Data-Driven Flow Field Prediction and Generation for Airfoils Using Artificial Intelligence Models

a project presented to
The Faculty of the Department of Aerospace Engineering
San José State University

in partial fulfillment of the requirements for the degree
Master of Science in Aerospace Engineering

by

Wilber A. Mendez Vides

May 2026

approved by

Dr. Yawo Ezunkpe
Faculty Advisor



© 2026
Wilber A. Mendez Vides
ALL RIGHTS RESERVED

ABSTRACT

Data-Driven Flow Field Prediction and Generation for Airfoils Using Artificial Intelligence Models

Wilber A. Mendez Vides

This document utilizes the DeepCFD framework in which CFD acquired data is used to train an AI model using a U-Net architecture. The model in this document was trained using CFD data acquired from Ansys Fluent. The training was done through a U-Net architecture. Using Local Normalization, Global Normalization, Dropout Method, and AdamW optimizer a successful model was developed. The model accurately predicts fluid flow fields – specifically velocity in the x and y directions and static pressure for symmetric, low-camber, medium-camber, and high-camber airfoils across varying thicknesses and angles of attack. When evaluated against 60 validation cases, the model demonstrated low absolute error compared to the ground truth data. The discrepancies within the models were primarily observed to be within the boundary layer, a result that can be attributed to a low resolution 256 x 256 grid and hardware limitations.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Yawo Ezunkpe for having convinced me to undertake a project of this kind. It is truly a remarkable skill of a brilliant educator to know exactly how to challenge his students with projects that balance challenge and growth. This experience has deeply expanded my comprehension of Artificial Intelligence and allowed me to build confidence as an engineer, undertaking even the most complex problems.

To my mother, Aydee Vides, the person that sacrificed her life so that my brother and I may have a chance at a better one. You are the strongest and most resilient woman I know. I only wish I could have a fraction of your strength.

To Sara, for giving me constant support through this time and being understanding of my absence. Thank you for all the times you heard me rant when I was feeling overwhelmed.

Finally, thank you to all of my family members, friends, and co-workers for having a strong belief in me and supporting me.

Table Of Contents

List of Figures vii
List of Tables ix
Symbols x
1. Introduction 1
1.1 Motivation 1
1.1.1 A Brief History 1
1.2 Literature Review 4
1.2.1 A Data-Driven Machine Learning Approach for Turbulent Flow Field Prediction Based on Direct Computational Fluid Dynamics Database 4
1.2.2 Deep Learning for Fluid Velocity Field Estimation: A Review 5
1.2.3 Machine Learning-Accelerated Computational Fluid Dynamics 6
1.2.4 Recent Advances on Machine Learning for Computation Fluid Dynamics: A survey . 6
1.2.5 Application of Artificial Intelligence in Computational Fluid Dynamics 7
1.2.6 Summary of Current Advancements and Limitations 9
1.3 Project Proposal 9
1.4 Methodology 10
2. Existing Models and Applications 11
2.1 Currently Used Models 11
2.1.1 Convolutional Neural Networks How They Work and Their Downsides 11
2.1.2 Examples of Data-Driven Convolutional Neural Networks 12
2.2 Chosen Project Model 17
3. Chosen Model 18
3.1 Preliminary Decision 18
3.2 Running DeepCFD 18
3.3 Next Steps 23
4. Current CFD Theory and Method for Airfoils 25
4.1 CFD Theory 25
4.2 CFD Setup 26
4.2.1 CAD Setup 27
4.2.3 Physics Set Up 35
4.2.4 Results and Discussion 36
5. Creating the Training Dataset Using Ansys Fluent 41
5.1 Creation of Dataset 41

6. Importing Data into PyCharm 44
6.1 Importing the Pressure and Velocity Data 44
6.2 Importing the Airfoil Geometry 45
7. Initial Training and Results 46
7.1 Version One 46
7.2 Version Two 53
7.3 Version Three 60
7.4 Boundary Challenge 67
8. Final Training and Results 69
8.2 Version Four 69
8.3 Version Five 79
8.4 Conclusion 94
9. Future Work 97
References (AIAA) 99
Appendix A 102
A.1 Airfoil Generator Code 102
A.2 Initial Gridder Code 103
A.3 Initial SDF Code 104
A.4 Master Trainer Version 1 Code 105
A.5 Comparison Script for Version 1 109
A.6 Master Trainer Version Two Code 111
A.7 Comparison Script for Version Two 113
A.8 Master Trainer Version Three Code 116
A.9 Comparison Script for Version Three 118
A.10 Fixed SDF Code 121
A.11 Fixed Gridder Code 122
A.12 Master Trainer Version Four Code 124
A.13 Comparison Script for Version Four 127
A.14 Global Normalization Parameters 129
A.15 Master Trainer Version Five Code 130
A.16 Comparison Script for Version Five 133

List of Figures

Figure 1.1 – Fitzroy's hand-drawn weather chart 2
Figure 1.2 - Comparison between image-based (red) and CFD-based (black) methods 5
Figure 1.3 – Schematic of a general CNN structure 6
Figure 1.4 – Data-driven surrogate models 7
Figure 1.5 – Physics-driven surrogate models 7
Figure 1.6 – Neural network structures 8
Figure 1.7 – Typical framework/structure of AI algorithms in aerodynamic models 8
Figure 2.1 – Comparison between simpleFOAMs and DeepCFDs velocity magnitude and pressure as well as their absolute error for circular shape 14
Figure 2.2 – Comparison between simpleFOAMs and DeepCFDs velocity magnitude and pressure as well as their absolute error for triangular shape 15
Figure 2.3 – Velocity profile of predicted model vs CFD results at M = 0.27, α = 3.5 16
Figure 2.4 – Velocity profile of predicted model vs CFD results at M = 0.44, α = 2.41 16
Figure 3.1 – Ground truth vs predicted results after training for 200 epochs 20
Figure 3.2 – Ground truth vs predicted results after training for 2000 epochs 21
Figure 3.3 – Ground truth vs predicted results after training for 500 epochs 23
Figure 4.1 – Resulting NACA0012 airfoil surface 28
Figure 4.2 – Air domain geometry around the airfoil 28
Figure 4.3 – Rectangular body of influence for mesh refinement 29
Figure 4.4 – Far away view of final geometry 29
Figure 4.5 – Close up view of final geometry 30
Figure 4.6 – Inflation mesh around the airfoil. 31
Figure 4.7 – Close up of inflation mesh around the airfoil 32
Figure 4.8 – Resulting mesh refinement around the airfoil 34
Figure 4.9 – Resulting mesh refinement at the wake region 34
Figure 4.10 – Mesh refinement for smooth transitions between layers and cells 34
Figure 4.11 – Velocity contour at 0° α 36
Figure 4.12 – Static pressure contour at 0° α 36
Figure 4.13 – Velocity contour at 2° α 36
Figure 4.14 – Static pressure contour at 2° α 37
Figure 4.15 – Velocity contour at 5° α 37
Figure 4.16 – Static pressure contour at 5° α 37
Figure 4.17 – Velocity contour at 8° α 38
Figure 4.18 – Static pressure contour at 8° α 38
Figure 4.19 – Velocity contour at 10° α 38
Figure 4.20 – Static pressure contour at 10° α 39
Figure 7.1 – Comparative fluid flow field analysis of v1: NACA0012 at α = 0° 48
Figure 7.2 – Comparative fluid flow field analysis of v1: NACA0012 at α = 10° 49
Figure 7.3 – Comparative fluid flow field analysis of v1: NACA2412 at α = 5° 50
Figure 7.4 – Comparative fluid flow field analysis of v1: NACA4412 at α = 8° 51
Figure 7.5 – Comparative fluid flow field analysis of v1: NACA6412 at α = 2° 52
Figure 7.6 – Comparative fluid flow field analysis of v2: NACA0012 at α = 0° 55
Figure 7.7 – Comparative fluid flow field analysis of v2: NACA0012 at α = 10° 56
Figure 7.8 – Comparative fluid flow field analysis of v2: NACA2412 at α = 5° 57

Figure 7.9 – Comparative fluid flow field analysis of v2: NACA4412 at α = 8°..... 58
Figure 7.10 – Comparative fluid flow field analysis of v2: NACA6412 at α = 2°..... 59
Figure 7.11 – Comparative fluid flow field analysis of v3: NACA0012 at α = 0°..... 62
Figure 7.12 – Comparative fluid flow field analysis of v3: NACA0012 at α = 10°..... 63
Figure7.13 – Comparative fluid flow field analysis of v3: NACA2412 at α = 5°..... 64
Figure 7.14 – Comparative fluid flow field analysis of v3: NACA4412 at α = 8°..... 65
Figure 7.15 – Comparative fluid flow field analysis of v3: NACA6412 at α = 2°..... 66
Figure 8.1 – Comparative fluid flow field analysis of v4: NACA0012 at α = 0°..... 70
Figure 8.2 – Comparative fluid flow field analysis of v4: NACA2412 at α = 5°..... 71
Figure 8.3 – Comparative fluid flow field analysis of v4: NACA4412 at α = 10°..... 72
Figure 8.4 – Comparative fluid flow field analysis of v4: NACA6418 at α = 2°..... 73
Figure 8.6 – Comparative fluid flow field analysis of v4: NACA2418 at α = 0°..... 75
Figure 8.7 – Comparative fluid flow field analysis of v4: NACA4426 at α = 5°..... 76
Figure 8.8 – Comparative fluid flow field analysis of v4: NACA6426 at α = 10°..... 77
Figure 8.9 – Comparative fluid flow field analysis of v4: NACA0026 at α = 2°..... 78
Figure 8.10 – Comparative fluid flow field analysis: NACA0012 at α = 0°..... 81
Figure 8.11 – Comparative fluid flow field analysis: NACA0018 at α = 5°..... 82
Figure 8.12 – Comparative fluid flow field analysis: NACA0026 at α = 10°..... 83
Figure 8.13 – Comparative fluid flow field analysis: NACA2412 at α = 5°..... 84
Figure 8.14 – Comparative fluid flow field analysis: NACA2418 at α = 10°..... 85
Figure 8.15 – Comparative fluid flow field analysis: NACA2426 at α = 0°..... 86
Figure 8.16 – Comparative fluid flow field analysis: NACA4412 at α = 5°..... 87
Figure 8.17 – Comparative fluid flow field analysis: NACA4418 at α = 5°..... 88
Figure 8.18 – Comparative fluid flow field analysis: NACA4426 at α = 5°..... 89
Figure 8.19 – Comparative fluid flow field analysis: NACA6412 at α = 10°..... 90
Figure 8.20 – Comparative fluid flow field analysis: NACA6418 at α = 0°..... 91
Figure 8.21 – Comparative fluid flow field analysis: NACA6426 at α = 5°..... 92

List of Tables

Table 4.1 – Global mesh settings 30
Table 4.2 – Mesh settings for inflation refinement around the airfoil 32
Table 4.3 – Mesh settings for sizing around the airfoil 32
Table 4.4 – Mesh settings for wake refinement behind trailing edge of the airfoil 33
Table 4.5 – Mesh settings to fix cell transitions from inflation layer to wake refinement 33
Table 5.1 – The x and y components for corresponding α 43

Symbols

Symbol	Definition	Units (SI)
δ	Partial derivative	Unitless
d	Derivative	Unitless
ρ	Density	$\frac{kg}{m^3}$
p	Static Pressure	Pa
t	Time	s
u/U_x	Velocity component in the x-direction	m/s
v/U_y	Velocity component in the y-direction	m/s
w/U_z	Velocity component in the z-direction	m/s
x	Normalized Spatial Direction	Unitless
y	Normalized Spatial Direction	Unitless
z	Normalized Spatial Direction	Unitless
Re_r	Reynolds Number	Unitless
τ	Stress	N/m^2
E_T	Total Energy	J
P_r	Prandtl Number	Unitless
q	Heat Flux	W/m^2
M	Mach Number	Unitless
α	Angle of Attack	Degrees
G_t	Ground Truth Value	m/s,Pa
P_v	Predicted Value	m/s,Pa
C_l	Coefficient of Lift	Unitless
V_{Inlet}	Inlet Velocity	m/s

1. Introduction

1.1 Motivation

Creating a good design relies heavily on being able to perform rigorous tests on prototypes through the reiterative design process. One of the things that makes aerospace engineering so difficult is the innate inability of humans to visualize the flow of air around designs. Aerospace engineers get around this problem by using computational fluid dynamics (CFD) simulations. This allows engineers to create a design in 3D using computer aided design (CAD) tools and then use CFD tools to simulate the expected conditions that the design will undergo. This has been a revolutionary tool as it cuts down on the costs of creating physical models and performing tests only to later be scrapped and redesigned. However, this method has its shortcomings.

CFD requires vast computations to be made by the computer. This is more apparent with higher fidelity models. For example, the computational cost and memory demand are very high when it comes to aerodynamic shape optimization [1]. The higher the fidelity the finer the mesh must be to truly capture the underlying physics. Having a finer mesh means discretizing the model into smaller finite volumes or areas. This greatly increases the computational time required to properly run a simulation. In most cases, running complex simulations requires companies to use extremely fast and expensive computers. Many companies must use an array of central processing units (CPUs) and graphics processing units (GPUs) in order to run complex high-fidelity simulations [2]. In addition, the cost of licenses for CFD programs can become quite expensive.

Technology has advanced significantly since the beginning of the first CFD programs were first introduced into the engineering fields. It is now time to implement newer technology into the tools that engineers have used to design. With the increase in artificial intelligence (AI) technology in the past few years, it is only logical to try to implement AI into the engineering design process.

1.1.1 A Brief History

Aerospace engineers have always been deemed to be some of the smartest people in the world. People often use the phrase, “It’s not rocket science”, to lower or completely neglect the difficulty of a problem in front of them. Why is this? Aerospace engineers are constantly dealing with one of the most difficult problems known to mankind, the behavior of air. Earth’s atmosphere is composed of many different elements and molecules. The behavior of these molecules is dictated by many different components such as temperature, density, viscosity, and even molecular composition. In essence an aerospace engineer must be knowledgeable in physics, mathematics, and chemistry to understand the core behavior of Earth’s atmosphere. It is this in addition to the inability of humans to see the air, amongst many other challenges, that makes aerospace engineering so difficult. So how did aerospace engineers first create and design airplanes?

Robert Fitzroy was a British naval officer during the early chapters of his life. His fame and contribution to society came later in life. In 1854 he founded the Meteorological (Met) Office, with the purpose of understanding the weather in the oceans, particularly in the Atlantic Ocean. This was done as a way to predict the weather and potentially save lives at sea. Initially, stations that recorded data were set up along the English and Irish coastlines. These stations transmitted

the data via telegraph to the Met Office in London. The data consisted of pressure, temperature, wind direction, wind force, cloud amount, and weather conditions; such as, rain, thunder, fog, and sea disturbances. In London calculations were made in order to predict the behavior of the weather. With the experience that Fitzroy had, he began to develop and pioneer the science of meteorology. His weather forecasts would predict the weather two days in advance. As time went on, he refined his methods; however, the criticism he received for his work took its toll on him. Although he was a pioneer, Fitzroy was not able to predict the weather to the precision that weather forecasters can in today's world. Eventually, he took his own life. [3] The ultimate dream he had was to be able to set up stations all over the world. Have the data be transmitted to a central building in which individuals would work to create a complete weather chart for the entire Earth. In doing so he hoped to create more accurate predictions with near instant results.

However, the work he had begun became more precise as time went on until it evolved into the systems that are used today. What's interesting about his work; however, is that Fitzroy would break down the map into sections. He would use the data gathered from his stations to predict the weather in a given section. Below is an image that depicts one of Fitzroy's hand drawn weather charts [4]. What Fitzroy had developed was a method similar to what CFD uses. In that, by discretizing an area or volume into small finite areas and volumes, calculations could be made regarding the behavior of air; thus, producing a visualization into the behavior of air. At the time, the methods that Fitzroy used were not very accurate. However, it is one of the very early documented examples of mankind seeking solutions to fluid dynamic problems.

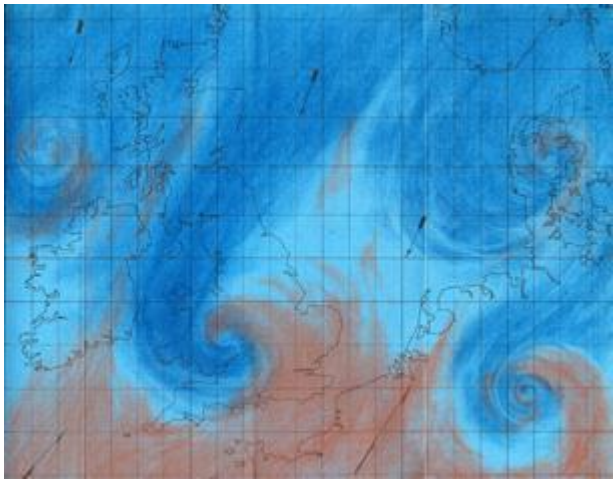


Figure 1.1 – Fitzroy's hand-drawn weather chart

The introduction of computers gave birth to many tools that helped advance many fields. One of these many tools was CFD programs. CFD is the science of using computer written programs that follow mathematical laws that describe the physics and chemistry of a given fluid or gas to predict their flows. These laws are famously known as the Law of Conservation of Mass, Law of Conservation of Momentum, and Law of Conservation of Energy. Many know these mathematical equations as the Navier Stokes Equations. Equation 1.1 depicts the Law of Conservation of Mass, otherwise known as the continuity equation. Equations 1.2-1.4 depict the Law of Conservation of Momentum. Momentum is a vector quantity; thus, all three independent spatial components must

be analyzed independently from the other two. Finally, equation 1.5 describes the Law of Conservation of Energy [5]. The reason it is said that aerospace engineers deal with one of the most difficult problems is that the physics involved in fluid dynamics is coupled. Meaning that any change in any parameter affects all of the equations. Solving these five equations can be rather difficult. These equations do not have analytical solutions. Rather numerical methods are used in order to approximate the solutions. These numerical methods usually require iterative computations and algorithms. Solving these equations for each individual cell is nearly impossible, which is why computers are used. Depending on the case, these equations can be simplified. One of the most famous simplifications comes in the form of Bernoulli's Equations which assume steady flow, incompressibility, inviscid flow, while assuming a one-dimensional streamline flow.

$$\frac{\delta\rho}{\delta t}+\frac{\delta(\rho u)}{\delta x}+\frac{\delta(\rho v)}{\delta y}+\frac{\delta(\rho w)}{\delta z}=0 \tag{1.1}$$

$$\frac{\delta(\rho u)}{\delta t}+\frac{\delta(\rho u^2)}{\delta x}+\frac{\delta(\rho uv)}{\delta y}+\frac{\delta(\rho uw)}{\delta z}=-\frac{\delta p}{\delta x}+\frac{1}{Re_r}\left(\frac{\delta\tau_{xx}}{\delta x}+\frac{\delta\tau_{xy}}{\delta y}+\frac{\delta\tau_{xz}}{\delta z}\right) \tag{1.2}$$

$$\frac{\delta(\rho v)}{\delta t}+\frac{\delta(\rho uv)}{\delta x}+\frac{\delta(\rho v^2)}{\delta y}+\frac{\delta(\rho vw)}{\delta z}=-\frac{\delta p}{\delta y}+\frac{1}{Re_r}\left(\frac{\delta\tau_{xy}}{\delta x}+\frac{\delta\tau_{yy}}{\delta y}+\frac{\delta\tau_{yz}}{\delta z}\right) \tag{1.3}$$

$$\frac{\delta(\rho w)}{\delta t}+\frac{\delta(\rho uw)}{\delta x}+\frac{\delta(\rho vw)}{\delta y}+\frac{\delta(\rho w^2)}{\delta z}=-\frac{\delta p}{\delta z}+\frac{1}{Re_r}\left(\frac{\delta\tau_{xz}}{\delta x}+\frac{\delta\tau_{yz}}{\delta y}+\frac{\delta\tau_{zz}}{\delta z}\right) \tag{1.4}$$

$$\frac{\delta(E_T)}{\delta t}+\frac{\delta(uE_T)}{\delta x}+\frac{\delta(vE_T)}{\delta y}+\frac{\delta(wE_T)}{\delta z}=-\frac{\delta(up)}{\delta x}-\frac{\delta(vp)}{\delta y}-\frac{\delta(wp)}{\delta z}-\frac{1}{Re_rPr_r}\left(\frac{\delta q_x}{\delta x}+\frac{\delta q_y}{\delta y}+\frac{\delta qz}{\delta z}\right)+\frac{1}{Re_r}\left[\frac{\delta}{\delta x}\left(u\tau_{xx}+v\tau_{xy}+w\tau_{xz}\right)+\frac{\delta}{\delta y}\left(u\tau_{xy}+v\tau_{yy}+w\tau_{zz}\right)+\frac{\delta}{\delta z}\left(u\tau_{xz}+v\tau_{yz}+w\tau_{zz}\right)\right] \tag{1.5}$$

The way in which most CFD programs work is actually quite straightforward. It usually begins by creating a CAD model; of what is going to be simulated. CAD models can be created in independent software and imported into CFD software, although most CFD softwares have simple CAD abilities as well. Next is to define the fluid flow domain, meaning what parts of the CAD will have fluid flowing through it. Next is to discretize the geometry into small finite sections. This is called the mesh. Meshes can be formed using different types of techniques [2]. Depending on the shape that is being evaluated certain techniques work better. This is one of the most crucial parts of CFD. Developing a good mesh can cut down on computational expenses while also producing accurate simulations. Usually, the finer the mesh is, the better it can capture the underlying physics. However, this also means that the program has to run more calculations. This can easily scale the amount of time needed to run a single simulation by hours.

CFD has allowed weather forecasting to approach the dream that Fitzroy had. Today weather forecasters use CFD as a crucial tool in order to accurately predict the weather up to a week in advance and more in some cases. Yet, the itch for instant fluid flow prediction still remains. Now more so than never. With the advancements in technology, mankind's goals get more ambitious. The next step is to see how AI can make CFD simulations even more efficient. Can AI be the tool that brings near instant flow field prediction and generation?

1.2 Literature Review

1.2.1 A Data-Driven Machine Learning Approach for Turbulent Flow Field Prediction Based on Direct Computational Fluid Dynamics Database

One of the most common techniques to predict fluid flow fields using machine learning is the employment of image-based techniques by mapping scattered CFD data onto cartesian coordinate grids. However, these methods encounter data scarcity in boundary layers and wake areas. Convolutional neural networks (CNN) have been used as a technique in CFD using flow field images as data sets for flow field prediction. The issue with this method is that CNN requires high pixel density for critical areas [6]. This increases the data input volume and drastically increases the model training time as well.

In order to overcome these issues authors M. Nemati and A. Jahangirian proposed a method in where datasets are generated directly from the flow field properties of CFD grid points. This method favors the network due to the characteristics of unstructured grids which provide varying point-spacing near the object surface and the far field. The construction of the dataset also eliminates the need for interpolation and extrapolation, which preserves the accuracy of the CFD data. This ultimately reduces the amount of input data needed to train the model which reduces the machine training cost [6].

The use of control volume method (CVM) to discretize the governing equations during the data generation phase ensures that the dataset preserves the mass, momentum and energy in the predicted flow field. This eliminates the worry of the model adding artificial effects. Image-based approaches discretize every point in the field uniformly. This is why it is hard to capture physics in critical areas such as the boundary layer or the wake. Making the discretized dimensions smaller results in the overall mesh being significantly finer which increases the computational time. Using the CFD data gives the neural network a much more refined mesh structure to use as the discretized dimensions vary and are fine only where they need to be [6]. Figure 1.2 shows the reduction and improvement that CFD-based structures provide over image-based. It can be seen in image (b) that the CFD-based structure uses bigger discretizing dimensions whereas image-based uses significantly more uniform discretizing dimensions. Additionally, in image (c) it can be seen that at the boundary layer, the CFD-based approach provides a good mesh structure, whereas the image-based approach simply lacks the structure to truly capture the physics within the boundary layer and its transition from it. Finally, image (d) shows the lack in refinement from the image-based method when it comes to the wake, where turbulent flow is predicted to occur.

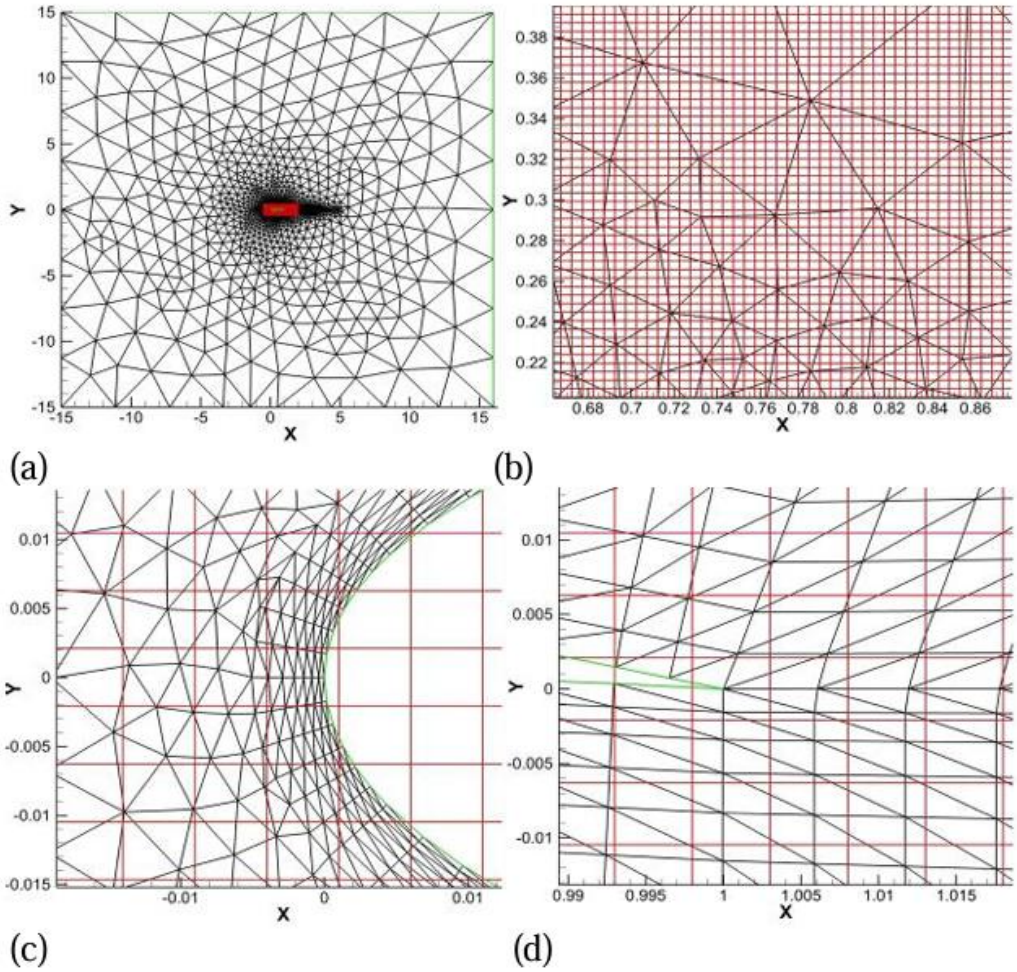


Figure 1.2 - Comparison between image-based (red) and CFD-based (black) methods

1.2.2 Deep Learning for Fluid Velocity Field Estimation: A Review

Deep learning models are architectures that typically contain multiple layers of neural networks. Many of the popular deep learning models include deep belief network (DBN), convolutional neural network (CNN), and Recurrent Neural Network (RNN). The field of fluid mechanics mainly uses CNN and RNN [7].

CNN is a feedforward neural network with convolutional computations, a method that was inspired by the receptive field mechanism of the cerebral cortex studied in biology. It is broken down into 5 layers. The first layer is the input layer, followed by the convolutional layer, followed by the pooling layer, which is fed into the fully connected layer, to finally be presented in the output layer. This structure can be seen in figure 1.3 [7]. There are three interesting operations in CNN: local receptive fields, weight sharing, and pooling layers. These layers allow deeper CNN networks to be built which effectively result in them learning more abstract features in images.

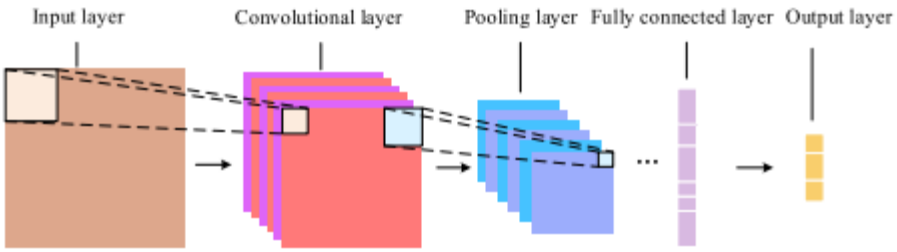


Figure 1.3 – Schematic of a general CNN structure

RNN are used when the input samples are not independent of each other. RNN ensures that the output of each time is not only related to the input at present but also related to the input at previous timestamps.

1.2.3 Machine Learning-Accelerated Computational Fluid Dynamics

The use of partial differential equations (PDEs) is central to the engineering fields and the physical sciences. The applications range from real world uses such as weather, climate, and wildfires; to theoretical applications such as plasma physics [8]. Although CFD solvers can give us good solutions, the problem of computational time still remains. For example, in turbulent flow, increasing the Re value by ten-fold can lead to a thousand-fold increase in computational cost [8]. Through the use of end to end deep learning the authors were able to improve approximations for two-dimensional turbulent flows. The results showed an improvement of 40 -80 times faster computational times while comparing with baseline solvers with 8-10 times finer resolutions [8]. Through the use of CNN and interpolation algorithms, they were able to accelerate the process of direct numerical simulations (DNS).

1.2.4 Recent Advances on Machine Learning for Computation Fluid Dynamics: A survey

Data-driven surrogates are AI models that are trained using only observed data. These models are capable of simulating complex fluid dynamics. There are two types of approaches: dependent on discretization and independent of discretization.

There are three subcategories for approaches dependent on discretization. The first is regular grids; regular grids refer to methods that discretize a scene into uniformly generated dimensions. Similar to the ones used in CNN applications mentioned above [6]. The second subsection is irregular meshes. Irregular meshes use adaptable solutions such as graph neural networks (GNN) for more diverse and size varying geometries [9]. Finally, Lagrangian particles; this method uses tens of thousands of particles effectively linking each particle to its neighbors and simulating Lagrangian fluid dynamics [9]. One of the most well-known applications of such methods is in water simulations used by visual effects artists for simulating water in films.

Similar to dependent discretization methods, independent discretization methods also have 3 subsections: deep operator network, in physical space, and in Fourier space. Deep operator network provides a way to learn mapping in functional space. In Fourier space parametrizes the

integral kernel directly in Fourier space. This creates expressive and efficient architecture [9]. Figure 1.4 below depicts a visualization into data-driven surrogate models.

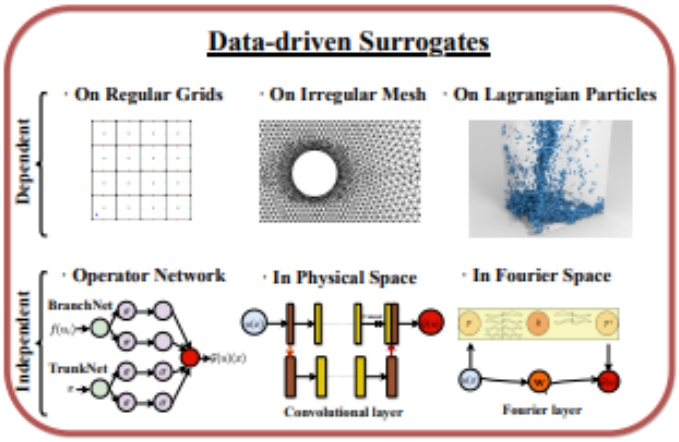


Figure 1.4 – Data-driven surrogate models

Physics-driven surrogates use physics-based priors to leverage physical laws in order to enhance the model's reliability and applicability. This ensures that the predictions made by the model adhere to physical principles. Physics-informed neural networks (PINN) blend deep learning with existing physical laws to solve complex fluid dynamics [9]. Figure 1.5 below depicts a visualization of physics-driven surrogates.

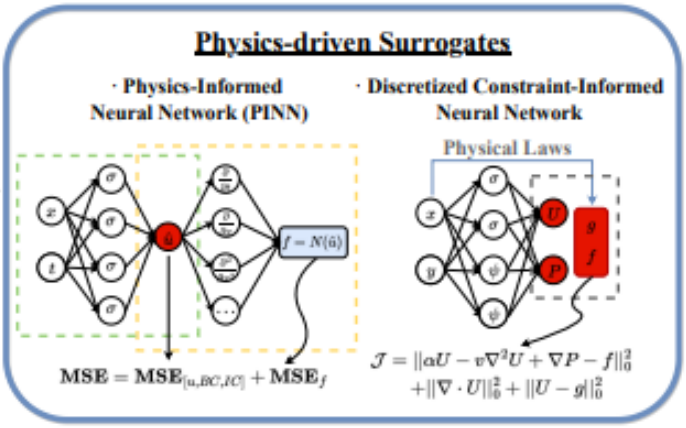


Figure 1.5 – Physics-driven surrogate models

1.2.5 Application of Artificial Intelligence in Computational Fluid Dynamics

AI algorithms can be divided into three main categories: supervised learning, unsupervised learning, and semi-supervised learning. Supervised learning involves labeled data being input into the model to obtain output results. The loss function is defined as the distance between the calculated value and the actual value [10]. In simpler terms, it can be conceived that training models is simply the refinement/minimization of the loss functions. Unsupervised learning has no labels or pretraining processes. Usually, this method is left for scenarios such as dimension reduction, clustering analysis, and anomaly detection to name a few. Semi-supervised learning is a mixture of the two previous methods.

In 2015 research into AI algorithms in the field of CFD was relatively small. However, since the growth in research and application has greatly increased. Now, there are many aspects of CFD that have been explored by authors attempting to optimize CFD with AI. Although there have been many improvements, AI models still have limitations. For example, since many experiments and training are done for specific scenarios, when it comes to generalized cases, the models fall short of the quality of typical CFD simulations [10]. Figures 1.6 and 1.7 below show the basic framework of neural networks and how they can be structured to be used in aerodynamic applications.

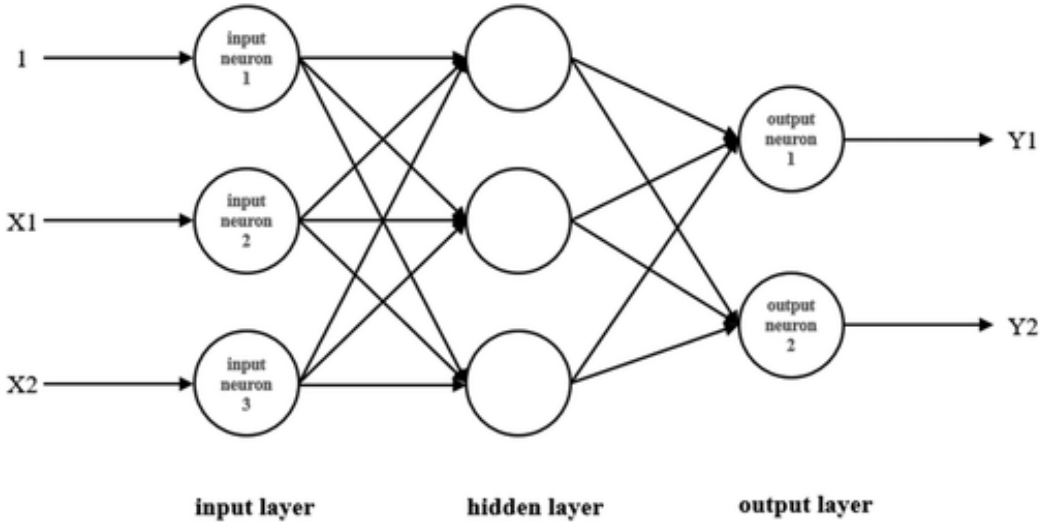


Figure 1.6 – Neural network structures

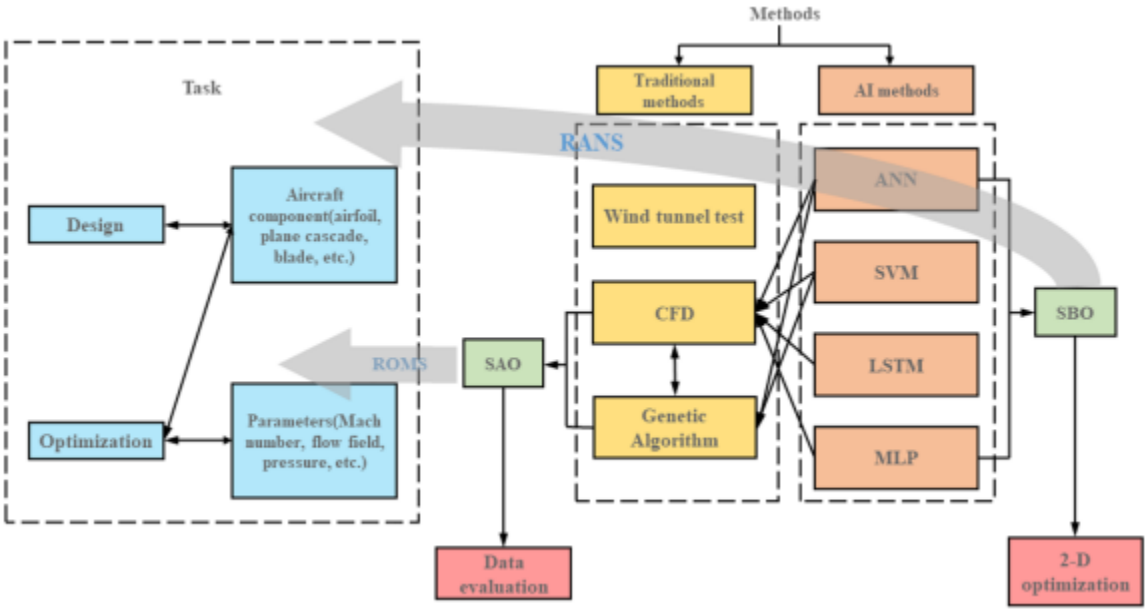


Figure 1.7 – Typical framework/structure of AI algorithms in aerodynamic models

1.2.6 Summary of Current Advancements and Limitations

Nearly all of the articles regarding the use of AI models to optimize CFD simulations talk about the current limitations of CFD. CFD is a great tool to use for understanding fluid flows given its vast range of applications. However, it also comes with its downsides. CFD faces many challenges mainly in terms of computational expensiveness and accuracy [11]. A need for real-time CFD simulation exists across a broad range of industries, such as in emergency management, wherein flow simulation aids in building evacuation plans in the event of a fire [12]. Companies are now turning to AI to more efficiently solve CFD simulations. Whether it be by replacing a few or all of the components of a CFD solver with neural networks. It is popular to use CNNs; however, the use of unstructured grids is unavoidable [13]. CNNs weakness lies in that it uses image-based training [6]. The best methods are those that incorporate the CFD data directly. In some cases, data-driven model prediction of the flow field was better than the predictions of baseline eddy viscosity and nonlinear eddy viscosity models [14]. Using data driven models not only makes simulations more accurate, in many cases incorporating dimensionality reduction creates significant improvements in computational speed and memory capacity [15].

Although many advancements have been made, AI still has some of the same limitations that current CFD programs have. For example, large scale 3-D simulations require large amounts of data in order to be trained. Even with the execution of the best DNNs the loss function still remains quite large [16]. Many authors agree that although basic experiments prove to be promising. These should not immediately replace classical methods that have matured over the past 50 or more years. If anything, these experiments raise more questions than they answer. It is still not clear what amount of data can be considered enough data to truly train the models [17].

1.3 Project Proposal

It's clear that even with the computational power and storage available in today's world CFD is still limited by computational costs. As computational power increases, so does human ambition. More complex simulations are being run every day. With it finer meshes are needed increasing computational costs. The introduction of AI brings with it the possibility to achieve near-instant simulations for fluid dynamics. It seems that most of the research that exists has focused mainly on using CNNs in which image-based training is used to train the model [6]. However, newer research shows that it's more beneficial to use data-driven training. The main reason for this is that image-based training can become computationally costly fairly easy. It uses uniform dimensionality to discretize the structure used. Where data-driven methods use CFD-based data in order to achieve dimensionality reduction to have a more refined structure in which dimensionality varies across critical areas [15]. This allows for faster simulation speeds. Additionally, using deep-learning models that incorporate PINNs allows for a decrease in the loss functions as it keeps the simulation within expected physical bounds, preserving the physics of mass, momentum, and energy.

The proposed project is to achieve flow field predictions and generations of a chosen NACA airfoil using data-driven deep-learning AI models. CFD-based training would be used in order to achieve a mesh that's more refined at critical areas. In order to reduce the loss function, reduce computational costs, and increase accuracy a PINN model would be used to ensure the model does

not add artificial noise to the results. This would keep the data bound by the physical laws of the Navier Stokes equations.

1.4 Methodology

In order to achieve this project, the first thing that needs to be done is to find an existing model that can be trained using CFD-based training. One of the most difficult parts of using AI is that it is unclear how much training it is needed in order to properly train a model. Therefore, it can be a hundred or a thousand datasets that need to be used in order to train the model. Additionally, there are many libraries that can be used to train models faster. Once the model is trained enough the next step would be to start generating flow fields and comparing them to benchmark results. Depending on the results, PINNs would have to be incorporated. It is important to note that given the time constraint the focus will be on 2-D cases only. If time allows it, then 3-D cases could be explored. However, this might prove difficult within the time constraint due to the massive computational costs that come with 3-D cases.

2. Existing Models and Applications

2.1 Currently Used Models

2.1.1 Convolutional Neural Networks How They Work and Their Downsides

Most Machine Learning (ML) models used for fluid flow prediction use some kind of CNN. However, the underlying algorithm used depends on the type of training that is being used to develop the model. For example, the leading method for training ML models to predict fluid flow was to use image-based training. In this case U-Net architecture was implemented since it performs pixel-level image segmentation. Image segmentation refers to dividing an image into different parts in order to identify specific objects [18]. This kind of algorithm is typically used in the medical field in cases where separating a tumor from healthy tissue is required.

The way U-Net architecture works is that an image typically in grayscale is fed into the network. The encoder then extracts increasingly abstract features through the application of convolutions and down sampling. The spatial size decreases with each level, while also increasing the number of feature channels, this allows the model to capture higher level patterns. The middle part of the algorithm is known as the bottleneck process. At this point the image is reduced the most that it can be. The decoder reconstructs the original image through upsampling. In order to maintain fine-grained spatial details, skip connections are used in order to combine encoder and decoder features. The final step is called final prediction where a convolution converts the refined feature maps into final segmentation maps. Each pixel is classified into foreground or background. The output has the same spatial resolution as the input image [18].

U-Net architecture is a type of CNN; CNNs use three-dimensional data for image classification and object recognition tasks [19]. Neural networks (NN) are a subset of machine learning, typically made up of node layers that contain an input layer, one or more hidden layers, and an output layer. Nodes are connected to one another and associated through weight and threshold. If the weight or threshold is high enough then the next node is activated, otherwise no data is passed to the next node within the network. Before the existence of CNNs, manual feature extraction methods were used in order to identify data within an image. CNNs have greatly improved the approaches used in image classification and object recognition tasks. Linear algebra methods such as matrix multiplication are used to identify patterns within an image. However, since they use linear algebra, high dimensional computations can result in high computational costs requiring many GPUs to train models [19].

CNNs have three main types of layers: convolutional layer, pooling layer, and fully connected (FC) layer. The convolutional layer is that main layer of a CNN. In the convolutional layer, the large majority of the computations occur. Convolutional layers require a few components: input data, a filter, and a feature map. Assuming an image is used, which is composed of a matrix of pixels in 3D, the input will have height, a width, and a depth that depends on the RBG value of the image. A feature detector, also known as a kernel or feature filter, moves across the receptive fields of the image looking for a feature. This is what is known as the convolution. The feature detector is a two-dimensional array composed of weights. The filter size is a square three-dimensional matrix that determines the size of the receptive field. When the filter is applied to an area of the

image a dot product is calculated between the input pixels and the filter, the result is then fed into the output. This repeats across the area of the image until the kernel has swept across the entire area. The result is known as the feature map, activation map, or a convolved feature. There are three hyperparameters that are set before training of the neural network can commence: the number of filters, stride, and zero-padding. Number of filter maps affect the output. The number of filters is the same as the resulting filter maps, which results in being the same number of depths that the output has. Stride is the number of pixels that kernel moved over the input matrix. It can be thought of as a step count. The greater the step count, the smaller the output. Typically strides of 2 or more are considered rare. Paddings are used when filters don't match the input data, setting elements that do not fall within the bounds of the input matrix to zero, so they do not affect the final output. This produces an output greater than or equal to the size of the input. There are three types of zero-padding: valid padding, same padding, and full padding. Valid padding or no padding makes it so that the last convolution is dropped or ignored if the dimensions do not match. Same padding ensures that the output layer and the input layer are the same size. Finally, full padding makes it so that the size of the output layer is larger by adding zeros to the border of the input [19].

The pooling layer or downsampling will follow the convolutional layer, performing dimensionality reduction. They work fairly similarly, in that, a filter will sweep across the input. However, instead of being populated by weights this kernel applies an aggregation function to the values inside the receptive field that will fill up the output array. The two types of pooling's that are used are: max pooling and average pooling. Max pooling sends the pixel with the maximum value to the output array. Average pooling calculated the average of the pixels and sends that value to the output array [19].

Finally, the FC layer. In this layer, the output nodes are directly linked to the nodes in the previous layer. It runs a softmax activation function that classifies the inputs producing a probability of 0 to 1.

One of the major issues that appear with the use of CNNs is the very same issue that it attempts to solve. As previously mentioned, CNNs use algorithms written in the mathematical language of linear algebra. The more complex the layers within the CNN architecture are the more computations that have to be run by the computer. While linear algebra provides a great way to solve these problems, it can also become computationally costly very quickly. Additionally, during the upsampling part of the CNN architecture it becomes possible for the algorithm to accidentally add values or take away values. This is a major issue within the use of CFD as mass, energy, and momentum must be conserved to accurately predict fluid flow behavior. One of the ways that engineers get around these issues is to implement data-driven CNNs. Through the use of data-driven CNNs it becomes possible to bound the model to physical laws. It also allows the training that is used to more accurately help the model capture the physics at crucial areas of the domain.

2.1.2 Examples of Data-Driven Convolutional Neural Networks

In November of 2021, a team from the German Aerospace Center published a paper in which they used a CNN model called DeepCFD in order to efficiently approximate solutions for non-uniform steady state laminar flows. They faced the same issue that many engineers do when using state of the art CFD solvers, high computational costs. In order to solve this problem, they sought to create a data-driven neural network. They created DeepCFD, a learning model based on CNN

architecture that provides solutions for velocity and pressure fields. They feed it ground-truth data of a channel flow around randomly shaped obstacles generated by their CFD solver [19].

They used the traditional CFD approach in order to give DeepCFD ground-truth data so that the model could be trained. The CFD solver they employed was OpenFOAM, an open source CFD solver. OpenFOAM solves the momentum equation in two separate steps. First, it provides an explicit solution for the velocity field based on the current pressure field at that time step. Secondly, it provides implicit corrections for the pressure field using information from the previous time step. This method is known as the Semi-Implicit Method or Simple Algorithm. This is repeated until the convergence occurs at a user-specified level of accuracy. This is part of CFD that becomes computationally costly. If a high-fidelity model is required, this means that high spatial resolution is also required. This increases the number of grid elements needed. Since each grid element needs to be solved for, this means the computational cost can increase rather quickly [19].

This is where DeepCFD takes over to make this process more efficient through the use of CNNs. Since the cases that were being analyzed were steady-state flows meaning that the solution does not change with time, this meant the input layer had to contain information regarding the geometric information of the flow or rather the geometric obstacle. In order to do this, they used the signed distance function (SDF). This provides the distance of individual points in the CFD grid to a given surface being used as an obstacle within the flow. Additionally, a multi-class channel with information regarding the flow region is provided. This channel includes 5 different categories: 0 is for the obstacle, 1 is for the free flow region, 2 is for the upper or bottom no-slip wall condition, 3 is for the constant velocity inlet condition, and 4 is for the zero-gradient velocity outlet condition [19]. Consecutive down sampling convolutional operations encode the input into a latent geometry representation (LGR), using transposed convolutions or deconvolutions a mapping between LGR, velocity field, and pressure field can be found. This is achieved by performing upsampling until the output is the same dimensional size as the input. The goal of the trained model is to achieve a minimization of the total error function. The team implemented a U-Net architecture in order to minimize the error and be able to develop a pressure field as well. The results they achieved can be seen below in figures 2.1 and 2.2 [19].

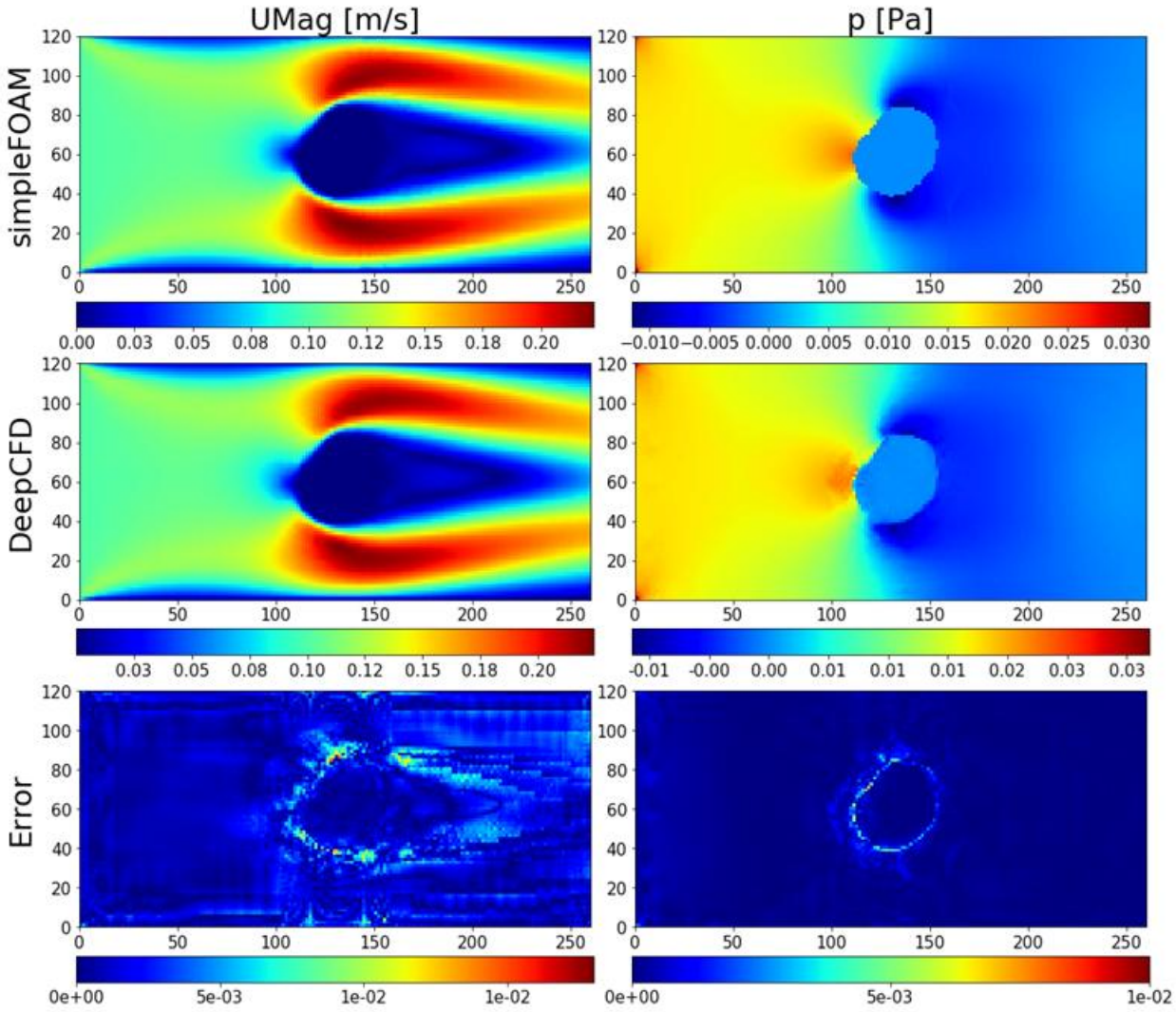


Figure 2.1 – Comparison between simpleFOAMs and DeepCFDs velocity magnitude and pressure as well as their absolute error for circular shape

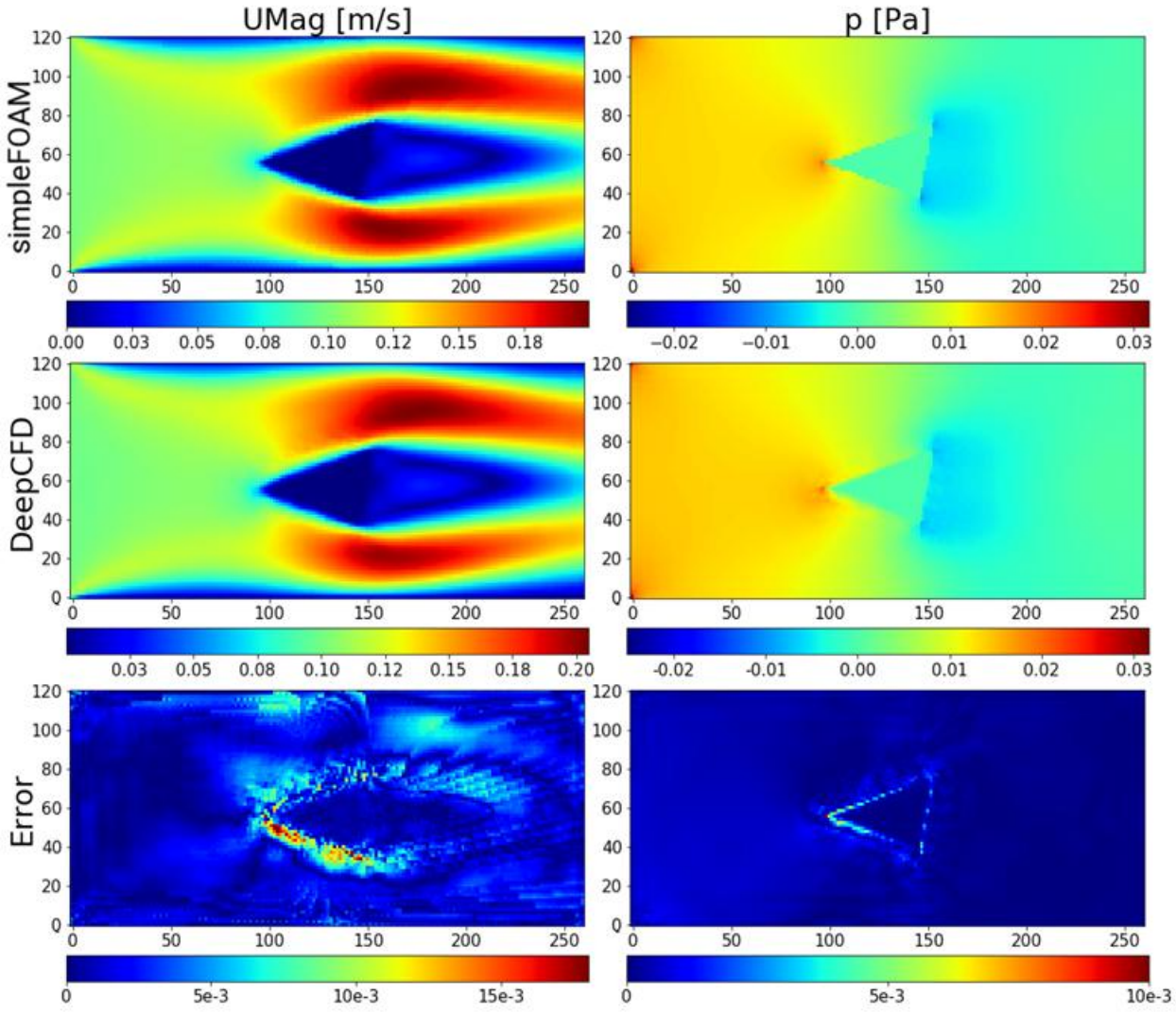
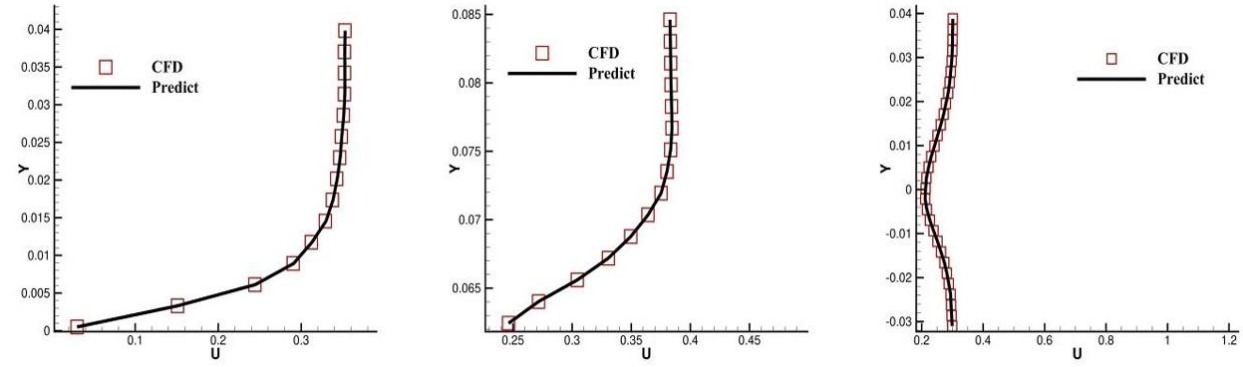


Figure 2.2 – Comparison between simpleFOAMs and DeepCFDs velocity magnitude and pressure as well as their absolute error for triangular shape

Figures 2.1 and 2.2 show the effectiveness of the DeepCFD in predicting the velocity and pressure fields of randomly generated obstacles while maintaining lower absolute errors. The team was able to achieve speedups of up to 5 orders of magnitude when using GPUs for the calculations.

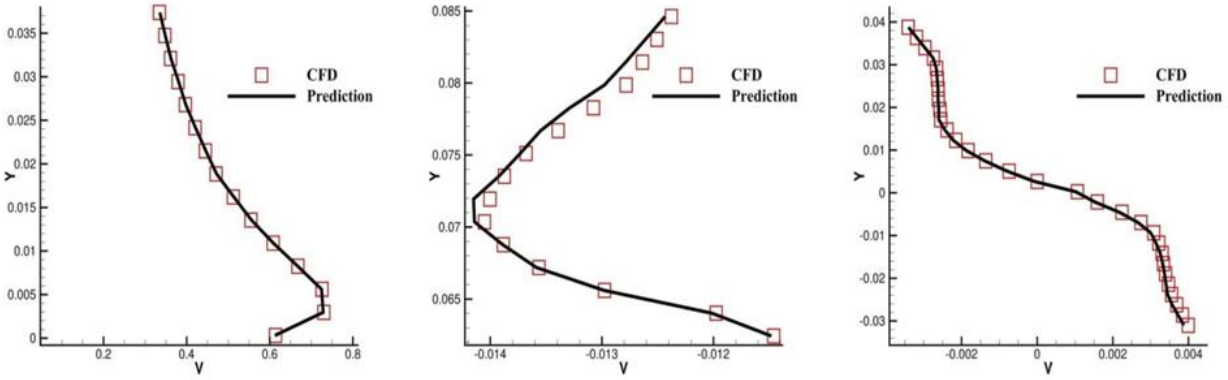
In another article published more recently by Journal of Applied Fluid Mechanics a team used a data-driven machine learning approach for turbulent flow field prediction. One of the downsides that this team found when using CNNs was that during the upsampling there was interpolation happening that would affect the true results of the prediction. In order to mitigate this issue, they proposed using CFD data instead. Additionally, they also used the control volume method in order to discretize the governing flow equations during the data generation process to ensure the preservation of mass, energy, and momentum. The use of CFD data allowed them to use a simpler structure, leading to significantly reduced training times [6]. They used a supervised training method since both the input and output are known. The inputs for this method were the airfoil

geometry, the Mach number (M), and the α. The output was the flow field with the flow parameters stored in the computational cells. Once the training was completed, the prediction could be completed in a few seconds with high levels of accuracy. In order to measure the accuracy of the model compared to the CFD results, the authors decided to measure the velocity profile at three points along the airfoil. The points they used to measure were the leading edge, the middle/ upper face, and at the wake region. These points can be seen in figures 2.3 and 2.4 represented by x = 0 of the chord length, x = 0.5 of the chord length, and x = 1.2 of the chord length.



(a) x=0 (b) x=0.5 (c) x=1.2

Figure 2.3 – Velocity profile of predicted model vs CFD results at M = 0.27, α = 3.5



(a) x=0 (b) x=0.5 (c) x=1.2

Figure 2.4 – Velocity profile of predicted model vs CFD results at M = 0.44, α = 2.41

From figures 2.1 – 2.4 it becomes clear that using CNNs with data-driven approaches and implementation of U-Net architectures really good prediction models can be achieved. The only downside that remains in these types of approaches is that since models are trained for specific cases having broader applications becomes harder. The model would require extensive training, or a newer model would have to be built for each specific case.

2.2 Chosen Project Model

Based on the available information and previous research done into using AI for flow field prediction it is clear that the best methods are those that implement data-driven techniques along with CNNs and U-Net architectures. The goal for this paper is to be able to use an existing model and train it using data-driven methods so that the error can be minimized as much as possible. This will also help retain the underlying physics as these methods allow for the conservation of mass, energy, and momentum. Additionally, this paper will attempt to implement a U-Net architecture to help create a more accurate model. The hope is that when compared to traditional CFD solvers, this new model will be faster and more efficient while maintaining accuracy.

3. Chosen Model

3.1 Preliminary Decision

Choosing a model to use is a complex task. Initially different models were considered, such as ResNet-18, DeepCFD, and the potential use of Gemini or ChatGPT to create a new model. However, the use of Gemini or ChatGPT could prove to be tricky for many factors including copyright laws, academic integrity, and feasibility. Perhaps someone with more advanced coding skills could use ChatGPT or Gemini to help them debug their own code. However, someone who is not as advanced in coding might take more time than the project allows to get working results. An advanced programmer might be more successful in using these tools for debugging. A less experienced programmer might find this route to be more challenging with a steeper learning curve and potentially impact the project deadline.

ResNet-18 was a promising option. It is seen by many as the benchmark for convolutional neural networks. It is capable of classifying images into thousands of categories, serving as strong backbone for computer vision tasks. ResNet-18 is a part of the ResNet architecture which refers to the residual network. It is a deep learning architecture that implements skip connections to allow training of deeper neural networks. ResNet-18 is referred to as the starting point because it balances speed, accuracy and simplicity. For higher accuracy, ResNet-34, ResNet-50, and ResNet-101 can be used. [20]

DeepCFD is an existing deep learning model that uses CNNs to provide fast approximate solutions to fluid dynamics problems. The way the model works is that it is trained on ground truth data from state-of-the-art CFD solvers such as OpenFOAM. Once the model is trained it can rapidly predict the complete velocity and pressure fields of new geometric configurations. It has a speed of up to three orders of magnitude with low error rates. DeepCFD was the model that was chosen since it already has CFD applications and focuses primarily on 2D steady-state problems.

3.2 Running DeepCFD

The AI model DeepCFD was downloaded from a GitHub link that was provided by the author [19]. Since the model was written in python, python had to be downloaded in order to run the model. Additionally, python can be run and operated through the command terminal. However, it is much easier to write code using an Integrated Development Environment (IDE). The chosen IDE for this project was PyCharm. PyCharm was a great option to use because it allowed cloning of GitHub repositories into the IDE. This greatly simplified the extraction of the AI model. There were a few issues that needed to be fixed in order to run the model. Since the model was made a couple of years ago there were a couple of inconsistencies within the code that needed to be fixed. Mostly, setting certain variables to be compatible with newer syntax formats. Once the model was downloaded the model was trained using toy channel data sets that were provided by the author of the model. This was done to test out how the model worked and also to try and compare results with those of the published paper. Initially the model was trained for 200 epochs, then 2000, and finally 500.

In order to train the models a few parameters had to be populated to direct the model's training behavior. The batch size is the number that determines how many simulation samples the model looks at before updating the weights. The number 32 was chosen for this as it is considered to be a sweet spot. Smaller batch sizes are considered noisy and can lead to an erratic training path, bigger batch sizes can lead the model to converge faster to an erroneous solution. The filters were set to be 8,16,32,32; this was done to match the filters that the authors had used for their best results. This allows for hierarchical learning in where filters 8 and 16 are used to detect simpler things such as the direction of the inlet flow and the edge of the object. The deeper layers 32 and 32 are there to detect the more complex physics, the changes in velocity, the changes in pressure, the way the top and bottom of the flow interact with one another, and most importantly, the turbulent behavior of the flow once it becomes detached. The kernel size was set to 5. Kernels are square matrixes meaning a 5x5 matrix was used. The job of the kernel is to scan over the image, thus by using a 5x5 matrix the physics can be better captured. This is because a bigger matrix can capture spatial gradients much easier. The learning rate was set to be 0.001, this can be thought of as the step size that's taken when the model is trying to minimize the error. This number is known as the Adam Optimizer, which is industry standard as it ensures there is no overshoot in the solution, but also allows for reasonable training times. Additionally, a training log was created to quantify the behavior of the training at every single epoch. Figures 3.1 – 3.3 below show the results comparing the ground truth data vs the predicted results of the models that were trained for 200, 2000, and 500 epochs respectively.

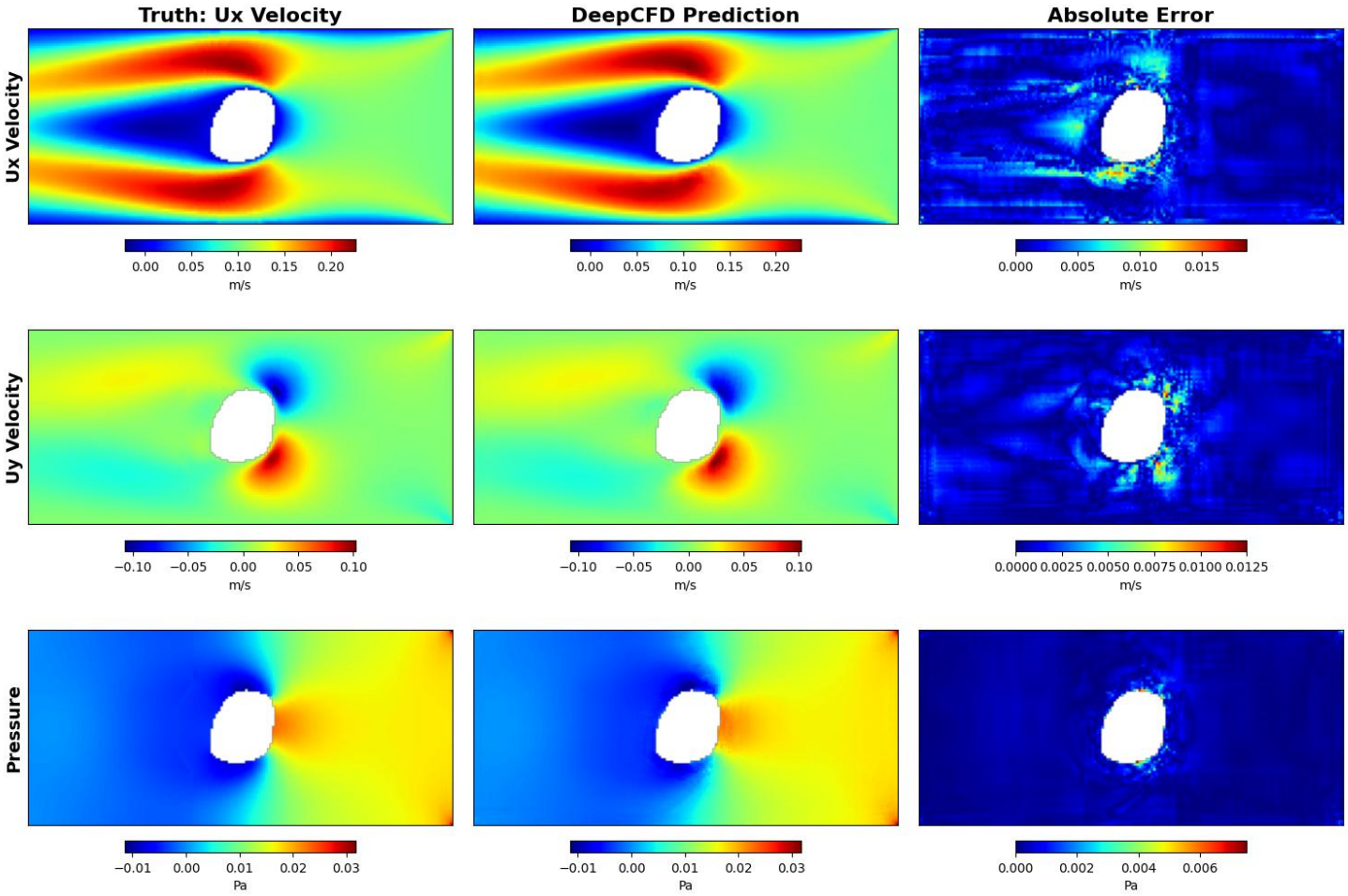


Figure 3.1 – Ground truth vs predicted results after training for 200 epochs

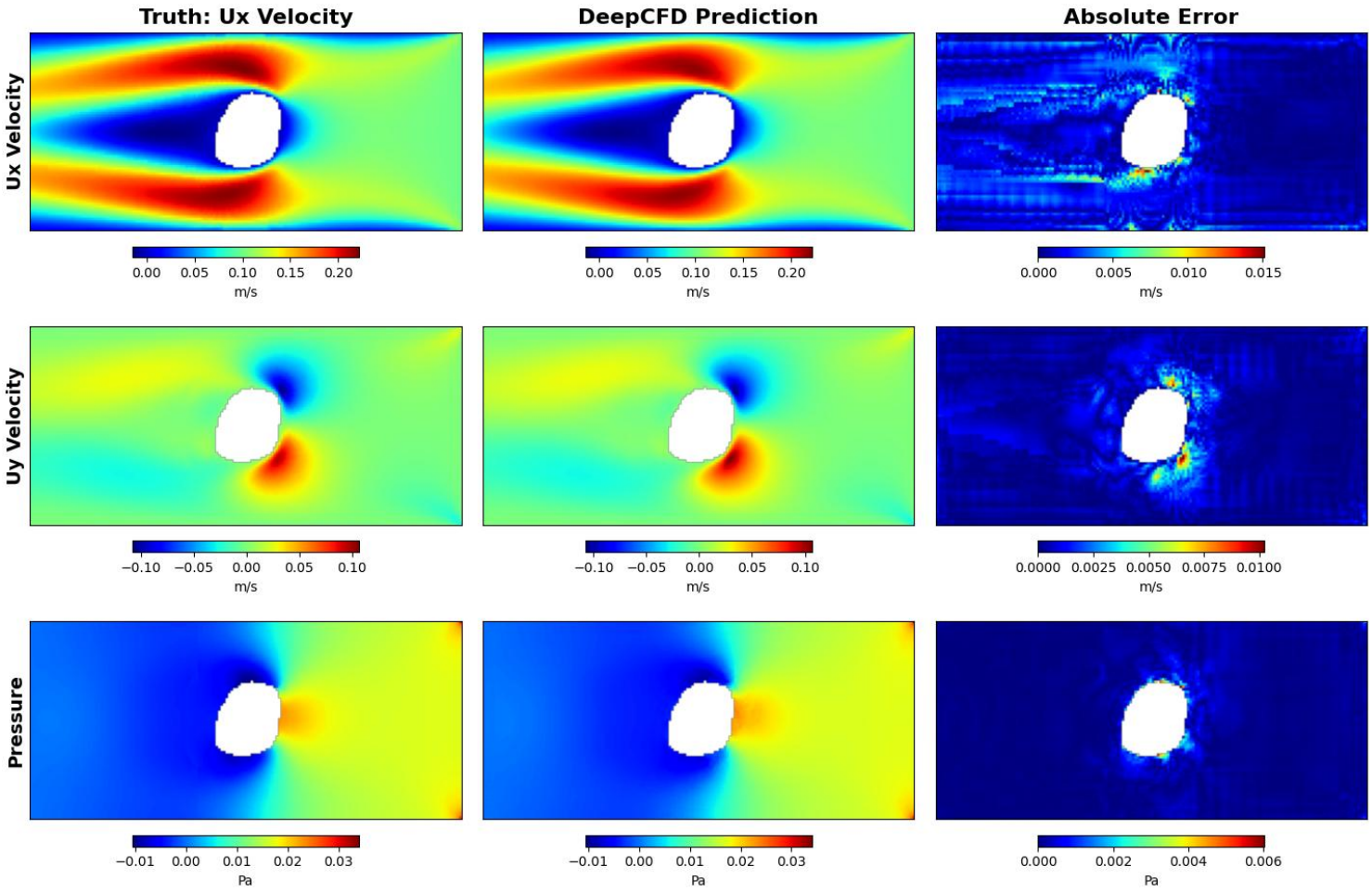


Figure 3.2 – Ground truth vs predicted results after training for 2000 epochs

Initially the model was trained for 200 epochs. Figure 3.1 shows the results of that initial training that was done. It is astonishing to see just how far humanity has come. The AI model trained well. The comparison between the ground truth data and that which was predicted by model is fairly similar. To the naked eye, it would seem that the model is precisely predicting the real-world fluid flow. However, it is known that most of the failures in CFD occur at the microscopic level where the details are far too small to truly capture the physics. Therefore, to be sure that the model is accurate the absolute error can be taken to see how far the predicted model is from the ground truth data. This can be done simply by incorporating a simple equation such as eq 3.1 in which the absolute error between the ground truth data and the predicted model is calculated.

$$Absolute\ Error = |G_t - P_v| \tag{3.1}$$

Looking at the absolute error, it becomes clear that the model does struggle when it comes to capturing the physics closer to the boundary layer and the wake from the detached airflow. To fix this issue a newer model was trained for 2000 epochs. The results can be seen in figure 3.2. Comparing the U_x, U_y velocities, and the pressure from the flow fields, a similar result to the 200-epoch model can be seen. The data seems to match perfectly; it isn't until the absolute error is analyzed that again the intricacies of capturing the physics at the microscopic scale can be seen.

Again, the absolute error shows that the error lies within the wake region. In order to see what is happening the training log had to be analyzed. Based on the results from the final epoch of the 200-epoch model, it could be seen that the Validation MSE (Mean Squared Error) was 0.648 which was lower than the Training MSE 2.113, which is something that is typically seen very early in training. The issue with this model was that since it is so early in its training the errors are considered to be high in absolute terms. The good thing is the model has not "memorized" the dataset yet, but it has also not learned the complexity of the physics hidden within the wake. This becomes apparent when looking at the absolute error. Now looking at the training log of the 2000 epoch model something different can be seen.

The model that was trained for 2000 epochs had a very low Training MSE of 0.164. This means that it is nearly perfect on every single object that it has seen. However, the Validation MSE is 1.416, significantly higher than the Training MSE. This means that overfitting is occurring, this occurs when the model has "memorized" the training noise. This tells the story that the model that trained for 200 epochs was learning but had not reached a point of memorizing the dataset yet. Meanwhile, the model that trained for 2000 epochs showed better results, but because it had memorized the dataset rather than trying to learn the physics of the wake region. Thus, a new model was trained for 500 epochs in hopes that this would allow the model to have a lower Training MSE while not giving it so much time that it "memorized" the training dataset. Figure 3.3 below shows the results of the model that was trained for 500 epochs.

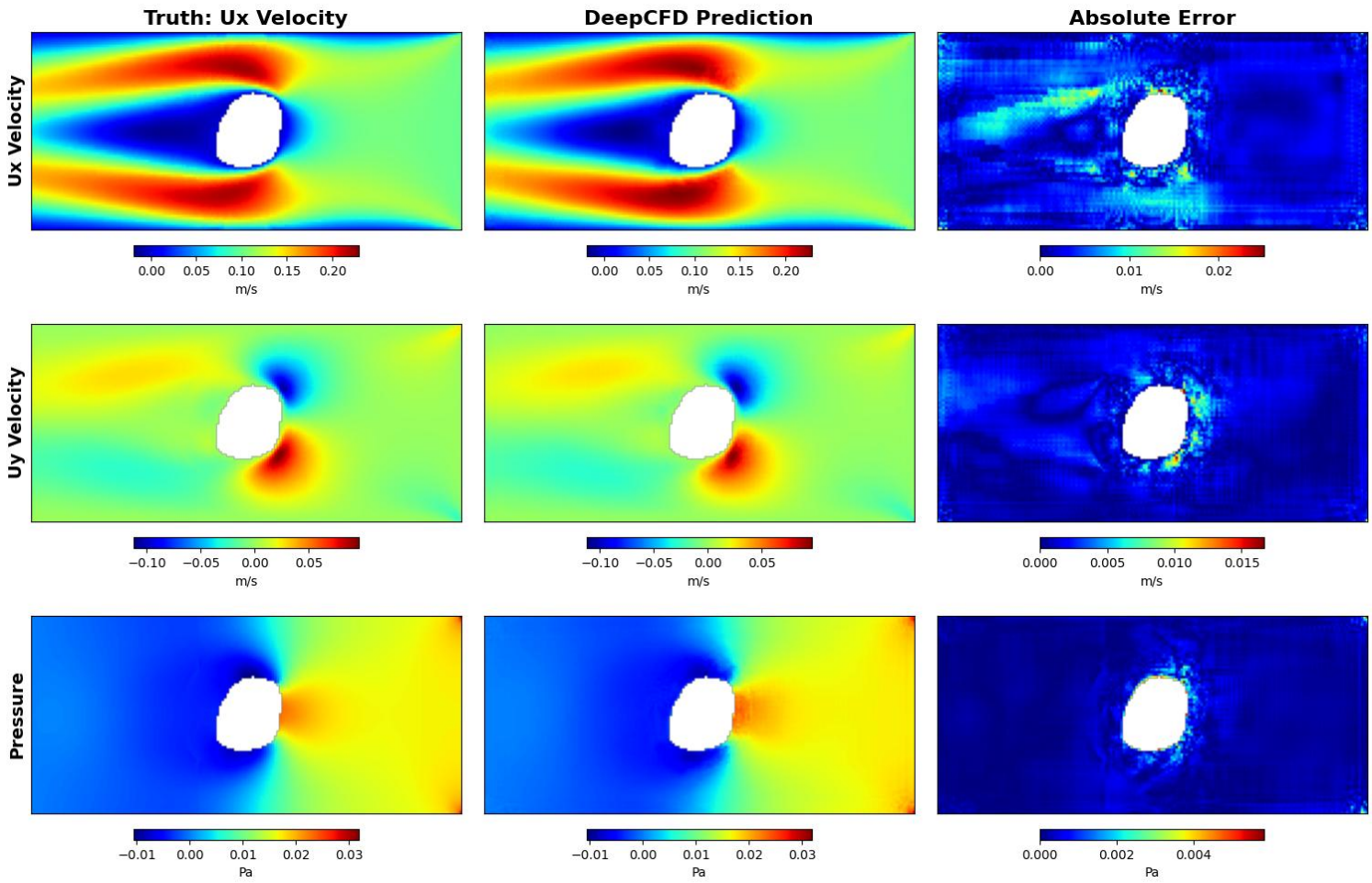


Figure 3.3 – Ground truth vs predicted results after training for 500 epochs

The 500-epoch model followed the similar trend that was seen with the 200-epoch model and the 2000-epoch model. To the naked eye, the velocity and pressure results seem to be identical. However, the absolute error still seems to be quite prominent within the wake region. Analyzing the training log tells the story underneath. The Training MSE is 0.95 and the Validation MSE is 2.90. This Validation MSE is even higher than it was for the 2000-Epoch model. This means that the model is currently in an unstable mode. Where the model has begun to memorize specific objects; however, it gets confused with certain objects when it sees the validation. Even though this doesn't give a better model than either the 200-epoch model or the 2000-epoch model it does give insight into how to handle the next steps of the project.

3.3 Next Steps

This initial toy dataset was meant to serve as a way to learn how to interact with the DeepCFD model and to learn how to use it. Three models were trained and results were able to be generated from each model. The use of incorporating training logs helped with understanding the underlying issues. It was discovered that the models were reaching a point where they began to "memorize" the training dataset. However, they did not have enough time to truly learn the physics that occur at the wake region of the fluid flow. Compared to the methods that the authors used, it becomes

clear that early stopping has to be introduced into the training of the model. The best model seems to exist somewhere between 200 and 500 epochs for this given dataset. Training a model to be able to predict the velocity and pressure flow fields of an airfoil will have to include an early stopping method. Additionally, the dataset used to train the model will have to be significantly ample. This is to avoid the issue that was encountered where the models "memorize" the dataset. It is much easier to remember a dataset of 500 than a dataset of 5,000.

The next steps for the project are to create a large ground truth dataset inside of Ansys Fluent. This data will then be formatted and exported into DeepCFD. Once the ground truth dataset is inside of DeepCFD a new model can be trained specifically for airfoils. This model will have early stopping incorporated into it as well to ensure that the model does not "memorize" the dataset if the dataset is not sufficiently big enough. Additionally, regularization can also be used in case the dataset is not big enough. One method would be to increase the weight decay, this forces the weights to stay small, forcing the model to find smoother patterns. Another method would be to add dropout. Dropout occurs during training, where a percentage of neurons in a layer for a batch are randomly turned to 0. This is considered to be a more robust method for training as it forces each individual neuron to learn to make connections that reach the solution as opposed to relying on any specific neuron to "memorize" specific solutions. When the model is tested against the validation set the dropout is turned off allowing all neurons to be on, resulting in a more accurate model.

A simple way of thinking about dropout method would be to picture a tree using its roots to get its nutrients. If a tree relied solely on one or two roots to get its nutrients, chances would be that if the tree got sick and those roots were affected it would die. However, because each individual root can provide the tree with nutrients if part of the tree gets sick and it loses a number of roots, the remaining roots would still be able to provide it with all its necessary nutrients.

Once the model is trained a comparison would be made similar to the one that the authors did for the toy dataset. Where the U_x, U_y, and p results could be compared to the ground truth data. Additionally, the absolute error would also be plotted to show how accurate the model is. Once this is done, plots can be generated and the discussion and future work section could be completed for this project.

4. Current CFD Theory and Method for Airfoils

4.1 CFD Theory

Creating a CFD simulation is a task that relies heavily in complex mathematics. Yet, modern software almost hides this complexity from users. It exists within the solver of a CFD program. Once the geometry, the mesh, and the boundary conditions are established, the solver takes over. The sole job of the solver is to solve Partial Differential Equations (PDEs). CFD solvers use the Navier-Stokes equations of conservation of Mass, Momentum and Energy in order to solve fluid dynamics problems. These equations can be seen in equations 1.1 – 1.5. The solver is programmed with these equations and treats every single cell as a control volume. Since the conservation laws are being applied the solver must make sure that the stuff coming into the volume is equal to the stuff leaving the volume.

What makes solving the Navier-Stokes equations so difficult is that they're non-linear coupled equations. For example, the pressure is dependent on velocity, temperature, chemical composition of the fluid being analyzed, and energy in the system; additionally, all of these variables are also dependent on the pressure values. There still exists no analytical solution for the Navier-Stokes equations. Instead, numerical approximations must be implemented in order to solve them. The easiest way to convert a coupled system composed of 5 PDEs into simpler math is to create a matrix. The solver will then use algorithms to approximate a solution until the residuals are small enough or a user-imposed limit is reached to consider the simulation correct.

There are two types of methods that can be used Spatial and Temporal Methods. Spatial Methods take a look at how certain parameters change from one point in space to another. Temporal Methods take a look at how parameters change from one point in time to the next. There are many ways that these changes can be approximated.

One spatial approximation method that may be used is the Central Approximation Method. This method seeks to look at what is happening one point in front of the current position and what was happening one point behind the current position. The average of these two is assumed to be what is currently happening. Equation 4.1 demonstrates this.

$$\frac{\delta\phi}{\delta x} \approx \frac{\phi_{i+1}-\phi_{i-1}}{2\Delta x} \tag{4.1}$$

Another Spatial Method that can be used by CFD solvers is the Three-point Central Approximation Method. Usually this is used for calculating things such as diffusivity or viscosity which require second derivatives. Similar to the first method it looks forward in both directions; however, in this case three points are being studied. Thus, the resulting equation is equation 4.2.

$$\frac{\delta^2\phi}{\delta x^2} \approx \frac{\phi_{i+1}-2\phi_i+\phi_{i-1}}{\Delta x^2} \tag{4.2}$$

The Temporal Methods are responsible for analyzing the change with respect to time. One of these methods is the Leap Frog Method. This method Leaps over the current time step to the next time step, it uses the slope of the current time step and applies it to the time step before the current time step. This method is known for preserving energy very well and is used in wave propagation and astrophysics. Equation 4.3 below shows the Leap Frog method.

$$\phi_{t+1} = \phi_{t-1} + 2\Delta t \times f(\phi_t) \tag{4.3}$$

Depending on the simulation settings that are selected for the underlying physics the solver will use different methods to approximate the solutions. Once the solutions are approximated the interpreter will perform interpolation to predict what values should be in between the approximated values. This creates a smooth numerical transition between approximated values. These numbers are then mapped to a visible spectrum usually from blue to red to indicate low values to high values respectively. At this moment in the process is when vector mapping also takes place. The interpreter takes the calculated values and directions to draw out arrows that follow the approximated flow calculations. Finally, images can be generated, and conclusions can be reached by researchers.

4.2 CFD Setup

Chapter 3 showed the attempt at replicating the results that the authors of DeepCFD achieved. Since the results were close to the author's attempt the next step is to move on to the more complex case of studying airfoils. This document's purpose is to be able to predict and generate velocity and pressure flow fields for airfoils at different angles of attack and different conditions using the DeepCFD framework. To compare current CFD and AI model predictions, current CFD methods must be used for an airfoil at different angles of attack to serve as a benchmark comparison for the AI model. This will also serve as the ground truth data to train the AI model.

Most state-of-the-art CFD programs work quite similarly. First a CAD model of the geometry in question must be created. This can be done either in the CFD program itself, such as OpenFOAM, StarCCM+, or Ansys Fluent; or in an independent CAD software program such as Inventor or SolidWorks. If the CAD model is done outside of a CAD program, the model must be imported into the program for use within that CFD program. Once the CAD model is inside of the CFD program and working the next step becomes to set up the mesh. The mesh is the process by which the CAD geometry is divided into small finite discrete areas or volumes, depending on the case being analyzed. Getting a good mesh depends on a couple of factors; some that come with experience working with CFD programs and real-life scenarios. For example, the size of the mesh should vary depending on where it is located within the domain (area where flow is being analyzed). Mesh at critical points within the domain should be finer than mesh at areas of less importance. For example, in the case of an airfoil the mesh at the boundary layer should be considerably finer than the mesh between the walls and the airfoil itself. The mesh behind the trailing edge of the airfoil should also be considerably finer. This is because at the trailing edge of the airfoil the flow becomes separated and turbulent. It is important to have finer mesh at this area in order to capture the physics of the turbulent air. Additionally, if a 3D case was being analyzed,

for example, a wing. Finer mesh would be used with the tips of the wing, since physics says that the pressure difference from the bottom and top of the wing would create vortices.

Once the mesh is finalized, the boundary conditions can be set. The boundary conditions allow the user to tell the program what physical conditions the model is undergoing. This part is a bit different depending on the type of CFD program that is being used. However, they all follow the same logic. Depending on the simulation the pressure/velocity inlet and outlets have to be named. Once named, the physical conditions at these points can be set. For example, at a velocity inlet the velocity might be set to 300 m/s. This would let the program know how fast the air is coming in at the inlet. This step is also when the wall boundary conditions are set. For example, if analyzing an airfoil, the airfoil geometry is set to be a wall, this way the simulated air will not go through it. This is arguably the most important step in CFD, because if the conditions are set up incorrectly, there is no way for the program to know what the user is trying to achieve. It only knows the physics it was given; therefore, it is easy to set up erroneous conditions and get the wrong results.

Additionally, this is the step where the physics models are chosen for the simulation. For a simple 2D steady – state simulation, a K-Omega SST physics model can be used to effectively model the airflow. Getting the data from the simulation works a bit differently depending on the software being used. The researcher might use dots or streamlines to capture the data, or in some cases the geometry in question can be selected and the program will have automatic ways to find lift coefficients (C_l), drag coefficients (C_d), and pressure coefficients (C_p). Additionally, velocity and pressure scenes could be used to visualize the data and the fluid flow around an object. Once the desired data is acquired, professionals will typically compare the data to theory, published papers or some other form of benchmark data that can help them validate their findings. Often this is a step that if overlooked could result in costly engineering mistakes.

4.2.1 CAD Setup

To better explain the process behind running a CFD simulation for an airfoil. Ansys Fluent will be used to simulate a NACA0012 airfoil at 5 different αs: 0°,2°,5°,8°, and 10°. Additionally, this will serve as a quick initial trail run for exporting the data into DeepCFD to make sure that data can be exported and understood by the DeepCFD architecture to eventually run the calculations.

The first step is to start with the geometry. As explained above in section 4.2 certain CFD programs will already have internal CAD softwares, this is true for Ansys Fluent. To begin creating the airfoil inside of Ansys Fluent, the Ansys workbench must be opened. Next, under the analysis systems Fluid Flow (Fluent) must be brought over to the Ansys Project Schematic. By right clicking on Geometry and selecting Edit Geometry in Design Modeler the desired CAD environment can be brought up. The best method to creating the desired airfoil shape is to use databases such as airfoil tools website [21] to get x and y – coordinates that make up the top and bottom shape of the airfoil. Once these points are downloaded, they can easily be loaded into the CAD environment by clicking concept, 3D curve and selecting the appropriately named file containing the data. Clicking Generate will generate a curve that displays the top and bottom shape of the airfoil. However, there might be one small issue. These shapes are usually not connected at trailing edge of the airfoil. To fix this issue a line that connects the top and bottom of the airfoils trailing edge needs to be generated. This is achieved by clicking on concept and choosing the lines

from points tool. Once the two points are selected the airfoil shape can be closed off. To create a two – dimensional surface from these edges, the tool surface from edges can be selected under the same concept tab. This will generate a two – dimensional surface that represents the NACA0012 airfoil. Figure 4.1 below shows these results.

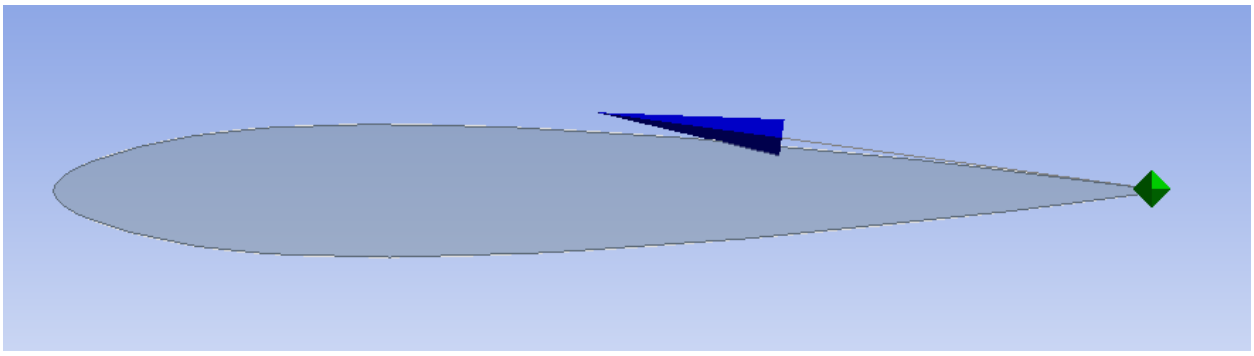


Figure 4.1 – Resulting NACA0012 airfoil surface

The next step is to create the air domain; essentially the surface around the airfoil where the air will be flowing through. In order to do this a new sketch must be created. The first shape to be drawn is an arc that will represent the inlet where the airflow comes through. To do achieve this inside the sketch, the arc by center option under draw must be selected. Next, under dimensions the general option is selected. This will allow the researcher to select how far away that arc will be from the leading edge of the airfoil. Usually, 15-20 chord lengths is a good benchmark. The chosen distance for this simulation was 15 chord lengths. The next step was to create the top, bottom and outlet. To do this, simple horizontal lines were drawn from the two ends of the arc and given a dimension of 25 chord lengths. Finally, the outlet was simply the line connecting the two previously made lines. Figure 4.2 below shows the resulting geometry. To create the actual surface the option surface from sketch was selected under the concept tab.

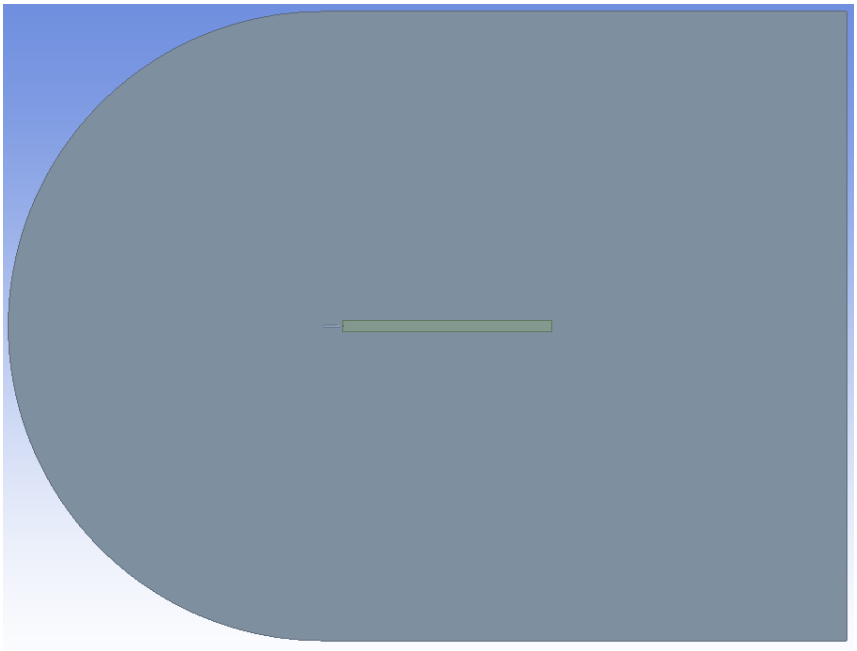


Figure 4.2 – Air domain geometry around the airfoil

Great engineers try to predict obstacles that they will encounter in the future and create early processes that will make jumping over these hurdles much simpler in the future. One of the harder things about capturing the physics of airfoils is being able to properly capture the wake behind the airfoil when the air becomes detached. To solve this problem, another geometry was added. A simple rectangle was added at the trailing edge. The rectangle had a width of 1 chord length and a length of 10 chord lengths. This geometry can be seen in figure 4.3. To create the surface, the same surface from sketch option that was used for the air domain was implemented. However, to avoid any conflicts between geometries this geometry was set to be added frozen instead of added material. This allows the user to use the surface, but it doesn't interfere with the air domain and the airfoil.



Figure 4.3 – Rectangular body of influence for mesh refinement

The next step was to subtract the airfoil from the air domain in order to create the boundary where air will not be able to go through. This is fairly simple to do. Underneath the Create tab there is a Boolean option, this option can be changed to Subtract. The air domain surface is picked as the main body, and the airfoil surface is used as the subtracting feature. Once this is generated the result is the surface body of the air domain with the airfoil shape being taken out of it. Figures 4.4 and 4.5 show the results.

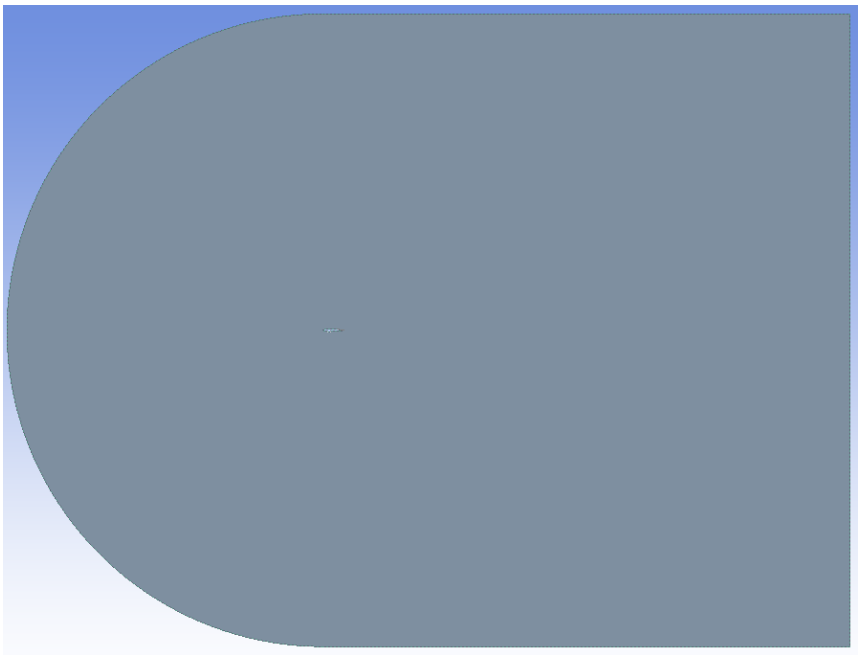


Figure 4.4 – Far away view of final geometry

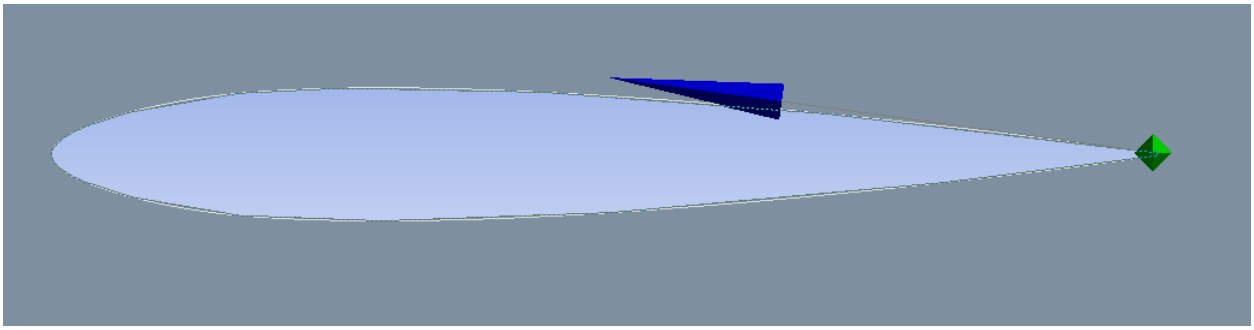


Figure 4.5 – Close up view of final geometry

Finally, the last thing to do is to create the named selections. This must be done to tell the program where the inlet, outlet, top, bottom, and airfoil are located. This is simply done by selecting the appropriate edges and right clicking to select the named selections option and naming them accordingly.

4.2.2 Meshing

The meshing step is arguably the most important. Detailed geometry is great but means nothing if the mesh does not capture the appropriate physics. Luckily the geometry that is being used for this project is not overly complicated. First step is to look at the global mesh settings. Table 4.1 below shows the chosen preferences for the global mesh parameters. These settings are responsible for setting up the parameters for the overall mesh structure. To make refinements, more specific Inflation and Sizing parameters can be implemented as shown below on Tables 4.2 – Table 4.5.

Table 4.1 – Global mesh settings

Display	
Display Style	Use Geometry Setting
Defaults	
Physics Preference	CFD
Solver Preference	Fluent
Element Size	5 m
Export Format	Standard
Export Preview Surface Mesh	No
Sizing	
Use Adaptive Sizing	No
Growth Rate	Default (1.2)
Mesh Defeaturing	Yes
Defeature Size	Default (2.5E-003m)
Capture Curvature	Yes
Curvature Min Size	Default (5.E-003m)
Curvature Normal Angle	Default (18.0°)
Capture Proximity	No
Bounding Box Diagonal	50.0 m
Average Surface Area	554.13 m^2

Minimum Edge Length	2.52E-003 m
Inflation	
Use Automatic Inflation	None
Inflation Option	First Layer Thickness
First Layer Height	1.E-003 m
Maximum Layers	2
Growth Rate	1.2
Inflation Algorithm	Post
View Advanced Options	Yes
Collision Avoidance	Layer Compression
Fix First Layer	No
Gap Factor	0.5
Maximum Height Over Base	1
Growth Rate Type	Geometric
Maximum Angle	140°
Fillet Ratio	1
Use Post Smoothing	Yes
Smoothing Iterations	5
Advanced	
Number of CPUs for Parallel Part Meshing	Program Controlled
Straight Sided Elements	
Rigid Body Behavior	Dimensionally Reduced
Triangle Surface Mesher	Program Controlled
Topology Checking	Yes
Use Sheet Thickness for Pinch	No
Pinch Tolerance	Default (4.5E-003 m)
Generate Pinch on Refresh	No
Sheet Loop Removal	NO
Automatic Methods	
Sheet Body Method	Quad Dominant
Sweepable Body Method	Sweep

Table 4.2 shows the use of Inflation controls to refine the mesh near the boundary layer of the airfoil. The inflation control encapsulates the airfoil geometry in layers that start out very small and increase in size until the last layer matches up with the size of the rest of the cells outside the inflation. Figures 4.6 and 4.7 below show the resulting mesh.

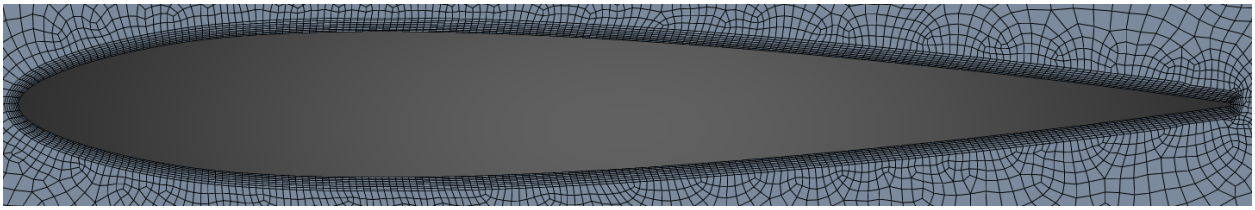


Figure 4.6 – Inflation mesh around the airfoil.

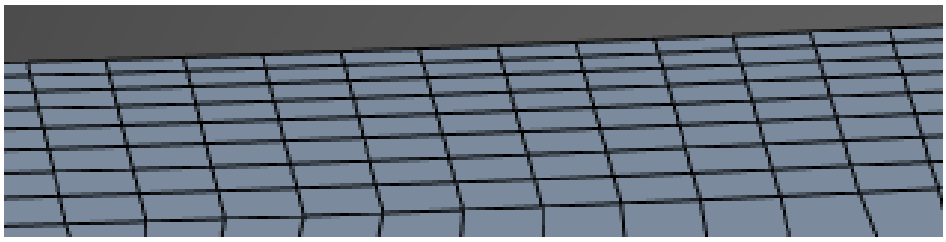


Figure 4.7 – Close up of inflation mesh around the airfoil

Table 4.2 – Mesh settings for inflation refinement around the airfoil

Scope	
Scoping Method	Geometry Selection
Geometry	1 Face
Definition	
Suppressed	No
Boundary Scoping Method	Named Selections
Boundary	Airfoil
Inflation Option	First Layer Thickness
First Layer Height	1.E-003 m
Maximum Layers	8
Growth Rate	1.1
Inflation Algorithm	Pre

The following refinements were sizing refinements that were implemented to the mesh around the airfoil, the wake region behind the trailing edge of the airfoil, and the trailing edge to ensure smooth transition due to tricky mesh geometry from so many refinements. Table 4.3 shows the sizing mesh settings for the sizing refinement around the airfoil to ensure smooth transition between the inflation layers, and the global mesh cells. Table 4.4 shows the sizing mesh settings for the wake refinement behind the trailing edge of the airfoil. This was done to ensure that the mesh was finer at this area to allow the CFD solver to capture the physics at this point. Table 4.5 shows the final size refinement done to fix the mesh transition at the trailing edge of the airfoil. This was done, because the inflation layers only set one cell at the back of the airfoil. Thus, as the growth rate increased the size of the layers the width of those cells increased to a point where they did not match the size of the wake refinement cell size. Thus, by diving the trailing edge into two partitions, it ensures a more refined and smoother transition between these cells. Figure 4.8 shows the resulting mesh near the airfoil. Figure 4.9 shows the refinement of the mesh at the wake region. Figure 4.10 shows a close up of the resulting mesh transition from the inflation layers to the wake refinement after the division was established.

Table 4.3 – Mesh settings for sizing around the airfoil

Scope	
Scoping Method	Named Selection
Named Selection	Airfoil
Definition	
Suppressed	No
Type	Element Size

Element Size	5.E-003 m
Advanced	
Behavior	Soft
Growth Rate	Default (1.2)
Capture Curvature	No
Capture Proximity	No
Bias Type	No Bias
Blend to Neighbors	Yes

Table 4.4 – Mesh settings for wake refinement behind trailing edge of the airfoil

Scope	
Scoping Method	Geometry Selection
Geometry	1 Body
Definition	
Suppressed	No
Type	Body of Influence
Bodies of Influence	1 Body
Element Size	2.E-003 m
Advanced	
Growth Rate	Default (1.2)

Table 4.5 – Mesh settings to fix cell transitions from inflation layer to wake refinement

Scope	
Scoping Method	Geometry Selection
Geometry	1 Edge
Definition	
Suppressed	No
Type	Number of Divisions
Number of Divisions	2
Advanced	
Behavior	Soft
Growth Rate	Default (1.2)
Capture Curvature	No
Capture Proximity	No
Bias Type	No Bias
Blend to Neighbors	Yes

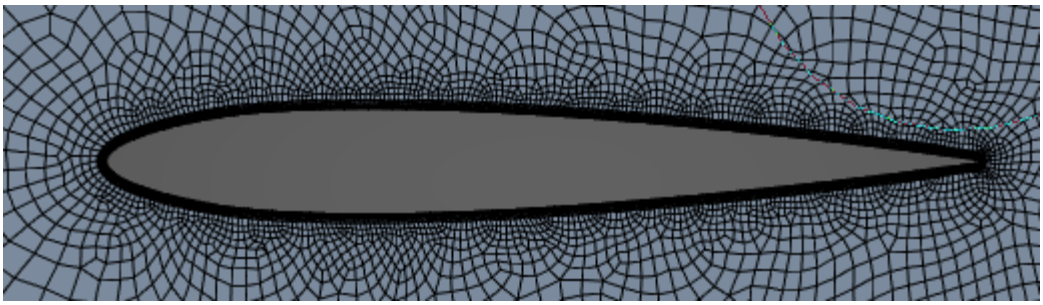


Figure 4.8 – Resulting mesh refinement around the airfoil

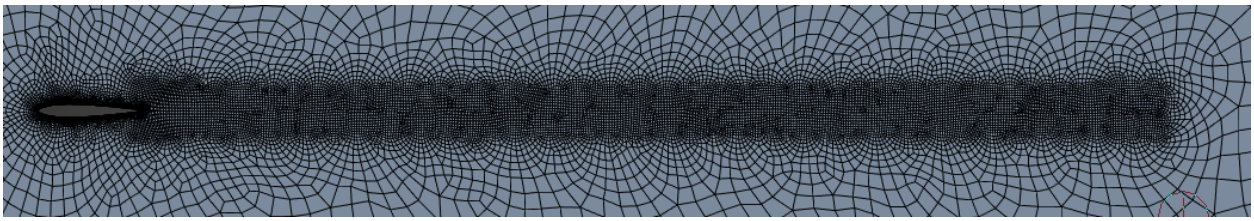


Figure 4.9 – Resulting mesh refinement at the wake region

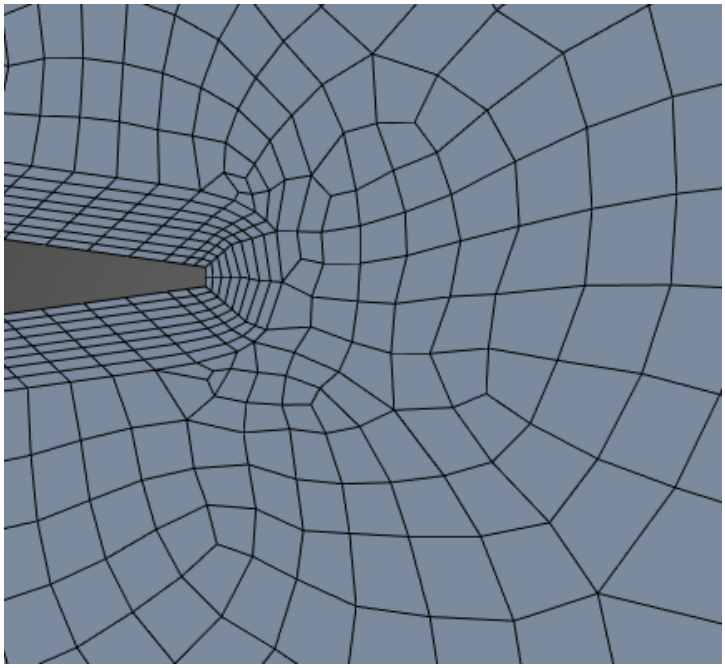


Figure 4.10 – Mesh refinement for smooth transitions between layers and cells

Once the mesh settings were established the mesh could be generated. It's important to note that the mesh had a total of 26,312 nodes. This is important because the simulations to generate the necessary training data will have to be generated using similar techniques as explained above. Thus, by having under 30k nodes the simulation can be quickly executed in less than 5 minutes. This is crucial as data set needs to be quite large and larger simulation time greatly increases the amount of time needed to generate the ground truth data. Additionally, it is at this step that the skewness of the mesh can be checked. The results show that on average the skewness of the mesh is about 0.15. This is great as anything below 0.25 is considered to be excellent. Additionally, when it comes to orthogonality, the average was about 0.96. This is also great, because the closer

the value is to one, the easier it is for the solver to find a solution and converge faster to said solution. Now that the mesh meets all satisfactory requirements, fluent can be used in order to set up the physics and the simulation can be run.

4.2.3 Physics Set Up

When using Ansys Fluent there are 3 main areas to focus on. The first area of importance is the setup. This is where the model for the simulation is chosen, where the fluid materials are chosen, where the chemistry is adjusted for the simulation, where boundary conditions and reference values are set. For the given project the model that was chosen was the SST k-Omega. This model is perfect for running steady state, 2 – Dimensional simulations. The fluid material was selected to be air with a density (ρ) of 1.225 kg/m^3 and a viscosity (μ) of 1.78E-05 kg/ms. For this case the solid material is not important however aluminum was chosen. Moving on to the boundary conditions, the inlet was set to be velocity inlet of 15m/s. Something important to note is that the inlet velocity can be set as components of the x and y direction. This is an incredibly powerful tool as it allows simulation times to be cut dramatically between cases. For the training data set, multiple αs will be used. Thus, by simply changing the inlet velocity components to the appropriate values the angles can be simulated without having to go back and change the geometry and remeshing. The outlet was set to be a pressure outlet; the airfoil shape itself was set to be a wall. Lastly, the top and bottom were set to symmetry, this was done to avoid using pressure outlet for these areas which could introduce an error. Symmetry treats these boundaries as going on forever. The last thing to look for in setup is the reference values; these are the values that the simulation will use to compare to. Important values for this simulation are the density, the length, the velocity, the viscosity, and the ratio of specific heats (γ). It is important that the density and the viscosity match up with that of the fluid. Additionally, it is important that the velocity matches the 15 m/s that are being used as the inlet velocity. The length of 1 m is also important as this is the value used to calculate Coefficient of Lift (C_l), and Coefficient of Drag (C_d). Even though heat transfer is not being analyzed in this simple scenario, matching γ to the value of 1.4 further strengthens the validation of air being fluid that's being analyzed.

The second area of importance is the solution. This is where limits for the residuals can be set. The residuals are the values that the simulation is trying to solve for, once it reaches a convergence value or limit the simulation will stop. It is in this section where the simulation initialization and button to run the calculation exist. Before running any simulation, it has to be initialized, for this project the Hybrid Initialization Method was used. The simulation was set to run for about 500 iterations, although the residuals converge quickly before the 500 iterations are reached.

Finally, the last area of importance is the results. This is where the velocity and pressure contours can be generated from the resulting simulation data. Additionally, this is where the C_l and C_d plots can be generated as well. Additionally, this is the area where animations of the simulation can also be generated. However, for this project, only ground truth data matters so that it can be imported into the DeepCFD AI model.

4.2.4 Results and Discussion

Once the setup for the model is complete, the simulation is initialized and run. The images below show the velocity and static pressure contours at αs 0°, 2°, 5°, 8°, and 10°.

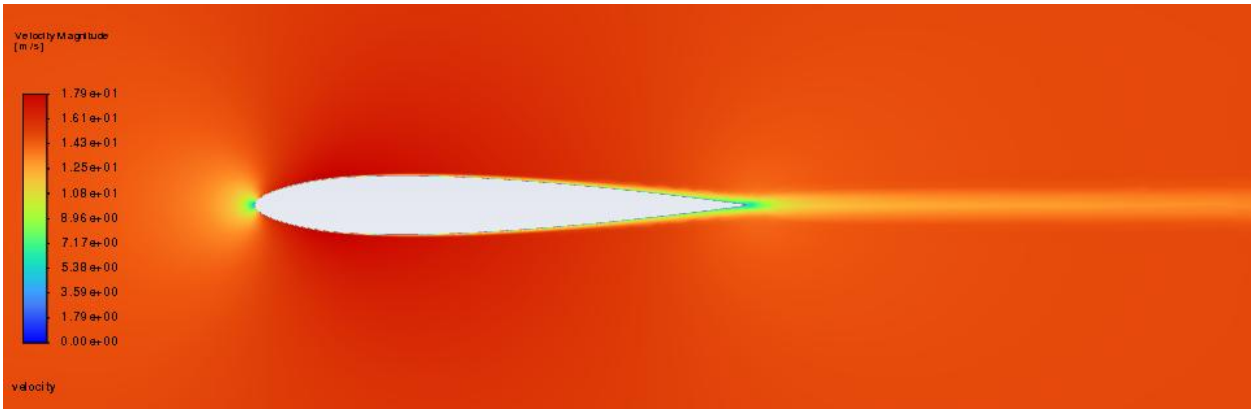


Figure 4.11 – Velocity contour at 0° α

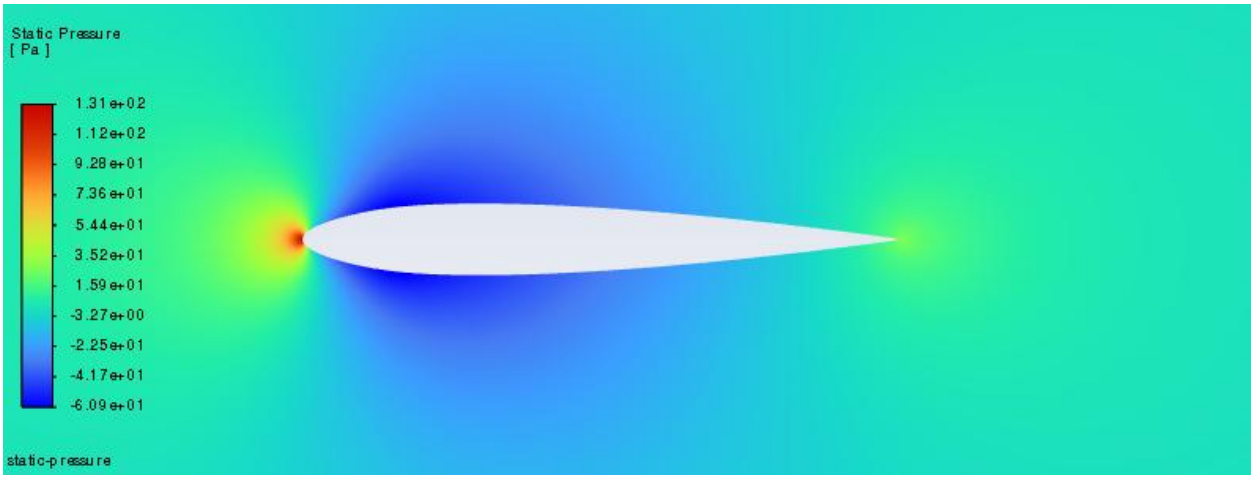


Figure 4.12 – Static pressure contour at 0° α

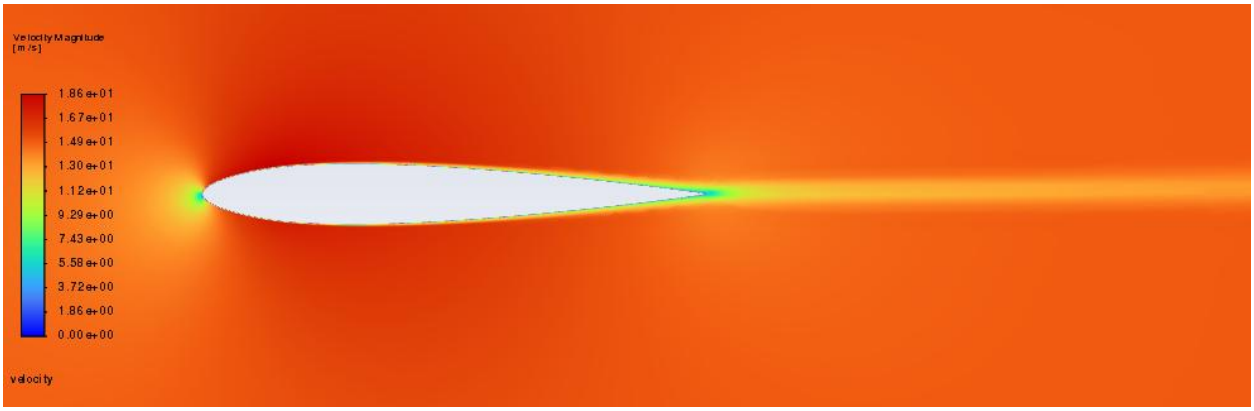


Figure 4.13 – Velocity contour at 2° α

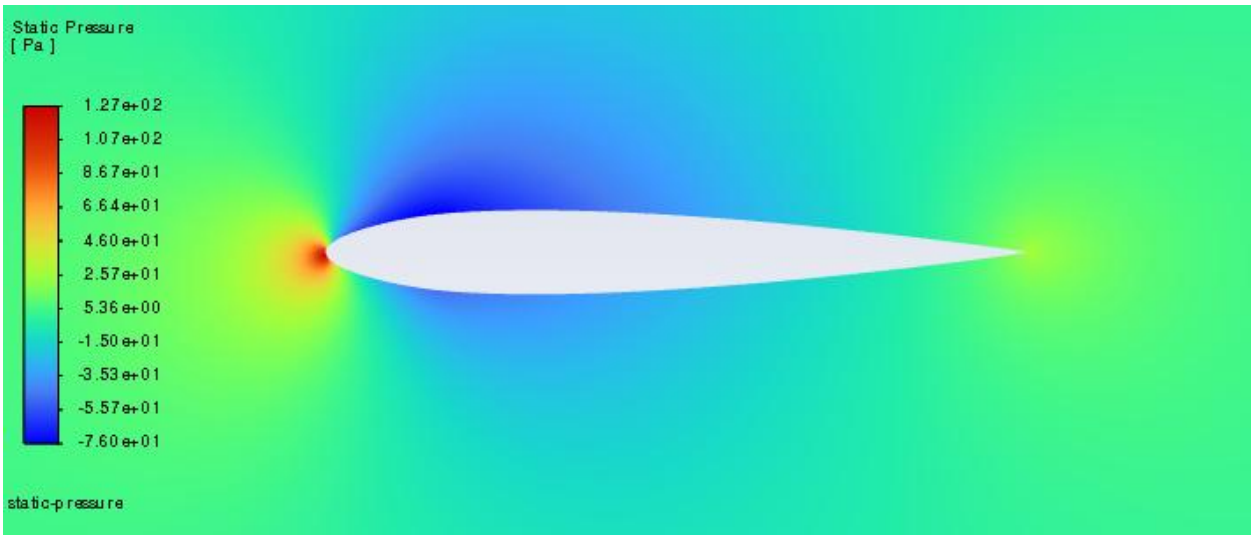


Figure 4.14 – Static pressure contour at 2° α

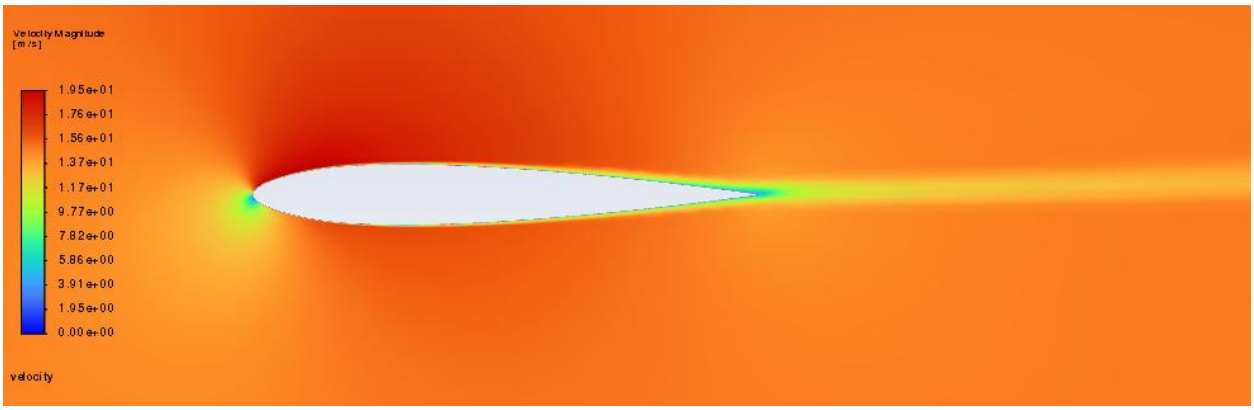


Figure 4.15 – Velocity contour at 5° α

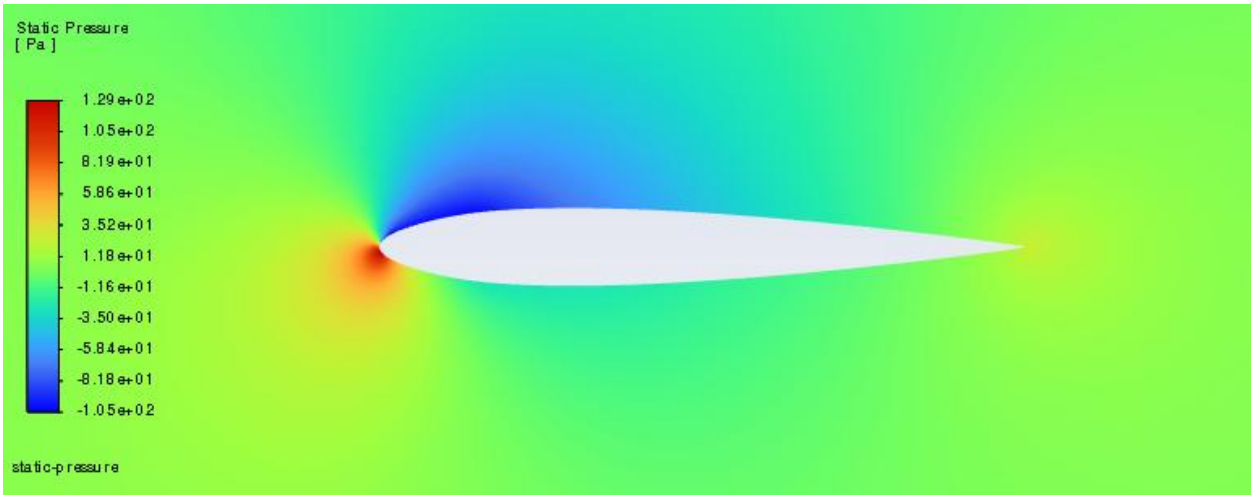


Figure 4.16 – Static pressure contour at 5° α

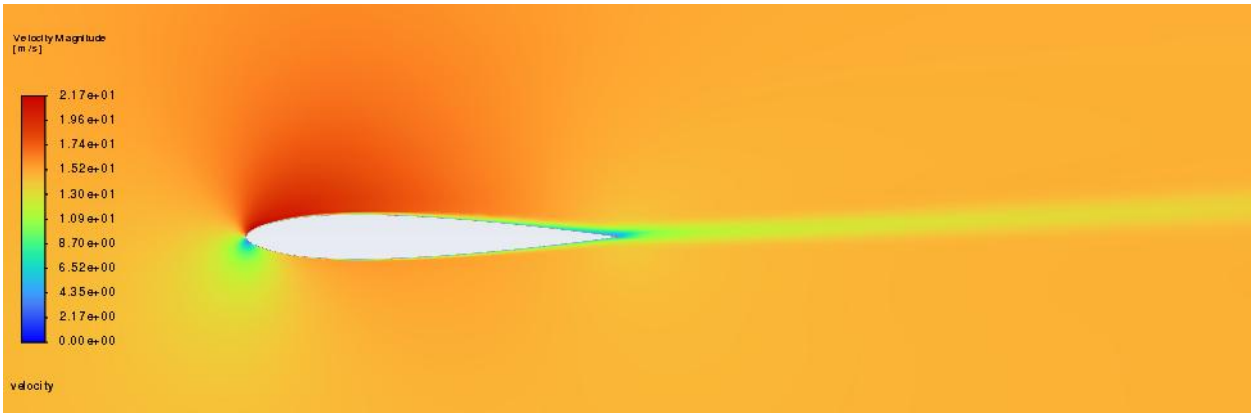


Figure 4.17 – Velocity contour at 8° α

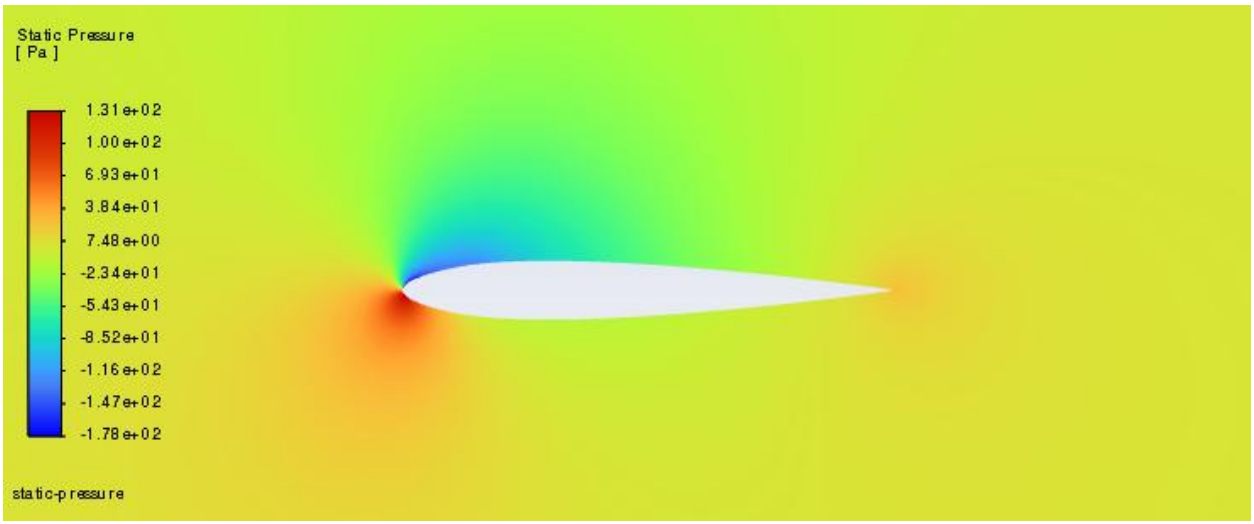


Figure 4.18 – Static pressure contour at 8° α

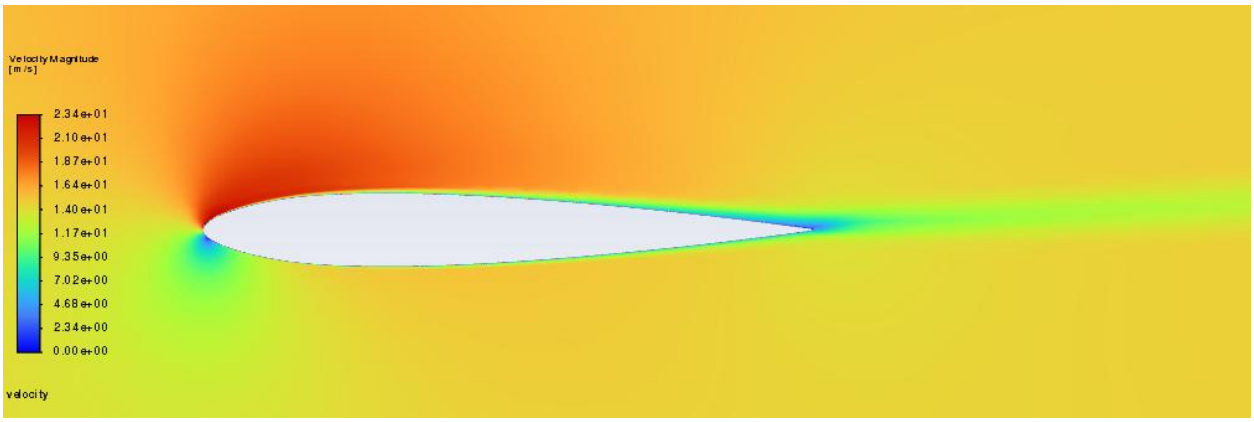


Figure 4.19 – Velocity contour at 10° α

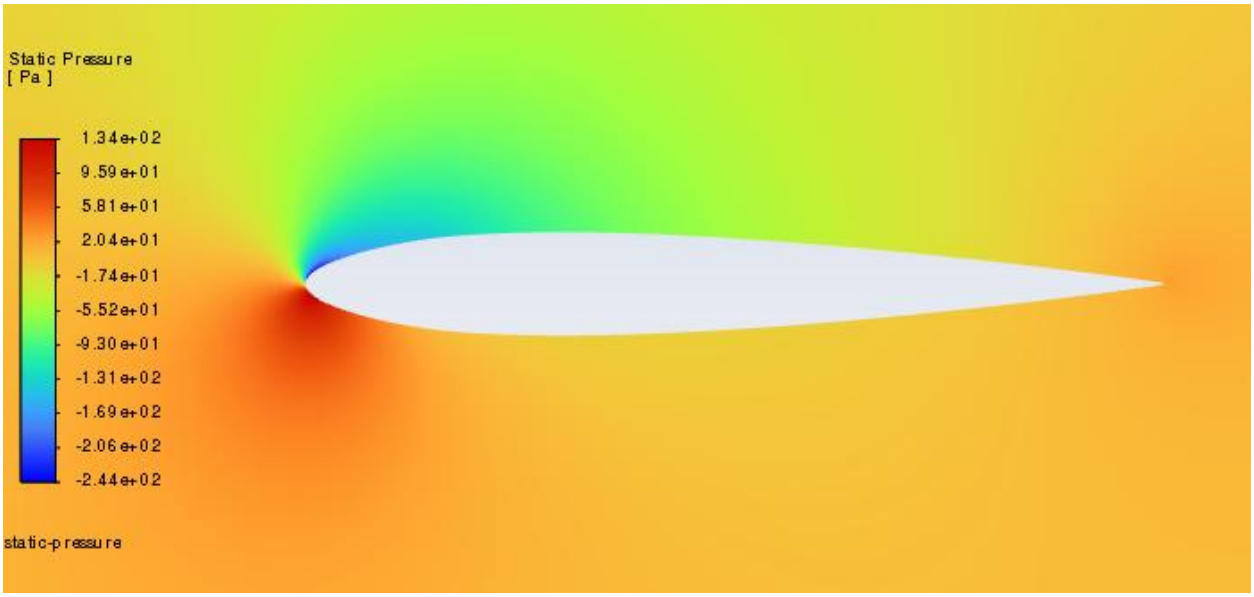


Figure 4.20 – Static pressure contour at 10° α

By looking at the change in the velocity contour and the static pressure contour it can be seen that the simulations were successful. At 0° on figure 4.11, it can be seen that the velocity at the top and bottom of the airfoil is pretty much the same. This is due to the use of the NACA0012 airfoil which by definition is a symmetrical airfoil. At 0° the air is hitting it directly parallel to its chord. Additionally, the trailing wake follows horizontally behind the trailing edge of the airfoil. As α is increased the velocity contours begin to change a bit. Figures 4.13 and 4.15 begin to show a change in the overall velocity in the domain. The background velocity becomes lighter in color while the velocity at the top of the airfoil near the leading edge becomes darker in color. This change in color signifies the change in velocity. While the inlet velocity remains the same at 15 m/s, at increasing αs the air at the top of the airfoil becomes accelerated leading to a higher velocity at this region. Additionally, looking at the trailing edge a slight shift upwards begins to occur. This slight shift is due to the pressure differentials between the top and bottom of the airfoil. When α increases, this causes the air at the bottom to have more resistance than the air at the top, thus; resulting in a higher pressure at the bottom. This is very important as this change in pressure differences is what allows lift to occur. When the top and bottom airflows meet again at the trailing edge of the airfoil the one with higher pressure has more energy, causing the wake to shift a little. Higher αs, we can see that these nuance differences become much more apparent. At 8° and 10° α the velocity of the flow at the top of the airflow becomes much faster and more apparent when compared to the background velocity. Additionally, at these angles, it can be seen that the bottom of the airfoil has much slower velocity when compared to the airfoil, but much more pressure when looking at the static pressure. This is to be expected, since at higher αs there is more air hitting the underside of the airfoil, meaning less velocity at the bottom, but higher pressure. Additionally, looking at the wake in figures 4.17 and 4.19 it can be seen that these wake regions bend upwards much more aggressively than at lower αs, further increasing the validity of the simulation.

The simulation has been successfully run for one single airfoil at 5 different αs. In order to create a robust training dataset, this process will have to be done around a hundred times over to generate the necessary data for 500 different cases. Meaning 100 different airfoils at 5 different

angles of attack. To achieve this, the 99 remaining geometries will be loaded inside of the already existing geometry and replace the NACA0012 airfoil shape. Using the same physics set-up the case will be run for 5 different angles of attack: 0°, 2°, 5°, 8°, and 10°. From each scenario a comma-separated value (CSV) file is saved containing the node number, the pressure, the x-velocity, the y-velocity, the x-coordinate, and the y-coordinate. This is the data that will be passed into DeepCFD to train the model. Additionally, the case/data files were saved to have the simulation data ready without running the simulation again.

5. Creating the Training Dataset Using Ansys Fluent

5.1 Creation of Dataset

To create the dataset needed to train the model, the airfoil data needed to be collected. Since the NACA 4-digit series was being used for this project, to simplify this data collection the NACA 4-digit equation could be used. The corresponding script can be found in Appendix A.1. The NACA 4-digit series is quite simple to understand. For starters, NACA is an acronym for National Advisory Committee for Aeronautics. This committee created this standardized series of airfoil shapes. These shapes are designated as NACAMPXX, where M refers to the maximum camber divided by 100. Camber refers to the line that represents the midpoint between the top and bottom of an airfoil shape. The maximum number in this case refers to the maximum deviation from the true midpoint. For example, if the maximum camber M is 0, then there is no deviation from the true midpoint; thus, the airfoil is said to be symmetrical as the top and bottom of the airfoil are equally distant from each other at all points. However, if M is 2, then this means that the maximum camber is 0.02 or 2% of the chord length. Chord length refers to the distance from the front (leading edge) of the airfoil to the back (trailing edge) of the airfoil. Additionally, if M is 4 or 6, then the maximum camber is 4 and 6 % of the chord length respectively. The following number P refers to the positioning of the maximum camber M divided by 10. For example, in symmetrical airfoils, since there is no maximum camber, then there is also no position for the max camber. Thus, P is designated to be 0 resulting in symmetrical airfoils having the denomination NACA00XX. However, if P is 4, then this means that the position of the airfoil is located at 40% of the chord length. Finally, the last two numbers represented by XX refer to the thickness. For example, a NACA6412 would have a maximum camber of 6% of the chord length, located at 40% of the chord length, with a thickness of 12% [21]. There exists a set of equations used in order to generate these airfoil shapes. They are known as the NACA 4-digit equations and can be seen in equations 5.1-5.10. A python script was used in order to generate the 100 airfoil shapes as .txt files.

Equations 5.1 and 5.2 are the equations used to calculate the camber of the airfoil. Equation 5.1 calculates the camber for the front of the airfoil, while equation 5.2 calculates the camber for the back of the airfoil.

$$Y_c = \frac{M}{p^2}(2Px - x^2) \tag{5.1}$$

$$Y_c = \frac{M}{(1-P)^2}(1 - 2P + 2Px - x^2) \tag{5.2}$$

Equations 5.3 and 5.4 are responsible for calculating the gradient. Similar to the camber equation 5.3 calculates the front gradient of the airfoil, while equation 5.4 calculates the back gradient of the airfoil.

$$\frac{dY_c}{dx} = \frac{2M}{P^2}(p - x) \tag{5.3}$$

$$\frac{dY_c}{dx} = \frac{2M}{(1-p)^2}(P - x) \tag{5.4}$$

Equation 5.5 calculates the thickness distribution of the airfoil. It is important to note that the variables $a_0 - a_4$ are actually constants of values: $a_0 = 0.2969$, $a_1 =$ -0.126, $a_2 =$ -0.3516, $a_3 =$ 0.2843, $a_4 =$ -0.1015, or -0.1036 (if a closed trailing edge is required). While the coefficients were generated through a baseline 20% thickness profile the $\frac{T}{0.2}$ parameter allows for linear scaling to be able to generate the airfoils from 6% thickness to 30% thickness.

$$Y_t = \frac{T}{0.2}(a_0 x^{0.5} + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4) \tag{5.5}$$

Finally, to get the upper and lower surface coordinates needed to plot the airfoil shape a thickness is applied perpendicularly to the camber. To do this the angle θ of the camber line must be found. To do this equation 5.6 is used. From here equations 5.7 – 5.10 can be used to generate the appropriate surface coordinates for the upper surface and lower surface of the airfoil. Equations 5.7 and 5.8 generate the upper portion, while equations 5.9 and 5.10 generate the bottom portion.

$$\theta = \arctan\left(\frac{dY_c}{dx}\right) \tag{5.6}$$

$$x_U = x - y_t \sin(\theta) \tag{5.7}$$

$$y_U = y_c + \cos(\theta) \tag{5.8}$$

$$x_L = x + y_t \sin(\theta) \tag{5.9}$$

$$x_L = y_c - y_t \cos(\theta) \tag{5.10}$$

The 100 selected airfoils have been chosen from a pool of 25 symmetrical airfoil shapes and 75 non-symmetrical airfoil shapes. Since 25 are symmetrical they fall under the nomenclature of NACA00xx, where xx represents the percent thickness from 6% thickness to 30% thickness. Out of the 75 non-symmetrical airfoils, 25 fall under the low camber designation with a nomenclature of NACA24xx, 25 fall under the medium camber designation with a nomenclature of NACA44xx, and 25 fall under the high camber designation with a nomenclature of NACA64XX.

Once the airfoil shapes have been generated the next step is to move into Ansys Fluent where the model that was used for generating the NACA0012 simulation data in chapter 4 would be utilized. To do this the geometry is changed from the NACA0012 to the desired airfoil shape. In the Ansys workbench the setup tab is updated, automatically updating the mesh and the physics setup to have the new shape. Once this is complete, the simulation can be run inside of the solution tab. Each airfoil requires 5 separate simulations to be run in order to get the data for the 5 αs. To do this without changing the geometry a simple mathematical trick can be applied. Fluent allows the user to input velocity as a form of x and y components. Thus, by applying trigonometry the x and y components can be easily calculated for the 5 αs: 0°, 2°, 5°, 8°, and 10°. Equations 5.11 and 5.12 below explain the trigonometric calculations. Table 5.1 shows the resulting velocity components used at each α.

$$x = \mathrm{Cos}(\alpha) * V_{inlet} \tag{5.11}$$

$$y = Sin(\alpha) * V_{Inlet} \tag{5.12}$$

Table 5.1 – The x and y components for corresponding α

Angle of Attack (α)	X – Component (U_x)	Y – Component (U_y)
0°	15 m/s	0 m/s
2°	14.99 m/s	0.52 m/s
5°	14.94 m/s	1.31 m/s
8°	14.85 m/s	2.09 m/s
10°	14.77 m/s	2.60 m/s

The setup and physics were exactly the same as they were for the simulation run in chapter 4. After every simulation the case and data files were stored for every simulation just in case the data was lost or corrupted, it could easily be recovered later on. Additionally, this is when the CSV file containing the data for the pressure, the x-velocity, the y-velocity, the x-coordinate, and y-coordinate for each node was collected and saved. This process was repeated until all 100 airfoils were simulated with the 5 corresponding αs. The resulting dataset was composed of 100 different airfoils ranging from symmetrical, low camber, medium camber, high camber, thin, thick, and differing angles of attack.

6. Importing Data into PyCharm

6.1 Importing the Pressure and Velocity Data

One of the most important steps in this project is transferring the data from Ansys Fluent into PyCharm. If at any point something gets corrupted or not transferred properly, the training is compromised and with it so is the model's ability to learn correctly. Inside of the CSV files collected from the 500 Ansys Fluent simulations exist the velocity and pressure data. Additionally, the node number for each corresponding cell and x and y coordinate are stored within this file. Each file has around 26,000 nodes of information for every single airfoil at each α. Thus, this process is extremely crucial. To import this into PyCharm the data had to be converted into a format that AI model could read and learn from. To do this a 256 x 256 grid was created in which the airfoil was at the center. This does a few things. Firstly, it creates a standardized window that the AI model looks at for every single airfoil. Without it the data has no structure, and it would be meaningless to the AI model. Additionally, since the DeepCFD framework uses U-Net architecture, this means that the model learns using tensors (multi-layered images) to learn from. The grid allows the velocity and the pressure to be placed into a standardized format; creating a spatial map. This standardization is what allows the AI model to be able to make connections at certain points in the grid. For example, stagnation points at the leading edge, accelerated flow with lower pressure at the top of the airfoil, and slower flow with higher pressure at the bottom of the airfoil.

To achieve this a python script was written inside of PyCharm that would import the data from the CSV files and create a 256 x 256 grid that would match the pressure and velocity data to the corresponding pixel based on the coordinates from the same CSV file. The code can be found in Appendix A.2. The logic behind the code is as follows. Firstly, the paths are defined. These paths are where the script pulls the CSV Fluent files and where it stores the resulting grid data. Next the 256 x 256 grid is created. The grid ensures that the x-coordinates cover from -0.5 to 1.5. This is done to ensure that the grid captures enough of the space in front of the airfoil to capture the stagnation point. Similarly, the y-coordinates covered by the grid stretch from -0.5 and 0.5. This is done so that the physics underneath and on top of the airfoil can also be captured. Next the script is told the name of the four categories that it needs to look for inside of the file path that it was initially given. These four categories are" Symmetrical, Low Camber, Medium Camber, and High Camber. Each one of these folders contains the CSV Fluent data for 25 airfoils at 5 different αs. For loops are used to navigate through the four categories looking for the CSV files. Once all 500 files are found, the data is then loaded. When the CSV Fluent files were initially saved it was noticed that the names of the columns had spaces. To make sure there were no errors due to spaces the spaces were deleted from all columns to ensure smooth transition of data. Next, the x and y-coordinates were extracted. These coordinates were used to appropriately fill in the pixels with the correct pressure, x-velocity, and y-velocity values. This created three separate 256 x 256 grids each corresponding to the pressure, x-velocity, and y-velocity. In order to save them, they were stacked and saved as NumPy files. This last step was taken for efficiency as it saves the file in a binary format. This format allows PyTorch to load it much faster during training than using a CSV file.

This code was used during the initial process of training the first three models, however, along the way an error was discovered which will be discussed later on in the section 7.4.

6.2 Importing the Airfoil Geometry

The next step after creating the 256 x 256 grid containing the pressure, x-velocity, and y-velocity data was to deal with the geometry of the airfoils. To do this a Signed Distance Function (SDF) generator was implemented. Due to the nature of the project looking at the way the air behaves around the airfoil, the distance from the airfoil geometry is crucial in understanding the physics. At the surface of the airfoil, the no-slip condition will apply. Close to the surface is the complicated physics of the boundary layer. Further from this is the pressure gradients and velocity changes. To achieve this the following logic was used in the code. This code can be found in Appendix A.3.

Firstly, the paths were defined for finding the .txt files that contained the coordinate information for each of the 100 airfoils. The path for where the images would be saved was also defined. Next the same 256 x 256 grid that was used for saving the pressure and velocity values was used. This is to match the geometry with the pressure and velocity data. The next step was to use a for loop that would go through all of the airfoil files and load columns three and four containing the x and y coordinates. Next the SDF image is created, ensuring each pixel is assigned the closest distance to the airfoil shape. Since α is being simulated through x and y velocity components instead of changing the geometry positioning, this means for each airfoil five identical copies are made for each α. Finally, the images are saved with the correct corresponding nomenclature. At this point both the 256 x 256 grid containing the pressure and velocity data, and the 256 x 256 SDF image were created. The following steps were to train the model and look at the results.

7. Initial Training and Results

7.1 Version One

One thing that is important to note is that DeepCFD is not just an existing model for training AI to analyze CFD around geometry. It is a structured framework that allows researchers to use that framework and change it according to their needs. DeepCFD uses a Signed Distance Function (SDF) that assigns a value to each pixel representing the distance from the nearest point to the geometry's surface. This allows the model to see where exactly the boundaries of the geometry are within the 256 x 256 grid. DeepCFD uses U-Net architecture, essentially the brain of the model. U-Net architecture allows the model to learn through the use of downsampling and upsampling through skip connections to learn about the overall flow around the geometry; while also differentiating between high pressure and low pressures zones. It uses SDF and U-Net architecture to produce approximations for the fluid flow field, in the form of velocity; in the x and y directions, and static pressure. By implementing this core framework, DeepCFD can be applied to train a model to predict fluid flow around airfoils at different angles of attack. While the exact model was only used to attempt to recreate the author's findings, the framework that was used for training a model for airfoils can be considered DeepCFD as the core framework is heavily derived from the author's initial model. It can be thought of as using DeepCFD specifically for airfoils.

Following the importation of the pressure and velocity values into PyCharm and the creation of the SDF images for each of the 100 airfoils, the next step was to train a model. Since this was the first model, it was only trained for 50 Epochs. The code for this model can be found under Appendix A.4.

The logic the code uses is as follows: First the paths are defined for the location of the SDF images, the Gridded data, the Training Log, and the saved model. Next, the parameters that will be used have to be defined. The Batch Size is the number of airfoils that are fed into the CPU at one time for the model to be trained. Given that this was a test run, the Batch Size was set to be 4. Next the number of Epochs was chosen to be 50. Epochs is the number of times that the model will train over the entire dataset. The Learning Rate was set to 0.001. The Learning Rate is the size of the steps taken for correcting mistakes that the model made. The Dropout Rate was set to be 0.2. Now, Dropout Rate is something that the original authors of DeepCFD did not incorporate into their model. What this does is that it takes a percentage of neurons and randomly shuts them down during training. This is done to ensure that the model truly learns the flow and does not memorize the dataset. In this case the Dropout Rate of 0.2 means 20 percent of the neurons will be turned off during training at random. The next step is to load the necessary data for training. In this part of the code the airfoil shapes are sorted into alphabetical order, the number of airfoils in the library is counted, finally the SDF and Gridded files are opened and converted from NumPy file to PyTorch Tensor. This is done because Neural Networks that use U-Net Architecture use Tensors in order to read the data. Additionally, in this step, 3 output channels are created for the x-velocity (U_x), y-velocity (U_y), and pressure (p).

The next part of the code focuses on the U-Net Architecture. Firstly, the initialization is defined using the Input Channels, the Output Channels, and the Dropout. Next the Convolutional Block is defined. The Convolutional Block is the part of the AI that goes over the image identifying

features such as the leading edge and trailing edge of the airfoil. This code also includes a function that tells the AI to ignore negative data or data that is considered useless by the weights. Next, the steps taken by the AI through the U-Net Architecture are defined. These steps include the encoder which are the layers that extract features from the image, the pooling layers which shrink the image to learn more about the image. It does this until it reaches the bottleneck part. This is the considered the deepest part of the U-Net Architecture and it's the part with the lowest resolution. From here the model proceeds to do Up-Sampling and Decoding the image. Up-Sampling and Decoder get the image back to its original size by using skip connections to form high resolution images. The final step is to take all of the layers and combine them into the 3 output channels U_x, U_y, and p. The definition forward simply defines the logical flow of the U-Net Architecture.

The following portion of the code is the Training Set-Up. Here 80% of the airfoils are taken to train the model and the remaining 20% are used to compare with the trained model and validate the training. In this part is where the batch of airfoils are fed into the training, but they are shuffled to prevent the model from memorizing the dataset. The model and optimizer are set in this part of the code as well. Before the training for loop is initiated, the Training Log is initiated. This training log contains: Epoch Number, Train Loss, Train MSE Total, Train MSE U_x, Train MSE U_y, Train MSE p, Validation Loss, Validation MSE Total, Validation MSE U_x, Validation MSE U_y, and Validation MSE p. The final part of the code is a for loop that trains the model through 50 epochs. In this part of the code the optimizer is reset every epoch, the AI takes a guess, the loss is calculated, the backpropagation takes place to see where it went wrong, the optimizer changes that weights. The training metrics and validation metrics are collected and saved into a CSV file. Finally, once the loop is done the model is saved. Once this model had finished training, the comparison script could be run to visualize the results of the training.

This comparison script can be found in Appendix A.5. The logic of this script is easy to understand. Similarly to the previous scripts, the paths for the SDF files, and the Gridded files are defined. The name of the saved model and the output folder are also defined in this section. The next part creates a folder in which the final comparison images will be saved. These images will compare the ground truth CFD data vs the predicted images from the AI model and also provide an absolute error graph between the two. The next step is to pick 5 test cases that will be used to compare ground truth with the predicted AI results. The next step is to load the U-Net Architecture that was used during the training process. However, this will be empty initially and will be populated later when the model data is loaded into the U-Net Architecture. The next step is to load the model. The next step is to run the for loop that will generate the comparison images for each of the test cases. The first step is to load the ground truth CFD data. Then the predicted data is loaded from the model. Lastly the CFD, AI predicted, and absolute error images are created and saved into the folder that was created at the beginning of the code. Figures 7.1 -7.5 below show the resulting comparison images.

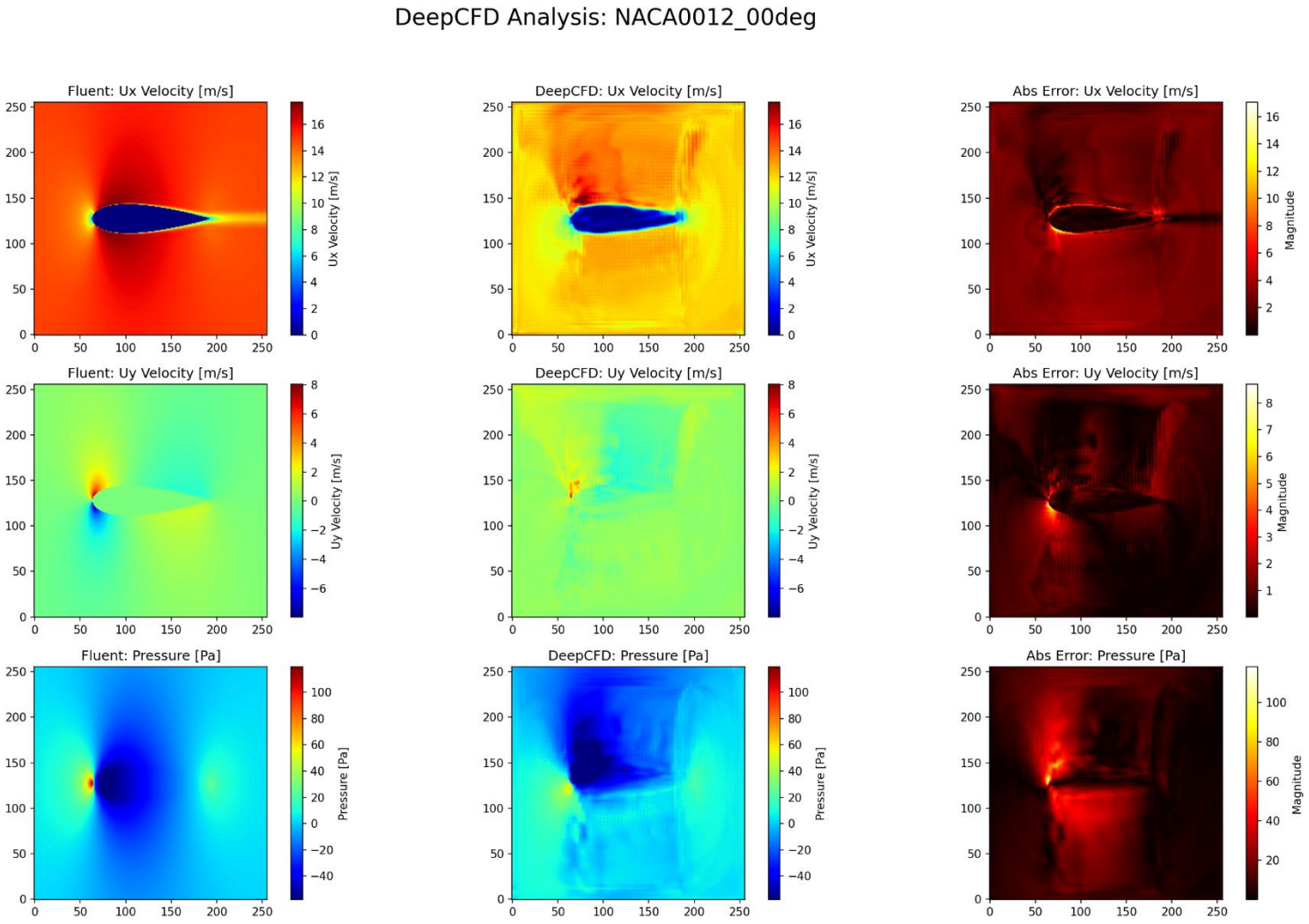


Figure 7.1 – Comparative fluid flow field analysis of v1: NACA0012 at $\alpha = 0°$

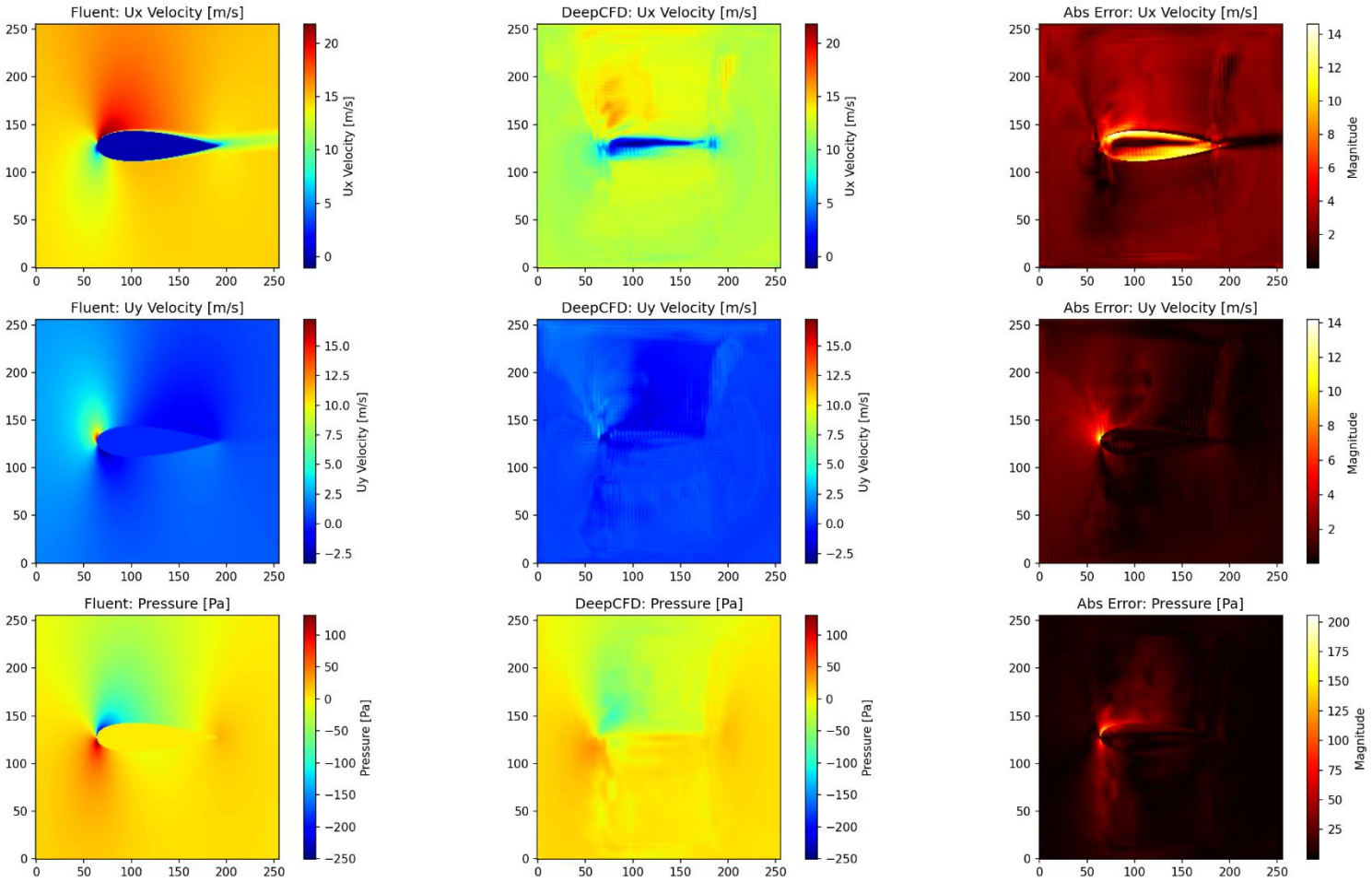


Figure 7.2 – Comparative fluid flow field analysis of v1: NACA0012 at $\alpha = 10°$

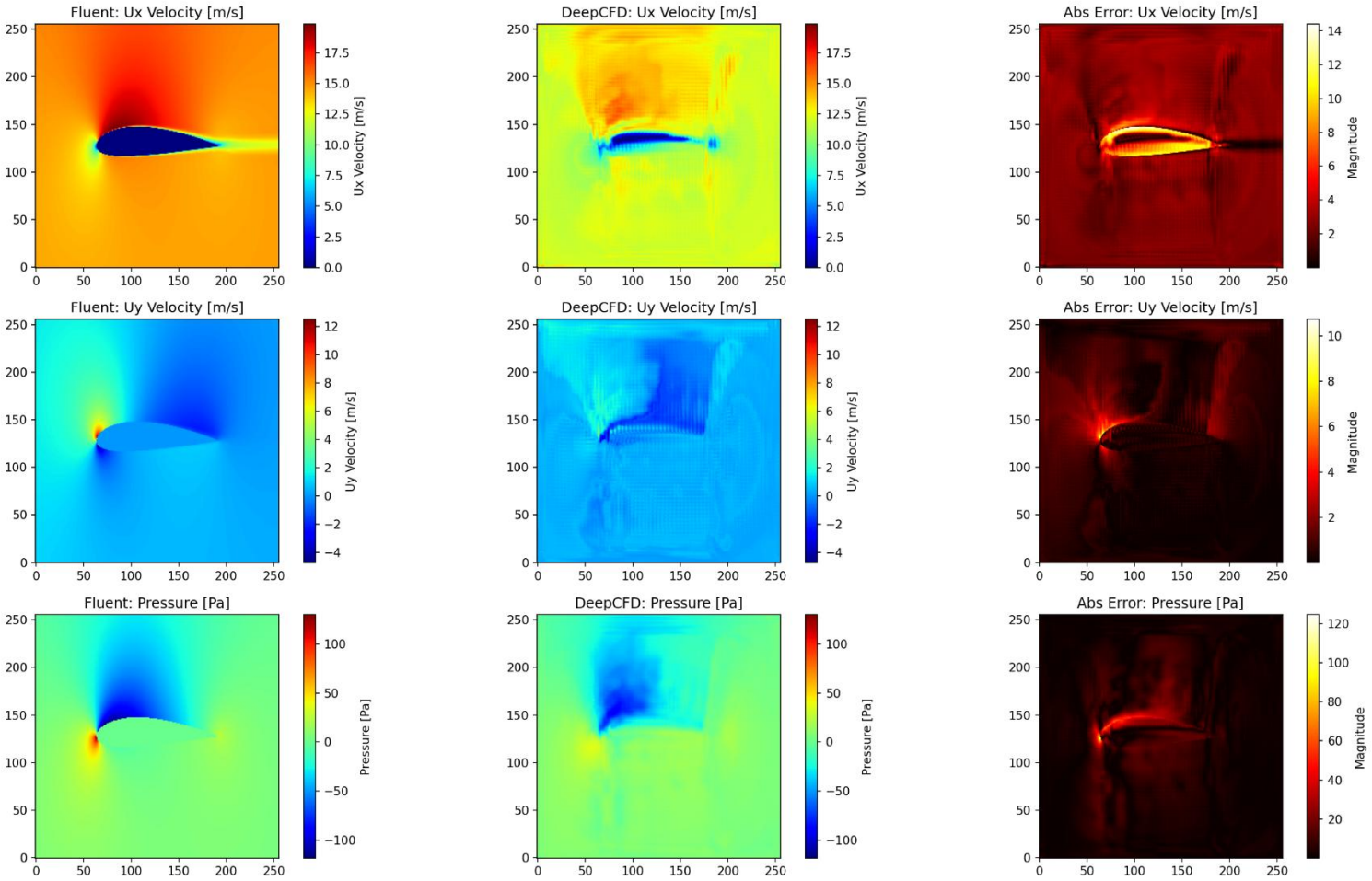


Figure 7.3 – Comparative fluid flow field analysis of v1: NACA2412 at $\alpha = 5°$

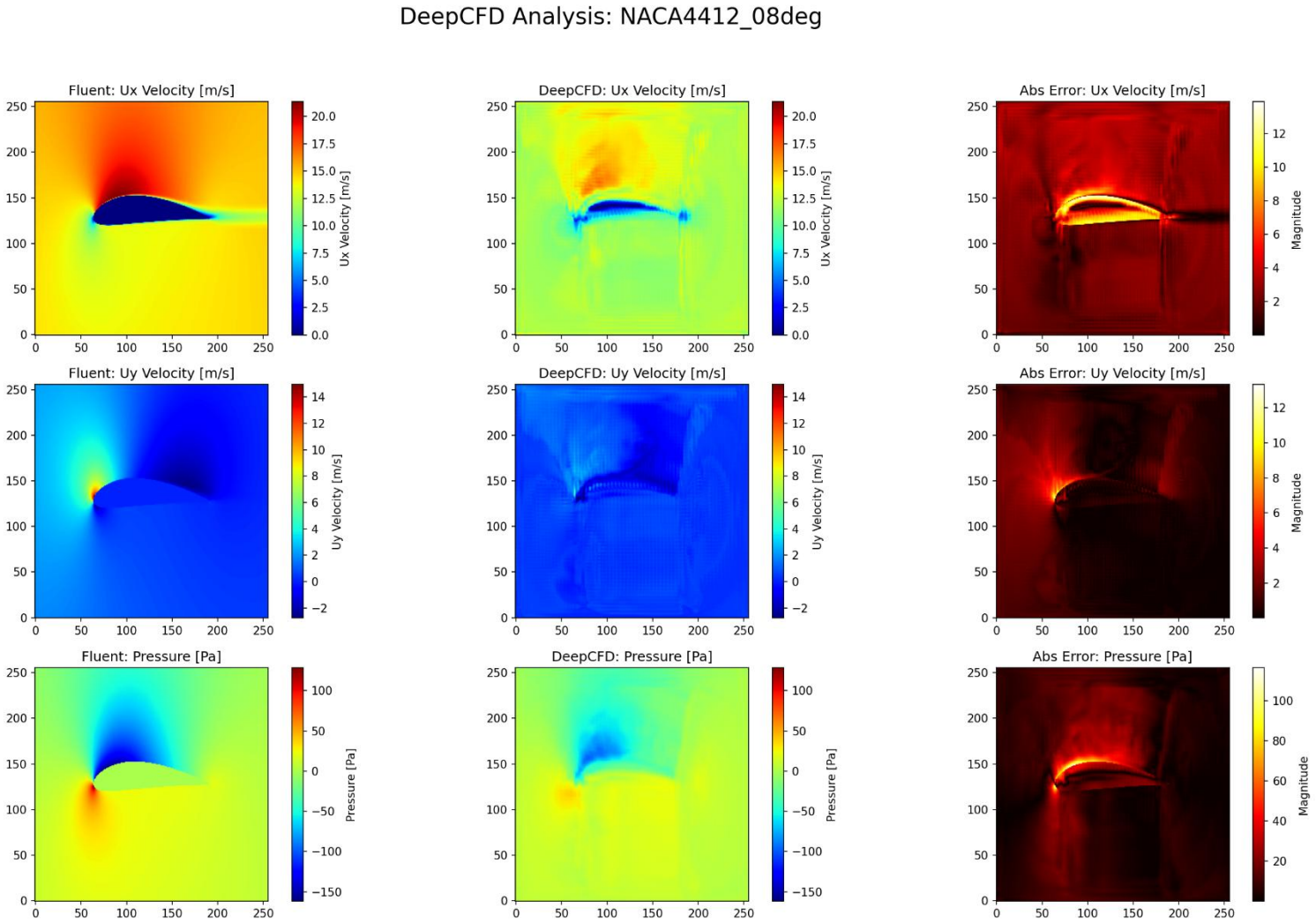


Figure 7.4 – Comparative fluid flow field analysis of v1: NACA4412 at $\alpha = 8°$

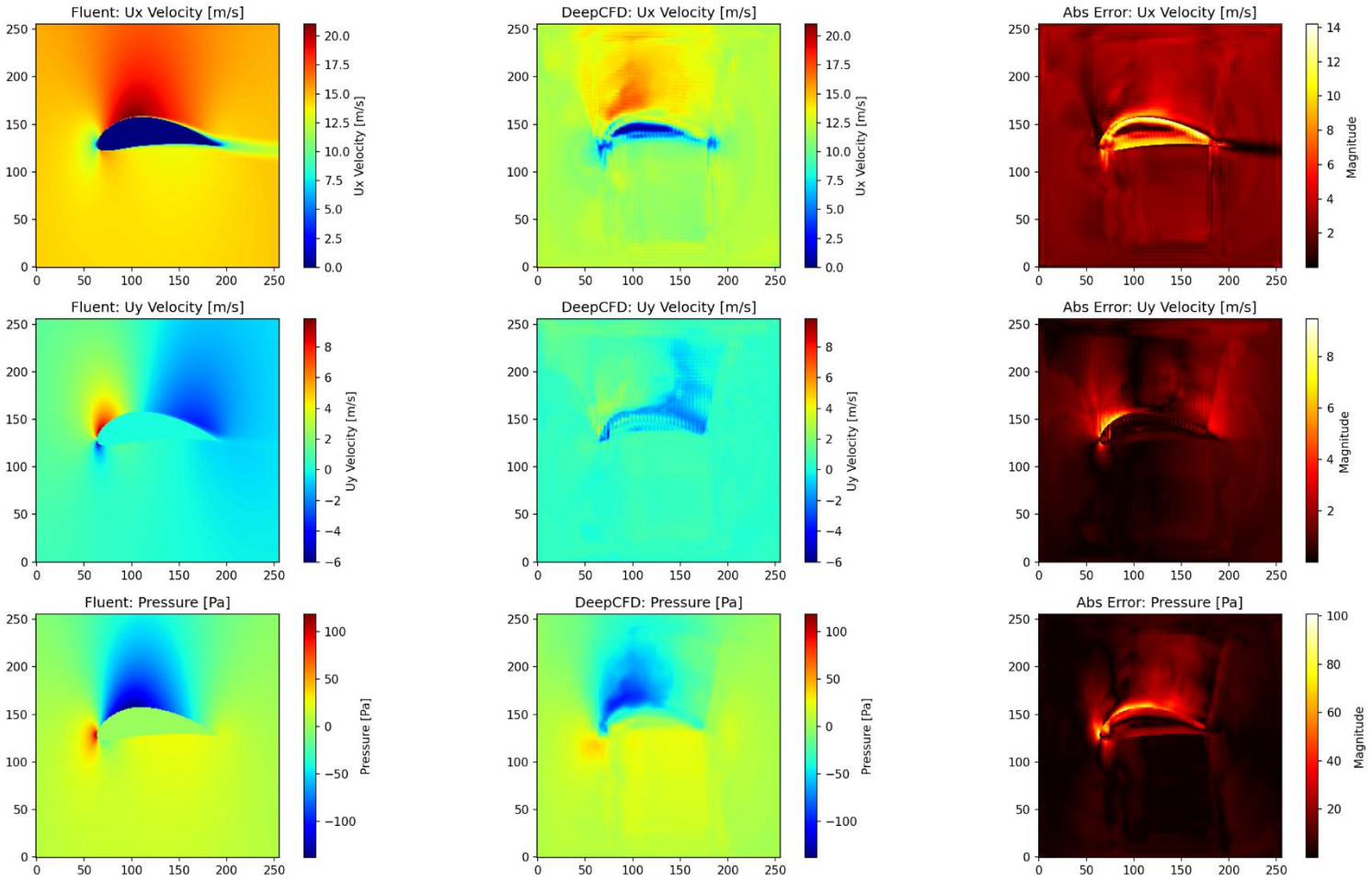


Figure 7.5 – Comparative fluid flow field analysis of v1: NACA6412 at $\alpha = 2°$

Figures 7.1 – 7.5 above show the results of five tests cases that were used to compare the Ansys Fluent CFD results to the predicted results from the trained DeepCFD AI model. Each of the five images shows nine 256 x 256 images. The left column is the ground truth data that was generated through the Ansys Fluent CFD software. The Middle column shows the predicted results from the trained DeepCFD AI model. The right column shows the absolute error between the CFD data, and the AI model predicted data.

Several observations can be made from these results. First, it can be seen that the model is not yet accurately predicting the flow. It is catching on to the shape of the airfoil, but it still requires further training. It is clear that the model is on track to learn the x-velocity quite well. It is, however, having issues with the y-velocity and the static pressure. The y-velocity seems to be lacking the shape of the airfoil as is the pressure prediction.

A closer look at the x-velocity component U_x, reveals that the model is picking up on the higher velocity at the top of the airfoil and the slower velocity at the bottom of the airfoil. However, this is also a double-edged sword. The dataset that was used to train this model was composed of 25% symmetrical airfoils, and 75% cambered airfoils. Additionally, out of those 25% symmetrical airfoils, only 20% were at 0° α; the rest were at an angle. This means that the majority of the data that was used to train the model had faster air flow velocity at the top of the airfoil and slower airflow velocity at the bottom. Thus, when the symmetrical case NACA0012 is analyzed at 0° α. The predicted image shows faster airflow at the top of the airfoil, instead of having equal airflow velocities at the top and bottom of the airfoil. Figure 7.1 shows this discrepancy. Additionally, it can be seen that the AI is also having trouble seeing the stagnation point at the leading edge of the airfoil. The AI is also struggling to see the difference between the wake behind the airfoil and the rest of the surrounding airflow. Furthermore, the absolute error image confirms that the AI still requires more training time.

Regarding the y-velocity component U_y, the error appears to be less, however this is mostly due to U_y having lower values than its counterpart U_x. Since the values are closer to 0, the AI is mostly just learning that this image should have lower velocity in general. However, looking at the predicted image, it's clear that it is struggling to understand the exact distribution of the airflow velocity.

When it comes to the pressure, the results differ. Technically in this initial version it was performing well when compared to the CFD data. For a model that had only been trained for 50 Epochs, the color gradients were extremely good, and the absolute error map further confirmed this. However, the true mistake with this comparison lied deep within the code. This mistake was not discovered until after the third version of the AI model. Section 7.4 will explain this mistake in detail and the solution that was used to fix the mistake.

7.2 Version Two

Version one was created as a baseline for the project. Firstly, its main purpose was to make sure that the data was being read and processed correctly by the DeepCFD model. Secondly, it was created to see if the model was being saved properly and could be called back to find the comparison images between the CFD data and the AI models predicted data. It was by no means meant to be a perfect model. In the engineering field, first models rarely if ever produce the correct results. Through the engineering process of reiterating is how correct and reliable results are reached.

Version two of the code looked to increase the number of epochs the model is trained for, lower the learning rate to reduce overshooting and divergence, and use normalization to better train the model. Normalization essentially takes the data and reduces it to a number between 1 and 0. This was an important implementation in this project as it is much easier for an AI model to look at numbers between 0 and 1 than it is to look at numbers between 5 and 101,000. With a bigger range, the AI will assume that lower numbers simply don't matter. Appendix A.6 shows the python script that was used. The logic behind the code is similar to version 1. It starts off by defining the file paths for where the SDF files and the Gridded files are saved. The paths for where the training log and the model will be saved are also defined in this section of the code. Next is the

parameters. The Batch Size was kept at 4. The number of Epochs; however, increased to 100 from 50. This was done to give the model more time to learn and hopefully get better results. Additionally, the Learning Rate was cut in half from 0.001 to 0.0005. This was done to ensure that the model doesn't make big jumps that lead to overshooting or potentially to build up errors that lead to divergence from the truth. Similar to version one, the dataset of airfoils is loaded, sorted, and opened. However, in this version normalization is added to the code. The SDF files are normalized to keep the distance values between 0 and 1. Additionally, a for loop is used to normalize U_x, U_y, and p independently. The importance of normalization comes down to the way that the AI model learns. Keeping values between 0 and 1 makes it much simpler for the AI model to learn. Whereas if the values were kept as they were, the range would be too great for the AI model to learn effectively. For example, the pressure values range from 100s of thousands to 0 to negative values. The velocity ranges from much smaller values. Therefore, when weights are applied it is possible that the AI model considered low velocity values insignificant because it is too focused on the high-pressure values. Therefore, by normalizing the SDF, and the 3 output channels U_x, U_y, and p independently it helps prevent the AI model from ignoring potentially important values. From here, the U-Net architecture is kept the same as version one. The division between the dataset and the airfoils used for comparison is kept the same. The way the data is fed for training and validation is also kept the same. The way the model is loaded is also the same as version one. The training for loop is similar to that of version one, except in this case only the training loss and the validation loss are logged. Additionally, since the model was going to be training for twice as long, heating became a big issue. To reduce the temperatures of the CPU a simple trick was implemented. After 10 every 10 Epochs a small 10 second window is given to allow the CPU a small break. This kept the CPU temperatures around 90°F. Once the model had been trained for 100 Epochs, the model was saved to the previously defined path.

Similar to version one, once the AI model had finished training a comparison script could be used to produce the comparison images between the Ansys Fluent CFD data and that AI model predicted data. The script that was used for this comparison can be found below in Appendix A.7. This script was very similar to the script used for version one. The difference between the two versions is that this version had to incorporate the normalization and by extension, the denormalization. The logic of this code is as follows. The path locations are defined for the SDF files, Gridded files, and saved model which will be called upon to gather the necessary data for plotting. Additionally, the path location for the folder containing the images is also established. 5 test cases are picked to use as the comparison cases. The U-Net architecture is redefined as it was for the training script. The model is loaded, and the for loop that will generate the images begins. The for loop loads the SDF and Gridded data which contain the Ansys Fluent ground truth data. It also implements the normalization, then loads the predicted results for U_x, U_y, and p from the model. Since the data was initially normalized, it must be denormalized so that the values are logical. A for loop is used to loop through the three channels: U_x, U_y, and p to denormalize the values independently. The next step is to plot the resulting data. Again, three columns are created similar to the version one comparison script. The left-hand side shows the true CFD data for U_x, U_y, and p, the middle column shows the AI model predicted values for U_x, U_y, and p, and the third column shows the absolute error between the CFD and AI model for U_x, U_y, and p. Figures 7.6-7.10 below show the results.

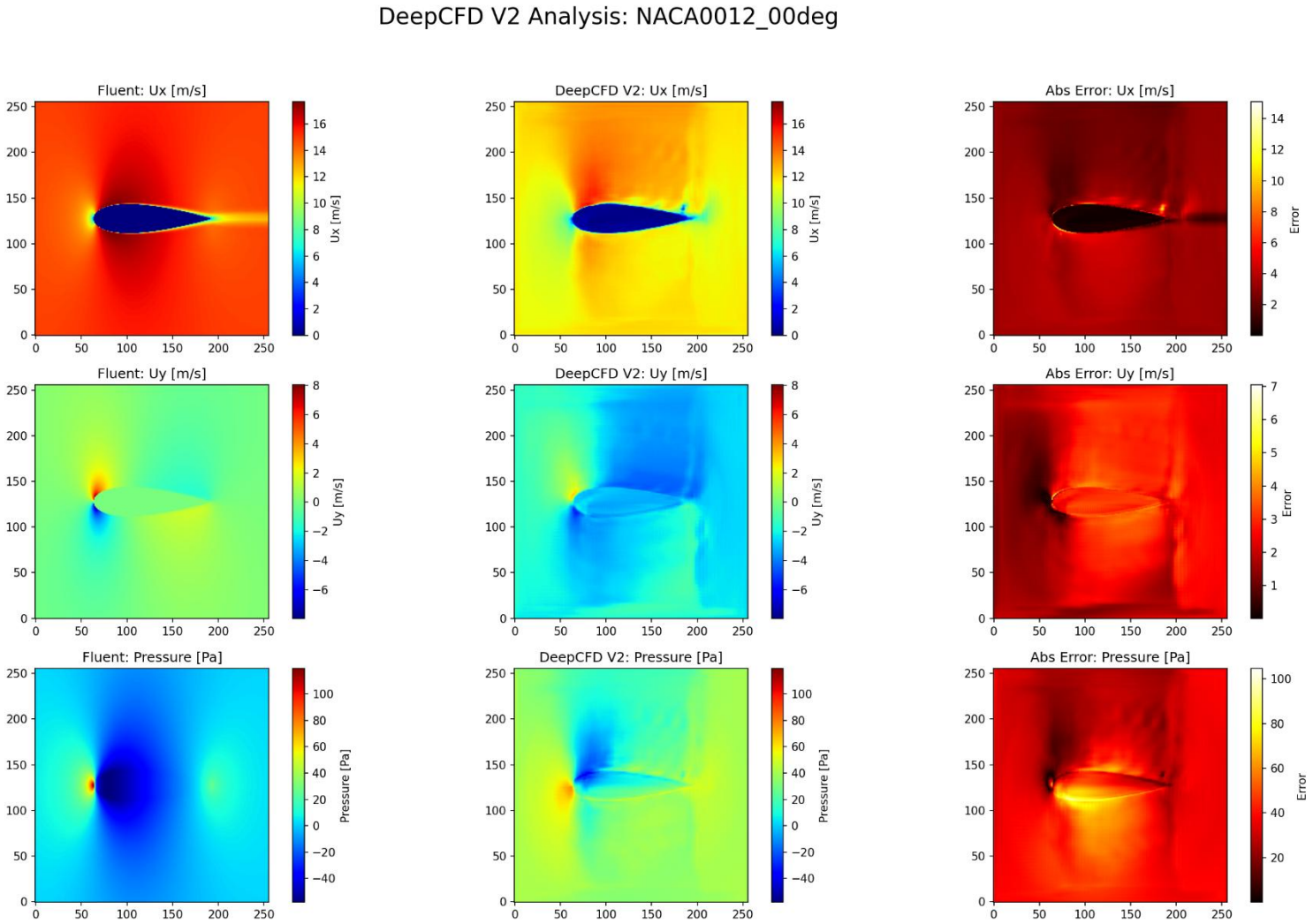


Figure 7.6 – Comparative fluid flow field analysis of v2: NACA0012 at $\alpha = 0°$

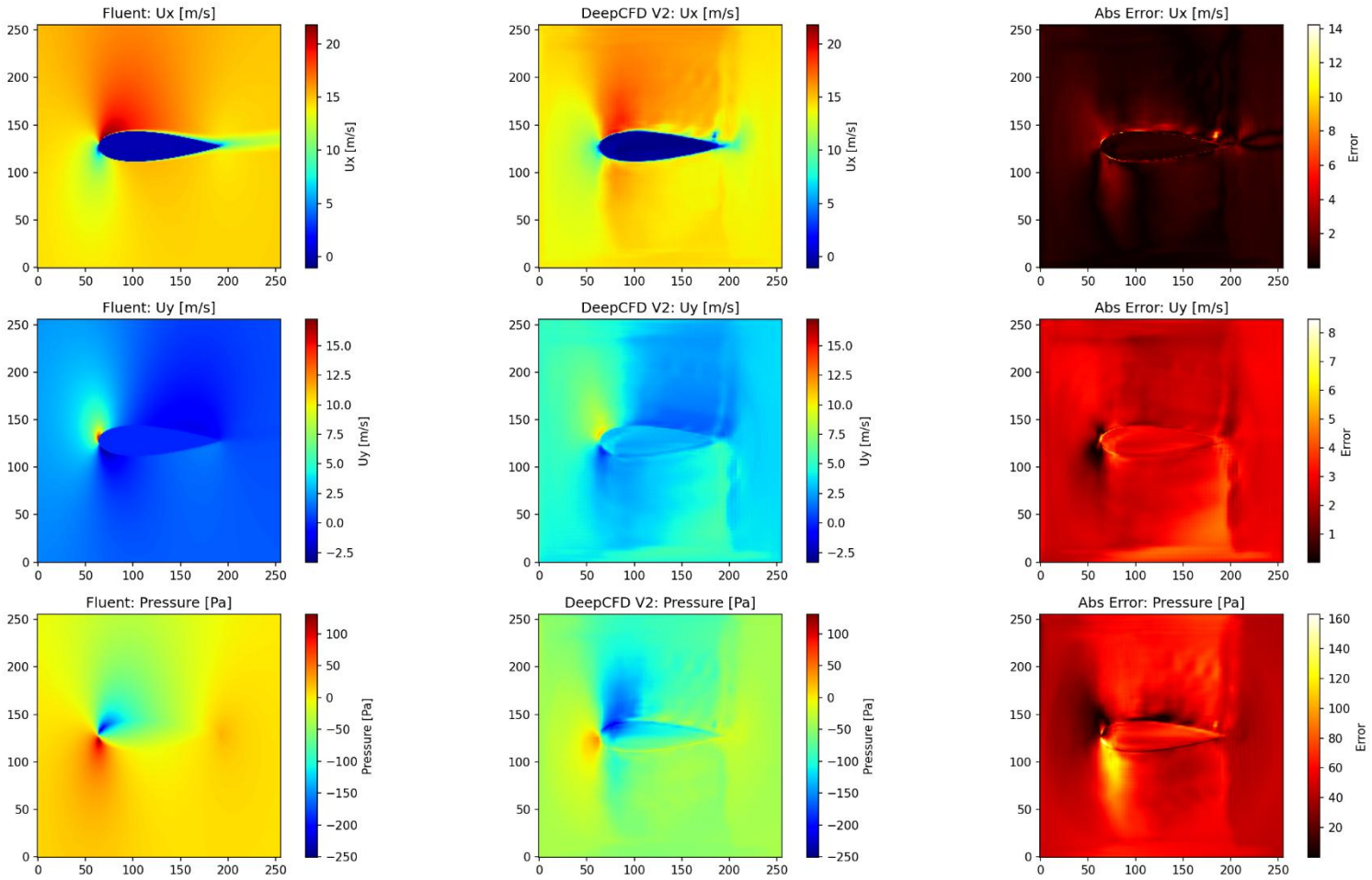


Figure 7.7 – Comparative fluid flow field analysis of v2: NACA0012 at $\alpha = 10°$

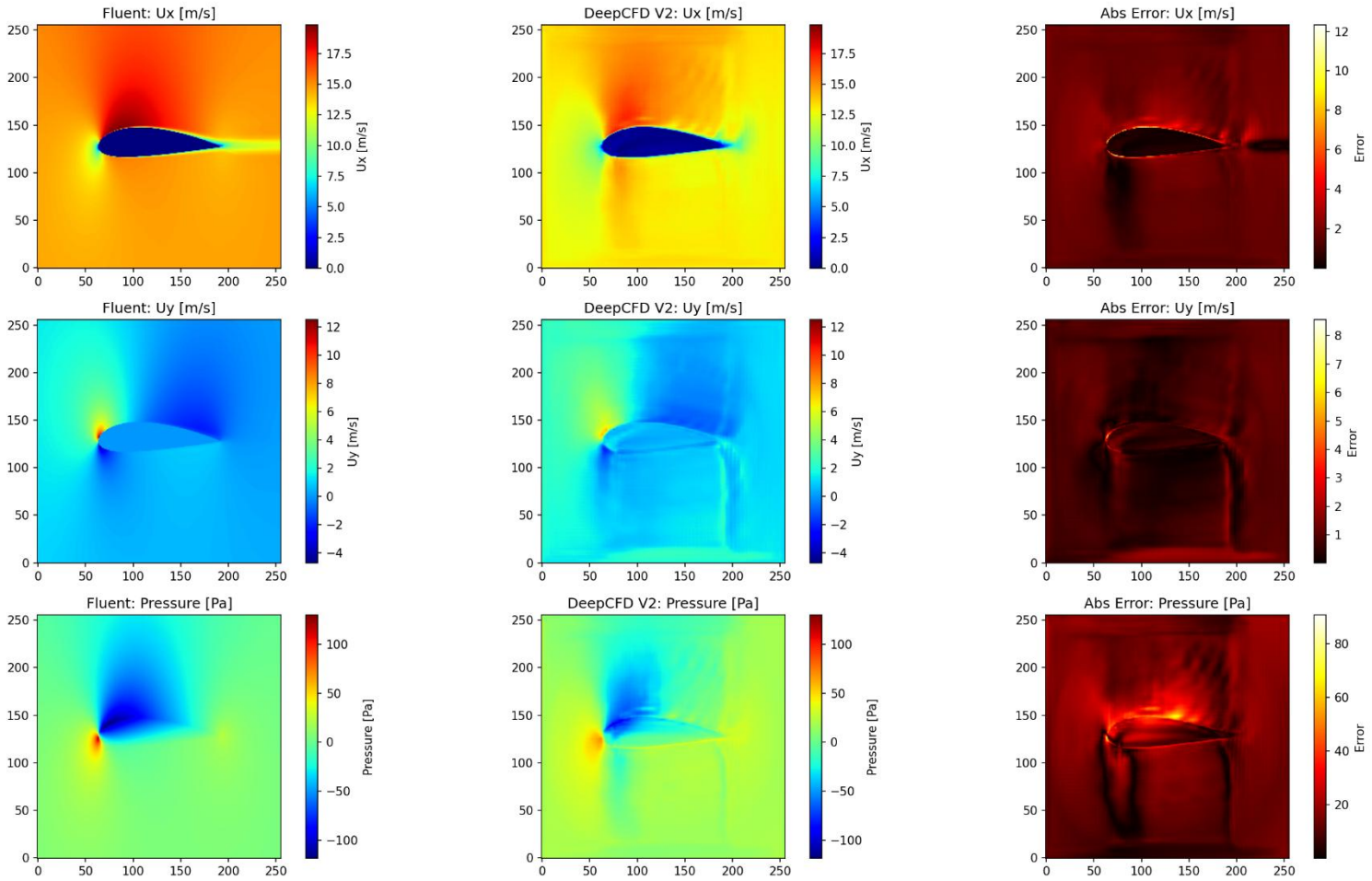


Figure 7.8 – Comparative fluid flow field analysis of v2: NACA2412 at $\alpha = 5°$

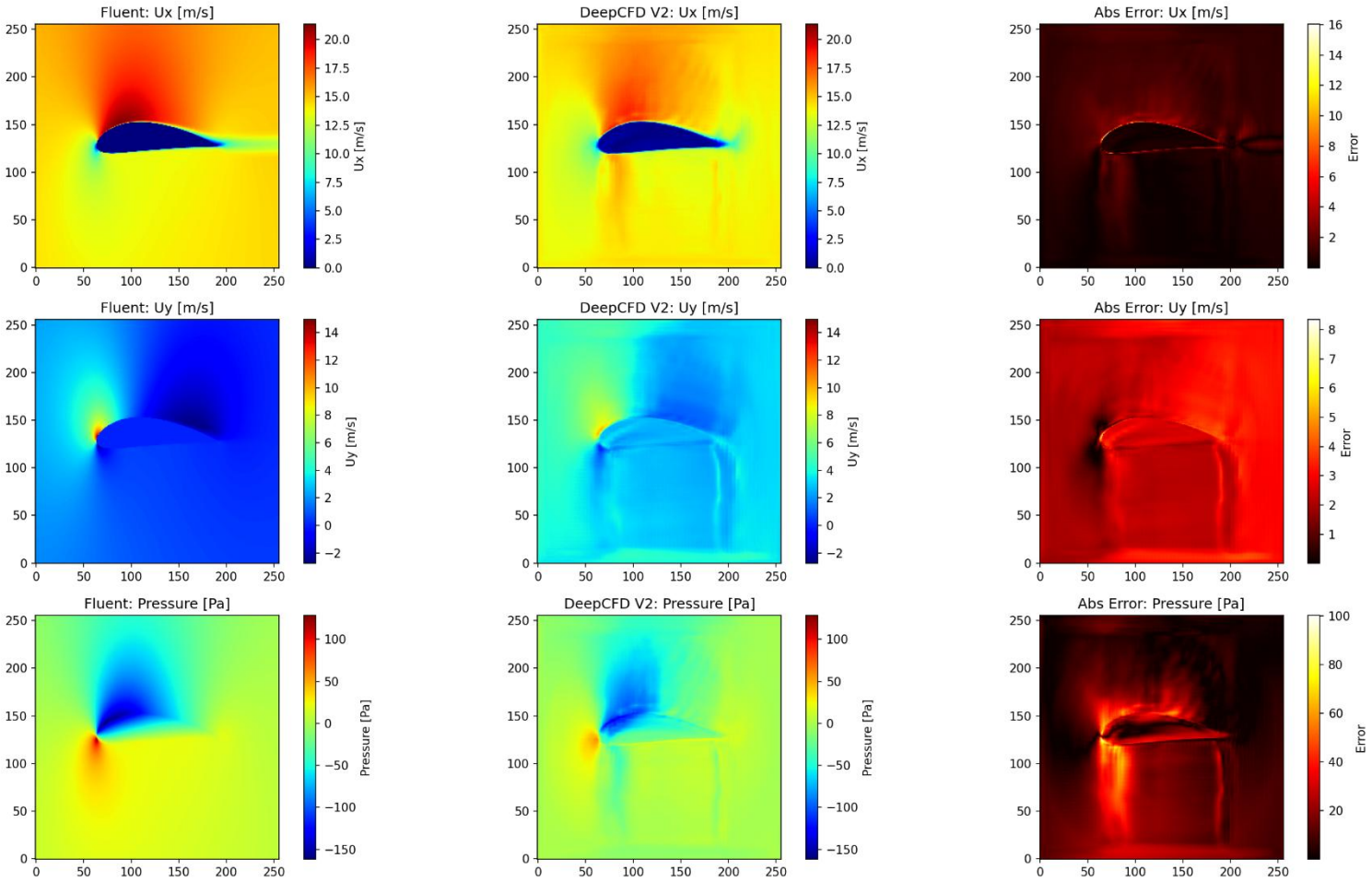


Figure 7.9 – Comparative fluid flow field analysis of v2: NACA4412 at $\alpha = 8°$

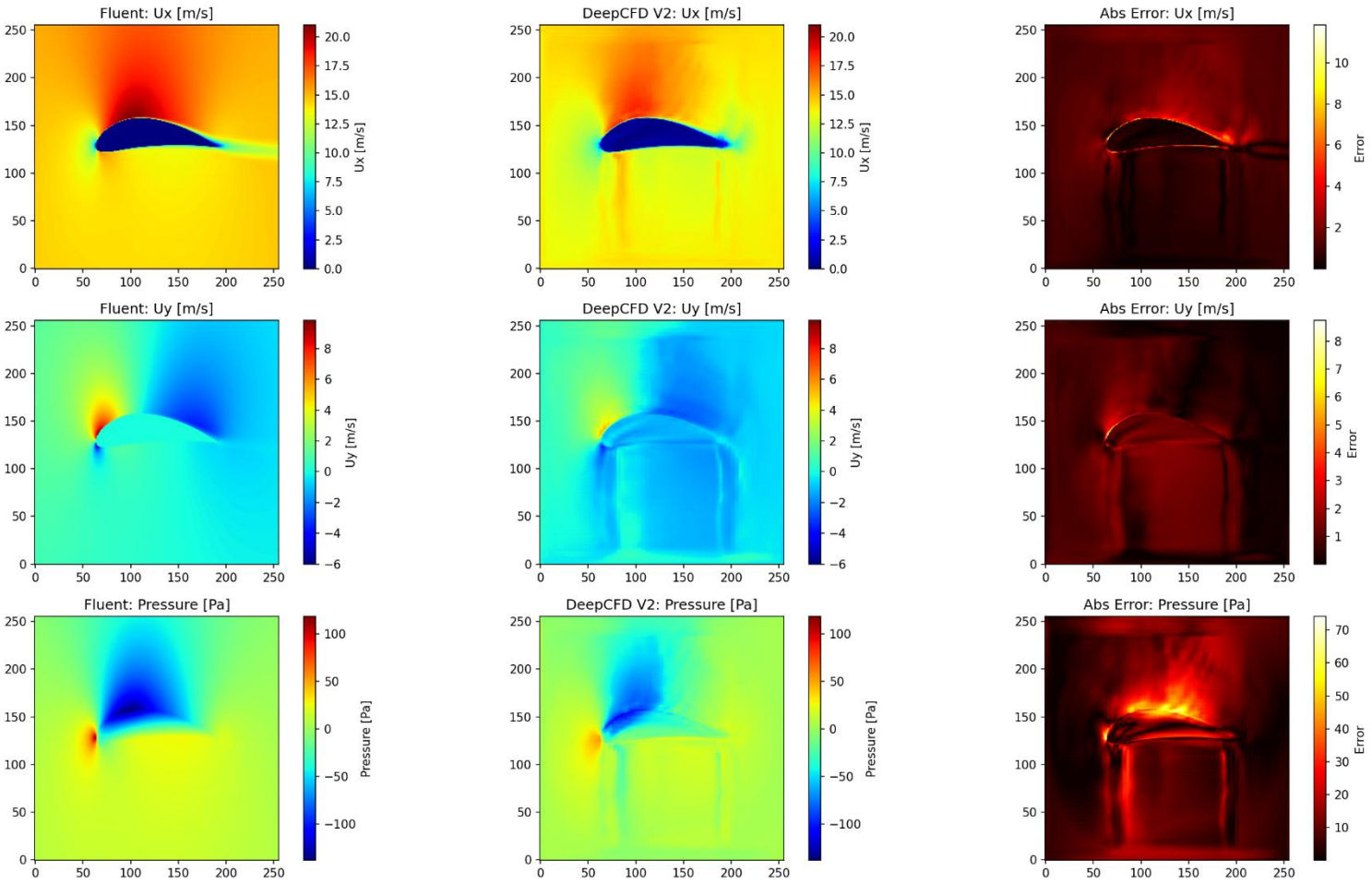


Figure 7.10 – Comparative fluid flow field analysis of v2: NACA6412 at $\alpha = 2°$

Version two utilized the same 5 airfoil cases as version one to compare the effectiveness of the trained models. It can be seen that version two is already more effective than version one. For the most part version one was a trial run to make sure everything was running accordingly. However, it was noticed that during that initial training, the model was learning U_x, better than it was learning U_y and p.

Despite these early results, the model remained significantly far from prefect. The second version shows that the model is beginning to comprehend the overall physics much better. For starters the error related to U_x is lower, and it can also be seen that the prediction results show smoother transitions between the high and low velocities.

In figure 7.8, the prediction is correctly showing faster airflow at the top of the airfoil, and slower airflow at the bottom of the airfoil. Additionally, version one of the AI model was not smart enough to capture the stagnation point at the leading edge or the wake at the back of the trailing

edge. While version two is not perfect, it is starting to differentiate these areas from the rest of the airflow.

U_y, also has noticeable improvement from its predecessor. In version one, the model was struggling to see the airfoil shape. In version two, the airfoil shape is becoming more apparent, and it is also starting to learn the correct velocity values. The absolute error shows that overall, there is still much more room for the model to learn. The performance of the model for p is also improved, the AI model is starting to differentiate between the high and low-pressure areas; yet the transitions still lack smoothness. These improvements can be attributed to the implementation of normalization and also giving the model more time to learn from the dataset. While the model was not perfect, it showed that the changes that were made were positively influencing the training of the model.

7.3 Version Three

Since the second version had shown signs of improvement, version three looked to improve upon that version. The Learning Rate was cut, from 0.0005 to 0.0001. This was done as a way to fine tune the errors. Additionally, the author's method of a hybrid loss function was used. In this hybrid loss function, Mean Square Error (MSE) is used for the U_x and U_y components. Where sharp changes can occur. For p, the Mean Average Error is used as it helps capture the smooth transition typically associated with p. Additionally, the Adam optimizer was swapped with the AdamW optimizer. The difference is that AdamW is a newer version of the Adam optimizer with weight decay. This optimizer decouples learning rate and weight decay which prevents overfitting. Overfitting is when the AI model essentially memorizes the dataset that it was given. While it may perform well for the given dataset, giving it something that slightly deviates from the training dataset would result in incorrect or nonsensical results.

The full script that was used to train this third version can be found in Appendix A.8. In this case the paths were defined similarly to versions one and two. The main difference is that this third version looked to improve on the second version of the code. Thus, the second version of the model was also called in to continue the training after 100 Epochs for another 100 Epochs with all the implemented changes. The training parameters were kept the same. The Batch Size was kept to 4, the Learning Rate was kept at 0.0001, and the Dropout Rate was kept at 0.2. Next the functions that load the dataset are defined as previously. Similar to version two of the code, the normalization for loop is used to normalize the three channels U_x, U_y, and p independently. The U-Net architecture is defined, with the forward function dictating how the down-sampling and up-sampling will work. The airfoils are divided into an 80/20 percent split, where 20 percent of the airfoils are used to train the model, and 20 percent are used for validating. Previously a random split was generated, however, in this newer version a seed was used instead. This was done so that if the code is run again or if someone else runs the code the results will be the same. The previous model is loaded with the AdamW optimizer. The author's weighted loss function is also defined where MSE is used for the velocity components and MAE is used for the pressure component. These losses are then added for the total loss. Before the training can begin the logger is loaded. This logger displays the Epoch number, the training loss and the validation loss. The training is then initiated and runs for 100 Epochs. To protect the CPU from high temperatures the same trick that was used in version two is implemented. After 10 Epochs, a small 10 second break is given to

allow temperatures to remain stable. After this model finished training for an additional 100 Epochs the Train Loss was 0.0782, and the Validation Loss was 0.0838. While these numbers may appear insignificant at first, they tell a much larger story. Firstly, what is important is not how big they are, but rather the gap between them. Since the gap between them is small it can be considered that the model is training well. If the validation loss was considerably higher than the validation loss this would mean that the AI model would be overfitting. This occurs when the model memorizes the given dataset, however it performs bad outside of the dataset. If the training loss was higher than the validation loss this would be a good thing. It's rare for this to occur, but quite possible. Since the Dropout method was implemented into the training this means that during training the AI model has 20 percent less neurons to use to form connections. Yet, when the model is tested against a validation case, all of the neurons are turned on. Therefore, this would mean that the training loss would be attributed to the Dropout method. In this case the validation loss is greater than the training loss, but the difference is not so much that overfitting is occurring. Rather it means that the model is being generalized, where a new airfoil shape can be given to the model, and the model would be able to predict the flow. While these numbers looked promising, the images told a different story. Figures 7.11 – 7.15 below show the results after running the comparison script which can be found in Appendix A.9.

The comparison script for version three of the model was nearly identical to that of version two. The paths are defined, the output folder is created, the test cases are chosen, the U-Net architecture is defined, the model is loaded, and the for loop is initiated. Inside of the for loop, the ground truth data is generated, the SDF is normalized to match the normalized training data, the AI model makes a prediction of the test case, and the prediction is denormalized. Finally, a for loop is used to plot the data containing the CFD data, the AI model's predicted data, and the Absolute Error between them.

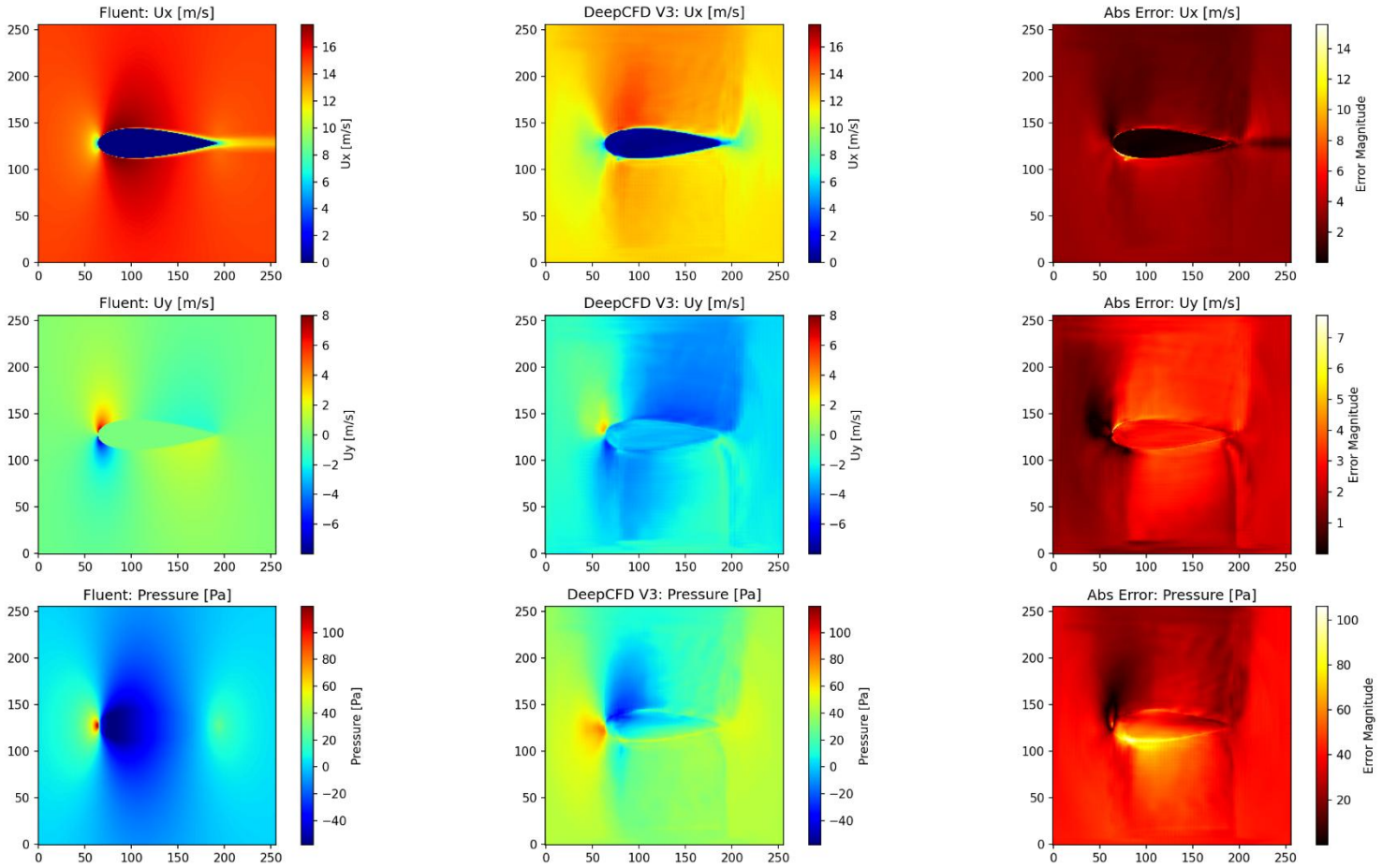


Figure 7.11 – Comparative fluid flow field analysis of v3: NACA0012 at $\alpha = 0°$

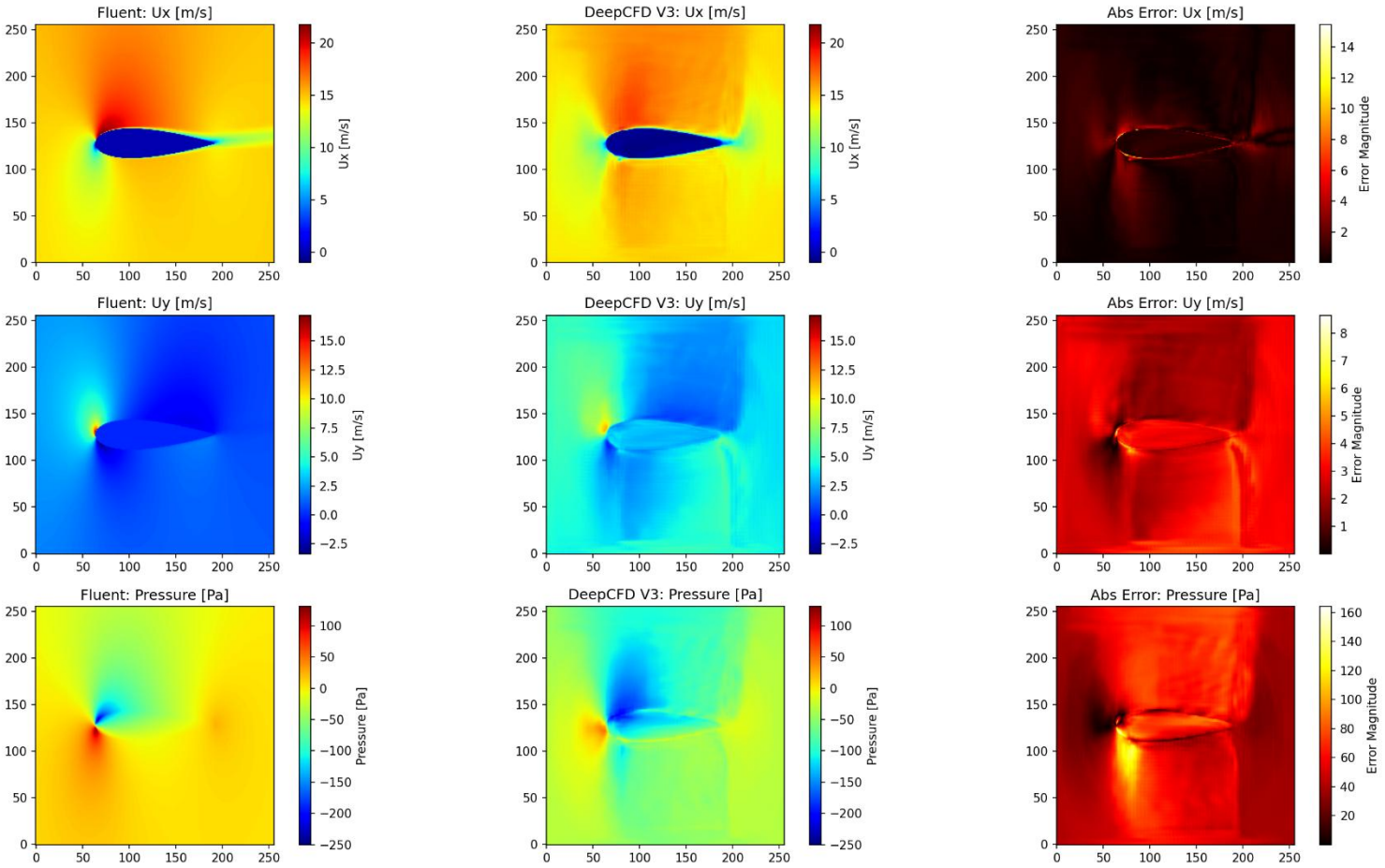


Figure 7.12 – Comparative fluid flow field analysis of v3: NACA0012 at $\alpha = 10°$

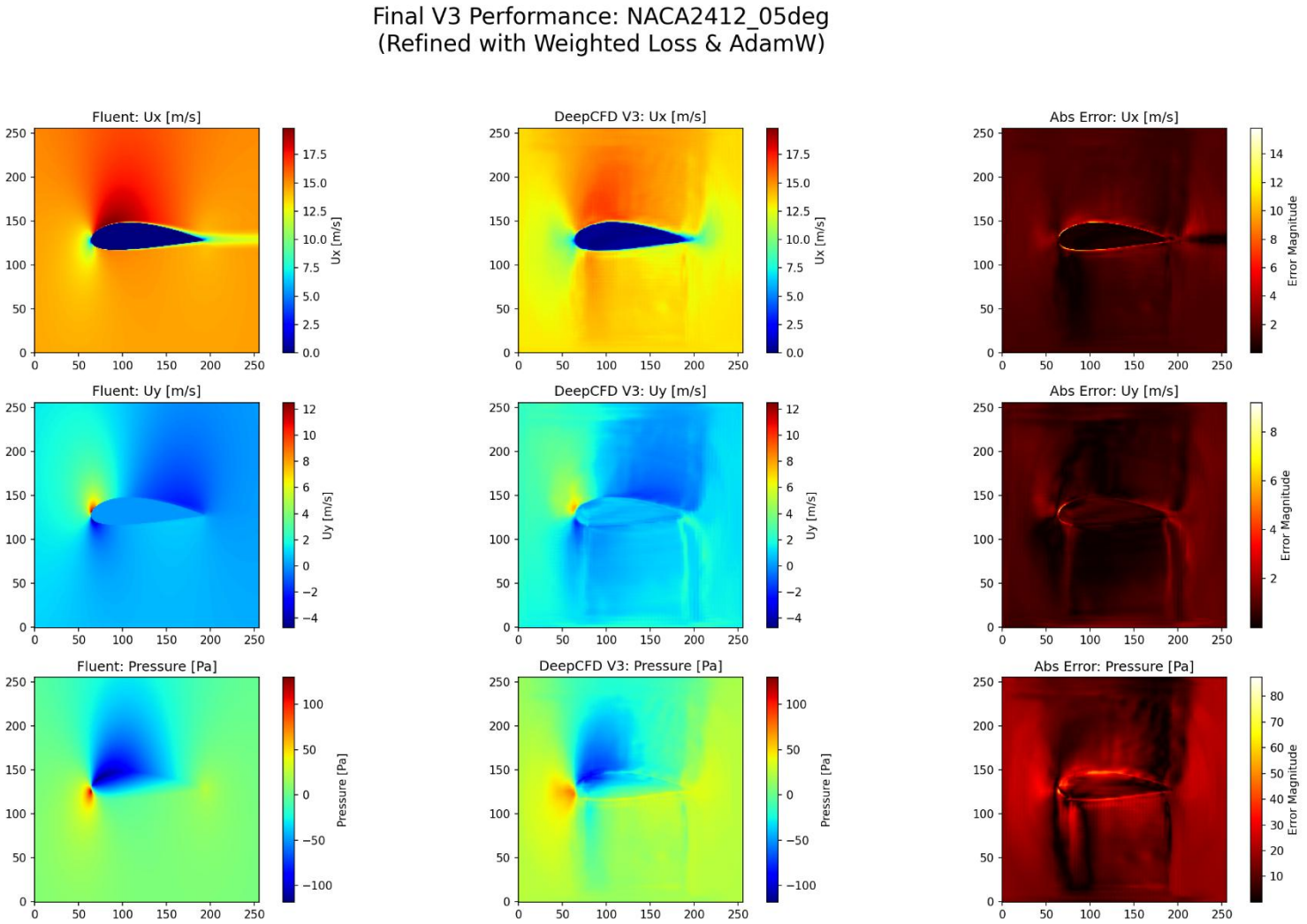


Figure7.13 – Comparative fluid flow field analysis of v3: NACA2412 at $\alpha = 5°$

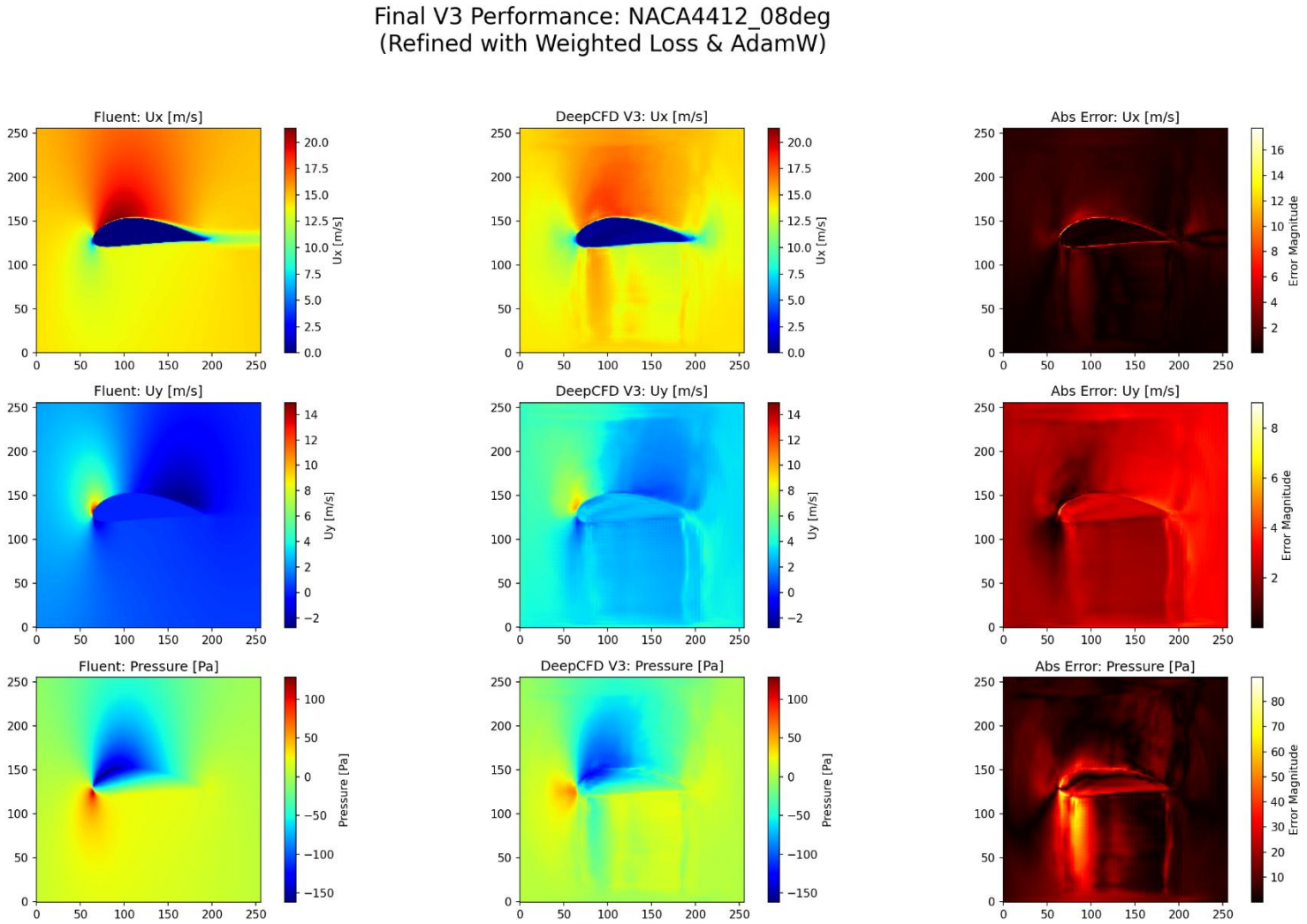


Figure 7.14 – Comparative fluid flow field analysis of v3: NACA4412 at $\alpha = 8°$

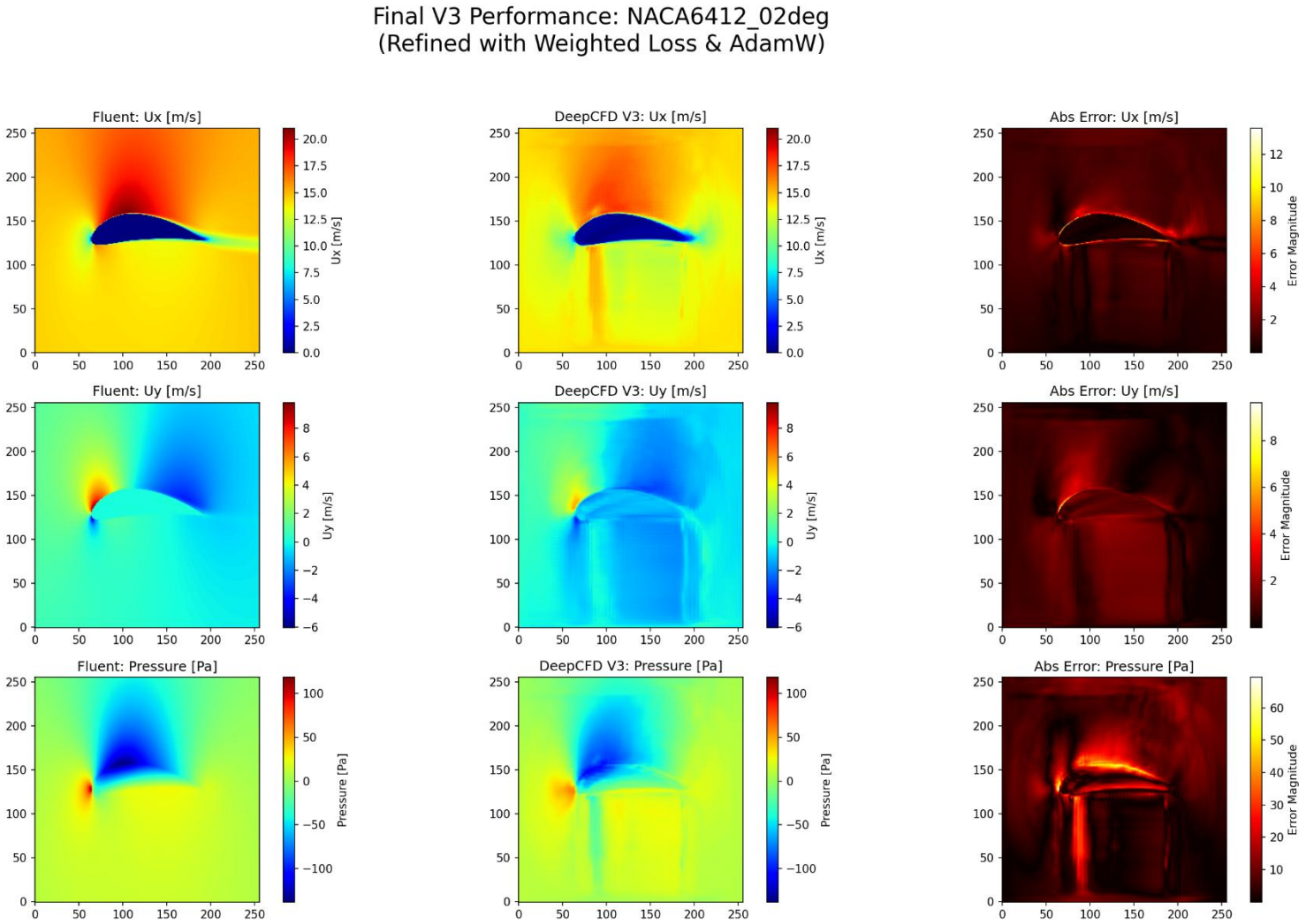


Figure 7.15 – Comparative fluid flow field analysis of v3: NACA6412 at $\alpha = 2°$

The comparison results of version three were not that different than version two. It seemed as though the model had reached a plateau in which it was no longer improving. Analyzing figures 7.6 – 7.15 and their respective counterparts between version 2 and version three, it is evident that overall, there had been some small refinement within the flow field but not a significant amount. The error was still high throughout the U_x, U_y, and p field. Even though the model had trained for an additional 100 Epochs, the model had not been refined as initially thought. Initially, it was thought that the model had reached a plateau, but that seemed odd considering that there was very little change from 100 Epochs to 200 Epochs. While trying to identify what parts the AI model was having trouble capture, it was noticed that the ground truth data concerning the pressure was corrupted. The pressure gradient seemed to be inside of the airfoil, completely ignoring the boundaries. Figures 7.11-7.15 further showed that the AI model was having a harder time identifying the pressure than the velocity components. While it was unclear whether this was the limiting factor preventing the AI model from converging closer to the solution, it was clearly a mistake that was hurting the training.

7.4 Boundary Challenge

Human beings tend to be good at knowing what direction they want their code to take, and computers are really good at reading over commands and executing them. However, one of the trickier aspects of coding is being able to effectively communicate the intent to the computer so that it may execute the commands as desired. At times, finding these miscommunications can be extremely time-consuming and frustrating. In the end, most of the time, the fix tends to be something simple, usually a misunderstanding or oversight into how the computer is executing the command.

Initially it was believed that the ground truth data had been corrupted at the moment of generating the data. However, once the simulations were re-loaded inside of Ansys Fluent it was clear that the data was correct and not corrupted. Therefore, the corruption most likely occurred at the time of importing the data into PyCharm. Following the computer logic, it was noticed that the issue had occurred inside of the initial Gridder script. The SDF created a 256 x 256 grid that held the distance of every single pixel to the closest airfoil boundary. The Gridder then came in and assigned values for U_x, U_y, and p into every single one of those pixels. The issue is that interpolation had to be used in order to match the mesh data with the pixelated data. The interpolator saw low pressure at the top of the airfoil and high pressure at the bottom of the airfoil and simply filled in those values in between with corresponding values. The result is that the pressure effectively ends up bleeding into the airfoil shape. It was noticed that this did not occur for U_x and U_y. The reason for this is the physics behind the velocity. During the physics set up for the 500 simulations, the no slip condition was used. The no-slip condition in fluid dynamics states that a viscous fluid has velocity of 0 at the surface of a boundary. This is what creates that boundary layer with a velocity gradient. When the Gridder sees these values, they decrease the closer they get to the surface due to the velocity gradient. Effectively becoming 0 as they reach the surface. Thus, the interpolation reaches a point where the values are set to 0 at the surface of the airfoil. Additionally, since the top and bottom of the airfoil are set to 0, the pixels in between, representing the airfoil shape, are also set to 0. The no slip condition effectively helps create the airfoil boundary. However, this was done by mistake rather than by design. To address this issue and solve the pressure gradient problem a new masked SDF and Gridder script had to be implemented. The job of the masked SDF will be to assign distances to each individual pixel. These distances will be how far away they are from the nearest point to the airfoil surface. Then the shape of the airfoil will be used to see if the values are inside of the airfoil or outside. If inside the values will be set to be negative and if they are outside of the airfoil, they will be kept positive. The Gridder script will then use this information to ensure that the values inside of the airfoil are all set to 0 for U_x, U_y, and p.

The new masked SDF script can be found in Appendix A.10 and the new Gridder script can be found in Appendix A.11. The logic for the SDF script follows very closely to the previous script. However, in this case two additional instructions were added. First, the points making up the airfoil are connected letting the computer know there is an existing boundary. Next, all points are checked to see if they are inside or outside of this boundary. If they are inside of the boundary, then the values are set to be negative. This logic is then used inside of the Gridder script. The new Gridder script is also similar to the old version. In the newer version the script looks for the x and y coordinates inside of the CSV file. It then proceeds to find the grid the three channels through

interpolation. However, this time around the fix is implemented. The script looks for the corresponding SDF file. It then looks for the values that are negative in the SDF file. This tells the Gridder script the points inside of the airfoil. The Gridder proceeds to grid these values as 0 for the three channels ensuring there are no velocity or pressure values bleeding into the airfoil.

8. Final Training and Results

8.2 Version Four

Once the boundary issue had been fixed a new version of the code was run to see the differences between the models. It was theorized that the bleeding is what was limiting the model from learning adequately. This new model was trained for 300 Epochs to allow it adequate time to learn the physics. Additionally, the batch size was increased from 4 to 10. This was done to increase the variety of shapes and αs that the model could see at the same time. Additionally, the model's Dropout rate was decreased from 20 percent to 10 percent. This was done so that the model could have more neurons during training. It was believed that the 20 percent Dropout rate that was previously chosen had limited the model's ability to train effectively. Thus, by reducing it to 10 percent, the model would have more neurons during training while still preventing the model from memorizing the dataset. Additionally, since the model was training for 300 Epochs, it was believed that the Dropout effect would compound overtime allowing the model to learn better. Additionally, the learning rate used in the AdamW optimizer was set to 0.001, giving the model a bit more freedom and not being so restrictive as it was going to be training for a longer time than before. This code also includes the min-max normalization that was used in previous versions to ensure the model isn't dealing with a wide range of values. Additionally, to reduce high CPU temperatures after every 5 Epochs, a small break of 60 seconds was taken to reduce temperatures. Additionally, the model was saved every 10 Epochs. Previously the training and validation sets were randomized. In this version of the code specific cases were used to validate the training. A split of 440 airfoils was used for training and 60 were used for validation. The reason for this change is that a wider range of airfoils was desired for comparison. Thus, the thicknesses 12, 18, and 26 percent were chosen. Additionally, each one of these thicknesses fell under the four categories of symmetrical, low camber, medium camber, and high camber airfoils. Additionally, each one of these had 5 corresponding αs, bringing the total of airfoils to 60. This was perfect as it gave a picture for the four categories, all αs, and different thicknesses; while being less than the previous validation stack. The script for this can be found in Appendix A.12.

Additionally, the script for generating the comparison was similar to the previous scripts. The file paths were defined. The U-Net architecture used to train the model was redefined to load up an empty model that would be filled later by the predicted results. The main differences were that this time around 60 specific cases were used to validate the AI model's learning. As mentioned above these were chosen to test the model in a wide range of cases and ensure proper learning was occurring. The way the data was plotted remained the same. The ground truth CFD data was plotted on the left column, the middle column held the AI predicted values after they had been denormalized, and the right column contained the absolute error between the ground truth CFD data and the AI predicted data. The accompanying script can be found in Appendix A.13.

The following images below show a wide variety of cases that were used to test the validity of the AI model's performance compared to the ground truth CFD data acquired through Ansys Fluent. Figures 8.1 – 8.9 show the performance of the trained model on varying airfoil thicknesses, camber, and αs.

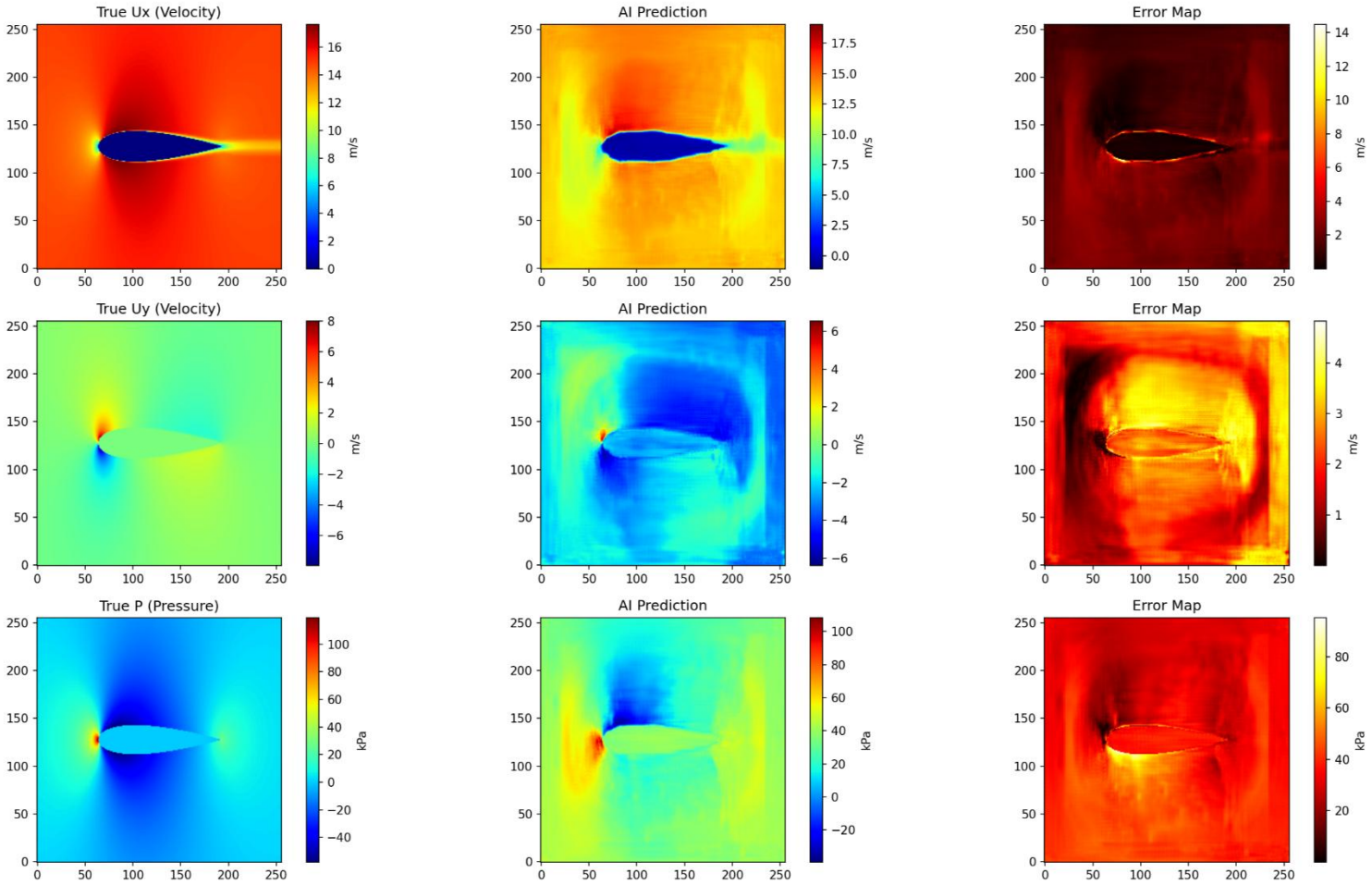


Figure 8.1 – Comparative fluid flow field analysis of v4: NACA0012 at $\alpha = 0°$

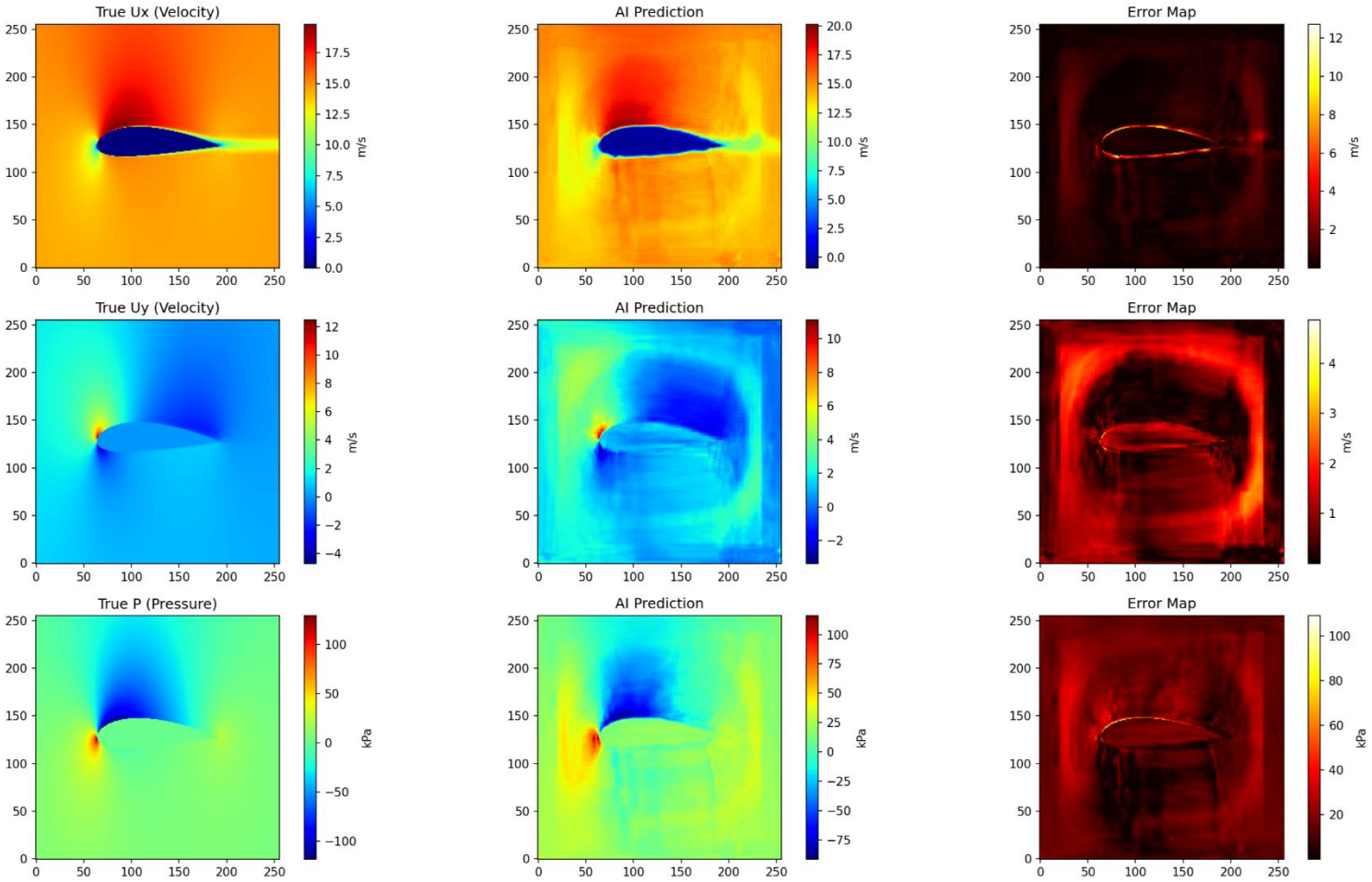


Figure 8.2 – Comparative fluid flow field analysis of v4: NACA2412 at $\alpha = 5°$

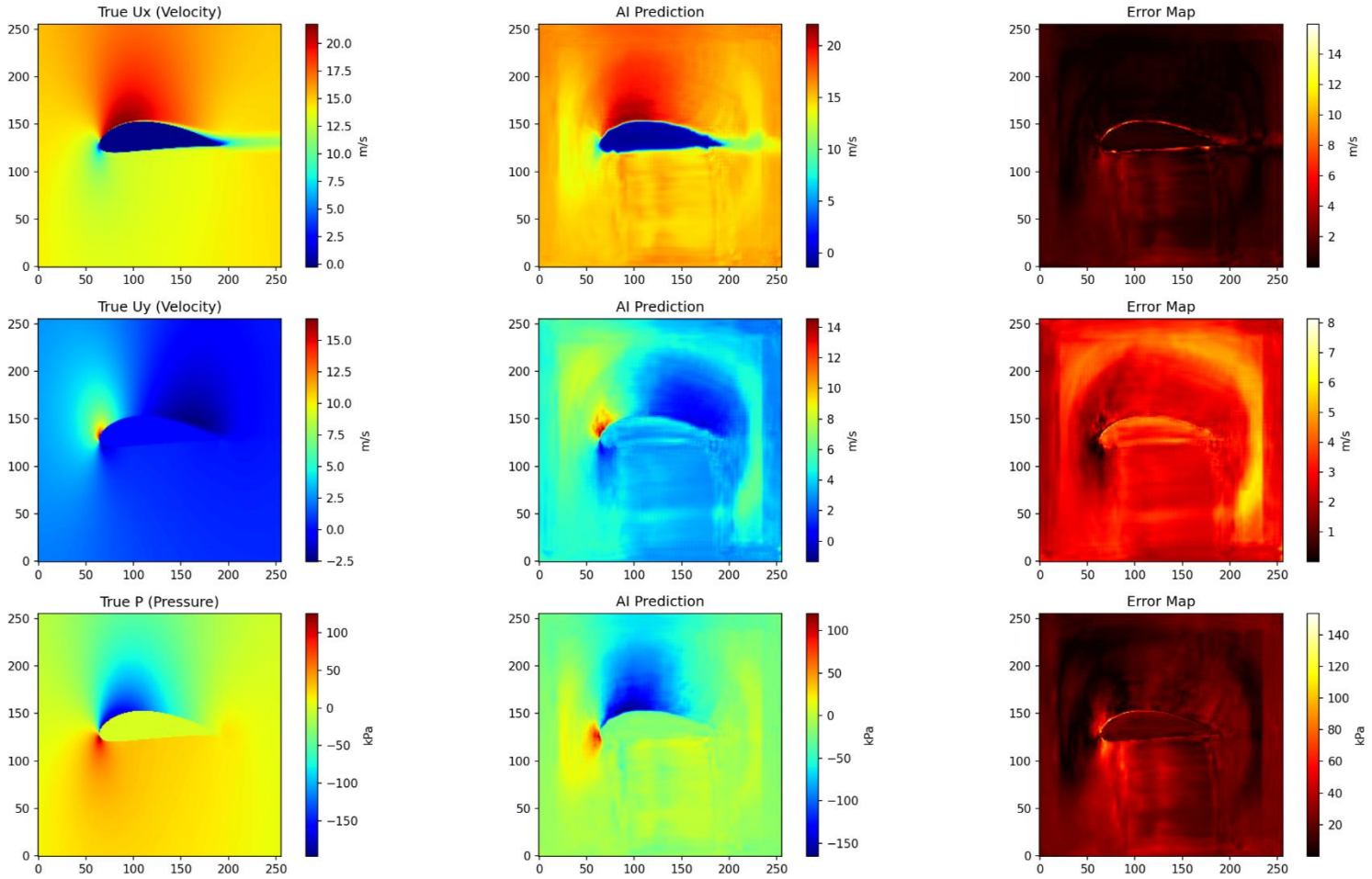


Figure 8.3 – Comparative fluid flow field analysis of v4: NACA4412 at $\alpha = 10°$

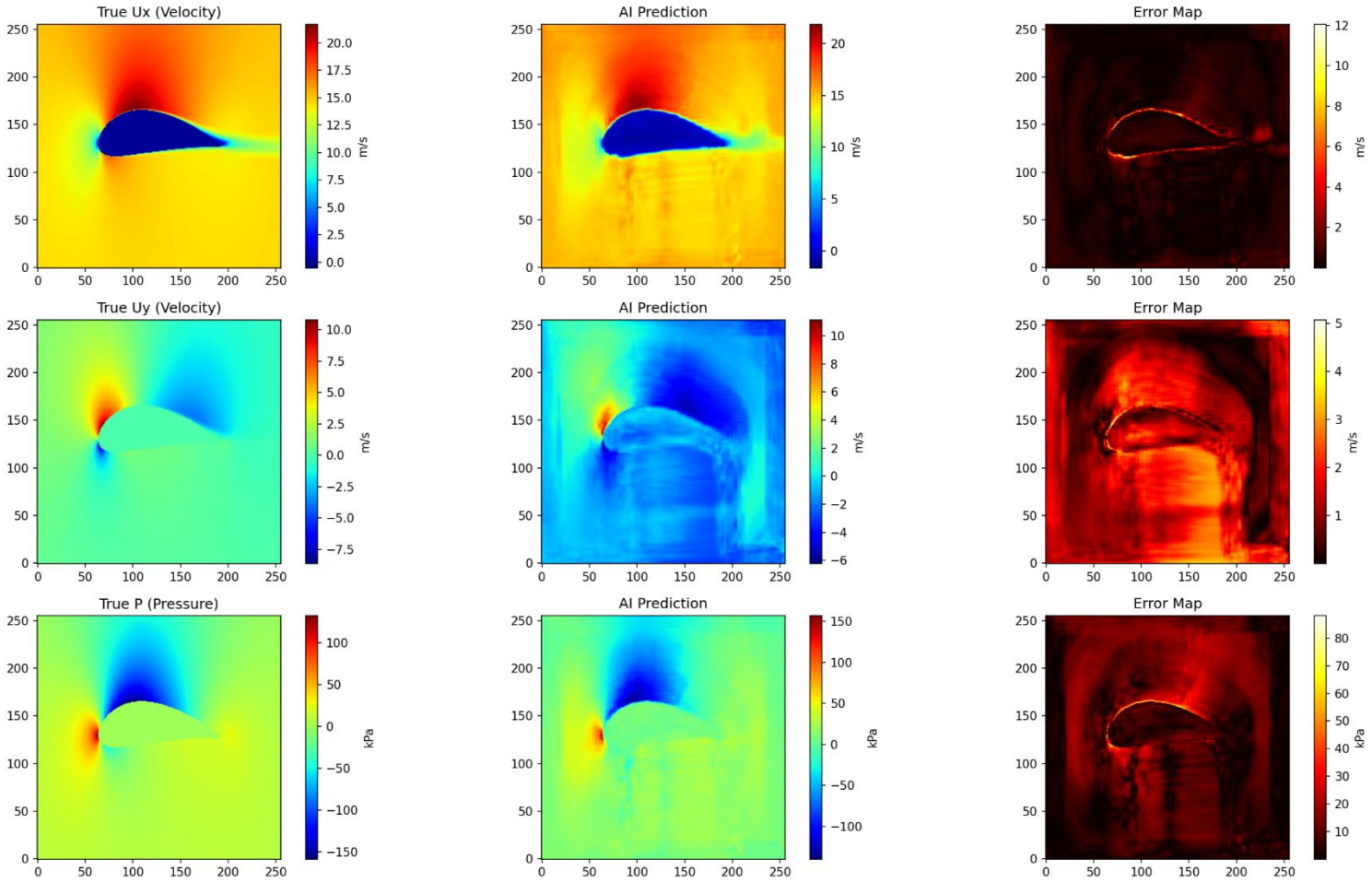


Figure 8.4 – Comparative fluid flow field analysis of v4: NACA6418 at $\alpha = 2°$

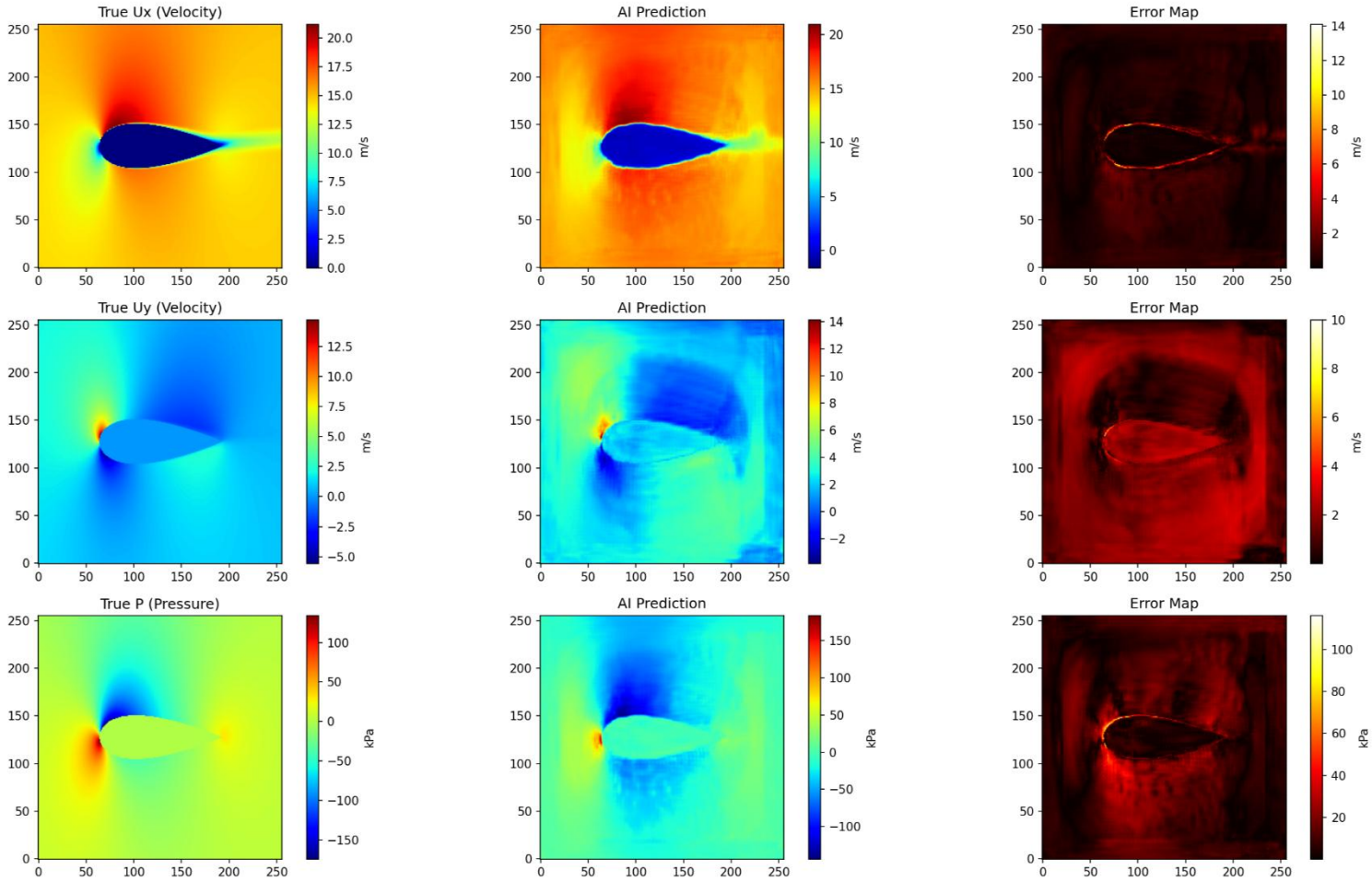


Figure 8.5 – Comparative fluid flow field analysis of v4: NACA0018 at $\alpha = 8°$

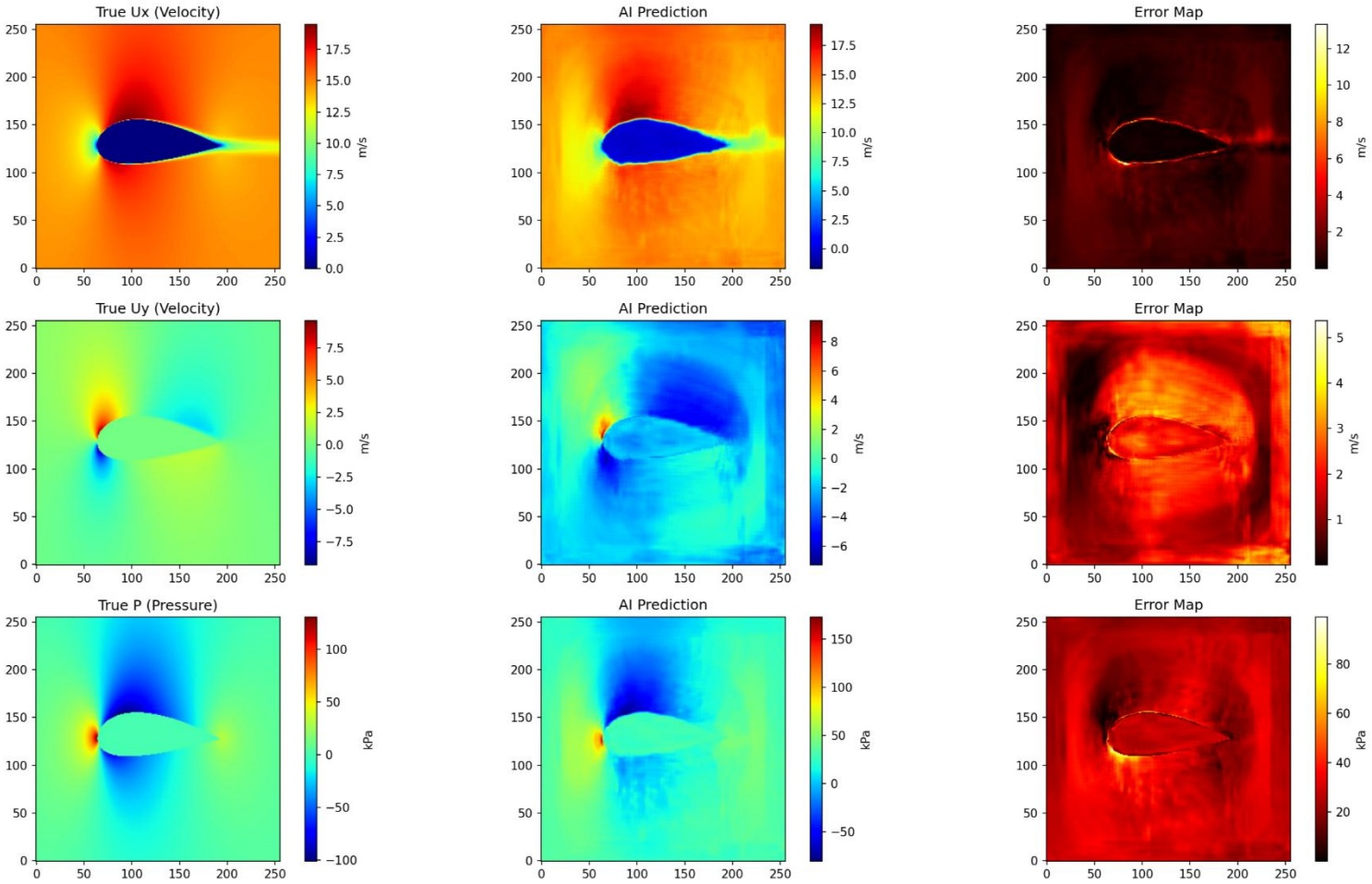


Figure 8.6 – Comparative fluid flow field analysis of v4: NACA2418 at $\alpha = 0°$

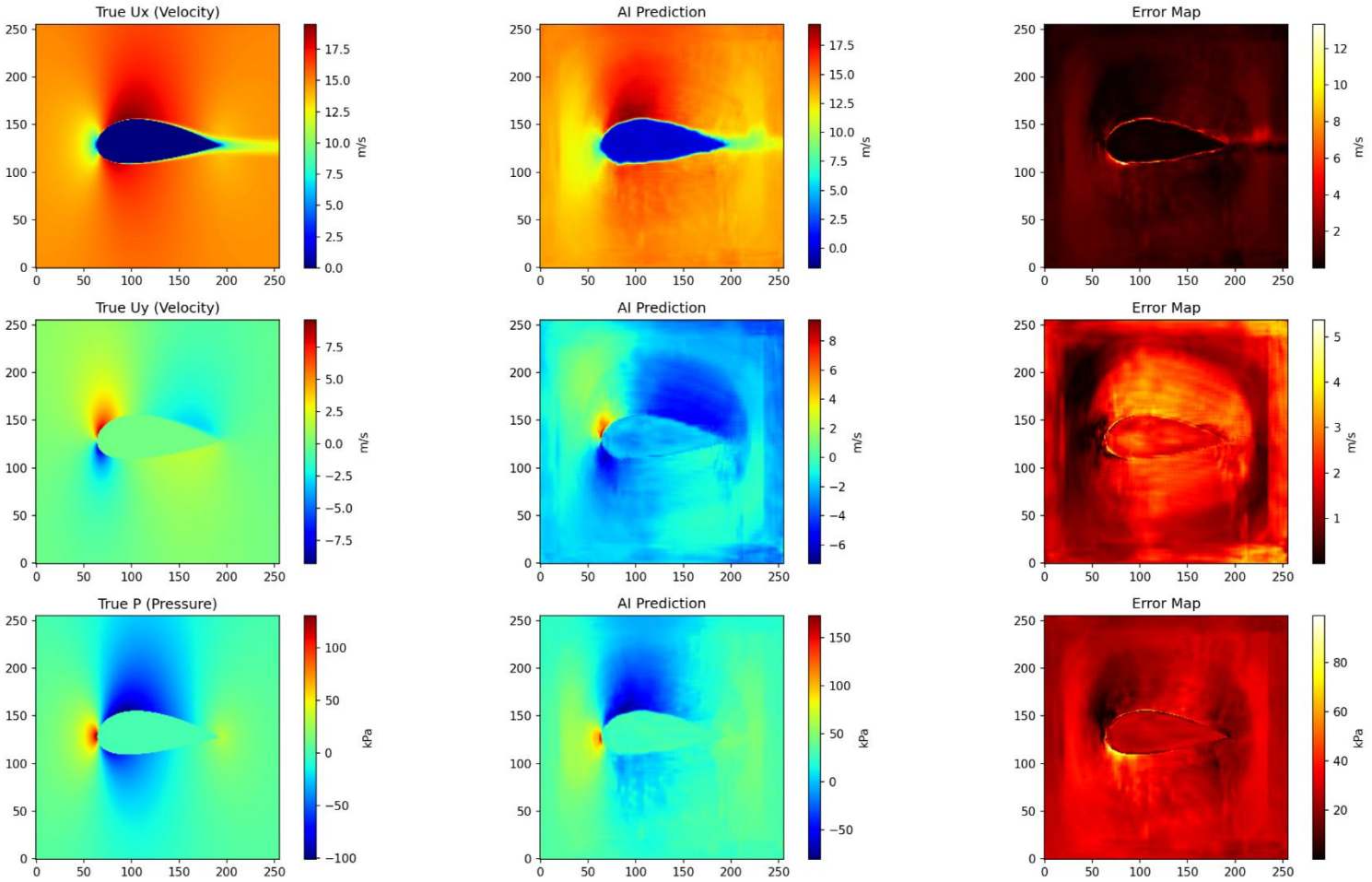


Figure 8.7 – Comparative fluid flow field analysis of v4: NACA4426 at $\alpha = 5°$

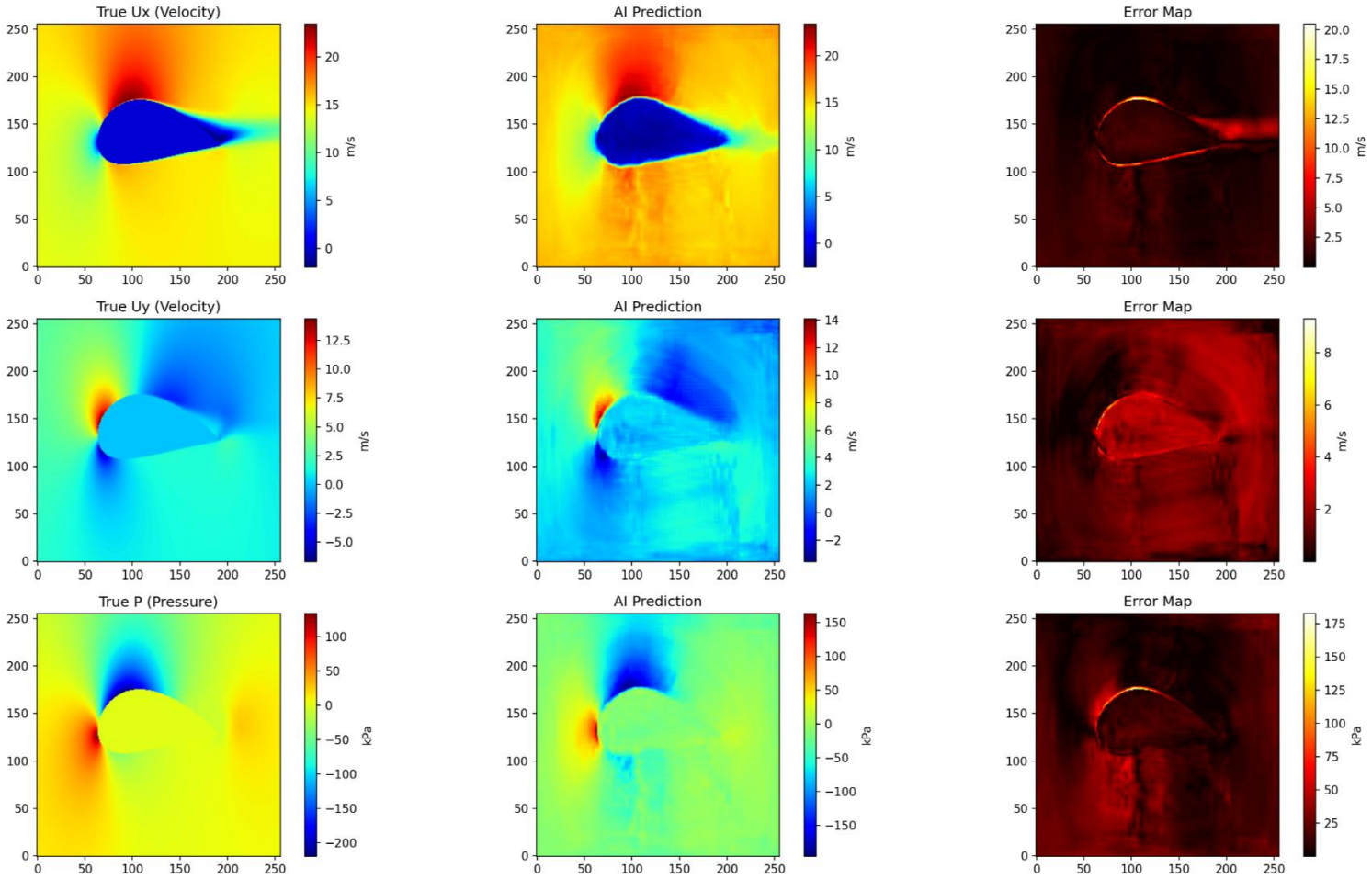


Figure 8.8 – Comparative fluid flow field analysis of v4: NACA6426 at $\alpha = 10°$

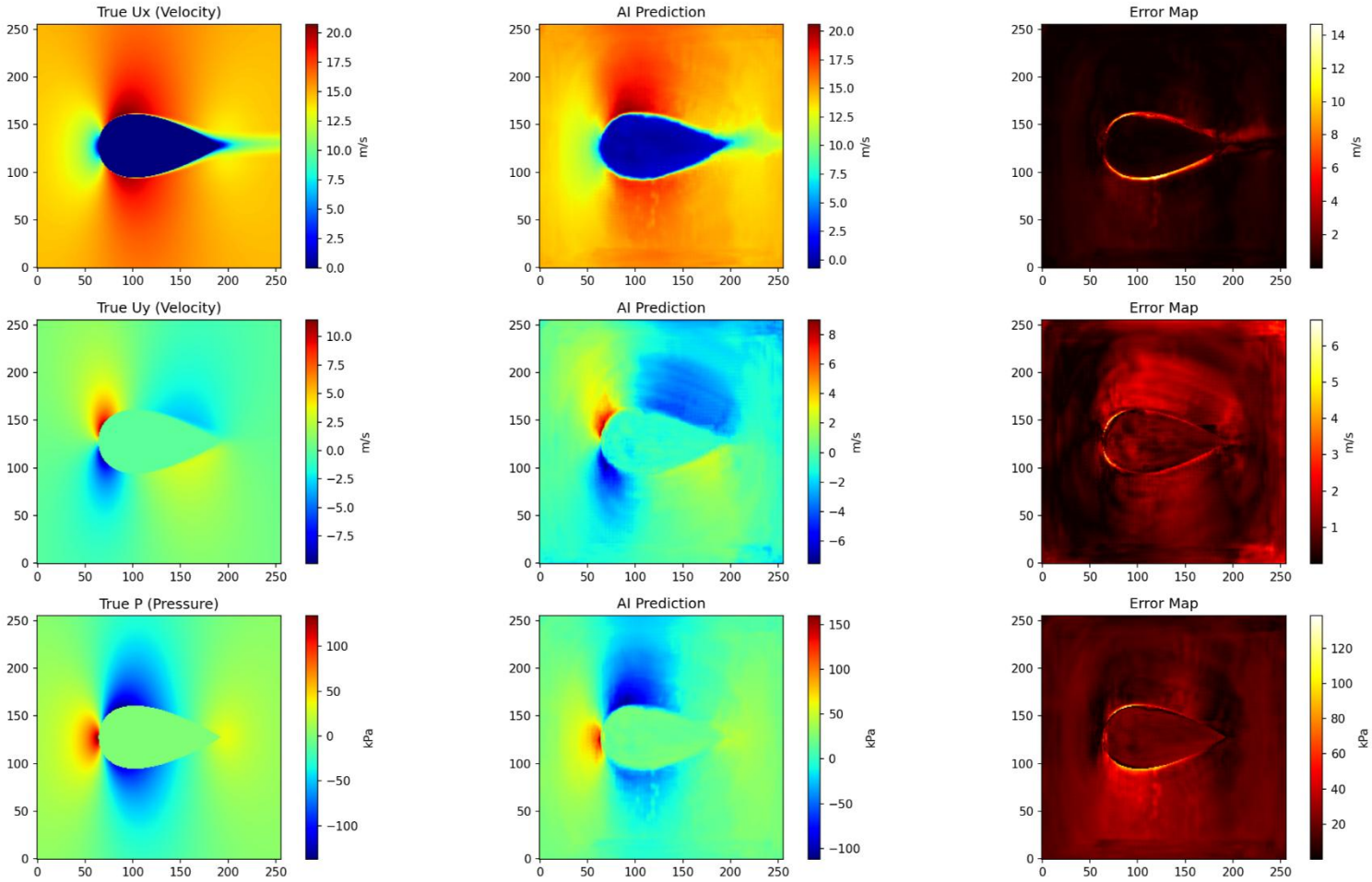


Figure 8.9 – Comparative fluid flow field analysis of v4: NACA0026 at $\alpha = 2°$

Figures 8.1 – 8.9 show a clear improvement in training from the earlier versions in which the airfoil boundary issue was occurring. The model is clearly picking up the airfoil shape and not showing bleeding into the inside of the airfoil. Additionally, the U_x channel has converged significantly closer to the ground truth CFD data. This can be noticed in both the AI prediction column as well as the Error Map column. In all cases, the U_x channel has significantly lower absolute error. The majority of the absolute error occurs at the boundary layer, which is expected. The boundary layer is small yet contains the most important physics relating to fluid flow. Similar to previous models, the model seems to struggle a bit more with symmetrical airfoils at 0° α, as seen in figure 8.1. It still predicts higher velocity at the top of the airfoil, even though at 0° α, the velocity at the top and bottom should be the same. At varying α's, the model performs better. The prediction results align quite nicely with the ground truth results. Figure 8.7 depicts this well. The AI predicted column has the higher airflow at the top of the airfoil, with the stagnation point defined well, and the wake being predicted to a good level as well. Additionally, the absolute error map in this case shows the better global prediction, with the boundary layer being the exception.

Moving on to the U_y channel, this model shows improved performance to previous models. There is still much refinement to be made within the overall smoothness; yet the model shows a rough understanding of the sharp velocity changes. Comparing thin airfoils to thick airfoils such as figures 8.2 and 8.7 respectively reveal an interesting find. The model has an easier time understanding U_y for thicker airfoils than it does for thinner airfoils. In this version of the code, U_y is the worst performing channel from the three channels.

When it comes to the p channel, this version proves that the boundary issue was preventing the model from accurately learning the behavior of the pressure. Figures 8.4,8.7, and 8.9 show that the model is learning to predict the pressure at medium αs very well. The prediction columns match up very closely with the ground truth data in these scenarios. Additionally, the Absolute Error map shows the least error in these scenarios. Figures 8.3. 8.5, and 8.8 reveal that the AI model is struggling a bit more when it comes to higher αs. The model seems to lean more towards predicting lower pressure values for higher angles of attack than the true values. Additionally, the model is struggling greatly with symmetrical airfoils at 0° α, as seen in figure 8.1. It can be seen that this particular case presents the greatest Absolute Error for p than all the other cases.

While the model was training, a log was implemented that displayed how certain parameters were behaving at each Epoch. The most important numbers that were logged were the Training Loss and the Validation Loss. The Training Loss after 300 Epochs was 0.00639 and the Validation Loss was 0.00575. These values are quite meaningful, for starters the magnitude of the numbers; which is less than one percent, shows that on average the difference between the ground truth CFD data and the AI predicted data is extremely small. This further emphasizes that the model is actually learning the physics. Additionally, it is known that the model is learning the physics because there is no overfitting occurring. Overfitting, as previously mentioned, occurs when the model begins to memorize the data rather than learn from the data. Usually this occurs when the Training Loss is lower than the Validation Loss, by a big margin. In this case, however, the Validation Loss is lower than the Training Loss. This can be attributed to the use of the Dropout Method that shuts off a percentage of neurons during training and turns them back on during validation. The model is considered to be less intelligent during training because it has less neurons to work with forcing the neurons to learn the physics. During validation, the neurons are turned back on increasing the intelligence of the model, allowing it to perform better.

Overall version four was very important to the development of the final AI model that was trained. Version four showed that the model was training well after 300 Epochs. It also showed that the fix to the boundary issue had worked well in allowing the model to converge to a more precise prediction. The model was performing well for U_x, it was beginning to learn p, but it was struggling most of all with U_y. To achieve a model that was closer to the ground truth data, two changes were made in version five; the normalization method and the Kernel Size.

8.3 Version Five

Changing the Kernel Size from 3 to 5 allows the filter to see a bigger area, allowing it to capture smooth gradients better. The Kernel can be thought of as the eye/filter that scans through the pixels of a layer learning patterns as it goes through. It is a square matrix defined by its size.

For example, a Kernel of size 3 is a matrix of 3 x 3 pixels. A Kernel of size 5 is a 5 x 5 matrix that captures a bigger field of the layer allowing to capture more of the physics and better gradient transitions. The hope was that by increasing the size of the Kernel it would be able help p and U_y converge closer to the ground truth data.

Additionally, for version five a new hybrid normalization technique was used. Previously the normalization technique that had been used was localized. It was the same technique that was used for the SDF. By taking the maximum and minimum values the values inside of each corresponding SDF file and using them to normalize the airfoil shape the AI model was able to learn what the shape of the airfoil looked like quite effectively. However, this works because each independent airfoil has a shape that is different from the other 99 airfoils in the dataset. When it comes to learning the physics behind U_x, U_y, and p it is crucial that the AI does not treat these cases as completely independent from one another. When the localized normalization method is used, the minimums and maximums that the AI model sees are only specific to that scenario. Thus, when it comes to figuring out the underlying physics, the AI model struggles because it has no way to connect the physics from each independent airfoil. In order to connect the physics, a global normalization technique must be used for the physics regarding U_x, U_y, and p. To do this a script is written to check through all 500 files and find the highest and lowest values for U_x, U_y, and p respectively. The logic of the script is very simple, the file path for the Gridded files is defined so that it can be opened up by the script and a list of each file can be made. The script then runs a for loop that opens each file one by one, the highest and lowest value for U_x, U_y, and p are found and stored. Once this is done, the script looks at the saved values and searches for the highest and lowest amongst these for U_x, U_y, and p. The result is the 6 values that define the global normalization parameters. The global maximum and minimum for U_x, U_y, and p. These now ensure that there is a link between each individual airfoil. The AI model will now be able to look at normalized values and know how they relate to one another from airfoil to airfoil and from varying α's. The script that was used can be found in Appendix A.14.

Additionally, one last thing that was changed in this version was the Weight Decay. During the training of the previous version an oversight occurred where the Weight Decay was not used even though the AdamW optimizer was implemented. In this version, the AdamW optimizer is used. The Learning Rate was set to be 0.001 and the Weight Decay was set to be 0.005. This was done to match the optimizer parameters that were used by the original author's of DeepCFD.

The training script for this fifth and final version can be found under Appendix A.15. The logic behind the script is as follows. Similar to all previous versions, the paths were defined for the appropriate locations that would be used. The Global Normalization Parameters were defined for later use. The 500 Gridded files containing the ground truth dataset were loaded into the script along with the 500 SDF files. Local normalization was used on the SDF files and Global Normalization was used on the Gridded files. The U-Net architecture is defined. In this case the Kernel Size is changed to 5 with a padding of 2. The padding serves a very important function. When the kernel is near an edge it will run out of pixels, this padding of 2 ensures that if the middle of the kernel is at an edge 2 more rows filled with 0's will be created in order to give the Kernel a 5x5 matrix. Once the U-Net architecture is defined the next step is to execute the training of the model. This is done by first loading up the dataset of 500 airfoils. The validation set of 60 is picked, and the training set and validation set can be loaded. The empty U-Net architecture is loaded

accordingly. The AdamW optimizer is set with a Learning Rate of 0.001 and a Weight Decay of 0.005. The training for loop is then initiated and runs for 300 Epochs. Thermal regulation is added into the for loop by telling the script to stop all calculations for 60 seconds every 5 Epochs. This was initially done, because the previous versions proved that overheating would occur after hours of constant training. Once the training was completed, the model was saved.

The final step was to run one final comparison script that would compare how this final model faired against ground truth data. The script can be found in Appendix A.16. The logic of this script is very similar to that of version four. The main difference is that in this script the U-Net architecture changed a bit since the Kernel Size was set to 5 to match that of the corresponding training script. Additionally, the Global Normalization Parameters were used to denormalize the data and compare it with the ground truth data. Figures 8.10 – 8.21 below show the results of the final trained AI model when compared to the ground truth data simulated using Ansys Fluent.

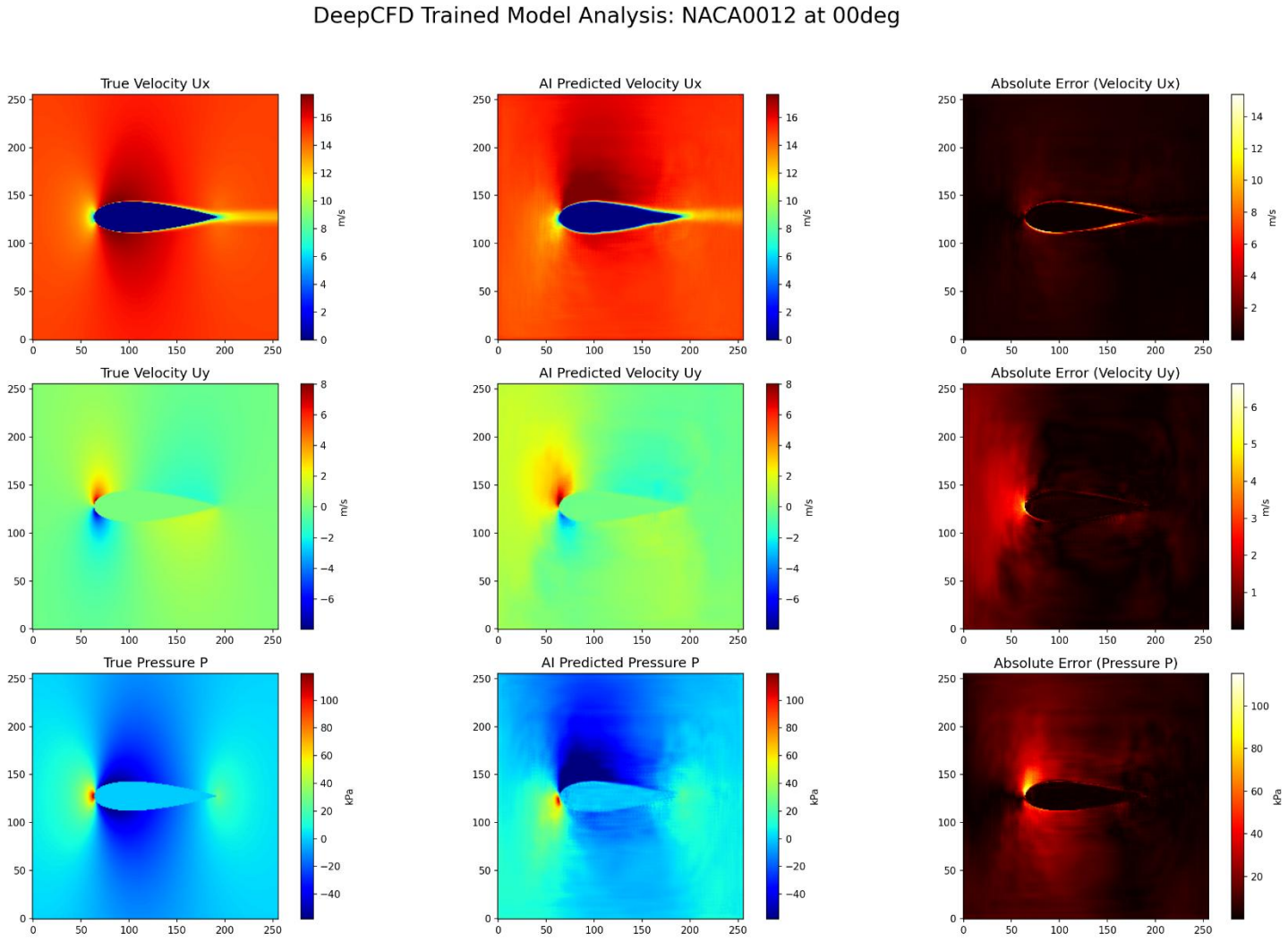


Figure 8.10 – Comparative fluid flow field analysis: NACA0012 at $\alpha = 0°$

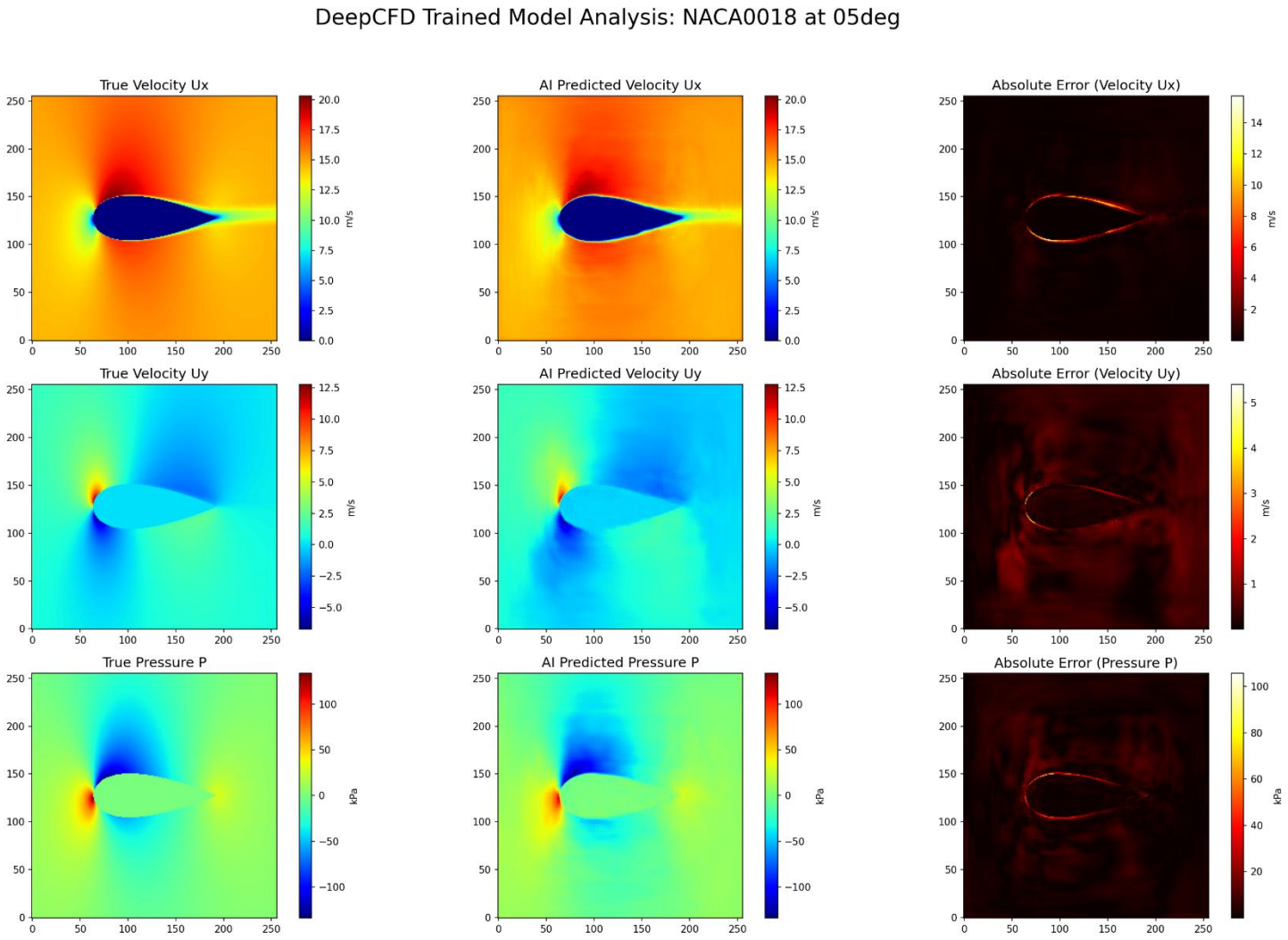


Figure 8.11 – Comparative fluid flow field analysis: NACA0018 at $\alpha = 5°$

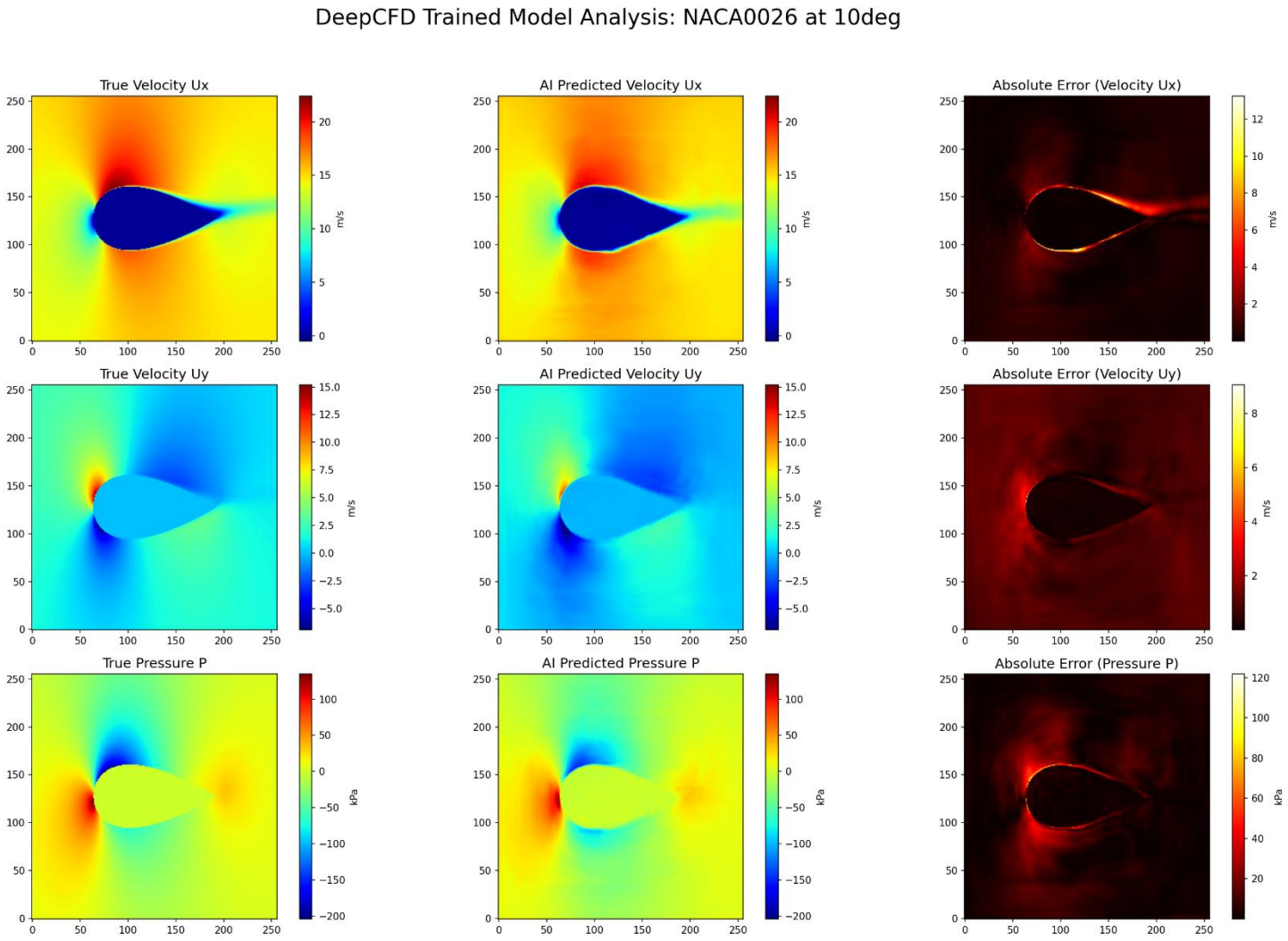


Figure 8.12 – Comparative fluid flow field analysis: NACA0026 at $\alpha = 10°$

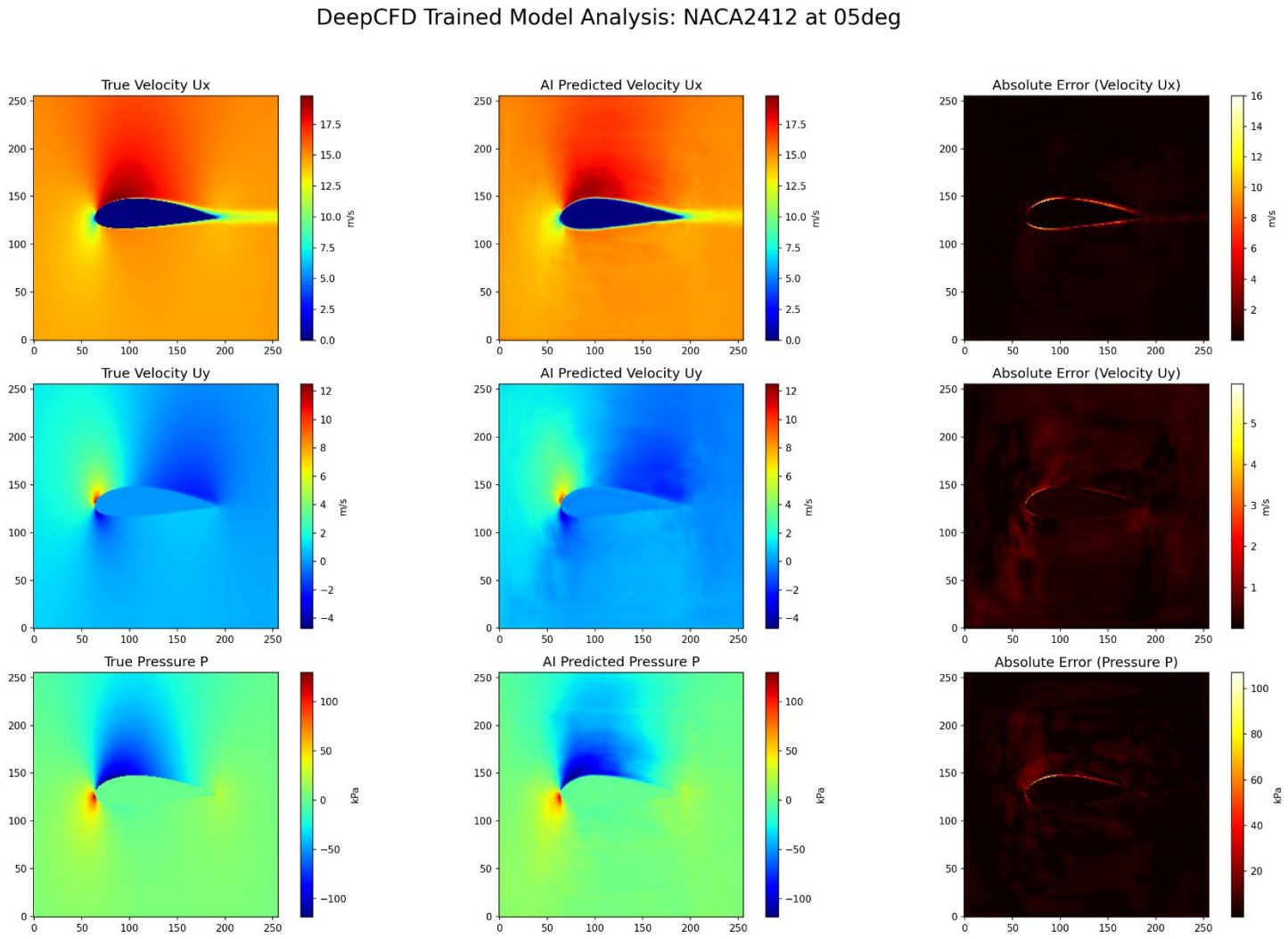


Figure 8.13 – Comparative fluid flow field analysis: NACA2412 at $\alpha = 5°$

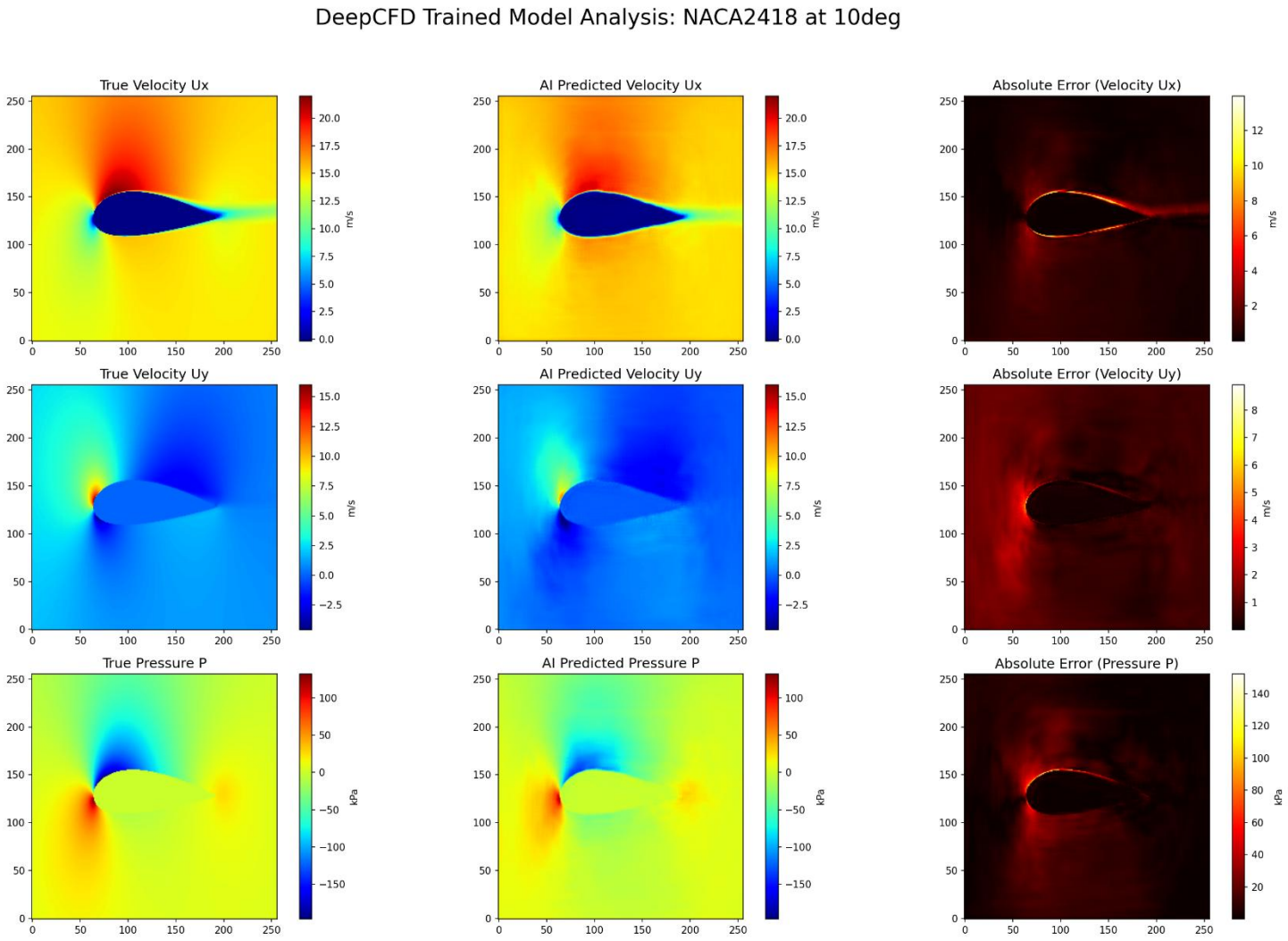


Figure 8.14 – Comparative fluid flow field analysis: NACA2418 at $\alpha = 10°$

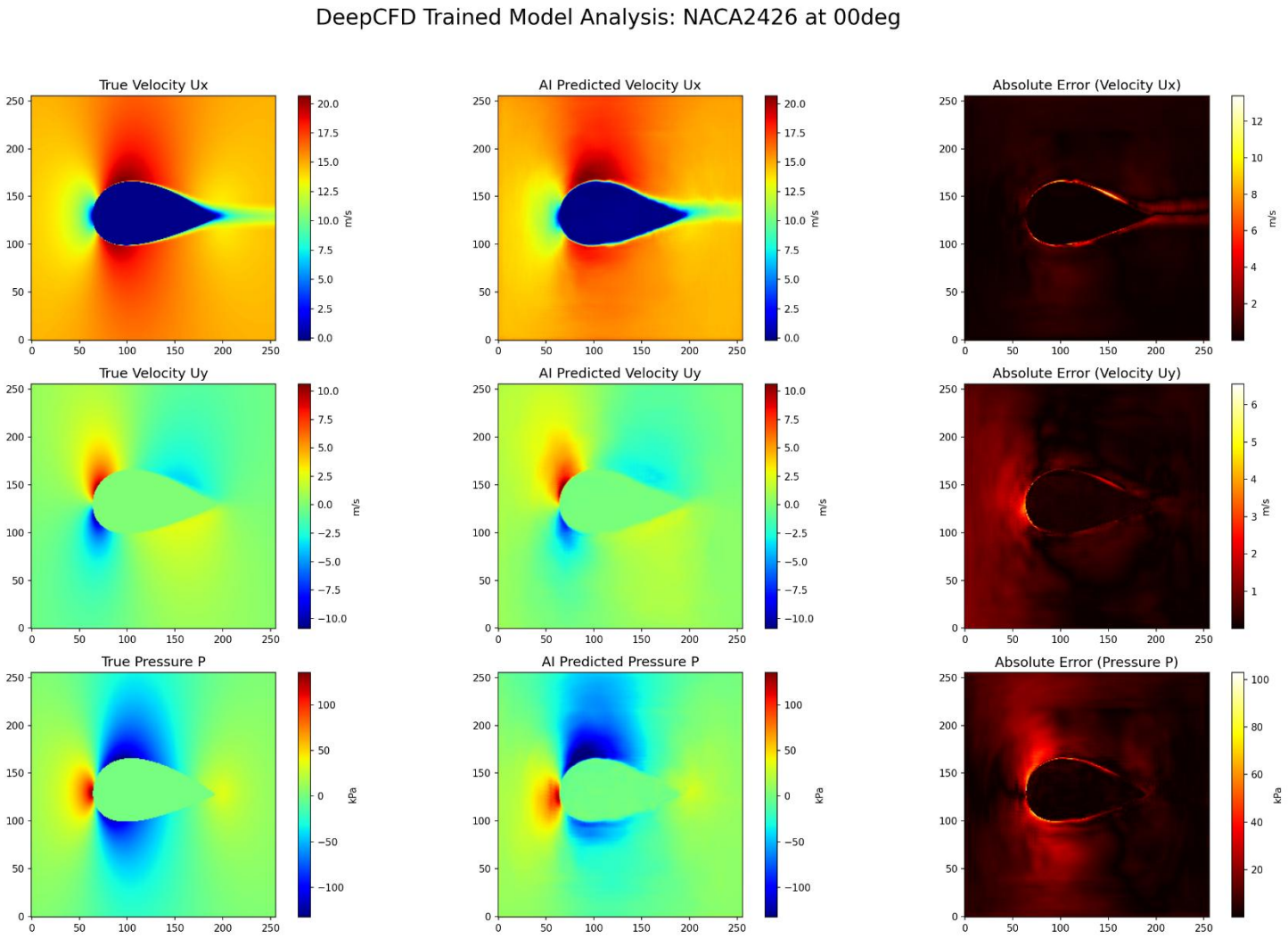


Figure 8.15 – Comparative fluid flow field analysis: NACA2426 at $\alpha = 0°$

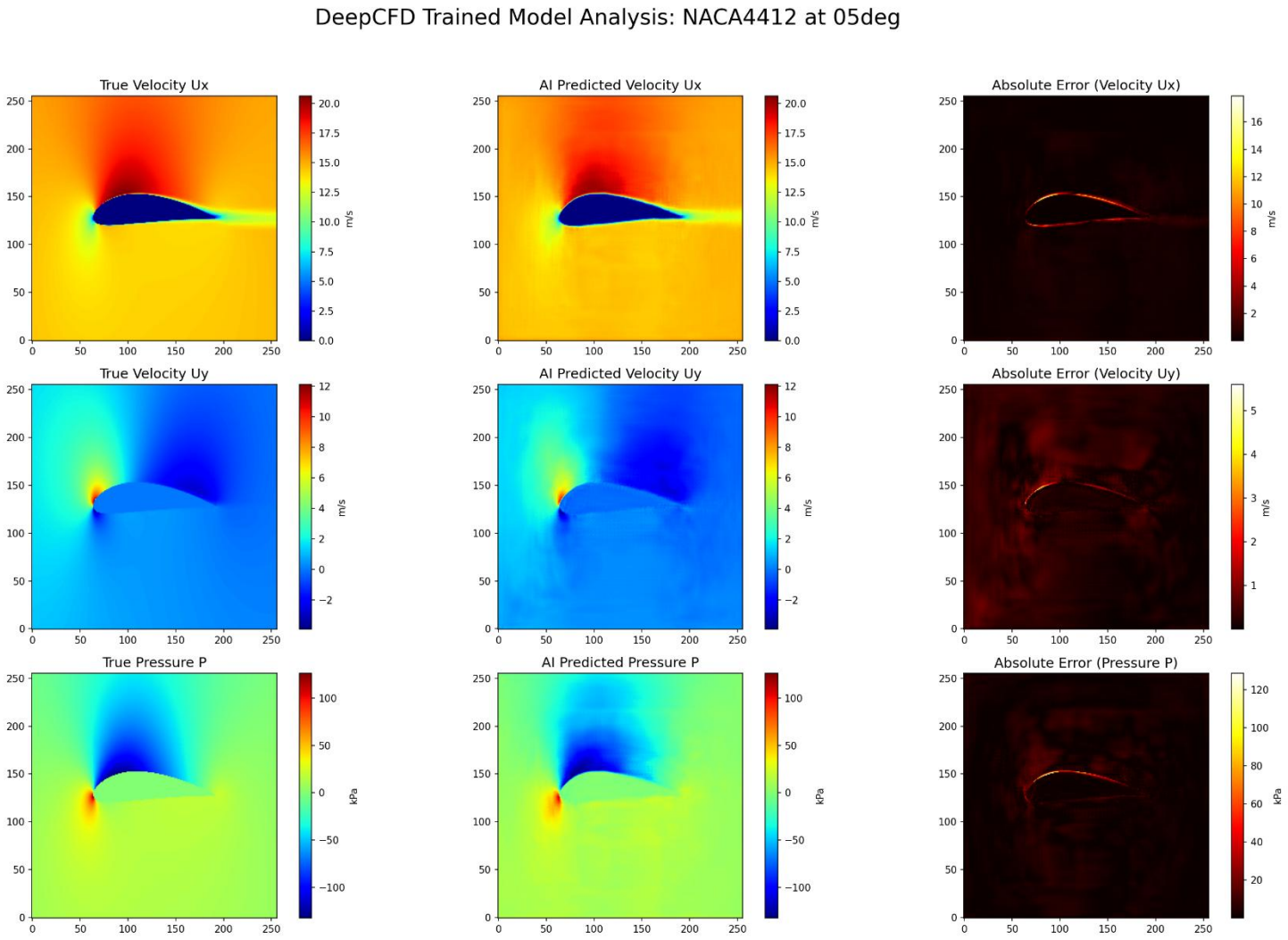


Figure 8.16 – Comparative fluid flow field analysis: NACA4412 at $\alpha = 5°$

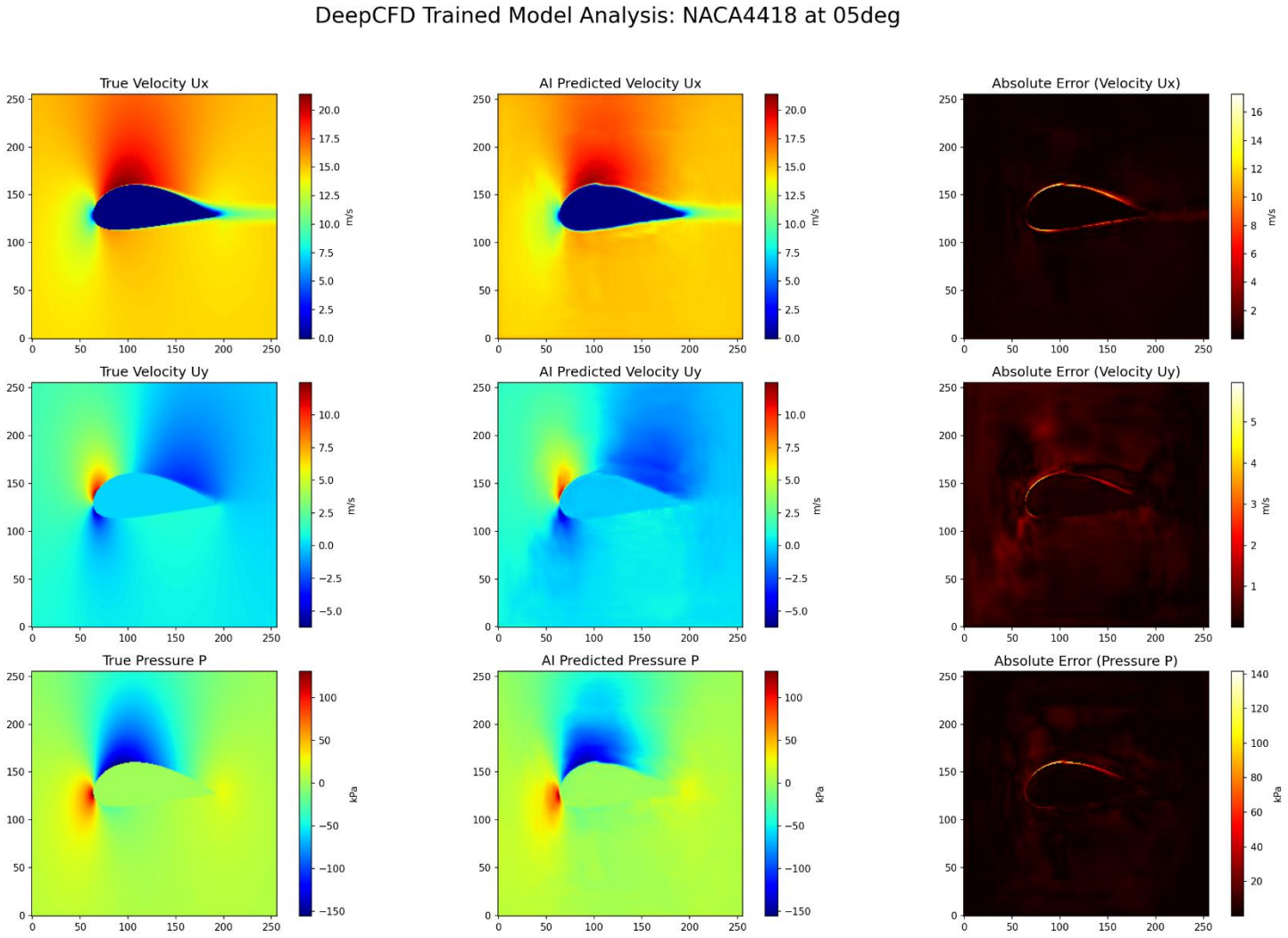


Figure 8.17 – Comparative fluid flow field analysis: NACA4418 at $\alpha = 5°$

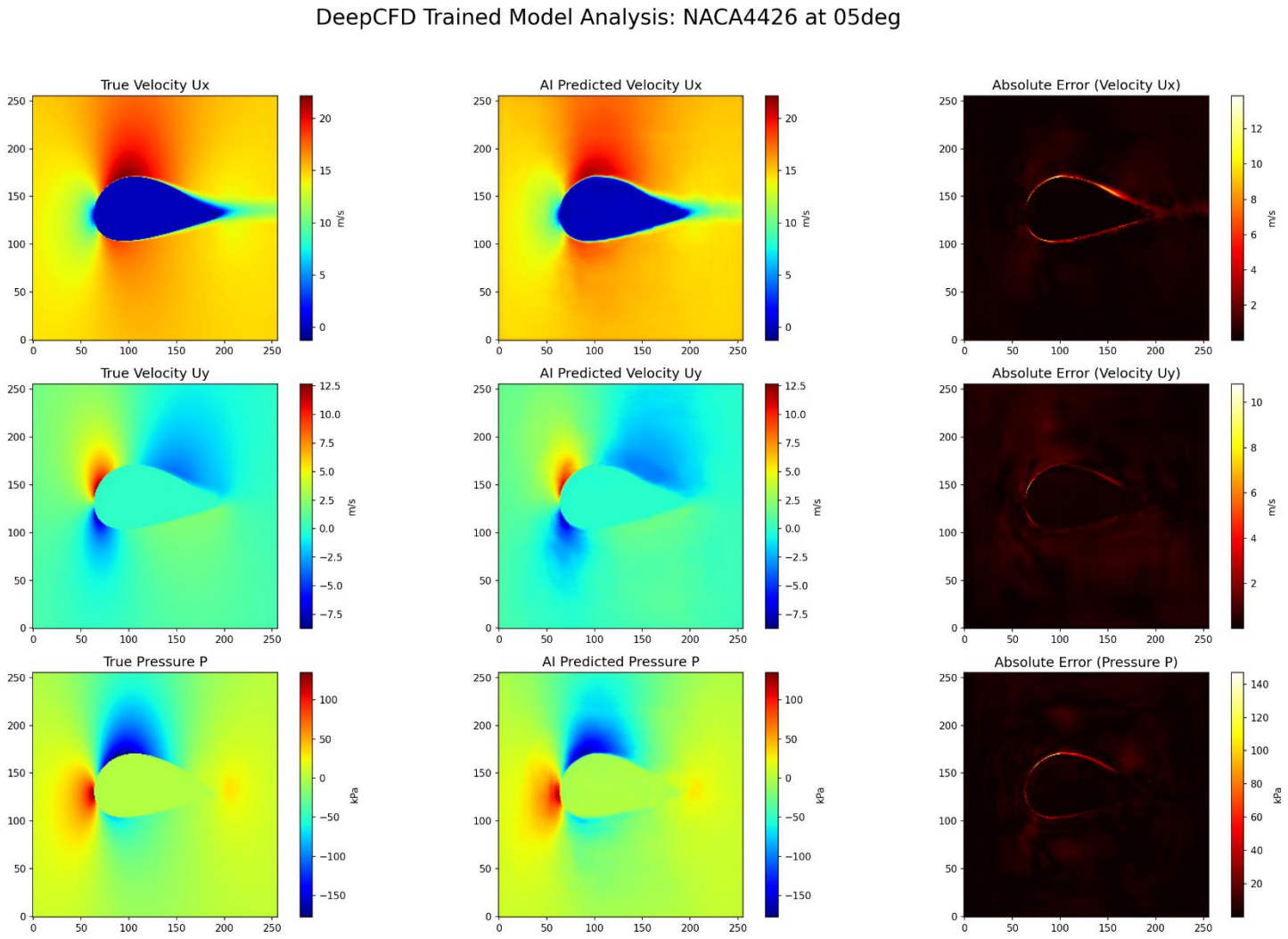


Figure 8.18 – Comparative fluid flow field analysis: NACA4426 at $\alpha = 5°$

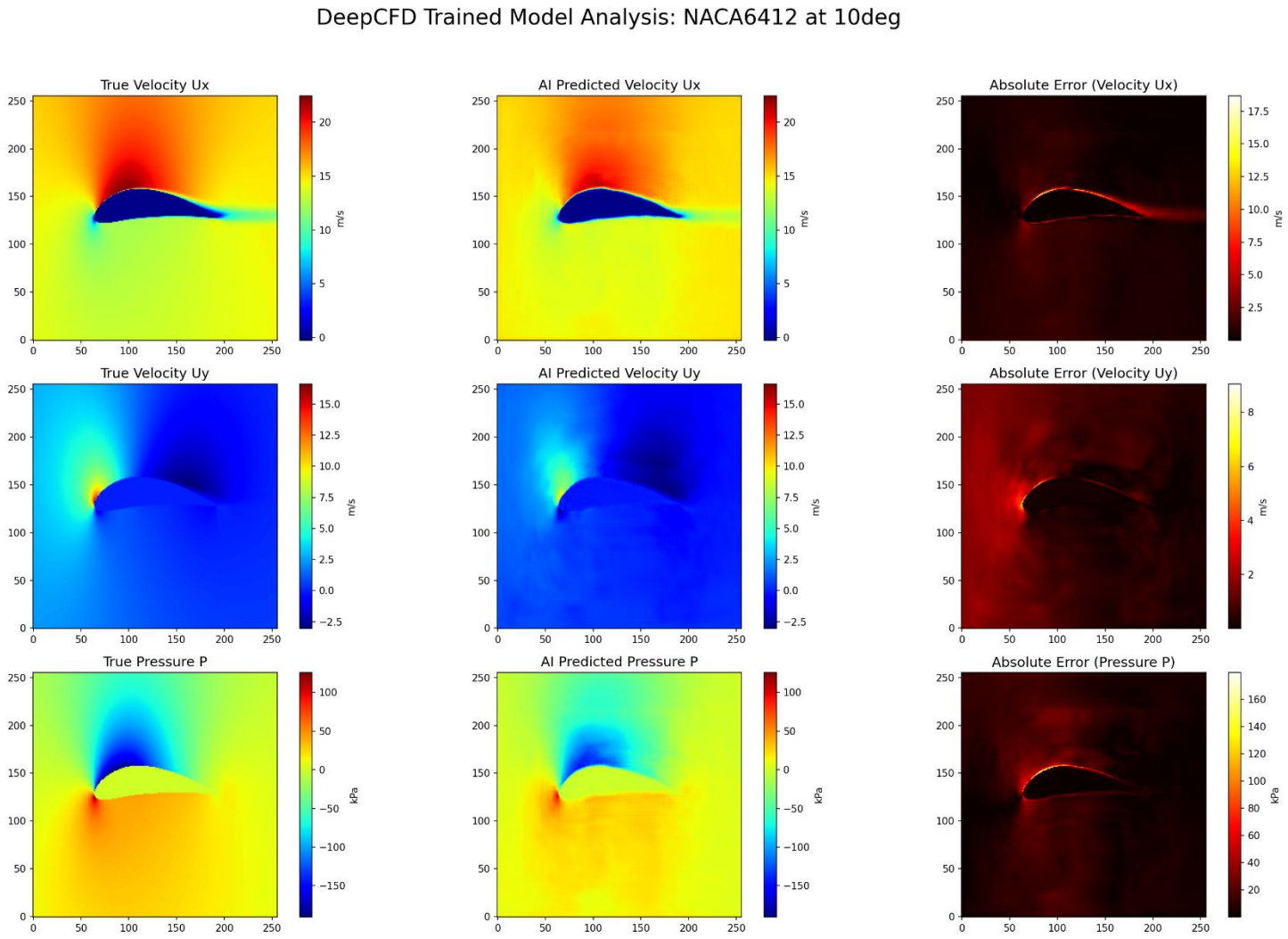


Figure 8.19 – Comparative fluid flow field analysis: NACA6412 at $\alpha = 10°$

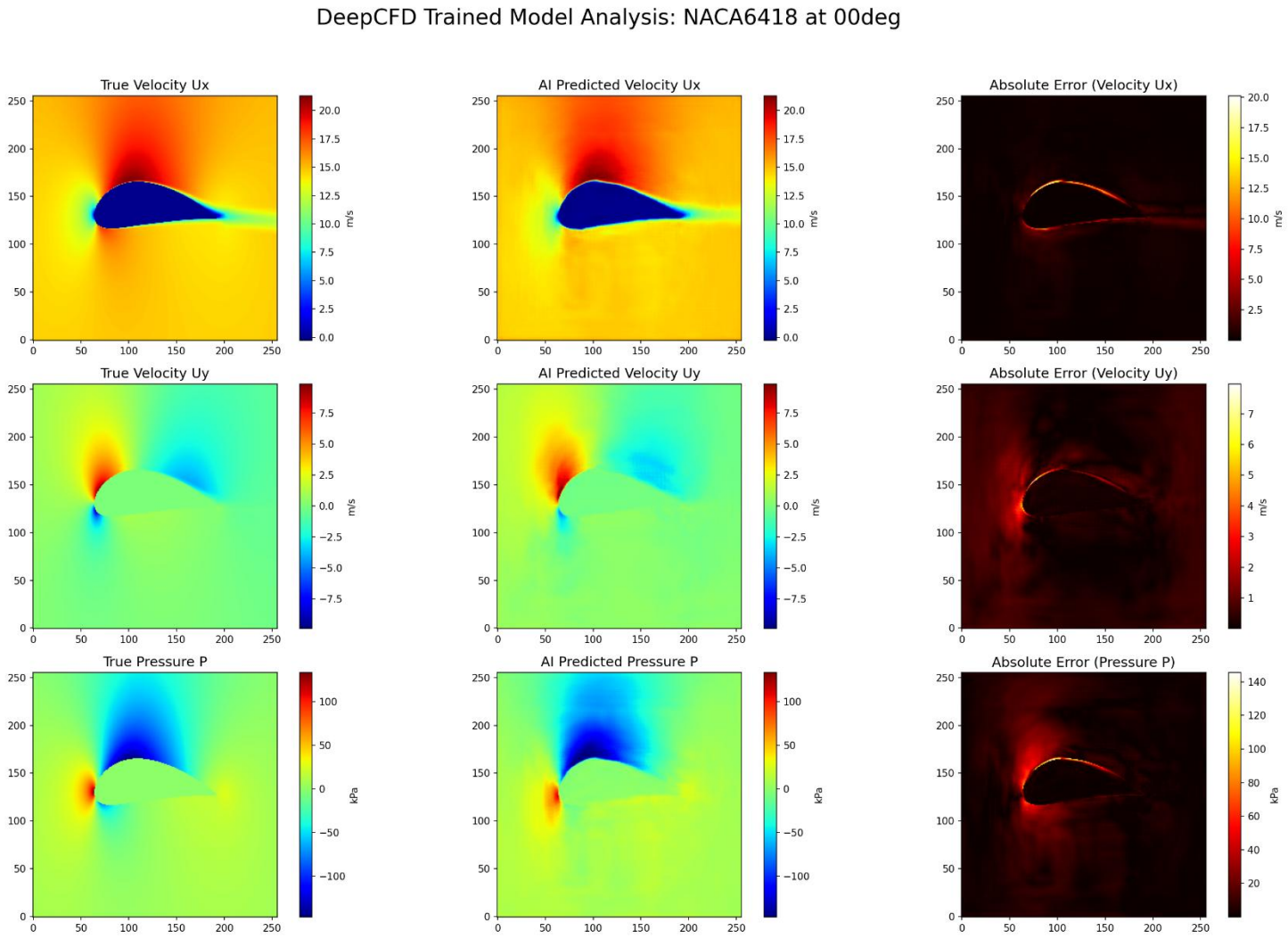


Figure 8.20 – Comparative fluid flow field analysis: NACA6418 at $\alpha = 0°$

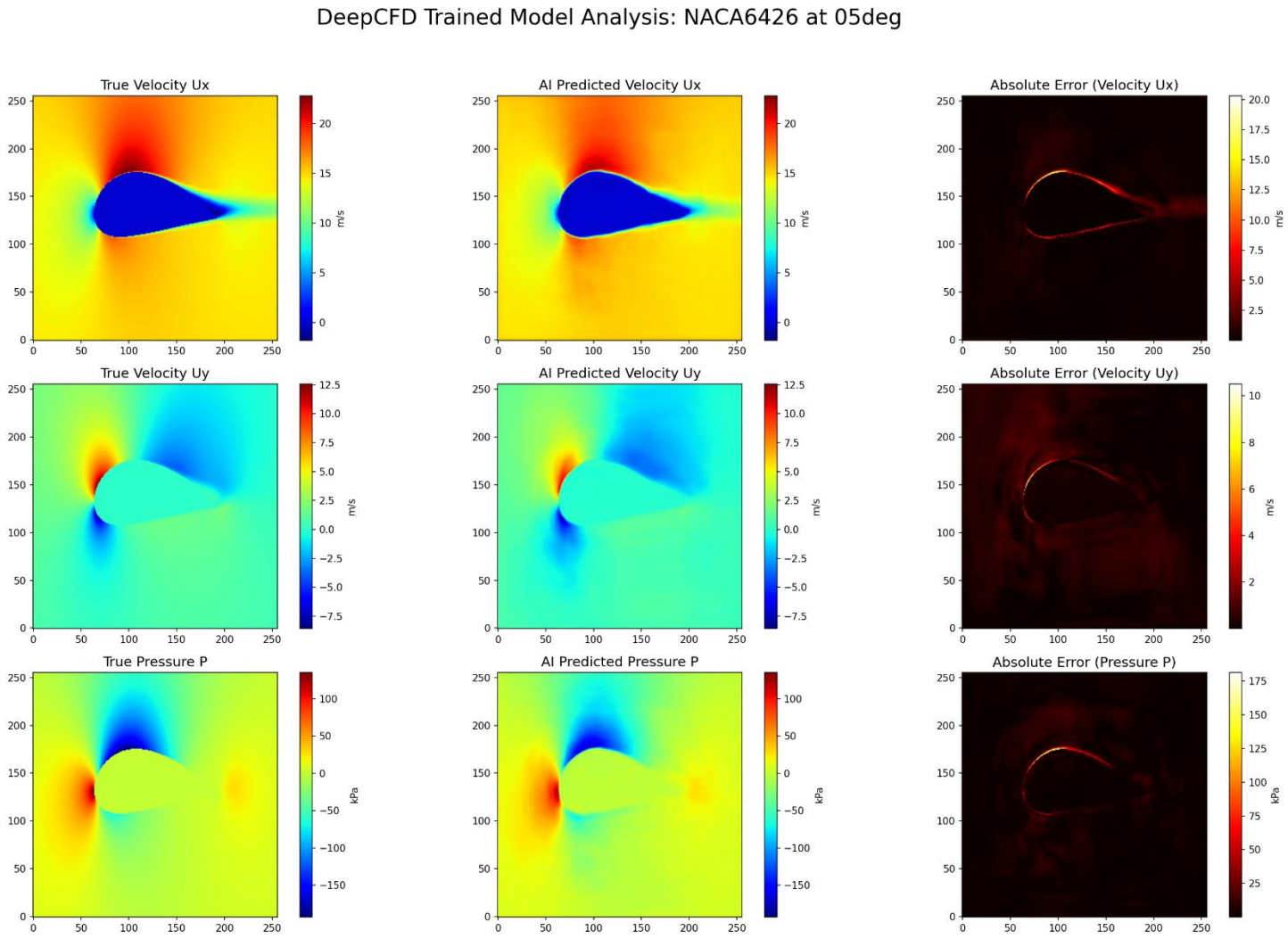


Figure 8.21 – Comparative fluid flow field analysis: NACA6426 at $\alpha = 5°$

Figures 8.10 – 8.21 show a small percentage of the validation set that proves the remarkable results that were achieved from training the fifth version. It is by far the best performing model that was trained. All of the 60 validation cases show similar results as the ones displayed in these images. Figures 8.10 – 8.21 were carefully selected to show the effectiveness of the model across a number of scenarios. These images show the four different categories: symmetrical, low camber, medium camber, and high camber airfoils. They also show three distinct thicknesses: 12, 18, and 26 percent. Additionally, three different αs were analyzed: 0°, 5°, and 10°. The 12 different images are a combination of these parameters to show in a small sample size the effectiveness of the AI model across a wide range of airfoil shapes, and α's.

Applying the hybrid normalization technique seems to have allowed the AI model to learn to predict gradients to a higher level of accuracy than the previous models. Additionally, because Global Normalization was used, the model was given a meaningful connection to help it establish connections between all of the airfoils. Whereas before the highs and minimums meant nothing

between different airfoils. Now, the model could determine whether a thicker airfoil at 5° α would have pressure values closer to the global maximum than a thinner airfoil at 5° α. It could determine if the velocity over an airfoil with higher camber would be higher than an airfoil with lower camber. One of the areas of importance in which previous models were falling to capture the physics was the smooth transition of the velocity and pressure gradients as the flow gets further from the airfoil. This version clearly shows understanding of the gradients and their behavior to a very high level. Increasing the Kernel Size allowed the model to be able to capture these transitions better and learn to predict smooth transitions rather than sharp changes. Taking a closer look at the images reveals just how much the AI model has mastered the prediction of fluid flow fields, but also its current shortcomings.

Figures 8.10 – 8.12 were specifically chosen to analyze the way the model predicts airflow for symmetrical airfoils with varying thicknesses at different α's. Looking at the U_x data it can be seen that the model performs really well for a medium thickness airfoil at 5° α. The model does perform less well with an airfoil of bigger thickness at higher α. However, the biggest deviation from the ground truth data can be seen with a symmetrical airfoil at 0° α. The model is still not quite capturing the symmetrical airflow between the top and bottom of the airfoil. This can again be attributed to the lack of data concerning symmetrical airfoils at 0° α. Out of the 500 airfoil cases only 25 actually deal with symmetrical airfoils at these conditions. Analyzing the wake reveals that the model has learned to predict this phenomenon quite well across most cases.

Analyzing U_y reveals a higher deviation from the ground truth data. However, most of the error seems to be concentrated specifically near the leading edge of the airfoil. Overall, all three cases show that the AI model has captured the physics really well, with a bit of refinement needed to polish out the remaining errors.

Previous models heavily struggled with the prediction of the static pressure field. Of course, part of that can be attributed to the boundary issue that was encountered with the early AI models. However, the previous version also struggled with static pressure. It struggled to predict low- and high-pressure fields, converging most accurately on cases of relative average α's. With the Global Normalization implemented, this AI model is able to accurately predict low-pressure fields and high-pressure fields.

Moving on to figures 8.13 – 8.15; these cases were meant to show the performance of the model against low camber airfoils with varying thicknesses and α's. Figure 8.13 was meant to represent low camber airfoils with thinner bodies, at relatively medium α's. Figure 8.14 was meant to represent low camber airfoils at relatively high α's. Figure 8.15 was meant to represent thick airfoils at low α's. The differences between the ground truth data and the AI predicted data were very small. All three channels show that the model is able to predict U_x, U_y, and p fields to a high level of accuracy. The model captures the maximums and minimums very well. Similar to the previous three cases, the AI model predicts the locations and smoothness of the gradients well, lacking only a bit more refinement.

Figures 8.16 – 8.18 were chosen to represent an average of the cases. Figure 8.16 shows a medium camber airfoil with a thin body at a medium α. Figure 8.17 shows a medium camber airfoil with medium thickness at a medium α. Figure 8.18 shows a medium camber airfoil of higher

thickness at a medium α. As expected, these cases were the ones that showed the least difference between the ground truth data and the AI predicted data. Initially, it was noticed in previous versions that the models seemed to be converging faster on cases that dealt with medium camber airfoils with medium thickness at α's: 2°, 5°, and 8°. These figures show near perfect predictions for U_x. Additionally, U_y and p show the lowest Absolute Error Maps out of all of the other 9 scenarios. The dataset that the model was given to train, had 125 cases in which the airfoil had medium camber. Additionally, it had 250 cases in which the airfoils had some kind of camber. The thicknesses that were used were from 6 to 30 percent. The average of these values was a thickness of 18 percent with 12 percent and 26 percent being right in between the minimum and maximum thicknesses that the model saw during training. Additionally, 5° α is the average α that was used with 2° and 8° being just a little more than halfway between the minimum and maximum α that the model saw during training. Therefore, the model saw more data that was closely related to these specific scenarios than for scenarios dealing closer to the minimums and maximums of the varying parameters. It's safe to assume that because the majority of the training data was close to these parameters, the AI model will perform better for these specific cases.

The last figures were meant to show the performance of the AI model against high camber airfoils of varying αs. Again, the model performed well in predicting the overall fluid flow fields. The only limitation is the lack of refinement within the velocity and static pressure gradients. The model is capable of accurately predicting the locations of the maximums and minimums but lacks the ability to formulate the small details to a high level of resolution.

Similar to version four, a training log was implemented that provided numerical data into how the model was performing Epoch by Epoch. After training this final version for 300 Epochs the Training Loss was 0.000527 and the validation loss was 0.000563. These numbers show 10 times better performance by this final version than by the previous version. In this case the Training Loss was lower than the Validation Loss, which is something that's more common within the world of AI. The difference between the two numbers suggests that the AI model is correctly converged to a model that is capable of predicting fluid flow fields of airfoils with great accuracy and to a high level of precision. One of the biggest worries is when Validation Loss is higher than Training Loss by a big gap. This usually means that overfitting is occurring in which the model is essentially memorizing the dataset. However, the generalization gap between these two values is 0.000036. This number suggests that the model is correctly learning the physics rather than memorizing the dataset.

8.4 Conclusion

The purpose of this document was to be able to train an AI model through data-driven methods, so that it could predict fluid flow fields to a high level of accuracy while reducing computational costs and time; burdens experienced with typical CFD software. After many iterations, version five proves that this is possible. The model was able to adequately predict the fluid flow field velocity components and static pressure for each of the 60 validation cases that were selected. These 60 cases were not seen by the model during training, only during validation to test out the model's performance. The minimal difference between the Training Loss and the Validation Loss confirms that the AI model was able to achieve optimal generalization, precisely reconstructing the flow field without overfitting. Additionally, looking at figures 8.10 – 8.21

reveals the high level of performance of the model across varying airfoil shapes and varying αs. The project ultimately achieved the core objective; however, many obstacles were encountered along the way that served as learning opportunities.

AI models typically work better with numbers between 0 and 1, initially a lot of error was encountered due to the wide ranges from real world values. To fix this, normalization was introduced into the script. Initially, local normalization was used for the SDF files and the Gridded files bringing all the values inside these files to a range between 0 and 1. Local normalization used the maximum and minimum values of each independent file. While this worked great for the SDF files, validation proved that the model was struggling to connect the dots between the velocity and static pressure along the entire dataset. The model understood relatively where the maximum and minimum values would go, but it failed to comprehend how these values faired when compared to the rest of the dataset. To fix this issue, global normalization was introduced into the Gridded files containing the ground truth CFD data. To achieve this the model looked at all of the values from the ground truth dataset and established the maximums and minimums for U_x, U_y, and p. Each of the 500 cases was then normalized using these values to see how the values changed depending on different cases. This change along with the increment of the Kernel Size to 5 led to the model being able to predict smoother and more accurate gradients.

The Kernel acts as the eye of the AI. It scans an area at a time capturing features that ultimately help the model create important connections about the behavior of the fluid flow field. By increasing the size of the area that the AI sees it is able to form better and smoother transitions that match the ground truth data. It allows the AI to see transitions that perhaps were missed by the smaller area. However, there is a limit to how big this area can be. A bigger area can also introduce too much data, causing the AI to become confused or formulate incorrect predictions leading to a divergence in its learning from the ground truth data. The AI can diverge from learning correctly through compounding errors introduced in many different forms. To mitigate this, different methods can be used.

In the fifth and final version two different methods were used. The first was the implementation of the Dropout method. What this does is that inside of the U-Net architecture a certain percentage of neurons are turned off in order to force the remaining neurons to form meaningful connections and learn from the data. At the time of training this makes it more difficult for the model. Version four showed that this method works, as the model had lower Validation Loss than it did Training Loss. This method is primarily used to prevent overfitting. Thus, it forces the model to truly learn the physics and achieve a higher level of convergence to the ground truth data. In addition to the Dropout Method, the AdamW optimizer was used. AdamW optimizer is an enhanced version of the Adam optimizer. AdamW optimizer decouples the Learning Rate and the Weight Decay into two independent variables. This prevents the Learning Rate from affecting the Weight Decay, effectively reducing unnecessary plateaus or unwanted divergence.

Additionally, one of the biggest issues in training AI models is that depending on the size of the data a large number of calculations are made during training. This leads to overuse of the CPU, which leads to high CPU temperatures that risk harming the hardware. To mitigate this, thermal management was introduced into the training. This was done as a way to reduce CPU temperatures and ensure that the training went along smoothly. Perhaps the biggest limiting factor at the moment

is hardware. Version five is a very effective model, yet one of the biggest regions where it struggles in every single case is the boundary layer. The boundary layer is so small yet contains vast amounts of important physics packed into a small finite area. Improvements can definitely be made to version five to achieve more precise models. However, most of these improvements require better equipment. Section 9 below talks about future work that can be done to improve the model and hopefully achieve a model that is superior to the current model.

9. Future Work

Chapter 8 above mentioned something that can be seen as contradictory. In section 8.3 the model is said to work well for certain cases better than others, and refinement is needed in the gradients to predict a smoother transition. Yet, later on in that same section it is said that the model has converged extremely close to the ground truth data. Both statements are correct. The model is performing extremely well. The remaining errors are small, yet in the world of engineering perfection is always sought after. Any little improvement can lead to major benefits for those that could potentially use the models in the future. While there is no more time to refine the current model, and hardware capabilities are limited, there is much that still can be done in the future to improve upon this work.

With better hardware a higher resolution grid could be used. Potentially, 512 x 512 or even a 1024 x 1024 grids. Higher resolution grids would give the model more pixels to look over resulting in more data to learn from. Additionally, in this case a bigger Kernel could be used to see even smoother transitions than what a 256 X 256 grid can provide. With a higher resolution grid, a deeper U-Net architecture could be used to extract more information for the model to learn from. Additionally, the authors of DeepCFD used more advanced techniques during the upsampling mapping phase. The authors used coordinate-based upscaling and exact pooling indices to create high spatial precision. The authors also used three separate decoders per each independent channel. For higher resolution grids this would be the method to implement as it ensures a high level of resolution in the upsampling mapping to produce the final images for the three channels. At one point the authors' hybrid loss function was used where MSE is used to calculate the error for the velocity and MAE is used to calculate the error for the static pressure. Ideally, these changes would result in higher resolution images, with the correct smoothness in gradient transition. In theory this would also help solve the boundary layer issue that the current model is experiencing. Having higher resolution images means having more pixels inside of the boundary layer, meaning there is more information that the AI model can learn from.

Additionally, due to the time constraint of this project many simplifications had to be made to the development of the dataset. In the future, the dataset could be expanded to include negative αs. For example, it is known from countless decades of research that cambered airfoils produce lift at negative αs. It would be extremely interesting to see how the AI model deals with these different scenarios. Additionally, the dataset could be expanded a bit to include airfoils that don't fall under the NACA 4-digit series. With these changes in mind, it would also be interesting to see how the model is able to predict stall characteristics at higher αs.

To achieve a more generalized model, different velocities could also be used. For this project an inlet velocity of 15 m/s was used. This number could be pushed to be higher and higher. Potentially even testing how the model does with supersonic cases.

It is known from current research that neural scaling laws exist which limit just how much a model can truly be perfected. Increasing the dataset increases the computing power required. Initially the error rate of AI models drops rather quickly and eventually levels off as if a plateau has been reached. Training a larger model drops this error rate to a lower number with a similar plateau effect. When bigger and bigger models are trained and compared using a logarithmic scale

a pattern emerges; an invisible boundary preventing the models from learning past it, a limitation. It would be interesting to do a project in which this phenomenon is explored. Where models are trained and tested to see which one is most optimal for accurate and precise results, while maintaining the lowest computing power possible.

References (AIAA)

[1] Ribeiro, M. D., Rehman, A., Ahmed, A., Dengel, A., "DeepCFD: Efficient Steady-State Laminar Flow Approximation with Deep Convolutional Neural Networks," *arXiv preprint*, arXiv:2004.08826, Apr. 2020. [Online]. Availabl. [Retrieved 1 Oct. 2025].

[2] "What Is Computational Fluid Dynamics?" ANSYS, n.d. [Online]. Available: https://www.ansys.com/simulation-topics/what-is-computational-fluid-dynamics . [Retrieved 24 Sept. 2025]

[3] "Robert FitzRoy and the Early Met Office," Met Office, n.d. [Online]. Available: https://www.metoffice.gov.uk/research/library-and-archive/archive-hidden-treasures/robert-fitzroy . [Retrieved 24 Sept. 2025].

[4] "The Birth of the Weather Forecast," BBC News, April. 2015. [Online]. Available: https://www.bbc.com/news/magazine-32483678 . [Retrieved 24 Sept. 2025].

[5] "N-S Equations — NASA Glenn Research Center," NASA Glenn Research Center, May. 2021. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/nseqs.html .[Retrieved 24 Sept. 2025].

[6] Nemati, M., Jahangirian, A.,"A Data-Driven Machine Learning Approach for Turbulent Flow Field Prediction Based on Direct Computational Fluid Dynamics Database," *Journal of Applied Fluid Mechanics (JAFM)*, Vol. 17, No. 1, pp. 60-74, 2023. Jan. 2024. [Online]. Available: https://www.jafmonline.net/article_2327.html . [Retrieved 30 Sept. 2025].

[7] Yu, X., Wang, C., and Liu, S., "Deep Learning for Fluid Velocity Field Estimation: A Review," *Ocean Engineering*, Vol. 281, 2023, Art. No. 115377. Mar. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S002980182300077X. [Retrieved 1 Oct. 2025].

[8] Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S., "Machine Learning–Accelerated Computational Fluid Dynamics," *Proceedings of the National Academy of Sciences*, Vol. 118, No. 21, 2021, Art. No. e2101784118. May. 2021. [Online]. Available: https://www.pnas.org/doi/10.1073/pnas.2101784118. [Retrieved 1 Oct. 2025].

[9] Wang, Y., Li, J., and Zhao, H., "Recent Advances on Machine Learning for Computational Fluid Dynamics: A Survey," *arXiv preprint*, arXiv:2408.12171, Aug. 2024. [Online]. Available: https://arxiv.org/abs/2408.12171. [Retrieved 1 Oct. 2025].

[10] Wang, F., and Wang, L., "Application of Artificial Intelligence in Computational Fluid Dynamics," *Industrial & Engineering Chemistry Research*, Vol. 60, No. 50, 2021, pp. 18552–18570. Feb. 2021. [Online]. Available: https://pubs.acs.org/doi/10.1021/acs.iecr.0c05045. [Retrieved 1 Oct. 2025].

[11] Calzolari, F., Mazzoleni, M., and Magrini, A., "Deep Learning to Replace, Improve, or Aid CFD Analysis in Built Environments," *Building and Environment*, Vol. 206, 2021, Art. No. 108348. Dec. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360132321007137. [Retrieved 1 Oct. 2025].

[12] Rana, A., and Gupta, P., "A Scalable Convolutional Neural Network Approach to Fluid Flow Prediction in Complex Environments," *Scientific Reports*, Vol. 14, 2024, Art. No. 45291. Oct. 2024. [Online]. Available: https://www.nature.com/articles/s41598-024-73529-y. [Retrieved 1 Oct. 2025].

[13] Kashefi, A., Rempe, D., and Guibas, L., "A Point-Cloud Deep Learning Framework for Prediction of Fluid Flow Fields on Irregular Geometries," *Physics of Fluids*, Vol. 33, No. 11, 2021, Art. No. 115101. Sep. 2021. [Online]. Available: https://arxiv.org/abs/2010.09469. [Retrieved 1 Oct. 2025].

[14] Panda, R., and Warrior, H., "Data-Driven Prediction of Complex Flow Field over an Axisymmetric Body of Revolution Using Machine Learning," *arXiv preprint*, arXiv:2111.07937, Nov. 2021. [Online]. Available: https://arxiv.org/abs/2111.07937. [Retrieved 1 Oct. 2025].

[15] Zuo, W., Sun, Y., and Lin, X., "Fast Simulation of Airfoil Flow Field via Deep Neural Network," *arXiv preprint*, arXiv:2312.04289, 2023. Dec. 2023 [Online]. Available: https://arxiv.org/abs/2312.04289. [Retrieved 1 Oct. 2025].

[16] Roznowicz, D., Lee, H., and Zhao, K., "Large-Scale Graph Machine Learning Surrogate Models for 3D Flow," *Advanced Modeling and Simulation in Engineering Sciences*, Vol. 11, 2024, Art. No. 259. Mar. 2024 [Online]. Available: https://amses-journal.springeropen.com/articles/10.1186/s40323-024-00259-1. [Retrieved 1 Oct. 2025].

[17] Raissi, M., Perdikaris, P., and Karniadakis, G. E., "Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear PDEs," *Journal of Computational Physics*, Vol. 378, 2019, pp. 686–707. Feb. 2019 [Online]. Available: https://doi.org/10.1016/j.jcp.2018.10.045 . [Retrieved 1 Oct. 2025].

[18] Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S., "Machine-Learning-Accelerated Computational Fluid Dynamics," *Proceedings of the National Academy of Sciences of the United States of America*, Vol. 118, No. 21, May 2021. https://doi.org/10.1073/pnas.2101784118 [Retrieved 29 Oct. 2025]

[19] Ribeiro, M. D., Rehman, A., Ahmed, S., and Dengel, A., "DeepCFD: Efficient Steady-State Laminar Flow Approximation with Deep Convolutional Neural Networks," *arXiv preprint* arXiv:2004.08826, Nov. 2021. [Online]. Available at: https://arxiv.org/abs/2004.08826 [Retrieved 31 Oct. 2025]

[20] He, K., Zhang, X., Ren, S., and Sun, J., *"Deep Residual Learning for Image Recognition,"* Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las

Vegas, NV, June 2016, pp. 770–778. https://doi.org/10.1109/CVPR.2016.90 [Retrieved 27 Nov. 2025]

[21] Airfoil Tools, *"CLARK Y Airfoil (clarky-il),"* AirfoilTools.com, http://airfoiltools.com/airfoil/details?airfoil=clarky-il, 2025 [Retrieved 11 Dec. 2025]

Appendix A

A.1 Airfoil Generator Code

```
import numpy as np
import os
import shutil

# Definition for the generation of the NACA 4 series airfoils
def generate_naca4(m, p, t, nodes=160):
    x = np.linspace(0, 1, nodes)
    if p == 0:
        yc = np.zeros_like(x)
        dyc_dx = np.zeros_like(x)
    else:
        yc = np.where(x < p,
                      m / p ** 2 * (2 * p * x - x ** 2),
                      m / (1 - p) ** 2 * ((1 - 2 * p) + 2 * p * x - x ** 2))
        dyc_dx = np.where(x < p,
                          2 * m / p ** 2 * (p - x),
                          2 * m / (1 - p) ** 2 * (p - x))
    yt = 5 * t * (0.2969 * np.sqrt(x) - 0.1260 * x - 0.3516 * x ** 2 + 0.2843
* x ** 3 - 0.1015 * x ** 4)
    theta = np.arctan(dyc_dx)
    xu, yu = x - yt * np.sin(theta), yc + yt * np.cos(theta)
    xl, yl = x + yt * np.sin(theta), yc - yt * np.cos(theta)
    # Combine upper and lower surfaces
    return np.concatenate([xu[::-1], xl[1:]]), np.concatenate([yu[::-1],
yl[1:]])

# Path to folder where the airfoils will be saved to
output_dir =
r'C:\Users\Wilbe\PycharmProjects\DeepCFD\Airfoil_Coordinates_100'

# Amking sure the file exists
if os.path.exists(output_dir):
    shutil.rmtree(output_dir)
os.makedirs(output_dir)

# Creating the list of 25 different thickness from 06 to 30 percent
# Creating teh 4 different categories
thicknesses = np.linspace(0.06, 0.30, 25)
categories = [
    {'m': 0.00, 'p': 0.0},  # Symmetrical
    {'m': 0.02, 'p': 0.4},  # Low Camber
    {'m': 0.04, 'p': 0.4},  # Mid Camber
    {'m': 0.06, 'p': 0.4}  # High Camber
]

# Loop that runs until the 100 airfoils have been created
count = 0
for cat in categories:
    for t in thicknesses:
        name = f"NACA{int(cat['m'] * 100)}{int(cat['p'] * 10)}{int(round(t *
100)):02d}"
```

```
        x_coords, y_coords = generate_naca4(cat['m'], cat['p'], t)

        file_path = os.path.join(output_dir, f"{name}.txt")
        with open(file_path, "w") as f:
            for i, (xi, yi) in enumerate(zip(x_coords, y_coords)):
                group_id = 1
                point_id = i + 1
                f.write(f"{group_id}\t{point_id}\t{xi:f}\t{yi:f}\t0.0\n")
        count += 1

print(f"Successfully generated {count} files in DesignModeler format!")
```

A.2 Initial Gridder Code

```
import pandas as pd
import numpy as np
import os
from scipy.interpolate import griddata

# Paths where the information is being pulled from, and where it is being
stored
base_path = r"C:\ANSYS_Working\100_Airfoil_Dataset"
output_dir =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
os.makedirs(output_dir, exist_ok=True)

# Defining the Standardized (256x256) grid and ensuring that the entire
airfoil is encompassed
# Ensures the grid begins at X = -0.5 (infront of the airfoil) and ends at
1.5 (behind the trailing edge capturing the wake)
# Ensures the grid starts at Y = -0.5 (capturing the bottome of the airfoil)
and ends at 0.5 (capturing the top of the airfoil)
# 256j creates 256 points in each direction respectively
grid_x, grid_y = np.mgrid[-0.5:1.5:256j, -0.5:0.5:256j]

# Defining the categories where the CSV Fluent data is stored
categories = ["Symmetrical", "Low Camber", "Medium Camber", "High Camber"]

print("Starting Space-Proof CSV Gridding...")
count = 0

# Navigates through the categories and searches for all the CSV Fluent files
for cat in categories:
    cat_path = os.path.join(base_path, cat)
    if not os.path.exists(cat_path): continue

    for airfoil_folder in os.listdir(cat_path):
        airfoil_path = os.path.join(cat_path, airfoil_folder)
        if not os.path.isdir(airfoil_path): continue

        csv_files = [f for f in os.listdir(airfoil_path) if
f.endswith('_FULL.csv')]

        for csv_name in csv_files:
            try:
```

```
                # Step 1 Loading the CSV Fluent data
                df = pd.read_csv(os.path.join(airfoil_path, csv_name))

                # Step 2 Deleting unnecessary spaces within the columns
                df.columns = df.columns.str.strip()

                # Step 3 Extracting the coordinates from the CSV Fluent file
                points_x = df['x-coordinate'].values
                points_y = df['y-coordinate'].values

                # Step 4 Creating the Grid, and filling in the corresponding
pressure and velocity component values
                # Having a linear method creates a smooth transition between
the points
                p_grid = griddata((points_x, points_y),
df['pressure'].values, (grid_x, grid_y), method='linear',
                                  fill_value=0)
                u_grid = griddata((points_x, points_y), df['x-
velocity'].values, (grid_x, grid_y), method='linear',
                                  fill_value=0)
                v_grid = griddata((points_x, points_y), df['y-
velocity'].values, (grid_x, grid_y), method='linear',
                                  fill_value=0)

                # Step 5 Stacking the three grids for: Ux, Uy, Pressure, and
saving them
                final_grid = np.stack([u_grid, v_grid, p_grid])
                save_name = csv_name.replace('_FULL.csv', '_gridded.npy')
                np.save(os.path.join(output_dir, save_name), final_grid)

                count += 1
                if count % 10 == 0:
                    print(f"Successfully gridded {count} files...")

            except Exception as e:
                print(f"Error on {csv_name}: {e}")

print(f"\nFINISH! Successfully processed {count} total simulations.")
print(f"Files saved in: {output_dir}")
```

A.3 Initial SDF Code

```
import numpy as np
import os
from scipy.spatial import distance

# Paths where the airfoil .txt files are and where the images containing the
SDF data will be saved
coords_folder =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\Airfoil_Coordinates_100"
output_dir =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
os.makedirs(output_dir, exist_ok=True)

# Ensuring the SDF generator is using the same exact grid where the pressure
```

```
and velocity values were saved
grid_x, grid_y = np.mgrid[-0.5:1.5:256j, -0.5:0.5:256j]
grid_points = np.vstack([grid_x.ravel(), grid_y.ravel()]).T

print("Starting SDF Generation from central coordinates folder...")
count = 0

# Loop that goes through every single .txt file inside of the
Airfoils_Coordinates_100 folder
txt_files = [f for f in os.listdir(coords_folder) if f.endswith('.txt')]

for txt_name in txt_files:
    txt_path = os.path.join(coords_folder, txt_name)
    airfoil_name = txt_name.replace('.txt', '')
    try:
        # Step 1 Load cloumns 3 and 4 which contain the x and y coordinates
for each point
        data = np.loadtxt(txt_path)
        geom_points = data[:, 2:4]

        # Step 2 calculating the SDF
        dists = distance.cdist(grid_points, geom_points).min(axis=1)
        sdf_image = dists.reshape(256, 256)

        # Step 3 creating 5 different images to be used for the 5 different
angles of attack
        # Saving the images by using the correct nomenclature
        for angle in [0, 2, 5, 8, 10]:
            save_name = f"{airfoil_name}_{angle:02d}deg_sdf.npy"
            np.save(os.path.join(output_dir, save_name), sdf_image)
            count += 1

        if count % 50 == 0:
            print(f"Processed {count} SDF images...")

    except Exception as e:
        print(f"Error processing {txt_name}: {e}")

print(f"\nSUCCESS! Created {count} SDF files.")
print(f"Final training inputs are ready in: {output_dir}")
```

A.4 Master Trainer Version 1 Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
import numpy as np
import os
import csv

# Defining the paths that were used to locate the SDF data, the Gridded data,
the Training Log, and the Saved Model
```

```
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
LOG_FILE = "NACA_100_Airfoil_Training_Log.csv"
MODEL_SAVE_PATH = "deepcfd_model.pth"

# Setup Parameters:
# Batch size - number of airfoils to look at before updating
# Epochs - how many times the AI will look through the dataset
# Learning rate - steps taken for correction
# Dropout rate - percent of neurons that are turned off during training
BATCH_SIZE = 4
EPOCHS = 50
LEARNING_RATE = 0.001
DROPOUT_RATE = 0.2

# Loading the Data
class AirfoilDataset(Dataset):

    # Step 1 Sorting the airfoil shapes
    def __init__(self, sdf_dir, flow_dir):
        self.sdf_files = sorted([f for f in os.listdir(sdf_dir) if
f.endswith('.npy')])
        self.sdf_dir = sdf_dir
        self.flow_dir = flow_dir

    # Step 2 Number of airfoils in the library
    def __len__(self):
        return len(self.sdf_files)

    # Step 3 Opening up the SDF files
    # opnening up the corresponding Gridded files
    # Converting NumPy into PyTorch Tensor, adds channels (Ux,Uy,P)
    def __getitem__(self, idx):
        sdf_name = self.sdf_files[idx]
        sdf = np.load(os.path.join(self.sdf_dir, sdf_name))

        flow_name = sdf_name.replace('_sdf.npy', '_gridded.npy')
        flow = np.load(os.path.join(self.flow_dir, flow_name))

        return torch.FloatTensor(sdf).unsqueeze(0), torch.FloatTensor(flow)


# The U-Net Architecture
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels, dropout_p):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c): # Convolutional Block
            return nn.Sequential(
                nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Dropout2d(p=dropout_p),  # Dropout
                nn.Conv2d(out_c, out_c, kernel_size=3, padding=1),
                nn.ReLU(inplace=True)
            )
```

```
        self.enc1 = conv_block(in_channels, 32) # Extraction Layer
        self.pool1 = nn.MaxPool2d(2) # Shrinking Layer
        self.enc2 = conv_block(32, 64)
        self.pool2 = nn.MaxPool2d(2)

        self.bottleneck = conv_block(64, 128) # Deepest part of the AI

        self.up2 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2) # Up-
Sampling Layer
        self.dec2 = conv_block(128, 64) # Decoder Layer
        self.up1 = nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)

        self.final = nn.Conv2d(32, out_channels, kernel_size=1) # Combining
the layers into 3 Output Channels

    # Logic flow, down-sampling, bottleneck, up-sampling, skip connections,
reconstruction
    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool1(e1))
        b = self.bottleneck(self.pool2(e2))

        d2 = self.up2(b)
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2)
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# Training Set-up, 80% of airfoils for training, 20% for comparison
dataset = AirfoilDataset(SDF_DIR, FLOW_DIR)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_ds, val_ds = random_split(dataset, [train_size, val_size])

# Feeding a batch of airfoils but shuffling them to ensure memorization
doesn't occur
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE)

# Setting up the model, and optimizer
model = UNet(in_channels=1, out_channels=3, dropout_p=DROPOUT_RATE)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

# Training Log is initialized
header = ['epoch', 'train_loss', 'train_mse_total', 'train_mse_ux',
'train_mse_uy', 'train_mse_p',
          'val_loss', 'val_mse_total', 'val_mse_ux', 'val_mse_uy',
'val_mse_p']

with open(LOG_FILE, 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(header)
```

```
print(f"Starting Training on CPU... (Epochs: {EPOCHS})")


# Training For Loop
for epoch in range(EPOCHS):
    model.train()
    train_metrics = np.zeros(5)

    for inputs, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            mse_ux = criterion(outputs[:, 0], targets[:, 0]).item()
            mse_uy = criterion(outputs[:, 1], targets[:, 1]).item()
            mse_p = criterion(outputs[:, 2], targets[:, 2]).item()
            train_metrics += [loss.item(), loss.item(), mse_ux, mse_uy,
mse_p]

    train_metrics /= len(train_loader)

    model.eval()
    val_metrics = np.zeros(5)
    with torch.no_grad():
        for inputs, targets in val_loader:
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            mse_ux = criterion(outputs[:, 0], targets[:, 0]).item()
            mse_uy = criterion(outputs[:, 1], targets[:, 1]).item()
            mse_p = criterion(outputs[:, 2], targets[:, 2]).item()
            val_metrics += [loss.item(), loss.item(), mse_ux, mse_uy, mse_p]

    val_metrics /= len(val_loader)

    # Logging the data to a CSV File
    row = [epoch + 1] + train_metrics.tolist() + val_metrics.tolist()
    with open(LOG_FILE, 'a', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(row)

    print(f"Epoch {epoch + 1}/{EPOCHS} | Train Loss: {train_metrics[0]:.6f} |
Val Loss: {val_metrics[0]:.6f}")

torch.save(model.state_dict(), MODEL_SAVE_PATH)
print("Training Complete! Model saved and log generated.")
```

A.5 Comparison Script for Version 1

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import os

# Defining the necessary paths
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
MODEL_PATH = "deepcfd_model.pth"
OUTPUT_FOLDER = "report_images"

# Creating the folder where the comparison images will be stored
if not os.path.exists(OUTPUT_FOLDER):
    os.makedirs(OUTPUT_FOLDER)

# Picking 5 test case airfoils to test against the model
test_cases = [
    "NACA0012_00deg",
    "NACA2412_05deg",
    "NACA4412_08deg",
    "NACA6412_02deg",
    "NACA0012_10deg"
]

# Model Class: Defining the U-Net Architectire
class UNet(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c):
            return torch.nn.Sequential(
                torch.nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True),
                torch.nn.Dropout2d(p=0.2),
                torch.nn.Conv2d(out_c, out_c, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True)
            )

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = torch.nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64);
        self.pool2 = torch.nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128)
        self.up2 = torch.nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec2 = conv_block(128, 64)
        self.up1 = torch.nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)
        self.final = torch.nn.Conv2d(32, out_channels, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x);
        e2 = self.enc2(self.pool1(e1));
```

```
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b);
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2);
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# Loading the Model
device = torch.device("cpu")
model = UNet(1, 3)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

# Loop that generates the comparison images
print("Generating report images...")
for airfoil in test_cases:
    print(f"Processing {airfoil}...")

    # Step 1 Loading the data
    sdf = np.load(os.path.join(SDF_DIR, f"{airfoil}_sdf.npy"))
    truth = np.load(os.path.join(FLOW_DIR, f"{airfoil}_gridded.npy"))

    # Step 2 Predicting the flow
    input_tensor = torch.FloatTensor(sdf).unsqueeze(0).unsqueeze(0)
    with torch.no_grad():
        prediction = model(input_tensor).squeeze(0).numpy()

    # Step 3 Plotting the results
    labels = ['Ux Velocity [m/s]', 'Uy Velocity [m/s]', 'Pressure [Pa]']
    fig, axes = plt.subplots(3, 3, figsize=(18, 12))

    for i in range(3):
        v_min, v_max = truth[i].min(), truth[i].max()

        # Loading up the ground truth data
        im1 = axes[i, 0].imshow(truth[i].T, origin='lower', cmap='jet',
vmin=v_min, vmax=v_max)
        axes[i, 0].set_title(f"Fluent: {labels[i]}")
        fig.colorbar(im1, ax=axes[i, 0]).set_label(labels[i])

        # Loading up the AI model predicted data
        im2 = axes[i, 1].imshow(prediction[i].T, origin='lower', cmap='jet',
vmin=v_min, vmax=v_max)
        axes[i, 1].set_title(f"DeepCFD: {labels[i]}")
        fig.colorbar(im2, ax=axes[i, 1]).set_label(labels[i])

        # Loading up the absolute error data
        error = np.abs(truth[i] - prediction[i])
        im3 = axes[i, 2].imshow(error.T, origin='lower', cmap='hot')
        axes[i, 2].set_title(f"Abs Error: {labels[i]}")
        fig.colorbar(im3, ax=axes[i, 2]).set_label('Magnitude')

    plt.suptitle(f"DeepCFD Analysis: {airfoil}", fontsize=20)
    plt.tight_layout(rect=[0.0, 0.03, 1.0, 0.95])

    # Saving to the folder that will contain the comparison images
```

```
    save_path = os.path.join(OUTPUT_FOLDER, f"Result_{airfoil}.png")
    plt.savefig(save_path, dpi=150)
    plt.close()  # Close to save memory

print(f"All images saved in the '{OUTPUT_FOLDER}' folder!")
```

A.6 Master Trainer Version Two Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
import numpy as np
import os
import csv
import time

# Defining the paths
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
LOG_FILE = "NACA_100_V2_Normalised_Log.csv"
MODEL_SAVE_PATH = "deepcfd_model_v2.pth"

# Set-up Parameters
BATCH_SIZE = 4
EPOCHS = 100  # Increase from 50 to 100
LEARNING_RATE = 0.0005  # Reducing it by half to lower overshooting and
potential divergence
DROPOUT_RATE = 0.2

class AirfoilDatasetV2(Dataset):
    def __init__(self, sdf_dir, flow_dir):
        self.sdf_files = sorted([f for f in os.listdir(sdf_dir) if
f.endswith('.npy')])
        self.sdf_dir = sdf_dir
        self.flow_dir = flow_dir

    def __len__(self): return len(self.sdf_files)

    def __getitem__(self, idx):
        sdf = np.load(os.path.join(self.sdf_dir, self.sdf_files[idx]))
        flow_name = self.sdf_files[idx].replace('_sdf.npy', '_gridded.npy')
        flow = np.load(os.path.join(self.flow_dir, flow_name))

        # Applying Min Max Normalization
        sdf_norm = (sdf - sdf.min()) / (sdf.max() - sdf.min() + 1e-8)

        # Normalizing Ux Uy and p independently
        flow_norm = np.zeros_like(flow)
        for i in range(3):
            f_min, f_max = flow[i].min(), flow[i].max()
            flow_norm[i] = (flow[i] - f_min) / (f_max - f_min + 1e-8)
```

```
        return torch.FloatTensor(sdf_norm).unsqueeze(0),
torch.FloatTensor(flow_norm)


# U-Net Archictecture
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels, dropout_p):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c):
            return nn.Sequential(
                nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Dropout2d(p=dropout_p),
                nn.Conv2d(out_c, out_c, kernel_size=3, padding=1),
                nn.ReLU(inplace=True)
            )

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64);
        self.pool2 = nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128)
        self.up2 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec2 = conv_block(128, 64)
        self.up1 = nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)
        self.final = nn.Conv2d(32, out_channels, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x);
        e2 = self.enc2(self.pool1(e1));
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b);
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2);
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# Training Set-up, 80% of airfoils for training, 20% for comparison
dataset = AirfoilDatasetV2(SDF_DIR, FLOW_DIR)
train_size = int(0.8 * len(dataset))
train_ds, val_ds = random_split(dataset, [train_size, len(dataset) -
train_size])

# Feeding a batch of airfoils but shuffling them to ensure memorization
doesn't occur
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE)

# Loading the model and optimizer
model = UNet(1, 3, DROPOUT_RATE)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

# Starting the training log
```

```
with open(LOG_FILE, 'w', newline='') as f:
    csv.writer(f).writerow(['epoch', 'train_loss', 'val_loss'])

print(f"Starting Training V2 (100 Epochs, Normalized)...")

# Training loop
for epoch in range(EPOCHS):
    model.train()
    t_loss = 0
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        loss = criterion(model(inputs), targets)
        loss.backward();
        optimizer.step()
        t_loss += loss.item()

    model.eval()
    v_loss = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            v_loss += criterion(model(inputs), targets).item()

    avg_t, avg_v = t_loss / len(train_loader), v_loss / len(val_loader)

# Logging the data
    with open(LOG_FILE, 'a', newline='') as f:
        csv.writer(f).writerow([epoch + 1, avg_t, avg_v])

    print(f"Epoch {epoch + 1}/100 | Train: {avg_t:.6f} | Val: {avg_v:.6f}")

    # Allowing small cooldown window after every 10 epochs
    if (epoch + 1) % 10 == 0: torch.save(model.state_dict(), MODEL_SAVE_PATH)
    time.sleep(10)

# Saving the model
torch.save(model.state_dict(), MODEL_SAVE_PATH)
print("V2 Training Complete!")
```

A.7 Comparison Script for Version Two

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import os

# Defining paths
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
MODEL_PATH = "deepcfd_model_v2.pth"
OUTPUT_FOLDER = "report_images_v2"

if not os.path.exists(OUTPUT_FOLDER):
    os.makedirs(OUTPUT_FOLDER)
```

```
# Test Cases
test_cases = [
    "NACA0012_00deg",
    "NACA2412_05deg",
    "NACA4412_08deg",
    "NACA6412_02deg",
    "NACA0012_10deg"
]


# Defining the U-Net Architecture
class UNet(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c):
            return torch.nn.Sequential(
                torch.nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True),
                torch.nn.Dropout2d(p=0.2),
                torch.nn.Conv2d(out_c, out_c, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True)
            )

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = torch.nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64);
        self.pool2 = torch.nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128)
        self.up2 = torch.nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec2 = conv_block(128, 64)
        self.up1 = torch.nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)
        self.final = torch.nn.Conv2d(32, out_channels, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x);
        e2 = self.enc2(self.pool1(e1));
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b);
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2);
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# Load Model
device = torch.device("cpu")
model = UNet(1, 3)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

print("Generating V2 report images...")
for airfoil in test_cases:
    print(f"Processing {airfoil}...")
```

```
    # Loading the SDF and Gridded data
    sdf_raw = np.load(os.path.join(SDF_DIR, f"{airfoil}_sdf.npy"))
    truth_raw = np.load(os.path.join(FLOW_DIR, f"{airfoil}_gridded.npy"))

    # Implementing normalization
    sdf_min, sdf_max = sdf_raw.min(), sdf_raw.max()
    sdf_norm = (sdf_raw - sdf_min) / (sdf_max - sdf_min + 1e-8)
    input_tensor = torch.FloatTensor(sdf_norm).unsqueeze(0).unsqueeze(0)

    # Predicting the flow using the trained model
    with torch.no_grad():
        pred_norm = model(input_tensor).squeeze(0).numpy()

    # Denormalizing so the data can be plotted
    pred_physics = np.zeros_like(pred_norm)
    for i in range(3):
        f_min, f_max = truth_raw[i].min(), truth_raw[i].max()
        pred_physics[i] = pred_norm[i] * (f_max - f_min + 1e-8) + f_min

    # Plotting the data
    labels = ['Ux [m/s]', 'Uy [m/s]', 'Pressure [Pa]']
    fig, axes = plt.subplots(3, 3, figsize=(18, 12))

    # Using for loop to plot
    for i in range(3):
        v_min, v_max = truth_raw[i].min(), truth_raw[i].max()

        # Loading up the ground truth data
        im1 = axes[i, 0].imshow(truth_raw[i].T, origin='lower', cmap='jet',
vmin=v_min, vmax=v_max)
        axes[i, 0].set_title(f"Fluent: {labels[i]}")
        fig.colorbar(im1, ax=axes[i, 0]).set_label(labels[i])

        # Loading up the predicted data
        im2 = axes[i, 1].imshow(pred_physics[i].T, origin='lower',
cmap='jet', vmin=v_min, vmax=v_max)
        axes[i, 1].set_title(f"DeepCFD V2: {labels[i]}")
        fig.colorbar(im2, ax=axes[i, 1]).set_label(labels[i])

        # loading up the absolute error between ground truth and predicted
data
        error = np.abs(truth_raw[i] - pred_physics[i])
        im3 = axes[i, 2].imshow(error.T, origin='lower', cmap='hot')
        axes[i, 2].set_title(f"Abs Error: {labels[i]}")
        fig.colorbar(im3, ax=axes[i, 2]).set_label('Error')

    plt.suptitle(f"DeepCFD V2 Analysis: {airfoil}", fontsize=20)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])

    # Saving the images to a folder for later use
    plt.savefig(os.path.join(OUTPUT_FOLDER, f"V2_{airfoil}.png"), dpi=150)
    plt.close()

print(f"All V2 images saved in '{OUTPUT_FOLDER}'!")
```

A.8 Master Trainer Version Three Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
import os
import csv
import time

# Defining the paths
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
LOG_FILE = "NACA_100_V3_AuthorWeighted_Log.csv"
PREVIOUS_MODEL = "deepcfd_model_v2.pth"
NEW_MODEL_SAVE = "deepcfd_model_v3.pth"

# Parameters for training
BATCH_SIZE = 4
REFINEMENT_EPOCHS = 100
LEARNING_RATE = 0.0001  # Fine-tuning speed
DROPOUT_RATE = 0.2

# Loading the dataset
class AirfoilDatasetV3(Dataset):
    def __init__(self, sdf_dir, flow_dir):
        self.sdf_files = sorted([f for f in os.listdir(sdf_dir) if
f.endswith('.npy')])
        self.sdf_dir = sdf_dir
        self.flow_dir = flow_dir

    def __len__(self): return len(self.sdf_files)

    def __getitem__(self, idx):
        sdf = np.load(os.path.join(self.sdf_dir, self.sdf_files[idx]))
        flow = np.load(os.path.join(self.flow_dir,
self.sdf_files[idx].replace('_sdf.npy', '_gridded.npy')))
        sdf_norm = (sdf - sdf.min()) / (sdf.max() - sdf.min() + 1e-8)
        flow_norm = np.zeros_like(flow)
        for i in range(3):
            f_min, f_max = flow[i].min(), flow[i].max()
            flow_norm[i] = (flow[i] - f_min) / (f_max - f_min + 1e-8)
        return torch.FloatTensor(sdf_norm).unsqueeze(0),
torch.FloatTensor(flow_norm)


# U-Net architecture
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels, dropout_p):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c):
            return nn.Sequential(
```

```
                nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Dropout2d(p=dropout_p),
                nn.Conv2d(out_c, out_c, kernel_size=3, padding=1),
                nn.ReLU(inplace=True)
            )

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64);
        self.pool2 = nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128)
        self.up2 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec2 = conv_block(128, 64)
        self.up1 = nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)
        self.final = nn.Conv2d(32, out_channels, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x);
        e2 = self.enc2(self.pool1(e1));
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b);
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2);
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# # Training Set-up, 80% of airfoils for training, 20% for comparison
device = torch.device("cpu")
dataset = AirfoilDatasetV3(SDF_DIR, FLOW_DIR)
train_size = int(0.8 * len(dataset))
train_ds, val_ds = torch.utils.data.random_split(dataset, [train_size,
len(dataset) - train_size],
generator=torch.Generator().manual_seed(42))

train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE)

# Loading the model
model = UNet(1, 3, DROPOUT_RATE)
model.load_state_dict(torch.load(PREVIOUS_MODEL, map_location=device))
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE,
weight_decay=0.005)

# Using DeepCFD auhtor's weighted loss
def author_loss_func(output, target):
    # MSE for Ux and Uy
    loss_u = (output[:, 0, :, :] - target[:, 0, :, :]) ** 2
    loss_v = (output[:, 1, :, :] - target[:, 1, :, :]) ** 2
    # MAE for p
    loss_p = torch.abs(output[:, 2, :, :] - target[:, 2, :, :])

    # Adding the weighted losses
    return torch.mean(loss_u + loss_v + loss_p)
```

```
# Loading the Logger
with open(LOG_FILE, 'w', newline='') as f:
    csv.writer(f).writerow(['epoch', 'train_loss', 'val_loss'])

print(f"Resuming Training V3 (Author-Weighted Refinement)...")

# Starting refinement training
for epoch in range(REFINEMENT_EPOCHS):
    model.train()
    t_loss = 0
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        output = model(inputs)
        loss = author_loss_func(output, targets)
        loss.backward();
        optimizer.step()
        t_loss += loss.item()

    model.eval()
    v_loss = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            v_loss += author_loss_func(model(inputs), targets).item()

    avg_t, avg_v = t_loss / len(train_loader), v_loss / len(val_loader)
    with open(LOG_FILE, 'a', newline='') as f:
        csv.writer(f).writerow([epoch + 101, avg_t, avg_v])

    print(f"Epoch {epoch + 101}/200 | Train: {avg_t:.6f} | Val: {avg_v:.6f}")
    if (epoch + 1) % 10 == 0: torch.save(model.state_dict(), NEW_MODEL_SAVE)
    time.sleep(10)

torch.save(model.state_dict(), NEW_MODEL_SAVE)
print("V3 Refinement Complete!")
```

A.9 Comparison Script for Version Three

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import os

# Defining the paths
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
MODEL_PATH = "deepcfd_model_v3.pth"
OUTPUT_FOLDER = "report_images_v3"

# Output folder is created where the images will be stored
if not os.path.exists(OUTPUT_FOLDER):
    os.makedirs(OUTPUT_FOLDER)
```

```
# Test Cases
test_cases = [
    "NACA0012_00deg",
    "NACA2412_05deg",
    "NACA4412_08deg",
    "NACA6412_02deg",
    "NACA0012_10deg"
]


# Defining the U-Net architecture
class UNet(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c):
            return torch.nn.Sequential(
                torch.nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True),
                torch.nn.Dropout2d(p=0.2),
                torch.nn.Conv2d(out_c, out_c, kernel_size=3, padding=1),
                torch.nn.ReLU(inplace=True)
            )

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = torch.nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64);
        self.pool2 = torch.nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128)
        self.up2 = torch.nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec2 = conv_block(128, 64)
        self.up1 = torch.nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)
        self.final = torch.nn.Conv2d(32, out_channels, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x);
        e2 = self.enc2(self.pool1(e1));
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b);
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2);
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# Loading the model
device = torch.device("cpu")
model = UNet(1, 3)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

print("Generating FINAL V3 report images...")
for airfoil in test_cases:
    print(f"Processing {airfoil}...")

    # Loading the ground truth data
```

```
    sdf_raw = np.load(os.path.join(SDF_DIR, f"{airfoil}_sdf.npy"))
    truth_raw = np.load(os.path.join(FLOW_DIR, f"{airfoil}_gridded.npy"))

    # Normalizing the sdf data
    sdf_min, sdf_max = sdf_raw.min(), sdf_raw.max()
    sdf_norm = (sdf_raw - sdf_min) / (sdf_max - sdf_min + 1e-8)
    input_tensor = torch.FloatTensor(sdf_norm).unsqueeze(0).unsqueeze(0)

    # Prediction is done by the AI model
    with torch.no_grad():
        pred_norm = model(input_tensor).squeeze(0).numpy()

    # Denormalizing the data using the ground truth min and max
    pred_physics = np.zeros_like(pred_norm)
    for i in range(3):
        f_min, f_max = truth_raw[i].min(), truth_raw[i].max()
        pred_physics[i] = pred_norm[i] * (f_max - f_min + 1e-8) + f_min

    # Plotting the data
    labels = ['Ux [m/s]', 'Uy [m/s]', 'Pressure [Pa]']
    fig, axes = plt.subplots(3, 3, figsize=(18, 12))

    # For loop that plots the comparison images and the absolute error
    for i in range(3):
        # Using ground truth data min and max to set the min and max
        v_min, v_max = truth_raw[i].min(), truth_raw[i].max()

        # CFD data
        im1 = axes[i, 0].imshow(truth_raw[i].T, origin='lower', cmap='jet',
vmin=v_min, vmax=v_max)
        axes[i, 0].set_title(f"Fluent: {labels[i]}")
        fig.colorbar(im1, ax=axes[i, 0]).set_label(labels[i])

        # AI model predicted
        im2 = axes[i, 1].imshow(pred_physics[i].T, origin='lower',
cmap='jet', vmin=v_min, vmax=v_max)
        axes[i, 1].set_title(f"DeepCFD V3: {labels[i]}")
        fig.colorbar(im2, ax=axes[i, 1]).set_label(labels[i])

        # Absolute Error
        error = np.abs(truth_raw[i] - pred_physics[i])
        im3 = axes[i, 2].imshow(error.T, origin='lower', cmap='hot')
        axes[i, 2].set_title(f"Abs Error: {labels[i]}")
        fig.colorbar(im3, ax=axes[i, 2]).set_label('Error Magnitude')

    plt.suptitle(f"Final V3 Performance: {airfoil}\n(Refined with Weighted
Loss & AdamW)", fontsize=20)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])

    # Saving images to output folder
    plt.savefig(os.path.join(OUTPUT_FOLDER, f"V3_Final_{airfoil}.png"),
dpi=150)
    plt.close()

print(f"Success! Your 5 final comparison images are in '{OUTPUT_FOLDER}'.")
```

A.10 Fixed SDF Code

```
import numpy as np
import os
from scipy.spatial import distance
import matplotlib.path as mpath
import matplotlib.pyplot as plt

# Defining the paths
COORDS_FOLDER =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\Airfoil_Coordinates_100"
OUTPUT_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
os.makedirs(OUTPUT_DIR, exist_ok=True)

# setting up the 256 x 256 grid
RES = 256
grid_x, grid_y = np.mgrid[-0.5:1.5:complex(RES), -0.5:0.5:complex(RES)]
grid_points = np.vstack([grid_x.ravel(), grid_y.ravel()]).T

# The Masked SDF Generator
def run_sdf_generation():
    txt_files = [f for f in os.listdir(COORDS_FOLDER) if f.endswith('.txt')]
    print(f"Found {len(txt_files)} airfoil coordinate files.")
    count = 0

    for txt_name in txt_files:
        txt_path = os.path.join(COORDS_FOLDER, txt_name)
        airfoil_name = txt_name.replace('.txt', '')

        try:
            # Loading the x and y coordinates from the .txt files
            data = np.loadtxt(txt_path)
            geom_points = data[:, 2:4]

            # Calculting the distances of the pixels from the airfoil
            dists = distance.cdist(grid_points, geom_points).min(axis=1)
            sdf_image = dists.reshape(RES, RES)

            # Creates airfoil boundary and checks to see if value is inside
the airfoil shape
            path = mpath.Path(geom_points)
            mask_inside = path.contains_points(grid_points).reshape(RES, RES)

            # Makes distances inside airfoil shape negative
            sdf_image[mask_inside] = -sdf_image[mask_inside]

            # creating copies for each angle of attack
            for angle in [0, 2, 5, 8, 10]:
                save_name = f"{airfoil_name}_{angle:02d}deg_sdf.npy"
                np.save(os.path.join(OUTPUT_DIR, save_name), sdf_image)
                count += 1

            if count % 50 == 0:
                print(f"Progress: Generated {count} Signed SDF files...")
```

```
        except Exception as e:
            print(f"Error processing {txt_name}: {e}")

    print(f"\nCOMPLETED! Created {count} SIGNED SDF files in {OUTPUT_DIR}")

# Displaying the SDF to make sure the script ran correctly
def verify_last_sdf(sdf_data, airfoil_name):
    plt.figure(figsize=(8, 6))
    plt.imshow(sdf_data.T, origin='lower', extent=[-0.5, 1.5, -0.5, 0.5],
cmap='RdBu')
    plt.colorbar(label="Distance (Negative = Inside)")
    plt.title(f"Verification: Signed SDF for {airfoil_name}")
    plt.contour(sdf_data.T, levels=[0], colors='black') # Draw the boundary
line
    plt.show()

# Running the code and ensuring only this file can run it
if __name__ == "__main__":
    run_sdf_generation()
```

A.11 Fixed Gridder Code

```
import pandas as pd
import numpy as np
import os
from scipy.interpolate import griddata
import matplotlib.pyplot as plt

# Define the paths
base_path = r"C:\ANSYS_Working\100_Airfoil_Dataset"
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
output_dir =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
os.makedirs(output_dir, exist_ok=True)

# Defining the 256 x 256 grid
grid_x, grid_y = np.mgrid[-0.5:1.5:256j, -0.5:0.5:256j]

# Defining the categories
categories = ["Symmetrical", "Low Camber", "Medium Camber", "High Camber"]

print("Starting MASKED CSV Gridding...")
count = 0

for cat in categories:
    cat_path = os.path.join(base_path, cat)
    if not os.path.exists(cat_path): continue

    for airfoil_folder in os.listdir(cat_path):
        airfoil_path = os.path.join(cat_path, airfoil_folder)
        if not os.path.isdir(airfoil_path): continue

        csv_files = [f for f in os.listdir(airfoil_path) if
f.endswith('_FULL.csv')]
```

```
        for csv_name in csv_files:
            try:
                # Loading the CSV file
                df = pd.read_csv(os.path.join(airfoil_path, csv_name))
                df.columns = df.columns.str.strip()

                # Extracting the coordinates
                points_x = df['x-coordinate'].values
                points_y = df['y-coordinate'].values

                # Gridding each chaneel Ux, Uy, and p through interpolation
                u_grid = griddata((points_x, points_y), df['x-
velocity'].values, (grid_x, grid_y), method='linear',
                                  fill_value=0)
                v_grid = griddata((points_x, points_y), df['y-
velocity'].values, (grid_x, grid_y), method='linear',
                                  fill_value=0)
                p_grid = griddata((points_x, points_y),
df['pressure'].values, (grid_x, grid_y), method='linear',
                                  fill_value=0)

                # Finding the corresponding CSV file with the SDF file
                sdf_name = csv_name.replace('_FULL.csv', '_sdf.npy')
                sdf_path = os.path.join(SDF_DIR, sdf_name)

                # Implementing the masked sfg fix
                # Interior is defined as all the values that are negative
                if os.path.exists(sdf_path):
                    sdf = np.load(sdf_path)
                    interior = sdf < 0

                    # Setting Ux, Uy, and p to be 0 inside of the airfoil
                    u_grid[interior] = 0
                    v_grid[interior] = 0
                    p_grid[interior] = 0
                else:
                    print(f"Warning: No SDF found for {csv_name}")

                final_grid = np.stack([u_grid, v_grid, p_grid])
                save_name = csv_name.replace('_FULL.csv', '_gridded.npy')
                np.save(os.path.join(output_dir, save_name), final_grid)

                count += 1
                if count % 20 == 0:
                    print(f"Successfully processed {count} MASKED files...")

            except Exception as e:
                print(f"Error on {csv_name}: {e}")

print(f"\nFINISH! Created {count} total MASKED simulations.")
```

A.12 Master Trainer Version Four Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, Subset
import numpy as np
import os
import pandas as pd
import time

# Defining the paths
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
LOG_DIR = r"C:\Users\Wilbe\PycharmProjects\DeepCFD\training_logs"

os.makedirs(LOG_DIR, exist_ok=True)


# Loading the dataset
class AirfoilDataset(Dataset):
    def __init__(self, sdf_dir, flow_dir):
        self.sdf_files = sorted([f for f in os.listdir(sdf_dir) if
f.endswith('.npy')])
        self.sdf_dir = sdf_dir
        self.flow_dir = flow_dir

    def __len__(self): return len(self.sdf_files)

    def __getitem__(self, idx):
        sdf = np.load(os.path.join(self.sdf_dir, self.sdf_files[idx]))
        flow = np.load(os.path.join(self.flow_dir,
self.sdf_files[idx].replace('_sdf.npy', '_gridded.npy')))

        # Local Normalization is implemented
        sdf_norm = (sdf - sdf.min()) / (sdf.max() - sdf.min() + 1e-8)
        flow_norm = np.zeros_like(flow)
        for i in range(3):
            flow_norm[i] = (flow[i] - flow[i].min()) / (flow[i].max() -
flow[i].min() + 1e-8)

        return torch.FloatTensor(sdf_norm).unsqueeze(0),
torch.FloatTensor(flow_norm)

# U-Net architecture is defined
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c, drop=False):
            layers = [nn.Conv2d(in_c, out_c, 3, padding=1), nn.ReLU(True),
nn.Conv2d(out_c, out_c, 3, padding=1),
                      nn.ReLU(True)]
            if drop: layers.insert(2, nn.Dropout2d(0.1))
```

```
            block = nn.Sequential(*layers)
            for m in block:
                if isinstance(m, nn.Conv2d):
nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            return block

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64, True);
        self.pool2 = nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128, True)
        self.up2 = nn.ConvTranspose2d(128, 64, 2, 2);
        self.dec2 = conv_block(128, 64, True)
        self.up1 = nn.ConvTranspose2d(64, 32, 2, 2);
        self.dec1 = conv_block(64, 32)
        self.final = nn.Conv2d(32, out_channels, 1)

    def forward(self, x):
        e1 = self.enc1(x);
        e2 = self.enc2(self.pool1(e1));
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b);
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2);
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# setting 440 airfoils for training and using 60 for comparing
def get_split_indices(full_dataset):
    all_files = full_dataset.sdf_files
    target_thicknesses = ["12", "18", "26"]
    val_indices = []

    for idx, filename in enumerate(all_files):
        airfoil_name = filename.split('_')[0]
        if any(airfoil_name.endswith(t) for t in target_thicknesses):
            val_indices.append(idx)

    train_indices = [i for i in range(len(full_dataset)) if i not in
val_indices]
    return train_indices, val_indices


# Executiung the training
if __name__ == "__main__":
    full_dataset = AirfoilDataset(SDF_DIR, FLOW_DIR)
    train_idx, val_idx = get_split_indices(full_dataset)

    train_loader = DataLoader(Subset(full_dataset, train_idx), batch_size=10,
shuffle=True)
    val_loader = DataLoader(Subset(full_dataset, val_idx), batch_size=10,
shuffle=False)

    # Setting up the model
    device = torch.device("cpu")
    model = UNet(1, 3).to(device)
```

```
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.MSELoss()
    chan_crit = nn.MSELoss()

    log_history = []

    print(f"Starting 300-Epoch Training...")
    print(f"Comparison set: {len(val_idx)} files (12%, 18%, 26%
thicknesses)")
    print(f"Training set: {len(train_idx)} files")

    for epoch in range(300):
        model.train()
        t_m = {"l": 0, "ux": 0, "uy": 0, "p": 0}
        for inputs, targets in train_loader:
            optimizer.zero_grad()
            out = model(inputs)
            loss = criterion(out, targets)
            loss.backward()
            optimizer.step()
            t_m["l"] += loss.item()
            t_m["ux"] += chan_crit(out[:, 0, :, :], targets[:, 0, :,
:]).item()
            t_m["uy"] += chan_crit(out[:, 1, :, :], targets[:, 1, :,
:]).item()
            t_m["p"] += chan_crit(out[:, 2, :, :], targets[:, 2, :,
:]).item()

        model.eval()
        v_m = {"l": 0, "ux": 0, "uy": 0, "p": 0}
        with torch.no_grad():
            for inputs, targets in val_loader:
                out = model(inputs)
                v_m["l"] += criterion(out, targets).item()
                v_m["ux"] += chan_crit(out[:, 0, :, :], targets[:, 0, :,
:]).item()
                v_m["uy"] += chan_crit(out[:, 1, :, :], targets[:, 1, :,
:]).item()
                v_m["p"] += chan_crit(out[:, 2, :, :], targets[:, 2, :,
:]).item()

        res = [epoch + 1, t_m["l"] / len(train_loader), t_m["ux"] /
len(train_loader), t_m["uy"] / len(train_loader),
               t_m["p"] / len(train_loader),
               v_m["l"] / len(val_loader), v_m["ux"] / len(val_loader),
v_m["uy"] / len(val_loader),
               v_m["p"] / len(val_loader)]
        log_history.append(res)

        # Thermal management every 5 epochs, rest for 60 seconds
        if (epoch + 1) % 5 == 0:
            print(f"Epoch {epoch + 1}/300 | Train Loss: {res[1]:.6f} | Val
Loss: {res[5]:.6f}")
            time.sleep(60)

        # Save every 10 epochs
        if (epoch + 1) % 10 == 0:
```

```
            cols = ['Epoch', 'T_Loss', 'T_Ux', 'T_Uy', 'T_P', 'V_Loss',
'V_Ux', 'V_Uy', 'V_P']
            pd.DataFrame(log_history,
columns=cols).to_csv(os.path.join(LOG_DIR, "300_epoch_log_v5.csv"),
index=False)

    torch.save(model.state_dict(), "deepcfd_300epochs_v5.pth")
    print("SUCCESS! Training complete.")
```

A.13 Comparison Script for Version Four

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import os

# Defining the paths
MODEL_PATH =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\deepcfd_300epochs_v5.pth"
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
IMAGE_DIR = r"C:\Users\Wilbe\PycharmProjects\DeepCFD\Comparison_Results_300"

os.makedirs(IMAGE_DIR, exist_ok=True)


# U-Net architecture
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c, drop=False):
            layers = [nn.Conv2d(in_c, out_c, 3, padding=1), nn.ReLU(True),
nn.Conv2d(out_c, out_c, 3, padding=1),
                      nn.ReLU(True)]
            if drop: layers.insert(2, nn.Dropout2d(0.1))
            return nn.Sequential(*layers)

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64, True);
        self.pool2 = nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128, True)
        self.up2 = nn.ConvTranspose2d(128, 64, 2, 2);
        self.dec2 = conv_block(128, 64, True)
        self.up1 = nn.ConvTranspose2d(64, 32, 2, 2);
        self.dec1 = conv_block(64, 32)
        self.final = nn.Conv2d(32, out_channels, 1)

    def forward(self, x):
        e1 = self.enc1(x);
        e2 = self.enc2(self.pool1(e1));
```

```
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b);
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2);
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# Loading up the model
device = torch.device("cpu")
model = UNet(1, 3).to(device)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

# Loading up the comparison scenarios
all_files = sorted([f for f in os.listdir(SDF_DIR) if f.endswith('.npy')])
target_thicknesses = ["12", "18", "26"]
val_files = [f for f in all_files if any(f.split('_')[0].endswith(t) for t in
target_thicknesses)]

print(f"Generating images for {len(val_files)} comparison scenarios...")

for filename in val_files:
    sdf = np.load(os.path.join(SDF_DIR, filename))
    true_flow = np.load(os.path.join(FLOW_DIR, filename.replace('_sdf.npy',
'_gridded.npy')))

    # Normalizing
    sdf_norm = (sdf - sdf.min()) / (sdf.max() - sdf.min() + 1e-8)
    sdf_tensor = torch.FloatTensor(sdf_norm).unsqueeze(0).unsqueeze(0)

    with torch.no_grad():
        prediction = model(sdf_tensor).squeeze(0).numpy()

    # Creating the plots
    fig, axes = plt.subplots(3, 3, figsize=(18, 12))
    airfoil_name = filename.split('_')[0]
    angle_val = filename.split('_')[1]
    fig.suptitle(f"DeepCFD V4 Analysis: {airfoil_name} at {angle_val}",
fontsize=20)

    labels = ['Ux (Velocity)', 'Uy (Velocity)', 'P (Pressure)']
    units = ['m/s', 'm/s', 'kPa']

    for i in range(3):
        # Denormalizing the predicted data
        f_min, f_max = true_flow[i].min(), true_flow[i].max()
        pred_actual = (prediction[i] * (f_max - f_min)) + f_min

        unit = units[i]

        # True CFD data
        im1 = axes[i, 0].imshow(true_flow[i].T, origin='lower', cmap='jet')
        axes[i, 0].set_title(f"True {labels[i]}")
        cbar1 = fig.colorbar(im1, ax=axes[i, 0])
        cbar1.set_label(unit)
```

```
        # AI Predicted data, denormalized
        im2 = axes[i, 1].imshow(pred_actual.T, origin='lower', cmap='jet')
        axes[i, 1].set_title(f"AI Prediction")
        cbar2 = fig.colorbar(im2, ax=axes[i, 1])
        cbar2.set_label(unit)

        # Absolute error
        error = np.abs(true_flow[i] - pred_actual)
        im3 = axes[i, 2].imshow(error.T, origin='lower', cmap='hot')
        axes[i, 2].set_title(f"Error Map")
        cbar3 = fig.colorbar(im3, ax=axes[i, 2])
        cbar3.set_label(unit)

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    save_path = os.path.join(IMAGE_DIR,
f"Result_{airfoil_name}_{angle_val}.png")
    plt.savefig(save_path, dpi=150)
    plt.close(fig)

print(f"Done! 60 images saved to: {IMAGE_DIR}")
```

A.14 Global Normalization Parameters

```
import numpy as np
import os

# File path for the gridded 500 files
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
files = [f for f in os.listdir(FLOW_DIR) if f.endswith('.npy')]

# Where the data will be stored
all_ux, all_uy, all_p = [], [], []

# Scanning through all 500 files finding the max and min values for Ux, Uy,
and p
print("Scouting global physical ranges...")
for f in files:
    data = np.load(os.path.join(FLOW_DIR, f))
    all_ux.append([data[0].min(), data[0].max()])
    all_uy.append([data[1].min(), data[1].max()])
    all_p.append([data[2].min(), data[2].max()])

# Saving the global values
ux_min, ux_max = np.min(all_ux), np.max(all_ux)
uy_min, uy_max = np.min(all_uy), np.max(all_uy)
p_min, p_max = np.min(all_p), np.max(all_p)

# Printing the values to use in training code
print(f"GLOBAL UX: {ux_min} to {ux_max}")
print(f"GLOBAL UY: {uy_min} to {uy_max}")
print(f"GLOBAL P:  {p_min} to {p_max}")
```

A.15 Master Trainer Version Five Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, Subset
import numpy as np
import os
import pandas as pd
import time

# Defining the paths
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
LOG_DIR = r"C:\Users\Wilbe\PycharmProjects\DeepCFD\training_logs"
os.makedirs(LOG_DIR, exist_ok=True)

# Global min max parameters for Ux, Uy, and p
G_STATS = {
    'ux': (-3.959923, 23.546990),
    'uy': (-11.595594, 18.777665),
    'p': (-297.672956, 137.619694)
}


# Loading up the dataset
class AirfoilDataset(Dataset):
    def __init__(self, sdf_dir, flow_dir):
        self.sdf_files = sorted([f for f in os.listdir(sdf_dir) if
f.endswith('.npy')])
        self.sdf_dir = sdf_dir
        self.flow_dir = flow_dir

    def __len__(self): return len(self.sdf_files)

    def __getitem__(self, idx):
        sdf = np.load(os.path.join(self.sdf_dir, self.sdf_files[idx]))
        flow = np.load(os.path.join(self.flow_dir,
self.sdf_files[idx].replace('_sdf.npy', '_gridded.npy')))

        # Local SDF Normalization
        sdf_norm = (sdf - sdf.min()) / (sdf.max() - sdf.min() + 1e-8)

        # Global Normalization
        flow_norm = np.zeros_like(flow)
        flow_norm[0] = (flow[0] - G_STATS['ux'][0]) / (G_STATS['ux'][1] -
G_STATS['ux'][0])
        flow_norm[1] = (flow[1] - G_STATS['uy'][0]) / (G_STATS['uy'][1] -
G_STATS['uy'][0])
        flow_norm[2] = (flow[2] - G_STATS['p'][0]) / (G_STATS['p'][1] -
G_STATS['p'][0])

        return torch.FloatTensor(sdf_norm).unsqueeze(0),
torch.FloatTensor(flow_norm)
```

```
# U-Net architecture
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()
        # Defining the convolutional block
        # Kernel size was set to 5
        def conv_block(in_c, out_c, drop=False):
            layers = [
                nn.Conv2d(in_c, out_c, kernel_size=5, padding=2),
                nn.ReLU(True),
                nn.Conv2d(out_c, out_c, kernel_size=5, padding=2),
                nn.ReLU(True)
            ]
            if drop: layers.insert(2, nn.Dropout2d(0.1))
            block = nn.Sequential(*layers)
            for m in block:
                if isinstance(m, nn.Conv2d):
                    nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
            return block

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64, True);
        self.pool2 = nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128, True)
        self.up2 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec2 = conv_block(128, 64, True)
        self.up1 = nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)
        self.final = nn.Conv2d(32, out_channels, kernel_size=1)

    # Defining the flow of the U_Net architecture
    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool1(e1))
        b = self.bottleneck(self.pool2(e2))

        d2 = self.up2(b)
        d2 = self.dec2(torch.cat([d2, e2], dim=1))

        d1 = self.up1(d2)
        d1 = self.dec1(torch.cat([d1, e1], dim=1))

        return self.final(d1)


# Executing the Training
if __name__ == "__main__":
    full_dataset = AirfoilDataset(SDF_DIR, FLOW_DIR)

    # Selecting the 60 airfoils (12, 18, 26 thickness) for comparison
    target_t = ["12", "18", "26"]
    val_idx = [i for i, f in enumerate(full_dataset.sdf_files) if
any(f.split('_')[0].endswith(t) for t in target_t)]
    train_idx = [i for i in range(len(full_dataset)) if i not in val_idx]
```

```
    # Loading up the training and validation set
    train_loader = DataLoader(Subset(full_dataset, train_idx), batch_size=10,
shuffle=True)
    val_loader = DataLoader(Subset(full_dataset, val_idx), batch_size=10,
shuffle=False)
    # Loading up the empty U-Net architecture
    device = torch.device("cpu")
    model = UNet(1, 3).to(device)

    # Setting up AdamW optimizer with weight decay using author weights
    optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.005)
    criterion = nn.MSELoss()

    log_history = []

    print(f"Starting Final Training Run (300 Epochs)...")
    print(f"Config: Global Normalization | 5x5 Kernels | AdamW")

    # Running the training for loop
    for epoch in range(300):
        model.train()
        train_l = 0
        for inputs, targets in train_loader:
            optimizer.zero_grad()
            out = model(inputs)
            loss = criterion(out, targets)
            loss.backward()
            optimizer.step()
            train_l += loss.item()

        model.eval()
        val_l = 0
        with torch.no_grad():
            for inputs, targets in val_loader:
                out = model(inputs)
                val_l += criterion(out, targets).item()


        res = [epoch + 1, train_l / len(train_loader), val_l /
len(val_loader)]
        log_history.append(res)

        # Setting thermal regulation break
        if (epoch + 1) % 5 == 0:
            print(f"Epoch {epoch + 1}/300 | Train Loss: {res[1]:.6f} | Val
Loss: {res[2]:.6f}")
            time.sleep(60)

        # Saving the training log every 10 epochs
        if (epoch + 1) % 10 == 0:
            cols = ['Epoch', 'Train_Loss', 'Val_Loss']
            pd.DataFrame(log_history,
columns=cols).to_csv(os.path.join(LOG_DIR, "global_norm_5x5_log.csv"),
                                                    index=False)

    # Saving the model
```

```
    torch.save(model.state_dict(), "deepcfd_global_5x5_v6.pth")
    print("SUCCESS! Final professional-grade model saved.")
```

A.16 Comparison Script for Version Five

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import os

# Defining the paths
MODEL_PATH =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\deepcfd_global_5x5_v6.pth"
SDF_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\sdf_256"
FLOW_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\data\naca_100_dataset\gridded_256"
IMAGE_DIR =
r"C:\Users\Wilbe\PycharmProjects\DeepCFD\Final_Pro_Results_Labeled"

os.makedirs(IMAGE_DIR, exist_ok=True)

# Globalization Parameters
G_STATS = {
    'ux': (-3.959923, 23.546990),
    'uy': (-11.595594, 18.777665),
    'p': (-297.672956, 137.619694)
}


# Defining the training U-Net architecture: Same as training
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def conv_block(in_c, out_c, drop=False):
            layers = [
                nn.Conv2d(in_c, out_c, kernel_size=5, padding=2),
                nn.ReLU(True),
                nn.Conv2d(out_c, out_c, kernel_size=5, padding=2),
                nn.ReLU(True)
            ]
            if drop: layers.insert(2, nn.Dropout2d(0.1))
            return nn.Sequential(*layers)

        self.enc1 = conv_block(in_channels, 32);
        self.pool1 = nn.MaxPool2d(2)
        self.enc2 = conv_block(32, 64, True);
        self.pool2 = nn.MaxPool2d(2)
        self.bottleneck = conv_block(64, 128, True)
        self.up2 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec2 = conv_block(128, 64, True)
        self.up1 = nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)
        self.dec1 = conv_block(64, 32)
```

```
        self.final = nn.Conv2d(32, out_channels, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool1(e1))
        b = self.bottleneck(self.pool2(e2))
        d2 = self.up2(b)
        d2 = self.dec2(torch.cat([d2, e2], dim=1))
        d1 = self.up1(d2)
        d1 = self.dec1(torch.cat([d1, e1], dim=1))
        return self.final(d1)


# Loading up the trained model
device = torch.device("cpu")
model = UNet(1, 3).to(device)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

# Loading up the validation files
all_files = sorted([f for f in os.listdir(SDF_DIR) if f.endswith('.npy')])
target_t = ["12", "18", "26"]
val_files = [f for f in all_files if any(f.split('_')[0].endswith(t) for t in
target_t)]

print(f"Generating labeled results for {len(val_files)} cases...")

# Loading up the SDF files and the true flow
for filename in val_files:
    sdf = np.load(os.path.join(SDF_DIR, filename))
    true_flow = np.load(os.path.join(FLOW_DIR, filename.replace('_sdf.npy',
'_gridded.npy')))

    # Normalizing SDF
    sdf_norm = (sdf - sdf.min()) / (sdf.max() - sdf.min() + 1e-8)
    sdf_tensor = torch.FloatTensor(sdf_norm).unsqueeze(0).unsqueeze(0)

    with torch.no_grad():
        prediction = model(sdf_tensor).squeeze(0).numpy()

    fig, axes = plt.subplots(3, 3, figsize=(20, 14))
    airfoil_name = filename.split('_')[0]
    angle_val = filename.split('_')[1]
    fig.suptitle(f"DeepCFD Trained Model Analysis: {airfoil_name} at
{angle_val}", fontsize=22)

    keys = ['ux', 'uy', 'p']
    labels = ['Velocity Ux', 'Velocity Uy', 'Pressure P']
    units = ['m/s', 'm/s', 'kPa']

    for i, k in enumerate(keys):
        # Denormaliing prediction using Global Parameters
        pred_actual = (prediction[i] * (G_STATS[k][1] - G_STATS[k][0])) +
G_STATS[k][0]
        unit = units[i]
        v_min, v_max = true_flow[i].min(), true_flow[i].max()
```

```
        # Ground Truth Ansys Fluent CFD Data
        im1 = axes[i, 0].imshow(true_flow[i].T, origin='lower', cmap='jet',
vmin=v_min, vmax=v_max)
        axes[i, 0].set_title(f"True {labels[i]}", fontsize=14)
        cbar1 = fig.colorbar(im1, ax=axes[i, 0])
        cbar1.set_label(unit)

        # AI Model Predicted Data
        im2 = axes[i, 1].imshow(pred_actual.T, origin='lower', cmap='jet',
vmin=v_min, vmax=v_max)
        axes[i, 1].set_title(f"AI Predicted {labels[i]}", fontsize=14)
        cbar2 = fig.colorbar(im2, ax=axes[i, 1])
        cbar2.set_label(unit)

        # Absolute Error
        error = np.abs(true_flow[i] - pred_actual)
        im3 = axes[i, 2].imshow(error.T, origin='lower', cmap='hot')
        axes[i, 2].set_title(f"Absolute Error ({labels[i]})", fontsize=14)
        cbar3 = fig.colorbar(im3, ax=axes[i, 2])
        cbar3.set_label(unit)
    # Overall plot layout for the nine grids
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    save_name = f"Result_{airfoil_name}_{angle_val}.png"
    plt.savefig(os.path.join(IMAGE_DIR, save_name), dpi=150)
    plt.close()

print(f"SUCCESS! All labeled images saved to: {IMAGE_DIR}")
```