National Technical University of Athens

School of Naval Architecture and Marine Engineering
Laboratory of Ship & Marine Hydrodynamics

Diploma Thesis

Fast flow prediction along airfoils operating at transonic conditions using Machine Learning

Rekoumis Konstantinos

February, 2022

Supervisor: Assistant Prof. George Papadakis
Supervisor: Associate Prof. George Papalamprou
Committee: Professor Gregory Gregoropoulos

Abstract

This Thesis's main focus is how we can couple Fluid Dynamics with Artificial Intelligence. This work is not an original idea as many other scientists have already explored this field [1][2][3], yet it is fascinating to further research and experiment as it shows immense potential. Based on the work of Hui et al. [1], a system of Convolutional Neural Networks will be created where each Network would be responsible for predicting the Coefficient of Pressure distribution for an airfoil's two sides (top and bottom sides). The airfoil of choice is the RAE-2822, operating under subsonic conditions close to the Critical Mach number. This way we can also study whether the Neural Network can predict the formation of sonic waves, phenomena with great mathematical and physical interest because of their extremely non-linear behavior.

Also, an interesting concept used by Hui et al. is utilizing the Signed Distance Function to colorize the input image's pixels. Signed Distance Function enables us to describe more complex geometry with less image resolution. It achieves that by colorizing each pixel according to the distance information between the pixel's center and its nearest geometry point. That leads to packing more information into fewer pixels by utilizing almost the entirety of available pixels.

To train and test the Neural Networks, the RAE-2822 is randomly uniformly deformed for deformation percentages $\in [-20, 20]$ %, with 1000 specimens being used for training and 500 for testing. Each variant's Cp distribution was calculated using the CFD solver MaPFlow. Then the Cp distribution's values were extracted at specific length intervals for each side, thus creating a file storing these values for each side. This process, along with the creation of the SDF formatted images, constitute the data generation process. Then this data were used to train the Neural Network.

After training the Networks, we study their accuracy in predicting the Cp distribution of previously unknown specimens and repeat the process for 4000 Epochs. Finally, the Networks' error convergence per Epoch history would be studied both for the training and the testing sets. Also, the time per Epoch data as long as the single case prediction time data would be studied to validate the Networks' prediction speed.

Following the verification of the Neural Networks' accuracy, we will study the influence of the minibatch size in the training-testing precision and speed. Also, we will examine the Networks' precision in predicting Cp Distribution for airfoils out of the original training and testing range. Finally, a short and simplistic application of the trained Networks' will be presented; a simple geometry optimizer, whose purpose is to maximize the Lift capacity by optimizing the original RAE-2822 geometry for specific Free-flow conditions.

Acknowledgements

This Diploma Thesis has been carried out at the Laboratory of Ship and Marine Hydrodynamics of the School of Naval Architecture and Marine Engineering of the National Technical University of Athens, under the Supervision of the Assistant Professor Mr. George Papadakis. Due to this Thesis focus on Machine Learning in conjunction with Aerodynamics, this Thesis was also supervised by Associate Professor Mr. George Papalambrou of the Laboratory of Marine Engineering of the School of Naval Architecture and Marine Engineering of the National Technical University of Athens.

First, I would like to thank Assistant Professor Mr. Papadakis for accepting my ambitious Thesis proposal and letting me carry it out under his supervision. Also, I would like to thank him for his constant support, understanding, motivation, as well as his valuable knowledge and feedback during the entirety of my Thesis.

Second, I would like to thank Associate Professor Mr. George Papalambrou for providing me with valuable knowledge and feedback, during the time I spent developing the Neural Networks for this Thesis. His expertise on the field of Machine Learning was a great motivation for me to conduct this Thesis as I had zero knowledge of Machine Learning prior to this Thesis.

I am also grateful to Professor Mr. Gregory Gregoropoulos for evaluating my work and being a member of my supervisors committee.

I would also like to thank Doctoral Student Mr. Dimitris Ntouras for helping me with creating my own airfoil mesh and passing me his knowledge on computational grid creation.

Also, I am grateful to my family, for their encouragement, support, and understanding all those years I devoted studying and completing my Msc of Naval Architecture and Marine Engineering. They always had faith in me, encouraging me to pursue my dreams and ambitions.

Last but not least, I want to thank my friends for all the good times we had together during our studies, who constantly supported me, and spent enormous amounts of time listening to me being excited or disappointed during my studies, and especially during my Thesis. Finally, I want to thank

Alex for being a constant source of motivation and support, never doubting me even when I did so myself. Also, I want to thank her for her constructive feedback while I was writing this essay.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Introduction to the physical problem

Understanding how nature works and then how to harness its power to create various constructs has always been the main point of interest of engineering. Acquiring this knowledge led to humans engineering many everyday things, from enormous skyscrapers to tiny wireless devices. In our field of expertise, Naval Architecture and Marine Engineering, it is crucial to understand and calculate with precision how fluids behave. Even from ancient times, humankind showed interest in studying fluid mechanics. The most known piece of research, from this time, on this field, was the work of Archimedes, who investigated fluid statics, buoyancy, and formulated the famous Archimede's principle (eq. 1.1). Later many renowned scientists, like Isaac Newton, Leonardo Da Vinci, Daniel Bernoulli, contributed to furthering our knowledge of fluid mechanics by running many experiments, writing equations, like Blaise Pascal's law (eq.: 1.2), and creating various contraptions, like Evangelista Torricelli's barometer. Many other scientists over the years have researched the fluids' nature (fluid statics and dynamics), trying the find the underlying mechanics of turbulence, viscosity, to name a few, with their collective work being described in the Navier-Stokes equations.

$$F = \rho g \nabla \quad \text{[N (SI)]} \tag{1.1}$$

, Buoyancy force F equals to ρ the fluid's density $[kg/m^3$ SI] times g gravitational acceleration $[m/s^2$ SI] times the submerged volume ∇ $[m^3$ SI] (The Archimede's Buoyancy principle)

The Navier–Stokes equations mathematically express conservation of momentum and mass for Newtonian fluids (eq. 1.3). Sometimes they are ac-

Figure 1.1: Leonardo DaVinci's turbulence sketch

Leonardo DaVinci has done many experiments to study the flow of fluids and especially water and blood. This figure is his sketch of observing the creation of eddies during turbulent flow. [4]

companied by an equation of state relating pressure, temperature, and density. They arise from applying Isaac Newton's second law to fluid motion, together with the assumption that the stress in the fluid is the sum of a diffusing viscous term (proportional to the gradient of velocity) and a pressure term—hence describing viscous flow. The Navier–Stokes equations are useful because they describe the physics of many phenomena of scientific and engineering interest. They are used to model the weather, ocean currents, water flow in a pipe, airflow around a wing, and many other natural phenomena [5][6].

$$\Delta p = \rho \ g \ \Delta h \tag{1.2}$$

,Δp is the hydrostatic pressure (given in pascals in the SI system), or the difference in pressure at two points within a fluid column, due to the weight of the fluid)
ρ is the fluid's density in kg/m^3 (SI)
g is the local acceleration in m/s^2 (SI)
Δh is the height of fluid above the point of measurement, or the difference in elevation between the two points within the fluid column (in meters).

(Blaise Pascal's principle of transmission of fluid-pressure)

Navier-Stokes equations mathematically express what people intuitively thought of fluids. Ship-building is one of the most ancient fields of application of fluid mechanics, but due to the lack of extensive understanding of fluid mechanics, ships were built based on intuition, observation, and experience. However, the beginning of the 20^{th} century introduced new technologies, thus challenges to overcome. Technologies like aviation, automobiles, and civil engineering emphasized the need to further our knowledge on how fluids (especially air and water) work.

$$\rho \frac{D\vec{u}}{Dt} = \rho\vec{g} - \nabla p + \mu\Delta\vec{u} \tag{1.3}$$

The Navier-Stokes equation for incompressible flow where,
$\frac{D\vec{u}}{Dt}$ is the material derivative of the fluid's velocity vector further analyzed as $\frac{D\vec{u}}{Dt} = \frac{\partial\vec{u}}{\partial t} + \nabla\vec{u}$,
p the pressure field,
μ the absolute or dynamic viscosity of the fluid

More and more scientists studied fluids in greater detail, focusing on discovering the secrets behind viscous and inviscid flow, turbulence, sonic and hypersonic flows, and many other aspects of fluid mechanics. This trend helped expand the boundaries of naval architecture as well, as the engineers transformed the empirical knowledge of the past into equations, normalized data charts, and scientifically proven methodologies to design a ship. Famous works in this domain are the :

1. Froude and Reynolds number equations:

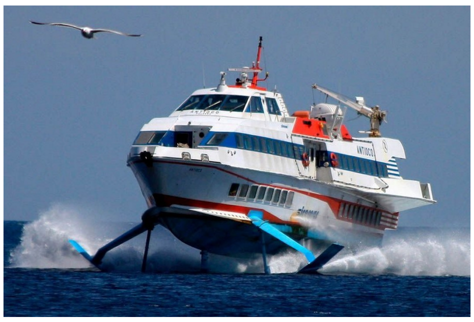
$$Re = \frac{Lu}{\nu} \quad \text{Reynold's equation,} \tag{1.4}$$

$$Fn = \frac{u}{\sqrt{gL}} \quad \text{Froude's equation} \tag{1.5}$$

   Where, u is the fluid's velocity, L the boundary's significant dimension, ν the fluid's kinematic viscosity, and g the acceleration of gravity

2. Froude and Hughes design methods for scaling models to actual size vessels

3. Normalized systematic series to design ships like Formdata, Taylor Series, to name a few.


(a) A Hydrofoil vessel during planning operation [7]

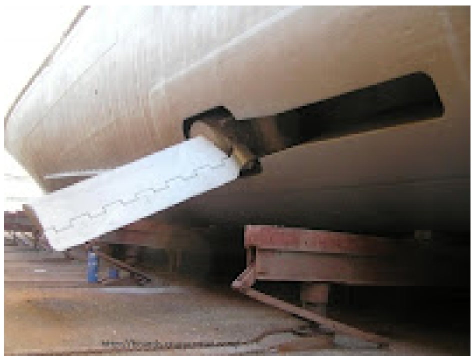

(b) Ship stabilizer fin [8]

Figure 1.2: Images of a Hydroplane Vessel and Ship Stabilizer Fins

4. Normalized systematic propeller series, like Wageningen B-series.

These pieces of work made Shipbuilding a consistent, cost-efficient, and scientifically proven process, deobfuscating any mysteries and inaccuracies surrounding ship construction in the past.

Even though our field of interest lies within naval architecture and marine engineering, studying airfoils and general air dynamics is quite beneficial. In their core airfoils, and hydrofoils are more similar than different. Actually, for some series, a foil is utilized both as hydrofoil and airfoil, for example, the NACA series (fig.:1.3). Hydrofoils in our field are mainly used to create Lift force, with which we can control a vessel's movements. To be precise, the ship's rudder is a hydrofoil controlling Lateral rotation (Yaw), and stabilizers, hydrofoils placed along the ship's hull, the Longitudinal and Transverse rotational movements (Pitch and Roll respectively) (fig.: 1.2b). Furthermore, they are used in Hydrofoil Crafts to create Lift to raise the ship's hull out of the water and then support its weight during operation (fig.: 1.2a). Finally, the propeller is the most complex hydrofoil a ship has, where each blade is a complex three-dimensional hydrofoil.

However, airfoils are also a crucial part of marine engineering. Airfoils are inside marine engines' turbochargers, air-compressors, steam turbines, and many other applications. In these applications, an airfoil might face conditions even worse than in aviation; being subjected to extreme pressures and temperatures, contrasting to being an airplane's wing, flying through the clear air. Nevertheless, studying an airfoil under heavy thermal and pressure loads can be very tough compared to studying its free-flow characteristics under various speed and temperature conditions.

Even-though, it can be easier to study the free flow of an airfoil, that does

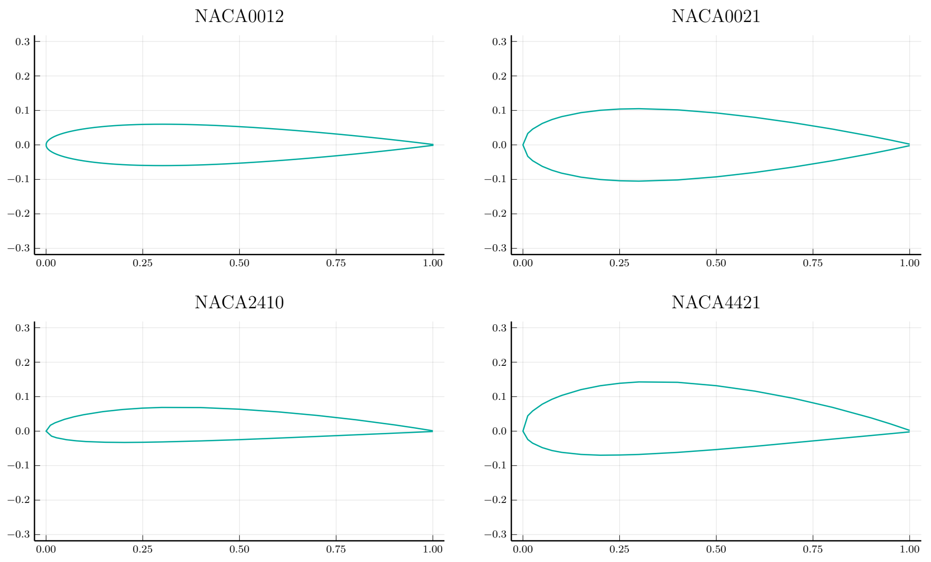


Figure 1.3: NACA series airfoil plots
NACA series are one of the earliest developed systematic series. In Naval Architecture, their symmetrical variants (like NACA0012 and NACA0021) are commonly used as the ship's rudder.

not eliminate the problem's complexity whatsoever. One cannot simply apply the Navier-Stokes equations (which are notoriously difficult to solve and have a "mathematical bounty on their head" [1]) to solve the flow problem around an airfoil and calculate the flow field and pressure distribution. Turbulence and viscosity introduce huge implications to any attempt to solve even the simplest fluid mechanics problem.

Viscosity is a physical property of the fluids, which quantifies the internal friction between the molecules of a fluid [6]. Viscosity is responsible for the boundary layers near the boundaries with solid geometry, as the fluid gets a gradient velocity u being 0 at the solid geometry's boundary, and then equal to the Free-flow velocity further from the boundary. Boundary layers are also created when two fluids are in relative motion with each other, for example, when the wind (air - gases are fluids as well) blows over the sea (water). Boundary layers' complexity can vary from very simple during laminar flow to extremely difficult to solve during turbulent flow (fig. 1.4).

[1]It has not yet been proven whether smooth solutions always exist in three dimensions—i.e. they are infinitely differentiable (or even just bounded) at all points in the domain. This is called the Navier–Stokes existence and smoothness problem. The Clay Mathematics Institute has called this one of the seven most important open problems in mathematics and has offered a 1 million USD prize for a solution or a counterexample.[5]

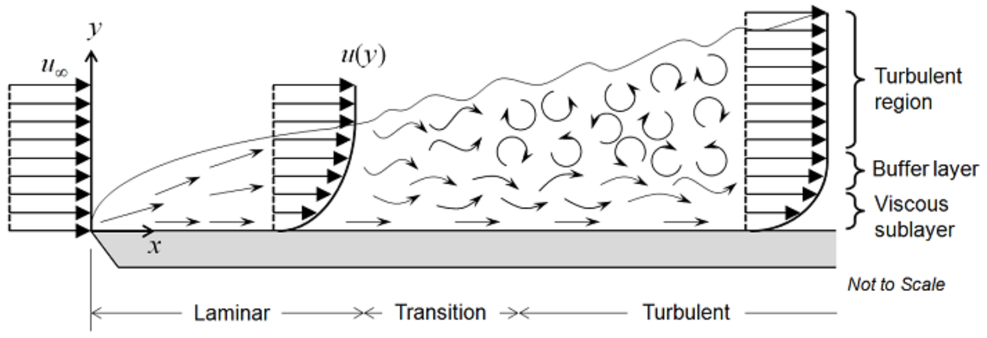


Figure 1.4: Boundary Layer over a flat plate
The boundary layer's state is heavily depended on the Raynolds number. So even when the $u_\infty = cons$, the boundary layer may enter a turbulent state if the plate is sufficiently long. That emphasizes even more the need to better understand turbulence, as rarely there wil be an application where only laminar flow in the boundary layer will exist.(image courtesy of [9]).

On the other hand, turbulence is the manifestation of chaos in the physical world. Many famous physicists and engineers have studied turbulence, but still, we have no clear answer on how it works. As quoted by the renowned physicist Richard Feynman: "Turbulence is the most important unsolved problem in classical physics." [10]. Turbulence is a quantity that cannot be deterministically calculated, as time or mass, and we use statistics to quantify it. However, turbulence is a natural phenomenon that we can predict when will occur by using the Reynolds Number eq.: 1.4. Reynolds Number normalizes the fluid velocity and the solid boundary's significant -for the problem- dimension against the fluid's kinematic viscosity. In this way, we can have a metric of whether the fluid's kinetic energy is high enough to overflow parallel flow (laminar flow) and the flow to -become turbulent, or enter a chaotic state. While studying ships [2] and foils (both hydrofoils and airfoils), the significant dimension is their longitudinal length. On the other side, studying flow inside tubes of any kind, the significant dimension is their diameter.

Focusing now on the problem at hand, during this thesis, we will focus on calculating an airfoil's pressure distribution during subsonic flight. A fluid achieves sonic flow when the fluid's velocity is equal to its speed of sound. Speed of sound is an innate property of every material, describing the speed

[2]With some imagination, we can observe that a ship resembles a symmetric hydrofoil with a variable thickness along the lateral axis)

with which sound waves travel through it and, of course, ranges from one to another. Subsonic flows around airfoils have various interesting properties to investigate, like shock waves. Shock waves form on the suction side of an airfoil where the fluid's speed locally exceeds the speed of sound.

To create Lift force, airfoils need to operate on different Pressure distributions between their two sides. They require a low and a high-pressure side. The difference between the two sides creates a differential pressure distribution that produces a resultant force to the desired direction. However, the Bernoulli theorem (eq.: 1.6) suggests that assuming no losses of kinetic energy, when studying a single line of flow, a pressure drop results in a velocity increase and vice versa. So on the low-pressure side, there is a local increase of the fluid's velocity, while on the high pressure side, there is a local decrease. This local increase results in the fluid acquiring Mach 1 speed, or sonic speed. However, as the fluid cannot maintain this unstable state for very long, leading the flow to violently collapse back to subsonic speed, creating an energy spike that dissipates through the fluid with the speed of sound. This energy spike is perceived by humans as a loud noise, thus calling it a shock wave. From a mathematical standpoint, shock waves are discontinuous and non-linear phenomena that introduce enormous complexity, and they are fascinating to research.

By now, one can easily understand that even though we have the tools to study fluids, the enormous mathematical complexity of their physics handicaps our efforts. To combat this problem, we tend to make reasonable simplifications that reduce the mathematical complexity, yet they introduce a small error to our calculations. For example, it is common to assume that air is an inviscid fluid [3]; an approximation that introduces minimal error and simplifies a lot the problem. Simplification enables us to reduce the complexity of the Navier-Stokes equations and transform them to simpler equations that are easier to solve.

$$\frac{p}{\rho} + gz + \frac{1}{2}u^2 = const \tag{1.6}$$

Where, u is the fluid's velocity, p the pressure, z is the lateral distance from the plane of reference, ρ the fluid's density, and g the acceleration of gravity (Bernoulli's principle)

Another common simplification is to study airfoils with high aspect ratio $(\frac{Length}{Beam})$ as infinitely long beams. Doing this we can find the flow field in two dimensions, as in an infinitely long beam the flow field would be the same at

[3] Air has a dynamic viscosity of 0.01803 mPa/s at 20℃ and 1 atm, while water has 1.0005 mPa/s under the same conditions

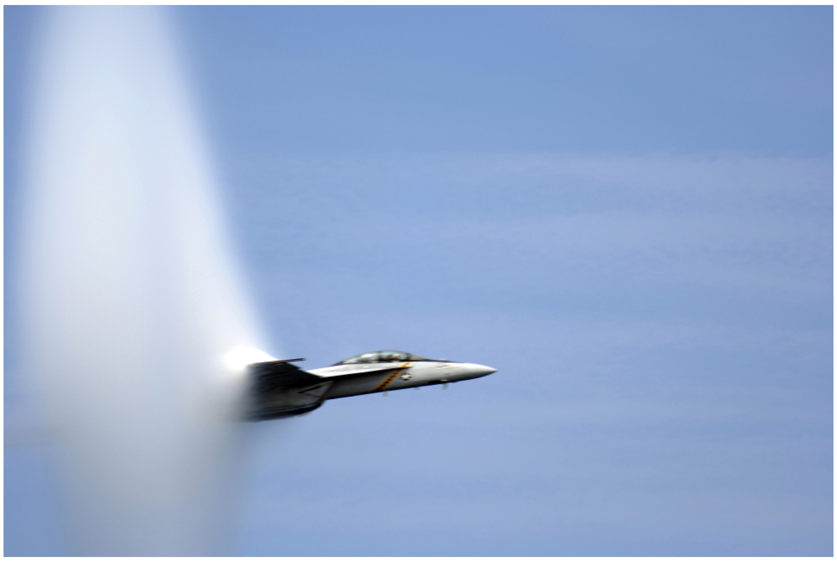


Figure 1.5: An F/A-18E Super Hornet reaches the speed of sound
As the airplane reaches the speed of sound, a sonic wave is formed. In the image it is the white mist that surounds the aft part of the plane. (photo courtesy of Jesse L. Dick [11])

every point along the airfoil's Length. Then we can solve the flow problem around only a transverse section of the airflow reducing the total calculations' time required. However, this is not absolutely right because this way we do not account for other significant phenomena, like the vortices which form at the foil's edge, or the flow where the foil is connected to main body of the structure (being an airplane's hull, a turbine's rotor drum, etc.).

Also, studying an foil in two dimensions enables us to use some clever mathematics to simplify the flow field's calculation. Many times the airfoils' geometry is not given by a single simple formula, like $y = cos(x)$ or $y = ax^2+bx+c$, making the integration vary hard. Instead, we can substitute the material nodes of the airfoil with a proper combination of fluid sources and sinks to emulate the flow around the foil [6]. This way, using complex analysis we can analytically solve the flow field around the airfoil. However, this method works under some important approximations, like having a steady-state laminar flow, a non viscous, non turbulent, non compressible fluid, etc.

As it becomes apparent, not all problems can be solved with approximations and geometry substitutions. When dealing with more complex problems, approximations can lead to significant inaccuracies. To deal with this need, engineers have created pieces of software that can resolve Navier-Stokes equations arithmetically. These programs, called Computational Fluid Dy-

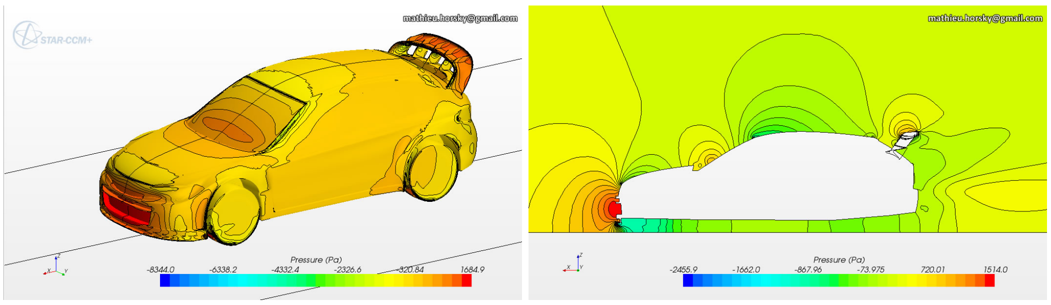


Figure 1.6: CFD simulation of a WRC racecar
In the images are presented CFD simulation's results for a WRC (World Rally Championship) racecar. Two different images are presented; one of the pressure distribution on the car's surface and one of the pressure distribution at the car's longitudinal midsection. CFD simulations are very useful in racing industry as small design changes can have a drastic impact on the car's aerodynamic performance, where engineers want to balance Downforce (or negative Lift) with Drag to achieve optimal performance. (photos courtesy of Matthieu Horsky [12])

namics Solvers (or CFD Solvers for short), can solve the flow field around any kind of geometry, under various conditions, and for every Newtonian fluid, where the Navier-Stokes equations are applicable. They can accurately model the boundary layer, vortex formation, flow field, pressure distribution, and, of course, in both three and two dimensions. Nowadays, there are many sophisticated CFD solvers available in the market, both open-source (free) and proprietary (paid), with notable mentions of OpenCFD's OpenFOAM™, ANSYS CFX™, Siemens Digital Industries STAR-CCM+™. For this thesis, MaPFlow is the program of choice when it comes to resolving Fluid Dynamics simulations. MaPFlow is developed and maintained by Assistant Professor Mr. George Papadakis in the Laboratory of Ship and Marine Hydrodynamics. MaPFlow was used instead of another solution as it is developed in-house, being easy to modify to our needs, and, of course, it is a stable and refined piece of software.

As technology progresses new demands and thus opportunities arise. 20 years ago there was less emphasis on optimization than there is today. Today designing the most optimal part is the main focus of engineers, as the agenda is pushed towards environmental protection by making everything more efficient and harnessing nature's renewable energy sources more effectively. CFD solvers can help us design efficient airplanes, cars, ships, and many more. However, CFD solvers have some significant deficiencies. First and foremost, they require large amounts of computational power to resolve even a simple simulation with acceptable accuracy. This problem is handled by either using more powerful and efficient Computational Systems or simply waiting for some inefficient machinery to resolve a simulation, consuming considerable amounts of energy in the meanwhile. However, even if we push current computational systems to their limits, the need to execute complicated simulations or optimization algorithms can be time and energy-consuming. Advances in software development and quantum computing may alleviate this problem, but for the time being, it is significant. Further information on the subject are presented in Chapter 3.

1.2 Introduction to Artificial Intelligence and its implementation in Fluid Dynamics

In recent years, the technological explosion of computers created fertile soil for breakthroughs in the Artificial Intelligence domain. Artificial Intelligence is not an old idea, however. Even in ancient Greece, many people conceptualized Artificial Intelligence and "materialized" it through magical beings. For example, the ancient greek poet Hesiodos wrote in his work about Talos, an enormous, brass human-like robot that was intelligent and was created by the god Hephaestus to guard the island of Crete. As the centuries passed the interest in artificial intelligence did not fade away, with many scientists searching the synthesis of human intelligence and how a machine would be made to embrace it. One of the most famous works in this field was the work of English mathematician Alan Turing during the 1950s. Turing set the foundations of computer science as we know it today and created a test where one shall determine if they have a conversation with a machine or not; the famous Turing test [13].

Nowadays, the immense progress of electronic computers has enabled researchers to bring forth the AI concepts of the past and create a mini-revolution. AI finds applications everywhere in our everyday lives, like smart home appliances, object recognition via camera feed, etc. These applications are mainly focused on easing our everyday lives, but the underlying framework is robust and can be applied to other applications as well. Many researchers found interest in using AI, and more specifically Deep Learning algorithms, to solve many fluid dynamics problems. Searching in the available bibliography one can see many contributions to the implementation of Deep Learning in Fluid Mechanics, targeting almost the entirety of Fluid Mechanics and Computational Fluid Mechanics. For example, there are attempts to augment turbulent models with experimental data-driven models, like the work of Wu et al. [14], Cruz et al. [15] used Neural Networks to improve the accuracy of RANS simulations. Xiaowei et al. [3] used Convolutional Neural Networks to predict the flow field around a cylinder under various Reynolds numbers. Zhou et al. [16] used Bayesian neural Networks to create an ice detecting system that could rapidly calculate the probability of ice formation. Also, many researchers investigated the use of Neural Networks to calculate the flow field or the pressure distribution around airfoils. Some works regarding that field are, with no particular order:

1. X. Hui et al. [1] use of Convolutional Neural Networks along with SDF formatted images to predict the pressure distribution around an airfoil.

2. H. Chen et al. [2] use Convolutional Neural Networks to predict the lift coefficient. Image was properly formatted to incorporate angle of attack numbers, by altering the image shading accordingly.

3. Y. Zhang et al. [17] application of Convolutional Neural Networks to predict lift coefficient under various Mach and angles of attack but approached differently from H. Chen [2].

4. V. Sekar [18] use a Neural Network with mixed architecture, both Convolutional and Fully Connected, to predict the flow field for various low Reynolds numbers, and varying angles of attack, targeting NACA symmetrical airfoils.

As it becomes apparent, there is a growing interest in studying how Artificial Intelligence can be utilized in the field of Fluid Dynamics. For the purpose of this thesis, we will focus on the work of Hui et al. [1] as it has some aspects that make it very interesting. *For the time being, we will explore whether their proposed solution holds any benefits over classic CFD solvers, regarding speed and precision.*

The main points of interest would be their use of Convolutional Neural Networks and the SDF formatted image. Convolutional neural networks are commonly used for image classification, and especially in computer vision. A very well-known application is algorithms that can tell apart animals from objects or humans, like in the CIFAR-10 and CIFAR-100 bibliography [19]. So it is very compelling to study how these algorithms, which are used mainly for something completely irrelevant to the task at hand, perform at it. Also, they use SDF formatted images to represent the airfoil's geometry. Signed Distance Function (or SDF for short) can be used to better represent a geometry under low image resolution (Further proof on how SDF works will be presented in Chapter 4). Using SDF, we can both limit the data size and simultaneously increase depiction accuracy, thus enabling our algorithm to perceive even more complicated geometrical features.

However, the main issue they address and is quite fascinating to study is their choice of airfoil design and Mach number. They use an RAE-2822 airfoil, and its randomly generated variants, that operate under transonic flight, specifically in Mach = 0.734, with a relatively small angle of attack. These conditions result in the development of one or more sonic waves along its suction side. As mentioned earlier, sonic waves formation is an extremely non-linear phenomenon that introduces enormous complexity to the solution and can even jeopardize a CFD solver's arithmetical stability. [4]

First and foremost, to train the Neural Network there needs to be created a lot of images data. For this purpose, we will use a random uniform deformation approach to create random airfoil variants that will be used to create lots of pressure distribution data. To have a sufficient training and validation pool there would be created 1500 airfoils, where 1000 are used for training, and the other 500 for testing the algorithm's accuracy to "blind" data. Testing to "blind" data is done to find whether the algorithm overfitted to training data, and therefore is inefficient against airfoils that did not belong to them. Here we will apply a uniform deformation of each face with a random and different, for each face, coefficient. These deformed airfoil variants will be depicted as SDF formatted images sized 32x32 pixels.

Then, using MaPFlow their Pressure distributions will be calculated. As we will tackle the problem in the normalized domain, where every dimension is normalized against the foil's length, it is better to calculate their Pressure Coefficient distribution instead of raw Pressure. Finally, all data will be formatted appropriately and will be used to train the neural network and test its accuracy to unknown data.

Having proven that our Neural Networks can accurately predict the Cp distributions of the training-testing variants' pool we will do some experimentation on the Networks. Namely, we will check the influence of minibatch size in training-testing precision and speed, and we will search for the Networks' precision boundaries by introducing them airfoils completely out of their original training and testing spectra.

Finally, a small application will be presented, where we will explore how the Machine Learning technology can be implemented in the design optimization procedure. This is done to showcase how we can potentially utilize Neural Networks to speed-up and ease the optimization of airfoils for use in Aerospace, Mechanical, and Marine Engineering.

[4]State-of-the-art solvers, with a properly built mesh, can rarely face such problems. However, it is still possible to happen on a more simplistic solver, like one developed for a School assignment.

1.3 Thesis structure

This thesis is structured in a simple hierarchy of first presenting the concepts that would be later used. Initially in Chapter 2, we will lay the required Fluid Mechanics theoretical background, and present how MaPFlow works and the mechanics of building a successful computational grid. In Chapter 3 we will present information about Artificial Intelligence and its components that are utilized in this work.

Subsequently, in Chapter 4 we will show how the Artificial Intelligence concepts presented in Chapter 2 are bound with the physical problem, presenting the proposed AI framework's structure and properties. Also, would specify the inner workings of SDF images, the reasoning behind Mesh selection and CFD simulation iterations, the software used and developed, and finally the project architecture.

In Chapter 5, the Neural Network results will be presented along with the results of minor experiments conducted to test the training process dependance in the mini-batch size and the Networks' ability to extrapolate and calculate Cp-distributions for airfoils out of the original training-testing range.

Chapter 6 contains information about a simple application of the trained Neural Networks in a geometry optimizer. The thesis will conclude with Chapter 7 where this work's conclusions are commented and some incentives for future work are proposed.

Chapter 2

Physical Problem

2.1 Flow around an airfoil

As mentioned in the introductory chapter 1, we are interested in studying the application of Neural Networks in the field of Fluid Dynamics. To be specific, we want to use Neural Networks to predict the Cp distribution around an airfoil under subsonic flow close to air's Mach Number. Under these circumstances, the flow around the airfoil is compressible and develops shock waves. Shock waves are non-linear phenomena that introduce complexity to the solution of the flow field. Not only are they expected to trouble the CFD solver but the Neural Networks, as well. It is important to note that the formation of the boundary layer imposes great difficulty in solving the flow problem. Also, modeling the flow close to the boundary is always a difficult task that if is not accounted properly, would probably lead to a poor solution of the flow field.

2.1.1 Airfoil operation basics

To avoid confusion about the boundary layer and shock waves, an introduction in airfoil operation mechanics is needed.

Airfoils operate on a very simple principle; they have two sides that operate on the different pressure distributions to generate a net force in the desired direction. Observing the figure 2.1, we can notice that airfoils operate in a way that enables them to displace the fluid flow in a direction opposite to the direction they want to produce force. Then by the Newton's 3^{rd} Law of Action-Reaction they produce a force with magnitude analogous to the mass of fluid they displace and a direction opposite to the displacement force.

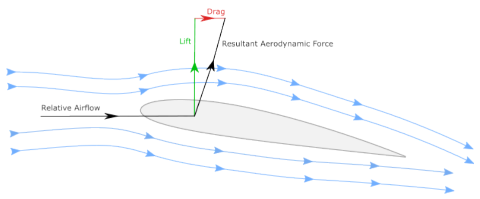


Figure 2.1: Forces on an airfoil [20]

Another way to look at it is using the Conservation of Mechanical Energy assuming the flow is inviscid and apply the Bernoulli's principle (eq.: 1.6). Observing again figure 2.1, we can notice that the airfoil constraints the flow's section on its top side and widens it on the bottom side. By the conservation of mass, as there is no mass transfer during the flow, we get:

$$A_1 \cdot u_1 = A_2 \cdot u_2 \Rightarrow u_2 = \frac{A_1}{A_2} u_1 \tag{2.1}$$

So if :

$$A_1 < A_2 \Rightarrow u_2 < u_1 \tag{2.2}$$

$$A_1 > A_2 \Rightarrow u_2 > u_1 \tag{2.3}$$

Using Bernoulli's principle we get that where the velocity is increased the pressure is decreased and vice versa. So, the bottom side will experience a increase in static pressure due to its flow deceleration and the top side will experience a pressure loss due to its flow acceleration.

Integrating the pressure distribution on each face we get the force applied by the air to each face due to Newton's 3^{rd} Law. Then by adding the two forces together, the net force is obtained. As seen in figure: 2.1, the net force is in most cases not normal to the Free-stream Velocity direction of air. Analyzing the net force into a normal and a perpendicular to the Free-stream Velocity direction of air components, we get two forces named Lift and Drag respectively. Lift is the useful component that we want to maximize and Drag is the component we want to minimize. As we want airfoils to convert fluid flow to a normal force with respect to some direction is only normal to want to maximize the normal component. Yet, Drag is something we cannot truly get rid of as it is heavily connected with the airfoil's capability to produce Lift, but we want to keep it minimal in all cases. For the time being it should be understood that Drag is produced due to the airfoil's geometry and has nothing to do with friction.

Having explored the basic mechanics of airfoils, some things become apparent. First, we want our airfoil to always produce as much Lift as possible with as less Drag as possible. So, designing a very optimized geometry is very important to the application. For example, if the need is to have unidirectional lift production (ship or airplane rudders), we may consider designing a symmetrical foil where we want a smooth and optimal geometry on both sides. On the contrary, if the application dictates generating normal force constantly in a single direction, we may want to develop an airfoil optimized for creating the maximum normal force under very specific angles of attack[1] spectrum.

Also, we notice that operating a foil on higher speeds can lead to further pressure difference between the two sides. So, a simplistic line of thought would be to increment the velocity to produce more Lift. However, this is not always the case. Different fluids' properties creates various problem to this simplistic approximation. Viscid fluids, like water, experience immense increase in Friction making it impossible to accelerate them beyond a certain point. Inviscid fluids [2], like air, do not experience this sudden increase in friction, yet they are compressible and that complicates things even more.

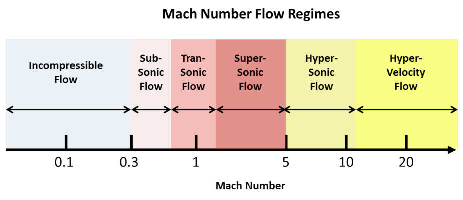
Air having low viscosity means we can accelerate an airfoil to enormous speeds, ranging from close to air's speed of sound in commercial airlines to hypersonic flight where the speed is many times greater than the air's speed of sound. To get a better metric of the flow's speed with respect to the air's speed of sound, we use the Mach number:

$$Mach = \frac{u_{airfoil}}{\alpha} \tag{2.4}$$

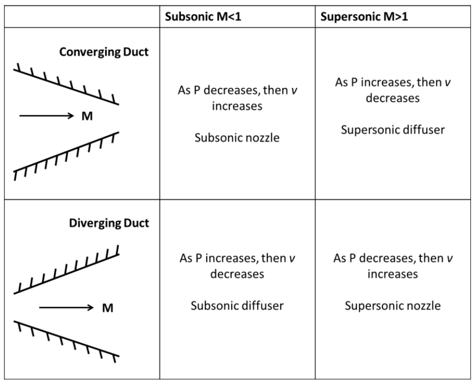
,where α is the local speed of sound. $Mach = 1$ is obtained when the $u_{airfoil} = \alpha$. As the Mach number increases the compressibility of air is getting more and more significant, affecting significantly the flow. Also, when the flow is hypersonic due to compressibility something paradoxical happens that contradicts the incompressible inviscid Bernoulli's principle; the flow is accelerated when the section area increases and decelerates when the section area decreases. This does not contradict the Conservation of Mass, as in equation 2.1 we treated air as incompressible or operating far from the sonic state where the compressibility is negligible and density ρ can be considered constant without too much error. Even when approximating $Mach \leq 1$ the flow is expected to behave like for $Mach << 1$ considering we account for the density change. However, for $Mach \geq 1$ the flow behaves differently because of it being compressible.

[1]Angle of Attack is the angle of the airfoil's chord relative to the Free-stream Velocity direction of air

[2]All commonly encountered fluids are viscid but in some cases the viscosity is almost negligible, like in atmospheric air and other gases. However, in water, oil, etc. viscosity is very important and is took into consideration when solving their flow field.



(a) Mach Number Spectrum [21]



	Subsonic M<1	Supersonic M>1
Converging Duct	As P decreases, then *v* increases Subsonic nozzle	As P increases, then *v* decreases Supersonic diffuser
Diverging Duct	As P increases, then *v* decreases Subsonic diffuser	As P decreases, then *v* increases Supersonic nozzle

(b) Nozzle and Diffuser behavior fo Subsonic and Supersonic flows [21]

Figure 2.2: Different flow characteristics with respect to $Mach$ number

So in figure : 2.1 the flow instead of slowing down near the top face's trailing edge, as it happens during $Mach << 1$, accelerates even more during hypersonic operation. However, for this thesis we are not particularly interested in pure hypersonic flow, rather transonic flow. Transonic flow happens when $Mach \in \approx [0.7, 1.3]$ as seen is figure 2.2a. The interest with transonic flows is that they are commonly used for aviation, gas transfer, steam turbines, etc., and revolve around the fine balance between alternating Mach Number in the vicinity of $Mach = 1$, developing shock waves, minimizing heat transfer between the gas and its environment, and other interesting phenomena.

In this thesis, we will study the RAE-2822 operating at $Mach = 0.734$, an angle of attack $a_a = 2.73^o$ and free-stream Reynolds number $Re = 6.5 \cdot 10^6$. Under this circumstances the flow is expected to briefly enter hypersonic state around $\approx 60\%$ of chord length from the leading edge. Due to reasons we will shortly explain, around this area is expected the formation of a shock wave and of course as the Reynolds number is high enough, a complicated Boundary Layer would form that would increase the complexity of calculating the pressure distribution around the airfoil.

Finally, we will calculate the Pressure distribution and subsequently Lift and Drag as non-dimensional constants. This is a standard practice in the field of Fluid Mechanics, where we prefer to solve the flow field in a normalized space and extract non-dimensional coefficients which in turn can be accordingly scaled to a specific case using Buckingham's Π Theorem.

Buckingham's Π Theorem indicates that validity of the laws of physics does not depend on a specific unit system. A statement of this theorem is that any physical law can be expressed as an identity involving only dimensionless combinations (ratios or products) of the variables linked by the law (for example, pressure and volume are linked by Boyle's law – they are inversely proportional). If the dimensionless combinations' values changed with the systems of units, then the equation would not be an identity, and Buckingham's theorem would not hold [22].

2.1.2 Boundary Layer

As already mentioned in section 1, to calculate to flow field around the airfoil we need to solve the Navier–Stokes equations. A partial differential equation, like Navier-Stokes, requires the input of proper boundary conditions to account for the spatial boundaries of the control volume and initial conditions to set the conditions for time $t = 0$. The boundary conditions for an airfoil flow problem are:

1. $U_\infty = cons$ far-away from the airfoil body (Undisturbed flow)
2. $\vec{u} = \vec{u_w}$ on the solid boundary (Non-slip boundary condition), where $\vec{u}$ is the fluid's velocity vector and $\vec{u_w}$ the solid boundary's velocity field

In the previous section, we analyzed how the air flow behaves far from the solid boundary, yet to solve the Navier–Stokes equations we also need to study the proximity of the solid boundary.

In order to apply the solid boundary's boundary condition we have to account for the fluid's viscosity. The already mentioned non-slip condition works only under the premise of a viscous flow, otherwise it is degraded to the non-intrusion boundary condition of $\frac{\partial \vec{u}}{\partial \vec{\eta}} = 0$, where the $\vec{\eta}$ is the boundary geometry's normal vector. This is done as in non-viscid flows the cohesion between the fluid's molecules is negligible and no shear forces are developed to restrict the fluid's motion. However, the molecules cannot penetrate the solid boundary.

In a viscous flow, the molecules' cohesion is not negligible and shear forces are developed to restrict the molecules relative motion. Viscosity is the measure of the molecules' cohesion. As we have a fluid that restricts the relative motion between two of its adjacent layers by developing shear forces, a new challenge emerges. Solving the problem in the far-field where the fluid's velocity is either constant or has smooth, gradual gradients we can almost treat the flow as inviscid without great error. Yet, near the solid boundary where the velocity gradient is steep we can not neglect the fluid's shear forces.

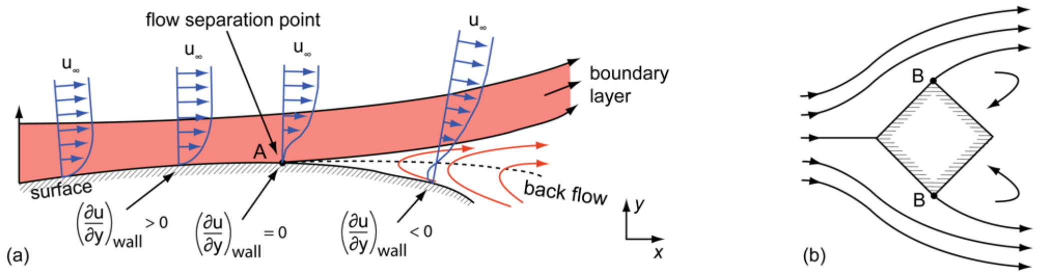

flow separation point
u_∞
boundary layer
A
surface
back flow
$\left(\frac{\partial u}{\partial y}\right)_{wall} > 0$
$\left(\frac{\partial u}{\partial y}\right)_{wall} = 0$
$\left(\frac{\partial u}{\partial y}\right)_{wall} < 0$
y
x
(a)
B
B
(b)

Figure 2.3: Flow separation over an airfoil [23]

Assuming a high Reynolds number ($Re >> 1$) over elongated bodies, we define the boundary layer as a thin area around the solid boundary where the cohesive phenomena are very intense, and steep gradients of the velocity distributions exist. Inside the area apply specially modified Navier–Stokes equation that take into consideration the viscous stresses [6]. Introducing the boundary layer we can split the problem in two pieces; one regarding the far-field and one for the solid boundary, keeping into consideration that the two fields must match in terms of $\vec{u}$, p in the boundary of the two fields.

As seen in figure : 1.4 the boundary layer constitutes of three somewhat discrete states. First, for local Reynolds number $Re_x < Re_{critical}$ the flow is laminar, having a steeper velocity distribution gradient and accelerating faster to the local average[3] velocity U. When the Re_x is in the vicinity of the $Re_{critical}$ and specifically $Re_x \geq Re_{critical}$, the flow transitions from laminar to turbulent. And finally for $Re_x >> Re_{critical}$ the flow is turbulent. The $Re_{critical}$ is a property that is heavily depended on the geometry of the solid boundary. For example, a flow inside a pipe has a $Re_{critical} = 2300$ and a flow over a flat plate has a $Re_{critical} = 3.2 \cdot 10^5 \div 10^6$. Of course these values are not absolute as the surface smoothness plays a detrimental role in the development of turbulent flow, but these numbers are under the assumption of a perfectly smooth surface, an assumption that we will also make for the current analysis.

As the Freestream Reynolds number of our scenarios is $Re_F = 6.5 \cdot 10^6$ it is apparent that the boundary layer will be mostly turbulent. The turbulent flow is characterized by consisting of random vortices that transfer momentum between the different layers of the fluid, thus creating smoother velocity distribution gradients in expense of losing average speed in the boundary layer. However, this is not always a bad thing as the flow slows down and

[3]In laminar flow the velocity is uniform and the average is redundant. However, in turbulent flows is better to separate the velocity into a average ‘deterministic’ component and a random noise component induced by vorticity.

the separation point is moved downstream, reducing the ‘dead area’ of the flow [6]. This is commonly utilized in golf balls where the pits on their surface induce the turbulent flow, propelling the ball further with the same amount of energy. This can also be useful when designing airfoils as having less separation can help with gaining more Lift, by extending the area affected by the fluid’s attached flow.

Flow separation is a phenomenon that happens naturally due to flow mechanics that are irrelevant to having or not a turbulent flow. Considering the flow over an airfoil, we already analyzed in the previous section how the flow accelerates and decelerates over the two faces. As the top side of the foil can be perceived as a nozzle and a diffuser in series, we can understand that the flow is decelerated and the pressure is increased towards the trailing edge. That increase in pressure the flow faces downstream causes the layer of fluid closer to the boundary that has low kinetic energy to change direction and move in the opposite direction [6], as seen in figure 2.3.

Separated flows can reattach to the foil’s body, forming defined, closed vortices on the solid boundary, yet they can also never reattach creating large vortices that are dampened due to viscous forces far from the body. In either case, there is a loss of net force due to the fluid’s recirculation. The separation is a phenomenon that occurs both for laminar and turbulent boundary layers as it is more connected to the geometry of the boundary than the Reynolds number of the flow.

To conclude, the formation of a boundary layer is a by-product of the fluid’s viscosity and an inherent property of fluids. Separation is also a phenomenon closely related with the basic mechanics of a fluid flow that is in most cases inevitable. Namely, even if we design the most aerodynamic foil in existence for a specific α, there are only a specific range of angles of attack where no separation would form. So, when designing an airfoil we have to take into consideration these phenomena and design the desired foil by respecting and utilizing our knowledge of boundary layers and separation.

However, even if we have some adequate knowledge around separation and boundary layers we are still not able to find an analytical solution for them. For laminar flow things are considerably easier due to the flow’s excellent mathematical modelling [4]. Nevertheless, turbulent boundary layers and separation in a turbulent context are extremely hard to solve analytically, and are solved analytically by using approximations. So, we resort to highly sophisticated models in order to get a glimpse inside turbulent boundary lay-

[4]Even-though we can analytically solve some special cases, there is no analytical solution for every case as solving the boundary layer flow field utilizes high order ODEs. Yet, we can numerically solve them with relative ease.

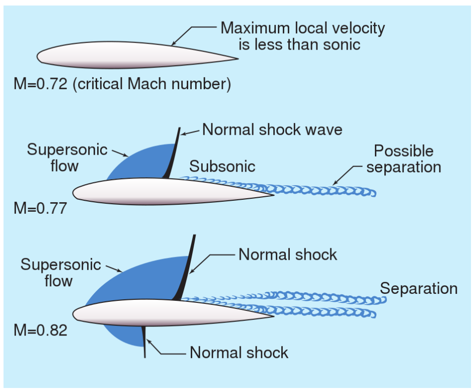


Figure 2.4: The formation of a shock wave during subsonic flight [24]

ers and separation by approximating the turbulent flow. Consequently, we get somewhat accurate results only after consuming a considerable energy amount for the CFD solvers to resolve these models.

2.1.3 Shock Waves

To understand shock waves we have to comprehend how a simple one - dimensional compressible flow works. The simplest flow to analyse is the flow of a compressible fluid through a Laval nozzle (fig.: 2.5). In order to proceed, we have to assume that the flow through the nozzle is isentropic, ie. reversible. This assumption requires that the flow is inviscid and no heat is transferred outside of the control volume. This assumption is done to simplify things a bit and not resort to complicated equations that are out of this thesis scope. However, this assumption is not far from reality as it is a special case of the Rayleigh and Fanno flows for compressible liquids [25].

Considering the pressure difference is sufficient, a hypersonic flow may develop and sustain itself. However, if the pressure difference is not sufficient for a hypersonic flow to develop, but is sufficient to reach sonic flow at the nozzle's neck, the flow will become shortly hypersonic and then will collapse into subsonic. This transition happens almost instantly both with respect to time and space. This phenomenon is called a normal shock wave which is a special case of a shock wave. During a shock wave, the flow is chocked, meaning that the pressure is increased and subsequently the velocity is decreased in a non isentropic way, thus there is no knowledge on what happens in the shock wave only what happens before and after it.

Shock waves also emerge when studying any kind of flow that its velocity is close to a fluid's speed of sound ($Mach = 1$). Studying any kind of airfoil or airplane that operates close to $Mach = 1$ will develop shock waves (see

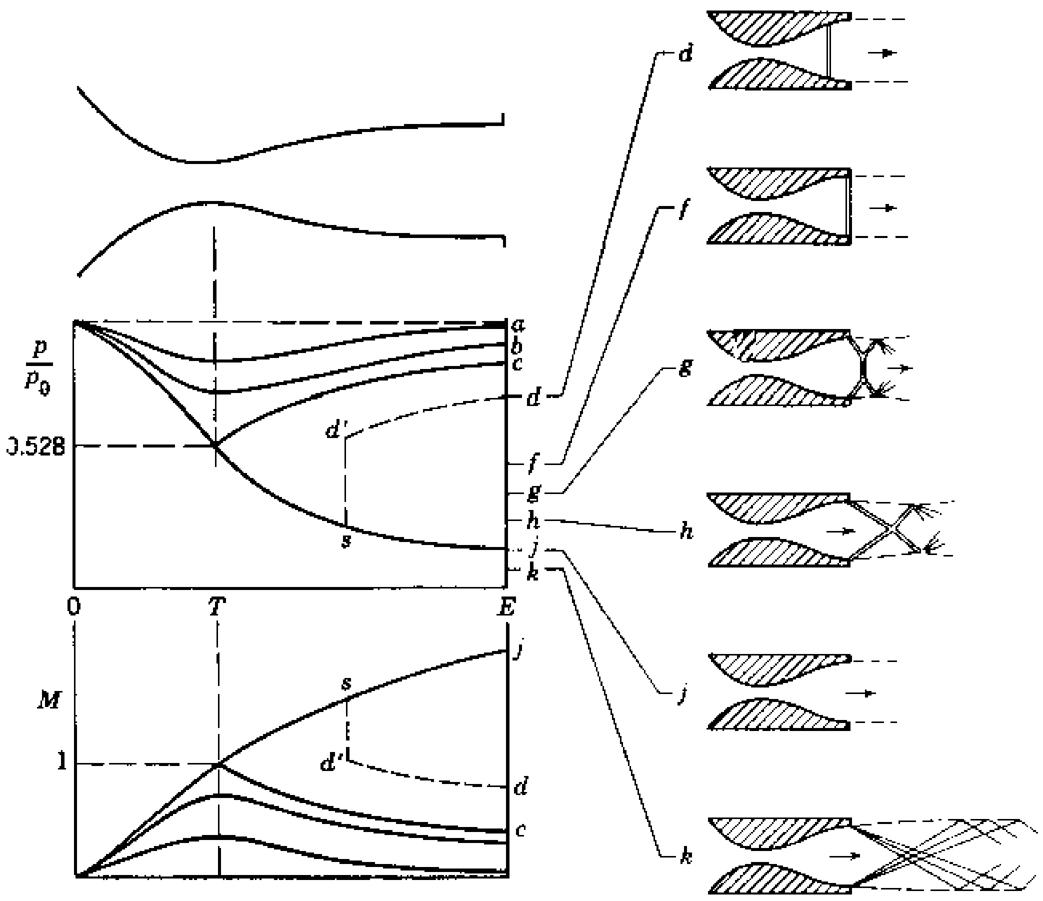


Figure 2.5: The Laval nozzle and different Pressure differences [26]
Denoting the fluid's properties in the inlet of the nozzle with o and the fluid's properties in the outlet with e we can observe how can the pressure difference between inlet and outlet can affect the fluid's velocity downstream. In this figure, we see the geometrical characteristics of the Laval nozzle, as well, how it behaves under different $\frac{p_o}{p_e}$. Along with, the diagrams showing the pressure ratios and local Mach number with respect to x (distance inside the nozzle), Liepmann and Rosko have placed sketches that show how the flow develops what shock waves manifest. For every $p_e \in (p_c, p_j)$, the pressure ratio is not sufficient to develop and maintain hypersonic flow, thus leading to the creation of shock waves. The shock waves have different geometries depending on the pressure ratio $\frac{p_o}{p_e}$, with them being normal for $p_e \leq p_c$ and $|p_e - p_c| \leq |p_e - p_j|$, and then becoming gradually more angular as $p_e \to p_j$. For $p_e = p_j$ the flow is hypersonic and for $p_e > p_j$ it develops expansion waves.

figure : 1.5). However, the normal shock waves introduced earlier are only a subset of the the shock waves. Shock wave's angle of formation depends not only in the velocity of the fluid but the boundary's geometry, as well. For example, for our RAE-2822 that operates in a transonic flow it is expected to develop at least one oblique shock wave on the low pressure side due to the flow locally reaching hypersonic status, as already mentioned earlier.

As mentioned earlier, for a converging duct if $Mach < 1$ then the flow is accelerated and for diverging duct if $Mach > 1$ the flow is again accelerated further. So, considering our airfoil operates under a sufficient Mach number that is less than 1 it can still locally develop hypersonic flow. This is something that happens inevitably due to the foil's geometry. As the hypersonic mode cannot be maintained because of the operation of the airfoil and the static pressure of the surrounding air, the flow develops a shock wave and drops to a subsonic state.

As the flow exists in the three-dimensional space, [5] no normal shock waves are produced, instead the flow generates oblique shock waves. Oblique shock waves are three dimensional phenomena that manifest for a three dimensional flow over three dimensional object, like an airfoil (see figure : 1.5). Nevertheless, we simplify things by studying them only on the two axis of the airfoil's cross section.

Shock waves are pretty interesting as they disturb the fluid's flow and insert non-linearities in the pressure distribution. Observing the Cp distribution diagram for a RAE-2822, figure : 2.6, we can immediately notice the effects of the shock wave on the pressure distribution. At around 50 % of the chord's the shock wave manifests, as the Cp coefficient suddenly exponentially increases from ≈ -1.2 to ≈ -0.5. Then as the flow has decelerated into a subsonic flow, the Cp is further increased as we approach the trailing edge. We expected that Cp would definitely increase in the top side, principally due to flow separation, but the shock wave deprived us from even more net area to develop Lift. Even-though, shock waves are inevitable along with flow separation, understanding them leads to better designed airfoil's that can perform under this circumstances with better efficiency.

Also, it should be noted that the formation of shock waves can help in the formation of flow separation. As already analyzed, downstream of the shock wave the pressure spikes and the formation of separation is inevitable. In figure : 2.4, we can see a schematic of this. So, when calculating the effect of the shock wave, we must also account for its effect on the boundary layer. As it can be easily understood, this makes calculations even more complicated

[5] Even-though our calculations do not take into consideration the airfoil's third axis due to having a uniform geometry along it

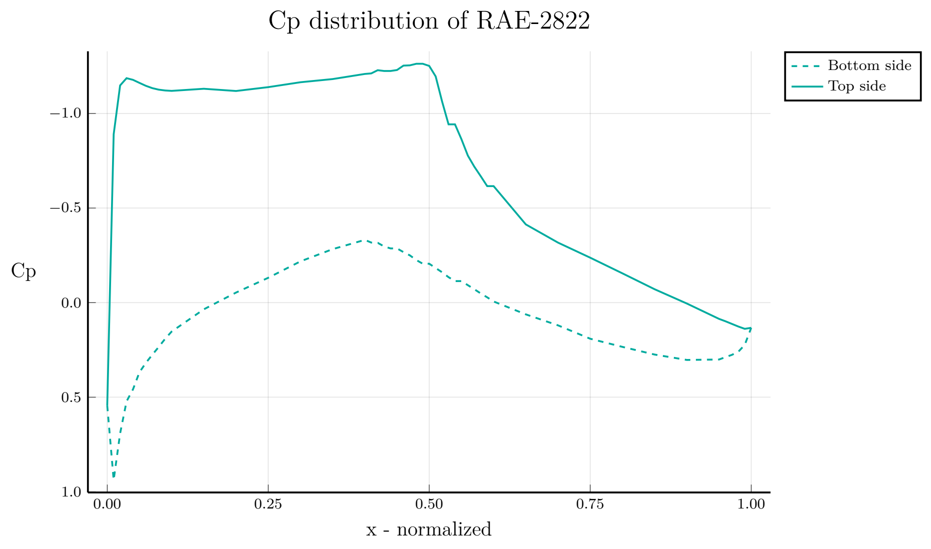


Figure 2.6: Cp distribution of a RAE-2822 airfoil

The Cp distribution of the above figure is based on experiments carried by the Berlin TU to validate various turbulence models in order to use them with their CFD solver [27]

and adds to the computational cost of CFD simulations.

2.2 MaPFlow internals

Calculating the flow around the airfoil is an intimidating task to address by analytically solving the Navier-Stokes equations. All the phenomena already discussed in the previous section impose extreme difficulty in doing so. However, using numerical analysis the Fluid Dynamics problem can be transferred from the continuous time-space to a discrete time-space. Then using various techniques of discretizing space to simple geometric entities we can approximate the solution on this discrete space and apply the results to the continuous time-space. This task can be manually done by utilizing complex analysis [6] for low Mach numbers and special algorithms designed to solve Navier-Stokes equations for everything more elaborate.

MaPFlow is a Computational Fluid Dynamics solver, developed by Mr. Papadakis [28]. MaPFlow is an Eulerian CFD solver that can solve both compressible and incompressible problems utilizing an unstructured Finite Volume computational grid to solve the Unsteady Reynolds Averaged Navier-

Stokes (*URANS*) equations.

In this segment we will present some of the concepts that make up MaPFlow. MaPFlow is developed and maintained by Mr. Papadakis, who also was kind enough to give me access to its documentation [28]. Thus the following analysis is heavily based on the dense, yet excellent analysis of MaPFlow given to me. MaPFlow has lots of components that enable it to solve any kind of fluid dynamics problem, however here are presented only the components that are useful in the present study.

2.2.1 Governing equations

Conservative form

Let D denote a volume of fluid and ∂D its boundary. By integrating the Governing equations over D, the following integral form is obtained:

$$\int_D \frac{\partial \vec{U}}{\partial t} dD + \oint_{\partial D} (\vec{F}_c dS - \vec{F}_v) dS = \int_D \vec{Q} dD \tag{2.5}$$

In (2.5) $\vec{U}$, is the vector of the Conservative Flow Variables,

$$\vec{U} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{pmatrix} \tag{2.6}$$

where ρ denotes the density, (u, v, w) the three components of the velocity field and E the total energy. $\vec{F}_c$ and $\vec{F}_v$ denote the Convective and Viscous Fluxes respectively,

$$\vec{F}_c = \begin{pmatrix} \rho V \\ \rho u V + n_x p \\ \rho v V + n_y p \\ \rho w V + n_z p \\ \rho(E + \frac{p}{\rho}) V \end{pmatrix} \tag{2.7}$$

$$\vec{F}_v = \begin{pmatrix} 0 \\ n_x \tau_x x + n_y \tau_x y + n_z \tau_x z \\ n_x \tau_y x + n_y \tau_y y + n_z \tau_y z \\ n_x \tau_z x + n_y \tau_z y + n_z \tau_z z \\ n_x \Theta_x + n_y \Theta_y + n_z \theta_z \end{pmatrix} \tag{2.8}$$

where V is the contravariant velocity, $V = \vec{u} \cdot \vec{n}$ and

$$\begin{aligned}
\Theta_x &= u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + k\frac{\partial T}{\partial x} \\
\Theta_y &= u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + k\frac{\partial T}{\partial y} \\
\Theta_z &= u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + k\frac{\partial T}{\partial z}
\end{aligned} \tag{2.9}$$

The above system is completed with the equation of state for perfect gas:

$$p = (\gamma - 1)\rho\left[E - \frac{u^2 + v^2 + w^2}{2}\right] \tag{2.10}$$

Variable Transformations

CFD solvers are formulated using the governing equations (2.5) written in primitive ($\vec{V}$) or characteristic ($\vec{V}_{ch}$) variables,

$$\vec{V} = \begin{pmatrix} \rho \\ u \\ v \\ w \\ p \end{pmatrix} \tag{2.11}$$

Primitive Variables Starting from the differential form of the governing equations,

$$\frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}_c}{\partial \vec{x}} - \frac{\partial \vec{F}_v}{\partial \vec{x}} = \frac{\partial \vec{Q}}{\partial \vec{x}} \tag{2.12}$$

and neglecting the viscous terms and using the chain rule:

$$\begin{aligned}
\frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}_c}{\partial \vec{U}}\frac{\partial \vec{U}}{\partial \vec{x}} &= \frac{\partial \vec{Q}}{\partial \vec{x}} \\
\frac{\partial \vec{U}}{\partial t} + A_c\frac{\partial \vec{U}}{\partial \vec{x}} &= \frac{\partial \vec{Q}}{\partial \vec{x}}
\end{aligned} \tag{2.13}$$

the Jacobian of the convective fluxes $A_c = \partial \vec{F}_c / \partial \vec{U}$ is obtained. By introducing the transformation matrix $M = \partial \vec{U} / \partial \vec{V}$, the system of equations is written in Primitive Variables form [29]:

$$\begin{aligned}
&\frac{\partial \vec{U}}{\partial \vec{V}}\frac{\partial \vec{V}}{\partial t} + A_c \frac{\partial \vec{U}}{\partial \vec{V}}\frac{\partial \vec{V}}{\partial t} = \frac{\partial \vec{Q}}{\partial \vec{x}} \\
&M\frac{\partial \vec{V}}{\partial t} + A_c M\frac{\partial \vec{V}}{\partial t} = \frac{\partial \vec{Q}}{\partial \vec{x}} \\
&\frac{\partial \vec{V}}{\partial t} + M^{-1}A_c M\frac{\partial \vec{V}}{\partial t} = M^{-1}\frac{\partial \vec{Q}}{\partial \vec{x}} \\
&\frac{\partial \vec{V}}{\partial t} + A_p\frac{\partial \vec{V}}{\partial t} = \frac{\partial \vec{Q}_v}{\partial \vec{x}}
\end{aligned} \tag{2.14}$$

The transformation matrix M is defined as:

$$M = \frac{\partial \vec{U}}{\partial \vec{V}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ u & \rho & 0 & 0 & 0 \\ v & 0 & \rho & 0 & 0 \\ w & 0 & 0 & \rho & 0 \\ \frac{u^2+v^2+w^2}{2} & \rho u & \rho v & \rho w & \frac{1}{\gamma-1} \end{bmatrix} \tag{2.15}$$

and its inverse as:

$$M^{-1} = \frac{\partial \vec{V}}{\partial \vec{U}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{u}{\rho} & \frac{1}{\rho} & 0 & 0 & 0 \\ -\frac{v}{\rho} & 0 & \frac{1}{\rho} & 0 & 0 \\ -\frac{w}{\rho} & 0 & 0 & \frac{1}{\rho} & 0 \\ \frac{\gamma-1}{2}(u^2+v^2+w^2) & -u(\gamma-1) & -v(\gamma-1) & -w(\gamma-1) & \gamma-1 \end{bmatrix} \tag{2.16}$$

Equation (2.14) has the same form as (2.13) but the convective flux Jacobian A_p is in primitive variables.

Characteristic Variables Diagonalization of A_c ($A_c = R\Lambda L$, where R,L contain the right and left eigenvectors respectively and Λ the eigenvalues) enables to transform (2.13) in characteristic variables and by that to decouple the system of equations.

$$\begin{aligned}
&\frac{\partial \vec{U}}{\partial t} + L^{-1}\Lambda L\frac{\partial \vec{U}}{\partial \vec{x}} = \frac{\partial \vec{Q}}{\partial \vec{x}} \\
&L\frac{\partial \vec{U}}{\partial t} + \Lambda L\frac{\partial \vec{U}}{\partial \vec{x}} = L\frac{\partial \vec{Q}}{\partial \vec{x}}
\end{aligned} \tag{2.17}$$

By defining $M_{ch} \equiv \partial\vec{U}/\partial\vec{V}_{ch} = L$ the decoupled system is acquired:

$$\frac{\partial\vec{V}_{ch}}{\partial t} + \Lambda\frac{\partial\vec{V}_{ch}}{\partial\vec{x}} = \frac{\partial\vec{Q}_{ch}}{\partial\vec{x}} \tag{2.18}$$

with:

$$\Lambda = \begin{bmatrix} V & 0 & 0 & 0 & 0 \\ 0 & V & 0 & 0 & 0 \\ 0 & 0 & V & 0 & 0 \\ 0 & 0 & 0 & V+c & 0 \\ 0 & 0 & 0 & 0 & V-c \end{bmatrix} \tag{2.19}$$

Depending on the variables chosen the diagonalization of the Jacobian Matrix (A_c for conservative variables,A_p for primitive) will lead to different eigenvectors. Of course, using the appropriate transformation matrix the eigenvectors can be transformed to the variables desired. For example in between primitive and characteristic variables the following hold:

$$A_p = M^{-1}A_cM = \left(M^{-1}R\right)\Lambda\left(LM\right) = R_p\Lambda L_p \tag{2.20}$$

where $R_p = M^{-1}R$ and $L_p = LM$ are the right and left eigenvectors in primitive variables. Using the inverse transformation matrix enables the transformation from primitive eigenvectors to conservative eigenvectors.

The right eigenvectors in primitive variables are :

$$R_p = \begin{bmatrix} n_x & 0 & n_z & -n_y & -\frac{n_x}{c^2} \\ n_y & -n_z & 0 & n_x & -\frac{n_y}{c^2} \\ n_z & n_y & -n_x & 0 & -\frac{n_z}{c^2} \\ 0 & n_x & n_y & n_z & -\frac{\lambda_1-\lambda_4}{\rho c^2} \\ 0 & -n_x & -n_y & -n_z & \frac{\lambda_1-\lambda_5}{\rho c^2} \end{bmatrix} \tag{2.21}$$

where $\vec{n} = (n_x, n_y, n_z)$ is the unit normal vector, $\lambda_1 = \lambda_2 = \lambda_3 = V$, $\lambda_4 = V + c$ and $\lambda_5 = V - c$.

The left eigenvectors in primitive variables are :

$$L_p = R_p^{-1} = \begin{bmatrix} n_x & n_y & n_z & \frac{\rho}{\lambda_4-\lambda_5} & \frac{\rho}{\lambda_4-\lambda_5} \\ 0 & -n_z & n_y & \frac{\lambda_1-\lambda_5}{\lambda_4-\lambda_5}n_x & \frac{\lambda_1-\lambda_4}{\lambda_4-\lambda_5}n_x \\ n_z & 0 & -n_x & \frac{\lambda_1-\lambda_5}{\lambda_4-\lambda_5}n_y & \frac{\lambda_1-\lambda_4}{\lambda_4-\lambda_5}n_y \\ -n_y & n_x & 0 & \frac{\lambda_1-\lambda_5}{\lambda_4-\lambda_5}n_z & \frac{\lambda_1-\lambda_4}{\lambda_4-\lambda_5}n_z \\ 0 & 0 & 0 & \frac{\rho c^2}{\lambda_4-\lambda_5} & \frac{\rho c^2}{\lambda_4-\lambda_5} \end{bmatrix} \tag{2.22}$$

2.2.2 Spatial Discretization

In MaPFlow the flow variables are calculated and stored at cell centers. Assuming that the cell volume remains unchanged:

$$\frac{\partial}{\partial t}\int_D \vec{U} dD = D\frac{\partial \vec{U}}{\partial t} \tag{2.23}$$

where:

$$\vec{U} = \frac{1}{D}\int_D \vec{U}_{exact} dD \tag{2.24}$$

Thus equation (2.5) becomes:

$$\frac{\partial \vec{U}}{\partial t} = -\frac{1}{D}[\oint_{\partial D}(\vec{F}_c - \vec{F}_v)dS - \int_D \vec{Q}dD] \tag{2.25}$$

The surface integral is approximated using piecewise constant fluxes over the cell faces that are calculated at their centers. For cell I,

$$\frac{d\vec{U}_I}{dt} = -\frac{1}{D_I}\underbrace{[\sum_{m=1}^{N_f}(\vec{F}_c - \vec{F}_v)_m \Delta S_m) - (\vec{Q}D)_I]}_{R_I} = -\frac{1}{D_I}\vec{R}_I \tag{2.26}$$

where N_f is the number of faces the cell has and ΔS_m is the area of face "m". The terms $(\vec{F}_c)_m, (\vec{F}_v)_m$ represent the convective and viscous fluxes through face m.

Reconstruction of variables In order to calculate the fluxes appearing in the right hand side of (2.26), the values of all flow variables at the face centers are needed. This information is absent, since all flow variables are defined at the cell centers. Passing the flow information from the cell centers to the faces is carried out by means of *variable reconstruction.*

Consider two cells *I,J* being in contact over face *f*. Variable reconstruction on *f* can be defined either starting from cell *I* or cell *J*. For compressible solvers it is assumed that across the face the flow experiences a jump defined by the left *L* and right *R* states. The L/R specification depends on the normal to *f* which directs from L to R.

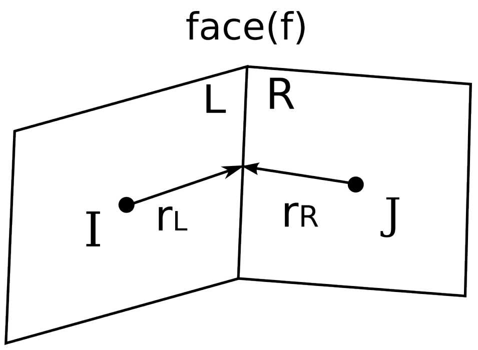


Figure 2.7: Reconstruction of variables on a face (f).

In MaPFlow, the Piecewise Linear Reconstruction (PLR) is used which is formally second order on regular grids [30]. PLR approach implies that the flow variables are linearly distributed over the control volume. Thus the Left and Right reconstructed states are defined as follows:

$$\vec{V}_L = \vec{V}_I + \Psi_I(\nabla\vec{V}_I \cdot \ \vec{r}_L) \tag{2.27}$$

$$\vec{V}_R = \vec{V}_J - \Psi_J(\nabla\vec{V}_J \cdot \ \vec{r}_R) \tag{2.28}$$

where $\vec{r}_L, \vec{r}_R$ denote the distance vectors pointing from the cell centers to the face center(Fig. 2.7) and Ψ a limiter function. In the above expression, the gradients are calculated at the corresponding cell centers using the Green-Gauss formulation:

$$\nabla\vec{V} \approx \frac{1}{D}\int_{\partial D}\vec{V}\vec{n}dS \tag{2.29}$$

which in the Cell-Centered scheme takes the form:

$$\nabla\vec{V}_I \approx \frac{1}{D}\sum_{J=1}^{N_f}\frac{1}{2}(\vec{V}_I + \vec{V}_J)\vec{n}_{IJ}\Delta S_{IJ} \tag{2.30}$$

Limiters Function Ψ appearing in (2.27) and (2.28) is a limiter function that reduces the gradients $\nabla\vec{V}_I, \nabla\vec{V}_J$. Limiter functions are widely used in compressible solvers in order to ensure convergence in areas with strong gradients. In MaPFlow, the Venkatakrishnan limiter is used due to its good convergence properties [31], [32].

$$\Psi_i = min_j \begin{cases} \frac{1}{\Delta_2}\left[\frac{\Delta_{1,max}^2+\epsilon^2)\Delta_2+2\Delta_2^2\Delta_{1,max}}{\Delta_{1,max}^2+2\Delta_2^2+\Delta_{1,max}\Delta_2+\epsilon^2}\right] & \text{if } \Delta_2 > 0 \\ \frac{1}{\Delta_2}\left[\frac{\Delta_{1,min}^2+\epsilon^2)\Delta_2+2\Delta_2^2\Delta_{1,min}}{\Delta_{1,min}^2+2\Delta_2^2+\Delta_{1,min}\Delta_2+\epsilon^2}\right] & \text{if } \Delta_2 < 0 \\ 1 & \text{if } \Delta_2 = 0 \end{cases} \tag{2.31}$$

where

$$\Delta_2 = \nabla \vec{V}_i \cdot \vec{r}_i \tag{2.32}$$

$$\Delta_{1,max} = \vec{V}_{max} - \vec{V}_i \tag{2.33}$$

$$\Delta_{1,min} = \vec{V}_{min} - \vec{V}_i \tag{2.34}$$

$\vec{V}_{max}, \vec{V}_{min}$ refer,to the maximum and minimum of $\vec{V}$ of all neighboring cells. The parameter ϵ^2 defines the amount of limiting. In practice ϵ is proportional to the length scale of the grid (Δh),

$$\epsilon^2 = (K\Delta h)^3 \tag{2.35}$$

where K is a free parameter. Small values of K make the limiter strict rendering the PLR first order, while $K = \infty$ leads to an unlimited scheme. Typically the value of $K = 5$ is used.

Convective Fluxes The discretization of the convective fluxes can be based on central, flux-vector or flux-difference schemes. Central schemes calculate the convective fluxes across faces as the arithmetic average of the values obtained at the two sides of the face plus an artificial dissipation term added to enhance stability [33]. Flux-vector schemes are based on *upwinding* which respects the direction of propagation of waves [34], [35]. Finally, flux-difference schemes calculate convective fluxes at cell faces solving the Riemann problem for the Left and Right states defined on the face [36].

MaPFlow uses Roe's approximate Riemann solver [36], which is a flux-difference scheme. Roe's scheme consists of constructing the convective flux as a sum of wave contributions:

$$(\vec{F}_c)_{I+\frac{1}{2}} = \frac{1}{2}[\vec{F}_c(\vec{V}_R) + \vec{F}_c(\vec{V}_L) - |A_{Roe}|_{I+\frac{1}{2}}(\vec{V}_R - \vec{V}_L)] \tag{2.36}$$

where the Left and Right states $(\vec{V}_L, \vec{V}_R)$ are calculated using (2.27) and (2.28) respectively. The Roe matrix A_{Roe} has the same form as the convective flux Jacobian but instead of formally averaged values, the following Roe-averaged variables are used:

$$\begin{aligned}
\tilde{\rho} &= \sqrt{\rho_L \rho_R} \\
\tilde{u} &= \tfrac{u_L\sqrt{\rho_L} + u_R\sqrt{\rho_R}}{\sqrt{\rho_L} + \sqrt{\rho_R}} \\
\tilde{v} &= \tfrac{v_L\sqrt{\rho_L} + v_R\sqrt{\rho_R}}{\sqrt{\rho_L} + \sqrt{\rho_R}} \\
\tilde{w} &= \tfrac{w_L\sqrt{\rho_L} + w_R\sqrt{\rho_R}}{\sqrt{\rho_L} + \sqrt{\rho_R}} \\
\tilde{H} &= \tfrac{H_L\sqrt{\rho_L} + H_R\sqrt{\rho_R}}{\sqrt{\rho_L} + \sqrt{\rho_R}}
\end{aligned}$$

$$\tilde{c} = \sqrt{(\gamma - 1)(\tilde{H} - \tilde{q}^2/2)}$$
$$\tilde{q} = \tilde{u}^2 + \tilde{v}^2 + \tilde{w}^2$$

In (2.36), $|A_{Roe}|$ is constructed using the absolute values of the eigenvalues and the the right eigenvector matrix R:

$$|A_{Roe}| = R^{-1}|\Lambda|R \tag{2.37}$$

Viscous Fluxes For the calculation of the Viscous Fluxes, variable values and space derivatives are needed. For the face in between cells I and J, variable values are obtained from simple averaging:

$$\vec{V}_{IJ} = \frac{1}{2}\left(\vec{V}_I + \vec{V}_J\right) \tag{2.38}$$

while for the gradients, the Green-Gauss formula is applied using the face averaged values $\vec{V}_{IJ}$ as defined in (2.38) but supplemented with a directional derivative[37]:

$$\nabla\vec{V}_{IJ} = \overline{\nabla\vec{V}_{IJ}} - \left[\overline{\nabla\vec{V}_{IJ}} \cdot \vec{t_{IJ}} - \left(\frac{\partial\vec{V}}{\partial l}\right)_{IJ}\right] \cdot \vec{t_{IJ}} \tag{2.39}$$

where,

$$\overline{\nabla\vec{V}_{IJ}} = \frac{1}{2}\left(\nabla\vec{V}_I + \nabla\vec{V}_J\right) \tag{2.40}$$

is the mean gradient,

$$\left(\frac{\partial\vec{V}}{\partial l}\right)_{IJ} \approx \frac{\vec{V}_J - \vec{V}_I}{l_{IJ}} \tag{2.41}$$

and l_{IJ} is the distance between cell centers I and J and $\vec{t_{IJ}}$ is the unit vector pointing from cell center I to cell center J.

2.2.3 Temporal Discretization

For the temporal discretization the method of lines is used. This means that temporal and spatial discretization are done separately leading for every control volume to the following equation:

$$\frac{d\left(D_I\vec{U}_I\right)}{dt} = -R_I \tag{2.42}$$

In comparison to (2.25) the form of equation (2.42)is more general in the sense that the control volume can vary with time.

Temporal discretization can be either explicit or implicit. Explicit methods use the $\vec{U}^n$ known solution and march in time using the corresponding residual $\vec{R}^n$ to obtain solution at $(t + \Delta t)$. On the other hand the implicit schemes use $R(\vec{U}^{n+1}) = \vec{R}^{n+1}$ to obtain the new solution and are favored because they allow larger time-steps. Since $\vec{R}^{n+1}$ is unknown, the following linear approximation is used:

$$\vec{R}^{n+1} \approx \vec{R}^n + \left(\frac{\partial \vec{R}}{\partial \vec{U}}\right)_n \cdot \Delta\vec{U}^n, \quad \Delta\vec{U}^n = \vec{U}^{n+1} - \vec{U}^n \tag{2.43}$$

In MaPFlow a finite difference scheme is used for the time derivative (see [38]):

$$\frac{1}{\Delta t}\left[\phi_{n+1}\left(D\vec{U}\right)^{n+1} + \phi_n\left(D\vec{U}\right)^{n} + \phi_{n-1}\left(D\vec{U}\right)^{n-1} + \phi_{n-2}\left(D\vec{U}\right)^{n-2} + \dots\right] = -R^{n+1} \tag{2.44}$$

Depending on the choice of ϕ_n the corresponding backwards difference formulae (BDF) of the temporal scheme is defined. $BDF2OPT$, refers to a class of optimized, second-order, backward difference methods with an error constant half as large as the conventional 2nd order scheme [39].

Table 2.1: Backwards Difference Schemes

order	ϕ_{n+1}	ϕ_n	ϕ_{n-1}	ϕ_{n-1}
1^{st}	1	-1	0	0
2^{nd}	3/2	-2	1/2	0
3^{rd}	11/6	-3	3/2	$-1/3$
$BDF2OPT$	$3/2 - \phi_{n-2}$	$-2 + 3\phi_{n-2}$	$1/2 - 3\phi_{n-2}$	$-0.58/3$

Steady State Computations Even when steady state simulations are considered a pseudo-unsteady technique is followed. For steady state simulations 1^{st} order scheme is chosen to march the solution in pseudo-time until convergence is reached. At 1^{st} order, after linearizing R^{n+1}, (2.44) becomes:

$$\frac{\left(D_I \Delta\vec{U}_I^n\right)}{\Delta t_I} = \vec{R}_I^n + \left(\frac{\partial \vec{R}}{\partial \vec{U}}\right)_I \Delta\vec{U}_I^n \tag{2.45}$$

By rearranging the terms the final system of discrete equations is obtained in which the system matrix defines the *implicit operator* of the scheme:

$$\underbrace{\left[\frac{(D)_I}{\Delta t_I} + \left(\frac{\partial \vec{R}}{\partial \vec{U}}\right)_I\right]}_{\text{Implicit Operator}} \Delta \vec{U}_I^n = -\vec{R}_I^n \tag{2.46}$$

Local Time Stepping

In order to facilitate convergence, the Local Time Step technique is used [40]. The time step for steady state calculation can be defined using the spectral radii of each cell. For every cell, a different time step is defined by:

$$\Delta t = CFL \frac{D_I}{\left(\hat{\Lambda}_c + C\hat{\Lambda}_v\right)_I} \tag{2.47}$$

where $\hat{\Lambda}_c, \hat{\Lambda}_v$ is the sum of convective and viscous eigenvalues over all cell faces. The convective spectral radii defined by:

$$(\hat{\Lambda}_c)_I = \sum_{J=1}^{N_f} (|\vec{u}_{IJ} \cdot \vec{n}_{IJ}| + c_i j)\Delta S_{IJ} \tag{2.48}$$

and the viscous spectral radii by:

$$(\hat{\Lambda}_v)_I = \frac{1}{D_I} \sum_{J=1}^{N_f} [max(\frac{3}{3\rho_{IJ}}, \frac{\gamma_{IJ}}{\rho_{IJ}})(\frac{\mu_L}{Pr_L} + \frac{\mu_T}{Pr_T})_{IJ}(\Delta S_{IJ})^2] \tag{2.49}$$

2.2.4 Boundary Conditions

In external aerodynamics the following boundary conditions are needed:

- Far-field Boundaries
- Solid wall Boundaries
- Symmetry Boundaries
- Periodic Boundaries

Before analyzing each one of the boundary condition types, it is important to discuss the concept of dummy cells. Dummy cells are additional virtual cells that extend the computational domain. Their purpose is to provide assistance in calculating the flow variables at the computational domain boundaries. Far-field and solid wall conditions are defined exactly at the boundary face while the other two are applied at the center of the dummy cell.

Far-field Boundaries

In the far-field, it is important to comply with the hyperbolic character of the problem as expressed when formulated in characteristic variables. The information provided is related to the sign of the eigenvalues of the flow state at the far-field boundary and the associated Riemann invariants. Both must be respected and so the far-field boundary conditions must be accordingly defined. The approach followed is based on the characteristics of the 1D Euler equations along the normal direction to the boundary [29], $u = \vec{V} \cdot \vec{n}$. The sign of each of the three eigenvalue $u, u+c, u-c$, defines the direction of propagation while along the corresponding characteristic the associated Riemann invariants $R, R+, R-$ (see Fig. 2.8).

$$R^{\pm} = u \pm \frac{2c}{\gamma - 1}, R = s \tag{2.50}$$

remain constant.

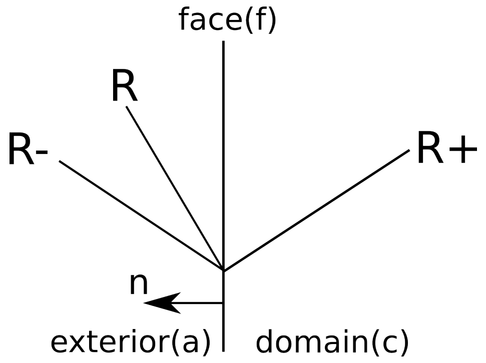


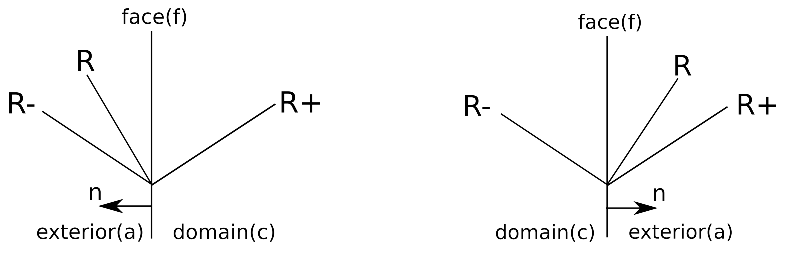
Figure 2.8: The case of a subsonic inlet face. Note that on an inlet face and the normal defined to point outwards, the normal to the boundary velocity component $u = \vec{V} \cdot \vec{n} < 0$. This means that in reality the flow information associated to $R, R-$ is provided by the state defined in (a).

Thus on an inflow face (and similarly for an outflow face), using the invariants:

$$
\begin{aligned}
&\text{from } R^+ && u_f + \frac{2c_f}{\gamma-1} &&= u_c + \frac{2c_c}{\gamma-1} \\
&\text{from } R^- && u_f - \frac{2c_f}{\gamma-1} &&= u_a - \frac{2c_a}{\gamma-1} \\
&\text{isentropic assumption} && s_f &&= s_a
\end{aligned}
\tag{2.51}
$$

Although the flow variables at boundary faces can be obtained as linear combinations of these invariants, in the present formulation, the characteristic equations are used instead:

$$
\begin{aligned}
d\rho - \frac{1}{c^2}dp = 0 &\qquad \text{along} \qquad \lambda_1 = u \\
du + \frac{1}{\rho c}dp = 0 &\qquad \text{along} \qquad \lambda_2 = u + c \\
du - \frac{1}{\rho c}dp = 0 &\qquad \text{along} \qquad \lambda_3 = u - c
\end{aligned}
\tag{2.52}
$$



(a) Subsonic Inlet (b) Subsonic Outlet

Figure 2.9: Riemann Invariants on a far-field subsonic boundary

Subsonic inlet-outlet For subsonic inflow and along the normal to the boundary, the two characteristics propagate information from outside of the domain while the third propagates flow information from inside of the domain (see Fig. 2.9). The situation reverses in case of subsonic outflow where two characteristics propagate information from inside the domain while the third propagates information from outside into the flow domain. Note that the three Riemann invariants, associated with the eigenvalues, are defined with respect to the normal direction. Thus for the subsonic inlet with the normal vector pointing outwards $u - c < 0$, $u < 0$, $u + c > 0$ while for the subsonic outlet $u - c < 0$, $u > 0$, $u + c > 0$.

Based on (2.52), with the normal pointing outwards of the inlet face:

$$\begin{aligned}
\frac{p_f}{\rho_f^\gamma} &= \frac{p_a}{\rho_a^\gamma} && \text{along} && \lambda_1 = u, (R) \\
p_f - p_c + \frac{1}{\rho_a c_a}(u_f - u_c) &= 0 && \text{along} && \lambda_2 = u + c, (R+) \\
p_f - p_a - \frac{1}{\rho_a c_a}(u_f - u_a) &= 0 && \text{along} && \lambda_3 = u - c, (R-)
\end{aligned} \tag{2.53}$$

By combining R^+ and R^- pressure and velocity at the boundary are determined. Density can be retrieved from the isentropic relation. As reference state at the inlet, that at the exterior of the domain is used. Similarly, under the assumption that the normal direction is pointing outside of the domain, at the outlet boundary,

$$\begin{aligned}
\frac{p_f}{\rho_f^\gamma} &= \frac{p_c}{\rho_c^\gamma} && \text{along} && \lambda_1 = u, (R) \\
p_f - p_c + \frac{1}{\rho_c c_c}(u_f - u_c) &= 0 && \text{along} && \lambda_2 = u + c, (R+) \\
p_f - p_a - \frac{1}{\rho_c c_c}(u_f - u_a) &= 0 && \text{along} && \lambda_3 = u - c, (R-)
\end{aligned} \tag{2.54}$$

in which the reference state is defined from inside of the computational domain.

Wall Boundary Conditions

Inviscid Wall When the fluid is assumed inviscid on solid boundaries,

$$(\vec{u} - \vec{u}_g) \cdot \vec{n} = 0 \tag{2.55}$$

where $\vec{u}_g$ denotes the grid velocity. Density and pressure are set equal to their values at the cell center next to the wall,

$$p_w = p_I,\ \rho_w = \rho_I \tag{2.56}$$

Viscous Wall In the general case, the fluid is viscous and the no slip wall condition is applied,

$$\vec{u} = \vec{u}_g \tag{2.57}$$

Density and pressure are treated as in the inviscid case. Regardless the assumptions made for the fluid, the convective fluxes take the form,

$$\vec{F}_{cwall} = \begin{pmatrix} 0 \\ n_x p_w \\ n_y p_w \\ n_z p_w \\ p_w V_g \end{pmatrix} \tag{2.58}$$

where $V_g = \vec{u}_g \cdot \vec{n}$

2.2.5 Turbulence Modeling

In order to account for turbulence modeling, flow variables are split in their mean:

$$\overline{u_i} = \lim_{T\to\infty} \frac{1}{T} \int_t^{t+T} u_i dt \tag{2.59}$$

and fluctuating parts u_i':

$$u_i = \overline{u}_i + u_i' \qquad \text{with} \qquad \overline{u_i'} = 0 \qquad \text{but} \qquad \overline{u_i' u_j'} \neq 0 \tag{2.60}$$

The above is known as Reynold's time averaging and is suitable for statistically stationary turbulence. In compressible flows due to the fluctuation of density, the Favre (Mass) Averaging is applied:

$$\widetilde{u}_i = \frac{1}{\overline{\rho}} \lim_{T\to\infty} \frac{1}{T} \int_t^{t+T} \rho u_i dt \tag{2.61}$$

$$u_i = \widetilde{u}_i + u_i'' \quad \text{with} \quad \widetilde{\rho u_i} = \overline{\rho}\tilde{u}_i, \quad \overline{\rho u''} = 0 \quad \text{but} \quad \overline{u_i''} \neq 0 \tag{2.62}$$

Favre's averaging is similar to the Reynold's one but not identical. Again, $\widetilde{u_i''} = 0$ and $\widetilde{u_i'' u_j''} \neq 0$.

Application of Favre's averaging to the governing equations, leads to a considerably more complex system. Thus, Reynold's averaging is only applied to density and pressure while Favre's averaging to all other variables [37]. Dropping the bar and the tilde sign for the averaged variables:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} &= 0 \\
\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} &= -\frac{\partial p}{\partial t} + \frac{\partial}{\partial x_j}\left(\tau_{ij} - \rho \widetilde{u_i'' u_j''}\right) \\
\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u_j H}{\partial x_j} &= \frac{\partial}{\partial x_j}\left(k\frac{\partial T}{\partial x_j} - \rho \widetilde{u_j'' h''} + \widetilde{\tau_{ij} u_i''} - \rho \widetilde{u_j'' K}\right) \\
&\quad + \frac{\partial}{\partial x_j}\left[u_i\left(\tau_{ij} - \rho \widetilde{u_i'' u_j''}\right)\right]
\end{aligned} \tag{2.63}$$

where $\rho K = 1/2\rho\widetilde{u_i'' u_i''}$ denotes the Turbulent Kinetic Energy.

The above system defines the compressible Reynolds-averaged Navier-Stokes equations or the Favre-averaged Navier-Stokes equations. By introducing the Favre-Averaged Reynolds-stress tensor as:

$$\tau_{ij}^F = -\overline{\rho}\widetilde{u_i'' u_j''} \tag{2.64}$$

and by neglecting temperature variations, molecular diffusion of K and turbulent transport, the equations (2.63) become:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} &= 0 \\
\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} &= -\frac{\partial p}{\partial t} + \frac{\partial}{\partial x_j}\left(\tau_{ij} + \tau_{ij}^F\right) \\
\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u_j H}{\partial x_j} &= \frac{\partial}{\partial x_j}\left[u_i\left(\tau_{ij} + \tau_{ij}^F\right)\right]
\end{aligned} \tag{2.65}$$

Eddy-Viscosity Hypothesis The turbulence models implemented in MaPFlow are first order closures based on the Boussinesq approximation for the Reynold's stresses:

$$\tau_{ij}^F = 2\mu_T\left(S_{ij} - \frac{1}{3}\frac{\partial u_k}{\partial x_k}\right) - \frac{2}{3}K\delta_{ij} \tag{2.66}$$

where:

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.67}$$

and μ_T denotes the turbulent molecular viscosity. Depending on the specific turbulence model, turbulent kinetic energy is either used or neglected in the state equation,

$$p = (\gamma - 1)\left[\rho E - \rho\frac{u^2 + v^2 + w^2}{2} - \rho K\right] \tag{2.68}$$

Menter $k - \omega$ SST Turbulence Model

Menter's Shear Stress Transport (SST) turbulence model [41] is a modification of Wilcox's two equation Eddy-Viscosity model [42] defined for the turbulence kinetic energy K and the specific dissipation rate ω. The transport equations of k and ω are given below:

$$\frac{\partial \rho K}{\partial t} + \frac{\partial}{\partial x_j}(\rho K u_j) = P - \beta^\star \rho \omega K + \frac{\partial}{\partial x_j}\left[(\mu + \sigma_K \mu_T)\frac{\partial K}{\partial x_j}\right] \tag{2.69}$$

$$\frac{\partial \rho \omega}{\partial t} + \frac{\partial}{\partial x_j}(\rho \omega u_j) = \frac{\gamma}{\nu_T} P - \beta^\star \rho \omega^2 + \frac{\partial}{\partial x_j}\left[(\mu + \sigma_\omega \mu_T)\frac{\partial K}{\partial x_j}\right] + 2(1-F_1)\frac{\rho \sigma_{\omega 2}}{\omega}\frac{\partial K}{\partial x_j}\frac{\partial \omega}{\partial x_j} \tag{2.70}$$

where:

$$P = \tau_{ij}\frac{\partial u_i}{\partial x_j}, \qquad \tau_{ij} = \mu_T\left(2S_i - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\right) - \frac{2}{3}\rho K \delta_{ij} \tag{2.71}$$

S_{ij} is the stress tensor given by (2.67) and $\nu_T = \mu_T/\rho$. Eddy viscosity μ_T is given by:

$$\mu_T = \frac{\rho \alpha_1 K}{max(\alpha_1 \omega, \Omega F_2)} \tag{2.72}$$

with Ω being the vorticity magnitude.

The constants for Menter's SST turbulence model are a blend of inner (1) and outer (2) constants defined by the following weighted expression:

$$\phi = F_1 \phi_1 + (1 - F_1)\phi_2 \tag{2.73}$$

with F_1 being defined by:

$$F_1 = tanh(\text{arg}_1^4), \qquad \text{arg}_1 = min\left[max\left(\frac{\sqrt{K}}{\beta^\star \omega d}, \frac{500\nu}{d^2\omega}\right), \frac{4\rho\sigma_{\omega 2} K}{CD_{K\omega} d^2}\right] \tag{2.74}$$

$$CD_{k\omega} = max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial K}{\partial x_j}\frac{\partial \omega}{\partial x_j}, 10^{-20}\right) \tag{2.75}$$

$$F_2 = tanh(\text{arg}_2), \qquad \text{arg}_2 = max\left(2\frac{\sqrt{K}}{\beta^\star \omega d}, \frac{500\nu}{d^2\omega}\right) \tag{2.76}$$

In the above, d is the distance from the cell center to the viscous wall boundary.

The inner constants (those denoted with subscript 1) are:

$$\gamma_1 = \frac{\beta_1}{\beta^\star} - \frac{\sigma_{\omega 1}\kappa^2}{\sqrt{\beta^\star}}, \tag{2.77}$$

$$\sigma_{k1} = 0.85, \qquad \sigma_{\omega 1} = 0.5, \qquad \beta_1 = 0.075$$
$$\beta^\star = 0.09, \qquad \kappa = 0.41, \qquad \alpha_1 = 0.31$$

The outer constants (those with subscript 2) are:

$$\gamma_2 = \frac{\beta_2}{\beta^\star} - \frac{\sigma_{\omega 2}\kappa^2}{\sqrt{\beta^\star}} \tag{2.78}$$

$$\sigma_{k2} = 1.0, \qquad \sigma_{\omega 2} = 0.856, \qquad \beta_2 = 0.0828$$

In [43], a limiter on the production term in the K equation is recommended. Hence, the production term in the K-equation is replaced by:

$$P = min(P, 20\beta^\star \rho K \omega) \tag{2.79}$$

Boundary Conditions The boundary conditions as defined in [41] are:

$$\frac{U_\infty}{L} < \omega_\infty < 10\frac{U_\infty}{L}, \qquad \frac{10^{-5}U_\infty^2}{Re_L} < K_\infty < \frac{10^{-1}U_\infty^2}{Re_L} \tag{2.80}$$

$$\omega_{wall} = \frac{6\nu}{\beta_1 d_1^2}, \qquad K_{wall} = 0 \tag{2.81}$$

where L is the approximate length of the computational domain and d_1 the distance to the next point off the wall.

Discretization

Convective terms: For the convective term first order up-winding is applied:

$$\rho u_j n_j U_T = max(\rho_L V_{Ln} U_{TL}, 0) + min(\rho_R V_{Rn} U_{TR}, 0) \tag{2.82}$$

in which the left and right states are the values at the cell, centers I, J and the corresponding contravariant velocity.

$$U_L = U_I, \qquad U_R = U_J, \qquad V_{(\cdot)n} = \vec{u}_{(\cdot)} \cdot \vec{n} \tag{2.83}$$

Diffusion terms The discretization of the diffusion terms consists of first order central differencing. The value of the diffusion terms at a face is taken as an arithmetic mean of the values at the center of the cells sharing the face. The gradient that appears on that term is calculated in the same manner as the gradients of flow variables that contribute to the governing equation diffusion terms (2.39).

Finally the discretization of the temporal and source terms is done in the same manner as for the governing equations.

2.2.6 Solution of the System

The final form of the discrete equations corresponds to a linear system

$$AX = B \tag{2.84}$$

of large dimension. The above system can be either solved directly or iteratively. Direct solvers are accurate but have demanding memory requirements and are not easily parallelised. On the contrary, iterative solvers might need many iterations to converge but are suitable for parallel coding and have limited memory requirements. Therefore in the present work, an iterative solver was chosen. Following the work of [44], the Jacobi iterative solver was implemented.

Noting that all but one terms in the discrete form of the equations for cell I,

$$\left[\frac{(D)_I}{\Delta t_I} + \left(\frac{\partial \vec{R}}{\partial \vec{U}}\right)_I\right] \Delta \vec{U}_I^n = -\vec{R}_I^n \tag{2.85}$$

refer to the cell in consideration, the following splitting

$$D_I \Delta \vec{U}_I^n + O_I \sum \Delta \vec{U}_J^n = -\vec{R}_I^n \tag{2.86}$$

is introduced. In (2.86), the first term is block diagonal and is linked to cell I, while the second contains the off diagonal contributions in (2.85) that are linked to $\left(\frac{\partial \vec{R}}{\partial \vec{U}}\right)_I$. This term involves the cells that surround I.

Jacobi iterative solver Equation 2.86 can be easily solved iteratively using the Jacobi method:

$$D_I \Delta \vec{U}_I^{n,k+1} = -\vec{R}_I^n - O_I \sum \Delta \vec{U}_J^{n,k} \tag{2.87}$$

where k is the Jacobi iterations index.

Gauss-Seidel iterative solver As an alternative, the Gauss-Seidel method can be used. It is similar to Jacobi solver, except the fact that the off diagonal terms are calculated using the current update for $\vec{U}$,

$$D_I \Delta\vec{U}_I^{n,k+1} = -\vec{R}_I^n - O_I \sum \Delta\vec{U}_L^{n,k+1} - O_I \sum \Delta\vec{U}_J^{n,k} \tag{2.88}$$

where U_L concerns the cell values that have been updated in $k+1$ iteration.

The performance of the Gauss-Seidel method strongly depends on the type of the matrix A in (2.84). If A is banded, the matrix can be split in an Upper and Lower part and thus Gauss-Seidel becomes:

$$D_I \Delta\vec{U}_I^{n,k+1} = -\vec{R}_I^n - O_I \sum \Delta\vec{U}_L^{n,k+1} - O_I \sum \Delta\vec{U}_R^{n,k} \tag{2.89}$$

However if the sparsity of A is substantial, the Gauss-Seidel solver has the same convergence properties as the Jacobi method [45].

In case the grid is structured, the matrix is indeed banded and the Gauss-Seidel solver will behave well. On the contrary, if an unstructured grid is used, because the band width of the matrix depends on the cell numbering, good performance is directly linked to proper renumbering. In this respect the Reverse Cuthill-Mckee (RCM) reordering scheme [45] substantially reduces the band-width and therefore the Gauss-Seidel methods outperforms the Jacobi solver.

It is important to note here that in a parallel environment even if Gauss-Seidel iterative solver is used the update ΔU_J^n must remain in the k iteration if U_J is a multi-block ghost cell. The reason behind this constraint is to ensure that the solution will be continuous across the blocks at all times.

2.2.7 Deforming Grids

Often the grid must deform, as in the case of a deformable trailing edge flap [46] or fluid-structure interaction. In such cases, on one hand the grid deformation must ensure that the grid lines do not overlap and that the change of the cell volume is taken into account.

For grid deformation, the work by Zhao [47] was followed. The idea in Zhao's scheme is to propagate the displacements of the solid boundaries into the grid without changing the far-field boundary while keeping the same grid topology. This is carried out at nodal level as follows:

$$\vec{dr}(node) = f(node)\vec{dr}(node_{wall}) \tag{2.90}$$

where $\vec{dr}$ is the displacement of the any grid node, $\vec{dr}(node_{wall})$ is the displacement of a node on the solid boundary and f is the propagation function.

For a two-dimensional problem:

$$f(x) = \frac{ly^2(x)}{lx^2(x) + ly^2(y)} \tag{2.91}$$

$$\begin{aligned} lx(x) &= \frac{1 - \exp(-d(x)/d_{max})}{(e-1)/e} \\ ly(x) &= \frac{1 - \exp(1 - d(x)/d_{max})}{(e-1)/e} \end{aligned} \tag{2.92}$$

where $d(x)$, is the distance of the node to the nearest solid node and d_{max} is the maximum distance of all nodes from the solid boundary.

Grid deformation will render the cell volume $D(t)$ time dependent. Thomas and Lombard [48] proposed the so called Geometric Conservation Law (GCL)

$$\frac{d}{dt}\int_{D(t)} dD = \oint_{\partial D(t)} \vec{V}_g \cdot \vec{n} dS \tag{2.93}$$

The principle of GCL is that a uniform flow solution must remain unchanged regardless of the grid motion.

Various numerical implementations of the GCL are found in the literature (e.g [49]). In the present work the implementation in [50] is adopted, which consists of adding a source term to the original equations. Starting from the integral form of the equations and assuming volume averaged approximation,

$$\frac{d}{dt}(\vec{U}D) + R = 0 \tag{2.94}$$

it follows that

$$\frac{d\vec{U}}{dt}D + \frac{dD}{dt}\vec{U} + R = 0 \tag{2.95}$$

So by introducing (2.93),

$$\frac{d\vec{U}}{dt}D + R = -\vec{U}\oint_{\partial D(t)} \vec{V}_g \cdot \vec{n} dS \tag{2.96}$$

and applying (2.44), the following discrete formulation is obtained,

$$\frac{1}{\Delta t}\left[\phi_{n+1}\vec{U}^{n+1} + \phi_n\vec{U}^n + \phi_{n-1}\vec{U}^{n-1} + \phi_{n-2}\vec{U}^{n-2} + \dots\right]\cdot D^{n+1} = -R^{n+1} - \vec{U}\oint_{\partial D(t)} \vec{V}_g \cdot \vec{n} dS \tag{2.97}$$

It is noted that for rigid body motions $\oint_{\partial D(t)} \vec{V}_g \cdot \vec{n} dS \approx 0$.

2.3 Mesh Generation

MaPFlow requires a computational grid (also called mesh) to solve the URANS equations on to. The mesh can either be two or three dimensional depending on the task at hand. For example, studying an airfoil does not require solving the problem in the three dimensions as the geometry is the same along the one out of the three axis and the problem can be simplified in two dimensions. On the other hand, studying the water's interaction with a ship's propeller is a problem that cannot be simplified to two dimensions. Contrast to an airplane's airfoil, a propeller's blade often has variable geometry along all of its axis, disabling us from further simplifying the problem. So a mesh can either be a computer surface or volume which describes the relationship between the target geometry and the control volume. In the current study, a 2-D mesh was developed to study the air flow around the RAE-2822 airfoil.

Creating a mesh is not an easy process as there are many thing to take into consideration, like the formation of the boundary layer, having enough cells to minimize arithmetic diffusion, refining the mesh where is needed to account for the wake, etc. Thus it becomes apparent that creating a successful is an elaborate process that prerequisites having a deeper understanding of the problem's physics and the arithmetic characteristics of the solver of choice. As already mentioned in the previous section, MaPFlow is an Eulerian solver that tackles the fluid simulations using an Eulerian frame of reference. That means that in each node of the mesh the fluid's properties are calculated with respect to a fixed location. Also, being an Finite Volume Solver means that the computational grid constitutes of smaller entities or sub-volumes in which the fluid's properties are stored while the simulation takes place.

These arguments imply that we need to construct the mesh with regard to physical phenomena that manifest around certain areas of the control volume. Areas like the proximity of the solid boundary or the area of the wake manifestation are filled with intricate phenomena that require greater grid resolution to yield an accurate result. On the other hand, the Far-field area where the flow has its free-stream characteristics is easier to resolve without requiring too much resolution. However, in the current study, we face another problem, as well. As the main objective is to train a Neural Network, there is a need to create lots of airfoil variants that would need to be resolved in a reasonable time frame. So, having a high resolution grid is not the optimal solution contrast to the general case as more detail requires more time to resolve the simulation. Considering the above-mentioned arguments, it becomes apparent that the created mesh needs special treatment. Of course, it is mandatory to have very refined mesh close to the solid boundary and the wake formation area yet some resolution "discounts" will happen in

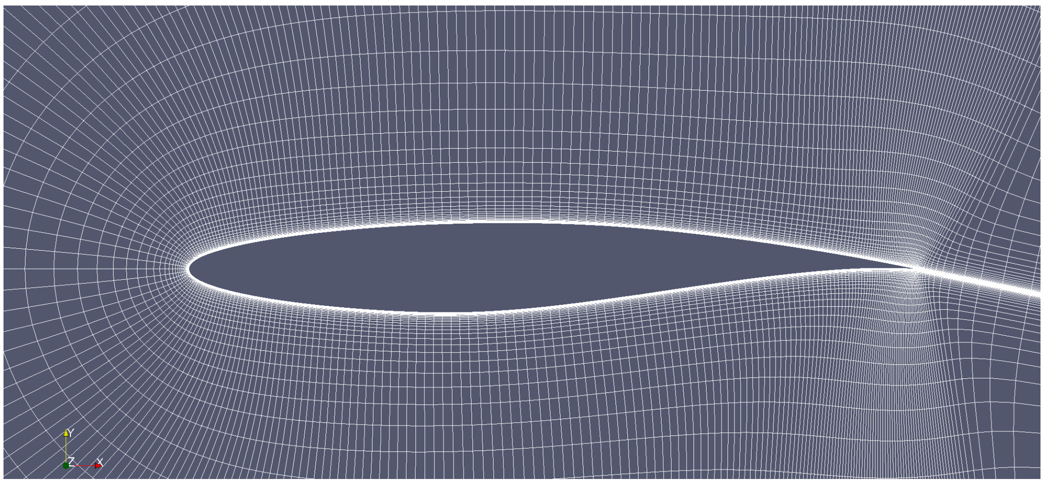


Figure 2.10: The boundary layer's area mesh structure

the Far-field cells.

First things first, the basic principles of Mesh generation must be presented. In meshes used by Eulerian CFD solvers, it is mandatory to capture a significant part of the Far-field to have a successful simulation. It is common practice to create a mesh with a far-field length of around 5-10 times the airfoil's chord length to accurately approximate the free-stream characteristics of the flow far from the solid boundary. In this area, there is no need to have a very refined mesh and a moderate resolution is used. However, as we get closer to the solid boundary there is a gradual refinement of the mesh to create a smooth transition from the far-field to the solid boundary.

On the other hand, the mesh needs to be refined close to the boundary. Having high resolution near the boundary is mandatory to capture the viscous boundary layer that forms. In case of not having fine enough mesh, the solution will probably experience numerical instabilities and will generally converge to a significantly high error. So, near the boundary layer the mesh is very refined, as seen in figure : 2.10, and also is structured compared to the entirety of the mesh which is unstructured. The structured cells have an almost rectangular shape, with one side being adjacent to the boundary and the other being normal to the boundary. Using a structured array of cells enables us to capture the longitudinal behavior of the flow better than having an unstructured one. Also, the cell's normal side is gradually declining while the boundary is approached from the far-field to better capture the development of the viscous boundary layer and analyze the flow field close to the solid boundary. Another area of interest is the airfoil's wake where the disturbed from the solid boundary flow develops. There we can observe the

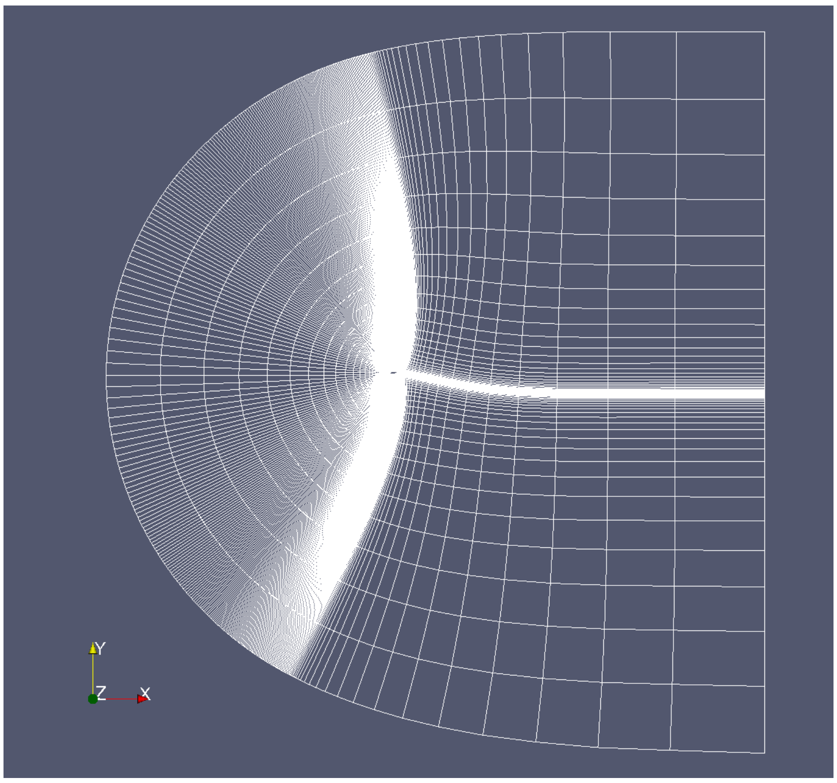


Figure 2.11: The active mesh structure used for the CFD simulations

formation of large or smaller vortices whose formation is explicitly connected to the flow around the airfoil.

All the above mentioned parameters must be taken into consideration while creating a mesh. However, there is no absolute way to build a mesh that would guarantee absolutely zero error. Building a mesh is a rigorous process that requires extensive knowledge of the physical phenomena, lots of trial and error, and of course understanding of how the solver of choice works. As will later be discussed in greater detail, in section 4.4, many computational grids were created to use for the simulation of the RAE-2822 with various degrees of precision. Even-though in theory all the meshes created complied with the above mentioned criteria, relatively small differences between them had significant implications in the accuracy of the results. Also, the meshes created by experts in the field with years of experience had better accuracy, while retaining the same number of cells with the one created by the author, who has the least experience on the field. This proves that creating the geometrical input of a CFD solver is no easy task, consuming great amounts of time and requiring extensive knowledge on the field.

Chapter 3

Artificial Intelligence

3.1 Mimicking the human brain

Artificial Intelligence is intelligence demonstrated by a machine. As defined Artificial Intelligence, for short AI, is pretty vaguely defined, thus hinting the fact that can be utilized in a great spectrum of applications, ranging from defeating the world chess champion Garry Kasparov in 1997 [51] to suggesting the next video Youtube shall play or even identifying a person, while analyzing video footage from CCTV. Such a great variety of applications may suggest that AI is more than meets the eye.

Artificial Intelligence can manifest in different ways, generalized in 2 categories: Problems defined by strict rules (ie. a game of Chess) and abstract problems (ie. hand-written numbers classification [52]). Problems dictated by strict rules can be run and programmed quite efficiently on a Computer as they involve strict logic and predetermined patterns [53]. On the other side, abstract problems are not that easy to solve by imposing pure predetermined logic as abstraction introduces incredible complexion. For example, the MNIST problem is a common problem associated with AI, where the computer needs to determine what number a person wrote. This problem is quite intimidating to solve with hard-coded logic. However, for us humans it is pretty easy to determine what a hand-drawn digit is, as our brains can recognize the patterns that imply what number a digit corresponds to. That is why AI researchers focused their efforts on mimicking the human brain. More specifically they wanted to mimic the brain's learning capability, its ability to learn and memorize new patterns by presented to raw data [53].

The process of training a machine to recognize certain patterns based on given raw data is called Machine Learning, or ML for short. ML is a subset of AI, as shown in figure 3.1, as it enables a machine to "understand" more

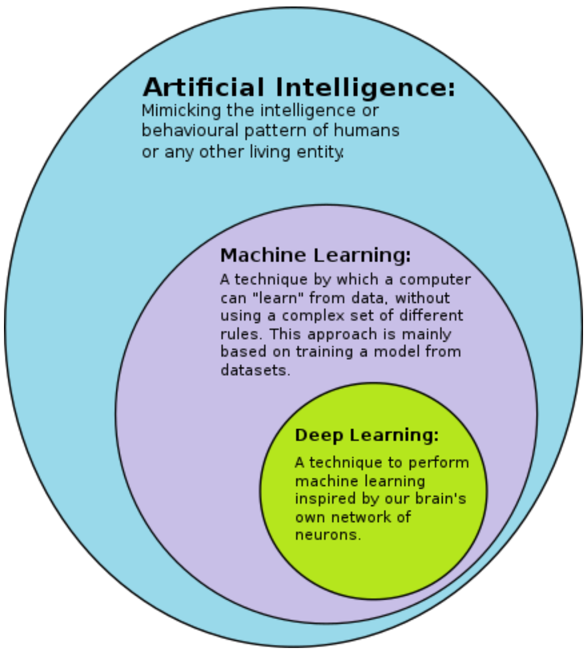

Artificial Intelligence:
Mimicking the intelligence or behavioural pattern of humans or any other living entity.
Machine Learning:
A technique by which a computer can "learn" from data, without using a complex set of different rules. This approach is mainly based on training a model from datasets.
Deep Learning:
A technique to perform machine learning inspired by our brain's own network of neurons.

Figure 3.1: Venn Diagram of Artificial Intelligence [54]

complex and abstract problems. ML as mentioned above relies on data to learn and enable itself to recognize patterns and solve abstract problems, yet giving a machine "raw data" to "crunch" is not that easy. Many Machine Learning algorithms rely on proper data structure to be able to learn. Using again the MNIST paradigm mentioned above, presenting an image to a computer or an matrix with "random" values as perceived by a computer, holds no valuable information for a "simplistic" ML algorithm. On the contrary, an ML algorithm presented with properly structured data can learn and evaluate a result. To overcome this problem, researchers started mimicking not only the human brain's capacity to learn but its structure too.

Deep Learning is the process where the machine can build complex concepts out of simpler concepts [53]. Of course, Deep Learning is a part of ML and AI, as shown in figure 3.1, as it is the AI method better suited to observe the true abstraction of the natural world. To achieve that feat, Deep Learning Algorithms utilize arithmetic models that resemble the human brain's structure. The cornerstone example of Deep Learning models, as claimed by [53], is the feedforward deep network of **multilayer percepetron** (MLP). The MLP is nothing more than a mathematical function that maps certain input to some output values. Specifically, the MLP consists of simple functions that are chained together in various formats, thus adding the required complexity to the function as shown in figure 3.2. These simple functions

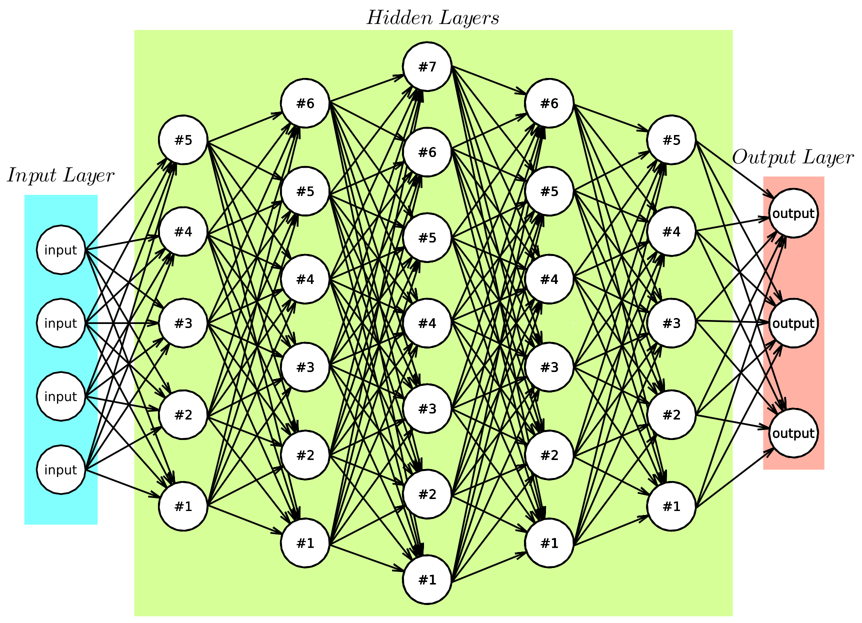
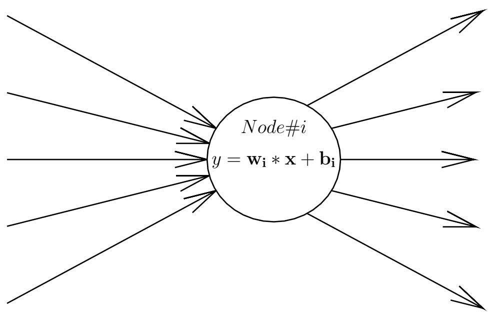


Figure 3.2: A sample Multi Layer Percepetron (MLP) Network

are graphically represented as multiple input-output nodes, excluding the input layer nodes that are the graphical representation of data entry, which are structured in layers. The layers are organized as **input layer** $\rightarrow$ **hidden layer(s)** $\rightarrow$ **output layer**, where the mathematical operations are performed on the hidden layer(s). Each node, as seen in 3.3a, 3.2, is a multi-argument function that translates its input into a single output that can be then transmitted to the next layer's nodes or be the Neural Network's final result.

To be specific, a proper mathematical model of the above-mentioned arguments shall be presented. Let $\mathbf{x}$ be the input vector of the node and y the output value. As is already well established, a node is a function that takes as an argument a vector and outputs a single value. To achieve that, let $\mathbf{w}$ be the weight matrix of the node and b the bias of the node. The weight matrix is multiplied with the input vector $\mathbf{x}$, and then the bias is added to give the output value y (equations 3.3b).



(a) An MLP node

$$y = \mathbf{w} * \mathbf{x^T} + b,$$
$$\mathbf{x} = [x_1, x_2, ..., x_N] \text{ input vector}$$
$$\mathbf{w} = [w_1, w_2, ..., w_N] \text{ weight vector}$$

(b) Node Mathematical Model

Figure 3.3: Graphical Representation and Mathematical Model of an MLP Node

3.2 Activation Functions

Nevertheless, not all problems can be approximated by linear models. To solve this problem, often are used non-linear functions that alter the input in a non-linear fashion (eq. 3.1). These Activation Functions filter the node's input, thus normalizing its output or excluding extreme input values that can jeopardize the Neural Network's stability.

$$y = \mathbf{w} * F(\mathbf{x^T}) + b, \tag{3.1}$$

where F($\mathbf{x}$) is the activation function

There are several Activation Functions used in Neural Network Algorithms, and figure 3.4 presents the graphs of three of the most common ones. The figure 3.4 contains the graphs of these three activation functions, along with the graph of the default linear behavior for reference. Observing the different functions one can observe how they alter the node's response to input. For example, Rectified Linear Unit (relu for short) has the same behavior as the linear response for positive values while excluding any negative values. On the other hand, both sigmoid and the Hyperbolic Tangent (tanh) functions normalize the input values around $(-1, 1)$, while having slightly different behavior.

$$y = max(0, x), \text{ Rectified Linear Unit } relu(x) \tag{3.2}$$

$$y = \frac{1}{1 + e^{-}x}, \text{ Sigmoid function } \sigma(x) \tag{3.3}$$

$$y = \frac{e^{2x} - 1}{e^{2x} + 1}, \text{ Hyperbolic Tangent } tanh(x) \tag{3.4}$$

Relu is very useful, having all the benefits of the linear activation function (ie. easy to differentiate and optimize the network) while filtering the input, excluding extreme values. The standard Relu is described by the equation 3.2 and during this thesis whenever the name relu is referred, this is the equation that would refer to. This is done to avoid further confusion later, excluding the other Relu variants that are commonly used, for example, Leaky Relu [55] and Parametric Relu [56], to name a few.

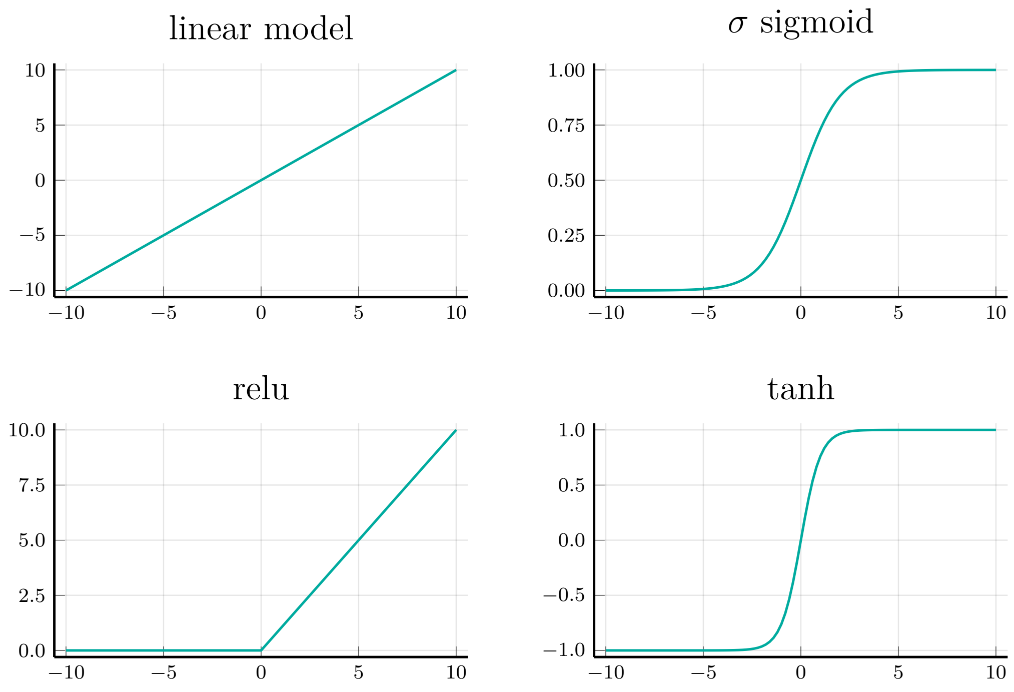


Figure 3.4: Activation Functions Plots

3.3 Introduction to Convolutional Neural Networks

Feed Forward Neural Networks, like MLP, have some deficiencies when it comes to dealing with two-dimensional input, like images. As seen in section 3.1, their input is a one-dimensional array of arithmetic data. On the other hand, an image is a file that consists of a three-dimensional array, which when read by an appropriate computer program is translated as different colored pixels on the end user's screen, print out, etc.

Natively, the computer cannot store colors as they manifest in nature, but it can relate colors on a visual output device to numerical values by composing them out of the three or more basic colors. The industry-standard formats when it comes to storing and projecting images to a computer screen are the RGB (Red Green Blue) and RGBA formats, where the latter includes an alpha channel to model transparency [1]. A screen's pixel element consists of 3 subelements that each corresponds to one of the three colors. By varying these elements' luminosity amplitude we create the illusion of the pixel having a color, providing we look at the screen from far enough. These amplitudes for each pixel's chromatic subelements are these RGB values, commonly ranging from 0 to 255 [2]. So an image is nothing more than a two-dimensional array that each cell holds an RGB triplet. However, for Machine Learning applications it is more convenient to think of it as a three-dimensional array, which holds three two-dimensional arrays one for each color value.

After the previous analysis, it becomes apparent that a Feed-Forward Net can not take image data natively, without some formatting. For example, we could arrange the pixel cells in order and then flatten the RGB values like

$$[RGB_1, RGB_2, ..., RGB_N] - > [Red_1, Green_1, Blue_1, ..., Red_N, Green_N, Blue_N]$$

, where N is the number of total pixels $N = height_{px} * length_{px}$.

Another attempt to solve this problem is to use binary colormaps (grayscale color spaces). Then the problem is simplified a lot as there is no need to input image data anymore. Using a grayscale image, the pixels have only alternating luminosity ranging from 0 to 255 rather than 3 distinct color channels. Thus, the image data can be normalized as values ranging from 0 to 1, in a single one dimensional array. So the multi-dimensional input data can be simplified to one dimensional. Some applications of grayscale images with Feed-Forward Networks are some of the classic MNIST's examples [52], fashion MNIST's examples [57], Sekar et al. work [18], and many more.

However, there is an even greater problem looming. For example, during image classification, we are especially interested in the patterns that are forming on the image and what underlying information they carry. To prepare for MLP's input, the image data are flattened, and locally forming patterns

[1]The alpha channel has no physical manifestation on a physical screen. It is something handled internally by the graphics card to generate overlap of different shaders and colorize the screen properly

[2]This range occurs as the common format for images is RGB 8bit. This protocol dictates that each chromatic value is represented by a variable with a size of 8 bits or 1 byte in computer memory. As there is no interest for negative numbers this variable is an Unsigned Integer of size 8 bit, with minimum value 0 and maximum value 255. ($8\ bits - > 2^8 = 256\ combinations$)

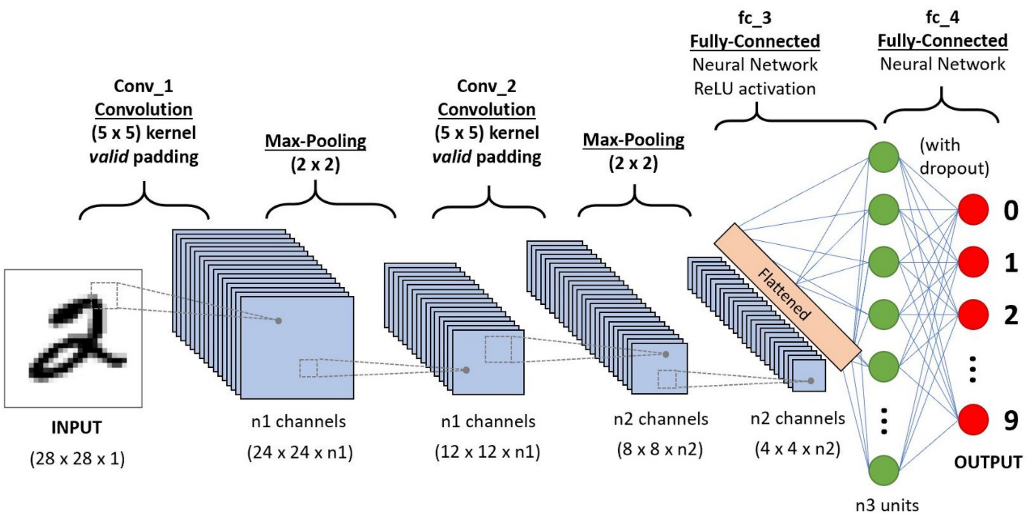


Figure 3.5: A typical Convolutional Neural Network [58]

can no longer be observed as the algorithm ruins any information carried by the image's dimensionality. This problem is even more apparent in colored images as there the color information is significant as well.

Contrary to the single dimensionality of MLPs exist the Convolutional Neural Networks (CNN). Drawing inspiration from the mammalian brain's visual system researchers were inspired to create CNNs [53]. The convolution operation can be described as follows:

$$s(t) = \int f(x)w(t-x)dx = (f \star w)(t) \tag{3.5}$$

,where $f(x)$ is a function in which the filter w is applied.

In Machine Learning function $f(x)$ is often referred as **input** while $w(t-x)$ is often referred as **kernel**. Also it is common to work with two-dimensional data, like images. So if we use a two-dimensional image I as our input, and a two-dimensional kernel K, the convolutional operation becomes as follows :

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{3.6}$$

A very accurate visual representation of how Convolution works is displayed in the figure: 3.6.

CNN's main parameters are Kernel size, stride, padding, and Channels. Kernel size is the kernel's dimensions. Channels are the number of kernels that scan the same input, extracting an equivalent number of new layers.

Stride defines how many columns of data does the kernel travels after each scan. Padding describes how many pseudo points are inserted at the edges of each layer's Channel input. Padding is useful as it can eliminate the shrinkage that occurs when the kernel is very big relative to the input data. CNNs usually consist of many Convolutional Layers with varying parameters. For example, even though it is common to have a Kernel with constant size, the number of Channels, the stride, and padding may vary.

CNNs have some significant benefits to MLPs regarding how they handle two(or more[3])-dimensional data. First, as already mentioned, they are built for having two-dimensional input. Convolution filters the current layer's information to the next, like MLPs, but, unlike MLPs, respects the position of the extracted information. As the kernel serially scans the input data, each scan passes to the next layer the extracted information at a specific discrete position. This way there is always reference to the layer above and the data do not mix, retaining any information about any locally formed patterns providing the kernel size is appropriate. Kernel size is a very important parameter to take into consideration while designing a CNN; too small and the kernel cannot extract any pattern information; too big and the kernel will not capture any detail whatsoever.

However, preserving the pattern's exact location and amplitude may not be always beneficial. There are cases were we are interested in knowing that a feature exist more than its precise location, like facial recognition where knowing that an eye exists in the image is more important than to know exactly where it is located. Also, we must not neglect the cases where an image may have some form of noise in it. This can be very common especially in classification algorithms that analyze image data from live feed, like CCTV footage where rain, sun, dust on the lens, and many more can introduce unwanted noise to the image. For mainly these reasons and many more, it is common after each convolutional layer a pooling layer to follow.

Pooling operation is vary similar to convolution as a kernel scan the input data and creates new data, but the kernel works completely different. In pooling layers the kernel is a function that transform the data in a predetermined way contrast to the trainable convolutional kernel. The most common kernel architectures are the max pooling and the average pooling. In max pooling the kernel is a maximum function, where from the input data scanned by the kernel only their maximum value is chosen and passed to the

[3]There is some research done towards the three-dimensional CNNs. However, during this thesis, we will not concern about them. For further reading here are some interesting papers: [59][60][61]

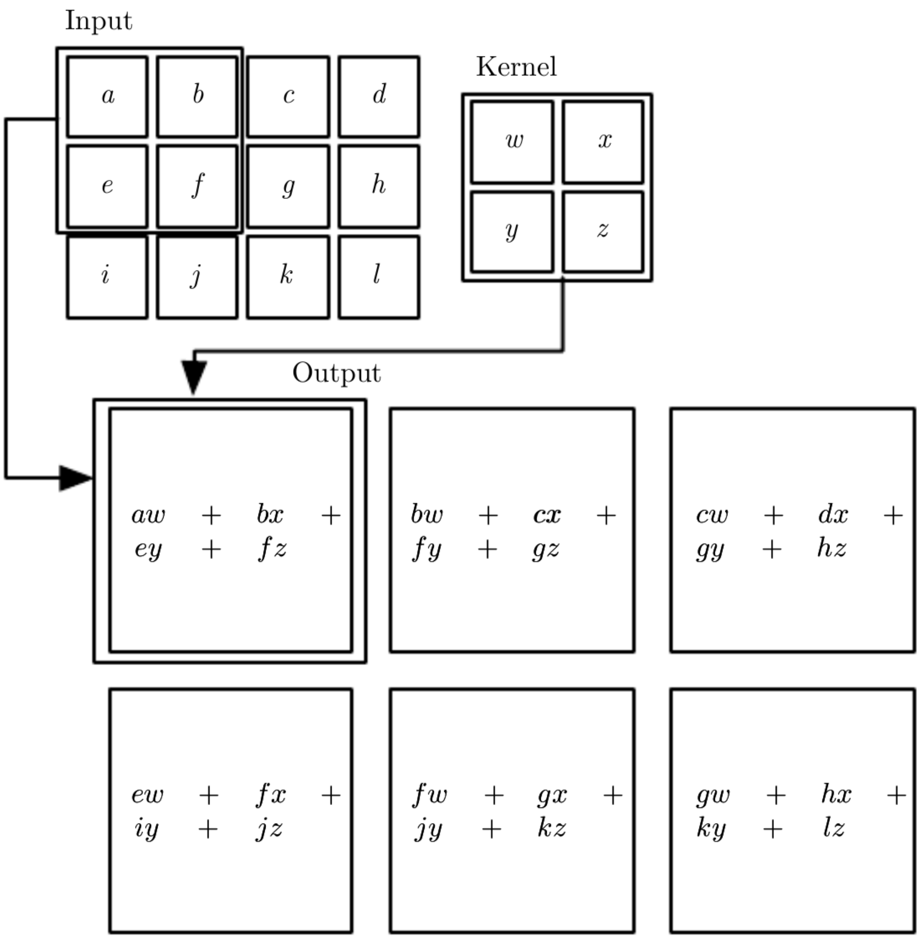


Figure 3.6: Graphical representation of the convolution operation over a two-dimensional array [53]

next layer (eq.: 3.7).

$$S(i,j) = \max_{m,\ n}(I(i-m, j-n)) \tag{3.7}$$

On the other hand, in average pooling the kernel is an averaging function, where from the input data scanned by the kernel their average value is calculated and passed to the next layer (eq.: 3.8).

$$S(i,j) = \frac{1}{n*m}\sum_{m}\sum_{n} I(i-m, j-n) \tag{3.8}$$

These pooling layers are most of the times quintessential to a CNN's architecture as the statistical benefits they bring are extremely significant. Filtering random noise or focusing the algorithm on what is important are significant in our efforts to maximize the Networks accuracy. They allow the Network to focus on the actual features we want to focus by either disregarding features that irrelevant or generalizing the input data. This way the statistical efficiency of the Network is augmented and because each Convolutional Layer is smaller there are less parameters to train boosting the training performance and cutting down training times. Yet sometimes filtering by pooling is not desirable as we want to preserve features' positional information and/ or noise is not relevant. For example, during this thesis, as we have absolute control over the image data, thus we have no undesirable noise in the data. Also, we have a keen interest in positional information as we study how small positional deformation affects the final result. So, we will not use any kind of pooling on this thesis CNNs. However, in order to boost the Networks' statistical efficiency we will use Batch Normalization [62], as proposed by Hui et al.[1], a technique that would be discussed in detail in section 3.4.3.

Finally, as seen in figure : 3.5, after the convolutional layers we usually use one or more Fully Connected Layers to flatten the data processed by the Convolutional Layers and output them. One may wonder if we use Activation Functions on the Convolutional Layers to filter the kernels output. As far as the author is concerned, there is no such thing as the kernel is responsible for capturing features that are then filtered by pooling layers. So there is no need to use an activation function. However, in the Fully Connected Layers that flatten and output data is common to use an Activation Function. For example, in figure : 3.5, the author has chosen that the Fully Connected Layers that flatten the data shall use the ReLU Activation Function.

3.4 Training and validation

Creating a Neural Network consist of two sequential process; first the Network's architecture is conceptualized and then the network is trained to fit the data. Training enables the Network to adapt to the dataset and predict whatever result is desired from the dataset. In parallel to training is useful to monitor the Network's error rate when faced with unknown to the training dataset items. By doing so it is possible to discern whether the Network has over-fitted the training dataset. Over-fitting the training dataset is something extremely harmful for the Network's performance as it renders the Network useless when presented to unknown data that are in the vicinity of the training dataset. A nice example of this problem is presented in figure 3.7. This piece is out of the work of Bishop [63], where the problem of over-fitting is demonstrated as fitting simple polynomial curves over some points that have some form of noise in them. Having a not sufficiently complex architecture for the task at hand can lead to incredible error as seen in the cases M = 0 and M = 1. On the other hand, having a more intricate system than the problem requires can be problematic, as well. As seen in case M = 9, a more intricate system can develop an over-fitting behavior, thus not being able to capture the problem's general behavior but a very specific subset of it. This analysis fully justifies the critical importance of testing the Network while it trains.

Testing is a pretty straightforward process; the algorithm is presented to new data and its performance is measured by a method to quantify error, like Mean Squared Error. On the other hand, training -which is arguably of outmost importance- is not. Similar to testing, in training we are interested in finding the algorithms error, as well. However, in training the error is used to optimize the algorithm compared to just observing its behavior. Training a Neural Network, involves around finding the algorithm's error and minimize it. To do so, we have to calculate the error and subsequently update the Network's parameters to minimize the error with an appropriate method. In order to converge to an error's local minimum, we have to repeat the training process, as defined above, many times and monitor both the training error rate and the testing error rate, observing whether the Network fitted and/or over-fitted the training data. In literature every training iteration is commonly referred to as an Epoch and that convention will be followed during this study as well.

Using Linear Algebra along with Variational Analysis, we can define the functions that are involved in training a Neural Network. First, the algorithm's error is modelled with respect to the algorithm's parameters. Let $\mathbf{x} = \{x_1, x_2, ..., x_N\}$ be the input vector of the training dataset, $f(x; \mathbf{w})$ the

Neural Network's function, and $\hat{\mathbf{y}} = \{\hat{y_1}, \hat{y_2}, ..., \hat{y_N}\}$ the target values of the training dataset. As it is convenient for later, let the Error function of the algorithm with respect to the algorithms parameters be :

$$E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} (f(x_n; \mathbf{w}) - \hat{y_n})^2 \tag{3.9}$$

This equation describes the Mean Squared Error equation with respect to the algorithm's parameters. MSE is not the only available function available as modelling the Network's error depends on the Network's task, namely cross-entropy is better for classic classification problems . MSE is arguably a good error function and finds many applications in Deep Learning due to its good statistical behavior [53].

Having defined the algorithm's error function, the next step is to minimize the error function. Variational Analysis describes that to find a function minima (or maxima) we must study its first order gradient. Moreover, local minima (maxima and saddle points) are located wherever the $\nabla E(\mathbf{w}) = 0$. So, the next steps are to find a way to calculate the Error Function's gradient and then optimize the Neural Network algorithm to reduce the error and reach a local minima.

3.4.1 Back propagation algorithm

Calculating the Error function's gradients can not be done analytically as the Function is data driven and is based on the very complex model's function. To solve this problem the Back Propagation algorithm is used. The Back Propagation algorithm relies on the concept of using the Network's Error, that is calculated on the output layer, to calculate the Error Function's gradient with respect to all the Network's parameters [63].

Let $E(\mathbf{w})$ be the Error Function, or $E(\mathbf{w}) = \sum_n E_n(\mathbf{w})$ where $E_n(\mathbf{w})$ is the error of a single data term. For many error functions commonly used in practice the above separation of the different error terms is totally valid. So, working with the error function for a single data term we can calculate its gradient as :

$$\frac{\partial E_n(\mathbf{w})}{\partial w_{ji}} = \frac{\partial E_n(\mathbf{w})}{\partial \alpha_j} \frac{\partial \alpha_j}{\partial w_{ji}} \tag{3.10}$$

Using the chain rule we can split the Error function's partial derivative with respect to the the parameters in the partial derivative of the Error Function with respect to the summed input α_j of unit j multiplied with the

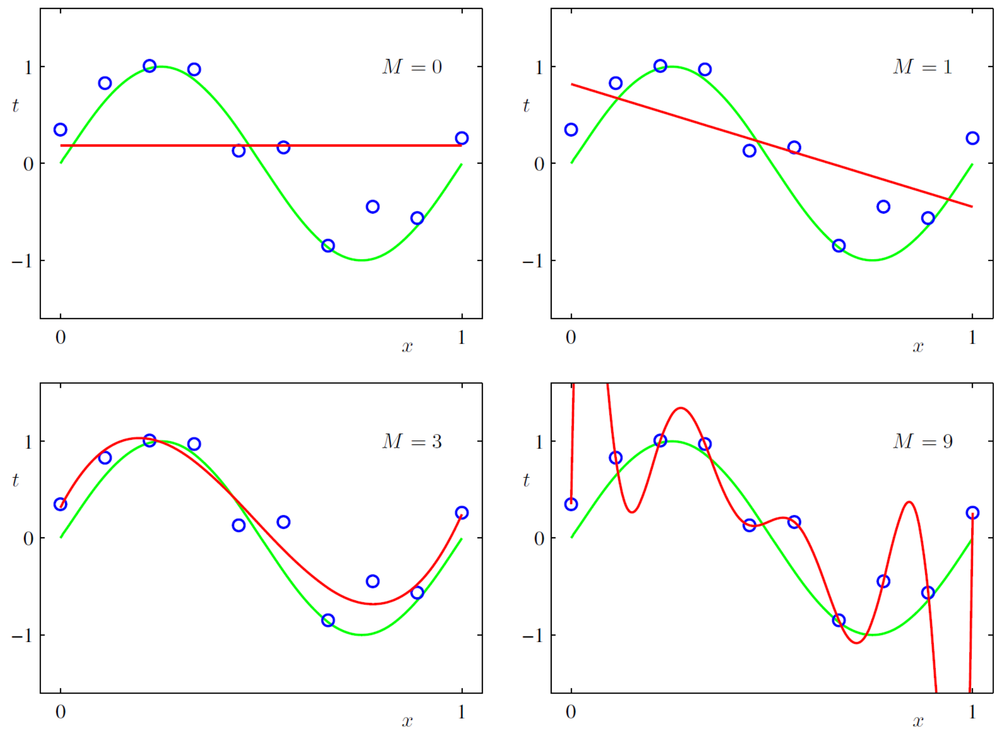


Figure 3.7: Polynomial curves fitting over sinusoid data with random noise [63]
Here is presented the fitting of different order polynomial curves to sinusoid data with some random noise. This is a simplification of a Neural Network architecture that makes crystal clear how overfitting the training dataset works. Let (x, y) be points of a cartesian grid where $y = sin(x) + N(x; \beta)$, with $N(x; \beta)$ be a Random Function of x and a Random Variable β. As seen the noise is taken as such as its magnitude only slightly shifts the y coordinate and does not radically change the data points' sinusoid behavior. Treating these data as completely random data, we want to approximate the underlying x to y transform using polynomial curves. As already known polynomials are functions where different powers of x are weighted and then added together, or $y = \sum_{i=0}^{M} a_i x^i$. It is easily observable that choosing a low order polynomial is not able to capture the dataset behavior. On the other hand, using a very high order polynomial can lead to overfitting the dataset as seen in case of M=9.

partial derivative of the α_j with respect to the parameter w_{ji}. Notating,

$$\delta_j \equiv \frac{\partial E_n(\mathbf{w})}{\partial \alpha_j} \tag{3.11}$$

and knowing that,

$$\alpha_j = \sum_i w_{ji} z_i \Rightarrow \frac{\partial \alpha_j}{\partial w_{ji}} = z_i \tag{3.12}$$

where the z_i is the activated (or not) input of the layer j . Then substituting these equations to the equation (3.10) we get:

$$\frac{\partial E_n(\mathbf{w})}{\partial w_{ji}} = \delta_j z_i \tag{3.13}$$

One useful property that arises from doing the above separation using chain rule is that the partial derivative of the Error Function with respect to the node's input can be further analyzed using the chain rule using the partial derivatives of its previous nodes. For example, to calculate the Error Function's derivative in node i we can use the information from its next node k and do the following:

$$\delta_j \equiv \frac{\partial E_n(\mathbf{w})}{\partial \alpha_j} = \sum_k \frac{\partial E_n(\mathbf{w})}{\partial \alpha_k} \frac{\partial \alpha_k}{\partial \alpha_j} = \sum_k \frac{\partial \alpha_k}{\partial \alpha_j} \delta_k \tag{3.14}$$

However as :

$$\alpha_k = \sum_k w_{jk}\ z_j \Rightarrow d\alpha_k = \sum_k w_{jk}\ dz_j \tag{3.15}$$

Also considering that unit j may not be linear activated and has an Activation Function, where:

$$z_j = f(\alpha_j) \Rightarrow dz_j = f'(\alpha_j) d\alpha_j \tag{3.16}$$

The equation 3.15 becomes :

$$d\alpha_k = \sum_k w_{jk}\ f'(\alpha_j)\ d\alpha_j \tag{3.17}$$

Then using equation 3.17 :

$$\delta_j \equiv \frac{\partial E_n(\mathbf{w})}{\partial \alpha_j} = f'(\alpha_j) \sum_k w_{jk} \delta_k \tag{3.18}$$

So using the Error of the last layers (with respect to the information vector during feedforward operation) we can calculate the error of their previous layer. The equation 3.18 can be generalized for the other units of layer j and of course for different layers as well. This way by evaluating the network during feedforward operation we can get half the data needed to calculate the Error Function's gradient and by calculating a layer's error with starting from the output and working backwards we calculate all the required errors. This makes the Back Propagation algorithm a very efficient method to evaluate the Error's parameter gradient.

3.4.2 ADAM optimization algorithm

Having calculated the Error Function's gradient with respect to the model's parameters we can finally optimize the model to reduce error. Probably the most common method to optimize a network is Gradient Descent. Gradient Descent is fairly simple to implement as it uses the Error gradient to update the Network's parameters as seen in equation 3.19.

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(x; \mathbf{w}^{(\tau)}) \tag{3.19}$$

where η is the learning rate; $\eta > 0$ is a constant arbitrarily small that is chosen by the designer of the network. Usually it is chosen in the region of $\approx 10^{-5}$ to improve the stability of the training process.

Even-though, the Gradient Descent is simple and straightforward to implement has its fair share of deficiencies. First and foremost, Gradient Descent requires the entire input dataset to be evaluated to calculate the Error's gradient making it a vary inefficient method. Even if, the input dataset are split into smaller batches to optimize the training times there are still problems using Gradient Descent. Also, the Gradient Descent is heavily reliant on the initial parameters of the Network (which are in most cases randomly selected) to achieve good convergence to low error. This behavior leads to training a single model loads of times in order to get a sufficiently low error model. These and many other problems lead the use of other training algorithms instead of Gradient Descent.

Hui et al. [1] proposed that the ADAM optimization algorithm [64] is used to train the Networks. ADAM is an algorithm for first-order gradient-based optimization of stochastic objective functions, based on estimates of lower-order moments. ADAM is based on the concept of combining the advantages of two other popular methods for optimization of Neural Networks and more specifically AdaGrad [65] and RMSProp [66]. ADAM uses first-order gradients that can be efficiently computed with the Back Propagation algorithm requiring relatively less memory than other methods. In order to train the model ADAM uses 1^{st} and 2^{nd} order moments estimates, that are further corrected with adaptive learning rates.

The ADAM algorithm similar to other optimization algorithms requires some hyper-parameters to be given by the user, namely the learning step-size α and the exponential decay rates for the moment estimates $\beta_1,\ \beta_2 \in [0, 1)$. For the current case, Hui et al. proposed that $\alpha = 0.0001$ and $\beta_1,\ \beta_2$ would be left with their default values which Kingma and Ba [64] propose as $\beta_1 = 0.9,\ \beta_2 = 0.999$.

Algorithm 1 ADAM optimization algorithm. Here the g_t^2 symbolizes the element-wise multiplication $g_t \odot g_t$. Also, β_1^t and β_2^t are raised to the power of t. By default $\epsilon = 10^{-8}$.

```
$m_0 \leftarrow 0$ (Initialization of $1^{st}$ order moment)
$u_0 \leftarrow 0$ (Initialization of $2^{nd}$ order moment)
$t \leftarrow 0$ (Initialization of timestep)
while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Gradient calculation)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $u_t \leftarrow \beta_2 \cdot u_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second moment estimate)
    $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute first bias-corrected moment estimate )
    $\hat{u}_t \leftarrow u_t/(1 - \beta_2^t)$ (Compute second bias-corrected moment estimate )
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{u}_t} + \epsilon)$ (Update parameters)
end while
return $\theta_t$ (Resulting parameters)
```

ADAM, contrast to Gradient Descent, utilizes a model of iterative calculations to converge on the new set of parameters. By continuously evaluating the moments of the Errors's gradient distribution the model's parameters converge to an optimal set that minimizes the Error Function. As the moments are initialized as vectors of zeros they are biased towards zero. This bias forces the convergence of the gradient towards zero faster than other methods although in some cases this is not desirable in some cases. This is where the bias correction comes into place.

3.4.3 Batch Normalization

A common problem that Neural Networks face during training is the Internal Covariate Shift. Internal Covariate Shift happens as the training alters each layer's parameters distribution. These changes are fed forwards in the Neural Network as changes in the previous nodes affect the input of their next ones. This effect can have negative effects in the training stability and efficiency of the optimization algorithm as it becomes significantly more difficult to train the latter layers. To generalize these problems, Ioffe and Szegedy define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training [62].

As Ioffe and Szegedy suggest, the Internal Covariate Shift may move the layers' output into the saturated regime of the non-linearity and eventually slow down convergence. They claim that using Rectified Linear Units for Activation, careful Initialization of the model's parameters and small learning rates can help reduce the Internal Covariate Shift effects. However, in many cases this is not applicable as the model's complexity or wanting to explore different hyper-parameters of the model could limit the effectiveness of said measures. This justifies the need to create a method in order to eliminate the effects of Internal Covariate Shift without limiting the features of our Network. Also, splitting the optimizer's performance from the model architecture is quite significant as training becomes more independent and can better preserve the model's nonlinear characteristics.

Ioffe and Szegedy [62] propose a technique to eliminate Internal Covariate Shift named Batch Normalization. Batch Normalization promises reduction of the Internal Covariate Shift by normalizing the layers input by fixing the means and variances of layers inputs. They also claim that Batch Normalization has beneficial effects on the gradient flow through the Network as it reduces the dependence of gradients on the scale of their parameters or their initial values. Finally, they suggest that Batch Normalization enables the Network to use saturating nonlinearities as the Batch Normalization prevents the Network from getting stuck in saturated modes during training.

Batch Normalization is a very straightforward method to implement on a Neural Network as it is based for normalizing data. Let $\mathbf{x} = [x_1, x_2, ..., x_N]$ the input vector to a layer and B a minibatch of size M that contains various instances of $\mathbf{x}$, let $B = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_M]$. Batch Normalization requires the calculation of the mean and variance for each x_n over the entirety of the minibatch based on equations : 3.20, 3.21.

$$\mu_n = \frac{1}{M}\sum_{m=1}^{M} x_{n,m} \tag{3.20}$$

$$\sigma_n^2 = \frac{1}{M}\sum_{m=1}^{M}(x_{n,m} - \mu_n)^2 \tag{3.21}$$

After obtaining the two moments using equation 3.22 the input vector parameters are normalized.

$$\hat{x_n} = \frac{x - \mu_n}{\sqrt{\sigma_n^2 + \epsilon}} \tag{3.22}$$

Just normalizing the batch based on the current batch data without respect to the entirety of the dataset is not desirable as it can probably change the layer's identity and restrict the model's capability of extracting any useful data. In order to eliminate that problem, the normalized $\hat{x_n}$ data are scaled and shifted according to equation 3.23:

$$\hat{y_n} = \gamma_n \cdot \hat{x_n} + \beta_n \tag{3.23}$$

In equation 3.23 the γ_n and the β_n are trainable parameters, that are trained to restore the representation power of the network.

According to Ioffe and Szegedy [62], Batch Normalization have applicable effects on a Neural Network that are concurrent with the expected theoretical ones. Their testing bench was a Convolutional Neural Network trained with Stochastic Gradient Descent using momentum. The Network was trained on the ImageNet classification task [67], having only one fully connected layer to output the class of the recognized object. Namely, they increased learning rate, removed Dropout, accelerated the learning rate decay, and many other things that can harm the training stability and efficiency. However, their model with Batch Normalization was able to outshine their standard benchmark model with the above mentioned features. More specifically, they noticed that even increasing the learning rate helped with accelerate the training times and then increase the model's accuracy. To conclude, Batch Normalization is a helpful mechanism to implement in the Neural Network as it can increase the training efficiency of the model.

Hui et al. [1] proposed the use of a Batch Normalization layer with ReLU instead of a pooling layer between the convolutional layers of the model. Studying the Ioffe and Szegedy study on Batch Normalization shines light on the reasons that made them choose Batch Normalization instead of Pooling. The promise of achieving better performance while maintaining the airfoil family's statistical properties sounds very promising. Sadly enough they did not check whether the same performance would be achieved by using pooling layers between convolutions, thus studying whether Batch Normalization has any advantages in our case over standard pooling. As the purpose of this thesis revolves around studying the ability of the Neural Network to approximate the physical phenomena rather than studying the properties of the Neural Network architecture, we will not study it as well.

Chapter 4

Coupling Machine Learning with Fluid Dynamics

4.1 Introduction to DL Algorithms in Fluid Mechanics applications

Having presented the physical problem and the Deep Learning basics will now commence introducing how to combine these two concepts to solve the fluid mechanics problem at hand. Using AI to solve Fluid Dynamics problems may sound odd at first, but there are numerous advantages in doing so. After all, that is the main point of this Diploma Thesis; how to use Convolutional Neural Networks to solve Fluid mechanics problems, focusing on estimating the Cp distribution around an airfoil during sonic flight.

As mentioned in Chapter 1, running CFD simulations requires significant CPU time, fast and efficient CFD programs, and large amounts of energy. An engineer's main task is to design an item that is optimal in one or more ways. That said, an airfoil's (or a hydrofoil's) designing procedure can be a difficult task, as finding an immediate optimal solution is, most of the time, very difficult. Nature, as analyzed before, is composed of various intricate systems that coexist in harmony, adding even more complexity to studying any natural phenomenon. For example, an airplane designed for hypersonic flight may experience issues during subsonic flight, even for short time amounts, because the airflow is significantly different during the two states. Problems like that impose the need to run multiple tests to confirm that a given design may optimally operate at service conditions while maintaining a semi-optimal or (at least) acceptable behavior throughout its spectrum of operation. However, even if the engineers "cut corners" and focus solely on designing a very efficient airfoil for a single mode of operation, there is no guarantee that it would be an easier task.

To get some solid data on how an airfoil would behave, engineers use sophisticated CFD programs that most of the time require an expert to operate them to return valid data. A CFD program is nothing more than an arithmetic solver that needs properly defined input to account for the airfoil's geometry, boundary conditions, turbulence modeling, fluid characteristics, and many other things. Nowadays, most contemporary state-of-the-art solvers come with integrated, sophisticated solutions to improve simulation parameter input, yet building a proper airfoil mesh remains a craft. To further illustrate this point, during this thesis, a total of 4 different meshes were used (one created by the author, two by the supervising Professor Mr. Papadakis, and one from a NASA paper [68]), either of them yielding different results, with varying degrees of accuracy -(fig. 4.5)-. All that work was for a single prototype design. Imagine now having to repeat that process over and over again for a plethora of slightly different designs. It becomes apparent that it can be a long and tedious process that hinders the optimization process. From my standpoint, there are no tools that can automatically build reliable meshes for use in CFD solvers without requiring an expert's touch. Instead, there are efforts to build tools that can automatically deform an existing mesh to account for relatively small geometrical differences between two different geometries [28].

However, solving the geometry input problem does not alleviate a CFD solver's principal deficiency. No matter how much geometrical input is optimized, there is no way to eliminate the fact that a CFD simulation remains a computationally heavy task. Although a mesh is built with the minimum possible elements to yield valid results, a CFD simulation will still require significant time and energy to compute. On the contrary, AI algorithms tend to be very fast and energy-efficient while running on the same piece of hardware as a CFD solver or solving another equally complex problem. Also, it is significant to emphasize that speed does not handicap the algorithm's accuracy, as in most cases, the algorithm's precision is at least adequate for the task at hand. Testament to these arguments is the following time, precision data gathered from different sources:

1. Hui et al.'s Convolutional Neural Network average prediction time per airfoil case 180 ms (Intel CPU) with mean Test Accuracy $\approx 96\%$ [1]

2. Open AI 5 managed to defeat Dota 2's [1] world reigning team (Team OG). Each timestep, or AI operation, was ≈ 133 ms, with the AI managing very complex input data and evaluating an even more complex set of actions [70].

Finally, as described in section 3.3, DL algorithms do not require a highly sophisticated input, unlike CFD solvers where the mesh is crucial to the simulation. A DL algorithm that approximates the solution of a CFD solver can take a plethora of different inputs. Some of them are:

1. Binary bitmap image where the value 1 corresponds to the airfoil's solid boundary and 0 to everything else [18].

2. Bitmap image that takes into consideration the Freestream Mach Number to colorize pixels while having a binary approach to describing the airfoil's boundary [17].

3. Bitmap image using Signed Distance Function (SDF) to colorize the image. This method can meticulously capture the airfoil's geometry and input precise data in the Neural Network while using a relatively small amount of pixels [1].

All these methods utilize Convolutional NN to handle input and some Fully Connected Layers to handle the output. Also, advanced MLPs are potential AI algorithms to solve this problem, the difference being shifting the input data to numerical rather than image data. However, studies show that, for this application, MLPs tend to be slower to train and less accurate than CNNs while being slightly faster to yield results [1]. In this thesis, following Hui et al. [1] paper, a CNN is used to estimate Cp data using an SDF image as input.

4.2 Software used in this Thesis

In this project, several programming languages were used. The neural network was developed using the Julia programming language, and its Flux module [71][72]. Julia language is an up-and-coming programming language

[1]Dota 2 is a multiplayer online battle arena (MOBA) videogame where 2 teams consisting of 5 players each, clash to destroy their enemies' base, whilst defending their own. Each player controls a character, with specific abilities and deficiencies, from a pool of 121 unique characters. Making things even more complicated, each team has partial vision of the map, fighting to establish better map control while depriving the enemy team information of their location at all times.[69]

that uses just-in-time (JIT) compilation and Dynamic typing. Julia is a high-level programming language similar to Python, yet due to JIT compilation, it can perform like C or C++. Also, Julia has native support for the NVIDIA™ GPUs, as it is written to facilitate easy use of their programming language CUDA, making Julia every efficient to write GPU executed code. Julia has a growing support for ATI™ GPUs as well. That said, Julia is very efficient in prototyping and writing easy-to-understand code without having any significant drawbacks in execution speed, making her very appealing to develop neural networks. Both Julia and Flux are actively maintained and developed by a large group of experts in Data science, computer science, and Academics, who are interested in the fields of computer science and data analysis. Among the many organizations that use and develop Julia and its packages are MIT, University of Cambridge, Microsoft, NASA, and many more, proving that it is a robust platform to develop software. Python was the other main language used to develop this project.

Python is a staple when it comes to General programming as it is both lightweight and efficient. Being a high-level programming language makes her very appealing to write small pieces of code that do various things ranging from simple text manipulation to creating large data sets. In this project, Python was used to randomly deform the RAE-2822 airfoil and generate the respective SDF images off its variants, using the popular and very useful NumPy [73], SciPy [74], and MatPlotLib [75] packages. Also, was utilized in various little scripts that did small tasks like manipulating the output of MaPFlow to create properly structured data files for the neural network, monitoring the computer resources use, during the CFD simulations, and sorting the airfoil variants' points for later use in automatically deforming the computational grid (Mesh).

Another programming language that was used during this project is Fortran98. MaPFlow and some other auxiliary programs, that found use during this project, are developed in Fortran98. Fortran98 is a robust and extremely fast, scientifically oriented programming language that may be old but is still relevant in the field of computer simulations because of its immense speed and efficiency. Last but not least, as the entire project ran on Ubuntu Linux, Bourne-Again-Shell (BASH) scripting language was used to make all the pieces of code work together in harmony. All the codes developed for this thesis are available on GitHub for everyone who wants to study them.

4.3 Convolutional Neural Network Architecture and Input Generation

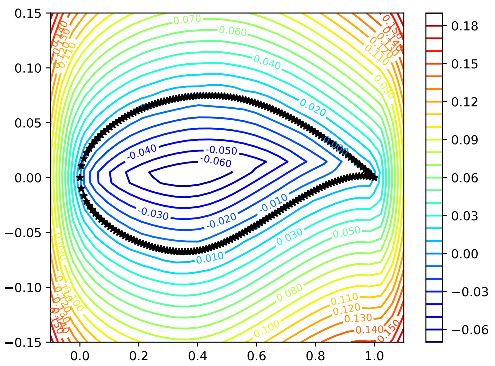
Based on Hui et al.'s work [1], a five-layer Convolutional Neural Network is used. This architecture is supposed to have the best accuracy during their experimentation with different Neural Networks solving this problem. This architecture consists of five convolutional layers, a Fully Connected Layer, and, of course, the output layer. The kernel size was constantly $5x5$ across all convolutional layers, while the padding, stride, and layer size changed between the layers according to the values on table 4.1. This combination of layers resulted in having a fully connected layer with 14400 nodes, which connect to the 49 output layer nodes. Between each convolution layer there exist a Batch Normalization layer, with ReLU as the activation function. Then two identical models were used to calculate the Cp distributions, one for the top and one for the bottom airfoil face. These models were trained and validated separately from each other, but they used the same input at all times.

In order to cut down training and testing times, the data and functions were translated to GPU related data and functions using Julia's CUDA.jl package [76]. The GPU is better suited for this kind of applications due to having more arithmetic units than a standard CPU. GPUs are dealing with image, video encoding and streaming that most of the time involve large matrix manipulation and linear algebra, compared to the general purpose load of a CPU which except from arithmetical operations deals with boolean operations, data registry manipulation, data IO, etc. That is why GPUs are loaded with a lot arithmetical operations units that work in parallel making them ideal for Neural Network Applications as the Neural Networks at their basis are consisted of enormous matrices.

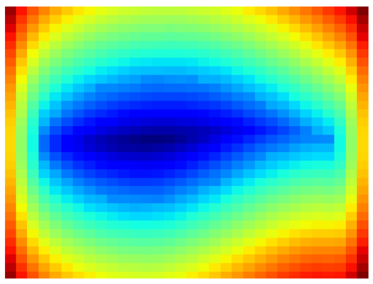
Table 4.1: Convolutional Neural Networks' Architecture

Convolutional Neural Network Architecture						
	input layer	padding	stride	kernel	output	Channels
Layer 1	32	2	1	5x5	32	20
Layer 2	32	1	1	5x5	30	40
Layer 3	30	1	1	5x5	28	60
Layer 4	28	1	1	5x5	26	80
Layer 5	26	1	2	5x5	12	100

Fully Connected Layer Nodes	14400
Output Layer Nodes	49



(a) An SDF function's contour plot



(b) An SDF image used for Neural Network's input (scaled)

Here are visible the iso-value curves
and the airfoil's boundary with black stars.

Figure 4.1: SDF function images

Mean Squared Error (MSE) was used for acquiring a measure of the algorithm's precision. ADAM algorithm was used for the algorithm's training, with learning rate of 10^{-4}. All of the above parameters and choice of activation function, optimizer,layer structure and so on, was after the model of Hui et al.. However, there were no data for each convolutional layer structure, namely the stride and padding values, and so there may be a divergence from the paper. They were discovered through trial and error, by comparing the resulting nodes of the fully connected layer to what their paper had proposed (14400) as a criteria of similarity between the two architectures.

Also, as proposed by Hui et al., the network was trained and tested using minibatch training. Minibatch training is a technique were the bulk of input data is split into smaller pieces (batches) before they are input to the Neural Network for training or validation. The minibatch training technique is relied on the observation that it is better to use less data and get a rougher estimate of the algorithm's gradient in less time than using all of the data at once and get a precise result. The modern training algorithms' converging characteristics are such that see no additional benefit from a precise result of the Network's gradient as even with less data they are rapidly converging [53]. As a result, we can use less data per batch, then update the Network through the optimizer function and repeat the process for the rest of the batches per epoch. This way we cut down total training time as each batch that is x times smaller than the total bulk takes x times less time to compute, while updating x times more the Network. For the task at hand, a minibatch of 50 variants size was chosen, similar to the Hui et al. paper.

SDF formatted images, as in the Hui et al. paper, were used to inform the network of the airfoil's geometry. SDF formatted images are very convenient for modeling an airfoil's geometry as it can capture more detail in fewer pixels than a simple image. Signed Distance Function is used in many ways in the field of computer graphics, some examples being font scaling [77], and in Computer Vision to get a better perception of depth [78].

Let ω be a subset of a metric space, X, with metric, d then the signed distance function, f, is defined by:

$$f(x) = \begin{cases} d(x, \partial\Omega) & if\ x \in \Omega \\ -d(x, \partial\Omega) & if\ x \in \Omega^c \end{cases} \tag{4.1}$$

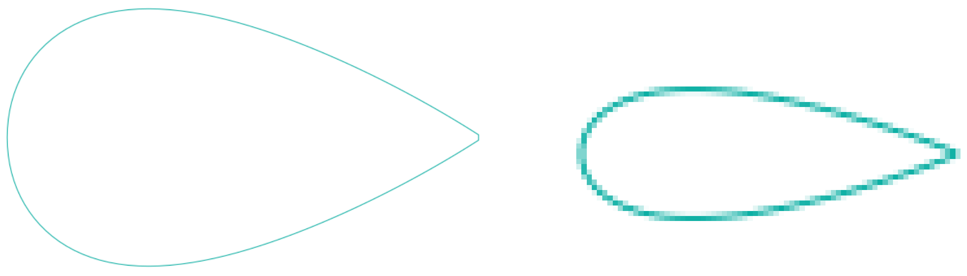
where $\partial\Omega$ denotes the boundary of Ω. For any $x \in \Omega$:

$$d(x, \partial\Omega) := \inf_{y \in \partial\Omega} d(x, y)$$

where inf denotes infimum.

Signed Distance Function (SDF) creates a surface, using a definite grid, where every node is assigned a value equal to the minimum distance of the node to the geometry's boundary. From a rather simplistic geometric standpoint, in every grid point, we calculate a circle tangent to the geometry's boundary, as shown in figure 4.4, whose radius is ultimately SDF's value at that node. Then whether the node is inside or outside the geometry, this distance is signed positive or negative. For this thesis, a node inside the airfoil will have a negative distance from the boundary, and a node outside the airfoil will have a positive. Using SDF, we can create an image packed with detail by simply mapping a color value to a distance value. Unlike SDF images, when pixels are colored with a single color to digitize geometry, it can result in accuracy loss the lower the image resolution is. In low-resolution images, it is common to get "staircase" edges as there are not enough pixels to create the illusion of curves, as seen in fig. 4.2. As a CNN perceives images in a human-like manner, using a low-resolution binary image can handicap the algorithm's precision, having less geometric information. A solution is to raise the image's resolution, thus increasing image data volume, resulting in higher execution and training times. On the other hand, using SDF, every pixel gets geometric information, thus utilizing the entirety of the image data available. This solution increases the entire process's accuracy while reducing data size, and execution, training times.

As said earlier, two different networks are trained; one responsible for predicting the Cp distribution on the top side and one for the bottom. So



(a) Image resolution: 640x360 px (b) Image resolution: 100x50 px

Figure 4.2: NACA-0012 airfoil bitmap image under different resolutions
Here are presented two different images that depict the same airfoil under different resolutions. The right image has a resolution of 100x50 px, while the left image 640x360 px. In the right image the staircase phenomenon is more intense as there are less pixels to capture the geometrical information compared to the right image. That has an effect the image to seem blurry and deprive as from further significant details.

Table 4.2: X axis points intervals

Start Position	End Position (not included)	Interval
0	0.1	0.01
0.1	0.4	0.05
0.4	0.6	0.01
0.6	0.95	0.05
0.95	1	0.01
1	-	0.01

there are required 49 Cp values for the top and 49 for the bottom face. Hui et al. suggested a point sequence to get these values, presented in table: 4.2. Studying the point interval they suggest, we notice that it is more dense at the leading and trailing edges, as well as the area $x \in [0.4, 0.6]$, as this the area were a sonic wave will probably form. MaPFlow provides the entire flow field and the Cp distribution at all mesh's boundary faces' centroids. So a Python script (*Cp_ export.py*) was used to search for the points of interest among all of the available and then appropriately format them to simple text files. Then the neural network was able to load per variant both its image and the Cp values for its respective face. As mentioned earlier, the airfoil variants are products of random homogeneous deformation of the father airfoil. Firstly, are extracted two spline curves from the RAE-2822, one for the upper and one for the bottom side. Then we can extract from the spline its knot-vector and have access to the original lines control nodes.

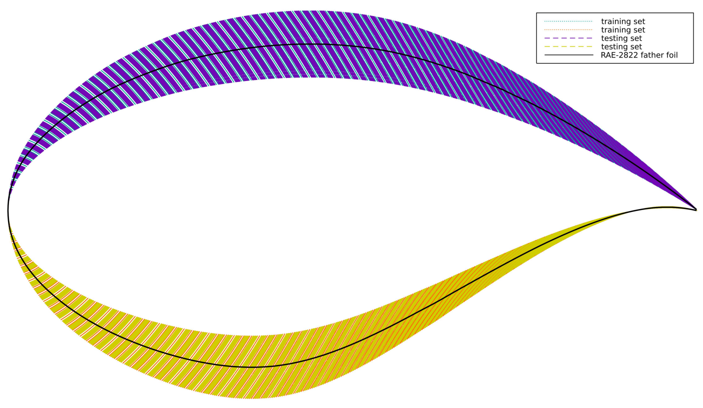


Figure 4.3: The entirety of airfoils plotted in one Figure

By multiplying the control nodes y- coordinate with the random deformation rate, we can shift all the control points up or down and deform the airfoil on the lateral axis. This approach may seem simplistic over other deformation algorithms, yet it is sufficient for this project.

The random coefficients used to deform the control points of the spline were generated using the Latin Hypercube Sampling algorithm, as Hui et al. suggested. Latin hypercube sampling algorithm is a random number generation algorithm that produces random values with respect to the previously generated values, thus minimizing overlaps [79]. In other words, by using the LHS algorithm, we minimize the risk of creating lots of duplicate airfoil variants. Another measure towards this direction is the use of different sampling procedures for the top and the bottom face. This is done to further increase randomness in the data generation process, improving the dataset's efficiency in training the network. Finally, it was chosen to have a relatively moderate spectrum of deformation ranging from -20% to 20% lateral deformation.

After the creation of the father airfoil variants, the following steps were to create a data file with the variant's coordinates as long as their respective SDF images. This procedure took place along with the generation of the Database structure, both for the training and the testing set.

4.4 CFD setup

Acquiring the Cp distribution data for each airfoil was the next problem at hand. Firstly a mesh of the father foil, RAE-2822, was created. As already mentioned in section 4.1 this is a tedious process that required expertise to yield a sufficient mesh. Many attempts were done to create a mesh, resulting

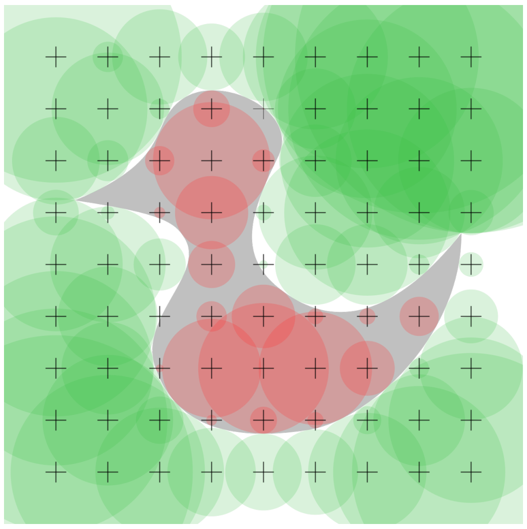


Figure 4.4: 2D Signed Distance Function transformed image
This is an image visualizing a discrete 2D signed distance field (SDF). The original represented shape (duck) is in grey. Crosses mark pixel centers where SDF values are stored. From each pixel center a circles expands that visualizes the value stored at the pixel. The radius represents magnitude, the color represents a sign (red is negative, inside the shape; green is positive, outside the shape). Unification of green areas define the set of points that are definitely outside the shape, the unification of red ones define the set of points that are definitely inside. We can see that although the SDF representation may be better than classical raster image at this resolution, with a limited number of pixels in the raster the image is still represented imperfectly, with areas that are neither green nor red, defining a set of points whose belonging to the shape is unknown and for which a separate algorithm has to be used to estimate this. The picture also shows how more pixels further away from the shape improve accuracy of the representation. [80]

in various meshes being used, with varying degrees of accuracy. As we can see in figure 4.5 out of the three available meshes only the two of them were close to the experimental data. The Mesh #1 was created by the author, the Mesh #2 from Mr. Papadakis, and the NASA was created from NASA laboratories [68]. Experimental data from a Berlin TU's experiment on an RAE-2822's performance during the same conditions were used to set a comparison reference between our meshes [27]. In that experiment, researchers wanted to test various turbulence models against data from a wind-tunnel experiment, where the flow conditions were similar to our own. These data were close to what Hui et al. proposed as the Cp distribution of the RAE-2822 airfoil from their simulations. So having crosschecked these data, we can assert that they are the ground truth to compare our meshes against them.

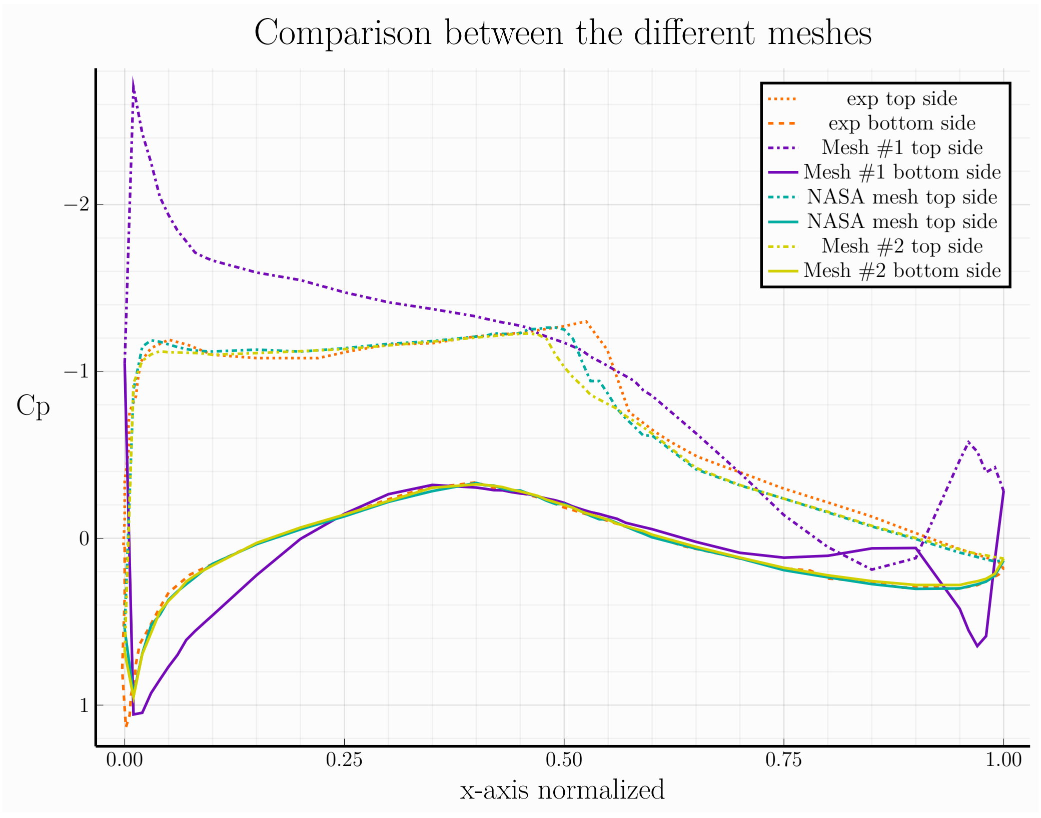


Figure 4.5: Comparison between the different meshes' accuracy

Observing the figure 4.5, we can notice various things regarding the available meshes. First of all, it becomes apparent that creating a decent mesh is something not trivial. Out of the three meshes, the one created by the person of least experience deviated the most from the baseline. Even though all experts' guidelines were followed, it becomes apparent that there was still room for error. This further solidifies the argument about the required expertise to operate CFD software.

Having said that, even the experts' meshes have some minor divergence from the experimental data. As at the time, we were more interested in testing whether the AI algorithm proposed by Hui et al. was capable of accurately predicting the Cp distribution, rather than setting a very precise simulation, so we opted to use NASA Mesh as it was closer to the experimental data. This behavior is reasonable shall we take into consideration the fact that we are more concerned about the algorithm accurately predicting the target Cp distribution than whether she is accurate herself. Even with the not-precise mesh, we got the non-linear phenomena that concern us, but not to the right extent. In figures 4.6a, 4.6b we can even observe the local relative difference from ground truth. Another factor that may contribute to not getting a more accurate Cp distribution may be the turbulence model used. However, like the mesh inaccuracies, this is not significant as the Neural Network is tested against the solver's data, not against the natural phenomenon.

After creating the airfoil mesh, the need to automate geometrical input to facilitate the data creation process became apparent. As said earlier, the author is not aware of any tools that can automatically generate high fidelity airfoil meshes, while creating 1500 different meshes for each variant is simply absurd. There may not be an automatic mesh builder, yet there are tools that can automatically deform a mesh to account for relatively small geometrical differences. Such a tool was developed for use in MaPFlow by Mr. Papadakis [28]. This diploma thesis presents an algorithm that can deform a grid given proper data sets for the initial geometry, that the mesh is built for, and the target geometry. Then by comparing the original geometry's points with the target geometry's points, the algorithm can map the target geometry's points to the grid's existing geometry points. Then it uses the relative difference of these points to deform the already existing mesh structure without altering its very core. Tools like this make it possible to fast prototype and test new designs, without spending lots of time creating and validating new meshes.

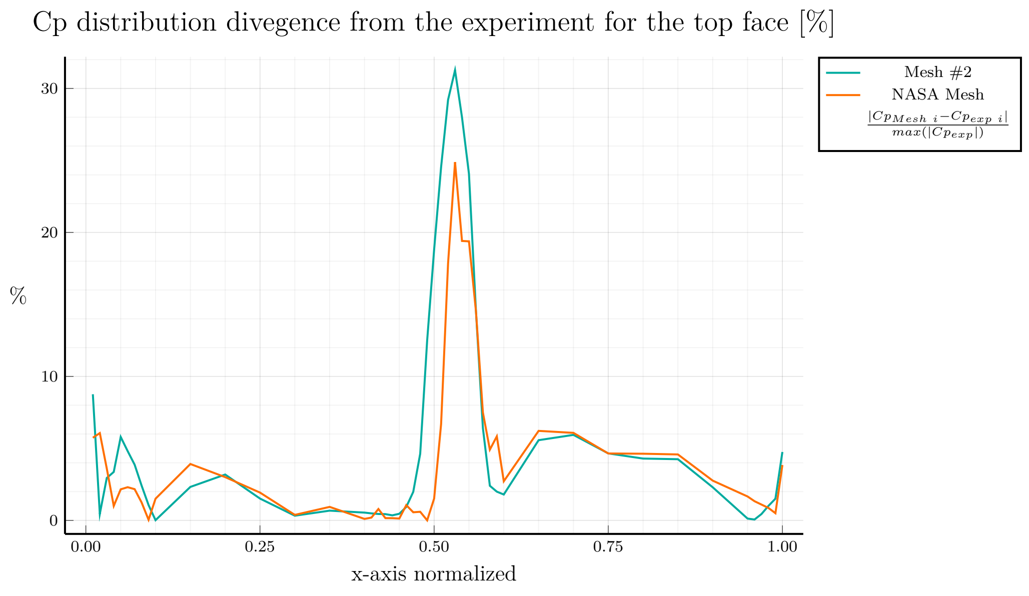
For this algorithm to work, some files that contain geometrical data are required. First, as the mesh is unstructured we have no clear indication of the nodes and boundary points order. Also, the solver utilizes finite volumes to solve the simulation, which creates the problem of duplicate points. Duplicates exist because a volume's perimeter points are overlapping with its adjacent volume ones. By eliminating duplicates and then properly sorting the points, we get a map that correlates each point to the respective volume's boundary. We can also create each face as a spline, during the dataset creation process. Splines make the deformation and SDF image creation process streamlined as the geometry is well defined. Working with an array of unsorted, duplicate points can be troublesome, especially in troubleshooting problems where the deformation failed or there was a logical error, resulting in geometry that is not smooth or continuous. Then by comparing these coordinates to the target ones, we can relate the new coordinates to existing mesh entities. Finally, the deformation algorithm resolves the geometrical relationship between new and existing points and deforms the grid accordingly.

Most of the work is handled by the grid deformation algorithm [28], yet creating the proper data to feed the algorithm is done separately. The geometrical boundary points of the original mesh are taken after a solver's complete simulation, being part of its output. Then a Python script (*sort.py*) excludes duplicates and sorts these points. On the other hand, in parallel with the creation of SDF images for each variant, a data file holding the airfoil geometry's points' coordinates is created. Then before resolving each variant, MaPFlow reads these two files (the original mesh's sorted points and the respective variant's file), deforms the mesh, and solves the simulation.

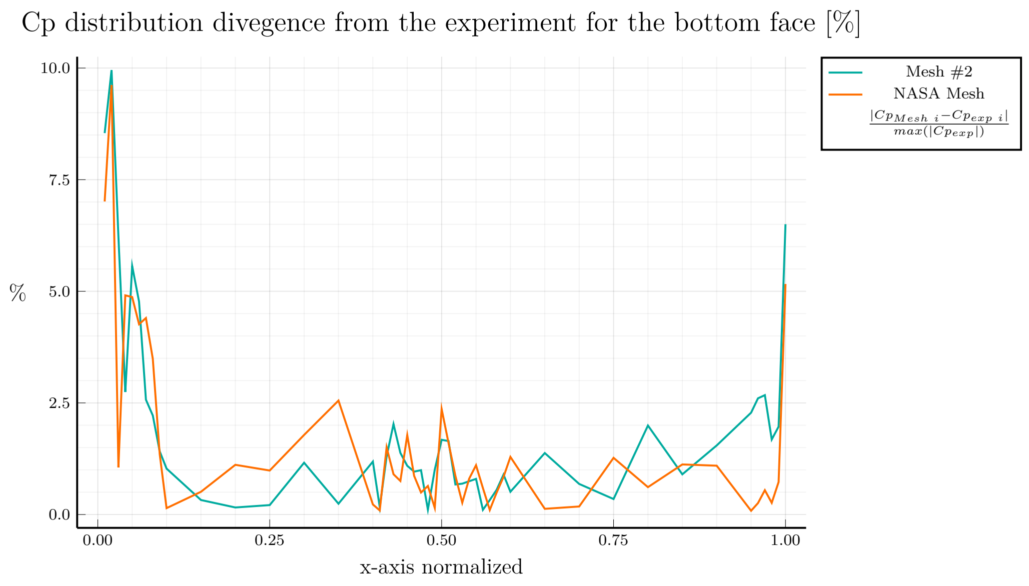
Last but not least, each simulation completed 5000 iterations. Not knowing the exact number of iterations for the solver to converge to zero[2], initially, we set 30000 iterations to study its convergence. As seen in figure: 4.7b the solver converges at around 20 thousand with a maximum absolute error of 0.03%. However, 20 thousand iterations require almost 27 minutes to calculate on the available hardware [3] which is fairly long for a single case, considering the bulk of cases. Instead, we opted to run the solver for 5 thousand iterations that took approximately four times less time per case. As seen in figure: 4.7a 5 thousand iterations were significantly less accurate than 15 thousand, yet the time saved was more beneficial than the extra precision. After all, there is already enough divergence from the experimental data due to the mesh structure and not choosing an appropriate turbulence model, for a maximum absolute error of 1.46% to impose a greater error to the solution.

[2]For a computer program, convergence to zero is impossible to achieve due to floating numbers precision. Here it is referred from a clear mathematical standpoint, parallelizing the solver's convergence to an infinite series result.

[3]Hardware and software characteristics are listed on Appendix A. For quick reference, the available CPU was an Intel™ I7-6700k running at 4 GHz utilizing 7 out of 8 cores.

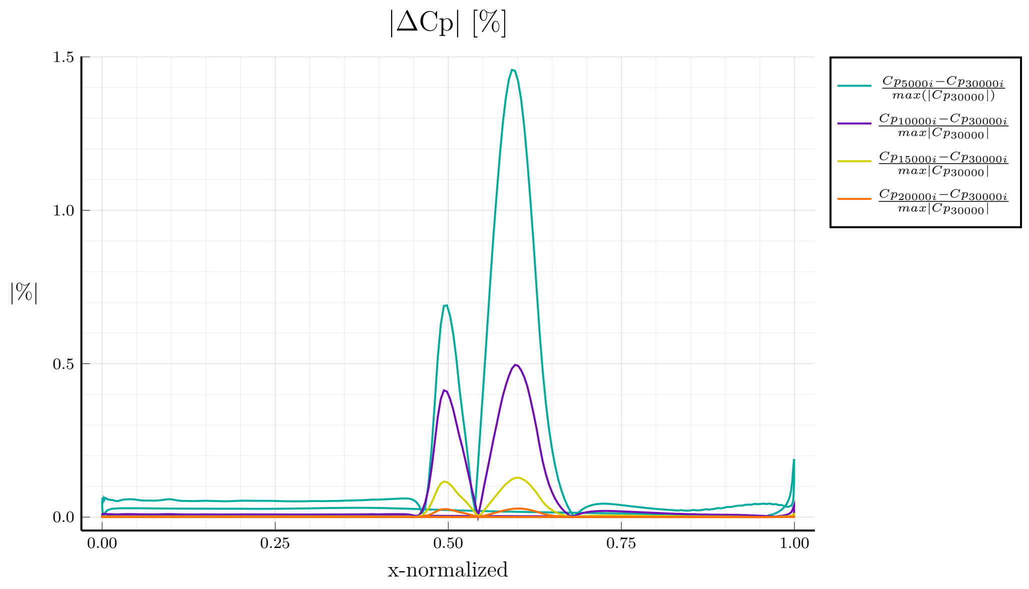


(a) Cp difference for the top face

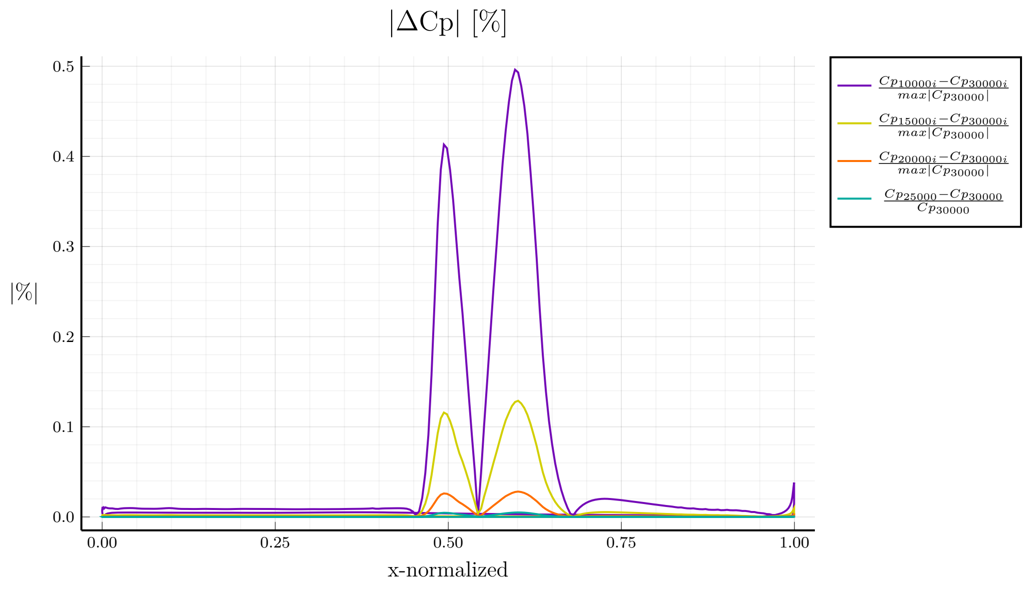


(b) Cp difference for the bottom face

Figure 4.6: Cp divergence for each of the two faces



(a) Iteration spectrum : {5000, 10000, 15000, 20000}



(b) Iteration spectrum : {10000, 15000, 20000, 25000}

Figure 4.7: Cp divergence [%] at ranging iteration cycles

4.5 Project Architecture

Having presented the different parts of this project, they will be put in order, like puzzle pieces, to complete the project pipeline. Writing a single piece of code to conduct the entire pipeline was a mandatory step to continue. As the project consisted of many smaller or bigger programs written in various programming languages, it was considered optimal to bind them together with a terminal script. As already mentioned in section 4.2, the project ran on Ubuntu Linux, and the terminal of choice was BASH. The conducting script (*cfd_pip.sh*) was developed to implement as lines of code the following analysis.

First of all, after creating the mesh using BETA CAE™ ANSA and exporting it for use in MaPFlow, the simulation of the original RAE-2822 takes place. Of course, for the original airfoil, there is no need to use the deformation algorithm. Then MaPFlow calculates the simulation and exports the results in a binary formatted file, whose contents are then transformed to ASCII text by an auxiliary program that comes with MaPFlow *(*2dcp.f90). However, the data needs to be formatted as Cp values at certain x-axis intervals for each airfoil's face. For this purpose, was developed a script that handles this task (*CP_export.py*). Also, during that step, the coordinates at the airfoil's geometrical boundary of the mesh are exported in a separate ASCII text file. This file is then input into the sorting program, -that was presented in the previous section- to sort the points and eliminate the duplicates.

After obtaining the sorted points file, the dataset generation process takes place. Then the sorted points are inserted in the Dataset Generation program (*Airfoil_DataSet_Generator_Randomizer.py*), where the airfoil faces are deformed, and their respective SDF images are created, as presented in section 4.3. The dataset generation program also creates the proper folder structure, creating separate folders for the training and testing datasets and then organizing the geometric, and image data for each variant in separate folders inside them. Then the conducting script places the other prerequisite files for the CFD simulation to execute, namely the simulation parameters input, the original mesh file, and the original airfoil's sorted points file.

After the database is complete, the CFD simulations commence. The database created by the Dataset Generation program is indexed, and the paths to each variant are documented in a text file. Also, a file is created to save the project's execution status, enabling us to pause and resume the project whenever the situation demands it. Then the conducting script reads these two files, compares their contents, and runs the CFD simulation of every variant that is not documented in the save-file.

The deformed airfoils are handled by the conducting script a bit differently than the original airfoil. As they are already variants, there is no need to save nor sort their boundary mesh points, and, of course, to recreate the database after each of their simulations. However, they require the use of the Mesh Deformation algorithm, as presented in the previous section. So their geometry data along with the original airfoil's sorted geometry data are input to the Deformation algorithm. Then the algorithm deforms the original mesh and inputs it to the solver that executes the simulation. Finally, similar to the original airfoil's run, the binary data from MaPFlow are initially translated to ASCII text and then to the formatted files for the Neural Network.

Ultimately, all these data created are used to train the Neural Network. Convenience sake, a single Julia Jupyter Notebook was created to facilitate all the Neural Network related functions (*Diploma Thesis Neural Net Notebook.ipynb*). Specifically, the Notebook contains the code responsible for data input, the Neural Networks' architecture, the loss and accuracy functions, the optimizer function used to train them, and the training function with a training loop for each Network. In the notebook are also included some minor functions that deal with data output, testing single cases and storing-loading the neural network. Data output is done through graphs were the respective Network's training and testing accuracy is logged for every epoch. The time required to complete a single epoch of training and validation per Network is logged for every epoch, as well.

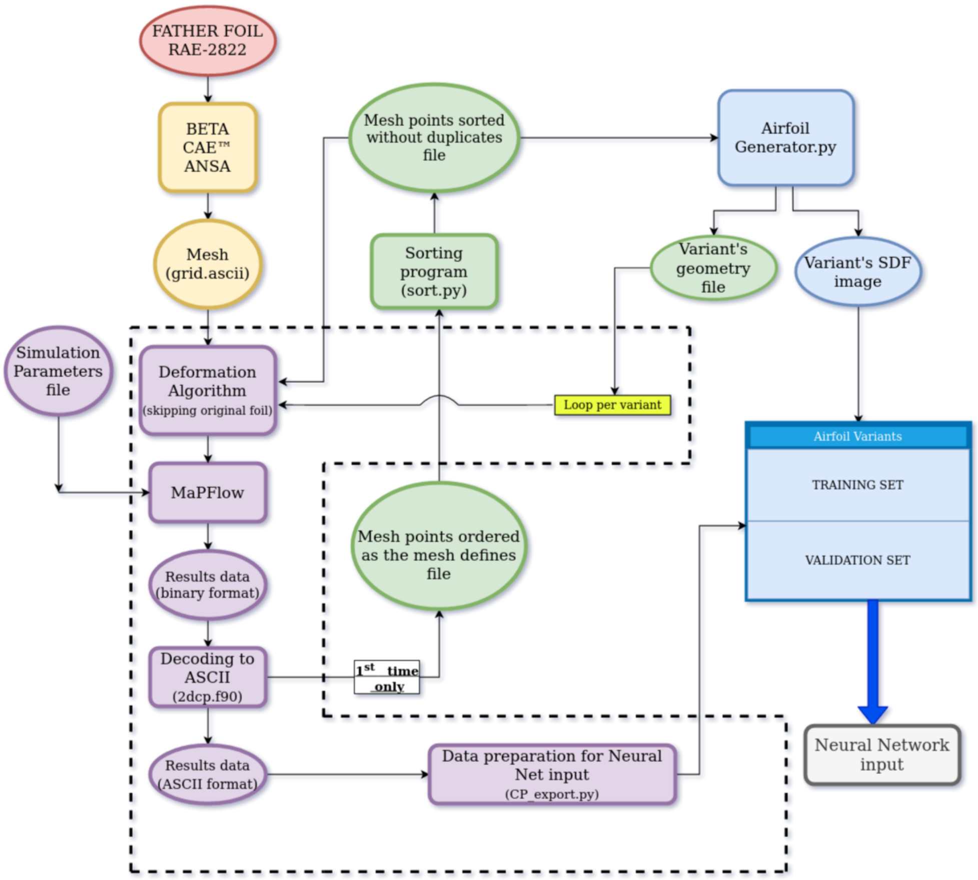

FATHER FOIL RAE-2822
BETA CAE™ ANSA
Mesh (grid.ascii)
Simulation Parameters file
Deformation Algorithm (skipping original foil)
MaPFlow
Results data (binary format)
Decoding to ASCII (2dcp.f90)
Results data (ASCII format)
Mesh points sorted without duplicates file
Sorting program (sort.py)
Mesh points ordered as the mesh defines file
1st time only
Data preparation for Neural Net input (CP_export.py)
Airfoil Generator.py
Variant's geometry file
Variant's SDF image
Loop per variant
Airfoil Variants
TRAINING SET
VALIDATION SET
Neural Network input

Figure 4.8: Project Pipeline Chart

Chapter 5

Results and further experimentation

5.1 Initial accuracy results

Having analyzed the underlying mechanics of Artificial Intelligence and Computational Fluid Dynamics and how these two concepts can coexist in this project, it is high time the results of this project were presented. There are many things to discuss but, first, we must address "the elephant in the room": " Does the concept actually works? ". The definite answer is yes, it does. As seen in figure: 5.3 both Neural Networks are rapidly reaching a training MSE of $\approx 10^{-4}$ in 1000 Epochs while converging to a final MSE of $\approx 2 \cdot 10^{-5}$ for the top side and $\approx 10^{-5}$ for the bottom side. Also, it should be noted that their testing accuracy is very well behaved, as well, as seen again in the figure: 5.3, following closely their training MSE. This is to expected though as the testing data are not completely new to the algorithms, but more on that in section 5.2.

Also, what is important to notice is the impact that the non-linear phenomena have on the algorithm's convergence. As seen in figure : 5.3, the top side's Neural Network MSE after 1000 Epochs oscillates around $\approx 10^{-4}$ for at least 2000 Epochs while converging to its final value close to 4000 Epochs. However, is highly probable that it would continue to oscillate have we let it train for more Epochs. On the other hand, the bottom side's precision was steadily increasing as the MSE was lessening every Epoch hitting a absolute minimum of $\approx 1.5 \cdot 10^{-6}$ for training while the verification error was $\approx 8.5 \cdot 10^{-7}$. However, after the minima the algorithm's errors converged to $\approx 10^{-5}$. This increase in MSE may actually be beneficial as the Epoch where the local minima emerged might have been the result of the algorithm over-fitting the data. Nevertheless, we will not know for sure as they probably have been a random incident caused by the network's random initial parameters and unfortunately the network was not saved during this exact state. This is not a problem as we are interested to see whether the proposed architecture can be accurate over our data without relying on the random occasion where the parameters where ideally initiated. To rephrase a bit, we want to prove that the model can be trained infinitely many times and converging to extremely low MSEs every single time. For ease of understanding how these oscillations affect the Prediction accuracy of the Networks the figure: 5.1 is presented, where the accuracy of the Networks during training and testing are plotted. The averages are taken at 1% of Epochs intervals to de-cluster the diagram. Observing the figure: 5.1, becomes immediately apparent that the accuracy converges to $> 98\%$ very quickly, further proving that the oscillations of $O(1e-4)$ and $O(1e-5)$ are not handicapping the Networks' performance at all. Finally, in figure: 5.2 we can also observe how the Cp distribution predicted by the Neural Networks compares to the experiment and the CFD simulation.

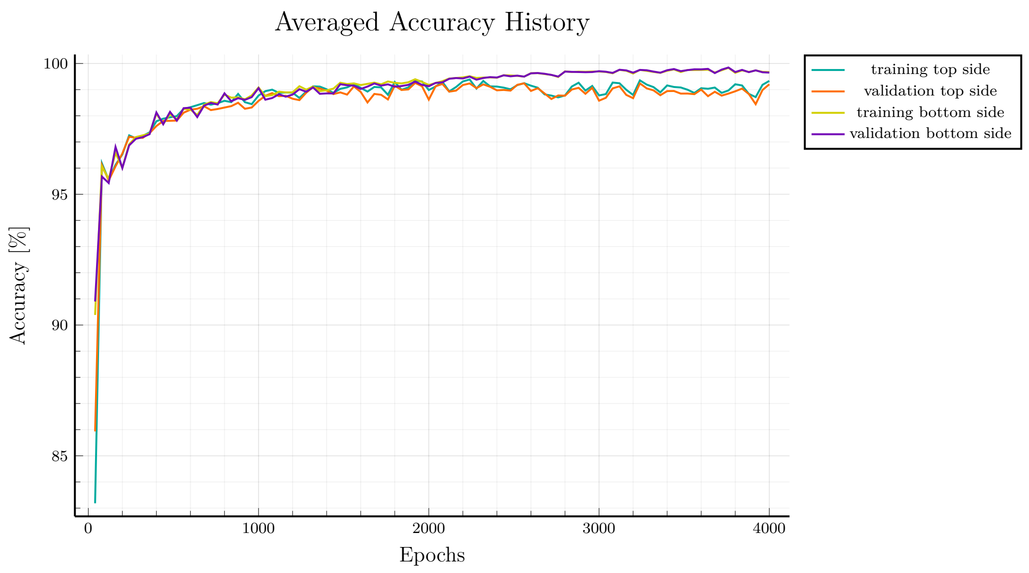


Figure 5.1: Accuracy convergence history per 1% of Epochs

The next main point of interest is whether there are any significant time gains over the standard CFD simulation. As seen in figure: 5.4 the Neural Networks' average training and testing time is ≈ 2.29 s per epoch per Network, totally adding up to 154.1 $mins$ for both the Neural Networks after 4000 epochs. Also, the average time to evaluate the original RAE-2822 airfoil

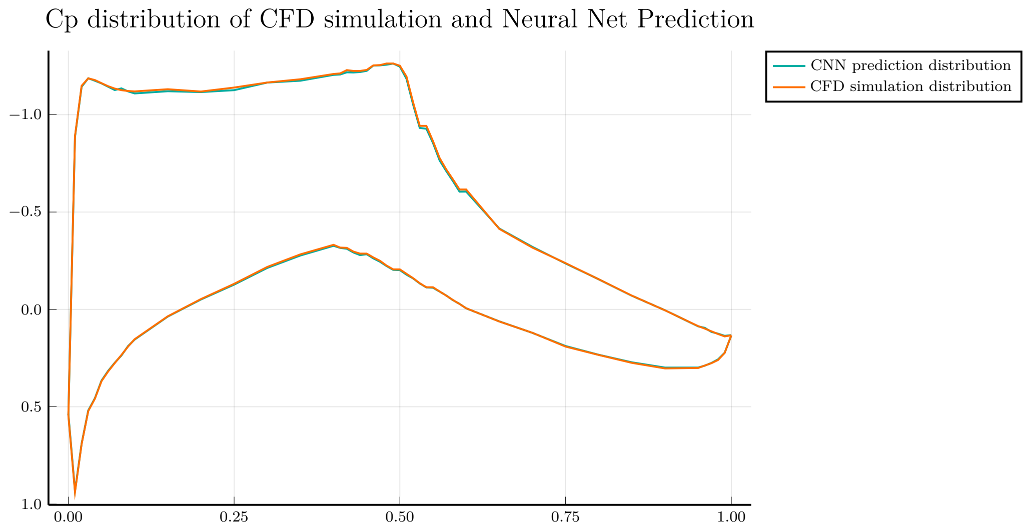


Figure 5.2: Cp distributions of CFD simulation and Neural Net Prediction for RAE-2822

and predict its Cp distribution is $\approx 5.4\ ms$ with MSE $\approx 4.8 \cdot 10^{-5}$ for the top side and $\approx 1.2 \cdot 10^{-5}$ for the bottom side. Even-though the accuracy may change for other cases the execution time probably wont so we have a very accurate estimate of the execution times. Comparing these numbers now to the CFD simulation times, fig.: 5.5 we observe that, on the same hardware, a single simulation lasts $\approx 5.5\ min$. While the time it takes to train and test the entirety of the Neural Network structure is 30 times greater than to evaluate a single airfoil, it is yet 50 times faster than running the entirety of the simulations. Also, we must bear in mind that the time to evaluate a single airfoil is insignificant compared to running a single CFD simulation case. There lies the true power and efficiency of our Neural Networks. As it was proven, the proposed Neural Networks require relatively moderate times to train and test them, while they are able to predict a single airfoil's Cp distribution in a fraction of a second.

In conclusion, Neural Networks are extremely fast to both train and use in singular cases while being sufficiently precise for the task at hand. So we can conclude that the work presented by Hui et al. [1] is accurate, and their proposed AI architecture is working as expected. That bears great significance as it opens the path for using AI in CFD simulation to quickly and accurately predict the pressure distribution around an airfoil, making prototyping a fast and efficient procedure.

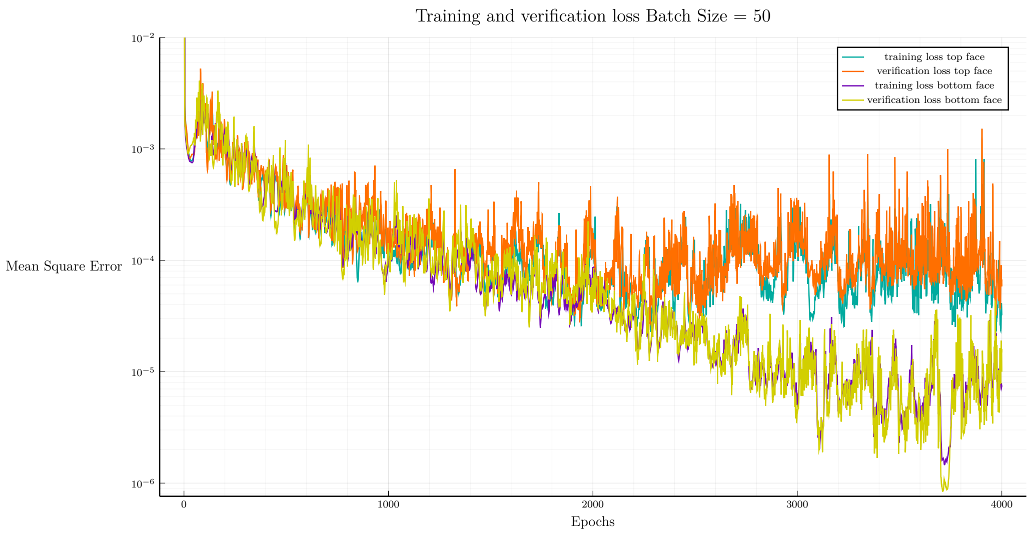


Figure 5.3: Convergence per Epoch for Batch Size of size 50

5.2 Batch size and Networks' flexibility experimentation

Having proven that the entire concept is fully functional there are some observations made that raised some questions. For example, observing the figure: 4.3 one thing becomes immediately apparent; both the training and the testing set have common geometrical boundaries as the airfoils of both sets were sampled randomly inside the range of $\pm 20\%$. Also, due to the number of variants in both sets is not impossible to encounter duplicates amongst the two sets. Is this a problem for our Networks' reliability? Have the Networks over-fitted the training data and because the validation dataset is quite similar to the training dataset we were not able to identify this problem ?

Another question that was raised was about the size of minibatches chosen by the authors. In theory, we know that splitting the data in minibatches is more beneficial for the algorithm's training speed as less data are involved in calculating the algorithm's gradient[53]. However, after the gradient's calculation we update the network by evaluating its loss function for the minibatch and updating the Network's parameters. So, is minibatch training more beneficial for speed or precision ?

Considering our Network Structure is successful in predicting the RAE-2822 Cp distribution and its variants, can we use to predict the Cp distribution for other airfoils with very similar geometry under the same flow con-

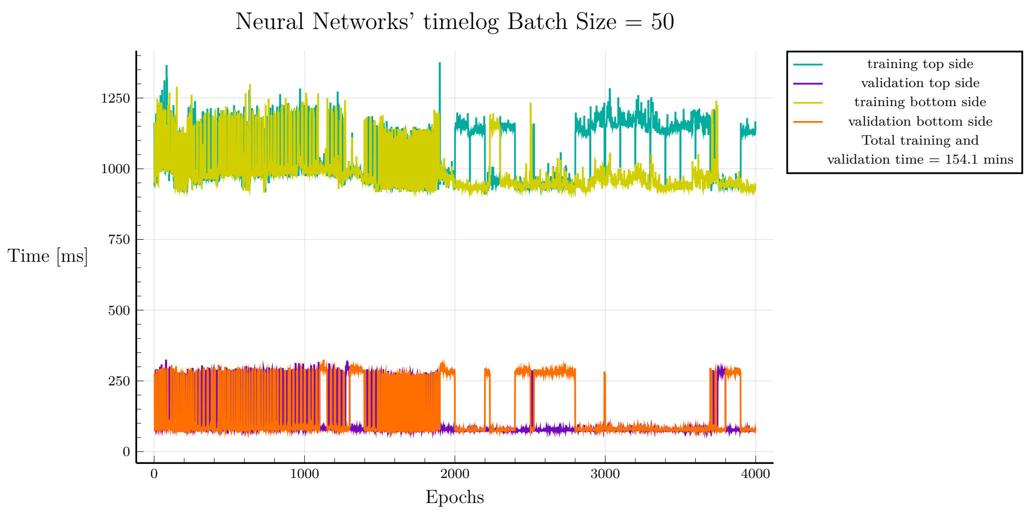


Figure 5.4: Time history per Epoch of training and validation for Batch Size of size 50

ditions ? For example, an airfoil similar to RAE-2822 is NASA SC(2)-0412. What would be our algorithms' precision in evaluating their Cp distributions?

During this section, we will focus on answering these questions and generally speaking researching the robustness and flexibility of our Networks. Initially, we will shed light on the effect of minibatch size as this is something closely related to the Networks' architecture. Then the focus will be shifted in identifying whether the Networks was over-fitted to the training data or not, by testing their performance on airfoils out of the original ±20% range.

5.2.1 Minibatch size investigation

As described in the section's intro, investigating how the minibatch size affects both the Epoch times and the algorithms' precision is quite interesting. Hui et al. have proposed a minibatch size of 50 items per batch. However, during an author's happy accident where the batch size was set to 200, significant time gains per Epoch were documented but with a penalty on precision. This accident raised the question of how the batch size affects precision and execution times; are the increased number of updates more computationally intense than simply evaluating the gradient of more data and what is the effect on precision. Hui et al.[1] did not disclose any information on how a batch size of 50 was selected.

To investigate the effects of minibatch size averaged-error graphs for each

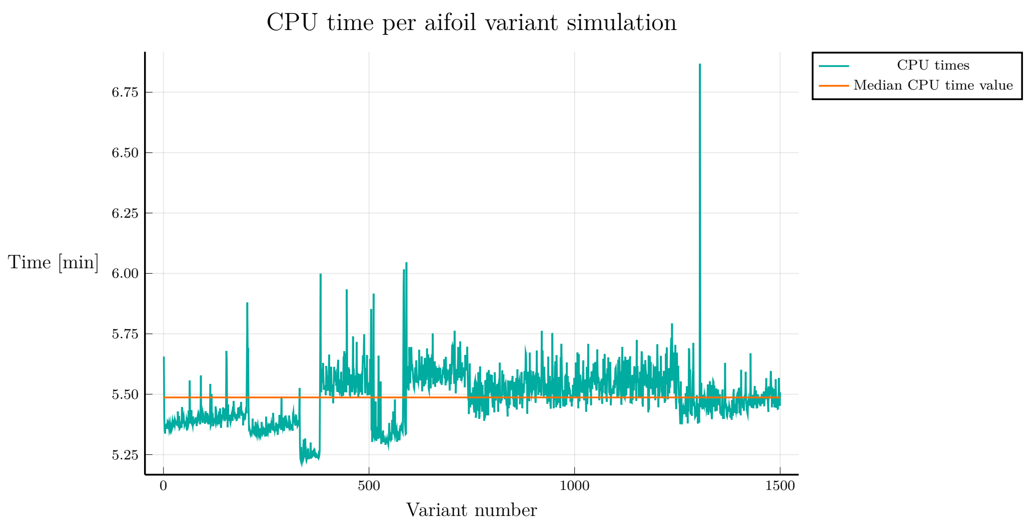
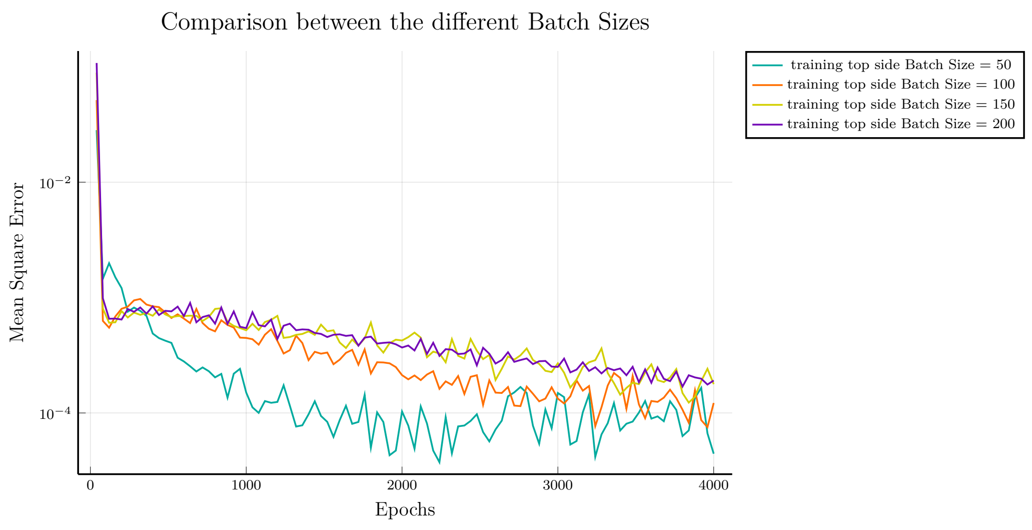


Figure 5.5: CFD execution times per airfoil case

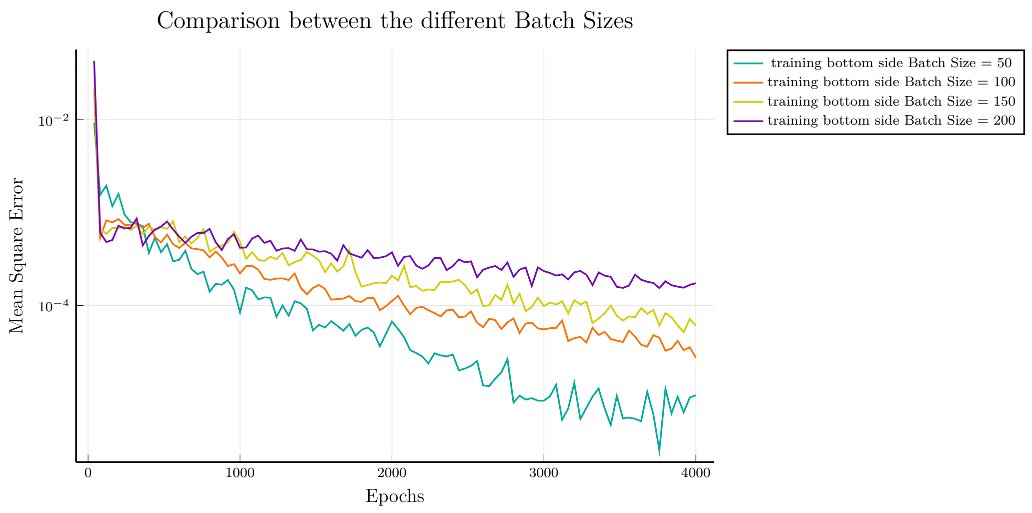
operation along with the time-Epoch graph were used. The averaged error graphs are used to de-cluster the clustered error-Epoch diagrams as we are more interested in comparing the Networks momentum towards convergence. It was chosen that the average would be taken every 1% of Epochs (40 Epochs). Observing the figures : 5.6a,5.6b we can notice some interesting things going on. Starting from the top side Neural Network we notice that with batch size 50 the error plummets at first yet it converges in a somewhat oscillatory behavior around 10^{-4}, something we have already seen in section 5.1. What is interesting is that when trained with the other batch sizes the convergence is almost linear (in the logarithmic scale) with some noise. Also, it should be noted that ultimately at 4000 Epochs they all share almost the same error except from batch size 50 curve.

Observing now the bottom side averaged error diagram becomes apparent that batch size 50 yields the optimal results. Even from the start of the training procedure at around 400 Epochs, the Network trained with batch size 50 converges pretty fast to very low error. In similar fashion, the batch size 100 Network from the 150 one, which in turn outperforms the 200 one. This may be a byproduct of the more linear and continuous (compared to the top side) field flow that develops around the bottom side. As the phenomena are more predictable and well-behaved, it is easier to train the Neural Network to predict them.

Having seen how the Networks trained with different batch size perform it is time to see how fast are they trained and check whether the small batch size leads to small training times. As we can see in figure : 5.7, contrast to



(a) Averaged Error Diagram for the top side NN



(b) Averaged Error Diagram for the bottom side NN

Figure 5.6: Averaged MSE diagrams for each NN

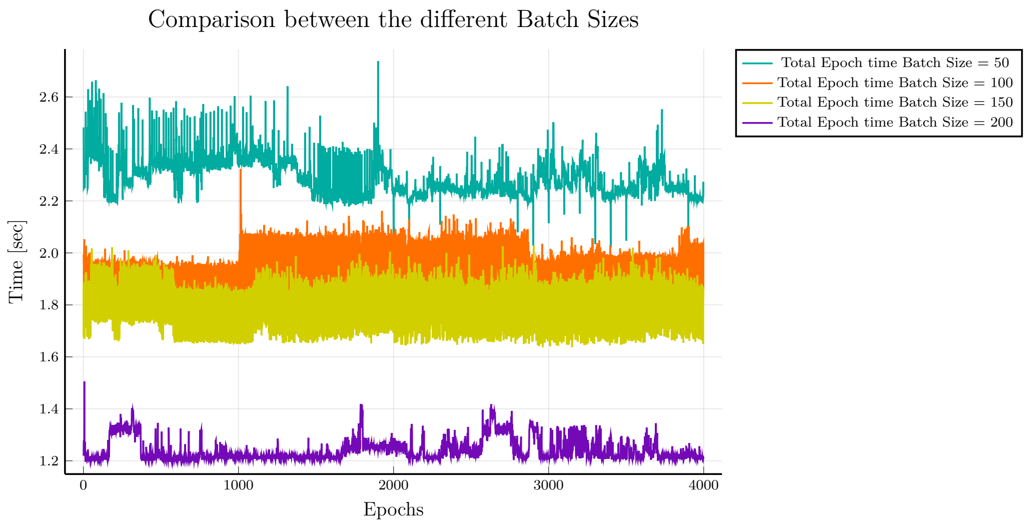


Figure 5.7: Time per Epoch for training with different batch size

Table 5.1: Times to train the Networks for different Batch Sizes

	Average Time per Epoch [sec]	Total Time to Train [min]
Batch Size 50	2.947	152.5
Batch Size 100	1.952	130.1
Batch Size 150	1.826	121.7
Batch Size 200	1.241	82.72

theory batch size 50 takes more time per Epoch out of all the other options. It is probable that due to how the Networks were structured as code that an Network update would be most computationally intense task than calculating the gradient for more data. However, we also notice that batch size 50 helped the algorithm have the best error characteristics out of all the other options. For reference, in table: 5.1 are the total times required to train the Networks along with the average time per Epoch. Batch size 200 took almost half the time required to train the network with batch size 50. It did so while maintaining an error close to 50's for the top side, and vastly different for the bottom side but with converging to a low error characteristics.

5.2.2 Investigating Neural Network's flexibility

As mentioned earlier, the Neural Networks are proven to predict the Cp distributions of the deformed airfoils with significant accuracy. Yet, it would be interesting to investigate whether the Networks have the capacity to "ex-

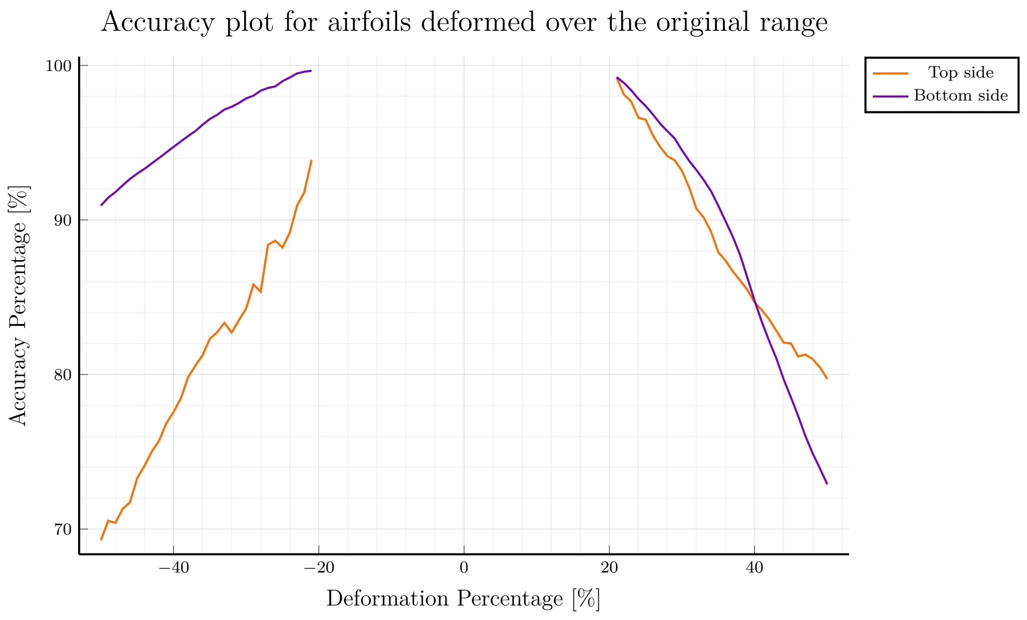


Figure 5.8: Accuracy of the Neural Networks for deformation rates out of ±20% used in training and validation

trapolate" and predict Cp distributions for airfoils out of the original ±20% range. There are many ways to test this, and for the sake of the current analysis we will create a linear spectrum of different deformation rates to deform both sides. The chosen range is ±50%, excluding the initial range of ±20%. The choice of the outer percentage bounds is not dictated by a specific problem that we may encounter, rather than stretching the Neural Networks' "imagination" to a significantly different deformation spectrum; a spectrum that is both wide enough to challenge them yet is not wide enough to imply the use of other types of foils. In another words, the choice was objective and somewhat based on intuition.

Observing the figure 5.8, it becomes apparent that the Neural Network is actually well trained and can extrapolate fairly well for specimens out of its training range. First, the Networks are well trained as their accuracy near the training edge is quite close to that of the testing procedure [1] and the fact that they have a buffer zone of having good accuracy ($> 90\%$) in predicting completely new foils is a sign of good training.

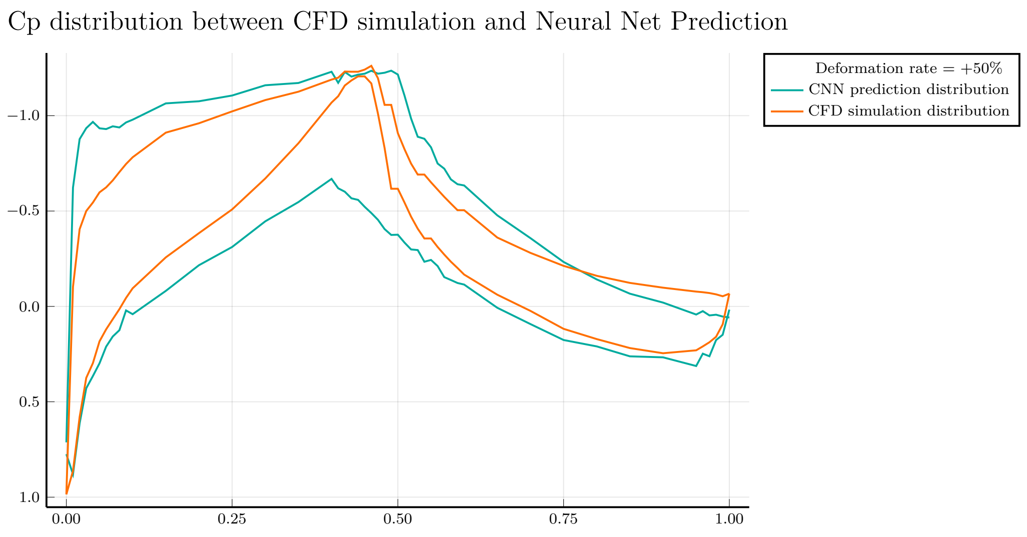
Second, we see that after that buffer zone that the two Networks have

[1]With the exception, of the top-side's Neural Network's accuracy for -20%. However, this does not constitute an anomaly as it is expected the predictions to have a slight variance.

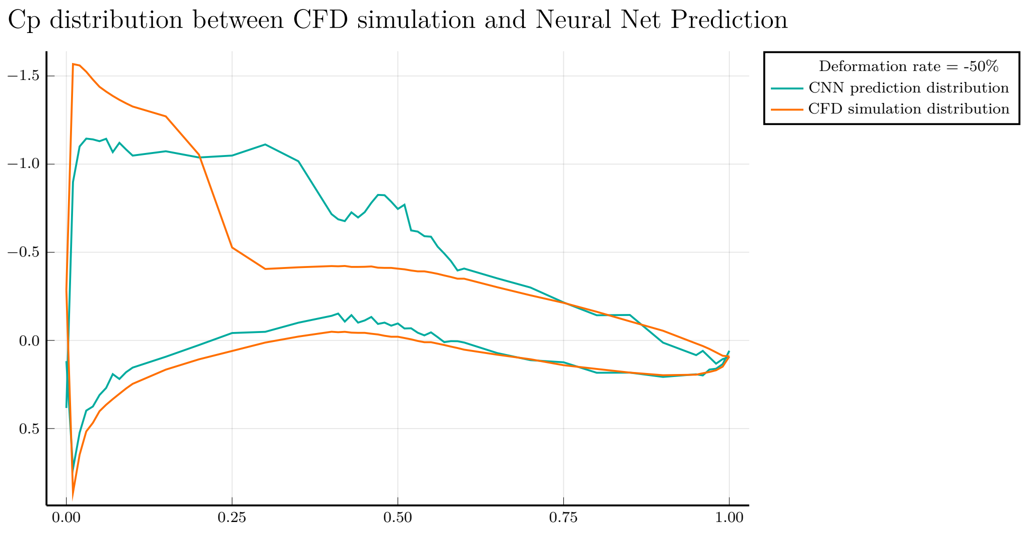
difference in how they behave. The best overall performance is exhibited by the Bottom side Neural Network. It has a slowly decaying accuracy in the $[-50\%, 20\%)$ range that manages to stay in the upside of 90%, while having a respectable accuracy up until $\approx +36\%$. These results are better than anticipated for this side proving that the Network is quite flexible and has definitely not over-fitted the training data.

On the other hand, the top side Neural Network has worse performance than the bottom side's. For deformation rates inside the $[-50\%, 20\%)$ range its performance is quite poor; rapidly decaying south of 90%. However, for deformation rates inside the $(-20\%, 50\%]$ range it holds its ground by having a similar decay to the bottom side's accuracy. Another difference between the two Networks is the fashion in which they lose their accuracy as the specimens further deviate from the original range. Contrast to the bottom side's smooth transition to less accuracy with somewhat linear segments, the top side's Neural Network has a more erratic behavior that may have to do with the fact that the top side faces the formation of shock waves and flow separation.

Observing how the Neural Networks predict the Cp distributions at -50% and 50% holds significant interest. First, observing the figure 5.9a where both Networks heavily underperform, what becomes immediately apparent is the fact the bottom side Network can not account for the flow separation happening due to the extreme alternating curvature of the bottom side, while not deviating much where the flow attaches again. Top side network still underperforms in a more consistent manner, contrast to the bottom side. Last, as seen in figure 5.9b, the bottom side performs pretty good and the top side Network can not accurately predict where the shock wave forms



(a) Deformation rate +50%



(b) Deformation rate -50%

Figure 5.9: Comparison between the CFD simulation and Neural Net Prediction for ±50% deformation rate

Chapter 6

Potential Applications

As already seen in the previous chapter, the Neural Networks are very reliable in accurately predicting the pressure distribution around airfoils. Especially, when the deformation rate is inside the training range they have stellar accuracy. Even though, the present Networks, developed for this Thesis, are a bit limited in terms of respecting changes in the flow's Mach and Reynolds number [1] they are still very useful for fast prototyping around these operation conditions. To further illustrate this point, a small application was created to demonstrate the efficiency of our Neural Networks.

6.1 A simplistic geometry optimizer

A simplistic optimizer was developed, using the openMDAO framework [81], to optimize the airfoil's geometry for the present flow conditions. OpenMDAO is an open-source framework for efficient multidisciplinary optimization, used to solve design problems involving coupled numerical models of complex engineering systems [81]. Using openMDAO is possible to make a computer model of the task at hand (ie. optimizing the airfoil design) and later to optimize the model by setting certain initial values and constraints. So, we could possibly try and bind openMDAO with our AI technology to create an incredibly fast optimization framework.

In order to, optimize the airfoil's design we have to initially specify the model(s) of the problem to be solved. During the present application a single explicit model was developed, whose inputs are the deformation rates for two

[1]Reminder that Mach and Reynolds numbers are kept constant along with the angle of attack, while training the Networks. However, with further research the author is optimistic that they can be integrated as dynamic data in the Networks as shown by [18], [17], [2]

airfoil's sides and its output is the Lift and Drag Coefficients. To be more specific, as it was not the main focus of this thesis, it was chosen to keep the design optimization fairly simple. To achieve that, the y-components of the airfoil are scaled with a simple coefficient r_i according to eq.: 6.1, unique for each side.

$$\hat{y} = (1 + r_i) * y_{RAE\ 2822}, \tag{6.1}$$

Then the new y-coordinates are passed to the image creation code to create an SDF image, that is then input to the Neural Networks to calculate the Pressure Distribution. Knowing the pressure distribution and the airfoil's geometry, we can integrate the pressure on the boundary geometry and obtain the Net Forces in each direction. What is important to notice is that only the Lift coefficient (the Vertical component of the Net Force) is of importance, as the Drag Coefficient is lacking the viscous terms and does not represent the reality of the physical problem[2].

After specifying the model, the driver of the problem is selected. As proposed by openMDAO's documentation, the SciPy Optimization Driver is a good choice when dealing with simple problems. The Driver contains among many things like Derivative Calculation, the optimization algorithm. For the present application, two of the most used optimization algorithms were used : COBYLA and SLSQP. COBYLA algorithm (Constrained Optimization By Linear Approximation) is a derivative-free optimization solver, developed by Michael J. Powell, that linearly approximates the problem using linear programming models [82]. On the other hand, SLSQP (Sequential Least SQuares Programming) minimizes a function of several variables with any combination of bounds, equality and inequality constraints. The SLSQP Optimization algorithm was originally implemented by Dieter Kraft and is ideal for mathematical problems for which the objective function and the constraints are twice continuously differentiable [83]. Both algorithms are proposed by the openMDAO documentation, yet COBYLA due to its lack of respect towards the model's derivatives is arguably better at exploring the design space than the SLSQP algorithm.

As minimizing the Coefficient of Drag is pointless, the optimizer targeted at maximizing the Coefficient of Lift, while constraining the allowable deformation of both sides inside the $[-20\%, +20\%]$ range. Also, both optimization

[2]Calculating the two forces via the Neural Network and comparing them to the forces calculated by MaPFlow, that problem becomes immediately apparent. Even-though, the Lift component is spot-on, the airfoil seems to create Thrust by having a negative Cd. MaPFlow accurately calculates the expected and logical positive Cd, pointing to the existence of Drag. As the Network predicts only pressure distribution, the Drag components that heavily rely on viscous terms are not calculated properly, so we will ignore them from here on.

Table 6.1: Optimization Results

	Default RAE-2822	COBYLA optimizer	SLSQP optimizer
r_{top}	0	-0.0453	-0.0257
r_{bottom}	0	-0.1997	-0.0113
C_L	0.7629	0.8077	0.7678
$T_{optimization}$ $[sec]$	-	48.9	51.5

methods were used in the code created to optimize the design (*optim.ipynb*) in order to measure their performance. Last the optimizations' results are presented in the table 6.1.

Studying the results the entire framework's efficiency and speed become apparent. Optimizing an airfoil in such short time is very important, considering that if we used the openMDAO optimization framework along with a classic CFD solver, like MaPFlow, the optimization would transition to minutes. Nevertheless, we must not forget that our Networks are limited by a specific operational range where their accuracy is relevant, compared to a CFD solver.

However, it is highly probable that using Neural Networks which are trained to take into consideration the $Mach$ and the Freestream *Re* we could use AI to create an optimal design for many operational conditions at the same time. This trivial application of the Neural Networks shows a glimpse of their potential to aid engineers in designing better and more efficient systems more efficiently.

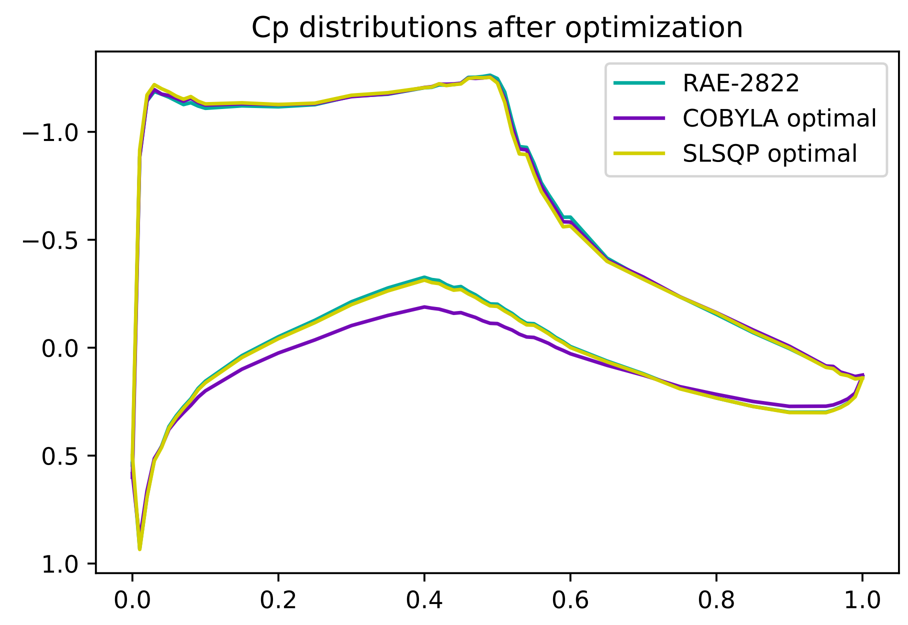


Figure 6.1: Cp distributions of the optimized airfoils

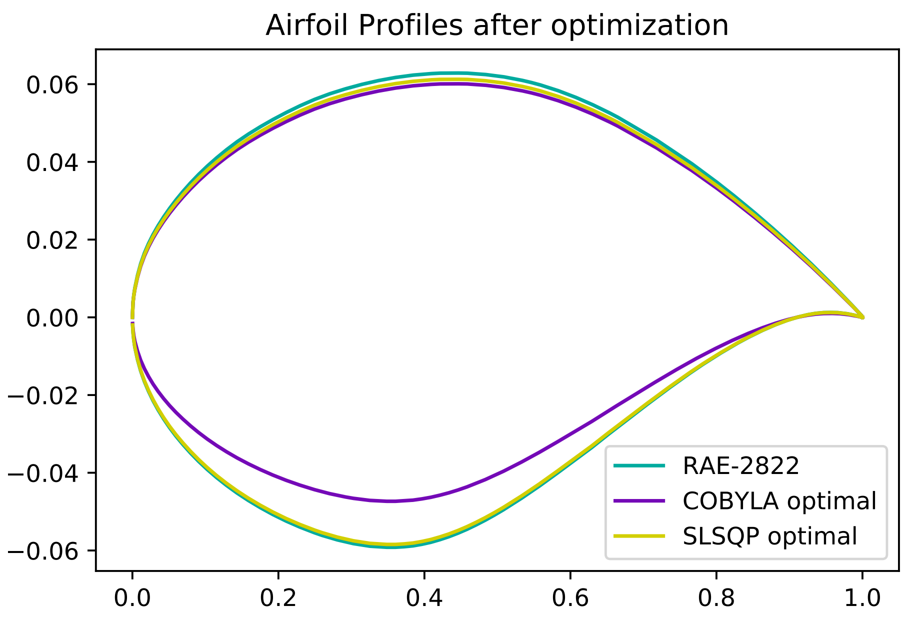


Figure 6.2: Profiles of the optimized airfoils

Chapter 7

Conclusions and Future Work

7.1 Conclusion

In this Thesis was investigated the use of Deep Learning techniques to rapidly predict the Pressure distribution along airfoils that operate under transonic flight speed, namely the RAE-2822 airfoil. Based on the work of Hui et al.[1], two Convolutional Neural Networks were trained to predict the Pressure distributions for the airfoil's top and bottom face respectively using SDF formatted pictures as input. The Neural Networks developed were 5 Convolutional layers and 2 Fully Connected layers deep. Between each Convolutional layer there is a Batch Normalization layer that helped with the statistical efficiency of the Neural Networks.

As there were no available proper training and testing data for the Neural Networks, they were created from the beginning. To begin, two computational grids (Meshs) were developed and one found in bibliography [68], to be used as an input to the CFD solver MaPFlow. All meshes were validated against experimental data [27], choosing the most accurate one.

Amongst the many capabilities of MaPFlow is the ability to automatically deform the computational grid given a file containing the exact geometric differences. That enabled us to automate the process of solving the CFD simulations for the training-testing variants, without altering the original Mesh, making the whole process feasible.

In order to create lots of different variants of the original airfoil to train the Neural Networks, the original airfoil was appropriately randomly deformed within a range of $\pm 20\%$ to create 1000 training variants and 500 testing variants. The deformation process concluded with the generation of a coordinates file for MaPFlow and the respective SDF formatted image of each variant.

After solving the CFD simulations for all variants, the Neural Networks were trained. The Neural Networks resulting values were compared to the CFD simulation results to estimate the Networks' error. Mean Squared Error was the error function of choice, and the Neural Networks were trained using the ADAM optimizer with its default hyper-parameters and 10^{-4} learning rate. Also, they were trained using the mini-batch technique with a batch size of 50.

During the training process, MSE and training time per Epoch data were documented. As it became immediately apparent the Neural Networks were highly performant, managing to predict the Cp-distributions of the given airfoils with an mean accuracy of over 98%. They were also incredibly fast in estimating a single case, with a airfoil's Cp-distribution prediction evaluating in mere milliseconds compared to the couple of minutes required by a CFD solver.

Additionally to the validation of the Neural Networks' precision and speed, the influence of the batch size in the training and the Neural Networks' ability to predict the Cp-distribution for airfoils out of the original training-testing range. Mini-batch size was very influential both in the Neural Networks' accuracy and training speed; a increase in batch size resulted in less training times yet an increase in MSE and vice versa.

On the other hand, the Neural Networks were capable to predict the Pressure distribution of airfoils out of the original range with a decaying accuracy. Namely, the bottom side Network had better performance than the top side Network, something completely expected as the top side is subjected to extreme non-linear phenomena.

Finally, a simple geometry optimizer was developed using the Neural Networks to calculate the Pressure distribution. The geometry optimizer's target was to maximize the airfoil's Lift capacity by uniformly deforming the airfoil's two sides. The deformation was done with a simple one weight per side y-coordinate scaling. This simplistic application proved the efficiency and applicability of Deep Learning techniques in the field of geometry optimization for aerodynamic applications.

7.2 Future Work

As of the year this Thesis was written no one can argue that Artificial Intelligence is a very prosperous field of research that shows promise and offers solutions to many fields of everyday life. The cornerstones of Artificial Intelligence are its adaptability and efficiency. During this study it became clear enough that Artificial Intelligence can be utilized in the field of Aerospace and Mechanical Engineering to predict Pressure distribution along airfoils.

It is significant to notice that this Thesis just proved the feasibility of using SDF images with CNNs to estimate Pressure distributions, yet the field of applications is quite limited. This is a by-product of only accounting for geometrical differences between the airfoils. Contrast to other works, for example Sekar et al.[18], the Neural Networks were trained for fixed Mach and Reynolds Numbers, and the Angle of Attack was kept constant as well. This is highly probable that handicapped the flexibility of the Neural Networks in generalizing and resolving more cases, where the flow particulars are slightly different. So, research on a Network of similar Architecture where the flow conditions are input and not constant is highly advised.

Furthermore, according to the author's point of view, SDF format has huge untapped potential in encoding data and this Thesis only scratched the surface. SDF results a number for its grid point that the plotting library automatically assigns to a color according a predetermined colormap. As the colormaps are compromised from colors within a varying RGB triplet, there are at most $255^3 = 16,581,375$ different colors to discretize the x-y plane, which are arguably more than enough. However, if a colormap uses only, for example, the Red and Green channels to map the SDF values, lets the Blue channel operate as a different indipendent colormap where, for example, the Reynolds Number or the Angle of Attack could be encoded. Still there are $255^2 = 65,025$ values to discretize the x-y plane resulting to an accuracy of $dr = 1.538e-5$, while there are another 255 values available to encode data other than the geometric data.

Moreover, something that could be assessed is the quantity of training and testing specimens. Considering that the Networks' precision does not suffer greatly by reducing the training data could lead to lessen the training's and the data generation's times as less simulations are required. This would increase even more the Neural Networks' efficiency. Also, it could lead the way in creating software solutions where Neural Networks would be trained in highly efficient supercomputers and then distributed to the general public, making CFD simulations accessible to more people lacking the hardware to run CFD solvers on their machines.

Another thing that could be investigated is how the present Networks trained on the RAE-2822 specimens can perform when subjected to relevant to RAE-2822 airfoils, like NASA SC(2)-0412. Considering that they will achieve the same performance as with RAE-2822, this could lead the way of implementing Free Form Deformation in the optimizer, where there would be more degrees of freedom in distorting an airfoil. This would result in further increasing the efficiency of optimizers, who would no longer be restrained in uniformly deforming a single airfoil and would be able to freely explore the design space.

To add up to the functionality of the optimizer, the system of two Networks could be augmented by a third Neural Network that can predict the airfoil's total Drag Coefficient, using as input, maybe, the resulting pressure distribution of the two Networks and/or the SDF image. Now there would be a complete approach to predicting the forces an airfoil experiences. This would eradicate the present optimizer's deficiency in calculating the airfoil's Drag and would further solidify the concept of a Deep Learning powered geometry optimizer.

To conclude, this work is not expected to give a definite answer on the utility of Neural Networks, rather to inspire further research as its results indicate the immense potential Deep Learning has in aiding the design of better and more efficient systems in Aerospace, Mechanical and Marine Engineering.

Appendix A

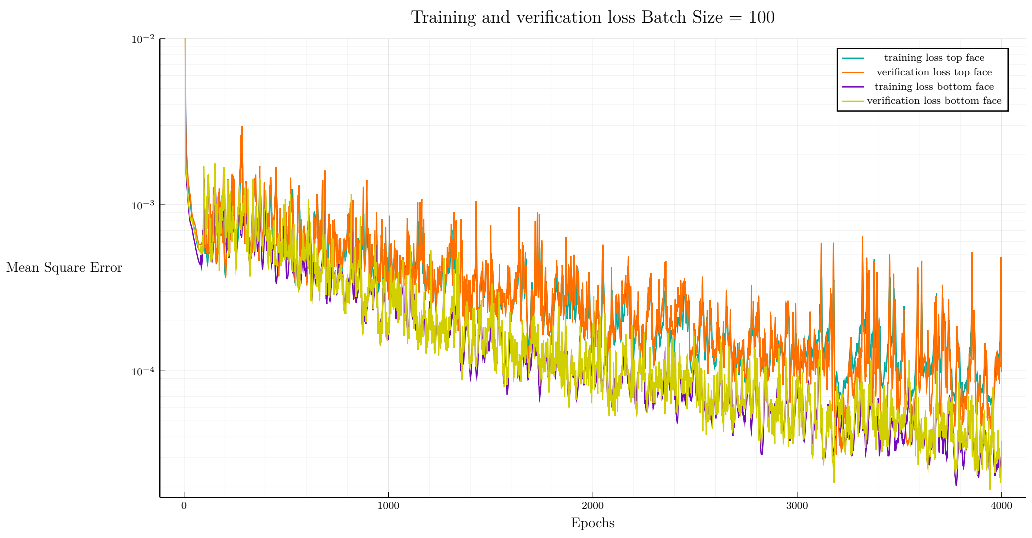
Hardware and Software specifics

Table A.1: Hardware and Software specifics of the Computational Framework used in this work

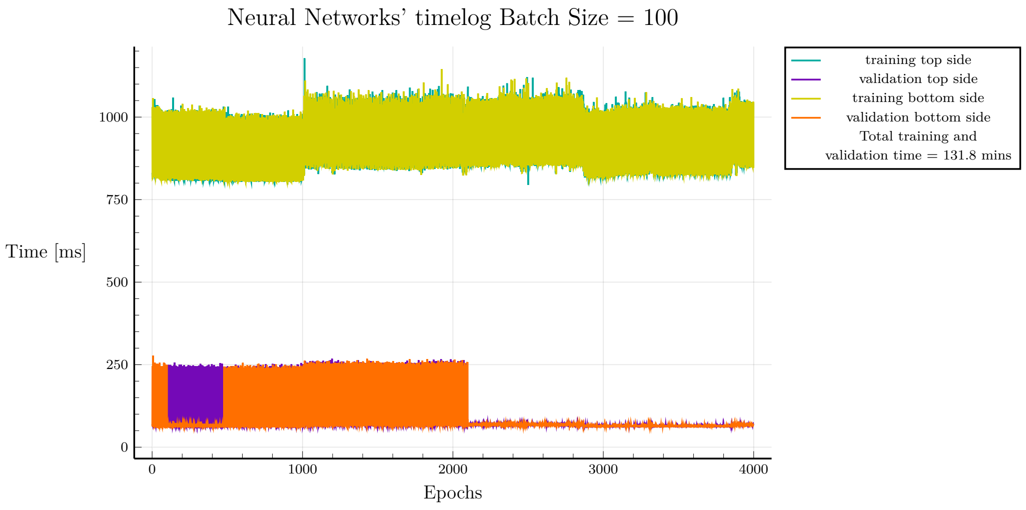
Hardware Particulars	
CPU	Intel(R) Core(TM) i7-6700K
CPU ClockSpeed [Hz]	4.00GHz
CPU Threads	4 cores (8 threads)
RAM	16 GB DIMM DDR4 2133 MHz
GPU	Nvidia GeForce GTX 1070
GPU ClockSpeed [Hz]	1506 MHz
VRAM	8 GB GDDR5
Storage	240GB Corsair Force LE SSD
Software Particulars	
OS	18.04.1-Ubuntu GNU\Linux
Kernel release	5.4.0-97-generic
Architecture	x86_64
CUDA ver.	11.0.0
Fortran Compiler	Intel(R) Parallel Studio XE 2020 Update 4 for Linux
Python ver.	3.7.7
Julia ver.	1.6.1
Flux ver.	0.12.8
CUDA.jl ver.	3.5.0

Appendix B

Convergence Characteristics and Epoch times for different Batch Sizes

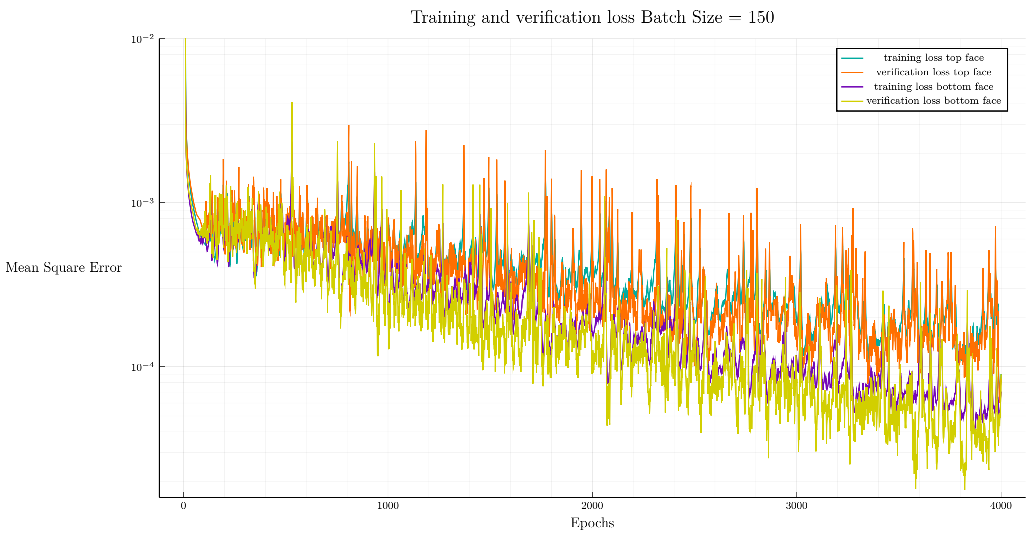


(a) Convergence per Epoch

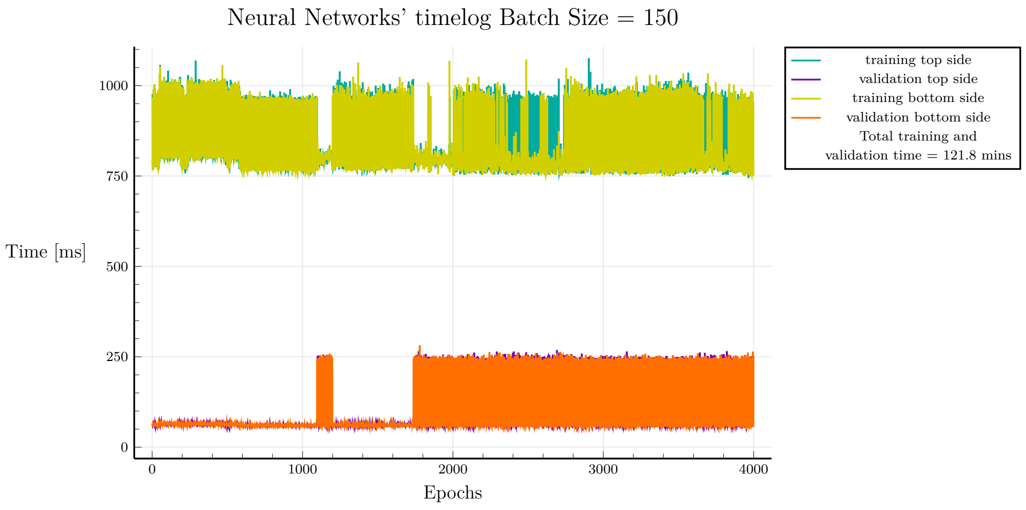


(b) Training and Validation per Epoch

Figure B.1: Batch Size = 100

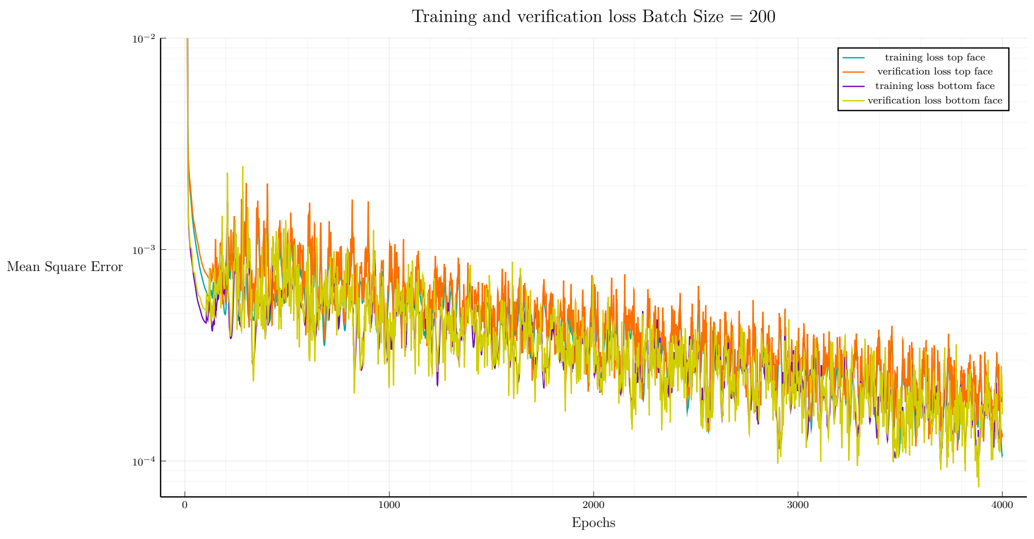


(a) Convergence per Epoch

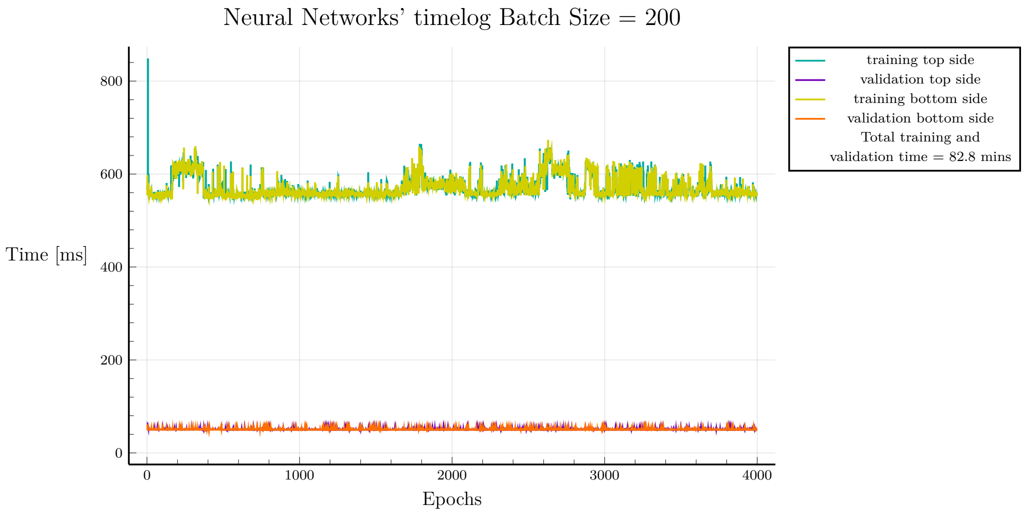


(b) Training and Validation per Epoch

Figure B.2: Batch Size = 150



(a) Convergence per Epoch



(b) Training and Validation per Epoch

Figure B.3: Batch Size = 200

Appendix C

Pictures of the different Mesh variants

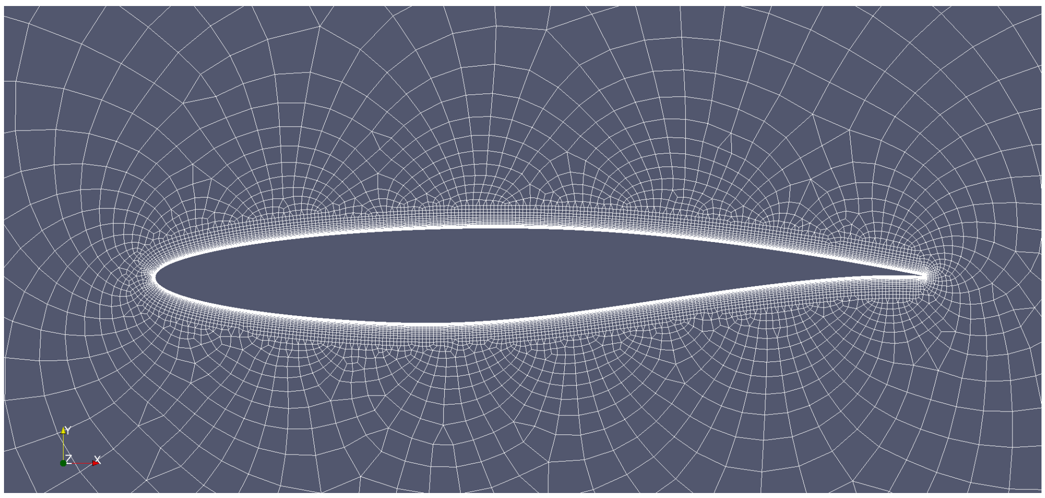
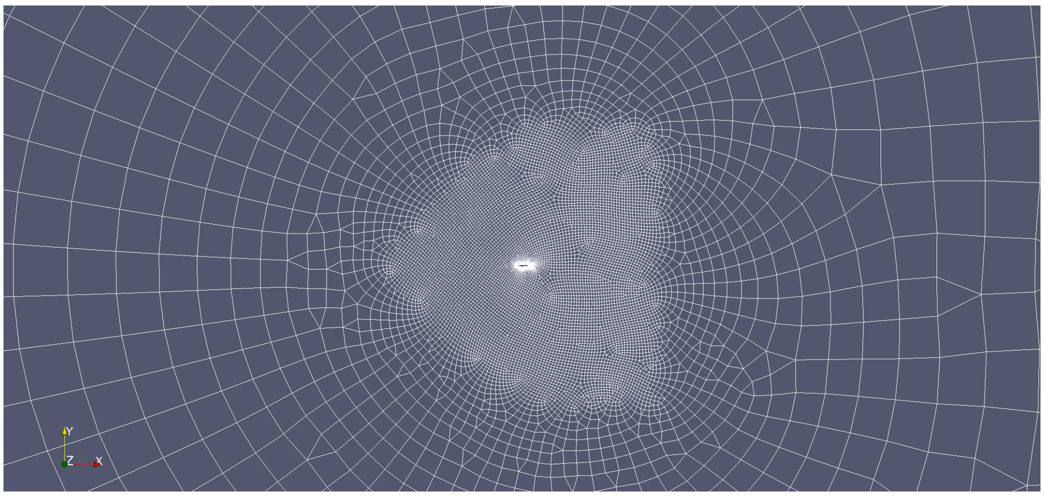
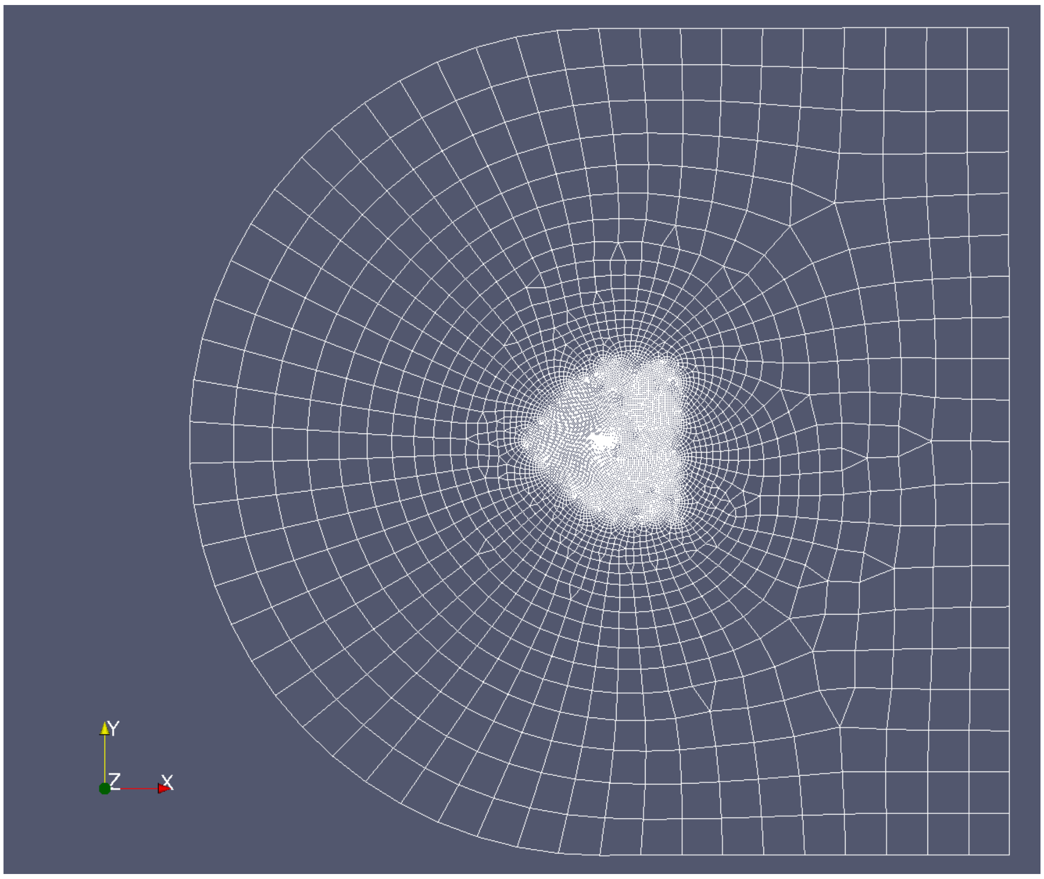


Figure C.1: Mesh #1 Boundary layer geometry

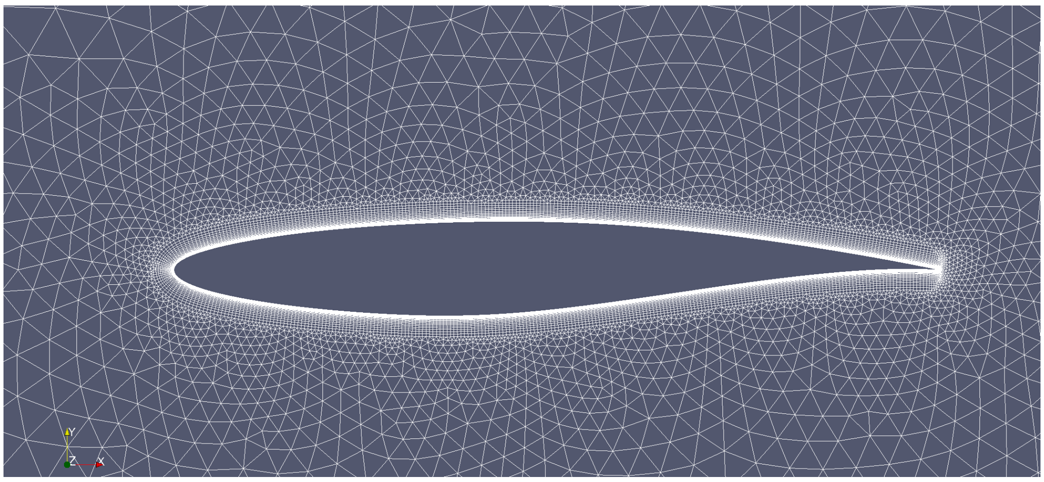


(a) Close field geometry

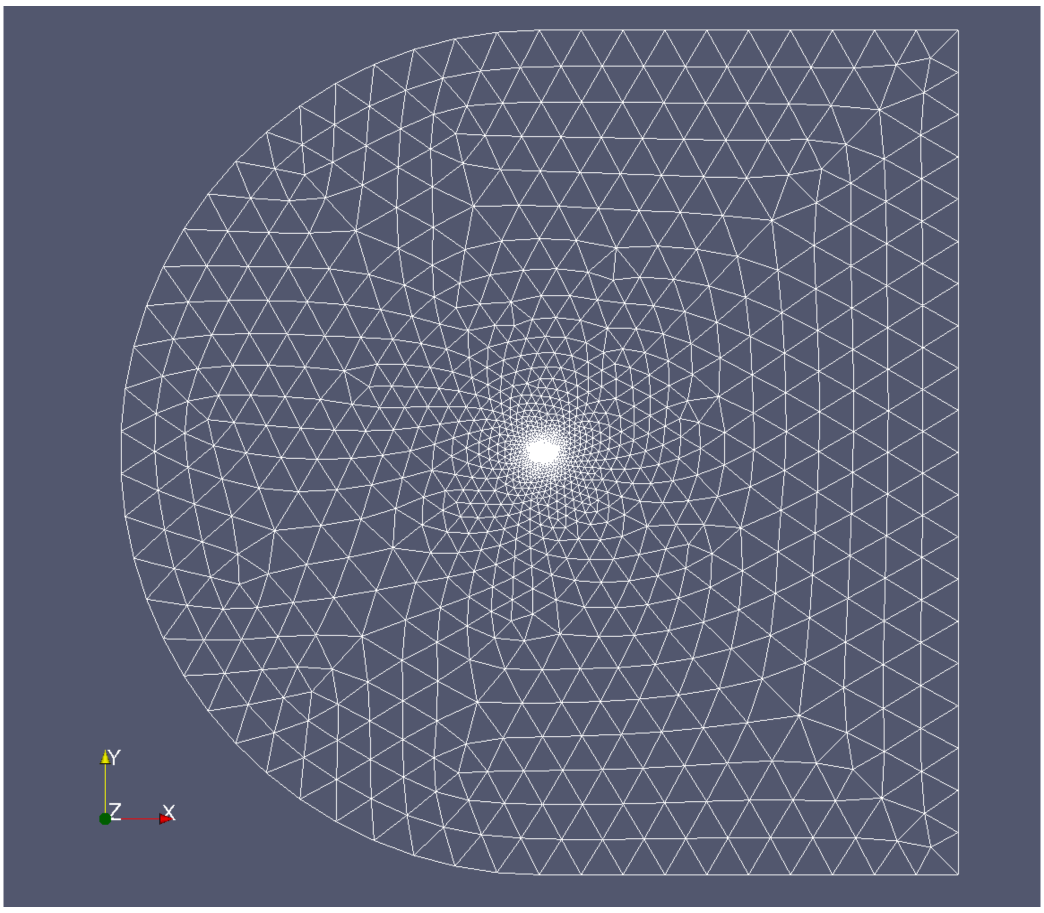


(b) General geometry

Figure C.2: Mesh #1

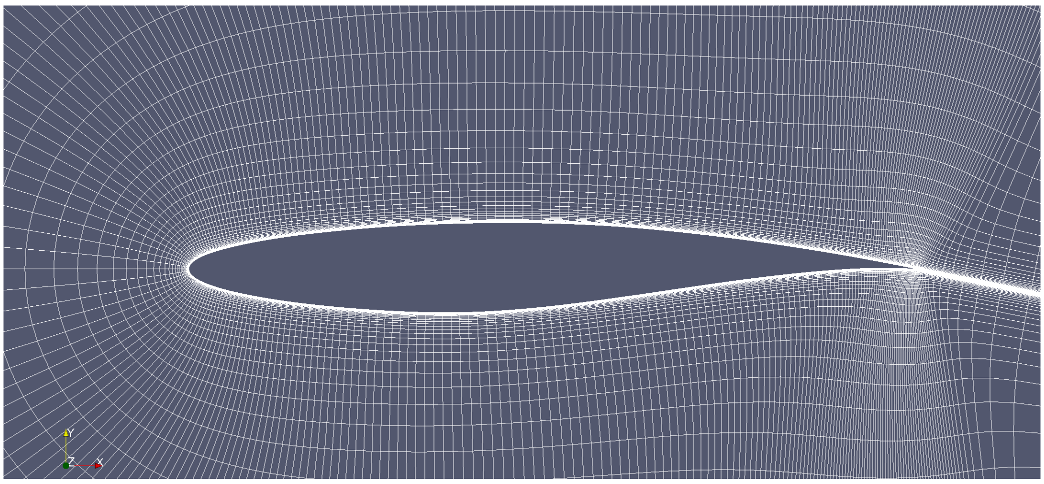


(a) Boundary layer geometry

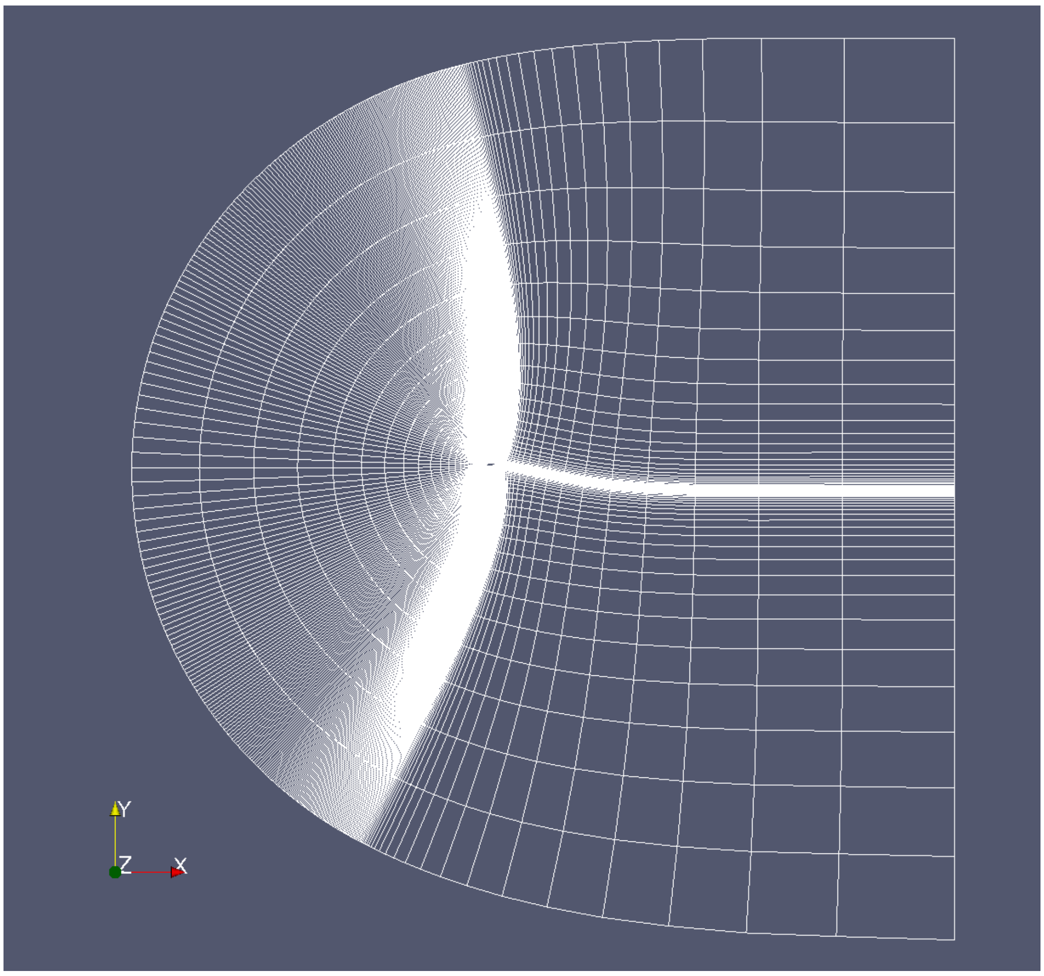


(b) General geometry

Figure C.3: Mesh #2



(a) Boundary layer geometry



(b) General geometry

Figure C.4: Mesh created by NASA

Bibliography

[1] X. Hui, J. Bai, H. Wang, and Y. Zhang, "Fast pressure distribution prediction of airfoils using deep learning", *Aerospace Science and Technology*, vol. 105, p. 105 949, 2020, ISSN: 1270-9638. DOI: https://doi.org/10.1016/j.ast.2020.105949. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1270963820306313.

[2] H. Chen, L. He, W. Qian, and S. Wang, "Multiple aerodynamic coefficient prediction of airfoils using a convolutional neural network", *Symmetry*, vol. 12, p. 544, Apr. 2020. DOI: 10.3390/sym12040544.

[3] X. Jin, P. Cheng, W.-L. Chen, and H. Li, "Prediction model of velocity field around circular cylinder over various reynolds numbers by fusion convolutional neural networks based on pressure on the cylinder", *Physics of Fluids*, vol. 30, no. 4, p. 047 105, 2018. DOI: 10.1063/1.5024595. eprint: https://doi.org/10.1063/1.5024595. [Online]. Available: https://doi.org/10.1063/1.5024595.

[5] Wikipedia. "Navier-stokes equations". (), [Online]. Available: https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations.

[6] Σωκράτης Τσαγγάρης, *Μηχανική των Ρευστών Θεωρία και ασκήσεις*. Εκδόσεις Τσότρας, 2016, ISBN: 978-618-5066-55-0.

[10] R. P. Feynman, R. B. Leighton, M. Sands, and S. B. Treiman, "The feynman lectures on physics", *Physics Today*, vol. 17, no. 8, pp. 45–46, 1964. DOI: 10.1063/1.3051743. eprint: https://doi.org/10.1063/1.3051743. [Online]. Available: https://doi.org/10.1063/1.3051743.

[13] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE", *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. [Online]. Available: https://doi.org/10.1093/mind/LIX.236.433.

[14] J.-L. Wu, H. Xiao, and E. Paterson, "Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework", *Physical Review Fluids*, vol. 3, no. 7, 2018, ISSN: 2469-990X. DOI: 10.1103/physrevfluids.3.074602. [Online]. Available: http://dx.doi.org/10.1103/PhysRevFluids.3.074602.

[15] M. A. Cruz, R. L. Thompson, L. E. Sampaio, and R. D. Bacchi, "The use of the reynolds force vector in a physics informed machine learning approach for predictive turbulence modeling", *Computers & Fluids*, vol. 192, p. 104 258, 2019, ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2019.104258. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045793019302257.

[16] B. Y. Zhou, N. R. Gauger, J. Hauth, X. Huan, M. Morelli, and A. Guardone, "Towards real-time in-flight ice detection systems via computational aeroacoustics and machine learning", *AIAA Aviation 2019 Forum*, p. 3103, 2019. [Online]. Available: www.scopus.com.

[17] Y. Zhang, W. J. Sung, and D. N. Mavris, "Application of convolutional neural network to predict airfoil lift coefficient", in *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. 2018. DOI: 10.2514/6.2018-1903. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2018-1903. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2018-1903.

[18] V. Sekar, Q. Jiang, C. Shu, and B. C. Khoo, "Fast flow field prediction over airfoils using deep learning approach", *Physics of Fluids*, vol. 31, no. 5, p. 057 103, 2019. DOI: 10.1063/1.5094943. eprint: https://doi.org/10.1063/1.5094943. [Online]. Available: https://doi.org/10.1063/1.5094943.

[19] A. Krizhevsky, *Learning multiple layers of features from tiny images*, 2009. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html.

[20] Swyde. "Airfoil". (), [Online]. Available: https://swyde.com/s/Airfoil.

[21] Wikipedia. "Compressible flow". (), [Online]. Available: https://en.wikipedia.org/wiki/Compressible_flow.

[22] ——, "Buckingham Π theorem". (), [Online]. Available: https://en.wikipedia.org/wiki/Buckingham_%CF%80_theorem.

[25] G. Bar-Meir, *Fundamentals of Compressible Fluid Mechanics*. Version 0.5.0, GNU Free Documentation License, 2021. [Online]. Available: https://potto.org/gd.pdf.

[26] H. Liepmann and A. Roshko, *Elements of Gasdynamics*. Wiley, 1957, ISBN: 9780471534600. [Online]. Available: https://books.google.gr/books?id=1BxRAAAAMAAJ.

[27] M. Franke. "Advanced turbulence modelling in aerodynamic flow solvers". (), [Online]. Available: https://www.cfd.tu-berlin.de/research/thermofluid/transport/aero.html.

[28] G. Papadakis, "Development of a hybrid compressible vortex particle method and application to external problems including helicopter flows", Dec. 2014. DOI: http://dx.doi.org/10.26240/heal.ntua.1582.

[29] C. Hirsch, *Numerical Computation of Internal and External Flows*. Wiley, 1990.

[30] A. M., G. D., and T. T.S., "Behavior of Linear Reconstruction Techniques on Unstructured Meshes", *AIAA Journal*, vol. 33, pp. 2038–2049, Nov. 1995.

[31] V. Venkatakrishnan, "On the Accuracy of Limiters and Convergence to Steady State Solutions", *AIAA paper 93-0880*, 1993.

[32] ——, "Convergence to Steady State Solutions of the Euler Equations on Unstructured Grids with Limiters", *Journal of Computational Physics*, vol. 118, no. 1, pp. 120–130, 1995.

[33] A. Jameson, W. Schmidt, E. Turkel, *et al.*, "Numerical solutions of the euler equations by finite volume methods using runge-kutta time-stepping schemes", *AIAA paper*, vol. 1259, 1981.

[34] B. Van Leer, "Flux-vector splitting for the euler equations", in *Eighth international conference on numerical methods in fluid dynamics*, Springer, 1982, pp. 507–512.

[35] J. L. Steger and R. Warming, "Flux vector splitting of the inviscid gasdynamic equations with application to finite-difference methods", *Journal of computational physics*, vol. 40, no. 2, pp. 263–293, 1981.

[36] P. L. Roe, "Approximate riemann solvers, parameter vectors, and difference schemes", *Journal of computational physics*, vol. 43, no. 2, pp. 357–372, 1981.

[37] J. Blazek, *Computational Fluid Dynamics: Principles and Applications*. Elsevier Science, 2001.

[38] B. Robert, V. Veer, and A. Harold, “Simulation of Unsteady Flows Using an Unstructured Navier-Stokes Solver on Moving and Stationary Grids”, *23rd AIAA Applied Aerodynamics Conference*, pp. 1–17, Jun. 2005.

[39] V. N. Vatsa, M. H. Carpenter, and D. P. Lockard, “Re-evaluation of an optimized second order backward difference (bdf2opt) scheme for unsteady flow applications”, *AIAA Paper*, vol. 122, p. 2010, 2010.

[40] D. Mavriplis and A. Jameson, “Multigrid solution of the Navier-Stokes equations on triangular meshes”, *AIAA Journal*, vol. 28, no. 8, pp. 1415–1425, Aug. 1990.

[41] F. Menter, “ Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications”, *AIAA Journal*, vol. 32, pp. 1598–1605. 1994.

[42] D. C. Wilcox *et al.*, *Turbulence modeling for CFD*. DCW industries La Canada, CA, 1998, vol. 2.

[43] F. Menter, “Zonal Two Equation k-omega Turbulence Models for Aerodynamic Flows”, *AIAA Paper 93-2906*, 1993.

[44] D. Koubogiannis, “Numerical Solution of the Navier-Stokes Equations on Unstructured Grids in a Parallel Processing Environment”, Ph.D. dissertation, Laboratory of Thermal Turbomachines, Fluids Section, Department of Mechanical Engineering, NTUA, 1998.

[45] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, ISBN: 0898715342.

[46] A. Theofilopoulos, “Numerical analysis of the dynamic behavior of airfoils with deformable and articulated trailing edge flap”, Dept. Mech. Engineering, Aerodynamics Laboratory, National Technical University of Athens , Greece, 2013.

[47] Y. Zhao, J. Tai, and F. Ahmed, “Simulation of micro flows with moving boundaries using high-order upwind fv method on unstructured grids”, *Computational mechanics*, vol. 28, no. 1, pp. 66–75, 2002.

[48] P. Thomas and C. Lombard, “Geometric conservation law and its application to flow computations on moving grids”, *AIAA journal*, vol. 17, no. 10, pp. 1030–1037, 1979.

[49] D. J. Mavriplis and Z. Yang, “Construction of the discrete geometric conservation law for high-order time-accurate simulations on dynamic meshes”, *Journal of Computational Physics*, vol. 213, no. 2, pp. 557–573, 2006.

[50] H. T. Ahn and Y. Kallinderis, “Strongly coupled flow/structure interactions with a geometrically conservative ale scheme on general hybrid meshes”, *Journal of Computational Physics*, vol. 219, no. 2, pp. 671–696, 2006.

[51] M. Campbell, A. J. Hoane, and F.-h. Hsu, “Deep blue”, *Artif. Intell.*, vol. 134, no. 1–2, pp. 57–83, Jan. 2002, ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1. [Online]. Available: https://doi.org/10.1016/S0004-3702(01)00129-1.

[52] Y. L. Cunn, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition”, *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf.

[53] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[55] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models”, in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[56] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, 2015. arXiv: 1502.01852 [cs.CV].

[57] H. Xiao, K. Rasul, and R. Vollgraf. “Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms”. arXiv: cs.LG/1708.07747 [cs.LG]. (Aug. 28, 2017).

[59] M. Khosla, K. Jamison, A. Kuceyeski, and M. R. Sabuncu, “3d convolutional neural networks for classification of functional connectomes”, in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, D. Stoyanov, Z. Taylor, G. Carneiro, *et al.*, Eds., Cham: Springer International Publishing, 2018, pp. 137–145, ISBN: 978-3-030-00889-5.

[60] S. Ji, W. Xu, M. Yang, and K. Yu, “3d convolutional neural networks for human action recognition”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 1, pp. 221–231, 2013. DOI: 10.1109/TPAMI.2012.59.

[61] A. Payan and G. Montana, *Predicting alzheimer’s disease: A neuroimaging study with 3d convolutional neural networks*, 2015. arXiv: 1502.02506 [cs.CV].

[62] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: 1502.03167 [cs.LG].

[63] C. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. Springer-Verlag New York, 2006, ISBN: 978-1-4939-3843-8.

[64] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[65] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html.

[66] T. T. and H. G. "Lecture 6.5 - rmsprop". (2012).

[67] O. Russakovsky, J. Deng, H. Su, *et al.*, *Imagenet large scale visual recognition challenge*, 2015. arXiv: 1409.0575 [cs.CV].

[68] J. W. Slater. "Rae 2822 transonic airfoil: Study # 4". (), [Online]. Available: https://www.grc.nasa.gov/WWW/wind/valid/raetaf/raetaf04/raetaf04.html.

[69] Wikipedia. "Dota 2". (), [Online]. Available: https://en.wikipedia.org/wiki/Dota_2.

[70] OpenAI, : C. Berner, *et al.*, *Dota 2 with large scale deep reinforcement learning*, 2019. arXiv: 1912.06680 [cs.LG].

[71] M. Innes, E. Saba, K. Fischer, *et al.*, "Fashionable modelling with flux", *CoRR*, vol. abs/1811.01457, 2018. arXiv: 1811.01457. [Online]. Available: https://arxiv.org/abs/1811.01457.

[72] M. Innes, "Flux: Elegant machine learning with julia", *Journal of Open Source Software*, 2018. DOI: 10.21105/joss.00602.

[73] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy", *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[74] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python", *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[75] J. D. Hunter, "Matplotlib: A 2d graphics environment", *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[76] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs", *IEEE Transactions on Parallel and Distributed Systems*, 2018, ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

[77] C. Green, "Improved alpha-tested magnification for vector textures and special effects", in *SIGGRAPH '07*, 2007.

[78] S. Izadi, D. Kim, O. Hilliges, *et al.*, "Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera", *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011.

[79] "Randomized designs - pydoe 0.3.6 documentation". (), [Online]. Available: https://pythonhosted.org/pyDOE/randomized.html#latin-hypercube.

[81] J. S. Gray, J. T. Hwang, J. R. R. A. Martins, K. T. Moore, and B. A. Naylor, "OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization", *Structural and Multidisciplinary Optimization*, vol. 59, no. 4, pp. 1075–1104, Apr. 2019. DOI: 10.1007/s00158-019-02211-z.

[82] "Cobyla". (), [Online]. Available: https://handwiki.org/wiki/COBYLA.

[83] "Slsqp". (), [Online]. Available: https://qiskit.org/documentation/stubs/qiskit.algorithms.optimizers.SLSQP.html.

Image bibliography

[4] M. Kemper, “Leonardo’s laboratory: Studies in flow”, in *Nature issue 571*, pp. 322–323. DOI: 10.1038/d41586-019-02144-z. [Online]. Available: https://doi.org/10.1038/d41586-019-02144-z.

[7] “How hydrofoil boat reduce drag force?” (), [Online]. Available: https://mfame.guru/wp-content/uploads/2021/05/Hydrofoil.jpg.

[8] “Ship stabilizers and different types of ship stabilizers”. (), [Online]. Available: https://www.marinesite.info/2013/09/ship-stabilizers-and-different-types-of.html.

[9] W. Frei. “Which turbulence model shoold i choose for my cfd application ?” (2017), [Online]. Available: https://www.comsol.com/blogs/which-turbulence-model-should-choose-cfd-application/.

[11] J. L. Dick. “080830-n-6107d-001 atlantic ocean (aug. 30, 2008) an f/a-18e super hornet, assigned to the "jolly rogers" of strike fighter squadron (vfa) 103, reaches the speed of sound near the nimitz-class aircraft carrier uss dwight d. eisenhower, cvn-69, during a friends and family day cruise off the coast of virginia.” (2008), [Online]. Available: https://commons.wikimedia.org/wiki/File:An_F-A-18E_Super_Hornet,_assigned_to_the_%22Jolly_Rogers%22_of_Strike_Fighter_Squadron_(VFA)_103,_reaches_the_speed_of_sound_near_the_Nimitz-class_aircraft_carrier_USS_Dwight_D._Eisenhower,_CVN-69.jpg.

[12] M. Horsky. “Wrc model cfd results”. (2013), [Online]. Available: https://mathieuhorsky.wordpress.com/2013/06/29/wrc-model-cfd-results/.

[23] H. Sturm, G. Dumstorff, P. Busche, D. Westermann, and W. Lang, “Boundary layer separation and reattachment detection on airfoils by thermal flow sensors”, *Sensors*, vol. 12, no. 11, pp. 14 292–14 306, 2012, ISSN: 1424-8220. DOI: 10.3390/s121114292. [Online]. Available: https://www.mdpi.com/1424-8220/12/11/14292.

[24] Wikipedia. “File:transonic flow patterns.svg”. (), [Online]. Available: https://commons.wikimedia.org/wiki/File:Transonic_flow_patterns.svg.

[54] ——, “Machine learning”. (), [Online]. Available: https://en.wikipedia.org/wiki/Deep_learning#/media/File:AI-ML-DL.svg.

[58] S. Saha. “A comprehensive guide to convolutional neural networks — the eli5 way”. (), [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[80] Wikipedia. “Signed distance function (sdf)”. (), [Online]. Available: https://en.wikipedia.org/wiki/Signed_distance_function#/media/File:Signed_distance_field_duck.svg.