}^+$	Set of positive integers
$\mathbb{Z}^{0+}$	Set of non-negative integers
M_{ij}	Element (i, j) of matrix $\boldsymbol{M}$
v_i	Element i of vector $\boldsymbol{v}$

Chapter 2

$\mathcal{G}$	Graph
$\mathcal{V}$	Vertex set
$\lvert\mathcal{V}\rvert$	Number of vertices
$\mathcal{N}_v$	Set of neighboring vertices of vertex v (Equation 2.12)
$\boldsymbol{A}$	Adjacency matrix (Equation 2.14)
J	Index set
$\pi : J \to J$	Permutation
$\boldsymbol{P}$	Permutation matrix (Equation 2.15)
$\boldsymbol{L}$	Graph Laplacian matrix (Equation 2.17)

$\boldsymbol{D}$	Degree matrix (Equation 2.18)
$\boldsymbol{W}$	Weight matrix
$\boldsymbol{b}$	Bias
σ	Activation function
$\boldsymbol{H} \in \mathbb{R}^{\mathcal{V} \times d}$	Vertex features
$\{\boldsymbol{h}_i\}_{i \in \mathcal{V}}$	Set of vertex features
$\{\boldsymbol{e}_{ij}\}_{(i,j) \in \mathcal{E}}$	Set of edge features
$\hat{\boldsymbol{A}}$	Renormalized adjacency matrix
$\tilde{\boldsymbol{A}}$	Adjacency matrix with added self-connections
$\tilde{\boldsymbol{D}}$	Degree matrix of $\tilde{\boldsymbol{A}}$
$\boldsymbol{I}_N$	Identity matrix of size N
G	Group
$\mathrm{GL}(n)$	General linear group
$\mathrm{SO}(n)$	Special orthogonal group
S_n	Symmetric group
$\mathrm{E}(n)$	Euclidean group
$\alpha(g, x)$	Group action of $g \in G$ to $x \in X$ (also denoted as $g \cdot x$)
δ_{ij}	Kronecker delta (Equation 2.48)
$\mathcal{D}$	Nonlinear differential operator (Equation 2.52)
Ω	Analysis domain (Equation 2.52)
$\partial\Omega$	Boundary of Ω
$\partial\Omega_{\mathrm{Dirichlet}}$	Dirichlet boundary (Equation 2.54)
$\partial\Omega_{\mathrm{Neumann}}$	Neumann boundary (Equation 2.55)
$\boldsymbol{r}$	Residual (Equations 2.60, 2.62)

$\boldsymbol{R}$	Residual vector (Equations 2.63)
$\boldsymbol{e}_x, \boldsymbol{e}_y$	The unit vectors in the X and Y directions

Chapter 3

$\boldsymbol{H}^{(p)} \in \mathbb{R}^{\lvert\mathcal{V}\rvert \times d_{\mathrm{f}} \times n^p}$	A rank-p discrete tensor field (Equation 3.5)
$\boldsymbol{G} \in \mathbb{R}^{\lvert\mathcal{V}\rvert \times 1 \times d^1}$	IsoAM (Equation 3.8)
$\tilde{\boldsymbol{G}} \in \mathbb{R}^{\lvert\mathcal{V}\rvert \times 1 \times d^1}$	Differential IsoAM (Equation 3.46)
$\boldsymbol{G}_{ij;;:} = \boldsymbol{g}_{ij} \in \mathbb{R}^n$	Slice in the spatial index of $\boldsymbol{G}$

Chapter 4

$\mathcal{D}_{\mathrm{NIsoGCN}}$	Nonlinear differential operator constructed using NIsoGCN

Chapter 1

Introduction

1.1 MOTIVATION

Partial differential equations (PDEs) are of great interest to many scientists due to their wide-ranging applications in fields such as mathematics, physics, and engineering. Numerical analysis is commonly used to solve PDEs since most real-life PDE problems cannot be solved analytically. For instance, predicting fluid behavior in complex shapes is of particular significance in various fields, including product design, disaster reduction, and weather forecasting. However, solving these problems using classical solvers is time-consuming and challenging. Machine learning has emerged as a promising alternative for addressing these complex problems because, unlike classical solvers, it can leverage data that is similar to the state being predicted.

The main challenge in tackling complex phenomena like fluids mechanics using machine learning is to achieve good generalization performance, mainly owing to the following two reasons:

- **Variable degrees of freedom**: Classical numerical analysis methods discretize continuous fields of physical quantities (e.g., temperature or velocity fields) into variables at finite points in a mesh. The number of points, which correspond to the degrees of freedom of the analysis model, can vary depending on the shape of inter-

est, which requires some flexibility of the machine learning model to tolerate such uncertainty.

- **Large number of degrees of freedom**: A practical analysis often consists of a huge number of degrees of freedom, typically over a million. This is considerably larger than typical machine learning datasets, such as CIFAR-10 (Krizhevsky et al., 2009), which has 3072 features per sample. The number of possible states in such a complex system can be large and a purely data-driven approach may not cover them due to the curse of dimensionality.

To address these challenges, we must incorporate appropriate assumptions and knowledge about the phenomena of interest into the machine learning model, which is known as inductive bias. Numerous studies have successfully introduced various inductive biases, such as local connectedness using graph neural networks (GNNs) (Chang & Cheng, 2020; Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021; Brandstetter et al., 2022). These studies have shown that GNNs are effective in constructing PDE solvers as they can handle inputs with an arbitrary number of degrees of freedom.

Although these methods have made significant progress in solving PDEs using machine learning, there is still room for improvement. Specifically, we can incorporate more inductive biases to reduce the numbers of degrees of freedom, for example, by considering only half of the analysis domain if the phenomenon has bilateral symmetry, such as in the aerodynamic analysis of a symmetric aircraft.

First, the physical symmetry regarding isometric transformation, i.e., $\mathrm{E}(n)$ transformations, must be addressed when considering PDEs in Euclidean spaces because the nature of physical phenomena in such spaces does not change under these transformations. Thus, models that can accurately reflect physical symmetries, which are known as *equivariant* functions regarding the transformation of interest, must be used.

Second, there is a need for an efficient and provable way to satisfy mixed boundary conditions, i.e., Dirichlet and Neumann. Rigorous fulfillment of Dirichlet boundary conditions is indispensable because they are hard constraints, with different Dirichlet conditions corresponding to different problems users would like to solve.

Finally, we need to enhance the treatment of global interactions to predict the state after a long time, when interactions tend to be global. GNNs have excellent generalization properties because of their locally connected nature, but they may miss global interactions owing to their localness.

1.2 Objective and Scope

In this dissertation, we focus on mesh-based time-dependent numerical analysis. Mesh-based methods are widely utilized in practical numerical analysis due to their ability to handle complex shapes often encountered in industrial design. Time-dependent analysis typically demands a significant amount of computational time, as compared with steady-state analysis, because of the small time step required to ensure stable computation of the time evolution. Therefore, we aim to exploit the full potential of machine learning for conducting mesh-based time-dependent analyses.

The objective of this study is to develop a machine learning method that addresses the challenges previously discussed. We aim to build a machine learning model with the following key features:

1. Flexibility to handle arbitrary meshes using GNNs
2. $\mathrm{E}(n)$-equivariance to account for physical symmetries
3. Computational efficiency to provide faster predictions than conventional numerical analysis methods
4. Capability to rigorously consider boundary conditions
5. Stability for predicting over long time steps by considering global interactions

In a previous study (Horie et al., 2021), we introduced *IsoGCN*, a computationally efficient GNN that features $\mathrm{E}(n)$- invariance and equivariance, hence, complying with the first three requirements outlined above. Specifically, this model simply modifies the definition of an adjacency matrix essential for describing a graph, to realize $\mathrm{E}(n)$-equivariance. Because the proposed approach relies on graphs, it can handle complex shapes that are usually modeled using mesh or point cloud data structures. Furthermore, a specific form of the

IsoGCN layer can describe essential physical laws by acting as a spatial differential operator. Additionally, we demonstrated the computational efficiency of the proposed approach for processing graphs with up to 1M vertices, which are common in real physical simulations, as well as its capacity to produce faster prediction with the same level of accuracy compared with conventional finite element methods. Consequently, an IsoGCN can suitably replace physical simulations thanks to its power to express physical laws and faster, scalable computation. The corresponding implementation code and dataset are available online[1].

Similarly, in a follow-up study (Horie & Mitsume (2022)), we proposed a *physics-embedded neural network (PENN)*, which is a machine learning framework featuring properties 3, 4, and 5 in the above list. We built PENN based on an IsoGCN to capture physical symmetry and ensure fast prediction. Furthermore, we developed a method for considering mixed boundary conditions and modified the stacking of GNNs using a nonlinear solver, enabling the natural inclusion of global interactions in GNNs through global pooling and improving their interpretability. By conducting numerical experiments, we demonstrated the improved predictive performance of the model when dealing with Neumann boundary conditions, as well as its ability to correctly fulfill Dirichlet boundary conditions. This method displayed state-of-the-art performance compared with that of a classical, well-optimized numerical solver and a baseline machine learning model in terms of speed-accuracy trade-off. The implementation code and dataset used for the experiments are also available online[2].

1.3 Outline of Dissertation

In Chapter 2, we provide an overview of the background necessary for discussing our research. We introduce essential machine learning models, particularly GNNs, that can learn PDEs on complex shapes. In addition, we establish the concept of equivariance, which is the focus of this study, and review the basics of numerical analysis and its relationship to graphs.

[1] https://github.com/yellowshippo/isogcn-iclr2021
[2] https://github.com/yellowshippo/penn-neurips2022

Chapter 3 presents IsoGCN, our essential computationally efficient GNN model with E(n)-equivariance. First, we elaborate on the motivation for equivariance. Then, we explain the method and prove its equivariance and its relationship to numerical analysis. Finally, we report numerical experiments that demonstrate the effectiveness of IsoGCNs.

Chapter 4 discusses PENNs, which can correctly satisfy boundary conditions and global interactions based on IsoGCNs. Here, we describe the methods for handling Dirichlet and Neumann boundary conditions, and for including global interactions using a nonlinear solver. Subsequently, we demonstrate the superiority of the proposed method through numerical experiments.

Finally, in Chapter 5, we summarize the main conclusions of this dissertation and mention the limitations of the research, pointing out an interesting future direction to address them.

Chapter 2

Background

2.1 Machine Learning

In this section, we review the basics of machine learning and the essential models for learning numerical analysis.

2.1.1 Foundations of Supervised Learning

For the purposes of this study, we focus on supervised learning, which is informally defined as constructing a function that maps a given input to a given output as accurately as possible.

Supervised learning involves minimizing the error between the given target and the prediction from the machine learning model. Let $\mathbb{D}_n := \{(\boldsymbol{x}_i \in \mathcal{X}, \boldsymbol{y}_i \in \mathcal{Y})\}_{i=1}^n$ denote a given training dataset, where $\mathcal{X}$ and $\mathcal{Y}$ are the input and output spaces, respectively. A machine learning model with a set of learnable parameters θ is defined as $\boldsymbol{f}_\theta : \mathcal{X} \to \mathcal{Y}$. Training is expressed as follows:

$$\theta^* := \arg\min_\theta R_n(\theta), \tag{2.1}$$

where the training loss $R_n(\theta)$ is:

$$R_n(\theta) := \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\boldsymbol{f}_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i). \tag{2.2}$$

and $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is the *loss function*, which serves as an error scale.

Although training is performed using a training dataset, the goal of supervised learning is to obtain a model applicable to the statistical population behind the dataset, unlike a typical optimization problem where it is sufficient to obtain an optimal model for the given data. However, because evaluating a model using a population in a practical setting is not feasible, we evaluate the trained model using a test dataset $\mathbb{D}^{\text{test}}_{n^{\text{test}}} := \{(\boldsymbol{x}_i^{\text{test}} \in \mathcal{X}, \boldsymbol{y}_i^{\text{test}} \in \mathcal{Y})\}_{i=1}^{n^{\text{test}}}$, Which is different from the training dataset but sampled from the same distribution. The population loss is approximated as follows:

$$\begin{aligned} R(\theta^*) :=& E[\mathcal{L}(\boldsymbol{f}_{\theta^*}(\boldsymbol{x}), \boldsymbol{y})] && (2.3)\\ \approx& \frac{1}{n^{\text{test}}}\sum_{i=1}^{n^{\text{test}}} \mathcal{L}(\boldsymbol{f}_{\theta^*}(\boldsymbol{x}_i^{\text{test}}), \boldsymbol{y}_i^{\text{test}}), && (2.4) \end{aligned}$$

where $E[\cdot]$ is the expected value.

As an example of supervised learning, let us consider linear regression. If $\mathcal{X} = \mathcal{Y} = \mathbb{R}$, it becomes a one-dimensional linear regression, which is the simplest case. In this case, the machine learning model is expressed as:

$$\begin{aligned} f_\theta(x) &= wx + b && (2.5)\\ \theta &= (w \in \mathbb{R}, b \in \mathbb{R}). && (2.6) \end{aligned}$$

Using the least squares method, we define the loss function as

$$\mathcal{L}(y_{\text{prediction}}, y_{\text{target}}) = (y_{\text{prediction}} - y_{\text{target}})^2. \tag{2.7}$$

This can be easily generalized to higher-dimensional cases by letting

$$\mathcal{X} = \mathbb{R}^{d_{\text{in}}} \tag{2.8}$$

$$\mathcal{Y} = \mathbb{R}^{d_{\text{out}}} \tag{2.9}$$

$$\boldsymbol{f}_\theta(x) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{2.10}$$

$$\theta = (\boldsymbol{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}, \boldsymbol{b} \in \mathbb{R}^{d_{\text{out}}}), \tag{2.11}$$

where d_{in} and d_{out} are the input and output dimensions, respectively. For more information on machine learning, including supervised learning, see Bishop (2006).

2.1.2 Graph Neural Networks (GNNs)

This section provides an overview of the foundations of graph neural networks (GNNs), which are a class of neural networks designed to handle graph-structured data. GNNs were first proposed by Baskin et al. (1997); Sperduti & Starita (1997), and subsequently improved by (Gori et al., 2005; Scarselli et al., 2008). Because various data can be regarded as graphs, GNNs have a broad range of application domains such as 3D shape recognition (Fey et al., 2018; Monti et al., 2017), structural chemistry (Gilmer et al., 2017; Klicpera et al., 2020), and social network analysis (Fan et al., 2019).

2.1.2.1 Graph

A finite *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined as a tuple of a finite set of vertices (nodes) $\mathcal{V}$ and edges $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$. In general, note that the edges are *directed*, i.e., that $(u, v) \in \mathcal{E}$ does not imply $(v, u) \in \mathcal{E}$. However, in this dissertation we assume that all graphs are *undirected* (i.e., $(u, v) \in \mathcal{E}$ implies $(v, u) \in \mathcal{E}$ for all $u, v \in \mathcal{V}$) because of Newton's third law of motion, which states that every action has an equal and opposite to reaction. The set of neighboring vertices of v is defined as:

$$\mathcal{N}_v := \{u \in \mathcal{V} | (v, u) \in \mathcal{E}\}. \tag{2.12}$$

The square matrix describing the edge connectivity of a graph is called the *adjacency matrix* and defined as:

$$\boldsymbol{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|} \tag{2.13}$$

$$A_{ij} = \begin{cases} 1 & \text{if edge } (v_i, v_j) \in \mathcal{E} \\ 0 & \text{otherwise,} \end{cases} \tag{2.14}$$

where $|\mathcal{V}|$ denotes the number of vertices in the index set $J = \{1, 2, \ldots, |\mathcal{V}|\}$. Although the definition of an adjacency matrix depends on the indexing of the vertices, adjacency matrices of the same graph but with different indexing can be shown to be isomorphic using the *permutation* $\pi : J \to J$ to describe the changes in indices. Using the permutation matrix, $\boldsymbol{P} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ defined as:

$$P_{ij} = \begin{cases} 1 & \text{if } \pi(i) = j \\ 0 & \text{otherwise,} \end{cases} \tag{2.15}$$

one can show that:

$$\boldsymbol{A}' = \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\top, \tag{2.16}$$

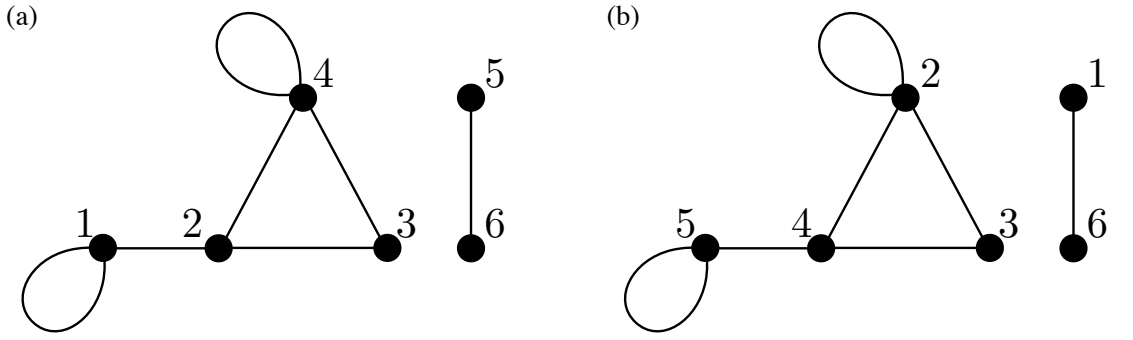
where $A'_{ij} = A_{\pi(i)\ \pi(j)}$ is the adjacency matrix with permutated indices. Figure 2.1 presents an example of a graph and its permutated representation. Because the discussion regarding permutations of graph vertex indices is well defined, in the subsequent discussions we represent vertices using an index, i.e., $v_i \mapsto i$.

The *graph Laplacian matrix* $\boldsymbol{L}$ can be defined as:

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}, \tag{2.17}$$

where $\boldsymbol{D}$ is the *degree matrix* of the graph, which is defined by:

$$D_{ij} := \begin{cases} \sum_k A_{ik} & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{2.18}$$



(c)

$$\boldsymbol{A} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \boldsymbol{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \boldsymbol{A}' = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2.1: (a) An example of a graph, (b) the same graph with permutated indices, and (c) corresponding adjacency and permutation matrices.

For the path graph with five vertices shown in Figure 2.2, the adjacency matrix is expressed as follows:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.19}$$

Thus, the graph Laplacian matrix is:

$$\boldsymbol{L} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{2.20}$$

As will be discussed in Section 2.2.4, the graph Laplacian matrix is closely related to the Laplacian operator.

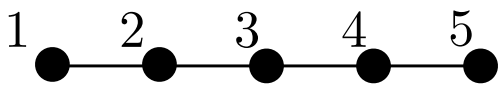


Figure 2.2: A path graph with five vertices.

A function defined at a set of vertices $\boldsymbol{f}_{\text{vertex}} : \mathcal{V} \to \mathbb{R}^N$ is called a vertex signal or *vertex feature*. Similarly, a function defined at a set of edges $\boldsymbol{f}_{\text{edge}} : \mathcal{E} \to \mathbb{R}^N$ is called an edge signal or *edge feature*. *Graph signal processing* is a research domain that deals with node and edge signals on graphs, that is, graph signals. For more details regarding graph signal processing, refer to, e.g., Ortega et al. (2018); Dong et al. (2020).

2.1.2.2 POINTWISE MLP

One of the most basic neural network models is the *multilayer perceptron* (MLP). An L-layer $\text{MLP}_L : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ is defined as a stacking of affine transformations and component-wise functions, called activation functions, as follows:

$$\text{MLP}(\boldsymbol{x}) := \sigma^{(L)} \circ \text{Affine}^{(L)} \circ \sigma^{(L-1)} \circ \text{Affine}^{(L-1)} \circ \cdots \circ \sigma^{(1)} \circ \text{Affine}^{(1)}(\boldsymbol{x}) \tag{2.21}$$

$$\text{Affine}^{(l)}(\boldsymbol{h}_l) := \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l)} + \boldsymbol{b}^{(l)} \qquad \forall l \in \{1, 2, \ldots, L\} \tag{2.22}$$

$$\boldsymbol{W}^{(l)} \in \mathbb{R}^{d^{(l+1)} \times d^{(l)}} \qquad \forall l \in \{1, 2, \ldots, L\} \tag{2.23}$$

$$\boldsymbol{b}^{(l)} \in \mathbb{R}^{d^{(l+1)}} \qquad \forall l \in \{1, 2, \ldots, L\} \tag{2.24}$$

$$\sigma^{(l)} \begin{pmatrix} \vdots \\ v_i \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ \sigma^{(l)}(v_i) \\ \vdots \end{pmatrix} \qquad \forall l \in \{1, 2, \ldots, L\}, \tag{2.25}$$

where $d^{(1)} = d_{\text{in}}$ and $d^{(L+1)} = d_{\text{out}}$, and $\boldsymbol{W}^{(l)}$, $\boldsymbol{b}^{(l)}$, and $\sigma^{(l)}$ are the *weight matrix*, *bias*, and *activation function*, respectively. An MLP is known as a universal approximator (Hornik,

1991; Cybenko, 1992; Nakkiran et al., 2021) that can approximate any continuous function if the number of hidden features $d^{(l)}$ ($l \neq 1, L$) is increased.

However, an MLP cannot handle an input with an arbitrary length by itself because the dimensions of the input are fixed. Instead, one can use a *pointwise MLP* to handle inputs with arbitrary lengths. An L-layer pointwise MLP, $\mathrm{PointwiseMLP}_L : \mathbb{R}^{|\mathcal{V}| \times d_{\mathrm{in}}} \to \mathbb{R}^{|\mathcal{V}| \times d_{\mathrm{out}}}$, is constructed by separately applying an L-layer MLP, $\mathrm{MLP}_L : \mathbb{R}^{d_{\mathrm{in}}} \to \mathbb{R}^{d_{\mathrm{out}}}$, to each point, as follows:

$$\mathrm{PointwiseMLP}_L(\boldsymbol{H}_{\mathrm{in}}) := \begin{pmatrix} \mathrm{MLP}_L(\boldsymbol{h}_{\mathrm{in},1}) \\ \mathrm{MLP}_L(\boldsymbol{h}_{\mathrm{in},2}) \\ \vdots \\ \mathrm{MLP}_L(\boldsymbol{h}_{\mathrm{in},N}) \end{pmatrix} \tag{2.26}$$

$$\boldsymbol{H}_{\mathrm{in}} = \begin{pmatrix} \boldsymbol{h}_{\mathrm{in},1} \\ \boldsymbol{h}_{\mathrm{in},2} \\ \vdots \\ \boldsymbol{h}_{\mathrm{in},|\mathcal{V}|} \end{pmatrix} \in \mathbb{R}^{|\mathcal{V}| \times d_{\mathrm{in}}}, \tag{2.27}$$

where every MLP represents an identical function. Figure 2.3 (a) presents the architecture of a pointwise MLP. It can be seen that an MLP "pointwise" is applied, resulting in the capability to incorporate an arbitrary input length.

Alternatively, a pointwise MLP is expressed as:

$$\mathrm{PointwiseMLP}_L(\boldsymbol{H}_{\mathrm{in}}) := \sigma^{(L)} \circ \mathrm{PointwiseAffine}^{(L)} \circ \cdots \circ \sigma^{(1)} \circ \mathrm{PointwiseAffine}^{(1)}(\boldsymbol{H}_{\mathrm{in}}), \tag{2.28}$$

where

$$\mathrm{PointwiseAffine}^{(l)}(\boldsymbol{H}^{(l)}) := \boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)} + \mathbf{1}_{|\mathcal{V}|}\boldsymbol{b}^{(l)} \qquad \forall l \in \{1, 2, \ldots, L\} \tag{2.29}$$

$$\boldsymbol{W}^{(l)} \in \mathbb{R}^{d^{(l+1)} \times d^{(l)}} \qquad \forall l \in \{1, 2, \ldots, L\} \tag{2.30}$$

$$\boldsymbol{b}^{(l)} \in \mathbb{R}^{1 \times d^{(l)}} \qquad \forall l \in \{1, 2, \ldots, L\} \tag{2.31}$$

$$\mathbf{1}_{|\mathcal{V}|} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \in \mathbb{R}^{\mathcal{V}\times 1}. \tag{2.32}$$

Because the trainable parameters in the model do not depend on the number of vertices, pointwise MLPs can handle arbitrary input lengths, i.e., arbitrary graphs, but they ignore the edges, that is, the connections between vertices. Nevertheless, pointwise MLPs are widely used as part of GNNs because of their simplicity.

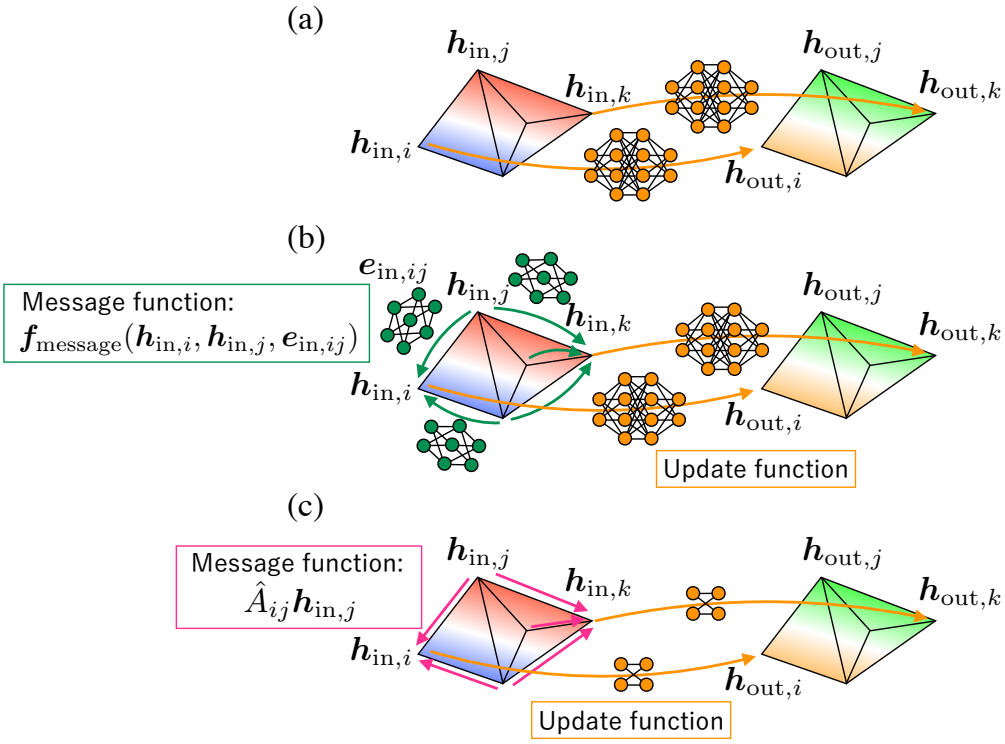


Figure 2.3: Schematic diagrams of (a) pointwise MLP, (b) MPNN, and (c) GCN.

2.1.2.3 Message Passing Neural Networks (MPNNs)

The term GNN is an umbrella denomination for any neural network that can handle graph-structured data. Although there are many GNN variants, most are unified under the concept of *message passing neural networks* (MPNNs) (Gilmer et al., 2017), which comprise two main parts: the message function $\boldsymbol{f}_{\mathrm{messaga}}$ and update function $\boldsymbol{f}_{\mathrm{update}}$. One

MPNN operation is defined as:

$$\mathrm{MPNN}(\{\boldsymbol{h}_{\mathrm{in},i}\}_{i\in\mathcal{V}}, \{\boldsymbol{e}_{\mathrm{in},ij}\}_{(i,j)\in\mathcal{E}}) := \boldsymbol{f}_{\mathrm{update}}(\boldsymbol{h}_{\mathrm{in},i}, \boldsymbol{m}_i) \tag{2.33}$$

$$\boldsymbol{m}_i := \sum_{j\in\mathcal{N}_i} \boldsymbol{f}_{\mathrm{message}}(\boldsymbol{h}_{\mathrm{in},i}, \boldsymbol{h}_{\mathrm{in},j}, \boldsymbol{e}_{\mathrm{in},ij}), \tag{2.34}$$

where $\{\boldsymbol{h}_{\mathrm{in},i}\}_{i\in\mathcal{V}}$ and $\{\boldsymbol{e}_{\mathrm{in},ij}\}_{(i,j)\in\mathcal{E}}$ are the vertex and edge features, respectively. Note that $\boldsymbol{f}_{\mathrm{message}}$ and $\boldsymbol{f}_{\mathrm{update}}$ are machine learning models usually based on neural networks.

Figure 2.3 (b) shows a schematic of an MPNN. The message function models the effect from neighboring vertices. A typical example of an update function is a pointwise MLP that predicts the state of vertices using vertex features and aggregated messages. The trainable parameters of the message and update functions independent of the number of vertices or edges, which implies that an MPNN can handle a graph with arbitrary dimensions. A single MPNN layer considers neighboring vertices as one hop away, hence, the information of vertices k-hops away can be considered by stacking k MPNN layers.

2.1.2.4 Graph Convolutional Network (GCN)

Generally, deep neural networks are used for message passing, which can incur tremendous computational cost. In contrast, the *Graph Convolutional Network* (GCN) developed by Kipf & Welling (2017) is a considerable simplification of an MPNN, that uses a linear message-passing scheme expressed as:

$$\mathrm{GCN}(\boldsymbol{H}_{\mathrm{in}}) := \mathrm{PointwiseMLP}_{L=1}(\hat{\boldsymbol{A}}\boldsymbol{H}_{\mathrm{in}}), \tag{2.35}$$

where $\hat{\boldsymbol{A}}$ denotes a renormalized adjacency matrix with self-loops and defined as:

$$\hat{\boldsymbol{A}} := \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}, \tag{2.36}$$

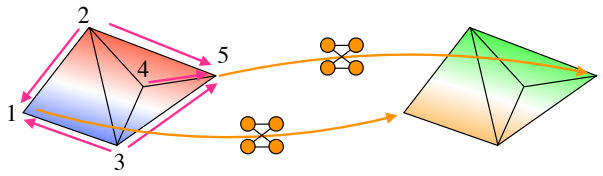
where $\tilde{\boldsymbol{A}}$ is an adjacency matrix of the graph with added self-connections and $\tilde{\boldsymbol{D}}$ is the degree matrix of $\tilde{\boldsymbol{A}}$.

The formulation of a GCN comprehends that of an MPNN, as follows:

$$\begin{aligned}\boldsymbol{m}_i = \sum_{j\in\mathcal{N}_i\cup\{i\}} \boldsymbol{f}_{\text{message}}(\boldsymbol{h}_{\text{in}}, \boldsymbol{h}_{\text{in},j}) &= \sum_{j\in\mathcal{N}_i\cup\{i\}} \hat{A}_{ij}\boldsymbol{h}_{\text{in},j} \\ &= \left[\hat{\boldsymbol{A}}\boldsymbol{H}_{\text{in}}\right]_i \qquad (2.37)\end{aligned}$$

$$\boldsymbol{f}_{\text{update}}(\boldsymbol{m}_i) = \text{MLP}_{L=1}(\boldsymbol{m}_i), \qquad (2.38)$$

Note that Equation 2.37 is derived based on the fact that $\hat{A}_{ij} = 0$ (if $j \notin \mathcal{N}_i \cup \{i\}$). From Equation 2.37, one can consider that GCNs use linearized message passing, which can accelerate their. Furthermore, if the graph is sparse, i.e., $|\mathcal{E}| \ll |\mathcal{V}|^2$, implying that the number of actual edges $|\mathcal{E}|$ is significantly smaller than the possible number of edges $|\mathcal{V}|^2$, efficient algorithms can be utilized for sparse matrix operations. Figure 2.3 (c) shows the architecture of a GCN and Figure 2.4 shows an example of GCN operation. Owing to its computational efficiency, a GCN is the basis for constructing our proposed IsoGCN, a fast machine learning model to learn physics, as presented in Chapter 3



$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \quad \tilde{\boldsymbol{D}} = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 4 \end{pmatrix} \quad \hat{\boldsymbol{A}} = \frac{1}{60}\begin{pmatrix} 20 & 4\sqrt{5} & 4\sqrt{5} & 0 & 0 \\ 4\sqrt{5} & 12 & 12 & 6\sqrt{5} & 6\sqrt{5} \\ 4\sqrt{5} & 12 & 12 & 6\sqrt{5} & 6\sqrt{5} \\ 0 & 6\sqrt{5} & 6\sqrt{5} & 15 & 15 \\ 0 & 6\sqrt{5} & 6\sqrt{5} & 15 & 15 \end{pmatrix}$$

$$\boldsymbol{h}_{\text{out},1} = [\text{GCN}(\boldsymbol{H}_{\text{in}})]_1 = \text{MLP}\left(\frac{1}{60}[20\boldsymbol{h}_{\text{in},1} + 4\sqrt{5}\boldsymbol{h}_{\text{in},2} + 4\sqrt{5}\boldsymbol{h}_{\text{in},3}]\right)$$

Figure 2.4: An example of a graph and its corresponding adjacency matrix $\boldsymbol{A}$, degree matrix $\tilde{\boldsymbol{D}}$, renormalized adjacency matrix $\hat{\boldsymbol{A}}$, and resulting output $\boldsymbol{h}_{\text{out},1}$. their can be seen that the GCN model considers information on neighboring vertices through a weighted sum determined from the graph structure.

If we consider a graph with no edges, then $A_{ij} = 0$ ($\forall i, j \in \{1, 2, \ldots, |\mathcal{V}|\}$) and $\tilde{\boldsymbol{D}} = \boldsymbol{I}_{|\mathcal{V}|}$, where $\boldsymbol{I}_{|\mathcal{V}|}$ denotes an identity matrix of size $|\mathcal{V}|$. In such case, the GCN layer becomes a pointwise MLP $\text{PointwiseMLP}_{L=1}(\boldsymbol{I}_{|\mathcal{V}|}\boldsymbol{H}_{\text{in}}) = \text{PointwiseMLP}_{L=1}(\boldsymbol{H}_{\text{in}})$. Therefore, the

GCN model can be considered a generalization of a pointwise MLP for a model-capturing graph structure.

2.1.3 EQUIVARIANCE

Equivariance is an essential concept for characterizing the predictable behavior of a function under certain transformations, such as rotation or translation. Because this is closely related to group theory, we first introduce groups and related concepts.

2.1.3.1 GROUP THEORY

A *group* is a set G with a binary operation (usually called "multiplication"), $\cdot : G \times G \to G$, that satisfies the following requirements:

$$\text{(Associativity)} \qquad \forall a, b, c \in G, \qquad (a \cdot b) \cdot c = a \cdot (b \cdot c) \tag{2.39}$$

$$\text{(Identity element)} \qquad \exists e \text{ s.t. } \forall a \in G, \qquad e \cdot a = a \cdot e = a \tag{2.40}$$

$$\text{(Inverse element)} \qquad \forall a \in G, \exists b \in G \text{ s.t.} \qquad a \cdot b = b \cdot a = e. \tag{2.41}$$

For a group G and a set X, a (left) *group action* is a function $\cdot : G \times X \to X$, which satisfies the following conditions:

$$\text{(Identity)} \qquad \forall x \in X, \qquad e \cdot x = x \tag{2.42}$$

$$\text{(Compatibility)} \qquad \forall a, b \in G, \forall x \in X, \qquad a \cdot (b \cdot x) = (a \cdot b) \cdot x, \tag{2.43}$$

where e is the identity element of the group. We denote $\alpha(a, x) := a \cdot x$ when we must clarify that the operation is a group action.

Groups appear in various fields, such as physics, engineering, and computer science; next, we provide a few examples of groups and their actions. A first example is the *general linear group* $\mathrm{GL}(n)$, the set of all n-dimensional invertible matrices. One can confirm $\mathrm{GL}(n)$ is a group because the multiplication of matrices is associative, the identity matrix is in $\mathrm{GL}(n)$, and by definition an inverse matrix always exists for a given element in $\mathrm{GL}(n)$.

Another example is the *orthogonal group* $\mathrm{O}(n)$, the set of n-dimensional orthogonal matrices representing rotation and reflection, which also satisfies the group requirements.

In addition, one can consider the multiplication between an element of $\mathrm{O}(n)$ and an n-dimensional vector $\boldsymbol{x} \in \mathbb{R}^n$. Such multiplication satisfies the definition of group action as well. Furthermore, a rank-2 tensor $\boldsymbol{T} \in \mathbb{R}^{n \times n}$ can be transformed into an orthogonal matrix $\boldsymbol{U} \in \mathrm{O}(n)$ By computing $\alpha(\boldsymbol{U}, \boldsymbol{T}) = \boldsymbol{U}\boldsymbol{T}\boldsymbol{U}^\top$, Which is also a group action. Therefore, the concrete form of a group action might differ depending on the set on which the group acts.

Two more examples of groups are the *symmetric group* S_n, a group of permutations, and the *Euclidean group* $\mathrm{E}(n)$, which is a group of isometric transformations, namely, translation, rotation, and reflection. In particular, $\mathrm{E}(n)$ plays an essential role in developing neural PDE solvers because most physical phenomena occur in the Euclidean space, with its essence remaining the same under $\mathrm{E}(n)$ transformations.

2.1.3.2 Equivariant Model

A function $f : X \to Y$ is said to be *G-equivariant* when:

$$\forall g \in G, \forall x \in X, f(g \cdot x) = g \cdot f(x), \tag{2.44}$$

assuming that the group G acts on both X and Y. The concept of equivariance is also explained by the following commutative diagram:

$$\begin{array}{ccc} X & \xrightarrow{g\cdot} & X \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle f} \\ Y & \xrightarrow{g\cdot} & Y \end{array}$$

In particular, when:

$$\forall g \in G, f(g \cdot x) = f(x), \tag{2.45}$$

f is said to be *G-invariant*, and the corresponding commutative diagram is as follows:

$$\begin{array}{ccc} X & \xrightarrow{g\cdot} & X \\ & \searrow^{f} & \downarrow{\scriptstyle f} \\ & & Y \end{array}$$

This invariance is a special case of equivariance because $g \cdot x = x$ qualifies as a group action (trivial group action).

Most numerical analysis schemes and models for physical simulations have equivariance. The principle of material objectivity (Ignatieff, 1996), which is similar to equivariance, is considered essential for constitutive laws. For instance, the tensor product between two rank-1 tensors $\boldsymbol{f}_{\text{prod}} : \mathbb{R}^n \times \mathbb{R}^n \ni (\boldsymbol{v}, \boldsymbol{u}) \mapsto \boldsymbol{v} \otimes \boldsymbol{u} \in \mathbb{R}^{n \times n}$ is $\mathrm{O}(n)$-equivariant because, for any orthogonal matrix $\boldsymbol{U}$:

$$\begin{aligned}
[\boldsymbol{f}_{\text{prod}}(\alpha(\boldsymbol{U}, \boldsymbol{v}), \alpha(\boldsymbol{U}, \boldsymbol{u}))]_{ij} &= [\boldsymbol{U}\boldsymbol{v} \otimes \boldsymbol{U}\boldsymbol{u}]_{ij} \\
&= [\boldsymbol{U}\boldsymbol{v}]_i [\boldsymbol{U}\boldsymbol{u}]_j \\
&= \sum_{kl} U_{ik} v_k U_{jl} u_l \\
&= \sum_{kl} U_{ik} v_k u_l U^{\top}_{lj} \\
&= \left[\boldsymbol{U}\boldsymbol{v} \otimes \boldsymbol{u}\boldsymbol{U}^{\top}\right]_{ij} \\
&= [\alpha(\boldsymbol{U}, \boldsymbol{f}_{\text{prod}}(\boldsymbol{v}, \boldsymbol{u}))]_{ij}\,,
\end{aligned} \tag{2.46}$$

satisfying the definition of equivariance (Equation 2.44). The squared norm operator $f_{\text{norm}} : \mathbb{R}^n \ni \boldsymbol{v} \mapsto \|\boldsymbol{v}\|^2 \in \mathbb{R}$ is $\mathrm{O}(n)$-invariant because

$$\begin{aligned}
f_{\text{norm}}(\alpha(\boldsymbol{U}, \boldsymbol{v})) &= \|\boldsymbol{U}\boldsymbol{v}\|^2 \\
&= (\boldsymbol{U}\boldsymbol{v}) \cdot (\boldsymbol{U}\boldsymbol{v}) \\
&= \sum_{ikl} U_{ik} v_k U_{il} v_l \\
&= \sum_{ikl} U^{\top}_{li} U_{ik} v_k v_l \\
&= \sum_{kl} \delta_{lk} v_k v_l \\
&= \sum_{k} v_k v_k \\
&= \|\boldsymbol{v}\|^2 \\
&= f_{\text{norm}}(\boldsymbol{v}),
\end{aligned} \tag{2.47}$$

which satisfies the definition of invariance (Equation 2.45). Here, δ_{lk} is the *Kronecker delta*, defined as:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{2.48}$$

In addition to $\mathrm{O}(n)$, a symmetric group S_n is also worth considering because it corresponds to the permutation of the vertex indices. In numerical analysis, we choose arbitrary indexing for the nodes and elements in the meshes. Therefore, permutation equivariance is an essential indicator for a preferable numerical analysis scheme[1]. We can demonstrate a GCN layer operation (Equation 2.35) is permutation equivariant for all permutation matrices as follows: $\boldsymbol{P}$,

$$\begin{aligned} \mathrm{GCN}(\alpha(\boldsymbol{P}, \boldsymbol{H})) &= \mathrm{PointwiseMLP}(\alpha(\boldsymbol{P}, \hat{\boldsymbol{A}}\boldsymbol{H})) \\ &= \mathrm{PointwiseMLP}(\boldsymbol{P}\hat{\boldsymbol{A}}\boldsymbol{P}^\top\boldsymbol{P}\boldsymbol{H})) \\ &= \mathrm{PointwiseMLP}(\boldsymbol{P}\hat{\boldsymbol{A}}\boldsymbol{H})) \\ &= \boldsymbol{P}\,\mathrm{PointwiseMLP}(\hat{\boldsymbol{A}}\boldsymbol{H})) \\ &= \alpha(\boldsymbol{P}, \mathrm{GCN}(\boldsymbol{H})). \end{aligned} \tag{2.49}$$

We use the fact that any permutation matrix is orthogonal, i.e., $\boldsymbol{P}^\top = \boldsymbol{P}^{-1}$, and that all pointwise MLP layers are trivially permutation equivariant.

Group equivariant convolutional neural networks (CNNs) were first proposed by Cohen & Welling (2016) for discrete groups. Subsequent studies have categorized such networks as continuous groups (Cohen et al., 2018), three-dimensional data (Weiler et al., 2018), and general manifolds (Cohen et al., 2019). These methods are based on CNNs; thus, they cannot directly handle mesh or point cloud data structures. Specifically, 3D steerable CNNs (Weiler et al., 2018) which use voxels (regular grids) and are deemed relatively easy to handle, are inefficient because they represent both occupied and empty parts of an object (Ahmed et al., 2018). In addition, a voxelized object tends to lose smoothness of

[1]However, several schemes are not permutation equivariant, e.g., one iteration in the successive over-relaxation (SOR) method. Nevertheless, we can assume that the entire SOR process is nearly permutation-equivariant if it converges to an accurate solution.

its shape, which can lead to a drastically different behavior in a physical simulation, as is typically observed in heat analyses and computational fluid dynamics.

Thomas et al. (2018); Kondor (2018) discussed how to provide rotation equivariance to point clouds. Specifically, Thomas et al. (2018) proposed a tensor field network (TFN), which is a point-cloud-based rotation and translation equivariant neural network, whose layer can be written as:

$$\tilde{\boldsymbol{H}}^{(l)}_{\mathrm{out},i} = \mathrm{TFN}_l(\{\tilde{\boldsymbol{H}}^{(k)}_{\mathrm{in},i}\}_{k\geq 0}) = w^{ll}\,\tilde{\boldsymbol{H}}^{(l)}_{\mathrm{in},i} + \sum_{k\geq 0}\sum_{j\neq i} \boldsymbol{W}^{lk}(\boldsymbol{x}_j - \boldsymbol{x}_i)\,\tilde{\boldsymbol{H}}^{(k)}_{\mathrm{in},j} \tag{2.50}$$

$$\boldsymbol{W}^{lk}(\boldsymbol{x}) = \sum_{J=|k-l|}^{k+l} \phi^{lk}_J(\|\boldsymbol{x}\|) \sum_{m=-J}^{J} Y_{Jm}(\boldsymbol{x}/\|\boldsymbol{x}\|)\boldsymbol{Q}^{lk}_{Jm}, \tag{2.51}$$

where $\tilde{\boldsymbol{H}}^{(l)}_{\mathrm{in},i}$ ($\tilde{\boldsymbol{H}}^{(l)}_{\mathrm{out},i}$): is a type-$l$ input (output) features at the ith vertex, $\phi^{lk}_J : \mathbb{R}_{\geq 0} \to \mathbb{R}$ is a trainable function, Y_{Jm} is the mth component of the Jth spherical harmonic, and $\boldsymbol{Q}^{lk}_{Jm}$ is the Clebsch-Cordan coefficient. The SE(3)-Transformer (Fuchs et al., 2020) is a TFN variant with self-attention. Dym & Maron (2020) showed that both the TFN and SE(3)-Transformer are universal in terms of translation, rotation, and permutation equivariance.

$\mathrm{E}(n)$-equivariance is essential for solving 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 into a simple neural network and a CNN to predict flow phenomena. Both studies showed that the predictive and generalization performance improved due to equivariance.

2.2 Numerical Analysis

In this section, we review the foundations of PDEs to clarify the problems we aim to solve and introduce related works in which machine learning models are used to solve PDEs.

2.2.1 Partial Differential Equations (PDEs) with Boundary Conditions

The general form of the spatiotemporal PDEs for a field, $\boldsymbol{u} : (0, T) \times \Omega \to \mathbb{R}^d$, of a d-dimensional physical quantity defined in an n-dimensional domain, $\Omega \subset \mathbb{R}^n$, can be

expressed as follows:

$$\frac{\partial \boldsymbol{u}}{\partial t}(t, \boldsymbol{x}) = \mathcal{D}(\boldsymbol{u})(t, \boldsymbol{x}) \qquad (t, \boldsymbol{x}) \in (0, T) \times \Omega \tag{2.52}$$

$$\boldsymbol{u}(t = 0, \boldsymbol{x}) = \hat{\boldsymbol{u}}_0(\boldsymbol{x}) \qquad \boldsymbol{x} \in \Omega \tag{2.53}$$

$$\boldsymbol{u}(t, \boldsymbol{x}) = \hat{\boldsymbol{u}}(t, \boldsymbol{x}) \qquad (t, \boldsymbol{x}) \in (0, T) \times \partial\Omega_{\text{Dirichlet}} \tag{2.54}$$

$$\hat{\boldsymbol{f}}(\nabla \boldsymbol{u}(t, \boldsymbol{x}), \boldsymbol{n}(\boldsymbol{x})) = \boldsymbol{0} \qquad (t, \boldsymbol{x}) \in (0, T) \times \partial\Omega_{\text{Neumann}}, \tag{2.55}$$

where $\partial\Omega_{\text{Dirichlet}}$ and $\partial\Omega_{\text{Neumann}}$ are mixed *Dirichlet* and *Neumann* boundary conditions, respectively, such that $\partial\Omega_{\text{Dirichlet}} \cap \partial\Omega_{\text{Neumann}} = \emptyset$ and $\partial\Omega_{\text{Dirichlet}} \cup \partial\Omega_{\text{Neumann}} = \partial\Omega$, $\partial\Omega$ denotes the boundary of Ω, $\hat{\cdot}$ is a known function, $\mathcal{D}$ is a known nonlinear differential operator, which can be nonlinear and contains spatial differential operators, and $\boldsymbol{n}(\boldsymbol{x})$ denotes the normal vector at $\boldsymbol{x} \in \partial\Omega$. Equation 2.54 is called the Dirichlet boundary condition, where the value of $\partial\Omega_{\text{Dirichlet}}$ is set as a constraint, whereas Equation 2.55 corresponds to the Neumann boundary condition, where the value of the derivative $\boldsymbol{u}$ in the direction of $\boldsymbol{n}$ is set to $\partial\Omega_{\text{Neumann}}$ rather than $\boldsymbol{u}$. $\boldsymbol{u}$ is the solution of the (initial) boundary value problem when it satisfies Equations 2.52 – 2.55.

Equation 2.52 may represent various types of PDEs. For instance, in the case of the heat equation:

$$\mathcal{D}_{\text{heat}}(u) = c\nabla \cdot \nabla u, \tag{2.56}$$

where u is the temperature field ($d = 1$) and c is the diffusion coefficient. For an incompressible Navier–Stokes equations:

$$\mathcal{D}_{\text{NS}}(\boldsymbol{u}) = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \frac{1}{\text{Re}}\nabla \cdot \nabla \boldsymbol{u} - \nabla p, \tag{2.57}$$

where $\nabla \cdot \boldsymbol{u} = 0$ expresses the incompressible condition, $\boldsymbol{u}$ denotes the flow velocity field, p is the pressure field, and Re denotesthe Reynolds number.

2.2.2 DISCRETIZATION

PDEs must be defined in a continuous space for the differentials to be meaningful. Discretization can be applied to both space and time to enable computers to easily solve the PDE.

In the numerical analysis of complex-shaped domains, we commonly use *meshes* (discretized shape data), which can be regarded as a graph, as shown in Figure 2.5. We denote the position of the ith vertex as $\boldsymbol{x}_i$ and the value of a function f, g, ... at $\boldsymbol{x}_i$ as f_i g_i, ... Therefore, $\{f_i\}_{i\in\mathcal{V}}$, $\{g_i\}_{i\in\mathcal{V}}$, and ... are the vertex features[2]. For concrete examples of spatial discretization, see Section 2.2.4.

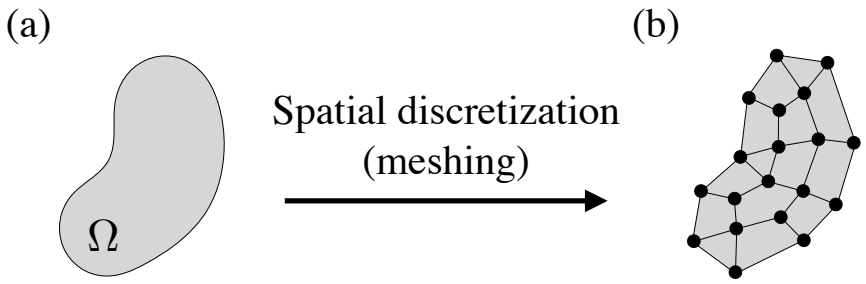


Figure 2.5: Examples of (a) a domain Ω and (b) a mesh representing the corresponding discretized domain.

One of the simplest methods to discretize time is the *explicit Euler method* which is formulated as:

$$\boldsymbol{u}(t+\Delta t, \boldsymbol{x}_i) \approx \boldsymbol{u}(t, \boldsymbol{x}_i) + \mathcal{D}(\boldsymbol{u})(t, \boldsymbol{x}_i)\Delta t, \tag{2.58}$$

where $\boldsymbol{u}(t, \boldsymbol{x}_i)$ is updated via a small increment $\mathcal{D}(\boldsymbol{u})(t, \boldsymbol{x}_i)\Delta t$. Another way to discretize time is the *implicit Euler method* formulated as:

$$\boldsymbol{u}(t+\Delta t, \boldsymbol{x}_i) \approx \boldsymbol{u}(t, \boldsymbol{x}_i) + \mathcal{D}(\boldsymbol{u})(t+\Delta t, \boldsymbol{x}_i)\Delta t, \tag{2.59}$$

[2]Strictly speaking, the 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.

which solves Equation 2.59 rather than simply updating the variables to ensure that the original PDE is numerically satisfied. The equation can be viewed as a nonlinear optimization problem by formulating it as:

$$\boldsymbol{r}(\boldsymbol{v}) := \boldsymbol{v} - \boldsymbol{u}(t, \cdot) - \mathcal{D}(\boldsymbol{v})\Delta t \tag{2.60}$$

$$\mathrm{Solve}_{\boldsymbol{v}}\ \boldsymbol{r}(\boldsymbol{v})(\boldsymbol{x}_i) = \mathbf{0},\ \forall i \in \{1, \ldots, |\mathcal{V}|\}, \tag{2.61}$$

where $\boldsymbol{r} : (\Omega \to \mathbb{R}^d) \to (\Omega \to \mathbb{R}^d)$ is the operator of the residual vector of the discretized PDE. Since $\boldsymbol{r}$ is a map from functions $\Omega \to \mathbb{R}^d$ to functions $\Omega \to \mathbb{R}^d$, $\boldsymbol{r}(\boldsymbol{v}) : \Omega \to \mathbb{R}^d$ is also a function. Therefore:

$$\boldsymbol{r}(\boldsymbol{v})(\boldsymbol{x}_i) = \boldsymbol{v}(\boldsymbol{x}_i) - \boldsymbol{u}(t, \boldsymbol{x}_i) - \mathcal{D}(\boldsymbol{v})(\boldsymbol{x}_i)\Delta t \in \mathbb{R}^d \tag{2.62}$$

corresponds to the error in the current numerical solution $\boldsymbol{v}$ at $\boldsymbol{x}_i$. If $\boldsymbol{r}(\boldsymbol{v})(\boldsymbol{x}_i) = \mathbf{0}$, $\boldsymbol{v}$ satisfies the discretized equation at $\boldsymbol{x}_i$. Here, by letting $\boldsymbol{V} = (\boldsymbol{v}(\boldsymbol{x}_1)^\top, \boldsymbol{v}(\boldsymbol{x}_2)^\top, \ldots, \boldsymbol{v}(\boldsymbol{x}_{|\mathcal{V}|})^\top)^\top \in \mathbb{R}^{d|\mathcal{V}|}$ and $\boldsymbol{U}(t) = (\boldsymbol{u}(t, \boldsymbol{x}_1)^\top, \boldsymbol{u}(t, \boldsymbol{x}_2)^\top, \ldots, \boldsymbol{u}(t, \boldsymbol{x}_{|\mathcal{V}|})^\top)^\top \in \mathbb{R}^{d|\mathcal{V}|}$, Equation 2.60 and Equation 2.61 become:

$$\boldsymbol{R}(\boldsymbol{V}) := \boldsymbol{V} - \boldsymbol{U}(t) - \mathcal{D}(\boldsymbol{V})\Delta t \in \mathbb{R}^{d|\mathcal{V}|} \tag{2.63}$$

$$\mathrm{Solve}_{\boldsymbol{V}}\ \boldsymbol{R}(\boldsymbol{V}) = \mathbf{0}. \tag{2.64}$$

The solution to Equation 2.64 corresponds to $\boldsymbol{U}(t + \Delta t) = (\boldsymbol{u}(t + \Delta t, \boldsymbol{x}_1)^\top, \boldsymbol{u}(t + \Delta t, \boldsymbol{x}_2)^\top, \ldots, \boldsymbol{u}(t + \Delta t, \boldsymbol{x}_{|\mathcal{V}|})^\top)^\top \in \mathbb{R}^{d|\mathcal{V}|}$.

2.2.3 Nonlinear Solver and Optimization

2.2.3.1 Basic Formula for Iterative Methods

Because Equation 2.64 can be a nonlinear and high-dimensional problem, there is no general formula for solving it. A common method to obtain an approximate solution is to

apply an iterative method to a linearized system, such as:

$$\boldsymbol{V}^{[0]} = \boldsymbol{U}(t) \tag{2.65}$$

$$\boldsymbol{V}^{[i+1]} = \boldsymbol{V}^{[i]} + \Delta\boldsymbol{V}^{[i]}, \tag{2.66}$$

where $\Delta\boldsymbol{V}^{[i]}$ is an unknown update of the approximate solution. The first-order approximation can be applied to obtain the update, as follows:

$$\boldsymbol{R}(\boldsymbol{V}^{[i]} + \Delta\boldsymbol{V}^{[i]}) \approx \boldsymbol{R}(\boldsymbol{V}^{[i]}) + \nabla_{\boldsymbol{V}} \otimes \boldsymbol{R}(\boldsymbol{V}^{[i]})\Delta\boldsymbol{V}^{[i]} = \boldsymbol{0}, \tag{2.67}$$

where $\nabla_{\boldsymbol{V}} \otimes \boldsymbol{R} \in \mathbb{R}^{d|\mathcal{V}| \times d|\mathcal{V}|}$ denotes the Jacobian matrix of $\boldsymbol{R}$ with respect to $\boldsymbol{V}$. Instead of using Equation 2.61, we can iteratively solve Equation 2.67.

If a function $\phi : \mathbb{R}^{d|\mathcal{V}|} \to \mathbb{R}$ satisfying $\nabla_{\boldsymbol{V}}\phi = \boldsymbol{R}$ exists, solving Equation 2.64 corresponds to the optimization of ϕ in an $(d|\mathcal{V}|)$-dimensional space, where $|\mathcal{V}|$ denotes the number of vertices in the considered mesh. Therefore, the implicit Euler method is closely related to optimization in a high-dimensional space. From this viewpoint, the Jacobian matrix $\nabla_{\boldsymbol{V}} \otimes \boldsymbol{R}$ corresponds to the Hessian matrix $\nabla_{\boldsymbol{V}} \otimes \nabla_{\boldsymbol{V}}\phi$. However, it should be noted that the Hessian matrix is always symmetric, which is not always the case for the Jacobian matrix.

2.2.3.2 Newton–Raphson Method and Quasi-Newton Method

The *Newton–Raphson method* solves Equation 2.67 as follows:

$$\Delta\boldsymbol{V}^{[i]} = -\left[\nabla_{\boldsymbol{V}} \otimes \boldsymbol{R}(\boldsymbol{V}^{[i]})\right]^{-1}\boldsymbol{R}(\boldsymbol{V}^{[i]}), \tag{2.68}$$

which requires solving a linear system with a large number of degrees of freedom, $(d|\mathcal{V}|)$. Solving such a large system of linear equations occasionally requires considerable computational resources and time. To address this issue, *quasi-Newton methods* approximate the inverse of the Jacobian matrix using a matrix $\boldsymbol{H}^{[i]}$ to obtain:

$$\Delta\boldsymbol{V}^{[i]} \approx -\boldsymbol{H}^{[i]}\boldsymbol{R}(\boldsymbol{V}^{[i]}). \tag{2.69}$$

Various methods can be used to initialize, compute, and update $\boldsymbol{H}^{[i]}$. A key concern in quasi-Newton methods is their massive memory consumption because $\boldsymbol{H}^{[i]}$ could be dense even if $\nabla_{\boldsymbol{V}} \otimes \boldsymbol{R}$ is sparse. Thus, a lot of effort has been dedicated to reducing the memory demand of this method, as in Liu & Nocedal (1989).

2.2.3.3 GRADIENT DESCENT METHOD

The *gradient descent* method implements yet another approximation, as follows:

$$\Delta \boldsymbol{V}^{[i]} \approx -\alpha^{[i]} \boldsymbol{R}(\boldsymbol{V}^{[i]}), \tag{2.70}$$

where $\alpha^{[i]} \in \mathbb{R}$ is a scalar that controls the update magnitude. The approximation has no error when all eigenvalues λ_i ($i \in 1, \ldots, d|\mathcal{V}|$) are the same, i.e., $\lambda_i = \lambda$, because:

$$\nabla_{\boldsymbol{V}} \otimes \boldsymbol{R}(\boldsymbol{V}^{[i]}) = \boldsymbol{Q} \begin{pmatrix} \lambda & & \\ & \ddots & \\ & & \lambda \end{pmatrix} \boldsymbol{Q}^{-1} \tag{2.71}$$

$$= \boldsymbol{Q} \lambda \boldsymbol{I}_{d|\mathcal{V}|} \boldsymbol{Q}^{-1} \tag{2.72}$$

$$= \lambda \boldsymbol{I}_{d|\mathcal{V}|}, \tag{2.73}$$

where $\boldsymbol{Q}$ where is the eigenvectors matrix. Thus, by letting $\alpha^{[i]} = 1/\lambda$, we can show that Equations 2.68 and 2.70 are the same. In contrast, if the eigenvalues are not identical and broadly distributed, the gradient descent approximation introduces some error. This fact is reasonable because such a situation corresponds to a linear system with a large condition number for the matrix $\nabla_{\boldsymbol{V}} \boldsymbol{R}(\boldsymbol{V}^{[i]})$ and, hence, constitutes a challenging problem.

The update using gradient descent is expressed as:

$$\boldsymbol{V}^{[i+1]} = \boldsymbol{V}^{[i]} - \alpha^{[i]} \boldsymbol{R}(\boldsymbol{V}^{[i]}). \tag{2.74}$$

This method is termed gradient descent because $\boldsymbol{R}(\boldsymbol{V}^{[i]})$ corresponds to the "gradient", and the equation is updated to reduce the error.

$\alpha^{[i]}$ is typically determined using line search. However, owing to the high computational cost of this search, $\alpha^{[i]}$ can be fixed to a small value α, which corresponds to the explicit Euler method with a time step size $\alpha\Delta t$ because

$$\begin{aligned}\boldsymbol{V}^{[i+1]} &= \boldsymbol{V}^{[i]} - \alpha\boldsymbol{R}(\boldsymbol{V}^{[i]}) && (2.75)\\ &= \boldsymbol{V}^{[i]} - \alpha\left[\boldsymbol{V}^{[i]} - \boldsymbol{U}(t) - \mathcal{D}(\boldsymbol{V}^{[i]})\Delta t\right] && (2.76)\\ &= (1-\alpha)\boldsymbol{V}^{[i]} + \alpha\boldsymbol{U}(t) + \mathcal{D}(\boldsymbol{V}^{[i]})\alpha\Delta t. && (2.77)\end{aligned}$$

If we explicitly write the first few steps:

$$\begin{aligned}\boldsymbol{V}^{[0]} &= \boldsymbol{U}(t) && (2.78)\\ \boldsymbol{V}^{[1]} &= (1-\alpha)\boldsymbol{V}^{[0]} + \alpha\boldsymbol{U}(t) + \mathcal{D}(\boldsymbol{V}^{[0]})\alpha\Delta t && (2.79)\\ &= \boldsymbol{U}(t) + \mathcal{D}(\boldsymbol{U}(t))\alpha\Delta t, && (2.80)\end{aligned}$$

obtaining the same update scheme as that in Equation 2.58. For more information regarding optimization, including quasi-Newton methods and gradient descent, see, e.g., Luenberger et al. (1984).

2.2.3.4 Barzilai–Borwein Method

Barzilai & Borwein (1988) suggested another simple, yet effective, way to determine the step size $\alpha^{[i]}$ in the gradient-descent method by using a two-point approximation of the secant equation underlying the quasi-Newton method. Using this method, we can derive the step size for the current state as:

$$\alpha^{[i]} \approx \alpha^{[i]}_{\mathrm{BB}} = \frac{\left[\boldsymbol{V}^{[i]} - \boldsymbol{V}^{[i-1]}\right] \cdot \left[\boldsymbol{R}(\boldsymbol{V}^{[i]}) - \boldsymbol{R}(\boldsymbol{V}^{[i-1]})\right]}{\left[\boldsymbol{R}(\boldsymbol{V}^{[i]}) - \boldsymbol{R}(\boldsymbol{V}^{[i-1]})\right] \cdot \left[\boldsymbol{R}(\boldsymbol{V}^{[i]}) - \boldsymbol{R}(\boldsymbol{V}^{[i-1]})\right]}, \tag{2.81}$$

We now derive Equation 2.81.

First, to avoid using future information, we assume that

$$\Delta\boldsymbol{V}^{[i-1]} = \boldsymbol{V}^{[i]} - \boldsymbol{V}^{[i-1]} \approx -\alpha^{[i]}\boldsymbol{R}(\boldsymbol{V}^{[i-1]}), \tag{2.82}$$

instead of using Equation 2.70, which contains the state at a future step $(i+1)$. Equation 2.82 implies:

$$\nabla_{\boldsymbol{V}} \otimes \boldsymbol{R}(\boldsymbol{V}^{[i-1]}) \approx \frac{1}{\alpha^{[i]}}. \tag{2.83}$$

By substituting Equation 2.83 into Equation 2.67 and replacing i with $(i-1)$, we obtain:

$$\begin{aligned} \boldsymbol{R}(\boldsymbol{V}^{[i]}) &\approx \boldsymbol{R}(\boldsymbol{V}^{[i-1]}) + \frac{1}{\alpha^{[i]}} \Delta \boldsymbol{V}^{[i-1]} \\ \Delta \boldsymbol{V}^{[i-1]} - \alpha^{[i]} \Delta \boldsymbol{R}^{[i-1]} &\approx \boldsymbol{0}, \end{aligned} \tag{2.84}$$

where $\Delta \boldsymbol{R}^{[i-1]} = \boldsymbol{R}(\boldsymbol{V}^{[i]}) - \boldsymbol{R}(\boldsymbol{V}^{[i-1]})$, We want to find a good $\alpha^{[i]}$ that best satisfies Equation 2.84 in terms of least squares. Thus, we obtain $\alpha^{[i]}_{\mathrm{BB}}$ as follows:

$$\alpha^{[i]}_{\mathrm{BB}} := \arg\min_{\alpha} \mathcal{L}^{[i]}(\alpha) \tag{2.85}$$

$$\mathbb{R} \ni \mathcal{L}^{[i]}(\alpha) := \frac{1}{2} \left\| \Delta \boldsymbol{V}^{[i-1]} - \alpha \Delta \boldsymbol{R}^{[i-1]} \right\|^2 \tag{2.86}$$

Because of the convexity of the problem, it is sufficient to find an α that satisfies:

$$\left. \frac{d\mathcal{L}^{[i]}}{d\alpha} \right|_{\alpha^{[i]}_{\mathrm{BB}}} = \left(\Delta \boldsymbol{v}^{[i-1]} - \alpha^{[i]}_{\mathrm{BB}} \Delta \boldsymbol{R}^{[i-1]} \right) \cdot \left(-\Delta \boldsymbol{R}^{[i-1]} \right) = 0. \tag{2.87}$$

Using the linearity of the inner product, we obtain:

$$-\Delta \boldsymbol{v}^{[i-1]} \cdot \Delta \boldsymbol{R}^{[i-1]} + \alpha^{[i]}_{\mathrm{BB}} \Delta \boldsymbol{R}^{[i-1]} \cdot \Delta \boldsymbol{R}^{[i-1]} = 0,$$

therefore,

$$\alpha^{[i]}_{\mathrm{BB}} = \frac{\Delta \boldsymbol{v}^{[i-1]} \cdot \Delta \boldsymbol{R}^{[i-1]}}{\Delta \boldsymbol{R}^{[i-1]} \cdot \Delta \boldsymbol{R}^{[i-1]}}. \tag{2.88}$$

Equation 2.88 is equivalent to Equation 2.81.

As can be seen, the derivation above aims to establish an $\alpha^{[i]}_{\mathrm{BB}}$ that satisfies Equation 2.84 as closely as possible for all vertices and all feature components. This means that $\alpha^{[i]}_{\mathrm{BB}}$ contains global information because it considers all vertices, making the inclu-

sion of global interactions possible. Additionally, $\alpha_{\mathrm{BB}}^{[i]}$ is $\mathrm{E}(n)$-invariant because it is scalar that is independent of coordinates. Therefore, $\alpha_{\mathrm{BB}}^{[i]}$ is suitable for realizing efficient PDE solvers with $\mathrm{E}(n)$-equivariance. Owing to its satisfactory balance between low computational cost and accuracy, the Barzilai–Borwein method is adopted to develop the neural nonlinear solver presented in Chapter 4

2.2.4 Numerical Analysis from a Graph Representation View

In this section, we provide an overview of several numerical analysis methods and discuss how they are related to graphs. In particular, we see that the discretized representation of spatial differentiation is closely related to graphs. For simplicity, we consider the heat equation $\mathcal{D} = c\nabla \cdot \nabla$. However, the same discussion holds for other PDEs.

2.2.4.1 Finite Difference Method

The *finite difference method* (FDM) is one of the most basic numerical analysis schemes. This method is typically applied to structured grids, where the space is discretized using lines (1D), squares (2D), or cubes (3D), as shown in Figure 2.6.

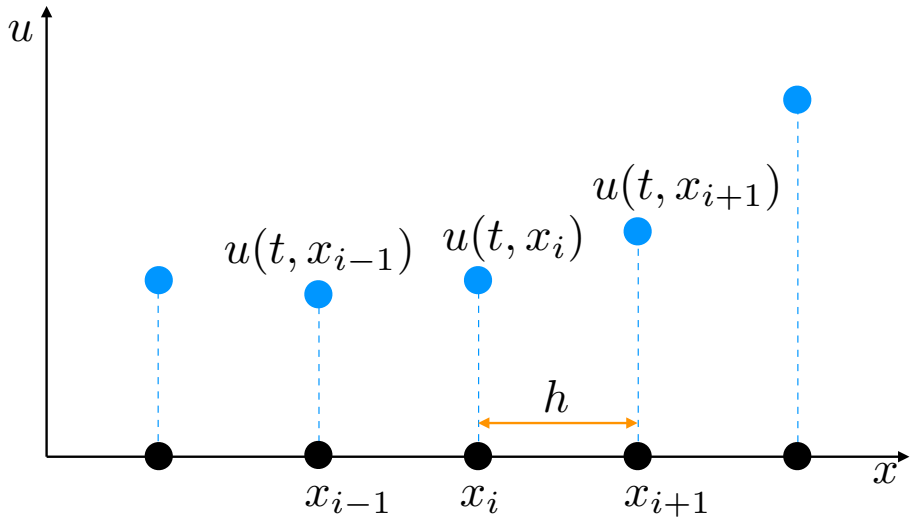


Figure 2.6: An example of a 1D u field spatially discretized using FDM.

In FDM, the gradient operator can be expressed as:

$$\frac{\partial}{\partial x}u(t,x) \approx \frac{u(t,x+h)-u(t,x)}{h}, \tag{2.89}$$

where h denotes the step size of the spatial discretization. The Laplacian operator is computed as follows:

$$\begin{aligned}\nabla \cdot \nabla u(t,x) &\approx \nabla \cdot \frac{u(t,x+h)-u(t,x)}{h} \\ &\approx \frac{1}{h}\frac{[u(t,x+h)-u(t,x)]-[u(t,x)-u(t,x-h)]}{h} \\ &= \frac{1}{h^2}\left[u(t,x+h)+u(t,x-h)-2u(t,x)\right].\end{aligned} \tag{2.90}$$

If the vertex positions are denoted using indices as follows:

$$\begin{aligned}x_{i+1} &= x+h \\ x_i &= x \\ x_{i-1} &= x-h.\end{aligned}$$

The spatially discretized heat equation becomes

$$\frac{\partial}{\partial t}u(t,x_i) = \frac{c}{h^2}\left[u(t,x_{i-1})-2u(t,x_i)+u(t,x_{i+1})\right]. \tag{2.91}$$

This expression involves interactions between vertices, that is, edge connectivity, implying a graphical structure. By using a matrix form, we can write:

$$\frac{\partial}{\partial t}\begin{pmatrix}\vdots\\ u(t,x_i)\\ \vdots\end{pmatrix} = -\frac{c}{h^2}\begin{pmatrix}\ddots & & & & \\ \cdots & -1 & 2 & -1 & \cdots \\ & & & & \ddots\end{pmatrix}\begin{pmatrix}\vdots\\ u(t,x_{i-1})\\ u(t,x_i)\\ u(t,x_{i+1})\\ \vdots\end{pmatrix}. \tag{2.92}$$

Note that the matrix appearing on the right-hand side has the same form as the Laplacian graph matrix, computed in Equation 2.20, meaning that the Laplacian operator corresponds

to the Laplacian graph matrix in a spatially discretized setting[3]. By denoting Equation 2.92 as

$$\frac{\partial}{\partial t}\boldsymbol{U}(t) = -\frac{c}{h^2}\boldsymbol{L}_{\mathrm{FDM}}\boldsymbol{U}(t), \tag{2.93}$$

one can see that

$$\mathcal{D} \approx -\frac{c}{h^2}\boldsymbol{L}_{\mathrm{FDM}} \tag{2.94}$$

in the present case. The temporal discretization methods discussed in Section 2.2.2 can be applied to Equation 2.93. For instance, using the explicit Euler method, we obtain:

$$\boldsymbol{U}(t+\Delta t) \approx \boldsymbol{U}(t) - \frac{c}{h^2}\boldsymbol{L}_{\mathrm{FDM}}\boldsymbol{U}(t)\Delta t. \tag{2.95}$$

where the coefficient $c\Delta t/h^2$, is the diffusion number, which must be less than $1/2$ for stable computation.

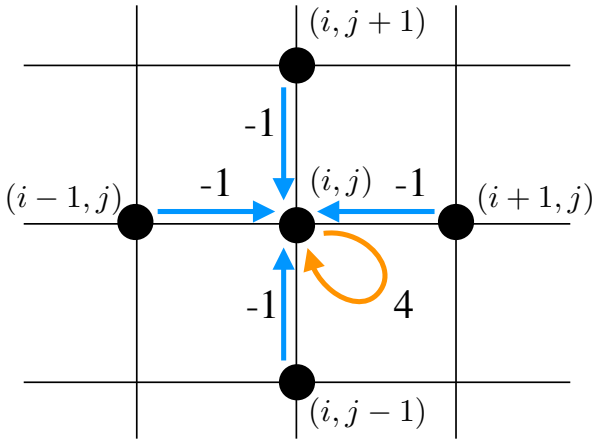


Figure 2.7: An example of 2D u field spatially discretized using FDM and its corresponding edge connectivity.

[3]The matrices may differ on the boundary, where some boundary conditions are required.

For the 2D case, we denote the vertex positions using the following indices:

$$\begin{array}{ccc} & \boldsymbol{x}_{i,j+1} = \boldsymbol{x}_i + h\boldsymbol{e}_y & \\ \boldsymbol{x}_{i-1,j} = \boldsymbol{x}_i - h\boldsymbol{e}_x & \boldsymbol{x}_{i,j} = \boldsymbol{x} & \boldsymbol{x}_{i+1,j} = \boldsymbol{x}_i + h\boldsymbol{e}_x \\ & \boldsymbol{x}_{i,j-1} = \boldsymbol{x}_i - h\boldsymbol{e}_y, & \end{array}$$

where $\boldsymbol{e}_x$ and $\boldsymbol{e}_y$ denote the unit vectors in the X and Y directions, respectively. A similar discussion leads to the following spatially discretized representation of the 2D heat equation:

$$\begin{aligned} \frac{\partial}{\partial t}u(t,\boldsymbol{x}_i) = -\frac{c}{h^2}\left[-u(t,\boldsymbol{x}_{i-1,j}) - u(t,\boldsymbol{x}_{i+1,j})\right. \\ \left. -u(t,\boldsymbol{x}_{i,j-1}) - u(t,\boldsymbol{x}_{i,j+1}) + 4u(t,\boldsymbol{x}_{i,j})\right], \end{aligned} \tag{2.96}$$

which also corresponds to the Laplacian matrix of a corresponding graph, as shown in Figure 2.7.

2.2.4.2 Finite Element Method (FEM)

The *finite element method* (FEM) utilizes a set of functions called shape functions, $N : \mathbb{R}^n \to \mathbb{R}$, for the spatial discretization of the weak form of the PDE of interest.

First, we obtain the weak form by integrating the PDE over the domain Ω And multiplying by an arbitrary test function v, as follows:

$$\int_\Omega v(\boldsymbol{x})\frac{\partial}{\partial t}u(t,\boldsymbol{x})d\Omega(\boldsymbol{x}) = \int_\Omega v(\boldsymbol{x})c\nabla\cdot\nabla u(t,\boldsymbol{x})d\Omega(\boldsymbol{x}). \tag{2.97}$$

Using

$$\begin{aligned} &\int_\Omega \nabla\cdot(v(\boldsymbol{x})\nabla u(t,\boldsymbol{x}))d\Omega(\boldsymbol{x}) \\ &\quad = \int_\Omega (\nabla v(\boldsymbol{x}))\cdot(\nabla u(t,\boldsymbol{x}))d\Omega(\boldsymbol{x}) + \int_\Omega v(\boldsymbol{x})\nabla\cdot\nabla u(t,\boldsymbol{x})d\Omega(\boldsymbol{x}), \end{aligned} \tag{2.98}$$

we obtain:

$$c\int_{\Omega} v(\boldsymbol{x})\nabla\cdot\nabla u(t,\boldsymbol{x})d\Omega(\boldsymbol{x}) \\ = c\int_{\Omega}\nabla\cdot(v(\boldsymbol{x})\nabla u(t,\boldsymbol{x}))d\Omega(\boldsymbol{x}) - c\int_{\Omega}(\nabla v(\boldsymbol{x}))\cdot(\nabla u(t,\boldsymbol{x}))d\Omega(\boldsymbol{x}). \tag{2.99}$$

Using Stokes' theorem, the first term on the right-hand side is transformed into:

$$c\int_{\Omega}\nabla\cdot(v(\boldsymbol{x})\nabla u(t,\boldsymbol{x}))d\Omega(\boldsymbol{x}) = c\int_{\partial\Omega} v(\boldsymbol{x})(\nabla u(t,\boldsymbol{x}))\cdot\boldsymbol{n}(\boldsymbol{x})d\Gamma(\boldsymbol{x}), \tag{2.100}$$

where $\boldsymbol{n}(\boldsymbol{x})$ is the normal vector at $\boldsymbol{x}\in\partial\Omega$. Now, if we assume $(\nabla u(t,\boldsymbol{x}))\cdot\boldsymbol{n}(\boldsymbol{x}) = 0$ for all $\boldsymbol{x}\in\partial\Omega$, i.e., the adiabatic condition, Equation 2.100 is equal to zero. Therefore, the equation to solve is:

$$\int_{\Omega} v(\boldsymbol{x})\frac{\partial}{\partial t}u(t,\boldsymbol{x})d\Omega(\boldsymbol{x}) = -c\int_{\Omega}(\nabla v(\boldsymbol{x}))\cdot(\nabla u(t,\boldsymbol{x}))d\Omega(\boldsymbol{x}). \tag{2.101}$$

Next, we consider the spatial discretization using the set of shape functions $\{N_i(\boldsymbol{x})\}_{i\in\{1,\ldots,|\mathcal{V}|\}}$, which is typically required to satisfy the following properties:

$$\forall\boldsymbol{x}\in\Omega,\ \sum_{i\in\{1,\ldots,|\mathcal{V}|\}} N_i(\boldsymbol{x}) = 1 \tag{2.102}$$

$$\forall i\in\{1,\ldots,|\mathcal{V}|\},\ \mathrm{supp}(N_i)\ :\ \text{compact} \tag{2.103}$$

$$\forall i,j\in\{1,\ldots,|\mathcal{V}|\},\ N_i(\boldsymbol{x}_j) = \delta_{ij}, \tag{2.104}$$

where $\mathrm{supp}(N_i) := \overline{\{\boldsymbol{x}\in\Omega | N_i(\boldsymbol{x})\neq 0\}}$ is the closed support of N_i ($\bar{\cdot}$ denotes closure), and compactness corresponds to the notion of a bounded and closed subset of the Euclidean space. A typical example of a shape function is the Lagrange interpolating polynomial shown in Figure 2.8. Using a Lagrange interpolating polynomial of degree one, one can approximate the field of u and v as follows:

$$u(t,\boldsymbol{x}) \approx \sum_{i\in\{1,\ldots,|\mathcal{V}|\}} N_i(\boldsymbol{x})u_i(t) \tag{2.105}$$

$$v(\boldsymbol{x}) \approx \sum_{i \in \{1,\ldots,|\mathcal{V}|\}} N_i(\boldsymbol{x}) v_i, \tag{2.106}$$

where $u_i(t)$ and v_i and denotes the value of $u(t, \boldsymbol{x}_i)$ and $v(\boldsymbol{x}_i)$, respectively.

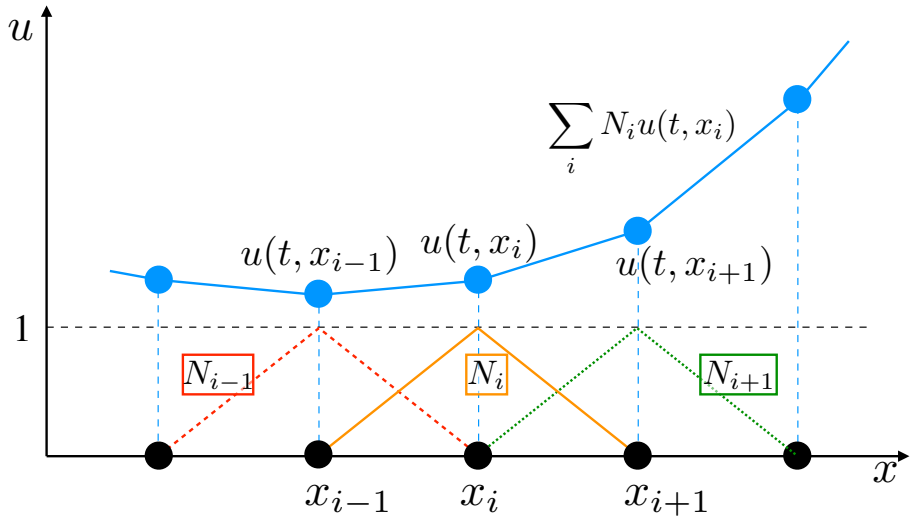


Figure 2.8: An example of a 1D u field spatially discretized using FEM.

Using the shape function, we discretize Equation 2.101 as follows:

$$\begin{aligned}
\sum_{ij} \int_\Omega N_i(\boldsymbol{x}) v_i \frac{\partial}{\partial t} N_j(\boldsymbol{x}) u_j(t) d\Omega(\boldsymbol{x}) &= -c \sum_{ij} \int_\Omega (\nabla N_i(\boldsymbol{x}) v_i) \cdot (\nabla N_j(\boldsymbol{x}) u_j(t)) d\Omega(\boldsymbol{x}) \\
\sum_{ij} \frac{\partial}{\partial t} u_j(t) \int_\Omega N_i(\boldsymbol{x}) N_j(\boldsymbol{x}) d\Omega(\boldsymbol{x}) &= -c \sum_{ij} u_j(t) \int_\Omega (\nabla N_i(\boldsymbol{x})) \cdot (\nabla N_j(\boldsymbol{x})) d\Omega(\boldsymbol{x}).
\end{aligned} \tag{2.107}$$

By letting

$$M_{ij} := \int_\Omega N_i(\boldsymbol{x}) N_j(\boldsymbol{x}) d\Omega(\boldsymbol{x}) \tag{2.108}$$

$$K_{ij} := -c \int_\Omega (\nabla N_i(\boldsymbol{x})) \cdot (\nabla N_j(\boldsymbol{x})) d\Omega(\boldsymbol{x}), \tag{2.109}$$

one can obtain:

$$\boldsymbol{M}\frac{\partial}{\partial t}\boldsymbol{U}(t) = \boldsymbol{K}\boldsymbol{U}(t). \tag{2.110}$$

In particular, for the 1D case:

$$K_{ij} = \begin{cases} -2c/h & \text{if } i = j \\ c/h & \text{if } |i-j| = 1 \\ 0 & \text{otherwise} \end{cases}, \tag{2.111}$$

thus establishing a relationship to the graph Laplacian matrix in the FEM case. Finally, we obtain:

$$\frac{\partial}{\partial t}\boldsymbol{U}(t) = \boldsymbol{M}^{-1}\boldsymbol{K}\boldsymbol{U}(t). \tag{2.112}$$

Again, we confirm that the spatial discretization introduces interactions between vertices, that is, graph-like message passing. The connectivity of the graph corresponding to the 2D case is shown in Figure 2.9. It should be noted that the connectivity of the graph is not necessarily the same as the edges of the mesh.

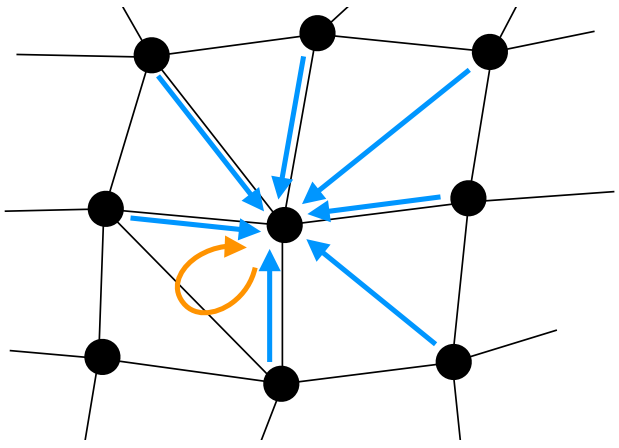


Figure 2.9: An example of 2D spatially discretized unstructured grid for FEM (black) and its corresponding edge connectivity (blue). The connectivity of the graph is not necessarily the same as the edges of the mesh.

2.2.4.3 LEAST SQUARES MOVING PARTICLE SEMI-IMPLICIT (LSMPS) METHOD

The *least qquares moving particle semi-implicit* (LSMPS) method is a mesh-free technique for solving PDEs proposed by Tamai & Koshizuka (2014). Although the scheme proposes a general method to approximate the differential up to an arbitrary order, for simplicity, we introduce only the first-order gradient model, which, using the LSMPS method, is expressed as

$$\langle \nabla u \rangle|_{\boldsymbol{x}_i} := \boldsymbol{M}_i^{-1} \sum_{j \in \mathcal{N}_i} \frac{u_j - u_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} w_{ij} \tag{2.113}$$

$$\boldsymbol{M}_i := \sum_l \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} w_{il}, \tag{2.114}$$

where $u : \Omega \to \mathbb{R}$ is a scalar field, u_i denotes $u(\boldsymbol{x}_i)$, and w_{ij} is a weight determined depending on the distance between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. Because this method does not require a mesh, $\mathcal{N}_i$ is determined using the effective radius set by the users.

The first-order model is derived using the first-order Taylor expansion as follows:

$$u_j \approx u_i + (\boldsymbol{x}_j - \boldsymbol{x}_i) \cdot (\nabla u)|_{\boldsymbol{x}_i}, \forall j \in \mathcal{N}_i. \tag{2.115}$$

Since $\nabla u|_{\boldsymbol{x}_i}$ is what we want to obtain, and we let $\nabla u|_{\boldsymbol{x}_i} = \boldsymbol{X}_i$, then we rewrite the equation as:

$$\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \cdot \boldsymbol{X}_i - \frac{u_j - u_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \approx 0, \forall j \in \mathcal{N}_i. \tag{2.116}$$

We consider a situation in which $|\mathcal{N}_i| \geq n$ (n denotes the spatial dimension), then one can obtain $\boldsymbol{X}_i$ in terms of least squares, by defining a weighted evaluation function $J(\boldsymbol{X}_i)$ as follows:

$$J(\boldsymbol{X}_i) := \frac{1}{2} \sum_{j \in \mathcal{N}_i} w_{ij} \left[\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \cdot \boldsymbol{X}_i - \frac{u_j - u_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \right]^2. \tag{2.117}$$

The best approximation to the gradient $\boldsymbol{X}_i^* \approx \nabla u|_{\boldsymbol{x}_i}$ in terms of least squares can be obtained when $\nabla_{\boldsymbol{X}_i} J(\boldsymbol{X}_i^*) = \mathbf{0}$, therefore, we solve:

$$\nabla_{\boldsymbol{X}_i} J(\boldsymbol{X}_i^*) = \sum_{j \in \mathcal{N}_i} w_{ij} \left[\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \cdot \boldsymbol{X}_i^* - \frac{u_j - u_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \right] \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} = \mathbf{0}. \tag{2.118}$$

For any vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$, the following condition holds:

$$(\boldsymbol{v} \cdot \boldsymbol{w})\boldsymbol{v} = (\boldsymbol{v} \otimes \boldsymbol{v})\boldsymbol{w}, \tag{2.119}$$

because

$$[(\boldsymbol{v} \cdot \boldsymbol{w})\boldsymbol{v}]_i = \sum_k v_k w_k v_i \tag{2.120}$$

$$= \sum_k (v_i v_k) w_k \tag{2.121}$$

$$= [(\boldsymbol{v} \otimes \boldsymbol{v})\boldsymbol{w}]_i. \tag{2.122}$$

Therefore, by substituting

$$\left[\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \cdot \boldsymbol{X}_i^* \right] \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} = \left[\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \right] \boldsymbol{X}_i^* \tag{2.123}$$

into Equation 2.118, we get:

$$\sum_{l \in \mathcal{N}_i} \left[\frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \right] \boldsymbol{X}_i^* = \sum_{j \in \mathcal{N}_i} \frac{u_j - u_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|}. \tag{2.124}$$

By solving this, we finally obtain:

$$\boldsymbol{X}_i^* = \left[\sum_{l \in \mathcal{N}_i} \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \right]^{-1} \sum_{j \in \mathcal{N}_i} \frac{u_j - u_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|}, \tag{2.125}$$

which is equivalent to Equation 2.113.

Although the Laplacian model can be derived in a different manner, one can apply the gradient operator twice, and then compute the trace to obtain a representation of the

Laplacian operator. Nevertheless, we again confirm that Equation 2.113 introduces edge connectivity through the spatial differentiation. The LSMPS model is also important because it is used as the foundation of our IsoGCN model, owing to its high generalizability, as seen in Chapter 3.

Chapter 3

IsoGCN: E(*n*)-Equivariant Graph Convolutional Network

3.1 Introduction

Graph-structured data embedded in Euclidean spaces can be utilized in many different fields such as object detection, structural chemistry analysis, and physical simulations. Graph neural networks (GNNs) have been introduced to deal with such data. The crucial properties of GNNs include permutation invariance and equivariance, as seen in Section 2.1.3.2. Besides permutations, $\mathrm{E}(n)$-invariance and equivariance must be addressed when considering graphs in Euclidean spaces because many properties of objects in the Euclidean space do not change under translation and rotation. Due to such invariance and equivariance, we can expect:

1. the interpretation of the model is facilitated;
2. the output of the model is stabilized and predictable; and
3. the training is rendered efficient by eliminating the necessity of data augmentation,

as discussed in the literature (Thomas et al., 2018; Weiler et al., 2018; Fuchs et al., 2020).

$\mathrm{E}(n)$-invariance and equivariance are inevitable, especially when applied to physical simulations, because every physical quantity and physical law is either invariant or equiv-

ariant to such a transformation. Another essential requirement for such applications is computational efficiency because the primary objective of learning a physical simulation is to replace a computationally expensive simulation method with a faster machine learning model.

In this chapter, we present *IsoGCNs*, a set of simple yet powerful models that provide computationally-efficient $\mathrm{E}(n)$- invariance and equivariance based on GCNs (Kipf & Welling, 2017). Specifically, by simply tweaking the definition of an adjacency matrix, the proposed model can realize $\mathrm{E}(n)$-invariance. Because the proposed approach relies on graphs, it can deal with the complex shapes that are usually presented using mesh or point cloud data structures. Besides, a specific form of the IsoGCN layer can be regarded as a spatial differential operator that is essential for describing physical laws. In addition, we have shown that the proposed approach is computationally efficient in terms of processing graphs with up to 1M vertices that are often presented in real physical simulations. Moreover, the proposed model exhibited faster inference compared to a conventional finite element analysis approach at the same level of accuracy. Therefore, IsoGCN models can suitably replace physical simulations regarding its power to express physical laws and faster, scalable computation. The corresponding implementation and the dataset are available online[1].

The main contributions of the present study can be summarized as follows:

- We construct $\mathrm{E}(n)$- invariant and equivariant GCNs, called IsoGCNs for the specified input and output tensor ranks.
- We demonstrate that an IsoGCN model enjoys competitive performance against state-of-the-art baseline models on the considered tasks related to physical simulations.
- We confirm that IsoGCNs can be scalable to graphs with 1M vertices and achieve inference considerably faster than conventional finite element analysis, while existing state-of-the-art baseline machine learning models cannot.

[1] `https://github.com/yellowshippo/isogcn-iclr2021`

3.2 Related Prior Work

3.2.1 GCN

Our IsoGCN models are based on GCN (Kipf & Welling, 2017), a lightweight GNN model, because GCN shows computational efficiency compared to other GNNs, where message functions are constructed using deep neural networks (Equations 2.35 and 2.37). In addition, GCN models can be $\mathrm{E}(n)$-invariant if all input features are $\mathrm{E}(n)$-equivariant because the renormalized adjacency matrix is also invariant.

However, since the message function in the GCN models is determined only by information on edge connectivities in the graphs, there have been difficulties in capturing geometrical information of meshes, e.g., the distance between vertices and angles between edges. GCN models can consider geometrical information, e.g., by feeding vertex positions to the model; however, this kind of ad-hoc solution will destroy the $\mathrm{E}(n)$-invariance, resulting in unstable prediction for geometrical data. The IsoGCN model successfully incorporates geometrical data through IsoAM (Equation 3.8), a set of adjacency matrices reflecting the geometry of meshes while retaining the computational efficiency of GCNs.

3.2.2 TFN

Another essential basis of our model is TFN (Thomas et al., 2018) (Equation 2.50). Their model incorporates SE(3)- invariance and equivariance, where SE(3) is a subgroup of E(3) without reflection. The idea of TFN is to guarantee SE(3)-equivariance using spherical harmonics, which are SE(3)-equivariant functions, and nonlinear neural networks are applied to the norm of relative positions of vertices so that equivariance is not destroyed due to nonlinearity.

The TFN model achieves high expressibility based on spherical harmonics and message passing with nonlinear neural networks. However, for this reason, considerable computational resources are required. In contrast, the present study allows a significant reduction in the computational costs because it eliminates spherical harmonics and nonlinear message passing. From this perspective, IsoGCNs are also regarded as a simplification of the TFN, as seen in equation 3.45.

3.2.3 GNN Model for Physical Simulation

Several related studies, including those by Sanchez-Gonzalez et al. (2018; 2019); Alet et al. (2019); Chang & Cheng (2020) focused on applying GNNs to learn physical simulations. These approaches allowed the physical information to be introduced to GNNs; however, addressing $\mathrm{E}(n)$-equivariance was out of the scope of their research.

In the present study, we incorporate $\mathrm{E}(n)$-invariance and equivariance into GCNs, thereby, ensuring the stability of the training and inference under $\mathrm{E}(n)$ transformation. Moreover, the proposed approach is efficient in processing large graphs with up to 1M vertices that have a sufficient number of degrees of freedom to express complex shapes.

3.3 Method

In this section, we discuss how to construct IsoGCN layers that correspond to the $\mathrm{E}(n)$-invariant and equivariant GCN layers. To formulate a model, we assume that:

1. only attributes associated with vertices and not edges; and
2. graphs do not contain self-loops.

Here, n denotes the dimension of the Euclidean space we are working on.

3.3.1 Discrete Tensor Field

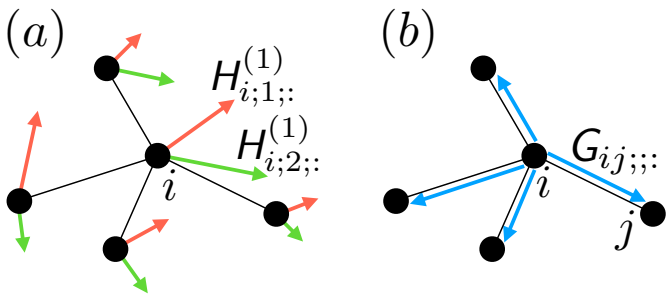


Figure 3.1: Schematic diagrams of (a) rank-1 tensor field $\boldsymbol{H}^{(1)}$ with the number of features equaling 2 and (b) the simplest case of $\boldsymbol{G}_{ij;;:} = \delta_{il}\delta_{jk}A_{ij}\boldsymbol{I}(\boldsymbol{x}_k - \boldsymbol{x}_l) = A_{ij}(\boldsymbol{x}_j - \boldsymbol{x}_i)$.

First, we introduce the concept of *discrete tensor fields*, which play an essential role to constract $\mathrm{E}(n)$-equivariant models. In the present study, we refer to tensor as geometric tensors, i.e., a rank-p tensor field $\boldsymbol{u} : \Omega \to^{(p)} \in \mathbb{R}^{n^p}$ is equivariant with regard to the orthogonal transformation using $\boldsymbol{U}$ expressed as:

$$\boldsymbol{U} : u^{(p)}_{k_1 k_2 \dots k_p} \mapsto U_{k_1 l_1} U_{k_2 l_2} \dots U_{k_p l_p} u^{(p)}_{l_1 l_2 \dots l_p}. \tag{3.1}$$

To exploit the expressive power of neural networks, we consider a collection of d_f tensors with rank-p such as:

$$\boldsymbol{h}^{(p)} := \underbrace{(\boldsymbol{u}^{(p)}, \boldsymbol{v}^{(p)}, \boldsymbol{w}^{(p)}, \dots)}_{d_\mathrm{f} \text{ items}} : \Omega \to \mathbb{R}^{d_\mathrm{f} \times n^p}. \tag{3.2}$$

This is a collection of rank-p tensor field, so note that

$$\boldsymbol{h}^{(p)}(\boldsymbol{x}) \in \mathbb{R}^{d_\mathrm{f} \times n^p} \tag{3.3}$$

and

$$\boldsymbol{U} : h^{(p)}_{g;k_1 k_2 \dots k_p} \mapsto U_{k_1 l_1} U_{k_2 l_2} \dots U_{k_p l_p} h^{(p)}_{g;l_1 l_2 \dots l_p} \tag{3.4}$$

hold.

Now, we consider a *discrete rank-p tensor field* $\boldsymbol{H}^{(p)} \in \mathbb{R}^{|\mathcal{V}| \times d_\mathrm{f} \times n^p}$, as follows:

$$\boldsymbol{H}^{(p)} := \begin{pmatrix} \boldsymbol{h}^{(p)}(\boldsymbol{x}_1) \\ \boldsymbol{h}^{(p)}(\boldsymbol{x}_2) \\ \\ \vdots \\ \\ \boldsymbol{h}^{(p)}(\boldsymbol{x}_{|\mathcal{V}|}) \end{pmatrix}, \tag{3.5}$$

where d_f denotes the number of features (channels) of $\boldsymbol{H}^{(p)}$, and $\boldsymbol{x}_i \in \Omega \subset \mathbb{R}^n$ is the position of the ith vertex. An example of the discrete tensor field is shown in Figure 3.1

(a). With the indices, we denote $H^{(p)}_{i;g;k_1k_2\ldots k_p}$, where i permutes under the permutation of vertices and $k_1, \ldots, k_p$ refers to the Euclidean representation. g is the index of features, so invariant with regard to permutation and $\mathrm{E}(n)$ transformation. Thus, under the permutation π, $\boldsymbol{H}^{(p)}$ is equivariant with regard to the vertex indices:

$$\pi : H^{(p)}_{i;g;k_1k_2\ldots k_p} \mapsto H^{(p)}_{\pi(i);g;k_1k_2\ldots k_p}, \tag{3.6}$$

and under orthogonal transformation $\boldsymbol{U}$, $\boldsymbol{H}^{(p)}$ is equivariant with regard to the dimensional indices:

$$\boldsymbol{U} : H^{(p)}_{i;g;k_1k_2\ldots k_p} \mapsto \sum_{l_1,l_2,\ldots,l_p} U_{k_1l_1}U_{k_2l_2}\ldots U_{k_pl_p}H^{(p)}_{i;g;l_1l_2\ldots l_p}. \tag{3.7}$$

We use discrete tensor fields for inputs, hidden state, and outputs of our IsoGCN models.

3.3.2 Isometric Adjacency Matrix (IsoAM)

Before constructing an IsoGCN, an *isometric adjacency matrix* (IsoAM), which is at the core of the IsoGCN concept must be defined.

3.3.2.1 Definition of IsoAM

An IsoAM $\boldsymbol{G} \in \mathbb{R}^{|\mathcal{V}|^2\times 1\times n}$ is defined as:

$$\mathbb{R}^d \ni \boldsymbol{G}_{ij;;:} := \boldsymbol{g}_{ij} := \sum_{k,l\in\mathcal{V},k\neq l} \boldsymbol{T}_{ijkl}(\boldsymbol{x}_k - \boldsymbol{x}_l), \tag{3.8}$$

where $\boldsymbol{G}_{ij;;:}$ is a slice in the spatial index of $\boldsymbol{G}$, and $\boldsymbol{T}_{ijkl} \in \mathbb{R}^{n\times n}$ is an untrainable transformation invariant and orthogonal transformation equivariant rank-2 tensor defined depending on the problem of interest. Note that we denote $G_{ij;;k}$ to be consistent with the notation of the discrete tensor field $H^{(p)}_{i;g;k_1k_2\ldots k_p}$ because i and j permutes under the vertex permutation and k represents the spatial index while the number of features is always 1. The IsoAM can be viewed as a weighted adjacency matrix for each direction and reflects spatial information while the usual weighted adjacency matrix cannot because a graph has only one adjacency matrix. Also, IsoAM can be viewed as a rank-1-tensor-valued matrix

expressed as:

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{g}_{11} & \boldsymbol{g}_{12} & \cdots & \boldsymbol{g}_{1|\mathcal{V}|} \\ \boldsymbol{g}_{21} & \boldsymbol{g}_{22} & \cdots & \boldsymbol{g}_{2|\mathcal{V}|} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{g}_{|\mathcal{V}|1} & \boldsymbol{g}_{|\mathcal{V}|2} & \cdots & \boldsymbol{g}_{|\mathcal{V}||\mathcal{V}|} \end{pmatrix}. \tag{3.9}$$

For the simplest case, one can define $\boldsymbol{T}_{ijkl} = \delta_{il}\delta_{jk}A_{ij}\boldsymbol{I}_n$ (Figure 3.1 (b)), where δ_{ij} is the Kronecker delta, $\boldsymbol{A}$ is the adjacency matrix of the graph, and $\boldsymbol{I}_n$ is the n-dimensional identity matrix that is the simplest rank-2 tensor. With the simplification, the definition of IsoAM (Equation 3.8) is expressed as:

$$\boldsymbol{g}_{ij} = A_{ij}(\boldsymbol{x}_j - \boldsymbol{x}_i). \tag{3.10}$$

In the case of the path graph with five vertices (Figure 2.2), it can be expressed as:

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{x}_2 - \boldsymbol{x}_1 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{x}_1 - \boldsymbol{x}_2 & \boldsymbol{0} & \boldsymbol{x}_3 - \boldsymbol{x}_2 & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{x}_2 - \boldsymbol{x}_3 & \boldsymbol{0} & \boldsymbol{x}_4 - \boldsymbol{x}_3 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{x}_3 - \boldsymbol{x}_4 & \boldsymbol{0} & \boldsymbol{x}_5 - \boldsymbol{x}_4 \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{x}_4 - \boldsymbol{x}_5 & \boldsymbol{0} \end{pmatrix}. \tag{3.11}$$

Therefore, one can see the IsoAMs are based on relative positions of vertices, which are translation invariant and orthogonal transformation equivariant.

In another case, $\boldsymbol{T}_{ijkl}$ can be determined from the geometry of a graph, as defined in Equation 3.47. Nevertheless, in the bulk of this section, we retain $\boldsymbol{T}_{ijkl}$ abstract to cover various forms of interaction, such as position-aware GNNs (You et al., 2019). Here, $\boldsymbol{G}$ is composed of only untrainable parameters and thus can be determined before training.

3.3.2.2 PROPERTY OF ISOAM

Here, we present the properties of the IsoAM defined by Equation 3.8. We let $\mathbb{R}^3 \ni \boldsymbol{d}(\boldsymbol{x}_l, \boldsymbol{x}_k) = (\boldsymbol{x}_k - \boldsymbol{x}_l)$ for the proofs. Note that $\boldsymbol{G}$ is expressed using $\boldsymbol{d}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ as

$$\boldsymbol{G}_{ij;;:} = \boldsymbol{g}_{ij} = \sum_{k,l \in \mathcal{V}, k \neq l} \boldsymbol{T}_{ijkl} \boldsymbol{d}(\boldsymbol{x}_l, \boldsymbol{x}_k). \tag{3.12}$$

Proposition 3.3.1. *IsoAM defined in Equation 3.8 is translation invariant and orthogonal transformation equivariant, i.e., for any* $\mathrm{E}(n)$ *transformation* $\forall \boldsymbol{t} \in \mathbb{R}^n, \boldsymbol{U} \in \mathrm{O}(n), T : \boldsymbol{x} \mapsto \boldsymbol{U}\boldsymbol{x} + \boldsymbol{t}$*,*

$$T : G_{ij;;k} \mapsto \sum_l U_{kl} G_{ij;;l}. \tag{3.13}$$

Proof. First, we demonstrate the invariance with respect to the translation with $\forall \boldsymbol{t} \in \mathbb{R}^d$. $\boldsymbol{d}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is transformed invariantly as follows under translation:

$$\begin{aligned} \boldsymbol{d}(\boldsymbol{x}_i + \boldsymbol{t}, \boldsymbol{x}_j + \boldsymbol{t}) &= [\boldsymbol{x}_j + \boldsymbol{t} - (\boldsymbol{x}_i + \boldsymbol{t})] \\ &= (\boldsymbol{x}_j - \boldsymbol{x}_i) \\ &= \boldsymbol{d}(\boldsymbol{x}_i, \boldsymbol{x}_j). \end{aligned} \tag{3.14}$$

By definition, $\boldsymbol{T}_{ijkl}$ is also translation invariant. Thus,

$$\begin{aligned} \sum_{k,l \in \mathcal{V}, k \neq l} \boldsymbol{T}_{ijkl} \boldsymbol{d}(\boldsymbol{x}_l + \boldsymbol{t}, \boldsymbol{x}_k + \boldsymbol{t}) &= \sum_{k,l \in \mathcal{V}, k \neq l} \boldsymbol{T}_{ijkl} \boldsymbol{d}(\boldsymbol{x}_l, \boldsymbol{x}_k) \\ &= \boldsymbol{G}_{ij;;:}. \end{aligned} \tag{3.15}$$

We then show an equivariance regarding the orthogonal transformation with $\forall \boldsymbol{U} \in \mathrm{O}(d)$. $\boldsymbol{d}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is transformed as follows by orthogonal transformation:

$$\begin{aligned} \boldsymbol{d}(\boldsymbol{U}\boldsymbol{x}_i, \boldsymbol{U}\boldsymbol{x}_j) &= \boldsymbol{U}\boldsymbol{x}_j - \boldsymbol{U}\boldsymbol{x}_i \\ &= \boldsymbol{U}(\boldsymbol{x}_j - \boldsymbol{x}_i) \\ &= \boldsymbol{U}\boldsymbol{d}(\boldsymbol{x}_i, \boldsymbol{x}_j). \end{aligned} \tag{3.16}$$

By definition, $\boldsymbol{T}_{ijkl}$ is transformed to $\boldsymbol{U}\boldsymbol{T}_{ijkl}\boldsymbol{U}^{-1}$ by orthogonal transformation. Thus,

$$\begin{aligned}\sum_{k,l\in\mathcal{V},k\neq l}\boldsymbol{U}\boldsymbol{T}_{ijkl}\boldsymbol{U}^{-1}\boldsymbol{d}(\boldsymbol{U}\boldsymbol{x}_l,\boldsymbol{U}\boldsymbol{x}_k)&=\sum_{k,l\in\mathcal{V},k\neq l}\boldsymbol{U}\boldsymbol{T}_{ijkl}\boldsymbol{U}^{-1}\boldsymbol{U}\boldsymbol{d}(\boldsymbol{x}_l,\boldsymbol{x}_k)\\&=\boldsymbol{U}\boldsymbol{G}_{ij;;:}.\end{aligned}\tag{3.17}$$

Therefore, $\boldsymbol{G}$ is translation invariant and an orthogonal transformation equivariant. □

Here, we define essential operations between IsoAMs and discrete tensor fields. Based on the definition of the GCN layer in the equation 2.35, let $\boldsymbol{G}*\boldsymbol{H}^{(0)}\in\mathbb{R}^{|\mathcal{V}|\times f\times d}$ denote the *convolution* between $\boldsymbol{G}$ and the rank-0 tensor field $\boldsymbol{H}^{(0)}\in\mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{f}}}$ as follows:

$$(\boldsymbol{G}*\boldsymbol{H}^{(0)})_{i;g;k}:=\sum_j\boldsymbol{G}_{ij;;k}H^{(0)}_{j;g;}.\tag{3.18}$$

With a rank-1 tensor field $\boldsymbol{H}^{(1)}\in\mathbb{R}^{|\mathcal{V}|\times f\times d}$, let $\boldsymbol{G}\odot\boldsymbol{H}^{(1)}\in\mathbb{R}^{|\mathcal{V}|\times f}$ and $\boldsymbol{G}\odot\boldsymbol{G}\in\mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ denote the *contractions* which are defined as follows:

$$(\boldsymbol{G}\odot\boldsymbol{H}^{(1)})_{i;g;}:=\sum_{j,k}G_{ij;;k}H^{(1)}_{j;g;k}\tag{3.19}$$

$$(\boldsymbol{G}\odot\boldsymbol{G})_{il;;}:=\sum_{j,k}G_{ij;;k}G_{jl;k}.\tag{3.20}$$

The contraction of IsoAMs $\boldsymbol{G}\odot\boldsymbol{G}$ can be interpreted as the inner product of each component in the IsoAMs. Thus, the subsequent proposition follows.

Proposition 3.3.2. *The contraction of IsoAMs $\boldsymbol{G}\odot\boldsymbol{G}$ is $\mathrm{E}(n)$-invariant, i.e., for any $\mathrm{E}(n)$ transformation $\forall\boldsymbol{t}\in\mathbb{R}^3,\boldsymbol{U}\in\mathrm{O}(d),T:\boldsymbol{x}\mapsto\boldsymbol{U}\boldsymbol{x}+\boldsymbol{t}$, $\boldsymbol{G}\odot\boldsymbol{G}\mapsto\boldsymbol{G}\odot\boldsymbol{G}$.*

Proof. Here, $\boldsymbol{G}\odot\boldsymbol{G}$ is translation invariant because $\boldsymbol{G}$ is translation invariant. We prove rotation invariance under an orthogonal transformation $\forall\boldsymbol{U}\in\mathrm{O}(n)$. In addition, $\boldsymbol{G}\odot\boldsymbol{G}$ is

transformed under $\boldsymbol{U}$ as follows:

$$\begin{aligned}
\sum_{j,k} G_{ij;;k} G_{jl;;k} &\mapsto \sum_{j,k,m,n} U_{km} G_{ij;;m} U_{kn} G_{jl;;n} \\
&= \sum_{j,k,m,n} U_{km} U_{kn} G_{ij;;m} G_{jl;;n} \\
&= \sum_{j,k,m,n} U^T_{mk} U_{kn} G_{ij;;m} G_{jl;;n} \\
&= \sum_{j,m,n} \delta_{mn} G_{ij;;m} G_{jl;;n} && (\because \text{property of the orthogonal matrix}) \\
&= \sum_{j} G_{ij;;m} G_{jl;;m} \\
&= \sum_{j,k} G_{ij;;k} G_{jl;;k}. && (\because \text{Change the dummy index } m \to k)
\end{aligned} \tag{3.21}$$

Therefore, $\boldsymbol{G} \odot \boldsymbol{G}$ is $\mathrm{E}(n)$-invariant. □

With a rank-p tensor field $\boldsymbol{H}^{(p)} \in \mathbb{R}^{|\mathcal{V}| \times f \times d^p}$, let $\boldsymbol{G} \otimes \boldsymbol{H}^{(p)} \in \mathbb{R}^{|\mathcal{V}| \times f \times d^{1+p}}$. and $\boldsymbol{G} \otimes \boldsymbol{G} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}| \times d^2}$ denote the **tensor products** defined as follows:

$$(\boldsymbol{G} \otimes \boldsymbol{H}^{(p)})_{i;g;km_1m_2\ldots m_p} := \sum_j \boldsymbol{G}_{ij;;k} H^{(p)}_{j;g;m_1m_2\ldots m_p}, \tag{3.22}$$

$$(\boldsymbol{G} \otimes \boldsymbol{G})_{il;;k_1k_2} := \sum_j \boldsymbol{G}_{ij;;k_1} \boldsymbol{G}_{jl;;k_2}. \tag{3.23}$$

The tensor product of IsoAMs $\boldsymbol{G} \otimes \boldsymbol{G}$ can be interpreted as the tensor product of each of the IsoAMs components. Thus, the subsequent proposition follows:

Proposition 3.3.3. *The tensor product of the IsoAMs $\boldsymbol{G} \otimes \boldsymbol{G}$ is $\mathrm{E}(n)$-equivariant in terms of the rank-2 tensor, i.e., for any $\mathrm{E}(n)$ transformation $\forall \boldsymbol{t} \in \mathbb{R}^3, \boldsymbol{U} \in \mathrm{O}(d), T : \boldsymbol{x} \mapsto \boldsymbol{U}\boldsymbol{x} + \boldsymbol{t}$, and $\forall i, j \in 1, \ldots, |\mathcal{V}|$, $(\boldsymbol{G} \otimes \boldsymbol{G})_{ij;;k_1k_2} \mapsto \boldsymbol{U}_{k_1l_1} \boldsymbol{U}_{k_2l_2} (\boldsymbol{G} \otimes \boldsymbol{G})_{ij;;l_1l_2}$.*

Proof. $\boldsymbol{G} \otimes \boldsymbol{G}$ is transformed under $\forall \boldsymbol{U} \in \mathrm{O}(n)$ as follows:

$$\begin{aligned}\sum_j G_{ij;;k} G_{jl;;m} &\mapsto \sum_{n,o} U_{kn} G_{ij;;n} U_{mo} G_{jl;;o} \\ &= \sum_{n,o} U_{kn} G_{ij;;n} G_{jl;;o} U^T_{om}. \end{aligned} \tag{3.24}$$

By regarding $G_{ij;;n} G_{jl;;o}$ as one matrix H_{no}, it follows the coordinate transformation of rank-2 tensor $\boldsymbol{U}\boldsymbol{H}\boldsymbol{U}^T$ for each i, j, and l. □

This proposition is easily generalized to the tensors of higher ranks by defining the pth tensor power of $\boldsymbol{G}$ as follows:

$$\bigotimes^0 \boldsymbol{G} = 1 \tag{3.25}$$

$$\bigotimes^1 \boldsymbol{G} = \boldsymbol{G} \tag{3.26}$$

$$\bigotimes^p \boldsymbol{G} = \bigotimes^{p-1} \boldsymbol{G} \otimes \boldsymbol{G} \qquad (p > 1). \tag{3.27}$$

Namely, $\bigotimes^p \boldsymbol{G}$ is $\mathrm{E}(n)$-equivariant in terms of rank-p tensor. Also, one can compute the tensor product between the rank-p IsoAM and rank-q discrete tensor field as follows:

$$\begin{aligned}\left(\bigotimes^p \boldsymbol{G}\right) \otimes \boldsymbol{H}^{(q)} &= \left(\bigotimes^{p-1} \boldsymbol{G}\right) \otimes \underbrace{(\boldsymbol{G} \otimes \boldsymbol{H}^{(q)})}_{\text{Let } \boldsymbol{H}^{(q+1)}} \\ &= \left(\bigotimes^{p-2} \boldsymbol{G}\right) \otimes \underbrace{(\boldsymbol{G} \otimes \boldsymbol{H}^{(q+1)})}_{\text{Let } \boldsymbol{H}^{(q+2)}} \\ &= \dots \\ &= \boldsymbol{H}^{(q+p)} \end{aligned} \tag{3.28}$$

Similarly, the convolution can be generalized for $\bigotimes^p \boldsymbol{G}$ and the rank-0 tensor field $\boldsymbol{H}^{(0)} \in \mathbb{R}^{|\mathcal{V}| \times f}$ as follows:

$$\left[\left(\bigotimes^p \boldsymbol{G}\right) * \boldsymbol{H}^{(0)}\right]_{i;g;k_1 k_2 \dots k_p} = \sum_j \left(\bigotimes^p \boldsymbol{G}\right)_{ij;;k_1 k_2 \dots k_p} H^{(0)}_{j;g;}. \tag{3.29}$$

The contraction can be generalized for $\bigotimes^p \boldsymbol{G}$ and the rank-q tensor field $\boldsymbol{H}^{(q)} \in \mathbb{R}^{|\mathcal{V}| \times f \times d^q}$ ($p \geq q$) as specified below:

$$\left[\left(\bigotimes^p \boldsymbol{G}\right) \odot \boldsymbol{H}^{(q)}\right]_{i;g;k_1k_2\ldots k_{p-q}} = \sum_{j,m_1,m_2,\ldots,m_q} \left(\bigotimes^p \boldsymbol{G}\right)_{ij;;k_1k_2\ldots k_{p-q}m_1m_2\ldots m_q} H^{(q)}_{j;g;m_1m_2\ldots m_q}. \tag{3.30}$$

For the case $p < q$, the contraction can be defined similarly.

By construction, one can see the the IsoAM is permutation equivariant as:

$$\pi : \boldsymbol{G} \mapsto \boldsymbol{P}\boldsymbol{G}\boldsymbol{P}^\top, \tag{3.31}$$

where $\boldsymbol{P}$ is the corresponding permutation matrix, as discussed in Maron et al. (2018). This property is the same as that of ordinary adjacency matrices. The contraction and tensor product of IsoAMs are also permutation equivariant because:

$$\pi : \boldsymbol{G} \odot \boldsymbol{G} \mapsto \boldsymbol{P}\boldsymbol{G}\boldsymbol{P}^\top \odot \boldsymbol{P}\boldsymbol{G}\boldsymbol{P}^\top \tag{3.32}$$

$$= \boldsymbol{P}\boldsymbol{G} \odot \boldsymbol{G}\boldsymbol{P}^\top \tag{3.33}$$

$$\pi : \boldsymbol{G} \otimes \boldsymbol{G} \mapsto \boldsymbol{P}\boldsymbol{G}\boldsymbol{P}^\top \otimes \boldsymbol{P}\boldsymbol{G}\boldsymbol{P}^\top \tag{3.34}$$

$$= \boldsymbol{P}\boldsymbol{G} \otimes \boldsymbol{G}\boldsymbol{P}^\top. \tag{3.35}$$

This discussion is also easily generalized for the higher order tensor cases.

Finally, we can conclude that convolution, contraction, and tensor product between rank-p IsoAM and discrete tensor field are permutation and $\mathrm{E}(n)$-equivariant because each component has such equivariance. Therefore, these operations are essential to construct $\mathrm{E}(n)$-equivariant GCN layers, IsoGCNs.

3.3.3 Construction of IsoGCN

Using the operations defined above, we can construct IsoGCN layers, which take the discrete tensor field of any rank as input, and output the tensor field of any rank, which can differ from those of the input.

3.3.3.1 $\mathrm{E}(n)$-INVARIANT LAYER

As can be seen in Proposition 3.3.1, the contraction of IsoAMs is $\mathrm{E}(n)$-invariant. Therefore, an $\mathrm{E}(n)$-invariant layer with a rank-0 input discrete tensor field and rank-0 output discrete tensor field, $\mathrm{IsoGCN}_{0\to 0} : \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{in}}} \ni \boldsymbol{H}_{\mathrm{in}}^{(0)} \mapsto \boldsymbol{H}_{\mathrm{out}}^{(0)} \in \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{out}}}$, can be constructed as

$$\boldsymbol{H}_{\mathrm{out}}^{(0)} = \mathrm{IsoGCN}_{0\to 0}(\boldsymbol{H}_{\mathrm{in}}^{(0)}) = \mathrm{PointwiseMLP}\left((\boldsymbol{G}\odot\boldsymbol{G})\,\boldsymbol{H}_{\mathrm{in}}^{(0)}\right), \tag{3.36}$$

where $\mathrm{PointwiseMLP} : \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{in}}} \to \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{out}}}$ is the pointwise MLP defined in Equation 2.26. By defining $\boldsymbol{L} := \boldsymbol{G}\odot\boldsymbol{G} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, it can be simplified as

$$\boldsymbol{H}_{\mathrm{out}}^{(0)} = \mathrm{PointwiseMLP}\left(\boldsymbol{L}\boldsymbol{H}_{\mathrm{in}}^{(0)}\right), \tag{3.37}$$

which has the same form as a GCN (equation 2.35), with the exception that $\hat{\boldsymbol{A}}$ is replaced with $\boldsymbol{L}$. It is noteworthy that $\boldsymbol{L}$ incorporates geometry information, which was missing in the GCN formulation, even though the equations are similar. Therefore, we see that IsoGCN successfully leverages geometry information in addition to graph topology.

An $\mathrm{E}(n)$-invariant layer with the rank-p input tensor field and rank-0 output tensor field, $\mathrm{IsoGCN}_{p\to 0} : \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{in}}\times n^p} \ni \boldsymbol{H}_{\mathrm{in}}^{(p)} \mapsto \boldsymbol{H}_{\mathrm{out}}^{(0)} \in \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{out}}}$, can be formulated as

$$\boldsymbol{H}_{\mathrm{out}}^{(0)} = \mathrm{IsoGCN}_{p\to 0}(\boldsymbol{H}_{\mathrm{in}}^{(p)}) = \mathrm{PointwiseMLP}\left(\left[\bigotimes^{p}\boldsymbol{G}\right]\odot\boldsymbol{H}_{\mathrm{in}}^{(p)}\right). \tag{3.38}$$

If $p = 1$, such approaches utilize the inner products of the vectors in $\mathbb{R}^d$, these operations correspond to the extractions of a relative distance and an angle of each pair of vertices, which are employed in Klicpera et al. (2020).

3.3.3.2 $\mathrm{E}(n)$-EQUIVARIANT LAYER

To construct an $\mathrm{E}(n)$-equivariant layer, one can use linear transformation, convolution and tensor product to the input tensors. If both the input and the output tensor ranks are greater than 0, one can apply neither nonlinear activation nor bias addition because these operations will cause an inappropriate distortion of the isometry because $\mathrm{E}(n)$ transforma-

tion does not commute with them in general. However, a conversion that uses only a linear transformation, convolution, and tensor product does not have nonlinearity, which limits the predictive performance of the model. To add nonlinearity to such a conversion, we can first convert the input tensors to rank-0 ones, apply nonlinear activations, and then multiply them to the higher rank tensors, as done in TFN model (Equation 2.50).

To achieve nonlinearity, first we define the $\mathrm{E}(n)$-*equivariant pointwise MLP* layer, $\mathrm{EquivariantPointwiseMLP} : \mathbb{R}^{|\mathcal{V}| \times d_{\mathrm{in}} \times n^p} \to \mathbb{R}^{|\mathcal{V}| \times d_{\mathrm{out}} \times n^p}$, as follows:

$$\mathrm{EquivariantPointwiseMLP}(\boldsymbol{H}_{\mathrm{in}}^{(p)}) := \mathrm{PointwiseMLP}\left(\left\|\boldsymbol{H}_{\mathrm{in}}^{(p)}\right\|^2\right) \circledast_{\mathrm{feat}} \boldsymbol{H}_{\mathrm{in}}^{(p)} \circledast_{\mathrm{feat}} \boldsymbol{W}, \tag{3.39}$$

where $\circledast_{\mathrm{feat}}$ is the multiplication in the feature direction and $\boldsymbol{W} \in \mathbb{R}^{d_{\mathrm{in}} \times d_{\mathrm{out}}}$ is a trainable weight matrix. The pointwise MLP, $\mathrm{PointwiseMLP} : \mathbb{R}^{|\mathcal{V}| \times d_{\mathrm{in}} \times n^p} \to \mathbb{R}^{|\mathcal{V}| \times d_{\mathrm{out}} \times n^p}$, is chosen to have the consistent output dimension. Using the index notation, Equation 3.39 turns into:

$$\begin{aligned}\left[\mathrm{EquivariantPointwiseMLP}(\boldsymbol{H}_{\mathrm{in}}^{(p)})\right]_{i;g;k_1k_2\ldots k_p} \\ = \sum_h \left[\mathrm{PointwiseMLP}\left(\left\|\boldsymbol{H}_{\mathrm{in}}^{(p)}\right\|^2\right)\right]_{i;h;} \left[\boldsymbol{H}_{\mathrm{in}}^{(p)}\right]_{i;h;k_1k_2\ldots k_p} W_{hg}.\end{aligned} \tag{3.40}$$

One can easily see that $\mathrm{EquivariantPointwiseMLP}$ defined in Equation 3.39 is translation invariant and orthogonal transformation equivariant because

$$\left\|\boldsymbol{H}_{\mathrm{in}}^{(p)}\right\|_{i;g;}^2 = \sum_{k_1k_2\ldots k_p} \left[\boldsymbol{H}_{\mathrm{in}}^{(p)}\right]_{i;g;k_1k_2\ldots k_p} \left[\boldsymbol{H}_{\mathrm{in}}^{(p)}\right]_{i;g;k_1k_2\ldots k_p} \tag{3.41}$$

is $\mathrm{E}(n)$-invariant. We use $\|\cdot\|^2$ instead of $\|\cdot\|$ in the function because computation of $\|\cdot\|$ requires computation of the square root, which leads extreme gradient around zero. One can regard $\mathrm{EquivariantPointwiseMLP}$ as an equivariant function that does not change the input tensor rank and may change the number of features.

The nonlinear $\mathrm{E}(n)$-equivariant layer with the rank-p input discrete tensor field and the rank-q ($p \leq q$) output discrete tensor field, $\mathrm{IsoGCN}_{p\to q} : \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{in}}\times n^p} \ni \boldsymbol{H}_{\mathrm{in}}^{(p)} \mapsto \boldsymbol{H}_{\mathrm{out}}^{(q)} \in \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{out}}\times n^q}$, can be defined as:

$$\boldsymbol{H}_{\mathrm{out}}^{(q)} = \mathrm{IsoGCN}_{p\to q}(\boldsymbol{H}_{\mathrm{in}}^{(p)}) := \mathrm{EquivariantPointwiseMLP}\left(\left[\bigotimes^{q-p}\boldsymbol{G}\right] \otimes \boldsymbol{H}_{\mathrm{in}}^{(p)}\right). \tag{3.42}$$

If $p = 0$, we regard $\boldsymbol{G} \otimes \boldsymbol{H}^{(0)}$ as $\boldsymbol{G} * \boldsymbol{H}^{(0)}$. If $p = q$, one can add the residual connection (He et al., 2016) in Equation 3.42. If $p > q$,

$$\boldsymbol{H}_{\mathrm{out}}^{(q)} = \mathrm{IsoGCN}_{p\to q}(\boldsymbol{H}_{\mathrm{in}}^{(p)}) := \mathrm{EquivariantPointwiseMLP}\left(\left[\bigotimes^{p-q}\boldsymbol{G}\right] \odot \boldsymbol{H}_{\mathrm{in}}^{(p)}\right). \tag{3.43}$$

In general, the nonlinear $\mathrm{E}(n)$-equivariant IsoGCN layer with the rank-$P_{\min}$ to rank-$P_{\max}$ input tensor field $\left\{\boldsymbol{H}_{\mathrm{in}}^{(p)}\right\}_{p=P_{\min}}^{P_{\max}}$ and the rank-q output tensor field $\boldsymbol{H}_{\mathrm{out}}^{(q)}$ can be defined as:

$$\begin{aligned}\boldsymbol{H}_{\mathrm{out}}^{(q)} =& \mathrm{IsoGCN}_{\cdot\to q}\left(\left\{\boldsymbol{H}_{\mathrm{in}}^{(p)}\right\}_{p=P_{\min}}^{P_{\max}}\right) && (3.44)\\ :=& \mathrm{EquivariantPointwiseMLP}(\boldsymbol{H}_{\mathrm{in}}^{(q)}) + \boldsymbol{F}_{\mathrm{gather}}\left(\left\{\mathrm{IsoGCN}_{p\to q}(\boldsymbol{H}_{\mathrm{in}}^{(p)})\right\}_{p=P_{\min}}^{P_{\max}}\right), && (3.45)\end{aligned}$$

where $\boldsymbol{F}_{\mathrm{gather}}$ denotes a function such as summation, product and concatenation in the feature direction. One can see that this layer is similar to that in the TFN (Equation 2.50), while there are no spherical harmonics and trainable message passing in the IsoGCN model.

To be exact, the output of the layer defined above is translation invariant. To output translation equivariant variables such as the vertex positions after deformation (which change accordingly with the translation of the input graph), one can first define the reference vertex position $\boldsymbol{x}_{\mathrm{ref}}$ for each graph, then compute the translation invariant output using equation 3.45, and finally, add $\boldsymbol{x}_{\mathrm{ref}}$ to the output. For more detailed information on IsoGCN modeling, see Section 3.3.5.

3.3.4 IsoAM Refined for Numerical Analysis

The IsoAM $\boldsymbol{G}$ is defined in a general form for the propositions to work with various classes of graph. In this section, we concretize the concept of the IsoAM to apply an IsoGCN to mesh-structured numerical analysis data. Here, a mesh is regarded as a graph regarding the points in the mesh as vertices of the graph and assuming two vertices are connected when they share the same element (cell), as seen in Figure 2.9.

3.3.4.1 Definition of Differential IsoAM

As seesn in Section 2.2.4, the graph connectivities are closely related to spatial differentiation. Therefore, it is natural to construct a graph reflecting the structure of spatial differentiation. Here, we define the *differential IsoAM*, a concrete instance of IsoAMs refined for numerical analysis.

The differential IsoAM $\tilde{\boldsymbol{G}}, \hat{\boldsymbol{G}} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|\times d}$ is defined as follows:

$$\tilde{G}_{ij;;k} = \hat{G}_{ij;;k} - \delta_{ij} \sum_l \hat{G}_{il;;k} \tag{3.46}$$

$$\hat{G}_{ij;;:} = \boldsymbol{M}_i^{-1} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|^2} w_{ij} A_{ij}(m) \tag{3.47}$$

$$\boldsymbol{M}_i = \sum_l \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} w_{il} A_{il}(m), \tag{3.48}$$

where $\mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|} \ni \boldsymbol{A}(m) := \min\left(\sum_{k=1}^m \boldsymbol{A}^k, 1\right)$ is an adjacency matrix up to m hops and $w_{ij} \in \mathbb{R}$ is an untrainable weight between the ith and jth vertices that is determined depending on the tasks[2]. Although one could define w_{ij} as a function of the distance $\|\boldsymbol{x}_j - \boldsymbol{x}_i\|$, w_{ij} was kept constant with respect to the distance required to maintain the simplicity of the model with fewer hyperparameters.

By regarding

$$\boldsymbol{T}_{ijkl} = \delta_{il}\delta_{jk} \boldsymbol{M}_i^{-1} w_{ij} \boldsymbol{A}_{ij}(m) / \|\boldsymbol{x}_j - \boldsymbol{x}_i\|^2 \tag{3.49}$$

[2] $\boldsymbol{M}_i$ is invertible when the number of independent vectors in $\{\boldsymbol{x}_l - \boldsymbol{x}_i\}_l$ is greater than or equal to the space dimension n, which is true for common meshes, e.g., a solid mesh in 3D Euclidean space.

in equation 3.8, one can see that $\hat{\boldsymbol{G}}$ is qualified as an IsoAM. Because a linear combination of IsoAMs is also an IsoAM, $\tilde{\boldsymbol{G}}$ is an IsoAM. Thus, they provide translation invariance and orthogonal transformation equivariance. $\tilde{\boldsymbol{G}}$ can be obtained only from the mesh geometry information, thus can be computed in the preprocessing step.

Here, $\tilde{\boldsymbol{G}}$ is designed such that it corresponds to the gradient operator model used in the LSMPS method(Tamai & Koshizuka, 2014) (Equation 2.113, while we added $A_{ij}(m)$ factor to work on graphs. As presented in Table 3.1, $\tilde{\boldsymbol{G}}$ is closely related to many differential operators, such as the gradient, divergence, Laplacian, Jacobian, and Hessian. Therefore, the considered IsoAM plays an essential role in constructing neural network models that are capable of learning differential equations. In the following sections, we discuss the connection between the IsoAM for numerical analysis $\tilde{\boldsymbol{G}}$ and the differential operators such as the gradient, divergence, the Laplacian, the Jacobian, and the Hessian operators.

Table 3.1: Correspondence between the differential operators and the expressions using the IsoAM $\tilde{\boldsymbol{G}}$.

Differential operator	**Expression**
Gradient	$\tilde{\boldsymbol{G}} * \boldsymbol{H}^{(0)}$
Divergence	$\tilde{\boldsymbol{G}} \odot \boldsymbol{H}^{(1)}$
Laplacian	$\tilde{\boldsymbol{G}} \odot \tilde{\boldsymbol{G}}\boldsymbol{H}^{(0)}$
Jacobian	$\tilde{\boldsymbol{G}} \otimes \boldsymbol{H}^{(1)}$
Hessian	$\tilde{\boldsymbol{G}} \otimes \tilde{\boldsymbol{G}} * \boldsymbol{H}^{(0)}$

3.3.4.2 Partial Derivative

First let us consider a partial derivative model of a rank-0 discrete tensor field $\boldsymbol{H}^{(0)}$ at the ith vertex and gth feature regarding the kth axis $\left\langle \partial \boldsymbol{H}^{(0)} / \partial x_k \right\rangle_{i;g;} \in \mathbb{R}$ $(k \in \{1, \ldots, n\})$. Recalling the gradient model of the LSMPS method (Equation 2.113),

$$\begin{aligned}\left\langle \frac{\partial \boldsymbol{H}^{(0)}}{\partial x_k} \right\rangle_{i;g;} :=& \boldsymbol{M}_i^{-1} \sum_j \frac{H^{(0)}_{j;g;} - H^{(0)}_{i;g;}}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \frac{x_{jk} - x_{ik}}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} w_{ij} A_{ij}(m) \\ =& \sum_j \hat{G}_{ijk} (H^{(0)}_{j;g;} - H^{(0)}_{i;g;}). \end{aligned} \tag{3.50}$$

3.3.4.3 GRADIENT

$\tilde{\boldsymbol{G}}$ is similar to a graph Laplacian matrix based on $\hat{\boldsymbol{G}}$; however, surprizingly, $\tilde{\boldsymbol{G}} * \boldsymbol{H}^{(0)}$ can be interpreted as the gradient within the Euclidean space. Let $\nabla \boldsymbol{H}^{(0)} \in \mathbb{R}^{|\mathcal{V}| \times f \times d}$ be an approximation of the gradient of $\boldsymbol{H}^{(0)}$. Using Equation 3.50, the gradient model can be expressed as follows:

$$\left\langle \nabla \boldsymbol{H}^{(0)} \right\rangle_{i;g;k} = \frac{\partial H^{(0)}_{i;g;}}{\partial x_k} \tag{3.51}$$

$$= \hat{G}_{ijk}(H^{(0)}_{j;g;} - H^{(0)}_{i;g;}). \tag{3.52}$$

Using this gradient model, we can confirm that $(\tilde{\boldsymbol{G}} * \boldsymbol{H}^{(0)})_{i;g;k} = (\nabla \boldsymbol{H}^{(0)})_{i;glk}$ because

$$\left\langle \tilde{\boldsymbol{G}} * \boldsymbol{H}^{(0)} \right\rangle_{i;g;k} = \sum_j \tilde{G}_{ij;;k} H^{(0)}_{j;g;} \tag{3.53}$$

$$= \sum_j (\hat{G}_{ij;;k} - \delta_{ij} \sum_l \hat{G}_{il;;k}) H^{(0)}_{j;g;}$$

$$= \sum_j \hat{G}_{ij;;k} H^{(0)}_{j;g;} - \sum_{j,l} \delta_{ij} \hat{G}_{il;;k} H^{(0)}_{j;g;}$$

$$= \sum_j \hat{G}_{ij;;k} H^{(0)}_{j;g;} - \sum_l \hat{G}_{il;;k} H^{(0)}_{i;g;}$$

$$= \sum_j \hat{G}_{ij;;k} H^{(0)}_{j;g;} - \sum_j \hat{G}_{ij;;k} H^{(0)}_{i;g;} \qquad (\because \text{Change the dummy index } l \to j)$$

$$= \sum_j \hat{G}_{ij;;k} (H^{(0)}_{j;g;} - H^{(0)}_{i;g;})$$

$$= \left\langle \nabla \boldsymbol{H}^{(0)} \right\rangle_{i;g;k}. \tag{3.54}$$

Therefore, $\tilde{\boldsymbol{G}}*$ can be interpreted as the gradient operator within a Euclidean space.

3.3.4.4 DIVERGENCE

We show that $\tilde{\boldsymbol{G}} \odot \boldsymbol{H}^{(1)}$ corresponds to the divergence. Using Equation 3.50, the divergence model $\left\langle \nabla \cdot \boldsymbol{H}^{(1)} \right\rangle \in \mathbb{R}^{|\mathcal{V}| \times f}$ is expressed as follows:

$$\left\langle \nabla \cdot \boldsymbol{H}^{(1)} \right\rangle_{i;g;} = \left\langle \sum_k \frac{\partial \boldsymbol{H}^{(1)}}{\partial x_k} \right\rangle_{i;g;} \tag{3.55}$$

$$= \sum_{j,k} \hat{G}_{ij;;k}(H^{(1)}_{j;g;k} - H^{(1)}_{i;g;k}). \tag{3.56}$$

Then, $\tilde{\boldsymbol{G}} \odot \boldsymbol{H}^{(1)}$ is

$$\begin{aligned}
\left\langle \tilde{\boldsymbol{G}} \odot \boldsymbol{H}^{(1)} \right\rangle_{i;g;} &= \sum_{j,k} \tilde{G}_{ij;;k} H^{(1)}_{j;g;k} \\
&= \sum_{j,k} \left(\hat{G}_{ij;;k} - \delta_{ij} \sum_{l} \hat{G}_{il;;k} \right) H^{(1)}_{j;g;k} \\
&= \sum_{j,k} \hat{G}_{ij;;k} H^{(1)}_{j;g;k} - \sum_{l,k} \hat{G}_{il;;k} H^{(1)}_{i;g;k} \\
&= \sum_{j,k} \hat{G}_{ij;;k} (H^{(1)}_{j;g;k} - H^{(1)}_{i;g;k}) \qquad (\because \text{Change the dummy index } l \to j) \\
&= \left\langle \nabla \cdot \boldsymbol{H}^{(1)} \right\rangle_{i;g;}.
\end{aligned} \tag{3.57}$$

3.3.4.5 Laplacian Operator

We prove that $\tilde{\boldsymbol{G}} \odot \tilde{\boldsymbol{G}}$ corresponds to the Laplacian operator within a Euclidean space.

Using Equation 3.50, the Laplacian model $\left\langle \nabla \cdot \nabla \, \boldsymbol{H}^{(0)} \right\rangle \in \mathbb{R}^{|\mathcal{V}| \times f}$ can be expressed as follows:

$$\begin{aligned}
\left\langle \nabla \cdot \nabla \, \boldsymbol{H}^{(0)} \right\rangle_{i;g;} &:= \sum_{k} \left\langle \frac{\partial}{\partial x_k} \left\langle \frac{\partial \boldsymbol{H}}{\partial x_k} \right\rangle_{i} \right\rangle_{i;g;} \\
&= \sum_{j,k} \hat{G}_{ij;;k} \left\langle \left\langle \frac{\partial \boldsymbol{H}}{\partial x_k} \right\rangle_{j;g;} - \left\langle \frac{\partial \boldsymbol{H}}{\partial x_k} \right\rangle_{i;g;} \right\rangle \\
&= \sum_{j,k} \hat{G}_{ij;;k} \left[\sum_{l} \hat{G}_{jl;;k} (H^{(0)}_{l;g;} - H^{(0)}_{j;g;}) - \sum_{l} \hat{G}_{il;;k} (H^{(0)}_{l;g;} - H^{(0)}_{i;g;}) \right] \\
&= \sum_{j,k,l} \hat{G}_{ij;;k} (\hat{G}_{jl;;k} - \hat{G}_{il;;k}) (H^{(0)}_{l;g;} - H^{(0)}_{j;g;}).
\end{aligned} \tag{3.58}$$

Then, $(\tilde{\boldsymbol{G}} \odot \tilde{\boldsymbol{G}}) \boldsymbol{H}^{(0)}$ is

$$((\tilde{\boldsymbol{G}} \odot \tilde{\boldsymbol{G}}) \boldsymbol{H}^{(0)})_{i;g;} = \sum_{j,k,l} \tilde{G}_{ij;;k} \tilde{G}_{jl;;k} H^{(0)}_{l;g;}$$

$$
\begin{aligned}
&= \sum_{j,k,l} \left(\hat{G}_{ij;;k} - \delta_{ij} \sum_m \hat{G}_{im;;k} \right) \left(\hat{G}_{jl;;k} - \delta_{jl} \sum_n \hat{G}_{jn;;k} \right) H^{(0)}_{l;g;} \\
&= \sum_{j,k,l} \hat{G}_{ij;;k} \hat{G}_{jl;;k} H^{(0)}_{l;g;} - \sum_{j,k,n} \hat{G}_{ij;;k} \hat{G}_{jn;;k} H^{(0)}_{j;g;} \\
&\quad - \sum_{k,l,m} \hat{G}_{im;;k} \hat{G}_{il;;k} H^{(0)}_{l;g;} + \sum_{k,m,n} \hat{G}_{im;;k} \hat{G}_{in;;k} H^{(0)}_{i;g;} \\
&= \sum_{j,k,l} \hat{G}_{ij;;k} \hat{G}_{jl;;k} H^{(0)}_{l;g;} - \sum_{j,k,n} \hat{G}_{ij;;k} \hat{G}_{jn;;k} H^{(0)}_{j;g;} \\
&\quad - \sum_{k,l,j} \hat{G}_{ij;;k} \hat{G}_{il;;k} H^{(0)}_{l;g;} + \sum_{k,j,n} \hat{G}_{ij;;k} \hat{G}_{in;;k} H^{(0)}_{i;g;} \\
&\qquad\qquad (\because \text{Change the dummy index } m \to j \text{ for the third and fourth terms}) \\
&= \sum_{j,k,l} \hat{G}_{ij;;k} (\hat{G}_{jl;;k} - \hat{G}_{il;;k}) (H^{(0)}_{l;g;} - H^{(0)}_{j;g;}) \\
&\qquad\qquad (\because \text{Change the dummy index } n \to l \text{ for the second and fourth terms}) \\
&= \left\langle \nabla^2 \boldsymbol{H}^{(0)} \right\rangle_{i;g;}. \qquad (3.59)
\end{aligned}
$$

3.3.4.6 Jacobian and Hessian Operators

Considering a similar discussion, we can show the following dependencies. For the Jacobian model, $\left\langle \nabla \otimes \boldsymbol{H}^{(1)} \right\rangle \in \mathbb{R}^{|\mathcal{V}| \times f \times d \times d}$,

$$
\begin{aligned}
\left\langle \nabla \otimes \boldsymbol{H}^{(1)} \right\rangle_{i;g;kl} &= \left\langle \frac{\partial \boldsymbol{H}^{(1)}}{\partial x_l} \right\rangle_{i;g;k} && (3.60) \\
&= \sum_j \hat{G}_{ij;;l} (H^{(1)}_{j;g;k} - H^{(1)}_{i;g;k}) && (3.61) \\
&= (\tilde{\boldsymbol{G}} \otimes \boldsymbol{H}^{(1)})_{i;g;lk}. && (3.62)
\end{aligned}
$$

For the Hessian model, $\left\langle \nabla \otimes \nabla \boldsymbol{H}^{(0)} \right\rangle \in \mathbb{R}^{|\mathcal{V}| \times f \times d \times d}$,

$$
\left\langle \nabla \otimes \nabla \boldsymbol{H}^{(0)} \right\rangle_{i;g;kl} = \left\langle \frac{\partial}{\partial x_k} \frac{\partial}{\partial x_l} \boldsymbol{H}^{(0)} \right\rangle_{i;g;} \qquad (3.63)
$$

$$= \sum_{j,m} \hat{G}_{ij;;k}[\hat{G}_{jm;;l}(H^{(0)}_{m;g;} - H^{(0)}_{l;g;}) - \hat{G}_{im;;l}(H^{(0)}_{m;g;} - H^{(0)}_{i;g;})] \quad (3.64)$$

$$= \left[(\tilde{\boldsymbol{G}} \otimes \tilde{\boldsymbol{G}}) * \boldsymbol{H}^{(0)}\right]_{i;g;kl}. \quad (3.65)$$

3.3.5 IsoGCN Modeling Details

To achieve $\mathrm{E}(n)$- invariance and equivariance, there are several rules to follow. Here, we describe the desired focus when constructing an IsoGCN model. In this section, a rank-p tensor denotes a tensor the rank of which is $p \geq 1$ and σ denotes a nonlinear activation function. $\boldsymbol{W}$ is a trainable weight matrix and $\boldsymbol{b}$ is a trainable bias.

3.3.5.1 Activation and Bias

As the nonlinear activation function is not $\mathrm{E}(n)$-equivariant, nonlinear activation to rank-p tensors cannot be applied, while one can apply any activation to rank-0 tensors. In addition, adding bias is also not $\mathrm{E}(n)$-equivariant, so one cannot add bias when performing an affine transformation to rank-p tensors. Again, one can add bias to rank-0 tensors.

Thus, for instance, if one converts from rank-0 tensors $\boldsymbol{H}^{(0)}$ to rank-1 tensors using IsoAM $\boldsymbol{G}$, $\boldsymbol{G} * \sigma(\boldsymbol{H}^{(0)}\boldsymbol{W} + \boldsymbol{b})$ and $(\boldsymbol{G} * \sigma(\boldsymbol{H}^{(0)}))\boldsymbol{W}$ are $\mathrm{E}(n)$-equivariant functions, however $(\boldsymbol{G} * \boldsymbol{H}^{(0)})\boldsymbol{W} + \boldsymbol{b}$ and $\sigma\left((\boldsymbol{G} * \sigma(\boldsymbol{H}^{(0)}))\boldsymbol{W}\right)$ are not due to the bias and the nonlinear activation, respectively. Likewise, regarding a conversion from rank-1 tensors $\boldsymbol{H}^{(1)}$ to rank-0 tensors, $\sigma\left((\boldsymbol{G} \odot \boldsymbol{H}^{(1)})\boldsymbol{W} + \boldsymbol{b}\right)$ and $\sigma\left(\boldsymbol{G} \odot (\boldsymbol{H}^{(1)}\boldsymbol{W})\right)$ are $\mathrm{E}(n)$-invariant functions; however, $\boldsymbol{G} \odot (\boldsymbol{H}^{(1)}\boldsymbol{W} + \boldsymbol{b})$ and $(\boldsymbol{G} \odot \sigma(\boldsymbol{H}^{(1)}))\boldsymbol{W} + \boldsymbol{b}$ are not.

To convert rank-p tensors to rank-q tensors ($q \geq 1$), one can apply neither bias nor nonlinear activation. To add nonlinearity to such a conversion, we can multiply the converted rank-0 tensors $\sigma((\bigotimes^p \boldsymbol{G} \odot \boldsymbol{H}^{(p)})\boldsymbol{W} + \boldsymbol{b})$ with the input tensors $\boldsymbol{H}^{(p)}$ or the output tensors $\boldsymbol{H}^{(q)}$.

3.3.5.2 Preprocessing of Input Feature

Similarly to the discussion regarding the biases, we have to take care of the preprocessing of rank-p tensors to retain $\mathrm{E}(n)$-invariance because adding a constant array and component-wise scaling could distort the tensors, resulting in broken $\mathrm{E}(n)$-equivariance.

For instance, $\boldsymbol{H}^{(p)}/\mathrm{Std}_{\mathrm{all}}\left[\boldsymbol{H}^{(p)}\right]$ is a valid transformation to retain $\mathrm{E}(n)$-equivariance, assuming $\mathrm{Std}_{\mathrm{all}}\left[\boldsymbol{H}^{(p)}\right] \in \mathbb{R}$ is a standard deviation of all components of $\boldsymbol{H}^{(p)}$. However, conversions such as $\boldsymbol{H}^{(p)}/\mathrm{Std}_{\mathrm{component}}\left[\boldsymbol{H}^{(p)}\right]$ and $\boldsymbol{H}^{(p)} - \mathrm{Mean}\left[\boldsymbol{H}^{(p)}\right]$ are not $\mathrm{E}(n)$-equivariant, assuming that $\mathrm{Std}_{\mathrm{component}}\left[\boldsymbol{H}^{(p)}\right] \in \mathbb{R}^{d^p}$ is a component-wise standard deviation.

3.3.5.3 Scaling

Because the differential IsoAM $\tilde{\boldsymbol{G}}$ corresponds to the differential operator, the scale of the output after operations regarding $\tilde{D}$ can be huge. Thus, we rescale $\tilde{\boldsymbol{G}}$ using the scaling factor $\left[\mathrm{Mean}_{\mathrm{sample},i}(\tilde{G}^2_{ii;;1} + \tilde{G}^2_{ii;;2} + \tilde{G}^2_{ii;;3})\right]^{1/2}$, where $\mathrm{Mean}_{\mathrm{sample},i}$ denotes the mean over the samples and vertices.

3.3.5.4 Tensor Rank

Although we defined $\mathrm{IsoGCN}_{p\to q}$ in Equations 3.42 and 3.43, there are other ways to model function converting from rank-p discrete tensor field to rank-q discrete tensor field. For instance, in the case of $p = 2$ and $q = 3$, one may also define as:

$$\boldsymbol{H}^{(3)}_{\mathrm{out}} = \mathrm{EquivariantPointwiseMLP}\left(\boldsymbol{G} \otimes \boldsymbol{G} \odot \boldsymbol{G} \otimes \boldsymbol{H}^{(2)}_{\mathrm{in}}\right), \tag{3.66}$$

or

$$\boldsymbol{H}^{(3)}_{\mathrm{out}} = \mathrm{IsoGCN}_{4\to 3} \circ \mathrm{IsoGCN}_{3\to 4} \circ \mathrm{IsoGCN}_{2\to 3}(\boldsymbol{H}^{(2)}_{\mathrm{in}}). \tag{3.67}$$

One guideline is to consider PDEs of interest when using the differential IsoAM, as done in the numerical experiments (Section 3.4). In the other case, generally, tensor rank should not be dropped unless necessary. Namely, transformation of the tensor rank $2 \to 3 \to 4 \to 3$ is more preferable compared to that of $2 \to 1 \to 0 \to 1 \to 2 \to 3$ to constract the IsoGCN model which converts a rank-2 discrete tensor field to a rank-3 discrete tensor field.

3.3.5.5 Implementation

Because an adjacency matrix $\boldsymbol{A}$ is usually a sparse matrix for a regular mesh, $\boldsymbol{A}(m)$ in equation 3.47 is also a sparse matrix for a sufficiently small m. Thus, we can leverage

sparse matrix multiplication in the IsoGCN computation. This is one major reason why IsoGCNs can compute rapidly. If the multiplication (tensor product or contraction) of IsoAMs must be computed multiple times the associative property of the IsoAM can be utilized.

For instance, it is apparent that

$$\left[\bigotimes^{k} \boldsymbol{G}\right] * \boldsymbol{H}^{(0)} = \boldsymbol{G} \otimes (\boldsymbol{G} \otimes \ldots (\boldsymbol{G} * \boldsymbol{H}^{(0)})). \tag{3.68}$$

Assuming that the number of nonzero elements in $\boldsymbol{A}(m)$ equals n and $\boldsymbol{H}^{(0)} \in \mathbb{R}^{|\mathcal{V}| \times f}$, then the computational complexity of the right-hand side is $\mathcal{O}(n|\mathcal{V}|fn^k)$. This is an exponential order regarding the spatial dimension n. However, n and k are usually small numbers (typically $n = 3$ and $k \leq 4$). Therefore one can compute an IsoGCN layer with a realistic spatial dimension n and tensor rank k fast and memory efficiently. In our implementation, both a sparse matrix operation and associative property are utilized to realize fast computation.

3.4 Numerical Experiments

To test the applicability of the proposed model, we composed the following two datasets:

1. a differential operator dataset of grid meshes; and

2. an anisotropic nonlinear heat equation dataset of meshes generated from CAD data.

In this section, we discuss our machine learning model, the definition of the problem, and the results for each dataset.

Using $\tilde{\boldsymbol{G}}$ defined in Section 3.3.4, we constructed a neural network model considering an encode-process-decode configuration (Battaglia et al., 2018). The encoder and decoder were comprised of component-wise MLPs and tensor operations. For each task, we tested $m = 2, 5$ in Equation 3.47 to investigate the effect of the number of hops considered.

In addition to the GCN (Kipf & Welling, 2017), we chose GIN (Xu et al., 2018), SGCN (Wu et al., 2019), Cluster-GCN (Chiang et al., 2019), and GCNII (Chen et al., 2020) as GCN variant baseline models.

For the equivariant models, we chose the TFN (Thomas et al., 2018) and SE(3)-Transformer (Fuchs et al., 2020) as the baseline. We implemented these models using PyTorch 1.6.0 (Paszke et al., 2019) and PyTorch Geometric 1.6.1 (Fey & Lenssen, 2019). For both the TFN and SE(3)-Transformer, we used implementation of Fuchs et al. (2020)[3] because the computation of the TFN is considerably faster than the original implementation, as claimed in Fuchs et al. (2020). For each experiment, we minimized the mean squared loss using the Adam optimizer (Kingma & Ba, 2014). The corresponding implementation and the dataset will be made available online.

3.4.1 Differential Operator Dataset

3.4.1.1 Task Definition

To demonstrate the expressive power of IsoGCNs, we created a dataset to learn the differential operators. We first generated a pseudo-2D grid mesh randomly with only one cell in the Z direction and 10 to 100 cells in the X and Y directions. We then generated scalar fields on the grid meshes and analytically calculated the gradient, Laplacian, and Hessian fields. We generated 100 samples for each train, validation, and test dataset.

For simplicity, we set $w_{ij} = 1$ in Equation 3.47 for all $(i, j) \in \mathcal{E}$. To compare the performance with the GCN models, we simply replaced an IsoGCN layer with a GCN or its variant layers while keeping the number of hops m the same to enable a fair comparison. We adjusted the hyperparameters for the equivariant models to ensure that the number of parameters in each was almost the same as that in the IsoGCN model.

We conducted the experiments using the following settings:

1. inputting the scalar field ϕ and predicting the gradient field $\nabla\phi$ (rank-0 $\rightarrow$ rank-1 tensor);

[3] `https://github.com/FabianFuchsML/se3-transformer-public`

2. inputting the scalar field ϕ and predicting the Hessian field $\nabla \otimes \nabla\phi$ (rank-0 $\rightarrow$ rank-2 tensor);

3. inputting the gradient field $\nabla\phi$ and predicting the Laplacian field $\nabla \cdot \nabla\phi$ (rank-1 $\rightarrow$ rank-0 tensor); and

4. inputting the gradient field $\nabla\phi$ and predicting the Hessian field $\nabla \otimes \nabla\phi$ (rank-1 $\rightarrow$ rank-2 tensor).

3.4.1.2 Model Architectures

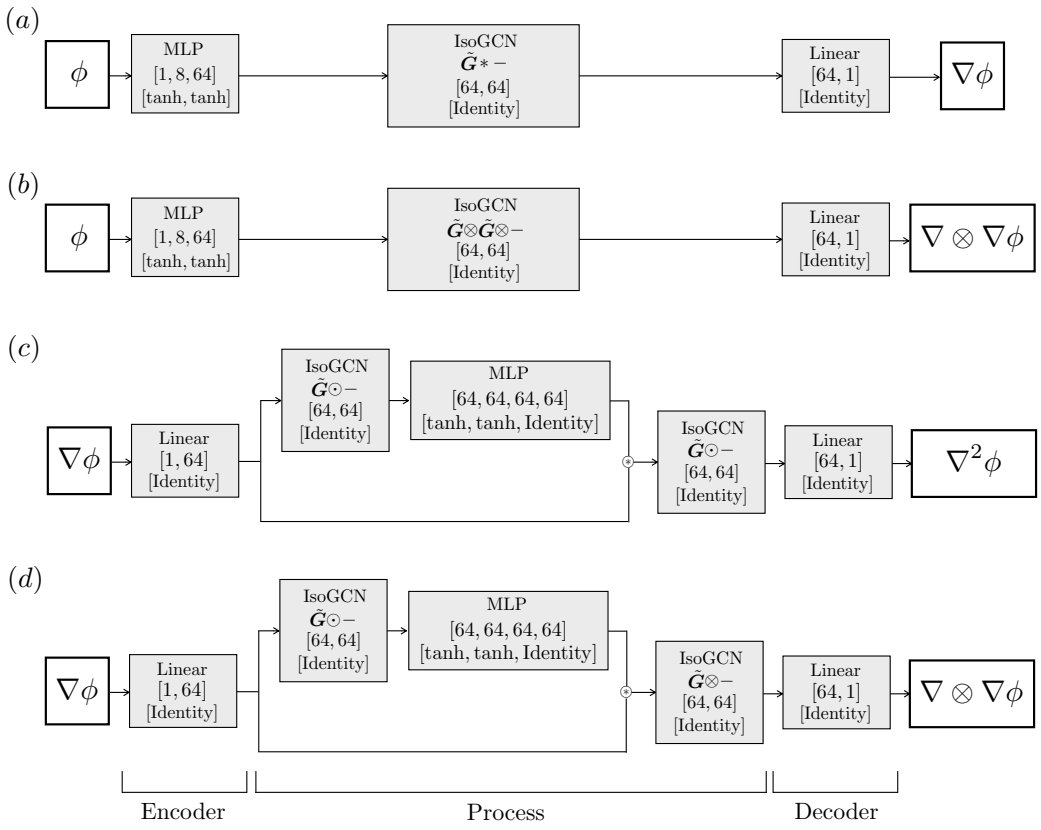


Figure 3.2: The IsoGCN models used for (a) the scalar field to the gradient field, (b) the scalar field to the Hessian field, (c) the gradient field to the Laplacian field, (d) the gradient field to the Hessian field of the gradient operator dataset. Gray boxes are trainable components. In each trainable cell, we put the number of units in each layer along with the activation functions used. $\circledast$ denotes the multiplication in the feature direction.

Figure 3.2 represents the IsoGCN model used for the differential operator dataset. We used the $\tanh$ activation function as a nonlinear activation function because we expect the target temperature field to be smooth. Therefore, we avoid using non-differentiable activation functions such as the rectified linear unit (ReLU) (Nair & Hinton, 2010). For GCN and its variants, we simply replaced the IsoGCN layers with the corresponding ones. We stacked $m\ (= 2, 5)$ layers for GCN, GIN, GCNII, and Cluster-GCN. We used an m hop adjacency matrix for SGCN.

For the TFN and SE(3)-Transformer, we set the hyperparameters to have almost the same number of parameters as in the IsoGCN model. The settings of the hyperparameters are shown in Table 3.2.

Table 3.2: Summary of the hyperparameter setting for both the TFN and SE(3)-Transformer. For the parameters not in the table, we used the default setting in the implementation of `https://github.com/FabianFuchsML/se3-transformer-public`.

	$\mathbf{0 \to 1}$	$\mathbf{0 \to 2}$	$\mathbf{1 \to 0}$	$\mathbf{1 \to 2}$
# hidden layers	1	1	1	1
# NL layers in the self-interaction	1	1	1	1
# channels	24	20	24	24
# maximum rank of the hidden layers	1	2	1	2
# nodes in the radial function	16	8	16	22

3.4.1.3 RESULTS

Figure 3.3 and Table 3.3 present a visualization and comparison of predictive performance, respectively. The results show that an IsoGCN outperforms other GCN models for all settings. This is because the IsoGCN model has information on the relative position of the adjacency vertices, and thus understands the direction of the gradient, whereas the other GCN models cannot distinguish where the adjacencies are, making it nearly impossible to predict the gradient directions.

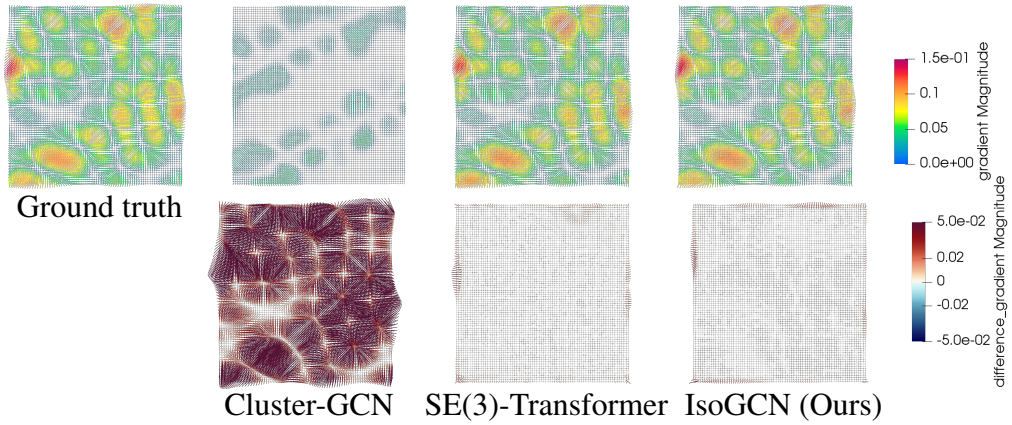


Figure 3.3: (Top) the gradient field and (bottom) the error vector between the prediction and the ground truth of a test data sample. The error vectors are exaggerated by a factor of 2 for clear visualization.

Adding the vertex positions to the input feature to other GCN models exhibited a performance improvement, however as the vertex position is not a translation invariant feature, it could degrade the predictive performance of the models. Thus, we did not input $\boldsymbol{x}$ as a vertex feature to the IsoGCN model or other equivariant models to retain their $\mathrm{E}(n)$- invariant and equivariant natures.

IsoGCNs perform competitively against other equivariant models with shorter prediction time as shown in Table 3.4. As mentioned in Section 3.3.4, $\tilde{\boldsymbol{G}}$ corresponds to the gradient operator, which is now confirmed in practice. Therefore, it can be found out the proposed model has a strong expressive power to express differential regarding space with less computation resources compared to the TFN and SE(3)-Transformer.

Table 3.3: Summary of the test losses (mean squared error $\pm$ the standard error of the mean in the original scale) of the differential operator dataset: $0 \to 1$ (the scalar field to the gradient field), $0 \to 2$ (the scalar field to the Hessian field), $1 \to 0$ (the gradient field to the Laplacian field), and $1 \to 2$ (the gradient field to the Hessian field). Here, if "$\boldsymbol{x}$" is "Yes", $\boldsymbol{x}$ is also in the input feature.

Method	**# hops**	$\boldsymbol{x}$	**Loss of $0 \to 1$** $\times 10^{-5}$	**Loss of $0 \to 2$** $\times 10^{-6}$	**Loss of $1 \to 0$** $\times 10^{-6}$	**Loss of $1 \to 2$** $\times 10^{-6}$
GIN	2	No	151.19 ± 0.53	49.10 ± 0.36	542.52 ± 2.14	59.65 ± 0.46
	2	Yes	147.10 ± 0.51	47.56 ± 0.35	463.79 ± 2.08	50.73 ± 0.40
	5	No	151.18 ± 0.53	48.99 ± 0.36	542.54 ± 2.14	59.64 ± 0.46
	5	Yes	147.07 ± 0.51	47.35 ± 0.35	404.92 ± 1.74	46.18 ± 0.39
GCNII	2	No	151.18 ± 0.53	43.08 ± 0.31	542.74 ± 2.14	59.65 ± 0.46
	2	Yes	151.14 ± 0.53	40.72 ± 0.29	194.65 ± 1.00	45.43 ± 0.36
	5	No	151.11 ± 0.53	32.85 ± 0.23	542.65 ± 2.14	59.66 ± 0.46
	5	Yes	151.13 ± 0.53	31.87 ± 0.22	280.61 ± 1.30	39.38 ± 0.34
SGCN	2	No	151.17 ± 0.53	50.26 ± 0.38	542.90 ± 2.14	59.65 ± 0.46
	2	Yes	151.12 ± 0.53	49.96 ± 0.37	353.29 ± 1.49	59.61 ± 0.46
	5	No	151.12 ± 0.53	55.02 ± 0.42	542.73 ± 2.14	59.64 ± 0.46
	5	Yes	151.16 ± 0.53	55.08 ± 0.42	127.21 ± 0.63	56.97 ± 0.44
GCN	2	No	151.23 ± 0.53	49.59 ± 0.37	542.54 ± 2.14	59.64 ± 0.46
	2	Yes	151.14 ± 0.53	47.91 ± 0.35	542.68 ± 2.14	59.60 ± 0.46
	5	No	151.18 ± 0.53	50.58 ± 0.38	542.53 ± 2.14	59.64 ± 0.46
	5	Yes	151.14 ± 0.53	48.50 ± 0.35	542.30 ± 2.14	25.37 ± 0.28
Cluster-GCN	2	No	151.19 ± 0.53	33.39 ± 0.24	542.54 ± 2.14	59.66 ± 0.46
	2	Yes	147.23 ± 0.51	32.29 ± 0.24	167.73 ± 0.83	17.72 ± 0.17
	5	No	151.15 ± 0.53	28.79 ± 0.21	542.51 ± 2.14	59.66 ± 0.46
	5	Yes	146.91 ± 0.51	26.60 ± 0.19	185.21 ± 0.99	18.18 ± 0.20
TFN	2	No	2.47 ± 0.02	OOM	26.69 ± 0.24	OOM
	5	No	OOM	OOM	OOM	OOM
SE(3)-Trans.	2	No	**1.79** ± 0.02	**3.50** ± 0.04	**2.52** ± 0.02	OOM
	5	No	2.12 ± 0.02	OOM	7.66 ± 0.05	OOM
IsoGCN (Ours)	2	No	2.67 ± 0.02	6.37 ± 0.07	7.18 ± 0.06	**1.44** ± 0.02
	5	No	14.19 ± 0.10	21.72 ± 0.25	34.09 ± 0.19	8.32 ± 0.09

Table 3.4: Summary of the prediction time on the test dataset. $0 \to 1$ corresponds to the scalar field to the gradient field, and $0 \to 2$ corresponds to the scalar field to the Hessian field. Each computation was run on the same GPU (NVIDIA Tesla V100 with 32 GiB memory). OOM denotes the out-of-memory of the GPU.

	$\mathbf{0 \to 1}$		$\mathbf{0 \to 2}$	
Method	**# parameters**	**Inference time [s]**	**# parameters**	**Inference time [s]**
TFN	5264	3.8	5220	OOM
SE(3)-Trans.	5392	4.0	5265	9.2
IsoGCN (Ours)	4816	0.4	4816	0.7

3.4.2 Anisotropic Nonlinear Heat Equation Dataset

3.4.2.1 Task Definition

To apply the proposed model to a real problem, we adopted the anisotropic nonlinear heat equation. We considered the task of predicting the time evolution of the temperature field based on the initial temperature field, material property, and mesh geometry information as inputs. We randomly selected 82 CAD shapes from the first 200 shapes of the ABC dataset (Koch et al., 2019), generate first-order tetrahedral meshes using a mesh generator program, Gmsh (Geuzaine & Remacle, 2009), randomly set the initial temperature and anisotropic thermal conductivity, and finally conducted a finite element analysis (FEA) using the FEA program FrontISTR[4] (Morita et al., 2016; Ihara et al., 2017).

For this task, we set

$$w_{ij} = V_j^{\text{effective}} / V_i^{\text{effective}}, \tag{3.69}$$

where $V_i^{\text{effective}}$ denotes the effective volume of the ith vertex (Equation 3.74.) Similarly to the differential operator dataset, we tested the number of hops $m = 2, 5$. However because we put four IsoAM operations in one model, the number of hops visible from the model is 8 ($m = 2$) or 20 ($m = 5$). As is the case with the differential operator dataset, we replaced an IsoGCN layer accordingly for GCN or its variant models.

In the case of $k = 2$, we reduced the number of parameters for each of the baseline equivariant models to fewer than the IsoGCN model because they exceeded the memory of the GPU (NVIDIA Tesla V100 with 32 GiB memory) with the same number of parameters. In the case of $k = 5$, neither the TFN nor the SE(3)-Transformer fits into the memory of the GPU even with the number of parameters equal to 10. For more details about the dataset and the model, see Section 3.4.2.

[4] `https://github.com/FrontISTR/FrontISTR`. We applied a private update to FrontISTR to deal with the anisotropic heat problem, which will be also made available online.

3.4.2.2 DATASET

The purpose of the experiment was to solve the anisotropic nonlinear heat diffusion under an adiabatic boundary condition. The governing equation is defined as follows:

$$\Omega \subset \mathbb{R}^3 \tag{3.70}$$

$$\frac{\partial T(t,\boldsymbol{x})}{\partial t} = \nabla \cdot \boldsymbol{C}(T(t,\boldsymbol{x}))\nabla T(t,\boldsymbol{x}) \qquad \text{in } \Omega \tag{3.71}$$

$$T(t=0,\boldsymbol{x}) = T_{0.0}(\boldsymbol{x}) \qquad \text{in } \Omega \tag{3.72}$$

$$\nabla T(t,\boldsymbol{x})|_{\boldsymbol{x}=\boldsymbol{x}_b} \cdot \boldsymbol{n}(\boldsymbol{x}_b) = 0 \qquad \text{on } \partial\Omega, \tag{3.73}$$

where T is the temperature field, $T_{0.0}$ is the initial temperature field, $\boldsymbol{C} \in \mathbb{R}^{d\times d}$ is an anisotropic diffusion tensor and $\boldsymbol{n}(\boldsymbol{x}_b)$ is the normal vector at $\boldsymbol{x}_b \in \partial\Omega$. The Neumann boundary condition expressed in Equation 3.73 corresponds to the adiabatic condition.

Here, $\boldsymbol{C}$ depends on temperature thus the equation is nonlinear. We randomly generate $\boldsymbol{C}(T=-1)$ for it to be a positive semidefinite symmetric tensor with eigenvalues varying from 0.0 to 0.02. Then, we defined the linear temperature dependency the slope of which is $-\boldsymbol{C}(T=-1)/4$. The function of the anisotropic diffusion tensor is uniform for each sample.

The task is defined to predict the temperature field at $t = 0.2, 0.4, 0.6, 1.0$ ($T_{0.2}, T_{0.4}, T_{0.6}, T_{0.8}, T_{1.0}$) from the given initial temperature field $T_{0.9}$, material property, and mesh geometry. However, the performance is evaluated only with $T_{1.0}$ to focus on the predictive performance. We inserted other output features to stabilize the trainings. Accordingly, the diffusion number of this problem is $\boldsymbol{C}\Delta t/(\Delta x)^2 \simeq 10.0^4$ assuming $\Delta x \simeq 10.0^{-3}$.

Figure 3.4 represents the process of generating the dataset. We generated up to 9 FEA results for each CAD shape. To avoid data leakage in terms of the CAD shapes, we first split them into training, validation, and test datasets, and then applied the following process.

Using one CAD shape, we generated up to three meshes using clscale (a control parameter of the mesh characteristic lengths) = 0.20, 0.25, and 0.30. To facilitate the training process, we scaled the meshes to fit into a cube with an edge length equal to 1.

Using one mesh, we generated three initial conditions randomly using a Fourier series of the 2nd to 10th orders. We then applied an FEA to each initial condition and material property determined randomly as described above. We applied an implicit method to solve time evolutions and a direct method to solve the linear equations. The FEA time step Δt was set to 0.01.

During this process, some of the meshes or FEA results may not have been available due to excessive computation time or non-convergence. Therefore, the size of the dataset was not exactly equal to the number multiplied by 9. Finally, we obtained 439 FEA results for the training dataset, 143 FEA results for the validation dataset, and 140 FEA results for the test dataset.

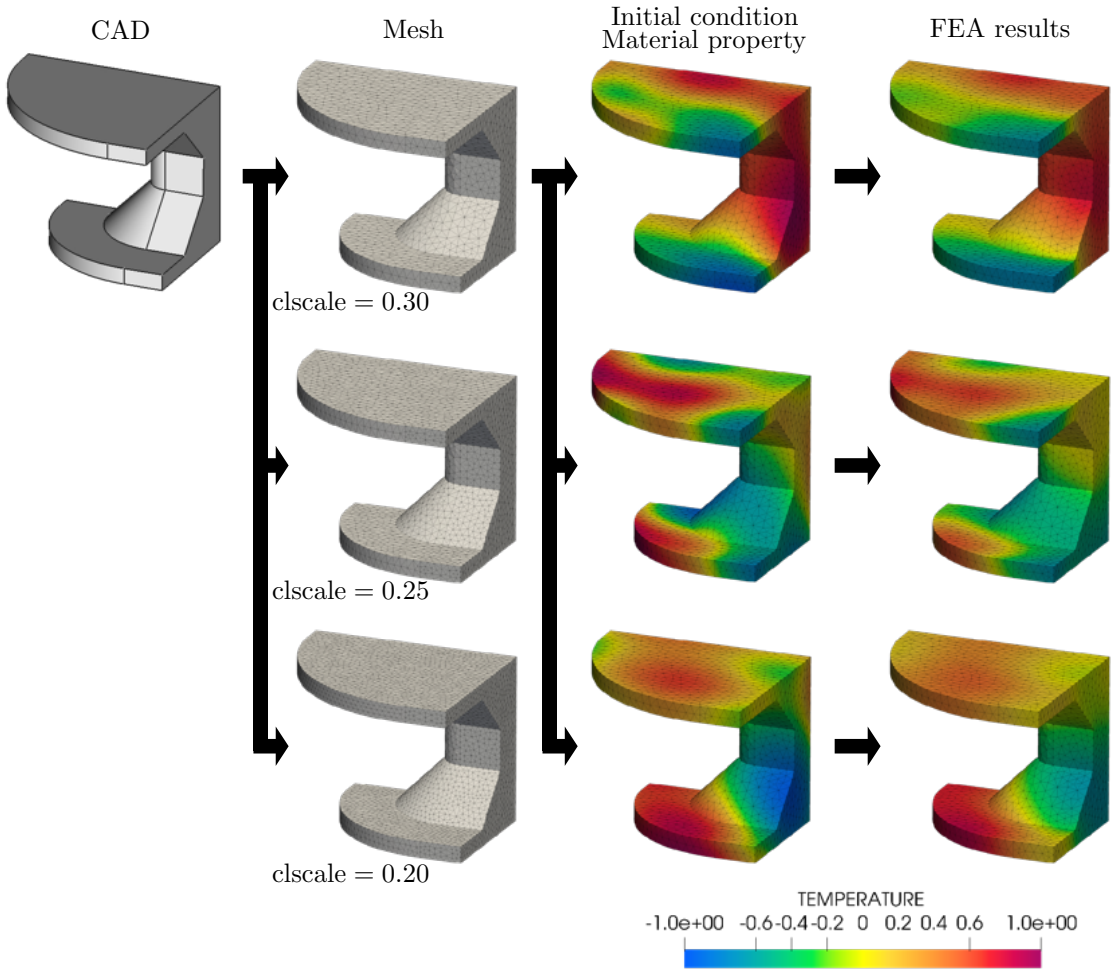


Figure 3.4: The process of generating the dataset. A smaller clscale parameter generates smaller meshes.

3.4.2.3 Input and Output Features

To express the geometry information, we extracted the effective volume of the ith vertex $V_i^{\text{effective}}$ and the mean volume of the ith vertex V_i^{mean}, which are defined as follows:

$$V_i^{\text{effective}} = \sum_{e \in \mathcal{N}_i^e} \frac{1}{4} V_e \tag{3.74}$$

$$V_i^{\text{mean}} = \frac{\sum_{e \in \mathcal{N}_i^e} V_e}{|\mathcal{N}_i^e|}, \tag{3.75}$$

where $\mathcal{N}_i^e$ is the set of elements, including the ith vertex.

For GCN or its variant models, we tested several combinations of input vertex features $T_{0.0}$, $\boldsymbol{C}$, $V^{\text{effective}}$, V^{mean}, and $\boldsymbol{x}$ (Table 3.6). For the IsoGCN model, inputs were $T_{0.0}$, $V^{\text{effective}}$, V^{mean}, and $\boldsymbol{C}$. Since we construct define the discrete tensor field for each tensor rank, we have

$$\boldsymbol{H}_{\text{in}}^{(0)} = \begin{pmatrix} T_{0.0}(\boldsymbol{x}_1) & V^{\text{effective}}(\boldsymbol{x}_1) & V_{\text{mean}}(\boldsymbol{x}_1) \\ T_{0.0}(\boldsymbol{x}_2) & V^{\text{effective}}(\boldsymbol{x}_2) & V_{\text{mean}}(\boldsymbol{x}_2) \\ \vdots & \vdots & \vdots \\ T_{0.0}(\boldsymbol{x}_{|\mathcal{V}|}) & V^{\text{effective}}(\boldsymbol{x}_{|\mathcal{V}|}) & V_{\text{mean}}(\boldsymbol{x}_{|\mathcal{V}|}) \end{pmatrix} \in \mathbb{R}^{|\mathcal{V}| \times 3 \times n^0} \tag{3.76}$$

$$\boldsymbol{H}_{\text{in}}^{(2)} = \left.\begin{pmatrix} \boldsymbol{C} \\ \boldsymbol{C} \\ \vdots \\ \boldsymbol{C} \end{pmatrix}\right\} |\mathcal{V}| \text{ rows} \in \mathbb{R}^{|\mathcal{V}| \times 1 \times n^2}, \tag{3.77}$$

for input discrete tensor fields and

$$\boldsymbol{H}_{\mathrm{out}}^{(0)} = \begin{pmatrix} T_{0.2}(\boldsymbol{x}_1) & T_{0.4}(\boldsymbol{x}_1) & T_{0.6}(\boldsymbol{x}_1) & T_{0.8}(\boldsymbol{x}_1) & T_{1.0}(\boldsymbol{x}_1) \\ T_{0.2}(\boldsymbol{x}_2) & T_{0.4}(\boldsymbol{x}_2) & T_{0.6}(\boldsymbol{x}_2) & T_{0.8}(\boldsymbol{x}_2) & T_{1.0}(\boldsymbol{x}_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ T_{0.2}(\boldsymbol{x}_{|\mathcal{V}|}) & T_{0.4}(\boldsymbol{x}_{|\mathcal{V}|}) & T_{0.6}(\boldsymbol{x}_{|\mathcal{V}|}) & T_{0.8}(\boldsymbol{x}_{|\mathcal{V}|}) & T_{1.0}(\boldsymbol{x}_{|\mathcal{V}|}) \end{pmatrix} \in \mathbb{R}^{|\mathcal{V}| \times 5 \times n^0} \tag{3.78}$$

for the output discrete tensor field in the present task.

3.4.2.4 Model Architectures

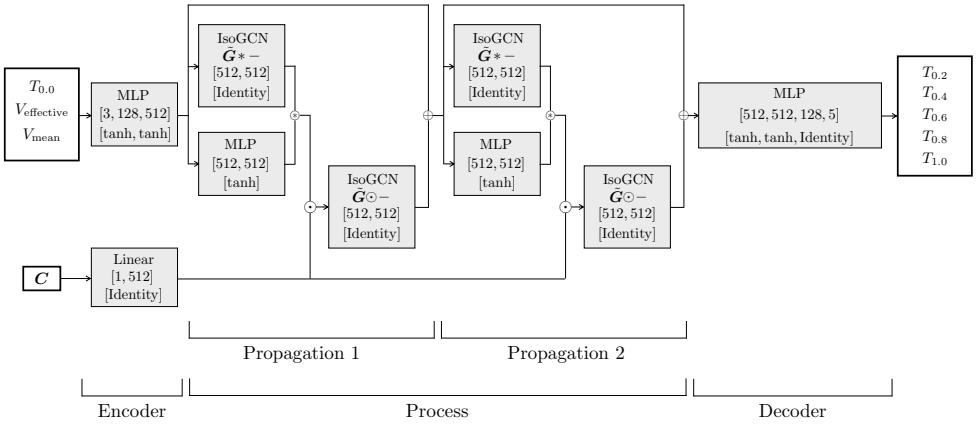


Figure 3.5: The IsoGCN model used for the anisotropic nonlinear heat equation dataset. Gray boxes are trainable components. In each trainable cell, we put the number of units in each layer along with the activation functions used. Below the unit numbers, the activation function used for each layer is also shown. $\circledast$ denotes the multiplication in the feature direction, $\odot$ denotes the contraction, and $\oplus$ denotes the addition in the feature direction.

Figure 3.5 represents the IsoGCN model used for the anisotropic nonlinear heat equation dataset. We adopted the encode-process-decode configuration (Battaglia et al., 2018) to leverage the expressive power of neural networks. The encoder embeds the input features to a higher dimensional space, 512 dimension in the present case. By increasing the

dimension of the encoded space, one can expect that the expressive power increases. The decoder takes the embedded features and outputs features in the desired dimension.

The process part contains two propagation blocks. Although the propagation block looks complicated, one can see it corresponds to the explicit Euler method (Equation 2.58):

$$T(t + \Delta t, \boldsymbol{x}) \approx T(t, \boldsymbol{x}) + \nabla \cdot \boldsymbol{C}(T(t, \boldsymbol{x}))\nabla T(t, \boldsymbol{x})\Delta t, \tag{3.79}$$

because one propagation block is expressed as

$$\mathrm{Propagation}_i(\boldsymbol{H}^{(0)}, \boldsymbol{H}^{(2)}) = \boldsymbol{H}^{(0)} + \tilde{\boldsymbol{G}} \odot \boldsymbol{H}^{(2)} \odot \mathrm{MLP}(\boldsymbol{H}^{(0)})\tilde{\boldsymbol{G}} * \boldsymbol{H}^{(0)}, \tag{3.80}$$

where $\boldsymbol{H}^{(0)}$ and $\boldsymbol{H}^{(2)}$ denotes the rank-0 and rank-2 tensor inputs to the considered propagation block ($i = 1, 2$). Thus, one propagation block proceeds time Δt because of the relationship to the Euler method. By stacking this propagation block r times, we can make time evolution by $r\Delta t$, making it possible to predict the state after the long time. However, increasing r may cause longer computation time. Therefore, we keep $r = 2$ for the experiment to retain computational efficiency.

For the nonlinear activation function, we used $\tanh$ because we expect the target temperature field to be smooth. Therefore, we avoid using non-differentiable activation functions such as the rectified linear unit (ReLU) (Nair & Hinton, 2010).

For GCN and its variants, we simply replaced the IsoGCN layers with the corresponding ones. We stacked m $(= 2, 5)$ layers for GCN, GIN, GCNII, and Cluster-GCN. We used an m hop adjacency matrix for SGCN.

For the TFN and SE(3)-Transformer, we set the hyperparameters to as many parameters as possible that would fit on the GPU because the TFN and SE(3)-Transformer with almost the same number of parameters as in IsoGCN did not fit on the GPU we used (NVIDIA Tesla V100 with 32 GiB memory). The settings of the hyperparameters are shown in Table 3.5.

Table 3.5: Summary of the hyperparameter setting for both the TFN and SE(3)-Transformer. For the parameters not written in the table, we used the default setting in the implementation of `https://github.com/FabianFuchsML/se3-transformer-public`.

# hidden layers	1
# NL layers in the self-interaction	1
# channels	16
# maximum rank of the hidden layers	2
# nodes in the radial function	32

3.4.2.5 RESULTS

Figure 3.6 and Table 3.6 present the results of the qualitative and quantitative comparisons for the test dataset. The IsoGCN demonstrably outperforms all other baseline models. Moreover, owing to the computationally efficient $\mathrm{E}(n)$-invariant nature of IsoGCNs, it also achieved a high prediction performance for the meshes that had a significantly larger graph than those considered in the training dataset. The IsoGCN can scale up to 1M vertices, which is practical and is considerably greater than that reported in Sanchez-Gonzalez et al. (2020). Therefore, we conclude that IsoGCN models can be trained on relatively smaller meshes[5] to save the training time and then used to apply the inference to larger meshes without observing significant performance deterioration.

Table 3.7 reports the preprocessing and inference computation time using the equivariant models with $m = 2$ as the number of hops and FEA using FrontISTR 5.0.0. We varied the time step ($\Delta t = 1.0, 0.5$) for the FEA computation to compute the $t = 1.0$ time evolution thus, resulting in different computation times and errors compared to an FEA with $\Delta t = 0.01$, which was considered as the ground truth. Clearly, the IsoGCN is 3- to 5- times faster than the FEA with the same level of accuracy, while other equivariant models have almost the same speed as FrontISTR with $\Delta t = 0.5$.

The results show that the inclusion of $\boldsymbol{x}$ in the input features of the baseline models did not improve the performance. In addition, if $\boldsymbol{x}$ is included in the input features, a loss of the

[5]However, it should also be sufficiently large to express sample shapes and fields.

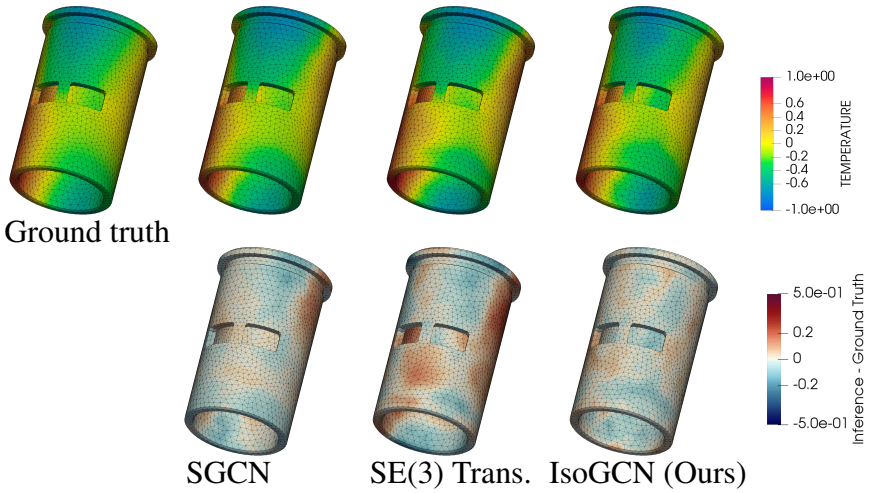


Figure 3.6: (Top) the temperature field of the ground truth and inference results and (bottom) the error between the prediction and the ground truth of a test data sample. The error is exaggerated by a factor of 2 for clear visualization.

generalization capacity for larger shapes compared to the training dataset may result as it extrapolates. The proposed model achieved the best performance compared to the baseline models considered. Therefore, we concluded that the essential features regarding the mesh shapes are included in $\tilde{\boldsymbol{G}}$.

Besides, IsoGCN can scale up to meshes with 1M vertices as shown in Figure 3.7. The result is surprizing because we trained relatively smaller meshes with several thousand vertices. Since IsoGCN successfully includes the information of the PDE, it can show such a high generalizability.

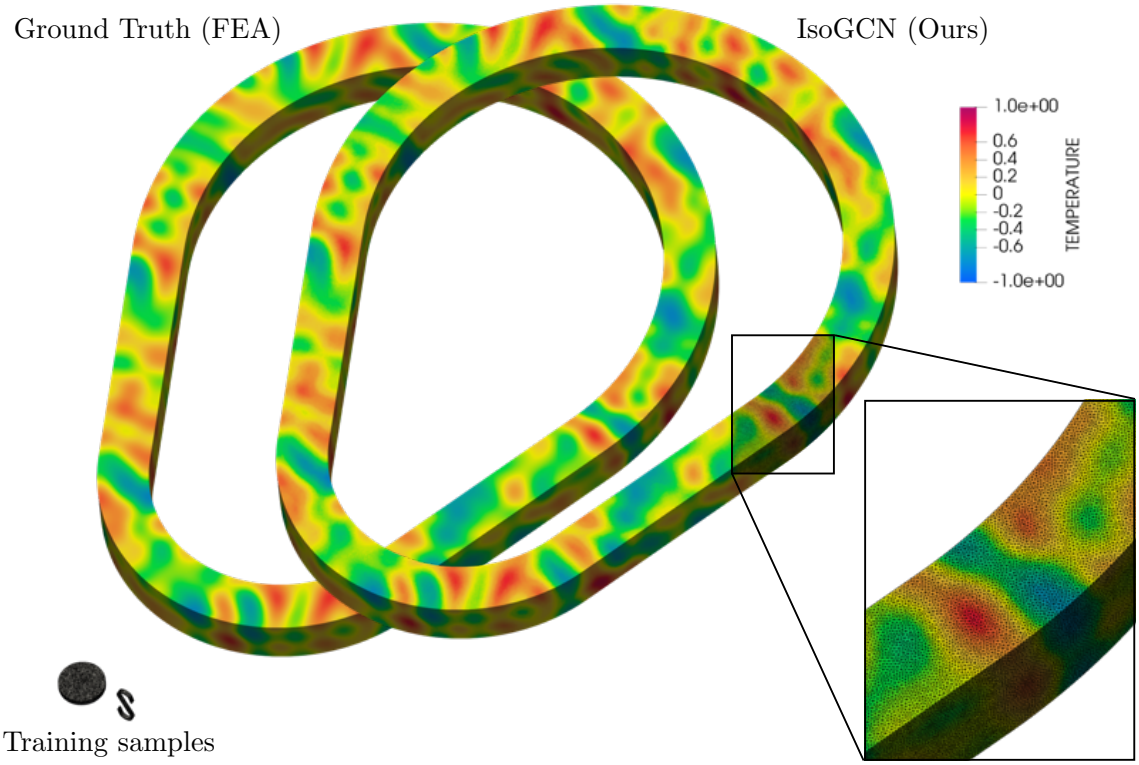


Figure 3.7: Comparison between (left) samples in the training dataset, (center) ground truth computed through FEA, and (right) IsoGCN inference result. For both the ground truth and inference result, $|\mathcal{V}| = 1,011,301$. One can see that IsoGCN can predict the temperature field for a mesh, which is much larger than these in the training dataset.

Table 3.6: Summary of the test losses (mean squared error $\pm$ the standard error of the mean in the original scale) of the anisotropic nonlinear heat dataset. Here, if "$\boldsymbol{x}$" is "Yes", $\boldsymbol{x}$ is also in the input feature. OOM denotes the out-of-memory on the applied GPU (32 GiB).

Method	**# hops**	$\boldsymbol{x}$	**Loss** $\times 10^{-3}$
GIN	2	No	16.921 ± 0.040
	2	Yes	18.483 ± 0.025
	5	No	22.961 ± 0.056
	5	Yes	17.637 ± 0.046
GCN	2	No	10.427 ± 0.028
	2	Yes	11.610 ± 0.032
	5	No	12.139 ± 0.031
	5	Yes	11.404 ± 0.032
GCNII	2	No	9.595 ± 0.026
	2	Yes	9.789 ± 0.028
	5	No	8.377 ± 0.024
	5	Yes	9.172 ± 0.028
Cluster-GCN	2	No	7.266 ± 0.021
	2	Yes	8.532 ± 0.023
	5	No	8.680 ± 0.024
	5	Yes	10.712 ± 0.030
SGCN	2	No	7.317 ± 0.021
	2	Yes	9.083 ± 0.026
	5	No	6.426 ± 0.018
	5	Yes	6.519 ± 0.020
TFN	2	No	15.661 ± 0.019
	5	No	OOM
SE(3)-Trans.	2	No	14.164 ± 0.018
	5	No	OOM
IsoGCN (Ours)	2	No	4.674 ± 0.014
	5	No	$\mathbf{2.470} \pm 0.008$

Table 3.7: Comparison of computation time. To generate the test data, we sampled CAD data from the test dataset and then generated the mesh for the graph to expand while retaining the element volume at almost the same size. The initial temperature field and the material properties are set randomly using the same methodology as the dataset sample generation. For a fair comparison, each computation was run on the same CPU (Intel Xeon E5-2695 v2@2.40GHz) using one core, and we excluded file I/O time from the measured time. OOM denotes the out-of-memory (500 GiB).

	$\|\mathcal{V}\| = 21,289$		$\|\mathcal{V}\| = 155,019$		$\|\mathcal{V}\| = 1,011,301$	
Method	**Loss** $\times 10^{-4}$	**Time [s]**	**Loss** $\times 10^{-4}$	**Time [s]**	**Loss** $\times 10^{-4}$	**Time [s]**
FrontISTR ($\Delta t = 1.0$)	10.9	16.7	6.1	181.7	2.9	1656.5
FrontISTR ($\Delta t = 0.5$)	0.8	30.5	0.4	288.0	0.2	2884.2
TFN	77.9	46.1	30.1	400.9	OOM	OOM
SE(3)-Transformer	111.4	31.2	80.3	271.1	OOM	OOM
IsoGCN (Ours)	8.1	**7.4**	4.9	**84.1**	3.9	**648.4**

3.5 CONCLUSION

In this chapter, we introduced the GCN-based $\mathrm{E}(n)$- invariant and equivariant models called IsoGCN. We discussed the differential IsoAM, an isometric adjacency matrix (IsoAM) for numerical analysis, that was closely related to the essential differential operators. The experiment results confirmed that the proposed model leveraged the spatial structures and can deal with large-scale graphs. The computation time of the IsoGCN model is significantly shorter than the FEA, which other equivariant models cannot achieve. Therefore, IsoGCN must be the first choice to learn physical simulations because of its computational efficiency as well as $\mathrm{E}(n)$- invariance and equivariance. Our demonstrations were conducted on the mesh structured dataset based on the FEA results. However, we expect IsoGCNs to be applied to various domains, such as object detection, molecular property prediction, and physical simulations using particles.

Chapter 4

Physics-Embedded Neural Network: Boundary Condition and Implicit Method

4.1 Introduction

In Chapter 3, we introduced IsoGCN, a lightweight $\mathrm{E}(n)$-equivariant graph neural network. It can:

- handle an arbitrary mesh thanks to the generalizability of GNN;
- reflect symmetries regarding $\mathrm{E}(n)$ transformation that exists in physical phenomena; and
- predict faster than conventional numerical analysis methods and complex GNNs based on linear message passing scheme.

However, we still miss the following keys to constructing general PDE solvers:

- **Treatment of mixed boundary conditions**: Mixed boundary condition contains Dirichlet and Neumann boundary conditions in disjoint boundary regions, as expressed in Equations 2.54 and 2.55. The IsoGCN model demonstrated in Section 3.4.2 considers only adiabatic boundary conditions corresponding to the ho-

mogeneous Neumann boundary condition (Equation 3.73). Therefore, we must provide a provable way to handle mixed boundary conditions.

- **An implicit manner for time evolution**: In Section 3.4.2, we constructed IsoGCN models based on the explicit Euler method (Equation 3.80). It can consider interactions between vertices k hop away by stacking k IsoGCN layers. However, global interaction may sometimes occur as in incompressible flow phenomena, where the speed of sound is regarded as infinity. These global interactions require IsoGCN layers stacked more than the number of vertices $|\mathcal{V}|$, which may result in huge computation time. Therefore, we must incorporate implicit time evolution that can consider global interaction.
- **Demonstration in various PDEs**: We demonstrated IsoGCN's expressibility using the heat equation in Section 3.4.2. However, there are many PDEs in addition to the heat equation. Thus, we must show the model can learn various phenomena, such as the advection-diffusion and incompressible flow problems.

Thus, we introduce *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 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 numerical 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 4.1 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.

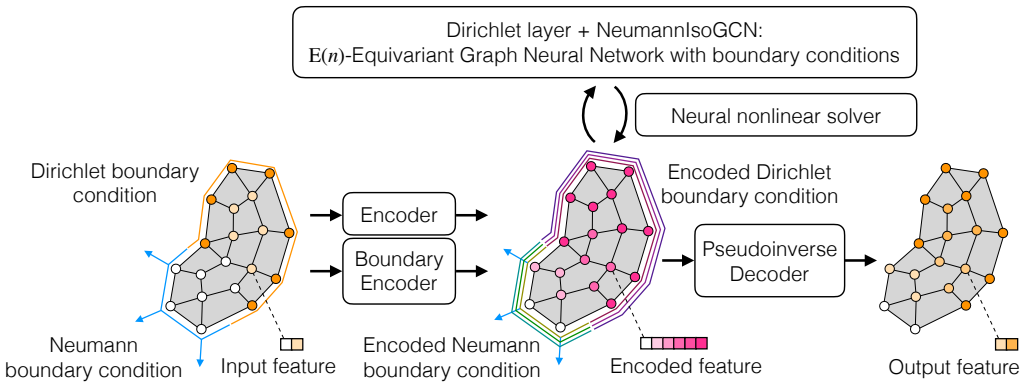


Figure 4.1: Overview of the proposed method. On decoding input features, we apply boundary encoders to boundary conditions. Thereafter, we apply a nonlinear solver consisting of an $\mathrm{E}(n)$-equivariant graph neural network in the encoded space. Here, we apply encoded boundary conditions for each iteration of the nonlinear solver. After the solver stops, we apply the pseudoinverse decoder to satisfy Dirichlet boundary conditions.

4.2 Related Prior Work

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

4.2.1 Physics-Informed Neural Network (PINN)

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 4.3, our model contains PDE information inside and does not take absolute positions of vertices, thus resulting in high generalizability (See Figure 4.16).

4.2.2 Graph Neural Network Based PDE Solver

As discussed in Sections 2.2.2, 2.2.4, and 3.2.3, one can regard a mesh as a graph and various existing studies demonstrated that GNNs can learn physical phenomena, as seen in Alet et al. (2019); Chang & Cheng (2020); Pfaff et al. (2021). Then, Brandstetter et al. (2022) advanced these works by suggesting temporal bundling and pushforward trick 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.

4.3 Method

We present our model architecture. Following the study done in Section 3.4, we adopt the encode-process-decode architecture, proposed by Battaglia et al. (2018), which has been applied successfully in various previous works, e.g., Pfaff 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. In this section, we continue to use the discrete tensor field (Equation 3.5) expressed as:

$$\boldsymbol{H} = \begin{pmatrix} \boldsymbol{h}_1 \\ \boldsymbol{h}_2 \\ \\ \vdots \\ \\ \boldsymbol{h}_{|\mathcal{V}|} \end{pmatrix}, \tag{4.1}$$

while we do not write the tensor rank explicitly unless needed.

4.3.1 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.

4.3.1.1 Boundary Encoder

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:

$$\boldsymbol{h}_i = \boldsymbol{f}_{\text{encode}}(\boldsymbol{u}_i) \qquad \text{in } \Omega \tag{4.2}$$

$$\hat{\boldsymbol{h}}_i = \boldsymbol{f}_{\text{encode}}(\hat{\boldsymbol{u}}_i) \qquad \text{on } \partial\Omega_{\text{Dirichlet}}, \tag{4.3}$$

where $\hat{\boldsymbol{u}}_i$ is the value of the Dirichlet boundary condition at $\boldsymbol{x}_i \in \partial\Omega_{\text{Dirichlet}}$.

4.3.1.2 DIRICHLET LAYER

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

$$\mathrm{DirichletLayer}(\boldsymbol{h}_i) = \begin{cases} \boldsymbol{h}_i, & \boldsymbol{x}_i \notin \partial\Omega_{\mathrm{Dirichlet}} \\ \hat{\boldsymbol{h}}_i, & \boldsymbol{x}_i \in \partial\Omega_{\mathrm{Dirichlet}}. \end{cases} \tag{4.4}$$

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

4.3.1.3 PSEUDOINVERSE DECODER

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

$$\boldsymbol{f}_{\mathrm{decode}} \circ \boldsymbol{f}_{\mathrm{encode}}(\hat{\boldsymbol{u}}_i) = \hat{\boldsymbol{u}}_i \text{ on } \partial\Omega_{\mathrm{Dirichlet}}. \tag{4.5}$$

This condition ensures that the encoded boundary conditions correspond to the ones in the original physical space. Demanding that Equation 4.5 holds for arbitrary $\hat{\boldsymbol{u}}$; we obtain:

$$\boldsymbol{f}_{\mathrm{decode}} \circ \boldsymbol{f}_{\mathrm{encode}} = \mathrm{Id}_{\boldsymbol{u}}, \tag{4.6}$$

where $\mathrm{Id}_{\boldsymbol{u}}$ denotes the identity map from the space of $\boldsymbol{u}$ to the same space. By applying $\boldsymbol{f}^{+}_{\mathrm{encode}}$, a left inverse function of the encoder, we have:

$$\boldsymbol{f}_{\mathrm{decode}} = \boldsymbol{f}^{+}_{\mathrm{encode}}, \tag{4.7}$$

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.

We can construct the pseudoinverse decoders for a wide range of neural network architectures. For instance, the pseudoinverse decoder for an MLP with one hidden layer

$$\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x}) = \sigma_2\left(\boldsymbol{W}_2\sigma_1(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2\right) \tag{4.8}$$

can be constructed as:

$$\boldsymbol{f}^{+}(\boldsymbol{h}) = \boldsymbol{W}_1^{+}\sigma_1^{-1}\left(\boldsymbol{W}_2^{+}\sigma_2^{-1}(\boldsymbol{h}) - \boldsymbol{b}_2\right) - \boldsymbol{b}_1, \tag{4.9}$$

where $\boldsymbol{W}^{+}$ is the pseudoinverse matrix of $\boldsymbol{W}$, satisfying $\boldsymbol{W}^{+}\boldsymbol{W} = \boldsymbol{I}$, and σ is an invertible activation function whose $\mathrm{Dom}(\sigma) = \mathrm{Im}(\sigma) = \mathbb{R}$. We can confirm that $\boldsymbol{f}^{+}$ is in fact the pseudoinverse of $\boldsymbol{f}$ as:

$$\begin{aligned}
\boldsymbol{f}^{+} \circ \boldsymbol{f}(\boldsymbol{x}) &= \boldsymbol{W}_1^{+}\sigma_1^{-1}\left(\boldsymbol{W}_2^{+}\sigma_2^{-1}\left(\sigma_2\left(\boldsymbol{W}_2\sigma_1(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2\right)\right) - \boldsymbol{b}_2\right) - \boldsymbol{b}_1 \\
&= \boldsymbol{W}_1^{+}\sigma_1^{-1}\left(\boldsymbol{W}_2^{+}\boldsymbol{W}_2\sigma_1(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 - \boldsymbol{b}_2\right) - \boldsymbol{b}_1 \\
&= \boldsymbol{W}_1^{+}\sigma_1^{-1}\left(\sigma_1(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)\right) - \boldsymbol{b}_1 \\
&= \boldsymbol{W}_1^{+}\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1 - \boldsymbol{b}_1 \\
&= \boldsymbol{x}.
\end{aligned} \tag{4.10}$$

For the activation function, we may choose LeakyReLU

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & (x \geq 0) \\ ax & (x < 0), \end{cases} \tag{4.11}$$

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.

4.3.2 Neumann Boundary Model

Matsunaga et al. (2020) proposed a wall boundary model to deal with Neumann boundary conditions for the LSMPS method (Tamai & Koshizuka, 2014) (Section 2.2.4.3), 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.

4.3.2.1 DEFINITION OF NEUMANNISOGCN (NISOGCN)

Our formulation of IsoGCN with Neumann boundary conditions, which is termed *NeumannIsoGCN* (NIsoGCN), is expressed as:

$$\begin{aligned}&\mathrm{NIsoGCN}_{0\to 1}(\boldsymbol{H}^{(0)})\\&\quad := \mathrm{EquivariantPointwiseMLP}\left(\boldsymbol{M}_i{}^{-1}\left[\sum_{j\in\mathcal{N}_i}\frac{\boldsymbol{h}_j^{(0)}-\boldsymbol{h}_j^{(0)}}{\|\boldsymbol{x}_j-\boldsymbol{x}_i\|}\frac{\boldsymbol{x}_j-\boldsymbol{x}_i}{\|\boldsymbol{x}_j-\boldsymbol{x}_i\|}w_{ij}+w_i\boldsymbol{n}_i\hat{\boldsymbol{g}}_i\right]\right)\end{aligned} \tag{4.12}$$

$$\boldsymbol{M}_i := \sum_{l\in\mathcal{N}_i}\frac{\boldsymbol{x}_l-\boldsymbol{x}_i}{\|\boldsymbol{x}_l-\boldsymbol{x}_i\|}\otimes\frac{\boldsymbol{x}_l-\boldsymbol{x}_i}{\|\boldsymbol{x}_l-\boldsymbol{x}_i\|}w_{ij}+w_i\boldsymbol{n}_i\otimes\boldsymbol{n}_i, \tag{4.13}$$

where $\hat{\boldsymbol{g}}_i$ is the encoded value of the Neumann boundary condition at $\boldsymbol{x}_i$ 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. Therefore, NIsoGCN can express any spatial differential operator, constituting $\mathcal{D}$ in PDEs.

4.3.2.2 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 orthonormal basis $\left\{\boldsymbol{e}_j \in \mathbb{R}^d \middle| \boldsymbol{e}_j \cdot \boldsymbol{e}_k = \delta_{jk}\right\}_{j=1}^n$, one can decompose a vector $\boldsymbol{v} \in \mathbb{R}^n$ using:

$$\boldsymbol{v} = \sum_j (\boldsymbol{v} \cdot \boldsymbol{e}_j)\boldsymbol{e}_j. \tag{4.14}$$

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

$$\boldsymbol{v} = \sum_j (\boldsymbol{v} \cdot \boldsymbol{b}_j)\boldsymbol{A}\boldsymbol{b}_j, \tag{4.15}$$

where $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ 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 $\{\boldsymbol{A}\boldsymbol{b}_j\}_{j=0}^{d'}$ is called a *dual frame* for $\boldsymbol{B}$. Recalling Equation 2.122, we can find the concrete form of $\boldsymbol{A}$ considering:

$$\begin{aligned}\boldsymbol{v} &= \boldsymbol{A}\sum_j (\boldsymbol{v} \cdot \boldsymbol{b}_j)\boldsymbol{b}_j \\ &= \boldsymbol{A}\sum_j (\boldsymbol{b}_j \otimes \boldsymbol{b}_j)\boldsymbol{v}.\end{aligned} \tag{4.16}$$

Requiring that Equation 4.16 holds for any $\boldsymbol{v} \in \mathbb{R}^d$, one can conclude $\boldsymbol{A} = \sum_j (\boldsymbol{b}_j \otimes \boldsymbol{b}_j)^{-1}$. Then, we obtain

$$\boldsymbol{v} = \left[\sum_l \boldsymbol{b}_l \otimes \boldsymbol{b}_l\right]^{-1} \sum_j (\boldsymbol{v} \cdot \boldsymbol{b}_j)\boldsymbol{b}_j. \tag{4.17}$$

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

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

$$\boldsymbol{B} = \left\{\sqrt{w_{ij}}\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|}\right\}_{j \in \mathcal{N}_i} \cup \{\sqrt{w_i}\boldsymbol{n}_i\}. \tag{4.18}$$

In addition, we assume the approximated gradient of a scalar field u at the ith vertex, $\langle \nabla u \rangle_i$, satisfies the following conditions:

$$\langle \nabla u \rangle_i \cdot \sqrt{w_{ij}} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} = \sqrt{w_{ij}} \frac{u_j - u_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \qquad (j \in \mathcal{N}_i) \tag{4.19}$$

$$\langle \nabla u \rangle_i \cdot \sqrt{w_i} \boldsymbol{n}_i = \sqrt{w_i} \hat{g}_i. \tag{4.20}$$

Equation 4.19 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 u in the same direction. Equation 4.20 is the Neumann boundary condition, which we want to satisfy. Finally, by substituting Equations 4.18, 4.19, and 4.20 into Equation 4.17, we obtain the gradient model considering the Neumann boundry consition as:

$$\begin{aligned}
\langle u \rangle_i =& \left[\sum_{l \in \mathcal{N}_i} \sqrt{w_{il}} \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \sqrt{w_{il}} \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \right]^{-1} \\
& \times \left[\sum_j \left(\sqrt{w_{ij}} \frac{u_j - u_i}{\boldsymbol{x}_j - \boldsymbol{x}_i} \sqrt{w_{ij}} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \right) + \sqrt{w_i} \hat{g}_i \sqrt{w_i} \boldsymbol{n}_i \right] \\
=& \left[\sum_{l \in \mathcal{N}_i} w_{il} \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \otimes \frac{\boldsymbol{x}_l - \boldsymbol{x}_i}{\|\boldsymbol{x}_l - \boldsymbol{x}_i\|} \right]^{-1} \\
& \times \left[\sum_j \left(w_{ij} \frac{u_j - u_i}{\boldsymbol{x}_j - \boldsymbol{x}_i} \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|} \right) + w_i \hat{g}_i \boldsymbol{n}_i \right].
\end{aligned} \tag{4.21}$$

If we apply the gradient model to a encoded features, we obtain the gradient model in the NIsoGCN layer, i.e., Equation 4.12. 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.

4.3.2.3 GENERALIZATION OF NISOGCN

To apply NIsoGCN to $\boldsymbol{H}^{(p)}$, a rank p discrete tensor field ($p \geq 1$), one can recursively define the operation as:

$$\mathrm{NIsoGCN}_{p\to p+1}(\boldsymbol{H}^{(p)}) := \begin{pmatrix} \mathrm{NIsoGCN}_{p-1\to p}(\boldsymbol{H}^{(p)}_{:,:,:,1\ldots}) \\ \mathrm{NIsoGCN}_{p-1\to p}(\boldsymbol{H}^{(p)}_{:,:,:,2\ldots}) \\ \mathrm{NIsoGCN}_{p-1\to p}(\boldsymbol{H}^{(p)}_{:,:,:,3\ldots}) \end{pmatrix}, \tag{4.22}$$

where $\boldsymbol{H}^{(p)}_{:,:,:,i\ldots} \in \mathbb{R}^{|\mathcal{V}|\times d_{\mathrm{feature}}\times n^{p-1}}$ is the ith component of $\boldsymbol{H}^{(p)}$ regarding the first spatial index, resulting in the rank $(p-1)$ discrete tensor field. In case of a three-dimensional rank one discrete tensor field $\boldsymbol{H}^{(1)}$, it can be formulated as:

$$\begin{aligned} \mathrm{NIsoGCN}_{1\to 2}(\boldsymbol{H}^{(1)}) &:= \begin{pmatrix} \mathrm{NIsoGCN}_{0\to 1}(\boldsymbol{H}^{(1)}_{:,:,:,1}) \\ \mathrm{NIsoGCN}_{0\to 1}(\boldsymbol{H}^{(1)}_{:,:,:,2}) \\ \mathrm{NIsoGCN}_{0\to 1}(\boldsymbol{H}^{(1)}_{:,:,:,3}) \end{pmatrix} \\ &\approx \begin{pmatrix} \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,1}/\partial x \right\rangle & \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,1}/\partial y \right\rangle & \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,1}/\partial z \right\rangle \\ \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,2}/\partial x \right\rangle & \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,2}/\partial y \right\rangle & \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,2}/\partial z \right\rangle \\ \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,3}/\partial x \right\rangle & \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,3}/\partial y \right\rangle & \left\langle \partial \boldsymbol{H}^{(1)}_{:,:,:,3}/\partial z \right\rangle \end{pmatrix} \qquad (4.23) \\ &= \left\langle \nabla \otimes \boldsymbol{H}^{(1)} \right\rangle, \qquad (4.24) \end{aligned}$$

which corresponds to the Jacobian tensor field of $\boldsymbol{H}^{(1)}$. Similarly, NIsoGCN to decrease tensor rank can be defined as:

$$\begin{aligned} \mathrm{NIsoGCN}_{p\to p-1}(\boldsymbol{H}^{(p)}) :=& \mathrm{NIsoGCN}_{p-1\to p}(\boldsymbol{H}^{(p)}_{:,:,:,1\ldots}) \\ &+ \mathrm{NIsoGCN}_{p-1\to p}(\boldsymbol{H}^{(p)}_{:,:,:,2\ldots}) \\ &+ \mathrm{NIsoGCN}_{p-1\to p}(\boldsymbol{H}^{(p)}_{:,:,:,3\ldots}). \end{aligned} \tag{4.25}$$

As discussed in Section 3.3.4.1, IsoGCNs (NIsoGCNs) correspond to spatial differential operators. Because NIsoGCN contains a learnable neural network (Equation 4.12), the component learns to predict the derivative of the corresponding tensor rank in an en-

coded space. This feature of NIsoGCNs enables us to construct machine learning models corresponding to PDE in the encoded space.

4.3.3 Neural Nonlinear Solver

4.3.3.1 Implicit Euler Method in Encoded Space

As reviewed in Section 2.2.1, one can regard solving PDEs as optimization. To construct a neural PDE solver using the implicit Euler method in a high-dimensional encoded space, we first define the residual and the nonlinear problem in the encoded space based on Equation 2.63 as:

$$\boldsymbol{R}_{\mathrm{NIsoGCN}}(\boldsymbol{H}') := \boldsymbol{H}' - \boldsymbol{H}(t) - \mathcal{D}_{\mathrm{NIsoGCN}}(\boldsymbol{H}')\Delta t \tag{4.26}$$

$$\mathrm{Solve}'_{\boldsymbol{H}} \boldsymbol{R}_{\mathrm{NIsoGCN}}(\boldsymbol{H}') = \boldsymbol{0}, \tag{4.27}$$

where $\boldsymbol{H}(t)$ and $\boldsymbol{H}'$ are discrete tensor fields, and $\mathcal{D}_{\mathrm{NIsoGCN}}$ is an $\mathrm{E}(n)$-equivariant GNN reflecting the structure of $\mathcal{D}$ using differential operators provided by NIsoGCN (See Section 4.4 for the concrete examples of $\mathcal{D}_{\mathrm{NIsoGCN}}$). Equation 4.27 corresponds to solving a PDE in a high-dimensional encoded space, where we can utilize the expressibility of neural networks.

One may consider solving Equation 4.27 by using the Newton–Raphson method. However, it may consume huge memory because we embed the input feature into a high-dimensional space, resulting in a large matrix to solve. In addition, we must use GPUs to accelerate the training of models, which makes memory limitation more strict. Furthermore, solving linear systems makes the computation graph extremely long, leading to unstable backpropagation. There are various existing studies to challenge this type of problem, e.g., Neural ODE (Chen et al., 2018) and implicit GNN (Gu et al., 2020). Besides, adopting limited-memory quasi-Newton methods (e.g., Liu & Nocedal (1989)) might be interesting as they are supposed to facilitate incorporating global interactions. Nevertheless, we applied the gradient descent method in this research for simplicity and computational efficiency. Based on Equation 2.70, gradient descent in the encoded space can be expressed

as:

$$\boldsymbol{H}^{[0]} := \boldsymbol{H}(t) \tag{4.28}$$

$$\boldsymbol{H}^{[i+1]} := \boldsymbol{H}^{[i]} - \alpha^{[i]} \boldsymbol{R}_{\mathrm{NIsoGCN}}(\boldsymbol{H}^{[i]}) \ \ i > 0, \tag{4.29}$$

where $\boldsymbol{H}^{[i]}$ denotes the approximated solution at ith step of the iterative nonlinear solver (as in Section 2.2.3), not a rank-i discrete tensor field.

4.3.3.2 Barzilai–Borwein Method for Neural Nonlinear Solver

As discussed in Section 2.2.3.3, $\alpha^{[i]}$ are determined by the line search, requiring additional computational resource. However, using a small constant value of α results in the explicit Euler method, which corresponds to simply stacking the GNN layers. Therefore, we adopt the Barzilai–Borwein method (Barzilai & Borwein, 1988) to approximate $\alpha^{[i]}$ in Equation 4.29. In our case, by applying Equation 2.81, the step size $\alpha^{[i]}$ of gradient descent is approximated as:

$$\alpha^{[i]} \approx \alpha^{[i]}_{\mathrm{BB}} := \frac{\left[\boldsymbol{H}^{[i]} - \boldsymbol{H}^{[i-1]}\right] \cdot \left[\boldsymbol{R}_{\mathrm{NIsoGCN}}(\boldsymbol{H}^{[i]}) - \boldsymbol{R}_{\mathrm{NIsoGCN}}(\boldsymbol{H}^{[i-1]})\right]}{\left\| \boldsymbol{R}_{\mathrm{NIsoGCN}}(\boldsymbol{H}^{[i]}) - \boldsymbol{R}_{\mathrm{NIsoGCN}}(\boldsymbol{H}^{[i-1]}) \right\|^2}. \tag{4.30}$$

Here, $\cdot$ denotes the inner product between two discrete tensor fields with the same shape, i.e.:

$$\boldsymbol{H}^{(p)} \cdot \boldsymbol{G}^{(p)} = \sum_{igk_1k_2\ldots k_p} H_{i;g;k_1k_2\ldots k_p} G_{i;g;k_1k_2\ldots k_p} \in \mathbb{R}, \tag{4.31}$$

for rank-p discrete tensor fields $\boldsymbol{H}^{(p)}$ and $\boldsymbol{G}^{(p)}$. Besides, $\|\boldsymbol{H}\|^2 := \boldsymbol{H} \cdot \boldsymbol{H}$. The inner product used here corresponds to that for rank-p continuous tensor fields, $\langle \boldsymbol{h}, \boldsymbol{g} \rangle$ (where $\boldsymbol{h}, \boldsymbol{g} : \Omega \to \mathbb{R}^{d_{\mathrm{feature}} \times n^p}$), because:

$$\langle \boldsymbol{h}, \boldsymbol{g} \rangle = \int_\Omega \boldsymbol{h}(\boldsymbol{x}) \cdot \boldsymbol{g}(\boldsymbol{x}) d\Omega(\boldsymbol{x}) \tag{4.32}$$

$$\approx \sum_i \boldsymbol{h}(\boldsymbol{x}_i) \cdot \boldsymbol{g}(\boldsymbol{x}_i) V_i \tag{4.33}$$

$$= \sum_{igk_1k_2\ldots k_p} H_{i;g;k_1k_2\ldots k_p} G_{i;g;k_1k_2\ldots k_p} V_i^{\mathrm{effective}}, \tag{4.34}$$

wherer $V_i^{\text{effective}}$ denotes the effective volume of the ith vertex (Equation 3.74). One must note that α_{BB} defined here is $\mathrm{E}(n)$-invariant because it is computed using the contraction between tensors. Therefore, the gradient descent update:

$$\boldsymbol{H}^{[i+1]} := \boldsymbol{H}^{[i]} - \alpha_{\text{BB}}^{[i]} \boldsymbol{R}_{\text{NIsoGCN}}(\boldsymbol{H}^{[i]}) \tag{4.35}$$

is $\mathrm{E}(n)$-equivariant.

In addition, one can see that computing $\alpha_{\text{BB}}^{[i]}$ corresponds to global pooling because the inner product is taken all over the mesh (graph). With that view, one can find similarities between Equation 4.35 and deep sets (Zaheer et al., 2017). A deep set layer is expressed as:

$$\text{DeepSet}(\boldsymbol{H}) := \sigma\left(\lambda \boldsymbol{H} + \gamma \mathbf{1}_{|\mathcal{V}|} \text{GlobalPooling}(\boldsymbol{H})\right), \tag{4.36}$$

where λ and γ are trainable parameters, $\text{GlobalPooling} : \mathbb{R}^{|\mathcal{V}| \times d_{\text{feature}} \times n^p} \to \mathbb{R}^{1 \times d_{\text{feature}} \times n^p}$ denotes an operation that aggregates all information in a graph, such as max, mean, and sum, and $\mathbf{1} = (1, 1, \ldots, 1)^T \in \mathbb{R}^{|\mathcal{V}| \times 1}$. Deep set is a successful method to learn point cloud data and has a strong background regarding permutation equivariance. However, $\mathrm{E}(n)$-equivariance is not considered in their model. Our gradient descent update (Equation 4.35) successfully incorporate the strength of the deep set model with $\mathrm{E}(n)$-equivariance and interpretability in terms of the implicit Euler method.

4.3.3.3 Formulation of Neural Nonlinear Solver

Our aim is to use Equation 4.35, approximating the nonlinear differential operator $\mathcal{D}$ in Equation 2.60 with NIsoGCN. By doing this, we expect the processor, the core of the encode-decode-processor Architecture, 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 2.2.2. 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 in an $\mathrm{E}(n)$-equivariant way, 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:

$$\boldsymbol{H}^{[i+1]} = \mathrm{DirichletLayer}\left(\boldsymbol{H}^{[i]} - \alpha_{\mathrm{BB}}^{[i]}\left[\boldsymbol{H}^{[i]} - \boldsymbol{H}^{[0]} - \mathcal{D}_{\mathrm{NIsoGCN}}(\boldsymbol{H}^{[i]})\Delta t\right]\right), \quad (4.37)$$

where $\boldsymbol{H}^{[0]}$ is the encoded $\boldsymbol{U}(t)$. Here, Equation 4.37 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.

4.4 Numerical Experiments

Using numerical experiments, we demonstrate the proposed model's validity, expressibility, and computational efficiency. We use three types of datasets:

1. the gradient dataset to verify the correctness of NIsoGCN; and
2. the advection-diffusion dataset to demonstrate capacity of the model for various PDE parameters; and
3. the incompressible flow dataset to demonstrate the speed and accuracy of the model.

We also present ablation study results to corroborate the effectiveness of the proposed method. The implementation of our model is based on the original IsoGCN's code.[1] Our implementation is available online.[2]

4.4.1 Gradient Dataset

As done in Section 3.4.1, we conducted experiments to predict the gradient field from a given scalar field to verify the expressive power of NIsoGCN.

4.4.1.1 Taks Definition

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

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

degree 10 and computed their gradient fields analytically. Our training, validation, and test datasets consisted of 100 samples.

4.4.1.2 Model Architecture

Figure 4.2 shows the architectures we used for the gradient dataset. The dataset is uploaded online.[3] 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 4.13, 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 4.2 shows model architectures used for the experiment.

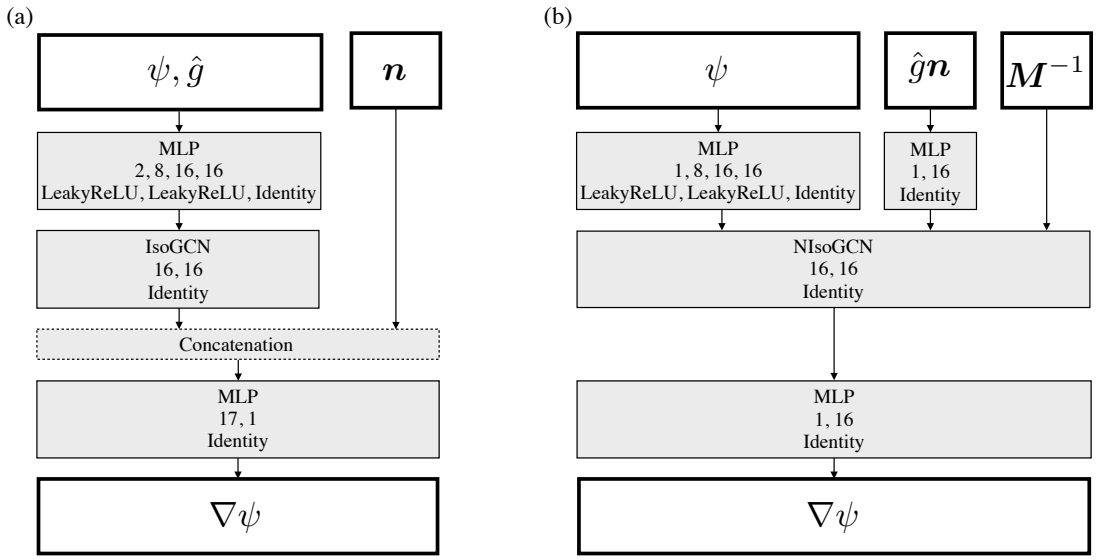


Figure 4.2: Architecture used for (a) original IsoGCN and (b) NIsoGCN training. In each trainable cell, we put the number of units in each layer along with the activation functions used.

[3] https://savanna.ritc.jp/~horiem/penn_neurips2022/data/grad/grad_data.tar.gz

4.4.1.3 RESULTS

Table 4.1 and Figure 4.3 show that the proposed NIsoGCN improves gradient prediction, especially near the boundary, showing that our model successfully considers Neumann boundary conditions.

Table 4.1: MSE loss ($\pm$ the standard error of the mean) on test dataset of gradient prediction. $\hat{g}_{\text{Neumann}}$ is the loss computed only on the boundary where the Neuman condition is set.

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

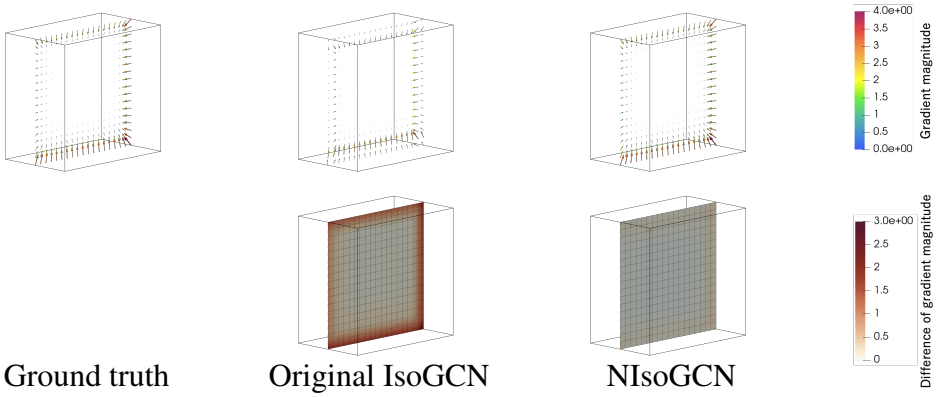


Figure 4.3: Gradient field (top) and the magnitude of error between the predicted gradient and the ground truth (bottom) of a test data sample, sliced on the center of the mesh.

4.4.2 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.

4.4.2.1 TASK DEFINITION

The governing equation regarding the temperature field T used for the experiment is expressed as:

$$\frac{\partial T}{\partial t} = -c \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \nabla T + D \nabla \cdot \nabla T \qquad (t, \boldsymbol{x}) \in (0, 1) \times \Omega \tag{4.38}$$

$$T(t = 0, \boldsymbol{x}) = 0 \qquad \boldsymbol{x} \in \Omega \tag{4.39}$$

$$T = \hat{T} \qquad (t, \boldsymbol{x}) \in \partial\Omega_{\text{Dirichlet}} \tag{4.40}$$

$$\nabla T \cdot \boldsymbol{n} = 0 \qquad (t, \boldsymbol{x}) \in \partial\Omega_{\text{Neumann}}, \tag{4.41}$$

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 $\partial\Omega_{\text{Neumann}} = \partial\Omega \setminus \partial\Omega_{\text{Dirichlet}}$.

4.4.2.2 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 with $\Delta t = 10^{-3}$ We generated fine meshes, ran numerical analysis with a classical solver, OpenFOAM,[4] and interpolated the obtained temperature fields onto coarser meshes so that we can obtain high-quality ground truth data. We split the generated data into training, validation, and test dataset containing 960, 120, and 120 samples. The dataset is uploaded online.[5]

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

[5] `https://savanna.ritc.jp/~horiem/penn_neurips2022/data/ad/ad_preprocessed.tar.gz`

4.4.2.3 Model Architecture

The strategy to construct PENN for the advection-diffusion dataset is consistent with one for the incompressible flow dataset (see Section 4.4.3.3). 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:

$$\boldsymbol{H}_T(t + \Delta t) = \boldsymbol{H}_T(t) + \mathcal{D}_{\text{NIsoGCN;A-D}}\left(\boldsymbol{H}_T\right)(t + \Delta t) \tag{4.42}$$

$$\mathcal{D}_{\text{NIsoGCN;A-D}}\left(\boldsymbol{H}_T\right) := -\boldsymbol{H}_{\boldsymbol{c}} \cdot \text{NIsoGCN}_{0\to1}(\boldsymbol{H}_T) + \boldsymbol{H}_D\, \text{NIsoGCN}_{0\to1\to0}(\boldsymbol{H}_T), \tag{4.43}$$

where encoded discrete tensor fields corresponds to the following:

- $\boldsymbol{H}_T$: Encoded rank-0 discrete tensor field of T
- $\boldsymbol{H}_D$: Encoded rank-0 discrete tensor field of c, D, $e^{-0.5d}$, $e^{-1.0d}$, and $e^{-2.0d}$

- $\boldsymbol{H}_{\boldsymbol{c}}$: Encoded rank-1 discrete tensor field of $\boldsymbol{c}$

The corresponding neural nonlinear solver is:

$$\boldsymbol{H}_T^{[i+1]} = \boldsymbol{H}_T^{[i]} - \alpha_{\mathrm{BB}}^{[i]} \left[\boldsymbol{H}_T^{[i]} - \boldsymbol{H}_T^{[0]} - \mathcal{D}_{\mathrm{NIsoGCN;A\text{-}D}}(\boldsymbol{H}_T^{[i]}) \Delta t \right], \tag{4.44}$$

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 4.4). Figures 4.5 and 4.6 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 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.

(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.

The training is performed for up to ten hours using the Adam optimizer for each setting.

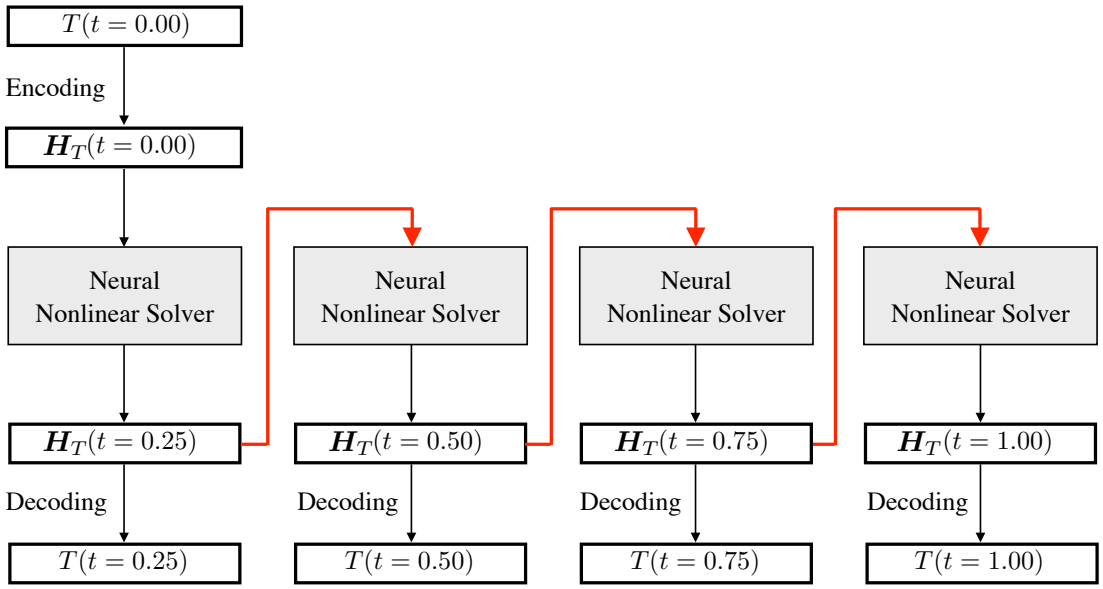


Figure 4.4: The concept of the neural nonlinear solver for time series data with autoregressive architecture. The solver's output is fed to the same solver to obtain the state at the next time step (bold red arrow). Please note that this architecture can be applied to arbitrary time series lengths.

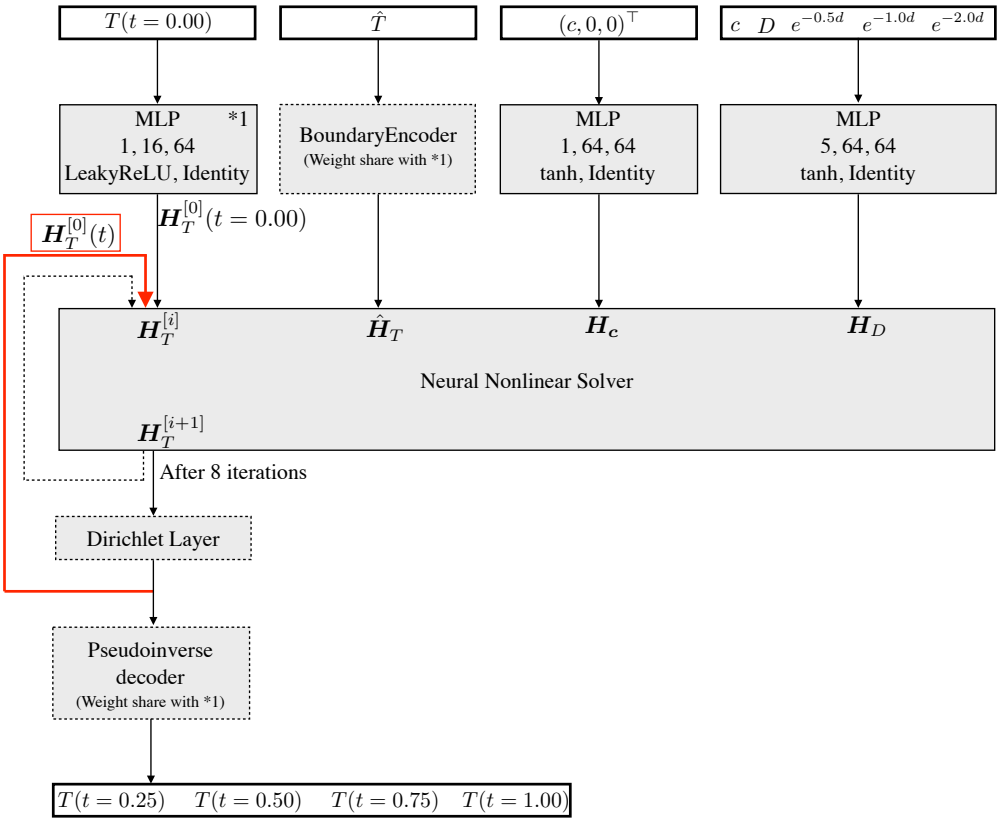


Figure 4.5: The overview of the PENN architecture for the advection-diffusion dataset. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. Arrows with dotted lines correspond to the loop. In each trainable cell, we put the number of units in each layer along with the activation functions used. The bold red arrow corresponds to the one in Figure 4.4.

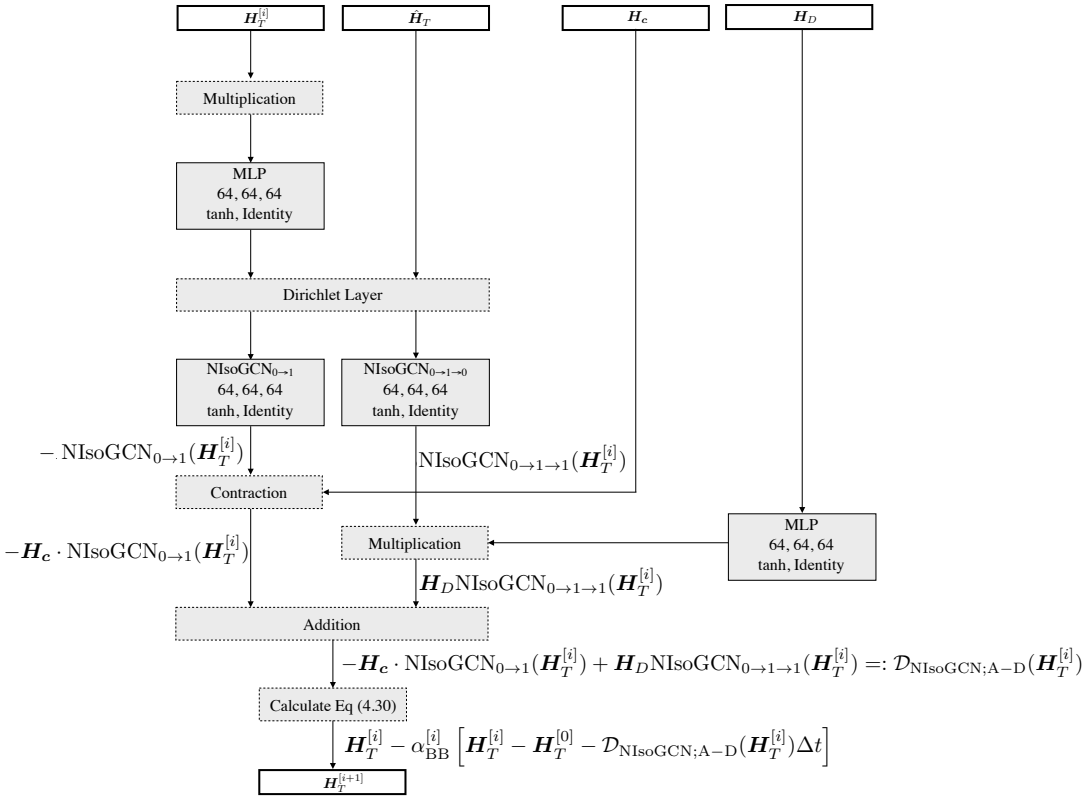


Figure 4.6: The overview of the PENN architecture for the advection-diffusion dataset. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. In each trainable cell, we put the number of units in each layer along with the activation functions used.

4.4.2.4 RESULTS

Table 4.2 presents the results of the ablation study. We found that the PENN model with all the proposed components achieved the best performance, showing that all the components we introduced contributed to performance. Because the boundary condition applied is relatively simple compared to the incompressible flow dataset (Section 4.4.3), 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 4.7, 4.8, and 4.9 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 4.7), pure diffusion (Figure 4.8), and mixed advection and diffusion (Figure 4.9).

Table 4.2: 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

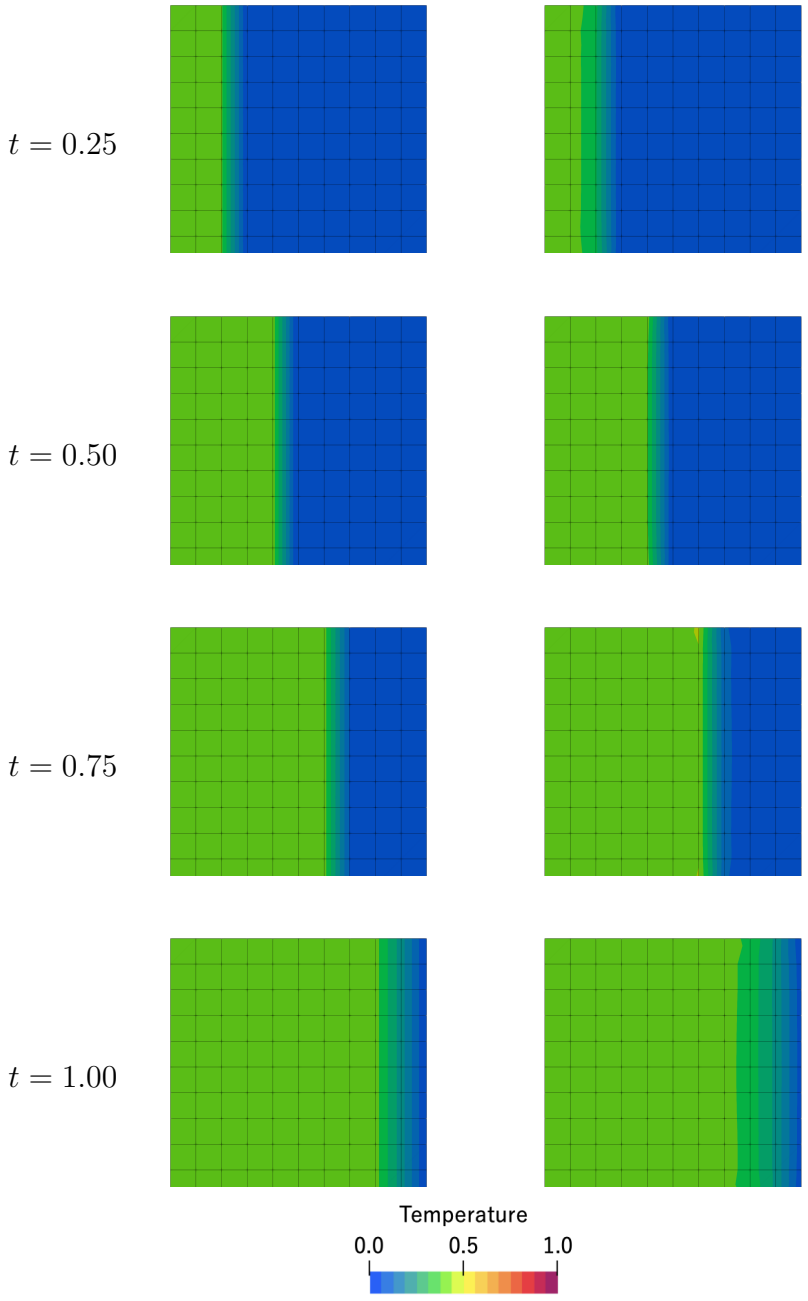


Figure 4.7: Visual comparison on a test sample between (left) ground truth obtained from OpenFOAM computation with fine spatial-temporal resolution and (right) prediction by PENN. Here, $c = 0.9$, $D = 0.0$, and $\hat{T} = 0.4$.

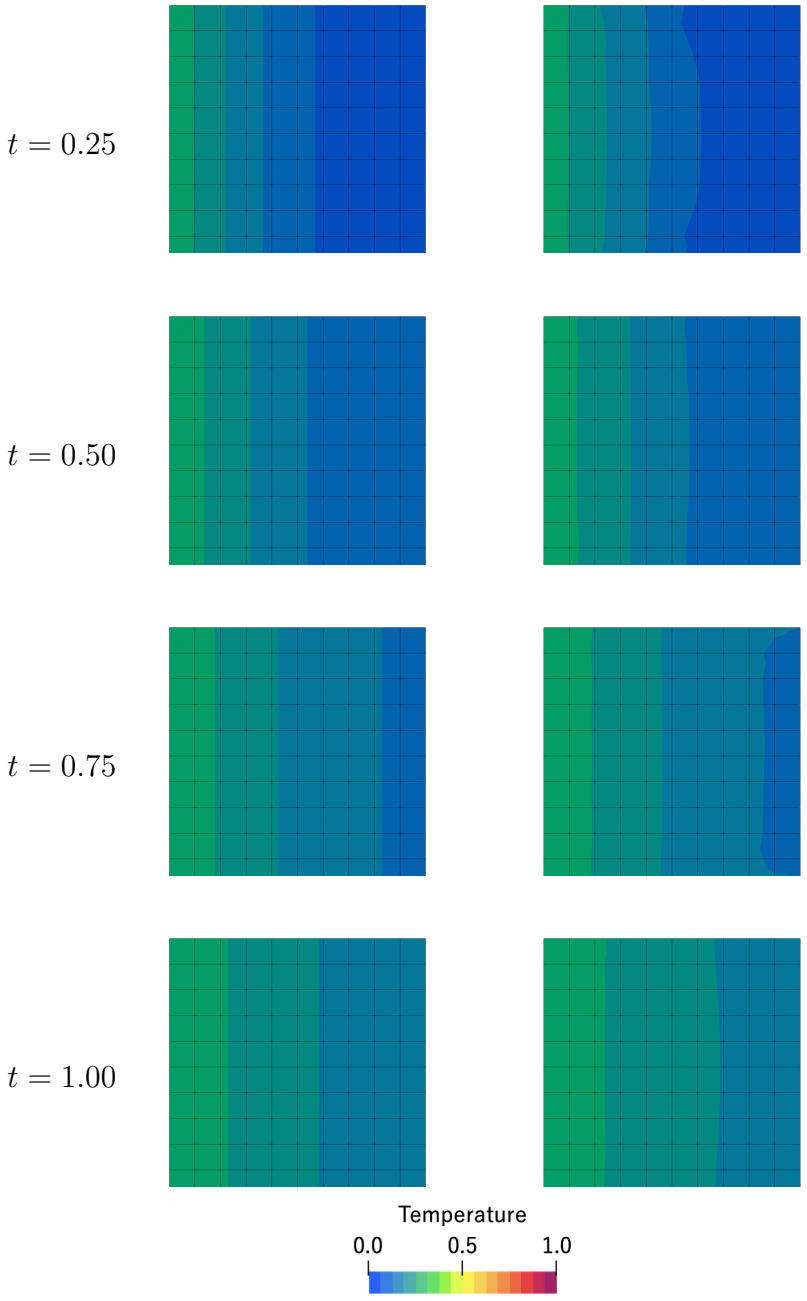


Figure 4.8: Visual comparison on a test sample between (left) ground truth obtained from OpenFOAM computation with fine spatial-temporal resolution and (right) prediction by PENN. Here, $c = 0.0$, $D = 0.4$, and $\hat{T} = 0.3$.

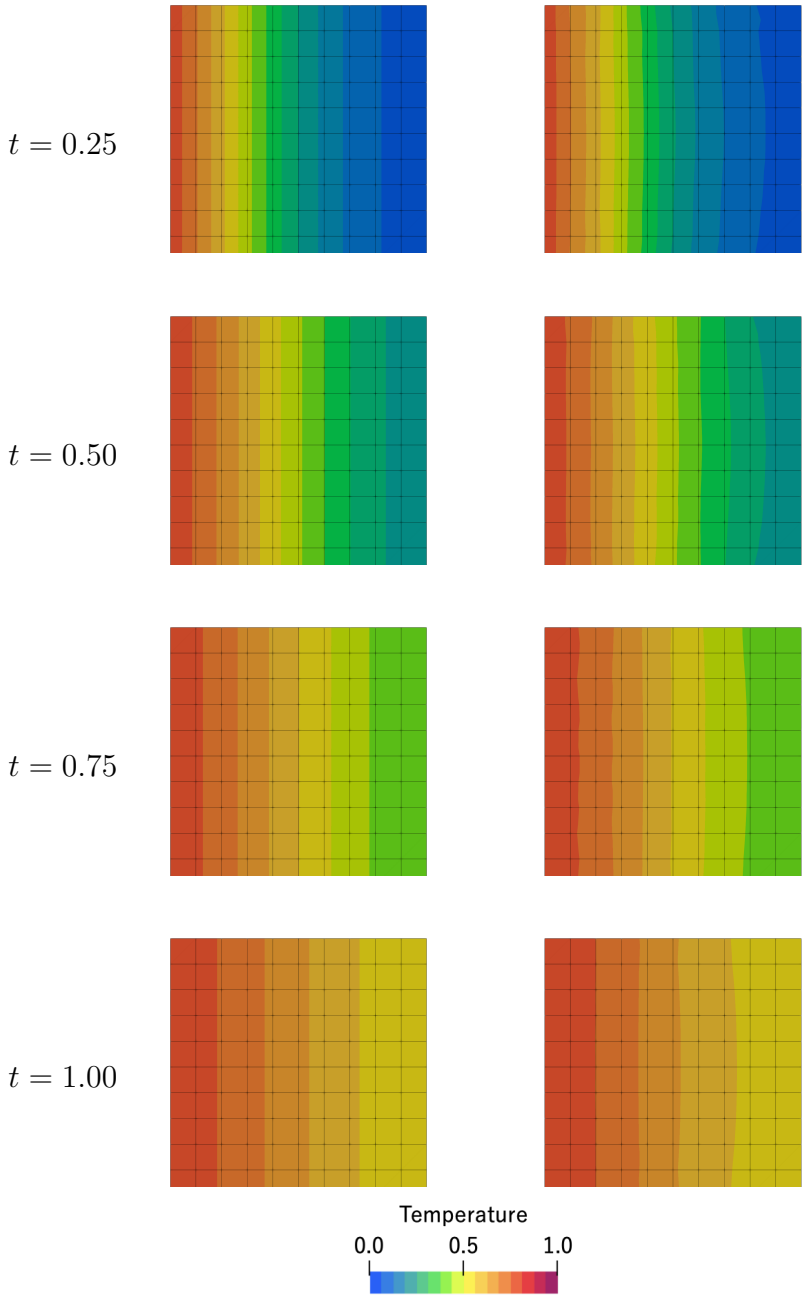


Figure 4.9: Visual comparison on a test sample between (left) ground truth obtained from OpenFOAM computation with fine spatial-temporal resolution and (right) prediction by PENN. Here, $c = 0.6$, $D = 0.3$, and $\hat{T} = 0.8$.

4.4.3 INCOMPRESSIBLE FLOW DATASET

We tested the expressive power our model by learning incompressible flow in complex shapes.

4.4.3.1 TASK DEFINITION

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

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \frac{1}{\mathrm{Re}}\nabla\cdot\nabla\boldsymbol{u} - \nabla p \qquad (t,\boldsymbol{x}) \in (0,T)\times\Omega \tag{4.45}$$

$$\boldsymbol{u} = \hat{\boldsymbol{u}} \qquad (t,\boldsymbol{x}) \in \partial\Omega^{(\boldsymbol{u})}_{\mathrm{Dirichlet}} \tag{4.46}$$

$$\left[\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T\right]\boldsymbol{n} = \boldsymbol{0} \qquad (t,\boldsymbol{x}) \in \partial\Omega^{(\boldsymbol{u})}_{\mathrm{Neumann}}. \tag{4.47}$$

We also consider the following incompressible condition:

$$\nabla\cdot\boldsymbol{u} = 0 \qquad (t,\boldsymbol{x}) \in (0,T)\times\Omega, \tag{4.48}$$

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:

$$\nabla\cdot\nabla p(t+\Delta t,\boldsymbol{x}) = \frac{1}{\Delta t}(\nabla\cdot\tilde{\boldsymbol{u}})(t,\boldsymbol{x}), \tag{4.49}$$

where

$$\tilde{\boldsymbol{u}} = \boldsymbol{u} - \Delta t\left(\boldsymbol{u}\cdot\nabla\boldsymbol{u} - \frac{1}{\mathrm{Re}}\nabla\cdot\nabla\boldsymbol{u}\right) \tag{4.50}$$

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

$$\boldsymbol{u}(t+\Delta t,\boldsymbol{x}) = \tilde{\boldsymbol{u}}(t,\boldsymbol{x}) - \Delta t\nabla p(t+\Delta t,\boldsymbol{x}). \tag{4.51}$$

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

$$p = 0 \qquad (t, \boldsymbol{x}) \in \partial\Omega^{(p)}_{\mathrm{Dirichlet}} \tag{4.52}$$

$$\nabla p \cdot \boldsymbol{n} = 0 \qquad (t, \boldsymbol{x}) \in \partial\Omega^{(p)}_{\mathrm{Neumann}}. \tag{4.53}$$

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

4.4.3.2 DATASET

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 OpenFOAM[6] 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 linear solvers used were generalized geometric-algebraic multi-grid for p and the smooth solver with the Gauss–Siedel smoother for $\boldsymbol{u}$.

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. 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.

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 4.10. 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 4.11 and 4.12. These design and boundary conditions were

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

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]

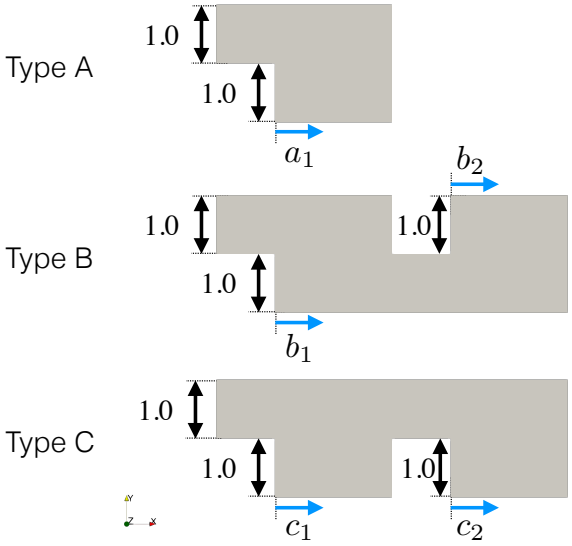


Figure 4.10: Three template shapes used to generate the dataset. a_1, b_1, b_2, c_1, and c_2 are the design parameters.

[7] https://savanna.ritc.jp/~horiem/penn_neurips2022/data/fluid/fluid_data.tar.gz.parta[a-e]

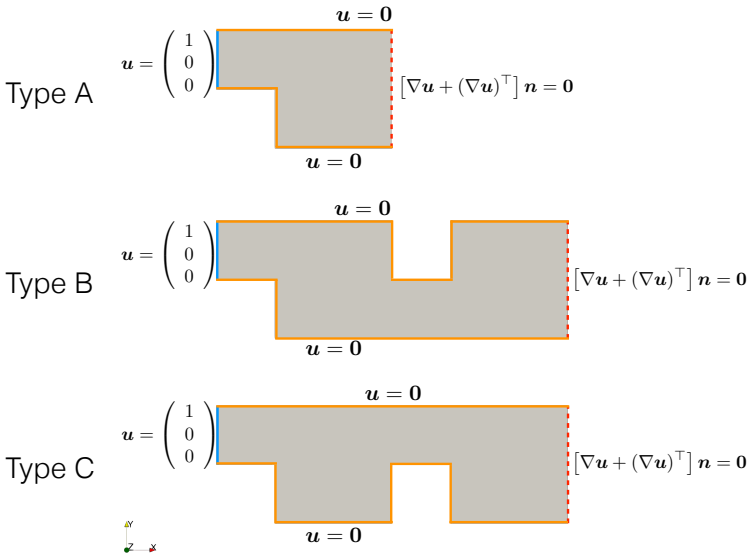


Figure 4.11: Boundary conditions of $\boldsymbol{u}$ used to generate the dataset. The continuous lines and dotted lines correspond to Dirichlet and Neumann boundaries.

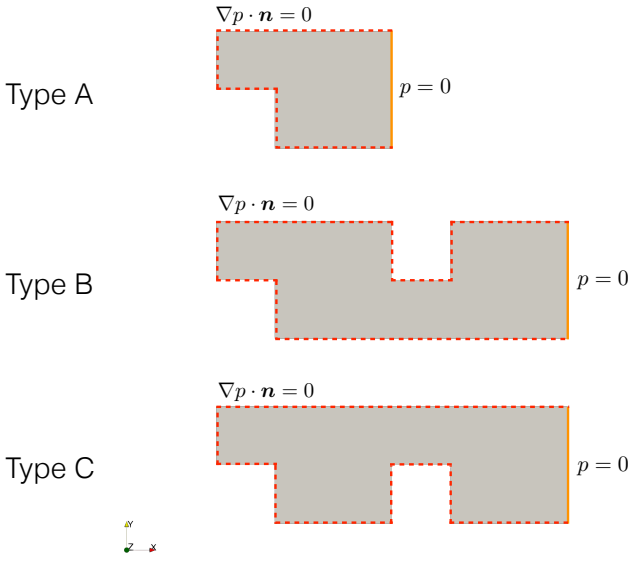


Figure 4.12: Boundary conditions of p used to generate the dataset. The continuous lines and dotted lines correspond to Dirichlet and Neumann boundaries.

4.4.3.3 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 NIsoGCNs and Dirichlet layers, reflecting the fractional step method (See Equations 4.50 and 4.51). 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 4.49). 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[8] 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).

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

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

MLP with one hidden layer. We encode each feature in a 16-dimensional space. After features are encoded, we apply a neural nonlinear solver containing NIsoGCNs and Dirichlet layers, reflecting the fractional step method (Equations 4.50 and 4.51).

The encoded equations are expressed as:

$$\begin{aligned}
&[\mathrm{NIsoGCN}_{1\to0} \circ \mathrm{NIsoGCN}_{0\to1}(\boldsymbol{H}_p)](t+\Delta t, \boldsymbol{x}) \\
&\qquad = \frac{1}{\Delta t}\left[\mathrm{NIsoGCN}_{1\to0}\left(\tilde{\boldsymbol{H}}_{\boldsymbol{u}}\right)\right](t, \boldsymbol{x}) \qquad (4.54)\\
&\tilde{\boldsymbol{H}}_{\boldsymbol{u}} := \boldsymbol{H}_{\boldsymbol{u}} - \Delta t \Bigg[\boldsymbol{H}_{\boldsymbol{u}} \cdot \mathrm{NIsoGCN}_{1\to2}(\boldsymbol{H}_{\boldsymbol{u}}) \\
&\qquad - \frac{1}{\mathrm{Re}}\mathrm{NIsoGCN}_{2\to1} \circ \mathrm{NIsoGCN}_{1\to2}(\boldsymbol{H}_{\boldsymbol{u}})\Bigg] \qquad (4.55)\\
&\boldsymbol{H}_{\boldsymbol{u}}(t+\Delta t, \boldsymbol{x}) = \tilde{\boldsymbol{H}}_{\boldsymbol{u}}(t, \boldsymbol{x}) - \Delta t\, \mathrm{NIsoGCN}_{0\to1}(\boldsymbol{H}_p)(t+\Delta t, \boldsymbol{x}), \qquad (4.56)
\end{aligned}$$

where $\boldsymbol{H}_{\boldsymbol{u}}$ is the encoded rank-1 discrete tensor field of $\boldsymbol{u}$ and $\boldsymbol{H}_p$ is the encoded rank-0 discrete tensor field of p. Note that these equations correspond to Equations 4.49, 4.50, and 4.51, by regarding IsoGCNs as spatial derivative operators. The corresponding neural nonlinear solvers are expressed as:

$$\begin{aligned}
\boldsymbol{H}_{\boldsymbol{u}}^{[i+1]} &= \boldsymbol{H}_{\boldsymbol{u}}^{[i]} - \alpha_{\mathrm{BB}}^{[i]}\left[\boldsymbol{H}_{\boldsymbol{u}}^{[i)]} - \boldsymbol{H}_{\boldsymbol{u}}^{(0)} - \mathcal{D}_{\mathrm{NIsoGCN;NS}}\left(\boldsymbol{H}_{\boldsymbol{u}}^{[i)]}, \boldsymbol{H}_p^{[i+1]}\right)\Delta t\right] \qquad (4.57)\\
\mathcal{D}_{\mathrm{NIsoGCN;NS}}\left(\boldsymbol{H}_{\boldsymbol{u}}^{[i]}, \boldsymbol{H}_p^{[i+1]}\right) &:= \Bigg[\boldsymbol{H}_{\boldsymbol{u}}^{[i]} \cdot \mathrm{NIsoGCN}_{1\to2}\left(\boldsymbol{H}_{\boldsymbol{u}}^{[i]}\right) \\
&\quad - \frac{1}{\mathrm{Re}}\mathrm{NIsoGCN}_{2\to1} \circ \mathrm{NIsoGCN}_{1\to2}\left(\boldsymbol{H}_{\boldsymbol{u}}^{[i]}\right) \\
&\quad + \mathrm{NIsoGCN}\left(\boldsymbol{H}_p^{[i+1]}\right)\Bigg], \qquad (4.58)
\end{aligned}$$

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

$$\begin{aligned}
\boldsymbol{H}_p^{[i;j+1]} &= \boldsymbol{H}_p^{[i;j]} - \alpha_{\mathrm{BB}}^{[i;j]}\mathcal{D}_{\mathrm{NIsoGCN;pressure}}(\boldsymbol{H}_p^{[i;j]}) \qquad (4.59)\\
\mathcal{D}_{\mathrm{NIsoGCN;pressure}}\left(\boldsymbol{H}_p^{[i;j]}\right) &:= \Bigg(\frac{1}{\Delta t}\,\mathrm{NIsoGCN}_{1\to0} \circ \mathrm{NIsoGCN}_{0\to1}\left(\boldsymbol{H}_p^{(:,j)}\right) \\
&\quad - \frac{1}{\Delta t}\mathrm{NIsoGCN}_{1\to0}\left(\hat{\boldsymbol{h}}_{\boldsymbol{u}}^{[i]}\right)\Bigg), \qquad (4.60)
\end{aligned}$$

for $\boldsymbol{H}_p$, where $\boldsymbol{H}_{\boldsymbol{u}}^{[0]} = \boldsymbol{H}_{\boldsymbol{u}}(t)$, $\boldsymbol{H}_p^{[0;0]} = \boldsymbol{H}_p(t)$, and $\boldsymbol{H}_p^{[i;0]} = \boldsymbol{H}_p^{[i]}$. Figures 4.13, 4.14, and 4.15 present the PENN model architecture used for the incompressible flow dataset.

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 $t = 4.0$

As seen in Figure 4.14, 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.

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.

4.4.3.4 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

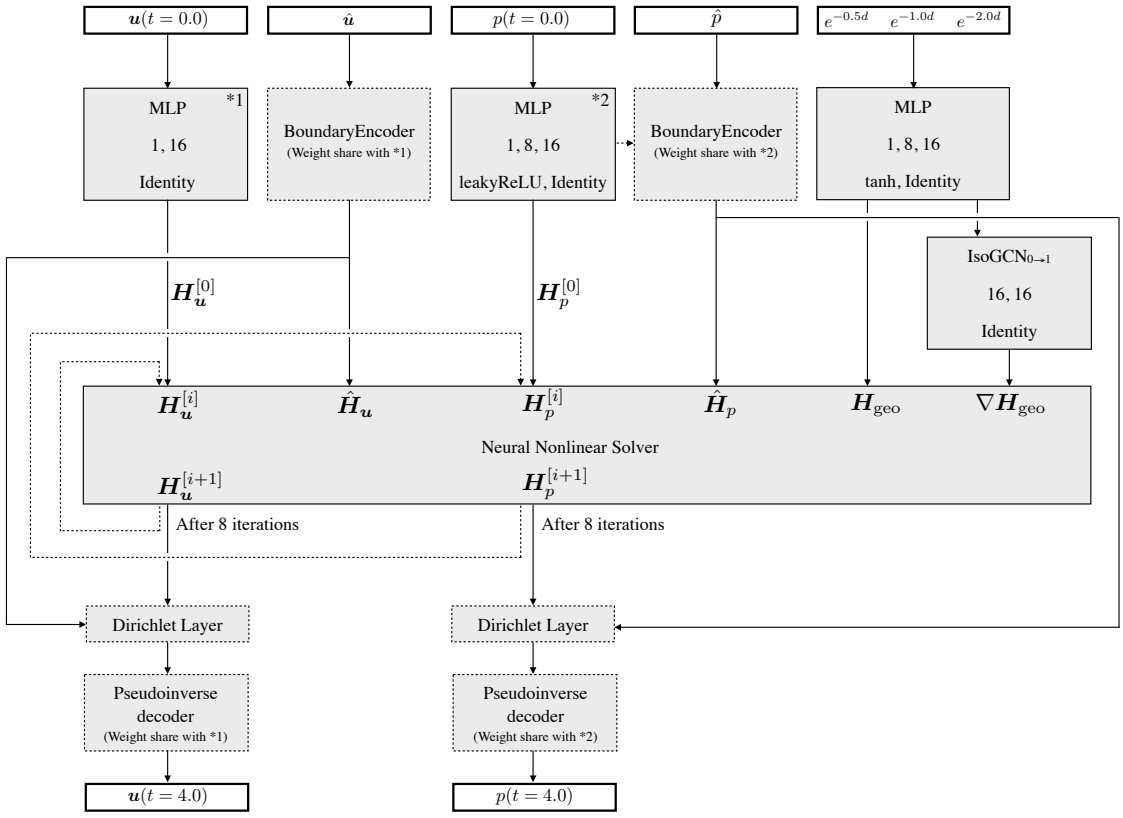


Figure 4.13: The overview of the PENN architecture for the incompressible flow dataset. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. Arrows with dotted lines correspond to the loop. In each trainable cell, we put the number of units in each layer along with the activation functions used.

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.

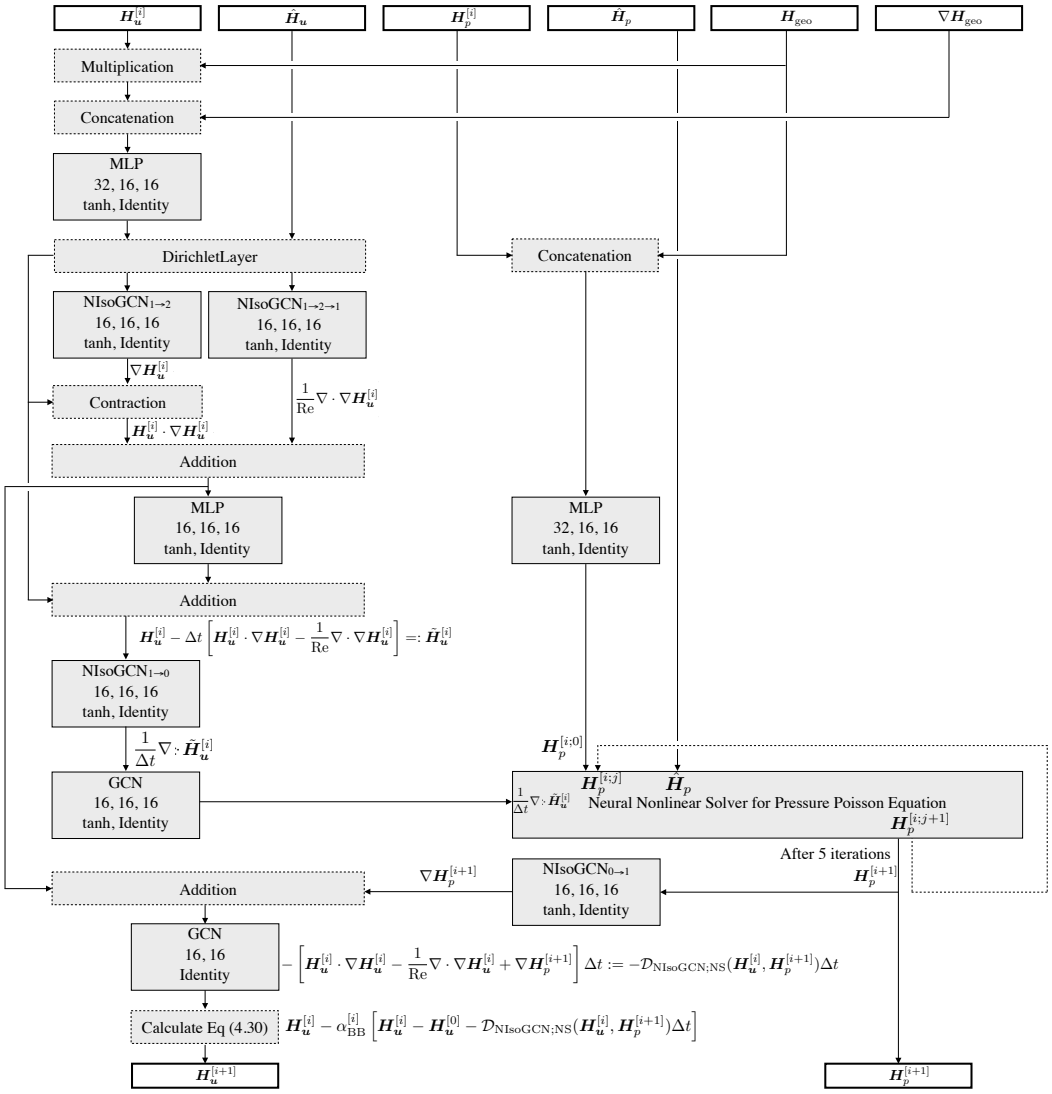


Figure 4.14: The neural nonlinear solver for velocity. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. Arrows with dotted lines correspond to the loop. In each trainable cell, we put the number of units in each layer along with the activation functions used.

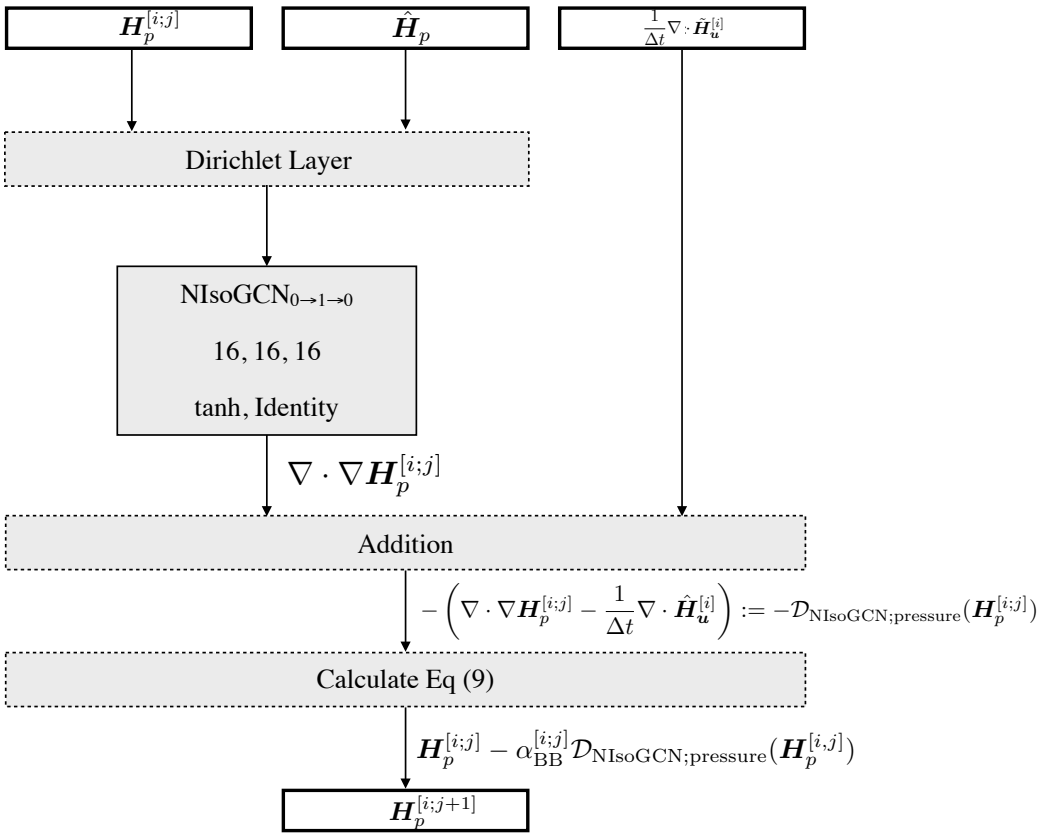


Figure 4.15: The neural nonlinear solver for pressure. Gray boxes with continuous (dotted) lines are trainable (untrainable) components. In each trainable cell, we put the number of units in each layer along with the activation functions used.

4.4.3.5 RESULTS

Table 4.3 and Figure 4.16 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 4.17 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.

Table 4.3: 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. $\hat{\cdot}_{\mathrm{Dirichlet}}$ 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 4.5.

Method	Trans.	$\boldsymbol{u}$ $(\times 10^{-4})$	p $(\times 10^{-3})$	$\hat{\boldsymbol{u}}_{\mathrm{Dirichlet}}$ $(\times 10^{-4})$	$\hat{p}_{\mathrm{Dirichlet}}$ $(\times 10^{-3})$
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	$\mathbf{4.36} \pm 0.03$	$\mathbf{1.17} \pm 0.01$	$\mathbf{0.00} \pm 0.00$	$\mathbf{0.00} \pm 0.00$

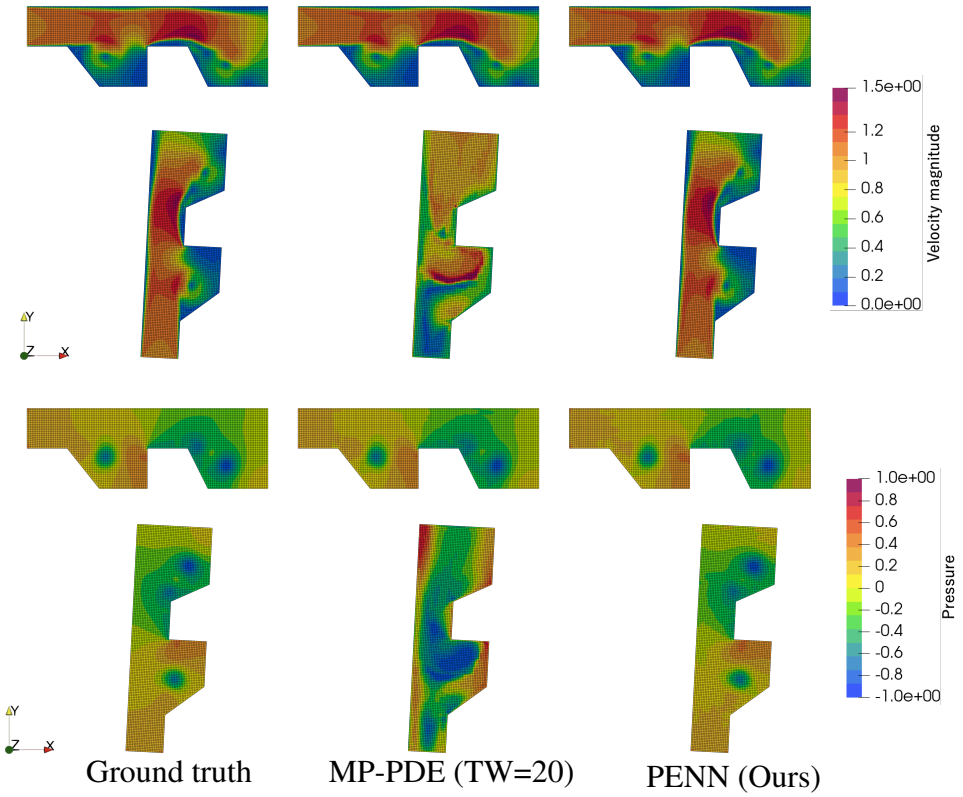


Figure 4.16: Comparison of the velocity field (top two rows) and the pressure field (bottom two rows) without (first and third rows) and with (second and fourth rows) random rotation and translation. PENN prediction is consistent under rotation and translation due to the $\mathrm{E}(n)$-equivariance nature of the model, while MP-PDE's predictive performance degrades under transformations.

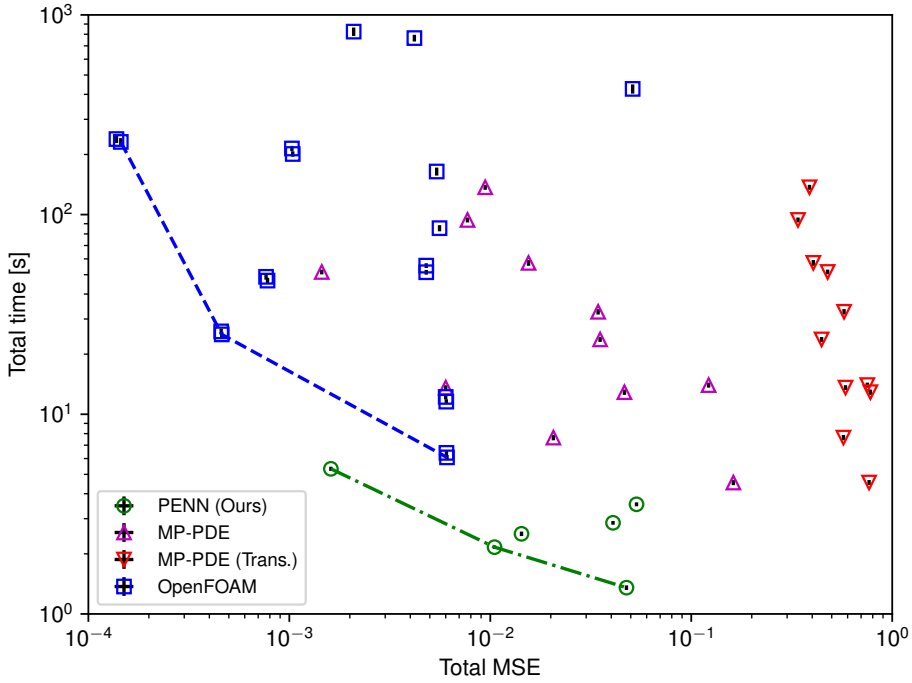


Figure 4.17: Comparison of computation time and total MSE loss ($\boldsymbol{u}$ and p) on the test dataset (with and without transformation) between OpenFOAM, MP-PDE, and PENN. The error bar represents the standard error of the mean. All computation was done using one core of Intel Xeon CPU E5-2695 v2@2.40GHz. Data used to plot this figure are shown in Tables 4.6, 4.7, and 4.8.

4.4.3.6 Ablation Study Results

Similar to the advection-diffusion dataset case, we 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.

(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.

Table 4.4 presents the results of the ablation study. Comparison between models with and without the proposed components shows that the proposed components, i.e., the boundary encoder, Dirichlet layer, pseudoinverse decoder, and neural nonlinear solver, significantly improve the models. The neural nonlinear solver in the encoded space turned out to have the biggest impact on the performance, while the Dirichlet layer ensured reliable models that strictly respect Dirichlet boundary conditions.

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 4.18.

Table 4.4: Ablation study on the 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 ($\sim 10^{14}$).

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

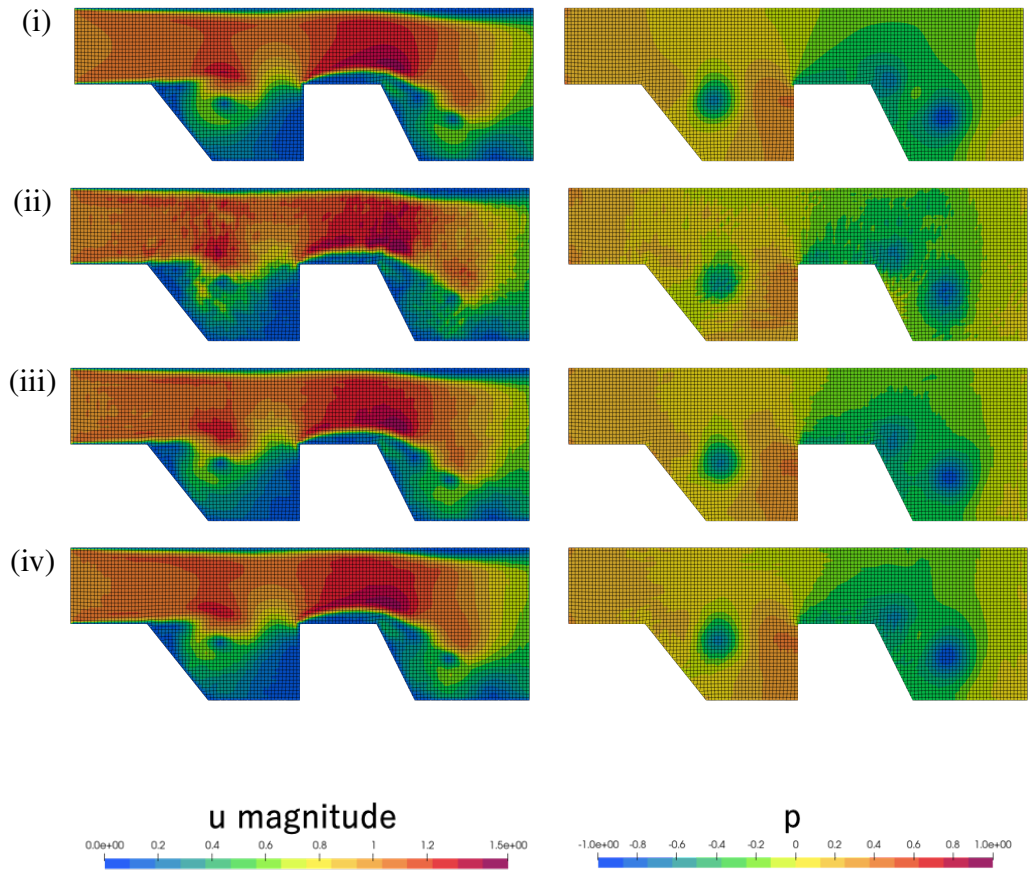


Figure 4.18: Visual comparison of the ablation study of (i) ground truth, (ii) the model without the neural nonlinear solver (Model (C)), (iii) the model without pseudoinverse decoder with Dirichlet layer after decoding (Model (G)), and (iv) PENN. It can be observed that PENN improves the prediction smoothness, especially for the velocity field.

4.4.3.7 DETAILED RESULTS

Table 4.5 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 4.6 and 4.7 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 4.8 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).

Table 4.5: 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 4.6: 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 4.7: 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 4.8: 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

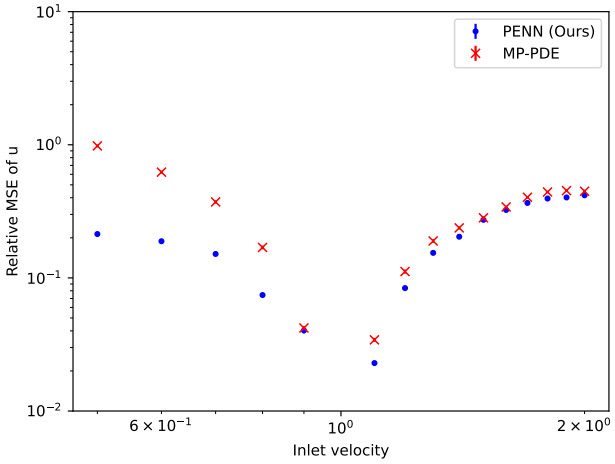


Figure 4.19: The relationship between the relative MSE of the velocity $\boldsymbol{u}$ and inlet velocity.

4.4.3.8 Evaluation of Out-of-Distribution Generalization

We evaluated the out-of-distribution generalizability of PENN and MP-PDE. The models with the best accuracy for each method are used for evaluation. The PENN model has 16 hidden features and eight iterations in the neural nonlinear solver, and the MP-PDE model has 128 hidden features and a time window size of 20.

First, we tested generalizability for Reynolds numbers. We varied Reynolds numbers from 500 to 2,000 by changing inlet velocity u_{inlet} from 0.5 to 2.0, while it was 1.0 for the training dataset. Figures 4.19 and 4.20 show the generalizability regarding inlet velocities, and Figure 4.21 shows the visualization of velocity fields with inlet velocities of 2.0 and 0.5 for each method. For evaluation, we used relative MSE because the magnitude of features may differ drastically with inlet velocity change.

From these figures, one can see that PENN has better accuracy in the lower Reynolds number range while almost no difference in the higher Reynolds numbers. That may be because PENN can deal with boundary conditions rigorously, and training data may contain subdomains where the Reynolds number is small locally.

Then, generalizability regarding shapes is evaluated. We generated ground truth data with the same procedure as that to generate the training dataset, except that the analysis domains used here are larger. Figures 4.22, 4.23, and Table 4.9 present the evaluation

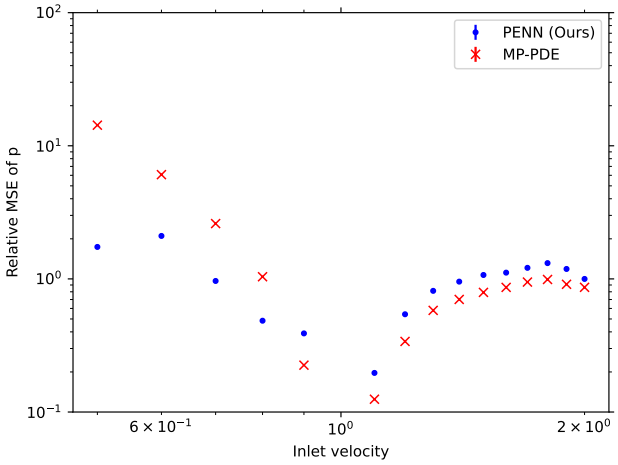


Figure 4.20: The relationship between the relative MSE of the pressure p and inlet velocity.

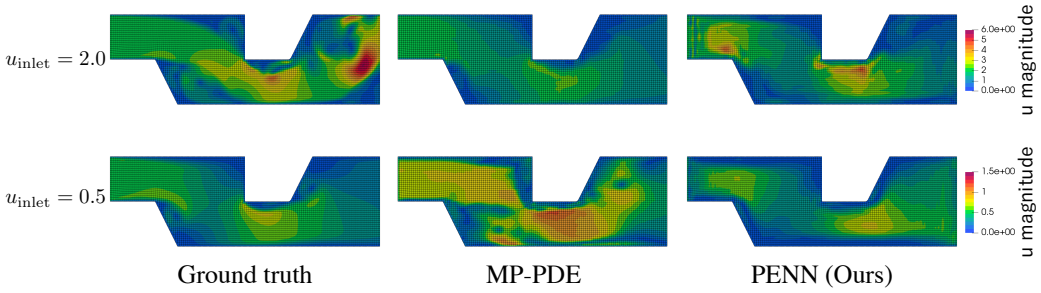


Figure 4.21: The visualization of velocity fields with inlet velocities u_{inlet} of 2.0 and 0.5.

results. Here, we did not observe strong generalizability, such as what was observed in Section 3.4.2. That may be because the global feature introduced by the neural nonlinear solver highly depends on the size of the analysis domain, resulting in relatively poor generalization ability regarding the analysis domain size. The performance degradation is more significant for pressure field prediction than the velocity because it may have stronger global interactions through the pressure Poisson equation, which is a static problem introducing global interaction.

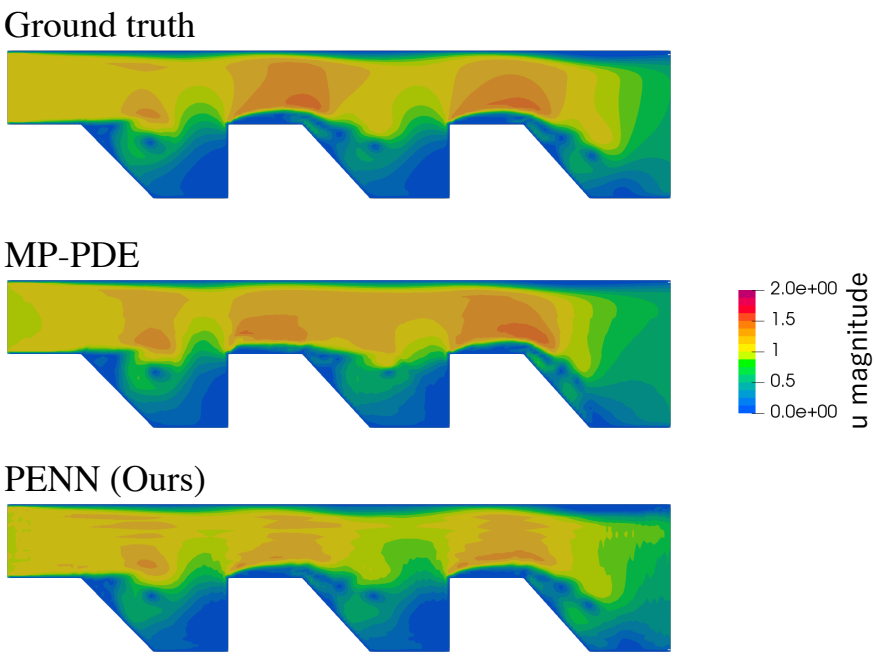


Figure 4.22: The visualization of velocity fields for a larger sample.

4.5 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.

Table 4.9: MSE loss ($\pm$ the standard error of the mean) on the dataset with larger samples. $\hat{g}_{\mathrm{Neumann}}$ is the loss computed only on the boundary where the Neuman condition is set.

Method	$\boldsymbol{u}(\times 10^{-3})$	$p(\times 10^{-2})$
MP-PDE	10.335 ± 0.033	$\mathbf{4.002} \pm 0.005$
PENN (Ours)	$\mathbf{4.132} \pm 0.009$	9.621 ± 0.009

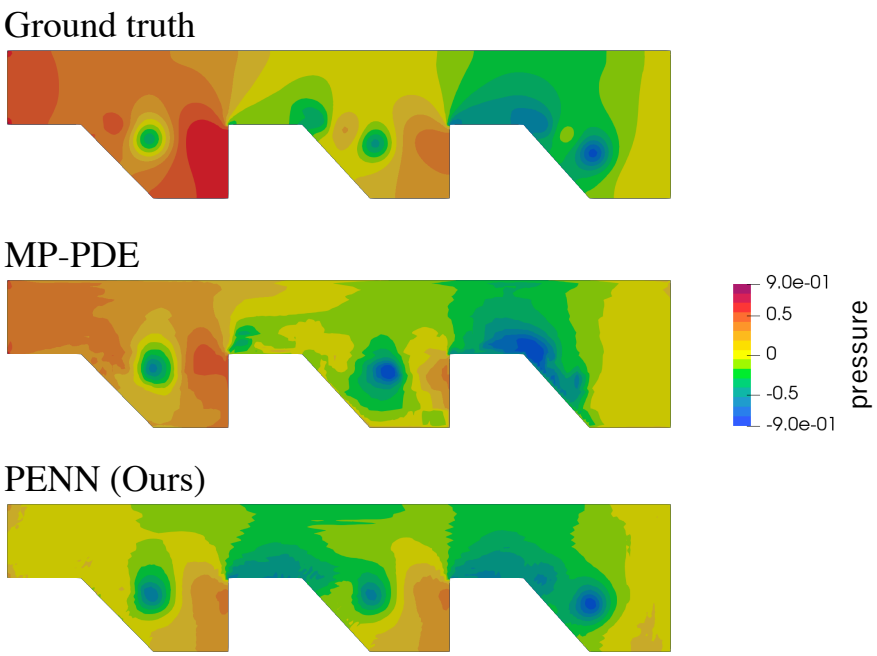


Figure 4.23: The visualization of pressure fields for a larger sample.

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. For instance, the proposed model cannot exploit its potential to solve inverse problems where explicit forms of the governing PDE are not available for such tasks. Therefore, combining PINNs and PENNs could be the next direction of the research community.

Chapter 5

Conclusion

The main contribution of this dissertation is the development of a general neural PDE solvers that are:

- $\mathrm{E}(n)$-equivariant thanks to the use of an IsoGCN (Chapter 3); and
- capable of handling mixed boundary conditions and global interactions by applying the implicit Euler method (Chapter 4).

Through numerical experiments, we demonstrated that our model is capable of accurately predicting heat phenomena on a mesh that is significantly larger than that used in the training phase (Section 3.4). Our approach was also successful in handling various boundary conditions and PDE parameters, in addition to the global interactions that occur in incompressible flow phenomena (Section 4.4). Hereunder, we revisit the objectives outlined in Chapter 1 and evaluate how they were addressed, indicating any existing limitations and suggesting potential avenues for future work.

Flexibility to treat arbitrary meshes based on GNNs In this dissertation, we highlight the flexibility of GNNs in treating arbitrary meshes. Additionally, GNNs offer other desirable characteristics, such as permutation equivariance (Section 3.3) and a generalizability coming from locally-connected nature (Section 3.4). Permutation equivariance is crucial because a mesh can be indexed in various ways, each corresponding to a permutation of indices. The locally-connected nature of GNNs allows for successful predictions on meshes

larger than those used during training, as demonstrated in Section 3.4. However, due to that feature, GNNs may struggle to capture global interactions that occur in fields such as incompressible flow, steady-state analysis, and structural analysis, which will be discussed in a subsequent work.

E(n)-equivariance to reflect physical symmetries Chapter 3 introduced the IsoGCN model, a GNN with E(n)-equivariance capable of learning mesh-discretized physical phenomena from a relatively small dataset. We confirmed that PENN models also have E(n)-equivariance (Section 4.4), which means that their added capability to handle boundary conditions and implicit time evolution is also E(n)-equivariant. However, physical phenomena have other symmetries, such as that with respect to unit changes corresponding to scaling. In particular, some PDEs do not change under scaling as long as certain dimensionless quantities, such as the Reynolds number, remain constant. In contrast, our current model depend on scaling because we use volume features to improve predictive performance. Therefore, a possible future direction could be developing a machine learning model that is, not only permutation- and E(n)-, but also scaling-equivariant. Moreover, incorporating the conservation property could also lead to more stable and accurate predictions.

Computational efficiency to realize faster predictions than with conventional numerical analysis methods The computational efficiency of the IsoGCN model is due to its linear message passing and utilization of the sparse structure of mesh-like graphs (Section 3.3). Additionally, PENN models achieve fast and stable predictions using the Barzilai–Borwein method, a simplified nonlinear solver for implicit time evolution (Section 4.3). However, the speedup observed in numerical experiments is of two to five times, rather than an order of magnitude (Sections 3.4 and 4.4). This might be because we utilized detailed meshes with a large amount of information. To improve computation speed, a possible future direction would be to reduce the number of degrees of freedom in the input mesh.

Accurate consideration of boundary conditions In Chapter 4, we presented a framework for dealing with mixed boundary conditions. Our approach rigorously handles Dirich-

let boundary conditions, but there is room for improvement in satisfying Neumann boundary conditions, which is currently done with a certain degree of error. This may be due to the discretization error inherent in the chosen spatial differential model. To address this issue, we may consider using higher-order approximations of LSMPS differential operators or exploring alternative formalizations. For instance, the weak formulation used in methods such as FEM and FVM might be more accurate in treating Neumann boundary conditions.

Stable prediction over long time steps by accounting for global interactions The PENN model achieves stable predictions for long-term states due to its use of the implicit Euler method, as described in Chapter 4. However, to reduce the computational cost, we were forced to introduce considerable approximations in the implicit formulation, which may compromise its benefits. In fact, as shown in Section 4.4.3.8, such approximations may limit the generalizability of the analysis domain size. To resolve this, we could consider using the quasi-Newton method instead of gradient descent to more accurately solve the implicit equation. Another option may be to use the multigrid method, where the mesh is coarsened within the solver to increase the physical distances visible by the one-hop operation. Additionally, we may also explore the application of the all-to-all connectivity used in the Transformer model (Vaswani et al., 2017).

Despite its limitations, the method proposed in this study lays a solid foundation for the development of practical neural PDE solvers, possessing some desired features, such as the ability to handle arbitrary shapes and boundary conditions. Thus, our work may be a crucial step towards achieving efficient, accurate, and versatile PDE solvers that can contribute to the further advancement of the productivity of human societies.

Bibliography

Eman Ahmed, Alexandre Saint, Abd El Rahman Shabayek, Kseniya Cherenkova, Rig Das, Gleb Gusev, Djamila Aouada, and Bjorn Ottersten. A Survey on Deep Learning Advances on Different 3D Data Representations. *arXiv preprint arXiv:1808.01462*, 2018.

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 *International Conference on Machine Learning*, 2019.

Jonathan Barzilai and Jonathan M Borwein. Two-Point Step Size Gradient Methods. *IMA Journal of Numerical Analysis*, 8(1):141–148, 1988.

Igor I Baskin, Vladimir A Palyulin, and Nikolai S Zefirov. A Neural Device for Searching Direct Correlations Between Structures and Properties of Chemical Compounds. *Journal of Chemical Information and Computer Sciences*, 37(4):715–721, 1997.

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.

Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

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.

Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and Deep Graph Convolutional Networks. *arXiv preprint arXiv:2007.02133*, 2020.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural Ordinary Differential Equations. *Advances in Neural Information Processing Systems*, 31, 2018.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 257–266, 2019.

Taco Cohen and Max Welling. Group Equivariant Convolutional Networks. In *International Conference on Machine Learning*, pp. 2990–2999, 2016.

Taco S Cohen, Mario Geiger, Jonas Köhler, and Max Welling. Spherical CNNs. In *International Conference on Learning Representations*, 2018.

Taco S Cohen, Maurice Weiler, Berkay Kicanaoglu, and Max Welling. Gauge Equivariant Convolutional Networks and the Icosahedral CNN. *International Conference on Machine Learning*, 2019.

George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 5(4):455, 1992.

Xiaowen Dong, Dorina Thanou, Laura Toni, Michael Bronstein, and Pascal Frossard. Graph Signal Processing for Machine Learning: A Review and New Perspectives. *IEEE Signal Processing Magazine*, 37(6):117–127, 2020.

Nadav Dym and Haggai Maron. On the Universality of Rotation Equivariant Point Cloud Networks. *arXiv preprint arXiv:2010.02449*, 2020.

Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference*, pp. 417–426, 2019.

Matthias Fey and Jan E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 869–877, 2018.

Fabian Fuchs, Daniel Worrall, Volker Fischer, and Max Welling. SE(3)-Transformers: 3D Roto-Translation Equivariant Attention Networks. *Advances in Neural Information Processing Systems*, 33, 2020.

Christophe Geuzaine and Jean-François Remacle. Gmsh: a Three-Dimensional Finite Element Mesh Generator with Built-in Pre- and Post-Processing Facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural Message Passing for Quantum Chemistry. In *International Conference on Machine Learning*, 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.

Fangda Gu, Heng Chang, Wenwu Zhu, Somayeh Sojoudi, and Laurent El Ghaoui. Implicit Graph Neural Networks. *Advances in Neural Information Processing Systems*, 33: 11984–11995, 2020.

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

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Masanobu Horie and Naoto Mitsume. Physics-Embedded Neural Networks: Graph Neural PDE Solvers with Mixed Boundary Conditions. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=B3TOg-YCtzo.

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.

Yurie A Ignatieff. Foundations of Rational Continuum Mechanics. In *The Mathematical World of Walter Noll*, pp. 107–125. Springer, 1996.

Yu Ihara, Gaku Hashimoto, and Hiroshi Okuda. Web-Based Integrated Cloud CAE Platform for Large-Scale Finite Element Analysis. *Mechanical Engineering Letters*, 3:17–00520, 2017.

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.

Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional Message Passing for Molecular Graphs. In *International Conference on Learning Representations*, 2020.

Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. ABC: A Big CAD

Model Dataset for Geometric Deep Learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

Risi Kondor. N-body Networks: a Covariant Hierarchical Neural Network Architecture for Learning Atomic Potentials. *arXiv preprint arXiv:1803.01588*, 2018.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning Multiple Layers of Features from Tiny Images. 2009.

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.

Dong C Liu and Jorge Nocedal. On the Limited Memory BFGS Method for Large Scale Optimization. *Mathematical Programming*, 45(1):503–528, 1989.

David G Luenberger, Yinyu Ye, et al. *Linear and Nonlinear Programming*, volume 2. Springer, 1984.

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.

Haggai Maron, Heli Ben-Hamu, Nadav Shamir, and Yaron Lipman. Invariant and Equivariant Graph Networks. *arXiv preprint arXiv:1812.09902*, 2018.

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.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric Deep Learning on Graphs and Manifolds using Mixture Model CNNs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5115–5124, 2017.

Naoki Morita, Kazuo Yonekura, Ichiro Yasuzumi, Mitsuyoshi Tsunori, Gaku Hashimoto, and Hiroshi Okuda. Development of 3× 3 DOF Blocking Structural Elements to Enhance the Computational Intensity of Iterative Linear Solver. *Mechanical Engineering Letters*, 2:16–00082, 2016.

Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *International Conference on Machine Learning*, pp. 807–814, 2010.

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.

Antonio Ortega, Pascal Frossard, Jelena Kovačević, José MF Moura, and Pierre Vandergheynst. Graph Signal Processing: Overview, Challenges, and Applications. *Proceedings of the IEEE*, 106(5):808–828, 2018.

Guofei Pang, Lu Lu, and George Em Karniadakis. fPINNs: Fractional Physics-Informed Neural Networks. *SIAM Journal on Scientific Computing*, 41(4):A2603–A2626, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, pp. 8024–8035. Curran Associates, Inc., 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 Involv-

ing 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.), *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`.

Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph Networks as Learnable Physics Engines for Inference and Control. *arXiv preprint arXiv:1806.01242*, 2018.

Alvaro Sanchez-Gonzalez, Victor Bapst, Kyle Cranmer, and Peter Battaglia. Hamiltonian Graph Networks with ODE Integrators. *arXiv preprint arXiv:1909.12790*, 2019.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W Battaglia. Learning to Simulate Complex Physics with Graph Networks. *arXiv preprint arXiv:2002.09405*, 2020.

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.

Alessandro Sperduti and Antonina Starita. Supervised Neural Networks for the Classification of Structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

Tasuku Tamai and Seiichi Koshizuka. Least Squares Moving Particle Semi-Implicit Method. *Computational Particle Mechanics*, 1(3):277–305, 2014.

Nathaniel Thomas, Tess Smidt, Steven Kearnes, Lusann Yang, Li Li, Kai Kohlhoff, and Patrick Riley. Tensor Field Networks: Rotation-and Translation-Equivariant Neural Networks for 3d Point Clouds. *arXiv preprint arXiv:1802.08219*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *Advances in Neural Information Processing Systems*, 30, 2017.

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.

Maurice Weiler, Mario Geiger, Max Welling, Wouter Boomsma, and Taco S Cohen. 3d Steerable CNNs: Learning Rotationally Equivariant Features in Volumetric Data. In *Advances in Neural Information Processing Systems*, pp. 10381–10392, 2018.

Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying Graph Convolutional Networks. In *International Conference on Machine Learning*, pp. 6861–6871. PMLR, 2019.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? *arXiv preprint arXiv:1810.00826*, 2018.

Jiaxuan You, Rex Ying, and Jure Leskovec. Position-Aware Graph Neural Networks. *arXiv preprint arXiv:1906.04817*, 2019.

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.

Index