Scalable GNN Solutions for CFD Simulations

Matthieu Nastorg

université PARIS-SACLAY

Scalable GNN Solutions for CFD Simulations

Solutions GNN évolutives pour les simulations CFD

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique Mathématique
Graduate School : Informatique et sciences du numérique.
Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Laboratoire interdisciplinaire des sciences du numérique** (Université Paris-Saclay, CNRS), sous la direction de **Marc SCHOENAUER**, directeur de recherche, le co-encadrement de **Michele-Alessandro BUCCI**, ingénieur de recherche, et **Guillaume CHARPIAT**, chargé de recherche

Thèse soutenue à Paris-Saclay, le 15 Avril 2024, par

Matthieu NASTORG

Composition du jury

Membres du jury avec voix délibérative

Alexandre ALLAUZEN Professeur, EPSCI et Université Paris-Dauphine PSL	Président
Victorita DOLEAN Professeure, Université technologique d'Eindhoven	Rapporteure & Examinatrice
Elie HACHEM Professeur, Mines Paris PSL	Rapporteur & Examinateur
Patrick GALLINARI Professeur, Sorbonne Université, Paris	Examinateur
Augustin PARRET-FRÉAUD Ingénieur de recherche HPC, Safran	Examinateur

THESE DE DOCTORAT
NNT : 2024UPASG020

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Solutions GNN évolutives pour les simulations CFD

Mots clés: Graph Neural Networks, CFD, Equation de Poisson, Décomposition de Domaine

Résumé: La Dynamique des Fluides Numérique (CFD) joue un rôle essentiel dans la prédiction de divers phénomènes physiques, tels que le climat, l'aérodynamique ou la circulation sanguine. Au coeur de la CFD se trouvent les équations de Navier-Stokes régissant le mouvement des fluides. Cependant, résoudre ces équations à grande échelle reste fastidieux, en particulier lorsqu'il s'agit des équations de Navier-Stokes incompressibles, qui nécessitent la résolution intensive d'un problème de Poisson de Pression, garantissant la contrainte d'incompressibilité. De nos jours, les méthodes d'apprentissage profond ont ouvert de nouvelles perspectives pour améliorer les simulations numériques. Parmi ces approches, les Graph Neural Networks (GNNs), conçus pour traiter des données de type graphe tels que les maillages, se sont révélés prometteurs. Cette thèse vise à explorer l'utilisation des GNNs pour améliorer la résolution du problème de Poisson de Pression. Une contribution clé implique l'introduction d'une nouvelle architecture GNN qui respecte intrinsèquement les conditions aux limites tout en exploitant la théorie des couches implicites pour ajuster automatiquement le nombre de couches GNN nécessaires à la convergence : ce nouveau modèle présente des capacités de généralisation améliorées, gérant efficacement des problèmes de Poisson de différentes tailles et formes. Néanmoins, ses limitations actuelles le restreignent aux problèmes à petite échelle, insuffisants pour les applications industrielles qui nécessitent souvent plusieurs milliers de noeuds. Pour mettre à l'échelle ces modèles, cette thèse explore la combinaison des GNNs avec les méthodes de Décomposition de Domaines, tirant parti des calculs en parallèle sur GPU pour produire des solutions d'ingénierie plus efficaces.

Title: Scalable GNN Solutions for CFD Simulations

Keywords: Graph Neural Networks, CFD, Poisson Equation, Domain Decomposition

Abstract: Computational Fluid Dynamics (CFD) plays an essential role in predicting various physical phenomena, such as climate, aerodynamics, or blood flow. At the core of CFD lie the Navier-Stokes equations governing the motion of fluids. However, solving these equations at scale remains daunting, especially when dealing with Incompressible Navier-Stokes equations. Indeed, the well-known splitting schemes require the costly resolution of a Pressure Poisson problem that guarantees the incompressibility constraint. Nowadays, Deep Learning methods have opened new perspectives for enhancing numerical simulations. Among existing approaches, Graph Neural Networks (GNNs), designed to handle graph data like meshes, have proven to be promising. This thesis is dedicated to exploring the use of GNNs to enhance the resolution of the Pressure Poisson problem. One significant contribution involves introducing a novel physics-informed GNN-based model that inherently respects boundary conditions while leveraging the Implicit Layer theory to automatically adjust the number of GNN layers required for convergence. This results in a model with enhanced generalization capabilities, effectively handling Poisson problems of various sizes and shapes. Nevertheless, its current limitations restrict it to small-scale problems, insufficient for industrial applications that often require thousands of nodes. To scale up these models, this thesis further explores combining GNNs with Domain Decomposition methods, taking advantage of batch parallel computing on GPU to produce more efficient engineering solutions.

Synthèse

L'étude de la mécanique des fluides joue un rôle essentiel dans de nombreuses applications, allant de l'étude de la respiration et de la circulation sanguine à la conception de technologies de pointe telles que les pompes, les turbines, l'aviation, et même les voitures de Formule 1. Au cœur de cette discipline se trouvent les équations de Navier-Stokes, qui régissent le mouvement des fluides. Cependant, la résolution analytique de ces équations est encore méconnue, et des outils numériques ont été développés pour en approximer les solutions, conduisant à l'émergence de la Dynamique des Fluides Numérique (CFD, en anglais). Une grande précision dans les résultats de la CFD est souvent obtenue en discrétisant le domaine de calcul en un maillage avec des milliers, voire des millions de nœuds, ce qui aboutit à un système d'équations d'au moins la taille du maillage à résoudre. Par conséquent, aborder de tels problèmes à grande échelle reste un défi complexe, limité par le coût computationnel de la résolution des plus petites échelles spatio-temporelles. Cela est particulièrement vrai lorsqu'il s'agit des équations de Navier-Stokes incompressibles. En effet, afin de résoudre ces équations, les méthodes de choix sont les méthodes de "splitting" qui nécessitent la résolution coûteuse d'un problème de Poisson pour calculer le champ de pression garantissant la contrainte d'incompressibilité. Malgré les progrès importants réalisés dans la communauté du calcul haute performance (HPC, en anglais), la solution du problème de Poisson reste la principale contrainte dans l'accélération des simulations numériques de la CFD.

De nos jours, les méthodes basées sur les données, en particulier celles basées sur les réseaux neuronaux profonds, redéfinissent le domaine des simulations numériques. Les réseaux neuronaux peuvent fournir des prédictions plus rapides, réduisant ainsi les temps de travail nécessaires en ingénierie et en science. Les premières tentatives d'utilisation de réseaux neuronaux profonds pour résoudre des équations aux dérivées partielles, telles que les équations de Poisson, ont interprété les données de simulation comme des images et ont capitalisé sur les remarquables avancées dans les réseaux neuronaux convolutionnels (CNN, en anglais). Malgré des résultats remarquables, ces modèles sont limités aux données structurées, semblables à des images, limitant leur capacité à manipuler les données non structurées rencontrées dans les simulations numériques telles que les maillages. Pour remédier à ces lacunes, des études récentes ont montré les capacités des réseaux neuronaux graphiques (GNN, en anglais) à apprendre à partir de données non structurées. Cependant, ces modèles sont fréquemment entraînés de manière supervisée, nécessitant des solutions de référence computationnellement coûteuses. Par conséquent, ces modèles rencontrent des difficultés de généralisation, également soulignées par le manque de considération explicite des conditions aux limites, un sujet crucial concernant les applications physiques. De plus, l'absence de garanties concernant la fiabilité et la convergence des méthodes d'apprentissage profond a empêché leur mise en œuvre généralisée aux premiers stades de la conception et de la production de nouvelles solutions d'ingénierie.

Cette thèse vise à explorer les approches GNN pour résoudre l'équation de Poisson de pression. Une contribution significative de cette recherche concerne le développement d'un modèle novateur basé sur les GNN qui résout de manière itérative un large éventail de problèmes de Poisson avec des conditions aux limites mixtes, ce qui est en accord avec les scénarios de la CFD. Le modèle proposé exploite la théorie des couches implicites pour ajuster dynamiquement le nombre de couches GNN nécessaires à la convergence, lui permettant de manipuler des maillages de tailles et de formes variables. Le modèle est entraîné à l'aide d'une fonction de perte "orientée-physique", et le processus

d'entraînement est stable par conception. De plus, son architecture originale prend explicitement en compte les conditions aux limites et peut s'adapter à toute solution initialement fournie. Néanmoins, ces modèles sont encore limités aux problèmes avec un petit nombre de nœuds, et les mettre à l'échelle pour des maillages plus grands, comme ceux rencontrés dans les applications industrielles, nécessite des efforts supplémentaires. À cette fin, une autre contribution de cette thèse concerne la combinaison des modèles GNN avec les méthodes de Schwarz couramment rencontrées en décomposition de domaine. En utilisant cette approche, il est possible de tirer parti du calcul parallèle par "batch" sur GPU pour produire des solutions d'ingénierie plus efficaces.

Extended abstract

The study of fluid mechanics plays an essential role in many applications, ranging from studying breathing and blood flow to the design of cutting-edge technologies such as pumps, turbines, aviation, and even Formula 1 cars. At the core of this discipline lie the Navier-Stokes equations, which govern the motion of fluids. However, the analytical resolution of these equations remains elusive, and numerical tools have been developed to approximate their solutions, leading to the emergence of Computational Fluid Dynamics (CFD). High precision in CFD results is often achieved by discretizing the computational domain into a mesh with thousands or even millions of nodes, resulting in a system of equations of at least the size of the mesh to solve. Consequently, tackling such problems at scale remains daunting, limited by the computational cost of resolving the smallest spatio-temporal scales. This is particularly true when dealing with incompressible Navier-Stokes equations. The well-known splitting method, indeed, requires the costly solution of a Poisson problem to compute the pressure field that guarantees the incompressibility constraint. Despite the important progress made in the High-Performance Computing (HPC) community, the solution of the Poisson problem remains the major bottleneck in the speedup of CFD numerical simulations.

Nowadays, data-driven methods, especially those based on deep neural networks, are reshaping the realm of numerical simulations. Neural networks can provide faster predictions, reducing turnaround time for workflows in engineering and science. Initial attempts to use deep neural networks for solving partial differential equations, such as Poisson equations, involved treating simulation data as images and capitalizing on the remarkable advancements in Convolutional Neural Networks (CNNs). Despite noteworthy results, these models are restricted to structured, image-like data, limiting their ability to handle the unstructured data encountered in numerical simulations like meshes. To address these shortcomings, recent studies have shown the abilities of Graph Neural Networks (GNNs) to learn from unstructured data. However, these models are frequently trained in a supervised manner, requiring computationally expensive ground-truth solutions. Consequently, they struggle with generalization, also emphasized by the lack of explicit consideration of boundary conditions, a crucial topic regarding physical applications. Besides, the absence of guarantees regarding the reliability and convergence of deep learning methods has prevented their widespread implementation in the early stages of designing and producing new engineering solutions.

This thesis aims to explore GNN approaches for solving the Pressure Poisson equation. One significant contribution of this research involves the development of a novel GNN-based model that iteratively solves a wide range of Poisson problems with mixed boundary conditions, aligning with CFD scenarios. The proposed model leverages the Implicit Layer theory to dynamically adjust the number of GNN layers required for convergence, enabling it to handle meshes of variable sizes and shapes. The model is trained using a “physics-informed” loss, and the training process is stable by design. Furthermore, its original architecture explicitly takes into account the boundary conditions and can adapt to any initially provided solution. Nevertheless, these models are still limited to problems with a small number of nodes, and scaling them up to larger meshes, as encountered in industrial applications, requires additional effort. To that end, another contribution of this thesis concerns combining GNN models with Schwarz methods commonly encountered in the field of Domain Decomposition. Using this approach allows taking advantage of batch parallel computing on GPUs to produce more efficient engineering solutions.

Thesis framework

This thesis is a collaboration between the two French laboratories IFPEN[1] and INRIA[2], in the framework of the ML4CFD (Machine Learning for CFD) project, funded by the DATAIA[3] institute.

IFPEN is a multi-disciplinary French research institute dedicated to new energy and environmental technologies. The simulation of complex fluid flow is an important research area that enables several applied research projects, from designing electric engine cooling systems and efficient wind propellers to optimizing exchange chemical reactors' performance.

The INRIA Saclay TAU team (TAckling the Underspecified) is well known for its Machine Learning and Artificial Intelligence activities. One of its central topics is the application of Machine Learning methods to scientific computing problems.

There is a long-term ongoing collaboration between the TAU team and IFPEN. Recently, the two laboratories have decided to focus on a joint effort in the domain of Machine Learning applied to scientific computing. A collaborative project has been created between the teams to enhance the performance of CFD simulations using machine learning. This thesis belongs to the "Artificial Intelligence and Learning" scientific priority of the DATAIA institute and deals with the challenge of integrating a priori scientific knowledge of physical models in the learning process.

Within IFPEN, this thesis is in line with the scientific policy of the research direction "Digital Sciences and Technologies". The targeted applications are the design of chemical reactors and the development of cooling systems for electric motors. Furthermore, it is part of the Data Science research axis of the scientific direction, which aims to optimize the processing of massive data flows from experimentation or simulation. In particular, this work contributes to a specific challenge entitled "Artificial Intelligence for Digital Simulation," which aims to develop methodologies and learning algorithms to improve existing digital simulation codes. This thesis is proposed within the "Transverse Fundamental Research" program of the scientific direction. It aims to strengthen IFPEN's scientific and technical mastery in the fields of high-performance computing and thus efficiently serve all of IFPEN's areas of activity, calling for simulation and computing.

[1] Institut français du pétrole et des énergies nouvelles : https://www.ifpenergiesnouvelles.fr/
[2] Institut national de recherche en sciences et technologies du numérique : https://www.inria.fr/fr
[3] https://www.dataia.eu/

Acknowledgements

To everyone who has accompanied and supported me throughout my Ph.D. journey,

To my supervisor Marc Schoenauer: I am grateful for your trust and for granting me the opportunity to pursue my thesis. Your unwavering availability, whether for scientific or administrative matters, has been essential.

To my numerous co-supervisors: Thank you, Michele-Alessandro Bucci, for your patience while guiding me throughout this journey. Your wealth of ideas and unwavering availability to address my numerous questions have been invaluable. Thibault Faney and Jean-Marc Gratien, I express my gratitude for your expert insights and feedback, which have played a crucial role in shaping my research, from academic to industrial applications. And to Guillaume Charpiat, I am deeply thankful for your participation in mentoring me on my PhD journey. Your expertise and our endless discussions on the theoretical aspects (but not only!) of my work have been immensely beneficial.

To my reviewers and examinators: Thank you, Victorita Dolean, Elie Hachem, Alexandre Allauzen, Patrick Gallinari, and Augustin Parret-Fréaud, for dedicating your time to review my work thoroughly. Your constructive feedback has been essential in enhancing the quality of my research.

To my colleagues and peers: Emmanuel and Lucas, thank you for the scientific discussions that have greatly enriched my journey, for the memorable conversations over drinks, and for our shared adventures during conference trips. I also thank Thibault, Manon, Rémy, Michele, Adrien, Stéphane, Cyriaque, and all my colleagues and peers from the TAU team and IFPEN. It has been a pleasure sharing knowledge and expertise with each of you, undoubtedly contributing to improving my work. Special mention goes to my former internship supervisor, Angelo Iollo, who allowed me to discover the research world and motivated me to pursue a PhD.

To my dear friends. The ones from Brive, who have been with me since childhood. The ones from La Rochelle University, with whom my research interest first started. The ones from Bordeaux University, where the idea for this PhD took root. Thank you for bringing me joy in both the good and more difficult times of this journey. I would like to extend a special mention to Alice. Thank you for being so supportive and caring, for our countless and entertaining discussions, and for our trips abroad that provided much-needed rest. Now, it's my turn to offer support, and I wish you all the best as you enter the final phase of your PhD journey.

À ma famille : Un grand merci à ceux qui m'ont accompagné tout au long de cette aventure. Mention spéciale à ma Maman pour ses encouragements permanents, mais surtout pour m'avoir soutenu (et supporté) pendant que je me creusais les méninges sur mes problèmes de recherche … et ce n'est pas fini ! Une pensée particulière pour Thé et Papijap, qui, j'en suis sûr, ont veillé sur moi depuis là-haut.

Contents

Motivations & Outline

The study of fluid mechanics, whether in motion or at rest, holds central importance in numerous applications, ranging from studying breathing and blood flow to the design of cutting-edge technologies such as pumps, turbines, aviation, and even Formula 1 cars. At the core of this discipline lies the Navier-Stokes equations, which govern the behaviour of fluids. However, the analytical resolution of these equations remains elusive, and numerical tools have been developed to approximate their solutions, leading to the emergence of Computational Fluid Dynamics (CFD). In the context of a CFD project, three steps are usually required to solve the problem: i) a pre-processing step in which the domain of interest is discretized into an unstructured mesh, ii) a resolution step where the continuous equations are discretized using numerical methods, and the induced system of equations is solved, and iii) a post-processing step that aims to analyze and visualize the results for decision-making. In the industrial context, achieving high accuracy in CFD results is mandatory, often achieved by considering a discretization of the computational domain into a mesh with thousands, or even millions of nodes, resulting in a system of equations of size at least the size of the mesh to solve. Consequently, solving such problems often comes at the expense of significant time and energy consumption.

Some fluids can be considered incompressible, as e.g., liquids. For such incompressible fluids, simpler versions of the Navier-Stokes equations can be derived: this thesis is dedicated to the numerical resolution of the incompressible Navier-Stokes equations. The numerical approximation of these equations is not straightforward and a popular approach is the special numerical strategy called *splitting schemes*. Splitting schemes aim to divide the resolution of the incompressible Navier-Stokes equations into three simpler steps. Notably, the second step involves solving a Poisson problem for pressure, responsible for ensuring the incompressibility constraint. This step is known as the Poisson Pressure Correction step and is the most computationally expensive task in the resolution of the incompressible Navier-Stokes equations. Despite the important progress made in the High-Performance Computing (HPC) domain, solving the Poisson Pressure problem remains the major bottleneck in the speedup of CFD numerical simulations. Therefore, this manuscript places particular focus on enhancing the resolution of the Poisson Pressure problem.

Nowadays, data-driven methods, especially those based on deep neural networks, are causing a profound transformation in the field of numerical simulations. Neural networks can provide faster predictions, reducing turnaround time for workflows in engineering and science. However, the lack of guarantees regarding the reliability and convergence of deep learning methods has prevented their widespread

implementation in the early stages of designing and producing new engineering solutions. For this reason, recent studies propose to leverage the high flexibility of neural networks to alleviate computationally demanding operations in CFD simulations without necessarily replacing them. The first attempts to apply deep neural networks to solve partial differential equations (PDEs), such as Poisson equations, involved treating simulation data as images and capitalizing on the remarkable advancements in Convolutional Neural Networks (CNNs). Despite achieving noteworthy results, these models are restricted to structured, image-like data, limiting their ability to handle the unstructured data typically encountered in numerical simulations that involve meshes. To address these shortcomings, recent studies have shown the abilities of Graph Neural Networks (GNNs) to learn from unstructured data and provide accurate solutions to PDEs. However, these models are often trained in a supervised manner, requiring a large amount of computationally expensive ground-truth solutions. Consequently, these models lack generalization capabilities, also emphasized by the lack of explicit consideration of boundary conditions, a crucial topic regarding physical applications. Moreover, these models do not provide any guarantees regarding their convergence, which prevents their application to industrial contexts.

This thesis aims to explore Graph Neural Network (GNN) approaches for solving the Pressure Poisson equation. A significant contribution of this research involves the development of a novel GNN-based model that enhances the generalization capabilities of state-of-the-art methods. The proposed GNN model is trained in a "physics-informed" manner, directly minimizing the residual of the discretized Poisson problem, with the aim of learning the underlying physics of the problem. Additionally, the proposed model adopts a node-oriented architecture that explicitly considers boundary conditions and leverages the theory of Implicit Layer to dynamically adjust the number of GNN layers required for convergence, enabling it to handle geometries of variable sizes and shapes. However, such models are still limited to problems with a small number of nodes, and scaling them up to larger geometries, as encountered in industrial applications, requires additional effort. To that end, we propose building hybrid methods by combining these GNN-based models within Domain Decomposition frameworks. Such approaches imply several use cases, among which is the use of such hybrid methods as preconditioners to enhance the convergence of well-established Krylov methods.

This manuscript is divided into three parts, as follows.

In Part I, we provide the necessary background and introduce the current state-of-the-art relevant to our work. Chapter 1 provides an introduction to Computational Fluid Dynamics, covering topics such as the derivation of the incompressible Navier-Stokes equations and the use of numerical strategies like splitting schemes to approximate them. It also discusses fundamental aspects of the Finite Element

method and includes illustrations of well-known synthetic CFD test cases. Chapter 2 introduces Deep Learning methods, beginning with core concepts such as Artificial Neural Networks and Convolutional Neural Networks. It then focuses on Graph Neural Networks, which serve as the central Deep Learning architecture used in this thesis. We conclude this first part with Chapter 3, where we bridge the gap between the first two chapters by exploring groundbreaking state-of-the-art Deep Learning methods applied to solving Computational Fluid Dynamics problems, and highlighting how this thesis contributes to the field. In the light of this quick survey, having set the scene, we are able to describe our contributions in Section 3.5.

In Part II, we present the core GNN-based models developed in this work. Following a specific framework detailed in Chapter 4, we further introduce three different GNN models. Chapter 5 provides a brief overview of the state-of-the-art Deep Statistical Solvers (DSS) approach, which serves as the foundation for many of our contributions. In a nutshell, DSS is a GNN-based solver that iteratively addresses Poisson problems using a fixed number of GNN layers to propagate information. However, this architecture has limitations in terms of generalization, particularly regarding problems of varying sizes and explicit consideration of boundary conditions. Chapter 6 presents DS-GPS, our first contribution, designed to enhance DSS. It introduces a recurrent architecture that significantly reduces the size of the model while enabling variable (though manually specified) numbers of GNN layers for convergence. However, this model still relies on manual setup for training, limiting its generalization potential across different mesh sizes. To address this limitation, Chapter 7 introduces our main contribution: the Ψ-GNN model. Ψ-GNN is a physics-informed GNN-based model designed to solve a wide range of Poisson problems with mixed boundary conditions. Leveraging Implicit Layer theory, it automatically and dynamically adjusts the number of GNN layers required for convergence, enabling generalization across meshes of various sizes while ensuring convergence. Moreover, the model inherently respects boundary conditions, enhancing its generalization capabilities. However, such models remain limited to small-size problems, which hinders their use for industrial applications.

In Part III, we address the challenge of scaling up GNN models to handle problems with a very large number of nodes. Our approach leverages well-established Schwarz methods from the field of Domain Decomposition, whose background information is provided in Chapter 8. The ultimate goal, as described in the concluding Chapter 9, is to combine the use of GNN models described in Part II with Schwarz methods. This innovative approach harnesses batch parallel computing on GPUs in Deep Learning to enhance the resolution of the local sub-problems, extending the applicability of GNN models to geometries with significantly more nodes.

The manuscript ends with some conclusions, sketching some possible research directions opened by this work.

Part I

Background and State-of-the-art

1 - Introduction to Computational Fluid Dynamics

Sommaire

Computational fluid dynamics (CFD) is a branch of fluid mechanics that employs computer-based numerical analysis to simulate, analyze, and solve fluid flow problems.

Throughout history, researchers have been fascinated by phenomena related to fluid flow. However, conducting accurate predictions through experimental studies has always been a time-consuming and expensive task. Consequently, scientists and engineers have sought to integrate mathematical models, numerical methods, and advancements in computer architecture to achieve faster results while maintaining high accuracy.

The roots of CFD can be traced back to 1917 when *Lewis Fry Richardson* (1881-1953) applied modern mathematical techniques to weather forecasting, performing manual calculations. However, the true development of CFD started in the 1940s with the introduction of the first computers. Notably, the earliest numerical solution for

flow over a cylinder was carried out in 1933 by *Thom* in England (Thom, 1933), followed twenty years later by *Kawaguti* in Japan, who employed a mechanical desk calculator (Kawaguti, 1953b).

Since the 1960s, numerous numerical methods have been developed and continue to be widely utilized within the CFD community. A significant milestone was reached in 1980 with the publication of "Numerical Heat Transfer and Fluid Flow" by *Suhas V. Patankar* (Patankar, 1980), which is regarded as one of the most influential books on CFD to date.

In the early 1980s, the adoption of commercial CFD software gained attraction among major companies worldwide. One of the primary challenges posed by the industrial sector was the reduction of computational costs and time. To that end, computers became more powerful, offering enhanced execution speeds and storage capacities. These advancements in computer architecture, coupled with studies in the field of High-Performance Computing (HPC), have significantly improved the computational cost of numerical simulations. More recently, contemporary research in Deep Learning, which harnesses GPU parallel computations, has showcased the tremendous potential for further augmenting simulation speed. However, these models do not currently provide the same level of reliability as traditional solvers, a topic discussed in more detail in Chapter 3.

The applications of CFD are extensive. Whether it involves designing a Formula 1 car, studying the blood flow in arteries, or analyzing wind propagation on a wind turbine blade, CFD has become a crucial step in industrial processes, and a common tool in engineering for studying fluid mechanics.

Regarded as a multidisciplinary field, CFD integrates rigorous mathematical concepts, fluid mechanics knowledge, and computer science techniques. The physical properties of fluid motion are described by mathematical equations, typically expressed as partial differential equations known as the *Navier-Stokes* equations. However, as these non-linear equations do not have a general analytical solution, the use of computer science techniques to obtain approximate solutions is usually used. The general procedure for solving a CFD problem is illustrated in Figure 1.1 and can be outlined as follows:

1. **Pre-processing:** Creation of the geometry, generation of the mesh, configuration of fluid properties, and boundary conditions.
2. **Solver:** Discretization of the governing equations in fluid mechanics using any numerical methods (Finite Element (FEM), Finite Volume (FVM), Finite Difference (FDM)) and resolution of the induced system.
3. **Post-processing:** Analysis of the results, plot of the quantities of interest, decision making, shape optimization ...

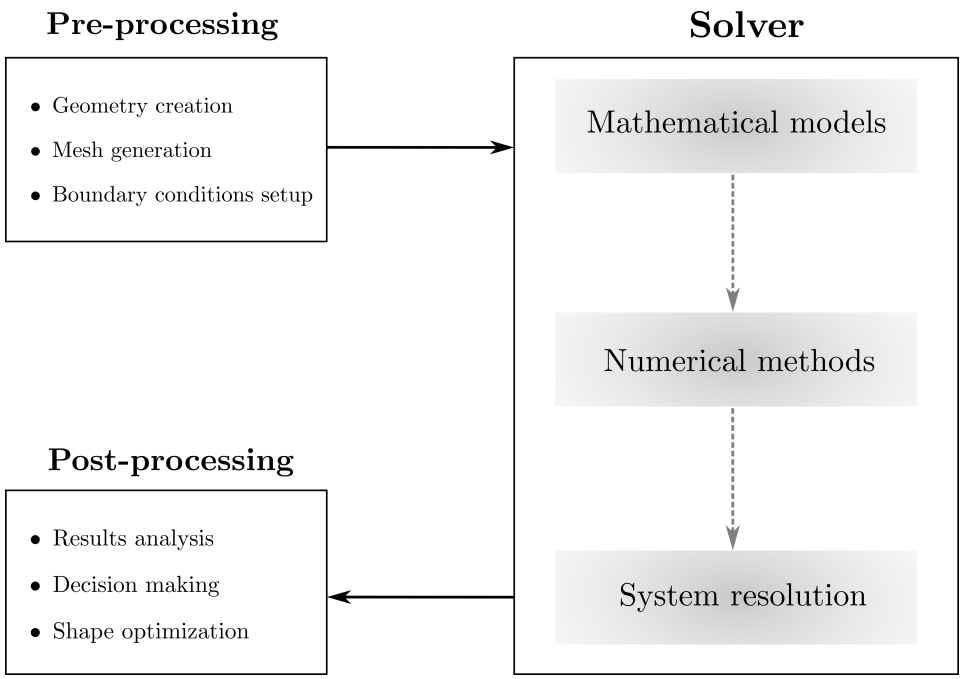


Figure 1.1: Illustration of a general CFD pipeline, divided into three components: Pre-processing, Solver, and Post-processing. The Pre-processing step involves creating the geometry, generating the mesh, and providing initial configurations. In the Solver section, an appropriate mathematical model is chosen and discretized using numerical methods, leading to a system of equations to be solved. The Post-processing step involves analyzing the results and making decisions based on the tackled problem. Note that this process can be repeated multiple times, notably in an optimization process, for instance.

In the following, Section 1.1 presents an introduction to the governing equations in fluid mechanics, establishing a fundamental framework for this thesis. Section 1.2 explores numerical strategies used to discretize the Navier-Stokes equations. In Section 1.3, a concise overview of the Finite Element method is provided. Finally, Section 1.4 applies the theoretical concepts to practical use by examining the numerical resolution of classical CFD problems.

1.1 . Governing equations in fluid mechanics

Fluid dynamics is a scientific field that aims to provide an accurate description of the movement of fluid particles in a flow by setting relationships between the various forces involved. Over several decades of research, mathematicians and physicists have developed local partial differential equations known as the Navier-Stokes equations, which connect velocity, pressure, and forces (such as volumetric and surface forces).

In this Chapter, we adopt the convention that any variable written in bold represents

a vector. For instance, $\mathbf{u} = (u, v, w)$ represents the flow velocity vector field, and u is its first component in the (x, y, z) reference frame in 3D.

To start with, we introduce essential definitions, properties, and classifications of fluids. Here are the key points:

Flow velocity ($\mathbf{u}$): A vector field that represents the velocity of the fluid in a flow. It is described by its components (u, v, w), which depend on the position (x, y, z) in 3D space and time t. The unit of velocity in the SI system is meters per second (m/s).

Pressure (p): The normal force exerted by a fluid per unit area. In the SI system, pressure is measured in newtons per square meter (N/m^2).

Density (ρ): The amount of matter contained in a unit volume of a substance. In the SI system, density is expressed in kilograms per cubic meter (kg/m^3).

Temperature (T): A measure of the hotness or coldness of a system. In thermodynamics, it represents the internal energy of a system. Temperature is measured in Kelvin (K) in the SI system using the absolute temperature scale.

Viscosity (μ or ν): A measure of the resistance of a fluid to a change in shape or movement of neighbouring portions relative to one another. It quantifies the internal friction between adjacent layers of the fluid in relative motion. Viscosity can be measured in two ways. The *dynamic viscosity* (μ) is determined by measuring the resistance of the fluid to flow under an applied external force, and it is expressed in kilograms per meter per second ($kg/(m.s)$). The *kinematic viscosity* (ν) is obtained by measuring the flow resistance of the fluid under the influence of gravity, and it is expressed in square meters per second (m^2/s). The relationship between dynamic and kinematic viscosities is given by $\nu = \frac{\mu}{\rho}$.

Compressibility: A measure of the change in volume of a fluid in response to a pressure change. An *incompressible* flow occurs when the density remains nearly constant throughout the flow. If the density variation during a flow exceeds 5%, it is considered *compressible*. Compressible fluids require additional equations to account for density changes.

Laminar vs Turbulent flow: In a *laminar* flow, fluid particles follow smooth paths in layers, with minimal mixing between adjacent layers. *Turbulent* flow, on the other hand, is characterized by chaotic changes in pressure and flow velocity. *Transitional* flow occurs when the flow alternates between laminar and turbulent states. The dimensionless *Reynolds number* is a critical parameter that determines whether a flow is laminar or turbulent.

Steady and Unsteady flow: A *steady* flow is one in which conditions such as velocity and pressure may vary from point to point but remain constant over time. In contrast, an *unsteady* flow occurs when the conditions at a given point change with

time.

By leveraging these fundamental concepts, we can delve into the Navier-Stokes equations and explore the dynamics of fluid motion more comprehensively. The governing equations in fluid mechanics are built by respecting the conservation laws of physics, which are:

1. Conservation of the mass of the fluid.
2. Newton's second law.
3. First law of thermodynamics.

To mathematically derive these equations, we consider an arbitrary closed control volume V, fixed in space and time. The boundary of V is described by the surface S and its normal unit vector $\mathbf{n}$, pointing towards the outside of V.

1.1.1 . The continuity equation

The principle of mass conservation asserts that the change in the rate of mass within the control volume is equal to the mass flow traversing the surface S:

$$\frac{\mathrm{d}}{\mathrm{dt}} \int_V \rho \mathrm{dV} = - \int_S \rho \mathbf{u} \cdot \mathbf{n} \mathrm{dS} \tag{1.1}$$

By applying Gauss' divergence theorem to the right part of (1.1) we get:

$$\int_V \left(\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) \right) \mathrm{dV} = 0 \tag{1.2}$$

Since (1.2) is valid for any control volume V, we must have:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{1.3}$$

If the fluid is *incompressible*, the density ρ is constant through time, i.e. $\dfrac{\partial \rho}{\partial t} = 0$, and we obtain the incompressibility condition for the flow field:

$$\nabla \cdot \mathbf{u} = 0 \tag{1.4}$$

1.1.2 . The momentum equation

Newton's second law asserts that the rate of change of momentum equals the net force F, which is the sum of all forces applied to the fluid. The momentum $\mathrm{d}p$ of a

small volume of fluid dV is given by $\mathrm{d}p = \rho \mathbf{u} \mathrm{dV}$. By taking into consideration that momentum can be transported in and out of the boundary S of the control volume V, we have the following equation:

$$\frac{\partial}{\partial t} \int_V \rho \mathbf{u} \mathrm{dV} + \int_S \rho \left(\mathbf{n} \cdot \mathbf{u} \right) \mathbf{u} \mathrm{dS} = F \tag{1.5}$$

The net force can be expressed as:

$$F = \int_V \left(\nabla \cdot \sigma + \rho f \right) \mathrm{dV} \tag{1.6}$$

where f is the body force (gravity for instance) and σ is the stress tensor.

Combining (1.5) and (1.6), using Gauss's divergence theorem and the same argument that the volume V is arbitrary, lead to:

$$\frac{\partial}{\partial t} \left(\rho \mathbf{u} \right) + \nabla \cdot \left(\rho \mathbf{u} \mathbf{u} \right) = \nabla \cdot \sigma + \rho f \tag{1.7}$$

By differentiating the left side of (1.7) using chain rules and conservation mass, we obtain:

$$\begin{aligned} \frac{\partial}{\partial t} \left(\rho \mathbf{u} \right) + \nabla \cdot \left(\rho \mathbf{u} \mathbf{u} \right) &= \mathbf{u} \frac{\partial \rho}{\partial t} + \rho \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \nabla \cdot \left(\rho \mathbf{u} \right) + \rho \left(\mathbf{u} \cdot \nabla \right) \mathbf{u} & (1.8) \\ &= \rho \frac{\partial \mathbf{u}}{\partial t} + \rho \left(\mathbf{u} \cdot \nabla \right) \mathbf{u} & (1.9) \end{aligned}$$

Hence we end up with the following:

$$\rho \frac{\partial \mathbf{u}}{\partial t} + \rho \mathbf{u} \cdot \nabla \mathbf{u} = \nabla \cdot \sigma + \rho f \tag{1.10}$$

For Newtonian fluids, the stress tensor can be expressed as:

$$\begin{cases} \sigma &= -pI + \tau \\ \tau &= \lambda \left(\nabla \cdot \mathbf{u} \right) I + 2 \mu \epsilon(\mathbf{u}) \end{cases}$$

where τ is the Cauchy stress tensor, $\epsilon(\mathbf{u}) = \frac{1}{2} \left(\nabla \mathbf{u} + \nabla \mathbf{u}^T \right)$ the strain-rate tensor, p the pressure, μ the dynamic viscosity and λ the second coefficient of viscosity.

We can reformulate (1.10) as follows:

$$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho\mathbf{u}\cdot\nabla\mathbf{u} = \rho f - \nabla p + \mu\nabla^2\mathbf{u} + (\lambda+\mu)\,\nabla\,(\nabla\cdot\mathbf{u}) \\ + (\nabla\cdot\mathbf{u})\,\nabla\lambda + 2\epsilon(\mathbf{u})\cdot\nabla\mu \tag{1.11}$$

In the case of an incompressible fluid, (1.11) reduces to:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu\Delta\mathbf{u} + f \tag{1.12}$$

where ν is the kinematic viscosity.

1.1.3 . The energy equation

The equation governing the conservation of energy is derived from the first law of thermodynamics. It asserts that the rate of change of energy is equivalent to the net rate at which heat is added to the system plus the net rate of work done on the system. Mathematically, this can be expressed as:

$$\frac{\partial}{\partial t}\int_V \rho E \mathrm{dV} + \int_S \rho E(\mathbf{n}\cdot\mathbf{u})\mathrm{dS} = -\int_S \mathbf{n}\cdot q\mathrm{dS} + \int_S \mathbf{n}\cdot(\sigma\cdot\mathbf{u})\mathrm{dS} \tag{1.13}$$

with q the heat flux, E the total specific energy defined as $E = e + \frac{1}{2}\mathbf{u}^2 - f\cdot\mathbf{u}$, where e is the specific internal energy, $\frac{1}{2}\mathbf{u}^2$ the specific kinetic energy and $-f\cdot\mathbf{u}$ the specific potential energy.

By using Gauss's divergence theorem, and since V is arbitrary, we get the expression of the energy equation:

$$\frac{\partial}{\partial t}(\rho E) + \nabla\cdot(\rho E\mathbf{u}) = -\nabla\cdot q + \nabla\cdot(\sigma\cdot\mathbf{u}) \tag{1.14}$$

Note that there are other forms of Equation (1.14) in non-conservative variables, including forms that involve the temperature T as a variable.

1.1.4 . Frame of the thesis

In this thesis, our focus will be solely on the case of isothermal incompressible flows with constant density ρ and viscosity μ. As a result, the energy equation becomes irrelevant, and the flow is completely governed by equations (1.4) and (1.12). Besides, we also restrict ourselves to two-dimensional flows, with the understanding that the following can be readily extended to three-dimensional flows.

To summarize, in this context, our objective is to determine the velocity $\mathbf{u}(\mathbf{x}, t)$ and pressure $p(\mathbf{x}, t)$ within a domain $\Omega \subset \mathbb{R}^2$, solution of the system of partial differential equations:

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} &= -\dfrac{1}{\rho} \nabla p + \nu \Delta \mathbf{u} + f \\ \nabla \cdot \mathbf{u} &= 0 \end{cases} \tag{1.15}$$

where $\nu = \dfrac{\mu}{\rho}$ is the kinematic viscosity.

In order to be consistent, a system like (1.15) needs to be equipped with proper boundary conditions. In the context of the incompressible Navier-Stokes equations, it is common to require that the velocity satisfies particular conditions on the boundary of the domain Ω, denoted as $\partial\Omega$. Here are the most common ones:

1. **Slip:** $\mathbf{u} \cdot \mathbf{n} = 0$
2. **No-slip:** $\mathbf{u} = g$
3. **Stress free:** $\sigma \cdot \mathbf{n} = 0$
4. **Do-nothing:** $\mathbf{n} \cdot \nabla \mathbf{u} - p\mathbf{n} = 0$

Slip boundary conditions indicate that the fluid flow runs parallel to the boundary. No-slip boundary conditions, on the other hand, require the velocity $\mathbf{u}$ to match a specific function g on the boundary (if $g = 0$, the fluid is considered to be at rest). Stress-free and do-nothing boundary conditions are commonly applied to the sections of the boundary where the flow exits the domain (outflow). Stress-free boundary conditions simulate free flow into a large reservoir, while do-nothing boundary conditions are utilized to truncate elongated channel-like domains.

Additionally, due to the presence of the time derivative in the velocity equation, it is necessary to provide an initial condition of the form $\mathbf{u}(\cdot, t_0) = \mathbf{u}_0$.

One of the most important challenges of CFD concerns the understanding of *turbulence*. The basic measure of the tendency of a fluid to produce turbulence is the dimensionless Reynolds number:

$$\text{Re} = \frac{UL}{\nu} \tag{1.16}$$

where U is the characteristic velocity, L a representative length scale of the computational domain and ν the kinematic viscosity.

A high Reynolds number indicates turbulent flow, while a low Reynolds number corresponds to a steady-state laminar flow. The critical Reynolds number is a specific value that delineates transitions between different flow regimes (laminar, transitional, turbulent), which can vary depending on the type of flow and geometry. For example, when examining fluid flow in a pipe, a laminar regime may be observed for Reynolds numbers up to Re $= 2300$, a transitional regime from Re $= 2300$ to Re $= 4000$, and a turbulent regime for Reynolds numbers greater than 4000.

In this thesis, our focus will be limited to laminar regimes, implying low Reynolds numbers.

For theoretical needs, one can find more information about fluid dynamics in Batchelor (2000). For a more applied perspective, a practical approach is presented in Tu et al. (2007).

1.2 . Numerical strategies for incompressible flow

This section focuses on numerical strategies for solving the incompressible Navier-Stokes equations, as described by the system (1.15). In the first part, we demonstrate that utilizing a natural discretization approach often gives rise to significant numerical challenges, such as saddle-point problems. To overcome these difficulties, various methods have been developed, employing different strategies such as *stabilization techniques*, *penalty methods*, or *operator splitting schemes*, as explained in Langtangen et al. (2002). Among these methods, splitting schemes have emerged as the most widely employed approach in practice, forming the central focus of this Chapter.

1.2.1 . Natural approach

A straightforward approach for solving (1.15) is to apply an explicit forward Euler scheme to the momentum equation:

$$\frac{\mathbf{u}^{n+1}-\mathbf{u}^{n}}{\delta t}+(\mathbf{u}^{n}\cdot\nabla)\mathbf{u}^{n}=-\frac{1}{\rho}\nabla p^{n}+\nu\Delta\mathbf{u}^{n}+f^{n} \tag{1.17}$$

Here, δt represents the time step, and n denotes the time level. Using numerical methods to discretize the space operators (e.g., Finite Element, Finite Volume, etc), obtaining the solution $\mathbf{u}^{n+1}$ is straightforward. However, $\mathbf{u}^{n+1}$ often fails to satisfy the divergence-free condition (i.e., the incompressibility equation $\nabla\cdot\mathbf{u}^{n+1}\neq 0$). Additionally, there is no explicit computation of p^{n+1}.

To overcome these drawbacks, one can introduce more implicitness in the velocity term, and explore a semi-implicit approach based on a backward Euler scheme, which uses a previous velocity in the convective term as a linearization technique:

$$\begin{cases} \dfrac{\mathbf{u}^{n+1}-\mathbf{u}^n}{\delta t} + (\mathbf{u}^n \cdot \nabla)\mathbf{u}^{n+1} &= -\dfrac{1}{\rho}\nabla p^{n+1} + \nu\Delta\mathbf{u}^{n+1} + f^{n+1} \\ \nabla \cdot \mathbf{u}^{n+1} &= 0 \end{cases} \tag{1.18}$$

which can be rewritten:

$$\begin{cases} \left(1 + \delta t(\mathbf{u}^n \cdot \nabla) + \delta t\nu\Delta\right)\mathbf{u}^{n+1} + \dfrac{\delta}{\rho}\nabla p^{n+1} &= \mathbf{u}^n + \delta t f^{n+1} \\ \nabla \cdot \mathbf{u}^{n+1} &= 0 \end{cases} \tag{1.19}$$

Again, using any spatial numerical method, the problem amounts to solving the following linear system:

$$\begin{bmatrix} N & Q \\ Q^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ 0 \end{bmatrix} \tag{1.20}$$

where $\mathbf{u}$ contains all spatial degrees of freedom of the vector velocity field and $\mathbf{p}$ is the vector of pressure degrees of freedom which, for first-order methods, corresponds to values at the grid points.

Systems like (1.20) are referred to as *saddle-point problems* and can become singular under certain circumstances. In fact, special spatial discretization or stabilization techniques may be necessary to ensure the invertibility of such systems. According to Langtangen et al. (2002), constructing preconditioners for these saddle-point problems can be highly challenging, making it impractical to directly employ standard iterative solvers. For further insights into the resolution of saddle-point linear systems, we refer the reader to block-preconditioners methods (Silvester and Wathen, 1994; Golub and Wathen, 1998).

Consequently, much of the research on approximating the solution of the incompressible Navier-Stokes equations (1.15) has sought to avoid solving systems like (1.20). Among many different approaches, *splitting methods* have become the most widely used in practice and are introduced in the following section.

1.2.2 . Splitting scheme strategies

In the literature, one can find various variants of operator splitting schemes, mainly depending on the initially chosen strategy (explicit, implicit, linearization of the non-convective term, etc.). However, all versions follow the same general steps. In order to provide an accessible introduction and a comprehensive understanding of its architecture, we first derive its simplest form using explicit schemes.

The overall idea is to decompose system (1.15) into a series of familiar and simpler equations. The evolution of the velocity $\mathbf{u}$ is primarily computed through two steps.

First, we advance the momentum equation by neglecting the pressure term p, resulting in a tentative velocity $\mathbf{u}^\star$. Subsequently, the tentative velocity $\mathbf{u}^\star$ is corrected through projection onto the divergence-free field, which necessitates solving a Poisson equation for the pressure.

The first step is to apply a forward Euler scheme to the momentum equation:

$$\mathbf{u}^{n+1} = \mathbf{u}^n - \delta t \left[(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n - \frac{1}{\rho}\nabla p^n + \Delta \mathbf{u}^n + f^n \right] \tag{1.21}$$

As mentioned in Section 1.2.1, $\mathbf{u}^{n+1}$ does not verify the divergence-free condition and does not explicitly compute p^{n+1}. To address these issues, we consider $\mathbf{u}^{n+1}$ as a "tentative" velocity, denoted $\mathbf{u}^\star$ and try to compute a correction, denoted $\tilde{\mathbf{u}}$ such that $\mathbf{u}^{n+1} = \mathbf{u}^\star + \tilde{\mathbf{u}}$.

Here is the equation for $\mathbf{u}^\star$:

$$\mathbf{u}^\star = \mathbf{u}^n - \delta t \left[(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n - \frac{1}{\rho}\nabla p^n + \Delta \mathbf{u}^n + f^n \right] \tag{1.22}$$

and the one for $\mathbf{u}^{n+1}$ with the pressure evaluated at time $n+1$:

$$\mathbf{u}^{n+1} = \mathbf{u}^n - \delta t \left[(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n - \frac{1}{\rho}\nabla p^{n+1} + \Delta \mathbf{u}^n + f^n \right] \tag{1.23}$$

Subtracting (1.22) and 1.23 yields to:

$$\tilde{\mathbf{u}} = -\frac{\delta t}{\rho}\nabla(p^{n+1} - p^n) \tag{1.24}$$

That is:

$$\mathbf{u}^{n+1} = \mathbf{u}^\star - \frac{\delta t}{\rho}\nabla(p^{n+1} - p^n) \tag{1.25}$$

Yet, $\mathbf{u}^{n+1}$ still has to fulfil the incompressibility constraint:

$$\nabla \cdot \mathbf{u}^{n+1} = \nabla \cdot \mathbf{u}^\star - \frac{\delta t}{\rho}\nabla \cdot \nabla(p^{n+1} - p^n) \tag{1.26}$$

By denoting $\Phi = (p^{n+1} - p^n)$ we obtain the so-called *Poisson pressure equation*:

$$\Delta \Phi = \frac{\rho}{\delta t}\nabla \cdot \mathbf{u}^\star \tag{1.27}$$

Solving (1.27) yields the updated pressure and velocity terms:

$$p^{n+1} = p^n + \Phi \tag{1.28}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^\star - \frac{\delta t}{\rho}\nabla\Phi \tag{1.29}$$

A key question arises regarding the appropriate boundary conditions to apply to the Poisson pressure problem (1.27). While there may be several approaches to derive these conditions, the matter is still the subject of theoretical debates. A comprehensive discussion on this topic can be found in Langtangen et al. (2002) and Gresho and Sani (1998). In any case, in order to obtain practical boundary conditions, one must first consider those imposed on the original incompressible Navier-Stokes problem (1.15).

In cases where problem (1.15) already possesses prescribed pressure boundary conditions, deriving corresponding conditions for the Poisson pressure problem (1.27) is straightforward (e.g., if Dirichlet boundary conditions are applied to the pressure variable, then Φ must satisfy homogeneous conditions). However, in most instances, the system (1.15) imposes boundary conditions on the velocity. In such scenarios, there are two approaches to obtain the boundary conditions for problem (1.27). The first approach involves directly computing $\frac{\partial p}{\partial n}$ by multiplying the momentum equation by the unit vector at the boundary. The second approach entails considering equation (1.25) on its boundary. If $\mathbf{u}$ satisfies Dirichlet boundary conditions (i.e., no-slip boundary conditions), the following holds:

$$\nabla\Phi = \frac{\delta t}{\rho}(\mathbf{u}^{n+1} - \mathbf{u}^n) = 0 \quad \text{on } \partial\Omega \tag{1.30}$$

which results in *homogeneous Neumann boundary* conditions for Φ. Although more complicated problems would lead to more difficult derivations of the pressure boundary conditions, examples used in this thesis will be restricted to the ones introduced above.

One of the oldest splitting schemes is probably the one developed by *Chorin* in Chorin (1967). However, in the following Section 1.2.3, we introduce the Incremental Pressure Correction Scheme (IPCS) (Selim et al., 2012), an improvement of Chorin's splitting scheme, as well as its variational formulation in sight of practical applications in Section 1.4.

1.2.3 . The Incremental Pressure Correction Scheme (IPCS)

The IPCS scheme uses the formulation of system (1.15), involving the strain tensor:

$$\begin{cases} \rho\left(\dfrac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) &= \nabla\cdot\sigma(\mathbf{u},p) + f \\ \nabla\cdot\mathbf{u} &= 0 \end{cases} \tag{1.34}$$

where $\sigma(\mathbf{u},p) = 2\mu\epsilon(\mathbf{u}) - pI$ and $\epsilon(\mathbf{u}) = \frac{1}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right)$.

First, a tentative velocity $\mathbf{u}^\star$ is computed from the momentum equation using a *midpoint* finite difference scheme in time:

$$\mathbf{u}^{n+\frac{1}{2}} = \frac{u^n + u^{n+1}}{2} \tag{1.35}$$

but using pressure p^n from the previous time step. Then, the nonlinear convective term is linearized using the previous known velocity u^n to obtain:

$$\rho\frac{\mathbf{u}^\star - \mathbf{u}^n}{\delta t} + \rho(\mathbf{u}^n\cdot\nabla)\mathbf{u}^n = \nabla\cdot\sigma(\mathbf{u}^{n+\frac{1}{2}}, p^n) + f^n \tag{1.36}$$

To discretize the spacial operators, we use the Finite Element Method (see Section 1.3 for more details). The first step consists of deriving the variational formulation of the equation. This is achieved by applying the Galerkin method with the following notations:

Algorithm 1 Incremental Pressure Correction Scheme (IPCS)

1. Compute tentative velocity $\mathbf{u}^\star$, with correct boundary conditions:

$$\begin{aligned} \langle\frac{\rho}{\delta t}(\mathbf{u}^\star - \mathbf{u}^n), v\rangle + \langle\rho(\mathbf{u}^n\cdot\nabla)\mathbf{u}^n, v\rangle + \langle\sigma(\mathbf{u}^{n+\frac{1}{2}}, p^n), \epsilon(v)\rangle \\ + \langle p^n\mathbf{n}\rangle_{\partial\Omega} - \langle\mu\nabla\mathbf{u}^{n+\frac{1}{2}}\cdot\mathbf{n}, v\rangle_{\partial\Omega} = \langle f^n, v\rangle \end{aligned} \tag{1.31}$$

2. Solve *Poisson Pressure* equation with correct boundary conditions:

$$\langle\nabla p^{n+1}, \nabla q\rangle = \langle\nabla p^n, \nabla q\rangle - \frac{1}{\delta t}\langle\nabla\cdot\mathbf{u}^\star, q\rangle \tag{1.32}$$

3. Update velocity $\mathbf{u}^{n+1}$:

$$\langle\mathbf{u}^{n+1}, v\rangle = \langle\mathbf{u}^\star, v\rangle - \delta t\langle\nabla(p^{n+1} - p^n), v\rangle \tag{1.33}$$

$$\langle u, u' \rangle = \int_\Omega uu' \, \mathrm{d}x, \qquad \langle u, u' \rangle_{\partial\Omega} = \int_{\partial\Omega} uu' \, \mathrm{d}s$$

In this context, let's define v and q as test functions, respectively linked to the velocity $\mathbf{u}$ and the pressure p. The variational formulation of equation (1.36) then reads as:

$$\langle \frac{\rho}{\delta t}(\mathbf{u}^\star - \mathbf{u}^n), v \rangle + \langle \rho(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n, v \rangle + \langle \nabla \cdot \sigma(\mathbf{u}^{n+\frac{1}{2}}, p^n), v \rangle = \langle f^n, v \rangle \tag{1.37}$$

Besides, it can be shown (Langtangen and Logg, 2017), that the integration by part of $\langle \nabla\sigma, v \rangle$ yields to:

$$\langle -\nabla\sigma, v \rangle = \langle \sigma, \epsilon(v) \rangle - \langle \sigma \cdot \mathbf{n} \rangle_{\partial\Omega} \tag{1.38}$$

where $\sigma \cdot \mathbf{n}$ is called the *boundary traction*.

If the problem is solved using *stress-free* boundary conditions, one can take $\sigma \cdot \mathbf{n} = 0$. However, in the following, we will rather use *do-nothing* boundary conditions. Hence, the remaining boundary term is $p\mathbf{n} - \mu\nabla\mathbf{u} \cdot \mathbf{n}$.

The second and third steps are derived following the same schemes as in Section 1.2.2, which corresponds to equations (1.27) and (1.29). We apply the Galerkin method to find their variational formulations. Finally, the entire procedure is summarized in Algorithm 1.

1.3 . The Finite Element method

The previous section introduced the splitting schemes, which are commonly used to approximate solutions to the incompressible Navier-Stokes equations. These schemes aim to break down the system (1.15) into a series of three equations that are easier to solve. However, in order to obtain an approximate solution, it is still necessary to discretize the space operators. Multiple numerical methods are available for this task, with the most straightforward being the Finite Difference method, which operates on regular grids known as cartesian grids. But the Finite Element and Finite Volume methods are more popular, as they can handle unstructured grids.

This section provides an introduction to the fundamental concepts of the Finite Element Method, and explains how to construct a discrete system from the original Partial Differential Equation (PDE). Given that the main focus of this thesis is to develop a Machine Learning method that enhances the resolution of the Poisson Pressure equation (refer to Section 1.4), the Finite Element Method will be explained in the context of solving a two-dimensional Poisson problem with mixed boundary

conditions (i.e. *Dirichlet* boundary conditions on one part of the domain and *homogeneous Neumann* conditions on another part).

Therefore, we consider solving the following 2D boundary-value problem:

$$\begin{cases} -\Delta u &= f & \in \Omega \\ u &= g & \in \partial\Omega_D \\ \frac{\partial u}{\partial \mathbf{n}} &= 0 & \in \partial\Omega_N \end{cases} \tag{1.39}$$

where $u = u(x, y)$ is the unknown function, $f = f(x, y)$ is the force function and $g = g(x, y)$ is the Dirichlet boundary function defined on the 2d domain Ω with boundary $\partial\Omega = \Omega_D \cup \Omega_N$. Besides, $\mathbf{n}$ denotes the outward normal vector on $\partial\Omega$ and Δ is the Laplace operator defined as:

$$\Delta u = \frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y}$$

The Finite Element method generally involves the following steps:

1. Conversion of the PDE into a *variational (weak)* formulation.
2. Discretizaton of the domain Ω into a *mesh* Ω_h.
3. Choice of basis functions.
4. Formulation of the discretized system.
5. Solution of the system.

The variational form of the PDE is obtained by multiplying the PDE by a test function v and integrating it over the domain Ω. The solution function u is referred to as the *trial* function. This yields the following equation:

$$-\int_\Omega (\Delta u)\, v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \tag{1.40}$$

where $\mathrm{d}x$ is the differential element for integration over the domain Ω.

A key principle in deriving the variational form is to minimize the order of the derivatives through integration by parts. By applying Green's formula, the left-hand side of (1.40) can be transformed as follows:

$$-\int_\Omega (\Delta u)\, v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial \mathbf{n}} v \,\mathrm{d}s \tag{1.41}$$

where ds is the differential element for integration over the boundary ∂_Ω.

Both the test and trial functions belong to specific *function spaces* denoted as V for the test function space and $\widehat{V}$ for the trial function space. They are defined as follows:

$$V = \{v \in \mathcal{H}^1(\Omega) \mid u = g \quad \text{on} \quad \partial\Omega_D\}$$
$$\widehat{V} = \{v \in \mathcal{H}^1(\Omega) \mid v = 0 \quad \text{on} \quad \partial\Omega_D\}$$

Here, $\mathcal{H}^1(\Omega)$ represents the mathematical Sobolev space containing functions v such that v^2 and $|v|^2$ have finite integrals over Ω. The solution to the underlying PDE must lie in a function space where the derivatives are also continuous, but the Sobolev space allows functions with discontinuous derivatives. This weaker continuity, resulting from the integration by parts, offers practical convenience in constructing finite element function spaces. Specifically, it enables the use of piecewise polynomial function spaces, which are constructed by joining polynomial functions on simple domains like triangles.

Besides, it can be noticed from (1.41) that:

$$\int_{\partial\Omega} \frac{\partial u}{\partial \mathbf{n}} v \, \mathrm{d}s = \int_{\partial\Omega_D} \frac{\partial u}{\partial \mathbf{n}} v \, \mathrm{d}s + \int_{\partial\Omega_N} \frac{\partial u}{\partial \mathbf{n}} v \, \mathrm{d}s = 0 \tag{1.42}$$

This is due to the test function $v \in \widehat{V}$ having compact support on $\partial\Omega_D$ and the homogeneous Neumann boundary conditions. Therefore, we obtain the following variational problem, whose existence and unicity are stated from the Lax-Milgram theorem (Langtangen and Mardal, 2019):

Find $u \in V$ such that:

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x \qquad \forall v \in \widehat{V} \tag{1.43}$$

The variational formulation (1.43) represents a continuous problem that defines the solution u in an infinite-dimensional space V. To find an approximate solution, the finite element method employs the *Galerkin* method, which replaces the infinite-dimensional function spaces with discrete (finite) spaces.

In this thesis, we consider discretisations (i.e., subdivisions) of the domain Ω into an unstructured triangular mesh Ω_h. We denote as N the number of nodes in Ω_h. The approximation space V_h is then constructed by using piecewise polynomial functions on each triangle K of Ω_h. The parameter h represents the maximal size of the elements K in Ω_h, defined as:

$$h = \max_{K \in \Omega_h} \text{diam}(K) \tag{1.44}$$

We consider here Lagrange $\mathbb{P}_1$ finite elements, although other types would also be suitable for this task. These $\mathbb{P}_1$ finite elements correspond to a first-order discretisation scheme (thus, the number of degrees of freedom matches the number of nodes N in Ω_h). These elements belong to the space of globally continuous functions that are affine functions on each triangle K in Ω_h.

Let us introduce the finite space $V_h \subset \widehat{V}$ as follows:

$$V_h = \{v \in \mathcal{C}^0(\Omega), v_{|K} \in \mathbb{P}_1, \ \forall K \in \Omega_H \text{ and } v_{|\partial\Omega} = 0\}$$

where $\mathbb{P}_1$ is the space of affine functions.

Let $(\phi_i)_{(1 \leq i \leq N)}$ be a set of basis function for V_h. Taking into account the Dirichlet boundary conditions, an approximation u_h of the solution u can be written as:

$$u_h = G + \sum_{i=1}^{N} u_i \phi_i$$

where G is the discretization of the boundary function g on the N degrees of freedom. Notably, $u = g$ if $\phi = 0$ on the boundary of Ω, which is the case if ϕ is a basis for V_h.

The discrete variational problem can then be formulated as follows:

Find $u_h \in V_h$ such that:

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, \mathrm{d}x = \int_\Omega f_h v_h \, \mathrm{d}x \qquad \forall v_h \in V_h \tag{1.45}$$

In practice, the discretized system of equations to be solved is derived from (1.45) without considering, at first, the boundary conditions. Once the system is formed, the Dirichlet boundary conditions are then enforced manually, yielding an equivalent problem. To achieve this, let $\widetilde{V}_h$ be defined such that:

$$\widetilde{V}_h = \{v \in \mathcal{C}^0(\Omega), v_{|K} \in \mathbb{P}_1, \ \forall K \in \Omega_H\}$$

A decomposition of u_h on all N degrees of freedom is:

$$u_h = \sum_{i=1}^{N} u_i \phi_i$$

where $(\phi_i)_{(1 \leq i \leq N)}$ are now basis functions of $\widetilde{V}_h$.

Using this approach and the discrete variational formulation (1.45), it is then possible to derive the following system of equations:

$$\int_\Omega \left(\sum_{j=1}^{N} u_j \nabla \phi_j \right) \cdot \nabla \phi_i \, \mathrm{d}x = \int_\Omega f_i \phi_i \, \mathrm{d}x \qquad \forall \, 1 \leq i \leq N \tag{1.46}$$

$$\sum_{j=1}^{N} u_j \int_\Omega \nabla \phi_j \cdot \nabla \phi_i \, \mathrm{d}x = \int_\Omega f_i \phi_i \, \mathrm{d}x \qquad \forall \, 1 \leq i \leq N \tag{1.47}$$

Introducing the *stiffness* matrix:

$$A = (a_{ij})_{(1 \leq i,j \leq N)} \quad \text{with} \quad a_{ij} = \int_\Omega \nabla \phi_j \cdot \nabla \phi_i \, \mathrm{d}x$$

and considering the solution vector $U = (u_i)_{(1 \leq i \leq N)}$ and the right-hand side vector

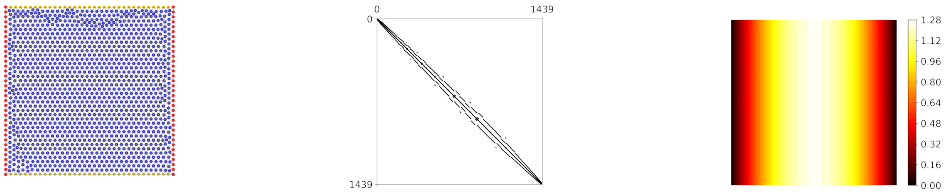
$$B = (b_i)_{(1 \leq i \leq N)} \quad \text{with} \quad b_i = \int_\Omega f_i \phi_i \, \mathrm{d}x$$

then problem (1.47) corresponds to a linear system to solve of the form:

$$AU = B \tag{1.48}$$

As previously stated, this method of discretization does not enforce boundary conditions. In the context of homogeneous Neumann boundary conditions, there is no need for particular adjustments, as these conditions naturally arise through the construction of the variational formulation (1.43). However, Dirichlet boundary conditions are not handled automatically. To enforce them, the linear system (1.48) is modified manually: for the indices corresponding to Dirichlet boundary conditions, the corresponding row of matrix A is set to 0, and a 1 is placed on the diagonal. Additionally, the value in vector B at the corresponding index is changed to match the discrete value of g.

In most FEM softwares, A and B are first computed on a reference element cell and then assembled to form the global system to solve. Besides, the Lagrange basis functions ϕ_i are constructed having the property:



(a) Mesh *(b) Stiffness matrix* *(c) Solution*

Figure 1.2: Resolution of a Poisson problem with mixed boundary conditions on a square geometry with $f = 10$ and $g = 0$ using the FEM. 1.2a is the discretization of the square domain into a mesh with 1439 nodes. Dirichlet boundary conditions (red nodes) are applied to the left and right parts of the boundary, while Neumann boundary conditions (yellow nodes) are set at the top and bottom parts. 1.2b represents the sparsity of the stiffness matrix for this particular problem and 1.2c depicts the solution after solving the induced linear system.

$$\phi_i(x_j) = \delta_{i,j} \quad \text{with} \quad \delta_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Hence, the stiffness matrix A is a sparse matrix. When h tends to 0, then the accuracy of the approximation gets better and better, and the matrix A is very sparse.

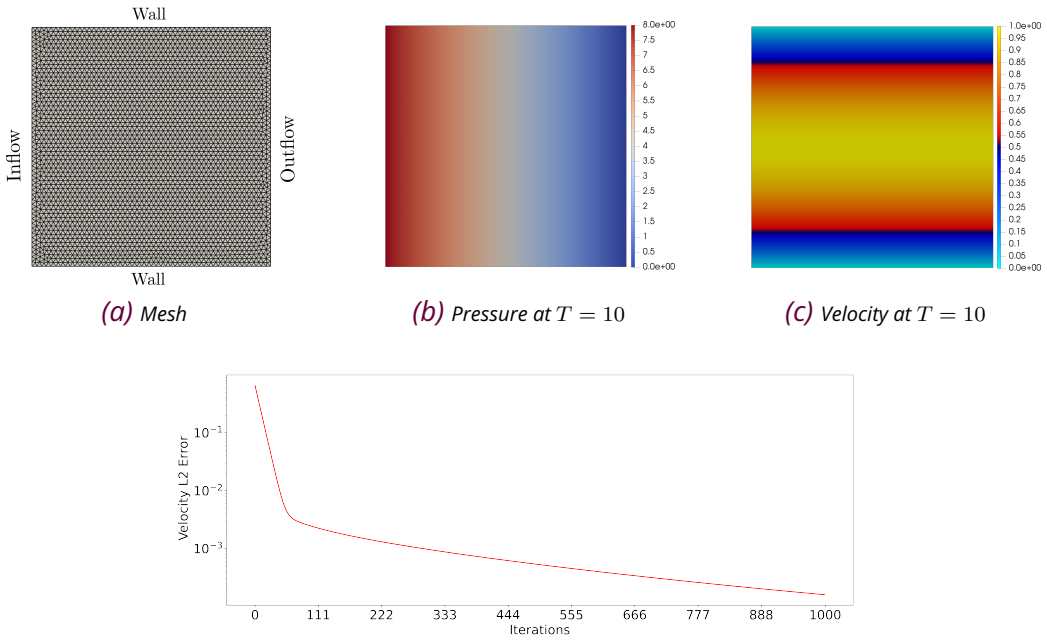
Figure 1.2 illustrates the resolution of a Poisson problem with mixed boundary conditions on a square geometry using the Finite Element method. Numerically, these results are obtained using the Python Finite Element library FEniCS (Langtangen and Logg, 2017).

1.4 . Synthetical test cases

This section aims to apply and demonstrate the theory developed in previous sections. To accomplish this, we present two well-known CFD benchmarks: the "Channel flow" and the "Flow around a cylinder" benchmarks.

For both benchmarks, we solve the Incompressible Navier-Stokes equations (1.15) using the Incremental Pressure Correction Scheme (IPCS), as detailed in Section 1.2. To achieve this, we discretize the variational formulations involved in the IPCS scheme (1) using the Finite Element Method and implement them numerically with the Finite Element Python library Fenics (Langtangen and Logg, 2017).

To handle such problems, additional mathematical details must be provided. Formally, let $(u_h, p_h) \in V_h \times Q_h$ be the approximations of the velocity and pressure functions $\mathbf{u}$ and p, respectively. Here, V_h and Q_h represent inf-sup stable combinations of Finite Element spaces, which are the preferred methods for addressing saddle point problems involving velocity-pressure coupling. In particular, we choose to work with the Taylor-Hood element, a combination of P2 and P1 Finite Elements



(a) Mesh *(b)* Pressure at $T = 10$ *(c)* Velocity at $T = 10$

(d) Evolution of the velocity L2 error across iterations.

Figure 1.3: Resolution of a Channel flow problem. The mesh is displayed in 1.3a, the pressure profile at the final time $T = 10$ in 1.3b, and the magnitude of the velocity field at time $T = 10$ in 1.3c. Figure 1.3d, at the bottom, depicts the evolution of the L2 error of the velocity between the approximate and exact solutions across iterations.

(Taylor and Hood, 1973). This choice results in a second-order scheme for velocity and a first-order scheme for pressure.

1.4.1 . Channel flow

The Channel flow problem, also known as Poiseuille flow, involves studying the behaviour of a fluid between two "infinite" plates.

The domain of interest is a square, denoted as $\Omega = [0, L] \times [0, h]$, with flow entering on the left side and exiting on the right side. This domain, Ω, is discretized into an unstructured triangular mesh, Ω_h, consisting of 3013 vertices, as illustrated in Figure 1.3a. In this example, both the fluid density, ρ, and viscosity, μ, are set to 1, and no external forces are considered, i.e., $f = 0$. The boundary conditions at the inflow and outflow are Dirichlet boundary conditions assigned to the pressure, p, such that $p(0, y) = P_0$ at the inflow and $p(L, y) = P_L$ at the outflow. On the upper and lower walls, no-slip boundary conditions are applied to the velocity function, $\mathbf{u}$, meaning that $\mathbf{u}_{|\text{walls}} = (0, 0)$.

With this configuration, the problem converges to a steady flow, for which analytical solutions for both velocity and pressure are known (refer to Langtangen and Logg (2017)) and defined as follows:

$$\begin{cases} u_{\text{ex}}(x,y) &= u_{\text{max}}\left(\dfrac{4y}{h} - \dfrac{4y^2}{h}\right) \qquad \text{with} \qquad u_{\text{max}} = \dfrac{-8h^2}{8\mu}\left(P_L - P_0\right) \\ p_{\text{ex}}(x,y) &= P_0 + \dfrac{P_L - P_0}{L}x \end{cases} \tag{1.49}$$

In this example, we consider Ω to be a unit square (i.e., $L = h = 1$), and we choose $P_0 = 8$ and $P_L = 0$. This defines the pressure drop and should result in a unit maximum velocity at the inlet and outlet and a parabolic velocity profile without any further specifications. The Reynolds number of this simulation, computed using (1.16), is then equal to 1. We run the simulation for 1000 iterations with a time step of $\delta t = 0.01$, resulting in a total physical time of $T = 10$.

Figure 1.3 illustrates the results of our simulation. Figure 1.3a represents the uniform triangular mesh Ω_h. Figures 1.3b and 1.3c show, respectively, the pressure and velocity profiles at the end of the 1000 iterations. Finally, Figure 1.3d displays the L2 error between the approximated velocity u_h and the analytical solution u_{ex} across the iterations, showcasing the efficiency of the IPCS scheme and validating its implementation.

1.4.2 . Flow around a cylinder

The second benchmark presents a more challenging yet classical problem frequently explored in fluid mechanics: the study of unsteady, incompressible flow over a cylinder placed within a channel at a right angle to the incoming fluid. This problem holds significant practical importance, as it mirrors scenarios found in offshore structures, bridge piers, single silos, and, to some extent, cooling towers, where cylindrical structures encounter fluid flow.

An essential phenomenon in this context is vortex shedding, which occurs over a broad range of Reynolds numbers. The frequency of vortex shedding and the analysis of vortex-induced vibrations play crucial roles in the design of such structures. According to Schäfer et al. (1996) and Kawaguti (1953a), at very low Reynolds numbers (Re < 5), no flow separation occurs, and fluid particles from upstream divide into two groups that flow symmetrically around the upper and lower parts of the cylinder, reaching the back of the cylinder without separation. As the Reynolds number increases, the twin vortices extend, leading to harmonic oscillations in the wake region, represented as a fixed pair of symmetric vortices for $5 <$ Re < 50. For Reynolds values ranging from 50 up to 200, vortexes begin to shed alternatively from the upper and lower parts of the cylinder, forming the well-known "Karman-Vortex Street". At even higher values, turbulence occurs.

In this test case, let Ω be defined as a rectangular domain of size $[0, L] \times [0, h]$, with dimensions $L = 2.2$ and $h = 0.41$, with the flow entering from the left side and

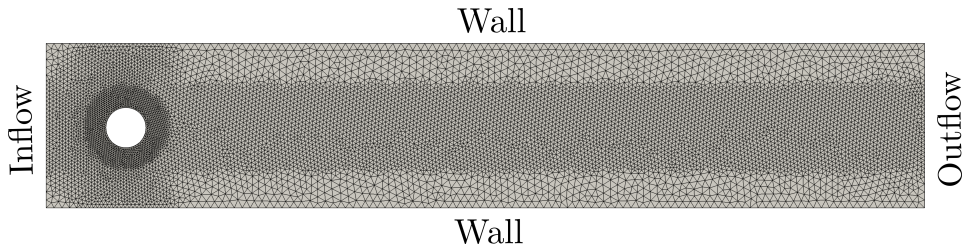


Figure 1.4: Mesh of the "flow around a cylinder" problem. The mesh is refined near the cylinder to enhance the accuracy of the method. The flow enters at the left (Inflow) and exits at the right (Outflow). Specific boundary conditions are prescribed on the top and bottom walls.

exiting from the right side. A cylindrical obstacle is positioned at coordinates $\mathbf{c} = (0.2, 0.2)$ with a diameter $D = 0.1$. The domain Ω is then discretised using GMSH (Geuzaine and Remacle, 2009) into an unstructured triangular mesh Ω_h. To enhance the accuracy of the prediction, the size of the mesh is refined near the cylinder. Figure 1.4 illustrates the created mesh.

The fluid density is set to $\rho = 1$, and no external forces are considered (i.e. $f = 0$). No-slip boundary conditions are applied on both the upper and lower walls, denoted as $\partial\Omega_W$, as well as on the surface of the cylinder, denoted as $\partial\Omega_C$, ensuring that $\mathbf{u}_{|\partial\Omega_W \cup \partial\Omega_C} = (0, 0)$. At the inflow boundary $\partial\Omega_I$, we prescribe a parabolic inflow profile such that:

$$\mathbf{u}(0, y) = \left(\frac{4u_{\text{max}}(h - y)}{h^2},\ 0 \right) \tag{1.50}$$

At the outflow boundary $\partial\Omega_O$, we enforce homogeneous Dirichlet boundary conditions for the pressure, satisfying $p_{|\partial\Omega_O} = (0, 0)$. To achieve varying Reynolds numbers, we set the dynamic viscosity μ as follows, ensuring that for all Reynolds numbers Re , the relationship holds:

$$\mu = \frac{u_{\text{mean}} \times L \times \rho}{\text{Re}} \tag{1.51}$$

where $u_{\text{mean}} = 1$, and $L = 0.1$ serves as the characteristic length.

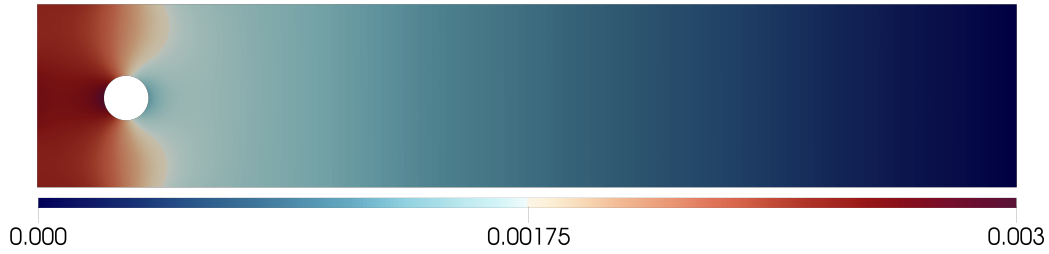
In the following, we consider three different experiments corresponding to three different Reynolds numbers, which are 1, 40, and 100, respectively. The simulations are run with a time step of $\delta = 0.0005$ for 6000 iterations, resulting in a total physical time of $T = 3$.

Figure 1.5 illustrates the pressure profile for the different Reynolds numbers at the final time $T = 3$. More importantly, Figure 1.6 displays the corresponding streamlines of the velocity magnitude for the different Reynolds numbers. With a Reynolds

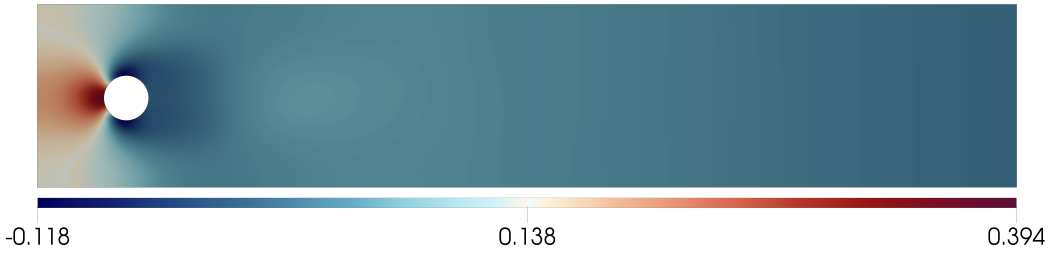
number of 1, as illustrated in Figure 1.6a, there are no vortices, and the flow is laminar, dividing into two symmetric streams when passing around the cylinder. At Reynolds number 40, Figure 1.6b shows the appearance of symmetric vortices, while at Reynolds number 100, vortices shed alternately from the upper and lower parts of the cylinder, as demonstrated in Figure 1.6c. These results are in line with those observed in practice.

1.5 . Conclusion

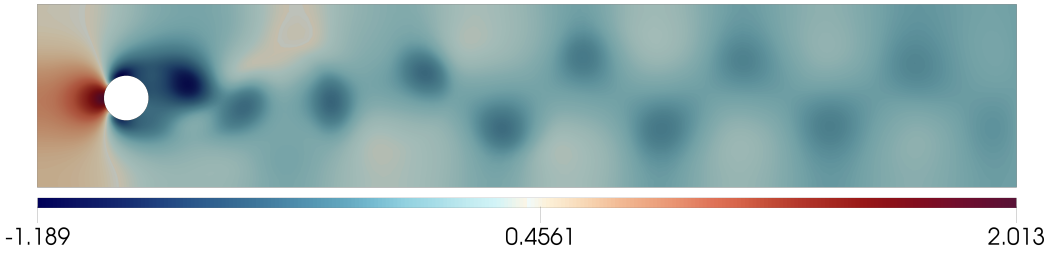
This Chapter provided an overview of Computational Fluid Dynamics (CFD) fundamentals, with a specific focus on solving the incompressible Navier-Stokes equations, described by the system (1.15). However, traditional numerical approaches to solving these equations often encounter instability issues. To address this challenge, Section 1.2 introduced a numerical strategy known as *splitting schemes*. This approach aims to split the resolution of the system (1.15) into a series of subproblems that are easier to solve, which are then discretized using the Finite Element method (FEM). Section 1.3 presented an introduction to FEM and its application in solving a Poisson problem with mixed boundary conditions. It is worth noting that solving the Poisson Pressure equation, as emphasized in Section 1.4, constitutes the most computationally intensive subproblem within the context of the splitting schemes. This result is illustrated in Figure 2.13 of Wang (2015), which investigates the cost of each step of a splitting scheme in relation to the number of CPU processes used. Given the dimension of the problem, iterative procedures are typically employed to solve this step, avoiding the storage of the Laplacian operator. In practice, engineers often employ efficient libraries like PETSc (Balay et al., 2019) to tackle this problem. These libraries leverage algebraic multigrid or highly parallelized domain decomposition methods (or a combination of both) as detailed in Saad (2003). Despite these advancements, the current algorithms for solving the Poisson Pressure equation remain computationally intensive. This limitation restricts the exploration of a wide range of design parameters in engineering applications and prevents practitioners from fully leveraging simulation capabilities. With the emergence of Machine Learning techniques, there is an opportunity to explore models that exploit the powerful parallelization capabilities of GPU computations, contrasting traditional methods that rely solely on CPU computations. These new models could either replace or, more importantly, complement traditional methods, thereby enhancing their performance. The latter forms the main focus of the work presented in this thesis.



(a) Pressure at $T = 3$ with a Reynolds of 1.

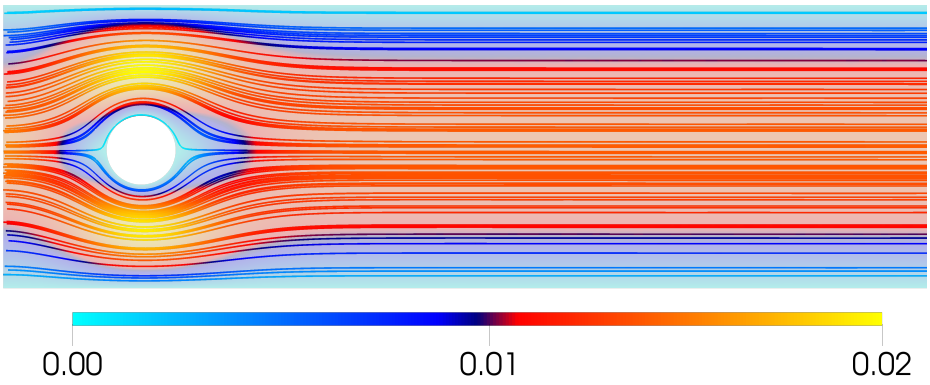


(b) Pressure at $T = 3$ with a Reynolds of 40.

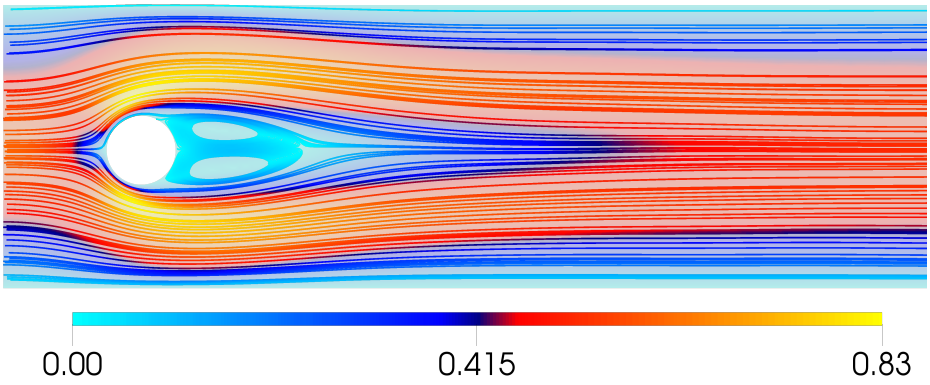


(c) Pressure at $T = 3$ with a Reynolds of 100.

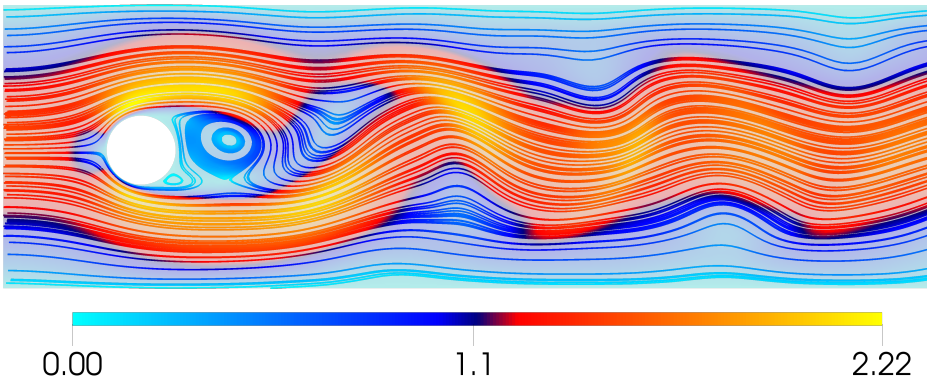
Figure 1.5: Pressure profile at the final time $T = 3$ with respect to different Reynolds number 1, 40 and 100.



(a) *Streamlines of the velocity magnitude at $T = 3$ with a Reynolds of 1.*



(b) *Streamlines of the velocity magnitude at $T = 3$ with a Reynolds of 40.*



(c) *Streamlines of the velocity magnitude at $T = 3$ with a Reynolds of 100.*

Figure 1.6: Streamlines of the velocity magnitude for different Reynolds numbers. The three figures represent a zoom-in on the area close to the cylinder, illustrating the different vortex behaviours.

2 - Introduction to Deep Learning

Sommaire

To address the challenges identified in previous Chapter 1, which related to the need for obtaining precise and fast results from numerical simulations, one approach involves the use of Scientific Machine Learning (SciML), an emerging field that combines classical scientific computing with recent Machine Learning techniques (Baker et al., 2019). SciML leverages state-of-the-art data science methods to enhance the efficiency of numerical modelling of physical and artificial systems, enabling tasks such as extensive scientific data analysis and intelligent automation for complex systems. This application of Machine Learning offers several (non-exhaustive) motivations:

Enhanced accuracy: Machine Learning can significantly improve accuracy by learning patterns from large-scale simulations. This is particularly true in scenarios such as turbulence simulations, for instance (Fukami et al., 2019).

Modeling complexity: Machine learning provides a powerful tool for modelling complex phenomena where capturing the underlying physics analytically is difficult (Recknagel, 2001).

Handling complex data: Machine learning techniques excel at handling complex data. For example, they can analyze medical images to detect and diagnose diseases (Erickson et al., 2017). In another context, they make it easier to process large astronomical datasets, allowing for more efficient and accurate identification and classification of celestial objects (Burke et al., 2019; Farrens et al., 2022).

Computational speed-up: By constructing Machine Learning-based surrogate models, it is possible to approximate computationally expensive simulations (Kochkov et al., 2021; Wiewel et al., 2019). This acceleration of computations significantly enhances efficiency, enables faster exploration of physical phenomena, and opens the door to their optimization and control.

Bridging theory and experiment: Machine learning facilitates the integration of theoretical models and experimental data, leading to improved predictions and a deeper understanding of physical systems (Um et al., 2020).

A key research direction of this thesis concerns Machine Learning-enhanced modelling and simulation, which has seen several significant developments in recent times (refer to Chapter 3).

The present chapter aims to provide a comprehensive introduction to Deep Learning methods, starting from the basics of Artificial Neural Networks (ANNs) and progressing to the construction of Graph Neural Networks (GNNs). Section 2.1 will cover the foundational architectures of ANNs, while Section 2.2 will explore the essential training algorithm known as backpropagation. In Section 2.3, an overview of Convolutional Neural Networks (CNNs) will be presented, as they serve as the basis for the architecture of interest in this thesis: Graph Neural Networks, discussed in more detail in Section 2.4. While CNNs are highly effective for learning from cartesian (regular) data, such as images, GNNs extend the concept of convolution to handle unstructured data, which is well-suited for numerical simulations conducted on unstructured meshes. This chapter serves as a comprehensive guide to Deep Learning, and the subsequent Chapter 3 will present an overview of state-of-the-art of what is now called Scientific Machine Learning, i.e., Deep Learning models applied to physics simulations. In case of additional information, Goodfellow et al. (2016) and Géron (2022) provide both theoretical and practical approaches to Machine Learning methods.

2.1 . Artificial Neural Networks

The first and simplest Artificial Neural Network (ANN) component is the Artificial Neuron. It is a computational unit implementing a function $f_{\mathbf{w}}$ of the d-dimensional input $\mathbf{x} \in \mathbb{R}^d$, defined by its weight vector $\mathbf{w} \in \mathbb{R}^d$, its bias $b \in \mathbb{R}$ and activation function φ. The output variable $y \in \mathbb{R}$ is then defined as:

$$y = f_{\mathbf{w}}(\mathbf{x}) = \varphi\left(\mathbf{w}^T\mathbf{x} + b\right) \tag{2.1}$$

Several activation functions can be considered. The most popular ones are the Sigmoid, the Hyperbolic Tangent (tanh) and the Rectified Linear Unit (ReLU). Refer to Dubey et al. (2022) for a comprehensive survey on activation functions used in Deep

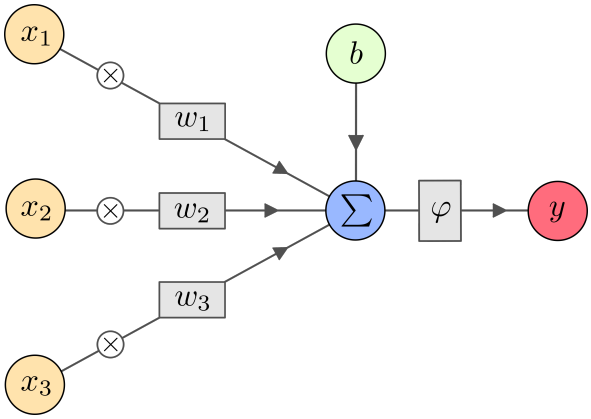


Figure 2.1: Illustration of the architecture of an Artificial Neuron. Each element x_1, x_2 and x_3 is multiplied by the corresponding weights w_1, w_2 and w_3, respectively. The results are aggregated together with the bias term b and then passed through the activation function φ to produce the output y. The weights w_1, w_2 and w_3, and the bias b are the quantities being learned during the training.

Learning. Figure 2.1 displays a graphical representation of an Artificial Neuron[1], showcasing the process of computing an output y from a 3D input $\mathbf{x}$.

In such context, an *Artificial Neural Network* is a set of neurons in which the outputs of some neurons are connected to the inputs of others, thus forming a directed graph, also called *the architecture of the network*. Some neurons receive external inputs (the input of the network) and the outputs of some other neurons are considered as the outputs of the network. When there is no loop in the graph of the network, the network is called *feedforward*: when the values of the external inputs are given, the activations and the outputs or all neurons can be computed in sequence, until reaching the outputs of the network. When there are loops in the graph, the network is called *recurrent* and can be used to model time series or other transient phenomena. In this thesis, only feed-forward networks will be considered (and the word "feedforward" will be omitted).

The simplest (and historically the oldest) network architecture is that of the multilayer perceptron (MLP): the neurons are organized in *layers*, and the output of a neuron in layer n are only connected to the inputs of neurons of layer $n+1$. When all the neurons in a layer are connected to all neurons in the previous layer, this layer is called *fully connected*. The neurons of the first layer receive as inputs the inputs of the network, and the outputs of the last layer are the outputs of the network.

The output vector $\mathbf{y} \in \mathbb{R}^{d_1}$ of one fully connected layer is expressed as :

$$\mathbf{y} = f_W(\mathbf{x}) = \varphi\left(W^T\mathbf{x} + \mathbf{b}\right) \tag{2.2}$$

[1]For the sake of simplicity, "neuron" will in this work stand for "artificial neuron".

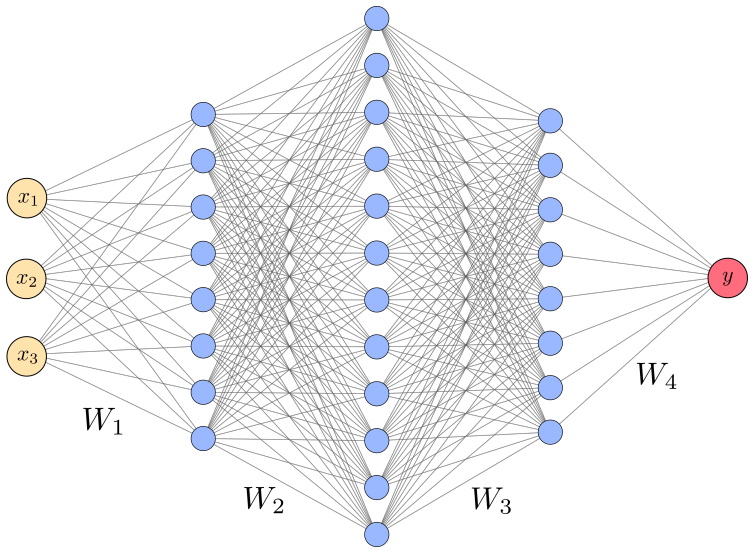


Figure 2.2: Illustration of an MLP with a 3D input $\mathbf{x}$ (yellow circles), three hidden layers of dimension 8, 12 and 8 (blue circles) and a 1D output y (red circle). The output y is computed following Equation (2.3).

where $\mathbf{x} \in \mathbb{R}^{d_0}$ is the vector of input of the layer (i.e., the output of the previous layer), $W \in \mathbb{R}^{d_0 \times d_1}$ is the weight matrix to go from the input dimension d_0 to the output dimension d_1, $\mathbf{b} \in \mathbb{R}^{d_1}$ is the bias vector and φ the activation function common to all neurons of the layer. Figure 2.2 illustrates an MLP with a 3D input $\mathbf{x}$, three hidden layers of dimension 8, 12 and 8, and a 1D output variable y. Each neuron in the hidden layers (blue circles) represents one neuron, as shown in Figure 2.1, i.e., including the weights, the bias and the activation function. Denoting the bias vectors $\mathbf{b}_1$, $\mathbf{b}_2$, $\mathbf{b}_3$ and $\mathbf{b}_4$ related to each weight matrix W_1, W_2, W_3 and W_4, and a unique activation function φ, the output y is computed as follows:

$$y = \varphi\left(W_4\varphi\left(W_3\varphi\left(W_2\varphi\left(W_1\mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2\right) + \mathbf{b}_3\right) + \mathbf{b}_4\right) \tag{2.3}$$

When an MLP contains a deep stack of hidden layers, it is called a Deep Neural Network (DNN). The field of Deep Learning studies DNNs and, more generally, models containing deep stacks of computations. The following section aims to describe how the DNN is trained, i.e. how the weights are optimised to construct a model that best fits the data.

2.2 . Training a Deep Learning model

Deep learning models have gained tremendous popularity thanks to their ability to learn from data. The key objective when training a DNN is to optimize its weights in such a way that it best fits the available data. Training can be categorized into

two types: supervised and unsupervised[2], with the possibility of leveraging semi-supervised techniques that combine elements from both.

In supervised learning, the model has access to input data as well as their corresponding outputs, i.e. labels, which represent the desired solutions for a given task. For instance, when classifying pictures of dogs and cats, images of dogs can be labelled with the word "dog", or value 1, and images of cats can be labelled with the word "cat", or value 0. On the other hand, unsupervised learning does not have access to labelled data, or only a small fraction of the data may be labelled in the case of semi-supervised learning. In such scenarios, alternative learning techniques need to be used (Van Engelen and Hoos, 2020).

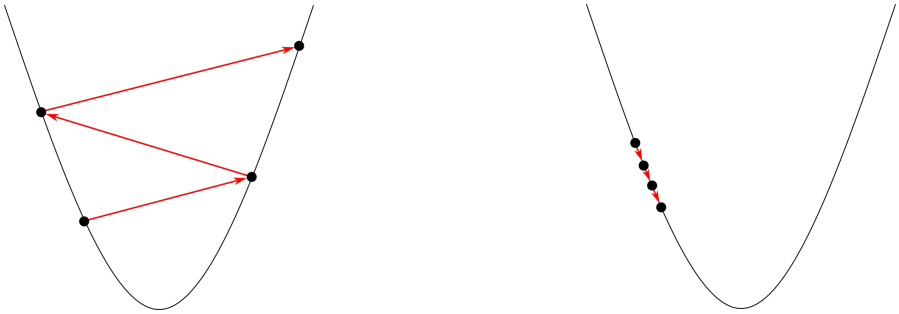
In supervised learning, once a large amount of labelled data is collected and available, one needs to define a *loss function* that quantifies the proximity of predictions of the network (i.e., outputs of the network) to the true labels. Training a Deep Learning model aims to minimise the loss function by modifying the weights of the DNN. Numerous loss functions are commonly used. For regression tasks, examples include the Mean Square Error (MSE) and Mean Absolute Error (MAE), while for classification tasks, the Binary Cross-Entropy and Categorical Cross-Entropy losses are commonly used. Besides, a wide range of customized loss functions exists, some of which are discussed in detail in the survey by Wang et al. (2020).

For many years, scientists struggled to find a way to train DNNs, and it was only in 1985 that David Rumelhard, Geoffrey Hinton and Ronald Williams published a groundbreaking paper entitled Learning Internal Representations by Error Propagations (Rumelhart et al., 1985) that introduced the well-known backpropagation algorithm. In a nutshell, backpropagation is an efficient algorithm that computes the gradient of the error of the network (i.e. the loss function) with respect to all the weights of the model. Once all the gradients are computed, it uses some variant of a classical Gradient Descent step to update the weights.

Gradient Descent (GD) is an iterative optimization algorithm, used to find a local optima of a differentiable function. The main idea behind GD is to take repeated steps in the opposite direction of the gradient of the function at the current point because this is the direction of the steepest descent. When the gradient is zero, the minimum has been found, and the algorithm stops. Formally, let $\mathbf{w}^k \in \mathbb{R}^n$ be the vector of weights at iteration k of the Gradient Descent algorithm, and let f be the loss function, then the weights at iteration $k+1$ are computed as follows:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \nabla f\left(\mathbf{w}^k\right) \tag{2.4}$$

[2]we will not mention here Reinforcement Learning, when training takes place in a dynamic context.



(a) Large learning rate *(b) Low learning rate*

Figure 2.3: Effect of both a large and low learning rate in the Gradient Descent algorithm.

where η denotes the *learning rate*, a use-defined scalar parameter, which determines the speed at which the algorithm converges towards the optimal weights. The learning rate is a critical parameter in the optimization process and must be appropriately chosen: it should neither be too small nor too large. Figure 2.3 illustrates the effects of both a large and a small learning rate, assuming a convex loss function. When the learning rate is too large, the algorithm may oscillate around the optimum and fail to converge, as demonstrated in Figure 2.3a. Conversely, if the learning rate is too small, the algorithm might eventually reach a local minimum, but it will do so very slowly, as depicted in Figure 2.3b.

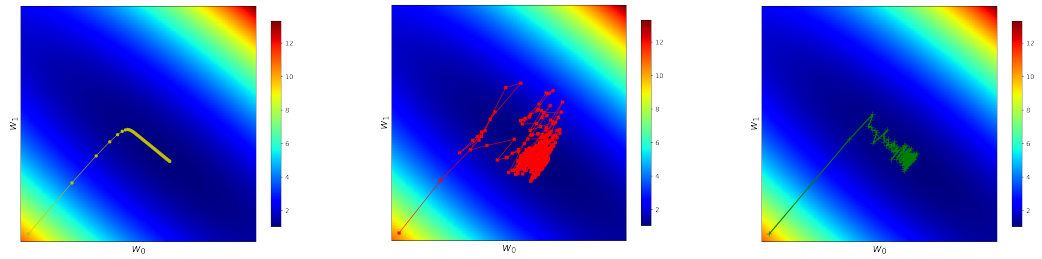
The learning rate is part of the hyperparameters of a Machine Learning training algorithm. Hyperparameters are parameters whose values control the learning process. They are used to improve the learning of the model, and their values have to be set by the user before starting the learning process. Hyperparameters include the choice of the optimization algorithm (Equation 2.4 is one among many possible variants, see below), the learning rate, the choice of activation function, the choice of loss function, the design of the architecture (e.g., number of neurons and layers in an MLP), and more. A poor combination of hyperparameters can hinder the efficiency of the learning phase. Therefore, the search for an optimal combination of hyperparameters is often a crucial step in the process of training a Machine Learning model. Today, many tools are available, including the straightforward and well-known Grid Search algorithm, which explores the space of hyperparameters and selects the model resulting in the best performance. However, for complex models, the number of hyperparameters can become very large, and some more advanced optimization frameworks for fine-tuning hyperparameters can be useful, such as Optuna (Akiba et al., 2019).

At this point, it is worth mentioning that the whole available dataset is usually split into three subsets: the training set ($\simeq 80\%$), the validation set ($\simeq 20\%$) and the test set ($\simeq 20\%$). The training set is used to train the model, the validation set to evaluate it while performing hyperparameters tuning, and the test set to evaluate the

final model. This splitting is intended to prevent the designer of the model from "cheating" by using the same examples for training, hyperparameter tuning, and testing. In particular, the test set should exclusively be used to evaluate the performance of the produced model, without being touched ever before during the learning process.

As already mentioned, there are different variants of Gradient Descent algorithms that mainly differ in the amount of data they use. The Vanilla Gradient Descent (VGD) calculates the error for each example within the training dataset but updates the weights only after evaluating all training examples, using equation 2.4. This method has the advantage of producing a stable gradient and achieving stable convergence (although not necessarily the optimal convergence speed), but it requires the entire dataset to be in memory, which is often not possible. Figure 2.4a shows the evolution of the weights using the VGD in the context of a simple 1D linear regression problem. To alleviate this issue, one could choose to select a random instance from the training set at each iteration and compute the gradient based on that single instance, a process we will refer to as One-Sample Gradient Descent (OSGD). Working with only one instance at a time involves very little data manipulation, allowing the training of very large datasets. However, due to its stochastic nature, this algorithm is less stable than Vanilla Gradient Descent, as depicted in Figure 2.4b. Instead of smoothly decreasing toward the minimum, the algorithm fluctuates, decreasing only on average. The common choice today for a good trade-off is to use mini-batches. At each step, instead of computing the gradient based on the entire training set (as in VGD) or on a single random instance (as in OSGD), it computes the gradient on small random sets of instances called mini-batches or simply "batches". This approach, referred to as Stochastic Gradient Descent (SGD), combines the advantages of both previous methods, as shown in Figure 2.4c, providing a better memory/stability compromise as well as a performance boost, especially when leveraging hardware optimizations for matrix operations, such as GPUs.

Training a very large DNN can be a slow process, and a significant speed boost can be achieved by using faster optimizers than standard Stochastic Gradient Descent. Nowadays, optimizers such as AdaGrad (Duchi et al., 2011), RMSProp (Tieleman et al., 2012), or Adam (Kingma and Ba, 2014) are among the most commonly used (Géron, 2022). These optimizers are based on the concept of momentum optimization. In brief, momentum optimization leverages information from previous gradients to update the weights. In other words, the gradient is used for acceleration, not just speed. This allows the Gradient Descent process to converge more rapidly with adaptive convergence speed. Among the various optimizers, Adam (Kingma and Ba, 2014), which combines the benefits of both AdaGrad and RMSProp, is the most widely used today. Formally, Adam updates the vector of weights $\mathbf{w}^k$ as follows:



(a) Vanilla GD *(b)* One-Sample GD *(c)* Stochastic GD

Figure 2.4: Resolution of a 1D linear regression problem of the form $y = w_0 + w_1 x$ (i.e., with two weights to optimize). Evolution of the solution (w_0, w_1) until the optimal solution is achieved using various Gradient Descent algorithms: Vanilla Gradient Descent (2.4a), One-Sample Gradient Descent (2.4b), and Stochastic Gradient Descent (2.4c). The coloured background of each Figure represents the value of the loss function, with blue indicating the lowest values and red indicating the highest.

$$\mathbf{m}^{k+1} = \beta_1 \mathbf{m}^k + (1-\beta_1)\nabla f\left(\mathbf{w}^k\right) \tag{2.5}$$

$$\mathbf{v}^{k+1} = \beta_2 \mathbf{v}^k + (1-\beta_2)\left[\nabla f\left(\mathbf{w}^k\right)\right]^2 \tag{2.6}$$

$$\widehat{\mathbf{m}}^{k+1} = \frac{\mathbf{m}^{k+1}}{1-\beta_1^{k+1}} \quad , \quad \widehat{\mathbf{v}}^{k+1} = \frac{\mathbf{v}^{k+1}}{1-\beta_2^{k+1}} \tag{2.7}$$

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \frac{\widehat{\mathbf{m}}^{k+1}}{\sqrt{\widehat{\mathbf{v}}^{k+1}} + \epsilon} \tag{2.8}$$

where β_1 and β_2 are scalar hyperparameters, η is the learning rate, and ϵ is a very small scalar. Under the hood, Adam keeps track of both an exponentially decaying average of past gradients and an exponentially decaying average of past squared gradients (i.e., Eq. (2.5) to (2.7)). Adam is an adaptive learning rate optimizer, thus requiring less manual tuning of the learning rate hyperparameter. In addition to the optimizer, it is possible to leverage a learning rate scheduler, whose purpose is to modify the value of the learning rate during training. If tuned correctly, a learning rate scheduler can significantly improve training.

At first, computing the gradients of such neural networks may seem challenging. However, the explicit form of Equation (2.3) makes it possible to apply the chain rule from the output layer back to each layer in turn. By repeatedly applying the chain rule to these operations, the gradients can be computed easily.

As of today, all Deep Learning libraries, such as Pytorch[3] or TensorFlow[4] for in-

[3] https://pytorch.org/

[4] https://www.tensorflow.org/?hl=fr

stance, provide tools for automatic differentiation (Paszke et al., 2017; Baydin et al., 2018) that take care of that.

To summarize, a general training procedure involves handling one mini-batch at a time and going through the whole training set multiple times. Each pass is called an *epoch*. Each mini-batch is fed to the network, and the output of the last layer is computed: it is the forward pass. Intermediate results are saved during this process as they are needed for the backward pass. The backward pass leverages the automatic differentiation framework to efficiently calculate the gradient of the error across all connection weights in the network by propagating it backwards until reaching the first layer. Finally, it performs a Gradient Descent step thanks to an efficient optimizer to update all the weights of the model.

Training a Deep Learning model can be challenging due to inherent issues that may arise. Fortunately, there are methods available to improve the training process. One common issue encountered in the training of Machine Learning models is known as overfitting. Overfitting is a concept in data science that occurs when a statistical model fits too closely to its training data. When this happens, the algorithm will struggle to generalize well to unseen data, defeating its intended purpose. To address overfitting, one effective approach is to train with more data (possibly using data augmentation techniques). If increasing the dataset size is not possible, alternative solutions include using early stopping techniques, applying regularization methods, or adding dropout layers (Srivastava et al., 2014).

Another well-known issue is the problem of "exploding gradients", which consists of a significant increase in the norm of gradients during backpropagation. A solution to address this problem is the use of Gradient Clipping (Zhang et al., 2019), which involves setting a threshold and then clipping the norm of the gradients to ensure it does not exceed this threshold. Another technique is $L2$ Regularization (Cortes et al., 2012), which incorporates a weight decay term into the loss function of the network to encourage smaller weight values, preventing the gradients from becoming too large.

2.3 . Convolutional Neural Networks

When using basic MLP architectures for image data, one straightforward approach would be to flatten images into vectors, as inputs to MLPs are 1D vectors. However, this incurs the loss of most spatial and structural information. Introduced in 1990 by LeCun (Cun et al., 1990) and inspired by the study of the virtual cortex of the brain, Convolutional Neural Networks (CNNs) have revolutionized image processing. Unlike MLPs, CNNs effectively leverage the spatial structure of the input data and have a more appropriate architecture.

In a CNN, the layers are arranged in three dimensions: width, height, and depth.

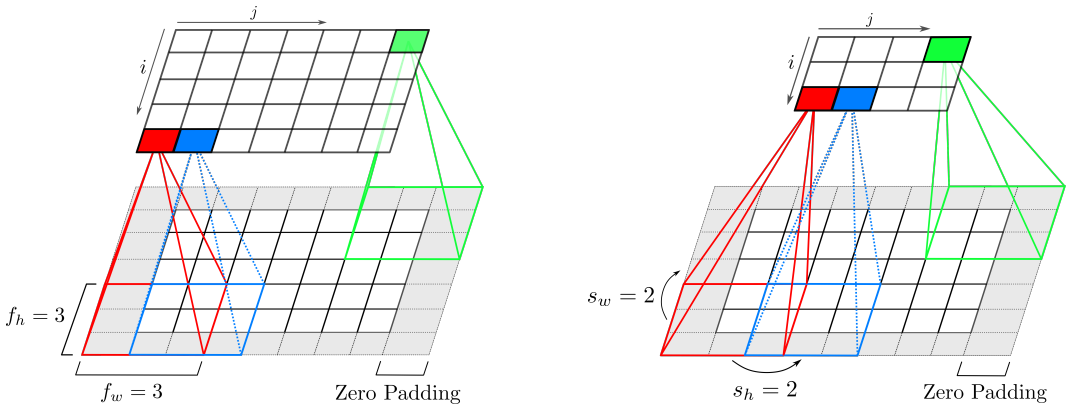


Figure 2.5: Illustration of two convolutional operations. In both figures, the input image (set of white squares at the bottom) is surrounded by pixels set to zero values (grey squares). This technique, known as zero padding, is employed to overcome boundary issues. In both scenarios, a 3×3 convolutional filter (f_h and f_w dimensions) is used. The output image corresponds to the set of white squares at the top of each figure. In 2.5a, the filter slides across the image with a stride of 1, yielding an output image of identical dimensions. Conversely, in 2.5b, the filter is applied with a stride of 2 in both horizontal and vertical directions (s_h and s_w variables), leading to a reduced output image size.

The typical structure of a CNN includes three main types of layers: convolutional layers, pooling layers, and fully connected layers. These layers work together to extract features and capture the hierarchical representations present in the input data.

The convolutional layer is the core building block of the CNN. It performs a dot product between two matrices, where one is the set of learnable parameters (weights), known as a kernel or filter, and the other is a restricted portion of the image (a portion of the image where the kernel is applied). The kernel is spatially smaller than an image but is more in-depth. This means that if the image is composed of three channels (e.g. coloured images with RGB channels), the kernel height and width will be spatially small, but the depth extends up to all three channels. During the forward pass, the kernel slides across the height and width of the image, producing the image representation of each restricted portion. This produces a two-dimensional representation of the image known as a feature map which gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called its stride.

Figure 2.5 displays the application of a convolutional filter with a single feature map to an input image. In 2.5a, the input image undergoes convolution using a 3×3 kernel and a stride of 1, resulting in an output image of identical dimensions. In contrast, in 2.5b, the same convolutional filter is used, but with a stride of 2 both vertically and horizontally, yielding a reduced output image size. To address boundary

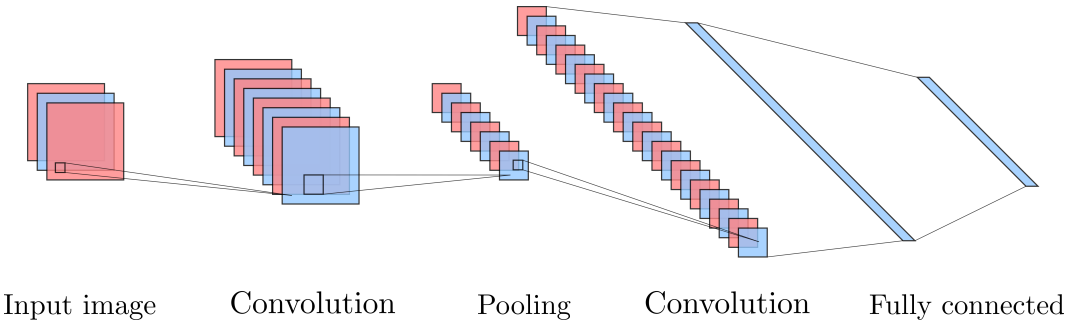


Figure 2.6: Illustration of a standard CNN workflow. A given input image undergoes a series of Convolution and Pooling operations until its dimensions are significantly reduced, enabling its transformation into a flattened vector. This vector is then used as input for a fully connected MLP. The result from this MLP is used to evaluate the loss function, chosen according to the tackled problem.

concerns, pixels with zero values are added around the input image. This technique is referred to as zero padding.

The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

The combination of convolutional layers and pooling layers reduces the images into a form which is easier to process without losing features which are critical for getting good predictions. Hence after performing several layers of such combinations, the original volumic size of the image is so diminished that it can easily be flattened without any loss of information and inputted in a fully connected MLP in order to perform the objective task. Figure 2.6 gives an illustration of a standard CNN workflow.

The success of CNNs lies in their ability to extract multi-scale spatial features from images. Going deeper into the understanding of CNNs, we realize that their effectiveness is attributed to key properties such as local connections, weight sharing, and the use of multiple layers. These properties enable CNN architectures to fit image datasets more effectively by reducing the number of parameters involved and allowing for the reuse of weights. The convolution layer within CNNs aims to extract high-level features from input images. As we stack multiple layers, it can be noticed that the initial layers focus on capturing low-level features like edges, colours, and gradient orientations. With additional layers, the architecture gains an understanding of high-level features as well, by assembling features extracted at the previous level.

CNNs have demonstrated remarkable results, offering state-of-the-art methods for

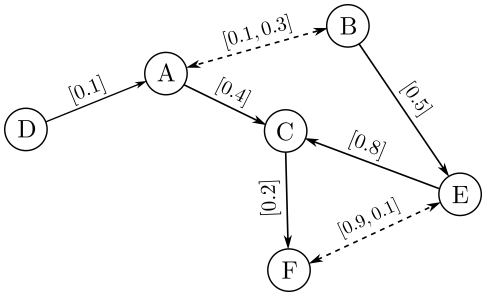
tasks like image segmentation (Krizhevsky et al., 2012), image recognition (Simonyan and Zisserman, 2014), and serving as a core building block for Generative Adversarial Networks (GANs) (Goodfellow et al., 2020). Despite these breakthroughs, the use of CNNs is restricted to regular Euclidean data. However, various data types, including graphs, are characterized by non-Euclidean (unstructured) data. Nowadays, there is an increasing demand for applications involving data that originate from non-Euclidean domains, such as graphs. As a result, there is a need to extend the capabilities of CNNs to handle graph-structured data, which led to the construction of Graph Neural Networks (GNNs), explained in more detail in the following section.

2.4 . Graph Neural Networks

Graphs are ubiquitous in various domains such as social networks, underground networks, molecules, computer networks, and, notably for this work, meshes for numerical simulations.

Graph Neural Networks (GNNs) were specifically designed to extend the concept of convolution to graphs. In the literature, there exists a vast variety of formulations that define GNN operators (i.e. convolution operators on graphs), which can be divided into two categories: spectral-based GNNs and spatial-based GNNs (Wu et al., 2020; Hamilton, 2020). Spectral-based GNNs, originally proposed by (Bruna et al., 2013), are based on graph signal processing and define the convolution operator in the spectral domain of the graph. Once the graph signal is transformed to the spectral domain using the Fourier transform, the convolution is performed through element-wise multiplication, and the resulting signal is transformed back using the inverse Fourier transform. Spatial-based GNNs, on the other hand, apply the convolution operation directly to the graph by aggregating information from each neighbour of a node along with information on their connectivity (definitions are further provided in Section 2.4.1). These techniques have given rise to many GNN operators such as Graph Attention Network (GAT) (Veličković et al., 2017a), MoNet (Monti et al., 2017), or GraphSAGE (Hamilton et al., 2017), for instance. In an effort to summarize all these formulations, scientists have introduced the Message-Passing process, which encompasses (almost) all Graph Neural Network formulations.

In the following, Section 2.4.1 introduces fundamental knowledge of graph theory, while Section 2.4.2 presents the properties that must be followed to define a GNN operator. Section 2.4.3 presents the Message-Passing process, from which we define the Message Passing Neural Network (MPNN) formulation. Recognized as the most versatile and expressive form of GNNs, MPNN is the preferred method in this work. Additionally, this section provides examples of two other famous GNNs, illustrating that both can be defined following this idea of Message-Passing. Section 2.4.4 further explains how to use Message-Passing in the design of a GNN model to propagate information throughout a graph, and Section 2.4.5 concludes by pre-



(a)

	A	B	C	D	E	F
A	0	**0.3**	**0.4**	0	0	0
B	**0.1**	0	0	0	**0.5**	0
C	0	0	0	0	0	**0.2**
D	**0.1**	0	0	0	0	0
E	0	0	**0.8**	0	0	**0.9**
F	0	0	0	0	**0.1**	0

(b)

Figure 2.7: 2.7a displays a directed and weighted graph with 6 nodes labelled from A to F. 2.7b represents the corresponding adjacency matrix. Due to the directionality of the edges, the adjacency matrix is asymmetric. For instance, considering the connection between nodes A and C, the matrix entry at row A, column C, is 0.4 (indicating a connection with weight 0.4), distinct from the 0 entry at row C, column A (indicating no connection).

senting several common Machine Learning tasks on graphs.

2.4.1 . Elements of graph theory

A graph is a mathematical structure that represents a set of objects, denoted as nodes, which are connected by relationships, called edges. Formally, a graph G is defined as a pair $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ denotes the set of N nodes and $\mathcal{E}$ is the set of M edges (a set of paired nodes). An edge $e_{ij} = (v_i, v_j) \in \mathcal{E}$ has two endpoints which represent the source node v_i pointing towards the target node v_j. An edge is said to be directed if the relationship between the source and target nodes exists in only one direction. On the contrary, if both e_{ij} and e_{ji} exist, the edge is said to be undirected. Overall, a graph is directed if all its edges are directed, and undirected if all its edges are undirected. The neighbourhood of a node v_i is defined as the set of nodes that have an edge common with v_i, i.e.:

$$\mathcal{N}(v_i) = \{v_j \in \mathcal{V},\ (v_i, v_j) \in \mathcal{E} \text{ or } (v_j, v_i) \in \mathcal{E}\}$$

The degree $d(v_i)$ of a node v_i is the number of neighbours of v_i and the k-hop neighbourhood of a node v_i is the set of nodes at a distance (node-wise) less than or equal to k from v_i (the neighbourhood is the 1-hop neighbourhood).

Numerically, the edges between nodes within a graph can be represented by an adjacency matrix. For a graph $G = (\mathcal{V},\ \mathcal{E})$ with N nodes, the adjacency matrix is represented by a squared matrix $A \in \mathbb{R}^{N \times N}$ such that :

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

Obviously, A is symmetrical if the graph is undirected. If the graph is weighted (i.e. the edge connecting two nodes v_i and v_j is weighted by some value a_{ij}), the adjacency matrix can include the weights, and be expressed as:

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

To further illustrate this, Figure 2.7 displays a directed and weighted graph, along with its corresponding adjacency matrix.

Apart from the directed/undirected property, graphs may have node attributes, represented as a node feature matrix $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)^T \in \mathbb{R}^{N \times d}$ with $\mathbf{x}_i \in \mathbb{R}^d$ the feature vector of a node v_i. In practice, node features characterise intrinsic information about individual nodes, which can vary depending on the context. For example, in physics simulations, node features may represent the velocity or pressure of a point in space (Pfaff et al., 2020), while in protein interaction graphs, they could indicate immunological signatures (Hamilton et al., 2017).

A graph may also have edge attributes, represented as an edge feature matrix $\mathbf{E} \in \mathbb{R}^{M \times c}$ with $\mathbf{e}_{i,j} \in \mathbb{R}^c$ representing the feature vector of an edge (v_i, v_j). When edges have features, they provide an indication of the strength of the connection between two nodes, like measuring the intensity of a friendship in a social network (Fan et al., 2019), for instance.

2.4.2 . Properties of a GNN operator

Unlike standard neural networks designed for vector data, or CNNs designed for grid-like data, GNNs were developed to handle the irregular structure of graph data. The main objective of a Graph Neural Network model is to learn and leverage the complex relationships between nodes and edges within a graph, enabling the model to make predictions or classifications based on local connectivity patterns. However, due to the increased difficulty of unstructured data compared to structured data, defining a GNN operator requires additional care and should adhere to certain properties.

For instance, a first idea for defining a Deep Neural Network on graphs might involve using the adjacency matrix as input to a DNN by flattening it and feeding the result to an MLP. Nevertheless, this method raises several issues: i) it is not suitable for graphs of varying sizes. When designing an architecture for graphs, the GNN model should handle varying input sizes, ii) it no longer considers the local structure of the graph as well as relationships between nodes, represented by edges that denote interactions between two nodes, iii) it depends on the ordering of nodes in the adjacency matrix. Yet, a crucial structural characteristic of graphs is the independence of their properties with respect to the order of their nodes, which implies that the

GNN model needs to be equivariant to any permutation of the node features of the graph: if the node features of a graph are permuted, the output of the model should be permuted accordingly. Formally, for any function $f : \mathbb{R}^{N \times d} \to \mathbb{R}^{N \times d}$, that takes as input node features matrix X, permutation equivariance should verify:

$$f(PX) = Pf(X) \tag{2.9}$$

where P is a permutation matrix (a square binary matrix that has exactly one entry of 1 in each row and each column and 0 elsewhere, see Hamilton (2020)).

These three considerations must be respected in the design of a GNN operator. Next, we introduce the Message-Passing process, which gives rise to GNN operators respectful of these properties.

2.4.3 . The Message-Passing process

Training a CNN model starts with learning the weights of a fixed-size operator (i.e. a filter or kernel), which slides across the image data to perform convolution (see Section 2.3). Learning such an operator is possible only because the input data of a CNN are structured, ensuring that each component has a constant and fixed number of neighbours. In unstructured data types, such as graphs, the number of neighbours can vary from node to node, making the use of CNN-like operators non-viable.

Despite this difference, the fundamental idea behind convolution remains valid, whether using CNNs or GNNs. In CNNs, the features of one pixel in an image are updated by gathering information from its direct neighbouring pixels, as shown in Figure 2.5. Since a pixel in an image has a fixed number of neighbours, the filter is a fixed-size square filter that can be slid across the image to update the features of each node.

The concept of gathering information from neighbouring pixels in CNNs forms the fundamental idea of the Message-Passing process. In GNNs, this process operates in a similar yet more flexible fashion: it transfers information along edges from one node to its neighbours. A node feature $\mathbf{x}_i$ is then mapped onto a new node feature, often referred to as a latent representation denoted by $\mathbf{h}_i \in \mathbb{R}^{\tilde{d}}$. Note that $\tilde{d}$ may differ from d. This latent representation is computed by aggregating features from the neighbours of $\mathbf{x}_i$, taking into account edge features if necessary. The aggregation of neighbouring features uses permutation-invariant functions such as sum, average, or max, ensuring that the final model is permutation equivariant. As a result, each latent representation of the latent feature matrix $H = (\mathbf{h}_1, \ldots, \mathbf{h}_N)^T \in \mathbb{R}^{N \times \tilde{d}}$ incorporates information from each node feature $\mathbf{x}_i$ and its direct neighbours, as depicted in Figure 2.8.

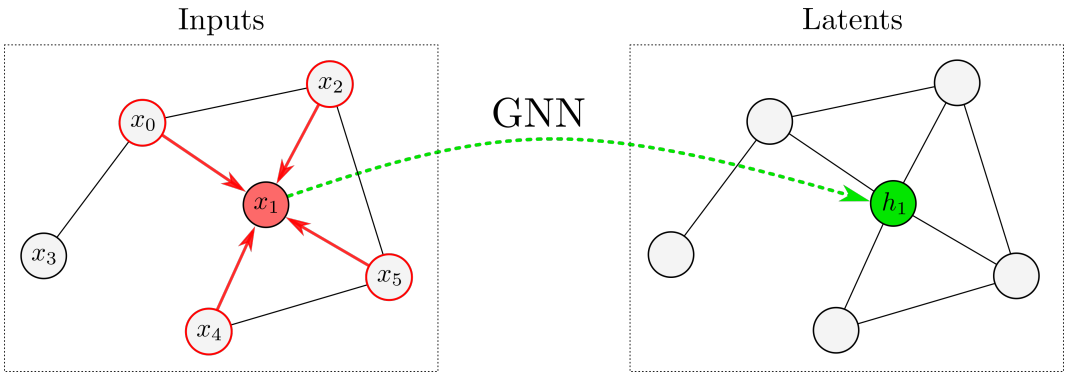


Figure 2.8: Process of updating node features within a graph. The new latent representation $\mathbf{h}_1$ associated with a node feature $\mathbf{x}_1$ is computed by the application of a GNN operator. The GNN operator follows the idea of Message-Passing and computes the latent representation by considering the features of both $\mathbf{x}_1$ itself and its immediate neighbours $\mathbf{x}_0$, $\mathbf{x}_2$, $\mathbf{x}_5$, $\mathbf{x}_4$ along with their relationships (red arrows).

Almost all recently developed GNNs adhere to the idea of Message-Passing. Next, we introduce three GNN operators: the Message Passing Neural Networks (MPNN) (Gilmer et al., 2020), the preferred operator in this work due to its high flexibility, as well as the Graph Convolutional Network (Kipf and Welling, 2016a) and the Graph Attention Network (Veličković et al., 2017a). All three of these can be viewed as instances of Message-Passing GNNs.

Message Passing Neural Network

Message Passing Neural Network (MPNN) represents the most versatile and expressive form of GNNs. Its formulation introduces the crucial notion of messages, taking into account explicit information exchanged between nodes. These messages consider both the node features and the edge features, effectively spreading information between nodes along the edges to update their latent representations. Mathematically, computing the latent representation of a node i using MPNN can be broken down into two steps:

1. Compute messages between node i and node j:

$$\mathbf{m}_{ij} = \psi(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{ij})$$

2. Aggregate the information using a permutation-invariant function:

$$\mathbf{h}_i = \phi\left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \mathbf{m}_{ij}\right)$$

where ϕ and ψ are trainable functions (e.g., MLPs) and $\bigoplus$ is a permutation-invariant aggregator (e.g., sum, average, or max).

Using MPNNs adhere to the previously mentioned properties. The MPNN operator is applied independently and simultaneously to each node in the graph, and thus applies to graphs of different sizes. It extracts the local structure of the graph by gathering information from neighbouring nodes and edge relationships. Finally, using MPNNs ensures that the GNN model is permutation-equivariant, which is achieved by applying a permutation-invariant function to aggregate information for every node in the graph.

In this thesis, MPNNs are the operators of choice for designing the upcoming GNN models, further introduced in Part II. They represent the most expressive form of GNNs, enabling explicit consideration of edge information when computing messages between two nodes. For instance, in the development of a GNN model applied to a mesh for predicting solutions to Partial Differential Equations (PDEs), edge information can be crucial, representing information such as the distance between two nodes or their relative positions (refer to Section 4.1).

Spectral-based GNNs: GCN example

Spectral-based GNNs were first proposed in Bruna et al. (2013). It uses graph signal processing and defines the convolution operator in the spectral domain. The fundamental concept is to directly apply a convolution filter to the eigenvalues of the Laplacian matrix. The Laplacian matrix is an improved version of the adjacency matrix A, expressed as $L = D - A$, where D is the degree matrix (i.e. a diagonal matrix with the degrees of each node on the diagonal). Spectral networks reduce the convolution filter to a diagonal matrix, whose coefficients are the learnable parameters of the network. However, this approach faces significant drawbacks. Firstly, the filter is applied to the entire graph, implying that there is no notion of locality. Additionally, it is computationally inefficient, as computing the eigenvalues of the Laplacian matrix requires computing the Singular Value Decomposition (SVD) of L, something far too expensive for large graphs. Lastly, all considered graphs must be of the same size. A major improvement which enhances the efficiency of spectral methods is to approximate the SVD using Chebyshev expansion. Consequently, the operator is constructed by acting directly on the powers of the Laplacian matrix, resulting in a lower computational complexity. Furthermore, this approach addresses the problem of locality by considering that the features representation should be influenced only by its K-hop neighbourhood: using Chebyshev expansion of order K allows the definition of a K-localized convolution. Famous convolutional operators are GCN (Kipf and Welling, 2016a), ChebNets (Defferrard et al., 2016), or SGC (Wu et al., 2019). In particular, GCN is a special case of ChebNets, where the order K of Chebyshev extension is equal to 1. Even though these methods are spectral-based, they still can be written node-wise in a Message-Passing formulation:

$$\mathbf{h}_i = \phi\left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} c_{i,j}\psi(\mathbf{x}_j)\right) \tag{2.10}$$

where ϕ and ψ are trainable functions (e.g., MLPs) and $\bigoplus$ is a permutation-invariant aggregator (e.g., sum, average, or max). The $c_{i,j}$ coefficients are fixed and directly depend on the adjacency matrix. For instance, in GCN (Kipf and Welling, 2016a), $c_{i,j} = \frac{1}{\sqrt{d_i d_j}}$, where d_i is the degree of a node i.

These methods are computationally efficient but do not directly support edge features and omit the notion of messages along graph edges, which reduces the expressivity of the network.

Graph Attention Networks

Graph Attention Networks (GAT) present a slightly more intricate approach. When the edges of a graph encode similarities between nodes, spectral-based GNNs might be a method of choice. However, if the edges represent more complicated relationships, GATs may be a suitable solution. In GATs, neighbouring node features are aggregated using implicit weights via attention mechanisms. This process allows the model to learn the strength of a relationship between two nodes, enhancing the expressivity of the network. Among the most renowned contributions in this domain are GAT (Veličković et al., 2017a) and GaAN (Zhang et al., 2018a). The Graph Attention operator can be mathematically described as follows:

$$\mathbf{h}_i = \phi\left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \alpha(\mathbf{x}_i, \mathbf{x}_j)\psi(\mathbf{x}_j)\right) \tag{2.11}$$

where ϕ and ψ are trainable functions (e.g., MLPs) and $\bigoplus$ is a permutation-invariant aggregator (e.g., sum, average, or max). Here $\alpha(\mathbf{x}_i, \mathbf{x}_j)$ represents a learnable attention mechanism. For instance, in the context of GAT (Veličković et al., 2017a), the authors proposed computing the coefficients based on node features, which are then passed into a trainable attention network. Then, the softmax function is applied to the attention weights to build a probability distribution.

Figure 2.9a illustrates the computation process of the attention coefficient. Moreover, significant improvements were observed when enhancing the attention process with multi-head attention (i.e. considering an aggregation of multiple attention coefficients computed for the same link) as depicted in 2.9b. This process has the double benefit of enhancing the expressivity of the network (by improving the capacity of the model to understand the strength of a connection) while simultaneously providing regularization.

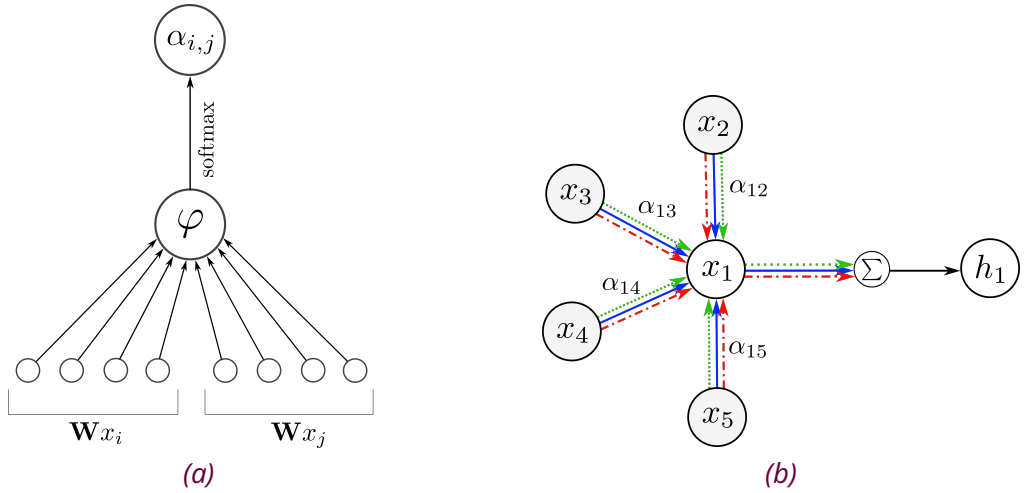


Figure 2.9: 2.9a illustrates the computation of the attention coefficient in GAT. Both node features x_i and x_j are multiplied by a trainable matrix W, followed by an activation function φ (LeakyReLU in the original paper). The result is then passed in a softmax function to produce α_{ij}. 2.9b displays the multi-head attention process used in GAT. Different replicas of the same attention coefficient but with different weights are computed (blue, red and green arrows) and then averaged to obtain the updated latent representation.

These methods prove highly effective when dealing with graphs that do not encode node similarities and are reasonably sized. However, it is crucial to note that this operation requires computing scalar values for each edge of the graph which can be computationally intensive for large graphs. Besides, Attentional operators do not depend on the graph structure but only on the node representations, which can also be a bottleneck when considering passing messages along edges.

2.4.4 . Information propagation

In previous section, we introduced three GNN operators, all of which follow the idea of Message-Passing. As mentioned earlier, the application of Message-Passing updates the features of a node toward a latent representation by aggregating information from its neighbours, along with their relationships (i.e., edge information): this new latent representation captures both feature-based and structural information of the node and its direct neighbours.

The architecture of a GNN model is often composed of several steps of Message-Passing, i.e., several GNN layers. This type of architecture is motivated by a more general concept referred to as information propagation, driven by the following idea: at each layer, every node aggregates information from its local neighbourhood. As these layers progress, each latent representation contains more information from further reaches of the graph. To be precise: after the first layer ($k = 1$), every node representation contains information from its 1-hop neighborhood, i.e., every node representation includes information about the features of its immedi-

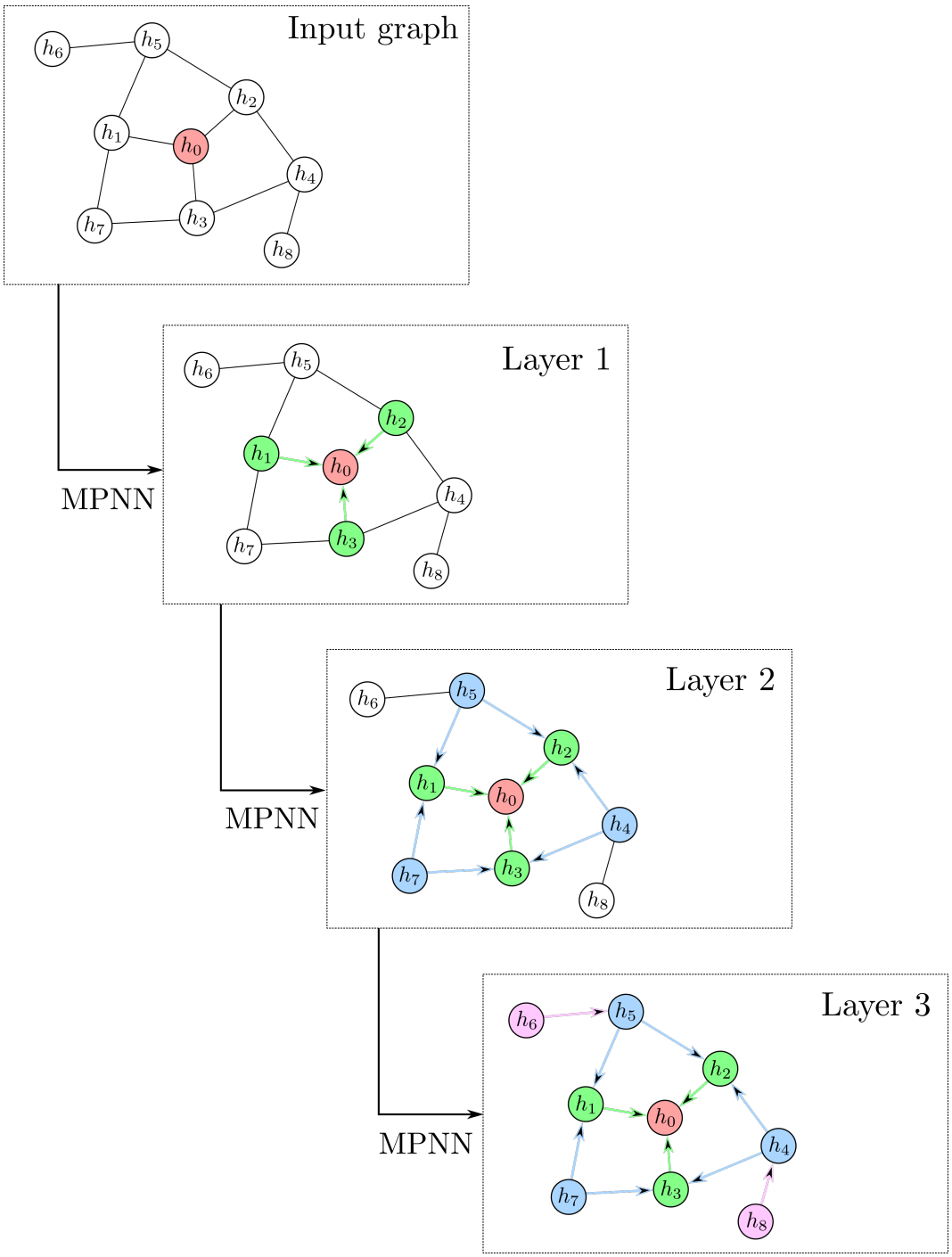


Figure 2.10: Visualization of information propagation in the context of a GNN model with 3 MPNN layers applied to a graph with 9 nodes. The figure illustrates the information received by the node h_0 (red node) as the number of GNN layers increases. In the final layer, h_0 encodes feature-based and structural information received from all nodes in the graph.

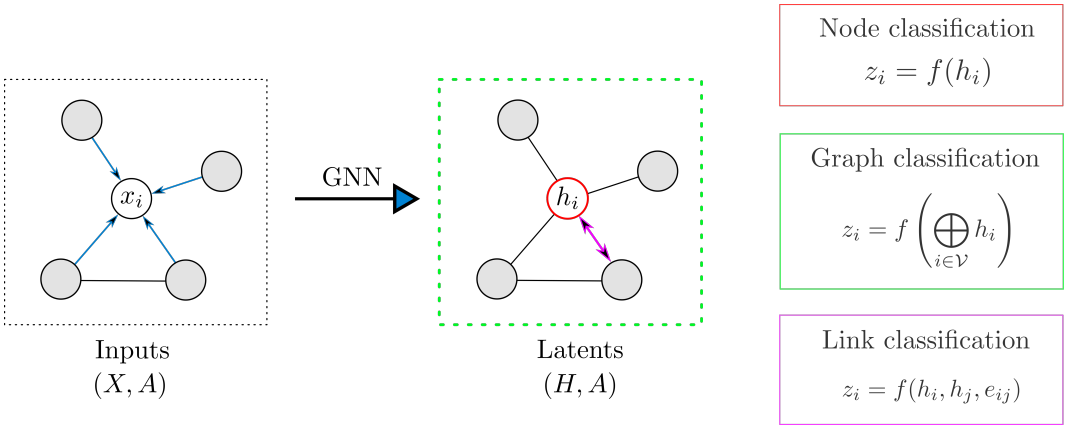


Figure 2.11: Illustration of three standard Machine Learning tasks that can be executed on graphs. Initially, an input graph is transformed into its latent representation using GNN layers. In the context of classification tasks, a classifier loss function f is used. Node classification involves the application of f to each latent representation h_i. Graph classification involves applying f to an aggregation of all latent representations. Link classification entails applying f to two representations h_i and h_j, along with their connection e_{ij}. It is important to note that the classifier f can be substituted with any other loss function depending on the nature of the problem (e.g., MSE for prediction tasks).

ate graph neighbors. And after k iterations, every node representation contains information about its k-hop neighbourhood.

This process is illustrated in Figure 2.10, which demonstrates the propagation of information in a GNN model with 3 MPNN layers. In this figure, the graph at the top represents the input graph. Then, three MPNNs are applied iteratively. At the first layer, h_0 receives information from its direct neighbours h_1, h_2, and h_3, along with edge information. In the second layer, the information received by h_0 also includes the features of h_5, h_4, and h_7 and relationships. After three layers, h_0 has incorporated feature-based and structural information from all nodes in the graph.

Therefore, the number of GNN layers (i.e. the number of Message-Passing steps) is an important hyperparameter of a GNN model that needs effective tuning. In fact, it is one of the main aspects developed in this thesis, regarding the development of GNN models applied to the resolution of PDEs on meshes (see Section 3.5, or more generally Part II).

2.4.5 . Machine Learning on graphs in practice

A variety of Machine Learning tasks can be performed on graphs, which can be classified into three categories:

- **Node-level tasks:** This class of tasks requires node-level representations. The objective may involve node classification or node regression tasks. For ex-

ample, in numerical simulations, predicting node representations in a mesh enables forecasting the physical solution of a Partial Differential Equation (Pfaff et al., 2020; Alet et al., 2019).

- **Edge-level tasks:** This class of tasks requires edge-level representations. Similar to node-level tasks, one can find tasks such as edge classification or predictions. Edge-level representation tasks are particularly useful in recommendation systems where the objective is link prediction (i.e., predicting the existence of an edge between pairs of nodes) (Fan et al., 2019; Zhang and Chen, 2018). The majority of works that have been successful in the task of link prediction are based on Graph Auto-Encoders (GAE) (Kipf and Welling, 2016c).
- **Graph-level tasks:** This class of tasks requires a single representation for the entire graph. Tasks such as graph classification or graph regression fall into this category. Real-world use cases of graph classification and regression often arise from bioinformatics datasets, where, given a graph structure of a molecule or protein, the prediction of a chemical or molecular property is needed (Gilmer et al., 2017; Zhang et al., 2018b).

Figure 2.11 provides a visual representation of these different tasks.

From a practical standpoint, there exist several libraries available for performing Machine Learning on graphs. For instance, in this manuscript, we used Pytorch Geometric[5]. Alternately, TensorFlow provides Spektral[6], while Deep Graph Library (DGL[7]) can be used on both PyTorch and TensorFlow. Additionally, Jax users can use Jraph[8] for similar purposes.

[5] https://pytorch-geometric.readthedocs.io/en/latest/
[6] https://graphneural.network/
[7] https://www.dgl.ai/
[8] https://github.com/deepmind/jraph

3 - Machine Learning for Physics Simulations

Sommaire

Chapter 1 introduced Computational Fluid Dynamics (CFD), which involves predicting the motion of a fluid by numerically approximating solutions to Partial Differential Equations (PDEs) like the Incompressible Navier-Stokes equations (1.15). Chapter 2 provided an overview of Deep Learning (DL) architectures and their practical applications, from using input vectors in Multilayer Perceptrons (MLPs) to handling structured image-like data with Convolutional Neural Networks (CNNs) and unstructured graph-like data with Graph Neural Networks (GNNs). This Chapter serves as a bridge between these two domains, aiming to provide a comprehensive overview of the latest Deep Learning approaches for predicting solutions to PDEs. Additionally, it clarifies the position of this thesis in relation to the current state-of-the-art. Nevertheless, the field of Machine Learning applied to Physics and Numerical Simulations is rapidly evolving and highly dynamic, making it likely that this state-of-the-art is already outdated.

In the past few years, the use of Machine Learning models to predict solutions of PDEs has very rapidly gained significant interest in the community, beginning in the 90s with some pioneering works of Lee and Kang (1990); Dissanayake and Phan-Thien (1994) and Lagaris et al. (1998). Since then, much research has focused on building more complex neural network architectures with a larger number of parameters, taking advantage of the increasing computational power as demonstrated in Smaoui and Al-Enezi (2004); Baymani et al. (2010) or Kumar and Yadav (2011).

3.1 . CNNs for physics simulations

Despite these convincing advances that harnessed Multilayer Perceptrons (see Section 2.1), these methods were quickly overtaken by the tremendous progress made

in the field of Computer Vision and the rise of Convolutional Neural Networks (see Section 2.3), thanks to the pioneering work of LeCun et al. (1995). For additional global information on advances in the field of CNNs, Gu et al. (2018) provides a comprehensive overview. In the realm of fluid mechanics, such networks have been used to solve the Navier-Stokes equations by considering solutions on rectangular grids and treating them as images (Guo et al., 2016; Yilmaz and German, 2017; Illarramendi et al., 2021). A significant amount of research has focused on using CNNs to solve the Poisson equation due to its significant engineering interest, as exemplified in Section 1.2. In Tang et al. (2017), a straightforward CNN architecture predicts the potential electric distribution in a square domain by approximating the solution of 2D and 3D Poisson problems. In Hsieh et al. (2019), a Machine Learning solver with a U-Net architecture (Ronneberger et al., 2015; Brunet et al., 2019) is designed to mimic multigrid methods (Briggs et al., 2000) and provides theoretical convergence guarantees when applied to the resolution of a 2D Poisson equation. In Özbay et al. (2021), a convolutional neural network is trained to solve the inverse Poisson problem through supervised learning. Cheng et al. (2021) use a physics-based loss function with a deep convolutional network to solve the Poisson equation in the context of plasma flows. These different approaches have shown promising results, providing rather accurate approximate solutions that are also computed faster than traditional solvers. However, CNNs can only be used with image-like data: either structured grids with uniform discretization, or projecting solutions on such a grid (Liu et al., 2021), thus adding projection error that degrade the accuracy of the inferred solutions: this makes them unsuitable to handle numerical methods that rely on unstructured meshes (see Chapter 1).

3.2 . GNNs for physics simulations

To address these shortcomings, recent studies have focused on Graph Neural Networks (see Section 2.4), a class of neural networks that can learn from unstructured data. Introduced in Battaglia et al. (2016), GNNs have experienced significant growth and seen a variety of applications thanks to the development of new techniques such as graph convolution (Kipf and Welling, 2016b), edge convolution (Hamilton et al., 2017), graph attention networks (Veličković et al., 2017b), or graph pooling (Lee et al., 2019) to name a few. An exhaustive survey on existing GNN architectures can be found in Wu et al. (2020). Regarding physical applications, several recent works have shown the ability of GNNs to learn dynamical systems accurately. For example, in Chang et al. (2016) and Sanchez-Gonzalez et al. (2018), GNNs are used to learn the motion of discrete systems of solid particles. Sanchez-Gonzalez et al. (2020) extend this approach to learn complex physics, including fluid simulation and solid deformation, considering graph nodes as particles. In Pfaff et al. (2020), the authors simulate the time dynamics of complex systems based on unstructured data. Li et al. (2020d) and Lino et al. (2021b) propose using a multi-level architecture to solve

PDEs on graphs with a larger number of nodes. In Horie and Mitsume (2022), a GNN model is trained through supervised learning to approximate the solution of the incompressible Navier-Stokes equation while preserving the boundary conditions. Similar to CNN-oriented research, some studies have focused on using GNNs to solve the Poisson equation, starting with the work of Alet et al. (2019). Li et al. (2020c) introduce a graph kernel network to approximate PDEs with a specific focus on the resolution of a 2D Poisson problem, and in Chen et al. (2022), a multi-level GNN architecture is trained through supervised learning to solve the Poisson Pressure equation in the context of fluid simulations. These approaches outperform the CNN-based model as they better generalize to meshes with different shapes and sizes, given that the mesh is an inherent part of the input to the model. Nevertheless, these methods still rely solely on supervised learning, requiring computationally expensive ground truth solutions and resulting in a sharp decrease in performances when applied to out-of-distribution examples. Additionally, the explicit consideration of boundary conditions remains elusive, presenting a significant challenge for practical use in industrial processes.

3.3 . The Physics-Informed approach (PINN)

In parallel to these architectural advancements, another research direction focuses on a new class of Deep Learning methods called "Physics-Informed Neural Networks" (PINNs). Introduced in Raissi et al. (2019a) and Raissi and Karniadakis (2018) for resolving known physical models, PINNs offer an innovative approach by directly integrating the PDE residual into the training loss. To achieve this, PINNs use a deep network to compute the PDE solution and leverage Automatic Differentiation to calculate partial derivatives and construct the residual equation. PINN training follows an unsupervised strategy and can be trained with a relatively small amount of data compared to supervised methods. Related approaches also rely on the PDE's weak formulation, i.e., applying a deep network to solve the PDE in variational form. This is the case of the Deep Ritz method (Yu et al., 2018), which formulates the PDE as an equivalent minimization problem, subsequently discretizing and solving it using a deep network combined with a numerical integration method. PINNs are mesh-free methods since they are trained on a carefully selected set of points within the domain known as collocation points. Once trained and provided with an input point within the domain, they yield an approximate solution at that point of the PDE. In Li et al. (2021), the authors investigated various approaches to loss functions, including data-driven, PDE-based (PINNs-like), and energy-based (DeepRitz-like) losses. They noted that the PDE-based loss function has more hyperparameters compared to the energy-based alternative. Additionally, they observed that while the energy-based strategy is more sensitive to the size and resolution of training samples compared to the PDE-based approach, it offers greater computational efficiency. PINNs have gained significant attention in the scientific community, particularly in fluid mechan-

ics, where their flexibility in solving a wide range of forward and inverse problems has motivated numerous studies (Raissi et al., 2019b, 2020; Sun et al., 2020; Cai et al., 2021). This field includes applications for compressible flows (Mao et al., 2020), laminar flows (Rao et al., 2020), turbulent flows (Wu et al., 2018; Yang et al., 2019; Lucor et al., 2021), biomedical flows (Yin et al., 2021; Kissas et al., 2020), and others, as detailed in Cai et al. (2022); Cuomo et al. (2022).

Despite their success, PINNs face significant challenges that hinder their effective deployment in industrial contexts. One major limitation lies in the spectral bias inherent in neural networks. Studies have shown that conventional fully connected architectures, such as those typically used in PINNs, are unable to properly learn functions with high frequencies (Bruna et al., 2013; Cao et al., 2019). Consequently, PINNs often struggle to achieve stable training and accurate predictions, particularly when dealing with PDE solutions that contain high frequencies or multi-scale features (Wang et al., 2022b). Furthermore, Mishra and Molinaro (2023) investigate whether PINNs converge to the correct solution by obtaining error estimates (under mild hypotheses) and identifying possible mechanisms by which PINNs can approximate PDEs. Among these hypotheses is the number of collocation points, which should be sufficiently large, which can be an issue when applied to larger problems. As the domain size increases, so does the complexity of the problem, necessitating larger networks to accurately capture all features. The time and storage requirements for computing partial derivatives through automatic differentiation also increase, which can quickly become overwhelming. Moreover, since the PINN loss function can be highly non-convex, higher problem complexity could lead to challenging optimization problems, resulting in reduced accuracy or no convergence at all (Krishnapriyan et al., 2021). We refer the reader to Section 9.2 for potential solutions to the large-scale issue. Additionally, PINNs are often evaluated on small-scale and simple 2D domains, which are far from industrial requirements. The sampling of collocation points is often problem-dependent, typically requiring prior knowledge of the domain (Nabian et al., 2021), and the distribution of residual points is a crucial parameter in PINNs, as it can dramatically alter the design of the loss function (Mao et al., 2020). Consequently, PINNs may face challenges in generalizing to new scenarios due to significant changes that can occur in the solution to a PDE when considering different domain shapes or boundary conditions, often necessitating retraining to provide satisfactory results. PINNs may also fail to approximate a solution not due to the lack of expressivity in the neural network architecture but due to soft PDE constraint optimization problems, which make the loss function very challenging to optimize (Cuomo et al., 2022). As suggested in (Krishnapriyan et al., 2021), one proposed solution is to use adaptive regularization in the loss function. Finally, boundary conditions constraints can be regarded as penalty terms. Many existing PINN frameworks use a soft approach to constrain the boundary conditions by creating extra loss components defined on the collocation points of boundaries. However, this technique has twofold disadvantages: accurately satisfying the

boundary conditions is not guaranteed, and the assigned weight of the boundary condition loss might affect learning efficiency. Zhu et al. (2021) proposes the Dirichlet boundary conditions in a hard approach by using a specific component of the neural network to purely meet the Dirichlet boundary conditions.

3.4 . Deep Statistical Solvers

The primary contributions of this thesis are closely related to the Deep Statistical Solvers (DSS) method (Donon et al., 2020). DSS was originally designed in the context of Power Grid simulations to better handle changes in the topology of the grid compared to supervised learning approaches. This is achieved by using a *physics-informed* approach, directly minimizing the residual of Kirchhoff's law in the loss function. Another case study in Donon et al. (2020) focuses on solving Poisson problems with Dirichlet boundary conditions. In this case, the physics-informed loss is the residual of the discretized Poisson problem. This approach differs significantly from most previous GNN-based methods, which are based on supervised learning and attempt to minimize the distance between the output of the model and a "ground-truth" solution. As a result, models like DSS can be trained without a large set of training sample solutions, which are often expensive to compute.

The DSS model is trained on a mesh-based architecture and benefits from the chosen discretization scheme, i.e., the Finite Element method, setting it apart from the PINN approach (a mesh-free method). In PINNs, each weight of the neural network influences the solution at all points, which makes the method inconsistent with the locality of the original differential problem. This is in contrast to the DSS method, which aims to learn the discretized operator. DSS uses an iterative architecture, propagating information through a manually set number of Message Passing Neural Network (MPNN) layers (see Section 2.4). Notably, each MPNN layer varies at each iteration. Interestingly, Donon et al. (2020) demonstrates, under mild hypotheses, the consistency and convergence of the approach, but it depends on the number of MPNN layers, which should be directly proportional to the diameter[1] of the meshes at hand.

DSS yields efficient results but has some limitations. It is constrained to geometries of similar sizes (diameters) since it is designed with a fixed number of iterations. To tackle geometries with a larger number of nodes, the number of iterations must be increased accordingly, leading to a significant increase in the size of the model. Additionally, when it comes to solving the Poisson equation, DSS is restricted to Dirichlet boundary conditions and does not explicitly address these boundary conditions. These limitations, among others presented in Chapter 5, hinder the generalization capabilities of the Deep Statistical Solvers model.

[1] the shortest path (node-wise) between the two farthest nodes in a mesh.

3.5 . Thesis contributions

In this manuscript, we build on recent advancements in state-of-the-art models to explore the development of Deep Learning algorithms for solving a wide range of Poisson problems with precision and efficiency. Our specific focus is on enhancing the Deep Statistical Solvers framework by introducing a novel GNN-based physics-oriented model. This model is designed to address the challenges posed by unstructured meshes of varying sizes and shapes while explicitly incorporating boundary conditions.

Two major contributions of this work are presented in Part II. The first contribution, discussed in Chapter 6 and referred to as DS-GPS, focuses on solving Poisson problems with mixed boundary conditions, thereby extending its application to CFD cases. Compared to the original DSS, DS-GPS features a recurrent architecture, resulting in a significantly reduced model size, and incorporates various enhancements. Notably, it follows a node-oriented architecture, allowing for the explicit treatment of boundary conditions by design, and includes an additional autoencoding structure that enhances its flexibility with respect to the initial solution. However, despite the recurrent architecture, where weights are shared between different iterations, the model must still be trained with a fixed number of recurrent iterations (extending the number of iterations is only possible when inferring a solution but still has to be manually set by the user). This limitation results in poor generalization capabilities with meshes of sizes significantly different than the ones used during training. An important observation about this approach is that, under a specific configuration, if the model is trained with a sufficient number of MPNN iterations, it tends to converge toward an equilibrium point. This observation motivated the second contribution of this manuscript.

Chapter 7 introduces our second and main contribution: the Ψ-GNN model. Ψ-GNN is designed to retain the strengths of DS-GPS while achieving better generalization across mesh sizes. It is an Implicit Graph Neural Network model specifically tailored for solving Poisson problems on unstructured meshes with mixed boundary conditions. Leveraging the Implicit Layer theory (Bai et al., 2019), Ψ-GNN automatically determines the required number of Message Passing layers, ensuring robust generalization across various mesh sizes and shapes. Similar to DS-GPS, Ψ-GNN effectively handles boundary conditions through a node-oriented approach and dynamically adapts to initial solutions via an autoencoding process. The model is trained end-to-end, minimizing the residual of the discretized Poisson problem, attempting to learn the underlying physics represented by the discretized Laplace operator. To maintain stability, a regularization cost function constrains the spectral radius of the Machine Learning solver, ensuring reliable convergence. Theoretical analysis confirms that Ψ-GNN exhibits a universal approximation property, underscoring its consistency and effectiveness. However, it is important to note that while Ψ-GNN

significantly enhances generalization, it remains best suited for small-sized problems. This limitation hinders its applicability to industrial processes, which often involve thousands of nodes.

To address this latter limitation and scale these models to larger geometries, we explore, in Part III, the integration of the previously mentioned models with Domain Decomposition methods. The concept is to leverage batch parallel computing on GPU to improve the resolution of subproblems within the context of well-established Schwarz methods. This is still ongoing work, and only the preliminary (but promising) results are presented in this dissertation.

The decision to use GNNs instead of mesh-free PINN methods is driven by practical considerations. The objective is to enhance the resolution of the Poisson Pressure problem within a step of a splitting scheme method, leveraging an existing mesh and computed quantities from previous steps. Therefore, the use of GNNs and their ability to learn from unstructured data was a natural choice. Moreover, we aimed for a model that can be flexible to domains with different shapes and sizes, an option potentially offered by GNNs compared to PINNs. However, the theoretical exploration of GNN models to address spectral bias was beyond the scope of this work. It is worth noting that our intention was never to entirely replace the Poisson Pressure problem with GNN models. Asking a Machine Learning model to provide convergence guarantees and sufficient accuracy to ensure the consistency of a splitting scheme is ambitious. Instead, we aimed to use Machine Learning solutions as complementary tools to existing linear solvers, such as employing them as good initializers or preconditioners.

A potential research direction could involve leveraging the spectral bias instead of attempting to mitigate it. Spectral bias indicates that the network solution struggles to learn high-frequency errors. Interestingly, this property complements classical relaxation methods such as Jacobi or Gauss-Seidel, which excel at quickly damping low-frequency errors but struggle with high-frequency ones (Saad, 2003). Hence, it may be possible to combine the strengths of both approaches to achieve an improved relaxation scheme. For instance, this would involve not only using a Machine Learning solution as an initializer but also incorporating Machine Learning as a correction term in a relaxation scheme to reduce both high and low frequencies of the solution simultaneously. Regarding multi-scale problems, although this aspect has not been investigated in this work, there are no major downsides in the proposed method that indicate significant failures when considering multi-scale problems. However, training might be more challenging. Nevertheless, I am confident that leveraging a hierarchical GNN architecture could potentially address this issue (Gao and Ji, 2019a; Liu et al., 2021).

Part II

Graph Neural Network Solvers for Poisson-like problems

The second Part of this thesis introduces Graph Neural Network (GNN)-based models for solving Poisson problems. The work developed in this study is inspired by and closely related to Deep Statistical Solvers (DSS) (Donon et al., 2020), a new class of trainable solvers for optimization problems on graphs. As detailed in the previous Chapter 3, a noteworthy aspect of the DSS research concerns its ability to solve linear systems arising from discretized partial differential equations. This yields efficient results but also reveals several limitations. The aim of this second Part is to enhance the generalization capabilities of the DSS approach. It is divided into four chapters as follows: Chapter 4 lays the foundation for all the subsequent approaches. Chapter 5 provides a detailed introduction to the original DSS approach (Donon et al., 2020), describing its architecture, evaluating its performance, and discussing its limitations. To address these limitations, two contributions are presented in the subsequent chapters. The first contribution involves using a Recurrent GNN architecture to enhance the capabilities of DSS and is described in Chapter 6. Chapter 7 studies the second contribution, exploring an implicit GNN architecture designed to overcome the drawbacks identified in the previous chapter.

4 - General framework

Sommaire

The present Chapter aims to establish the foundation for all the subsequent methods, particularly by introducing the dataset based on synthetic data. One research direction involves extending the original DSS approach, which primarily focuses on solving Poisson problems with Dirichlet boundary conditions, to handle Poisson problems with mixed boundary conditions, aligning with CFD frameworks (as discussed in Chapter 1). These problem variations are detailed in Section 4.1. The Chapter then introduces the statistical problem common to all subsequent approaches in Section 4.2. Finally, Section 4.3 provides a detailed description of the dataset used in all the experiments.

4.1 . Problem statement

This Section aims to introduce and define the problems addressed in the following of this manuscript. We are interested in solving two different Poisson problems: Poisson problems with Dirichlet boundary conditions, as was done in the original Deep Statistical Solvers study (Donon et al., 2020), and Poisson problems with mixed boundary conditions, aligning with CFD frameworks.

Let $\Omega \subset \mathbb{R}^2$ be a bounded open domain with smooth boundary $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$. To ensure the existence and uniqueness of the solution, special constraints (referred to as *boundary conditions*) must be specified on the boundary $\partial\Omega$ of Ω (Langtangen and Mardal, 2019): *Dirichlet conditions* assign a known value for the solution u on $\partial\Omega_D$, and *homogeneous Neumann boundary conditions* impose that no part of the solution u is leaving the domain through $\partial\Omega_N$ (see Chapter 1 for additional details). More formally, let f be a continuous function defined on Ω, g a continuous function defined on $\partial\Omega_D$ and n the outward normal vector defined on $\partial\Omega$.

Poisson problem with Dirichlet boundary conditions

The Poisson problem with Dirichlet boundary conditions consists of finding a real-valued function u, defined on Ω, that satisfies:

$$\begin{cases} -\Delta u &= f & \in \Omega \\ u &= g & \in \partial\Omega \end{cases} \tag{4.1}$$

Poisson problem with Mixed boundary conditions

The Poisson problem with mixed boundary conditions (i.e. Dirichlet and homogeneous Neumann boundary conditions) consists of finding a real-valued function u, defined on Ω, that satisfies:

$$\begin{cases} -\Delta u &= f & \in \Omega \\ u &= g & \in \partial\Omega_D \\ \frac{\partial u}{\partial n} &= 0 & \in \partial\Omega_N \end{cases} \tag{4.2}$$

For both problems (4.1) and (4.2), except in very specific instances, no analytical solution can be derived, and its solution must be numerically approximated: the domain Ω is first discretized into an unstructured mesh, denoted Ω_h. The problem is then spatially discretized using the Finite Element Method (FEM) (Reddy, 2019), also described in Section 1.3. The approximate solution u_h is sought as a vector of values defined on all N degrees of freedom of Ω_h. Besides, N depends on the type of elements chosen (i.e., the precision order of the approximation). In this thesis, we restrict ourselves to first-order Lagrange elements, hence N matches the number of nodes in Ω_h. The discretization of the variational formulation using Galerkin's method yields the resolution of the linear system:

$$AU = B \tag{4.3}$$

where $A \in \mathbb{R}^{N\times N}$ is sparse and represents the discretization of the continuous Laplace operator, the vector $B \in \mathbb{R}^N$ comes from the discretization of the forcing term f and of the mixed boundary conditions, and $U \in \mathbb{R}^N$ is the solution vector to be sought. Details regarding the derivation of the linear system (4.3) can be found in Section 1.3.

Let $\mathcal{F}$ be a set of continuous functions on Ω and $\mathcal{G}$ a set of continuous functions on $\partial\Omega_D$. We denote as $\mathcal{P}$ a set of Poisson problems, such that any element $E_p \in \mathcal{P}$ is described as a triplet:

$$E_p = (\Omega_p,\ f_p,\ g_p)$$

where $f_p \in \mathcal{F}$ and $g_p \in \mathcal{G}$. For all $E_p \in \mathcal{P}$, let $E_{h,p} \in \mathcal{P}_h$ denote its discretization, such that:

$$E_{h,p} = (\Omega_{h,p},\ A_p,\ B_p)$$

where A_p and B_p are defined similar to (4.3).

The fundamental concept, common to all models introduced in the subsequent chapters, is, considering a continuous Poisson problem $E_p \in \mathcal{P}$, to build a Machine Learning solver, parametrized by a vector θ, which outputs an approximate solution U_p of its respective discretized form $E_{h,p} \in \mathcal{P}_h$:

$$U_p \ = \ \text{Solver}_\theta\left(E_{h,p}\right) = \ \text{Solver}_\theta\left(\Omega_{h,p},\ A_p,\ B_p\right) \tag{4.4}$$

4.2 . Statistical problem

In (4.3), it should be noted that the structure of matrix A encodes the geometry of its corresponding mesh (i.e. A can be viewed as the adjacency matrix of its corresponding mesh). Indeed, for each node, using first-order finite elements leads to the creation of local stencils, which represent local connections between mesh nodes. The Machine Learning solvers introduced in the following chapters are built using Graph Neural Networks (GNNs), which can handle graph data effectively (see Section 2.4). When using higher-order finite elements, building a graph from the stiffness matrix A is still possible, but its relationship to the mesh becomes more intricate and requires extra care. To simplify the process and align with the original DSS study, this work solely focuses on the use of first-order finite elements.

A crucial upside of using GNNs in physics simulations is related to the right treatment of boundary conditions. In the linear system (4.3), implementing Dirichlet boundary conditions involves modifying the sparse matrix A by setting the off-diagonal elements of the rows corresponding to Dirichlet nodes to 0 and the diagonal element to 1. Similarly, the corresponding index in vector B is set to the discrete value of g. As a result, when solving (4.3), Dirichlet conditions are enforced, ensuring that $u = g$ (refer to Section 1.3). Figure 4.1a displays the sparsity pattern of matrix A for a problem with 17 nodes before applying Dirichlet boundary conditions. In contrast, Figure 4.1b illustrates the sparsity pattern of matrix A for the same problem after applying Dirichlet boundary conditions. This comparison reveals that such modifications break the symmetry of the matrix A. As a consequence, some Interior connections (blue squares in Figure 4.1b) and Neumann connections (yellow squares in Figure 4.1b), linked to Dirichlet nodes no longer have symmetrical counterparts. As a result of this specific construction, the induced graph is *directed* at those Dirichlet boundary nodes, sending information only to its neighbours without

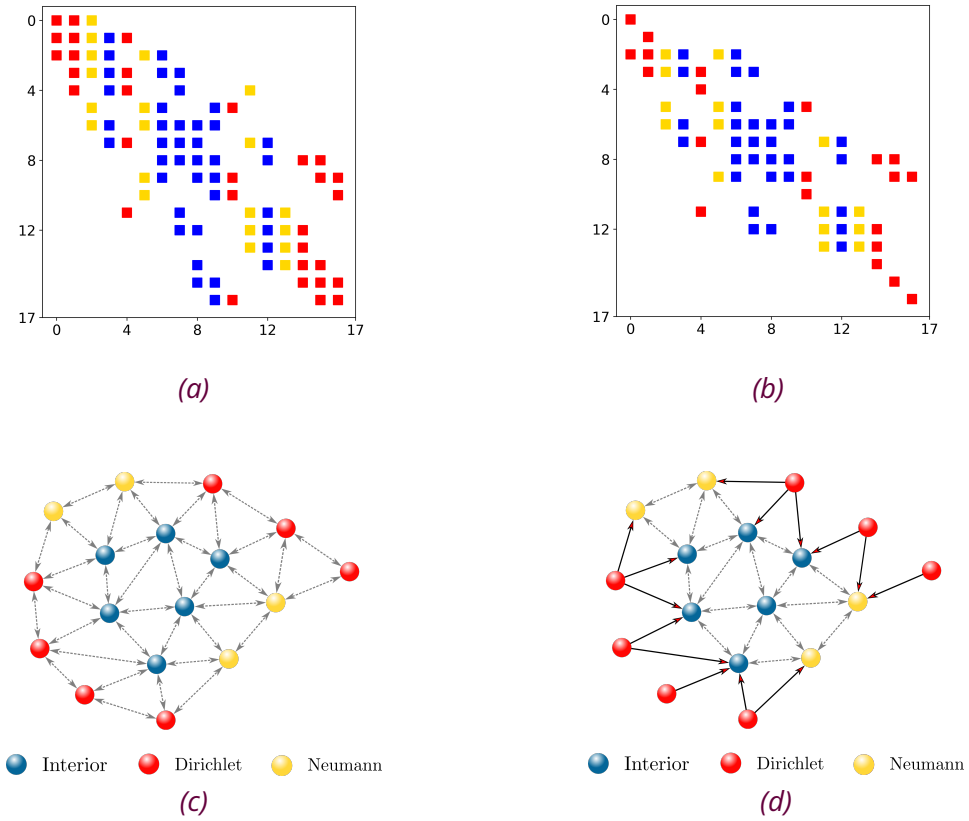


Figure 4.1: Figure 4.1a illustrates the sparsity pattern in matrix A, obtained by discretizing the Laplace operator in (4.2) using FEM for a problem with 17 nodes before applying Dirichlet boundary conditions. Using this matrix as an adjacency matrix, the induced graph is shown in 4.1c, resulting in a fully undirected graph. Figure 4.1b displays the sparsity pattern of the same matrix A after applying Dirichlet boundary conditions. The related graph is shown in 4.1d, resulting in an undirected graph for Interior and Neumann nodes (blue and yellow nodes) and a directed graph for Dirichlet nodes (red nodes). For both 4.1a and 4.1b, Interior, Neumann and Dirichlet connections correspond to the blue, yellow and red squares, respectively.

receiving any. Conversely, Interior and Neumann nodes induce an *undirected* graph with bi-directional edges, thus exchanging information with each other. To further illustrate this, Figure 4.1c displays the graph obtained using the adjacency matrix depicted in 4.1a (i.e. *before* applying Dirichlet boundary conditions). Since 4.1a is symmetrical, all edges are bi-directional. On the contrary, Figure 4.1d illustrates the graph obtained by considering the adjacency matrix 4.1b (i.e. *after* applying Dirichlet boundary conditions), showcasing the specific directionality of the edges based on the node types.

A discretized Poisson problem $E_h = (\Omega_h,\ A,\ B)$ with N degrees of freedom can then be interpreted as a graph problem $G = (N,\ A,\ B)$, where N is the number of nodes in the graph, $A = (a_{ij})_{(i,j)\in[N]^2}$, is the weighted adjacency matrix that rep-

resents the interactions between the nodes and $B = (b_i)_{i\in[N]}$ denotes some local external inputs applied to each node i in the graph. Vector $U = (u_i)_{i\in[N]}$ represents the state of the graph, u_i being the state of the node i.

Additionally, let $\mathcal{S}$ be the set of all such graphs G, $\mathcal{U}$ the set of all states U, and $\mathcal{L}_{\text{res}}$ the real-valued function which computes the mean squared error (MSE) of the discretized residual equation such that:

$$\mathcal{L}_{\text{res}}(U,\, G) = \text{MSE}\,(AU - B) \tag{4.5}$$

$$= \frac{1}{N} \sum_{i\in[N]} \Big(-b_i + \sum_{j\in[N]} a_{i,j} u_j \Big)^2 \tag{4.6}$$

The objective is, given a graph G in $\mathcal{S}$, to find an optimal state in $\mathcal{U}$ that minimizes (4.6). Therefore, we define a Machine Learning solver, parameterized by θ, that predicts from G a solution U in order to solve the following statistical problem:

Given a distribution $\mathcal{D}$ on space $\mathcal{S}$ and a loss function $\mathcal{L}_{res}$, solve:

$$\theta^\star = \underset{\theta}{\text{argmin}} \underset{G\sim\mathcal{D}}{\mathbb{E}} \left[\mathcal{L}_{\text{res}}\left(\text{Solver}_\theta\left(G\right), G\right)\right] \tag{4.7}$$

In practice, the expectation (4.7) is empirically computed using a finite (and sufficiently large) number of graphs sampled from $\mathcal{D}$. The result of this empirical minimization is a parameter $\widehat{\theta}$. The Machine Learning solver can then be used at inference to compute, for any $G \in supp(\mathcal{D})$, an approximate solution $\widehat{U}$ of $U^\star$ such that:

$$\widehat{U} = \text{Solver}_{\widehat{\theta}}(G) \tag{4.8}$$

This approach differs from most previous Machine Learning methods, which solve a supervised learning problem, training the model by minimizing the distance with a "ground-truth" solution. Although these supervised learning approaches have shown promising results (see e.g., Chapter 3), they require a large dataset containing such ground-truth solutions, which can be particularly expensive to obtain in the context of physics simulations. In contrast, the proposed DSS-based approach does not require such solutions as it directly minimizes the discretized objective function of the problem. Figure 4.2 illustrates the difference between the supervised (i.e. using a ground-truth solution) and the considered unsupervised approach.

All the work presented in the following adheres to this framework, and try to solved the statistical problem described above. The focus now lies on the architectural specificities of the considered GNN solvers. For each model, various experiments

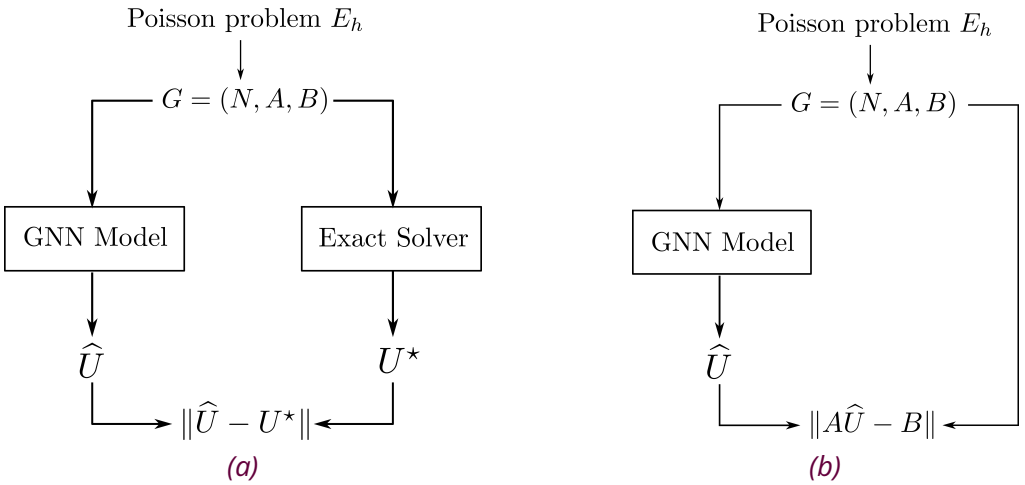


Figure 4.2: Figure 4.2a illustrates the supervised approach. In this configuration, the GNN model is trained by minimizing the difference between the model output $\widehat{U}$ and a ground-truth solution $U^\star$. In contrast, Figure 4.2b displays the unsupervised approach used in this thesis. In this configuration, the GNN model is trained by directly minimizing the discretized residual equation.

will showcase its strengths and limitations, encouraging continuous improvement. To provide a fair comparison between each method, these experiments require a common dataset to be trained and tested on, which is the topic of next Section 4.3.

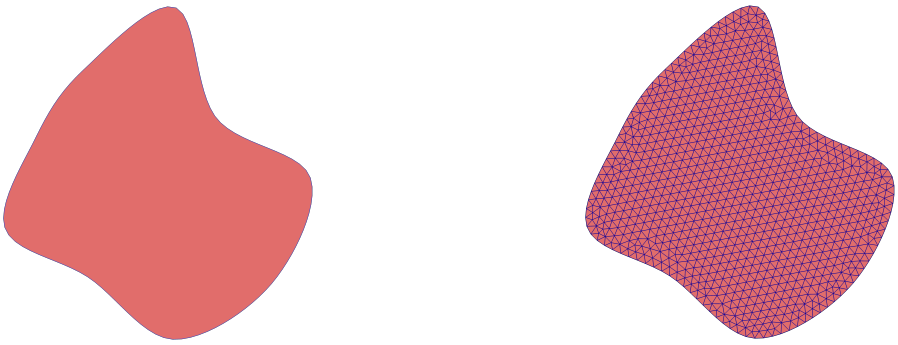
4.3 . Dataset description

This section aims to explain the generative process used to construct the datasets for training and testing the subsequent models. Multiple datasets will be considered, but each one will consist of various numbers of training, validation, and test samples of discretized Poisson problems derived from (4.1) or (4.2). The specific number of samples for each experiment will be provided where appropriate.

Each sample is generated as follows: Random 2D domains Ω are generated using 10 points, randomly sampled in the unit sphere. These points are then connected using Bezier curves to form the boundary of domain Ω. The "MeshAdapt" mesher from GMSH[1] (Geuzaine and Remacle, 2009) is used to discretize Ω into an unstructured triangular mesh Ω_h. Figure 4.3 illustrates a random 2d domain Ω and its corresponding mesh Ω_h.

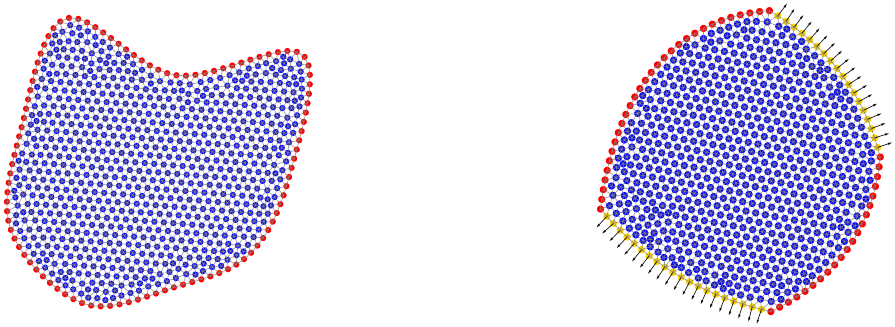
The upcoming analysis will involve multiple datasets to evaluate the performance of various models, considering variations in the number of nodes and the type of boundary conditions. For each experiment, a specific range for the number of nodes will be specified. To achieve different node counts, two configurations will be considered: the first involves densifying geometries by adjusting their mesh sizes,

[1] https://gmsh.info/



(a) Continous domain (b) Mesh

Figure 4.3: 4.3a displays a random 2d domain Ω and 4.3b its associated mesh Ω_h.



(a) (b)

Figure 4.4: Random meshes generated to build the various datasets. 4.4a: geometry used for solving Poisson problems with Dirichlet boundary conditions. 4.4b: geometry used for solving Poisson problems with mixed boundary conditions. Blue, red and yellow nodes correspond to Interior, Dirichlet and Neumann nodes, respectively. The arrows in 4.4b depict the outward normal vectors.

and the second involves preserving the mesh size fixed while increasing the radius of each geometry. These configurations will be detailed for each experiment, where appropriate.

When considering Poisson problems with Dirichlet boundary conditions (4.1), Dirichlet boundary nodes are applied to the entire boundary of the mesh. An example of this configuration is given in Figure 4.4a. When considering Poisson problems with mixed boundary conditions (4.2), the boundary is randomly divided into four sections and Dirichlet boundary conditions are applied on two opposite sections, while Neumann boundary conditions are imposed on the remaining opposite sections. This configuration is illustrated in Figure 4.4b.

(a) Forcing function

(b) Boundary function

Figure 4.5: Figures 4.5a and 4.5b display the discrete values of a forcing term f and a boundary function g, on a mesh sampled from the test set. The coefficients of f and g, uniformly sampled in $[-10, 10]$, are $r_1 = 3.2$, $r_2 = -7.5$, $r_3 = 1.1$, $r_4 = 5.7$, $r_5 = -9.5$, $r_6 = 0.47$, $r_7 = -8.8$, $r_8 = 9.11$, and $r_9 = 3.5$. The considered mesh is shown as a grey shadow in the plot.

Forcing functions f and boundary functions g from (4.1) or (4.2) are defined as random quadratic polynomials with coefficients sampled from uniform distributions and are given by:

$$f(x, y) = r_1(x - 1)^2 + r_2 y^2 + r_3 \qquad (x, y) \in \Omega \tag{4.9}$$

$$g(x, y) = r_4 x^2 + r_5 y^2 + r_6 xy + r_7 x + r_8 y + r_9 \qquad (x, y) \in \partial\Omega \tag{4.10}$$

where $(r_i)_{i \in [1, \cdots, 9]}$ are uniformlysampled in $[-10, 10]$. Figure 4.5 illustrates examples of such f and g functions on a sampled mesh.

5 - Deep Statistical Solvers

Sommaire

This chapter introduces the Deep Statistical Solvers (DSS) model. To start with, Section 5.1 gives a brief reminder of the essential aspects developed in previous Chapter 4. Next, Section 5.2 presents the methodology concerning the development of the DSS model, including its architecture, in 5.2.1, and training materials, in 5.2.2. Section 5.3 investigates the performance of DSS, and Section 5.4 highlights its limitations.

5.1 . Introduction

The original Deep Statistical Solvers study (Donon et al., 2020) focuses on solving Poisson problems with Dirichlet boundary conditions, described in (4.1). The DSS approach aligns with the previously mentioned statistical problem, which was developed in Section 4.2.

To summarize and using the same notations, the idea is to define a Graph Neural Network (GNN)-based model, parametrized by a vector θ, that predicts from a discretized Poisson problem defined as a graph $G = (N, A, B)$ a solution U in order to solve the following statistical problem:

Given a distribution $\mathcal{D}$ on space $\mathcal{S}$ and a loss function $\mathcal{L}_{res}$, solve:

$$\begin{aligned}\theta^\star &= \underset{\theta}{\operatorname{argmin}} \underset{G\sim\mathcal{D}}{\mathbb{E}} \left[\mathcal{L}_{\text{res}}\left(\text{DSS}_\theta\left(G\right), G\right)\right] \\ &= \underset{\theta}{\operatorname{argmin}} \underset{G\sim\mathcal{D}}{\mathbb{E}} \left[\mathcal{L}_{\text{res}}\left(\text{DSS}_\theta\left(N, A, B\right), G\right)\right]\end{aligned}$$

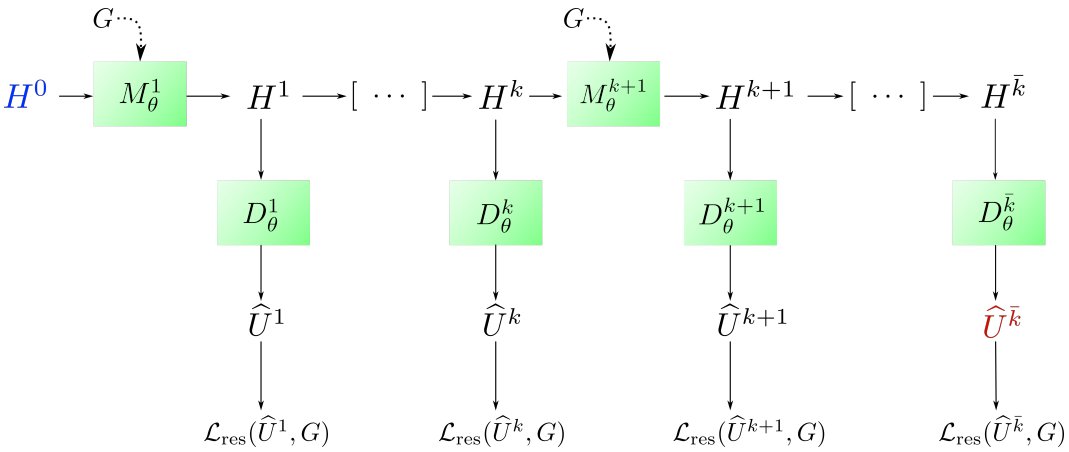


Figure 5.1: Visualization of the original DSS architecture (Donon et al., 2020): The model is initialized with a null latent state H^0. An iterative process then propagates the information for $\bar{k}$ iterations using $M_\theta^{\bar{k}}$ distinct blocks of MPNNs. During each iteration k, a Decoder D_θ^k translates the latent state H^k into a physical state $\widehat{U}^k$, upon which an intermediate loss is computed. The green squares correspond to trainable functions, and their weights are learned during training. The final state $\widehat{U}^{\bar{k}}$, shown in red in the figure, represents the output of the model.

Now, the focus lies on exploring the architecture of the DSS model.

5.2 . Methodology

This section gives the methodology used in Deep Statistical Solvers. Section 5.2.1 introduces the architecture of DSS, and Section 5.2.2 provides some specifics on the training procedure.

5.2.1 . Architecture

In Chapter 2, Section 2.4 introduced Graph Neural Networks (GNNs) as a specific Machine Learning architecture capable of handling graph data efficiently. In the Deep Statistical Solvers model, the authors decided to leverage the Message Passing Neural Network (MPNN) framework. Among the various GNN operators, MPNNs are considered one of the most versatile and particularly well-suited structures for physics applications.

The overall architecture of DSS is displayed in Figure 5.1. It consists of an iterative process that acts on a latent state $H = (H_i)_{i\in[N]} \in \mathcal{H}$ with $H_i \in \mathbb{R}^d$, $d \geq 1$ for $\bar{k}$ iterations. The entire architecture can be divided into three steps: an Initialization step, a Message Passing step and a Decoding step, described as follows.

Initialization

To start with, all latent states in H^0 are initialized to a null vector. The latent state H can be interpreted as an embedding of the physical state U into a space of a larger dimension. This architectural choice is motivated by the need to provide enough space for information propagation throughout the whole network.

Message Passing

The Message Passing step, responsible for the flow of information within the graph, performs $\bar{k}$ updates on the latent state variable H using $\bar{k}$ updating blocks of neural networks $\left(M_\theta^k\right)_{1\leq k\leq \bar{k}}$.

To achieve this, at each iteration k, two different messages $\phi_{\rightarrow}^k$ and $\phi_{\leftarrow}^k$ are first computed using multi-layer perceptrons (MLPs) $\Phi_{\rightarrow,\theta}^{k+1}$ and $\Phi_{\leftarrow,\theta}^{k+1}$. These messages correspond to outgoing and ingoing links and are defined, node-wise, as follows:

$$\phi_{\rightarrow,i}^{k+1} = \sum_{j\in\mathcal{N}(i)} \Phi_{\rightarrow,\theta}^{k+1}(H_i^k, H_j^k, a_{ij}) \tag{5.1}$$

$$\phi_{\leftarrow,i}^{k+1} = \sum_{j\in\mathcal{N}(i)} \Phi_{\leftarrow,\theta}^{k+1}(H_i^k, H_j^k, a_{ji}) \tag{5.2}$$

where a_{ij} and a_{ji} corresponds to the coefficients of A. The updated latent state H^{k+1} is then computed using an MLP Ψ_θ^{k+1} in a ResNet-like (He et al., 2015) fashion such that, node-wise:

$$H_i^{k+1} = H_i^k + \alpha\, \Psi_\theta^{k+1}(H_i^k, b_i, \phi_{\rightarrow,i}^{k+1}, \phi_{\leftarrow,i}^{k+1}) \tag{5.3}$$

For purpose of clarity, operations (5.1), (5.2) and (5.3) can be grouped into a single updating block M_θ^{k+1} such that the next latent state H^{k+1} is computed as follows:

$$H^{k+1} = M_\theta^{k+1}(H^k,\, G) \tag{5.4}$$

Decoder

A Decoding step is applied after each iteration to convert the latent state H^{k+1} into a meaningful actual state $\widehat{U}^{k+1}$ by using an MLP D_θ^{k+1} such that :

$$\widehat{U}^{k+1} = D_\theta^{k+1}(H^{k+1}) \tag{5.5}$$

5.2.2 . Training materials

This section aims to provide training materials, describing the loss used to train the model and detailing additional data pre-processing.

Training loss

The final state $\widehat{U}^{\bar{k}}$ represents the actual output of the algorithm, which corresponds to the approximate solution $\widehat{U}$ from equation (4.8). At first, the authors proposed constructing the training loss by only considering the final state. However, it turned out to be much more stable when all intermediate states were involved. Therefore, the training loss is computed as a discounted sum of all intermediate losses, as follows:

$$\text{Training Loss } = \sum_{k=1}^{\bar{k}} \gamma^{\bar{k}-k} \mathcal{L}_{\text{res}}(\widehat{U}^k,\ G) \tag{5.6}$$

where $\gamma \in (0, 1)$ is a decay factor so that the last partial solutions have more weight than the early ones.

All the trainable functions $\Phi^k_{\rightarrow,\theta}$, $\Phi^k_{\leftarrow,\theta}$, Ψ^k_θ, and D_θ, have distinct weights. They are all trained simultaneously, with the gradient of the training loss (5.6) backpropagated through the model. Additionally, the model has several hyperparameters, including the latent state dimension d, the number of iterations $\bar{k}$, the update coefficient α in (5.3), and the discounted coefficient γ from (5.6), which need to be tuned for optimal behaviour of the training process.

One key question concerns choosing an appropriate value for $\bar{k}$. The authors of DSS suggest that the nature of information exchange may vary between the beginning and end of the process. This intuition motivated the construction of a model where the trainable blocks differ at each iteration. However, this choice comes at the expense of a fixed amount of propagation step $\bar{k}$. While $\bar{k}$ could be treated as a regular hyperparameter to tune, another approach was considered. Indeed, one Message Passing step in the network enables a local propagation of information, where a node exchanges information with its immediate neighbours. When performing $\bar{k}$ MPNNs in DSS, we then propagate the information from one node to other nodes located $\bar{k}$ connections away. To ensure that the information has propagated between all nodes in the graph, the number of iterations $\bar{k}$ is fixed to the average diameter[1] of the considered meshes in the dataset. This intuition is actually theoretically demonstrated by the authors of DSS.

[1]The shortest path (node-wise) between the two farthest nodes in a mesh.

Change of variables

To ensure optimal performance of a Machine Learning model, it is crucial to properly normalize the input data. Failing to do so, particularly when the data have varying ranges of values, can often lead to training issues such as exploding gradients or challenges in finding global/local minima. In DSS, the authors propose a change of variable to allow for better normalization as well as lighter storage.

To start with, the elements of B defined in the linear system (4.3) constitute a multimodal distribution. Indeed, coefficients b_i that correspond to Dirichlet boundary nodes have different ranges than the ones representing Interior nodes. To tackle this issue, the following change of variable is considered by transforming B into a vector $B' = (b'_i)_{i\in[N]}$ such that:

$$b'_i = \begin{cases} \begin{bmatrix} b_i & 0 & 0 \end{bmatrix} & \text{if} \quad \text{Interior} \\ \begin{bmatrix} 0 & 1 & b_i \end{bmatrix} & \text{if} \quad \text{Dirichlet} \end{cases} \tag{5.7}$$

As mentioned in Section 4.2, the application of Dirichlet boundary conditions alters the shape of matrix A. For a row i of A corresponding to Dirichlet nodes, the coefficients are of the form $a_{ii} = 1$ and $a_{ij} = 0$ for $i \neq j$. Moreover, for Interior nodes, the corresponding coefficients follow the subsequent redundant formula:

$$a_{ii} = -\sum_{j\in[N]\setminus\{i\}} a_{ij}$$

As a result, for Interior nodes, the diagonal information can always be retrieved from the off-diagonal components. To allow for lighter storage, the following change of variables is considered by transforming the matrix A into a matrix $A' = (a'_{ij})$ such that :

$$a'_{ij} = \begin{cases} a_{ij} & \text{if } i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

The changes of variables (5.8) and (5.7) necessitate modifying the loss function (4.6) to a new equivalent loss, denoted as $\mathcal{L}'_{\text{res}}$, which is defined as follows:

$$\mathcal{L}'_{\text{res}}(U,\ G) = \sum_{i\in[N]} \left((1 - b'^2_i)(-b'^1_i) + b'^2_i(u_i - b'^3_i) + \sum_{j\in[N]} a'_{ij}(u_j - u_i) \right)^2 \tag{5.9}$$

where b_i^p denotes the p-th component of the vector b'_i. In essence, this change of variables allows for more effective normalization by independently normalizing the b_i coefficients depending on their type (Interior or Dirichlet nodes) and for lighter storage of matrix A.

Data normalization

Considering these changes in variables, all the input data can now be normalized independently. To achieve this, we calculate $\mu_{A'}$ and $\sigma_{A'}$, which are the mean and standard deviation of A', computed on the entire training dataset. Similarly, we calculate $\mu_{B'}$ and $\sigma_{B'}$, which are the mean and standard deviation of B', computed on the entire training dataset. We then introduce the matrix $\widetilde{A}' = (\widetilde{a}'_{ij})_{i,j\in[N]}$ and the vector $\widetilde{B}' = (\widetilde{b}'_i)_{i\in[N]}$ as the normalized counterparts of A' and B', defined as follows:

$$\widetilde{a}'_{ij} = \frac{a'_{ij} - \mu_{A'}}{\sigma_{A'}} \qquad\qquad \widetilde{b}'_i = \frac{b'_i - \mu_{B'}}{\sigma_{B'}}$$

To summarize, during inference, the model requires normalized inputs (i.e. in equations (5.1), (5.2) and (5.3)), which are $\widetilde{A}'$ and $\widetilde{B}'$. Meanwhile, during training, the loss function used is $\mathcal{L}'_{\text{res}}$. It is important to note that when evaluating the loss function, the non-normalized data A' and B' are used.

5.3 . Experiments & Results

This section aims to present the results of the Deep Statistical Solvers model in its original configuration. Section 5.3.1 describes the experimental setup, while Section 5.3.2 describes the results.

5.3.1 . Experimental setup

This section introduces the experimental setup, presenting the dataset, the metrics, and the hyperparameters employed.

Synthetic dataset

The dataset used in this experiment consists of $6000/2000/2000$ training/validation/test samples of Poisson problems with Dirichlet boundary conditions, generated following the process described in Section 4.3, and more precisely in Figure 4.4a. All meshes in this dataset have approximately 500 nodes, with the number of nodes ranging precisely from 427 for the geometry with the fewest nodes to 556 for the geometry with the most nodes. This represents an average diameter of 30.

Metrics

In this context, we consider the solution of the discretized Poisson problem, given by the classical LU decomposition method, as the 'ground truth.' The reported metrics include the Residual Loss (Equation 4.6) and the Mean Squared Error (MSE) between the output of the model and the 'ground-truth' LU solution.

Model setup

The original Deep Statistical Solvers model is implemented using TensorFlow. Here, the results are obtained after reproducing the model in PyTorch using the Graph Neural Network library PyTorch Geometric. The model is trained using the original hyperparameters: the number of iterations $\bar{k}$ is set to the average diameter of the meshes in the dataset, which is 30, and the dimension d of the latent space $\mathcal{H}$ is set to 10. Each neural network in Equations (5.2), (5.1), (5.3), and (5.5) has one hidden layer of dimension 10 with a ReLU activation function. All model parameters are initialized using Xavier initialization (Glorot and Bengio, 2010). In Equation (5.3), coefficient α is 10^{-3}, and in the training loss, the discount factor γ is 0.9. Training is performed for 400 epochs on 2 P100 GPU using the Adam optimizer with a learning rate of 10^{-2} and a batch size of 50. Gradient clipping is employed to prevent exploding gradient issues and is set to 10^{-2}.

5.3.2 . Results

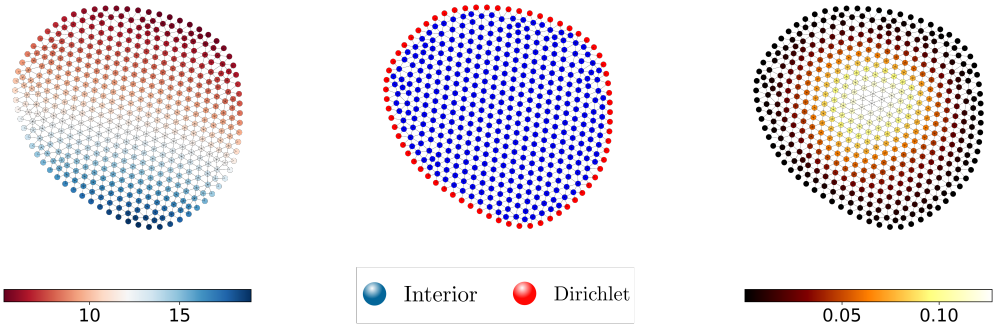
Table 5.1 presents the results of DSS averaged across the entire test set, highlighting various metrics. These results validate the effectiveness of the DSS approach in accurately solving Poisson problems with Dirichlet boundary conditions. Furthermore, they align closely with the ones found in the original Deep Statistical Solvers study, validating our implementation.

Metrics	Residuals (10^{-4})	MSE w/LU	Nb of weights
DSS	2.25 ± 2	0.03 ± 0.02	36930

Table 5.1: Results of DSS averaged over the whole test set.

Additionally, Figure 5.2 provides a visual representation of the results obtained for a single sample from the test set with 465 nodes. At the final iteration, the computed metrics are as follows: a Residual error of 1.9×10^{-4} and an MSE w/LU of 3.7×10^{-2}. Of particular interest, Figure 5.2c illustrates the map of the L2 error between the data-driven solution and the LU solution. These results reveal that the highest errors are concentrated in the central region of the geometry.

To explain this phenomenon, Figure 5.3 illustrates the evolution of the L2 error map throughout the 30 iterations of the DSS model. This result suggests that information



(a) Data-driven solution *(b) Node types* *(c) Error map*

Figure 5.2: Illustration of the resolution of a Poisson problem with 465 nodes, extracted from the test set. Figure 5.2a displays the data-driven solution, while Figure 5.2c shows the map of error between the LU solution and the data-driven solution. Figure 5.2b displays the different types of nodes, blue for Interior nodes and red for Dirichlet nodes.

effectively propagates from the boundary nodes toward the centre of the domain, resulting in the highest error values being situated farthest from the boundary. This behaviour can be attributed to the structure of the graphs generated for such problems, as detailed in Section 4.2. Indeed, Figure 4.1d from Section 4.2 illustrates that these graphs are directed at the Dirichlet boundary nodes, driving the flow of information from these nodes toward the centre of the mesh.

5.4 . Limitations

Previous section 5.3 has demonstrated the effectiveness of the Deep Statistical Solvers method in solving Poisson problems with Dirichlet boundary conditions. However, the DSS approach also encounters several limitations.

Dependancy to the numerical method

To predict the solution of a Poisson problem, the DSS model requires as inputs the graph structure as well as coefficients from matrix A and vector B arising from (4.3). While these coefficients are necessary for training the model due to the structure of the loss function (4.6), one research direction involves exploring methods to avoid using them during inference: once the model is trained, it becomes possible to infer a solution based only on the original data problem and graph structure, eliminating the need to derive the linear system (4.3). This approach should be beneficial for tackling larger problems: avoiding the computation of the linear system can significantly reduce the computational cost and make the method independent of the

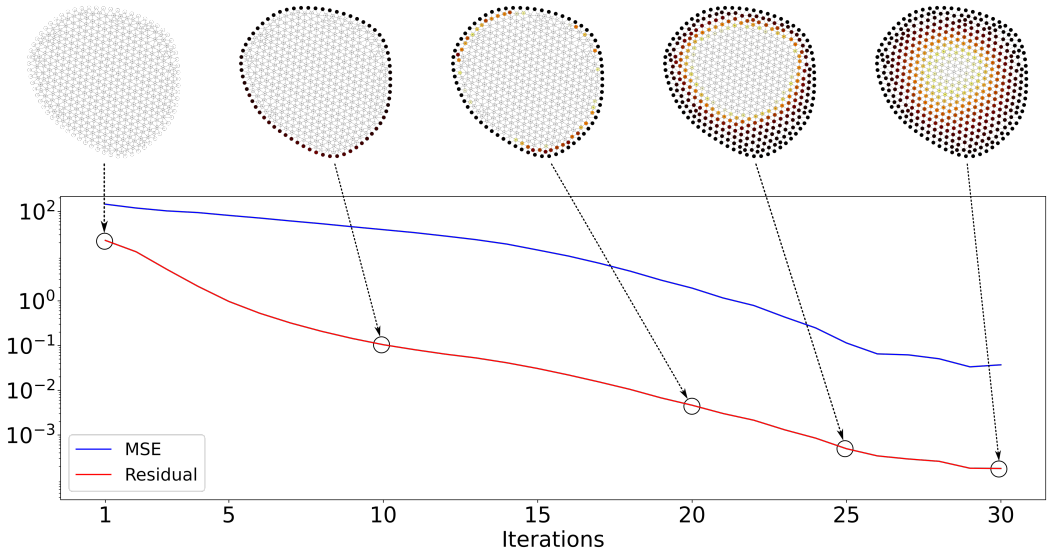


Figure 5.3: Considering the resolution of the same problem as in Figure 5.2, illustration of the propagation of the error across the 30 iterations of the DSS model. The two curves represent the evolution of the Residual (in red) and the MSE (in blue) across these iterations, achieving a precision of $1.9e{-4}$ for the Residual error and $3.7e{-2}$ for the MSE.

numerical method.

Fixed size meshes

The architecture of DSS is designed with a fixed number of iterations corresponding to a fixed number of messages propagated through the network, as discussed in Section 5.2.1. This constraint limits the meshes in the dataset to have approximately the same size in terms of the number of nodes. As a result, the generalization capabilities of the model are limited, as illustrated in Figure 5 of Donon et al. (2020). Furthermore, in order to consider meshes with a larger number of nodes, the number of iterations must be increased accordingly, resulting in a larger Machine Learning model. Our objective in the following is to develop a GNN-based architecture that can adjust its iteration count depending on the size of the considered mesh. However, in order to achieve such flexibility, we will rather consider using a Recurrent architecture (investigated in Chapter 6), iterating over the same updating block, or an Implicit approach (presented in Chapter 7).

Model initialization

Another characteristic of the Deep Statistical Solvers architecture is that it propagates information through a latent state variable. This latent state variable has a higher dimension than the node state variable (i.e., $d \geq 1$). While this approach is

reasonable, as Message Passing requires sufficient space to propagate information, all latent states are initialized to null vectors. One potential research direction is to focus on building an encoder that can map the physical space to the latent space, enabling appropriate initialization of any provided initial solution. Moreover, considering a highly flexible model, it becomes possible to adjust the iteration counts based on the proximity of the initial solution to the final solution.

Boundary conditions

When dealing with Machine Learning applied to physics simulations, properly handling boundary conditions is crucial. In the Deep Statistical Solvers approach, there is no explicit treatment of boundary conditions, as illustrated in Figure 5.4. This figure displays the MSE w/LU for Dirichlet boundary nodes across the inference iterations. It reveals that the Dirichlet boundary conditions, which are inherently imposed on the original problem, are "learned" during inference. While the final error remains low, providing guarantees about the correct treatment of boundary conditions remains challenging. Additionally, in order to be able to cope with CFD applications, it is essential to extend this model to handle Poisson problems with Neuman and mixed boundary conditions, i.e., designed to solve problems like (4.2).

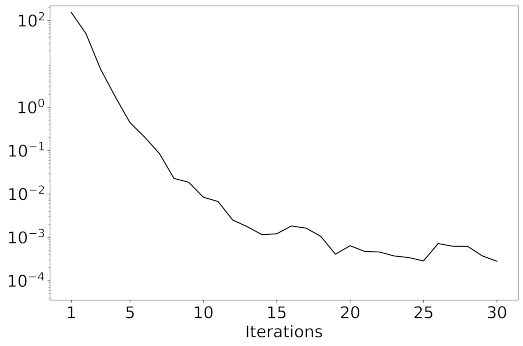


Figure 5.4: Evolution of the MSE w/LU at Dirichlet nodes across the different iterations of an inference.

6 - DS-GPS : A Recurrent GNN Solver

Sommaire

This chapter introduces DS-GPS, a Recurrent Graph Neural Network model to solve Poisson-like problems. DS-GPS is specifically designed to enhance the generalization capabilities of the previously described Deep Statistical Solvers method (Chapter 5). To start with, Section 6.1 gives an introduction of DS-GPS, explaining the diverse strategies used to address the limitations of DSS. Next, Section 6.2 presents the methodology used to construct and train DS-GPS, including its architecture in 6.2.2 and training materials in 6.2.3. Section 6.3 investigates the performance of the proposed model, and Section 6.4 highlights its limitations.

6.1 . Introduction

The Deep Statistical Solvers model encounters several limitations, which are highlighted in Section 5.4. This introductory section explains the key ideas used in the DS-GPS model to address these shortcomings.

One of the first downsides of DSS involves using as inputs coefficients from the linear system (4.3) to infer a solution. While these coefficients are necessary to train the model due to the specific loss function (4.6) used, a solution involves replacing them with some data derived from the original problem. Notably, the edge features (initially, the coefficients from A) are replaced by the distance between the nodes. Moreover, the external node features (initially, the coefficients from B) are replaced by the discretized values of the forcing function f and boundary function g from

(4.1). As a result, once the model is trained, inference only requires information about the mesh and the original problem, eliminating the need for the expensive computation of the elements in the linear system (4.3).

Secondly, the architecture of DSS follows an iterative process that incorporates a fixed number of Message Passing Neural Network (MPNN) layers to achieve convergence. In each layer, these MPNNs have different weights. Moreover, the authors of DSS prove that the number of layers should be proportional to the diameter of the meshes included in the dataset. As a result, the DSS model can only successfully handle meshes with roughly the same number of nodes. Furthermore, the larger the meshes, the greater the number of MPNN layers, resulting in an increasingly larger model. To address this limitation, DS-GPS suggests combining Graph Neural Networks with a Recurrent architecture, enabling the iteration over the same block of MPNNs, thus drastically decreasing the number of weights to be learned.

The architecture of DS-GPS incorporates additional features to improve the generalization capabilities of the DSS model. Notably, it includes an auto-encoding process to bridge the gap between the physical space and the latent space where GNN layers are applied. Notably, the encoder enables the accurate mapping of the original Dirichlet boundary conditions, leading to a node-oriented architecture that intrinsically respects the Dirichlet boundary conditions. Indeed, Dirichlet boundary nodes, whose proper encoding is crucial for propagating information in the latent space, can be preserved throughout the whole process when they are appropriately encoded and decoded. Leveraging a node-oriented architecture also enables extending the model to handle Poisson problems with mixed boundary conditions (4.2), where separate MPNNs can be used to treat the homogeneous Neumann boundary conditions.

DS-GPS aligns with the statistical problem discussed in Section 4.2. By using the same notations and considering a discretized Poisson problem represented as a graph $G = (N,\ A,\ B)$, it becomes feasible to construct a structurally equivalent graph $\bar{G} = (\bar{A},\ f_h,\ g_h)$. Here, $\bar{A}$ represents the adjacency matrix computed from the mesh Ω_h, taking into account the specific directionality of the edges (similar to the process described in Figures 4.1b and 4.1d). The terms f_h and g_h denote the discretized values of the forcing function f and boundary function g from (4.1), evaluated at each node of the graph. To summarize, the idea is now to define a Graph Neural Network (GNN)-based model, parametrized by a vector θ, that predicts from a discretized Poisson problem a solution U in order to solve the following statistical problem:

Given a distribution $\mathcal{D}$ on space $\mathcal{S}$ and a loss function $\mathcal{L}_{res}$, solve:

$$\begin{aligned}\theta^\star &= \underset{\theta}{\text{argmin}} \underset{G\sim\mathcal{D}}{\mathbb{E}} \left[\mathcal{L}_{\text{res}}\left(\text{DS-GPS}_\theta(\bar{G}), G\right)\right] \\ &= \underset{\theta}{\text{argmin}} \underset{G\sim\mathcal{D}}{\mathbb{E}} \left[\mathcal{L}_{\text{res}}\left(\text{DS-GPS}_\theta(\bar{A},\ f_h,\ g_h), G\right)\right]\end{aligned}$$

6.2 . Methodology

This section presents the methodology used in DS-GPS. The first Section 6.2.1 provides a brief introduction to Recurrent Neural Networks, and Section 6.2.2 develops the architecture of DS-GPS. Finally, Section 6.2.3 delves into the specifics of the training procedure.

6.2.1 . Recurrent Neural Network architecture

Recurrent Neural Networks (RNNs) belong to a class of neural networks designed to handle time series data. At each time step t, an RNN takes as inputs a state variable x_t and a latent state h_t, and outputs the next latent state h_{t+1}. RNNs are particularly useful for compressing the history of the network into a latent state. However, such implicit memory suffers from an exponential decay and numerical challenging problems arise, such as unstable (exploding or vanishing) gradients. To address these issues, researchers have developed specialized neural network cell structures that can effectively retain past information in the long term. The most commonly used cells are Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) and Gated Recurrent Unit (GRU) (Cho et al., 2014). Notably, GRU cells offer a simplified version of LSTMs, with fewer weights. GRU cells can be described as follows:

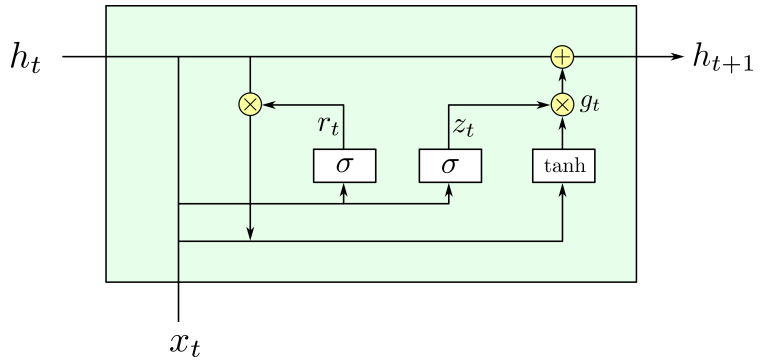


Figure 6.1: Illustration of a GRUMod cell. The cell takes as inputs the current latent state h_t and some input state x_t and performs several operations to produce an update latent state h_{t+1}.

At iteration t, a GRU cell updates the latent state h_t using the series of operations:

$$z_t = \sigma\left(W_z x_t + U_z h_t + b_z\right) \tag{6.1}$$
$$r_t = \sigma\left(W_r x_t + U_r h_t + b_r\right) \tag{6.2}$$
$$g_t = \tanh\left(W_g x_t + U_g\left(r_t \otimes h_t\right) + b_g\right) \tag{6.3}$$
$$h_{t+1} = z_t \otimes h_t + (1 - z_t) \otimes g_t \tag{6.4}$$

where W_z, U_z, W_r, U_r, W_g and U_g are learnable neural networks (usually MLPs) and b_z, b_r and b_g are the corresponding bias vectors. r_t is called a reset gate and decides how much of the past information to forget in g_t. z_t is called an update gate and determines how much of past information is to be passed to the next step. Empirically, both LSTM and GRU seem to perform equivalently, as demonstrated in Chung et al. (2014).

In DS-GPS, an adapted version of the GRU cell is introduced. The final operation of the GRU cell (6.4) updates the next latent state h_{t+1} using a barycenter-like calculation between the previous latent state h_t and an update g_t, based on a learned coefficient z_t. Building upon the approach proposed in DSS, we aim to update the latent state in a ResNet-style, which involves replacing the last update with a residual operation. This novel RNN cell, referred to as GRUMod, is illustrated in Figure 6.1 and is mathematically updated as follows.

At iteration t, a GRUMod cell updates the latent state h_t by:

$$z_t = \sigma\left(W_z x_t + U_z h_t + b_z\right) \tag{6.5}$$
$$r_t = \sigma\left(W_r x_t + U_r h_t + b_r\right) \tag{6.6}$$
$$g_t = \tanh\left(W_g x_t + U_g\left(r_t \otimes h_t\right) + b_g\right) \tag{6.7}$$
$$h_{t+1} = h_t + z_t \otimes g_t \tag{6.8}$$

Additionally, Hajiramezanali et al. (2019) and Ruiz et al. (2020) provide further information regarding the combination of Graph Neural Networks with Recurrent Neural Network architectures.

6.2.2 . Architecture

Figure 6.2 illustrates a global view of DS-GPS architecture. Similar to the DSS approach, the training of the DS-GPS model uses an iterative process to update a latent state $H = (H_i)_{i\in[N]} \in \mathcal{H}$ with $H_i \in \mathbb{R}^d$, $d \geq 1$ for $\bar{k}$ iterations. It is designed with three core components: an *Encoder*, a *Message Passing* block, and a *Decoder*. The "Encoder-Decoder" process facilitates the connection between the physical space, where the solution lives, and the latent space, on which GNN layers are applied. The Message Passing block is responsible for propagating the information within the network by updating the latent state at each iteration through a recurrent process while properly taking into account the boundary conditions by design.

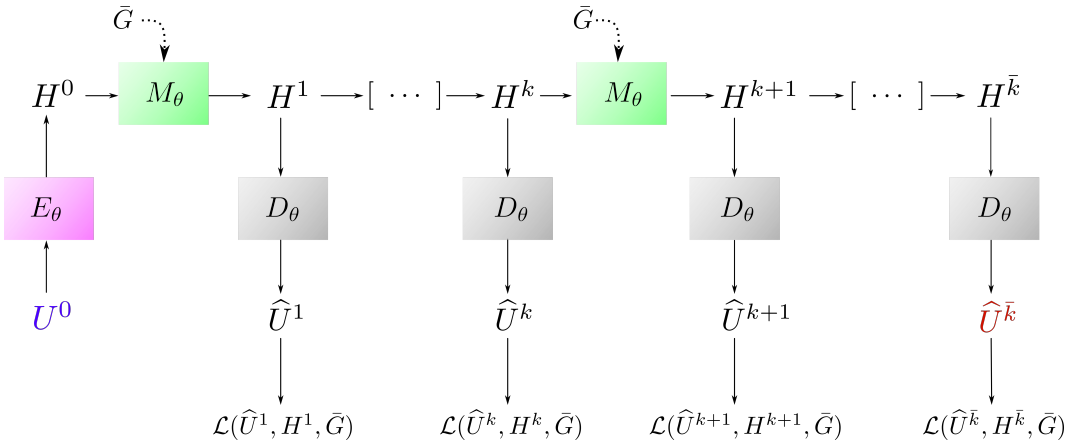


Figure 6.2: Visualization of the DS-GPS architecture: an initial solution U^0, depicted in blue in the figure, is fed into an Encoder (purple box) to give an initial latent state H^0. An iterative process is then used to propagate the information for $\bar{k}$ iterations using the same Message Passing block M_θ (green boxes). At each iteration k, the same Decoder D_θ (grey boxes) translates the latent state H^k into the physical state $\widehat{U}^k$, upon which an intermediate loss is computed. The coloured boxes correspond to trainable functions, and their weights are learned during end-to-end training. The final state $\widehat{U}^{\bar{k}}$, shown in red in the figure, represents the output of the model.

In contrast to the DSS approach, the DS-GPS architecture first introduces an Encoder block to initialize the model. Additionally, DS-GPS is a recurrent model, meaning that it uses the *same* Message Passing block and Decoder at each iteration. Consequently, the DS-GPS model is significantly lighter than the DSS model and offers the flexibility to manually adjust the number of iterations for inferring a solution. Finally, the Message Passing block follows a node-oriented architecture, i.e. a specific architecture can be designed for each node type, allowing for proper treatment of the boundary conditions.

Encoder

The Encoder E_θ (purple box in Figure 6.2) maps an initial solution $U^0 \in \mathcal{U}$ to a $d-$dimensional latent state $H^0 \in \mathcal{H}$, $d > 1$. The provided initial U^0 solution must fulfil the Dirichlet boundary conditions. This trainable function, designed as a multilayer perceptron (MLP), projects the physical space $\mathcal{U}$ to a higher dimensional latent space $\mathcal{H}$ on which the GNN layers will be applied.

Message Passing block

The Message Passing block M_θ (green boxes in Figure 6.2) is responsible for the flow of information within the network by updating, at each iteration k, the current latent state H^k to H^{k+1}. Notably, it uses a specialized approach for each node type, ensuring compatibility with the boundary conditions. To achieve this, *Interior* and *Neumann* nodes are updated through combinations of trainable functions,

effectively capturing the stencils of the discretized Laplace operator. For *Dirichlet* boundary nodes, the corresponding latent variable remains constant, equal to the imposed value.

Interior nodes messages At each iteration k, two separate messages are computed for each node i, corresponding to the outgoing and incoming links, using MLPs $\Phi^{\mathtt{I}}_{\rightarrow,\theta}$ and $\Phi^{\mathtt{I}}_{\leftarrow,\theta}$:

$$\phi^{\mathtt{I},k}_{\rightarrow,i} = \sum_{j\in\mathcal{N}(i)} \Phi^{\mathtt{I}}_{\rightarrow,\theta}\left(H^k_i, H^k_j, d_{ij}, \|d_{ij}\|\right) \tag{6.9}$$

$$\phi^{\mathtt{I},k}_{\leftarrow,i} = \sum_{j\in\mathcal{N}(i)} \Phi^{\mathtt{I}}_{\leftarrow,\theta}\left(H^k_i, H^k_j, d_{ji}, \|d_{ji}\|\right) \tag{6.10}$$

where $j \in \mathcal{N}(i)$ stands for all the nodes j in the one-hop neighbourhood of i, and d_{ij} and $\|d_{ij}\|$ represent the relative position vector and the Euclidean distance. Messages (6.10) and (6.9) are then combined with problem-specific data b_i (further described in Section 6.2.3) and passed through a GRUMod cell to compute the Interior latent variable $\mathbf{z}^{\mathtt{I},k} = (z^{\mathtt{I},k}_i)_{i\in[N]}$:

$$z^{\mathtt{I},k}_i = \text{GRUMod}\left(H^k_i, b_i, \phi^{\mathtt{I},k}_{\rightarrow,i}, \phi^{\mathtt{I},k}_{\leftarrow,i}\right) \tag{6.11}$$

Neumann nodes messages At each iteration k, one message from an incoming link ensures homogeneous Neumann boundary conditions. It is constructed in a similar manner to (6.10), using the MLP $\Phi^{\mathtt{N}}_{\leftarrow,\theta}$:

$$\phi^{\mathtt{N},k}_{\leftarrow,i} = \sum_{j\in\mathcal{N}(i)} \Phi^{\mathtt{N}}_{\leftarrow,\theta}\left(H^k_i, H^k_j, d_{ji}, \|d_{ji}\|\right) \tag{6.12}$$

The Neumann latent variable $\mathbf{z}^{\mathtt{N},k} = (z^{\mathtt{N},k}_i)_{i\in[N]}$ is computed by combining message $\phi^{\mathtt{N},k}_{\leftarrow,i}$ from Eq. (6.12) above with problem-related data b_i and the information on the normal vector n_i, and passing the result through an MLP Ψ_θ as follows:

$$z^{\mathtt{N},k}_i = \Psi_\theta\left(H^k_i, b_i, n_i, \phi^{\mathtt{N},k}_{\leftarrow,i}\right) \tag{6.13}$$

Updated latent state All previous computations from (6.9) to (6.13) can be grouped into a single Message Passing block M_θ. The updated latent state H^{k+1}, designed to preserve Dirichlet boundary values while separating Interior and Neumann messages, is obtained as follows:

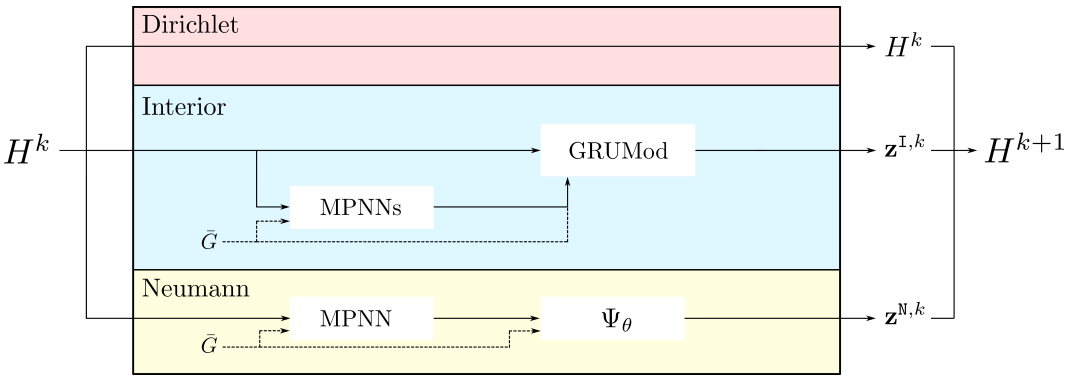


Figure 6.3: Zoom in on the Message Passing block M_θ, responsible for updating the latent state H^k to H^{k+1}, where specific treatments are applied to each node type. For Dirichlet nodes (in red), their values are preserved throughout the iterations. For Interior nodes (in blue), MPNNs are computed in both directions and then fed into a GRUMod cell to produce the Interior latent variable $\mathbf{z}^{I,k}$. Finally, for Neumann nodes (in yellow), an MPNN is computed and then fed into an MLP Ψ_θ, the output of which is the Neumann latent variable $\mathbf{z}^{N,k}$. All three variables are combined to generate the updated latent state H^{k+1}.

$$H^{k+1} = M_\theta(H^k, \bar{G}) = \begin{cases} H^k & \text{if Dirichlet} \\ \mathbf{z}^{\mathrm{I},k} & \text{if Interior} \\ \mathbf{z}^{\mathrm{N},k} & \text{if Neumann} \end{cases} \tag{6.14}$$

Figure 6.3 provides a detailed view of the Message Passing block M_θ, highlighting specific computations for each node type: a blue section for Interior nodes, a red section for Dirichlet nodes, and a yellow section for Neumann nodes.

Decoder

At each iteration k, the Decoder D_θ (grey boxes in Figure 6.2) maps the current latent state $H^k \in \mathcal{H}$ into a meaningful physical solution $U^k \in \mathcal{U}$, upon which the residual loss is computed. The Decoder is designed as an MLP, and additional losses in the training process ensure that the Decoder performs the inverse operation of the Encoder.

6.2.3 . Training materials

This section aims to provide additional training materials, describing the training loss used, explaining the vector format considered in Section 6.2.2 and the data normalization process.

Training loss

The entire DS-GPS model is trained by minimizing the following cost function:

$$\text{Training Loss} = \sum_{k=1}^{\bar{k}} \mathcal{L}(\widehat{U}^k,\ H^k,\ \bar{G}) \tag{6.15}$$

where $\mathcal{L}(\widehat{U}^k,\ H^k,\ \bar{G})$ is defined such that:

$$\mathcal{L}(\widehat{U}^k,\ H^k,\ \bar{G}) = \gamma^{\bar{k}-k} \times \mathcal{L}_{\text{res}}(\widehat{U}^k, \bar{G}) \tag{6.16}$$

$$+ \lambda \times \text{MSE}\left(\widehat{U}^k - U^{\text{ex}}\right) \tag{6.17}$$

$$+ \text{MSE}\left(D_\theta\left(E_\theta\left(\widehat{U}^k\right)\right) - \widehat{U}^k\right) \tag{6.18}$$

Equation (6.16) represents the residual loss, as described in (4.6), which is weighted by a discount factor γ. When considering the incorporation of Neumann boundary conditions, achieving satisfactory results with the residual loss alone becomes more challenging due to increased problem conditioning. To overcome this issue, we introduce a slightly weighted additional supervised loss, as defined in Equation (6.17). In this loss, U^{ex} represents the LU ground truth, and λ serves as a small weighting factor (see 6.3.2 for more information). Finally, Equation (6.18) is designed to facilitate the learning of the autoencoding mechanism.

Data information

The architecture of the model described in 6.2.2 relies on several inputs whose format needs to be detailed. Equations (6.11) and (6.13) require as input some problem-related data, encoded into a vector b_i such that:

$$b_i = \begin{cases} \begin{bmatrix} f_i & 0 & 0 \end{bmatrix} & \text{if } i \text{ is Interior} \\ \begin{bmatrix} 0 & g_i & 0 \end{bmatrix} & \text{if } i \text{ is Dirichlet} \\ \begin{bmatrix} 0 & 0 & f_i \end{bmatrix} & \text{if } i \text{ is Neumann} \end{cases} \tag{6.19}$$

where f_i and g_i are the discretized values of the force function f and the Dirichlet boundary function g from the Poisson problem (4.2).

Data normalization

When designing a Machine Learning model, normalizing the input features is essential to enhance the performance of the model during the training phase. To achieve this, the distances d_{ij} in equations (6.10), (6.9), (6.12), the problem-related vector b_i in equations (6.11) and (6.13), as well as the normal vector n_i in equation (6.13), are all normalized. This normalization is performed by subtracting the mean and dividing by the standard deviation, which are computed based on the entire training

dataset. When dealing with problem-related data, the normalization is conducted column-wise, taking into account the statistics of the entire training dataset.

6.3 . Experiments & Results

The results are presented in the next two sections. The first Section 6.3.1 provides results for solving Poisson problems with Dirichlet boundary conditions only, allowing for a fair comparison with the Deep Statistical Solvers approach. The second Section 6.3.2 showcases the efficiency of the model when extended to solve Poisson problems with mixed boundary conditions. Both sections also analyze the effects of some hyperparameters regarding the training of the model, revealing interesting insights.

Throughout these experiments, we consider the solution of the discretized Poisson problem given by the classical LU decomposition method as the “ground truth". The reported metrics are the Residual Loss (4.6) and the Mean Squared Error (MSE) between the output of the model and the “ground-truth” LU solution.

6.3.1 . Poisson problems with Dirichlet boundary conditions

In this first section, we analyze the efficiency of DS-GPS in solving Poisson problems with Dirichlet boundary conditions, enabling a direct comparison with the Deep Statistical Solvers (DSS) approach. Additionally, we provide further analysis of the capabilities of the model, revealing interesting properties when a proper configuration is selected.

Experimental setup

The dataset used in this experiment is the same as the one used for training the Deep Statistical Solvers model (see Section 4.3): it consists of $6000/2000/2000$ training/validation/test samples of Poisson problems with Dirichlet boundary conditions. All meshes have approximately 500 nodes, corresponding to a mean mesh diameter of 30. Provided results regarding the DSS model are those presented in the previous Chapter 5.

DS-GPS is implemented in PyTorch using the PyTorch Geometric library (Fey and Lenssen, 2019). The DS-GPS model used for this experiment follows the architectural specifications outlined in Section 6.2.2, without of course the functionalities related to Neumann nodes. The dimension d of the latent space $\mathcal{H}$ is set to 10. Each neural network block in the architecture has one hidden layer of dimension 10 with a ReLU activation function, except for those involved in the GRUMod cell, whose activation functions are predetermined by definition. All model weights are initialized using Xavier initialization (Glorot and Bengio, 2010). For training, the initial solution U^0 is set to 0 everywhere, except at the Dirichlet boundary nodes, which are as-

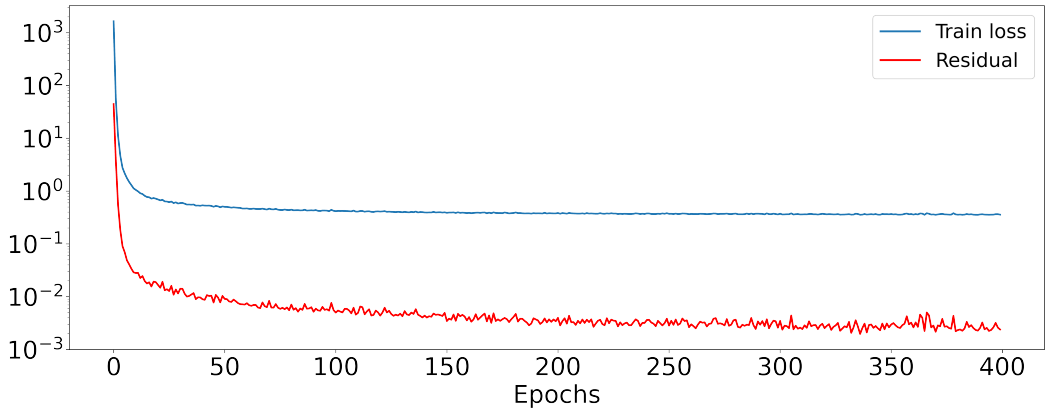


Figure 6.4: Evolution of the Training (blue) and Residual (red) losses during the 400 epochs.

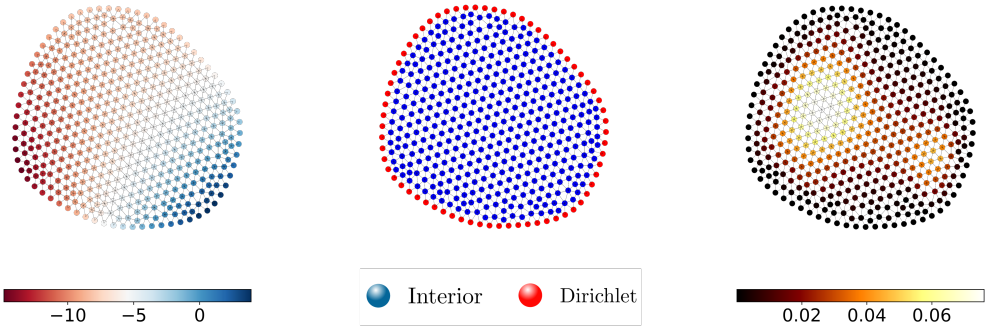
signed their exact discrete values. During training, the model is optimized using the loss function (6.15) with $\lambda = 0$ (no supervised loss) and $\gamma = 0.9$. The number of iterations is set to the average diameter of the meshes, which is $\bar{k} = 30$. Training is conducted for 400 epochs using P100 GPUs and the Adam optimizer with its default PyTorch hyperparameters, except for the initial learning rate, which is set to 0.01. Finally, gradient clipping is used to prevent exploding gradient issues, with a threshold set to 0.01. Figure 6.4 displays the evolution of the training and residual losses during the 400 epochs, showcasing that the training has been effective. Most of the hyperparameters of DS-GPS have been selected identically to the DSS approach (latent space size, γ). The remaining hyperparameters were chosen after numerous trials and have proven to be robust. For optimal performance, it is possible to fine-tune these parameters using tools such as Optuna (Akiba et al., 2019), for example.

Results

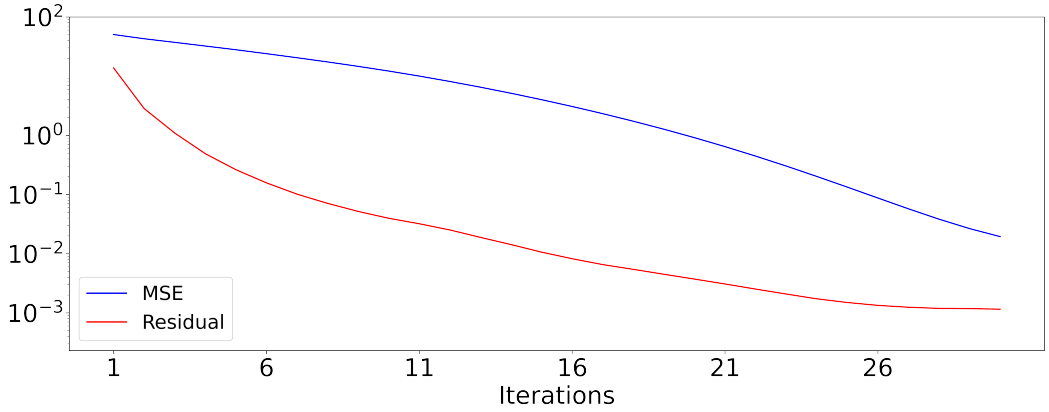
Figure 6.5 displays the resolution of a Poisson problem with 463 nodes, extracted from the test set, using DS-GPS. This figure validates the effectiveness of this new approach, as it successfully solves the problem with accuracy, achieving a Residual loss of $1.1\text{e}{-3}$ and an MSE w/LU of $1.5\text{e}{-2}$ at the last iteration. Furthermore, it highlights how DS-GPS propagates information through the graph by iterating on the same neural network block. This is depicted by the red and blue curves, representing the evolution of the Residual and MSE w/LU across the 30 iterations.

Comparison with DSS

Table 6.1 presents the Residual and MSE w/LU averaged over the entire test set for DS-GPS and DSS. These results demonstrate that the DSS approach still outperforms DS-GPS but at the cost of a significantly larger model. Specifically, DSS has



(a) Data-driven solution *(b)* Node types *(c)* Error map



(d) Residual (red) and MSE w/LU (blue) losses

Figure 6.5: Illustration of the resolution of a Poisson problem extracted from the test set using DS-GPS. Figure 6.5a shows the data-driven solution obtained at the last iteration, while Figure 6.5c displays the map of squared errors between the data-driven solution and the LU solution. Figure 6.5b illustrates the different types of nodes. At the bottom, Figure 6.5d depicts the evolution of the Residual (in red) and MSE w/LU (in blue) across the 30 iterations of the model.

36930 weights, while DS-GPS only has 1961 parameters. These results were to be expected, but they illustrate the possibility of building a lighter model that does not degrade the performance of a state-of-the-art model. Besides, the DSS model is lim-

Metrics	Residuals (10^{-3})	MSE w/LU	Nb of weights
DSS	0.225 ± 0.2	0.03 ± 0.02	36930
DS-GPS	1.3 ± 0.1	0.063 ± 0.02	1961

Table 6.1: DS-GPS and DSS results averaged over the whole test set.

ited to a fixed number of 30 iterations for both training and inference. To address larger meshes would necessitate the addition of extra layers, leading to an increasing size of the model. On the other hand, DS-GPS is a recurrent model, demonstrating that information propagation in this context can be achieved through iterative processing within the same layer of Message Passing Neural Networks. Besides, even though DS-GPS is trained with a predefined number of iterations, its recurrent nature allows for the possibility of manually extending the iteration count, a characteristic explored in the subsequent paragraph.

Convergence towards an equilibrium point

In this experiment, we analyze the convergence of the model when training it with a larger number of iterations. In particular, Figure 6.6 illustrates the convergence of the Residual loss during the 400 epochs for different configurations of the DS-GPS model. What we are investigating here is the sensitivity of the model to the hyperparameter γ, which plays a role in the cost function (6.15). This hyperparameter is a weight factor which decays exponentially with $\bar{k} - k$, where k represents the current iteration. As the total number of iterations $\bar{k}$ increases, the influence of the early iterations becomes negligible. Consequently, the model no longer propagates the early iterations, making it unable to converge towards the solution of the problem, as evidenced by the green curve, which illustrates the training convergence of a model for $\gamma = 0.9$ and $\bar{k} = 70$ iterations. One idea to address this issue would be to make γ closer to 1, but the problem persists with an increase in iterations. Another possible configuration would be to set γ to 1. In this context, all iterations in the cost function (6.15) would have an equivalent weight. This leads to a slight degradation in the convergence of the model, as shown by the blue curve trained for 30 iterations with $\gamma = 0.9$, which exhibits slightly better convergence than the orange curve trained for 30 iterations with $\gamma = 1$. However, when training for 70

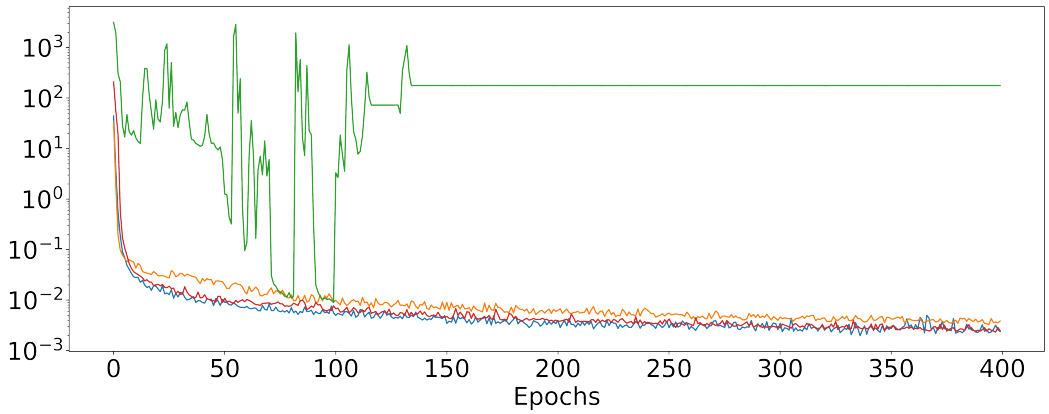


Figure 6.6: Evolution of the residual loss during training for different configurations of DS-GPS: 30 iterations, $\gamma = 0.9$; 30 iterations, $\gamma = 1$; 70 iterations, $\gamma = 0.9$; 70 iterations, $\gamma = 1$.

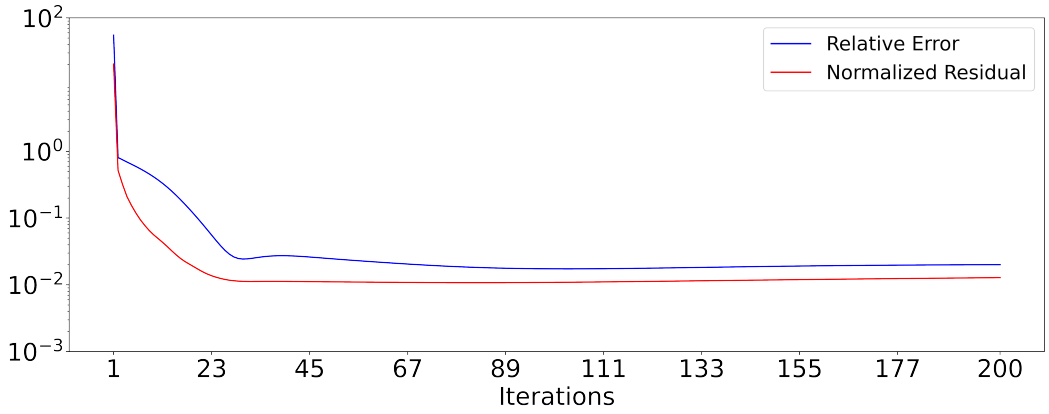


Figure 6.7: Resolution of a Poisson problem extracted from the test set using DS-GPS trained with $\gamma = 1$ and $\bar{k} = 70$ iterations. The solution is inferred by leveraging the recurrent nature of the model and during 200 iterations. The figure illustrates the evolution of the Normalized Residual (in red) and Relative Error w/LU (in blue) across the 200 iterations.

iterations with $\gamma = 1$, we observe that the model converges very well, matching the original convergence, as shown by the red curve.

Figure 6.7 illustrates the resolution of a Poisson problem from the test set using the DS-GPS model, which was trained for 70 iterations with $\gamma = 1$ (i.e., the red curve in Figure 6.6). In this example, the Poisson problem is inferred with a total of $\bar{k} = 200$ iterations, even though it was trained with only 70 iterations. We can observe that under this configuration, the solution converges toward an equilibrium point. This convergence is indicated by the red and blue curves, representing the evolution of the normalized residual $\frac{||A\hat{U}^k - B||}{||B||}$ and the relative error $\frac{||\hat{U}^k - U_{\text{ex}}||}{||U_{\text{ex}}||}$, respectively, during the 200 iterations. They converge to a normalized residual of 1.28e-2 and a relative error of 2.0e-2 at the last iteration. Furthermore, we can note a significant difference in the relative error for small variations in the normalized residual (between iterations 1 and 45), aligning with well-established theoretical results regarding error and residual analysis (refer to Chapter 1 in (Saad, 2003)). However, it is important to note that convergence is not guaranteed by the method. Indeed, when evaluating the model using 200 iterations instead of the 70 corresponding to its initial training configuration, we can observe a degradation in the results. This is illustrated in Table 6.2, which presents the Residual and MSE w/LU results averaged over the entire test set, using both the initial configuration with 70 iterations and the one with 200 iterations. Nevertheless, this behaviour of convergence toward a fixed point motivated the model presented in the following chapter.

6.3.2 . Poisson problems with mixed boundary conditions

This section aims to assess the performance of DS-GPS when extended to handle Poisson problems with mixed boundary conditions. Note that comparison with the

$\bar{k}$	Residuals (10^{-3})	MSE w/LU
70	1.62 ± 0.1	0.035 ± 0.01
200	3.3 ± 0.4	0.25 ± 0.02

Table 6.2: Results of DS-GPS trained with $\gamma = 1$ and 70 iterations averaged over the whole test set. The table compares the results between the original configuration with 70 iterations and an alternative configuration that infers solutions using 200 iterations by leveraging the recurrent nature of the model.

original DSS is not possible any more.

Experimental setup

In this experiment, the dataset consists of 6000 training samples, 2000 validation samples, and 2000 test samples of Poisson problems with mixed boundary conditions. These problems were generated following the process described in Section 4.3, specifically detailed in Figure 4.4b. Each mesh in this dataset has approximately 500 nodes, with the node count ranging from a minimum of 407 to a maximum of 542. This range corresponds to an average diameter of approximately 30, similar to previous configurations.

The DS-GPS model used in this experiment adheres to the architectural specifications described in Section 6.2.2. The dimension d of the latent space $\mathcal{H}$ is set to 10. In the model architecture, each neural network block has a single hidden layer of dimension 10 followed by a ReLU activation function, except for those involved in the GRUMod cell, whose activation functions are predefined by design. All model parameters are initialized using Xavier initialization (Glorot and Bengio, 2010). For the training process, the initial solution U^0 is set to 0 everywhere, except at the Dirichlet boundary nodes, where their values are assigned based on the exact discrete values. During training, the optimization is done using the loss function (6.15) with $\gamma = 1$, in line with the results of the previous section. Several configurations are tested, especially regarding the setup of the λ coefficient within the training loss (6.15). This coefficient determines the extent of supervised learning integrated into the model. The total number of iterations $\bar{k}$ for training will also vary, and the specific values will be provided when appropriate. Training is done for a total of 400 epochs, using P100 GPUs and the Adam optimizer with its default PyTorch hyperparameters. The initial learning rate is set to 0.001, and gradient clipping with a threshold of 0.01 is employed to avoid exploding gradients.

$\bar{k}$	λ	Residuals (10^{-3})	MSE w/LU
30	0	2.63 ± 0.3	1.66 ± 0.4
30	0.001	2.91 ± 0.2	0.63 ± 0.2
50	0	1.91 ± 0.1	0.65 ± 0.1
50	0.001	2.2 ± 0.1	0.37 ± 0.1

Table 6.3: Results averages over the whole test set for different training runs of DS-GPS with various configurations. The tests involve variations of the total number of iterations $\bar{k}$ and the amount of supervised learning in the loss function, quantified by the parameter λ.

Results

Table 6.3 presents the results of Residual and MSE w/LU averaged over the entire test set for four training runs of DS-GPS with various configurations. Our focus here is the sensitivity with respect to the total training iterations $\bar{k}$, considering either 30 or 50 iterations, and the amount of supervised learning, quantified by the coefficient λ, considering either $\lambda = 0$ or $\lambda = 0.001$. We observe that in terms of Residual, the model consistently and effectively converges regardless of the configuration. However, a slight improvement is noticeable when extending the number of iterations from 30 (previously considered the default value due to the average diameter of the meshes) to 50. The most significant differences are observed when examining the MSE w/LU. For the model with $\bar{k} = 30$ and $\lambda = 0$ (i.e., the first row in the table), the MSE error is high, equal to 1.66. This can be improved by increasing the λ parameter, as shown in the second row of the table, which presents an MSE w/LU of 6.3×10^{-1}. Similar results can be observed by increasing the number of iterations from 30 to 50, this time without supervised learning, as shown in the third row of the table. Finally, the best results are observed in the last row when the number of iterations is 50, and $\lambda = 0.001$. We can propose two possible explanations here. The first explanation, of a geometric nature, concerns the structure of the graph. With Neumann boundary conditions, the induced graphs are directed only in certain parts of the boundary,

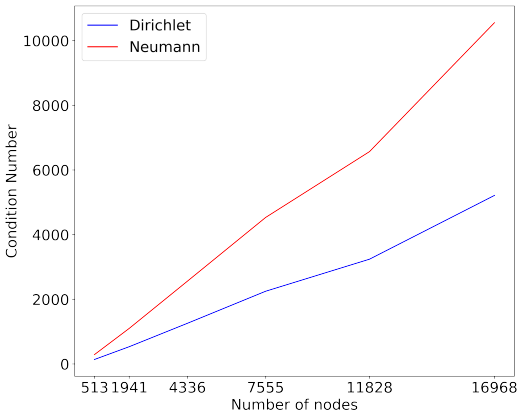
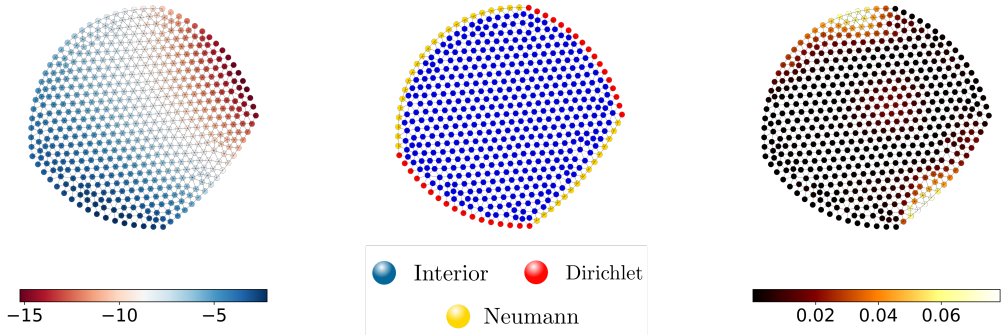
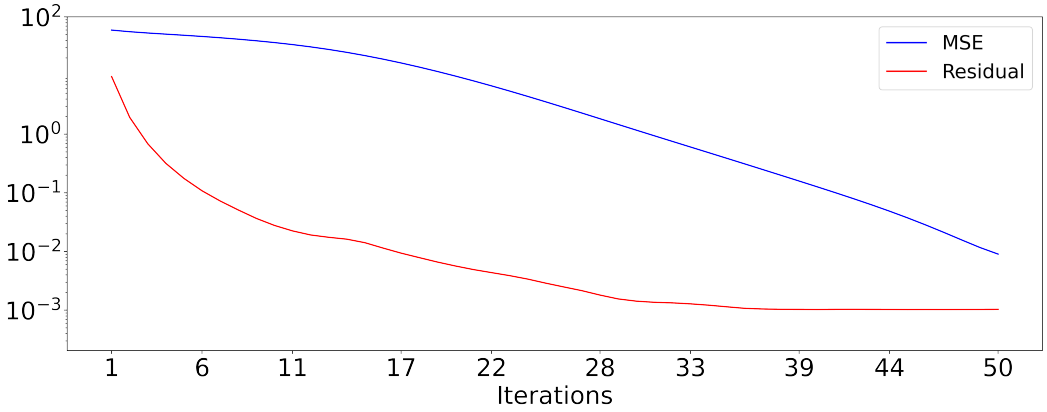


Figure 6.8: Condition number of the stiffness matrix A with respect to the number of nodes in a mesh for Poisson problems with Dirichlet (blue) and Mixed (red) boundary conditions.



(a) *Data-driven solution* (b) *Node types* (c) *Error map*



(d) *Residual (red) and MSE w/LU (blue) losses*

Figure 6.9: Illustration of the resolution of a Poisson problem extracted from the test set using DS-GPS. Figure 6.9a shows the data-driven solution obtained at the last iteration, while Figure 6.9c displays the map of squared errors between the data-driven solution and the LU solution. Figure 6.9b illustrates the different types of nodes. At the bottom, Figure 6.9d depicts the evolution of the Residual (in red) and MSE w/LU (in blue) across the 50 iterations of the model.

specifically at the Dirichlet nodes. The Neumann nodes increase the difficulty of information propagation since the edges are bidirectional at the Neumann nodes. In this regard, a model trained with more iterations will be better at propagating information and achieving better results. The second, possible explanation is more theoretical, involving the relationship between the exact error (quantified by MSE w/LU) and the Residual. Indeed, the exact error is bounded by the Residual multiplied by the condition number of the stiffness matrix A (i.e., the ratio between the largest and smallest eigenvalues of A - refer to Chapter 1 of Saad (2003) for additional detail). In the context of solving Poisson problems, the addition of homogeneous Neumann boundary conditions increases the conditioning of the matrix. This

result is illustrated in Figure 6.8, where Poisson problems with Dirichlet and Mixed boundary conditions are solved on a square mesh with a growing number of nodes, ranging from 513 to 16978. The figure demonstrates two points: i) it confirms that adding homogeneous Neumann boundary conditions increases the conditioning, and ii) the conditioning of the problem grows with the number of nodes in the mesh. Hence there is no guarantee here that our model, primarily trained by minimizing the Residual, will result in a small MSE w/LU. One way to improve these results is to introduce slight supervised learning in the cost function, as demonstrated by the results of Table 6.3.

Figure 6.9 displays the resolution of a Poisson problem with 517 nodes extracted from the test set, using DS-GPS trained with $\bar{k} = 50$ iterations and $\lambda = 0.001$. This figure validates the effectiveness of DS-GPS for solving Poisson problems with mixed boundary conditions, as it accurately solves the problem, achieving a Residual loss of 1.03×10^{-3} and an MSE w/LU of 7.6×10^{-2} at the last iteration. The red and blue curves represent the evolution of the Residual and MSE w/LU across the 50 iterations, showing how DS-GPS effectively propagates information through the graph. The highest errors are primarily located at the Neumann nodes, as depicted in Figure 6.9c, confirming the increased difficulty of the problem when Neumann boundary conditions are considered. This increased difficulty is alleviated when introducing slight supervised learning through λ and additional iterations.

6.4 . Conclusion and Limitations

This chapter introduces DS-GPS, a GNN-based model which iteratively solves Poisson problems with Dirichlet and mixed boundary conditions. DS-GPS is a recurrent model, successfully propagating information through the graph by iterating over the same neural block. This is in contrast to the DSS approach, which uses a fixed number of iterations with different blocks at each iteration. As a result, DS-GPS is a much lighter model, and the number of iterations can be extended without modifying the architecture due to its recurrent nature. However, this number of iterations remains fixed by the user, preventing the model from adapting to and accommodating meshes of varying sizes. The model also manages to solve Poisson problems with mixed boundary conditions, although the increased complexity of the problem requires the addition of supervised learning in the cost function. Future work could also consider using software to tune DS-GPS hyperparameters to achieve the best possible results. Subsequently, our goal is to make the model capable of dynamically adapting to different mesh sizes, i.e., automatically adjusting the number of GNN layers required for convergence. This is presented in the following chapter, which builds upon a property introduced in the analysis of DS-GPS: with a sufficiently large number of training iterations, the model tends to converge to a fixed point.

7 - Ψ-GNN : An Implicit GNN Solver

Sommaire

This chapter introduces Ψ-GNN, an original Graph Neural Network-based approach to solving Poisson problems such as (4.1) or (4.2). The Ψ-GNN method is built by drawing inspiration from the results achieved with the DS-GPS model presented in previous Chapter 6. The primary objective of this new approach is to enhance the generalization abilities of DS-GPS by constructing a model capable of dynamically adjusting the number of Message-Passing iterations to reach a solution.

To begin, Section 7.1 explains the motivations behind this novel approach by discussing the limitations and strengths of DS-GPS and subsequently presenting the core concept of Ψ-GNN. Then, Section 7.2 details the methodology used to construct and train Ψ-GNN: Its architecture in Section 7.2.1; How to achieve model stability in Section 7.2.2; And the training materials needed, in Section 7.2.3. Furthermore, Section 7.3 provides a theoretical analysis, proving that there exists a parametrization of Ψ-GNN that yields an optimal solution for the considered task, which showcases its consistency. Finally, Section 7.4 investigates the performance of Ψ-GNN, and Section 7.5 concludes this chapter while opening up prospects for future improvements.

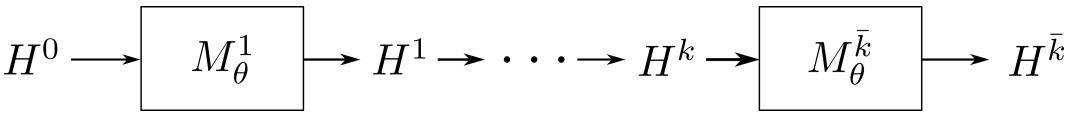
7.1 . Introduction

In previous Chapter 6, we introduced DS-GPS, a new GNN-based model designed to enhance the generalization capabilities of the state-of-the-art Deep Statistical Solvers approach, presented in Chapter 5. DS-GPS incorporates several new concepts into its architecture, which can be summarized as follows:

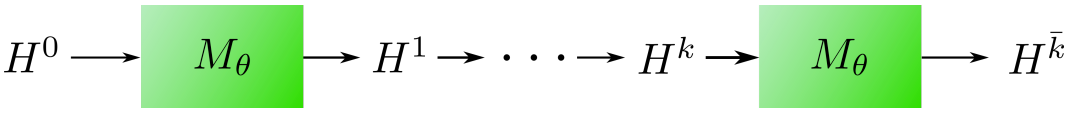
1. Use of data from the original Poisson problem (4.2) instead of data from the discretized equation. This approach avoids the computationally costly construction of the linear system (4.3), the "assembly" step of the FEM method to infer a solution.
2. Development of a Recurrent architecture. The same Message-Passing block and Decoder are used at each iteration, resulting in a significantly lighter model compared to the DSS approach.
3. Modification of the Message-Passing block into a node-oriented approach, inherently respecting the boundary conditions.
4. Introduction of an Encoder to properly map the Dirichlet boundary conditions into the latent space. Once the Dirichlet values are correctly mapped, they can both be preserved and decoded with accuracy until the last iteration, as well as be used consistently to propagate messages within the network.

Results presented in Section 6.3 have demonstrated that these new concepts indeed enhance some generalization capabilities of the original Deep Statistical Solvers approach. For instance, DS-GPS can handle boundary conditions more accurately and is a lighter model that can infer solutions without explicitly computing the linear system from the FEM. However, DS-GPS still faces a significant limitation. As proved in the original DSS paper Donon et al. (2020), the number of Message-Passing steps required to achieve convergence must increase proportionally to the diameter of the considered meshes. Consequently, both DSS and DS-GPS models are trained with a fixed number of iterations $\bar{k}$, set to the average diameter of the meshes in the dataset. Although DS-GPS is of recurrent type, this property can only be leveraged for inferring a solution. During training, the number of iterations must be manually fixed. Nevertheless, an interesting observation arises from the DS-GPS model and the results depicted in Figure 6.7. In this figure, DS-GPS is trained with a weighted factor set to one (i.e. the state obtained at each iteration has the same weight in the final training loss), and the fixed number of iterations for training is larger than the minimum required. In this context, one can notice that the solution tends towards an equilibrium point. More importantly, when the number of iterations increases, the solution remains stable around this equilibrium point.

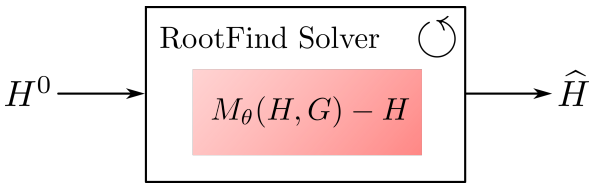
Building upon these experimental results, we propose to enhance the DS-GPS architecture by incorporating a black-box root-finding solver to directly find the equi-



(a) The DSS approach: $H^{k+1} = M_\theta^{k+1}(H^k, G)$



(b) The DS-GPS approach: $H^{k+1} = M_\theta(H^k, G)$



(c) The Ψ-GNN approach: $\widehat{H} = \mathit{RootFind}(M_\theta(H, G) - H)$

Figure 7.1: Comparison of the three considered architectures regarding message propagation. Figure 7.1a illustrates the Deep Statistical Solvers approach, characterized by stacking a fixed number of different GNN blocks. Figure 7.1b depicts the DS-GPS approach, in which the same GNN block is iteratively applied for a fixed number of iterations. Figure 7.1c displays the Ψ-GNN approach, where the final latent state is computed by solving the fixed point of a GNN function. In this approach, iterations occur implicitly within the RootFind solver, and the iteration count is solely determined by the precision set for the solver.

librium point of the model. To implement this, we leverage the Implicit Layer theory (Bai et al., 2019) to model an "infinitely" deep network, thereby eliminating the need for empirical tuning of the number of Message-Passing steps required to achieve convergence. In this approach, the iterations (i.e. the flow of information) are performed implicitly until convergence within the root-finding solver. This method eliminates the need for a fixed number of iterations, as the required number of Message-Passing steps is now solely determined by the precision imposed on the solver, resulting in a more adaptable and flexible approach. To illustrate this further, Figure 7.1 provides a summary of the different approaches: Figure 7.1a depicts the Deep Statistical Solvers method, which employs a fixed number of different Message-Passing steps; Figure 7.1b illustrates the DS-GPS approach, which iterates in a recurrent manner on the same Message-Passing step for a fixed number of iterations; and Figure 7.1c showcases the Ψ-GNN approach, which uses a black-box

	MPNN propagation	Physics-informed	Various shapes	Various sizes	Boundary conditions	Initial solutions	Convergence guarantees
DSS	Fixed	✓	✓	✗	✗	✗	✗
DS-GPS	Recurrent	✓	✓	✗	✓	✗	✗
Ψ-GNN	Implicit	✓	✓	✓	✓	✓	✓

Table 7.1: Comparison between DSS, DS-GPS and Ψ-GNN regarding several features. See text for details.

root-finding solver to automatically (implicitly) determine the number of Message-Passing steps required for convergence.

Table 7.1 presents an operational comparison among the three proposed methods: DSS, DS-GPS, and Ψ-GNN. The comparison highlights several essential features. In the table, *MPNN propagation* denotes how the models propagate information throughout the graphs, while *Physics-informed* indicates whether the models are trained by minimizing the discretized residual equation or not. *Various shapes* refers to the ability of the models to handle meshes of varying shapes, and *Various sizes* indicates whether the architecture of the models can adjust their number of Message-Passing steps, thereby extending their capacities to meshes with varying numbers of nodes. *Boundary conditions* specifies whether the architecture explicitly considers boundary conditions, and *Initial solutions* refers to the capability of the models to adjust their number of steps to achieve convergence with respect to any provided initial solution. Finally, *Convergence guarantees* indicates whether the models provide assurance of convergence toward the solution or not.

In addition to its ability to automatically adjust its number of Message-Passing steps, Ψ-GNN has several new features, such as the capability to adapt to any initially provided solutions. This capability is made possible through an enhanced training process of the autoencoding mechanism, which is further described in Section 7.2.3. Additionally, Ψ-GNN introduces convergence guarantees by incorporating an additional loss term aimed at enforcing the contractive nature of the GNN function M_θ, a feature detailed in Section 7.2.2. Finally, Ψ-GNN aligns with the statistical problem discussed in Section 4.2. Following the same notations as described in the last paragraph of Section 6.1, the statistical problem reads as:

Given a distribution $\mathcal{D}$ on space $\mathcal{S}$ and a loss function $\mathcal{L}_{res}$, solve:

$$\begin{aligned}\theta^\star &= \underset{\theta}{\text{argmin}} \underset{G\sim\mathcal{D}}{\mathbb{E}} \left[\mathcal{L}_{\text{res}}\left(\Psi\text{-GNN}_\theta(\bar{G}), G\right)\right] \\ &= \underset{\theta}{\text{argmin}} \underset{G\sim\mathcal{D}}{\mathbb{E}} \left[\mathcal{L}_{\text{res}}\left(\Psi\text{-GNN}_\theta(\bar{A},\ f_h,\ g_h), G\right)\right]\end{aligned}$$

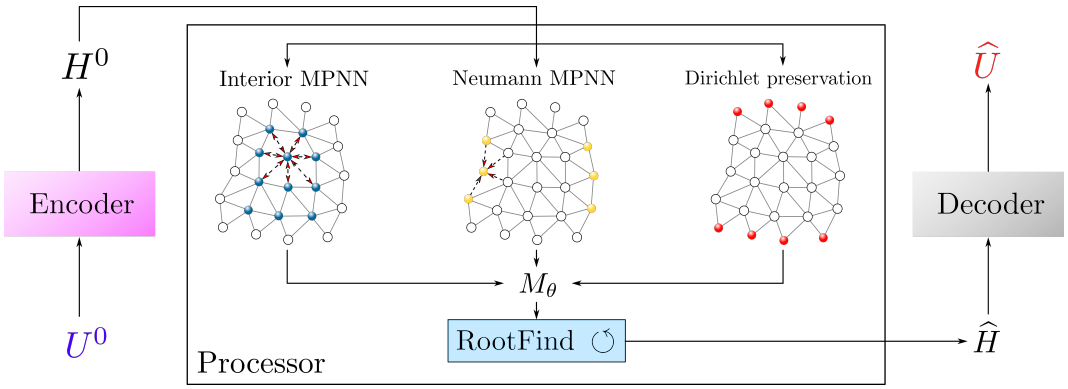


Figure 7.2: Diagram of Ψ-GNN: The model uses an Encode-Process-Decode architecture. The Encoder (purple box) maps an initial solution U^0 (in blue) to some latent representation H^0. The processor outputs a final latent state $\widehat{H}$ by considering a different treatment for each node type. Dirichlet nodes are preserved during the process, and specific MPNN (red arrows) for Interior and Neumann nodes are computed to build a GNN function M_θ. A black-box "root-finding" solver automatically propagates the information through the graph by finding the fixed point $\widehat{H}$ of M_θ, starting from the initial guess H^0. The Decoder (grey box) maps $\widehat{H}$ back to the physical space to get the final solution $\widehat{U}$ (in red).

7.2 . Methodology

7.2.1 . Architecture

Figure 7.2 gives a global view of Ψ-GNN, a Graph Neural Network model with three main components: an *Encoder*, a *Processor*, and a *Decoder*. The "Encoder-Decoder" mechanism facilitates the connection between the physical space, where the solution lives, and the latent space, where the GNN layers are applied. It is the same as for DS-GPS (see Figure 6.2). The Processor is the core component of the model, responsible for propagating the information in the graph. It is specifically designed with two key features: i) it automatically controls the number of Message-Passing steps required for convergence, and ii) it properly takes into account the boundary conditions by design.

Encoder & Decoder

The Encoder E_θ (purple box in Figure 7.2) and the Decoder D_θ (grey box in Figure 7.2) are defined in the exact same way as in the DS-GPS model (refer to Section 6.2.2). Both are designed as multilayer perceptrons (MLPs) and aim to bridge the gap between the physical space $\mathcal{U}$ and a higher-dimensional latent space $\mathcal{H}$ upon which the GNN layers will be applied.

Processor

The Processor uses a specialized approach for each node type to ensure consistency with the boundary conditions. To propagate the information, the processor constructs a GNN-based function M_θ that updates both the *Interior* and *Neumann* nodes, effectively capturing the distinct stencils of the discretized Laplace operator. For *Dirichlet* boundary nodes, the corresponding latent variable is kept constant, equal to the imposed value.

Interior nodes messages Two separate messages are computed for each node, corresponding to the outgoing and incoming links, using MLPs $\Phi^{\mathtt{I}}_{\rightarrow,\theta}$ and $\Phi^{\mathtt{I}}_{\leftarrow,\theta}$ such that:

$$\phi^{\mathtt{I}}_{\rightarrow,i} = \sum_{j\in\mathcal{N}(i)} \Phi^{\mathtt{I}}_{\rightarrow,\theta}\left(H_i, H_j, d_{ij}, \|d_{ij}\|\right) \tag{7.1}$$

$$\phi^{\mathtt{I}}_{\leftarrow,i} = \sum_{j\in\mathcal{N}(i)} \Phi^{\mathtt{I}}_{\leftarrow,\theta}\left(H_i, H_j, d_{ji}, \|d_{ji}\|\right) \tag{7.2}$$

where $j \in \mathcal{N}(i)$ stands for all the nodes j in the one-hop neighbourhood of i, and d_{ij} and $\|d_{ij}\|$ represent the relative position vector and the Euclidean distance. The updated Interior latent variable $\mathbf{z}^{\mathtt{I}} \coloneqq (z^{\mathtt{I}}_i)_{i\in[N]}$ is computed in a Res-Net fashion of the form:

$$z^{\mathtt{I}}_i = H_i + \Lambda_{i,\theta}\left(H_i, b_i, \phi^{\mathtt{I}}_{\rightarrow,i}, \phi^{\mathtt{I}}_{\leftarrow,i}\right) \tag{7.3}$$

To compute $\Lambda_{i,\theta}$, two MLPs are used, Ψ^1_θ and Ψ^2_θ, which both take the same inputs. These inputs include the current latent state H_i, computed MPNNs (7.2) and (7.1), and problem-specific data b_i. The MLPs compute α_i and ζ_i, respectively. The final $\Lambda_{i,\theta}$ is obtained by multiplying α_i and ζ_i as follows:

$$\alpha_{i,\theta} = \Psi^1_\theta(H_i, b_i, \phi^{\mathtt{I}}_{\rightarrow,i}, \phi^{\mathtt{I}}_{\leftarrow,i}) \tag{7.4}$$

$$\zeta_{i,\theta} = \Psi^2_\theta(H_i, b_i, \phi^{\mathtt{I}}_{\rightarrow,i}, \phi^{\mathtt{I}}_{\leftarrow,i}) \tag{7.5}$$

$$\Lambda_{i,\theta} = \alpha_{i,\theta} \times \zeta_{i,\theta} \tag{7.6}$$

Although Ψ^1_θ and Ψ^2_θ share the same architecture with respect to the number of layers and neurons, Ψ^1_θ uses a Sigmoid activation function, whereas Ψ^2_θ uses a ReLU activation function. As a result, α_i is restricted to $(0, 1)$, and can be interpreted as an attention layer, prioritizing the most crucial steps to enhance the performance of our model.

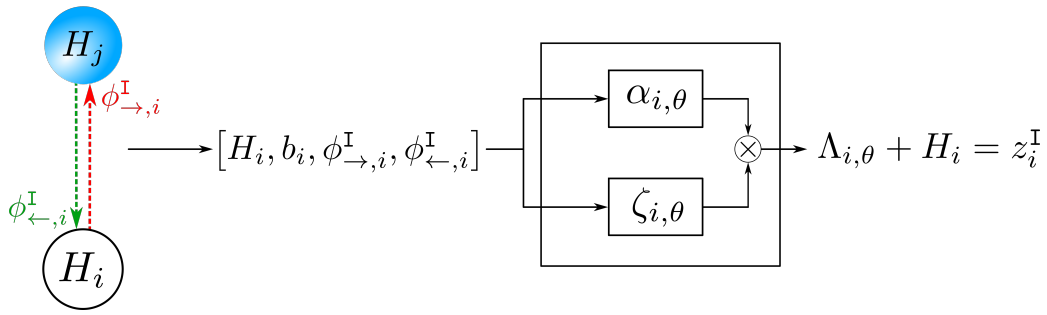


Figure 7.3: Process of updating the Interior latent variable z_i^I: Firstly, two MPNNs, $\Phi^I_{\rightarrow,\theta}$ (in red) and $\Phi^I_{\leftarrow,\theta}$ (in green), are computed to account for the bi-directionality of the edges. These computed messages are then concatenated with problem-specific data b_i and the actual latent state H_i and passed through two trainable functions $\alpha_{i,\theta}$ and $\zeta_{i,\theta}$. These two functions are multiplied together to form $\Lambda_{i,\theta}$, which is finally used to calculate z_i^I in a Res-Net fashion.

Figure 7.3 displays the process of updating the Interior node variable.

Neumann nodes messages One message from an incoming link is designed to enforce the homogeneous Neumann boundary conditions. It is constructed in a similar manner to (7.2), using the MLP $\Phi^{\mathtt{N}}_{\leftarrow,\theta}$ such that:

$$\phi^{\mathtt{N}}_{\leftarrow,i} = \sum_{j \in \mathcal{N}(i)} \Phi^{\mathtt{N}}_{\leftarrow,\theta} \left(H_i, H_j, d_{ji}, \|d_{ji}\| \right) \tag{7.7}$$

The updated Neumann latent variable $\mathbf{z}^{\mathtt{N}} := (z_i^{\mathtt{N}})_{i \in [N]}$ is computed by combining message (7.7) with problem-related data b_i and the information on the normal vector n_i, and passing the result through an MLP Ψ_θ as follows:

$$z^{\mathtt{N}}_i = \Psi_\theta \left(H_i, b_i, n_i, \phi^{\mathtt{N}}_{\leftarrow,i} \right) \tag{7.8}$$

GNN-based function The GNN-based function M_θ, designed to preserve Dirichlet boundary values and separate Interior and Neumann messages, is given by:

$$M_\theta(H,\, G) = \begin{cases} H^0 & \text{if Dirichlet} \\ \text{LN}\left(\mathbf{z}^{\mathtt{I}}\right) & \text{if Interior} \\ \text{LN}\left(\mathbf{z}^{\mathtt{N}}\right) & \text{if Neumann} \end{cases} \tag{7.9}$$

where LN stands for the Layer Normalization operation (Ba et al., 2016), which consists of normalizing each sample in the minibatch such that the features in the sam-

ple have zero mean and unit variance. This operation plays a crucial role in stabilizing M_θ by constraining its output, resulting in more efficient computation of the subsequent fixed point problem.

Note that it is possible to stack several layers as above for constructing the GNN-based function M_θ, at the cost of an increased number of parameters of the model. However, in practice, only one layer is sufficient in our experiments.

Fixed-point problem One step of Message-Passing only propagates information from one node to its immediate neighbours, as presented in Section 2.4. In order to propagate information throughout the graph, the DS-GPS approach introduced earlier (Chapter 6) performed the Message-Passing step repeatedly, i.e., looped over the function M_θ for a fixed number of iterations until the problem converges. The results presented in 6.3 showed that iterating until convergence amounts to solving a fixed-point problem:

$$\widehat{H} = M_\theta\left(\widehat{H},\, G\right) \tag{7.10}$$

Hence, the idea is now to use a black-box root-finding procedure to directly solve the fixed-point problem:

$$\widehat{H} = \text{RootFind}\left(M_\theta\left(H,\, G\right) - H\right) \tag{7.11}$$

This approach eliminates the need for a predefined number of iterations of M_θ and only requires a threshold precision of the root-finding solver, resulting in a more adaptable and flexible approach.

To solve Equation (7.11), Newton's method is the method of choice, thanks to its quadratic convergence guarantee. However, to avoid the costly computation of the inverse Jacobian at each Newton iteration, the quasi-Newton Broyden algorithm (Broyden, 1965) is used, which employs low-rank updates to maintain an approximation of the Jacobian. It is important to note that, regardless of the chosen root-finding solver, the algorithm starts with the initial guess H^0.

7.2.2 . Stabilization

In Section 7.2.1, we modelled a GNN-based network with an "infinite" depth by using a black-box solver to find the fixed point of the function M_θ, enabling unrestricted information flow throughout the entire graph. However, such implicit models suffer from two significant downsides: they tend to be unstable during the training phase and are very sensitive to architectural choices: small changes to M_θ can lead

to large numerical instabilities. The stability of the model around the fixed point $\widehat{H}$ is determined by the spectral radius ρ of the Jacobian $J_{M_\theta}(\widehat{H})$. Following Bai et al. (2021), who add a constraint on ρ, we add a penalization term in the loss function described in 7.2.3. However, since computing the spectral radius is far too computationally costly, and because the Frobenius norm of the Jacobian is an upper bound for its spectral radius, we adopt the method outlined in Bai et al. (2021), which estimates this Frobenius norm using the Hutchinson estimator (Hutchinson, 1990) such that:

$$||J_{M_\theta}||_F^2 = \mathbb{E}_{\epsilon \in \mathcal{N}(0,I_d)} \left[||\epsilon^T J_{M_\theta}||_2^2 \right] \tag{7.12}$$

where $J_{M_\theta} \in \mathbb{R}^{d \times d}$. The expectation (7.12) can be estimated using a Monte-Carlo method for which a single sample suffices to work well Bai et al. (2021).

Using this approach, the model satisfies, post-training, $\rho < 1$: the function M_θ becomes contractive, thus ensuring the stability of the fixed point. As a result, during inference, one can simply iterate on the Processor until convergence (i.e. loop over the M_θ function). This method aligns closely with the DS-GPS approach, using the configuration for which we observed its convergence towards an equilibrium point. More importantly, the model could work with any kind of root-finding solver. This approach offers strong convergence guarantees and addresses the stability issues commonly encountered in implicit models.

7.2.3 . Training materials

This section aims to provide additional training materials. It covers an explanation of the training loss, instructions on training an implicit model, a description of the vector format discussed in Section 7.2.1, and the procedure for data normalization.

Training loss

The entire Ψ-GNN model is trained by minimizing the following cost function:

$$\begin{aligned}
\mathcal{L} &= \mathcal{L}_{\text{res}}(\widehat{U}, G) && (7.13)\\
&+ \lambda \times MSE\left(\widehat{U} - U^{\text{ex}}\right) && (7.14)\\
&+ \beta \times ||J_{h_\theta}(\widehat{H})||_{\text{F}}^2 && (7.15)\\
&+ \text{MSE}\left(E_\theta(\widehat{U}) - \widehat{H}\right) && (7.16)\\
&+ \text{MSE}\left(D_\theta(E(\widehat{U})) - \widehat{U}\right) && (7.17)
\end{aligned}$$

Line (7.13) represents the residual loss described in (4.6), and line (7.14) is an addi-

tional supervised loss (U^{ex} being the LU ground truth and λ a small weight (as discussed in Section 6.3.2). Line (7.15) is the regularizing term defined in Section 7.2.2. Lines (7.16) and (7.17) are designed to learn the autoencoding mechanism together: Line (7.16) aims to properly encode a solution while Line (7.17) steers the decoder to be the inverse of the encoder. To handle the minimization of the overall structure, a single optimizer is used with two different learning rates, one for the autoencoding process and one for the Message-Passing process. This process allows the model to learn the representation of Dirichlet boundary conditions in the latent space faster than the main process, and ensures that the autoencoding process is solely used for the purpose of bridging the physical and latent spaces, with no direct impact on the accuracy of the computed solution. A similar technique has been used in Otto and Rowley (2019).

Training an implicit model

The training of the proposed model has been found to be computationally intensive or even infeasible when backpropagating through all the operations of the fixed-point solver. However, using the approach outlined in Theorem 1 in Bai et al. (2019) significantly enhances the training process by differentiating directly at the fixed point, thanks to the implicit function theorem. This methodology requires the resolution of two fixed-point problems, one during the forward phase and the other during the backpropagation phase.

Forward pass The resolution of a fixed point for the forward pass is clear and represents the core of our approach. A root-finding solver (i.e. a quasi-Newton method to avoid computing the inverse Jacobian of a Newton method at each iteration step) is used to determine the fixed point $\widehat{H}$ of the function M_θ (7.9) such that:

$$\widehat{H} = \text{RootFind}\left(M_\theta(H, G) - H\right) \tag{7.18}$$

Backward pass However, using a black-box solver forbids the use of explicit backpropagation through the exact operations performed in the forward pass. Thankfully, Bai et al. (2019) proposes a simpler alternative procedure that requires no knowledge of the black-box solver by directly computing the gradient at the fixed point. The gradient of the loss $\mathcal{L}$ with respect to the weights θ is then given by:

$$\frac{\partial \mathcal{L}}{\partial \theta} = -\frac{\partial \mathcal{L}}{\partial \widehat{H}} \left(J^{-1}_{M_\theta}|_{\widehat{H}}\right) \frac{\partial M_\theta(\widehat{H}, G)}{\partial \theta}. \tag{7.19}$$

where $J^{-1}_{M_\theta}|_{\widehat{H}}$ is the inverse Jacobian of M_θ evaluated at $\widehat{H}$. To avoid computing the expensive $-\frac{\partial \mathcal{L}}{\partial \widehat{H}} \left(J^{-1}_{M_\theta}|_{\widehat{H}}\right)$ term in (7.19), one can alternatively solve the following

root finding problem using Broyden's method (or any other root-finding solver) and the autograd packages from Pytorch:

$$\left(J^T_{M_\theta}|_{\widehat{H}}\right) x^T + \left(\frac{\partial \mathcal{L}}{\partial \widehat{H}}\right)^T = 0 \tag{7.20}$$

Consequently, the model is trained using a RootFind solver to compute the linear system (7.20) and directly backpropagate through the equilibrium using (7.19). In contrast to traditional methods, this approach removes the need to open the black-box, and only requires constant memory.

Data information and normalization

The structure of the b_i vector used in equations (7.4), (7.5), and (7.8), as well as the normalizing process of the data, is analogous to that of the previous chapter (see Section 6.2.3).

7.3 . Theoretical properties

This section investigates several theoretical properties of the proposed approach. Specifically, we demonstrate that Ψ-GNN satisfies a property of universal approximation, i.e. the model is able to approximate the optimal solution to the considered statistical problem (4.7) up to any arbitrary precision, provided the network is large enough.

Following the approach and notations outlined in Section 4.2, a discretized Poisson problem $E_h = (\Omega_h, A, B)$ can be seen as a graph problem $G = (N, A, B)$, where A and B are respectively the interaction and individual terms on the N nodes of the graph G. The original problem at hand searches for an optimal solution $U^\star_G \in \mathbb{R}^N$ as follows:

$$U^\star_G = \underset{U \in \mathbb{R}^N}{\text{argmin}}\ \mathcal{L}_{\text{res}}\left(U,\ G\right) \tag{7.21}$$

However, instead of searching directly for $U^\star_G$, we seek for a function $h^\star_G : \mathbb{R}^N \to \mathbb{R}^N$ whose fixed point is $U^\star_G$:

$$h^\star_G = \underset{h:\ \mathbb{R}^N \to \mathbb{R}^N}{\text{argmin}}\ \mathcal{L}_{\text{res}}\left(\text{FixedPoint}(h),\ G\right) \tag{7.22}$$

The first step in proving the consistency of the approach is to determine whether problems (7.21) and (7.22) are equivalent, i.e., if solving (7.22) yields the solution to the original problem (7.21).

Proposition 1 (Equivalence of direct and fixed-point formulations)
Problems (7.21) *and* (7.22) *are equivalent, i.e., for any problem* G*, any solution* $U_G^\star$ *of* (7.21) *can be turned into a solution* $h_G^\star$ *of* (7.22) *and vice versa.*

Proof. If $h_G^\star$ is a solution to the problem (7.22), then its fixed point is a candidate solution to the problem (7.21) and we have:

$$\mathcal{L}_{\text{res}}(\text{FixedPoint}(h_G^\star),\ G) \geqslant \mathcal{L}_{\text{res}}(U_G^\star,\ G)$$

Reciprocally, for a fixed G, if $U_G^\star$ is a solution to the problem (7.21), then the function $h_G(H) = U_G^\star$ which always outputs the same value has a unique fixed point, namely $U_G^\star$. Considering this function as a candidate to the problem (7.22), we have:

$$\mathcal{L}_{\text{res}}(U_G^\star,\ G) = \mathcal{L}_{\text{res}}(\text{FixedPoint}\left(h_G\right),\ G) \geqslant \mathcal{L}_{\text{res}}(\text{FixedPoint}(h_G^\star),\ G)$$

Consequently,

$$\mathcal{L}_{\text{res}}(U_G^\star,\ G) = \mathcal{L}_{\text{res}}(\text{FixedPoint}(h_G^\star),\ G)$$

and the problems are equivalent. □

The next step investigates whether the Ψ-GNN architecture is able to find an approximation of $h_G^\star$. To address this, we begin by proving that the problem (7.21) satisfies the hypotheses of Corollary 1 of Deep Statistical Solvers (DSS) (Donon et al., 2020).

Proposition 2 (Satisfying the hypotheses of Corollary 1 in DSS)
Problem (7.21)*, which can be rewritten as searching for the function:*

$$\begin{array}{rccl} & \mathcal{S} & \to & \mathbb{R}^N \\ \varphi: & G=(N,A,B) & \mapsto & U^\star(G) := \underset{U}{\textit{argmin}}\ \mathcal{L}_{\textit{res}}(U,\ G) \end{array}$$

satisfies the hypothesis of Corollary 1 *in Donon et al. (2020).*

Proof. Corollary 1 in Donon et al. (2020) assumes that four hypotheses must be satisfied:

1. The loss function $\mathcal{L}_{\text{res}}$ is continuous and permutation-invariant.
2. The solution $U_G^\star$ is unique.

3. The problem distribution G satisfies permutation-invariance, compactness, connectivity (each graph having a single connected component), and separability of external outputs (ensuring node identifiability).

4. The solution is continuous with respect to G.

The first hypothesis is immediately fulfilled by the specific loss function, which is suitable for problems involving GNNs and similar to the one described in Donon et al. (2020). Additionally, the existence and uniqueness of the solution are guaranteed thanks to a FEM analysis of problems (4.1) or (4.2) which includes at least one Dirichlet boundary condition. This analysis, based on the Lax-Milgram theorem (Larson and Bengzon, 2013), validates the second hypothesis. Regarding the properties of the problem distribution, our dataset generator (see Section 4.3) generates graphs with smooth boundaries within a bounded domain, following a rotation-equivariant law and ensuring that its inside has only one connected component. Node identifiability within a graph is achieved through the use of edge features representing distances to the closest neighbours, which are never equal, thereby validating the third hypothesis. If full identifiability is desired (not just almost surely), an additional descriptor can be added to each node. Experiments have actually been run in that setting, with no observable difference in performance, thereby validating the third hypothesis. The continuity of the solution $U_G^\star$ with respect to G depends on the specific choice of the loss function $\mathcal{L}_{\text{res}}$. For Poisson-like problems, similar to those considered in this thesis (4.2), continuity can be established based on the following Lemma 1, and this will conclude the proof. □

Lemma 1 (Continuity of φ)
The mapping

$$\begin{array}{rccl} & \mathcal{S} & \to & \mathbb{R}^N \\ \varphi : & G = (N, A, B) & \mapsto & U^\star(G) := \underset{U}{\textit{argmin}} \, \|AU - B\|^2 \end{array}$$

is continuous w.r.t. A and B.

Proof. Linear systems such as (4.3), which result from the FEM discretization of Poisson problems (4.1) or (4.2), have a unique solution given by $U^\star(G) = A^{-1}B$. This solution is linear in B and thus continuous with respect to B. The question at hand is whether the mapping from A to its inverse A^{-1} is also continuous. We can express A^{-1} as

$$A^{-1} = \frac{\text{adj}(A)}{\det(A)}$$

where $\text{adj}(A)$ denotes the adjoint of A and $\det(A)$ represents the determinant of A. Both the adjoint operation and the determinant are continuous operations. The

remaining aspect to consider is whether the determinant of A can be zero or not. In the specific settings where A arises from the discretization of (4.1) or (4.2), including boundary conditions, the determinant of A is always non-negative. This completes the proof. □

Let $\mathcal{H}$ be the space of functions that can be realized by GNN neural networks, as defined in Donon et al. (2020). The previously mentioned Corollary 1 in Donon et al. (2020) establishes the existence of a network with a GNN-based architecture, denoted as $\widehat{\varphi} \in \mathcal{H}$, that can approximate $\varphi : G \mapsto U_G^\star$ with arbitrary precision for any given graph G from the considered problem distribution. We have the following corollary:

Corollary 1 (Existence of a GNN model approximating φ)
For any $\varepsilon > 0$, there exists a GNN-based model $\widehat{\varphi} \in \mathcal{H}$ such that for any problem G from our problem distribution $\mathcal{D}$:

$$\|\widehat{\varphi}(G) - \varphi(G)\| \leqslant \varepsilon$$

Based on Proposition 1 and its proof, this result can be extended to approximate the solution of (7.22), yielding the following universal approximation property for our Ψ-GNN method:

Theorem 1 (Universal Approximation Property)
For any precision $\varepsilon > 0$, there exists a parameterization θ_ε of a Ψ-GNN architecture with sufficiently large layers, namely, a function Ψ-GNN$_{\theta_\varepsilon} : \mathcal{S} \to \mathbb{R}^N$, which, for any problem G (i.e., for any mesh, boundary conditions and force terms), approximates the optimal solution of problem (7.21) with precision less than ε:

$$\forall\, G, \quad \|\Psi\text{-GNN}_{\theta_\varepsilon}(G) - \varphi(G)\| \leqslant \epsilon$$

Proof. The architecture of $\widehat{\varphi}$ obtained by Corollary 1 can be decomposed into two blocks: a $\widehat{\varphi}_{\text{GNN}}$ which consists of GNN layers that are applied to a hidden state H, and a $\widehat{\varphi}_{\text{Dec}}$ block that represents the decoder (see Section 5.2.1). Here, we attempt to build a function whose fixed point w.r.t. H approximates the output of the $\widehat{\varphi}_{\text{GNN}}$ block. To do this, we consider the function

$$h_{\theta_\varepsilon} : \begin{array}{ccc} \mathbb{R}^{N\times d} \times \mathcal{S} & \to & \mathbb{R}^{N\times d} \\ (H,\, G) & \mapsto & \widehat{\varphi}_{\text{GNN}}(G) \end{array}$$

where d is the dimension of the hidden space. For any fixed G, this function h_{θ_ε} is constant w.r.t. H, and, consequently has a unique fixed point w.r.t. H, which is $\widehat{\varphi}_{\text{GNN}}(G)$. As Ψ-GNN encompasses the $\widehat{\varphi}$ architecture, we can indeed represent exactly h_{θ_ε} using the Ψ-GNN Processor defined in Section 7.2.1. The complete architecture of Ψ-GNN can be written as follows:

$$\widehat{U} = \text{Dec}\left(\text{FixedPoint}(h_{\theta_\varepsilon}\left(\text{Enc}\left(U\right), G\right)\right)$$

where the decoder Dec is $\widehat{\varphi}_{\mathsf{Dec}}$. The encoder Enc, which is an additional feature of Ψ-GNN architecture, can be chosen arbitrarily since, in this proof, h_{θ_ε} always outputs the same value, regardless of H. This completes the proof. □

Lastly, the set of functions h that admit a unique fixed point close to $\widehat{\varphi}_{\mathsf{GNN}}(G)$ can be big (although all these solutions lead approximately to the same fixed point). It is possible to select an optimal solution within this space that is also contractive. This can be achieved in a Lagrangian spirit instead of a supplementary constraint by introducing a penalty term to the loss function, as shown in equation (7.15). However, it is important to note that the resulting function can only be assumed to be predominantly contractive with respect to H:

Proposition 3 (Contractivity of $h_{\theta\varepsilon}$)
For any precision $\varepsilon > 0$, the function $h_{\theta_\varepsilon}(H, G)$ in the Processor of the Ψ-GNN architecture obtained by Theorem 1 can be assumed to be contractive with respect to H for any pair of points farther than $\sqrt{\varepsilon}$.

Proof. The trainable function $h_{\theta_\varepsilon}(H, G)$ from Theorem 1 can be assumed to be an approximation of a contractive function f (e.g., as built in the proof). For the sake of simplicity, we omit the Decoder here. Thus, there exists $\lambda < 1$ such that, for any problem G and any latent values H, H':

$$d(f(H), f(H')) \leqslant \lambda\, d(H, H')$$

where d denotes the Euclidean distance. Note that $\lambda \in [0, 1[$ can even be 0, e.g. for $f(H) = U_G^\star$. Consequently, we have :

$$d(h(H), h(H')) \leqslant d(f(H), f(H')) + 2\varepsilon \leqslant \lambda\, d(H, H') + 2\varepsilon$$

given that $\|h(H) - f(H)\| \leqslant \varepsilon$ and $\|h(H') - f(H')\| \leqslant \varepsilon$.

Suppose ε is small enough such that $\mu = \lambda + 2\sqrt{\varepsilon} < 1$. Then, for H, H' satisfying $d(H, H') > \sqrt{\varepsilon}$, we have:

$$d(h(H), h(H')) \leqslant \mu\, d(H, H')$$

since $\lambda\, d(H, H') + 2\varepsilon < \mu\, d(H, H')$ is equivalent to $d(H, H') > \frac{2\varepsilon}{\mu - \lambda} = \sqrt{\varepsilon}$.

Thus h is contractive for all pairs of points farther than $\sqrt{\varepsilon}$ from each other. In particular, if $d(H, h_G^\star) > \sqrt{\varepsilon}$, then $d(h(H), h(h_G^\star)) \leqslant \mu\, d(H, h_G^\star)$, which implies the exponential convergence of an iterative power method from any initialization H to

the ball of radius $\sqrt{\varepsilon}$ around the fixed point $h^\star_G$, which is itself at distance at most ε from the true optimal solution $U^\star_G$. Thus the function h is predominantly contractive in that sense. Using $\varepsilon' = \varepsilon + \sqrt{\varepsilon}$ then gets rid of square roots. □

7.4 . Experiments & Results

This section presents an in-depth evaluation of Ψ-GNN based on experiments on synthetic data. The first Section 7.4.1 assesses the performance of Ψ-GNN for solving Poisson Poisson problems with Dirichlet boundary conditions, while Section 7.4.2 is focused on Poisson problems with mixed boundary conditions. Furthermore, Section 7.4.3 proposes various generalization tests to conduct a thorough assessment of Ψ-GNN performance, showcasing its originality. Finally, Section 7.4.4 investigates the inference complexity of the proposed model.

Throughout these experiments, we consider the solution of the discretized Poisson problem given by the classical LU decomposition method as the "ground truth". The reported metrics are the Residual Loss (4.6) and the Mean Squared Error (MSE) between the output of the model and the "ground-truth" LU solution.

7.4.1 . Poisson problems with Dirichlet boundary conditions

This section aims to assess the performance of Ψ-GNN for solving Poisson problems with Dirichlet boundary conditions, allowing for direct comparison with the Deep Statistical Solvers (DSS) and DS-GPS.

Experimental setup

The dataset used in this experiment is the same as the one used for training the Deep Statistical Solvers and DS-GPS models. As a reminder, it consists of 6000 training, 2000 validation, and 2000 test samples of Poisson problems with Dirichlet boundary conditions generated following the process described in Section 4.3. All meshes have approximately 500 nodes. Provided results regarding the DSS model are those presented in Chapter 5. The DS-GPS model used is the one from Chapter 6, trained with 30 iterations and $\gamma = 1$.

Ψ-GNN is implemented in Pytorch using the Pytorch-Geometric library (Fey and Lenssen, 2019) to handle graph data. The dimension d of the latent space $\mathcal{H}$ is set to 10. Each neural network block in the architecture has one hidden layer of dimension 10 with a ReLU activation function. All model weights are initialized using Xavier initialization (Glorot and Bengio, 2010). For the training, the provided initial solution is set to zero everywhere except at the Dirichlet nodes, which are assigned to their corresponding exact value. The model requires, at each iteration, the solution of two fixed point problems, one for the forward pass and one for the backward pass, as outlined in Section 7.2.3. These problems are solved using Broyden's method

(Broyden, 1965) with a relative error as the stopping criteria. The latter is set to 10^{-5} with a maximum of 500 iterations for the forward pass and to 10^{-8} with a maximum of 500 iterations for the backward pass. The error on the backward pass is intentionally made smaller than that of the forward pass because it ensures better stability during backpropagation, while the forward pass does not need to be as precise. However, the smaller the error, the greater the number of iterations required to find the fixed point, and consequently, the more costly the training becomes. These hyperparameters were chosen after numerous trials, but a possible direction for further exploration would be to determine the limits of these hyperparameters that ensure a better trade-off between precision and training time. The proposed Ψ-GNN model follows the architectural specifications outlined in Section 7.2.1, with the exception of the part related to Neumann nodes. During training, the model is optimized using the loss function (7.13), with $\lambda = 0$ (no additional minimization of the MSE w.r.t. the "ground-truth") and $\beta = 1$. Training is done using Nvidia P100 GPUs and the Adam optimizer with its default Pytorch hyperparameters, except for the initial learning rate, which is set to 0.05 for the autoencoding process and 0.01 for the main process, as discussed in Section 7.2.3. The *ReduceLROnPlateau* scheduler from Pytorch is used to progressively reduce the learning rate from a factor of 0.5 during the process. Some experiments not shown here have demonstrated that this enhanced the training. Gradient clipping is employed to prevent exploding gradient issues and set to 10^{-2}. Figure 7.4 displays the evolution of the training and residual losses during the 700 epochs, showcasing that the training has been effective.

Results

Figure 7.5 illustrates the resolution with Ψ-GNN of a test instance, a Poisson problem with 527 nodes. The figure presents the evolution of the Residual (the loss function) and the MSE w/LU (though it was not used during training) along the 68 iter-

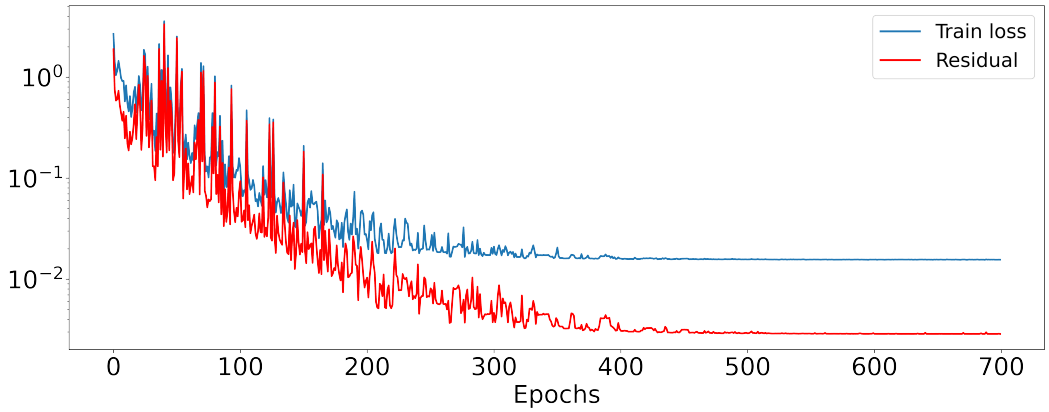


Figure 7.4: Evolution of the Training (blue) and Residual (red) losses during the 700 epochs.

ations of the Broyden algorithm. At convergence, the Residual reaches a value of 3.36e-3 and an MSE w/LU of 1.03e-2. Similar results on all test examples validate the Ψ-GNN approach, showcasing its ability to solve Poisson problems with Dirichlet boundary conditions.

Table 7.2 presents the Residual and MSE w/LU averaged over the entire test set for the three methods Ψ-GNN, DSS, and DS-GPS. From this table, it is possible to observe that both DSS and DS-GPS provide slightly better results than Ψ-GNN in terms of Residual. This may be due to the fact that the training loss no longer minimizes the Residual only but also incorporates several other components within the training procedure, such as the stabilization process, which adds additional constraints to the weights of the model. However, the results take a different flavour when looking at the MSE w/LU. There, we can see that Ψ-GNN outperforms both other models, without the addition of any supervised loss during training! This can be explained by the ability of the model to automatically adjust its number of Message-Passing steps, as well as the autoencoding mechanism that better encodes and decodes Dirichlet boundary conditions up to a precision of order 10^{-6}. This enhanced feature allows for better information flow since the initial latent state is properly initialized, and results in a better MSE w/LU. Additionally, Ψ-GNN is able to outperform these models with only 1444 parameters, in contrast to the DSS model, which requires 36930 iterations (performing only 30 iterations!). Finally, Ψ-GNN is trained on the same dataset as DSS and DS-GPS, which contains meshes of approximately 500 nodes. However, Ψ-GNN, even though it is trained on fixed-size meshes, has the remarkable capability of adjusting its iteration count by itself to handle meshes of varying sizes, a feature analyzed in detail in Section 7.4.3.

7.4.2 . Poisson problems with mixed boundary conditions

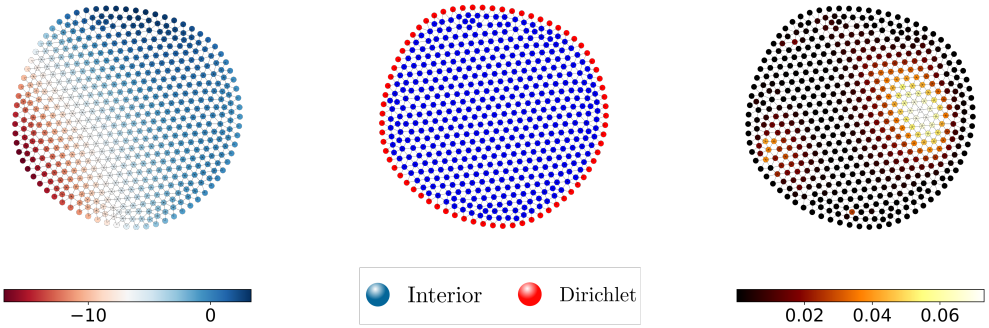
This section aims to assess the performance of Ψ-GNN when extended to solve Poisson problems with mixed boundary conditions.

Experimental setup

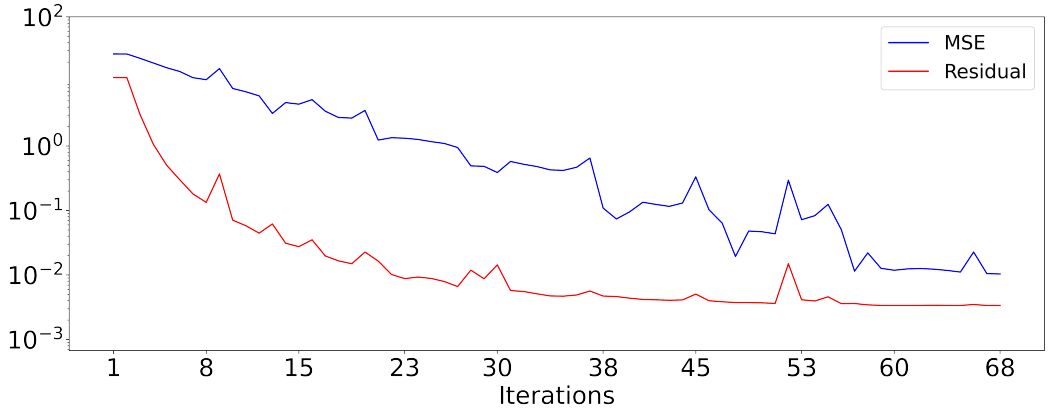
The dataset used in this experiment is the same as in Section 6.3.2. For this experiment, the Ψ-GNN model follows the architectural specifications outlined in Sec-

	Residual (10^{-3})	MSE w/LU (10^{-2})	Nb of weights
Ψ-GNN	2.69 ± 0.4	0.85 ± 2	1444
DSS	0.23 ± 0.2	3.0 ± 2	36930
DS-GPS	1.30 ± 0.1	6.37 ± 2	1961

Table 7.2: Results of Ψ-GNN, DSS and DS-GPS averaged over the whole test set



(a) *Data-driven solution* (b) *Node types* (c) *Error map*



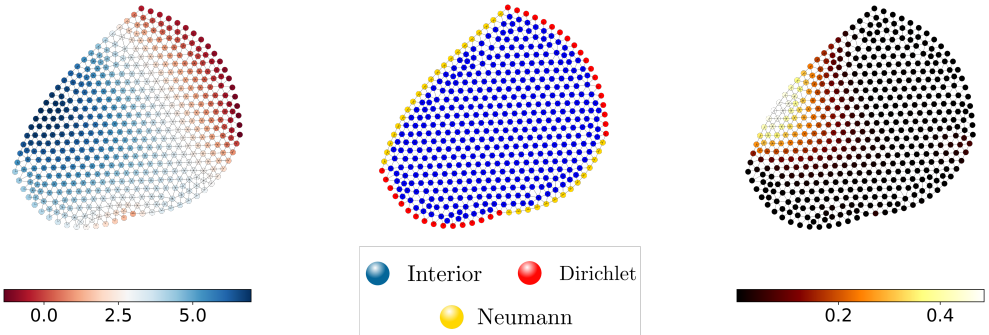
(d) *Residual (red) and MSE w/LU (blue) losses*

Figure 7.5: Illustration of the resolution of a Poisson problem extracted from the test set using Ψ-GNN. Figure 7.5a shows the data-driven solution obtained at the last iteration, while Figure 7.5c displays the map of squared errors between the data-driven solution and the LU solution. Figure 7.5b illustrates the different types of nodes. At the bottom, Figure 7.5d depicts the evolution of the Residual (in red) and MSE w/LU (in blue) across the 68 iterations of the model.

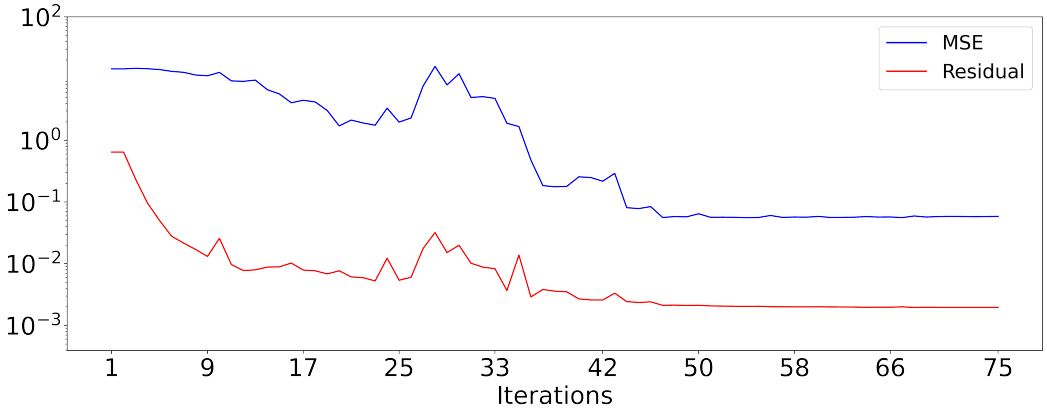
tion 7.2.1. The hyperparameters used are similar to those employed in the Dirichlet boundary condition section (previous Section 7.4.1), except for the λ parameter, which is set to 0.001. This parameter introduces a small additional supervised loss to enhance the training, as discussed in Section 6.3.2.

Results

Figure 7.6 displays the solution to a specific problem with 470 nodes extracted from the test set. At convergence, the model reaches a Residual of 1.9e-3 and an MSE w/LU of 5.8e-2, showcasing the effectiveness of the method on this test sample.



(a) *Data-driven solution* (b) *Node types* (c) *Error map*



(d) *Residual (red) and MSE w/LU (blue) losses*

Figure 7.6: Illustration of the resolution of a Poisson problem extracted from the test set using Ψ-GNN. Figure 7.6a shows the data-driven solution obtained at the last iteration, while Figure 7.6c displays the map of squared errors between the data-driven solution and the LU solution. Figure 7.6b illustrates the different types of nodes. At the bottom, Figure 7.6d depicts the evolution of the Residual (in red) and MSE w/LU (in blue) across the 50 iterations of the model.

The autoencoding process ensures accurate encoding and decoding of the Dirichlet boundary conditions, thereby preserving them throughout the iterations with an error of magnitude 10^{-6}. The error map reveals that the highest errors are primar-

Metrics	Residuals (10^{-3})	MSE w/LU	Nb of weights
Ψ-GNN	3.16 ± 0.2	0.14 ± 0.04	2175
DS-GPS	2.2 ± 0.1	0.37 ± 0.1	2711

Table 7.3: Results of Ψ-GNN and DS-GPS averaged over the whole test set.

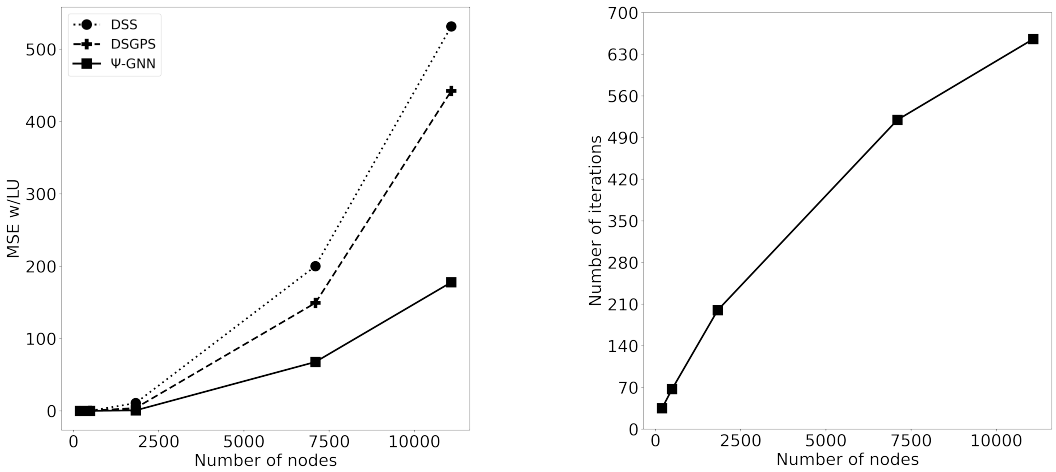
ily concentrated near the Neumann boundary nodes, aligned with expectations. As discussed in Section 4.2, the graphs considered in this study are directed from the Dirichlet boundary nodes toward the interior of the graph. Consequently, the flow of information is propagated from these nodes towards the inner region of the domain. However, in the case of Neumann nodes, which involve bidirectional edges, the task of propagating information across the entire graph becomes more challenging compared to the fully Dirichlet problem. This is because, in the full Dirichlet problem, information can flow from the entire boundary of the domain. However, in the presence of Neumann nodes, the bidirectional edges complicate the propagation of information throughout the graph, and the Neumann values must be gradually approached. This is in contrast to Dirichlet values, which are considered "exact". As a result, the model requires more iterations to attain the solution. Additionally, Table 7.3 presents the averaged Residual and MSE w/LU errors obtained by Ψ-GNN and DS-GPS on the entire test set. Results show that Ψ-GNN outperforms DS-GPS in terms of MSE w/LU. This is attributed to the capability of Ψ-GNN to better adapt its number of Message-Passing steps required for solving such problems. Additionally, these results are comparable in terms of Residual and slightly higher in terms of MSE w/LU compared to those obtained in Section 7.4.1. This increase is attributed to the higher conditioning of the problem. However, these results demonstrate the accuracy of Ψ-GNN in solving Poisson problems with mixed boundary conditions. Notably, the model has 2175 parameters, which is slightly more than the one developed in Section 7.4.1, due to the additional networks required to take into account the Neumann boundary conditions.

7.4.3 . Sensitivity analyzes

This section delves into several generalizations aspects of Ψ-GNN. First, we demonstrate that Ψ-GNN remains consistent even with an increasing number of nodes in the graph. Then, we demonstrate the flexibility of Ψ-GNN through its insensitivity with respect to its initialization. Moreover, we provide evidence of the contractive nature of the constructed GNN function M_θ, as discussed in Section 7.2.2. Lastly, we address the out-of-distribution generalization issue, effectively highlighting the better generalization capabilities of Ψ-GNN compared to DS-GPS and DSS.

Size of the mesh

This paragraph explores the performance and generalization capabilities of Ψ-GNN in solving Poisson problems with Dirichlet boundary conditions on meshes with a growing number of nodes, even though it was initially trained on meshes with approximately 500 nodes. To conduct this experiment, we address the resolution of multiple Poisson problems on meshes with varying numbers of nodes. To maintain consistency in the distribution of inputs, we keep the same element sizes, similar to those used to generate the dataset, while varying the number of nodes by in-



(a) *(b)*

Figure 7.7: Figure 7.7a illustrates the averaged MSE w/LU for different mesh sizes for the three models: DSS, DS-GPS and Ψ-GNN. Figure 7.7b displays the averaged number of iterations performed by the Broyden solver in Ψ-GNN to reach its target threshold with respect to the number of nodes per mesh.

creasing the mesh radius. The force and boundary functions are randomly sampled following Section 4.3 but rescaled according to the selected radius. We consider six different setups corresponding to meshes with approximately 200, 500, 2000, 7000, and 11000 nodes per mesh. For each setup, we solve 200 Poisson problems using DSS, DS-GPS, and Ψ-GNN. To handle larger mesh sizes, the DS-GPS model is allowed up to 500 iterations for solution inference. The Broyden method used in the Ψ-GNN model is configured with a stopping criterion of 10^{-5} and a maximum of 1000 iterations. Notably, regardless of the mesh size, the DS-GPS model performs exactly 500 iterations, while the Ψ-GNN model adjusts its iteration count thanks to the root-finding procedure. The DSS model, as used in Chapter 5, infers solutions using only 30 iterations. Figure 7.7a displays the MSE w/LU evolution (averaged over the 200 problems) for each setup and each of the three models. The figure clearly shows that the DSS model diverges for larger meshes. While the DS-GPS model initially achieves better results, it eventually diverges due to inconsistencies in the convergence of the method. In contrast, the Ψ-GNN model remains consistent when applied to larger meshes. Furthermore, Figure 7.7b displays the averaged iteration counts of the Broyden solver in the Ψ-GNN model with respect to the number of nodes per mesh. This figure clearly demonstrates that Ψ-GNN can adapt its number of Message-Passing layers to attain a solution.

Initialization

This paragraph aims to demonstrate the flexibility of Ψ-GNN and its insensitivity w.r.t. its initialization. By "flexible", we mean that Ψ-GNN can dynamically adapt

	Original	Random Initializer	Forward Iteration
Residual (10^{-3})	3.16 ± 0.2	3.18 ± 0.3	3.14 ± 0.3
MSE w/LU	0.15 ± 0.04	0.16 ± 0.09	0.16 ± 0.04

Table 7.4: Results averaged over the whole test set for different experiments. First column: original results of Section 7.4.2; Second columns: random initial solution; Third column: replacing Boyden with forward iterations.

its number of iterations based on the distance from the initial solution to the final solution. And by "insensitive", we refer to the property that, regardless of the initial solution, the model consistently converges to the same fixed point, representing the desired solution. These advantageous characteristics are enabled by the auto-encoding process that maps the initial physical state to the latent space, where GNN layers are applied, and back. Flexibility is demonstrated thanks to Figure 7.8, which displays the evolution of the Residual error across the iterations of the Broyden algorithm for the same Poisson problem considering three different initial conditions: the "Original" initial condition (black curve) is the one used during training, where the initial state is initialized to 0. The "Far" initial condition (blue curve) involves applying a large random noise (uniform noise in the range $[-1000, 1000]$) to the solution obtained from the LU "ground-truth" method. The "Close" condition (red curve) incorporates a small perturbation (uniform noise in the range $[0, 1]$) to the target "ground-truth" solution. For all these initial conditions, the true value of Dirichlet boundary conditions is preserved. This experiment showcases that the number of iterations required for convergence varies depending on the proximity of the initial solution to the final solution. This adaptive behaviour enables the model to adjust its iteration count effectively, optimizing convergence efficiency. Insensitivity to initial solutions is demonstrated in the second column of Table 7.4, which evaluates the performance of Ψ-GNN using various initial solutions. This experiment demonstrates that Ψ-GNN is robust to the choice of initial solution: regardless of the initial solution provided, the algorithm consistently converges to the desired solution. Specifically, the "Random Initialization" entry in Table 7.4 reports the metrics averaged over the entire test set, using an initial solution randomly generated by perturbing the ground-truth solution with uniformly random values ranging from -1000 to 1000. Remarkably, the results obtained with random initialization are almost identical to those obtained with the "Original setup" (first column of 7.4 identical to the results from Section 7.4.2) indicating that our model is insensitive to the specific choice of the initial solution. In particular, this is a significant improvement compared to the DSS approach, which lacks this capability since it does not have any encoding process.

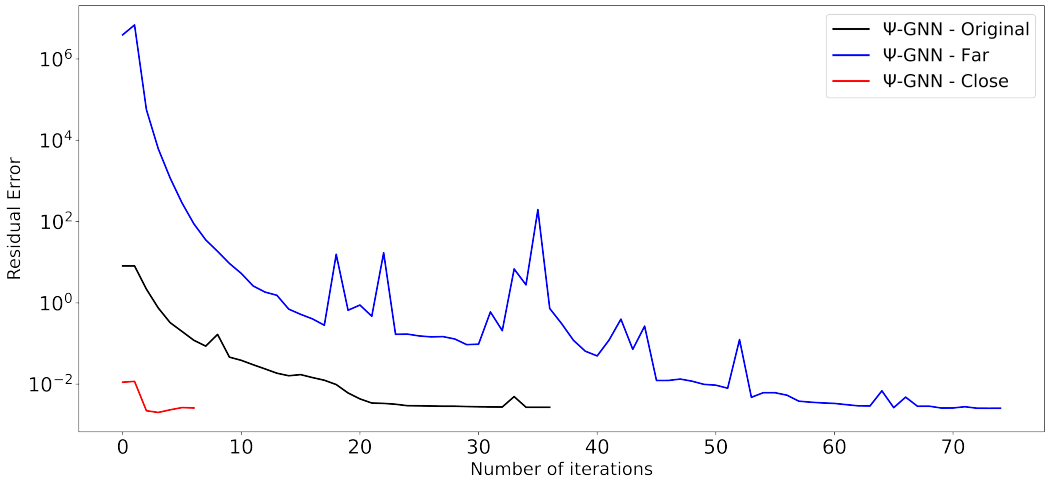


Figure 7.8: Evolution of the Residual error across iteration of Ψ-GNN for the same Poisson problem but considering three different initial solutions, demonstrating the adaptability of Ψ-GNN with respect to various initial solutions.

Contractivity

The regularization term (7.15) serves the purpose of constraining the spectral radius of the Jacobian $J_{M_\theta}(\widehat{H})$ in order to ensure the stability of the model around the fixed point $\widehat{H}$, as discussed in Section 7.2.2. Figure 7.9 depicts the evolution of the spectral radius during the training phase, evaluated for each mini-batch at every validation step. The figure illustrates that the regularization term indeed forces the spectral radius of the Jacobian to converge toward a value close to 1. The results obtained on the test set indicate the effectiveness of this regularization, with an average spectral radius of $0.993 \pm 4\text{e-}4 < 1$. This value is estimated a posteriori using the Power Iteration method (Golub and Van der Vorst, 2000). Furthermore, we demonstrate this generalization results in the third column of Table 7.4. In this experiment, we evaluate Ψ-GNN on the entire test set, but instead of using the root-finding Broyden algorithm employed during training, we simply iterate on the Processor described in the architecture in Section 7.2.1. Once again, the obtained results are almost identical to those obtained in

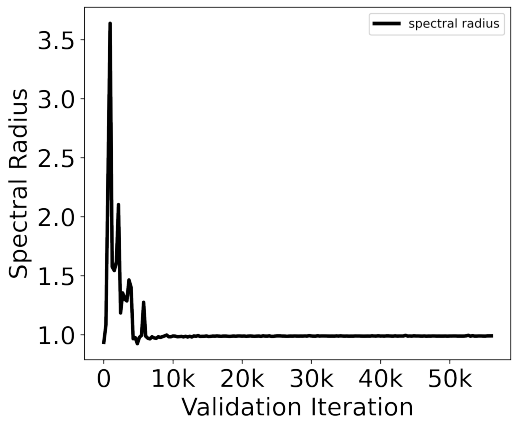


Figure 7.9: Evolution of the spectral radius.

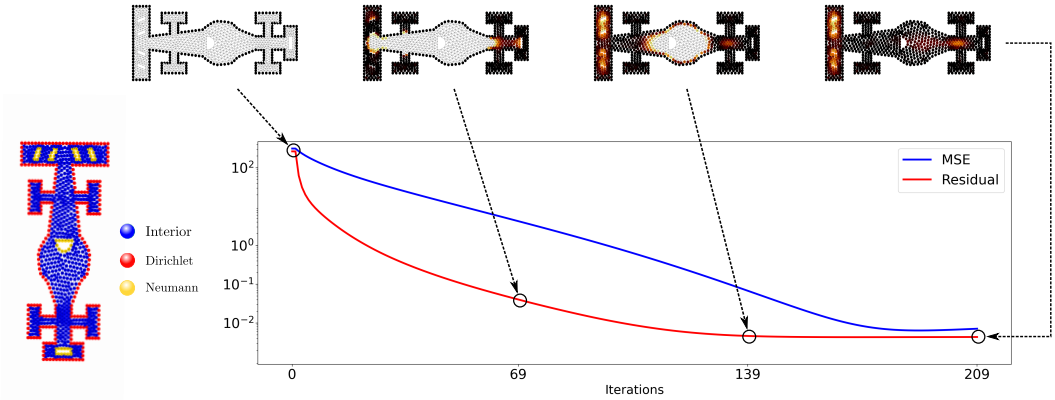


Figure 7.10: Generalization on the "out-of-distribution" F1 shape, 1219 nodes. Central plot: Residual and MSE during the 253 iterations of the Processor (i.e., without using RootFind), demonstrating the contractivity of M_θ. Left: The boundary conditions. Top: the visual evolution of the squared error, displaying the flow of information from Dirichlet nodes inward.

the original setup. This demonstrates the contractive nature of the GNN-function M_θ and highlights the robustness and flexibility of Ψ-GNN in its ability to adapt to different solvers.

Out-of-distribution generalization

We conduct an experiment on a mesh representing a caricatural Formula 1 with 1219 nodes. This mesh includes "holes" (such as a cockpit and front and rear wing stripes) and is larger (1219 nodes) than those seen in the training dataset, providing a challenging test of the model's ability to generalize to out-of-distribution (with respect to the geometry of the domain, as well as the mesh size) examples. We impose Dirichlet boundary conditions on all exterior nodes (pink nodes in the vertical plot at the left of Figure 7.10) and Neumann conditions on the nodes within the "holes" (yellow nodes). Functions f and g of (4.2) are randomly sampled from the same distribution as for the training set (Section 4.3). The equilibrium of the model is found by simply iterating on the Processor instead of relying on Broyden's algorithm (see above Section). The stopping criteria is the relative error set to 10^{-4}. The results Figure 7.10 again shows the contracting nature of the learned function, that converges to the fixed point when iterated. Furthermore, it also gives an example of the generalization capacity of the learned model to some out-of-distribution examples. Additionally, the figure also illustrates how the information propagates through the graph, starting from the Dirichlet nodes to gradually filling the whole domain.

7.4.4 . Inference complexity

To determine the expected performance of Ψ-GNN as the number of nodes N in a graph varies, it is essential to analyze its complexity. In Ψ-GNN, all neural networks have a single hidden layer of dimension d, so the complexity of applying a neural network to one node is $\mathcal{O}\left(d^3\right)$. Let us assume that m is the average number of neighbours for each node in the graph. The complexity of computing the output of a GNN layer relative to one node in a mesh is then of $\mathcal{O}\left(md^3\right)$. Considering all N nodes in the graph and iterating for K updates, the complexity becomes $\mathcal{O}\left(KNmd^3\right)$. This represents the theoretical complexity of Ψ-GNN when the Processor is iterated upon for K iterations.

When using Ψ-GNN with Broyden's algorithm, the complexity is found to be of a higher order. Broyden's method is a quasi-Newton method that computes the next iterate H^{k+1} as follows:

$$H^{k+1} = H^k + J^{-1}_{|M_\theta}(H^k)M_\theta(H^k) \tag{7.23}$$

In Equation (7.23), we already know the complexity of $M_\theta(H^k)$, which is computed in a similar manner as previously explained and has a complexity of $\mathcal{O}\left(Nmd^3\right)$. In his paper (Broyden, 1965), Broyden suggests using the Sherman-Morrison formula (Sherman and Morrison, 1950) to update the inverse of the Jacobian matrix directly, as follows:

$$B^k_{|M_\theta} = B^{k-1}_{|M_\theta} + \frac{\Delta H^k - B^{k-1}_{|M_\theta}\Delta M^k_\theta}{(\Delta H^k)^T B^{k-1}_{|M_\theta}\Delta M^k_\theta}(\Delta H^k)^T B^{k-1}_{|M_\theta} \tag{7.24}$$

Here, $B^k_{|M_\theta} = J^{-1}_{|M_\theta}(H^k)$, $\Delta H^k = H^k - H^{k-1}$, and $\Delta M^k_\theta = M_\theta(H^k) - M_\theta(H^{k-1})$.

In Equation (7.24), all operations are matrix-vector products of size N, so the complexity of updating the Jacobian matrix is $\mathcal{O}\left(N^2\right)$. Back to equation (7.23), the total complexity to compute the next iterate is then of $\mathcal{O}\left(N^2\right) + \mathcal{O}\left(Nmd^3\right)$, which corresponds to the cost of updating the Jacobian matrix and the cost of evaluating the GNN model. This is applied for M iterations of the Broyden algorithm, resulting in a global complexity of $\mathcal{O}\left(MN^2\right)$, since the quadratic complexity dominates the linear one.

Experimentally however, at the current stage of our work, whatever the approach (iterations of the Processor, or Broyden iterations), the model does not show any improvement in computational speed compared to traditional solvers, primarily due to the small sizes of the meshes and the complexity of the Broyden algorithm. Nevertheless, a promising future research direction involves considering multiple layers within the function M_θ. In essence, the idea would be to explore a trade-off between stacking GNN layers, which have a linear complexity, within the M_θ function

to reduce the number of iterations of the root-finding solver, which has a quadratic complexity. This trade-off could eventually enhance computational efficiency and is worth exploring further.

7.5 . Discussion and Conclusions

In this chapter, we have introduced Ψ-GNN, a novel Machine Learning-based approach that combines Graph Neural Networks and Implicit Layer Theory to effectively solve a wide range of Poisson problems. The model, trained in a "physics-informed" manner, is found to be robust, stable, and adaptable to varying mesh sizes, domain shapes, boundary conditions, and initialization. To the best of our knowledge, this approach is distinct from any previous Machine Learning-based methods for Poisson resolution, and outperforms state-of-the-art models, both quantitatively and qualitatively. Furthermore, Ψ-GNN can be extended to other steady-state partial differential equations, and its application to 3-dimensional domains is straightforward.

Despite its generalization capabilities, Ψ-GNN is still limited to handling small-size meshes (with less than 1000 nodes). Indeed, since the main optimization process is driven by minimizing the discretized residual equation, the performance of the model when comparing it with ground-truth solutions is constrained by the conditioning of the considered problem. This conditioning increases with both the number of nodes and the difficulty of the problem, such as adding homogeneous Neumann boundary conditions. Nevertheless, although it is trained on fixed-size meshes, Ψ-GNN has been found to be robust when applied to meshes with a larger number of nodes. However, this result comes at the expense of an increasing number of iterations of the root-finding method, which significantly increases the computational time for inferring a solution. The question now is how to efficiently scale these GNN models to solve large-scale problems, and there are potentially promising directions. One approach is to use these models as a solver for domain decomposition algorithms, leveraging the batch parallel framework of Machine Learning models to solve multiple subproblems simultaneously. In that context, one could combine the strengths of well-established numerical methods and the efficient computational power of Machine Learning models. More importantly, this would allow the model to produce solutions within its range of nodes by decomposing the full domain into subdomains of reasonable sizes for the ML-based approach. This analysis is explored in Part III of this manuscript. Another potential direction for future research, which has not been explored in this thesis, is the development of a fully hierarchical architecture, similar to the work done in Liu et al. (2021) or Lino et al. (2021a).

Part III

Hybrid Solvers

In the field of Computational Fluid Dynamics (CFD), the objective is to forecast the behaviour of a fluid within a domain subjected to physical constraints. As explained in Chapter 1, the initial step involves deriving the Partial Differential Equations (PDEs) that describe the motion of fluids, referred to as the Navier-Stokes equations. These equations are often solved using splitting schemes, which necessitate the intensive resolution of a Poisson Pressure problem, as described in Section 1.2 and highlighted in Figure 2.10 of Wang (2015). The Poisson Pressure PDE is then discretized on a mesh using numerical tools like the Finite Element Method (see Section 1.3), which yields the resolution of a linear system of the form

$$A\mathbf{u} = \mathbf{b}$$

whose size equals the number of degrees of freedom, denoted as N. In this system, $A \in \mathbb{R}^{N\times N}$ denotes the stiffness matrix, $\mathbf{b} \in \mathbb{R}^N$ represents the right-hand side vector, and $\mathbf{u} \in \mathbb{R}^N$ is the solution vector to be sought.

For complex industrial problems, accurate predictions are mandatory, which implies an increased number of degrees of freedom and, thus, a very large system to solve. For example, it is often not surprising that the system to solve has a number of degrees of freedom of the order of the million. To solve such systems, there exist different kinds of methods that can be divided into two categories: direct and iterative methods.

Direct methods

Direct methods aim to find the exact solution in a finite number of steps, and the algorithm has to complete all the steps to obtain a solution. A simple way would be to invert the matrix A and multiply the result with the right-hand side $\mathbf{b}$. However, for large systems, an inverse is never computed explicitly. Instead, more common approaches to directly compute the solution include Gaussian elimination, LU, Cholesky, or QR decompositions (Davis, 2006). Direct methods are robust and, in the absence of rounding errors, would provide an exact solution to the system. However, the challenge with direct methods is that of scaling: they are rarely used in practice for dense matrices when the system of equations is larger than $\simeq 1000$. The main reason concerns the complexity of these algorithms. For instance, Gaussian elimination has a complexity of order $\mathcal{O}(N^3)$, which means that the algorithm will perform on the order of 10^{15} floating point operations to obtain the solution of a problem with $100,000$ degrees of freedom, a process that can be extremely time-consuming. Specific variants of these algorithms were developed to handle sparse systems of equations with a complexity of $\mathcal{O}(N^\alpha)$, where $1 \leq \alpha \leq 2$ (Rose and Tarjan, 1978). These methods can still be used to solve 2D and even 3D problems with thousands of unknowns but are still scarcely used in practice for very large systems. Another reason, for dense matrices, is that for large problems, it is unlikely that the entire matrix can be stored in memory or even assembled in the FEM. Refer

to Davis (2006) or the survey by Davis et al. (2016) for an exhaustive presentation of these methods.

Iterative methods

Iterative methods, on the other hand, gradually improve an approximation of the solution that will (hopefully) converge to the true solution. In contrast to the direct method, where all the steps of the algorithm must be completed to obtain a solution, iterative methods let the user decide how much work (how many iterations) she wants to invest, depending on how accurate the current approximation is to the true solution. Moreover, these methods are matrix-free methods because they do not necessarily require the explicit form of the matrix A. Instead, they access the matrix by evaluating matrix-vector products. Such methods can be preferable when the matrix is so big that storing and manipulating it would cost a lot of memory and computing time, even with the use of methods for sparse matrices.

Before introducing common iterative methods used for solving large systems, it is essential to define key concepts. In the following discussion, the term "robustness" refers to the guarantees provided by an iterative method for converging to the solution. Next, "efficiency" relates to speed (the number of iterations) of convergence to reach a given precision, and is directly linked to the convergence rate of the iterative method: a higher convergence rate implies fewer iterations needed to find the solution. Furthermore, the notion of "scalability" can be split into two concepts: *strong* and *weak* scalability. Strong scalability quantifies how the convergence rate changes with respect to the quantity of resources (computational resources, e.g., the number of CPU cores) for a given problem size. Ideally, for a fixed problem size, if computational resources double, the solution time should be twice as fast. On the other hand, weak scalability concerns how the solution time varies with the number of resources as the problem size increases. Ideally, the solution time should remain constant for a fixed ratio between the problem size and the number of resources. *In the following, the term scalability will only refer to the concept of weak scalability.*

The most basic iterative methods are the so-called stationary methods, in which all iterations use the same formula. Examples of iterative methods are Jacobi, Gauss-Seidel, or SOR (Greenbaum, 1997). These methods are computationally well-suited for solving large systems of equations since they have a complexity of $\mathcal{O}(N)$ for sparse systems[1].

However, lack of robustness and efficiency are widely recognized weaknesses of iterative methods compared to direct methods. For instance, when the spectral radius[2] of the iteration matrix (i.e., the matrix that updates a current approximate solution to the next) is greater than 1, the method diverges. And in general, there is

[1] $\mathcal{O}(N^2)$ if the matrix is dense, still better than direct methods

[2] the largest eigenvalue (in absolute value)

no guarantee regarding the spectral radius. Besides, even if convergent, the algorithm may converge very slowly before reaching a sufficiently accurate solution. A spectral analysis of the convergence of the error reveals that these iterative methods can rapidly solve the high frequencies but have very slow convergence for the lowest frequencies, which hampers their practical use in industrial contexts.

However, several other iterative methods were developed to tackle these issues. Among these, the most famous ones are the Krylov methods. The fundamental concept behind Krylov methods is to seek the solution within a Krylov subspace. Essentially, stationary methods compute an approximate solution within the same Krylov subspace but with "frozen" coefficients. In contrast, Krylov methods aim to determine the optimal coefficients that lead to a much more optimal solution compared to stationary methods, as well as faster convergence. For this reason, the iterative methods of choice for solving large linear systems primarily consist of Krylov methods, the most renowned of which include Conjugate Gradient (CG) for positive definite A matrices, or Bi-Conjugate Gradient Stabilized (BiCGStab), and Generalized Minimal Residual (GMRES) otherwise.

Under certain specific assumptions, these algorithms are robust, implying that they always converge toward the solution of the system. For example, Conjugate Gradient is guaranteed to converge if A is symmetric and positive definite. However, despite their enhanced robustness, these methods might still suffer from slow convergence. In fact, the rate of convergence of Krylov methods depends on the condition number (or conditioning)[3] of A, and as the size of the problem increases, so does the conditioning of A (see Figure 6.8). In conclusion, while Krylov methods provide low complexity (e.g. $\mathcal{O}(N)$ for CG) and robustness, they still face efficiency and scalability challenges.

Preconditioning

The efficiency of Krylov methods can be significantly enhanced through the use of preconditioning. Preconditioning is simply a way of transforming the original linear system into one that has the same solution but is easier to solve with an iterative method. Generally, the reliability of iterative methods depends much more on the choice and quality of the preconditioner than on the specific Krylov method used.

In practice, we aim to find a preconditioning matrix $M \in \mathbb{R}^{N\times N}$, which has the same size as A, and to use a Krylov method to solve the following preconditioned problem:

$$M^{-1}A\mathbf{u} = M^{-1}\mathbf{b} \tag{7.25}$$

[3] the ratio between the largest and lowest eigenvalue of A

The system defined in Equation (7.25) has the same solution as the original system. The rationale is to choose M such that this preconditioned system is considerably more efficient to solve using Krylov methods than the original one. The preconditioned matrix M can be defined in various ways, but it must meet certain minimal requirements. In practice, the main requirement for M is that it is inexpensive to invert, as a preconditioned Krylov method would necessitate computing the solution of a linear system with the matrix M at each iteration. The second requirement is that M should be close to A in some sense and should be nonsingular. In fact, it is advised to choose M such that the spectral radius of $M^{-1}A$ is close to 1, and is smaller than the spectral radius of A. This ensures that the conditioning of the preconditioned system is lower than that of the original system, leading to a faster rate of convergence as well as an enhanced robustness of the Krylov method.

There is no miracle recipe for finding a good preconditioner. For instance, let us define the following fixed-point algorithm:

$$\begin{aligned} \mathbf{u}^{n+1} &= \mathbf{u}^n + M^{-1}\left(\mathbf{b} - A\mathbf{u}^n\right) \\ &= \left(I - M^{-1}A\right)\mathbf{u}^n + M^{-1}\mathbf{b} \end{aligned} \tag{7.26}$$

where $I \in \mathbb{R}^{N\times N}$ is the identity matrix. The stationary algorithm 7.26, when convergent, will converge to the solution of the preconditioned system 7.25. The stationary methods which are based on a splitting of A (like Jacobi or Gauss-Seidel methods mentioned earlier), are, in fact, equivalent to a stationary method on a preconditioned system. As a result, simple preconditioners can be derived from well-established stationary iterative methods. For instance, from the Jacobi method, it is possible to derive a preconditioner $M = D$ where D is the diagonal of A, and, for Gauss-Seidel, the induced preconditioner is the lower triangular part of A. These preconditioners are, however, not much used in practice, but another very successful technique consists, for instance, of choosing as a preconditioner an incomplete LU factorization of A. Besides, in Equation (7.26), $\left(I - M^{-1}A\right)$ is referred to as the iteration matrix. As mentioned earlier, its spectral radius must be lower than 1 to ensure convergence and be as small as possible to provide a fast rate of convergence. This motivates the choice of a preconditioner M such that the spectral radius of $M^{-1}A$ is as close to 1 as possible.

In the latter example, M can be defined explicitly, but it is unlikely that, in practice, M and $M^{-1}A$ can be computed explicitly. Instead, the iterative process, i.e. the Krylov method, can be written such that they operate with A and M^{-1} independently, whenever needed. This is one of the reasons why computing the inverse of M should be inexpensive. There is a difference to make between the rate of convergence (i.e. the number of iterations required to achieve convergence) and the computational time of one iteration of a Krylov method. Using a preconditioner

should enhance the convergence rate of the matrix (because the preconditioned problem has lower conditioning) but be sufficiently fast to apply to avoid increasing too much the resolution time of one step of the Krylov method too. Krylov methods used with a good preconditioner alleviate the issue of robustness and efficiency of iterative methods.

But a last challenge remains to be tackled, which concerns their scalability. Indeed, even though (well-chosen) preconditioners drastically reduce the conditioning of the problem and help the Krylov methods converge faster, the number of iterations required to solve a problem still increases with the size of the problem.

Another crucial point involves the evolution of computer architecture. In the past, improving the performance of numerical methods was only a matter of time, waiting for the next generation of processors. But since 2005, the clock speed of processors stagnated at $2-3$GHz, and the increase in performance was entirely due to the increase in the number of cores per processor[4]: all machines became parallel machines. Enhancing the resolution of large systems no longer relied upon the better performance of the machines but on the development of new parallel algorithms instead.

Scalability and Parallel methods

Nowadays, there exist several parallel iterative frameworks available for solving large systems of equations, as discussed in Wang (2015). In practice, the most common and used method is known as the Multigrid method (Briggs et al., 2000). In brief, the Multigrid method solves large linear systems arising from the discretization of PDEs by using a hierarchy of discretizations, i.e., a hierarchy of the same mesh at different resolutions, ranging from the finest to the coarsest mesh. It achieves the solution through two complementary processes: smoothing and coarse grid correction. Smoothing involves applying a "smoother", which typically consists of a few iterations of a cost-effective method such as Jacobi or Gauss-Seidel (Saad, 2003). On the other hand, coarse grid correction involves transferring the information to a coarser level through restriction, solving a problem on the coarse grid, and then transferring the solution back to the fine grid through interpolation. Smoothing helps reduce high-frequency errors, while coarse grid correction is responsible for eliminating low-frequency errors. Doing so alleviates the issue of slow convergence in stationary iterative methods mentioned earlier. The most renowned Multigrid algorithm is the Algebraic Multigrid Method (AMG) (Ruge and Stüben, 1987). In contrast to classical Multigrid algorithms, which construct the coarse grids based on the problem geometry, AMG relies exclusively on the matrix entries of the linear system. Multigrid methods have optimal complexity, optimal memory requirement, and good scalability. While multigrid methods are not intrinsically parallel,

[4]number of computational resources.

they can be effectively combined with other techniques, such as Domain Decomposition Methods (further introduced), to achieve good parallel efficiency. Although they can be used as iterative solvers, Multigrid methods are primarily employed as preconditioners for Krylov methods, which makes the whole method extensible. In fact, Krylov methods with an AMG preconditioner are today the most widely used iterative framework for solving large systems of equations.

Another well-known fully parallel framework for solving large linear systems is referred to as Domain Decomposition Methods (DDM). The general concept behind DDMs leverages the principle of "divide and conquer": the global problem is partitioned into sub-problems of reasonable size whose resolution can be treated in parallel on multiple processor cores. Similar to multigrid algorithms, many variants of DDM algorithms exist. Still, they can be divided into two classes, depending on whether they require an overlapping decomposition of the domain or not. For instance, the earliest methods, referred to as Schwarz methods, leverage overlapping, whereas more advanced algorithms such as FETI or Neumann-Neumann don't. We refer the reader to Dolean et al. (2015) for an extensive description of these algorithms. These methods can be used as iterative solvers, but similar to multigrid methods, they are mostly used as preconditioners for Krylov methods. Original Domain Decomposition methods are not extensible as the required number of iterations to converge rises with the number of subdomains. This problem is solved in their two-level variant by introducing a coarse space correction. As a result, Krylov methods can be extensible if used with a two-level Domain Decomposition algorithm as a preconditioner.

Contributions

With the rise of Machine Learning, another class of methods has emerged, which can be referred to as statistical methods. These Machine Learning methods typically act as "black-box" approaches, learning to solve physical problems originating from a training distribution of instances. The field of Machine Learning for solving physical problems is a rapidly advancing domain. A state-of-the-art review[5] of existing methods applied to the resolution of fluid dynamics problems is provided in Chapter 3.

The primary advantage of these statistical methods lies in the speed with which they produce approximate solutions. As mentioned earlier, these methods often operate as "black-box" approaches, taking the domain structure and some additional data as inputs and producing an approximation of the solution, without going through the costly computations of creating and solving the system of equations. Machine Learning methods are also particularly fast because they are extremely well-suited for harnessing parallel computations on GPUs. This is in contrast to the

[5] maybe already outdated

traditional methods (i.e. the iterative methods), which use computations on CPUs, whose parallelization on multiple CPU cores can be quickly limited. However, Machine Learning methods can suffer from several issues, such as:

Generalization: Machine Learning models can provide approximate solutions to problems within a specific distribution on which they have been trained, but out-of-distribution samples are often poorly solved.

Level of Accuracy: The precision of the solution of a Machine Learning model is constrained by the accuracy of the trained model: the approximate solution will be as precise as the capabilities of the trained model permit, and it often deteriorates further when handling out-of-distribution samples. The level of accuracy proposed by a Machine Learning model may be sufficient in cases where only a rough idea of the solution is needed. However, when considering the resolution of the Pressure Poisson equation in a CFD problem (see Chapter 1), it becomes imperative that the precision of the induced Pressure Poisson problem is significantly better to ensure the consistency of the other steps and the convergence of the simulation.

In Part II, we introduced three Graph Neural Network-based models designed to address Poisson problems. The initial model, Deep Statistical Solvers (DSS) (Donon et al., 2020), played a foundational role in the development of this thesis. It was studied for its consistency (theoretical properties), enhanced training process (minimization of the discretized residual equation), and the high accuracy of the produced results (see Section 5.3). However, DSS faces challenges in terms of generalization, particularly when dealing with problems involving meshes with a larger number of nodes (refer to Figure 7.7a). Hence, Part II of this work focused on addressing the generalization issue for this specific type of Machine Learning model.

To that end, Chapter 6 and, more importantly, Chapter 7 introduced two contributions to this thesis: the DS-GPS and Ψ-GNN models, respectively. These models were specifically designed as a more robust Machine Learning solution for solving Poisson problems. While DS-GPS is a recurrent architecture, Ψ-GNN leverages the Implicit Layer Theory to automatically determine the number of Message-Passing steps required to achieve convergence. Additionally, significant effort has been put into the development of these models to construct architectures that are less "black-box" and more respectful of physical constraints, such as explicitly treating boundary conditions. Furthermore, the analogy between traditional iterative methods and the GNN models developed in Part II is direct: all three of these methods can be seen as stationary iterative methods, with the M_θ function (see Figure 7.1) being the iteration matrix. The convergence guarantees of Ψ-GNN, obtained by constraining the spectral radius of M_θ through minimization of its Frobenius norm (recall that post-training, the spectral radius equals approximately 0.98 - refer to Section 9.3.3), are actually analogous to the theoretical properties that an iteration matrix must fulfill in a stationary method (i.e., spectral radius less than 1). As a result, these contribu-

tions allow for great generalization, enabling the accurate resolution of problems involving up to ten thousand nodes. However, their scalability becomes a challenge when addressing problems on very large meshes ($100,000$ nodes or more). Additionally, these models display an MSE with an exact solution of no higher than 10^{-2}, a precision that may not ensure the convergence of a splitting method, for instance.

The final part of this thesis is dedicated to addressing the second issue, which involves the level of accuracy of the solutions of Machine Learning models. Additionally, we seek to scale up the capabilities of these models for solving Poisson problems on meshes with a very large number of nodes. To achieve this, we propose to construct a hybrid solver, which combines GNN models with Schwarz methods arising from the field of Domain Decomposition (DDM). We explore the application of this hybrid solver in two frameworks: as a stationary iterative method, and as a preconditioner for a Krylov method. The objective is to leverage GNN models to solve the multiple sub-problems in the context of a Schwarz method, with three goals: i) enhancing the capabilities of GNN models to handle large meshes becomes feasible, as it allows the selection of a sub-problem size aligned with the best capabilities of the GNN models; ii) using a two-level variant of a Schwarz method addresses the scalability issue of GNN models, where the number of Message-Passing required for convergence grows as the size of the problem increases; iii) harnessing the "per-batch" structure of GNN models to solve multiple sub-problems in parallel on GPUs, with the hope to speed up the process. In our best-proposed framework, this hybrid method is used as a preconditioner for Krylov methods. As a result, the proposed approach converges to the solution with any desired precision, thanks to the Krylov method. The robustness and efficiency of the Krylov method are significantly enhanced by using a fully GNN-based preconditioner. This GNN-based preconditioner, whose architecture leverages a two-level DDM approach, is capable of handling meshes at a very large scale and making the Krylov method extensible. Moreover, its parallel execution on GPUs ensures its fast application within the Krylov method.

In the following, Chapter 8 provides an introduction to Schwarz methods, and Chapter 9 presents the proposed hybrid methods, which uses knowledge from both the GNN models in Part II and Schwarz methods in Chapter 8.

8 - Introduction to Schwarz methods

Sommaire

This chapter aims to provide background information on Domain Decomposition Methods. While numerous techniques have been developed to enhance such methods, this thesis specifically relies on one of the earliest variants: Schwarz methods. The first section, Section 8.1, introduces three continuous formulations of Schwarz methods.

Subsequently, in Section 8.2, detailed insights into the computational aspects associated with these methods are provided. The following two sections address the use of Schwarz methods: as iterative solvers in Section 8.3 and as preconditioners for Krylov methods in Section 8.4. In Section 8.5, the two-level methods are introduced to achieve scalability with respect to the number of subdomains. All presented results are generated using our self-developed Python code. In the context of this introduction, all methods are presented to tackle Poisson problems with Dirichlet boundary conditions, similar to (4.1).

8.1 . Overview of Schwarz methods

8.1.1 . Original Schwarz method

The original Schwarz algorithm was first proposed by a German analyst named *Hermann Schwarz* in the 19th century. At that time, solutions to Poisson problems were only possible for very simple geometries through the use of the Fourier transform, since numerical methods were not yet available. To further complicate the problem,

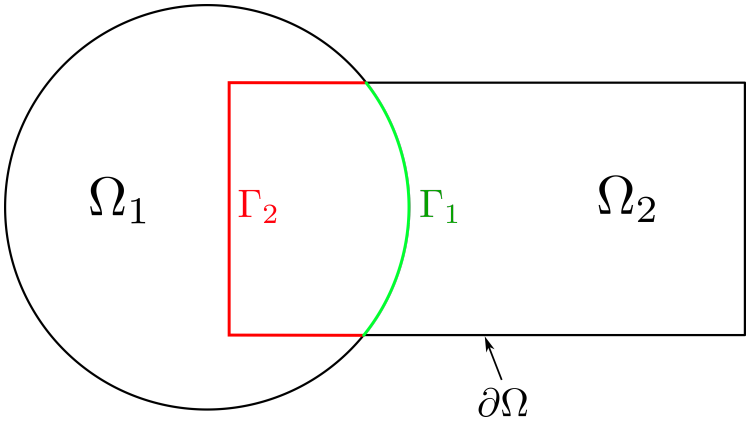


Figure 8.1: Illustration of the geometry used by H.Schwarz when introducing his Alternating Method. The full domain Ω with boundary $\partial\Omega$ is composed of two simple shapes: a disk Ω_1 and a rectangle Ω_2 with interfaces $\Gamma_1 = \partial\Omega_1 \cap \Omega_2$ in green, and $\Gamma_2 = \partial\Omega_2 \cap \Omega_1$ in red.

H. Schwarz extended his considerations to encompass more complicated geometries, such as compositions of multiple simple shapes (e.g. a union of a circle and a rectangle, as depicted in Figure 8.1). To address this challenge, H. Schwarz proposed an iterative algorithm that consists of solving a Poisson equation for each simple geometry in every iteration and subsequently transferring information across the interfaces between these domains (Schwarz, 1870).

Formally, let Ω be the union of a disk Ω_1 and a rectangle Ω_2, with interfaces $\Gamma_1 = \partial\Omega_1 \cap \Omega_2$ and $\Gamma_2 = \partial\Omega_2 \cap \Omega_1$, as illustrated in Figure 8.1. Let us consider the resolution of a Poisson problem with Dirichlet boundary conditions, which consists in finding a real-valued function u, defined on Ω, that satisfies:

$$\begin{cases} -\Delta u &= f & \in \Omega \\ u &= g & \in \partial\Omega \end{cases} \tag{8.1}$$

To solve Equation (8.1), H. Schwarz invented the *Alternating Schwarz Method*, an iterative method that uses solutions on both the disk, denoted as u_1, and the rectangle, denoted as u_2. The method starts with an initial guess u_2^0 along Γ_1 and iteratively computes the solutions u_1^{n+1} and u_2^{n+1} according to the following algorithm:

$$\begin{cases} -\Delta(u_1^{n+1}) &= f & \text{in } \Omega_1 \\ u_1^{n+1} &= g & \text{on } \partial\Omega_1 \cap \partial\Omega \\ u_1^{n+1} &= u_2^n & \text{on } \Gamma_1 \end{cases} \qquad \begin{cases} -\Delta(u_2^{n+1}) &= f & \text{in } \Omega_2 \\ u_2^{n+1} &= g & \text{on } \partial\Omega_2 \cap \partial\Omega \\ u_2^{n+1} &= u_1^{n+1} & \text{on } \Gamma_2 \end{cases} \tag{8.2}$$

The particularity of this algorithm lies in the exchange of information along the interfaces Γ_1 and Γ_2 at each iteration, highlighted in green and red in (8.2), respec-

tively. Additionally, H. Schwarz demonstrated the convergence of this approach, establishing the well-posedness of the Poisson problem in complex geometries.

With the advancements in computing technology, these methods have gathered practical interest both as iterative solvers (refer to Section 8.3) and as preconditioners for Krylov solvers (refer to Section 8.4). Besides, with the rise of parallel computing, a slight modification of the original algorithm (8.2) has made it compatible with modern parallel architectures. In the following sections, three continuous Schwarz methods will be briefly surveyed: the Jacobi-Schwarz method (JSM), the Restricted Additive Schwarz (RAS) method, and the Additive Schwarz method (ASM). These methods represent extensions of the original Alternating Schwarz method and are introduced here to address problems with multiple subdomains. While JSM and RAS methods are theoretically equivalent (ASM being a variant of RAS), they differ in their local problem-solving approaches, making them applicable in various contexts (refer to the subsequent Chapter 9).

8.1.2 . Jacobi-Schwarz method

Let Ω be an open and bounded domain and let us consider an overlapping decomposition of Ω into K open subdomains $(\Omega_i)_{1\leq i\leq K}$ such that:

$$\Omega = \bigcup_{i=1}^{K} \Omega_i$$

The alternating Schwarz algorithm defined in Equation (8.2) operates on local functions defined within each subdomain Ω_i. However, when formulating algorithms that work with global functions (i.e. functions defined across the entire domain Ω), one needs to define operators able to bridge the gap between the subdomains and the global domain. This role is fulfilled by *Extension* operators and *Partition of Unity* functions, which are defined as follows:

Definition 8.1.1 (Extension Operators & Partition of Unity functions)
Let $(\Omega_i), i \in [1, K]$ be an overlapping partition of some domaine Ω.The Extension operator E_i maps functions defined on Ω_i to their extension on Ω that takes value 0 outside Ω_i. Let us also define Partition of Unity functions $\chi_i : \Omega_i \to \mathbb{R}$, $\chi_i \geq 0$ and $\chi_i(x) = 0$ for $x \in \partial\Omega_i$ and such that, for any function $w : \Omega \to \mathbb{R}$:

$$w = \sum_{i=1}^{K} E_i(\chi_i w_{|\Omega_i})$$

E_i being the Extension Operator for partition (Ω_i).

Consider the resolution of the Poisson problem (8.1) withinΩ. The Jacobi-Schwarz method, described in Algorithm 2, is an extension of the original Schwarz method

Algorithm 2 Jacobi-Schwarz Method (JSM)

Let u^n be an approximate solution to the Poisson problem (8.1). The updated approximate solution u^{n+1} is computed as follows:

1. Solve for all subproblems $i = 1, \dots, K$:

$$\begin{cases} -\Delta(w_i)^{n+1} &= f & \text{in } \Omega_i \\ w_i^{n+1} &= g & \text{on } \partial\Omega_i \cap \partial\Omega \\ w_i^{n+1} &= u^n & \text{on } \partial\Omega_i \backslash \partial\Omega \end{cases} \tag{8.3}$$

2. Glue them together:

$$u^{n+1} = \sum_{i=0}^{K} E_i(\chi_i w_i^{n+1})$$

(8.2). This approach, which operates on the global function instead of local ones, offers the advantage of complete parallelizability.

8.1.3 . Restricted Additive & Additive Schwarz methods

Algorithm 3 introduces the Restricted Additive Schwarz (RAS) and Additive Schwarz (ASM) methods, two alternative formulations of the JSM Algorithm. These new formulations are defined in terms of the continuous residual $r = f + \Delta u$. All three algorithms are fundamentally parallel. Furthermore, JSM and RAS have been proven to be equivalent, as demonstrated in Dolean et al. (2015). Note that the only difference between the RAS and ASM methods is that ASM does not rely on Partition of Unity functions.

While the Jacobi-Schwarz method is scarcely implemented due to the duplication of unknowns it involves, the RAS algorithm (Cai and Sarkis, 1999) is the default parallel solver in the well-known C++ PETSC package[1], and the ASM offers a plethora of theoretical results, as discussed in Toselli and Widlund (2004). For further information, an in-depth history of Schwarz methods can be found in Gander et al. (2008).

8.2 . Discrete formulations

Previous Section 8.1 presented the continuous formulation of three Domain Decomposition Methods: the Jacobi-Schwarz method (JSM), the Restricted Additive Schwarz method (RAS), and the Additive Schwarz method (ASM). However, to implement these algorithms and compute approximate solutions, discrete counterparts

[1] https://petsc.org/release/

Algorithm 3 Restricted Additive (RAS) & Additive Schwarz Method (ASM)

Let u^n be an approximate solution to the Poisson problem (8.1). The updated approximate solution u^{n+1} is computed as follows:

1. Compute the residual at iteration n:

$$r^n = f + \Delta u^n \tag{8.4}$$

2. For $i = 1, \ldots, K$, solve for local corrections:

$$\begin{cases} -\Delta(v_i)^n &= r^n \quad \text{in } \Omega_i \\ v_i^n &= 0 \quad \text{on } \partial\Omega_i \end{cases} \tag{8.5}$$

3. Compute an average of the local corrections and update u^n:
   - Restricted Additive Schwarz (RAS):

$$u^{n+1} = u^n + \sum_{i=1}^{K} E_i(\chi_i v_i^n) \tag{8.6}$$

   - Additive Schwarz Method (ASM):

$$u^{n+1} = u^n + \sum_{i=1}^{K} E_i(v_i^n) \tag{8.7}$$

for these methods must be defined.

The key elements introduced at the continuous level are as follows:

- An open and bounded domain Ω.
- A decomposition of Ω into K overlapping open subdomains $(\Omega_i)_{1 \leq i \leq K}$ such that:
$$\Omega = \bigcup_{i=1}^{K} \Omega_i$$
- A solution function $u : \Omega \to \mathbb{R}$.
- Extension operators $(E_i)_{1 \leq i \leq K}$, which extend a function defined on a local subdomain to the global domain by zero, as defined in Definition 8.1.1.
- Partition of Unity functions $(\chi_i)_{1 \leq i \leq K}$ defined in Definition 8.1.1, which satisfy,

for all global functions $u : \Omega \to \mathbb{R}$:

$$u = \sum_{i=1}^{K} E_i(\chi_i u_{|\Omega_i})$$

At the discrete level, it is possible to provide similar definitions:

- A mesh Ω_h which defines a set of degrees of freedom $\mathcal{N}$.
- An overlapping decomposition into K subsets $(\mathcal{N}_i)_{1 \leq i \leq K}$ such that:

$$\mathcal{N} = \bigcup_{i=1}^{K} \mathcal{N}_i$$

  The total number of degrees of freedom of $\mathcal{N}$ is referred to as N and, for each subset $1 \leq i \leq K$, we denote as k_i the number of degrees of freedom of the subset $\mathcal{N}_i$. Note that the subset of indices $\mathcal{N}_i$ also defines a sub-mesh $\Omega_{h,i}$ of the mesh Ω_h.
- A solution vector $\mathbf{u} \in \mathbb{R}^N$.
- The restriction of a vector $\mathbf{u} \in \mathbb{R}^N$ to a subset $\mathcal{N}_i$ can be expressed as $R_i\mathbf{u}$ where R_i is a rectangular boolean matrix of size $k_i \times N$. The extension operator is defined as the transpose matrix R_i^T.
- At the discrete level, Partition of Unity functions correspond to diagonal matrices $(D_i)_{1 \leq i \leq K}$ of size $k_i \times k_i$, with non-negative entries such that, for all vector $\mathbf{u} \in \mathbb{R}^N$, the following holds:

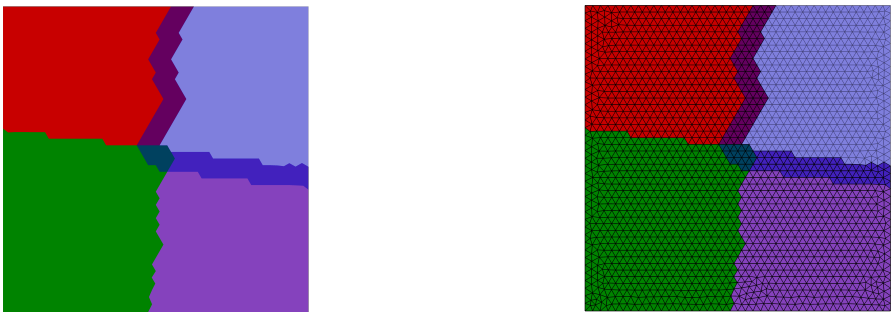
$$\mathbf{u} = \sum_{i=1}^{K} R_i^T D_i R_i \mathbf{u} \tag{8.8}$$

  or in other words:

$$I_d = \sum_{i=1}^{K} R_i^T D_i R_i \tag{8.9}$$

  where $I_d \in \mathbb{R}^{N \times N}$ is the identity matrix.

In this thesis, the set of indices $\mathcal{N}$ is partitioned using METIS (Karypis and Kumar, 1997), an automatic graph partitioner. In this algorithm, the distribution of indices ensures that the number of elements assigned to each partition is approximately equal while also minimizing the number of adjacent elements assigned to different partitions. The goal of the first condition is to balance the computations among



(a) *(b)*

Figure 8.2: 8.2a: continuous square domain divided into 4 overlapping subdomains. 8.2b; its discretization into an unstructured triangular mesh with 2314 nodes. The mesh is decomposed into 4 subdomains with an overlap of 2 using METIS partitioner. The overlap is depicted for the blue subdomain.

the partitions. The goal of the second condition is to minimize the communication resulting from the placement of adjacent elements to different partitions.

An important theoretical property states that Schwarz methods require an overlapping decomposition to guarantee convergence (Toselli and Widlund, 2004). In this context, careful design of the Partition of Unity matrices D_i is essential. For instance, when no overlapping is considered, and R_i represents the restriction matrix from the set $\mathcal{N}$ to a subset $\mathcal{N}_i$, choosing D_i as the identity matrix of size $k_i \times k_i$ satisfies relation (8.9). Given the necessity of an overlapping decomposition, let us now examine the scenario where each subset $\mathcal{N}_i$ is expanded to enclose its direct neighbours, resulting in $\mathcal{N}_i^{\delta=1}$. In this case, let R_i denote the restriction matrix from the set $\mathcal{N}$ to the subset $\mathcal{N}_i^{\delta=1}$. To maintain relation (8.9), D_i must be appropriately configured. One potential approach is to introduce, for all $j \in \mathcal{N}$, the set of subsets that contain j as an element:

$$\mathcal{M}_j = \{i \in [1, K] \text{ such that } j \in \mathcal{N}_i^{\delta=1}\}$$

and then define D_i as:

$$(D_i)_{jj} = \frac{1}{\text{Card}(M_j)} \qquad \forall j \in \mathcal{N}_i^{\delta=1}$$

Figure 8.2 illustrates a mesh of a square domain and its decomposition into 4 submeshes with an overlap of 2.

8.3 . Schwarz methods as iterative solvers

To solve Poisson problems like (8.1), Schwarz methods can be used in several ways. The most straightforward approach involves using them as iterative solvers. Building upon the notations introduced in previous Section 8.2, the continuous domain Ω is discretized into an unstructured mesh Ω_h with N nodes, following the process depicted in Figure 4.3. The Poisson PDE (8.1) is then discretized using the Finite Element Method with first-order finite elements[2] (as discussed in Section 1.3), yielding a linear system of the form $A\mathbf{u} = \mathbf{b}$. With these foundations, it is possible to define the following iterative Schwarz methods:

Definition 8.3.1 (Restricted Additive Schwarz (RAS) algorithm)
The iterative RAS algorithm is the preconditioned fixed point iteration defined by:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + M_{RAS}^{-1}\,\mathbf{r}^n \quad \textit{with} \quad \mathbf{r}^n := \mathbf{b} - A\mathbf{u}^n \tag{8.10}$$

where M_{RAS}^{-1} is called the RAS preconditioner and is defined by :

$$M_{RAS}^{-1} = \sum_{i=1}^{K} R_i^T D_i (R_i A R_i^T)^{-1} R_i \tag{8.11}$$

Definition 8.3.2 (Additive Schwarz Method (ASM) algorithm)
The iterative ASM algorithm is the preconditioned fixed point iteration defined by:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + M_{ASM}^{-1}\,\mathbf{r}^n \quad \textit{with} \quad \mathbf{r}^n := \mathbf{b} - A\mathbf{u}^n \tag{8.12}$$

where M_{ASM}^{-1} is called the ASM preconditioner and is defined by :

$$M_{ASM}^{-1} = \sum_{i=1}^{K} R_i^T (R_i A R_i^T)^{-1} R_i \tag{8.13}$$

Numerous theoretical properties have been proved from the convergence analysis of these methods, as discussed in Gander et al. (2008). Among these properties, one stands out as particularly crucial: the size of the overlap between subdomains. Using Fourier analysis, it is possible to show that the high-frequency components of the error are damped very fast due to the presence of overlap. However, addressing the low-frequency components requires specialized treatment, such as considering a coarse grid correction (as explored in Section 8.5). For the Schwarz methods presented here, i.e., considering an overlapping decomposition and interface communication with Dirichlet conditions, zero overlap prevents these methods from converging, and convergence accelerates with an increasing ratio of overlap size to

[2] can also work with higher-order finite elements.

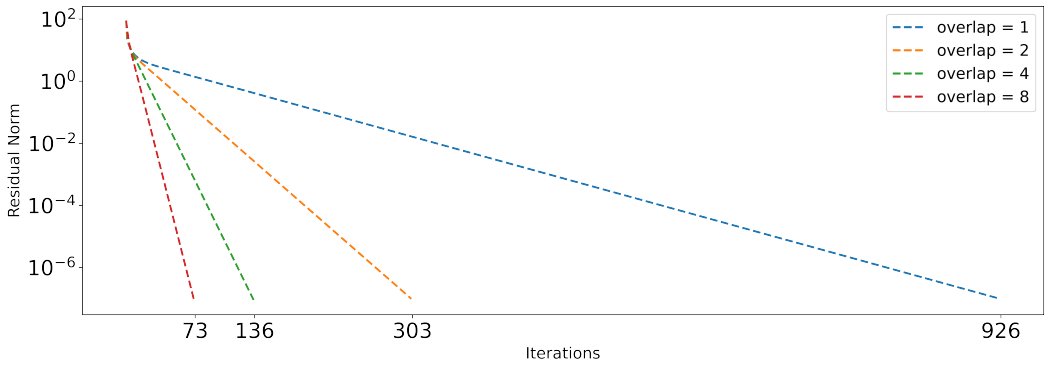


Figure 8.3: Resolution of the same Poisson problem in a square domain divided into 4 subdomains using RAS algorithm. The Figure displays the iteration count to achieve convergence to 10^{-7} residual with respect to different sizes of overlap.

subdomain size. However, this assumption is not always true for more advanced algorithms[3] such as the Optimized Schwarz methods, which leverage Robin interface conditions (Lions, 1990) for instance.

Schwarz methods are not frequently used as iterative solvers because their convergence to a sufficiently accurate solution is very slow, although it improves when the overlap is larger. Besides, it is obvious that the ASM method does not converge when used as an iterative solver due to the absence of Partition of Unity functions. Interestingly, this absence causes the ASM operator to be symmetric, in contrast to the RAS operator. While this property might seem like a limitation, it turns out to be advantageous when ASM is used as a preconditioner for Krylov methods, a topic explored in the subsequent Section 8.4.

To further illustrate this, consider the following experiment: the same Poisson problem is solved in a square domain divided into 4 subdomains, similar to Figure 8.2b. For each experiment, the size of overlap is 1, 2, 4, and 8. The RAS algorithm is used as an iterative solver and is considered converged when the norm of the residual reaches an order of 10^{-7}. Figure 8.3, which illustrates the number of iterations required to achieve convergence considering the different sizes of overlap, clearly shows that the larger the overlap, the faster the convergence.

8.4 . Schwarz methods as preconditioners

As mentioned in the introductory section of this part, the lack of robustness and efficiency are widely recognized weaknesses of stationary iterative methods, such as those defined in Equations (8.10), or (8.12). Fortunately, other iterative methods have been developed to enhance efficiency and accelerate the convergence of these

[3] outside the scope of this work.

Algorithm 4 Preconditioned Conjugate Gradient

Compute $\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0$, $\mathbf{z}_0 = M^{-1}\mathbf{r}_0$, $\mathbf{p}_0 = z_0$
for $i = 0, 1, \ldots$ **do**
  $\rho_i = \langle \mathbf{r}_i, \mathbf{z}_i \rangle$
  $\mathbf{q}_i = A\mathbf{p}_i$, $\alpha_i = \dfrac{\rho_i}{\langle \mathbf{p}_i, \mathbf{q}_i \rangle}$
  $\mathbf{u}_{i+1} = \mathbf{u}_i + \alpha_i \mathbf{p}_i$
  $\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i \mathbf{q}_i$
  if $\|\mathbf{r}_{i+1}\| <$ tol **then**
    Break;
  end if
  $\mathbf{z}_{i+1} = M^{-1}\mathbf{r}_{i+1}$, $\rho_{i+1} = \langle \mathbf{r}_{i+1}, \mathbf{z}_{i+1} \rangle$, $\beta_{i+1} = \dfrac{\rho_{i+1}}{\rho_i}$
  $\mathbf{p}_{i+1} = \mathbf{z}_{i+1} + \beta_{i+1}\mathbf{p}_i$
end for

stationary methods. Among these methods, the most famous ones are called Krylov methods, which have given rise to renowned algorithms such as Conjugate Gradient (CG) or Bi-Conjugate Gradient Stabilized (BiCGStab). However, despite providing enhanced robustness and efficiency compared to stationary iterative methods, they still face challenges as the size of the problem grows, limiting their practical use in an industrial context. To improve their performance, extended Krylov methods with preconditioning have been developed, resulting in even more efficient algorithms, that do not require the explicit matrix form of the operators A or the preconditioner M. Instead, they access the matrix by evaluating matrix-vector products, similar to the stationary method. In this context, the RAS and ASM preconditioners (i.e., M_{RAS}^{-1} and M_{ASM}^{-1}) emerge as solid candidates.

Within the scope of this thesis, two preconditioned Krylov methods have been considered. The first, known as Preconditioned Conjugate Gradient (PCG), is detailed in Algorithm (4). PCG proves to be highly effective, delivering rapid convergence and straightforward implementation. However, it is exclusively applicable when the problem is symmetric (A must be symmetric and positive-definite). Therefore, the ASM preconditioner becomes essential due to its symmetric nature, which is in contrast to the RAS algorithm. Although ASM cannot serve as an independent iterative solver, it works remarkably well as a preconditioner, especially suited for symmetric algorithms like PCG. The second algorithm extends PCG to handle non-symmetric problems and is referred to as the Preconditioned Bi-Conjugate Gradient Stabilized method (PBiCGStab), outlined in Algorithm 5. This algorithm can effectively use both ASM and RAS preconditioners.

To further illustrate this, consider the resolution of a Poisson problem on a mesh representing a square domain. The mesh is divided into 4 subdomains with an

Algorithm 5 Preconditioned Bi-Conjugate Gradient Stabilized

Compute $\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0$, $\mathbf{z}_0 = M^{-1}\mathbf{r}_0$, $\mathbf{p}_0 = \mathbf{z}_0$, $\widetilde{\mathbf{r}} = \mathbf{r}_0$
for $i = 1, 2, \ldots$ **do**
  $\rho_{i-1} = \langle \widetilde{\mathbf{r}}, \mathbf{r}_i \rangle$
  if $\rho_{i-1} = 0$ **then**
    Method fails
  end if
  if $i = 1$ **then**
    $\mathbf{p}_i = \mathbf{r}_0$
  else
    $\beta_{i-1} = \frac{\rho_{i-1}}{\rho_{i-2}} \frac{\alpha_{i-1}}{\omega_{i-1}}$
    $\mathbf{p}_i = \mathbf{r}_{i-1} + \beta_{i-1}(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$
  end if
  $\hat{\mathbf{p}} = M^{-1}\mathbf{p}_i$, $\mathbf{v}_i = A\hat{\mathbf{p}}$, $\alpha_i = \frac{\rho_{i-1}}{\langle \widetilde{\mathbf{r}}, \mathbf{v}_i \rangle}$
  $\mathbf{s} = \mathbf{r}_{i-1} - \alpha_i \mathbf{v}_i$
  if $\|\mathbf{s}\| <$ tol **then**
    $\mathbf{u}_i = \mathbf{u}_{i-1} + \alpha_i \hat{\mathbf{p}}$
    Break;
  end if
  $\hat{\mathbf{s}} = M^{-1}\mathbf{s}$, $\mathbf{t} = A\hat{\mathbf{s}}$, $\omega_i = \frac{\langle \mathbf{t}, \mathbf{s} \rangle}{\mathbf{t}, \mathbf{t}}$
  $\mathbf{u}_i = \mathbf{u}_{i-1} + \alpha_i \hat{\mathbf{p}} + \omega_i \hat{\mathbf{s}}$
  $\mathbf{r}_i = \mathbf{s} - \omega_i \mathbf{t}$
  if $\|\mathbf{r}_i\| <$ tol **then**
    Break;
  end if
end for

overlap of 2, similar to Figure 8.2b. Three methods are used to solve this problem: the iterative RAS algorithm, the Conjugate Gradient method, and the ASM-Preconditioned Conjugate Gradient method. All three methods are considered convergent when the norm of the residual reaches an order of magnitude of 10^{-7}. Figure 8.4 displays the iteration count required to achieve convergence for the three methods. Not surprisingly, the Conjugate Gradient method outperforms the RAS iterative solver with 117 iterations instead of 303. More importantly, it is worth noting that the ASM preconditioner significantly enhances the Conjugate Gradient method, requiring only 25 iterations to achieve convergence.

Furthermore, consider another experiment designed to assess the performance of Krylov methods preconditioned with Schwarz method, regarding the resolution of a Poisson problem on a square mesh using the Conjugate Gradient method precon-

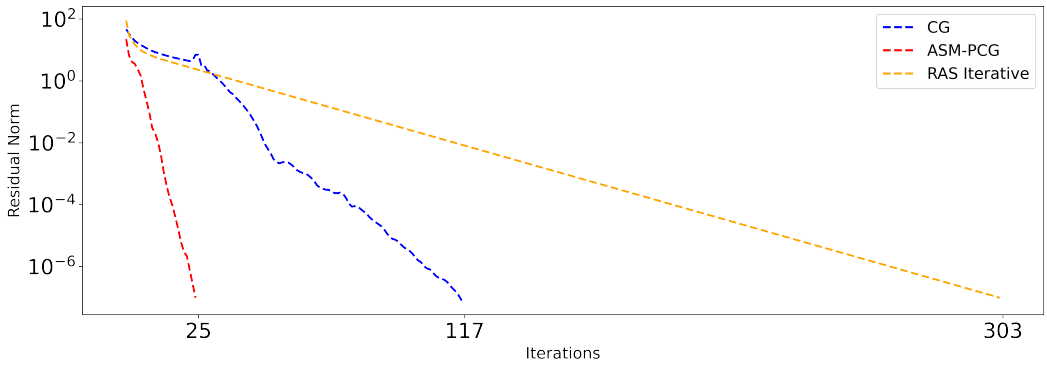


Figure 8.4: Resolution of a Poisson problem in a square mesh divided into 4 subdomains with an overlap of 2 using three methods: the RAS algorithm as an iterative solver, the original Conjugate Gradient method (CG) and the ASM-Preconditioned Conjugate Gradient (ASM-PCG). The Figure displays the iteration count to achieve convergence with respect to the different methods.

ditioned with ASM. This time, the mesh is partitioned into an increasing number of submeshes: 4, 16, 36, and 64. Whatever the configuration, the size of the overlap is set to 2. The number of points within each submesh remains constant across all experiments, resulting in a progressively larger square (additional details are available in Section 8.5). The method is considered convergent when the norm of the residual reaches an order of magnitude of 10^{-7}. Figure 8.5 displays the iteration count required to achieve convergence relative to the number of subdomains in the mesh. It demonstrates that while the method is effective, it necessitates a greater number of iterations as the number of subdomains increases, which highlights that Krylov methods, preconditioned with Schwarz methods, do not scale well with respect to the number of subdomains.

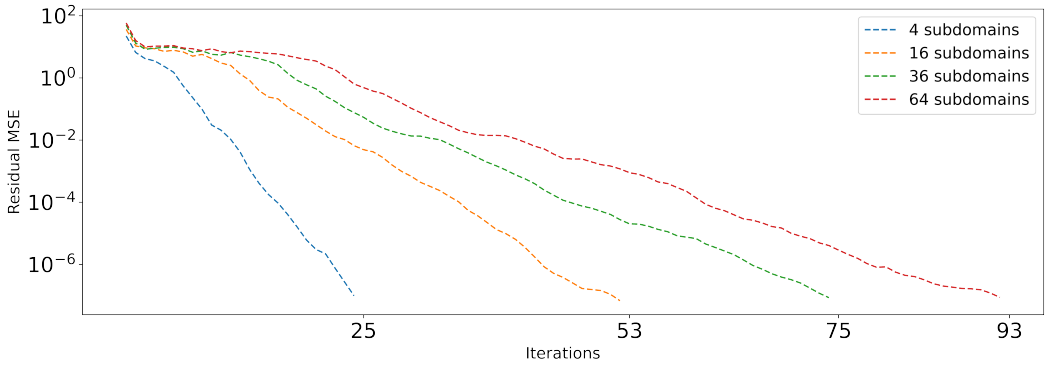


Figure 8.5: Resolution of a Poisson problem in a square mesh using the ASM-Preconditioned Conjugate Gradient (ASM-PCG). The mesh is divided into a growing number of submeshes: 4, 16, 36 and 64. The Figure displays the iteration count to achieve convergence with respect to the growing number of subdomains.

8.5 . Two-level methods

Results obtained in previous sections indicate that the performances of Krylov methods preconditioned with Schwarz methods decrease strongly when the number of subdomains increases (i.e. they are not scalable in the weak sense). This challenge can be effectively addressed through the use of a two-level method. In this approach, Schwarz methods as defined in (8.3.1), and (8.3.2) are augmented by solving a coarse problem, whose size is of the order of magnitude of the number of subdomains.

When dealing with a large number of subdomains, a convergence plateau is clearly visible in the performance of Schwarz methods (Figure 8.5). The primary issue with the one-level method arises from the absence of a global exchange of information since it propagates only among directly neighbouring subdomains. The length of this plateau is thus directly linked to the number of subdomains in a single direction until the information spans the entire domain.

In this thesis, the focus will be on addressing the scalability property of these methods through the introduction of a coarse space method. One mechanism to enhance scalability involves implementing a two-level preconditioner with a coarse space correction. In two-level methods, the smaller (first level) problem connects all subdomains during each iteration, facilitating the exchange of information.

From a theoretical perspective, the plateau observed in one-level methods corresponds to a few low eigenvalues within the spectrum of the preconditioned problem. While Schwarz preconditioners remove the impact of very large eigenvalues associated with high-frequency modes, the presence of small eigenvalues hinders convergence. These small eigenvalues are linked to low-frequency modes that relate to specific global information. This global information can be efficiently handled using a coarse space correction. In the case of a Poisson problem, these slow modes correspond to constant functions in the kernel of the Laplace operator (Dolean et al., 2015).

Define $Z \in R^{N \times K}$ as a rectangular matrix whose columns correspond to these slow modes. Here, N represents the number of nodes in the mesh, and K denotes the total number of submeshes. In scenarii where A is semi-positive definite, the foundation for developing a coarse space correction (as outlined in Mandel* and Sousedík (2010)) involves finding the optimal correction to an approximate solution $\mathbf{u}$ using a vector $Z\beta$ within the vector space spanned by the columns of Z. This minimization problem can be defined as follows:

$$\min_{\beta} \|A\left(\mathbf{u} + Z\beta\right) - \mathbf{b}\| \tag{8.14}$$

and whose solution is:

$$Z\beta = (Z^T AZ)^{-1} Z^T (\mathbf{b} - A\mathbf{u}) = (Z^T AZ)^{-1} Z^T \mathbf{r} \tag{8.15}$$

When dealing with Poisson problems, the creation of matrix Z can be achieved using Nicolaides coarse space, as outlined in (Nicolaides, 1987). In this approach, Z is defined by vectors that have local support in the subdomains and such that the constant function $\mathbf{1}$ belongs to the vector space spanned by the columns of Z. In line with the notations introduced in Section 8.2, the Nicolaides coarse space matrix Z is defined in the following manner:

$$Z = \begin{bmatrix} D_1 R_1 \mathbf{1} & 0 & \cdots & 0 \\ 0 & D_2 R_2 \mathbf{1} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & D_K R_K \mathbf{1} \end{bmatrix} \tag{8.16}$$

By reformulating as $R_0 = Z^T$, the two-level RAS and ASM preconditioners are:

Definition 8.5.1 (Two-level RAS)
The two-level Restricted Additive Schwarz preconditioner is defined as:

$$M^{-1}_{RAS,2} = R_0^T (R_0 A R_0^T)^{-1} R_0 + \sum_{k=1}^{K} R_i^T D_i (R_i A R_i^T)^{-1} R_i \tag{8.17}$$

Definition 8.5.2 (Two-level ASM)
The two-level Additive Schwarz preconditioner is defined as:

$$M^{-1}_{ASM,2} = R_0^T (R_0 A R_0^T)^{-1} R_0 + \sum_{k=1}^{K} R_i^T (R_i A R_i^T)^{-1} R_i \tag{8.18}$$

Several observations can be made. The first relates to the structure of the preconditioner, which remains consistent with the one-level method. However, in contrast to the one-level approach where only local subproblems need to be solved in parallel, the two-level method introduces the resolution of a global linear system, denoted as $R_0 A R_0^T$, referred to as the coarse problem. This coarse problem effectively couples all subproblems during each iteration. Despite its global nature, the matrix size of this coarse problem is small (i.e., $K \times K$), rendering the additional computational cost negligible when compared to the resulting benefits. This holds true provided that the number of subdomains does not grow excessively large.

To provide further illustration, let us revisit the results obtained in the previous example, where we used the one-level ASM method as a preconditioner. Let us use

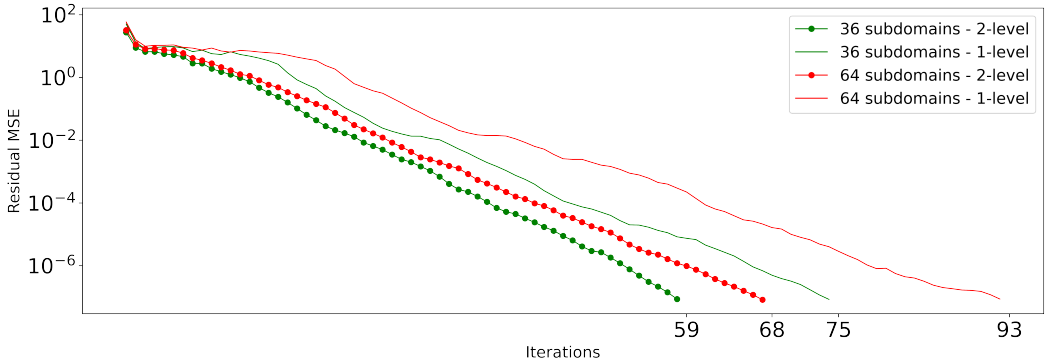


Figure 8.6: Resolution of a Poisson problem in a square mesh using the one and two-level ASM methods as preconditioners for the Conjugate Gradient method. Two configurations are considered: dividing the mesh into 36 and 64 submeshes. The Figure displays the iteration count to achieve convergence with respect to the different mesh configurations and methods used.

the two-level method to compare the results for the configurations involving mesh subdivisions into 36 and 64 submeshes. Figure 8.6 illustrates the iteration counts required to achieve convergence. Both the one- and two-level ASM preconditioners for the Conjugate Gradient method are used, considering both the 36 and 64 subdomain configurations. This experimentation highlights the efficiency of the two-level approach, displaying faster convergence compared to the one-level method. Notably, the two-level approach effectively overcomes the observed plateau.

In this Chapter, we have set up the scene for Schwarz methods, which belong to the field of Domain Decomposition methods and are used to solve linear systems such as 4.3. We introduced three formulations of Schwarz methods: the Jacobi-Schwarz (JS), the Restricted Additive (RAS), and the Additive Schwarz (ASM) methods. We demonstrated that all three methods can be used as iterative solvers. However, when used in this manner, these methods are observed to converge slowly. Nonetheless, RAS and ASM, thanks to their algebraic form, can serve as good preconditioners for Krylov methods, significantly enhancing convergence. It is worth noting that the Krylov method, even preconditioned with Schwarz methods, scales poorly with the number of subdomains. To address this issue, Schwarz methods can be enhanced with a two-level method, making the entire framework extensible. In the next chapter, we present our contributions, which involve the development of hybrid methods, by combining the GNN models developed in II with Schwarz methods.

9 - Hybrid Solvers for Large-Scale Problem Solving

Sommaire

9.1 . Introduction

This chapter explores the construction of hybrid models by combining GNN-based models presented in Part II with Schwarz methods described in previous Chapter 8. The main objective in developing such hybrid approaches is to leverage GNN models to solve multiple Poisson sub-problems within the context of a Domain Decomposition method. As highlighted in the introductory section of this part, these hybrid methods aim to address the challenges that emerge when using GNN-based models to solve large-scale Poisson problems, i.e. those involving meshes with a significantly large number of nodes. These challenges are reminded as follows:

1. In large-scale scenarios, GNN-based models may encounter challenges such as a loss of accuracy and memory issues. With this hybrid approach, the models are no longer used to solve the entire problem; instead, they are used for resolving sub-problems within a Domain Decomposition framework. This enables the selection of a sub-problem size aligned with the optimal capabilities of the GNN-based models.

2. The GNN models require an increasing number of Message-Passing steps as the number of nodes in a mesh increases (see Figure 7.7b). This limitation can be addressed by combining them with a two-level variant of the Schwarz method.

3. The accuracy of the solutions obtained by the GNN models is limited by the capabilities of the trained model. This precision is usually sufficient if only a rough solution is required but can be inadequate to ensure the convergence of other steps in a splitting scheme method, for instance. In the last section of this chapter, we propose using this hybrid method as a preconditioner for a Krylov method, thereby enabling the convergence of the whole algorithm to any desired precision, whatever the precision of the solutions derived from the GNN.

4. Using the GNN models as solvers for multiple subdomains allows harnessing the "per-batch" structure inherent in Machine Learning models. This enables the parallel resolution of multiple sub-problems on GPUs, with the hope of accelerating the process.

In the following, Section 9.3 explores an initial approach that combines the Ψ-GNN model presented in Chapter 7 with the Jacobi-Schwarz iterative solver. While this approach is shown to be convergent, it presents challenges due to the high number of iterations required by the Jacobi-Schwarz method for convergence. Additionally, the method can only converge up to a precision directly linked to the accuracy of the trained Ψ-GNN. To address these challenges, Section 9.4 proposes coupling a GNN-based model with a two-level Additive Schwarz method (ASM) and using this hybrid approach as a preconditioner for a Conjugate Gradient method.

9.2 . Machine Learning and Domain Decomposition

Merging Domain Decomposition and Machine Learning is a recent and promising field of research. In Heinlein et al. (2021a) and more recently in Klawonn et al. (2023), the authors provide an extensive overview and classification of the different methods developed in this topic. This section aims to summarize some of the various existing algorithms, present their strengths and limitations, and explain the position of our method with respect to this literature.

A vast part of the literature involves combining Domain Decomposition with PINNs (Raissi et al., 2019a) (see also Chapter 3). PINNs may face challenges when applied to larger problems. As the domain size grows, so does the complexity of the problem, necessitating larger networks to accurately capture all features. Consequently, the time and storage requirements for computing the partial derivatives through automatic differentiation also increase, which can become overwhelming. Moreover,

since the PINN loss function can be highly non-convex, higher problem complexity could lead to a challenging optimization problem, resulting in reduced accuracy or no convergence at all (Krishnapriyan et al., 2021). By breaking down the global optimization problem into many smaller local sub-problems, Domain Decomposition offers a potential solution to scale up PINN networks. This approach helps parallelize the training process and potentially reduces the complexity of the global optimization problem, while also potentially mitigating the spectral bias of neural networks (Wang et al., 2022a).

A subset of these methods incorporates Machine Learning models that use a Domain Decomposition architecture with *non-overlapping* subdomains. Essentially, these methods train distinct PINN networks on each subdomain and ensure local communication between subdomains by enforcing continuity across subdomain interfaces as additional loss terms in the loss function. This is the case in cPINNs (Jagtap et al., 2020), which are applied to systems of conservation laws and further expanded to handle generic PDEs in arbitrary space-time domains in XPINNs (Karniadakis, 2020). Hu et al. (2022) conducted a comparison between PINNs and XPINNs, demonstrating that the use of Domain Decomposition indeed reduces the complexity of the problem and can enhance generalization. However, each sub-problem is trained with a smaller amount of data, which tends to cause the local models to overfit. Related approaches include DPINN (Dwivedi et al., 2021), which introduced a new interface loss inspired by the flux conditions of the finite volume method, and hp-VPINN (Kharazmi et al., 2021). In contrast to previous methods, hp-VPINN leverages the variational formulation of the PDE residuals, with piecewise polynomial test functions that induce, by definition, an implicit decomposition of the domain. As a result, the authors in (Kharazmi et al., 2021) use a single deep network to approximate the solution over the whole domain despite implicitly decomposing it. However, it is more difficult to parallelize such a method. All of these methods have demonstrated satisfactory results for training PINNs at a larger scale. However, one of their key drawbacks is related to the use of non-overlapping subdivision of the domain. Since the interface conditions, which ensure the continuity of the solution across subdomains, are only weakly constrained in the loss function, they can lead to artificial discontinuities at the subdomain interfaces. Additionally, the inclusion of extra interface terms not only introduces additional hyperparameters that need to be tuned to train the best model but also may compete with the PDE losses (Wang et al., 2022a).

Another subset of these methods combines Machine Learning with Domain Decomposition on *overlapping* subdomains. In D3M (Li et al., 2020a) and DeepDDM (Li et al., 2020b), the subdomain solvers in a parallel iterative Schwarz method are replaced by the Deep Ritz method (Yu et al., 2018) and PINNs, respectively. Note that the Deep Ritz method is closely related to PINNs in the sense that it uses dense neural networks to solve the PDE in its variational form. In both cases, the exchange of

information is enforced via additional boundary conditions, which change in each iteration until the convergence of the method. An extension of D3M is proposed in (Li et al., 2023) to reduce the spectral bias observed with the local PINN sub-solvers by introducing new multi-Fourier feature networks in each local subdomain. Previous methods trained PINNs locally, i.e., considering a local loss function in each subdomain, which may appear counterproductive. Instead, the training can be approached globally. For instance, in Moseley et al. (2023), the authors draw inspiration from the finite element method, where the solution of a PDE is expressed as a sum of a finite set of basis functions with compact support. The method, referred to as Finite Basis Physics-Informed Neural Network (FBPINN), uses local window functions to form a global function, i.e., the global solution on the entire domain. The global function is expressed as a sum of the local window functions that are learned by local PINNs. In particular, the local windows (hence basis functions) are defined over small, overlapping subdomains such that, in FBPINNs, no additional interface condition is necessary. FBPINNs are further extended in Dolean et al. (2024), in which additive, multiplicative, and hybrid iteration methods based on a Schwarz-like domain decomposition method for the training of FBPINNs are introduced. Dolean et al. (2024) also provides a preliminary implementation of a coarse space correction to FBPINNs to enhance scalability regarding the number of subdomains. In Dolean et al. (2023), these ideas are extended even further by adding multiple levels of domain decompositions, similar to classical multilevel Schwarz methods. This results in improved convergence properties and increased accuracies in the considered PDE solutions and, at the same time, mitigates convergence problems related to the spectral bias of neural networks.

The previously described methods can be viewed from two perspectives: either as using a Domain Decomposition strategy to parallelize and accelerate the training of PINNs, or to use PINNs (and its variant) as subdomain solvers within Domain Decomposition algorithms. Regardless, another class of methods involves leveraging Machine Learning to enhance the convergence properties or computational efficiency of Domain Decomposition methods. Typically, this is achieved by learning or approximating optimal interface conditions or other crucial parameters. For instance, Chung et al. (2021), Heinlein et al. (2021b), Heinlein et al. (2019), and Klawonn et al. (2024) all concentrate on leveraging Machine Learning models to learn and refine the construction of adaptive coarse spaces. Other research focuses on learning optimal interface conditions, such as in Knoke et al. (2023), where the authors approximate the interface operator in an optimized Schwarz method applied to the resolution of time-harmonic Maxwell's equations. Finally, both the work in Taghibakhshi et al. and Taghibakhshi et al. (2023) emphasize the use of GNNs to learn subdomain interfaces in one or two-level optimized Schwarz methods, respectively.

The thesis work presented in this chapter falls under the category of harnessing

Machine Learning models to replace subdomain solvers in an overlapping Schwarz method. From a practical standpoint, we build upon the groundwork developed in Part II and explore the application of GNN models within a Domain Decomposition framework. Unlike the methods discussed earlier, which rely on training separate networks for individual subdomains, we propose using a single GNN model. Once trained, this GNN model can infer solutions across various numbers of subdomains without necessitating retraining. Furthermore, the resulting model can adapt to global problem sizes of varying shapes, as it is mesh-based and partitioning can be accomplished using a well-known partitioner, such as METIS (Karypis and Kumar, 1997). Moreover, the aforementioned methods are typically assessed on 1D problems, and when extended to 2D, the problem geometries are simpler (usually squared shapes) with fixed partitioning, and their size still remains far from meeting industrial requirements. Given the critical nature of accurately sampling collocation points in PINNs, these methods would encounter challenges in generalizing across problems with diverse sizes and shapes, without requiring retraining or prior knowledge of the domain shape. In Section 9.3, we present an initial attempt, exploring the use of Ψ-GNN as a subdomain solver within a parallel overlapping Schwarz method. While this approach demonstrates convergence and yields satisfactory results, its generalization remains challenging due to the substantial number of required training samples and discontinuities at interfaces, which decrease precision with a larger number of subdomains. Nevertheless, all these methods represent complete Machine Learning solutions, where despite achieving satisfactory precision, their accuracy relies solely on the capabilities of the trained model, without potential for post-training improvements. To address this, in Section 9.4, we propose constructing a GNN-based model leveraging an Additive Schwarz structure. This GNN model, adaptable to any shape and size due to its Domain Decomposition structure, is then used as a preconditioner for a Krylov method. The resulting solution achieves the desired accuracy, thanks to the Krylov method, whose convergence is boosted by a GNN-based preconditioner leveraging batch parallel computations on GPUs to solve multiple sub-problems simultaneously.

9.3 . Ψ-GNN-Jacobi-Schwarz iterative method

Section 9.3.1 introduces the methodology, and Section 9.3.2 discusses the dataset and the configuration of the Ψ-GNN model. Section 9.3.3 presents the results, and Section 9.3.4 highlights the limitations of this approach.

9.3.1 . Methodology

We consider the resolution of Poisson problems with Dirichlet boundary conditions, as defined in Equation (4.1). Using the same configuration as the one introduced in Section 4.1, Poisson problems are discretized using the Finite Element Method with first-order finite elements, resulting in linear systems to solve in the form of $A\mathbf{u} = \mathbf{b}$,

with $A \in \mathbb{R}^{N \times N}$, $\mathbf{b} \in \mathbb{R}^N$, $\mathbf{u} \in \mathbb{R}^N$ is the solution vector to be sought, and N is the number of degrees of freedom.

In this Section, the main idea is to replace the resolution of local Poisson problems (i.e., Poisson problems restricted to sub-meshes) with an inference of a Machine Learning model. At each iteration of the Jacobi-Schwarz method and for each sub-mesh, a Poisson problem with Dirichlet boundary conditions has to be solved, as described in Equation 8.3. This aligns with the framework defined in Section 4.1, and the use of the GNN-based models developed in Part II to solve these local Poisson problems is possible. Given that the Dirichlet boundary conditions vary at each iteration, whether the nodes are at the boundary of the global mesh or at the interface between sub-meshes, the use of the Ψ-GNN model developed in Chapter 7 is pertinent since it offers enhanced generalization capabilities for handling boundary conditions.

Formally and following the notations introduced in Section 8.2, we use Ψ-GNN as a GNN-based solver that, given a discretized Poisson problem with respect to a sub-mesh $\Omega_{h,i}$ at iteration n of the Jacobi-Schwarz method described in Algorithm 2, outputs an approximate solution $\tilde{\mathbf{u}}_i^n$ such that:

$$\tilde{\mathbf{u}}_i^n = \Psi\text{-GNN}\left(\Omega_{h,i},\ R_i A R_i^T,\ R_i \mathbf{b}^n\right) \tag{9.1}$$

It is worth noting that the interface values in $\mathbf{b}^n$ are updated at each iteration using the solution computed at iteration n. The next iterate $\mathbf{u}^{n+1}$ is then computed by assembling all sub-solutions together using partition of unity as follows:

$$\mathbf{u}^{n+1} = \sum_{i=1}^{K} R_i^T D_i \tilde{\mathbf{u}}_i^n \tag{9.2}$$

In the field of Machine Learning, data is typically fed to the model in a batch structure rather than one by one (refer to Chapter 2). The current work aims to harness this "per-batch" structure to enhance the resolution of local Poisson problems by using GPU parallelization to simultaneously infer multiple solutions. Furthermore, this hybrid approach enables the selection of a sub-mesh size aligned with the optimal capabilities of the GNN models. The iterative Ψ-GNN-Jacobi-Schwarz (Ψ-GNN-JSM) algorithm is detailed in Algorithm 6.

9.3.2 . Dataset description and Ψ-GNN training

In the following, we propose to address the resolution of Poisson problems with Dirichlet boundary conditions, in the same configuration as the one described in Section 4.3. A first idea would be to directly apply the Ψ-GNN model trained in Chapter 7 to solve the sub-problems from the Jacobi-Schwarz method. However,

Algorithm 6 Ψ-GNN-Jacobi-Schwarz Method (Ψ-GNN-JSM)

Let $\mathbf{u}^n$ be an approximate solution to the Poisson problem (4.1) at iteration n. The updated approximate solution $\mathbf{u}^{n+1}$ is computed as follows:

1. Update $\mathbf{b}^n$ at the interfaces between subdomains using $\mathbf{u}^n$.
2. Solve all K sub-problems using Ψ-GNN :

$$[\tilde{\mathbf{u}}_1^n, \dots, \tilde{\mathbf{u}}_K^n] = \Psi\text{-GNN}\left([E_{h,1}, \dots, E_{h,K}]\right) \tag{9.3}$$

   where $E_{h,i} = \left(\Omega_{h,i},\ R_i A R_i^T,\ R_i \mathbf{b}^n\right)$ represents a discretized Poisson problem on a sub-mesh i. Note that in this formulation, the resolution of (9.3) implies that all subdomains are solved simultaneously in one inference. However, if the number of sub-problems becomes too large, $[E_{h,1}, \dots, E_{h,K}]$ can be partitioned into N_b batches, allowing all problems to be solved using N_b inferences of Ψ-GNN.

3. Assembling everything together using partition of unity :

$$\mathbf{u}^{n+1} = \sum_{i=1}^{K} R_i^T D_i \tilde{\mathbf{u}}_i^n$$

adopting this approach would result in a suboptimal solution, as the distribution of the sub-problems differs significantly from that of the global problem. This distinction is particularly true at the interfaces between subdomains, where values vary at each iteration and differ from the distribution of the boundary conditions in the global problem (see Equation (8.3)). Here, we suggest training the Ψ-GNN directly on sub-problems sampled from the original Jacobi-Schwarz method, which poses a challenging task.

In Part II, all GNN-based models are trained using $10,000$ samples, aiming to cover the largest distribution possible while reducing the computational time for training (already 48h to train Ψ-GNN in that setting). However, when it comes to constructing a dataset for training Ψ-GNN within the context of the Ψ-GNN-JSM method, some additional complexities emerge.

For the Ψ-GNN-JSM algorithm to function effectively, the Ψ-GNN model must be trained on a sufficient number of sub-problems sampled from the original Jacobi-Schwarz method. This means that at each iteration of the Jacobi-Schwarz method for a given global Poisson problem, all local Poisson problems must be retained for building the dataset. This process leads to a rapid growth in the size of the dataset when considering a sufficient amount of global problems. Thus, achieving adequate

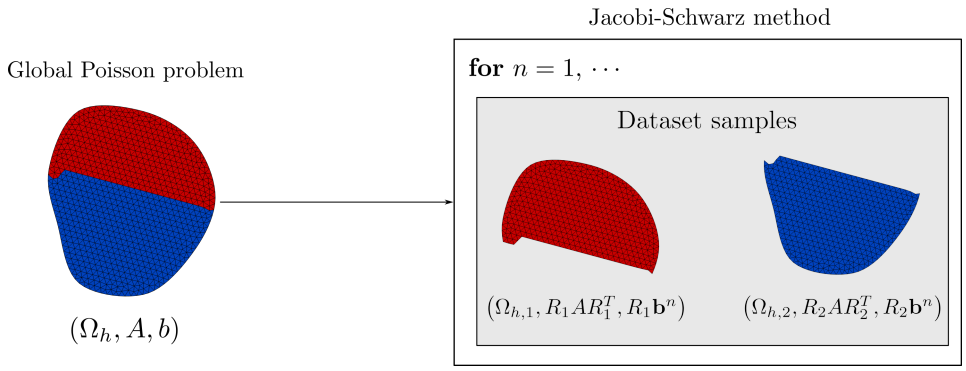


Figure 9.1: Illustration of the samples included in the dataset for training the Ψ-GNN model. These samples correspond to discretized Poisson problems, each restricted to a subdomain derived from the original global problem. They are extracted at each iteration n of the Jacobi-Schwarz method until the required precision is achieved.

coverage of the distribution becomes exceedingly challenging.

To illustrate this, let us imagine building a dataset with only 100 Poisson problems. For each of these problems, let us suppose that the mesh is further subdivided into 4 sub-meshes with an overlap of 2. Under that configuration, the solver typically undergoes around 300 iterations to attain a residual norm precision of approximately 10^{-7}, as illustrated by the green curve in Figure 8.3 of Chapter 8. As a result, the dataset now includes approximately $120,000$ samples $(300 \times 100 \times 4)$.

To facilitate training and provide a "proof-of-concept" for the convergence of the method, we construct a simplified dataset: 100 Poisson problems with Dirichlet boundary conditions are generated following the process described in Section 4.3. For each of these Poisson problems, the mesh contains around 800 to 1000 nodes. We also use constant force and boundary functions: $f(x, y) = \alpha_0$ within Ω and $g(x, y) = \alpha_1$ along $\partial\Omega$, where α_0 and α_1 are drawn from a uniform distribution within the range of $[1, 10]$. For each Poisson problem, the mesh is subdivided into 2 sub-meshes with an overlap of 2 using the partitioner Metis (Karypis and Kumar, 1997). This results in sample sub-meshes of approximately 500 nodes. For each Poisson problem, all sub-problem data are saved in the dataset at each iteration until the JSM method reaches a residual error of 10^{-7}. In that configuration, the JSM requires around 40 iterations to converge. By the end of the generation process, the dataset contains 200 different sub-meshes (100 meshes split into 2 sub-meshes) and a total of 7858 samples. Figure 9.1 illustrates the samples included in the dataset, extracted from the Jacobi-Schwarz method and the subdivision of a global Poisson problem.

Regarding the Ψ-GNN model, it is trained on the dataset described above, using the hyperparameters presented in the result Section 7.4.1. The model is trained using two P100 Nvidia GPUs for 400 epochs during approximately 30h. Table 9.1

shows the results obtained after training when averaged over the whole test set. The obtained results are better than those obtained in Chapter 7, which is expected since the distribution of problems is easier to tackle.

Metrics	Residuals (10^{-4})	MSE w/LU (10^{-5})	Nb of weights
Ψ-GNN	3.1 ± 0.4	7.6 ± 2	1455

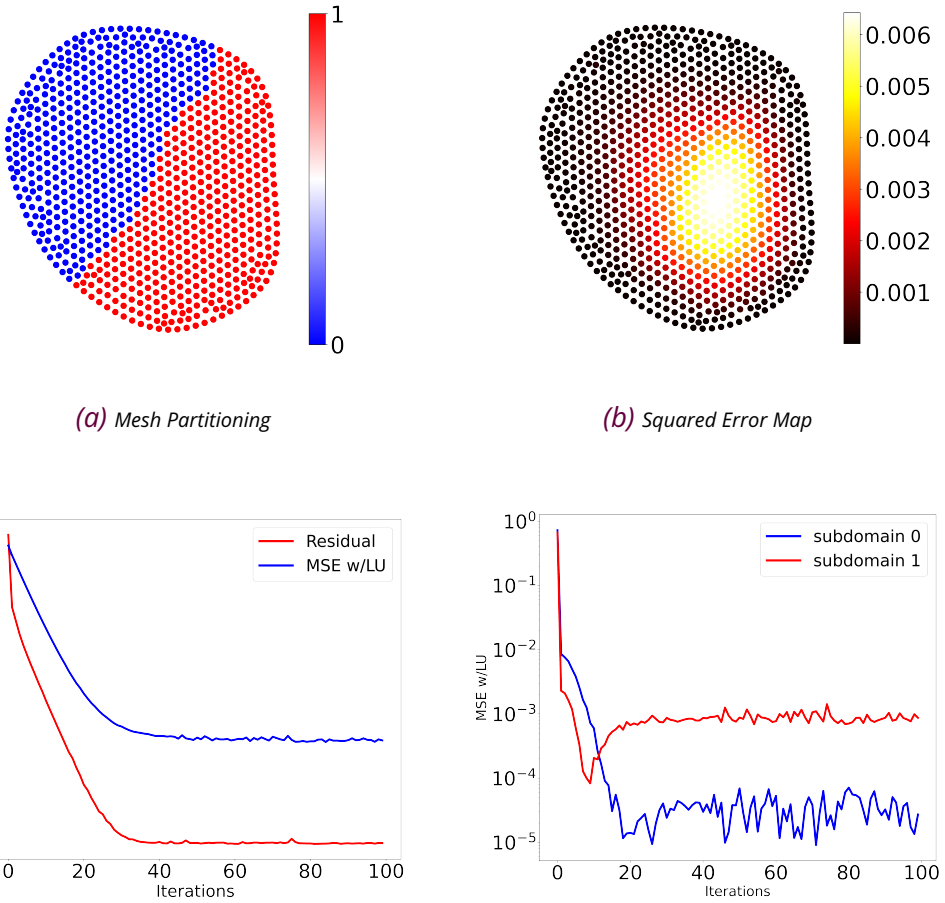
Table 9.1: Results of Ψ-GNN averaged over the whole test set.

9.3.3 . Results

This section presents several results to assess the performance of this method while demonstrating its clear limitations.

First, Figure 9.2 illustrates the resolution of a Poisson problem with 881 nodes using the Ψ-GNN-JSM method. In this example, the considered Poisson problem is derived from the distribution of global Poisson problems used to construct the dataset in Section 9.3.2: the mesh is partitioned into 2 subdomains, as illustrated in Figure 9.2a, the force function is $f = 2$, and the Dirichlet boundary function is $g = 6$. The method is run for 100 iterations. Figure 9.2c depicts the evolution of the Residual and MSE w/LU (i.e., MSE between the global solution obtained using Ψ-GNN-JSM and a global "ground-truth" solution computed by LU decomposition). At the last iteration, the Residual reaches a value of 2.2×10^{-5} and an MSE w/LU of 1.4×10^{-3}, highlighting the capability of the proposed approach for solving this Poisson problem. It is worth noting that the method stagnates at some point: the predicted subproblems become very similar (small oscillations in Figure 9.2c), and the model no longer improves. In addition, Figure 9.2d depicts the MSE w/LU for each subdomain during the 100 iterations. This figure measures how each subdomain is solved using Ψ-GNN with respect to a "ground-truth" subdomain solution. In this example, it is evident that subdomain 0 (blue curve) is solved more accurately than subdomain 1 (red curve), and the global solution converges to a precision no greater than that achieved in the least accurate subdomain. As a result, it is not surprising to observe that the highest errors in the squared error map, shown in Figure 9.2b, are located at the bottom part of the mesh, corresponding to subdomain 1 (red subdomain in Figure 9.2a).

Table 9.2 reports the performance of Ψ-GNN-JSM in solving multiple Poisson problems, considering different configurations involving a variation in the number of nodes and subdomains. Force and boundary functions are sampled similarly to Section 9.3.2. For each configuration, we solve 100 global Poisson problems using Ψ-GNN-JSM with 100 iterations. The method is, however, stopped if the MSE between two iterates becomes lower than 10^{-5}, indicating some stagnation. Regardless of the size of the Poisson problem, the mesh is partitioned into sub-meshes



(a) Mesh Partitioning

(b) Squared Error Map

(c) Residual and MSE w/LU

(d) Subdomain MSE w/LU

Figure 9.2: Resolution of a global Poisson problem using the Ψ-GNN-JSM method. At the top, Figure 9.2a illustrates the mesh partitioning, and Figure 9.2b represents the squared error map between the Ψ-GNN-JSM solution and a "ground-truth" LU solution. At the bottom, Figure 9.2c depicts the evolution of the Residual and MSE w/LU during the 100 iterations, and Figure 9.2d shows the evolution of the MSE w/LU for each subdomain across the 100 iterations.

Nb nodes	Nb subdomains	MSE w/LU
838 (± 44)	2 (± 0)	0.027 (± 0.04)
3234 (± 177)	7 (± 1)	1.23 (± 2)
7119 (± 382)	15 (± 1)	16.9 (± 21)

Table 9.2: Averaged results ($\pm$ standard deviation) of Ψ-GNN-JSM for solving Poisson problems in different configurations.

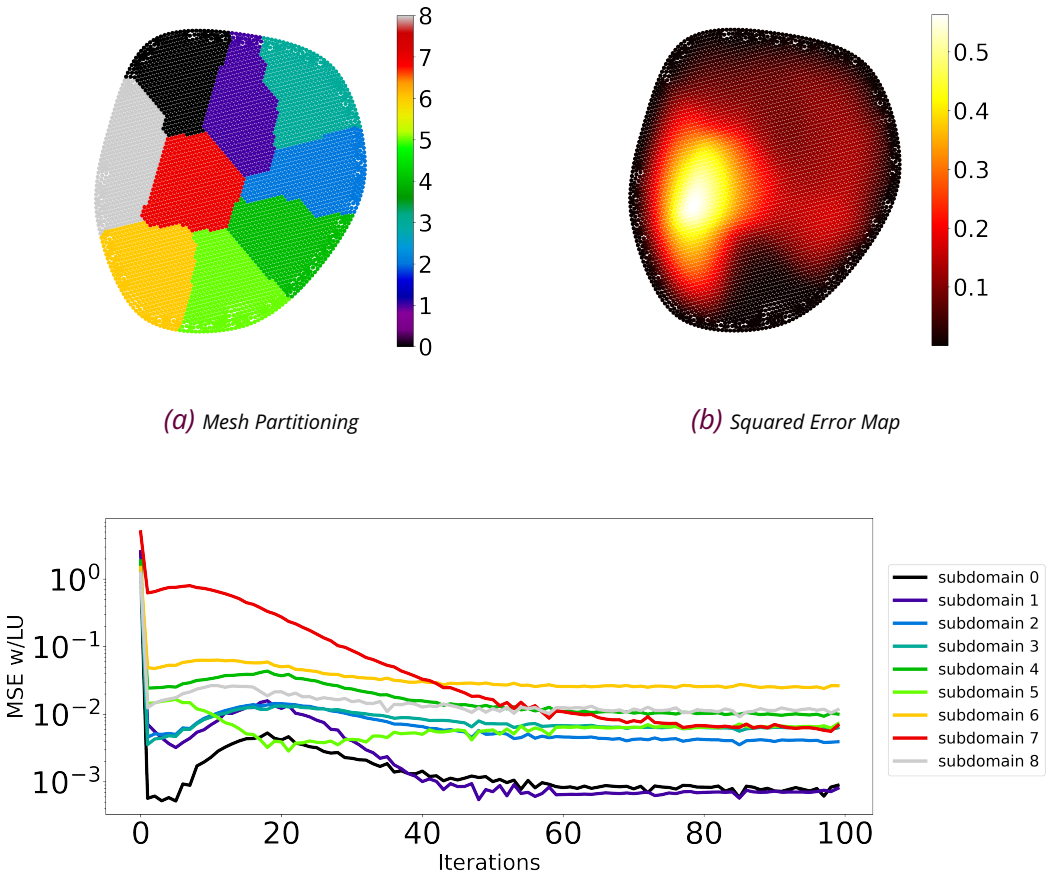
with 500 nodes. The first line corresponds to 100 Poisson problems with an average of 838 nodes, which is the configuration used for constructing the dataset (see Section 9.3.2). The results show an average MSE w/LU of 0.027, demonstrating that the proposed approach can consistently solve these types of Poisson problems with good accuracy.

The second line represents 100 Poisson problems with around 3234 nodes and 7 subdomains, solved using Ψ-GNN-JSM with an average MSE w/LU of 1.23. The third line corresponds to 100 Poisson problems with around 7119 nodes and 14 subdomains, solved using Ψ-GNN-JSM with an average MSE w/LU of 16.9. These results reveal that the proposed method encounters challenges in generalizing to meshes with an increasing number of nodes and subdomains. This difficulty arises from two main issues: Firstly, the sub-problems are approximately solved with Ψ-GNN, and the approximation error is propagated through the mesh at each iteration, causing the sub-problems to slowly deviate from those learned by the GNN model. Secondly, interior subdomains (those not in contact with the global boundary of the domain) are poorly solved, as they do not belong to the distribution of trained instances, and this error is then propagated to other subdomains. This challenge is illustrated in Figure 9.3, which depicts the resolution of a Poisson problem on a mesh with 4386 nodes split into 9 sub-meshes with $f = 4$ and $g = 4$. Figure 9.3c shows the MSE w/LU for each subdomain, highlighting that the interior subdomain (the red subdomain in Figure 9.3a, corresponding to the red curve in 9.3a), is solved with the highest error at the first iteration.

9.3.4 . Discussions & Limitations

This section explored combining the Ψ-GNN model from Chapter 7 with the iterative Jacobi-Schwarz method described in Chapter 8. The presented results showed a "proof-of-concept" of the potential of this approach but highlighted numerous limitations. While the method indeed converges to a solution, it stagnates to a precision that corresponds, at best, to the precision of the trained model: this method cannot provide more accurate predictions.

Moreover, it is difficult to construct a dataset that allows generalization to more complex problems due to the high number of iterations required for the Jacobi-Schwarz method to converge. As a result, considering problems with a larger number of nodes, and hence a larger number of subdomains yields worse results because many sub-problems do not belong to the original trained distribution. These out-of-distribution sub-problems are solved with some large error, and this error is propagated to all other subdomains subsequently. To alleviate the issue of dataset construction, some research directions could consider increasing the size of the overlap to reduce the number of iterations, which would, however, increase the size of the sub-meshes. Another idea would be to not consider all sub-problems at each iteration, since, at some point, the sub-problems to solve are close to each



(a) Mesh Partitioning

(b) Squared Error Map

(c) Subdomain MSE

Figure 9.3: Resolution of a Poisson problem on a mesh with 4386 nodes using the Ψ-GNN-JSM method. At the top, Figure 9.3a illustrates the mesh partitioning, and Figure 9.3b represents the squared error map between the Ψ-GNN-JSM solution and a "ground-truth" LU solution. At the bottom, Figure 9.3c shows the evolution of the MSE w/LU for each subdomain across the 100 iterations.

other. However, this method did not really show any improvement in the experimental tests compared to the presented method. As a result, it is then essential to explore another approach.

9.4 . GNN-based Schwarz Preconditioner

In previous Section 9.3, we discussed the development of a hybrid iterative solver by combining the Ψ-GNN model with the Jacobi-Schwarz method. Given the limitations observed in this approach, this Section explores a different hybrid approach.

We propose constructing another hybrid solver, still combining the GNN-based mod-

els presented in Part II with Schwarz methods introduced in Chapter 8. However, this section explores the application of this hybrid solver as a preconditioner for a Krylov method. Similar to the previous section, the objective is to leverage the GNN models to solve the multiple sub-problems in the context of a Schwarz preconditioner.

Using such an approach offers several benefits. Thanks to the Krylov method, the proposed approach converges to the solution of the problem with any desired precision. The robustness and efficiency of the Krylov method are significantly enhanced by using a fully GNN-based preconditioner. This GNN-based preconditioner, whose architecture leverages a two-level DDM approach, is capable of handling meshes of any size and making the Krylov method scalable with respect to the number of subdomains. Moreover, its parallel execution on GPUs ensures its fast application within the Krylov method.

In the following, Section 9.4.1 presents the methodology, Section 9.4.2 covers the generation process of the dataset, and Section 9.4.3 presents the configuration of the considered GNN-based model. Finally, Section 9.4.4 discusses the results obtained.

9.4.1 . Methodology

Similar to the previous section, let us consider the resolution of Poisson problems with Dirichlet boundary conditions, as defined in the system (4.1). Using the same configuration as the one introduced in Section 4.1, the Poisson problems are discretized using the Finite Element Method with first-order finite elements, resulting in linear systems to solve in the form $A\mathbf{u} = \mathbf{b}$.

To address such problems, we suggest using a well-known Krylov solver: the Conjugate Gradient (CG). The performance of CG can be significantly improved through preconditioning, as detailed in Section 8.4, and illustrated in Figure 8.4. In the following, the proposed hybrid method will be formulated by drawing inspiration from the Additive Schwarz method (ASM) described in Algorithm 3, given its efficacy as a preconditioner for CG (refer to Chapter 8).

ASM belongs to the family of Schwarz methods introduced in the previous Chapter 8. It is a variant of the Restricted Additive Schwarz method (RAS) that does not rely on Partition of Unity functions (see Algorithm 3). In contrast to the Jacobi-Schwarz method (JSM) introduced in the previous section, ASM presents the problem in a different formulation: ASM uses an algebraic formulation, directly addressing the discretized residual equation $\mathbf{r} = \mathbf{b} - A\mathbf{u}$. As a result, the local Poisson problems (i.e., the Poisson problems to be solved in each subdomain) differ from those in the Jacobi-Schwarz method. In ASM, the local Poisson problems incorporate $\mathbf{r}$ as a source term and impose zero Dirichlet boundary conditions on all their boundaries, whether at the global domain boundary or subdomain interfaces. This configura-

tion significantly simplifies the distribution of sub-problems at the interfaces compared to the Jacobi-Schwarz method, where boundary conditions are non-zero and vary at each iteration.

Furthermore, source terms of local Poisson have increasingly small norms as the Conjugate Gradient algorithm aims to minimize the residual. This poses a challenge to our methodology. When the residual norm becomes very small, implying that the source term norm tends toward zero, the Machine Learning model may struggle to generalize properly, often resulting in a trivial solution, i.e., a solution equal to zero everywhere. This can lead the Conjugate Gradient algorithm to stagnate at a certain precision threshold since the GNN-based preconditioner would, at a certain point, always provide a null correction to update the solution. To address this issue, we propose to normalize the source term of the local Poisson problems.

The performance of the ASM preconditioner can also be improved by using a coarse space correction, as detailed in Section 8.5. In its discrete formulation, the two-level ASM preconditioner applied to a residual vector $\mathbf{r} \in \mathbb{R}^N$ can be written as follows:

$$M^{-1}_{\text{ASM},2}\mathbf{r} = \underbrace{R_0^T (R_0 A R_0^T)^{-1} R_0}_{\text{coarse problem}} \mathbf{r} + \underbrace{\sum_{k=1}^{K} R_i^T (R_i A R_i^T)^{-1} R_i}_{\text{local problems}} \mathbf{r} \tag{9.4}$$

where K is the number of sub-meshes, $(R_i)_{1 \leq i \leq K}$ are restricted operators from the global mesh to local sub-meshes, and R_0 is the restriction operator from the global mesh to the coarse mesh.

In this context, we propose using the Deep Statistical Solvers (DSS) model introduced in Chapter 5 with some minor modifications, which will be detailed further in Section 9.4.2. There are several reasons for this choice. Firstly, since the boundary conditions of the local Poisson problems are always zero, there is no need for an enhanced model at these particular points. Additionally, DSS can be trained with more samples faster and provides quicker solutions than Ψ-GNN, which requires the use of the Broyden solver (see Section 7.4.4). The last point is a practical consideration. At the time of writing this thesis, we are currently investigating the performance of such a hybrid method within an industrial solver written in C++. The architecture of DSS is, in fact, much easier to script from Pytorch to C++ compared to Ψ-GNN, where we would also need to develop the Broyden solver. Note that it is still possible to use Ψ-GNN, which, on the other hand, could provide better sub-problem solutions as the number of nodes for a sub-mesh grows.

Formally, we use DSS as a GNN-based solver that, given a discretized Poisson problem with respect to a sub-mesh i, $1 \leq i \leq K$, outputs an approximate residual solution $\tilde{\mathbf{r}}_i$ as follows:

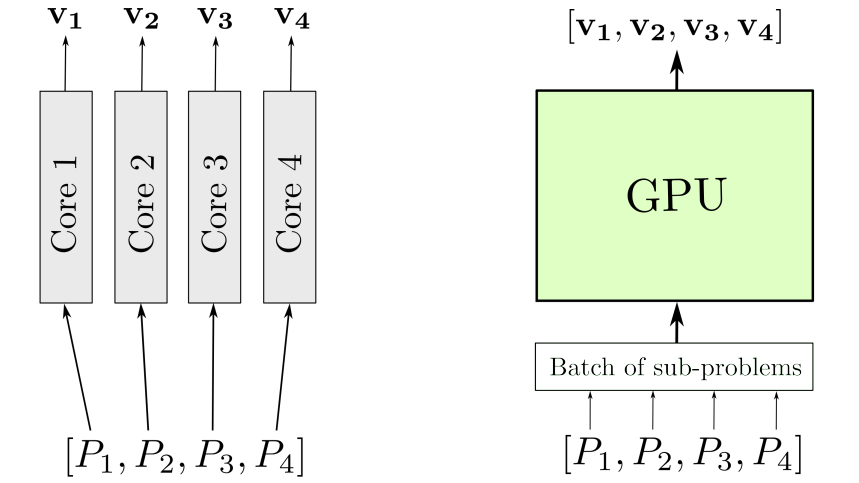


Figure 9.4: Illustration of the resolution of local Poisson sub-problems $[P_1, \dots]$ in (9.4a): the classic approach where they are solved in parallel on CPU cores, and (9.4b): the hybrid approach where they are solved in parallel on a GPU using GNN-based models.

$$\tilde{\mathbf{r}}_i = \mathsf{DSS}\left(\Omega_{h,i},\ R_i A R_i^T,\ \frac{R_i \mathbf{r}}{\|R_i \mathbf{r}\|}\right) \tag{9.5}$$

The application of the two-level hybrid preconditioner to a residual vector $\mathbf{r}$ is then computed as follows:

$$M^{-1}_{\text{hybrid},2}\mathbf{r} = R_0^T (R_0 A R_0^T)^{-1} R_0 \mathbf{r} + \sum_{k=1}^{K} R_i^T \|R_i \mathbf{r}\| \tilde{\mathbf{r}}_i \tag{9.6}$$

The proposed hybrid preconditioner, denoted ASM-GNN, is described in Algorithm 7 and the updated Conjugate Gradient preconditioned with the ASM-GNN method, denoted as CG-ASM-GNN, is detailed in Algorithm 8. Similar to the previous Section, we hope to capitalize on the "per-batch" structure to accelerate the resolution of local Poisson problems. This approach harnesses GPU parallelization to infer multiple solutions simultaneously (see Figure 9.4b). This is in contrast with traditional approaches that solve the local Poisson problems using parallel computations on CPUs (see Figure 9.4a). It is well-known that GPUs have more parallel computational power than CPUs. However, legacy codes often struggle to leverage GPU computations. The proposed hybrid method can act as a black-box approach that can be easily integrated into such frameworks to leverage GPU computations.

When used with a one-level method (i.e., without the coarse space correction), the ASM-GNN preconditioner can be considered as a full Machine Learning-based pre-

Algorithm 7 ASM-GNN Preconditioner

Input: $\mathbf{r}$, **Output:** $\mathbf{z}$

1. Solve the coarse space problem :

$$\mathbf{r}_c = R_0^T (R_0 A R_0^T)^{-1} R_0 \mathbf{r}$$

2. Solve all K sub-problems using DSS:

$$[\tilde{\mathbf{r}}_1, \dots, \tilde{\mathbf{r}}_K] = \text{DSS}\left([E_{h,1}, \dots, E_{h,K}]\right) \tag{9.7}$$

where $E_{h,i} = \left(\Omega_{h,i},\ R_i A R_i^T,\ \frac{R_i \mathbf{r}^n}{\|R_i \mathbf{r}^n\|}\right)$ represents a discretized Poisson problem on a sub-mesh i. Note that in this formulation, the resolution of (9.7) suggests that all subdomains are solved simultaneously in one inference. However, if the number of sub-problems becomes too large, $[E_{h,1}, \dots, E_{h,K}]$ can be partitioned into N_b batches, allowing all problems to be solved using N_b inferences of DSS.

3. Glue everything:

$$\mathbf{z} = \mathbf{r}_c + \sum_{i=1}^{K} R_i^T \|R_i \mathbf{r}\| \, \tilde{\mathbf{r}}_i$$

Algorithm 8 CG-ASM-GNN

Compute $\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0$, $\mathbf{z}_0 = \text{ASM-GNN}(\mathbf{r}_0)$, $\mathbf{p}_0 = z_0$

for $i = 0, 1, \dots$ **do**

$\rho_i = \langle \mathbf{r}_i, \mathbf{z}_i \rangle$

$\mathbf{q}_i = A\mathbf{p}_i$, $\alpha_i = \dfrac{\rho_i}{\langle \mathbf{p}_i, \mathbf{q}_i \rangle}$

$\mathbf{u}_{i+1} = \mathbf{u}_i + \alpha_i \mathbf{p}_i$

$\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i \mathbf{q}_i$

if $\|\mathbf{r}_{i+1}\| <$ tol **then**

Break;

end if

$\mathbf{z}_{i+1} = \text{ASM-GNN}(\mathbf{r}_{i+1})$, $\rho_{i+1} = \langle \mathbf{r}_{i+1}, \mathbf{z}_{i+1} \rangle$, $\beta_{i+1} = \dfrac{\rho_{i+1}}{\rho_i}$

$\mathbf{p}_{i+1} = \mathbf{z}_{i+1} + \beta_{i+1} \mathbf{p}_i$

end for

conditioner. The introduction of a coarse correction tends to make the preconditioner "hybrid", as the coarse problem is solved using a direct solver. Nevertheless, there is also the possibility of exploring the training of another model to address

the resolution of the coarse space problem.

9.4.2 . Dataset description

In Chapter 5, we trained the DSS model to solve global Poisson problems generated following the process described in Section 4.3. In this chapter, our goal is to tackle the same global Poisson problems on a larger scale, leveraging the DSS model to address the local Poisson problems, as mentioned earlier. The partitioning of the mesh can be done such that the local Poisson problems share the same structural characteristics (e.g. in terms of the number of nodes) as the samples in the distribution of the original DSS model. However, the distribution of the local Poisson problems, with respect to the source term and boundary conditions, logically differs from the global problem. Consequently, similar to the Ψ-GNN-JSM approach, the DSS model should be trained on a new set of samples arising from a correct distribution of instances.

In this context, generating a consistent dataset is, in fact, much easier compared to the previous section where we used the Jacobi-Schwarz iterative solver. Similar to the previously described process, we are interested in extracting and storing all sub-problems involved in the ASM preconditioner at each iteration of the Conjugate Gradient method. Unlike the previous section, Conjugate Gradient, when applied using a classic two-level ASM preconditioner (where all sub-problems are solved using a direct solver), requires far fewer iterations to achieve high precision compared to the iterative Jacobi-Schwarz method, as illustrated in Figure 8.4. This enables us to consider a more extensive set of global problems, with force and boundary functions similar to those detailed in Section 4.3.

Considering the same example as in the previous section[1], the number of iterations required to achieve a precision of 10^{-7} using CG preconditioned with ASM (CG-ASM) is, on average, 25, instead of the 300 iterations needed for the Jacobi-Schwarz iterative solver. As a result, in this configuration, the number of samples generated would be $10,000$ ($100 \times 4 \times 25$) with CG-ASM, as opposed to $120,000$ in the previous section.

The dataset used for training DSS is generated as follows. We focus on solving Poisson problems with Dirichlet boundary conditions, following the process detailed in Section 4.3. The generated meshes have between 6000 and 8000 nodes. Each mesh is divided into sub-meshes of around 1000 nodes with an overlap of 2, resulting in 7 or 8 partitions per mesh. We use the Conjugate Gradient method preconditioned with a classic two-level ASM method[2] to solve global problems up to a residual norm precision of 10^{-12}. The dataset comprises discretized Poisson sub-problems from the two-level ASM preconditioner, extracted at each iteration of the CG algorithm.

[1] 100 global Poisson problems, divided into 4 subdomains.
[2] sub-problems are solved using LU decomposition.

On average, the Conjugate Gradient method requires 30 iterations to converge in that configuration. We aim to solve 700 Poisson problems. On average, the generated dataset should consist of $700 \times 30 \times 7 = 147,000$ different problems. The exact number of generated samples is $164,012$.

9.4.3 . Model configuration and training

As mentioned earlier, the GNN model used in this study is the Deep Statistical Solvers (DSS) model (Donon et al., 2020), described in Chapter 5. However, we propose some slight modifications to its architecture and training. Firstly, the edge features between nodes, originally defined in DSS as the coefficients of the stiffness matrix A, are replaced by structural information of the mesh. As a result, the Message-Passing functions defined in Equations (5.2) and (5.1) are now formulated as follows:

$$\phi^{k+1}_{\rightarrow,i} = \sum_{j \in \mathcal{N}(i)} \Phi^{k+1}_{\rightarrow,\theta}(H^k_i, H^k_j, d_{ij}, \|d_{ij}\|) \tag{9.8}$$

$$\phi^{k+1}_{\leftarrow,i} = \sum_{j \in \mathcal{N}(i)} \Phi^{k+1}_{\leftarrow,\theta}(H^k_i, H^k_j, d_{ji}, \|d_{ji}\|) \tag{9.9}$$

where d_{ij} represents the relative position vector and $\|d_{ij}\|$ its Euclidean distance.

Besides, we also modify the structure of the matrix b'_i defined in Equation (5.7). This matrix originally allowed for distinguishing the values of the right-hand side of the linear system for Interior and Dirichlet nodes, representing a multimodal distribution. In our case, as the boundary conditions remain the same and equal to 0 in this context, there is no need to make this distinction, and b'_i is now solely defined as the normalized residual vector $\frac{\mathbf{r}}{\|\mathbf{r}\|}$. The cost function used for training the model is directly the residual of the discretized Poisson problem, similar to what has been considered so far, but without the change of variable described in Equation (5.6).

The model is trained using the same hyperparameters as those employed in Section 5.2.2, with a few exceptions. The number of iterations $\bar{k}$ is set to 50 due to the increased size of the sample meshes (now 1000 nodes). In the training loss, the discount factor is increased to $\gamma = 0.95$ because of the higher number of iterations. Training is conducted on 2 P100 GPUs using the Adam optimizer with a learning rate of 10^{-2} and a batch size of 500. Additionally, the *ReduceLrOnPlateau* scheduler from PyTorch is applied, reducing the learning rate by a factor of 0.1 during training. These hyperparameters were selected after several experiments and have yielded satisfactory results, but fine-tuning them could be beneficial for further performance improvement.

Table 9.3 shows the averaged results on the entire test dataset, demonstrating that DSS has efficiently learned to solve the sub-problems. This is particularly due to the

distribution considered here, which is much more limited than the one used in the models in Part II.

Metrics	Residuals (10^{-7})	MSE w/LU (10^{-5})	Nb of weights
DSS	2.046 ± 0.1	3.501 ± 0.2	37530

Table 9.3: Results of DSS averaged over the whole test set.

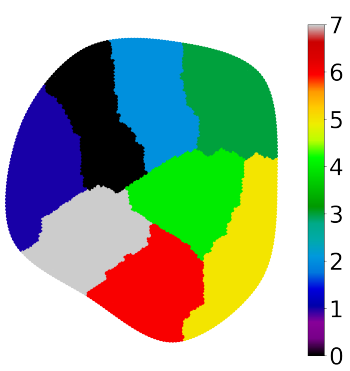
9.4.4 . Results

This section aims to present several results to assess the performance of the proposed approach.

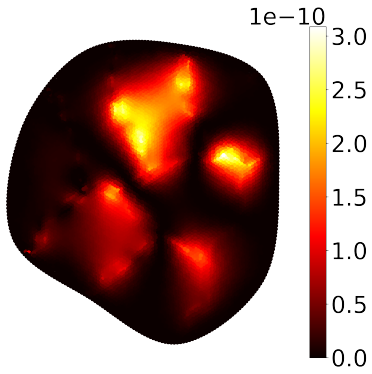
Single test sample

To begin, Figure 9.5c illustrates the resolution of a Poisson problem with Dirichlet boundary conditions using the CG-ASM-GNN method. The considered Poisson problem belongs to the distribution of global Poisson problems used for generating the dataset in Section 9.4.2. The Poisson problem is solved on a mesh with 7310 nodes, split into 8 sub-meshes with approximately 1000 nodes per sub-mesh, as shown in Figure 9.5a. The results obtained with CG-ASM-GNN are compared with those obtained using Conjugate Gradient preconditioned with a classic[3] ASM method (CG-ASM-LU) and a Conjugate Gradient method without preconditioning (CG). All methods are run until the residual norm reaches an error of 10^{-12}. Figure 9.5c shows the evolution of the Residual error for the three configurations. A first observation is to notice that our hybrid method indeed converges to the desired precision, which is, for the first time, in contrast to all other Machine Learning methods studied so far in previous Section 9.3 or in Part II, that stagnate at a certain precision threshold. Besides, Figure 9.5b displays the map of squared error with a "ground-truth" solution[4]. This map shows squared errors of magnitude 10^{-10}, which could be even lower if necessary. Moreover, the method is able to solve this problem in significantly fewer iterations than a Conjugate Gradient method without preconditioning. When compared to the classic CG-ASM-LU method (in fact, the method used to generate the dataset samples), we observe that our method needs a little bit more iterations. This is not surprising since the Conjugate Gradient method receives, with our hybrid preconditioner, approximate solutions, whereas the classic ASM preconditioner provides "exact" solutions. This gap is, however, not substantial here since it only consists of 9 additional iterations for our proposed hybrid solver. As a result, we have illustrated that the proposed model can indeed efficiently solve this Poisson problem and converges almost as well as the classic

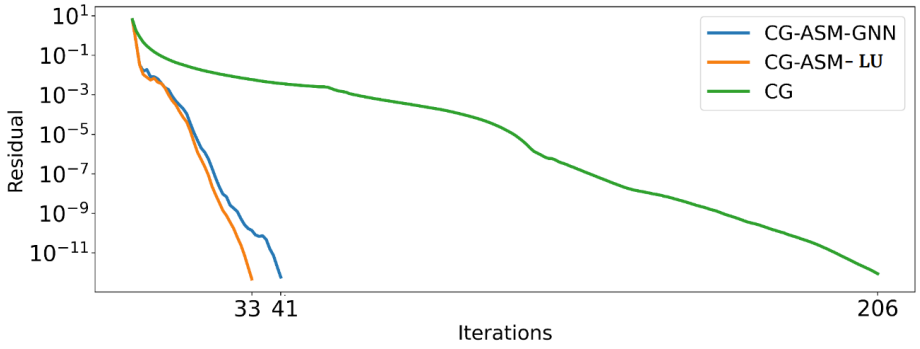
[3] where all sub-problems are solved using LU decomposition.
[4] computed with LU decomposition.



(a) *Mesh Partitioning*



(b) *Squared Error Map*



(c) *Residual error*

Figure 9.5: Resolution of a global Poisson problem on a mesh with 7310 nodes using CG-ASM-GNN, CG-ASM-LU and CG methods. At the top, Figure 9.5a illustrates the mesh partitioning, and Figure 9.5b represents the squared error map between the CG-ASM-GNN solution and a "ground-truth" LU solution. At the bottom, Figure 9.5c depicts the evolution of the Residual error for the three methods.

CG-ASM-LU method. The difference lies in the fact that CG-ASM-GNN leverages GPU parallelization to apply the preconditioner, whereas the classic CG-ASM-LU method uses computations on CPUs.

Multiple tests in various configurations

We further investigate the generalization performance of CG-ASM-GNN by considering the resolution of multiple Poisson problems in various configurations. For each setup, we solve 100 global Poisson problems with Dirichlet boundary conditions, generated following the process described in 9.4.2. The first configuration involves meshes with approximately 2500 nodes, the second configuration with approximately 7000 nodes, and the third configuration with 34000 nodes. Regardless

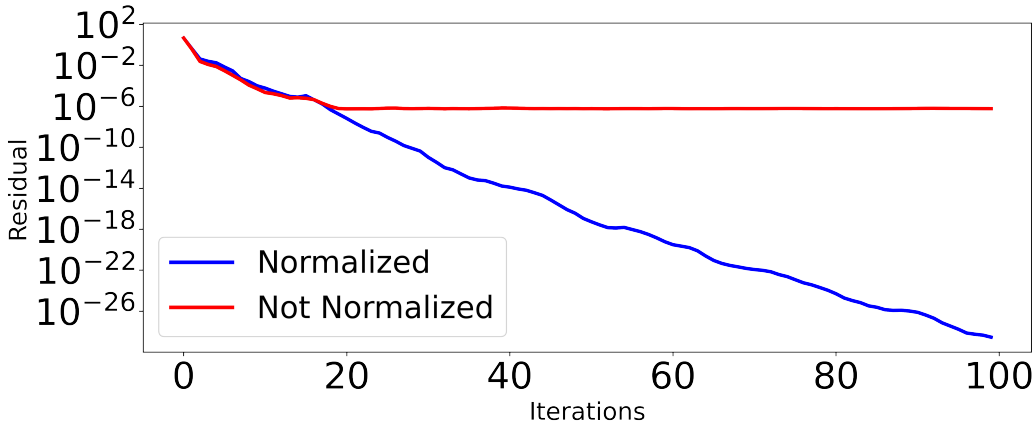
Configuration		Nb Iterations		
Nodes	Subdomains	CG-ASM-GNN	CG-ASM-LU	CG
$2593\ (\pm 144)$	$3\ (\pm 0)$	$26\ (\pm 1)$	$21\ (\pm 1)$	$126\ (\pm 4)$
$7094\ (\pm 385)$	$8\ (\pm 1)$	$40\ (\pm 2)$	$32\ (\pm 2)$	$203\ (\pm 7)$
$34057\ (\pm 1924)$	$34\ (\pm 2)$	$82\ (\pm 4)$	$63\ (\pm 3)$	$427\ (\pm 13)$

Table 9.4: Average ($\pm$ standard deviation) number of iterations required to converge for CG-ASM-GNN, CG-ASM-LU, and CG when applied to solve multiple Poisson problems on meshes with varying numbers of nodes and subdomains.

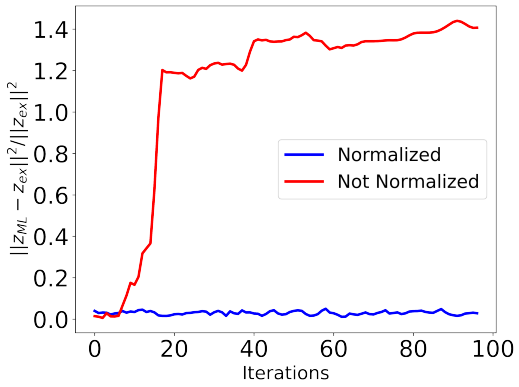
of the number of nodes per mesh, each mesh is divided into sub-meshes of around 1000 nodes, resulting in varying numbers of subdomains for each configuration. Table 9.4 compares, for each of these configurations, the average number of iterations required by CG-ASM-GNN, CG-ASM-LU, and CG algorithms to converge to a residual norm of 10^{-12}. The first observation is that, regardless of the number of nodes and subdomains, the proposed hybrid CG-ASM-GNN indeed converges to the required precision for multiple Poisson problems, validating the consistency of our approach. Secondly, CG-ASM-GNN requires significantly fewer iterations than the CG method. As the number of iterations increases, the required iterations for CG-ASM-GNN also increase but far less than for the CG method. This is attributed to CG-ASM-GNN being built following a two-level ASM method (see Section 8.5). Comparing our hybrid method with CG-ASM-LU reveals that our method converges in slightly more iterations. As the number of subdomains grows, the difference between the number of iterations of CG-ASM-GNN and CG-ASM-LU also grows. This gap could be reduced with a higher stopping criterion but would continue to expand if aiming for a lower stopping criterion. However, this difference is not substantial (requiring 5 and 8 more iterations for 3 and 8 subdomains, respectively, and a gap of 19 iterations for 34 subdomains). As a result, the CG-ASM-GNN method provides an efficient alternative to CG-ASM-LU, with the distinction that CG-ASM-GNN leverages computations on GPUs to apply the hybrid preconditioner, whereas CG-ASM-LU relies on computations on CPUs.

Normalization of the Residual vector

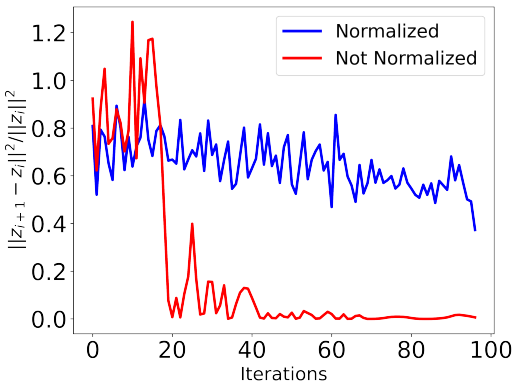
In this section, we motivate the crucial feature of normalizing the residual term in Equation (9.5), which is used as input to the GNN-based model. In the initial tests conducted during the development of this method, we trained a DSS model in the same configuration as described in Section 9.4.3, but we used samples that are



(a)



(b)



(c)

Figure 9.6: Illustration of the results obtained for solving a Poisson problem considering two configurations: one with non-normalized (red curves) and the other with normalized (blue curves) residual vectors as input to the ASM-GNN preconditioner. For both configurations, Figure 9.6a displays the evolution of the global residual error. Figure 9.6b shows the evolution of the relative error between z vectors computed with ASM-GNN and the "exact" ASM, and Figure 9.6c shows the relative error between two consecutive z vectors computed with the ASM-GNN preconditioner.

not normalized[5]. In this section, we use this early model to solve a Poisson problem similar to the one described in Figure 9.5, considering two configurations: one with non-normalized and the other with normalized residual vectors as input to the model. For both configurations, the resolution of this Poisson problem is stopped at 100 iterations, regardless of the achieved accuracy.

Figure 9.6 presents several results to compare these two approaches. In each figure, the red curve corresponds to the non-normalized configuration, while the blue

[5]the input to the model is $\mathbf{r}$ and not $\frac{\mathbf{r}}{\|\mathbf{r}\|}$

curve represents the normalized one. At the top, Figure 9.6a illustrates the evolution of the global Residual error in both configurations during the 100 iterations. The non-normalized setup converges and stagnates at a precision threshold of 10^{-7}, showing no further improvement with increasing iterations. Conversely, the normalized setup converges to a value of magnitude 10^{-24}, which could further improve with additional iterations. In Figure 9.6b, we present the relative error, at each iteration, between the z vector[6] from Algorithm 8 computed with the hybrid ASM-GNN preconditioner and the "exact" ASM preconditioner. The figure demonstrates that this error increases as the number of iterations grows for the non-normalized configuration, whereas it remains quite stable for the normalized one. This implies that the GNN-based model produces progressively worse results as iterations increase. This phenomenon can be attributed to the decreasing norm of the residual vector, which tends to 0 as the number of iterations grows (while the residual vector is rescaled to have a unit norm if normalized). The consequence is that the GNN-based model can no longer differentiate very small residual vectors and ends up outputting, at a certain threshold of the residual, the same results. As a result, the Krylov method receives the same information at each iteration and stops improving, leading to the plateau observed in Figure 9.6a. This is further illustrated in Figure 9.6c, which shows the relative error between two consecutive iterations of z. While the normalized setup produces z vectors that differ from each other at each iteration, this is not the case for the non-normalized one, as the observed relative error tends to 0.

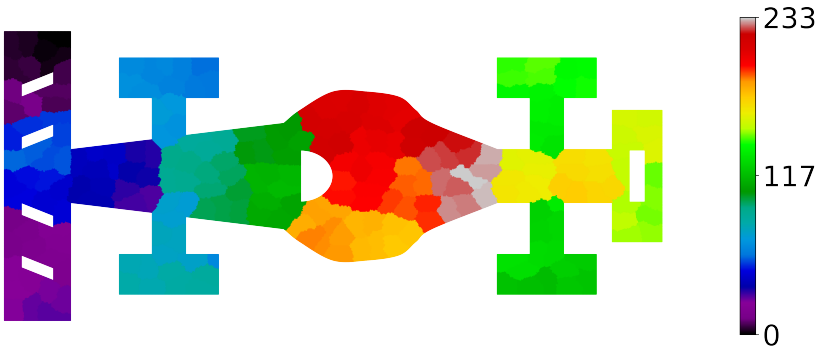
These results motivated the decision to use normalized residual vectors as input for the GNN-based model and, specifically, to train the DSS model with normalized vectors as described in Section 9.4.3. This choice is crucial for developing a hybrid preconditioner that enables the Krylov method to converge to any desired precision.

Large-Scale Poisson Problem

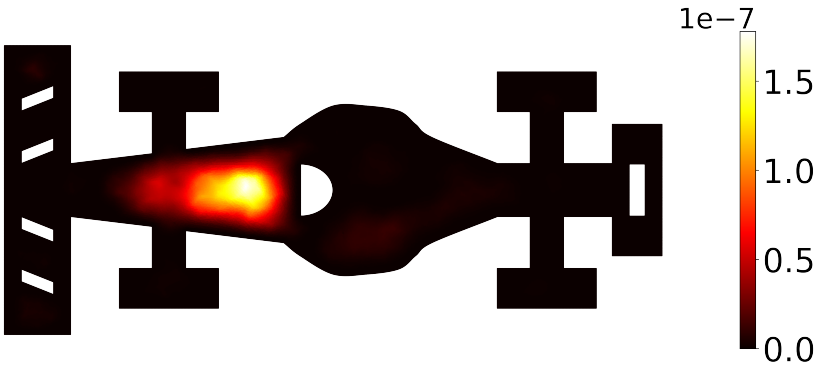
Here, we further investigate the resolution of a Poisson problem at a very large scale. Figure 9.7 illustrates the solution of a Poisson problem with Dirichlet boundary conditions on a mesh representing a caricatural Formula 1 with $233,259$ nodes. Force and Dirichlet boundary functions are sampled following the process described in 4.3. The mesh is divided into sub-meshes of approximately 1000 nodes, resulting in a total of 234 sub-meshes, as shown in Figure 9.7a. Similar to previous results, we solve this Poisson problem using the CG-GNN-ASM, CG-ASM-LU, and CG methods until the Residual error reaches a magnitude of 10^{-12}.

Figure 9.7c displays the evolution of the residual error for the three methods, and Figure 9.7b depicts the squared error map between the CG-ASM-GNN solution and

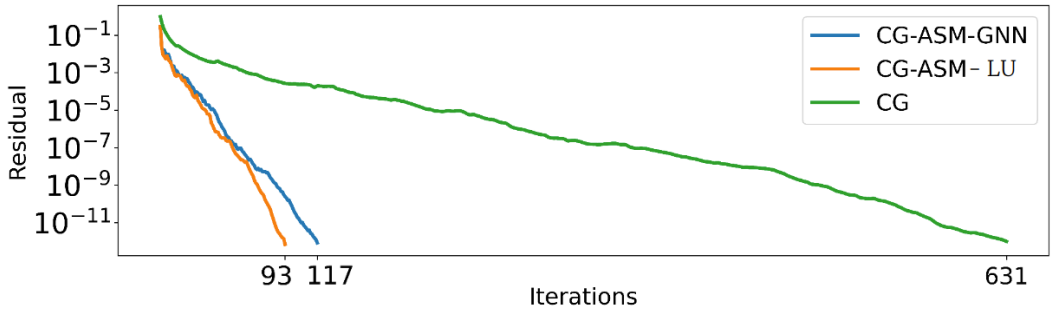
[6] the output of the preconditioner when applied to a residual vector.



(a) Mesh Partitioning



(b) Squared Error Map



(c) Residual Error

Figure 9.7: Resolution of a Poisson problem on a mesh with $233,259$ nodes using CG-ASM-GNN, CG-ASM-LU and CG methods. Figure 9.7a illustrates the mesh partitioning, and Figure 9.7b represents the squared error map between the CG-ASM-GNN solution and a "ground-truth" LU solution. Figure 9.7c depicts the evolution of the Residual error for the three methods.

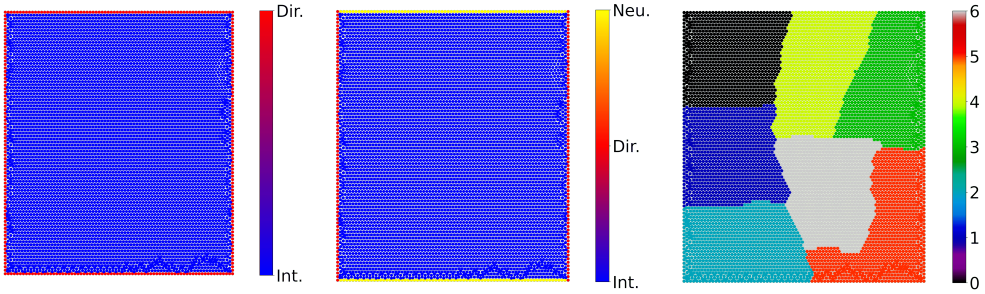
a 'ground-truth' solution computed with LU decomposition. Both figures highlight that the proposed CG-ASM-GNN method can efficiently solve this problem in 117 iterations up to the desired precision. Compared to the other methods, it requires significantly fewer iterations than the CG methods (631 iterations) and only a slightly, though not substantially, higher number of iterations than the CG-ASM-LU method (93 iterations). This is considering that all subdomains in the CG-ASM-GNN framework are solved in parallel on GPUs in 1 inference on the DSS model (i.e., all 234 sub-problems construct one batch).

Chasing the limits

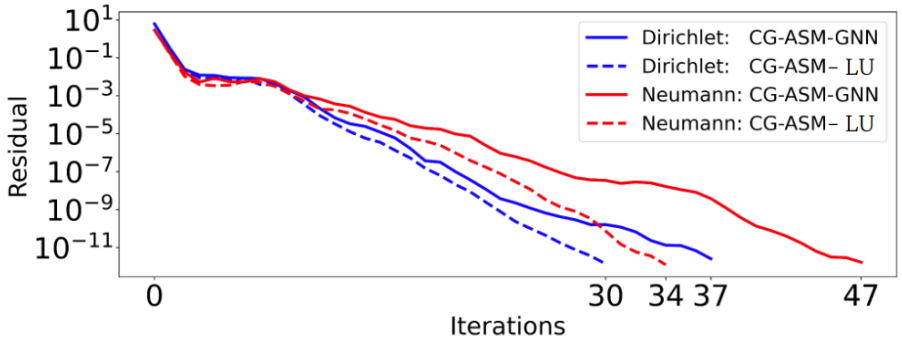
The hybrid CG-ASM-GNN method can generalize to Poisson problems with various configurations, some of which may be more challenging than those previously analyzed. By "generalize", we mean that the Conjugate Gradient method, preconditioned with ASM-GNN, will converge to the solution with the desired precision. In fact, the output of ASM-GNN does not need to be optimal (i.e., obtained from a direct solver), and approximate solutions can significantly help the Conjugate Gradient method converge faster than its original variant (i.e., CG without preconditioner). However, the number of iterations required for the CG-ASM-GNN method to converge greatly depends on the accuracy of the approximate local solutions: the more precise the resolution of the local Poisson problem, the closer the CG-ASM-GNN method will be to the optimal CG-ASM-LU method.

In this section, we address the resolution of two Poisson problems, both on a square mesh with 6707 nodes. The first, referred to as "Dirichlet", is a Poisson problem with Dirichlet boundary conditions applied to its entire boundary, as illustrated in Figure 9.8a. The force and boundary functions are generated following the process described in Section 4.3. The second, referred to as "Neumann", is the same Poisson problem but with homogeneous Neumann boundary conditions on the upper and bottom parts of the domain, as depicted in Figure 9.8b. For both problems, the mesh is partitioned into 7 sub-meshes, each with approximately 1000 nodes, as shown in Figure 9.8c. We compare the results obtained with the CG-ASM-GNN method with the CG-ASM-LU method. Both algorithms run until the Residual error reaches a magnitude of 10^{-12}.

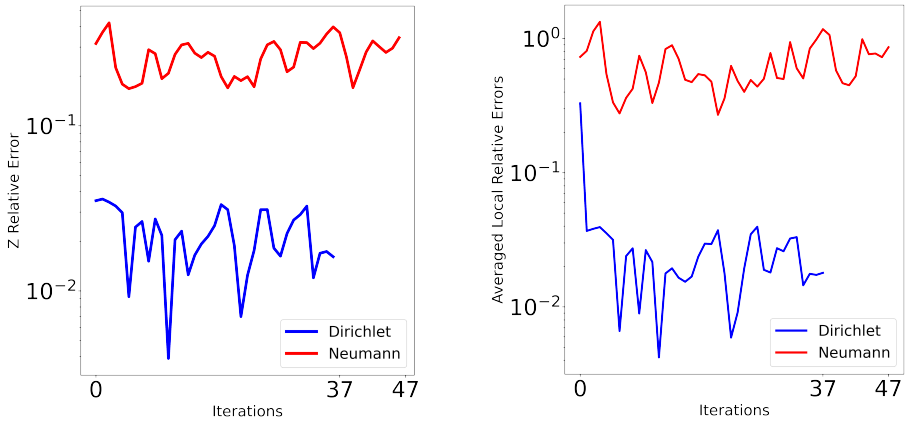
Figure 9.8d shows the evolution of the residual error for the Dirichlet (blue curves) and Neumann (red curves) problems using CG-ASM-GNN (solid lines) and CG-ASM-LU (dashed lines). Since the Neumann problem is more complex, it converges in more iterations, with 30 iterations for CG-ASM-LU on Dirichlet and 34 iterations for CG-ASM-LU on Neumann. Notably, CG-ASM-GNN achieves convergence up to the desired precision on the Neumann problem, even though some of the sub-problems are not part of the training distribution. However, the difference in iterations between CG-ASM-GNN and CG-ASM-LU for the Neumann problem is greater



(a) Dirichlet problem (b) Neumann problem (c) Mesh Partitioning



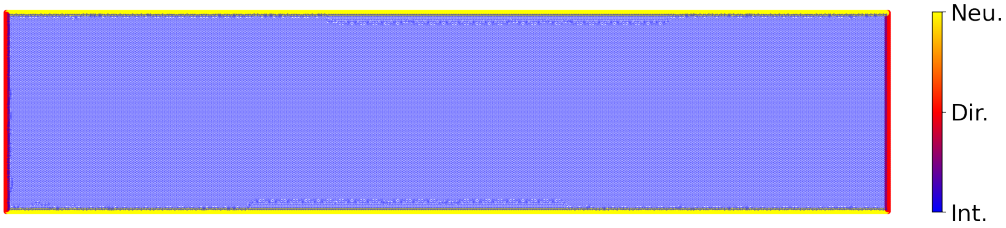
(d) Residual Error



(e) z Relative Error (f) Averaged Local Relative Error

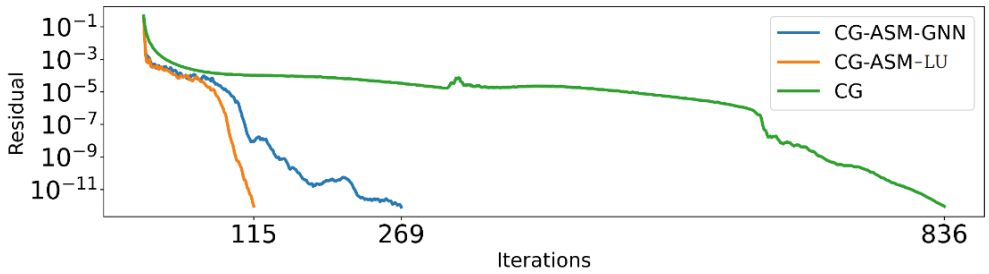
Figure 9.8: Comparison between the resolution of a Poisson problem with Dirichlet boundary conditions and a Poisson problem with mixed boundary conditions using CG-ASM-GNN and CG-ASM-LU.

than for the Dirichlet problem (13 versus 7), indicating that CG-GNN-ASM faces more difficulty in converging on the Neumann problem than in the Dirichlet case. This result is confirmed after testing the resolution of 100 multiple Poisson problems in the same configuration but with varying force and boundary functions. In this test, the



(a) Node types

(b) Mesh Partitioning



(c) Residual Error

Figure 9.9: Resolution of a Poisson problem with mixed boundary conditions using CG-ASM-GNN, CG-ASM-LU and CG methods on a mesh with $58,539$ nodes.

averaged difference in iterations between CG-ASM-GNN and CG-ASM-LU for Neumann problems is 16, whereas it is 7 for Dirichlet problems.

To further understand why, let us examine, in Figure 9.8e, the relative error between the z vector from Algorithm 7 computed with ASM-GNN and ASM preconditioners at each iteration. The results show that the solutions obtained with ASM-GNN are closer to the exact one in the Dirichlet case at a magnitude of 10^{-2}, whereas, in the Neumann case, they are solved with less accuracy at a magnitude of 10^{1}. This difference arises from the resolution of the local Poisson problem using the GNN-based model, which is less precise in the Neumann case than the Dirichlet one, due to the many out-of-samples local problems. This is illustrated in Figure 9.8f, which, at each

iteration, shows the average relative error of local problems, computed as follows: at each iteration, we compute the relative error between the solution of the GNN model and an exact LU solution for each local problem and average across all subdomains. The figure clearly demonstrates that the local problems in the Neumann case are much less accurate than in the Dirichlet case.

To conclude, Figure 9.9 displays the results obtained for solving a Poisson problem in an extreme configuration. The considered problem is a Poisson problem with mixed boundary conditions on a rectangular mesh with $58,539$ nodes. Dirichlet boundary conditions are enforced on the left and right parts of the domain, while homogeneous boundary conditions are set on the top and bottom parts of the domain. Figure 9.9a displays the mesh with different node types: blue for interior nodes, red for Dirichlet nodes, and yellow for Neumann nodes. Note that homogeneous Neumann boundary conditions intentionally cover a large part of the boundary of the global domain. The mesh is split into 59 sub-meshes, as illustrated in Figure 9.9b, resulting in sub-meshes of approximately 1000 nodes. Finally, Figure 9.9c shows the evolution of the residual error using CG-ASM-GNN, CG-ASM-LU, and CG. The first observation is that, even in this extreme configuration, CG-ASM-GNN converges to the desired precision, but it does so at the cost of a significantly larger number of iterations than the CG-ASM-LU method. However, it does not diverge and is able to solve the problem in much fewer iterations than the CG method.

9.5 . Conclusion & Discussions

This section explores the construction of ASM-GNN, a hybrid method that combines GNN-based models and Schwarz methods. ASM-GNN follows a two-level Additive Schwarz method in which the resolution of local Poisson problems is addressed thanks to a GNN model. ASM-GNN is used as a preconditioner for a Krylov method, specifically the Conjugate Gradient method, to build the hybrid solver named CG-ASM-GNN. The results demonstrate that the proposed method can generalize to various Poisson problems and consistently converge even in extreme configurations, such as the one displayed in Figure 9.9. The results presented here can be viewed as a "proof-of-concept" of the potential of such a hybrid method.

The primary goal behind developing this hybrid solver is to harness the computational power of GPUs for concurrently solving multiple sub-problems in parallel using the "per-batch" format offered in the Machine Learning area. With this method, we aim for increased speed compared to existing methods that typically rely on parallel computations on CPUs. Deliberately, we focused on comparing the number of iterations needed to solve the problem rather than providing detailed computation time. Subsequent research should focus on assessing the computational efficiency of applying the ASM-GNN preconditioner compared to other methods. However, conducting a comprehensive comparison presents challenges for several reasons:

i) choice of programming language: Here, all results are presented using our self-developed Python code, which might not be the optimal programming language for evaluating High-Performance Computing frameworks, ii) parallel computations on CPUs: While Python supports parallel computations on CPUs, it is often not optimized for this purpose, iii) appropriate optimization of the developed codes. During the current thesis writing, we are in the process of scripting the proposed method in a C++ industrial solver to thoroughly assess its performance.

Conclusion

The initial objective of this thesis was to leverage Machine Learning techniques to accelerate the resolution of incompressible Navier-Stokes equations on general unstructured meshes. To achieve this, we proposed to investigate Machine Learning methods for solving the fastidious and time-consuming Pressure Poisson problem in the context of splitting schemes. Among the various existing Machine Learning approaches, Graph Neural Networks (GNN) proved to be particularly suitable for learning on unstructured data such as meshes. However, GNN-based models often suffer from poor generalization capabilities. While these models may yield reasonably accurate results, they frequently lack guarantees, and their precision may not be sufficient to ensure the consistency of an entire CFD simulation process. As a result, this thesis explored the development of Graph Neural Network-based models with enhanced generalization capabilities and accuracy, applied to the resolution of Poisson Pressure problems in the context of CFD simulations.

Contributions

Primary contributions presented in Part II introduced two novel and original models: DS-GPS (Chapter 6) and Ψ-GNN (Chapter 7). Both of these contributions were trained by directly minimizing the residual equation of the discretized Poisson problem, drawing inspiration from the Deep Statistical Solvers (DSS) approach (Donon et al., 2020). DS-GPS and Ψ-GNN, which mainly differ regarding their architectures, were designed to address the issue of solving Poisson problems on meshes with varying numbers of nodes. In the original DSS study, the number of Message-Passing steps required for convergence was fixed and led to challenges with larger meshes. To alleviate this, DS-GPS was formulated with a recurrent architecture, iterating on a single block of Message Passing Neural Networks to propagate information through the mesh, hence resulting in a significantly lighter model. While DS-GPS can extend its number of Message-Passing steps due to its recurrent design, this iteration count still needs to be fixed by the user at different stages, whether for training or predicting. Building on the results obtained with DS-GPS, we further proposed the Ψ-GNN model, leveraging the Implicit Layer Theory to build an "infinitely" deep model that can automatically determine the number of Message-Passing steps required for convergence. This approach significantly enhanced the generalization capabilities of the model on meshes with varying numbers of nodes. Additionally, both models were designed with several additional features, and significant effort was put into the development of architectures that were less "black-box" and more respectful of physical constraints. This included explicitly treating boundary conditions as an intrinsic part of the architecture, for instance. Additional

features related to the Ψ-GNN approach demonstrated the capability of the model to adapt its iteration count based on any initially provided solution, while also providing convergence guarantees. Essentially, this research has established robust theoretical groundwork for the development of GNN-based models with improved capabilities. These models are suitable for addressing linear systems derived from discretized PDEs in graphs of various sizes. As a result, these advancements are not limited solely to solving Poisson equations, and they can be applied across various scenarios involving the discretization of PDEs.

Despite the enhanced generalization abilities of these models, challenges persist in providing accurate predictions for meshes with a very large number of nodes, such as those encountered in industrial contexts. Furthermore, their accuracy is limited to a precision linked to the capacities of the trained model, which presents a bottleneck in cases where extremely accurate predictions are crucial (e.g., for ensuring the consistency of a splitting scheme method). To tackle this issue, the last part of this thesis introduced the concept of hybrid solvers, integrating GNN models with Schwarz methods from the field of Domain Decomposition. Such hybrid approaches enable scaling these models to handle meshes with a significant number of nodes (100,000 nodes and more). In these frameworks, the GNN-based models are used to solve the multiple sub-problems, whose sizes are tailored to the optimal capabilities of the GNN models. In our optimal framework, the hybrid solver is used as a preconditioner for Krylov methods, such as Conjugate Gradient methods. As a result, the proposed approach can converge to the solution with any desired precision, thanks to the Krylov method, and the convergence of the Krylov method is significantly enhanced using the hybrid approach as a preconditioner. The novelty is that the constructed preconditioner is a fully GNN-based method. It is applicable to meshes of any size due to its Domain Decomposition approach and, more importantly, its application is done in parallel on GPUs, yielding efficient execution, in contrast to traditional methods that use CPU computations.

Perspectives & Further Works

The research presented in this thesis raises several additional questions and may lead to numerous perspectives for future research.

Firstly, although the initial objective was to accelerate the numerical resolution of Navier-Stokes equations, we deliberately did not provide any concrete results on computation times. In fact, the work presented in this thesis contrasts with studies such as those by Pfaff et al. (2020) or Sanchez-Gonzalez et al. (2020), which attempt to solve CFD problems using GNN models. These models often take meshes as input and additional information about the problem at hand, directly predicting a solution profile at a given time t. Thus, they offer significant time savings as they skip all the steps of a real numerical simulation (see Chapter 1). These models can

indeed provide a rough idea of the solution, which can be useful in some cases, such as obtaining a quick profile for design optimization. However, they often lack generalization ability, and cannot be extended to industrial cases of very large dimensions. In this thesis, our focus was on solving the Poisson Pressure equation in the context of splitting schemes, which occurs at each timestep of a numerical simulation. By doing so, the goal was to leverage Machine Learning approaches to solve the Poisson Pressure equation more rapidly than classical solvers, while retaining the guarantees provided by numerical simulators. This facilitates the integration of Machine Learning models into industrial High-Performance Computing (HPC) frameworks. As a result, the models developed in this thesis are to be directly compared with well-known and established methods for solving linear systems. This includes, for instance, direct and iterative methods as introduced at the beginning of Part III. However, to compare computation times fairly, a significant additional effort must be made. For example, the GNN-based models in Part III do not really show a performance gain due to the small meshes considered in these test cases and the already well-optimized methods used for solving such problems. On the other hand, the development of such Machine Learning models finally makes sense in Part III. The real issue with traditional methods arises when considering very large meshes, where, for example, an LU decomposition becomes impractical, or iterative methods take too long to converge. By leveraging GNN-based models as preconditioners for Krylov solvers, we could build a preconditioner whose application is essentially parallelized on GPUs while providing all guarantees of the Krylov method. For instance, in the example of "Formula 1" shape decomposed into 233 sub-domains (Figure 9.7), solving the 233 sub-problems takes about 6 seconds with the GNN-based model (i.e., one batch of 233 sub-problems solved in parallel on one GPU), compared to 14 seconds on one CPU. Obviously, these data are very experimental as they are presented using Python, are not optimized, and calculations are done sequentially on one CPU core. However, this still provides an indication of the potential performance of such hybrid methods. Therefore, as of writing this thesis, we are investigating the potential performances of such approaches by integrating them into industrial codes in C++. In that setting, it will be possible to compare the performance of the proposed hybrid approach fairly with state-of-the-art preconditioners such as Algebraic Multigrid (AMG) or Incomplete Cholesky (Saad, 2003). It is worth noting that this work might also be of great interest to the HPC community, as it is currently very challenging to translate existing codes to run on GPUs. The method developed in this thesis offers a straightforward way to leverage GPU computations by using reliable Machine Learning models in complex industrial codes.

But other questions still arise. One initial task to develop, for instance, would be to consider the construction of a dataset in relation to real Poisson Pressure equations. Indeed, the test cases used in this work were highly academic, and exploiting these models in real-world scenarios necessitates further consideration in creating the dataset. This involves aspects such as the number of global simulations in the

training set (i.e., a simulation range with varying Reynolds numbers), and how many Poisson problems to extract per simulation. Additionally, exploring the extension of these models to 3D geometries could be worthwhile. This should be easily feasible since the models only require a graph, i.e., a set of connections between nodes. However, the graph would be significantly larger, and the minimization process (i.e., through the minimization of the discretized residual) could become more challenging due to the increased conditioning of problems. One idea to enhance this could be, for example, to directly precondition the discretized residual equation used for training the models in Part II.

Finally, two last points can be discussed. Firstly, the models developed in this thesis can be applied to problems other than the Poisson problem. For example, one could consider using such methods to improve the control of adaptive methods (Adaptive Mesh Refinements). Before addressing hybrid GNN/Domain Decomposition methods, we had also considered scaling up these methods by building multigrid GNN-based models, inspired by Ronneberger et al. (2015); Gao and Ji (2019b); Lino et al. (2021a) or Liu et al. (2021). However, to be truly competitive with the hybrid method developed in this thesis, these novel multigrid models should be able to consider meshes of any size. In this sense, one could think of using the same GNN block that could act on any mesh hierarchy from fine to coarse meshes and leveraging Attention GNNs to transfer information. Such an approach, if efficient, would contrast with existing methods that generally consider a fixed number of mesh hierarchies and a fixed architecture. Another query directly related to this topic could concern investigating the use of GNNs to provide efficient pooling methods in the context of numerical simulations, where preserving crucial information from a mesh is essential for obtaining accurate approximations.

“Le mot de la fin”

With the rise of Machine Learning, the field of numerical simulation has been profoundly reshaped. Nowadays, the development of these “Artificial Intelligence” models for solving Partial Differential Equations is a highly dynamic and rapidly advancing area, to the extent that the state-of-the-art presented in this thesis may already be outdated. Reconciling both worlds (i.e., Machine Learning and Numerical Simulation) is often a complex task due to the "black-box" nature of Machine Learning methods, which contrasts with the well-established guarantees of classical equation-solving methods. In this thesis, we proposed a first step towards Machine Learning methods that are more respectful of physical constraints and ensure method convergence. This is evidenced by the latest work in this thesis, which aims at the hybridization between classical solvers and Machine Learning methods, and this, on a large scale. In particular, Machine Learning methods provide a simple and effective way to leverage parallel computations on GPUs, where the translation of well-established legacy codes into such frameworks is still particularly challenging.

Although multiple research efforts are deployed, the development of such large-scale models for complex industrial cases is still in its early stages. However, given the current advances, optimism is warranted, and the potential for the development of Machine Learning models applied to the resolution of PDEs is immense.

Bibliography

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019.

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

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. *Advances in Neural Information Processing Systems*, 2019.

S. Bai, V. Koltun, and J. Z. Kolter. Stabilizing equilibrium models by jacobian regularization. *arXiv preprint arXiv:2106.14342*, 2021.

N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee. Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence. 2 2019.

S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, et al. Petsc users manual. 2019.

G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.

P. Battaglia, R. Pascanu, M. Lai, D. Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 2016.

A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 2018.

M. Baymani, A. Kerayechian, and S. Effati. Artificial neural networks approach for solving stokes problem. *Applied Mathematics*, 2010.

W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial*. SIAM, 2000.

C. G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of computation*, 1965.

J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

J.-N. Brunet, A. Mendizabal, A. Petit, N. Golse, E. Vibert, and S. Cotin. Physics-based deep neural network for augmented reality during liver surgery. In *Medical Image Computing and Computer Assisted Intervention–MICCAI 2019: 22nd International Conference, Shenzhen, China, October 13–17, 2019, Proceedings, Part V 22*. Springer, 2019.

C. J. Burke, P. D. Aleo, Y.-C. Chen, X. Liu, J. R. Peterson, G. H. Sembroski, and J. Y.-Y. Lin. Deblending and classifying astronomical sources with mask r-cnn deep learning. *Monthly Notices of the Royal Astronomical Society*, 2019.

S. Cai, Z. Wang, S. Wang, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks for heat transfer problems. *Journal of Heat Transfer*, 2021.

S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis. Physics-informed neural networks (pinns) for fluid mechanics: A review. *Acta Mechanica Sinica*, 2022.

X.-C. Cai and M. Sarkis. A restricted additive schwarz preconditioner for general sparse linear systems. *Siam journal on scientific computing*, 1999.

Y. Cao, Z. Fang, Y. Wu, D.-X. Zhou, and Q. Gu. Towards understanding the spectral bias of deep learning. *arXiv preprint arXiv:1912.01198*, 2019.

M. B. Chang, T. Ullman, A. Torralba, and J. B. Tenenbaum. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

R. Chen, X. Jin, and H. Li. A machine learning based solver for pressure poisson equations. *Theoretical and Applied Mechanics Letters*, 2022.

L. Cheng, E. A. Illarramendi, G. Bogopolsky, M. Bauerheim, and B. Cuenot. Using neural networks to solve the 2d poisson equation for electric field computation in plasma fluid simulations. *arXiv preprint arXiv:2109.13076*, 2021.

K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, 2014.

A. J. Chorin. A numerical method for solving incompressible viscous flow problems. *Journal of Computational Physics*, 1967.

E. Chung, H.-H. Kim, M.-F. Lam, and L. Zhao. Learning Adaptive Coarse Spaces of BDDC Algorithms for Stochastic Elliptic Problems with Oscillatory and High Contrast Coefficients. *Mathematical and Computational Applications*, June 2021.

J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, 2014.

C. Cortes, M. Mohri, and A. Rostamizadeh. L2 regularization for learning kernels. *arXiv preprint arXiv:1205.2653*, 2012.

Y. L. Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson. *Handwritten Digit Recognition with a Back-Propagation Network*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli. Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next. *J Sci Comput*, July 2022.

T. A. Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 2016.

M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 2016.

M. Dissanayake and N. Phan-Thien. Neural-network-based approximations for solving partial differential equations. *communications in Numerical Methods in Engineering*, 1994.

V. Dolean, P. Jolivet, and F. Nataf. *An introduction to domain decomposition methods: algorithms, theory, and parallel implementation*. SIAM, 2015.

V. Dolean, A. Heinlein, S. Mishra, and B. Moseley. Multilevel domain decomposition-based architectures for physics-informed neural networks, Dec. 2023.

V. Dolean, A. Heinlein, S. Mishra, and B. Moseley. Finite Basis Physics-Informed Neural Networks as a Schwarz Domain Decomposition Method. In *Domain Decomposition Methods in Science and Engineering XXVII*, Cham, 2024. Springer Nature Switzerland.

B. Donon, Z. Liu, W. Liu, I. Guyon, A. Marot, and M. Schoenauer. Deep statistical solvers. *Advances in Neural Information Processing Systems*, 2020.

S. R. Dubey, S. K. Singh, and B. B. Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 2022.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 2011.

V. Dwivedi, N. Parashar, and B. Srinivasan. Distributed learning machines for solving forward and inverse problems in partial differential equations. *Neurocomputing*, 2021.

B. J. Erickson, P. Korfiatis, Z. Akkus, and T. L. Kline. Machine learning for medical imaging. *Radiographics*, 2017.

W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin. Graph neural networks for social recommendation. In *The world wide web conference*, 2019.

S. Farrens, A. Lacan, A. Guinot, and A. Vitorelli. Deep transfer learning for blended source identification in galaxy survey data. *Astronomy & Astrophysics*, 2022.

M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

K. Fukami, K. Fukagata, and K. Taira. Super-resolution reconstruction of turbulent flows with machine learning. *Journal of Fluid Mechanics*, 2019.

M. J. Gander et al. Schwarz methods over the course of time. *Electron. Trans. Numer. Anal*, 2008.

H. Gao and S. Ji. Graph u-nets. In *international conference on machine learning*. PMLR, 2019a.

H. Gao and S. Ji. Graph u-nets. In *international conference on machine learning*. PMLR, 2019b.

A. Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 2009.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 2017.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Message passing neural networks. *Machine learning meets quantum physics*, 2020.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010.

G. H. Golub and H. A. Van der Vorst. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 2000.

G. H. Golub and A. J. Wathen. An iteration for indefinite systems and its application to the navier–stokes equations. *SIAM Journal on Scientific Computing*, 1998.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Communications of the ACM*, 2020.

A. Greenbaum. *Iterative methods for solving linear systems*. SIAM, 1997.

P. M. Gresho and R. L. Sani. Incompressible flow and the finite element method. volume 2: Incompressible flow and finite element. 12 1998.

J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, et al. Recent advances in convolutional neural networks. *Pattern recognition*, 2018.

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

E. Hajiramezanali, A. Hasanzadeh, K. Narayanan, N. Duffield, M. Zhou, and X. Qian. Variational graph recurrent neural networks. *Advances in neural information processing systems*, 2019.

W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 2017.

W. L. Hamilton. *Graph representation learning*. Morgan & Claypool Publishers, 2020.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, 2015.

A. Heinlein, A. Klawonn, M. Lanser, and J. Weber. Machine Learning in Adaptive Domain Decomposition Methods—Predicting the Geometric Location of Constraints. *SIAM J. Sci. Comput.*, Jan. 2019.

A. Heinlein, A. Klawonn, M. Lanser, and J. Weber. Combining machine learning and domain decomposition methods for the solution of partial differential equations—A review. *GAMM-Mitteilungen*, Mar. 2021a.

A. Heinlein, A. Klawonn, M. Lanser, and J. Weber. Combining Machine Learning and Adaptive Coarse Spaces—A Hybrid Approach for Robust FETI-DP Methods in Three Dimensions. *SIAM J. Sci. Comput.*, Jan. 2021b.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 12 1997.

M. Horie and N. Mitsume. Physics-embedded neural networks: E (n)-equivariant graph neural pde solvers. *arXiv preprint arXiv:2205.11912*, 2022.

J.-T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon. Learning neural pde solvers with convergence guarantees. *arXiv preprint arXiv:1906.01200*, 2019.

Z. Hu, A. D. Jagtap, G. E. Karniadakis, and K. Kawaguchi. When Do Extended Physics-Informed Neural Networks (XPINNs) Improve Generalization? *SIAM J. Sci. Comput.*, Oct. 2022.

M. F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 1990.

E. A. Illarramendi, M. Bauerheim, and B. Cuenot. Performance and accuracy assessments of an incompressible fluid solver coupled with a deep convolutional neural network. *arXiv preprint arXiv:2109.09363*, 2021.

A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. *Computer Methods in Applied Mechanics and Engineering*, June 2020.

A. D. J. . G. E. Karniadakis. Extended Physics-Informed Neural Networks (XPINNs): A Generalized Space-Time Domain Decomposition Based Deep Learning Framework for Nonlinear Partial Differential Equations. *CiCP*, June 2020.

G. Karypis and V. Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

M. Kawaguti. Numerical solution of the navier-stokes equations for the flow around a circular cylinder at reynolds number 40. *Journal of the Physical Society of Japan*, 1953a.

M. Kawaguti. Numerical solution of the navier-stokes equations for the flow around a circular cylinder at reynolds number 40. *Journal of the Physical Society of Japan*, 1953b.

E. Kharazmi, Z. Zhang, and G. E. Karniadakis. hp-vpinns: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, 2021.

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

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016a.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016b.

T. N. Kipf and M. Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016c.

G. Kissas, Y. Yang, E. Hwuang, W. R. Witschey, J. A. Detre, and P. Perdikaris. Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4D flow MRI data using physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, Jan. 2020.

A. Klawonn, M. Lanser, and J. Weber. Machine learning and domain decomposition methods – a survey, Dec. 2023.

A. Klawonn, M. Lanser, and J. Weber. Learning adaptive coarse basis functions of FETI-DP. *Journal of Computational Physics*, Jan. 2024.

T. Knoke, S. Kinnewig, S. Beuchler, A. Demircan, U. Morgner, and T. Wick. Domain Decomposition with Neural Network Interface Approximations for time-harmonic Maxwell's equations with different wave numbers, Mar. 2023.

D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 2021.

A. Krishnapriyan, A. Gholami, S. Zhe, R. Kirby, and M. W. Mahoney. Characterizing possible failure modes in physics-informed neural networks. *Advances in Neural Information Processing Systems*, 2021.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 2012.

M. Kumar and N. Yadav. Multilayer perceptrons and radial basis function neural network methods for the solution of differential equations: a survey. *Computers & Mathematics with Applications*, 2011.

I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 1998.

H. P. Langtangen and A. Logg. *Solving PDEs in python: the FEniCS tutorial I*. Springer Nature, 2017.

H. P. Langtangen and K.-A. Mardal. *Introduction to Numerical Methods for Variational Problems*. 01 2019.

H. P. Langtangen, K.-A. Mardal, and R. Winther. Numerical methods for incompressible viscous flow. *Advances in Water Resources*, 2002.

M. G. Larson and F. Bengzon. *The finite element method: theory, implementation, and applications*. Springer Science & Business Media, 2013.

Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 1995.

H. Lee and I. S. Kang. Neural algorithm for solving differential equations. *Journal of Computational Physics*, 1990.

J. Lee, I. Lee, and J. Kang. Self-attention graph pooling. In *International conference on machine learning*. PMLR, 2019.

K. Li, K. Tang, T. Wu, and Q. Liao. D3M: A Deep Domain Decomposition Method for Partial Differential Equations. *IEEE Access*, 2020a.

S. Li, Y. Xia, Y. Liu, and Q. Liao. A deep domain decomposition method based on fourier features. *Journal of Computational and Applied Mathematics*, 2023.

W. Li, X. Xiang, and Y. Xu. Deep Domain Decomposition Method: Elliptic Problems. In *Proceedings of The First Mathematical and Scientific Machine Learning Conference*. PMLR, Aug. 2020b.

W. Li, M. Z. Bazant, and J. Zhu. A physics-guided neural network framework for elastic plates: Comparison of governing equations-based and energy-based approaches. *Computer Methods in Applied Mechanics and Engineering*, Sept. 2021.

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

Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, A. Stuart, K. Bhattacharya, and A. Anandkumar. Multipole graph neural operator for parametric partial differential equations. *Advances in Neural Information Processing Systems*, 2020d.

M. Lino, C. Cantwell, A. A. Bharath, and S. Fotiadis. Simulating continuum mechanics with multi-scale graph neural networks. *arXiv preprint arXiv:2106.04900*, 2021a.

M. Lino, C. Cantwell, A. A. Bharath, and S. Fotiadis. Simulating continuum mechanics with multi-scale graph neural networks. *arXiv preprint arXiv:2106.04900*, 2021b.

P.-L. Lions. On the schwarz alternating method. iii: a variant for nonoverlapping subdomains. In *Third international symposium on domain decomposition methods for partial differential equations*. SIAM Philadelphia, 1990.

W. Liu, M. Yagoubi, and M. Schoenauer. Multi-resolution graph neural networks for pde approximation. In *Artificial Neural Networks and Machine Learning–ICANN 2021: 30th International Conference on Artificial Neural Networks, Bratislava, Slovakia, September 14–17, 2021, Proceedings, Part III 30*. Springer, 2021.

D. Lucor, A. Agrawal, and A. Sergent. Physics-aware deep neural networks for surrogate modeling of turbulent natural convection, Mar. 2021.

J. Mandel* and B. Sousedík. Coarse spaces over the ages. In *Domain decomposition methods in science and engineering XIX*. Springer, 2010.

Z. Mao, A. D. Jagtap, and G. E. Karniadakis. Physics-informed neural networks for high-speed flows. *Computer Methods in Applied Mechanics and Engineering*, Mar. 2020.

S. Mishra and R. Molinaro. Estimates on the generalization error of physics informed neural networks (pinns) for approximating pdes, 2023.

F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

B. Moseley, A. Markham, and T. Nissen-Meyer. Finite basis physics-informed neural networks (FBPINNs): a scalable domain decomposition approach for solving differential equations. *Adv Comput Math*, July 2023.

M. A. Nabian, R. J. Gladstone, and H. Meidani. Efficient training of physics-informed neural networks via importance sampling. *Computer-Aided Civil and Infrastructure Engineering*, 2021.

R. A. Nicolaides. Deflation of conjugate gradients with applications to boundary value problems. *SIAM Journal on Numerical Analysis*, 1987.

S. E. Otto and C. W. Rowley. Linearly recurrent autoencoder networks for learning dynamics. *SIAM Journal on Applied Dynamical Systems*, 2019.

A. G. Özbay, A. Hamzehloo, S. Laizet, P. Tzirakis, G. Rizos, and B. Schuller. Poisson cnn: Convolutional neural networks for the solution of the poisson equation on a cartesian mesh. *Data-Centric Engineering*, 2021.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

S. V. Patankar. *Numerical heat transfer and fluid flow*. Hemisphere Publishing Corporation (CRC Press, Taylor & Francis Group), 1980.

T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

M. Raissi and G. E. Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 2018.

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

M. Raissi, Z. Wang, M. S. Triantafyllou, and G. E. Karniadakis. Deep learning of vortex-induced vibrations. *Journal of Fluid Mechanics*, Feb. 2019b.

M. Raissi, A. Yazdani, and G. E. Karniadakis. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science*, 2020.

C. Rao, H. Sun, and Y. Liu. Physics-informed deep learning for incompressible laminar flows. *Theoretical and Applied Mechanics Letters*, 2020.

F. Recknagel. Applications of machine learning to ecological modelling. *Ecological modelling*, 2001.

J. N. Reddy. *Introduction to the finite element method*. McGraw-Hill Education, 2019.

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015.

D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 1978.

J. W. Ruge and K. Stüben. Algebraic multigrid. In *Multigrid methods*. SIAM, 1987.

L. Ruiz, F. Gama, and A. Ribeiro. Gated graph recurrent neural networks. *IEEE Transactions on Signal Processing*, 2020.

D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning internal representations by error propagation, 1985.

Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*. PMLR, 2018.

A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*. PMLR, 2020.

M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher. *Benchmark computations of laminar flow around a cylinder*. Springer, 1996.

H. A. Schwarz. *Ueber einen Grenzübergang durch alternirendes Verfahren*. Zürcher u. Furrer, 1870.

K. Selim, A. Logg, and M. Larson. An adaptive finite element splitting method for the incompressible navier-stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 05 2012.

J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 1950.

D. Silvester and A. Wathen. Fast iterative solution of stabilised stokes systems part ii: Using general block preconditioners. *SIAM Journal on Numerical Analysis*, 1994.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

N. Smaoui and S. Al-Enezi. Modelling the dynamics of nonlinear partial differential equations using neural networks. *Journal of Computational and Applied Mathematics*, 2004.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 2014.

L. Sun, H. Gao, S. Pan, and J.-X. Wang. Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data. *Computer Methods in Applied Mechanics and Engineering*, 2020.

A. Taghibakhshi, T. Zaman, L. Olson, N. Nytko, and S. MacLachlan. Learning Interface Conditions in Domain Decomposition Solvers.

A. Taghibakhshi, N. Nytko, T. U. Zaman, S. Maclachlan, L. Olson, and M. West. MG-GNN: Multigrid Graph Neural Networks for Learning Multilevel Domain Decomposition Methods. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, July 2023.

W. Tang, T. Shan, X. Dang, M. Li, F. Yang, S. Xu, and J. Wu. Study on a poisson's equation solver based on deep learning technique. In *2017 IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*. IEEE, 2017.

C. Taylor and P. Hood. A numerical solution of the navier-stokes equations using the finite element technique. *Computers & Fluids*, 1973.

A. Thom. The flow past circular cylinders at low speeds. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 1933.

T. Tieleman, G. Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 2012.

A. Toselli and O. Widlund. *Domain decomposition methods-algorithms and theory*. Springer Science & Business Media, 2004.

J. Tu, G. H. Yeoh, and C. Liu. *Computational Fluid Dynamics: A Practical Approach*. Butterworth-Heinemann, USA, 2007.

K. Um, R. Brand, Y. R. Fei, P. Holl, and N. Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 2020.

J. E. Van Engelen and H. H. Hoos. A survey on semi-supervised learning. *Machine learning*, 2020.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017a.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017b.

Q. Wang, Y. Ma, K. Zhao, and Y. Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 2020.

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

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

Y. Wang. *Solving incompressible Navier-Stokes equations on heterogeneous parallel architectures*. Theses, Université Paris Sud - Paris XI, Apr. 2015.

S. Wiewel, M. Becher, and N. Thuerey. Latent space physics: Towards learning the temporal evolution of fluid flow. In *Computer graphics forum*. Wiley Online Library, 2019.

F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*. PMLR, 2019.

J.-L. Wu, H. Xiao, and E. Paterson. Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework. *Physical Review Fluids*, 2018.

Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

X. I. A. Yang, S. Zafar, J.-X. Wang, and H. Xiao. Predictive large-eddy-simulation wall modeling via physics-informed neural networks. *Phys. Rev. Fluids*, Mar. 2019.

E. Yilmaz and B. German. A convolutional neural network approach to training predictors for airfoil performance. In *18th AIAA/ISSMO multidisciplinary analysis and optimization conference*, 2017.

M. Yin, X. Zheng, J. D. Humphrey, and G. E. Karniadakis. Non-invasive inference of thrombus material properties with physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, Mar. 2021.

B. Yu et al. The deep ritz method: a deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 2018.

J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018a.

J. Zhang, T. He, S. Sra, and A. Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. *arXiv preprint arXiv:1905.11881*, 2019.

M. Zhang and Y. Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 2018.

M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, 2018b.

Q. Zhu, Z. Liu, and J. Yan. Machine learning for metal additive manufacturing: predicting temperature and melt pool fluid dynamics using physics-informed neural networks. *Computational Mechanics*, 2021.