UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AERONÁUTICA Y DEL ESPACIO

MÁSTER UNIVERSITARIO EN INGENIERÍA AERONÁUTICA

TRABAJO FIN DE MÁSTER

Accelerating High-Order Discontinuous Galerkin solvers using Neural Networks

AUTOR: Fernando MANRIQUE DE LARA LOMBARTE

INTENSIFICACIÓN: Propulsión Aeroespacial

TUTOR DEL TRABAJO: Esteban FERRER VACCAREZZA

Junio de 2022

Abstract

High order Discontinuous Galerkin methods generate very accurate solutions when increasing the polynomial order inside each element. Even when the desired solution requires only the low order modes from the solution, they are still affected by the high order modes and hence cannot be neglected. The effect of these high order modes can be seen as a source term in a problem where only the low order solution is considered. We propose to model the source term with a neural network and get an accurate evolution of the low order modes with less degrees of freedom and less time step restrictions in order to speed up the computations. Based on these ideas, in this final master thesis, we develop and test a new methodology with the 1D Burgers' equation and the 3D compressible Navier-Stokes equations in a Taylor Green Vortex test case.

Keywords: Discontinuous Galerkin, Computational Fluid Dynamics, Machine Learning, Neural Networks, Burgers equation, Navier-Stokes equation

Resumen

El método high order Discontinuous Galerkin genera soluciones muy precisas al aumentar el orden polinómico dentro de cada elemento. Incluso cuando la solución deseada requiere sólo los modos de bajo orden de la misma, estos todavía se ven afectados por los modos de alto orden y por lo tanto no pueden ser despreciados. El efecto de estos modos de alto orden puede verse como un término fuente en un problema en el que sólo se considera la solución de bajo orden. El autor propone modelar el término fuente con una red neuronal y obtener una evolución precisa de los modos de bajo orden con menos grados de libertad y menos restricciones de paso de tiempo con el objetivo de reducir el coste computacional de las simulaciones. En este trabajo de fin de máster, se desarrolla matemáticamente la metodología y se prueba con la ecuación de Burgers en 1D y las ecuaciones de Navier-Stokes compresibles en 3D en un caso de prueba de Taylor Green Vortex.

Keywords: Discontinuous Galerkin, Dinámica de fluidos computacional, Machine Learning, Neural Networks, ecuación de Burgers, ecuaciones de Navier-Stokes

Acknowledgement

First of all I would like to thank my supervisor, Esteban Ferrer, not only for the opportunity he has given me to do this master's thesis, but also for the trust in me from the very first moment, the integration he has given me in the NUMATH group and the knowledge and time dedicated to me. Thank you for everything you have taught me and for letting and helping me develop my own ideas.

I would like to thank other members of the department, especially Gonzalo Rubio, Wojciech Laskowski and Gerasimos Ntoukas who have listened to me and helped me without asking for anything in return. I would also like to make a special mention to Ricardo Vinuesa for reading the project and collaborating between UPM and KTH.

To my girlfriend Belén, thank you for spending so many hours listening to my disappointments when things didn't work out and my illusions when the results seemed promising. Thank you for always being there, in the good times when things went well and thank you for encouraging me to keep going in difficult situations.

Thanks to my parents for supporting me at home, for having taught me the culture of effort, of work and for helping me give the best version of myself. Since I was a child, you have helped me and this work would never have been possible without you, thank you very much.

Last but not least, thank you Adolfo, Armando and David. You have always been there helping me and giving me an example to keep going and never give up.

To all of you, thank you very much!

Agradecimientos

En primer lugar me gustaría agradecer a mi tutor, Esteban Ferrer no solo la oportunidad que me ha bridado de realizar este trabajo fin de master, sino también la confianza que me ha dado desde el primer momento, la integración que me ha dado en el grupo de NUMATH y tanto conocimiento y tiempo que me ha dedicado. Me has hecho sentir cómodo, me has enseñado muchísimo y has confiado en mí para desarrollar mis propias ideas, estoy muy agradecido.

Quiero agradecer a otros miembros del departamento, en especial a Gonzalo Rubio, Wojciech Laskowski y Gerasimos Ntoukas que me han escuchado y me han ayudado sin pedir nada a cambio. También quiero hacer mención especial a Ricardo Vinuesa por haber leído el proyecto y querer continuarlo colaborando entre la UPM y el KTH.

A mi novia Belén, gracias por dedicarme tantas horas escuchando mis decepciones cuando las cosas no salían, o las ilusiones cuando los resultados parecían prometedores. Gracias por estar siempre, en las buenas cuando las cosas salían bien y gracias por animarme para seguir adelante en las malas.

Gracias a mis padres por haberme apoyado en casa, por haberme inclulcado la cultura del esfuerzo, del trabajo y a querer dar lo mejor de mí mismo. Desde pequeño me habéis ayudado y este trabajo nunca habría sido posible sin vosotros, muchas gracias.

Por último pero no menos importante, gracias a Adolfo, Armando y David. Habéis estado siempre ahí ayudándome y dándome ejemplo para, a pesar de que las situaciones no sean óptimas, seguir adelante y no tirar la toalla.

A todos, ¡muchas gracias!

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

Many of the relevant mathematical-physical problems of science and industry are related with the formulation of systems of non-linear partial differential equations (PDEs). For the vast majority of the cases, we cannot solve these systems analytically or the necessary simplifications to do so reduce the accuracy of the models. In many cases experimental testing is too expensive, and due to the great development in computation in the last decades, the use of numerical tools for simulation has increased in the design process.

Fluid mechanics is a part of physics very keen to be simulated due to its presence in most of technological applications and industries such as aviation, automobile, nautical industry etc. It is particularly challenging due to the fact that the PDEs governing the behaviour of fluids is highly non-linear leading to complex phenomena such as turbulence which we still not fully understood. The numerical and programming techniques responsible for the approximate resolution of these equations also turn out to be very sophisticated and fall into the group known as Computational Fluid Dynamics (CFD).

Without going into too much detail, the standard way of solving the Navier-Stokes equations for a time marching solution involves a spatial discretisation, where the domain is divided into elements or nodes, and then a temporal discretization where the system is integrated in time [1]. One of the first methods used for spatial discretisation is the finite difference method (FDM, [2]), in which we divide the domain into a structured

mesh where we know the fluid field at the nodes. From the values at the nodes, we approximate the derivatives and spatial operators, generating a system of ordinary differential equations (ODEs) that give the time evolution of the flow field at each node. One of the issues with this formulation is that it does not always conserve mass energy and momentum (i.e. the system can generate or disappear any of these three variables if the discretisation is not correct).

For this reason, most of the CFD codes have changed the differential formulation for an inherent conservative formulation based on the integral equations, Finite Volume Method (FVM, [3]). This scheme has been an attraction in the industry and scientific community and the basis for a lot of study and development in the past decades.

Another methodology that can also be used in CFD, although it is not as widespread as the FVM, is the finite element method (FEM, [4]). In FEM, the domain is divided into elements and approximates the solution inside of each with a polynomial. This method uses the weak formulation to obtain a system of ODEs that gives a time evolution equation for each of the degrees of freedom of the polynomial.

1.1.1 Low order methods

In practice these three methods use either just a few neighbours to construct the interpolant (in FDM and FVM) or a low-order polynomial (P<3) to approximate the spatial operators inside the element (FEM). Both methodologies are known as low order methods and they avoid round-off errors [5] easily since interpolation problems are usually not ill-conditioned (unless the mesh is very fine). However, low order schemes loose accuracy in the form of dissipation and dispersion due to interpolation errors [5, 6]. The error decreases as a power of the mesh size, and in low order methods, this power is a low number and hence the error decreases slowly when refining a mesh.

An alternative is to increase the order of the schemes to increase the accuracy of the numerical schemes within a fixed mesh. For example, in FVM we find schemes that instead of assuming constant solutions in each element, interpolate the solution within the elements and increase the order of the schemes (see ENO [7] or WENO [8] schemes). However, to increase the order of the schemes significantly we require information from multiple adjacent elements, which are usually not placed in a structured way. This implies that, with high probability, as we increase the order the interpolation matrices start to be ill-conditioned and we start having round-off errors.

1.1.2 High order methods

To avoid reconstructing the solution from external nodes, which leads to round-off errors, the CFD community have looked into variaties of the FEM combined with spectral methods [9, 10]. From the FEM (low-order) there has been some work with Continuous Galerkin (CG) and Discontinuous Galerkin (DG). DG methods allow the solution to be discontinuous at the element interfaces and the fluxes at the interfaces are calculated in the same way as the fluxes for the FVM. These discontinuities introduce numerical dissipation which increases the robustness of the scheme. Furthermore, in the continuous formulation it is necessary to make certain nodes "slaves" of others in a sense that they do not have a ODE for their evolution because they need to adopt an imposed value to fulfil the continuity of the solution at the interface.

From the FEM schemes, the CFD community has chosen the DG method due to its greater robustness in comparison to CG. Now, the DG method is modified and combined with spectral methods to give birth to the discontinuous Galerkin spectral element method (DGSEM). The main difference between the two methods (DG and DGSEM) is that in the case of nodal formulation, DGSEM places the nodes placed at Gauss or Gauss-Lobatto nodes reducing the Lebesgue constant (reducing round-off errors). Under certain conditions these collocation node allow exact numerical quadratures and spectral convergence.

1.2 Objective and methodology

DGSEM also known as high order discontinuous Galerkin might exhibit spectral convergence behaviour and thus very accurate solutions e.g. [11, 12, 13, 14]. However, when increasing the polynomial order, the number of degrees of freedom increase to the power of the spatial dimension, and hence the cost increases substantially.

Even if we are only interested in the “mean” values from the field we have to solve all the scales and thus solve the system with a high enough polynomial order. The reason for it is that the evolution equations for the low order modes in the presence of high order modes is not equal to a problem where there are not high order modes. By taking a look at the equations we can see that the effect of the high order modes can be seen as a source term in the equations for only the low order modes. If we were able to get this forcing terms we could decrease the polynomial order inside each element, have a proper evolution of the low order modes without computing the high order ones and then reduce the number of degrees of freedom and increase the time step due to a reduction in the time step restriction due to stability.

On the other hand, neural networks (NN) have shown to be very useful taking

decisions with minimal supervision. They have been a very attractive tool in many scientific fields but also for applications in fluid mechanics, see the review by Brunton et al. [15] or Garnier et al. [16]. These NN can be used in different ways in the field of Computational Fluid Dynamics (CFD), for instance Bar-Sinai et al. [17]) estimated the spatial derivatives of partial differential equations using NNs for the Burgers' equation on a low resolution grid. More work has been done in the context of turbulence modelling with NN as Stachenfeld et al. [18], Beck and Kurz [19]. Vinuesa and Brunton [20] made a summary of the possible enhancements of fluid dynamic simulations using machine learning. Other work from show different possibilities as using these NNs to predict flow in this paper by Guastoni et al. [21] or as showed by Güemes et al. [22] where they genera high resolution models from low resolution data.

With this context we present a methodology in which we try to accelerate the computation of a High Order DG solution by evolving the low order modes from a high order solution by correcting with a NN the evolution of a low order solution. This corrective forcing relieved the computational cost by reducing the degrees of freedom in the problem (only computing the low order modes) and by reducing the time step restrictions (i.e. higher Courant–Friedrichs–Levy or CFL limit). This approximation should be seen as a numerical method to accelerate time evolving PDEs.

This final project is the result from two papers [23], where we test and apply the methodology in the 1D Burgers' equation and the 3D compressible Navier-Stokes equations and check if it works and how well it extrapolates from one to three dimensional cases. For that, we have chosen the Taylor Green Vortex test case where we test this acceleration method for different Reynolds numbers as well as different stages of the simulation.

1.3 Outline

The rest of the thesis is structured as follows: first, a background chapter presents the mathematical basis of the formulation of the high order discontinuous Galerkin methodology. It also explains the different types of temporal schemes for the integration of EDOs, something that will be fundamental since the methodology is implemented within these integrators. Subsequently, we give a brief introduction to machine learning and its applications to fluid mechanics.

After this background, we present the new methodology in chapter 3 with both the methodology and some mathematical analysis of the error as well. We apply the methodology to the 1D Burgers equation as well as to the 3D Navier-Stokes equations in the Taylor Green Vortex case, and then present them in a results section.

Finally, in chapter 4 we address some conclusions of the work, showing the strengths

and weaknesses of the methodology and future work to be done that might be interesting for improving the methodology.

1.4 Highlights and novel contributions

In this work we present a new methodology that uses NNs to generate models that maintain a correct evolution of the low-order modes despite the lack of a high-order scheme. This methodology was tested with the 1D Burgers' equation and published in the journal Computers & Fluids [23]. We have also presented the results of the Navier-Stokes tests and it is under review. The topic has also been the subject of conferences, both presented by the author (ISUDEF 2022) and by Esteban Ferrer (Eccomas). We hope this methodology to become the beginning of a line of study and application to reduce the computational cost of high-order methods.

1.4.1 List of publications

Journal Publications

- F. Manrique de Lara and E. Ferrer. Accelerating high order discontinuous Galerkin with Neural networks: 1D Burgers' equation, Computers & Fluids (2022)

- F. Manrique de Lara and E. Ferrer . Accelerating high order discontinuous Galerkin with Neural networks: 3D Navier-Stokes equation, Under review (2022)

- E. Ferrer, G. Rubio, G. Ntoukas, W. Laskowski, O.A. Mariño, S. Colombo, A. Mateo-Gabín, F. Manrique de Lara, D. Huergo, J.Manzanero, A.M. Rueda-Ramírez, D.A. Kopriva, E. Valero. HORSES3D: a high order discontinuous Galerkin solver for flow simulations and multi-physics applications, Computer Physics Communications (2022)

Short papers

- F. Manrique de Lara and E. Ferrer (2022). Using neural networks to accelerate high order discontinuous Galerkin, ISUDEF (2022)

1.4.2 Conferences

- F. Manrique de Lara and E. Ferrer (2022). Using neural networks to accelerate high order discontinuous Galerkin, ISUDEF (2022)
- E.Ferrer, F. Manrique de Lara and K.Otmani Accelerating High Order Discontinuous Galerkin solvers using neural networks, ECCOMAS Congress (2022)

CHAPTER 2

Background: Numerical methods and Machine learning

Industrial and scientific studies related to fluid flows are mathematical problems in continuous media are expressed in partial differential equations (PDEs). Before jumping into the details of the formulation and physics, it is important to have a background about systems of PDEs. Physically, these systems can be divided into two types of problems: evolution problems and boundary problems.

Boundary problems are usually expressed in the domain of space and are often associated with elliptic problems [5] in which the speed of sound of the system is infinite and the entire domain and boundary conditions affect the solution at every point in the domain. Mathematically a boundary problem is formulated as:

$$\mathcal{L}(q) = 0 \tag{2.1}$$

where $q : \mathbb{R}^D \rightarrow \mathbb{R}^H$ is a vector containing all the spatial variables with D representing the number of spatial dimensions and H the number of variables, finally $\mathcal{L} : \mathbb{R}^D \rightarrow \mathbb{R}^D$ is a spatial operator with partial derivatives that includes the boundary conditions.

On the other hand, evolution problems present a time derivative through which the solution can be integrated and evolved. Evolution problems are associated with hyperbolic and parabolic systems. Although this derivative from which it is integrated is usually time, it can also be applied to spatial derivatives in parabolic problems, as in the study of boundary layer or parabolic Navier-Stokes in which the direction of flow is used

as the integration variable [24]. In our case we will focus on time marching system of PDEs, which are expressed as:

$$\begin{aligned} \frac{\partial q}{\partial t} &= \mathcal{L}(q;t), \\ q(\mathbf{x},0) &= q_0, \\ \mathcal{R}(q,t)|_{\partial\Omega} &= 0. \end{aligned} \tag{2.2}$$

Here $t : \mathbb{R}$ is time, now the state variables are defined in space but also in time $q : \mathbb{R}^D \times \mathbb{R} \to \mathbb{R}^H$. We also have initial conditions $q_0 : \mathbb{R}^H$ and a spatial operator $\mathcal{R}$ applied at the boundary of the domain $\partial\Omega$ that are the boundary conditions.

The methods for solving boundary problems and evolution problems have a different philosophy. Although it is true that in many cases the solution of boundary problems is sought iteratively as stationary solutions of the associated evolution problem. For this reason, in this master's thesis we are going to focus on the formulation and solution of evolution problems.

2.1 Method of lines

For the numerical resolution of a system of PDEs, the presence of spatial and temporal variables must be taken into account. The most common method used is known as the method of lines. Applying this method generates three different types of mathematical problems that introduce different types of errors and are interesting to analyse. Problem I is the formulation in spatial and temporal variables:

$$\boxed{\begin{aligned} \frac{\partial q}{\partial t} &= \mathcal{L}(q;t), \\ q(\mathbf{x},0) &= q_0, \\ \mathcal{R}(q,t)|_{\partial\Omega} &= 0. \end{aligned}} \tag{2.3}$$

Problem I. Continuous formulation

Subsequently, the spatial discretisation is carried out. In this process, the domain is divided into elements, nodes or modes where the variables of the problem are known in order to reconstruct the continuous operators by discrete operators. In our case, the spatial discretisation will be done with high order discontinuous Galerkin, which will be formulated later. The result of this process is the disappearance of the spatial operators

and variables, giving rise to a system of ODEs that evolve in time (Problem II):

$$\boxed{\begin{gathered}\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{q}; t), \\ \boldsymbol{q}(0) = \boldsymbol{q}_0, \\ \text{Problem II. Temporal } \textit{ODEs} \text{ system}\end{gathered}} \tag{2.4}$$

where $\boldsymbol{q}$ is a vector containing the discrete variables of the problem, $\boldsymbol{f}$ is the discretised spatial operator and $\boldsymbol{q}_0$ are the initial conditions.

However, despite the fact that the continuous variables of the problem have disappeared, we are not able to analytically integrate such a large and complex system of ODEs as the one presented. To do so, we must carry out the temporal discretisation and integrate the system in time. There are a variety of schemes to carry out these operations, each with its advantages and disadvantages, which will be analysed later in the text. As a result of the temporal discretisation, we have a system of algebraic equations that can now be solved by the computer (Problem III).

$$\boxed{\begin{gathered}\mathbf{g}(\boldsymbol{q}^{n+1}, \boldsymbol{q}^{n}, ..., \boldsymbol{q}^{n+1-p}) = 0 \\ \boldsymbol{q}^{0} = \boldsymbol{q}_0 \\ \text{Problem III. Finite Difference formulation}\end{gathered}} \tag{2.5}$$

Where n is the time step, $\mathbf{g}$ is a generally non-linear operator resulting from the temporal and spatial discretisation, p represents the number of steps needed to construct the scheme. Depending on the type of scheme used, a way to evaluate the system at the next time step can be explicitly derived from this system of equations. This will have consequences for the stability of the system.

After showing the method of lines, in the rest of the chapter we will present with more detail the methods we have used for both the spatial discretization and temporal discretization. For the spatial discretization we present the formulation of the DG schemes and for the temporal discretization we give an insight into linear stability analysis and some time integrators. Finally we go through a brief introduction on Artificial Intelligence and Machine Learning which are the necessary basis to understand the tools we use for the methodology.

2.2 Discontinuous Galerkin method

Let us consider a conservative system of PDEs:

$$\frac{\partial q}{\partial t} + \nabla \cdot \vec{f} = 0, \tag{2.6}$$

$$\mathcal{R}(q,t) = 0, \quad q(\boldsymbol{x},0) = q_0. \tag{2.7}$$

This system can represent many different physical systems, for example the Burgers' equation or the Navier-Stokes equations. In order to discretise eq. (2.7) we first divide the domain into multiple elements. Inside each element we can multiply the equation with a test function $v^e(\boldsymbol{x})$ which is zero outside the e element:

$$v^e \frac{\partial q}{\partial t} + v^e \nabla \cdot \vec{f} = 0. \tag{2.8}$$

Since the $v^e(\boldsymbol{x})$ is not time dependent it can be introduced inside the time derivative:

$$\frac{\partial v^e q}{\partial t} + v^e \nabla \cdot \vec{f} = 0, \tag{2.9}$$

and then we can apply the chain rule for the second term of the equation and reduce the order of the derivatives applied on the state vector $\vec{f}$ and get:

$$v^e \nabla \cdot \vec{f} = \nabla \cdot (v^e \vec{f}) - \vec{f} \cdot \nabla v^e. \tag{2.10}$$

We then have:

$$\frac{\partial v^e q}{\partial t} + \nabla \cdot (v^e \vec{f}) - \vec{f} \cdot \nabla v^e = 0. \tag{2.11}$$

By integrating eq. (2.11) in the domain Ω_e we get:

$$\int_{\Omega_e} \frac{\partial v^e q}{\partial t} d\Omega + \int_{\Omega_e} \nabla \cdot (v^e \vec{f}) d\Omega - \int_{\Omega_e} \vec{f} \cdot \nabla v^e d\Omega = 0. \tag{2.12}$$

For notation simplicity, since we are integrating everything in Ω_e the index e will be omitted in the test function. The time derivative can be taken out of the first integral, and then we can use the Gauss theorem [25] to get the Weak formulation of eq. (2.7):

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_{\Omega_e} q v d\Omega + \int_{\partial\Omega_e} v \vec{f} \cdot \vec{n} dS - \int_{\Omega_e} \vec{f} \cdot \nabla v d\Omega = 0. \tag{2.13}$$

where $\partial\Omega_e$ is the boundary of the element. The weak formulation for a conservative PDE has three terms: a time evolving term, a flux term and a volume integral. All these three integrals will have to be evaluated either analytically or in an approximated way.

Before doing so, we need to describe the state variables and flux tensor in terms of some basis functions. In a general way we can approximate the state function inside each element as some coefficients multiplying some given spatial basis:

$$q(\boldsymbol{x},t) \approx \sum_{k=0}^{N} \hat{q}_k(t)\varphi_k^e(\boldsymbol{x}), \qquad \vec{f}(\boldsymbol{x},t) \approx \sum_{k=0}^{N} \hat{\vec{f}}_k(t)\varphi_k^e(\boldsymbol{x}), \tag{2.14}$$

where $N+1$ is the number of degrees of freedom used to describe q. For simplicity we will omit the e index since all the operations will be done for each element.

Once these terms are described, the unknowns are the time dependent coefficients, and hence we need to generate a system of $N+1$ equations. The way to generate them is by doing the process to get eq. (2.13) with $N+1$ different test functions:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_e} q v_j d\Omega + \int_{\partial\Omega_e} v_j \vec{f}\cdot\vec{n}dS - \int_{\Omega_e} \vec{f}\cdot\nabla v_j d\Omega = 0, \qquad j=0,\cdots,N. \tag{2.15}$$

There are many possibilities in order to choose the test function v_j. The scheme is called Galerkin if the test functions are the same as the function basis used to expand the solution variables, φ. With that in mind, eq. (2.16) can be expressed as:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_e} q \varphi_j d\Omega + \int_{\partial\Omega_e} \varphi_j \vec{f}\cdot\vec{n}dS - \int_{\Omega_e} \vec{f}\cdot\nabla \varphi_j d\Omega = 0, \qquad j=0,\cdots,N. \tag{2.16}$$

Note that if we want to keep continuity at the interfaces, then some degrees of freedom will not follow an expression as in (2.16) and but a different one to assure continuity. These type of methods are called continuous Galerkin (CG). If we allow the solution to have discontinuities at the interfaces, the method is called discontinuous Galerkin (DG).

If the method is discontinuous, note that the flux which is built from the states variables will also be discontinuous, and hence the surface flux is not defined. In that case the fluxed used is a numerical flux $\vec{f}^*$ which has to be defined as a function of the flux in both elements in each side of the surface

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_e} q \varphi_j d\Omega + \int_{\partial\Omega_e} \varphi_j \vec{f}^*\cdot\vec{n}dS - \int_{\Omega_e} \vec{f}\cdot\nabla \varphi_j d\Omega = 0. \tag{2.17}$$

These type of numerical fluxes are typically an approximation to the Riemann problem [26] and sometimes they can introduce dissipation that might make the solver to be more robust. That is one of the reasons on why the use of DG is more extended in comparison to CG schemes.

Eq. (2.17) is the weak formulation of the conservative system for a discontinuous Galerkin scheme, but it can be integrated by parts again to get the so called weak-strong

formulation

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_e} q\varphi_j d\Omega + \int_{\Omega_e} \varphi_j \nabla\cdot\vec{f}\, d\Omega + \int_{\partial\Omega_e} \varphi_j(\vec{f}^* - \vec{f})\cdot\vec{n} dS = 0, \tag{2.18}$$

since we have the divergence of the flux which is from the original strong formulation. Both expressions eq. (2.17) and eq. (2.18) can also be expressed in term of inner products as:

$$\frac{\mathrm{d}\langle q, \varphi_j\rangle_e}{\mathrm{d}t} - \langle \vec{f}, \nabla\varphi_j\rangle_e + \int_{\partial\Omega_e} \varphi_j \vec{f}^*\cdot\vec{n} dS = 0, \tag{2.19}$$

and

$$\frac{\mathrm{d}\langle q, \varphi_j\rangle_e}{\mathrm{d}t} + \langle \nabla\cdot\vec{f}, \varphi_j\rangle_e + \int_{\partial\Omega_e} \varphi_j(\vec{f}^* - \vec{f})\cdot\vec{n} dS = 0. \tag{2.20}$$

From both expressions we see that the communication of information between elements is done via surfaces integrals, and the operations inside each elements are only done with inner products. From both eq. (2.19) and eq. (2.20) we will be working with the first expression, the weak formulation.

It is interesting to see that for a constant test function $\varphi_j = 1$, eq. (2.19) takes a very interesting form. The second inner product vanishes since the gradient of a constant function is zero leaving:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_e} q d\Omega + \int_{\partial\Omega_e} \vec{f}^*\cdot\vec{n} dS = 0, \tag{2.21}$$

where we have used integral notation instead of inner products, which is the same formulation of a classical Finite Volume scheme. This means that discontinuous Galerkin schemes can be used as Finite Volume solvers when using specific basis functions: constant inside each element.

After seeing the particular case of discontinuous Galerkin scheme it is interesing to see that by combining the basis expansion of the state variables given by eq (2.14), the inner product that is temporally derived takes a particular shape:

$$\langle q, \varphi_j\rangle_e = \int_{\Omega_e} q\varphi_j d\Omega = \int_{\Omega_e} \sum_{k=0}^{N} \hat{q}_k^e(t)\varphi_k^e(\boldsymbol{x})\varphi_j d\Omega = \sum_{k=0}^{N} \hat{q}_k^e(t)\int_{\Omega_e} \varphi_k\varphi_j d\Omega. \tag{2.22}$$

We can introduce the definition of a mass matrix in each element:

$$\boldsymbol{M}^e \rightarrow M_{jk}^e = \int_{\Omega_e} \varphi_k\varphi_j d\Omega \tag{2.23}$$

hence the inner product can be expressed as:

$$\langle q, \varphi_j\rangle_e = \sum_{k=0}^{N} M_{jk}^e \hat{q}_k^e, \tag{2.24}$$

and by arranging all the coefficients in a vector such as:

$$\boldsymbol{q}^e = \begin{bmatrix} \hat{q}_0^e(t) & \hat{q}_1^e(t) & \cdots & \hat{q}_N^e(t) \end{bmatrix}^T, \tag{2.25}$$

the inner product can be expressed as a matrix product:

$$\langle q, \varphi_j \rangle_e = \boldsymbol{M}^e \boldsymbol{q}^e = \begin{bmatrix} \langle \varphi_0, \varphi_0 \rangle & \langle \varphi_0, \varphi_1 \rangle & \cdots & \langle \varphi_0, \varphi_N \rangle \\ \langle \varphi_1, \varphi_0 \rangle & \langle \varphi_1, \varphi_1 \rangle & \cdots & \langle \varphi_1, \varphi_N \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \varphi_N, \varphi_0 \rangle & \langle \varphi_N, \varphi_1 \rangle & \cdots & \langle \varphi_N, \varphi_N \rangle \end{bmatrix} \begin{bmatrix} \hat{q}_0^e(t) \\ \hat{q}_1^e(t) \\ \vdots \\ \hat{q}_N^e(t) \end{bmatrix} \tag{2.26}$$

The flux terms are more complicated to evaluate since they are generally non linear. However they can be expressed as a general non linear term as:

$$\boldsymbol{H}^e(\boldsymbol{q}) = -\langle \vec{f}, \nabla \varphi_j \rangle_e + \int_{\partial \Omega_e} \varphi_j \vec{f}^* \cdot \vec{n} dS. \tag{2.27}$$

It is worth noticing that the input of this non linear function is not only the state variables at element $\boldsymbol{q}^e$, but also the state variables at the neighbours, reason why the state variables does not have a super index e. We now have a final expression for the discretized equations as:

$$\boldsymbol{M}^e \frac{\mathrm{d}\boldsymbol{q}^e}{\mathrm{d}t} + \boldsymbol{H}^e(\boldsymbol{q}) = 0 \tag{2.28}$$

2.2.1 Numerical fluxes

The flux tensor $\vec{f}$ is generally a function of the state variables and their spatial derivatives that can be decomposed into an advection flux $\vec{f}_a$ and a viscous flux $\vec{f}_v$ (see appendix A):

$$\vec{f}(q, \nabla q) = \vec{f}_a(q) - \vec{f}_v(q, \nabla q) \tag{2.29}$$

The advection flux represents the transport of the state variables with the velocity of the flow, whereas the viscous flux dissipate the state variables. In a general case for a single variable the form of the each flux is:

$$\vec{f}_a(q) = \boldsymbol{v}(q) q, \tag{2.30}$$

where $\boldsymbol{v}(q)$ is the transport velocity and

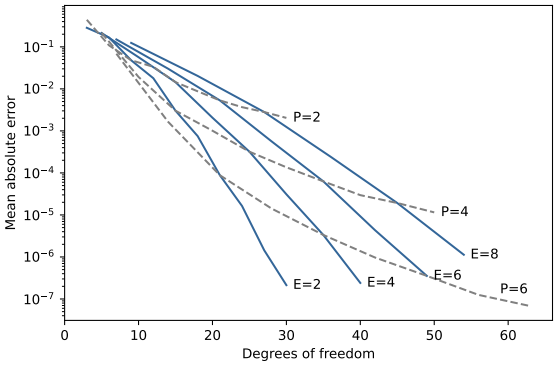
$$\vec{f}_v(q, \nabla q) = \nu(q) \nabla q, \tag{2.31}$$

$\nu(q)$ gives the diffusion of the variables.

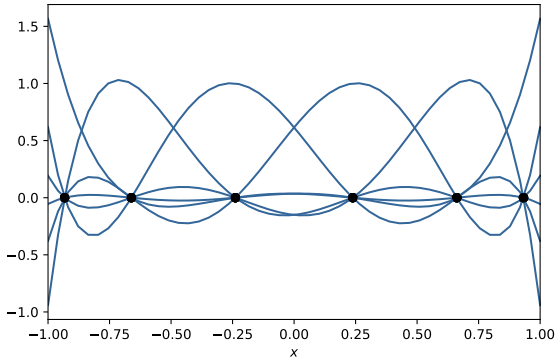
In DG schemes, the flux at the interface is not defined and depends on the solution at both sides of the surface. The resolution of the flux within a discontinuity has been deeply studied and it is given by the Riemann problem, see [26]. There are different types of schemes for the advection and diffusion fluxes. Some of the most used advection fluxes are Roe [27] and Lax-Friedrichs [28], for the viscous terms we have the Bassi-Rebay 1 scheme [29].

2.2.2 Discontinuous Galerkin spectral element method

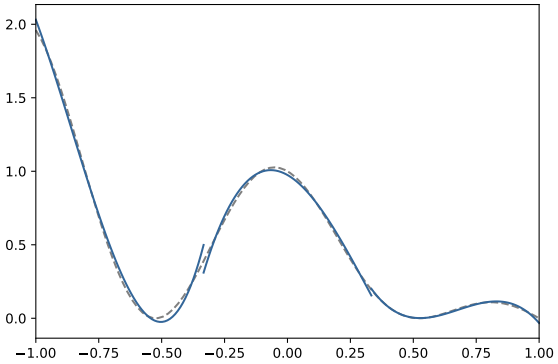
Discontinuous Galerkin spectral element method (DGSEM) is a special formulation for the classic DG method [30, 31, 32] in which the basis functions are taken to have very high accuracy and spectral convergence. In our case this is achieved by using a discretisation of each element into Gaussian nodes on which a Lagrangian polynomial basis is built (see figure 2.1b) [13, 31, 32, 33, 34].



(a) Spectral convergence example for an interpolation



(b) Lagrangian basis for $P = 7$ and one element



(c) Discontinuous approximation with DGSEM.

Figure 2.1: Lagrange basis functions used to compute the interpolation error for $y = (1-x)\cdot\cos(3x)^2$. Figure a) doted lines show a constant Polynomial (P) order and increasing elements (E). Continuous lines show constant elements and increasing P. Spectral convergence shows that increasing P is better than increasing E (for equal DOF).

With this discretisation, by keeping the degrees of freedom constant, it is much more profitable to increase the order of the polynomial than the number of elements. We show an example approximating the funtion $y = (1-x)\cdot\cos(3x)^2$ to show the effect of the polynomial and also to see a discontinuous solution from the DGSEM scheme (see figure 2.1a and 2.1c using Lagrange polynomials from figure 2.1b).

2.3 Temporal discretization

The initial value problem is a kind of problem where the variables start in an initial condition and the system evolves in time following certain laws given by a system of ODEs [5, 35]. The mathematical formulation for this problem can be expressed as follows:

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{u}, t), \tag{2.32}$$

$$\boldsymbol{u}(0) = \boldsymbol{u}_0. \tag{2.33}$$

$\boldsymbol{u}(t) : \mathbb{R} \to \mathbb{R}^N$ is the vector of variables we want to evolve, N is the number of variables, $\boldsymbol{f}(\boldsymbol{u}, t) : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$ gives the information about the evolution of the variables, $\boldsymbol{u}_0 \in \mathbb{R}^N$ is the initial condition and $t \in \mathbb{R}$ is time. It is important to recall that a system of ODEs which can be originially second order (or higher) can be formulated into a first order system.

In CFD, the system given by eq. (2.32) comes from the spatial discretization of the Navier-Stokes operator[1] which is highly non linear and very high dimensional. Both reasons make it impossible to solve the system analytically, that is why we have to make a temporal discretization and approximate the time integration for the system eq. (2.32).

To perform the temporal discretization, first we divide the continuous variable t into discrete temporal steps $t \to \{t_0, t_1, t_2, \cdots, t_n, \cdots\}$. The approximate solution will be given in each time step and the solution in each time step will be constructed with information from the previous steps:

$$\boldsymbol{G}(\boldsymbol{u}_{n+1}, \boldsymbol{u}_n, \cdots, \boldsymbol{u}_{n-r}) = 0. \tag{2.34}$$

Eq. (2.34) is a general expression of how to calculate the solution in the time step $n+1$ with the solution in the previous time steps. We classify the different schemes depending on the mathematical structure from eq. (2.34). The most important classification is discerning explicit and implicit time schemes.

Explicit schemes present the variables in the next step as a definite and unique function of previous time steps. They are much easier to implement than implicit schemes and much cheaper to evaluate at each iteration.

On the other hand, implicit schemes require solving a system of equations in order to determine the variables in the next time step, which is computationally expensive and harder if the system is non-linear.

[1] or simplified equations such as RANS, LES, etc. Also note that this is valid for the compressible N-S equations which has a time evolution equation for each variable. In the incompressible case the situation is different.

A priori it might seem that choosing an implicit scheme is not an alternative. However, in an analytically integrated system that is stable, explicit schemes require a sufficiently low time step to ensure that when integrating numerically the system is stable as well.

To understand these concepts, let us consider a linear or linearised system of ODEs:

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \boldsymbol{A}\boldsymbol{u}, \tag{2.35}$$

where $\boldsymbol{A} \in \mathcal{M}_{n\times n}$ is a diagonalizable matrix. With this condition we can rewrite the system 2.35 in their eigenvalues:

$$\frac{\mathrm{d}w_k}{\mathrm{d}t} = \lambda_k w_k, \qquad k = 1, \cdots, n \tag{2.36}$$

where $\lambda_k \in \mathbb{C}$ are the eigenvalues from matrix $\boldsymbol{A}$ and w_k components are a linear combination of the vector $\boldsymbol{u}$ corresponding to the eigenvectors. The analytical solution for each variable from eq. (2.36) is given by:

$$w_k = C\exp(\lambda_k t), \tag{2.37}$$

where C is a complex constant that depends on the initial condition. From eq. (2.37) we can see that if the real part of the eigenvectors are negative or zero, the system is stable:

$$\mathrm{Re}(\lambda_k) \leq 0 \quad \forall k, \tag{2.38}$$

because the amplitude of the solution in each direction is:

$$w_k(t) \propto \exp(\mathrm{Re}(\lambda_k)t). \tag{2.39}$$

This explanation can be extended to non diagonalizable systems [36] but we will not cover it here, see [37].

2.3.1 Explicit schemes

Now let us see on what conditions an explicit scheme is capable of having an stable solution for a system of ODEs with negative eigenvalues. Imagine we have an explicit Euler scheme [5] where:

$$\frac{w^{n+1} - w^n}{\Delta t} = \lambda w^n. \tag{2.40}$$

Note we are omitting the index k for notation simplicity. From eq. (2.40) we can get the gain G in the amplitude for a time step. If it is lower than one $G < 1$, then the amplitude decreases with each time step and hence, the solution is damped as we expect from the analytical solution in eq. (2.37). The gain is:

$$\frac{w^{n+1}}{w^n} = 1 + \lambda\Delta t \tag{2.41}$$

and the norm (amplitude) is:

$$G = \left\| \frac{w^{n+1}}{w^n} \right\| = \|1 + \lambda \Delta t\| \tag{2.42}$$

With eq.(2.42) it is possible to draw the value of G for each position in the complex plane ($\lambda \Delta t$). Notice that if we take a circle around the point with $-1 + 0i$ and a radioud of one, the Gain is constant and equal to one. All the positions inside this circle have a stable gain $G < 1$ and thus it is called stability region (see figure 2.2a).

Depending on the temporal scheme, the stability region changes. One important explicit temporal scheme that includes imaginary eigenvalues inside the stability region is the third order Runge-Kutta (see figure 2.2b) which can be calculated as:

$$\boldsymbol{k}_1 = \boldsymbol{f}(\boldsymbol{u}^n, t_n), \tag{2.43}$$

$$\boldsymbol{k}_2 = \boldsymbol{f}(\boldsymbol{u}^n + \Delta t \boldsymbol{k}_1/2, t_n + \Delta t/2), \tag{2.44}$$

$$\boldsymbol{k}_3 = \boldsymbol{f}(\boldsymbol{u}^n + \Delta t(2\boldsymbol{k}_2 - \boldsymbol{k}_1), t_n + \Delta t), \tag{2.45}$$

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \frac{\Delta t}{6}(\boldsymbol{k}_1 + 4\boldsymbol{k}_2 + \boldsymbol{k}_3). \tag{2.46}$$

In any case it can be seen that even if the scheme is stable, $\mathrm{Re}(\lambda) \leq 0$ it is possible that for large enough values of Δt some eigenvalues fall in a $G > 1$ region and hence the numerical integration is unstable. That means that the time step has to bee low enough to assure the scheme is stable (e.g. more time steps).

2.3.2 Implicit schemes

Implicit schemes are characterised by a much wider stability region than explicit schemes. This removes restrictions on the time step that can be taken because if the system of ODEs is stable, so will be the numerical integration.

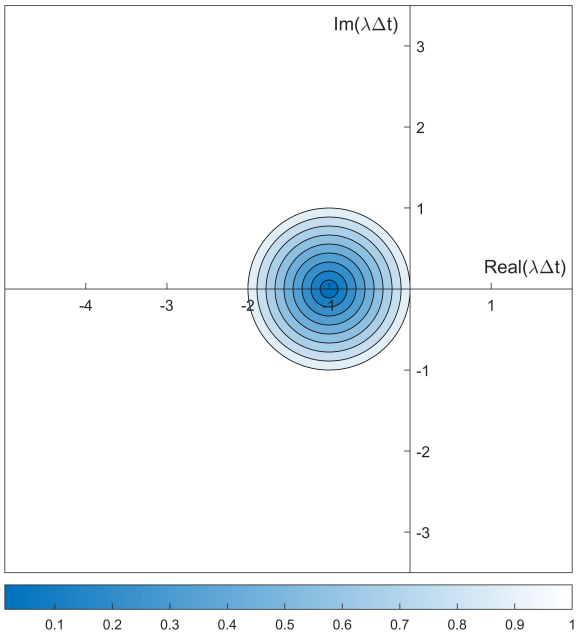
The simplest implicit scheme is the implicit Euler scheme. For this scheme we approximate the system of ODEs as:

$$\frac{w^{n+1} - w^n}{\Delta t} = \lambda w^{n+1}. \tag{2.47}$$

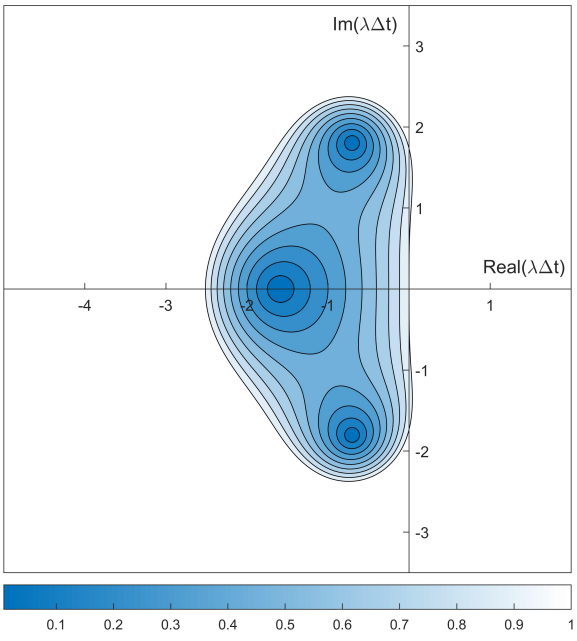
which can be manipulated to obtain the gain in a similar way as for the Explicit Euler scheme:

$$G = \left\| \frac{w^{n+1}}{w^n} \right\| = \frac{1}{\|1 - \lambda \Delta t\|}. \tag{2.48}$$

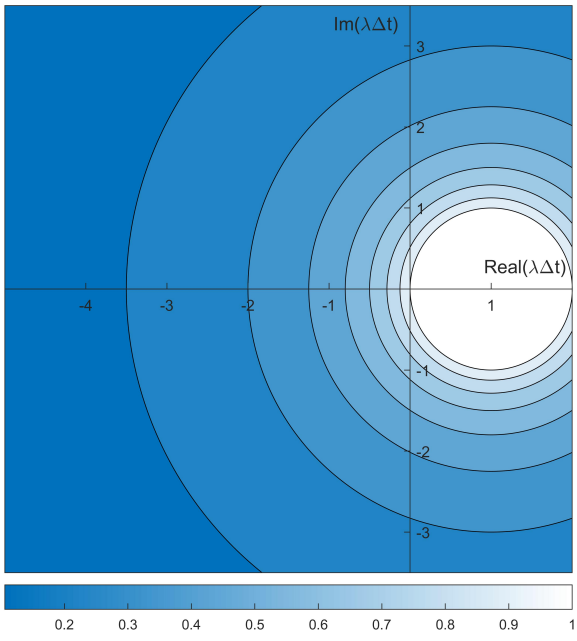
From eq. (2.48) we can see that the stability region is all the complex plain but a circle of a unity radius centred in the coordinate $1 + 0i$ (see figure 2.2c). The stability region



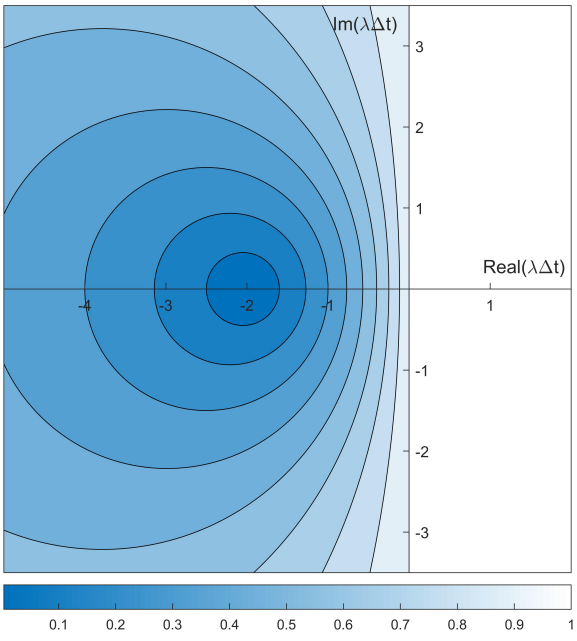
(a) Stability region for Explicit Euler



(b) Stability region for third order Runge-Kutta



(c) Stability region for implicit Euler



(d) Stability region for Crank Nicolson

Figure 2.2: Stability regions for several temporal schemes. Only stable parts are presented in color, for $G > 1$ the picture is white.

is much larger than in the case of explicit schemes, however this system damps solutions with very unstable eigenvalues (large positive real part).

There is another implicit scheme, Crank Nicolson, that has a more accurate stability region. This scheme is one of the most popular when it comes to implicit schemes and can be formulated as:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \frac{\Delta t}{2}(\boldsymbol{f}^n + \boldsymbol{f}^{n+1}). \tag{2.49}$$

This scheme has as stability region for any eigenvalue with negative real part, just as in the analytical integration (see figure 2.2d) reason why it is very attractive.

Nevertheless, in comparison to an explicit scheme, in any implicit scheme we have to solve a system of equations for every time step which can be non linear an very expensive to compute. For that reason there is not a clear decision whether it is better to use explicit or implicit schemes.

2.3.3 Time integration errors

A part from the stability part, there is another concern related to temporal schemes which is the local and global error [5, 36]. The local error is related to the order of the scheme which tells us how the derivative approximation behaves with respect to the time step:

$$e \sim \Delta t^n \tag{2.50}$$

the error is approximately proportional to a time step power (n) which is the order of the scheme. The larger the order, the lower the error and hence the better approximation we have. More information can be found at [5, 36, 38].

A summary of the four mentioned schemes can be found at table 2.1

Name	Type of scheme	Order of the error	Numer of evaluations	Stability region
Explicit Euler	Explicit	Δt	1	Figure 2.2a
3rd order Runge-Kutta	Explicit	Δt^3	3	Figure 2.2b
Implicit Euler	Implicit	Δ	Not defined	Figure 2.2c
Crank-Nicolson	Implicit	Δt^2	Not defined	Figure 2.2d

Table 2.1: Summary of the four presented time schemes.

2.4 Machine learning

Artificial intelligence (AI) is a branch of data science in which a machine/algorithm is able to perform tasks and make decisions. This definition is slightly ambiguous but sufficiently precise to understand that machine learning and neural networks are a subset of this broad group [39, 40].

Machine learning (ML) is a branch of artificial intelligence in which a machine is able to learn from data and perform tasks automatically. Machine learning algorithms extract certain properties and patterns from the data and automatically search for solutions to the proposed problems.

Machine learning algorithms can be classified into three types depending on how the data is learned [41] :

- Supervised learning. In this type of learning, the machine has a series of inputs associated with outputs. In learning, an attempt is made to optimise the machine's parameters so that the difference between the machine's output and the desired output is minimal. It is said to be supervised because an attempt is made to optimise the error of the neural network on the basis of data, which basically supervises the learning. Some examples of use are regression or decision trees.

- Unsupervised learning. This type of learning is used in algorithms dedicated to clustering carried out through the search for patterns in the data.

- Reinforcement learning. The algorithm is trained in the following way: it interprets the environment and makes decisions that are evaluated to corroborate whether they are correct or not. If the decision is correct, it is given a reward and if not, a punishment, so that the algorithm learns from past situations how to make decisions. This type of learning is widely used in Markov Decision Process [42].

Among the different machine learning algorithms, we find Deep learning, which differs from the rest of the methods in that it uses logical structures that resemble the nervous system of mammals and that have artificial neurons that are processing units with which it can obtain existing characteristics in the data [43].

2.4.1 Neural networks

Although the use of neural networks is now widespread, the first work on neural networks dates back to 1943 when they attempted to mathematically model the functioning of a

neuron. Since then there has been a great development of ideas and architectures, it is recommended to read a brief summary presented in [44] (see figure 2.3).

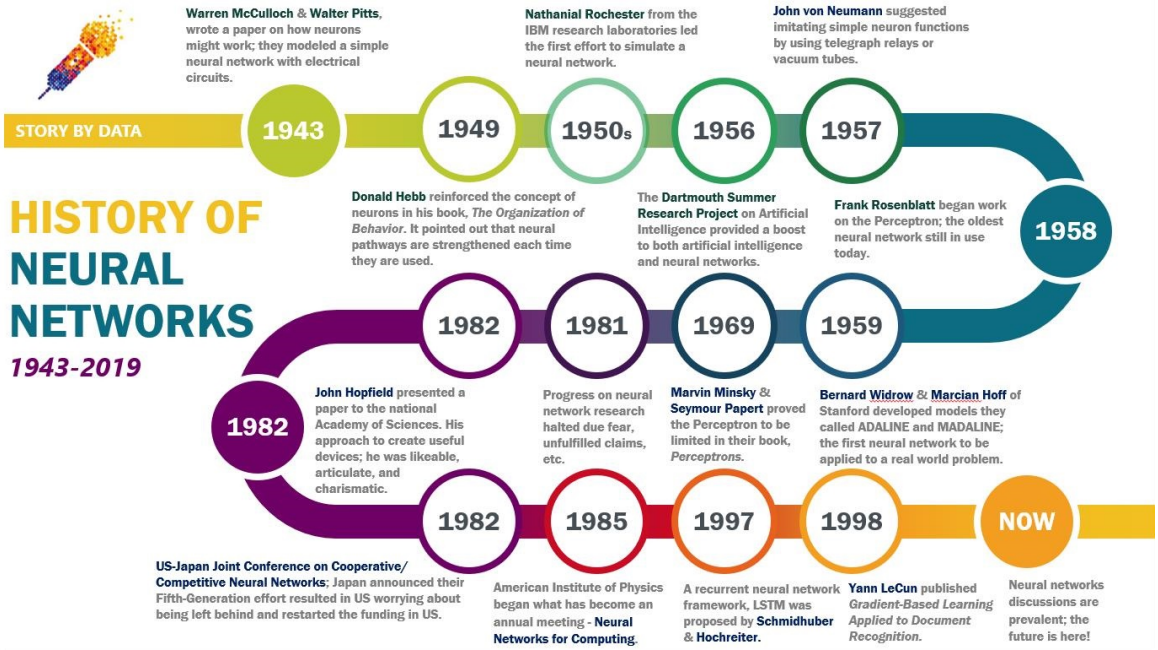


Figure 2.3: Brief summary of the neural networks history, from [44]

Neural networks are models inspired by the way neurons function [45, 46]. In the human brain there are an estimated 86 billion neurons, each of which is in turn connected to 10,000 other neurons. Neurons receive signals from each other and depending on the potential received they are activated or not. Inspired by this, artificial neural networks emerged.

An artificial neural network (NN) is a type of scheme that acts similarly to the brain neurons that receives an input and after some operations gives an output. The input to the NN is stored in a vector $\boldsymbol{x}^{(1)} \in \mathbb{R}^n$ where n is the number of components in the vector. With the terminology used in the AI community, each component of the vector is called neuron. This vector is also called as input layer, to which we do some simple operations and get the following vector $\boldsymbol{x}^{(2)} \in \mathbb{R}^m$ that has m neurons/components. This middle vector is also called hidden layer since it is an intermediate result and it is not presented for the user (see an example in figure 2.4).

To explain how the algorithmics between the layers work, let us present a set of two consecutive of layers connected where the input is $\boldsymbol{x}^{(1)} \in \mathbb{R}^n$ and the output is $\boldsymbol{x}^{(2)} \in \mathbb{R}^m$, with n and m representing the number of neurons the scheme has. The typical operations performed between layers is a linear operation to the input vector and then a non linear

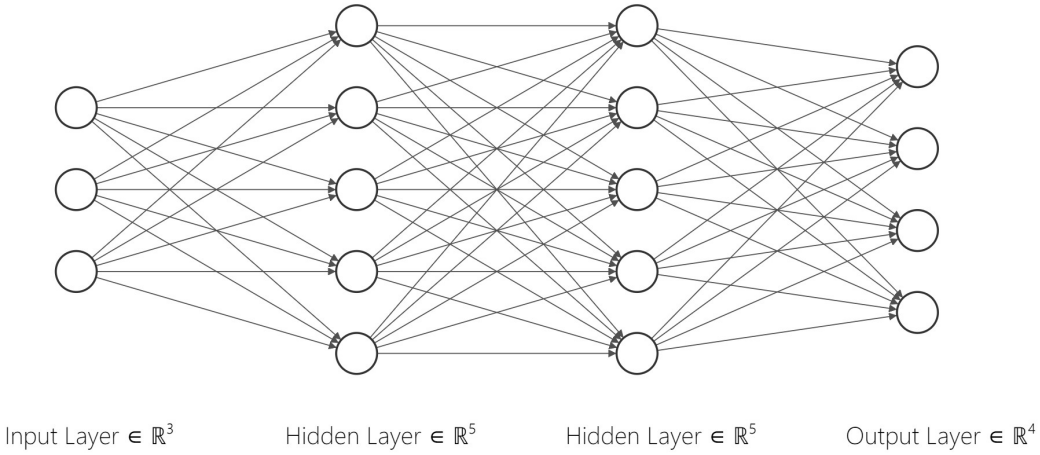


Figure 2.4: Example of a simple deep NN. This NN presents an input layer with 3 neurons, two hidden layers with 5 layers each and an output layer with 4 neurons. Graphic designed with `https://alexlenail.me/NN-SVG/`

operation to each component:

$$\boldsymbol{y} = \boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)} + \boldsymbol{w}^{(1)} \tag{2.51}$$

where $\boldsymbol{y} \in \mathbb{R}^m$ is an intermediate result, $\boldsymbol{W}^{(1)} \in \mathcal{M}_{m\times n}$ is a matrix containing where its components are called weights, and $\boldsymbol{w}^{(1)} \in \mathbb{R}^m$ is a vector called bias. To get the value at the next layer we then perform an operation to each k component of $\boldsymbol{y}$ to get:

$$x_k^{(2)} = \sigma(y_k). \tag{2.52}$$

Although each operation performed to get the next layer are simple (see figure 2.5) a combination of multiple layers can generate very complex systems that can be very useful for a wide range of applications.

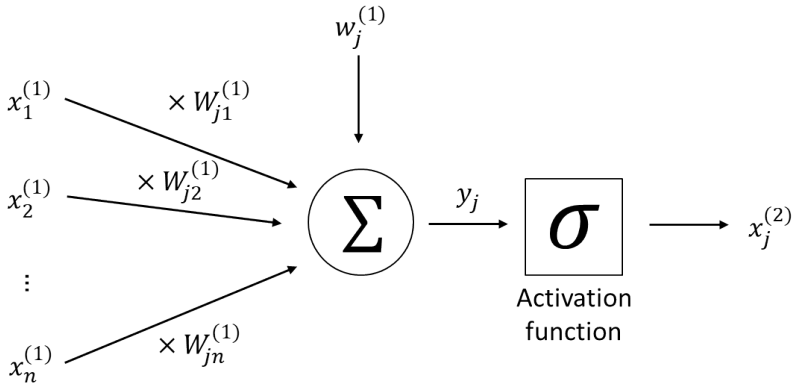


Figure 2.5: Scheme of the operations done to achieve each value for the following neuron

The operator σ is called the activation function and there are multiple options. The first activation function to be tested was the binary step which is inspired by the threshold activation in a neuron. If the signal received by the neuron surpasses a threshold ($y > 0$) then the activation function returns 1, whereas if it does not it returns 0. However there has been a development into the use of different activation functions. The most common ones used in applications are the sigmoid activation function and due to its simplicity and fast evaluation the reLU activation function (see figure 2.6).

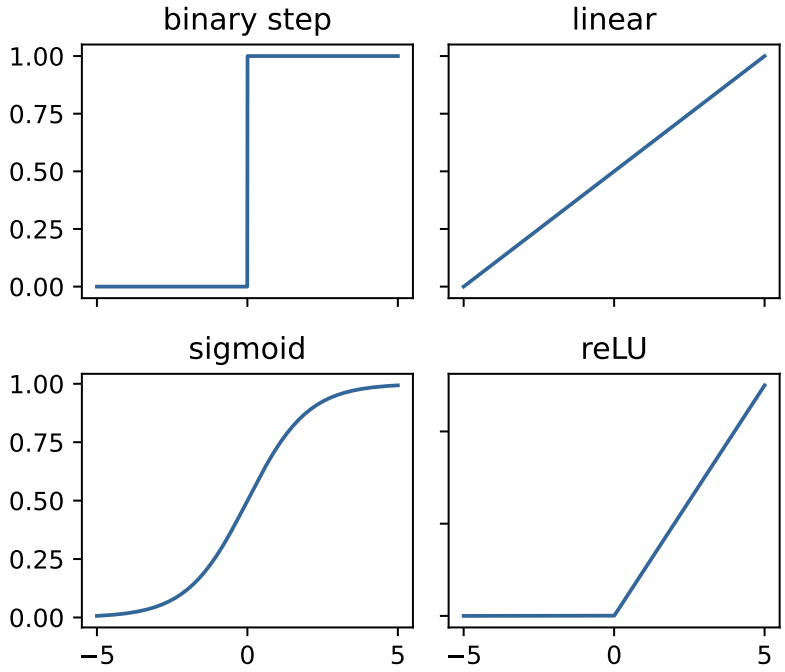


Figure 2.6: Different activation functions.

We now know how to evaluate a NN, however we have to determine all the weights and biases. In supervised learning, after evaluating the NN, the output $\boldsymbol{x}_{(n)}$ is compared to a desired output $\boldsymbol{x}$ and a cost function $\mathcal{S} \in \mathbb{R}$ is constructed to basically return a large value (scalar) if the result is too different from the desired result and a low variable if the error is small:

$$\mathcal{S} = \mathcal{S}(\boldsymbol{x}_1 - \boldsymbol{x}^N_{NN}, \boldsymbol{x}_2 - \boldsymbol{x}^N_{NN}, \cdots, \boldsymbol{x}_{samples} - \boldsymbol{x}^N_{NN}). \tag{2.53}$$

Some interesting cost functions are the euclidean norms of the error between the achieved and the desired output:

$$\mathcal{S}_n = \sum_{k=1}^{samples} \left\| \boldsymbol{x}_k - \boldsymbol{x}^N_{NN} \right\|_n \tag{2.54}$$

This cost function can give us an insight into when we have a better model or a worse model. Our objective is to minimise the cost function $\boldsymbol{S}$ for all the training data that we have. Although some clarification regarding the cost function and training we have an insight into what the training is about: minimising a cost function with respect to weights and biases.

2.4.2 Optimizers

To reach the minimum value for the cost function $\mathcal{S}$, we have to use an optimizer. In machine learning these algorithms are gradient based algorithms. These types of algorithms search for the minimum by following the direction in the $\boldsymbol{x}$ domain in which the cost function reduces the most, which locally is opposite direction to the gradient. For a k iteration, the weights and biases $\boldsymbol{x}_k$ are updated as:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \eta \nabla \mathcal{S}_k \tag{2.55}$$

where $\nabla \mathcal{S}_k$ is the gradient of the cost function at evaluated at $\boldsymbol{x}_k$, and η called learning rate sets how large these steps should be. The problem with eq. (2.55) is that the evaluation of the gradient is the summation of the gradient of the NN for every sample. When we compute the gradient accurately taking into account all the samples we have a Batch Gradient Descent (BGD).

To avoid all the cost related to the gradient computation it is possible to approximate the gradient by evaluating just a piece of the sample. If we approximate the gradient by evaluating the gradient of a random element from the sample we have what is called as Stochastic Gradient Descent (SGD) [47]:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \eta \nabla \mathcal{S}_k^j \tag{2.56}$$

where $\nabla \mathcal{S}_k^j$ stands for the gradient at iteration k from just the sample j. Due to the fact that the gradient is not actually defined it will have much more error than in the case of the Batch Gradient Descent, however the evaluation is much faster (see figure 2.7).

Both algorithms are extreme cases, in practice it is common to use an intermediate approach and take a some samples to approximate the gradient. The number of samples used is called batches and the number of training iterations is called epochs.

A common problem to appear is when the gradient is very close to zero in one direction in comparison to the rest. It is very likely that the scheme suffers oscillation in a perpendicular direction to the relevant one because the gradients are much larger in those perpendicular directions. A very common solution to those problems relies in what is called as momentum [48, 49, 50]. With the same concept of a ball rolling down a hill that gains momentum in the falling direction, some algorithms take into account

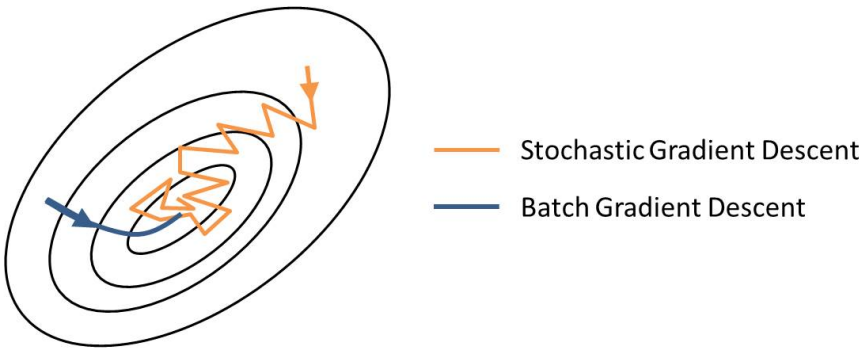


Figure 2.7: Typical behaviour for the SGD and BGD in a two variable problem. Since the SGD does not approximate the gradient properly it does not find the minimum as in the BGD, however the evaluation is much faster.

the downhill from previous steps and reduce oscillations and increase the drop in the relevant direction.

Momentum keeps information from previous steps that give an insight into what has been the “mean" gradient direction. In a very general case, to do this we need an auxiliary vector $\boldsymbol{y}$ and a new parameter γ to have:

$$\boldsymbol{y}_k = \gamma \boldsymbol{y}_{k-1} + \eta \nabla \mathcal{S}_k \tag{2.57}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{y}_k \tag{2.58}$$

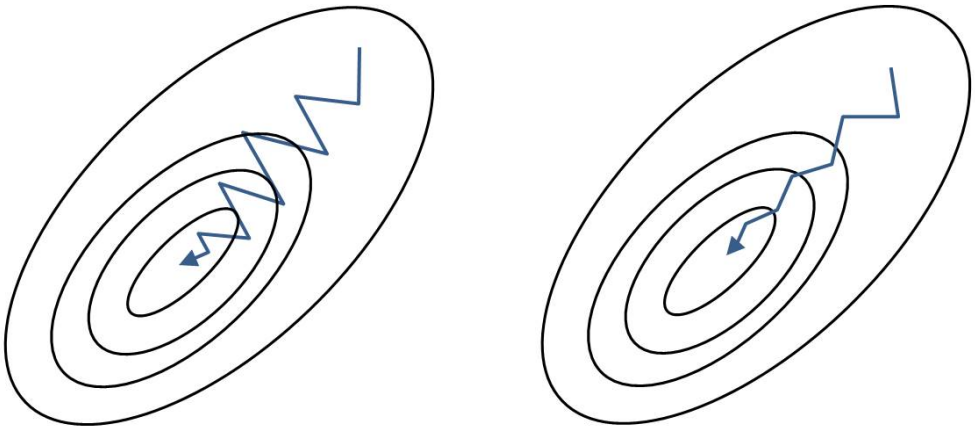
The parameter γ is in the range of zero to one. If γ is equal to zero then the method is equal to the previous ones and no momentum is built. As we increase the value of γ, the vector $\boldsymbol{y}$ is more influenced by the previous gradient directions. If we for instance had pure oscillations in a particular direction, the effect of having the history is to “cancel" that swinging (see figure 2.8).

There are many optimisers that take advantage of this momentum approach with different variants, such as: SGD, Adam, RMSprop, Adadelta etc. Perhaps the most common and one of the best algorithms in general cases is the Adam optimiser [51].

2.4.3 Backpropagation

We have presented different types of Gradient Based optimiser, but how do we actually compute the gradient of a cost function in practice? Instead of using classical methods such as finite differences or other methods as complex step differentiation we will use backpropagation which is a special case of automatic differentiation [52, 53].

The backpropagation algorithm performs an exact differentiation of the cost function



(a) Model without momentum (b) Model with momentum

Figure 2.8: Scheme to show the effect of momentum in the optimiser.

$\mathcal{S}$ by taking analytical derivatives. For that, it is interesting to recover the mathematical expression for the process inside the NN:

$$\boldsymbol{x}^{(n)} = \sigma(\boldsymbol{w}^{(n-1)} + \boldsymbol{W}^{(n-1)}\boldsymbol{x}^{(n-1)}), \tag{2.59}$$

but the same expression can be done with $\boldsymbol{x}^{(n-1)}$ as it is:

$$\boldsymbol{x}^{(n-1)} = \sigma(\boldsymbol{w}^{(n-2)} + \boldsymbol{W}^{(n-2)}\boldsymbol{x}^{(n-2)}), \tag{2.60}$$

which can be introduced in eq. (2.59) to get:

$$\boldsymbol{x}^{(n)} = \sigma(\boldsymbol{w}^{(n-1)} + \boldsymbol{W}^{(n-1)}\sigma(\boldsymbol{w}^{(n-2)} + \boldsymbol{W}^{(n-2)}\boldsymbol{x}^{(n-2)})). \tag{2.61}$$

This process can be done until we get an explicit expression where $\boldsymbol{x}^{(n)}$ depends only on all the weights, biases and the input to the NN, $\boldsymbol{x}^{(1)}$. To compute the backpropagation algorithm we apply the chain rule. If the variable for the cost function is the error $\boldsymbol{e}$:

$$\boldsymbol{e} = \begin{bmatrix} e_1 & e_2 & \cdots & e_N \end{bmatrix}^T \tag{2.62}$$

The derivative of the cost function with respect to some W_{ij}^k weight will be:

$$\frac{\partial \mathcal{S}}{\partial W_{ij}^k} = \frac{\partial \mathcal{S}}{\partial e_1}\frac{\partial e_1}{\partial W_{ij}^k} + \frac{\partial \mathcal{S}}{\partial e_2}\frac{\partial e_2}{\partial W_{ij}^k} + \cdots \frac{\partial \mathcal{S}}{\partial e_N}\frac{\partial e_N}{\partial W_{ij}^k} \tag{2.63}$$

but there is not a simple expression for $\partial e_k / \partial W_{ij}^k$, and for each spatial derivative we can apply the chain rule up to the next layer:

$$\frac{\partial e_1}{\partial W_{ij}^k} = \frac{\partial x_1^{(n)}}{\partial W_{ij}^k} = \frac{\partial \sigma}{\partial x_1^{(n-1)}}\frac{\partial x_1^{(n-1)}}{\partial W_{ij}^k} + \cdots + \frac{\partial \sigma}{\partial x_N^{(n-1)}}\frac{\partial x_N^{(n-1)}}{\partial W_{ij}^k}. \tag{2.64}$$

Note that the error is:

$$\boldsymbol{e} = \boldsymbol{x}^{(n)} - \boldsymbol{x} \tag{2.65}$$

where $\boldsymbol{x}$ is the desired output and $\boldsymbol{x}^{(n)}$ is the result from the NN with n layers. That justifies the fact that:

$$\frac{\partial e_1}{\partial W_{ij}^k} = \frac{\partial (\boldsymbol{x}^{(n)} - \boldsymbol{x})}{\partial W_{ij}^k} = \frac{\partial x_1^{(n)}}{\partial W_{ij}^k}. \tag{2.66}$$

After this clarification we can do the same process for each derivative in eq. (2.64) until we get to $k+1$ layer in which the derivatives with respect to the weights and biases is explicitly calculated as:

$$\frac{\partial x_1^{(k+1)}}{\partial W_{ij}^k} = \frac{\partial \sigma}{\partial x_1^{(k)}} x_j^{(k)}. \tag{2.67}$$

The backpropagation allow us to have an analytical computation for each derivative and compute the gradient in a fast and exact form and this can be done thanks to knowing the analytical expression to compute $\mathcal{S}$ from the input $\boldsymbol{x}^{(1)}$. To have a deeper understanding read [53] or [54].

2.4.4 Neural network architectures

There are multiple architectures that have been developed for different applications which differ according to the connection of the neurons and the operations performed, many of them are presented in figure 2.12 from [55]. Three of the most common architectures that have place in this work (or could have in the future) are [56, 57]:

- Feedforward Neural Network (FNN)
- Recurrent Neural Network (RNN)
- Convolutional Neural Network (CNN)

Feedforward neural network

FNN is the classical type of a deep NN in which the information goes from the input to the output without any closed loop connection neither storage any previous information. The classical FNN has an input layer that storages information as a vector and after all the operations it gives an output in the form of another vector (see figure 2.4). In this work we have used this type of NN because they are implemented in the library Fortran to Keras Bridge (FKB) [58].

They have been used in a wide range of applications and they are the most simple type of NN, their main drawback for our application is that we will loose spatial information due to the transformation from a spatial tensor to a vector and a lost in the temporal order. That is the reason for presenting both RNN and CNN, becasue they keep temporal and spatial order intact respectively and they should be considered in future alternatives.

Recurrent neural network

RNNs are a type of NN that are prepared for time series, i.e. several times in a row [59]. They differ from FNNs in the fact that, apart from the input and output, the NN needs an extra vector from the previous evaluation to compute the output. Also a new vector will be generated to become an entry for future evaluations. In a simple case and given a certain iteration, an input $\boldsymbol{x}$ and a vector $\boldsymbol{z}$ enter the neural network, then the output $\boldsymbol{y}$ is calculated as well as another vector $\boldsymbol{z}$, which will be used as input in the next evaluation (see figure 2.9).

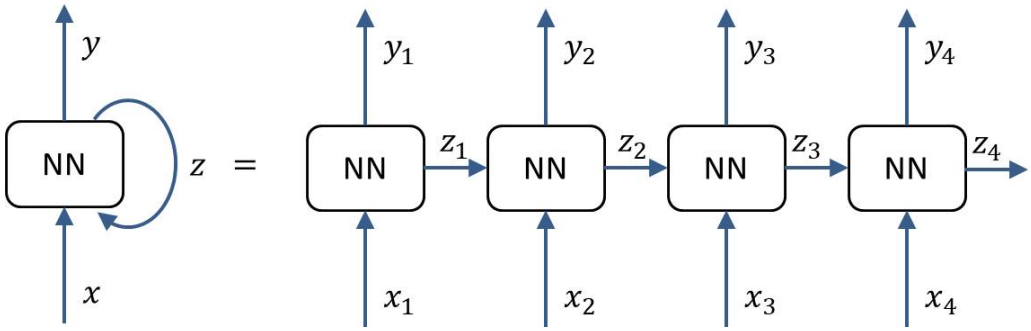


Figure 2.9: Recurrent Neural Network scheme. The box represents the NN, whereas x is the input and y the output. There is a new variable exported from the NN z that is another input when evaluating the NN in the next step.

Although we have presented a simple model (see figure 2.9), there are many alternatives into how the previous information is sent to forward evaluations: one to one, one to many, many to one or many to many [59, 60]. These type of NN are typically used for speech recognition and language processing since for instance in a sentence, words are linked together an the order is important [61]. But they can also be used for time series as shown in [62] and they do not loose information related to the order in the time series, which is the reason why, in future work when changing the type of NN, RNNs should be considered.

Convolutional neural network

CNNs are a type of FNN that is mainly used in the processing of data where its spatial placement is relevant, such as images. In this type of NN, in order to pass information between layers, a mathematical operation known as convolution is performed. To perform the convolution, the concept of kernel is introduced, which is a matrix that moves through each node of the layer and is scalarly multiplied by the nodes adjacent to a the selected one (see figure 2.10).

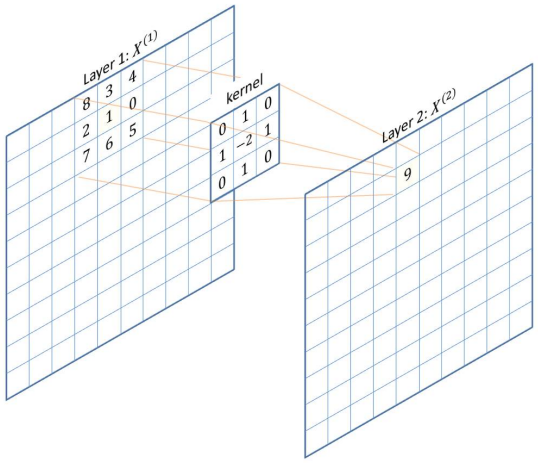


Figure 2.10: Convolution step in a CNN. Each value of the kernel multiplies a value in layer 1 and then they are summed up. The value in the second layer would be: $0\times 8+1\times 2+0\times 7+1\times 3-2\times 1+1\times 6+0\times 4+1\times 0+0\times 5=9$. The same process is done for the rest of the nodes.

CNN are mainly used on image processing (see figure 2.11) and classification. However, they have also been used in fluid mechanics as in [63, 64] due to the fact that they keep spatial information intact and that they are very useful at finding spatial patterns.

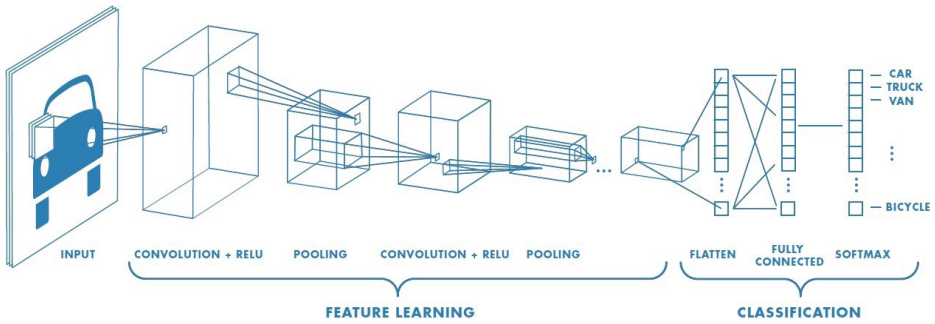


Figure 2.11: CNN architecture for image classification, from [56]

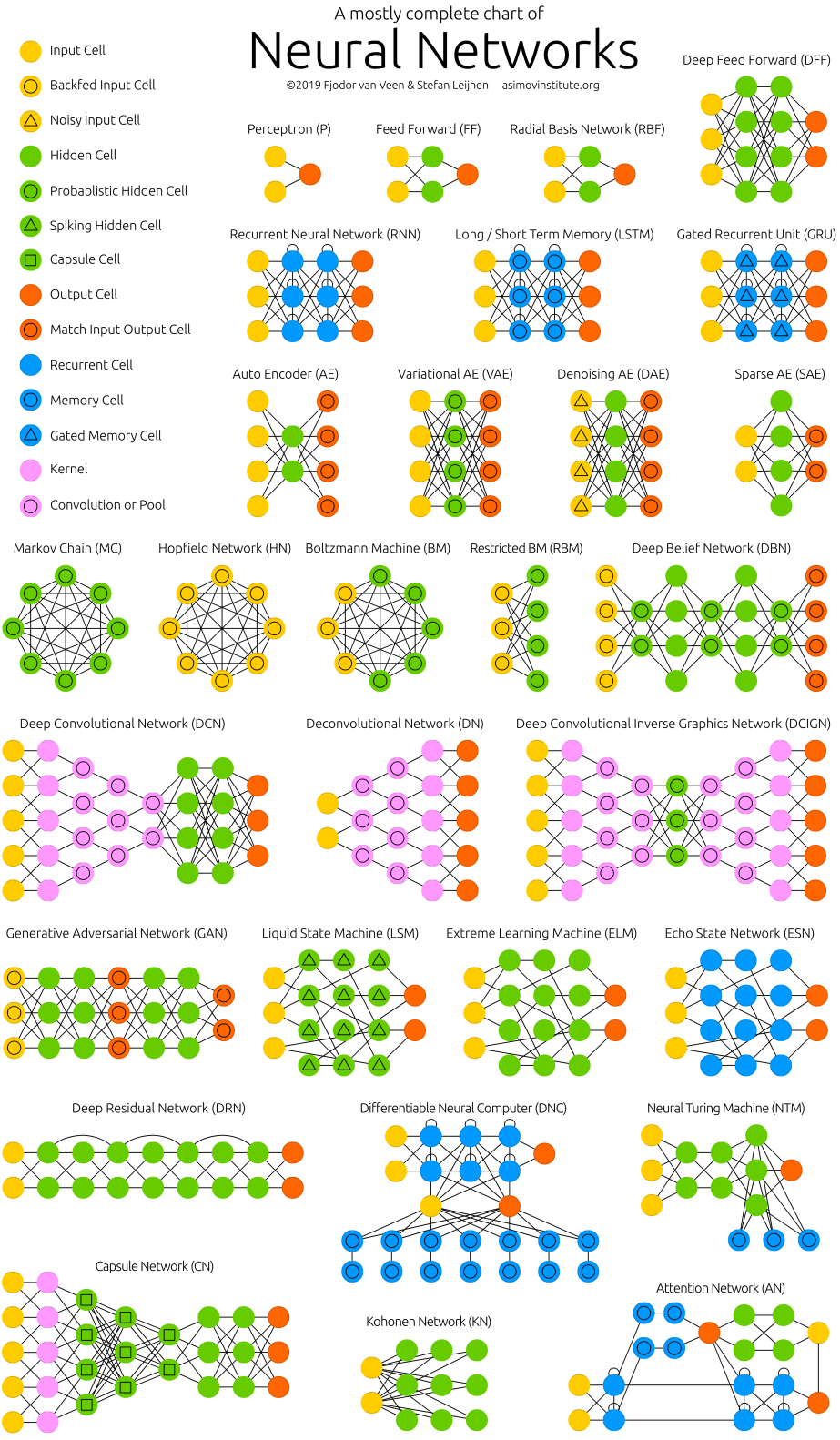


Figure 2.12: Types of NN. Figure from [55]

CHAPTER 3

Accelerating High Order Discontinuous Galerkin

3.1 Methodology

Let us consider a time-marching system of PDEs with its boundary conditions and its initial condition, that can be expressed in a general form as:

$$\frac{\partial q}{\partial t} = \mathcal{L}(q;t), \tag{3.1}$$

$$\mathcal{R}(q, \partial\Omega; t) = 0, \quad q(\boldsymbol{x};0) = q_0. \tag{3.2}$$

The state variables are included in $q(\boldsymbol{x};t) : \mathbb{R}^D \times \mathbb{R} \rightarrow \mathbb{R}^H$, where D is the spatial dimension and H is the number of state variables. $\mathcal{L} : \mathbb{R}^H \times \mathbb{R} \rightarrow \mathbb{R}^H$ is the spatial operator and $\mathcal{R}$ is the boundary conditions operator applied at the boundary of the domain $\partial\Omega$. Eq. (3.1) can represent a variety system of equations, for instance the one dimensional Burgers' equation, the heat equation, the compressible Navier-Stokes equations[1] etc. For the Burgers equation, $H = 1$:

$$q = u \tag{3.3}$$

and for the compressible Navier-Stokes ($H = 5$) the state variables are:

$$q = \begin{bmatrix} \rho & \rho u & \rho v & \rho w & \rho e \end{bmatrix}^T, \tag{3.4}$$

[1] Mass conservation, energy conservation and Navier-Stokes (momentum conservation).

where ρ is the density, u, v and w are the velocity in the x, y and z axis respectively and e is the total energy. For more details on the Navier-Stokes formulation we are using, see appendix A.

In order to solve the system of PDEs we follow the method of lines [1], starting by the spatial discretization in which we approximate the continuous space variables are approximated by a discrete variables that evolve in time:

$$q(\boldsymbol{x};t) \approx \sum_{k=1}^{N} \hat{q}_k(t)\phi_k(\boldsymbol{x}). \tag{3.5}$$

The spatial basis functions $\phi_k(\boldsymbol{x})$ will depend on the type of discretization performed. In this case we will focus on the high order discontinuous Galerkin discretization with a nodal formulation (Gauss-Legendre nodes) [13, 31, 32]. The coefficients $\hat{q}_k$ are yet time dependent whereas the basis functions $\phi_k(\boldsymbol{x})$ are known Lagrange polynomials. Before moving on, for simplicity in the notation, we can arrange all the degrees of freedom $\hat{q}_k$ into a vector:

$$\boldsymbol{q}(t) = \begin{bmatrix} \hat{q}_1(t) & \hat{q}_2(t) & \cdots & \hat{q}_N(t) \end{bmatrix}^T, \tag{3.6}$$

and the basis functions as well:

$$\boldsymbol{\phi}(\boldsymbol{x}) = \begin{bmatrix} \phi_1(\boldsymbol{x}) & \phi_2(\boldsymbol{x}) & \cdots & \phi_N(\boldsymbol{x}) \end{bmatrix}^T, \tag{3.7}$$

to leave a much simpler expression for eq. (3.5) in vector form as:

$$q(\boldsymbol{x};t) \approx \sum_{k=1}^{N} \hat{q}_k(t)\phi_k(\boldsymbol{x}) = \boldsymbol{q}(t)\cdot\boldsymbol{\phi}(\boldsymbol{x}). \tag{3.8}$$

With the approximation from eq. (3.8), the system of PDEs in eq. (3.1) turns into a system of ODEs where the unknowns are the components from the coefficient vector $\boldsymbol{q}$. A general formulation for the system of ODEs after the spatial discretization is:

$$\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{q};t), \tag{3.9}$$

which is our starting point for the methodology. Before integrating eq. (3.9), let us remark that any spatial operator that acts linearly on the continuous variable q can be discretized into a linear operator acting on the discretized variables[2]:

$$\overline{q} = \mathcal{G}(q) \rightarrow \overline{\boldsymbol{q}} = \mathbf{G}\boldsymbol{q}. \tag{3.10}$$

Here $\mathcal{G}$ represents a linear and continuous spatial operator and $\mathbf{G}$ is the discretized spatial operator (matrix). This operator can represent a linear filter operation that takes the high order solution and only recovers the low order modes. $\overline{q}$ is a new variable that,

[2] As long as the interpolation is linear with respect to $\boldsymbol{q}$

after the spatial discretization, is characterized by its degrees of freedom $\overline{\boldsymbol{q}}$. It can be seen that the system of ODEs responsible for the evolution of this new variable $\overline{\boldsymbol{q}}$ can be obtained by multiplying eq. (3.9) with the linear operator $\boldsymbol{G}$:

$$\frac{\mathrm{d}\overline{\boldsymbol{q}}}{\mathrm{d}t} = \mathbf{G}\boldsymbol{f}(\boldsymbol{q};t). \tag{3.11}$$

Unfortunately, the discretized operator $\boldsymbol{f}$ is non-linear in general, and thus, to compute the time evolution for this new variable we need the original variable $\boldsymbol{q}$ to evaluate its evolution accurately $\boldsymbol{f}(\boldsymbol{q};t)$ and then filter it. Before trying to fix this problem we can temporally discretise (integrate) both systems of ODEs eq. (3.9) and eq. (3.11) between a time step t_n and t_{n+1}, with $\Delta t = t_{n+1} - t_n$ to get:

$$\boldsymbol{q}^{n+1} = \boldsymbol{q}^n + \Delta t\, \boldsymbol{p}(\boldsymbol{q}^n;t_n), \tag{3.12}$$

$$\overline{\boldsymbol{q}}^{n+1} = \overline{\boldsymbol{q}}^n + \Delta t\, \boldsymbol{G}\boldsymbol{p}(\boldsymbol{q}^n;t_n). \tag{3.13}$$

Note that the operator $\boldsymbol{p} : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$ depends on the spatial discretisation but also on the temporal scheme. If we are interested in the filtered variable $\overline{\boldsymbol{q}}$ because our goal does not require the higher order modes from $\boldsymbol{q}$, we first must know that the system given by eq. (3.13) will not evolve accurately by simply using the low order solution given by:

$$\overline{\boldsymbol{q}}^{n+1} \neq \overline{\boldsymbol{q}}^n + \Delta t\, \boldsymbol{p}^*(\overline{\boldsymbol{q}}^n;t_n), \tag{3.14}$$

where $\boldsymbol{p}^*$ is the result of the spatial and temporal discretization of the operator ($\mathcal{L}$) for the filtered variable ($\overline{\boldsymbol{q}}$). Note that the mathematical dimensions for this operator are different to operator $\boldsymbol{p}$, but the origin and the meaning of both operators is the same.

Both of them represent the same physical operator, however the evaluation of the filtered variable is not equal to the evaluation of the unfiltered equation because the high order modes affect the low order modes. We can add a source term $\boldsymbol{s}^n$ to the operator $\boldsymbol{p}^*(\overline{\boldsymbol{q}}^n;t_n)$ such that:

$$\boldsymbol{G}\boldsymbol{p}(\boldsymbol{q}^n;t_n) = \boldsymbol{p}^*(\overline{\boldsymbol{q}}^n;t_n) + \boldsymbol{s}^n, \tag{3.15}$$

However the true evaluation for the source term can only be done by having the high order variable $\boldsymbol{s}(\boldsymbol{q}^n;t)$, which has the information on the low and high order modes. We are interested in this computation from data that we already have or that we are generating for a latter process in the methodology, leaving:

$$\boldsymbol{s}^n = \boldsymbol{G}\boldsymbol{p}(\boldsymbol{q}^n;t_n) - \boldsymbol{p}^*(\overline{\boldsymbol{q}}^n;t_n). \tag{3.16}$$

What we want from now on is to generate a correlation between this second term $\boldsymbol{s}^n$ and the filtered variable $\overline{\boldsymbol{q}}^n$ with a function:

$$\boldsymbol{s}^n \longleftrightarrow \boldsymbol{s}^n. \tag{3.17}$$

If we manage to get a good approximation for the source term with a correlation from eq. (3.17), the evolution for filtered variables would be accurate and have the following expression:

$$\overline{\boldsymbol{q}}^{n+1} = \overline{\boldsymbol{q}}^{n} + \Delta t\, \boldsymbol{p}^{*}(\overline{\boldsymbol{q}}^{n}; t_n) + \Delta t\, \boldsymbol{s}^{n}(\overline{\boldsymbol{q}}^{n}), \tag{3.18}$$

which is a closed formulation to calculate $\overline{\boldsymbol{q}}^{n+1}$ with information from only the filtered variable $\overline{\boldsymbol{q}}^{n}$. If the source term is good enough we can reduce the computational time for two reasons:

1. Reduction in degrees of freedom.
2. In explicit schemes, there is less restriction in the time step due to the CFL (Courant–Friedrichs–Lev number).

Both reasons make us have a lower number of iterations and much faster evaluations for each time step. Now that we have presented the mathematical formulation and the motivation on why we are interested in this approach, from here in the rest of the section we will clarify the algorithm of the methodology, perform some brief analysis on the errors and give a little insight into the filtering operation and how we model the forcing term.

3.1.1 Algorithm

For this methodology we will use a NN to perform the correlation between the forcing term $\boldsymbol{s}^{n}$ and the filtered state variable $\overline{\boldsymbol{q}}^{n}$. For that we have three mandatory steps (see figure 3.2) we will explain right away:

1. High order evolution
2. Neural network train
3. Low order evolution

1. High order evolution. In order to train/construct this NN we need some data from the simulation. Hence, we need one first step that evolves the high order solution and the low order solution, so that we can compute the definition for the forcing term $\boldsymbol{s}^{n}$ from eq. (3.16) and storage it together with the state variable $\overline{\boldsymbol{q}}^{n}$. This first step will be called *High order evolution*. We want to recall that besides a lower number of degrees of freedom, the filtered variable will have a much lower Δt time restriction ($\Delta t_{LO} > \Delta t_{HO}$, where Δt_{LO} is the time step for the low order solution and Δt_{HO} is the time step for the high order solution[3]), and thus for integrating both solutions in one time step Δt_{LO}

[3]For explicit time schemes which is what we consider for this work

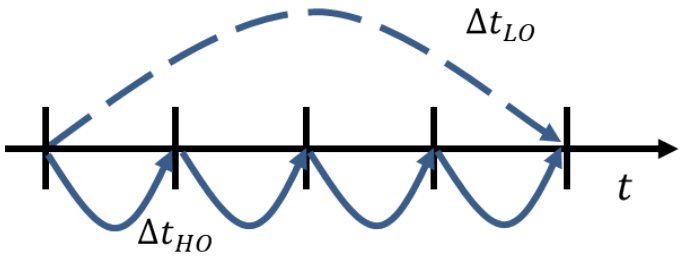


Figure 3.1: Sketch of time advancement for low and high order discretisations, filtered and unfiltered respectively. The low order time-steps (modelled with NNs) Δt_n are larger that the High Order time step Δt_m.

we need only one step for the low order variables and multiple steps for the high order solution (see figure 3.1).

2. Neural network train. Followed by that, the next step would be to train the NN, which will be explained with more details in a following section.

3. Low order evolution. Once the NN is trained, the third step would be to integrate the low order solution with the new forcing term we have developed, using less degrees of freedom and a larger time step in comparison to the high order solution (see figure 3.2 and algorithm 1).

Algorithm 1: Pseudo-algorithm of the proposed methodology. Modified version from [23]

`// High order evolution`
for *n in number of high order iterations* **do**
  match low order with filtered high order $\overline{\boldsymbol{q}}^n_{NN} = \overline{\boldsymbol{q}}^n_{HO}$
  save and advance $\overline{\boldsymbol{q}}^n_{NN} \rightarrow \overline{\boldsymbol{q}}^{n+1}_{NN}$ (without correction)
  for *m in* $\Delta t_{LO}/\Delta t_{HO}$ **do**
    advance $\overline{\boldsymbol{q}}^{m+1}_{HO}$
  filter $\boldsymbol{q}^{n+1}_{HO}$ to obtain $\overline{\boldsymbol{q}}^{n+1}_{HO}$
  compute and save $\mathbf{s}_n = (\overline{\boldsymbol{q}}^{n+1}_{HO} - \overline{\boldsymbol{q}}^{n+1}_{NN})/\Delta t_n$
`// NN training`
Train NN model as $\mathbf{s}_n = \mathbf{s}_n(\overline{\boldsymbol{q}}^n_{NN})$
`// Low order evolution`
for *n in number of low order iterations* **do**
  advance $\overline{\boldsymbol{q}}^{n+1}_{NN}$ (without correction)
  compute $\mathbf{s}_n = \mathbf{s}_n(\overline{\boldsymbol{q}}^n_{NN})$
  correct $\overline{\boldsymbol{q}}^{n+1}_{NN} \leftarrow \overline{\boldsymbol{q}}^{n+1}_{NN} + \Delta t_n \mathbf{s}_n$

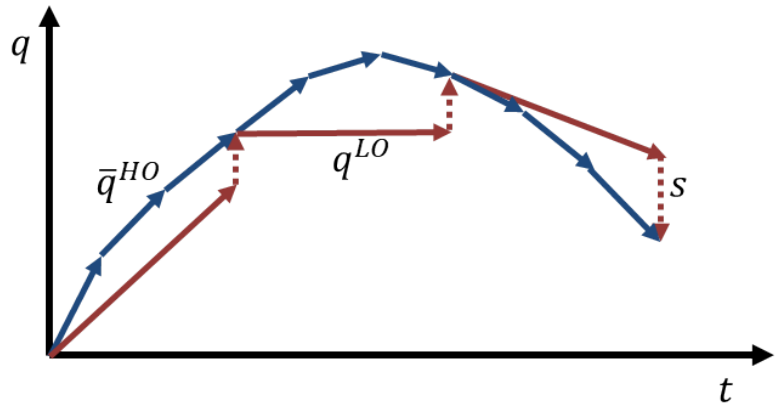


Figure 3.2: Sketch to visualise the process. In blue we see the evolution of the high order variables, which evolve several times to match one single time step from the low order solution. In red and continuous line we find the evolution of the low order variables given by $\boldsymbol{p}*$, which needs to be corrected by the source term in red and discontinuous line.

These three steps can be followed by an optional step. This new step would include a super-resolution neural network that would recover the high order solution from the low order solution. Although we have used a deep NN to test it on the Burgers' equation, there are different approaches such as Generative Adversarial Networks (GANs) that look very promising for this type of problems as shown by Güemes et al. [22]. Although this step is very interesting, since it allows us to reconstruct the high order modes, for this work we will test it only on the Burgers' case.

3.1.2 Filtering

We have mentioned the concept of filtering, high order and low order modes but we still have not given an accurate definition for these terms. To understand these ideas, we will expand the high order solution in a Legendre series, we will refer to the high order solution if we keep all the modes. If we only keep some terms of the series we will referring to it as the low order solution. Finally, the process of projecting the solution into the Legendre series and just keeping the high order modes is what we call filtering. Mathematically, we express the projection into the basis and keep the low order term as:

$$\bar{q} = \sum_{k=0}^{P_{LO}} \frac{\varphi_k}{\|\varphi_k\|^2} \int_{\Omega_e} q\varphi_k d\Omega, \tag{3.19}$$

where $\varphi_k(\boldsymbol{x})$ is a basis function:

$$\varphi_k(\boldsymbol{x}) = L_i(x)L_j(y)L_n(z), \tag{3.20}$$

constructed with the product of Legendre polynomials. For simplicity in the notation we use $\varphi_k(\boldsymbol{x})$ instead of the product of the Legendre polynomials. The norm of the basis functions can be defined by:

$$\|\varphi_k\|^2 = \int_{\Omega_e} \varphi_k(\boldsymbol{x})\varphi_k(\boldsymbol{x})d\Omega, \tag{3.21}$$

where Ω_e represents the domain of each element. For the discretized variable, operation performed in eq. (3.19) is done with Gauss quadrature which is exact for the considerations done for the discretized variable [31]. The algebraic result from these quadrature can be seen as a linear matrix product performed on the coefficient vector $\boldsymbol{q}$

$$\overline{q} = \boldsymbol{G}\boldsymbol{q}. \tag{3.22}$$

The filter matrix $\boldsymbol{G}$ can be decomposed as the product of two different matrix. The first one transforms the nodal basis (Lagrange) into the modal basis (Legendre), $\boldsymbol{M}$. The second matrix takes only the first modes and evaluates them at the nodes of the grid, $\boldsymbol{N}$, leaving:

$$\bar{u} = \boldsymbol{N}\boldsymbol{M}\boldsymbol{q}, \tag{3.23}$$

where, the components for each matrices are:

$$M_{ij} = \frac{1}{\|\varphi_i\|^2} \int_{\Omega_e} \varphi_i(\boldsymbol{x})\phi_j^{HO}(\boldsymbol{x})d\Omega, \tag{3.24}$$

and

$$B_{ij} = \varphi_i(\boldsymbol{x}_j^{LO}). \tag{3.25}$$

The basis function ϕ_j^{HO} is a product from the Lagrange polynomials from the high order mesh:

$$\phi_n^{HO} = \ell_i^{HO}(x)\ell_j^{HO}(y)\ell_k^{HO}(z) \tag{3.26}$$

and ℓ^{HO} represents a Lagrange polynomial with zeros in the Gauss nodes from the high order mesh. Note that these Lagrange polynomials depend on the nodal basis we are using, that is the reason we insist on the super-index HO, because the low order polynomials for the Lagrange series is different.

Another important thing to notice is that the dimension for the matrices, in which $M_{ij} \in \mathcal{M}_{HDof \times HDof}$ and $B_{ij} \in \mathcal{M}_{LDof \times HDof}$ where $HDof$ is the number of degrees of freedom in the high order variables and $LDof$ is the number of degrees of freedom in the low order solution (see figure 3.3)

We have shown how the filter operation can be expressed as a matrix product for the discretised variables. This matrix is decomposed into a product from two different matrix. In practice we compute both $\boldsymbol{M}$ and $\boldsymbol{B}$, perform the matrix product and only store $\boldsymbol{G}$.

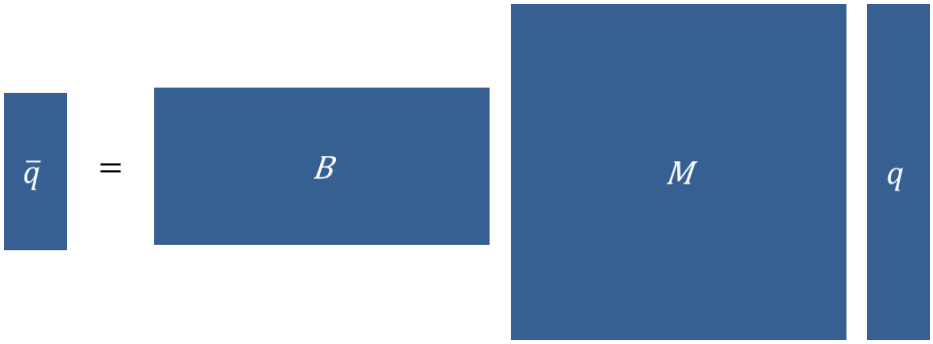


Figure 3.3: Scheme for the matrix product to perform the filtering operation. Matrix M times the state variables returns all the modes from the solution, whereas the product with B recovers the evaluation of the low order modes in the Gauss nodes from the low order solution.

3.1.3 Force modelling

In order to model the forcing term we have chosen a neural network. This neural network is trained in python using keras and then all the weights and biases are exported to Fortran using *Fortran to keras bridge*, a very useful tool from [58]. From now, all the operations presented in *Force modelling* are done for each element. With this in mind, to construct the forcing term we have:

$$\boldsymbol{s}_n \approx \boldsymbol{s}_{NN}(\overline{\boldsymbol{q}}^n). \tag{3.27}$$

For the neural network we have chosen a deep neural network, fully connected and with *relu* and *linear* activation functions (see figure 3.12). In future work we might include more complex architectures involving neighbours information. For the time been we have used this approach because the high order code *horses3d* [65] is in FORTRAN and to evaluate the NN we are using the library in Fortran To Keras Bridge from [58] that allows us to have a fast evaluation of the NN and only provides us the deep NN architecture.

The specific layout for the NN has N_{la} number of layers, where the first and the last layers are *linear* and the rest can be either *linear* or *relu*. To explain the reason we have forced the first and last layer to be linear we first will explain the training process.

During the first step where we generate data *High Order evolution* to train the neural network. Afterwards we have N_{train} number of fields for the state variables and N_{train} number of forcing fields (see tag 1 in figure 3.4). We can now choose whether to have a different NN for each element, or to have one single NN for every element. In our case we have chosen the second option, the same NN for every element.

For the training process we arrange in each time all the elements from both fields (see tag 2 in figure 3.4). We then arrange all the time steps together leading to the data

set (see tag 3 in figure 3.4). The aim is that a NN can get any solution as in the left box in tag 3 (figure 3.4) and then return the forcing field.

The next problem with this is that some variables have different scales than others. This means that the energy equation will have much more weight during the optimization process since its values are much larger than for instance the density or the velocity. In order to avoid this problematic we perform a scaling in each node and each variable. In practice all the information for one element is arranged in a vector of size N_{vec}:

$$N_{vec} = N_{eq} \times (P_{LO} + 1)^3, \tag{3.28}$$

where N_{eq} is the number of equations we have. This vector can be re-scaled so that each value in the vector from the training data goes from 0 to 1 which is a linear operation:

$$\hat{\boldsymbol{q}} = \boldsymbol{A}\overline{\boldsymbol{q}} + \boldsymbol{c}, \tag{3.29}$$

where $\hat{\boldsymbol{q}}$ is the re-scaled state variables (each component goes from 0 to 1) and $\boldsymbol{A}$ is a diagonal matrix with $\boldsymbol{c}$ a vector. The same process can be done with the output:

$$\boldsymbol{s} = \boldsymbol{B}\hat{\boldsymbol{s}} + \boldsymbol{k}. \tag{3.30}$$

By choosing linear activation functions in the first and last layer, after the training process, we can embed these two scaling operations inside the NN. If the input to the NN is $\hat{q}^{(0)}$ and the next layer is $\hat{q}^{(1)}$ calculated by the following linear layer:

$$\hat{\boldsymbol{q}}^{(1)} = \boldsymbol{W}^{(1)}\hat{\boldsymbol{q}}^{(0)} + \boldsymbol{w}^{(1)}, \tag{3.31}$$

but the scaled entry can be written as in eq. (3.29):

$$\hat{\boldsymbol{q}}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{A}\overline{\boldsymbol{q}}^{(0)} + \boldsymbol{W}^{(1)}\boldsymbol{c} + \boldsymbol{w}^{(1)}, \tag{3.32}$$

hence the new weights ($\boldsymbol{W}_*^{(1)}$) and biases $\boldsymbol{w}_*^{(1)}$ for the entry layer are:

$$\boldsymbol{W}_*^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{A}, \tag{3.33}$$

$$\boldsymbol{w}_*^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{c} + \boldsymbol{w}^{(1)}. \tag{3.34}$$

We can do the same process with the output. With the last layer in mind as follows:

$$\hat{\boldsymbol{s}}^{(N_{la})} = \boldsymbol{W}^{(N_{la})}\hat{\boldsymbol{q}}^{(N_{la}-1)} + \boldsymbol{w}^{(N_{la})}, \tag{3.35}$$

and recovering eq. (3.30), we can get the new weights and biases as:

$$\boldsymbol{W}_*^{(N_{la})} = \boldsymbol{B}\boldsymbol{W}^{(N_{la})}, \tag{3.36}$$

$$\boldsymbol{w}_*^{(N_{la})} = \boldsymbol{B}\boldsymbol{w}^{(N_{la})} + \boldsymbol{k}. \tag{3.37}$$

This shows that by having a linear layer at the first and last layers we can automatically scale the input and output inside the NN and can prevent doing this process “manualy" before evaluating the NN.

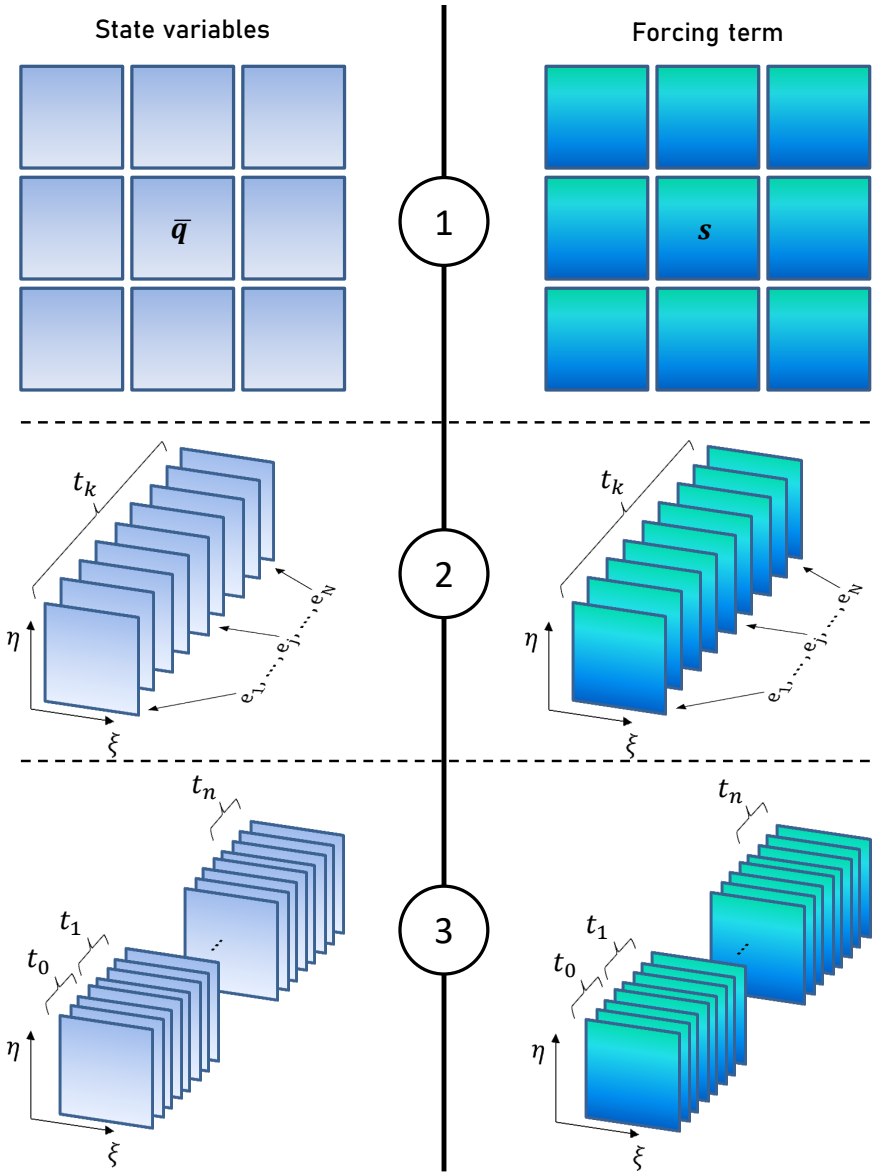


Figure 3.4: Pre-processing data for the neural network training process. In the tag number one, on the left we have the information of each element states variables for one time step, on the right we have the forcing terms. For a local methodology, the following step (tag number two) is to arrange each element in a block of elements for the time-step t_k both for the state variables (left) and forcing term (right). We then unite all the elements from all the time steps we want for both state variables and forcing term.

3.1.4 Error analysis

It is interesting to perform a simple analysis on the errors, trying to understand predict what we will see on the results. For that we would like to understand how the error is going to increase when having a modelling term or when it is null. Just for this analysis we can introduce the following variables:

$$\overline{\boldsymbol{q}}_{NN} = \overline{\boldsymbol{q}} + \varepsilon_{NN}, \tag{3.38}$$

and

$$\overline{\boldsymbol{q}}_{T} = \overline{\boldsymbol{q}} + \varepsilon_{T}, \tag{3.39}$$

where sub-index NN means it is the solution following the neural network scheme and T that the solution is simply truncated (Low order without forcing). The variable ε_{NN} and ε_{T} are the error in each case. Now, we can present the the evolution for the exact evolution in each time step for the state variable:

$$\frac{\Delta\overline{\boldsymbol{q}}}{\Delta t} = \boldsymbol{p}^{*}(\overline{\boldsymbol{q}}) + \boldsymbol{s}(\overline{\boldsymbol{q}}). \tag{3.40}$$

The evolution system without any type of modelling would look like:

$$\frac{\Delta\overline{\boldsymbol{q}}_{T}}{\Delta t} = \boldsymbol{p}^{*}(\overline{\boldsymbol{q}}_{T}), \tag{3.41}$$

whereas when using a modelling term, which is not exact, it will look like:

$$\frac{\Delta\overline{\boldsymbol{q}}_{NN}}{\Delta t} = \boldsymbol{p}^{*}(\overline{\boldsymbol{q}}_{NN}) + \boldsymbol{s}_{NN}(\overline{\boldsymbol{q}}_{NN}). \tag{3.42}$$

Using both definitions from eq. (3.38) and eq. (3.39) and introducing them into their respective evolution equations we get:

$$\frac{\Delta\overline{\boldsymbol{q}}}{\Delta t} + \frac{\Delta\varepsilon_{T}}{\Delta t} = \boldsymbol{p}^{*}(\overline{\boldsymbol{q}} + \varepsilon_{T}), \tag{3.43}$$

and

$$\frac{\Delta\overline{\boldsymbol{q}}}{\Delta t} + \frac{\Delta\varepsilon_{NN}}{\Delta t} = \boldsymbol{p}^{*}(\overline{\boldsymbol{q}} + \varepsilon_{NN}) + \boldsymbol{s}_{NN}(\overline{\boldsymbol{q}} + \varepsilon_{NN}). \tag{3.44}$$

It is possible to subtract eq. (3.40) from both eq. (3.43) and eq. (3.44) to get respectively:

$$\frac{\Delta\varepsilon_{T}}{\Delta t} = \boldsymbol{p}^{*}(\overline{\boldsymbol{q}} + \varepsilon_{T}) - \boldsymbol{p}^{*}(\overline{\boldsymbol{q}}) - \boldsymbol{s}(\overline{\boldsymbol{q}}), \tag{3.45}$$

and

$$\frac{\Delta\varepsilon_{NN}}{\Delta t} = \boldsymbol{p}^{*}(\overline{\boldsymbol{q}} + \varepsilon_{NN}) + \boldsymbol{s}_{NN}(\overline{\boldsymbol{q}} + \varepsilon_{NN}) - \boldsymbol{p}^{*}(\overline{\boldsymbol{q}}) - \boldsymbol{s}(\overline{\boldsymbol{q}}). \tag{3.46}$$

In both cases if the error is small enough in comparison to the state varibales order of magnitude, then for a few time steps it is possible to expand the equations operator $\boldsymbol{p}^*$ and $\boldsymbol{s}_{NN}$ in Taylor series:

$$\boldsymbol{p}^*(\overline{\boldsymbol{q}}+\varepsilon) \approx \boldsymbol{p}^*(\overline{\boldsymbol{q}}) + \frac{\partial \boldsymbol{p}^*}{\partial \overline{\boldsymbol{q}}}\varepsilon + O(\varepsilon^2), \tag{3.47}$$

and

$$\boldsymbol{s}_{NN}(\overline{\boldsymbol{q}}+\varepsilon) \approx \boldsymbol{s}_{NN}(\overline{\boldsymbol{q}}) + \frac{\partial \boldsymbol{s}_{NN}}{\partial \overline{\boldsymbol{q}}}\varepsilon + O(\varepsilon^2). \tag{3.48}$$

It is also useful to see that the forcing term from the neural network has an error in comparison to the perfect forcing $\boldsymbol{s}$:

$$\boldsymbol{s}_{NN} = \boldsymbol{s} + \boldsymbol{e}, \tag{3.49}$$

where $\boldsymbol{e}$ is the error of the NN. However it is interesting to see that the forcing comes out from scaling a forcing that moves from zero to one:

$$\hat{\boldsymbol{s}}_{NN} = \hat{\boldsymbol{s}} + \hat{\boldsymbol{e}}, \tag{3.50}$$

and recovering the appropriate scale we have:

$$\boldsymbol{s}_{NN} = \boldsymbol{B}\hat{\boldsymbol{s}} + \boldsymbol{k} + \hat{\boldsymbol{B}}\hat{\boldsymbol{e}}, \tag{3.51}$$

which is

$$\boldsymbol{s}_{NN} = \hat{\boldsymbol{s}} + \hat{\boldsymbol{B}}\hat{\boldsymbol{e}}. \tag{3.52}$$

The error the NN generates is proportional to a non scaled error $\hat{\boldsymbol{e}}$ and the scaling factor $\hat{\boldsymbol{B}}$. The not scaled error $\boldsymbol{e}$ could be estimated with the training and testing process and should be low as long as the training error is low. This error has to be revised because the training data and data in the simulations are not equal, so one could expect an increase in this error when moving away from the trained part unless the flow behaviour is similar. On the other hand, another part to take into account is the proportionality with $\hat{\boldsymbol{B}}$. This basically means that the larger the forcing term is (the larger $\hat{\boldsymbol{B}}$ is) then the larger the error will be.

After all the approximations presented, we can now introduce eq. (3.47), eq. (3.48) and eq. (3.52) into eq. (3.45) and eq. (3.46) to get respectively:

$$\frac{\Delta \boldsymbol{\varepsilon}_T}{\Delta t} = \frac{\partial \boldsymbol{p}^*}{\partial \overline{\boldsymbol{q}}}\boldsymbol{\varepsilon}_T - \boldsymbol{s}(\overline{\boldsymbol{q}}), \tag{3.53}$$

and

$$\frac{\Delta \boldsymbol{\varepsilon}_{NN}}{\Delta t} = \frac{\partial \boldsymbol{p}^*}{\partial \overline{\boldsymbol{q}}}\boldsymbol{\varepsilon}_{NN} + \frac{\partial \boldsymbol{s}_{NN}}{\partial \overline{\boldsymbol{q}}}\boldsymbol{\varepsilon}_{NN} + \hat{\boldsymbol{B}}\hat{\boldsymbol{e}}. \tag{3.54}$$

Each vector such as ε_{NN} or ε_T represents a solution in a domain. Since the solution is nodal we see the rate of evolution in each node in the low order domain. It is interesting to take the norms from eq. (3.53) and eq. (3.54):

$$\frac{\Delta\left\|\varepsilon_T\right\|}{\Delta t} \leq \left\|\frac{\partial \boldsymbol{p}^*}{\partial \overline{\boldsymbol{q}}}\right\|\left\|\varepsilon_T\right\| + \left\|\boldsymbol{s}(\overline{\boldsymbol{q}})\right\|, \tag{3.55}$$

$$\frac{\Delta\left\|\varepsilon_{NN}\right\|}{\Delta t} \leq \left\|\frac{\partial \boldsymbol{p}^*}{\partial \overline{\boldsymbol{q}}}\right\|\left\|\varepsilon_{NN}\right\| + \left\|\frac{\partial \boldsymbol{s}_{NN}}{\partial \overline{\boldsymbol{q}}}\right\|\left\|\varepsilon_{NN}\right\| + \left\|\hat{\boldsymbol{B}}\right\|\|\hat{\boldsymbol{e}}\|. \tag{3.56}$$

In these equations there are two different types of terms in the equations. Those multiplied by the error itself, which are sensitivity types of error, and those that are not linearly proportional to the error. In the error from the truncated solution (eq. (3.55) without NN model) we see that the error is proportional to the Jacobian of the governing equations (NS) and there is a source of error that is the forcing term, which was missing in the formulation of the evolving equations. On the other hand, when we model the forcing term with a NN in eq. (3.56) we have the sensitivity from both the governing equations (NS) and also the sensitivity from the NN. In addition to that there is a source of error coming from the fact that the NN is not perfect.

We can make a very simplified model in order to check what kind of error curves we might find. Both the Jacobians from the NS equations and the NN will be time dependent in a sense that their maximum value will change depending on the solution and hence the moment in time. The first simplification is to consider some “effective" and constant sensitivity coefficient α which for the truncated case represents the sensitivity of the NS equations but for the NN case is the sensitivity for both the NS equations and the NN. Regarding the source term we can also introduce some “effective" source term β which for the NS equation is the missing source term but for the NN model is the error made by the NN:

$$\frac{\Delta\left\|\varepsilon_{NN}\right\|}{\Delta t} \approx \alpha_1\left\|\varepsilon_{NN}\right\| + \beta_1, \tag{3.57}$$

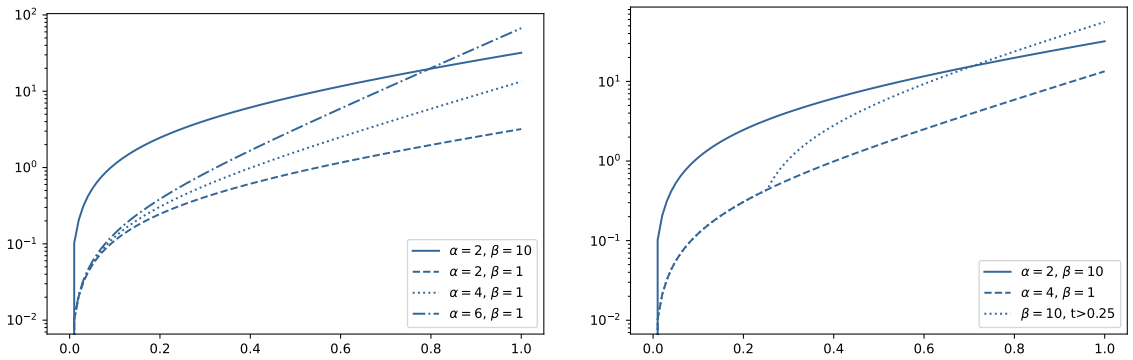
$$\frac{\Delta\left\|\varepsilon_T\right\|}{\Delta t} \approx \alpha_2\left\|\varepsilon_T\right\| + \beta_2. \tag{3.58}$$

If we consider that the time step is very small (limit as $\Delta t \to 0$) we can remove the time errors and consider the increment as a time derivative. That assumption lets us integrate both eq. (3.57) and eq. (3.58) and get analytical solutions:

$$\left\|\varepsilon_{NN}\right\| \approx \frac{\beta_1}{\alpha_1}\left(e^{\alpha_1 t} - 1\right), \tag{3.59}$$

and

$$\left\|\varepsilon_{NN}\right\| \approx \frac{\beta_2}{\alpha_2}\left(e^{\alpha_2 t} - 1\right). \tag{3.60}$$



(a) Different values for α

(b) Effect of changing β for $t > 0.25$

Figure 3.5: Theoretical evolution for the simplified error analysis

We can show with some values (made up) what we can expect from the low order solution with and without neural network. With the forcing term with the neural network we expect the system to be more sensitive to the data $\alpha_2 > \alpha_1$ but lower errors in the model $\beta_2 < \beta_1$ (see figure 3.5a). When the sensitivity is larger we see how the system ends up with a larger slope, but if the error in the model β is reduced then the starting point in the logarithmic graphic is in a much lower position.

We can expect that sometimes the NN might only be a good model within the period of training data. To see that effect we can take a simple model that works properly ($\alpha = 4$ and $\beta = 1$ from figure 3.5a) and change the value from $\beta = 1$ to $\beta = 10$ from a determined period of time $t > 0.25$, showing in the results a relevant increase in the error (see figure 3.5b). Although the error models are very simplified, these two cases (constant β or not) might give us an insight into what to expect from the experimental results and now that if for instance we find a curve such as in figure 3.5b, the model is not properly extrapolating.

Super-resolution NN

We can also add a NN to recover the high order modes from the low order solution. To have a very simple analysis, let us consider a re-constructor NN as a function $\boldsymbol{r} : \mathbb{R}^{LO} \rightarrow \mathbb{R}^{HO}$. We can do a very simple analysis by considering:

$$\varepsilon_S = \boldsymbol{q}^n - \boldsymbol{r}(\overline{\boldsymbol{q}}^n_{NN}) = \boldsymbol{q}^n - \boldsymbol{r}(\overline{\boldsymbol{q}}^n_{HO} - \varepsilon_{NN}) \approx \boldsymbol{q}^n - \boldsymbol{r}(\overline{\boldsymbol{q}}^n_{HO}) + \frac{\partial \boldsymbol{r}}{\partial \overline{\boldsymbol{q}}}\varepsilon_{NN} \tag{3.61}$$

where since ε_S is the error after applying the super-resolution scheme. On the other hand, reconstructing NN is not perfect, even if the entry to this NN is perfect there is

some error: $\boldsymbol{r}(\overline{\boldsymbol{q}}^n_{HO}) = \boldsymbol{q}^n_{HO} - \varepsilon_r$, leaving:

$$\varepsilon_S = \varepsilon_r + \frac{\partial \boldsymbol{r}}{\partial \overline{\boldsymbol{q}}}\varepsilon_{NN} \quad \rightarrow \quad \|\varepsilon_S\| \leq \|\varepsilon_r\| + \left\|\frac{\partial \boldsymbol{r}}{\partial \overline{\boldsymbol{q}}}\right\| \|\varepsilon_{NN}\| \tag{3.62}$$

3.2 Results: 1D Burgers equations

In this section we test the methodology on the 1D Burgers' equation, which reads as:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(-\frac{u^2}{2} + \nu\frac{\partial u}{\partial x}\right), \tag{3.63}$$

$$u(-1;t) = f_l(t), \quad u(1;t) = f_r(t), \quad u(x;0) = f_0(x), \tag{3.64}$$

where $t \in \mathbb{R}^+$, $x \in [-1,1]$, $u(x,t) : \mathbb{R}\times\mathbb{R} \rightarrow \mathbb{R}$ is the fluid variable, $f_l(t) : \mathbb{R} \rightarrow \mathbb{R}$ and $f_r(t) : \mathbb{R} \rightarrow \mathbb{R}$ represent the left and right boundary conditions respectively, $f_0(x) : \mathbb{R} \rightarrow \mathbb{R}$ is the initial condition and $\nu \in \mathbb{R}$ is the kinematic viscosity. The discretised variables will be written with u instead of q, $\boldsymbol{u} \longleftrightarrow \boldsymbol{q}$ and $\bar{\boldsymbol{u}} \longleftrightarrow \bar{\boldsymbol{q}}$.

Here, we have selected an unsteady boundary condition at the left end of the domain $f_l(t) = 1 + \sin(10t)/2$ and a steady boundary at the right end $f_r(t) = 1$. The initial condition is $f_0 = 1$ at $t = 0$ and the simulation time range is $t \in [0, 105]$.

The configurations for the test cases are summarised in table 3.1. We select three 1D cases with varying number of elements and polynomial orders: Case 01 uses a high order polynomial ($P_{HO} = 25$) and only one mesh element, Case 02 uses three elements and a lower polynomial ($P_{HO} = 7$), and Case 03 uses six elements and an even lower polynomial order ($P_{HO} = 5$). The filtered solution (low order) is $P_{LO} = 4$, $P_{LO} = 2$ and $P_{LO} = 0$ for Case 01, Case 02 and Case 03, respectively.

We observe that all cases show low ($< 1 \times 10^{-2}$) filtered errors $\bar{\varepsilon}_{max}$, illustrating that the methodology is accurate for different settings (see also figures below). Before comparing costs, let us recall the main steps in the methodology: first the simulation begins and generates data (high order simulation) for training, second a NN is trained to model the corrective forcing, third a low order solution evolves with the corrective forcing term being modelled by the NN. In table 3.1, C_{HO} represents the amount of time the third step would take using a High Order simulation, whereas C_{LO} is the time taken for the Low Order corrected simulation.

When we compare the cost of advancing in high order *Cost HO* and the of advancing in low order *Cost LO*, we see a substantial increase in efficiency, leading to ratios $CHO/CLO = 75$, $CHO/CLO = 22$ and $CHO/CLO = 59$ in Case 01, Case 02, Case 03, respectively.

	Case 01	Case 02	Case 03
ν	0.1	0.1	0.1
N_{El}	1	3	6
P_{HO}	28	7	5
P_{LO}	4	2	0
r	10	10	5
$\Delta t_n/\Delta t_m$	100	100	100
Epoch	500	500	500
Batch	36	36	36
N_{la}	20	20	25
N_{nf}	5	5	0
$\bar{\varepsilon}_{max}$	6.38 E-3	9.10 E-3	7.69 E-3
Cost HO	30.3	12.3	13.0
Cost LO	0.41	0.56	0.22
CHO/CLO	75	22	59

Table 3.1: Summary of cases with varying mesh elements and polynomial orders (fixed viscosity). We include the viscosity ν, the number of elements N_{El}, the Polynomial for the High Order solution P_{HO}, the polynomial for the Low order solution P_{LO}, the number of time steps r used as entry for the NN used to compute the corrective forcing. The relation between time steps $\Delta t_n/\Delta t_m$, the number of layers N_{la}, the increased number of neurons in the middle layer for flexibility N_{nf}, and the maximum value of the filtered error after 100 seconds of simulation $\bar{\varepsilon}_{max}$.

Let us note that the NN training cost in not included in the table and remains almost constant for all cases: $C_{training} = 24.5$, 27.6 and 26.9 seconds, for Case 01, Case 02 and Case 03, respectively. To generate the data for training and to obtain the corrective forcing, we only require $C_{HO-data-generation} = 0.3$ seconds of the high simulation and not a long high order simulation. The overall cost of the corrected low order simulation is computed as: $C_{NN} = C_{LO} + C_{HO-data-generation} + C_{training}$, providing 25.21, 28.46, 27.42 seconds for Case 01, Case 02 and Case 03, respectively. Note that C_{LO} includes the evaluation of the forcing with the trained network and advancing in time the corrected low order simulation.

The main cost is associated to the training $C_{training}$ in all cases. This cost could be largely reduced if the training is parellelised or if better performing programming languages are used, such as Fortran. For example [66] shows that Fortran can be 100-1000 times faster than Python.

In any case, and despite the training cost, for long enough low order simulations, the accelerations provided by our methodology mask the training cost, unveiling the potential of the proposed methodology.

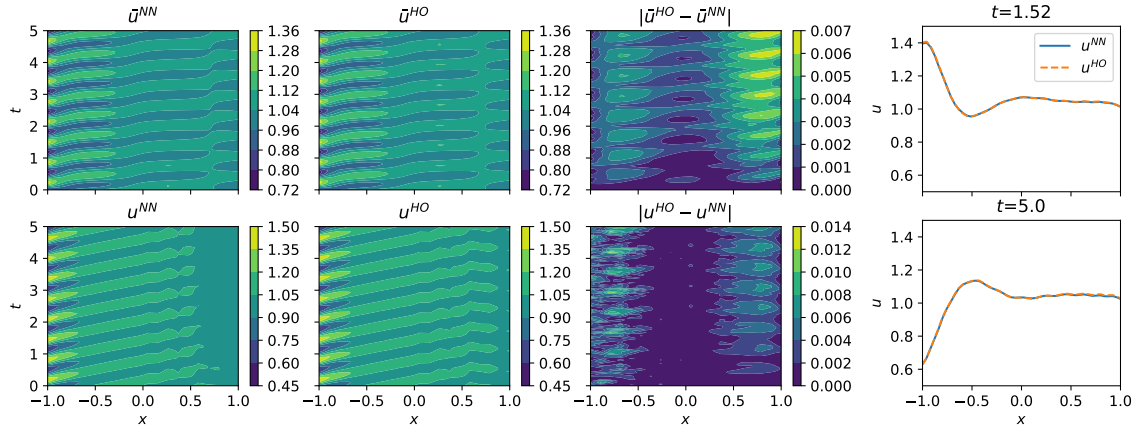
Figures 3.6a, 3.6b and 3.6c show the solutions and errors for Case 01, Case 02 and Case 03, respectively. The contour plots show the space-time $(x-t)$ evolution of filtered and unfiltered variables. The three figures on top show contour of the filtered solution of the NN $\bar{\mathbf{u}}^{NN}$, the filtered high order solution $\bar{\mathbf{u}}^{HO}$ and the associated error $||\bar{\mathbf{u}}^{HO} - \bar{\mathbf{u}}^{NN}||$. The three bottom contours are the un-filtered solutions from the NN $\mathbf{u}^{NN}$ (reconstructed with the post-processing NN from $\bar{\mathbf{u}}^{NN}$), then the High Order solution $\mathbf{u}^{HO}$ and the error $||\mathbf{u}^{HO} - \mathbf{u}^{NN}||$. We observe that the filtered high order solution is very similar to the solution interpolated using the NN. The maximum errors tend to be located, for all cases, near the left boundary condition, which is the unsteady boundary driving the solution, see equation (A.2). The accuracy of the reconstructed solution $\mathbf{u}^{NN}$ is better preserved when the low order polynomial is not zero, since case 03 exhibits similar errors in the filtered approximation (0.008) but higher when recovering the High Order data (0.04). The figures illustrate that the space-time behaviour is well captured when approximated using NNs and we conclude that the methodology performs well for a variety of meshes and polynomials.

For completeness, we also include line plots on the right of Figures 3.6a, 3.6b and 3.6c that compare the exact high order solution and the reconstructed $\mathbf{u}^{NN}$ at two given times. The two solutions are almost indistinguishable.

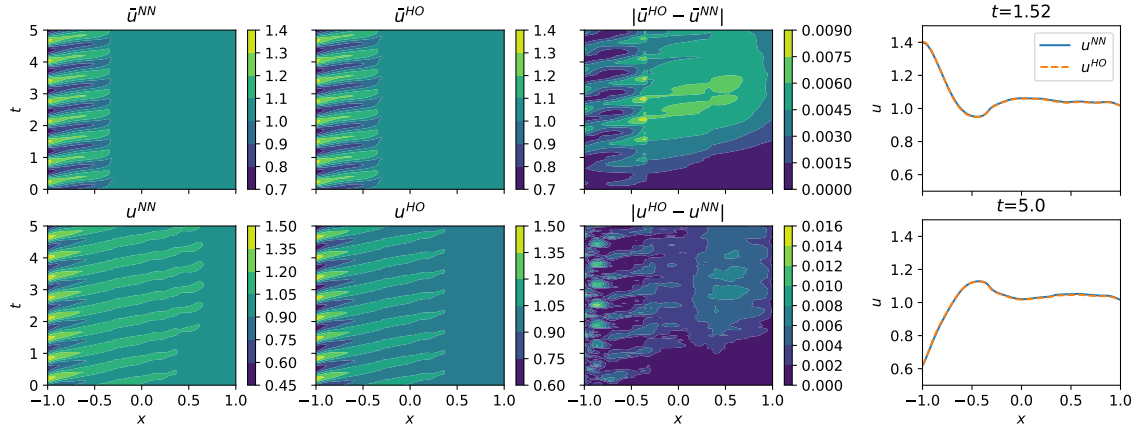
Note that for the reconstruction of the high order solution, a post-processing NN is used, which is similar to the main NN used to estimate the corrective forcing. The post processing NN had 15-25 layers and 800-1000 Epochs with 36 batches. This NN is purely for post-processing and does not influence the accuracy or cost of the time advancement. For this reason we have prioritised the study of main NN used to compute the corrective forcing term.

The table and figures show that Case 01 presents the most potential, since there is a big gain when going form a very high order polynomial $P_{HO} = 28$ to a low one $P_{LO} = 4$, leading to an acceleration ratio between High Order Cost and Low Order Cost of 75. This increase in computation efficiency relates directly to the increase in the allowed time step Δt (fewer iterations to reach the final time) and a lower number of degrees of freedom (lower time per iteration), when advancing the low order solution.

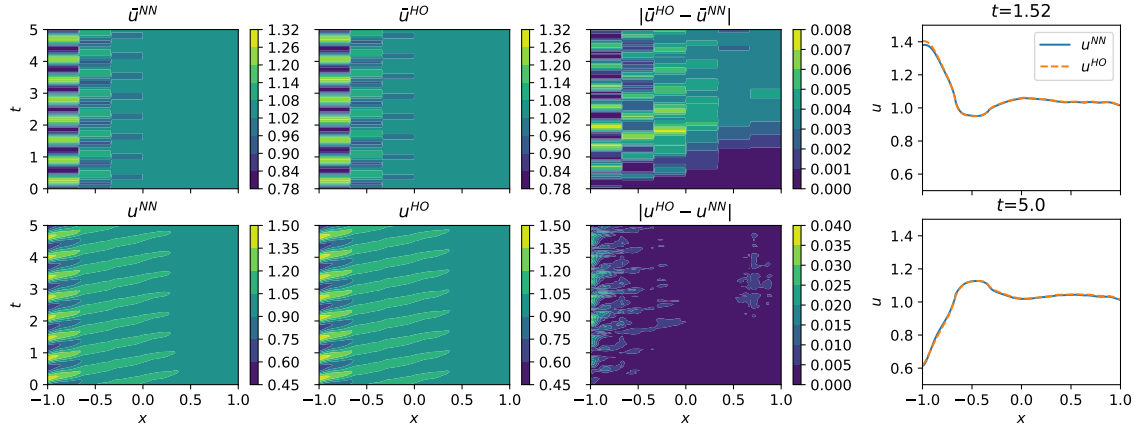
In addition, we check if the corrective forcing degrades the stability of the low order system (requiring a lower CFL number). Using polynomial order 4, we calculate: $u/\Delta x_{min} \approx 16$ and compare it to the maximum value of s, which is 1.26. This suggests that the CFL stability is dominated by the discrete numerical solution and that the corrective forcing does not significantly modify the numerical stability.



(a) Results for Case 01.

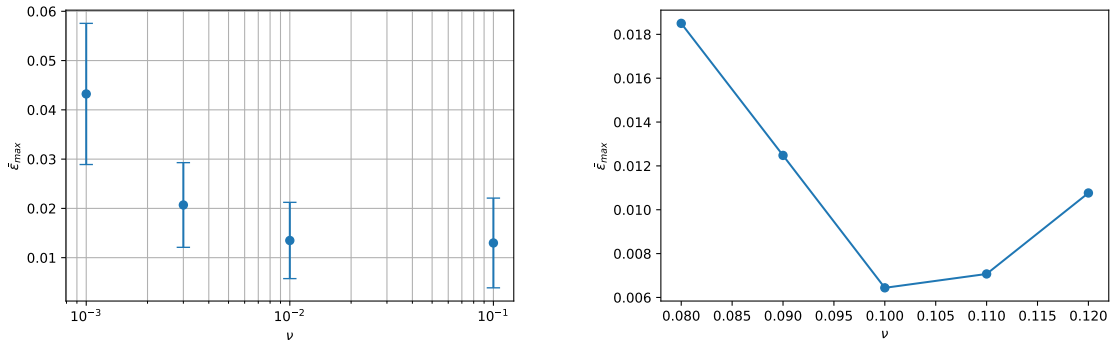


(b) Results for Case 02.



(c) Results for Case 03.

Figure 3.6: Results for the three presented cases.



(a) Effect of how the same NN adapts to lower viscosity

(b) Effect of reusing a NN (trained for $\nu = 0.1$) can be used for different viscosities

Figure 3.7: Variations in the kinematic viscosity ν.

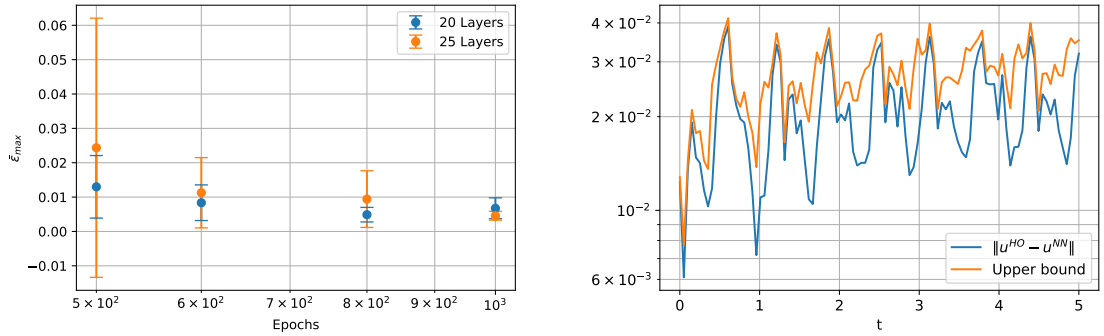
3.2.1 Numerical Parameters, Hyper-parameters and Error Bounds

We include additional insights into the parameters influencing the accuracy of these cases. Namely, we study the effect of the viscosity and check if a corrective forcing obtained form a NN trained at one viscosity can be used at other viscosity values (transfer training). Additionally, we asses the importance of the number of Epochs and the number of layers in the NN. We finally compare the error bounds derived in 3.1.4 with exact values to illustrate their sharpness.

We select Case 01 for these test, as it has shown to be the most promising, and is hereafter considered the baseline configuration. Since the *Adam* optimiser is stochastic, it can find a different result for similar simulations. For this reason, when changing the parameters, we run each simulation ten times, to derive a mean value and the standard deviation.

Figure 3.7a shows the maximum filtered error $\bar{\varepsilon}_{max}$ for a range of viscosities ν. We observe that it is more difficult to simulate low viscosities. This behaviour is expected since low viscosities generate a larger amount of modes (due to non-linear interactions) and are generally more difficult to model.

Figure 3.7b shows the maximum filtered error $\bar{\varepsilon}_{max}$ for a range of viscosities ν. In this case, we check if a NN trained for one viscosity $\nu = 0.1$ can be used to approximate the other viscosities (i.e. transfer learning). We observe that if the viscosity values are not too far apart and the NN is well trained, the NN can be reused providing accurate solutions. We observe accurate results for high viscosities (i.e. smooth flows) and larger errors for low viscosity values, which relate to the reduction of flow smoothness, leading to more complex flow physics (e.g. sharp shocks when solving Burgers' equations).



(a) Representation of a mean value and its standard deviation of $\bar{\varepsilon}_{max}$.

(b) Results of comparing the error with equation (3.62)

Figure 3.8: Numerical experiments with the number of Epochs and layers, as well as a visualisation of the upper bound.

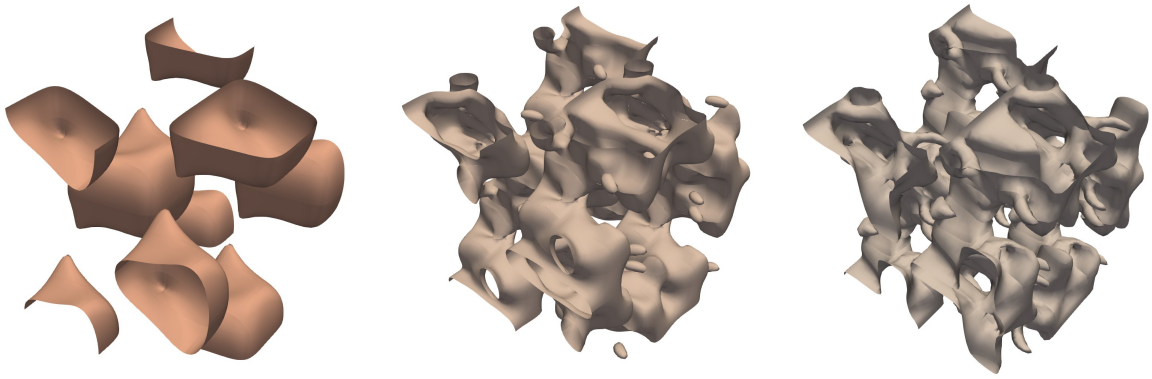
Figure 3.8a represents the mean and variance of the filtered error $\bar{\varepsilon}_{max}$ after 10 tests with different values of number of Epochs for 20 layers and for 25 layers. We observe that if the NN is not sufficiently trained (not enough Epochs), its error and standard deviation are large. To reduce the error and the standard deviation, one can consider increasing the Epochs (fixing the layers) or increasing the number of layers (fixing the Epochs). Indeed, 3.8a shows that for a fixed and low number of Epochs, the errors can be compensated by increasing the number of layers (to a certain extend), whilst for large number of Epochs the number of layers has minimum effect, see figure (3.8a). Note that this tuning has to be done carefully to not overfit the training data, which has been checked by supervising the error (from test data).

Finally, we also check the upper bounds in (3.62), using data from simulations to validate the analytical bounds, see figure 3.8b. The bound seems to predict well the time evolution of the error, and can be used in the future to asses the accuracy of the methodology in more complex 3D cases.

3.3 Results: 3D Navier-Stokes equations

3.3.1 Cases configuration

In order to visualize the potential of this methodology we have tested it with the Navier-Stokes equations in a Taylor-Green Vortex [67] case with different configurations e.g. different Reynolds number, hyper-parameters and different stages of the simulation (see table 3.2). We have organized all the cases into three main groups depending according



(a) Re = 30. (b) Re = 200. (c) Re = 1600 LES.

Figure 3.9: Iso surface for $u = 2\,m/s$. Mesh 8^3

to the Reynolds number:

- Case 1: Reynolds number 30
- Case 2: Reynolds number 200
- Case 3: Reynolds number 1600

We have selected different Reynolds numbers because we expect that for very low Reynolds number the solution has spectral convergence [31, 32, 68, 69] and hence we expect the smoother flows to be easier to describe with a lower polynomial order (see figure 3.9), thus expect the forcing term to not be so relevant.

Another important thing to take into account is the fact that during the start, the flow is very smooth since the initial condition is given by C_∞ functions. The flow has not generated all the relevant modes and the flow is far from homogeneous. That made us suspect that during the start, the physics of the flow might change a lot in a sense that the flow is generating very different structures to the ones at the beginning. Hence we can expect the NN to not perform well outside the training region and error curves similar to the ones in figure 3.5b. When the flow is developed ($t > 6\,s$) the solution is much more homogeneous and many more modes are already generated. That explanation might be a good reason to think that the training data information can be extrapolated and be used outside the training region, at least for a short period of time. This is the reason why we have tested the methodology for each Reynolds number in both the starting condition and a in a more advanced part of the simulation.

Regarding the simulation we have used horses3d [65], a nodal Discontinuous Galerkin Spectral Element Method (DGSEM), written in modern object–oriented Fortran. For

evaluating the deep neural networks with at the same time of the solver, we have used a library called Fortran to Keras Bridge (FKB) [58], although the neural networks were trained within Python. Horses3d is compiled with -O3 with gfortran and parallelized with OpenMPI, python executions (training) were run in serial. The simulations were run in a desktop computer with Intel(R) Core(TM) i7 CPU 3.07 GHz and RAM 14,0 GB.

As for the mesh, it is a very coarse mesh with just 8 elements in each direction, with a total number of elements of $8^3 = 512$ (see figure 3.10). Inside each element the formulation is nodal with Lagrange polynomials in Gauss-Legendre nodes and Roe [70] and Bassi Rebay 1 [71] methods for the advection and diffusion fluxes.

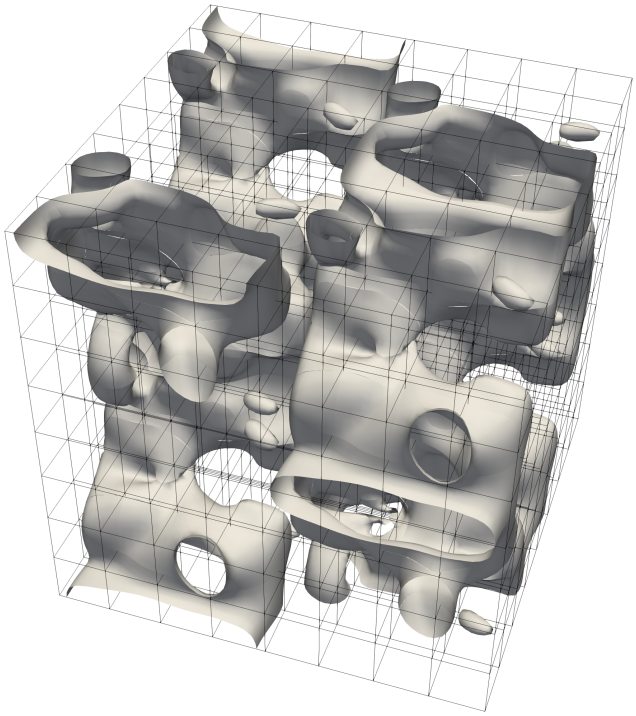
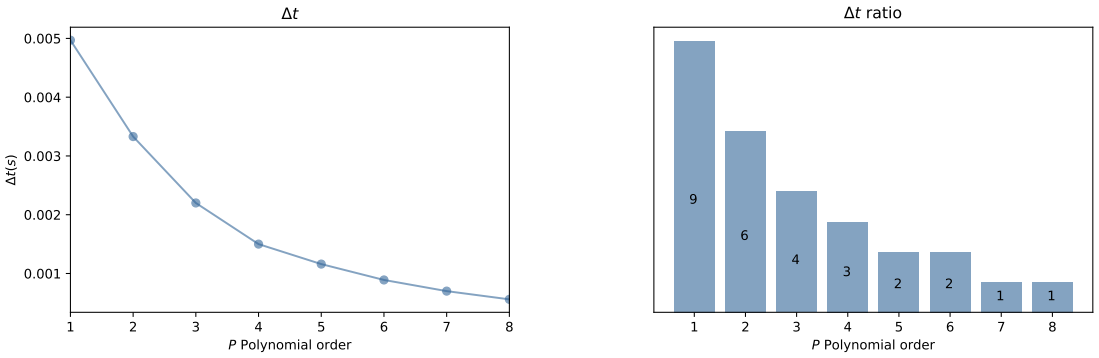


Figure 3.10: Coarse mesh with 8 elements in each direction (8^3) and $P = 8$. Inside the mesh we present a iso-surface for $v = 2\,m/s$ with $Re = 200$.

For the high order solution we have selected a high polynomial order $P_{HO} = 8$. In the case of the low polynomial order we have tested both $P_{LO} = 2$ and $P_{LO} = 3$. The reason for this is that having lower or higher P_{LO} has some advantages and disadvantages. Some of them are:

- Using smaller low order polynomial
  - Lower memory usage



(a) Δt for a 8^3 mesh with CFL=0.4. (b) Ratio $\Delta t/\Delta t_8$ for a 8^3 mesh with CFL=0.4

Figure 3.11: Temporal time step difference depending on the time step.

  - For a constant number of layers, the number of weights and biases is lower and hence the optimization is faster.

- Using larger low order polynomial

  - For a constant number of layers, the number of weights and biases is higher and hence a more complex model can be approximated.

In order to select the time step for the high and low order schemes we have run a few steps for each polynomial with a constant Courant–Friedrichs–Lev number (CFL). By knowing the Δt for each polynomial (see figure 3.11a) we can calculate the ratio available between the time step in low order and high order. This ratio can be truncated to an integer number (see figure 3.11b) and hence know how many iterations we must run the high order solution for a single low order iteration. Please note that this ratio should be checked in each stage of the simulation, since the CFL condition depends on the flow condition.

Another important thing to take into account is that the flow is almost incompressible since $M = 0.08$, thus we expect the density and energy to have very small spatial variations. For this reason the density and energy do not need a high order resolution neither a correction for the low order solution. With these ideas in mind, the chosen NN only takes as an input information from the velocity (ρu, ρv and ρw) and only corrects the momentum equations:

$$\Delta \begin{Bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{Bmatrix} = \Delta t \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{Bmatrix} + \Delta t \begin{Bmatrix} 0 \\ s_2(\rho u, \rho v, \rho w) \\ s_3(\rho u, \rho v, \rho w) \\ s_4(\rho u, \rho v, \rho w) \\ 0 \end{Bmatrix}. \tag{3.65}$$

Case Number	Case 1		Case 2			Case 3		
Reynolds Number	30		200			1600		
sub case	a	b	a	b	c	a	b	c
HO scheme equations	NS	NS	NS	NS	NS	NS	LES	LES
LO scheme equations	NS + NN	NS + NN	NS + NN	NS + NN	NS + NN	NS + NN	NS + NN	LES + NN
Starting time	0	7	0	7	7	0	7	7
Polynomial order, P_{HO}	8	8	8	8	8	8	8	8
Polynomial order, P_{LO}	2	2	2	2	3	2	3	3
time step Δt_{HO}	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$
time step Δt_{LO}	$3 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$
Number of Layers	4	4	4	4	4	4	5	5
Number of epochs	30	30	30	30	30	30	35	50
Number of batches	20	20	20	20	20	20	20	20
train time interval (s)	[0, 0.3]	[7, 7.3]	[0, 0.3]	[7, 7.3]	[7, 7.15]	[0, 0.3]	[7, 7.15]	[7, 7.15]
HO time 400 iter (s)	1176.62	1443.83	1351.88	1273.51	601.08	1393.66	725.02	741.94
LO time 400 iter (s)	18.04	29.57	35.34	23.58	50.84	26.45	43.65	59.49
Ratio HO/LO	65	49	38	54	11.8	53	16.6	12.5
LO no nn (s)	14.5	20.89	21.83	20.63	31.35	17.91	26.94	34.29
NN train time (s)	163.75	181.39	171.36	175.60	317.36	178.67	435.64	829.10
Loss Error	$8.98 \cdot 10^{-5}$	$1.07 \cdot 10^{-4}$	$8.9033 \cdot 10^{-5}$	$9.72 \cdot 10^{-5}$	$7.31 \cdot 10^{-5}$	$8.95 \cdot 10^{-5}$	$1.09 \cdot 10^{-4}$	$1.60 \cdot 10^{-4}$

Table 3.2: Summary of the test cases. Case 1, 2 and 3 group test cases with Reynolds numbers of 30, 200 and 1600 respectively. In the three cases, subcases a) have been used for a test starting at the beginning of the simulation, whereas subcases b) and c) have been used for an already started simulation (Starting time). In each subcase we have used some equations for the high order scheme (HO scheme equations) and different ones for the low order scheme (LO scheme equations) where NS stands for Navier-Stokes, LES for Large Eddy simulation (NS + Smagorinsky) and NN is the neural network forcing. The polynomial for the high and low order solutions are given by the rows Polynomial order, P_{HO} and Polynomial order, P_{LO} respectively. We also present the time steps for both the high order and low order schemes Δt_{HO} and Δt_{LO}. The number of layers, epochs and batches are also given. The data used for training is in the interval given by the row train time interval. From the results we see how long it take the high order scheme to perform 400 low order time steps in column HO time 400 iter (S), and how long it takes the low order scheme with and without neural network to do 400 low order time steps in rows LO time 400 iter (S) and LO no nn (s) respectively. The ratio between the HO time cost and the LO time cost is presented in the row Ratio HO/LO. We finally include the time to train the nn in row NN train time(s) and the loss error achieved which is a mean square error.

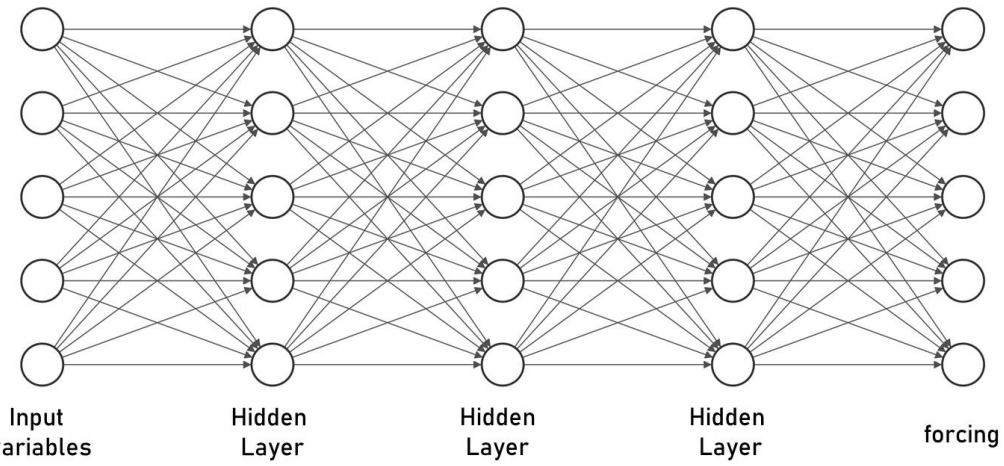


Figure 3.12: Deep neural network scheme. The input is a vector containing the nodal values for ρu, ρv and ρw inside the element, and the output is a vector containing the nodal values for the forcing (s_2, s_3 and s_4 from eq. (3.65)) inside the element.

Before commenting the results, all the variables at the y axis are the the infinite norm from the difference between the low order solution and the high order solution filtered:

$$\boldsymbol{e}_n = \left\| \overline{\boldsymbol{q}}^n_{LO} - \overline{\boldsymbol{q}}^n_{HO} \right\|_\infty. \tag{3.66}$$

This means that we take the high order solution evolution, we filter it to the low polynomial order and then compare its evolution with the low order model with and without NN. For each result there are three graphics, a) which is the error regarding the variable ρu, b) is the error in ρv and c) which is the error in ρw. We do not present errors in ρ or ρe because the flow is almost incompressible, hence the density is almost constant and the energy is decoupled.

3.3.2 Results

For each Reynolds number (case 1, case 2 and case 3) there are tests of the methodology both at the start and at the middle of the simulation. In the three cases at the start, when we only train with a short period of time the NN is only capable of keeping a lower error with the neural network during two times the training time approximately (See results on figures 3.13, 3.14 and 3.15). This means that the model from the neural network trained with $t < t_{train}$ is not extrapolable to further periods of time $t > t_{train}$ following a similar result as seen in figure 3.5b. This seems to confirm our hypothesis in which during the start the physics is changing (generating a lot of modes) and thus training data is critical for the neural network to behave properly.

In the middle of the simulation ($t = 7\,s$) for a low Reynolds number (case 1 with

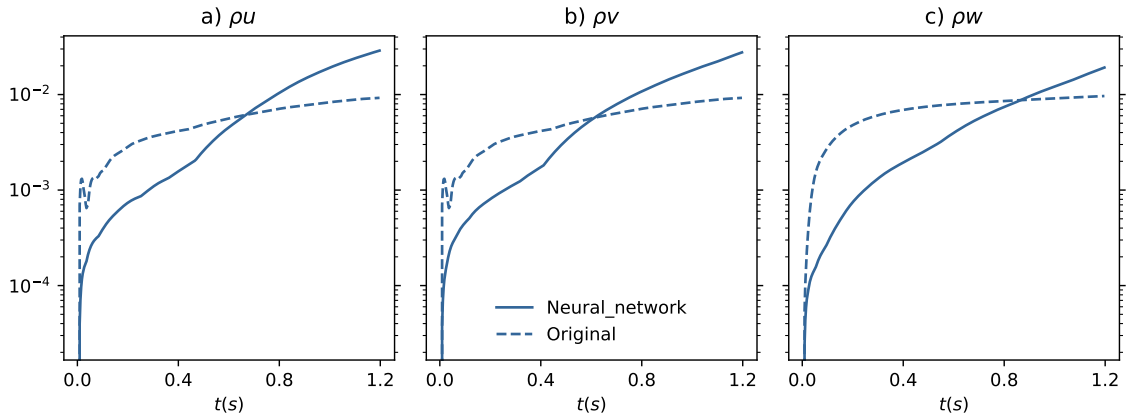


Figure 3.13: Infinite error in ρu, ρv and ρw case 1 a, $Re = 30$ and transformation from $P_{HO} = 8$ to $P_{LO} = 2$ with a starting time $t = 0$

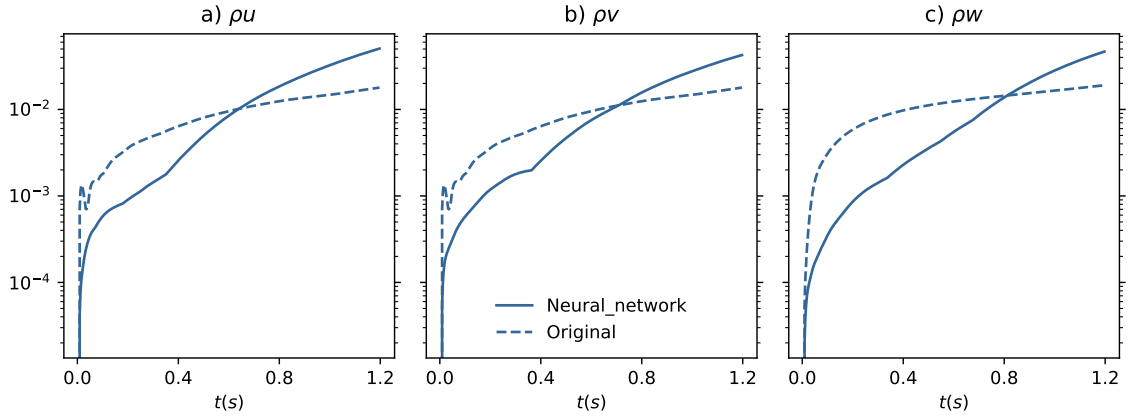


Figure 3.14: Infinite error in ρu, ρv and ρw case 2 a, $Re = 200$ and transformation from $P_{HO} = 8$ to $P_{LO} = 2$ with a starting time $t = 0$.

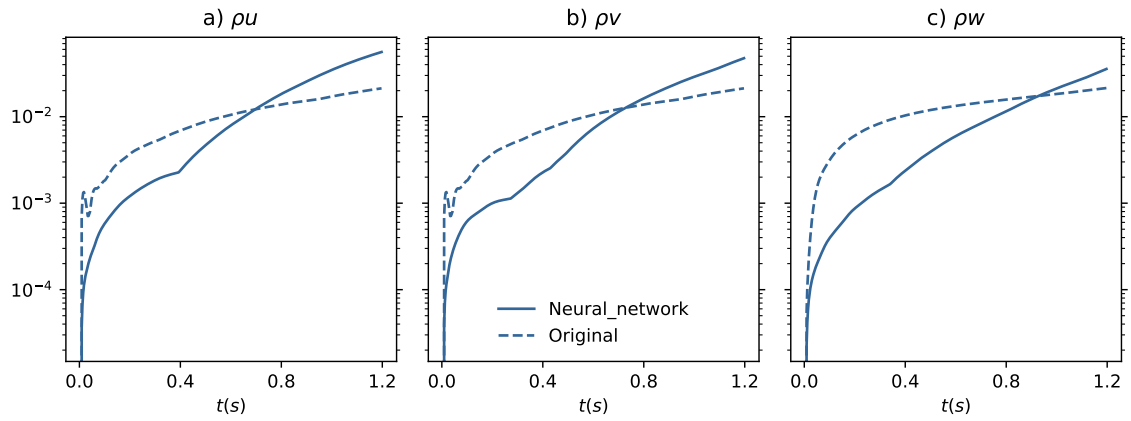


Figure 3.15: Infinite error in ρu, ρv and ρw case 3 a, $Re = 1600$ and transformation from $P_{HO} = 8$ to $P_{LO} = 2$ with a starting time $t = 0$.

Re = 30) the NN can extrapolate the model trained with data from $t = 7\,s$ to $t = 7.3\,s$ and extend the its use for longer still achieving less error than without the NN. We have also measured the time for 400 iterations with the high order scheme and the low order scheme and seen that the cost of high order was 65 times larger than for the neural network case.

However when increasing the Reynolds number (200), going from $P_{HO} = 8$ to $P_{LO} = 2$ there is a similar problematic as in the cases tested at the beginning (see figure 3.17). We however expect the problem to be related to a not enough accurate NN and not due to a change in the physics behaviour. In order to solve this problem we have increased the low order polynomial to $P_{LO} = 3$. This small increase in the low order polynomial improves the solution and with training data for 100 iterations since we are capable of having less error for 400 iterations (see figure 3.18)

When increasing the Reynolds number to 1600 we find in a high order solution the mesh to be to coarse to do a DNS simulation without having stability issues. For this case we have used the high order polynomial P8 with a Smagorinsky LES scheme. Regarding the NN model, we have tested two cases: just DNS + NN for the low order evolution (Case 3b) and DNS + LES + NN at for the low order solution (Case 3c). We can see that the method case 3b where the neural network acts as a sub-grid model works better than computing the LES term and correcting it (see figure 3.19).

3.3.3 Parameters analysis

After applying this methodology in a middle stage of the simulation we are interested in tuning parameters from the neural network and check how to improve the solution. For

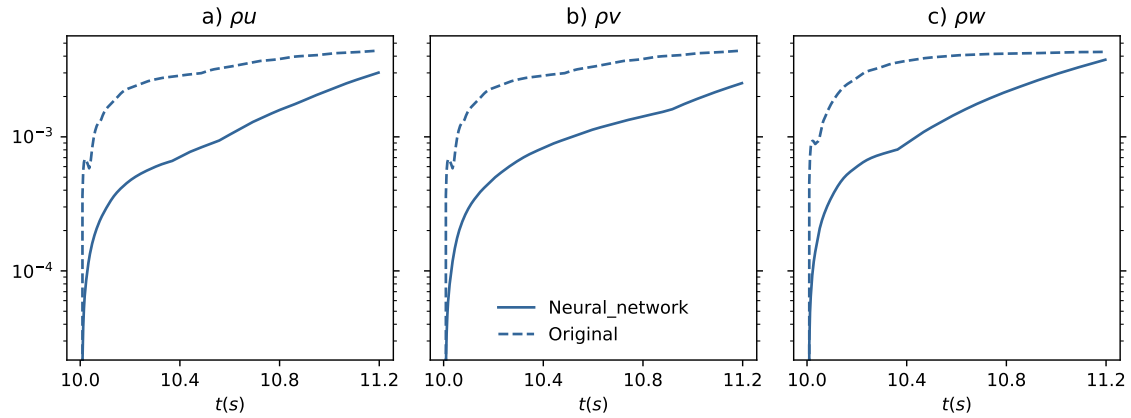


Figure 3.16: Infinite error in ρu, ρv and ρw case 1 b, $Re = 30$ and transformation from $P_{HO} = 8$ to $P_{LO} = 2$ with a starting time $t = 7$.

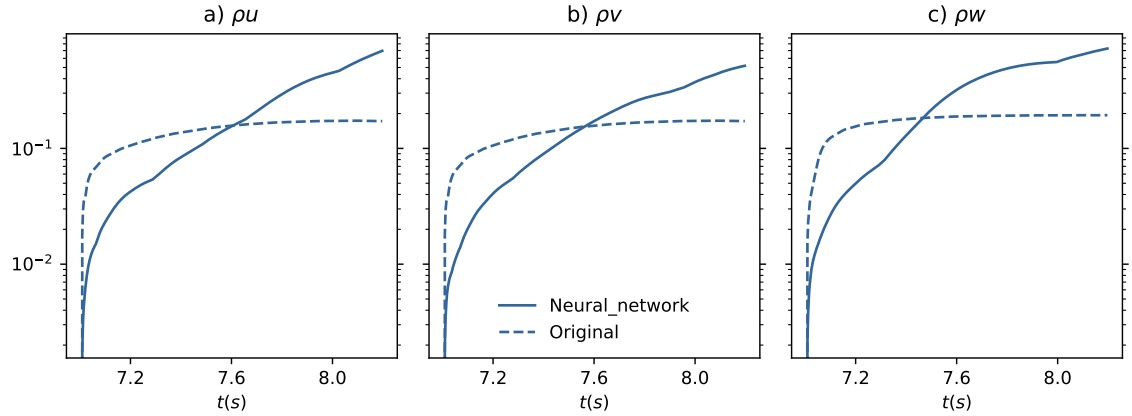


Figure 3.17: Infinite error in ρu, ρv and ρw case 2 b, $Re = 200$ and transformation from $P_{HO} = 8$ to $P_{LO} = 2$ with a starting time $t = 7$.

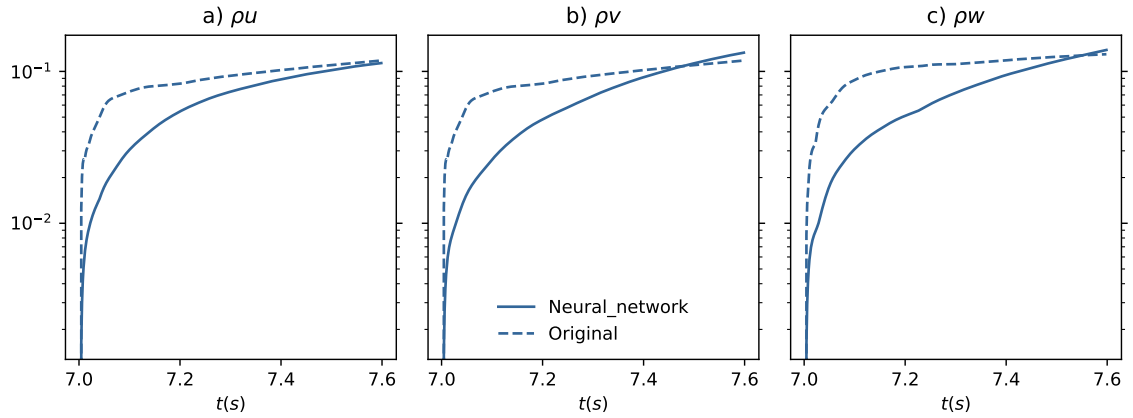


Figure 3.18: Infinite error in ρu, ρv and ρw case 2 c, $Re = 200$ and transformation from $P_{HO} = 8$ to $P_{LO} = 3$ with a starting time $t = 7$.

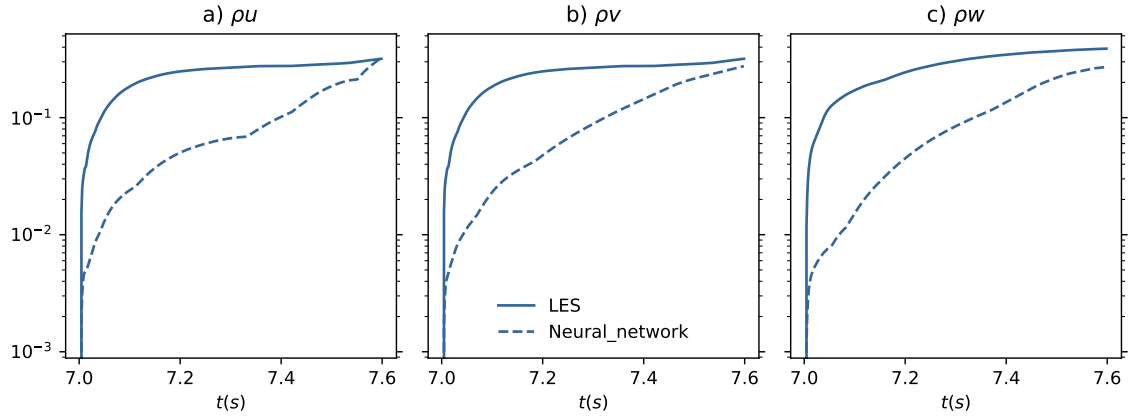


Figure 3.19: Infinite error in ρu, ρv and ρw case 3 b, $Re = 1600$ and transformation from $P_{HO} = 8$ to $P_{LO} = 3$ with a starting time $t = 7$. Only the high order solution has a LES term.

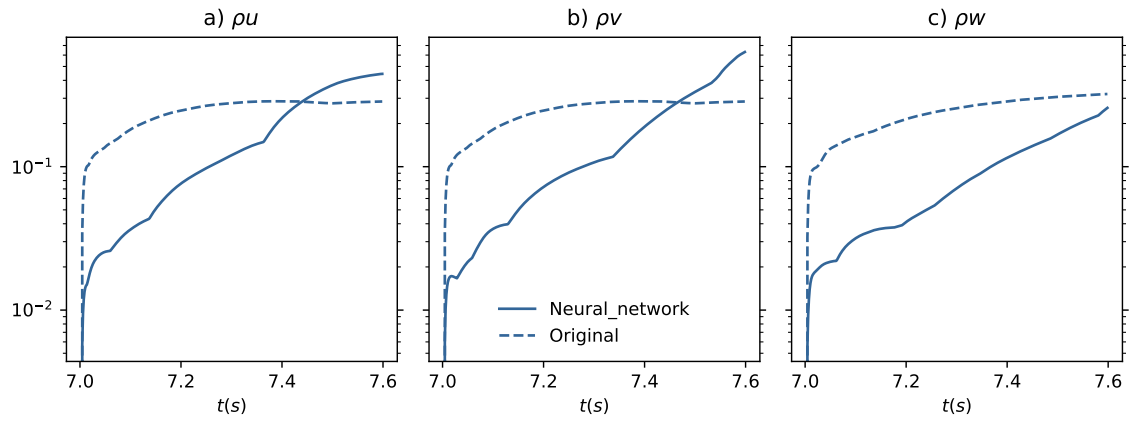


Figure 3.20: Infinite error in ρu, ρv and ρw case 3 c, $Re = 1600$ and transformation from $P_{HO} = 8$ to $P_{LO} = 3$ with a starting time $t = 7$. Both the high order and low order solutions have a LES term.

that we will be focusing on Case 1b (starting in $t = 10$) and we will try to improve the solution. For that we will use case 1b an check how the increase in layers and epochs:

- Original: without neural network
- Neural network 1: case 1b
- Neural network 2: case 1b + 30 epochs → 40
- Neural network 3: case 1b + 30 epochs → 40 layers + 4 layers → 5 layers

By increasing both the number of layers and number of epochs the model can slightly reduce the error (see figure 3.21).

It is also interesting to test the original neural network from the data at the start and use them at the middle of the simulation (starting_nn). In this case, with the NN from case 1a and with only 10 extra epochs (Retrained_nn) we are capable of reaching similar solutions spending 69.995 seconds for training instead of 181.39 seconds (see figure 3.22)

Another possibility to explore is to check if taking a NN trained for case 3b (Reynolds number 1600) was useful for case 2c (Reynolds number 200). By testing the same model we can see that the error is much larger than both the equations without neural network and the case 2c neural network. However if we use its weights and bias as an initial condition to optimize the neural network with just 20 iterations with 227.00 seconds of training time instead of 440 seconds we achieve similar errors in comparison to the modeled trained from scratch (see figure 3.23).

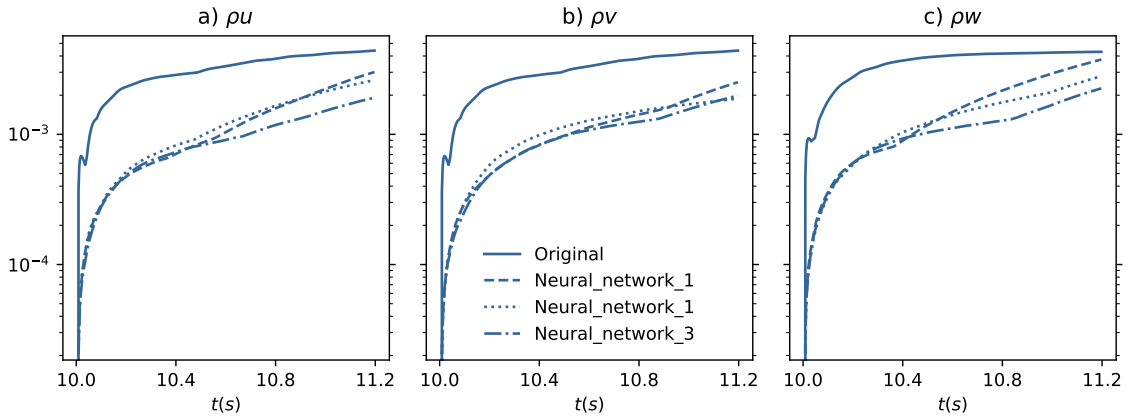


Figure 3.21: Infinite error in ρu, ρv and ρw case 1b with modifications, $Re = 30$ and transformation from $P_{HO} = 8$ to $P_{LO} = 2$ with a starting time $t = 10$

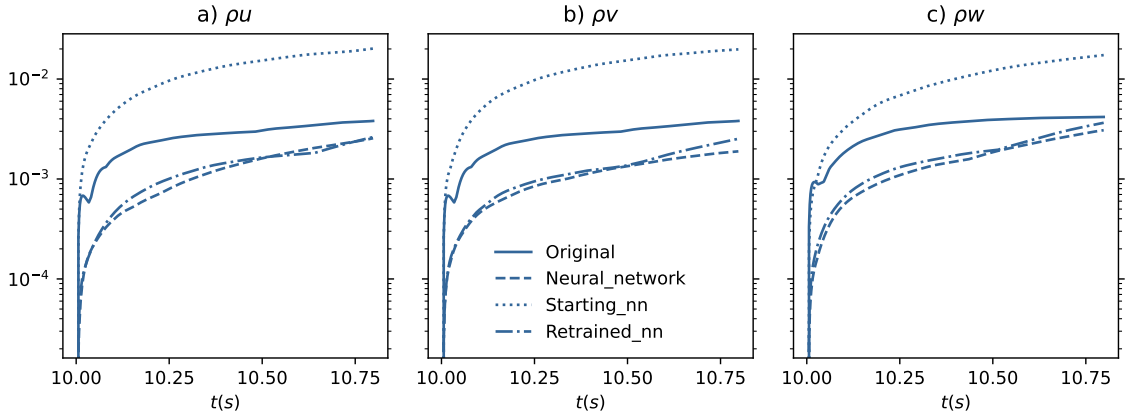


Figure 3.22: Infinite error in ρu, ρv and ρw case 1b with another nn, $Re = 30$ and transformation from $P_{HO} = 8$ to $P_{LO} = 2$ with a starting time $t = 10$

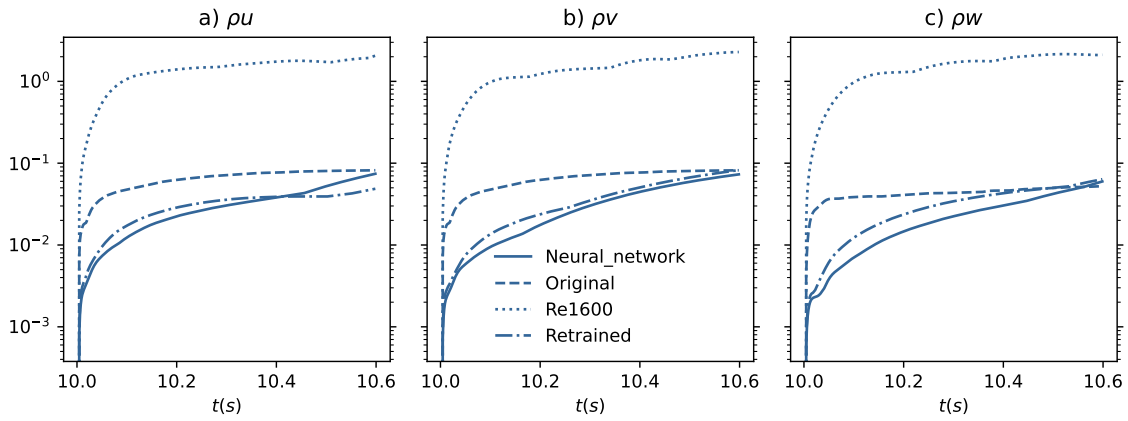


Figure 3.23: Infinite error in ρu, ρv and ρw case 2c with another nn.

3.3.4 Effective acceleration

On the presented cases, the error achieved by the NN has been in multiple cases lower than the error given by the low order solution without NN. But we can ask our selves, how much accuracy have we gained? because another option would be to increase the polynomial order to an intermediate order $P_{LO} < P_{IO} < P_{HO}$ and check when the error from the NN model is similar and see if the NN is faster. This is actually the key question, because it answers the question whether this model pays off and shows an effective acceleration.

For that purpose, we will use as a base the case 3b in which we have a 1600 Reynolds number, with a high order polynomial order of $P_{HO} = 8$ and a reduction to a low order polynomial of $P_{LO} = 3$. We have checked several intermediate polynomial orders (see table 3.3):

Case	Order	Δt	Cost (s)/ 200 iter	Ratio
P_{LO}	3	$20 \cdot 10^{-4}$	13.47	0.50
P_{LO} +NN	3	$20 \cdot 10^{-4}$	26.88	1
P_{IO}	4	$10 \cdot 10^{-4}$	36.16	1.35
	5	$6.7 \cdot 10^{-4}$	86.94	3.23
	6	$6.7 \cdot 10^{-4}$	132.79	4.94
P_{HO}	8	$5.0 \cdot 10^{-4}$	532.07	19.79

Table 3.3: Tests to evaluate the effective acceleration.

From the results, the error with the NN correction (see figure 3.24) is similar to

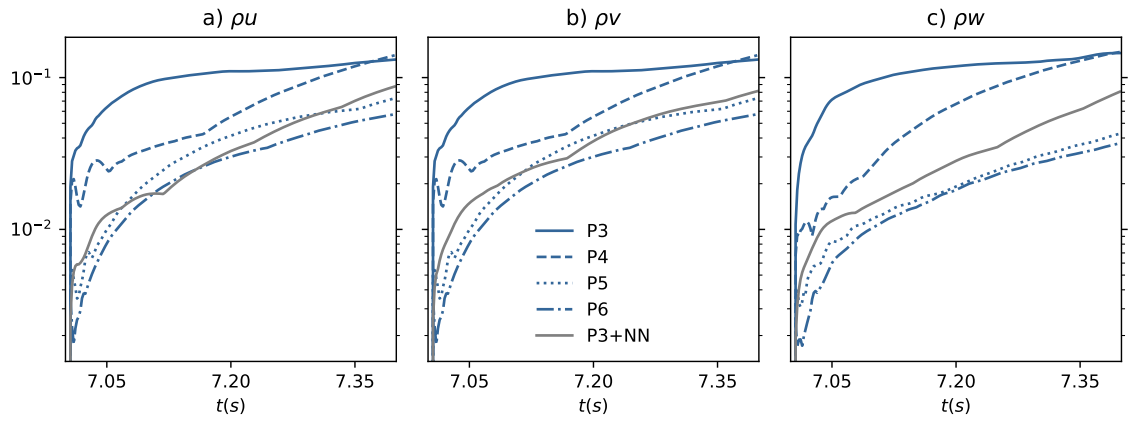


Figure 3.24: Infinite error in ρu, ρv and ρw compared to the error from other polynomial methods, with Reynolds number of 1600. For the neural network we have used 8 layers with 30 epochs.

$P_{IO} = 5$ and $P_{IO} = 6$ in ρu and ρv, but the cost is 3.23 and 4.94 times lower respectively. The NN model gets $P \sim 5/6$ order accuracy (in the low order modes) by running just a $P = 3$ polynomial with a correction, which turns out to be around three to five times faster. This is the real acceleration this method gets with the Navier-Stokes equations with this local model.

3.3.5 Checking other stages of the simulation

As an addition to the effective acceleration we wanted to check that the methodology works in more stages of the situation apart from than in $t_0 = 7$. We have applied the method in $t_0 = 4$ (see top figure 3.25), in $t_0 = 10$ (see middle figure 3.25) and in $t_0 = 13$ (see bottom figure 3.25) which together with the the test in $t_0 = 0$ (see figure 3.15) and in $t_0 = 7$ (see figure 3.19) give an insight into what to expect from it. These three situations have chosen because they are representative conditions in the TGV test case (see figure A.1).

Final comments

We have presented the formulation of a new methodology that aims to use NN to model the effect of higher-order modes in a DGSEM scheme in order to be able to leave them and speed up the computations. We have tested the methodology with Burgers and Navier-Stokes showing its strengths and weaknesses. In the following chapter we will address all the conclusions from this project as well as some future work.

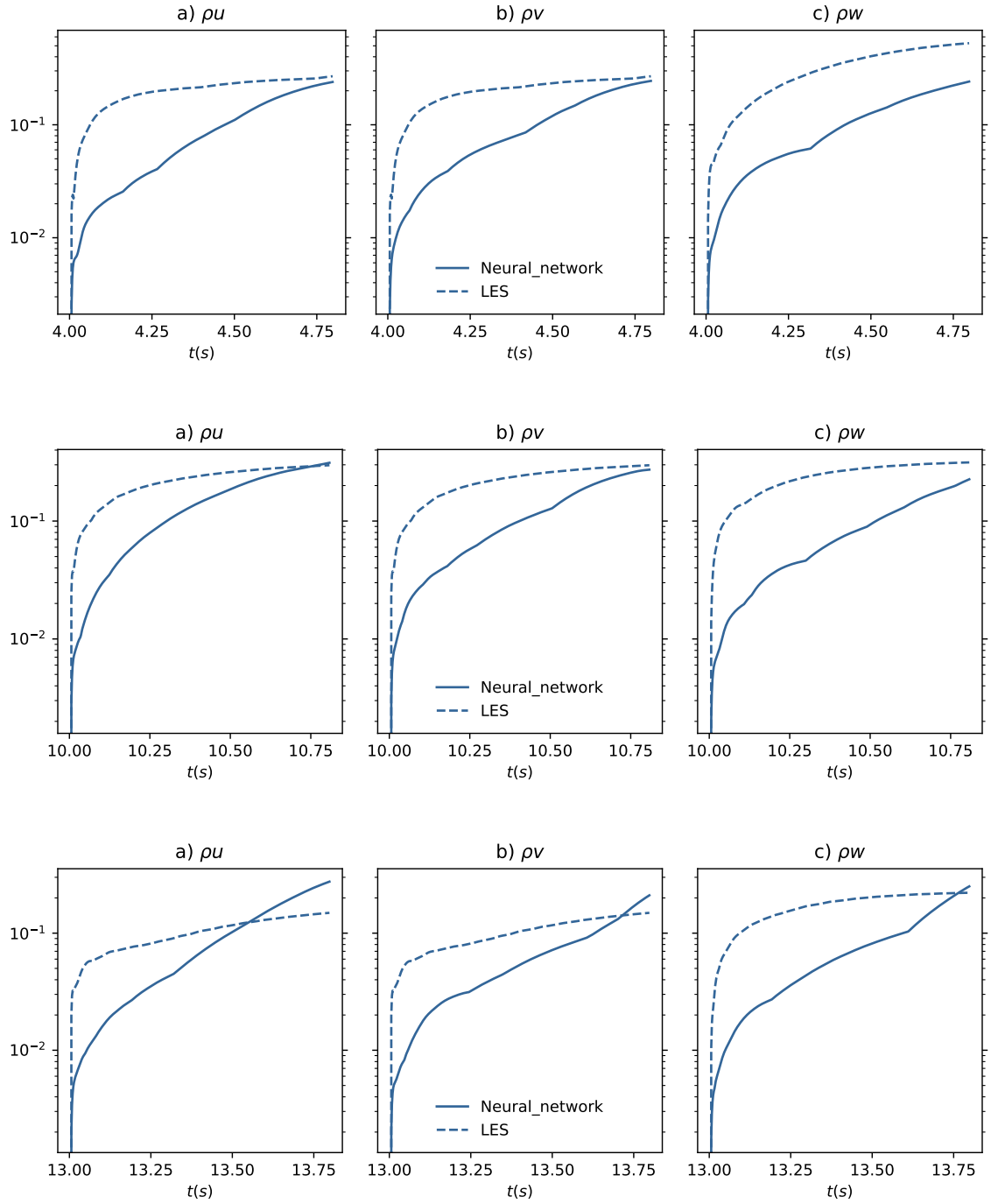


Figure 3.25: Different starting times, 4 (top), 10 (middle) and 13 (bottom). Three cases with $Re = 1600$ and transformation from $P_{HO} = 8$ to $P_{LO} = 3$

CHAPTER 4

Conclusions and future work

4.1 Conclusions

In this work we propose a new methodology using NNs to accelerate high order DG methods by evolving a low order solution and modelling the effect of the high order modes with a NN. We introduce the method with the corresponding mathematical formulation, some analysis on the error of the method and test it on (I) the 1D Burgers' equation and (II) the 3D compressible Navier-Stokes equation.

As for the implementation details, for the 1D Burgers' equation we used a test code developed by David Kopriva and Gonzalo Rubio that resolved the DG formulation. As for the Navier-Stokes equations we embed the method as a new time integrator for horses3d [65] where we evaluated the NN with the library Fortran to Keras Bridge (FKB) which has been previously trained using Keras.

Regarding the Burgers' test case we tested the problem with an unsteady boundary conditions. We checked different meshes (different number of elements and polynomial orders) and parameters, the results showed how the method is capable of modelling the source term and even recover an approximation of the high order solution.

On the Navier-Stokes equations we found more difficulties to sort out before applying the methodology. We found out that the scaling the flow variables and forcing terms reduces the error in the model significantly. Also, we kept the density and energy

variables out of the NN model. It prevents the model to be very sensitive to numerical perturbations in the density since the flow is almost incompressible. Another important thing we have noticed is that using a NN for the whole domain is very expensive. We also found that using a NN for each element reduces the error but increases the sensitivity. After considering the alternatives we had, we chose to use a single NN that constructs the forcing term with only the information from inside the element.

Following some work implementing and setting the parameters for the Navier-Stokes we found very interesting results were we see the methodology capable of adapting to a wide range of Reynolds numbers (30, 200 and 1600 with LES). The smoother the solution, the less forcing the system needs and hence the lower the error. However we found that in high Reynolds number when using a coarse mesh the NN gives us a better sub-grid model than a Smagorinsky LES model.

In these cases we used a high polynomial order of P8 but since the NN is not perfect we loose accuracy in the low order modes. We found that we get a similar error as for P5 or P6 and the cost per iteration is even lower than in a P4 case, showing that this method can be seen as a correction relieving the computational cost for DGSEM schemes.

4.2 Future work

There are many possibilities to test on this new methodology. Their main objective should be to increase the accuracy, reduce even more the cost and increase the robustness. For example, it would be interesting to train the NN during the time the scheme is generating data and hence reduce the training process or testing different equations (RANS), larger applications and wall bounded turbulence.

Another thing to try is the use of CNN to prevent the loss of spatial information and/or RNN where we would not loose information regarding temporal order. These types of NNs could be applied locally if the cost of evaluating these NNs ends up being too large. It might also be interesting is to classify the domain into different regions and use different NNs in each region that might capture the physics faster.

Regarding the reconstruction, we showed in the 1D case how a deep neural network is capable of that. However we should address that other NNs such as GANs have shown good results in reconstructions for fluid mechanics [22].

These models are very general and do not have any particular mathematical form. Another thing that could be tested is to have a given formulation for the forcing term (e.g. advection diffusion form, production terms, sinks etc) where we determine the different terms of the equations. This approach is more classical but might also be more physical and understandable.

APPENDIX A

Equations

In this appendix we will present the formulation for the equations we have used for testing the methodology: 1D Burgers' equations and 3D compressible Navier-Stokes equations. We also give a brief insight into the Taylor Green vortex test case with its initial conditions and its characteristics, which is used for the 3D test cases.

Burgers' equation

The 1D Burgers' equation is the non linear version of the advection diffusion equation, where now the variable u is also the transport velocity. The viscous Burgers' equation reads:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(-\frac{u^2}{2} + \nu\frac{\partial u}{\partial x}\right), \tag{A.1}$$

$$u(-1;t) = f_l(t), \quad u(1;t) = f_r(t), \quad u(x;0) = f_0(x), \tag{A.2}$$

where $t \in \mathbb{R}^+$ is time, $x \in [-1,1]$ is the spatial coordinate, $u(x,t) : \mathbb{R}\times\mathbb{R} \to \mathbb{R}$ is the fluid variable, $f_l(t) : \mathbb{R} \to \mathbb{R}$ and $f_r(t) : \mathbb{R} \to \mathbb{R}$ represent the left and right boundary conditions respectively, $f_0(x) : \mathbb{R} \to \mathbb{R}$ is the initial condition and $\nu \in \mathbb{R}$ is the kinematic viscosity.

Compressible 3D Navier-Stokes equations

The 3D Navier–Stokes equations can be compactly written as,

$$\vec{u}_t + \nabla \cdot \vec{\boldsymbol{F}}_e = \nabla \cdot \vec{\boldsymbol{F}}_v, \tag{A.3}$$

where $\vec{u}$ is the vector of conservative variables $\vec{u} = [\rho, \rho v_1, \rho v_2, \rho v_3, \rho e]^T$, $\vec{\boldsymbol{F}}_e$ are the inviscid, or Euler equations fluxes,

$$\boldsymbol{F}_e = \begin{bmatrix} \rho v_1 & \rho v_2 & \rho u_3 \\ \rho v_1^2 + p & \rho v_1 v_2 & \rho v_1 v_3 \\ \rho v_1 v_2 & \rho v_2^2 + p & \rho v_2 v_3 \\ \rho v_1 v_3 & \rho v_2 v_3 & \rho v_3^2 + p \\ \rho v_1 H & \rho v_2 H & \rho v_3 H \end{bmatrix}, \tag{A.4}$$

where ρ, e, H and p are the density, total energy, total enthalpy, and pressure respectively, and $\vec{v} = [v_1, v_2, v_3]^T$ is the velocity. Additionally, $\vec{F}_v$ defines the viscous fluxes,

$$\vec{F}_v(\mu, \vec{v}, \nabla\vec{v}) = \begin{bmatrix} 0 & 0 & 0 \\ \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \\ \sum_{j=1}^{3} v_j \tau_{1j} + \kappa T_x & \sum_{j=1}^{3} v_j \tau_{2j} + \kappa T_y & \sum_{j=1}^{3} v_j \tau_{3j} + \kappa T_z \end{bmatrix}, \tag{A.5}$$

where κ is the thermal conductivity, T_x, T_y and T_z denote the gradients of temperature and the stress tensor $\boldsymbol{\tau}$ is defined as $\boldsymbol{\tau} = \mu(\nabla\vec{v} + (\nabla\vec{v})^T) - 2/3\mu\boldsymbol{I}\nabla\cdot\vec{v}$, with μ the dynamic viscosity, and $\boldsymbol{I}$ is the three-dimensional identity matrix.

Discontinuous Galerkin discretisation

We consider the approximation of systems of conservation laws,

$$\partial_t \mathbf{q} + \nabla \cdot \vec{\boldsymbol{f}} = \mathbf{0}, \quad \text{in } \Omega, \tag{A.6}$$

subject to appropriate boundary conditions, where $\mathbf{q}$ is the state vector of conserved variables, and $\vec{\boldsymbol{f}}$ is the flux block vector, which depends on $\mathbf{q}$. In an advection-diffusion conservation law, such as the Navier-Stokes equations, the flux vector can be written as

$$\vec{\boldsymbol{f}} = \vec{\boldsymbol{f}}^a(\mathbf{q}) - \vec{\boldsymbol{f}}^v(\mathbf{q}, \nabla\mathbf{q}), \tag{A.7}$$

where $\vec{f}^a$ is the advective flux and $\vec{f}^v$ is the diffusive flux. Because of the dependency of the diffusive flux on $\nabla\mathbf{q}$, (A.6) is a second order PDE. Following Arnold et al. [**?**], (A.6) can be rewritten as a first-order system,

$$\begin{cases} \partial_t \mathbf{q} + \nabla \cdot \left(\vec{f}^a(\mathbf{q}) - \vec{f}^v(\mathbf{q}, \vec{g}) \right) = \mathbf{0} \ , \text{ in } \Omega, & \text{(A.8a)} \\ \nabla \mathbf{q} = \vec{g}, \text{ in } \Omega. & \text{(A.8b)} \end{cases}$$

To obtain the DGSEM-version of (A.8), all variables are approximated by piece-wise Lagrange interpolating polynomials of order N that are continuous in each element: $\mathbf{q} \leftarrow \mathbf{q}^N$, $\vec{f} \leftarrow \vec{f}^N$ and $\vec{g} \leftarrow \vec{g}^N$. Furthermore, (A.8a) and (A.8b) are multiplied by an arbitrary polynomial (test function) of order N and numerically integrated by parts inside each element of a mesh with a quadrature rule of order N, to obtain

$$\begin{cases} J_j w_j \partial_t \mathbf{q}_j^N - \int_{\Omega^e}^N \vec{f}^N \cdot \nabla \phi_j d\Omega^e + \int_{\partial\Omega^e}^N f\, \phi_j \mathrm{d}S^e = \mathbf{0}, & \text{(A.9a)} \\ \qquad - \int_{\Omega^e}^N \mathbf{q}^N \nabla \phi_j d\Omega^e + \int_{\partial\Omega^e}^N \phi_j \hat{\mathbf{q}} \vec{n} dS^e = J_j w_j \vec{g}_j^N & \text{(A.9b)} \end{cases}$$

for each degree of freedom of each element. In (A.9), f and q are the numerical traces of the flux and the solution, respectively, the functions ϕ_j are the so-called basis functions, the J_j are the Jacobians of the geometry transformation the mesh is created with, and the w_j are the weights of the quadrature rule. The derivation of (A.9) is given in [31].

Taylor Green Vortex

Numerical experiments have been performed with the Taylor–Green Vortex (TGV) problem [67] for a range of Reynolds number $Re = 30 - 1600$. The TGV problem has been widely used to study Large Eddy Simulation turbulent closure models in the context of DG discretisations [72, 73, 74]. The configuration of the TGV problem is a three dimensional periodic box $[-\pi, \pi]^3$ with the initial condition,

$$\begin{aligned} \rho &= \rho_0, \\ v_1 &= V_0 \sin x \cos y \cos z, \\ v_2 &= -V_0 \cos x \sin y \cos z, \\ v_3 &= 0, \\ p &= \frac{\rho_0 V_0^2}{\gamma M_0^2} + \frac{\rho_0 V_0^2}{16} (\cos 2x + \cos 2y)(\cos 2z + 2). \end{aligned} \tag{A.10}$$

The Mach number is $M_0 = 0.08$ in all simulations included in this text. The reported quantities to measure the simulations accuracy are the kinetic energy rate,

$$\epsilon = -\frac{dK}{dt} = -\frac{1}{|\Omega|}\frac{d}{dt}\int_\Omega \frac{1}{2}\rho V^2 d\boldsymbol{x}, \tag{A.11}$$

the enstrophy,

$$\zeta = \frac{1}{2|\Omega|}\int_\Omega (\nabla \times \boldsymbol{v})^2 d\boldsymbol{x}, \tag{A.12}$$

the numerically introduced dissipation estimated with both ϵ and ζ,

$$\mu \simeq \frac{\epsilon}{2\zeta} = -\frac{dK/dt}{2\zeta}, \tag{A.13}$$

We can also present some particular results for Reynolds number of 1600 together with LES where the dissipation curve with polynomial order $P = 8$ is:

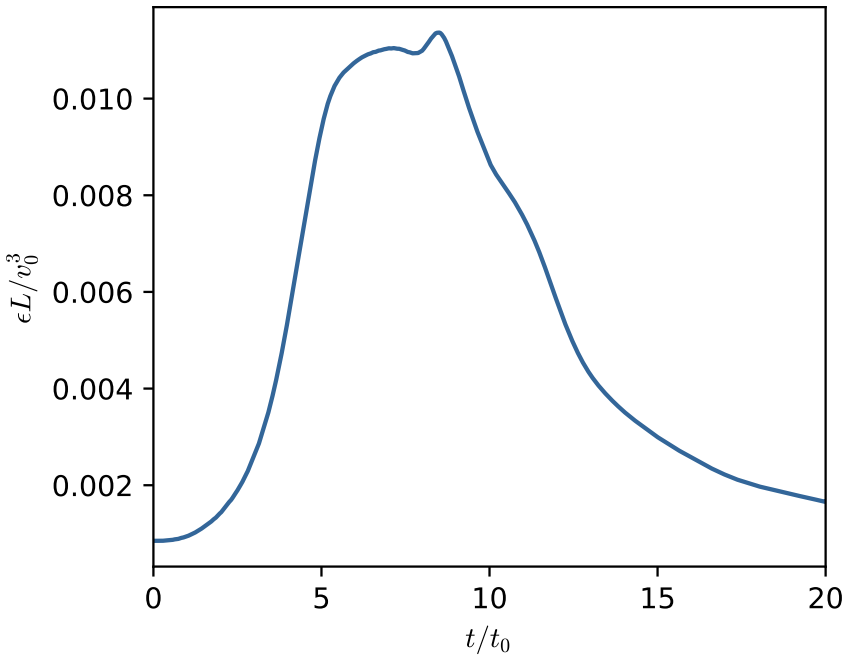


Figure A.1: Taylor Green Vortex dissipation for Reynolds number 1600 and P8, DNS references can by found at [75].

Bibliography

[1] W. E. Schiesser, The Numerical Method of Lines: Integration of Partial Differential Equations, 1991.

[2] G. D. Smith, G. D. Smith, G. D. S. Smith, Numerical solution of partial differential equations: finite difference methods, Oxford university press, 1985.

[3] R. Eymard, T. Gallouët, R. Herbin, Finite volume methods, Vol. 7 of Handbook of Numerical Analysis, Elsevier, 2000. doi:https://doi.org/10.1016/S1570-8659(00)07005-8.
URL https://www.sciencedirect.com/science/article/pii/S1570865900070058

[4] O. C. Zienkiewicz, R. L. Taylor, J. Z. Zhu, The finite element method: its basis and fundamentals, Elsevier, 2005.

[5] J. Hernandez, M. Barros, Interpolación polinómica de alto orden Métodos espectrales Aplicación a problemas de contorno y de condiciones iniciales, 2020.

[6] B. Leonard, Note on the von neumann stability of explicit one-dimensional advection schemes, Computer Methods in Applied Mechanics and Engineering 118 (1) (1994) 29–46. doi:https://doi.org/10.1016/0045-7825(94)90105-8.
URL https://www.sciencedirect.com/science/article/pii/0045782594901058

[7] G.-S. Jiang, C.-W. Shu, Efficient implementation of weighted eno schemes, Journal of Computational Physics 126 (1) (1996) 202–228. doi:https://doi.org/10.1006/jcph.1996.0130.
URL https://www.sciencedirect.com/science/article/pii/S0021999196901308

[8] C.-W. Shu, High order weighted essentially nonoscillatory schemes for convection dominated problems, SIAM Review 51 (1) (2009) 82–126. arXiv:https://doi.org/10.1137/070679065, doi:10.1137/070679065.
URL https://doi.org/10.1137/070679065

[9] C. Canuto, M. Y. Hussaini, A. Quarteroni, T. A. Zang, Spectral methods: fundamentals in single domains, Springer Science & Business Media, 2007.

[10] C. Canuto, M. Y. Hussaini, A. Quarteroni, A. Thomas Jr, et al., Spectral methods in fluid dynamics, Springer Science & Business Media, 2012.

[11] M. Kompenhans, G. Rubio, E. Ferrer, and E. Valero, Adaptation strategies for high order discontinuous Galerkin methods based on tau-estimation, Journal of Computational Physics 306 (2016) 216 – 236.

[12] M. Kompenhans, G. Rubio, E. Ferrer, E. Valero, Comparisons of p-adaptation strategies based on truncation- and discretisation-errors for high order discontinuous Galerkin methods, Computers and Fluids 139 (2016) 36–46. doi:10.1016/j.compfluid.2016.03.026.

[13] E. Ferrer, An interior penalty stabilised incompressible discontinuous galerkin–fourier solver for implicit large eddy simulations, Journal of Computational Physics 348 (2017) 754–775. doi:10.1016/j.jcp.2017.07.049.

[14] A. M. Rueda-Ramírez, J. Manzanero, E. Ferrer, G. Rubio, E. Valero, A p-multigrid strategy with anisotropic p-adaptation based on truncation errors for high-order discontinuous Galerkin methods, Journal of Computational Physics 378 (2019) 209–233. arXiv:1806.11456, doi:10.1016/j.jcp.2018.11.009.

[15] S. L. Brunton, J. N. Kutz, Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control, Cambridge University Press, 2019. doi:10.1017/9781108380690.

[16] P. Garnier, J. Viquerat, J. Rabault, A. Larcher, A. Kuhnle, E. Hachem, A review on deep reinforcement learning for fluid mechanics, Computers & Fluids 225 (2021) 104973. doi:https://doi.org/10.1016/j.compfluid.2021.104973.
URL https://www.sciencedirect.com/science/article/pii/S0045793021001407

[17] Y. Bar-Sinai, S. Hoyer, J. Hickey, M. P. Brenner, Learning data-driven discretizations for partial differential equations, Proceedings of the National Academy of Sciences 116 (31) (2019) 15344–15349. doi:10.1073/pnas.1814058116.
URL http://dx.doi.org/10.1073/pnas.1814058116

[18] K. Stachenfeld, D. B. Fielding, D. Kochkov, M. Cranmer, T. Pfaff, J. Godwin, C. Cui, S. Ho, P. Battaglia, A. Sanchez-Gonzalez, Learned coarse models for efficient turbulence simulation (2021). doi:10.48550/ARXIV.2112.15275.
URL https://arxiv.org/abs/2112.15275

[19] A. Beck, M. Kurz, A perspective on machine learning methods in turbulence modeling, GAMM-Mitteilungen 44 (1) (2021) e202100002. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/gamm.202100002, doi:https://doi.org/10.1002/gamm.202100002.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.202100002

[20] R. Vinuesa, S. L. Brunton, The potential of machine learning to enhance computational fluid dynamics (2021). arXiv:2110.02085.

[21] L. Guastoni, A. Güemes, A. Ianiro, S. Discetti, P. Schlatter, H. Azizpour, R. Vinuesa, Convolutional-network models to predict wall-bounded turbulence from wall quantities, Journal of Fluid Mechanics 928 (2021) A27. doi:10.1017/jfm.2021.812.

[22] A. Güemes, S. Discetti, A. Ianiro, B. Sirmacek, H. Azizpour, R. Vinuesa, From coarse wall measurements to turbulent velocity fields through deep learning, Physics of Fluids 33 (7) (2021) 075121. arXiv:https://doi.org/10.1063/5.0058346, doi:10.1063/5.0058346.
URL https://doi.org/10.1063/5.0058346

[23] F. Manrique de Lara, E. Ferrer, Accelerating high order discontinuous galerkin solvers using neural networks: 1d burgers' equation, Computers & Fluids 235 (2022) 105274. doi:https://doi.org/10.1016/j.compfluid.2021.105274.
URL https://www.sciencedirect.com/science/article/pii/S0045793021003698

[24] H. Schlichting, K. Gersten, Boundary-Layer Theory, 8th Edition, Springer, 2000. doi:10.1007/978-3-642-85829-1.
URL http://dx.doi.org/10.1007/978-3-642-85829-1

[25] F. Miller, A. Vandome, J. McBrewster, Gauss's Law, VDM Publishing, 2009.
URL https://books.google.es/books?id=jgeNQgAACAAJ

[26] E. Toro, Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction, 2009. doi:10.1007/b79761.

[27] P. Roe, Approximate riemann solvers, parameter vector, and difference schemes, Journal of Computational Physics 43 (1981) 357–372. doi:10.1016/0021-9991(81)90128-5.

[28] P. D. Lax, Weak solutions of nonlinear hyperbolic equations and their numerical computation, Communications on Pure and Applied Mathematics 7 (1) (1954) 159–193. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/

cpa.3160070112, doi:https://doi.org/10.1002/cpa.3160070112.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.3160070112

[29] F. Bassi, S. Rebay, A high-order accurate discontinuous finite element method for the numerical solution of the compressible navier–stokes equations, Journal of Computational Physics 131 (2) (1997) 267–279. doi:https://doi.org/10.1006/jcph.1996.5572.
URL https://www.sciencedirect.com/science/article/pii/S0021999196955722

[30] K. Black, A conservative spectral element method for the approximation of compressible fluid flow, Kybernetika 35 (1) (1999) [133]–146.
URL http://eudml.org/doc/33415

[31] D. A. Kopriva, Implementing spectral methods for partial differential equations: Algorithms for scientists and engineers, Springer Science & Business Media, 2009.

[32] E. Ferrer, R. Willden, A high order Discontinuous Galerkin finite element solver for the incompressible Navier-Stokes equations, Computers & Fluids 46 (1) (2011) 224–230.

[33] A. M. Rueda-Ramírez, Efficient space and time solution techniques for high-order discontinuous galerkin discretizations of the 3d compressible navier-stokes equations (2019).

[34] J. M. Torrico, A high?order discontinuous galerkin multiphase flow solver for industrial applications (Enero 2020).
URL https://oa.upm.es/65136/

[35] F. M. de Lara, Método numérico de volúmenes finitos para la simulación de flujos aerodinámicos (Junio 2020).
URL http://oa.upm.es/67882/

[36] J. A. H. Ramos, Cálculo Numérico en Ecuaciones Diferenciales Ordinarias, Aula Documental de Investigación, 2001.

[37] M. Hirsch, S. Smale, Differential equations, dynamical systems, and linear algebra, no. 60 in Pure and applied mathematics, Acad. Press, 1974.
URL http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+022705937&sourceid=fbw_bibsonomy

[38] J. D. Lambert, et al., Numerical methods for ordinary differential systems, Vol. 146, Wiley New York, 1991.

[39] K. Burnham, Artificial intelligence vs. machine learning: What's the difference? (2020).
URL https://www.northeastern.edu/graduate/blog/artificial-intelligence-vs-machine-learning-whats-the-difference/

[40] Stefanini, What's the difference between machine learning and artificial intelligence? (2021).
URL https://stefanini.com/en/trends/news/difference-between-machine-learning-and-artificial-intelligence

[41] S. Ray, Commonly used machine learning algorithms (with python and r codes) (2017).
URL https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/

[42] R. Bellman, Dynamic Programming, Dover Publications, 1957.

[43] R. Arrabales, Deep learning: qué es y por qué va a ser una tecnología clave en el futuro de la inteligencia artificial (2016).
URL https://www.xataka.com/robotica-e-ia/deep-learning-que-es-y-por-que-va-a-ser-una-tecnologia-clave-en-el-futuro-de-la-inteligencia-artificial

[44] K. Strachnyi, Brief history of neural networks (2019).
URL https://medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec

[45] M. T. Jones, A neural networks deep dive (2017).
URL https://developer.ibm.com/articles/cc-cognitive-neural-networks-deep-dive/

[46] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Networks 61 (2015) 85–117. doi:10.1016/j.neunet.2014.09.003.
URL https://doi.org/10.1016%2Fj.neunet.2014.09.003

[47] L. Bottou, On-line learning and stochastic approximations, in: In On-line Learning in Neural Networks, Cambridge University Press, 1998, pp. 9–42.

[48] B. Polyak, Some methods of speeding up the convergence of iteration methods, USSR Computational Mathematics and Mathematical Physics 4 (5) (1964) 1–17. doi:https://doi.org/10.1016/0041-5553(64)90137-5.
URL https://www.sciencedirect.com/science/article/pii/0041555364901375

[49] Z. Hao, Y. Jiang, H. Yu, H. Chiang, Adaptive learning rate and momentum for training deep neural networks, CoRR abs/2106.11548 (2021). arXiv:2106.11548.
URL https://arxiv.org/abs/2106.11548

[50] S. Ruder, An overview of gradient descent optimization algorithms (2016).
URL https://ruder.io/optimizing-gradient-descent/index.html#batchgradientdescent

[51] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization (2014). doi:10.48550/ARXIV.1412.6980.
URL https://arxiv.org/abs/1412.6980

[52] S. Linnainmaa, Taylor expansion of the accumulated rounding error, BIT 16 (2) (1976) 146–160. doi:10.1007/BF01931367.
URL https://doi.org/10.1007/BF01931367

[53] R. P. Adams, Computing gradients with backpropagation.
URL https://www.cs.princeton.edu/courses/archive/fall18/cos324/files/backprop.pdf

[54] J. Witaszek, Backpropagation: Theory, architectures, applications: By yves chauvin and david e. rumelhart (eds). lawrence erlbaum, hillsdale, nj, hove, uk, 1995. isbn 0-8058-1258-x, pp. 561 (1995).

[55] F. V. Veen, The neural network zoo (2016).
URL https://www.asimovinstitute.org/author/fjodorvanveen/

[56] S. Saha, A comprehensive guide to convolutional neural networks (2018).
URL https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[57] A. Pai, Cnn vs. rnn vs. ann – analyzing 3 types of neural networks in deep learning (2020).
URL https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/

[58] J. Ott, M. Pritchard, N. Best, E. Linstead, M. Curcic, P. Baldi, A fortran-keras deep learning bridge for scientific computing (2020). arXiv:2004.10652.

[59] I. C. Education, Recurrent neural networks (2020).
URL https://www.ibm.com/cloud/learn/recurrent-neural-networks

[60] Z. M. Kim, H. R. Oh, H.-G. Kim, C.-G. Lim, K.-J. Oh, H.-J. Choi, Modeling long-term human activeness using recurrent neural networks for biometric data, BMC Medical Informatics and Decision Making 17 (05 2017). doi:10.1186/s12911-017-0453-1.

[61] Y. Bengio, R. Ducharme, P. Vincent, C. Janvin, A neural probabilistic language model, J. Mach. Learn. Res. 3 (null) (2003) 1137–1155.

[62] H. Hewamalage, C. Bergmeir, K. Bandara, Recurrent neural networks for time series forecasting: Current status and future directions, International Journal of Forecasting 37 (1) (2021) 388–427. doi:https://doi.org/10.1016/j.ijforecast.2020.06.008.
URL https://www.sciencedirect.com/science/article/pii/S0169207020300996

[63] M. Morimoto, K. Fukami, K. Zhang, A. G. Nair, K. Fukagata, Convolutional neural networks for fluid flow analysis: toward effective metamodeling and low dimensionalization, Theoretical and Computational Fluid Dynamics 35 (5) (2021) 633–658. doi:10.1007/s00162-021-00580-0.
URL https://doi.org/10.1007%2Fs00162-021-00580-0

[64] L. Guastoni, A. Güemes, A. Ianiro, S. Discetti, P. Schlatter, H. Azizpour, R. Vinuesa, Convolutional-network models to predict wall-bounded turbulence from wall quantities (2020). doi:10.48550/ARXIV.2006.12483.
URL https://arxiv.org/abs/2006.12483

[65] E. Ferrer, G. Rubio, G. Ntoukas, W. Laskowski, O. Mariño, S. Colombo, A. M. Gabín, F. Manrique de Lara, D. Huergo, J. Manzanero, A. M. Rueda-Ramirez, D. Kopriva, E. Valero., Horses3d: a high order discontinuous galerkin solver for flow simulations and multi-physics applications, Computer Physics Communications (under review) (2022).

[66] A. S. Iskhakov, N. T. Dinh, E. Chen, Integration of neural networks with numerical solution of pdes for closure models development, Physics Letters A 406 (2021) 127456. doi:10.1016/j.physleta.2021.127456.
URL http://dx.doi.org/10.1016/j.physleta.2021.127456

[67] G. I. Taylor, A. E. Green, Mechanism of the Production of Small Eddies from Large Ones, Proceedings of the Royal Society of London Series A 158 (895) (1937) 499–521. doi:10.1098/rspa.1937.0036.

[68] W. Laskowski, G. Rubio, E. Valero, E. Ferrer, A functional oriented truncation error adaptation method, Journal of Computational Physics 451 (2022) 110883. doi:https://doi.org/10.1016/j.jcp.2021.110883.

URL https://www.sciencedirect.com/science/article/pii/S0021999121007786

[69] G. Rubio, Truncation error estimation in the discontinuous galerkin spectral element method, Ph.D. thesis, PhD thesis, Technical University Madrid, 2015.(Cited on pages 9, 14, 17, 18 ... (2015).

[70] P. Roe, Approximate riemann solvers, parameter vectors, and difference schemes, Journal of Computational Physics 43 (2) (1981) 357–372. doi:https://doi.org/10.1016/0021-9991(81)90128-5.
URL https://www.sciencedirect.com/science/article/pii/0021999181901285

[71] F. Bassi, S. Rebay, A high-order accurate discontinuous finite element method for the numerical solution of the compressible navier-stokes equations, Journal of Computational Physics 131 (2) (3 1997). doi:10.1006/jcph.1996.5572.
URL https://www.osti.gov/biblio/494292

[72] G. J. Gassner, A. R. Winters, D. A. Kopriva, Split form nodal discontinuous galerkin schemes with summation-by-parts property for the compressible euler equations, Journal of Computational Physics 327 (2016) 39–66. doi:10.1016/j.jcp.2016.09.013.
URL https://doi.org/10.1016%2Fj.jcp.2016.09.013

[73] R. Moura, P. Fernandez, G. Mengaldo, Diffusion and dispersion characteristics of hybridized discontinuous Galerkin methods for under-resolved turbulence simulations, in: APS Division of Fluid Dynamics Meeting Abstracts, APS Meeting Abstracts, 2017, p. F31.007.

[74] J. Manzanero, E. Ferrer, G. Rubio, E. Valero, Design of a smagorinsky spectral vanishing viscosity turbulence model for discontinuous galerkin methods, Computers & Fluids 200 (2020) 104440. doi:https://doi.org/10.1016/j.compfluid.2020.104440.
URL https://www.sciencedirect.com/science/article/pii/S0045793020300165

[75] M. Chávez-Modena, A. Martínez-Cava, G. Rubio, E. Ferrer, Optimizing free parameters in the d3q19 multiple-relaxation lattice boltzmann methods to simulate under-resolved turbulent flows, Journal of Computational Science 45 (2020) 101170. doi:https://doi.org/10.1016/j.jocs.2020.101170.
URL https://www.sciencedirect.com/science/article/pii/S1877750320304713