Neural Network-Based Surrogate Models of Lagrangian Continuum Simulators

Dody Dharma

Supervisors: Prof. Peter K Jimack, Dr. He Wang

Submitted in accordance with the requirements for the degree of
Doctor of Philosophy

in the

University of Leeds

School of Computing

March, 2025

Declaration

I hereby certify that:

1. This thesis consists solely of my original work conducted for the PhD,
2. Proper acknowledgment has been given in the text to all other sources used,
3. The thesis contains fewer than 100,000 words, not including tables, maps, bibliographies, and appendices.

Dody Dharma, March, 2025

Abstract

The simulation of fluid and deformable solid dynamics is a cornerstone in disciplines ranging from engineering and environmental modeling to computer graphics and virtual reality. Traditional computational methods, despite their power, often struggle to balance accuracy, efficiency, and scalability, particularly in real-time and large-scale applications. This thesis addresses these challenges by developing neural network-based surrogate models specifically tailored for Lagrangian continuum simulations.

The research explores temporal learning within continuum simulations, starting with an in-depth analysis of encoding and decoding techniques that transform Euclidean coordinate-based continuum data into latent space representations. This transformation is essential for training neural networks to accurately model complex physical phenomena. The study systematically compares traditional time series prediction methods with advanced neural architectures, such as Long Short-Term Memory (LSTM) networks. Experimental results demonstrate that LSTM networks, when combined with Multi-Layer Perceptrons (MLP), significantly outperform traditional methods in capturing the intricate multi-material interactions and long-term dependencies inherent in Lagrangian simulations.

A key contribution of this research is the introduction of a Self-Supervised Graph Attention Operator, which enhances the neural network's ability to capture and conserve critical physical properties like vorticity and energy across simulated particles. This operator enables accurate and stable predictions of complex fluid and deformable solid dynamics, overcoming the limitations of existing surrogate models that often compromise between computational efficiency and physical accuracy. Rigorous evaluations, including long-term stability tests and comparative analyses with traditional methods, demonstrate that the proposed models advance the state-of-the-art in surrogate modeling for continuum simulations.

Acknowledgements

All praise is due to Allah, the Most Glorified, the Most High, who has blessed me with strength and guidance to complete this thesis. Peace and salutations be upon Prophet Muhammad (PBUH).

My heartfelt thanks go to my beloved wife, Melati, whose love and companionship was my anchor, and to my son, Adam, whose joy and innocence have brought light to my days.

I am deeply grateful to my mother, father, mother-in-law, father-in-law, brother, and brother-in-law (Mama, Papa, Mande, Ummi, Abi, Rifan, and Umar) for their unwavering prayers and support throughout my life. Their encouragement has been a constant source of strength.

I would like to extend my sincere gratitude to my supervisors, Professor Peter Jimack and Dr. He Wang, for their invaluable support, mentorship, and guidance throughout my doctoral journey. Their wisdom and encouragement have been instrumental in the completion of this work.

I gratefully acknowledge the **LPDP** (Indonesia Endowment Fund for Education Agency) for funding my PhD and this research, and also thank **NVIDIA** for the GPU/Academic Grant Program, which provided the RTX-A5000 GPUs used in our experiments.

A special thanks to Goweser Leeds (Indonesia Cycling Club), CURDIS Leeds, KIBAR Leeds Community, and Doctrine UK (Doctoral Epistemic of Indonesians in the United Kingdom) for their support and for providing an inspiring community during my time in the UK. Your friendship and encouragement have greatly enriched my experience.

Contents

1 Introduction **1**

1.1 Motivation for Lagrangian based Neural Network . . . 3

1.2 Contributions . . . 4

1.3 Thesis Outline . . . 5

2 Related Works **7**

2.1 Navier-Stokes and the Reynolds Number . . . 7

2.2 Lagrangian and Eulerian Perspectives . . . 9

2.3 Hybrid Method . . . 13

2.3.1 PIC, FLIP, APIC . . . 14

2.3.2 MPM . . . 15

2.3.3 Coupling . . . 16

2.4 Deep Learning and Lagrangian Continuum Simulation . . . 18

2.4.1 Deep Learning and fluids . . . 20

2.4.2 Deep Learning and deformable solids . . . 22

2.4.3 Deep Learning and Vorticity . . . 23

2.4.4 Graph Network . . . 24

PointNetConv (PointNet++) . . . 25

Graph Transformer . . . 26

GATConv . . . 26

GATv2Conv . . . 26

2.4.5 Activation Functions . . . 26

3 Methodology **29**

3.1 Analysis of Time Complexity of Deterministic Simulator 29

3.2 Surrogate Model Efficiency 31

3.2.1 Training Efficiency 33

3.2.2 Inference Efficiency 34

3.2.3 Impact of Network Size on Performance 34

3.3 Dataset Generation 35

3.4 General Training Procedure 36

3.5 Evaluation Method 38

4 Temporal Learning of Continuum Simulation **41**

4.1 Encoding and Decoding to Latent Space Representations 42

4.1.1 Encoding Continuum Particles Using MLP 42

4.1.2 Decoding Latent Space Representations 43

4.1.3 Training the Encoder-Decoder Model 43

4.2 Temporal Learning Analysis 44

4.2.1 Traditional Time Series Prediction Methods 45

4.2.2 Temporal Learning with Long Short-Term Memory 46

4.3 Ground Truth for Continuum Simulation 51

4.3.1 Continuum Domain 52

4.3.2 Simulation Process 52

4.3.3 Data Generation and Description 54

4.4 Data Pre-processing and Feature Standardization 55

4.5 Training Procedure 58

4.6 Network Architecture and Hyperparameter Tuning 60

4.6.1 Selection of Activation Function 60

4.6.2 Layer Size 63

4.6.3 Number of Layers 65

4.6.4 Dataset Scaling and Layer Optimization 65

4.7 Proposed Temporal Model With End-to-End Latent Space and LSTM 68

4.7.1 High-Level Architecture 69

4.7.2 Sequential Inputs and Window Size 69

4.7.3 Learned Latent Space Versus Fixed PCA . . . 69
4.7.4 Hidden and Cell States in the LSTM . . . 70
4.7.5 MLP Before the Output . . . 70
4.7.6 Training Details . . . 70
4.8 Evaluation . . . 72
4.8.1 Homogeneous Fluid Surrogate Model Prediction Evaluation . . . 72
4.8.2 Multi-material Surrogate Model Prediction Evaluation . . . 73
4.8.3 Comparison with ARIMA: Large-Scale VAR Approach . . . 75
4.9 Ablation Study . . . 78
4.10 Conclusion . . . 78

5 Vorticity Conservation on Surrogate Model 81
5.1 Motivation . . . 82
5.2 Conservation of Angular Momentum . . . 83
5.2.1 Incompressible Fluid . . . 83
5.2.2 Elastic Fluid / Deformable (Soft) Solid . . . 84
5.2.3 Comparison and Implications . . . 84
5.3 Conservation of Energy Analysis . . . 84
5.3.1 Kinetic Energy in Lagrangian Fluid Simulation . . . 85
5.3.2 Conservation of Kinetic Energy . . . 85
5.4 Measuring Vorticity . . . 85
5.4.1 Curl . . . 86
5.4.2 Vorticity . . . 87
5.5 Vorticity Distribution Evaluation Methods . . . 88
5.5.1 Pearson Correlation . . . 89
5.5.2 Residual Analysis . . . 89
5.5.3 Kullback-Leibler (KL) Divergence . . . 90
5.6 Design Method . . . 92
5.6.1 Network Architecture . . . 92
5.6.2 Self Supervised Graph Attentional Operator . . . 95
5.7 Training Process . . . 98
5.7.1 Data Preparation . . . 98

5.7.2 Metadata and Data Normalization . . . 99
5.7.3 Initialization . . . 101
5.7.4 Loss Functions . . . 102
Defining and Adjusting Loss Weights . . . 104
5.8 2D Lid Driven Cavity Flow Test . . . 106
5.8.1 Dataset Generation . . . 108
Dataset Preprocessing . . . 112
5.8.2 Results . . . 113
Results on Various Test Sets . . . 114
Benchmark: Long-Range Predictions Stability . . . 115
Benchmark: Vorticity distribution . . . 117
Benchmark: Kinetic Energy Conservation . . . 127
5.9 Curl-Noise Flow Test . . . 129
5.9.1 Problem Setting . . . 129
Perlin Noise . . . 130
Curl Noise Generation . . . 130
Simulation Setup . . . 131
5.9.2 Dataset Generation . . . 133
Simulation Setup and Execution . . . 133
Scenario ID, Noise Scale, and Randomness . . . 135
Data Conversion and Organization . . . 137
5.9.3 Training Process . . . 138
5.9.4 Results . . . 140
Results on Various Test Sets . . . 140
Benchmark: Energy Conservation . . . 146
5.10 Conclusion . . . 148

6 Conclusions and Further Investigations 149
6.1 Conclusion . . . 149
6.2 Further Investigations . . . 151

List of Figures

1.1 This Liquid-fabric simulation using physics simulator is astonishing but computationally expensive; running this simulation could take more than 200 seconds per frameFei et al. (2018) 2

2.1 Illustration of Eulerian (a) and Lagrangian (b) standpoin.(York II (1997)) . . . 9

2.2 Particle Interpolation in SPH Method (Wang et al. (2016)) 10

2.3 MAC (Marker Cell Method) (Cline et al. (2005)) 11

2.4 (a) PIC averages particles to a grid node. (b) An equivalent view, particles are naturally fuzzy and have size, which makes it meaningful for particles to have properties like angular momentum. (c) a basic comparison of PIC, FLIP and APIC (Jiang et al. (2015)) 13

2.5 MPM algorithm diagram.(Jiang et al. (2016)) 15

2.6 Algorithm for one- and strong two-way coupling (Benra et al. (2011)) 17

2.7 Neural Network: (a) Comparison of Sparse Connectivity between Convolutional Neural Network (top) and Fully connected network (bottom); (b)(Example of 2D Convolution (Goodfellow et al. (2016)) 19

2.8 Block diagram of a cell of LSTM (Long short term memory) recurrent network (Goodfellow et al. (2016)) 20

2.9 An illustration of a single time step of MLS-MPM. With for forward simulation (top arrows) and backpropagation process (bottom arrows) Hu, Liu, Tenenbaum, Freeman, Wu, Rus & Matusik (2018) 22

2.10 An illustration of a single time step of MLS-MPM. With for forward simulation (top arrows) and backpropagation process (bottom arrows) Holden et al. (2019) 23

3.1 Plots showing the relationship between the quality parameter (q^2) and simulation time per frame (t_{frame}) for the MPM Simulator. (b) The red curve represents a second-degree polynomial fit, highlighting the super-linear relationship that reflects the increasing computational demands at larger scales. 32

4.1 Schematic diagram of a single LSTM memory cell. The cell contains three gates: the input gate i_t, the forget gate f_t, and the output gate o_t. These gates control the flow of information into, within, and out of the cell, allowing the network to manage long-term dependencies effectively. 47

4.2 Dataset for multi-material multi-phase simulation. Top-left: homogeneous liquid; top-middle: snow; top-right: rope and jelly ball; bottom-left: liquid and snow; bottom-middle: rope and snow; bottom-right: rope and liquid. 51

4.3 Dataset of fluid simulation with 2,500 particles per domain. Each row represents a different initial configuration and its evolution over time: **Top row:** A circular fluid body, initially positioned near the center, spreading out and interacting with the boundaries. **Second row:** A rectangular fluid body, similarly positioned near the center, deforming and dispersing under gravity and boundary interaction. **Third row:** A fluid body subjected to an initial force or velocity directed towards the right wall, resulting in the fluid rapidly moving and accumulating against the boundary. **Bottom row:** A complex-shaped splash, with particles having random initial velocities, followed by a free fall under gravity 56

4.4 Comparison of training loss between different activation functions. The horizontal axis represents the number of training epochs. Networks with ELU and ReLU demonstrate faster learning. 61

4.5 Learning curve showing validation and training loss for different activation functions. The small gap between the validation loss and training loss indicates that the network is not overfitted. The horizontal width of each plot corresponds to 500 epochs (unlabeled to simplify the comparison, as the four plots were merged into one image for easier side-by-side comparison). 61

4.6 Comparison of training loss and MSE for networks with different neuron sizes: (a) Epoch-training loss comparison; (b) MSE versus neuron size, highlighting the instability in Configuration 6. 64

4.7 Overview of the selected baseline architecture: ELU [2,500, 1,250, 1,250, 2,500, 10,000] 66
4.8 Training Loss of Selected Architecture (y-axis: MSE, x-axis: number of epochs) 67
4.9 Our LSTM Fluid Network. Two previous frames are used as input to predict the current frame. 68
4.10 Architecture diagram for the proposed model with encoder-decoder and LSTM. 71
4.11 Qualitative comparison of network performance on predicting a liquid state. The **top row** shows ground-truth (physics-based) simulation results, while the **three rows below** show predictions from LSTM+DNN, DNN, and LSTM models (top to bottom). Each column represents a different time step in the simulation. 72
4.12 Qualitative comparison of liquid-snow simulation. The model accurately predicts the deformation and stabilization of both materials. 74
4.13 Qualitative comparisons for various simulations: (top) Snow-snow simulation, (middle) Liquid-rope simulation, and (bottom) Jelly ball-rope simulation. . . . 75

5.1 Propagation of vortex rings in a water tank. The red color indicates the particles with the highest magnitude of vorticity. 81
5.2 Illustration of angular velocity components for a fluid element. The figure shows a square fluid element with initial vertices labeled O, A, and B. The fluid element is subjected to velocity gradients $\frac{\partial u}{\partial y}$ and $\frac{\partial v}{\partial x}$, which cause the element to undergo small angular displacements $d\alpha$ and $d\beta$. These displacements result in a rotational motion with angular velocity ω, represented by the counterclockwise rotation indicated in the figure. 87
5.3 Proposed Neural Network Architecture (OURS) to Conserve Vorticity in Lagrangian Fluid Simulations. The model builds upon the architecture of Sanchez-Gonzalez et al. (2020), with key modifications, including the substitution of the original convolutional layer with the Self-Supervised Graph Attention Operator (SuperGAT). This modification enables the network to better focus on local particle interactions, crucial for vorticity conservation. The Particle Dynamics Stack and SuperGATConv layers have been adapted to ensure that vorticity, angular momentum, and energy are better preserved. 94

5.4 An illustration from Kim & Oh (2022) showing the attention mechanism of SuperGATs (GO, DP, MX, and SD). Blue circles denote the unnormalized attention values before the softmax function, while red diamonds show the edge probability between nodes i and j. The dashed rectangle contains the attention mechanism of the original GAT (Veličković et al. (2018)). 96

5.5 Comparison of a driven cavity flow simulation: (Left) Ground truth, (Middle) Prediction without vorticity loss, and (Right) Prediction with vorticity loss. . . 103

5.6 2D Lid Cavity Simulation. Top Right: weakly compressible MPM Lid Cavity Test, Bottom: Particle Velocity field vs Streamline of 2D Lid Cavity flow dataset. Flow patterns reveal the formation of primary and secondary vortices, including the main vortex driven by the lid and any smaller vortices near the corners of the cavity. The velocity field provides the distribution of velocity components u and v within the cavity 106

5.7 Visualization of the 30 out of 48 initial conditions used for the training dataset. The simulations here are run using a weakly compressible Material Point Method (MPM) model with an initial velocity field under solid boundary conditions outside the shown areas. Color mapping indicates velocity magnitude, and streamlines illustrate the flow field at frame f of certrain secario (example), computed by mapping the particle velocities to a regularly sampled grid using cubic spline interpolants. 110

5.8 Visualization of the 9 out of 48 initial conditions used for the training dataset. Particle indicates position while Color mapping indicates velocity magnitude 111

5.9 Velocity field predicted by our proposed model for frame 54, showcasing stability over extended rollouts 113

5.10 MSE of long term prediction using 6 different test case on OURS model 114

5.11 MSE benchmark comparing the performance of 6 models over long-range prediction. The red color represents our model, showing stability. During intermediate frames (10-40), $\_OURS$ maintains a consistently lower MSE compared to $\_graph_{n}etwork - T0$. The latter shows occasional spikes and higher error variability, indicating less stability in its predictions. 115

5.12 Histogram depicting the vorticity distribution at frames 54. The black lines represent the ground truth vorticity magnitude binned as the reference. 117

5.13 Pearson Correlation Coefficients for Surrogate models and Ground Truth at frame 54. The "OURS" model exhibits the highest correlation. 119

5.14 Residual Plots of vorticity of the surrogate models at frame 54. Proposed model (green-colored $OURS - T0$) pattern has relatively minor residuals that are uniformly spread around the zero axis, demonstrating a strong correlation with the actual data 120

5.15 Scatter Plots of Residuals for Vorticity Models vs. Ground Truth with Cell Index (grid dimension: 40x40) on the X-Axis. The model with the lowest residuals is highlighted in green, while other models are shown in blue. The red dashed line represents zero residuals 121

5.16 KL Divergence of Surrogate Models Compared to Ground Truth. The model with the lowest KL divergence is highlighted in green, indicating the closest match to the ground truth vorticity distribution 122

5.17 Qualitative comparison of predictions from 6 surrogate models at frame 54. The *_ours* model demonstrates the lowest error both in vorticity and velocity prediction 125

5.18 Vorticity distribution (left to right, top to bottom) at frames [6, 9],[18, 21],[48, 51] . The black lines represent the ground truth vorticity magnitude binned as the reference, `_GATv2Conv` (blue) , `_graph_transformer` (orange) , `_GNS` (purple) , `_GATConv` (yellow), `_pointnet_conv` (green) , and `_OURS` (red) 126

5.19 Kinetic Energy Conservations Benchmark of the surrogate models 127

5.20 llustrations of 36 out of (28.8k) conditions employed for the training dataset. The simulations depicted are weakly compressible and generated using the MPM model, starting with a velocity field that is initially divergence-free and constrained by solid boundaries outside the displayed areas. Velocity magnitude is shown through color mapping, while streamlines represent the flow field at time t = f, computed by interpolating particle velocities onto a regular grid using cubic interpolation. 134

5.21 Training Loss of different models using Curl noise Dataset with acccelerationbased prediction. 138

5.22 Quiver plots of Rollout test 0. Our proposed surrogate model predicts the subsequent frames (1-40) given the initial frame (0). 140

5.23 Streamline view for Rollout Tests 0 and 3. First row: Ground Truth Test case 0. Third row: Ground Truth Test case 3. Second and Fourth row are the predictions of our porposed model. 141

5.24 Our-proposed model rollout prediction on frame 35 (with SuperGAT Convolution operator) 142
5.25 Vorticity plot for Rollout 0. The top row shows the target vorticity, while the bottom row shows the predicted vorticity across frames 1, 5, 10, 20, and 30. . . 143
5.26 Results for Rollout Tests 1 and 2 for curl noise simulation. 144
5.27 Results for Rollout Test case 3 and 4 for curl noise simulation. 145
5.28 Comparison of Conservation of Kinetic Energy of our proposed model with relative to others surrogate models. 146

List of Tables

3.1 MPM Computation Time using Parallelism on GPU in various configurations . 30

4.1 Environment Specification for data generation and training on Temporal Learning Experiment 53
4.2 MSE results comparison from a different type of activation function (after 500 epochs) 62
4.3 Benchmarking different unit sizes on Nework with ELU activation 63
4.4 Comparison of different Neural Networks with different layers 65
4.5 Neural Network Training and Validation Metrics on Larger Datasets 66
4.6 Details of the proposed Spatial LSTM architecture with MLP interleaved. . . . 71
4.7 Quantitative network performance on liquid simulation. 73
4.8 Ablation Study. We compare the average error (lower is better) between the ground-truth particle states and the predicted future frames. 79

5.1 Statistical Summary Data Set for Vorticity Conservation Model 100
5.2 MPM Parameters for the Fluid Setting 108
5.3 Statistic of the 2D Lid cavity flow dataset 112
5.4 Comparison of model predictions with ground truth using MAE, MSE, and RMSE at frame and 54 (This was selected as a typical case). 116
5.5 KL Divergence of the vorticity of each surrogate model vs the ground truth at frame 54 (This was selected as a typical case) 122

Chapter 1

Introduction

From manipulating water to fluttering a flag in the breeze or squeezing out a drenched cloth, the interaction between liquids, gases, and solids is pervasive in the physical realm. These phenomena have spurred extensive research into the simulation of continuum materials (fluids and deformable solids). Figure 1.1 illustrates recent work attempting to realistically simulate various continuum materials (fluid, fabric, and porous media phenomena) in a single scene. Research into physics-based continuum simulation techniques mainly aims to enhance visual fidelity (Qu et al. (2019)), simulate new phenomena (Huang et al. (2019)); and improve computational efficiency (Goldade et al. (2019)). In this work, our motivation bridges both enhancing visual realism (through better vorticity modeling) and speeding up computation (via machine learning) to enable real-time interactive graphics applications. By leveraging machine learning algorithms, we aim to conserve critical flow features (e.g. vorticity, density, kinetic energy) while significantly reducing the computational overhead needed for fluid simulations.

Given the need for interactive real-time graphics in domains such as augmented reality and virtual reality, extensive research has been carried out on efficient fluid simulation (Wojtan et al. (2017), Gao et al. (2017)) and deformable solid simulation methods (Overby et al. (2017), Bender et al. (2013), Oh et al. (2008)). Like fluids, simulating deformable solids presents challenges due to the high number of degrees of freedom (DoFs), contact detection and resolution for deformable objects, and potential self-collisions. The complexity increases when simulating interactions between fluids and deformable solids due to the requirement to simulate fluid interactions with solid interfaces. The researchers aim to accelerate simulation

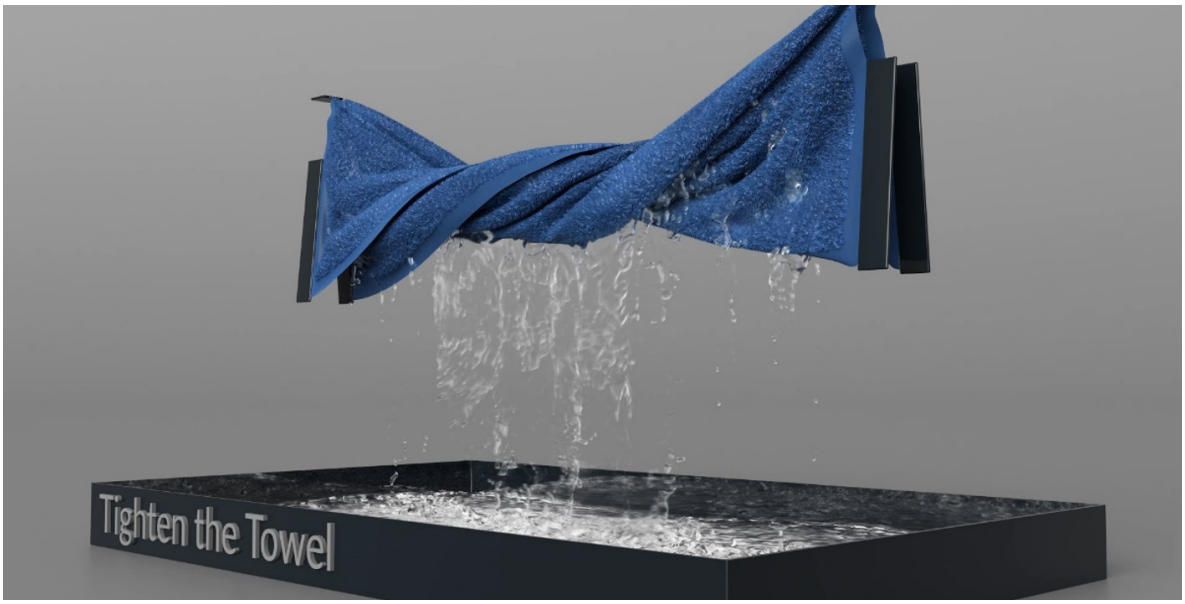


Figure 1.1: *This Liquid-fabric simulation using physics simulator is astonishing but computationally expensive; running this simulation could take more than 200 seconds per frameFei et al. (2018)*

by optimizing algorithms by parallel computing on GPUs (Nie et al. (2015), Dharma et al. (2017), Gao et al. (2018)) or CPUs (Gropp et al. (2001)). Over the past decade, numerous researchers have also employed advanced Machine Learning techniques in simulators (Tompson et al. (2017), Xie et al. (2018), Chu & Thuerey (2017), Kim et al. (2019)).

This research poses the question: can we achieve real-time interactive application of unified multi-material - multi-phase solid-fluid simulation through advances in Machine Learning techniques? Recent works offer initial steps toward this goal, but the efforts remain fragmented between fluid and deformable solid simulations. Some research focuses on accelerating fluid simulation using neural networks (Ladický et al. (2015), Tompson et al. (2017), Kim et al. (2019)), while others aim to speed up fabric simulation (Lee et al. (2019), Laehner et al. (2018), Holden et al. (2019)). There is also recent work adopting the Lagrangian representation to generalize fluid-solid boundary interactions (Ummenhofer et al. (2019)). This PhD research aims to build on these efforts, accelerating the simulation of solids and liquids both together and separately through novel Deep Neural Networks.

Even with recent progress, accurately depicting flow characteristics like vorticity and eddies, discerning underlying dynamic systems, and forecasting future states pose significant

challenges in scientific machine learning, particularly in the context of Lagrangian representation. The majority of current research focuses on Eulerian representations, whereas the focus of this thesis is around Lagrangian, particle-based, simulations.

Comprehending hidden flow dynamics poses significant challenges. Fluid systems governed by the Navier-Stokes equations intricately interconnect multiple physical variables, including velocity, pressure, and density. Nonetheless, in practical applications, only density information is commonly available. The intricacy, uncertainty, and non-linearity of these systems make it unfeasible to infer underlying dynamics directly from observable density data. Effective learning frequently requires supervision of velocity or pressure, adding complexity to real-world implementation.

1.1 Motivation for Lagrangian based Neural Network

Capturing flow features is a complex challenge in scientific machine learning. Traditional methods for representing fluid dynamics often rely on storing velocity fields on regularly spaced grids or employing smooth neural networks. While these approaches have shown potential in modeling relatively smooth flow phenomena, they struggle with irregular, continuous data structures and turbulent fluid systems across different scales. Challenges such as the curse of dimensionality in detailed space-time domains, local discontinuities, and hidden constraints persist. Consequently, there is a growing need for more compact and structured representation spaces and data structures to advance Lagrangian-oriented approaches in scientific machine learning.

Lagrangian continuum simulation is widely used in real-time applications, as evidenced by works like Monaghan (1992*b*), Macklin & Müller (2013), Auer (2009), and Ihmsen et al. (2014). Despite the popularity of these approaches, there has been a growing interest in training the surrogate models within the Eulerian representation, as seen in various studies (Tompson et al. (2017)). This includes PDE-based frameworks Raissi et al. (2019), applications in robotic control and planning (Schenck & Fox (2018), Hu, Liu, Tenenbaum, Freeman, Wu, Rus & Matusik (2018)), and more efficient cloth simulations (Lee et al. (2019), Tan et al. (2019)). For instance, Chu et al. (2022) optimizes fluid dynamics from sparse video data, while Deng et al. (2023) focuses on learning vortex particle dynamics from regular grid data using single videos without requiring ground truth velocity.

However, the development of fluid or deformable solid predictors in particle-based or material point-based methods remains limited. Recent efforts, such as continuous convolution Ummenhofer et al. (2019) and Graph Network simulators (Ladický et al. (2015)), have shown promise, but the field is still in its infancy. Similarly, Ummenhofer et al. (2020) employs continuous convolutions to process sets of moving particles, while Prantl et al. (2022) guarantees the conservation of linear momentum in learned physics simulations.

Additionally, Xiong et al. (2023) combines the Discrete Vortex Method with neural networks, further highlighting the trend towards integrating vorticity into deep learning models for fluid dynamics. Despite these advancements, significant challenges remain in effectively merging these techniques with Lagrangian representations.

There are a number of challenges that we address in this field of research. One of the key challenges is the active exploration of temporal modeling of fluids, which aims to manage simulation errors in long-term predictions involving fluids or highly deformed solids (Xie et al. (2019), Kim et al. (2019), Wiewel et al. (2020)). Additionally, while many attempts have been made to develop surrogate models for fluid vorticity in Eulerian representation, studies concerning the conservation of fluid vorticity in Lagrangian representation remain scarce.

1.2 Contributions

The main contributions of our investigation are:

1. We explore temporal learning in continuum simulations, focusing on encoding and decoding techniques, including the transformation of Euclidean data into latent space. We also address challenges like hyperparameter tuning and dataset size impacts on model performance.

2. We extend the architecture of Long Short-Term Memory Networks by integrating a Multi-Layer Perceptron, aiming to improve prediction accuracy in unified multimaterial simulations, particularly for liquids and deformable solids.

3. We propose a novel approach that incorporates the Self-Supervised Graph Attention Networks (SuperGAT) convolution operator into a surrogate model for fluid simulation. This ensures more accurate predictions of rotational dynamics by conserving vorticity.

4. We enhance the training framework of the Graph-Based Simulator surrogate model to include both acceleration-based and velocity-based predictions, thereby increasing the model's versatility in capturing dynamic behaviors.

5. We develop an extended loss function framework designed to capture the key physical aspects of Lagrangian-based simulations. This framework includes loss terms for acceleration, velocity, position, density, divergence, and vorticity, ensuring a comprehensive understanding of the dynamics being modeled and improving the model's ability to predict complex physical systems holistically.

6. We extend the evaluation metrics to robustly assess the surrogate model's performance in conserving vorticity using multiple methods. Our analysis involves comparing the similarity between ground truth and model predictions by interpolating particle-based quantities onto a fixed-size uniform grid to obtain vorticity values across the entire domain. We employ three methods for this comparison: Pearson correlation, residual analysis, and Kullback-Leibler (KL) divergence, each offering distinct advantages.

1.3 Thesis Outline

Chapter 2. This chapter provides an overview of the fundamental issues in computational fluid dynamics (CFD), focusing on the differences between Lagrangian and Eulerian perspectives. It discusses various simulation methods, including PIC, FLIP, APIC, and MPM, and how these are combined in hybrid approaches. The chapter also reviews the intersection of fluid mechanics with machine learning, highlighting the use of deep learning in fluid and deformable solid simulations.

Chapter 3. The first part of our contributions is presented in this chapter, where we explore temporal learning in continuum simulations. We delve into encoding and decoding techniques, specifically focusing on the transformation of Euclidean data into latent space representations. This transformation is crucial for effectively training neural networks to model complex physical phenomena. The chapter also addresses challenges such as hyperparameter tuning and the impact of dataset size on model performance, ensuring the development of robust and scalable models.

Chapter 4. The second key contribution of this thesis is detailed here, where we extend the architecture of Long Short-Term Memory (LSTM) Networks by integrating a Multi-Layer Perceptron (MLP). This integration aims to improve prediction accuracy in unified multi-material simulations, particularly for liquids and deformable solids. The chapter provides a comprehensive analysis of temporal learning, comparing traditional time series prediction methods with our advanced LSTM-MLP architecture. We introduce and evaluate our proposed temporal model, demonstrating its effectiveness in capturing the dynamics of complex systems over long-term simulations.

Chapter 5. This chapter delves into the conservation of vorticity in surrogate models, a critical aspect of accurately simulating fluid dynamics. We propose a novel approach that incorporates the Self-Supervised Graph Attention Networks (SuperGAT) convolution operator into a surrogate model for fluid simulation. This ensures more accurate predictions of rotational dynamics by conserving vorticity, addressing a key limitation in existing surrogate models. The chapter also enhances the training framework of the Graph-Based Simulator surrogate model to include both acceleration-based and velocity-based predictions, thereby increasing the model's versatility in capturing dynamic behaviors. Additionally, we introduce an extended loss function framework designed to capture the key physical aspects of Lagrangian-based simulations, including acceleration, velocity, position, density, divergence, and vorticity. We also incorporate a robust evaluation framework to assess the surrogate model's performance in conserving vorticity. Our evaluation involves comparing the similarity between ground truth and model predictions by interpolating particle-based quantities onto a fixed-size uniform grid to obtain vorticity values across the entire domain. We employ multiple methods for this comparison, including Pearson correlation, residual analysis, and Kullback-Leibler (KL) divergence, each offering distinct advantages. The chapter concludes with potential extensions and improvements to the proposed methods, particularly in the context of real-time applications and more complex fluid-solid interactions.

Chapter 2

Related Works

This chapter briefly introduces related work on fluid or deformable solid simulation used in Computer Graphics. Since this research is driven by the goal of enabling real-time interactive graphics, we will briefly discuss core Computational Fluid Dynamics (CFD) concepts, optimizations for computational performance, and the intersection of fluid mechanics with machine learning. The contents of this section are mainly derived from (Bridson (2015*a*)). We summarise the essential concepts for understanding continuum material simulation, without delving into the details of the Navier-Stokes equations. The chapter begins with a description of discretization techniques for fluid domains from Lagrangian, Eulerian, and hybrid perspectives, followed by a discussion on the coupling between fluid and solid simulations. Finally, we will examine the state-of-the-art in fluid simulation using Deep Learning techniques.

2.1 Navier-Stokes and the Reynolds Number

The incompressible Navier-Stokes equations describe the behavior of a Newtonian incompressible fluid. The first equation, which expresses the incompressibility condition, is given by

$$\nabla \cdot \vec{u} = 0, \tag{2.1}$$

where $\vec{u}$ is the fluid velocity vector. This equation ensures that the volume of any fluid element remains constant over time, meaning that there is no net compression or expansion of the fluid.

The momentum equation, which governs the dynamics of the fluid, is expressed as

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \Delta \vec{u}. \tag{2.2}$$

In this equation, $\frac{\partial \vec{u}}{\partial t}$ denotes the local (or unsteady) acceleration, which is the rate of change of velocity at a fixed point in space. The term $\vec{u} \cdot \nabla \vec{u}$ represents the convective acceleration, describing the change in velocity that a fluid particle experiences as it moves through a spatially varying velocity field. The pressure gradient force, $\frac{1}{\rho} \nabla p$, drives the fluid from regions of high pressure to low pressure, while $\vec{g}$ represents external body forces such as gravity. Finally, the term $\nu \Delta \vec{u}$ accounts for the diffusion of momentum due to viscous effects, where ν is the kinematic viscosity.

Non-dimensional analysis is a fundamental approach in fluid dynamics that enables the comparison of flow behaviors across different systems and scales. One of the most important non-dimensional parameters is the Reynolds number, defined as

$$Re = \frac{\rho U L}{\mu}, \tag{2.3}$$

where ρ is the fluid density, U is a representative velocity scale (such as the mean or maximum velocity), L is a characteristic length scale (for example, the diameter of a pipe or the size of an obstacle), and μ is the dynamic viscosity of the fluid. The Reynolds number essentially represents the ratio of inertial forces to viscous forces. When $Re \ll 1$, viscous forces dominate, leading to smooth and steady flows often described as creeping or Stokes flow. For flows where Re is on the order of 10^2, both inertial and viscous forces play important roles and the flow generally remains laminar. In contrast, when Re exceeds approximately 10^3, inertial effects become dominant and the flow is likely to transition to turbulence, which is characterized by chaotic fluctuations and enhanced mixing.

Note: The dynamic viscosity μ is also related to the density ρ and the kinematic viscosity ν by the relation

$$\mu = \rho \, \nu. \tag{2.4}$$

(?)

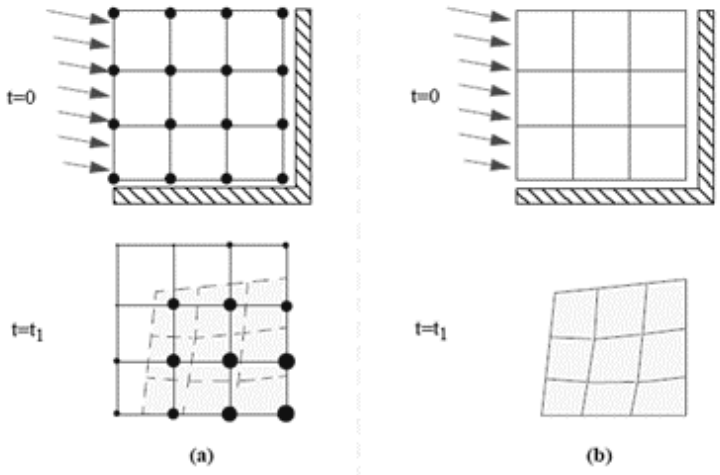


Figure 2.1: *Illustration of Eulerian (a) and Lagrangian (b) standpoin.(York II (1997))*

2.2 Lagrangian and Eulerian Perspectives

There are two common perspectives to tracking the motion when observing a moving continuum (fluid or deformable solid): Lagrangian standpoint and Eulerian viewpoint. The Lagrangian view represents the continuum as an evolving system through the tracking of particles (with or without a mesh connecting up the particles). Every single point in the continuum may be specified as a particle that has properties of position and velocity. However, the Eulerian method uses a different technique. It looks at static points in space (fixed grid) and evaluates the change of fluid quantities (such as velocity, temperature, density, etc.) at those points in time Bridson (2015*a*).

In the Eulerian method (Figure 2.1a, the grid nodes (black dots) represent the quantities of material, such as mass. At the next time step in the simulation, t1, due to material deformation governed by external and internal forces, the mass at the grid nodes has changed (convected), but the grid remains fixed in space. In Figure 2.1a, the size of the black dots also shows the change in the amount of mass. The material boundaries are designated with dashed lines and describe the corresponding mass distribution. In Figure 2.1b (Lagrangian), the material is represented by the grid which distorts with the material

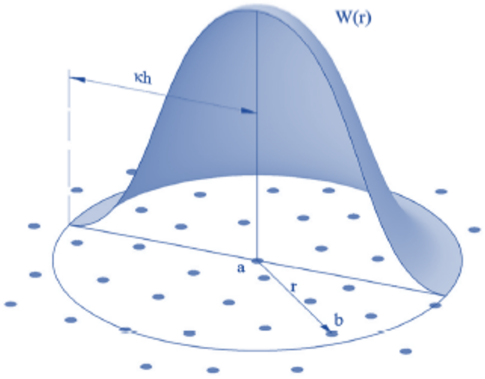


Figure 2.2: *Particle Interpolation in SPH Method (Wang et al. (2016))*

The Lagrangian approach is intuitive in defining the material boundaries, especially in the multiphase or history-dependent simulation, since it does not need a particular procedure to separate the interface of the interacting materials (provided that the grid deforms accurately with the material).

An example of a Langrangian Approach is Lagrangian Finite Element Method (Ahamadi & Harlen (2008*a*)). In this method, the deforming computational domain is discretized by means of a finite element mesh that deforms with the flow (Ahamadi & Harlen (2008*a*)). The method allows simulations of suspensions under planar extensional flow to be conducted to large strains in a truly periodic cell. The method is illustrated for both isotropic and anisotropic two-dimensional particles and can be easily extended to viscoelastic fluids (Ahamadi & Harlen (2008*b*)) and to non-rigid particles (Ahamadi & Harlen (2008*a*)).

Another prominent example of the Lagrangian approach is SPH (Smoothed Particle Hydrodynamics) (Wang et al. (2016), Gingold & Monaghan (1977), Auer (2009), Ihmsen et al. (2014), Müller et al. (2003), Monaghan (1992*b*)). Initially used in astronomy to simulate gas dynamics on a large scale (astrophysics), but later it was also applied to the problem of incompressible flow such as simulations of ocean waves and liquid mud in tanks (Auer (2009)), and interactive application (Ihmsen et al. (2014)), (Müller et al. (2003)). In the SPH method, the computational domain is discretized into a finite number of particles. This can be illustrated briefly by Figure 2.2. The continuous integral representation of the field variable f(x) can then be approximated by summation over the neighbouring particles using the smoothing/kernel function W and smoothing length, r, defining the influence domain of W (Wang et al. (2016)).

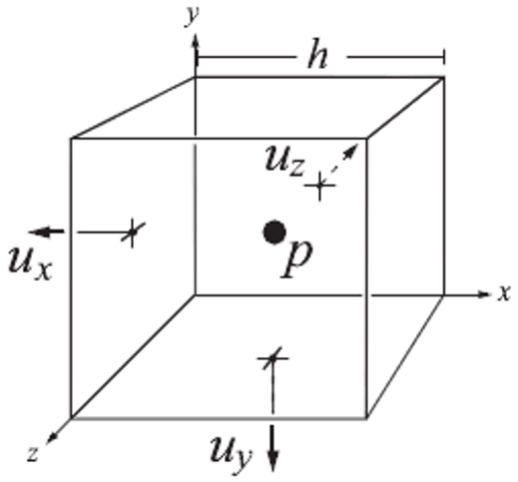


Figure 2.3: *MAC (Marker Cell Method) (Cline et al. (2005))*

To Calculate spatial derivatives, SPH uses analytical differentiation of interpolation formulae. The essential part of SPH is an interpolation method which allows any function to be expressed in terms of its values at a set of disordered particles (Monaghan (1992*b*)). SPH distributes local quantities to the neighbouring particles of each particle using radial symmetrical smoothing kernels (Müller et al. (2003)). Compared to the Lagrangian approach, the main advantage of the Eulerian method is the ability to simulate large-scale deformations and high distortion flow (*The Science of Fluid Sims* (2011)). Because of these characteristics, the grid approach is suitable for creating large oceanic effects. However, to achieve sufficient quality details for the close-up visual effects (thin fluids), the grid needs adaptive mesh refinement (AMR) (Wang et al. (2017)) . One example of the Eulerian method is the MAC (Marker and Cell method) (Cline et al. (2005)).

Figure 2.3 shows how MAC discretizes the space into cubes cell with width h. Every cell has pressure, p, which defined at the centre of the cube. The cube has velocity $u = (u_x, u_y, u_z)$, however, the velocity components are placed at the centre of the cell's surfaces. u_x at the $x-minimum$ surface, u_y at $y-minimum$, and u_z at $z-minimum$. The velocity component position is varied to improve the simulation stability rather than put all the components at the centre of the cube cell (Cline et al. (2005)). Pixar Animation Studios make use of this method for water jet animation in a film called Ratatouille (Bird & Pinkava (2007)). To reduce the number of cells and the computational cost, instead of discretizing water into uniform cubes,

Pixar modify the cube into cells with different heights (bars), placing exact cubes on the surface and the bottom of fluid geometry but placing a bar (tall or long) cell at the centre of the geometry (areas that are not visually significant), thus increasing speed of simulation (Seymour (2014)).

2.3 Hybrid Method

Hybrid techniques for fluid simulation seek to merge the benefits of both Eulerian and Lagrangian frameworks. Conventionally, this has been done in computational fluid dynamics using arbitrary Lagrangian-Eulerian (ALE) approaches (refer to (Walkley et al. (2005)) for more details). Nonetheless, this section emphasizes the progression of other hybrid methods within computer graphics, including PIC (Particle in Cell), FLIP (Fluid Implicit Particle), APIC (Affine Particle in Cell), and MPM (Material Point Method).

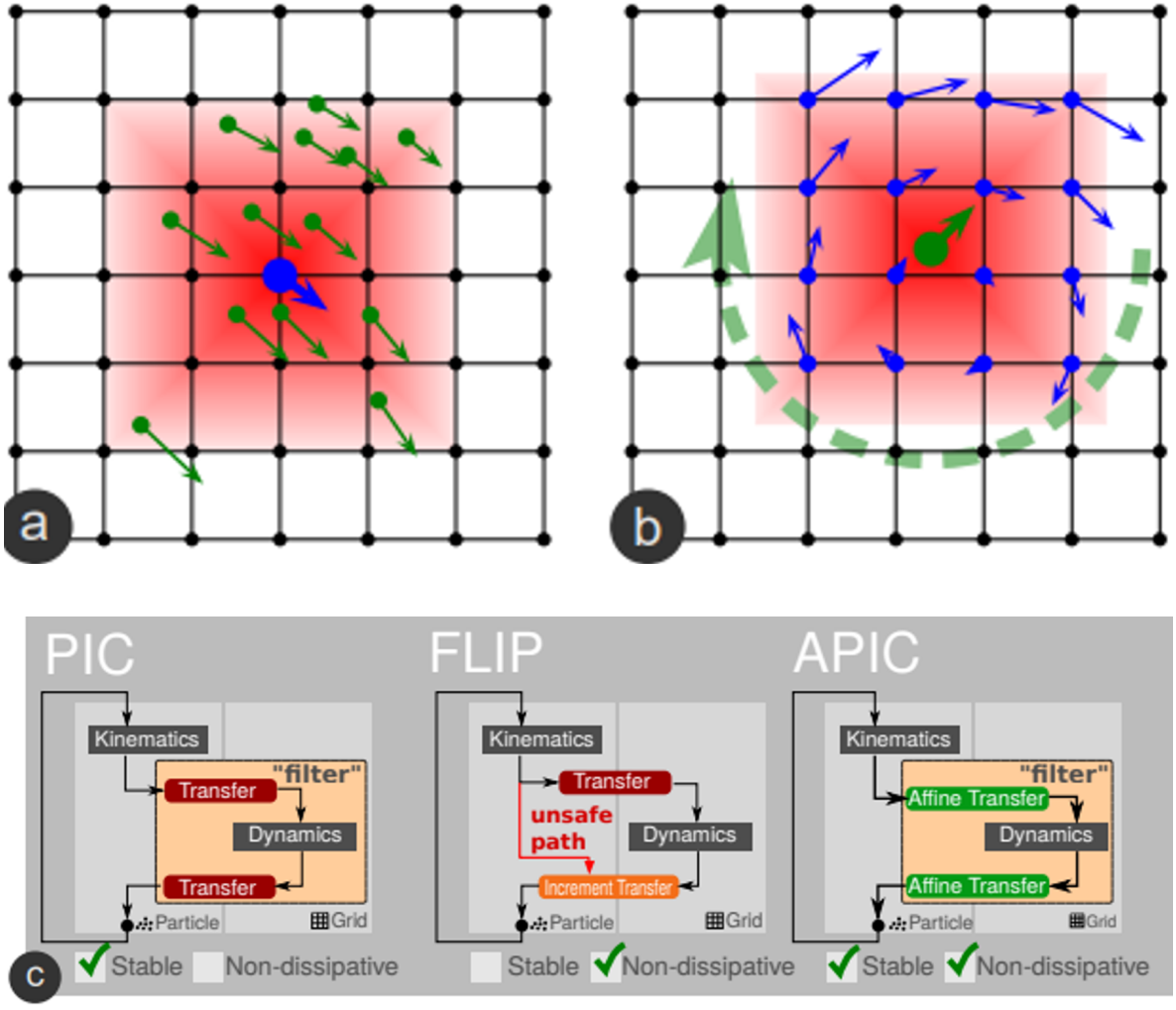


Figure 2.4: *(a) PIC averages particles to a grid node. (b) An equivalent view, particles are naturally fuzzy and have size, which makes it meaningful for particles to have properties like angular momentum. (c) a basic comparison of PIC, FLIP and APIC (Jiang et al. (2015))*

2.3.1 PIC, FLIP, APIC

Figure 2.4 (a) illustrates how PIC averages particles to a grid node. (b) Depicts an equivalent view where particles are inherently fuzzy and possess size, making it possible for particles to have properties such as angular momentum. (c) Presents a basic comparison of PIC, FLIP, and APIC (Jiang et al. (2015)). Reference (Bridson (2015*b*)) offers a comprehensive review of hybrid methods for fluid simulation that aim to merge the benefits of the Eulerian and Lagrangian approaches. The innovator of this hybrid method is the PIC (Particle in Cell) introduced by (Harlow (1963)). In PIC, all quantities, including velocities, are initially stored on particles that sample the entire fluid domain. At each time step, these quantities are transferred to the grid. All non-advection quantities (e.g., pressure projection, acceleration, boundary resolution, viscosity) are computed on the grid. The step ends by interpolating quantities from the grid to the particles, followed by advection of the particles in the velocity field of the grid. However, PIC can suffer from significant numerical dissipation as fluid quantities are averaged from particles to the grid, introducing some smoothing. Due to the bidirectional transfer process in each cycle, considerable smoothing accumulates at each time step.

To address this, FLIP method was developed (Brackbill & Ruppel (1986)). Instead of replacing the particle value during the grid-to-particle transfer, FLIP interpolates the change in quantity (calculated on the grid) to increment the particle value. A drawback of FLIP is that it can generate noise; velocity fluctuations in particles can average down to zero and vanish from the grid in some time steps, while appearing as unexpected perturbations in others (Bridson (2015*b*)). To maintain the stability of PIC without the noise issues of FLIP, the APIC method was introduced (Jiang et al. (2015)).

APIC mitigates the dissipation problem by enhancing each particle with a local affine velocity description, preserving information during grid-particle transfers. This approach effectively eliminates dissipation and allows for the conservation of angular momentum during the transfer process. The comparison of PIC, FLIP, and APIC is summarized in Figure 2.4. In Figure 2.4a, particles are transferred to grid nodes using PIC. In Figure 2.4b, the curved arrows represent particles' angular momentum properties. In Figure 2.4c, FLIP addresses PIC's dissipation issue by creating an unstable path, whereas APIC uses affine transfer to eliminate dissipation and maintain stability.

2.3.2 MPM

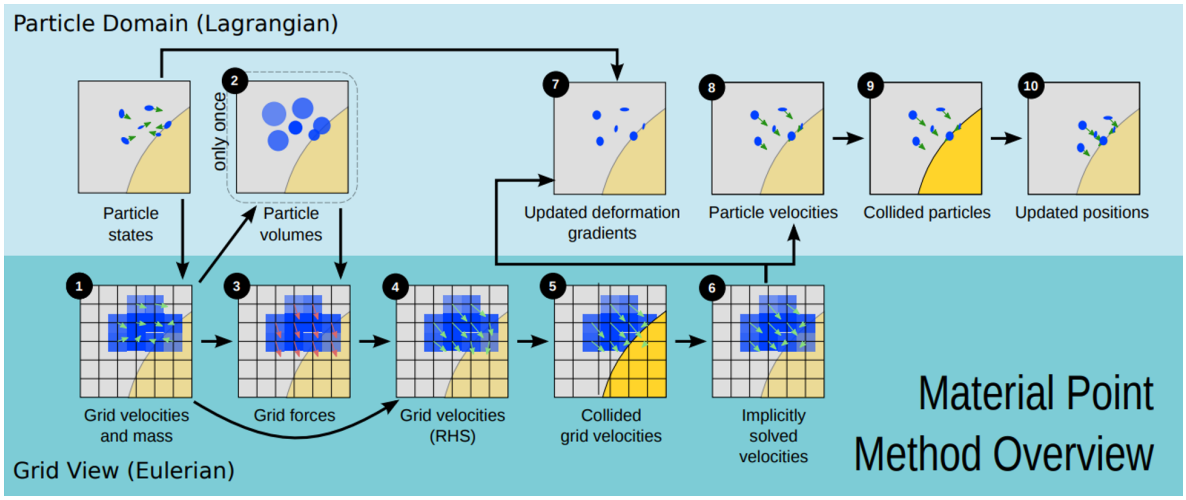


Figure 2.5: *MPM algorithm diagram.(Jiang et al. (2016))*

The hybrid methods mentioned previously have been used primarily in liquid simulation. Another hybrid method that was used as a generalization of PIC to handle deformable solid mechanics is MPM (Material Point Method) (Sulsky et al. (1994)). MPM grew in popularity after 2013 when it was used to simulate snow (Stomakhin et al. (2013)) in Disney's Frozen movie. MPM uses a continuum description of the governing equations and utilizes user-controllable elastoplastic constitutive models. The hybrid nature of MPM allows using a regular Cartesian grid to automate treatment of self-collision and fracture. Like other particle methods such as SPH (Monaghan (1992*b*)), topology change is easy because of the lack of explicit connectivity between Lagrangian particles. Furthermore, MPM allows for a grid-based implicit integration scheme that has conditioning independent of the number of Lagrangian particles. MPM also provides a unified particle simulation framework similar to Position Based Dynamics (PBD) (Macklin & Müller (2013)) for easy coupling of different materials.

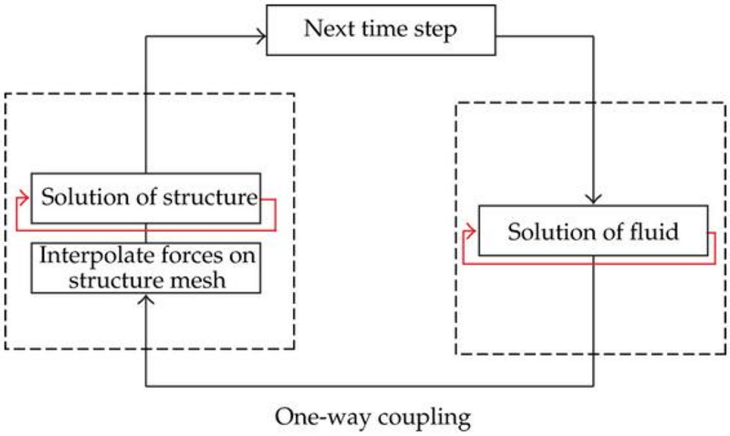
Later, MPM has been extensively developed to simulate various phenomena. This includes, the dynamics of non-Newtonian fluids and foams (Ram et al. (2015)), (Yue et al. (2015)) , the melting process or phase changing through heat transfer (Stomakhin et al. (2014)), (Gao et al. (2017)), frictional contact between granular materials (Klár et al. (2016),Yue et al. (2018)), Rigid-body coupling and cutting (Hu, Fang, Ge, Qu, Zhu, Pradhana & Jiang (2018)), porous media and solid-fluid coupling (Fei et al. (2018), Tampubolon et al. (2017), Fei et al. (2019)),

contact and collision with volumetric elastic objects (Jiang et al. (2015)), (Zhu et al. (2017)), frictional contact among hyper-elastic materials like hair and clothes (Guo et al. (2018), Jiang et al. (2017), Fei et al. (2018)) and dynamic fracture animation (Wolper et al. (2019)).

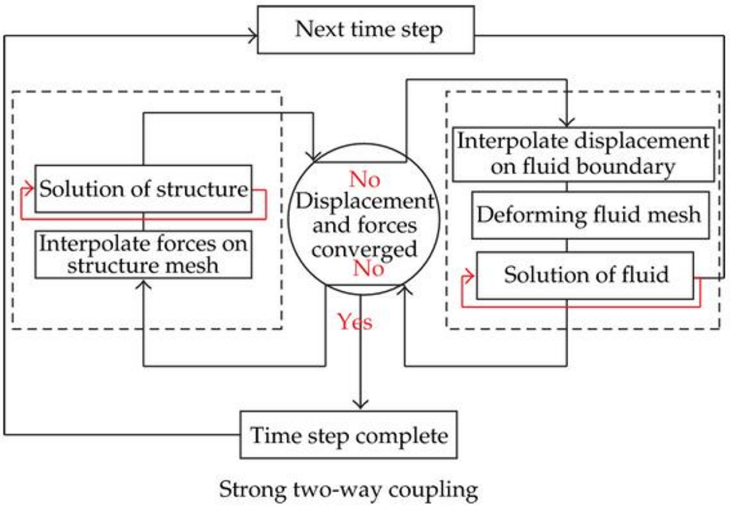
The efficiency of the MPM algorithm has also been improved to boost the computational performance of the simulator. This includes an adaptive MPM scheme to optimize computation resources in the area of interest (Gao et al. (2017)), Parallel MPM using GPU with compute shader or CUDA (Dharma et al. (2017),Dharma et al. (2018)), and A Moving Least Squares MPM (MLS-MPM) (Hu, Fang, Ge, Qu, Zhu, Pradhana & Jiang (2018)) which is twice faster than the previous one. Figure 6 illustrates a time step of implicit MPM. In the beginning, the particle quantities (mass, velocity) are transferred to the grid (P2G). Grid operations are then carried out as in the Eulerian method. The particle velocities at the next time step are updated by transferring the calculated grid velocity (G2P) to the particles.

2.3.3 Coupling

Computation of fluid-solid interaction is often challenging because of its strong nonlinearity, especially when large deformation is considered (Wang et al. (2017)). When the fluid and solid are interacting, the information for the solution at the interface or boundary between the solids and fluids is shared between the fluid solver and the solid solver (Benra et al. (2011)). The interaction between the fluid and the solid can be classified as one-way coupling when the solids get their motion from the fluid but not the other way around; two-way coupling is required for fairly massive objects that significantly affect the fluid flow, or for very soft solids that are affected in turn by the fluid; strong coupling occurs when the forces of the solid and the fluid are computed concurrently, and the motion of each directly impacts the others (Bridson (2015*b*)). As shown in Figure 2.6, in the one-way coupling (2.6a), only the fluid pressure acting at the solid interface is transferred to the solid solver. While in two-way-coupling (2.6b), the displacement of the solid is also transferred to the fluid solver (Benra et al. (2011)).



(a) *One way Coupling*



(b) *Two way Coupling*

Figure 2.6: *Algorithm for one- and strong two-way coupling (Benra et al. (2011))*

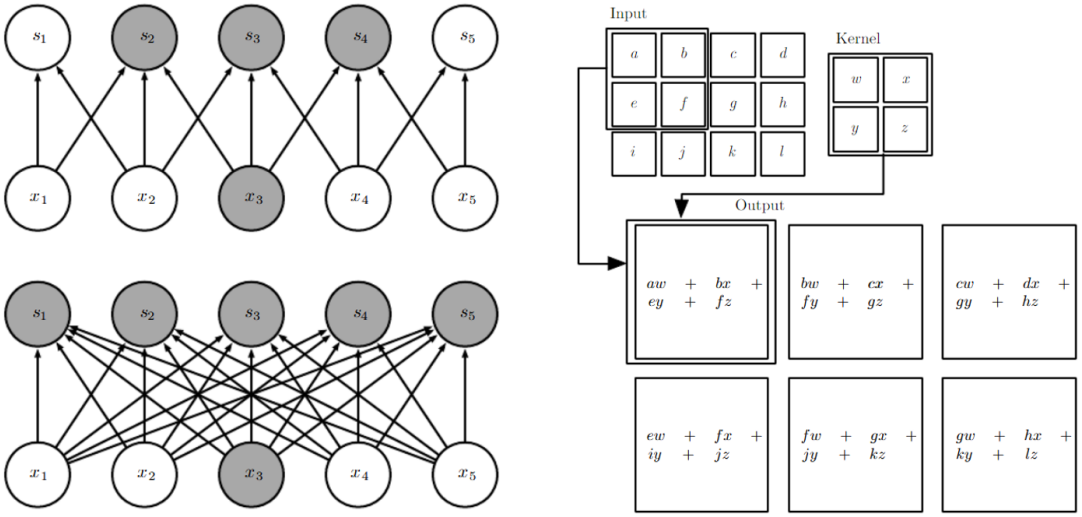
2.4 Deep Learning and Lagrangian Continuum Simulation

Numerous simulations of deformable and fluid fluid material based on physical principles in computer graphics have achieved high levels of realistic visual quality (Qu et al. (2019), Stomakhin et al. (2013), Tampubolon et al. (2017), Jiang et al. (2017)); however, the speed of simulation remains far from real time. For example, simulating cloth and sand with the Anisotropic Elastoplasticity model in (Jiang et al. (2017)) takes 876 seconds to compute 7.23 million particles per simulation frame. Despite advances in parallel computing and more efficient algorithms for solving governing equations in recent years, real-time simulations of fluids and deformable solids have only been feasible under restricted conditions.

The rise of Machine Learning Techniques, particularly Deep Neural Networks (DNN), is currently propelling research into enhancing physical simulation and modeling physical functions. Applications cover a range of areas such as rapid approximations for numerical fluid solvers (Ladický et al. (2015), Tompson et al. (2017)), robotic control and planning (Schenck & Fox (2018), Hu, Liu, Tenenbaum, Freeman, Wu, Rus & Matusik (2018)), and more efficient cloth simulation (Lee et al. (2019), Tan et al. (2019)).

Originally dubbed the Deep feedforward network, the DNN employs a framework loosely inspired by neuroscience, hence the term neural. These networks are generally composed of a combination of various functions. The primary objective of a DNN is to approximate certain functions (Goodfellow et al. (2016)).

Figure 2.7 presents a depiction of various neural networks. In Figure 2.7a, the sparse connectivity of two different neural network types is compared. The top part of 2.7a depicts a convolutional neural network, while the bottom part displays a multi-layer perceptron. A highlighted unit, $x3$, serves as an input example, and the highlighted units in s are the outputs influenced by this input. According to the top of Figure 2.7a, only three outputs are influenced by x when s is generated by convolution (with a kernel width of 3). Conversely, when s is produced via matrix multiplication (bottom of Figure 2.7b), the connectivity is not sparse, resulting in all outputs being influenced by $x3$ (Goodfellow et al. (2016)). According to (Goodfellow et al. (2016)), convolutional networks are a type of neural network that utilize a convolution operation instead of matrix multiplication in at least one layer. Figure 2.7b illustrates a 2-D convolution process. By applying the kernel to the top-left part of the input, the corresponding top-left element of the output tensor is produced. In sequence modeling,



(a) (b)

Figure 2.7: *Neural Network: (a) Comparison of Sparse Connectivity between Convolutional Neural Network (top) and Fully connected network (bottom); (b)(Example of 2D Convolution (Goodfellow et al. (2016))*

which deals with sequential data, there exists a category of neural networks known as recurrent neural networks (RNNs). An RNN is designed to handle sequences of values $x^1, ..., x^t$ in a manner similar to how convolutional networks manage grid values like those in an image (Goodfellow et al. (2016)).

Gated RNNs are examples of other types of neural networks that are recognized for their efficiency in sequence modeling (Goodfellow et al. (2016)). These networks include long-short-term memory (LSTM) units and are built upon the gated recurrent unit framework. The concept of LSTM was initially introduced by (Hochreiter & Schmidhuber (1997*a*)). The core idea involves self-loops to establish routes that allow gradients to propagate for extended periods. Figure 2.8 illustrates this concept. Cells are recurrently connected, replacing traditional hidden units. An input is processed using a standard neuron, and its value can be stored in the state if permitted by the sigmoidal input gate. The state unit features a linear self-loop regulated by the forget gate's weight. The cell output can be disabled through the output gate.

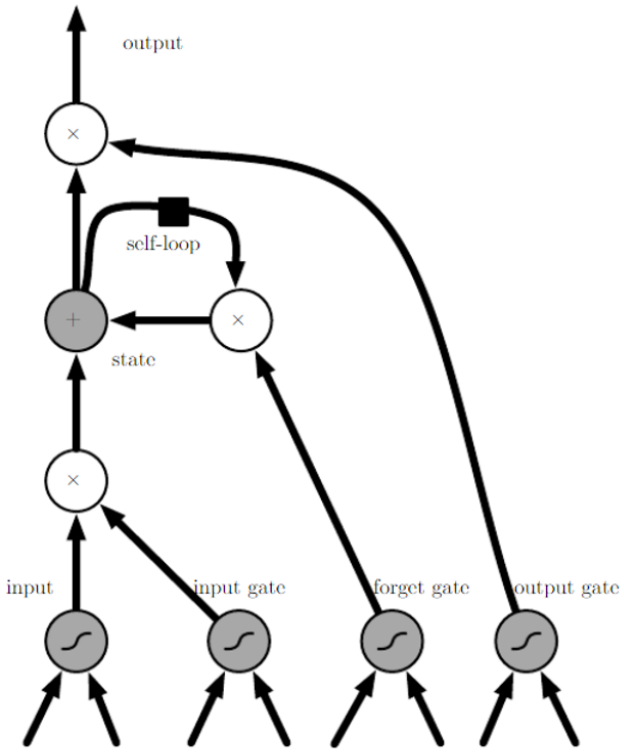


Figure 2.8: *Block diagram of a cell of LSTM (Long short term memory) recurrent network (Goodfellow et al. (2016))*

2.4.1 Deep Learning and fluids

Recently, a fairly comprehensive study of machine learning in fluid mechanics in the literature was published by (Brunton et al. (2020)). In this summary, we will highlight some recent work which is mostly related to computer graphics, especially in interactive applications. Ladicky et al. (Ladický et al. (2015)) and Yang et al. (Yang et al. (2016)) first time proposed real-time fluid simulation using a data-driven approach. They used a supervised learning perspective to train a black-box machine learning system to predict the output of a deterministic fluid simulator. (Ladický et al. (2015)) used the Random Regression Forest to predict the output produced by a Lagrangian Solver, SPH, while (Yang et al. (2016)) used a patch-based neural network to predict ground truth pressure. Since the methods are entirely dependent on a dataset provided by a conventional solver, their method suffers from the generalization

problem, which cannot extrapolate the model far outside the data observed during training.

To improve the generalization properties of the accelerated fluid model, (Tompson et al. (2017)) proposed the unsupervised training-loss method to accelerate the smoke simulation by inferring the PCG (Poisson conjugate gradient) part of the calculation step in the Eulerian Solver with Convolutional Network (Helfenstein & Koko (2012)). The proposed method still uses the advantage of the understanding and modeling power of classical approaches. While the proposed approach cannot ensure an exact solution to the pressure projection step, its runtime and accuracy are better than the Jacobi method (Black et al. (2024)).

There is also increasing interest in differentiable physics simulation (Schenck & Fox (2018), Hu, Liu, Tenenbaum, Freeman, Wu, Rus & Matusik (2018)), to efficiently solve inverse problems such as optimal control and motion planning, as it can be incorporated into gradient-based optimization algorithms. It also can be combined with other learning methods to provide physically consistent predictions. (Schenck & Fox (2018)) proposes an embedded Position-Based Fluid (PBF) (Macklin & Müller (2013)) solver in convolutional neural networks called SPNets. It adds two new layers to the neural network toolbox, which enable physical computing interactions with unordered particle sets. These layers are then used in combination with standard neural network layers to directly implement fluid dynamics inside a deep network, where the parameters of the network are the fluid parameters themselves (e.g., viscosity, cohesion, etc.). The generated models of fluid dynamics are fully differentiable since it is implemented as a neural network. The solution demonstrates the ability to learn fluid parameters from data, perform liquid control tasks, and learn policies to manipulate liquids.

While SPNets focuses on a fluid manipulation task in robotics, (Hu, Liu, Tenenbaum, Freeman, Wu, Rus & Matusik (2018)) proposes a differentiable Moving Least Squares-Material Point Method (MPM) simulator for deformable solids (soft robots). The system stores the object data at every simulation step so that the gradient can be calculated out of the box. The solution can simulate deformable objects including contact and can be seamlessly incorporated into inference, control and co-design systems. Both forward simulation and backward gradient can be computed precisely.

Figure 2.9 illustrates a time step of MLS-MPM. At time t^n, the particle transferred to the grid (P2G). Grid operations are then carried out as in the Eulerian method. The particle speed at $t^(n+1)$ is then updated trough transfer by transferring the calculated grid velocity (G2P). All data in P2G, Grid op, and G2P will be stored in caches that will be used later for

Gradient calculation in the backward process.

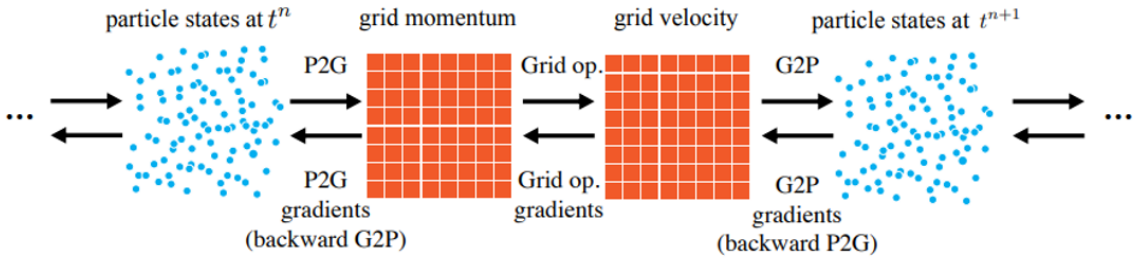


Figure 2.9: *An illustration of a single time step of MLS-MPM. With for forward simulation (top arrows) and backpropagation process (bottom arrows) Hu, Liu, Tenenbaum, Freeman, Wu, Rus & Matusik (2018)*

Temporal modeling in fluid simulation has been actively explored recently. Autoencoders (AE) are used by (Xie et al. (2019), Kim et al. (2019), Wiewel et al. (2020)) to compress the dimensionality of the simulation into latent space prior to temporal prediction. (Xie et al. (2019)), in the separated training process from the AE, uses an LSTM network in latent space to predict the temporal change of the fluid. This approach leads to large speed-ups (despite the not "apple-to-apple" comparison, CPU-based traditional solver with the GPU-based model predictor). (Kim et al. (2019)) uses a generative approach with CNN to re-synthesize / upscale the dynamic flow fields for both smoke and liquid. This simulation is attractive for re-simulation scenarios in a specified container where input interactions can be parameterized.

2.4.2 Deep Learning and deformable solids

Most previous research in reducing the computational cost for simulation based on physical evidence using DNN has focused on fluid simulation. More recently, a study of the DNN model for cloth simulation has been considered (Lee et al. (2019), Laehner et al. (2018), Holden et al. (2019), Tan et al. (2019)). (Holden et al. (2019)) proposes a simulation of a physical cloth character driven by data that meets the requirements of computation and memory. As illustrated in Figure 2.10, (Holden et al. (2019)) combines subspace simulation techniques, PCA (Principal Component Analysis, which emphasizes variation and reveals strong patterns in a dataset, compressed representation), with machine learning which enables efficient simulation of subspace-only physics that supports interactions with external objects.

It acquires training data X and Y offline using a cloth simulator and reduced the representation Z and W by performing PCA, before the neural network training process.

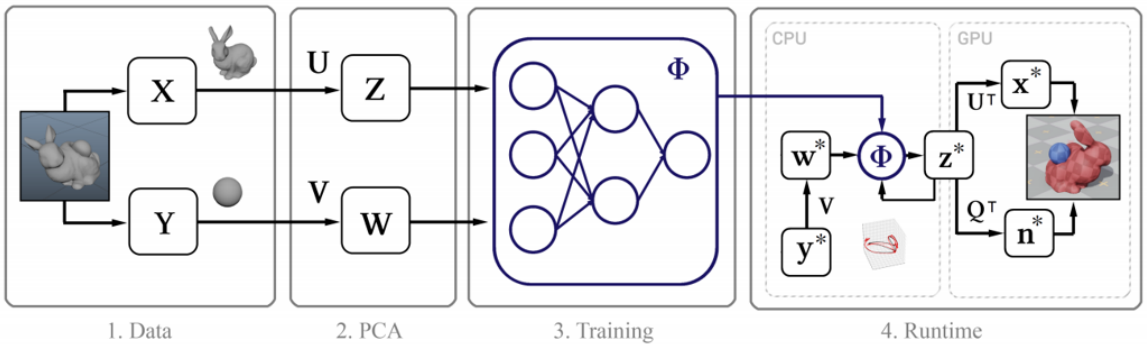


Figure 2.10: *An illustration of a single time step of MLS-MPM. With for forward simulation (top arrows) and backpropagation process (bottom arrows) Holden et al. (2019)*

2.4.3 Deep Learning and Vorticity

Recent advancements in deep learning have introduced data-driven approaches using Lagrangian representations to improve the accuracy of fluid predictions, specifically targeting velocity and linear momentum prediction. However, these methods do not explicitly address angular momentum conservation, leading to the absence of a direct metric for vorticity in Lagrangian fluid simulations.

For instance, the framework proposed by Sanchez-Gonzalez et al. (2020), termed "Graph Network-based Simulators" (GNS), represents the state of a physical system using particles as nodes in a graph and computes dynamics via learned message-passing. Their results demonstrate the model's ability to generalize from single-timestep predictions with thousands of particles during training to different initial conditions and thousands of timesteps at test time. However, while the model mitigates error accumulation by corrupting training data with noise, it does not address vorticity explicitly.

Similarly, Ummenhofer et al. (2020) propose a method that uses continuous convolutions to process sets of moving particles describing fluids in space and time without building an explicit graph structure. This method generalizes to arbitrary collision geometries and can

be used for inverse problems. However, it also lacks an explicit mechanism for vorticity conservation.

Prantl et al. (2022) take a step further by guaranteeing the conservation of linear momentum in learned physics simulations through antisymmetrical continuous convolutional layers. While their method substantially increases the physical accuracy of the learned simulator, it focuses on linear momentum conservation and does not provide explicit measures for vorticity.

On the other hand, approaches aiming to address vorticity often operate within Eulerian frameworks or on regular grid data or images, rather than particle representations in continuous space. For example, Guan et al. (2022) introduce "NeuroFluid," a method that infers fluid state transitions from sequential visual observations of the fluid surface. Their approach involves a particle-driven neural renderer and a particle transition model optimized to reduce discrepancies between rendered and observed images, though it still lacks a direct focus on vorticity.

Chu et al. (2022) propose a method to reconstruct dynamic fluid using physics-informed neural fields, leveraging the governing physics like the Navier-Stokes equations on regularly spaced grids. This method optimizes fluid dynamics from sparse video data but does not explicitly conserve vorticity.

Xiong et al. (2023) introduce the "Neural Vortex Method," combining the Discrete Vortex Method with neural networks. This method relies on ground truth velocity sequences of vortex particles to represent fluid dynamics, which is a step towards integrating vorticity into neural models.

Finally, Deng et al. (2023) propose a method that learns vortex particle dynamics from regular grid data using single videos without needing ground truth velocity. Their approach, termed the "Differentiable Vortex Particle (DVP) Method," provides a novel way to infer and predict fluid dynamics, including vorticity, from limited observational data.

These methods demonstrate both the gaps and the recent trend towards incorporating vorticity into deep learning models for fluid dynamics, but challenges remain in effectively integrating this with Lagrangian representations.

2.4.4 Graph Network

Graph networks offer a powerful and flexible *overall framework* for modeling physical dynamics, including fluid simulations. In this approach, particles are treated as nodes, while edges

encode the interactions among them. This makes it straightforward to incorporate a variety of graph neural network (GNN) convolutional layers, each bringing its own strengths to the prediction of complex fluid behaviors. Below, we highlight several noteworthy GNN variants that can be plugged into such a framework.

- **Handling Irregular Geometries:** Unlike traditional grid-based methods, graph networks are not constrained by regular grids and can accommodate irregular and dynamic geometries. This flexibility is critical for simulating real-world fluid systems where particle distributions are often non-uniform and change over time.

- **Rotational Invariance:** Graph representations inherently maintain rotational invariance, a crucial property in fluid dynamics where the relative positions and interactions of particles matter more than their absolute positions. This ensures that simulation results remain consistent regardless of the global orientation of the particle set.

- **Capturing Local Interactions:** Graph networks excel at modeling local interactions among particles. Each node (particle) easily shares information with its neighbors (connected edges), effectively capturing the local dependencies essential for accurate fluid-dynamics modeling.

- **Scalability:** By leveraging sparse connectivity, graph networks can scale efficiently to large particle systems. Only the relevant local interactions need be considered, reducing computational overhead compared to methods requiring exhaustive, global interactions.

To evaluate the performance of our framework, we compare it against multiple GNN convolutional layers that have been used in fluid-like or particle-based simulations. Each represents a distinct approach to learning from graph-structured data:

PointNetConv (PointNet++)

PointNetConv, based on the PointNet++ architecture, extends the original PointNet model with hierarchical feature learning. This extension captures both local and global geometric features from point clouds, making it effective in tasks such as 3D object recognition and segmentation. Its ability to learn multi-scale features also proves advantageous in fluid simulations, where particle distributions can vary significantly Qi et al. (2017).

Graph Transformer

The Graph Transformer architecture integrates the principles of the Transformer model, originally developed for sequence processing, into graph-based data. By leveraging self-attention mechanisms, the Graph Transformer can model long-range dependencies within a graph, making it particularly well-suited for tasks that require capturing global context, such as node and graph classification Shi et al. (2021). Yuan et al. (2022) use Transformer with Implicit Edges (TIE), to capture the rich semantics of particle interactions in an edge-free manner.

GATConv

GATConv is the original Graph Attention Network (GAT) formulation, introducing attention-based mechanisms into GNNs. By computing attention scores between a node and its neighbors, GATConv can preferentially highlight the most relevant particle interactions for fluid simulation. It has served as a foundational model for tasks involving graph-structured data Veličković et al. (2018).

GATv2Conv

GATv2Conv is an enhanced version of the Graph Attention Network (GAT) architecture. It refines the attention mechanism to improve the model's expressive power and stability. By dynamically weighting neighboring nodes' contributions, GATv2Conv excels in capturing complex relationships within a fluid system, focusing on the most critical interactions for accurate predictive performance Brody et al. (2022).

2.4.5 Activation Functions

Activation functions are essential in the construction of neural networks, significantly influencing the network's capacity to learn and provide accurate predictions. This subsection examines the frequently employed activation functions in neural network arhictecture discussed in this thesis, with a focus on Sigmoid, Tanh, ReLU, and ELU.

Sigmoid The sigmoid activation function, also known as the logistic function, is one of the earliest and most widely used activation functions in neural networks. It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

The sigmoid function maps input values to a range between 0 and 1, making it suitable for binary classification problems. Its smooth gradient is advantageous for backpropagation, ensuring that the gradient descent process is stable. However, the sigmoid function suffers from the vanishing gradient problem, where gradients can become very small, slowing down the learning process. This issue is particularly problematic for deep networks, as it hampers their ability to learn efficiently.

Tanh The hyperbolic tangent function (tanh) is another commonly used activation function, defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.6}$$

The tanh function maps input values to a range between -1 and 1. Compared to the sigmoid function, tanh is zero centered, which can help centering the data and thus lead to a more efficient training process. The gradients of the tanh function are steeper than those of the sigmoid, which helps mitigate the vanishing-gradient problem to some extent. However, tanh is not immune to the issue of the vanishing gradient and can still slow down the training of very deep networks.

ReLU The Rectified Linear Unit (ReLU) is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2.7}$$

ReLU has become the default activation function for many neural networks because of its simplicity and effectiveness. It introduces non-linearity by outputting zero for any negative input and the input value itself for any positive input. ReLU is computationally efficient and helps mitigate the problem of vanishing gradients, as it does not saturate positive values. However, ReLU can suffer from the "dying ReLU" problem, where neurons can get stuck during training, always outputting zero. This issue arises when the learning rate is too high or due to poor weight initialization.

ELU The Exponential Linear Unit (ELU) is defined as

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{2.8}$$

where α is a hyperparameter that controls the value to which an ELU saturates for negative net inputs. ELU improves upon ReLU by allowing negative values when the input is less than zero, which helps to bring the mean activation closer to zero. This zero-centered nature can speed up learning. ELUs tend to mitigate the "dying ReLU" problem and maintain a stronger gradient during training. The choice of the α parameter can affect performance and should be carefully selected based on the specific task.

Chapter 3

Methodology

This chapter will discuss methods and techniques used in the work that follows, including the analysis of time complexity of deterministic / conventional simulators, the generation of fluid data sets, spatial and temporal learning analysis, energy conservation analysis, design of network architecture, training procedures, and evaluation methods.

3.1 Analysis of Time Complexity of Deterministic Simulator

The Material Point Method (MPM) algorithm described in Chapter 2 is used to generate training data for simulations involving fluids or deformable solids. Each subsequent frame is forecast from the preceding frame by performing several sub-iterations. An experiment was carried out to determine the required number of sub-steps to achieve different levels of simulation quality (by increasing the number of particles and grid resolution). The findings of this experiment are shown in Table 3.1, which details how simulation resolution relates to computational time when employing GPU-based parallelism (via CUDA in Taichi Lang). For instance, in a simulation with 26 k particles (row 4 in Table 3.1), the per-frame time exceeds 70 ms, yielding a simulation frame rate of around 13.58 fps. This drop occurs because a higher number of sub-steps is required to preserve accuracy at larger scales, making the simulation impractical for interactive use.

Parameter q and the Timing Setup. In Table 3.1, the parameter q is a *quality scaling factor* that controls both the number of particles and the MPM grid resolution. Specifically, the number of particles is proportional to q^2, and the grid dimensions (e.g., width and height)

Table 3.1: *MPM Computation Time using Parallelism on GPU in various configurations*

Quality			MPM Grid size			MPM Simulation Time (ms)			
q	q^2	Particles	Width	Height	Nodes	substeps/f	t/substep	t/frame	fps
1	1	16,384	128	128	16,384	20	0.06	1.17	854.70
2	4	65,536	256	256	65,536	20	0.14	5.33	187.62
3	9	147,456	384	384	147,456	40	0.51	41.46	24.12
4	16	262,144	512	512	262,144	40	0.73	73.66	13.58
5	25	589,824	768	768	589,824	80	0.85	136.09	7.348
6	36	819,024	896	896	801,216	80	1.14	180.05	5.55
7	49	1,024,576	1024	1024	1,048,576	120	1.60	238.69	4.19
8	64	1,048,576	1024	1024	1,048,576	160	2.25	361.89	2.76
9	81	1,324,104	1152	1152	1,324,104	180	3.08	562.43	1.78

likewise scale with q. Consequently, when q increases, the simulation employs a higher grid resolution and more particles, thus necessitating more sub-steps per frame to maintain accuracy. Each row of the table represents a distinct two-dimensional fluid or deformable-solid simulation scenario with incrementally larger problem sizes and correspondingly greater computational demands. The "time per frame" (t_{frame}) column then captures how the parallelized MPM implementation scales as q grows.

To calculate the time complexity $\mathcal{O}(n)$ from the relationship between q^2 and the column t_{frame} in the table, we examine how t_{frame} scales with q^2. The slope m of a log-log plot can be found by linear regression on the log-transformed data:

$$\log_{10}(t_{\text{frame}}) = m \log_{10}(q^2) + \log_{10}(k). \tag{3.1}$$

q^2	$\log_{10}(q^2)$	$\log_{10}(t_{\text{frame}})$
1	0.000	0.068
4	0.602	0.727
9	0.954	1.617
16	1.204	1.868
25	1.398	2.134
36	1.556	2.255
49	1.690	2.378
64	1.806	2.558
81	1.908	2.750

Figure 3.1 illustrates the relationship between $\log_{10}(q^2)$ and $\log_{10}(t_{\text{frame}})$ (Equation 3.1). The best-fit line in Figure 3.1a has a slope of approximately 1.41, indicating the exponent in the polynomial relationship between t_{frame} and q^2. Accordingly:

$$t_{\text{frame}} \propto (q^2)^{1.41}. \tag{3.2}$$

Since q^2 is proportional to the number of particles n, the time complexity $\mathcal{O}(n)$ can be approximated as:

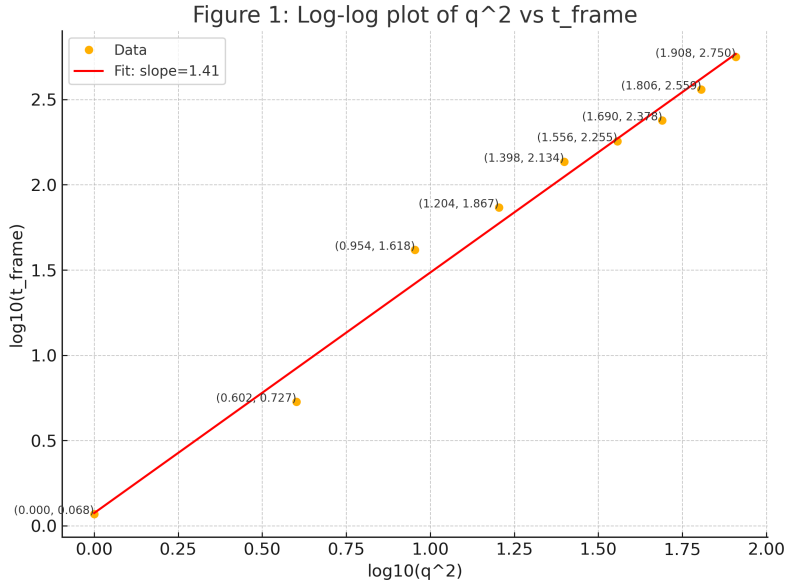
$$\mathcal{O}(n) \approx \mathcal{O}\big(n^{1.41}\big).$$

This implies that the MPM simulation time per frame increases slightly faster than linearly as the number of particles grows.

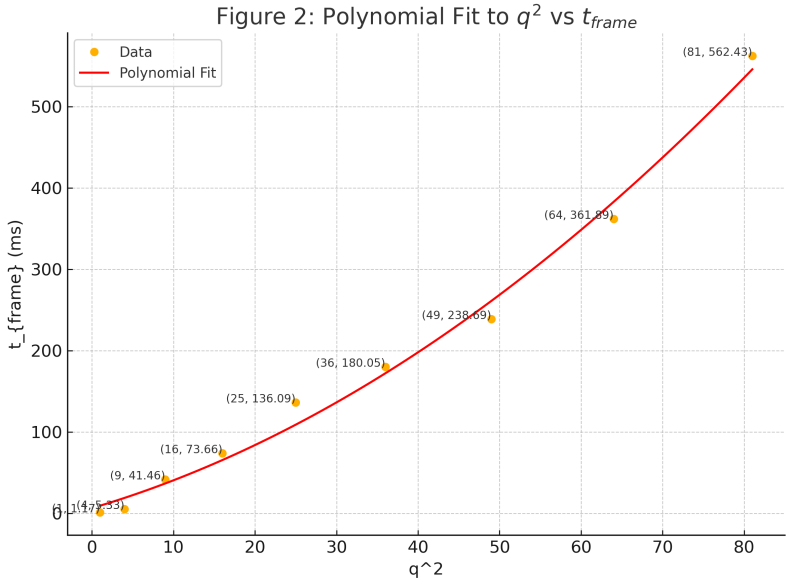
At smaller scales (e.g., a few thousand particles), the GPU-accelerated MPM simulator can be run in real time or near real time. However, as problem size grows to tens or hundreds of thousands of particles, per-frame computation times escalate due to the compounding cost of sub-steps. Once the simulation reaches millions of particles, real-time performance typically becomes infeasible unless significant parallelization or algorithmic optimizations are employed.

3.2 Surrogate Model Efficiency

The efficiency of a surrogate model is a critical aspect of its design and performance, particularly when applied to large-scale Lagrangian fluid simulations. This subsection delves into



(a) ***Log-log plot of*** q^2 ***vs*** t_{frame}. *This plot shows the relationship between the quality parameter* (q^2) *and the simulation time per frame* (t_{frame}) *in milliseconds. The points represent log-transformed data, labeled with their* $\log_{10}(q^2)$ *and* $\log_{10}(t_{frame})$ *values. The red line, with a slope of approximately 1.41, indicates that the time complexity increases slightly faster than linearly with* q^2.



(b) ***Polynomial Fit to*** q^2 ***vs*** t_{frame} ***of the MPM Simulator.***

Figure 3.1: *Plots showing the relationship between the quality parameter* (q^2) *and simulation time per frame* (t_{frame}) *for the MPM Simulator. (b) The red curve represents a second-degree polynomial fit, highlighting the super-linear relationship that reflects the increasing computational demands at larger scales.*

the factors that influence the efficiency of a surrogate model, examining both the theoretical underpinnings and practical considerations.

3.2.1 Training Efficiency

During the training phase of the surrogate model, efficiency is significantly influenced by several factors, including the complexity of the underlying neural network architecture, the choice of optimizer, and the selection of training parameters such as learning rates and batch sizes.

The training data sets consist of sequential frames generated by the Material Point Method (MPM) simulator, capturing the dynamic evolution of fluid and solid particles over multiple simulation sub-steps. The neural network is trained to predict subsequent simulation frames within a latent (lower-dimensional) space, minimizing the need for extensive computational substeps during inference.

Key factors affecting training efficiency include:

Network Parameterization: The number of parameters within the neural network, such as the depth and width of layers, directly impacts the computational resources required during training. A higher number of parameters generally leads to increased computational demands but may also improve model accuracy Goodfellow et al. (2016).

Choice of Optimizer: The selection of an optimizer plays a critical role in the efficiency of training. Common optimizers include Stochastic Gradient Descent (SGD), Adam, and RMSprop, each with its own advantages and trade-offs. For instance, the Adam optimizer is widely used for its adaptive learning rate capabilities, which can lead to faster convergence and improved efficiency, especially in complex models Kingma & Ba (2014). However, the choice of optimizer must be matched with the specific needs of the model, as some optimizers may be more computationally expensive or less effective for certain types of data.

Training Parameters: The efficiency of training is also influenced by the choice of key parameters, such as the learning rate and batch size:

- *Learning Rate:* The learning rate determines the size of the steps the optimizer takes during each iteration. A higher learning rate can speed up training but may lead to instability or overshooting minima. Conversely, a lower learning rate may provide more precise convergence but requires more iterations, thus increasing training time.

- *Batch Size:* The batch size affects how many data samples are processed before the model's parameters are updated. Larger batch sizes can lead to more stable gradient estimates and faster training times, but they also require more memory. Smaller batch sizes, while requiring less memory, may lead to noisier updates and longer training times Keskar et al. (2017).

Data Dimensionality Reduction: The use of dimensionality reduction techniques, such as encoding the particle states into a latent space, can reduce the computational burden during training. This approach allows the network to focus on the most salient features of the simulation data, thereby improving both training speed and generalization capabilities Goodfellow et al. (2016).

By carefully selecting the optimizer and tuning the learning rate and batch size, along with considering network parameterization and dimensionality reduction, the training efficiency of the surrogate model can be significantly enhanced. This balance is crucial to ensure that the model not only trains efficiently, but also generalizes well to unseen data.

3.2.2 Inference Efficiency

The surrogate model is designed to enhance efficiency during the inference phase by predicting future simulation frames without the need for multiple computational substeps. This is achieved by **latent space forecasting**, where the model operates within a latent space, predicting the next state of the system based on its current state. This approach significantly reduces computational overhead compared to traditional methods, which require iterative calculations for each substep of the simulation (Chollet (2018)). Furthermore, **linear time complexity** is anticipated due to the structure of the latent space predictions, which means that the surrogate model will likely exhibit linear time complexity as the size of the simulation increases. This contrasts with the **super-linear time complexity typically observed in deterministic simulators** (e.g. MPM) as discussed in Section 3.1, where computation time grows more rapidly with the number of particles (Xie et al. (2019)).

3.2.3 Impact of Network Size on Performance

The performance of the surrogate model is closely tied to the size and complexity of the neural network. A larger network with more layers and neurons may offer improved accuracy in

capturing complex particle interactions, but at the cost of increased computational demands. On the contrary, a smaller network may offer faster predictions but could struggle to maintain accuracy, particularly in simulations involving highly dynamic or chaotic systems (Goodfellow et al. (2016)).

Key considerations include the **trade-off between accuracy and speed**, where the network must balance the need for accurate simulations with the requirement for real-time or near-real-time performance. Achieving this balance often involves careful tuning of hyperparameters, such as the number of layers, neurons per layer, and the choice of activation functions (Goodfellow et al. (2016)). Additionally, **scalability** is a critical factor, since the ability of the surrogate model to scale with increasing simulation sizes must ensure that its efficiency and accuracy are maintained as the number of particles in the simulation grows, thus ensuring viability for large-scale simulations (Goodfellow et al. (2016)).

3.3 Dataset Generation

To generate a high-quality dataset for training purposes, we employ the Material Point Method (MPM) (Sulsky et al. (1994), Jiang et al. (2016)) based simulator. The MPM is a powerful computational technique for simulating the behavior of materials under various physical conditions, making it particularly suitable for fluid dynamics and other complex material simulations.

For implementing the MPM, we use the Taichi programming language (Hu et al. (2019)). Taichi is a domain-specific language designed for high-performance computational tasks, particularly in computer graphics and physics simulations. It leverages a just-in-time (JIT) compiler to efficiently map computationally intensive operations to multicore CPUs and highly parallel GPUs, ensuring significant performance improvements in simulation tasks.

One of the key advantages of using Taichi is its deep integration with Python. This integration simplifies the process of incorporating Taichi-based simulations into workspaces that utilize common Deep Learning frameworks, many of which are Python-based. This compatibility allows for a seamless workflow when embedding simulation data into machine learning pipelines, facilitating the training of neural networks with realistic, high-fidelity datasets.

Our dataset generation process is centered around the creation of scenarios. Each scenario begins with an initial configuration of the material or fluid, which is then evolved over time

using the MPM simulator. This evolution is captured in a sequence of frames, each representing a state of the system at a particular time step. For supervised learning tasks, pairs of consecutive frames from these sequences can be used as training data. Each frame provides a snapshot of the system's state, while the subsequent frame captures the resulting state after a time step, forming a natural input-output pair for training predictive models.

This approach not only provides a rich set of training data but also captures the temporal dynamics of the system, making it highly relevant for tasks that require an understanding of time-dependent behaviors. The ability to generate such data in a controlled and precise manner is critical for developing models that can generalize well to real-world scenarios.

3.4 General Training Procedure

In this project, the training was conducted in a supervised fashion. Supervised learning involves using labeled datasets, where each input data point is paired with the corresponding output label. The model learns to map input to output by minimizing the error between the predicted output and the actual labels.

The training loss function used in this experiment is the Mean Squared Error (MSE). The MSE is a common loss function for regression tasks, defined as the average of the squares of errors between predicted and actual values. Minimizing the MSE helps the model make accurate predictions by reducing the overall error.

The framework used to implement the training procedure depends on the specific experiment being performed. We utilized both TensorFlow and PyTorch, two of the most popular deep learning frameworks. TensorFlow and PyTorch offer versatile and effective tools for constructing and training Neural Networks. The selection of the framework depends on the specific needs of the experiment and compatibility considerations with preexisting methods.

Listing 3.1: TensorFlow example

```
import tensorflow as tf
model = tf.keras.models.Sequential([...])
model.compile(optimizer='adam', loss='mse', metrics=['mse'])
model.fit(x_train, y_train, epochs=100, batch_size=32)
```

Listing 3.2: PyTorch example

```
import torch
import torch.nn as nn
model = nn.Sequential([...])
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
for epoch in range(100):
    optimizer.zero_grad()
    outputs = model(x_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()
```

3.5 Evaluation Method

For the quantitative analysis, we use several metrics to evaluate the performance of our model, including the mean squared error (MSE), the mean absolute error (MAE) and an energy conservation plot. To provide a visual understanding of the model's performance, we include several qualitative assessments (Predicted vs Actual Vorticity Plot, Error Plot, Temporal Evolution of Vorticity).

Quantitative Metrics We assess each technique using Mean Squared Error (MSE) and Mean Absolute Error (MAE), providing a numerical evaluation of prediction accuracy. These metrics measure how closely the predicted velocity and vorticity values match the actual measurements across different test scenarios. Our model's MSE and MAE are compared with those of other techniques, demonstrating our model's accuracy in relative to these.

Generalizability: We further evaluate the generalizability of each model by examining their performance on unseen data. Our model's ability to accurately predict velocity and vorticity in previously unseen cases indicates a degreen of generalizability compared to other techniques that might overfit specific training conditions.

Mean Squared Error (MSE) The Mean Squared Error is a measure of the average of the squares of errors. It is used to quantify the difference between the predicted and actual values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.3}$$

Mean Absolute Error (MAE) The Mean Absolute Error measures the average magnitude of the errors in a set of predictions without considering their direction. It is the average over the test sample of the absolute differences between the prediction and the actual observation, where all individual differences have the same weight.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.4}$$

Energy Conservation: We include an energy conservation plot to compare how well each model maintains adherence to the physical principle of energy conservation in fluid dynamics. Our model's performance in maintaining energy consistency underscores its alignment with fundamental physical laws, distinguishing it from other technique's degree of conserve energy.

Vorticity Distribution Evaluating the distribution of vorticity is crucial for understanding fluid dynamics in simulations. Different methods can be employed to compare the predicted vorticity distribution from various models against the ground truth. We use three primary evaluation methods: Pearson Correlation (Dowdy et al. (2004), Sul et al. (2014)), Residual Analysis (Hoaglin (2003), Brady (2018)), and Kullback-Leibler (KL) Divergence (Buhl (2023)). Each evaluation method offers unique insights into the similarity between predicted and ground truth vorticity distributions. Pearson Correlation provides a simple linear measure, residual analysis offers a visual error representation, and KL Divergence captures complex distribution differences. However, careful consideration of normalization is essential when using KL Divergence to ensure meaningful and accurate comparisons.

Qualitative Analysis: Beyond numerical metrics, we offer qualitative comparisons through visual assessments such as Predicted vs Actual Vorticity Plot and Temporal Evolution of Vorticity. These visuals illustrate how well each model captures the complex flow dynamics, velocity and vorticity patterns over time, with our model showing fidelity in preserving key flow characteristics.

Velocity and Vorticity Field Plot This plot shows a comparison between the predicted and actual vorticity values. The fluid domain is discretized into a certain number of grid cells, and the particles' velocities are interpolated to the nearest grid nodes to calculate the vorticity magnitude using finite differences. The values are colored using a specific scale.

Error Plot The error plot highlights the difference between actual ground truth values and those predicted by the model. A reliable surrogate model, as illustrated by the error plot, will exhibit errors that are both low in magnitude and uniformly small across various data points, demonstrating its precision. The errors should be randomly dispersed around the horizontal zero axis, with no visible patterns or systematic trends, indicating the model's lack of significant biases and its competency under different conditions. Symmetrical distribution

of errors around zero suggests that the model's predictions are balanced, neither consistently overestimating nor underestimating. Furthermore, the errors should show minimal variability, which reinforces the model's reliability and consistency. The lack of outliers in the error plot signifies that the model manages all data aspects effectively without notable deviations. Additionally, the errors should not show a correlation with the input values' magnitude, indicating that the model scales appropriately without introducing proportional errors. In essence, a robust model's error plot should feature small, random, and evenly spread errors, reflecting dependable and generalizable performance.

By integrating these comparative analyses, we provide a holistic view of the strengths and limitations of our model relative to other established methods. The combination of quantitative performance metrics and qualitative fidelity highlights the ability of our model to deliver accurate, efficient, and reliable predictions in fluid simulations.

Chapter 4

Temporal Learning of Continuum Simulation

To model Lagrangian fluid and deformable solid simulations, it is crucial to understand how to effectively utilize velocity, position, material properties, and other attributes derived from Euclidean coordinate-based continuum data and encode them into a latent space. This chapter explores the essential temporal modeling techniques applied to these Lagrangian simulations, focusing on how these physical attributes evolve over time.

Initially, we investigate the potential of a baseline architecture using a Multi-Layer Perceptron (MLP / DNN). The MLP serves as a foundational model to capture the basic temporal relationships within the data, offering a simple yet effective approach to understanding the temporal dynamics in Lagrangian simulations.

In subsequent experiments, we shift our focus to more advanced neural networks, particularly Long Short-Term Memory (LSTM) networks. LSTMs are well-suited for tasks involving time series data, as they are designed to capture long-term dependencies and temporal patterns. By employing LSTM networks, we aim to enhance the predictive capabilities of our models, allowing them to more accurately forecast the evolution of fluid and solid states over time.

To efficiently conduct these experiments, we employ an iterative prototyping approach using basic 2D fluid data, as recommended by Bridson Bridson (2015*a*) for fluid simulation prototyping. This methodology allows for rapid adjustments and refinements to the models, accelerates network training, and facilitates quicker qualitative evaluations.

4.1 Encoding and Decoding to Latent Space Representations

Effective encoding and decoding of latent space representations is crucial to the success of neural network-based surrogate models. This subsection delves into techniques for transforming Euclidean coordinates-based continuum data into latent space and reconstructing the original data from these latent representations.

4.1.1 Encoding Continuum Particles Using MLP

To accurately model Lagrangian fluid and deformable solid simulations, it is essential to encode the attributes of the continuum particles, such as position, velocity, and material properties, into a latent space. A Multi-Layer Perceptron (MLP) is a suitable choice for this encoding task because of its ability to learn complex mappings from input to output space.

Consider a set of continuum particles in a 2D simulation, each characterized by their position (x, y), velocity (v_x, v_y), and material properties $\mathbf{m}$ (e.g., coefficients such as density or elasticity). The goal is to encode these attributes into a latent vector $\mathbf{z}$, which serves as a compressed representation of the state of the particle.

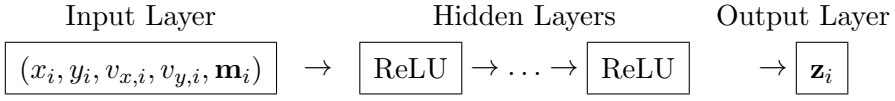
The MLP used for encoding can be described by the following notation: The input is $\mathbf{p}_i = (x_i, y_i, v_{x,i}, v_{y,i}, \mathbf{m}_i)$ for particle i, the output is $\mathbf{z}_i$ (latent vector for particle i), and the architecture consists of L layers with varying numbers of neurons and activation functions. The encoding process is modeled as follows:

$$\mathbf{z}_i = \text{MLP}(\mathbf{p}_i; \theta), \tag{4.1}$$

where θ represents the learnable parameters of the MLP.

The MLP architecture for encoding consists of an input layer, several hidden layers, and an output layer. Each hidden layer is typically followed by a non-linear activation function, such as ReLU. The input layer takes the concatenated particle attributes $\mathbf{p}_i$, the hidden layers form a sequence of layers with neurons and activation functions, and the output layer produces the latent vector $\mathbf{z}_i$.

Below is a schematic diagram of the MLP architecture for encoding continuum particles:



4.1.2 Decoding Latent Space Representations

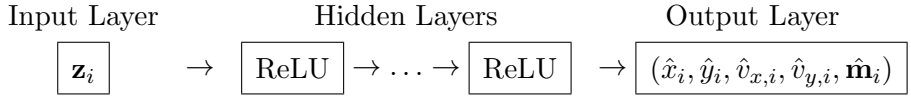
Decoding is the process of reconstructing the original attributes of the particles from their latent space representations. This is achieved by using a decoder network, typically another MLP, which takes the latent vector $\mathbf{z}_i$ as input and outputs the reconstructed particle attributes $\hat{\mathbf{p}}_i = (\hat{x}_i, \hat{y}_i, \hat{v}_{x,i}, \hat{v}_{y,i}, \hat{\mathbf{m}}_i)$. The decoding process is modeled as:

$$\hat{\mathbf{p}}_i = \text{MLP}_{\text{dec}}(\mathbf{z}_i; \phi), \tag{4.2}$$

where ϕ represents the learnable parameters of the MLP decoder.

The architecture of the MLP decoder mirrors that of the encoder, with the input being the latent vector $\mathbf{z}_i$ and the output being the reconstructed particle attributes. The input layer takes the latent vector $\mathbf{z}_i$, followed by a sequence of hidden layers with neurons and activation functions, and the output layer produces the reconstructed particle attributes $\hat{\mathbf{p}}_i$.

Below is a schematic diagram of the MLP architecture for decoding latent space representations.



4.1.3 Training the Encoder-Decoder Model

The encoder-decoder model is trained end-to-end using backpropagation through time (BPTT). This involves computing the gradient of the loss function with respect to the model's parameters and updating those parameters to minimize the loss. The loss function measures the discrepancy between the original particle attributes $\mathbf{p}_i^t$ and the reconstructed attributes $\hat{\mathbf{p}}_i^t$ at each time step t. The objective function, typically based on Mean Squared Error (MSE), is given by:

$$\mathcal{L} = \frac{1}{NT} \sum_{t=1}^{T} \sum_{i=1}^{N} \|\mathbf{p}_i^t - \hat{\mathbf{p}}_i^t\|^2, \tag{4.3}$$

where N is the number of particles, and T is the number of time steps in the sequence.

Optimization is performed using gradient-based methods such as Stochastic Gradient Descent (SGD) or its adaptive variants like Adam. During training, the gradients of the loss function with respect to the encoder parameters θ and decoder parameters ϕ are computed

through backpropagation through time (BPTT). These gradients are then used to update the parameters iteratively:

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}, \tag{4.4}$$

$$\phi \leftarrow \phi - \eta \frac{\partial \mathcal{L}}{\partial \phi}, \tag{4.5}$$

where η is the learning rate, a hyperparameter that controls the step size during optimization.

The choice of optimization algorithm and learning rate significantly influences the model's convergence and the quality of the learned representations. Learning rate schedules or adaptive learning rates are often employed to improve convergence and prevent the model from getting stuck in local minima. Additionally, regularization techniques such as dropout or weight decay may be applied to prevent overfitting and ensure better generalization to unseen data.

4.2 Temporal Learning Analysis

Temporal learning is a critical aspect of modeling dynamic systems like Lagrangian fluid and deformable solid simulations, where the state of the system evolves over time. The ability to accurately predict the future state of the system based on its current and past states is essential for creating reliable and efficient simulations. Temporal learning involves understanding and modeling the temporal dependencies and interactions within the data, allowing for the prediction of future states with high accuracy.

In the context of Lagrangian simulations, temporal learning must handle the continuous and often complex interactions between particles, which are influenced by physical properties like velocity, pressure, and material constraints. These interactions are inherently non-linear and can involve long-term dependencies, where the effect of a particular state can influence the system far into the future. Therefore, sophisticated temporal modeling techniques are necessary to capture these dynamics effectively.

Temporal modeling in fluid simulation has been actively explored recently. Autoencoders (AE) are used by (Xie et al. (2019), Kim et al. (2019), Wiewel et al. (2020)) to compress the dimensionality of the simulation into latent space prior to temporal prediction. (Xie et al.

(2019)), in the separated training process from the AE, uses an LSTM network in latent space to predict the temporal change of smoke. This approach leads to large speed-ups (despite the not "apple-to-apple" comparison, CPU-based traditional solver with the GPU-based model predictor). (Kim et al. (2019)) uses a generative approach with CNN to re-synthesize / upscale the dynamic flow fields for both smoke and liquid. This simulation is attractive for re-simulation scenarios in a specified container where input interactions can be parameterized.

This section explores various temporal learning techniques, emphasizing the importance of capturing both short-term and long-term dependencies in the data. We begin by introducing traditional methods for time series prediction and gradually move toward advanced deep learning architectures, specifically focusing on Long Short-Term Memory (LSTM) networks, which have proven to be highly effective in modeling temporal sequences.

4.2.1 Traditional Time Series Prediction Methods

Rethinking the conventional approaches to predicting time series. These approaches have been extensively used in numerous domains, such as finance, meteorology, and engineering, and offer a solid basis for grasping temporal relationships.

Autoregressive (AR) Models (Yule (1927)) predict the future value of a variable as a linear combination of its past values. The AR model is defined as:

$$y_t = c + \sum_{i=1}^{p} \phi_i y_{t-i} + \epsilon_t, \tag{4.6}$$

where y_t is the value at time t, c is a constant, ϕ_i are the parameters of the model, and ϵ_t is the error term. The order of the model, p, indicates the number of past values used for prediction.

Moving Average (MA) Models The Moving Average model (Kendall & Stuart (1964), Kendall et al. (2007)) is another linear model that predicts the future value based on past error terms. It is defined as:

$$y_t = \mu + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t, \tag{4.7}$$

where μ is the mean of the series, θ_i are the parameters of the model, and ϵ_t is the error term. The order q indicates the number of past error terms used (Brockwell & Davis (2009)).

Autoregressive Integrated Moving Average (ARIMA) Models ARIMA models combine both AR and MA models and also include differencing to make the time series stationary Box et al. (2016). The general form is:

$$y'_t = c + \sum_{i=1}^{p} \phi_i y'_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t, \tag{4.8}$$

where y'_t is the differenced series, making it stationary. ARIMA models are widely used for time series prediction but are limited by their linear nature and the assumption of stationarity.

Limitations of Traditional Methods While these traditional methods are effective for certain types of data, they struggle with non-linearities and long-term dependencies often present in complex simulations like Lagrangian fluid dynamics. They also require careful manual feature engineering and model selection, which can be time-consuming and less adaptive to the evolving nature of the data.

4.2.2 Temporal Learning with Long Short-Term Memory

As the limitations of traditional time series prediction methods became apparent, researchers turned to more sophisticated approaches capable of capturing complex temporal dependencies, particularly in the context of non-linear and non-stationary data. Long Short-Term Memory (LSTM) networks, a type of Recurrent Neural Network (RNN), emerged as a powerful tool for temporal learning due to their ability to retain information over extended sequences and manage long-term dependencies effectively.

LSTM networks Hochreiter & Schmidhuber (1997*b*) were introduced by Hochreiter and Schmidhuber in 1997 as an improvement over traditional RNNs, which suffered from the problem of gradients vanishing and exploding during training. This problem made it difficult for RNNs to learn long-term dependencies in sequential data. LSTMs address this issue by introducing a memory cell and a series of gates that regulate the flow of information, allowing the network to retain important information over long sequences and discard irrelevant data.

The architecture of an LSTM network is composed of a series of memory cells (Figure 4.1), each containing three key components: the input gate, the forget gate, and the output gate. These gates control the flow of information into and out of the cell, allowing the network to learn which information to retain, which to update, and which to discard.

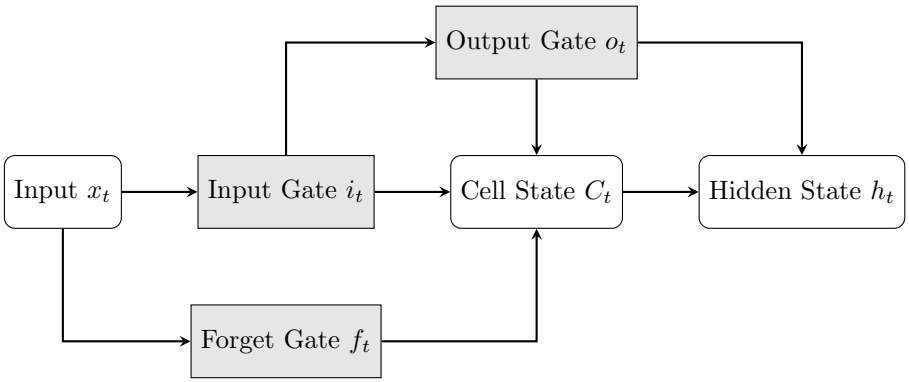


Figure 4.1: *Schematic diagram of a single LSTM memory cell. The cell contains three gates: the input gate i_t, the forget gate f_t, and the output gate o_t. These gates control the flow of information into, within, and out of the cell, allowing the network to manage long-term dependencies effectively.*

Memory Cell The memory cell is the core of the LSTM network, responsible for storing the cell state C_t. The cell state acts as a conveyor belt with minor linear interactions, allowing the information to pass through unchanged unless regulated by the gates.

Input Gate The input gate controls how much of the new information from the current input x_t should be added to the cell state. It is defined as:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \tag{4.9}$$

where i_t is the activation of the input gate, W_i and b_i are the weights and biases, σ is the sigmoid function, h_{t-1} is the previous hidden state, and x_t is the current input.

Forget Gate The forget gate determines how much of the previous cell state C_{t-1} should be forgotten. It is defined as:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \tag{4.10}$$

where f_t is the activation of the forget gate.

Output Gate The output gate decides what the next hidden state h_t should be based on the current cell state. The hidden state is used for prediction and passed to the next time step:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o), \tag{4.11}$$

$$h_t = o_t \cdot \tanh(C_t), \tag{4.12}$$

where o_t is the activation of the output gate, and tanh is the hyperbolic tangent function applied to the cell state C_t.

Applying LSTM Networks to Lagrangian Continuum Simulations In Lagrangian simulations, each particle's state (position, velocity, and material properties) evolves over time as it interacts with its surroundings. The challenge is to predict the future state of each particle based on its past states, which requires capturing both short-term dynamics (e.g., immediate changes in velocity due to collisions) and long-term dependencies (e.g., gradual accumulation of forces over time).

Input Representation The input to the LSTM network at each time step consists of multiple frames of the state of a particle, represented by a matrix:

$$\mathbf{P}_t = \begin{bmatrix} \mathbf{p}_{t-k} \\ \mathbf{p}_{t-k+1} \\ \vdots \\ \mathbf{p}_t \end{bmatrix},$$

where each $\mathbf{p}_t = (x_t, y_t, v_{x,t}, v_{y,t}, \mathbf{m}_t)$ is a vector representing the state of the particle at a specific time frame. This sequence of frames is fed into the LSTM cell, which updates its internal state and produces a hidden state h_t that represents the learned temporal features.

Training the LSTM The LSTM network is trained using *multiple sequences of particle states over time.* In practice, we have N distinct sequences, each with a length of T time steps. For example, N might range from hundreds to thousands, depending on the dataset, and T could be between 2 and 5 time steps—balancing the need to capture temporal dependencies with computational efficiency.

During training, the network learns to predict the next state of the particle based on its multiple past states. The loss function, typically Mean Squared Error (MSE), measures the difference between the predicted state $\hat{\mathbf{p}}_t$ and the actual state $\mathbf{p}_t$ at each time step:

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \|\mathbf{p}_t - \hat{\mathbf{p}}_t\|^2, \tag{4.13}$$

where T is the length of the sequence, $\mathbf{p}_t$ is the actual state of the particle, and $\hat{\mathbf{p}}_t$ is the predicted state at time step t.

The training process employs the Adam optimizer (Kingma & Ba (2014)), a widely used algorithm due to its efficiency and adaptability across various tasks. Adam combines the advantages of two other extensions of stochastic gradient descent—namely, Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp)—to compute individual adaptive learning rates for each parameter. Typically, the learning rate is initialized at a small value, commonly around $\eta = 0.001$, to ensure stable convergence. Additionally, Adam utilizes two hyperparameters, β_1 and β_2, which control the exponential decay rates of the moving averages of the first and second moments of the gradients. Standard values for these hyperparameters are $\beta_1 = 0.9$ and $\beta_2 = 0.999$, which generally provide good performance in practice. The training continues for a set number of epochs or until convergence criteria are met, often using early stopping techniques to avoid overfitting by halting training when the model's performance on a validation set begins to deteriorate.

Advantages of LSTM Networks LSTM networks offer several advantages over traditional methods and even standard RNNs. One significant benefit is their ability to capture long-term dependencies, which makes them particularly well-suited for modeling the complex interactions present in Lagrangian simulations. Unlike simpler models, LSTMs excel at handling the temporal relationships that are critical for accurately predicting the evolution of particle states over time.

Additionally, LSTMs can effectively model non-linear relationships in the data. This non-linearity is crucial for accurately simulating the behavior of fluids and deformable solids, where interactions between particles often involve complex, non-linear dynamics. Another advantage of LSTM networks is their robustness to noise. Due to their ability to selectively forget irrelevant information through mechanisms like the forget gate, LSTMs can filter out noise that is common in real-world simulation data, leading to more accurate predictions.

Challenges of LSTM Networks Despite their strengths, LSTM networks also present several challenges. One of the primary challenges is their computational complexity. Compared to simpler models like MLPs, LSTMs are more computationally expensive, both in terms of training time and memory usage. This is due to their intricate architecture, which

includes multiple gates and memory cells designed to manage information flow effectively.

Another challenge lies in hyperparameter tuning. The performance of LSTMs is highly dependent on the careful selection of hyperparameters, such as the number of layers, hidden units, and learning rate. These hyperparameters, as discussed earlier in this thesis, require careful tuning to ensure that the model performs optimally. Additionally, the length of the input sequence can significantly impact LSTM performance. Very long sequences might increase computational costs, while very short sequences may fail to capture all relevant dependencies, leading to suboptimal predictions.

Alternative RNN Architectures In addition to LSTM networks, other types of RNNs, such as Gated Recurrent Units (GRUs) and vanilla RNNs, are also employed for temporal learning. Vanilla RNNs, as described by Hopfield (1982) and Feng (n.d.), are simpler versions of RNNs that do not have memory cells or gates. While they are less computationally expensive than LSTMs, they suffer from issues such as vanishing and exploding gradients, which make them less effective for capturing long-term dependencies.

GRUs, on the other hand, are similar to LSTMs but feature a simplified architecture. Introduced by Cho et al. (2014), GRUs combine the forget and input gates into a single update gate, making them faster to train while still being able to capture long-term dependencies. However, in very complex tasks, LSTMs generally outperform GRUs, making them the preferred choice when accuracy and the ability to model intricate temporal relationships are paramount.

4.3 Ground Truth for Continuum Simulation

In this section, we provide a detailed overview of the datasets and simulation environments used in our research on fluid and deformable solid simulations. The primary goal is to establish a comprehensive ground truth that serves as the basis for training and validating our models. This involves generating diverse scenarios that capture various physical interactions between different materials, such as liquids, snow, ropes, and jelly-like substances. Figure 4.2 illustrates examples of the different types of multi-material simulations used in our study. These include homogeneous liquid, snow, rope and jelly ball interactions, as well as mixed simulations where different materials interact dynamically.

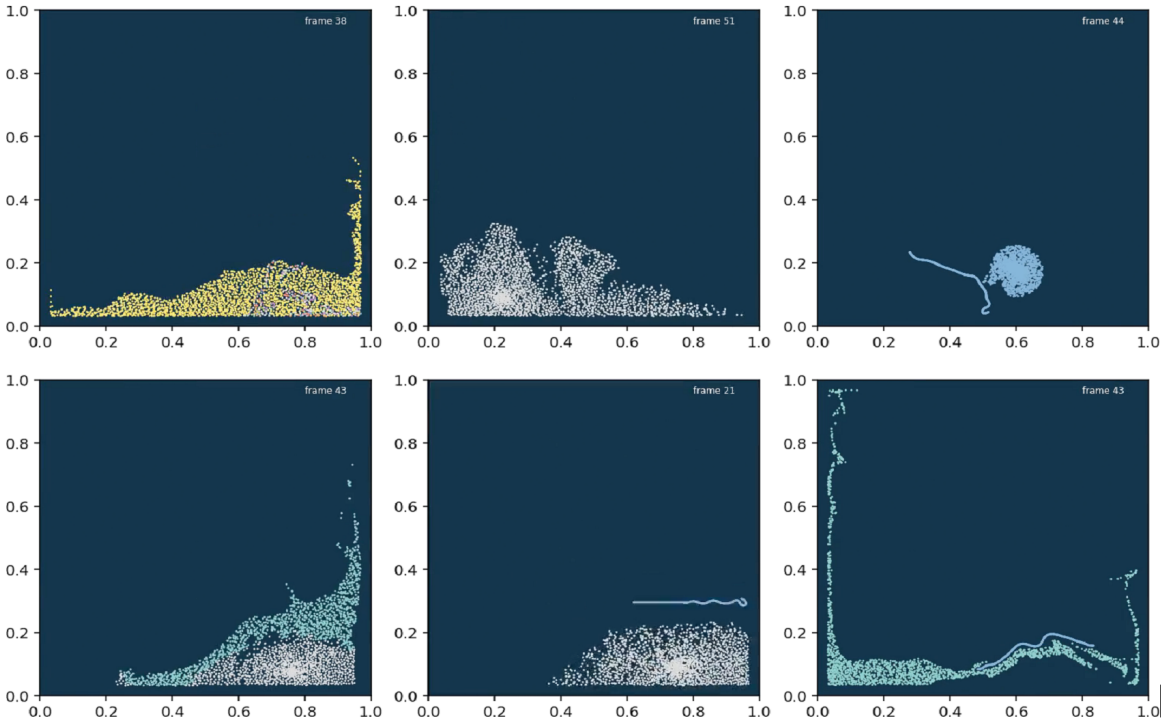


Figure 4.2: *Dataset for multi-material multi-phase simulation. Top-left: homogeneous liquid; top-middle: snow; top-right: rope and jelly ball; bottom-left: liquid and snow; bottom-middle: rope and snow; bottom-right: rope and liquid.*

4.3.1 Continuum Domain

These experiments involved two primary types of simulations: homogeneous fluid simulation and multi-material simulation. For multi-material simulations, we generated 60 two-dimensional scenes using predefined scenarios specified by different scenario IDs, each with unique initial configurations involving varying shapes, sizes, initial positions, and velocities of fluid bodies, jelly, and snow within a square container of 1 by 1 unit. Similarly, for fluid-only simulations, we generated 40 distinct scenes, each focused solely on fluid dynamics, with varied initial configurations of fluid particles.

The duration of the multi-material scenes was uniformly set to 10 seconds to capture the full range of material interactions. For the fluid-only simulations, scene durations varied from 5 to 25 seconds. These simulations were designed to run until significant fluid dynamics had subsided, avoiding the bias of prolonged stationary scenarios in the training data. By ending the simulations when little activity remained, we ensured that the dataset represented a diverse set of dynamic behaviors without over-representing static configurations.

All simulations were conducted at 60 fps, with 20 substeps per frame, ensuring high temporal resolution. Each scene featured 2,500 particles confined within a grid of 64×64 cells for the multi-material simulations and 128×128 cells for fluid-only simulations.

To further characterize the flow in these experiments, we refer to the Reynolds number as defined in Eq. 2.3. Based on the normalized parameters used in our simulations (with $\rho = 1$, a characteristic length $L \approx 1$, and a dynamic viscosity $\mu = 0.5$), the Reynolds number is estimated to vary between approximately 0.2 and 40. This range indicates that, although the flow experiences dynamic changes, the overall regime remains predominantly laminar.

4.3.2 Simulation Process

Environment and Setup We conducted our simulations using the MPM (Material Point Method) solver on a GPU with CUDA 10, running in an Ubuntu 18 environment.Table 4.1 outlines the hardware and software specifications used in this work.

The MPM solver operates through several key steps to accurately simulate the physical interactions of particles and materials. Depending on the type of simulation, different configurations are used to optimize the accuracy and relevance of the generated data.

For **multi-material simulations**, the simulation is initialized with 2,500 particles con-

Table 4.1: *Environment Specification for data generation and training on Temporal Learning Experiment*

Environment	Specifications
CPU	6-Core Intel i7-5930K 64 bits Haswell, clock: 3.7GHz L2: 15.0 MiB
RAM	64 GB DDR4 2133MHz
GPU	2 X NVIDIA GP102 [TITAN Xp]
Physics Simulator	MPM on Taichi 0.6.3 CUDA enabled
Training Framework	TensorFlow 2

fined within a grid of 64×64 cells. This setup allows for the detailed modeling of interactions between different materials such as fluids, elastic materials (like jelly), and snow. Each material is characterized by distinct parameters, including Young's modulus, Poisson's ratio, and a dynamically adjusted **hardening coefficient**, which influences material stiffness based on deformation.

In contrast, **fluid-only simulations** employ a higher-resolution grid of 128×128 cells, also initialized with 2,500 particles. This increased grid resolution enhances the simulation's ability to capture fine details of fluid dynamics, focusing solely on the behavior of fluid particles without the complexity of multi-material interactions.

In both types of simulations, the grid resolution and particle count are determined by a quality factor, resulting in a spatial resolution with grid cells of size $\Delta x = \frac{1}{64}$ for multi-material simulations and $\Delta x = \frac{1}{128}$ for fluid-only simulations.

The simulation advances in time steps of $\Delta t = 1 \times 10^{-4}$ seconds, calculated to maintain numerical stability according to the Courant–Friedrichs–Lewy (CFL) condition (Courant et al. (1928)). The CFL condition is given by:

$$\Delta t \leq \frac{C \cdot \Delta x}{v_{\max}} \tag{4.14}$$

where C is the CFL number (typically less than 1 for stability), Δx is the grid cell size, and $v_{\max}$ is the maximum velocity in the simulation. With $C = 0.9$ and assuming a reasonable maximum velocity $v_{\max} = 10$ units/second, the CFL condition ensures that our chosen $\Delta t = 1 \times 10^{-3}$ seconds per sub-step remains well within the stability limit.

To ensure accuracy, each frame of the simulation is computed using 20 sub-steps, with

each sub-step involving the transfer of particle state (position, velocity, material properties) to the grid. The grid serves as the computational domain where forces such as gravity and interactions with boundaries are applied.

The deformation of particles in multi-material simulations is captured using a deformation gradient $\mathbf{F}$, which is updated during each sub-step. Particles' velocity fields and material properties are then interpolated back from the grid, updating their states accordingly. Fluid-only simulations follow a similar process but focus solely on fluid dynamics, without the need to account for different material properties.

Boundary conditions are enforced to confine the particles within the simulation domain, preventing them from crossing the grid boundaries. Additionally, in both types of simulations, an attractor force can be applied to influence particle motion towards a specified point, simulating scenarios such as gravity wells or magnetic attraction. This attractor force is only applied before the initial fluid data recording starts, serving to generate a variety of initial conditions without affecting the actual dynamics captured during the simulation.

Throughout the simulation, minimum and maximum values of particle positions and velocities are tracked and normalized for data storage. This normalization is particularly important in fluid-only simulations, as it adjusts velocities within a range suitable for training machine learning models and ensures that the dataset captures a broad spectrum of fluid dynamics.

The simulation setup produces highly detailed data, with each scene generating 2,400 frames over a 2-second duration at 60 frames per second for multi-material simulations. Fluid-only simulations, however, vary in duration between 5 to 25 seconds, depending on the scenario, and are terminated once significant fluid dynamics have subsided, ensuring that the dataset remains representative of active rather than stationary scenarios.

4.3.3 Data Generation and Description

Figure 4.3 illustrates examples from the dataset used for fluid simulation, while Figure 4.2 displays example datasets for multi-material simulations. For the multi-material simulations, initial configurations were generated using a shell script that automated the creation of 60 unique scenarios by running the python code of MPM Simulation with varying scenario IDs. These scenarios included diverse initial conditions for fluids, jelly, and snow, ensuring a wide range of material interactions.

For fluid-only simulations, a separate script was used to automate the generation of 40

different scenarios, each focused solely on fluid dynamics. These scenarios were executed using the python code, which varied the initial configurations of fluid particles to capture different fluid behaviors.

In both types of simulations, we captured the position (p_x, p_y), velocity (v_x, v_y), and material type (m) of each particle in every frame. Positions and velocities were normalized relative to the simulation domain and a predefined maximum velocity to ensure consistency across different scenes. The resulting datasets provide a comprehensive collection of fluid and multi-material interactions, as depicted in Figures 4.3 and 4.2.

Data Storage and Compression To facilitate data compression and portability between the fluid simulator and TensorFlow, we stored the simulation data in HDF5 (H5) files. This binary format significantly reduces file size compared to text formats like CSV, JSON, or XLSX, and allows for faster loading without the need for encoding or parsing. The datasets generated from both the multi-material and fluid-only simulations were stored in this format.

For multi-material simulations, we generated 60 distinct scenarios, each producing data for 10 seconds at 60 fps, resulting in 36,000 frames. For larger dataset training, we generated an additional 15 scenarios, each running for 20 seconds at 60 fps, resulting in 18,000 frames. In total, this yielded 54,000 frames of data for multi-material simulations.

For fluid-only simulations, we generated 40 scenarios with varying durations: one scenario ran for 25 seconds, two scenarios for 20 seconds each, three scenarios for 15 seconds each, four scenarios for 10 seconds each, ten scenarios for 7 seconds each, and twenty scenarios for 5 seconds each. This resulted in a total of 19,200 frames for the fluid-only simulations.

Altogether, these simulations produced a substantial dataset with a diverse range of physical interactions, efficiently stored in approximately 2 gigabytes of HDF5 files. This comprehensive dataset is well-suited for training machine learning models to learn complex fluid dynamics and multi-material interactions.

4.4 Data Pre-processing and Feature Standardization

In the data pre-processing phase, our primary objective is to prepare the simulation data for effective training of the machine learning model. The simulation data comprises feature vectors that include positions (x, y), velocities (v_x, v_y), and potentially material parameters.

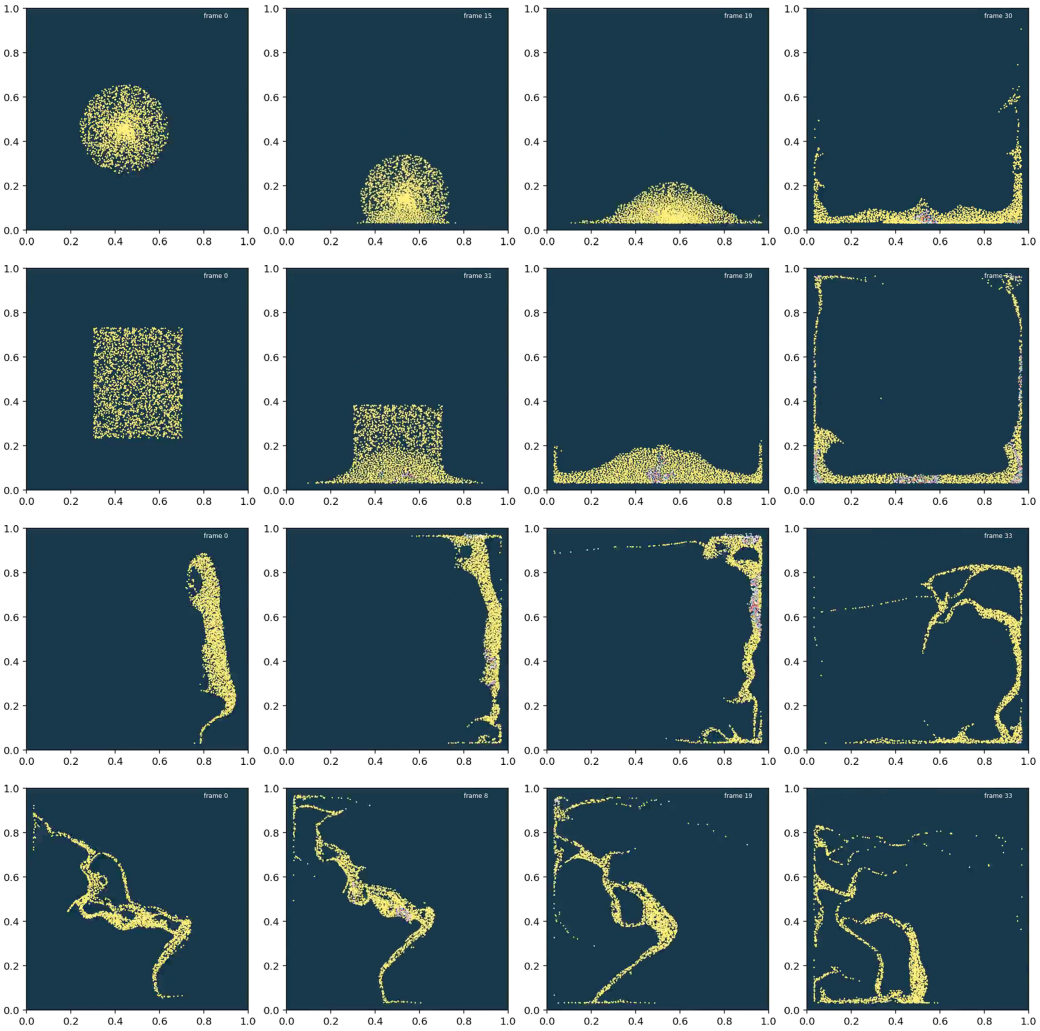


Figure 4.3: *Dataset of fluid simulation with 2,500 particles per domain. Each row represents a different initial configuration and its evolution over time:* ***Top row:*** *A circular fluid body, initially positioned near the center, spreading out and interacting with the boundaries.* ***Second row:*** *A rectangular fluid body, similarly positioned near the center, deforming and dispersing under gravity and boundary interaction.* ***Third row:*** *A fluid body subjected to an initial force or velocity directed towards the right wall, resulting in the fluid rapidly moving and accumulating against the boundary.* ***Bottom row:*** *A complex-shaped splash, with particles having random initial velocities, followed by a free fall under gravity*

These features naturally have different ranges of values, which must be standardized to ensure that each feature contributes equally during model training.

Data Loading and Preparation The multi-material simulation data is stored in HDF5 files, with each file containing multiple frames of particle data. Each frame consists of 2,500 particles, each represented by several features. To facilitate the standardization process, the data from all simulation frames is loaded into memory and reshaped from a 3D array of dimensions *(totalSimulationFrames, 2500, nFeatures)* to a 2D array of dimensions *(totalSimulationFrames * 2500, nFeatures)*. This reshaping allows us to treat each feature independently during standardization.

Handling Different Ranges of Values Given that the feature vector X includes different types of data—positions, velocities, and potentially material parameters—each feature may have a distinct range of values. For instance, positions might be within a normalized range [0, 1], while velocities could vary more widely depending on the dynamics of the simulation. To ensure that no single feature disproportionately influences the model's learning process, it is essential to standardize these features independently.

Standardization Process We standardize each feature in the dataset using the Z-Score function, a common method that scales the data so that each feature has a mean of 0 and a standard deviation of 1. The standard score for each sample X is calculated as follows:

$$z = \frac{X - \mu}{\sigma} \tag{4.15}$$

where X is the individual feature value, μ is the mean of the feature across the entire training dataset, and σ is the standard deviation of the feature across the training dataset.

By applying this standardization process, we ensure that all features, regardless of their original range, are on a comparable scale. This prevents features with larger numerical ranges from dominating the training process, allowing the model to learn effectively from all features.

Implementation of Standardization The standardization is implemented using the *StandardScaler* function from Scikit-Learn. After loading and reshaping the data, the scaler, previously fitted on the entire dataset, is used to transform both the input (X) and target (Y) data. This transformation normalizes each feature independently, making the dataset more suitable for training machine learning models.

After standardization, the data is reshaped back to its original 3D format to preserve the spatial and temporal relationships between particles across frames. The standardized data is

then stored in HDF5 format for use in the subsequent training process.

Data Splitting Following standardization, the data is divided into training, validation, and test sets. Care is taken to ensure that each set contains distinct simulation scenarios, allowing for effective model evaluation and validation.

This comprehensive pre-processing pipeline ensures that the simulation data is standardized and properly formatted, ready for the training process, and capable of contributing effectively to the machine learning model's learning process.

4.5 Training Procedure

We train our fluid and multi-material simulation network in a supervised manner, using particle trajectories generated by a classic physics-based simulation as the "ground truth." The loss function, $\mathcal{L}^{n+1}$, is defined as the Mean Squared Error (MSE) between the predicted and actual particle *attributes* at time step $n+1$. Specifically,

$$\mathcal{L}^{n+1} \;=\; \frac{1}{N\,k}\sum_{i=1}^{N}\sum_{j=1}^{k}\Big(y_{i,j}^{n+1} \;-\; \hat{y}_{i,j}^{n+1}\Big)^2, \tag{4.16}$$

where:

- N is the number of particles.
- k is the number of scalar attributes per particle (e.g., (x, y, v_x, v_y), densities, material parameters, etc.).
- $y_{i,j}^{n+1}$ denotes the ground truth value of the j-th attribute for the i-th particle at time step $n+1$.
- $\hat{y}_{i,j}^{n+1}$ is the corresponding predicted value from the network.

By defining the loss in this manner, we allow the framework to handle any set of attributes of interest (positions, velocities, densities, material properties, etc.), making it generalizable to a variety of multi-material settings.

The data is split into distinct subsets, with approximately 80% of the scenarios allocated for training, 10% for validation, and 10% for testing. Specifically, we use 52 scenarios for

training, 6 for validation, and 6 for testing. This splitting ensures that the model is tested on a diverse range of unseen scenarios.

The training is implemented using TensorFlow 2, with TensorBoard utilized for real-time monitoring of the training process. We optimize the loss function $\mathcal{L}$ using the Adam optimizer Kingma & Ba (2015), with the default learning rate of 0.001. After experimenting with various batch sizes, we settled on a batch size of 4096, which provided a balance between training stability and speed.

4.6 Network Architecture and Hyperparameter Tuning

In this section, we explain our systematic experimentation process, guided by the validation set's error (Tan et al. (2019)), to identify the best configuration for our network architecture. This includes choosing the activation function, adjusting various hyperparameters, and experimenting with larger datasets.

4.6.1 Selection of Activation Function

The choice of activation function is critical for the performance and efficiency of training deep neural networks. We experimented with several popular nonlinear activation functions, such as $Sigmoid$, $Tanh$, $ReLU$, and ELU (Nwankpa et al. (2018)). To determine the best activation function, we trained a simple deep neural network (MLP) with three layers, each containing 200 units, and an output layer with 10,000 units to predict the next simulation frame based on input data from the previous frame.

Each frame consists of 2,500 particles, with each particle characterized by four properties: (px, py, vx, vy) for the fluid model and five properties (px, py, vx, vy, m) for the multi-material model. Thus, both the input and output layers have 10,000 units. We compared the Mean Squared Error (MSE) after 500 epochs for different activation functions, as shown in Table 4.2. The results indicated that the network with ELU activation achieved the lowest MSE (0.1131 for training and 0.1187 for validation).

The exponential linear unit (ELU) is essentially a smoothed ReLU function. As shown in Figure 4.4, ELU reached the lowest MSE among the activation functions within 500 epochs, indicating faster training. Figure 4.5 further illustrates the relationship between training time and the minimum loss achieved with different activation functions. The plot also shows a small gap between the validation loss and training loss, which reflects that the network is not overfitted. From this comparison, it is evident that ELU and ReLU reduce the loss much faster than Sigmoid and Tanh, with ELU achieving the lowest final training and validation loss. Additionally, ELU maintains stable training dynamics throughout, with minimal fluctuation in the loss curve. Given its superior performance in terms of rapid convergence, low final loss, and training stability, we decided to use ELU activation in most of the subsequent experiments.

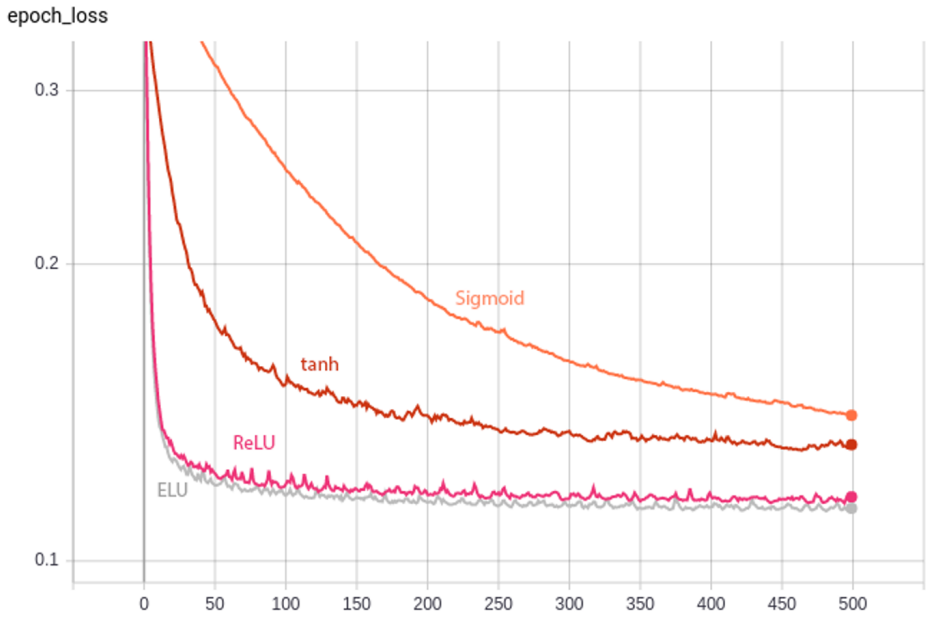


Figure 4.4: *Comparison of training loss between different activation functions. The horizontal axis represents the number of training epochs. Networks with ELU and ReLU demonstrate faster learning.*

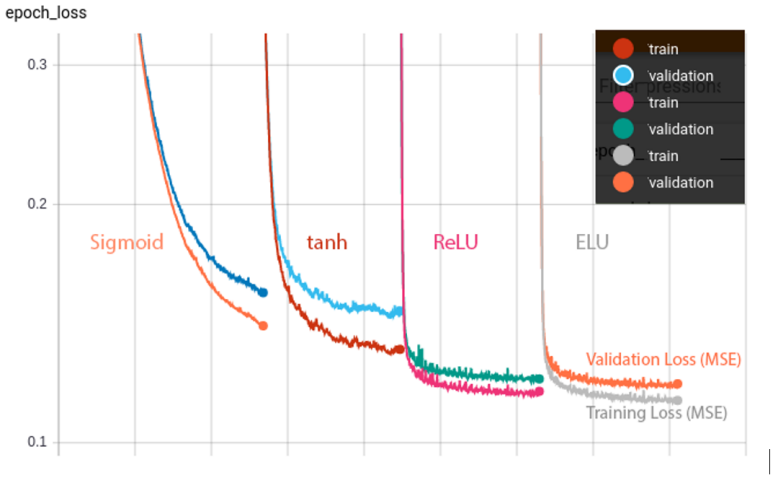


Figure 4.5: *Learning curve showing validation and training loss for different activation functions. The small gap between the validation loss and training loss indicates that the network is not overfitted. The horizontal width of each plot corresponds to 500 epochs (unlabeled to simplify the comparison, as the four plots were merged into one image for easier side-by-side comparison).*

Table 4.2: *MSE results comparison from a different type of activation function (after 500 epochs)*

No	Model	Activation	Number of Neurons				Train.	Val.
			Dense Network Layer			Pred.		
			L1	L2	L3	Layer		
1	3LSigmoid	sigmoid	200	200	200	10000	0.1405	0.1547
2	3LTanh	tanh					0.1312	0.1467
3	3LReLU	ReLU					0.1161	0.1204
4	3LELU	ELU					0.1131	0.1187
Total Params	(4,090,600)		2,000,200	40,200	40,200	2,010,000		

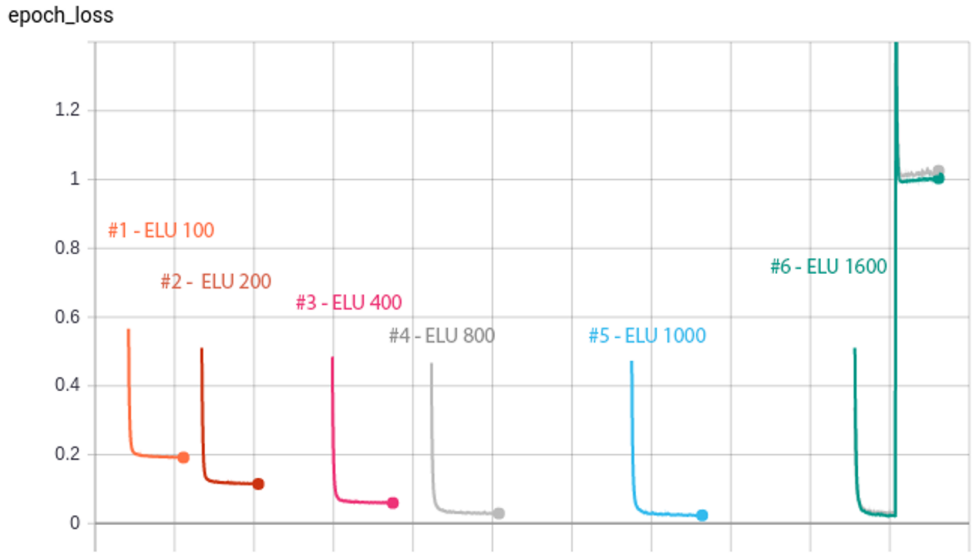
4.6.2 Layer Size

To determine the impact of the number of neurons on the network's predictive performance, we trained six neural networks with ELU activation, varying only the number of neurons in each layer. Each network was trained for 200 epochs with a batch size of 512.

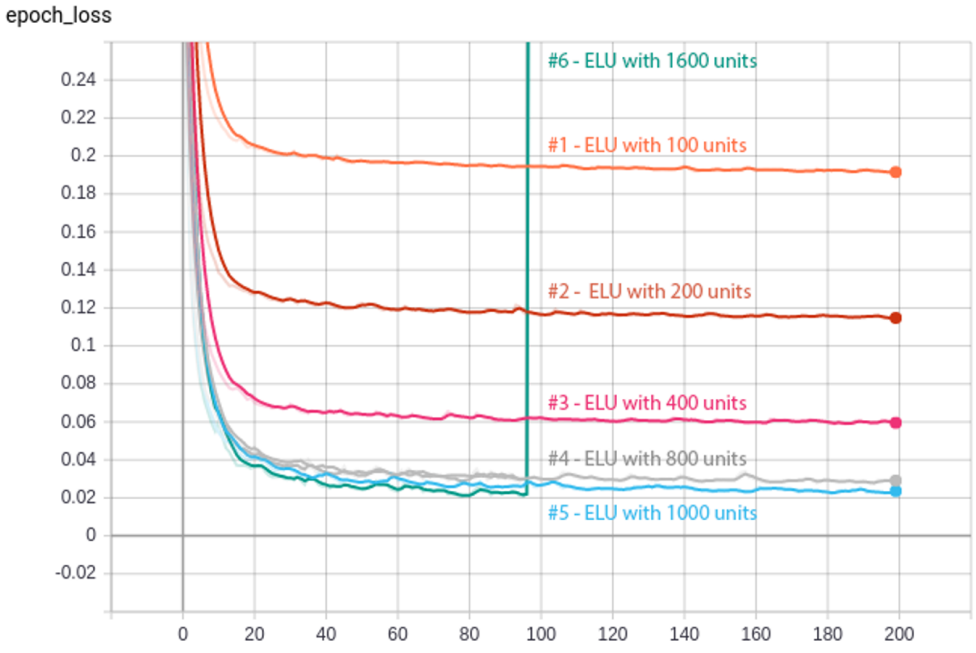
Table 4.3 shows that increasing the number of neurons generally lowers the MSE, but at the cost of significantly higher memory usage due to the increased number of parameters. Notably, configuration 6, which had the largest number of neurons, exhibited instability, with the MSE increasing dramatically after 96 epochs. This issue is likely due to the combination of a relatively small batch size and the aggressive learning behavior associated with the ELU activation function. In Figure 4.6b, the MSE increases sharply just before reaching 100 epochs. Our observation suggests that this behavior results from the relatively small batch size of 512 combined with the highly progressive learning behavior of the ELU activation function. To address this instability, we opted to use a larger batch size in subsequent experiments. This adjustment not only enhances learning stability but also increases GPU occupancy, thereby reducing overall training time. Figure 4.6a provides a side-by-side comparison of the training loss across different neuron sizes. All networks reached a flat loss curve by around 40 epochs, with the main differences being the minimum loss achieved, which correlates with the number of neurons. For configuration 6, training instability led to the MSE jumping to values greater than 1, resulting in poor model accuracy.

Table 4.3: *Benchmarking different unit sizes on Nework with ELU activation*

#	1	2	3	4	Total Params	Train MSE	Val MSE
1	100	100	100	10000	2,030,300	0.1915	0.1968
2	200	200	200	10000	4,090,600	0.1147	0.1194
3	400	400	400	10000	8,331,200	0.0591	0.0657
4	800	800	800	10000	17,292,400	0.0290	0.0362
5	1000	1000	1000	10000	22,013,000	0.0235	0.0309
6	1600	1600	1600	10000	37,134,800	1.0036	1.0247



(a) *Side-by-side comparison of epoch-training loss for networks with different neuron sizes (y-axis: MSE). The horizontal width of each plot corresponds to 500 epochs(unlabeled to simplify the comparison, as the four plots were merged into one image for easier side-by-side comparison)*



(b) *The higher the number of neurons, the lower the MSE. Configuration 6 (green) shows instability after 96 epochs.*

Figure 4.6: *Comparison of training loss and MSE for networks with different neuron sizes: (a) Epoch-training loss comparison; (b) MSE versus neuron size, highlighting the instability in Configuration 6.*

4.6.3 Number of Layers

We also investigated the effect of varying the number of layers on training loss. Five neural networks with 1,000 neurons in each layer were trained using ELU activation (excluding the output layer with 10,000 units). The training ran for 250 epochs with a batch size of 1024, using 25,200 data frames (46.7% data).

Table 4.4: *Comparison of different Neural Networks with different layers*

#	#Layers	#Params	Train MSE	Validation MSE
1	3 ELU	22,013,000	0.0197	0.0268
2	4 ELU	23,014,000	0.0185	0.0267
3	5 ELU	24,015,000	0.0200	0.0279
4	6 ELU	25,016,000	0.0241	0.0306
5	7 ELU	26,017,000	0.0244	0.0322

Our results indicated that while increasing the number of layers can reduce training loss, this effect diminishes as the network deepens. For example, while the 4-layer network outperformed the 3-layer network, adding more layers did not yield significant improvements and in some cases, led to overfitting, as shown by the higher validation MSE in the 7-layer network (4.4).

4.6.4 Dataset Scaling and Layer Optimization

We hypothesized that increasing the dataset size would enhance the network's performance by providing more diverse training data, which is crucial for capturing the variability inherent in complex fluid and material simulations. Initially, the network was trained on 25,200 frames from 45 scenarios. To test our hypothesis, we generated 15 additional scenes, each running for 20 seconds at 60 fps, bringing the total dataset to 43,200 frames (80% of the data).

The larger dataset was expected to improve the model's generalization by exposing it to a broader range of physical interactions, thus reducing the risk of overfitting. Additionally, this increase in data volume necessitated adjustments in the network architecture to maintain efficient training.

To accommodate the larger dataset, we used a batch size of 4096, which helped to improve training stability by smoothing out the gradient updates and enhancing the occupancy of GPU

resources, thus accelerating the training process.

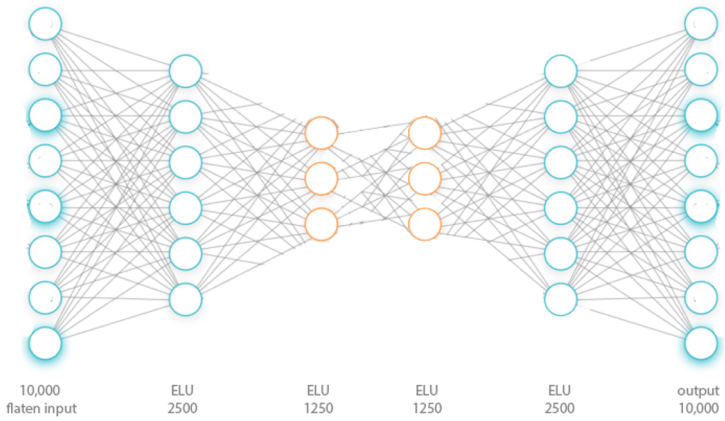


Figure 4.7: *Overview of the selected baseline architecture: ELU [2,500, 1,250, 1,250, 2,500, 10,000]*

Figure 4.7 provides a visual overview of one of our candidate multi-layer perceptron (MLP) architectures. We experimented with several configurations, varying the number of layers and neurons per layer, as summarized in Table 4.5.

Table 4.5: *Neural Network Training and Validation Metrics on Larger Datasets*

Config.	#Layers	Neurons	#Params	Epochs	Train MSE	Val MSE
1	3	3 x 1000	22,013,000	250	0.0287	0.0345
2	4	4 x 1000	23,014,000	400	0.0339	0.0282
3	4	2 x 2500, 2 x 1250	57,830,000	600	0.0091	0.0152
4	4	4 x 2500	68,770,000	500	0.0107	0.0162

Given that simply adding more layers does not always lead to better performance, we focused on optimizing the network with either 3 or 4 layers. Our experiments revealed that configuration 3, consisting of 4 layers with a mix of 2,500 and 1,250 neurons, achieved the lowest mean squared error (MSE) during training. This configuration is depicted in Figure 4.7 and proved to be the most efficient in terms of both performance and memory usage (see Table 4.5).

The choice of 2,500 and 1,250 neurons was based on balancing the need for a sufficient number of parameters to model complex interactions with the computational constraints. This

configuration provided a good trade-off between training speed and accuracy, while avoiding the pitfalls of overfitting that can arise with deeper networks.

Since the validation loss remains close to the training loss (within 0.1 to 0.2), as illustrated in Figure 4.8, we can conclude that the network generalizes well and is not prone to overfitting. Therefore, we selected this architecture as the baseline for further experiments and comparisons with more sophisticated models.

This optimized configuration will serve as a reference point for future architectural explorations, guiding the development of more advanced models that may incorporate additional layers, neurons, or even entirely different network structures to further enhance performance.

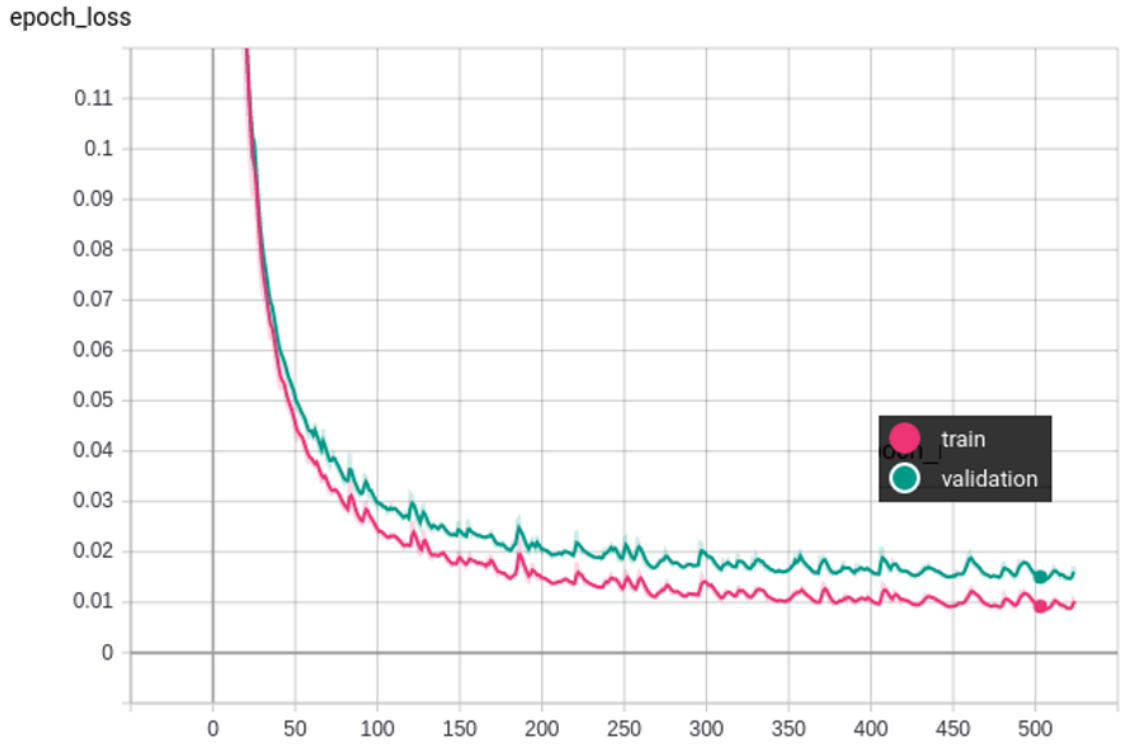


Figure 4.8: *Training Loss of Selected Architecture (y-axis: MSE, x-axis: number of epochs)*

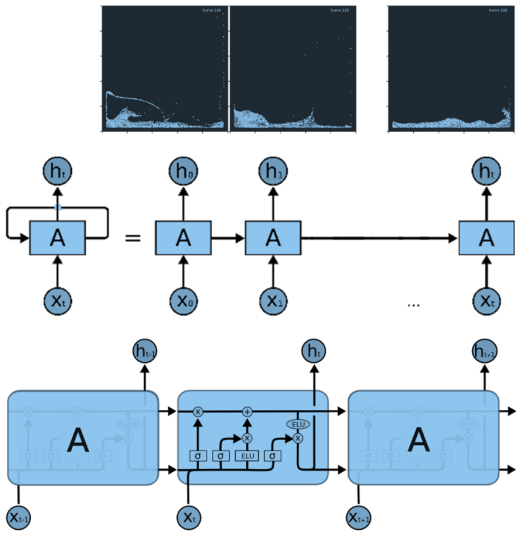


Figure 4.9: *Our LSTM Fluid Network. Two previous frames are used as input to predict the current frame.*

4.7 Proposed Temporal Model With End-to-End Latent Space and LSTM

Figure 4.9 (middle) illustrates the loops of a recurrent neural network (RNN), denoted as A. At each time step t, the RNN processes input x^t and produces a hidden state h^t, thereby carrying information forward from multiple previous frames. In an LSTM (Figure 4.9 bottom), the repeating module features four interacting layers—gating mechanisms that enable the network to better preserve and emphasize relevant parts of that history Olah (2015). Consequently, while both vanilla RNNs and LSTMs make use of multiple past time steps, the LSTM's memory mechanism can selectively retain crucial information more effectively. Our network leverages these capabilities to predict the next frame (Figure 4.9 top) by using a window of two consecutive frames as input, allowing for more accurate corrections.

In this section, we extend our LSTM-based approach by incorporating the encoder–decoder methodology described in 4.1 directly into our fluid and deformable solid predictions. We learn our latent space end-to-end via pointwise MLP encoders and decoders. This lets us avoid fixed, precomputed dimensionality reductions and more flexibly capture nonlinear relationships among particle states.

4.7.1 High-Level Architecture

Figure 4.10 (schematic) shows the pipeline, which processes two consecutive input frames $\mathbf{x}_{t-1}, \mathbf{x}_t$ (each containing $N = 2500$ particles, with per-particle attributes (x, y, v_x, v_y)) and predicts the next state $\mathbf{h}_t$. Each frame is first encoded into a latent embedding via a pointwise MLP applied independently to each particle and a global pooling step. A two-layer LSTM with dropout processes these frame embeddings over time, producing a final hidden state. Finally, a decoder MLP expands this hidden state back to the full ($N \times$ features) dimension.

The Encoder (Per-Frame) involves a pointwise MLP, which maps each particle's $\big(x,\ y,\ v_x,\ v_y\big)$ into a 1250-dimensional latent feature. We then apply mean pooling across all $N = 2500$ latent features to obtain a single 1250-D vector representing the entire frame. This yields a compact learned latent representation $\mathbf{z}_t \in R^{1250}$ at each time step t.

The Temporal LSTM Module processes the sequence $\{\mathbf{z}_{t-1}, \mathbf{z}_t\}$ using a two-layer LSTM, where each layer has a hidden size of 1250. Dropout is applied after each LSTM layer to reduce overfitting (e.g., 0.2 after the first LSTM and 0.1 after the second). The LSTM's final hidden state $\mathbf{h}_t \in R^{1250}$ serves as a compressed representation of the last few frames.

The Decoder (Frame Reconstruction) uses a small MLP (either a single linear layer or a multi-layer MLP) that takes $\mathbf{h}_t$ and predicts all $N \times 4$ outputs corresponding to the next frame's particle states $\hat{\mathbf{x}}_{t+1}$. In our setup, $\hat{\mathbf{x}}_{t+1}$ has the shape $\big(2500,\ 4\big)$, which matches the positions and velocities for each particle. This decoding is learned jointly with the encoder and LSTM.

4.7.2 Sequential Inputs and Window Size

We use a window of two frames as input ($T = 2$) to predict the next frame. Specifically, at training time, each sample consists of the frames $\big(\mathbf{x}_{t-1}, \mathbf{x}_t\big)$, which together predict $\mathbf{h}_t$. At inference time (rollout), once we predict $\mathbf{h}_t$, we set $\mathbf{x}_{t+1} = \mathbf{h}_t$ as the "newest" frame in the sequence. To predict $\mathbf{h}_{t+1}$, we feed in $\big(\mathbf{x}_t, \mathbf{h}_t\big)$, and this closed-loop approach can continue arbitrarily far into the future.

4.7.3 Learned Latent Space Versus Fixed PCA

The latent space is learned end-to-end along with the LSTM. Both the encoder MLP and decoder MLP weights are updated jointly via backpropagation. This approach typically results

in richer, nonlinear embeddings compared to a fixed linear PCA mapping. Moreover, it allows the network to adapt the encoding to the dynamics that need to be predicted.

4.7.4 Hidden and Cell States in the LSTM

Each LSTM layer internally maintains hidden and cell states of size 1250. The output of the first layer is fed through an MLP block (or possibly through a second LSTM layer, depending on the variant), which also has hidden units of size 1250, maintaining its own hidden and cell state. Ultimately, we take the final hidden state $\mathbf{h}_t$ as the LSTM's output. This design offers a balance between expressiveness and memory constraints for our 2D fluid/solid simulations.

4.7.5 MLP Before the Output

As Figure 4.10 shows, the final LSTM hidden state $\mathbf{h}_t$ is passed through a simple MLP or fully-connected layer (sometimes just a linear projection) to produce the 2500×5 outputs. A small nonlinearity, such as an ELU activation, is often added in this decoder step for better stability. The decoder may also be labeled as "Frame Decoder" or "Output MLP" in the diagrams.

4.7.6 Training Details

We train the network end-to-end using a standard mean-squared-error (MSE) objective on positions and velocities, supplemented by an additional density-based loss to encourage globally correct particle distributions. Specifically:

- Dataset Size (T and N): We use $N = 2500$ particles per frame, each containing 2D positions (x, y) and velocities (v_x, v_y). We collect a total of T_{frames} frames from various fluid or solid simulations. Each training sample consists of a pair of frames $(\mathbf{x}_{t-1}, \mathbf{x}_t)$ as input, and $\mathbf{x}_{t+1}$ as the target.

- Batching and Loss: We create minibatches of size B (e.g., 192), each containing partial sequences. The total loss $\mathcal{L}$ combines position MSE, velocity MSE, and density map MSE (4.1.3). Optimization uses Adam with a moderate learning rate (e.g., 10^{-3}), typically for several hundred epochs.

- Relation to **T** and **N**: During training, each sample contains 2 frames in and 1 frame out. We iterate over all frames in the simulation (from 0 to $T_{\text{frames}} - 1$), with each frame

containing $N = 2500$ particles. The same MLP is applied pointwise to each particle, and no explicit PCA is needed.

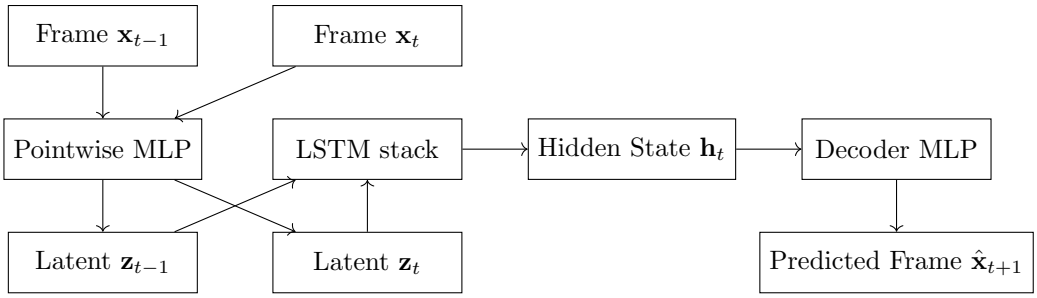


Figure 4.10: *Architecture diagram for the proposed model with encoder-decoder and LSTM.*

We enhance temporal modeling by combining a two-layer LSTM with an MLP interleaved between the layers. We use a window of two consecutive frames as input, and dropout is applied after the first and second LSTM layers for regularization to improve network performance. In the first LSTM layer, the sequence of frame embeddings is passed, and the output is processed by an MLP layer to transform the hidden dimensions. Finally, the network outputs the predicted frame using a decoder MLP. The details of the architecture are shown in Table 4.6. By learning the encoder, LSTM, and decoder weights simultaneously, the model constructs a latent-space representation on the fly that best suits the prediction task. The next section presents both qualitative and quantitative results showing how well this end-to-end approach captures long-term fluid and solid behavior.

Table 4.6: *Details of the proposed Spatial LSTM architecture with MLP interleaved.*

#	Layers Type	Activation	Neurons	Output Shape
1	Input 2 Frames (LSTM)	ELU	2500	(batch, seq_len, 1250)
	Dropout 0.2			(batch, seq_len, 1250)
2	LSTM 1st Layer	ELU	1250	(batch, seq_len, 1250)
	Dropout 0.1			(batch, seq_len, 1250)
3	MLP (Hidden transformation)	ELU	1250	(batch, seq_len, 1250)
4	LSTM 2nd Layer	ELU	1250	(batch, seq_len, 1250)
5	Output MLP	ELU	1250	(batch, 2500, 5)

4.8 Evaluation

In this section, we evaluate the performance of the proposed models across various simulation tasks, including homogeneous fluid simulations and multi-material interactions. We compare the predictions of different neural network architectures—namely, Deep Neural Networks (MLP), Long Short-Term Memory networks (LSTM), and hybrid LSTM+MLP models—against ground truth data. The evaluation is carried out both qualitatively and quantitatively, focusing on the accuracy of the predicted particle positions and velocities, as well as the models' ability to maintain the physical integrity of the simulated materials over time. Through this comprehensive analysis, we aim to identify the strengths and limitations of each model, providing insights into their suitability for different types of fluid and multi-material simulations.

4.8.1 Homogeneous Fluid Surrogate Model Prediction Evaluation

We first evaluated our models' performance on a homogeneous liquid simulation to establish a baseline before training on more complex multi-material scenarios. We compared three trained models: a Deep Neural Network (MLP), a Long Short-Term Memory network (LSTM), and a hybrid LSTM+MLP model. The qualitative differences between these models are significant, as illustrated in Figure 4.11.

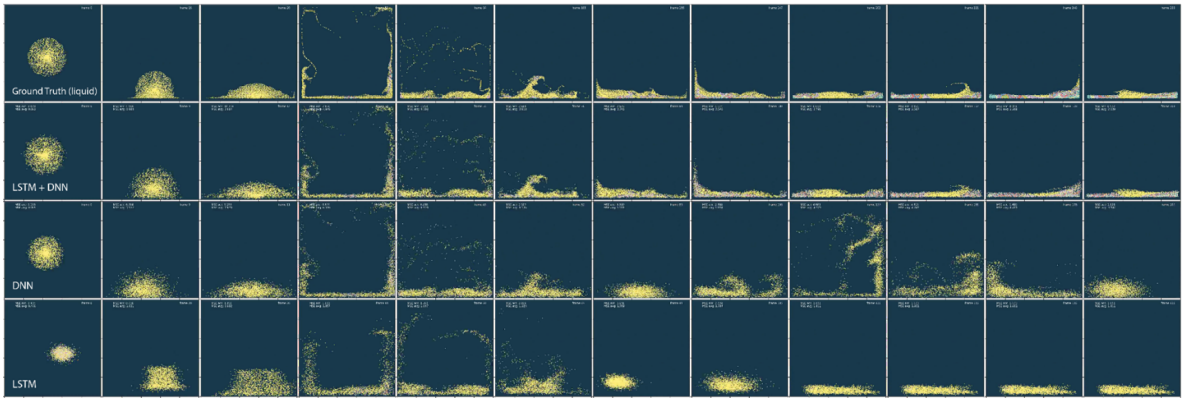


Figure 4.11: *Qualitative comparison of network performance on predicting a liquid state. The* ***top row*** *shows ground-truth (physics-based) simulation results, while the* ***three rows below*** *show predictions from LSTM+DNN, DNN, and LSTM models (top to bottom). Each column represents a different time step in the simulation.*

The MLP model, while better at predicting the overall shape of the liquid, fails to maintain temporal consistency. This is evident in the right-most column of the third row in Figure 4.11, where the prediction drifts significantly, indicating a jump to an incorrect simulation state. The LSTM model, in contrast, excels at preserving the temporal trajectory of the fluid state but struggles with accurately modeling the liquid's shape, particularly in scenarios involving significant deformation, as shown in the fourth row.

The hybrid LSTM+MLP model delivers the best performance, combining the strengths of both approaches. As seen in the second row, it successfully maintains both the fluid's temporal trajectory and its shape. To quantify these observations, we computed the Mean Squared Error (MSE) of particle positions and velocities at each frame relative to the ground truth. The results, summarized in Table 4.7, reinforce the qualitative findings.

Table 4.7: *Quantitative network performance on liquid simulation.*

ID	Scene Description	Average MSE			Sequential End MSE		
		MLP	LSTM	LSTM + MLP	MLP	LSTM	LSTM + MLP
26	Falling circled liquid	3.342	3.583	1.485	2.127	4.489	0.222

4.8.2 Multi-material Surrogate Model Prediction Evaluation

To evaluate the generalization ability of our models in multi-material simulations, we trained the same networks using datasets that include various material combinations. We used six combinations of materials identified by their hardness coefficient (liquid-snow, snow-rope, liquid-liquid, snow-snow, jelly ball-rope, and rope-liquid). For each combination, we generated ten different initial states by varying particle positions and velocities according to predefined constraints. Particle positions were initialized within specific geometric regions, such as rectangles or circles, ensuring they were uniformly distributed within these areas. The initial positions were randomized within these boundaries, but the particles were constrained to remain within the specified limits. Velocities were also typically uniform within each scenario, with some scenarios including a controlled degree of randomization within a defined range. This method ensured that the initial conditions were physically plausible and consistent across different simulations.

In total, we generated 60 scenes, which were randomized to ensure diverse training, validation, and testing datasets. The dataset was split into 80% for training, 10% for validation,

and 10% for testing.

We performed roll-out predictions using the testing data, with qualitative comparisons illustrated in Figures 4.12 through 4.13 (top). In Figure 4.12, the deformation of both liquid and snow is well maintained throughout the sequence. However, as the liquid stabilizes, the model predicts a slightly faster cessation of small-scale surface oscillations than observed in the ground truth.

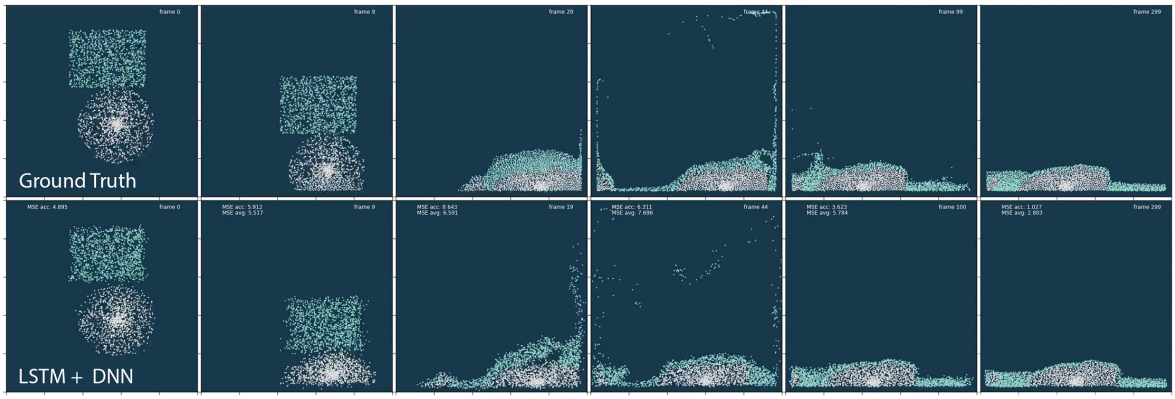


Figure 4.12: *Qualitative comparison of liquid-snow simulation. The model accurately predicts the deformation and stabilization of both materials.*

In the snow-snow simulation (Figure 4.13 top), the model's predictions are visually accurate throughout, with the final frames showing no wave oscillations, matching the expected physical behavior.

Simulating the interaction between a rope and a liquid presents a more challenging case, as seen in Figure 4.13 (middle). While the model successfully predicts the rope's coarse deformation, it struggles to maintain the rope's connectivity, with particles appearing slightly disjointed at certain frames.

Figure 4.13 (bottom) shows the simulation of an elastic jelly ball interacting with a rope. A notable observation is the bouncing of the jelly ball, which the model replicates accurately. However, during high-speed motion, the model struggles to preserve the solid shape, with particles dispersing before reverting to their original form, particularly evident in the fourth column.

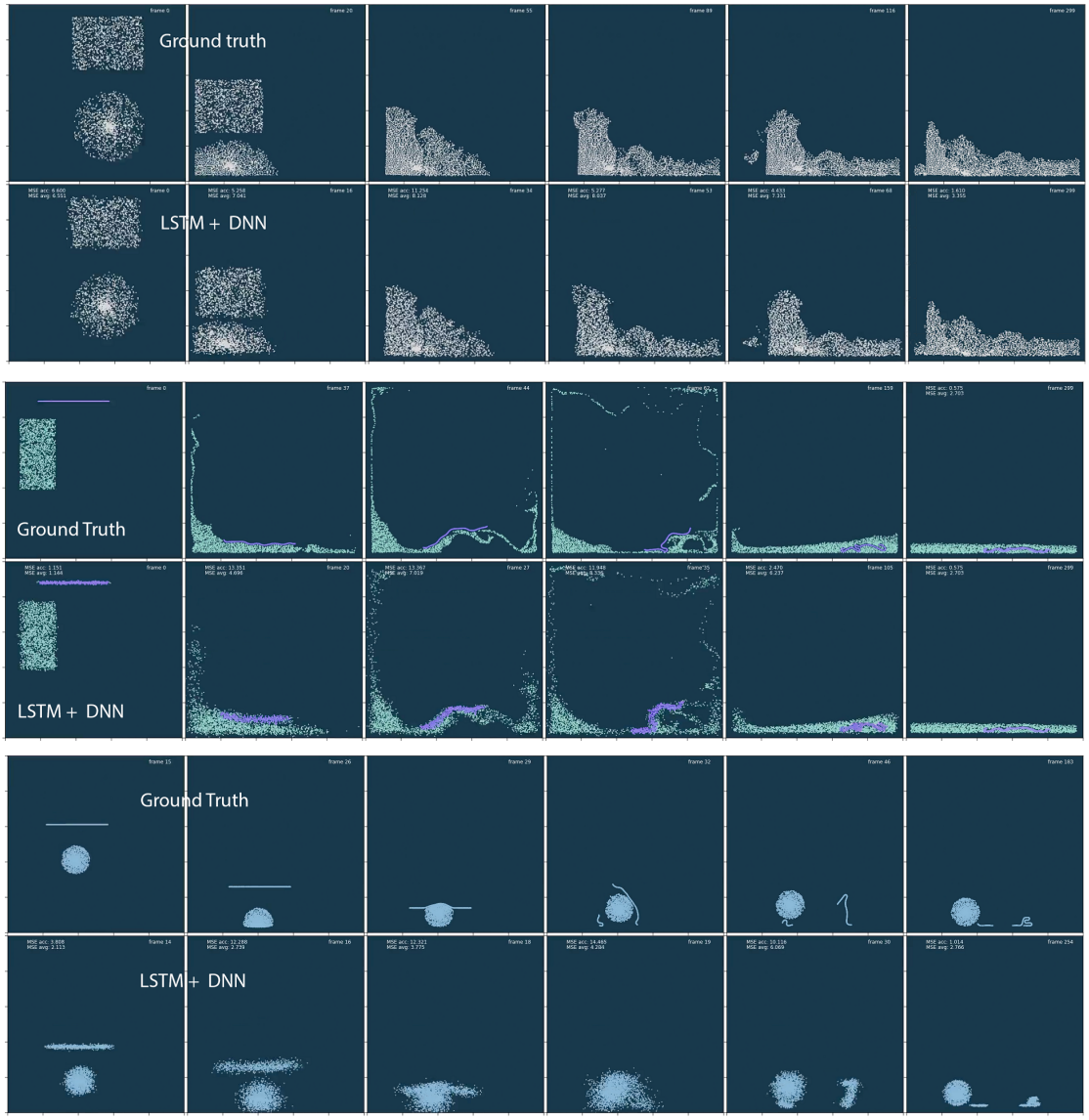


Figure 4.13: *Qualitative comparisons for various simulations: (top) Snow-snow simulation, (middle) Liquid-rope simulation, and (bottom) Jelly ball-rope simulation.*

4.8.3 Comparison with ARIMA: Large-Scale VAR Approach

Although our primary focus is on neural network–based surrogates (DNN, LSTM, and hybrid LSTM+MLP), we also investigated the feasibility of using a classical ARIMA-like model for these fluid and multi-material simulations. Specifically, we attempted to apply a mul-

tivariate Vector Autoregression (VAR) model (from the `statsmodels` library) to the same high-dimensional particle data.

Rationale. An ARIMA or VAR model estimates linear dependencies among time series. In principle, with 2,500 particles $\times$ 4 features (position and velocity), each simulation frame becomes a vector of 10,000 dimensions. A VAR(p) model then tries to learn how each of these 10,000 components depends on the previous p time steps of all 10,000 components.

Implementation. As with the neural network approach, we first loaded all frames from the `.h5` files, standardized them (each feature across all frames), and then flattened each frame to size $[1 \times 10{,}000]$. We concatenated all frames into a single $(T \times 10{,}000)$ array, where T is the total number of time steps. We then used a simple train/validation/test split by chronological order. A code snippet illustrating this procedure is shown in Listing 4.1.

Listing 4.1: Fitting a large-scale VAR(1) model to the flattened particle data.

```
from statsmodels.tsa.api import VAR

# Suppose 'var_data' has shape (T, D) = (total_frames, 10000).
# We use T_train = 70% for training, etc.
train_data = var_data[:T_train]
val_data   = var_data[T_train:T_train+T_val]
test_data  = var_data[T_train+T_val:]

model = VAR(endog=train_data)
results = model.fit(maxlags=1)    # VAR(1)

# Attempt forecasting the validation set
forecast_steps = val_data.shape[0]
forecast = results.forecast(train_data[-1:], steps=forecast_steps)
```

Results. While fitting the VAR(1) model on 10,000 dimensions theoretically addresses the same forecasting problem as the LSTM, the parameter space grows prohibitively large. Even with a single lag ($p = 1$), the coefficient matrix has $(10{,}000 \times 10{,}000)$ parameters, plus intercept

terms. On typical hardware, this often leads to out-of-memory errors or extremely long runtimes.

On our Intel Core i7-5930K system (6 cores, 12 threads), the solver allocated about 19 GB of virtual memory and actively consumed over 16 GB of RAM while running all CPU cores at full capacity for more than half an hour. Although the solver did eventually complete in around 33 minutes, forecasting on the validation set yielded an MSE of approximately 3.5×10^{18}, demonstrating that the model's predictions were numerically unstable and physically implausible.

Empirically, we found that attempts to run this VAR approach for the multi-material dataset often exhausted system memory or failed to converge in a reasonable timeframe. Moreover, even in cases where the model returned a solution, the forecasts diverged rapidly, providing no meaningful approximation of fluid–solid dynamics. These issues underscore that, unlike the neural network models, an ARIMA/VAR approach does not scale well to thousands of features. Furthermore, VAR relies on a linear assumption between features, whereas fluid and deformable-solid simulations exhibit inherently nonlinear, neighbor-based interactions.

Conclusion. Our experiments confirm that ARIMA-like methods (via large-scale VAR) are *not viable* for high-dimensional particle simulations. Although such models can be applied to smaller or low-dimensional problems, the neural network surrogates (particularly the hybrid LSTM+MLP) remain far more practical and effective for large-scale multi-material fluid simulations, both in terms of runtime feasibility and the physical plausibility of predictions.

4.9 Ablation Study

To measure long-term similarity, we report the average Mean Squared Error (MSE) of particle states (positions, velocities, etc.) between the ground-truth simulation and the network's predictions over an entire sequence. We let F denote the total number of frames, and thus the final frame is the F-th frame. Building upon the loss function from (1), we define the average MSE, MSEAvg, for a scene with F frames as:

$$\text{MSEAvg} = \frac{1}{F}\sum_{i=1}^{F} L^i, \tag{4.17}$$

where L^i is the MSE at the i-th frame (aggregated over all attributes).

We also consider a "sequential end MSE," which evaluates the error at frame F (i.e., the final frame in the sequence):

$$\text{MSEseq} = L^F. \tag{4.18}$$

Large errors at the final frame can produce visually noticeable artifacts, so end-of-sequence accuracy is especially important.

Table 4.8 shows the ablation study results, reporting both the average MSE across all frames (columns 3–5) and the end-of-sequence MSE at frame F (columns 6–8) for multiple scenarios, ranging from homogeneous liquid simulations to multi-material setups.

The results demonstrate that combining a Long Short-Term Memory (LSTM) module with a Multi-Layer Perceptron (MLP) backbone consistently yields lower MSE than either component alone. In particular, the LSTM+MLP model significantly improves accuracy at frame F, reducing visible artifacts and enhancing overall flow realism in multi-material simulations.

4.10 Conclusion

We explored the essential aspects of temporal learning in the context of continuum simulations. The chapter began with an examination of encoding and decoding techniques, emphasizing the importance of transforming Euclidean coordinates-based continuum data into latent space representations. This process can be effective in training neural networks to model complex physical phenomena.

Table 4.8: *Ablation Study. We compare the average error (lower is better) between the ground-truth particle states and the predicted future frames.*

ID	Scene Description	Average MSE			Sequential End MSE		
		MLP	LSTM	LSTM + MLP	MLP	LSTM	LSTM + MLP
0	Falling rect. liquid	4.413	2.58	0.58	5.019	3.74	2.274
26	Falling circled liquid	2.127	3.583	0.222	3.342	4.489	1.485
54	Liquid and Snow	2.344	2.485	1.019	4.231	3.891	2.668
55	Rope and Snow	3.156	3.132	1.279	3.521	4.132	2.886
56	Rope and liquid	2.123	2.987	0.63	4.124	3.798	2.775
57	Snow and snow	3.891	3.567	1.571	3.277	4.235	3.04
58	Rope and jelly ball	3.325	2.576	0.998	4.362	3.234	2.549
59	Liquid and liquid	2.315	3.478	0.494	3.897	4.561	2.756

The chapter proceeded to discuss traditional time series prediction methods, including Autoregressive (AR), Moving Average (MA), and ARIMA models. Although these methods have been foundational in many fields, their limitations in handling nonlinearities and long-term dependencies highlighted the need for more advanced approaches. This led to the introduction of long-short-term memory (LSTM) networks, which are particularly well suited for capturing the temporal dependencies in dynamic systems like fluid simulations.

Through experimentation, the chapter demonstrated that LSTM combine with MLP networks significantly outperform traditional methods and even basic RNNs in modeling the complex interactions and long-term dependencies inherent in Lagrangian simulations. The experiments highlighted the importance of selecting appropriate network architectures, activation functions, and hyperparameters to achieve optimal performance.

The chapter also delved into the challenges of training deep neural networks, including the need for careful hyperparameter tuning and the impact of dataset size on model performance. The experiments carried out provided valuable information on the trade-offs between model complexity, computational efficiency, and prediction accuracy.

In general, this chapter underscored the importance of temporal learning in the accurate simulation of continuum systems. Using advanced neural network architectures such as LSTM, the research presented here contributes to the development of more robust and accurate

models to simulate the complex dynamics of fluids and deformable solids. The findings of this chapter lay a strong foundation for future work in improving the predictive capabilities of neural networks in continuum simulations.

Chapter 5

Vorticity Conservation on Surrogate Model

This chapter focuses on understanding a surrogate model designed to preserve vorticity in Lagrangian fluid simulations. It begins with a summary that introduces the problem in a two-dimensional context, using the lid-driven cavity flow as an example, and outlines the key contributions of the research.

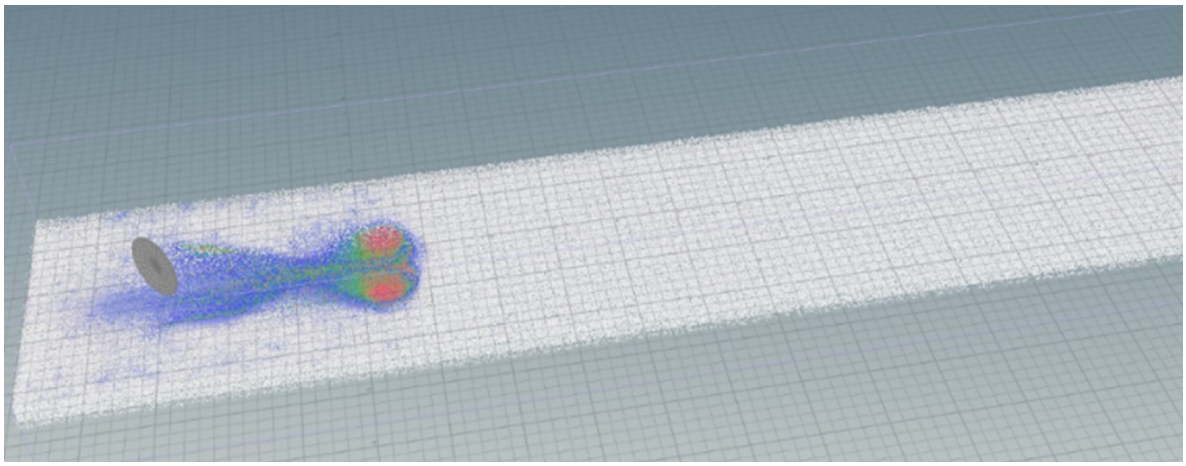


Figure 5.1: *Propagation of vortex rings in a water tank. The red color indicates the particles with the highest magnitude of vorticity.*

5.1 Motivation

Imagine a calm pool of water, its surface undisturbed until a sudden motion—perhaps a hand thrust through the water—generates a mesmerizing circular vortex ring. This ring, a toroidal vortex structure, travels gracefully through the fluid, propelled by internal pressure differences within the rotating water. This phenomenon, known as vortex-ring propagation, illustrates the captivating dynamics of fluid motion, where the interplay of velocity, viscosity, and shear forces creates complex, self-sustaining structures.

Capturing and maintaining these fluid structures in simulations presents significant challenges, especially in Lagrangian fluid simulations. Accurately conserving vorticity, which represents the local spinning motion of fluid, is essential for modeling phenomena like vortex shedding and turbulence. While the models developed in Chapter 4, such as MLP and LSTM networks, were useful in predicting the evolution of fluid states, they faced limitations when it came to representing rotational dynamics accurately.

The primary challenge with these models was that they processed the entire fluid system as a whole, assuming global particle interactions. This approach has limitation, as it requires considering the entire fluid domain, even when interactions occur only in specific, localized regions. Fluid simulations often involve behaviors that are confined to smaller areas, such as vorticity and vortex dynamics. The models in Chapter 4 were not well-suited for these localized interactions, as they did not focus on specific regions of the fluid.

This limitation led us to switch to a Graph Neural Network (GNN), which allows for spatially focused learning. Rather than treating the entire fluid domain uniformly, the GNN focuses on localized regions of the fluid, capturing the interactions between neighboring particles. This approach was key to preserving vorticity and other rotational dynamics, as it allowed the model to better handle the dynamic and variable nature of fluid flows.

The proposed GNN architecture, enhanced with a Self-Supervised Graph Attention Operator, significantly improved the model's ability to conserve vorticity and angular momentum. Evaluations of the model showed that it outperformed previous approaches, achieving lower mean squared errors and better alignment with ground truth data. These improvements highlight the model's potential for realistic and scalable fluid dynamics simulations, offering a promising direction for future research and refinement of surrogate modeling techniques.

5.2 Conservation of Angular Momentum

The conservation of angular momentum is a fundamental principle in continuum mechanics, describing how the rotational motion of a system remains constant in the absence of external torques. This principle is crucial in the study of both incompressible fluids and deformable soft solids (elastic fluids). In this section, we examine the conservation of angular momentum for these two distinct types of materials, highlighting the differences and implications for fluid dynamics.

5.2.1 Incompressible Fluid

For an incompressible fluid, the angular momentum $\mathbf{L}$ is given by:

$$\mathbf{L} = \int_{\Omega} \mathbf{r} \times (\rho \mathbf{u})\, dV \tag{5.1}$$

where $\mathbf{r}$ is the position vector, ρ is the fluid density, $\mathbf{u}$ is the velocity field, and Ω is the fluid domain.

In the context of the Navier-Stokes equations, the fluid's incompressibility is described by:

$$\nabla \cdot \hat{\mathbf{u}} = 0 \quad \text{Incompressibility condition on } \Omega \tag{5.2}$$

The momentum conservation for an incompressible fluid is governed by:

$$\rho \frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \quad \text{Momentum conservation on } \Omega \tag{5.3}$$

Here, ρ denotes the fluid density and μ its viscosity. Equation 5.2 states that the fluid is incompressible, implying a divergence-free velocity field. Equation 5.3 describes the balance of momentum, considering the contributions from the pressure gradient, viscous forces, and external forces Batchelor (1967).

To derive the angular momentum conservation equation, we take the cross product of the momentum equation with the position vector $\mathbf{r}$:

$$\frac{d\mathbf{L}}{dt} = \int_{\Omega} \mathbf{r} \times \left(-\nabla p + \mu \nabla^2 \mathbf{u}\right)\, dV \tag{5.4}$$

In the absence of external body forces or torques, the total angular momentum remains conserved:

$$\frac{d\mathbf{L}}{dt} = 0 \tag{5.5}$$

5.2.2 Elastic Fluid / Deformable (Soft) Solid

For a deformable soft solid, or elastic fluid, the conservation of angular momentum also incorporates the material's elasticity and deformation characteristics. The angular momentum $\mathbf{L}$ for such a material is similarly defined as:

$$\mathbf{L} = \int_{\Omega} \mathbf{r} \times (\rho \mathbf{u})\, dV \tag{5.6}$$

However, the stress tensor σ in this context includes both viscous and elastic components:

$$\sigma = -p\mathbf{I} + \mu \nabla \mathbf{u} + \lambda(\nabla \cdot \mathbf{u})\mathbf{I} + 2G\mathbf{e} \tag{5.7}$$

Here, λ and G are the Lamé parameters, $\mathbf{I}$ is the identity matrix, and $\mathbf{e}$ is the strain tensor. The angular momentum conservation equation for elastic fluids is derived by taking the cross product of the position vector $\mathbf{r}$ with the divergence of the stress tensor:

$$\frac{d\mathbf{L}}{dt} = \int_{\Omega} \mathbf{r} \times (\nabla \cdot \sigma)\ dV \tag{5.8}$$

Again, in the absence of external torques, the total angular momentum remains conserved:

$$\frac{d\mathbf{L}}{dt} = 0 \tag{5.9}$$

5.2.3 Comparison and Implications

Although both incompressible fluids and deformable soft solids follow the principle of angular momentum conservation, the presence of elasticity in soft solids introduces additional complexity in the form of elastic stress components. This requires consideration of viscous and elastic forces in the momentum and angular momentum equations, leading to a more complex behavior under deformation and rotation.

Ensuring the conservation of angular momentum in simulations that involve either type of material is crucial to accurately capture rotational dynamics and the effects of forces and torques within the system.

5.3 Conservation of Energy Analysis

In fluid dynamics, the conservation of energy principle is a fundamental aspect that ensures the accurate representation of physical systems. This section discusses the conservation of

kinetic energy in the context of Lagrangian fluid simulation.

5.3.1 Kinetic Energy in Lagrangian Fluid Simulation

Lagrangian fluid simulation is a technique where fluid "particles" are tracked over time, and their motion is governed by the principles of Newtonian mechanics. Each fluid "particle" carries properties such as mass, velocity, and position, and the simulation computes the evolution of these properties over time.

The kinetic energy (E_k) of a fluid particle in a Lagrangian framework is given by:

$$E_k = \frac{1}{2}mv^2 \tag{5.10}$$

where m is the mass of the particle and v is its velocity. The total kinetic energy of the fluid is the sum of the kinetic energies of all the particles:

$$E_{k,\text{total}} = \sum_{i=1}^{N} \frac{1}{2}m_i v_i^2 \tag{5.11}$$

where N is the total number of particles, m_i is the mass of the i-th particle, and v_i is the velocity of the i-th particle.

5.3.2 Conservation of Kinetic Energy

In an ideal, inviscid (non-viscous) fluid with no external forces, the total kinetic energy should be conserved. This implies that the total kinetic energy at the beginning of the simulation should be equal to the total kinetic energy at any later time, assuming no energy is added to or removed from the system.

Mathematically, this can be expressed as:

$$\frac{dE_{k,\text{total}}}{dt} = 0$$

This conservation can be affected by various factors in a simulation, such as numerical dissipation, external forces, and boundary conditions. Therefore, ensuring energy conservation is a crucial aspect of validating the accuracy and stability of a Lagrangian fluid simulation.

5.4 Measuring Vorticity

Vorticity is a fundamental concept in fluid dynamics that characterizes the local rotational motion of fluid particles. It provides insight into the swirling and rotational behaviors within

a fluid flow. Mathematically, vorticity ($\boldsymbol{\omega}$) is defined as the **curl** of the velocity field ($\mathbf{u}$):

$$\boldsymbol{\omega} = \nabla \times \mathbf{u}$$

Here, $\nabla\times$ denotes the curl operator, which measures the rotation at a point within the fluid. A high magnitude of vorticity indicates strong local rotation, while a low magnitude suggests minimal rotational motion.

Understanding vorticity is crucial for analyzing various fluid phenomena, including the formation of vortices, eddies, and turbulent flows. In ideal (inviscid) fluids, vorticity is conserved along the paths of fluid particles. However, in real-world scenarios where viscosity plays a role, vorticity can diffuse and dissipate over time.

In computational fluid dynamics (CFD), numerical methods are employed to calculate and visualize vorticity. Accurate computation and representation of vorticity enable realistic simulations of complex fluid behaviors, which are vital in applications such as weather forecasting, aerodynamics, and the creation of visual effects in entertainment. Effective visualization of vorticity not only enhances the understanding of intricate fluid dynamics but also contributes to more immersive and accurate simulations.

5.4.1 Curl

The curl of a vector vield $\vec{v}(x, y, z) = v_z\vec{e}_z + v_y\vec{e}_y + v_x\vec{e}_x$ is a vector function that can be represented as:

$$\operatorname{curl}\vec{v} = \left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}\right)\vec{e}_x + \left(\frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x}\right)\vec{e}_y + \left(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}\right)\vec{e}_z = \nabla \times \vec{v} \tag{5.12}$$

The curl at a point is proportional to the on-axis torque to which a tiny pinwheel would be subjected if it were centred at that point. The vector product operation can be visualized as a pseudo-determinant:

$$\nabla \times \vec{v} = \begin{vmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ v_x & v_y & v_z \end{vmatrix} \tag{5.13}$$

5.4.2 Vorticity

Figure 5.2 depicts a small fluid element (a square) subjected to velocity gradients, illustrating the concept of angular velocity.

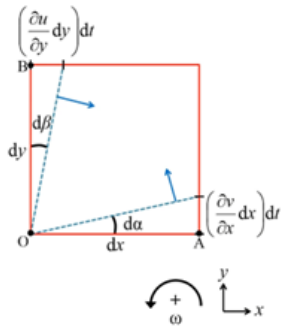


Figure 5.2: *Illustration of angular velocity components for a fluid element. The figure shows a square fluid element with initial vertices labeled O, A, and B. The fluid element is subjected to velocity gradients $\frac{\partial u}{\partial y}$ and $\frac{\partial v}{\partial x}$, which cause the element to undergo small angular displacements $d\alpha$ and $d\beta$. These displacements result in a rotational motion with angular velocity ω, represented by the counterclockwise rotation indicated in the figure.*

The fluid element has initial vertices labeled O, A, and B. The velocity gradients are represented by $\frac{\partial u}{\partial y}$, which denotes the change in the u-component (horizontal) of the velocity in the vertical direction, and $\frac{\partial v}{\partial x}$, which denotes the change in the v-component (vertical) of the velocity in the horizontal direction. The small displacements dx and dy correspond to movements in the x and y directions, respectively. The angles $d\alpha$ and $d\beta$ represent the infinitesimal rotations of the sides OA and OB due to the respective velocity gradients. The angular velocity, ω, is derived as the average rate of these rotations and is expressed by the formula:

$$\omega_z = \frac{1}{2}\left(\omega_{OA} + \omega_{OB}\right) \tag{5.14}$$

$$= \frac{1}{2}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) \tag{5.15}$$

This equation encapsulates how the fluid element rotates due to the differences in velocity components across its edges, providing a measure of the element's rotational motion within

the fluid flow. Using a similar method, the average angular velocity around the x and y axes is:

$$\omega_x = \frac{1}{2}\left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}\right) \tag{5.16}$$

$$\omega_y = \frac{1}{2}\left(\frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}\right) \tag{5.17}$$

The angular velocity vector is given by:

$$\vec{\omega} = \frac{1}{2}\left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}\right)\hat{i} + \frac{1}{2}\left(\frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x}\right)\hat{j} + \frac{1}{2}\left(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}\right)\hat{k} = \frac{1}{2}\nabla \times \vec{V} \tag{5.18}$$

The vorticity is preferred over the angular velocity in fluid mechanics:

$$\vec{\zeta} = 2\vec{\omega} \tag{5.19}$$

The vorticity of particle a is estimated by Monaghan (1992a) as follows:

$$\mathbf{w}_a = (\nabla \times \mathbf{v})_a = \sum_b m_b \frac{\mathbf{v}_a - \mathbf{v}_b}{\rho_b} \times \nabla W_{ab} \tag{5.20}$$

In this equation, $\mathbf{w}_a$ represents the vorticity of particle a. The summation is performed over all neighboring particles b, where m_b is the mass of particle b, $\mathbf{v}_a$ and $\mathbf{v}_b$ are the velocities of particles a and b, respectively, and ρ_b is the density of particle b. The term ∇W_{ab} represents the gradient of the smoothing kernel W_{ab}, which is a function used in Smoothed Particle Hydrodynamics (SPH) to approximate field quantities. In this context, W_{ab} is a weighting function that depends on the distance between two particles a and b. The gradient ∇W_{ab} measures how this weighting function changes with respect to the position of the particles. It is used to calculate the influence of nearby particles on each other, particularly in the context of estimating vorticity, where it helps to determine the rotational effects by considering the velocity differences between neighboring particles. This formulation captures the rotational motion of the fluid particles by considering the differences in velocities weighted by the smoothing kernel gradient, ensuring that the density ρ_b associated with particle b is used in the calculation.

5.5 Vorticity Distribution Evaluation Methods

In the analysis of vorticity data, it is crucial to employ robust evaluation methods to assess the similarity between the ground truth and model predictions. Since the fluid domain is discretized as "particles" in a Lagrangian representation, for accurate evaluation, we interpolate

all the particles' quantities to a fixed-sized uniform grid to obtain the vorticity values across the entire domain. This section discusses three methods for comparing the distribution of the vorticity: Pearson correlation, residual analysis, and Kullback-Leibler (KL) divergence. Each method has its advantages and weaknesses, which are described below.

5.5.1 Pearson Correlation

The Pearson correlation is a statistical tool used to assess the linear dependency between two datasets Dowdy et al. (2004), Sul et al. (2014). This coefficient is computed to determine the similarity between the actual values and the predictions of the model. It gives a measurement of the linear relationship between two datasets, displaying a value within the range of -1 to 1. A value of 1 denotes a perfect positive linear relationship, 0 indicates no linear relationship, and -1 represents a perfect negative linear relationship. The Pearson correlation coefficient r is expressed as follows:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 \sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{5.21}$$

where x_i and y_i are the vorticity values for the ground truth and the model, respectively, and $\overline{x}$ and $\overline{y}$ are their mean values. Pearson correlation is advantageous because it provides a simple and intuitive interpretation, measuring the strength and direction of the linear relationship. However, it only captures linear relationships and may not reflect non-linear similarities. Additionally, it is sensitive to outliers, which can distort the correlation coefficient.

5.5.2 Residual Analysis

Residual analysis involves examining the differences (residuals) between ground truth and model predictions Hoaglin (2003), Brady (2018). The residual r_i for the i-th data point is defined as:

$$r_i = y_i - x_i \tag{5.22}$$

where x_i is the ground truth vorticity and y_i is the predicted vorticity of the model. Residual analysis provides insights into the distribution and magnitude of prediction errors, helping identify systematic biases and trends in the model's predictions. However, it does not provide a single summary statistic, making comparisons more complex. Moreover, residuals can be affected by the scale of the data.

5.5.3 Kullback-Leibler (KL) Divergence

KL divergence measures the difference between two probability distributions Buhl (2023). For vorticity data, we first normalize the vorticity values to form probability distributions. The KL divergence $D_{KL}(P \parallel Q)$ is defined as:

$$D_{KL}(P \parallel Q) = \sum_{i=1}^{n} P_i \log\left(\frac{P_i}{Q_i}\right) \tag{5.23}$$

where P represents the normalized vorticity distribution of the ground truth data, Q represents the normalized vorticity distribution of the model predictions, and P_i and Q_i are the corresponding normalized vorticity values at each point in the domain. In this context, "probabilities" refer to the normalized vorticity values. These values are treated as probabilities after normalization because they are non-negative and sum to one across the domain. This approach allows the use of probability distribution tools like KL divergence to compare the similarity between the ground truth and model-predicted vorticity distributions.

Normalization Process Normalization is crucial for calculating KL divergence as it converts vorticity values into a form that can be treated as probability distributions:

$$P_i = \frac{|v_i| + \epsilon}{\sum_{j=1}^{n}(|v_j| + \epsilon)} \tag{5.24}$$

where v_i is the vorticity value at a given point and ϵ is a small constant added to avoid zero values. The result is a set of normalized values P_i that sum to one, resembling a probability distribution.

Entropy Entropy, in the context of KL divergence, measures the unpredictability or randomness of a distribution. It quantifies the amount of "surprise" or information content in a probability distribution. When comparing two distributions using KL divergence, entropy helps determine how much one distribution diverges from another. Specifically, KL divergence can be seen as the additional entropy (or information loss) incurred when using the model's distribution Q to approximate the ground truth distribution P.

Handling Infinite Values While computing KL divergence, infinite values can occasionally appear. This problem usually occurs in two main situations: First, when there are zero values in the distribution, which causes the logarithm function in the KL divergence formula

to become undefined, leading to infinite values. Second, when there are non-overlapping distributions, meaning that if there are values in the ground truth that are zero where the model has non-zero values, or vice versa, the divergence becomes infinite because the relative entropy calculation involves division by zero.

Regularization Term To address these issues, a small regularization term (epsilon) is added to both the ground truth and model distributions. This adjustment ensures that all probabilities are non-zero, avoiding the undefined logarithm and division by zero problems. The algorithm follows these steps:

First, calculate the magnitudes of the vorticity values to treat them as probability distributions. This approach simplifies the handling of the data by focusing on the absolute differences. Then, normalize both the ground truth and model distributions so that they sum up to 1, ensuring that we are comparing valid probability distributions. A small epsilon value (e.g., 1×10^{-10}) is added to each element of the distributions, ensuring that there are no zero probabilities in the distributions. Here, p and q are the original probability distributions, and ϵ is the small regularization term. Finally, the KL divergence is recalculated using the regularized distributions:

$$D_{KL}(P'||Q') = \sum p' \log \frac{p'}{q'} \tag{5.25}$$

This equation calculates the relative entropy between the regularized ground truth distribution P' and the model distribution Q'.

Conclusion In evaluating the similarity between the ground truth and model predictions of vorticity distributions, each method—Pearson's correlation, residual analysis, and KL divergence—offers unique insights and has distinct advantages and limitations. Pearson correlation effectively measures the linear relationship between datasets, providing a simple and intuitive metric, but it is limited to capturing linear relationships and is sensitive to outliers. Residual analysis offers detailed insights into the magnitude and distribution of prediction errors, allowing for the identification of systematic biases, but it lacks a single summary statistic for straightforward comparison. KL divergence, on the other hand, quantifies the divergence between two probability distributions, capturing both magnitude and directional differences, though it requires careful normalization to avoid biases and handle zero values. Together, these methods provide a comprehensive toolkit for assessing model performance, each com-

plementing the others to give a fuller picture of how well models replicate the ground truth vorticity data.

5.6 Design Method

5.6.1 Network Architecture

This subsection provides a detailed overview of the proposed network architecture. Our goal is to enhance not only fluid velocity prediction but also vorticity by integrating a Self-Supervised Graph Attention Operator with convolutional operations. This integration increases the model's capacity to learn the features of neighboring particles, which is crucial for capturing the angular momentum of each particle.

The overall architecture is based on the work of Sanchez-Gonzalez et al. (2020), which uses Graph Networks (GNs) to simulate complex physical systems. In their work, the model is designed to simulate fluid dynamics by learning particle interactions over time using a graph-based structure. We have adapted their model by substituting the original convolutional layers with the Self-Supervised Graph Attention Operator (SuperGAT) from Kim & Oh (2022). This substitution allows the model to better capture local particle interactions and improve the conservation of vorticity, which is essential for realistic fluid simulations.

The Self-Supervised Graph Attention Operator (SuperGAT) is a novel attention mechanism that dynamically adjusts the importance of neighboring nodes (particles) based on their features and interactions. Unlike traditional graph attention mechanisms, SuperGAT utilizes self-supervised learning to refine the attention coefficients and ensure the network focuses on the most important local interactions. This mechanism, when applied within the Graph Networks (GNs) framework, allows the model to more effectively preserve vorticity, angular momentum, and energy, leading to superior performance in fluid dynamics simulations. This novelty in attention mechanism design significantly improves the model's ability to simulate rotational dynamics and other complex fluid behaviors.

Furthermore, we have modified the architecture by adapting both the layers and the loss function to deal more effectively with vorticity conservation. This adaptation includes a more refined attention mechanism that emphasizes local interactions, helping the model preserve key fluid characteristics such as vorticity, angular momentum, and energy. These modifications ensure that the model can maintain the essential physical properties of fluid

dynamics throughout the simulation.

We build the architecture by taking advantage of the graph representation. In a graph, particles are represented as nodes, and interactions between particles are represented as edges. This structure allows for a more natural and flexible representation of the complex relationships and interactions in a fluid system. The key advantages include:

- **Handling Irregular Geometries:** Unlike grid-based methods, graph representations are not restricted to regular grids and can handle irregular and dynamic geometries. This flexibility is essential for simulating real-world fluid systems where the particle distribution can be non-uniform and dynamic.

- **Rotational Invariance:** One significant advantage of graph representation is its ability to maintain rotational invariance. In fluid dynamics, the relative positions and interactions of particles are more important than their absolute positions. Graph-based methods can inherently respect this property, ensuring that the simulation results are consistent regardless of the orientation of the particles.

- **Capturing Local Interactions:** Graphs can effectively model local interactions between particles, which are crucial for accurately simulating fluid dynamics. Each node (particle) can easily share information with its neighboring nodes (particles) through the edges, capturing the local dependencies and influences.

- **Scalability:** Graph-based methods can scale to large numbers of particles by leveraging sparse connectivity. Only the relevant local interactions need to be considered, reducing the computational complexity compared to methods that require global interactions.

Figure 5.3 illustrates the flow of data and operations through the network. In this figure, the revised Particle Dynamics Stack and the Self-Supervised Graph Attention Operator are shown as central components of the architecture, highlighting the novelty of their inclusion for improved vorticity conservation.

The architecture starts with a neighborhood search for fluid particles, followed by the construction of a graph incorporating node and edge features. These features serve as input to the network. The architecture consists of three main components: an encoder, a PArticle Dynamics Stack, and a decoder.

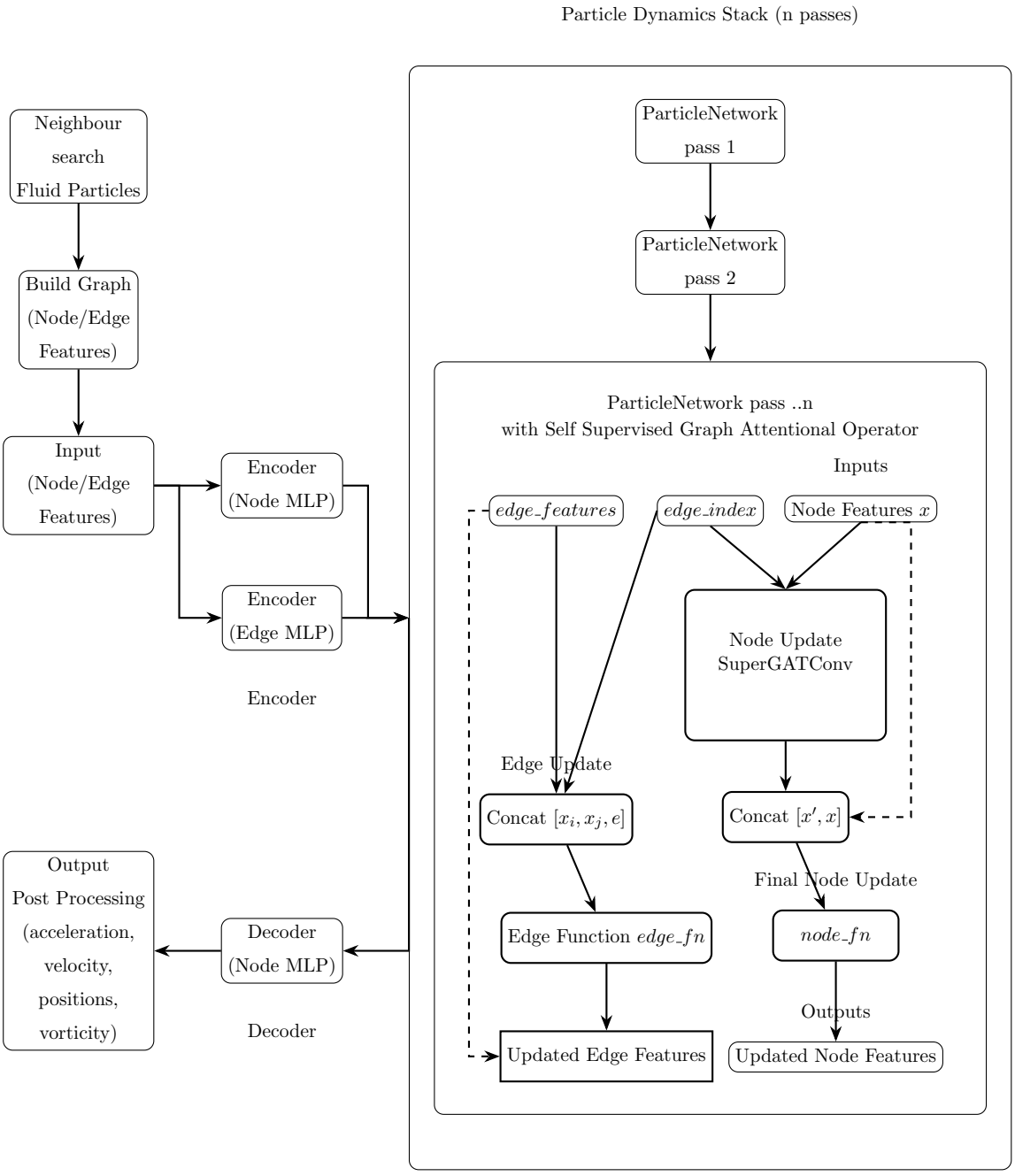


Figure 5.3: *Proposed Neural Network Architecture (OURS) to Conserve Vorticity in Lagrangian Fluid Simulations. The model builds upon the architecture of Sanchez-Gonzalez et al. (2020), with key modifications, including the substitution of the original convolutional layer with the Self-Supervised Graph Attention Operator (SuperGAT). This modification enables the network to better focus on local particle interactions, crucial for vorticity conservation. The Particle Dynamics Stack and SuperGATConv layers have been adapted to ensure that vorticity, angular momentum, and energy are better preserved.*

- **Encoder:** The encoder processes the input node and edge features using two separate Multi-Layer Perceptrons (MLPs). The Node MLP encodes the node features, and the Edge MLP encodes the edge features. These encoded features are then fed into the Particle Dynamics Stack.

- **Particle Dynamics Stack:** is the core of the architecture, comprising a stack of ParticleNetwork layers. Each layer in the stack applies a self-supervised Graph Attention Operator to update the features. This operator dynamically adjusts the importance of neighboring nodes during feature aggregation, enhancing the representation of node features. The Particle Dynamics Stack iterates through multiple passes, refining the features with each pass.

- **Decoder:** The decoder takes the processed node features and applies a Node MLP to generate the output features. These outputs include physical quantities such as acceleration, velocity, position, and vorticity of the fluid particles.

The architecture is designed to conserve vorticity in Lagrangian fluid simulations by effectively capturing the interactions between particles through the graph-based representation and attention mechanism.

5.6.2 Self Supervised Graph Attentional Operator

The self-supervised graph attentional operator introduced by Kim & Oh (2022)

$$\mathbf{x}_i' = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j, \tag{5.26}$$

$\mathbf{x}_i'$ represents the updated feature vector for node i. This is the new representation of node i after applying the graph attention layer. The term $\alpha_{i,i}$ is the attention coefficient for the self-loop of node i, indicating how much attention node i gives to itself in the update process. The matrix $\mathbf{\Theta}$ is a learnable weight matrix that transforms the input feature vectors to a new space. This transformation is shared across all nodes and their neighbors. $\mathbf{x}_i$ refers to the feature vector of node i, representing the initial or previous layer's features of node i. The summation $\sum_{j \in \mathcal{N}(i)}$ is taken over all neighbors j of node i. Here, $\mathcal{N}(i)$ denotes the set of all neighboring nodes of node i, and the summation aggregates information from all these neighbors. The term $\alpha_{i,j}$ is the attention coefficient for the edge between node i and its

neighbor j, representing the importance or weight of the neighboring node j's features when updating the feature vector of node i.

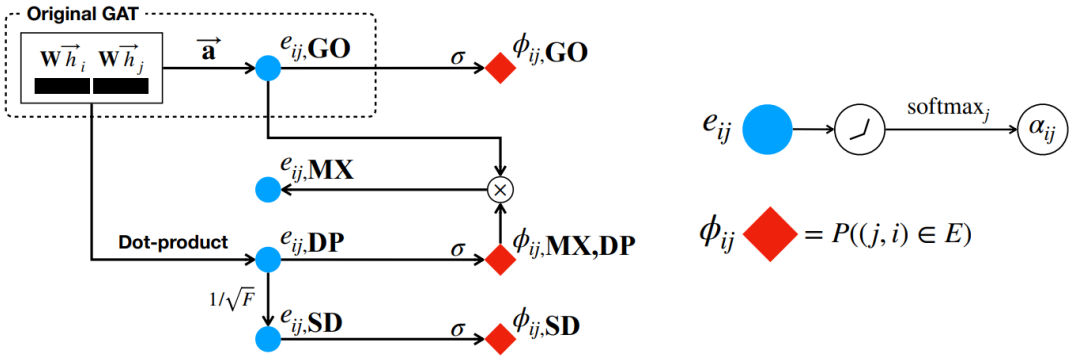


Figure 5.4: *An illustration from Kim & Oh (2022) showing the attention mechanism of SuperGATs (GO, DP, MX, and SD). Blue circles denote the unnormalized attention values before the softmax function, while red diamonds show the edge probability between nodes i and j. The dashed rectangle contains the attention mechanism of the original GAT (Veličković et al. (2018)).*

This equation is part of the graph attention mechanism, which updates the feature vector of a node by aggregating features from its neighbors and itself, weighted by attention coefficients. The self-loop term $\alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i$ considers the contribution of node i's own features to its updated feature vector. The feature vector $\mathbf{x}_i$ is first transformed by the weight matrix $\mathbf{\Theta}$, and then weighted by the attention coefficient $\alpha_{i,i}$.

The neighborhood aggregation term $\sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j$ aggregates the features of all neighboring nodes j of node i. Each neighbor's feature vector $\mathbf{x}_j$ is transformed by the same weight matrix $\mathbf{\Theta}$, then weighted by the corresponding attention coefficient $\alpha_{i,j}$, and summed up. In essence, this equation describes how a node i updates its feature vector by considering both its own transformed features and a weighted combination of the transformed features of its neighbors, with the weights determined by the attention mechanism.

The two types of attention $\alpha_{i,j}^{\text{MX or SD}}$ are computed as:

$$
\begin{aligned}
\alpha_{i,j}^{\text{MX or SD}} &= \frac{\exp\left(\text{LeakyReLU}\left(e_{i,j}^{\text{MX or SD}}\right)\right)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}}\exp\left(\text{LeakyReLU}\left(e_{i,k}^{\text{MX or SD}}\right)\right)} \\
e_{i,j}^{\text{MX}} &= \mathbf{a}^{\top}[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\cdot\sigma\left(\left(\mathbf{\Theta}\mathbf{x}_i\right)^{\top}\mathbf{\Theta}\mathbf{x}_j\right) \\
e_{i,j}^{\text{SD}} &= \frac{\left(\mathbf{\Theta}\mathbf{x}_i\right)^{\top}\mathbf{\Theta}\mathbf{x}_j}{\sqrt{d}}
\end{aligned}
\tag{5.27}
$$

In these equations, $\alpha_{i,j}^{\text{MX or SD}}$ denotes the attention coefficient between nodes i and j for either the MX (Mix) or SD (SumDot) attention mechanism. The attention coefficient is computed using a softmax function, where the numerator is the exponential of a leaky ReLU activation function applied to $e_{i,j}^{\text{MX or SD}}$. The denominator normalizes this value by summing over the exponential of the leaky ReLU applied to $e_{i,k}^{\text{MX or SD}}$ for all nodes k in the neighborhood $\mathcal{N}(i)$ of node i including node i itself.

The term $e_{i,j}^{\text{MX}}$ represents the attention energy between nodes i and j for the MX mechanism. It is calculated as the dot product of a learnable weight vector $\mathbf{a}$ and the concatenation of the transformed feature vectors $\mathbf{\Theta}\mathbf{x}_i$ and $\mathbf{\Theta}\mathbf{x}_j$. This concatenation is denoted by $\|$. The result is then multiplied by the sigmoid activation function σ applied to the dot product of the transformed feature vectors $(\mathbf{\Theta}\mathbf{x}_i)^{\top}\mathbf{\Theta}\mathbf{x}_j$.

The term $e_{i,j}^{\text{SD}}$ represents the attention energy between nodes i and j for the SD mechanism. It is calculated as the dot product of the transformed feature vectors $(\mathbf{\Theta}\mathbf{x}_i)^{\top}\mathbf{\Theta}\mathbf{x}_j$ divided by the square root of the dimensionality of the feature vectors, $\sqrt{d}$.

Overall, these equations describe how the attention coefficients $\alpha_{i,j}^{\text{MX or SD}}$ are computed, which determine the importance of the features of neighboring nodes j when updating the feature vector of node i. The two types of attention mechanisms, MX and SD, use different methods to compute the attention energies $e_{i,j}^{\text{MX}}$ and $e_{i,j}^{\text{SD}}$.

The self-supervised task is a link prediction using the attention values as input to predict the likelihood $\phi_{i,j}^{\text{MX or SD}}$ that an edge exists between nodes:

$$
\begin{aligned}
\phi_{i,j}^{\text{MX}} &= \sigma\left(\left(\mathbf{\Theta}\mathbf{x}_i\right)^{\top}\mathbf{\Theta}\mathbf{x}_j\right) \\
\phi_{i,j}^{\text{SD}} &= \sigma\left(\frac{\left(\mathbf{\Theta}\mathbf{x}_i\right)^{\top}\mathbf{\Theta}\mathbf{x}_j}{\sqrt{d}}\right)
\end{aligned}
\tag{5.28}
$$

5.7 Training Process

This section provides a detailed overview of each stage in the training pipeline, emphasizing the critical techniques that allow iterative refinement of model parameters through a robust and adaptive optimization strategy. Leveraging distributed training across multiple GPUs.

The training process involves iterative weight updates, where we utilize the Adam optimizer, known for its adaptive learning rate capabilities.[1] We begin with an initial learning rate of 1×10^{-4}, which is decayed by a factor of 0.1 every 5 million steps to ensure gradual convergence. The model's performance is evaluated using rollouts, each spanning 90 steps, allowing for systematic adjustment of model parameters and ensuring both convergence and stability throughout the training process.

The primary training loop is implemented in PyTorch within the `train` function. This loop facilitates distributed training across multiple GPUs, where the environment is initialized, metadata is loaded, and the simulator and optimizer are set up. The optimizer's state is transferred to the appropriate device to ensure that gradients and parameters are correctly handled during the distributed training process. The training loop iterates over the data, computing loss, performing backpropagation, updating the optimizer, and logging progress through `wandb`. Both the model and the training state are periodically saved to disk, enabling easy resumption in case of interruptions.

5.7.1 Data Preparation

Data preparation for both experimental setups—the Lid-Driven Cavity Flow and the Curl-Noise Flow Test—follows an identical process to ensure consistency across standard and complex flow scenarios.

In the **Lid-Driven Cavity Flow** experiment, a two-dimensional fluid domain is simulated by applying a parallel force along one boundary to induce shear flow. The domain is discretized into approximately 8,000 particles and a 40×40 grid is used to compute velocity and pressure fields. Key simulation parameters—such as the time step, particle density, grid spacing, and boundary conditions—are detailed in Table 5.2. The flow regime is predominantly laminar,

[1] The Adam optimizer adjusts learning rates for each parameter based on estimations of first and second moments of the gradients. This adaptation helps to stabilize updates and improve convergence characteristics, particularly in complex models and large data scenarios. Kingma & Ba (2014)

with Reynolds numbers estimated between 0.2 and 40. A total of 48 simulation scenarios are generated, each spanning 600 frames, yielding time-series data (positions, velocities, and accelerations) stored in HDF5 format.

Similarly, the **Curl-Noise Flow Test** experiment generates more complex flow conditions. Divergence-free velocity fields are initialized using Curl Noise, computed as the curl of the gradient of a Perlin noise function Robert Bridson & Nordenstam (2007), Perlin (1985, 2002). This method produces smooth, spatially varying vorticity fields that respect solid boundaries, resulting in datasets that capture intricate and dynamic flow behaviors and challenge the surrogate model to accurately conserve vorticity and simulate complex dynamics.

For both experiments, the preprocessing steps are identical:

- **File Processing:** Read and sort HDF5 files by frame number.
- **Data Extraction:** Extract position data from each file and compute velocities and accelerations using finite differences between consecutive frames.
- **Normalization:** Calculate running statistics (means and standard deviations) for the extracted data to enable normalization.
- **Data Storage:** Save the processed data as NPZ files, divided into training (48 scenarios), validation (6 scenarios), and test sets (6 scenarios), along with accompanying metadata (see Table 5.3).

5.7.2 Metadata and Data Normalization

The metadata stored in the `metadata.json` file includes critical statistical information about the dataset, such as the bounds of the simulation domain, sequence length, and connectivity radius (particles neighbourhod radius). Most importantly, it provides the precomputed means and standard deviations for positions, velocities, and accelerations, which are essential for normalizing the dataset before training the neural network. These statistics are summarized in Table 5.1 below:

- **Position Statistics**: The position means and standard deviations indicate that the particle positions are centered near the middle of the simulation domain, with a spread that is relatively uniform across both dimensions.

Table 5.1: *Statistical Summary Data Set for Vorticity Conservation Model*

Statistic	Mean Value	Standard Deviation
Position (x, y)	(0.499, 0.4999)	(0.272, 0.272)
Velocity (x, y)	(-1.8180e-05, 4.789e-07)	(1.600-03, 1.6e-03)
Acceleration (x, y)	(-3.116e-09, 7.532-08)	(7.334e-05, 7.315e-05)

- **Velocity Statistics**: The small mean values for velocity suggest that the overall flow is well-balanced with minor deviations, while the standard deviations indicate the typical range of velocity fluctuations in the simulation.

- **Acceleration Statistics**: The acceleration values, both in mean and standard deviation, are quite small, reflecting the subtle forces acting on the particles due to the simulated fluid dynamics.

These statistics are vital for ensuring that the neural network training process operates on data that is normalized, promoting faster convergence and more stable learning. In our implementation, normalization is applied to both positions and velocities using the precomputed mean and standard deviation values stored in `metadata.json`. The normalization is primarily performed in the `_encoder` and `normalize_sequences` methods. In `normalize_sequences`, both the position and velocity sequences are normalized using the statistics provided in `self._normalization_stats`. Specifically, the position sequence is normalized by subtracting the mean and dividing by the standard deviation, while the velocity sequence, computed as the finite difference between subsequent positions, is normalized in a similar manner. After the model predicts the normalized outputs (either velocity or acceleration), these values are converted back to their original scale in the `_decoder` method. This inverse normalization is crucial for obtaining the final physical quantities from the model's predictions. By integrating normalization in both the preprocessing and postprocessing stages, the model benefits from more stable and efficient training, as the input features are on a consistent scale. This approach also improves generalization, as the model becomes less sensitive to the scale of the inputs, allowing it to focus on learning the underlying patterns in the data.

5.7.3 Initialization

Distributed training is initialized across multiple GPUs. The optimizer state is transferred to the specified device to ensure that the optimizer's parameters and gradients are moved correctly during training.

1. The distributed training environment is initialized (with `distribute.setup`).

2. Dataset metadata is read from the specified dataset path.

3. The Surrogate simulator model is initialized with the given hyperparameters (given metadata, noise standard deviations, device, model type, and number of MLP layers). The input features are normalized, and the simulator is set up with the specified parameters (See Table **??**).

4. The simulator is wrapped with `torch.nn.parallel.DistributedDataParallel` to enable distributed training.

5. The optimizer is initialized with the learning rate scaled by the number of processes. [2]

6. The logger is initialized only on the primary process (rank 0 GPU), logging the training progress if specified in the flags.

7. Training data is loaded using a distributed data loader (`distribute.get_data_distributed_dataloader_by_samples`), ensuring each process gets a different subset of the data.

[2]The optimizer is initialized with a learning rate that is scaled by the number of processes. This adjustment is necessary to maintain the effective learning rate during distributed training. In a distributed setup, each process (e.g., each GPU) computes gradients based on a subset of the data. These gradients are then averaged across all processes before updating the model parameters. Without scaling the learning rate, the effective learning rate would be lower than expected, leading to slower convergence and potentially requiring re-tuning of hyperparameters. By scaling the learning rate by the number of processes, the optimizer ensures that the training dynamics remain consistent regardless of the number of GPUs or processes used. This practice, known as the "linear scaling rule," Vaswani et al. (2017) is critical for maintaining stable and efficient training. It compensates for the fact that each process is working on a smaller portion of the data, thus preserving the magnitude of gradient updates and avoiding issues like gradient underflow. This adjustment allows for effective utilization of hardware resources and ensures that the model converges as it would in a single-process training setup.

5.7.4 Loss Functions

The total loss is composed of several components, each targeting a specific aspect of the simulation. Let N denote the number of **kinematic** particles (i.e., particles that are free to move, not fixed boundary particles), and let D be the dimensionality (2D or 3D). The predicted and target positions of particle i are $\mathbf{p}_i^{\text{pred}}$ and $\mathbf{p}_i^{\text{target}}$, while $\mathbf{v}_i^{\text{pred}}$ and $\mathbf{v}_i^{\text{target}}$ are the corresponding velocities. Similarly, $\mathbf{a}_i^{\text{pred}}$ and $\mathbf{a}_i^{\text{target}}$ are the predicted and target accelerations. The predicted density $\mathbf{d}^{\text{pred}}$ and target density $\mathbf{d}^{\text{target}}$ are computed on a grid, as are the predicted and target divergence fields $\nabla \cdot \mathbf{v}^{\text{pred}}$ and $\nabla \cdot \mathbf{v}^{\text{target}}$, and the predicted and target vorticities ω^{pred} and ω^{target}.

Throughout, $\| \cdot \|_p$ can represent either the L2 norm (mean squared error, MSE) or the L1 norm (mean absolute error, MAE), depending on the chosen training configuration. **In our implementation, we select L2 (MSE) because it provides higher sensitivity to large errors, penalizing significant deviations more heavily.**

1. Acceleration Loss ($\mathcal{L}_{\text{acc}}$)

$$\mathcal{L}_{\text{acc}} = \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{a}_i^{\text{pred}} - \mathbf{a}_i^{\text{target}}\|_p \tag{5.29}$$

Acceleration is a key factor in capturing dynamic changes. By penalizing the discrepancy between predicted and target accelerations, the model is encouraged to learn correct force and motion relationships.

2. Velocity Loss ($\mathcal{L}_{\text{vel}}$)

$$\mathcal{L}_{\text{vel}} = \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{v}_i^{\text{pred}} - \mathbf{v}_i^{\text{target}}\|_p \tag{5.30}$$

Velocity is fundamental to fluid or particle transport. Minimizing this loss aligns flow speeds and directions with the ground truth.

3. Position Loss ($\mathcal{L}_{\text{pos}}$)

$$\mathcal{L}_{\text{pos}} = \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{p}_i^{\text{pred}} - \mathbf{p}_i^{\text{target}}\|_p \tag{5.31}$$

Position error is often useful when tracking particle locations or ensuring that fluid front evolution is accurate.

4. Density Loss ($\mathcal{L}_{\text{dens}}$) After projecting the particle velocities and masses onto a grid, the predicted density $\mathbf{d}^{\text{pred}}$ and target density $\mathbf{d}^{\text{target}}$ are compared:

$$\mathcal{L}_{\text{dens}} = \|\mathbf{d}^{\text{pred}} - \mathbf{d}^{\text{target}}\|_p \tag{5.32}$$

The norm is computed over all grid points. This helps ensure that the model correctly represents the distribution of mass or particles in space.

5. Divergence Loss ($\mathcal{L}_{\text{div}}$) To compute divergence, central finite differences are applied on the velocity grid:

$$\mathcal{L}_{\text{div}} = \|\nabla \cdot \mathbf{v}^{\text{pred}} - \nabla \cdot \mathbf{v}^{\text{target}}\|_p \tag{5.33}$$

Penalizing divergence encourages realistic mass conservation. For incompressible flows, minimizing divergence is especially crucial to prevent artificial sources or sinks in the fluid.

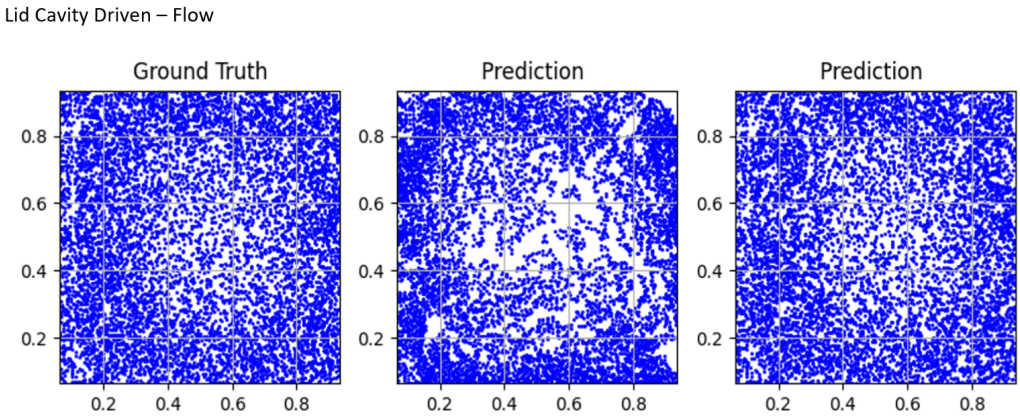


Figure 5.5: *Comparison of a driven cavity flow simulation: (Left) Ground truth, (Middle) Prediction without vorticity loss, and (Right) Prediction with vorticity loss.*

6. Vorticity Loss ($\mathcal{L}_{\text{vor}}$) Vorticity quantifies the local spinning or swirling motion of a fluid, and capturing this rotational flow is essential for physically realistic simulations. We compute the vorticity field using central finite differences on the velocity grid:

$$\mathcal{L}_{\text{vor}} = \|\omega^{\text{pred}} - \omega^{\text{target}}\|_p. \tag{5.34}$$

Introducing a vorticity term in the loss function helps preserve these rotational features. Figure 5.5 highlights the importance of this component: the middle panel shows results from

a baseline GNS model without vorticity loss, while the right panel includes the vorticity term. In the baseline model, the flow develops fewer or weaker rotational structures compared to the ground truth (left). By contrast, adding vorticity loss yields significantly better alignment with the reference flow, indicating that the model now captures swirling behaviors more accurately. This not only enhances visual fidelity but also leads to more physically consistent dynamics.

Total Loss ($\mathcal{L}_{\text{total}}$)

$$\mathcal{L}_{\text{total}} = w_{\text{acc}} \cdot \mathcal{L}_{\text{acc}} + w_{\text{vel}} \cdot \mathcal{L}_{\text{vel}} + w_{\text{pos}} \cdot \mathcal{L}_{\text{pos}} + w_{\text{dens}} \cdot \mathcal{L}_{\text{dens}} + w_{\text{div}} \cdot \mathcal{L}_{\text{div}} + w_{\text{vor}} \cdot \mathcal{L}_{\text{vor}} \quad (5.35)$$

Here, $w_{\text{acc}}, w_{\text{vel}}, w_{\text{pos}}, w_{\text{dens}}, w_{\text{div}}, w_{\text{vor}}$ are user-defined weights that control the relative importance of each term.

Defining and Adjusting Loss Weights

In practice, choosing weightings $(w_{\text{acc}}, w_{\text{vel}}, \dots)$ determines how strongly each physical property influences the model updates. For example, increasing the vorticity weight (w_{vor}) penalizes errors in rotational flow more strongly and tends to produce more stable velocity fields, while increasing divergence weight (w_{div}) helps enforce near-incompressibility or mass conservation. Emphasizing velocity, acceleration, or position fosters more accurate dynamic predictions but can lessen the focus on global flow characteristics if not balanced properly.

During an initial training stage, the *raw sizes* of each loss term (before weighting) may vary significantly. For example, one might see:

$$\mathcal{L}_{\text{acc}} \approx 0.7, \quad \mathcal{L}_{\text{vel}} \approx 0.1, \quad \mathcal{L}_{\text{pos}} \approx 3 \times 10^{-9}, \quad \mathcal{L}_{\text{dens}} \approx 0.03, \quad \mathcal{L}_{\text{div}} \approx 1.5, \quad \mathcal{L}_{\text{vor}} \approx 1.6.$$

Observing these magnitudes can help guide decisions on how heavily to weight each term. If, for instance, the vorticity or divergence losses are intrinsically higher, boosting those weights might be beneficial.

In many practical situations, particularly when there is limited time for hyperparameter tuning, assigning equal weights to every component (i.e., setting $w_{\text{acc}} = w_{\text{vel}} = \cdots = 1$) often provides a balanced solution. For example, you might set all of the following to 1:

$$\text{vorticity\_loss\_weight} = 1, \quad \text{divergence\_loss\_weight} = 1, \quad \text{density\_loss\_weight} = 1,$$

$$\text{velocity\_loss\_weight} = 1, \quad \text{position\_loss\_weight} = 1, \quad \text{acceleration\_loss\_weight} = 1.$$

This ensures that each loss component is penalized equally during training.

During training, each component's raw loss is logged periodically. Below is an example from one training run at step 507, which showed an Acceleration Loss (3.566×10^{-2}), Velocity Loss (2.56×10^{-4}), Density Loss (1.708×10^{-2}), Position Loss (almost zero), Divergence Loss (1.209×10^{-1}), and Vorticity Loss (1.278×10^{-1}), summing to a total of 3.016×10^{-1}. As training continued, those losses generally decreased further; by a later snapshot, the overall Train Loss was about 6.443×10^{-1}, with the Acceleration, Velocity, Density, Position, Divergence, and Vorticity Losses each having also lowered in value.

Even with uniform weighting, the network can learn to balance all aspects—acceleration, velocity, density, position, divergence, and vorticity—sufficiently well for a physically consistent simulation. Adjusting individual weights more finely could further improve specific facets (e.g., mass conservation or rotational fidelity), but doing so might overemphasize one component at the expense of others if not done carefully.

Summary. By combining acceleration, velocity, position, density, divergence, and vorticity losses into one framework—and either carefully or uniformly weighting each term—the model can capture both fine-grained particle motion and large-scale fluid properties. Tracking raw magnitudes of each loss component and logging their values during training helps ensure that no single aspect is neglected. This integrated approach ultimately yields more accurate and physically plausible simulations for a wide range of fluid and particle dynamics problems.

5.8 2D Lid Driven Cavity Flow Test

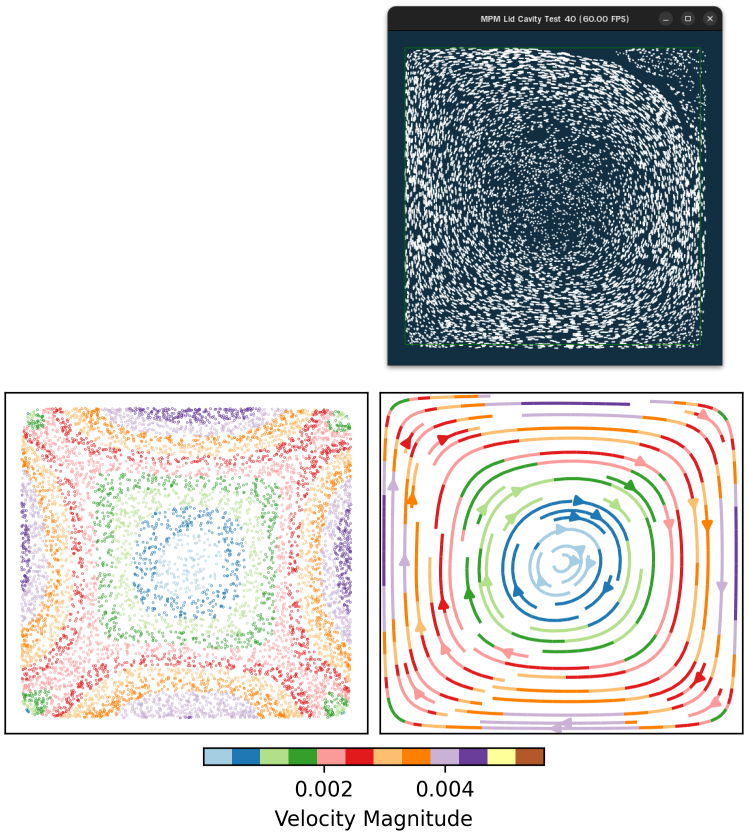


Figure 5.6: *2D Lid Cavity Simulation. Top Right: weakly compressible MPM Lid Cavity Test, Bottom: Particle Velocity field vs Streamline of 2D Lid Cavity flow dataset. Flow patterns reveal the formation of primary and secondary vortices, including the main vortex driven by the lid and any smaller vortices near the corners of the cavity. The velocity field provides the distribution of velocity components u and v within the cavity*

Problem Setting The first step to examine the vorticity conservation performance of the surrogate models, we use the 2D lid-driven cavity flow simulation as a test case. The 2D lid-driven cavity flow is a classic benchmark problem in computational fluid dynamics (CFD) used to study the behavior of fluid flow in a closed cavity. The physical setup consists of a square cavity where the fluid is enclosed. The boundaries of this cavity include three stationary walls (bottom, left, and right) and a top wall that moves horizontally. The top wall, also known as the lid, drives the flow by moving at a constant velocity from left to right.

For the boundary conditions, the top wall has a specified horizontal velocity $u = U_{\text{lid}}$ and no vertical velocity $v = 0$. The other three walls (bottom, left, and right) are stationary, enforcing a no-slip condition where $u = 0$ and $v = 0$, meaning there is no relative motion between the fluid and these walls.

The continuity equation ensures mass conservation and is given by:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{5.36}$$

The momentum equations describe the conservation of momentum in the fluid. For the horizontal (x-direction) momentum, the equation is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{5.37}$$

For the vertical (y-direction) momentum, the equation is:

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{5.38}$$

where u and v are the velocity components in the x and y directions, respectively, p is the pressure, ρ is the fluid density (constant for incompressible flow), and ν is the kinematic viscosity.

The main objectives of simulating the 2D lid-driven cavity flow are to analyze and understand various fluid flow characteristics such as the velocity field, vorticity distribution, pressure distribution, and flow patterns. The velocity field provides the distribution of velocity components u and v within the cavity. The vorticity distribution, which measures the local rotation of the fluid, helps identify regions of rotational flow. The pressure distribution shows the variation of pressure within the cavity. Additionally, flow patterns reveal the formation of primary and secondary vortices, including the main vortex driven by the lid and any smaller vortices near the corners of the cavity, see Figure 5.6.

5.8.1 Dataset Generation

In the two-dimensional fluid setting, a parallel force is applied from one side of the boundary over a specified time duration. This force is exerted parallel to the surface of the fluid from the boundary edge, influencing the motion and behavior of the fluid within the defined two-dimensional space.

To simulate the 2D lid-driven cavity flow, the domain is discretized into 8000 particles and 40 by 40 square background grid of cells where the velocity and pressure fields are computed Using The Material point method (2.3.2). This grid has boundaries where forces can be applied. To accurately simulate fluid behavior, properties such as density and viscosity are defined and appropriate initial and boundary conditions are established.

The parallel force is applied to one of the boundaries of the fluid domain. The force can be modeled either as a constant or as a time-varying function, depending on the specific simulation scenario. This applied force induces shear flow, which initiates fluid motion.

The parameters for the fluid setting are provided in Table 5.2.

Table 5.2: *MPM Parameters for the Fluid Setting*

Parameter	**Value**
Number of particles ($n_{\text{particles}}$)	8192
Number of grid cells (n_{grid})	40
Grid spacing (dx)	$\frac{1}{n_{\text{grid}}}$ (dimensionless)
Time step (dt)	2×10^{-3} seconds
Particle density (p_ρ)	0.0001 kg/m^3
Particle volume (p_{vol})	$(dx \times 0.5)^2$ m^2
Particle mass (p_{mass})	$p_{\text{vol}} \times p_\rho$ kg
Gravity	0.1 m/s^2
Boundary thickness (bound)	3 cells (dimensionless)
Modulus (E)	0.0005 Pa (Pascals)
Recording substeps	50 (dimensionless)
Output dimension	2 (spatial dimensions)
Maximum frames	600 (frames)
Maximum substep force (maxsubstepforce)	150,000 + random.randint$(1, 10) \times$ 25,000 N

To further characterize the flow in these experiments, we refer to the Reynolds number as defined in Eq. 2.3. Based on the normalized parameters used in our simulations ($\rho = 1$, a characteristic length $L \approx 1$, and a dynamic viscosity $\mu = 0.5$), the Reynolds number is estimated to vary between approximately 0.2 and 40. This range indicates that, although the flow experiences dynamic changes, the overall regime remains predominantly laminar.

The fluid domain is simulated using the Material Point Method (MPM) algorithm (see Section 2.3.2 for details) with the parameters detailed in the table (5.2). These parameters control the resolution of the simulation, the physical properties of the fluid, and the recording settings. We generate 48 simulations scenario with 600 frames each. The simulation produces time-series data for the position, velocity, acceleration in the fluid domain. The visualization of the data sets provided in figure 5.7 (streamline) and figure 5.8 (Particles). The data set consists of *positions* and *opt.attr*. we store all data per frame in the hdf5 file format.

To further characterize the flow in these experiments, we refer to the Reynolds number as defined in Eq. 2.3. Based on the normalized parameters used in our simulations (with $\rho = 1$, a characteristic length $L \approx 1$, and a dynamic viscosity $\mu = 0.5$), the Reynolds number is estimated to vary between approximately 0.2 and 40. This range indicates that, although the flow experiences dynamic changes, the overall regime remains predominantly laminar.

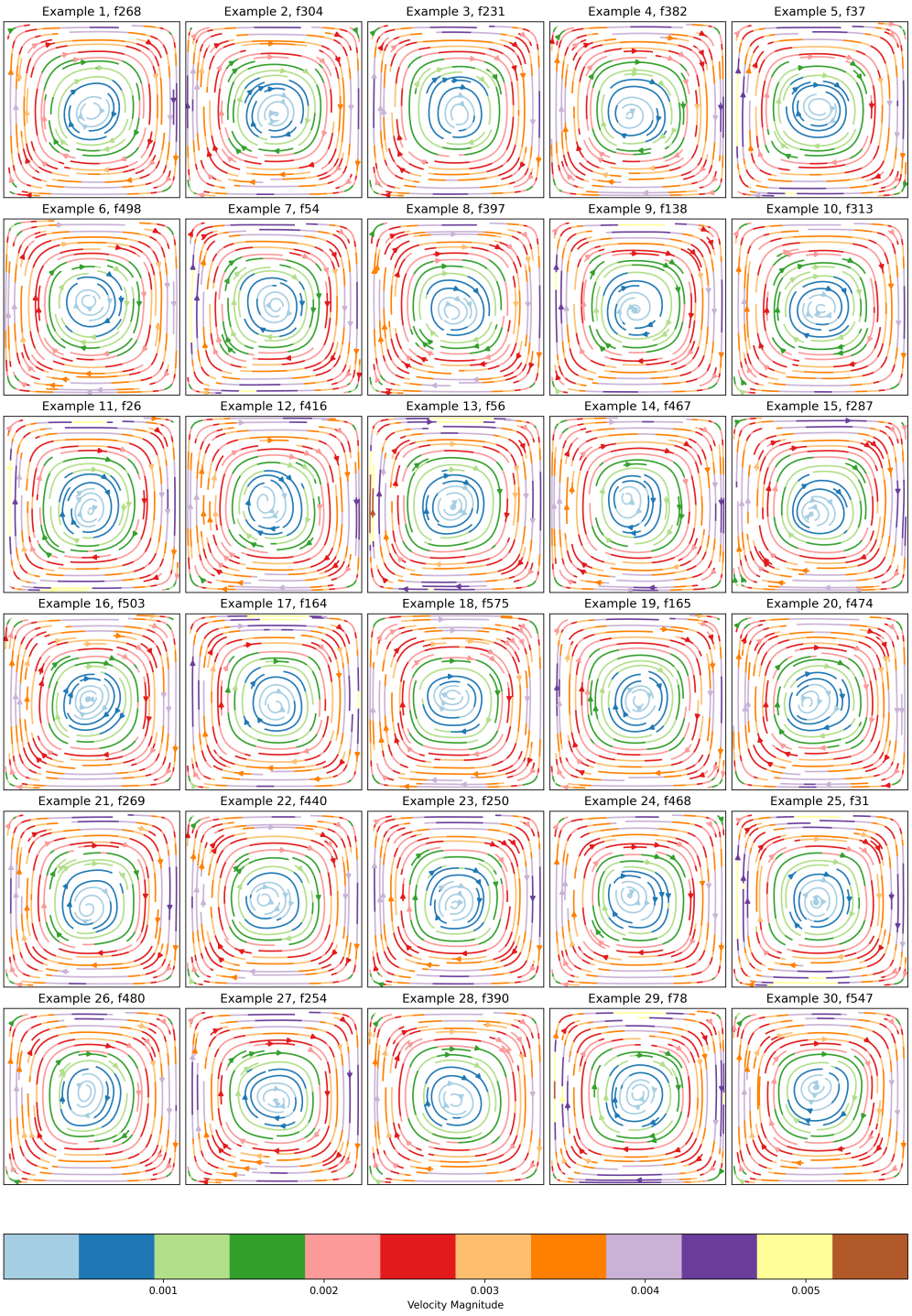


Figure 5.7: *Visualization of the 30 out of 48 initial conditions used for the training dataset. The simulations here are run using a weakly compressible Material Point Method (MPM) model with an initial velocity field under solid boundary conditions outside the shown areas. Color mapping indicates velocity magnitude, and streamlines illustrate the flow field at frame f of certrain secario (example), computed by mapping the particle velocities to a regularly sampled grid using cubic spline interpolants.*

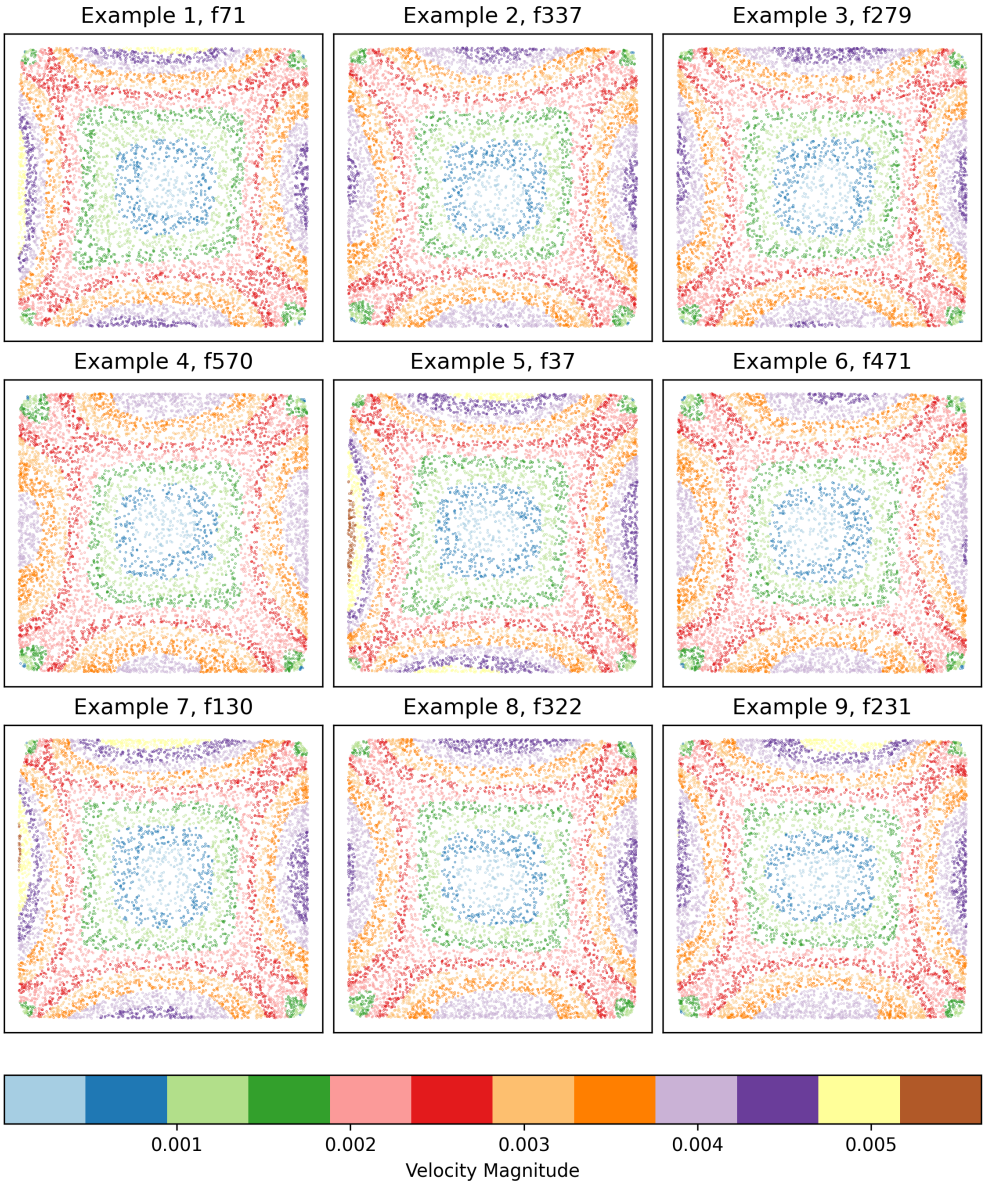


Figure 5.8: *Visualization of the 9 out of 48 initial conditions used for the training dataset. Particle indicates position while Color mapping indicates velocity magnitude*

Dataset Preprocessing

Data set pre-processing begins by reading all HDF5 files. It extracts frame numbers from the filenames and sorts them accordingly. The position data are then read from each HDF5 file into a position array.

The next step involves initializing variables to manage running statistics and setting up dictionaries for running sums, sums of squares, counts, and concatenated data. These dictionaries are used to compute statistics for velocities and accelerations based on the dimensionality of the input data, which can be 2D or 3D.

Key	**Value**
boundary	[[0.065, 0.935], [0.065, 0.935]]
sequence length"	600
neighbour radius"	0.015
dimension	2
dt	1
velocity mean"	[-3.5108e-05, -2.6359e-06]
acceleration mean	[8.5449e-10, 1.3844e-07]
velocity std	[0.002191, 0.002195]
acceleration std"	[9.9393e-05, 9.9605e-05]

Table 5.3: *Statistic of the 2D Lid cavity flow dataset*

For each directory containing HDF5 files, the process reads and extracts the position data. The files are sorted by frame number, and the position data from each file is read into an array. Velocity and acceleration are then calculated using differences in positions between consecutive substeps. The function returns the processed data for further analysis and updates the running statistics by computing sums, sums of squares, and counts for velocities and accelerations while concatenating the data. The running statistics dictionaries are updated with the new data. Finally, statistics across all trajectories are calculated and printed, including the mean and standard deviation for each dimension of velocity and acceleration.

The processed trajectory data are then saved to an NPZ file using *numpy.savez_compressed*. The NPZ file is divided into three files: *train.npz* (48 scenarios), *valid.npz* (6 scenarios), and *test.npz* (6 scenarios). Each NPZ file consists of a collection of tuples of varying length,

each tuple representing a unique training trajectory and containing positional information. This NPZ file is ready for use in training. The metadata file containing dataset and statistic information is stored as illustrated in Table 5.3:

5.8.2 Results

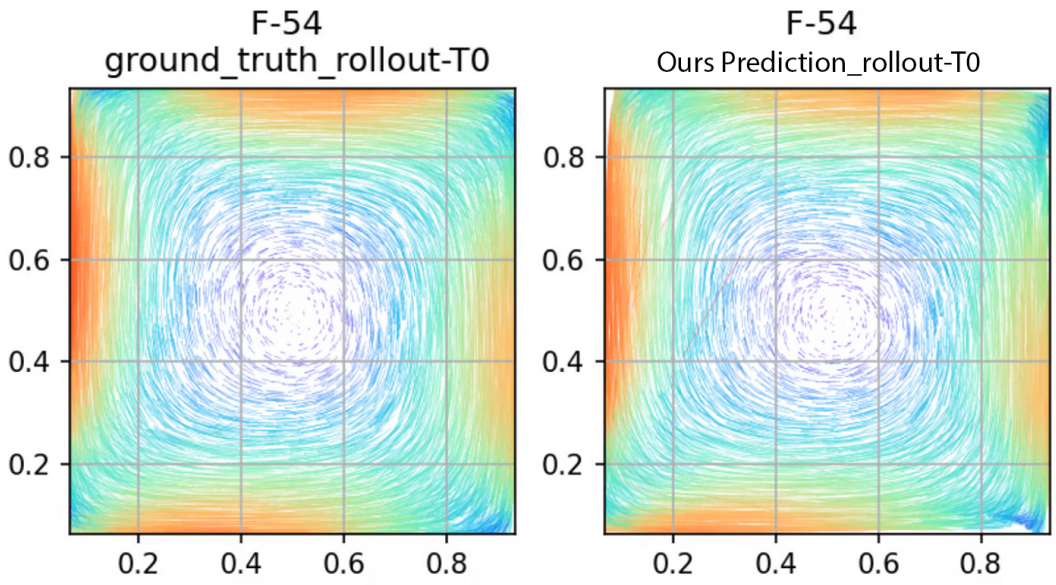


Figure 5.9: *Velocity field predicted by our proposed model for frame 54, showcasing stability over extended rollouts*

In this section, we present both quantitative and qualitative results to provide a comprehensive evaluation of our vorticity prediction model. For quantitative analysis, we used several metrics to evaluate the performance of our model, including the mean squared error (MSE), mean absolute error (MAE), and an Energy Conservation Plot. To provide a visual understanding of the model's performance, we include several qualitative assessments (Predicted vs Actual Vorticity Plot, Residuals Plot, Temporal Evolution of Vorticity). These results will help to understand the performance and effectiveness of our model.

Results on Various Test Sets

Figure 5.10 illustrates the performance of the *_OURS* model in conserving vorticity over long-range predictions, measured by the mean squared error (MSE). The chart includes six curves representing six different test cases (T0 to T5).

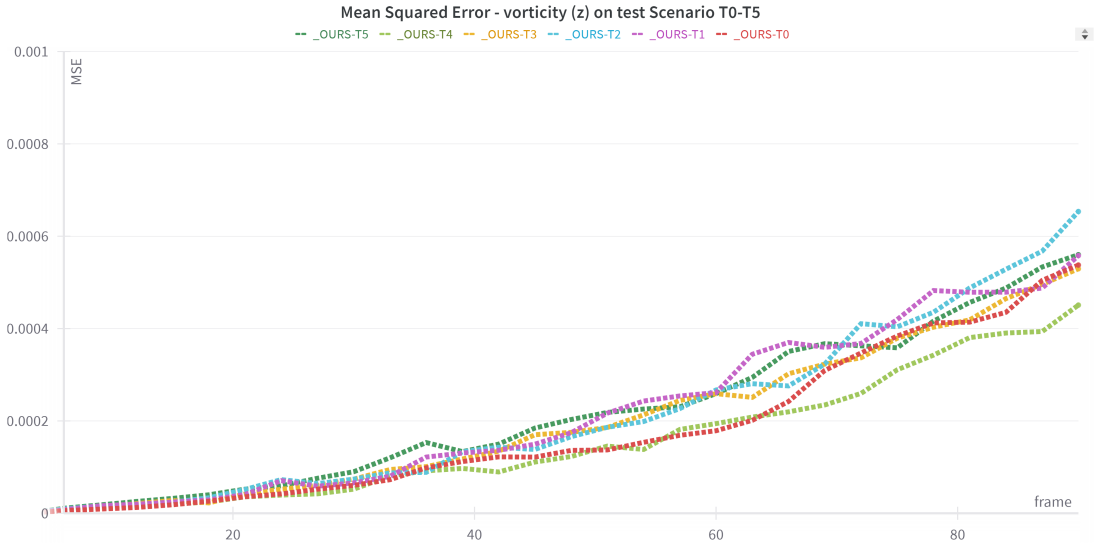


Figure 5.10: *MSE of long term prediction using 6 different test case on OURS model*

The MSE for all test cases starts very low, close to 0, and gradually increases as the number of frames progresses. In particular, the MSE remains below 0.0002 for the first 40 frames in all test cases, indicating stable performance in short-term predictions. An important observation is that MSE grows almost linearly over time for each test scenario. This almost-linearity implies a steady rate of error build-up as the frame count rises. After surpassing 50 frames, the MSE continues to climb, emphasizing the challenge of achieving long-term prediction accuracy. However, the model keeps the MSE increase relatively controlled, with values remaining below 0.0004 up to frame 80. Each test case shows minor differences in MSE values, indicating some variations depending on the particular test scenario, yet the overarching pattern stays stable. These minor variations suggest that the model has the potential to generalize across various scenarios. Essentially, the model's performance isn't significantly influenced by the specific test case, indicating its robustness and dependability in diverse conditions.

Benchmark: Long-Range Predictions Stability

Figure 5.11: *MSE benchmark comparing the performance of 6 models over long-range prediction. The red color represents our model, showing stability. During intermediate frames (10-40), _OURS maintains a consistently lower MSE compared to* $\_graph_network - T0$*. The latter shows occasional spikes and higher error variability, indicating less stability in its predictions.*

In the comparative experiments presented in this section, we kept the overall model, loss function, and training methodology consistent across all models. The key difference lies in how the node features are updated in the Particle Network layer. For the original GNS model, the node embedding update is performed through the original message passing technique used in graph neural networks. In contrast, in the other comparison models, we replace this part with different convolutional operator. Specifically, in (_OURS), we replace the standard propagation method with the SuperGATConvolution Operator, which improves the model's ability to capture local interactions and conserve vorticity. All the models in this comparative study use the overall GNS framework, and the only difference between them lies in the convolutional layer used for node feature updates.

Figure 5.11 displays the mean squared error (MSE) in frames for various models predicting

the vorticity (z-axis) using test scenario T0. The models included in the chart are _GATv2Conv (blue) [Brody et al. (2022)], _graph_transformer (orange) [Shi et al. (2021)], _GNS (Graph Network simulator, purple) [Sanchez-Gonzalez et al. (2020)], _GATConv (yellow) [Veličković et al. (2018)], _pointnet_conv (Pointnet++, green) [Qi et al. (2017)] , and _OURS (red). The x-axis represents the frames, while the y-axis represents the MSE.

In the initial frames (10-30), _GATConv, _GNS and _OURS exhibit low MSE values. However, _OURS starts with a slightly lower MSE, indicating marginally better performance from the beginning.

During intermediate frames (30-40), _OURS maintains a consistently lower MSE compared to _graph_network-T0. The latter shows occasional spikes and higher variability in error, indicating less stability in its predictions.

In the later frames (40-60), _OURS continues to outperform _GNS by maintaining a lower MSE. The gap between the two models becomes more apparent in these frames, highlighting the robustness and accuracy of _OURS over time.

Overall, the _OURS model demonstrates superior performance with consistently lower MSE across all frames, suggesting better predictive accuracy and generalization. On the other hand, the _graph_network-T0 model, while performing reasonably well, exhibits higher errors and variability, particularly in the later frames. This indicates potential issues with long-term predictions or generalization.

Figure 5.11 illustrates that the OURS model consistently achieves lower mean square error (MSE) values in every frame compared to the GNS model. A precise numerical analysis for frame 54 is provided in Table 5.4.

Model	**MAE**	**MSE**	**RMSE**
OURS	0.007715	0.000151	0.012297
GATConv	0.010443	0.000289	0.017006
Pointnet_conv	0.012597	0.000420	0.020484
GNS	0.009757	0.000449	0.021191
GATv2Conv	0.011476	0.000447	0.021138
Graph_transformer	0.014519	0.000724	0.026903

Table 5.4: *Comparison of model predictions with ground truth using MAE, MSE, and RMSE at frame and 54 (This was selected as a typical case).*

Benchmark: Vorticity distribution

To analyze the vorticity distributions of the models with ground truth as the benchmark, we assess the alignment of each model's vorticity distribution with the ground truth on frame 54 as depicted in Figure 5.12. We further explore this using the Pearson coefficient index, residual plot, and KL divergence. Additional histograms for frames 6, 9, 18, 21, 48, and 51 are illustrated in Figure 5.18.

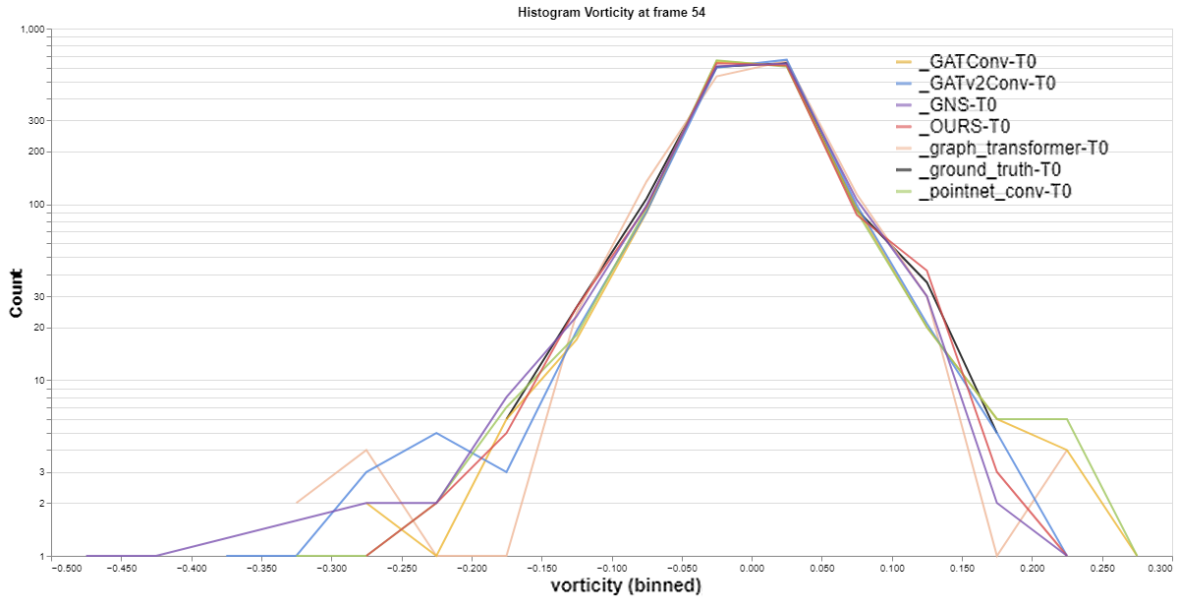


Figure 5.12: *Histogram depicting the vorticity distribution at frames 54. The black lines represent the ground truth vorticity magnitude binned as the reference.*

.

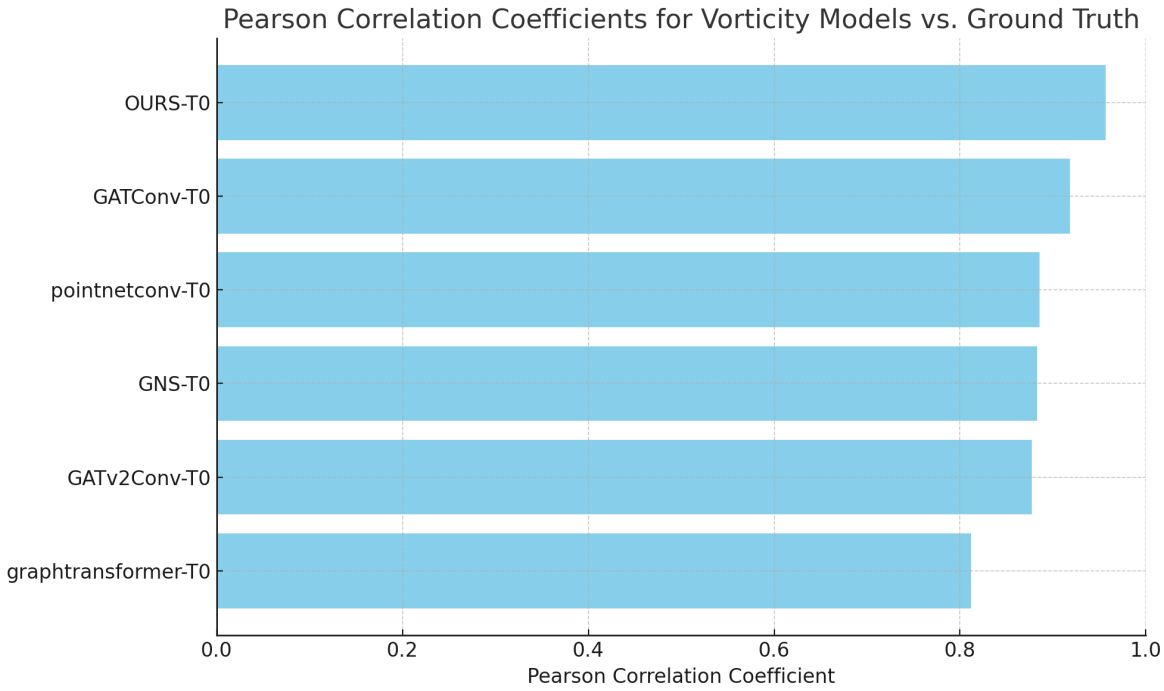
Peak Alignment The **OURS-T0 (Red Line)** aligns well with the ground truth around the peak vorticity values, showing similar peak height and position. Other models also show varying degrees of alignment: **GATConv-T0 (Orange)** and **GATv2Conv-T0 (Blue)** show peaks close to the ground truth but with slight deviations in height and width. The **GNS-T0 (Purple)** and **graph_transformer-T0 (Green)** lines have peaks that align reasonably well with the ground truth but exhibit more noticeable variations. The **pointnet (Yellow)** model has the largest deviation from the ground truth peak, with a wider spread and lower peak height.

Distribution Shape The overall shape of the **OURS-T0 (Red Line)** is quite similar to the ground truth, indicating a good match. Other models also show a similar distribution shape but with some differences in the tails: **GATConv-T0 (Orange)** and **GATv2Conv-T0 (Blue)** align reasonably well but exhibit some differences at the tails. The GNS-T0 (Purple) and graph_transformer-T0 (green) models show more uneven distribution shapes compared to the ground truth, and some bins show higher or lower counts. The **pointnet (Yellow)** model exhibits the largest deviation in shape, with a broader distribution and more significant discrepancies between bins.

Tails (Extreme Values) The tails of the **OURS-T0 (Red Line)** are relatively close to the ground truth, with minor deviations. Other models have tail behavior similar to the ground truth but with some fluctuations: **GATConv-T0 (Orange)** and **GATv2Conv-T0 (Blue)** align reasonably well, but exhibit some fluctuations. The **GNS-T0 (Purple)** and **graph_transformer-T0 (Green)** models show more pronounced differences in the tails, with some bins having higher counts than the ground truth. The **pointnet (Yellow)** model shows the most variation at the tails, with significant differences from the ground truth.

Smoothing and Noise The **OURS-T0 (Red Line)** is relatively smooth, indicating consistent predictions with some noise. Other models also exhibit varying levels of smoothness and noise: **GATConv-T0 (Orange)** and **GATv2Conv-T0 (Blue)** are smooth but with minor noise. The **GNS-T0 (Purple)** and **graph_transformer-T0 (Green)** models exhibit more noise and an uneven distribution, indicating less consistent predictions. The **pointnet (Yellow)** model has the most noise and an uneven distribution, indicating the least consistent predictions.

Analysis of Pearson Correlation Coefficient at frame 54, as illustrated in Table 5.13, reveals the correlation between the vorticity data of each surrogate model and the actual ground truth. The model that aligns most closely with the ground truth is"$\_OURS$", with a correlation coefficient of about 0.957.



Model	Pearson Correlation Coefficient
OURS-T0	0.956670
GATConv-T0	0.918301
pointnetconv-T0	0.885932
GNS-T0	0.883097
GATv2Conv-T0	0.877435
graphtransformer-T0	0.811620

Figure 5.13: *Pearson Correlation Coefficients for Surrogate models and Ground Truth at frame 54. The "OURS" model exhibits the highest correlation.*

The residual vorticity plots for frame 54 as illustrated in Figure 5.14 show that the $OURS-T0$ pattern has relatively minor residuals that are uniformly spread around the zero axis, demonstrating a strong correlation with the actual data.

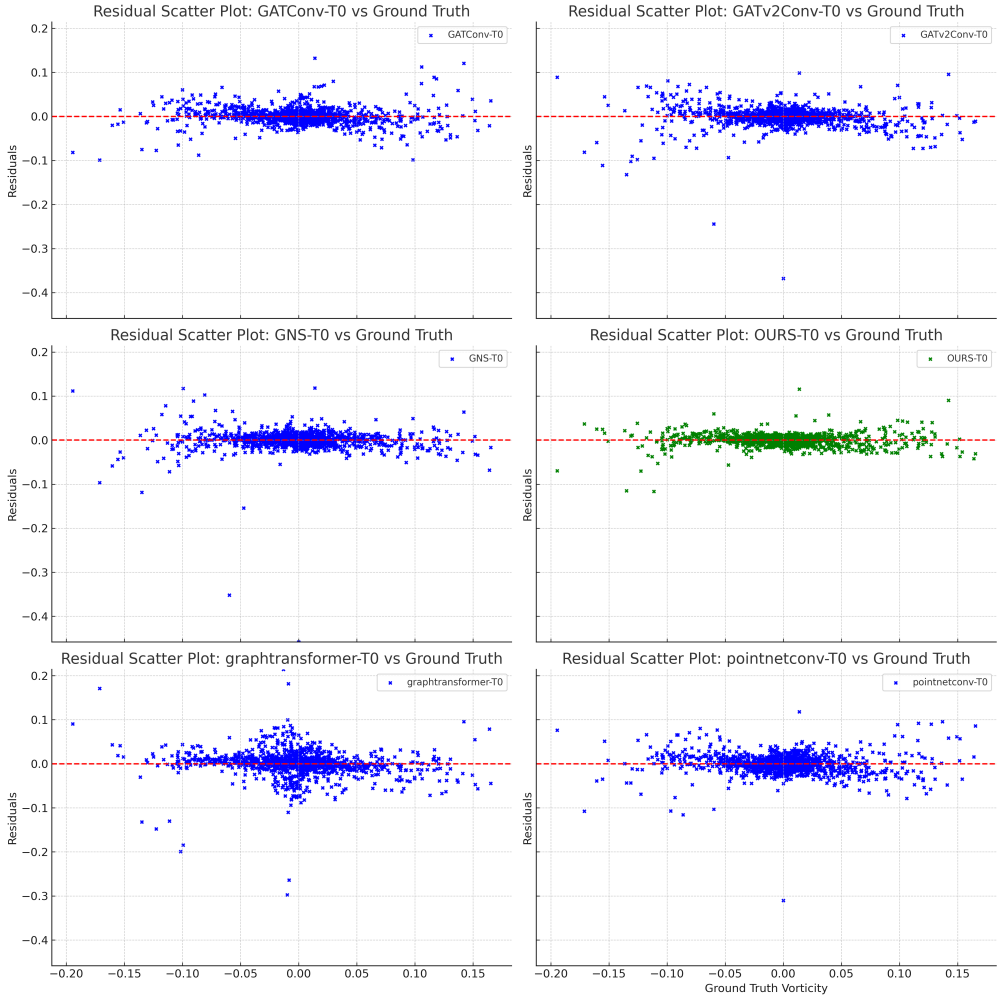


Figure 5.14: *Residual Plots of vorticity of the surrogate models at frame 54. Proposed model (green-colored $OURS-T0$) pattern has relatively minor residuals that are uniformly spread around the zero axis, demonstrating a strong correlation with the actual data*

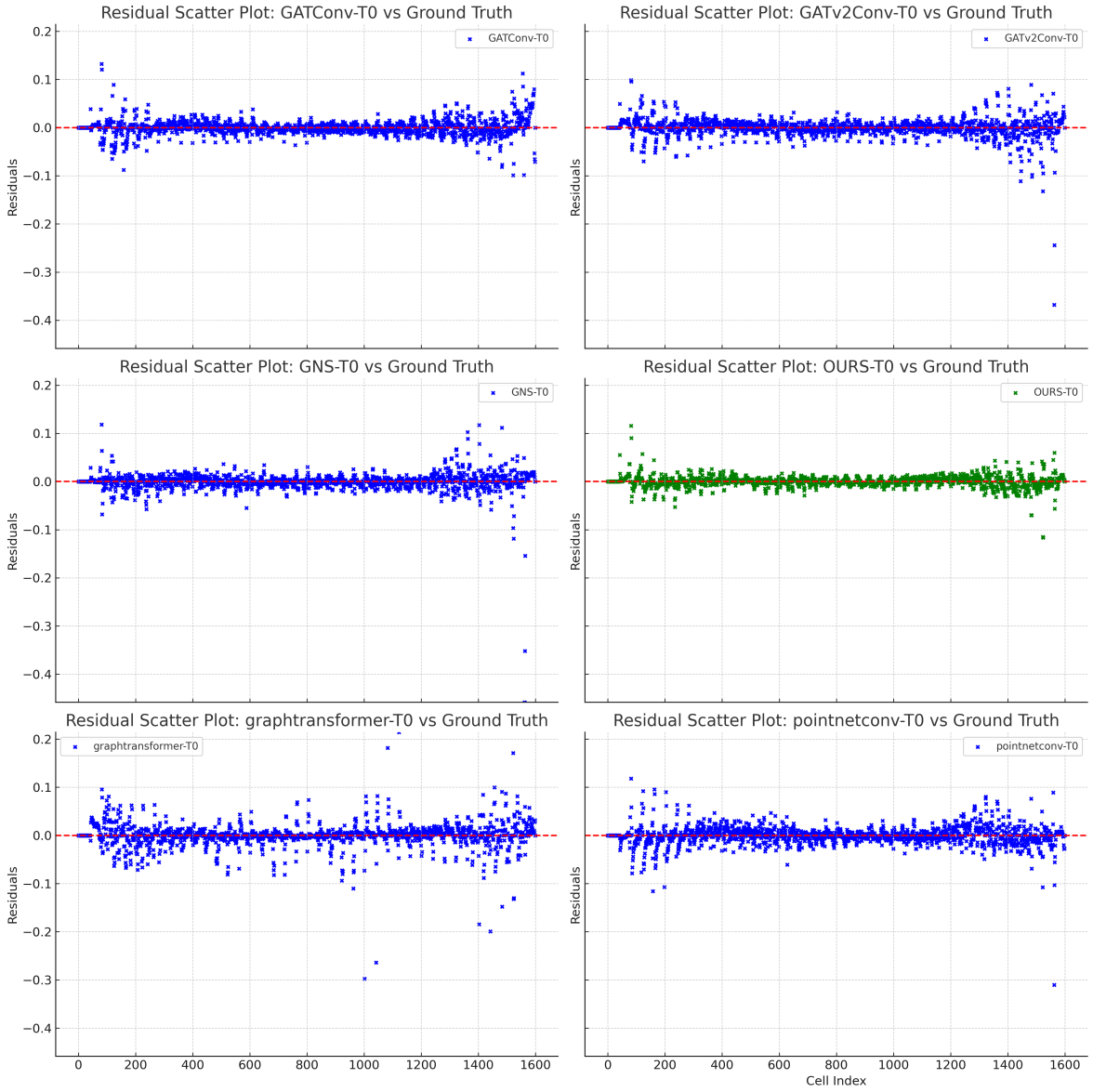


Figure 5.15: *Scatter Plots of Residuals for Vorticity Models vs. Ground Truth with Cell Index (grid dimension: 40x40) on the X-Axis. The model with the lowest residuals is highlighted in green, while other models are shown in blue. The red dashed line represents zero residuals*

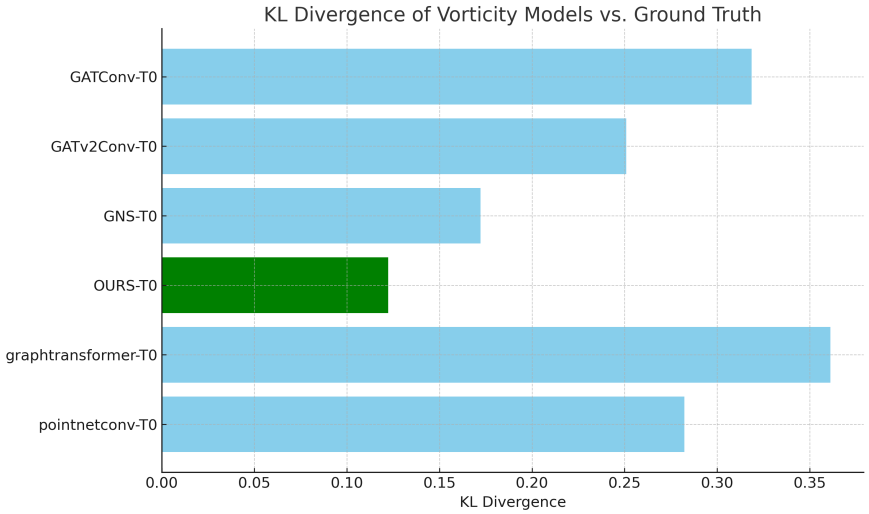


Figure 5.16: *KL Divergence of Surrogate Models Compared to Ground Truth. The model with the lowest KL divergence is highlighted in green, indicating the closest match to the ground truth vorticity distribution*

Model	MAE	MSE	RMSE	KL Divergence
_OURS-T0	0.007715	0.000151	0.012297	0.127755
_GNS-T0	0.009757	0.000449	0.021191	0.172230
_pointnet_conv-T0	0.012597	0.000420	0.020484	0.296943
_GATv2Conv-T0	0.011476	0.000447	0.021138	0.269795
_GATConv-T0	0.010443	0.000289	0.017006	0.354617
_graph_transformer-T0	0.014519	0.000724	0.026903	0.391100

Table 5.5: *KL Divergence of the vorticity of each surrogate model vs the ground truth at frame 54 (This was selected as a typical case)*

Table 5.5 shows that model **_OURS-T0** has the lowest KL divergence, indicating the highest similarity to the ground truth. The low KL Divergence for **_OURS-T0** suggests that this model effectively captures the underlying distribution of the vorticity data, likely benefiting from its nonlinear modeling capabilities. The model's architecture may include advanced nonlinear transformations that allow it to better represent complex relationships within the vorticity data, leading to a closer match with the ground truth.

One of the key motivations for examining the vorticity distribution at **frame 54** rather

than an early frame stems from the long-range rollout context. As shown in Figure 5.11, each surrogate model's mean squared error (MSE) accumulates over time, making later frames such as frame 54 more stringent tests of a model's predictive stability and accuracy. While the MSE trend indicates how quickly or slowly each model's predictions deviate from the ground truth on a pointwise basis, it does not capture whether the *overall shape* or *range* of the predicted vorticity distribution remains physically plausible.

Complementary Role of MSE and KL Divergence: By integrating both MSE and KL divergence metrics, we gain a deeper insight into the models' performance:

- *MSE* pinpoints how closely individual predicted vorticity values match the ground truth across the domain. A lower MSE indicates smaller average discrepancies at each spatial point.

- *KL Divergence* determines how well the *probability distribution* of predicted vorticity magnitudes aligns with the real distribution. Specifically, it evaluates whether the model reproduces the *relative frequencies* of low, medium, and high vorticity magnitudes.

Hence, a model could exhibit a respectable MSE yet still fail to capture the correct *distributional profile*, for example, if it consistently underestimates high-vorticity regions while overestimating moderate-vorticity areas. Conversely, a low KL divergence coupled with high MSE might signal that, despite getting the overall histogram roughly correct, certain regions of the flow field are spatially misaligned. Thus, neither metric alone suffices; but their combination, especially at a later frame like frame 54, provides a more robust portrait of both pointwise accuracy and distributional realism.

Physical Relevance of Vorticity Distributions: In fluid dynamics, the distribution of vorticity across the domain is intimately tied to flow structures such as vortices, shear layers, and turbulent eddies. Capturing the shape of the vorticity histogram (via KL divergence) is crucial for ensuring that the model has learned the fundamental statistics of the flow. The fact that our **_OURS-T0** model retains a relatively *low* MSE over the entire temporal horizon (up to frame 60) *and* achieves the *lowest KL divergence* at frame 54 demonstrates its ability to maintain both pointwise accuracy and global distributional fidelity under long-term rollout conditions.

Why Frame 54 is Instructive: Examining predictions at frame 54 offers a snapshot of *late-stage* behavior, where error accumulation and compounding inaccuracies begin to mani-

fest. A model that accurately predicts both the pointwise values (MSE) and the distribution (KL divergence) at this challenging frame is likely to be robust throughout the simulation. Thus, the low KL divergence at frame 54 serves as a strong indicator that _**OURS-T0** captures the essential statistical properties of the flow—beyond just matching individual vorticity values—better than the other surrogate models.

In summary, the selection of frame 54 for the KL divergence analysis, combined with the long-horizon MSE benchmarks, paints a comprehensive picture of how well each surrogate model maintains both local and global accuracy as the simulation progresses. This dual perspective is invaluable for fluid simulation tasks, where capturing the correct vorticity distribution can be just as critical as reducing overall numerical errors.

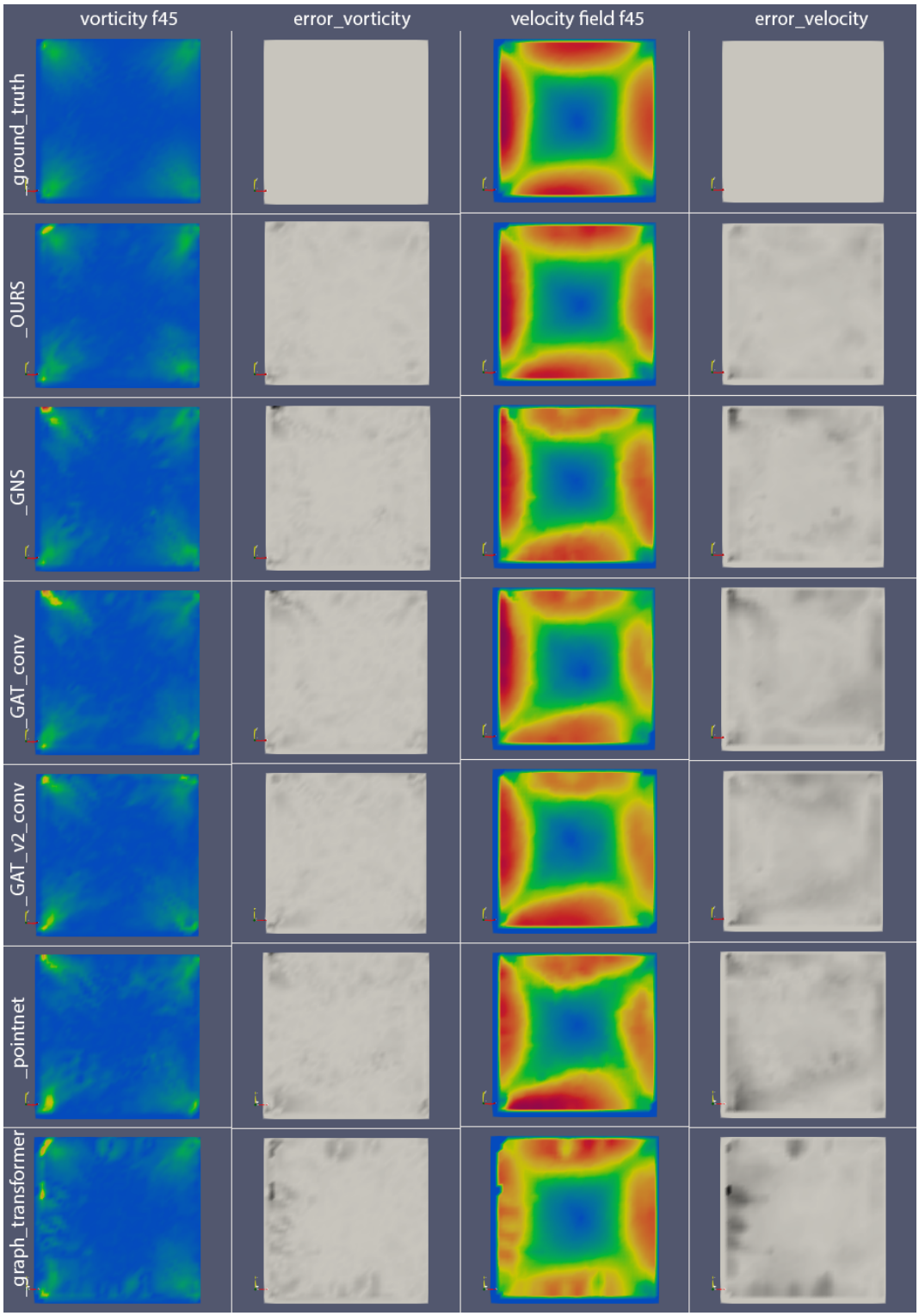


Figure 5.17: *Qualitative comparison of predictions from 6 surrogate models at frame 54. The _ours model demonstrates the lowest error both in vorticity and velocity prediction*

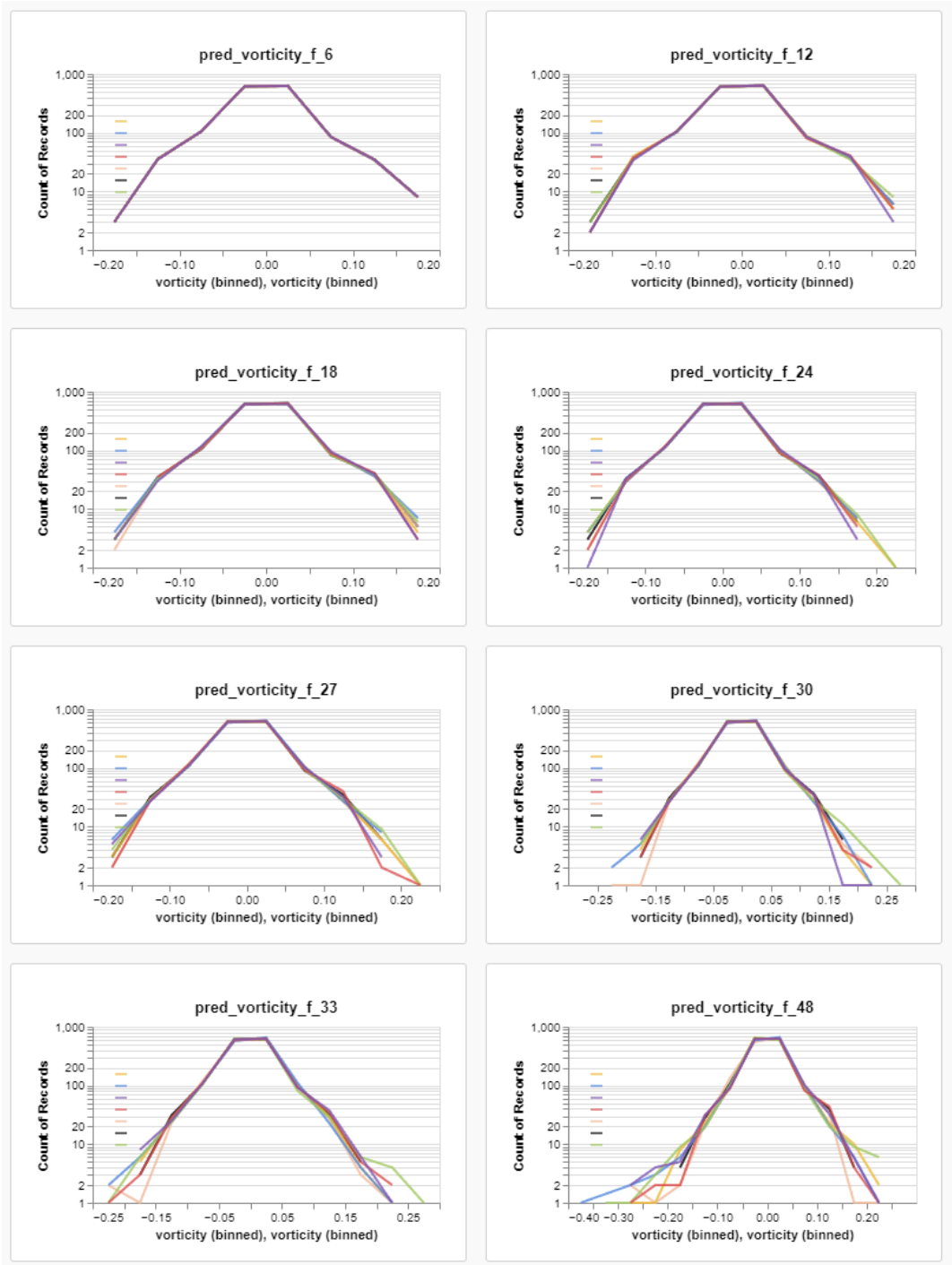


Figure 5.18: *Vorticity distribution (left to right, top to bottom) at frames [6, 9],[18, 21],[48, 51] . The black lines represent the ground truth vorticity magnitude binned as the reference,* `_GATv2Conv` *(blue) ,* `_graph_transformer` *(orange) ,* `_GNS` *(purple) ,* `_GATConv` *(yellow),* `_pointnet_conv` *(green) , and* `_OURS` *(red)*

Benchmark: Kinetic Energy Conservation

The provided plot shows the kinetic energy percentage over time for various models. The following observations can be made regarding the performance of these models in terms of kinetic energy conservation:

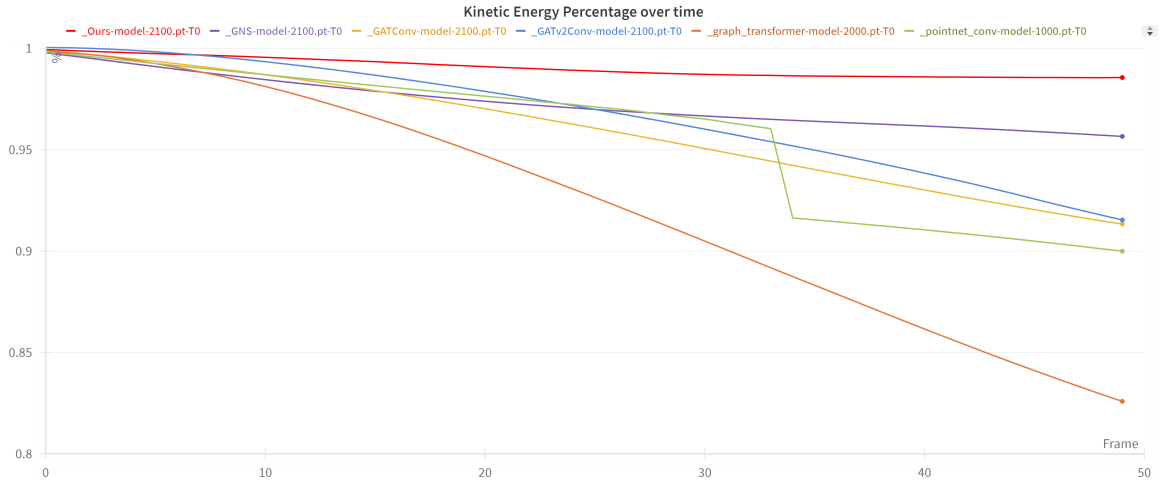


Figure 5.19: *Kinetic Energy Conservations Benchmark of the surrogate models*

The model **_Ours-model-2100.pt-T0** (represented by the red line) maintains the highest kinetic energy percentage over time, indicating the best performance in conserving kinetic energy. The kinetic energy decreases very slowly for this model, suggesting that it is highly effective in retaining the initial kinetic energy throughout the time frames.

The _GNS-model-2100.pt-T0 also maintains a high percentage of kinetic energy. The _GATv2Conv-model-2100.pt-T0 shows moderate performance, with a steady decline in energy conservation. However, the _GATConv-model-2100.pt-T0, _graph_transformer-model-2000.pt-T0, and _pointnet_conv-model-1000.pt-T0 exhibit significant drops in kinetic energy conservation, indicating the need for improvement in their energy conservation capabilities.

Conclusion The **OURS** proposed model aligns closely with the ground truth in peak alignment, distribution shape, tails, high Pearson correlation coefficient, low residual, low KL divergence, and Highest Kinetic Energy Conservation. It is the best-performing model in terms of matching the ground truth. Other models, such as **GATConv (Orange)** and **GATv2Conv (Blue)**, has slightly larger deviations from the ground truth. The **GNS (Pur-**

ple) and **graph_transformer (Green)** models have more noticeable differences and noise. The **pointnet (Yellow)** model shows the most significant deviation, with a broader and noisier distribution. In general, **OURS (red line)** is one of the best models for approximate ground truth, suggesting an effective performance to capture the vorticity distribution.

5.9 Curl-Noise Flow Test

Building upon the findings presented in the previous chapter, in which the performance of the surrogate model was evaluated using the lid-driven cavity flow test, this chapter extends the investigation to more complex flow scenarios. The focus is on assessing the surrogate model's ability to conserve vorticity in these more complex flow conditions, which pose significant challenges due to their potential to exhibit both ordered and chaotic behaviors.

To generate these complex flow conditions, the dataset utilized in this chapter is initialized using **Curl Noise** Robert Bridson & Nordenstam (2007), a method of generating smooth, divergence-free vector fields by taking the curl of the gradient of a **Perlin noise function** (Perlin (1985), Perlin (2002)) . This approach efficiently creates velocity fields with multiple vorticities that exactly respect solid boundaries and whose amplitude can be modulated in space as desired. This initialization method ensures that the initial vorticity field possesses the complexity necessary to examine the model's performance across a spectrum of flow behaviors.

This chapter provides a detailed exploration of the model's capability to accurately represent such dynamic and potentially unstable flow environments. The results of these simulations are intended to offer valuable insights into the robustness and applicability of the surrogate model.

5.9.1 Problem Setting

Curl noise is a technique for generating divergence-free vector fields, meaning that the vector field has no net flux in or out of any region. This is essential in fluid dynamics simulations where incompressibility (no divergence) is a key property. The curl noise method is particularly useful for simulating turbulent flows, which exhibit complex, swirling patterns.

Given a 2D vector field $\mathbf{v}(x, y) = (u(x, y), v(x, y))$, the curl of this field in 2D is defined as:

$$\text{curl}(\mathbf{v}) = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \tag{5.39}$$

To create a divergence-free field, curl noise uses the curl of the gradient of scalar noise fields (like Perlin noise). This ensures that the resulting vector field $\mathbf{v}_{\text{curl}}$ is divergence-free by construction.

Perlin Noise

Perlin noise, introduced by Perlin (1985), is a gradient noise function that produces smooth, continuous pseudo-random values. It is widely used in procedural texture generation, terrain generation.

Given a 2D point $\mathbf{p} = (x, y)$, Perlin noise is generated by computing a weighted sum of dot products between random gradient vectors at the corners of the grid cell containing $\mathbf{p}$ and the vectors from these corners to $\mathbf{p}$. The process can be summarized as follows:

1. **Grid Cell Identification**: Identify the grid cell in which $\mathbf{p}$ is located. The four corners of this cell are denoted as $\mathbf{p}_{00}, \mathbf{p}_{10}, \mathbf{p}_{01}, \mathbf{p}_{11}$.

2. **Gradient Vector Assignment**: Assign a random gradient vector $\mathbf{g}_{ij}$ at each corner $\mathbf{p}_{ij}$ of the grid cell.

3. **Dot Product Calculation**: Compute the dot product between each gradient vector $\mathbf{g}_{ij}$ and the vector from the corner $\mathbf{p}_{ij}$ to the point $\mathbf{p}$:

$$d_{ij} = \mathbf{g}_{ij} \cdot (\mathbf{p} - \mathbf{p}_{ij}) \tag{5.40}$$

4. **Interpolation**: Interpolate these dot products using a fade function $f(t)$, typically a smooth polynomial, to blend the contributions of the grid corners:

$$\text{Perlin}(\mathbf{p}) = \text{lerp}(f(x), \text{lerp}(f(y), d_{00}, d_{10}), \text{lerp}(f(y), d_{01}, d_{11})) \tag{5.41}$$

   where *lerp* denotes linear interpolation.

The smoothness and continuity of Perlin noise make it suitable for generating natural-looking textures and, more importantly, for simulating the underlying structure of turbulence in fluid simulations.

Curl Noise Generation

Once the Perlin noise is generated, its gradients are used to produce a curl noise vector field.

Given two scalar fields $f(x, y)$ and $g(x, y)$ generated by Perlin noise, the gradients of these fields are:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right), \quad \nabla g = \left(\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y} \right) \tag{5.42}$$

The resulting curl noise vector field $\mathbf{v}_{\text{curl}}$ is defined as:

$$\mathbf{v}_{\text{curl}}(x, y) = \left(\frac{\partial g}{\partial y} - \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} - \frac{\partial g}{\partial x} \right) \tag{5.43}$$

This field is divergence-free, making it suitable for driving incompressible flow simulations.

Simulation Setup

The simulation uses the MPM algorithm (see Section 2.3.2) to represent the fluid domain, with particles representing fluid elements. The velocities of these particles are initialized using a curl noise field by interpolating the velocity from the curl-noised grid to uniformly distributed particles across the fluid domain. The curl-noised grid is used only once at the beginning; the rest of the simulation is purely carried out using MPM steps.

Position Initialization: Each particle position is distributed almost uniformly by based on the cell size of the grid with dimension (40x40), with a small random displacement (jitter) added.

$$\mathbf{x}_i = \text{base\_pos} + \text{jitter} \tag{5.44}$$

- **Base Position:** The base position base_pos of a particle is determined by its grid coordinates, which are calculated as:

  $$\text{base\_pos} = \text{spacing} \times \mathbf{index} + \mathbf{offset}$$

  where ‘spacing‘ is the distance between grid points, and ‘offset‘ is an initial offset to ensure particles are well within the domain.

- **Jitter:** The jitter is a small random displacement added to each particle's position to avoid perfect alignment and to simulate natural irregularities. It is calculated as:

  $$\text{jitter} = (\mathbf{r} - 0.5) \times \text{spacing} \times 0.25$$

  where $\mathbf{r}$ is a random vector in the range [0, 1]. The factor of 0.25 reduces the jitter to a reasonable size, preventing particles from overlapping grid boundaries while still introducing variability.

Particle Velocity Initialization: The initial velocity $\mathbf{v}_i$ of each particle is set based on the curl noise field:

$$\mathbf{v}_i = \text{velocity\_scale} \times \mathbf{v}_{\text{curl}}(i, j) \tag{5.45}$$

Here, *velocity_scale* is a scaling factor that controls the magnitude of the initial velocities.

Substep Function The 'substep()' function performs the core simulation operations, which include updating the grid and particles based on the current state.

Grid Update

1. **Grid Reset:** All grid velocities $\mathbf{v}_g$ and masses m_g are initialized to zero at the start of each substep:

$$\mathbf{v}_g[i, j] = \mathbf{0}, \quad m_g[i, j] = 0 \tag{5.46}$$

2. **Particle to Grid Transfer:** Each particle's velocity and mass are transferred to the grid using a weighted kernel function w. This ensures that momentum is conserved:

$$\mathbf{v}_g[i, j] += w[i, j] \times (m_p \times \mathbf{v}_p + \text{affine momentum}) \tag{5.47}$$

$$m_g[i, j] += w[i, j] \times m_p \tag{5.48}$$

   where $\mathbf{v}_p$ is the particle velocity, m_p is the particle mass, and 'affine momentum' accounts for stress contributions.

3. **Grid Velocity Update:** After transferring particle data to the grid, the grid velocities are updated by applying external forces, such as gravity:

$$\mathbf{v}_g[i, j].y -= \Delta t \times \text{gravity} \tag{5.49}$$

   Boundary conditions are enforced to keep the particles within the simulation domain.

Particle Update

1. **Grid to Particle Transfer:** The particles' velocities are updated by interpolating from the grid velocities, ensuring a smooth transfer of motion from the grid back to the particles:

$$\mathbf{v}_p = \sum_{i,j} w[i, j] \times \mathbf{v}_g[i, j] \tag{5.50}$$

The particle positions are then updated based on the new velocities:

$$\mathbf{x}_p += \Delta t \times \mathbf{v}_p \tag{5.51}$$

2. **Deformation Gradient Update:** The deformation gradient J_p is updated to track changes in the local volume around each particle:

$$J_p \times = 1 + \Delta t \times \text{trace}(\mathbf{C}) \tag{5.52}$$

A volume correction term is applied to maintain stability and prevent excessive expansion or compression:

$$\mathbf{x}_p += \text{volume\_correction} \times (\mathbf{x}_p - \mathbf{X}_p) \times \Delta t \tag{5.53}$$

Divergence Check A key aspect of the simulation is ensuring that the velocity field remains as close to divergence-free as possible. The divergence of the ground truth is calculated as:

$$\text{div}(\mathbf{v}) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{5.54}$$

This check is intended to monitor whether the curl noise effectively enforces the incompressibility condition. Although the divergence may not be exactly zero, the purpose of this monitoring is to ensure that the divergence remains relatively low, indicating that the curl noise is successfully contributing to a near-incompressible flow field.

5.9.2 Dataset Generation

This subsection details the comprehensive methodology used to generate the dataset using curl noise simulation, including the role of scenario IDs, the influence of the noise scale on vorticity, and the systematic organization of the dataset into distinct subsets.

Simulation Setup and Execution

We simulates fluid behavior using 8096 particles projected across a 40 x 40 grid which velocities are initialized with curl noise. To automate the execution of this simulation across multiple scenarios, batch script is employed. This batch file systematically runs the simulation program multiple times, each time with different initial conditions or parameters as dictated by the scenario ID. The script is designed to manage and organize the outputs into three primary datasets:

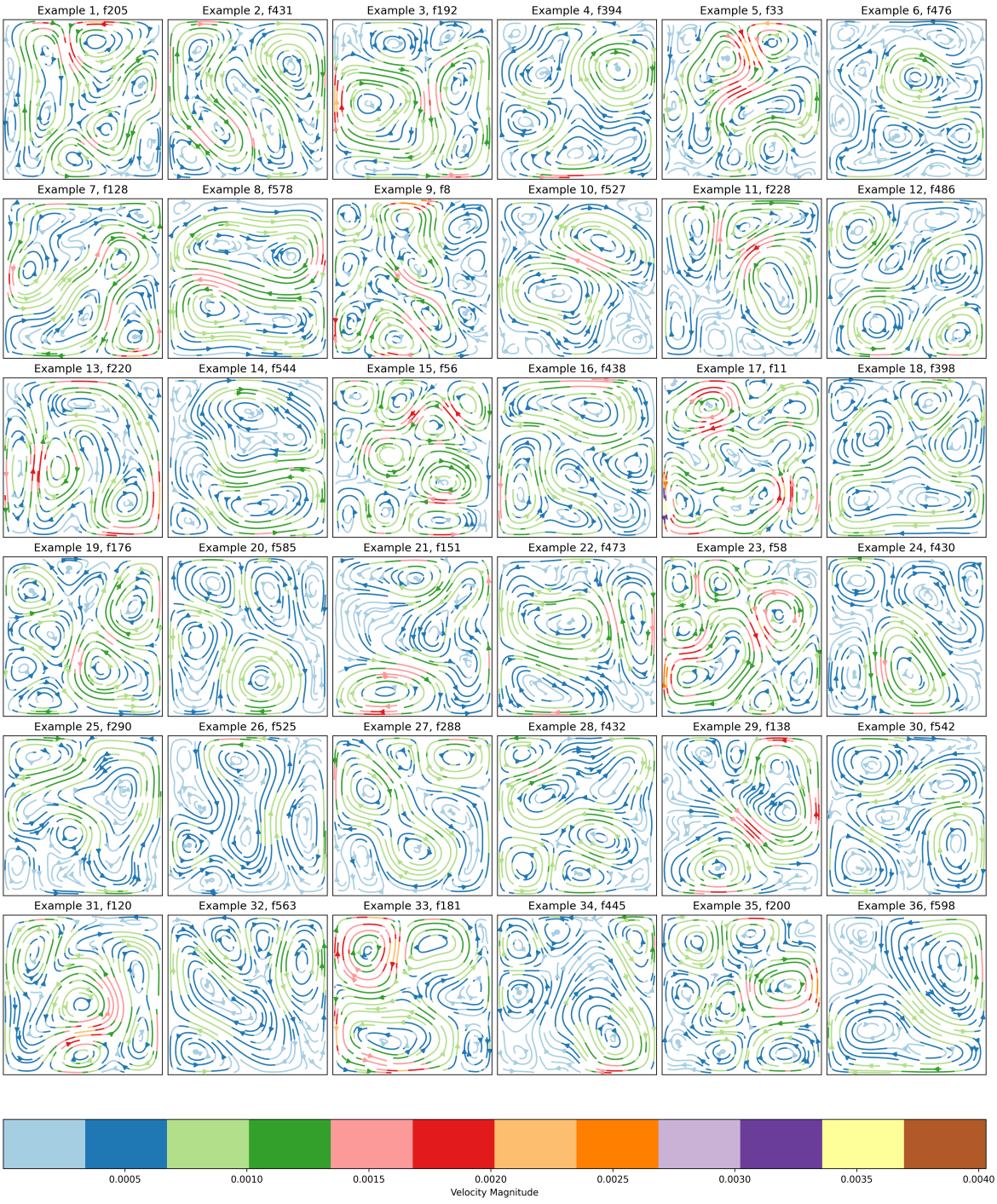


Figure 5.20: *llustrations of 36 out of (28.8k) conditions employed for the training dataset. The simulations depicted are weakly compressible and generated using the MPM model, starting with a velocity field that is initially divergence-free and constrained by solid boundaries outside the displayed areas. Velocity magnitude is shown through color mapping, while streamlines represent the flow field at time $t = f$, computed by interpolating particle velocities onto a regular grid using cubic interpolation.*

- **Training Set**: Comprising 48 scenarios, this set is used to train the neural network model.
- **Validation Set**: Comprising 6 scenarios, this set is utilized for hyperparameter tuning and to prevent overfitting.
- **Testing Set**: Comprising 6 scenarios, this set is reserved for evaluating the final performance of the trained model.

Each scenario produces an output in HDF5 (`.h5`) format, which is a common format for storing large numerical datasets, particularly those involving multi-dimensional arrays. The outputs are organized into directories corresponding to the training, validation, and testing datasets.

Scenario ID, Noise Scale, and Randomness

We generated 48 Scenario for Training data (values ranging from 0 to 49). The The variability across simulation scenarios is primarily driven by the `scenarioID` variable, which significantly influences key simulation parameters, most notably the `noiseScale`. The relationship between `scenarioID` and `noiseScale` is defined as follows:

$$noiseScale = scenarioID\%10 + 4 \tag{5.55}$$

This formula results in `noiseScale` values ranging from 4 to 13, depending on the scenario ID:

- `scenarioID % 10`: This operation yields a remainder between 0 and 9, effectively cycling through a set of values as the scenario ID increases.
- **+ 4**: This ensures that the `noiseScale` always starts from 4, producing possible values between 4 and 13.

The `noiseScale` parameter is directly linked to the complexity and characteristics of the fluid flow:

- **Higher `noiseScale` values** lead to more frequent variations in the curl noise field, generating smaller, more numerous vortices. This results in a more complex flow pattern, typical of high-energy fluid dynamics.

- **Lower `noiseScale` values** cause less frequent variations, producing larger but fewer vortices. The flow pattern becomes smoother and more orderly, simulating lower-energy fluid dynamics.

By varying `noiseScale` across different scenarios, the simulation generates a diverse range of fluid behaviors, from smooth and laminar flows to highly complex and chaotic ones. This diversity is crucial for training a neural network capable of generalizing across various fluid dynamics scenarios.

Incorporating Randomness for Unique Scenarios While the `noiseScale` repeats every 10 scenarios, each scenario remains unique due to the inherent randomness in the noise generation process. The Perlin and curl noise functions introduce stochastic elements that ensure no two scenarios are exactly alike.

Here's a brief overview of the noise generation process:

1. **Perlin Noise Generation**: The Perlin noise function generates a gradient noise that is smooth and continuous. It creates a grid of random gradient vectors, which are then used to compute noise values across the simulation space.

2. **Curl Noise Calculation**: Curl noise is derived from the gradient of the Perlin noise, resulting in a vector field with swirling patterns. This vector field is used to simulate fluid-like motions, with the curl noise dictating the velocity vectors of particles in the simulation.

3. **Scenario Initialization**:

   - **Position Initialization**: Each particle's position is initialized with a base grid location and a small random jitter to ensure variation within the grid cells.
   - **Velocity Initialization**: The velocity of each particle is sampled from the curl noise field, scaled by a factor to adjust the flow dynamics.

These random elements, particularly the Perlin noise gradients and the random jitter in particle positioning, ensure that even with the same `noiseScale`, each scenario exhibits unique fluid dynamics. The simulation's ability to generate a broad spectrum of fluid behaviors depends not only on the deterministic `noiseScale` but also on the underlying randomness

introduced in each scenario. This makes each scenario distinct, despite the cyclic nature of the `noiseScale`.

Data Conversion and Organization

After the simulation generates the raw data in `.h5` format, the next step involves converting this data into a more efficient format for neural network training: the `.npz` format. This conversion is processes the raw data as follows:

- **Data Extraction**: The script reads the `.h5` files from the `train`, `valid`, and `test` directories, extracting key variables such as particle positions, velocities, and potentially accelerations.
- **Statistical Analysis**: It computes the mean and standard deviation of the particle positions, velocities , and acceleration across all frames in each scenario. These statistics are essential for normalizing the input data during neural network training, ensuring that the data fed into the network is centered around zero and scaled to a consistent range.
- **NPZ File Creation**: The processed data, along with the computed statistics, is saved into `.npz` files (`train.npz`, `valid.npz`, `test.npz`). These files are organized into their respective folders, ensuring a clear separation of the datasets for training, validation, and testing purposes.

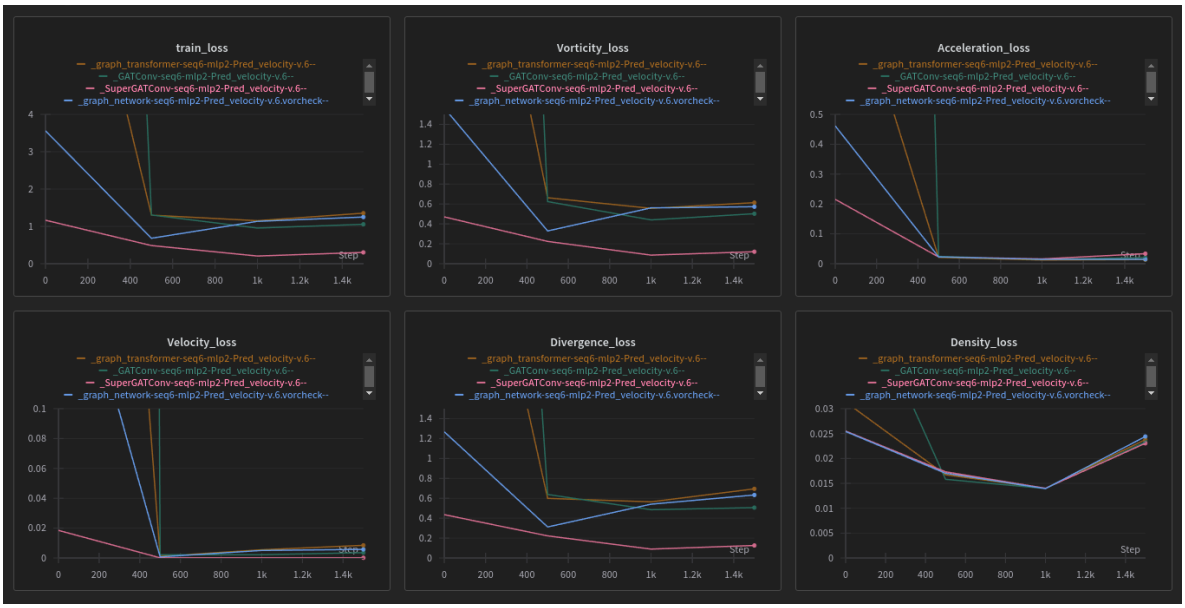


Figure 5.21: *Training Loss of different models using Curl noise Dataset with accceleration-based prediction.*

5.9.3 Training Process

In the training process of our network, we employ two primary prediction strategies: acceleration-based and velocity-based. The model is trained to predict the future states of particles by learning both acceleration and velocity cues, which are critical to accurately modeling physical systems. The training process involves iteratively adjusting the model's parameters to minimize a composite loss function that incorporates multiple aspects of the system's dynamics, including acceleration, velocity, density, divergence, position, and vorticity.

During training, the loss function is a weighted sum of these individual losses, with each component contributing equally (the Acceleration Loss, Velocity Loss, Density Loss, Position Loss, Divergence Loss, and Vorticity Loss), as indicated by the weight ratio of 1:1:1:1:1:1. This balanced approach ensures that the model learns to predict all aspects of the system's dynamics equally well, without overemphasizing one component at the expense of others.

As the training progresses, the loss values for each component fluctuate, reflecting the model's ongoing adjustments. For instance, at specific training steps, acceleration loss, velocity loss, density loss, position loss, divergence loss, and vorticity loss contribute to the overall Train Loss. A snapshot from our real training process illustrates these dynamics:

At training step 507 5.21, the total loss was 0.301686, with contributions from acceleration loss (0.035660), velocity loss (0.000256), density loss (0.017083963), position loss (almost zero), divergence loss (0.120924) and vorticity loss (0.127761), all equally weighted. As the training continued, these losses were periodically logged, showing variations as the model improved its predictions across different components. By the end of the training, the network achieved the following loss values: Acceleration loss (0.16061589), density loss (0.0268355), divergence loss (0.25397250), position loss (0.00000000249), train loss (0.64425) and velocity loss (0.000623). The vorticity loss at this point was 0.2022085. These values indicate that the model has effectively learned to balance the various physical properties, achieving a comprehensive understanding of the system it was trained to simulate.

The equal weight ratio applied throughout the training ensured that the network developed a robust capability to predict the system's dynamics holistically, without biasing the model towards any single component of the loss function. This balanced training strategy is crucial for accurately simulating complex physical phenomena where multiple interacting forces must be accounted for simultaneously.

5.9.4 Results

Results on Various Test Sets

In this section, we present the results of our proposed model using curl-noise dataset on multiple test sets, identified as Rollout 0 through Rollout 5. For each rollout, the ground truth and predicted particle movements are visualized using three primary methods: quiver plots, streamline plots, and vorticity plots. These visualizations provide a comprehensive view of the fluid dynamics at key frames within each simulation, showcasing velocity magnitude, flow patterns, and rotational characteristics.

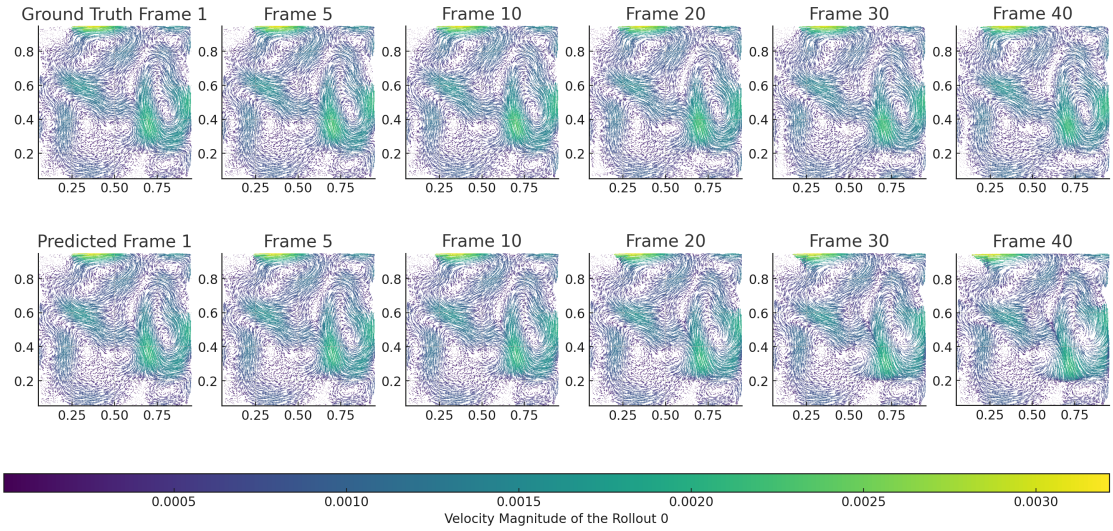


Figure 5.22: *Quiver plots of Rollout test 0. Our proposed surrogate model predicts the subsequent frames (1-40) given the initial frame (0).*

Quiver Plots The quiver plots, as shown in Figure 5.22, provide a direct representation of the velocity vectors at each timestep. The vectors' lengths and directions depict the magnitude and direction of the particle velocities. This method is particularly effective in highlighting the differences between the ground truth and predicted rollouts, as the color intensity correlates with the velocity magnitude. We have displayed the results at frames 1, 5, 10, 20, 30, and 40 for each rollout. In these plots, the predicted rollouts generally follow the ground truth closely, though some deviations can be observed in certain regions, particularly at later frames and near the boundaries of the simulation domain. These discrepancies are more apparent in

the quiver plots, where the vector directions and magnitudes can differ noticeably from the ground truth.

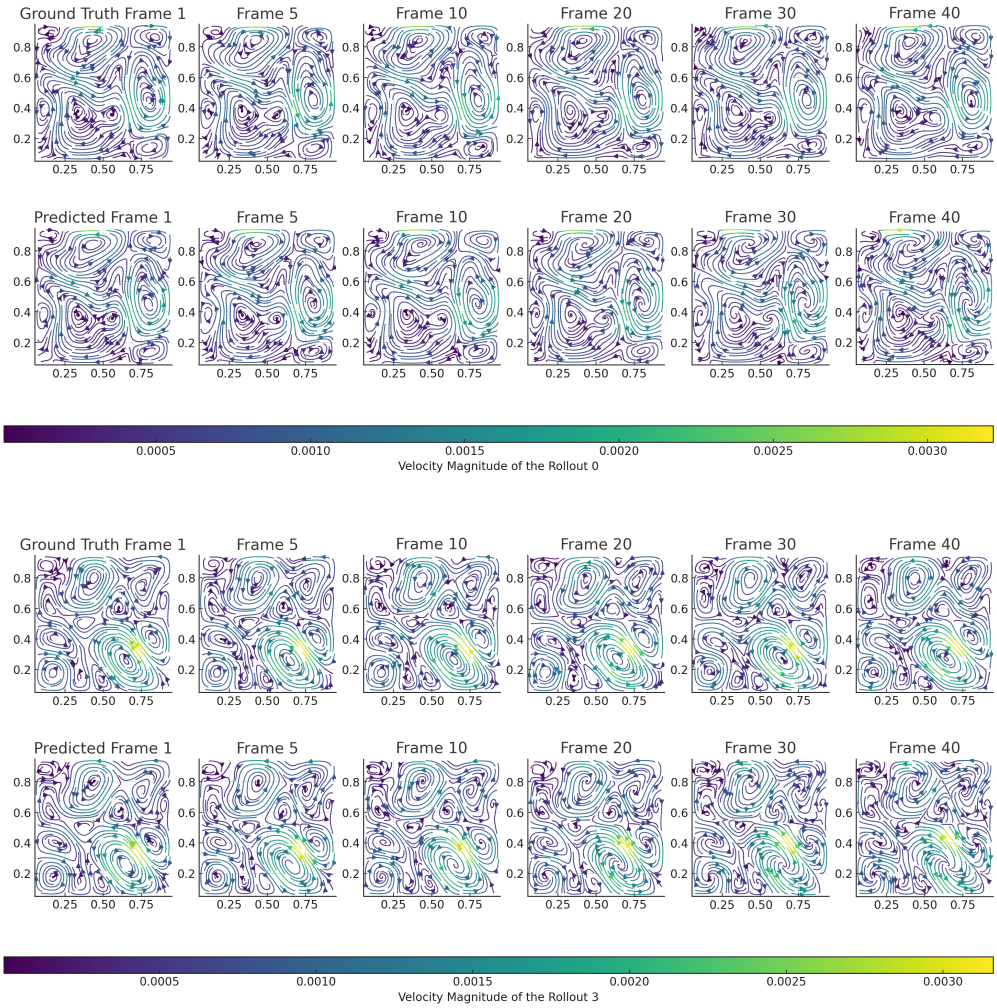


Figure 5.23: *Streamline view for Rollout Tests 0 and 3. First row: Ground Truth Test case 0. Third row: Ground Truth Test case 3. Second and Fourth row are the predictions of our porposed model.*

Streamline Plots The streamline plots, illustrated in Figure 5.23, offer a continuous visualization of the fluid flow, providing insight into the overall flow structure generated by the curl-noise simulation. The streamlines trace the paths that particles would follow under the simulated velocity field, with colors representing the velocity magnitude. These plots provide

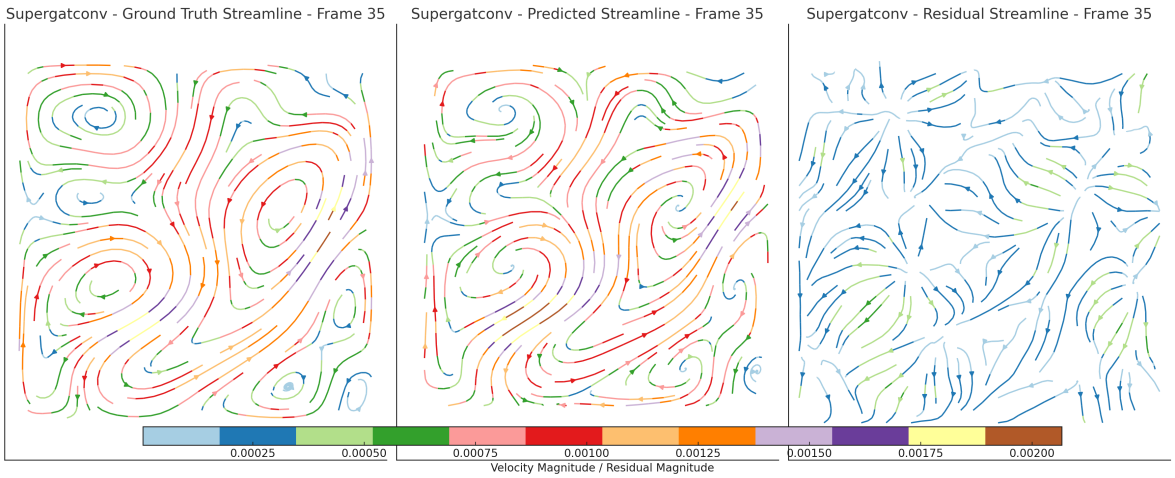


Figure 5.24: *Our-proposed model rollout prediction on frame 35 (with SuperGAT Convolution operator)*

a clearer understanding of the flow coherence and divergence within the simulated domain. Across the test sets, Rollouts 0 to 5, both quiver and streamline visualizations reveal the fidelity of the surrogate model in approximating the ground truth fluid dynamics. Most of the predicted paths align with the ground truth flow structures, indicating that the model captures the global flow patterns effectively.

Vorticity Plots The vorticity plots, shown in Figure 5.25, provide an analysis of the rotational characteristics of the fluid flow, representing the local spinning motion of the fluid. Vorticity is particularly important in fluid dynamics as it highlights areas where the flow exhibits circular or rotational behavior. The plots compare the vorticity fields of the ground truth (target) and predicted simulations across five key frames: 1, 5, 10, 20, and 30. The color map used is offering a clear visualization of the vorticity magnitude.

In the early frames (1 and 5), the predicted vorticity fields closely match the target, indicating that the model is well-calibrated to the initial conditions and can accurately simulate the early development of fluid flow, including the formation of initial vortices. However, as the simulation progresses to frames 10, 20, and 30, discrepancies between the target and predicted vorticity fields become more pronounced. While the overall patterns of vorticity are preserved, the exact magnitudes and positions of vortices begin to differ, especially in regions

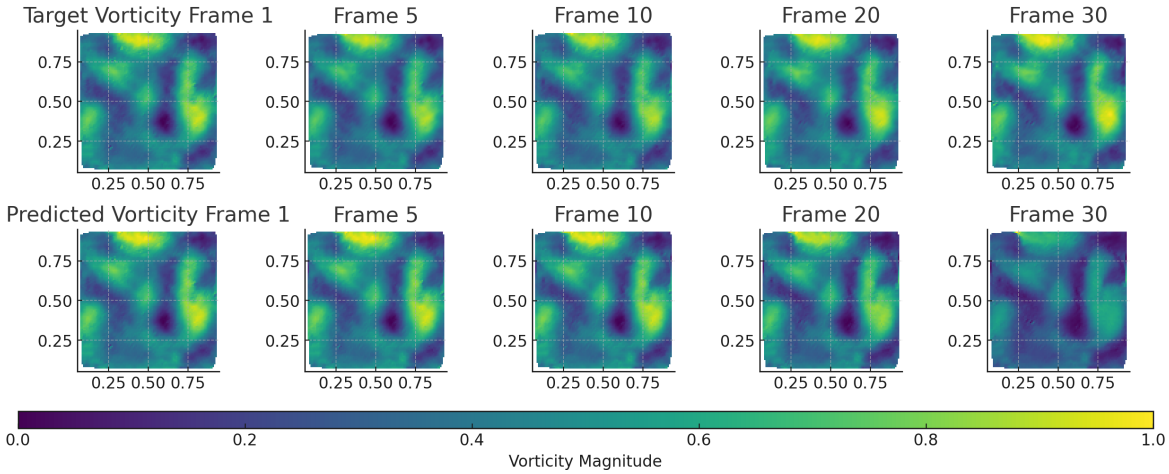


Figure 5.25: *Vorticity plot for Rollout 0. The top row shows the target vorticity, while the bottom row shows the predicted vorticity across frames 1, 5, 10, 20, and 30.*

with complex interactions or high magnitude of velocity field.

By frame 30, the differences more noticeable, with some vortices in the predicted fields either displaced or differing in intensity compared to the ground truth. This suggests that the model, while effective in capturing general dynamics, may struggle with fine-scale details as the simulation evolves, particularly in complex regions where small initial discrepancies can amplify over time. The use of a single color bar across all frames enhances the comparability between the target and predicted vorticity fields, making it easier to identify regions where the model's predictions diverge from the ground truth.

Overall Assessment The combined analysis from quiver, streamline, and vorticity plots provides a comprehensive understanding of the surrogate model's performance in simulating fluid dynamics. The quiver and streamline plots demonstrate that the model captures the global flow patterns effectively, maintaining coherence with the ground truth. The vorticity plots, however, reveal areas where the model's predictions deviate from the ground truth, particularly in terms of rotational dynamics. These findings suggest that while the model is generally effective, there is room for improvement, particularly in accurately capturing the fine-scale rotational details of the fluid flow over time.

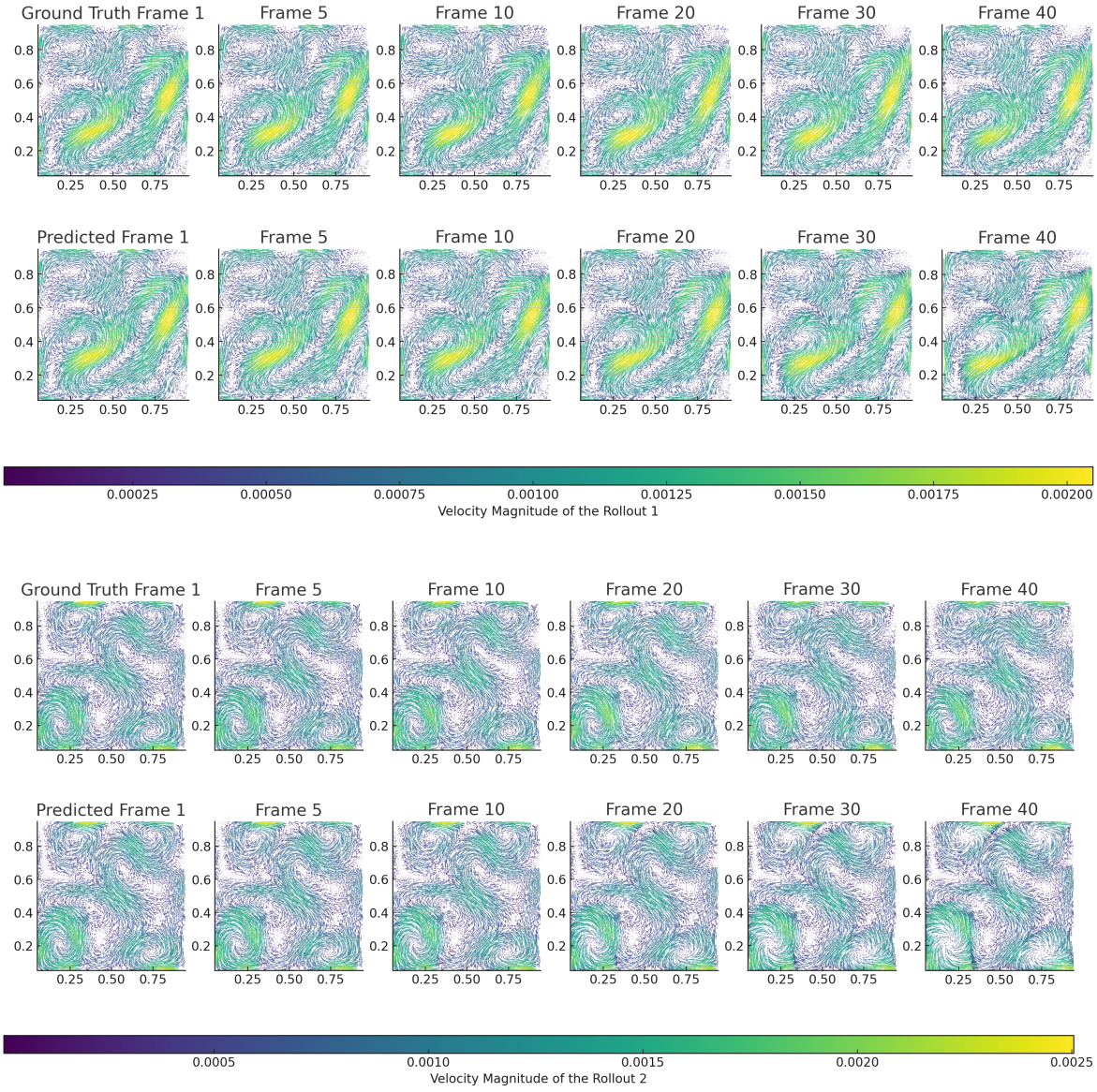


Figure 5.26: *Results for Rollout Tests 1 and 2 for curl noise simulation.*

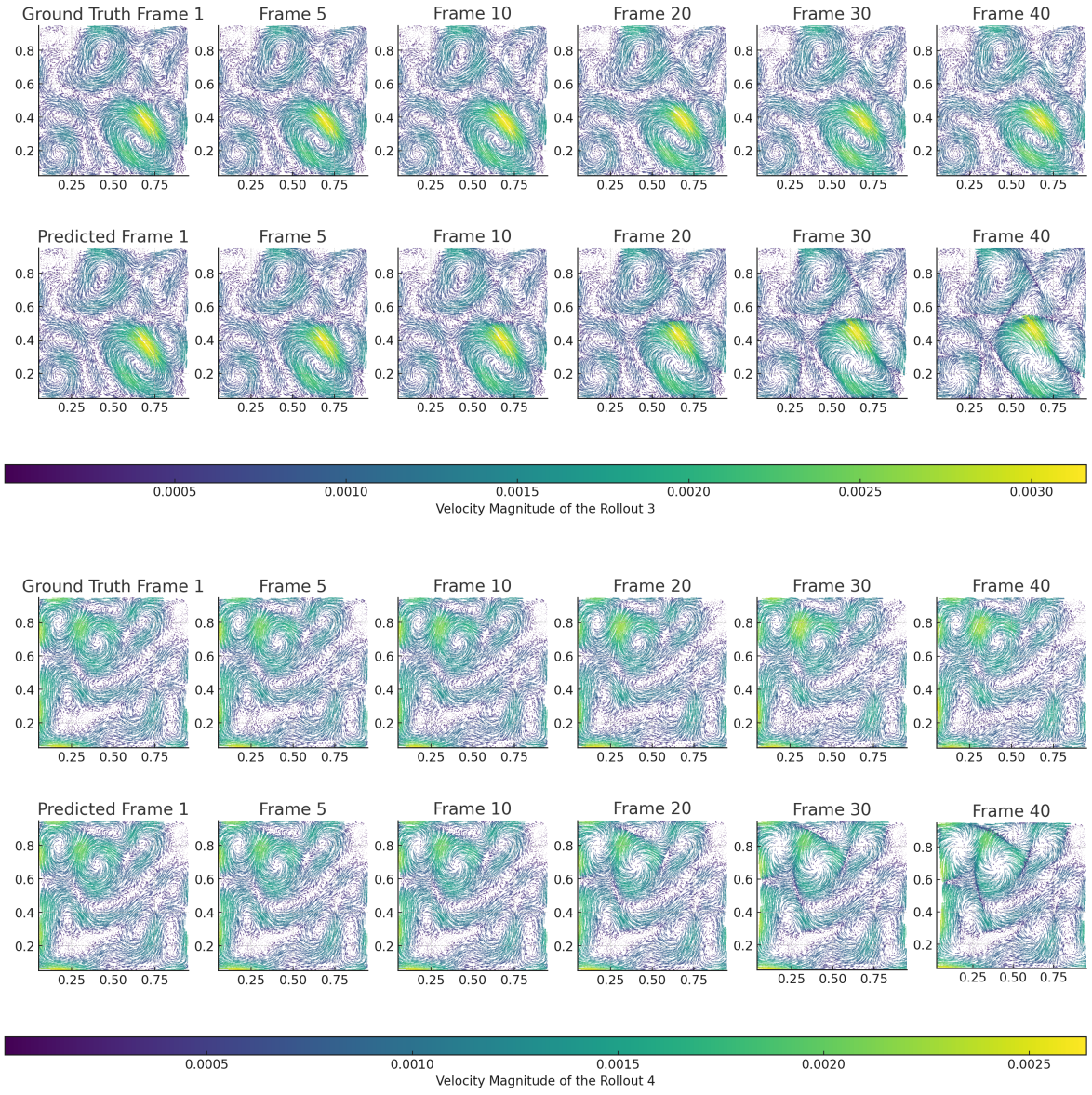


Figure 5.27: *Results for Rollout Test case 3 and 4 for curl noise simulation.*

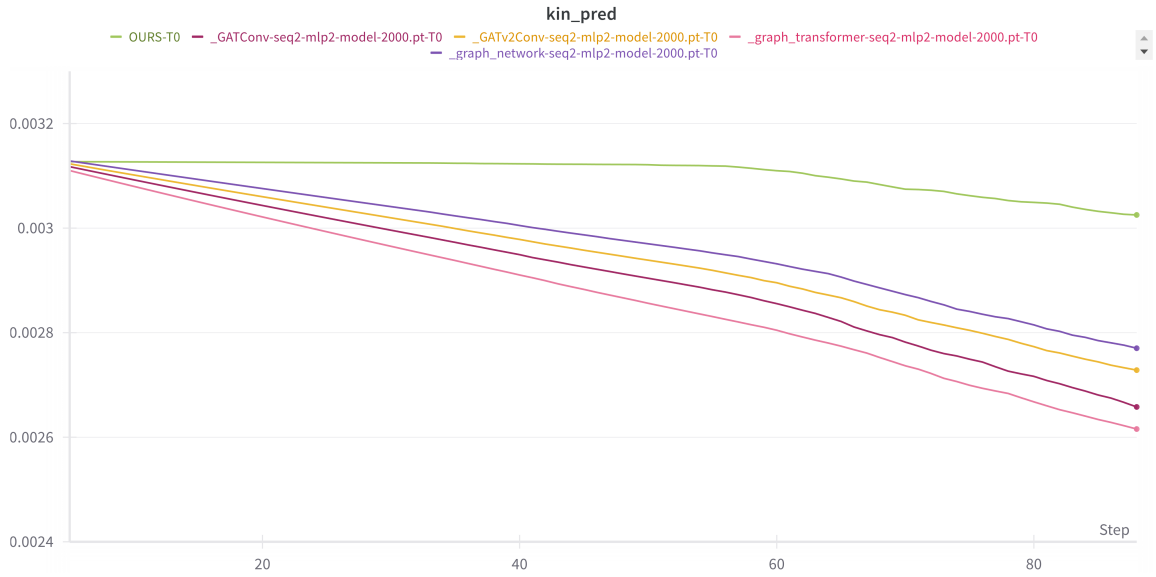


Figure 5.28: *Comparison of Conservation of Kinetic Energy of our proposed model with relative to others surrogate models.*

Benchmark: Energy Conservation

The plot on figure 5.28 compares the performance of different models in predicting the conservation of kinetic energy during a curl noise simulation using Test case T0. The y-axis represents the predicted kinetic energy, while the x-axis shows the progression of the simulation over time steps.

The green line represents our model $OURS - T0$, which shows the highest kinetic energy values throughout the simulation. This indicates that it conserves kinetic energy better than the other models. The curve is relatively flat, meaning that the kinetic energy does not decrease as much over time, suggesting strong conservation properties.

The purple line, representing the $GATConv$, shows a moderate decrease in kinetic energy over time. This model performs worse than $OURS$, but better than the GATv2Conv model. Its ability to conserve kinetic energy is less robust compared to our model, with a slightly steeper decline.

The yellow line corresponds to the $GATv2Conv$ model, which demonstrates a slightly faster decline in kinetic energy. This indicates that it does not conserve kinetic energy as effectively as the $GATConv$ model. The steeper decline suggests that this model may lose

energy more quickly, potentially leading to less accurate predictions in the context of fluid dynamics.

The pink line, representing the $graphtransformer$, exhibits the fastest decline in kinetic energy among the models shown. This suggests that the graph transformer is the least effective in conserving kinetic energy, which may result in a less accurate simulation of fluid dynamics, particularly in scenarios where maintaining energy levels is critical.

Finally, the magenta line represents the $graphnetwork$. The performance of this model falls between the graph transformer and the GATConv models. It has a moderate rate of energy loss, but it is still worse than the GATConv models and our proposed model.

OUR proposed mode shows the best performance in conserving kinetic energy during the curl noise simulation, maintaining a relatively stable kinetic energy level even over relatively longer time steps.

5.10 Conclusion

In this chapter, we explored the development and evaluation of a surrogate model designed to conserve vorticity in Lagrangian fluid simulations. The key contributions included an in-depth analysis of vorticity conservation methods, particularly focusing on the conservation of angular momentum and energy in continuum simulations. The research highlighted the importance of accurately modeling vorticity and introduced a novel network architecture incorporating a Self-Supervised Graph Attention Operator. This architecture significantly enhanced the model's ability to capture the complex interactions between particles, ensuring better preservation of vorticity.

Through rigorous evaluation, including tests on various datasets and comparisons with other models, the proposed model proved to be robust, achieving lower mean squared errors and better alignment with ground truth data. The model's superior performance in conserving kinetic energy and predicting long-range behaviors further underscores its potential for practical applications in fluid dynamics simulations.

While the chapter affirms the effectiveness of the proposed model in maintaining vorticity conservation, marking a substantial advancement in the field of Lagrangian fluid simulation, we acknowledge that vorticity conservation is still not perfect and that further improvements are necessary. The insights gained from this research pave the way for future explorations into more complex fluid dynamics scenarios and the continued refinement of surrogate modeling techniques.

Chapter 6

Conclusions and Further Investigations

6.1 Conclusion

Temporal Modeling One of the significant contributions of this thesis is the development and application of advanced temporal learning techniques within the context of Lagrangian continuum simulations. The methodologies explored in Chapter 3 provided a robust framework for transforming Euclidean data into latent space representations, enabling neural networks to effectively model the intricate dynamics of fluid and deformable solid simulations. This transformation process, coupled with a focus on optimizing efficiency and scalability, has been essential in advancing the state-of-the-art in real-time and large-scale simulations.

Building on this foundation, Chapter 4 introduced an innovative approach to temporal modeling by integrating Long Short-Term Memory (LSTM) Networks with Multi-Layer Perceptrons (MLP). This hybrid architecture was specifically designed to address the challenges of capturing complex interactions and long-term dependencies in multimaterial simulations. Through rigorous comparisons with traditional time series prediction methods, the LSTM-MLP model demonstrated superior performance, particularly in scenarios involving fluids and deformable solids.

The advancements presented in these chapters have not only enhanced the predictive accuracy of neural network-based surrogate models but have also provided valuable insights into the design and evaluation of temporal models for continuum simulations. The successful

implementation of these techniques marks a significant step forward in the pursuit of more efficient, scalable, and accurate simulation methods, laying the groundwork for future research and applications in this domain.

Conservation of Vorticity in Surrogate Models Chapter 5 addresses a key question of this thesis: How can neural network-based surrogate models be enhanced to accurately conserve vorticity in fluid simulations? The challenge of conserving physical properties such as vorticity in Lagrangian simulations is critical for ensuring the accuracy and stability of the models over long-term predictions.

The primary contribution of Chapter 5 is the introduction of the Self-Supervised Graph Attention Operator (SuperGAT), a novel approach developed by Kim and Oh (2022) Kim & Oh (2022), designed to preserve key physical properties, including vorticity and energy, across simulated particles. This operator significantly improves the neural network's ability to make stable and accurate predictions in complex fluid and solid dynamics simulations. By incorporating this Self-Supervised Graph Attention Operator within a Graph Neural Network (GNS) framework, this approach addresses a key limitation found in existing models, which often struggle to balance computational efficiency with the preservation of physical accuracy. The novelty of SuperGAT lies in its self-supervised learning mechanism, which dynamically adjusts the importance of neighboring nodes (particles) to focus on critical local interactions, enhancing the model's ability to conserve rotational dynamics and other key fluid properties.

Additionally, Chapter 5 enhances the training framework of the Graph-Based Simulator by integrating both acceleration-based and velocity-based predictions. This dual approach increases the model's versatility, allowing it to capture a broader range of dynamic behaviors within the simulated environment.

The chapter also introduces an extended loss function framework specifically designed for Lagrangian-based simulations. This framework includes terms for acceleration, velocity, position, density, divergence, and vorticity, ensuring a comprehensive understanding of the dynamics being modeled. The enhanced evaluation framework proposed in this chapter rigorously assesses the surrogate model's performance in conserving vorticity through various methods, including Pearson correlation, residual analysis, and Kullback-Leibler (KL) divergence. These methods provide a multi-faceted evaluation, highlighting the model's strengths in maintaining physical accuracy over time.

The research conducted in Chapter 5 answers the posed research question by demonstrating that the integration of the Self-Supervised Graph Attention Operator and the extended loss function framework can indeed lead to more accurate and stable predictions in fluid and solid dynamics simulations. These advancements pave the way for future research to further refine these models, ensuring their applicability in increasingly complex and real-time simulation environments.

6.2 Further Investigations

Temporal Modeling The LSTM-MLP architecture developed in this thesis was implemented as a Fully Connected Network. Despite its simplicity and effectiveness in simulating multimaterial interactions, there is a need to extend its capabilities to handle a dynamic number of particles. This can be achieved by incorporating convolutional operators into the surrogate model, which would enhance its flexibility and applicability to more complex scenarios.

Conservation of Vorticity in Surrogate Models One challenge identified in this research is the accumulation of errors during long-term rollout predictions, which can deviate the accuracy of the predictions over time. To address this, future work could focus on extending the loss function to incorporate averaged accumulated loss, which can be applied to various physical loss terms. This extension would help maintain the accuracy of predictions, particularly in preserving vorticity and divergence in fluid simulations.

Furthermore, there is a need to extend the problem setting from 2D to 3D domains to further evaluate the capability of the surrogate models in predicting dynamics in 3D space. This expansion would not only test the models' robustness but also broaden their applicability to real-world simulations involving complex phenomena such as vortex rings, vortex shedding, and other three-dimensional fluid dynamics. Addressing these challenges in 3D environments is crucial for ensuring the models' effectiveness in more realistic and complex scenarios.

Bibliography

Ahamadi, M. & Harlen, O. (2008*a*), 'A lagrangian finite element method for simulation of a suspension under planar extensional flow', *Journal of Computational Physics* **227**, 7543–7560.

Ahamadi, M. & Harlen, O. (2008*b*), 'A lagrangian finite element method for simulation of a suspension under planar extensional flow', *Journal of Computational Physics* **227**, 7543–7560.

Auer, S. (2009), Realtime particle-based fluid simulation, Technical report, Technische Universität München.

Batchelor, G. K. (1967), *An Introduction to Fluid Dynamics*, Cambridge University Press. **URL:** *https://doi.org/10.1017/CBO9780511800955*

Bender, J., Weber, D. & Diziol, R. (2013), 'Fast and stable cloth simulation based on multi-resolution shape matching', *Computers and Graphics* **37**(8), 9445–954.

Benra, F.-K., Dohmen, H. J., Pei, J., Schuster, S. & Wan, B. (2011), 'A comparison of one-way and two-way coupling methods for numerical analysis of fluid-structure interactions', *Journal of Applied Mathematics* **2011**.

Bird, B. & Pinkava, J. (2007), 'Ratatouille', Film. Pixar Animation Studios.

Black, N., Moore, S. & Weisstein, E. W. (2024), 'Jacobi method'. From MathWorld–A Wolfram Web Resource.

Box, G. E. P., Jenkins, G. M., Reinsel, G. C. & Ljung, G. M. (2016), *Time Series Analysis: Forecasting and Control*, 5th edn, John Wiley & Sons, Hoboken, NJ.

Brackbill, J. & Ruppel, H. (1986), 'Flip: a method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions', *Journal of Computational Physics* **65**, 314–343.

Brady, I. (2018), 'Stds assignment 1 - vignette - residual analysis'.
URL: *https://rpubs.com/iabrady/residual-analysis*

Bridson, R. (2015*a*), *Fluid Simulation for Computer Graphics*, CRC Press, Boca Raton.

Bridson, R. (2015*b*), *Fluid Simulation for Computer Graphics*, CRC Press, Boca Raton.

Brockwell, P. J. & Davis, R. A. (2009), Autoregressive moving average (arma) models, *in* R. A. Meyers, ed., 'Encyclopedia of Complexity and Systems Science', Springer, New York, NY, p. 447–457.
URL: *https://link.springer.com/referenceworkentry/10.1007/978-3-642-04898-2*$_3$80

Brody, S., Alon, U. & Yahav, E. (2022), 'How attentive are graph attention networks?'.
URL: *https://arxiv.org/abs/2105.14491*

Brunton, S. L., Noack, B. R. & Koumoutsakos, P. (2020), 'Machine learning for fluid mechanics', *Annual Review of Fluid Mechanics* **52**, 477–508.

Buhl, N. (2023), 'Kl divergence in machine learning', *Encord* .
URL: *https://encord.com/blog/kl-divergence-in-machine-learning*

Cho, K., van Merrienboer, B., Bahdanau, D. & Bengio, Y. (2014), 'On the properties of neural machine translation: Encoder-decoder approaches', *arXiv preprint arXiv:1409.1259* .
URL: *https://arxiv.org/abs/1409.1259*

Chollet, F. (2018), *Deep Learning with Python*, 1st edn, Manning Publications Co., Shelter Island, NY.

Chu, M., Liu, L., Zheng, Q., Franz, E., Seidel, H.-P., Theobalt, C. & Zayer, R. (2022), 'Physics informed neural fields for smoke reconstruction with sparse data', *ACM Transactions on Graphics* **41**(4), 1–14.
URL: *http://dx.doi.org/10.1145/3528223.3530169*

Chu, M. & Thuerey, N. (2017), 'Data-driven synthesis of smoke flows with cnn-based feature descriptors', *ACM Transactions on Graphics (SIGGRAPH)* **36**(4).

Cline, D., Cardon, D. & Egbert, P. K. (2005), 'Tutorial of the marker and cell method in computer graphics'.

Courant, R., Friedrichs, K. & Lewy, H. (1928), 'Über die partiellen differenzengleichungen der mathematischen physik', *Mathematische Annalen* **100**, 32–74. Translated as: "On the Partial Difference Equations of Mathematical Physics".

Deng, Y., Yu, H.-X., Wu, J. & Zhu, B. (2023), 'Learning vortex dynamics for fluid inference and prediction'.
URL: *https://arxiv.org/abs/2301.11494*

Dharma, D., Jonathan, C., Kistidjantoro, A. I. & Manaf, A. (2017), Material point method based fluid simulation on gpu using compute shader, *in* '2017 International Conference on Advanced Informatics, Concepts, Theory, and Applications (ICAICTA)'.

Dharma, D., Kusnadi, L. F. & Manaf, A. (2018), Fluid simulation based on material point method on gpu using cuda, *in* 'International Conference on Electrical Engineering and Computer Science (ICEECS) 2018'.

Dowdy, S., Weardon, S. & Chilko, D. (2004), *Statistics for Research*, John Wiley and Sons.
URL: *https://sdp2013.wordpress.com/wp-content/uploads/2013/09/statistics-for-research-dowdy.pdf*

Fei, Y., Batty, C., Grinspun, E. & Zheng, C. (2018), 'A multi-scale model for simulating liquid-fabric interactions', *ACM Transactions on Graphics* **37**(4), 51.

Fei, Y. R., Batty, C., Grinspun, E. & Zheng, C. (2019), 'A multi-scale model for coupling strands with shear-dependent liquid', *ACM Transactions on Graphics* **38**(6).

Feng, C. (n.d.), 'Recurrent neural networks', `https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks`. Accessed: 2024-08-09.

Gao, M., Tampubolon, A., Jiang, C. & Sifakis, E. (2017), 'An adaptive generalized interpolation material point method for simulating elastoplastic materials', *ACM Transactions on Graphics* **36**(6), 223.

Gao, M., Wang, X., Wui, K., Pradhana, A., Sifakis, E., Yuksel, C. & Jiang, C. (2018), 'Gpu optimization of material point methods', *ACM Transactions on Graphics* **37**(6).

Gingold, R. & Monaghan, J. (1977), 'Smoothed particle hydrodynamics: theory and application to non-spherical stars', *Monthly Notices of the Royal Astronomical Society* **181**, 375–389.

Goldade, R., Wang, Y., Aanjaneya, M. & Batty, C. (2019), 'An adaptive variational finite difference framework for efficient symmetric octree viscosity', *ACM Transactions on Graphics (TOG)* **38**(4), 94.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press.

Gropp, W. D., Kaushik, D. K., Keyes, D. E. & Smith, B. F. (2001), 'High-performance parallel implicit cfd', *Parallel Computing* **27**(4), 337–362.

Guan, S., Deng, H., Wang, Y. & Yang, X. (2022), 'Neurofluid: Fluid dynamics grounding with particle-driven neural radiance fields'.
URL: *https://arxiv.org/abs/2203.01762*

Guo, Q., Han, X., Fu, C., Gast, T., Tamstorf, R. & Teran, J. (2018), 'A material point method for thin shells with frictional contact', *ACM Transactions on Graphics* **37**(4), 147.

Harlow, F. (1963), 'The particle-in-cell method for numerical solution of problems in fluid dynamics', *Experimental Arithmetic, High-Speed Computations and Mathematics* .

Helfenstein, R. & Koko, J. (2012), 'Parallel preconditioned conjugate gradient algorithm on gpu', *Journal of Computational and Applied Mathematics* **236**(15), 3584–3590.

Hoaglin, D. C. (2003), 'John w. tukey and data analysis'.
URL: *https://www.jstor.org/stable/3182748*

Hochreiter, S. & Schmidhuber, J. (1997*a*), 'Long short-term memory', *Neural Computation* **9**(8).

Hochreiter, S. & Schmidhuber, J. (1997*b*), 'Long short-term memory', *Neural Computation* **9**(8), 1735–1780.
URL: *https://www.bioinf.jku.at/publications/older/2604.pdf*

Holden, D., Duong, B. C., Datta, S. & Nowrouzezahrai, D. (2019), Subspace neural physics: Fast data-driven interactive simulation, *in* 'SCA '19: The 18th annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation'.

Hopfield, J. J. (1982), 'Neural networks and physical systems with emergent collective computational abilities', *Proceedings of the National Academy of Sciences* **79**(8), 2554–2558.

Hu, Y., Fang, Y., Ge, Z., Qu, Z., Zhu, Y., Pradhana, A. & Jiang, C. (2018), 'A moving least squares material point method with displacement discontinuity and two-way rigid body coupling', *ACM Transactions on Graphics* **37**(4), 150:1–150:14.

Hu, Y., Li, T., Anderson, L., Ragan-Kelley, J. & Durand, F. (2019), 'Taichi: a language for high-performance computation on spatially sparse data structures', *ACM Transactions on Graphics* **38**(6), 1–6.

Hu, Y., Liu, J., Tenenbaum, J. B., Freeman, W. T., Wu, J., Rus, D. & Matusik, W. (2018), 'Chainqueen: A real-time differentiable physical simulator for soft robotics', *32nd Conference on Neural Information Processing Systems (NIPS 2018)* .

Huang, L., Hadrich, T., L, D. & Michels (2019), 'On the accurate large-scale simulation of ferrofluids', *ACM Transactions on Graphics* **38**(4), 93.

Ihmsen, M., Cornelis, J., Solenthaler, B., Horvath, C. & Teschner, M. (2014), 'Implicit incompressible sph', *IEEE Transactions on Visualization and Computer Graphics* **20**(3), 426–435.

Jiang, C., Gast, T. & Teran, J. (2017), 'Anisotropic elastoplasticity for cloth, knit and hair frictional contact', *ACM Transactions on Graphics (TOG)* **36**(4), 152.

Jiang, C., Schroeder, C., Selle, A., Teran, J. & Stomakhin, A. (2015), 'The affine particle-in-cell method', *ACM Transactions on Graphics (TOG)* **34**(4), 51.

Jiang, C., Schroeder, C., Teran, J., Stomakhin, A. & Selle, A. (2016), 'The material point method for simulating continuum materials', *ACM SIGGRAPH 2016 Courses* p. 24.

Kendall, M. G. & Stuart, A. (1964), 'The advanced theory of statistics, volume 2, "inference and relationship"', *The Annals of Mathematical Statistics* **35**(3), 1371–1380.
URL: *https://www.jstor.org/stable/2238272*

Kendall, W. S., Marin, J. M. & Robert, C. P. (2007), 'Confidence bands for brownian motion and applications to monte carlo simulation', *Statistics and Computing* **17**, 1–10.
URL: *https://doi.org/10.1007/s11222-006-9001-z*

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M. & Tang, P. T. P. (2017), On large-batch training for deep learning: Generalization gap and sharp minima, *in* 'Proceedings of the International Conference on Learning Representations (ICLR)'.
URL: *https://arxiv.org/abs/1609.04836*

Kim, B., Azevedo, V. C., Thuerey, N., Kim, T., Gross, M. & Solenthaler, B. (2019), Deep fluids: A generative network for parameterized fluid simulations, *in* 'Computer Graphics Forum (Proceedings of Eurographics 2019)'.

Kim, D. & Oh, A. (2022), 'How to find your friendly neighborhood: Graph attention design with self-supervision'.
URL: *https://arxiv.org/abs/2204.04879*

Kingma, D. P. & Ba, J. (2014), 'Adam: A method for stochastic optimization', *arXiv preprint arXiv:1412.6980* .
URL: *https://arxiv.org/abs/1412.6980*

Kingma, D. P. & Ba, J. (2015), 'Adam: A method for stochastic optimization'.

Klár, G., Gast, T., Pradhana, A., Fu, C., Schroeder, C., Jiang, C. & Teran, J. (2016), 'Drucker-prager elastoplasticity for sand animation', *ACM Transactions on Graphics* **35**(4), 103:1–103:12.

Ladický, L., Jeong, S., Solenthaler, B., Pollefeys, M. & Gross, M. (2015), 'Data-driven fluid simulations using regression forests', *ACM Transactions on Graphics (TOG)* **34**(6), 199.

Laehner, Z., Cremers, D. & Tung, T. (2018), Deepwrinkles: Accurate and realistic clothing modeling, *in* '15th European Conference on Computer Vision (ECCV)'.

Lee, T. M., Oh, Y. J. & Lee, I.-K. (2019), 'Efficient cloth simulation using miniature cloth and upscaling deep', *ACM Transactions on Graphics* **38**(1).

Macklin, M. & Müller, M. (2013), 'Position based fluids', *ACM Transactions on Graphics* **32**(4).

Monaghan, J. (1992*a*), 'Smoothed particle hydrodynamics', *Annual Review of Astronomy and Astrophysics* **30**, 543–574.
URL: *https://www.sciencedirect.com/science/article/abs/pii/0021999192900026*

Monaghan, J. J. (1992*b*), 'Smoothed particle hydrodynamics', *Annual Review of Astronomy and Astrophysics* **30**, 543.

Müller, M., Charypar, D. & Gross, M. (2003), 'Particle-based fluid simulation for interactive applications', *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* pp. 154–159.

Nie, X., Chen, L. & Xiang, T. (2015), 'Real-time incompressible fluid simulation on the gpu', *International Journal of Computer Games Technology* .

Nwankpa, C. E., Ijomah, W., Gachagan, A. & Marshall, S. (2018), Activation functions: Comparison of trends in practice and research for deep learning, *in* '2018 International Conference on Learning Representations'.

Oh, S., Noh, J. & Wohn, K. (2008), 'A physically faithful multi grid method for fast cloth simulation', *Computer Animation and Virtual Worlds* **19**, 479–492.

Olah, C. (2015), 'Understanding lstms'.
URL: *https://colah.github.io/posts/2015-08-Understanding-LSTMs/*

Overby, M., Brown, G. E., Li, J. & Narain, R. (2017), 'Admm - projective dynamics: Fast simulation of hyperelastic models with dynamic constraints', *IEEE Transactions on Visualization and Computer Graphics* **23**(10), 2222–2234.

Perlin, K. (1985), An image synthesizer, *in* 'Proceedings of the 12th annual conference on Computer graphics and interactive techniques (SIGGRAPH '85)', ACM, New York, NY, USA, pp. 287–296.

Perlin, K. (2002), 'Improving noise', *ACM Transactions on Graphics (TOG)* **21**(3), 681–682.

Prantl, L., Ummenhofer, B., Koltun, V. & Thuerey, N. (2022), 'Guaranteed conservation of momentum for learning particle-based fluid dynamics'.
URL: *https://arxiv.org/abs/2210.06036*

Qi, C. R., Yi, L., Su, H. & Guibas, L. J. (2017), 'Pointnet++: Deep hierarchical feature learning on point sets in a metric space'.
URL: *https://arxiv.org/abs/1706.02413*

Qu, Z., Zhang, X., Gao, M., Jiang, C. & Chen, B. (2019), 'Efficient and conservative fluids using bidirectional mapping', *ACM Transactions on Graphics* **38**(4), 128.

Raissi, M., Perdikaris, P. & Karniadakis, G. E. (2019), 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations', *Journal of Computational Physics* **378**, 686–707.
URL: *https://www.sciencedirect.com/science/article/pii/S0021999118307125*

Ram, D., Gast, T., Jiang, C., Schroeder, C., Stomakhin, A., Teran, J. & Kavehpour, P. (2015), 'A material point method for viscoelastic fluids, foams and sponges', *ACM SIGGRAPH/Eurographics Symposium on Computer Animation* p. 157–163.

Robert Bridson, J. H. & Nordenstam, M. (2007), Curl-noise for procedural fluid flow, *in* 'Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation', SCA '07, Association for Computing Machinery, New York, NY, USA, p. 177–185.
URL: *https://doi.org/10.1145/1276377.1276435*

Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J. & Battaglia, P. W. (2020), 'Learning to simulate complex physics with graph networks'.
URL: *https://arxiv.org/abs/2002.09405*

Schenck, C. & Fox, D. (2018), 'Spnets: Differentiable fluid dynamics for deep neural networks', *2nd Conference on Robot Learning (CoRL 2018)* .

Seymour, M. (2014), 'The science of fluid sims', Online Article. Accessed: 2024-08-13.
URL: *https://www.fxguide.com/fxfeatured/the-science-of-fluid-sims/*

Shi, Y., Huang, Z., Feng, S., Zhong, H., Wang, W. & Sun, Y. (2021), 'Masked label prediction: Unified message passing model for semi-supervised classification'.
URL: *https://arxiv.org/abs/2009.03509*

Stomakhin, A., Schroeder, C., Chai, L., Teran, J. & Selle, A. (2013), 'A material point method for snow simulation', *ACM Transactions on Graphics (TOG)* **32**(4), 102.

Stomakhin, A., Schroeder, C., Jiang, C., Chai, L. & Teran, J. (2014), 'Augmented mpm for phase-change and varied materials', *ACM Transactions on Graphics (TOG)* **33**(4), 138.

Sul, B., Wallqvist, A., Morris, M., Reifman, J. & Rakesh, V. (2014), 'A computational study of the respiratory airflow characteristics in normal and obstructed human airways', *Computers in Biology and Medicine* **52**.

Sulsky, D., Chen, Z. & Schreyer, H. (1994), 'A particle method for history-dependent materials', *Computational Methods in Applied Mechanics and Engineering* **118**(1), 179–196.

Tampubolon, A. P., Gast, T., Klár, G., Fu, C., Teran, J., Jiang, C. & Museth, K. (2017), 'Multi-species simulation of porous sand and water mixtures', *ACM Transactions on Graphics* **36**(4).

Tan, Q., Pan, Z., Gao, L. & Manocha, D. (2019), 'Realtime simulation of thin-shell deformable materials using cnn-based mesh embedding', *arXiv preprint arXiv:1909.12354* .

The Science of Fluid Sims (2011).
URL: *http://www.fxguide.com/featured/the-science-of-fluid-sims*

Tompson, J., Schlachter, K., Sprechmann, P. & Perlin, K. (2017), Accelerating eulerian fluid simulation with convolutional networks, *in* 'Proceedings of the 34th International Conference on Machine Learning'.

Ummenhofer, B., Prantl, L., Thuerey, N. & Koltun, V. (2019), Lagrangian fluid simulation with continuous convolutions, *in* 'ICLR (International Conference on Learning Representations)'.

Ummenhofer, B., Prantl, L., Thuerey, N. & Koltun, V. (2020), Lagrangian fluid simulation with continuous convolutions, *in* 'International Conference on Learning Representations'.
URL: *https://openreview.net/forum?id=B1lDoJSYDH*

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Łukasz Kaiser & Polosukhin, I. (2017), 'Attention is all you need'.
URL: *https://arxiv.org/abs/1706.03762*

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P. & Bengio, Y. (2018), 'Graph attention networks'.
URL: *https://arxiv.org/abs/1710.10903*

Walkley, M. A., Gaskell, P. H., Jimack, P. K., Kelmanson, M. A. & Summers, J. L. (2005), 'Finite element simulation of three-dimensional free-surface flow problems with dynamic contact lines', *International Journal For Numerical Methods in Fluids* **47**(10), 1359–1373.

Wang, C., Wang, Y., Peng, C. & Meng, X. (2016), 'Smoothed particle hydrodynamics simulation of water-soil mixture flows', *Journal of Hydraulic Engineering* **142**(10).

Wang, Y., Jimack, P. K. & Walkley, M. A. (2017), 'One-field monolithic fictitious domain method for fluid-structure interactions', *Computer Methods in Applied Mechanics and Engineering* **317**, 1168–1196.

Wiewel, S., Kim, B., Azevedo, V. C., Solenthaler, B. & Thuerey, N. (2020), Latent space subdivision: Stable and controllable time predictions for fluid flow, *in* 'International Conference on Machine Learning'.

Wojtan, C., Jeschke, S., Chentanez, N., Macklin, M. & Müller-Fischer, M. (2017), 'Large-scale interactive water simulation with directional waves', *ACM SIGGRAPH 2017 Real Time Live!* p. 19.

Wolper, J., Fang, Y., Li, M., Lu, J., Gao, M. & Jiang, C. (2019), 'Cd-mpm: Continuum damage material point methods for dynamic fracture animation', *ACM Transactions on Graphics* **38**(4).

Xie, W., Yang, Z., Yu, F. & Jiang, C. (2019), 'Latent space physics: Towards learning the temporal evolution of fluid flow', *Computer Graphics Forum* **38**(2), 71–82.

Xie, Y., Franz, E. & Chu, M. (2018), 'tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow', *ACM Transactions on Graphics (SIGGRAPH)* **37**(4), 1–15.

Xiong, S., He, X., Tong, Y., Deng, Y. & Zhu, B. (2023), 'Neural vortex method: from finite lagrangian particles to infinite dimensional eulerian dynamics'.
URL: *https://arxiv.org/abs/2006.04178*

Yang, C., Yang, X. & Xiao, X. (2016), 'Data-driven projection method in fluid simulation', *Computer Animation and Virtual Worlds* **27**, 415–424.

York II, A. R. (1997), 'Development of modifications to the material point method for the simulation of thin membranes, compressible fluids, and their interactions', *Sandia Report* .

Yuan, Y., Mehta, A. & Yu, R. (2022), 'Transformer with implicit edges for particle-based physics simulation'.
URL: *https://arxiv.org/abs/2207.10860v1*

Yue, Y., Smith, B., Chen, P., Chantharayukhonthorn, M., Kamrin, K. & Grinspun, E. (2015), 'Continuum foam: a material point method for shear-dependent', *ACM Transactions on Graphics* **37**(4), 160.

Yue, Y., Smith, B., Chen, P., Chantharayukhonthorn, M., Kamrin, K. & Grinspun, E. (2018), 'Hybrid grains: adaptive coupling of discrete and continuum simulations of granular media', *ACM Transactions on Graphics* **37**(6), 283.

Yule, G. U. (1927), 'On a method of investigating periodicities in disturbed series, with special reference to wolfer's sunspot numbers', *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* **226**, 267–298.
URL: *https://royalsocietypublishing.org/doi/10.1098/rsta.1927.0007*

Zhu, F., Zhao, J., Li, Y., Tang, S. & Wang, G. (2017), 'Dynamically enriched mpm for invertible elasticity', *Computer Graphics Forum* **36**, 381–392.