Improving the confidence of CFD results by deep learning

Lianfa Wang

PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à MINES Paris

Improving the confidence of CFD results by deep learning

Amélioration de la confiance dans les résultats CFD par apprentissage profond

Soutenue par
Lianfa WANG
Le 18/07/2024

École doctorale n°364
Sciences Fondamentales et Appliquées

Spécialité
Mathématiques Numériques, Calcul Intensif et Données

Composition du jury :

Dr. Fabien Casenave SafranTech Paris	*Rapporteur*
Dr. Guillaume Houzeaux SuperComputing Center Barcelona	*Rapporteur*
Prof. Damien Tromeur-Dervout University Lyon 1	*Examinateur*
Dr. Hervé Guillard Inria Sophia Antipolis	*Examinateur*
Dr. Luisa Silva Centrale Nantes	*Examinateur*
Dr. Charles Moulinec USFTC Manchester	*Examinateur*
Dr. Jean-François Wald EDF Chatou	*Co-directeur de thèse*
Mr. Yvan Fournier EDF Chatou	*Co-directeur de thèse*
Prof. Youssef MESRI Mines Paris - PSL	*Directeur de thèse*

MINES PARIS | PSL

Acknowledgements

At the moment of this thesis being about to be finished, there are thousands of thoughts in my mind. I embarked on this unknown journey of PhD study in a completely disparate world than my homeland about more than three years ago driven by boldness, curiosity and youthful ambition. My heart is full of happiness and sorrow at the same time right now that this journey finally comes to the end after three years of anxiety and depression, three years of confusion and hard work, three years of growth and harvest in many aspects. If I had made any progress in these three years, it is because of those who have made this journey possible and those who have helped me all the time.

Thanks to the cooperation between Xi'An Jiaotong University and Electricite de France which gave me this PhD opportunity.

Many friends and former leaders contributed invaluable efforts to help me during the PhD application process. I would like to express my gratitude to director of EDF R&D (Beijing) Dr. Wenhui DU, Group managers Tian CHEN and Anthony DYAN.

I reserve my special respect to Dr. Jiesheng MIN. Without your recommendation and operation, I could not have gone so far. I humbly dedicate all my achievements, if I had achieved any, to you.

I am deeply grateful to my tutor Yvan FOURNIER. Thanks for having given me PhD opportunity and autonomy to research in this emerging area. I would never forget your impeccable and patient guidance in the last three years.

I am equally grateful to my supervisor, Prof. Youssef MESRI for having admitted my as a PhD student in MinesParis, PSL, for having been nice and tolerant of my ignorance. Without your countless helps and encouragements, I could not come to the end. Language is feeble to express my admiration for you.

I hold the same appreciation for my co-tutor, Dr. Jean-Francois WALD for your help, guidance and priceless discussions that we had.

I would like thank the reporters, Dr. Guillaume Houzeaux and Dr. Hervé Guillard, for your meticulous proofreading and invaluable suggestions.

I have always felt lucky to be in the same office with Federico BARAGLIA who is always kind, optimistic, cheering, versatile and helpful. It will become a pity for me in the future that we did not have another hiking together. If there is one reason that one day I am going to regret that I have left France, it would be I having lost a friend like you.

Many other friends have helped me a lot in both quotidian life and academic research. I owe my appreciations to Antoine BOUFFARD, Mathiew GUINGO, Li MA, Benoit de LAAGE de MEUX, Franchine NI, Samuel GAUCHER, Luc BERTOLOTTI, Jérome BONELLE, Javier ANEZ, Thomas FONTY, Chai KOREN, Pierre VILANOBA.

I also thank other colleagues in Groupe I8A, MFEE, Jérome LAVIEVILLE, Vincent STOBIAC, Nicholas MERIGOUZ, Emmanuel LATOUR.

Contents

List of Figures

List of Tables

1 Introduction

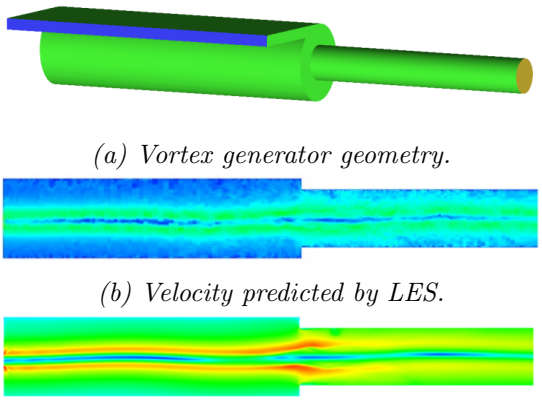
Contents

1.1 Context

Turbulent fluid flows play an important role in many industrial processes, including most large scale power generation means. Detailed understanding of turbulent flows is not only important in the context of the optimization of production means such as wind farms, but essential for nuclear energy production in pressurized water reactors, where thermal hydraulics plays a central role in safety studies and demonstrations. In the regulatory context of nuclear power production, Validation, Verification, and uncertainty Quantification (VVUQ) aspects must be at the state of the art when fluid flow simulations are involved. As the flows involved in both normal and accidental conditions can be very complex, confidence in simulation results relies on both identifying an ranking the expected flow features, and ensuring the appropriate physical flow features are handled by the models used. As local CFD simulations are very resource intensive, simpler models are preferred when sufficient, and more complex ones where necessary. This requires good expert knowledge of how various modeling options predict different phenomena. Even though the governing equations for fluid flows have been developed by Navier and Stokes for more than a century, its modelling was not rendered possible until the 1970s by the invention of many numerical methods and turbulence models, such as $k - \varepsilon$ model, in computational fluid dynamics (CFD). Even after another half century of exponential development of the high performance computing resources, the direct numerical simulation (DNS) of turbulent flows remains out of reach for the industrial applications owing to its stochastic nature and the wide range of temporal and spatial scales. To overcome the computational burden induced by the turbulent structures with universal characteristics at small scales, large eddy simulation (LES) was proposed to solve the large scale turbulent structures while modelling the small scale ones, which significant reduces the computational costs compared to DNS but not cheap enough to be massively employed in industry.

As a result, the Reynolds-Averaged Navier-Stokes (RANS) simulations supplemented by turbulence closure models still dominate the engineering design, performance analysis and optimization of the industrial equipment. Even though different turbulence models are indiscriminately applied to simulate various flow phenomena, their accuracy in the

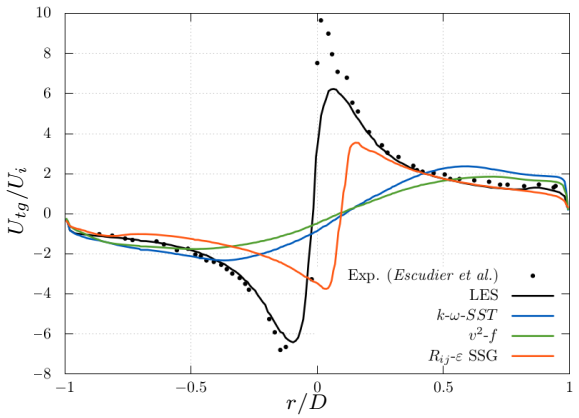
specific scenario is still questionable. For example, the standard $k-\varepsilon$ model is valid in the simple flows but not in complex flows such as the adverse pressure gradient flows, separation and reattachment, jet impingement. The standard $k-\varepsilon$ turbulence model overestimates the turbulent kinetic energy level at the stagnation point. A modified $k-\varepsilon$ turbulence model with linear production term can avoid this anomaly. While Wilcox's $k-\omega$ model is superior than $k-\varepsilon$ model for the prediction of flow with adverse pressure gradient, the eddy viscosity in the free stream is very sensitive to the small variation of ω value which leads to the proposal of Menter's shear stress transport (SST) $k-\omega$ model. The linear eddy-viscosity models fail to predict the secondary flow in the square duct flow case, but the quadratic $k-\varepsilon$ model and second-order Reynolds Stress models (RSM) can reproduce this secondary flow structure due to anisotropy. The RSMs solving six transport equations for the components of Reynolds stresses are deemed more suitable for anisotropic flows such as swirling flows and secondary flows at the cost of much higher computational expense than the first-order turbulence models. As shown in Fig. 1.1, the experimental data shows that the swirling flow inside the vortex generator has the highest swirling magnitude near the cylinder center indicated by the peak tangent velocity. As expected, LES well predicts the positions of the peak values and the general profiles of the tangential velocity. As a comparison to the failure of $k-\omega$ SST on predicting the tangent velocity profile in the center region, $R_{ij}-\varepsilon$ SSG can well capture the strong rotation in the center region though the predicted peak magnitude is lower than the experimental data. Despite its superiority compared with the first-order turbulence models, RSMs also have the drawbacks such as the stability problem with regard to the mesh quality.



(a) Vortex generator geometry.

(b) Velocity predicted by LES.

(c) Velocity predicted by $R_{ij}-\varepsilon$ SSG.



(d) Tangent velocity profiles.

Figure 1.1: Tangent velocity profiles predicted by different turbulence models in vortex generator.

1.2 Objectives and strategy

As already shown in the previous section, some turbulence models are better adapted for various flow phenomena. How to accurately simulate each flow phenomenon requires not only the knowledge about the underlying physics but also the expertise of the code. At EDF, various simulation tools have been used and developed for many years, whether for less detailed but complete system tools, specialized "component level" models, or general purpose CFD models when detailed flow modeling is needed. Over decades of development of the finite volume method CFD solver code_saturne, the I8A group of the

département mécanique des fluides, energie et environnement (MFEE) as the development team accumulated around 100 verification and validation cases for diverse flow phenomena. Before the release of each new version of code_saturne, these cases are carefully run to ensure the correctness of the coding and validity of the models. For each case, the simulation results obtained with different turbulence models and numerical methods are compared with the experimental data. These cases can form a potential optimal model configuration pool as the reference to better simulate corresponding flow phenomenon. As a long term goal, integrating this expertise into the code to make the simulation more automatic or to give suggestions to the user can improve user's confidence on the result.

Deep learning (DL) or machine learning (ML) can be a promising direction to solve this challenging task. Recently, we have witnessed a great success of ML on the pattern recognition in images and audios such as YOLO, discovering the potential relationship from biomedical scans and observations to the pathology, human knowledge integration such as ChatGPT, recommendation system over films and commercial products, generated content sush as DALL·E. Nowadays, all these available ML techniques to extract knowledge from ever-growing data have become the alternatives for the fluid dynamics community to understand the underlying mechanics. Enthralled by its excellent capability of fitting the non-linear relationship between input and output, many CFD researchers have embraced MLs to address various tasks, such as turbulence models closure, model form uncertainty quantification, flow fields super-resolution, accelerating or completely replacing the cumbersome CFD solvers and so on. However, these models are not in general predictive enough to constitute an alternative to general CFD simulation. These tools show more short term promise where used to represent complex physical property behavior or as a component of turbulence or phase change models, but even there, the lack of explainability of associated simulation results would make their use difficult to justify in a nuclear safety regulatory context. Whereas when used to assist an engineer in the identification of flow types, so as to help verifying and checking model choices, while keeping the human in the loop, AI models could increase engineer productivity and confidence in simulation results even when using classical simulation tools.

The general roadmap to realize this long term goal is shown in Fig. 1.2. The user sets the initial configuration of the CFD and launches the calculation. A machine learning (ML) algorithm identifies whether a certain flow phenomenon exists inside the initial CFD result based on the selected features characterizing the flow phenomenon. This ML model can be trained on the dataset formed by the experimental data and high-fidelity CFD data, especially code_saturne validation cases. Industrial flows can combine several different flow phenomena, which may each have a small to dominant aspect on the overall flow. For example, buoyancy phenomena can drive the flow when no other forces are present, lead to stratifications where the velocity is not strong, or be a negligible factor when the flow is driven by stronger convection. Presence of vortices related to turbulence may lead to mixing and prevent a stratification from appearing, and vortices may be generated both by geometric features such as backward-facing steps or other obstactles even in low velocity conditions, and by turbulence related to shear in the flow. In our case, we will start by trying to identify a few different common separate flow phenomena, such as the presence of large vortices or of stratification, and if that detection is successful, work on detecting combinations of these features. At this stage, as most basic phenomena can be present in 2D flows, the lower resulting computational cost, associated turnaround time, and data volume associated with these flows can allow for more experimentation and finer analysis. Then, as some features such as secondary vortices may be detected by

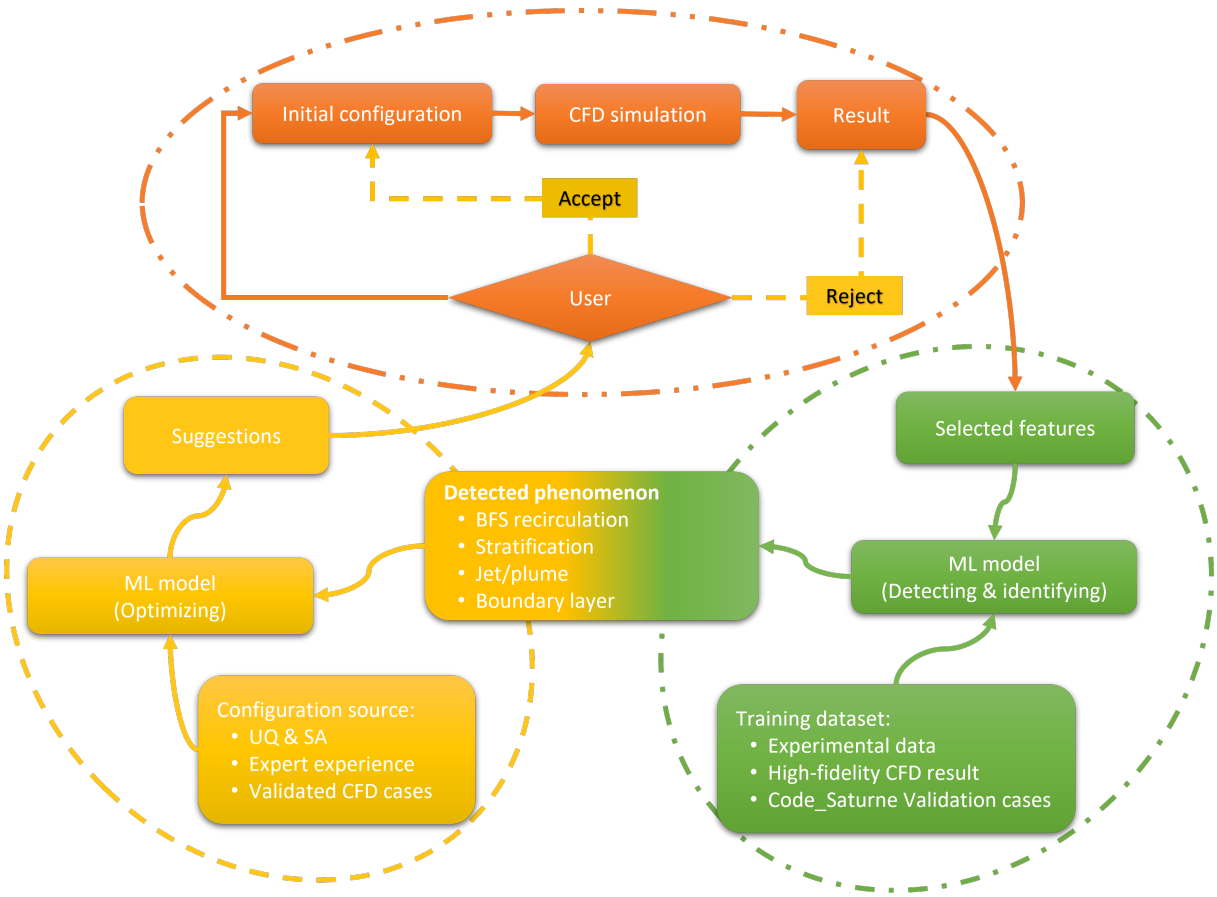


Figure 1.2: General strategy to give suggestions on CFD configuration.

some modeling options, and not reproduced by some simplified turbulence models, we will test and adapt the detection to at least one such example. If multiple flow phenomena exist, the phenomena identification and ranking table (PIRT) method can be used to find out the one that has the greatest impact on the result accuracy.

Based on the detected flow phenomenon, the second machine learning algorithm trained on the dataset consisting of the configuration knowledge for various flow phenomena can propose to the user the optimal model configuration. The candidate configurations for the flow phenomena can come from different sources, such as uncertainty quantification and sensitivity analysis of turbulence models and numerical schemes, expert experience and code_saturne verification and validation cases. The user finally decides whether to accept the suggestion, modify the model configuration and relaunch the simulation or to reject the proposal and keep the initial result.

This manuscript focuses on the first step indicated by the green circle in the figure - flow phenomena identification. Since local features can not fully characterize certain flow phenomena which normally have a spatial distribution, it is natural to use the MLs which are able to process data on meshes. Thus we firstly use convolutional neural network (CNN) to identify flow structures. Our objective is to detect the presence of a specific flow phenomenon in the CFD result and distinguish it from the other regions which is very similar to the object detection/segmentation in computer vision (CV). Different from the object detection/segmentation in CV, there are several difficulties to overcome:

- **Contradiction between small data-set and broad generality.** On the contrary of the presence of large public dataset in CV, such as MNIST, CIFAR-100,

ImageNet, etc, no widely accepted data-set exists in CFD community. There will not be such a dataset coming out in the near future due to several facts. Different input features are needed for corresponding task-specific algorithm. Higher dimensionality of CFD results data increases the requirement for the data storage and computation resources. The vast range of influencing parameters, such as different Re numbers, geometries, mesh topologies, turbulence models, make the large data-set impossible. Consequently, all the ML applications in fluid dynamics are trained on the small data-set and lose generality once applied to outlier cases. We would like to find an algorithm with a good generalizability to scenarios of different Re numbers, turbulence models, geometry and mesh topologies.

- **Adaptability to unstructured meshes.** Most of the ML flow detection algorithms are not suitable for unstructured meshes which are usually used for industrial cases to fit the complex geometries. Deforming the mesh or interpolate data onto structured meshes is plausible remedy for theses methods but unrealistic and very complex for intricate geometries. We extend the CNN paradigm to graph neural networks (GNNs) to resolve this problem.

- **Computational efficiency.** Computational cost becomes significant for GNNs compared with CNNs especially for 3D cases with large quantity of cells. An appropriate ML model should not bring excessive cost to the simulation.

1.3 Manuscript organisation

This manuscript is organised as follows:

Chapter 2 introduces the basic knowledge of turbulent flows and its simulation. The Reynolds-averaged Navier-Stokes (RANS) simulations with the commonly used turbulence models are described firstly. The main numerical methods employed in the finite volume method (FVM) solver code_saturne are detailed including time and space discretisation methods, wall modeling models and the algebraic multigrid method.

Chapter 3 introduces the fundamental concepts of machine learning such as the general procedure in the supervised training, the typical terminologies in the neural networks are introduced at the beginning. Two main machine learning algorithms used in this research, CNN and GNN, are described.

Chapter 4 is a review of the current applications of MLs in CFD domain. Although the current work is confined to identification of flow phenomena inside the CFD results, the literature also covers more applications of MLs such as turbulence model closure, surrogate modelling, flow field super-resolution in the hope that these studies can enlighten the following researches.

Chapter 5 details how to apply CNN to identify flow phenomena on Cartesian meshes. The factors that may influence the detecting accuracy are investigated firstly, such as data interpolation methods and mesh refinement. The feasibility of using CNN to identify the vortex shedding behind the backward-facing step by CNN is shown. A vortex auto-labelling algorithm based on the random walking on the directed graph is proposed to label the vortexes on 2D mesh. The results show that CNN can efficiently and accurately locate the vortexes based on the rudimentary velocity field. The CNN model can be easily extended to detect other flow phenomena such as thermal stratification by slightly modified the architecture.

Chapter 6 extends the methodology of identifying flow phenomena from Cartesian meshes in the previous chapter to unstructured meshes by using GNN. Three graph convolution kernels, the direct counterparts of the conventional CNN convolution kernel, are tested on identifying 2D vortexes. The computational cost is more prominent in the GNNs and can be addressed by using U-Net architecture on the graph hierarchy generated by the algebraic multigrid method in the code_saturne. A new GNN kernel is proposed which is more suitable to identify the flow phenomena on graphs derived from CFD meshes and more computational efficient. The generalization of the proposed framework to 3D flow phenomena is exemplified on detecting the 3D vortex inside the vortex generator case at the end.

Chapter 7 concludes the manuscript and points out the future perspectives.

2 Fundamental knowledge of computational fluid dynamics

Contents

The turbulent flow characteristics and its governing equations are firstly described in this chapter. Though different strategies to simulate turbulent flows exist, we mainly focus on simulations using Reynolds-Averaged Navier-Stokes (RANS) equations and turbulence models. Thus RANS equations and the most used turbulence models are reviewed. The main numerical methods employed in code_saturne are introduced briefly at the end.

2.1 Turbulent flow and its governing equations

2.1.1 Turbulent flow

Most of the engineering flow problems focus on the turbulent flow where the velocity and other properties vary in a random and chaotic way. The velocity measurement at a point in the turbulent flow is shown in Fig. 2.1.

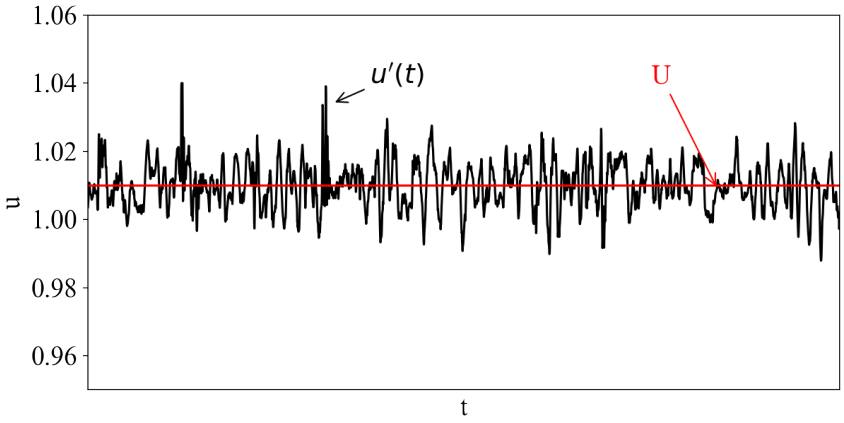


Figure 2.1: Velocity fluctuation in turbulent flow.

The fluctuation of the velocity in the turbulent flow is due to the existence of the eddies of a wide range of length, time and velocity scales. The length (l), velocity (u) scales of the largest eddies are of the same order as the velocity scale U and length scale L of the mean flow. Thus the Reynolds number for the large eddies $Re_l = ul/\nu$ is large indicating the largest eddies are dominated by inertia effects and viscous effects are negligible. The fluctuation of the largest eddies are different in different directions and affected by the boundary condition, thus anisotropic.

The largest eddies interact with and obtain energy from the mean flow through vortex stretching. The large eddies are unstable and break up. The energy is transferred from large eddies to smaller and smaller scales till the smallest scales where the turbulent energy is dissipated into thermal internal energy. This process is termed as energy cascade.

The characteristic scales of the smallest turbulent motions are called the Kolmogorov scales including the length (η), time (τ_η) and velocity (u_η) scales formed by the rate of dissipation of turbulent energy ε and kinematic viscosity ν:

$$\eta \equiv \left(\frac{\nu^3}{\varepsilon}\right)^{1/4}, \tag{2.1}$$

$$\tau_\eta \equiv \left(\frac{\nu}{\varepsilon}\right)^{1/2}, \tag{2.2}$$

$$u_\eta \equiv (\nu\varepsilon)^{1/4}. \tag{2.3}$$

The Reynolds number based on the Kolmogorov scales is unity:

$$Re_\eta = \frac{\eta u_\eta}{\nu} = 1, \tag{2.4}$$

indicating the inertial and viscous affects are of equal strength. The ratio of the length, time and velocity scales $\eta,\ \tau_\eta,\ u_\eta$ of the smallest eddies to the length, time and velocity scales $l,\ T, u$ of the largest eddies are estimated as:

$$\frac{\eta}{l} \approx Re_l^{-3/4},\ \frac{\tau_\eta}{T} \approx Re_l^{-1/2},\ \frac{u_\eta}{u} \approx Re_l^{-1/4}, \tag{2.5}$$

which means the length and time scales of the smallest eddies decrease as the large-eddy Reynolds number Re_l increases. The smallest eddies are isotropic, their directionality is smeared out by the diffusive action of viscosity.

2.1.2 Governing equations

The incompressible Newtonian fluid flow is governed by the continuity equation and momentum conservation equations (Navier-Stokes equations):

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{2.6}$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + S_{M_i}. \tag{2.7}$$

where $t, p,\ \rho,\ \nu,\ S_M$ are the time, pressure, fluid density, kinematic viscosity, momentum source term, respectively, and the transport equation for the possible scalar ϕ:

$$\frac{\partial \phi}{\partial t} + u_i \frac{\partial \phi}{\partial x_i} = \Gamma_e \frac{\partial^2 \phi}{\partial x_i \partial x_i} + S_i \phi + S_e, \tag{2.8}$$

where Γ_e represents the effective diffusivity of ϕ, $S_i\phi$ and S_e are the implicit and explicit parts of the source term, respectively.

2.1.3 Reynolds-averaged Navier-Stokes equations

The Reynolds decomposition decomposes instantaneous velocity $u(t)$ into the steady mean value U and the fluctuating component $u'(t)$:

$$u(t) = U + u'(t), \tag{2.9}$$

by time averaging:

$$U = \frac{1}{\Delta t}\int_0^{\Delta t} u(t)dt, \tag{2.10}$$

where the time duration Δt is selected to be larger than the time scale of the slowest variation. The time average of the fluctuation part $u'(t)$ is zero:

$$\overline{u'(t)} = \frac{1}{\Delta t}\int_0^{\Delta t} u'(t)dt = 0. \tag{2.11}$$

The time averages of instantaneous variables $\phi = \Phi + \phi'$ and $\psi = \Psi + \Psi'$ and their summation, derivatives and integrals are summarized here:

$$\begin{array}{ccccc} \overline{\phi'} = \overline{\psi'} = 0, & & \overline{\Phi} = \Phi, & \overline{\frac{\partial \phi}{\partial x}} = \frac{\partial \Phi}{\partial x}, & \overline{\int \phi dx} = \int \Phi dx, \\ \overline{\phi + \psi} = \Phi + \Psi, & \overline{\phi\psi} = \Phi\Psi + \overline{\phi'\psi'}, & & \overline{\phi}\overline{\Psi} = \Phi\Psi, & \overline{\phi'}\overline{\Psi} = 0. \end{array}$$

Applying the time averaging process to every term in the Navier-Stokes equations we obtain the Reynolds-averaged Navier-Stokes (RANS) equations for incompressible Newtonian flow:

$$\frac{\partial U_i}{\partial x_i} = 0, \tag{2.12a}$$

$$\frac{\partial U_i}{\partial t} + U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial P}{\partial x_i} + \nu \frac{\partial^2 U_i}{\partial x_j \partial x_j} - \frac{\partial \overline{u_i' u_j'}}{\partial x_j} + S_{M_i}. \tag{2.12b}$$

The extra terms at the right hand side of the RANS momentum equations $\overline{u_i' u_j'}$ are called Reynolds stresses which contains six unknown variables and should be modeled to close the equation set.

2.2 Turbulent flow simulation strategies

A typical energy cascade of turbulent flow is shown in Fig. 2.2. The left end part of this turbulence energy spectrum plot contains the large-scale vortexes which are influenced by the geometry, extract the energy from the mainstream flow and contain the majority of the total energy. The right end part of the spectrum contains the smallest eddies at Kolmogorov scale which account for only a small portion of the total turbulent energy. Between the energy-containing range and the dissipation range there is an inertial subrange where the eddies are determined by inertial effects, accept the energy transferred from the larger eddies in the energy-containing range and further pass it down to the smaller eddies in the dissipation range. There are three levels of simulating the turbulent flow depending on to which scale of turbulence it solves: direct numerical simulation (DNS), large eddy simulation (LES) and simulation of Reynolds-averaged Navier-Stokes (RANS) equations together with turbulence models.

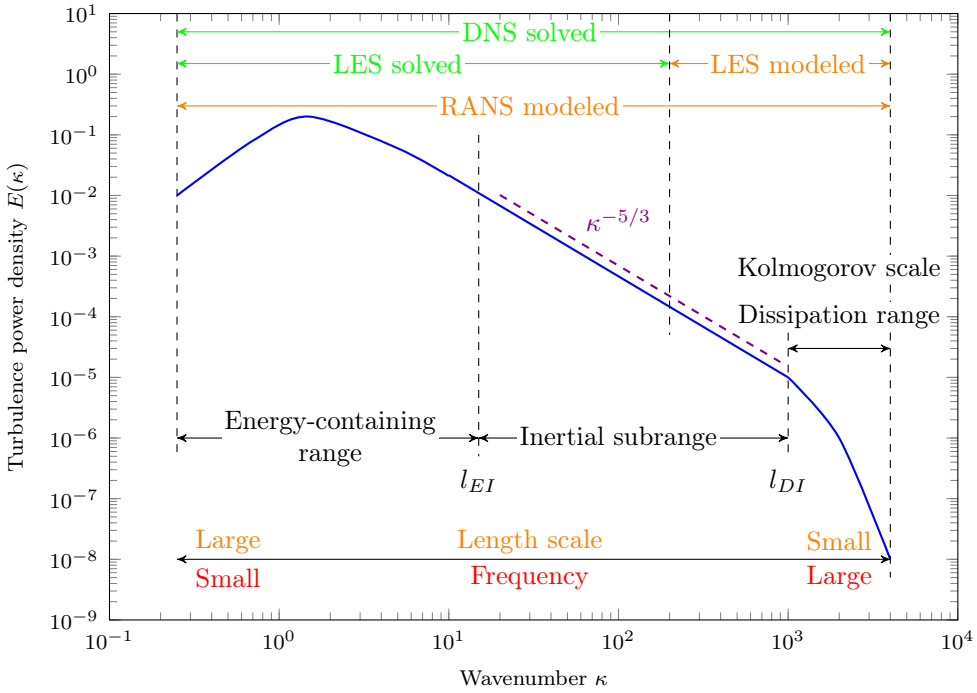


Figure 2.2: Turbulence energy wavenumber spectrum.

Direct numerical simulation. The Navier-Stokes equations are solved on spatial meshes sufficiently fine with sufficiently small time step to resolve the smallest eddies at Kolmogorov scale. According to Eq. 2.5, both the length scale and time scale of the smallest eddies increase as the Reynolds number Re_l increases. As a result, the total computational cost of a DNS of three dimensional flow is proportional to Re_l^3 which constrains the DNS to the flows of low-Reynolds-numbers with simple geometry at the current stage and even in a foreseeable future.

Large eddy simulation. Instead of spending most of the computational effort on resolving the small-scale motions with universal character like DNS, LES directly computes the most energetic and anisotropic large-scale motions and models the effect of the small-scale motions. LES uses the filter operation to decompose the velocity $\mathbf{u}(x,t)$ into the sum of a filtered/resolved component $\overline{\mathbf{U}}(x,t)$ and a residual/subgrid-scale (SGS) component $\mathbf{u}'(x,t)$. The equations for the filtered velocity component $\overline{\mathbf{U}}(x,t)$ derived from

Navier-Stokes equations containing the SGS stress tensor are resolved. An eddy viscosity model (EVM) is used for the modeling of the SGS stress tensor to close the equations. Though much computational economical compared with DNS, LES is much expensive than Reynolds stress models, the most complex RANS models, due the facts that the transient calculation and a small filter width located in the inertial subrange, embodied in the small cell size, are needed to resolve the transient energetic-containing motions. Consequently, the applications of LES in the industry is still limited despite the recent rapid development of high performance computing resources.

RANS simulation. The last category of simulating turbulent flows solves the Reynolds-averaged Navier-Stokes equations 2.12 and turbulence models - additional transport equations to model the Reynolds stresses. Since RANS simulation models the turbulent motions of all ranges, it is the cheapest method to obtain the time-averaged mean flow characteristics which are the most important information for most of the engineering design. A large diversity of turbulence models makes the RANS simulation the most used method for a broad range of flow phenomena.

2.3 Turbulence models

Many turbulence models have been proposed to close the RANS equations. According to the number of the extra transport equations to be solved, these models can be divided into zero-, one-, two- and seven-equation turbulence models. These model can be grouped into two categories: eddy viscosity models (first-order models) and second-order models, as shown in Table 2.1.

Table 2.1: Categorisation of turbulence models.

Type	Nb. extra transport equations	Model
Eddy viscosity model	Zero	Mixing length model
	One	Spalart-Allmaras model
	Two	$k-\varepsilon$ model
		$k-\omega$ model
Second-order model	Seven	Reynolds stress model

Despite their computational efficiency, the zero- and one-equation models are limited to the specific type of flows, due to their incompleteness (from to their simplifying assumptions). For example, the mixing length model (zero-equation) is only suitable for thin shear layers and the Spalart-Allmaras model (one-equation) is limited to the aerodynamics. In this work, these models are not used and thus omitted here. The seven-equation models (i.e. Reynolds stress models (RSM)) which solve the transport equations for each of six component of Reynolds stresses and for another variable to provide the length or time scale, such as ε or ω are the most complex turbulence models, thus very computationally heavy. One of the most used RSMs, RSM Speziale, Sarkar and Gatski (SSG) model, is introduced at the end of this section.

Two-equation turbulence models, such as $k-\varepsilon$, $k-\omega$, are the most used models in the industrial turbulent flow simulations thanks to their good balance between accuracy, computational efficiency and wide range applicability. Three most used two-equation turbulence models are: standard $k-\varepsilon$, Wilcox $k-\omega$ and Menter shear stress transport (SST) $k-\omega$ models are introduced firstly in this section. These models adopt Boussinesq

eddy viscosity assumption which assumes that the Reynolds stresses are proportional to the rate of deformation of mean flow by making analogy to the viscous stresses:

$$-\overline{u_i' u_j'} = 2\nu_t S_{ij} - \frac{2}{3} k \delta_{ij}, \tag{2.13}$$

where ν_t is the kinetic turbulent eddy viscosity which should be modeled, S_{ij} is strain rate tensor:

$$S_{ij} = \frac{1}{2}\left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right), \tag{2.14}$$

$k = \frac{1}{2}\overline{u_i' u_i'}$ is the turbulent kinetic energy per unit mass, and δ_{ij} is Kronecker delta:

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases} \tag{2.15}$$

2.3.1 Standard $k - \varepsilon$ model

The standard $k - \varepsilon$ model proposed by Launder and Sharma (1974) solves the transport equations for turbulent kinetic energy and its dissipation rate ε:

$$\begin{aligned} \frac{\partial k}{\partial t} + U_i \frac{\partial k}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_i}\right] + 2\nu_t S_{ij} \cdot S_{ij} - \varepsilon, \\ \frac{\partial \varepsilon}{\partial t} + U_i \frac{\partial \varepsilon}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_\varepsilon}\right)\frac{\partial \varepsilon}{\partial x_i}\right] + C_{1\varepsilon}\frac{\varepsilon}{k} 2\nu_t S_{ij} \cdot S_{ij} - C_{2\varepsilon}\frac{\varepsilon^2}{k}, \end{aligned} \tag{2.16}$$

where the kinetic eddy viscosity ν_t is defined as :

$$\nu_t = C_\mu \frac{k^2}{\varepsilon}. \tag{2.17}$$

Five empirical adjustable constants are:

C_μ	σ_k	σ_ε	$C_{1\varepsilon}$	$C_{2\varepsilon}$
0.09	1.00	1.30	1.44	1.92

2.3.2 Wilcox $k - \omega$ model

Another widely used two-equation turbulence model is $k-\omega$ model Wilcox (1993); Wilcox et al. (1998); Wilcox (1994) which uses the turbulence frequency $\omega = \varepsilon/k$ as the second variable. The kinetic eddy viscosity is defined as $\nu_t = k/\omega$. The transport equations for k and ω are defined as:

$$\begin{aligned} \frac{\partial \rho k}{\partial t} + U_i \frac{\partial k}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_i}\right] + P_k - \beta^* k\omega, \\ \frac{\partial \omega}{\partial t} + U_i \frac{\partial \omega}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_i}\right] + \alpha S_{ij} \cdot S_{ij} - \beta\omega^2, \end{aligned} \tag{2.18}$$

where P_k is the production of k:

$$P_k = 2\nu_t S_{ij} \cdot S_{ij}, \tag{2.19}$$

and the model constants are:

σ_k	σ_ω	β^*	α	β
2.0	2.0	0.09	0.553	0.075

2.3.3 Menter SST $k-\omega$ model

Menter (1994) proposed a shear stress transport (SST) $k-\omega$ model to retain the robust and accuracy of the Wilcox $k-\omega$ model in the near wall region and to take advantage of the freestream independence of the $k-\varepsilon$ in the fully turbulent region. The original $k-\omega$ model reads:

$$\begin{aligned} \frac{\partial k}{\partial t}+U_i\frac{\partial k}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[(\nu+\sigma_{k1}\nu_t)\frac{\partial k}{\partial x_i}\right]+P_k-\beta^* k\omega, \\ \frac{\partial \omega}{\partial t}+U_i\frac{\partial \omega}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[(\nu+\sigma_{\omega 1}\nu_t)\frac{\partial \omega}{\partial x_i}\right]+\frac{\alpha_1}{\nu_t}P_k-\beta_1\omega^2, \end{aligned} \tag{2.20}$$

The original $k-\varepsilon$ model is transformed into $k-\omega$ formulation:

$$\begin{aligned} \frac{\partial k}{\partial t}+U_i\frac{\partial k}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[(\nu+\sigma_{k2}\nu_t)\frac{\partial k}{\partial x_i}\right]+P_k-\beta^* k\omega, \\ \frac{\partial \omega}{\partial t}+U_i\frac{\partial \omega}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[(\nu+\sigma_{\omega 2}\nu_t)\frac{\partial \omega}{\partial x_i}\right]+\frac{\alpha_2}{\nu_t}P_k-\beta_2\omega^2+\underbrace{\frac{2\sigma_{\omega 2}}{\omega}\frac{\partial \omega}{\partial x_k}\frac{\partial k}{\partial x_k}}_{\text{Cross diffusion}}. \end{aligned} \tag{2.21}$$

Eqs. 2.20 and 2.21 are multiplied by F_1 and $(1-F_1)$, respectively, and are added correspondingly to give the $k-\omega$ SST model:

$$\begin{aligned} \frac{\partial k}{\partial t}+U_i\frac{\partial k}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[(\nu+\sigma_{k}\nu_t)\frac{\partial k}{\partial x_i}\right]+P_k-\beta^* k\omega, \\ \frac{\partial \omega}{\partial t}+U_i\frac{\partial \omega}{\partial x_i} &= \frac{\partial}{\partial x_i}\left[(\nu+\sigma_{\omega}\nu_t)\frac{\partial \omega}{\partial x_i}\right]+\frac{\alpha}{\nu_t}P_k-\beta\omega^2+2(1-F_1)\frac{\sigma_{\omega 2}}{\omega}\frac{\partial \omega}{\partial x_k}\frac{\partial k}{\partial x_k}, \end{aligned} \tag{2.22}$$

where the production of k is:

$$P_k=\left(\nu_t S_{ij}-\frac{2}{3}k\delta_{ij}\right)\frac{\partial u_i}{\partial x_j}. \tag{2.23}$$

Let ϕ represent any constant in Eqs. 2.22, and ϕ_1 and ϕ_2 the corresponding constants in Eqs. 2.20 and 2.21, respectively. Then, the relation between them is:

$$\phi=\phi_1 F_1+\phi_2(1-F_1). \tag{2.24}$$

The constants ϕ_1 in Wilcox model are

α_{k1}	$\sigma_{\omega 1}$	β_1	γ_1
0.5	0.5	0.075	5/9

The constants ϕ_2 in standard $k-\varepsilon$ model are:

α_{k2}	$\sigma_{\omega 2}$	β_2	γ_2
1.0	0.856	0.0828	0.44

The blending function F_1 between $k-\omega$ model and $k-\varepsilon$ model is defined as:

$$F_1=tanh(\arg_1^4), \tag{2.25}$$

$$\arg_1 = min\left[max\left(\frac{\sqrt{k}}{\beta^* \omega y}, \frac{500\nu}{y^2\omega}\right), \frac{4\rho\sigma_{\omega 2}k}{CD_{k\omega}y^2}\right], \tag{2.26}$$

where $CD_{k\omega}$ is positive part of the cross diffusion term:

$$CD_{k\omega} = max\left(\frac{2\rho}{\sigma_{\omega 2}}\frac{\partial\omega}{\partial x_i}\frac{\partial k}{\partial x_i}, 10^{-20}\right). \tag{2.27}$$

The eddy viscosity is defined as:

$$\nu_t = \frac{a_1 k}{max(a_1\omega, SF_2)}, \; S = 2\sqrt{2S_{ij}S_{ij}}. \tag{2.28}$$

The blending function F_2 is one for boundary layer flows and zero for free shear layers:

$$F_2 = tanh\left[\arg_2^2\right], \; \arg_2 = max\left(\frac{2\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right). \tag{2.29}$$

The constants of $k-\omega$ SST are:

$$\beta^* = 0.09 \; a_1 = 0.31. \tag{2.30}$$

2.3.4 Reynolds stress SSG model

The Reynolds stress SSG model solves six partial differential equations: one for each of six independent components of Reynolds stresses $u_i'u_j'$ along with another transport equation of the dissipation rate ε:

$$\frac{\partial\overline{u_i'u_j'}}{\partial t} + U_k\frac{\partial\overline{u_i'u_j'}}{\partial x_k} = -\frac{\partial\overline{u_i'u_j'u_k'}}{\partial x_k} + \mathcal{P}_{ij} + \Pi_{ij} - \varepsilon_{ij} + D_{ij}, \tag{2.31}$$

$$\frac{\partial\varepsilon}{\partial t} + U_i\frac{\partial\varepsilon}{\partial x_i} = C_\varepsilon\frac{\partial}{\partial x_i}\left(\frac{k}{\varepsilon}\overline{u_i'u_j'}\frac{\partial\varepsilon}{\partial x_j}\right) + C_{\varepsilon 1}\frac{\mathcal{P}\varepsilon}{k} - C_{\varepsilon 2}\frac{\varepsilon^2}{k}, \tag{2.32}$$

where $\mathcal{P}_{ij}$ is the production term:

$$\mathcal{P}_{ij} = -\left(\overline{u_i'u_k'}\frac{\partial U_j}{\partial x_k} + \overline{u_j'u_k'}\frac{\partial U_i}{\partial x_k}\right), \; \mathcal{P} = \overline{u_i'u_j'}\frac{\partial U_i}{\partial x_j}, \tag{2.33}$$

Π_{ij} is the velocity-pressure-gradient term:

$$\Pi_{ij} = -\frac{1}{\rho}\left(\overline{u_i'\frac{\partial p'}{\partial x_j} + u_j'\frac{\partial p'}{\partial x_i}}\right), \tag{2.34}$$

ε_{ij} is the dissipation term:

$$\varepsilon_{ij} = 2\nu\overline{\frac{\partial u_i}{\partial x_k i}\frac{\partial u_j'}{\partial x_k}}, \tag{2.35}$$

D_{ij} is the diffusion term:

$$D_{ij} = \nu\frac{\partial^2\overline{u_i'u_j'}}{\partial x_k\partial x_k}, \tag{2.36}$$

In the Reynolds-stress model, U_i, P, $u_i'u_j'$ and ε are known, thus the two terms on the left hand side of Eq. 2.31, and the production term $\mathcal{P}_{ij}$ are in closed form. The triple velocity

correlation term $\overline{u_i' u_j' u_k'}$, the dissipation term ε_{ij} and the velocity-pressure-gradient term Π_{ij} should be modeled.

Daly and Harlow (1970) proposed the following model for the third oder term:

$$\overline{u_i' u_j' u_k'} = -C_s \frac{k}{\varepsilon} \overline{u_k' u_l'} \frac{\partial \overline{u_i' u_j'}}{\partial x_l}. \tag{2.37}$$

Speziale et al. (1991) proposed the following model for the velocity-pressure-gradient term:

$$\begin{aligned} \Pi_{ij} = & -(C_1 \varepsilon + C_1^* \mathcal{P}) b_{ij} \\ & + C_2 \varepsilon (b_{ik} b_{kj} - \frac{1}{3} b_{mn} b_{mn} \delta_{ij}) \\ & + (C_3 - C_3^* \sqrt{b_{ij} b_{ij}}) k S_{ij} \\ & + C_4 (b_{ik} S_{jk} + b_{jk} S_{ik} - \frac{2}{3} b_{mn} S_{mn} \delta_{ij} \\ & + C_5 k (b_{ik} \Omega_{jk} + b_{jk} \Omega_{ik}), \end{aligned} \tag{2.38}$$

where b_{ij} is the anisotropy tensor:

$$b_{ij} = \frac{\overline{u_i' u_j'}}{2k} - \frac{1}{3} \delta_{ij}, \tag{2.39}$$

S_{ij}, Ω_{ij} are the mean rate of strain tensor and mean rotation tensor, respectively:

$$S_{ij} = \frac{1}{2} \left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i} \right), \ \Omega_{ij} = \frac{1}{2} \left(\frac{\partial U_i}{\partial x_j} - \frac{\partial U_j}{\partial x_i} \right). \tag{2.40}$$

In the high-Reynolds-number flows, the dissipation is isotropic:

$$\varepsilon_{ij} = \frac{2}{3} \varepsilon \delta_{ij}, \tag{2.41}$$

Rotta (1951) proposed the following model for the near wall region:

$$\varepsilon_{ij} = \frac{\overline{u_i' u_j'}}{k} \varepsilon. \tag{2.42}$$

The constants in the model are summarized here:

C_s	C_1	C_1^*	C_2	C_3	C_3^*	C_4	C_5	$C_{\varepsilon 1}$	$C_{\varepsilon 1}$
0.22	3.4	1.8	4.2	0.8	1.3	1.25	0.4	1.44	1.83

(2.43)

2.3.5 Models for turbulent scalar fluxes

Code_saturne provides several models to estimate the turbulent scalar fluxes such as simple gradient diffusion hypothesis (SGDH) and generalized gradient diffusion hypothesis (GGDH).

SGDH. The SGDH is the most commonly used model to estimate turbulent scalar flux for two-equation turbulence models. Being analogous to Fick's law of molecular diffusion, SGDH assumes that the turbulent transport of a scalar ϕ is proportional to the gradient of its mean concentration:

$$\overline{\phi' u_i'} = -\frac{\nu_t}{\sigma_\phi} \frac{\partial \phi}{\partial x_i}, \tag{2.44}$$

where σ_ϕ is the turbulent Schmidt number for turbulent mass transfer or turbulent Prandtl number for turbulent heat transfer.

GGDH. The GGDH takes the anisotropy of the Reynolds stresses on the turbulent scalar flux as:

$$\overline{\phi' u_i'} = -C_\phi \frac{k}{\varepsilon} \overline{u_i' u_j'} \frac{\partial \phi}{\partial x_j}, \tag{2.45}$$

where constant $C_\phi = 0.235$. GGDH is the most used model for Reynolds stress models.

2.3.6 Turbulence model selection

The choice of turbulence model used in a simulation can have a large impact on the obtained solution. First-order models such as $k-\varepsilon$ and $k-\omega$ assume isotropic turbulence structures, whereas RSM models can handle anisotropic structures (and different variants of the model can handle isotropic or anisotropic diffusivity).

For some flows with complex vortex interactions, and many flows with junctions for injection or extraction of fluid, reproducing the correct physical structure may require using at least an RSM model. While RSM models are usually more precise than first-order RANS models, they involve a significantly higher computational cost, and tend to be less stable, especially for lower quality meshes. So for flows where they are not necessary, using a first order RANS model may be preferred.

To determine which type of model is preferred, or needed, a user may refer to a validation test suite and a PIRT, representing expert knowledge. In a series of simulations, running at least one simulation with a different model may help ensure no important flow phenomena is missed. Using assistance from machine learning could be very useful here, and is one of the motivating factors of this work.

2.4 Numerical methods in code_saturne

Code_sature is an open-source CFD software developed by EDF for the simulation of the laminar or turbulent, incompressible or weakly dilatable flows on structured or unstructured meshes Archambeau et al. (2004). The code is provided with different turbulence models, from RANS to LES models can be run in parallel with massive number of mesh cells. Plentiful specific physical modules are developed, such as gas, and coal combustion, semi-transparent radiative transfer, particle-tracking with Lagrangian modeling, Joule effect, electrics arcs, weakly compressible flows, atmospheric flows, rotor/stator interaction for hydraulic machines.

Code_saturne is based on finite volume method (FVM) which consists of three steps:

- Integration of the governing equations over the control volumes of the domain and over time,
- Discretisation of the integral equations into an algebraic equations system,
- Solving the algebraic equations system by iterative method.

2.4.1 Time discretisation

Code_saturne utilizes a segregated solver based on semi-implicit method for pressure-linked equations-consistent (SIMPLEC) to solve the time incremental term of the RANS equations which mainly contains a velocity prediction step and pressure correction step.

2.4.1.1 Velocity prediction step

The predicted velocity field $\tilde{u}^{n+1}$ is obtained by solving the following equation:

$$\rho\frac{\tilde{u}_i^{n+1}-u_i^n}{\Delta t}+\rho u_j^n\frac{\partial\tilde{u}_i^{n+\theta}}{\partial x_j}=-\frac{\partial p^{n-1+\theta}}{\partial x_i}+\mu\frac{\partial^2\tilde{u}^{n+\theta}}{\partial x_j\partial x_j}-\rho\frac{\partial^2 R_{ij}^{n+\theta_S}}{\partial x_i\partial x_j}+S_M^{n+\theta_S}, \tag{2.46}$$

using θ-scheme:

$$\tilde{u}^{n+\theta}=\theta\tilde{u}^{n+1}+(1-\theta)u^n,\ \begin{cases}\theta=0, & \text{explicit scheme,}\\ \theta=1/2, & \text{Crank-Nicolson scheme,}\\ \theta=1, & \text{fully implicit scheme.}\end{cases} \tag{2.47}$$

where R_{ij} is Reynolds stress $R_{ij}=\overline{u_i'u_j'}$, and the exponent $n+\theta_S$ indicates the time extrapolation point for Reynolds stresses and momentum source terms.

2.4.1.2 Pressure correction step

The predicted velocity field is not divergence free. The second step corrects the pressure by imposing the nullity of the stationary constraint for the velocity computed at time $n+1$. The following equations are solve:

$$\begin{cases}\rho(\underline{u}^{n+1}-\underline{\tilde{u}}^{n+1})/\Delta t=-\nabla\delta P^{n+\theta},\\ \nabla\cdot\underline{u}^{n+1}=0,\end{cases} \tag{2.48}$$

where $\underline{u}$, ∇, $\nabla\cdot$ represent velocity vector, gradient and divergence operators, respectively, and the pressure increment $\delta P^{n+\theta}=P^{n+\theta}-P^{n-1+\theta}$. Taking the divergence of the above equation, the following Poisson equation for the pressure is obtained:

$$\nabla\cdot(\Delta t\nabla\delta P^{n+\theta})=\nabla\cdot(\rho\underline{\tilde{u}}^{n+1}). \tag{2.49}$$

Then the velocity is corrected by:

$$\underline{u}^{n+1}=\underline{u}^n-\frac{\Delta t}{\rho}\nabla\delta P^{n+\theta}. \tag{2.50}$$

2.4.2 Space discretisation

Code_saturne uses the co-located finite volume method which solves and stores the velocity, pressure and other scalar variables at the same points - the cell centers. A schematic of two adjacent cells and their entities in the FVM mesh is shown in Fig. 2.3. On a mesh with N_{cell} cells each discretized field ϕ has N_{cell} degrees of freedom, denoted by ϕ_i:

$$\phi_i\equiv\frac{1}{|\Omega_i|}\int_{\Omega_i}\phi d\Omega,\ \ i\in[1,\cdots,N_{cell}], \tag{2.51}$$

where Ω_i is the volume of the i-th cell. Since the discretized field ϕ is linear in every cell, ϕ_i can be identified by the value of the field at I, the cell center of Ω_i:

$$\phi_I=\phi_i. \tag{2.52}$$

The face value of the discretized field ϕ on the face defined as:

$$\phi_f\equiv\frac{1}{|S_f|}\int_f\phi dS, \tag{2.53}$$

is linear on the face f, thus Y_f is associated to the face center F:

$$\phi_F=\phi_f. \tag{2.54}$$

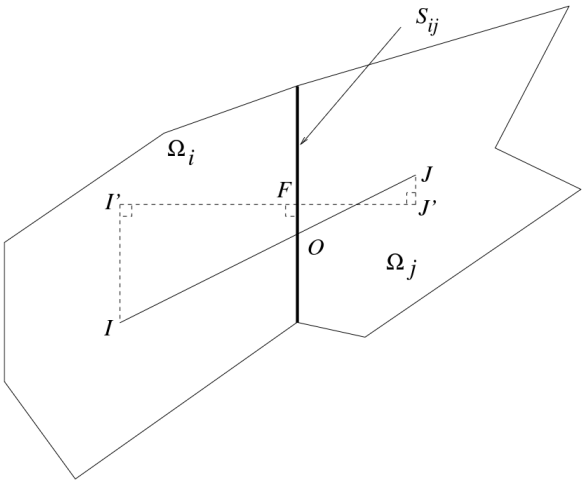


Figure 2.3: Schematic of two internal adjacent cells in the mesh.

2.4.2.1 Velocity prediction step

The integrated form of the Eq. 2.46 reads:

$$\rho\frac{|\Omega_i|}{\Delta t}\left(\tilde{\underline{u}}_i^{n+1}-\underline{u}_i^n\right)+\sum_{j\in V(i)}\tilde{\underline{u}}_{f_{ij}}^{n+\theta}(\rho\underline{u}\cdot\underline{n})_{f_{ij}}^n|S_{ij}|=-|\Omega_i|\nabla(p_i)^{n-1+\theta}+\sum_{j\in V(i)}(\mu\nabla\tilde{\underline{u}}\cdot\underline{n})_{f_{ij}}^{n+\theta}|S_{ij}|+|\Omega_i|A_i^n. \tag{2.55}$$

The divergence theorem is used for the integration of the convective and diffusive terms:

$$\int_{\Omega_i}\nabla\cdot\underline{u}=\sum_{f\in\mathcal{F}_i}\underline{u}_{f_i}\underline{S}_{f_i}. \tag{2.56}$$

Convective term

For the convective term, the mass flux $\rho\underline{u}_{f_{ij}}^n$ is available from the last iteration, the value at the interior face $\tilde{\underline{u}}_{f_{ij}}^{n+\theta}$ should be evaluated. Code_saturne provides three spatial discretisation schemes to evaluate the variable $\phi_{f_{ij}}$ at the interior face:

- Upwind scheme
  The 1^{st}-order scheme reads:

$$\phi_{f_{ij}}=\begin{cases}\phi_i, & \text{if } (\rho\underline{u}\cdot\underline{n})_{f_{ij}}^n\geq 0,\\ \phi_j, & \text{if } (\rho\underline{u}\cdot\underline{n})_{f_{ij}}^n<0.\end{cases} \tag{2.57}$$

- Centered scheme
  The 2^{nd}-order centered scheme reads:

$$\phi_{f_{ij}}=\alpha_{ij}\phi_i+(1-\alpha_{ij})\phi_j+\frac{1}{2}[\nabla\phi_i+\nabla\phi_j]\cdot\underline{OF}, \tag{2.58}$$

  where α_{ij} is the weighting factor to measure the cell center I to the face f relative to the neighbor cell J:

$$\alpha_{ij}=\frac{\overline{FJ'}}{\overline{I'J'}}, \tag{2.59}$$

  where $\overline{FJ'}$ and $\overline{I'J'}$ are defined as:

$$\overline{FJ'}\equiv\frac{\underline{FJ'}\cdot\underline{S}_{ij}}{|\underline{S}_{ij}|},\ \overline{I'J'}\equiv\frac{\underline{I'J'}\cdot\underline{S}_{ij}}{|\underline{S}_{ij}|}. \tag{2.60}$$

- Second order linear upwind (SOLU) scheme
  The 2^{nd}-order linear upwind scheme reads:

$$\phi_{f_{ij}} = \begin{cases} \phi_i + \nabla\phi_i \cdot \underline{IF} & \text{if } (\rho\underline{u}\cdot\underline{n})^n_{f_{ij}} \geq 0, \\ \phi_j + \nabla\phi_j \cdot \underline{JF} & \text{if } (\rho\underline{u}\cdot\underline{n})^n_{f_{ij}} < 0. \end{cases} \tag{2.61}$$

Diffusive term

The diffusivity μ in the diffusive term should be interpolated from the neighboring cells to the interior face. Two interpolation methods are available: harmonic interpolation and arithmetic interpolation:

$$\mu_{f_{ij}} = \begin{cases} \mu_i\mu_j/[\alpha_{ij}\mu_i + (1-\alpha_{ij})\mu_j], & \text{harmonic mean}, \\ (\mu_i+\mu_j)/2, & \text{arithmetic mean}. \end{cases} \tag{2.62}$$

The gradient of a variable on the interior face $\phi_{f_{ij}}$ is discretiszed by:

$$\nabla\phi_{f_{ij}} = \begin{cases} (\phi_j-\phi_i)/\overline{I'J'}, & \text{for non-reconstructed field}, \\ \left[(\phi_j-\phi_i) + (\underline{JJ'}-\underline{II'})\cdot(\nabla\phi_i+\nabla\phi_j)/2\right]/\overline{I'J'}, & \text{for reconstructed field}. \end{cases} \tag{2.63}$$

2.4.2.2 Pressure correction step

The Poission equation 2.49 is discretized as:

$$\Delta t \sum_{j\in V(i)} (\nabla\delta p\cdot\underline{n})^{n+1}_{f_{ij}}|S_{f_{ij}}| = \sum_{j\in V(i)} (\rho\underline{u}\cdot\underline{n})^{n+1}_{f_{ij}}|S_{f_{ij}}|. \tag{2.64}$$

Additional filter term of Rhie and Chow Rhie and Chow (1983) is added to the mass flux to avoid the pressure spatial oscillation caused by the pressure and velocity decoupling:

$$(\rho\underline{u}\cdot\underline{n})_{f_{ij}} = (\rho\underline{u}\cdot\underline{n})_{f_{ij}} + \mathcal{F}_{f_{ij}}. \tag{2.65}$$

2.4.3 Wall modeling

As shown in Fig. 2.4, the turbulent boundary layer can be divided into three layers in terms of the relation of the dimensionless velocity scale u^+ to the dimensionless wall distance y^+: viscous sub-layer, buffer layer and log-law layer.

u^+ is defined as:

$$u^+ \equiv \frac{u}{u_\tau}, \tag{2.66}$$

where u is the velocity component projected to the plane tangent to the wall, u_τ is the friction velocity derived from the wall shear stress τ_w:

$$u_\tau = \sqrt{\frac{\tau_w}{\rho}}. \tag{2.67}$$

y^+ is the non-dimensionalized distance of the first layer cell center to the wall:

$$y^+ = \frac{y_p u_\tau}{\nu}. \tag{2.68}$$

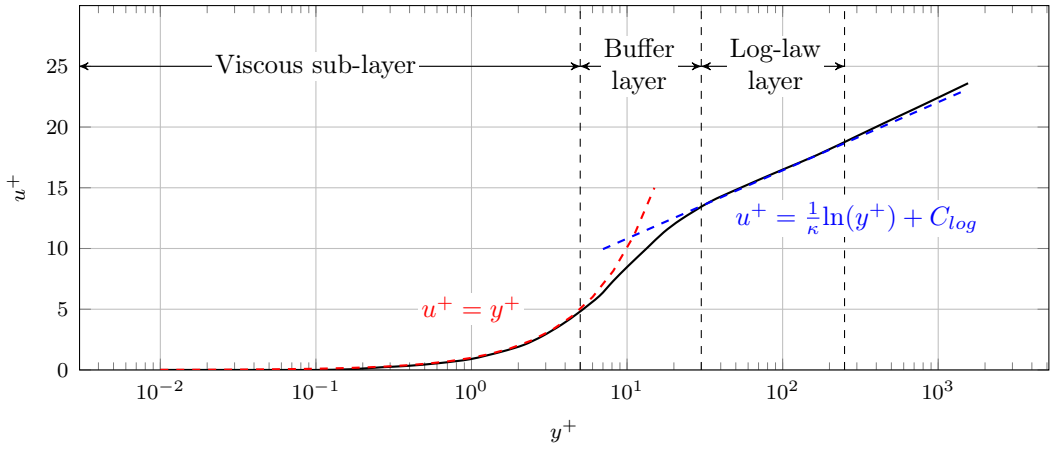


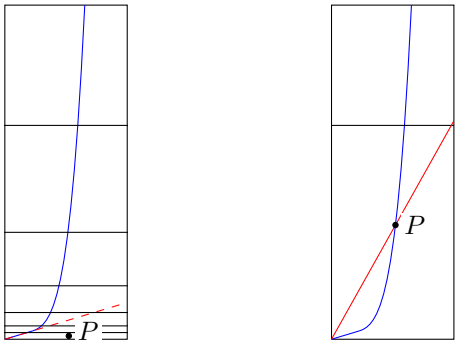
Figure 2.4: Velocity profile in the turbulent boundary layer Moser et al. (1999).

Viscous sub-layer ($y^+ < 5$): In the layer directly adjacent to the wall the flow is dominated by the viscosity, the dimensionless velocity profile is linear to the dimensionless wall distance:

$$u^+ = y^+. \tag{2.69}$$

Log-law layer ($y^+ > 30$): Close to the turbulent core the inertial force is dominant and the velocity profile is logarithmic with respect to wall distance:

$$u^+ = \frac{1}{\kappa}\ln(y^+) + C_{log}. \tag{2.70}$$



(a) LR mesh. *(b) HR mesh.*

Figure 2.5: Two types of meshes in the near wall region: (a) Low-Reynolds (LR) mesh and (b) High-Reynolds (HR) mesh.

where von Karman's constant $\kappa \approx 0.41$ and $C_{log} = 5.2$.

Buffer layer ($5 < y^+ < 30$): The transition region between the above two layers is called buffer layer.

If the steep gradient in the near wall vicinity should be captured, the first layer cell center must be small enough to attain $y^+ < 5$ thus leading to the so-called low-Reynolds mesh, as shown in Fig. 2.5a, which significantly increase the quantity of the mesh cells and computational cost. In the industrial applications, the high-Reynolds meshes, as show in Fig. 2.5b, are usually used together with a wall function to estimate the variables near the wall so that the first layer cell center can be located in the log-law region and as a result the mesh quantity is significantly reduced. Multiple wall functions are supplied in code_saturne, such as one friction velocity scale based on wall shear stress, two friction velocity scales based on both wall shear stress and turbulent kinetic energy, and adaptive wall functions suitable for all range of y^+. More details of these wall functions are referred to saturne support@edf.fr (2021); Wald (2016).

2.4.4 Algebraic multigrid method (AMG)

AMG Driss et al. (2018) is widely used in CFD codes to accelerate the iteration convergence of the large sparse system of linear equations resulted from the discretisation of

the second-order elliptic partial differential equations. The general idea is that the high-frequency of the solution error on the fine mesh will become low-frequency at a coarser mesh and thus can be easily removed. A hierarchy of successively coarsened graphs should be constructed to restrict the solution error to the coarsest level where the equations can be solved directly.

2.4.4.1 General ideas

The following linear algebraic equation system

$$A_h u^h = b^h \text{ or } \sum_{j \in \Omega^h} a_{ij}^h u_j^h = b_i^h \ (i \in \Omega^h) \tag{2.71}$$

is resulted from the discretisation of NS equations in the fluid domain, where matrix A of size $n \times n$ contains the coefficients a_{ij} $(i, j = 1, ..., n)$, determined by discretisation. The vector u of length n is composed of the unknown node values of the function to be solved. The vector b consists of the coefficients resulted by the discretisation and by the given values on the boundary. The unknown values at the mesh nodes $\Omega = \{1, ..., n\}$ are denoted by $u_i (i = 1, ..., n)$.

The interpolation operator I_H^h is used to transfer the solution from coarse mesh to the fine mesh, and the operator of restriction to the coarse mesh level I_h^H obtained from the condition obtained using the Galerkin method

$$A_H = I_h^H A_h I_H^h, \text{ where } I_h^H = (I_H^h)^T. \tag{2.72}$$

The fine mesh solution is obtained from the equation

$$u_{new}^h = u_{old}^h + I_H^h e^H, \tag{2.73}$$

where e^H is the correction of the solution on the coarse mesh which is obtained by solving

$$A_H e^H = r^H \text{ or } \sum_{j \in \Omega^H} a_{ij}^H e_j^H = r_i^H \ (i \in \Omega^H), \tag{2.74}$$

where $r^H = I_h^H(r_{old}^h)$ and $r_{old}^h = b^h - A_h u_{old}^h$. The error of the numerical solution $e^h = u_*^h - u^h$ (the asterisk marks the exact solution) is found using the coarse mesh correction operator

$$e_{new}^h = K_{h,H} e_{old}^h, \text{ where } K_{h,H} = I_h - I_H^h A_H^{-1} I_h^H A_h, \tag{2.75}$$

where I_h is the identity operator.

The solution is smoothed using the operator S_h. In the smoothing phase

$$u^h \to \overline{u}^h, \text{ where } \overline{h}^h = S_h u^h + (I_h - S_h) A_h^{-1} b^h, \tag{2.76}$$

where $\overline{u}$ indicates smoothed solution. Smoothing the solution error yields $e^h \to \overline{e}^h$, where $\overline{e}^h = S_h e^h$.

The set of coarse mesh variables is split into two disjoint subsets $\Omega^h = C^h \cup F^h$, where C^h is the set of coarse mesh variables and F^h is the set of fine mesh variables. The solution error $e^h = I_H^h e^H$ is interpolated by the rule

$$e_i^h = (I_H^h e^H)_i = \begin{cases} e_i^H, & \text{if } i \in C^h, \\ \sum\limits_{k \in P_i^h} w_{ik}^h e - k^H, & \text{if } i \in F^h, \end{cases} \tag{2.77}$$

where $P_i^h \in C^h$ is the set of variables involved in the interpolation. When the solution is interpolated from the coarse mesh to the fine mesh, the solution error e_i^h is set to e_i^H if i is a C-variables and it is set equal to a weighted sum of variables from the set P_i^h if i belongs to the subset F. The set P_i^h is a small subset of the set of C-variables that are close to the variable i, which guarantees that the matrix A_H is sparse. On the other hand, the set P_i^h contains a sufficiently large number of variables to which i is strongly coupled.

2.4.4.2 Implementation steps

The set of coarse mesh variables is split into the subsets of C-variables and F-variables. The mesh level $\Omega^k (k = 1, 2, ..., M-1)$ is assigned the subsets C^k and F^k. Each mesh level is also assigned the mesh operators $A^1, A^2, ..., A^M$ (where $A^1 = A$), and also the interpolation operators $P^k = I_{k+1}^k$ and the restriction operators $R^k = I_k^{k+1}$, where $k = 1, 2, ..., M-1$. The restriction is the transposed interpolation operator $R^k = (P^k)^T$. To construct the matrix of the system on the coarse mesh level, the Galerkin product is calculated, $R^k A^k P^k$. The error on each mesh level is smoothed using the smoothing operator $S^k (k = 1, 2, ..., M-1)$. The coarsening procedure is repeated until the size of the system becomes sufficiently low for the system of difference equations to be solved by a direct method.

The subsets C^k and F^k, as well as the interpolation, restriction, and smoothing operators, are constructed in the setup phase.

Upon the execution of the setup phase, the solution to the original equation is found by recursively applying the multigrid procedure. The number of smoothing iterations and the number of recursive calls of the method on each mesh level are specified. The sensitivity of the solution phase to the choice of the smoothing procedure is typically relatively low, and classical iterative methods are usually used.

2.4.4.3 Construction of mesh levels

The convergence of iteratively solving linear algebraic equation system can be accelerated by considering strong couplings between variables. The nodes in meshes are interpreted as the nodes of a directed graph related to the given matrix. The mesh node $i \in \Omega^h$ (associated with the variable u_i^h) is connected to the variable $j \in \Omega^h$ if $a_{ij}^h \neq 0$. The set of variables adjacent to the variable i has the form

$$N_i^h = \left\{ j \in \Omega^h : j \neq i, a_{ij}^h \neq 0 \right\} \ (i \in \Omega^h). \tag{2.78}$$

The variable i strongly depends on the variable j, and the variable j strongly affects the variable i if the magnitude of the matrix element a_{ij} is greater than all the off-diagonal coefficients in the matrix

$$-a_{ij} \geq \theta \max_{k \neq i} \{-a_{ik}\}. \tag{2.79}$$

The parameter $0 < \theta \leq 1$ controls the number of strong couplings between variables (typically, $\theta = 0.25$). The set S_i is the set of all variables j that are strongly coupled with the variable i (the set of variables that strongly affect the variable i)

$$S_i = \left\{ j : j \neq i, -a_{ij} \geq \theta \max_{k \neq i} (-a_{ik}) \right\}. \tag{2.80}$$

The available theoretical approaches are applied to solving scalar second-order elliptic partial differential equations whose discretisation yields a system of difference equations with an M-matrix. If there are both negative and positive off-diagonal elements, the following definitions are used:

$$a_{ij}^{-} = \begin{cases} a_{ij}, & \text{if } a_{ij} < 0, \\ 0, & \text{if } a_{ij} \geq 0, \end{cases} \quad a_{ij}^{+} = \begin{cases} 0, & \text{if } a_{ij} \leq 0, \\ a_{ij}, & \text{if } a_{ij} > 0. \end{cases} \tag{2.81}$$

In this case, two disjoint subsets of variables are

$$N_i^{-} = \{j \in N_i : a_{ij}^h < 0\}, N_i^{+} = \{j \in N_i : a_{ij}^h > 0\}. \tag{2.82}$$

In practice, C/F is constructed in such a way that the set of C-variables is approximately the maximally independent set (within the set C, the variables have no strong couplings between themselves), and the F-variables are surrounded by interpolatory C-variables.

2.4.4.4 AMG in code_saturne

Algorithm 1: Algebraic multigrid method (V-cycle). The iteration numbers are normally set to $\nu_1 = \nu_2 = \beta = 1$.

Pre-smoothing: do ν_1 times $x^l \leftarrow \mathcal{S}^l(x^l, b^l)$;
Restriction:
$l = 1$;
while $l + 1 < L$ **do**
  $b^{l+1} \leftarrow P_l^T(b^l - A_l x^l)$;
  iterate β times of $x^{l+1} \leftarrow \mathcal{S}^{l+1}(x^{l+1}, b^{l+1})$; $l \leftarrow l + 1$;
end
Solve: $A_L x^L = b^L$;
Prolongation:
while $l > 0$ **do**
  Correction: $x^l \leftarrow x^l + P_l x^{l+1}$;
end
Post-smoothing: do ν_2 times $x^1 \leftarrow \mathcal{S}^1(x^1, b^l)$.

Code_saturne uses an AMG, similar though not identical to Volkov (2018); Vanek et al. (1994); Notay (2010), to construct such graph hierarchy based on the coupling strength of adjacent cells. Two adjacent cells i and j at level l are considered strongly coupled if $\frac{a_{ij}^l}{\sqrt{a_{ii}^l a_{jj}^l}}$ is relatively large and then aggregated. After the automatic coarsening process, a series of full rank prolongation matrices P_l of size $n_l \times n_{l+1}$, $l = 1, ..., L-1$, is created that:

$$A_{l+1} = P_l^T A_l P_l. \tag{2.83}$$

To solve the system $A_l x^l = b^l$, an approximate solution x^l is obtained by a smoothing operator $\mathcal{S}^l(x^l, b^l)$ which is one of the iterative methods. Then the error $P_l^T(b^l - A_l x^l)$ is restricted as the right hand side of the equation to the next coarse graph level A_{l+1} where this smoothing and restriction procedure is repeated until the largest level L. At the level L the matrix A_L is small enough to be solved using direct methods. The solution obtained at level L is then prolonged to the finer level $L-1$ to correct the solution x^{L-1}:

$x^{L-1} \leftarrow x^{L-1} + P_{L-1}x^L$. This prolongation and correction is carried out reversely till level $l = 1$. The process of using V-cycle AMG to solve the linear equation system is shown in algorithm 1. The source code used in code_saturne to generate the mesh hierarchy is included in B.

2.5 Summary

In this chapter, the characteristics of turbulent flow and its governing equations are introduced firstly. While the high-fidelity simulation strategies, such DNS and LES, can reveal the stochastic details of turbulent flows, their prohibitive computational cost preclude extensive application in addressing the three-dimensional flow phenomena encountered in the industrial cases. On the other hand, RANS simulations in conjunction with turbulence models which focus on the time-averaged mean flow fields can satisfy the engineering design requirements at the lowest computational expense, thus have been applied widely. The most used two equation turbulence models and one of the second-order turbulence models, such as $k - \varepsilon$, $k - \omega$, $SST\ k - \omega$ and $RSM\ SSG$, are described.

Code_saturne is an open-source CFD software based on FVM method. The FVM numerical methods employed in the code including the time and spatial discretisation methods, semi-implicit method for pressure linked equations and wall modeling, AMG method to accelerate the convergence of linear equations system, are also reviewed.

3 Fundamental knowledge of neural networks

Contents

In this chapter, the general training procedure in supervised way is introduced along with its potential problems, followed by the basic concepts of neural networks. The main components and popular architectures of convolutional neural networks, the most successful machine learning algorithms in computer vision, are introduced too. Its counterpart in graph neural networks is searched to overcome CNNs' inability of process unstructured meshes by comparing several graph convolution kernels. A brief summary is made at the end.

3.1 Supervised training

Training procedure. Most of the applications of data-driven algorithms in CFD domain train the machine learning model in supervised way. Given a data-set $\mathcal{D}(\boldsymbol{x}, \boldsymbol{y})$, a ML algorithm is supposed to find the mapping function $f := \boldsymbol{x} \mapsto \boldsymbol{y}$ from the input $\boldsymbol{x}$ to the provided ground-truth label/output $\boldsymbol{y}$ by searching the global minimal of the loss function $\mathcal{L}$. It mainly contains two steps: feedforward calculation from input to prediction and backpropagation of the error to update trainable parameters. In the feedforward step, the signals are propagated in the direction from the input layer to the output layer where the neurons at each layer process the signals according to their trainable weights. At the output layer, the deviation between the prediction $\hat{\boldsymbol{y}}$ and the ground-truth label $\boldsymbol{y}$ is calculated according to the problem-dependent loss function $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$. In the latter step, the gradients of the loss with respect to each trainable variable are calculated using a gradient descent algorithm and the trainable parameters are updated gradually with the

given small learning ratio. It is called one training epoch if all the data is fed to the model and the trainable parameters are updated once accordingly.

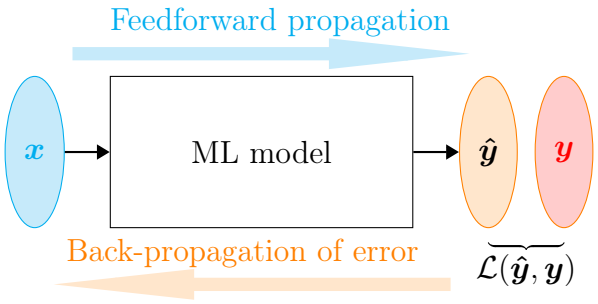


Figure 3.1: Supervised training of neural networks.

Underfitting and overfitting. A common potential problem for supervised training is the underfitting and overfitting. The dataset is normally divided into training set and testing test over which the losses are monitored during the training process, as shown in Fig. 3.2. At the beginning of the training, both training loss and testing loss decrease. At this stage, the model is unable to well fit the data which is called underfitting. Overfitting happens when the training loss continues to decrease while the testing loss increases as the training continues, indicated by the region colored by yellow color in Fig. 3.2. It signifies that the model is trapped in local minima where it overfits the training cases and fails to fit the testing cases. Even though an ideal model should be robust and easy to train, overfitting is quite common due to the non-convex nature of most of the optimization problems where many local minima exist. There are several practices which can help the model avoid this problem, such as early stopping, batch normalization, regularization and etc. Though the early stopping is widely used to stop the training at the region shaded by yellow color in Fig. 3.2, it is merely a remedy solution since it prevents further decreasing the prediction accuracy. A more efficient strategy to evade this problem is the stochastic training where the trainable parameters are updated over the loss estimated on a batch of data which contains a certain number of data points. Between training epochs, the data points are randomly shuffled to form different batches. Compared with feeding on the entire data-set, the convergence of the training is accelerated feeding on batches since the trainables are updated many times within one epoch.

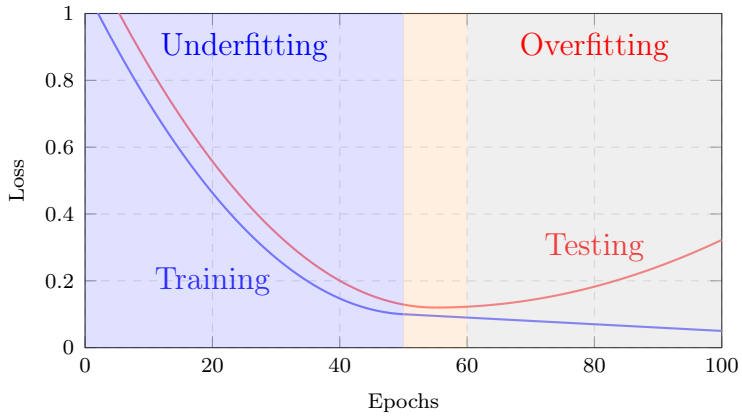


Figure 3.2: Underfitting and overfitting of the supervised training.

3.2 Basic concepts of neural networks

An artificial neural networks (ANNs), which are analogous to the biological neural networks, contain one input layer, one output layer and a certain number of hidden layers. Each layer contains a series of neurons which receive, process and transfer the signal to those in the succeeding layer. A multilayer perceptron (MLP) network with one input layer, one hidden layer and one output layer is shown in Fig. 3.3.

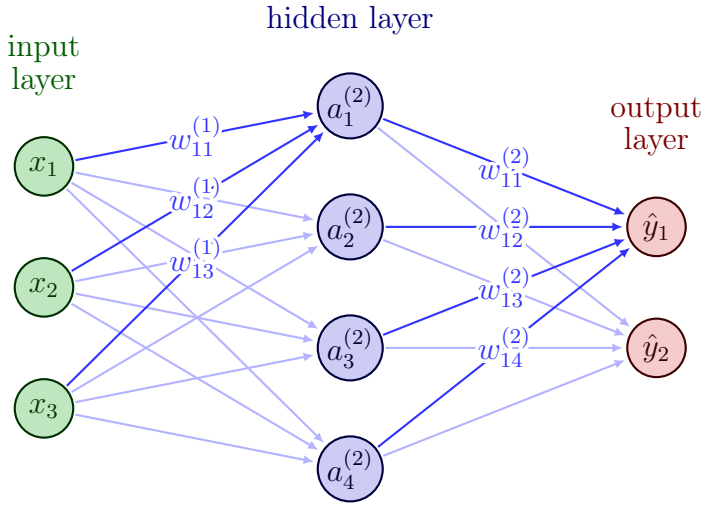


Figure 3.3: A multilayer perceptron with one hidden layer.

The MLP is a **fully connected** (FC) neural network which means that each neuron in the preceding layer is connected to all the neurons in the succeeding layer with associated trainable weight to each connection. The j-th neuron in the $(l+1)$-th hidden layer computes the linear combination of the inputs and the weights firstly:

$$z_j^{l+1} = \sum_{i=1}^{N} w_{ji}^l x_i + b_j^l, \tag{3.1}$$

where N is the total number of neurons or inputs in the l-th layer, w_{ji}^l the trainable weight connecting the i-th neuron in l-th layer to the j-th neuron in the $(l+1)$-th layer, b_j^l the trainable bias of the j-th neuron in the l-th layer, and then applies an activation function:

$$a_j^{l+1} = \sigma(z_j^{l+1}), \tag{3.2}$$

where σ represents a non-linear **activation function**. Several commonly used activation functions are listed in Table 3.1 and plotted in Fig. 3.4. The term a_j^{l+1} denotes the activation at the j-th neuron in the $(l+1)$-th layer. Similarly, the output layer linearly combines the hidden features in the previous layer with the corresponding weights:

$$\hat{y_j} = \sum_{i=1}^{N} w_{ji}^L a_i^L. \tag{3.3}$$

where L represents the last hidden layer.

Thus the activation in the hidden layer shown in Fig. 3.3 is calculated by:

$$\begin{bmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \\ a_4^2 \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & w_{43}^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \\ b_4^1 \end{bmatrix} \right), \tag{3.4}$$

Logistic sigmoid	Hyperbolic tangent	Rectified linear unit (ReLU)	Softplus	Softmax
$\frac{1}{1+\exp(-x)}$	$tanh(x)$	$max(0, x)$	$\frac{1}{\beta}\log(1+\exp(\beta x))$	$\frac{\exp(x_i)}{\sum_j \exp(x_j)}$

Table 3.1: Five most used activation functions $\sigma(x)$.

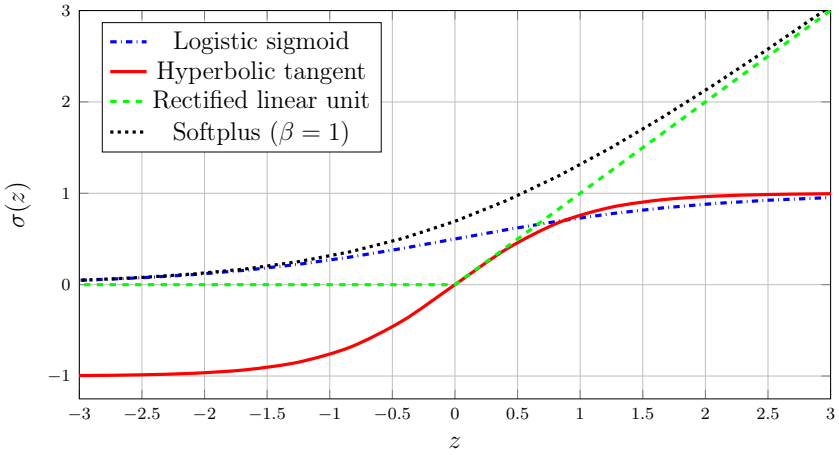


Figure 3.4: Plot of four activation functions.

and the output layer gives prediction by:

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 & w_{14}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 & w_{24}^2 \end{bmatrix} \begin{bmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \\ a_4^2 \end{bmatrix}. \tag{3.5}$$

With the supervised training method, a proper **loss function** or **cost function** should be defined depending on the nature of the training task to measure the difference between the prediction $\hat{\boldsymbol{y}}$ and the ground-truth label $\boldsymbol{y}$. For a regression task, the mean squared error (MSE) loss function are usually selected:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2, \tag{3.6}$$

where N represents the total number of predictions. All the trainables in the neural network are updated iteratively to minimize the loss function using **back-propagation** algorithm. In back-propagation, the gradient of the loss with respect to each trainable is calculated by chain rule:

$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}. \tag{3.7}$$

For the weights in the last layer in Fig. 3.3 the gradients are calculated by:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_{ji}^2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}_i}\frac{\partial \hat{y}_j}{\partial w_{ji}^2} \\ &= \frac{2}{N}(\hat{y}_j - y_j)a_i^2. \end{aligned} \tag{3.8}$$

The gradients with respect to the weights and biases in the first layer are calculated by:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{ik}^1} &= \frac{\partial \mathcal{L}}{\partial a_i^2}\frac{\partial a_i^2}{\partial z_i^1}\frac{\partial z_i^1}{\partial w_{ik}^1} \\ &= \left(\sum_{j=1}^{2}\frac{\partial \mathcal{L}}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial a_i^2}\right)\frac{\partial a_i^2}{\partial z_i^1}\frac{\partial z_i^1}{\partial w_{ik}^1} \\ &= \left(\sum_{j=1}^{2}\frac{2}{N}(\hat{y}_j - y_j)w_{ji}^2\right)\sigma'(z_i^1)x_k.\end{aligned} \tag{3.9}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b_k^1} &= \frac{\partial \mathcal{L}}{\partial a_i^2}\frac{\partial a_i^2}{\partial z_i^1}\frac{\partial z_i^1}{\partial b_k^1} \\ &= \left(\sum_{j=1}^{2}\frac{2}{N}(\hat{y}_j - y_j)w_{ji}^2\right)\sigma'(z_i^1).\end{aligned} \tag{3.10}$$

Once the gradient of the loss with respect to each trainable is obtained, a new value can be updated by:

$$w := w - \eta\frac{\partial \mathcal{L}}{\partial w}, \tag{3.11}$$

$$b := b - \eta\frac{\partial \mathcal{L}}{\partial b}, \tag{3.12}$$

where η is the learning rate. A proper learning rate value should be used in order to avoid no improvement on loss function with very low learning rate value and avoid divergence with too high value. In practice, the optimization of the loss function is implemented by more complicated stochastic gradient-based optimization such as stochastic gradient descent (SGD) Sutskever et al. (2013) and Adam algorithm Kingma and Ba (2014). The latter, which will be used in the later chapters, dynamically adapts the learning rate for each trainable based on estimates of first and second moments of the gradients. The pseudo-code of Adam algorithm is described in Algorithm 2.

3.3 Convolutional neural networks

3.3.1 Main components

A Convolutional neural network (CNN), whichever architecture it employs, mainly contains a cascade of convolutional layers, pooling layers and fully connected layer with drop-out. As shown in Fig. 3.5, a **convolutional layer** performs elementwise multiplication between the input tensor $f_{i,j}^l$ at location (i, j) and the rectangle convolutional kernel/filter $\theta_{2M+1,2N+1}$ of shape $(2M+1, 2N+1)$ producing the hidden feature $f_{i,j}^{l+1}$ as the input for the next layer:

$$f_{ij}^{l+1} = \sum_{m,n=-M,-N}^{M,N} f_{i+m,j+n}^l \star \theta_{m,n}^l, \tag{3.13}$$

where $\star$ represents the convolution operation. Each value in the next layer is only connected to a local region, namely the local receptive field, of the previous layer covered by

Algorithm 2: Adam stochastic optimization method. The variables are: first moment vector m, second moment vector v, learning rate η, exponential decay rates for the moment estimates β_1, β_2, loss function $\mathcal{L}$, trainables θ, timestep t, gradient g, weight decay λ and term added to denominator for the sake of stability $\epsilon = 1e^{-8}$.

initialize: $m_0 \leftarrow 0$, $v_0 \leftarrow 0$, $t \leftarrow 0$;
while θ_t *not converged* **do**
  $t \leftarrow t + 1$;
  $g_t \leftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1}) + \lambda\theta_{t-1}$ (Get gradients w.r.t. stochastic objective $\mathcal{L}$ at time step t);
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first order moment estimate);
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate);
  $\widehat{m_t} \leftarrow m_t / (1 - \beta_2^t)$ (Compute bias-corrected first moment estimate);
  $\widehat{v_t} \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second moment estimate);
  $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \widehat{m_t} / (\sqrt{\widehat{v_t} + \epsilon})$ (Update trainables);
end
return θ_t;

the shared kernel. Each kernel traverses the entire image at a specified stride to extract a certain shift-invariant feature pattern, thus leading to significantly reduced number of parameters. Sometimes in order to keep the image dimension unchanged, zeros are padded at the borders. The convolutional layers at shallower levels extract low-level features such as dots, lines and etc, whereas those at higher levels extract high-level features.

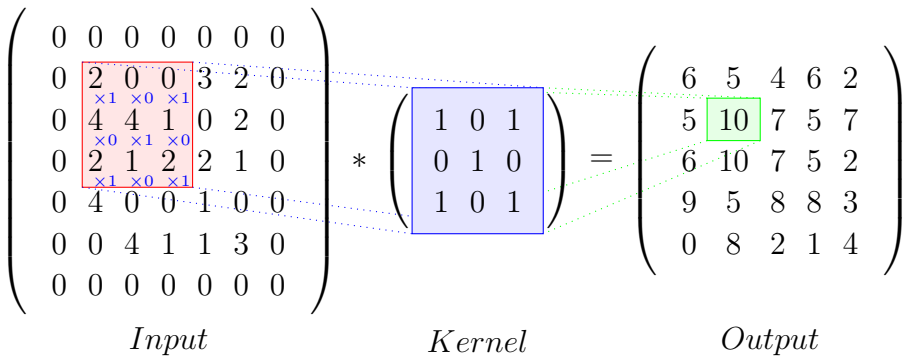


Figure 3.5: Convolution operation between a 5×5 input tensor padded with zeros at borders and a 3×3 kernel with stride of 1 resulting in a 5×5 output tensor.

Intuitively, stacking more convolutional layers can fulfill feature extraction mission but inefficiently since high-level features do not need high resolution. Therefore **pooling layers** are used to down-sample the data. As shown in Fig. 3.6, two primary types of pooling are mainly used: max pooling returns the maximum value within the input tensor covered by the pooling kernel, while average pooling returns the average of the corresponding patch of the input. Following the pooling layer, the number of values in the image is reduced by the pooling kernel size, resulting in a significant reduction in computational complexity. The inverse operation of pooling is upsampling which enlarges the image size. The most used two upsampling methods, bilinear interpolation and nearest interpolation, are shown in Fig. 3.7.

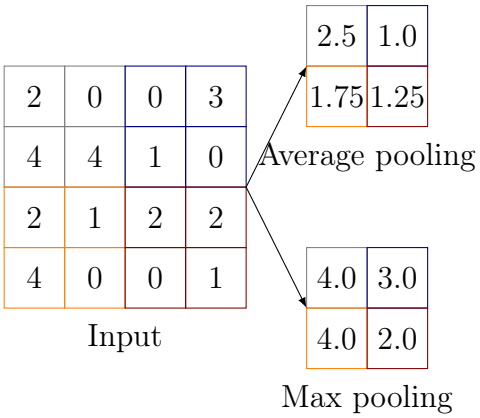
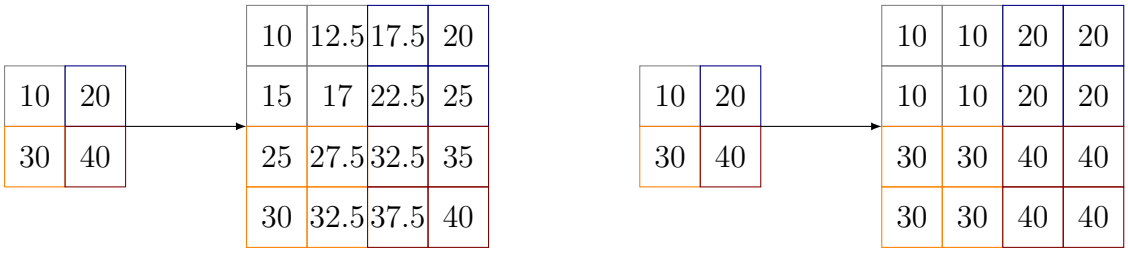


Figure 3.6: Two types of pooling operation of size 2×2 *with stride of 2.*



(a) Bilinear interpolation. *(b) Nearest interpolation.*

Figure 3.7: Two types of upsample operation of size 2×2 *with stride of 2.*

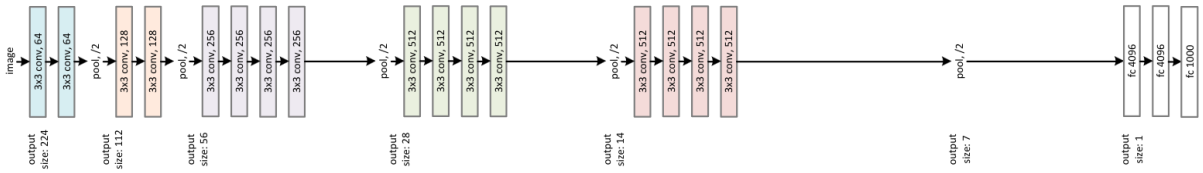
For image classification tasks, a fully connected layer is usually added to the end of the model to reshape the image into a column vector of the size corresponding to the number of categories. To mitigate overfitting, **dropout**, introduced by Hinton et al. (2012), is commonly applied in the final fully connected layer during the training stage. Dropout randomly removes a specified portion of connections between layers at each training epoch, resulting in error back-propagation not being performed for these removed connections. By using dropout, the prediction becomes less dependent on specific paths within the model, thus enhancing robustness.

3.3.2 CNN architectures

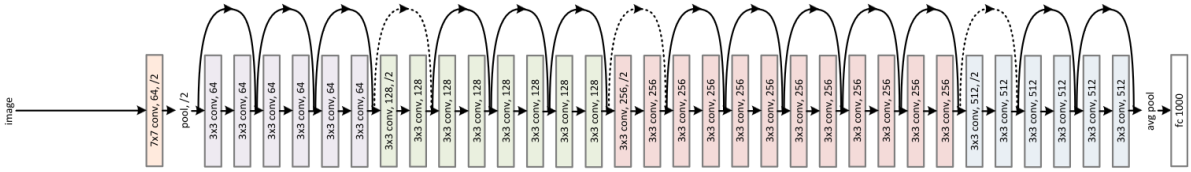
During the last decades, CNNs have experienced tremendous successes in numerous applications, such as image object detection and segmentation, natural language process Chen et al. (2018), audio classification Nanni et al. (2020); Hershey et al. (2017), etc, with various task-dependent implementations. Despite different forms, it is clear by looking back to the CNN development history that its general performance is largely decided by the architecture and each innovation on the architecture marks a significant improvement on object detection accuracy. There are three landmarks on the development of CNN architecture: 1) sequential architecture; 2) skip connection or residual connection architecture and 3) U-Net architecture.

Sequential architecture. The foundation of CNNs goes back to 1960s where artificial neural network with manually designed convolution kernels and non-linear activation function was proposed to extract the basic features, such as dots and line segments, from images by imitating to the same procedure in biological visual systems Fukushima (1969).

Later, LeCun et al. (1989) designed a convolution neural network with two convolutional layers to recognize the handwritten digits by applying back-propagation algorithm using stochastic gradient search to automatically optimize the trainable weights. Until 2010s, the depths of CNNs are relatively shallow, such as LeCun et al. (1998, 2010), no more than five convolutional layers used in AlexNet Krizhevsky et al. (2012) which showed higher categorical object classification accuracy than other methods in the large data-set ImageNet. VGG-19, as shown in Fig. 3.8a, Simonyan and Zisserman (2014) demonstrated that increasing the depth of network till 19 convolutional layers with very small (3×3) convolution kernels is conducive for the classification accuracy. However, further sequentially stack convolutional layers leads to the training convergence problem due to the vanishing/exploding gradient Bengio et al. (1994); Glorot and Bengio (2010).



(a) VGG-19



(b) ResNet-34

Figure 3.8: Two architectures adapted from He et al. (2016)

Skip connection architecture. In order to exploit the benefit of depth, He et al. (2016) proposed the residual network (ResNet) featured by the shortcut connections between every two succeeding layers which avoids the aforementioned optimization convergence problems even for a network of more than 1000 layers, even though such an unnecessarily deep architecture has higher test error than much shallower counterpart on small data-set. The ResNet-34 architecture with residual blocks is shown in Fig. 3.8b. A residual block is a shortcut connection and element-wise addition defined as:

$$y = \mathcal{F}(x, \theta_i) + x, \tag{3.14}$$

where function $\mathcal{F}(x, \theta_i)$ represents the residual mapping. A typical residual has two convolution layers, $\mathcal{F} = \theta_2\sigma(\theta_1 x)$ and ReLU activation function is used for σ, as shown in Fig. 3.9. It was demonstrated that using skip-connection can flat the loss landscape which enables the training to proceed smoothly Li et al. (2018).

U-Net architecture. Sequentially stacking more convolutional layers in theory linearly increases a model's receptive field, which is defined as the largest region in the input image from which a certain value in the output can gather information. However this strategy can quickly meet the performance ceiling showcased in the very deep, over 1000-layer model in He et al. (2016). It is due to the fact that the effective receptive field (ERF) is Gaussian distributed, which means that the peripheral pixels have much less influence on the prediction than the central ones, and much smaller than the theoretical value

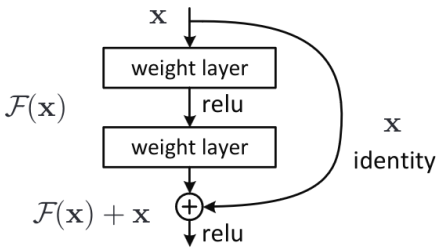


Figure 3.9: Residual block adapted from He et al. (2016)

Luo et al. (2016). Through dilated convolution kernel and down-sampling can efficiently increase ERF, the former still linearly increases ERF and can not be straightforwardly implemented in the graph neural networks thus loses the generalization. As a comparison, using down-sampling layers multiplicatively enlarges the ERF and is generalizable to GNNs. In this regard, U-Net is a very performant architecture which is demonstrated by the fact that it has been used in many models for different tasks, beyond the original biomedical image semantic segmentation task Ronneberger et al. (2015), including flow feature extraction Deng, Bao, Wang, Yang, Zhao, Wang, Bi and Guo (2022), surrogate model for flow field prediction Yang et al. (2022) and etc.

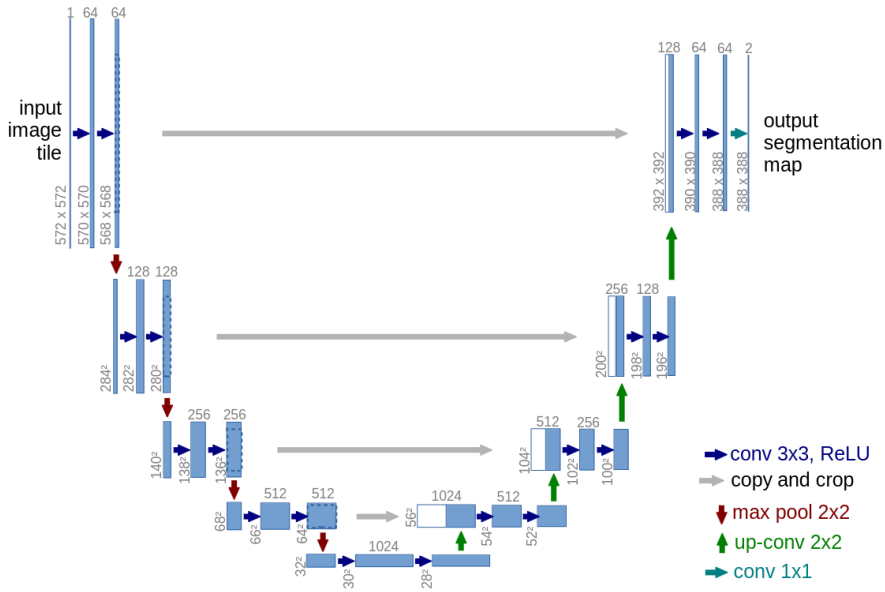


Figure 3.10: The original 5-depth 23-layer U-Net architecture for biomedical image segmentation adapted from Ronneberger et al. (2015).

As shown in Fig. 3.10, the original U-net architecture contains one contracting part (left side) as the encoder and one extracting part (right side) as the decoder. It has 5 depths and 23 convolutional layers, where each of them uses 3×3 convolutions followed by ReLU activation function. In the contracting part, 2×2 max pooling layer is added after two consecutive convolutional layers to decrease the image size by a factor of 4 while the number of feature channels doubles from one depth to the next deeper level. Correspondingly, a 2×2 up-sampling layer is used between layers at different depths to progressively restore the initial image size in the extracting part. One each level, the skip-connection exists between the leading convolutional blocks at the left side and

the following convolutional block at the right side which prevents the gradient vanishing problem.

3.4 Graph neural networks

CNN is designed for processing images, thus not suitable for unstructured data. However, unstructured data or graph-structured data is omnipresent, such as citation network, social network and traffic transport system, which graph neural networks (GNNs) can easily process. GNNs process the features stored on a stack of graph layers using a certain kernel function. Let $\mathcal{G}(\mathcal{V}, \mathcal{E}, A)$ denote a graph where $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ is the set of edges. $A \in \mathbb{R}^{N \times N}$ represents the adjacency matrix and N is the number of nodes. As shown in Fig. 3.11, the features can be stored on edges and nodes.

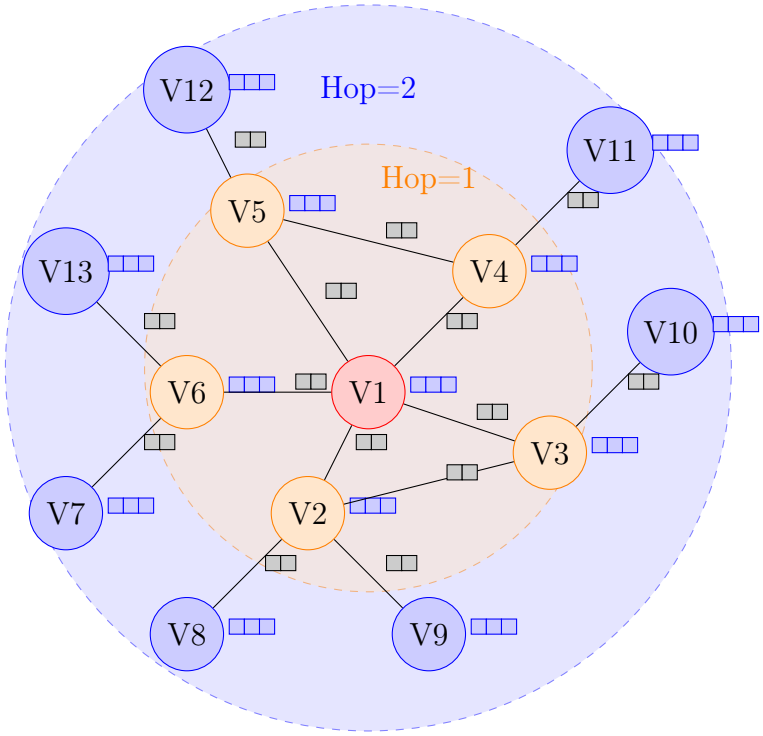


Figure 3.11: A graph with features stored on both nodes and edges. Circles represent the nodes and edges represent the connections between nodes. Both nodes and edges can store features represented by blue and grey squares, respectively. The nodes of 1- and 2-hops away from the center node $V1$ are colored in orange and blue respectively.

The diversity of CNNs is manifested by their architectures, as a comparison, the variety of GNNs is demonstrated in the numerous kernel functions depending on the task. Here we focus on graph convolutional neural networks. A graph convolution updates the features from one layer to the next generally following this equation:

$$f^{l+1}(i) = \sigma(\theta(f^l \star g^l)(i) + b^l), \tag{3.15}$$

where $f^l(i) \in \mathbb{R}^{C_l}$ and $f^{l+1}(i) \in \mathbb{R}^{C_{l+1}}$ represent the features on node i at l-th and $(l+1)$-th layer respectively, g is the convolution kernel, $\theta \in \mathbb{R}^{C_l \times C_{l+1}}$ is trainable weights which change the feature dimension from C_l to C_{l+1}, $b \in \mathbb{R}^{C_{l+1}}$ is the bias, and $(f \star g)$ represents the convolution operation which has a variety of different forms. The most

prominent types of graph convolutional kernels are that works on spectral domain and that on spatial domain. Two spectral graph convolutional kernels, ChebConv Defferrard et al. (2016) and graph convolutional networks (GCN) Kipf and Welling (2016), and two spatial graph convolutional kernels, Gaussian mixture model (GMM) Monti et al. (2017) and SplineCNN Fey et al. (2018), are introduced next. The source code of these kernels is shown in A.3.

3.4.1 Kernels on spectral domain

ChebConv. Defferrard et al. (2016) extended the convolutional kernel to spectral space by means of Chebyshev expansion of the graph Laplacian which lowers computational complexity by avoiding compute the eigenvectors of the Laplacian and yielding spatially localized filters. The essential operator in spectral graph analysis is the graph Laplacian defined as $L = D - A \in \mathbb{R}^{n\times n}$ where $D \in \mathbb{R}^{n\times n}$ is the diagonal degree matrix with $D_{ii} = \sum_j A_{ij}$, and normalized definition is $L = I_n - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ where I_n is the identity matrix. The orthonormal eigenvectors $\{u_l\}_{l-0}^{n-1} \in \mathbb{R}^n$ of L are the graph Fourier modes and the associated ordered real nonnegative eigenvalues $\{\lambda_l\}_{l=0}^{n-1} \in \mathbb{R}^n$ are the frequencies of the graph. The Laplacian is diagonalized by the Fourier basis $U = [u_0, ..., u_{n-1}] \in \mathbb{R}^{n\times n}$ such that $L = U\Lambda U^T$ where $\Lambda = diag([\lambda_0, ..., \lambda_{n-1}]) \in \mathbb{R}^{n\times n}$. The graph Fourier transform of a signal $x \in \mathbb{R}^n$ is then defined as $\hat{x} = U^T x \in \mathbb{R}^n$, and its inverse as $x = U\hat{x}$. The convolution operator on graph $\star$ is defined in the Fourier domain such that $x \star y = U((U^T x) \odot (U^T y))$, where $\odot$ is the element-wise Hadamard product. Thus a signal f is filtered by g_θ as

$$g_\theta(L) \star f = g_\theta(U\Lambda U^T)f = Ug_\theta(\Lambda)U^T f, \tag{3.16}$$

where f is the input feature. Evaluating Eq. 3.16 is computationally expensive due to the quadratic cost of the multiplication with the vector U. To solve this, a recursive fast filter as the truncated Chebyshev expansion

$$g_\theta = \sum_{k=0}^{K-1} \theta_k T_k(\tilde{\Lambda}) \tag{3.17}$$

of order $K - 1$ was proposed, where the parameter $\theta \in \mathbb{R}^{\mathbb{K}}$ is a vector of Chebyshev coefficients and $T_k(\tilde{\Lambda}) \in \mathbb{R}^{n\times n}$ is the Chebyshev polynomial of order k evaluated at $\tilde{\Lambda} = 2\Lambda/\lambda_{max} - I_n$, a diagonal matrix of scaled eigenvalues that lie in $[-1, 1]$. The filtering operation can then be written as

$$g_\theta(L) \star f = \sum_{k=0}^{K-1} \theta_k T_k(\tilde{L})f, \tag{3.18}$$

where $T_k(\tilde{L}) \in \mathbb{R}^{\mathbb{n}\times\mathbb{n}}$ is the Chebyshev polynomial of order k evaluated at the scaled Laplacian $\tilde{L} = 2L/\lambda_{max} - I_n$. Since it depends only on nodes that are at most K steps away from the central node, this filter is K-localized. The computation complexity is $\mathcal{O}(K|\mathcal{E}|)$, i.e. linear w.r.t. the filter support's size K and the number of edges $|\mathcal{E}|$.

GCN. In order to alleviate the problem of overfitting on local neighborhood structures for graphs with a wide variety of node degree distributions, Kipf and Welling (2016) proposed GCN by restricting the layer-wise convolution to $K = 1$ and stacking K such

layers to convolve K^{th}-order neighborhood of a node. They further approximate $\lambda_{max} \approx 2$, thus Eq 3.18 simplifies to

$$\begin{aligned} g_\theta \star f &\approx \theta_0 f - \theta_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} f, \\ &\approx \theta \left(I_n + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \right) f, \end{aligned} \tag{3.19}$$

where two free parameters θ_0 and θ_1 are reduced to a single parameter θ using $\theta_0 = -\theta_1$. To alleviate the numerical instabilities and exploding/vanishing gradients normally encountered in deep neural networks, the renormalization trick $I_n + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \rightarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$, with $\tilde{A} = A + I_n$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, leading to

$$g_\theta \star f = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} f \theta. \tag{3.20}$$

3.4.2 Kernels on spatial domain

GMM.

Monti et al. (2017) extended the traditional CNN convolution to graphs by using Gaussian Mixture Model. GMM kernel gathers the hidden features from neighbors to center node with the following equation:

$$(f \star g)(i) = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} f(j) \cdot g_k(\mathbf{e}_{ij}), \tag{3.21}$$

where hyperparameter K represents the number of directions learned in the kernel, $|\mathcal{N}(i)|$ denotes the degree of node i, the edge $\mathbf{e}_{ij}$ points from central node i to neighbor node j:

$$\mathbf{e}_{ij} = [x_j, y_j]^T - [x_i, y_i]^T, \tag{3.22}$$

and $g_k(\mathbf{e}_{ij})$ measures the alignment between the edge $\mathbf{e}_{ij}$ and the k-th direction μ_k in the kernel:

$$g_k(\mathbf{e}_{ij}) = \exp\left\{ -\frac{1}{2} (\mathbf{e}_{ij} - \mu_k)^T \sum\nolimits_k^{-1} (\mathbf{e}_{ij} - \mu_k) \right\}, k \in [1, 2, ..., K], \tag{3.23}$$

where two trainables μ_k and $\sum_k$ represent the learned directions in the kernel and its variance matrix respectively. The more aligned the edge $\mathbf{e}_{ij}$ and the k-th learned direction μ_k are, the higher $g_k(\mathbf{e}_{ij})$.

An ideal learned GMM kernel is visualised in Fig. 3.12. The elliptical circles represent the learnable directions in the kernel which are located at different distances to the center. All the edges that fall inside the circle are considered to be aligned with the corresponding direction.

SplineCNN. Fey et al. (2018) generalized the traditional CNN convolution kernel to interpolate the edge importance from fixed positions to desired positions using B-spline basis functions with the following equation:

$$(f \star g)(i) = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} f(j) \cdot \sum_{\mathbf{k} \in \mathcal{K}} B_{\mathbf{k}}(\mathbf{e}_{ij}) \cdot g_{\mathbf{k}}, \tag{3.24}$$

where $\mathcal{K}$ is the Cartesian product of the B-spline bases:

$$\mathcal{K} = N_{k_1,p}^1 \times ... \times N_{k_D,p}^D, \ \mathbf{k} = (K_1, ..., K_D), \tag{3.25}$$

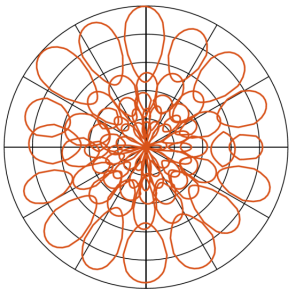


Figure 3.12: Ideal learned GMM kernel adapted from Monti et al. (2017). (Each elliptical circle in orange represents a learned direction and all the edges falling inside the circle are considered aligned with it.

$\mathbf{k}$ represents the numbers of control points on each dimension of the D-dimensional kernel, and $g_{\mathbf{k}}$ is the trainables and the control points as well associated with the corresponding product $B_{\mathbf{k}}(\mathbf{e})$ of the basis functions in $\mathcal{K}$:

$$B_{\mathbf{k}}(\mathbf{e}) = \Pi_{d=1}^{D} N_{k_d,p}^{d}(e^d). \tag{3.26}$$

The k-th B-spline basis function of degree p, written as $N_{k,p}(e^d)$, is defined recursively as follows:

$$N_{k,0}(e^d) = \begin{cases} 1, & \text{if } u_k \leq e^d < u_{k+1}. \\ 0, & \text{otherwise.} \end{cases} \tag{3.27}$$

$$N_{k,p}(e^d) = \frac{e^d - u_k}{u_{k+p} - u_k} N_{k,p-1}(e^d) + \frac{u_{k+p+1} - e^d}{u_{k+p+1} - u_k} N_{k+1,p-1}(e^d). \tag{3.28}$$

where u_k is the knot vector and e^d is the edge coordinate on d-th dimension.

Different from those used in Fey et al. (2018), multiple knots are used at the ends of x and y directions to force the interpolated surface to converge to the control points at the ends. Three example convolution kernel surfaces are visualised in Fig. 3.13. To guarantee a fair comparison with traditional CNN kernel, SplineCNN kernels are using B-spline of degree $p = 2$ with 3 control points are used in all the later experiments.

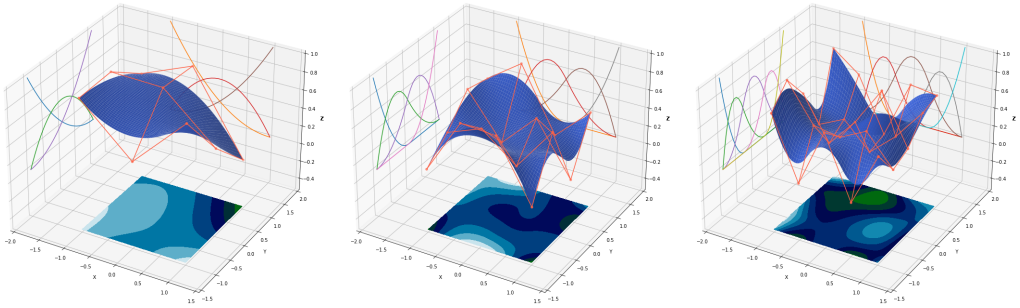


Figure 3.13: Examples of SplineCNN convolution kernels using B-spline basis of degree $p = 2$ with control points number $K_1 = K_2 = 3$, 4 and 5, from left to right. The orange points connected by orange wireframe represent the randomly sampled control points. The curves on the yz-plane and xz-plane represent basis functions for each control point on corresponding direction.

3.5 Summary

In this chapter, the supervised training procedure of neural networks along with their basic concepts are introduced firstly. The main components of convolutional neural networks, which are selected as the backbone of our algorithm to detect flow phenomena due to their effectiveness on extracting spatial features in images, are explained in detail. To overcome the CNNs' inability to consume unstructured data a proper graph neural network should be constructed inheriting the merits of CNNs. Therefore, graph convolution kernels equivalent to that in CNNs are reviewed and will be compared in later chapter.

4 Literature review of machine learning for fluid mechanics

Contents

Although the traditional methods in fluid mechanics, such as numerical simulations using partial differential equation (PDE) solvers and experimental measurement techniques like particle image velocimetry (PIV), have been widely used to solve various engineering fluid flow problems, they still face significant challenges. These include difficulties with inverse problems, experimental data assimilation, hidden pattern finding, and balancing high accuracy with low computational requirements. In contrast, machine learning (ML) algorithms, serving as universal approximators, can efficiently address these issues given sufficient data, which conventional methods can provide. As a result, the fluid mechanics community has adopted different ML approaches to solve different problems, as summarized in Fig.4.1 by Sharma et al. (2023). For current research, it is necessary and conducive to have a comprehensive global view of ML applications in fluid mechanics. This chapter aims to provide extensive overview of how ML is being applied in the field of fluid mechanics.

This chapter is organized as follows. Section 4.1 covers the latest advancements in using ML, particularly Convolutional Neural Networks (CNNs), to identify flow phenomenon in CFD results and observed data. Section 4.2 shows how MLs can be employed as reduced order models (ROMs) to compress CFD data and accelerate engineering design and optimization. Section 4.3 includes the recent progress in using ML to imporove and upscale low resolution data to high resolution data. Section 4.4 reviews strategies to incorporating physical principles into ML models to ensure physics-consistent predictions.

4.1 Flow phenomena identification

Most of the researches that aim at identifying flow phenomena focus on vortex detection because of both its importance on understanding the flow mechanism and its difficulties using the current local and global detection methods. Local vortex detection methods, imply the use of specific criteria such as the Vorticity ω, the Q-criterion, the λ_2-criterion

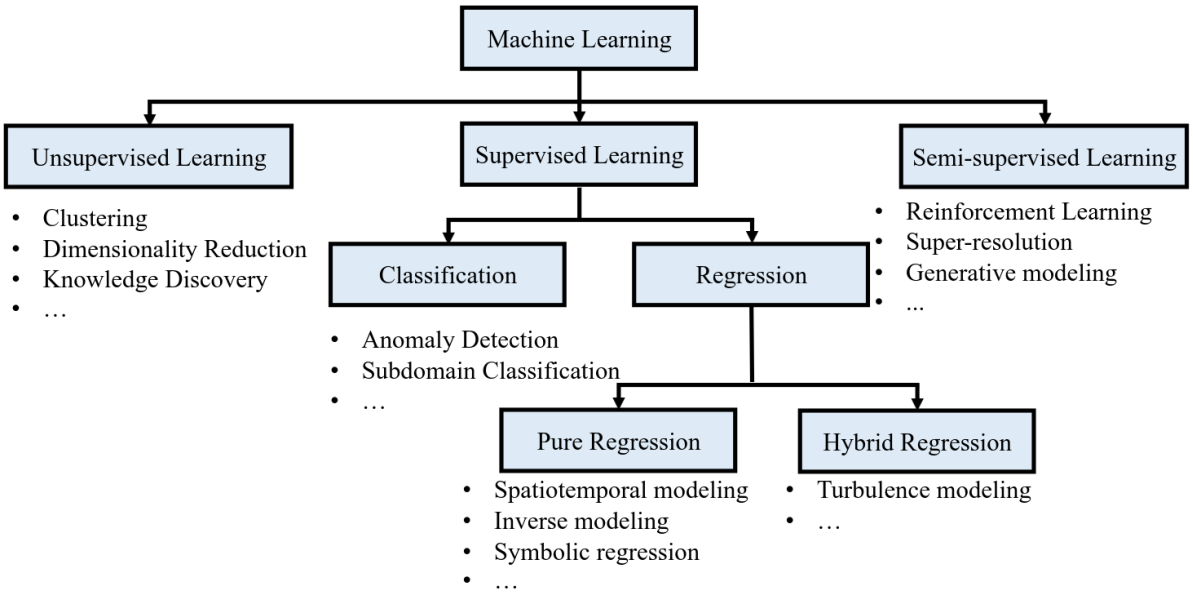


Figure 4.1: ML applications in fluid mechanics. Adapted from Sharma et al. (2023).

and Δ-criterion, that all rely on the gradient of the velocity field.
The Vorticity ω is defined as:

$$\omega = \nabla \times \mathbf{u}. \tag{4.1}$$

The Q-criterion Hunt (1987) considers the region

$$Q = \frac{1}{2}(||\mathbf{\Omega}||^2 - ||\mathbf{S}||^2) > Q_{thresh} \tag{4.2}$$

as vortex region, where Ω and S are the rotation tensor and strain-rate tensor:

$$\mathbf{\Omega} = \frac{1}{2}(\nabla\mathbf{u} - \nabla\mathbf{u}^T),\ \mathbf{S} = \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T). \tag{4.3}$$

λ_2-criterion Jeong and Hussain (1995) is the second largest eigenvalue of $S^2 + \Omega^2$.
Δ-criterion proposed by Chong et al. (1990) defines the vortex region as:

$$\Delta = \left(\frac{Q}{3}\right)^3 + \left(\frac{det(\nabla\mathbf{u})}{2}\right)^2 > 0, \tag{4.4}$$

where $det(\nabla u)$ is the determinant of velocity gradient tensor. All these methods not only require a user specified threshold which greatly limits the generality from case to case, but also generate false and missing detections compared with global methods.

The instantaneous vorticity deviation (IVD) proposed by Haller et al. (2016) is one of the global detection methods which is defined as the absolute value of the difference between the vorticity at a point and the spatially averaged vorticity of the global field larger than a threshold as vortex region:

$$IVD(\mathbf{x}, t) = |\omega(\mathbf{x}, t) - \bar{\omega}(\mathbf{x}, t)|. \tag{4.5}$$

This method should select the IVD-based boundaries on 2D planes that enclose the local maxima. Though the IVD method is objective, it is computationally intensive and require too much computing time. As a result, it is not suitable for large-scale applications.

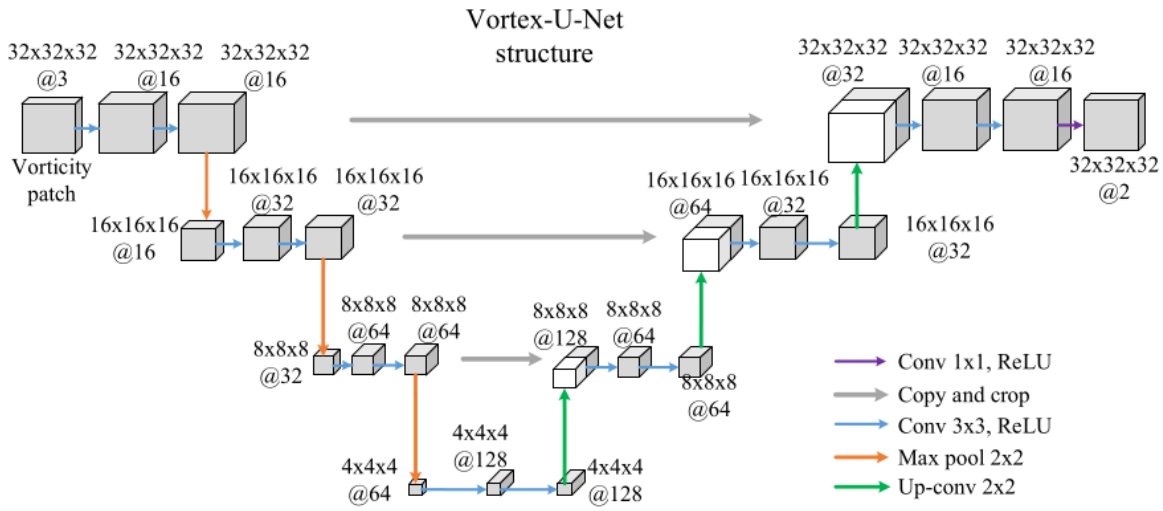


Figure 4.2: Vortex-U-Net from Deng, Bao, Wang, Yang, Zhao, Wang, Bi and Guo (2022) contains three steps. The preprocessing step transforms non-uniform velocity fields into uniform vorticity patches as the inputs of the neural networks. The neural network generates the predicted labels for all input patches. The post-processing step puts together all predicted label patches to obtain the whole predicted field.

Though there are other ML approaches applied to identify vortex structures such as recurrent neural network Rajendran et al. (2018), the most successful ones are CNNs Ye et al. (2019, 2020); Monfort et al. (2017); Deng et al. (2019); Bai et al. (2019); Berenjkoub et al. (2020); Ströfer et al. (2018); Kim and Günther (2019); Wang, Guo, Wang, Deng, Wang and Li (2020). Deng et al. (2019) proposed a Vortex-Net model to identify the vortex region with velocity as input which achieved higher precision and recall on the identification compared with the local methods and is faster than global methods such as IVD. Vortex-Seg-Net proposed by Wang et al. (2021) discards the fully-connected layer in the Vortex-Net to reduce the number of parameters to be more computationally efficient and deepens the model depth to improve the identification performance. Deng, Bao, Wang, Yang, Zhao, Wang, Bi and Guo (2022) proposed the Vortex-U-Net model, as shown in Fig. 4.2, to identify vortex region feeding on local patches in the vorticity field in 3D cases. The IVD, a global vortex detection method, was used as the ground-truth label. Vortex-U-Net model achieves higher vortex identification accuracy in terms of recall and F1 score than the local detection methods and its execution time is two order of magnitude lower than that of IVD. A multi-view U-Net (MVU-Net) was proposed by Deng, Chen, Wang, Chen, Wang and Liu (2022) to combine multiple variables while reducing the model complexity due to the juxtaposition of three identical U-Net models with each feeding on one of three variables: $\mathbf{u}$, ω and IVD. The comparative work of Berenjkoub et al. (2020) shows that after being trained on synthetic dataset the 8-layer U-Net CNN model outperforms both the 2-layer plain CNN model and Resnet20 model on the accuracy of the vortex boundary extraction. As the most utilized visualization method to analyze the flow structure in CFD result, streamlining in the entire domain is computationally prohibitive thus leading to missing some of the prominent flow structures. The CNN-based deep regression model (DRM) proposed by Lee and Park (2021) reduces the streamline creation time required to reveal the prominent flow features.

However, the design of the rectangular-like convolution kernel limits the CNNs to only accept the data stored in Cartesian grid which constrains its generality to the data stored

in unstructured meshes widely used in the industrial CFD cases. Therefore, most of the current applications of CNNs on detecting flow phenomena are limited to simple flow cases. When dealing with the unstructured meshes and complex geometries, either the mesh deformation Ströfer et al. (2018); Wang, Guo, Wang, Deng, Wang and Li (2020); Deng, Bao, Wang, Yang, Zhao, Wang, Bi and Guo (2022) or data interpolation Ye et al. (2020) are used which are cumbersome and inevitably introduce numerical errors.

4.2 Reduced order model

Despite recent advancements in computational resources, high-fidelity simulations like direct numerical simulation (DNS) and large-eddy simulation (LES) remain prohibitive for turbulent flows in high Reynolds number regions with complex geometries. Even simulations with Reynolds-averaged Navier-Stokes (RANS) models, which are less computationally demanding, become costly when optimizing a large number of parameters. Reduced order models (ROMs), such as proper orthogonal decomposition (POD), can be used to expand the database from a limited set of full order model results Sinha et al. (2022).

POD, initially introduced by Lumley (1967), serves to represent the spatio-temporal velocity field as a combination of an infinite number of spatial structures with amplitudes varying in time, forming linearly uncorrelated modes. The POD modes are ranked according to the preserved energy, with higher modes being prioritized, and then truncated. However, as the Reynolds number increases, given a fraction of the total energy, the convergence rate of the truncated POD modes decreases due to nonlinear interactions among different modes in the turbulent flow - a phenomenon characterized by the slow decay rate of the Kolmogorov n-width, as noted by Ahmed and San (2020). Milano and Koumoutsakos (2002) showed for the first time that linear multilayer perceptrons (MLPs) are equivalent to the POD, whereas nonlinear MLPs are superior in compression and reconstruction capabilities. A similar conclusion was drawn for CNNs with autoencoder (AE) architecture later by Murata et al. (2020). Furthermore, Lee and Carlberg (2020) demonstrated that using deep convolutional autoencoders to project the dynamical systems onto nonlinear manifolds can outperform the classical linear-subspace ROMs like POD, even overcoming the Kolmogorov n-width barrier in advection-dominated flows, albeit at significant higher computational cost.

Plenty of ROMs used CNN-AE as the feature extractor to build low-order models for different flows, such as advection-dominated systems Maulik et al. (2021), laminar and turbulent flows Fukami, Hasegawa, Nakamura, Morimoto and Fukagata (2021) and 3D incompressible and transient flow Akkari et al. (2022). Despite the widely accepted use of CNN-AEs as ROMs, the comparison study of fully connected neural network (FCNN), CNN and graph convolutional neural networks (GCNN) against POD show that the superior architecture highly depends on both the size of latent space and the task Gruber et al. (2022). Other drawbacks of AE architecture, such as the learned entangled latent and a blindly predefined large enough latent dimension, make the these ROMs physically uninterpretable and inefficient. Targeting on these problems, β-variational autoencoder (VAE) introduced by Higgins et al. (2016) was employed by many researchers to disentangle the latent variables by adjusting the Kullback-Leibler (KL) divergence term Kullback and Leibler (1951) with a weighting factor β in the original VAE Kingma and Welling (2013) loss function:

$$\mathcal{L}(\mathbf{x}) = \underbrace{\frac{1}{N_t}\sum_{i=1}^{N_t}(\mathbf{x}-\tilde{\mathbf{x}})^2}_{\text{Reconstruction loss}} + \underbrace{\frac{\beta}{2}\sum_{i=1}^{d}(1+\log(\sigma_i^2)-\mu_i^2-\sigma_i^2)}_{\text{- KL divergence}} \tag{4.6}$$

It was found that increasing β suppresses the KL-divergence while the reconstruction loss increases and the latent variables become more disentangled. Wang et al. (2024) put forward a CNN-β-VAE model, as shown in Fig. 4.3, to reduce the dimensionality of the transient fields in the wall-mounted square cylinder. Comparison with the results obtained by POD method suggest that with the same number of decomposition modes, CNN-β-VAE can recover more information of the flow field. Increasing the latent dimension enhances the reconstruction quality while sacrificing the orthogonality of the decomposed modes. With larger β value, the CNN-β-VAE model tend to learn a more orthogonal latent space at the cost of lower reconstruction precision. To address the uninterpretability problem, Kang et al. (2022) made the latent variables in β-VAE physics aware by introducing the Gaussian process regression (GPR) to correlate the latent variables with the input physical parameters. Their results confirmed that the proposed physics aware β-VAE can pinpoint the latent variables corresponding to the generating factors, such as Mach number and angle of attack (AoA) in the transonic flow around airfoil case.

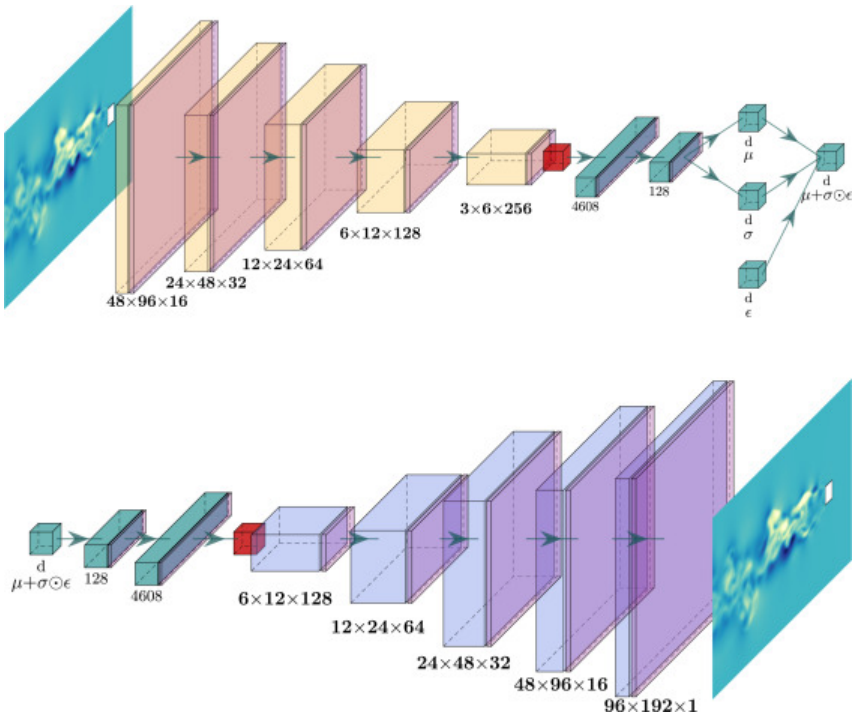


Figure 4.3: CNN-β-VAE architecture extracted from Wang et al. (2024).

Most of the ML ROMs focus on the structured meshes using CNNs. When applied to unstructured meshes where the data reshaping is necessary, the CNN reconstruction accuracy degrades. This deficiency of CNN can be overcome by utilising GCNN as evidenced in framework of Gruber et al. (2022) based on the GCNII graph convolution operation Chen et al. (2020). More GNN-based ROMs for unstructured meshes can be expected in the future.

4.3 Super-resolution

Driven by the various demands in fluid dynamics such as data compression of DNS simulations, acceleration of high fidelity CFD simulations Kochkov et al. (2021) and recover more details from sparse and irregular particle image velocity data points Cai et al. (2019); Morimoto et al. (2021), and benefited from the state of the art development in image super-resolution Dong et al. (2015); Yang et al. (2019); Lu et al. (2022); Li et al. (2022); Karras et al. (2017); Ho et al. (2020), different super-resolution models have been proposed recently to reconstruct high-resolution CFD results from the low-resolution counterparts Fukami et al. (2023).

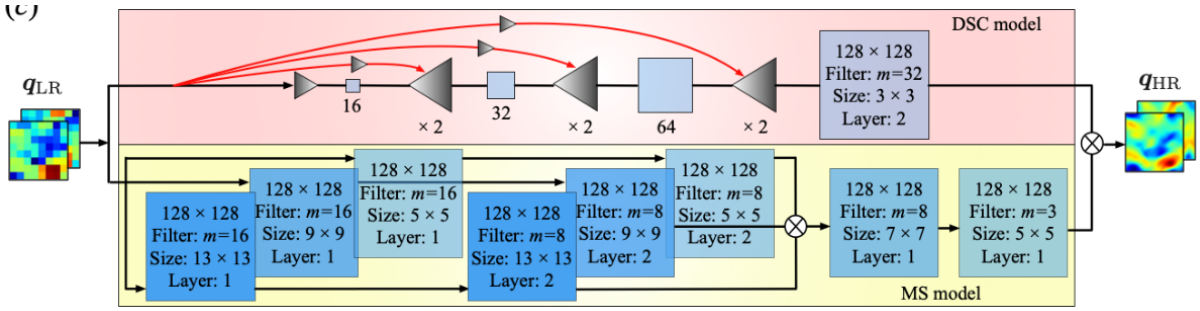


Figure 4.4: Schematic of the hybrid Downsampled Skip-Connection Multi-Scale (DSC/MS) model extracted from Fukami et al. (2019).

As shown in Fig. 4.4, Fukami et al. (2019) devised a hybrid downsampled skip-connection/multi-scale (DSC/ MS) model to reconstruct the super-resolution results from low-resolution results. The proposed DSC/MC model demonstrated superiority on the very coarse cases than traditional methods like bicubic interpolation in recovering more details. This model is later extended to recover the temporal evolution of flow field between two time instants Fukami, Fukagata and Taira (2021). Obiols-Sales et al. (2021) proposed a transfer learning-based super-resolution flow network SURFNet. The tested two types of transfer learning strategies, one shot transfer learning (OSTL) where the network is pretrained on the coarsest grid and then transferred to the finest grid dataset in one shot, while incremental transfer learning (ITL) passes the network gradually through intermediate resolutions step-by-step from the lowest resolution to the target resolution. As a comparison of the increasing loss of OSTL method as the resolution ratio increases, ITL loss remains relatively constant and much lower. Compared with baseline model, the transfer learning significantly reduces the data collection size and the training time. More super-resolution works can be found in Zhou et al. (2022); Liu et al. (2020); Xie et al. (2018); Hu et al. (2024); Bao et al. (2023).

The above direct mapping models suffer from two problems: the reconstructed flow field does not satisfy the continuity and the reconstruction accuracy degrades once applied to low-fidelity input that significantly deviates from the training dataset in terms of resolution or a Gaussian blurring process. While adding physics based loss terms like continuity loss to the conventional L_p norm loss seems a plausible remedy, Pant and Farimani (2020); Li and McComb (2022) suggested that such method even deteriorates the reconstruction accuracy. For the latter problem, these direct mapping models have to be retrained when a out-of-distribution test data is given. To address these problems, Shu et al. (2023) cast super-resolution as a problem of denoising and proposed a denoising diffusion probabilistic model (DDPM) which incorporates the PDE information. This

diffusion based model is trained to minimize the Kullback-Leibler divergence between the forward diffusion process and the backward diffusion process. As a result, it is only sensitive to data reconstruction error in the statistical sense with the presence of Gaussian noise. Compared with direct mapping models, the proposed DDPM achieved similar L_2 loss and remains accurate in terms of kinetic energy spectrum and PDE residual loss for different input data distributions.

While the supervised training requires the paired low resolution input and high resolution output which hinders the practical application of super-resolution works, the unsupervised training models especially generative adversarial network (GAN) are the promising ones to avoid this problem. Yousif et al. (2021) designed a multi-scale enhanced super-resolution generative adversarial network (MS-ESRGAN), as indicated in Fig. 4.5, to reconstruct the high-resolution wall-bounded turbulent flow fields from sparse information. Besides the regular adversarial loss term used in GAN-based models, the other three loss terms, L_{pixel}, $L_{perceptual}$, $L_{gradient}$ and $L_{ReynoldsStress}$ indicating the pixel-wise regression difference, extracted features difference, the gradient difference, Reynolds stresses difference between the generated data and the ground truth data, respectively, are combined and weighted as the discriminator loss function. The model is trained and tested on two turbulent channel flow DNS results at friction Reynolds numbers $Re_\tau = 180,\ 550$. The reconstructed fields agree remarkably well the ground-truth data on the Reynolds stresses, velocity profiles in the boundary layer and the energy spectra as well. Kim et al. (2021) applied the CycleGAN Zhu et al. (2017) to generate DNS-resolution flow field from LES-resolution result. CycleGAN outperformed the bicubic interpolation and supervised method on reproducing the small-scale structures, energy spectra. More GAN-based super-resolution models were developed for multiphase fluid simulations Li and McComb (2022), and CFD acceleration Kochkov et al. (2021).

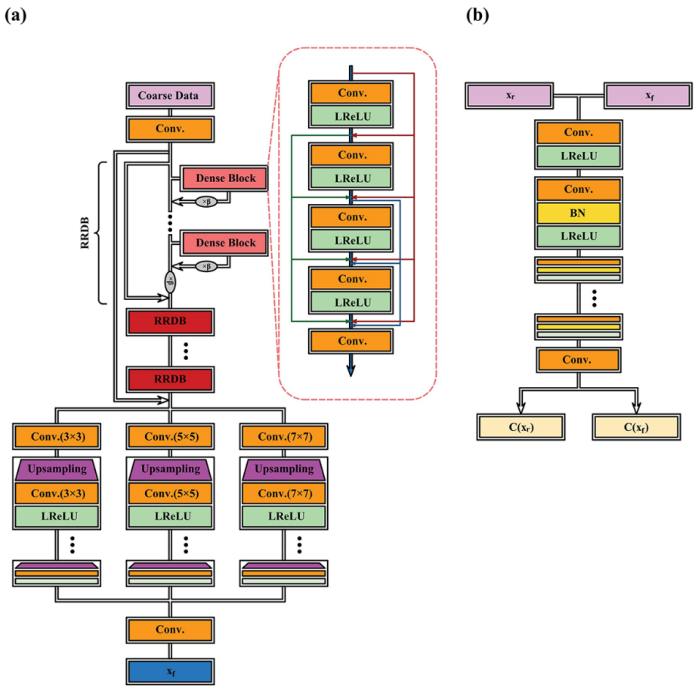


Figure 4.5: MS-ESRGAN architecture adapted from Yousif et al. (2021): (a) generator and (b) discriminator.

4.4 Physics-informed Machine learning

While the conventional PDE solvers are widely employed to forward problems, they fall short when it comes to addressing ill-posed problems such as noisy or incomplete boundary conditions, assimilating observed data into calculations, or solving inverse problems to infer model parameters from observations Raissi and Karniadakis (2018). In contrast, machine learning (ML) models have been employed to address these challenges by finding hidden patterns in noisy data. However, purely data-driven ML models tend to perform well only on small training datasets due to their strong universal approximation capability, but they struggle to generalize to data outside the training distribution. Their predictions often do not adhere to physical laws such as continuity and momentum conservation. By incorporating physical knowledge into ML models, these shortcomings can be mitigated. Embedding Domain knowledge can occur in three key aspects of the ML framework: a) model input and output, b) model architecture and c) loss function Karniadakis et al. (2021); Sharma et al. (2023). This integration not only reduces the reliance on large datasets but also ensures that predictions align with physical principles and exhibit better generalization to unseen data.

Physics-informed input and output. The first approach involves incorporating physical intuitions and principles through data preprocessing and feature selection. This physics embedding strategy is widely employed in the data-driven turbulence modeling. Ling, Jones and Templeton (2016) compared methods for constructing invariant inputs against methods using multiple transformations of raw input data for turbulence modeling. They found that embedding invariance properties into input features led to better performance at considerably reduced training cost. This finding is further validated by the tensor basis neural network (TBNN) introduced by Ling, Kurzawski and Templeton (2016). Their work used TBNN to model the relationship between five Galilean invariants and the coefficients of ten tensor basis to reconstruct the Reynolds anisotropy tensor using a multiplicative layer neural network as shown in Fig. 4.6.

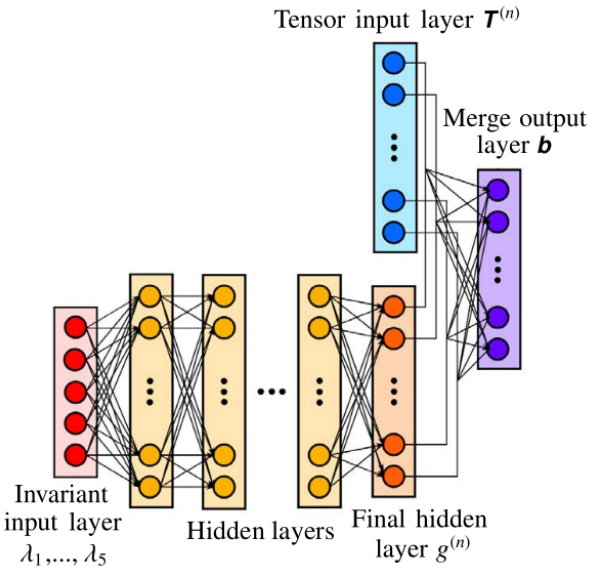


Figure 4.6: Multiplicative layer neural network proposed by Ling, Kurzawski and Templeton (2016)

In order to take into account the effect of the wall curvature, Xie et al. (2021) used

a local artificial neural network (LANN) to reconstruct the turbulent heat flux using average input features transformed into a local reference frame orthogonal to the nearest wall. The LANN model produced more accurate results than algebraic turbulence models and closely matched DNS results for periodic hill flow with varying slope ratios at different Reynolds numbers. However, directly incorporating Reynolds stress tensors predicted by ML models into RANS turbulence models can lead to significant velocity deviations, as errors from high-fidelity data and ML model are amplified Wu et al. (2019); Brener et al. (2021). Instead of reconstructing Reynolds stress tensors, Cruz et al. (2019) focused on modeling the Reynolds force vector - the divergence of Reynolds stress tensors - using a two-layer neural network. This network used the strain rate tensor and the non-persistence-of-straining tensor proposed by Thompson and de Souza Mendes (2011), and their divergence as input features. This approach produced smaller propagation errors of the Reynolds stress tensors in the mean momentum balance equation, leading to a more accurate secondary flow predictions in the square duct flow compared to using Reynolds stress tensors directly. Using physics-informed features and labels, as the simplest way to introduce the physics, is the weakest mechanism to embed physics into the model and still requires a considerable amount of data.

Physics-informed architecture. The second approach involves embedding physics into customized functions or architectures of the ML model. Ling, Kurzawski and Templeton (2016) proposed a tensor basis neural network (TBNN) with two multiplicative input layers to embed Galilean invariance for reconstructing the Reynolds anisotropy tensor. To ensure translation invariance, transport linearity and Galilean invariance required by the turbulent scalar transport equation, Frezat et al. (2021) introduced a subgrid transport neural network (SGNN). This SGNN features customized periodic padding and two parallel convolution operators - one without activation for processing velocity and another for processing the transported scalar. The SGNN framework outperformed algebraic and non-physical constrained models in predicting turbulent scalar flux in extrapolated flow regimes and showed better stability as a surrogate model in LES. In contrast to ensuring rotational invariance through data augmentation (extending the dataset by rotating available data) as done by Frezat et al. (2021), Wang, Walters and Yu (2020) achieved this by applying the E(2)-CNN proposed by Weiler and Cesa (2019), which outperformed the baselines trained with data augmentation.

To address different scales in turbulent flow, Wang, Kashinath, Mustafa, Albert and Yu (2020) developed the Turbulent-Flow Net (TF-Net), as shown in 4.7. This network contains three independent encoders for velocity components at three scales - time-averaged mean flow $\overline{w}$, spatially-filtered large-scale component $\tilde{w}$, and the fluctuating quantity w' - and a shared decoder. Each encoder-decoder pair can be seen as a U-Net. This customized architecture demonstrated significant advantages in predicting turbulent Rayleigh-Bénard convection flow, reducing root mean square error (RMSE) and flow divergence, and perserving physical quantities such as turbulence kinetic energy and energy spectrum. It outperformed both purely data-driven spatiotemporal deep learning models and hybrid physics-informed deep learning models like the Deep Hidden Physics Model (DHPM) by Raissi (2018).

Physics-informed loss function. The third strategy incorporates physics directly into the loss function, a technique often used in the so-called physics-informed neural networks (PINN) first introduced by Raissi et al. (2017). By integrating the residual loss of the governing equations, the training process of the neural network is regularized, resulting in significantly reduced training data and costs. While there is less interest in

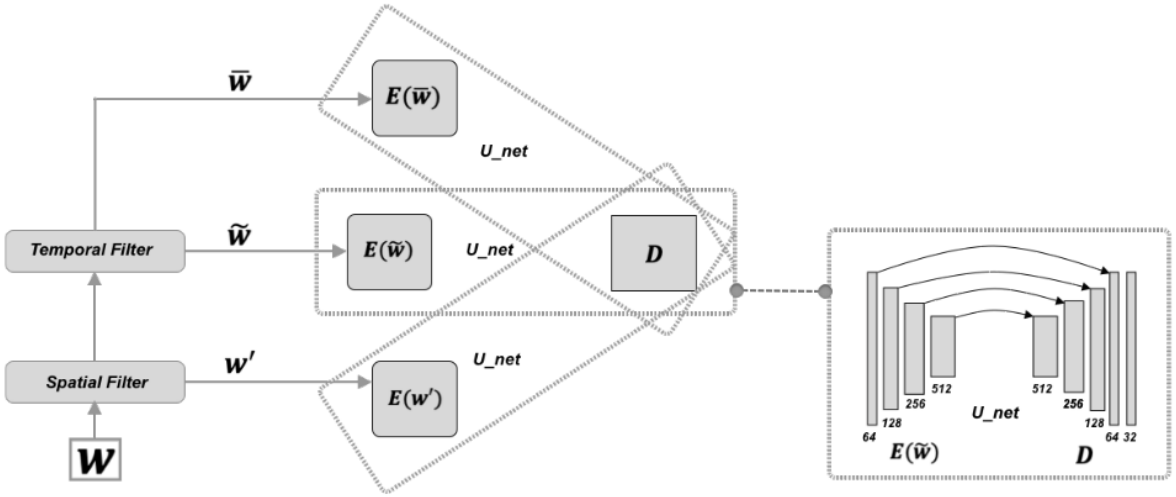


Figure 4.7: Turbulent-Flow Net: three identical encoders to learn the transformations of the three components of different scales, and one shared decoder that learns the interactions among three scales to predict 2D velocity field at the next instant. Extracted from Wang, Kashinath, Mustafa, Albert and Yu (2020).

solving forward problems, which can be addressed using classical PDE solvers, solving inverse problems - such as deducing physical fields in the entire fluid domain from sparse observed data - has gained more attention.

Hanrahan et al. (2023) developed a PINN, as shown in Fig. 4.8, to reconstruct Reynolds stress from sparse experimental data points. The proposed PINN contains a deep neural network (DNN) to predict the physical quantities field from the coordinates, followed by automatic differential (AD) layers to calculate the derivatives. The loss function is defined as:

$$L = \underbrace{\frac{1}{N_b}\sum_{j=1}^{N_d}\sum_{i=1}^{N_b}|\mathbf{T}_{ij} - \hat{\mathbf{U}}_{ij}|^2}_{L_{Data}} + \lambda_r \underbrace{\frac{1}{N_r}\sum_{j=1}^{N_e}\sum_{i=1}^{N_r}\epsilon_{ij}^2}_{L_{PDE}} \tag{4.7}$$

where L_{Data} represents the mean square error (MSE) between the network's predictions and the training data, and L_{PDE} is the residual loss of the governing equations, which weighted by the factor λ_r. The results demonstrated that while the network could predict the reattachment point from sparse experimental data, it failed to accurately predict velocity magnitudes in the recirculation region if no data points were provided within this area. The authors concluded that using PINNs with open RANS equations remains a data-driven methodology. This means that adjusting network hyperparameters does not improve prediction convergence; instead, increasing the amount of training data enhances prediction quality.

The traditional PINNs stuggle with multi-scale problems, such as boundary layer prediction, where high-frequency features in the refined near wall region make it difficult for the PINNs to converge. To address this, Huang et al. (2024) designed a multi-scale PINN (msPINN) that includes two independent neural networks for solving outer region and boundary layer inner region, respectively. While the traditional PINNs fail to capture the asymptotic velocity profile in the semi-infinite flat plate flow, the msPINN's predictions align well with reference numerical results.

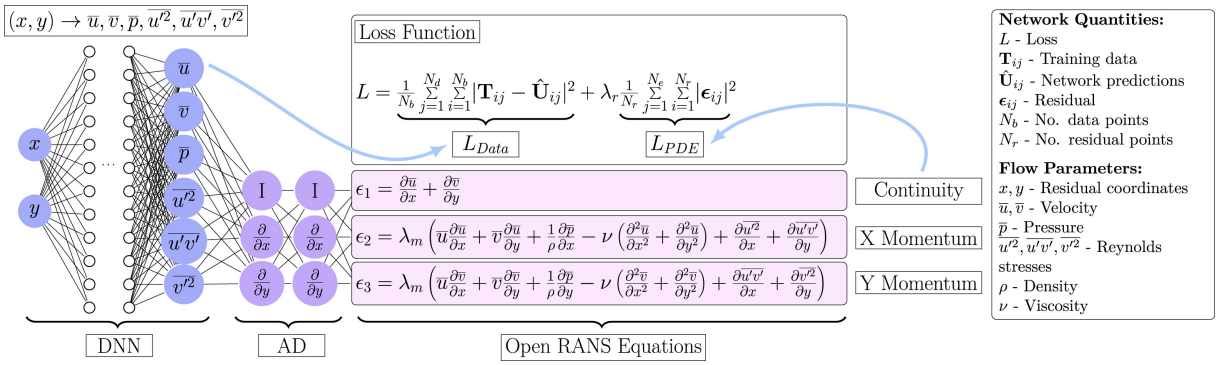


Figure 4.8: The PINN architecture with embedded RANS equations from Hanrahan et al. (2023).

4.5 Summary

Recent fast advancements in neural networks, particularly deep learning, has made it a powerful alternative for addressing longstanding problems in fluid mechanics. Deep learning has achieved success in different tasks, including flow phenomena identification, reduced order modeling, and the super-resolution of the flow field. However, there is no universally accepted superior design for even the same task, making benchmarking among different customized models both critical and challenging.

Another issue with deep learning is the generalizability. Data-driven algorithms provide meaningful predictions when testing data aligns well with training data, but they lose accuracy and reliability otherwise. Incorporating physics into data, model architecture, and loss function can extend the model's applicability to broader range of senarios. However, at this stage, deep learning remains a black box, introducing additional uncertainty and inexplainability to fluid mechanics when applied directly to CFD solvers. Users should exercice caution in such applications.

5 CNN identification of flow phenomena on structured meshes

Contents

Deep learning has been employed to identify flow characteristics from Computational Fluid Dynamics (CFD) results to assist the researcher to better understand the flow field, to optimize the geometry design and to select the correct CFD configuration for corresponding flow characteristics. Among all the machine learning algorithms, convolutional Neural Network (CNN) is the most popular algorithm used to extract and identify flow features. Application of CNNs in CFD domain has at least several differences compared with feature detection/object identification in the images: 1) depending on the nature of the task, the input features are different from case to case; 2) CNNs' generality is limited not only by the small size of the dataset and 3) CFD simulations have more selections, such as various mesh refinement levels, different turbulence models and flow regimes, which significantly confines a trained CNN to the same dataset over which it is trained.

In this chapter, the limitations of CNNs' application on flow feature identification is firstly explored through prediction of the pressure distribution around cylinder wall based on the velocity field in the wake region, concerning their performance on datasets constructed using different data interpolation methods on different types of meshes and meshes of varied refinement level. Then the whole process of CNN identification of independent vortexes behind 2D backward-facing step is investigated, including ground-truth label generation, input feature selection and result analysis. The feasibility of using CNN

to identify other flow phenomena such as thermal stratification region is showcased at the end.

5.1 Prediction of the pressure around cylinder from velocity field

Many factors may influence the prediction accuracy of a trained machine learning algorithm, such as data sampling strategy, data interpolation method, CFD mesh type and so on. To investigation to which extent these factors influence model's accuracy and generality, a series of CNNs are trained on different datasets to predict the pressure distribution around cylinder based on the velocity field in the wake region. As shown in Fig. 5.1, our CNN architecture contains two convolutional layers to extract spatial features, a flatten layer and a dense layer to map the learned hidden features to 32 pressure coefficient, $C_p = p/\frac{1}{2}\rho u^2$, around the cylinder. The input to the model is velocity components on x and y directions which are sampled from an array of points in the wake region.

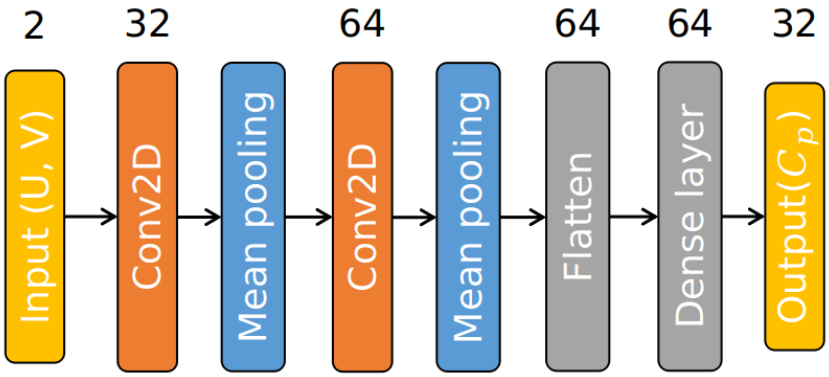


Figure 5.1: CNN model architecture to predict pressure distribution around cylinder. The number over each block indicates the channel width.

5.1.1 Dataset generation

The transient vortex shedding behind the cylinder is simulated using $k-\omega$ SST turbulence model with the geometry shown in Fig. 5.2. The parabolic velocity profile is imposed at the inlet boundary: $u(y) = 4u_m y(H-y)/H^2$, where H is the channel width, u_m is the maximum velocity. Ten flow cases with Reynolds number ranging from 10^3 to 10^4 with the interval of 10^3 are calculated. Reynolds number here is defined as $Re = \bar{u}D/\nu$ where D is the cylinder diameter and $\bar{u} = 2/3u(H/2)$. The second-order linear upwind (SOLU) scheme is used for the convective term of momentum equations and turbulence equations. The second-order Crank-Nicolson time-stepping scheme is used. The time step is set to 0.005s.

As the reference training dataset, the velocity components (U, V) at 41×17 points, as shown in Fig. 5.2, in the wake region are sampled from 300 consecutive time steps after the flow is fully developed are sampled as the input for the CNN model. The pressure coefficient C_p at 32 points evenly distributed around the cylinder is also extracted as the ground-truth label. The cases of $Re = [1000, 3000, 5000, 7000, 9000]$ are used to form the training dataset while cases of $Re = [2000, 4000, 6000, 8000, 10000]$ are used as the test dataset.

Table 5.1: Flow around cylinder meshes information.

Mesh type	Unstructured	Structured		
		Coarse	Medium	Fine
Cells	8132	9600	4×9600	16×9600

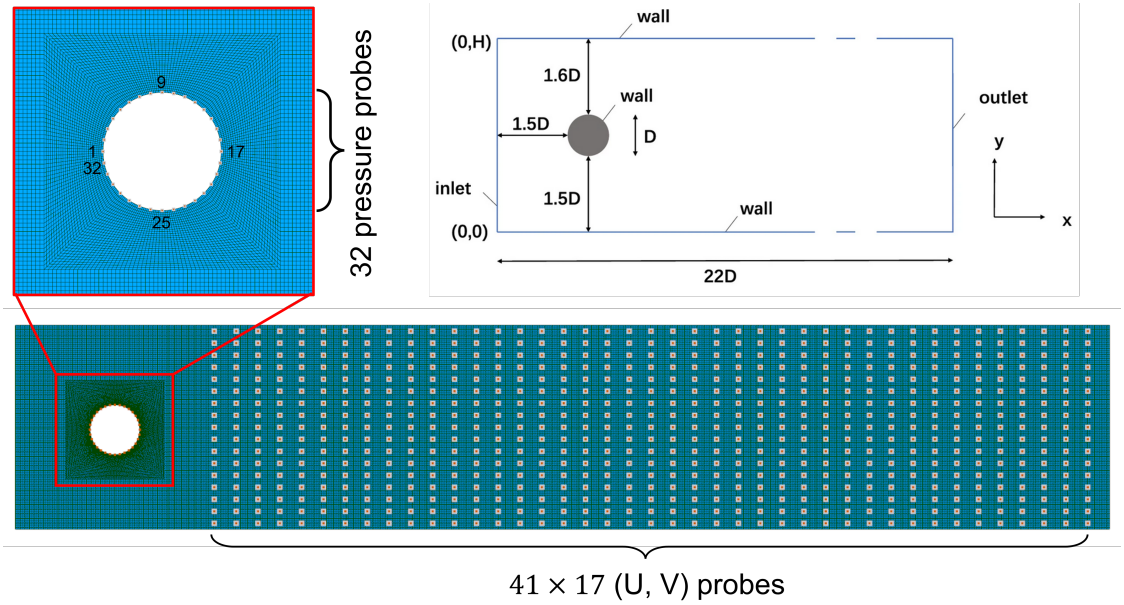


Figure 5.2: Flow around cylinder case geometry and data sampling points.

5.1.2 Results analysis

5.1.2.1 Training details

Each CNN model is trained for 100 epochs using Adam 2 as optimizer with default values $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ adopted in the training. Root mean square error (RMSE) as loss function:

$$RSME = \sqrt{\frac{1}{n}\sum_{i}^{n}(y_i - \widehat{y}_i)^2}, \tag{5.1}$$

where the y is the ground-truth value and $\widehat{y}$ is the CNN prediction.

The coefficient of determination R^2 which measures how well a regression fits the observed data is used to evaluate the CNN model prediction accuracy:

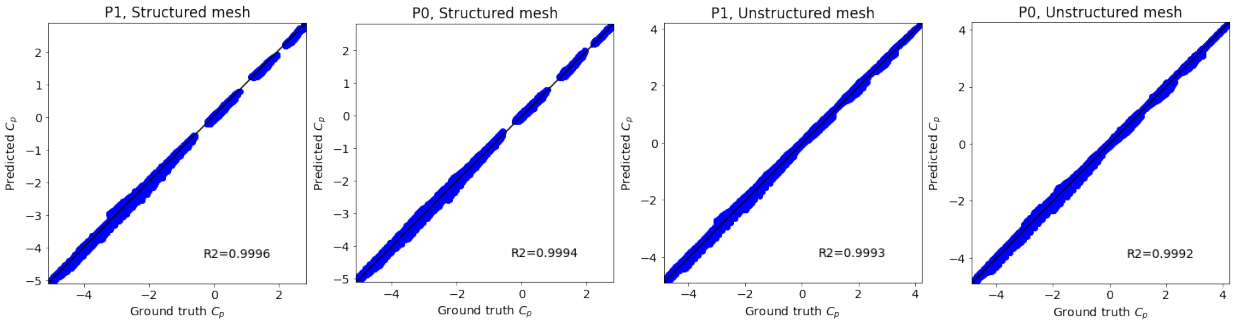
$$R^2 = 1 - \frac{\sum_i (y_i - \widehat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2}, \tag{5.2}$$

where y and $\bar{y}$ are the observed values and its mean, respectively, and $\widehat{y}$ is the predicted value.

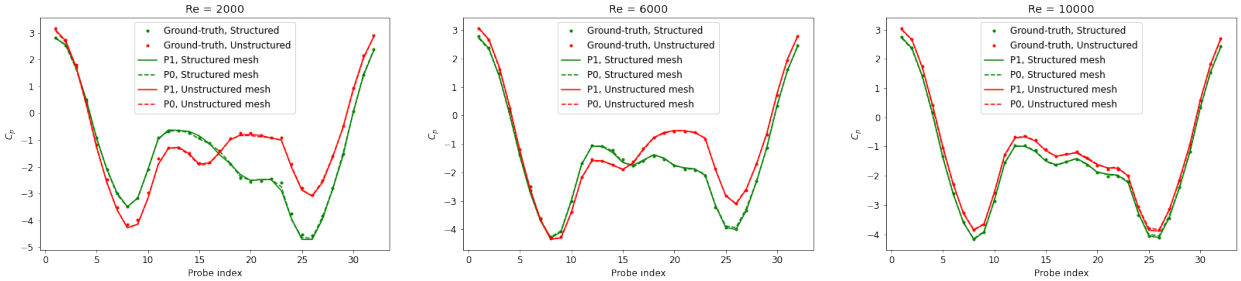
5.1.2.2 Data interpolation and mesh types

CNN algorithms are suitable for structured data or Cartesian mesh in CFD, while applied to unstructured mesh data interpolation is inevitable to sample the required input. The

influence of data interpolation methods on the obtained CNN model over unstructured mesh should be investigated by being compared with structured mesh. The comparisons between the ground-truth label sampled from two types of meshes using P0/P1 interpolation and the corresponding CNN predictions are visualized in Fig. 5.3a, the mesh type and data interpolation method have no noticeable influence on the CNN prediction accuracy. The predicted pressure coefficient at one time step for flows of three Reynolds numbers is plotted Fig. 5.3b, 5.3c and 5.3d, respectively. The CNN prediction agrees well with the ground-truth on both the interpolation cases ($Re = 2000, 6000$) and the extrapolation case ($Re = 10000$).



(a) Comparison of CNN prediction trained on dataset sampled using P0/P1 interpolation on structured/unstructured meshes. (From left to right: P1 interpolation from coarse structured mesh; P0 interpolation from coarse structured mesh; P1 interpolation from unstructured mesh and P0 interpolation from unstructured mesh.)



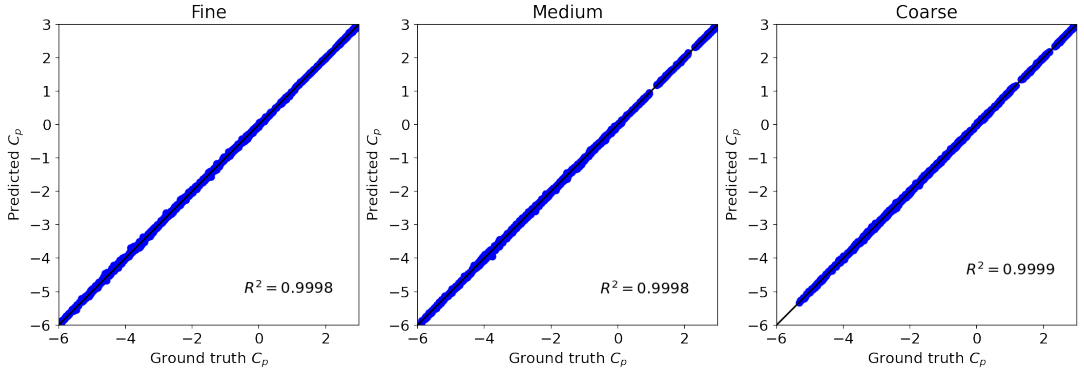
(b) $Re = 2000$ *(c) $Re = 6000$* *(d) $Re = 10000$*

Figure 5.3: Comparison of ground-truth with predictions of CNNs trained on dataset sampled using different interpolation methods on coarse structured mesh and unstructured mesh.

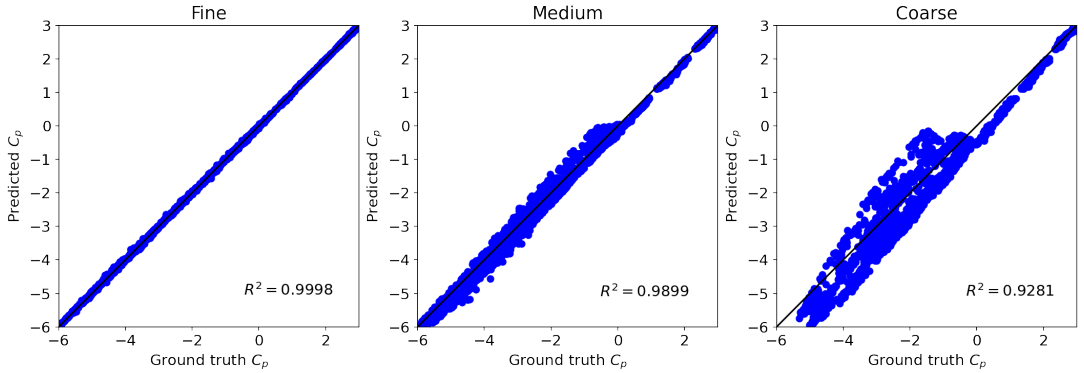
5.1.2.3 Mesh refinement

A trained model loses generality on the data of the mesh different from that used to form dataset even for the same flow case. The dataset is regenerated using structured meshes at three refinement levels which contain 16×9600, 4×9600 and 9600 cells for fine, medium and coarse meshes, respectively. As shown in Fig. 5.4a, the predictions of CNN models agree well with the ground-truth values of the dataset on which CNN model is trained. However, the prediction deteriorates significantly either when the CNN is trained on fine-mesh dataset and predicts on two coarser meshes, as shown in Fig. 5.4b, or when the CNN is trained on coarse-mesh dataset and predicts on the other two finer meshes, as shown in Fig. 5.4c. The predictions of CNNs trained on different datasets at one time step on flow case with $Re = 6000$ and medium mesh are shown in Fig. 5.5. A trained CNN model makes precise prediction on the same dataset over which it is trained

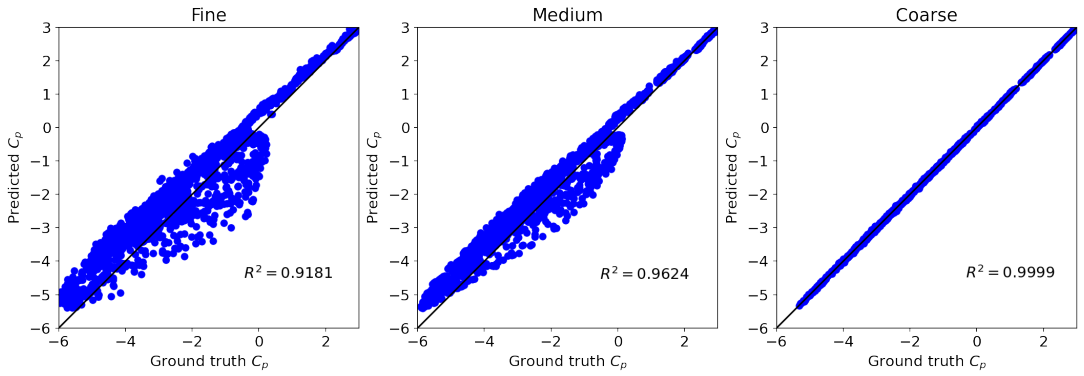
(Fig. 5.5b, however its prediction degrades significantly when the mesh encountered in the inference stage is either coarsened (Fig . 5.5a) or refined (Fig. 5.5c), especially in the vortex detaching region (probe index from 5 to 28). The prediction over medium-mesh dataset of the CNN trained on fine-mesh dataset correlates with the ground-truth label better than that of the CNN trained on coarse-mesh dataset, which can be ascribed to the fact that the CFD result using this coarse mesh is not mesh independent.



(a) CNN trained on the datasets of corresponding meshes.

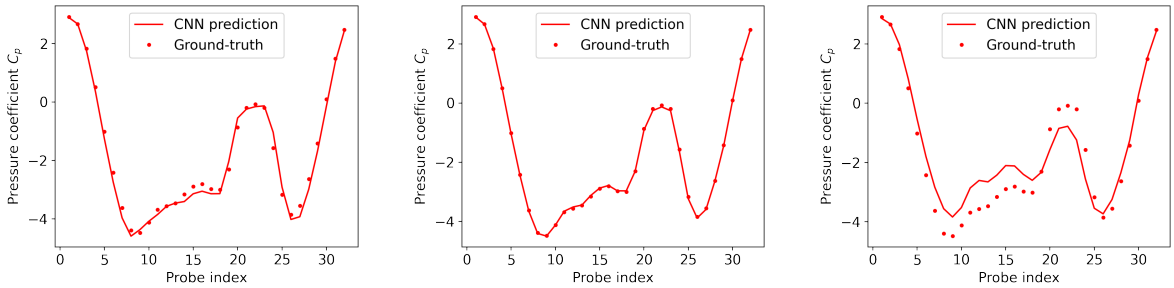


(b) CNN trained on fine-mesh dataset.



(c) CNN trained on coarse-mesh dataset.

Figure 5.4: Comparison of the ground-truth C_p with the prediction of CNN trained on different datasets. (From left to right: CNN predictions on fine, medium and coarse resolution meshes for flow cases with $Re = 6000$.)



(a) CNN is trained on fine-mesh dataset. *(b) CNN is trained on medium-mesh dataset.* *(c) CNN is trained on coarse-mesh dataset.*

Figure 5.5: Comparison of the ground-truth C_p of medium-mesh dataset with CNN predictions trained on datasets formed by different meshes for $Re = 6000$ case on one time step.

5.2 Identification of vortex shedding behind backward-facing step

In this section, how to detect vortex regions behind a 2D BFS using CNN model is described. The input for the CNN model, including CFD simulation and vortex labeling method, is introduced firstly. A series of potential input sets to characterise vortexes are selected and compared in terms of the identification performance of the corresponding trained CNN models. The general steps of flow phenomena identification in the CFD results by machine learning models is shown in Fig. 5.6.

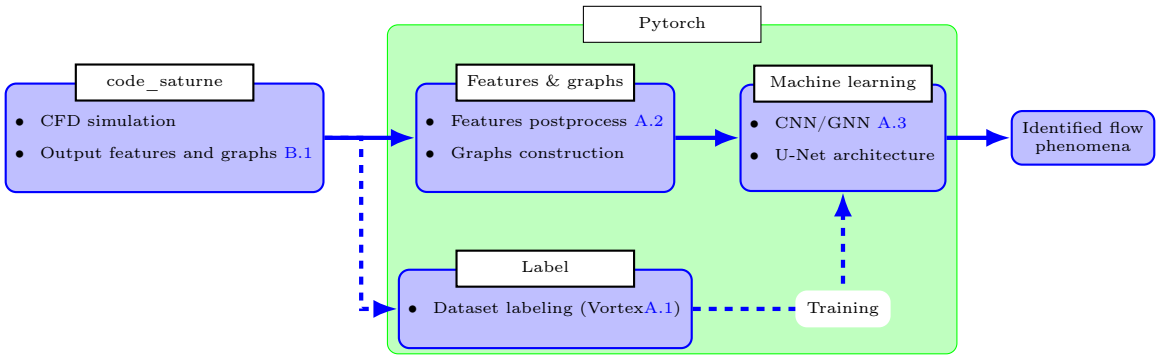


Figure 5.6: Flowchart of flow phenomena identification in the CFD results by machine learning models.

5.2.1 Dataset generation

In order to generate the dataset, the vortex shedding behind the 2D BFS Le et al. (1997) is simulated using transient code_saturne solver solving Navier-Stokes equations and and $R_{ij} - \epsilon$ SSG turbulence model. The flow configuration is shown in Fig. 5.7. The computational domain consists of an inlet section $L_i = 10h$ prior to the sudden expansion. The total length in streamwise direction is $L_x = 30h$ and the vertical height is $L_y = 6h$. The mean velocity with the turbulence intensity of 5% of the mean velocity magnitude is imposed at the inlet boundary. The Neumann condition is imposed at the outlet boundary. The scalable wall function is applied to the bottom wall. The upper boundary is set to symmetry boundary condition. The Reynolds number based on step height h and bulk velocity is $Re_h = 5100$. The total number of computational cells in x direction after the step is 240 and that in y direction is 220. Only one cell is used for z direction.

Ten flow-throughs were calculated before outputting the results in order to make sure the flow field is fully developed. Then the physical fields at 100 time steps with an interval of 0.1s were output and divided into three groups with the splitting ratio of 80/10/10 for training, validation and test cases, respectively.

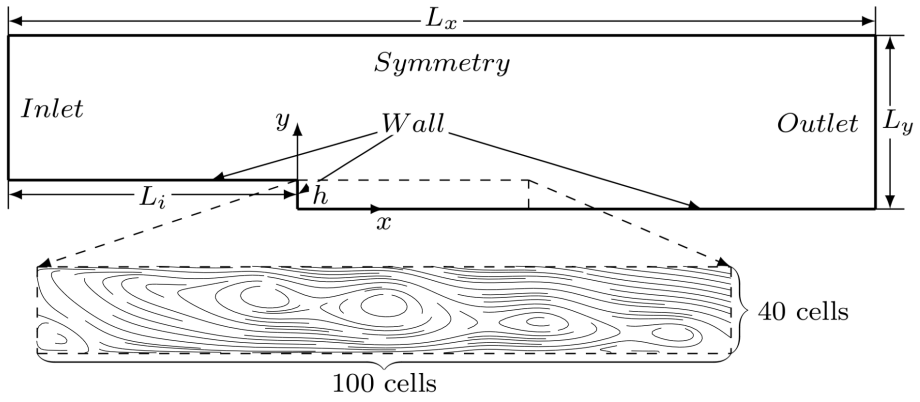


Figure 5.7: Backward-facing step flow configuration.

5.2.2 Vortex labeling

Since the supervised training of the neural networks is adopted in our study, a binary ground-truth label for each node indicating whether the node is in the vortex region or not should be provided. One of our contribution of our work is to the proposition of an auto-labeling method using Depth First Search (DFS) and Biased Random Walking (BRW) on the directed graph derived from the velocity field. DFS is an algorithm for traversing or searching tree and graph structured data and is employed for various applications, such as finding bridges or strongly connected components on the graph, maze generation. BRW algorithm explores the graph structure by going from one state/node to the next potential state/node with a probability and it is widely used for the search engine, to investigate the advertisement diffusion on the social networks, distribution of the animal community and etc. Here we apply these two algorithms to label the vortex region on the CFD meshes with velocity field which can be naturally viewed as a weighted directed graph. As shown in Fig. 5.8, the edge in the directed graph is the shared face of adjacent cells in the mesh whose direction is from upwind cell to downwind cell according to the velocity field.

As shown in algorithm 3, this vortex auto-labeling algorithm labels the vortex in 2D CFD results based on the closure of the streamline in two steps: a) locating the vortex core with DFS algorithm Jungnickel and Jungnickel (2005); b) enlarging the vortex region by BRW. The pseudo codes of DFS and BRW algorithms are shown in algorithm 4 and algorithm 5, respectively.

The vortex core is assumed to exist among a limited number, 4 for 2D structured mesh and 8 for 2D unstructured mesh, of connected cells. The DFS algorithm traverses all the nodes on the graph and recursively search the downstream nodes until it goes back to the beginning node within predefined steps. Once the vortex core is located, the BRW algorithm enlarges the vortex region starting from the vortex core. The probability of random walking from central node to a downstream node is proportional to the ratio of mass flux entering the corresponding downstream node from central node to the total

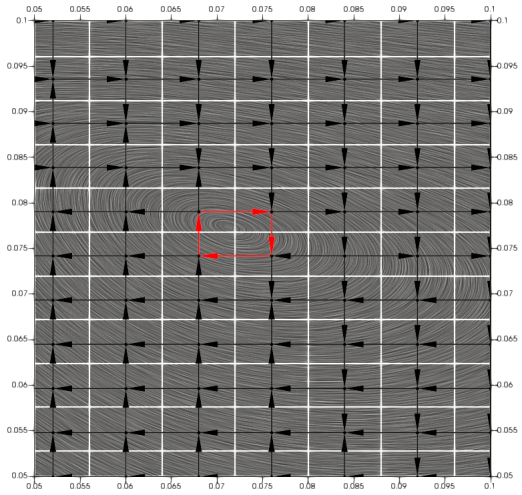


Figure 5.8: Directed graph superimposed on streamline background. The red arrows connect the vortex core cells. The white wire frame is the CFD mesh.

Algorithm 3: Vortex auto-labeling algorithm in 2D CFD cases

input : Nodes $\mathcal{V}$, Mass flux $\{M_{u,v} : u \in \mathcal{V}, v \in \mathcal{V}\}$, Neighborhood $\mathcal{N}(u) = \{v \in \mathcal{V} : M_{u,v} > 0\}$

output: Vortex nodes: $\mathcal{V}_{vortex}$

Build weighted directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with edge weight $e_{u,v} = \frac{M_{u,v}}{\sum_{v \in \mathcal{N}(u)} M_{u,v}}$;
initialize vortex core set $\mathcal{V}_{core}$;
for *each* $u \in \mathcal{V}$ **do**
    add `DFS` $(\mathcal{G}, v, 0)$ to $\mathcal{V}_{core}$;
end
create working graph $\mathcal{G}_w \leftarrow \mathcal{G}$;
while $\mathcal{G}_w$ *changes* **do**
    for *icore in* $\mathcal{V}_{core}$ **do**
        randomly select a node u from *icore*;
        find the loop enclosing u: $L_u \leftarrow$`BRW` $(\mathcal{G}_w, u)$;
        remove the nodes enclosed by L_u from $\mathcal{G}_w$;
    end
end
return $\mathcal{V}_{vortex} = \mathcal{G}.nodes() - \mathcal{G}_w.nodes()$;

mass flux exiting the central node. One walking path either ends at the beginning node leading to the success of enlarging the vortex region or ends at the outlet nodes leading to the failure of enlarging vortex region. To make sure the final labeled vortexes are separate from each another, a single enlarged vortex region should not contain multiple vortex cores. The vortex location labeling process is terminated when the searched vortex boundary is not changed from the last random walking.

The ground-truth labels superimposed on the streamline plot at one time instant obtained using this method from a structured mesh of 74400 nodes and an unstructured mesh of 131118 nodes for 2D BFS case are shown in Fig. 5.9. All the cells in vortex zone are represented by the points over the streamline background. It should be noted that although the vortex boundary labeled in this method does not precisely reflect the

Algorithm 4: Depth first search algorithm

input : Graph $\mathcal{G}$, starting node v, depth count d
output: Vortex core nodes

```
d++1;
for w in G.adj[v] do
    if w is unvisited and d < depth then /* Set depth to 4 for structured
     mesh, 8 for unstructured mesh.                                    */
        if DFS (G, w, d) then
            return list(w, DFS (G, w, d));
        end
    else if w is visited then
        return w;
    else
        return None
    end
    ;
end
```

Algorithm 5: Biased random walking

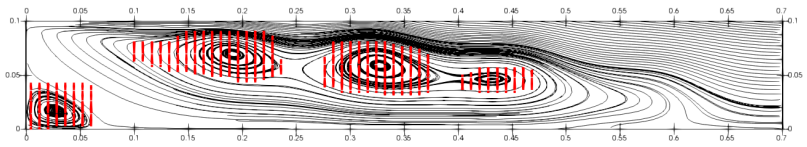
input : Graph $\mathcal{G}$, Starting point u
output: Nodes on the loop $\mathbf{V}_{path}$ enclosing u

```
while step < threshold do /* Threshold depends on mesh size.          */
    add u to path;
    select next node v in G.adj[u] with possibility P(v) ∝ e_{u,v};
    if v in V_core then
        return V_path;
    end
    walk to next v : u ← v;
    step++1;
end
```

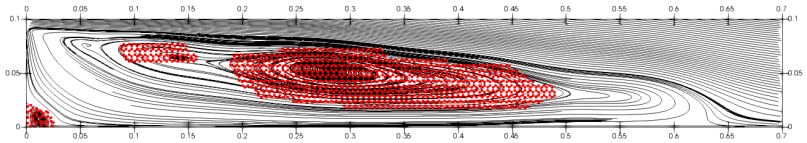
real vortex shape leading to a noisy dataset, it has the merit of auto-labeling and can label vortex location for massive snapshots. And we expect that the machine learning algorithm outperforms the ground-truth label on identifying the vortex position and shape after trained on this noisy dataset. The auto-labeling algorithms takes 292s and 122s on average to label vortexes at one time step on the structured mesh and on the unstructured mesh, respectively.

5.2.3 Input feature selection

To find out the optimal vortex indicators as the input for, five features, summarized in Table 5.2, were selected such as velocity field, Q-criterion, turbulence intensity, deviation from shear flow and pressure gradient along streamline. The latter four features are visualised in Fig. 5.10 along with ground-truth label and streamline. Intuitively, the closure of the streamline is the intrinsic feature and has sufficient information to characterize the presence of vortex. However, the deterministic methods which derive local value from



(a) Structured mesh.



(b) Unstructured mesh.

Figure 5.9: Labeled vortex region in 2D BFS case meshes superimposed on streamline background.

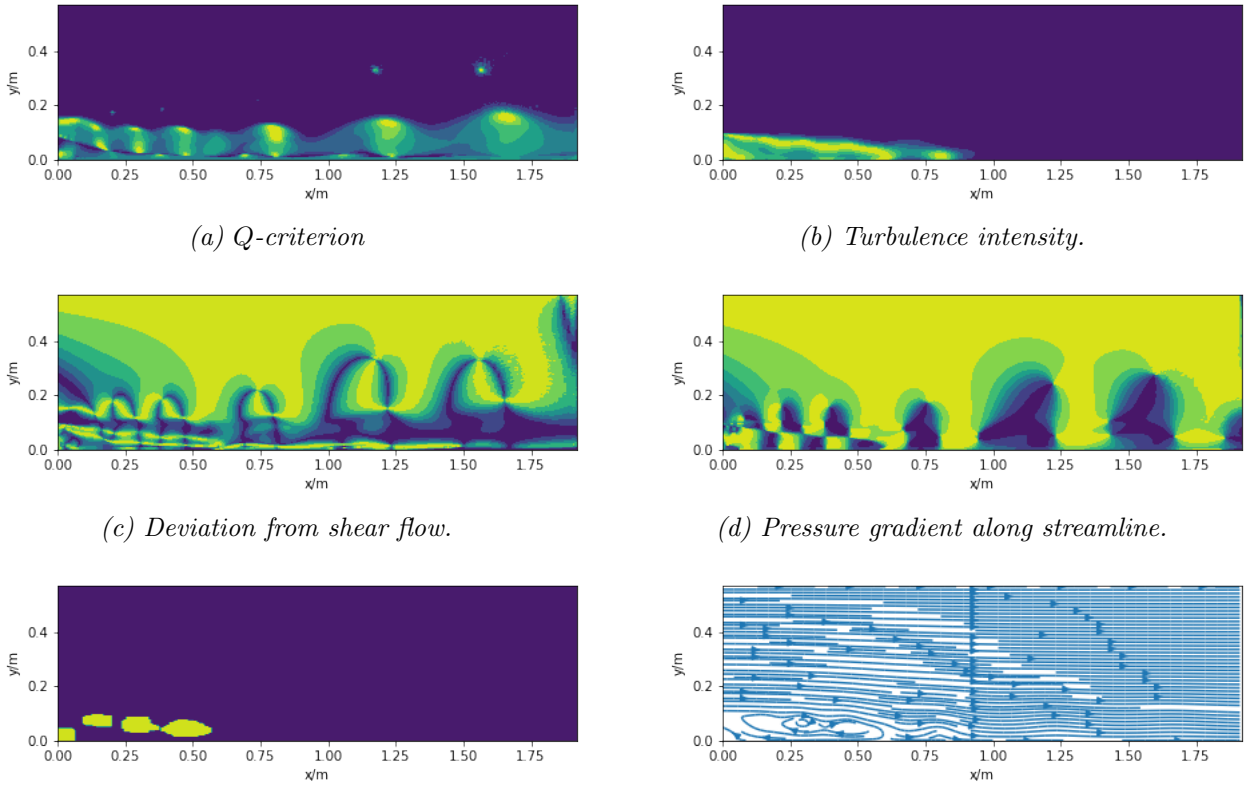
the velocity field and its gradient field fail to capture the global view of the streamline closure, such as the widely accepted vortex indicator Q-criterion $Q = \|\mathbf{R}\|^2 - \|\mathbf{S}\|^2$, where $\mathbf{R} = \frac{1}{2}(\nabla\mathbf{U} - (\nabla\mathbf{U})^T)$ and $\mathbf{S} = \frac{1}{2}(\nabla\mathbf{U} + (\nabla\mathbf{U})^T)$. It can be seen from Fig. 5.10a that Q-criterion has false high value in both the near wall region and the far-wake region in $x > 1.0m$. In the vortex region, the turbulence level is generally higher than in the pure shear flow region which makes the turbulence intensity a natural choice as vortex indicator. The latter two variables are both related to the vortex physics. The deviation from shear flow characterizes the streamline curvature. The pressure at the vortex center is lower than the peripheral area which causes two perpendicular characterising lines in the pressure-gradient-along-streamline field located at the vortex center as shown in Fig. 5.10d. These features are normalized following the practice in Ling and Templeton (2015): $\widehat{a} = \frac{\|\alpha\|}{\|\alpha\|+\|\beta\|}$, where α is the original feature, β is the normalization factor.

Table 5.2: Non-dimensional input features characterizing vortex as input for CNNs. The features are normalized following the practice in Ling and Templeton (2015): $\widehat{a} = \frac{\|\alpha\|}{\|\alpha\|+\|\beta\|}$

Feature	α	β
Normalized velocity	$\mathbf{U}$	0
Q-criterion	$\frac{1}{2}(\|\mathbf{R}\|^2 - \|\mathbf{S}\|^2)$	$\|\mathbf{S}\|^2$
Turbulence intensity	k	$0.5U_iU_i + k$
Pressure gradient along streamline	$U_k\frac{dP}{dx_k}$	$\sqrt{\frac{dP}{dx_j}\frac{dP}{dx_j}U_iU_i}$
Deviation from parallel shear flow	$\lvert U_kU_l\frac{dU_k}{dx_l}\rvert$	$\sqrt{U_nU_nU_i\frac{dU_i}{dx_j}U_m\frac{dU_m}{dx_j}}$

Five combinations of the above mentioned variables, as shown in Table 5.3, are tested. The original velocity is used as the baseline input set #1. The normalized velocity fields is used as input set #2 to show investigate the normalization influence. The Q-criterion is used as single input for input set #3. The last three variables in Table 5.2 form the input set #4 since each one of them characterizes vortex by a certain spatial pattern, not by a threshold, The combination of input sets #1 and #4 form the last input set.

A CNN model with a 4-level and 8-layer U-Net architecture containing 55633 trainable parameters is built to test different input sets. The channel dimensions are shown in Fig. 5.11. The kernel size is 3×3 for all convolutional layer. Each input set is tested five times by training the CNN model for 100 epochs with the parameters initialized using different random seed each time. The mean and standard deviation of the training loss and



(a) Q-criterion

(b) Turbulence intensity.

(c) Deviation from shear flow.

(d) Pressure gradient along streamline.

(e) Ground-truth label.

(f) Streamline.

Figure 5.10: The contour plot of four input features $\widehat{a}$, ground-truth label and streamline of the BFS test case at time=12.65s.

Table 5.3: Five input sets of vortex indicators.

Input set	Features
1	U, V
2	U_{norm}, V_{norm}
3	Q-criterion
4	Turbulence intensity Deviation from shear flow Pressure gradient along streamline
5	U, V Turbulence intensity Deviation from shear flow Pressure gradient along streamline

classification evaluation metrics are calculated. Adam optimizer with the initial learning rate of 10^{-3} and the weight decay rate of 5×10^{-4} was used. The binary cross entropy (BCE) was used as the loss function:

$$BCE = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\widehat{y}_i) + (1 - y_i)\log(1 - \widehat{y}_i), \tag{5.3}$$

where N is the number of nodes, y_i is the label for node i, 1 for that in vortex region and 0 for that in non-vortex region, and $\widehat{y}_i$ is the predicted category value by the model at

node i. The learning rate and weight decay rate are set to 10^{-3} and 10^{-5}, respectively. The batch size is set to 5.

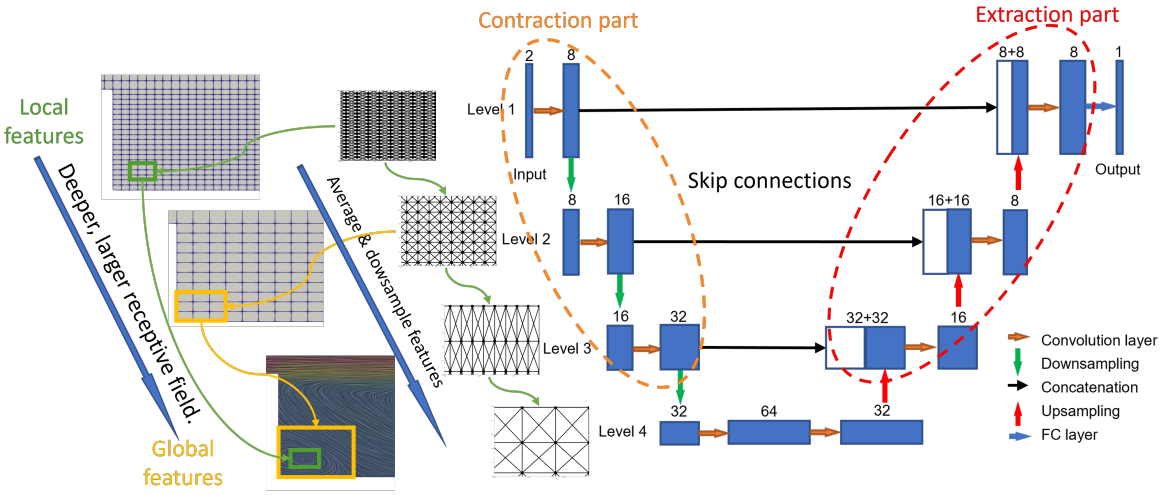


Figure 5.11: The 4-level 8-layer U-Net architecture.

5.2.4 Results analysis

As indicated by the training loss histories in Fig. 5.12, all the trainings proceed smoothly. The input sets #1 #2 and #5 have losses very close to each other and are noticeably lower than the other two input sets. The input set #3, Q-criterion, has the highest standard deviation and is the only scenario where the validation loss is higher than training loss.

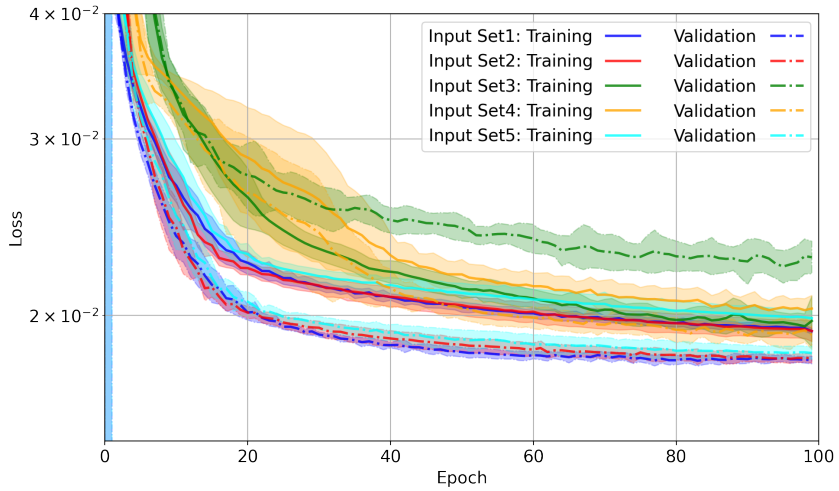


Figure 5.12: Training loss history CNNs with five input sets. The curve and shaded region represent the mean and standard deviation of five trainings, respectively.

The output value of CNNs ranges from 0 to 1 indicating the probability whether a point is in the vortex region. A threshold should be selected to separate non-vortex region from vortex region. A good model should give consistent predictions within a large threshold range. A receiver operating characteristic (ROC) curve Fawcett (2006) is a graphical plot measures the consistency of the diagnostic ability of a binary classifier

as the discrimination threshold varies. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The ideal prediction model should yield a point in the upper left corner or coordinate (0,1) in the ROC space, representing no false negatives and no false positives. As shown in Fig. 5.13, the ROC curves of all input sets are very close and input set #2 is slightly better than the others. As shown in Fig. 5.14, the shapes of the identified vortexes are very similar among all input sets except the input set #3 which is the worst.

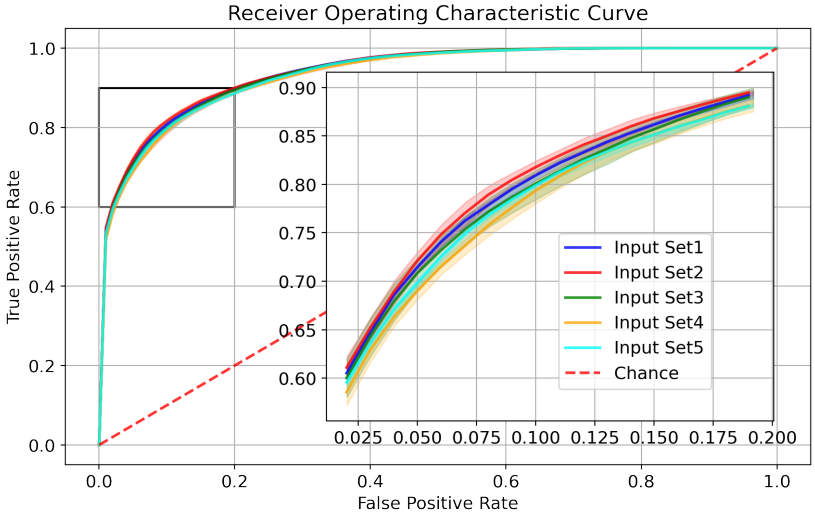


Figure 5.13: Receiver operating characteristic curves of CNN-Unet with different input sets. (The curve and shaded region represent the mean and standard deviation of five trainings, respectively.)

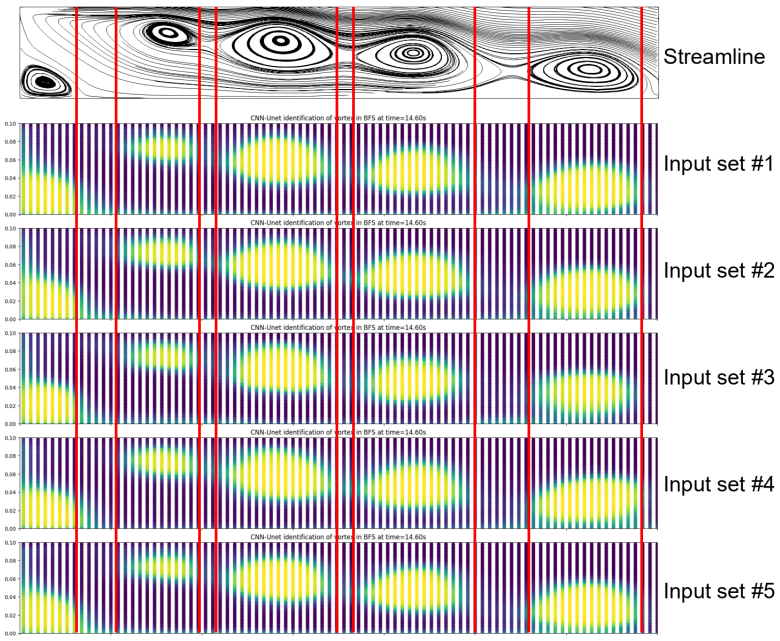


Figure 5.14: Vortexes at time=14.60s identified by CNN-Unet algorithm with different input sets. (From top to bottom: streamline, input sets from #1 to #5.)

Since the dataset is a biased one where only a small portion of points belong to vortex region, therefore the following four classification metrics are used to evaluate the classification performance: accuracy, precision, recall and F1 score:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{5.4}$$

$$Precision = \frac{TP}{TP + FP}, \tag{5.5}$$

$$Recall = \frac{TP}{TP + FN}, \tag{5.6}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}, \tag{5.7}$$

where TP, TN, FP, FN are the numbers of true positives, true negatives, false positives and false negatives, respectively. The performances of the CNNs trained on five input sets are evaluated in the vortex region behind the step $[[0, 0.8m], [0, 0.14m]]$ and summarized in Table 5.4. The highest and lowest values of each of four classification evaluation metrics among all input sets are labeled in green and red color, respectively. The input sets #1 and #2 have close classification performance and are generally better than the other input sets reflected by the fact that they have no worst mean values for four metrics. The normalization of the velocity field significantly accelerates the training. The input sets #3 has the highest standard deviations and the longest training time which suggests that the CNN model struggles to find the correlation between Q-criterion and the ground-truth label. Compared with input set #1, additional features in input set #5 do not bring performance improvement but longer training time.

Table 5.4: Classification performance of CNNs trained on five different input sets. (Green: best value; Red: worst value.)

Input set	Accuracy	Precision	Recall	F1 score	Time/epoch
1	88.03±0.16	90.86±0.48	61.73±0.81	73.51±0.50	2.63±0.79s
2	88.02±0.27	91.82±0.49	60.92±0.94	73.24±0.74	1.85±0.13s
3	87.83±1.10	89.38±3.97	62.74±7.78	73.22±4.32	4.43±0.34s
4	87.53±0.31	90.42±0.86	60.03±1.23	72.15±0.89	2.42±0.27s
5	87.64±0.12	91.44±0.49	59.68±0.61	72.22±0.37	3.32±1.63s

5.3 Thermal stratification region identification

5.3.1 Dataset generation

To form the training dataset, the transient mixed thermal convection process inside a 2D square cavity is simulated whose geometry and dimensions are illustrated in Fig. 5.15. The cavity has the width and height of H. The inlet and outlet are located at the upper and lower corners of the right side wall, respectively and their width are both $0.1H$. The cavity is initially filled with static fluid at temperature T_c. A jet at velocity of u_0 and temperature T_h enters the domain through the inlet and pushes the initial colder fluid out of the cavity through the outlet. To enlarge the dataset, nine cases including three Richardson numbers $Ri = [1, 10, 760]$ and three inclination angles $\alpha = [0, 10°, 30°]$, are

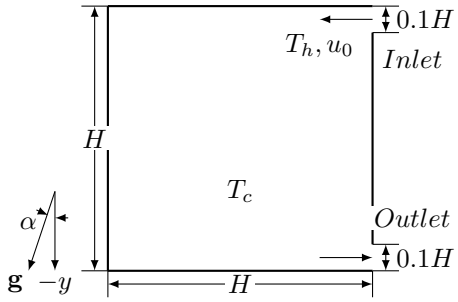


Figure 5.15: Schematic of transient mixed convection in a 2D square cavity.

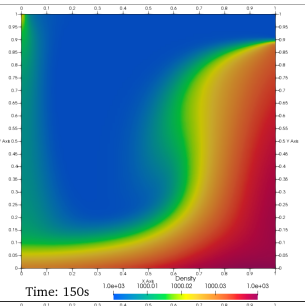
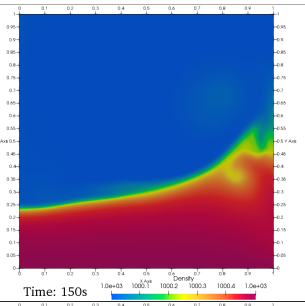
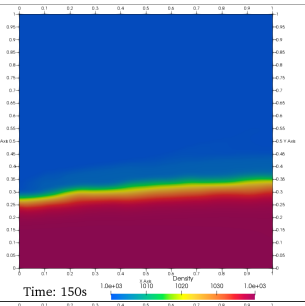
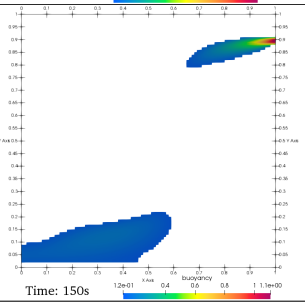
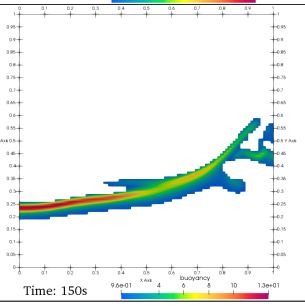
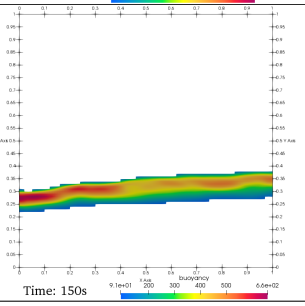
simulated. Richardson number is the measurement of the ratio of natural convection to forced convection:

$$Ri = \frac{g\beta H(T_h - T_c)}{u_0^2}, \tag{5.8}$$

where β is the coefficient of thermal expansion of the material. The inclination angle α represents the angle between gravity direction and $-y$ direction. All fluid properties are constant except density which is linearly dependent on the temperature:

$$\rho = \rho_0 - \beta\rho_0(T - T_c). \tag{5.9}$$

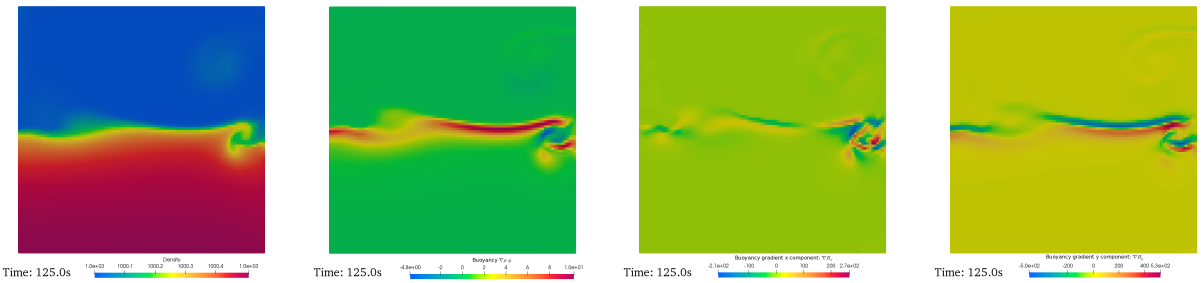
Table 5.5: Density and labeled thermal stratification region plots for cases with $Ri = [1, 10, 760]$ and $\alpha = 0$ at the last time step.

	$Ri = 1$	$Ri = 10$	$Ri = 760$
Density plot			
Label plot			

A duration of 150s is simulated with the time step of 0.1s from which 30 time steps are sampled to form the dataset. The buoyancy term $B = \nabla\rho \cdot \mathbf{g}$ is calculated to label the thermal stratification region. Due to the simplicity of the geometry and the flow structure, the cells of the highest 10% buoyancy value are selected as thermal stratification region for cases with $Ri = [10, 760]$ where the stratification is maintained due to larger buoyancy force compared with inertial force, while no stable thermal stratification region selected for case with $Ri = 1$ since inertial force becomes dominant. The density plots for cases with $Ri = [1, 10, 760]$ and $\alpha = 0$ at the last time step and their labeled thermal stratification regions are shown in Table 5.5.

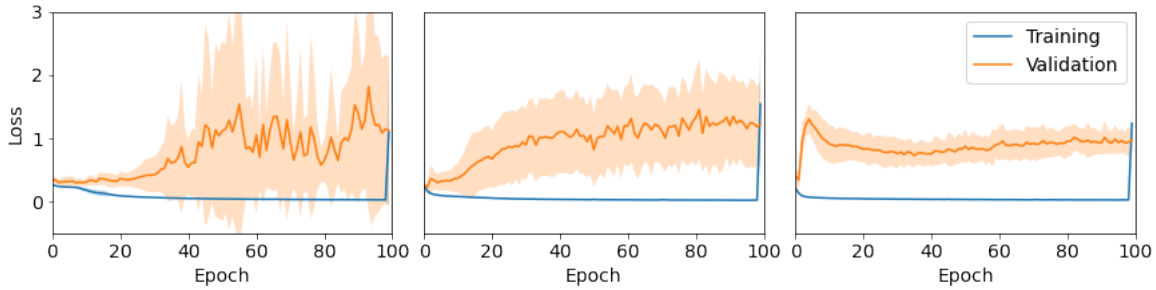
5.3.2 Input feature selection and architecture design

Three input sets are tested on the CNN architecture of the previous section: 1) density ρ, 2) buoyancy: $B = \nabla\rho \cdot g$ and 3) buoyancy gradient vector: $(\nabla B_x, \nabla B_y)$. These these input sets for case with $Ri = 10$ and inclination degree $\alpha = 0$ at one time step are visualised in Fig. 5.16. Intuitively, these features well characterise the boundary between hot fluid and cold fluid.

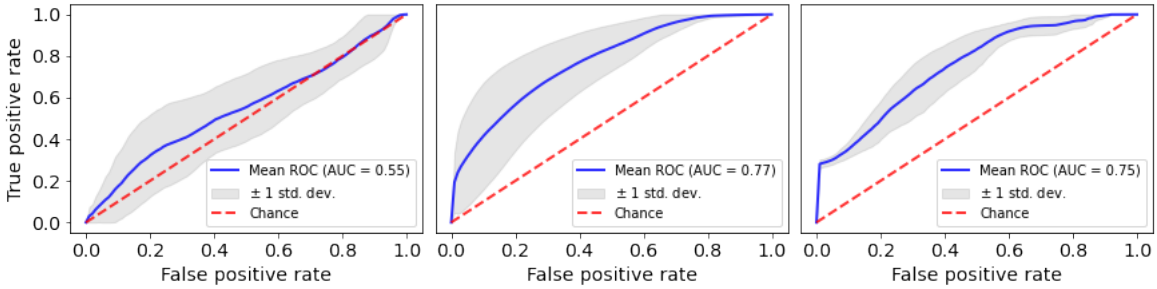


(a) ρ. *(b) B.* *(c) ∇B_x.* *(d) ∇B_y.*

Figure 5.16: Features to characterise thermal stratification region for case with $Ri = 10$ and inclination degree $\alpha = 0$ at time $t = 125s$.



(a) Loss history plots.



(b) ROC plots.

Figure 5.17: Loss history and ROC plots with different inputs and the previous architecture. (From left to right: input #1, input #2 and input #3.)

However, as shown in Fig. 5.17, training CNN models with these features as input cannot converge. Take the case of $Ri = 1$ and inclination angle $\alpha = 30°$ for example, as shown in Fig. 5.18, CNN model may extract the same hidden features at thermal stratification position A and non thermal stratification position B. What makes these two positions intrinsically differently from each other is the angle between the gravity direction and the orientation of the thermal stratification layer, the orientation of the thermal stratification layer is perpendicular to the gravity direction at position A, while

these two directions are aligned at position B. Thus, as shown in Fig. 5.19, in order to inform the CNN model the gravity direction, a second input layer was introduced before the final output layer with gravity direction as input.

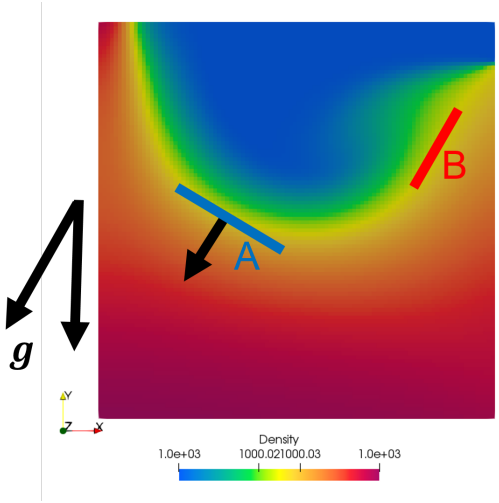


Figure 5.18: The angle between gravity direction and the orientation of thermal stratification layer distinguishes position A from B.

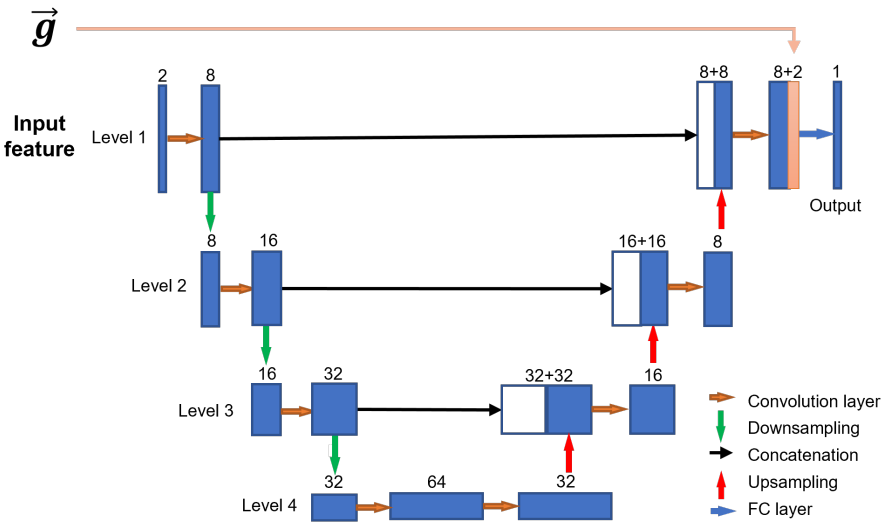
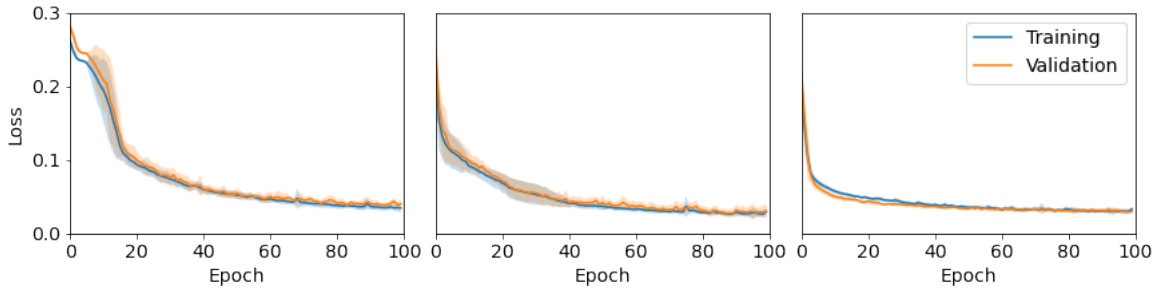


Figure 5.19: Architecture used to identify thermal stratification region.

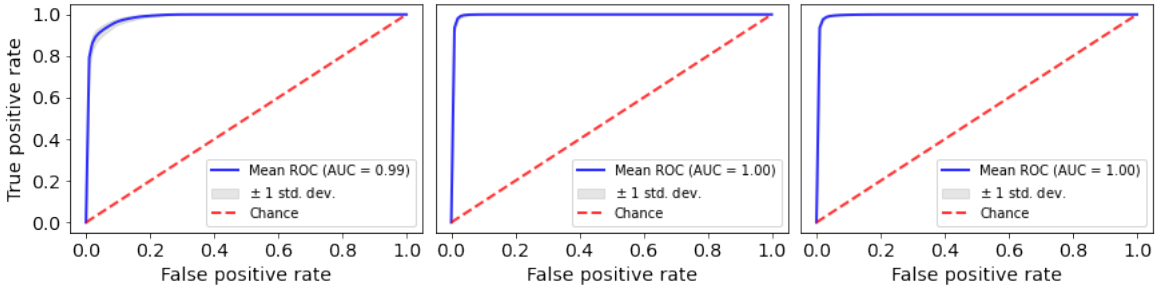
The training histories and ROC plots of CNN models identifying thermal stratification region with the above-mentioned three input sets are plotted in Fig. 5.20 and the corresponding identification performances are summarized in Table 5.6. Among three input sets, no specific one posses significant advantage over the other two in terms of four classification evaluation criteria.

5.3.3 Generalization on unseen case

The trained CNN model with $(\nabla B_x, \nabla B_y)$ as input is tested in an unseen case - 2D thermal stratification in T-junction, one of the code_saturne validation cases. As shown in Fig. 5.21, the hot water enters the pipe from the right inlet while the cold water enters



(a) Loss history plots.



(b) ROC plots.

Figure 5.20: Loss history and ROC plots with different inputs and the architecture with a gravity input layer. (From left to right: input #1, input #2 and input #3.)

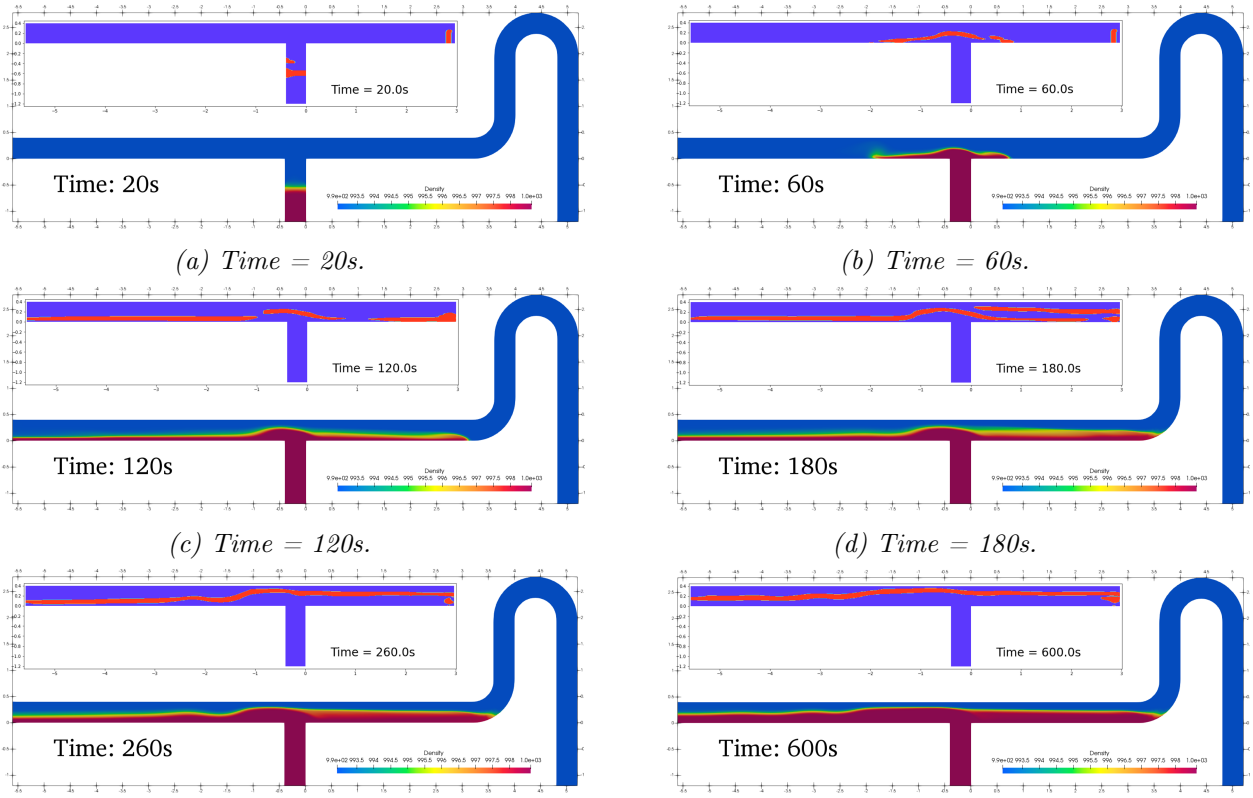
Table 5.6: CNN identification performance of thermal stratification region with different input sets. (The best values are labeled in red)

Input	Accuracy	Precision	Recall	F1 score
ρ	98.83±0.12	88.06±2.45	90.73±1.42	89.34±0.90
B	98.93±0.09	92.61±2.58	89.43±4.03	90.87±0.99
$(\nabla B_x, \nabla B_y)$	98.66±0.25	93.72±2.98	77.97±7.23	84.82±3.67

through the middle inlet. Temperature difference of two inlets is $20.24°C$. Hot water and colder meet at the horizontal part of the pipe and a stable thermal stratified region forms. Even though a structured mesh is used, the CFD result is interpolated into a uniformly distributed array of points before being fed to CNN model due to the complex geometry. CNN successfully identified the interface between cold and hot water for most time steps as suggested in Fig. 5.21, however it fails to capture the mixing process when the cold water first encounters the hot water at the T-junction region as shown in Fig. 5.22. The inaccuracy of CNN model can be partly ascribed to the small size dataset and incompleteness of the input feature. More variables other than buoyancy term, such as the velocity field, should be considered.

5.4 Summary

In this chapter, possible factors that might influence the performance of the trained CNN algorithms, such as data interpolation method, mesh type and mesh refinement, are experimentally investigated. As the suggested by the CNN regression of the pressure around the cylinder based on the velocity field in the wake region, data interpolation method (P0/P1) and mesh type have negligible influence on the CNN model accuracy. A more concerning problem for data-driven algorithms is its generality on the dataset different

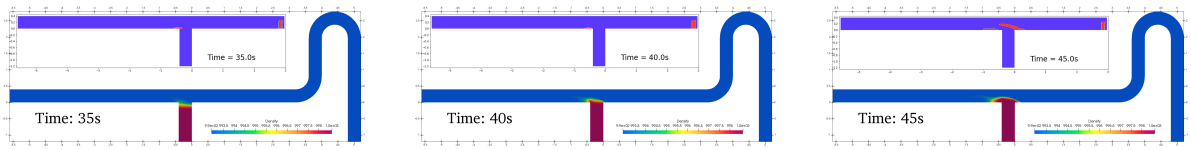


(a) Time = 20s. *(b) Time = 60s.*

(c) Time = 120s. *(d) Time = 180s.*

(e) Time = 260s. *(f) Time = 600s.*

Figure 5.21: Comparison between density plots and thermal stratification region identified by CNN (upper left panel) at different time steps.



(a) Time = 35s. *(b) Time = 40s.* *(c) Time = 45s.*

Figure 5.22: CNN fails to capture the thermal stratification region when cold water enters the horizontal pipe.

from the one over which it is trained. The learned feature pattern by CNN fails to generalize when the mesh is either refined or coarsened. This is a key difference of using CNN in CFD domain than in computer vision domain where more options in the CFD simulation process introduce much larger variance in the dataset, thus causing that most of the data-driven algorithms are scenario-specific or dataset specific.

In the second section, the possibility of using CNN to detect independent vortex behind a 2D backward-facing step is explored. A novel vortex labeling method for 2D mesh based on depth first search algorithm and random walking algorithm on directed graph is introduced. Experiment shows that CNN is able to identify separated vortexes from the velocity vector without additional postprocess of the CFD result thanks to its ability to extract non-local spatial features which may have better generality on cases of different geometries.

The feasibility of CNN identifying other flow phenomenon such as thermal stratification is also demonstrated at the end.

6 GNN identification of flow phenomena on unstructured meshes

Contents

In the previous chapter, the flow phenomena identification was conducted on cartesian meshes which has huge limitations when applied to industrial cases where unstructured meshes are normally used to fit the curved surfaces. Therefore the pursuit of a methodology suitable for unstructured meshes is necessary. Graph neural networks can be a promising candidate. Unlike the uniqueness of the convolution kernel in CNNs, there are various graph convolution kernels and their performance should be evaluated firstly. A proper convolution suitable for CFD meshes kernel should be searched or proposed. A good architecture for GNN model should be evaluated as well in terms of both computational efficiency and identification accuracy. This chapter will address aforementioned problems by firstly introducing the proposed Fast-GMM kernel. The superiority of the U-Net architecture with graph hierarchy generated from AMG as input is demonstrated by comparing against other two architectures, skip-connection and sequential-connection architectures. The generality of the proposed approach to different mesh topologies, turbulence models at different Reynolds numbers and the kernel differences are experimentally demonstrated. The feasibility of the proposed framework with proposed kernel identifying 3D vortex is showcased at the end.

6.1 Proposed Fast-GMM kernel

CNNs are computationally efficient in terms of training time per epoch and training epochs to achieve convergence due to the structured representation of data. On the contrary, equivalent GNNs need much more training time per epoch and more epochs before giving meaningful prediction due to unstructured representation of data, all connections and relative positions of connected nodes must be explicitly stored. During our previous tests, we found that GNNs with the GMM kernel require considerable training time per epoch compared to those with GCN kernels, almost the simplest kernel of GNNs, which further worsens the efficiency of GNN training. Training efficiency becomes an important concern since we intend to detect flow phenomena on industrial 3D unstructured meshes in the future. Its training inefficiency comes from the fact that two variables should be trained simultaneously, covariance matrix $\sum_k$ and mean direction vector $\mu_{\mathbf{k}}$. In addition, if a learned GMM kernel with all the directions $\mu_{\mathbf{k}}$ having the same distribution as the CNN kernel, shown in Fig. 6.1a, and the covariance matrix $\sum_k$ being $\mathbf{1}$ is used to distinguish the directions of a edge distribution as shown in Fig. 6.1b, the obtained e_{ij}-μ_k alignment matrix as defined in 3.23 has a distribution as shown in Fig. 6.1d. It can be seen that the alignment matrix obtained by GMM kernel is more diffuse than that obtained by CNN kernel, which is binary, 1 for most well aligned pair and 0 for not well aligned pairs.

In order to make the training more efficient and improve the resolution of detecting edge directions, several simplifications are made to the original GMM kernel by using analogies to the traditional CNN kernel. In the traditional CNN kernel, the relative directions of the neighboring weight to the central weight are fixed. Take the 3×3 kernel for example, the central weight has eight neighboring weights which are fixed at eight directions evenly distributed around the central weight. Each trainable weight is uniquely associated to the neighboring pixel on that corresponding direction. The distance between the central pixel and the neighboring pixel is not learned by the kernel since in the image all the pixels are equidistant. Therefore, the relative directions of the neighboring cells, instead of their distances, from the central cell are important in capturing some spatial patterns residing on the mesh. What is more, the distance between nodes on the CFD mesh is normalized, which will be introduced in Section 6.2, to avoid poor generality brought by the cell size variation from one case to another. Thus, the trainable covariance matrix $\sum_k$ in the original GMM kernel is removed and the trainable mean vectors are replaced by predefined directions. The square of the min-max scaled alignment is calculated to make the most aligned $e_{ij} - \mu_k$ pair distinguished from the other non-aligned pairs. Thus, the alignment between the edges and the directions in predefined template reads:

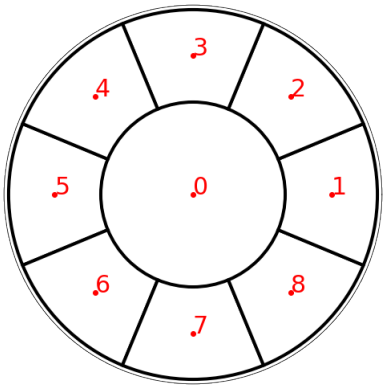
$$g_k(\mathbf{e}_{ij}) = \left| \left(\exp\left(-\frac{1}{2} (\mathbf{e}_{ij} - \mu_k)^T (\mathbf{e}_{ij} - \mu_k) \right) \right) \right|^2_{MinMax}, \tag{6.1}$$

where min-max scaling is defined as:

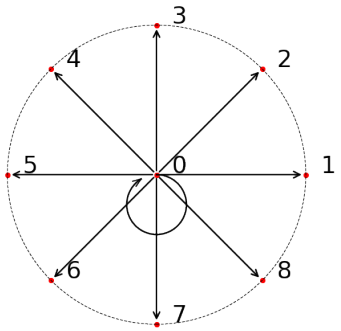
$$\|x\|_{MinMax} = \frac{x - x_{min}}{x_{max} - x_{min}}. \tag{6.2}$$

The proposed kernel is termed as Fast Gaussian Mixture Model (Fast-GMM). We keep one direction in the template for the center node and evenly divide 2π into $k-1$ directions for the neighboring nodes. Since the following trainings done on graphs derived from 2D structured mesh where each center cell has eight neighboring cells, as shown in 6.1a, we

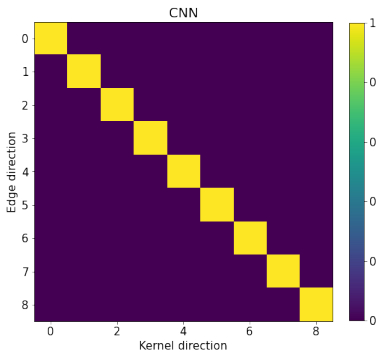
set the hyperparameter $k = 9$ which is analogous to the 3×3 convolution kernel in CNNs. As shown in Fig. 6.1e, the alignment matrix obtained by Fast-GMM is closer to that of CNN kernel compared with the original GMM. This characteristic enables the Fast-GMM kernel to have a higher resolution on discerning the edge directions and thus leading to identify sharper vortex boundary compared to the original GMM which will be shown in the following.



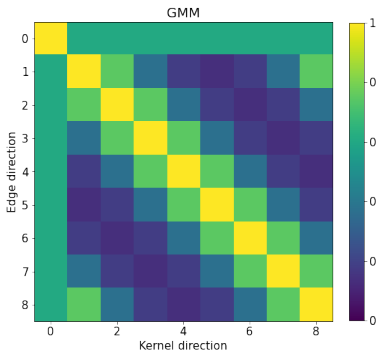
(a) Kernel directions



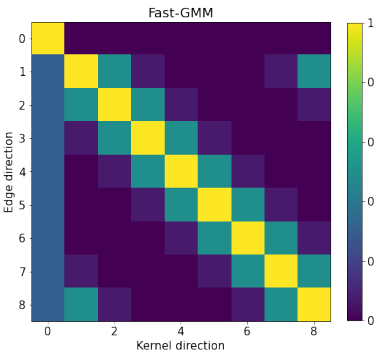
(b) Edge directions in graph.



(c) Alignment matrix obtained by CNN kernel.



(d) Alignment matrix obtained by GMM kernel.



(e) Alignment matrix obtained by Fast-GMM kernel.

Figure 6.1: Alignment between edges directions and trainable directions obtained by different kernels.

6.2 Framework of graph neural networks

6.2.1 Graph construction

The graphs fed to the GNNs are derived from the CFD meshes where the cells and the vertices shared by adjacent cells in CFD meshes become the nodes and the edges in graphs. The derived graph is a dual mesh of the CFD mesh. The input data are stored on the nodes. To regularize the graph, the normalized direction $\mathbf{e}_{ij}$ pointing from center node i to neighbor node j is stored on the edge connecting the two nodes:

$$\mathbf{e}_{ij} = \frac{\mathbf{x}_j - \mathbf{x}_i}{||\mathbf{x}_j - \mathbf{x}_i||} \tag{6.3}$$

where, $\mathbf{x_i}$ and $\mathbf{x_j}$ are the center node and neighbor node coordinates, respectively. There exists one forward edge and one backward edge between two connected nodes since every

node can be a center node and a self-loop is added to each node with the distance of **0**. Therefore the bi-directed graphs with self-loops are generated, as shown in Fig. 6.2.

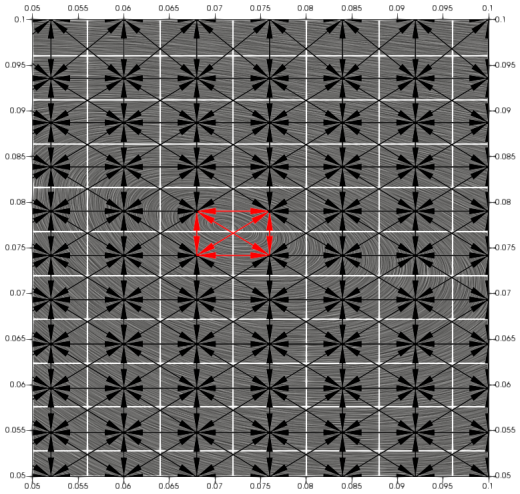


Figure 6.2: Bi-directed graph superimposed on streamline background. The red arrows connect the vortex core cells. The self-loop of each node is not show here. The white wire frame is the CFD mesh.

GNNs perform feature aggregating and updating on the graphs. The transformation from CFD mesh to graph is very straightforward. Each cell in a CFD mesh corresponds to one node in the graph. The connectivity between the nodes in the graph can be obtained based on either shared vertex or shared face between adjacent cells in the CFD mesh. The graph derived from a 2D structured mesh based on shared-vertex connectivity is the natural analogue to the images for CNNs where each node in the graph has eight neighbors except for those on the border. A directed graph with self-loops is constructed with the attributes $\mathbf{e}_{ij}$ stored on the edges representing the relative orientation of the neighbors to the center. Only the edge direction is kept in order to regularize the graph since the magnitude of the distance between cells may vary significantly from case to case:

$$\mathbf{e}_{ij} = \frac{\mathbf{x}_j - \mathbf{x}_i}{||\mathbf{x}_j - \mathbf{x}_i||}. \tag{6.4}$$

The velocity direction $\mathbf{u}_{norm}$ is stored on the nodes as the node features:

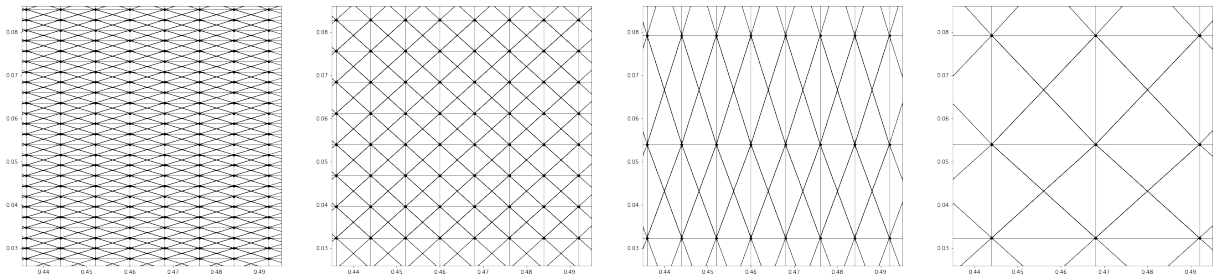
$$\mathbf{u}_{norm} = \frac{\mathbf{u}}{||\mathbf{u}||}. \tag{6.5}$$

6.2.2 Graph hierarchy generation

Since we intend to use U-net architecture to build the GNN model whose advantages over other architectures will be shown in the following subsection, a hierarchy of coarsened graphs to down-sample and up-sample the hidden features should be provided to the multiple levels in U-Net architecture. Unlike the already existed pooling and unpooling operations in CNNs, there is no widely accepted down-sampling and up-sampling method in GNNs. Thus a proper graph coarsening algorithm should be proposed and evaluated in order to build a mapping relationship between fine and coarse graphs. In the following trainings, the average down-sampling as shown in Fig. 3.6 and nearest upsampling as

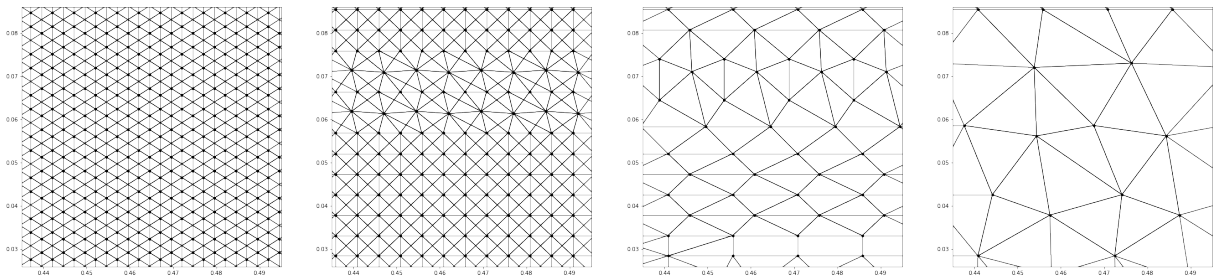
shown in Fig. 3.7b are used for GNNs. Here, we use the algebraic multigrid method embedded in code_saturne to generate this hierarchy of graphs and the connectivity between two successive graphs. The inspiration of using AMG method is that it is originally designed to accelerate the convergence of linear algebraic equations by removing different wave-length components of errors, which we believe will also accelerate the training procedure. As a byproduct of CFD simulation, using AMG graph hierarchy brings no additional computational expense.

Fig. 6.3 and Fig. 6.4 respectively show a small portion of the graphs at four levels generated from 2D structured mesh and unstructured mesh for backward-facing step (BFS) case based on shared-vertices. The graph of level 1 in Fig. 6.3 is directly generated from original mesh and the graphs of other levels are generated by the AMG algorithm. Each node in the graph has a self-loop which is not shown for the clarity of visualization. It is clear that the graphs are evenly coarsened. The graphs' information of four levels are summarized in Table. 6.1.



(a) Level 1. *(b) Level 2.* *(c) Level 3.* *(d) Level 4*

Figure 6.3: The details of graphs generated from the 2D BFS structured mesh at different levels.



(a) Level 1. *(b) Level 2.* *(c) Level 3.* *(d) Level 4.*

Figure 6.4: The details of graphs generated from BFS unstructured mesh at different level.

6.2.3 Architecture selection

The U-Net architecture firstly proposed for biomedical image segmentation Ronneberger et al. (2015) is adopted because of three merits: 1) training efficiency, 2) accuracy and 3) stability. The training cost is significantly reduced because it successively down-samples the data from fine graph level to the next coarse level. Since the input data is down-sampled, the graph size shrinks and only the main features are kept to the next level. The same kernel size covers larger regions in coarser levels than in the fine levels, which enables the model to capture global features. In other words, this architecture is accurate because it captures the details at shallower levels and global features at deeper levels. The

Table 6.1: The details of graphs.

Graph	Level	Nodes	Edges	Average node degree
BFS structured mesh	1	74400	666124	9.0
	2	24720	220002	8.9
	3	8231	72401	8.8
	4	2719	23727	8.7
BFS unstructured mesh	1	131118	907510	6.9
	2	50780	407126	8.0
	3	17127	118619	6.9
	4	5596	38384	6.9

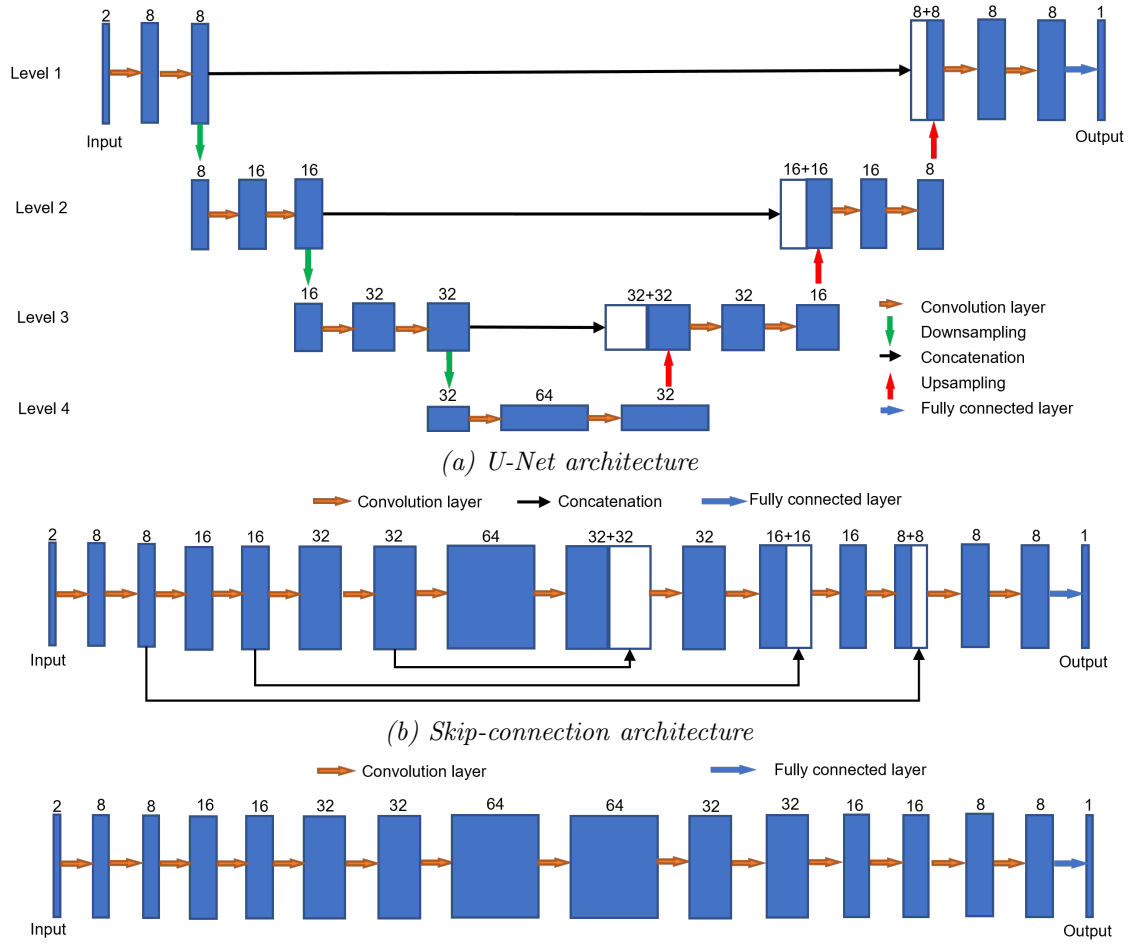
skip-connections between the leading contraction blocks and the corresponding trailing extraction blocks make the training more stable.

As shown in Fig. 6.5a, a four-level 14-layer U-Net architecture is used for GNN models which is similar to that used in section 5.2 except deeper. To demonstrated the feasibility and advantage of using U-Net architecture with AMG graph hierarchy, another two architectures, skip-connection and sequential connection, are constructed. Unlike the U-Net using a hierarchy of different coarsen levels of graphs, skip-connection architecture has no down-sampling and up-sampling between different levels of graphs and performs all the computation on the same graph. Thus, the skip-connection can be viewed as an intermediate architecture between U-Net and sequential architecture. Further removing the skip-connection leads to the sequential architecture. The number of trainables for each architecture is summarized in Table 6.2. All the three architectures have the same number of convolution layers, the same channel number at the corresponding layers. Compared with the other two architectures, the sequential one has more trainables which is the result of removing skip-connections while keeping the channel number at the second half layers the same as those in the other two architectures. Other hyper-parameters, such as convolutional kernel and activation function, are kept the same as the U-Net architecture in section 5.2.

Table 6.2: Parameter details of different architectures.

No.	Kernel	Architecture	Layers	Kernel size	Trainables
1.	GMM	U-Net	14	9	86793
2.	GMM	Skip-connection	14	9	86793
3.	GMM	Sequential	14	9	111157

The performance of U-Net architecture with AMG graph hierarchy is evaluated on the data-set introduced in section 5.2 using GMM kernel. As shown in Fig. 6.6, the original GMM kernel has a large fluctuation on validation loss for all the three architectures considered. U-Net architecture does not distinguish itself from the other two in term of training loss history. However, as shown in Table 6.3, it requires significantly less training time per epoch and inference time per case, only about one third of those of the other two architectures, while achieving the highest mean values for all four classification evaluation criteria and smallest standard deviation values except for Accuracy thanks to the usage of multigrid down-sampling and up-sampling of the hidden features. The model with sequential architecture has the lowest mean value on four criteria and largest standard



(a) U-Net architecture

(b) Skip-connection architecture

(c) Sequential architecture

Figure 6.5: Three tested architectures.

deviation values except for Precision even though it has more trainables. As shown in Fig. 6.7, U-Net architecture has a ROC curve closest to the upper left corner while sequential architecture behaves the worst.

Table 6.3: Performance summary of three architectures on test cases. (Green: best value; Red: worst value.)

No.	Model	Accuracy	Precision	Recall	F1 score	Training time/epoch	Inference time/case
1.	GMM-Unet	88.58±0.28	91.15±0.91	63.80±1.31	75.04±0.80	85.5±0.3s	0.498s
2.	GMM-Skip	87.75±0.26	88.53±1.55	62.64±1.84	73.33±0.89	244.8±1.0s	1.433s
3.	GMM-Sequ	86.40±0.51	86.44±1.40	58.76±3.12	69.88±1.89	250.9±0.8s	1.467s

6.3 2D Vortex identification

6.3.1 Training details

The superiority of our proposed Fast-GMM kernel was demonstrated by comparing it with CNN, GMM, SplineCNN and GCN convolution kernels on U-Net architecture. The

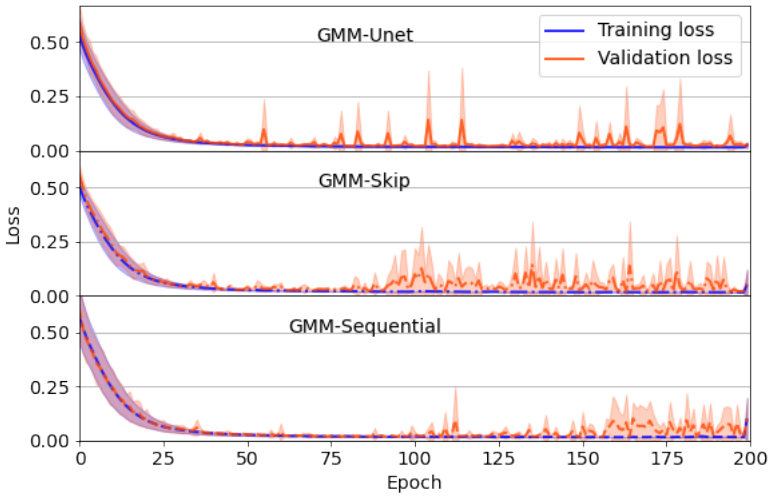


Figure 6.6: Loss history of GMM models with three different architectures.

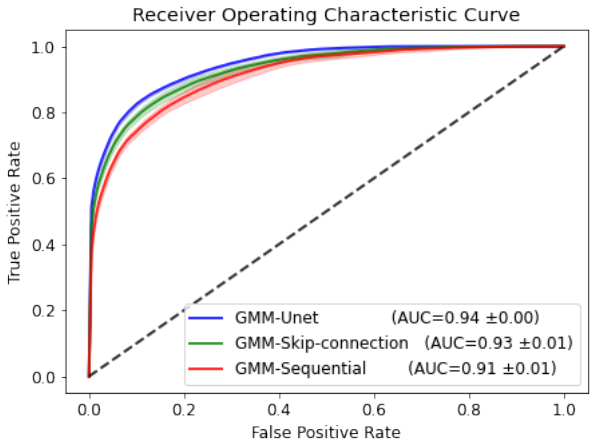


Figure 6.7: Receiver operating characteristic curves of GMM models with different architectures evaluated on test cases of BFS structured mesh.

main parameters of the tested models are summarized in Table. 6.4. It is noteworthy that the model with GCN kernel has much less trainables than other models as a result of its layers and channels being kept the same as other models instead of the trainables. According to our previous tests, the performance of a GNN model with GCN kernel can not be improved even if its trainables are increased to the same level of the others by increasing channel number proportionally at each layer.

To investigate the influence of different convolutional kernels and to show the adaptability of proposed approach to unstructured mesh, all the models are trained five times with different random seeds for 200 epochs on dataset generated from 2D BFS structured mesh introduced in section 5.2. The results are summarized in subsection 6.3.2 and 6.3.3, respectively. For all the GNN based models, the velocity on all the 74400 cells in the BFS structured mesh were used as the input. For CNN model, only 240×80 cells located in a rectangular region behind the step $(0 < x < 8h, 0 < y < h)$ were used as the input. Other traininig details are kept the same as those in section 5.2. The vortex identification

Table 6.4: Summary of hyper-parameters of different models.

No.	Kernel	Architecture	Layers	Kernel size	Trainables
1.	CNN	U-Net	14	3*3	85697
2.	GMM	U-Net	14	9	86793
3.	Fast-GMM	U-Net	14	9	85697
4.	SplineCNN	U-Net	14	3*3	85697
5.	GCN	U-Net	14	-	9793

performances of all the models are evaluated in terms of the four classification metrics: accuracy, precision, recall and F1 score, on the points in the vortex region behind the step ($0 < x < 8h, 0 < y < h$) over test cases to ensure the fairness of the performance comparison between CNNs and GNNs. To further reveal the generality of different GNN convolutional kernels, the GNN models are trained on data-set formed by both structured and unstructured BFS meshes and then tested on unseen cases with different mesh topologies and different turbulence models at different Reynolds numbers. The results are included in subsection 6.3.4. Mesh types of the training and testing cases in the following sections are summarized in Table 6.5.

Table 6.5: Mesh types of the training and testing cases in the following sections.

	Training mesh type	Testing mesh type
Section 6.3.2	Structured	Structured
Section 6.3.3	Structured	Unstructured
Section 6.3.4	Structured and unstructured	Structured and unstructured

6.3.2 Kernel influence

Since the U-Net architecture leads to faster model training and a more accurate classification, we continue to test all kernels on this architecture. As shown in Table 6.6, all the highest values and lowest values for each column are colored in green and red respectively among all models except the one with GCN kernel whose bad performances are expected. The model with our proposed Fast-GMM kernel outperforms the other models on all classification evaluation criteria by a large margin, except for precision which is very close to the highest at the price of slightly larger standard deviation. And it requires significantly shorter training time compared with those with original GMM kernel and SplineCNN kernel (even though it is far behind the CNN-Unet model on computational efficiency), despite its inference time per case being slightly higher than GMM-Unet. The training time per epoch for CNN-Unet model is at least one order of magnitude lower not only because it takes one fifth of the points used by GNN models in our training but also mainly because the connectivity and orientation information between the neighboring points are already implicitly embedded in the arrangement of the array.

As shown in Fig. 6.8, CNN-Unet model converges the fastest, requiring around 15 epochs to give meaningful classification according to the authors' experience, while other GNN model needs at least 5 times as many epochs. The GNN model with our proposed Fast-GMM kernel converges faster than GMM-Unet and has no fluctuation in validation loss history. The validation loss curve overlaps the training loss curve for all the models except the GMM-Unet, which implies that no overfitting happens in the training.

Table 6.6: Performance summary of five kernel functions on test cases. (Among CNN-Unet, GMM-Unet, Fast-GMM Unet, SplineCNN-Unet: Green - best value; Red - worst value.)

No.	Model	Accuracy	Precision	Recall	F1 score	Training time/epoch	Inference time/case
1.	CNN-Unet	88.88±0.12	91.11±1.61	65.09±1.86	75.89±0.70	5.6±0.3s	0.008s
2.	GMM-Unet	88.58±0.28	91.15±0.91	63.80±1.31	75.04±0.80	85.5±0.3s	0.498s
3.	Fast-GMM-Unet	89.62±0.32	90.94±1.01	68.27±1.71	77.97±0.92	51.4±0.1s	0.580s
4.	SplineCNN-Unet	88.79±1.20	89.99±2.91	65.68±3.05	75.91±2.78	125.8±0.1s	0.831s
5.	GCN-Unet	77.74±1.08	66.50±1.68	35.06±7.83	45.32±7.25	62.3±0.8s	0.205s

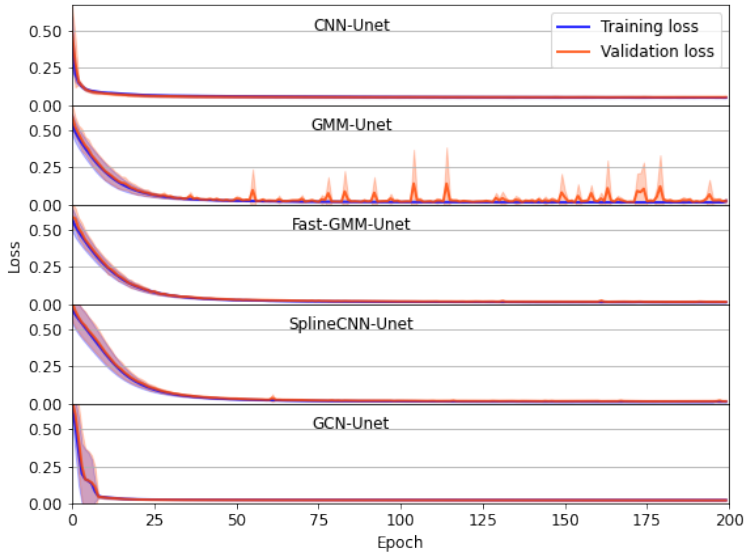


Figure 6.8: Loss history of five models with U-Net architecture.

The identifications by different models at a given time step in test cases are shown in Fig. 6.9. Among all the models, CNN-Unet model can identify the vortices with the sharpest boundary and with the shape closest to the ground-truth, while GCN-Unet model fails to identify separate vortexes which is as expected. By contrast, GMM-Unet model identifies the vortices smaller than the ground-truth with diffuse boundary which is especially visible for the second and last vortices counting from the left. Fast-GMM-Unet model and SplineCNN-Unet model are the two GNN-based models which can identify the vortices' shape closest to those identified by CNN-Unet and to ground-truth.

The performance of the Fast-GMM-Unet model is slightly higher than CNN-Unet judging from ROC plot shown in Fig. 6.10, and slightly better than both CNN-Unet and other GNN models. The SplineCNN-Unet model has the largest standard deviation value for a large range of FPR compared to other direction-aware GNN models which is aligned with the statistics in Table 6.6.

6.3.3 Adaptability to unstructured mesh

In this section, we demonstrate the generality of our proposed framework and Fast-GMM kernel to unstructured meshes and case not included in the training. A BFS case at

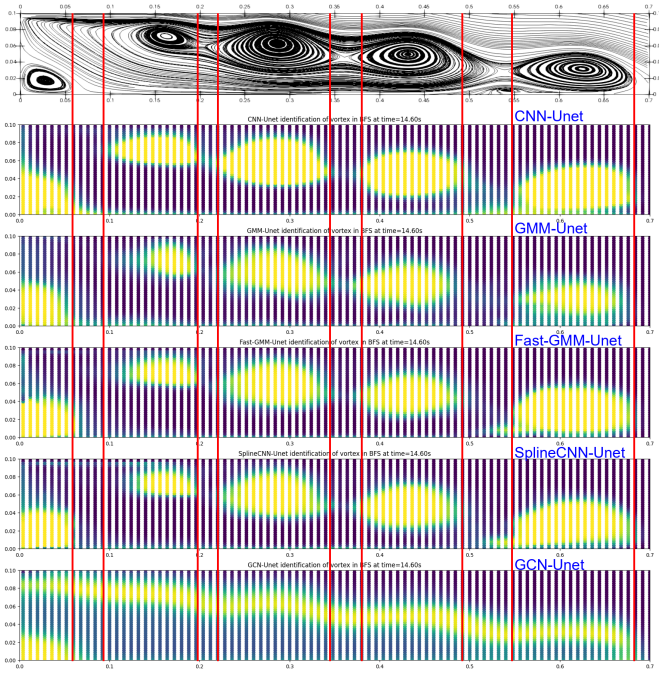


Figure 6.9: Comparison of the identificed vortices of five models on test cases of BFS structured mesh. (From top to bottom: Streamline of vortices in BFS; CNN-Unet; GMM-Unet; Fast-GMM-Unet; SplineCNN-Unet, GCN-Unet.)

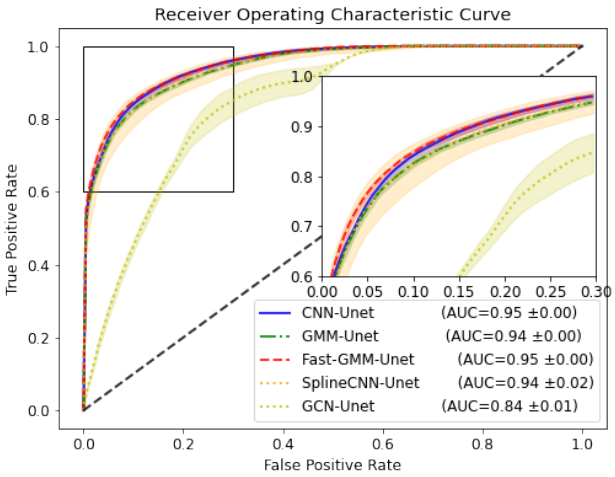
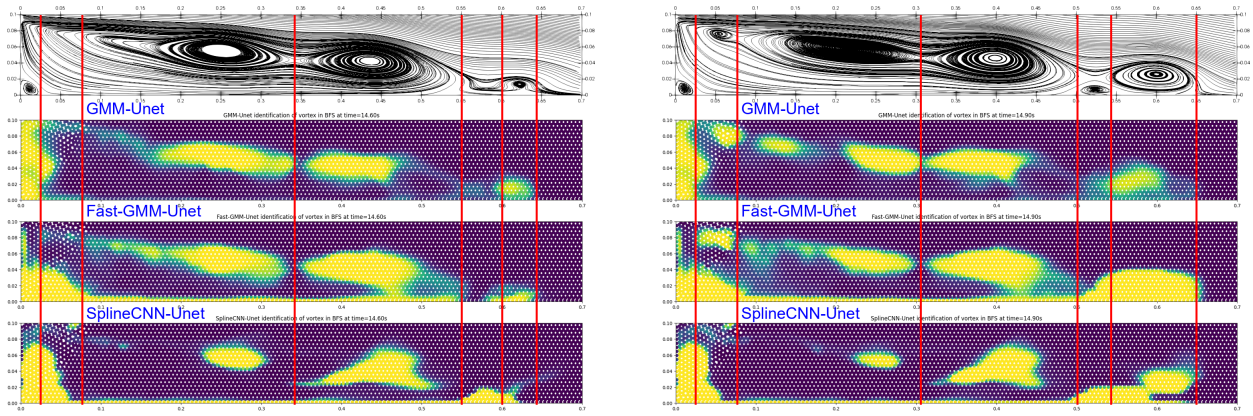


Figure 6.10: Receiver operating characteristic curves of different models with U-Net architecture evaluated on test cases of BFS structured mesh.

the same Reynolds number as that included in the dataset was simulated with the same turbulence model using an unstructured mesh. All the models are trained on graphs built from BFS structured mesh and directly tested on flow field and graphs obtained from BFS unstructured mesh and unseen periodic hill (PH) structured mesh. The graphs from unstructured BFS case are shown in Fig. 6.4. The graphs for BFS unstructured mesh are based on shared-faces since no significant model performance improvement observed when the connectivity is increased using shared-vertices. All the graphs' information including number of nodes, edges and average node degree is summarized in Table 6.1.

The vortices at two time steps identified by GMM-Unet, Fast-GMM-Unet and SplineCNN-

Unet are shown in Fig. 6.11 together with the streamline plots. While SplineCNN-Unet model fails to correctly identify the shape of most vortices, both GMM-Unet and Fast-GMM-Unet models capture the vortices morphology closer to the ground-truth. The Fast-GMM-Unet model tends to identify larger vortex regions and has the closest identification to the ground-truth than the other two models. All three models misidentify the near-wall region at the left as vortex, which could be ascribed to the differences between the structured mesh included in the dataset and the unstructured mesh. They also struggle to detect the elongated vortices, especially the stretched tail part which is possibly due to morphological differences between vortices in the dataset and those on this unseen unstructured mesh. The vortices' shapes are more compact on the structured mesh and more elongated on the unstructured mesh. Even with so many differences between the cases in the dataset and the unseen case, our proposed framework combined with Fast-GMM kernel still gives very satisfactory results. Therefore, we have good reasons to believe that with more cases of unstructured mesh included in the training dataset, the performance of our model can be further improved.



(a) *(b)*

Figure 6.11: Comparison of vortices identified by GMM-Unet, Fast-GMM-Unet and SplineCNN-Unet models on unseen BFS unstructured mesh at two time steps: (a) Time = 14.60s; (b) Time = 14.90s. (From top to bottom: streamline, GMM-Unet, Fast-GMM-Unet and SplineCNN-Unet.)

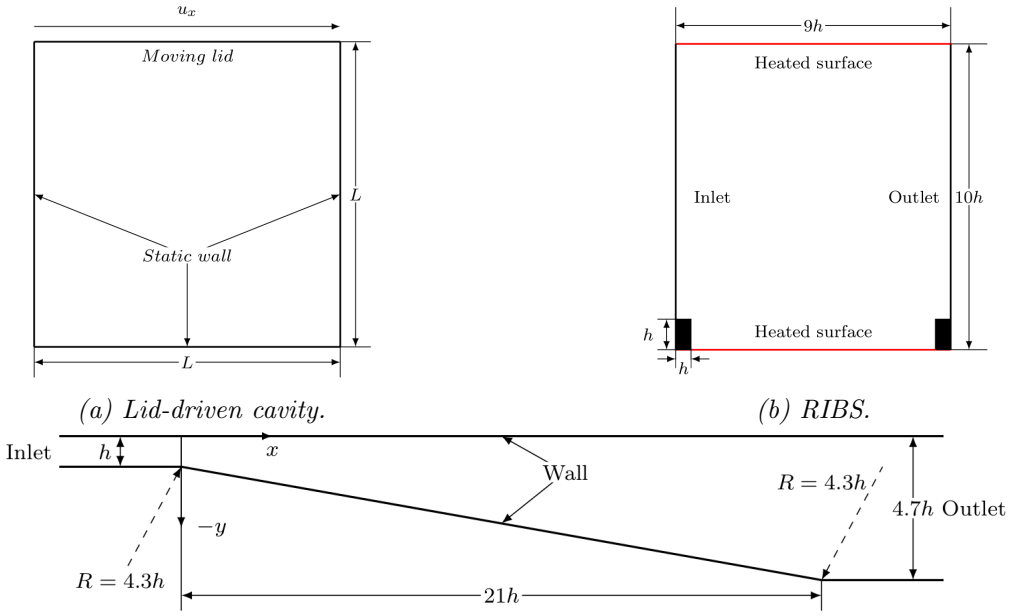
6.3.4 Generality analysis

To further evaluate the generality of the proposed approach on detecting vortexes with respect to different meshes in terms of mesh type, mesh density and mesh aspect ratio for structured mesh, different turbulence models and different Reynolds numbers, the GMM-Unet and SplineCNN-Unet models are trained again on the dataset formed by the BFS results obtained using both BFS structured and unstructured meshes and then tested on three of code_saturne validation cases: lid-driven cavity flow, heat transfer in a cooling channel with periodic ribs (RIBS) Rau et al. (1998); Arts et al. (2007), asymmetric plane diffuser flow Buice (1997); Obi et al. (1993). The main simulation details of these cases are summarized in Table 6.7. The configuration of three cases are shown in Fig. 6.12.

Lid-driven cavity. The lid-driven cavity flow configuration is shown in Fig. 6.12a. The top lid moves towards right direction and other walls are static. The no-slip conditions are applied on the walls. The Reynolds number is 5000. The $k-\omega$ SST turbulence model was

Table 6.7: Simulations details of four cases.

Case	Re	Turbulence model	Mesh		
			Type	AR	$No.\ cells_{(N_x \times N_y)}$
BFS	5100	$R_{ij} - \epsilon\ SSG$	Structured	3.2	74400
			Unstructured	-	131118
Cavity	5000	$k - \omega\ SST$	Structured	1	$90000_{300\times300}$
			Structured	1	$40000_{200\times200}$
			Structured	1	$10000_{100\times100}$
			Structured	1	$2500_{50\times50}$
			Structured	3	$2494_{29\times86}$
			Structured	6	$2500_{20\times125}$
			Structured	9	$2499_{14\times147}$
RIBS	30000	$k - \epsilon\ LP$ $k - \omega\ SST$ $R_{ij} - \epsilon\ SSG$	Structured	3	18296
			Unstructured	-	7735
Diffuser	18000	$k - \omega\ SST$	HR structured	$0.3 \sim 47.7$	$21648_{328\times66}$
			LR structured	$0.3 \sim 119.8$	$31488_{328\times96}$



(a) Lid-driven cavity. *(b) RIBS.*

(c) Diffuser.

Figure 6.12: Configurations of three cases.

used to simulated the cavity flow on the finest mesh which contains $N_x \times N_y = 300 \times 300$ cells. The velocity field is then interpolated to coarser meshes of different refinement levels from 2500 to 40000 cells of the same aspect ratio $AR = 1$, and two meshes of different aspect ratios from 1 to 9 with the cell number around 2500. The aspect ratio is defined as the ratio of the cell's length on the x direction to its width on the y direction.

RIBS. The RIBS configuration is shown in Fig. 6.12b. Air flows from the left to the right, at the atmospheric pressure and temperature. The left and right boundaries are set to be periodic. The top and bottom walls are heated while the two ribs are not. The Reynolds number and Prandtl number are 30000 and 0.71, respectively. The RIBS

case was simulated using three turbulence models with both structured and unstructured meshes.

Diffuser. The 2D flow inside a planar asymmetric diffuser, as shown in Fig. 6.12c, is simulated using $k-\omega$ SST turbulence model with fully-developed turbulent inlet at $Re = 18000$ based on the bulk inlet velocity and the inlet channel height with two types of meshes: high-Reynolds (HR) mesh and low-Reynolds (LR) mesh.

Mesh density influence. As show in Fig. 6.13, the vortex regions identified by both

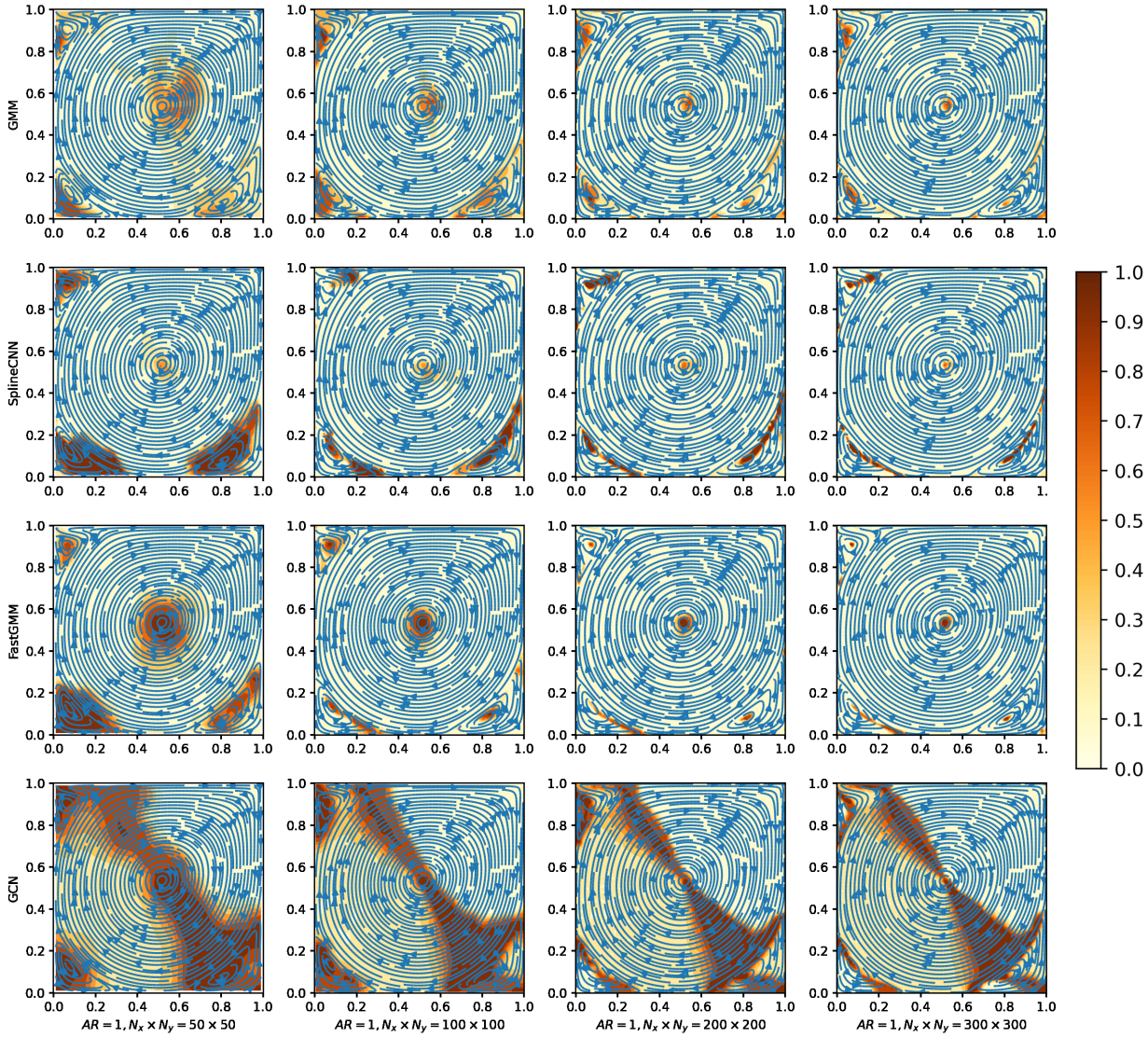


Figure 6.13: Identifications of vortexes on lid-driven cavity meshes of different refinement levels by GNN models.

GMM-Unet and SplineCNN-Unet in the coarsest mesh are the largest in the coarsest mesh. The identified regions become smaller as the mesh refinement level increases. The capability of recognizing a certain pattern of all the pure convolution-based machine learning models is limited to the size of effective receptive field (ERF). A trained model fails to correlate two points separated by a distance larger than the ERF which is determined by both the hyper-parameters of the model and the dataset. As indicated by one typical snapshot of streamline plot in Fig. 5.7, for the BFS structured mesh, 100×40 cells are

distributed in the vortex region where normally four to five vortexes exist. Thus, a single vortex in the training dataset covers no more than 25 cells on one specific direction. As a result, these two models trained on this dataset can only detect the vortexes of comparable sizes in terms of how many mesh cells they span.

Mesh aspect ratio influence. The aspect ratio of the BFS structured mesh included in the training dataset is $AR = 3.2$. As a result, the two models trained on this dataset well identified the vortexes on the mesh of $AR = 3$ as shown in Fig. 6.14. As the aspect ratio deviates from 3.2, their identification performances deteriorate which is more evident for SplineCNN-Unet. The SplineCNN-Unet model successfully identified three secondary vortexes in the corners on the mesh of $AR = 1$, all vortexes on the mesh of $AR = 3$, and the primary vortex center on the mesh of $AR = 6$, but failed on the mesh of $AR = 9$. Compared to SplineCNN-Unet, the GMM-Unet model identified more or less the same vortex region on the four meshes and thus has a better generality to the variation of the mesh aspect ratio. However, the vortex regions identified by the GMM-Unet model do not cover the vortex center and have diffuse boundaries compared with the SplineCNN-Unet.

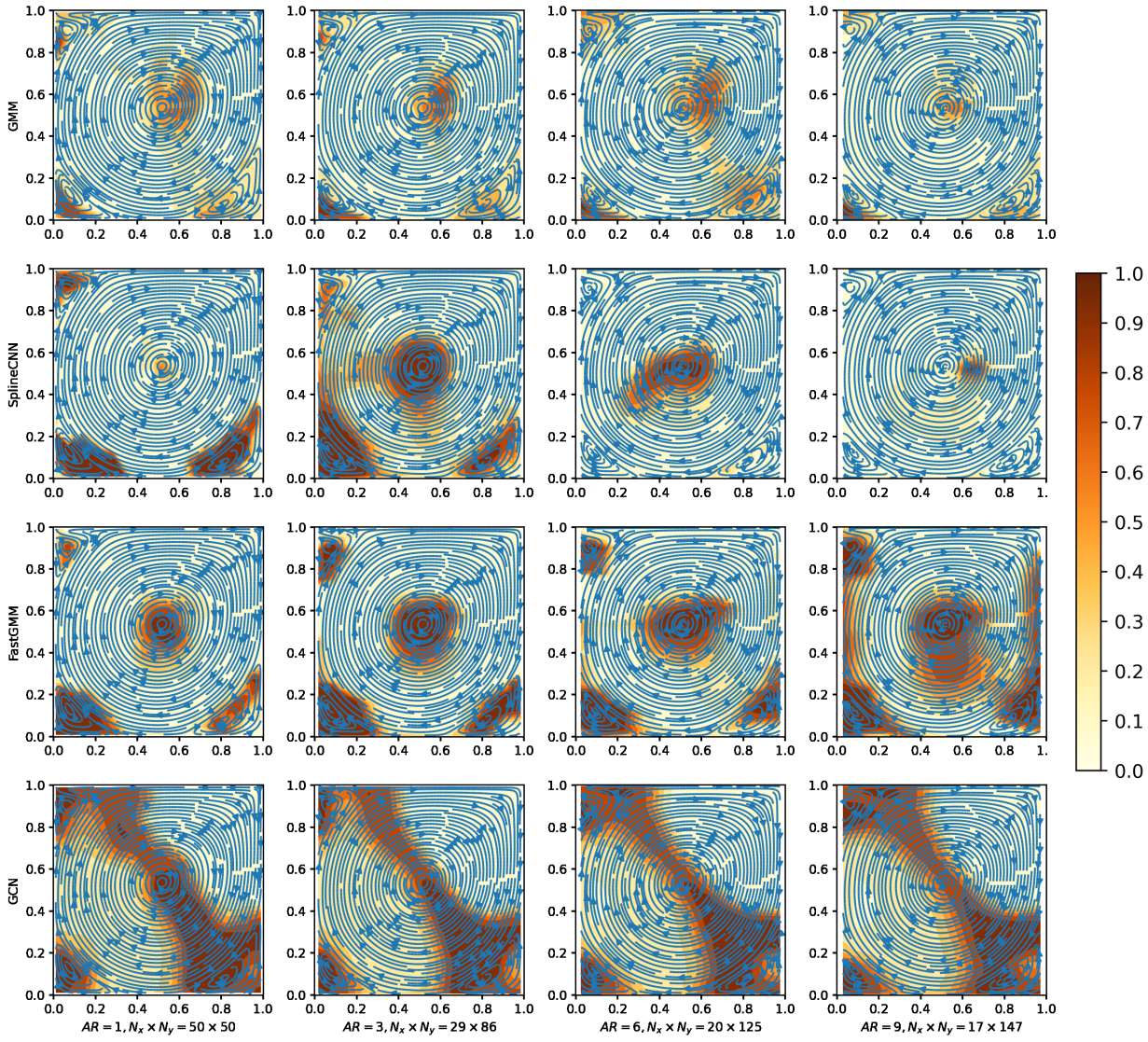
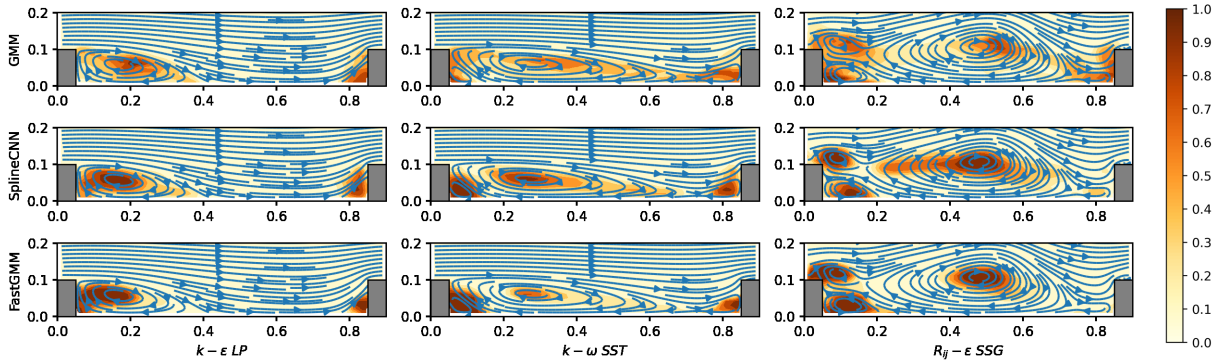
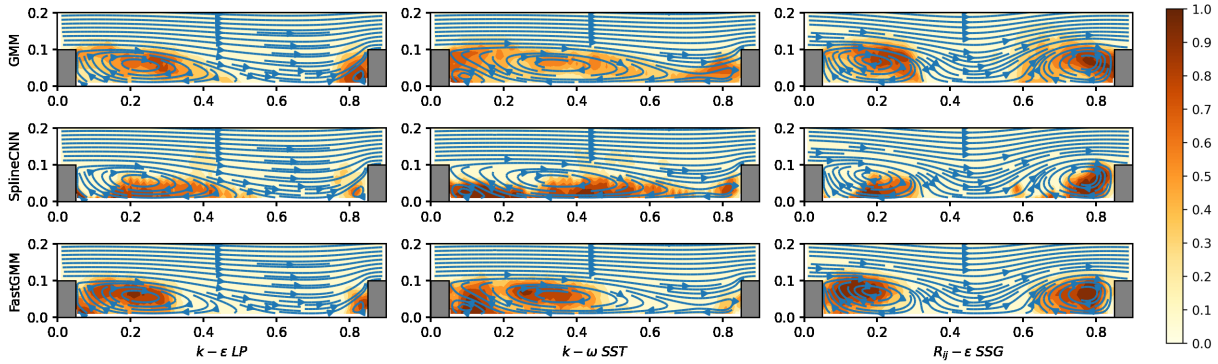


Figure 6.14: Identifications of vortexes on lid-driven cavity meshes of different aspect ratios by GNN models.

Mesh type influence. The mixture of both structured and unstructured meshes in the dataset poses no problem to the training. As shown in Fig. 6.15, while the SplineCNN-Unet model better identified the vortex center and shape on the structured mesh compared with the GMM-Unet model, but degraded more on the unstructured mesh where it only identified the lower half of the vortexes. The GMM-Unet model once again has better generality to the mesh type.



(a) Structured mesh.



(b) Unstructured mesh

Figure 6.15: Identified Vortexes by different GNN kernels on RIBS structured and unstructured meshes. GNN kernels from top to bottom: GMM, SplineCNN and Fast-GMM. Turbulence models from left to right: $k-\epsilon$ LP, $k-\omega$ SST and $R_{ij}-\epsilon$ SSG.

Turbulence models' influence. To test the sensitivity of the proposed approach to the turbulence models, three commonly used models, $k-\epsilon$ linear production, $k-\omega$ SST and $R_{ij}-\epsilon$ SSG, were selected because we intend to detect the vortexes generated by RANS turbulence models. As shown in Fig. 6.15, the turbulence models have no visible influence on the identification performance of the proposed approach. The robustness of our models to different turbulence models is explainable since they identify the vortexes based on the topological distribution of the velocity field which is universal among different turbulence models.

Mesh size scaling influence. As shown in Fig. 6.16, the cell size along the wall normal direction in the low-Reynolds diffuser mesh increases continuously, while in high-Reynolds diffuser mesh, the cell size from the first to the second layer perpendicular to the wall decreases abruptly. As shown in Fig. 6.17, both GMM-Unet and SplineCNN-Unet models can capture the vortexes on two meshes. On the high-Reynolds mesh, the identified vortex regions by both models are non-connected while those on the low-Reynolds mesh are closer

to the real vortex topology. Therefore the proposed approach is quite robust to the mesh size scaling but sensitive to the abrupt scaling jump.

(a) High-Reynolds mesh. *(b) Low-Reynolds mesh*

Figure 6.16: Two diffuser meshes.

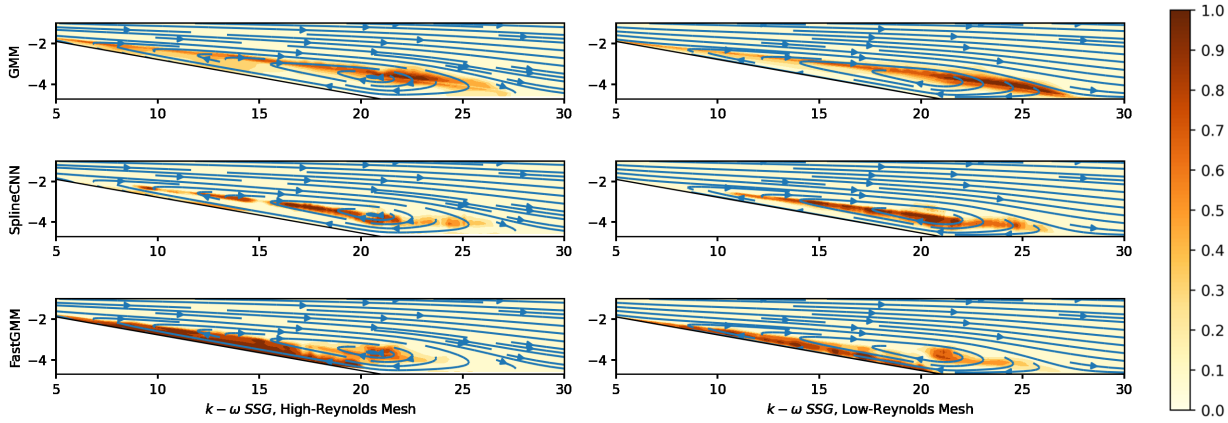


Figure 6.17: Identified vortexes of differet GNN kernels on two diffuser meshes. GNN kernels from top to bottom: GMM, SplineCNN and Fast-GMM. Turbulence models from left to right: high-Reynolds mesh and low-Reynolds mesh.

6.4 3D Vortex identification

In this section, the extensibility of the previous Fast-GMM-Unet model to 3D vortex identification is demonstrated on the vortex generator case Derksen (2005). The 3D counter-rotating vortex pair formed due to the injection of a high-velocity jet to the low-velocity crossflow is simulated as the 3D vortex dataset Karvinen and Ahlstedt (2005).

6.4.1 Dataset generation

The schematic of the computational domain is shown in Fig. 6.18. The computational domain contains a rectangle part with size of $40D \times 15D \times 15D$ at (x, y, z) directions and a vertical round pipe with diameter D. The right-ward horizontal crossflow enters the domain at velocity $1.435m/s$ from the left inlet located $10D$ upstream of the branch pipe through which the jet is injected to the crossflow. The outlet is located $30D$ downstream the pipe. The bottom boundary is wall while the rest three side boundaries are set to symmetry. To obtain a large dataset, a sinusoidal pulsating velocity component is added to the initial constant jet inlet velocity of $4.98m/s$ after the static flow field is fully established leading to the following jet velocity inlet:

$$w_j = 4.95 + sin(\frac{\pi t}{2}). \tag{6.6}$$

The Reynolds number Re_D based on the jet velocity and pipe diameter is in the range [6488, 9772]. The mapped inlet boundary is used for two inlets where the velocity profile at $10D$ downstream the inlet is mapped to the inlet to obtain a fully developed velocity profile.

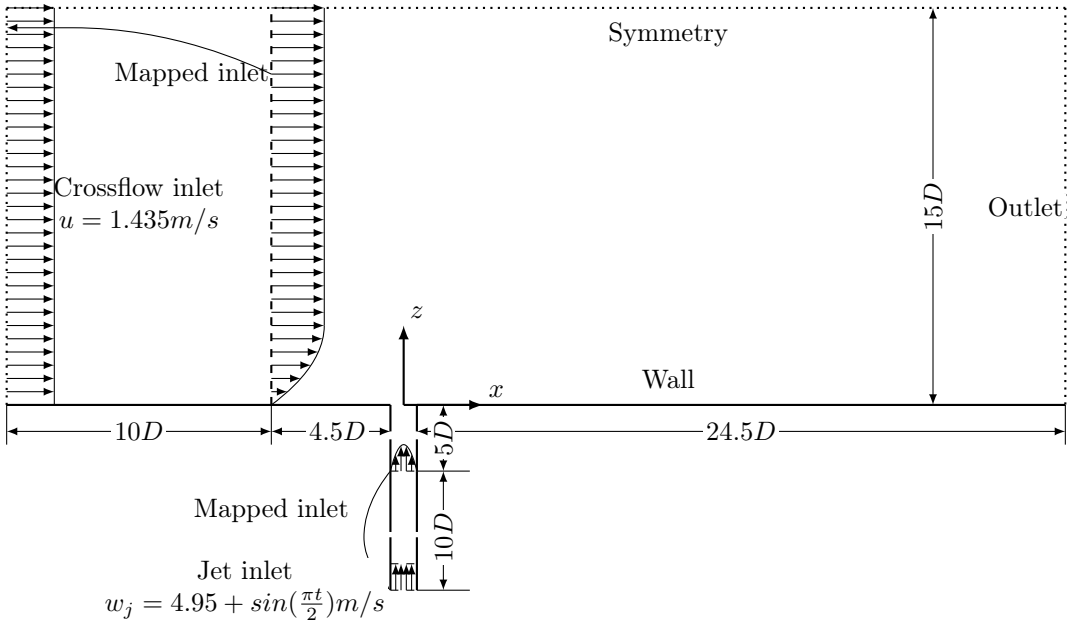
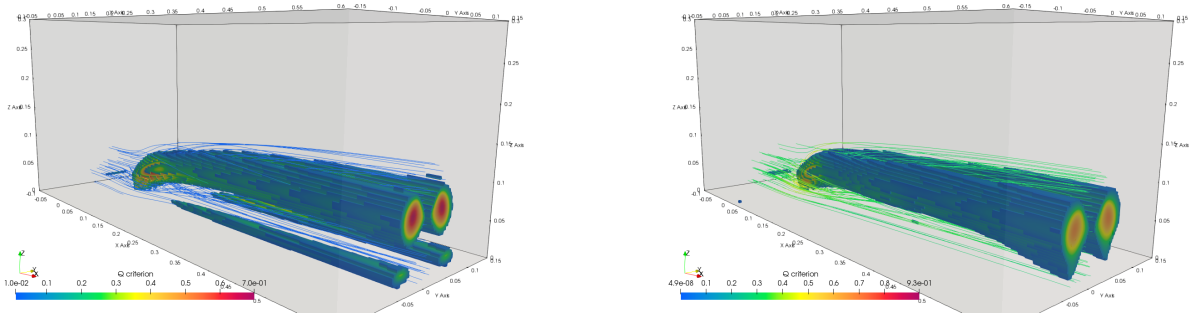


Figure 6.18: Schematic of the cross-section on OXZ plane through the jet inlet axis of 3D jet-in-crossflow case.

The jet-in-crossflow case is simulated by two turbulence models, $k-\epsilon$ and $k-\omega$ SST using transient solver with a structured mesh of 921600 cells. The results at 20 time instances evenly distributed within one pulsating period are sampled to form the dataset. The dataset is further augmented by rotating the results 90° around y axis resulting in 80 samples in the final dataset. Cells with Q-criterion larger than 0.01 are labeled as vortex region. This threshold is selected to remove the false vortex region near the bottom wall. The labeled vortex regions for two simulations at one time instance are shown in Fig. 6.19.



(a) $k-\epsilon$. *(b) $k-\omega$ SST.*

Figure 6.19: Vortex region labeled using Q-criterion in 3D jet-in-crossflow case obtained by two turbulence models.

6.4.2 Results analysis

As shown in Fig. 6.20, a 3D Fast-GMM kernel which contains 9 directions on 3 layers at Z direction, thus 27 direction in total, is constructed. The 8-layer architecture in Section 5.2 is utilised instead of the 14-layer architecture in the previous section due to the consideration of reducing memory consumption as a result of the tripling of the direction number in the kernel, leading to 166729 trainable parameters. This model was trained on one GPU node with 4 Tesla V100 GPUs on EDF Cronos cluster for 200 epochs which took around 1.7 hours.

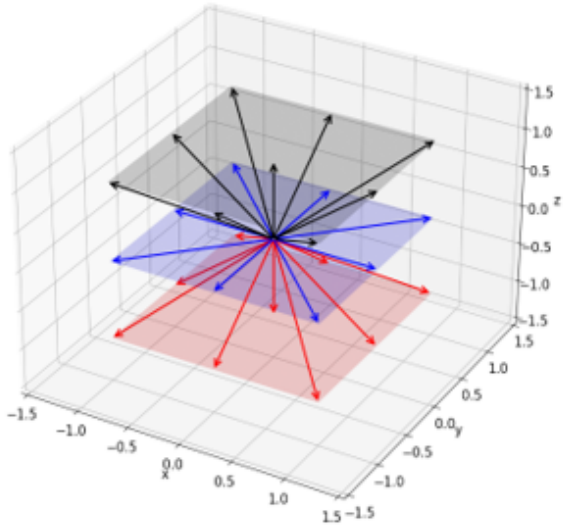


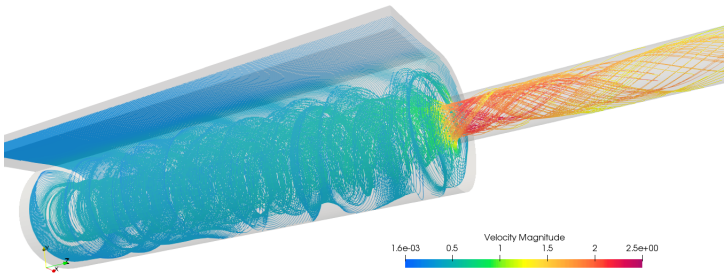
Figure 6.20: 3D Fast-GMM kernel.

The trained GNN model is tested on identifying the confined swirling flow inside a vortex generator Derksen (2005). Two inlet scenarios are simulated, one with inlet at the side slot, as shown in Fig. 6.21, and another one with inlet at the end of the large swirl tube, as shown in Fig. 6.22.

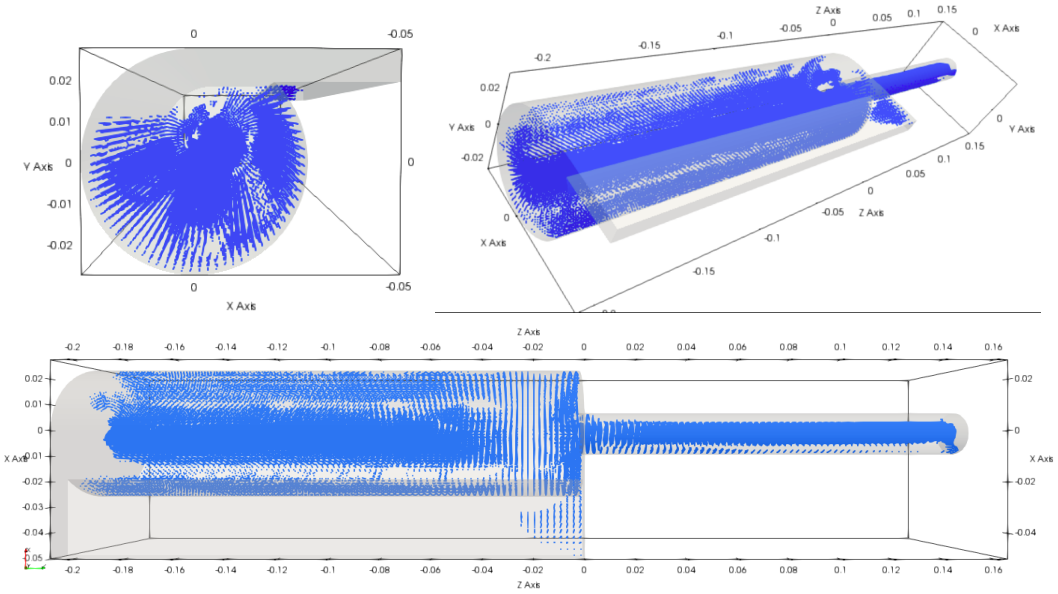
For the first scenario, GNN model successfully identified the main body of the vortex in both the swirl tube and exit pipe. However, its identification is not continuous, there is a clearance between the vortex core and the identified peripheral part of the vortex. It also misclassified the corner in the inlet slot near the contraction as a vortex region. While for the case with inlet at the end of the swirl tube there only exist vortexes near the contraction, the GNN model correctly identified a vortex region near the contraction but also made wrong classification near the inlet and inside the exit pipe.

6.5 Summary

In this chapter, we proposed a GNN-based framework to identify the flow phenomena based on CFD computations with the proposed Fast-GMM as the convolution kernel and U-Net architecture, which accepts unstructured data and therefore can be directly applied to the CFD results produced from real industrial cases using unstructured meshes. The Fast-GMM significantly increases the computational efficiency compared with the original GMM while maintaining a comparable identification performance to that achieved by the traditional CNNs. The U-Net architecture uses the graph hierarchy generated by an algebraic multigrid method (which is used in many CFD codes to accelerate the



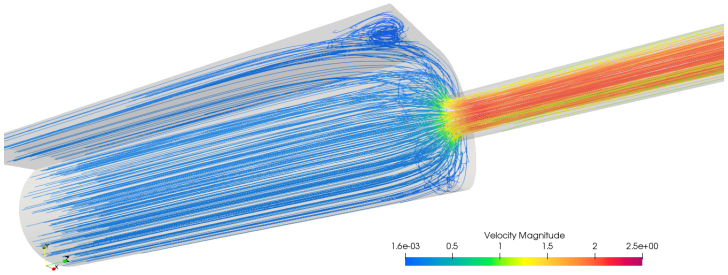
(a) Streamline.



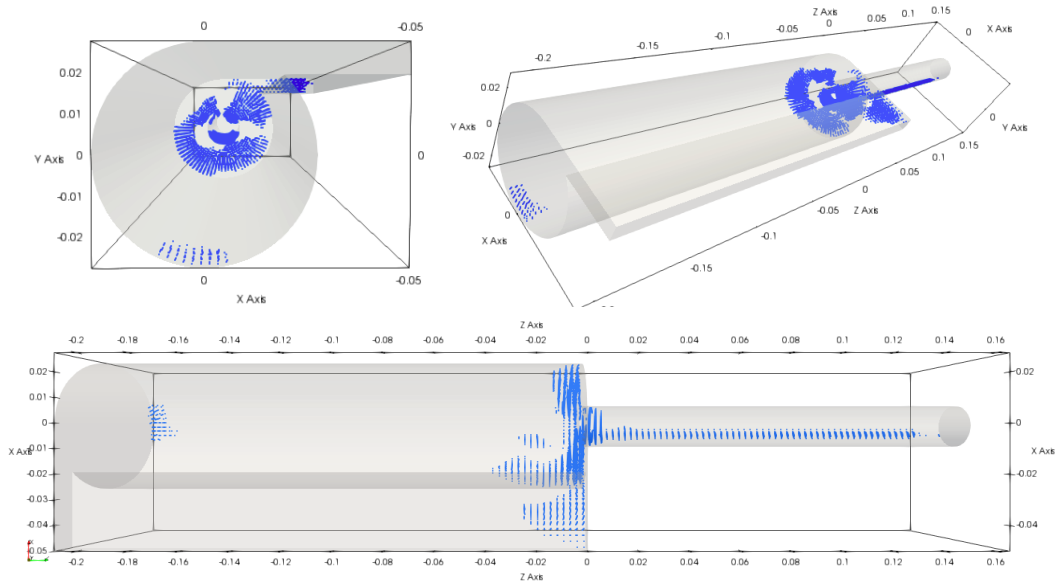
(b) Vortex region identified by GNN model.

Figure 6.21: Streamline plot and vortex region identified by GNN on vortex generator case with side inlet.

convergence of the iterative elliptic solvers), to shorten the training time and thus endows the framework the potential to be organically integrated into our open source CFD solver code_saturne in the future. The superiority of the proposed Fast-GMM kernel against the original GMM, SplineCNN and GCN, and the advantage of U-Net architecture against skip-connection and sequential architectures are experimentally demonstrated in terms of identification performance and computational efficiency on detecting vortices in 2D cases with both structured and unstructured meshes. The proposed framework on detecting 3D vortex is also exemplified in the vortex generator case.



(a) Streamline.



(b) Vortex region identified by GNN model.

Figure 6.22: Streamline plot and vortex region identified by GNN on vortex generator case with left inlet.

7 | Conclusions and perspectives

Contents

7.1 Conclusions

In the current work, state-of-the-art algorithms from the computer vision domain were adapted to fluid dynamics to identify the flow phenomena on the CFD results.

First of all, a review of machine learning applications in several aspects of fluid dynamics, including flow phenomena identification, super resolution, surrogate modelling, turbulence modelling, reduced order modeling, was conducted. CNNs are widely adopted in these different tasks because of their capability of processing the spatial features and good generality.

Before applying the CNN model to identify flow phenomena, the possible influencing factors on the CNN's accuracy were explored in regression of the pressure around cylinder case based on the velocity field in the wake region. We started by using CNNs to detect the first phenomenon on 2D meshes - the vortex region behind the backward facing step. An automatic 2D vortex labelling method based on first-depth search and random walking algorithms was proposed to label the vortex regions on both structured and unstructured meshes. The potential features characterizing the vortex region were evaluated as input for the CNN model with U-Net architecture. The results showed that CNN can well identify the separated vortexes from the normalized velocity field. The extendability of this model to other flow phenomenon, provided with a properly labelled dataset, was also demonstrated on thermal stratification, requiring only a slight modification to the architecture - adding gravity direction.

This CNN-UNet model template was then extended to GNNs in pursuit of directly consuming data on unstructured meshes. The methodology of constructing the bidirectional graphs from CFD meshes was detailed. Several promising graph convolutional kernels, such as GMM, SplineCNN and GCN, were compared with the conventional kernel in CNNs on structured mesh. A novel graph convolutional kernel - FastGMM was proposed by simplifying the original GMM which is more suitable for graphs derived from unstructured CFD meshes. The results suggested that the proposed FastGMM kernel has better identification accuracy on unstructured meshes and is more computational efficient. Inspired by the idea of AMG accelerating the iteration convergence of linear algebraic equations, the graph hierarchy generated from AMG method was used for the U-Net architecture to enable the model to gather and update hidden features of multi-scales.

The results evidenced that the AMG U-Net architecture leads to a higher identification accuracy and considerable improved computational efficiency compared with sequential- and skip-architecture. The proposed framework is also able to detect 3D flow phenomena such as the vortex structure inside the 3D vortex generator case.

7.2 Perspectives

As the first exploration in the département mécanique des fluides, energie et environnement (MFEE) of using ML algorithms to detect flow phenomena in CFD results, some deficiencies are inevitable and there remain several directions to be explored.

At the current stage, only two flow phenomena were detected, vortex and thermal stratification regions. In the future, more flow features, such as boundary layer and jet, can be identified. The grid coarsening could be based on flow phenomenon related systems instead of that used for the pressure correction. Take the vortex detection for example, we could merge cells having the most similar values of velocity direction and dynamic pressure.

CNN/GNN methods are perplexed by the problem induced by the fixed effective receptive field which is limited by the model hyperparameters such as architecture, kernel size, and training data. It is a problem when the model is applied to a case with the same geometry but with meshes of different refinement levels. Including more target flow phenomena of large scales in the training dataset can alleviate this problem to some extend. There is potential of detecting larger flow structures for the U-Net architecture. Including more levels of graphs in the U-Net architecture will enlarge the effective receptive field. Another possible direction is to compress the flow feature from the original mesh to deeper graphs and directly perform the feature identification on this deep graph. This can be combined with other types of neural networks such as transformer or recurrent neural network. However the former performs the calculation on the entire matrix which is very expensive, while the latter consumes a series of unlimited number of inputs which is compatible with non predefined number of depth in AMG hierarchy.

This work adopts supervised training that requires labeled datasets, which are not easily obtained for flow phenomena identification tasks. Since for the recommendations of CFD case configuration, we do not need to know exactly where the target flow features are located, but only some elements/statistics (i.e. strong presence, size, probably some orientation related elements, ...), so the requirements of the input for the machine learning model may be slightly relaxed (though finer detection is always useful for understanding/debugging what the models does). In the future, unsupervised training algorithms such as generative adversarial network can be explored.

This work currently focuses on the first step for improving confidence on CFD result - identifying flow phenomena. There is one possible direction for the second step - giving suggestions for CFD user to better configure CFD calculation case. Since the model form uncertainty in the turbulence models' prediction of Reynolds stress highly influences CFD result accuracy, it can be used as an indicator of CFD result accuracy. One possible direction is to build a surrogate model trained on high fidelity dataset formed by DNS and LES simulations to predict the Reynolds stress from the time averaged flow fields. The deviation between the Reynolds stress tensor predicted by turbulence model and that reconstructed by surrogate model indicates whether the result should be trusted.

A | Machine learning code snippets

The machine learning source code is included in this appendix.

The vortex labeling algorithm code on 2D CFD mesh is shown in 'labelSeparateVortex.py' A.1.

The input data including flow features, label and mesh information is extracted from the csv file output by code_saturne and prepared in 'preDataset.py' A.2. The data is split into training, tesing and validation datasets and saved into three files respectively which are readable for pytorch.

The kernel functions and model architecture are written in 'model.py' A.3. Every kernel/layer/model is a class. All the parameters related to the kernel/layer/model are defined in the __init__() property. How the kernel/layer/model calculates the output features from the input features is defined in forward() function.

The training loop is initiated by running 'train.py' A.4. This script reads the datasets and load the models one by one from 'model.py' A.3. Before entering the training loop, an optimizer is initialized using Adam algorithm with predefined learning rate. The graphs and the corresponding flow features in the dataset are fed to the model sequentially. The loss between the label and the output of the model is calculated and the gradients for all the trainable parameters are accumulated for every batch of data. After reading a batch of data, the trainables are updated by optimizer.step() function and the gradient are set to zero by optimizer.zero_grad(). It is called one training epoch when all the data is fed to the model one time. At the end of one epoch, the model with updated trainables is evaluated to obtain the training, testing and validation losses.

The hyperparameters about the model architecture, such as the dimensions of the hidden feature in each layer, depth of UNet architecture, are defined in 'param.py' A.5.

Other useful functions are included in 'utils.py' A.6. The algorithm (binary cross entropy selected here) used for loss calculation is defined in loss_eval(). The ROC and the area under ROC are calculated by roc_eval() and auc_eval(), respectively. Four classficiation evaluation criteria are calculated by metrics(). Other functions are used to output the loss plot, ROC plot, AUC plot and identified vortex region plot.

Code Listing A.1: labelSeparateVortex.py

```
# Python program to detect cycle
# in a graph
import os
import math
import numpy as np
import random
import pandas as pd
from copy import deepcopy
import matplotlib.pyplot as plt
from collections import defaultdict
import networkx as nx
```

```
from joblib import Parallel, delayed


def noPtInSet(ptset, pts):
    no_pt_included = True
    for ipt in pts:
        if (ipt in ptset):
            no_pt_included = False
            break
    return no_pt_included

def allPtInSet(ptset, pts):
    all_pt_included = True
    for ipt in pts:
        if (ipt not in ptset):
            all_pt_included = False
            break
    return all_pt_included


def twoHopsDsPt(graph, pts):
    ptfound = False
    for icore in pts:
        """
        if (icore in bounds_of_all_vortices):
            continue
        """
        if (ptfound == True) :
            break
        for ids_core in list(graph.adj[icore]):
            if (ptfound == True) :
                break
            if (ids_core not in pts):
                for ids_ds_core in list(graph.adj[ids_core]):
                    if (ids_ds_core not in pts):
                        start_node = ids_ds_core
                        ptfound = True
                        break
    if (ptfound == False):
        print("Starting point is not found !!!")
        return False
    else:
        return start_node

def delSubgraph(graph, subg):
    graph.remove_nodes_from(subg)
    for iedge in list(graph.edges()):
        if (iedge[0] in subg or iedge[1] in subg):
            graph.remove_edge(iedge[0], iedge[1])
    graph.normFlux()
    return graph


def plotBound(coord, bound, core_list, figname):
    x_min = 0.0
    x_max = 0.80
    y_min = 0.00
    y_max = 0.120
```

```
    size = 40
    diag_size = math.sqrt(pow(y_max-y_min, 2)+pow(x_max-x_min, 2))

    fig, ax = plt.subplots(figsize=((x_max-x_min)/diag_size* size, (y_max-y_min)/diag_size*size),
                           constrained_layout=True)

    bound_coord = coord[bound]
    ax.scatter(bound_coord[:, 0], bound_coord[:, 1], c='b')
    for i, icore in enumerate(core_list):
        core_coord = coord[list(icore)]
        ax.scatter(core_coord[:, 0], core_coord[:, 1], c='r')
    ax.set_aspect('equal')
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    #plt.savefig(figname, transparent=True)
    plt.show()
    plt.close()
    return

def plotBoundInGraph(g, coord, bound, vortex_core, more_in_list, more_out_list, figname):
    print("   Saving Figure!!!")
    width = 0.0001
    scatter_size = 200

    x_min = 0.0
    x_max = 0.80
    y_min = 0.00
    y_max = 0.120

    size = 20
    diag_size = math.sqrt(pow(y_max-y_min, 2)+pow(x_max-x_min, 2))

    fig, ax = plt.subplots(figsize=((x_max-x_min)/diag_size* size, (y_max-y_min)/diag_size*size),
                               constrained_layout=True)

    for start in list(g.nodes()):
        x = coord[start][0]
        y = coord[start][1]
        if (x<x_max and x>x_min and y<y_max and y>y_min):
            for end in list(g.adj[start]):

                dx = coord[end][0] - x
                dy = coord[end][1] - y
                length = math.sqrt(pow(dx,2)+pow(dy,2))

                arrow_color = 'k'
                """
                if (dx>0 or dy >0):
                    arrow_color = 'g'
                """
                if (start in bound and end in bound):
                    arrow_color = 'r'
                if (start in vortex_core and end in vortex_core):
                    arrow_color = 'r'

```

```
                ax.arrow(x, y, dx, dy,
                         width = width, length_includes_head=True,
                         head_width=0.1*length, head_length=length*0.3,
                         ec=arrow_color, fc=arrow_color)

                if (start in bound ):
                    ax.scatter(x, y, s=scatter_size, c='r')
                elif (start in more_in_list):
                    ax.scatter(x, y, s=scatter_size, c='g')
                elif (start in more_out_list):
                    ax.scatter(x, y, s=scatter_size, c='b')
                else:
                    ax.scatter(x, y, s=scatter_size, c='k')
    #ax.set_xticklabels([])
    #ax.set_yticklabels([])
    ax.set_aspect('equal')
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.savefig(figname, transparent=True)
    #plt.show()
    return

class MyGraph(nx.DiGraph):
    def __init__(self,vertices):
        super(MyGraph, self).__init__()
        self.graph = defaultdict(list)
        self.upstream = defaultdict(list)
        self.flux = defaultdict(list)
        self.V = vertices


    def normFlux(self):
        for inode in self.nodes():
            sumflux = 0.0
            for iflu in self[inode].values():
                sumflux = sumflux+abs(iflu['flux'])
            for iout in self[inode].keys():
                self[inode][iout]['weight'] = abs(self[inode][iout]['
    flux'])/sumflux
        return

    def findVortCoreUtil(self, v, visited, depth, max_depth):

        for neighbour in list(self.adj[v]):
            if visited[neighbour] == True:
                return True
            elif depth < max_depth:
                if self.findVortCoreUtil(neighbour, visited, depth+1,
    max_depth) == True:
                    return True
        return False

    def findVortCore(self, roi = None, max_depth = 4):
        vort_core_list = []

        if roi == None:
            roi = range(self.number_of_nodes())

```

```
        for node in roi:
            visited = [False] * (self.V + 1)
            visited[node] = True

            if self.findVortCoreUtil(node, visited, 0, max_depth) ==
    True:
                vort_core_list.append(node)

        graphCopy = deepcopy(self)
        for inode in self.nodes():
            if (inode not in vort_core_list):
                graphCopy.remove_node(inode)

        return list(nx.weakly_connected_components(graphCopy))


    def findBoundary(self, start_node=3843):
        closed = False
        count = 0
        while closed == False:
            loop_list = []
            loop_list.append(start_node)
            prev_node = deepcopy(start_node)
            count = count+1
            length = 0

            while (True):
                #print("start_node ={}, count = {}, node = {}".format(
    start_node, count, node))
                if (len(self[prev_node]) == 0):
                    break
                elif (len(self.out_edges(prev_node)) == 1):
                    next_node = list(self.adj[prev_node])[0]
                else:
                    next_node = np.random.choice(list(self.adj[prev_node
    ]),
                                                 p=[i['weight'] for i in self[
    prev_node].values()])

                if next_node == start_node:
                    closed = True
                    print("count = {}, closed flag = {}, length = {}".
    format(count, closed, length))
                    #print(loop_list)
                    break
                elif (next_node in loop_list[1:]):
                    break
                loop_list.append(next_node)
                prev_node = deepcopy(next_node)
                length = length+1
            if (count > 10000):
                print("#### Maximum searching times 10000 reached! ####"
    )
                loop_list = []
                break
        return set(loop_list)

    def insideBoundary(self, bd_ids):
```

```
233
234         bd_coord_ar = np.zeros((len(bd_ids), 2))
235
236         inside_bound = []
237         for i, ibd_id in enumerate(bd_ids):
238             bd_coord_ar[i][0] = self.nodes[ibd_id]['coord'][0]
239             bd_coord_ar[i][1] = self.nodes[ibd_id]['coord'][1]
240
241         barycenter = np.mean(bd_coord_ar, axis=0)
242         print("barycenter = ", barycenter)
243         xc = barycenter[0] *100
244         yc = barycenter[1] *100
245         min_coord = np.min(bd_coord_ar, axis=0)
246         max_coord = np.max(bd_coord_ar, axis=0)
247
248         for ipt in self.nodes():
249             if (self.nodes[ipt]['coord'][0]>min_coord[0] and
250                 self.nodes[ipt]['coord'][0]<max_coord[0] and
251                 self.nodes[ipt]['coord'][1]>min_coord[1] and
252                 self.nodes[ipt]['coord'][1]<max_coord[1]):
253
254                 xp = self.nodes[ipt]['coord'][0] *100
255                 yp = self.nodes[ipt]['coord'][1] *100
256                 outside_flag = False
257                 for ibd in range(len(bd_ids)):
258                     x0 = bd_coord_ar[ibd][0] *100
259                     y0 = bd_coord_ar[ibd][1] *100
260                     x1 = bd_coord_ar[(ibd+1)%len(bd_ids)][0] *100
261                     y1 = bd_coord_ar[(ibd+1)%len(bd_ids)][1] *100
262
263                     A = np.array([[yc-yp, -xc+xp],
264                                   [y1-y0, -x1+x0]])
265                     b = np.array([[-(xc-xp)*yp+xp*(yc-yp)],
266                                   [-(x1-x0)*y0+x0*(y1-y0)]])
267                     inter = np.linalg.solve(A, b)
268
269                     if ((inter[1]-yc)*(inter[1]-yp)<=0 and (inter[0]-xc)
    *(inter[0]-xp)<=0):
270                         outside_flag = True
271                         break
272                 del A, b, inter
273                 if (outside_flag == False):
274                     inside_bound.append(ipt)
275         inside_bound = list(set(inside_bound) - set(bd_ids))
276         return inside_bound
277
278
279 def labelSeparatedVortex(path, step, anchor_pts = [0, 7200]):
280
281     print("LABELING ", os.path.join(path, 'directededge_'+step))
282     edges = np.array(pd.read_csv(os.path.join(path, 'directededge_'+step
    ), usecols = ['x', 'y'],
283                                  skipinitialspace=True, delimiter=",",
    dtype=np.int64))
284     flux = np.array(pd.read_csv(os.path.join(path, 'directededge_'+step)
    , usecols = ['massflux'],
285                                 skipinitialspace=True, delimiter=",",
    dtype = np.float64))
```

```
286     coord = np.array(pd.read_csv(os.path.join(path, 'nodes_'+step),
    usecols = ['x', 'y'],
287                                  skipinitialspace=True, delimiter=",",
    dtype = np.float64))
288
289     roi = [i for i, item in enumerate(coord) if item[0] < 0.7 and item
    [1] < 0.1]
290
291     g = MyGraph(len(edges))
292
293     for i in range(len(coord)):
294         g.add_node(i, coord=coord[i])
295
296     for i, item in enumerate(edges):
297         g.add_edge(item[0], item[1], flux=abs(flux[i][0]))
298
299     g.normFlux()
300     print("num of nodes = ", g.number_of_nodes())
301
302     vortex_core = g.findVortCore(max_depth = 6)
303
304     sepCore = deepcopy(g)
305     for inode in g.nodes():
306         if (inode not in vortex_core):
307             sepCore.remove_node(inode)
308     core_list = list(nx.weakly_connected_components(sepCore))
309     print("core_list = ", core_list)
310     plotBound(coord, list(vortex_core), core_list, "figname")
311
312     bound_list = deepcopy(core_list)
313     bound_bk_list = []
314     allbound = []
315
316
317     boundnotchanged = True
318     g_nonvort = deepcopy(g)
319     while (boundnotchanged):
320         for num_core, icore in enumerate(core_list):
321             print("*** number = {}, icore = {}".format(num_core, icore))
322             print(coord[list(icore)[0]][0], coord[list(icore)[0]][1])
323             if (removed_node[num_core] == 1):
324                 startpt = twoHopsDsPt(g, icore)
325             else:
326                 startpt = random.choice(list(bound_list[num_core]))
327
328             bound = g_nonvort.findBoundary(start_node = startpt)
329             print("boundary = ", bound)
330             plotBound(coord, list(bound), core_list, "figname")
331
332             g_temp = deepcopy(g)
333             g_temp.remove_nodes_from(bound)
334             subgraphs = list(nx.weakly_connected_components(g_temp))
335
336             otherCoreInBound = False
337
338             for isubg in subgraphs:
339                 if (noPtInSet(isubg, anchor_pts) and allPtInSet(isubg.
    union(bound), icore)):
```

```
                    if (noPtInSet(isubg.union(bound), set(vortex_core)-icore) ):
                        print("rest_core = ", set(vortex_core)-icore)
                        print("## Deleting subgraph ! Deleting points number = {}##".format(len(isubg)))
                        delSubgraph(g_nonvort, isubg)
                        bound_list[num_core] = deepcopy(bound)

                        break
                    else:
                        otherCoreInBound = True
                        print("## Other core in Bound! ##")
                        break

        if (bound_list == bound_bk_list):
            boundnotchanged = False
            print("#### All bounds not changed, searching terminated!! ####")
        else:
            bound_bk_list = deepcopy(bound_list)
            print("########## Bound changed, continue searching!! #########")
        allbound = [ipt for ibound in bound_list for ipt in ibound]

    delSubgraph(g_nonvort, allbound)

    print("Search completed!!!")
    return


def createGraph(path, step, anchor_pts = [0, 7200]):

    print("LABELLING ", step)
    edges = np.array(pd.read_csv(os.path.join(path, 'directededge_'+step), usecols = ['x', 'y'],
                                 skipinitialspace=True, delimiter=",", dtype=np.int64))
    flux = np.array(pd.read_csv(os.path.join(path, 'directededge_'+step), usecols = ['massflux'],
                                skipinitialspace=True, delimiter=",", dtype = np.float64))
    coord = np.array(pd.read_csv(os.path.join(path, 'nodes_'+step), usecols = ['x', 'y'],
                                 skipinitialspace=True, delimiter=",", dtype = np.float64))

    roi = [i for i, item in enumerate(coord) if item[0] < 0.7 and item[1] < 0.1]

    g = MyGraph(len(edges))

    for i in range(len(coord)):
        g.add_node(i, coord=coord[i])

    for i, item in enumerate(edges):
        g.add_edge(item[0], item[1], flux=abs(flux[i][0]))

    g.normFlux()
```

```
    return g

def mergeNearbyCores(core_list):
    icore_center = np.zeros((len(core_list), 2), dtype = np.float64)
    icore_diam = np.zeros(len(core_list))
    for i, icore in enumerate(core_list):
        icore_coord = coord[list(icore)]
        icore_coord_min = np.min(icore_coord, axis = 0)
        icore_coord_max = np.max(icore_coord, axis = 0)
        icore_center[i] = np.mean(icore_coord, axis = 0)
        icore_diam[i] = np.max(icore_coord_max - icore_coord_min)
    #print(icore_coord_min, icore_coord_max, icore_center, icore_diam)

    merge_list = []
    for i in range(len(core_list) - 1):
        for j in range(i+1, len(core_list)):
            dist = icore_center[i] - icore_center[j]

            dist = math.sqrt(dist[0]*dist[0]+ dist[1]*dist[1])
            if (dist < icore_diam[i]+icore_diam[j]):
                merge_list.append([i, j])

    for imerge in merge_list:
        core_list[imerge[0]] = core_list[imerge[0]].union(core_list[
    imerge[1]])
        core_list[imerge[1]] = core_list[imerge[1]].union(core_list[
    imerge[0]])
    index = set(range(0, len(core_list))) - set([imerge[1] for imerge in
     merge_list])
    return [core_list[i] for i in index]


def labelIndependentVortex(g, core_list, anchor_pts = [0, 7200]):
    vortex_core = [ipt for icore in core_list for ipt in list(icore)]
    bound_list = deepcopy(core_list)

    bound_bk_list = []
    allbound = []

    removed_node = np.zeros(len(core_list), dtype= np.int32)
    otherCoreInCount = np.zeros(len(core_list), dtype= np.int32)
    maxSearchTimeReached = np.zeros(len(core_list), dtype= np.int32)

    boundnotchanged = 0
    g_nonvort = deepcopy(g)
    while (boundnotchanged<2):
        for num_core, icore in enumerate(core_list):
            if (otherCoreInCount[num_core] >3 or maxSearchTimeReached[
    num_core] == 1):
                continue

            print("*** number = {}, icore = {}".format(num_core, icore))
            print(coord[list(icore)[0]][0], coord[list(icore)[0]][1])
            if (len(bound_bk_list)<len(core_list)):
                startpt = twoHopsDsPt(g, icore)
            else:
                #print('bound_list = ', bound_list)
                #print('core_list = ', core_list)
```

```
                startpt = random.choice(list(bound_list[num_core]))

            bound = g_nonvort.findBoundary(start_node = startpt)
            plotBound(coord, list(bound), core_list, "figname")

            g_temp = deepcopy(g)
            g_temp.remove_nodes_from(bound)
            subgraphs = list(nx.weakly_connected_components(g_temp))
            if len(bound) == 0 :
                maxSearchTimeReached[num_core] = 1

            if (len(subgraphs)==1 and len(bound) > 0):    #No points
    inside the boundary
                bound_list[num_core] = deepcopy(bound)
            else:
                for isubg in subgraphs:
                    if (noPtInSet(isubg, anchor_pts)):
                        if (noPtInSet(isubg.union(bound), set(
    vortex_core)-icore) ):
                            print("rest_core = ", set(vortex_core)-icore
    )
                            print("## Deleting subgraph ! Deleting
    points number = {}##".format(len(isubg)))
                            delSubgraph(g_nonvort, isubg)
                            bound_list[num_core] = deepcopy(bound)
                            removed_node[num_core] = 1
                            otherCoreInCount[num_core] = 0
                            break
                        else:
                            otherCoreInCount[num_core] =
    otherCoreInCount[num_core] +1
                            print("## Other core in Bound! ##")
                            break

        if (bound_list == bound_bk_list):
            boundnotchanged = boundnotchanged +1
            print("#### All bounds not changed, searching terminated!!
    ####")
        else:
            bound_bk_list = deepcopy(bound_list)
            print("########## Bound changed, continue searching!!
    #########")
        allbound = [ipt for ibound in bound_list for ipt in ibound]

    delSubgraph(g_nonvort, allbound)
    print("Search completed!!!")

    g_temp = deepcopy(g)
    g_temp.remove_nodes_from(g_nonvort.nodes())
    subgraphs = list(nx.weakly_connected_components(g_temp))

    pts_idx = []
    for igraph in subgraphs:
        if len(igraph )> 8:
            for ipt in igraph:
                pts_idx.append(ipt)

    return pts_idx
```

```
492
493 def writeLabel(pts_idx, istep):
494     label = np.zeros((len(coord), 1), dtype=np.int64)
495     for i, ipt in enumerate( pts_idx):
496         label[ipt] = 1
497
498     np.savetxt('label_'+istep.split('.')[0]+'.csv', label)
499     return
500
501
502 def writePtsFile(pts_idx, istep):
503     pts = np.ones((len(pts_idx), 7))*255
504     for i, ipt in enumerate( pts_idx):
505         pts[i][0] = coord[ipt][0]
506         pts[i][1] = coord[ipt][1]
507         pts[i][2] = 0.001
508
509     np.savetxt('label_'+istep.split('.')[0]+'.pts', pts)
510     f = open('label_'+istep.split('.')[0]+'.pts', 'r+')
511     content = f.read()
512     f.write(str(len(pts_idx))+'\n'+content)
513     f.close()
514     return
515
516 def labelVortex(path, istep, anchor_pts):
517     graph = createGraph(path, istep, anchor_pts = anchor_pts)
518
519     roi = [i for i, icoord in enumerate(coord) if icoord[0] < 0.7 and
    icoord[1] < 0.1]
520     core_list = graph.findVortCore(roi, max_depth = 5)
521
522     core_list = mergeNearbyCores(core_list)
523     pts_idx = labelIndependentVortex(graph, core_list, anchor_pts)
524     writeLabel(pts_idx, istep)
525
526     figname = 'label'+istep.split('.')[0]+'.png'
527     plotBound(coord, pts_idx, core_list, figname)
528
529 if __name__ == "__main__":
530     path = "../BFS/rij_e/RESU/unstructured_yplus_129830_restart/"
531
532     files = [file for file in os.listdir(path) if file.endswith('0.csv')
     and file.startswith('nodes_')]
533     files.sort(reverse=False)
534
535     step = [step.split('_')[1] for step in files]
536
537     labelled_steps = [file for file in os.listdir('./') if file.endswith
    ('0.csv') and file.startswith('label_')]
538     labelled_steps = [labelled.split('_')[1] for labelled in
    labelled_steps]
539
540     step = list(set(step) - set(labelled_steps))
541     step.sort()
542
543
544     print(len(step), 'steps to be labelled')
545     print("step = ", step)
```

```
546
547     coord = np.array(pd.read_csv(os.path.join(path, 'nodes_10050.csv'),
        usecols = ['x', 'y'],
548                                  skipinitialspace=True, delimiter=","))
549
550     anchor_pts = [ipt for ipt, icoord in enumerate(coord) if icoord[1] >
        0.2]
551
552     Parallel(n_jobs=30)(delayed(labelVortex)(path, istep, anchor_pts)
        for istep in step)
```

Code Listing A.2: preDataset.py

```
1  import os
2  import numpy as np
3  import pandas as pd
4  import progressbar
5  from time import sleep
6
7  import torch
8  from params import net_params, input_path
9
10 def load_dataset(files, root):
11     nodepath = root + "graph_data/nodes_"
12     labelpath = root + "label/label_"
13     dataset = dict()
14
15     bar = progressbar.ProgressBar(maxval=len(files),
16                                   widgets=[progressbar.Bar('=', '[', ']'
       ), ' ', progressbar.Percentage()])
17     bar.start()
18     for i, ifile in enumerate(files):
19         bar.update(i+1)
20         sleep(0.1)
21         nd = pd.read_csv(nodepath+ifile+'.csv', skipinitialspace = "True
       ",
22                          usecols=['x', 'y', 'u', 'v', 'turb_inten', '
       dev_shear', 'p_grad_al_st', 'q_criteria'])
23         label = np.loadtxt(labelpath+ifile+'.csv').reshape((-1, 1))
24         label = pd.DataFrame(label, columns=['label'])
25         nd = pd.concat([nd, label], axis=1)
26
27         nd = np.array([i for i in np.array(nd.sort_values(by=['y', 'x'])
       ) if i[0]>0])
28
29         xlen, ylen = len(set(nd[:, 0])), len(set(nd[:, 1]))
30         nd = pd.DataFrame(nd, columns = ['x','y', 'u', 'v', 'turb_inten'
       , 'dev_shear', 'p_grad_al_st', 'q_criteria','label'])
31         nd = nd.sort_values(by=['x', 'y'])
32
33         coord = np.array(nd[['x','y']]).reshape((xlen, ylen, 2))
34         coord = np.moveaxis(coord, -1, 0)
35
36         label = np.array(nd[['label']]).reshape((xlen, ylen))
37
38         dataset[ifile] = {'node_feat': nd, 'label': label, 'coord':
       coord}
39     bar.finish()
40     return dataset
```

```
41
42
43 if __name__ == "__main__":
44     rootpath = "/home/h79380/test_cases/saturne_vnv/BFS/rij_e/RESU/
       structured_yplus_restart/"
45     path = rootpath + "graph_data/"
46
47     input_files = [file.split('_')[-1].split('.')[0] for file in os.
       listdir(path) if file.endswith('0.csv') and file.startswith('nodes_')
       ]
48     input_files.sort(reverse=False)
49
50     train_files = input_files[:80]
51     valid_files = input_files[80:90]
52     test_files = input_files[90:100]
53
54     torch.save(load_dataset(train_files, rootpath), input_path['train'])
55     torch.save(load_dataset(valid_files, rootpath), input_path['valid'])
56     torch.save(load_dataset(test_files, rootpath), input_path['test'])
57     print('Data loading finishied')
```

Code Listing A.3: model.py

```
1  import math
2  import numpy as np
3
4  import torch
5  import torch.nn as nn
6  from torch.nn import init
7  import torch.nn.functional as F
8  import dgl.function as fn
9
10 class FastGMMLayer(nn.Module):
11     """
12     [!] code adapted from dgl implementation of GMMConv
13     Parameters
14     ----------
15     in_dim :
16         Number of input self.hidden_dim.
17     out_dim :
18         Number of output self.hidden_dim.
19     dim :
20         Dimensionality of pseudo-coordinte.
21     kernel :
22         Number of kernels :math:`K`.
23     aggr_type :
24         Aggregator type (``sum``, ``mean``, ``max``).
25     dropout :
26         Required for dropout of output self.hidden_dim.
27     batch_norm :
28         boolean flag for batch_norm layer.
29     residual :
30         If True, use residual connection inside this layer. Default: ``
       False``.
31     bias :
32         If True, adds a learnable bias to the output. Default: ``True``.
33
34     """
35     def __init__(self, in_dim, out_dim, dim, kernel, aggr_type, dropout,
```

```
                 batch_norm, residual=False, bias=True):
        super().__init__()

        self.in_dim = in_dim
        self.out_dim = out_dim
        self.dim = dim
        self.kernel = kernel
        self.batch_norm = batch_norm
        self.residual = residual
        self.dropout = dropout

        if aggr_type == 'sum':
            self._reducer = fn.sum
        elif aggr_type == 'mean':
            self._reducer = fn.mean
        elif aggr_type == 'max':
            self._reducer = fn.max
        else:
            raise KeyError("Aggregator type {} not recognized.".format(
aggr_type))

        self.mu = torch.zeros(self.kernel, self.dim)
        for i, theta in enumerate(np.arange(self.kernel-1)*2*math.pi/(
self.kernel-1)):
            self.mu[i+1, 0] = math.cos(theta)
            self.mu[i+1, 1] = math.sin(theta)

        self.fc = nn.Linear(in_dim, kernel * out_dim, bias=False)

        self.bn_node_h = nn.BatchNorm1d(out_dim, affine=False)

        if in_dim != out_dim:
            self.residual = False

        if bias:
            self.bias = nn.Parameter(torch.Tensor(out_dim))
        else:
            self.register_buffer('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        """Reinitialize learnable parameters."""
        gain = init.calculate_gain('relu')
        init.xavier_normal_(self.fc.weight, gain=gain)
        if self.bias is not None:
            init.zeros_(self.bias.data)

    def forward(self, g, h, vector):
        h_in = h # for residual connection

        g = g.local_var()
        g.ndata['h'] = self.fc(h).view(-1, self.kernel, self.out_dim)
        E = g.number_of_edges()

        # compute gaussian weight
        gaussian = -0.5 * ((vector.view(E, 1, self.dim) -
                            self.mu.view(1, self.kernel, self.dim)) **
2)
```

```
91
92          gaussian = torch.exp(gaussian.sum(dim=-1, keepdim=True)) # (E, K
    , 1)
93
94
95          min_kernel, _ = torch.min(gaussian, 1)
96          max_kernel, _ = torch.max(gaussian, 1)
97          gaussian = (gaussian - min_kernel.view(-1, 1, 1))/(max_kernel -
    min_kernel).view(-1, 1, 1)
98
99          gaussian = gaussian**2
100
101         g.edata['w'] = gaussian
102         g.update_all(fn.u_mul_e('h', 'w', 'm'), self._reducer('m', 'h'))
103         h = g.ndata['h'].sum(1)
104
105         if self.batch_norm:
106             h = self.bn_node_h(h) # batch normalization
107
108         h = F.relu(h) # non-linear activation
109
110         if self.residual:
111             h = h_in + h # residual connection
112
113         if self.bias is not None:
114             h = h + self.bias
115
116         h = F.dropout(h, self.dropout, training=self.training)
117         return h
118
119
120 class FastGMM(torch.nn.Module):
121     def __init__(self, params):
122         super().__init__()
123         self.name = 'FastGMM'
124         self.depth = len(params['hidden_dim'])
125
126         self.channel = params['hidden_dim']
127         self.kernel = params['kernel']                          # for MoNet
128         dim = params['pseudo_dim_MoNet']                        # for MoNet
129         self.n_classes = params['n_classes']
130         self.dropout = params['dropout']
131         self.batch_norm = params['batch_norm']
132         self.residual = params['residual']
133         self.device = params['device']
134         self.depth = params['depth']
135         self.aggr_type = "mean"
136
137         self.contract = nn.ModuleList()
138         for idepth in range(self.depth-1):
139             self.contract.append(FastGMMLayer(self.channel[idepth][0],
    self.channel[idepth][-1], dim, self.kernel, self.aggr_type, self.
    dropout, self.batch_norm, self.residual))
140
141         self.bottom = nn.ModuleList()
142         self.bottom.append(FastGMMLayer(self.channel[-1][0], self.
    channel[-1][-1], dim, self.kernel, self.aggr_type, self.dropout, self
    .batch_norm, self.residual))
```

```
143         self.bottom.append(FastGMMLayer(self.channel[-1][-1], self.
    channel[-1][0], dim, self.kernel, self.aggr_type, self.dropout, self.
    batch_norm, self.residual))
144 
145         self.expand = nn.ModuleList()
146         for idepth in range(self.depth-1):
147             if idepth == 0:
148                 self.expand.append(FastGMMLayer(self.channel[idepth
    ][-1]*2, self.channel[idepth][-1],dim, self.kernel, self.aggr_type,
    self.dropout, self.batch_norm, self.residual))
149             else:
150                 self.expand.append(FastGMMLayer(self.channel[idepth
    ][-1]*2, self.channel[idepth][0], dim, self.kernel, self.aggr_type,
    self.dropout, self.batch_norm, self.residual))
151 
152         self.out = nn.Linear(self.channel[0][-1], 1, bias=True)
153 
154 
155     def forward(self, case):
156         g = case['graph']
157         coarsen = [torch.tensor(icoarsen) for icoarsen in case['coarsen'
    ]]
158         h = torch.from_numpy(case['node_feat']).float()
159 
160         h_concat_depth = []
161         for idepth in range(self.depth-1):
162             h = self.contract[idepth](g[idepth], h, g[idepth].edata['vec
    '].float())
163             h_concat_depth.append(h)
164             h = torch.scatter_reduce(h, 0, torch.matmul(coarsen[idepth
    ][:, 1:],
165                                                         torch.ones(1, h.
    shape[1], dtype=int)),
166                                      reduce="mean")   ### Downsampling
    the node features from fine graph to coarse graph
167 
168         h = self.bottom[0](g[-1], h, g[-1].edata['vec'].float())
169         h = self.bottom[1](g[-1], h, g[-1].edata['vec'].float())
170 
171         for i in range(self.depth-1):
172             idepth = self.depth - 2 - i
173             h = torch.cat((torch.index_select(h, 0,
174             coarsen[idepth][:, 1]),
175             h_concat_depth[idepth]),
176             1)
177             h = self.expand[idepth](g[idepth], h, g[idepth].edata['vec'
    ].float())
178 
179         return self.out(h)
180 
181 
182 class CNN(nn.Module):
183     def __init__(self, params):
184         super().__init__()
185         self.name = 'CNN'
186 
187         self.dim = params['dim']
188         self.channel = params['channels']
```

```
        self.depth = len(self.channel)

        self.contract = nn.ModuleList()
        for idepth in range(self.depth-1):
            self.contract.append(nn.Conv2d(self.channel[idepth][0], self.channel[idepth][-1],
                                           (3, 3), stride=(1, 1), padding=(1, 1)))

        self.bottom = nn.ModuleList()
        self.bottom.append(nn.Conv2d(self.channel[-1][0], self.channel[-1][-1], (3, 3), stride=(1, 1), padding=(1, 1)))
        self.bottom.append(nn.Conv2d(self.channel[-1][-1], self.channel[-1][0], (3, 3), stride=(1, 1), padding=(1, 1)))

        self.expand = nn.ModuleList()
        for idepth in range(self.depth-1):
            if idepth == 0:
                self.expand.append(nn.Conv2d(self.channel[idepth][-1]*2, self.channel[idepth][-1], (3, 3), stride=(1, 1), padding=(1, 1)))
            else:
                self.expand.append(nn.Conv2d(self.channel[idepth][-1]*2, self.channel[idepth][0], (3, 3), stride=(1, 1), padding=(1, 1)))

        self.out = nn.Conv2d(self.channel[0][-1], 1, (1, 1), stride=(1, 1))


    def forward(self, h):
        h_concat_depth = []
        for idepth in range(self.depth -1):
            h = F.relu(self.contract[idepth](h))
            h_concat_depth.append(h)
            h = F.avg_pool2d(h, (2, 2), stride = (2, 2))

        h = F.relu(self.bottom[0](h))
        h = F.relu(self.bottom[1](h))

        for i in range(self.depth-1):

            order = self.depth - 2 - i
            idepth = self.depth - 2 - i
            h = F.interpolate(h,
                              size = (int(self.dim[0]/pow(2, order)), int(self.dim[-1]/pow(2, order))),
                              mode='nearest')
            h = torch.cat((h_concat_depth[idepth], h), 1)
            h = F.relu(self.expand[idepth](h))

        return self.out(h)

from BsplineInterpolate import BSpline
class SplineLayer(nn.Module):
    """
    [!] code adapted from dgl implementation of GMMConv
    Parameters
    ----------
    in_dim :
```

```
239         Number of input self.hidden_dim.
240     out_dim :
241         Number of output self.hidden_dim.
242     dim :
243         Dimensionality of pseudo-coordinte.
244     kernel :
245         Number of kernels :math:`K`.
246     aggr_type :
247         Aggregator type (``sum``, ``mean``, ``max``).
248     dropout :
249         Required for dropout of output self.hidden_dim.
250     batch_norm :
251         boolean flag for batch_norm layer.
252     residual :
253         If True, use residual connection inside this layer. Default: ``
    False``.
254     bias :
255         If True, adds a learnable bias to the output. Default: ``True``.
256 
257     """
258     def __init__(self, in_dim, out_dim, dim, kernel, aggr_type, dropout,
259                  batch_norm, residual=False, bias=True):
260         super().__init__()
261 
262         self.in_dim = in_dim
263         self.out_dim = out_dim
264         self.dim = dim
265         self.kernel = kernel
266         self.batch_norm = batch_norm
267         self.residual = residual
268         self.dropout = dropout
269 
270 
271         if aggr_type == 'sum':
272             self._reducer = fn.sum
273         elif aggr_type == 'mean':
274             self._reducer = fn.mean
275         elif aggr_type == 'max':
276             self._reducer = fn.max
277         else:
278             raise KeyError("Aggregator type {} not recognized.".format(
    aggr_type))
279 
280         self.weight = nn.Parameter(torch.Tensor(self.kernel, out_dim,
    in_dim))
281         self.bn_node_h = nn.BatchNorm1d(out_dim, affine=False)
282 
283         if in_dim != out_dim:
284             self.residual = False
285 
286         if bias:
287             self.bias = nn.Parameter(torch.Tensor(out_dim))
288         else:
289             self.register_buffer('bias', None)
290         self.reset_parameters()
291 
292 
293     def reset_parameters(self):
```

```
        """Reinitialize learnable parameters."""
        gain = init.calculate_gain('relu')
        init.xavier_normal_(self.weight, gain=gain)

        if self.bias is not None:
            init.zeros_(self.bias.data)

    def forward(self, g, h):
        h_in = h # for residual connection
        g.ndata['h'] = h.view(-1, self.in_dim)
        E = g.number_of_edges()


        g.apply_edges(lambda edges: {'w': torch.matmul(edges.data['BFs'
    ].view(-1, self.kernel),
                                                       self.weight.view(
    self.kernel, -1)).view(-1, self.out_dim, self.in_dim)})
        g.update_all(fn.u_dot_e('h', 'w', 'm'), fn.sum('m', 'h'))
        h = g.ndata['h'].squeeze()

        if self.batch_norm:
            h = self.bn_node_h(h) # batch normalization

        h = F.relu(h) # non-linear activation

        if self.residual:
            h = h_in + h # residual connection

        if self.bias is not None:
            h = h + self.bias

        h = F.dropout(h, self.dropout, training=self.training)
        return h


class SplineCNN(torch.nn.Module):
    def __init__(self, params):
        super().__init__()
        self.name = 'SplineCNN'
        self.depth = len(params['hidden_dim'])

        self.channel = params['hidden_dim']
        self.kernel = params['kernel']                          # for MoNet
        dim = params['pseudo_dim_MoNet']                        # for MoNet
        self.n_classes = params['n_classes']
        self.dropout = params['dropout']
        self.batch_norm = params['batch_norm']
        self.residual = params['residual']
        self.device = params['device']
        self.depth = params['depth']
        self.aggr_type = "sum"
        self.BSpline = BSpline(a = -1, b = 1, deg = 2, kts = 3)
        self.contract = nn.ModuleList()
        for idepth in range(self.depth-1):
            self.contract.append(SplineLayer(self.channel[idepth][0],
    self.channel[idepth][-1], dim, self.kernel,
                                             self.aggr_type, self.
    dropout, self.batch_norm, self.residual))
```

```
        self.bottom = nn.ModuleList()
        self.bottom.append(SplineLayer(self.channel[-1][0], self.channel
[-1][-1], dim, self.kernel,
                                       self.aggr_type, self.dropout,
self.batch_norm, self.residual))
        self.bottom.append(SplineLayer(self.channel[-1][-1], self.
channel[-1][0], dim, self.kernel,
                                       self.aggr_type, self.dropout,
self.batch_norm, self.residual))

        self.expand = nn.ModuleList()
        for idepth in range(self.depth-1):
            if idepth == 0:
                self.expand.append(SplineLayer(self.channel[idepth
][-1]*2, self.channel[idepth][-1], dim, self.kernel,
                                               self.aggr_type, self.
dropout, self.batch_norm, self.residual))
            else:
                self.expand.append(SplineLayer(self.channel[idepth
][-1]*2, self.channel[idepth][0], dim, self.kernel,
                                               self.aggr_type, self.
dropout, self.batch_norm, self.residual))

        self.out = nn.Linear(self.channel[0][-1], 1, bias=True)


    def forward(self, case):
        g = case['graph']
        coarsen = [torch.tensor(icoarsen) for icoarsen in case['coarsen'
]]
        h = torch.from_numpy(case['node_feat']).float()

        h_concat_depth = []
        for igraph in g:
            igraph.apply_edges(lambda edges: {'BFs': self.BSpline.
Interpolate(edges.data['vec'][:, 0], edges.data['vec'][:, 1]).float()
})
        for idepth in range(self.depth-1):
            h = self.contract[idepth](g[idepth], h)
            h_concat_depth.append(h)
            h = torch.scatter_reduce(h, 0, torch.matmul(coarsen[idepth
][:, 1:],
                                                        torch.ones(1, h.
shape[1], dtype=int)),
                                     reduce="mean")   ### Downsampling
the node features from fine graph to coarse graph

        h = self.bottom[0](g[-1], h)
        h = self.bottom[1](g[-1], h)

        for i in range(self.depth-1):
            idepth = self.depth - 2 - i
            h = torch.cat((torch.index_select(h, 0,
            coarsen[idepth][:, 1]),
            h_concat_depth[idepth]),
            1)
            h = self.expand[idepth](g[idepth], h)

```

```
        return self.out(h)

"""
    GMM: Gaussian Mixture Model Convolution layer
    Geometric Deep Learning on Graphs and Manifolds using Mixture Model CNNs (Federico Monti et al., CVPR 2017)
    https://arxiv.org/pdf/1611.08402.pdf
"""
class GMMLayer(nn.Module):
    """
    [!] code adapted from dgl implementation of GMMConv
    Parameters
    ----------
    in_dim :
        Number of input self.hidden_dim.
    out_dim :
        Number of output self.hidden_dim.
    dim :
        Dimensionality of pseudo-coordinte.
    kernel :
        Number of kernels :math:`K`.
    aggr_type :
        Aggregator type (``sum``, ``mean``, ``max``).
    dropout :
        Required for dropout of output self.hidden_dim.
    batch_norm :
        boolean flag for batch_norm layer.
    residual :
        If True, use residual connection inside this layer. Default: ``False``.
    bias :
        If True, adds a learnable bias to the output. Default: ``True``.

    """
    def __init__(self, in_dim, out_dim, dim, kernel, aggr_type, dropout,
                 batch_norm, residual=False, bias=True):
        super().__init__()

        self.in_dim = in_dim
        self.out_dim = out_dim
        self.dim = dim
        self.kernel = kernel
        self.batch_norm = batch_norm
        self.residual = residual
        self.dropout = dropout

        if aggr_type == 'sum':
            self._reducer = fn.sum
        elif aggr_type == 'mean':
            self._reducer = fn.mean
        elif aggr_type == 'max':
            self._reducer = fn.max
        else:
            raise KeyError("Aggregator type {} not recognized.".format(aggr_type))

        self.mu = nn.Parameter(torch.Tensor(kernel, dim))
        self.inv_sigma = nn.Parameter(torch.Tensor(kernel, dim))
```

```
        self.fc = nn.Linear(in_dim, kernel * out_dim, bias=False)

        self.bn_node_h = nn.BatchNorm1d(out_dim, affine=False)

        if in_dim != out_dim:
            self.residual = False

        if bias:
            self.bias = nn.Parameter(torch.Tensor(out_dim))
        else:
            self.register_buffer('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        """Reinitialize learnable parameters."""
        gain = init.calculate_gain('relu')
        init.xavier_normal_(self.fc.weight, gain=gain)
        init.normal_(self.mu.data, 0, 0.1)
        init.constant_(self.inv_sigma.data, 1)
        if self.bias is not None:
            init.zeros_(self.bias.data)

    def forward(self, g, h, vector):
        h_in = h # for residual connection
        g = g.local_var()
        g.ndata['h'] = self.fc(h).view(-1, self.kernel, self.out_dim)
        E = g.number_of_edges()

        # compute gaussian weight
        gaussian = -0.5 * ((vector.view(E, 1, self.dim) -
                            self.mu.view(1, self.kernel, self.dim)) **
    2)
        gaussian = gaussian * (self.inv_sigma.view(1, self.kernel, self.
    dim) ** 2)
        gaussian = torch.exp(gaussian.sum(dim=-1, keepdim=True)) # (E, K
    , 1)
        g.edata['w'] = gaussian
        g.update_all(fn.u_mul_e('h', 'w', 'm'), self._reducer('m', 'h'))
        h = g.ndata['h'].sum(1)

        if self.batch_norm:
            h = self.bn_node_h(h) # batch normalization

        h = F.relu(h) # non-linear activation

        if self.residual:
            h = h_in + h # residual connection

        if self.bias is not None:
            h = h + self.bias

        h = F.dropout(h, self.dropout, training=self.training)
        return h


class GMM(torch.nn.Module):
    def __init__(self, params):
        super().__init__()
```

```
502         self.name = 'GMM'
503         self.depth = len(params['hidden_dim'])
504 
505         self.channel = params['hidden_dim']
506         self.kernel = params['kernel']                          # for MoNet
507         dim = params['pseudo_dim_MoNet']                        # for MoNet
508         self.n_classes = params['n_classes']
509         self.dropout = params['dropout']
510         self.batch_norm = params['batch_norm']
511         self.residual = params['residual']
512         self.device = params['device']
513         self.depth = params['depth']
514         self.aggr_type = "sum"
515 
516         self.contract = nn.ModuleList()
517         for idepth in range(self.depth-1):
518             self.contract.append(GMMLayer(self.channel[idepth][0], self.
    channel[idepth][-1], dim, self.kernel, self.aggr_type, self.dropout,
    self.batch_norm, self.residual))
519 
520         self.bottom = nn.ModuleList()
521         self.bottom.append(GMMLayer(self.channel[-1][0], self.channel
    [-1][-1], dim, self.kernel, self.aggr_type, self.dropout, self.
    batch_norm, self.residual))
522         self.bottom.append(GMMLayer(self.channel[-1][-1], self.channel
    [-1][0], dim, self.kernel, self.aggr_type, self.dropout, self.
    batch_norm, self.residual))
523 
524         self.expand = nn.ModuleList()
525         for idepth in range(self.depth-1):
526             if idepth == 0:
527                 self.expand.append(GMMLayer(self.channel[idepth][-1]*2,
    self.channel[idepth][-1],dim, self.kernel, self.aggr_type, self.
    dropout, self.batch_norm, self.residual))
528             else:
529                 self.expand.append(GMMLayer(self.channel[idepth][-1]*2,
    self.channel[idepth][0], dim, self.kernel, self.aggr_type, self.
    dropout, self.batch_norm, self.residual))
530 
531         self.out = nn.Linear(self.channel[0][-1], 1, bias=True)
532 
533 
534     def forward(self, case):
535         g = case['graph']
536         coarsen = [torch.tensor(icoarsen) for icoarsen in case['coarsen'
    ]]
537         h = torch.from_numpy(case['node_feat']).float()
538 
539         h_concat_depth = []
540         for idepth in range(self.depth-1):
541             h = self.contract[idepth](g[idepth], h, g[idepth].edata['vec
    '].float())
542             h_concat_depth.append(h)
543             h = torch.scatter_reduce(h, 0, torch.matmul(coarsen[idepth
    ][:, 1:],
544                                                         torch.ones(1, h.
    shape[1], dtype=int)),
545                                      reduce="mean")    ### Downsampling
```

```
    the node features from fine graph to coarse graph

        h = self.bottom[0](g[-1], h, g[-1].edata['vec'].float())
        h = self.bottom[1](g[-1], h, g[-1].edata['vec'].float())

        for i in range(self.depth-1):
            idepth = self.depth - 2 - i
            h = torch.cat((torch.index_select(h, 0,
            coarsen[idepth][:, 1]),
            h_concat_depth[idepth]),
            1)
            h = self.expand[idepth](g[idepth], h, g[idepth].edata['vec'
    ].float())

        return self.out(h)

from dgl.nn import GraphConv
"""
    GCN: Graph Convolutional Networks
    Thomas N. Kipf, Max Welling, Semi-Supervised Classification with
    Graph Convolutional Networks (ICLR 2017)
    http://arxiv.org/abs/1609.02907
"""
class GCNLayer(nn.Module):
    def __init__(self, in_dim, out_dim):
        super().__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.conv = GraphConv(self.in_dim, self.out_dim, weight=True,
    bias=True, activation=None, allow_zero_in_degree=False)
        self.act = nn.ReLU()

    def forward(self, g, h):
        h = self.conv(g, h)
        return self.act(h)

class GCN(nn.Module):
    def __init__(self, params):
        super().__init__()
        self.name = 'GCN'
        self.depth = len(params['hidden_dim'])

        self.channel = params['hidden_dim']
        self.kernel = params['kernel']                          # for MoNet
        dim = params['pseudo_dim_MoNet']                        # for MoNet
        self.n_classes = params['n_classes']
        self.dropout = params['dropout']
        self.batch_norm = params['batch_norm']
        self.residual = params['residual']
        self.device = params['device']
        self.depth = params['depth']
        self.aggr_type = "sum"
        self.contract = nn.ModuleList()
        for idepth in range(self.depth-1):
            self.contract.append(GCNLayer(self.channel[idepth][0], self.
    channel[idepth][-1]))
        self.bottom = nn.ModuleList()
        self.bottom.append(GCNLayer(self.channel[-1][0], self.channel
```

```
    [-1][-1]))
599         self.bottom.append(GCNLayer(self.channel[-1][-1], self.channel
    [-1][0]))
600 
601         self.expand = nn.ModuleList()
602         for idepth in range(self.depth-1):
603             if idepth == 0:
604                 self.expand.append(GCNLayer(self.channel[idepth][-1]*2,
    self.channel[idepth][-1]))
605             else:
606                 self.expand.append(GCNLayer(self.channel[idepth][-1]*2,
    self.channel[idepth][0]))
607         self.out = nn.Linear(self.channel[0][-1], 1, bias=True)
608 
609 
610     def forward(self, case):
611         g = case['graph']
612         coarsen = [torch.tensor(icoarsen) for icoarsen in case['coarsen'
    ]]
613         h = torch.from_numpy(case['node_feat']).float()
614 
615         h_concat_depth = []
616         for idepth in range(self.depth-1):
617             h = self.contract[idepth](g[idepth], h)
618             h_concat_depth.append(h)
619             h = torch.scatter_reduce(h, 0,
620             torch.matmul(coarsen[idepth][:, 1:],
621             torch.ones(1, h.shape[1], dtype=int)),
622             reduce="mean")   ### Downsampling the node features from
    fine graph to coarse graph
623 
624         h = self.bottom[0](g[-1], h)
625         h = self.bottom[1](g[-1], h)
626 
627         for i in range(self.depth-1):
628             idepth = self.depth - 2 - i
629             h = torch.cat((torch.index_select(h, 0,
630             coarsen[idepth][:, 1]),
631             h_concat_depth[idepth]),
632             1)
633             h = self.expand[idepth](g[idepth], h)
634 
635         return self.out(h)
```

Code Listing A.4: train.py

```
1  import os
2  import time
3  import numpy as np
4  import pandas as pd
5  import progressbar
6  from joblib import Parallel, delayed
7  from copy import deepcopy
8 
9  import torch
10 import torch.nn.functional as F
11 from sklearn.metrics import roc_auc_score
12 
13 from utils import *
```

```
from params import input_path, net_params
from model import CNN, SplineCNN, GMM, GCN, FastGMM

def train(net_params, model_name, train_no, out_path, train_input, valid_input, test_input, epoch):
    model = modelSelection(net_params, model_name)
    optimizer = torch.optim.Adam(model.parameters(), lr = 0.001, weight_decay=5e-4)
    optimizer.zero_grad()

    begin_time = time.time()
    batch_size = 5

    history = np.zeros((epoch+1, 3))
    print('{:^5s}, {:^10s}, {:^10s}, {:^10s}'.format('Epoch', 'Train loss', 'Valid loss','Time'))

    for iepoch in range(epoch):
        model.eval()
        valid_loss = loss_eval(valid_input, model)
        train_loss_epoch = np.zeros(len(train_input))
        model.train()
        tt = time.time()

        for idx, idata in enumerate(train_input):
            #bar.update(idx+1)
            #time.sleep(0.1)
            out = model(train_input[str(idata)])
            loss = F.binary_cross_entropy(torch.sigmoid(out).reshape(-1, 1),torch.from_numpy(train_input[str(idata)]['label'].reshape(-1, 1)).float())

            train_loss_epoch[idx] = loss

            loss = loss/batch_size
            loss.backward()
            if ((idx+1)%batch_size ==0 or batch_size+1 == len(train_input)):
                optimizer.step()
                optimizer.zero_grad()

        model.eval()
        history[iepoch] = [np.mean(train_loss_epoch), valid_loss, time.time()-tt]
        print('{:5d}, {:10.5f}, {:10.5f}, {:10.5f}'
              .format(iepoch, history[iepoch][0], history[iepoch][1], history[iepoch][2]))

        np.savetxt(out_path+'/history'+str(train_no)+'.csv', history, delimiter=',')
        if (iepoch%20==0 or iepoch == epoch-1):
            torch.save({'epoch': iepoch,
                        'model_state_dict': model.state_dict(),
                        'optimizer_state_dict': optimizer.state_dict(),
                        'loss': loss,},
                       out_path+"/best_model_Train"+str(train_no))

    history[iepoch+1] = [loss_eval(train_input, model), loss_eval(
```

```
    valid_input, model), time.time()-tt]
63      print('{:5d}, {:10.5f}, {:10.5f}, {:10.5f}'
64            .format(iepoch+1, history[iepoch+1][0], history[iepoch+1][1],
    history[iepoch+1][2]))
65
66      history = pd.DataFrame(history, columns=['Train_loss', 'Valid_loss',
    'Time'])
67      history.to_csv(out_path+'/history'+str(train_no)+'.csv')
68
69      print('Total_time = ', time.time() - begin_time)
70
71  if __name__ == "__main__":
72      train_input_all = torch.load(input_path['train'])
73      valid_input_all = torch.load(input_path['valid'])
74      test_input_all = torch.load(input_path['test'])
75      train_input = {icase: train_input_all[icase] for i, icase in
    enumerate(train_input_all) if i < 80}
76      valid_input = {icase: valid_input_all[icase] for i, icase in
    enumerate(valid_input_all) if i < 10}
77      test_input = {icase: test_input_all[icase] for i, icase in enumerate
    (test_input_all) if i < 10}
78
79      models = ['FastGMM', 'GMM', 'SplineCNN', 'GCN']
80
81      num_training, epoch = (5, 100)
82
83      for imodel in models:
84          out_path = imodel
85          if not os.path.exists(out_path):
86              os.mkdir(out_path)
87
88          file = open(out_path+'/log.txt', 'w')
89
90          print("{} training begun".format(imodel))
91          t_start = time.time()
92          Parallel(n_jobs=num_training)(delayed(train)(net_params, imodel,
    itrain, out_path,train_input, valid_input, test_input, epoch) for
    itrain in range(num_training))
93          print("{} training finished".format(imodel))
94
95          file.write('\nTotal_time = {}\n'.format(time.time()-t_start))
96          logTrainInfo(net_params, imodel, out_path, file, test_input)
97          file.close()
98      print("All trainings finished")
```

Code Listing A.5: param.py

```
1   net_params = {
2       "depth": 4,
3       "hidden_dim": [[  2,     8],
4                      [  8,    16],
5                      [ 16,    32],
6                      [ 32,    64]],
7       "kernel": 9,
8       "pseudo_dim_MoNet": 2,
9       "n_classes": 1,
10      "dropout": 0.0,
11      "batch_norm": True,
12      "residual": False,
```

```
    "readout": "mean",
    "device": "cpu"}

input_path = {
    'train': '../train_input_AMG_unstructureBFS_uvnorm.pt',
    'valid': '../valid_input_AMG_unstructureBFS_uvnorm.pt',
    'test': '../test_input_AMG_unstructureBFS_uvnorm.pt',
    'unseen': '../unseen_input_AMG_unstructureBFS_uvnorm.pt'
}
```

Code Listing A.6: utils.py

```
import os
import time
import math
import json
import numpy as np
import pandas as pd
from copy import deepcopy
import matplotlib.pyplot as plt

from sklearn.metrics import roc_auc_score, roc_curve, RocCurveDisplay
from sklearn.preprocessing import LabelBinarizer

import torch
import torch.nn.functional as F

from model import CNN, SplineCNN, GMM, GCN, FastGMM
from params import input_path


def logTrainInfo(net_params, model_name, out_path, file, test_input):
    model = modelSelection(net_params, model_name)

    LossPlot(out_path)
    RocPlot(model_name, out_path, net_params, test_input)
    ClassificationEvaluate(model_name, out_path, net_params, file,
    test_input)

    checkpoint = torch.load(out_path+"/best_model_Train0")
    model.load_state_dict(checkpoint['model_state_dict'])
    model_parameters = filter(lambda p: p.requires_grad, model.
    parameters())
    model_identification_plot(test_input, model, out_path,
    identification=False)

    file.write("\nTrainable parameters = {}\n".format(sum([np.prod(p.
    size()) for p in model_parameters])))
    file.write(json.dumps(net_params))
    file.write('\n'+str(model.eval()))
    return

def modelSelection(net_params, model_name):
    if model_name == 'SplineCNN':
        model = SplineCNN(net_params).to('cpu')
    if model_name == 'GMM':
        model = GMM(net_params).to('cpu')
    if model_name == 'GCN':
        model = GCN(net_params).to('cpu')
```

```
    if model_name == 'CNN':
        model = CNN(net_params).to('cpu')
    if model_name == 'FastGMM':
        model = FastGMM(net_params).to('cpu')
    return model

def loss_eval(dataset, model):
    average_loss = np.zeros(len(dataset))
    for i, idata in enumerate(dataset):
        out = model(dataset[str(idata)])
        loss = F.binary_cross_entropy(torch.sigmoid(out).reshape(-1, 1),
                                      torch.tensor(dataset[str(idata)]['label'].reshape(-1, 1)).float())
        average_loss[i] = loss
    return np.mean(average_loss)

def roc_curve(dataset, model, epoch):
    y_truth = []
    y_pred = []

    mean_fpr = np.linspace(0, 1, epoch)
    for idx, casename in enumerate(dataset):
        idata = dataset[casename]
        pred = torch.sigmoid(model(idata)).detach().numpy().squeeze()

        truth = idata['label'].reshape(-1)
        try:
            coord = idata['coord']
        except ValueError:
            coord = idata['graph'][0]['coord']

        for i in range(len(truth)):
            if (coord[i][0] >0.0 and coord[i][0]<0.8 and coord[i][1]>0.0 and coord[i][1]<0.1):
                y_truth.append(truth[i])
                y_pred.append(pred[i])

    viz = RocCurveDisplay.from_predictions(np.array(y_truth), np.array(y_pred))
    interp_tpr = np.interp(mean_fpr, viz.fpr, viz.tpr)
    interp_tpr[0] = 0.0
    return interp_tpr, viz.roc_auc


def auc_eval(dataset, model):
    average_auc = np.zeros(len(dataset))
    for i, casename in enumerate(dataset):
        idata = dataset[casename]
        pred = torch.sigmoid(model(idata)).detach().numpy().squeeze()
        average_auc[i] = roc_auc_score(np.array(idata['label']).astype(int), pred)
    return np.mean(average_auc)


def metrics(dataset, model):
    metrics = np.zeros((len(dataset), 4))
    for idx, casename in enumerate(dataset):
        idata = dataset[casename]
```

```
        pred = torch.sigmoid(model(idata)).detach().numpy().squeeze()

        truth = idata['label'].reshape(-1)
        try:
            coord = idata['coord']
        except ValueError:
            coord = idata['graph'][0]['coord']

        threshold = 0.5
        TP = TN = FP = FN = 0
        for i in range(len(truth)):
            if (coord[i][0] >0.0 and coord[i][0]<0.8 and coord[i][1]
    >0.0 and coord[i][1]<0.1):
                if (pred[i]>threshold and truth[i]>threshold):
                    TP = TP+1
                if (pred[i]<threshold and truth[i]<threshold):
                    TN = TN+1
                if (pred[i]>threshold and truth[i]<threshold):
                    FP = FP+1
                if (pred[i]<threshold and truth[i]>threshold):
                    FN = FN+1
        metrics[idx] = [TP, TN, FP, FN]
    metrics = np.sum(metrics, axis=0)
    accuracy = (metrics[0] + metrics[1])/np.sum(metrics)
    precision = metrics[0]/(metrics[0]+metrics[2])
    recall = metrics[0]/(metrics[0]+metrics[3])
    F1 = 2*precision*recall/(precision+recall)
    return accuracy, precision, recall, F1

#### Loss history plot ####
def LossPlot(out_path):
    epoch = torch.load(out_path+"/best_model_Train0")['epoch']+1

    hist = np.zeros((5, epoch+1, 2))

    for i in range(5):
        f = open(out_path+'/history'+str(i)+'.csv', "r")
        if ('loss' in f.readline()):
            hist[i] = np.array(pd.read_csv(out_path+'/history'+str(i)+'.
    csv', skipinitialspace=True,
                                           usecols=['Train_loss','Valid_loss'])
    )
        else:
            hist[i] = np.array(pd.read_csv(out_path+'/history'+str(i)+'.
    csv', skipinitialspace=True))[:epoch+1,:2]

    fig, ax = plt.subplots()
    ax.set_xlim([0, epoch])
    #ax.set_ylim([0, 0.1])
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')

    labels = ['Training loss', 'Validation loss']
    for i in range(2):
        plt.fill_between(x=np.arange(hist.shape[1]-1),
                         y1=np.mean(hist[:,1:,i], axis = 0) - np.std(
    hist[:,1:,i], axis=0),
                         y2=np.mean(hist[:,1:,i], axis = 0) + np.std(
```

```
    hist[:,1:,i], axis=0),
                            alpha=0.25)
        plt.plot(np.arange(len(np.mean(hist[:,1:,i], axis=0))), np.mean(
    hist[:,1:,i], axis=0),
                 label = labels[i])
    plt.legend()
    plt.savefig(out_path+'/Training loss history.png')
    plt.show()

    fig, ax = plt.subplots()
    ax.set_xlim([0, epoch])
    ax.set_ylim([0., 1.0])
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')
    return


#### Classification evaluation ####
def ClassificationEvaluate(model_name, out_path, net_params, file,
    test_input):
    model = modelSelection(net_params, model_name)

    file.write("{:^8s}, {:^12s}, {:^12s}, {:^12s}, {:^12s}\n".format("
    Train", "Accuracy",    "Precision",    "Recall",    "F1"))
    eval_metrics = np.zeros((5, 4))
    for i in range(5):
        checkpoint = torch.load(out_path+"/best_model_Train"+str(i))
        model.load_state_dict(checkpoint['model_state_dict'])

        acc, pre, rec, F1 = metrics(test_input, model)
        eval_metrics[i, :] = (acc, pre, rec, F1)

        file.write("{:^8d}, {:^12.4f}, {:^12.4f}, {:^12.4f}, {:^12.4f}\n
    ".format(i, acc, pre, rec, F1))

    eval_metrics_mean, eval_metrics_std = (np.mean(eval_metrics, axis=0)
    , np.std(eval_metrics, axis=0))

    file.write("{:^8s}, {:>6.2f}{}{:<4.2f}, {:>6.2f}{}{:<4.2f}, {:>6.2f
    }{}{:<4.2f}, {:>6.2f}{}{:<4.2f}\n"
           .format("Summary",
                   eval_metrics_mean[0]*100, u"\u00B1", eval_metrics_std
    [0]*100,
                   eval_metrics_mean[1]*100, u"\u00B1", eval_metrics_std
    [1]*100,
                   eval_metrics_mean[2]*100, u"\u00B1", eval_metrics_std
    [2]*100,
                   eval_metrics_mean[3]*100, u"\u00B1", eval_metrics_std
    [3]*100))
    return


#### ROC Curve plot ####
def RocPlot(model_name, out_path, net_params, test_input):
    model = modelSelection(net_params, model_name)

    epoch = torch.load(out_path+"/best_model_Train0")['epoch']+1

```

```
    tprs = np.zeros((epoch, 5))
    aucs = np.zeros(5)
    for i in range(5):
        checkpoint = torch.load(out_path+"/best_model_Train"+str(i))
        model.load_state_dict(checkpoint['model_state_dict'])
        tprs[:, i], aucs[i] = roc_curve(test_input, model, epoch)

    mean_fpr = np.linspace(0, 1, epoch)

    fig, ax = plt.subplots()
    ax.plot([0, 1], [0, 1], linestyle="--", lw=2, color="r", label="Chance", alpha=0.8)

    mean_tpr = np.mean(tprs, axis=1)
    mean_tpr[-1] = 1.0
    mean_auc, std_auc = (np.mean(aucs), np.std(aucs))
    ax.plot(
        mean_fpr,
        mean_tpr,
        color="b",
        label=r"Mean ROC (AUC = %0.2f $\pm$ %0.2f)" % (mean_auc, std_auc),
        lw=2,
        alpha=0.8,
    )

    std_tpr = np.std(tprs, axis=1)
    tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
    tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
    ax.fill_between(
        mean_fpr,
        tprs_lower,
        tprs_upper,
        color="grey",
        alpha=0.2,
        label=r"$\pm$ 1 std. dev.",
    )

    ax.set(xlim=[-0.05, 1.05],
        ylim=[-0.05, 1.05])
        #title="Receiver operating characteristic example",)
    ax.set_xlabel("False positive rate")
    ax.set_ylabel("True positive rate")

    ax.legend(loc="lower right")
    plt.savefig(out_path+'/receiver operating characteristic.png')
    #plt.show()

    np.savetxt(out_path+'/mean_true_positive_rate.csv', mean_tpr)
    np.savetxt(out_path+'/standard_deviation_true_positive_rate.csv', std_tpr, delimiter=',')
    np.savetxt(out_path+'/aucs.csv', aucs, delimiter=',')
    return

def model_identification_plot(dataset, model, out_path, identification=False):
    savepath = out_path+'/predictionContour'
    if not os.path.exists(savepath):
```

```
252         os.mkdir(savepath)
253     for idx, casename in enumerate(dataset):
254         idata = dataset[casename]
255         pred = torch.sigmoid(model(idata)).detach().numpy().squeeze()
256 
257         try:
258             coord = idata['coord']
259         except ValueError:
260             coord = idata['graph'][0]['coord']
261 
262         size = 20
263         x_min,x_max,y_min,y_max  = (0.0, 0.80, 0.0, 0.120)
264         diag_size = math.sqrt(pow(y_max-y_min, 2)+pow(x_max-x_min, 2))
265         fig, ax = plt.subplots(figsize=((x_max-x_min)/diag_size* size, (
    y_max-y_min)/diag_size*size),
266                                constrained_layout=True)
267         ax.scatter(coord[:, 0], coord[:, 1], c=pred)
268 
269         title = title = '{}-Unet identification of vortex in BFS at time
    ={:.2f}s'.format(model.name, int(casename.split('.')[0])/1000)
270         ax.set_title(title)
271         ax.set_xlim([0, 0.7])
272         ax.set_ylim([0, 0.1])
273         #ax.set_xlabel('x/m')
274         #ax.set_ylabel('y/m')
275         ax.set_aspect(aspect='equal')
276 
277         plt.savefig(os.path.join(savepath, title+'.png'))
278         #plt.show()
279     return
```

B | Code_saturne code snippets

Two main functions related to the postprocessing of the mesh hierarchy and flow field in code_saturne are written in the file 'cs_user_postprocess.c' B.1. The algebraic multigrid theory for mesh coarsening used in this function is introduced in 2.4.4.

Function cs_post_first_grid_output outputs the mesh structure and flow field variables at the highest level (the original mesh).

Function cs_post_multigrid_output outputs the mesh structures starting from the second level and mesh coarsening mapping between two successive levels.

Code Listing B.1: cs_user_postprocess.c

```
#include "cs_defs.h"

/*----------------------------------------------------------------------
 * Standard C library headers
 *----------------------------------------------------------------------*/

#include "stdlib.h"
#include "string.h"

/*----------------------------------------------------------------------
 * Local headers
 *----------------------------------------------------------------------*/

#include "cs_headers.h"
#include "cs_user_post_variables.h"

static void *_mg_context = NULL;


typedef struct _cs_grid_t {

  int                 level;        /* Level in multigrid hierarchy */

  bool                conv_diff;    /* true if convection/diffusion case
    ,
                                       false otherwhise */
  bool                symmetric;    /* Symmetric matrix coefficients
                                       indicator */
  bool                use_faces;    /* True if face information is
    present */

  cs_lnum_t           db_size;      /* Block sizes for diagonal */
  cs_lnum_t           eb_size;      /* Block sizes for extra diagonal */

  cs_gnum_t           n_g_rows;     /* Global number of rows */

```

```
  cs_lnum_t           n_rows;       /* Local number of rows */
  cs_lnum_t           n_cols_ext;   /* Local number of participating
   cells
                                       (cells + ghost cells sharing a
   face) */

  cs_lnum_t           n_elts_r[2];  /* Size of array used for
   restriction
                                       operations ({n_rows, n_cols_ext}
   when
                                       no grid merging has taken place)
   */

  /* Grid hierarchy information */

  const struct _cs_grid_t  *parent; /* Pointer to parent (finer) grid */

  /* Connectivity information */

  cs_lnum_t           n_faces;      /* Local number of faces */
  const cs_lnum_2_t  *face_cell;    /* Face -> cells connectivity (1 to
   n) */
  cs_lnum_2_t        *_face_cell;   /* Face -> cells connectivity
                                       (private array) */

  /* Restriction from parent to current level */

  cs_lnum_t          *coarse_row;   /* Fine -> coarse row connectivity;
                                       size: parent n_cols_ext */
  cs_lnum_t          *coarse_face;  /* Fine -> coarse face connectivity
                                       (1 to n, signed:
                                       = 0 fine face inside coarse cell
                                       > 0 orientation same as parent
                                       < 0 orientation opposite as
   parent);
                                       size: parent n_faces */

  /* Geometric data */

  cs_real_t          relaxation;    /* P0/P1 relaxation parameter */
  const cs_real_t  *cell_cen;       /* Cell center (shared) */
  cs_real_t        *_cell_cen;      /* Cell center (private) */

  const cs_real_t  *cell_vol;       /* Cell volume (shared) */
  cs_real_t        *_cell_vol;      /* Cell volume (private) */

  const cs_real_t  *face_normal;    /* Surface normal of internal faces.
                                       (shared; L2 norm = face area) */
  cs_real_t        *_face_normal;   /* Surface normal of internal faces.
                                       (private; L2 norm = face area) */

  /* Parallel / periodic halo */

  const cs_halo_t  *halo;           /* Halo for this connectivity (
   shared) */
  cs_halo_t        *_halo;          /* Halo for this connectivity (
   private) */

```

```
84   /* Matrix-related data */
85
86   const cs_real_t  *da;              /* Diagonal (shared) */
87   cs_real_t        *_da;             /* Diagonal (private) */
88
89   const cs_real_t  *xa;              /* Extra-diagonal (shared) */
90   cs_real_t        *_xa;             /* Extra-diagonal (private) */
91   cs_real_t        *xa_conv;         /* Extra-diagonal (except level 0)
      */
92   cs_real_t        *xa_diff;         /* Extra-diagonal (except level 0)
      */
93
94   const cs_real_t  *xa0;             /* Symmetrized extra-diagonal (
     shared) */
95   cs_real_t        *_xa0;            /* Symmetrized extra-diagonal (
     private) */
96   cs_real_t        *xa0_diff;        /* Symmetrized extra-diagonal */
97
98   cs_real_t        *xa0ij;
99
100  cs_matrix_structure_t   *matrix_struct;  /* Associated matrix
     structure */
101  const cs_matrix_t       *matrix;         /* Associated matrix (shared)
       */
102  cs_matrix_t             *_matrix;        /* Associated matrix (private
     ) */
103
104
105 #if defined(HAVE_MPI)
106
107  /* Additional fields to allow merging grids */
108
109  int               merge_sub_root;    /* sub-root when merging */
110  int               merge_sub_rank;    /* sub-rank when merging
111                                          (0 <= sub-rank <
     merge_sub_size) */
112  int               merge_sub_size;    /* current number of merged ranks
113                                          for this subset */
114  int               merge_stride;      /* total number of ranks over
     which
115                                          merging occurred at previous
     levels */
116  int               next_merge_stride; /* total number of ranks over
     which
117                                          merging occurred at current
     level */
118
119  cs_lnum_t        *merge_cell_idx;    /* start cell_id for each sub-
     rank
120                                          when merge_sub_rank = 0
121                                          (size: merge_size + 1) */
122
123  int               n_ranks;           /* Number of active ranks */
124  MPI_Comm          comm;              /* Associated communicator */
125
126 #endif
127 } cs_grid_user;
128
```

```
/*----------------------------------------------------------------------*/
/*!
 * \brief Build a coarse grid based on multigrid coarsening of a
   diffusion
 *        type matrix (symmetric matrix).
 *
 * \param[in, out]  domain   pointer to a cs_domain_t structure
 *
 * \return multigrid structure.
 */
/*----------------------------------------------------------------------*/

static cs_multigrid_t *
_build_coarse_grid(cs_domain_t  *domain)
{
  const cs_mesh_t *m = domain->mesh;

  const cs_lnum_t n_cells_ext = m->n_cells_with_ghosts;
  const cs_lnum_t n_i_faces = m->n_i_faces;
  const cs_lnum_t n_b_faces = m->n_b_faces;

  const cs_mesh_quantities_t *mq = cs_glob_mesh_quantities;

  /* Build face viscosity array */

  cs_real_t   *c_visc, *i_visc, *b_visc, *rovsdt, *cofbfp;
  BFT_MALLOC(c_visc, n_cells_ext, cs_real_t);
  BFT_MALLOC(i_visc, n_i_faces, cs_real_t);
  BFT_MALLOC(b_visc, n_b_faces, cs_real_t);

  BFT_MALLOC(rovsdt, n_cells_ext, cs_real_t);
  BFT_MALLOC(cofbfp, n_b_faces, cs_real_t);

  for (cs_lnum_t cell_id = 0; cell_id < n_cells_ext; cell_id++) {
    c_visc[cell_id] = 1.;
    rovsdt[cell_id] = 1e-7;
  }
  for (cs_lnum_t face_id = 0; face_id < n_b_faces; face_id++) {
    cofbfp[face_id] = 1.;  /* 1 for Neumann, 0 for Dirichlet ? */
  }

  cs_face_viscosity(m, mq, 0, c_visc, i_visc, b_visc);

  BFT_FREE(c_visc);

  /* Build matrix coefficients */

  cs_real_t *da, *xa;
  BFT_MALLOC(da, n_cells_ext, cs_real_t);
  BFT_MALLOC(xa, n_i_faces, cs_real_t);

  cs_sym_matrix_scalar(m,
                       1,  /* idiffp */
                       1,  /* thetap */
                       cofbfp,
                       rovsdt,
                       i_visc,
                       b_visc,
```

```
                        da,
                        xa);

  printf("%g\n", da[1650]);

  /* Free memory */

  BFT_FREE(cofbfp);
  BFT_FREE(rovsdt);

  BFT_FREE(i_visc);
  BFT_FREE(b_visc);

  /* Assemble/convert matrix */

  bool symmetric = true;

  cs_matrix_t *a = cs_matrix_msr(symmetric,
                                 1,
                                 1);

  cs_matrix_set_coefficients(a,
                             symmetric,
                             1,
                             1,
                             m->n_i_faces,
                             (const cs_lnum_2_t *)(m->i_face_cells),
                             da,
                             xa);

  /* Now build multigrid structure */

  cs_multigrid_t *mg = cs_multigrid_create(CS_MULTIGRID_V_CYCLE);

  cs_multigrid_set_coarsening_options
    (mg,
     4,                            /* aggregation_limit */
     CS_GRID_COARSENING_DEFAULT,   /* coarsening_type */
     10,                            /* n_max_levels */
     50,                           /* min_g_rows */
     0.95,                         /* p0p1_relax */
     20);                          /* postprocess (default 0) */

  int verbosity = 1;

  cs_multigrid_setup(mg,
                     "Graph corsening",
                     a,
                     verbosity);

  BFT_FREE(xa);
  BFT_FREE(da);

  return mg;
}

/*----------------------------------------------------------------------*/
/*!
```

```
 * \brief This function is called at the end of each time step.
 *
 * It has a very general purpose, although it is recommended to handle
 * mainly postprocessing or data-extraction type operations.
 *
 * \param[in, out]  domain   pointer to a cs_domain_t structure
 */
/*----------------------------------------------------------------------*/

static void
_fine_to_coarse(const cs_multigrid_t  *mg,
                const cs_grid_t       *c,
                cs_lnum_t              stride,
                const cs_real_t       *f_var,
                cs_real_t             *c_var)
{
  cs_lnum_t  n_f_rows = 0, n_f_cols_ext = 0;
  cs_lnum_t  n_c_rows = 0, n_c_cols_ext = 0;
  int c_level = 0;

  /* Fine grid info */
  {
    const cs_grid_t *f = cs_multigrid_get_grid(mg, 0);

    cs_grid_get_info(f,
                     NULL,
                     NULL,
                     NULL,
                     NULL,
                     NULL, /* n_ranks */
                     &n_f_rows,
                     &n_f_cols_ext,
                     NULL,
                     NULL);
  }

  /* Coarse grid info */
  {
    cs_grid_get_info(c,
                     &c_level,
                     NULL,
                     NULL,
                     NULL,
                     NULL, /* n_ranks */
                     &n_c_rows,
                     &n_c_cols_ext,
                     NULL,
                     NULL);
  }

  /* Loop on stride (assume matrix is scalar, de-interlace
     variable */

  cs_real_t *b[2] = {NULL, NULL};
  cs_lnum_t b_size[2] = {n_f_cols_ext, 0};

  BFT_MALLOC(b[0], n_f_cols_ext, cs_real_t);

```

```
302   for (cs_lnum_t c_id = 0; c_id < stride; c_id++) {
303
304     for (cs_lnum_t i = 0; i < n_f_cols_ext; i++)
305       b[0][i] = f_var[i*stride + c_id];
306
307     /* Project one level at a time */
308
309     int i_f = 0, i_c = 1;
310
311     for (int i = 1; i < c_level; i++) {
312
313       i_f = (i-1)%2;
314       i_c = (i)%2;
315
316       const cs_grid_t *g_f = cs_multigrid_get_grid(mg, i-1);
317       const cs_grid_t *g_c = cs_multigrid_get_grid(mg, i);
318
319       cs_lnum_t n_cols_ext = 0;
320
321       cs_grid_get_info(g_c,
322                        NULL,
323                        NULL,
324                        NULL,
325                        NULL,
326                        NULL, /* n_ranks */
327                        NULL, /* n_rows */
328                        &n_cols_ext,
329                        NULL,
330                        NULL);
331
332       if (n_cols_ext > b_size[i_c]) {
333         b_size[i_c] = n_cols_ext;
334         BFT_REALLOC(b[i_c], b_size[i_c], cs_real_t);
335       }
336
337       cs_grid_restrict_row_var(g_f,
338                                g_c,
339                                b[i_f],
340                                b[i_c]);
341
342       const cs_matrix_t *a = cs_grid_get_matrix(g_c);
343       const cs_halo_t *h = cs_matrix_get_halo(a);
344
345       if (h != NULL)
346         cs_halo_sync_var(h, CS_HALO_STANDARD, b[i_c]);
347
348     }
349
350     for (cs_lnum_t i = 0; i < n_c_cols_ext; i++)
351       c_var[i*stride + c_id] = b[0][i];
352
353   }
354
355   BFT_FREE(b[1]);
356   BFT_FREE(b[0]);
357 }
358
359 void
```

```
cs_post_first_grid_output(void)
{
  //if (cat_id == CS_POST_MESH_VOLUME){
    const cs_mesh_quantities_t *mq = cs_glob_mesh_quantities;
    const cs_mesh_t *m = cs_glob_mesh;
    const cs_time_step_t *ts = cs_glob_time_step;

    const cs_real_3_t *xyz = (const cs_real_3_t *)mq->cell_cen;
    const cs_real_3_t *vel = (const cs_real_3_t *)CS_F_(vel)->val;

    const cs_lnum_2_t *edge = (const cs_lnum_2_t *)m->i_face_cells;

    cs_matrix_t *mat = cs_matrix_native(true, 1, 1);
    const cs_gnum_t *g_cell_id = cs_matrix_get_block_row_g_id(mat);

    /*
    const cs_mesh_adjacencies_t *ma = cs_glob_mesh_adjacencies;
    char adj_file[20];
    sprintf(adj_file, "c2c_rank%d.csv", cs_glob_rank_id);
    FILE *fadj = fopen(adj_file, "w");
    for (cs_lnum_t i = 0; i < m->n_cells; i ++){
      cs_lnum_t s_id = ma->cell_cells_idx[i];
      cs_lnum_t e_id = ma->cell_cells_idx[i+1];
      for (cs_lnum_t j = s_id; j < e_id; j++){
        fprintf(fadj, "%8ld %8ld\n ", g_cell_id[i], g_cell_id[ma->
   cell_cells[j]]);
      }

      cs_lnum_t s_e_id = ma->cell_cells_e_idx[i];
      cs_lnum_t e_e_id = ma->cell_cells_e_idx[i+1];
      for (cs_lnum_t j = s_e_id; j < e_e_id; j++){
        fprintf(fadj, "%8ld %8ld\n", g_cell_id[i], g_cell_id[ma->
   cell_cells_e[j]]);
      }
    }
    fclose(fadj);
    */
    cs_lnum_t *cell_list = NULL;
    BFT_MALLOC(cell_list, m->n_cells, cs_lnum_t);
    cs_lnum_t n_cells = 0;
    cs_selector_get_cell_list("all[]",
                              &n_cells,
                              cell_list);

    cs_real_t *q_norm = NULL;
    BFT_MALLOC(q_norm, n_cells, cs_real_t);
    cs_post_norm_q_criterion(q_norm);

    /* Output flow field */
    char nodefile[30];
    sprintf(nodefile, "nodes_L0_Rank%d_Ts%d.csv", cs_glob_rank_id, ts->
   nt_cur);
    FILE *fnode = fopen(nodefile, "w");
    fprintf(fnode,
    "%10s, %10s, %10s, %10s, %10s, %10s, %10s, %10s\n",
    "id",            "x",              "y",            "z",
                     "u",              "v",            "w",
                     "q_norm");
```

```
415     for (cs_lnum_t iloc = 0; iloc < n_cells; iloc++){
416         cs_lnum_t icel = cell_list[iloc];
417         if (icel < n_cells){
418           fprintf(fnode,
419           "%10ld,  %10.8f, %10.8f, %10.8f, %10.8f, %10.8f, %10.8f, %10.8
    f\n",
420           g_cell_id[icel],  xyz[icel][0], xyz[icel][1], xyz[icel][2],
421                             vel[icel][0], vel[icel][1], vel[icel][2],
422                             q_norm[icel]);
423         }
424     }
425     fclose(fnode);
426     BFT_FREE(q_norm);
427 
428     /* Output wall nodes */
429     cs_lnum_t n_b_faces = 0, *b_faces = NULL;
430     BFT_MALLOC(b_faces, m->n_cells_with_ghosts, cs_lnum_t);
431     cs_selector_get_wall_face_list(&n_b_faces,
432                                    b_faces);
433     cs_lnum_t *b_face_cell =(cs_lnum_t *) m->b_face_cells;
434     char wallfile[30];
435     sprintf(wallfile, "wallnodes_L0_Rank%d_Ts%d.csv", cs_glob_rank_id,
    ts->nt_cur);
436     FILE *fwall = fopen(wallfile, "w");
437     fprintf(fwall, "%10s\n", "id");
438     for (cs_lnum_t ifac = 0; ifac < n_b_faces; ifac++){
439       fprintf(fwall, "%10d\n", b_faces[ifac]);
440     }
441     fclose(fwall);
442     BFT_FREE(b_faces);
443 
444     /* output edge info */
445     cs_real_3_t coord_min, coord_max;
446     for (int j = 0; j < 3; j++){
447       coord_min[j] = xyz[0][j];
448       coord_max[j] = xyz[0][j];
449     }
450     for (int i = 0; i < n_cells; i++){
451       cs_lnum_t icel = cell_list[i];
452       for (int j = 0; j < 3; j++){
453         coord_min[j] = CS_MIN(coord_min[j], xyz[icel][j]);
454         coord_max[j] = CS_MAX(coord_max[j], xyz[icel][j]);
455       }
456     }
457     cs_parall_min(3, CS_REAL_TYPE, coord_min);
458     cs_parall_max(3, CS_REAL_TYPE, coord_max);
459 
460     cs_real_t dist_max = cs_math_3_distance(coord_min, coord_max)/sqrt
    (3);
461     char edgefile[30];
462     sprintf(edgefile, "edges_L0_Rank%d_Ts%d.csv", cs_glob_rank_id, ts->
    nt_cur);
463     FILE *fedge = fopen(edgefile, "w");
464     fprintf(fedge, "%8s, %8s\n", "node1", "node2");
465     for (int id = 0; id < m->n_i_faces; id++){
466       cs_lnum_t left = edge[id][0], right = edge[id][1];
467       cs_real_3_t left_coord, right_coord;
468       for (int j = 0; j<3; j++){
```

```
469         left_coord[j] = xyz[left][j];
470         right_coord[j] = xyz[right][j];
471       }
472       cs_real_t dist = cs_math_3_distance(left_coord, right_coord);
473       if (right < n_cells && dist < dist_max){
474         fprintf(fedge, "%81d, %81d\n",  g_cell_id[left], g_cell_id[right
    ]);
475       }
476     }
477     fclose(fedge);
478     BFT_FREE(cell_list);
479   //}
480 }
481
482 /*----------------------------------------------------------------
483  * Compute coarse row variable values from fine row values average by
     volume
484  *
485  * parameters:
486  *   f       <-- Fine grid structure
487  *   c       <-- Fine grid structure
488  *   f_var   <-- Variable defined on fine grid rows
489  *   c_var   --> Variable defined on coarse grid rows
490  *----------------------------------------------------------------*/
491
492 void
493 cs_grid_restrict_row_var_weighted_by_volume(const cs_grid_t  *f,
494                                             const cs_grid_t  *c,
495                                             int               stride,
496                                             const cs_real_t  *f_var,
497                                             cs_real_t        *c_var)
498 {
499   cs_lnum_t f_n_rows = f->n_rows;
500   cs_lnum_t c_n_cols_ext = c->n_elts_r[1];
501
502   const cs_real_t *f_cell_vol = f->cell_vol;
503   const cs_real_t *c_cell_vol = c->cell_vol;
504
505   const cs_lnum_t *coarse_row;
506   const cs_lnum_t db_size = stride;
507
508   assert(f != NULL);
509   assert(c != NULL);
510   assert(c->coarse_row != NULL || f_n_rows == 0);
511   assert(f_var != NULL || f_n_rows == 0);
512   assert(c_var != NULL || c_n_cols_ext == 0);
513
514   /* Set coarse values */
515
516   coarse_row = c->coarse_row;
517
518   cs_lnum_t _c_n_cols_ext = c_n_cols_ext*stride;
519
520 # pragma omp parallel for  if(_c_n_cols_ext > CS_THR_MIN)
521   for (cs_lnum_t ii = 0; ii < _c_n_cols_ext; ii++)
522     c_var[ii] = 0.;
523
524   if (db_size == 1) {
```

```
525     for (cs_lnum_t ii = 0; ii < f_n_rows; ii++) {
526       cs_lnum_t i = coarse_row[ii];
527       if (i >= 0)
528         c_var[i] += f_var[ii] * f_cell_vol[ii];
529     }
530     for (cs_lnum_t i = 0; i < c_n_cols_ext; i++) {
531       if (i >= 0)
532         for (cs_lnum_t j = 0; j < db_size; j++)
533         c_var[i] /= c_cell_vol[i];
534     }
535   }
536   else {
537     for (cs_lnum_t ii = 0; ii < f_n_rows; ii++) {
538       cs_lnum_t i = coarse_row[ii];
539       if (i >= 0) {
540         for (cs_lnum_t j = 0; j < db_size; j++)
541           c_var[i*db_size+j] += f_var[ii*db_size+j]*f_cell_vol[ii];
542       }
543     }
544     for (cs_lnum_t i = 0; i < c_n_cols_ext; i++) {
545       if (i >= 0) {
546         for (cs_lnum_t j = 0; j < db_size; j++)
547           c_var[i*db_size+j] /= c_cell_vol[i];
548       }
549     }
550   }
551
552 #if defined(HAVE_MPI)
553
554   /* If grid merging has taken place, gather coarse data */
555
556   if (c->merge_sub_size > 1) {
557
558     MPI_Comm  comm = cs_glob_mpi_comm;
559     static const int tag = 'r'+'e'+'s'+'t'+'r'+'i'+'c'+'t';
560
561     /* Append data */
562
563     if (c->merge_sub_rank == 0) {
564       int rank_id;
565       MPI_Status status;
566       assert(cs_glob_rank_id == c->merge_sub_root);
567       for (rank_id = 1; rank_id < c->merge_sub_size; rank_id++) {
568         cs_lnum_t n_recv = (  c->merge_cell_idx[rank_id+1]
569                             - c->merge_cell_idx[rank_id]);
570         int dist_rank = c->merge_sub_root + c->merge_stride*rank_id;
571         MPI_Recv(c_var + c->merge_cell_idx[rank_id]*db_size,
572                  n_recv*db_size, CS_MPI_REAL, dist_rank, tag, comm, &
    status);
573       }
574     }
575     else
576       MPI_Send(c_var, c->n_elts_r[0]*db_size, CS_MPI_REAL,
577                c->merge_sub_root, tag, comm);
578   }
579
580 #endif /* defined(HAVE_MPI) */
581 }
```

```
582
583
584 void
585 cs_post_multigrid_output(cs_domain_t     *domain)
586 {
587     cs_multigrid_t *mg = _build_coarse_grid(domain);
588     int max_level;
589     const cs_grid_user *coarest_grid = cs_multigrid_get_grid(mg, -1);
590     cs_grid_get_info(coarest_grid,
591                      &max_level,
592                      NULL,
593                      NULL,
594                      NULL,
595                      NULL, /* n_ranks */
596                      NULL,
597                      NULL,
598                      NULL,
599                      NULL);
600     cs_real_t **vel = NULL;
601     BFT_MALLOC(vel, max_level+1, cs_real_t *);
602     vel[0] = (cs_real_t *)CS_F_(vel)->val;
603     for (int ilv = 1; ilv < max_level+1; ilv++){
604       cs_grid_user *f = cs_multigrid_get_grid(mg, ilv-1);
605       cs_grid_user *c = cs_multigrid_get_grid(mg, ilv);
606
607       int stride = 3;
608       BFT_MALLOC(vel[ilv], stride*c->n_elts_r[1], cs_real_t);
609       cs_grid_restrict_row_var_weighted_by_volume(f,
610                                                   c,
611                                                   stride,
612                                                   vel[ilv-1],
613                                                   vel[ilv]);
614     }
615     for (int f_level = 0; f_level < max_level; f_level++){
616
617       const cs_grid_user *c = cs_multigrid_get_grid(mg, f_level+1);
618       const cs_grid_user *f = cs_multigrid_get_grid(mg, f_level);
619
620       cs_lnum_t n_c_rows = 0, n_c_cols_ext = 0, n_c_entries = 0;
621       cs_gnum_t n_c_g_rows = 0;
622
623       cs_grid_get_info(c,
624                        NULL,
625                        NULL,
626                        NULL,
627                        NULL,
628                        NULL, /* n_ranks */
629                        &n_c_rows,
630                        &n_c_cols_ext,
631                        &n_c_entries,
632                        &n_c_g_rows);
633
634       cs_lnum_t n_f_rows = 0, n_f_cols_ext = 0, n_f_entries = 0;
635       cs_gnum_t n_f_g_rows = 0;
636
637       cs_grid_get_info(f,
638                        NULL,
639                        NULL,
```

```
640                         NULL,
641                         NULL,
642                         NULL, /* n_ranks */
643                         &n_f_rows,
644                         &n_f_cols_ext,
645                         &n_f_entries,
646                         &n_f_g_rows);
647       cs_matrix_t *f_mat = cs_grid_get_matrix(f);
648
649       cs_gnum_t *f_g_c_id = NULL;
650       BFT_MALLOC(f_g_c_id, n_f_rows, cs_gnum_t);
651       const cs_gnum_t *f_g_cell_id = cs_matrix_get_block_row_g_id(f_mat)
    ;
652       memcpy(f_g_c_id, f_g_cell_id, n_f_rows*sizeof(cs_gnum_t));
653
654       cs_gnum_t *c_g_c_id = NULL;
655       cs_matrix_t *c_mat = cs_grid_get_matrix(c);
656       BFT_MALLOC(c_g_c_id, n_c_cols_ext, cs_gnum_t);
657       const cs_gnum_t *c_g_cell_id = cs_matrix_get_block_row_g_id(c_mat)
    ;
658       memcpy(c_g_c_id, c_g_cell_id, n_c_cols_ext*sizeof(cs_gnum_t));
659
660
661       /* Export the fine to coarse mesh correspondence */
662       char *f2cfile = NULL;
663       BFT_MALLOC(f2cfile, 40, char);
664       sprintf(f2cfile, "L%dL%d_Rank%d_Ts%d.csv",
665               f_level, f_level+1, cs_glob_rank_id, domain->time_step->
    nt_cur);
666
667       FILE *f2c = fopen(f2cfile, "w");
668       fprintf(f2c, "%8s, %8s\n", "f_node", "c_node");
669       for (int id = 0; id < n_f_rows; id++){
670         fprintf(f2c, "%8ld, %8ld\n",  f_g_c_id[id], c_g_c_id[c->
    coarse_row[id]]);
671       }
672       fclose(f2c);
673       BFT_FREE(f2cfile);
674       /* Export the fine to coarse mesh correspondence */
675
676       /* Export the cell connectivities at coarse level */
677       cs_real_3_t coord_min, coord_max;
678       for (int j = 0; j < 3; j++){
679         coord_min[j] = c->cell_cen[j];
680         coord_max[j] = c->cell_cen[j];
681       }
682       for (int irow = 0; irow < n_c_rows; irow++){
683         for (int j = 0; j < 3; j++){
684           coord_min[j] = CS_MIN(coord_min[j], c->cell_cen[irow*3+j]);
685           coord_max[j] = CS_MAX(coord_max[j], c->cell_cen[irow*3+j]);
686         }
687       }
688       cs_parall_min(3, CS_REAL_TYPE, coord_min);
689       cs_parall_max(3, CS_REAL_TYPE, coord_max);
690
691       cs_real_t dist_max = cs_math_3_distance(coord_min, coord_max)/sqrt
    (3);
692       //printf("graph level = %8d, distance = %8f\n", f_level+1,
```

```
   dist_max);
693
694        const cs_lnum_t *row_index, *col_id;
695        cs_matrix_get_msr_arrays(c_mat,
696                                 &row_index,
697                                 &col_id,
698                                 NULL,
699                                 NULL);
700        char *edge_coarse_file = NULL;
701        BFT_MALLOC(edge_coarse_file, 40, char);
702        sprintf(edge_coarse_file, "edges_L%d_Rank%d_Ts%d.csv",
703                f_level+1, cs_glob_rank_id, domain->time_step->nt_cur);
704
705        FILE *coarse_level = fopen(edge_coarse_file, "w");
706        fprintf(coarse_level, "%8s, %8s\n", "node1", "node2");
707
708        for (int irow = 0; irow < n_c_rows; irow++){
709          const cs_lnum_t *restrict col_id_irow = col_id + row_index[irow
   ];
710          const cs_lnum_t n_cols = row_index[irow+1] - row_index[irow];
711          for (int icol = 0; icol < n_cols; icol++){
712            cs_lnum_t left, right;
713            left = c_g_c_id[irow];
714            right = c_g_c_id[col_id_irow[icol]];
715            cs_real_3_t left_coord, right_coord;
716            for (int j = 0; j<3; j++){
717              left_coord[j] = c->cell_cen[left*3+j];
718              right_coord[j] = c->cell_cen[right*3+j];
719            }
720            cs_real_t dist = cs_math_3_distance(left_coord, right_coord);
721            if (left < right && dist < dist_max){
722              fprintf(coarse_level, "%8d, %8d\n", left, right);
723            }
724          }
725        }
726        fclose(coarse_level);
727        BFT_FREE(edge_coarse_file);
728        /* Export the cell connectivities at coarse level */
729
730        /* Export the cell coordinates at coarse level */
731        char *coord_filename = NULL;
732        BFT_MALLOC(coord_filename, 40, char);
733        sprintf(coord_filename, "nodes_L%d_Rank%d_Ts%d.csv",
734                f_level+1, cs_glob_rank_id, domain->time_step->nt_cur);
735
736        FILE *coord_file = fopen(coord_filename, "w");
737        fprintf(coord_file, "%8s, %12s, %12s, %12s, %12s, %12s, %12s\n",
738                            "id", "x", "y", "z", "u", "v", "w");
739        for (int irow = 0; irow < n_c_rows; irow++){
740          fprintf(coord_file, "%8ld, %12.6f, %12.6f, %12.6f, %12.6f, %12.6
   f, %12.6f \n",
741                  c_g_c_id[irow], c->cell_cen[irow*3], c->cell_cen[irow
   *3+1], c->cell_cen[irow*3+2],
742                        vel[f_level+1][irow*3], vel[f_level+1][irow*3+1],
    vel[f_level+1][irow*3+2]);
743        }
744        fclose(coord_file);
745        BFT_FREE(coord_filename);
```

```
746       /* Export the cell coordinates at coarse level */
747
748       BFT_FREE(f_g_c_id);
749       BFT_FREE(c_g_c_id);
750     }
751
752     for (int ilv = 1; ilv < max_level+1; ilv++){
753       BFT_FREE(vel[ilv]);
754     }
755     BFT_FREE(vel);
756     /*
757     cs_matrix_dump(cs_grid_get_matrix(c),
758                    "coarse_graph_matrix.bd");
759     */
760     //c = NULL;
761
762     cs_multigrid_free(mg);
763
764     _mg_context = mg;
765 }
766
767
768 /*----------------------------------------------------------------------*/
769 /*!
770  * Output the mass flux on each interior face from central cell to
       downwind cell
771  */
772 /*----------------------------------------------------------------------*/
773
774 void
775 output_directed_edges(const cs_time_step_t *ts)
776 {
777     const cs_mesh_t *m = cs_glob_mesh;
778
779     const cs_lnum_2_t *restrict i_face_cells
780       = (const cs_lnum_2_t *restrict)m->i_face_cells;
781
782     const int kimasf = cs_field_key_id("inner_mass_flux_id");
783     const cs_real_t *restrict i_massflux =
784       cs_field_by_id(cs_field_get_key_int(CS_F_(vel), kimasf))->val;
785
786     cs_matrix_t *mat = cs_matrix_native(true, 1, 1);
787     const cs_gnum_t *g_cell_id = cs_matrix_get_block_row_g_id(mat);
788
789     char *filename = NULL;
790     BFT_MALLOC(filename, 30, char);
791     sprintf(filename, "directededge_rank%d_ts%d.csv", cs_glob_rank_id,
       ts->nt_cur);
792     FILE *file = fopen(filename, "w");
793     //FILE *file = fopen("directed_edge.csv", "w");
794     fprintf(file, "%5s, %5s, %15s\n", "x", "y", "massflux");
795     for (int face_id = 0; face_id < m->n_i_faces; face_id++){
796       cs_lnum_t ic, id;
797       cs_lnum_t ii = i_face_cells[face_id][0];
798       cs_lnum_t jj = i_face_cells[face_id][1];
799       //if (ii >=m->n_cells || jj >=m->n_cells)
800       if (jj >=m->n_cells)
801         continue;
```

```
802       /* Determine central and downwind sides w.r.t. current face */
803       cs_central_downwind_cells(ii,
804                                 jj,
805                                 i_massflux[face_id],
806                                 &ic,  /* central cell id */
807                                 &id); /* downwind cell id */
808
809       fprintf(file, "%5ld, %5ld, %15.10f\n", g_cell_id[ic], g_cell_id[id
    ], i_massflux[face_id]);
810     }
811     fclose(file);
812     BFT_FREE(filename);
813 }
```

C | Publications and conferences attended

C.1 Publications

Two articles are published during this research:

1. Lianfa Wang, Yvan Fournier, Jean-François Wald, and Youssef Mesri. "A graph neural network-based framework to identify flow phenomena on unstructured meshes.". Physics of Fluids 35, no. 7 (2023).

2. Lianfa Wang, Yvan Fournier, Jean-François Wald, and Youssef Mesri. "Identification of vortex in unstructured mesh with graph neural networks.". Computers and Fluids 268 (2024): 106104.

C.2 Conferences attended

During this research, I've made oral presentations in three conferences:

1. Identifying flow characteristics with graph convolutional neural networks. 33rd Parallel CFD international conference, 25-27 May 2022, Alba, Italy.

2. A vortex identification method based on Convolutional Neural Network on unstructured mesh. Congrès Français de Mécanique 2022, 29 Aug – 02 Sep 2022, Nantes, France.

3. A GNN-based framework to identify flow phenomena on unstructured meshes. 34th Parallel CFD international conference, 2023, Online presentation.

Bibliography

[1] Ahmed, S. E. and San, O. (2020), 'Breaking the kolmogorov barrier in model reduction of fluid flows', *Fluids* **5**(1), 26.

[2] Akkari, N., Casenave, F., Hachem, E. and Ryckelynck, D. (2022), 'A bayesian nonlinear reduced order modeling using variational autoencoders', *Fluids* **7**(10).
URL: *https://www.mdpi.com/2311-5521/7/10/334*

[3] Archambeau, F., Méchitoua, N. and Sakiz, M. (2004), 'Code saturne: A finite volume code for the computation of turbulent incompressible flows-industrial applications', *International Journal on Finite Volumes* **1**(1), http–www.

[4] Arts, T., Benocci, C. and Rambaud, P. (2007), Experimental and numerical investigation of flow and heat transfer in a ribbed square duct, *in* '3rd International symposium on integrating CFD and experiments in aerodynamics', pp. 20–21.

[5] Bai, X., Wang, C. and Li, C. (2019), 'A streampath-based rcnn approach to ocean eddy detection', *IEEE Access* **7**, 106336–106345.

[6] Bao, K., Zhang, X., Peng, W. and Yao, W. (2023), 'Deep learning method for super-resolution reconstruction of the spatio-temporal flow field', *Advances in Aerodynamics* **5**(1), 1–16.

[7] Bengio, Y., Simard, P. and Frasconi, P. (1994), 'Learning long-term dependencies with gradient descent is difficult', *IEEE transactions on neural networks* **5**(2), 157–166.

[8] Berenjkoub, M., Chen, G. and Günther, T. (2020), Vortex boundary identification using convolutional neural network, *in* '2020 IEEE Visualization Conference (VIS)', IEEE, pp. 261–265.

[9] Brener, B. P., Cruz, M. A., Thompson, R. L. and Anjos, R. P. (2021), 'Conditioning and accurate solutions of reynolds average navier–stokes equations with data-driven turbulence closures', *Journal of Fluid Mechanics* **915**, A110.

[10] Buice, C. U. (1997), *Experimental investigation of flow through an asymmetric plane diffuser*, Stanford University.

[11] Cai, S., Zhou, S., Xu, C. and Gao, Q. (2019), 'Dense motion estimation of particle images via a convolutional neural network', *Experiments in Fluids* **60**, 1–16.

[12] Chen, M. C., Ball, R. L., Yang, L., Moradzadeh, N., Chapman, B. E., Larson, D. B., Langlotz, C. P., Amrhein, T. J. and Lungren, M. P. (2018), 'Deep learning to classify radiology free-text reports', *Radiology* **286**(3), 845–852.

[13] Chen, M., Wei, Z., Huang, Z., Ding, B. and Li, Y. (2020), Simple and deep graph convolutional networks, *in* 'International conference on machine learning', PMLR, pp. 1725–1735.

[14] Chong, M. S., Perry, A. E. and Cantwell, B. J. (1990), 'A general classification of three-dimensional flow fields', *Physics of Fluids A: Fluid Dynamics* **2**(5), 765–777.

[15] Cruz, M. A., Thompson, R. L., Sampaio, L. E. and Bacchi, R. D. (2019), 'The use of the reynolds force vector in a physics informed machine learning approach for predictive turbulence modeling', *Computers & Fluids* **192**, 104258.

[16] Daly, B. J. and Harlow, F. H. (1970), 'Transport equations in turbulence', *The physics of fluids* **13**(11), 2634–2649.

[17] Defferrard, M., Bresson, X. and Vandergheynst, P. (2016), 'Convolutional neural networks on graphs with fast localized spectral filtering'.

[18] Deng, L., Bao, W., Wang, Y., Yang, Z., Zhao, D., Wang, F., Bi, C. and Guo, Y. (2022), 'Vortex-u-net: An efficient and effective vortex detection approach based on u-net structure', *Applied Soft Computing* **115**, 108229.

[19] Deng, L., Chen, J., Wang, Y., Chen, X., Wang, F. and Liu, J. (2022), 'Mvu-net: a multi-view u-net architecture for weakly supervised vortex detection', *Engineering Applications of Computational Fluid Mechanics* **16**(1), 1567–1586.

[20] Deng, L., Wang, Y., Liu, Y., Wang, F., Li, S. and Liu, J. (2019), 'A cnn-based vortex identification method', *Journal of Visualization* **22**(1), 65–78.

[21] Derksen, J. (2005), 'Simulations of confined turbulent vortex flow', *Computers & Fluids* **34**(3), 301–318.

[22] Dong, C., Loy, C. C., He, K. and Tang, X. (2015), 'Image super-resolution using deep convolutional networks', *IEEE transactions on pattern analysis and machine intelligence* **38**(2), 295–307.

[23] Driss, Z., Necib, B. and Zhang, H. C. (2018), 'Cfd techniques and thermo-mechanics applications || multigrid and preconditioning techniques in cfd applications'.

[24] Fawcett, T. (2006), 'An introduction to roc analysis', *Pattern recognition letters* **27**(8), 861–874.

[25] Fey, M., Lenssen, J. E., Weichert, F. and Müller, H. (2018), Splinecnn: Fast geometric deep learning with continuous b-spline kernels, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 869–877.

[26] Frezat, H., Balarac, G., Le Sommer, J., Fablet, R. and Lguensat, R. (2021), 'Physical invariance in neural networks for subgrid-scale scalar flux modeling', *Physical Review Fluids* **6**(2), 024607.

[27] Fukami, K., Fukagata, K. and Taira, K. (2019), 'Super-resolution reconstruction of turbulent flows with machine learning', *Journal of Fluid Mechanics* **870**, 106–120.

[28] Fukami, K., Fukagata, K. and Taira, K. (2021), 'Machine-learning-based spatio-temporal super resolution reconstruction of turbulent flows', *Journal of Fluid Mechanics* **909**, A9.

[29] Fukami, K., Fukagata, K. and Taira, K. (2023), 'Super-resolution analysis via machine learning: a survey for fluid flows', *Theoretical and Computational Fluid Dynamics* pp. 1–24.

[30] Fukami, K., Hasegawa, K., Nakamura, T., Morimoto, M. and Fukagata, K. (2021), 'Model order reduction with neural networks: Application to laminar and turbulent flows', *SN Computer Science* **2**, 1–16.

[31] Fukushima, K. (1969), 'Visual feature extraction by a multilayered network of analog threshold elements', *IEEE Transactions on Systems Science and Cybernetics* **5**(4), 322–333.

[32] Glorot, X. and Bengio, Y. (2010), 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, pp. 249–256.

[33] Gruber, A., Gunzburger, M., Ju, L. and Wang, Z. (2022), 'A comparison of neural network architectures for data-driven reduced-order modeling', *Computer Methods in Applied Mechanics and Engineering* **393**, 114764.

[34] Haller, G., Hadjighasem, A., Farazmand, M. and Huhn, F. (2016), 'Defining coherent vortices objectively from the vorticity', *Journal of Fluid Mechanics* **795**, 136–173.

[35] Hanrahan, S., Kozul, M. and Sandberg, R. (2023), 'Studying turbulent flows with physics-informed neural networks and sparse data', *International Journal of Heat and Fluid Flow* **104**, 109232.
URL: *https://www.sciencedirect.com/science/article/pii/S0142727X23001315*

[36] He, K., Zhang, X., Ren, S. and Sun, J. (2016), Deep residual learning for image recognition, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 770–778.

[37] Hershey, S., Chaudhuri, S., Ellis, D. P., Gemmeke, J. F., Jansen, A., Moore, R. C., Plakal, M., Platt, D., Saurous, R. A., Seybold, B. et al. (2017), Cnn architectures for large-scale audio classification, *in* '2017 ieee international conference on acoustics, speech and signal processing (icassp)', IEEE, pp. 131–135.

[38] Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., Mohamed, S. and Lerchner, A. (2016), beta-vae: Learning basic visual concepts with a constrained variational framework, *in* 'International conference on learning representations'.

[39] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. R. (2012), 'Improving neural networks by preventing co-adaptation of feature detectors', *arXiv preprint arXiv:1207.0580* .

[40] Ho, J., Jain, A. and Abbeel, P. (2020), 'Denoising diffusion probabilistic models', *Advances in neural information processing systems* **33**, 6840–6851.

[41] Hu, B., Yin, Z., Hamrani, A., Leon, A. and McDaniel, D. (2024), 'Super-resolution-assisted rapid high-fidelity cfd modeling of data centers', *Building and Environment* **247**, 111036.

[42] Huang, J., Qiu, R., Wang, J. and Wang, Y. (2024), 'Multi-scale physics-informed neural networks for solving high reynolds number boundary layer flows based on matched asymptotic expansions', *Theoretical and Applied Mechanics Letters* **14**(2), 100496. **URL:** *https://www.sciencedirect.com/science/article/pii/S2095034924000072*

[43] Hunt, J. (1987), 'Vorticity and vortex dynamics in complex turbulent flows', *Transactions of the Canadian Society for Mechanical Engineering* **11**(1), 21–35.

[44] Jeong, J. and Hussain, F. (1995), 'On the identification of a vortex', *Journal of fluid mechanics* **285**, 69–94.

[45] Jungnickel, D. and Jungnickel, D. (2005), *Graphs, networks and algorithms*, Vol. 3, Springer.

[46] Kang, Y.-E., Yang, S. and Yee, K. (2022), 'Physics-aware reduced-order modeling of transonic flow via β-variational autoencoder', *Physics of Fluids* **34**(7).

[47] Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S. and Yang, L. (2021), 'Physics-informed machine learning', *Nature Reviews Physics* **3**(6), 422–440.

[48] Karras, T., Aila, T., Laine, S. and Lehtinen, J. (2017), 'Progressive growing of gans for improved quality, stability, and variation', *arXiv preprint arXiv:1710.10196* .

[49] Karvinen, A. and Ahlstedt, H. (2005), Comparison of turbulence models in case of jet in crossflow using commercial cfd code, *in* 'Engineering Turbulence Modelling and Experiments 6', Elsevier, pp. 399–408.

[50] Kim, B. and Günther, T. (2019), Robust reference frame extraction from unsteady 2d vector fields with convolutional neural networks, *in* 'Computer Graphics Forum', Vol. 38, Wiley Online Library, pp. 285–295.

[51] Kim, H., Kim, J., Won, S. and Lee, C. (2021), 'Unsupervised deep learning for super-resolution reconstruction of turbulence', *Journal of Fluid Mechanics* **910**, A29.

[52] Kingma, D. P. and Ba, J. (2014), 'Adam: A method for stochastic optimization', *arXiv preprint arXiv:1412.6980* .

[53] Kingma, D. P. and Welling, M. (2013), 'Auto-encoding variational bayes', *arXiv preprint arXiv:1312.6114* .

[54] Kipf, T. N. and Welling, M. (2016), 'Semi-supervised classification with graph convolutional networks', *arXiv preprint arXiv:1609.02907* .

[55] Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P. and Hoyer, S. (2021), 'Machine learning–accelerated computational fluid dynamics', *Proceedings of the National Academy of Sciences* **118**(21), e2101784118.

[56] Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012), 'Imagenet classification with deep convolutional neural networks', *Advances in neural information processing systems* **25**.

[57] Kullback, S. and Leibler, R. A. (1951), 'On information and sufficiency', *The annals of mathematical statistics* **22**(1), 79–86.

[58] Launder, B. E. and Sharma, B. I. (1974), 'Application of the energy-dissipation model of turbulence to the calculation of flow near a spinning disc', *Letters in heat and mass transfer* **1**(2), 131–137.

[59] Le, H., Moin, P. and Kim, J. (1997), 'Direct numerical simulation of turbulent flow over a backward-facing step', *Journal of fluid mechanics* **330**, 349–374.

[60] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. and Jackel, L. D. (1989), 'Backpropagation applied to handwritten zip code recognition', *Neural computation* **1**(4), 541–551.

[61] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998), 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE* **86**(11), 2278–2324.

[62] LeCun, Y., Kavukcuoglu, K. and Farabet, C. (2010), Convolutional networks and applications in vision, *in* 'Proceedings of 2010 IEEE international symposium on circuits and systems', IEEE, pp. 253–256.

[63] Lee, J.-Y. and Park, J. (2021), 'Deep regression network-assisted efficient streamline generation method', *IEEE Access* **9**, 111704–111717.

[64] Lee, K. and Carlberg, K. T. (2020), 'Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders', *Journal of Computational Physics* **404**, 108973.

[65] Li, H., Xu, Z., Taylor, G., Studer, C. and Goldstein, T. (2018), 'Visualizing the loss landscape of neural nets', *Advances in neural information processing systems* **31**.

[66] Li, H., Yang, Y., Chang, M., Chen, S., Feng, H., Xu, Z., Li, Q. and Chen, Y. (2022), 'Srdiff: Single image super-resolution with diffusion probabilistic models', *Neurocomputing* **479**, 47–59.

[67] Li, M. and McComb, C. (2022), 'Using physics-informed generative adversarial networks to perform super-resolution for multiphase fluid simulations', *Journal of Computing and Information Science in Engineering* **22**(4), 044501.

[68] Ling, J., Jones, R. and Templeton, J. (2016), 'Machine learning strategies for systems with invariance properties', *Journal of Computational Physics* **318**, 22–35.

[69] Ling, J., Kurzawski, A. and Templeton, J. (2016), 'Reynolds averaged turbulence modelling using deep neural networks with embedded invariance', *Journal of Fluid Mechanics* **807**, 155–166.

[70] Ling, J. and Templeton, J. (2015), 'Evaluation of machine learning algorithms for prediction of regions of high reynolds averaged navier stokes uncertainty', *Physics of Fluids* **27**(8).

[71] Liu, B., Tang, J., Huang, H. and Lu, X.-Y. (2020), 'Deep learning methods for super-resolution reconstruction of turbulent flows', *Physics of Fluids* **32**(2).

[72] Lu, Z., Li, J., Liu, H., Huang, C., Zhang, L. and Zeng, T. (2022), Transformer for single image super-resolution, *in* 'Proceedings of the IEEE/CVF conference on computer vision and pattern recognition', pp. 457–466.

[73] Lumley, J. L. (1967), 'The structure of inhomogeneous turbulent flows', *Atmospheric turbulence and radio wave propagation* pp. 166–178.

[74] Luo, W., Li, Y., Urtasun, R. and Zemel, R. (2016), 'Understanding the effective receptive field in deep convolutional neural networks', *Advances in neural information processing systems* **29**.

[75] Maulik, R., Lusch, B. and Balaprakash, P. (2021), 'Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders', *Physics of Fluids* **33**(3).

[76] Menter, F. R. (1994), 'Two-equation eddy-viscosity turbulence models for engineering applications', *AIAA journal* **32**(8), 1598–1605.

[77] Milano, M. and Koumoutsakos, P. (2002), 'Neural network modeling for near wall turbulent flow', *Journal of Computational Physics* **182**(1), 1–26.

[78] Monfort, M., Luciani, T., Komperda, J., Ziebart, B., Mashayek, F. and Marai, G. E. (2017), A deep learning approach to identifying shock locations in turbulent combustion tensor fields, *in* 'Modeling, Analysis, and Visualization of Anisotropy', Springer, pp. 375–392.

[79] Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J. and Bronstein, M. M. (2017), Geometric deep learning on graphs and manifolds using mixture model cnns, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 5115–5124.

[80] Morimoto, M., Fukami, K. and Fukagata, K. (2021), 'Experimental velocity data estimation for imperfect particle images using machine learning', *Physics of Fluids* **33**(8).

[81] Moser, R. D., Kim, J. and Mansour, N. N. (1999), 'Direct numerical simulation of turbulent channel flow up to re τ= 590', *Physics of fluids* **11**(4), 943–945.

[82] Murata, T., Fukami, K. and Fukagata, K. (2020), 'Nonlinear mode decomposition with convolutional neural networks for fluid dynamics', *Journal of Fluid Mechanics* **882**, A13.

[83] Nanni, L., Costa, Y. M., Aguiar, R. L., Mangolin, R. B., Brahnam, S. and Silla, C. N. (2020), 'Ensemble of convolutional neural networks to improve animal audio classification', *EURASIP Journal on Audio, Speech, and Music Processing* **2020**(1), 1–14.

[84] Notay, Y. (2010), 'An aggregation-based algebraic multigrid method', *Electronic transactions on numerical analysis* **37**, 123–146.

[85] Obi, S., Aoki, K. and Masuda, S. (1993), Experimental and computational study of turbulent separating flow in an asymmetric plane diffuser, *in* 'Ninth symposium on turbulent shear flows', Vol. 305, pp. 305–312.

[86] Obiols-Sales, O., Vishnu, A., Malaya, N. P. and Chandramowlishwaran, A. (2021), Surfnet: Super-resolution of turbulent flows with transfer learning using small datasets, *in* '2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)', IEEE, pp. 331–344.

[87] Pant, P. and Farimani, A. B. (2020), 'Deep learning for efficient reconstruction of high-resolution turbulent dns data', *arXiv preprint arXiv:2010.11348* .

[88] Raissi, M. (2018), 'Deep hidden physics models: Deep learning of nonlinear partial differential equations', *The Journal of Machine Learning Research* **19**(1), 932–955.

[89] Raissi, M. and Karniadakis, G. E. (2018), 'Hidden physics models: Machine learning of nonlinear partial differential equations', *Journal of Computational Physics* **357**, 125–141.

[90] Raissi, M., Perdikaris, P. and Karniadakis, G. E. (2017), 'Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations', *arXiv preprint arXiv:1711.10561* .

[91] Rajendran, V., Kelly, K. Y., Leonardi, E. and Menzies, K. (2018), Vortex detection on unsteady cfd simulations using recurrent neural networks, *in* '2018 Fluid dynamics conference', p. 3724.

[92] Rau, G., C¸akan, M., Moeller, D. and Arts, T. (1998), 'The effect of periodic ribs on the local aerodynamic and heat transfer performance of a straight cooling channel'.

[93] Rhie, C. M. and Chow, W.-L. (1983), 'Numerical study of the turbulent flow past an airfoil with trailing edge separation', *AIAA journal* **21**(11), 1525–1532.

[94] Ronneberger, O., Fischer, P. and Brox, T. (2015), U-net: Convolutional networks for biomedical image segmentation, *in* 'Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18', Springer, pp. 234–241.

[95] Rotta, J. (1951), 'Statistische theorie nichthomogener turbulenz', *Zeitschrift für Physik* **129**, 547–572.

[96] saturne support@edf.fr (2021), *code_saturne 7.1 Theory Guide*, https://www.code-saturne.org/documentation/7.1/theory.pdf.

[97] Sharma, P., Chung, W. T., Akoush, B. and Ihme, M. (2023), 'A review of physics-informed machine learning in fluid mechanics', *Energies* **16**(5), 2343.

[98] Shu, D., Li, Z. and Farimani, A. B. (2023), 'A physics-informed diffusion model for high-fidelity flow field reconstruction', *Journal of Computational Physics* **478**, 111972.

[99] Simonyan, K. and Zisserman, A. (2014), 'Very deep convolutional networks for large-scale image recognition', *arXiv preprint arXiv:1409.1556* .

[100] Sinha, A., Kumar, R. and Umakant, J. (2022), 'Reduced-order model for efficient generation of a subsonic missile's aerodynamic database', *The Aeronautical Journal* **126**(1303), 1546–1567.

[101] Speziale, C. G., Sarkar, S. and Gatski, T. B. (1991), 'Modelling the pressure–strain correlation of turbulence: an invariant dynamical systems approach', *Journal of fluid mechanics* **227**, 245–272.

[102] Ströfer, C. M., Wu, J., Xiao, H. and Paterson, E. (2018), 'Data-driven, physics-based feature extraction from fluid flow fields', *arXiv preprint arXiv:1802.00775* .

[103] Sutskever, I., Martens, J., Dahl, G. and Hinton, G. (2013), On the importance of initialization and momentum in deep learning, *in* 'International conference on machine learning', PMLR, pp. 1139–1147.

[104] Thompson, R. L. and de Souza Mendes, P. R. (2011), 'A constitutive model for non-newtonian materials based on the persistence-of-straining tensor', *Meccanica* **46**(5), 1035–1045.

[105] Vanek, P., Mandel, J. and Brezina, M. (1994), 'Algebraic multigrid on unstructured meshes', *UCD/CCM Report* **34**, 123–146.

[106] Volkov, K. (2018), 'Multigrid and preconditioning techniques in cfd applications'.

[107] Wald, J.-F. (2016), Lois de paroi adaptatives pour un modèle de fermeture du second ordre dans un contexte industriel, PhD thesis, Université de Pau et des Pays de l'Adour.

[108] Wang, J., Guo, L., Wang, Y., Deng, L., Wang, F. and Li, T. (2020), 'A vortex identification method based on extreme learning machine', *International Journal of Aerospace Engineering* **2020**.

[109] Wang, R., Kashinath, K., Mustafa, M., Albert, A. and Yu, R. (2020), Towards physics-informed deep learning for turbulent flow prediction, *in* 'Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining', pp. 1457–1466.

[110] Wang, R., Walters, R. and Yu, R. (2020), 'Incorporating symmetry into deep dynamics models for improved generalization', *arXiv preprint arXiv:2002.03061* .

[111] Wang, Y., Deng, L., Yang, Z., Zhao, D. and Wang, F. (2021), 'A rapid vortex identification method using fully convolutional segmentation network', *The Visual Computer* **37**(2), 261–273.

[112] Wang, Y., Solera-Rico, A., Vila, C. S. and Vinuesa, R. (2024), 'Towards optimal β-variational autoencoders combined with transformers for reduced-order modelling of turbulent flows', *International Journal of Heat and Fluid Flow* **105**, 109254.

[113] Weiler, M. and Cesa, G. (2019), 'General e (2)-equivariant steerable cnns', *Advances in neural information processing systems* **32**.

[114] Wilcox, D. C. (1993), 'Comparison of two-equation turbulence models for boundary layers with pressure gradient', *AIAA journal* **31**(8), 1414–1421.

[115] Wilcox, D. C. (1994), 'Simulation of transition with a two-equation turbulence model', *AIAA journal* **32**(2), 247–255.

[116] Wilcox, D. C. et al. (1998), *Turbulence modeling for CFD*, Vol. 2, DCW industries La Canada, CA.

[117] Wu, J., Xiao, H., Sun, R. and Wang, Q. (2019), 'Reynolds-averaged navier–stokes equations with explicit data-driven reynolds stress closure can be ill-conditioned', *Journal of Fluid Mechanics* **869**, 553–586.

[118] Xie, C., Xiong, X. and Wang, J. (2021), 'Artificial neural network approach for turbulence models: A local framework', *Physical Review Fluids* **6**(8), 084612.

[119] Xie, Y., Franz, E., Chu, M. and Thuerey, N. (2018), 'tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow', *ACM Transactions on Graphics (TOG)* **37**(4), 1–15.

[120] Yang, W., Zhang, X., Tian, Y., Wang, W., Xue, J.-H. and Liao, Q. (2019), 'Deep learning for single image super-resolution: A brief review', *IEEE Transactions on Multimedia* **21**(12), 3106–3121.

[121] Yang, Z., Dong, Y., Deng, X. and Zhang, L. (2022), 'Amgnet: Multi-scale graph neural networks for flow field prediction', *Connection Science* **34**(1), 2500–2519.

[122] Ye, S., Zhang, Z., Song, X., Wang, Y., Chen, Y. and Huang, C. (2020), 'A flow feature detection method for modeling pressure distribution around a cylinder in non-uniform flows by using a convolutional neural network', *Scientific reports* **10**(1), 1–10.

[123] Ye, Z., Chen, Q., Zhang, Y., Zou, J. and Zheng, Y. (2019), 'Identification of vortex structures in flow field images based on convolutional neural network and dynamic mode decomposition.', *Traitement du Signal* **36**(6), 501–506.

[124] Yousif, M. Z., Yu, L. and Lim, H.-C. (2021), 'High-fidelity reconstruction of turbulent flow from spatially limited data using enhanced super-resolution generative adversarial network', *Physics of Fluids* **33**(12).

[125] Zhou, X.-H., McClure, J. E., Chen, C. and Xiao, H. (2022), 'Neural network–based pore flow field prediction in porous media using super resolution', *Physical Review Fluids* **7**(7), 074302.

[126] Zhu, J.-Y., Park, T., Isola, P. and Efros, A. A. (2017), Unpaired image-to-image translation using cycle-consistent adversarial networks, *in* 'Proceedings of the IEEE international conference on computer vision', pp. 2223–2232.

RÉSUMÉ

La dynamique des fluides numérique (CFD) s'est imposée depuis plusieurs années comme un outil indispensable pour l'étude des phénomènes d'écoulement complexes en recherche et en industrie. La précision des simulations CFD dépend de plusieurs paramètres – géométrie, maillage, schémas, solveurs, etc. – ainsi que de connaissances phénoménologiques que seul un ingénieur expert en CFD peut configurer et optimiser. L'objectif de ce travail de thèse est de proposer un assistant IA pour aider les utilisateurs, qu'ils soient experts ou non, à mieux choisir les options de simulation et à garantir la fiabilité des résultats pour un phénomène d'écoulement cible. Dans ce cadre, des algorithmes d'apprentissage profond sont explorés pour identifier les caractéristiques des écoulements calculés sur des maillages structurés et non structurés de géométries complexes. Dans un premier temps, des réseaux de neurones convolutifs (CNN), réputés pour leur capacité à extraire des motifs sur des images, sont utilisés pour identifier des phénomènes d'écoulement tels que les tourbillons et la stratification thermique sur des maillages structurés en 2D. Bien que les résultats obtenus sur maillages structurés soient satisfaisants, les réseaux CNN ne peuvent être appliqués qu'à ce type de maillage. Pour surmonter cette limitation, un cadre de réseau neuronal basé sur les graphes (GNN) est proposé. Ce cadre utilise l'architecture U-Net et une hiérarchie de graphes successivement déraffinés grâce à la mise en oeuvre d'une méthode multigrille (AMG) inspirée de celle utilisée dans le code de simulation Code_Saturne. Par la suite, une étude approfondie des fonctions à noyau a été menée selon des critères de précision d'identification et d'efficacité d'entraînement pour mieux filtrer les différents phénomènes sur maillages non structurés. Après avoir comparé des fonctions à noyau disponibles dans la littérature, une nouvelle fonction à noyau basée sur le modèle de mélange gaussien a été proposée. Cette fonction est mieux adaptée à l'identification de phénomènes d'écoulement sur des maillages non structurés. La supériorité de l'architecture et de la fonction à noyau proposées est démontrée par plusieurs expériences numériques d'identification des tourbillons en 2D, ainsi que par son adaptabilité à l'identification des caractéristiques d'un écoulement en 3D.

MOTS CLÉS

Identification des phénomènes d'ecoulement; Réseau neuronal convolutif; Réseau neuronal en graphes; Méthode algébrique multigrille

ABSTRACT

Computational Fluid Dynamics (CFD) has become an indispensable tool for studying complex flow phenomena in both research and industry over the years. The accuracy of CFD simulations depends on various parameters – geometry, mesh, schemes, solvers, etc. – as well as phenomenological knowledge that only an expert CFD engineer can configure and optimize. The objective of this thesis is to propose an AI assistant to help users, whether they are experts or not, to better choose simulation options and ensure the reliability of results for a target flow phenomenon. In this context, deep learning algorithms are explored to identify the characteristics of flows computed on structured and unstructured meshes of complex geometries. Initially, convolutional neural networks (CNNs), known for their ability to extract patterns from images, are used to identify flow phenomena such as vortices and thermal stratification on structured 2D meshes. Although the results obtained on structured meshes are satisfactory, CNNs can only be applied to structured meshes. To overcome this limitation, a graph-based neural network (GNN) framework is proposed. This framework uses the U-Net architecture and a hierarchy of successively refined graphs through the implementation of a multigrid method (AMG) inspired by the one used in the Code_Saturne CFD code. Subsequently, an in-depth study of kernel functions was conducted according to identification accuracy and training efficiency criteria to better filter the different phenomena on unstructured meshes. After comparing available kernel functions in the literature, a new kernel function based on the Gaussian mixture model was proposed. This function is better suited to identifying flow phenomena on unstructured meshes. The superiority of the proposed architecture and kernel function is demonstrated by several numerical experiments identifying 2D vortices and its adaptability to identifying the characteristics of a 3D flow.

KEYWORDS

Flow phenomena identification; Convolutional neural networks; Graph neural networks; Algebraic multigrid method