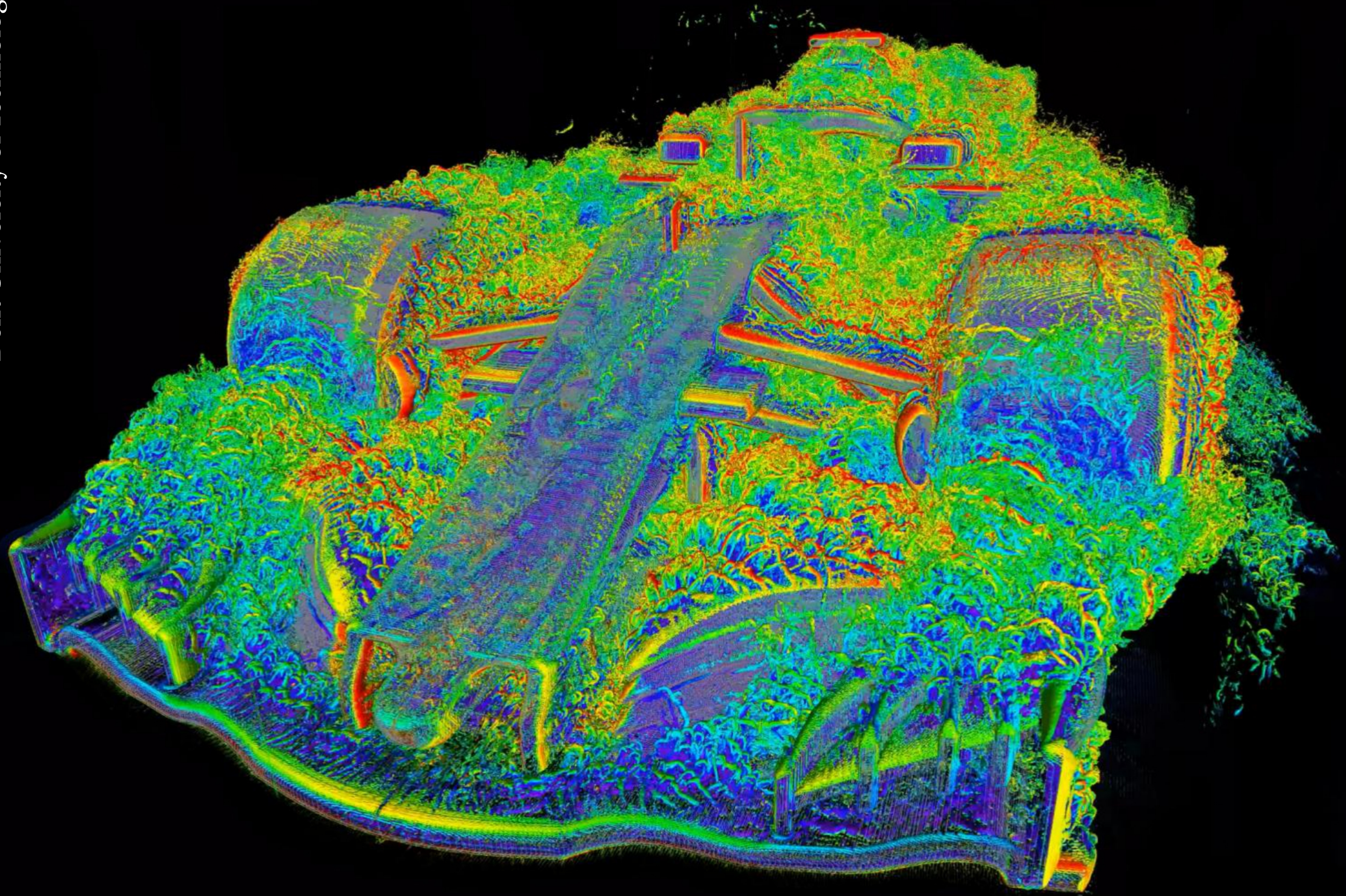
Acceleration of an AMG pressure solver using graph neural networks

Eric Chillón Lizana

Delft University of Technology



TU Delft

Acceleration of an AMG pressure solver using graph neural networks

by

Eric Chillón Lizana

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Thursday December 18, 2025 at 9:30 AM.

Student number:	6070604	
Project duration:	March 20, 2025 – December 18, 2025	
Thesis committee:	Dr. Bernat Font,	TU Delft, Supervisor
	Dr. Nguyen Anh Khoa Doan,	TU Delft, Supervisor
	Dr. Gabriel Weymouth,	TU Delft, Examiner
	Dr. Richard Dwight,	TU Delft, Chair

Cover: Dr. Moritz Lehmann (FluidX3D)

Style: TU Delft Report Style, with modifications by Eric Chillon and Daan Zwaneveld

TU Delft

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisors, Dr. Bernat Font and Nguyen Anh Khoa Doan. Thank you for your invaluable feedback, your guidance, and for always being available whenever I faced challenges during this project.

I would also like to extend a special thanks to Dr. Artur Lidtke. Even though you were not officially part of the committee, your willingness to help and your insights were incredibly supportive throughout this process.

To my colleagues and friends, both those I met here at TU Delft and my old friends from home: thank you for being there during the moments when I needed to disconnect and recharge. A special mention goes to Aitor, for those much-needed relaxing afternoons, and to Joaquin, for the padel sessions that made the hard work easier to bear.

Eric Chillón Lizana
Graus, December 2025

Abstract

Computational fluid dynamics simulations are commonplace and crucial in many industries, yet obtaining flow solutions requires solving large, sparse linear systems of equations at a significant time and energy cost. This becomes particularly expensive, especially on unstructured meshes. One of the most effective methods for solving such systems is the algebraic multigrid (AMG). Still, its performance depends heavily on the efficiency of its internal components, particularly the smoother. Many existing data-driven AMG enhancements mainly target structured grids or focus on other operators rather than smoothing, leaving a gap in unstructured mesh performance optimization.

This study develops a graph neural network (GNN) inspired by AutoAMG, a GNN-based approach to enhance AMG performance by predicting optimal strength parameters. The GNN in this work is trained to predict polynomial coefficients used to construct a sparse pseudo-inverse smoother matrix, improving the AMG smoothing step. The framework is implemented in PyTorch with graphics processing unit (GPU) acceleration, while the training and evaluation datasets are generated using the CFD solvers WaterLily and ReFRESCO. These cover synthetic problems, canonical external flows, and industry-relevant 2D simulations of airfoils (from the AirfRANS dataset) for both structured and unstructured meshes. The framework includes the graph generation, GNN architecture, AMG integration, and a training and validation pipeline.

Excellent generalization is demonstrated for structured cases, achieving approximately 25% wall-clock time reduction and nearly perfect convergence on grids up to 128 times larger than the ones used in training. Similarly, for unstructured 2D/3D problems, data diversity proves to be the primary driver of generalization. A mixed dataset produces stable acceleration across all meshes, achieving 17-18% solve time reductions on grids up to 20 times larger number of cells. Tests further demonstrate that, while different GNN sizes can enhance solving times, performance is ultimately limited by the smoother itself. Increasing the pseudo-inverse complexity, such as using a high polynomial functions to construct the coefficients, can cause overfitting.

Testing on the complex AirfRANS dataset shows that the model generalizes to significantly larger problems and unseen flow regimes, reducing iterations by nearly 20%. However, due to higher sparse smoother costs, this resulted in a 4.7% net slowdown in solve time. In conclusion, this research demonstrates that a GNN-tuned Jacobi smoother can accelerate AMG solvers on diverse grids, averaging 17-25% wall-clock reductions. The framework demonstrates that a GNN can provide quantifiable speedups, given that the reduction in iteration count outweighs the higher computational cost of the sparse smoother.

Contents

Acknowledgments i

Abstract ii

Acronyms xi

1 Introduction 1
- 1.1 The relevance of computational fluid dynamics 1
- 1.2 Solver challenges 1
- 1.3 Algebraic multigrid 3
- 1.4 Research proposal 5
- 1.5 Report structure 6

2 Literature review 7
- 2.1 Data-driven surrogate solvers 8
- 2.2 Optimizing algebraic multigrid 8
  - 2.2.1 Optimal strength parameters θ 8
  - 2.2.2 Enhanced coarsening algorithm 9
  - 2.2.3 Improving prolongation 9
  - 2.2.4 Addressing coarsening and prolongation 10
  - 2.2.5 Tuning smoothers 11
- 2.3 Summary 11
- 2.4 Knowledge gaps 12
- 2.5 Research questions 12

3 Foundation of Algebraic Multigrid 14
- 3.1 The essence of multigrid 14
- 3.2 AMG Basics 15
  - 3.2.1 Algebraic smoothness 16
- 3.3 AMG Components 17
  - 3.3.1 Coarsening 17
  - 3.3.2 Interpolation 18
  - 3.3.3 Smoothing 19
  - 3.3.4 AMG cycles 20

4 Basics of Graph Neural Networks 22
- 4.1 Overview of Deep Learning 22
  - 4.1.1 Neural networks 22
  - 4.1.2 Optimization algorithms 25
  - 4.1.3 Regularization 26

4.2 Introduction to Graph Neural Networks . . . 28
4.2.1 Definition of graph . . . 28
4.2.2 Graph network block . . . 28
4.2.3 Designing graph network architectures . . . 30
4.2.4 Graph convolutional network . . . 31
4.2.5 Graph isomorphism network . . . 32

5 Methodology **34**
5.1 Data generation and processing . . . 34
5.1.1 WaterLily . . . 35
5.1.2 ReFRESCO . . . 37
5.1.3 AirfRANS dataset . . . 42
5.1.4 Dataset summary . . . 43
5.2 AMG-GNN framework . . . 43
5.2.1 GNN architecture . . . 44
5.2.2 Graph generation . . . 47
5.2.3 AMG implementation . . . 47
5.2.4 Training process . . . 50
5.2.5 Validation process . . . 56

6 Hyperparameter tuning & optimization **59**
6.1 Optimization strategy . . . 59
6.2 Hyperparameter study . . . 60
6.2.1 Learning rate and scheduler . . . 60
6.2.2 Mini-batch Size . . . 61
6.2.3 Model architecture (depth and width) . . . 61
6.2.4 Optimization and regularization . . . 61
6.3 Consolidated optimal setup . . . 62

7 AMG-GNN Generalization **64**
7.1 Structured dataset . . . 64
7.1.1 Synthetic dataset . . . 65
7.1.2 Unsteady cases . . . 74
7.1.3 Key findings . . . 79
7.2 Unstructured dataset . . . 80
7.2.1 Two-dimensional problems . . . 80
7.2.2 Three-dimensional problems . . . 83
7.2.3 Key findings from unstructured datasets . . . 87
7.3 AirfRANS . . . 89
7.3.1 Generalization performance . . . 89
7.3.2 Sensitivity to strength parameter θ . . . 89

8 Conclusions **92**
8.1 Main research question . . . 92
8.2 Secondary research questions . . . 93
8.3 Future work . . . 95

A Extra methodology **102**
A.1 Tools and computational resources . . . 102
A.2 Training setup examples . . . 103

A.2.1 Adam and AdamW . . . 103

B GNN Optimization results **104**
B.1 Initial tests . . . 104
B.1.1 Model evaluation . . . 106
B.2 Hyperparameter study . . . 107
B.2.1 Learning rate . . . 107
B.2.2 Mini-batch size . . . 108
B.2.3 Model size . . . 109
B.2.4 Scheduler . . . 110
B.2.5 Dropout . . . 111
B.2.6 L-BFGS . . . 112

C Structured generalization **113**
C.1 Synthetic 2D/3D . . . 113
C.2 Unsteady . . . 114

D Unstructured generalization **115**
D.1 Impact of data diversity . . . 115
D.2 Effect of mini-batch size . . . 116
D.3 Influence of multi-V-cycle training . . . 117
D.4 Enhanced dataset problems . . . 119
D.5 Non-convergence classification . . . 120
D.6 3D-only dataset . . . 122
D.7 Mixed dataset . . . 123

List of Figures

1.1.1 Flow over the A320 tRs prototype: Q-criterion contours colored with velocity magnitude and density gradients displayed at the symmetry plane. Source: [8] 2
1.2.1 Matrices of coefficients from diverse problems and grids. 3
1.3.1 Performance of different solvers with block Jacobi as preconditioner for a lid-driven cavity flow in a coarse and fine mesh. Source: ExaSimple, a previous project from MARIN [21] funded by the EuroHPC inno4scale program. 4
1.3.2 Graph representations of matrices of coefficients for two diverse grids. Darker colors are associated with larger absolute values. 5

2.2.1 Performance gain P of an AMG-ANN algorithm and maximum theoretical performance P_{MAX} for three test cases. Source: [23]. 9
2.2.2 Comparison between RL and greedy algorithms on the F-fraction (higher is better) and effective convergence (lower is better) metrics, for all families of test grids. Source: [52]. 9
2.2.3 Asymptotic convergence factors (smaller is better) for FEM problems. Each problem is tested on V- and W-cycles, and averaged over 100 runs for each problem size. Source: [27]. 10
2.2.4 Number of iterations and runtime required by multigrid solvers for solving the rotated Laplacian problems on a cylinder domain (top two figures) and an L-shaped domain (bottom two figures). Source: [29]. 11

3.1.1 The error reduction coefficients for the stationary Richardson iteration applied to a linear system of equations for $n = 11$ eigenvalues. Source: [50]. 15
3.1.2 Simple scheme of the effect of coarsening in the frequency component of a variable. Source: [59]. 15
3.2.1 Simple schematic of interpolation error (aliasing) as a result of using a non-smooth variable. Source: [59]. 16
3.2.2 Illustration of algebraic smoothness and anisotropy effects in AMG. Source: [20] 18
3.3.1 V-cycle on a 6-level multigrid. Source: [65]. 20

4.1.1 Schematic of a feedforward network with one hidden layer. Each node represents a hidden unit (blue), an input unit (green), or an output unit (red). The arrows indicate the direction in which the information flows. 23
4.1.2 Schematic of a small section of a neural network with back-propagation. The bar over z indicates that the value carries the back-propagation ($\bar{z} = \partial z / \partial x_i$). Source: [66]. 24
4.1.3 Activation functions (blue) and their derivatives (orange). 25
4.1.4 Non-convex landscape of the cost function of two modern neural network architectures. Source: [70]. 25
4.1.5 Simple illustration of underfitting (a), adequate fitting (b) and overfitting (c). . 26

4.1.6 Illustration of training (blue) and validation (green) loss. Notice how the validation error increases after a certain number of epochs, even though the blue curve keeps decreasing. Source: [31]. 27
4.2.1 Graph representation based on this work. Adapted from: [32]. 28
4.2.2 Updates within a GN block. Blue highlights the element being updated, while black elements are the inputs to the update function. The pre-update value of the blue element is also used as an input. Source: [32] 30

5.1.1 Initial residual fields for each of the synthetic cases ($n = 32$). 36
5.1.2 Pressure fields for each of the unsteady cases with $p = 5$. 37
5.1.3 Illustrative schemes of Poiseuille and turbulent channel flows. 38
5.1.4 Illustrative schemes of convection-diffusion and flat plate flows. 39
5.1.5 Example of meshes used throughout the study. 42
5.1.6 Illustration of a grid for an AirfRANS case. 43
5.2.1 GCIN architecture. Definitions: node degree (deg), hidden dimension (HD), diagonal dominance factor (DDF), size of the coefficients matrix (N_A). 45
5.2.2 Loss history for three different activation functions before the predictive MLP. . 46
5.2.3 Illustration of level 2 of AMG hierarchy for a plate flow with N_c and triangle Delaunay meshing. 48
5.2.4 Loss history for two different polynomial degrees for constructing $\mathbf{Q}^{-1}$. 50
5.2.5 Loss history for two different GNN initializations. 54
5.2.6 Train and test loss history on 2D static synthetic cases. Note that, after a minimum loss is reached, both train and test losses increase again. 55
5.2.7 Train/test flowchart. 55
5.2.8 Epoch flowchart. 56
5.2.9 Effect of mini-batch size on validation granularity. Smaller batches ($B = 5$) reveal specific problem outliers that are smoothed out in larger batches ($B = 20$). 57
5.2.10 Validation flowchart. 58

7.1.1 Loss history for a 2D static dataset ($n = 32$) with the old GNN architecture. . . 65
7.1.2 Loss history for the union dataset with the new GNN architecture. 66
7.1.3 Residual reduction factor ($\log_{10}$) for two models. NaNs represent residual divergence. Vertical axis: test dataset. Horizontal axis: train dataset. 67
7.1.4 AMG-GNN residual reduction loss without fixed colormap limits. 68
7.1.5 Residual fields of diverse two-dimensional problems before and after V-cycle for static (a,b,c), dipole (d,e,f), and sphere (g,h,i). Size is 32^2. 69
7.1.6 Validation results (total time) for 2D sphere dataset for 50 problems 128^2. . . . 70
7.1.7 Validation results (total time) for 3D sphere dataset for 50 problems 64^3. . . . 70
7.1.8 Validation results (new training union) for 3D sphere dataset for 50 problems 64^3 and tolerance reduced to $\delta = 1 \times 10^{-6}$. 71
7.1.9 Solve time for each of the six synthetic cases validated on the new union model. 72
7.1.10 Residual norm versus the number of V-cycles for 3D static, dipole, and sphere cases of size 32^3. 72
7.1.11 Solve time for each of the six synthetic cases validated on four different models. 73
7.1.12 Loss history for third-degree polynomial functions for $\tilde{\mathbf{A}}^{-1}$. 73
7.1.13 Relative solve time of synthetic cases for ten trained models. 74
7.1.14 Residual reduction for finest circle, donut, and TGV cases for two different solvers. 75
7.1.15 Residual fields of diverse two-dimensional problems before and after V-cycle for circle (a,b,c), donut (d,e,f), and TGV (g,h,i). Size is $p = 5$. 76

7.1.16 Relative solve time to relaxed Jacobi for each of the three unsteady cases using basic setups with and without global attributes. 77
7.1.17 Relative solve time to relaxed Jacobi for each of the three unsteady cases using different train models. 77
7.1.18 Residual norm versus the number of V-cycles for circle, donut, and TGV cases of size $p = 5$. Results are obtained using the transfer union. 78
7.1.19 Relative solve time to relaxed Jacobi for each of the three unsteady cases based on models trained on different numbers of V-cycles. 78
7.1.20 Relative solve time as a function of the test loss for the three unsteady cases. Using synthetic transfer models. 79
7.2.1 Relative solve time to relaxed Jacobi for 5 models trained on the union dataset. Evaluated on datasets of size $N_c = 79, 111$ and 159. 81
7.2.2 Relative solve time to relaxed Jacobi for 5 models trained on the union dataset and float32. Evaluated on datasets of size $N_c = 79, 111$ and 159 and $\delta = 10^{-4}$. 82
7.2.3 Relative solve times for 5 trained models with different data types. Validated in 2D ($N_c = 156$) and 3D ($N_c = 15, 24$). 85
7.2.4 Relative solve times for 5 trained models on FP32 and a larger strength parameter ($\theta = 0.5$). Validated in 3D ($N_c = 15, 24$). 85
7.2.5 Relative solve times for 5 trained models with base configuration. Validated in 2D ($N_c = 156$) and 3D ($N_c = 15, 24$). 87
7.3.1 Validation results for the best test model for mixed dataset (2D/3D), FP32 and $B = 10$. 89
7.3.2 Number of V-cycle iterations of three different problems as a function of the strength parameter. Source: [24]. 90
7.3.3 Validation results for the best test model for mixed dataset (2D/3D), FP32 and $B = 10$ and $\theta = 0.2$. 90

A.2.1 Loss history for two different optimizers. 103

B.1.1 First two training on different hidden dimensions. 104
B.1.2 Effect on training of two more changes maintaining HD64. 104
B.1.3 Effect of five different learning rates on training. 105
B.1.4 Effect on training of three schedulers. The learning rate history was not implemented at this point. 105
B.1.5 Total time to converge (Setup + Solve) for AMG-GNN and AMG-Jacobi. 106
B.1.6 Effect on training of two more changes maintaining HD64. 106
B.2.1 Loss history for the different learning rates. 107
B.2.2 Total times of diverse configurations of learning rates. The validation dataset is 50 problems with $N_c = 28$. 107
B.2.3 Loss history for the different batch sizes. 108
B.2.4 Total times of diverse configurations of batch sizes. The validation dataset is 50 problems with $N_c = 28$. 108
B.2.5 Total times for $B = 5$ for models saved at different epochs. 108
B.2.6 Loss history for the different model sizes. 109
B.2.7 Total times of diverse configurations of model sizes. The validation dataset is 50 problems with $N_c = 28$. 109
B.2.8 Total times for $B = 5$ for models saved at different epochs. 109
B.2.9 Loss history for the different schedulers. 110

B.2.10 Total times of diverse configurations of schedulers. The validation dataset is 50 problems with $N_c = 28$. . . . 110
B.2.11 Loss history for the different dropouts. . . . 111
B.2.12 Total times of diverse configurations of dropout. The validation dataset is 50 problems with $N_c = 28$. . . . 111
B.2.13 Training and validation results pre-L-BFGS. . . . 112
B.2.14 Training and validation results for L-BFGS. . . . 112
B.2.15 Validation results (total time) for different datasets. . . . 112

C.1.1 Train and test losses for different types of synthetic data. The dataset is a 200/50 split for $n = 32^2$ (2D) and $n = 16^3$ (3D). . . . 113
C.2.1 Loss history of unsteady problems for different setups: circle (a,b,c), donut (d,e,f), and TGV (g,h,i) for $p = 3$, combination of $p = [3, 4]$, and $p = 4$ with multi-v-cycle loss. . . . 114

D.1.1 Train and test loss history for two Poiseuille-only trainings. . . . 115
D.1.2 Train and test loss history for union trainings. . . . 115
D.1.3 Relative solve time to relaxed Jacobi for 10 models trained on Poiseuille only. Evaluated on Poiseuille datasets of size $N_c = 79$ and $N_c = 111$. . . . 116
D.1.4 Relative solve time to relaxed Jacobi for 5 models trained on the union dataset without global attributes. Evaluated on datasets of size $N_c = 79, 111$ and 159. . . . 116
D.2.1 Relative solve time to relaxed Jacobi for 3 models trained on the union dataset (up to $N_c = 40$) and $B = 1, 5, 10$. Evaluated on datasets of size $N_c = 79, 111$ and 159. . . . 117
D.3.1 Relative solve time to relaxed Jacobi for 6 models trained on the union dataset and different V-cycles. Evaluated on datasets of size $N_c = 79, 111$ and 159 and $\delta = 10^{-4}$. . . . 118
D.4.1 Train history for train dataset including $N_c = 56$, using $B = 6$ and FP32. . . . 119
D.4.2 Relative solve time to relaxed Jacobi for 5 models trained on the union dataset up to $N_c = 56$ ($B = 6$, FP32). Evaluated on datasets of size $N_c = 79, 111$ and 159 and $\delta = 10^{-4}$. . . . 119
D.6.1 Train and test loss history for two Poiseuille-only trainings. . . . 122
D.6.2 Effect of modifying the strength parameter on the sparsity (and size) of the coefficient matrix. Graphs $G^{(3)}$ are presented to illustrate the change in nodes. 122
D.7.1 Train and test losses for different types of unstructured data. The dataset is a 700/300 split of 2D/3D problems. . . . 123
D.7.2 Relative solve times for 5 trained models with HD96. Trained and validated in 2D and 3D. . . . 123
D.7.3 Relative solve times for 5 trained models with different datasets. Validated in $N_c = 15$ and 24 (3D). . . . 124
D.7.4 Relative solve times for 7 trained models with $B = 6$. Validated in 3D. . . . 124
D.7.5 Relative solve times for 5 trained models with model changes. Validated in $N_c = 15$ and 24 (3D). . . . 125

List of Tables

5.1.1 Main setup for the unsteady data generation problems. 37
5.1.2 CFD setup per case. Density is $\rho = 1$. 39
5.1.3 Boundary conditions per case (axes as in the dataset). 40
5.1.4 Number of cells per case, N_c, and mesh type for two-dimensional data. Definitions: i) Structured, ii) Triangle Delaunay, iii) Triangle-Quad Delunay (Quad-dominant), iv) Triangle Advancing Front, v) Triangle-Quad Advancing Front, and v) Triangle Advancing Front Ortho (with orthogonal smoothing). 41
5.1.5 Number of cells per case, N_c, and mesh type for three-dimensional data. Definitions: i) Structured, ii) Triangle Delaunay, iii) Triangle-Quad Delunay (Quad-dominant), iv) Triangle Advancing Front, v) Triangle-Quad Advancing Front, and v) Triangle Advancing Front Ortho (with orthogonal smoothing). 41
5.1.6 Summary of different datasets used in the study. 43

6.2.1 Base training setup for the hyperparameter study. 60
6.3.1 Consolidated setup. 63

7.2.1 Initial training setup for two-dimensional datasets. 81
7.2.2 Setup changes for the 3D-only dataset. 84
7.2.3 Setup changes for the mixed dataset. 86

Acronyms

Acronym	Definition
1-WL	Weisfeiler-Lehman
2D	Two-dimensional
3D	Three-dimensional
AMG	Algebraic Multigrid
ANN	Artificial Neural Network
BFGS	Broyden-Fletcher-Goldfarb-Shanno
BiCGSTAB	Biconjugate gradient stabilized method
CFD	Computational fluid dynamics
CG	Conjugate gradient
CNN	Convolutional Neural Network
ConvDiff	Convection-Diffusion
CPU	Central Processing Unit
CSR	Compressed Sparse Row
DDF	Diagonal Dominance Factor
DL	Deep Learning
EC	Exit code
FEM	Finite Element Method
FP32	Floating Point 32-bit (Single Precision)
FP64	Floating Point 64-bit (Double Precision)
FVM	Finite Volume Method
GA	Global Attributes
GCIN	Graph Convolutional Isomorphism Network
GCN	Graph Convolutional Network
GIN	Graph Isomorphism Network
GMG	Geometric Multigrid
GMP	Global mean pool
GMRES	Generalized minimal residual method
GN	Graph Network
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GS	Gauss-Seidel
HD	Hidden Dimension
HPC	High-Performance Computing
ILU	Incomplete LU factorizations
JIT	Just-In-Time compilation
LBFGS	Limited-BFGS
LWSQ	Local weighted least squares
MAE	Mean absolute error
MARIN	Maritime Research Institute Netherlands
MG	Multigrid methods
MLP	Multilayer Perceptron

Acronym	Definition
MPNN	Message Passing Neural Network
MSE	Mean squared error
MVC	Multi-V-Cycle
NACA	National Advisory Committee for Aeronautics
PDE	Partial Differential Equation
PINNs	Physics-Informed Neural Networks
PISO	Pressure-Implicit with Splitting of Operators
PReLU	Parametric Rectified Linear Unit
RAM	Random Access Memory
RANS	Reynolds-Averaged Navier-Stokes
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SIMPLE	Semi-Implicit Method for Pressure Linked Equations
SOR	Successive over-relaxation
SPD	Symmetric positive definite
SST	Shear Stress Transport
TAGConv	Topology Adaptive Graph Convolution
TGV	Taylor-Green vortex
VRAM	Video Random Access Memory

1

Introduction

1.1. The relevance of computational fluid dynamics

Computational fluid dynamics (CFD) is nowadays a vital tool used in several sectors such as the aerospace, for aircraft design aiming for fuel reduction [1], reducing transonic instabilities [2] (see Figure 1.1.1 for transonic flow visualization) or diminishing noise by analyzing the pressure fluctuations [3], the energy with optimal design of wind turbines for better energy production [4], the climate where large scale turbulence with multiphase flow are combined for atmospheric dispersion prediction [5], or the biomedical, simulating blood flow in complex geometries after assist devices are installed to ensure smooth flows and clot formation [6]. Furthermore, the maritime sector also has important advances in ship propulsion performance thanks to CFD [7].

Each engineering sector solves a different problem, but at the heart of any fluid challenge there are the Navier-Stokes equations, which may be subject to simplifications depending on the characteristics of the problem. Accordingly, the aerodynamics sector tackles all types of flows. That is, subsonic incompressible and compressible, transonic, supersonic, and hypersonic. However, the other areas of industry and research generally focus on incompressible flow due to working with low velocities relative to the sound speed or as a result of dealing with incompressible fluids such as water or oil. For instance, pollution particle transport over a city, or the flow over a ship hull, are well-known examples. Therefore, incompressible flows are one of the most researched flows, and work made to improve the simulation capabilities of this flow would be largely beneficial.

1.2. Solver challenges

Solving the incompressible Navier-Stokes equations presents a significant challenge due to the lack of an explicit equation for pressure. In incompressible flow, the continuity equation acts as a constraint on the velocity field, ensuring it remains divergence-free, but it does not directly provide a way to determine the pressure. To overcome this, a Poisson equation for pressure is derived by taking the divergence of the momentum equation. This decouples the calculation of velocity and pressure. Among the various approaches, the most common are the pressure projection method [9] and pressure-velocity coupling methods like SIMPLE and PISO [10]. These methods solve a Poisson-like elliptic equation for the pressure using an intermediate or *predicted* velocity field, which, after the discretization, becomes a large and sparse linear equation system of the form $\mathbf{A}\mathbf{u} = \mathbf{f}$, where $\mathbf{A}$ is the matrix of coefficients that results from the spatial discretization. This process involves dividing the continuous physical domain of the fluid flow into a finite number of discrete volumes or cells. The governing partial differential equations are then approximated by a system of algebraic equations over these discrete elements. For each cell, this results in a set

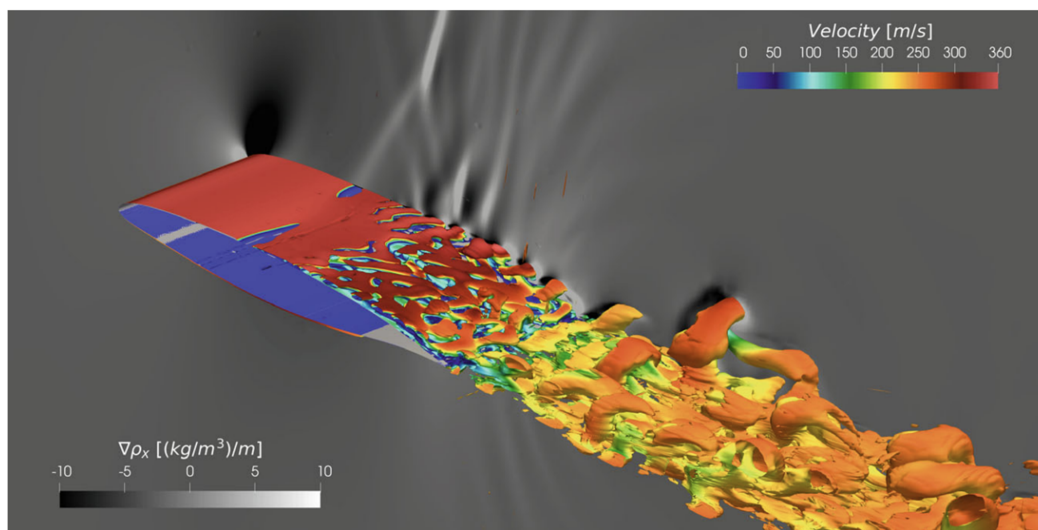
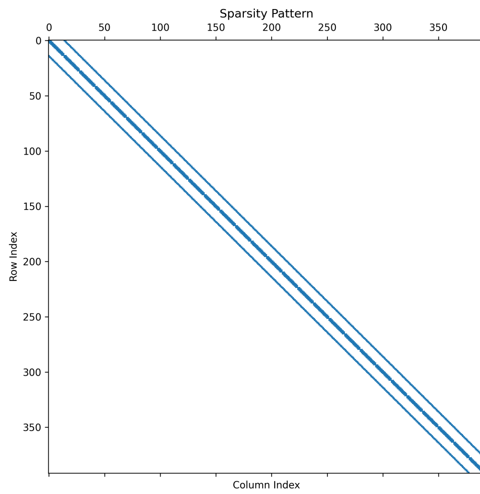


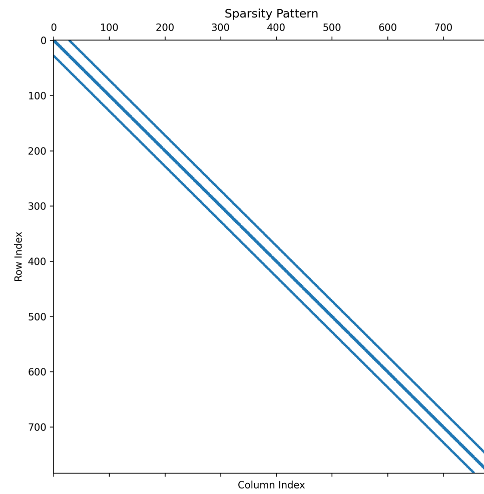
Figure 1.1.1: Flow over the A320 tRs prototype: Q-criterion contours colored with velocity magnitude and density gradients displayed at the symmetry plane. Source: [8]

of linear equations relating the variable (e.g., pressure) in that cell to its neighbors, ultimately forming the large, sparse linear system. In these flows, the solution of this equation generally dominates the computation time of each time step and thus, the entire simulation time. This can take up to 95% of the total wall-time on large-scale systems with thousands of processor cores [11]. It should be noted, however, that on systems with extremely large numbers of processors, communication time between them can consume up to half of the task's time [12]. Hence, the wise selection of a linear solver to obtain an efficient solution for the Poisson equation is a key aspect of CFD, which is constantly investigated, as computation time is one of the main challenges that incompressible workflows address.

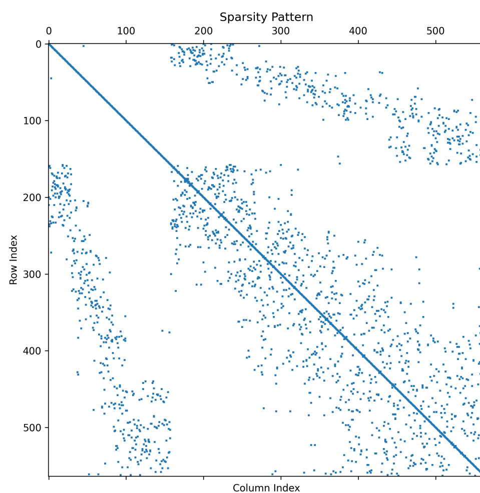
Concerning the solver, problems are sometimes limited by geometry or meshes. For complex geometries, unstructured meshes are required to fit the shape, which gives flexibility but produces sparse matrices without a regular structure and varying coefficients due to different element sizes and shapes (see Figure 1.2.1 for visualization of sparsity patterns of two distinct problems and meshes). This can produce ill-conditioned linear systems with a large condition number and a wide range of eigenvalues, sometimes referred to as the stiffness of the system, and slows convergence for iterative solvers [13]. While direct solvers are more robust, problems that require large meshes (high cell count) are not feasible to be solved by direct solvers due to extremely large memory constraints and computational blow-up. The solve complexity ranges between $\mathcal{O}(n^{1.5})$ and $\mathcal{O}(n^2)$ for sparse matrices, and $\mathcal{O}(n^3)$ for dense [14], where n is the size of the matrix ($n \times n$). Therefore, the only feasible solvers are iterative in nature and as parallelizable as possible [15]. Unlike direct solvers, the computational cost per iteration for these methods scales much more favorably. With respect to iterative methods, the computational cost will depend on the iteration cost and the number of iterations. In general, dense matrices will present an iteration cost of approximately $\mathcal{O}(n^2)$, while sparse matrices will require $\mathcal{O}(nnz)$ operations, with nnz referring to the number of non-zero entries of the matrix [16]. The number of iterations will generally be influenced by the condition number of the matrix.



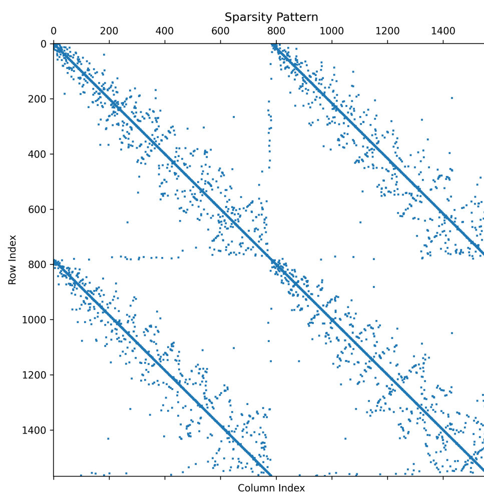
(a) Poiseuille flow. Structured grid.



(b) Convection-diffusion flow. Structured grid.



(c) Poiseuille flow. Triangle and quadrilateral grid (Advancing front).



(d) Convection-diffusion flow. Triangle and quadrilateral grid (Advancing front).

Figure 1.2.1: Matrices of coefficients from diverse problems and grids.

1.3. Algebraic multigrid

Within the existing families of linear solvers, the choice is heavily influenced by the scale of modern simulations. As computational power has grown, so has the size and complexity of the grids used, often containing millions or billions of cells. For the massive linear systems generated from these large grids, direct solvers are not practical due to their prohibitive computational and memory costs. Consequently, iterative solvers represent the only reasonable method to use. The foundational class of iterative solvers includes stationary methods like the Jacobi method, Gauss-Seidel (GS), and Successive over-relaxation (SOR). While these methods are often the simplest to implement, they generally provide the slowest rates of convergence. Furthermore, the sequential nature of GS and SOR makes them difficult to parallelize. The Jacobi method is a notable exception. Its

update scheme is inherently parallel, and it is also frequently used as a simple preconditioner [17]. The next solvers in terms of performance are the Krylov subspace methods such as the Conjugate gradient (CG), Generalized minimal residual method (GMRES), or Biconjugate gradient stabilized method (BiCGSTAB), which present a faster convergence with a good preconditioner, which is generally required, and can be applied to a wide range of systems [17]. The most advanced and complex linear solvers are based on multigrid methods, such as Geometric Multigrid (GMG) and Algebraic Multigrid (AMG) [18]. While linear solvers are agnostic to the underlying discretization scheme, multigrid methods are exceptionally effective for solving the large, sparse, and often ill-conditioned linear systems that typically arise from Finite Volume Method (FVM) and Finite Element Method (FEM) discretizations of partial differential equations. One of the limitations of the GMG is that it is restricted mainly to structured grids, although it provides faster results, whereas the AMG operates only on the equation system matrix without any prior knowledge of the grid, making it a powerful option for unstructured grids [19]. In addition, multigrid methods are renowned for their high efficiency, with a computational cost that can scale near-linearly with the number of unknowns, n (often cited as $\mathcal{O}(n)$). This optimal performance is achieved through a methodology involving several distinct steps, such as grid coarsening, grid prolongation, and smoothing. These individual components, among others, offer multiple, independent areas for enhancement and optimization [20]. This multi-component methodology is precisely what makes AMG such a powerful and promising tool. Unlike more monolithic iterative solvers, each distinct step of the AMG process can be individually analyzed, tuned, and accelerated. This modularity offers a greater potential for optimization, which, when combined with its ability to operate independently of the grid geometry, makes it a highly adaptable and improvable solver for unstructured meshes.

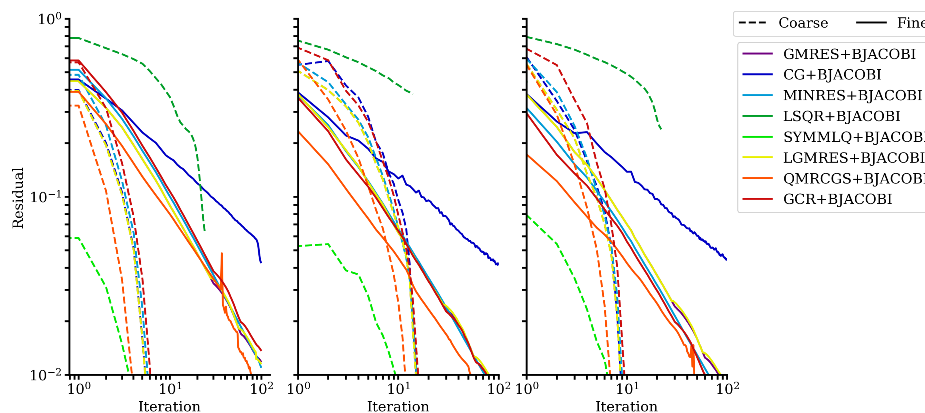
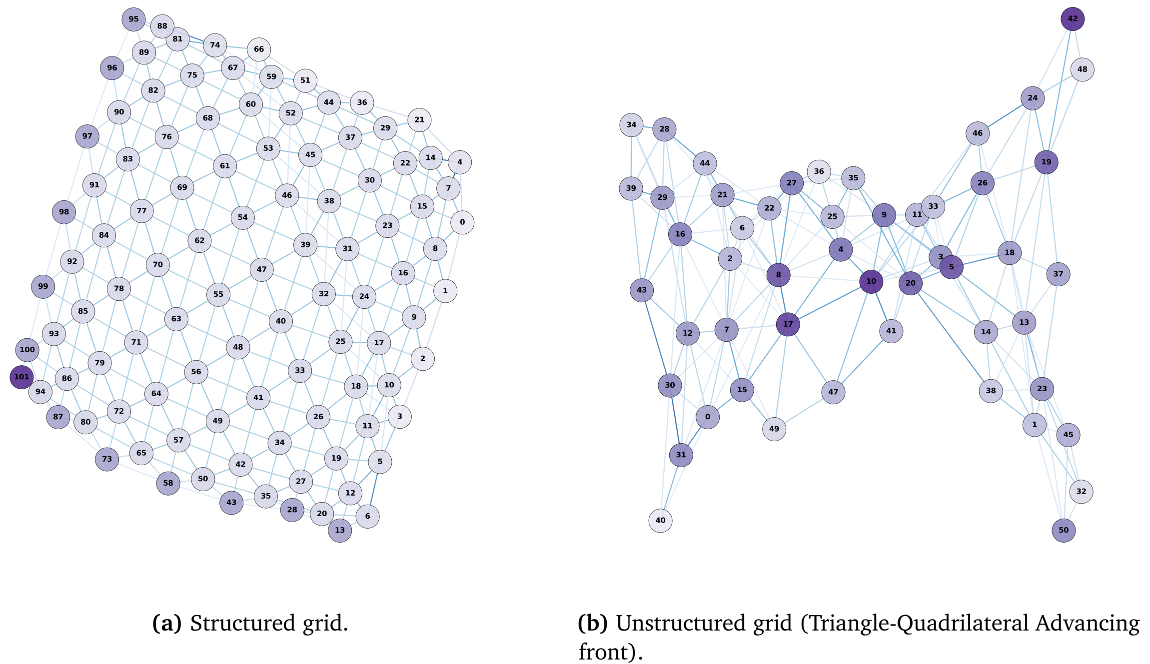


Figure 1.3.1: Performance of different solvers with block Jacobi as preconditioner for a lid-driven cavity flow in a coarse and fine mesh. Source: ExaSimple, a previous project from MARIN [21] funded by the EuroHPC inno4scale program.

Within the AMG framework, which will be expanded in Chapter 3, there are three main steps: the coarsening (or restriction), the smoothing, and the interpolation (or prolongation). There are numerous examples in the literature using data-driven techniques to create more efficient algebraic multigrid solvers tackling each of the different areas [22–30]. However, the vast majority are based on architectures such as Multilayer Perceptrons (MLPs) or Convolutional Neural Networks (CNNs). The former assumes the input is a vector and the matrix structure is inherently lost, while the latter can optimally extract features with kernels when the spatial connections are fixed (e.g., pixels of an image), and therefore are limited to structured grids [31]. One way to handle this connectivity issue on unstructured grids, due to the spatial variation of the matrix of coefficients, is to use a

Graph Neural Network (GNN). This is a result of the sparse matrix of coefficients representing connectivity between grid cells, and so a graph can be directly associated with it. Therefore, using



(a) Structured grid.

(b) Unstructured grid (Triangle-Quadrilateral Advancing front).

Figure 1.3.2: Graph representations of matrices of coefficients for two diverse grids. Darker colors are associated with larger absolute values.

GNNs that learn features based on the graph connectivity could allow the network to generalize to other unstructured grids as opposed to fixed kernels (in terms of structure) from CNNs [32]. In this respect, two graphs corresponding to the middle levels of the AMG hierarchy for a structured and unstructured grid are presented in Figure 1.3.2. On the unstructured grid (Figure 1.3.2b), the complex and not fixed relations make a CNN not a viable approach. CNNs assume that similar spatial relationships repeat across the grid.

With the architecture identified, the remaining challenge is selecting which AMG component to optimize. While data-driven methods for structured grids have covered all AMG steps, the *smoother* for unstructured meshes remains largely unexplored. This gap is significant because, as demonstrated in structured grid research, optimizing the smoothing step typically yields the highest returns in terms of solve time acceleration [29, 30]. Consequently, this research focuses on bridging this gap by developing a learned smoother for unstructured grids.

1.4. Research proposal

Drawing from the state-of-the-art analysis in Chapter 2, this work identifies the smoother as a key component for optimization within the AMG framework. The core of this thesis is to accelerate the AMG solver by developing a data-driven enhancement for the Jacobi smoother. The choice of a Jacobi smoother is to provide a follow-up to the work of Weymouth [30] in the optimization of GMG for structured grids. The choice is also motivated by the ease of implementation and the parallelization properties of the Jacobi smoother.

To achieve this, a GNN architecture is proposed, inspired by the AutoAMG model [24]. This GNN is trained to predict optimal polynomial coefficients, which are then used to construct a sparse pseudo-inverse matrix. This matrix serves as a more effective smoother, a methodology that builds upon the approach presented in [30]. The entire framework is implemented in a PyTorch environment, specifically designed to take advantage of GPU acceleration for the computationally demanding matrix operations involved and the automatic differentiation toolbox that it provides.

1.5. Report structure

The work is structured as follows:

First, Chapter 2 presents a literature review analyzing current data-driven approaches to linear solvers, identifying the optimization of the smoother on unstructured grids as a critical knowledge gap.

Following, Chapter 3 establishes the theoretical foundation of algebraic multigrid, detailing the core components such as coarsening, interpolation, and smoothing, as well as the cycling strategies. Further, Chapter 4 introduces the basics of Deep Learning and graph neural networks, covering the specific architectures used in this work, including graph convolutional networks and graph isomorphism networks.

Further, Chapter 5 outlines the methodology, describing the computational resources, the data generation pipelines (using WaterLily, ReFRESCO, and AirfRANS), and the specific design and implementation of the AMG-GNN framework.

Concerning model training and performance, Chapter 6 details the hyperparameter tuning process, analyzing the model's sensitivity to learning rates, mini-batch sizes, and architecture depth to establish a consolidated, optimal setup for validation. This is followed by Chapter 7, which evaluates the model's generalization capabilities, first validating against structured synthetic and unsteady datasets, then assessing performance on unstructured 2D and 3D grids on aerodynamic problems, and finally testing on the industry-relevant AirfRANS aerodynamic dataset.

Finally, Chapter 8 draws conclusions based on the results, answering the primary and secondary research questions and suggesting routes for future work.

2

Literature review

Research aimed at improving the performance of CFD is extensive, addressing challenges across the entire simulation workflow. This includes advancements in physical modeling, such as turbulence modeling [33–37]. Concurrently, a vast amount of research focuses on enhancing the efficiency of the linear solver system itself. Within this domain, the development of powerful preconditioning techniques [38–42] is a critical area of focus, as these are integral components that accelerate the solver's convergence. These parallel efforts, in both physics and numerics, are driven by the constant search for greater accuracy and computational efficiency in CFD. The efficiency of the linear solver is critical because nearly all other advancements, whether in adaptive meshing, precise time-stepping, or complex turbulence models, ultimately rely on an iterative solution step. The only notable exception involves purely data-driven surrogate approaches that bypass the iterative process entirely to infer a direct solution, as demonstrated by [43]. However, while these methods offer rapid inference, they rely on approximation and lack the mathematical guarantee of convergence required for high-fidelity analysis. Consequently, for applications where precision is non-negotiable, the iterative framework remains essential, making the solver the primary computational bottleneck. Therefore, optimizing this component acts as a force multiplier. For instance, by reducing the number of iterations [44] or optimizing data communication [45], total simulation times can be reduced from weeks to days without sacrificing physical accuracy.

Before deep learning became commonplace, hardware constraints limited multigrid optimization to three main approaches: developing new algorithms, refining existing ones, or applying simple machine learning techniques compatible with available resources. Nowadays, the acceleration and optimization approaches revolve almost completely around deep learning, but they can be divided into two strategies.

The first strategy involves Physics-Informed Neural Networks (PINNs). PINNs represent a surrogate modeling approach that aims to replace the entire traditional solver pipeline. By training a neural network to directly approximate the solution of the continuous PDE, the discretization step and the concept of a linear solver are bypassed entirely.

The second strategy, which is the focus of this thesis, maintains the classical numerical framework to solve the discretized PDE. Instead of replacing the solver, this approach utilizes deep learning to enhance the existing iterative process. The objective is to accelerate performance by learning to optimize specific internal components of the solver, such as the smoother or the prolongation operator within an AMG cycle. By retaining the underlying solver structure, this method ensures that the physics of the discretized system are rigorously solved, while deep networks are used to handle complex, high-dimensional features, particularly on unstructured meshes, that classical heuristics struggle to address. Nonetheless, because this approach relies on learning from simulation history, the risk of overfitting must be considered if the training dataset is not sufficiently large.

2.1. Data-driven surrogate solvers

There is a significant amount of recent research on multigrid methods using data-driven techniques that focus on structured and unstructured data, and the variety of approaches is considerable. Starting from studies that take the use of data-driven techniques further, some researchers decided to mimic the multigrid nature but substitute all the steps and algorithms with a network architecture creating a surrogate model of fluid dynamics [46] based on the U-Net architecture [47]. However, a comparison with canonical multigrid was missing to evaluate the performance more rigorously. In [48], a two-level multigrid was constructed, where each of the levels was completely built by artificial neural networks (ANNs). The main idea revolved around using the second level as a coarse correction for the prediction of the first level. Nonetheless, the results presented are limited to small problems on 2D structured grids due to the ANN capabilities of capturing unstructured features. In [22], the algebraic multigrid nature was followed using multiwavelet transforms as restriction and prolongation operators with learnable weights. This aided in the process of capturing global trends and detailed features within the PDE solutions and obtaining promising results, but was limited to two-dimensional uniform grids. Further, the cost of running the model was not presented. Additionally, although not AMG related, [49] developed multilevel domain decomposition methods using a multigrid approach to GNNs. The multilevel GNN was able to learn from a fine and coarse grid simultaneously, and was used to learn Schwarz preconditioners. The use of GNNs allowed this approach to be based on unstructured meshes.

2.2. Optimizing algebraic multigrid

There are three main ways an AMG solver can be optimized: tuning the coarsening, the smoother, or the prolongation. In this respect, coarsening can be improved either by optimizing the algorithm itself or selecting an optimal strength parameter [20].

2.2.1. Optimal strength parameters θ

At the heart of the AMG coarsening process is the concept of "strength of connection", which quantifies the influence between two variables, i and j, in the linear system. This is typically measured based on the magnitude of the off-diagonal matrix entry a_{ij} relative to others in its row. The strength parameter, typically denoted as θ, acts as a user-defined threshold to formalize this concept. A connection between two nodes is classified as "strong" if its measured strength exceeds θ. This classification is fundamental, as it guides the subsequent algorithm that partitions the grid nodes into coarse-grid points (C-points) and fine-grid points (F-points) [20].

Given its critical role in defining the coarse grids, it is clear that the choice of θ can significantly impact the solver's overall efficiency. Therefore, tuning the strength parameter has become an active area of research, and data-driven approaches have shown promising results in this domain. For instance, [50] used a simple ANN to predict the optimal strength parameter using a pooling analogous to the pooling employed in CNNs. One of the drawbacks of this approach is that the testing was limited to two-dimensional diffusion equations on structured grids, similar to the work of the same authors in [51], which was a tuned version of the original study. A similar research in three-dimensional problems was [23], where the same structure underwent modifications and refinement to provide better predictions, specifically by introducing CNNs, cutting the time cost of the solution up to 30% in some cases (see Figure 2.2.1 to observe how an ANN accelerated significantly the solver, although in some cases it worsens). Although the authors claimed to address unstructured meshes, their use of purely hexahedral grids simplified the task for the

CNNs. An interesting approach to the optimization of the strength parameter was proposed by [24], where the authors used a Graph Convolutional Isomorphism Network (GCIN), which was comprised of a Graph Convolutional Network (GCN) and a Graph Isomorphism Network (GIN). This choice allowed them to create a model that was able to predict the most efficient strength parameter even on unstructured and three-dimensional grids, achieving two-fold speedups or more on average on diffusion problems.

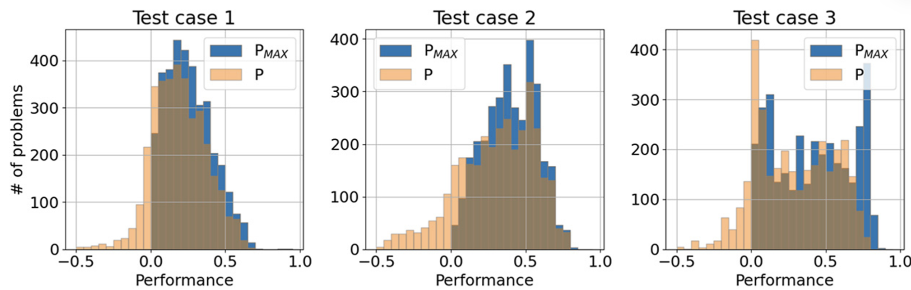


Figure 2.2.1: Performance gain P of an AMG-ANN algorithm and maximum theoretical performance P_{MAX} for three test cases. Source: [23].

2.2.2. Enhanced coarsening algorithm

Concerning the coarsening algorithm, [52] used reinforcement learning in order to train an agent that performed the coarsening rather than an established algorithm. Utilizing a reward that tries to minimize the number of coarse nodes, they obtained a coarsening that requires less memory at the cost of a slight underperformance in the AMG convergence factor (Figure 2.2.2). This makes it an interesting approach for problems that are memory-limited by AMG due to a large hierarchy.

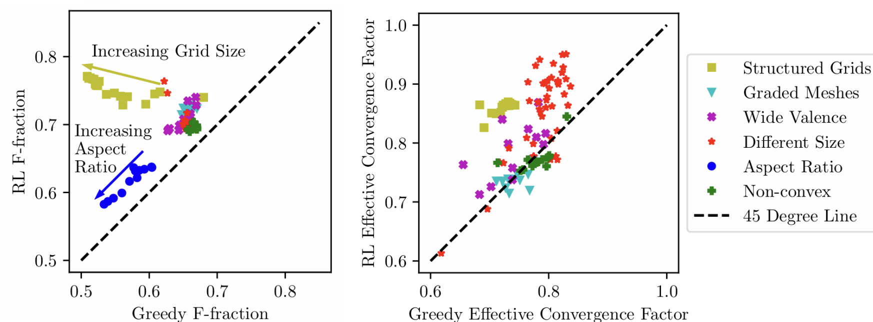


Figure 2.2.2: Comparison between RL and greedy algorithms on the F-fraction (higher is better) and effective convergence (lower is better) metrics, for all families of test grids. Source: [52].

2.2.3. Improving prolongation

Many researchers put all their effort into obtaining a more efficient prolongation operator while using an established coarsening algorithm. In this respect, [27, 53] followed the same approach. Using the prolongation structure associated with the coarsening algorithm, they trained a NN and a GNN, respectively, to obtain optimal weights for the non-zero terms of the prolongation operator. The results in [27] showed a 2-8% improvement in the convergence factor in 79% of

graph Laplacian problems on unstructured grids. Other problems, such as FEM diffusion equations, turned out to perform slightly worse (see Figure 2.2.3 for results with FEM problems). The weak point of these studies is that, in order to obtain efficient training, they trained on a special matrix dataset that limits the richness of the data and features extracted from the GNN, which is also followed by the fact that no clock times for the forward pass of the GNN are presented. In addition to this work, [54] used the previous research as a base to create a more complex architecture that introduced a learning-based local weighted least squares (LWSQ) method. Hence, the objective was to obtain a data-driven approach that is able to interpolate the coarse grid solution into the finer one more effectively, and therefore reduce the number of iterations. In the end, this model outperformed the work from the former papers marginally, considering that the complexity of the architecture was increased significantly.

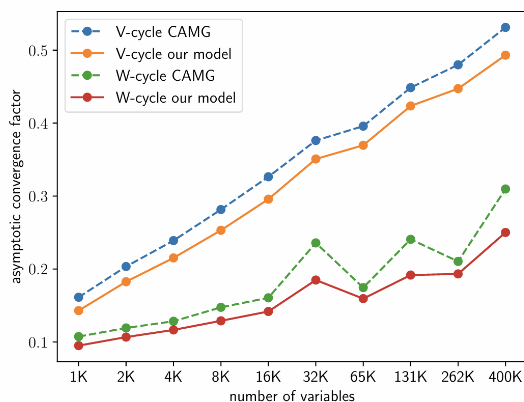


Figure 2.2.3: Asymptotic convergence factors (smaller is better) for FEM problems. Each problem is tested on V- and W-cycles, and averaged over 100 runs for each problem size. Source: [27].

2.2.4. Addressing coarsening and prolongation

Two works, [26] and [25], tried to optimize both coarsening and prolongation within a single framework rather than separately. For example, a recent thesis [26] utilized a graph neural network architecture using message passing neural networks (MPNNs) blocks. This approach tried to mimic iterations of a Lloyd aggregation algorithm to generate different grid levels and their coarsening and prolongation matrices. Further, this strategy always surpassed the original aggregation algorithms in terms of solver convergence for heterogeneous diffusion problems. However, the tested grids were considerably small. An interesting addition to the literature from this paper is the use of a genetic evolution strategy for the training, as the aggregation algorithms result in non-differentiable operations, which makes it more expensive to train than other methods. Additionally, a different approach is presented in [25], which, while not directly altering the coarsening or prolongation algorithms, focused on simplifying the resulting coarse-grid operator. The authors used a neural network to find a spectrally equivalent coarse-grid operator that was significantly sparser, thereby reducing computational complexity while requiring virtually the same number of iterations to converge. Normally, this operator results from the coarsening and prolongation operators. This paper learned a new optimal one from the original local stencil. Nonetheless, it is limited to structured data due to the input of the NN being a constant stencil shape. Further, no evaluation times for the network are provided, an important analysis metric that is often overlooked in the literature.

2.2.5. Tuning smoothers

The optimization of smoothers is considered next. In [28], a neural network was fed with the discretization stencil of the grid. The network then provided the optimal values of the relaxation parameters for a Jacobi or a 4-color Gauss-Seidel smoother. This work also introduced a technique consisting in performing a few genetic algorithm iterations after the Stochastic Gradient Descent (SGD) to obtain a better minimum, resulting in an 18% improvement in average compared to common relaxation values, although it is limited to structured grids. Additionally, authors from [29] introduced a method that substituted the smoother with CNNs. Accordingly, by creating an architecture using convolutional neural networks and fully connected layers that are fed a different number of discretization stencils, one can obtain smoothing kernels that are applied as a convolution to the residual, effectively smoothing it. This approach, although limited to structured grids, presented one of the best performing data-driven methods with 18% reduction in time, which is in general harder to reduce than the iteration count, as a new method can be more computationally expensive than the original per iteration (see Figure 2.2.4). Finally, [30] used a tuned Jacobi smoother for a GMG. Instead of using the inverse of the diagonal as the smoother matrix, a pseudo-inverse with the same sparsity as the original equation matrix was built. This new pseudo-inverse was generated using quadratic formulas whose coefficients are obtained via a data-driven approach, using automatic differentiation in the whole multigrid algorithm, and aiming to reduce the residual as much as possible after a GMG V-cycle. This results in a tuned smoother that surpasses common smoothers such as Gauss-Seidel or SOR by an 80-210% relative speedup in convergence time, yet it is limited to structured grids.

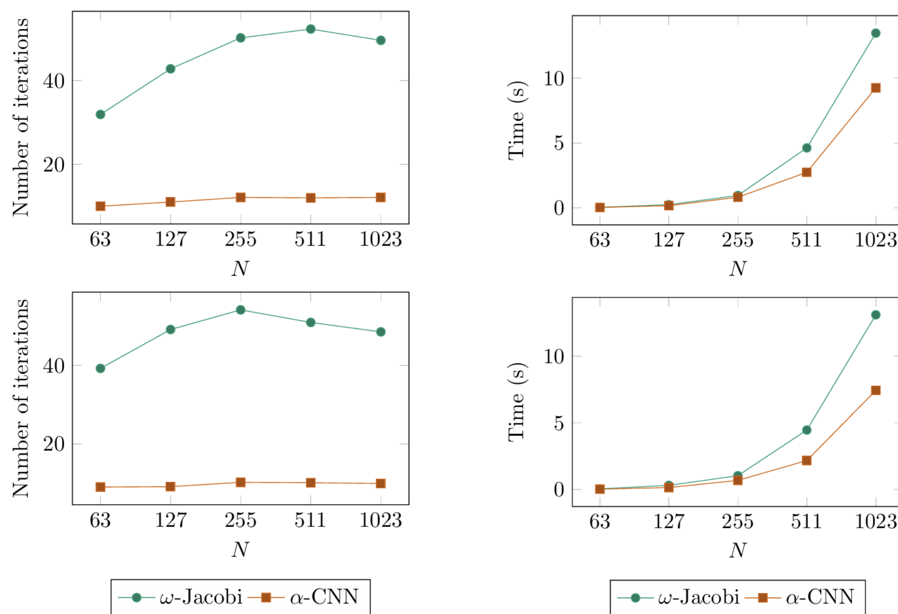


Figure 2.2.4: Number of iterations and runtime required by multigrid solvers for solving the rotated Laplacian problems on a cylinder domain (top two figures) and an L-shaped domain (bottom two figures). Source: [29].

2.3. Summary

In conclusion, this review of the extensive research on accelerating multigrid solvers reveals a clear and promising direction for future work. The most effective approach for achieving significant

performance gains, particularly for practical applications, is to focus on optimizing the AMG smoother. Furthermore, the use of graph neural networks is mandatory for this task, as they are uniquely capable of handling the unstructured grids that define these problems.

2.4. Knowledge gaps

This approach aims to address some limitations of the scientific literature while at the same time exploring areas that have not yet been investigated. The main gaps in the scientific research that have been observed are:

- Most of the foundational work has been done on structured grids because convolutional (or other) approaches are easier to apply and test [22, 24, 25, 28–30, 46, 48, 50, 51, 53]. Also, while GNNs have enabled work on unstructured grids, few papers validate their GNN-based methods on structured grids to show they can generalize.
- To compare with existing literature, researchers usually tackle diffusion equations with jumps of diffusion coefficients or rotated Laplacian equations to test their model [23–29, 50, 51, 53, 54], which gives information about robustness from jumping coefficients or rotated anisotropies, for instance. However, a more applied focus is missing in most of them, and testing on common industry-relevant flows could provide more insight into the model's applicability to aerodynamic or hydrodynamic performance.
- Many proof-of-concept studies are limited to 2D (or not specified) to reduce computational and data generation costs [22, 26–29, 46, 48–54]. However, real-world engineering problems are almost always 3D, and the complexity of both the solver and the ML model can increase significantly in 3D.
- Many papers show noticeable reductions in iteration counts but fail to report the total time of the solver plus the network forward pass, or provide limited insights about it [22–27, 29, 30, 49–51, 53, 54]. An expensive model could negate the savings from fewer solver iterations, resulting in no effective speedup. This is a significant gap in demonstrating practical utility.

2.5. Research questions

As mentioned in the previous chapter, the main goal of this study is to develop and evaluate a GNN-based approach for tuning an AMG Jacobi smoother to accelerate the solution of the pressure-Poisson equation in CFD simulations, particularly on unstructured grids. Consequently, the following research question has been formulated to provide further insight into the acceleration of AMG in unstructured grids, evaluating the applicability of a tuned Jacobi smoother:

- To what extent can a GNN-based approach for optimizing a Jacobi AMG smoother accelerate the solution of the pressure-Poisson equation in CFD simulations compared to traditional AMG implementations with fixed smoothers?

Followed by the subquestions:

- How can GNN architectures be effectively designed and implemented to learn optimal smoothing coefficients for a tuned Jacobi smoother in an AMG solver on structured and unstructured meshes?
- What is the quantifiable impact of a GNN-tuned Jacobi smoother on key AMG performance metrics, such as reduction in the number of AMG cycles and overall solver wall-clock time, for representative structured and unstructured grid problems?

- To what degree can a single GNN model, trained on a diverse dataset of CFD problem types (e.g., Poiseuille, channel, plate flows, etc.) and mesh configurations (e.g., varying cell count, cell types, grid dimension, etc.), generalize its ability to effectively tune AMG smoothers for unseen but related CFD problems and meshes without retraining?

3

Foundation of Algebraic Multigrid

The algebraic multigrid method is part of the Multigrid methods (MG), which are a class of iterative methods for linear systems of equations aimed at solving algebraic problems arising from discretized elliptic partial differential equations [55, 56]. These methods can deliver convergence rates independent of the dimension of the linear system of equations, which in turn is associated with the size of the computational grid [57, 58]. This is, the number of iterations will be fixed even though the same problem uses a coarser or finer grid, and thus, the work per iteration is $\mathcal{O}(n)$, with n being the number of unknowns. Hence, methods such as AMG can solve larger problems on proportionally large computers (to the increase in unknowns) in essentially the same amount of time, making them a perfect solver for high-performance computing [20]. This aspect is possible due to MG methods using more information related to the problem compared to other methods, such as Krylov methods [50].

3.1. The essence of multigrid

When solving a linear system of equations iteratively, many approaches are available with the primary objective of obtaining an efficient solution to the discrete system of equations. Nonetheless, one of the main problems common iterative schemes face is reducing the low-frequency residual components. Accordingly, a small number of iterations of these iterative methods can decrease the high-frequency error components almost completely. However, the low-frequency ones will reduce only after many cycles. Figure 3.1.1 illustrates a stationary (Richardson) iteration method applied to a linear system of equations. The graph shows how these solvers reduce the large eigenvalues (high-frequency) extremely fast, while the lower ones do so at a much slower pace.

High-frequency errors can be reduced fast, even with simple relaxation-type iterative methods that are slow on convergence (such as ω-Jacobi, Gauss-Seidel, or SOR). In this respect, the multigrid approach takes advantage of the fact that on coarse grids, the small eigenmodes are mapped into high-frequency ones (see Figure 3.1.2 for a simple example). Hence, MG methods try to achieve full scalability by applying relaxation techniques at different levels. For instance, if the hierarchy of scales is a nested sequence of meshes and results from successive refinements, the corresponding algorithms are known as geometric multigrid methods [50].

The AMG methods are a response to some of the drawbacks presented by the GMG methods. Mainly, one of the problems that GMG faces is that for unstructured and complex meshes, the availability of a hierarchy of geometric grids becomes nearly impossible, limiting their applicability in practical applications with complex geometries [50]. Additionally, there are sets of problems, such as diffusion ones with large jumps in coefficients, where geometric information is not enough to solve them, and AMG is capable of achieving a solution by ignoring geometric information altogether [20]. In this regard, AMG is a generalization of the GMG, which only uses algebraic

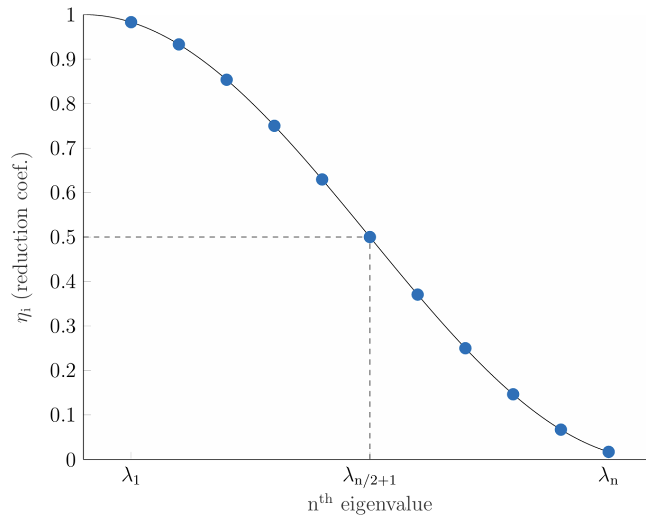


Figure 3.1.1: The error reduction coefficients for the stationary Richardson iteration applied to a linear system of equations for $n = 11$ eigenvalues. Source: [50].

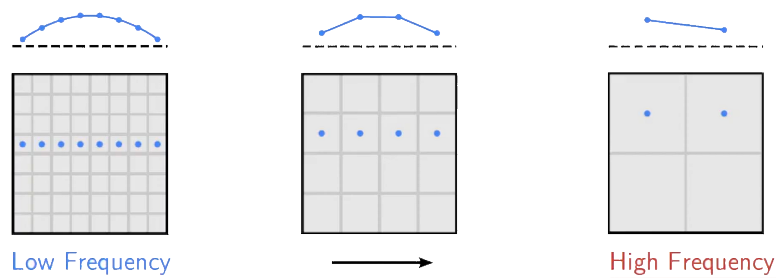


Figure 3.1.2: Simple scheme of the effect of coarsening in the frequency component of a variable. Source: [59].

information contained in the system of equations without any assumption on the underlying discretization method [60].

3.2. AMG Basics

The objective is to solve the linear system

$$\mathbf{A}\mathbf{u} = \mathbf{f}, \tag{3.2.1}$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a sparse, Symmetric positive definite (SPD) matrix arising from the discretization of the pressure Poisson equation. This is typically true in pressure projection methods for incompressible flow.

AMG constructs a hierarchy of levels based purely on the algebraic information in $\mathbf{A}$. The transfer of information between the fine level ($\mathbb{R}^n$) and the coarse level ($\mathbb{R}^{n_c}$) is defined by the prolongation operator $\mathbf{P} : \mathbb{R}^{n_c} \to \mathbb{R}^n$ and the restriction operator $\mathbf{R} : \mathbb{R}^n \to \mathbb{R}^{n_c}$. To maintain symmetry and minimize the error in the energy norm, the restriction is set as $\mathbf{R} = \mathbf{P}^T$, and the

coarse-level operator is constructed using the Galerkin approach [20]:

$$\mathbf{A}_c = \mathbf{P}^T\mathbf{A}\mathbf{P}. \tag{3.2.2}$$

A key consideration in multigrid methods is the choice of variable to transfer between grid levels. A straightforward approach would be to restrict the solution $\mathbf{u}$ to the coarse grid and solve

$$\mathbf{A}_c\mathbf{u}_c = \mathbf{f}_c.$$

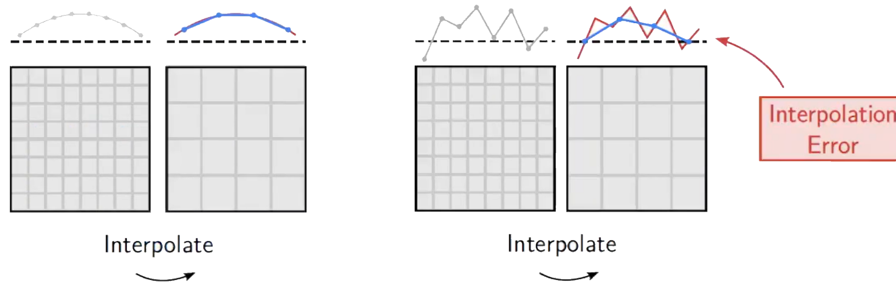


Figure 3.2.1: Simple schematic of interpolation error (aliasing) as a result of using a non-smooth variable. Source: [59].

However, this is generally ineffective because the solution itself is not necessarily smooth. As a result, restricting $\mathbf{u}$ introduces significant interpolation (aliasing) errors, as illustrated in Figure 3.2.1, where fine-scale information is lost during the transfer to the coarse grid.

An alternative is motivated by the observation that the error

$$\mathbf{e} = \mathbf{u} - \mathbf{u}^{exact}, \tag{3.2.3}$$

becomes smooth after a few relaxation steps, making it a candidate variable for coarse-grid correction. Since the exact solution is unknown, $\mathbf{e}$ is not known; the residual is used, which displays similar smoothing behavior and can be computed explicitly.

$$\mathbf{r} = \mathbf{f} - \mathbf{A}\mathbf{u}. \tag{3.2.4}$$

Both variables are related by $\mathbf{r} = \mathbf{A}\mathbf{e}$.

The process for a two-level method is presented in Algorithm 1, which aims to solve the linear system of equations (3.2.1). Note that the error is not actually being solved. When moving between levels, the **approximation of the error** is computed.

This algorithm can be easily extended to higher levels by substituting the solve step (4) in Algorithm 1 with a recursive approach, continuing until all levels have been processed and the coarsest level is solved directly. Hence, the step is substituted by $\mathbf{e}_c \leftarrow$ two_level_iteration$(\mathbf{e}_c, \mathbf{A}_c, \mathbf{f}, \nu_1, \nu_2, \mathbf{P}_c)$.

3.2.1. Algebraic smoothness

In both AMG and GMG, the error not removed by the smoother is called *smooth error*, and the coarsening and prolongation steps aim at reducing the remaining part. In geometric multigrid methods, a smooth error will also be geometrically smooth due to the nature of the coarse-grid correction [20]. Nonetheless, for AMG, splitting the fine levels into coarse ones is not done uniformly and depends on the PDE and domain where it is being solved. Hence, smooth error

Algorithm 1 One Iteration of the two-level AMG method.
$\mathbf{u}^{(k+1)} = \text{two_level_iteration}(\mathbf{u}^{(k)}, \mathbf{A}, \mathbf{f}, \nu_1, \nu_2, \mathbf{P})$

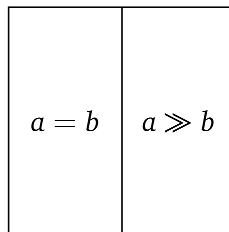
1: $\mathbf{u}_h^{(*)} \leftarrow \text{smooth}^{\nu_1}(\mathbf{A}, \mathbf{u}^{(k)}, \mathbf{f})$ ▷ Do ν_1 smoothing steps on $\mathbf{Au} = \mathbf{f}$;
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathbf{Au}^{(*)}$ ▷ Compute residual $\mathbf{r} = \mathbf{f} - \mathbf{Au} = \mathbf{Ae}$;
3: $\mathbf{r}_c \leftarrow \mathbf{P}^T \mathbf{r}$ ▷ Move residual to a coarser grid;
4: $\mathbf{e}_c \leftarrow \text{solve}(\mathbf{A}_c, \mathbf{r}_c)$ ▷ Solve the coarse system $\mathbf{A}_c \mathbf{e}_c = \mathbf{r}_c$;
5: $\mathbf{u}^{(*)} \leftarrow \mathbf{u}^{(*)} + \mathbf{Pe}_c$ ▷ Update fine level unknown with interpolated error
6: $\mathbf{u}^{(k+1)} \leftarrow \text{smooth}^{\nu_2}(\mathbf{A}, \mathbf{u}^{(*)}, \mathbf{f})$ ▷ Do ν_2 smoothing steps on $\mathbf{Au} = \mathbf{f}$;

may actually be geometrically oscillatory, and the term *algebraically smooth* is used to make a distinction between the two.

This is illustrated by the 2D diffusion problem with discontinuous coefficients [20]:

$$\begin{aligned} -a\frac{\partial^2 u}{\partial x^2} - b\frac{\partial^2 u}{\partial y^2} = f \quad & \text{on } \Omega, \\ u = g \quad & \text{on } \Gamma, \end{aligned} \tag{3.2.5}$$

where the domain is split into isotropic ($a = b$) and anisotropic ($a \gg b$) regions.



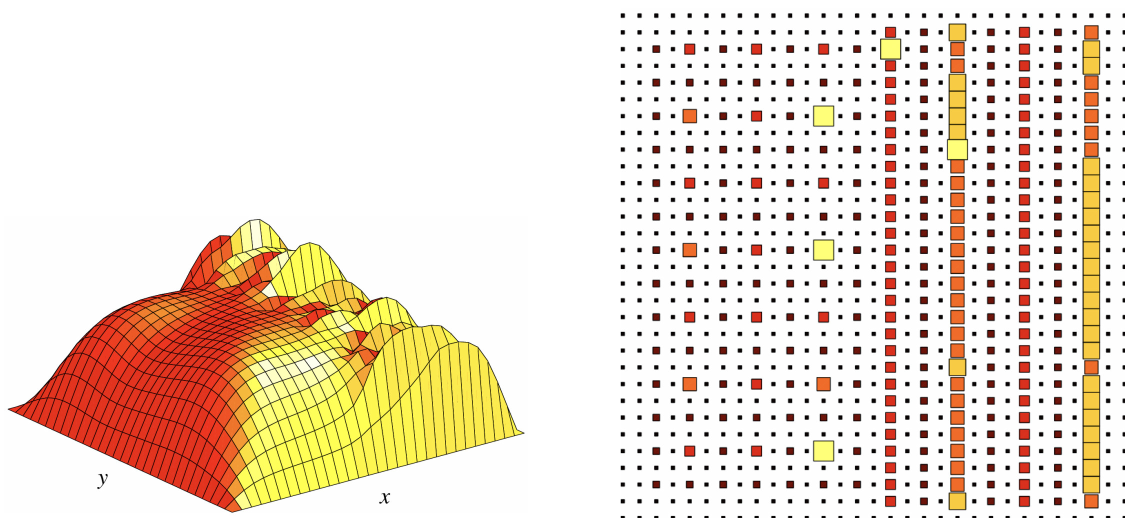
Standard relaxation (e.g., Gauss-Seidel) smooths the error in all directions on the left, but only along the x-axis (strong coupling) on the right (Figure 3.2.2a). The AMG algorithm detects this algebraic smoothness, coarsening the grid uniformly in the isotropic region while semi-coarsening (only in x) in the anisotropic region (Figure 3.2.2b), thereby effectively handling anisotropies that would stall fixed geometric grids.

3.3. AMG Components

The primary components of AMG are coarsening, interpolation, and smoothing. Additionally, the type of AMG cycle (V, W, or F) determines how these components are traversed. This section focuses on the methods most relevant to the GNN-based approach, specifically classical coarsening and direct interpolation, while briefly introducing the smoother.

3.3.1. Coarsening

The coarsening process partitions the nodes into a set of coarse (C) and fine (F) points. This splitting directly impacts the interpolation quality and the complexity of the Galerkin operator $\mathbf{A}_c$. Too dense operators may create a more accurate coarse space, but memory, computation, and setup costs will increase. Conversely, a small complexity operator can present diminished



(a) Smooth error after 7 iterations of Gauss-Seidel in linear equation system from (3.2.5).

(b) AMG levels based on Eq. (3.2.5). Larger squares correspond to coarser grids.

Figure 3.2.2: Illustration of algebraic smoothness and anisotropy effects in AMG. Source: [20]

scalability due to communication overhead, and slower convergence (or even divergence) due to poor interpolation quality.

3.3.1.1. Classic coarsening

The classic approach for coarsening is based on the fact that algebraically smooth error varies slowly in the direction of relatively large (negative) coefficients of the matrix [20]. Therefore, this gives place to one major concept of the classical algorithm: the **strength of connection**. This idea works in a way such that, given a threshold $0 < \theta < 1$, the variable u_i strongly depends on the variable u_j if

$$-a_{ij} \geq \theta \max_{k \neq i}\{-a_{ik}\}. \tag{3.3.1}$$

The strength of connection is measured relative to the largest off-diagonal entry in a row, and positive off-diagonal connections are directly *weak* connections. Usually, strength is assumed symmetric for simplicity [20].

The core of the AMG coarsening process involves evaluating the strength of connections for all non-zero coefficients. By performing a pass over each row's off-diagonal elements, a strength matrix is constructed where coefficients are replaced by the count of their strong connections (or zero if all are weak). A coefficient with a high number of strong connections indicates significant influence relative to other row entries (i.e., it is a large negative coefficient), making it an ideal candidate for coarsening.

3.3.2. Interpolation

An AMG method requires a well-constructed interpolation operator in order to achieve good convergence while keeping efficiency, avoiding high complexities on the coarse system and problems arising from parallel interpolation. In essence, the interpolation of the error approximation at an

F-point i is written as

$$e_i = \sum_{j \in C_i} w_{ij} e_j \tag{3.3.2}$$

where w_{ij} is an interpolation weight that determines the contribution of the value e_j in the final value e_i, and C_i is the subset of C-points whose values are used to interpolate a value at i [61]. Thus, the error approximation can only be interpolated to a finer level using values from coarse points.

In classical AMG, the fact that smooth error is characterized by small eigenmodes is used, from which it can be obtained that smooth error is also characterized by small residuals ($\mathbf{r} \approx \mathbf{0}$). Then, the following expression is used to calculate the interpolation value at an F-point i:

$$a_{ii} e_i + \sum_{j \in N_i} a_{ij} e_j = 0, \tag{3.3.3}$$

where N_i is the neighborhood of i, that is, the set of all points that influence i. Usually, this point neighborhood is divided, in the "classical" interpolation, into a set of C-points strongly connected to i (C_i), F-points strongly connected to i (F_i^s), and all other points weakly connected to i (F_i^w). Then, the interpolation equation can be rewritten as

$$a_{ii} e_i = -\sum_{j \in C_i} a_{ij} e_j - \sum_{j \in F_i^s} a_{ij} e_j - \sum_{j \in F_i^w} a_{ij} e_j. \tag{3.3.4}$$

Nonetheless, this interpolation formula will fail whenever the interpolation condition is not satisfied [61]. That is, strong F-F connections exist without a common C-neighbor.

3.3.2.1. Direct interpolation

When the interpolation condition is violated, there are other ways to construct the interpolation weights, although they generally lead to worse convergence rates. Accordingly, *direct interpolation* uses only immediate neighbors without the need for the interpolation condition, yielding

$$w_{ij} = -\left(\frac{\sum_{k \in N_i} a_{ik}}{\sum_{l \in C_i} a_{il}} \right) \frac{a_{ij}}{a_{ii}}, \tag{3.3.5}$$

which is applicable both sequentially and in parallel.

3.3.3. Smoothing

The smoother is a critical component of the algebraic multigrid. The classical approach of AMG was centered around a Gauss-Seidel smoother, which, while being an effective method, is inherently serial. Nevertheless, over the years, new methods have been implemented, such as Jacobi, Incomplete LU factorizations (ILU), polynomial smoothers, or multicoloring approaches, for instance [61].

The general definition of a smoother S applied to a system $\mathbf{Au} = \mathbf{f}$ is

$$\mathbf{e}_{n+1} = \mathbf{S}\mathbf{e}_n \quad \text{or} \quad \mathbf{u}_{n+1} = \mathbf{S}\mathbf{u}_n + (\mathbf{I} - \mathbf{S})\mathbf{A}^{-1}\mathbf{f}, \tag{3.3.6}$$

where $\mathbf{e}_n = \mathbf{u}_n - \mathbf{u}$ is the error. In many cases, $\mathbf{S}$ is the iteration matrix of an iterative method, defined as:

$$\mathbf{S} = \mathbf{I} - \mathbf{Q}^{-1}\mathbf{A}, \tag{3.3.7}$$

with $\mathbf{Q}$ representing part of a splitting $\mathbf{A} = \mathbf{Q} + (\mathbf{A} - \mathbf{Q})$. For example, in Gauss-Seidel, $\mathbf{Q}$ is the lower-triangular part of $\mathbf{A}$. Other smoother families take different approaches. *Polynomial smoothers* and *approximate inverse methods*, for instance, replace $\mathbf{Q}^{-1}$ with a more accessible approximation of $\mathbf{A}^{-1}$. Iterative schemes used as smoothers are often expressed in the form:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \mathbf{Q}^{-1}(\mathbf{f} - \mathbf{A}\mathbf{u}_n). \tag{3.3.8}$$

3.3.3.1. Parallel Relaxation Schemes

There are various relaxation schemes that are parallel by nature, such as the Jacobi method or the block Jacobi algorithm. In this case, $\mathbf{Q}$ is the (block) diagonal matrix constructed with the (block) diagonal elements of $\mathbf{A}$. Accordingly, they are one of the simplest methods to implement, especially when a high number of processors (CPU or GPU) are being used for scalability. However, their simplicity comes at a cost of slower convergence than other methods, such as Gauss-Seidel. These methods generally require a well-chosen relaxation parameter for good convergence (or to avoid divergence).

3.3.3.2. Approximate Inverse

One class of iteration schemes of particular interest is based on approximate inverses. In this respect, the smoother matrix $\mathbf{Q}^{-1}$ is chosen as any approximation to $\mathbf{A}^{-1}$, for example via a sparse approximate inverse. This involves two main challenges. First, the inverse of a sparse matrix is typically dense, so constructing a sparse approximation inevitably sacrifices accuracy. Second, the selection of the matrix $\mathbf{Q}^{-1}$ must satisfy certain spectral or stability criteria to guarantee convergence of the iteration. There are many ways this can be implemented.

The use of an approximate inverse in the multilevel methods context can be read in [62–64].

3.3.4. AMG cycles

The coarse-grid correction is a crucial element in the multigrid approach. Therefore, the choice of how to traverse from fine levels to coarse levels is an important one. This work primarily utilizes the V-cycle, which descends directly through the hierarchy to the coarsest level and immediately ascends back to the fine grid (see Figure 3.3.1). Due to its simplicity, the V-cycle minimizes the number of grid visits, making it the most computationally efficient choice per iteration when the coarse-grid correction is sufficiently strong.

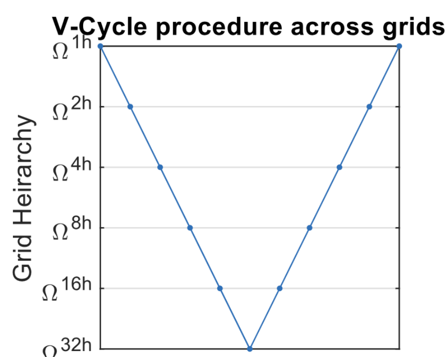


Figure 3.3.1: V-cycle on a 6-level multigrid. Source: [65].

Alternative cycling strategies exist for problems where standard correction is insufficient (e.g., high anisotropy). The W-cycle performs recursive descents at coarse levels to aggressively reduce persistent low-frequency errors, though this significantly increases computational cost. The F-cycle offers an intermediate balance between the two, utilizing nested V-cycles during the ascent phase.

4

Basics of Graph Neural Networks

Graph neural networks are a crucial element for the optimization of the algebraic multigrid method. In order to provide some background to this tool, a brief introduction to the field of deep learning is needed, along with a basic explanation of graphs. This section, then, presents a short explanation of the basics of deep learning that are key to the graph neural networks. First, an overview of the most basic neural network model, the feedforward network, is provided. It is then followed by the effects of the choice of the loss, the fundamentals of back-propagation, as well as the activation functions. Then, a review of the optimization algorithms, together with a discussion of the techniques used to improve training, such as regularization, is given.

Further sections provide an understanding of how computational graphs are built and why they are beneficial to certain problems. This is followed by the introduction of the theory behind graph neural networks, including how they work and the different ways one can operate with them.

4.1. Overview of Deep Learning

4.1.1. Neural networks

4.1.1.1. Multilayer Perceptrons

A deep forward network, or Multilayer Perceptron (MLP), approximates a function f^* by defining a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the learnable parameters. The function f is composed of K layers, where the k-th layer is defined as [50]:

$$\begin{cases} \mathbf{a}^{(k)} = \mathbf{W}^{(k)}\mathbf{x}^{(k-1)} + \mathbf{b}^{(k)} \\ \mathbf{x}^{(k)} = h^{(k)}(\mathbf{a}^{(k)}) \end{cases} \quad \forall k = 1, \dots, K$$
$$\mathbf{x} = \mathbf{x}^{(0)},\ \mathbf{y} = \mathbf{y}^{(K)},\ N_0 = N,\ N_K = M, \tag{4.1.1}$$

Here, $\mathbf{W}^{(k)} \in \mathbb{R}^{N_k \times N_{k-1}}$ and $\mathbf{b}^{(k)} \in \mathbb{R}^{N_k}$ are the parameters $\boldsymbol{\theta}^{(k)}$, with $\mathbf{W}^{(k)}$ being the *weights* and $\mathbf{b}^{(k)}$ the *biases*. Further, $h^{(k)}(\cdot)$, known as *activation function*, is a scalar non-linear function that is applied component-wise to $\mathbf{a}^{(k)}$. Without non-linear activation functions, even the most complex neural network will be unable to solve non-linear problems.

4.1.1.2. Cost function

To train the parameters $\boldsymbol{\theta}$, a *cost function* $J(\boldsymbol{\theta})$ is used to evaluate the model's performance. For a single mini-batch, the discrepancy between the prediction $f(\mathbf{x}; \boldsymbol{\theta})$ and the target $\mathbf{y}$ is measured

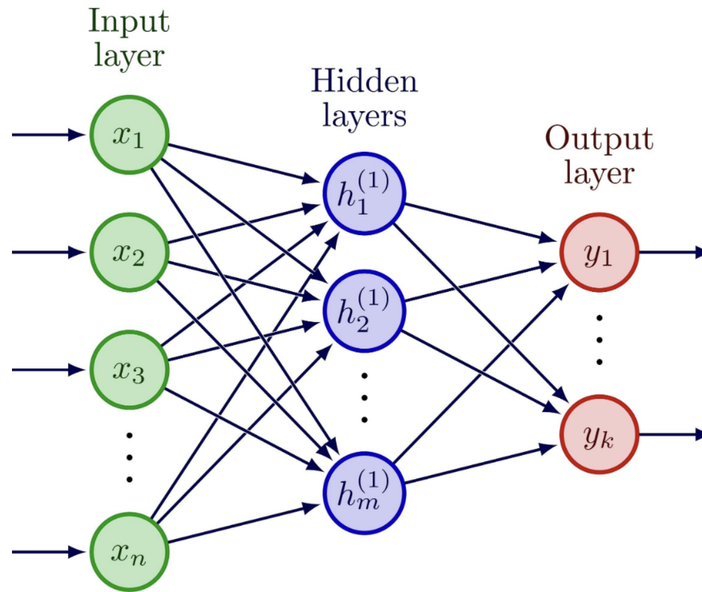


Figure 4.1.1: Schematic of a feedforward network with one hidden layer. Each node represents a hidden unit (blue), an input unit (green), or an output unit (red). The arrows indicate the direction in which the information flows.

by a loss function L. Common regression losses include the Mean squared error (MSE) ($\|\cdot\|_2^2$) and Mean absolute error (MAE) ($\|\cdot\|_1$). While MSE penalizes outliers heavily, MAE provides robustness at the cost of a discontinuous derivative [31].

The cost function generalizes this over the training set and may include regularization penalties (see Section 4.1.3):

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L\left(f\left(\mathbf{x}^{(i)};\boldsymbol{\theta}\right),\mathbf{y}^{(i)}\right). \tag{4.1.2}$$

The ultimate goal is to minimize the *generalization error*, ensuring the model performs well on unseen data rather than just memorizing the training set.

4.1.1.3. Back-propagation

Optimizing the network requires computing the gradient of the cost function, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This is achieved via *back-propagation*, which implements the chain rule of calculus. For a composite function $z = f(g(\mathbf{x})) = f(\mathbf{y})$, with $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$, the gradient is computed as:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{4.1.3}$$

Note that activation functions must also be differentiated, and, in special cases, the derivative is not continuous. Figure 4.1.2 shows a small section of a network as an illustration of the back-propagation.

Practical implementations, such as in PyTorch, construct a dynamic computational graph from the cost function back to the inputs. While robust, this graph can consume significant memory during training, a constraint discussed in Chapter 7.

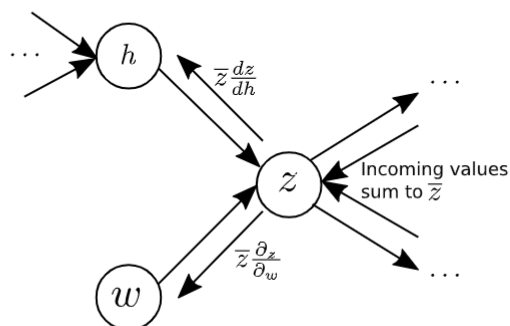


Figure 4.1.2: Schematic of a small section of a neural network with back-propagation. The bar over z indicates that the value carries the back-propagation ($\bar{z} = \partial z / \partial x_i$). Source: [66].

4.1.1.4. Activation function

The activation function provides the ability of the model to deal with more complex and non-linear problems. It is a scalar function that takes a single scalar as input, and is thus applied element-wise in the feedforward layer. The selection of an activation function is not trivial, and a proper choice can drastically improve the accuracy of the model [67]. Furthermore, activation functions are crucial for back-propagation too. If the derivative is not correctly balanced, the activation function can be biased towards large positive or negative input values, or even produce a near-zero gradient that does not practically update the network parameters.

There are several types of activation functions, such as the sigmoid, the hyperbolic tangent, and the Rectified Linear Unit (ReLU), as well as modifications to improve them. Here, the sigmoid and ReLU will be briefly introduced as they are the ones used in this work. For more information, see [31, 67–69].

First, the sigmoid is an activation function that takes a real-valued number and "squashes" it into the range (0,1), making it an excellent choice for classification tasks (see Figure 4.1.3a). It is mathematically defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{4.1.4}$$

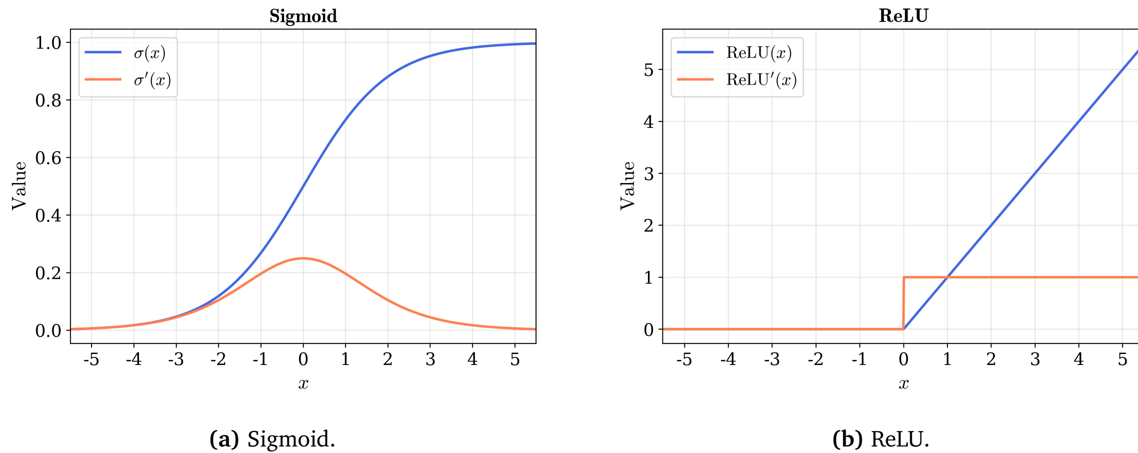
Some of the drawbacks of this activation function are related to the near-zero gradients for large positive and negative inputs, producing almost no signal for the optimizer to update the parameters of the hidden cell and thus, slowing down gradient-based learning [31]. Further, since the outputs are not zero-centered, they could lead to unstable dynamics during the gradient updates for the parameters. However, this issue is not as detrimental and can be mitigated by batch normalization [50].

The Rectified Linear Unit activation function is a pass filter only for positive numbers (see Figure 4.1.3b), as it is defined as

$$\text{ReLU}(x) = \max(0, x). \tag{4.1.5}$$

This converts it into a simple activation function, presenting both advantages and disadvantages. Some of the pros are that avoiding exponentials gives a faster evaluation of the activation function and, using a random initialization of the network parameters, gives place to only half of the hidden units having a non-zero output, called sparse activation [50], producing not so heavy computations. On the con side, it is not zero-centered as the sigmoid, and has a zero gradient for negative values. Hence, it is possible that a certain combination of weights update produces a negative input to the ReLU for all data points. With zero gradient the neuron stops learning and is

effectively "dead", also called the "dying" problem.



(a) Sigmoid. **(b)** ReLU.

Figure 4.1.3: Activation functions (blue) and their derivatives (orange).

4.1.2. Optimization algorithms

Modern neural networks feature non-convex cost landscapes characterized by many local minima and saddle points. Accordingly, gradient-based methods are used to find a sufficiently good local minimum. To showcase the non-convex nature of the cost function, two landscapes are shown in Figure 4.1.4 to illustrate this issue.

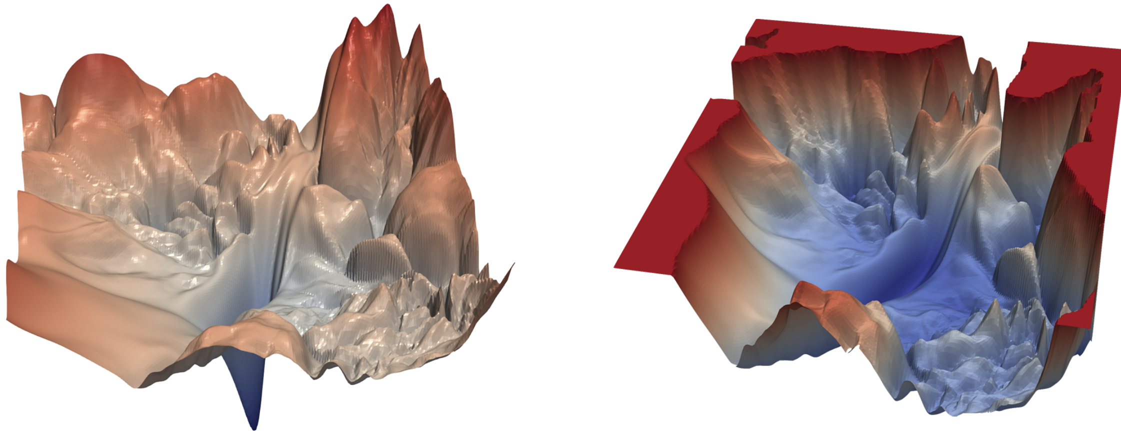


Figure 4.1.4: Non-convex landscape of the cost function of two modern neural network architectures. Source: [70].

4.1.2.1. Stochastic gradient descent

Standard gradient descent is computationally prohibitive for large datasets. Hence, stochastic gradient descent (SGD) is the common choice for the optimization step [31]. SGD approximates the gradient using a small subset of data, or *mini-batch*, of size m':

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L\left(f\left(\mathbf{x}^{(i)}; \boldsymbol{\theta}\right), \mathbf{y}^{(i)}\right). \tag{4.1.6}$$

The mini-batch size is a critical hyperparameter. Larger batches provide precise gradients but are expensive, while smaller batches introduce noise that can, in some cases, act as regularization. The step size is controlled by the *learning rate* ϵ, often adjusted dynamically by a scheduler (e.g., ReduceOnPlateau).

Advanced optimizers like Adam combine momentum (history of gradients) and RMSprop (scaling based on gradient variance) to improve convergence speed and stability [31].

4.1.2.2. Second-order methods

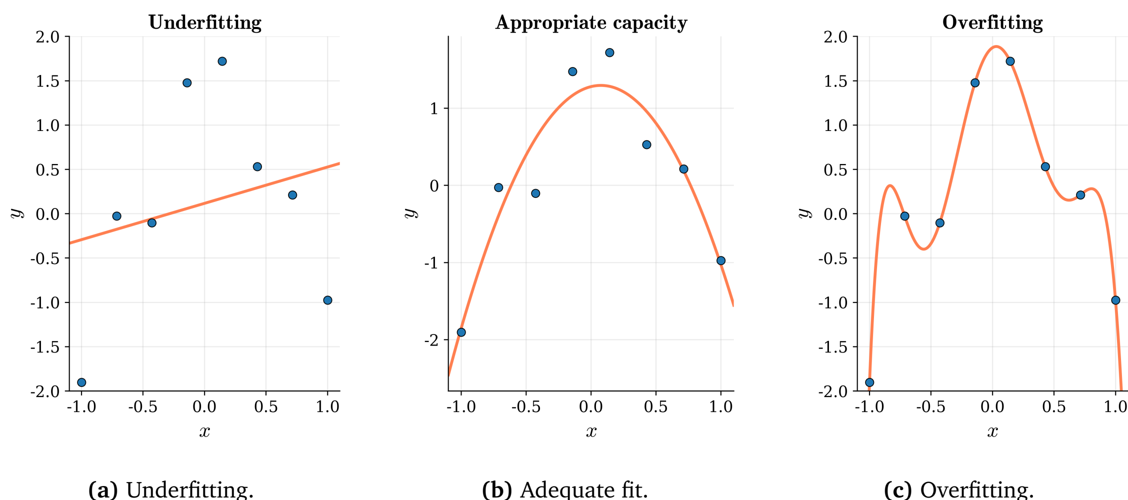
First-order methods, while faster, generally leave information behind, such as the curvature of the cost function landscape. Second-order methods use the Hessian to search for the minima, getting a better selection of parameters due to an improved local minima. Furthermore, they are computationally more expensive than the first-order methods, given that the second derivatives must be stored, and to avoid fluctuations in the Hessians, mini-batches are not used or have large sizes.

Common second-order optimizers are Newton's method or the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method [31]. A more useful implementation of the BFGS method is the Limited-BFGS (LBFGS) method, which does not use the complete approximation of the inverse Hessian matrix on each step.

For more information on second-order methods, see [31, 71, 72].

4.1.3. Regularization

Applying an optimization algorithm to the cost function can lead to a neural network that memorizes the training set, known as overfitting. This translates into a model that only knows how to work with the training data, and when given new one, the performance would be poor. This is a result of the model fitting the noise in the data as if it were part of the underlying structure. As a simple example, Figure 4.1.5 represents how a linear function is not enough to capture the structure of the data, while a quadratic function appropriately gets the data pattern without trying to capture all points, as the overfitting model, which ends up with a large generalization error. This section aims to briefly present some of the techniques used in this work to reduce overfitting and improve generalization error.



(a) Underfitting. **(b)** Adequate fit. **(c)** Overfitting.

Figure 4.1.5: Simple illustration of underfitting (a), adequate fitting (b) and overfitting (c).

4.1.3.1. Parameter norm penalization. Weight decay

This method diminishes overfitting by adding a penalty to the cost function as a new term, in addition to the loss. This regularization term can be an L_2 or L_1 regularization, that is, $\frac{\alpha}{2}\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\theta}$, or $\alpha||\boldsymbol{\theta}||_1$, respectively. Accordingly, by encouraging smaller weights during each gradient update, the model becomes less sensitive to noise in the training data [31].

Another similar technique is weight decay. In essence, this approach subtracts a fraction of the weights from their update during gradient descent rather than adding a penalty term in the cost function. An implementation is the optimizer AdamW from PyTorch, which provides better generalization than L_2 regularization.

4.1.3.2. Early stopping

Overfitting cannot be detected by monitoring the training loss alone. The idea behind early stopping is to introduce a different dataset, different from the training one, so that during training, it is constantly monitored. Ideally, the error from the new dataset should constantly decrease, although it is possible that it reaches a plateau and afterwards increases while the training error diminishes. This means that the model is overfitting. It is getting better at working with the training dataset, but worsening its performance on unseen data. Figure 4.1.6 shows how the validation error increases even though the training keeps decreasing marginally.

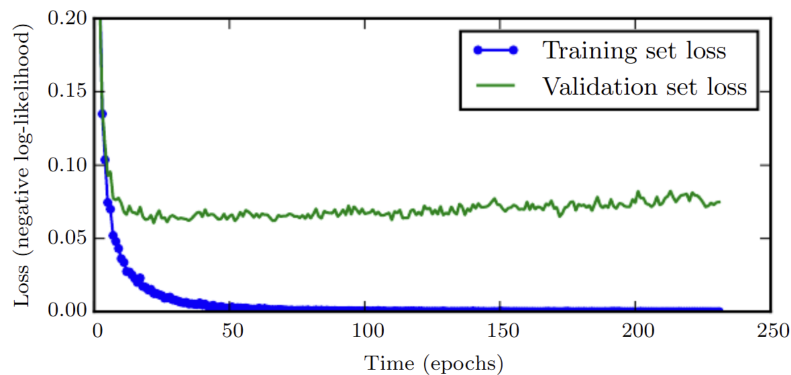


Figure 4.1.6: Illustration of training (blue) and validation (green) loss. Notice how the validation error increases after a certain number of epochs, even though the blue curve keeps decreasing. Source: [31].

4.1.3.3. Initialization

Another important aspect of an effective neural network training is the parameter initialization. The main goal is to "break symmetry", so that different hidden units can learn diverse features by starting with distinct random values. The initialization can be done by different functions, each one producing slightly better results depending on the activation functions that are used, such as Xavier or Kaiming distributions. Eventually, the initialization should generate small weights so that instabilities are avoided due to large gradients [31].

4.1.3.4. Batch normalization

Batch normalization is a technique aimed at accelerating the convergence during training of deep neural models. The motivation behind this regularization method is based on the fact that updating a parameter does not depend only on its layer, but also on the other layers of the network [73]. To this end, this technique normalizes the inputs to a hidden layer to have a mean of zero

and a standard deviation of one for each training mini-batch. Then it scales and shifts them with two learnable parameters (γ, β) so the network can still represent any transformation, i.e., it does not lose expressive power as it can learn how to undo it. The input is scaled as

$$\begin{aligned} \mathbf{z}_i &= \frac{\mathbf{x}_i - \mu_{m'}}{\sqrt{\sigma^2_{m'} + \epsilon}}, \\ \hat{\mathbf{z}}_i &= \gamma \mathbf{z}_i + \beta, \end{aligned} \tag{4.1.7}$$

where $\hat{z}_i$ is the normalized sample of the mini-batch ($i = 1, \ldots, m'$). Further, μ_B, σ^2_B are the mini-batch mean and variance, and ϵ is a small positive value for numerical stability.

4.1.3.5. Dropout

Dropout is an effective regularization technique that consists of temporarily removing a random fraction of the hidden units (along with their connections) from the network during each training step. The main advantage of this technique is that by removing a certain number of hidden units, the size of the network is effectively reduced, making it harder to adapt to the noise of the training dataset, learning more robust features. However, sometimes this requires increasing the size of the network to compensate for the reduced capacity during the dropout.

4.2. Introduction to Graph Neural Networks

4.2.1. Definition of graph

Within the graph neural network framework that is used in this work (based on [32]), a graph is defined as a 3-tuple $G = (\mathbf{u}, V, E)$. Here $\mathbf{u}$ is a global attribute vector (e.g., number of nodes in the graph, etc.). Further, $V = \{\mathbf{v}_i\}_{i=1:N^v}$ is the set of nodes (with N^v nodes), where each $\mathbf{v}_i$ is a node's attribute (for instance, the value of the matrix A_{ii}). Finally, $E = \{(\mathbf{e}_k, r_k, s_k)\}_{k=1:N^e}$ is the set of edges (of cardinality N^e), where each $\mathbf{e_k}$ is the edge's attribute, r_k is the index of the receiver node, and s_k is the index of the sender node (e.g., value of matrix A_{ij}).

An example of a graph structure from this study is illustrated in Figure 4.2.1. There, a directed (one-way edges), attributed (edges and nodes have attributes), with a global attribute graph is presented.

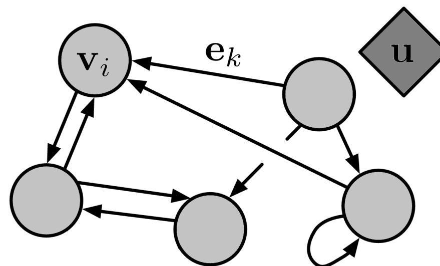


Figure 4.2.1: Graph representation based on this work. Adapted from: [32].

4.2.2. Graph network block

This section presents a brief introduction to the basic Graph Network (GN) block structure and direct applications.

4.2.2.1. Internal structure of a GN block

A complete GN block contains three "update" functions, ϕ, and three "aggregation" functions, ρ, such that updated edges, nodes and global attributes are determined by

$$\begin{aligned} \mathbf{e}'_k &= \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}), & \overline{\mathbf{e}'_i} &= \rho^{e\to v}(E'_i), \\ \mathbf{v}'_i &= \phi^v(\overline{\mathbf{e}'_i}, \mathbf{v}_i, \mathbf{u}), & \bar{\mathbf{e}}' &= \rho^{e\to u}(E'), \\ \mathbf{u}' &= \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}), & \bar{\mathbf{v}}' &= \rho^{v\to u}(V'), \end{aligned} \tag{4.2.1}$$

where E' and V' are the updated set of edges and nodes, and E'_i is the set of updated edges going to node i ($r_k = i$).

Note that ϕ are mappings, which means they can be functions such as a multilayer perceptron, or a recurrent neural network, for example. As an important note, the aggregation functions ρ must be invariant to permutations of their inputs, and should allow a variable number of arguments (for instance, element-wise summation, mean, maximum, etc.) [32].

4.2.2.2. Computational steps of a GN block

To keep the explanation simple yet informative, the following Algorithm 2 from [32] is presented, where the full GN block computational steps are presented. For further details on the process, the reader is encouraged to see the reference.

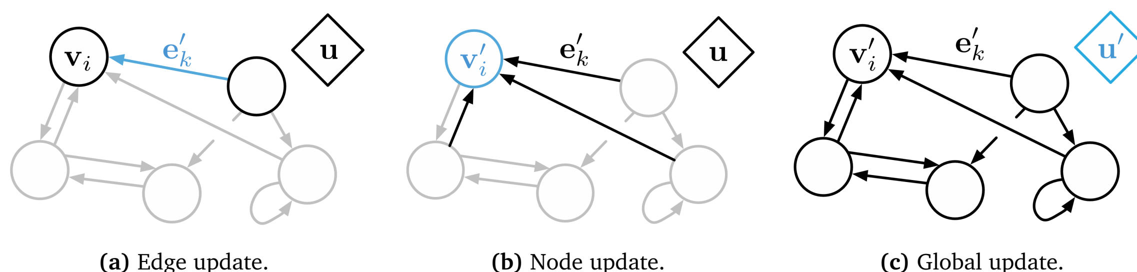
From the algorithm, the basic full block for a graph network consists in updating the edges' attributes using a mapping ϕ^e taking the edge values, the nodes connected by the edges, and the global attributes (1). Next, for each node, all edges directed to it are aggregated into a single edge attribute (2), which is later used to update the node attributes, together with the older node and global attributes using a mapping ϕ^v (3). It is as if a message was being passed from the edges to the nodes. Further computations involve aggregating all the updated edge attributes into a single global edge attribute (4). This is done identically for the updated nodes' attributes (5). Finally, the global edge and nodes attributes, as well as the global attributes, are mapped ϕ^u to an updated global attribute (6).

Algorithm 2 Steps of computation in a full GN block. Source: [32]

1: **function** GRAPHNETWORK($E, V, \mathbf{u}$)
2:     **for** $k \in \{1 \ldots N^e\}$ **do**
3:         $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$ ▷ 1. Compute updated edge attributes
4:     **end for**
5:     **for** $i \in \{1 \ldots N^n\}$ **do**
6:         **let** $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$
7:         $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\to v}(E'_i)$ ▷ 2. Aggregate edge attributes per node
8:         $\mathbf{v}'_i \leftarrow \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$ ▷ 3. Compute updated node attributes
9:     **end for**

10:     **let** $V' = \{\mathbf{v}'_i\}_{i=1:N^v}$
11:     **let** $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$
12:     $\bar{\mathbf{e}}' \leftarrow \rho^{e\to u}(E')$ ▷ 4. Aggregate edge attributes globally
13:     $\bar{\mathbf{v}}' \leftarrow \rho^{v\to u}(V')$ ▷ 5. Aggregate node attributes globally
14:     $\mathbf{u}' \leftarrow \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$ ▷ 6. Compute updated global attribute
15:     **return** $(E', V', \mathbf{u}')$
16: **end function**

To provide some illustration of the process, Figure 4.2.2 presents the updates (1), (3), and (6), respectively. Gray colored nodes and edges are not used during the update.



(a) Edge update. **(b)** Node update. **(c)** Global update.

Figure 4.2.2: Updates within a GN block. Blue highlights the element being updated, while black elements are the inputs to the update function. The pre-update value of the blue element is also used as an input. Source: [32]

4.2.3. Designing graph network architectures

The full graph network block that has been presented recently is not strictly used equally in all graph network architectures. The idea behind graph networks is to support highly flexible graph representations in terms of the representation of the attributes and the structure of the graph. Therefore, the GN block will be constructed to satisfy the demands of the task, which also involves tailoring the input and the output.

4.2.3.1. Flexible representations

Attributes

Regarding the attributes, the edge, node, and global attributes of a GN block can use arbitrary representational formats. For instance, real-valued vectors or tensors could be used. But more complex structures, such as sets or graph's can also be used.

The requirements of the problem will generally determine what will be used as attribute representations (e.g., tensors for image patches). Accordingly, a graph neural network can be designed to work in conjunction with other deep learning building blocks such as MLPs, CNNs, and RNNs. Hence, the edge and node outputs of a GN block could correspond to a list of vectors or tensors that will be passed to the other DL architectures afterward. In addition, the output of a GN block can also be customized to the problem specifics [32]:

- An *edge-focused* GN uses the edges as output, for instance, to make a prediction about the relationship or interaction between nodes (e.g., chemical bond).
- An *node-focused* GN uses the nodes as output, for example, to reason about a specific entity based on its properties and its neighborhood (e.g., velocity of a planet in a solar system).
- A *graph-focused* GN uses the globals as output, for example, to make a single prediction about the entire system as a whole (e.g., classifying an image).

Note, however, that more than one of the attributes can be used in subsequent deep learning blocks, and a mix of the above types of GN blocks can exist.

Graph structure

With respect to the graph structure itself, it can be either explicitly defined by the input data or must be inferred or assumed from it. In this study, the relational structure is clearly specified, such as in molecules, social networks, or road maps.

4.2.3.2. Configurable within-block structure

A key feature of the GN block is its flexibility, allowing the internal functions and their inputs to be configured in various ways to control how edge, node, and global attributes are updated. Accordingly, each mapping ϕ from Equation 4.2.1 must be implemented with some function, f, which should adapt to the specifics of the problem. In this regard, each of the mappings ϕ can be defined as neural networks, and the nature/architecture of the network used will depend on the actual focus of the GNN. For instance, one possibility is to use RNNs, which require an additional hidden state, i.e., a graph representing the hidden state, as input and output. Additionally, MLPs are the common choice when dealing with vector attributes, whereas tensors and image feature maps are more suited for CNNs. Denoting the mappings ϕ^* by NN_*, and using element-wise sumation for the aggregation ρ implementations, the following set of equations for the GN block structure can be obtained:

$$\begin{aligned}
\phi^e(\mathbf{e}_k,\mathbf{v}_{r_k},\mathbf{v}_{s_k},\mathbf{u}) &:= f^e(\mathbf{e}_k,\mathbf{v}_{r_k},\mathbf{v}_{s_k},\mathbf{u}) &&= \mathrm{NN}_e([\mathbf{e}_k,\mathbf{v}_{r_k},\mathbf{v}_{s_k},\mathbf{u}]),\\
\phi^v(\bar{\mathbf{e}}'_i,\mathbf{v}_i,\mathbf{u}) &:= f^v(\bar{\mathbf{e}}'_i,\mathbf{v}_i,\mathbf{u}) &&= \mathrm{NN}_v([\bar{\mathbf{e}}'_i,\mathbf{v}_i,\mathbf{u}]),\\
\phi^u(\bar{\mathbf{e}}',\bar{\mathbf{v}}',\mathbf{u}) &:= f^u(\bar{\mathbf{e}}',\bar{\mathbf{v}}',\mathbf{u}) &&= \mathrm{NN}_u([\bar{\mathbf{e}}',\bar{\mathbf{v}}',\mathbf{u}]),\\
\rho^{e\to v}(E'_i) &:= &&= \sum_{\{k:r_k=i\}} \mathbf{e}'_k,\\
\rho^{v\to u}(V') &:= &&= \sum_i \mathbf{v}'_i,\\
\rho^{e\to u}(E') &:= &&= \sum_k \mathbf{e}'_k.
\end{aligned} \tag{4.2.2}$$

For more information about different internal GN block configurations that are used in the literature, see [32].

4.2.3.3. Composable multi-block architectures

A central design principle of graph networks is their modularity. Complex models can be constructed by composing multiple GN blocks. Thus, each block takes a graph as input and outputs another graph with the same structural elements (nodes, edges, and global attributes). This graph-to-graph interface allows sequential composition, similar to how layers are stacked in conventional deep learning models [32].

It is possible to combine two or more GN blocks either with unshared parameters, analogous to separate layers in a CNN, or with shared parameters, as in recurrent or message-passing architectures. When weights are shared across repeated GN applications, information is propagated iteratively through the graph, enabling nodes to aggregate data from increasingly distant neighbors after each step [32].

4.2.4. Graph convolutional network

Graph convolutional networks (GCNs) are an extension of the convolution concept used in regular Euclidean domains, such as those used in CNNs. Accordingly, instead of aggregating spatially structured information after the application of filters in local neighborhoods, a GNC defines a

message passing mechanism in which each node updates its attributes through the aggregation of features from its neighbors. This is due to the locality being defined by node connectivity rather than spatial adjacency, which makes classical convolutions unfeasible.

The computational block of a GCN, in a node-wise formulation (where $\mathbf{v}_i$ is now referred to as $\mathbf{x}_{v_i}$), is defined as

$$\mathbf{x}_{v_i}^{(k)} = \mathrm{MLP}^{(k)}\left(\sum_{v_j \in N(v_i)\cup v_i} \frac{w_{ij}}{\sqrt{\hat{d}_i \hat{d}_j}} \mathbf{x}_{v_j}^{(k-1)}\right), \tag{4.2.3}$$

where w_{ij} is the weight if the edge e_{ij}, if the graph is unweighted, then $w_{ij} = 1$. $\hat{\mathbf{D}}$ is the diagonal degree matrix and $\hat{d}_i$ is the degree of node v_i (number of connections) in the graph. Further, $N(v_i)$ is the set of neighbors from node v_i, so the summation considers all direct connections as well as the node itself, as a usual filter convolution. Note that this is a node-focused GNN, where edge and global attributes are not updated. Edges are only used for message-passing in the node update. Moreover, written in matrix form, the GNC forward pass can be read as

$$\mathbf{x}_v^{(k)} = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{1/2}\mathbf{x}_v^{(k-1)}\mathbf{\Theta}^{(k)}, \tag{4.2.4}$$

where $\mathbf{\Theta}^{(k)}$ is the weight matrix for the $\mathrm{MLP}^{(k)}$ that are optimized during training and $\hat{\mathbf{A}}$ is the adjacency matrix of the graph with added self-loops, i.e., $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$, so each node is connected to itself. Notice that this GCN block can be related to the full GN block, in which the mappings and aggregations for the edges and global attributes are not used. Now ϕ^v is the feature vector of neighbor nodes, and $\rho^{e\to v}$ is a weighted average [24, 74, 75].

4.2.5. Graph isomorphism network

Graph isomorphism networks (GINs) are a type of GNNs that were designed to be able to differentiate different graph structures. The idea behind them is to maintain the simple message-passing form while maintaining the same ability as the Weisfeiler-Lehman (1-WL) graph isomorphism test, which is a classical algorithm usted to check if two graphs are different [76]. By being able to differentiate between different graph structures, the model can recognize patterns of connectivity that, although similar, are topologically distinct. In essence, differentiating graph structures allows the network to learn operators that are sensitive to the actual problem geometry [77].

In a graph isomorphism network, a node-focused GN is used, where the feature update x_{v_i} for a node at a layer k is given by

$$\mathbf{x}_{v_i}^{(k)} = \mathrm{MLP}^{(k)}\left(w_{ii}(1+\epsilon^{(k)})\cdot \mathbf{x}_{v_i}^{(k-1)} + \sum_{v_j \in N(v_i)} w_{ij}\mathbf{x}_{v_j}^{(k-1)}\right), \tag{4.2.5}$$

where w_{ii} is the weight of the node's self-loop, w_{ij} is the weight of edge e_{ij}, and $\epsilon^{(k)}$ can be a trainable parameter or a fixed constant number. The matrix form of one GIN layer can be written as

$$\mathbf{x}_v^{(k)} = \left[\mathbf{A} + (1+\epsilon^{(k)})\cdot \mathbf{I}\right]\cdot \mathbf{x}_v^{(k-1)}\mathbf{\Theta}^{(k)} \tag{4.2.6}$$

Notice that, compared to a GCN, a GIN uses a summation instead of a weighted average. This is in accordance with the authors of GIN, who demonstrated that in specific scenarios, MEAN and MAX functions can impair the expressiveness of the GNN. Hence, aside from the aggregation operator, *readout* functions in a GIN should be summations [24, 77]. The readout function on a GNN is an operation on the edges and/or nodes used to collapse all the graph data into a single scalar or

vector for node and edge attributes. For instance, the recommended readout function for a GIN is

$$\begin{aligned} \mathbf{x}_g &= \mathrm{CONCAT}\left(\mathrm{SUM}\left(\mathbf{x}_v^{(k)}\right) \,\middle|\, k = 0, 1, \ldots, K \right), \\ \mathrm{SUM}\left(\mathbf{x}_v^{(k)}\right) &= \sum_{i=1}^{|V|} \mathbf{x}_{v_i}^{(k)}. \end{aligned} \tag{4.2.7}$$

where CONCAT concatenates the vectors resulting from each layer SUM into a long vector.

5

Methodology

The process of designing, training, and assessing a complex deep neural network involves a well-optimized workflow and a robust implementation of the features within the framework. This chapter aims to present an overview of the methodology, from generating/acquiring the data and processing it, to how the architecture has been designed and implemented within the context of an AMG solver. The computational resources and software utilized during the development are presented in Appendix A.1.

5.1. Data generation and processing

Training this AMG-GNN solver requires matrices and vectors associated with a Poisson equation of the kind $\mathbf{Au} = \mathbf{f}$. This approach uses matrices of coefficients $\mathbf{A}$, the RHS vector $\mathbf{f}$, and an initial vector guess $\mathbf{u}_0$ (to generate the initial residual $\mathbf{r}_0$) as inputs, after they have been converted to graphs, for the training and evaluation loop.

An AMG solver can handle both structured and unstructured grids. Therefore, it is necessary to implement data generation algorithms to create Poisson equations for both types of problems. Additionally, problems can be of a synthetic nature, that is, they do not have to be representative of a practical problem, to test how the model performs in such cases. Accordingly, the data acquisition algorithm is divided into two main branches. The first one is centered around Julia 1.8 (programming language) and WaterLily.jl for structured data generation (Section 5.1.1), and the second one uses ReFRESCO and a custom pipeline to generate unstructured two- and three-dimensional problems (Section 5.1.2).

Data that has been generated can be used in both the model training and validation scripts, and the model accepts heterogeneous problems. If desired, one can train 2D/3D structured and unstructured data simultaneously and evaluate the model afterward. A wide variety of problems is generated, and thus, modularity is a key aspect to take into consideration when designing an optimized workflow. In this respect, these are the guidelines followed to implement the scripts to read data:

- Different types of cases are generated within each CFD solver. The pipeline should be able to select which kind of problems to load, given the user input.
- The matrices and vectors created come from two- and three-dimensional problems. Although the matrices and vectors are 2D, the inner structure of the matrix of coefficients can change completely with respect to the spatial dimension. The script should allow for mixing or selecting a specific dimension.
- Training and validating a deep network model is generally limited by memory constraints. Further, a well-trained model requires a varied dataset to avoid overfitting and allow the

model to have more examples to learn from. Hence, problems are generated for a wide range of grid sizes.

- Sometimes it is necessary to repeat an experiment with a slightly different setup, to determine the effect on performance of a specific change. This implies using the same data from the original experiment. Nonetheless, investigating how the model performs with new data is also essential. Therefore, a shuffle has been implemented with the option to use a seed for the randomizer, so that the mixing is deterministic when needed.

5.1.1. WaterLily

WaterLily is a simple and fast fluid simulator written in pure Julia, aiming to accelerate and enhance fluid simulations [78]. It is used to generate structured data of synthetic and unsteady problems based on the previous work on the Tuned Jacobi smoother (for GMG) in structured grids [30]. In this data generation algorithm are three synthetic and three unsteady cases.

5.1.1.1. Synthetic cases

The synthetic cases are generated with simple functions on uniform 2D or 3D Cartesian grids using n grid cells in each direction. For training $n = 16, 32$ were used, and for validating, $n = 64, 128$. There are *static*, *dipole* and *sphere* 2D and 3D cases, which are explained in the next lines following [30].

Static cases (Figure 5.1.1a) are generated from a constant gradient solution

$$x_{0,i} = \hat{\mathbf{m}} \cdot \mathbf{q}_i, \tag{5.1.1}$$

where $\mathbf{q}_i$ is the vector to the center of the grid-cell i and $\hat{\mathbf{m}}$ is a random unit vector. The matrix $\mathbf{A}$ is defined by $a_{ij} = 1$ on cell faces, $a_{ij} = 0$ on domain faces, and $a_{ii} = -\sum_j a_{ij}$. Therefore, the initial residual is then $\mathbf{r}_0 = \mathbf{A}\mathbf{u}_0$.

The dipole problems (Figure 5.1.1b) are generated using a localized residual function

$$r_{0,i} = \hat{\mathbf{m}} \cdot (\mathbf{q}_i - \mathbf{Q}) \exp\left(-\frac{|\mathbf{q}_i - \mathbf{Q}|^2}{\sigma^2}\right), \tag{5.1.2}$$

where $\mathbf{Q}$ is a random location in the fluid domain and σ is a random width. The same uniform $\mathbf{A}$ matrix from the static case is used.

Regarding the synthetic sphere cases (Figure 5.1.1c), these are generated using the residual from an immersed sphere with random center $\mathbf{Q}$ and radius s, moving in a random direction $\hat{\mathbf{m}}$ in an initially still fluid. The coefficient matrix $\mathbf{A}$ is obtained by applying the equation arising from the Boundary Data Immersion Method [79, 80]:

$$\oint_{\partial\Omega_i} \beta \left(\nabla\phi \cdot \hat{\mathbf{n}}\right)\, ds = \oint_{\partial\Omega_i} (1-\beta)\left(\hat{\mathbf{m}} \cdot \hat{\mathbf{n}}\right)\, ds, \tag{5.1.3}$$

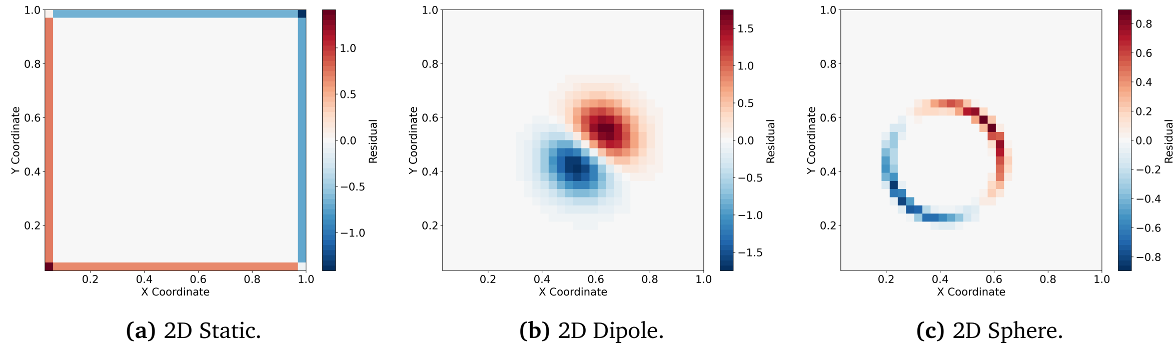
where $\partial\Omega_i$ are the faces of the grid cell i, $\hat{\mathbf{n}}$ is the face normal, ϕ is the scaled pressure, and β is a coefficient which smoothly transitions from 1 to 0 as a function of the signed distance d from the grid cell face to the immersed boundary. For instance, $\beta = \min\left[1, \max\left(0, d + \frac{1}{2}\right)\right]$ with $d = |\mathbf{q} - \mathbf{Q}| - s$ is an option.

To generate the synthetic data ($\mathbf{A}$, $\mathbf{f}$, $\mathbf{u}_0$) for each of the cases, a separate script in the WaterLily library has been created. The main process consists in:

1) Selecting the number of samples to be saved (e.g., 150) for training and validation.

2) Choosing problem types (static, dipole, sphere), sizes (16, ..., 128), and dimensions (2,3).
3) Using the building functions for each of the synthetic cases, each problem type is called $N_{samples}$ times (the function has a randomizer implemented) to generate the matrix and vectors, and the fields are extracted in sparse format.

Note that for the static and dipole problems, the matrix **A** obtained is constant and equal. Whereas for the sphere cases, a different coefficient matrix is obtained for each sample. Having small to no variation on the training dataset can lead to overfitting, so having the sphere problems alleviates this issue.



(a) 2D Static. **(b)** 2D Dipole. **(c)** 2D Sphere.

Figure 5.1.1: Initial residual fields for each of the synthetic cases ($n = 32$).

5.1.1.2. Unsteady simulations

With respect to the unsteady cases, these are generated using general flow scenarios, and the specifications of each problem are as follows:

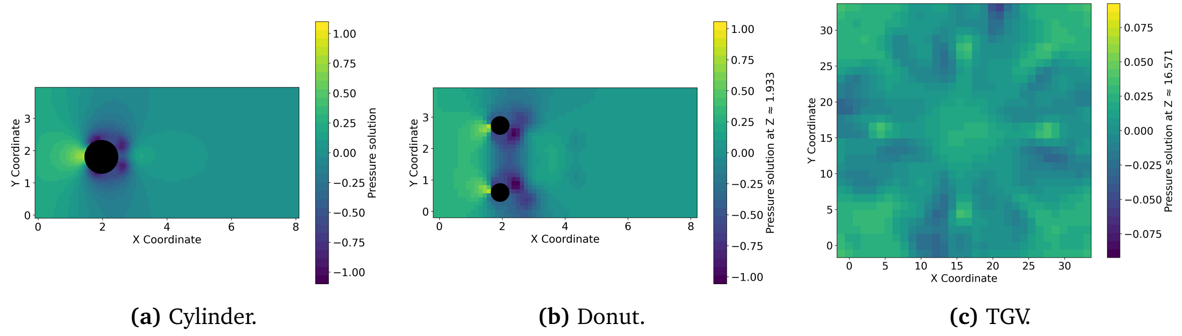
(a) **2D flow past a static circular cylinder**. The radius of the cylinder is $r = 2^{p-2}$ grid cells, with a domain size of $16r$ by $8r$ cells and p being a refinement parameter. Further, the cylinder center is located at $(4r, 4r)$, and the Reynolds number is $\mathrm{Re} = Ur/\nu = 250$.

(b) **3D flow past a static donut**. For this case, the major radius of the donut is defined as $R = 2^{p-2}$ cells, with p being the refinement parameter. The small radius is given by $r = R/4$ cells and the domain size is $8R \times 4R \times 4R$. The center of the donut is located at $(2R, 2R, 2R)$ and $\mathrm{Re} = RU/\nu = 10^3$.

(c) **3D Taylor-Green vortex (TGV) turbulent decay**. The initial velocity function is given by $u = -\sin(kx)\cos(ky)\cos(kz)$, $v = \cos(kx)\sin(ky)\cos(kz)$, $w = 0$ with $k = \pi/n$, where $n = 2^p$. The domain size is n^3, and $\mathrm{Re} = 10^5$.

Regarding the generation of unsteady data, it involves a more convoluted process than the synthetic one, as the WaterLily solver is needed to extract the samples. The process followed to create the matrices and vectors is:

1) Define the number of samples saved for each configuration.
2) Choose a refinement for the problem p (3,4, ...).
3) Select the runtime before any sample is stored t_{sample} and the sampling rate Δt_{sample} (simulation runtime between samples).
4) Run the WaterLily simulation until $t = t_{0,sampling}$, then save **A**, **f** and $\mathbf{u}_0$ when $\Delta t = \Delta t_{sample}$.



(a) Cylinder. **(b)** Donut. **(c)** TGV.

Figure 5.1.2: Pressure fields for each of the unsteady cases with $p = 5$.

The sampling starts at a certain time to follow the methodology from [30], but also, it is better to have the flow developed first for the training dataset (specifically, better behaved **f**).

For generating the data, there is one aspect that has to be taken into consideration. The training dataset is $1/N^{\text{th}}$-scale of the validation dataset, meaning that each problem will have $1/N^M$ times fewer spatial points, with M being the dimension number. To keep consistency, the temporal data is also reduced by $1/N$. Therefore, if the full simulation generated data (validation) for $t_{sampling} = N_{samples} \Delta t_{sample}$, the scaled dataset will only generate data up to

$$t_{sampling}^{1/N} = \frac{N_{samples}}{N} \Delta t_{sample} + t_{0,sampling}, \tag{5.1.4}$$

which is an effective reduction of the time step. Hence, it is direct that having p_{val} for a certain problem type, the training dataset will be generated using $N = 2^{p_{val} - p}$. Following this logic, the main generation setup used for the unsteady problems is presented in Table 5.1.1.

Table 5.1.1: Main setup for the unsteady data generation problems.

Problem	p_{val}	p_{train}	$t_{0,\text{sampling}}$	Δt_{sample}
Circle	7	(3, 4)	2.0	$8.0/N_{\text{samples}}$
Donut	5	(3, 4)	2.0	$8.0/N_{\text{samples}}$
TGV	6	(3, 4)	4.0	$16.0/N_{\text{samples}}$

As a final note, it should be noted that **A** matrices are constant since the WaterLily simulation is at constant parameters and without moving geometries. Unless the training data is mixed, this could lead to overfitting.

5.1.2. ReFRESCO

To generate unstructured data, the ReFRESCO software is used. ReFRESCO is a multi-phase viscous flow solver focused on maritime applications under active development by the Maritime Research Institute Netherlands (MARIN) experts [21]. The main problems that comprise the unstructured dataset are Poiseuille, channel, Convection-Diffusion (ConvDiff), and plate flows. They are common CFD problems that allow for evaluating the GNN model outside synthetic benchmark problems for AMG (e.g., diffusion equation with heterogeneous coefficients), and show performance towards a more applicable end. Additionally, to test the model on an industry-relevant case scenario, the geometry and setup from AirfRANS (High Fidelity Computational Fluid Dynamics Dataset for Approximating Reynolds-Averaged Navier-Stokes Solutions) is used, which

consists of two-dimensional airfoils at a subsonic Mach number regime and for different angles of attack [81].

5.1.2.1. CFD setup

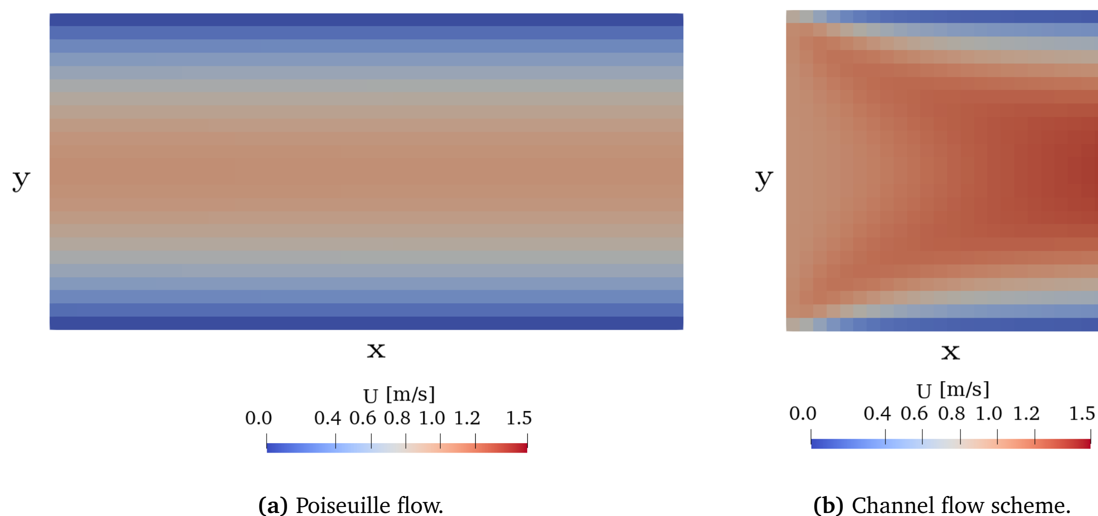
The CFD setup in ReFRESCO for the four main problem types is presented in this section. They are simulated using a SIMPLE algorithm without turbulence modeling, and thus, are laminar. For simplicity, only the 2D setup is discussed (3D is identical but with cells in the z-direction).

Poiseuille flow is a steady, fully developed, laminar flow driven by a pressure gradient, where a parabolic velocity profile is generated by the viscous forces (see Figure 5.1.3a). Hence, the setup should be in accordance with the laminarity of the flow, and is presented in Table 5.1.2.

Channel flow is similar to Poiseuille flow, with subtle differences. Generally, it refers to a

- Laminar or turbulent,
- Pressure-driven, body-force-driven, or even periodic, and
- Steady or unsteady

In this case, the channel flow that is being used is laminar (see Figure 5.1.3b), with the simulation setup from Table 5.1.2. The difference between the Poiseuille and channel flows in the dataset is that Poiseuille is simulated using a parabolic velocity distribution at the inlet ($u_{max} = 1$). In contrast, Channel starts with a uniform velocity $u = 1$ and develops towards a parabolic shape, as both have a fixed BC pressure at the outlet. Note, however, that Poiseuille flow has a fixed user input $p_{out} = 1$. See Table 5.1.3 for more information about the boundary conditions.



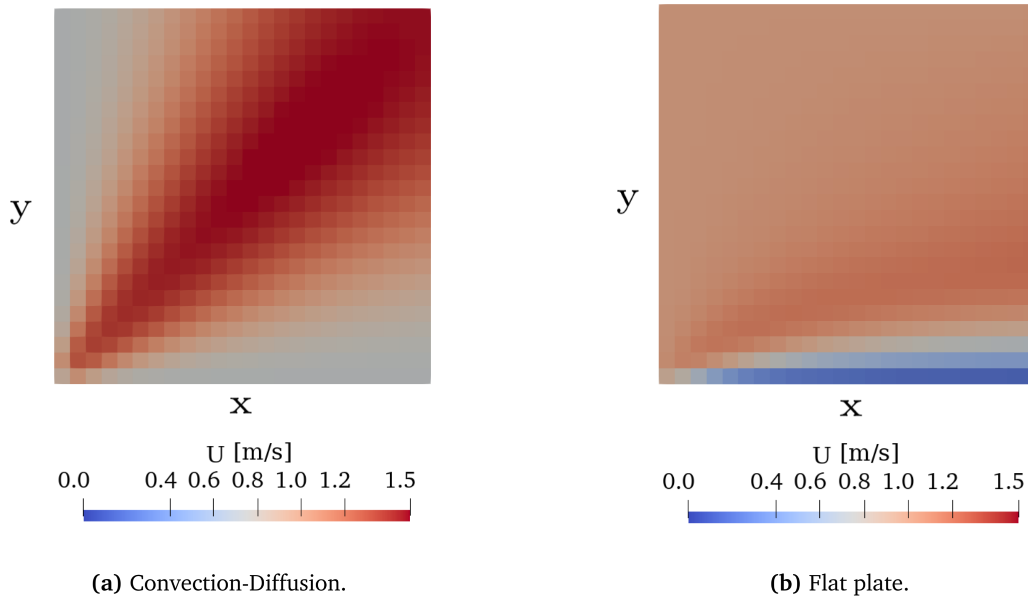
(a) Poiseuille flow. **(b)** Channel flow scheme.

Figure 5.1.3: Illustrative schemes of Poiseuille and turbulent channel flows.

The convection-diffusion (ConvDiff) flow involves the motion of fluid due to velocity and pressure with two inlets and two outlets. In this case, the following Table 5.1.2 presents the CFD setup that has been used to generate the dataset. A two-dimensional scheme is presented in Figure 5.1.4a.

Finally, the plate flow consists in an open domain where the flow interacts with a flat plate parallel to the flow direction. It is interesting as it is a fundamental case in fluid mechanics and

viscous flows, as it shows boundary layer development, shear stress, and transition to turbulence (Figure 5.1.4b). The configuration used in ReFRESCO is shown in Table 5.1.2.



(a) Convection-Diffusion.

(b) Flat plate.

Figure 5.1.4: Illustrative schemes of convection-diffusion and flat plate flows.

Table 5.1.2: CFD setup per case. Density is $\rho = 1$.

Case	Velocity	ν	Reynolds
Poiseuille	$u_{max} = 1$	10	$Re = \frac{H}{10}$
Channel	$u = 1$	2×10^{-2}	$Re = 50H$
Plate (flat plate)	$u = 1$	10^{-2}	$Re = 100x$
ConvDiff	$u = 0.705$, $v = 0.705$	10	$Re = \frac{H}{10}$

5.1.2.2. Dataset generation

The data generation pipeline produces both unstructured and structured cases through a setup and simulation phase. The setup phase comprises the mesh generation given a problem geometry, and the simulation step consists in solving the flow case and sampling the $\mathbf{A}$, $\mathbf{f}$, and $\mathbf{u}_0$ fields over time.

Concerning the meshing phase, there are several aspects that need to be covered. Most importantly, there are six different types of meshes, although only the first three are used in three dimensions:

i) Structured,

ii) Triangle Delaunay,

iii) Triangle-Quad Delunay (Quad-dominant),

iv) Triangle Advancing Front,

v) Triangle-Quad Advancing Front, and

vi) Triangle Advancing Front Ortho (with orthogonal smoothing).

The way the meshes are generated is by defining the number of surface cells in all directions, N_c. This translates to a mesh where the sides have a surface mesh of $N_c \times 1$ (in 2D). Hence, constructing a structured mesh is direct. A 2D mesh will have a domain size of $N_c \times N_c \times 1$ (due to ReFRESCO being 3D-based), and a 3D mesh will be N_c cubed. For the other methods, the surface meshing strategy varies: for triangle Delaunay, triangles are seeded on the surfaces, which then leads to tetrahedra in the volume. For triangle-quad Delaunay, triangles are initially seeded, then recombined into a hex-dominant surface mesh, which generates tetrahedra and pyramids in the volume. This can give place, especially in three dimensions, to a drastic jump in cell counts from a structured to a more complex mesh. For instance, the following Tables 5.1.4 and 5.1.5 show how the same N_c parameter can produce very different mesh sizes.

Table 5.1.3: Boundary conditions per case (axes as in the dataset).

Boundary	Poiseuille	Channel	Plate	ConvDiff
x^-	Velocity (parabolic)	Velocity $(1,0,0)$	Velocity $(1,0,0)$	Inflow $u = 0.705$
x^+	Pressure (fixed)	Pressure (outlet)	Outflow (zero-gradient)	Pressure (open)
y^-	Wall (no-slip)	Wall (no-slip)	Wall (no-slip)	Inflow $v = 0.705$
y^+	Wall (no-slip)	Wall (no-slip)	Pressure (open)	Pressure (open)

Following, some examples of meshes are presented in Figure 5.1.5. In the first row, structured meshes are presented for various N_c, while on the lower row, different types of meshes are displayed for the same N_c.

With respect to the simulation process, based on the CFD setup corresponding to the problem and the desired permutations, the ReFRESCO simulation is initialized, and based on a sampling parameter, the Poisson matrix and vectors are stored in sparse format once the iteration count reaches a certain number. To keep the dataset diverse with a wide range of **A** matrices and **f** vectors, there are 20 samples per problem configuration. This means that for each problem, mesh size, and type, 20 different **f** vectors are sampled for a specific configuration. This methodology works indistinctively for 2D and 3D problems, although care has to be taken with the higher-dimensional ones, as the cost of training and storing them increases extremely fast with the parameter N_c.

Table 5.1.4: Number of cells per case, N_c, and mesh type for two-dimensional data. Definitions: i) Structured, ii) Triangle Delaunay, iii) Triangle-Quad Delunay (Quad-dominant), iv) Triangle Advancing Front, v) Triangle-Quad Advancing Front, and v) Triangle Advancing Front Ortho (with orthogonal smoothing).

Problem	N_c	i)	ii)	iii)	iv)	v)	vi)
Poiseuille*	**10**	200	626	358	494	281	400
	14	392	1248	713	970	564	784
	20	800	2488	1413	1894	1085	1600
	28	1568	4876	2761	3734	2082	3136
	40	3200	10016	5686	7570	4204	6400
	56	6272	19702	11181	14682	8022	12544
	79	12482	39076	22221	29168	15812	24964
	111	24642	77256	43813	57908	31280	49284
	156	48672	-	86143	64267	97344	-
	220	96800	-	-	-	-	-
Channel ConvDiff Plate †	**10**	100	316	179	256	143	200
	14	196	602	338	492	284	392
	20	400	1248	708	974	562	800
	28	784	2472	1403	1886	1072	1568
	40	1600	5040	2850	3808	2133	3200
	56	3136	9890	5615	7406	4091	6272
	79	6241	19612	11107	14634	8037	12482
	111	12321	38582	21935	28810	15629	24642
	156	24336	76132	43189	58074	31340	48672
	220	48400	-	85794	-	64775	96800
	311	96721	-	-	-	-	-

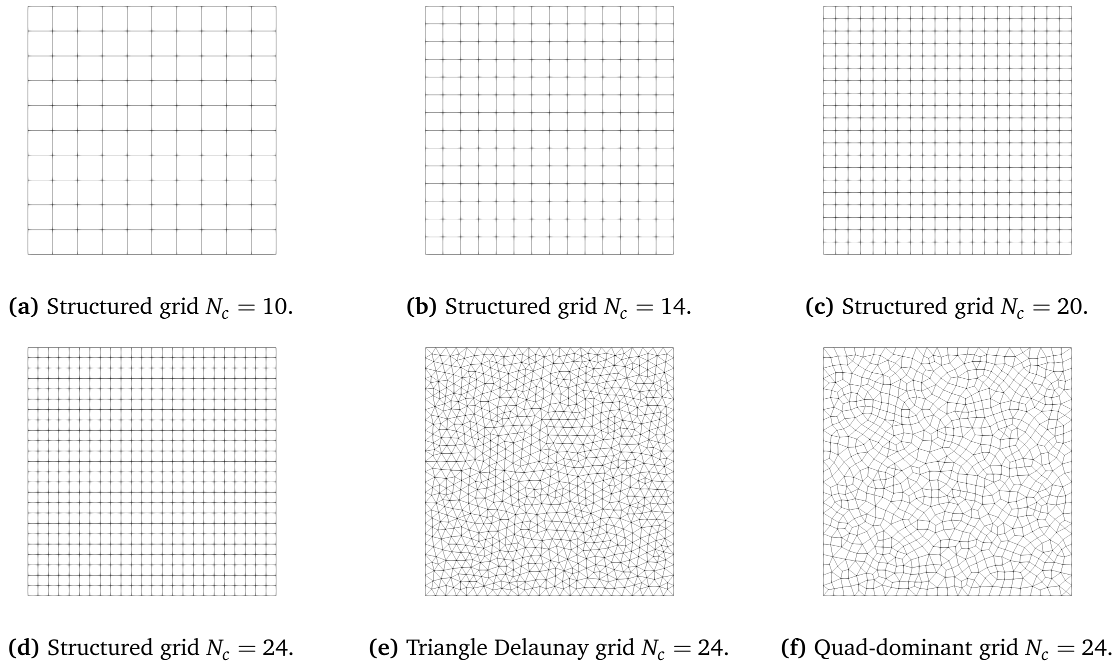
*Poiseuille data is for the $2.0 \times 1.0 \times 1.0$ box size, as no $1.0 \times 1.0 \times 1.0$ data is present.

†Channel, ConvDiff, and Plate data for the $1.0 \times 1.0 \times 1.0$ box size are identical.

Table 5.1.5: Number of cells per case, N_c, and mesh type for three-dimensional data. Definitions: i) Structured, ii) Triangle Delaunay, iii) Triangle-Quad Delunay (Quad-dominant), iv) Triangle Advancing Front, v) Triangle-Quad Advancing Front, and v) Triangle Advancing Front Ortho (with orthogonal smoothing).

Problem	N_c	i)	ii)	iii)	iv)	v)	vi)
Poiseuille Channel ConvDiff Plate *	**8**	512	6654	11057	-	-	-
	9	729	8718	15313	-	-	-
	10	1000	12276	21177	-	-	-
	11	1331	13601	23552	-	-	-
	12	1728	18653	32047	-	-	-
	13	2197	23424	40285	-	-	-
	14	2744	29717	51622	-	-	-
	15	3375	36416	61969	-	-	-
	16	4096	44358	75893	-	-	-
	24	13824	142582	239499	-	-	-

* Since several geometries are available, an approximate number of cells is provided that represents all domain sizes.



(a) Structured grid $N_c = 10$. **(b)** Structured grid $N_c = 14$. **(c)** Structured grid $N_c = 20$.

(d) Structured grid $N_c = 24$. **(e)** Triangle Delaunay grid $N_c = 24$. **(f)** Quad-dominant grid $N_c = 24$.

Figure 5.1.5: Example of meshes used throughout the study.

5.1.3. AirfRANS dataset

Aside from the WaterLily and ReFRESCO datasets, the AirfRANS dataset allows testing the results on practical aerodynamic simulations in two-dimensional airfoils. With this, a missing piece in the literature is fulfilled, i.e., testing the actual models in simulations that actually represent the physics of relevant aerodynamic flows.

Referencing the original motivation behind the creation of AirfRANS, this dataset focuses on practical two-dimensional cases. Despite their simplicity, these cases capture several key challenges inherent to applying machine learning methods in fluid dynamics [81]. Specifically with relation to this study these challenges are:

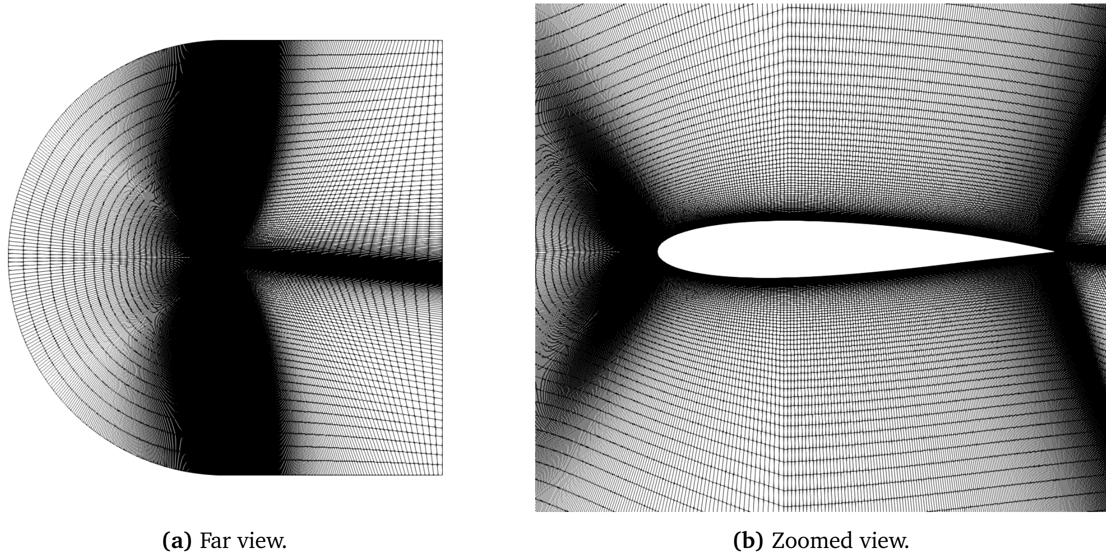
- Working with structured data coming from raw numerical simulations,
- Being able to deal with the number of nodes required in simulation meshes (in this case, a quarter million of cells), and
- Using realistic Reynolds-number test cases.

Furthermore, this dataset is characterized by:

- 1000 simulations,
- Reynolds number between 2 and 6 million,
- Angle of attacks between -5° and 15°, and
- Airfoil drawn in the NACA 4 and 5 digit series

This dataset is provided in raw OpenFOAM, which unfortunately does not include the Poisson system and thus $\mathbf{A}, \mathbf{f}, \mathbf{u_0}$. Luckily, some grids, BCs, and flow conditions could be used in ReFRESCO. The flow is no longer laminar, and the Reynolds number is certainly large. Thus, a turbulence model must be used in the simulation to generate the dataset. Accordingly, for each of the geometries, the turbulence model $k-\omega$ SST has been used (Menter 2003). An average of

approximately 280,000 cells has been used over the dataset. The minimum and maximum number of cells are 245,686 and 324,166, respectively. An example of a mesh for an airfoil case is presented in the following Figure 5.1.6.



(a) Far view. **(b)** Zoomed view.

Figure 5.1.6: Illustration of a grid for an AirfRANS case.

5.1.4. Dataset summary

The following Table 5.1.6 summarizes the different types of problems that are used throughout the project.

Table 5.1.6: Summary of different datasets used in the study.

Source	Categories	Problems	Dim	Mesh Type
WaterLily	Synthetic Unsteady	Static / Dipole / Sphere Circle / Donut / TGV	2D / 3D	Structured
ReFRESCO	Aerodynamic flows	Poiseuille / Channel / ConvDiff / Plate	2D† / 3D††	Structured / Unstructured
	AirfRANS	Airfoils	2D	Structured

† For 2D, the unstructured mesh types are: Triangle Delaunay, Triangle-Quad Delaunay (Quad-dominant), Triangle Advancing Front, Triangle-Quad Advancing Front, and Triangle Advancing Front Ortho (with orthogonal smoothing).
‡ For 3D, the unstructured mesh types are: Triangle Delaunay, and Triangle-Quad Delaunay (Quad-dominant).

5.2. AMG-GNN framework

Implementing a GNN within the AMG cycle is an intricate process that has to be done with care, especially within the PyTorch framework. This section presents the architecture that has been adopted in this research and the reasoning behind the elements that constitute it, as well as

some past ideas of smoother tuning. The introduction of the AMG-GNN framework continues with a showcase of how the GNN was introduced within the AMG V-cycle. Further, the training and validation methodologies are discussed, with a thorough explanation of the setup that was used.

5.2.1. GNN architecture

As mentioned in the introduction (Section 1.4), the acceleration of a Jacobi smoother, and thus AMG solver, is achieved using a modified graph convolutional isomorphism network (GCIN) from AutoAMG(θ) [24], which is combined with a prediction layer (MLP) to determine the optimal polynomial coefficients. The idea behind this network is to combine a GCN and a GIN into a single GNN, as they were tested separately without success. The motivation for this two choices are, starting from the GCN, that a graph convolutional network is a simple, efficient, and widely used alternative to a CNN for graphs, although it is limited to immediate neighbors (1-hop)[1]. Further, a graph isomorphism network aims to achieve a more expressive network that can distinguish similar but topologically different graph structures. The latter is of special interest when using unstructured grids, as the topology completely changes from one to another.

The architecture is presented in Figure 5.2.1, and the base setup is presented in Section 6.1, where the optimal GNN parameters are chosen based on an initial study. A thorough explanation of the different blocks that constitute the network is presented in the following sections, from the main GNN blocks to the different activation functions and what the inputs and outputs of the network are.

5.2.1.1. Graph network blocks

The GCIN consists of a GCN block to which the expressiveness power has been increased by removing the weighted average (see Equation 4.2.3), and removing the self-loops as they are already included in the graphs. Removing the weighted average is equivalent to setting no normalization on the GCN. The PyTorch definition of the layer is `GCNConv(nodes_in_dimension, nodes_out_dimension, add_self_loops=False, normalize=False)`. After the GCIN, a batch normalization layer (`BatchNorm1d`) is used to stabilize and speed up convergence, as well as prevent the activations from drifting as the network learns (Section 4.1.3.4). Following, a ReLU layer is applied and Global mean pool (GMP) (`global_mean_pool`) is done on the graph so that the features from all nodes are condensed into a single feature vector with the same size, i.e.,

$$\mathbf{x}_g = \frac{1}{|V|}\sum_{i=1}^{|V|}\mathbf{x}_{v_i}. \tag{5.2.1}$$

This allows the pooled result to be passed through a linear transformation (multilayer perceptron) so that, if more layers of GNN with different hidden dimensions are used, the result after pooling will have the same size.

The previous blocks can be considered the bulk of the GNN. The next steps of the network, more related to the prediction of the optimal coefficients, consist in a readout function that combines the outputs from the GNN (from all layers if more than one) and an MLP used to combine all learned information from the initial graph (matrix of coefficients) to predict the polynomial coefficients used to construct a pseudo-inverse smoother matrix.

[1] A more powerful convolution that combines information from more neighbors is the topology adaptive graph convolution (TAGConv [82]).

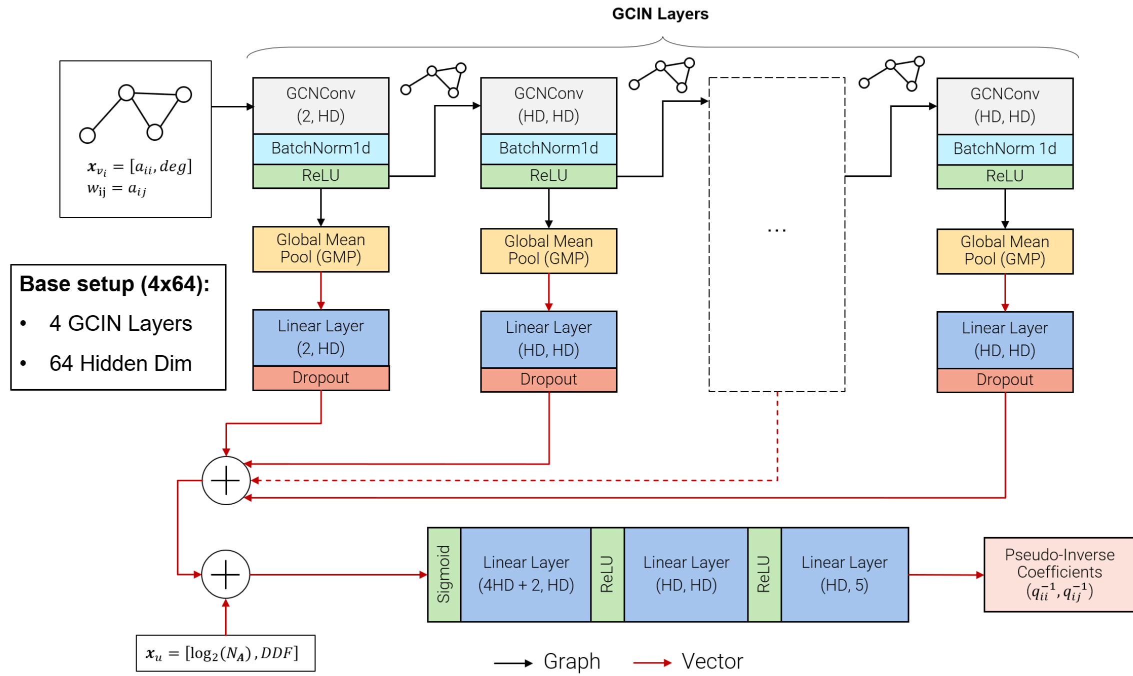


Figure 5.2.1: GCIN architecture. Definitions: node degree (deg), hidden dimension (HD), diagonal dominance factor (DDF), size of the coefficients matrix (N_A).

The custom readout function for the GNN is a concatenation of each of the layer pooling outputs (after the linear transformation),

$$\mathbf{x}_G = \text{CONCAT}\left(\mathbf{x}_g^{(k)} \,\middle|\, k = 0, 1, \ldots, K\right). \tag{5.2.2}$$

Finally, the readout output, which is concatenated with appropriate global attributes representing the input graph (e.g., number of non-zeros, matrix size, diagonal dominance factor, etc.),

$$\mathbf{x}_{GA} = \text{CONCAT}\left(\mathbf{x}_G \,\middle|\, \mathbf{x}_u\right), \tag{5.2.3}$$

is passed through a sigmoid and introduced into an MLP with a hidden layer. This prediction network has ReLU activation functions except on the output, whose size depends on the degree of the polynomial that is used to construct the smoother matrix.

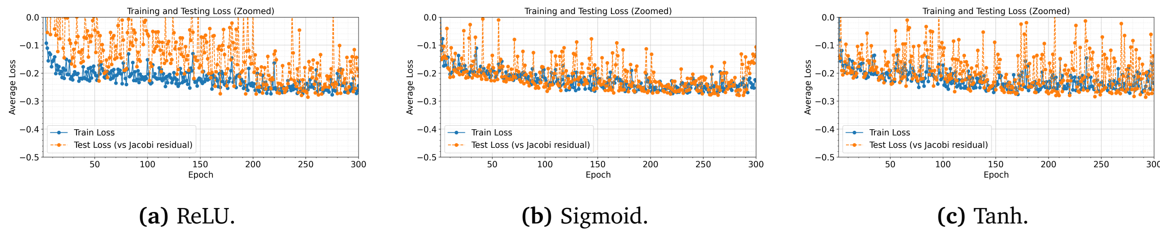
5.2.1.2. Activation functions

In the previous section, two different activation functions have been presented in the GCIN: the ReLU and the sigmoid.

Nowadays, ReLU is the go-to activation function in large deep networks due to its advantages, such as being computationally simple, avoiding saturation (unless it becomes a dead ReLU), encouraging sparsity (around half the activations become zero), and a better gradient scaling, i.e., no squashing. There are other ReLU variations aimed at improving some of the drawbacks of the activation function, such as leaky ReLU, which keeps a small slope for negative inputs, or parametric ReLU (PReLU), which includes a trainable parameter to learn how much leakage is optimal. More complex ReLU options are available, but the performance of the basic ReLU layers

did not produce behaviors that required a modification.

As it may be noticed, a sigmoid activation function is also used before the final MLP. Although sigmoids compress the data and limit the expressiveness of the GCIN, the end result of the network is to generate polynomial coefficients which, generally, are relatively small. Having too large values at the input may hinder the gradient propagation due to the weights of the prediction layer being too small, which also affects the prior network blocks. Hence, compressing the values of $\mathbf{x}_{GA}$ provided the faster training and slightly better performance. This can be observed in Figure 5.2.2. There, although the train loss is similar in both cases, the test loss is consistently lower, and the range of variation is considerably narrower.



(a) ReLU. **(b)** Sigmoid. **(c)** Tanh.

Figure 5.2.2: Loss history for three different activation functions before the predictive MLP.

5.2.1.3. Input and output

In this GNN architecture, the input of the network must be a graph. Note that being limited to a graph input does not mean that mini-batching cannot be done. PyTorch Geometric handles it smartly by joining different graphs together into a large graph, which are separated by ghost nodes. It should not be forgotten that the vector of global attributes is also an input of the network, although not for the GCIN block, but for the final prediction. Each of the inputs is characterized as follows:

- **Input graph**:
  - Node features $\mathbf{x}_{v_i}$: a $\mathbb{R}^2$ vector that stores at node v_i the diagonal value of the matrix $\mathbf{A}$ (a_{ii}) and its degree.
  - Edge features w_{ij}: weight of edge connecting from node j (sender) and i (receiver). The weight is the off-diagonal value of the matrix a_{ij}, unless the edge is from a self-loop ($i = j$). In that case, the diagonal value is stored.
- **Global attributes**:
  - Number of rows/columns of coefficient matrix: $\log_2(N_{\mathbf{A}})$.
  - Average diagonal dominance factor: measure of how strongly it is coupled to its neighbors,

$$\bar{d} = \frac{1}{N_{\mathbf{A}}} \sum_{i=1}^{N_{\mathbf{A}}} \frac{|a_{ii}|}{\sum_{i \neq j} |a_{ij}|}. \tag{5.2.4}$$

The reason behind each of the global attributes is as follows. First, the idea of providing the GNN with the size of the graph/matrix could allow it to infer better polynomial coefficients if a matrix is tested larger than the training dataset, as long as an adequate mapping is learned. Second, the diagonal dominance factor is directly correlated to how effective a Jacobi smoother will be. A strongly diagonally dominant matrix will converge faster. Further, it can also provide information on how ill-conditioned the system is and how aggressive the smoother should be.

5.2.2. Graph generation

The methodology to generate graphs to feed the GNN is straightforward. The graphs are created together with the AMG hierarchy levels, that is, once all the levels of the AMG have been created along with their respective coefficient matrices $\mathbf{A}^{(l)}$, a graph is constructed, given the associated matrix for each level $G^{(l)}$.

To do so, the structured and unstructured data must be loaded first, which is stored in numpy format (.npz), storing $\mathbf{A}$, $\mathbf{f}$, and $\mathbf{u}_0$ together. The vectors are extracted, while the coefficient matrix must undergo a bit of work. The indices and values must be extracted, and from them, a sparse matrix in Compressed Sparse Row (CSR) format is constructed, as this format provides more efficient matrix computations.

Then, the graphs are generated using the row, columns, and matrix values as follows:

1) The edge indices (receptor and sender node) are constructed from the rows and columns stacked.
2) The edge attribute/weight is directly determined from the off-diagonal value of the matrix, a_{ij} (indices: row i, column j). The choice of a single edge scalar attribute is given by the node-based nature of the network, more in Section 5.2.1.
3) With respect to the node features, two attributes have been selected to represent the matrix as a graph:
   a) The first node feature is the diagonal coefficient matrix value, a_{ii}.
   b) The second attribute is aimed at providing the GNN information about the mesh and connectivity. Hence, the degree (number of connections) of the node is also stored as a feature.
4) The graph is constructed using the PyTorch Geometric library by simply using the `torch_geometric.data.Data` function.

Notably, the graph is constructed without global attributes. These are system-wide features, i.e., shared by all nodes but featured only once in the graph, such as the matrix dimension ($\log_2 N_A$) or the average diagonal dominance. As mentioned, this is a choice due to the need to modify the GCNConv layers to handle them.

As an example of how the graph construction works, in Figure 5.2.3, the matrix of coefficients from the AMG level 2 (base level is 0), $\mathbf{A}^{(2)}$, is displayed along with its associated graph, where the node value is a_{ii}. The problem associated with the matrix is a flat plate flow for $N_c = 10$ and triangle Delaunay meshing (626 cells).

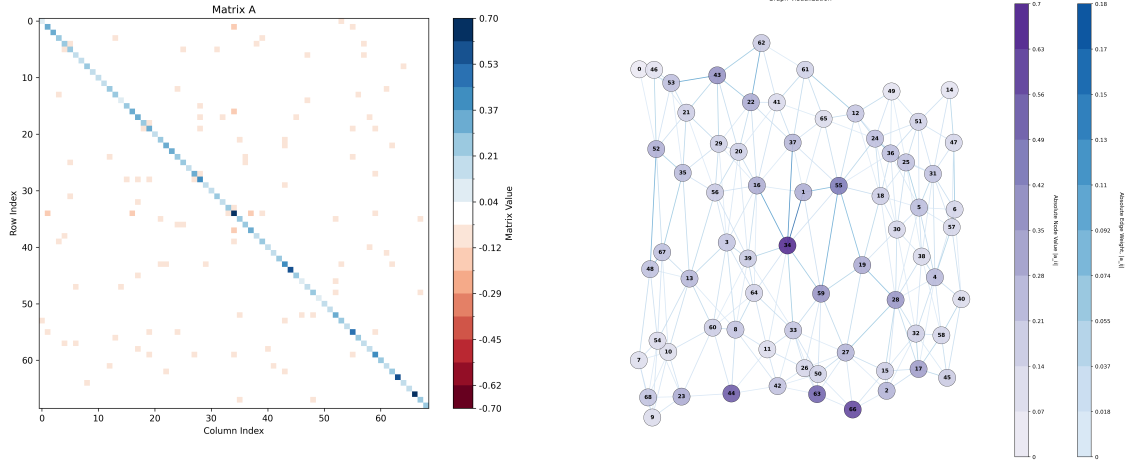
5.2.3. AMG implementation

The network architecture that handles the polynomial coefficients for the pseudo-inverse smoother matrix has been presented. To follow up on the methodology of the AMG-GNN framework, it is necessary to explain how the model is implemented within the algebraic multigrid V-cycle.

In this respect, the AMG-GNN framework integrates a learned smoother into a standard Ruge–Stüben multigrid hierarchy generated with pyamg, while executing the V-cycle and all model components in PyTorch to leverage GPU acceleration and automatic differentiation.

5.2.3.1. V-cycle

The structure of the V-cycle is kept as in a common V-cycle. Remember that in an AMG solver, the needed components for each level are the matrix of coefficients $\mathbf{A}^{(l)}$, the restriction $\mathbf{P}^{(l)}$ and



(a) Matrix of coefficients $\mathbf{A}^{(2)}$.

(b) Associated graph $G^{(2)}$. Only the first node feature is displayed (a_{ii}).

Figure 5.2.3: Illustration of level 2 of AMG hierarchy for a plate flow with N_c and triangle Delaunay meshing.

the prolongation $\mathbf{R}^{(l)}$ operators, and the smoother matrix $(\mathbf{Q}^{-1})^{(l)}$. The residual $\mathbf{r}^{(l)}$ and error (approximation) $\mathbf{e}^{(l)}$ are obtained from the process.

To generate the hierarchy of levels in the V-cycle, the original matrix (finest level) from the Poisson equation is required $\mathbf{A}^{(0)}$. With it, using the `pyamg.ruge_stuben_solver(A_sparse, max_levels, strength)` a hierachy is built of L levels, based on a type of strength. In this case, the strength used is the default classical $\theta = 0.25$, and the number of levels will be dependent on the size of the original matrix and the `max_levels` parameter. If it is small, then $L < L_{max}$, otherwise $L = L_{max}$. For this study, $L_{max} = 10$, as having a small number will limit the size of the V-cycle, and the coarse direct solve will take over a lot of work, which can also worsen performance in terms of solve time for larger matrices.

During the generation of the hierarchy, the graph associated with each level is constructed $G^{(l)}$, together with the global attributes $\mathbf{x}_u^{(l)}$. Both are later fed to the GNN in order to generate the pseudo-inverse smoother.

The V-cycle implemented follows a similar structure to Algorithm 1. For each level:

(1) Pre-smooth with the chosen smoother $(\mathbf{Q}^{-1})^{(l)}$ for ν_1 iterations.

(2) Compute the residual $\mathbf{r}^{(l)} = \mathbf{f}^{(l)} - \mathbf{A}^{(l)}\mathbf{u}^{(l)}$.

(3) Restrict the residual to the lower level $\mathbf{r}^{(l+1)} = \mathbf{R}^{(l)}\mathbf{r}$

(4) Check if it is the coarsest level.

   a) In that case, perform a direct solve with `torch.linalg.solve`, in the system $\mathbf{e}^{(L)} = \mathbf{A}^{(L)}\mathbf{r}^{(L)}$ with $l = L - 1$.

   b) Otherwise, go to (1) with $l \leftarrow (l+1)$.

(5) Prolongate the coarse error to a finer level $\mathbf{u}^{(l)} = \mathbf{u}^{(l)} + \mathbf{P}^{(l+1)}\mathbf{e}^{(l+1)}$.

(6) Post-smooth with the chosen smoother $(\mathbf{Q}^{-1})^{(l)}$ for ν_2 iterations. Go to (5) with $l \leftarrow (l-1)$.

In this implementation, the number of pre- and post-smoothing steps has been set to $\nu_1, \nu_2 = 2$. Some experiments have been conducted with a different value and can be found in Section 7.1.2.2.

5.2.3.2. Batched V-Cycle

Ideally, when implementing code for deep network training, the code should be parallelizable and allow working with batches as much as possible. For instance, in a CNN, images of the same size can all be processed together in a batch. However, when working with matrices coming from the incompressible flow Poisson equation, it is highly probable that, although having the same size, the values and their distribution on the matrix of coefficients will differ. Since the Ruge-Stüben algorithm produces hierarchies where matrices at the same level can differ in size ($n_1 \neq n_2$) and sparsity, they cannot be stacked into a standard dense tensor for batched processing. As a result, training the AMG-GNN through a V-cycle prevents the use of simple batching strategies, hindering the extraction of the full parallel benefits of the GPU.

This is an actual issue in this implementation, as the loss function compares the residual at the start and after the V-cycle. Other works use other kinds of loss functions that require a lot of computational effort offline. More on the loss function in Section 5.2.4.2.

5.2.3.3. Pseudo-inverse smoother

The pseudo-inverse smoothers are built before running the V-cycle and right after generating the hierarchy of levels. There are two main parts regarding the construction of the smoother matrices: the prediction of the polynomial coefficients and the construction of the pseudo-inverse.

The process of generating the prediction of coefficients is straightforward. For each level of the AMG hierarchy that contains $G^{(l)}$ and $\mathbf{x}_u^{(l)}$, a forward step of the GNN is executed, obtaining the polynomial coefficients for each level $\mathbf{p}^{(l)}$.

$$\mathbf{p}^{(l)} = \mathrm{GCIN}\left(G^{(l)}, \mathbf{x}_u^{(l)} \;\middle|\; \boldsymbol{\theta}\right). \tag{5.2.5}$$

Note that the GCIN shares the same weights for all levels; a single GNN is trained. Also, the output size of the polynomial coefficients directly depends on the degree used to build the pseudo-inverse.

The sparse pseudo-inverse is constructed in two parts, using diagonal terms and off-diagonal. Each of them will follow a different polynomial function to generate the smoother matrix, i.e.,

$$q_{ii}^{-1} = \frac{f_d(a_{ii}/s)}{a_{ii}}, \qquad q_{ij}^{-1} = \frac{f_o(a_{ij}/s)}{a_{ii}+a_{jj}} \;\; (i \neq j), \tag{5.2.6}$$

where the function inputs are scaled by the maximum off-diagonal $s = \max(\mathbf{A}-\mathbf{D})$. Following the original work [30], a second-degree polynomial is used that satisfies that off-diagonal elements that are zero must remain non-zero $f_o(0) = 0$. Otherwise, the matrix might not be sparse anymore. Hence, the diagonal and off-diagonal functions are given by

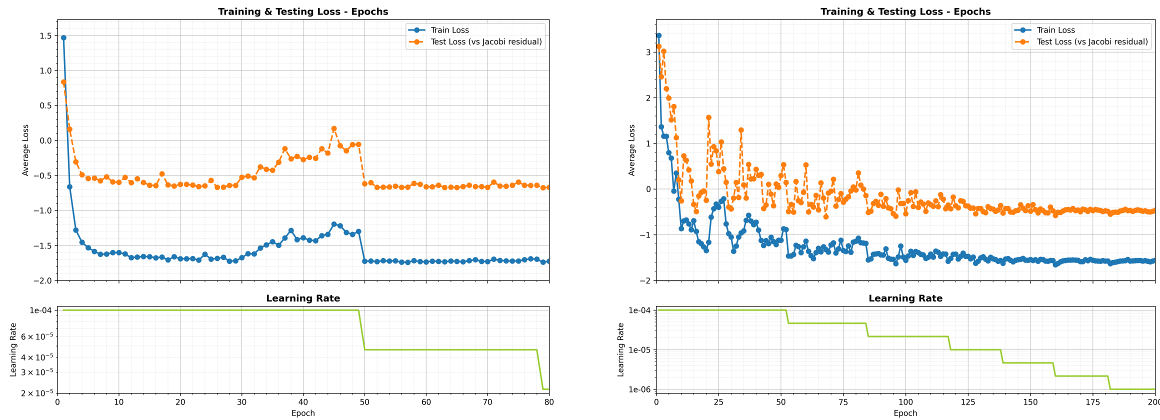
$$\begin{aligned} f_d(\alpha \mid \mathbf{p}^{(l)}) &= 1 + p_0^{(l)} + \alpha\left(p_1^{(l)} + \alpha p_2^{(l)}\right), \\ f_o(\alpha \mid \mathbf{p}^{(l)}) &= \alpha\left[p_3^{(l)}(\alpha-2) + p_4^{(l)}(\alpha-1)\right]. \end{aligned} \tag{5.2.7}$$

Note that the standard Jacobi smoother is defined as $f_d = 1, f_o = 0$. Therefore, five coefficients are needed per level if a quadratic is used, $\mathbf{p} \in \mathbb{R}^{L\times 5}$.

Note that in the original study, higher degrees were tested with similar results. However, for the GCIN, using a third-degree polynomial completely overfits the data. To show this, Figure 5.2.4 presents the difference in training between a second-order and third-order polynomials for the pseudo-inverse matrix. The higher degree requires many more iterations to reach convergence.

Regarding the construction of the pseudo-inverse, it is a time-consuming function due to its sequential nature, given the sparse format. One of the more intense steps is especially computing the off-diagonal term q_{ij}^{-1}. This is due to a_{ii} and a_{jj} being needed, and the fact that a large loop through the sparse matrix is required to find the coefficients. In this respect, a new formulation for the off-diagonal function has been implemented, where both terms are substituted by a_{ij}, and results are presented in Figure B.1.6a.

$$q_{ij}^{-1} = \frac{f_o(a_{ij}/s)}{a_{ij}} \quad (i \neq j). \tag{5.2.8}$$



(a) Second-degree (80 epochs).

(b) Third-degree (200 epochs).

Figure 5.2.4: Loss history for two different polynomial degrees for constructing $\mathbf{Q}^{-1}$.

To finalize, it should be mentioned that this is not the only smoother that has been implemented. Accordingly, a relaxed Jacobi smoother with relaxation parameter $\omega = 2/3$ can be found in the code and is actually used to compare against the tuned Jacobi (pseudo-inverse Jacobi). Further, Gauss-Seidel and SOR smoothers have also been implemented using the pyamg wrapper to compare performance versus the AMG-GNN framework.

5.2.4. Training process

Training the AMG-GNN framework involves building up a PyTorch environment as efficiently as possible in terms of mini-batching. As previously mentioned, the V-cycle function has a fixed hierarchy, hindering the parallel capabilities of the GPU. In this regard, this section aims to introduce the different choices that were made with respect to each of the elements of the training loop in the search for an effective performance.

5.2.4.1. Data loading and mini-batch size

Training a GNN with large graphs (matrices) is a memory-intensive process. For optimal performance, one should aim to train to the maximum capabilities of the GPU while maintaining a

representative mini-batch size, neither too large nor too small.

Now, before delving into how the mini-batches are handled, it is important to examine the structure of the data loading (DataLoader). In this regard, the data loading has been optimized to consume just the necessary memory. To do so, a class `GNNProblemDataset` has been created such that it stores only the problem IDs and the data where they are stored. The object containing the indices is an argument to the data loader, and there is one for training, testing, and validation. On demand, a method from the class loads $\mathbf{A}^{(0)}$, $\mathbf{f}$, and $\mathbf{u}_0$ from the disk.

In addition, a custom collate function filters out items and returns lists (of variable length) for matrices and vectors. This intentionally avoids stacking into fixed-size tensors and is an argument for the PyTorch DataLoader. Therefore, keeping batches as lists and loading on demand keeps memory predictable while still leaving room for GNN mini-batching inference.

Between the GCIN and the V-cycle, there is a compromise. The GNN can handle a large mini-batch, while the V-cycle can only manage one problem ($\mathbf{A}^{(0)}$) and its hierarchy at a time. To solve this as efficiently as possible, training over an epoch is done as follows.

1) Read a batch from the custom DataLoader, getting a list of matrices $\mathbf{A}_b^{(0)}$ and vectors $\mathbf{f}_b$, $\mathbf{u}_{0,b}^{(0)}$ of size equal to the mini-batch ($b = 0, 1 \ldots, B$).
2) Per-graph AMG setup (not batched). For each problem in the batch, the AMG hierarchy is built, producing a list of levels. Each level contains information about $\mathbf{A}_b^{(l)}, \mathbf{P}_b^{(l)}, \mathbf{R}_b^{(l)}, G_b^{(l)}$ and $\mathbf{x}_{u,b}^{(l)}$. For each level that is not the coarsest (as the coarsest only stores $\mathbf{A}_b^{(L)}$ and is not used in the GNN), the matrices of coefficients, their graphs, and their global attributes are appended to a list. To keep track of which problem b and level l they corresponded to, a mapping is also stored.
3) Batch the data used in the GNN inference. To do so, a simple PyTorch Geometric batch is generated from the graph data list. Also, the global attributes are stacked into one dense tensor. Then, one forward pass of the GNN is performed over this batched level-set, yielding a parameter vector for every level in the whole mini-batch.
4) Batched smoother construction. With all predicted polynomial coefficients, a method from the smoother tuned Jacobi object builds all pseudo-inverse smoother matrices for all levels in the mini-batch $\left(\mathbf{Q}_b^{-1}\right)^{(l)}$ in one go. Then, using the mapping that was previously introduced, the list of smoother matrices is redistributed, aligning with the mini-batch and each problem hierarchy.
5) V-cycle(s) and loss for each problem. For every case in the mini-batch, the PyTorch V-cycle with its problem-specific $\left(\mathbf{Q}_b^{-1}\right)^{(l)}$ and ν_1, ν_2 is run. Then, the one-cycle (or multi-cycle) loss is computed. For the mini-batch, the losses resulting from each problem are averaged. Then, using the autograd graph, the weights of the GNN are updated using backpropagation.

With this design, optimal memory usage and computational efficiency are achieved. Nonetheless, the training is still memory-intensive. The autograd graph that is built for each AMG hierarchy within the mini-batch grows in size extremely fast with larger problems. Thus, constraining the memory is very easy when training with three-dimensional data, and the mini-batch size is limited by memory.

Choosing an optimal mini-batch size is a compromise between accuracy and memory usage. In general, the value of the hyperparameter that performed best in terms of both constraints is $B = 10$. A study on this hyperparameter is presented in Section 6.2, where it is compared to $B = 5$ and $B = 20$, where an initial study was carried out to select optimal hyperparameters for the GNN setup.

5.2.4.2. Loss function

The selection of a loss function has a large impact on the algorithm used for training the GNN. In the literature, there are three main approaches: 1) reducing the spectral radius of the two-level AMG error propagation matrix (e.g. [27]), 2) minimizing the difference between the solver solution and exact solution (e.g., [29]), and 3) reducing the residual of the equation system after the V-cycle (e.g. [30]). The latter two are not used as much, and the last one is the method that has been chosen for this thesis.

Starting from the first option, it is common to use a two-level algorithm to train and optimize the AMG. As mentioned, the main focus revolves around reducing the two-level error propagation matrix $\mathbf{M}$, defined as

$$\mathbf{M} = \mathbf{S}^{\nu_2}\mathbf{C}\mathbf{S}^{\nu_1}, \tag{5.2.9}$$

with $\mathbf{S} = \mathbf{I} - \mathbf{Q}^{-1}\mathbf{A}$ being the smoother matrix, and

$$\mathbf{C} = \mathbf{I} - \mathbf{P}\left(\mathbf{P}^{\mathrm{T}}\mathbf{A}\mathbf{P}\right)^{-1}\mathbf{P}^{\mathrm{T}}\mathbf{A} \tag{5.2.10}$$

is the error propagation matrix of the coarse-level correction. One of the issues with this approach is that the inverse matrix of the coarse system must be computed for each sample of the dataset. This implies that a lot of offline work has to be done on the dataset before training. Furthermore, the matrices that are used should not be too large, as computing the inverse can be extremely costly, unless an algorithm to handle special matrices is developed, which is what is done in this case. This limits the quality and variety of the meshes that are used, as block-periodic triangular meshes are required (two-dimensional). Additionally, the approach consists of a two-level algorithm, which constrains the coarse-grid correction of AMG. Small problem matrices (e.g., $14^2 \times 14^2$) will generally have 3 or more levels, and having fewer levels in the training process can hinder the performance when evaluating the model.

With respect to the second approach, minimizing the difference of the AMG solution versus the exact solution is embedded perfectly with the AMG V-cycle. This loss function is defined as follows:

$$L = ||\Phi^{(0)}(\mathbf{u_0}, \mathbf{f}, k) - \mathbf{u}_*||_2, \tag{5.2.11}$$

where $\Phi^{(0)}$ denotes the multigrid hierarchy from the finest level 0, $\mathbf{u_0}$ is the initial guess, $\mathbf{f}$ is the RHS vector, k is the number of V-cycles and $\mathbf{u}_*$ is the exact solution. Thus, $\Phi^{(0)}(\mathbf{u_0}, \mathbf{f}, k)$ is equivalent to the approximate solution $\mathbf{u_k}$ by performing k steps of V-cycles with $\Phi^{(0)}$.

The advantage of minimizing the above equation instead of the norm of the associated iteration matrix is that it can be evaluated and optimized more efficiently [29]. The drawback of this method is that it requires a large computational graph for automatic differentiation, taking a lot of memory in the process, which can limit the size of the mini-batches and the matrices of the training dataset.

Finally, the last approach is similar to the latter, although with a subtle difference. In this case, instead of searching for a solution that is equal to the exact, the V-cycle focuses on reducing the residual as much as possible after a V-cycle. Nonetheless, the original loss function from [30] has been modified in the AMG-GNN implementation to handle several V-cycles. Now, the loss function reads:

$$L = \sum_{n=1}^{N_{vc}} \log_{10}\left(\frac{|\mathbf{r}_n|_2}{|\mathbf{r}_{n-1}|_2}\right). \tag{5.2.12}$$

That is, the L2 norm of the residual after the V-cycle is compared to the residual before, which can be computed by the current solution $\mathbf{u}$ or the initial guess $\mathbf{u}^{(0)}$. This metric has the same

advantages as the previous one, but the use of the logarithm makes it easier to notice in the loss history whether the GNN is producing nice results or not, as a negative loss would imply that the residual is being reduced after the V-cycle. Further, during the training, a testing dataset is also evaluated to assess the model's performance and avoid overfitting. The loss function for this testing dataset is slightly different. Instead of using the initial residual, it uses the residual after a V-cycle using relaxed Jacobi (ω-J).

$$L = \log_{10}\left(\frac{|\mathbf{r}_{i+1}|_2}{|\mathbf{r}_{i+1}^{\omega-J}|_2}\right). \tag{5.2.13}$$

Hence, when training, if the testing loss is negative, it means that the tuned Jacobi is performing better than the baseline Jacobi smoother. Still, the performance of the validation dataset, which is tested in the model evaluation after training, may be different than that of the testing dataset, especially if the matrices are more complex.

5.2.4.3. GNN initialization

One important aspect of deep neural networks with non-linear activation functions is the weight initialization of the NN, that is, which distribution is used to generate random weights for each of the hidden units.

In this study, the default initialization has been used for all the layers. For the graph convolutional block, GCNConv, Xavier uniform initialization is used for weights, and zero for biases. The batch norm blocks are initialized with weights equal to 1 and bias to 0. Further, the linear layers from the predictive MLP are initialized with Kaiming uniform with $a = \sqrt{5}$.

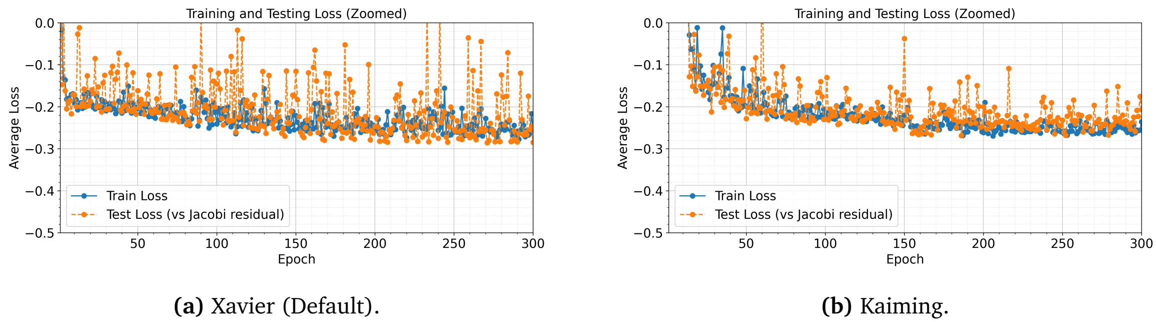
Usually, Kaiming (or He) initializations are more suited for layers where a ReLU activation function is present, while Xavier (or Glorot) are more suited to sigmoid or tanh. The default choice for the batch norm layers is in accordance with the common line of research. Moreover, the Kaiming initialization is well-suited for the last MLP due to the presence of ReLU functions. The idea behind this initialization is that it must ensure that the variance of the hidden unit output stays stable after the forward pass, obtaining a well-behaved convergence for the SGD. This is an issue with Xavier initialization, as it was developed before ReLU and does not take into account that the variance after ReLU is reduced roughly by half, as ReLU discards half the values.

A distinct feature of the standard GCN implementation is the utilization of Xavier initialization, despite the presence of ReLU non-linearities. Xavier uniform was designed for linear layers with symmetric activations (such as tanh), where the forward and backward variance stay roughly constant across layers. Nonetheless, a GCN block is not a standard linear layer. The input to the linear transformation inside the GCNConv is a sum of features from a variable number of neighbors. Thus, this neighbor aggregation increases the variance (roughly proportional to the average degree) that is later adapted by the batch norm layer, hence having an adequate input variance to the ReLU activation function.

To show the effect of a Kaiming initialization on the GCIN, Figure 5.2.5 displays the loss history for the default and updated weights. There, it can be noticed how the Xavier initialization starts significantly better than the Kaiming one. Moreover, the performance of the model from the default initialization keeps its lead over the epochs, especially in the test loss.

5.2.4.4. Optimizer, learning rate and scheduler

To update the neural network weights, the **AdamW** optimizer was selected as the standard for the AMG-GNN framework. While the standard Adam optimizer was initially tested, AdamW provides more robust regularization against overfitting on the structured dataset. The weight decay w_d



(a) Xavier (Default).

(b) Kaiming.

Figure 5.2.5: Loss history for two different GNN initializations.

is maintained at 1×10^{-5} to keep regularization minimal, and gradient clipping is employed to restrict gradients larger than 1.0 and avoid excessively large updates [83]. A comparative analysis of Adam vs. AdamW performance is provided in Figure A.2.1.

The learning rate (η) was tuned between 1×10^{-3} and 1×10^{-4} (see Section 6.2). To adjust the rate during training, a custom **ReduceLROnPlateau** scheduler was implemented. Although Cosine Annealing was evaluated, the plateau-based approach offered the most predictable convergence for this specific application. The scheduler is configured with a reduction factor $\phi = 10^{-1/3}$, a patience of 20 epochs, and a threshold $\tau = 1 \times 10^{-5}$.

Crucially, this implementation was tuned following [84]. When the patience limit is reached, the system reloads the model weights from the best train recorded epoch before reducing the learning rate.

Finally, quasi-second-order optimization using L-BFGS was explored. While it demonstrated potential for lower test loss, memory constraints regarding curvature information made it impractical for the standard setup. Detailed results of the scheduler comparisons and L-BFGS experiments are available in Appendix B.2.6.

5.2.4.5. Test batches and "early-stopping"

When introducing the data loader, it was mentioned that there are three datasets: training, testing, and validation. In this study, there are two that are used during the training phase: the training dataset, which is used to actually update the network parameters based on the loss metric, and the testing dataset, which is used to evaluate the AMG-GNN performance, comparing directly online to a relaxed Jacobi smoother. The testing dataset is of similar characteristics to the training dataset. The matrices are of the same size, although the problem type and mesh generation are different.

One of the main ideas behind the testing dataset is to implement an early-stopping mechanism that does not halt the training loop. That is, the testing dataset gives a measure of how well the model is performing against relaxed Jacobi, but also on new data (although of similar quality to the training one). In some datasets in which training has been performed (i.e., WaterLily structured data), the model has shown overfitting from the start (see Figure 5.2.6, and notice how the loss only increases from the beginning), and it was observed that the final model performed poorly when tested on unseen and larger matrices. Therefore, it was implemented in the testing loop a way to store the model parameters when the testing loss was the lowest, and if reduced, store the new lowest state (best test model). A cooldown parameter was introduced that works as follows: if the last best test model has been updated within *cooldown* epochs, then overwrite the saved data. Otherwise, store the model parameters in a new PyTorch file.

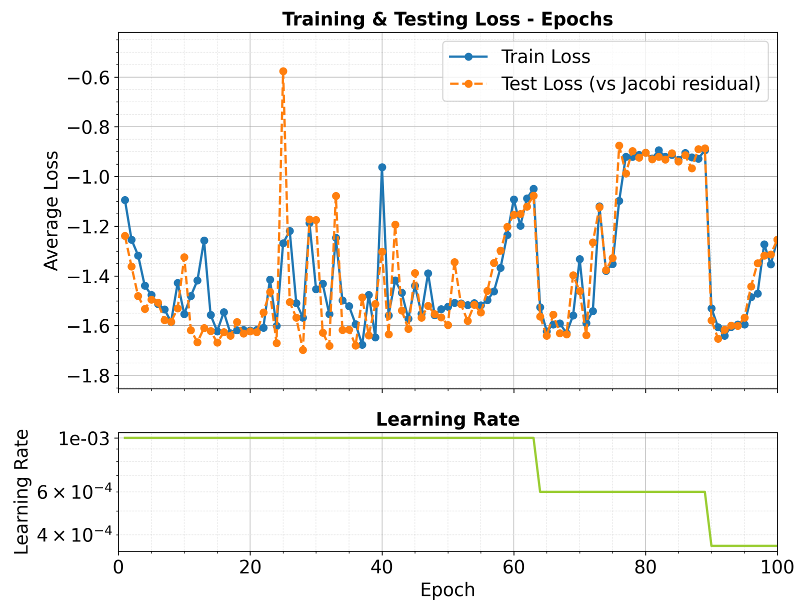


Figure 5.2.6: Train and test loss history on 2D static synthetic cases. Note that, after a minimum loss is reached, both train and test losses increase again.

5.2.4.6. Workflow

To provide more insight into the workflow that has been followed for the training/testing processes, Figures 5.2.7 and 5.2.8 illustrate a simplified flowchart of the different steps that are taken in the AMG-GNN training.

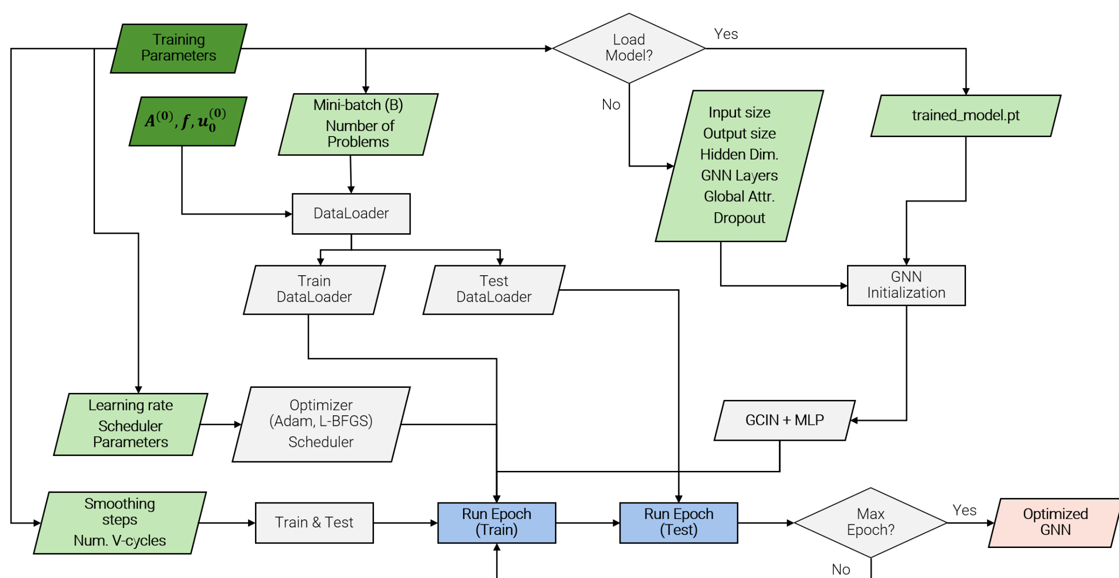


Figure 5.2.7: Train/test flowchart.

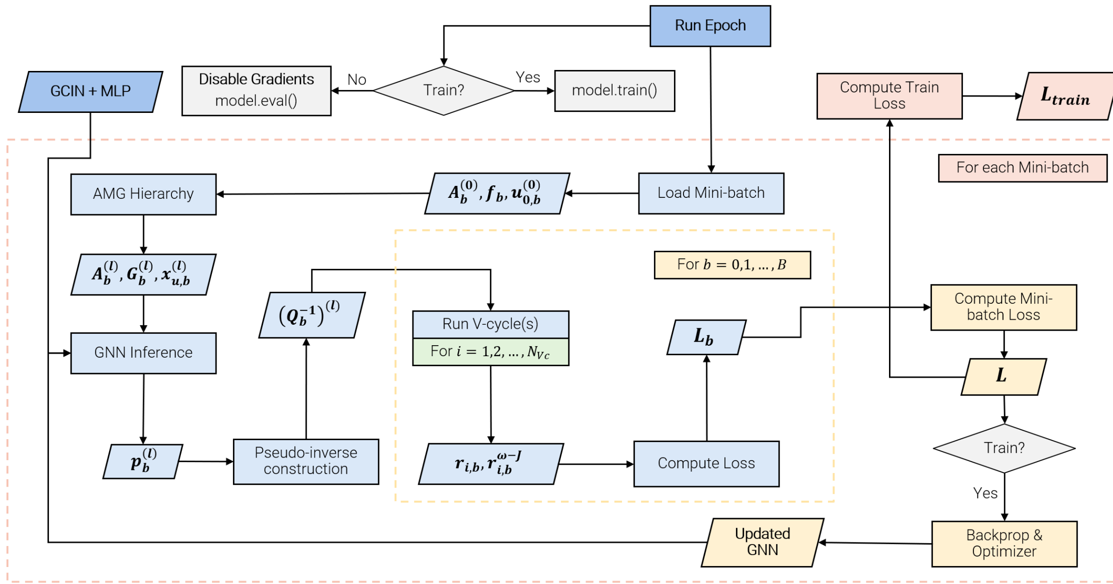


Figure 5.2.8: Epoch flowchart.

5.2.5. Validation process

Once the model has been trained, the performance with respect to the relaxed Jacobi can be estimated through the test loss. Nonetheless, the problems in the test dataset are different from those in the validation dataset. Matrices are significantly larger, the coefficients have a more intricate distribution, and different values. Therefore, it is necessary to validate that the model is useful and generalizes across a wide range of test conditions.

This section presents the approach that has been followed to evaluate the model performance on more difficult and larger problems, outside of the training dataset distribution. The methodology starts from loading a validation dataset, as well as the model that will be validated. The latter is evaluated using a particular setup similar to the training, although more smoothers are used for the comparison. During the evaluation, it should be expected that on some occasions, the model may not perform as expected, and divergence of the solution may be produced. Each of the problems tested will be classified according to whether it converged or not. Finally, the results are processed so that relative solve times are obtained with respect to the baseline relaxed Jacobi AMG.

5.2.5.1. Model and data loading

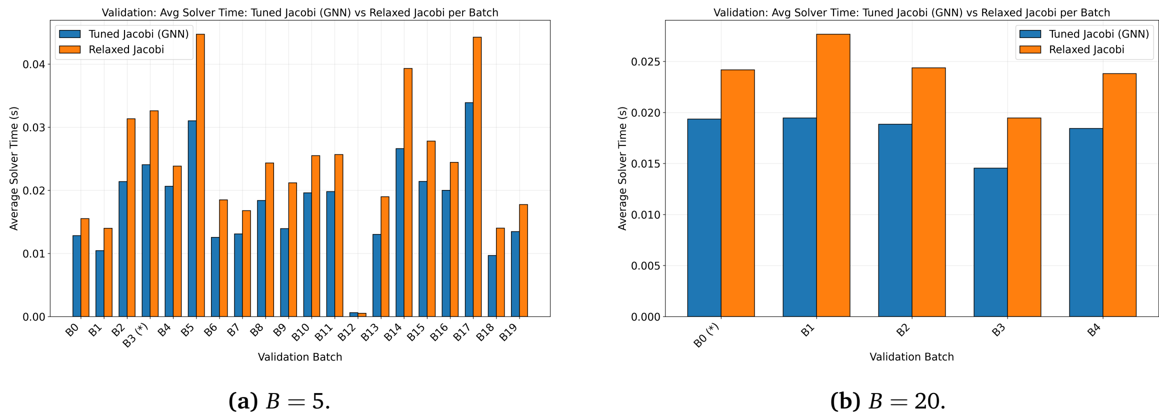
Dataset generation for evaluation follows the training protocol but utilizes larger, more complex matrices. Instead of initializing new parameters, the pre-trained model weights are loaded directly from a saved state. Crucially, validation does not require gradient propagation through the V-cycle, which significantly reduces memory overhead. Despite this, memory constraints can still be reached during GNN inference on the finest meshes, particularly in the unsteady WaterLily cases.

5.2.5.2. Model evaluation

Unlike training, validation inference is performed per-problem rather than per-batch. This approach isolates the setup and solve times for each specific hierarchy, enabling a direct comparison

between the GNN-tuned Jacobi and standard baselines (relaxed Jacobi, Gauss-Seidel, and SOR). Consequently, smaller mini-batch sizes are preferred in the data loader to prevent excessive averaging of metrics across diverse problems, as illustrated in Figure 5.2.9.

To ensure accurate profiling, each problem is solved multiple times. The first iteration is discarded to account for JIT compilation and GPU warm-up, and the subsequent runs are averaged. The AMG solver uses pre- and post-smoothing steps $\nu_1 = \nu_2 = 2$ with a maximum of 250 iterations. The absolute solver tolerance has been adjusted from $\delta = 1 \times 10^{-8}$ to 1×10^{-4} to accommodate the precision limitations of 32-bit floating-point operations (FP32). Finally, the evaluation pipeline includes safeguards to flag non-converging solutions or instances where the GNN significantly underperforms compared to the baseline.



(a) $B = 5$. **(b)** $B = 20$.

Figure 5.2.9: Effect of mini-batch size on validation granularity. Smaller batches ($B = 5$) reveal specific problem outliers that are smoothed out in larger batches ($B = 20$).

Non-convergence classification

For classifying non-convergent cases, three different situations have been considered: the maximum number of iterations is reached, the residual explodes in a short number of iterations (divergence), or the residual remains constant within a tolerance (stagnation).

To identify these cases, the AMG solver has been implemented with an Exit code (EC):

- Solution converged, EC $= 0$.
- Maximum iterations reached, EC $= 0$, Iterations $=$ MAX.
- Divergence, EC $= -1$.
- Stagnation, EC $= -2$.

By checking the exit code after the solve, the validation process removes the problem from the metric average for all smoothers tested. This means that if a tougher problem is removed for one smoother, it will see a net advantage as the average metric will be effectively smaller than the other smoother. This approach is improved and discussed for the three-dimensional unstructured cases in Section 7.2.2 to make the comparison between smoothers as fair as possible.

Another way in which the divergence can be represented is by using a label flag in the bar plots. In general, when a large dataset is used, and when a problem has diverged in the mini-batch, an asterisk $*$ is added to the label. In small validation batches for visualization purposes, the following labeling is used (# is the number of cases):

- m#: Tuned smoother hits max iterations.

- d#: Tuned smoother diverged.
- s#: Tuned smoother stagnated.

Using an uppercase in the label symbolizes that both tuned and relaxed Jacobi did not converge. For instance, Figure B.2.5a shows how the labeling works (and a remarkably poor performance).

5.2.5.3. Workflow

Similarly to the training methodology, a workflow of the validation process is presented in Figure 5.2.10 to provide the reader a comprehensive understanding of the process that is followed during the validation of the AMG-GNN model versus different smoothers.

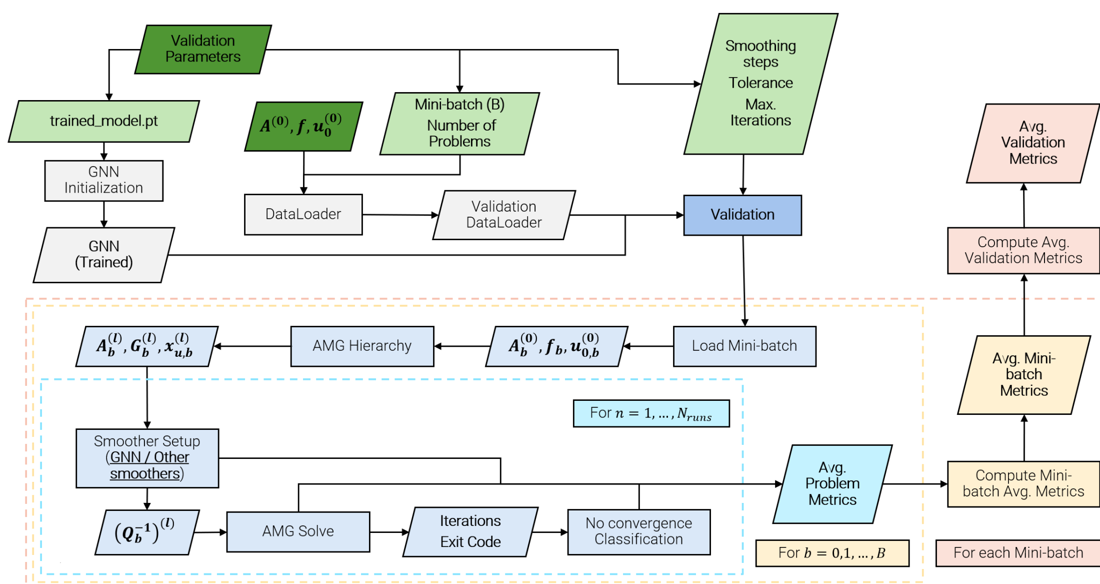


Figure 5.2.10: Validation flowchart.

6

Hyperparameter tuning & optimization

A robust and optimized set of hyperparameters must be established before the AMG-GNN framework can be evaluated on the target CFD datasets. The performance of a deep learning model, particularly a GNN, is critically sensitive to a wide range of configuration choices. These include not only architectural parameters, such as the number of GNN layers and the size of the hidden dimension, but also the complex details of the training process itself, such as learning rate, mini-batch size, and the choice of optimizer.

A non-optimal setup can lead to slow convergence, numerical instability, or a model that fails to generalize, learning only to solve the specific problems in the training set. Such a result would hide the true potential and limitations of the data-driven approach.

This chapter details the systematic process of identifying an optimal and robust configuration for the GCIN model. The investigation begins with initial exploratory tests that reveal the model's high sensitivity to these parameters. This motivates the subsequent sections, which present a one-at-a-time study to isolate the impact of learning rate, mini-batch size, model size, and regularization. The chapter concludes by selecting a final, validated *base setup* that is then carried forward for the comprehensive performance evaluations in the main results chapter.

6.1. Optimization strategy

The optimization of the GNN architecture and training hyperparameters is conducted to maximize wall-clock solve time acceleration while ensuring training stability. To isolate the effects of individual parameters, a one-at-a-time hyperparameter study is performed on a representative subset of the 2D unstructured dataset ($10 \leq N_c \leq 28$, 400 problems), comprising Poiseuille, channel, ConvDiff, and plate flows. This subset provided the optimal balance between graph complexity and computational cost for rapid iteration.

Preliminary tests (detailed in Appendix B.1) have identified distinct sensitivities in the AMG-GNN framework, leading to the following architectural prerequisites:

- **Activation Function:** A sigmoid activation function has been adopted before the predictive MLP. This proves critical for smoothing the training process and accelerating convergence compared to other activation types (see Figure B.1.2b).
- **Gradient Clipping:** To mitigate the instability inherent in training a GNN within a multigrid V-cycle, gradient clipping has been implemented, significantly reducing loss unsteadiness (Figure B.1.1).

- **Learning Rate:** The model has shown high sensitivity to the learning rate due to the stochastic nature of the mini-batches. A standard value of $\eta = 10^{-3}$ has been selected as it offers the most stable convergence profile (Figure B.1.3).

Other hyperparameters, such as dropout (Figure B.1.4a) and scheduler choice (Figure B.1.4), have shown negligible impact on the baseline performance during these initial screenings.

A critical finding from the validation phase is the trade-off between inference cost and solver acceleration. While the AMG-GNN consistently reduces the V-cycle iteration count, the sparse matrix multiplication required by the generated smoother introduces a computational overhead (Figure B.1.6b). Consequently, total time reduction (setup + solve time) is only achieved when the iteration reduction is substantial enough to offset the GNN inference time. However, in the context of unsteady CFD simulations, this overhead is compensated, as the setup is performed once while the solution step is repeated for hundreds of time steps. Moreover, a trade-off between smoother performance and setup time can be achieved by using smaller models and simpler pseudo-inverse matrices (Figure B.1.6b).

6.2. Hyperparameter study

To establish the consolidated setup, a systematic hyperparameter analysis has been conducted on the representative 2D unstructured dataset using the baseline setup from Table 6.2.1. An important aspect of this methodology is to keep the network initialization fixed across the training runs. Although it is fixed for the hyperparameter runs, mini-batch shuffling remains nondeterministic. This is due to some CUDA backends not being able to be set as deterministic. This introduces small variations in the training trajectory, which can lead to noticeable differences in test loss and solve-time performance, especially for smaller batch sizes. The following lines summarize the trade-offs identified for key hyperparameters and justify the selections made for the final model configuration.

The base setup used in this hyperparameter study is presented next in Table 6.2.1.

Table 6.2.1: Base training setup for the hyperparameter study.

Item	Setting
Data type	Float64 (FP64)
Train / test problems	200 / 50
Mini-batch size B	10
Problem size N_c	$10 \leq N_c \leq 28$
Learning rate (η_0)	1×10^{-3}
GNN layers (GNN layers)	4
Hidden dimension (HD)	64
Global attributes (GA)	Yes
Dropout	No
Optimizer	Adam
Scheduler	No
Pre/Post smoothing steps	2
Activation function before MLP	Sigmoid

6.2.1. Learning rate and scheduler

The training process shows significant sensitivity to the learning rate, especially for the larger values B.2.1. Lower rates ($\eta = 10^{-4}$) provided smoother profiles for both train and test losses, at

the cost of frequent stagnation in suboptimal local minima and slower convergence. In contrast, higher rates ($\eta > 5 \times 10^{-3}$) display large variability and exploration of the landscape, preventing the model from achieving a stable minimum. From the different options, a base rate of $\eta = 10^{-3}$ is the final choice, as it offers an optimal balance between exploration and loss stability. Solve times demonstrate that similar test and train losses can lead to substantially different model performance (Figure B.2.2). This is a common finding for all hyperparameters.

Furthermore, to reduce training stagnation, dynamic learning rate scheduling has been evaluated. The cosine annealing scheduler allows for more loss landscape explorations, visiting multiple minima. Nonetheless, while initial tests suggest minor impact, the full hyperparameter sweep reveals that *save* ReduceOnPlateau gives the most stable generalization. Validation solve times position the *save* ReduceOnPlateau scheduler slightly above its two opponents in performance (Figure B.2.10).

6.2.2. Mini-batch Size

The choice of mini-batch size (B) was driven by the need to balance gradient estimation accuracy with memory constraints and training speed. Small mini-batches ($B = 5$) produced high variance in gradient estimation, leading to a higher number of divergences in the validation set if early-stopping is not used (Figure B.2.4). On the contrary, larger batches ($B = 20$) provided averaged gradients and a smoothed loss landscape that hindered the model from finding sharp minima. In the end, the intermediate size of $B = 10$ was adopted as the baseline, offering robust convergence while keeping memory usage within GPU limits during backpropagation.

6.2.3. Model architecture (depth and width)

The model size has been evaluated by varying the number of GNN layers ($2-4$) and the hidden dimension size ($24-64$). On the limited 2D tuning dataset, smaller networks (e.g., 2×64) achieve marginally lower training losses and faster setup times than larger configurations (Figure B.2.7). However, these results are specific to the low-complexity training data ($N_c \leq 28$). The larger baseline configuration (4×64) is retained to ensure sufficient expressiveness for the more complex 3D and highly anisotropic problems encountered in later validation stages, as the inference overhead is considered acceptable with respect to the potential accuracy gains.

6.2.4. Optimization and regularization

While the Adam optimizer serves as the robust baseline, second-order optimization via L-BFGS has been investigated to potentially break through loss plateaus. Although L-BFGS achieves lower testing losses, it produces a significant memory overhead and does not translate into a considerable reduction in the actual AMG solve time during validation (Figure B.2.15). This is possibly due to memory overhead preventing large mini-batch sizes, which are preferred for these second-order methods. Accordingly, the first-order Adam optimizer is maintained.

Similarly, regularization via dropout has been tested (20% and 50% deactivation) but has not produced significant performance improvement over the baseline, especially comparing the final train and best test models (Figure B.2.12). This suggests that the data variety in the training (with respect to dataset size) is sufficient to prevent overfitting without explicit regularization.

6.3. Consolidated optimal setup

The extensive experimental tuning reveals the high sensitivity of the AMG-GNN framework to its architectural and training hyperparameters. This iterative process of isolating variables has converged on the optimized configuration detailed in Table 6.3.1.

While the Adam optimizer has been used for the majority of the sensitivity analysis, AdamW has been selected for the consolidated setup. Its decoupled weight decay provides superior regularization stability. Given the low weight decay values employed (1×10^{-5}), this change preserves the performance characteristics observed with Adam.

The parameters in Table 6.3.1 represent the optimal balance between solve-time acceleration and training stability established for the 2D tuning dataset. This configuration serves as the foundational baseline for the comprehensive validation in Chapter 7. However, it is important to note that this setup is not static. Specific adaptations are required to handle the computational scaling of complex 3D problems:

- **Precision (FP64 to FP32):** While Float64 is the baseline for accuracy, the framework will transition to **Float32** in Section 7.2.1.2. This is a critical adaptation to overcome GPU memory bottlenecks associated with large 3D graphs.
- **Global attributes:** Although included here, global attributes will be removed in later tests (e.g., Sections 7.2.1, 7.2.2) after revealing they offered no tangible benefit to generalization while contributing to overfitting.

Table 6.3.1: Consolidated setup.

Item	**Setting**	**Justification**
Data type	Float64 (FP64)	*Not tested in this chapter.* Base setting for computational accuracy.
Mini-batch size B	10	Best balance of training speed, memory usage, and stable gradients. Smaller batches (B=5) showed higher variance and more non-convergence (Figure B.2.4).
Learning rate (η_0)	1×10^{-3}	Base rate. Provides a good balance of fast convergence without the instability and high variance of larger rates (Figure B.2.1).
GNN layers (GNN layers)	4	Base model size. While 3-layer models perform well on the small tuning dataset, the full 4-layer capacity is retained for more complex 3D problems (Figure B.2.7).
Hidden dimension (HD)	64	Base model size. Initial tests (Figure B.1.1) show that a smaller dimension (HD32) struggles to converge. While with new updates the model performed well (Figure B.2.7), the larger size was adopted for more complex 3D problems.
Global attributes (GA)	Yes	Included in the base model to provide graph-level features (e.g., matrix size) to the final predictive MLP.
Dropout	No	Tests show dropout does not offer significant performance gains over the baseline (Figure B.2.12) and is thus deemed unnecessary for the full dataset.
Optimizer	AdamW	Standard, robust first-order optimizer with superior decoupled weight decay regularization. A more complex second-order optimizer (L-BFGS) has been tested and has shown no significant benefit (Figure B.2.15).
Scheduler	ReduceOnPlateau (Save)	Provided the most stable training path. Outperformed other schedulers and "No Scheduler" by reloading the best model state (Figure B.2.10).
Pre/Post smoothing steps	2	Standard value in classical AMG, adopted as a constant from the initial setup (Table 6.2.1).
Activation function before MLP	Sigmoid	Initial tests (Figure B.1.2) show this was a critical component for stabilizing training and achieving significantly lower and more consistent losses.

7

AMG-GNN Generalization

With a robust and optimized model established in the previous chapter, the central research questions can now be addressed. This chapter evaluates the model's performance and, most importantly, its ability to generalize to new and more complex problems not seen during training.

To thoroughly test the capabilities and limitations of the AMG-GNN framework, this evaluation is structured as a series of progressively more challenging datasets:

1) **Structured datasets** (Section 7.1). First, validate the model on structured grids (synthetic and unsteady flow data). This serves a dual purpose: it provides a direct benchmark against the data-driven GMG approach from prior literature and tests the model's ability to generalize across different flow physics and problem scales (e.g., training on small, simple meshes and testing on large, unsteady flow data ones).
2) **Unstructured datasets** (Section 7.2). Next, test the model's primary design objective: handling geometric and topological complexity. This section will analyze the model's performance on 2D and 3D unstructured meshes, which represent the core challenge for classical AMG and the main focus of this thesis.
3) **Industry-relevant geometries** (Section 7.3). Finally, assess the model's performance on the AirfRANS dataset. This represents the ultimate test of generalization: applying the model, trained on simple geometries, to complex, practical aerodynamic geometries (NACA airfoils) for which it has no prior knowledge.

Throughout this chapter, performance is measured not only by the final acceleration in solver wall-clock time but also by the model's robustness, as indicated by its convergence rates across this diverse and challenging range of CFD problems.

7.1. Structured dataset

This section is the first milestone for the AMG-GNN model. Previously, the model has been tested on a small dataset on unstructured problems, but generalization and robustness are traits that are desired for the tuned Jacobi AMG. For this reason, the model is checked on several problems based on structured grids that have already been tested by the data-driven GMG [30]. As a result, the performance of the AMG-GNN framework is evaluated on purely structured grids, and the acceleration is compared to the tuned Jacobi GMG model.

To do so, synthetic and unsteady problems are generated on several grids, both in two and three dimensions. Further, while the problems are straightforward to generate with the WaterLily CFD solver, most of them suffer from a lack of diversity in terms of GNN input. That is, the only characteristics that change from problem to problem are the problem size (n, p), the initial guess

vector $\mathbf{u}_0$, and the RHS vector $\mathbf{f}$. The matrix of coefficients does not present any variation from sample to sample. This happens for all 2D/3D synthetic and unsteady problems except for the sphere cases, where each sample comes with a new matrix of coefficients $\mathbf{A}$. This is due to the immersed boundary method used in the sphere problems, which creates an effective geometry based on the sphere's position. Thus, a different distribution of coefficients. Overall, this generates high chances of overfitting to the noise of the RHS vectors. Even though they are not included as input to the GNN, it is learning indirectly from them by tuning the polynomial coefficients to reduce the loss even further.

7.1.1. Synthetic dataset

The training and evaluation follow the original structure from the data-driven GMG paper [30]. First, individual problem types and dimensions are tested, and then mixed together into a single dataset. That is, a two- or three-dimensional static, dipole, and sphere, or a union of these. Moreover, for evaluating the capabilities of the model, two different measures are used: the residual reduction after a V-cycle and the solve time compared to relaxed Jacobi ($\omega = 2/3$) and other smoothers such as Gauss-Seidel and SOR.

Regarding the successive over-relaxation smoothers, they have been chosen to provide more context into the solve time of other smoothers, and the relaxation parameter has been chosen rather arbitrarily; $\omega_1 = 0.75$ and $\omega_2 = 1.25$. This is due to the GS and SOR smoothers being serial implementations, which are at a disadvantage compared to Jacobi smoothers, which can be easily parallelized.

7.1.1.1. Initial tests

Initial experiments have been conducted to assess the model's performance on the new structured dataset. These tests have revealed critical limitations in the preliminary architecture, specifically its inability to converge on three-dimensional sphere cases. Consequently, the model has been updated to the architecture detailed in Section 5.2.1 before it is tested on unseen data.

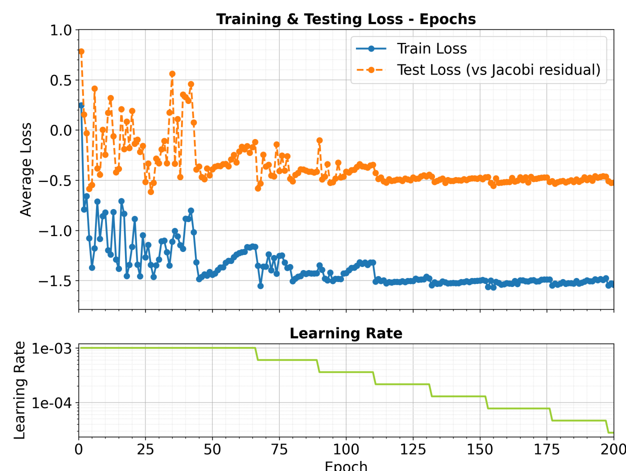


Figure 7.1.1: Loss history for a 2D static dataset ($n = 32$) with the old GNN architecture.

Furthermore, training on a 2D static dataset ($n = 32$) has revealed significant overfitting, as shown in Figure 7.1.1. The test loss reaches a minimum early in the training process before degrading, even as training loss continues to improve. To address this, a checkpointing mechanism

has been implemented to save the "best test model" during the early stages of training, ensuring the optimal state is retained before the model fits the noise of the dataset. This is the model checkpoint that is used for validation throughout the study.

7.1.1.2. AMG-GNN residual reduction

This section compares the residual reduction capabilities of the AMG-GNN model against the data-driven GMG. A distinction regarding the loss metric must be addressed first. While the original work utilizes loss reduction to evaluate the GMG V-cycle performance, this section equates the test loss to the train loss expression only for specific datasets (e.g., 2D static). For the combined dataset, the modified test loss versus relaxed Jacobi is used. This selection ensures the best test model is chosen to maximize performance in the residual reduction comparison.

To ensure a comprehensive comparison, each problem type and dimension is trained separately. Subsequently, a union of all cases is generated and trained to validate the model across all problem types. Each specific dataset consists of a 200/50 train/test split, with sizes of $n = 32$ (32^2) for two-dimensional problems and $n = 16$ (16^3) for three-dimensional ones. The union dataset comprises 75 problems of each type, with a 50/25 split, totaling 300/150 problems.

The training histories for the six separate cases are detailed in Figure C.1.1 in Appendix C.1. The training history for the union dataset is presented in Figure 7.1.2. Notably, training with a richer combined dataset results in stable convergence during the initial epochs (approx. 40), rather than rapidly fitting the data and overfitting. The ReduceOnPlateau scheduler (patience of 20 epochs) indicates that the model fitted the dataset noise around epoch 36, after which train/test losses increase. This subsequent decrease in loss is attributable to the scheduler resetting to the best train model. Furthermore, the train/test losses in this structured dataset are significantly larger than those observed in the initial 2D unstructured problems. This is likely because the model more easily learns relations from the coefficient matrices to accelerate problems when the meshes are identical or highly similar.

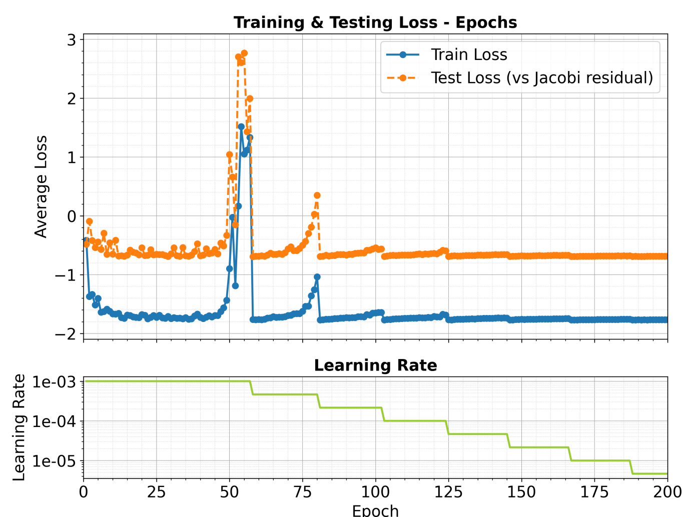
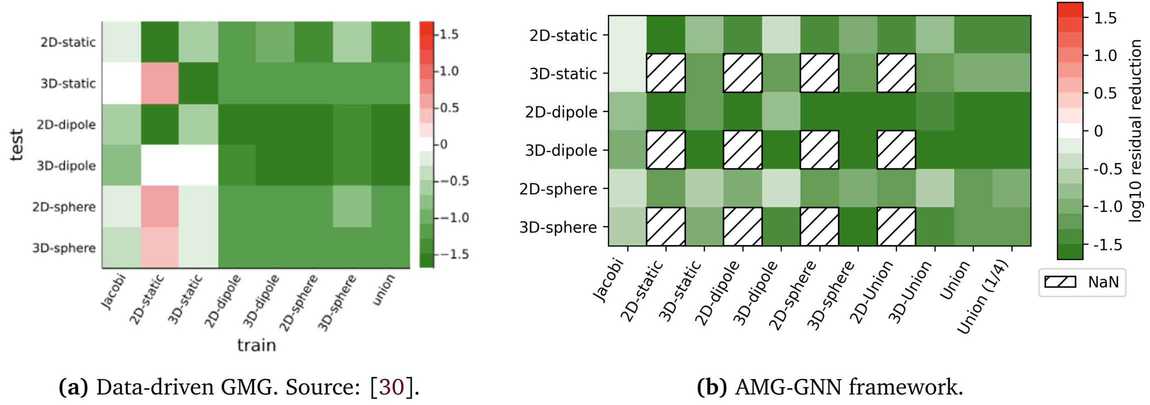


Figure 7.1.2: Loss history for the union dataset with the new GNN architecture.

Following the training of models for different datasets, the results are compared to the original work. The best residual reduction factor for each dataset is obtained using the model with the best test loss, which is crucial for evaluating generalization without overfitting. Figures 7.1.3a and

7.1.3b present the residual reduction factor for each training and validation dataset. The former corresponds to the original data-driven GMG, while the latter represents the tuned AMG approach. The testing datasets consist of 50 problems each, sized 32^2 for 2D and 16^3 for 3D.



(a) Data-driven GMG. Source: [30]. **(b)** AMG-GNN framework.

Figure 7.1.3: Residual reduction factor ($\log_{10}$) for two models. NaNs represent residual divergence. Vertical axis: test dataset. Horizontal axis: train dataset.

Notably, NaNs (divergent V-cycles) occur when a model trained solely on two-dimensional data is tested on three-dimensional problems. This originates from the GCN layer formulation: neighbor node features are aggregated via a weighted average with edge weights and passed through an MLP. In 2D data, node degrees are 3 or 4 (triangular/quadrilateral elements), whereas structured 3D hexahedra have 6 connections. This increased summation significantly modifies the input value distribution seen by the MLP during training, leading to the generation of suboptimal polynomial coefficients and subsequent divergence.

Setting aside the connectivity artifact, the tuned AMG behavior aligns with the data-driven GMG approach. However, the first column, representing the residual reduction of a non-relaxed ($\omega = 1$) Jacobi smoother, differs between the two. This discrepancy arises because, although both methods utilize a multigrid V-cycle, the level hierarchy is different (i.e., distinct $\mathbf{A}^{(l)}$ and number of levels). This is a consequence of the difference in how coarsening and interpolation are performed in GMG compared to AMG. This results in different smoothing effects.

A direct comparison reveals that performance is similar, provided the GNN is not required to generalize from 2D to 3D. Residual reduction is generally large when testing on the same dataset type used for training (e.g., 2D static or 3D dipole). Conversely, the 2D and 3D static cases show poor performance on the data-driven GMG, indicated by divergence (red zones). Among the specific training datasets, the 2D/3D spheres demonstrate the best generalization. This is likely due to the dataset containing diverse matrices rather than a unique one, allowing the model to learn a more robust set of parameters. Even though the dataset utilizes a single matrix $\mathbf{A}^{(0)}$, the GNN receives coarse matrices $\mathbf{A}^{(l)}$ from multiple levels, which aids generalization.

Regarding the union datasets, models are trained on 2D-only, 3D-only, and a full union (2D/3D). The union (1/4) model is trained on the same union dataset but tested on 128^2 and 64^3 problems. It is a $1/4^m$-th scaled model (where m is the spatial dimension). The 2D union suffers from the same 3D generalization issues as the specific datasets, slightly underperforming the 2D static model despite the increased diversity. Similarly, in the 3D union, the 3D sphere dataset outperforms the combined dataset. However, the true union datasets demonstrate strong performance across all problem types, often surpassing specific datasets. Crucially, the residual reduction exceeds that of the Jacobi smoother. The final dataset highlights the GNN's generalization capability: the model performs nearly identically on fine grids as it does on the smaller testing dataset.

To ensure a rigorous comparison, the colormap in Figure 7.1.4 matches the original work. As illustrated, the GNN accelerates certain cases further than the data-driven GMG, achieving a reduction factor of up to −2.213 for the 3D sphere on 3D dipole.

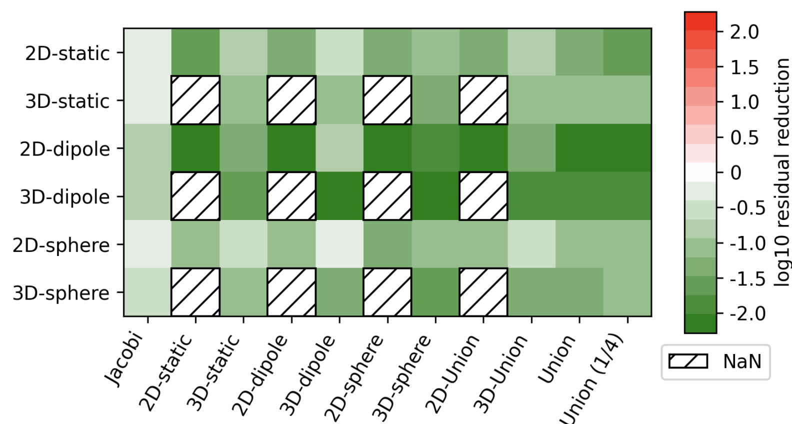


Figure 7.1.4: AMG-GNN residual reduction loss without fixed colormap limits.

The results indicate that the effectiveness of the AMG-GNN is comparable to, and in some cases superior to, the data-driven GMG approach, provided an adequate training dataset is selected. This finding is particularly significant for the unsteady flow cases.

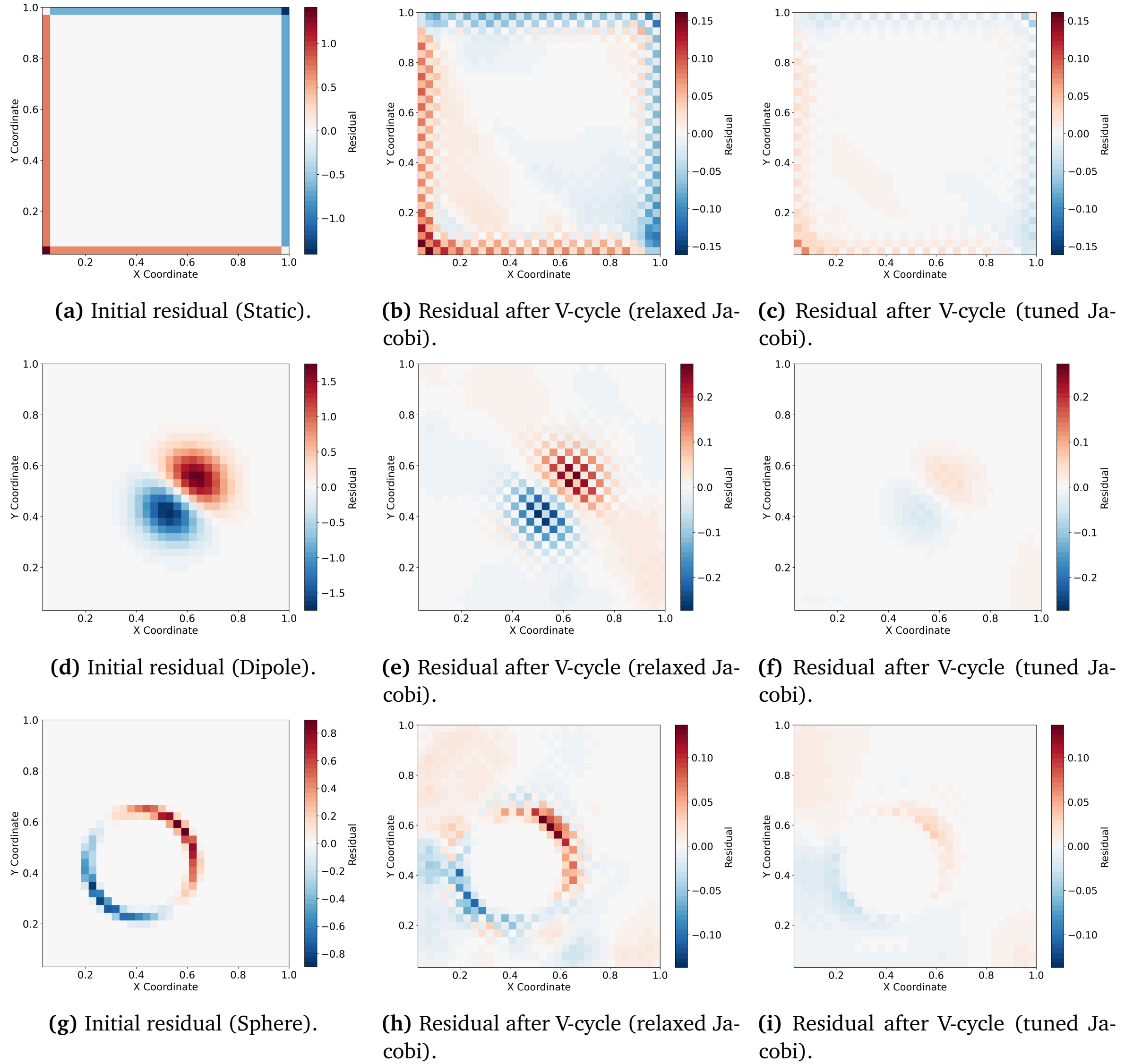
Finally, Figure 7.1.5 visualizes the residual reduction of the relaxed ($\omega = 2/3$) versus tuned Jacobi. The initial residual is compared against the residual after one V-cycle, demonstrating that the AMG-GNN V-cycle is clearly superior in reducing the residual, even against the higher-performance relaxed Jacobi.

7.1.1.3. AMG-GNN solve time

A direct comparison of solve time acceleration with the data-driven GMG [30] is constrained by two factors. First, the different coarsening algorithms between GMG and AMG result in distinct matrix hierarchies, preventing a one-to-one comparison of V-cycles. Second, the implementation frameworks differ (PyTorch vs. Julia). The PyTorch implementation requires wrappers for sequential smoothers (Gauss-Seidel, SOR), introducing significant overhead due to Python's poor performance with inner loops. Consequently, the results presented here contrast the parallelized tuned smoother against sequential benchmarks and the parallelized relaxed Jacobi smoother. Metrics are reported as either total time (setup + solve time) or relative solve time.

To assess the generalization of the union-trained model to structured problems, validations are performed on 2D sphere (128^2) and 3D sphere (64^3) datasets, each containing 50 problems. The results are displayed in Figures 7.1.6 and 7.1.7.

In the two-dimensional case, parallel smoothers significantly outperform sequential ones. Gauss-Seidel is approximately two times slower than the Jacobi variants. Furthermore, suboptimal relaxation parameters for SOR severely degrade performance. Although the tuned AMG achieves a 30% faster solve time with a 98% convergence rate, the total time exceeds that of the relaxed Jacobi. As shown in Figure 7.1.6b, the GNN setup constitutes a major fraction of the total computational cost. The ratio of solve time to GNN setup time is approximately 2.7. This is notably lower than in unstructured datasets (e.g., ratio of 7.53 for $N_c = 156$). Although GNN inference is substantial, a CFD simulation requires solving the linear system across many time steps (or



(a) Initial residual (Static). **(b)** Residual after V-cycle (relaxed Jacobi). **(c)** Residual after V-cycle (tuned Jacobi).

(d) Initial residual (Dipole). **(e)** Residual after V-cycle (relaxed Jacobi). **(f)** Residual after V-cycle (tuned Jacobi).

(g) Initial residual (Sphere). **(h)** Residual after V-cycle (relaxed Jacobi). **(i)** Residual after V-cycle (tuned Jacobi).

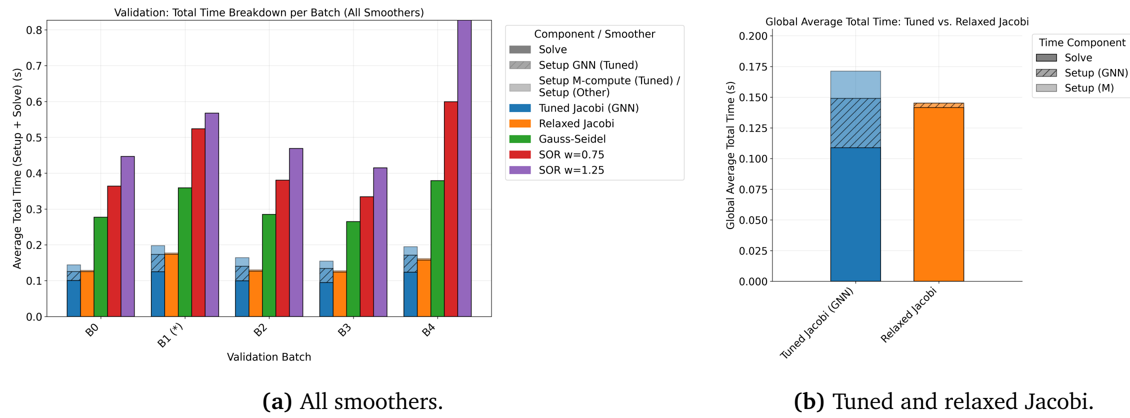
Figure 7.1.5: Residual fields of diverse two-dimensional problems before and after V-cycle for static (a,b,c), dipole (d,e,f), and sphere (g,h,i). Size is 32^2.

steady-state iterations), whereas the setup phase is performed only once. This allows the tuned AMG to amortize its setup cost over the full simulation.

As the problem size increases in three dimensions, the efficiency gap between parallel and sequential operations widens. Figure 7.1.7a shows that sequential smoothers like Gauss-Seidel become over 16 times slower than tuned Jacobi. However, the GNN setup cost also increases disproportionately. The ratio of solve time to GNN setup drops to 0.11. While the pseudo-inverse matrix construction remains efficient, the inference of the 4×64 GNN architecture proves excessive for these problems. With only a 5.2% acceleration in solve time, the solver would require over 172 calls to compensate for the setup cost according to the total time equation:

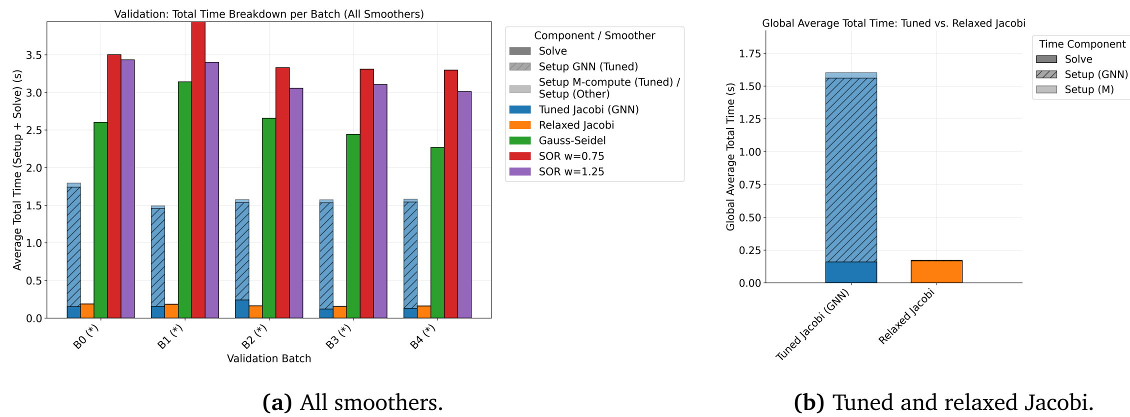
$$\text{Total time} = \text{Setup } \tilde{\mathbf{A}}^{-1} + \text{Setup GNN} + \text{Iterations} \times \text{Cost per iteration}$$

While reducing the GNN size (layers or hidden dimensions) could mitigate this latency, the primary research focus remains on acceleration capability rather than optimizing setup overhead.



(a) All smoothers. **(b)** Tuned and relaxed Jacobi.

Figure 7.1.6: Validation results (total time) for 2D sphere dataset for 50 problems 128^2.



(a) All smoothers. **(b)** Tuned and relaxed Jacobi.

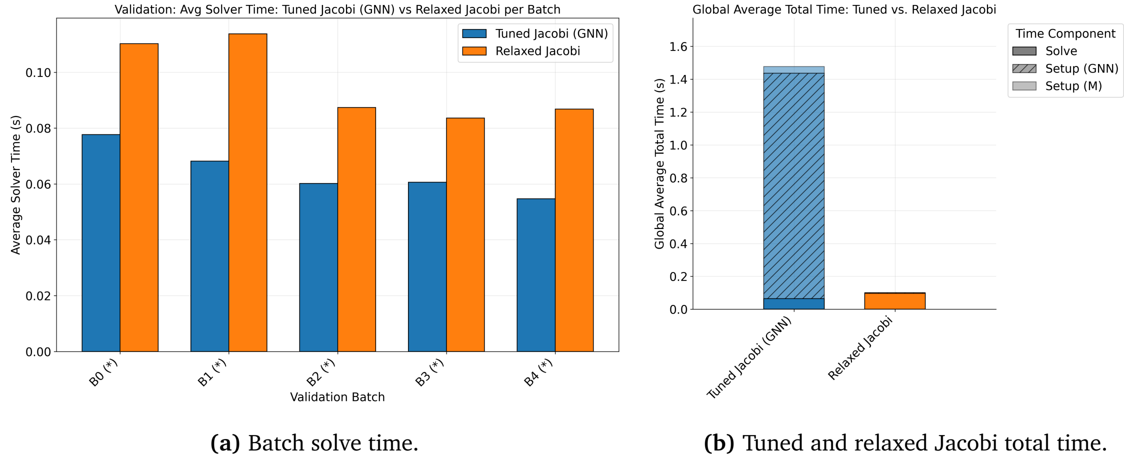
Figure 7.1.7: Validation results (total time) for 3D sphere dataset for 50 problems 64^3.

Hence, the model size is kept.

The impact of convergence tolerance is also investigated. The previous tolerance of 1×10^{-10} is stricter than standard CFD practices with FP64. Relaxing the tolerance to $\delta = 1 \times 10^{-6}$ using a new trained model (Figure 7.1.8) results in a 50% reduction in average solve time compared to relaxed Jacobi. This significant improvement suggests that the best test model selected from the previous union training is not adequate for the 3D sphere dataset under strict tolerances.

Utilizing this improved union model, validation is repeated for all six synthetic cases. Figure 7.1.9 presents the solve times relative to relaxed Jacobi. Dimensional scaling effects are evident: in 2D, sequential smoothers are 2-4 times slower, whereas in 3D, they are up to 16 times slower. Crucially, the tuned AMG demonstrates excellent generalization on meshes up to 64 times larger than the training set (32^2 and 16^3), achieving solve time accelerations between 5% and 45%. Note that dipole cases achieve convergence in 1-2 iterations, adding a lot of variability into the time metrics due to machine uncertainty.

Convergence behavior is detailed in Figure 7.1.10 for 32^3 problems. While the tuned Jacobi consistently requires fewer V-cycles than other smoothers, the increased computational cost per iteration (due to sparse matrix operations) partially offsets the reduction in iteration count. The relaxed Jacobi, Gauss-Seidel, and SOR exhibit similar performance profiles in terms of residual reduction per cycle.



(a) Batch solve time.

(b) Tuned and relaxed Jacobi total time.

Figure 7.1.8: Validation results (new training union) for 3D sphere dataset for 50 problems 64^3 and tolerance reduced to $\delta = 1 \times 10^{-6}$.

7.1.1.4. Multi-V-cycle loss

Testing on structured problems has revealed that the tuned AMG often outperforms relaxed Jacobi when fewer V-cycles are required. This aligns with the original loss function design, which minimized residual reduction over a single V-cycle. Consequently, the model is optimized for the first iteration (based on $\mathbf{A}$, $\mathbf{u_0}$, and $\mathbf{f}$), but performance degrades in subsequent iterations where the input $\mathbf{u_n}$ differs from the training conditions.

To address this, a multi-V-cycle (MVC) loss function is implemented to optimize performance over N_{vc} iterations. The loss is defined as:

$$L = \sum_{n=1}^{N_{vc}} \log\left(\frac{|\mathbf{r}_n|_2}{|\mathbf{r}_{n-1}|_2}\right). \tag{7.1.1}$$

During training, this loss is averaged over the number of V-cycles to prevent gradient explosion. The test metric remains unaveraged but compares the final residual against a relaxed Jacobi baseline:

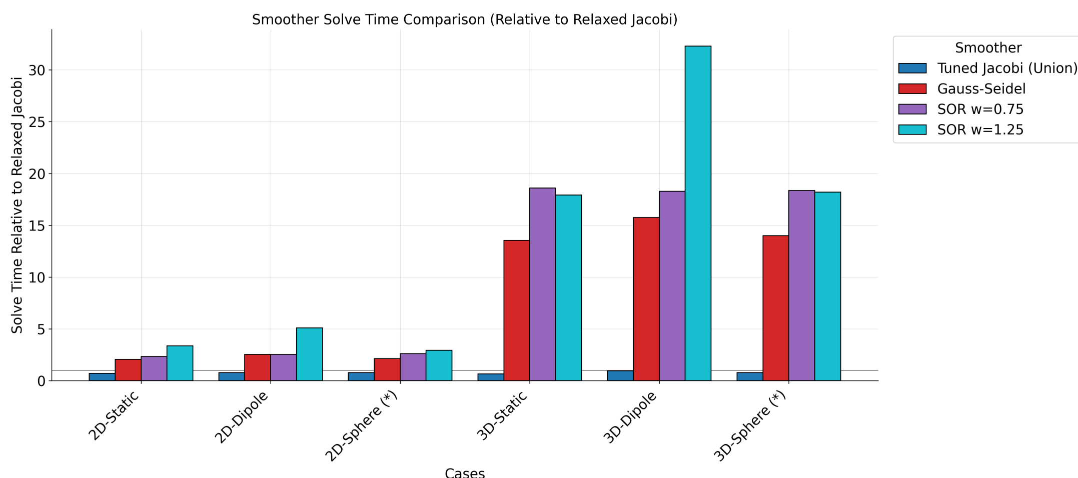
$$L = \log\left(\frac{|\mathbf{r}_{N_{vc}}|_2}{|\mathbf{r}^{\omega-J}_{N_{vc}}|_2}\right). \tag{7.1.2}$$

Training with multiple V-cycles increases computational cost approximately linearly with N_{vc}.

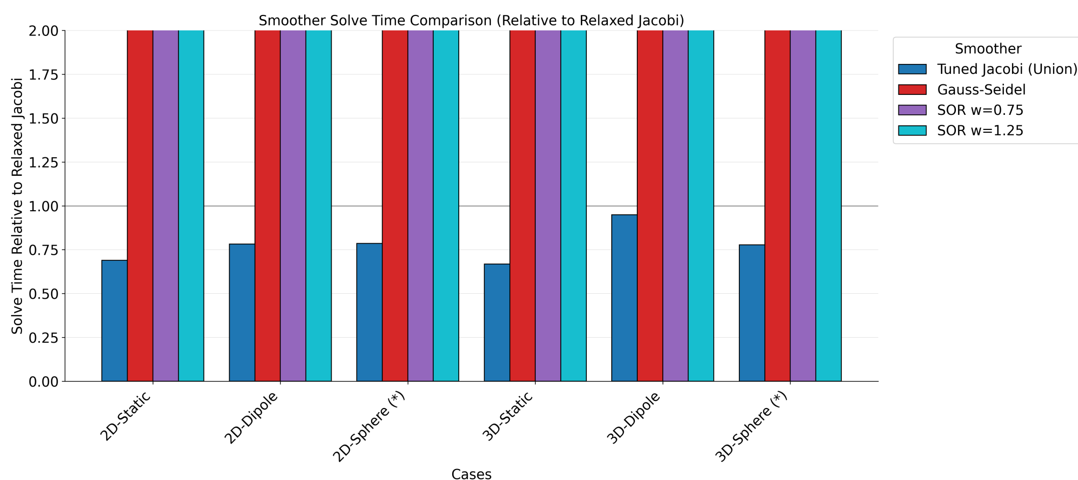
To evaluate this approach, the six synthetic cases are validated against four models trained on the union dataset: the original baseline, a 5-V-cycle model (5VC), a 5VC model with reduced hidden dimension (HD32), and a 5VC model with three GNN layers (GNN3). Results are shown in Figure 7.1.11. Note that increasing training V-cycles does not affect the inference setup cost.

For 2D and 3D dipole datasets, results between the base model and the control show slight variations despite identical setups. This is attributed to the rapid convergence (1-2 iterations) of these cases, where machine precision uncertainty becomes a significant factor in the solve time metric.

In two-dimensional problems, performance is consistent across models, with minor variations likely due to stochastic training effects. In three dimensions (excluding dipoles), the GNN3 model outperforms the others. This performance gain suggests that a simpler model reduces overfitting on unstructured data, where coefficient relations are more easily learned. Furthermore, reducing the GNN depth from four to three layers decreases setup cost by approximately 24%. The 5VC-only

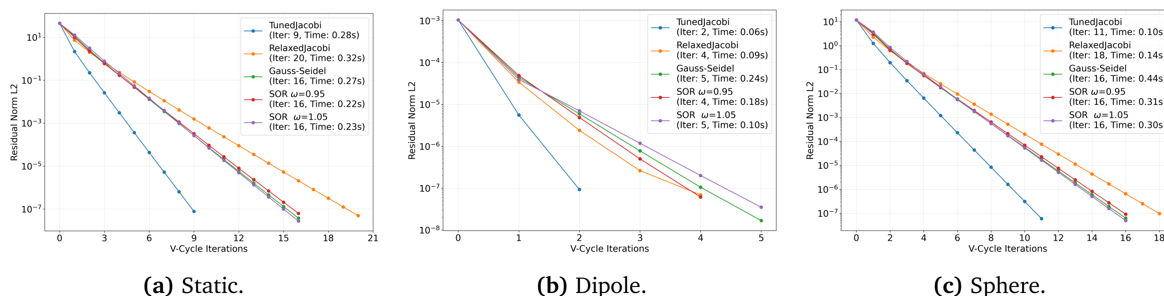


(a) Global view.



(b) Zoomed.

Figure 7.1.9: Solve time for each of the six synthetic cases validated on the new union model.



(a) Static. **(b)** Dipole. **(c)** Sphere.

Figure 7.1.10: Residual norm versus the number of V-cycles for 3D static, dipole, and sphere cases of size 32^3.

model achieved the best convergence rate (92%), comparable to relaxed Jacobi (94%), while others reached 86%. Overall, the significant increase in computational training cost (5 times) yields only marginal performance benefits.

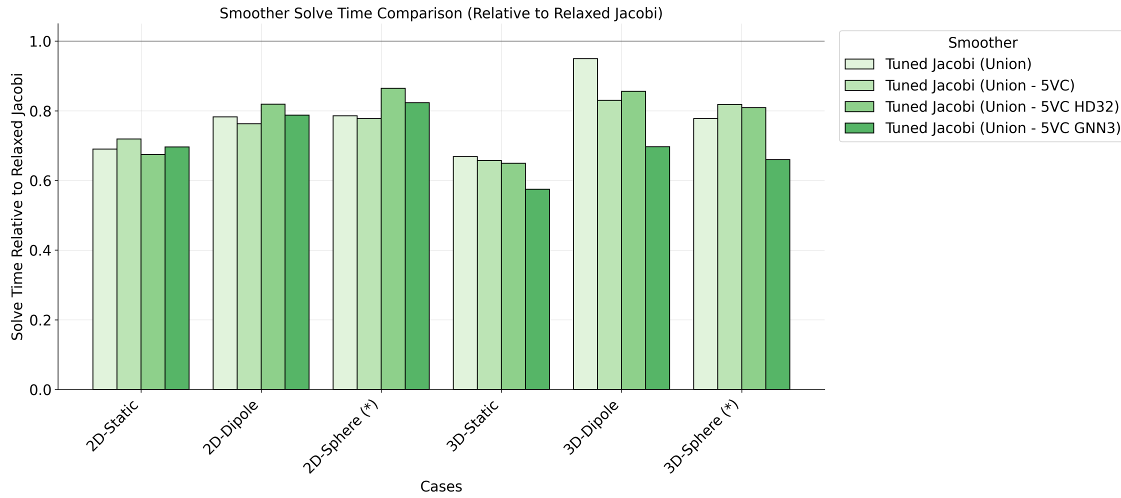
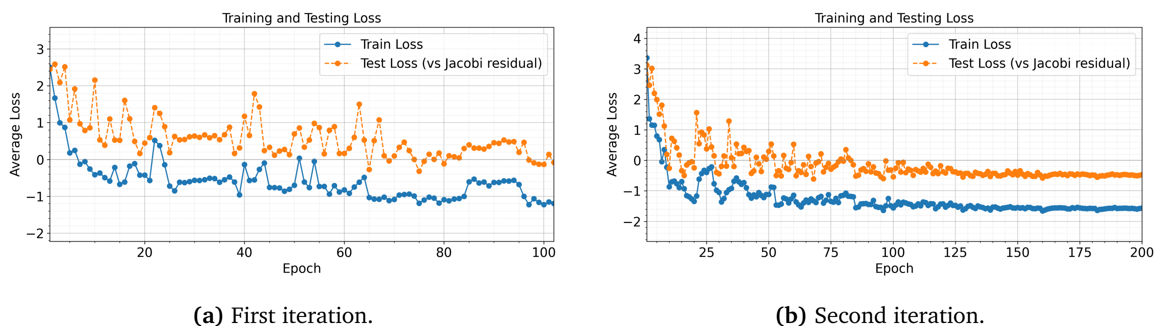


Figure 7.1.11: Solve time for each of the six synthetic cases validated on four different models.

7.1.1.5. Third-degree polynomial

Following the precedent of the data-driven GMG [30], which utilized higher-order pseudo-inverse functions, this section evaluates the effectiveness of a third-degree polynomial for constructing the tuned AMG smoother matrices. While the original GMG approach observed minimal sensitivity in solve time to the polynomial degree, this test aims to determine if the additional complexity yields performance gains within the AMG-GNN framework.

Training histories are presented in Figure 7.1.12. Initial training attempts (Figure 7.1.12a) demonstrate instability, requiring 100 epochs to achieve consistently negative test losses (i.e., outperforming relaxed Jacobi). A subsequent iteration (Figure 7.1.12b) achieves better stability but reveals significant drawbacks. Convergence requires up to 200 epochs, considerably more than the default second-degree polynomial. Furthermore, the final converged loss values for both training and testing are higher than those achieved with the second-degree model, indicating a degradation in performance despite the increased complexity.



(a) First iteration.

(b) Second iteration.

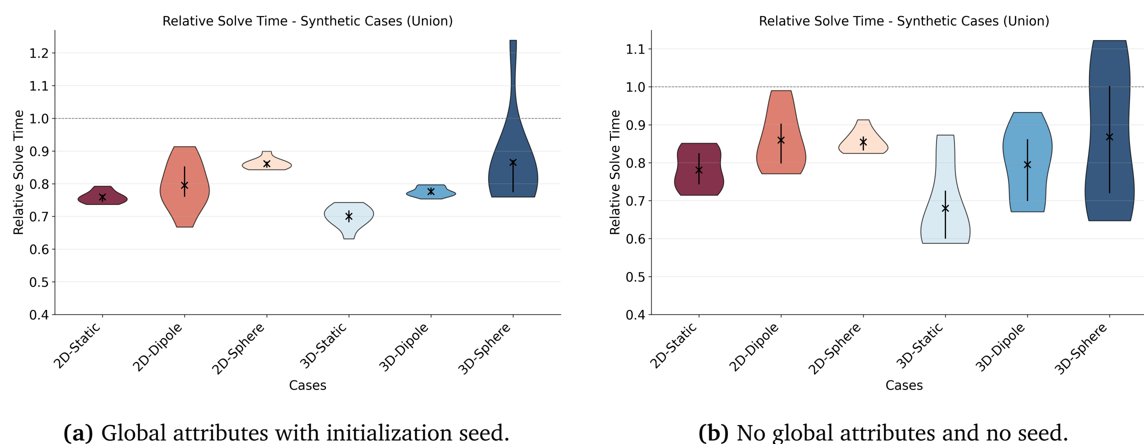
Figure 7.1.12: Loss history for third-degree polynomial functions for $\tilde{\mathbf{A}}^{-1}$.

Validation results for the converged model show a complete lack of generalization, with all test problems across datasets resulting in divergence. This failure suggests that the higher-degree polynomial induces overfitting to the training data, analogous to the high variance observed in cubic versus quadratic regression. These results confirm that the simpler second-degree polynomial provides the optimal balance of expressiveness and generalization, rendering higher-order

polynomials non-viable for this framework. Although potentially interesting, training on simpler functions like first-degree polynomials has not been considered.

7.1.1.6. Effect of stochastic training

Access to high-performance computing resources facilitates the training of multiple model batches to quantify the impact of stochasticity on evaluation performance. Two batches of 10 models have been trained to isolate these effects. The first batch utilizes a fixed GNN initialization seed and includes global attributes, restricting variability solely to the shuffling of training mini-batches. The second batch removes both the initialization seed and global attributes. Note that the exclusion of global attributes has a minimal impact on outcomes, as proven by subsequent sections.



(a) Global attributes with initialization seed.

(b) No global attributes and no seed.

Figure 7.1.13: Relative solve time of synthetic cases for ten trained models.

Figure 7.1.13 presents the relative solve time distributions using violin plots[1] to visualize probability density. In the fixed-seed batch (Figure 7.1.13a), variability is generally constrained to a $5-10\%$ range. Exceptions are observed in the 2D dipole and 3D sphere cases. Accordingly, the 3D sphere distribution reveals a distinct outlier requiring approximately 20% more time than relaxed Jacobi, while the remaining models achieved faster convergence.

Under full stochastic training (Figure 7.1.13b), the performance distribution widens, with minimum variance increasing to around 10%. The most significant deviations occur in the 3D dipole and 3D sphere cases. The variance in the dipole cases is attributed to machine precision uncertainty arising from extremely low iteration counts $(1-2)$. Conversely, the 3D sphere results indicate that approximately 75% of the trained models outperform relaxed Jacobi. While the performance range expands, instances of improved acceleration (compared to the fixed-seed batch) are observed, particularly in the 2D and 3D sphere cases.

These results quantify the inherent variability in solve time performance. They confirm that the single-model results presented in previous sections are representative samples that fall within these established distributions, thereby validating the general performance trends despite the stochastic nature of the training process.

7.1.2. Unsteady cases

The structured dataset includes unsteady flow problems, which offer a more representative evaluation of CFD performance than synthetic cases. A key distinction is the initialization: unsteady

[1] The cross mark is the mean of the set, and the thick line symbolizes the interquartile range, from the first percentile (Q1) to the third (Q3).

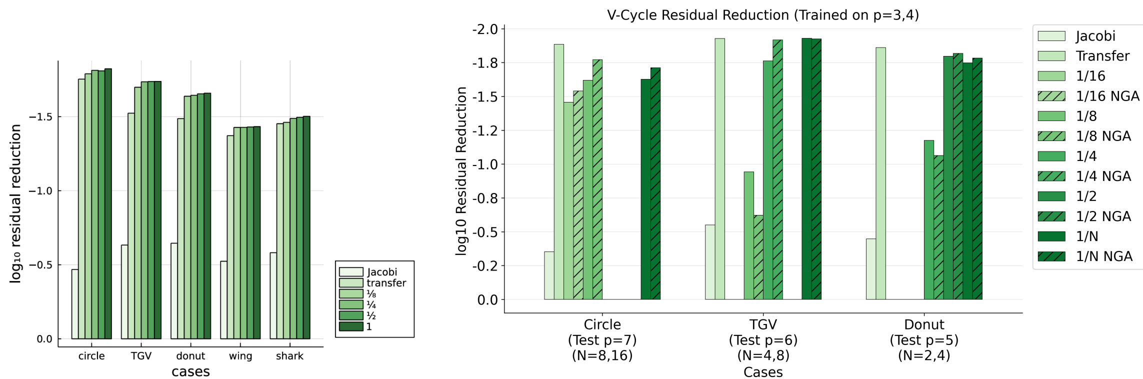
flow simulations utilize the solution from the previous time step as the initial guess ($\mathbf{u}_0$), rather than a zero vector. This provides a realistic test of the tuned AMG solver within a time-stepping loop.

To evaluate generalization, models trained exclusively on synthetic data ("transfer" models) are also tested on these unsteady problems. Overfitting remains a significant concern due to the lack of variety in the unsteady training set, where coefficient matrices $\mathbf{A}$ remain constant for a given mesh and problem, severely limiting the diversity of training examples. Hence, the previous early-stopping implementation is a crucial component when training with these datasets. Consequently, non-overfit models are used for evaluation.

7.1.2.1. AMG-GNN residual reduction

This section assesses the AMG-GNN's ability to reduce residuals in unsteady flow cases. Training is conducted on problem sizes $p = 3$ and $p = 4$ (100/50 split) and combined datasets $p = [3, 4]$ (150/100 split). Recall that p is the parameter that represents the grid size. The mesh size is $2^{p+2} \times 2^{p+1}$ for circle, $2^{p+1} \times 2^p \times 2^p$ for donut, and $2^p \times 2^p \times 2^p$ for TGV. Loss histories (see Appendix C.2, Figure C.2.1) confirm that the limited dataset diversity leads to rapid overfitting, although performance gains over relaxed Jacobi are still observed. To mitigate this, models are also trained without global attributes (NGA), as the limited range of matrix sizes and diagonal dominance factors in the training set could negatively impact generalization on different evaluation cases. Avoiding these global attributes forces the GNN to rely solely on local graph structures, preventing it from overfitting to the specific problem dimensions seen during training.

Figure 7.1.14 compares the residual reduction of the AMG-GNN framework against the data-driven GMG [30]. The notation $1/N$ indicates a model trained on size $p_t = p - \log_2(N)$, where p is the evaluation size. For example, if evaluation is at $p = 7$, a $1/8$ model was trained on $p = 4$. The comparison includes models trained on specific unsteady cases and the synthetic transfer model. Note that the GMG method evaluates on $p = 7$ for all cases, whereas GNN memory constraints for inference limited the evaluation size for TGV and donut cases.



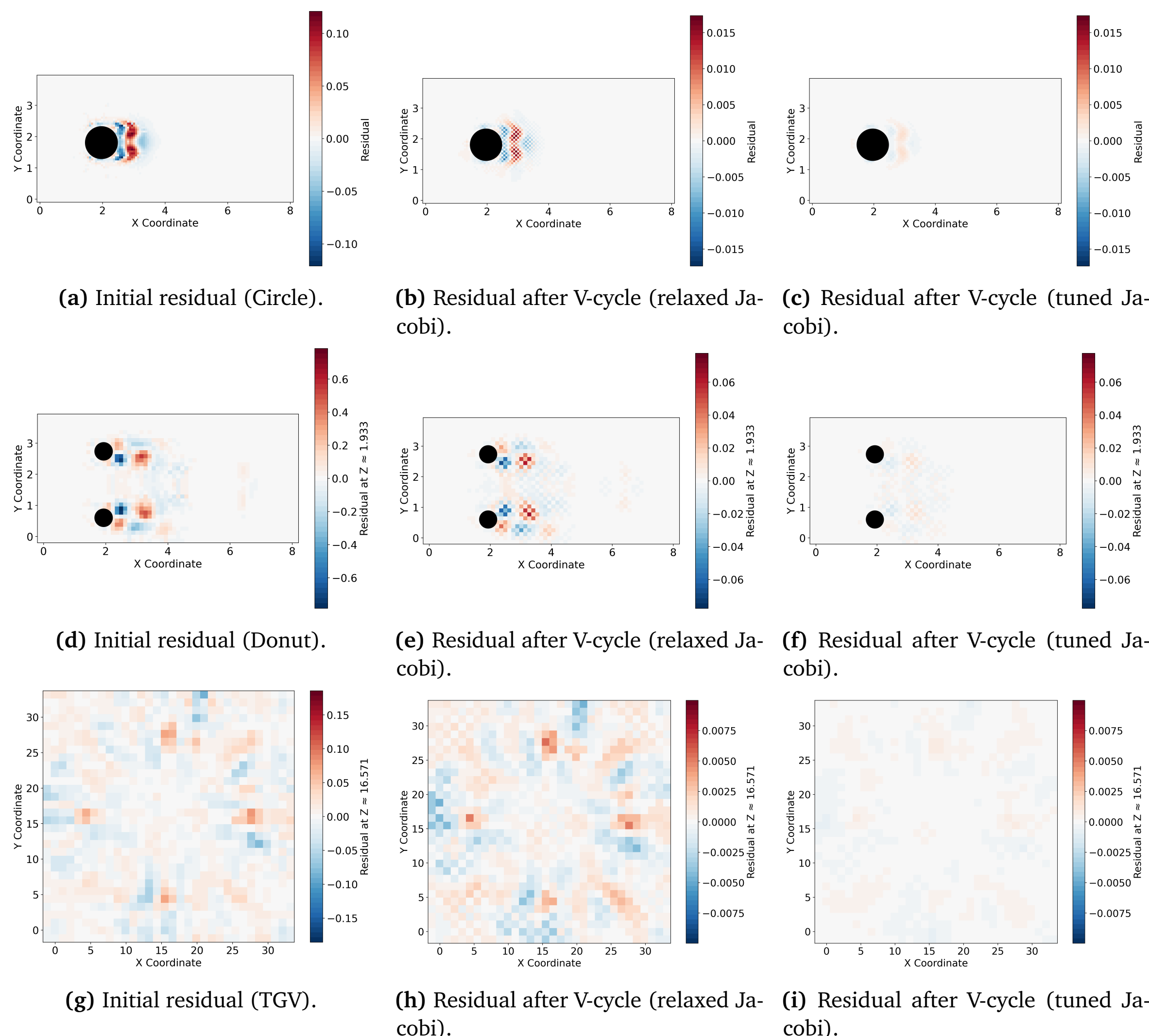
(a) Data-driven GMG. Source: [30].

(b) AMG-GNN (50 problems per case). Cases written as "$1/N$" correspond to combined training of $p = [N_1, N_2]$, where N_1, N_2 are displayed under the problem name. NGA refers to no global attributes.

Figure 7.1.14: Residual reduction for finest circle, donut, and TGV cases for two different solvers.

Results indicate that the transfer model (trained on synthetic union data) consistently achieves the best performance, matching or exceeding the residual reductions of the GMG method. This suggests that the GNN generalizes better when trained on the richer, more diverse synthetic dataset. As expected, training on larger grid sizes ($p = 4$) yields better performance than $p = 3$,

particularly for 3D cases where the coarser mesh ($p = 3$) provides insufficient context. Combined training ($p = [3, 4]$) performs similarly to or slightly worse than $p = 4$ alone.



(a) Initial residual (Circle).

(b) Residual after V-cycle (relaxed Jacobi).

(c) Residual after V-cycle (tuned Jacobi).

(d) Initial residual (Donut).

(e) Residual after V-cycle (relaxed Jacobi).

(f) Residual after V-cycle (tuned Jacobi).

(g) Initial residual (TGV).

(h) Residual after V-cycle (relaxed Jacobi).

(i) Residual after V-cycle (tuned Jacobi).

Figure 7.1.15: Residual fields of diverse two-dimensional problems before and after V-cycle for circle (a,b,c), donut (d,e,f), and TGV (g,h,i). Size is $p = 5$.

Removing global attributes generally improves performance, confirming that the limited training distribution of these attributes hinders generalization. Despite the limited training data, the AMG-GNN demonstrates adequate performance compared to the data-driven GMG, with the transfer model highlighting the benefits of dataset diversity. Figure 7.1.15 visualizes the positive impact of the AMG-GNN V-cycle on residual reduction for $p = 5$ cases.

7.1.2.2. AMG-GNN solve time

Solve time is a critical metric, as a reduction in iteration count does not always translate to wall-clock acceleration due to the increased cost per iteration of the GNN-tuned smoother. This section evaluates the solve time performance across different configurations, mirroring the synthetic data analysis.

Figure 7.1.16 presents the relative solve times (versus relaxed Jacobi) for the initial evaluation batch. A significant finding is the poor generalization of models trained on coarse 3D grids

($p = 3$), specifically for the donut and TGV cases (8^3 for TGV and 16×8^2 for donut), where solve times exceed twice that of the baseline. Incorporating finer meshes into the training set ($1/N$ models) effectively mitigates this issue. Remarkably, the transfer model demonstrates excellent performance on circle and donut cases, though it matches the baseline relaxed Jacobi for TGV. Overall, robust models achieved approximately 20% acceleration with full convergence ratios, and removing global attributes generally improved solve times.

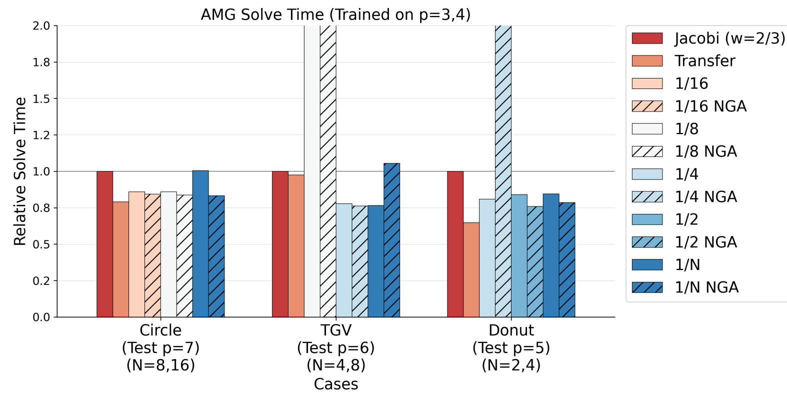
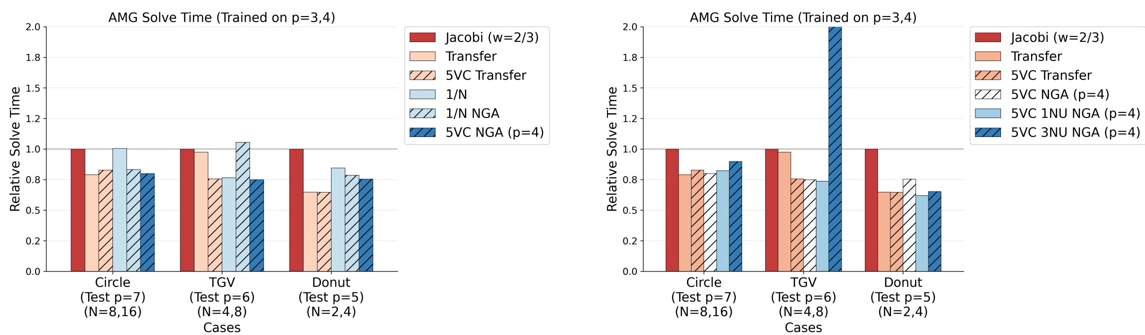


Figure 7.1.16: Relative solve time to relaxed Jacobi for each of the three unsteady cases using basic setups with and without global attributes.

Further tuning via a multi-V-cycle (MVC) loss function is investigated. Figure 7.1.17a compares four models trained with a 5-V-cycle loss. The MVC approach significantly benefits the TGV case, allowing the transfer model to surpass its 1-cycle counterpart. This improvement is likely driven by the transitional nature of the TGV flow. Unlike the circle and donut cases, which remain within a consistent flow regime, the TGV evolves from a laminar state to fully developed turbulence before decaying back to laminar. This rapid temporal evolution means that the solution from the previous time step (used as an initial guess) differs significantly from the current field, necessitating the more robust V-cycles provided by MVC training.



(a) Diverse multi-V-cycle train models.

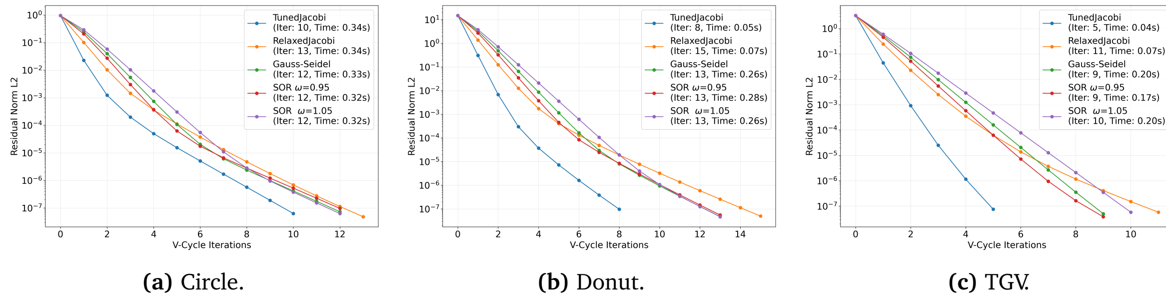
(b) Modifying the (pre and post) smoothing steps.

Figure 7.1.17: Relative solve time to relaxed Jacobi for each of the three unsteady cases using different train models.

The impact of the number of smoothing steps (ν) is also analyzed (Figure 7.1.17b). Increasing ν to 3 degraded efficiency, particularly for TGV. This is due to the higher computational cost per smoothing step of tuned AMG. The large solve time jump for TGV is possibly a result of the trained model failing to generalize, rather than a smoothing step issue. Conversely, reducing to $\nu = 1$ maintains or improves performance (e.g., for the donut case). This proves that the tuned Jacobi

smoother is more powerful in reducing high-frequency errors in a single step than the relaxed Jacobi. Nonetheless, absolute solve times degrade from $\nu = 2$ to $\nu = 1$, showing that selecting a configuration based on relative performance is not always the optimal choice in terms of solver efficiency.

Convergence behavior is detailed in Figure 7.1.18 for $p = 5$ cases. The tuned Jacobi exhibits a steeper initial residual reduction than other smoothers. This behavior aligns with the training objective, which optimizes reduction for the first V-cycle based on the initial guess $\mathbf{u}_0$. This characteristic is highly advantageous for CFD applications, where convergence targets are typically in the range of 10^{-3} to 10^{-4}, a region where the AMG-GNN clearly outperforms standard smoothers.



(a) Circle. **(b)** Donut. **(c)** TGV.

Figure 7.1.18: Residual norm versus the number of V-cycles for circle, donut, and TGV cases of size $p = 5$. Results are obtained using the transfer union.

Finally, Figure 7.1.19 summarizes the effect of varying the number of training V-cycles. While circle and donut cases show no benefit (or even degradation) from increased V-cycles, the TGV case is optimized with 3 − 5 V-cycles. Given the proportional increase in training cost, MVC is only recommended for problems with significant temporal evolution like TGV.

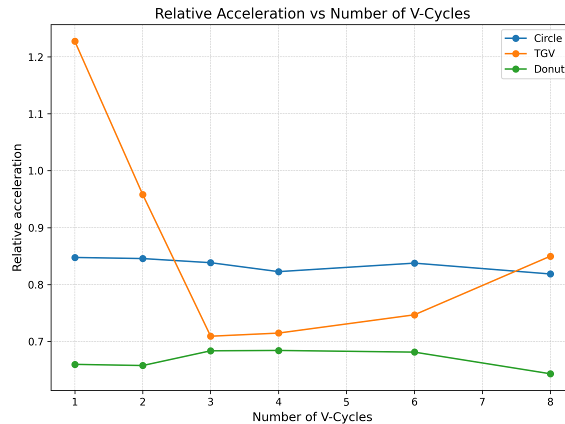
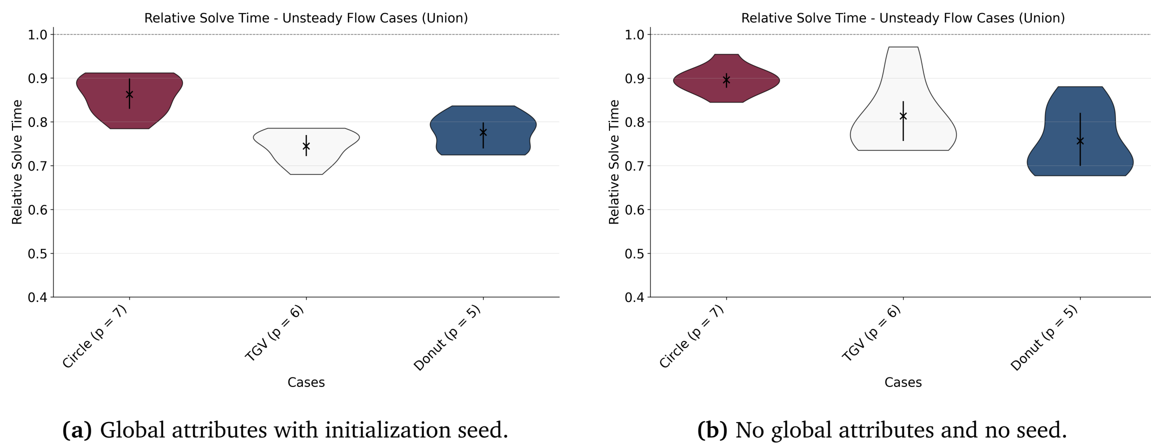


Figure 7.1.19: Relative solve time to relaxed Jacobi for each of the three unsteady cases based on models trained on different numbers of V-cycles.

7.1.2.3. Effect of stochastic training

To evaluate the impact of stochastic training on unsteady flow datasets, multiple model batches are trained and evaluated on the circle, TGV, and donut cases. Two configurations are tested: one utilizing global attributes with a fixed initialization seed of GNN parameters, and another excluding global attributes and randomizing the GNN initialization.

Figure 7.1.20 displays the relative solve time distributions. Compared to the synthetic datasets, the unsteady cases exhibit lower sensitivity to setup variations. However, fixed initialization generally provides a more favorable starting point. Removing the seed increased the range of relative solve times, typically leading to performance degradation in the circle and TGV cases. Conversely, the donut case benefits from randomized initialization, suggesting that the fixed weights are suboptimal for this specific system.



(a) Global attributes with initialization seed.

(b) No global attributes and no seed.

Figure 7.1.20: Relative solve time as a function of the test loss for the three unsteady cases. Using synthetic transfer models.

Despite the variability, the model demonstrates significant generalization capability. Notably, the circle case is evaluated on a 512×256 mesh while trained on only 32×32. Even with this 128-fold increase in cell count, the model achieves up to 25% acceleration over the relaxed Jacobi baseline, indicating potential for further gains with optimized GNN architectures.

7.1.3. Key findings

The evaluation across unsteady flows (circle, TGV, donut) confirms the effectiveness of the tuned AMG, particularly its ability to handle complex problems starting from a robust initial guess ($\mathbf{u}_0$ from the previous timestep). The following key conclusions are drawn from the unsteady dataset analysis:

- **Great generalization (the transfer model)**: The model trained exclusively on the synthetic union dataset consistently matches or outperforms models specifically trained on the unsteady flow data (Figures 7.1.16, 7.1.17). This finding is crucial, suggesting the GNN successfully learns the generalized algebraic features of the Poisson system rather than overfitting to specific flow solutions, which is vital for real-world applicability.
- **TGV Sensitivity to Multi-V-Cycle Loss**: The TGV problem, characterized by a more rapidly evolving velocity field, uniquely benefits from being trained with a multi-V-cycle (MVC) loss function (Figure 7.1.19). This shows that while a single V-cycle loss is sufficient for most problems, transitional cases may benefit from a loss function that accounts for performance across multiple iterations.
- **Smoother complexity limit**: Tests on polynomial degree (Section 7.1.1.5) reveal that a second-degree polynomial is optimal. Increasing to a third-degree polynomial results in severe overfitting and divergence, confirming that a simpler, constrained smoother matrix provides the best balance of expressiveness and stability.

- **Stochasticity and model selection**: Training outcomes exhibit significant variance due to the stochastic nature of mini-batch shuffling and initialization (Figures 7.1.13, 7.1.20). Consequently, training a single instance is insufficient. A robust methodology requires training a batch of models and selecting the optimal candidate based on validation performance.

7.2. Unstructured dataset

This section presents the key workload that the AMG-GNN has to be evaluated on. Before, the model has been trained and tested thoroughly on structured data, and slightly on a small portion of the actual unstructured data to optimize the model (Chapter 6). During these sections, the model has proven robust and generalizable within the datasets that have been used. Nonetheless, one of the critical research objectives is to demonstrate that this model can do that on a variety of two- and three-dimensional unstructured problems, which is presented in detail in the next part of the document.

To accomplish this, the performance of the tuned Jacobi smoother is evaluated on unstructured grids generated from two- and three-dimensional problems in ReFRESCO, after training on them. In contrast to the structured dataset, the data generated from ReFRESCO does not suffer from a lack of variety, and many different sets of $\mathbf{A}, \mathbf{u}_0$ and $\mathbf{f}$ are available for training and evaluation. Therefore, the chances of suffering from overfitting are lower. Furthermore, 2D and 3D cases differ a little. The 2D problems use all types of meshes presented during the methodology, while 3D problems use structured, triangle Delaunay, and quad-dominant meshes. This is to avoid excessively large matrices for the dataset, as the other 2D mesh types generate more cells. Finally, the workload in this section is executed on a high-performance workstation (details in Appendix A.1). This hardware enables the training of larger datasets and model batches (typically 5 models per configuration), allowing for the quantification of the stochastic variance inherent in GNN training.

7.2.1. Two-dimensional problems

The evaluation on unstructured grids requires a robust methodology to account for the increased complexity of the dataset.

A primary objective in this section is to validate the feasibility of single precision (FP32) training. While double precision (FP64) offers higher numerical stability, FP32 significantly reduces memory footprint, a critical bottleneck for GNNs on large 3D meshes, and accelerates inference. Unlike the structured WaterLily dataset, where extremely small coefficients ($\sim 10^{-8}$) led to singular coarse-grid operators in FP32, the ReFRESCO unstructured matrices are robust enough to support lower precision without corruption.

The baseline training configuration for the 2D analysis is detailed in Table 7.2.1. Based on the findings from the unsteady flow problems (Section 7.1.2), global attributes have been removed from the configuration baseline, as they provided no quantifiable benefit. For the solver evaluation, the AMG convergence tolerance is initially set to $\delta = 1 \times 10^{-6}$ with a maximum of 250 iterations, although this is relaxed to $\delta = 1 \times 10^{-4}$ in later FP32 experiments to align with practical CFD standards.

7.2.1.1. Impact of data diversity

To determine the degree of generalization capability inherent to the GNN, the influence of dataset diversity is rigorously evaluated. The optimization strategy initially assesses a model trained on a single problem type before transitioning to a combined dataset.

Table 7.2.1: Initial training setup for two-dimensional datasets.

Item	Setting
Data type	Float64 (FP64)
Train / test problems	600 / 100
Mini-batch size B	10
Problem size N_c	$10 \leq N_c \leq 28$
Learning rate (η_0)	1×10^{-3}
GNN layers (GNN layers)	4
Hidden dimension (HD)	64
Global attributes (GA)	No
Dropout	No
Optimizer	AdamW
Scheduler	ReduceOnPlateau (Save)
Pre/Post smoothing steps	2
Activation function before MLP	Sigmoid

First, a baseline model is trained exclusively on Poiseuille flow problems (100/20 split for $10 \leq N_c \leq 28$). As detailed in Appendix D.1 (see Figure D.1.3), this model fails to generalize effectively. When evaluated on larger meshes ($N_c = 79, 111$), it exhibits high variance and frequent non-convergence, with more than half of the models underperforming the relaxed Jacobi baseline. This indicates that training on a single problem type limits the model's ability to learn robust algebraic features, leading to overfitting on specific flow topologies.

Accordingly, shifting the training strategy to a union dataset comprising Poiseuille, Channel, ConvDiff, and Plate flows (600/100 split) increases diversity and yields significant improvements. As shown in Figure 7.2.1, the union-trained model provides consistent acceleration across all problem types.

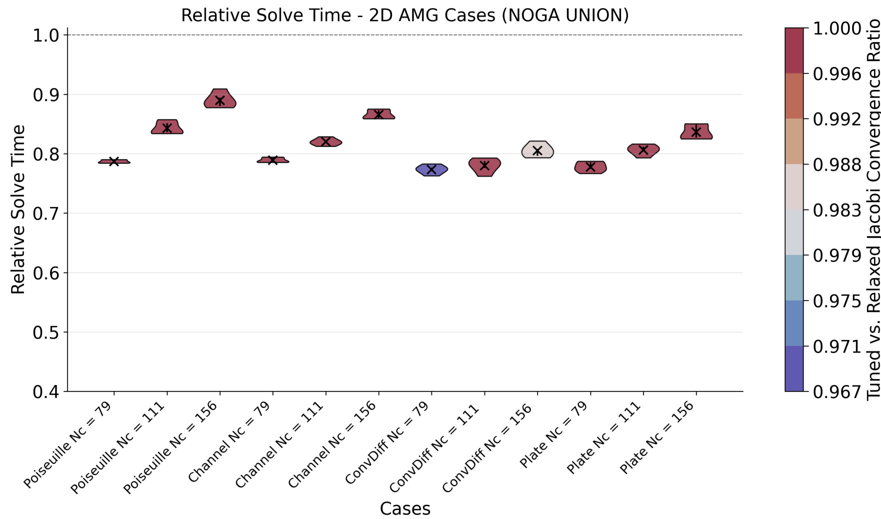


Figure 7.2.1: Relative solve time to relaxed Jacobi for 5 models trained on the union dataset. Evaluated on datasets of size $N_c = 79, 111$ and 159.

The model achieves average solve time accelerations of 28% for $N_c = 79$, 21% for $N_c = 111$, and 17% for $N_c = 159$ on average. Crucially, the convergence consistency matches that of the robust relaxed Jacobi smoother. The decay in performance on the largest meshes ($N_c = 159$)

suggests that while the model generalizes well, its learned mapping is less optimal for graphs significantly larger than those in its training set ($N_c \leq 28$). Nevertheless, the clear contrast with the Poiseuille-only results confirms that mesh and physics diversity, rather than dataset size alone, is the primary driver of generalization.

Finally, the impact of including global attributes (GA) in the GNN architecture is evaluated. Comparative tests between models with and without GAs (detailed in Appendix D.1, Figure D.1.4) reveal virtually identical performance in terms of solve time and convergence ratios. Since the inclusion of global attributes has added architectural complexity without providing evident benefits in generalization or acceleration, they have been definitely removed from the subsequent tests to streamline the model.

7.2.1.2. Impact of precision and solver tolerance

A key limitation to applying GNNs to large-scale 3D CFD problems is memory consumption. To address this bottleneck, the framework is switched from float64 (FP64) to float32 (FP32), and the solver convergence tolerance is adjusted from 1×10^{-6} used in initial testing to a standard CFD practice of $\delta = 1 \times 10^{-4}$.

The performance impact of these changes is evaluated by relaxing the tolerance and subsequently retraining and validating the entire framework in native FP32. The results for the FP32-native configuration are presented in Figure 7.2.2.

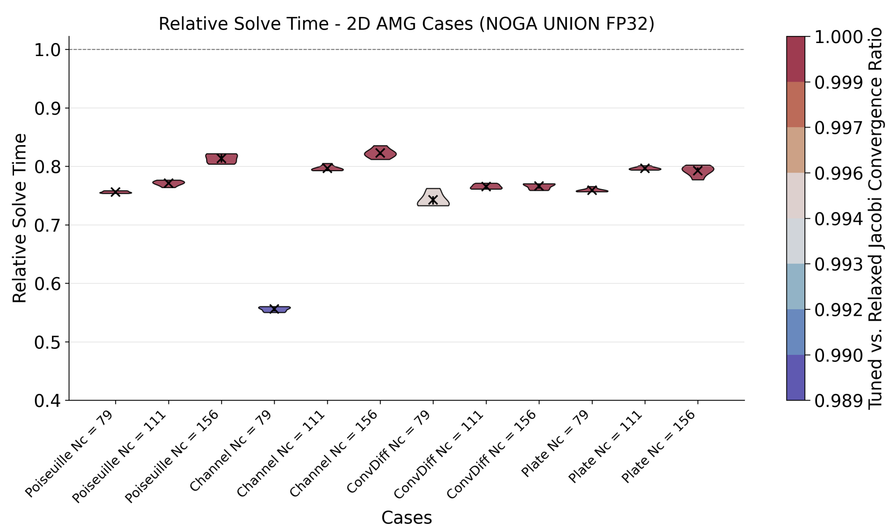


Figure 7.2.2: Relative solve time to relaxed Jacobi for 5 models trained on the union dataset and float32. Evaluated on datasets of size $N_c = 79, 111$ and 159 and $\delta = 10^{-4}$.

As shown, the FP32-native model improves the performance of the FP64 baseline, achieving further acceleration while slightly reducing the variance in solve times. Essentially, moving to single precision halves the memory footprint of the GNN inference and V-cycle operations while providing faster solve times. This memory reduction is the primary mechanism for training on 3D meshes, where GPU VRAM is the limiting factor.

Reducing the solver tolerance to 10^{-4} also improves the relative acceleration of the GNN-tuned smoother. Since the tuned smoother decreases residual norm most aggressively in the initial V-cycles (as seen in Figure 7.1.18), relaxing the tolerance allows the solver to converge earlier. This is observed especially for the channel flow ($N_c = 79$), with ∼70% acceleration in solve time.

Another positive aspect of switching from FP64 to FP32 is the faster GPU computations. Given that the GNN inference is compute-bound, the GNN setup speed is doubled, whereas the V-cycle only sees a 0-10% improvement due to the memory-bound nature of sparse matrices. This can alleviate the burden of long GNN setup times for large problems.

7.2.1.3. Sensitivity analysis

Following the establishment of the optimal FP32-union baseline, a comprehensive sensitivity analysis is conducted to determine if further gains could be achieved through advanced hyperparameter tuning allowed by the high-performance workstation. This investigation focuses on three primary variables: mini-batch size, the number of training V-cycles, and the scale of the training dataset.

Tests are conducted with mini-batch sizes of $B = 1, 5$, and 10 and larger dataset $N_c \leq 40$ (see Appendix D.2, Figure D.2.1). Results indicate that small batches ($B = 1$) lead to overfitting and poor generalization, as the model learns to fit the noise of individual problems (i.e., $\mathbf{f}$ vector) rather than the general algebraic structure. Increasing the batch size to $B = 5$ and $B = 10$ significantly improves convergence rates (versus relaxed Jacobi) and solve time performance. Specifically, the larger batch size ($B = 10$) combined with a richer dataset (including $N_c = 40$) produces the most robust validation results, minimizing the variance in solve time acceleration. Nonetheless, increasing the size of the grids to $N_c = 40$ does not have a noticeable effect on average solve time performance. Further, the results suggest that the best choice of mini-batch size lies around $B = 5$. While variance is larger than $B = 10$, average solve times are nearly identical. This is beneficial for the memory constraints.

To potentially improve robustness on complex flows, models are trained using a loss function accumulated over MVC ($N_{vc} = 1, \ldots, 8$). As shown in Appendix D.3 (Figure D.3.1), increasing the number of training V-cycles does not yield a consistent performance advantage. In fact, for cases like convection-diffusion, higher N_{vc} counts degrade convergence rates. The single V-cycle approach ($N_{vc} = 1$) consistently provides the best balance of training efficiency and solve-time acceleration, confirming that optimizing one V-cycle is sufficient for the GNN smoother for this set of problems in unstructured grids.

Finally, the impact of extending the training dataset to include larger meshes (up to $N_c = 56$) is evaluated, as shown in Figure D.4.2. The mini-batch size was reduced to $B = 6$ to allow larger grids. This reduces the training loss (see Figure D.4.1), and it results in a slight performance degradation on the target validation meshes ($N_c \geq 79$) compared to the closest configuration (FP32 $B = 10$, Figure 7.2.2). This slight performance degradation is likely primarily driven by the reduced mini-batch size ($B = 6$), which can lead to noisier gradient estimates and less stable convergence compared to the baseline ($B = 10$). However, since the average solve times are generally maintained with changes in mini-batch size between $B = 5 - 10$ (Figures D.2.1b and D.2.1c), this suggests a trade-off in the GNN's learning capacity. By forcing the network to learn a mapping across a wider range of scales ($10 \leq N_c \leq 56$), the model may reach a solution that is less optimal for extrapolation than the mapping learned from a smaller distribution ($10 \leq N_c \leq 28$). Note that going from $N_c = 40$ to $N_c = 56$ nearly doubles the number of cells.

These results raise an early conclusion that the framework's performance is mainly bounded by the complexity of the sparse pseudo-inverse smoother itself, rather than the training configuration.

7.2.2. Three-dimensional problems

The methodology for the three-dimensional problems is equivalent to the previous section, following the batched training of several models using the same configuration. However, the use of the

more powerful workstation is mandatory, as well as several modifications in the basic setup. This is due to the dramatic growth in cell counts and, thus, the size of the matrices and graphs fed to the network. This increases training and evaluation times, which hinders the rate of obtaining results. In addition, several problem datasets are used, which are comprised of 3D-only or mixed (2D and 3D) problems. The latter dataset is focused on obtaining an all-round model that performs best for both two- and three-dimensional problems.

Regarding the setup used, it has several variations depending on the dataset that is utilized for training. For 3D-only, the main changes to the configuration are presented in Tables 7.2.2a and 7.2.2b, while for the mixed dataset, the differences are displayed in Table 7.2.3. Compared to the previous problems, the RAM limitation for these trainings is significantly larger, making the use of FP32 almost mandatory to train on the same mini-batch size ($B = 10$).

The AMG solve setup is modified to use a tolerance $\delta = 10^{-4}$ to avoid plausible stagnation (FP32), and the maximum iterations are increased to 500. This makes the approach less prone to inaccuracies when comparing both smoothers. This is due to the solution of some problems requiring lots of iterations, especially for relaxed Jacobi, which easily reaches the iteration limit.

Furthermore, a significant refinement is made to the classification regarding non-convergence. This is due to the three-dimensional dataset generating more divergent cases for both smoothers (especially for tuned AMG). This produced apparent faster solve times for the smoother that diverged the most, since metrics from complex problems are removed. For detailed information on the specific process, see Appendix D.5.

7.2.2.1. 3D-only datasets

This section presents the training and evaluation of datasets consisting only of unstructured problems in three dimensions. Three types of models have been trained in 5 batches: two base setups for FP32 and FP64 data types, and one FP32 where the strength parameter has been increased to $\theta = 0.5$. In the literature, the value $\theta = 0.25$ is common for 2D problems (default setup), whereas $\theta = 0.5$ is the preferred choice for 3D. Hence, the performance of the AMG-GNN is investigated using the larger strength parameter. Due to memory constraints, the setups for FP32 and FP64 differ slightly in terms of mini-batch size, and the main changes from the two-dimensional setup are presented in Table 7.2.2. Both configurations use the same convergence tolerance when evaluated.

Table 7.2.2: Setup changes for the 3D-only dataset.

(a) FP64 config.

Item	Setting
Data type	Float64 (FP64)
Train / test problems	600 / 200
Mini-batch size B	6
Problem size N_c	$8 \leq N_c \leq 9$

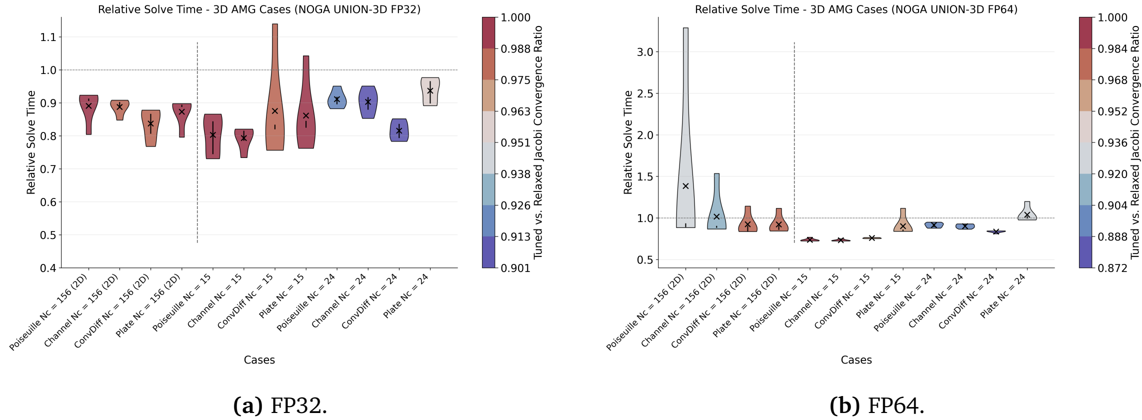
(b) FP32 config.

Item	Setting
Data type	Float32 (FP32)
Train / test problems	600 / 200
Mini-batch size B	10
Problem size N_c	$8 \leq N_c \leq 9$

The comparison between FP32 and FP64 models (Figure 7.2.3) reinforces the findings from the 2D analysis. Despite the FP64 model achieving lower training losses (likely due to overfitting on the smaller $B = 6$ batches, see Figure D.6.1), the FP32 model demonstrates slightly superior generalization on the validation set.

Performance on the largest validation meshes ($N_c = 24$, roughly 18 times larger than the largest training problems) shows that the tuned Jacobi remains faster than the baseline, though the acceleration is less pronounced than in 2D. Notably, the lowest convergence rates dropped to

$\sim 87\%$, with specific divergence observed in many of the structured grids of the 3D plate meshes. This confirms that while 3D-only training is viable, it lacks robustness when extrapolating to significantly larger scales.



(a) FP32. **(b)** FP64.

Figure 7.2.3: Relative solve times for 5 trained models with different data types. Validated in 2D ($N_c = 156$) and 3D ($N_c = 15, 24$).

Increasing the strength parameter to $\theta = 0.5$, standard for 3D geometric multigrid, creates denser coarse-grid operators (see Figure D.6.2 for a visual example). As shown in Figure 7.2.4, this configuration yields remarkable acceleration, with solve times reducing by $\sim 33\%$ on $N_c = 15$ and $\sim 30\%$ on $N_c = 24$ compared to the relaxed Jacobi.

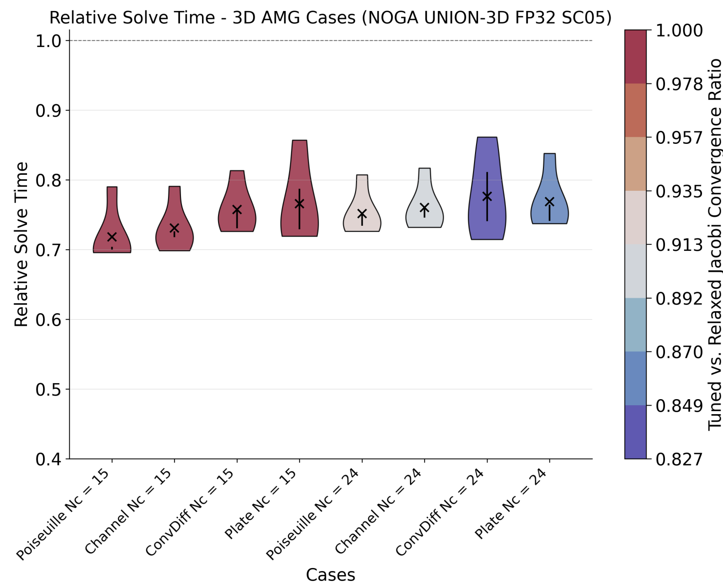


Figure 7.2.4: Relative solve times for 5 trained models on FP32 and a larger strength parameter ($\theta = 0.5$). Validated in 3D ($N_c = 15, 24$).

However, this configuration proved inferior in practical terms. First, convergence rates dropped by 10% compared to the $\theta = 0.25$ baseline. Second, and importantly, the absolute solve times increased noticeably as the number of iterations rose. The denser matrices generated by $\theta = 0.5$ significantly raised the computational cost per iteration, which the tuned smoother failed to counter with a sufficient reduction in V-cycles. This confirms that while $\theta = 0.5$ is standard for

multigrid in 3D problems, the sparser operators produced by $\theta = 0.25$ offer a superior trade-off between iteration speed and robustness for this algebraic framework and problems. Consequently, the standard parameter is retained.

7.2.2.2. Mixed-dimension dataset

To overcome the variance issues observed in 3D-only training and create a robust all-round model, a mixed dataset is introduced. This dataset combines 600 3D problems and 400 2D problems (setup detailed in Table 7.2.3), hypothesizing that the GNN could learn universal algebraic smoothing strategies from the diverse 2D topologies that would improve performance on 2D and 3D problems.

Table 7.2.3: Setup changes for the mixed dataset.

Item	Setting
Data type	Float32 (FP32)
Train / test problems	700 / 300 (3D: 600 / 2D: 400)
Mini-batch size B	10
Problem size N_c	3D: $8 \le N_c \le 9$ 2D: $10 \le N_c \le 56$

The performance of the mixed-training strategy (Base FP32) is presented in Figure 7.2.5. The results validate the hypothesis: adding 2D data significantly reduces the variance in 3D solve times, particularly for the challenging $N_c = 24$ cases, where solve time acceleration has improved by an additional $\sim 10\%$ compared to the 3D-only baseline.

Furthermore, 2D problems have also seen their solve times boosted similarly, restoring performance to levels comparable with the 2D-specific training. By seeing a richer, more varied set of graphs, the GNN learns a more robust and stable mapping, less prone to being affected negatively by a new problem. This mixed-dimension model results to be the most robust configuration in the entire study and is selected as the final candidate for the AirfRANS evaluation.

Following the success of the mixed dataset, a final series of experiments has been conducted to probe the limits of the framework. These tests, detailed in Appendix D.7, are categorized into three specific objectives: 1) testing the performance ceiling, 2) the efficiency and memory, and 3) the generalization.

First, to determine if the model is capacity-limited, experiments have been run with an increased hidden dimension ($HD = 96$) and a higher-complexity multi-polynomial (4-color) smoother, inspired by a 4-color Gauss-Seidel [85]. See Appendix D.7 for further details on the 4-color polynomial. Neither approach yields a net improvement. The larger model increases overfitting without reducing solve time (see Figure D.7.2), while the 4-color smoother degrades performance (see Figure D.7.5b). This confirms that the bottleneck is the mathematical limit of the sparse Jacobi smoother itself, not the GNN's capacity. That is, because the smoother is forced to be a sparse matrix constructed from a simple second-degree polynomial, it can only approximate the inverse up to a certain point. A larger GNN cannot extract more performance because the underlying mathematical structure of that specific smoother does not allow for faster convergence. To break this limit, one would probably need to switch to a different, more expensive algorithm, not just tune the current one better. In addition, the 4-color polynomial approach, as well as previous third-degree polynomials, demonstrates a compromise between smoother expressiveness and overfitting.

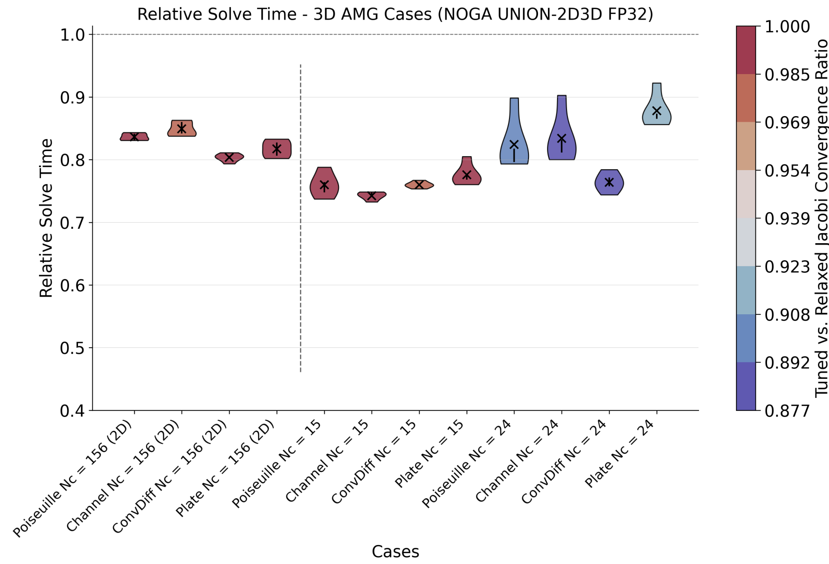


Figure 7.2.5: Relative solve times for 5 trained models with base configuration. Validated in 2D ($N_c = 156$) and 3D ($N_c = 15, 24$).

Second, to address memory constraints, models have been trained with a reduced mini-batch size ($B = 6$) and a drastically reduced architecture (2×24 GNN). The reduced GNN is able to accelerate the solve time at the cost of a slight decrease in solve time performance (see Figure D.7.5a). This indicates that much lighter models could be deployed in memory-constrained environments with acceptable performance loss. However, the main benefit of the smaller GNN resides in the reduction in GNN inference time. Concerning memory issues, reducing the model barely reduced memory occupancy. This is a result of the V-cycle computational graph for automatic differentiation having the main memory footprint.

Third, to assess robustness to unseen physics, models have been trained on datasets excluding specific flow types (no-ConvDiff) or enriched with a type of flow (plate-boosted). The model trained without convection-diffusion problems successfully generalizes to them during validation, proving that the GNN learns algebraic structures rather than flow-specific physics (see Figure D.7.3a). Similarly, the plate-boosted dataset effectively corrects performance deficits in plate flow cases (D.7.3b) while maintaining the acceleration for other problems.

7.2.3. Key findings from unstructured datasets

The extensive testing on 2D and 3D unstructured problems in this chapter has been designed to evaluate the AMG-GNN framework against its primary design objective: accelerating solve times on geometrically complex and varied meshes. The individual experiments in Sections 7.2.1 and 7.2.2, which tested numerous data configurations, training setups, and model architectures, converge on four critical findings. These conclusions define the practical capabilities, fundamental limitations, and future potential of this data-driven approach.

- **Data diversity is the main driver for generalization**. The single most important factor for building a successful model is the diversity of the training data. A large but homogeneous dataset is less effective than a smaller, more varied one.
  In this regard, the model trained only on Poiseuille-flow data (Section 7.2.2.1) fails to generalize, showing poor performance and high variance on new problems, even of the

same type (Figure D.1.3). In contrast, the 2D union dataset (Figure 7.2.1) provides stable and consistent acceleration of 17-28% across all problem types. This proves the GNN is not learning features of a specific flow, but rather the general algebraic properties of diverse graph structures. This principle was further confirmed by the mixed 2D/3D dataset (Figure 7.2.5), which produced the most robust model by reducing 3D variance and improving 2D performance simultaneously. A rich, mixed-dimension dataset is therefore a fundamental requirement.

- **Practical viability is unlocked by FP32 training**. The framework's practical application, especially for large 3D problems, is highly dependent on memory efficiency. The tests confirm that FP32 is not only viable but superior.
  As identified in Section 7.2.2, GPU VRAM is the primary bottleneck for training and inference on 3D problems. Accordingly, the FP32 training (Figures 7.2.2, 7.2.5) is highly effective, yielding acceleration with lower variance than the FP64 models. This finding is critical. It demonstrates that the GNN's prediction does not depend on high-precision (FP64) inputs. By halving the memory footprint of the model, setup, and V-cycle operations, FP32-native training is the only viable path for scaling this framework to real-world 3D problem sizes.
- **Performance is limited by the smoother, not the GNN**. A clear performance limit has been observed, which could not be surpassed by simply increasing the GNN's complexity.
  In this respect, increasing the GNN size to HD96 (Figure D.7.2) does not improve acceleration. It merely increases variance and overfitting. Conversely, even a drastically smaller 2×24 GNN (Figure D.7.5a) still provides reasonable acceleration. Similarly, increasing the smoother's complexity via a 4-color polynomial (Figure D.7.5b) also fails, leading to overfitting and worse performance. In essence, the GNN is not the bottleneck. The tuned Jacobi acceleration is limited by the inherent simplicity of the pseudo-inverse smoother (a sparse, 2nd-degree polynomial). A GNN with increased predictive potential (number of trainable parameters or non-linear complexity) cannot extract more performance from these fixed-complexity matrices. This suggests that a simpler, more robust GNN is preferable to a larger, over-parameterized one. Nonetheless, a larger GNN (up to a reasonable limit) benefits from acceleration if early-stopping is implemented.
- **Fundamental trade-off exists between speed and robustness**. The fastest configuration is not always the best. The results repeatedly show that aggressive optimization for speed comes at the direct cost of reliability.
  Accordingly, the clearest example is the 3D strength parameter test ($\theta = 0.5$) in Figure 7.2.4. This model achieves a remarkable 30% acceleration on large 3D problems, far exceeding the base model. However, this speed comes at the cost of a 10% drop in convergence rate. This highlights a critical trade-off. For any practical solver, robustness is non-negotiable. A solver that is 30% faster but fails on 10% more problems is less useful than a reliable one but slower. This leads to the conclusion that the more conservative, stable setup (like the base FP32 model with $\theta = 0.25$) is the superior and more reliable choice for a general-purpose tool.

These findings provide a clear picture of the tuned AMG capabilities and inform the final, decisive test: applying the best-performing mixed-dimension model to the industry-relevant AirfRANS dataset.

7.3. AirfRANS

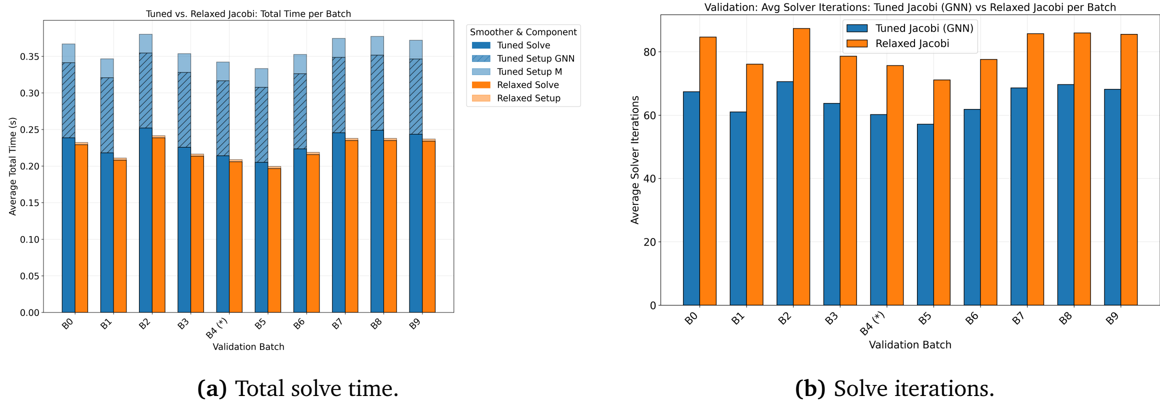
The final phase of validation assesses the tuned AMG generalization capability on the AirfRANS dataset. This represents a significant jump in complexity compared to the training data: the problems involve Reynolds-averaged turbulent flows (different boundary layers than laminar flows), high Reynolds numbers, and unstructured meshes with anisotropic inflation layers that are up to 16.5 times larger than the largest training graphs. See Section 5.1.3 for more information on the dataset. The training data reaches up to 19,702 grid cells, while the AirfRANS dataset reaches up to 324,166 cells. Accordingly, the model has to infer adequate polynomial coefficients for considerably larger graphs that present a different distribution of data due to the prism layers (e.g., exponential growth that the model has not seen before).

Since the AirfRANS dataset provides only the problem geometry and flow conditions, the linear systems were reconstructed using ReFRESCO. All validations were performed using the best-performing model from the previous section (Mixed 2D/3D, FP32, $B = 10$), with the solver configured to FP64 to avoid stagnation for the given tolerance (10^{-4}).

7.3.1. Generalization performance

The performance of the GNN-tuned smoother on the best test model is presented in Figure 7.3.1. These results reveal a distinct split between algorithmic success and computational cost.

The model demonstrates remarkable robustness. Despite never seeing an inflation layer or a RANS turbulent flow solution during training, the GNN successfully generalized to these physics, reducing the number of V-cycles by 19.7% on average compared to the relaxed Jacobi baseline (Figure 7.3.1b). The convergence ratio was near-perfect compared to relaxed Jacobi (99.9%), proving that the learned algebraic mapping holds even for highly distinct topologies.



(a) Total solve time. **(b)** Solve iterations.

Figure 7.3.1: Validation results for the best test model for mixed dataset (2D/3D), FP32 and $B = 10$.

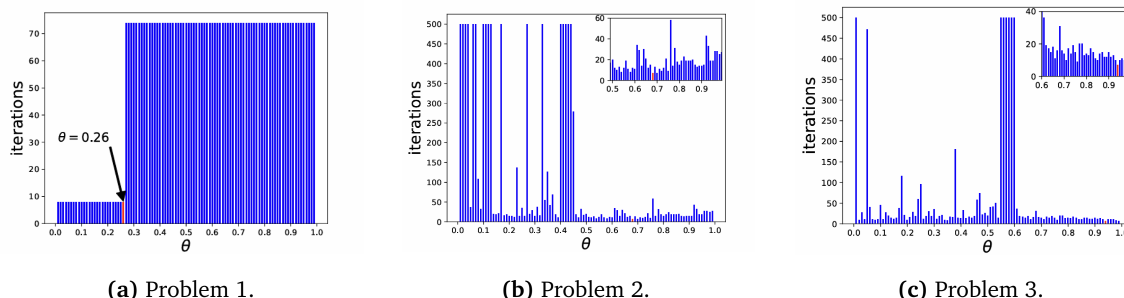
However, this iteration reduction does not translate into solve-time acceleration. The total solve time increased by 4.7% (Figure 7.3.1a). This net slowdown is driven by the computational overhead of the sparse pseudo-inverse smoother, which is denser and more expensive to apply than the diagonal relaxed Jacobi. For this specific problem class, the 19.7% reduction in V-cycles is insufficient to compensate for the higher cost per iteration and the one-time GNN setup cost.

7.3.2. Sensitivity to strength parameter θ

To determine if performance was limited by the AMG hierarchy construction rather than the smoother itself, a sensitivity analysis has been conducted on the strength parameter θ. The

AirfRANS problems have proved highly sensitive to this parameter. Increasing θ beyond 0.3 causes the V-cycle numbers to explode, confirming that these problems are optimally solved on sparser hierarchies (low θ).

Concerning solver efficiency, the sensitivity of AMG to the strength parameter is critical. The problem nature can have an important influence on the optimal θ as illustrated by Figure 7.3.2.

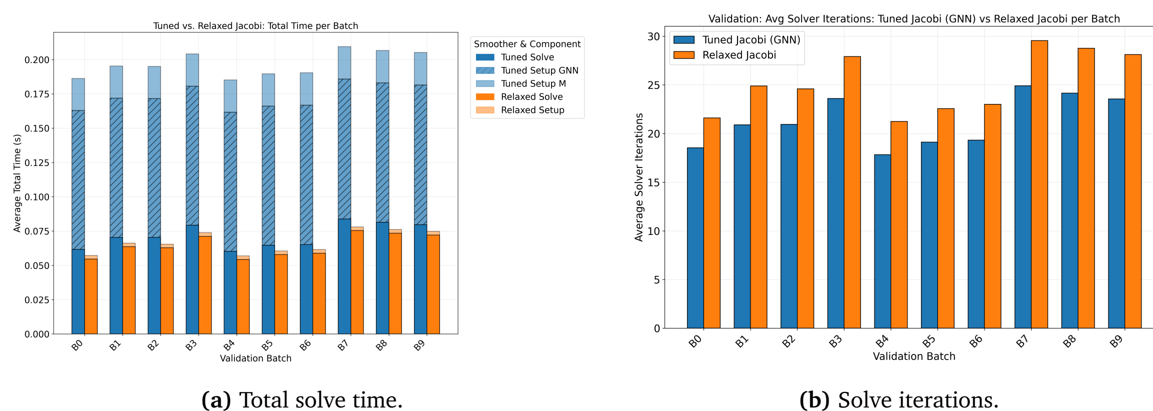


(a) Problem 1. **(b)** Problem 2. **(c)** Problem 3.

Figure 7.3.2: Number of V-cycle iterations of three different problems as a function of the strength parameter. Source: [24].

The AirfRANS dataset is characterized by a positive bias towards low strength parameter values similar to Figure 7.3.2a. Reducing the parameter to $\theta = 0.2$ drastically reduces the iteration count for both smoothers (by approximately 70%), as shown in Figure 7.3.3. While convergence rates remained perfect, the difference in relative performance between tuned and relaxed Jacobi was maintained.

The change in θ made the solver extremely efficient, reducing absolute solve times and the number of V-cycles. However, it left the GNN setup as the main time-consuming step: the solve time became so fast that the fixed GNN inference cost became the dominant factor, nearly doubling the solve time.



(a) Total solve time. **(b)** Solve iterations.

Figure 7.3.3: Validation results for the best test model for mixed dataset (2D/3D), FP32 and $B = 10$ and $\theta = 0.2$.

7.3.2.1. Conclusive analysis for AirfRANS

The evaluation on the AirfRANS dataset has served as the definitive test of the model's generalization, applying it to a complex, turbulent, practical aerodynamics problem for which it has not been trained. The results are not a simple pass/fail but instead reveal a complex interplay of costs,

smoother effectiveness, and problem-specific sensitivities that define the framework's practical boundaries.

The initial validation (Figure 7.3.1) highlights a core trade-off. The GNN-tuned smoother successfully learns a generalizable policy, converging in 19.7% fewer iterations than the relaxed Jacobi baseline with a near-perfect 99.9% convergence rate. However, this success does not translate to a wall-clock speedup. The solver is 4.7% slower. This is due to the GNN-tuned V-cycle being more computationally expensive than the relaxed Jacobi's diagonal operation. The 19.7% iteration reduction is simply not large enough to overcome this cost deficit for this specific problem.

The subsequent sensitivity analysis, testing different θ values, provides a much clearer picture. First, with respect to problem-specific sensitivity, the AirfRANS problem itself is highly sensitive to θ, performing optimally with a low value of $\theta = 0.2$ (Figure 7.3.3) and poorly with $\theta \geq 0.3$. This low θ value creates sparser matrices, which are more memory-bound and do not fully leverage the GPU's parallel compute capabilities. This, however, does not invalidate the framework's potential. As has been demonstrated in the 3D unstructured tests (Figure 7.2.4), problems that do benefit from a larger strength parameter ($\theta = 0.5$) can see a $\sim$30% relative acceleration. For more compute-bound problems, the GNN-smoother's denser matrices and higher cost-per-cycle can be effectively compensated for if they carry a noticeable reduction in iterations, and thus, in absolute solve times.

This analysis makes it clear that the GNN's viability is constrained by two distinct bottlenecks: the GNN's effectiveness at increasing convergence rate and its cost per iteration. The GNN's iteration reduction is limited. This is almost certainly due to a training data mismatch. The model has been trained on simple, laminar-flow graphs and has therefore been unprepared for the anisotropic inflation layers and turbulent flow physics of AirfRANS. This suggests that a model trained on this more representative data could learn a far more effective mapping, achieving a much greater iteration reduction. On the other hand, the $\theta = 0.2$ test (Figure 7.3.3) reveals a more fundamental issue. Because the solver has become so efficient, the fixed one-time cost of the GNN's setup-time inference pass has become the dominant factor, more than doubling the actual solve times.

The AMG-GNN framework, in its current form, is not a universally practical accelerator. Its viability is conditional, and it shows clear potential for problems where the tuned smoother significantly outperforms the baseline in terms of residual reduction. For these cases, a GNN, if trained on representative data, could provide an iteration reduction significant enough to overcome both its higher V-cycle cost and its expensive setup inference cost. For problems like AirfRANS, which have not been used in training and prefer sparser matrices, the pseudo-inverse smoother is not effective enough. Thus, the current tuned AMG performance and setup cost make it an impractical choice.

8

Conclusions

The analysis of the AMG-GNN performance for both structured and unstructured datasets presented in the previous chapter allows to provide answers to the main and secondary research questions defined in Section 2.5. Furthermore, based on the observations and results obtained, suggestions for future work are proposed.

8.1. Main research question

To what extent can a GNN-based approach for optimizing a Jacobi AMG smoother accelerate the solution of the pressure-Poisson equation in CFD simulations compared to traditional AMG implementations with fixed smoothers?

The results presented in Chapter 7 demonstrate that a GNN-based approach can effectively enhance AMG performance across a wide range of CFD problems. The AMG-GNN framework consistently reduces solve times for both structured and unstructured grids, including 3D cases that remain largely unexplored in existing literature. Moreover, several of these test cases, particularly those drawn from aerospace and maritime applications, have not been considered in prior machine-learning-based AMG studies. The GNN-predicted polynomial coefficients enable the construction of a sparse pseudo-inverse smoother that is more effective than the classical relaxed Jacobi operator, leading to faster convergence in most scenarios.

Moreover, the trained models also generalize well to significantly larger and previously unseen problems. When exposed to a sufficiently diverse dataset, the GNN learns algebraic patterns that extend beyond specific geometries or flow configurations, confirming the viability of graph-based learning as a scalable tool for accelerating AMG in realistic CFD settings. The extent of this acceleration is defined by three key findings.

First, the AMG-GNN framework demonstrates successful scaling to larger grids, achieving wall-clock solve time reductions averaging 17% to 25% across diverse structured and unstructured meshes. Remarkably, this performance extended to problems significantly larger than the training set, up to 64 times larger for 3D structured grids and 20 times larger for 2D unstructured grids, while maintaining high convergence rates.

Further, the framework is robust to unseen complex geometries. The model successfully generalized to practical aerodynamic simulations (AirfRANS) without prior knowledge of their specific geometries or flow regimes. It successfully learned the underlying algebraic structure of the system, evidenced by a 20% reduction in V-cycles on these complex, unseen cases.

Finally, the extent of the acceleration is ultimately limited by a trade-off between iteration reduction and computational overhead. While the GNN consistently reduces the required number of solver iterations, the sparse pseudo-inverse smoother introduces a higher cost per iteration. Consequently, the framework provides net acceleration only when the reduction in V-cycles is

substantial enough to offset this added cost. This limit was observed in the AirfRANS test case, where the overhead resulted in a 4.7% net slowdown despite the reduction in iterations.

Ultimately, this study concludes that the AMG-GNN acceleration capabilities are heavily dependent on the pseudo-inverse matrix's ability to reduce the residual. While the GNN successfully learns algebraic patterns across all tiers, the extent of practical speedup is determined by whether the reduction in iterations is sufficient to outweigh the computational overhead of the learned smoother.

8.2. Secondary research questions

How can GNN architectures be effectively designed and implemented to learn optimal smoothing coefficients for a tuned Jacobi smoother in an AMG solver on (un)structured meshes?

This framework demonstrates that graph convolutional isomorphism networks are a viable approach, even though inference times can be significant, given that GNNs are expensive by nature, especially as the graph size grows. In addition, this framework shows high sensitivity to hyperparameters such as GNN depth, hidden dimension, learning rate, and batch size..

The key findings that drive GNN performance found in this thesis are essential for building a robust and feasible model. First, larger models offer better accuracy but are more prone to overfitting, especially when the dataset is limited in diversity, for instance, by not having a large number of different coefficient matrices. At the same time, smaller GNNs are able to provide meaningful acceleration while remaining more computationally efficient. This converts them into more useful tools when the AMG-GNN setup is constrained by memory or time. Although, unless the acceleration provided by the model is sufficient, setup time is not generally a limitation.

Moreover, two of the most important design choices for the GNN are an adequate GCIN output management and a wise selection of the data type. This model prediction of polynomial coefficients is based on a final MLP layer, and concatenating outputs from each GNN layer proved to be the most robust and effective design in terms of performance. Further, the performance of the GNN is dependent on the richness of the training dataset. Using a low-precision data type is a critical design choice that allows larger datasets while maintaining near identical performance, as long as the data can be adequately represented at that precision.

Still, more work is needed to refine the architecture and fully exploit the potential of data-driven smoothers. This includes evaluating alternative message-passing operators, such as TAGConv or other higher-order and attention-based GNNs, which may better capture the algebraic structure of AMG systems, particularly on highly irregular unstructured meshes. In addition, exploring cost-aware models, adaptive depth mechanisms, and improved output parameterizations could help balance smoother expressiveness with overfitting. Overall, while the present framework establishes a solid foundation, continued investigation of diverse GNN architectures and training strategies will be essential to further enhance performance and generalization across the full breadth of CFD problem types.

What is the quantifiable impact of a GNN-tuned Jacobi smoother on key AMG performance metrics, such as reduction in the number of AMG cycles and overall solver wall-clock time, for representative (un)structured grid problems?

The impact of the AMG-GNN framework varies by grid topology (structured or unstructured) and problem complexity. Quantifiable performance gains are assessed through two primary metrics: the reduction in V-cycle counts and the net acceleration in wall-clock time.

First, the tuned smoother consistently lowers the required iteration count compared to the relaxed Jacobi baseline. Structured grids show the largest improvement, with average reductions of

36% for 3D/unsteady problems and 32% for 2D problems. Meanwhile, for unstructured problems, due to more complex connectivity, the reductions are lower but significant, achieving 29% for 2D and 24% for 3D data. Even on unseen, complex cases like AirfRANS, the model consistently achieves a 20% reduction in V-cycles, demonstrating great generalization.

Second, net acceleration depends on whether the reduction in V-cycles outweighs the higher per-iteration cost of the sparse pseudo-inverse smoother. In structured grids, the tuned smoother achieves the highest efficiency, with approximately 25% faster solve times. Moreover, both 2D and 3D unstructured problems achieve accelerations of 17% and 18%, respectively. In the AirfRANS case, the increased computational cost of the tuned smoother surpasses the V-cycle reduction, resulting in a 4.7% net slowdown.

This shows that, even with its higher per-iteration cost, the AMG-GNN smoother provides a clear and measurable improvement to AMG performance across most representative CFD problems. Structured grids benefit most strongly, and unstructured cases still achieve meaningful acceleration. While the AirfRANS cases reveal the current limitations of the approach, the overall results confirm that a GNN-tuned Jacobi smoother can significantly enhance multigrid convergence behaviour and solver performance across a wide range of practical problem configurations.

To what degree can a single GNN model, trained on a diverse dataset of CFD problem types (e.g., Poiseuille, channel, plate flows, etc.) and mesh configurations (e.g., varying cell count, cell types, grid dimension, etc.), generalize its ability to effectively tune AMG smoothers for unseen but related CFD problems and meshes without retraining?

The GNN model demonstrates strong generalization capabilities across three distinct dimensions: problem type, mesh topology, and problem scale.

First, the GNN learns the underlying algebraic structure of Poisson systems rather than specific flow features. For unseen flow regimes, models trained purely on synthetic data outperformed models trained specifically on unsteady datasets when tested on unsteady cases. Further, a mixed 2D/3D model trained without convection-diffusion problems successfully accelerated ConvDiff cases during validation. Even on unseen RANS flows, the model reduced V-cycles for the AirfRANS dataset, proving it can handle physics not present in the training set, such as different boundary layers in terms of size and gradients.

Second, mesh diversity, rather than dataset size, drives generalization performance. Models trained on pure 3D unstructured data successfully accelerated unseen 2D mesh types (e.g., advancing front mesh types), although with a higher rate of outliers. Further, mixed-dimension and mixed-mesh datasets produce the most robust models. However, training solely on 2D data does not effectively generalize to 3D problems.

Furthermore, a notable observation is that the GNN can extrapolate its learned mapping to graphs far larger than those in training. In structured 2D and 3D problems, the GNN maintained acceleration even when the grid grew by factors of 128 and 64 in total cell count, respectively. Meanwhile, in unstructured datasets, effective performance persisted up to 20 and 18.5 times more cells, although acceleration gradually diminished the larger the difference.

This thesis demonstrates that a GNN-tuned Jacobi smoother can substantially accelerate AMG solvers across a wide spectrum of CFD problems while retaining strong robustness and generalization. The approach provides a practical, GPU-accelerated framework that complements and extends classical AMG methods with little to no retraining.

8.3. Future work

Based on the findings, further research is recommended to better understand and define the limits of using a GNN-based tuned smoother by exploring currently untested design and training choices:

- The model demonstrated generalization to aerodynamic 2D simulations, yet the smoother effectiveness was insufficient to provide net solve acceleration. To improve generalization on these complex topologies, future training datasets should incorporate reduced samples containing coarse information about geometry and inflation layers.
- Memory constraints during training have been a persistent issue throughout the whole study. Aside from overfitting, the AMG-GNN framework is balanced between smoother effectiveness and training cost. Accordingly, the limitations in the mini-batch size and the dataset matrix sizes limit the learning capacity of the model. Thus, several approaches aiming towards cheaper training in terms of memory at the cost of a slight reduction in model capacity are provided:
  - Instead of using the GNN-tuned Jacobi throughout the whole V-cycle, a mixed approach where a relaxed Jacobi smoother is used on the descending (coarsening) part of the V-cycle and the tuned Jacobi on the ascending (prolongation) part should be tested. One advantage can be deduced. The autograd graph memory usage is nearly halved, so larger mini-batch sizes or problems can be used for training.
  - Furthermore, one design choice for faster training consists in inferring the whole mini-batch graphs at once. This has an impact on memory usage, especially when the graphs are already considerably large. As a result, by avoiding the creation of many graph copies with larger data usage (due to the number of hidden dimensions), a relevant portion of the RAM could be freed, possibly allowing to introduce more graphs in the mini-batch.
- Training on problems that contain information on fine properties is not feasible due to memory limitations. This is a large drawback of the current model, as large anisotropies as the ones in fine boundary layers, or turbulent fine scales at large Reynolds numbers, cannot be learned by the GNN. One interesting approach is using these large grids and building equivalent problems that can actually be fed into the GNN. To do this, the fine system $\mathbf{Au} = \mathbf{f}$ can be written as

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{pmatrix}. \tag{8.3.1}$$

  If the system is solved ($\mathbf{u}^*$), then a smaller equivalent system containing the fine grid effects in a subdomain can be constructed. For $\mathbf{A}_{11}$ this would be

$$\mathbf{A}_{11}\mathbf{u}_1 = \mathbf{f}_1 - \mathbf{A}_{12}\mathbf{u}_2^* = \mathbf{f}'. \tag{8.3.2}$$

  This smaller but equivalent system with *artificial* boundary conditions can be used to train the GNN in a feasible way. However, due to the artificial cut, extra care should be taken to avoid the GNN learning strange behavior (e.g., not using the boundary nodes for computing the loss).
- The AMG-GNN setup cost on inference is limited by two main factors: the GNN inference time and the memory usage. As discussed during the results, the GNN time can be drastically reduced by designing a smaller GNN in terms of layers and hidden dimension. Nonetheless,

on many occasions, time and memory costs are still large. One plausible solution that would worsen smoother effectiveness, but alleviate the two issues, is to train and evaluate the model using pooled matrices. By extending the approach from [23] to graphs, performing a pooling on the matrices of coefficients that are converted to graphs for GNN inference hinders the learned features, but allows for faster GNN inferences and lower memory costs. This is achieved by maintaining the most important features in a smaller representative graph.

- The GNN design has remained fairly steady throughout the study. Hence, the effectiveness of other designs has yet to be determined. Specifically, one direct modification to the GNN could be interchanging the GCNConv with TAGConv layers. Having many GCNConv layers effectively gives information to the network about far neighbors, although smoothed out. In contrast, the TAGConv layers can decide how many hops of neighbors are included for the convolution. This could provide the network with more expressiveness, with fewer layers but at a higher cost. However, performance and overfitting should be carefully evaluated.
- During the thesis, the framework performance has revolved around a constant compromise between smoother expressiveness and overfitting to the training dataset. Further investigation could aim at finding other pseudo-inverse generation methods.

Bibliography

[1] Federica Gattere et al. “Turbulent drag reduction with streamwise-travelling waves in the compressible regime”. In: *Journal of Fluid Mechanics* 987 (May 2024), A30. ISSN: 0022-1120. DOI: 10.1017/jfm.2024.408.

[2] Esther Lagemann et al. “Towards extending the aircraft flight envelope by mitigating transonic airfoil buffet”. In: *Nature Communications* 15 (1 June 2024), p. 5020. ISSN: 2041-1723. DOI: 10.1038/s41467-024-49361-3.

[3] Lei Wang, Xiaomin Liu, and Dian Li. “Noise reduction mechanism of airfoils with leading-edge serrations and surface ridges inspired by owl wings”. In: *Physics of Fluids* 33 (1 Jan. 2021). ISSN: 1070-6631. DOI: 10.1063/5.0035544.

[4] Vincent Cognet, Sylvain Courrech du Pont, and Benjamin Thiria. *Material optimization of flexible blades for wind turbines*. 2020. arXiv: 2001.09072 [physics.app-ph]. URL: https://arxiv.org/abs/2001.09072.

[5] Wei Tang et al. “Application of CFD simulations for short-range atmospheric dispersion over open fields and within arrays of buildings”. In: *86th AMS Annual Meeting* (May 2006).

[6] Paul D Morris et al. “Computational fluid dynamics modelling in cardiovascular medicine.” In: *Heart (British Cardiac Society)* 102 (1 Jan. 2016), pp. 18–28. ISSN: 1468-201X. DOI: 10.1136/heartjnl-2015-308044.

[7] Daejeong Kim and Tahsin Tezdogan. “CFD-based hydrodynamic analyses of ship course keeping control and turning performance in irregular waves”. In: *Ocean Engineering* 248 (2022), p. 110808. ISSN: 0029-8018. DOI: https://doi.org/10.1016/j.oceaneng.2022.110808. URL: https://www.sciencedirect.com/science/article/pii/S0029801822002542.

[8] Cesar Jimenez Navarro et al. “Numerical Study and Physical Analysis of the Transonic Interaction and Its Modification Through Morphing Around Supercritical Wings at High Reynolds Number”. In: *Towards Effective Flow Control and Mitigation of Shock Effects in Aeronautical Applications*. Ed. by Pawel Flaszynski, Holger Babinsky, and Piotr Doerffer. Cham: Springer Nature Switzerland, 2025, pp. 221–248. ISBN: 978-3-031-86605-0. DOI: 10.1007/978-3-031-86605-0_11. URL: https://doi.org/10.1007/978-3-031-86605-0_11.

[9] R. Fernandez-Feria and E. Sanmiguel-Rojas. “An explicit projection method for solving incompressible flows driven by a pressure difference”. In: *Computers & Fluids* 33 (3 Mar. 2004), pp. 463–483. ISSN: 00457930. DOI: 10.1016/S0045-7930(03)00062-8.

[10] R.I Issa. “Solution of the implicitly discretised fluid flow equations by operator-splitting”. In: *Journal of Computational Physics* 62 (1 Jan. 1986), pp. 40–65. ISSN: 00219991. DOI: 10.1016/0021-9991(86)90099-9.

[11] Stephen Turnock et al. “Performance Analysis of Massively-Parallel Computational Fluid Dynamics”. In: Oct. 2014.

[12] Massimo Bernaschi et al. “Petaflop biofluidics simulations on a two million-core system”. In: Nov. 2011, p. 4. DOI: 10.1145/2063384.2063389.

[13] C. Brezinski. “Numerical Approximation and Analysis”. In: *Encyclopedia of Condensed Matter Physics*. Elsevier, 2005, pp. 132–134. DOI: 10.1016/B0-12-369401-9/00563-5.

[14] Marcel Ferrari. “A Comparison of Sparse Solvers for Severely Ill-Conditioned Linear Systems in Geophysical Marker-In-Cell Simulations”. In: (Sept. 2024).

[15] John N. Jomo et al. “Robust and parallel scalable iterative solutions for large-scale finite cell analyses”. In: (Sept. 2018). DOI: 10.1016/j.finel.2019.01.009.

[16] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Jan. 2003. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003.

[18] P. Wesseling. *An Introduction to Multigrid Methods*. PURE and APPLIED MATHEMATICS: a WILEY INTERSCIENCE SERIES of TEXTS MONOGRAP HS and TRACTS Series. Wiley, 1992. ISBN: 9780471930839. URL: https://books.google.es/books?id=MznvAAAAMAAJ.

[19] Achi Brandt and Oren E. Livne. *Multigrid Techniques*. Society for Industrial and Applied Mathematics, Jan. 2011. ISBN: 978-1-61197-074-6. DOI: 10.1137/1.9781611970753.

[20] R.D. Falgout. “An introduction to algebraic multigrid”. In: *Computing in Science Engineering* 8.6 (2006), pp. 24–33. DOI: 10.1109/MCSE.2006.105.

[21] MARIN Netherlands. *ReFRESCO: Versatile solver, tailored for maritime challenges*. https://www.marin.nl/en/about/facilities-and-tools/software/refresco. Accessed: 2025-10-10. 2025.

[22] Zhihao Li et al. “M2NO: Multiresolution Operator Learning with Multiwavelet-based Algebraic Multigrid Method”. In: (Oct. 2024).

[23] Matteo Caldana, Paola F. Antonietti, and Luca Dede’. “A deep learning algorithm to accelerate algebraic multigrid methods in finite element solvers of 3D elliptic PDEs”. In: *Computers & Mathematics with Applications* 167 (Aug. 2024), pp. 217–231. ISSN: 08981221. DOI: 10.1016/j.camwa.2024.05.013.

[24] Haifeng Zou Haifeng Zou et al. “AutoAMG(θ): An Auto-Tuned AMG Method Based on Deep Learning for Strong Threshold”. In: *Communications in Computational Physics* 36 (1 Jan. 2024), pp. 200–220. ISSN: 1991-7120. DOI: 10.4208/cicp.OA-2023-0072.

[25] Ru Huang et al. “Reducing Operator Complexity of Galerkin Coarse-grid Operators with Machine Learning”. In: *SIAM Journal on Scientific Computing* 46 (5 Oct. 2024), S296–S316. ISSN: 1064-8275. DOI: 10.1137/23M1583533.

[26] Nicolas Nytko. “Learning aggregates and interpolation for algebraic multigrid”. PhD thesis. University of Illinois, Apr. 2022.

[27] Ilay Luz et al. “Learning algebraic multigrid using graph neural networks”. In: *Proceedings of the 37th International Conference on Machine Learning*. ICML’20. JMLR.org, 2020.

[28] Dmitry Kuznichov. *Learning Relaxation for Multigrid*. 2022. arXiv: 2207.11255 `[cs.LG]`. URL: https://arxiv.org/abs/2207.11255.

[29] Ru Huang, Ruipeng Li, and Yuanzhe Xi. “Learning Optimal Multigrid Smoothers via Neural Networks”. In: *SIAM Journal on Scientific Computing* 45 (3 June 2023), S199–S225. ISSN: 1064-8275. DOI: 10.1137/21M1430030.

[30] Gabriel D. Weymouth. “Data-driven Multi-Grid solver for accelerated pressure projection”. In: *Computers & Fluids* 246 (Oct. 2022), p. 105620. ISSN: 00457930. DOI: 10.1016/j.compfluid.2022.105620.

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[32] Peter W. Battaglia et al. “Relational inductive biases, deep learning, and graph networks”. In: (Oct. 2018). URL: http://arxiv.org/abs/1806.01261.

[33] H. Xiao et al. “Quantifying and reducing model-form uncertainties in Reynolds-averaged Navier–Stokes simulations: A data-driven, physics-informed Bayesian approach”. In: *Journal of Computational Physics* 324 (Nov. 2016), pp. 115–136. ISSN: 00219991. DOI: 10.1016/j.jcp.2016.07.038.

[34] Jin-Long Wu, Heng Xiao, and Eric Paterson. "Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework". In: *Physical Review Fluids* 3 (7 July 2018), p. 074602. ISSN: 2469-990X. DOI: 10.1103/PhysRevFluids.3.074602.

[35] Shirindokht Yazdani and Mojtaba Tahani. "Data-driven discovery of turbulent flow equations using physics-informed neural networks". In: *Physics of Fluids* 36 (3 Mar. 2024). ISSN: 1070-6631. DOI: 10.1063/5.0190138.

[36] Mustafa Z. Yousif et al. "A deep-learning approach for reconstructing 3D turbulent flows from 2D observation data". In: *Scientific Reports* 13 (1 Feb. 2023), p. 2529. ISSN: 2045-2322. DOI: 10.1038/s41598-023-29525-9.

[37] R. Maulik et al. "Subgrid modelling for two-dimensional turbulence using neural networks". In: *Journal of Fluid Mechanics* 858 (Jan. 2019), pp. 122–144. ISSN: 0022-1120. DOI: 10.1017/jfm.2018.770.

[38] Yichen Li et al. "Learning Preconditioners for Conjugate Gradient PDE Solvers". In: *Proceedings of the 40th International Conference on Machine Learning*. Vol. 202. Proceedings of Machine Learning Research. PMLR, July 2023, pp. 19425–19439. URL: https://proceedings.mlr.press/v202/li23e.html.

[39] David Millard et al. *PEARL: Preconditioner Enhancement through Actor-critic Reinforcement Learning*. Jan. 2025. DOI: 10.48550/arXiv.2501.10750.

[40] Yousef Saad. "Schur Complement Preconditioners for Distributed General Sparse Linear Systems". In: *Domain Decomposition Methods in Science and Engineering XVI*. Springer Berlin Heidelberg, pp. 127–138. DOI: 10.1007/978-3-540-34469-8_11.

[41] Michele Benzi and Zhen Wang. "Analysis of Augmented Lagrangian-Based Preconditioners for the Steady Incompressible Navier–Stokes Equations". In: *SIAM Journal on Scientific Computing* 33 (5 Jan. 2011), pp. 2761–2784. ISSN: 1064-8275. DOI: 10.1137/100797989.

[42] Gunnar A. Staff, Kent-Andre Mardal, and Trygve K. Nilssen. "Preconditioning of fully implicit Runge-Kutta schemes for parabolic PDEs". In: *Modeling, Identification and Control: A Norwegian Research Bulletin* 27 (2 2006), pp. 109–123. ISSN: 0332-7353. DOI: 10.4173/mic.2006.2.3.

[43] Ruilin Chen, Xiaowei Jin, and Hui Li. "A machine learning based solver for pressure Poisson equations". In: *Theoretical and Applied Mechanics Letters* 12 (5 Sept. 2022), p. 100362. ISSN: 20950349. DOI: 10.1016/j.taml.2022.100362.

[44] Francisco Holguin, GS Sidharth, and Gavin Portwood. "Accelerating multigrid solver with generative super-resolution". In: (Mar. 2024).

[45] Massimo Bernaschi et al. "Communication-reduced Conjugate Gradient Variants for GPU-accelerated Clusters". In: (Jan. 2025).

[46] Quang Tuyen Le and Chinchun Ooi. "Surrogate modeling of fluid dynamics with a multigrid inspired neural network architecture". In: *Machine Learning with Applications* 6 (Dec. 2021), p. 100176. ISSN: 26668270. DOI: 10.1016/j.mlwa.2021.100176.

[47] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[48] Jeong-Kweon Seo. "A reduced-form multigrid approach for ANN equivalent to classic multigrid expansion". In: *Neural Computing and Applications* 36 (33 Nov. 2024), pp. 20907–20926. ISSN: 0941-0643. DOI: 10.1007/s00521-024-10311-1.

[49] Ali Taghibakhshi et al. "MG-GNN: Multigrid graph neural networks for learning multilevel domain decomposition methods". In: *Proceedings of the 40th International Conference on Machine Learning*. ICML'23. Honolulu, Hawaii, USA: JMLR.org, 2023.

[50] Matteo Caldana. "Improving Efficiency of Algebraic Multigrid Methods through Artificial Neural Networks". PhD thesis. Politecnico di Milano, 2019.

[51] Paola F. Antonietti, Matteo Caldana, and Luca Dede'. "Accelerating Algebraic Multigrid Methods via Artificial Neural Networks". In: *Vietnam Journal of Mathematics* 51 (1 Jan. 2023), pp. 1–36. ISSN: 2305-221X. DOI: 10.1007/s10013-022-00597-w.

[52] Ali Taghibakhshi et al. "Optimization-based algebraic multigrid coarsening using reinforcement learning". In: *Proceedings of the 35th International Conference on Neural Information Processing Systems*. NIPS '21. Red Hook, NY, USA: Curran Associates Inc., 2021. ISBN: 9781713845393.

[53] Daniel Greenfeld et al. "Learning to Optimize Multigrid PDE Solvers". In: Feb. 2019. DOI: 10.48550/arXiv.1902.10248.

[54] Fan Wang et al. "Learning-based local weighted least squares for algebraic multigrid method". In: *Journal of Computational Physics* 493 (Nov. 2023), p. 112437. ISSN: 00219991. DOI: 10.1016/j.jcp.2023.112437.

[55] J. H.. Bramble. *Multigrid methods*. Chapman & Hall/CRC, 2019. ISBN: 0367449714.

[56] Achi Brandt. "Multi-Level Adaptive Technique (MLAT) for Fast Numerical Solution to Boundary Value Problems". In: vol. 18. Feb. 2008, pp. 82–89. ISBN: 978-3-540-06170-0. DOI: 10.1007/BFb0118663.

[57] James H. Bramble, Do Y. Kwak, and Joseph E. Pasciak. "Uniform Convergence of Multigrid V-Cycle Iterations for Indefinite and Nonsymmetric Problems". In: *SIAM Journal on Numerical Analysis* 31.6 (1994), pp. 1746–1763. DOI: 10.1137/0731089. eprint: https://doi.org/10.1137/0731089. URL: https://doi.org/10.1137/0731089.

[58] James H Bramble and Xuejun Zhang. *UNIFORM CONVERGENCE OF THE MULTIGRID V-CYCLE FOR AN ANISOTROPIC PROBLEM*. Tech. rep. 2000, pp. 1222–1231.

[59] Aidan Wimshurst. *Multi-Grid for CFD (Part 1)*. May 2023.

[60] K. Stüben. "A review of algebraic multigrid". In: *Journal of Computational and Applied Mathematics* 128.1 (2001). Numerical Analysis 2000. Vol. VII: Partial Differential Equations, pp. 281–309. ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00516-1. URL: https://www.sciencedirect.com/science/article/pii/S0377042700005161.

[61] Ulrike Meier Yang. "Parallel Algebraic Multigrid Methods — High Performance Preconditioners". In: *Numerical Solution of Partial Differential Equations on Parallel Computers*. Ed. by Are Magnus Bruaset and Aslak Tveito. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 209–236. ISBN: 978-3-540-31619-0.

[62] Allison Baker et al. "Preparing Algebraic Multigrid for Exascale". In: (May 2015).

[63] Gérard Meurant. "A multilevel AINV preconditioner". In: *Numerical Algorithms* 29 (2002), pp. 107–129.

[64] Wei-Pai Tang and Wing Lok Wan. "Sparse Approximate Inverse Smoother for Multigrid". In: *SIAM Journal on Matrix Analysis and Applications* 21 (4 Jan. 2000), pp. 1236–1252. ISSN: 0895-4798. DOI: 10.1137/S0895479899339342.

[65] Wikipedia contributors. *Multigrid method - Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Multigrid_method&oldid=1301939943. Accessed: 2025-09-25. 2025.

[66] Jan van Gemert. *DL05 Optimization*. Feb. 2024.

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 25. 2012, pp. 1097–1105.

[68] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *arXiv preprint arXiv:1511.07289* (2015).

[69] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1026–1034.

[70] Hao Li et al. “Visualizing the Loss Landscape of Neural Nets”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 6389–6399.

[71] Bruce Christianson. “Automatic Hessians by Reverse Accumulation”. In: *IMA Journal of Numerical Analysis* 12.2 (1992), pp. 135–150.

[72] Jorge Nocedal. “Updating Quasi-Newton Matrices with Limited Storage”. In: *Mathematics of Computation* 35.151 (1980), pp. 773–782.

[73] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *arXiv preprint arXiv:1502.03167* (2015).

[74] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG]. URL: https://arxiv.org/abs/1609.02907.

[75] PyG Team. *torch_geometric.nn.conv.GCNConv — PyTorch Geometric Documentation*. https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.GCNConv.html. Accessed: 2025-10-08. 2025.

[76] Nino Shervashidze et al. “Weisfeiler-Lehman Graph Kernels”. In: *Journal of Machine Learning Research* 12.77 (2011), pp. 2539–2561. URL: http://jmlr.org/papers/v12/shervashidze11a.html.

[77] Keyulu Xu et al. *How Powerful are Graph Neural Networks?* 2019. arXiv: 1810.00826 [cs.LG]. URL: https://arxiv.org/abs/1810.00826.

[78] G.D. Weymouth and B. Font. “WaterLily.jl: A differentiable and backend-agnostic Julia solver for incompressible viscous flow around dynamic bodies”. In: *Computer Physics Communications* 315 (2025), p. 109748. DOI: 10.1016/j.cpc.2025.109748.

[79] Andrew P. Maertens and Gabriel D. Weymouth. “Accurate Cartesian-grid simulations of near-body flows at intermediate Reynolds numbers”. In: *Computer Methods in Applied Mechanics and Engineering* 283 (2015), pp. 106–129. DOI: 10.1016/j.cma.2014.09.007.

[80] M. Lauber, G. D. Weymouth, and G. Limbert. “Immersed boundary simulations of flows driven by moving thin membranes”. In: *Journal of Computational Physics* (2022), p. 111076. DOI: 10.1016/j.jcp.2022.111076.

[81] Florent Bonnet et al. *AirfRANS: High Fidelity Computational Fluid Dynamics Dataset for Approximating Reynolds-Averaged Navier-Stokes Solutions*. 2023. arXiv: 2212.07564 [cs.LG]. URL: https://arxiv.org/abs/2212.07564.

[82] Jian Du et al. *Topology Adaptive Graph Convolutional Networks*. 2018. arXiv: 1710.10370 [cs.LG]. URL: https://arxiv.org/abs/1710.10370.

[83] Jingzhao Zhang et al. *Why gradient clipping accelerates training: A theoretical justification for adaptivity*. 2020. arXiv: 1905.11881 [math.OC]. URL: https://arxiv.org/abs/1905.11881.

[84] Víctor Cayetano Hernández Sánchez. “PIV-based surface pressure reconstruction using Physics-Informed Neural Networks”. Accessed: 2025-10-18. MA thesis. Delft, The Netherlands: Delft University of Technology, 2025. URL: https://repository.tudelft.nl/file/File_afcf1f9b-2f68-4854-a42a-96b176610de7?preview=1.

[85] Liu Yang and Jian Yang. “A multi-colored Gauss-Seidel solver for aerodynamic simulations of a transport aircraft model on graphics processing units”. In: *Advances in Aerodynamics* 7 (1 Dec. 2025), p. 8. ISSN: 2097-3462. DOI: 10.1186/s42774-024-00200-5.

A

Extra methodology

A.1. Tools and computational resources

This research involved several tools and two main workstations to carry out the data generation, training, and evaluation of the AMG-GNN model.

Regarding the computational resources, the main work was performed using an Aorus 15 XE5, featuring the following specifications:

- **CPU**: 12th Gen Intel(R) Core(TM) i7-12700H (2.30 GHz)
- **GPU**: NVIDIA GeForce RTX 3070Ti 8GB
- **RAM**: 16.0 GB dedicated memory
- **OS**: Windows 11 Home 24H2

Nonetheless, during the second half of the period, access to a workstation was granted, which allowed training on larger problems and datasets, while doing so faster. The specifications of the server are:

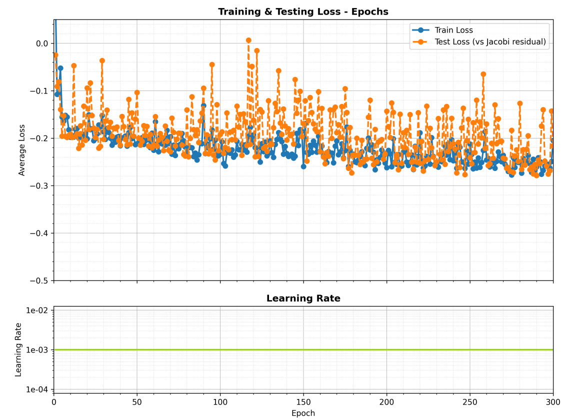
- **CPU**: Intel(R) Xeon(R) W-2275 CPU @ 3.30GHz
- **GPU**: NVIDIA Quadro RTX 8000 48GB
- **RAM**: 128.0 GB dedicated memory
- **OS**: Linux Ubuntu 22.04.5 LTS

Regarding the software used, the generation of structured data was performed using Julia 1.8 and WaterLily 0.2.3, while for unstructured data, ReFRESCO was employed. Concerning the training and evaluation of the model, the critical tools were PyTorch 2.6.0 (CUDA 11.8) and PyTorch Geometric 2.6.1.

A.2. Training setup examples

A.2.1. Adam and AdamW

(a) Adam.



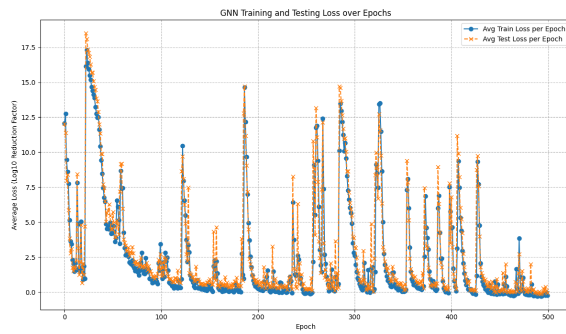
(b) AdamW ($w_d = 1 \times 10^{-5}$).

Figure A.2.1: Loss history for two different optimizers.

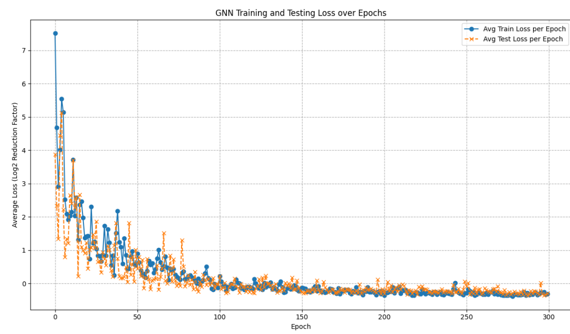
B

GNN Optimization results

B.1. Initial tests

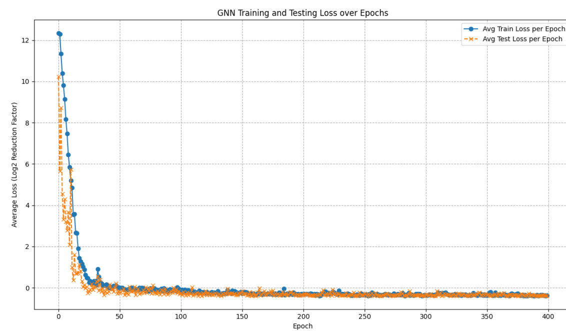


(a) HD32 + No gradient clipping.

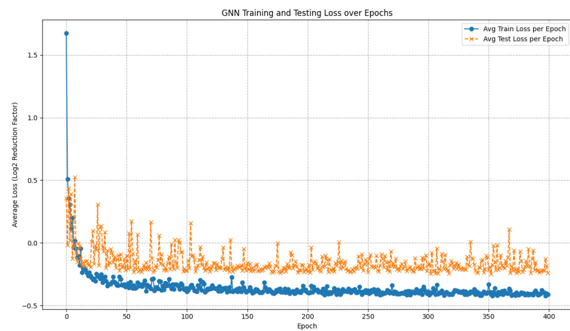


(b) HD64.

Figure B.1.1: First two training on different hidden dimensions.

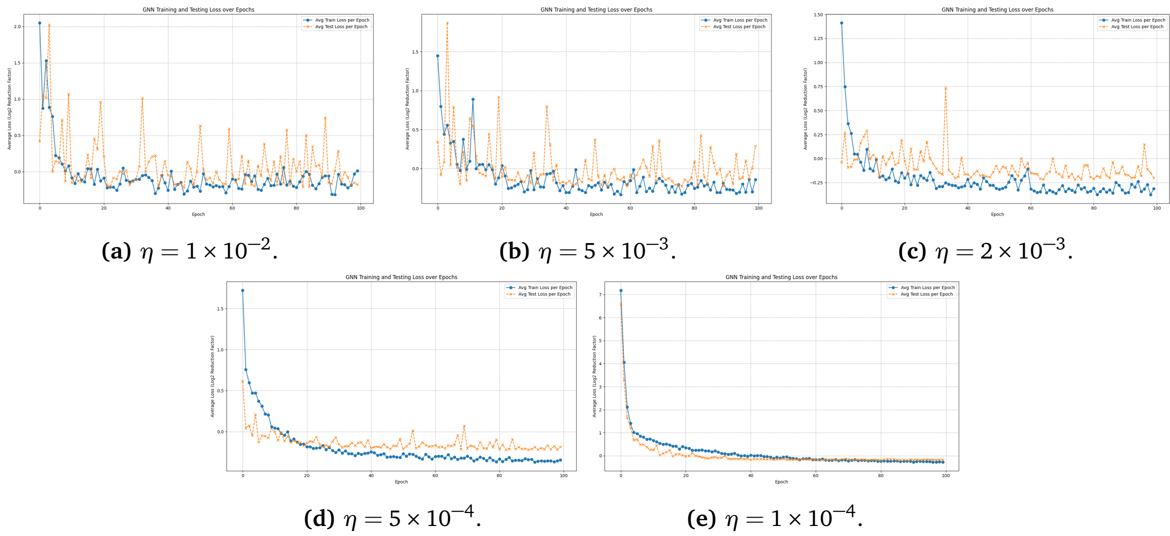


(a) Dropout 20%.



(b) Sigmoid activation function.

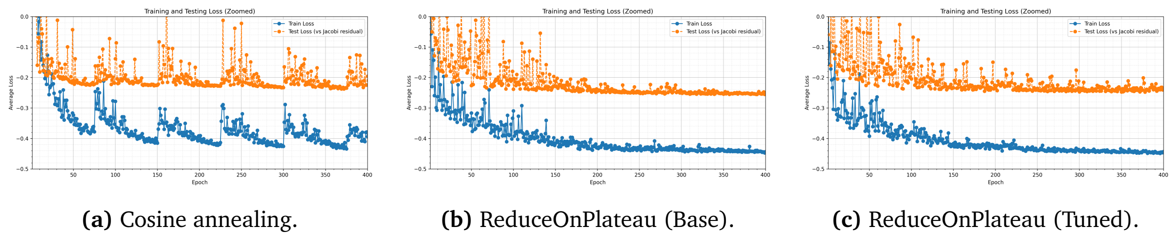
Figure B.1.2: Effect on training of two more changes maintaining HD64.



(a) $\eta = 1 \times 10^{-2}$. **(b)** $\eta = 5 \times 10^{-3}$. **(c)** $\eta = 2 \times 10^{-3}$.

(d) $\eta = 5 \times 10^{-4}$. **(e)** $\eta = 1 \times 10^{-4}$.

Figure B.1.3: Effect of five different learning rates on training.



(a) Cosine annealing. **(b)** ReduceOnPlateau (Base). **(c)** ReduceOnPlateau (Tuned).

Figure B.1.4: Effect on training of three schedulers. The learning rate history was not implemented at this point.

B.1.1. Model evaluation

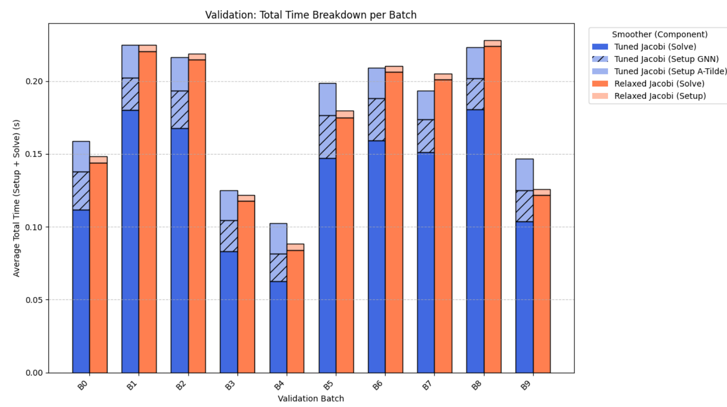
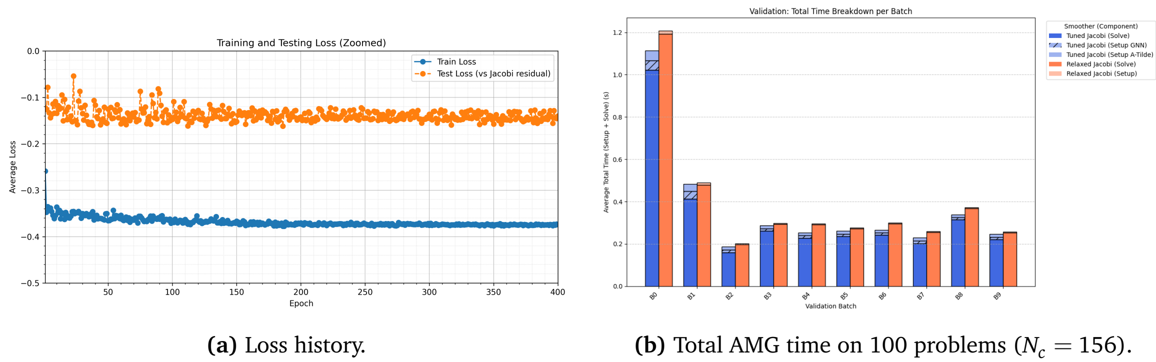


Figure B.1.5: Total time to converge (Setup + Solve) for AMG-GNN and AMG-Jacobi.



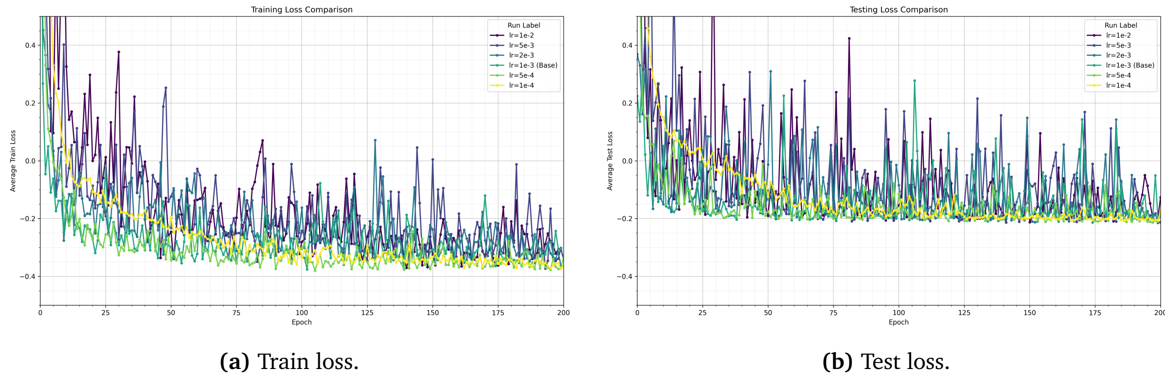
(a) Loss history.

(b) Total AMG time on 100 problems ($N_c = 156$).

Figure B.1.6: Effect on training of two more changes maintaining HD64.

B.2. Hyperparameter study

B.2.1. Learning rate



(a) Train loss. **(b)** Test loss.

Figure B.2.1: Loss history for the different learning rates.

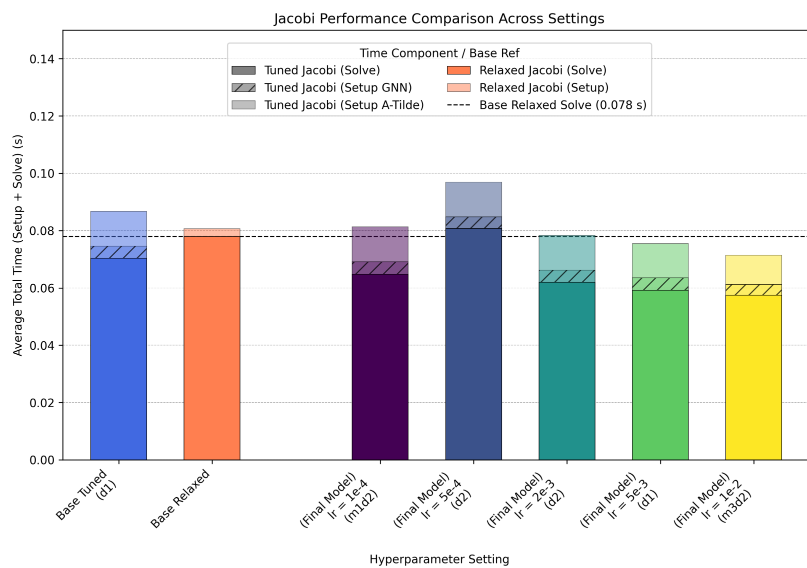
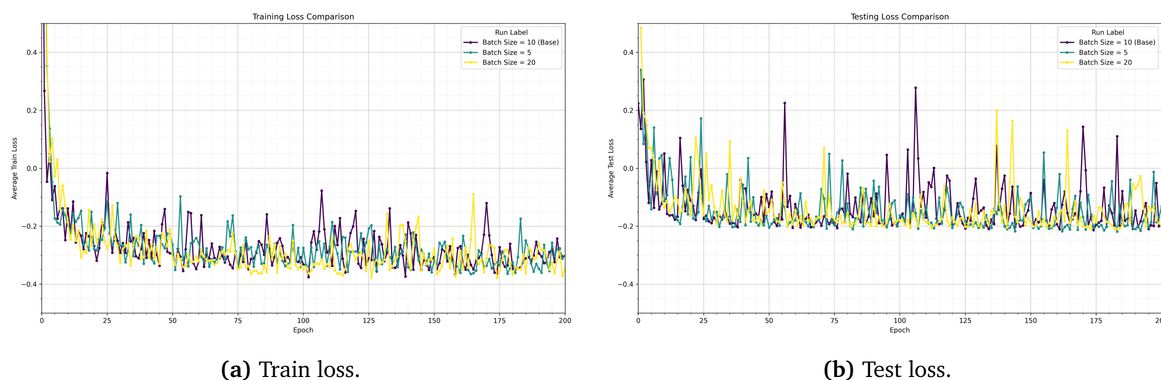


Figure B.2.2: Total times of diverse configurations of learning rates. The validation dataset is 50 problems with $N_c = 28$.

B.2.2. Mini-batch size



(a) Train loss. **(b)** Test loss.

Figure B.2.3: Loss history for the different batch sizes.

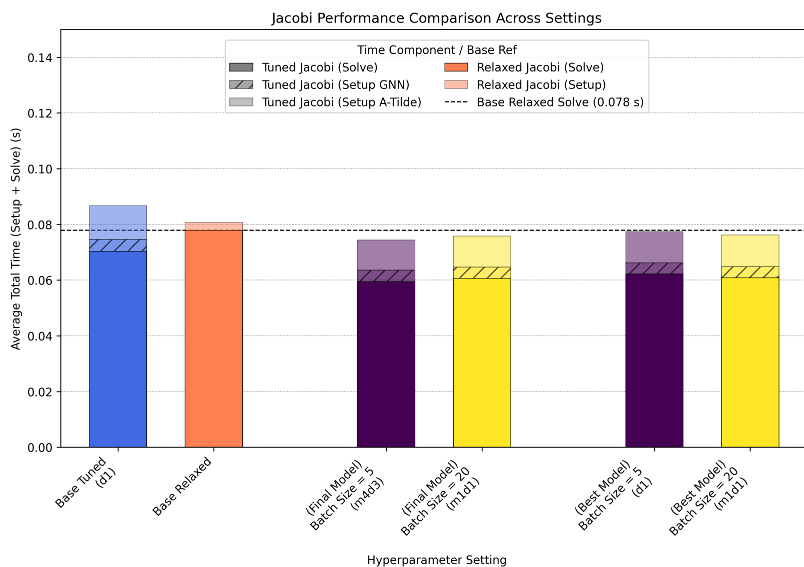
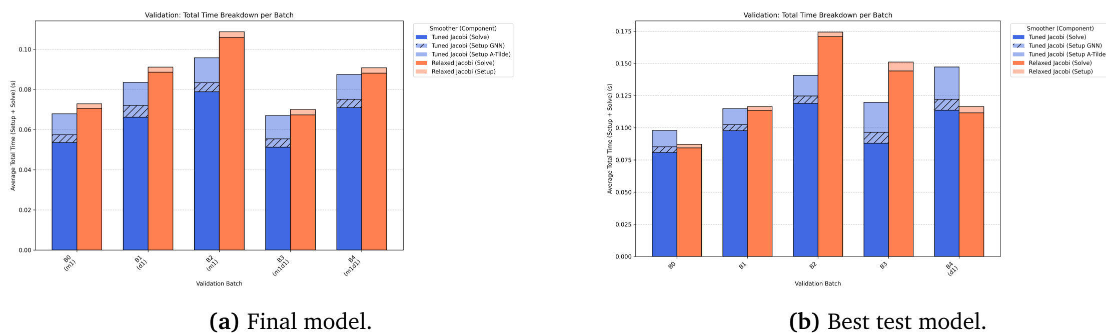


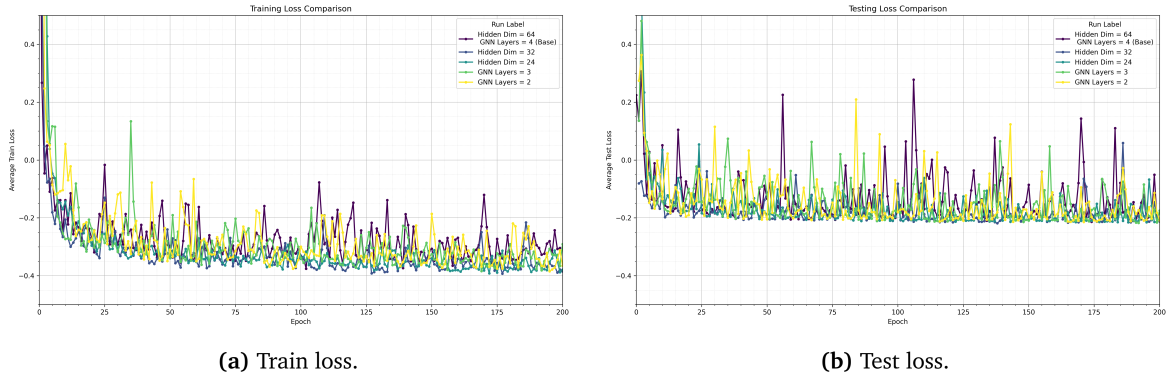
Figure B.2.4: Total times of diverse configurations of batch sizes. The validation dataset is 50 problems with $N_c = 28$.



(a) Final model. **(b)** Best test model.

Figure B.2.5: Total times for $B = 5$ for models saved at different epochs.

B.2.3. Model size



(a) Train loss. **(b)** Test loss.

Figure B.2.6: Loss history for the different model sizes.

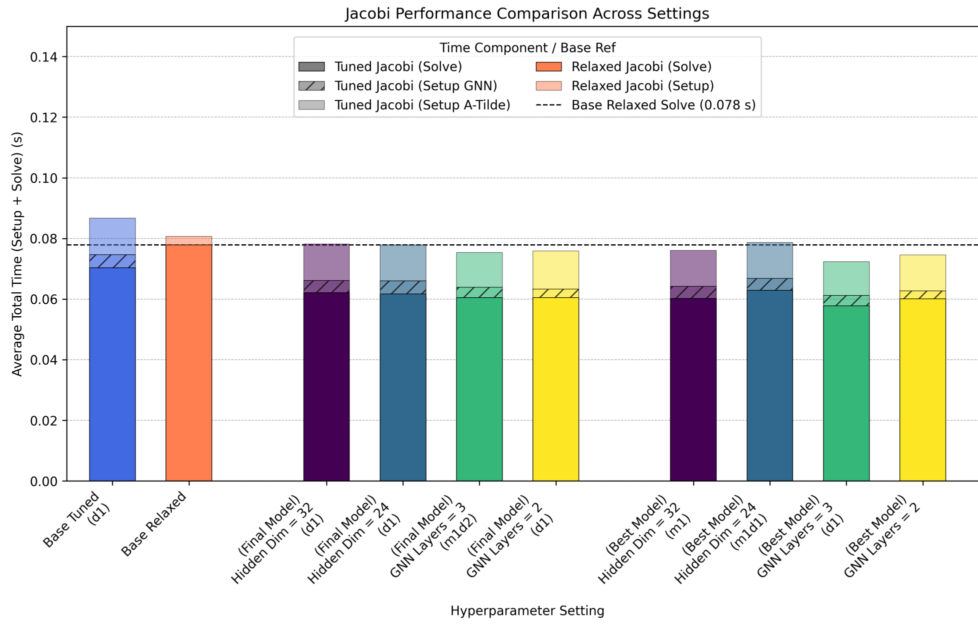
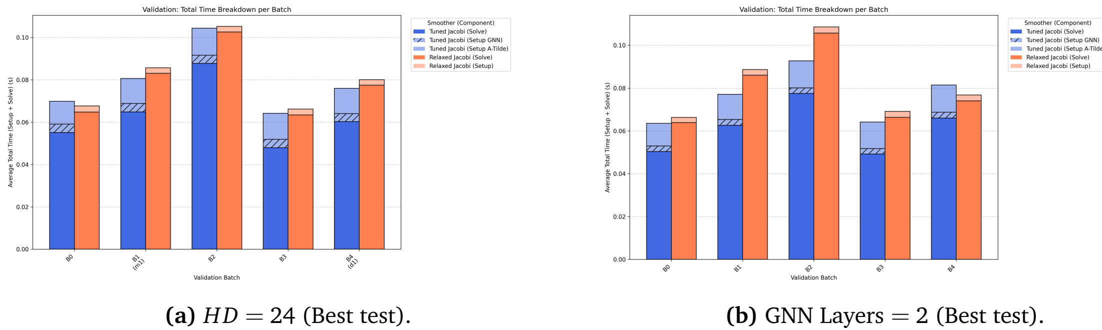


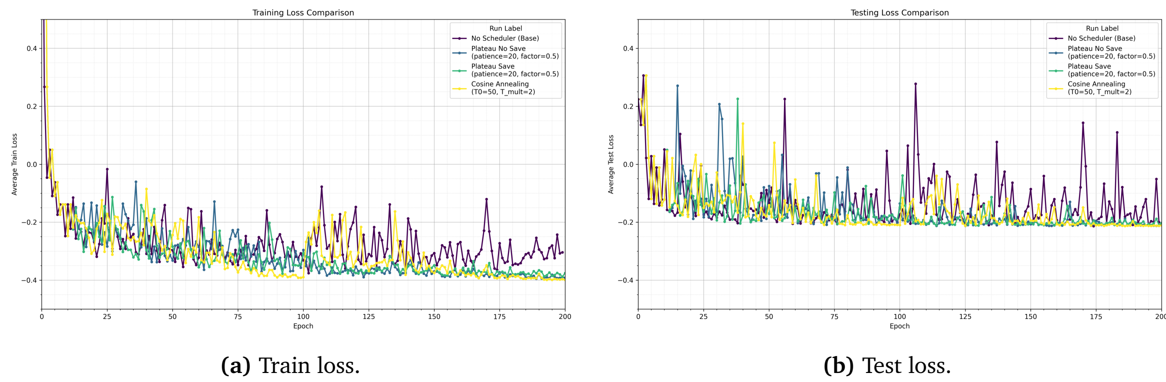
Figure B.2.7: Total times of diverse configurations of model sizes. The validation dataset is 50 problems with $N_c = 28$.



(a) $HD = 24$ (Best test). **(b)** GNN Layers $= 2$ (Best test).

Figure B.2.8: Total times for $B = 5$ for models saved at different epochs.

B.2.4. Scheduler



(a) Train loss.

(b) Test loss.

Figure B.2.9: Loss history for the different schedulers.

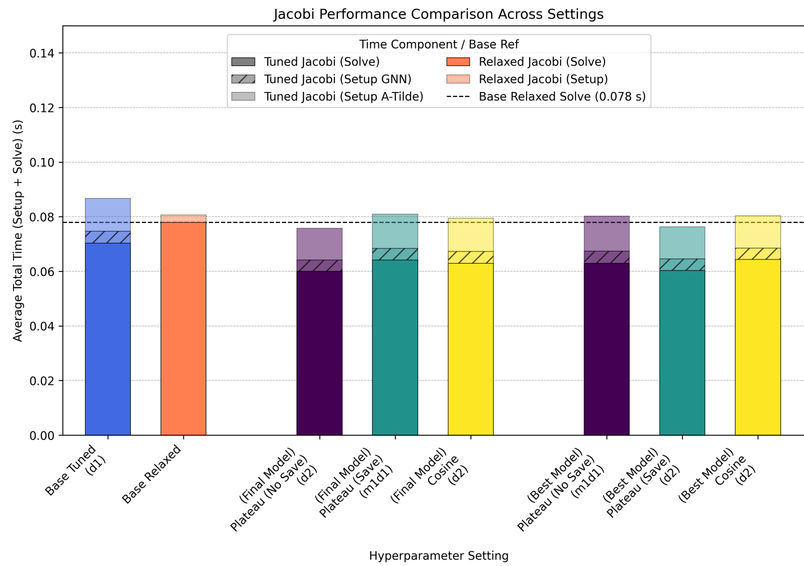
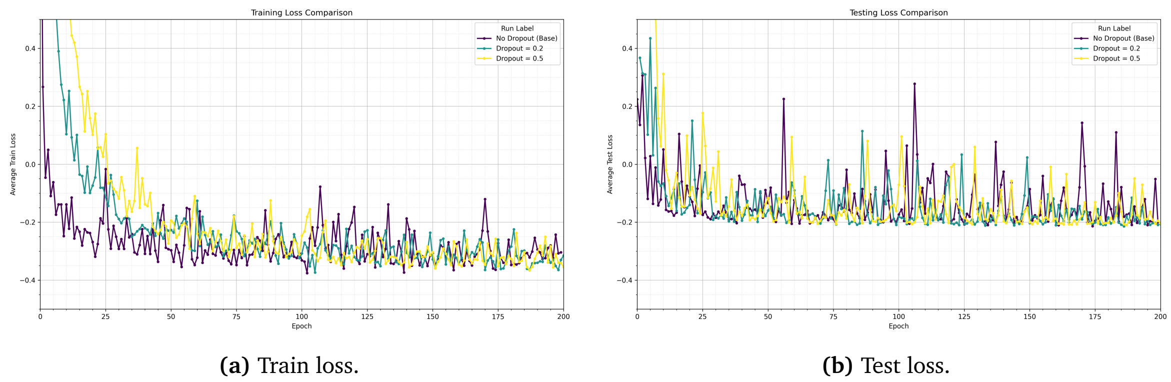


Figure B.2.10: Total times of diverse configurations of schedulers. The validation dataset is 50 problems with $N_c = 28$.

B.2.5. Dropout



(a) Train loss. **(b)** Test loss.

Figure B.2.11: Loss history for the different dropouts.

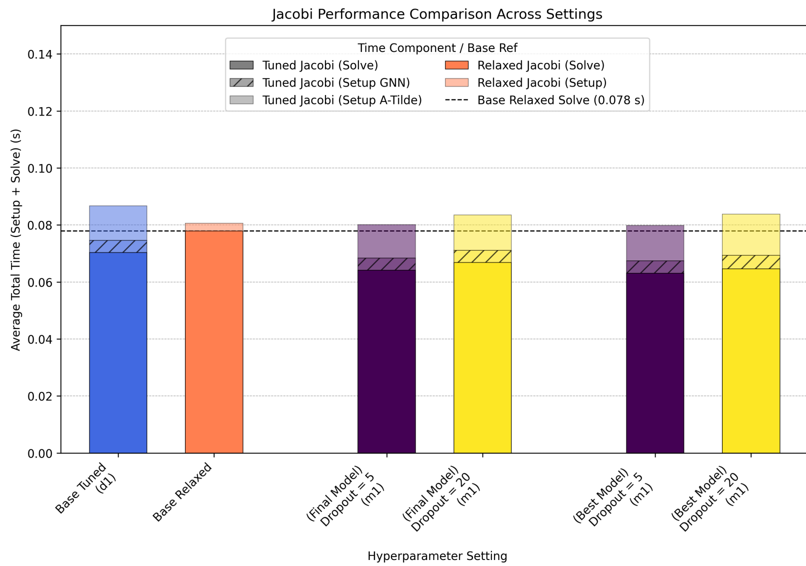
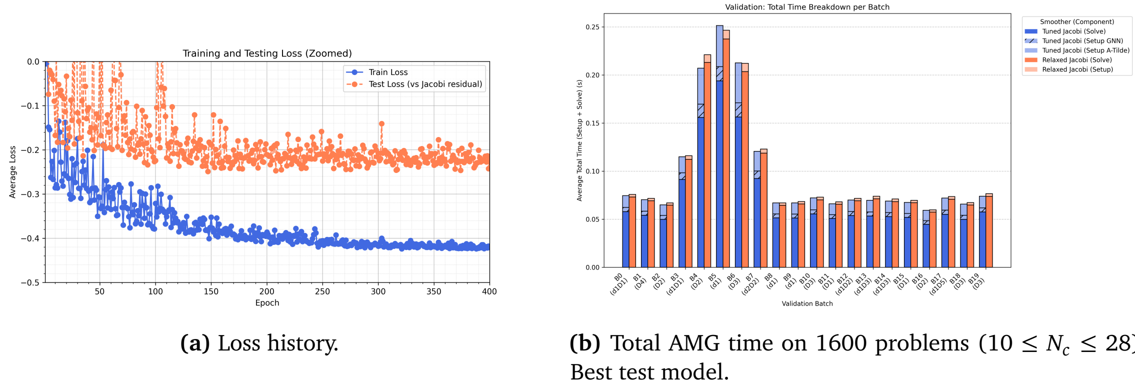


Figure B.2.12: Total times of diverse configurations of dropout. The validation dataset is 50 problems with $N_c = 28$.

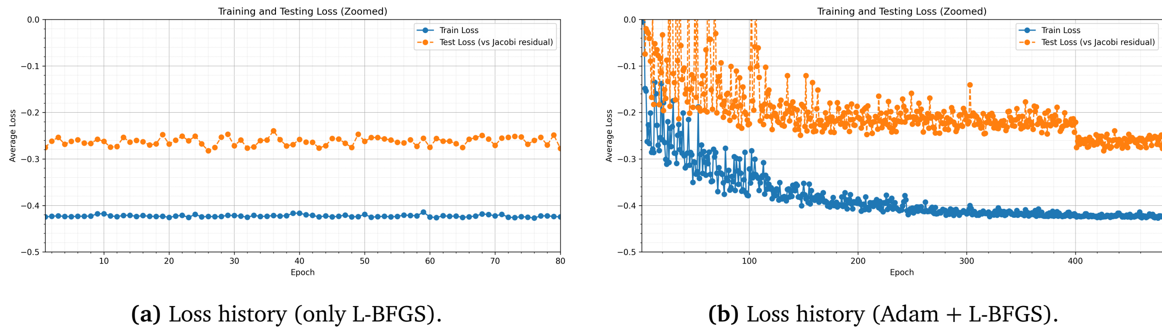
B.2.6. L-BFGS



(a) Loss history.

(b) Total AMG time on 1600 problems ($10 \leq N_c \leq 28$). Best test model.

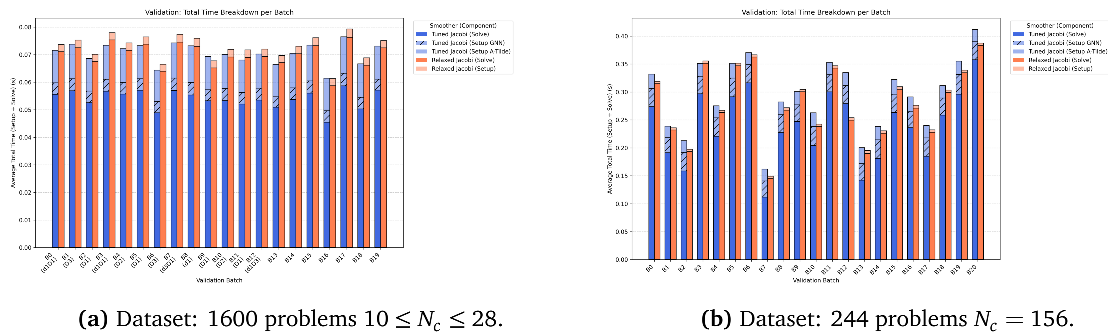
Figure B.2.13: Training and validation results pre-L-BFGS.



(a) Loss history (only L-BFGS).

(b) Loss history (Adam + L-BFGS).

Figure B.2.14: Training and validation results for L-BFGS.



(a) Dataset: 1600 problems $10 \leq N_c \leq 28$.

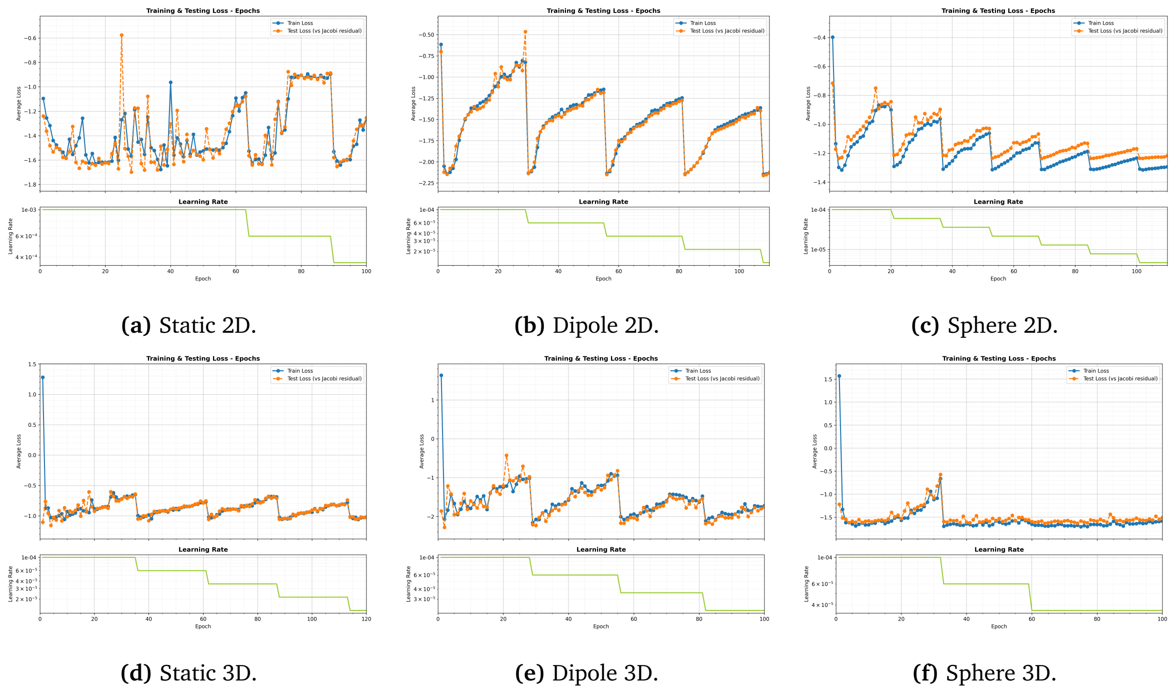
(b) Dataset: 244 problems $N_c = 156$.

Figure B.2.15: Validation results (total time) for different datasets.

C

Structured generalization

C.1. Synthetic 2D/3D

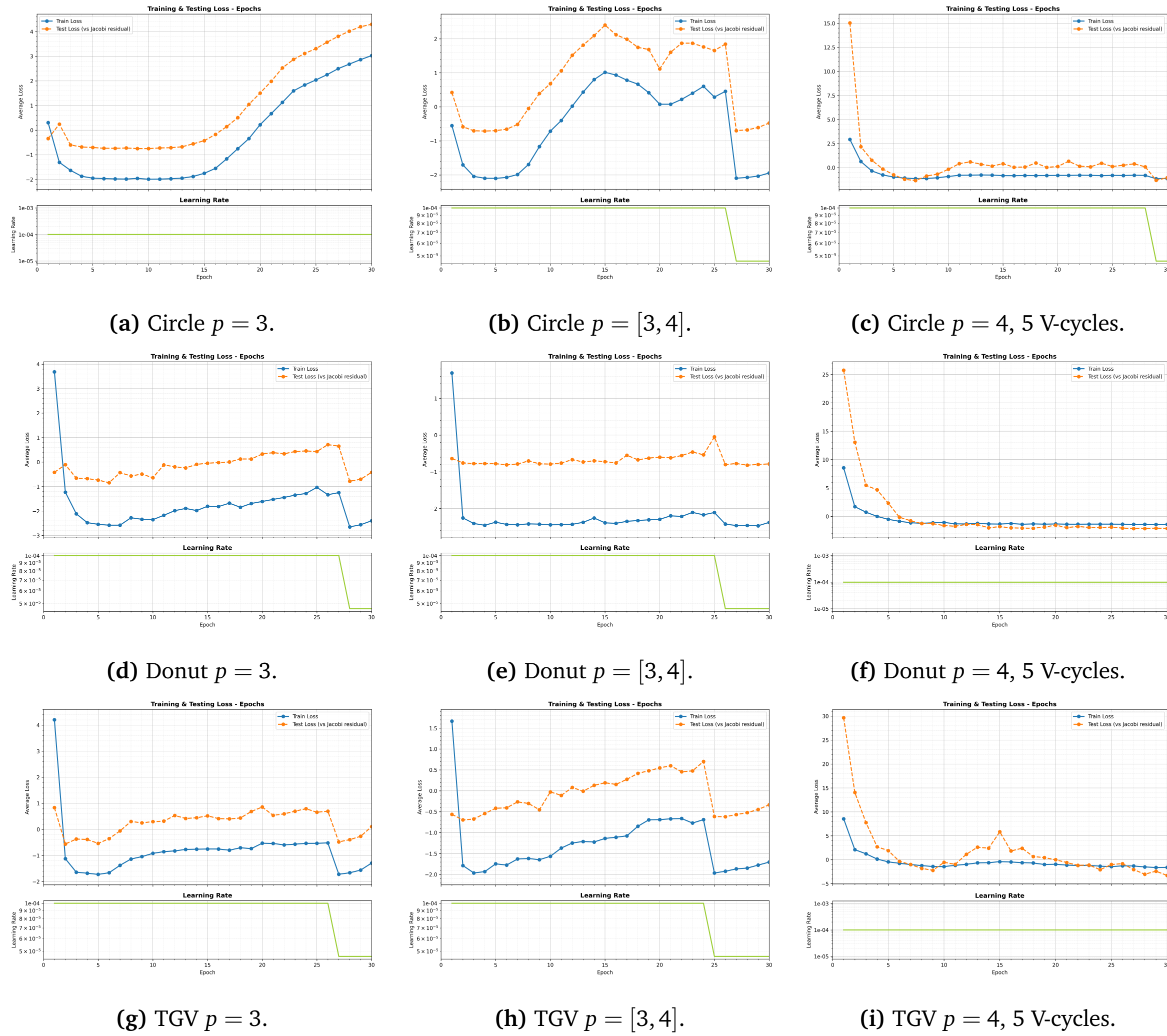


(a) Static 2D. **(b)** Dipole 2D. **(c)** Sphere 2D.

(d) Static 3D. **(e)** Dipole 3D. **(f)** Sphere 3D.

Figure C.1.1: Train and test losses for different types of synthetic data. The dataset is a 200/50 split for $n = 32^2$ (2D) and $n = 16^3$ (3D).

C.2. Unsteady



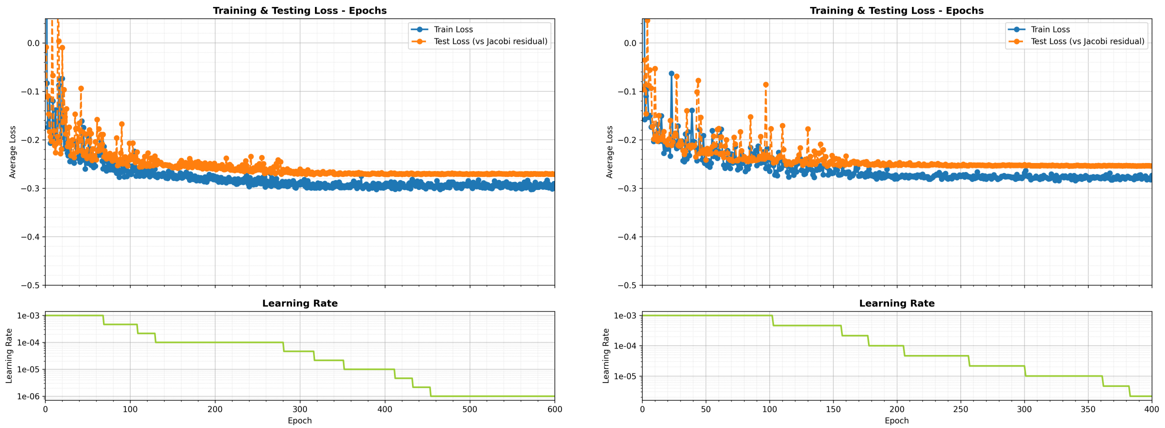
(a) Circle $p = 3$.
(b) Circle $p = [3, 4]$.
(c) Circle $p = 4$, 5 V-cycles.
(d) Donut $p = 3$.
(e) Donut $p = [3, 4]$.
(f) Donut $p = 4$, 5 V-cycles.
(g) TGV $p = 3$.
(h) TGV $p = [3, 4]$.
(i) TGV $p = 4$, 5 V-cycles.

Figure C.2.1: Loss history of unsteady problems for different setups: circle (a,b,c), donut (d,e,f), and TGV (g,h,i) for $p = 3$, combination of $p = [3, 4]$, and $p = 4$ with multi-v-cycle loss.

D

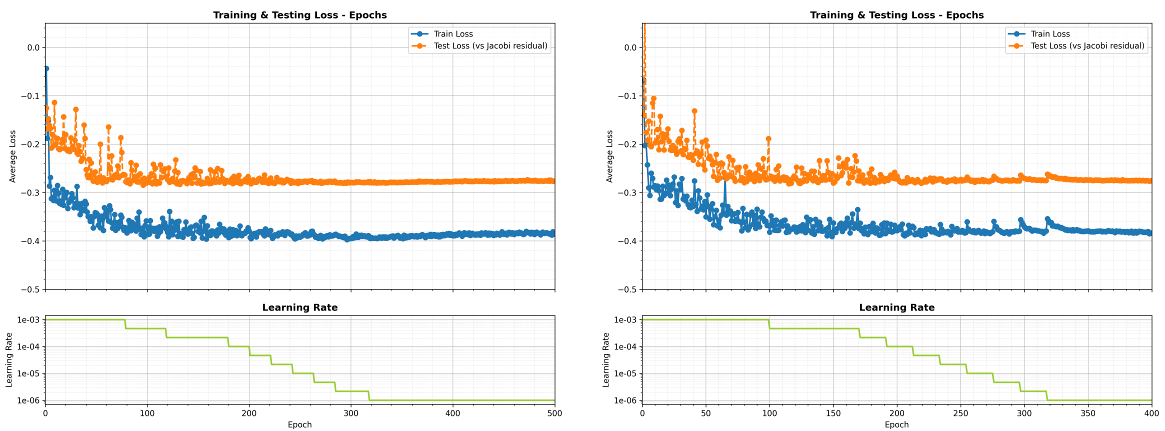
Unstructured generalization

D.1. Impact of data diversity



(a) First train. **(b)** Second train.

Figure D.1.1: Train and test loss history for two Poiseuille-only trainings.



(a) First train. **(b)** Second train.

Figure D.1.2: Train and test loss history for union trainings.

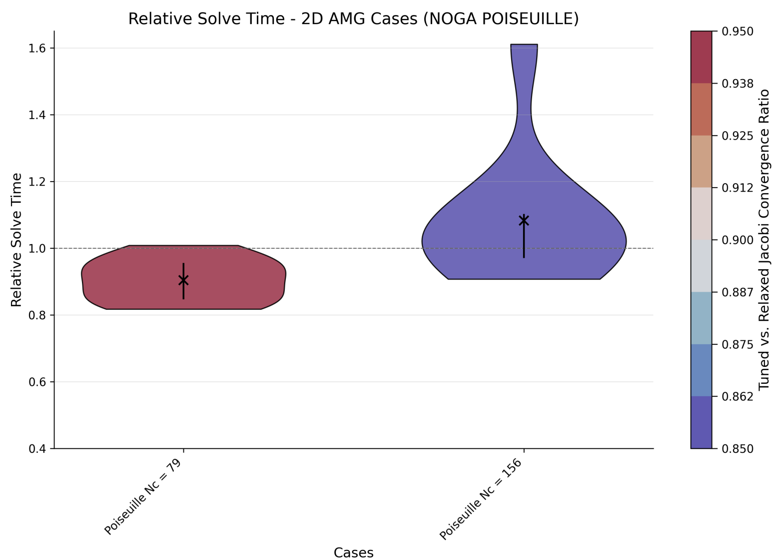


Figure D.1.3: Relative solve time to relaxed Jacobi for 10 models trained on Poiseuille only. Evaluated on Poiseuille datasets of size $N_c = 79$ and $N_c = 111$.

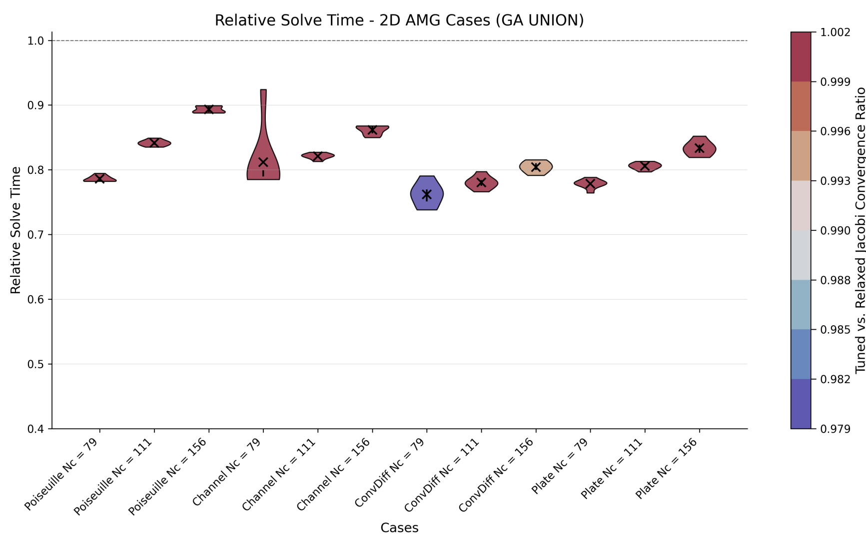
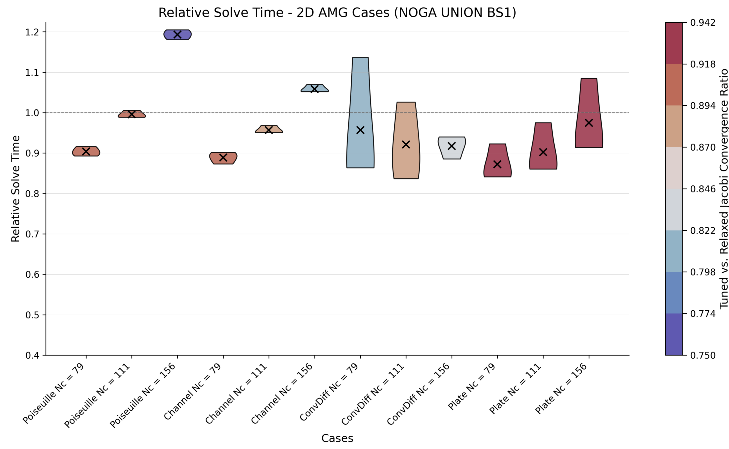
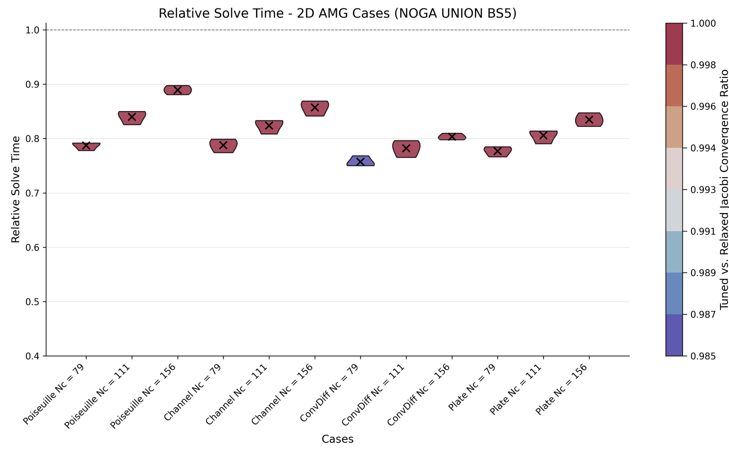


Figure D.1.4: Relative solve time to relaxed Jacobi for 5 models trained on the union dataset without global attributes. Evaluated on datasets of size $N_c = 79, 111$ and 159.

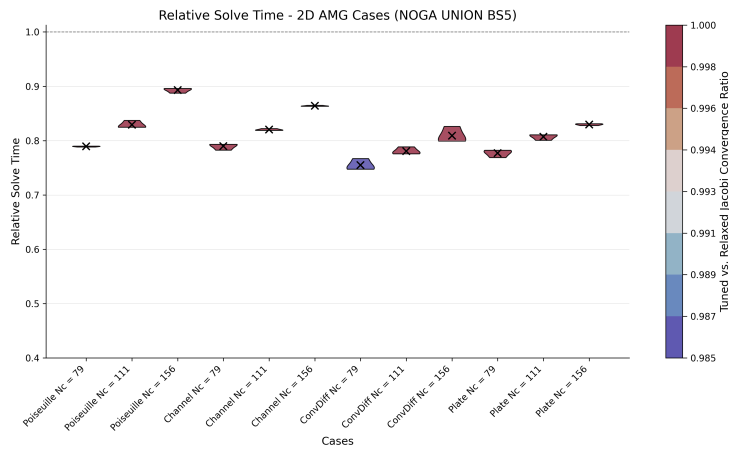
D.2. Effect of mini-batch size



(a) $B = 1$.



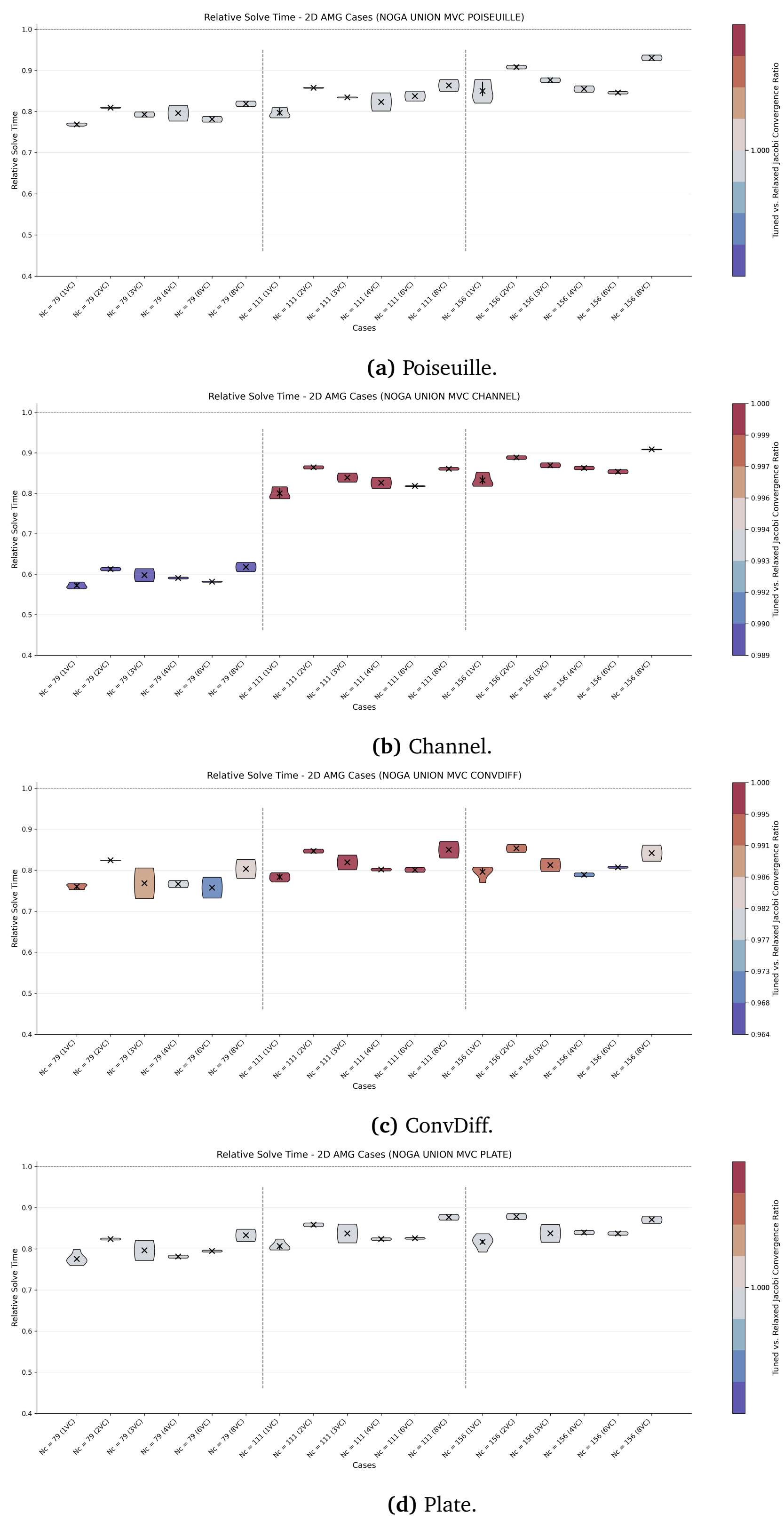
(b) $B = 5$.



(c) $B = 10$.

Figure D.2.1: Relative solve time to relaxed Jacobi for 3 models trained on the union dataset (up to $N_c = 40$) and $B = 1, 5, 10$. Evaluated on datasets of size $N_c = 79, 111$ and 159.

D.3. Influence of multi-V-cycle training



(a) Poiseuille.

(b) Channel.

(c) ConvDiff.

(d) Plate.

Figure D.3.1: Relative solve time to relaxed Jacobi for 6 models trained on the union dataset and different V-cycles. Evaluated on datasets of size $N_c = 79, 111$ and 159 and $\delta = 10^{-4}$.

D.4. Enhanced dataset problems

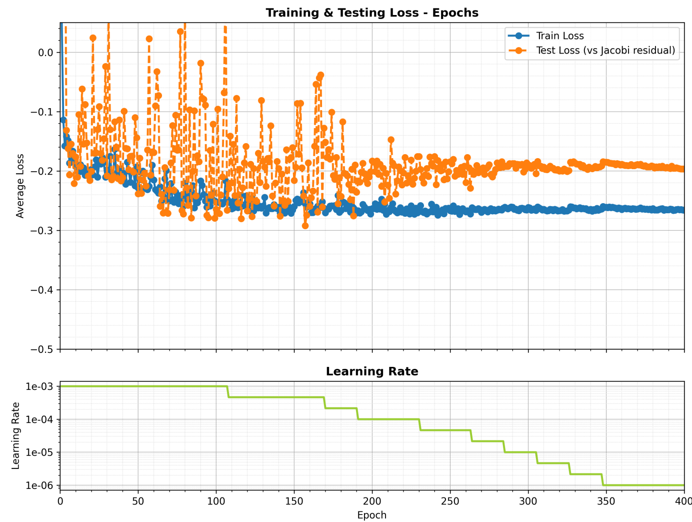


Figure D.4.1: Train history for train dataset including $N_c = 56$, using $B = 6$ and FP32.

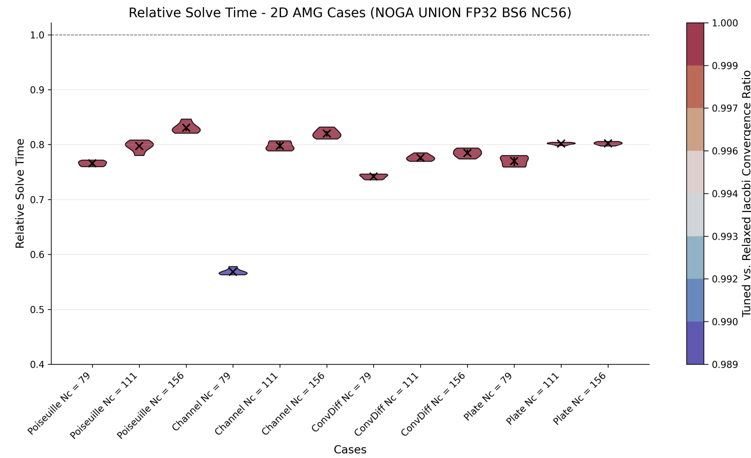


Figure D.4.2: Relative solve time to relaxed Jacobi for 5 models trained on the union dataset up to $N_c = 56$ ($B = 6$, FP32). Evaluated on datasets of size $N_c = 79, 111$ and 159 and $\delta = 10^{-4}$.

D.5. Non-convergence classification

In relation to the non-convergence classification, the initial method worked as follows: if any smoother did not converge for a determined problem, the associated time/iteration metric was not included in the average. This was done individually. That is, if one converged and the other did not, only the non-converged metric was removed. Therefore, if the number of cases that converge is high and similar between the two, the results will adequately represent the performance of the model. However, when a large disparity is present in the number of converged cases between smoothers, then this approach can under- or overestimate the performance of one of them. To explain this, note why each of the different non-convergence types can be obtained:

- **Maximum iterations**. Relaxed and tuned Jacobi can reach the threshold of iterations due to the problem being too complex or the initial solution not being adequate. However, tuned Jacobi can reach this state differently. The smoother generates NaNs in the solution in the first iteration, and for safety measures, the coarse correction is set to zero. Hence, for each V-cycle, the coarse correction is reset, and the same error occurs (i.e., technically a divergence camouflaged as max. iterations).
- **Stagnation**. This is the same for both smoothers. It happens mainly when FP32 is used instead of FP64. The end result is a residual norm that is constant within a relative tolerance.
- **Divergence**. Also equal for both smoothers. This occurs when the residual norm increases too fast after some iterations.

Accordingly, if tuned Jacobi diverges way more than relaxed Jacobi, it generally is due to the problems being complex and the GNN not being able to choose the appropriate polynomial coefficients. When this happens, the metrics are removed from tuned AMG but not for relaxed Jacobi. The latter, on average, will have larger solve times due to not removing these tougher cases. This puts the tuned Jacobi at an effective and unfair advantage **which is only noticeable in the three-dimensional validation dataset** and on the most complex dataset ($N_c = 24$).

The new logic is that if tuned Jacobi does not converge, relaxed Jacobi metrics are removed with it. Whereas if relaxed Jacobi does not converge while tuned does, it only removes relaxed Jacobi. Further, reaching maximum iterations is no longer considered non-convergence for relaxed Jacobi. This seems to put relaxed Jacobi metrics at a disadvantage, but the following points demonstrate that it is not the case.

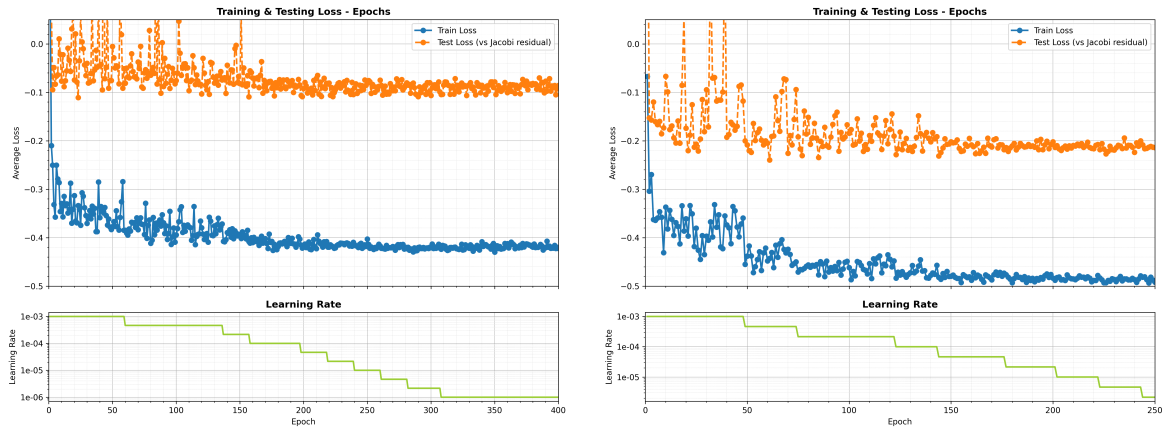
First, with respect to maximum iterations. If relaxed Jacobi reaches maximum iterations, it is a result of the smoother not being effective enough. While, if tuned, Jacobi reaches maximum iterations, it is because it diverged or performance was not good enough within the iteration margin. It should be noted that in virtually all cases in which both converged, tuned Jacobi always had a lower number of iterations. Hence, if the flag from tuned Jacobi is triggered, it is always correct to remove relaxed Jacobi. On the contrary, removing tuned Jacobi for a flag triggered in relaxed Jacobi only hinders performance from tuned Jacobi, as it basically always had a lower number of iterations.

Regarding the stagnation, almost any problem presented stagnation that was not due to using FP32 instead of FP64 (which happens for both tuned and relaxed). Similar to the previous scenario, if one smoother stagnates, it is generally accompanied by the other. However, if only the relaxed Jacobi is being removed, the tuned Jacobi is precisely seeing worse performance, so no advantage for relaxed here.

In terms of residual divergence, out of more than a thousand validation cases, if relaxed Jacobi diverged, it is a result of the problem being too complex to solve, and tuned Jacobi also diverged. Only one or two problems, if any, presented divergence only for relaxed Jacobi. Consequently, this

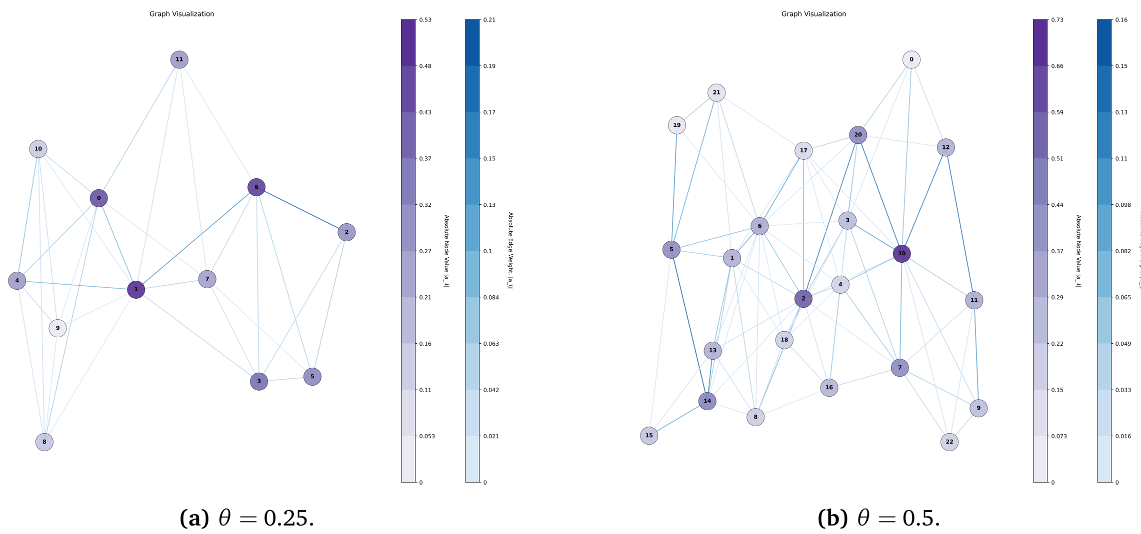
would put tuned Jacobi at an unfavorable position, not otherwise, given that the tough problem metric would be saved only for tuned Jacobi.

D.6. 3D-only dataset



(a) FP32. **(b)** FP64.

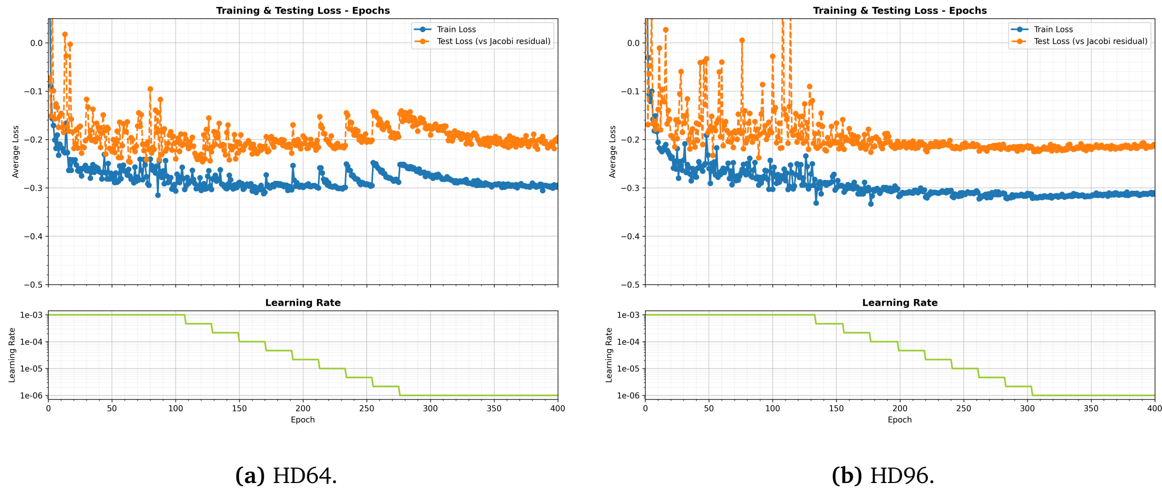
Figure D.6.1: Train and test loss history for two Poiseuille-only trainings.



(a) $\theta = 0.25$. **(b)** $\theta = 0.5$.

Figure D.6.2: Effect of modifying the strength parameter on the sparsity (and size) of the coefficient matrix. Graphs $G^{(3)}$ are presented to illustrate the change in nodes.

D.7. Mixed dataset



(a) HD64. **(b)** HD96.

Figure D.7.1: Train and test losses for different types of unstructured data. The dataset is a 700/300 split of 2D/3D problems.

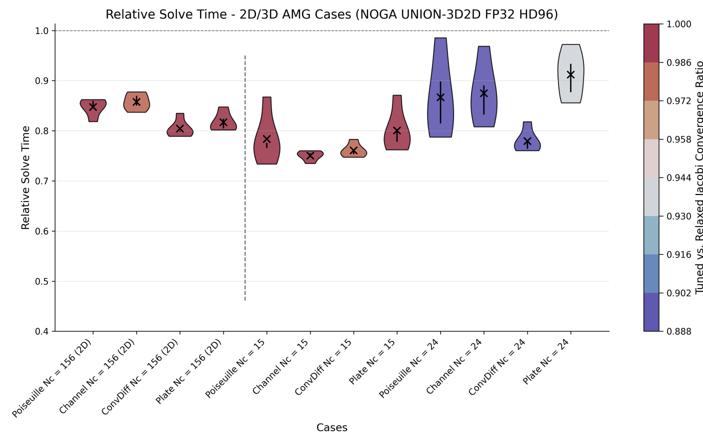
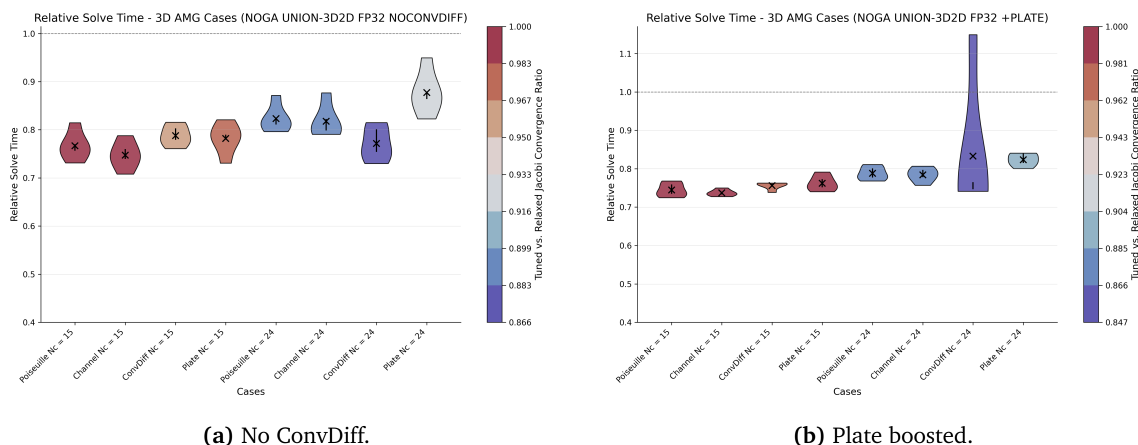


Figure D.7.2: Relative solve times for 5 trained models with HD96. Trained and validated in 2D and 3D.



(a) No ConvDiff.

(b) Plate boosted.

Figure D.7.3: Relative solve times for 5 trained models with different datasets. Validated in $N_c = 15$ and 24 (3D).

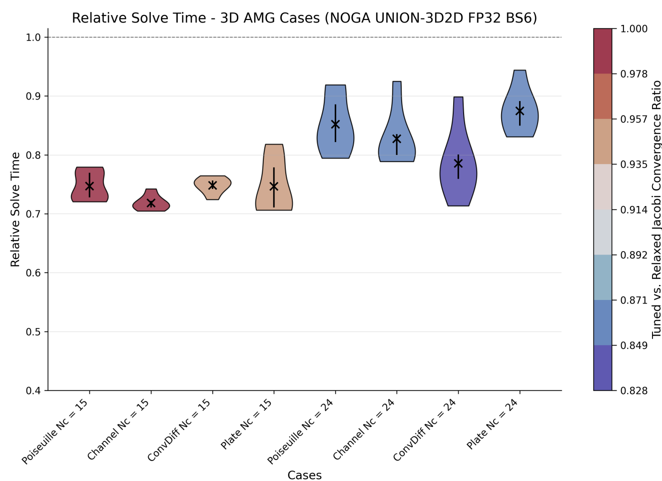
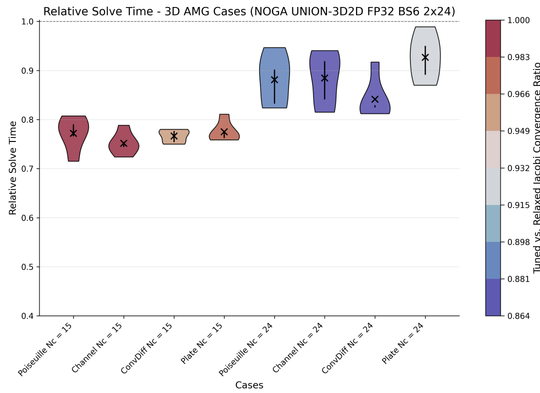
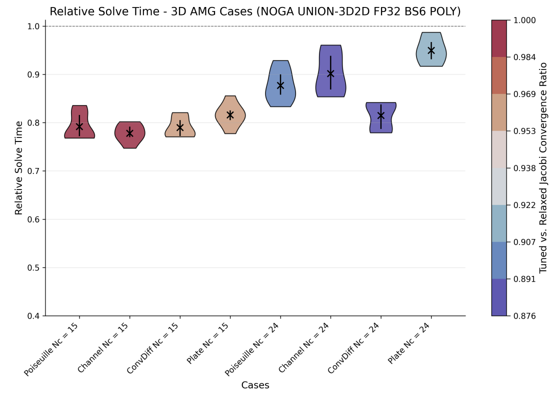


Figure D.7.4: Relative solve times for 7 trained models with $B = 6$. Validated in 3D.

The multi-polynomial pseudo-inverse is based on the idea of 4-color Gauss-Seidel [85]. In this respect, the polynomial coefficients vary by a *color class* (e.g., 2/3/4 classes), chosen from the row/column indices of the sparse matrix. Hence, each color has an associated diagonal and off-diagonal sets of polynomial coefficients. This method tries to give the model more freedom to select appropriate coefficients that can provide further acceleration at the cost of overfitting.



(a) GNN size 2×24.



(b) 4-color pseudo-inverse.

Figure D.7.5: Relative solve times for 5 trained models with model changes. Validated in $N_c = 15$ and 24 (3D).