DEEP LEARNING BASED SPEED UP OF FLUID DYNAMICS SOLVERS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

DENIZ ALPER ACAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
AEROSPACE ENGINEERING

SEPTEMBER 2022

Approval of the thesis:

DEEP LEARNING BASED SPEED UP OF FLUID DYNAMICS SOLVERS

submitted by **DENIZ ALPER ACAR** in partial fulfillment of the requirements for the degree of **Master of Science in Aerospace Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil KALIPÇILAR
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Serkan ÖZGEN
Head of Department, **Aerospace Engineering** _____

Prof. Dr. Oğuz UZOL
Supervisor, **Aerospace Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Serkan Özgen
Aerospace Engineering, Middle East Technical University _____

Prof. Dr. Oğuz Uzol
Aerospace Engineering, Middle East Technical University _____

Assoc. Prof. Dr. Hande Alemdar
Computer Engineering, Middle East Technical University _____

Assist. Prof. Dr. Özgür Uğraş Baran
Mechanical Engineering, Middle East Technical University _____

Assoc. Prof. Dr. Onur Baş
Mechanical Engineering, TED University _____

Date: 08.09.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Deniz Alper Acar

Signature :

ABSTRACT

DEEP LEARNING BASED SPEED UP OF FLUID DYNAMICS SOLVERS

Acar, Deniz Alper
M.S., Department of Aerospace Engineering
Supervisor: Prof. Dr. Oğuz UZOL

September 2022, 112 pages

In this thesis, two distinct deep learning-based methods for the speed-up of fluid dynamics solvers are proposed. The first method called Parametric Encoded Physics informed neural network (PEPINN), is utilized to solve transient fluid dynamics problems. PEPINN is an alternative to the Physics informed neural networks (PINN) and is based on the parametric encoding of the problem domain. In PEPINN the automatic differentiation for calculation of the problem residual is replaced with finite difference kernels which improve PEPINN's time and memory complexity. This model can achieve up to $40\times$ speed up in wall time for the solution of the Taylor-Green Vortex problem compared to the best alternative vanilla PINN model with no loss in solutions mean squared error. It is also shown that PEPINN can be trained on up to $183\times$ larger data compared to the alternative vanilla PINN methods in a GTX 1080 Ti GPU. The second proposed method in this thesis is based on the hypothesis that providing the predicted solution of the steady-state Navier-Stokes equations as their initial condition might speed up the solution process. In this method, an Unet-based architecture is trained on a discretized representation of the whole problem domain given its initial and boundary conditions. The trained model is used to predict the converged solution of similar cases and the obtained results are transferred to the computational mesh

of that problem. This method is tested on the steady, incompressible, subsonic flow around arbitrary airfoils.

Keywords: Partial Differential Equations, Physics informed neural networks, Deep learning, Computational Fluid Dynamics, Parametric Encoding

ÖZ

AKIŞKANLAR DİNAMİĞİ ÇÖZÜCÜLERİNİN DERİN ÖĞRENMEYE DAYALI HIZLANDIRILMASI

Acar, Deniz Alper
Yüksek Lisans, Havacılık ve Uzay Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Oğuz UZOL

Eylül 2022 , 112 sayfa

Bu tezde, akışkanlar dinamiği çözücülerinin hızlandırılması için iki farklı derin öğrenme tabanlı yöntem önerilmektedir. Parametrik Kodlanmış Fizik Bilgili Sinir Ağı (PEPINN) olarak adlandırılan ilk yöntem, geçici akışkan dinamiği problemlerini çözmek için kullanılmaktadır. Fizik bilgili sinir ağları (PINN) için bir alternatif olan Parametrik Kodlanmış Fizik Bilgili Sinir Ağı (PEPINN), problem alanının parametrik olarak kodlanmasına dayanır. PEPINN yönteminde, problem kalıntısının (residual) hesaplanması için otomatik türevlendirme, sonlu fark çekirdekleri (kernels) ile değiştirilir, bu da PEPINN'in zaman ve bellek karmaşıklığını düşürür.

Bu model, Taylor-Green Vortex probleminin çözümü için en iyi alternatif olan orijinal fizik bilgili sinir ağları (PINN) modeline kıyasla, gerçek zamanda (wall time) 40 kata kadar hızlanma sağlayabilir ve çözümlerin ortalama karesel hatasında herhangi bir kayıp olmaz. Ayrıca Parametrik Kodlanmış Fizik Bilgili Sinir Ağı'nın (PEPINN), GTX 1080 Ti GPU'da alternatif orijinal fizik bilgili sinir ağları (PINN) yöntemlerine kıyasla 183 kata kadar daha büyük veri üzerinden eğitilebildiği tespit edilmiştir.

Bu tezde önerilen ikinci yöntem ise, kararlı durum Navier-Stokes denklemlerinin öngörülen çözümünün başlangıç koşulu olarak sağlanmasının, çözüm sürecini hızlandırabileceği hipotezine dayanmaktadır. Bu yöntemde, U-net tabanlı bir mimari, başlangıç ve sınır koşulları belirlenerek, bütün problem alanının ayrıklaştırılmış bir temsili üzerinden eğitilmektedir. Eğitilen model, benzer durumların yakınsak çözümünü tahmin etmek için kullanılır ve elde edilen sonuçlar o problemin hesaplama ağına aktarılır. Bu yöntem, gelişigüzel aerodinamik profil etrafındaki kararlı, sıkıştırılamaz, sabsonik akış üzerinde test edilmiştir.

Anahtar Kelimeler: Kısmi Diferansiyel Denklemler, Fizik bilgili sinir ağları, Derin öğrenme, Hesaplamalı Akışkanlar Dinamiği, Parametik kodlama

To my family for their endless support.

ACKNOWLEDGMENTS

First and foremost, I would like to express my most sincere gratitude to Prof Dr. Oğuz Uzol for his constant support, guidance, and patience throughout the preparation of this study. I am deeply glad to have the chance to benefit from his invaluable experiences and insights.

I would like to give my special thanks to Dr. Erdal Oktay and EDA Engineering Design Analysis Ltd. Co. for their understanding and support throughout my Master's degree.

I must also express my gratitude to Assoc. Prof. Nilay Sezer Uzol and Asst. Prof. Dr. Ramazan Gökberk Cinbiş for their suggestions and comments during this study.

I also would like to express my gratitude to Suat Çiftçi who has helped in the configuration of the second part of the thesis.

It is a pleasure to thank my past and present colleagues and friends Ramin, Suat, and Fatma for their help and encouragement. I owe my deepest and warmest thanks to my family who gave me strength and encouragement throughout my studies.

I am so grateful to my mother Fatemeh (Mitra), my father Shahram, and my brother Danesh for their love and support throughout my whole life. This study would not have been possible without them.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xxi

CHAPTERS

1 INTRODUCTION 1

1.1 Motivation, Problem Definition and Contributions 1

1.2 Literature Review 4

1.2.1 Concepts Introduction 4

1.2.2 Prediction of the Solution Variables 5

1.2.3 Physics Informed Neural Networks 8

1.2.4 Domain Encoding 9

1.3 The Outline of the Thesis 11

2 SPEED UP OF PHYSICS INFORMED NEURAL NETWORKS 13

2.1 Introduction 13

2.2 PINN Formulation for Forward Problems 14
2.2.1 Formulation 15
2.3 Problem Description 19
2.4 Base Line Model Selection 20
2.5 Adam Optimizer 26
2.6 PINN Computation Graph Analysis 27
2.7 Finite Difference Kernels 30
2.8 Parametric Domain Encoding 35
2.9 Results 38
2.9.1 Baseline Model 38
2.9.2 Parametric Encoded PINN 43
3 SPEED-UP OF SOLUTION OF STEADY-STATE PDES 55
3.1 Introduction 55
3.2 Problem Description 56
3.3 Airfoil Generation 57
3.4 Mesh Generation 60
3.5 OpenFOAM Solver Setup 65
3.6 Trainable Model 66
3.7 Model Inputs and Outputs 68
3.8 Training Set Generation 72
3.9 Training 73
3.10 Results Inference and Updating the Initial and Boundary Conditions . 74
3.11 Results 74

3.11.1 Speed up Results With OpenFOAM Normalized Residuals . . 77
3.11.2 Residual Analysis 83
4 CONCLUSIONS 89
4.1 Future Work 90
REFERENCES 93
A QUALITATIVE RESULTS 101
A.1 Qualitative Results of Custom UNet Architecture 101
A.2 Qualitative Results of speed up of DE solvers by initial condition . . 104
A.3 Qualitative Results of the Parametric PINN model 108

LIST OF TABLES

TABLES

Table 2.1 Finite difference Kernels. 34

Table 2.2 PINN architecture selection results. The model with 8 layers and 120 neurons at the inner layers is selected as the baseline architecture. . . . 39

Table 2.3 PINN activation function selection results for architecture with 80 neurons and 6 layers. 41

Table 2.4 Architecture of Parametric PINN. 44

Table 3.1 Ranges for initial condition values of variables of RANS equations. Here the turbulence viscosity is set to be in accordance with OpenFOAM defaults for faster convergence. 62

Table 3.2 Modified UNet Encoder and decoder model architectures for learning the solution of RANS equations for incompressible, subsonic, steady state flow around 2D airfoils. (left) Encoder architecture building blocks. (right) Decoder Architecture building blocks. 68

Table 3.3 Scaling operations for model inputs and outputs. 73

LIST OF FIGURES

FIGURES

Figure 2.1 Tri-Linear channel-wise interpolation for parametric encoding. Here the feature grid represents each channel as well as the configuration of the parametric domain encoding. Each Block represents the corresponding feature that encompasses the domain and each smaller block represents the local sub-domain. In other words, here each block (large cubes) corresponds to the smallest bounding box that encompasses the domain. And there are "C" many copies of this bounding box. Also, every vertex on each block represents a real number. And the values for these real numbers are different from block to block. For a query point in the Spatio-Temporal domain corresponding to a sub-block (smaller cubes), the 8 bounding points of the sub-block are interpolated to obtain the representation of that point for that block. This operation is repeated for all the blocks to obtain a vector of size "C" that corresponds to the representation of the query point in the parametric domain. 37

Figure 2.2 The mean squared error of selected architecture with 8 layers and 120 intermediate neurons at each time station. Here the MSE for both velocity components $u_x = u$ (Blue), $u_y = v$ (Orange), as well as the pressure P (Green) is presented. The MSE is calculated for a spatial cross-section of the solution at different values of time (time stations) . 40

Figure 2.3 The mean squared error of selected architecture with GELU activation function provided an MLP with 6 layers and 80 intermediate neurons. Here the MSE for both velocity components $u_x = u$ (Blue), $u_y = v$ (Orange), as well as the pressure P (Green) is presented. The MSE is calculated for a spatial cross-section of the solution at different values of the time. 42

Figure 2.4 The mean squared error of selected architecture with GELU activation function provided an MLP with 8 layers and 120 intermediate neurons. Here the MSE for both velocity components $u_x = u$ (Blue), $u_y = v$ (Orange), as well as the pressure P (Green) is presented. The MSE is calculated for a spatial cross-section of the solution at different values of the time. 43

Figure 2.5 Comparison of the PEPINN with signature $4 \times 4 \times 3 \times 128$ with the selected baseline model and the MLP architecture used in RNN-DCT paper[62] when query point permutations of size $TYX = [16 \times 32 \times 32]$ is provided to all of the models for 20000 training iterations. . 45

Figure 2.6 Comparison of the PEPINN with signature $4 \times 4 \times 3 \times 128$ with baseline model and the model used in [62] when a query points permutation of size $16 \times 32 \times 32$ is provided to selected baseline models and query points permutation of size $32 \times 64 \times 64$ is provided for the PEPINN model during 20000 iterations. 46

Figure 2.7 Comparison of the PEPINN with signature $4 \times 4 \times 3 \times 128$ with baseline model and the model used in [62] when a query points permutation of size $16 \times 32 \times 32$ is provided to baseline models and maximum query points permutation of size $16 \times 32 \times 32$ is provided for the PEPINN model during 20000 iterations. 48

Figure 2.8 Comparison of the PEPINN with signature $4 \times 4 \times 3 \times 128$ with baseline model and the model used in [62] when a query points permutation of size $16 \times 32 \times 32$ is provided to baseline models and maximum query points permutation of size $32 \times 64 \times 64$ is provided for the PEPINN model during 20000 iterations. 49

Figure 2.9 Training PEPINN for 30000 iteration using different learning rates in Adam [75] optimizer. As it can be seen the model training is not affected by the learning rate value. 50

Figure 2.10 Accuracy of PEPINN vs Vanilla PINN models with 100/120 neurons and 6/8 layers with Tanh/GELU for x component of velocity. . 52

Figure 2.11 Accuracy of PEPINN vs Vanilla PINN models with 100/120 neurons and 6/8 layers with Tanh/GELU for y component of velocity. . 53

Figure 2.12 Accuracy of PEPINN vs Vanilla PINN models with 100/120 neurons and 6/8 layers with Tanh/GELU for pressure. 53

Figure 3.1 First 6 airfoils out of 1000 generated airfoils using CST method. It can be observed that even in this small subset of the generated airfoils some are similar to the airfoils used in practical applications while others are just airfoil-like shapes that might be impractical to use in engineering applications. Yet even in this limited number of cases a large range of variations in shape is present. 59

Figure 3.2 Ordering for mapping from a hexagon to a blockMesh block. The Black integers represent vertex numbering and the Red integers show the edge numbers. A block is represented in the code by ordering the vertices in accordance with the numbering presented above. The edges then are automatically assigned between the vertices in the presented order. Note that the edges represent curves rather than straight lines and thus are arc-length parameterized. 61

Figure 3.3 Top view of the blockMesh settings for generation of mesh around the airfoil. The blocks in this setting are configured such that the x_3 direction is defined span-wise. The mesh in this direction corresponds to extrude mesh of constant thickness. 62

Figure 3.4 Generated mesh around an arbitrary airfoil. The spacing of edge meshes over the airfoil in blocks B3 and B4 are non-uniform. They are selected such that more points are assigned to the nose with a shorter distance. Also, the spacing between the edge divisions increases geometrically as the airfoil upper and lower curves are traced from the leading edge to the trailing edges. This allows for a smoother transition in meshes from the nose radius to the arc created in front of the leading edge. 63

Figure 3.5 Generated mesh around an arbitrary airfoil. 64

Figure 3.6 Generated mesh around an arbitrary airfoil. 65

Figure 3.7 Trainable model for learning a mapping between the inputs of RANS equations to solver outputs after convergence. 67

Figure 3.8 Uniform grid representation of airfoils which is referred to as airfoil mask in the text. The image is in grey scale where 1 is mapped to white color and 0.0 is mapped to black color. 69

Figure 3.9 Input configuration of the model. For presentation purposes, the placement of channels is shown in a reversed order here. The mathematical representation precedence is the order with airfoil map, u, v, $\tilde{\nu}$ values for each query point stacked channel-wise. 70

Figure 3.10 Probe/interpolation query points. After generation, these points are queried in the solutions computation mesh to create the model's output references to be learned. 72

Figure 3.12 Original (top) and Refined (bottom) versions of the Goe115 airfoil shape. 76

Figure 3.13 Speed-Up ratio in the number of iterations for each test case. The dashed line (Red) represents the mean speed-up for all the cases in the test set. 78

Figure 3.14 Speed-Up ratio in Wall-time for each test case. The dashed line (Red) represents the mean speed-up for all the cases in the test set. . . . 79

Figure 3.15 Speed-Up in iterations vs number of solution iterations to converge with the original initial conditions. 80

Figure 3.16 Speed-Up in wall-time vs simulation Wall-time to converge with the original initial conditions. 80

Figure 3.17 Number of iterations before convergence (red) original cases, (blue) proposed cases for each test case. 81

Figure 3.18 Wall-time before the convergence of (red) original cases, (blue) proposed cases for each test case. 82

Figure 3.19 "c_p" distribution over the airfoil for the highest speedup in test case (naca65206, u=32.66, v=-2.28). 83

Figure 3.20 Pressure absolute residual of predicted vs original case. 84

Figure 3.21 Pressure normalized residual of predicted vs original case. 85

Figure 3.22 Pressure absolute residual of predicted vs original case. 86

Figure 3.11 Input and reference values for training the model. 87

Figure A.1 u_x component of the predicted field. 101

Figure A.2 u_y component of the predicted field. 102

Figure A.3 P; Pressure component of the predicted field. 102

Figure A.4 ν_t component of the predicted field. 103

Figure A.5 $\tilde{\nu}$ component of the predicted field. 103

Figure A.6 Predicted initial condition for magnitude of the velocity over the whole domain. 104

Figure A.7 Converged results for magnitude of the velocity over the whole domain. 105

Figure A.8 Predicted initial condition for pressure over the whole domain. . 105

Figure A.9 Converged results for pressure over the whole domain. 106

Figure A.10 Predicted initial condition for the magnitude of the velocity in the vicinity of the airfoil. 106

Figure A.11 Converged results for the magnitude of the velocity in the vicinity of the airfoil. 107

Figure A.12 Predicted initial condition for pressure in the vicinity of the airfoil. 107

Figure A.13 Converged results for pressure in the vicinity of the airfoil. . . . 108

Figure A.14 Converged results for $\tilde{\nu}$ in the vicinity of the airfoil. 109

Figure A.15 Converged results for $\tilde{\nu}$ in the vicinity of the airfoil. 109

Figure A.16 Converged results for $\tilde{\nu}$ in the vicinity of the airfoil. 110

Figure A.17 Converged results for $\tilde{\nu}$ in the vicinity of the airfoil. 110

Figure A.18 Converged results for $\tilde{\nu}$ in the vicinity of the airfoil. 111

Figure A.19 Converged results for $\tilde{\nu}$ in the vicinity of the airfoil. 111

Figure A.20 Converged results for $\tilde{\nu}$ in the vicinity of the airfoil. 112

LIST OF ABBREVIATIONS

1D	1 Dimensional
2D	2 Dimensional
3D	3 Dimensional
TGV	Taylor Green Vortex
PINN	Physics Informed Neural Networks
DL	Deep Learning
DB	Discretization Based
lr	Learning Rate
CFD	Computational Fluid Dynamics
DE	Domain Encoding
PD	Parametric Domain Encoding
DFE	Differential Equations
PDE	Partial Differential Equation
MLP	Multi-Layer Perceptron
CNN	Convolutional Neural Network
RNN	Recursive Neural Network
GAN	Generative adversarial network
cGAN	conditional generative adversarial networks
EDM	Encoder-Decoder Model
AE	Auto-Encoder
NN	Neural Network
NO	Neural Operator
NeRF	Neural Radiance Field
BC	Boundary Condition

IC	Initial Condition
IR	Implicit Representation
IRL	Implicit Representation Learning
LS	Latent Space
LV	Latent Variable
FCNN	Fully Connected Neural Network
FEM	Finite Element Method
FVM	Finite Volume Method
FDM	Finite Difference method
FDD	Finite Difference Based Derivatives
AD	Automatic Differentiation
ADD	Automatic Differentiation Based Derivatives
MSE	Mean Squared Error
RANS	Reynolds-averaged Navier–Stokes
CST	Class Shape Transformation
FLOPS	Floating Point Operations
ReLU	Rectified Linear Unit

CHAPTER 1

INTRODUCTION

1.1 Motivation, Problem Definition and Contributions

There is a rapid growth in the application of Deep Neural networks (DN) for solving differential equations. This newly emerging topic has attracted many researchers from different disciplines to itself and slowly started to produce fast and comparably more accurate results. Of course, there is still a substantial gap between the accuracy of DN models compared to discretization-based differential equation solvers, albeit a large portion of the DL-based models produces results orders of magnitude faster compared to their counterparts. The aforementioned DL-based solvers can be divided into two main categories. The first category learns the internal physics of the problem given enough data and is sometimes used for extrapolating to other unseen problems, whereas the second category aims to solve each problem by fitting a non-linear and differentiable model that satisfies the constraints of the problem setting (residual, initial, and boundary conditions of a partial differential equation-based problem).

Most of these methodologies either themselves can be sped up, or they can be used in conjunction with the existing discretization-based methods to make their solution process faster. Both of these approaches are investigated in this thesis.

In chapter 2, the shortcomings of physics-informed neural networks (PINNs) are investigated and a new method for training such implicit models is proposed. The motivation behind chapter 2 is that the vanilla PINN training performance suffers from the application of automatic differentiation (AD) (for both forward and backward modes) for higher-order differential equations solved by models with deep architectures. The AD constraints the time and memory complexity of training PINNs as a function of

the order of PDE and the number of layers of the architecture used for learning the mapping which will be explored in more detail in section 2.6. In chapter 2 the automatic differentiation in the PINN models is replaced by the finite difference kernels in order to estimate the derivatives of the solution variables with respect to domain variables. This removes the dependency of the derivative calculation time complexity with the model size and order of the PDE. The second motivation behind chapter 2 is that the learning capacity of PINNs can be increased (with no/minimal additional computational cost) and their training process can be made more robust by introducing dense parametric domain encoding as a backbone of the architectures used in training such models. With dense parametric encoding, the domain is divided into subdomains that are represented by local weights at the bounding vertices of each division. The (latent) representation of any query points belonging to a subdomain is then obtained by interpolating the weights representing that subdomain. Thus this model learns to satisfy the conditions imposed by the residual, as well as the initial and boundary conditions locally in the sub-domain parametric encoding rather than globally as the weights of a neural network. Another advantage of using parametric domain encoding is that it provides additional learnable parameters thus allowing a smaller decoder model (like multi-layer perceptron (MLP)) to be used for mapping the latent representation learned by the parametric encoding to the solution domain. In order to test the performance of the proposed method Taylor-Green Vortex problem is solved in chapter 2. It is then shown that applying the methods mentioned earlier speeds up the training process and allows for larger training data to be provided to the model for the same memory limit.

In chapter 3, a simple methodology for combining the results of the DL-based models trained on existing data is proposed where the inaccurate results of the prediction of the DL model are used as the initial and boundary conditions for the solution of steady-state differential equations. The main motivation in chapter 3 is that during design processes that involve the solution of differential equations a large number of candidate cases are needed to be simulated which requires a significant amount of time and resources to be dedicated in order to search for the best design candidates. Here it is hypothesized that this process can be made faster and reduce its required resources by simply training an end-to-end neural network that establishes a

mapping between the inputs of a small number of design candidates to their outputs. After training such a model, the trained mapping can be used to predict the solutions for the cases among the remaining design candidates. The predicted solutions will replace the initial and boundary conditions of each case and will be provided to the discretization-based solver to speed up its convergence process. The performance of this method is tested on the solution of Reynolds-averaged Navier–Stokes (RANS) equations for steady, incompressible, subsonic, and two-dimensional (2D) flow around airfoil shapes by solving the Spalart Allmaras turbulence model in OpenFOAM open-source software.

Our contributions can be divided into two categories. In the first category which involves the speed up of physics-informed neural networks for transient problems, the following contributions are made:

- The asymptotic time complexity of the PINN models is investigated and it is shown that the automatic differentiation for calculation of high order derivatives is not a good choice.
- Proposed a new architecture for training PINNs based on parametric domain encoding and the finite difference method that speeds up the training process with a minimum loss in accuracy. This model is named Parametric Encoded Physics Informed Neural Network (PEPINN).
- A procedure for faster, more robust training of PEPINN models is proposed.
- It is shown that the training process of PEPINN models is robust in terms of changes in learning rate.

For the second category which involves the speed up of steady state PDEs with learned solution predictions as the initial condition, the following contributions are made:

- We propose to learn the whole domain in RANS equations settings solving flow around 2D airfoils, rather than only the vicinity of the airfoil. And use the output of the model as the initial condition for existing solvers for steady-state problems. It is hypothesized that learning the whole domain makes the

convergence process of the solver more stable thus eliminating the requirement to iterate over the prediction-solution process for convergence.

- We use a non-uniform grid to learn the flow features around arbitrary airfoils and argue that using such a setting would allow higher resolution information about the parts of the domain that change drastically (for instance the vicinity of the airfoil) to be stored and learned.

- Unlike previous methods a one-to-one mapping between the inputs and outputs is not learned instead in order to provide more information about the geometry, the input is created in the vicinity of the geometry while the output is queried over the whole domain from a non-uniform grid.

1.2 Literature Review

1.2.1 Concepts Introduction

Machine learning approaches for learning to infer the solution or directly solve the PDEs have become an attractive topic in recent years. Of course, the advances in deep learning model architectures over the past decade, their astonishing results in processing, synthesis, or mapping signals in different modalities (image, sound, language), and the new suitable hardware development had a great impact on the adaptation of such models in a large range of research topics. Supported by the universal approximation theorem [1, 2, 3] these models achieve astonishing results -which sometimes surpass human ability- in image classification [4, 5, 6, 81, 8, 9], semantic segmentation [10, 11, 12, 13], image generation [14, 15, 16, 17, 18, 19, 20, 21], natural language processing and machine translation [22, 23, 24, 25], reinforcement learning [26, 27, 28, 29, 30] etc. Scientific machine learning has regained new momentum in recent years in the wake of advances in deep learning. Some of the early work in the adaptation of neural networks in the solution of differential equations can be traced back to [31], which used a multi-layer perceptron (A variant of Perceptron [32] with more than one layer) to estimate unmeasured process parameters in a fed-batch bioreactor and [33] where they also employed MLPs to solve ordinary differential equations with initial and boundary conditions by configuring the trial function such

that it consists of two parts. The first part of this trail function satisfied only the boundary conditions with no learnable parameters, whereas the second term involved the transformation of the output of the MLP such that it satisfied the ODE while being zero at the boundaries. Recently similar models (e.g. physics informed neural networks [34] by Raissi et al.) have been proposed. These models aim to find a nonlinear mapping from domain variables to the solution fields such that it satisfies given problem constraints. In this approach, similar to the discretization-based solvers, one model is trained on one problem which upon inference will yield the solution variables at each query point. There exists another approach in the solution of differential equations that use already available data for different problem settings to estimate/learn the statistics of the solution of such problems affected by the variations in problem settings. These methods either aim to find surrogate estimation to some final attributes of the solution variables like estimating the lift, or drag of airfoils from their shape (either as points or images) [35, 36, 37, 38] or they directly estimate the solution variables [39, 40, 41, 42, 43, 44]. In the following sections, the scientific research done on both of the aforementioned approaches will be discussed in more detail.

1.2.2 Prediction of the Solution Variables

Learning a mapping from the problem settings to the solution of differential equations is one way of solving such problems. Because of the complexity of the PDEs, the domain they are defined in and their initial and boundary conditions it was generally hard to obtain an analytical solution for such problems. As a result, a particular solution for a PDE given its constraints was approximately solved utilizing methods like finite element or finite volume. Yet for some special cases, other methods surfaced that could estimate the solution without directly solving the differential equation. For instance, thin airfoil theory or application of the panel method can be considered as such methods. The progress in machine learning and their ever-improving accuracy in learning the mapping between complex domains in an end-to-end manner, motivated researchers to test such models in learning the complex mapping between the input and solution domains of different types of differential equations. For instance, Farimani et al, [39] proposed adopting conditional generative adversarial networks

(cGAN) [45] methodology so that the distribution of possible solutions for a set of randomly generated problems for simple transport phenomenon to be learned. cGAN is different from the vanilla GAN model in that it provides some condition c in both the generator and discriminator/critic. It allows the model to learn the conditional probability of the generated output given some latent variable z and the conditional variable c. (A latent variable is a point in a high dimensional space which is referred to as a latent/unknown space. They are called latent spaces (and points/vectors defined in these spaces as latent vectors) because these high-dimensional spaces are learned through optimization and generally there is no clear description of why they turned out so.) In the case of the "Deep Learning the Physics of Transport Phenomena" [39] (motivated by [46]) the conditional variable c represented the domain shape as well as initial and boundary conditions. Thus the discriminator learned the joint probability between the complex inputs of the problem and its possible solutions. The discriminator in turn guided the generator of the model to output possible solutions to the specified condition c. Also using the conditional GAN methodology allowed the discriminator to implicitly learn the underlying differential equation in small convolution patches in their model.

Motivated by [39] Jiang et al. proposed FluidGAN [47] which learned a set of time-dependent, convective flow problems coupled with energy transport without providing the underlying governing equations. They showed that FluidGAN can learn the underlying coupling between pressure and momentum or/and energy and momentum in convective transport without specifically enforcing such an objective. Later Thuerey et al. [41] proposed to use an encoder-decoder setting utilizing UNet architecture to find a mapping to the solution of steady, 2D, incompressible flow around airfoils from UIUC [48] database. They proposed using the airfoil image as one channel of the input and the initial conditions for velocity components as the other two channels of the input. This Unet-based model outputs the two components of the velocity as well as the pressure in the vicinity of the 2D airfoil shape. In a similar work, Obiols-Sales et al [44] proposed CFDNet to solve flow around different shapes (for RANS simulation) in an iterative manner. They utilized a discretization-based solver as well as a CNN-based network to predict the steady state solution for flow variables given the results of the solver at particular iterations. As a result, the loop between

the discretization-based solver and the CNN-based solver allowed the solution to be obtained faster.

Another approach in the prediction of the solution of differential equations utilizes the reduced order latent variables in an encoder-decoder setting. In this approach, the ultra-high dimensional discretized problem is converted to a reduced-order latent variable. For instance, each problem with a large number of computational mesh cells (say 100000) is compressed to a considerably lower dimensional vector (say 128-dimensional latent vector) via an encoder and then this latent vector is decompressed to the same problem domain or the solution domain via a decoder. As an example, Farimani et al used a UNet [49] based network architecture in their generator model thus reducing the input $c \in \mathbb{R}^{1\times64\times64}$ to a latent representation in $\mathbb{R}^{512\times1\times1}$ leading to 8 times compression of the information in inputs. Yet [39] did not utilize the reduced order information represented in the latent space. Wiewel et al. in their paper "Latent Space Physics: Towards Learning the Temporal Evolution of Fluid Flow" [42] used a LSTM [50] based model to predict the changes in solution field variables over time using the latent space representation of the problem. They used a convolution-based encoder-decoder model (EDM) to encode and decode the domain information. The encoder in the EDM model was used to convert the problem settings to a latent vector and the decoder was used to convert the given latent vector representations to their Spatio-temporal manifestations. The latent representation of the domain and intermediate solutions for the fluid flow then were used in a recurrent model to learn the variations of the latent vector representation of the flow field in the domain and the change in its state (its solution) as a function of time.

Similar to [42] Kim et al. proposed Deep Fluids [43] which used a CNN-based autoencoder to compress the information in any instance of the fluid flow into a reduced dimension latent space. Then they trained another network which they refer to as a latent space integration network to learn the evolution of the flow in time subjected to the latent representation of the flow at the previous time step. This approach resulted in up to 700x speed up of velocity field generations, and domain information compression rates of up to 1300x. In addition, Li et al. [51] proposed Fourier Neural Operator which applies Fourier transform to the input and outputs sampled in a uniform grid for the inverse problem and then learns their mapping in the Fourier

space. Using the Fourier transform allows the model to predict higher resolution solutions as such solutions can directly be sampled from the learned Fourier transform representation of the solution.

1.2.3 Physics Informed Neural Networks

The physics-informed neural networks (PINN) [34] methodology has become an attractive research topic recently. According to [52] only in 2021, about 1300 papers related to the PINN method have been published. These papers progressively tackled different types of partial differential equation-based problems. The vanilla PINN model proposed solutions for Allen–Cahn equation, the Korteweg–de Vries equation, or the 1D nonlinear Schrödinger problem. Mao et al. [53] proposed to use PINNs to solve Euler equations for high-speed aerodynamic flows in forward and inverse problems for 1D and 2D domain settings. Jin et al [54] proposed NSFNets to simulate Navier-Stokes Equations . More specifically they experimented on incompressible Navier-Stokes flows, including 2D steady Kovasznay flow, 2D unsteady cylinder wake, and 3D unsteady Beltrami flow as well as turbulent channel flow. Cai et al [55] used PINN to solve various heat transfer problems.

The first application of the idea behind PINNs dates back to the 1990's [31, 33] yet they become popular after the publication of the [34] by Raissi et al. This research paper was motivated by [56, 57] which introduced methods to handle forward and inverse problems related to the solution of differential equations. They proposed using Gaussian processes to either determine the solution of the differential equation given its constraints or find the differential equation parameters given a set of data. Ever since the introduction of the PINNs by Raissi et al [34] which utilized MLPs for solving differential equations some new neural architectures have been proposed. Ren et al. [58] proposed PhyCRNet; a physics informed convolutional-recurrent neural network. Their model consisted of an encoder-decoder setting with a convLSTM [59], where the previous domain state with boundary conditions was provided as the input of the encoder, which then was passed to a convLSTM before applying the decoder. The decoder upsampled its input and applied a 2D convolution to its output. They utilized a recurrent model in order to preserve the previous temporal information

about the state of the solution thus increasing the accuracy of the model. They utilized finite difference rather than automatic differentiation as their input was a uniform grid even so training their model on Burger's equation took about 24 hours. Sun et al. [60] proposed physics-constrained neural networks.

This model enforced the initial and boundary conditions directly by the model rather than using them in the loss. This was done by a transformation of the output of the network as a function of the distance to the Spatio-temporal boundaries. Jagtap et al [61] proposed dividing the spatial domain into a set of disjoint domains and applying different MLP models at each subdomain. This approach requires the conservation lows to be satisfied in the boundaries of each subdomain. Most recently Wu et al proposed RNN-DCT [62]. This model motivated by Fourier Neural Operators predicts a parametric encoding of the initial and boundary conditions and then interpolates the query points in the grid in accordance with PINN methodology.

1.2.4 Domain Encoding

Encoding the inputs to higher dimensional spaces can be seen in the one-hot [63] as well as the kernel trick [64] where more complex combinations of the data can be created. A more complicated input encoding was proposed for RNNs in [24]. Later Vaswani et al. [25] proposed positional encoding in the architecture of the transformer where scalar positions were represented as a multi-resolution sequence of $L \in \mathbb{N}$ sine and cosine function pairs 1.1.

$$\begin{aligned} PE(x, 2i) &= sin(\frac{x}{10000^{\frac{2i}{d_{model}}}}) \\ PE(x, 2i+1) &= cos(\frac{x}{10000^{\frac{2i}{d_{model}}}}) \\ i &\in [0, L) \end{aligned} \tag{1.1}$$

Motivated by the observation by Rahaman et.al. [65] about the spectral bias of the neural networks which indicates that these models are biased toward learning low-frequency modes and that higher frequency information gets easier to learn when

these data are mapped to higher dimensional spaces using high-frequency functions, [66] proposed using another positional encoding of the form 1.2

$$\gamma(x) = (sin(2^0\pi x), cos(2^0\pi x), ..., sin(2^{L-1}\pi x), cos(2^{L-1}\pi x)) \tag{1.2}$$

Tancik et. al [67] also showed that a standard MLP fails to learn high frequencies data and in order to overcome that shortcoming they proposed random Fourier feature mapping and showed that this transformation can drastically improve the performance of coordinate-based MLPs.

Another type of encoding later emerged where additional trainable parameters in an auxiliary data structure like octrees or grids are arranged and looked up. These parameters sometimes are interpolated to estimate the local latent information in each subdomain. This model trades smaller computational cost for a larger memory requirement. In this approach during updating process of the model, only a subset of weights of the parametric encoding is changed. For instance, in 2D/3D domain only four/eight sets of c dimensional encoding weights are updated if bi/tri-linear interpolation is used. This allows for the addition of a larger learning capacity while keeping the computation cost at each optimization step comparably smaller. Utilizing parametric encoding, Chabra et al [68] used gird parametric encoding in their model to locally learn and compress the volumetric signed distance functions of geometric scenes. Each sub-grid then was used to reconstruct the geometric shapes using a decoder. Jiang et al [69] used an auto-encoder to learn the local embedding of the small shapes in a large 3D scene, then used the trained decoder in a dense grid parametric encoding setting to solve for the appropriate local latent codes of each subdomain. Liu et al [70] proposed Neural Sparse Voxel Fields which defines the grid parametric encoding on the bounds of voxels and uses a sparse octree to organize each subdomain and then progressively prune the underlying voxel structure. This allowed them to achieve an order of magnitude faster 3D scene reconstruction inference compared to Vanilla NeRF [66]. Peng et al [71] used an encoder to define the local parametric encoding on a grid, then utilized a bi/tri-Linear interpolation to obtain the features at each subdomain. Mehta et al [72] proposed Modulated Periodic Activations which maps the local grid parametric encoding (or local latent variable) corresponding to the

target signal to the parameters that modulate the amplitude of the periodic activations of another model called a synthesizer. Similar to [69] this method, by providing the encoded information of subsets of the domain allowed the model to be able to learn more generalized representations of the domain as a function of the local information. Yu et al [73] instead of utilizing a decoder directly used spherical harmonics (as a parametric encoding) in a voxel grid to directly infer the attenuation of the light through the medium and construct a radiance field. This method allowed the model to be optimized about two orders of magnitude faster than Neural Radiance Fields with no loss in visual quality. Müller et al [74] proposed a multi-resolution hash table of trainable parameters that can be optimized for different grid resolutions via gradient descent. This model reduced the training time of Radiance Field representations to only a couple of seconds compared to days in vanilla Nerf [66] or minutes in Plenoxels [73].

1.3 The Outline of the Thesis

In chapter 2 the Parametric Encoded Physics Informed Neural Network (PEPINN) model proposed in this thesis is presented. In section 2.6, the computation graph of the vanilla PINN models are investigated and it is shown that using automatic differentiation is not an efficient method for calculation of the residual of high order differential equations. Also in that chapter, the adaptation of the PEPINN model to solve Taylor-Green Vortex problem is presented.

In chapter 3, a methodology for the speed-up of fluid dynamics solvers by providing a predicted solution as an initial condition is presented. Also, the adaptation of this method to solve incompressible subsonic steady flow around 2D airfoils subjected to Reynolds-averaged Navier–Stokes equations is presented in that chapter.

In appendix A the qualitative results of each model is presented.

CHAPTER 2

SPEED UP OF PHYSICS INFORMED NEURAL NETWORKS

2.1 Introduction

Modern differential equation solvers which utilize the approximation capability of Neural Networks (NN) based models, automatic differentiation (AD), and gradient-based optimization techniques have become an attractive domain for scientific exploration in recent years. The advances in the Machine Learning, deep learning areas, hardware design and development of platforms that facilitate programming and training models that run on clusters, multi-GPUs, etc. made the development and adaptation of NN-based differential equation solvers much faster. Indeed according to [52] only in 2021 about 1300 related papers have been published. These differential equation solvers are divided into two different categories. In the first category, a nonlinear, and differentiable, model is trained to learn the implicit representation of the solution of a partial differential equation which is constrained by the residual of the PDE, and its initial and boundary conditions. The second method, on the other hand, learns the governing relations for solutions of families of problems by training a model that learns the likelihood of a specific solution given the problem description. An example of such a method was presented in chapter 3 whereas, in this chapter, the first method is investigated. Methods that are based on the first idea, constitute a new scientific machine learning technique for the solution of partial differential equations (PDE). In this method, the PDE solution is approximated by a Neural Network which minimizes a loss that consists of some transformation of the residual at random points inside the problem's space-time (ST) domain, as well as the initial and boundary conditions on points on the boundaries of ST domain. These models are referred to -in the literature- as physics-informed neural networks (PINNs). As the NN model used

in the training of the PINN is nonlinear, continuous, and differentiable, the differential operators can be directly calculated at each point (for the particular model) via automatic differentiation thus eliminating the need for the discretization of the domain and the governing differential equations. In other words, PINN does not require a computational mesh to be generated for each problem.

In this chapter, the vanilla PINN model proposed by Raissi et. al [34] will be introduced and adopted for the Taylor-Green Vortex problem which is a particular case of Navier-Stokes equations with analytical solutions. Later in this chapter, an architecture search will be introduced to select a baseline vanilla PINN model for the solution to Taylor-Green Vortex problem. Next, the issues with the utilization of automatic differentiation in the residual calculation for PINNs will be investigated. Also, the performance of PINN models will be improved by introducing a novel model; Parametric Encoded Physics Informed Neural Network (PEPINN) which utilizes Finite difference kernels and parametric domain encoding to solve the Taylor-Green Vortex problem. This chapter will be concluded by presenting the performance results of the PEPINN model for the solution of Taylor-Green Vortex problem.

2.2 PINN Formulation for Forward Problems

Physics-informed neural networks are a subset of implicit neural representation problems that are interested in a class of functions ϕ that satisfy equations of the form:

$$F(x, \phi, \nabla_x \phi, \nabla_x^2 \phi, ...) = 0; \phi : x \mapsto \phi(x) \tag{2.1}$$

The inputs of the forward problem are the Spatio-temporal coordinates $x \in \mathbb{R}^{d_x}$ (Throughout this study it is assumed that the temporal and spatial domains form a Banach space and any Spatio-temporal point can be represented by a d_x dimensional vector). Here ϕ is defined implicitly which can be parameterized by a neural network that minimizes function F in equation 2.1. In the subsequent sections, the general formulation of implicit neural representations and their adaptation to the solution of partial differential equations will be discussed.

2.2.1 Formulation

The objective in physics informed neural network models in a general sense is to find $\phi(x)$ subject to a set of M constraints C_m 2.2;

$$\begin{aligned} C_m(\gamma(x), \phi(x), \nabla(\phi(x)), ...) = 0, \\ \forall x \in \Omega_m, m = 1, ..., M \end{aligned} \tag{2.2}$$

Here $\gamma(x)$ represents the parameters of particular problem, and $\Omega_m \subset \Omega$ indicates the sub-domain for which the constraint $C_m(.)$ applies.

This problem can be reformulated in terms of a loss function that penalizes the deviations of the model from the constraints on their domain:

$$\mathcal{L} = \int_{\Omega} \sum_{m=1}^{M} \delta_m(x) \| C_m(\gamma(x), \phi(x), \nabla(\phi(x)), ...) \| d\Omega \tag{2.3}$$

$$\delta_m(x) = \begin{cases} 1, & x \in \Omega_m \\ 0, & otherwise \end{cases} \tag{2.4}$$

This integration is intractable but can be approximated by the Monte Carlo integration for input points x_i sampled from a uniform distribution in Ω;

$$\mathcal{L} \approx \frac{c_\Omega}{N} \sum_{i=1}^{N} \sum_{m=1}^{M} \delta_m(x_i) \| C_m(\gamma(x_i), \phi(x_i), \nabla(\phi(x_i)), ...) \| \tag{2.5}$$

In the equation 2.5 c_Ω represents a constant corresponding to $\int_\Omega d\Omega$ which generally is neglected in the formulation of PINNs.

In a physics-informed neural network, the loss is constructed from a combination of the constrained defined by the partial differential equation's residual, boundary, and initial conditions and measured points. Thus the general loss formulation 2.5 can be modified to represent the aforementioned factors.

$$\mathcal{L} = [\mathcal{L}_R + \mathcal{L}_B + \mathcal{L}_I + \mathcal{L}_{measured}] \tag{2.6}$$

with $\mathcal{L}_R$, $\mathcal{L}_B$, $\mathcal{L}_I$, $\mathcal{L}_{measured}$ corresponding to the constraint imposed by the PDE residual, boundary conditions, initial conditions, and measured data respectively. Each term in 2.6 can further be expanded to conform with the general form of the vanilla PINN model proposed by [34];

$$\begin{aligned} \mathcal{L}_R &\approx \frac{c_\Omega}{N} \sum_{i=1}^{N} \|R(x_r)\|^2 \\ \forall x_r \in \Omega, c_\Omega &= \int_\Omega d\Omega \end{aligned} \tag{2.7}$$

In literature the constant term c_Ω is generally neglected arriving at the following equation for residual loss:

$$\mathcal{L}_R \approx \frac{1}{N} \sum_{i=1}^{N} \|R(x_r)\|^2; \forall x_r \in \Omega \tag{2.8}$$

The boundary loss can be divided into four different categories which represent the loss for Dirichlet ($\mathcal{L}_{B_D}$), Neumann ($\mathcal{L}_{B_N}$), Mixed ($\mathcal{L}_{B_M}$), and periodic ($\mathcal{L}_{B_P}$) boundary conditions.

$$\mathcal{L}_B = \mathcal{L}_{B_D} + \mathcal{L}_{B_N} + \mathcal{L}_{B_M} + \mathcal{L}_{B_P} \tag{2.9}$$

where each loss term can be defined in the form presented in equation 2.10.

$$\begin{aligned} \mathcal{L}_{B_k} \approx \frac{c_{\partial\Omega_k^t}}{N_k} \sum_{i=1}^{N_k} \| f_k(x_i^k, \phi(x_i^k), \nabla\phi(x_i^N)) - f_k(x_i^k, \Phi(x_i^k), \nabla\Phi(x_i^k)) \|^2 \\ \forall x_i^k \in \partial\Omega_k^t, c_{\partial\Omega_k^t} = \int_{\partial\Omega_k^t} d\partial\Omega_k^t \end{aligned} \tag{2.10}$$

In equation 2.10 the term $\partial\Omega_k^t$ refers to the spatial boundary of the problem, Φ refers to the value of the boundary condition at point x_i^k, ϕ represents the model output at

point x_i^k, and $c_{\partial\Omega_k^t}$ is the area of the spatial boundary where the boundary condition k applies. For simplification of the notation, the spatial boundary will be denoted by $\partial\Omega$ which spans the whole temporal range of the problem, and $\partial\Omega^0$ will represent the span of all the spatial points at time zero.

Each term in the 2.9 subjected to 2.10 is expanded in equations 2.11-2.14.

$$\mathcal{L}_{B_D} \approx \frac{c_{\partial\Omega_D}}{N_D} \sum_{i=1}^{N_D} \|\phi(x_i^D) - \Phi(x_i^D)\|^2 \tag{2.11}$$

$$\mathcal{L}_{B_N} \approx \frac{c_{\partial\Omega_N}}{N_N} \sum_{i=1}^{N_N} \|\nabla\phi(x_i^N) - \nabla\Phi(x_i^N)\|^2 \tag{2.12}$$

$$\begin{aligned}\mathcal{L}_{B_M} \approx \frac{c_{\partial\Omega_M}}{N_M} \sum_{i=1}^{N_M} \|[a(x_i^M)\phi(x_i^M) + b(x_i^M)\nabla\phi(x_i^M)] - \\ -[a(x_i^M)\Phi(x_i^M) + b(x_i^M)\nabla\Phi(x_i^M)]\|^2\end{aligned} \tag{2.13}$$

$$\mathcal{L}_{B_P} \approx \frac{c_{\partial\Omega_P}}{N_P} \sum_{i=1}^{N_P} [\|\phi(x_i^P) - \Phi(x_i^P)\|^2 + [\nabla\phi(x_i^P) - \nabla\Phi(x_i^P)]\|^2] \tag{2.14}$$

The initial conditions can be treated in the same manner as the Dirichlet boundary conditions which lead to the loss 2.15;

$$\begin{aligned}\mathcal{L}_{B_I} \approx \frac{c_{\partial\Omega_I^0}}{N_I} \sum_{i=1}^{N_I} \|\phi(x_i^I) - \Phi(x_i^I)\|^2; \\ \forall x_i^I \in \partial\Omega_I^0\end{aligned} \tag{2.15}$$

Another term can be added to the loss of the Physics informed neural network models to fine-tune the implicit representation of the experimental or known results in the domain. These measured instances are treated as fixed points and thus the mean squared error of the model results and the measured instances is minimized for these points 2.16.

$$\mathcal{L}_{B_F} \approx \frac{1}{N_F}\sum_{i=1}^{N_F} \|\phi(x_i^F) - \Phi(x_i^F)\|^2; \quad \forall x_i^I \in D = \{x_i^F, \Phi_i^F\}_{i=1}^{N_F} \tag{2.16}$$

The final loss for the PINN model is summarized in equation 2.17.

$$\mathcal{L} = \mathcal{L}_R + \mathcal{L}_{B_D} + \mathcal{L}_{B_N} + \mathcal{L}_{B_M} + \mathcal{L}_{B_P} + \mathcal{L}_{B_I} + \mathcal{L}_{B_F} \tag{2.17}$$

This equation (2.17) refers to the main loss that is used in the vanilla PINN-based models. If any of the loss terms do not apply to the problem they can be taken out of equation 2.17. For instance, for steady problems without any points, the terms $\mathcal{L}_{B_I}$ and $\mathcal{L}_{B_F}$ can be ignored.

In the equation 2.1, it is assumed that there exists a continuous, and differentiable mapping from the input domain to the solution domain, satisfying a set of constraints that were defined in the section 2.2.1. In the vanilla PINN multi-layer perceptron (MLP) is utilized to learn such a mapping. Equation 2.18 presents the recursive formulation of the MLP models.

$$f_i = g_i(z_{i-1}) \tag{2.18a}$$

$$z_{i-1} = f_{i-1}W_{i-1} + B_{i-1} \tag{2.18b}$$

$$f_0 = x \tag{2.18c}$$

$$f_L = \phi(x;\theta) \tag{2.18d}$$

Here W_i and B_i correspond to the weights and biases (trainable variables) of the model. g_i are a set of the nonlinear functions applied to the affine transformation z_i. f_i represents the output of each layer thus f_L represents the output of MLP. With this definition, it can be assumed that input to the model is the output of the 0^{th} layer. The recursive formulation presented in the equation 2.18 also will be used to construct the computation graph of the set of transformations where traversing this

graph from inputs to outputs is oftentimes referred to as the forward propagation. In order to compute the gradients of the loss with respect to the trainable variables, the vector-Jacobian product is computed at each layer from outputs to the inputs in the computation graph which is referred to as backward propagation or reverse mode automatic differentiation. After the gradients are computed the trainable variables are updated with some form of gradient decent. This procedure is continued until a satisfactory loss is obtained. More details on the computation graph of the PINNs are presented in the preceding sections.

2.3 Problem Description

As a test case, Taylor-Green Vortex (TGV) problem is solved in this chapter. It is used as a benchmark problem for testing the differential equation solvers, as it has an analytical solution that makes it attractive when the accuracy of such solvers is investigated. Besides, it is based on Navier-Stokes Equations with the exclusion of energy terms. The governing equations of the TGV are presented in equation 2.19.

$$\nabla.\mathbf{u} = 0 \tag{2.19a}$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}.\nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{f} \tag{2.19b}$$

In equation 2.19, $\mathbf{u} = \{u, v, 0\}$ represents the velocity vector, and $\mathbf{f}$ represents the external forces applied to the fluid which is assumed to be zero. The exact closed form solution of the Taylor-Green Vortex over the domain $\Omega = x \times y \times t \in [0, 2\pi] \times [0, 2\pi] \times [0, T]$ are shown in the equation 2.20.

$$u_x = cos(x)sin(y)H(t) \tag{2.20a}$$

$$u_y = -sin(x)cos(y)H(t) \tag{2.20b}$$

$$p = \frac{-\rho}{4}(cos(2x) + cos(2y))H^2(t) \tag{2.20c}$$

$$H(t) = e^{-2\nu t} \tag{2.20d}$$

For this problem, the initial and boundary conditions can be calculated directly from the analytical solution presented in equations 2.20 by assigning the time as 0 for initial conditions and spatial boundaries in order to obtain the value of the boundary conditions.

The PINN loss for TGV consists of the terms presented in equation 2.21. Note that only the Dirichlet boundary conditions are present in the loss 2.21. It is also possible to add the Neumann boundary conditions by differentiating the analytical solution but they are ignored in this study.

$$\mathcal{L} = \mathcal{L}_R + \mathcal{L}_{B_D} + \mathcal{L}_{B_I} \tag{2.21}$$

The residual term for the loss can be further expanded to accommodate for each term in its governing equations. It is divided into three terms that represent the loss for continuity equation $\mathcal{L}_C$, and two momentum equations $\mathcal{L}_{M_x}$ and $\mathcal{L}_{M_y}$.

$$\mathcal{L}_R = \mathcal{L}_C + \mathcal{L}_{M_x} + \mathcal{L}_{M_y} \tag{2.22}$$

Then each loss term in $\mathcal{L}_R$ can be described as shown in equation 2.23.

$$\mathcal{L}_C = \frac{1}{N}\sum_{i=1}^{N} \| \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} \|^2; \forall x, y \in \Omega \tag{2.23a}$$

$$\mathcal{L}_{M_x} = \frac{1}{N}\sum_{i=1}^{N} \| \frac{\partial u_x}{\partial t} + u_x\frac{\partial u_x}{\partial x} + u_y\frac{\partial u_x}{\partial y} + \frac{1}{\rho}\frac{\partial p}{\partial x} - \nu(\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2}) \|^2 \tag{2.23b}$$

$$\mathcal{L}_{M_y} = \frac{1}{N}\sum_{i=1}^{N} \| \frac{\partial u_y}{\partial t} + u_x\frac{\partial u_y}{\partial x} + u_y\frac{\partial u_y}{\partial y} + \frac{1}{\rho}\frac{\partial p}{\partial y} - \nu(\frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2}) \|^2 \tag{2.23c}$$

2.4 Base Line Model Selection

In this section, the selection process for a baseline PINN architecture to solve the Taylor-Green Vortex problem is presented. Here a simple architecture search will

be performed instead of guessing the architecture for the PINN model in order to arrive at a fairly good model to map the domain variables to the particular solution of the Taylor-Green Vortex given its initial and boundary conditions. After selection, this baseline model will be used to investigate and compare the performance of the Parametric Encoded PINN proposed in this thesis. In order to select the baseline model, the candidate architectures must be evaluated by an objective function. The objective function utilized here is similar to the one used in the objective function used in [81] with some modifications. The original objective function used in [81] is presented in equation 2.24.

$$E(m) = ACC(m) \times [FLOPS(m)/T]^w \tag{2.24}$$

With ACC(m) and FLOPS(m) representing the accuracy and number of floating point operations of the model m and T denoting the training time. The objective function used here is a modification of the one presented in equation 2.24. The model has three outputs that have different scaling that needs to be considered in the calculation of the accuracy. Instead of accuracy, a combination of the residual of the model and its normalized mean squared error (MSE) is used in the calculation of the evaluation metric for the selection of the most suited architecture. The normalized MSE is presented in equation 2.25.

$$M(x) = \frac{\frac{1}{N}\sum_{n=1}^{N}(x - \hat{x})^2}{\frac{1}{N}\sum_{n=1}^{N}(x)^2} + \epsilon \tag{2.25}$$

Here the mean squared error of each model output is normalized to have the same scaling. The $\epsilon = 10^{-10}$ is added to have a strictly positive metric. The architecture evaluation metric is shown in equation 2.26.

$$E(m) = [\frac{-log(M(u)M(v)M(p))^{w_1} log(\mathcal{L}_R\mathcal{L}_I\mathcal{L}_B)}{2 * 6^3}][\frac{NL^2n^2}{T}]^{w_2} \tag{2.26}$$

In equation 2.26 the high error and high loss values are both penalized in favor of a lower normalized mean squared error. Both of these metrics are used in order to consider both the model's performance during training as well as its performance after the training. The w_1 and w_2 are hyper-parameters used for controlling the trade-off

between the different entities in the evaluation metric. Here w_1 is used to control the trade-off between the model error and model loss which is set to the value 2. The w_2 is used for controlling the trade-off between the model error indicators (before and after training), and the model's FLOPS which is set to the value of 0.07 (opposite of the value used in [81]). Model FLOPS is approximated by the asymptotic time complexity of calculation of the residual for each model which is discussed in section 2.6. In the equation, 2.26, N, L, and n represent the number of samples, number of layers, and number of neurons per layer respectively. As the convergence of PINN models take a long time the training process is limited to 20000 optimization iterations for all the cases. As a result, the evaluation metric is slightly modified such that the lowest loss values during training are used in the metric instead of the convergence/termination loss values. As the model convergence might not be possible during the limited number of iterations, the lowest loss is used as an estimate of the lower bounds of the loss that the model can achieve.

Here each vanilla PINN model is set to learn the solution of Taylor-Green Vortex for the domain $\Omega = x \times y \times t \in [0, 2\pi] \times [0, 2\pi] \times [0, 1]$. As used in vanilla PINN, here the MLP architectures are investigated where two different search operations are independently performed. The best number of layers and neurons for each layer is chosen provided Tanh function as the nonlinear activation of each layer. The best activation then is selected from the set of activation functions presented in equations 2.28-2.50 on a vanilla PINN model with 6 layers and 80 neurons per layer. The general form of the architectures adopted to the Taylor-Green Vortex problem is presented in equation 2.27.

$$f_i = g(z_{i-1}) \text{ for } i = 1, ..., L-1 \tag{2.27a}$$
$$z_{i-1} = f_{i-1}W_{i-1} + B_{i-1} \tag{2.27b}$$
$$f_0 = x \tag{2.27c}$$
$$f_L = \phi(x;\theta) = z_{i-1} \tag{2.27d}$$
$$W_i \in \mathbb{R}^{n\times n}; i = 1, ..., L-2 \tag{2.27e}$$
$$W_0 \in \mathbb{R}^{3\times n} \tag{2.27f}$$
$$W_{L-1} \in \mathbb{R}^{n\times 3} \tag{2.27g}$$
$$B_i \in \mathbb{R}^{1\times n}; i = 0, ..., L-2 \tag{2.27h}$$
$$B_{L-1} \in \mathbb{R}^{1\times 3} \tag{2.27i}$$

With L the number of layers and n the number of neurons in each layer. The variable g represents the activation function which is set to the following nonlinear functions for every layer in the model:

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1+\exp(-x)} \tag{2.28}$$

$$\text{Tanh}(x) = \tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)} \tag{2.29}$$

$$\text{HardTanh}(x) = \begin{cases} 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \\ x & \text{otherwise} \end{cases} \tag{2.30}$$

$$\text{Threshold}(x) = \begin{cases} x, & \text{if } x > \text{threshold} \\ \text{value}, & \text{otherwise} \end{cases} \tag{2.31}$$

$$\text{ReLU}(x) = (x)^+ = \max(0, x) \tag{2.32}$$

$$\text{LeakyRELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases} \tag{2.33}$$

$$\text{RReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{otherwise } ; a \sim \mathcal{U}(\frac{1}{8}, \frac{1}{3}) \end{cases} \tag{2.34}$$

$$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ wx, & \text{otherwise} \end{cases} \tag{2.35}$$

$$\text{ReLU6}(x) = \min(\max(0, x), 6) \tag{2.36}$$

$$\text{Hardsigmoid}(x) = \begin{cases} 0 & \text{if } x \leq -3, \\ 1 & \text{if } x \geq +3, \\ x/6 + 1/2 & \text{otherwise} \end{cases} \tag{2.37}$$

$$\text{silu}(x) = x * \sigma(x) \tag{2.38}$$

$$\text{Softplus}(x) = \frac{1}{\beta} * \log(1 + \exp(\beta * x)) \tag{2.39}$$

$$\text{Mish}(x) = x * \text{Tanh}(\text{Softplus}(x)) \tag{2.40}$$

$$\text{Hardswish}(x) = \begin{cases} 0 & \text{if } x \leq -3, \\ x & \text{if } x \geq +3, \\ x \cdot (x + 3)/6 & \text{otherwise} \end{cases} \tag{2.41}$$

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (e^x - 1), & \text{if } x \leq 0 \end{cases} \tag{2.42}$$

$$\text{CELU}(x) = \max(0, x) + \min(0, \alpha(e^{\frac{x}{\alpha}} - 1)) \tag{2.43}$$

$$\text{SELU}(x) = s(\max(0, x) + \min(0, \alpha * (e^x - 1))) \tag{2.44a}$$

$$s = 1.05070098735, \alpha = 1.6732632423 \tag{2.44b}$$

$$\text{GELU}(x) = x * \Phi(x); \tag{2.45}$$

Where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases} \tag{2.46}$$

The default value of the λ is 0.5.

$$\text{LogSigmoid}(x) = \log\left(\frac{1}{1 + e^{-x}}\right) \tag{2.47}$$

$$\text{SoftShrinkage}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases} \tag{2.48}$$

The default value of the λ is 0.5.

$$\text{SoftSign}(x) = \frac{x}{1 + |x|} \tag{2.49}$$

$$\text{Tanhshrink}(x) = x - \tanh(x) \tag{2.50}$$

For activation function selection $L = 6$ and $n = 80$ are used. On the other hand the number of layers and neurons on a set of models obtained from the the permutation $A = L \times n = [4, 6, 8, 10] \times [20, 40, 60, 80, 100, 120]$ with activation function Tanh 2.29 are tested. Each model is given the same inputs as all others during the 20000 iterations. These inputs are randomly generated and saved before training the models. The input size is limited by the largest input set that can occupy the memory of a Nvidia GTX 1080 Ti for the largest model i.e. $L = 10, n = 120$. For training the models Adam optimizer [75] was utilized with $lr = 10^{-3}$ and default β values.

2.5 Adam Optimizer

In this thesis, Adam [75] optimizer is utilized to update the model parameters. Different types of optimization algorithms based on gradient descent algorithms have been proposed. Depending on the type of problem, a different optimization algorithm may be used. For updating models with first-order derivatives, gradient descent or stochastic gradient descent (SGD) [82] may be utilized. Although the SGD is a popular algorithm and is widely used, its convergence is generally prolonged. One of the methods proposed to remedy this issue is the momentum idea [83]. This acts similarly to inertia and limits the rate of change of the gradients.

Even if the momentum algorithm and its alternative Nesterov Accelerated Gradient Descent [84] improve the convergence performance in a gradient descent setting they do not modify the learning rate. Many methods for the adoptive modification of the learning rate during the training process have been proposed. Of these AdaGrad [85], AdaDelta [86], RMSProp [87], and Adam [75] were widely used.

Adaptive moment estimation (Adam) [75] is another advanced SGD method, which introduces an adaptive learning rate for each parameter. It combines the adaptive learning rate and momentum methods. In addition to storing an exponentially decaying average of past squared gradients, like AdaDelta and RMSProp, Adam also keeps an exponentially decaying average of past gradients. Adam works well in practice and compares favorably to other adaptive learning rate algorithms. The momentum term in the Adam optimizer can be computed as an exponentially decaying average

of the past gradients (equation 2.51).

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.51}$$

With g_t referring to the gradient and m_t to the momentum term. The exponentially decaying average of the past squared gradients V_t is presented in equation 2.52.

$$V_t = \sqrt{\beta_2 V_{t-1} + (1 - \beta_2)(g_t)^2} \tag{2.52}$$

In equation 2.52, β_1 and β_2 are exponentially decaying rates which are generally set to values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$. These two terms $(V - t, m_t)$ are combined as presented in equation 2.53 to update the parameter θ.

$$\theta_{t+1} = m_t - \eta \frac{\sqrt{1 - \beta_2}}{1 - \beta_1} \frac{m_t}{V_t + \epsilon} \tag{2.53}$$

As $V_t \geq 0$ the $\epsilon = 10^{-8}$ is used in the denominator to prevent divisions by zero. In practice, optimization algorithms perform each update using a number of samples rather than the whole data set. The training data set can be divided randomly into batches to speed up the learning process. Then, the objective function can be computed by taking the average over only each batch. However, the use of batches in training provides a trade-off between the stability, convergence speed, and memory requirement. The generalization is usually better by utilizing small batches than large batches. Doing so, however, results in a high variance in gradient estimates. This situation can lead to the need for small learning rates to prevent instabilities in the training and the convergence of the training takes more epochs and memory.

2.6 PINN Computation Graph Analysis

In this section, the computation graph of MLP architectures for the calculation of the residual via automatic differentiation in PINN models is investigated. The recursive and simplified computation graph of an MLP is presented in equation 2.54.

$$z_i = f_i(z_{i-1}W_i) \tag{2.54a}$$

$$f_{i+1} = g_{i+1}(z_i) \tag{2.54b}$$

In equation 2.54 for simplified calculations, the bias term in affine transformations of each layer is neglected. Equation 2.55 shows the recursive formula to calculate the first derivative of outputs with respect to the inputs.

$$\frac{\partial f_{i+1}}{\partial x} = \frac{\partial f_{i+1}}{\partial g_{i+1}(z_i)} \frac{\partial g_{i+1}(z_i)}{\partial z_i} \frac{\partial z_i}{\partial f_i} \frac{\partial f_i}{\partial x} \tag{2.55a}$$

$$\frac{\partial f_{i+1}}{\partial x} = g'_{i+1}(z_i) W_i \frac{\partial f_i(z_{i-1})}{\partial x} \tag{2.55b}$$

The calculation of the first derivatives requires traversing the computation graph once for each output. This can be simplified by artificially providing the vector in the vector-Jacobian product. This vector would have the value one for each element and will have the same size as the output. In other words, for the Taylor-Green Vortex problem with the output variables $< u, v, p >$ one may assume a scalar output $y = u + v + p$, Then the partial derivative of y with respect to each of the variables u, v, p will be 1. As a result, one may provide y instead of the set of outputs $< u, v, p >$ as the scalar output of the automatic differentiation or provide the pre-computed gradients [1,1,1] as the vector in the vector-Jacobian product.

Investigating the computation graph of the first derivative, one may notice that for each layer, now there are two terms that are a function of the inputs, and these terms are multiplied together. The significance of these multiplied terms can be revealed in the calculation of higher order derivatives. The recursive computation graph for the calculation of the second derivatives is presented in equation 2.56.

$$\frac{\partial \frac{\partial f_{i+1}}{\partial x}}{\partial x} = \frac{\partial g'_{i+1}(z_i)}{\partial x} W_i \frac{\partial f_i(z_{i-1})}{\partial x} + g'_{i+1}(z_i) W_i \frac{\partial \frac{\partial f_i(z_{i-1})}{\partial x}}{\partial x} \tag{2.56a}$$

$$\frac{\partial g'_{i+1}(z_i)}{\partial x} = \frac{\partial g'_{i+1}(z_i)}{\partial z_i} \frac{\partial z_i}{\partial x} \tag{2.56b}$$

One may re-arrange equation 2.56 to arrive at the equation 2.57 which is a more compact representation for the computation graph of the second derivative.

$$\frac{\partial \frac{\partial f_{i+1}}{\partial x}}{\partial x} = g''_{i+1}(z_i)(\frac{\partial z_i}{\partial x})^2 + g'_{i+1}(z_i) W_i \frac{\partial \frac{\partial f_i(z_{i-1})}{\partial x}}{\partial x} \tag{2.57}$$

Assuming that the term $\frac{\partial z_i}{\partial x}$ is not stored (which is generally the case in backward mode automatic differentiation), it can be seen in the equation 2.57 that at each level of the calculation of the second derivative the recursion splits into two sub recursions in the computation graph. In order to illustrate this, let the following operators I(.) and G(.) represent two different parts of recursion at each layer (equation 2.58);

$$\begin{aligned} I(L) &= g''_L(z_{L-1})(\frac{\partial z_{L-1}}{\partial x})^2 \\ G(L) &= g'_L(z_{L-1}) W_{L-1} \end{aligned} \tag{2.58}$$

Using the operators I(.) and G(.) one may write the expanded form of the computation graph;

$$\begin{aligned} \frac{\partial^2 f_L}{\partial x^2} &= I(L) + G(L) \frac{\partial^2 f_{L-1}(z_{L-2})}{\partial x^2} \\ &= I(L) + G(L) I(L-1) + G(L) G(L-1) \frac{\partial^2 f_{L-2}(z_{L-3})}{\partial x^2} \\ &= I(L) + G(L) I(L-1) + ... + G(L) G(L-1) ... I(1) \end{aligned} \tag{2.59}$$

Thus the calculation of the second derivatives with automatic differentiation would require the backward traversal of the computation graph of the first derivative 2.55 to be performed L times as a result of the chain rule. The time complexity of the calculation of the first derivative of the model is $O(NLn^2)$ and the second derivative

is $O(NL^2n^2)$. With the same procedure, it can be shown that the time complexity of the calculation d^{th} order derivative of the outputs with respect to inputs is of the time and memory complexity of order $O(NL^dn^2)$.

Another observation that can be made here is that the high-order derivatives become zeros when activation functions with a second derivative equal to zero are used. For instance, it should be expected that the MLP models utilizing Rectified Linear Unit (ReLU) family (ReLU [76], LeakyReLU [77], PReLU [78], ...) activation functions would be faster in terms of performance for the calculation of higher order derivatives. Yet for those situations, one also should expect higher error values for the residual calculation that utilizes some transformation of higher-order derivatives of the outputs of the model with respect to its inputs (as they will be all zeros).

2.7 Finite Difference Kernels

It is evident from the section 2.6 that automatic differentiation is a poor choice for calculating the higher-order derivative terms in the PDE residual. Indeed the effectiveness of automatic differentiation is appreciated when the derivative of the output with respect to a large number of weights is to be calculated (e.g. training a neural network) which renders the alternative methods inefficient. On the other hand in the calculation of the residuals of partial differential equations, the alternative discretization-based differentiation methods would perform better as their calculation depends only on the size of the inputs and outputs of the model rather than the model size itself. To that end, the finite difference kernels might be used to calculate the residual independent of the model architecture thus making the training of PINNs considerably faster while decreasing the memory requirements of the calculation of higher order derivatives (and eventually the residual).

The finite difference kernels can be calculated by manipulating the Taylor series expansion of continuous functions. Here the uni-variate Taylor series expansion is used to show the method for estimation of the derivatives. This method then will be expanded for the calculation of multi-variable functions. In equation 2.60 the Taylor series expansion of a real valued function $f(x)$ which is differentiable at point $x = x_0$

is presented.

$$f(x) = f(x_0) + \sum_{i=1}^{\infty} \frac{\partial^i f(x_0)}{\partial x^i} \frac{(x - x_0)^i}{i!} \tag{2.60}$$

In the same way the equation 2.60 can be used to obtain the result of function f in the vicinity of x with some distance h which is shown in equation 2.61.

$$\begin{aligned} f(x + h) &= f(x) + \sum_{i=1}^{\infty} \frac{\partial^i f(x)}{\partial x^i} \frac{(h)^i}{i!} \\ f(x - h) &= f(x) + \sum_{i=1}^{\infty} \frac{\partial^i f(x)}{\partial x^i} \frac{(-h)^i}{i!} \end{aligned} \tag{2.61}$$

It can be assumed that higher order terms numerically vanish and the Taylor series can be approximated up to its n^{th} derivative term. For instance equation 2.62 presents the equation 2.61 approximated up to the second order derivative terms.

$$\begin{aligned} f(x + h) &\approx f(x) + h\frac{\partial f(x)}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} \\ f(x - h) &\approx f(x) - h\frac{\partial f(x)}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} \end{aligned} \tag{2.62}$$

The first order derivative of function $f(x)$ can be estimated by subtracting $f(x - h)$ from $f(x + h)$ which is presented in equation 2.63.

$$f(x + h) - f(x - h) \approx 2h\frac{\partial f(x)}{\partial x} \implies \frac{\partial f(x)}{\partial x} \approx \frac{f(x + h) - f(x - h)}{2h} \tag{2.63}$$

Using equation 2.63 one may configure a FD kernel as follows:

$$\frac{\partial f(x)}{\partial x} = [f(x + h), f(x - h)] \cdot [1, -1] * \frac{1}{2h} \tag{2.64}$$

Here $[f(x + h), f(x - h)]$ represent a vector of function values at points $x + h, x - h$ and the dervative is estimated by a dot product between function values vector and

FD kernel $\kappa = [1, -1]$ scaled by scaling variable $\frac{1}{2h}$. The error of this approximation is of order $O(h^2)$ as the second-order derivative terms were also included in the calculation.

A more general approach can be followed to obtain the kernel values. It involves finding a suitable set of multipliers so that only the derivative term of interest remains in the equation.

Let equation 2.65 be an approximation of the first order derivative of f with respect to the function variable x. Here coefficients A-E are selected so that the terms including $f(x)$, $\frac{\partial^2 f(x)}{\partial x^2}$ and,$\frac{\partial^3 f(x)}{\partial x^3}$ be eliminated. The resultant kernel would have an error of order $O(h^3)$.

$$\frac{\partial f(x)}{\partial x} \approx Af(x-2h) + Bf(x-h) + Cf(x+h) + Df(x+2h) \tag{2.65}$$

The expanded approximation of each term in equation 2.65, is presented in equation 2.66. Here the right hand side vector is configured to make all other derivative terms other than the first derivative vanish in the set of equations.

$$\begin{aligned} f(x+2h) &\approx f(x) + 2h\frac{\partial f(x)}{\partial x} + \frac{(2h)^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} + ... \\ f(x+h) &\approx f(x) + h\frac{\partial f(x)}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} + ... \\ f(x-h) &\approx f(x) - h\frac{\partial f(x)}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} + ... \\ f(x-2h) &\approx f(x) - 2h\frac{\partial f(x)}{\partial x} + \frac{(2h)^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} + ... \end{aligned} \tag{2.66}$$

One may rearrange equations 2.65, 2.66 to arrive at equation 2.67.

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ -2 & -1 & 1 & 2 \\ 2 & \frac{1}{2} & \frac{1}{2} & 2 \\ -\frac{8}{6} & \frac{-1}{6} & \frac{1}{6} & \frac{8}{6} \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{h} \\ 0 \\ 0 \end{bmatrix} \tag{2.67}$$

This results in the coefficients;

$$\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} \frac{1}{6} \\ \frac{-4}{3} \\ \frac{4}{3} \\ \frac{-1}{6} \end{bmatrix} \frac{1}{2h} \tag{2.68}$$

For multi-variable functions, a similar approach may be employed.

$$f(x+dx, y+dy) \approx f(x,y) + (dx\frac{\partial f(x,y)}{\partial x} + dy\frac{\partial f(x,y)}{\partial y}) + ... \tag{2.69}$$

After the Taylor series expansion of the function with respect to the independent variables is constructed, the partial derivatives of the function with respect to the inputs can be estimated at each principal direction. for instance for estimation of partial derivatives in the x direction in equation 2.69, dy will be set to 0 and the same approach as before is followed keeping the terms in y direction constant.

Of course, utilizing finite difference methods for the calculation of derivatives will add a source of error to the calculation of the residual as finite difference-based kernels are calculated from a combination of Taylor series expansions of arbitrary functions and as a result have an inherent error (which is of some specific order of the step size used for calculation of the derivatives). The second problem with the utilization of finite difference methods is the generation of a uniform grid. Of course, a non-uniform grid might be used for query points. This on the other hand would require the modification of the formulation of the finite difference kernels for each set of points as well as modifying the implementation of cross correlation operation to add the kernel change for each set of points. In this thesis, the former point query setting, i.e. using a uniform grid is considered due to the fact that obtaining the derivatives on a non-uniform grid requires extra calculations to be performed and thus is inefficient. The creation of a uniform grid might be realized in three different approaches. In the first approach, one may create a uniform grid over the whole domain. This approach requires a large number of query points to satisfy the accuracy requirements of the residual. In the second approach, one may create a set of smaller uniform grids in

subsets of the domain. This approach requires the optimization methodology and the loss to be modified. Also, this approach might require more iterations to converge yet it can be configured such that fewer points can be provided for a more accurate estimation of the residual. In the third approach, one may apply the model used for learning a mapping from the input domain to the solution domain, at random points in the vicinity of each point in different directions. This method for instance requires 11 extra queries of each input point with small perturbations to be able to obtain the residual for Taylor-Green Vortex via central difference. These extra queries will only contribute to the calculation of the derivatives for the initial random set of points. In this chapter, the first method is used to calculate the residual of Taylor-Green Vortex . The finite difference scheme used for the calculation of the derivatives in residual is presented in table 2.1.

Table 2.1: Finite difference Kernels.

Differentiation Order	Accuracy Order	Scheme	Kernel	scale	Error
First Derivative	Second order	Central difference	[-1,0,1]	$\frac{1}{2h}$	$O(h^2)$
First Derivative	Forth order	Central difference	[1,-8,0,8,-1]	$\frac{1}{12h}$	$O(h^4)$
First Derivative	Second order	Forward difference	[-3,4,-1]	$\frac{1}{2h}$	$O(h^2)$
First Derivative	second order	Backward difference	[1, -4, 3]	$\frac{1}{2h}$	$O(h^2)$
Second Derivative	Second order	Central difference	[1,-2,1]	$\frac{1}{h^2}$	$O(h^2)$
Second Derivative	Forth order	Central difference	[-1,16,-30,16,-1]	$\frac{1}{12h^2}$	$O(h^4)$
Second Derivative	Second order	Forward difference	[2, -5, 4, -1]	$\frac{1}{h^3}$	$O(h^2)$
Second Derivative	second order	Backward difference	[-1, 4, -5, 2]	$\frac{1}{h^3}$	$O(h^2)$

For calculation of the first and second-order derivatives in space, the fourth-order central difference is used for inner points, and for the boundaries either second-order forward or backward differences were utilized. The orientation of the kernel was properly adjusted for the calculation of partial derivatives in each spatial coordinate for data permutations of the form $t \times y \times x$. Here the permutation of x and y was reversed so the order in which they are represented by the Pytorch tensors resembles their geometric orientation in the domain. For the first-order derivatives in time, the second-order central difference was used for inner points. The second order forward difference was used at time 0 and the second order backward difference was utilized at termination time. In order to calculate the derivatives efficiently in Pytorch,

the 2D convolution function (Conv2D) was utilized to calculate the space derivatives overriding the convolution kernel with the finite difference kernels. For temporal difference calculations, 1D convolution/cross-correlation function (Conv1D) implementation was used where its kernel was changed with finite difference kernels. Inner and boundary points' derivatives were calculated separately and concatenated after the completion of convolution operations.

2.8 Parametric Domain Encoding

Input/Domain encoding has found popularity as it was used in the Transformer architecture [25] which is one of the most successful architectures in learning the interrelation between tokens. Since then, many more domain encoding models were proposed, tested, and analyzed. One of these models proposed in [79] has found great success in implicit neural representations. This model referred to as the parametric domain encoding, allows the encoding of the model to be determined by the optimization and to be stored locally in the domain. This model providing great flexibility in the learning process has been utilized in the neural radiance field domain which strives to solve the 3D representation of space (density and emitted color at each point in the volume) given its boundary conditions as 2D images at different stations solving the equation for the Attenuation. Training a vanilla Neural Radiance Field (NeRF) generally takes a long time (sometimes days) yet new models based on various domain parametric encoding decreased this time in orders of magnitude to somewhere close to a couple of minutes in [73] to even seconds in [74]. Both of these methods utilize parametric domain encoding to achieve these results. There is a great similarity between NeRF models and PINNs where the former aims to learn the implicit representation of the solution of line integrals over the light rays given a set of boundary conditions, while the latter tries to learn the solution for a more generalized set of PDE's given their initial and boundary conditions. This suggests that the methods used in NeRF can also be utilized in the training of the physics-informed neural networks. In this thesis, the dense domain parametric encoding with tri-linear interpolation is used as a backbone for solving partial differential equations. Figure 2.1 shows how different features are interpolated for each query point in the dense

domain encoding presented in this thesis.

For each query point in the parametric domain encoding, 8 points constituting the bounds of the corresponding subdomain are interpolated channel-wise to create the vector that represents the encoding of the any query point inside that subdomain. This encoding is a higher dimensional representation of the location of the query point that will be learned by the locally represented model and stored in its 8 bounding vertices constituting the subdomain. After the calculation of the high-dimensional representation of query points, an MLP model (decoder) will be applied to this encoding to map the latent representation of query points to the solution of the differential equation.

One issue that must be noted here is that the Taylor-Green Vortex is an elliptic differential equation thus it requires the information to travel in all directions. As a result, the settings where the solution information is presented locally might require some means to carry the information between different subdomains. This in turn will make the solver converge in more iterations. The same phenomenon happens in discretized solvers like finite element and finite volume where this issue is remedied by the introduction of flux term in the solution at each cell of the computational mesh. The same issue applies to parametric encoding. Yet as the information about the solution of two adjacent subdomains is shared in the weight matrices on their common boundary and their representation manifestation also satisfies the PDE, no other means for the information transfer is required. To speed up the solution and limit the number of adjusting iterations (for information transfer) the domain is represented by weights $w \in \mathbb{R}^{4\times4\times3\times128}$ thus dividing the domain into 18 subdomains. These weights are initialized from a normal distribution. An MLP is used as a decoder for the parametric encoding to map the interpolated representations to the output variables of the Taylor-Green Vortex problem.

Note that the decoder used in the parametric encoding can have less parameters and be substantially smaller than a counterpart MLP that directly maps the domain coordinates to the solution domain. To explain that, assume there exist an MLP with L layers and n neurons for each layer which maps the input coordinates to the solution of the differential equation. Also assume that the output of the m^{th} layer of this MLP which will be called $f_{1:m}(x)$ is the same as the parametric encoding that is learned. In

other words the $L - m$ final layers of the MLP which for simplicity will be referred to as $f_{m:L}(v)$ would work as the decoder of the PEPINN model. If one assume that $f_{m:L}(v)$ is the same for both models then a sufficiently large $f_{1:m}(x) \mapsto v$ is required to learn a mapping from the problem domain coordinates to the latent representation v. This implies that for a PEPINN model a smaller decoder (less parameters/layers) can be utilized to learn the solution of the PDE compared to a vanilla PINN.

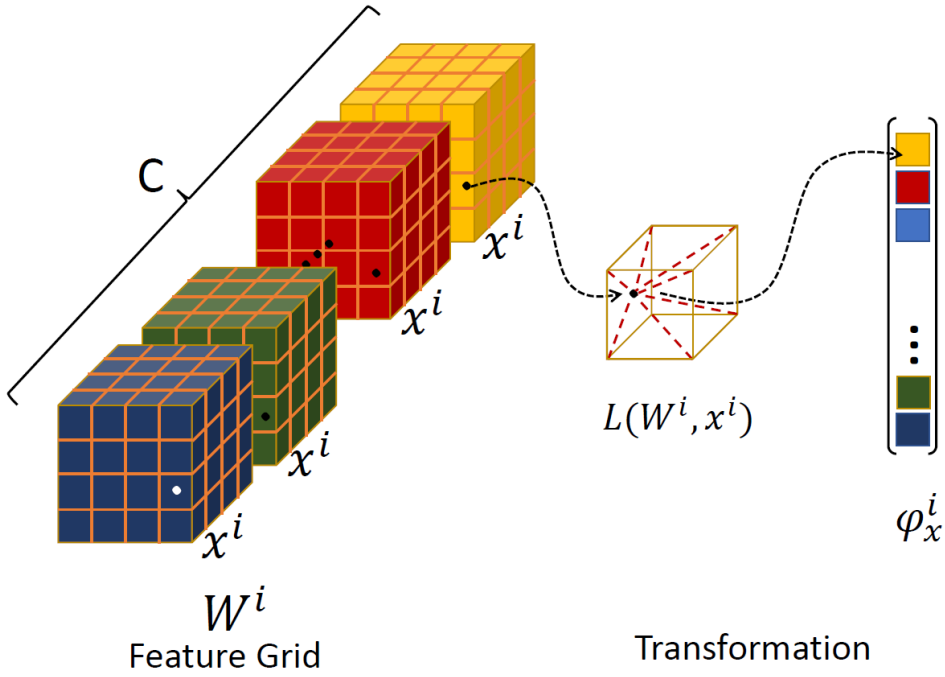

Figure 2.1: Tri-Linear channel-wise interpolation for parametric encoding. Here the feature grid represents each channel as well as the configuration of the parametric domain encoding. Each Block represents the corresponding feature that encompasses the domain and each smaller block represents the local sub-domain. In other words, here each block (large cubes) corresponds to the smallest bounding box that encompasses the domain. And there are "C" many copies of this bounding box. Also, every vertex on each block represents a real number. And the values for these real numbers are different from block to block. For a query point in the Spatio-Temporal domain corresponding to a sub-block (smaller cubes), the 8 bounding points of the sub-block are interpolated to obtain the representation of that point for that block. This operation is repeated for all the blocks to obtain a vector of size "C" that corresponds to the representation of the query point in the parametric domain.

2.9 Results

In this section, the performance of the parametric encoded physics-informed neural network proposed in this thesis is investigated. Its results are compared with the performance of the selected baseline PINN model for the solution of the Taylor-Green Vortex problem. This part consists of two sections. In the first section, the baseline model for the solution of the Taylor-Green Vortex problem is selected, and the performance and shortcomings of different PINN architectures are investigated. In the second section, the performance of the PEPINN model is compared to the selected baseline PINN architecture.

2.9.1 Baseline Model

In order to select a baseline model a series of experiments with Taylor-Green Vortex equations given the kinematic viscosity with the value of 0.0001 have been conducted and the architecture with the highest score is selected as the baseline model. This baseline model is used to compare the performance of the parametric encoded PINN architectures. As noted in the chapter 2.4 a series of MLP architectures are tested with the Tanh activation function for 20000 training iterations given a domain point query size of $[32 \times 32 \times 16]$, boundary query points of size $[32 \times 32 \times 100]$, and initial condition query points of size $[32 \times 32 \times 1]$. Query points are the set of points that are provided to the model at each iteration. The physical coordinates of these points are provided to the PINN models. In the case of the PEPINN their coordinates and the coordinates of the local bounding subdomain are used to interpolate the corresponding latent vector corresponding to the query points. For activation function selection the same data points are provided to a MLP model with 80 neurons for intermediate layers with 6 layers in total. This model's performance in training a PINN for Taylor-Green Vortex is investigated given the activation functions discussed in chapter 2.4. In order to select the most suitable architecture, the scoring equation 2.26 presented in chapter 2 section 2.4 is used.

The score is calculated as a function of minimum residual, initial, and boundary condition losses during training, the wall time, FLOPS approximation, and the accuracy

of the model at the end of 20000 iterations. The effect of the number of neurons and the number of the MLP layers in the PINN model training is presented in table 2.2.

Table 2.2: PINN architecture selection results. The model with 8 layers and 120 neurons at the inner layers is selected as the baseline architecture.

n-L	Wall-time	min Loss	min Residual	min IC	min BC	MSE(u)	MSE(v)	MSE(p)	Score
20 - 4	2.04e+03	4.56e-04	3.68e-04	5.01e-05	3.80e-05	1.44e-05	1.47e-05	1.83e-05	1.52e+02
40 - 4	2.16e+03	1.48e-04	1.14e-04	1.81e-05	1.54e-05	6.64e-06	6.06e-06	9.44e-06	2.13e+02
60 - 4	2.23e+03	1.82e-04	1.26e-04	3.78e-05	1.81e-05	6.22e-06	8.29e-06	8.71e-06	2.15e+02
80 - 4	2.66e+03	1.51e-04	1.22e-04	1.67e-05	1.26e-05	6.08e-06	5.30e-06	1.13e-05	2.33e+02
100 - 4	2.86e+03	4.90e-04	2.19e-04	1.93e-04	7.81e-05	6.49e-06	6.29e-06	1.10e-05	2.00e+02
120 - 4	2.95e+03	1.03e-03	3.37e-04	5.33e-04	1.60e-04	4.55e-06	3.43e-06	8.35e-06	2.01e+02
20 - 6	2.25e+03	3.31e-04	2.41e-04	5.34e-05	3.58e-05	1.47e-05	1.61e-05	1.36e-05	1.64e+02
40 - 6	2.49e+03	1.17e-03	5.31e-04	4.55e-04	1.84e-04	4.00e-05	4.59e-05	7.95e-05	1.18e+02
60 - 6	2.65e+03	6.27e-04	4.77e-04	8.08e-05	6.92e-05	4.31e-05	3.21e-05	9.68e-05	1.40e+02
80 - 6	3.50e+03	2.60e-04	1.27e-04	9.42e-05	3.92e-05	7.05e-06	5.33e-06	1.04e-05	2.19e+02
100 - 6	3.81e+03	1.87e-04	9.78e-05	6.08e-05	2.81e-05	4.66e-06	7.15e-06	1.14e-05	2.33e+02
120 - 6	3.99e+03	2.48e-04	9.32e-05	1.10e-04	4.42e-05	5.67e-06	5.49e-06	8.99e-06	2.34e+02
20 - 8	2.30e+03	2.35e-03	8.50e-04	1.32e-03	1.81e-04	3.49e-05	3.54e-05	3.10e-05	1.15e+02
40 - 8	2.79e+03	2.90e-03	9.41e-04	1.50e-03	4.63e-04	3.36e-05	2.43e-05	6.88e-05	1.16e+02
60 - 8	3.08e+03	9.72e-04	3.21e-04	4.87e-04	1.63e-04	1.18e-05	1.09e-05	1.06e-05	1.76e+02
80 - 8	4.23e+03	1.70e-04	9.87e-05	4.67e-05	2.46e-05	6.99e-06	6.88e-06	1.13e-05	2.32e+02
100 - 8	4.67e+03	7.95e-04	1.99e-04	4.08e-04	1.88e-04	3.94e-06	3.99e-06	9.63e-06	2.13e+02
120 - 8	**4.89e+03**	**1.68e-04**	**4.65e-05**	**8.79e-05**	**3.32e-05**	**2.68e-06**	**2.47e-06**	**5.05e-06**	**2.81e+02**
20 - 10	2.58e+03	3.60e-03	2.02e-03	1.33e-03	2.56e-04	2.12e-04	3.08e-04	1.25e-03	6.30e+01
40 - 10	3.18e+03	3.61e-04	2.22e-04	9.92e-05	3.99e-05	1.36e-05	1.18e-05	2.80e-05	1.82e+02
60 - 10	3.47e+03	1.26e-04	7.13e-05	3.83e-05	1.64e-05	4.98e-06	4.40e-06	1.17e-05	2.50e+02
80 - 10	4.97e+03	4.69e-04	1.72e-04	2.09e-04	8.84e-05	7.16e-06	7.36e-06	1.26e-05	2.07e+02
100 - 10	5.58e+03	2.75e-04	1.15e-04	1.12e-04	4.85e-05	3.87e-06	4.30e-06	7.34e-06	2.48e+02
120 - 10	5.90e+03	2.59e-04	1.04e-04	1.20e-04	3.42e-05	2.30e-06	2.88e-06	6.81e-06	2.71e+02

Here the architecture that has six intermediate layers with 120 neurons is selected. This model achieves a moderate loss with a value of 1.68e-04 and a residual loss with a value of 4.65e-05. The accuracy of the model which achieves the lowest loss value during training is presented in figure 2.2. As it can be seen the error of the solution increases rapidly (especially for the pressure) as the time increases. Yet the average values for the MSE are within satisfactory margins.

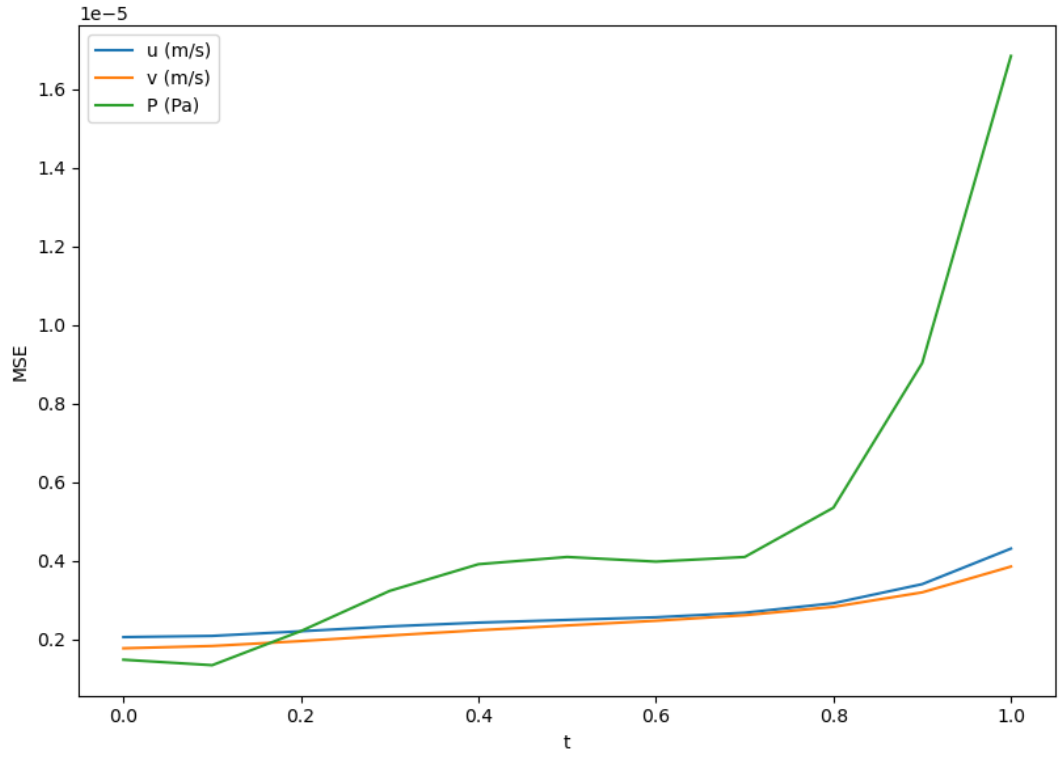

Figure 2.2: The mean squared error of selected architecture with 8 layers and 120 intermediate neurons at each time station. Here the MSE for both velocity components $u_x = u$ (Blue), $u_y = v$ (Orange), as well as the pressure P (Green) is presented. The MSE is calculated for a spatial cross-section of the solution at different values of time (time stations)

The second experiment for the selection of the baseline model involves the selection of a proper activation function. here it is assumed that the performance of the model due to architecture parameters (number of neurons and number of layers) is independent of the activation function selection. To that end, for activation function selection the model with 80 neurons and 6 layers is used. The comparison of the highlighted results of using different nonlinear activation functions in the training of the PINN model on the Taylor-Green Vortex problem is presented in the table 2.3. Note that the ReLU-based activation functions are still used in the activation function selection experiments. They do not produce considerable results as they are not able to return a valid residual as the second order terms will always be zero for MLPs using these activation functions (see chapter 3 section 2.6).

Table 2.3: PINN activation function selection results for architecture with 80 neurons and 6 layers.

Activation	Wall-time	min Loss	min Residual	min IC	min BC	MSE(u)	MSE(v)	MSE(p)	Score
Tanh	3.50e+03	2.60e-04	1.27e-04	9.42e-05	3.92e-05	7.05e-06	5.33e-06	1.04e-05	2.19e+02
LeakyReLU01	3.00e+03	7.37e-03	6.65e-03	4.97e-04	2.23e-04	5.26e-04	4.40e-04	4.39e-04	6.89e+01
LeakyReLU02	3.01e+03	6.91e-03	6.27e-03	4.25e-04	2.18e-04	2.01e-04	2.16e-04	5.97e-04	7.82e+01
LeakyReLU03	3.01e+03	9.14e-03	7.43e-03	1.16e-03	5.49e-04	3.01e-04	2.85e-04	5.54e-04	6.70e+01
ReLU	3.03e+03	8.37e-03	7.52e-03	5.81e-04	2.65e-04	2.52e-04	3.66e-04	5.72e-04	7.12e+01
ELU	3.68e+03	1.07e-03	5.14e-04	4.12e-04	1.47e-04	8.17e-06	7.06e-06	1.52e-05	1.77e+02
Hardshrink	3.02e+03	3.11e-02	4.03e-03	2.10e-02	6.07e-03	2.52e-02	1.65e-02	2.82e-02	1.15e+01
Hardtanh	3.01e+03	3.18e-02	2.39e-02	5.15e-03	2.85e-03	2.64e-03	2.16e-03	4.52e-03	2.81e+01
LogSigmoid	3.64e+03	1.62e-04	1.11e-04	3.34e-05	1.74e-05	5.35e-06	5.50e-06	6.11e-06	2.44e+02
PReLU	3.88e+03	6.82e-03	6.17e-03	4.28e-04	2.22e-04	2.03e-04	1.72e-04	4.54e-04	7.99e+01
RELU6	3.03e+03	7.48e-03	6.73e-03	5.18e-04	2.32e-04	2.06e-04	2.06e-04	5.31e-04	7.78e+01
RReLU	3.05e+03	1.89e+00	3.71e-05	1.17e+00	7.14e-01	2.04e-01	2.29e-01	3.56e-01	1.08e+00
SELU	3.67e+03	2.06e-03	1.90e-03	1.04e-04	5.89e-05	2.18e-04	2.72e-04	1.91e-04	9.76e+01
CELU	3.68e+03	2.56e-04	2.12e-04	2.88e-05	1.47e-05	6.66e-06	7.35e-06	6.82e-06	2.33e+02
GELU	**4.85e+03**	**1.39e-05**	**1.11e-05**	**1.55e-06**	**1.20e-06**	**5.96e-07**	**7.12e-07**	**1.81e-06**	**4.00e+02**
Sigmoid	3.56e+03	3.04e-04	1.23e-04	1.37e-04	4.42e-05	4.53e-06	3.75e-06	5.83e-06	2.32e+02
SiLU	4.31e+03	1.36e-04	3.68e-05	7.26e-05	2.63e-05	4.64e-07	4.49e-07	1.33e-06	3.33e+02
Mish	5.07e+03	6.35e-05	2.35e-05	3.01e-05	1.00e-05	9.34e-07	7.97e-07	3.32e-06	3.19e+02
Softplus	3.74e+03	4.32e-04	1.51e-04	2.12e-04	6.90e-05	4.47e-06	3.57e-06	4.44e-06	2.26e+02
Softshrink	3.03e+03	7.96e-01	0.00e+00	6.02e-01	1.94e-01	2.50e-01	2.50e-01	1.02e-01	-
Softsign	6.04e+03	3.60e-04	3.12e-04	2.91e-05	1.97e-05	2.05e-05	1.08e-05	1.15e-05	1.95e+02
Tanhshrink	3.75e+03	1.56e-03	4.15e-04	7.07e-04	4.38e-04	1.81e-05	2.05e-05	2.56e-05	1.45e+02

According to the scoring function presented in 2.26, the GELU nonlinear activation function has the highest score in learning the solution domain of the Taylor-Green Vortex problem. Another noteworthy activation function here is the softshrink which does not permit the model to learn the solution domain. It should be noted that the residual of the model with softshrink is equal to zero yet the loss for the initial and boundary conditions have high values. This happens when the gradient vanishes during the training steps. Reconsidering the Navier-Stokes equation for Taylor-Green Vortex 2.19 it can be seen that if the gradients with respect to the input are zero then the residual also becomes zero. The error of the model chosen for the activation function is presented in figure 2.3.

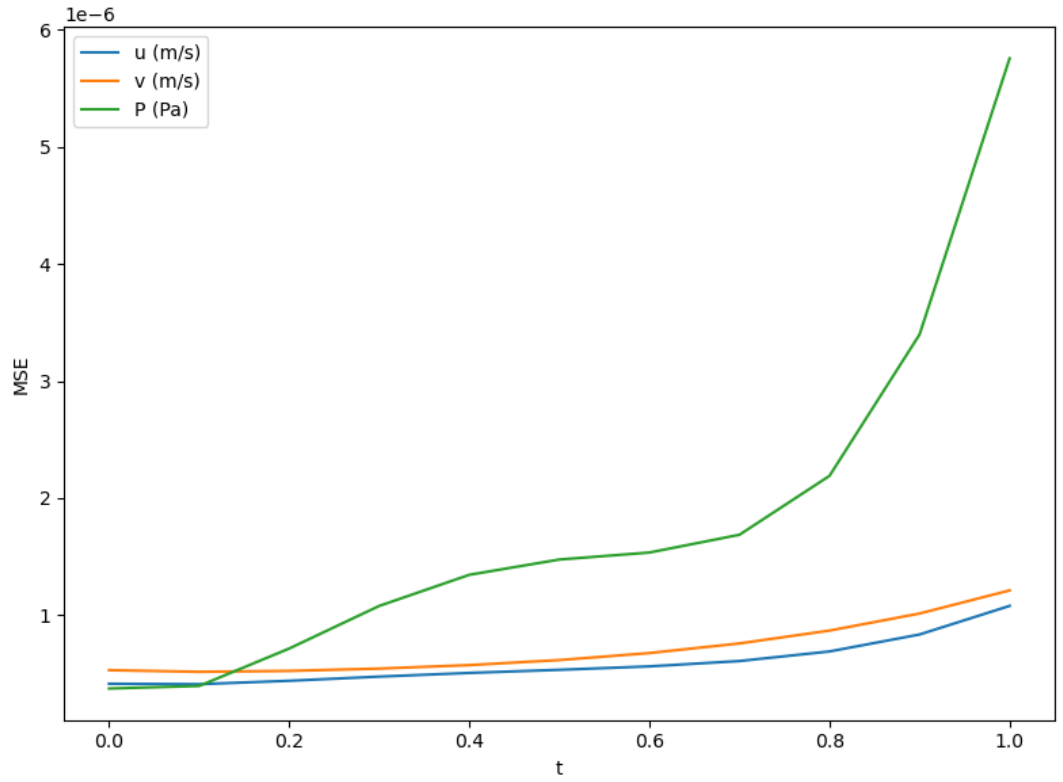

Figure 2.3: The mean squared error of selected architecture with GELU activation function provided an MLP with 6 layers and 80 intermediate neurons. Here the MSE for both velocity components $u_x = u$ (Blue), $u_y = v$ (Orange), as well as the pressure P (Green) is presented. The MSE is calculated for a spatial cross-section of the solution at different values of the time.

It can be seen that the MSE error of the architecture with the GELU activation function is one order of magnitude lower than that of the Tanh-based architecture. After the selection of the number of neurons and number of layers for the MLP as well as its activation function, this new model is trained on the same data. The error of the baseline model with 8 layers and 120 neurons for internal layers with the GELU activation function is presented in figure 2.4. As it can be observed the MSE for the pressure is about 5 times less than that of the model results (the model with 6 layers and 80 Neurons per layer) presented in figure 2.3.

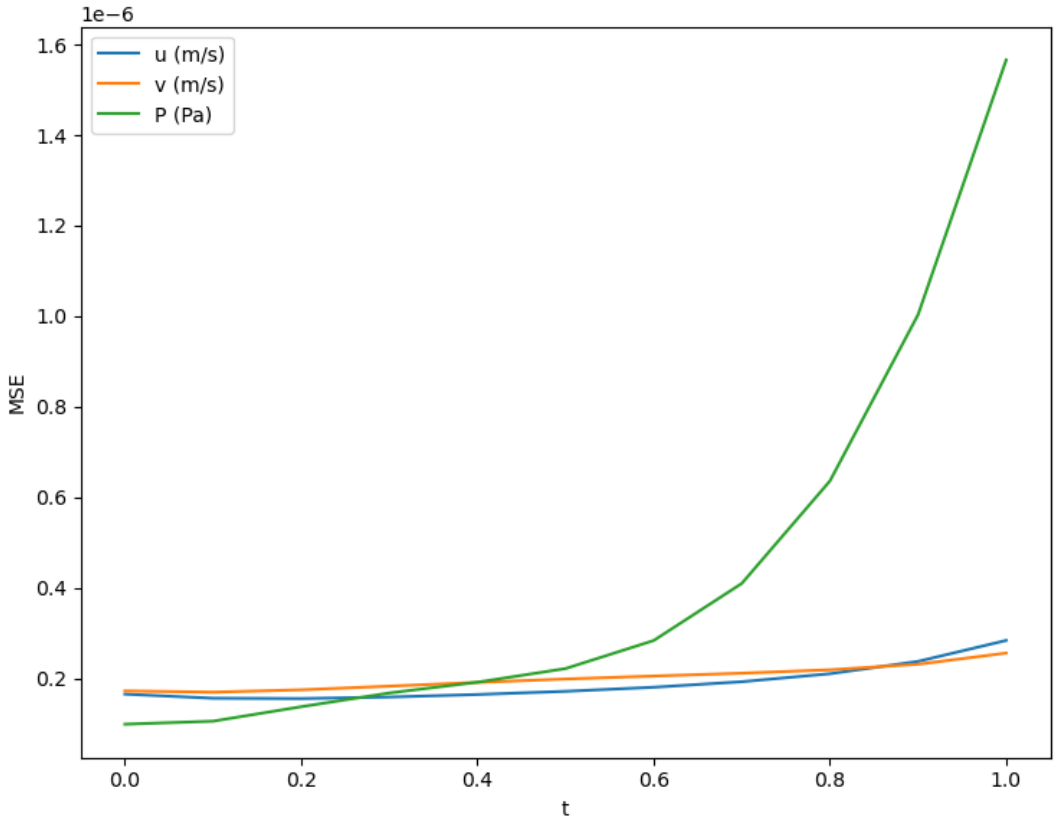

Figure 2.4: The mean squared error of selected architecture with GELU activation function provided an MLP with 8 layers and 120 intermediate neurons. Here the MSE for both velocity components $u_x = u$ (Blue), $u_y = v$ (Orange), as well as the pressure P (Green) is presented. The MSE is calculated for a spatial cross-section of the solution at different values of the time.

2.9.2 Parametric Encoded PINN

There have been many proposed approaches for the architecture design of PINN models. As the objective of this thesis is to speed up such models (with minimum loss in accuracy) a multipurpose method based on the parametric encoding of the domain is proposed and tested. This model -which will be referred to as PEPINN or parametric encoded PINN- divides the Spatio-temporal domain into a set of subdomains via local parametric encoding. A decoder is then used to translate the latent space represented by the local encoding to the solution variables. One such architecture is presented in

table 2.4. This simple architecture will be used for all the comparisons in this section.

Table 2.4: Architecture of Parametric PINN.

Layer	Act.	Output Shape
-	-	$NTYX \times 3$
PE $CX \times CY \times CT \times C$	-	$NTYX \times 128$
Linear 128×128	Tanh	$NTYX \times 128$
Linear 128×128	Tanh	$NTYX \times 128$
Linear 128×128	Tanh	$NTYX \times 128$
Linear 128×3	Tanh	$NTYX \times 3$

With N representing the batch size and T, Y, and X corresponding to the permutations of the data points in time, and spatial y and x directions. Here CX, XY, and CT represent the divisions of the Saptio-Temporal coordinates in x, y, and t directions respectively. C represents the feature size of each weight in the parametric encoding. In other words, the x-axis will be represented by CX many trainable vectors of length C. The same applies to the y and t principal axes of the problem. The default value for N is set to 1 on the other hand the values of the TYX permutation might change during the training. The loss values are calculated in the same manner as PINN yet the finite difference method is used for derivative calculations thus disconnecting the residual calculation time and memory complexity from the architecture size and its number of parameters. Of course, the application of the finite difference can be considered as an additional source of error albeit this extra error can be compensated by a larger size of query points available to the PEPINN model. In this section, two possible scenarios are investigated. The first scenario is related to the speed-up performance of the model given the same data query size used for the baseline model selection. The second scenario involves providing a proper number of query data and controlling the accuracy of the residual calculation. After testing the aforementioned cases the training performance of the proposed model under different learning rates is investigated and this section is concluded by comparing the accuracy of PEPINN vs the accuracy of vanilla PINN.

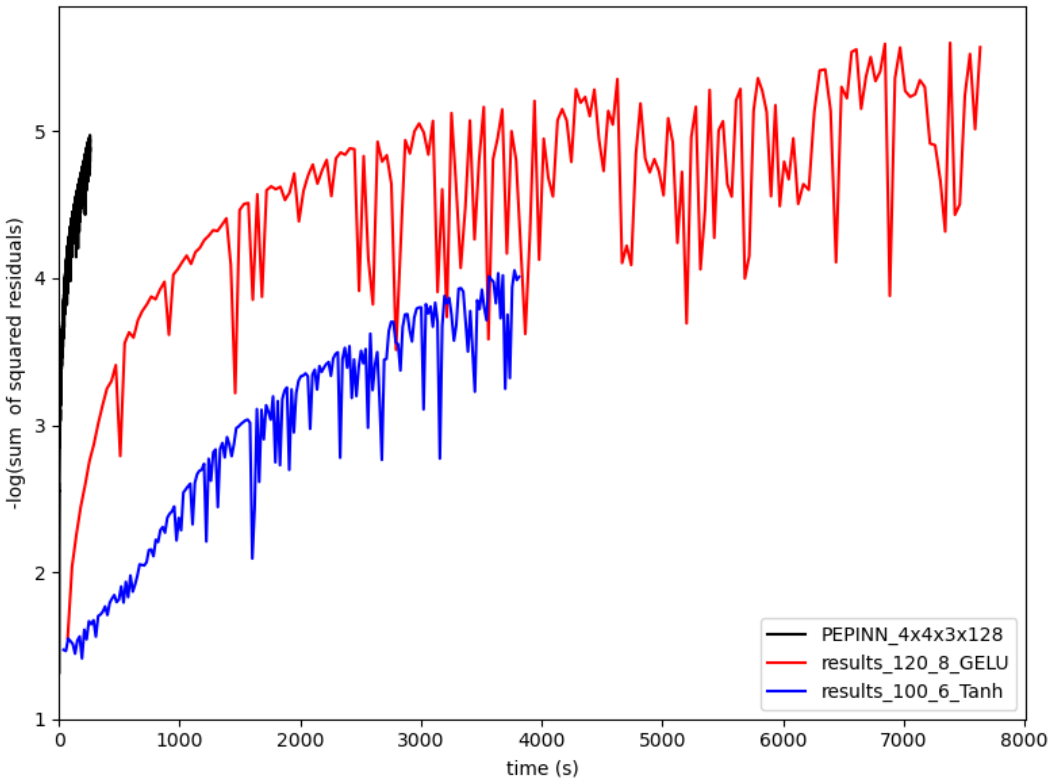

Figure 2.5: Comparison of the PEPINN with signature $4\times4\times3\times128$ with the selected baseline model and the MLP architecture used in RNN-DCT paper[62] when query point permutations of size $TYX = [16 \times 32 \times 32]$ is provided to all of the models for 20000 training iterations.

In figure 2.5 the residual of the PEPINN model with domain encoding of size $CX = 4$, $CY = 4$, $CT = 3$, $C = 128$ is compared with the residual of the selected baseline model as well as the MLP architecture used in RNN-DCT paper [62]. All the cases are trained for 20000 iterations where query points with size $[16 \times 32 \times 32]$ are provided to each model. The learning rate with a value of 0.0005 is provided to the Adam optimizer for all the cases. According to the results, the PEPINN takes 256s to perform 20k iterations with the negative log of the sum of squared residuals of 4.64 whereas the baseline model takes 7633s to reach the maximum value of 5.6 for residual loss and the model used in [62] takes 3813s which results in a solution with value 4.05 for residual metric. As it can be seen in the figure 2.5 the value of the negative log of squared residuals of the PEPINN model is slightly lower than that of

the selected baseline model. Yet using PEPINN results in a $29.8\times$ speed up compared to the baseline model and $14.9\times$ speed up compared to the baseline used in the RNN-DCT paper.

Another way of examining the performance of the PEPINN is to train the model with a higher number of query points. To that end, an 8 times larger set of query points is provided to the PEPINN, and its performance is compared to both baseline models which include the baseline model selected here, the one used in the RNN-DCT paper with the originally selected set of query points of size $[16 \times 32 \times 32]$. Figure 2.6 shows the performance of this model.

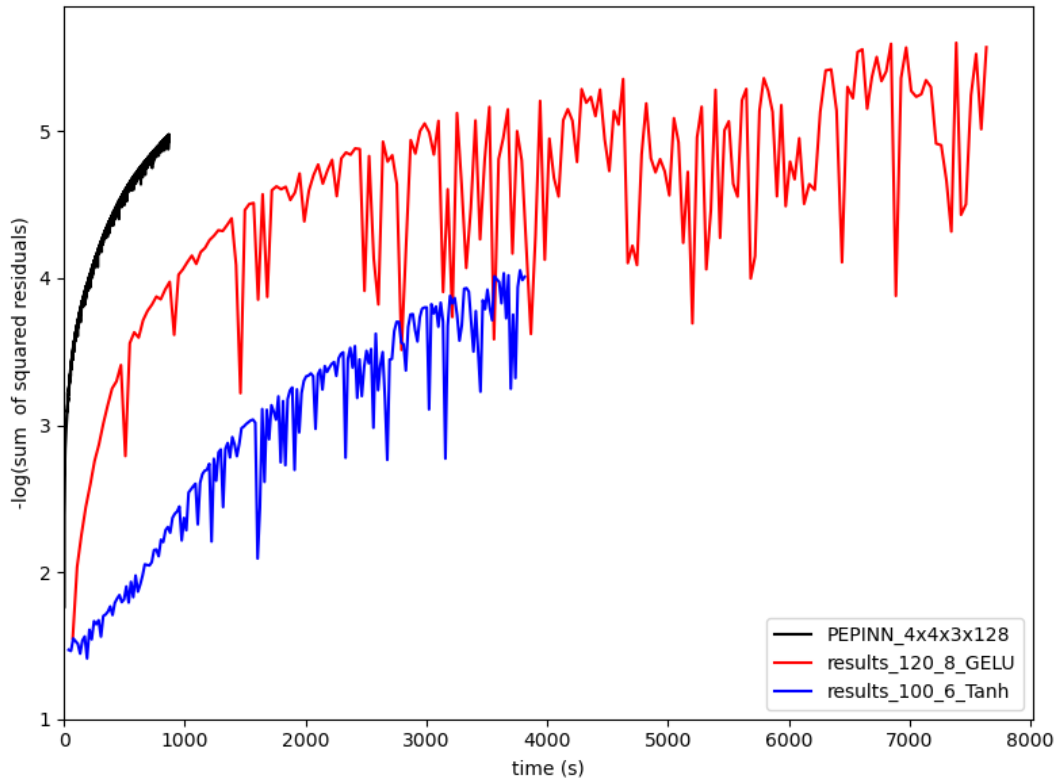

Figure 2.6: Comparison of the PEPINN with signature $4 \times 4 \times 3 \times 128$ with baseline model and the model used in [62] when a query points permutation of size $16 \times 32 \times 32$ is provided to selected baseline models and query points permutation of size $32 \times 64 \times 64$ is provided for the PEPINN model during 20000 iterations.

The PEPINN model takes 868s to perform 20k iterations with the negative log of the sum of squared residuals of 4.98. This model reaches similar residual values about 8.8

times faster compared to the selected baseline and 4.4 times faster than the baseline model used in RNN-DCT given an 8 times larger set of data points. Here for training the PEPINN model the learning rate is slightly increased to the value of 0.001 for faster convergence on larger data (The effect of the optimizer's learning rate will be discussed later).

The performance of the PEPINN can further be improved by scheduling the number of data points provided to the model during training. Motivated by [65] initially a smaller number of query points is provided to the model and the number of these points is increased gradually during the training of the PEPINN model. With this procedure, low-frequency solution representation under the problem constraints is learned by the model which in turn allows it to converge faster. Of course, a low number of query points would imply a higher residual calculation error at the initial stages of the training. The error in the calculation of the residual and thus the PEPINN error can then be minimized by gradually increasing the number of the query points. Providing a large number of query points decreases the error bounds for the calculation of the derivatives via the finite difference method. This procedure results in faster and more stable convergence of the PEPINN model which achieves lower loss values compared to when this procedure is not used with minimum loss in accuracy. The performance of this training procedure is presented in figures 2.7, 2.8. The PEPINN model presented in figure 2.7, finishes training in 191s and achieves a negative log of the sum of squared residuals value of 5.27 at 20k iterations. While training this model, initially a query point permutation of size $[5 \times 5 \times 5]$ is provided to the PEPINN model. The number of permutation points at each dimension is increased by one increment every 600 iterations for spatial coordinates, and 1200 iterations for temporal coordinates until the maximum permutation size of $16 \times 32 \times 32$ is reached. For this case, for instance, the model is trained with the maximum number of query points (i.e. query point permutation of size $16 \times 32 \times 32$) after the 16200^{th} training iteration is performed. Thus during 16201-20000 iterations, the model trains on the lowest error data permutation provided. This model achieves $40\times$ speed up compared to the selected baseline model while achieving about $20\times$ speed up compared to the vanilla PINN model used in the RNN-DCT paper. A similar improvement in results is obtained when a query point with a larger size is provided to the model. The model

presented in figure 2.8, takes 335s to train compared to 868s when the scheduled training method was not used and achieves a negative log of the sum of squared residuals value of 5.42 compared to the value 4.98 presented in figure 2.6 at 20k iterations. This indicates a significant improvement in both training time and the negative log of the sum of squared residuals achieved. While training this model, initially a query point permutation of size $[5 \times 5 \times 5]$ is provided to the PEPINN model. The number of permutation points at each dimension is increased by one increment every 300 iterations for spatial coordinates, and 600 iterations for temporal coordinates until the maximum permutation size of $32 \times 64 \times 64$ is reached. Comparing the training time of this model with the selected baseline a $22.8\times$ speed up is achieved.

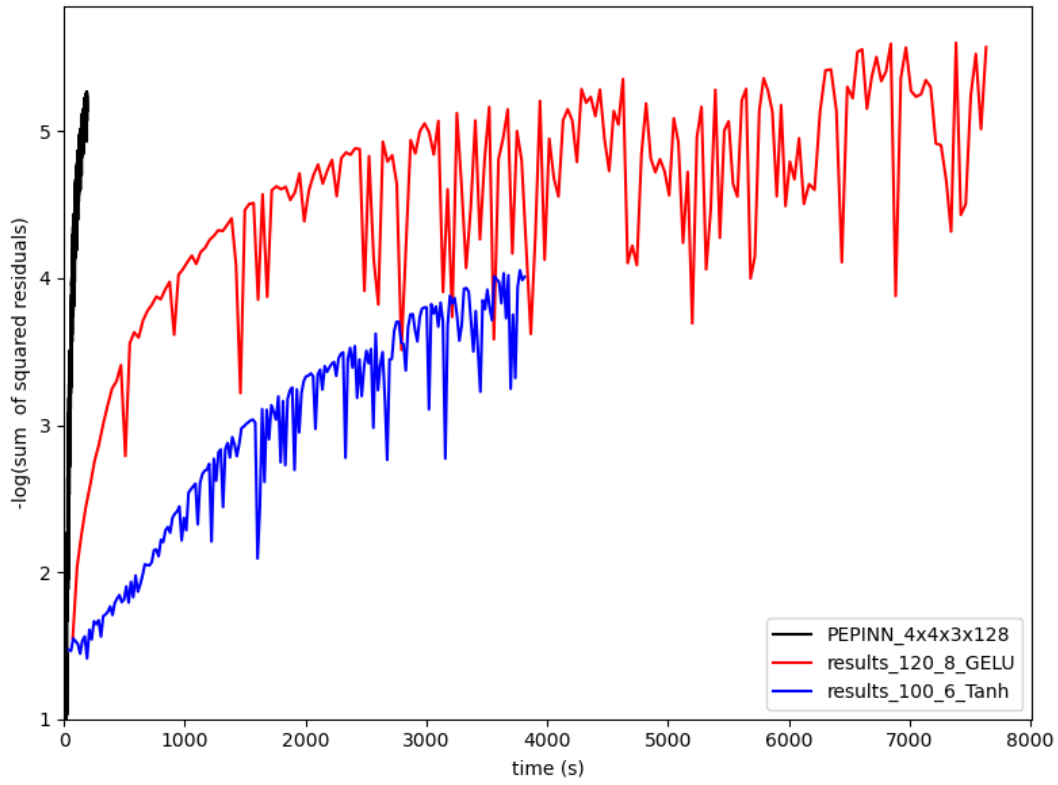

Figure 2.7: Comparison of the PEPINN with signature $4 \times 4 \times 3 \times 128$ with baseline model and the model used in [62] when a query points permutation of size $16 \times 32 \times 32$ is provided to baseline models and maximum query points permutation of size $16 \times 32 \times 32$ is provided for the PEPINN model during 20000 iterations.

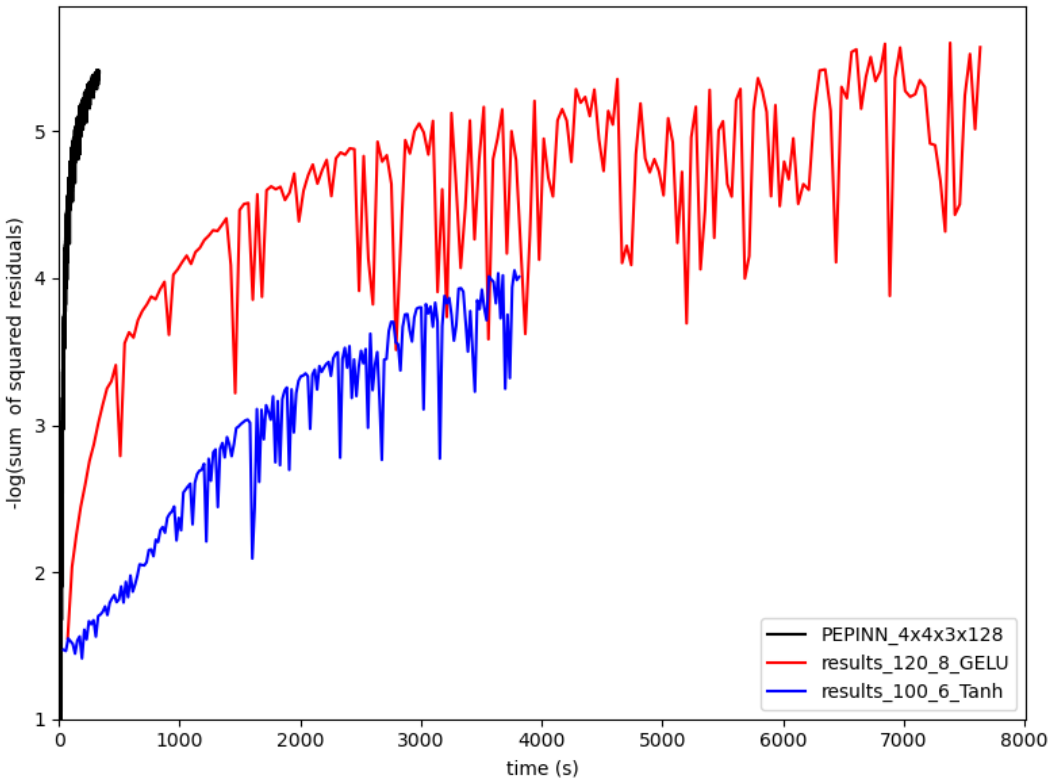

Figure 2.8: Comparison of the PEPINN with signature $4 \times 4 \times 3 \times 128$ with baseline model and the model used in [62] when a query points permutation of size $16 \times 32 \times 32$ is provided to baseline models and maximum query points permutation of size $32 \times 64 \times 64$ is provided for the PEPINN model during 20000 iterations.

One issue that needs to be considered is that the size of the query points directly affects the training and accuracy of the PEPINN model. As the Spatio-temporal domain in PEPINN is divided into subdomains represented by the dense parametric encoding weights, enough data needs to be presented to the model so that a viable latent representation on each subdomain is learned. This raises an issue in learning the parametric encoding in the PEPINN as the generated query points are required to lie on a uniform grid (assuming a Banach space for the Spatio-temporal domain). For a more general case, the number of domain subdivisions for a PEPINN might be larger than that of the query points supplied to the model. This poses an issue as some of the subdomains parameterized by the parametric encoding will not be trained to conform to the constraints imposed by the loss on the model which in turn will result in high

loss values for that solution. This phenomenon can also be intuitively observed. Let us assume that the domain encoding divides the space and time to N subdivisions along each principal direction thus obtaining a representation $\omega \in \mathbb{R}^{N\times N\times N\times C}$ in 2D space. If this model is trained by query points from a uniform grid of size $n \times n \times n$ where n « N then a large portion of the domain with the parametric encoding weights will not be affected thus the accuracy of the model will be low. One solution for this problem is to use query points that belong to a smaller bounding box in the domain (where this bounding box is presumably generated randomly). This remedy is not investigated here instead a domain encoding with fewer subdivisions is tested.

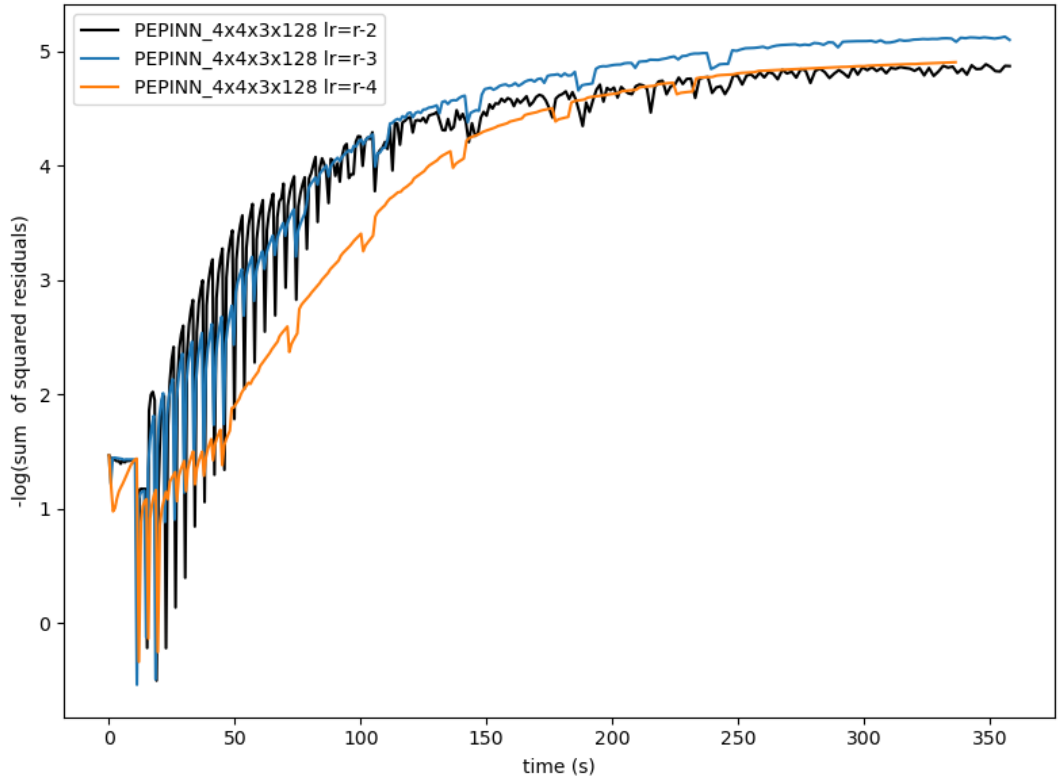

Figure 2.9: Training PEPINN for 30000 iteration using different learning rates in Adam [75] optimizer. As it can be seen the model training is not affected by the learning rate value.

Another factor to be considered is the training performance of the PEPINN model under different learning rates. Generally, the learning rate (lr) is set as an optimizer hyper-parameter for DL architectures and the training performance of those models

is greatly affected by how the learning rate is set up during training. A model eliminating such a hyper-parameter would make the design and training of those models more efficient and make the training more robust by eliminating the process of hyper-parameter search. Figure 2.9 shows the training performance of the model provided three different lr values to the Adam optimizer with default Beta values.

As it can be observed in the figure 2.9 PEPINN model is robust under different learning rate values and converges to almost the same result. Even though they converge to almost the same loss value, the residual of the lr = 10^{-4} (case 3) seems the most stable among the three whereas the training loss behavior under lr = 10^{-2} (case 1) has the fastest settling time. Yet the lowest residual value is achieved when the learning rate is set to 10^{-3} (case 2).

So far it was shown that the PEPINN architecture can achieve about 40 times the speed-up compared to the selected baseline model. In terms of memory usage, the PEPINN can be trained on query points of size 3,579,096 on an RTX 1080 Ti GPU while the baseline model with automatic differentiation can accept only 19456 query points. It allows about $183\times$ larger set of query points to be trained on PEPINN.

Another important aspect of PEPINN that must be investigated is its accuracy compared to the baseline model. To that end, the mean squared error (MSE) of the model at different time points is calculated for PEPINN as well as the baseline model selected in this thesis and the baseline model tested in [62]. For error calculations, at each time a uniform grid of size (256×256) is generated and fed to the solution model and its result is compared to the analytic solution of the Taylor-Green Vortex. Figures 2.10, 2.11, 2.12 compare the accuracy of PEPINN vs vanilla PINNs with 100/120 neurons and 6/8 layers subjected to Tanh/GELU nonlinear activation functions. For the calculation of the accuracy of the vanilla PINN architectures, the models saved at the training termination are used. From these figures, it can be observed that in most cases the vanilla PINN model trained on a set of query points of size $[32 \times 64 \times 64]$ yields lower mean squared errors for the velocity components. For the pressure accuracy on the other hand the PEPINN model performs worst compared to the vanilla PINN after the time 0.8s. Generally, in initial condition-based solutions, the error increases as the solution proceed in time. This phenomenon is also observed in solu-

tions based on vanilla PINN. Yet except for pressure the error of the PEPINN model stays the same or even decreases as time increases. For the pressure accuracy on the other hand, at time 0.5s and the interval closer to the termination time a surge in error is observed. Both time t = 0.5s and t = 1.0 s are where each subdomain divided by the local parametric encoding weights share a common face. As the local weights in these faces are shared between two subdomains, they require a longer time to be optimized. The same phenomenon can be observed in faces where spatial subdomains share a common face. Indeed it can be observed in figures A.17 and A.16 where the highest error is observed at the locations where different subdomains share a common face.

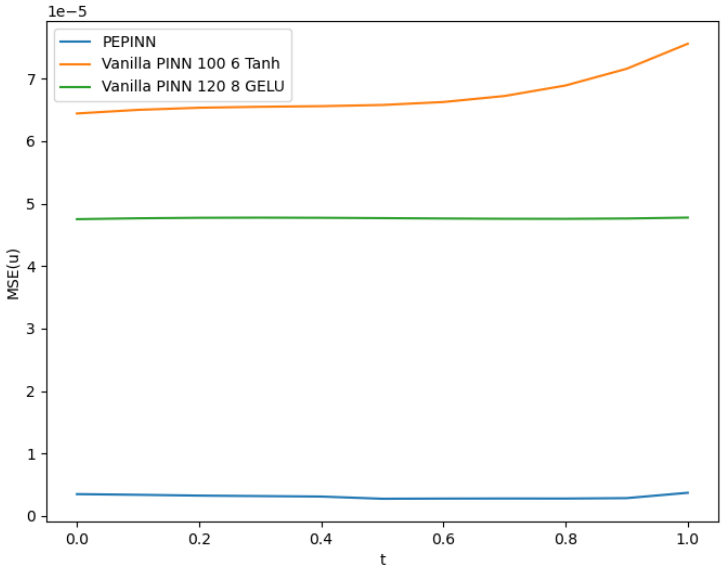

Figure 2.10: Accuracy of PEPINN vs Vanilla PINN models with 100/120 neurons and 6/8 layers with Tanh/GELU for x component of velocity.

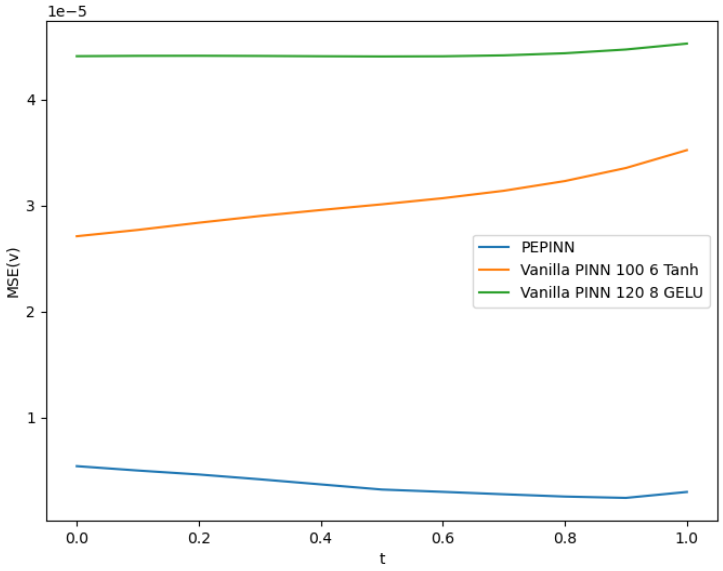

Figure 2.11: Accuracy of PEPINN vs Vanilla PINN models with 100/120 neurons and 6/8 layers with Tanh/GELU for y component of velocity.

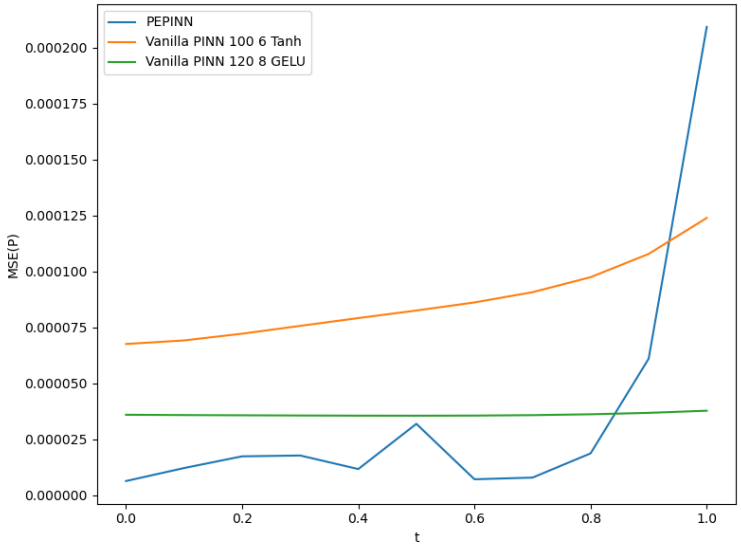

Figure 2.12: Accuracy of PEPINN vs Vanilla PINN models with 100/120 neurons and 6/8 layers with Tanh/GELU for pressure.

CHAPTER 3

SPEED-UP OF SOLUTION OF STEADY-STATE PDES

3.1 Introduction

There is a rapid growth in the application of Deep Neural networks (DN) for modeling physical simulations. However, these models lack the robustness, reliability, and accuracy of discretization-based (DB) solvers. On the other hand, there is a substantial gap between the simulation time of DN models compared to DB solvers. In this section, the effect of utilizing the predicted solution variables of PDE-based problems obtained through trained DN models as initial conditions for DB solvers, with a focus on the reduction in time and number of iterations is investigated. The hypothesis presented in this chapter is tested for the solution of the Navier-Stokes equations. More specifically, the model presented here is trained on a modest dataset of 1400 samples, created via the open source physics solver OpenFOAM , for a collection of two-dimensional, steady, subsonic, in-compressible flow solutions around randomly generated airfoils at a randomly selected angle of attack and Reynolds numbers. Here the Spalart-Allmaras turbulence model was used for all solutions and a custom DN model was trained on the data to predict the velocity, pressure, ν_t, and $\tilde{\nu}$ fields which are then interpolated to the computational mesh of the CFD solver only once as its initial condition. This hypothesis is motivated by the observation that if a close enough solution is provided to the solver as the restart condition then the number of required iterations for convergence of the solution decreases. This process previously was applicable only in a restricted number of cases. For instance, this approach was used in CFD problems when the solution for an angle of attack close to that of the test case on the same mesh was available, or when the solver has performed a number of iterations but has not yet converged and its states were saved. For both of these

situations, providing the values of their flow variables as initial and boundary conditions to the problem would generally result in a speed-up of the solution. Here it is assumed that the applicability of this approach can be extended with the help of the method proposed in this chapter.

3.2 Problem Description

Machine learning (ML) has become one of the most popular research directions, which yields astonishing results in many areas such as computer vision, natural language processing, decision making, and synthetic data generation. One of these research areas, where deep learning is slowly being integrated to, is the physical simulations domain. Albeit most of the methods that utilize ML models trade accuracy for speed and are mostly used in situations where fast, low fidelity but realistic-looking simulations are required. On the other side of this spectrum lies the de-facto method for physical simulations which are based on the discretization of both the domain and the simulation's governing equations. These methods (e.g., finite element and finite volume-based solvers) are relatively more accurate but require significant time to set up and produce solutions. In this chapter, an alternative to the methods that are used to speed up the performance of steady-state Navier-Stokes solutions in subsonic, incompressible regions is presented. Specifically, the performance of the solution of Reynolds-Averaged Navier-Stokes (RANS) equations utilizing the Spalart-Allmaras turbulence model to solve the flow around two-dimensional (2D) airfoils is investigated. Here a DL model is trained on a small dataset of randomly generated samples. This model then is used to predict the solutions for a set of test cases that were unseen during training. The predicted solution fields then are transferred to the computational mesh of the same problem cases and will be used as the steady-state solution's initial condition.

To that end, a training and a test set are generated. For the training set, a number of randomly generated airfoils are created via Class/Shape Transformation (CST) method. These airfoils then are used to generate a computational mesh to be utilized in the finite volume-based solver configured to solve the Spalart-Allmaras turbulence model. A number of query points are then extracted from solution fields in the com-

putational mesh to obtain the solution array that the DL-based model will be trained on. After the model is trained, its inference results for new cases will be transferred to the computational mesh as an initial condition of the corresponding solution field in cell centers for internal meshes. The solutions at the boundary faces are also transferred to the face centers to construct the proper boundary conditions. These cases then are solved in the same manner given both the original initial conditions as well as the inferred initial conditions and their solution time and the number of iterations are compared.

3.3 Airfoil Generation

Any airfoil shape can be represented using Class/Shape Transformation (CST) methodology [88]. CST is a decoder that utilizes Bernstein polynomials and is based on two arrays of coefficients. The coefficients in arrays represent the upper and lower surfaces/curves of the airfoils. As these coefficients directly correspond to the shape of the airfoil, a large variety of the geometries can be generated by sampling these coefficients from a distribution with a large variance. Equations for these curves in CST are given as:

$$\zeta_U(\psi) = C(\psi)S_U(\psi) + \psi\Delta\zeta_U \tag{3.1a}$$

$$\zeta_L(\psi) = C(\psi)S_L(\psi) + \psi\Delta\zeta_L \tag{3.1b}$$

Where $\psi = x/c$, $\zeta = z/c$. In equation 3.1 subscripts U and L represent the upper and lower curves of the airfoil.

A 2D airfoil geometry can be constructed in CST via the following class function:

$$C_{1.0}^{0.5} = \psi^{0.5}.(1-\psi)^{1.0} \tag{3.2}$$

In Equation 3.2, coefficients of 0.5 and 1.0 are constants to produce 2D airfoil shape in CST. The final shape function to define the specific shape for the upper and lower

curves is defined in 3.3.

$$S_k(\psi) = \sum_{i}^{N_k} A_k.S(\psi, i), k = U, L \tag{3.3}$$

In Equation 3.3, A is the weight input coefficient and decisive parameter in CST method. S is called the component shape function. The component shape function is represented by a Bernstein polynomial 3.4 of degree N.

$$S(\psi, i) = K_i^N.\psi^i.(1-\psi)^{N-i} \tag{3.4}$$

Here the binomial coefficient (K) in equation 3.4 is defined in equation 3.5. These coefficients can be easily computed by constructing the Pascal's triangle where the binomial coefficient K would correspond to the elements in the N^{th} row of the Pascal triangle.

$$K_i^N = \frac{N!}{i!(N-i)!} \tag{3.5}$$

The final equation representing the upper and lower curves are obtained by combining the previous equations:

$$\zeta_U = \psi^{0.5}.(1-\psi)^{1.0} \sum_{i=0}^{N_U} [A_U(i).\frac{N_U!}{i!(N_U-i)!}.\psi^i.(1-\psi)^{N_U-i}] + \psi.\Delta\zeta_U \tag{3.6a}$$

$$\zeta_L = \psi^{0.5}.(1-\psi)^{1.0} \sum_{i=0}^{N_L} [A_L(i).\frac{N_L!}{i!(N_L-i)!}.\psi^i.(1-\psi)^{N_L-i}] + \psi.\Delta\zeta_L \tag{3.6b}$$

In equation 3.6, the first and last weighting coefficients for the upper and lower surface (A_U and A_L) are known to determine the leading edge radius and the trailing edge angle, respectively. The rest of the weighting coefficients vary the thickness distribution in general [89]. Since the main objective of this study is to train a neural network for learning the correlation between the variation in shapes and their corresponding aerodynamic footprint, a wide range of airfoil shapes based on randomly selected values

for the weighting coefficients (AU and AL), and for the order of Bernstein polynomials (N_L and N_U in equation 3.6) as well as for the exponents of the class shape functions (equation 3.2) are generated.

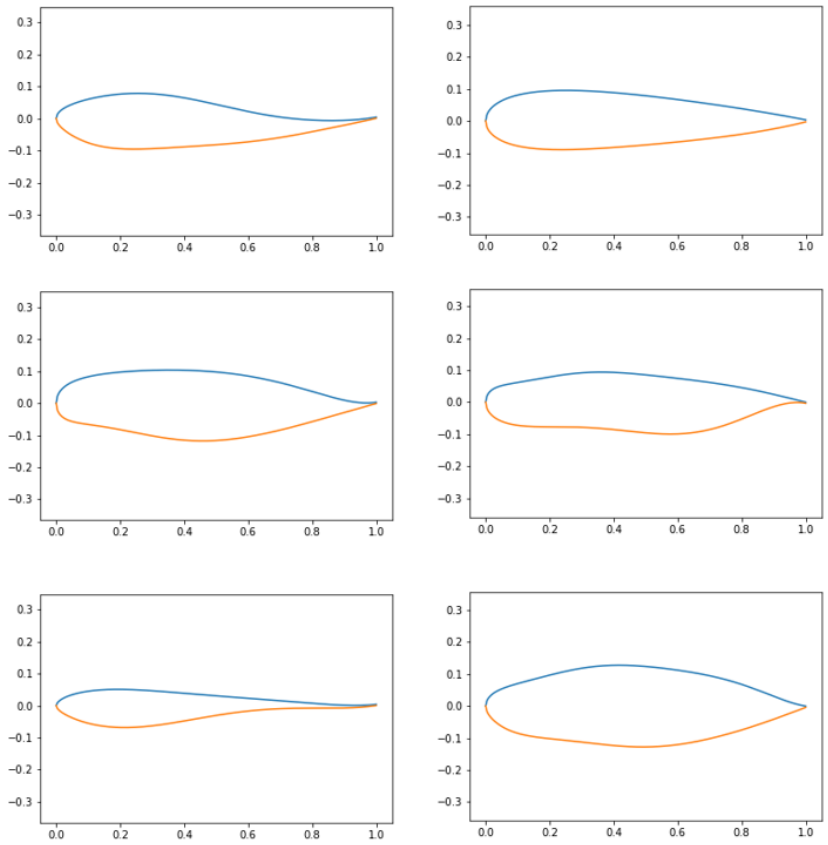

Figure 3.1: First 6 airfoils out of 1000 generated airfoils using CST method. It can be observed that even in this small subset of the generated airfoils some are similar to the airfoils used in practical applications while others are just airfoil-like shapes that might be impractical to use in engineering applications. Yet even in this limited number of cases a large range of variations in shape is present.

The random selection process is performed using a uniform distribution in order to sample a stochastic range of shapes. The order of Bernstein polynomials are randomly selected from integers between 2 and 10, and the class function exponents are selected from the ranges 0.45 to 0.55 and 0.95 to 1, instead of using constant values of 0.5 and 1. The first weighting function that determines the leading edge radius is selected within a range of 0.01 to 0.05, and the last weighting function for the trailing

edge angle is selected from integers between -10 to 20. The rest of the weighting coefficients are selected within a range of -0.1 to 0.6 for the upper curve and -0.6 to 0.1 for the lower curve. The upper and lower boundaries for these ranges were selected such that the upper and lower curves will not cross and intersect with each other (except at leading and trailing vertices). The selection of these ranges allowed us to create a wide range of airfoil shapes, some of which actually can be "impractical" for aerodynamic applications. However, this wide range of airfoils would provide large-enough shape variations for the neural network to establish a good mapping between the variations in shape and the flow field around it. As a result, the "impractical" airfoils were intentionally not excluded from the training samples.

Using the methodology described above a set of 1000 airfoil shapes was generated and analyzed using OpenFoam software which is discussed in the subsequent sections. The first 6 generated airfoils are presented in figure 3.1.

3.4 Mesh Generation

The finite volume, finite element, and finite difference solvers require the problem domain to be discretized and the partial differential equation to be modified accordingly. In this part of the thesis, the OpenFOAM solver was used to obtain the results of RANS equations on steady, two-dimensional problems investigating the flow around the airfoils. In order to set up each problem case, a discretization of the problem domain with proper boundary and initial conditions is required to be specified. This discretization is referred to as either line mesh (for the discretization of curves), shell mesh (for the discretization of surfaces bounded by its surrounding curves), or volume mesh (for the discretization of the spatial volume surrounded by its bounding surfaces) in the literature. For this thesis, two open source software GMSH and blockMesh are investigated for the generation of the computational meshes and the latter is selected to minimize the time spent constructing the mesh around airfoils for each training or test problem. The generated meshes are bounded by a far field, large enough not to be affected by the change in flow properties around the airfoil shapes. In this study, the size of the airfoil and the far field is kept constant as is shown in figure 3.6 and the Reynolds Number is changed via variations in velocity, density,

and viscosity. Also instead of the rectangular far-field, here a combination of a half circle and rectangle is used to obtain a more general domain representation (see 3.7 for the justification of why such a far field was selected).

The blockMesh software generates structured meshes by mapping a hexagonal domain to a shape with eight points and twelve curves connecting each point/vertex to their three neighboring points (These spatial entities are referred to as a geometric block). In order for the mapping to be proper, the points and the curves must be ordered complying with the setting represented in figure 3.2. Each block has a local, right-handed coordinate system (x_1, x_2, x_3). A right-handed 3 dimensional coordinate system with axes (x_1, x_2, x_3) can be defined by equation 3.7.

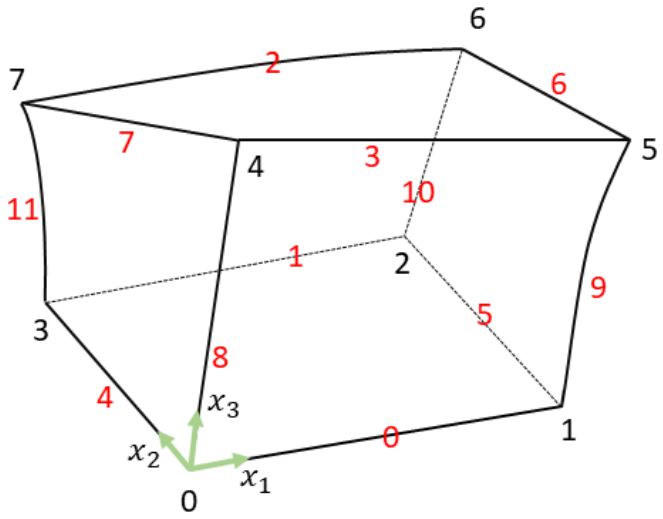

Figure 3.2: Ordering for mapping from a hexagon to a blockMesh block. The Black integers represent vertex numbering and the Red integers show the edge numbers. A block is represented in the code by ordering the vertices in accordance with the numbering presented above. The edges then are automatically assigned between the vertices in the presented order. Note that the edges represent curves rather than straight lines and thus are arc-length parameterized.

$$\bar{x}_3^T(\tilde{x}_1\bar{x}_2) > 0. \tag{3.7}$$

With $\bar{x}_i$ the column vector representation of axis i and $\tilde{x}_j$ as the skew-symmetric configuration of the axis j. More intuitively, a right-handed 3-dimensional coordinate system is defined such that to an observer looking down the x_3 axis, through the coordinate system origin O, the arc from a point on the x_1 axis to a point on the x_2 axis is in a clockwise sense. The ordering of the points in a geometric block is to be defined such that the desired local coordinate system and the coordinate system presented in 3.2 coincide (assuming that the 12 bounding curves are the same). Figure 3.3 represents the block sectors used for meshing the domain around the airfoil.

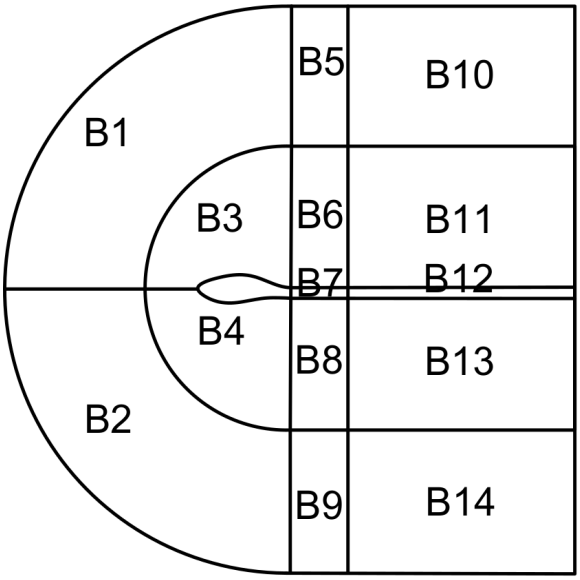

Figure 3.3: Top view of the blockMesh settings for generation of mesh around the airfoil. The blocks in this setting are configured such that the x_3 direction is defined span-wise. The mesh in this direction corresponds to extrude mesh of constant thickness.

Table 3.1: Ranges for initial condition values of variables of RANS equations. Here the turbulence viscosity is set to be in accordance with OpenFOAM defaults for faster convergence.

Variable	Unit	Range	Description
U	$\frac{m}{s}$	$[20, 200]$	Magnitude of the velocity
α	degrees	$[-10, 10]$	Angle of attack
P	$\frac{kg}{ms^2}$	0	Pressure
ν_t	$\frac{m^2}{s}$	calculated from $\tilde{\nu}$	turbulence viscosity
$\tilde{\nu}$	$\frac{m^2}{s}$	$[0.13, 0.15]$	modified turbulence viscosity

The inside of the airfoil is treated as a hole and no blocks are defined for it. Each block is created such that the x_1 and x_2 directions lie on the curves of the block presented in 3.3 and the x_3 direction is perpendicular to the plane facing towards the viewer. The final mesh is presented in 3.5, 3.4. Far-field and airfoil chord sizes, as well as boundary annotations, are presented in figure 3.6.

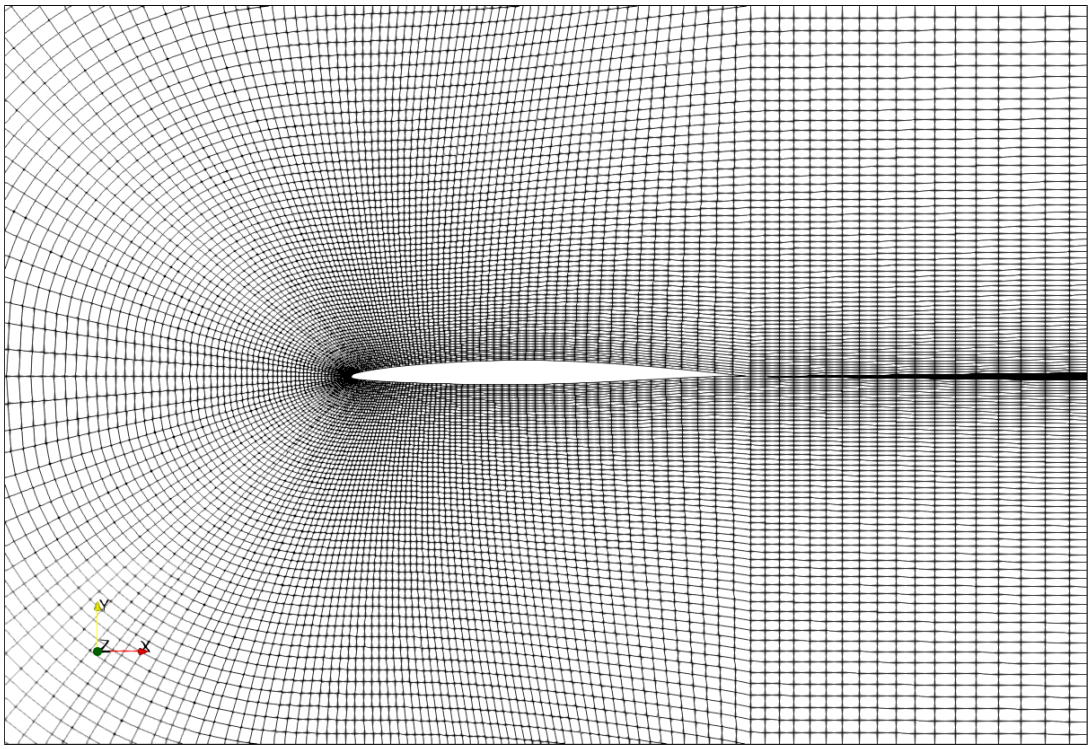

Figure 3.4: Generated mesh around an arbitrary airfoil. The spacing of edge meshes over the airfoil in blocks B3 and B4 are non-uniform. They are selected such that more points are assigned to the nose with a shorter distance. Also, the spacing between the edge divisions increases geometrically as the airfoil upper and lower curves are traced from the leading edge to the trailing edges. This allows for a smoother transition in meshes from the nose radius to the arc created in front of the leading edge.

The boundaries with annotation inlet, outlet, top, and bottom were assigned a patch-type boundary condition. In contrast, the boundaries with annotation wall were set to have nutUSpaldingWallFunction type boundary condition for nut variable, fixed value for the nuTilda, zeroGradient for pressure, and noSlip for the velocity. The rest of the boundary conditions are set to freestream. Initial conditions are sampled from a uniform distribution in the ranges provided in Table 3.1 with the exception of ν_t

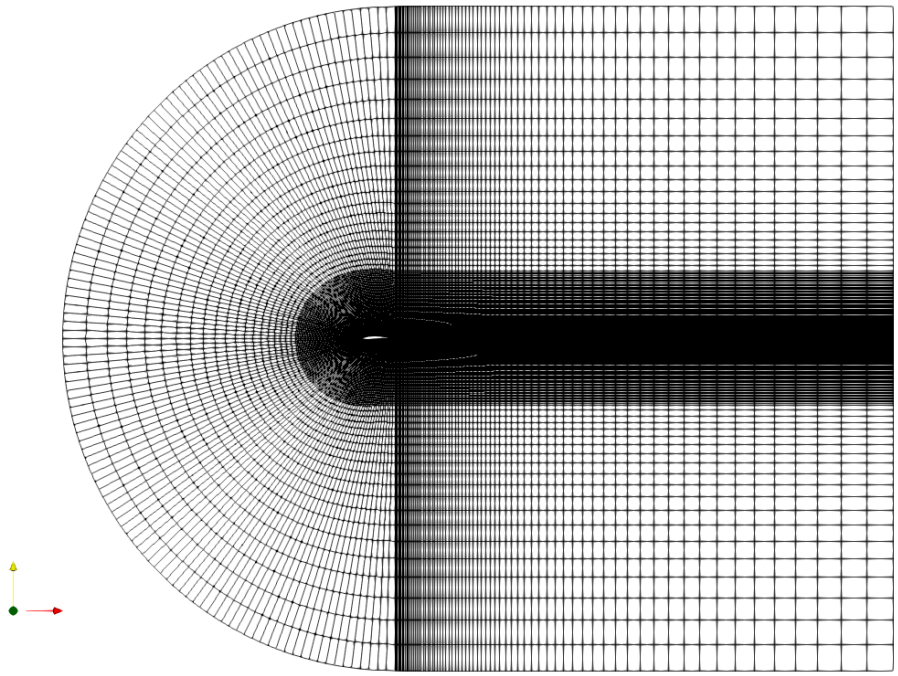

Figure 3.5: Generated mesh around an arbitrary airfoil.

which is calculated from the value of $\tilde{\nu}$ for each case. Here the turbulence viscosity is set to be in accordance with OpenFOAM defaults for faster convergence of original cases. For more accurate results $\tilde{\nu}$ should be set to a range between $3\nu_\infty - 5\nu_\infty$.

The mesh around the airfoil is generated so that the first cell size is equal to 300 y^+. This size is estimated by the $\Delta y = \sqrt{74}Re^{\frac{-13}{14}}y^+$. As the internal arc around the airfoil is generated by a shape offset of value 0.8, the size of the first mesh layer can be easily calculated. The reason for selecting a large y^+ is that a wall function is used on the airfoil thus permitting a larger mesh size to capture the boundary layer. In order to limit the solution time a fairly coarse mesh is used where the number of mesh elements in the x-direction are 120 (over the airfoil curves), 40 (in sector B7), 80 (in sectors B12), and in the y direction are 80 (from airfoil LE to the inner offset), 60 (from inner arc to outer arc), 8 (at the TE of the airfoil in sectors B7 and B12). The mesh expansion in sections B3 and B4 are controlled by the first cell size, whereas the expansions of the other sectors is selected to have a smooth transition between joining sectors.

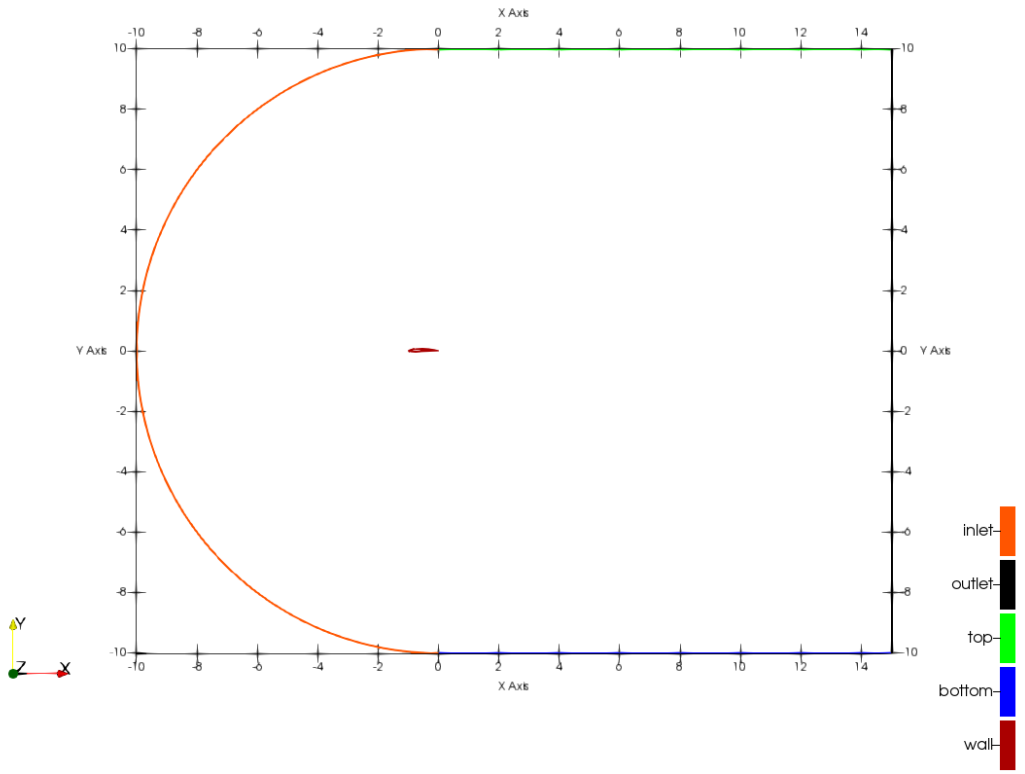

Figure 3.6: Generated mesh around an arbitrary airfoil.

3.5 OpenFOAM Solver Setup

For the solution of Reynold-Averaged Navier-Stokes (RANS) equations around two-dimensional (2D) airfoils, the Spalart-Allmaras turbulence model was utilized. The OpenFOAM software implementation of this model is used for all the solutions. The Spalart-Allmaras is a one equation model based on a modified turbulence viscosity $\tilde{\nu}$. The modified version of this equation is presented in equation 3.8. In the original Spalart-Allmaras equation there exists another term (denoted by f_{t2}) which is not added in the OpenFOAM implementation of this equation. As a result, this term is ignored in 3.8.

$$\frac{D}{Dt}(\rho\tilde{\nu}) = \nabla.(\rho D_{\tilde{\nu}}\tilde{\nu}) + \frac{C_{b2}}{\sigma_{\nu_t}}\rho|\nabla\tilde{\nu}|^2 + C_{b1}\rho\tilde{S}\tilde{\nu} - (C_{w1}f_w)\rho\frac{\tilde{\nu}^2}{\tilde{d}^2} + S_{\tilde{\nu}} \tag{3.8}$$

In equation 3.8, $\tilde{\nu}$ represents the viscosity like variable whereas the $C_{b2}, \sigma_{\nu_t}, C_{b1}, C_{w1}$

are constants. Here, the turbulence viscosity "ν_t" is obtained by $\nu_t = \tilde{\nu} f_{v1}$ where the derivation of function f_{v1} is presented in equation 3.9.

$$f_{v1} = \frac{\left(\frac{\tilde{\nu}}{\nu}\right)^3}{\left(\frac{\tilde{\nu}}{\nu}\right)^3 + C_{v1}^3} \tag{3.9}$$

For all the solutions the default values of the OpenFOAM implementation of Spalart-Allmaras were used. In addition to the constants used for the Spalart-Allmaras equation, the density is set to 1.225 $\frac{kg}{m^3}$ and the kinematic viscosity is set to $10^{-5}\frac{m^2}{s}$ Also, the default finite volume scheme used in the airFoil2D case in simpleFOAM is used for the solution of RANS equations.

3.6 Trainable Model

In order to learn a mapping from the initial conditions and the domain properties to the solutions of RANS equations a CNN-based encoder, decoder pair model is utilized. This model is a modified version of the UNet [49] architecture. The basic architecture for this model is presented in the Figure 3.7 and Table 3.2. In Figure 3.7 a specific encoding is used to indicate different building blocks of the model with the encoding of type CCXKXSXPXCX and DCXKXSXPXCX to represent different model blocks. The initial letter in the encoding indicates if the block is a convolutional block (C) or a transposed convolution block (D). The rest of the letters and numbers describe the kernel shape where the number after letters C, K, S, and P represent channel size, 2D kernel size, strides, and padding. The first integer after the letter C represents the input channels and the second number proceeding with the same letter C indicates the output channel size. Each single digit number after letters K, S, and P represents a network layer For instance encoding CC4K33S11P11C32 means that the block is a 2D convolution block that accepts 4 channels and outputs a result with 32 channels. It consists of two layers with encoding CC4K3S1P1C32 and CC32K3S1P1C32. Each layer has a kernel size (3,3), stride (1,1), and padding (1,1). After each layer throughout the network, the Leaky-ReLU activation function with a slope of 0.1 is used. This model is implemented in Pytorch and trained using its automatic differentiation.

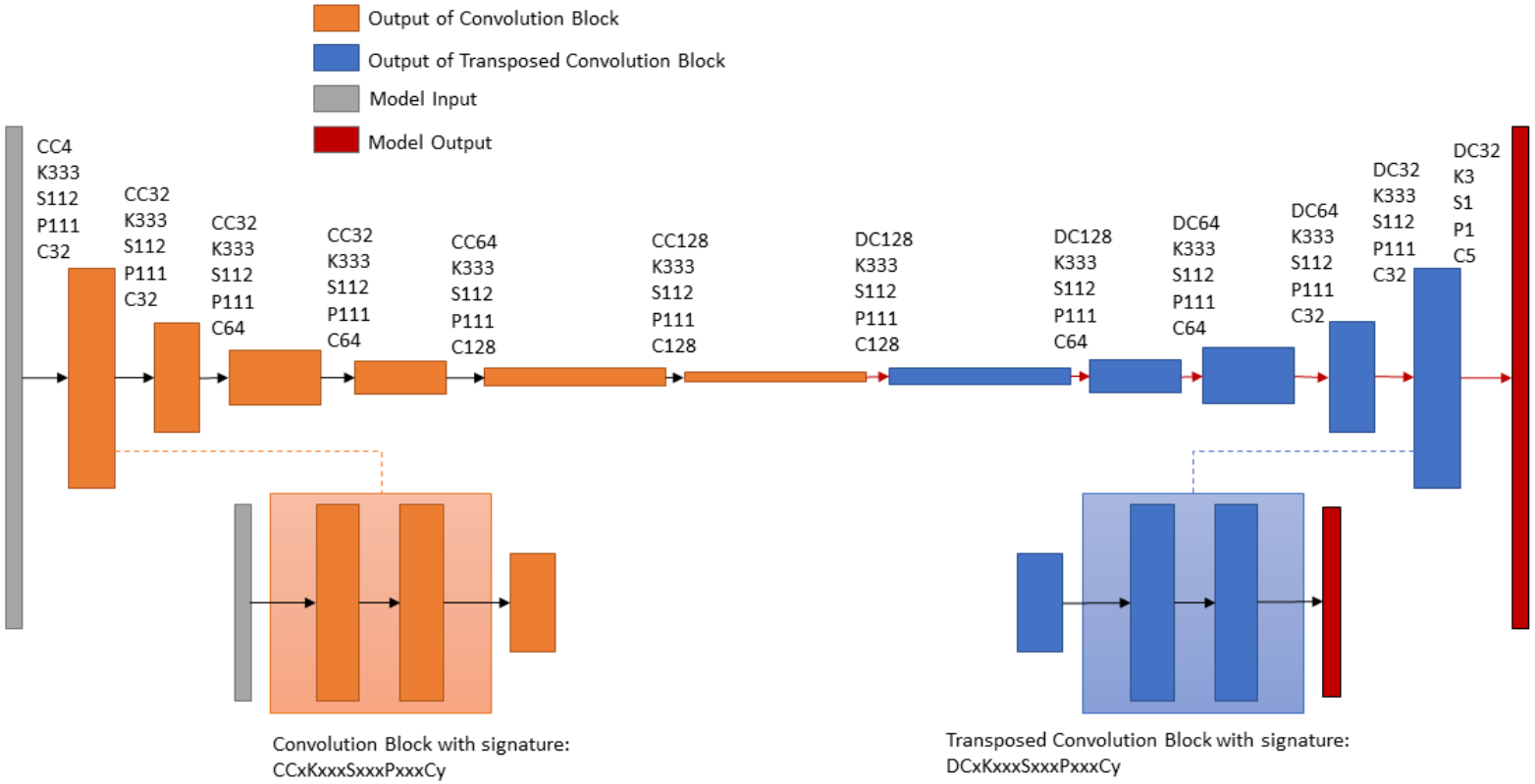

Figure 3.7: Trainable model for learning a mapping between the inputs of RANS equations to solver outputs after convergence.

Table 3.2: Modified UNet Encoder and decoder model architectures for learning the solution of RANS equations for incompressible, subsonic, steady state flow around 2D airfoils. (left) Encoder architecture building blocks. (right) Decoder Architecture building blocks.

Encoder Layers	Act.	Stride	Padding	Output Shape	Decoder Layers	Act.	Stride	Padding	Output Shape
-	-	-	-	4 x 256 x 256	-	-	-	-	128 x 8 x 8
Conv3x3	LReLU	1x1	1x1	32 x 256 x 256	DConv3x3	LReLU	1x1	1x1	128 x 8 x 8
Conv3x3	LReLU	1x1	1x1	32 x 256 x 256	DConv3x3	LReLU	1x1	1x1	128 x 8 x 8
Conv3x3	LReLU	2x2	1x1	32 x 128 x 128	DConv3x3	LReLU	2x2	1x1	128 x 16 x 16
Conv3x3	LReLU	1x1	1x1	32 x 128 x 128	DConv3x3	LReLU	1x1	1x1	128 x 16 x 16
Conv3x3	LReLU	1x1	1x1	32 x 128 x 128	DConv3x3	LReLU	1x1	1x1	128 x 16 x 16
Conv3x3	LReLU	2x2	1x1	64 x 64 x 64	DConv3x3	LReLU	2x2	1x1	64 x 32 x 32
Conv3x3	LReLU	1x1	1x1	64 x 64 x 64	DConv3x3	LReLU	1x1	1x1	64 x 32 x 32
Conv3x3	LReLU	1x1	1x1	64 x 64 x 64	DConv3x3	LReLU	1x1	1x1	64 x 32 x 32
Conv3x3	LReLU	2x2	1x1	64 x 32 x 32	DConv3x3	LReLU	2x2	1x1	64 x 64 x 64
Conv3x3	LReLU	1x1	1x1	64 x 32 x 32	DConv3x3	LReLU	1x1	1x1	64 x 64 x 64
Conv3x3	LReLU	1x1	1x1	64 x 32 x 32	DConv3x3	LReLU	1x1	1x1	64 x 64 x 64
Conv3x3	LReLU	2x2	1x1	128 x 16 x 16	DConv3x3	LReLU	2x2	1x1	32 x 128 x 128
Conv3x3	LReLU	1x1	1x1	128 x 16 x 16	DConv3x3	LReLU	1x1	1x1	32 x 128 x 128
Conv3x3	LReLU	1x1	1x1	128 x 16 x 16	DConv3x3	LReLU	1x1	1x1	32 x 128 x 128
Conv3x3	LReLU	2x2	1x1	128 x 8 x 8	DConv3x3	LReLU	2x2	1x1	32 x 256 x 256
-	-	-	-	-	Conv3x3	LReLU	1x1	1x1	5 x 256 x 256

3.7 Model Inputs and Outputs

The model accepts an array of size $\mathbb{R}^{N\times 4\times 256\times 256}$ and returns an output of size $\mathbb{R}^{N\times 5\times 256\times 256}$ with N representing the patch size. Treating these four-dimensional arrays as a stack of N images with shapes $\mathbb{R}^{N\times \text{feature size}\times \text{height}\times \text{width}}$ one can describe the underlying elements of both inputs and outputs which will be provided to the model. Each element in height and width of input and output describes a quantity measured at a specific position on a uniform grid. Each measured quantity is stored in the feature vector of the input or output. The input of the model is created by concatenation of the airfoil shape representation and the initial conditions at the same query points that were used to obtain the shape representation. Figure 3.8 shows the airfoil shape representation which for simplicity will be referred to airfoil mask.

The airfoil mask is created by querying if each point in a uniform 256x256 grid in the bounding box [(-1.2, 0.35), (0.2, -0.35)] lies outside the airfoil or not (It is assumed that the airfoil spans the x-axis from -1 to 0). For a continuous setting, the points

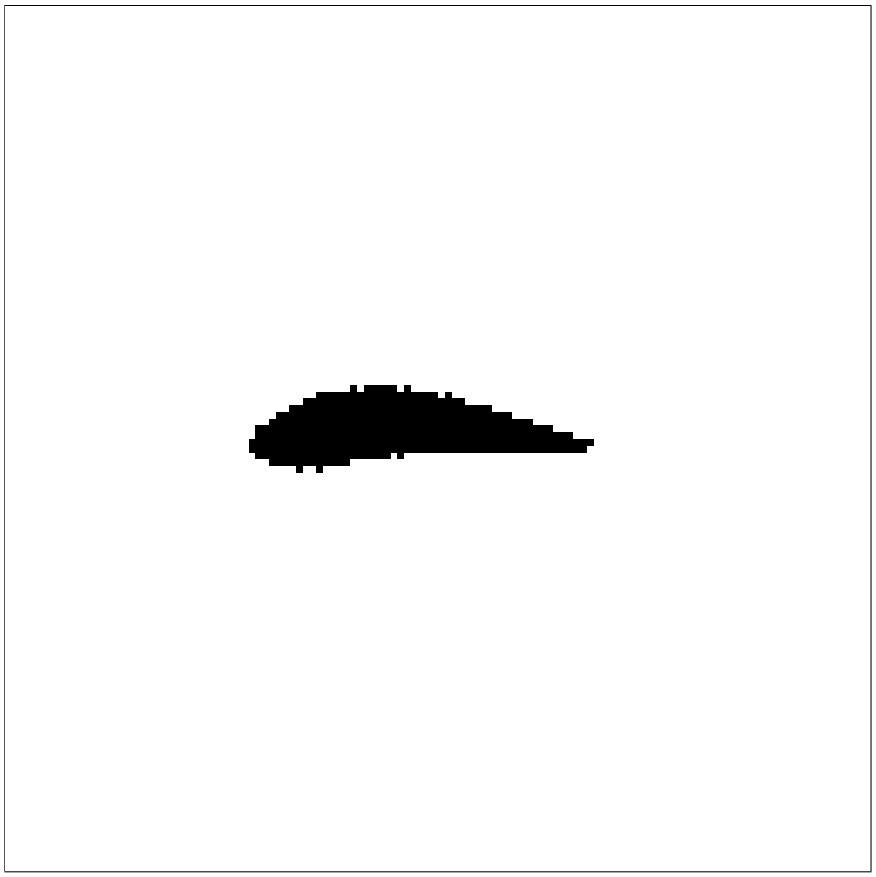

Figure 3.8: Uniform grid representation of airfoils which is referred to as airfoil mask in the text. The image is in grey scale where 1 is mapped to white color and 0.0 is mapped to black color.

outside the airfoil are given a value of 1.0 and the elements inside the airfoil are set to 0.0. On the other hand, a discretized setting requires the calculation of the common area between each pixel and the domain covered by the airfoil. If this area is larger than half of the area of that pixel its value is set to zero otherwise its value will remain 1.0. The airfoil map used in the inputs is created via the discretized setting. The first feature in the feature vector of the input is the airfoil mask and the other features are two velocity components and $\tilde{\nu}$. The input configuration is presented in figure A.12.

Here the reference outputs for training are not queried from a uniform grid. These points are obtained from a different position permutation. This is done because of the fact that a large portion of the change in solution variables happens in the vicinity of the airfoil and the deviation from the initial conditions decreases as the query points

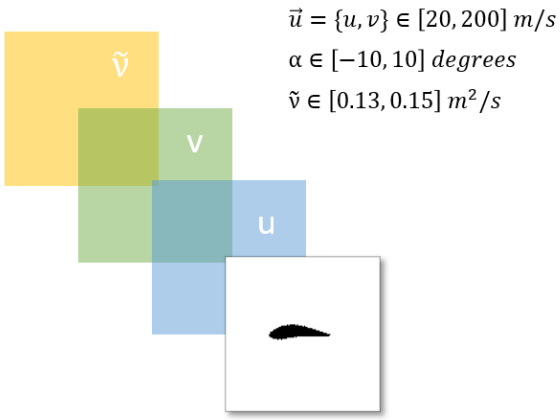

Figure 3.9: Input configuration of the model. For presentation purposes, the placement of channels is shown in a reversed order here. The mathematical representation precedence is the order with airfoil map, u, v, $\tilde{\nu}$ values for each query point stacked channel-wise.

get farther from the airfoil surface (in a physical domain bounded by a sufficiently large far field). Thus for a more accurate solution, more query points are required in the vicinity of the airfoil shape whereas, fewer and fewer points are required traveling farther from the airfoil. Indeed during the process of creating the computational mesh representation of the domain, almost the same logic is applied. Generally, a higher density of mesh cells is added closer to the airfoil than to the far field. In this study, a uniform grid of size $\mathbb{R}^{128\times128}$ is queried in the bounding box (BB) [(-1.4, 0.5), (0.4, -0.5)] (which is in the vicinity of the airfoil) and the rest of the points are generated with an exponential growth rate of 1.1 in both x and y directions. The generated exponentially spaced points are fitted to their corresponding bounding box starting from one or two vertices of the bounding box of the uniform grid to a vertex or edges of the smallest bounding box that encloses the whole problem domain (i.e. a bounding box with corner vertices [(-10,10),(15,-10)] which will be referred to as outer bounding box in the rest of the thesis). The query points placement for the output is presented in figure 3.10.

After the query points are generated and probed or interpolated from the solution

mesh to create the training reference outputs (objective arrays), the points that are inside the airfoil are set to zero. In order to do so, the airfoil mask instead of the airfoil geometry is used. This is because some points in the output query list might be outside the airfoil but have a value of 0.0 in the airfoil mask. In other words, these points are geometrically outside the airfoil but are represented in the airfoil mask as if they were inside the spatial domain enclosed by the airfoil. This happens when the physical domain surrounded by the airfoil shape occupies more than 50 percent of a pixel and the query point that physically lies outside the airfoil also lies inside that pixel. In such a case, in order to enforce the consistency between the inputs and outputs, the value of the output should be affected by the discretized version of the input rather than the continuous version of it. The same procedure is applied for the query points that are outside the problem domain. In other words, the points that are not enclosed by the far-field are also set to zero for all feature vector values in the output.

There are some other points that need to be clarified for the model's input and output settings. Firstly, one may notice that the spatial ranges that the input and outputs represent are not the same. The input's airfoil mask represents a bounding box in the vicinity of the airfoil while the output query points represent the whole spatial bounds of the problem. This will not be an issue as there is no reason to learn a one-to-one mapping from input physical locations to the same output locations. Also as the vicinity of the airfoil is provided in the airfoil mask -rather than the whole domain-, it would contain more information about the airfoil shape compared to when a one-to-one mapping is established. This in turn would result in a more accurate prediction of the solution features as more information about the geometric variations of the input shape is provided to the model. The other point that needs to be clarified here is the fact that the non-uniform property of the output query points might negatively affect the training as the distances are not known by the model and the non-uniform query points are treated as uniformly distant ones by the model (The model treats the inputs as images thus it assumes that each entry in height and width are equidistant). Yet this would not pose a problem here as the query point locations are the same for all simulations thus the variations in the same locations are learned. The non-uniform query setting not only does not affect the training process negatively but also allows

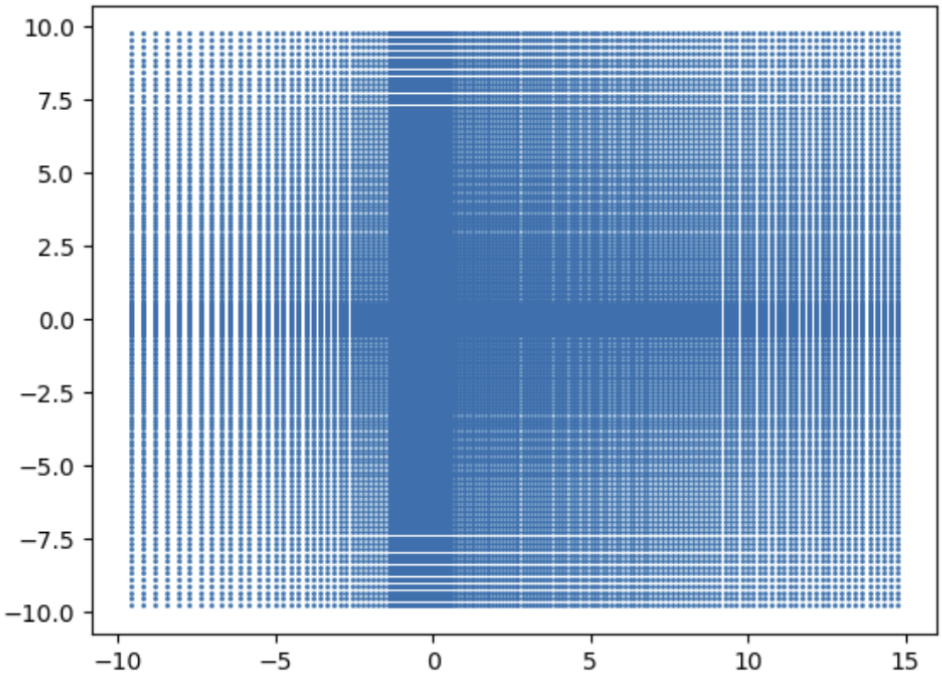

Figure 3.10: Probe/interpolation query points. After generation, these points are queried in the solutions computation mesh to create the model's output references to be learned.

more variation in output features to be captured and a more accurate mapping to be learned.

3.8 Training Set Generation

The training set consists of 1400 converged samples created by permuting the previously generated airfoils with different initial and boundary conditions (sampled from the ranges provided in table 3.1) and the 1000 generated airfoils. The airfoils are sampled uniformly by their ID (from 1 to 1000), thus some airfoils might not appear in the data set while some others might have appeared more than once. For each case, the mesh is generated in accordance with instructions provided in section 3.4. Then its initial and boundary conditions are set in their corresponding files. All the

information about the mesh settings (the instructions for generating the mesh), initial and boundary conditions, and the airfoil shape are also stored in a JSON file to be used later. For the solution of RANS equations, OpenFOAM software's SimpleFOAM solver with the settings described in section 3.5 is used for all the solutions. When the solver is converged the query points are provided to the OpenFOAM post-processing unit. Its output then is parsed to create the output reference array which is stored in a pickle file in binary format. In order to minimize the size of storage required for each input-output pair, the input arrays are not stored and are directly generated from the JSON files at the beginning of the training.

3.9 Training

There are some modifications that are applied to the inputs and outputs in order to facilitate the training of the model. These modifications as well as the training setup and procedure are described in this section. The ranges of the initial conditions that are provided in table 3.1 indicate that the scale of different inputs varies in different orders of magnitude which would result in some problems during optimization. The same problem is present in the outputs as well. The scaling issue is solved by introducing scaling factors for both inputs and outputs. These scaling factors are applied channel-wise to their corresponding location in the feature vector. The scaling formula applied to each variable in input and output is presented in table 3.3.

Table 3.3: Scaling operations for model inputs and outputs.

Variable	Unit	Range	Scaling formula	Variation Range
u_x	$\frac{m}{s}$	$[19.69, 200]$	$u_x := \frac{u_x}{100.}$	$[0.196, 2]$
u_y	$\frac{m}{s}$	$[-34.72, 34.72]$	$u_y := \frac{u_y}{100(sin(10))}$	$[-2, 2]$
P	$\frac{kg}{ms^2}$	$[0, U_\infty^{max}]$	$P := \frac{P}{U_\infty^{max}}$	$[0, 1]$
ν_t	$\frac{m^2}{s}$	$[0.13, 0.15]$	$\nu_t := \frac{\nu_t}{0.15}$	-
$\tilde{\nu}$	$\frac{m^2}{s}$	$[0.13, 0.15]$	$\tilde{\nu} := \frac{\tilde{\nu}}{0.15}$	-

With U_∞^{max} corresponding to the maximum dynamic pressure.

$$U_{\infty}^{max} = \frac{\rho \times max(U)^2}{2.} \tag{3.10}$$

The Adam [75] optimizer with learning rate $lr = 10^{-3}$, and $\beta_1 = 0.95$, $\beta_2 = 0.99$ is utilized to update the model parameters. The batch size is decreased from 200 samples to 5 samples gradually during the training and the model is trained until the MSE of the target and the predicted solution for all the cases fell below 1e-6 threshold. Figure 3.11 shows the input and reference output channels used for the training of the model.

3.10 Results Inference and Updating the Initial and Boundary Conditions

To prepare the new initial and boundary conditions for the solver the results of the model have to be transferred to mesh center points and boundary faces. As the output query points are constructed by permutation of two vectors corresponding to the points in the x and y directions, the pixel containing any point in the problem domain can be easily found via binary search in O(N log N) time complexity. This in turn speeds up the process of transferring the solutions to the mesh. After the model prediction for the solution of a specific case is obtained (inference), these results are transferred to the cell centers for the inner cells and the face centers for the boundary faces of the computational mesh. Here for each cell center (or face center) in the computational mesh of the problem, the corresponding pixel which contains that point is queried. Then the corresponding values of the solution variables in the feature vector of that pixel are transferred to the cell center (or face center) by interpolating the values of four results bounding that cell.

3.11 Results

In this part, the performance of the model explained in this chapter is explored. For the reader's reference, the intermediate results of the model are also presented in Appendix A.1. After obtaining the intermediate results, they are transferred to the mesh and a traditional finite volume-based solver is used to obtain the converged

solution of the RANS equations. The predicted initial/boundary conditions that are transferred to the computational mesh of the solver as well as the converged results for randomly selected airfoil geo115 given the initial flow conditions, $\tilde{\nu}, \nu_t = 0.145\frac{m^2}{s}$, $u_x = 36.60\frac{m}{s}$, and $u_y = 2.56\frac{m}{s}$ are also provided in the appendix (A.2).

In order to test the speed-up performance of the procedure proposed 2795 random cases are solved. These test cases are generated by sampling the airfoils from UIUC airfoil database [48]. The initial conditions of the problem are selected from a uniform distribution from the same ranges used for the training shown in table 3.1. Here the only difference is that the airfoils are refined before mesh generation whereas for the training set the original randomly generated airfoils are used. The process of refining airfoil shapes is adopted from open source software PyAero by Andreas Ennemoser [90]. For each airfoil first, a b-spline is fitted to the provided points, and a set of new points for the upper and lower curves are generated that are homogeneously distanced. This is done using the arc length of each curve as b-splines are arc-length parameterized. After the generation of these equidistant points, the algorithm traverses the edges of the airfoil and checks if the angle between two adjacent edges does satisfy a specific criterion. Here instead of storing the edges, the vertices constituting each edge are stored consecutively in a counter-clockwise manner in a doubly-linked list thus the angle between two edges can be simply calculated using each set of three successive entries in the list. The criterion is set to 172 degrees threshold, in other words, if the angle between two adjacent edges on the airfoil has a value less than 172 degrees, they will not meet the criterion. If the criterion is not met, two new points are sampled in the half distance within the segments represented by those two edges and added to the list. This process is iterated 3 times (The number of iterations is defined as a parameter). Then the trailing edge is refined by selecting a predefined number of points in both upper and lower airfoil curves, and sampling a new number of equidistant points in the segments represented by that number. Here for the airfoil trailing edge refinement, the last 3 points on both the upper and the lower curves of the trailing edge are re-sampled to 6 points. An example of the refined version of the Goe115 is presented in figure 3.12. In this example, the algorithm refines both the leading and trailing edges of the airfoil as can be seen by the high density of the circle markers in those areas. This refinement in the leading edges allows the mesh genera-

tion algorithm to discretize a smoother shape conforming to the previously explained criterion.

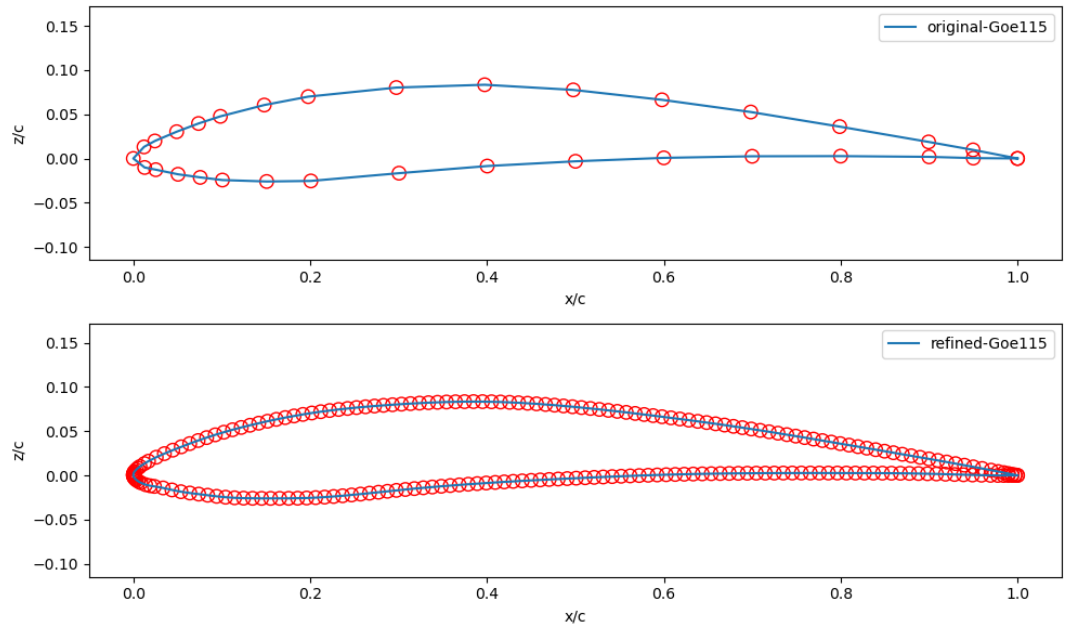

Figure 3.12: Original (top) and Refined (bottom) versions of the Goe115 airfoil shape.

After the airfoils are refined, the computational mesh for the random set of test cases containing the initial and boundary conditions of the problem is created. For each configuration in the test set, two different cases are created where the first one contains only the original initial and the boundary conditions and the second one contains the predicted solution as its initial and boundary condition. Both of these cases are then solved by the finite-volume method-based solver and then their performance is compared to each other. Here two metrics are defined to provide the means to compare the performance of the solver when each set of two different cases discussed above is supplied to the solver. The first metric is the speed up in the number of iterations which is defined as the ratio of the number of iterations it takes for the solver to converge given the original initial and boundary conditions vs when the predicted initial and boundary conditions are provided. Here the number of time steps the solver takes to converge to a steady solution (given the convergence criteria) is referred to as the number of iterations instead of the exact number of iterations. This is because the exact number of internal iterations is not known. The second metric is the speed up in

the wall time, which is defined in the same way by changing the number of iterations by the wall time. In the next section, both of these metrics are used to compare the speed up of the method presented in this chapter.

3.11.1 Speed up Results With OpenFOAM Normalized Residuals

The OpenFOAM software uses normalized residuals in its solvers' implementations. In order to change the residual calculation, each specific solver has to be modified and the software to be recompiled. Here the default residual calculations methodology in OpenFOAM software is used in this section. More detail on the residual calculation methodology in OpenFOAM as well as the analysis of the results of this section in terms of the absolute residuals is presented in the following section.

The speed up (in the number of iterations and Wall time) of the results of the proposed model on each test case is provided in figures 3.13, and 3.14. It can be seen that for all the cases that are tested, the process results in a speed-up of the solution of that particular case for the investigation of subsonic, incompressible, 2D flow around airfoils given the RANS equations. The results in this moderate test set show that an average of $2.28\times$ speed-up in the number of iterations and $2.61\times$ speed-up in wall-time can be obtained. According to the results presented in figures 3.13, and 3.14, there is a difference between the speed-up attained for the number of iterations and the Wall-time. These results should be generally close to each other but many factors may lead to the difference between these two metrics. A part of this difference might be due to the interference of system processes in the wall-time calculation. Yet as each pair of the solutions are performed in parallel such a dominant difference is not likely to be due to system process scheduling interference. Another reason might be due to the iterative manner of convergence of the smooth-solver at each time step (iteration) of the solution. It is observed that the smooth-solver performs more iterations at each time step of the solution when the original initial and boundary conditions is provided compared to when the predicted solution is used as the initial and boundary conditions. Indeed this observation can be validated by providing the final converged solution of each case to the solver as initial and boundary conditions. For instance when the solution of the Goe155 airfoil with the initial and boundary

conditions $\tilde{\nu}, \nu_t = 0.145\frac{m^2}{s}$, $u_x = 36.60\frac{m}{s}$, and $u_y = 2.56\frac{m}{s}$ is provided to the solver it converges at 42 time steps on 1.11s wall time. The same case converges at 884 time-steps in 27.45s wall time which is indicative of the similar difference in speed-up metric results. The exact reasons why there is a difference between the speed-up metrics in wall time and the number of iterations are out of the scope of this thesis and are not further investigated here.

The best speed-up in the number of iterations attained in the test set solutions was $4.86\times$ whereas the worst case had just a 3 percent improvement in the number of iterations. This method also achieved $5.69\times$ and $1.08\times$ speed up for the best and worst case in wall time.

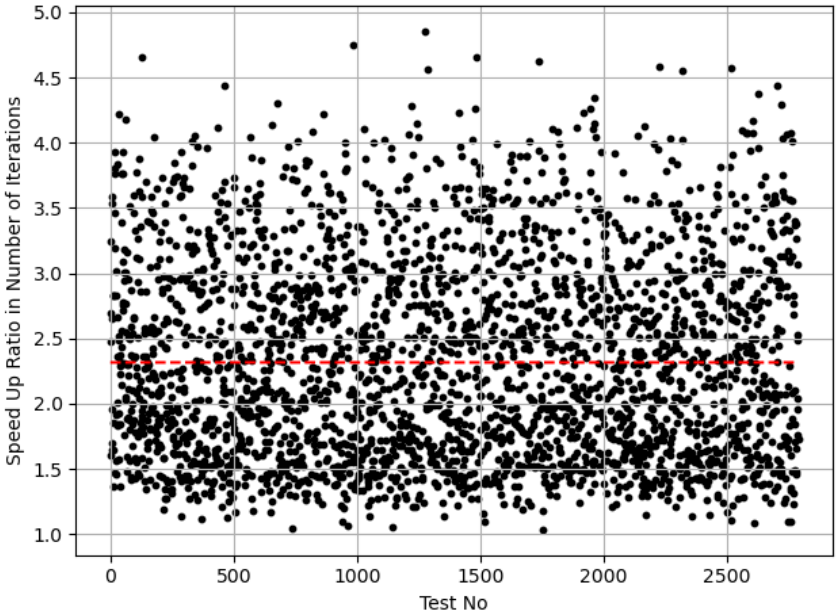

Figure 3.13: Speed-Up ratio in the number of iterations for each test case. The dashed line (Red) represents the mean speed-up for all the cases in the test set.

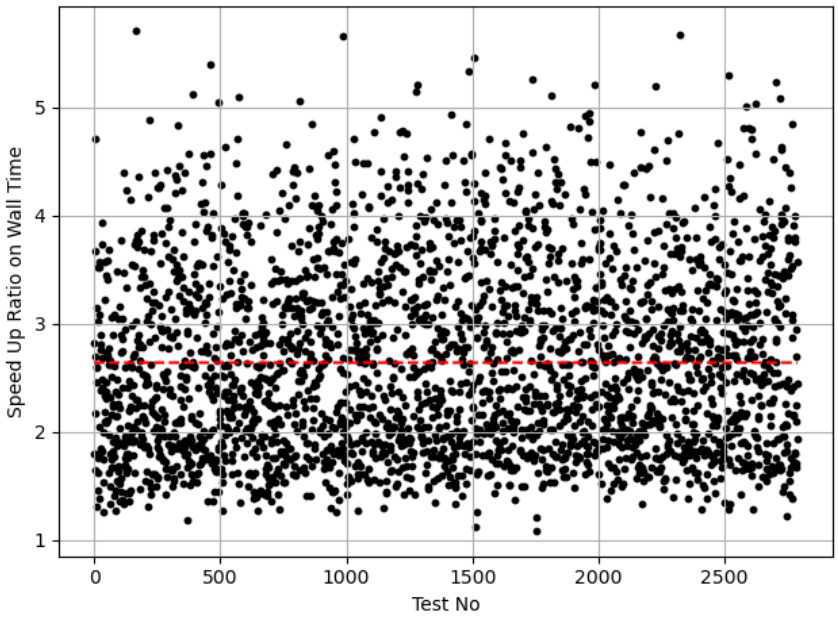

Figure 3.14: Speed-Up ratio in Wall-time for each test case. The dashed line (Red) represents the mean speed-up for all the cases in the test set.

Other interesting observations can be made when the speed up in the number of iterations or wall time is investigated as a function of the number of iterations and wall time of the original initial and boundary conditions respectively. These results for the test set are presented in figures 3.15, 3.16. For the rest of this part, the cases with original initial and boundary conditions will be referred to as the original cases. The cases where the predicted solution is provided as the initial condition will be referred to as predicted cases for the rest of this section.

One may observe that there exists a correlation between the number of iterations it takes for the solution of the original cases to converge and the speed up attained with the proposed cases. In other words, in figure 3.15 one can observe the following trend. Here as the number of iterations required for convergence of the original cases increases the speed up obtained by providing the predicted initial and boundary conditions to the solver also increases. A similar trend can be seen in figure 3.16 where the speed up of wall time as a function of wall time of the original cases is presented.

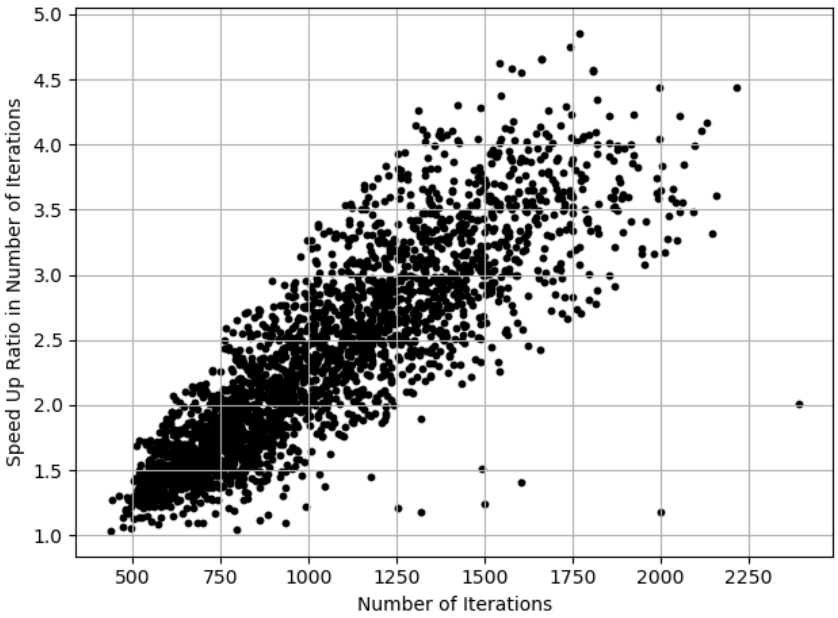

Figure 3.15: Speed-Up in iterations vs number of solution iterations to converge with the original initial conditions.

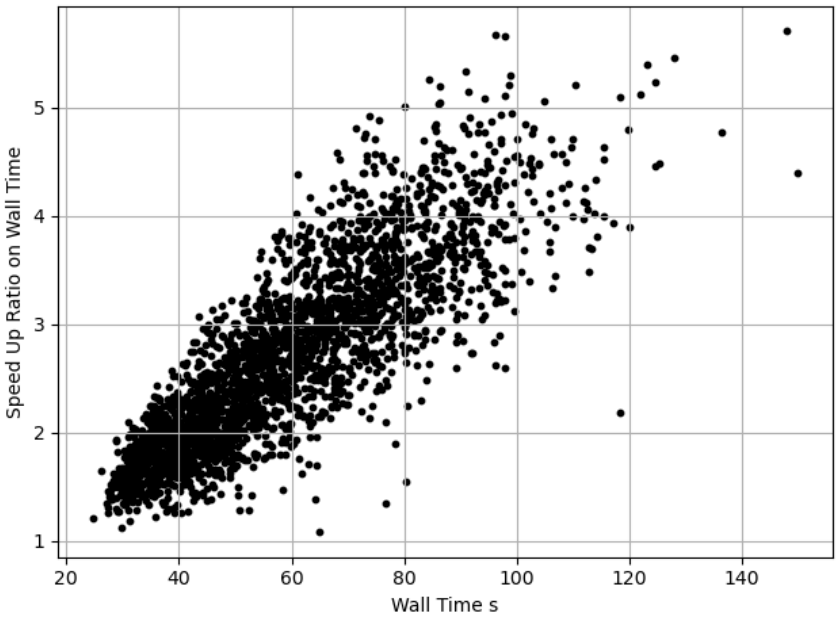

Figure 3.16: Speed-Up in wall-time vs simulation Wall-time to converge with the original initial conditions.

Indeed investigating the time or the number of iterations it takes for each case to converge one can observe that a large portion of the predicted cases converge almost at the same time no matter how many iterations it took for the original cases to converge. These results can be observed in figures 3.17, 3.18. It is a very interesting observation that suggests that the proposed method makes the solutions more robust. It might be argued that the reason for the convergence delay in some of the original cases is due to the causes of instability in the solution convergence, as the generated mesh is almost similar in all the cases (neglecting the airfoil shape) and the initial values for the flow variables are from a fairly limited range. One might also argue that the solutions obtained when the proposed method is used, are not affected by these causes of instability in the solution convergence as almost all the cases where the predicted solution is provided as the initial and boundary conditions to the solver, converge at almost the same time.

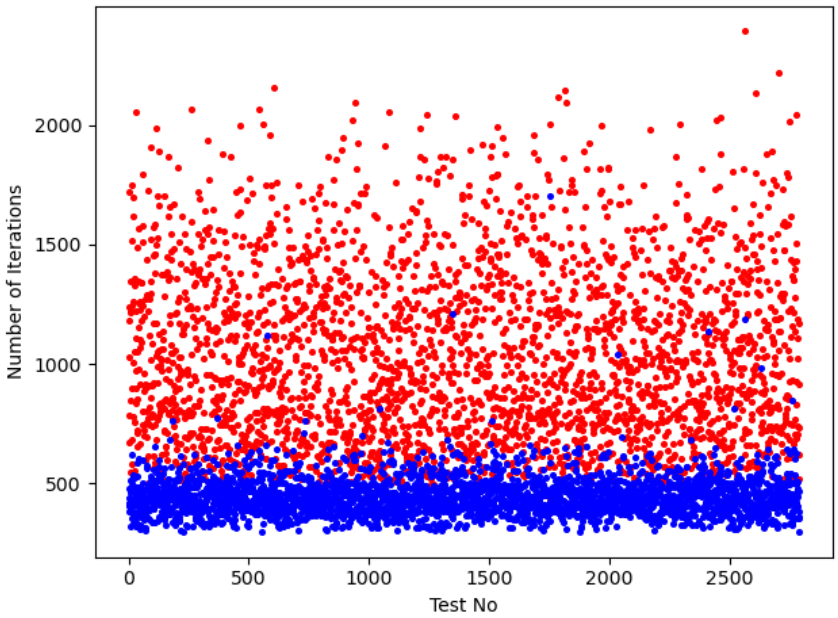

Figure 3.17: Number of iterations before convergence (red) original cases, (blue) proposed cases for each test case.

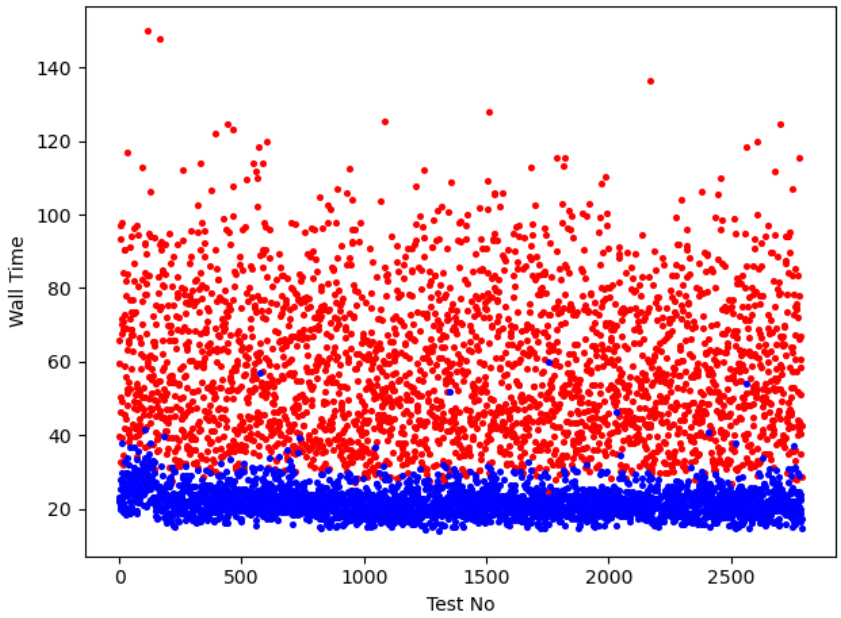

Figure 3.18: Wall-time before the convergence of (red) original cases, (blue) proposed cases for each test case.

Another parameter that can be investigated here is the predicted and the converged values of the pressure coefficient on the airfoil shapes. The pressure coefficient "c_p" is defined in equation 3.11 and is the measure of the ratio of pressure forces to inertial forces which is measured on the surface of the airfoil.

$$c_p = \frac{P - P_\infty}{P_0 - P_\infty} = \frac{P - P_\infty}{\frac{1}{2}\rho U_\infty^2} \tag{3.11}$$

Here the pressure distribution for the case with the highest speed up is presented in figure 3.19. The airfoil shape used in this case is naca65206 with $\tilde{\nu}, \nu_t = 0.148\frac{m^2}{s}$, $u_x = 32.66\frac{m}{s}$, and $u_y = -2.28\frac{m}{s}$.

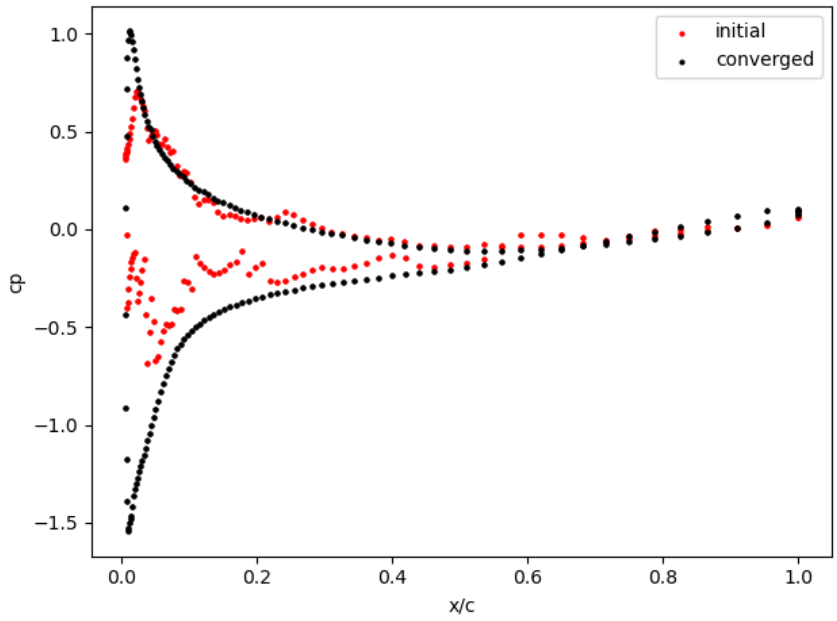

Figure 3.19: "c_p" distribution over the airfoil for the highest speedup in test case (naca65206, u=32.66, v=-2.28).

The "c_p" distribution for the predicted initial condition (red) and the final converged solution (black) are in agreement for most of the upper and lower curves of the airfoil, whereas they show differences in the vicinity of the stagnation point. As expected the highest error in the model's prediction is present where the highest curvature in airfoil shape is observed. These parts are where the highest rate of change in flow variables is observed. On the other hand, due to the discretization of the airfoil shape to the airfoil map, most of the information about the curvature is lost, resulting in higher error in those areas. Yet even if there exists an error in the pressure coefficient in the predicted solution, using them as the initial and boundary conditions for the model both improves the stability and convergence performance of the solution.

3.11.2 Residual Analysis

The OpenFOAM software uses normalized residuals for each solution variable to terminate the solution. The residual normalization procedure is presented in equation 3.12.

$$\bar{r} = \frac{\sum |\mathbf{b} - A\mathbf{x}|}{\sum(|A\mathbf{x} - A\bar{\mathbf{x}}| + \sum(|\mathbf{b} - A\bar{\mathbf{x}}|)} \tag{3.12}$$

The solution is terminated when the normalized residual falls below a predefined tolerance for each variable. This poses a problem when testing the hypothesis behind this chapter. As the residual is normalized by the deviation from the average of the solution vector, a large absolute residual for any variable might be normalized to a value close to the termination tolerance.

To that end, the source code of the OpenFOAM software is modified for both "GMAG" and "smoothsolver" so that they use the absolute residual instead of its normalized version. It turns out, in the beggining the residual of the predicted case, is larger than that of the original case.

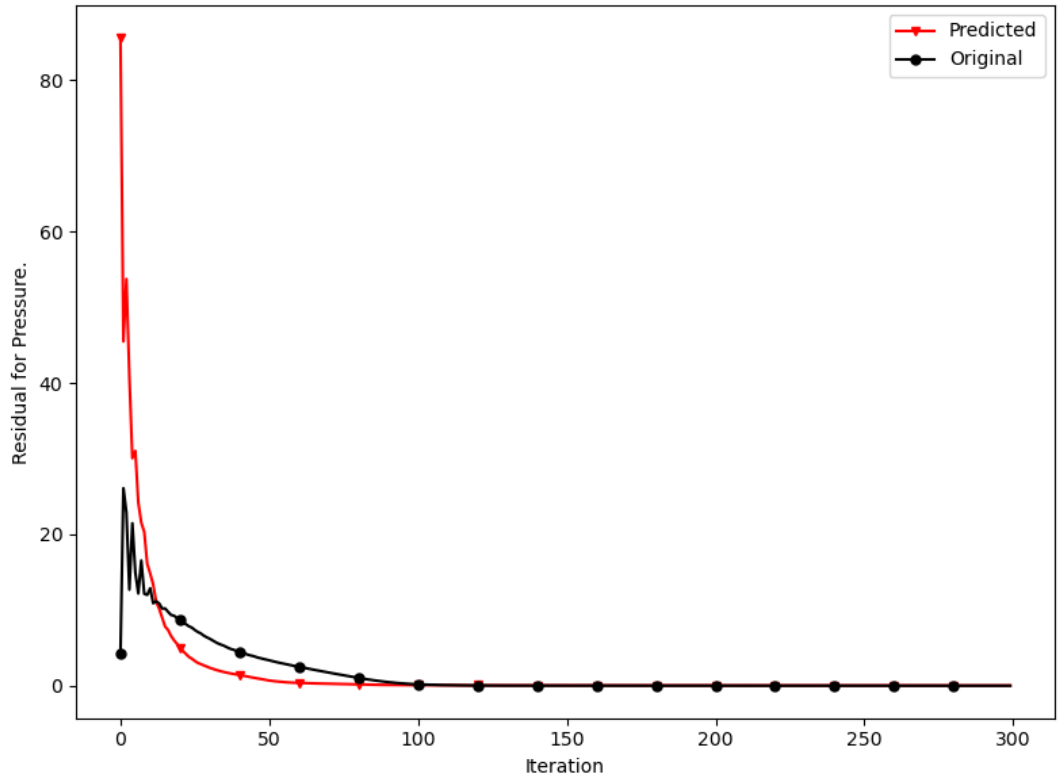

Figure 3.20: Pressure absolute residual of predicted vs original case.

Figure 3.20 compares the absolute values of initial residual for pressure during the first 300 iterations of the simpleFoam solver for the Goe155 airfoil with the initial

and boundary conditions $\tilde{\nu}, \nu_t = 0.145\frac{m^2}{s}$, $u_x = 36.60\frac{m}{s}$, and $u_y = 2.56\frac{m}{s}$. Here it is evident that the actual residual of the predicted case is initially higher but gets closer and closer to the residual of the original case in a couple of iterations. On the other hand, when the normalized residuals are used, due to a larger normalization factor the residual of the predicted case starts from a smaller value compared to 1.0 in the original case. This is shown in figure 3.21.

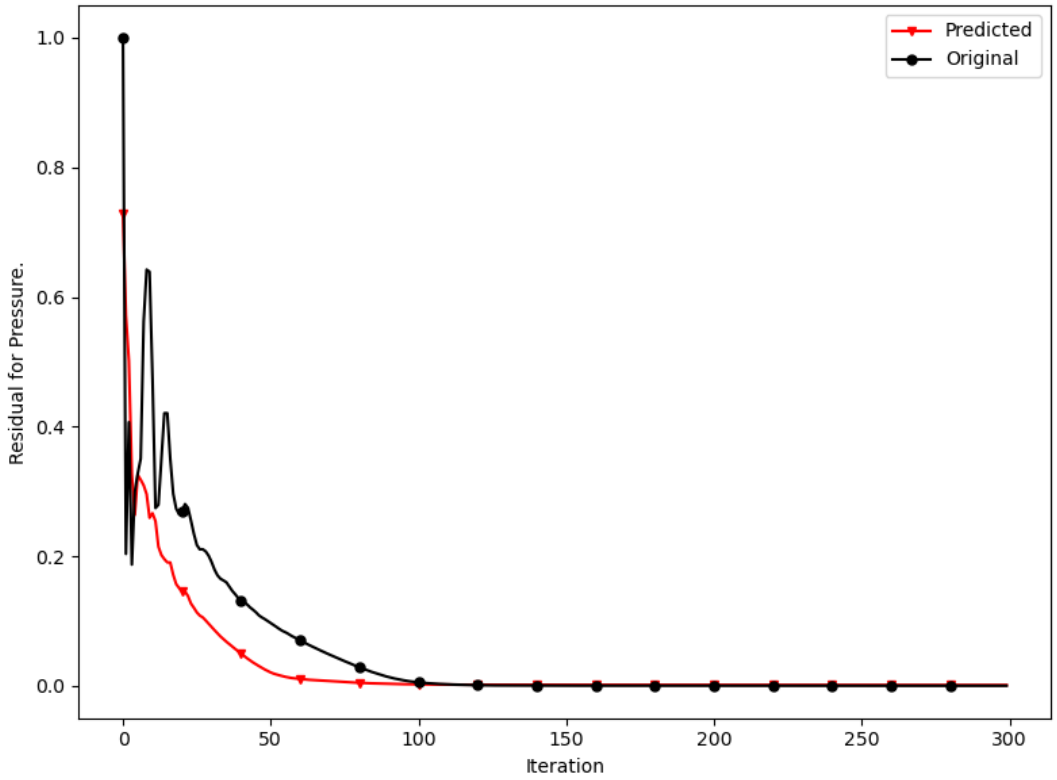

Figure 3.21: Pressure normalized residual of predicted vs original case.

Although both of these cases are terminated for the same termination residual tolerances, the converged residual of the predicted case is higher than that of the original case. Yet this residual decreases rapidly in the first couple of time iterations for the predicted case. This phenomenon can be explained by considering how the error is decreased during each simulation iteration for both cases. In the original case, the main cause of the error is the deviation of the uniform initial conditions from the solution of the RANS equations in both the vicinity of the airfoil and its wake. However, the main cause of the error in the predicted case is generally the disruption of

residual during the results transfer (to mesh) phase of this method. For the predicted case, the error is mitigated by simple local adjustments of the solution vectors while for the original case, this happens by slowly adjusting the solution vector from the airfoil shape towards the free stream.

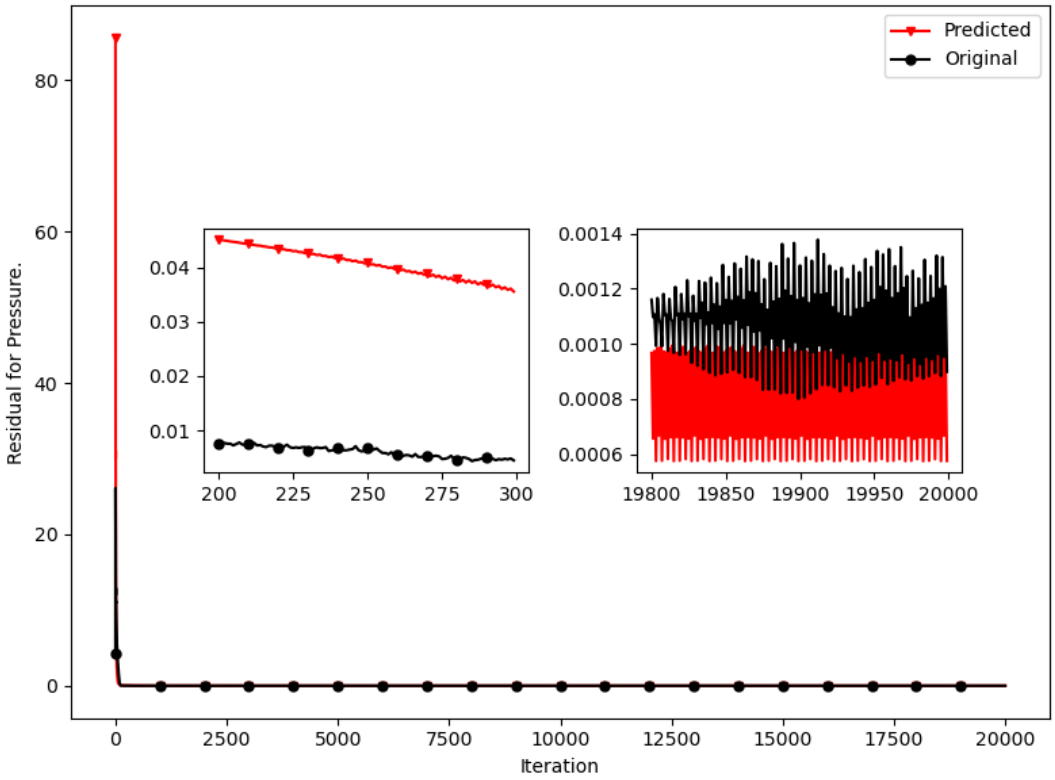

Figure 3.22: Pressure absolute residual of predicted vs original case.

Even if the initial value of the predicted case starts from a comparably higher value, it converges to the absolute residual of the original case as can be observed in figure 3.22. Another observation that can be made here is the fact that the rate of change of the absolute residual for the predicted case is higher than that of the original case which suggests that there is some speed up even when the absolute residual is used.

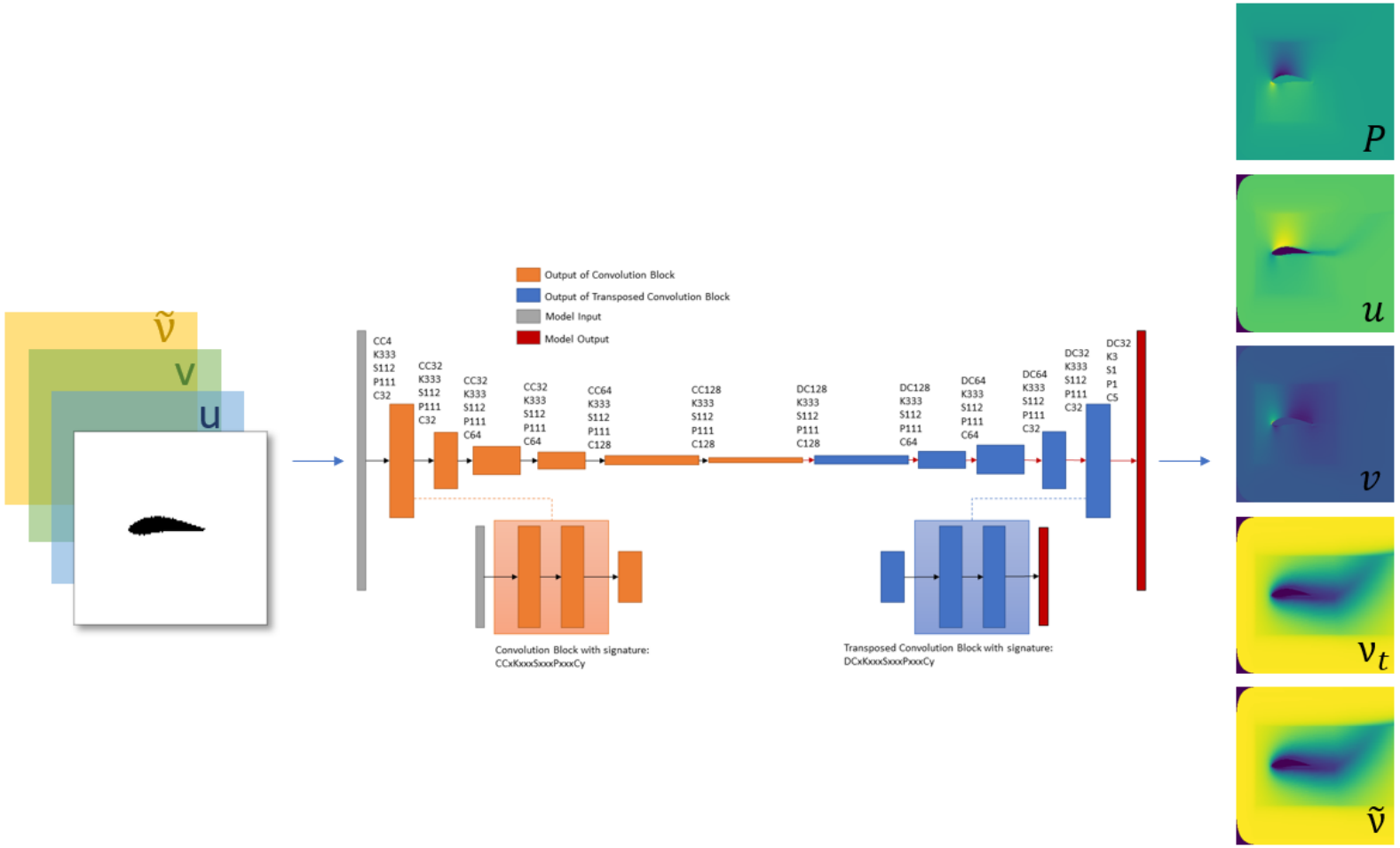

Figure 3.11: Input and reference values for training the model.

CHAPTER 4

CONCLUSIONS

The main objective of this thesis is to speed up PDE solvers by utilizing state-of-the-art deep learning methods. The main motivation behind this idea is the fact that for the design of systems that involve solutions of differential equations (like flow over objects, heat transfer, etc.) a large set of viable candidates must be analyzed which generally takes a long time to achieve. As a result, it would of advantageous if the solution time of these solvers could be decreased significantly. To that end, two distinct methods for speed-up of two different paradigms for the solution of fluid dynamics problems are presented in this thesis.

In the first paradigm nonlinear, differentiable, and trainable models are utilized to solve transient problems by minimizing a loss that is a function of the residual of the governing equations, and error in initial and boundary conditions and as well as the measured data, The trainable parameters of these models are optimized with a gradient descent-based optimizer utilizing automatic differentiation. The second paradigm, on the other hand, involves the speed-up of the solution process of steady-state problems with discretization-based solvers

In case of transient problems, a parametric domain encoding-based physics-informed neural network is proposed which utilizes the finite-difference method to estimate the PDE's residual. It is shown that automatic differentiation is not a suitable method for the calculation of the residual for high-order derivatives as its time and memory complexity increases with the model size and the order of the derivatives to be calculated. As a result, to improve the performance of the proposed PEPINN model, the automatic differentiation is only used for updating the model parameters and the derivative calculations for residual are handled via finite difference kernels. The PEPINN model

utilizing the parametric domain encoding allows the solution to be locally stored in the latent encoding vectors, instead of globally in weights of a MLP for the whole domain. Here by globally, it is meant that the same weights are applied for all the points in the domain whereas in the locally stored weights each subdomain would have its own interpretation of the solution and thus different weights. In order to test the PEPINN model, Taylor-Green Vortex problem with Dirichlet boundary conditions is solved and the performance of the proposed model is compared to the baseline PINN model selected to solve the same problem. Also, it is shown that the performance of the PEPINN model both in terms of the speed-up and loss minimization, can be further improved by gradually increasing the number of query points sent to the model during the training process.

In the case of steady problems, the solution of the PDE is learned for a small dataset, then the trained model is used to obtain predictions of the solutions for unseen but similar cases, and these predictions are supplied to the solver as its starting conditions. It is empirically observed that if a close enough prediction of the problem variables is provided to the solver it will speed up the solution's convergence time. This idea is tested for the solution of the RANS equation for steady, subsonic, incompressible flow over 2D airfoils where a custom Unet-based model is trained on the solution of randomly generated samples with random airfoil shapes. The trained model later is used to predict the solution of samples generated using the UIUC airfoil database. Later these solutions are transferred to the computational mesh of the problem as its starting conditions. This process has yielded satisfactory results, by speeding up all of the 2795 cases tested for this thesis under the normalized residual calculation metric of the OpenFOAM software.

4.1 Future Work

The training of the PEPINN model proposed in this thesis requires a large number of query points covering the whole problem Spatio-temporal domain in order to both achieve high accuracy (as a result of the application of finite difference for derivative calculations) and lower loss values. As an alternative training method, a stochastic query point generation method and its training process can also be investigated in the

context of PEPINN models. In other words instead of a uniform grid covering the whole domain a batch of randomly generated smaller bounding boxes that are not disjoint with the problem domain can be used to train PEPINN models. Of course, this would require more iterations to satisfy the residual of the differential equation as well as initial and boundary conditions but can be made more accurate by selecting shorter point spacings at each batch. Also, the performance of this model can be investigated for the solution of other partial differential equations.

REFERENCES

[1] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” Math Control Signals Syst, vol. 2, no. 4, pp. 303–314, 1989, doi: 10.1007/bf02551274.

[2] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” Neural Networks, vol. 2, no. 5, pp. 359–366, 1989, doi: 10.1016/0893-6080(89)90020-8.

[3] F. Scarselli and A. C. Tsoi, “Universal Approximation Using Feedforward Neural Networks: A Survey of Some Existing Methods, and Some New Results,” Neural Networks, vol. 11, no. 1, pp. 15–37, 1998, doi: 10.1016/s0893-6080(97)00097-x.

[4] “ImageNet Classification with Deep Convolutional Neural Networks,” n.d..

[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” Arxiv, 2015

[6] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” Arxiv, 2015.

[7] Q. Xie, M.-T. Luong, E. Hovy, and Q. V. Le, “Self-training with Noisy Student improves ImageNet classification,” Arxiv, 2019.

[8] A. Dosovitskiy et al., “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” Arxiv, 2020.

[9] J. Yu, Z. Wang, V. Vasudevan, L. Yeung, M. Seyedhosseini, and Y. Wu, “CoCa: Contrastive Captioners are Image-Text Foundation Models,” Arxiv, 2022.

[10] E. Shelhamer, J. Long, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” Ieee T Pattern Anal, vol. 39, no. 4, pp. 640–651, 2016, doi: 10.1109/tpami.2016.2572683.

[11] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN,” Arxiv, 2017.

[12] J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement,” Arxiv, 2018.

[13] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, “SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers,” Arxiv, 2021.

[14] I. J. Goodfellow et al., “Generative Adversarial Networks,” Arxiv, 2014.

[15] “UNSUPERVISED REPRESENTATION LEARNING GENERATIVE ADVERSARIAL NETWORKS,”

[16] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein GAN,” Arxiv, 2017.

[17] “Glow: Generative Flow with Invertible 1×1 Convolutions,”

[18] T. Karras, T. Aila, S. Laine, and J. Lehtinen, “Progressive Growing of GANs for Improved Quality, Stability, and Variation,” Arxiv, 2017.

[19] A. Karnewar and O. Wang, “MSG-GAN: Multi-Scale Gradients for Generative Adversarial Networks,” Arxiv, 2019.

[20] T. Karras, S. Laine, and T. Aila, “A Style-Based Generator Architecture for Generative Adversarial Networks,” Arxiv, 2018.

[21] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, “Self-Attention Generative Adversarial Networks,” Arxiv, 2018.

[22] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” Arxiv, 2014.

[23] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” arXiv.org, 19-May-2016. [Online]. Available: https://arxiv.org/abs/1409.0473.

[24] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional Sequence to Sequence Learning,” Arxiv, 2017.

[25] A. Vaswani et al., “Attention Is All You Need,” Arxiv, 2017.

[26] V. Mnih et al., “Playing Atari with Deep Reinforcement Learning,” Arxiv, 2013.

[27] M. Hausknecht and P. Stone, "Deep Recurrent Q-Learning for Partially Observable MDPs," Arxiv, 2015.

[28] V. Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning," Arxiv, 2016.

[29] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," Arxiv, 2015.

[30] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," Arxiv, 2015.

[31] D. C. Psichogios and L. H. Ungar, "A hybrid neural network-first principles approach to process modeling," AIChE Journal, vol. 38, no. 10, pp. 1499–1511, Oct. 1992, doi: 10.1002/aic.690381003.

[32] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," Psychol Rev, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.

[33] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," IEEE Transactions on Neural Networks, vol. 9, no. 5, pp. 987–1000, Sept. 1998, doi: 10.1109/72.712178.

[34] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," J Comput Phys, vol. 378, pp. 686–707, 2019, doi: 10.1016/j.jcp.2018.10.045.

[35] Y. Zhang, W. J. Sung, and D. N. Mavris, "Application of Convolutional Neural Network to Predict Airfoil Lift Coefficient," 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 2018, doi: 10.2514/6.2018-1903.

[36] V. Sekar and B. C. Khoo, "Fast flow field prediction over airfoils using deep learning approach," Physics of Fluids, 2019.

[37] H. Moin, H. Z. I. Khan, S. Mobeen, and J. Riaz, "Airfoil's Aerodynamic Coefficients Prediction using Artificial Neural Network," arXiv, 2021.

[38] C. Duru, H. Alemdar, and O. U. Baran, “A deep learning approach for the transonic flow field predictions around airfoils,” Computers & Fluids, vol. 236, p. 105312, 2022, doi: https://doi.org/10.1016/j.compfluid.2022.105312.

[39] A. B. Farimani, J. Gomes, and V. S. Pande, “Deep Learning the Physics of Transport Phenomena,” Arxiv, 2017.

[40] Y. Xie, E. Franz, M. Chu, and N. Thuerey, “Data-driven synthesis of smoke flows with CNN-based feature descriptors,” Acm Transactions Graph Tog, vol. 36, no. 4, pp. 1–14, 2017, doi: 10.1145/3072959.3073643.

[41] N. Thuerey, K. Weißenow, L. Prantl, and X. Hu, “Deep Learning Methods for Reynolds-Averaged Navier–Stokes Simulations of Airfoil Flows,” Aiaa J, vol. 58, no. 1, pp. 25–36, 2019, doi: 10.2514/1.j058291.

[42] S. Wiewel, M. Becher, and N. Thuerey, “Latent Space Physics: Towards Learning the Temporal Evolution of Fluid Flow,” Comput Graph Forum, vol. 38, no. 2, pp. 71–82, 2019, doi: 10.1111/cgf.13620.

[43] B. Kim, V. C. Azevedo, N. Thuerey, T. Kim, M. Gross, and B. Solenthaler, “Deep Fluids: A Generative Network for Parameterized Fluid Simulations,” Comput Graph Forum, vol. 38, no. 2, pp. 59–70, 2019, doi: 10.1111/cgf.13619.

[44] O. Obiols-Sales, A. Vishnu, N. Malaya, and A. Chandramowliswharan, “CFDNet,” Jun. 2020. doi: 10.1145/3392717.3392772.

[45] M. Mirza and S. Osindero, “Conditional Generative Adversarial Nets,” Arxiv, 2014.

[46] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-Image Translation with Conditional Adversarial Networks,” Arxiv, 2016.

[47] C. Jiang and A. B. Farimani, “Deep Learning Convective Flow Using Conditional Generative Adversarial Networks,” Arxiv, 2020.

[48] M. S. Selig, “UIUC airfoil data site,” 1996.

[49] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” Arxiv, 2015.

[50] S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[51] Z. Li et al., "Fourier Neural Operator for Parametric Partial Differential Equations," Arxiv, 2020.

[52] S. Cuomo, V. S. di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli, "Scientific Machine Learning through Physics-Informed Neural Networks: Where we are and What's next," Arxiv, 2022.

[53] Z. Mao, A. D. Jagtap, and G. E. Karniadakis, "Physics-informed neural networks for high-speed flows," Comput Method Appl M, vol. 360, p. 112789, 2020, doi: 10.1016/j.cma.2019.112789.

[54] X. Jin, S. Cai, H. Li, and G. E. Karniadakis, "NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations," J Comput Phys, vol. 426, p. 109951, 2021, doi: 10.1016/j.jcp.2020.109951.

[55] S. Cai, Z. Wang, S. Wang, P. Perdikaris, and G. E. Karniadakis, "Physics-Informed Neural Networks for Heat Transfer Problems," J Heat Transf, vol. 143, no. 6, 2021, doi: 10.1115/1.4050542.

[56] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Inferring solutions of differential equations using noisy multi-fidelity data," J Comput Phys, vol. 335, pp. 736–746, 2017, doi: 10.1016/j.jcp.2017.01.060.

[57] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Machine learning of linear differential equations using Gaussian processes," J Comput Phys, vol. 348, pp. 683–693, 2017, doi: 10.1016/j.jcp.2017.07.050.

[58] P. Ren, C. Rao, Y. Liu, J.-X. Wang, and H. Sun, "PhyCRNet: Physics-informed convolutional-recurrent network for solving spatiotemporal PDEs," Comput Method Appl M, vol. 389, p. 114399, 2022, doi: 10.1016/j.cma.2021.114399.

[59] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W. Wong, and W. Woo, "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting," Arxiv, 2015.

[60] L. Sun, H. Gao, S. Pan, and J.-X. Wang, "Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data," Comput Method Appl M, vol. 361, p. 112732, 2020, doi: 10.1016/j.cma.2019.112732.

[61] A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis, "Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems," Comput Method Appl M, vol. 365, p. 113028, 2020, doi: 10.1016/j.cma.2020.113028.

[62] B. Wu, O. Hennigh, J. Kautz, S. Choudhry, and W. Byeon, "Physics Informed RNN-DCT Networks for Time-Dependent Partial Differential Equations," Arxiv, 2022.

[63] D. M. Harris and S. L. Harris. 2013. 3.4.2 - State Encodings. In Digital Design and Computer Architecture (second ed.). Morgan Kaufmann, Boston, 129–131. https://doi.org/10.1016/B978-0-12-394424-5.00002-1

[64] S. Theodoridis. 2008. Pattern Recognition. Elsevier.

[65] N. Rahaman et al., "On the Spectral Bias of Neural Networks," Arxiv, 2018.

[66] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis," Arxiv, 2020.

[67] M. Tancik et al., "Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains," Arxiv, 2020.

[68] R. Chabra et al., "Deep Local Shapes: Learning Local SDF Priors for Detailed 3D Reconstruction," Arxiv, 2020.

[69] C. M. Jiang, A. Sud, A. Makadia, J. Huang, M. Nießner, and T. Funkhouser, "Local Implicit Grid Representations for 3D Scenes," Arxiv, 2020.

[70] L. Liu, J. Gu, K. Z. Lin, T.-S. Chua, and C. Theobalt, "Neural Sparse Voxel Fields," Arxiv, 2020.

[71] S. Peng, M. Niemeyer, L. Mescheder, M. Pollefeys, and A. Geiger, "Convolutional Occupancy Networks," Arxiv, 2020.

[72] I. Mehta, M. Gharbi, C. Barnes, E. Shechtman, R. Ramamoorthi, and M. Chandraker, “Modulated Periodic Activations for Generalizable Local Functional Representations,” Arxiv, 2021.

[73] A. Yu, S. Fridovich-Keil, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, “Plenoxels: Radiance Fields without Neural Networks,” Arxiv, 2021.

[74] T. Müller, A. Evans, C. Schied, and A. Keller, “Instant Neural Graphics Primitives with a Multiresolution Hash Encoding,” Arxiv, 2022.

[75] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", Proc. Int. Conf. Learn. Representat., 2015.

[76] V. Nair and G. E. Hinton, “Rectified Linear Units Improve Restricted Boltzmann Machines,” 2010.

[77] A. L. Maas, “Rectifier Nonlinearities Improve Neural Network Acoustic Models,” 2013.

[78] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” 2015 IEEE International Conference on Computer Vision (ICCV), pp. 1026–1034, 2015.

[79] J. N. P. Martel, D. B. Lindell, C. Z. Lin, E. R. Chan, M. Monteiro, and G. Wetzstein, “ACORN: Adaptive Coordinate Networks for Neural Scene Representation,” Arxiv, 2021.

[80] J. Yu, L. Lu, X. Meng, and G. E. Karniadakis, “Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems,” Arxiv, 2021.

[81] M. Tan and Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” Arxiv, 2019.

[82] H. Robbins and S. Monro, “A Stochastic Approximation Method,” The Annals of Mathematical Statistics, vol. 22, no. 3, pp. 400–407, 1951, doi: 10.1214/aoms/1177729586.

[83] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” USSR Computational Mathematics and Mathematical Physics, vol. 4, no. 5, pp. 1–17, 1964, doi: https://doi.org/10.1016/0041-5553(64)90137-5.

[84] Y. Nesterov, “A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$,” 1983.

[85] J. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” Journal of Machine Learning Research, vol. 12, no. 61, pp. 2121–2159, 2011, [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html

[86] M. D. Zeiler, ADADELTA: An Adaptive Learning Rate Method. arXiv, 2012. doi: 10.48550/ARXIV.1212.5701.

[87] T. Tieleman, G. Hinton (2012) Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude. COURSERA: Neural Networks for Machine Learning, 4, 26-31.

[88] B. M. Kulfan, “Universal Parametric Geometry Representation Method,” J Aircraft, vol. 45, no. 1, pp. 142–158, 2012, doi: 10.2514/1.29958.

[89] K. Oguz and N. Sezer-Uzol, “AERODYNAMIC OPTIMIZATION OF HORIZONTAL AXIS WIND TURBINE ROTOR BY USING BEM, CST METHOD AND GENETIC ALGORITHM,” 2019.

[90] A. Ennemoser, “Pyaero” GitHub repository, https://github.com/chiefenne/PyAero.

Appendix A

QUALITATIVE RESULTS

A.1 Qualitative Results of Custom UNet Architecture

Here the result of the model described in 3.6 is presented in figures A.1-A.5. As it was explained this model outputs a set of points concatenated in five channels which are the two components of the velocity, pressure, ν_t, and $\tilde{\nu}$. Each pixel in these images corresponds to a query point of the same order in the probe query points list 3.10.

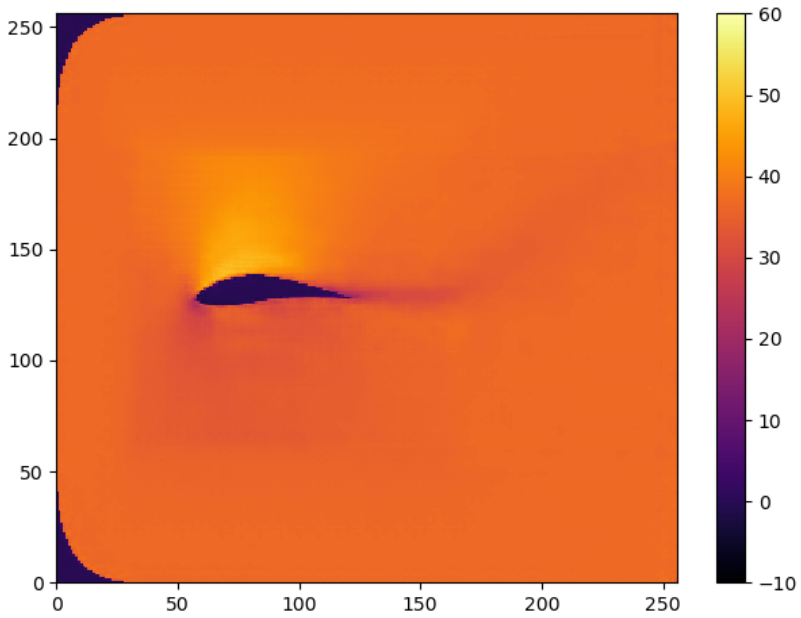

Figure A.1: u_x component of the predicted field.

Because there are more query points in the vicinity of the airfoil this section almost takes up half of the provided images also notice that the wake gets an abrupt angle

in the images and the reason for that is the fact that a large distance aft the airfoil is presented by points which their distance increases exponentially yet here this exponentially increasing distance is re-scaled to a uniform distance.

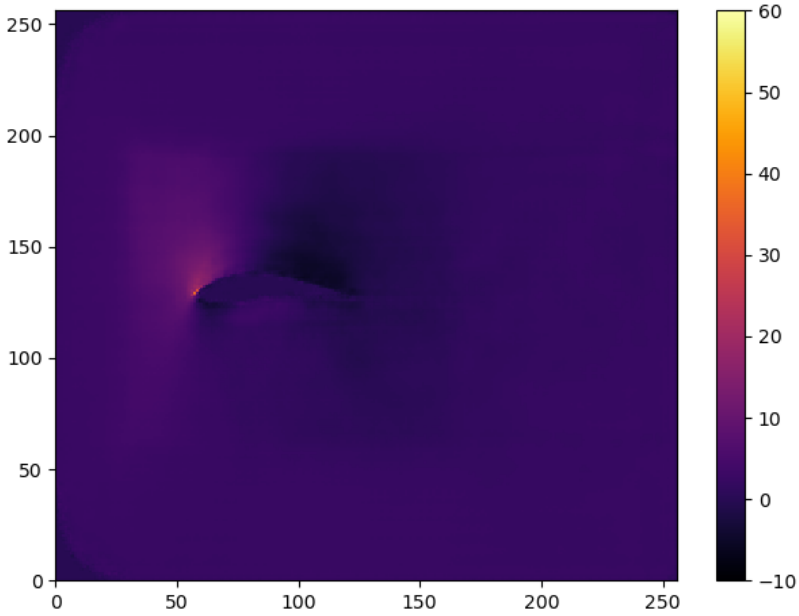

Figure A.2: u_y component of the predicted field.

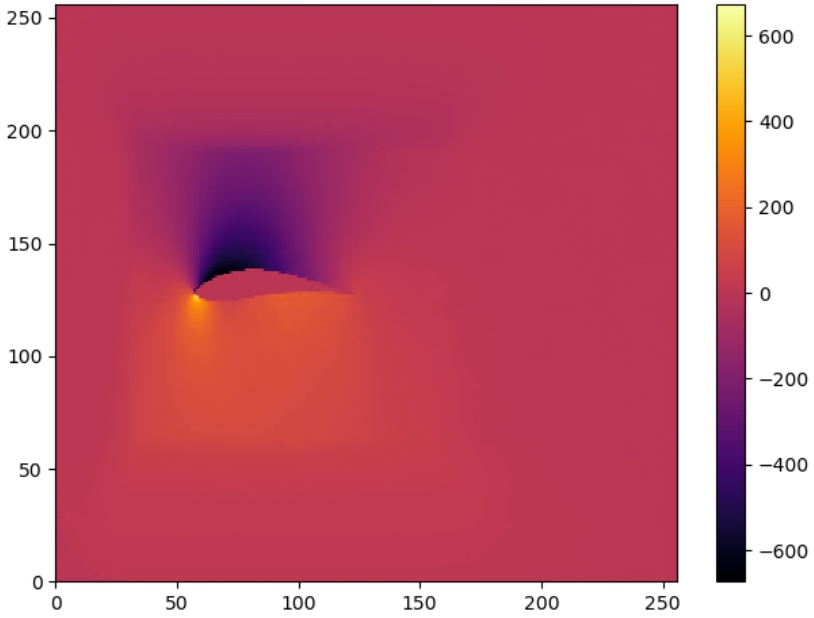

Figure A.3: P; Pressure component of the predicted field.

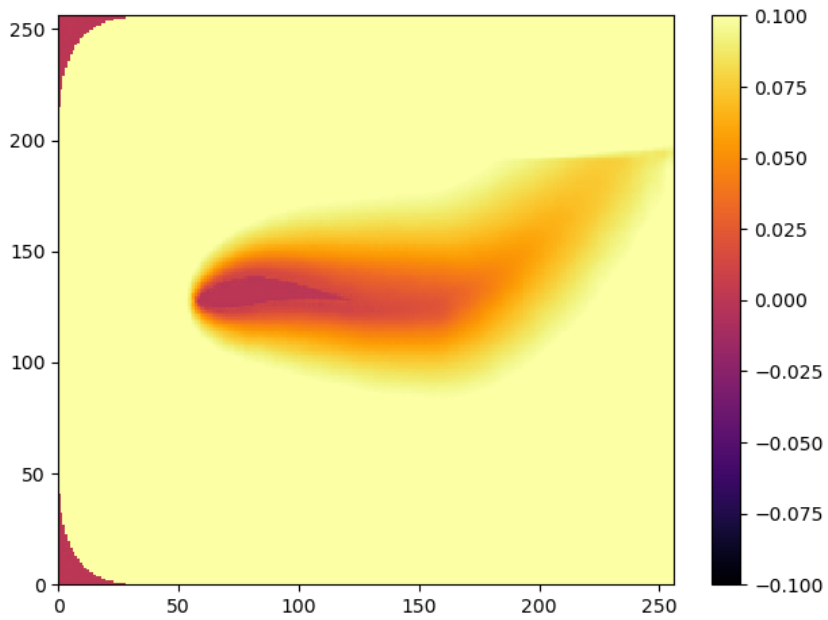

Figure A.4: ν_t component of the predicted field.

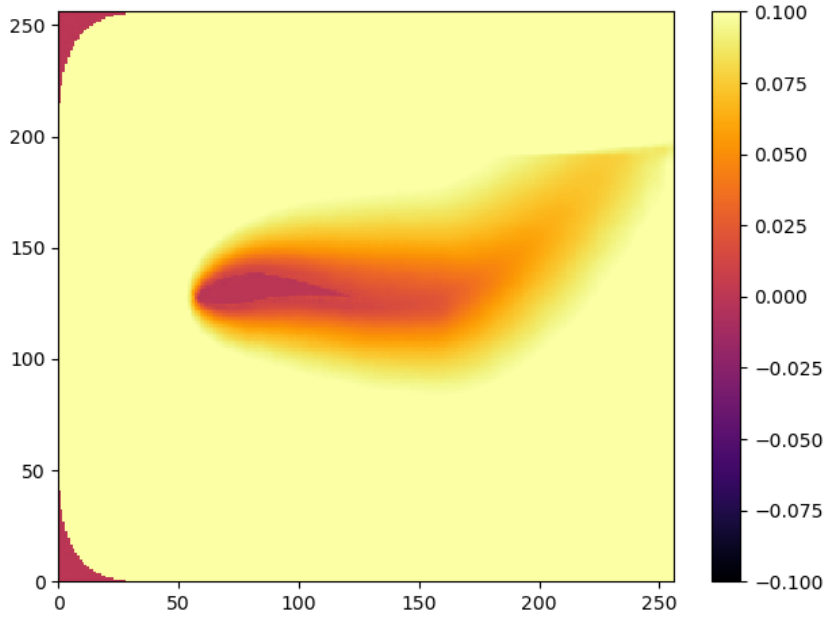

Figure A.5: $\tilde{\nu}$ component of the predicted field.

A.2 Qualitative Results of speed up of DE solvers by initial condition

This section presents the qualitative results of the inferred initial conditions and converged results in figures A.6-A.13. As described in section 3.10 the predicted results of the query points are interpolated back to the mesh points to construct the initial condition for that particular problem. These interpolated initial conditions are referred to as predicted initial condition in the figures A.6-A.13.

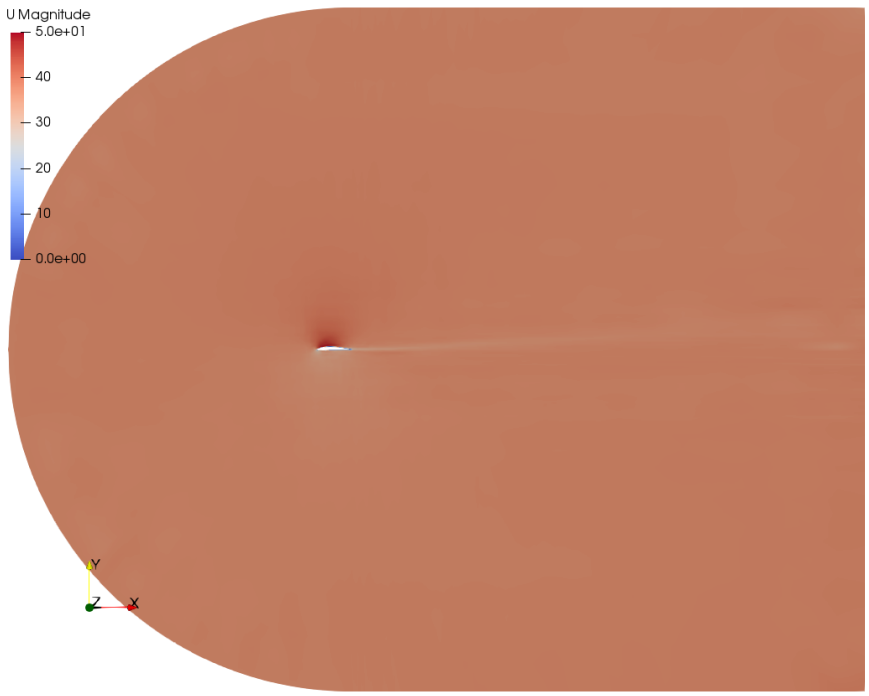

Figure A.6: Predicted initial condition for magnitude of the velocity over the whole domain.

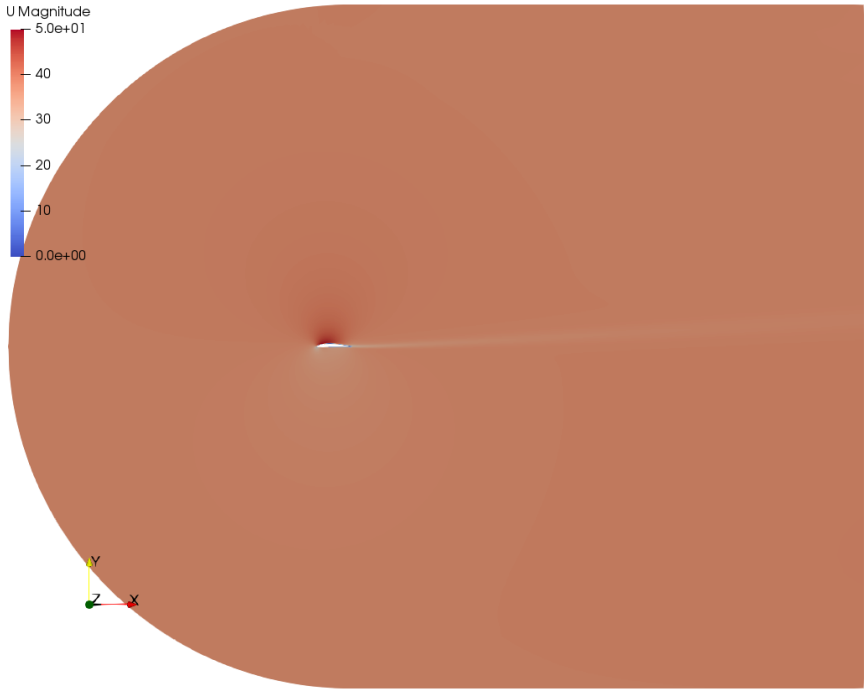

Figure A.7: Converged results for magnitude of the velocity over the whole domain.

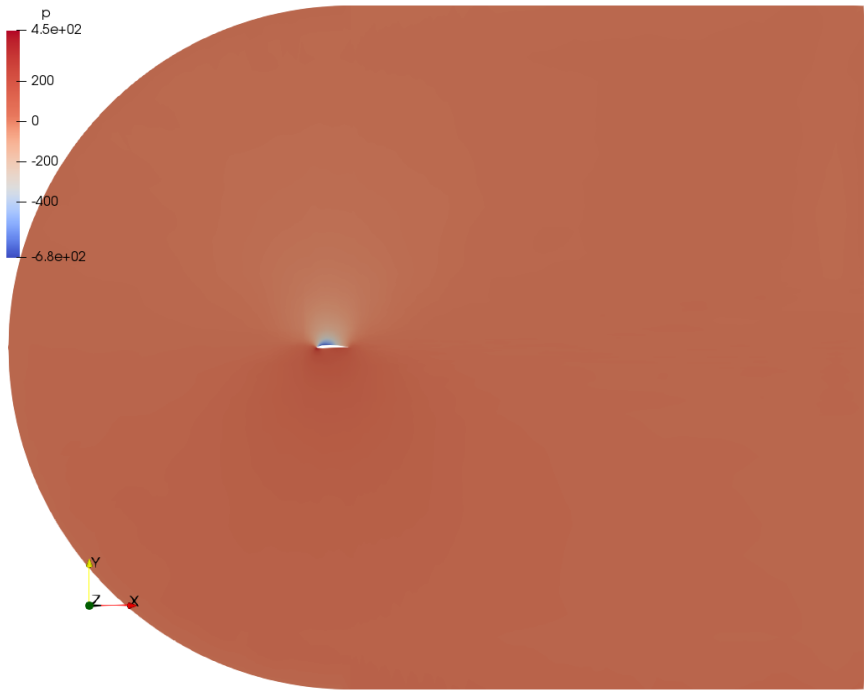

Figure A.8: Predicted initial condition for pressure over the whole domain.

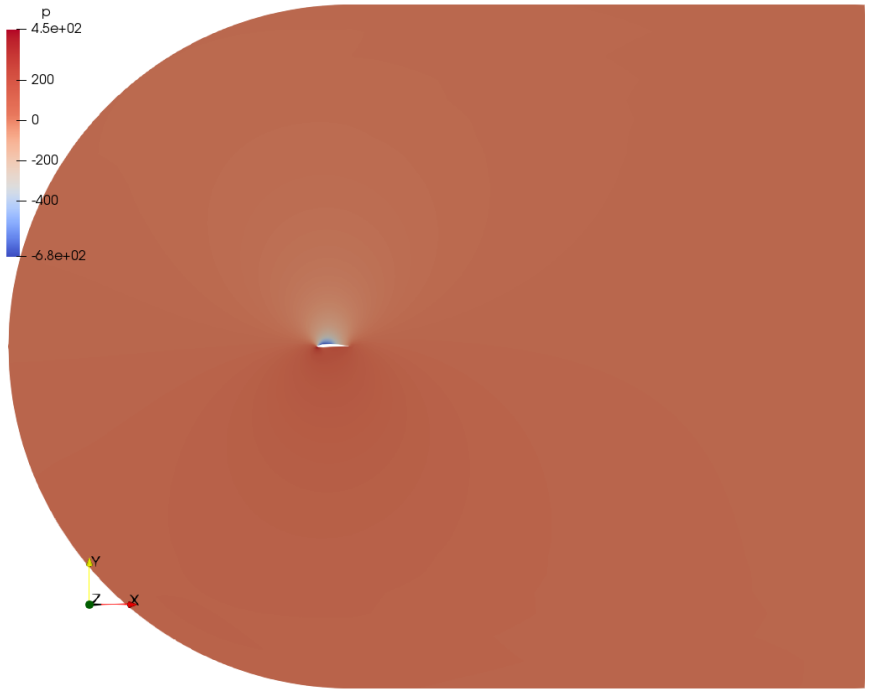

Figure A.9: Converged results for pressure over the whole domain.

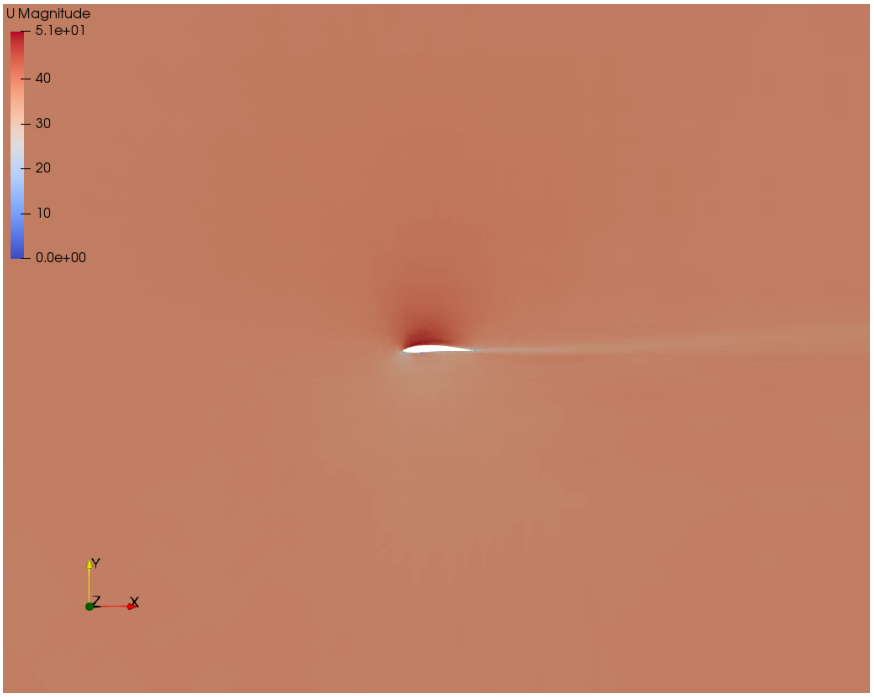

Figure A.10: Predicted initial condition for the magnitude of the velocity in the vicinity of the airfoil.

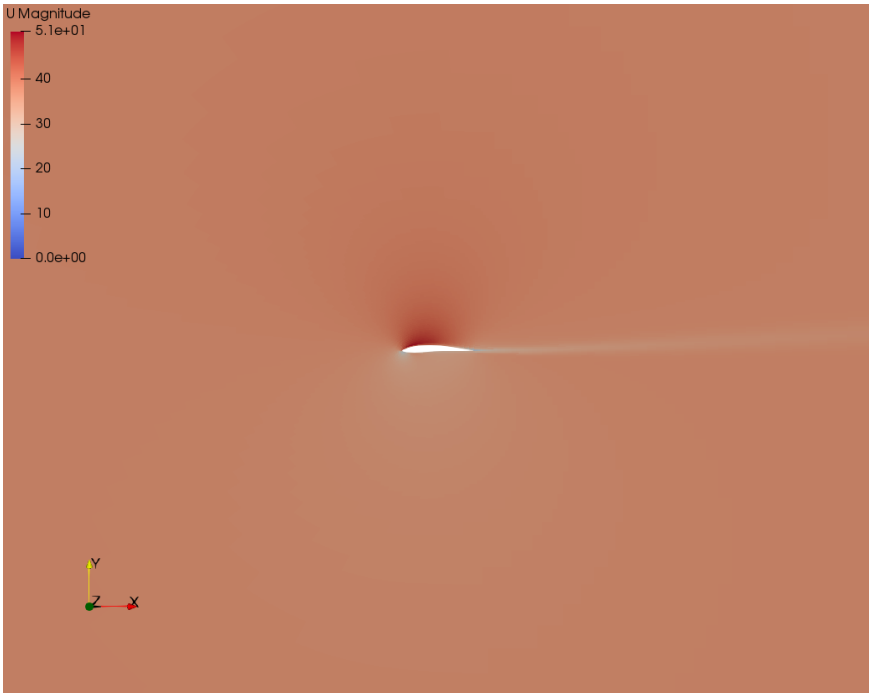

Figure A.11: Converged results for the magnitude of the velocity in the vicinity of the airfoil.

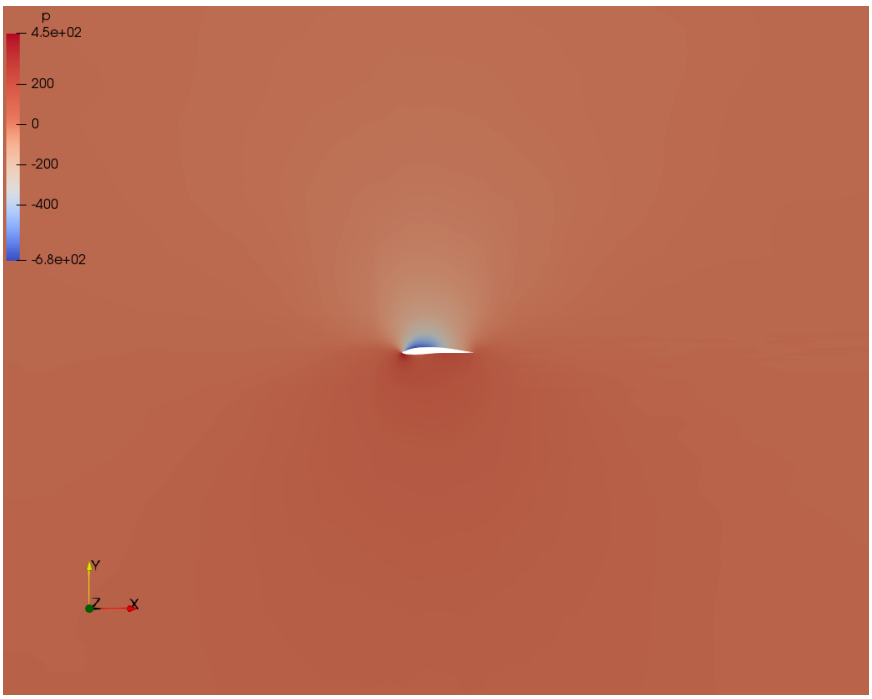

Figure A.12: Predicted initial condition for pressure in the vicinity of the airfoil.

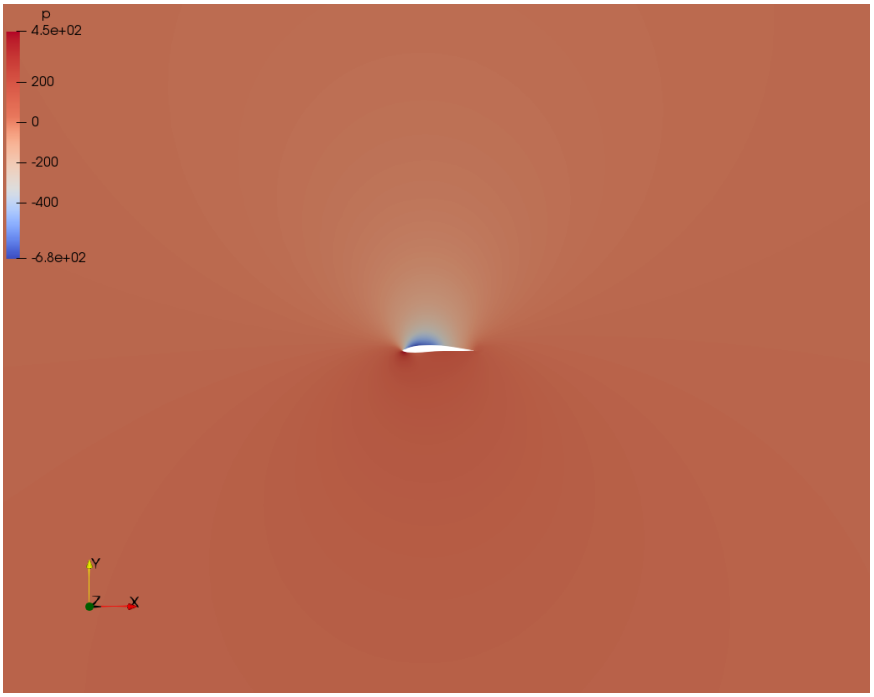

Figure A.13: Converged results for pressure in the vicinity of the airfoil.

A.3 Qualitative Results of the Parametric PINN model

In this section, the results of the PEPINN model at different time steps are presented and compared to the analytical solution starting from the initial condition up to t=1s with 0.2s step-sizes.

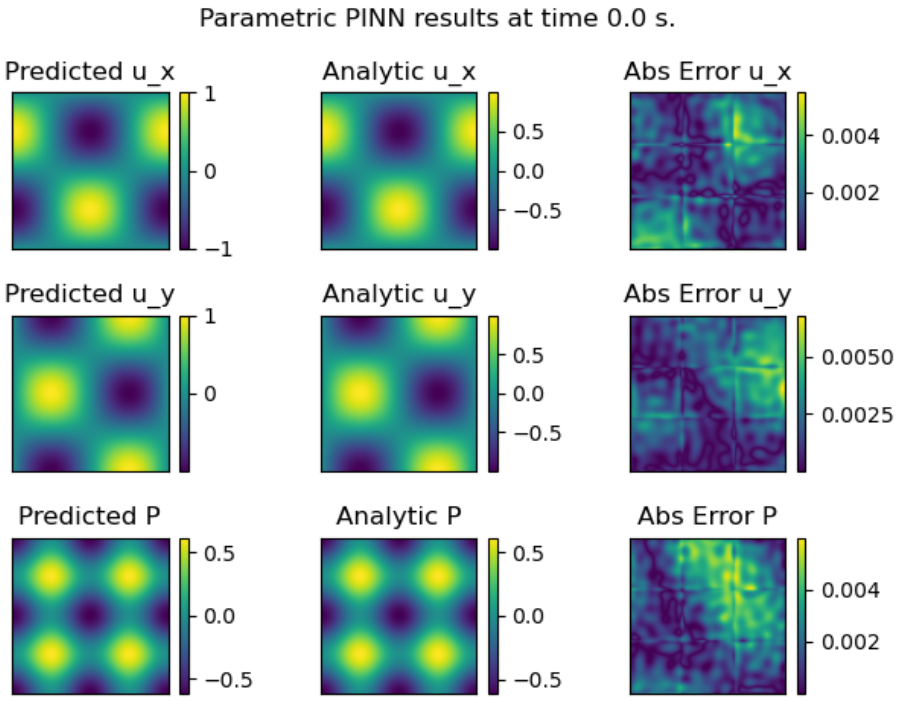

Figure A.14: Converged results for $\tilde{\nu}$ in the vicinity of the airfoil.

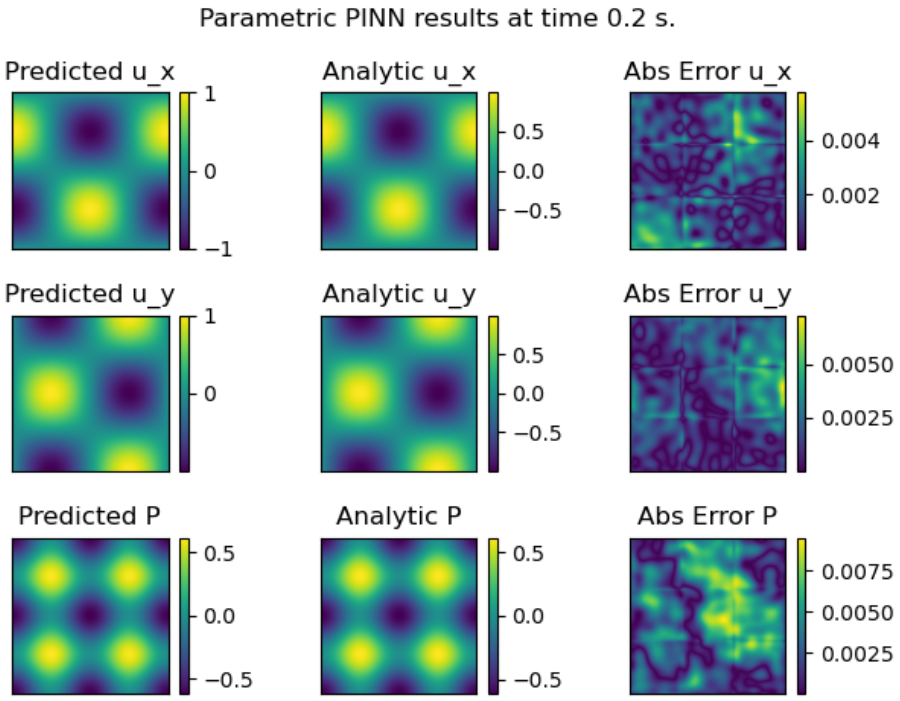

Figure A.15: Converged results for $\tilde{\nu}$ in the vicinity of the airfoil.

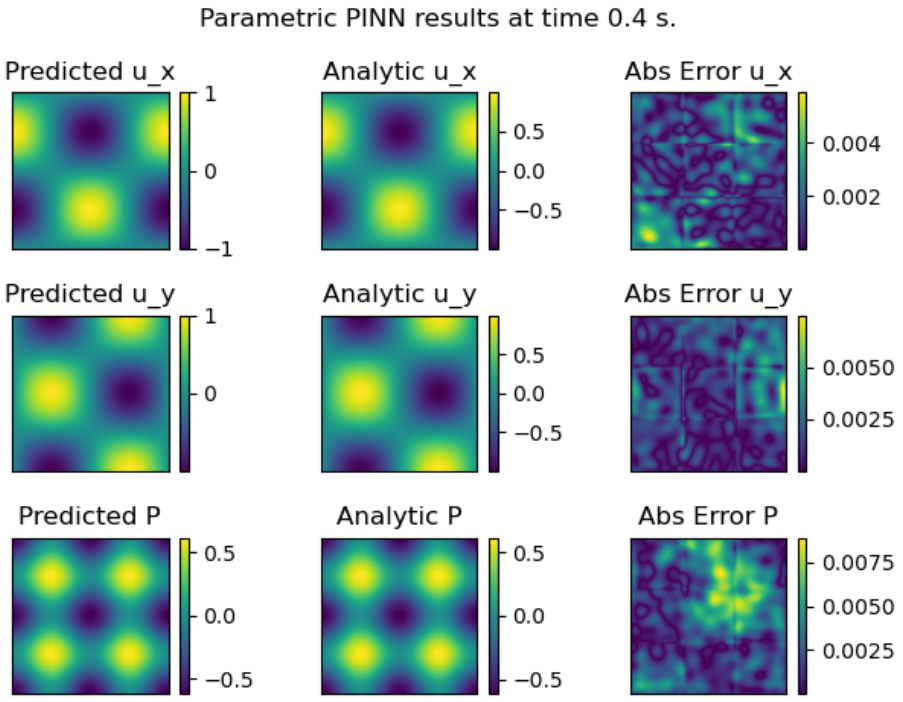

Figure A.16: Converged results for $\tilde{\nu}$ in the vicinity of the airfoil.

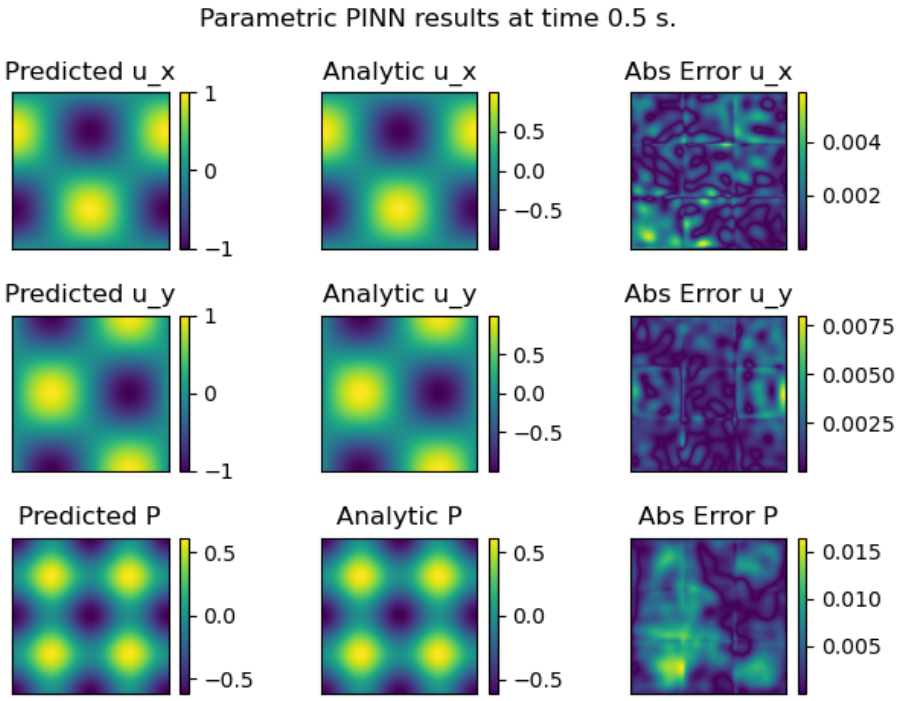

Figure A.17: Converged results for $\tilde{\nu}$ in the vicinity of the airfoil.

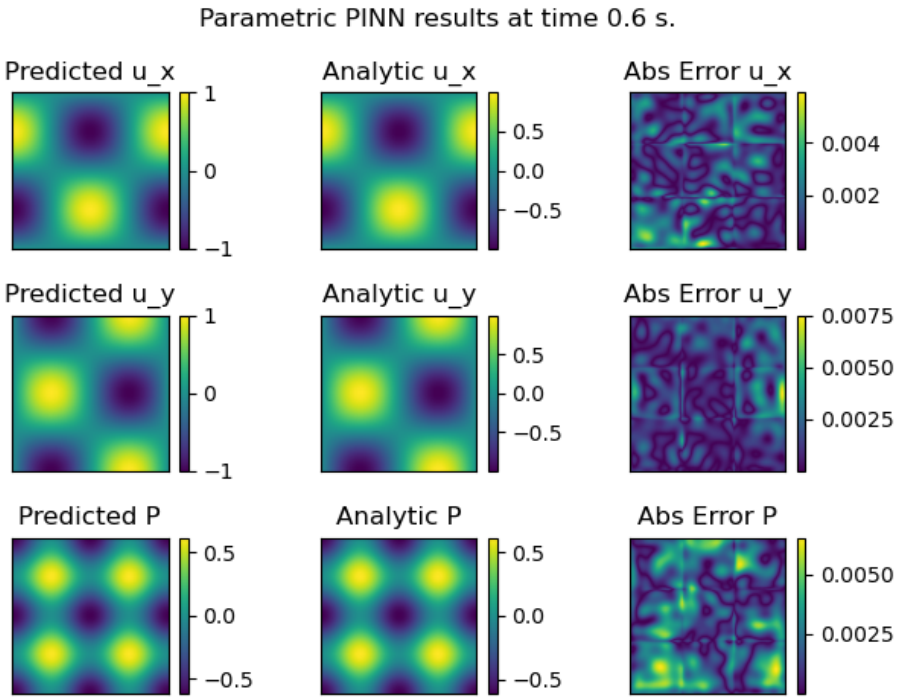

Figure A.18: Converged results for $\tilde{\nu}$ in the vicinity of the airfoil.

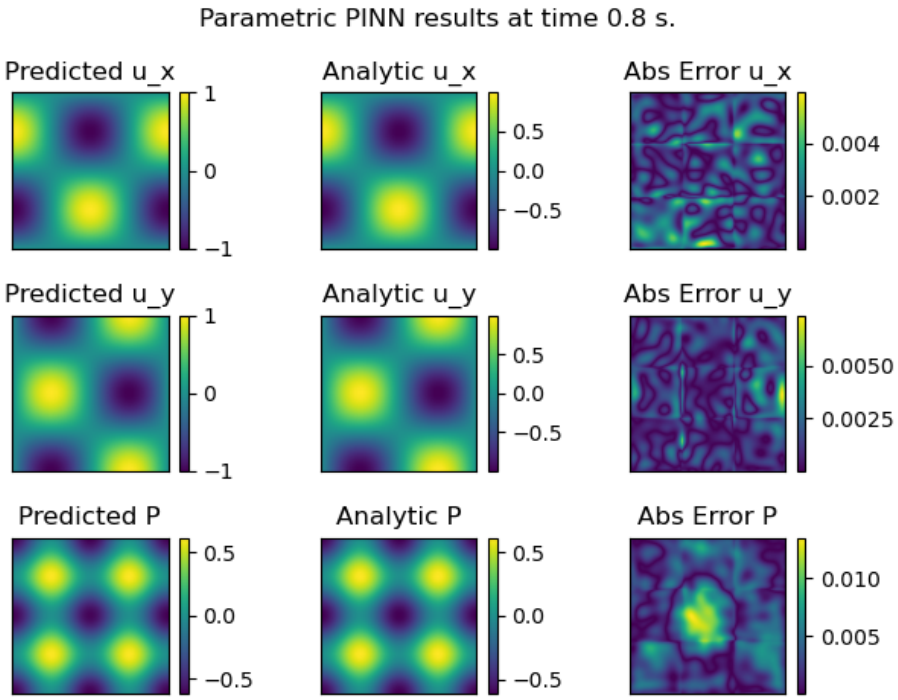

Figure A.19: Converged results for $\tilde{\nu}$ in the vicinity of the airfoil.

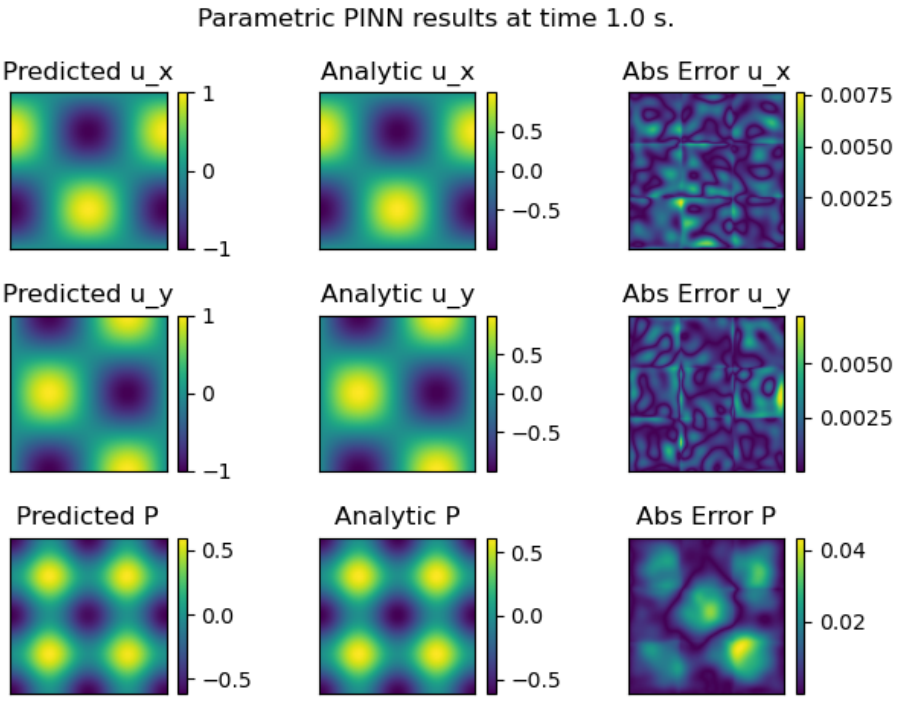

Figure A.20: Converged results for $\tilde{\nu}$ in the vicinity of the airfoil.