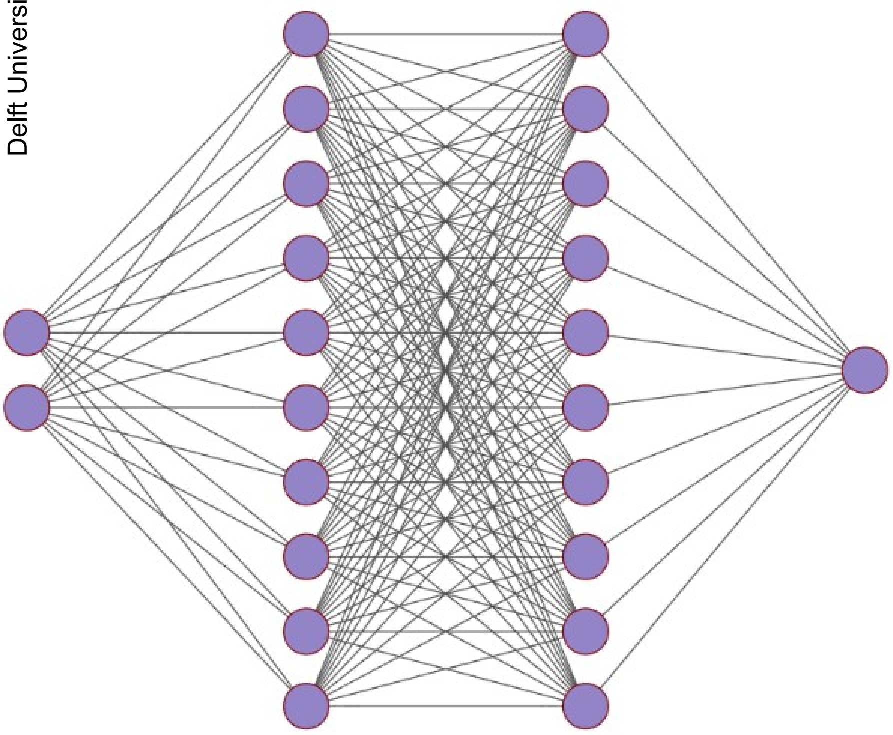
Physics-Informed Deep Learning for Computational Fluid Flow Analysis

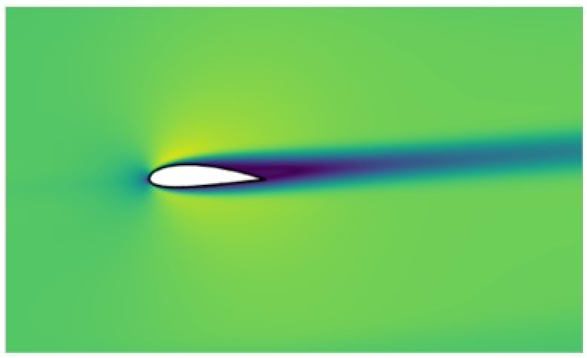
Coupling of physics-informed neural networks and autoencoders for aerodynamic flow predictions on variable geometries

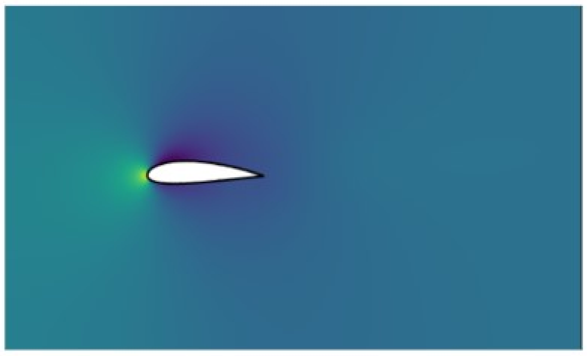
ME55035: Master Thesis Report

Author: Samarth Kakkar (3ME)

Delft University of Technology







TUDelft

Physics-Informed Deep Learning for Computational Fluid Flow Analysis

Coupling of physics-informed neural networks and autoencoders for aerodynamic flow predictions on variable geometries

by

Author: Samarth Kakkar

Student Name	Student Number
Samarth Kakkar	5274435

Academic Supervisor and Chairman (3ME TU Delft)	: Prof. Rene Pecnik
Academic Supervisor (EEMCS TU Delft)	: Dr. Matthias Möller
Industry Supervisor (Monolith AI)	: Dr. William Jennings
Committee Member (EEMCS TU Delft)	: Dr. Deepesh Toshniwal
Committee Member (3ME TU Delft)	: Dr. Jurriaan Peeters
Project Duration	: November 2021 - August 2022,
Defence Date	: 24th August 2022
Faculty	: Faculty of 3ME (Process and Energy), TU Delft

Acknowledgement

I am deeply elated as I got this opportunity to pursue my masters at an internationally distinguished university like TU Delft in my favourite field of thermo-fluids engineering. I would like to thank my parents Mr. Sanjay Kakkar and Mrs. Sangeeta Kakkar for their constant encouragement and support throughout my life and for motivating me during the difficult phases of my studies. I would also like to thank my academic supervisor, Dr. Matthias Möller for introducing me to the world of scientific machine learning through his courses and generating my interest in this field, in addition to his guidance during my master thesis. Also, I am highly obligated to my faculty supervisor Dr. Rene Pecnik for helping me enhance my skills in the fields of turbomachinery and numerical techniques for CFD through his courses as well as through his guidance during my thesis related simulations.

Author: Samarth Kakkar
Delft, August 2022

Summary

The main objective of this thesis was to explore the capabilities of neural networks in terms of representing governing differential equations, primarily in the purview of fluid/aero dynamic flows. The governing differential equations were accommodated within the loss functions for training the neural networks, thereby making them 'physics-informed'. Subsequently, this idea of physics-informed neural networks (PINNs) was extended to parameterized geometries generated with the help of commercial auto-encoders developed by the UK based company Monolith AI pvt. ltd. because neural networks have the capability to learn the desired PDEs over variable/parameterized geometries without the need to recompute the solution for every minor change in the input geometry, which proves out to be a huge advantage over classical numerical techniques. The advantages, limitations and scope for further research in the field of physics-informed deep learning have been discussed in the contents of the underlying thesis report.

Contents

1 Introduction

The past couple of decades have witnessed an exponential growth in various fields of technology, owing to the research and development in computational sciences and applied mathematics. Most of the physical systems surrounding us like electrodynamical and thermo-fluidic systems can now be simulated on computers using a system of partial differential equations, solved with the help of numerical techniques. This methodology has enhanced the scope of design, analysis and optimization of technical systems to a much greater extent while minimizing the need for building physical prototypes, saving this process solely for final testing and validation.

The domain of engineering which has been studied in this thesis is that of computational fluid dynamics (CFD), which is the process of simulating fluid/aerodynamic flows by solving the well-known Navier Stokes equations. The general workflow of a CFD simulation is illustrated in figure 1.1.

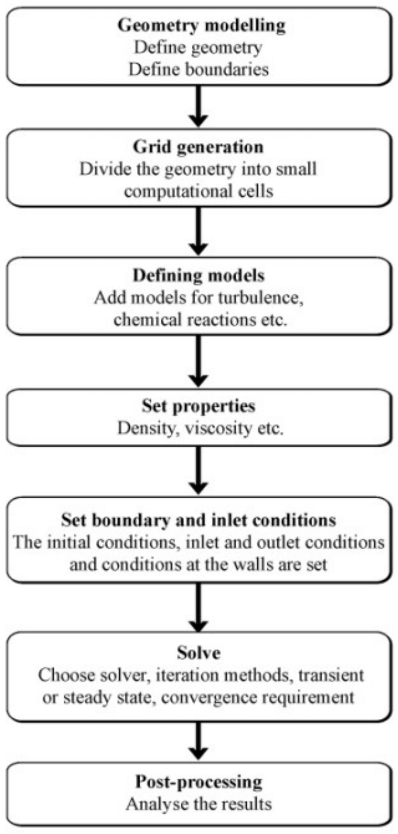

Geometry modelling
Define geometry
Define boundaries
Grid generation
Divide the geometry into small computational cells
Defining models
Add models for turbulence, chemical reactions etc.
Set properties
Density, viscosity etc.
Set boundary and inlet conditions
The initial conditions, inlet and outlet conditions and conditions at the walls are set
Solve
Choose solver, iteration methods, transient or steady state, convergence requirement
Post-processing
Analyse the results

Figure 1.1: General workflow of a CFD/Numerical Simulation
Image Source: Computational Fluid Dynamics for Engineers [2]

The algorithmic steps explained in figure 1.1 can be easily understood with the help of a sample simulation on airfoil flow which has been performed and used for validating the simulations of this thesis. The details of this simulation are explained in what follows.

1. **Geometry Modelling**: This is the starting phase of any numerical simulation where the desired geometry is prepared as an error-free CAD model, compatible to be read by the desired numerical solver, for example ANSYS Fluent.

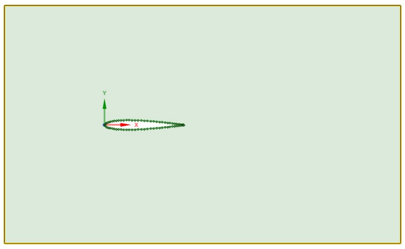


Figure 1.2: Prepared CAD geometry for airfoil simulation

2. **Grid Generation**: This is the process where the desired geometry is divided into smaller elements in order to facilitate the conversion of desired governing PDEs into a system of linear equations of the form A**x** = b, using one of the many discretization techniques such Finite Difference Methods (FDM), Finite Volume Methods (FVM) or Finite Element Methods (FEM).

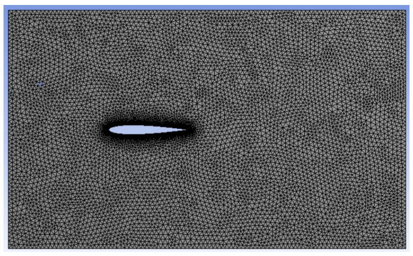
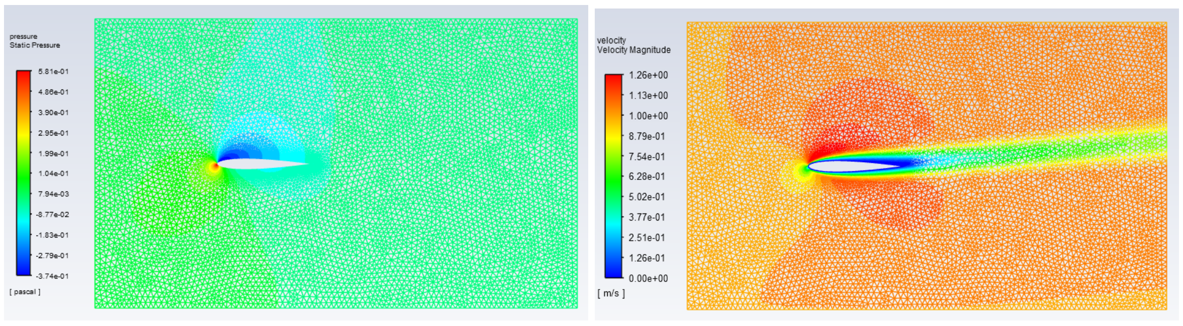


Figure 1.3: Prepared mesh/grid for airfoil simulation

3. **Defining Models and Parameters**: After successfully generating a high-quality smooth mesh or grid, we can setup our simulation by defining a suitable turbulence model/laminar flow setting and the fluid flow parameters like compressibility condition, gas model, viscosity model, etc.
4. **Setting Up Boundary Conditions**: Any solution method (analytical or numerical) for a system of PDEs cannot provide the complete problem specific solution without providing it with the appropriate boundary conditions, and in case of time dependent problems the initial conditions. For example, in CFD simulations, we usually specify the velocity magnitudes at flow inlets, the static pressure values at flow outlets and the conventional no slip boundary condition persists on the wall boundaries.
5. **Solver Schemes**: After prescribing all the boundary conditions and the associated flow parameters, the linear system of equations generated via discretization of the flow domain and linearization of the nonlinear terms if any, can be solved by either direct or iterative numerical schemes. Iterative solver schemes are preferred over direct solvers for large problems due to computational memory limitations. The most common iterative solver algorithms used in CFD are the SIMPLE algorithm and the pressure velocity coupled algorithm.
6. **Post Processing**: After the solver scheme has converged to a desirable error limit, we can extract and compute the desired quantitative variables like lift and drag and plot various solution contours like velocity and pressure.



(a) Pressure contours

(b) Velocity contours

Figure 1.4: Results after post processing

It can be observed that the above mentioned methodology for simulating physics via numerical simulation has an inherent dependence on the grid generated after the discretization, which directly influences the system of linear equations derived from the discretized governing PDEs. Hence, classical numerical simulation techniques cannot accommodate even very minute changes in the input geometry, compelling the need to perform a new simulation for each geometry.

The past couple of decades have witnessed an advent rise in the popularity of data-driven techniques to perform various technological and scientific tasks. The ever-increasing computational hardware and software capabilities have made it possible for computers to perform tasks which conventionally require abstract thinking at human level rather than some quantitative formulations which could be hard-coded in computers. These developments in the field of artificial intelligence have generated vast technological leaps in the field of computer vision, voice/speech recognition, image classification, object detection and so on. One of the main advantages of data-driven techniques is their ability to generate outputs almost instantaneously for new inputs after sufficient training, without human intervention. This capability is now starting to attract the communities involved in scientific computing to explore these data-driven techniques either as a replacement or as an aid to the classical numerical methods. In the framework of classical numerical methods, redundant and repetitive simulations are needed to be performed even for minute changes in geometry, domain shapes or initial and boundary conditions, making the iterative design process computationally very expensive. With a fully trained data-driven pipeline, we expect to remove the need for such time consuming simulations while simultaneously ensuring the physical validity of the results generated. We restrict ourselves to the deep learning domain for our current study, many of which have been summarised by Calzolari and Liu [4]. The distinction among artificial intelligence (AI), machine learning (ML) and deep learning (DL) can be appreciated from figure 1.5

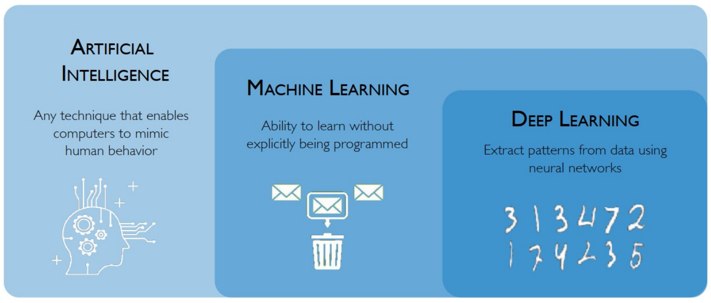

Artificial Intelligence
Any technique that enables computers to mimic human behavior
Machine Learning
Ability to learn without explicitly being programmed
Deep Learning
Extract patterns from data using neural networks

Figure 1.5: Distinction among AI, ML and DL
Image Source: MIT Deep Learning 6.S191 [1]

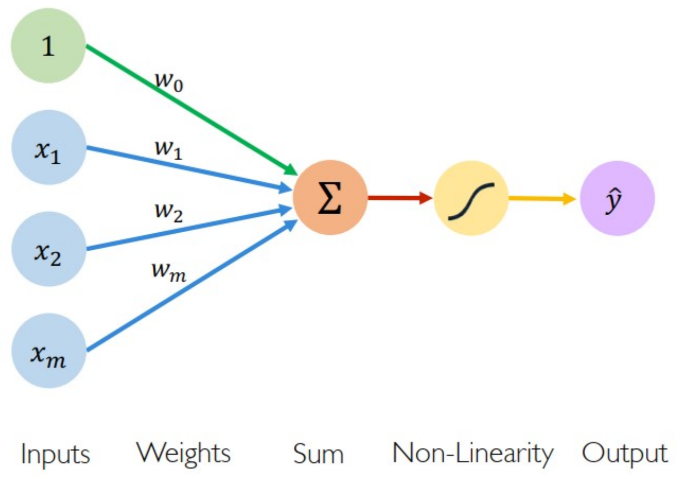
As mentioned before, the main strength of data-driven techniques like deep learning is their ability to map a set of input and output spaces, as an abstract relationship is learned during the training process for a deep learning pipeline, without needing to know the relationship in advance. This advantage in the field of image/speech recognition turns into a disadvantage while dealing with scientific computing, where maintaining a physically consistent relationship between the inputs and outputs defined by governing differential equations is of utmost importance. Therefore, the main challenge that the research community is striving to overcome in the field of scientific machine learning (SciML) is to integrate the correct physical relationships into deep learning mechanisms and side by side utilising the potential of reduced redundancy in deep learning methods compared to numerical simulations. A large variety of deep learning architectures have been explored in this thesis pertaining to fluid flow simulations. In a broad sense, these architectures are used mainly for two major tasks: generation of domains and geometries of interest, and solution of governing differential equation on the generated domains.

In this thesis, an extensive study has been performed to explore the capabilities and limitations of physics-informed neural networks (PINNs) with parameterized geometry inputs either as a replacement or as an aid to classical numerical techniques for simulating physical systems. In chapter 2, we study the theory behind PINNs, whereas chapter 3 provides insights into an advanced methodology of PINNs called operator learning, which is capable of learning a general operator as opposed to a fixed PDE. In chapter 4, we explore various techniques of generative modelling which can be used to generate parameterized forms of complex geometries, suitable to be fed as inputs to PINNs. Subsequently in chapters 5, 6 and 7, we perform simulations with PINNs for self-parameterized geometry parameters over Poisson, Stokes and Navier Stokes equations, respectively. Finally we conclude with chapter 8 where we combine generative modelling techniques for abstract geometry parameterization of cambered airfoils via autoencoders with the PINN framework for Navier Stokes equations. The conclusions and recommendations with some insightful discussion of the results found while pursuing this thesis have also been discussed in chapter 9. The simulation codes developed for numerical simulations as well as PINN models have been provided in the respective appendices.

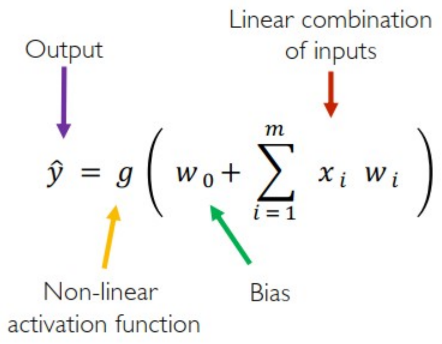
2 Theory on Physics Informed Neural Networks (PINNs)

2.1. Deep Neural Networks

Deep neural networks form the fundamental architecture of deep learning. It can be argued that they take inspiration from a human nervous system where a biological neuron is replaced by a perceptron as follows:



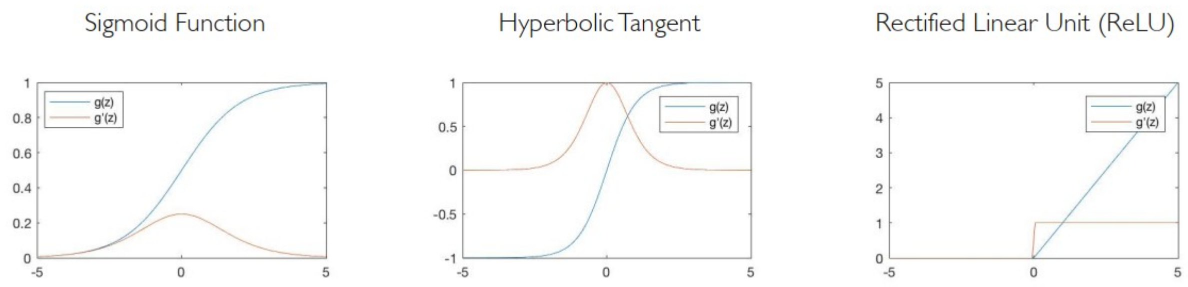
(a) Perceptron


Output
Linear combination of inputs
$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$
Non-linear activation function
Bias

(b) Output evaluation in a neuron/perceptron of a neural network

Figure 2.1: Structure of perceptron
Image Source: MIT Deep Learning 6.S191 [1]

It can be inferred from figure 2.1 that the inputs $x_1, x_2, ..., x_m$ are multiplied with a respective weight value $w_1, w_2, ..., w_m$ and added together along-with an additional bias w_0, and the result is fed into a nonlinear function called an activation function g. This sequence of operations maps the given inputs to the output $\hat{y}$. Most common activation functions used in deep learning are sigmoid, relu, tanh (hyperbolic tangent) etc. These nonlinear activation functions inherently provide the neural networks capability to represent complex nonlinear functions. Details of these activation functions have been given in figure 2.2.



$$g(z) = \frac{1}{1 + e^{-z}} \qquad g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \qquad g(z) = \max(0, z)$$

$$g'(z) = g(z)(1 - g(z)) \qquad g'(z) = 1 - g(z)^2 \qquad g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Figure 2.2: Commonly used activation functions
Image Source: MIT Deep Learning 6.S191 [1]

A large number of these perceptrons subsequent to the input layer form a hidden layer. The number of perceptrons in a hidden layer define its 'width'. A neural network as shown in figure 2.3, with a single hidden layer, input layer and output layer is called a shallow neural network. When all the perceptrons of each layer have connections to all the corresponding inputs/previous hidden layers, then such a neural network is called fully connected neural network

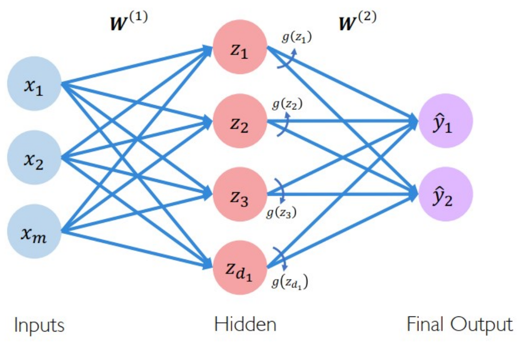


Figure 2.3: A fully connected shallow neural network
Image Source: MIT Deep Learning 6.S191 [1]

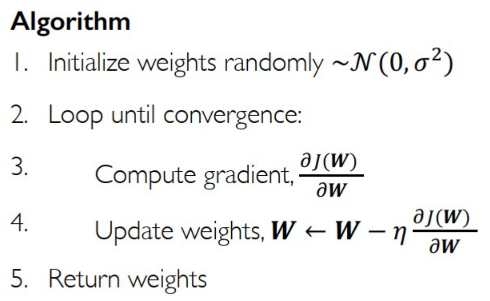
We can also stack multiple hidden layers in between the input layer and output layer. Such a neural network having more than one hidden layer is called deep neural network, and the number of hidden layers in such a neural network define the 'depth' of the neural network. One of the main reasons for the success of fully connected deep neural networks in modern data-driven techniques comes from the following theorem given by Cybenko [10] in 1989.

Universal Approximation Theorem: A shallow neural network can be used to approximate any continuous function to any desired precision.

This capability of shallow neural networks to be able to approximate any continuous function has brought them at the forefront of all data-driven methods. The main challenge that still poses one of the biggest questions in the world of deep learning is to figure out the desired number of neurons in the hidden layer of a shallow network, in order to be able to approximate a particular function with a desired accuracy. Moreover, switching to deep (more than one hidden layer) neural networks can also make it easier to train complex functions with steep gradients [25]. The depth and width of a neural network are among the most important hyperparameters to which detailed studies can be dedicated, although we do not pursue that direction of research in our current work.

2.1.1. Training of Neural Networks

The weights and biases of a neural network (w_i for $i = 0, 1, ..., m$) are the variables of the neural network which are randomly initialised and need to be trained in order to be able to correctly map the inputs to outputs in a desired manner. This training procedure may be supervised (with the help of some available data with known output labels) or unsupervised (without any help from data). The ultimate aim of the training process of a neural network is to minimize a quantity called loss function, which defines a measure of deviation of the neural network output from the true/desired output. This can be framed as an optimization problem with the sole objective to minimize the loss function. The most widely used optimization algorithm for this task is based on the gradient descent principle, which is summarised as follows:



Algorithm

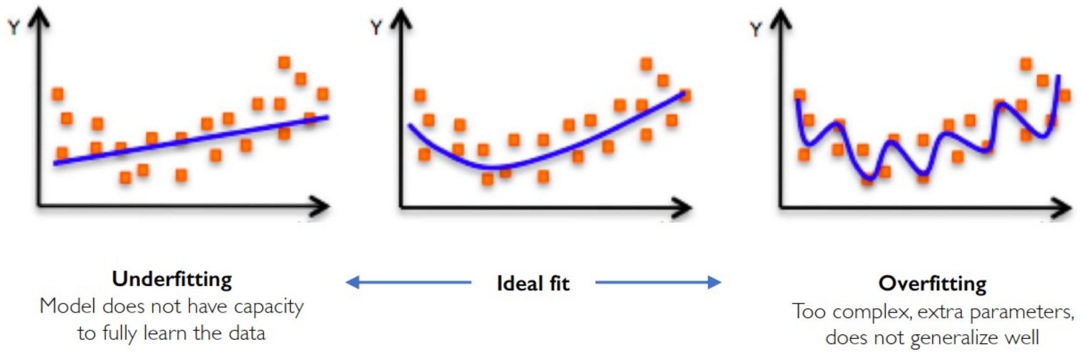
```
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3.       Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4.       Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights
```

Figure 2.4: Gradient descent algorithm for minimizing loss function ***J*** by updating weights ***W***
Image Source: MIT Deep Learning 6.S191 [1]

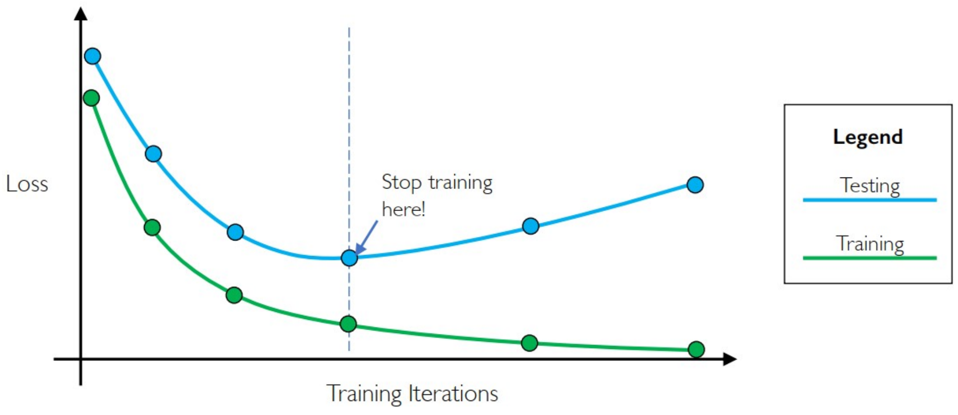
In figure 2.4, the parameter η is called the learning rate, which defines the stride length which the algorithm takes in the direction opposite to the gradient. A large η may lead to faster convergence but it can be prone to overstepping the actual minima, leading to oscillations in the loss value above the local minima. A small η will ensure stability of the algorithm but it may prolong the convergence. Various techniques have been developed to modify this learning rate during the training process to balance the interplay of accelerated convergence along with stability. Some of these techniques are Nesterov acceleration, momentum generation, learning rate scheduling etc., which can be studied in detail from the work of Aston Zhang et al. [44]. The most commonly used gradient descent algorithm in deep learning is the Adam optimizer, which uses an optimal combination of all the techniques mentioned above for learning rate modification.
Another class of optimizers that can be used instead of gradient descent comes from Newton's optimization algorithms, where the direction of descent is inferred by multiplying the gradient of the objective function with an inverse hessian, provided it is positive definite [31]. Newton's methods exhibit quadratic rate of convergence, but are computationally expensive. This led to the development of a new class of algorithms called quasi-Newton optimizers, which ensure a balance between accelerated convergence and computational efficiency [31]. The most popular quasi-Newton method which is used in deep learning is the L-BFGS algorithm. The mathematical studies of this class of algorithms is beyond the scope of this work, but interested readers can refer to the textbook on numerical optimization by Nocedal and Wright [31] for a detailed explanation of such algorithms.

2.1.2. Challenges for Overfitting and Underfitting of Neural Networks

The main aim of the neural network training process is to minimize the loss function, such that the neural network becomes expressive enough to represent the correct output label/value for its corresponding input, belonging to the training set as well as outside the training set. This means that the neural network should be able to generalise for any data corresponding to its domain of application whether it has been seen in the training process or not. If we train the neural network with sufficient amount of data for large number of epochs/iterations such that the loss function residuals decrease to very low values, the phenomenon of overfitting the data set comes into picture, where the neural network learns to represent all the data points within the training set with very low errors but it does not produce the desired output for unseen input values. Therefore, we need to minimize the loss without losing the ability to properly handle the unseen data. This is ensured by simultaneously testing the neural network on data which has not been used in training. As long as the errors for both the test data as well as training data are reducing, we continue the training process. When the testing error starts to increase while the training error is still reducing, we conclude the onset of overfitting in our training process and stop the training of the neural network any further. This phenomenon is called early stopping, which is visually represented in the following figure 2.5.



(a) Moving from underfitting to overfitting (left to right) during the training process



(b) Early Stopping

Figure 2.5: Underfitting, overfitting and early stopping
Image Source: MIT Deep Learning 6.S191 [1]

2.2. Physics-Informed Neural Networks (PINNs)

Physics-informed neural networks are deep learning architectures which use neural networks to solve partial differential equations. These networks harness the power of automatic differentiation, which can evaluate derivatives of the outputs of neural networks with respect to their inputs. The derivatives evaluated using automatic differentiation are machine precision accurate and do not require any discretisation. Therefore, such a PINN model that works on point-wise values is called the collocation-based PINN, which can be used to evaluate the output for any resolution of grid after training without the need for being retrained, making the entire framework mesh independent, unlike most classical numerical techniques.

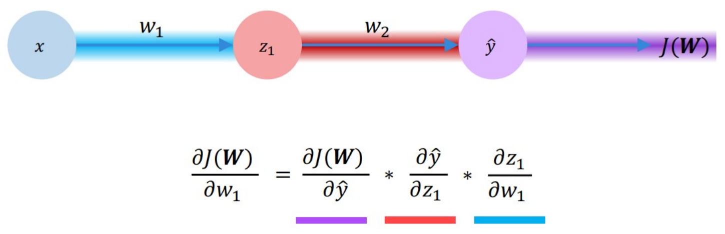

x
w_1
z_1
w_2
$\hat{y}$
$J(\boldsymbol{W})$
$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$

Figure 2.6: Backpropagation for automatic differentiation
Image Source: MIT Deep Learning 6.S191 [1]

As illustrated in figure 2.6, the derivative of the loss J is evaluated with respect to weight w_1. Similarly, the derivative of the neural network output $\hat{y}$ with respect to an input x can be evaluated using backpropagation, which is the repeated application of the chain rule. This ability of evaluating derivatives can be used to construct PDE residuals which can be used as a loss metric to train the PINN framework in a semi-supervised manner (when loss is a combination of PDE residual and mean squared error in boundary conditions and/or initial conditions, along with some solution data if available). A typical architecture for a PINN framework involving Burgers' equation is illustrated in figure 2.7.

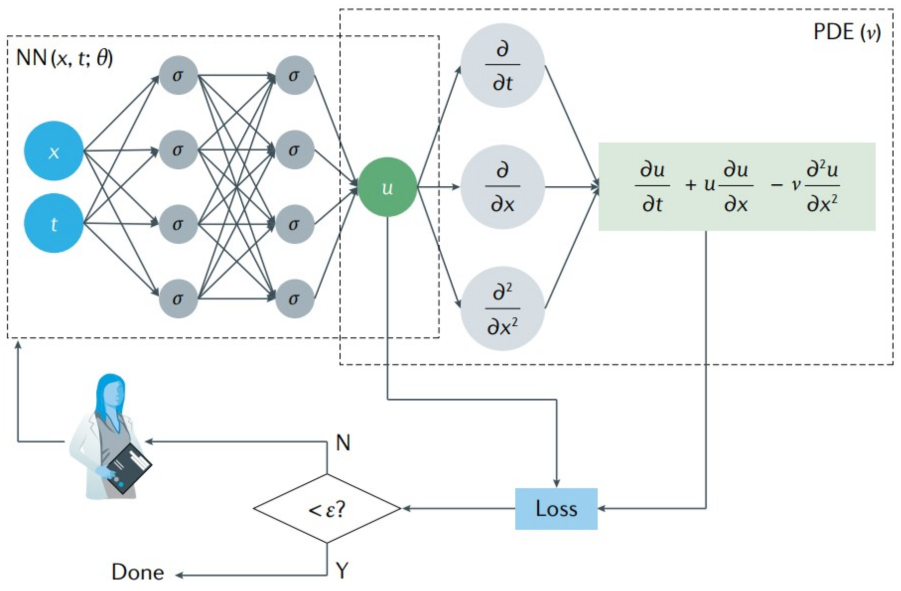

NN (x, t; θ)
x
t
σ
u
PDE (ν)
$\frac{\partial}{\partial t}$
$\frac{\partial}{\partial x}$
$\frac{\partial^2}{\partial x^2}$
$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2}$
Loss
$< \varepsilon ?$
N
Y
Done

Figure 2.7: PINN architecture for Burgers' equation
Image Source: Karniadakis et al. [19]

One of the first papers that brought PINNs into limelight was written by Raissi et al. [34] in 2019. This work was later transformed into an open-source python library called DeepXDE [25]. Many such open source libraries for PINNs were developed after this such as SciANN [13], NeuroDiffEq [5], Elvet [3], NVIDIA Simnets (now known as Modulus) [14] and Julia based NeuralPDE [46] to name a few. The main advantage of using data-driven techniques like PINNs is that not only can they be used to solve PDEs (referred to as forward problem in literature), but also for inferring the differential equations from the given solution data. This line of work was presented as deep hidden physics by Raissi [33] and hidden fluid mechanics [35]. The study on inverse PINNs is not undertaken in our work, whereas the idea of forward PINNs has been explored in great detail.

2.2.1. Training of PINNs

One of the main attractions for using PINNs to solve PDEs comes from its ability to utilise a combination of available data as well as available knowledge of physical laws and constraints. The availability of data may range from the small data regime, where only the initial and boundary value solutions are known beforehand along with the governing differential equations, or the big data regime where data is available in such an abundance that one might consider skipping the incorporation of physical laws without losing any generalisation capabilities of the neural network [19]. In the practical world of computational science and engineering, we usually lie in the 'small' data regime (figure 2.8) or at best in the 'some' data regime where we have limited availability of training data due to computational limitations of classical numerical techniques used in engineering simulations.

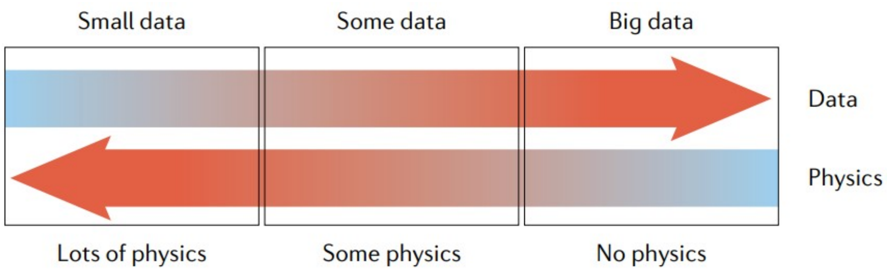


Figure 2.8: Data regimes for PINNs
Image Source: Karniadakis et al. [19]

The process of making a deep learning algorithm physically consistent involves the introduction of various biases into the framework such as observational biases, inductive biases and learning biases[19]. The details of these biases are discussed as follows:

1. **Observational Biases:** this kind of biasing simply involves exposing the deep learning frame work to more and more data, with the assumption that the larger the variety of data that is seen during the training process, the better would be the generalisation capability of the network. It assumes that as the real world data inherently agrees with the principles of physics, such physics can be integrated into the neural networks with the help of 'big' data. The most fundamental limitation of this approach is the cost of generation of such huge amounts of data.
2. **Inductive Biases:** this kind of biasing involves the generation/manipulation of neural network architectures in such a manner that suits the kind of physics it is being used to simulate. For example, inductive biasing was used by Cranmer et al. [9] to simulate a system of 4 particles interacting with each other. A corresponding graph neural network can be generated with exactly 4 nodes (mimicking those 4 particles) and the edges representing their mutual interactions as shown in figure 2.9.

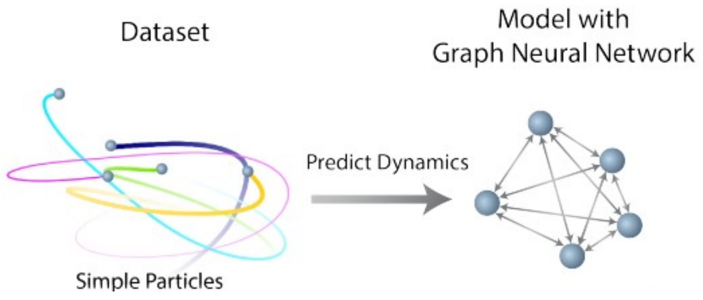


Figure 2.9: Design of graph neural network for particle interactions with inductive biasing
Image Source: Cranmer et al. [9]

3. **Learning Bias:** the method of inductive biasing has inherent limitations that we would need to develop a new deep learning architecture to simulate a problem from every distinct field of physics, and sometimes the prior knowledge of physics might not be sufficient to introduce inductive biasing. Therefore we resort to a new school of thought termed as learning biasing, where we focus on developing training algorithms which can simultaneously accommodate the available data as well as the physical laws by penalising the loss functions, on a limited set of neural network architectures (fully connected, convolutional and recurrant neural networks etc.). This is the most favourable method of biasing due to the enhanced degree of freedom embedded in it. From this point forward, we focus our study on the simulation of physical systems (mostly pertaining to fluid flows) using this form of biasing.

2.2.2. Loss Functions for PINNs

As discussed in the previous subsection 2.2.1 that in learning bias, we need to accommodate the available training data as well as the knowledge of physics into the loss function of a neural network. We take an example of the Burgers' equation in a dimensional coordinate (x) and time coordinate (t) to solve for velocity (u). The Burgers' equation is given as:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}. \tag{2.1}$$

The corresponding loss functions can be given by the following expressions:

$$\mathcal{L}_{\text{total}} = w_{\text{data}}\mathcal{L}_{\text{data}} + w_{\text{pde}}\mathcal{L}_{\text{pde}}, \tag{2.2}$$

where,

$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}}\sum_{i=1}^{N_{\text{data}}}(u(x_i,t_i) - u_i)^2,$$

$$\mathcal{L}_{\text{pde}} = \frac{1}{N_{\text{pde}}}\sum_{j=1}^{N_{\text{pde}}}\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2}\right)^2_{x_j,t_j},$$

Here, we have two sets of points termed as data points x_i, t_i, including initial/boundary condition points and some training data if available, and collocation points x_j, t_j which lie in the interior of the domain, where the PDE needs to be evaluated. The weights w_{data} and w_{pde} which map the intensity of contribution of their respective loss components, are also important hyperparameters which affect the convergence of the PINN training process. Their impact on convergence would be studied in later sections of this study.

2.2.3. Applications of PINNs

Burgers' Equation

One of the first results using PINN approach for the Burgers' equation was given by Raissi et al. [34] for the following set of initial and boundary conditions:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \frac{0.01}{\pi}\frac{\partial^2 u}{\partial x^2} \quad : x \in [-1,1], \;\; t \in [0,1], \tag{2.3}$$

$$u(0,x) = -sin(\pi x),$$
$$u(t,-1) = u(t,1) = 0.$$

For the training process by Raissi et al. [34], the number of collocation points were set to 10000. The quasi-Newton based L-BFGS optimizer was used to train the neural network. A parametric study was performed by varying the number of hidden layers, neurons per layer and the number of data points for the PINN, but the results represented in figure 2.10 have been obtained with 9 hidden layers, 20 neurons per layer and 100 data points corresponding to initial and boundary conditions. 'tanh' was used as activation function for all neurons.

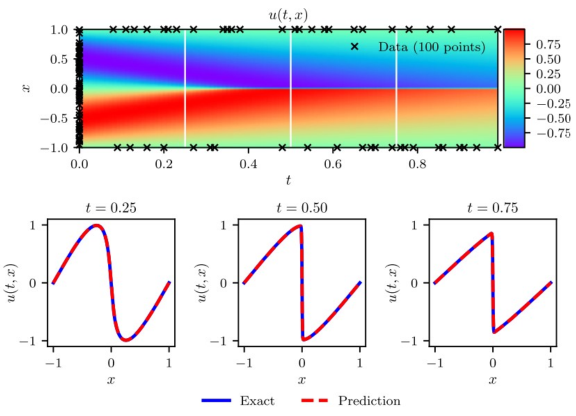


Figure 2.10: Burgers' equation solved using PINN
Image Source: Raissi et al. [34]

A parametric study was performed by varying the number of data points and the dimensions of the neural network, the results of which have been summarized in figure 2.11. It is evident from the results that the predictive accuracy increases by increasing all these parameters.

N_u \ N_f	2000	4000	6000	7000	8000	10000
20	2.9e−01	4.4e−01	8.9e−01	1.2e+00	9.9e−02	4.2e−02
40	6.5e−02	1.1e−02	5.0e−01	9.6e−03	4.6e−01	7.5e−02
60	3.6e−01	1.2e−02	1.7e−01	5.9e−03	1.9e−03	8.2e−03
80	5.5e−03	1.0e−03	3.2e−03	7.8e−03	4.9e−02	4.5e−03
100	6.6e−02	2.7e−01	7.2e−03	6.8e−04	2.2e−03	6.7e−04
200	1.5e−01	2.3e−03	8.2e−04	8.9e−04	6.1e−04	4.9e−04

(a) Total loss with variation in number of data points

Layers \ Neurons	10	20	40
2	7.4e−02	5.3e−02	1.0e−01
4	3.0e−03	9.4e−04	6.4e−04
6	9.6e−03	1.3e−03	6.1e−04
8	2.5e−03	9.6e−04	5.6e−04

(b) Total loss with variation in hidden layers and neurons

Figure 2.11: Results of the parametric study by varying the neural network hyperparameters to study their impact on the final minimum value of the loss function attained after the training process for the Burgers' equation
Image Source: Raissi et al. [34]

Navier Stokes Equation (NSFNets)

One of the first attempts to solve Navier Stokes equations using PINNs was by Jin et al. [18] by developing a library called NSFNets (Navier Stokes Flow nets). This framework of PINNs is capable of solving incompressible laminar as well as turbulent flows in fully unsupervised manner (the only training data provided is for initial and boundary conditions). One of the key advantages of employing PINNs for Navier Stokes equations comes from the use of automatic differentiation for evaluating analytical derivatives with machine precision accuracy, which makes the framework independent from the resolution of the input collocation and boundary points, unlike classical numerical methods, where the resultant system of the discretized linear equations is a direct consequence of the input mesh/grid. Therefore, the discretization errors in the form of numerical diffusion and dispersion errors can be avoided using this approach. Two variants of Navier Stokes equations are simulated using this approach for comparison in terms of convergence as well as accuracy: the VP formulation and the VV formulation. The VP formulation of Navier Stokes evaluates the instantaneous velocity and pressure fields for the corresponding space and time inputs using the vanilla form of Navier Stokes equations. Similarly the VV formulation evaluates the vorticity field and the velocity field for the space and time inputs corresponding to the vorticity equation. The VP formulation for the Navier Stokes reads:

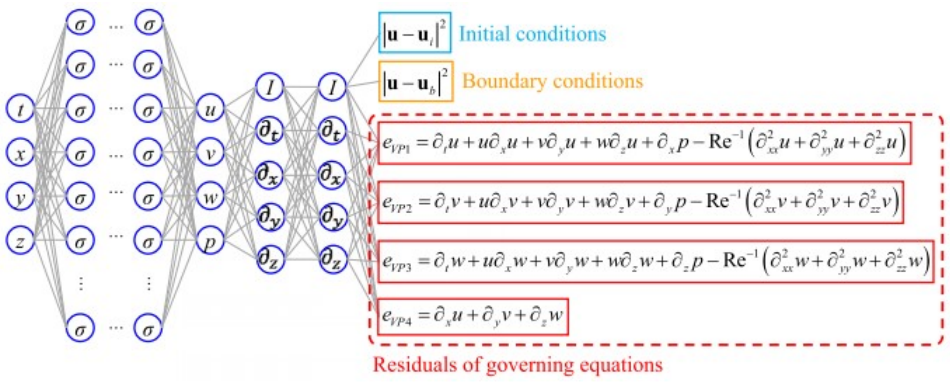
$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}.\nabla)\mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2 \mathbf{u} \quad \text{in } \Omega \quad \text{(momentum)}, \tag{2.4}$$

$$\nabla.\mathbf{u} = 0 \quad \text{in } \Omega \quad \text{(continuity)}, \tag{2.5}$$

$$\mathbf{u} = \mathbf{u}_\Gamma \quad \text{on } \Gamma_D, \tag{2.6}$$

$$\frac{\partial \mathbf{u}}{\partial n} = 0 \quad \text{on } \Gamma_N. \tag{2.7}$$

Here, all the variables t, u and p are in dimensionless form, normalised using their respective reference conditions such as U_{ref}, D_{ref} for $Re = U_{\text{ref}} D_{\text{ref}}/\nu$



t
x
y
z
σ
u
v
w
p
I
∂_t
∂_x
∂_y
∂_z
$|\mathbf{u}-\mathbf{u}_i|^2$ Initial conditions
$|\mathbf{u}-\mathbf{u}_b|^2$ Boundary conditions
$e_{VP1} = \partial_t u + u\partial_x u + v\partial_y u + w\partial_z u + \partial_x p - \mathrm{Re}^{-1}(\partial^2_{xx} u + \partial^2_{yy} u + \partial^2_{zz} u)$
$e_{VP2} = \partial_t v + u\partial_x v + v\partial_y v + w\partial_z v + \partial_y p - \mathrm{Re}^{-1}(\partial^2_{xx} v + \partial^2_{yy} v + \partial^2_{zz} v)$
$e_{VP3} = \partial_t w + u\partial_x w + v\partial_y w + w\partial_z w + \partial_z p - \mathrm{Re}^{-1}(\partial^2_{xx} w + \partial^2_{yy} w + \partial^2_{zz} w)$
$e_{VP4} = \partial_x u + \partial_y v + \partial_z w$
Residuals of governing equations

Figure 2.12: PINN framework for VP (velocity and pressure) formulation of Navier Stokes equations using NSFNets
Image Source: Jin et al. [18]

The fully connected network used for VP formulation can be inferred from figure 2.12. The neural network uses 'tanh' activation functions. The loss function for this framework is represented as follows:

$$L = L_e + \alpha L_b + \beta L_i, \tag{2.8}$$

where,

$$L_e = \frac{1}{N_e}\sum_{i=1}^{4}\sum_{n=1}^{N_e} |e^n_{VPi}|^2,$$

$$L_b = \frac{1}{N_b}\sum_{n=1}^{N_b} |\mathbf{u}^n - \mathbf{u}^n_b|^2,$$

$$L_i = \frac{1}{N_i}\sum_{n=1}^{N_i} |\mathbf{u}^n - \mathbf{u}^n_i|^2.$$

Here, L_e, L_b and L_i represent the pde loss, boundary condition loss and initial condition loss respectively. N_e, N_b and N_i are the number of collocation/interior points, boundary points and initial condition points. $\mathbf{u}^n_b$ and $\mathbf{u}^n_i$ are the given velocity vector data for boundary condition and initial condition, whereas e_{VPi} represents the residual for its respective equation in the system. α and β are the weights for their respective loss terms which have been dynamically updated during the training process using the strategy proposed by Wang et al. [38], to obtain faster convergence. For the k^{th} iteration, the weights α^k and β^k can be modified as follows:

$$\alpha^{k+1} = (1-\lambda)\alpha^k + \lambda\hat{\alpha}^{k+1}, \qquad \beta^{k+1} = (1-\lambda)\beta^k + \lambda\hat{\beta}^{k+1},$$

where,

$$\hat{\alpha}^{k+1} = \frac{max_\theta|\nabla_\theta L_e|}{|\nabla_\theta \alpha^k L_b|}, \qquad \hat{\beta}^{k+1} = \frac{max_\theta|\nabla_\theta L_e|}{|\nabla_\theta \beta^k L_i|}.$$

The mathematical details of this technique for updating the dynamic weights is beyond the scope of this work, but it is essential to note that another hyperparameter λ is introduced in this framework. There is no theory behind estimating an appropriate value for λ, however a value of λ = 0.1 has been used in this case, which shows promising results. Moreover, to further accelerate the convergence, residual based adaptive refinement (RAR) of collocation points was implemented [25],[27]. The concept behind RAR is to add more collocation points during the training process in regions with higher residual values. The solution accuracy can increase by upto 50 percent by using RAR in NSFNets. The analogous PINN framework for the VV formulation is presented in figure 2.13, whereas the system of equations describing the VV formulation are given as follows:

$$\frac{\partial \omega}{\partial t} + \nabla \times (\omega \times \mathbf{u}) = -\frac{1}{Re}\nabla \times \nabla \times \omega \quad \text{in } \Omega, \tag{2.9}$$

$$\nabla^2 \mathbf{u} = -\nabla \times \omega \quad \text{in } \Omega, \tag{2.10}$$

$$\omega = \nabla \times \mathbf{u} \quad \text{on } \Gamma, \tag{2.11}$$

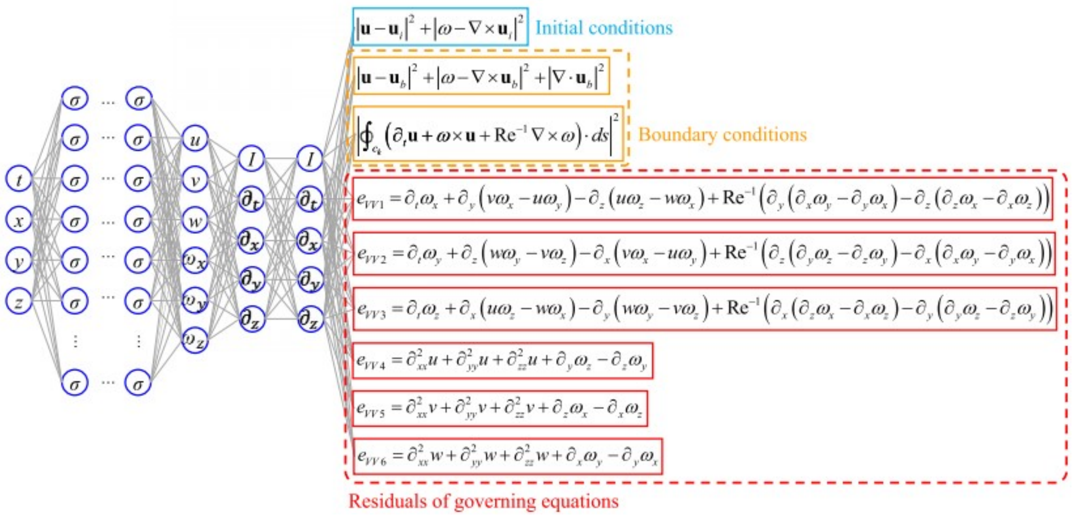
$$\oint_{c_k} \left(\frac{\partial \mathbf{u}}{\partial t} + \omega \times \mathbf{u} + \frac{1}{Re}\nabla \times \omega \right) . ds = 0, \quad k = 1, ..., q, \tag{2.12}$$

$$\mathbf{u} = \mathbf{u}_\Gamma \quad \text{on } \Gamma_D, \tag{2.13}$$

$$\frac{\partial \mathbf{u}}{\partial n} = 0 \quad \text{on } \Gamma_N, \tag{2.14}$$

$$\nabla . \mathbf{u} = 0 \quad \text{at one point on } \Gamma, \tag{2.15}$$

$$\omega = \nabla \times \mathbf{u} \quad \text{at } t = 0 \text{ in } \Omega. \tag{2.16}$$



$|\mathbf{u}-\mathbf{u}_i|^2 + |\omega - \nabla\times\mathbf{u}_i|^2$
Initial conditions
$|\mathbf{u}-\mathbf{u}_b|^2 + |\omega - \nabla\times\mathbf{u}_b|^2 + |\nabla\cdot\mathbf{u}_b|^2$
$\left|\oint_{c_k} \left(\partial_t \mathbf{u} + \omega\times\mathbf{u} + \mathrm{Re}^{-1}\nabla\times\omega\right)\cdot ds\right|^2$
Boundary conditions
$e_{VV1} = \partial_t\omega_x + \partial_y(v\omega_x - u\omega_y) - \partial_z(u\omega_z - w\omega_x) + \mathrm{Re}^{-1}(\partial_y(\partial_x\omega_y - \partial_y\omega_x) - \partial_z(\partial_z\omega_x - \partial_x\omega_z))$
$e_{VV2} = \partial_t\omega_y + \partial_z(w\omega_y - v\omega_z) - \partial_x(v\omega_x - u\omega_y) + \mathrm{Re}^{-1}(\partial_z(\partial_y\omega_z - \partial_z\omega_y) - \partial_x(\partial_x\omega_y - \partial_y\omega_x))$
$e_{VV3} = \partial_t\omega_z + \partial_x(u\omega_z - w\omega_x) - \partial_y(w\omega_y - v\omega_z) + \mathrm{Re}^{-1}(\partial_x(\partial_z\omega_x - \partial_x\omega_z) - \partial_y(\partial_y\omega_z - \partial_z\omega_y))$
$e_{VV4} = \partial^2_{xx}u + \partial^2_{yy}u + \partial^2_{zz}u + \partial_y\omega_z - \partial_z\omega_y$
$e_{VV5} = \partial^2_{xx}v + \partial^2_{yy}v + \partial^2_{zz}v + \partial_z\omega_x - \partial_x\omega_z$
$e_{VV6} = \partial^2_{xx}w + \partial^2_{yy}w + \partial^2_{zz}w + \partial_x\omega_y - \partial_y\omega_x$
Residuals of governing equations
t x y z
σ
u v w ω_x ω_y ω_z
I ∂_t ∂_x ∂_y ∂_z

Figure 2.13: PINN framework for VV (vorticity and velocity) formulation of Navier Stokes equations using NSFNets *Image Courtsey: Jin et al.* [18]

The corresponding loss function is given by:

$$L = L_e + \alpha L_b + \beta L_i, \tag{2.17}$$

where,

$$L_e = \frac{1}{N_e}\sum_{i=1}^{6}\sum_{n=1}^{N_e} |e^n_{VVi}|^2,$$

$$L_b = \frac{1}{N_b}\sum_{n=1}^{N_b} \left(|\mathbf{u}^n - \mathbf{u}^n_b|^2 + |\omega^n - \nabla \times \mathbf{u}^n_b|^2 + |\nabla . \mathbf{u}^n_b|^2 \right),$$

$$L_i = \frac{1}{N_i}\sum_{n=1}^{N_i}\left(|\mathbf{u}^n - \mathbf{u}_i^n|^2 + |\omega^n - \nabla \times \mathbf{u}_i^n|^2\right),$$

where, ω^n denotes the vorticity for the n^{th} data point, the domain is q-multiply connected and c_ks are the q independent contours.

The results for 3D unsteady Beltrami flow solved using both the formulations of NSFNets have been illustrated in figure 2.15. Two stage optimization was used by starting with the Adam optimizer followed by the L-BFGS optimizer. It was found that the VV formulation provides an order of magnitude more accurate results compared to the VP formulation for this case, which has been illustrated in figure 2.16

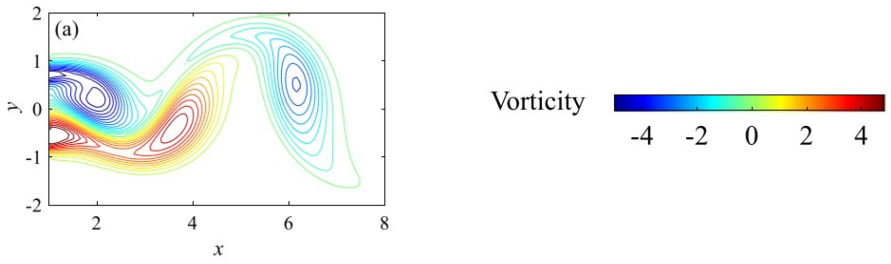
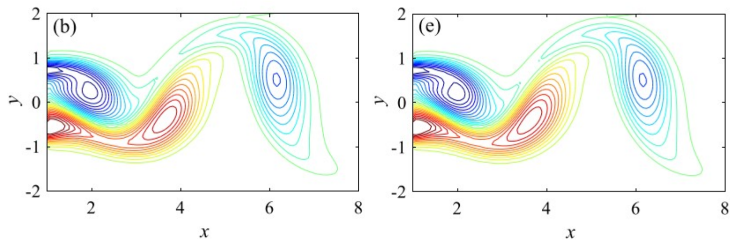


Figure 2.14: DNS solution for Beltrami flow (t=1.0, z=0) for reference
Image Source: Jin et al. [18]



(a) Results for VP **(b)** Results for VV

Figure 2.15: Comparsion of velocity-pressure (VP) and vorticity-velocity (VV) formulations for Navier Stokes equations solved using NSFNets for Beltrami Flow
Image Source: Jin et al. [18]

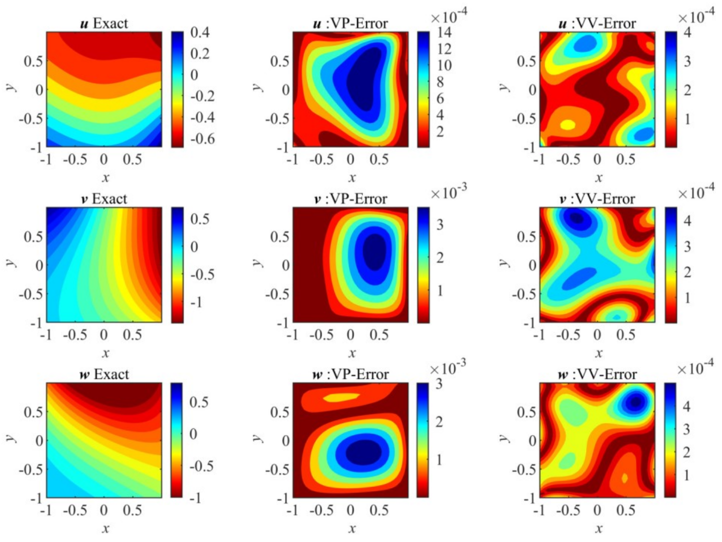


Figure 2.16: Errors for Beltrami flow obtained for VP and VV formulations of Navier Stokes equations using NSFNets
Image Source: Jin et al. [18]

Next, it was tested if NSFNets (only the VP formulation) are capable of sustaining turbulent flows for a pipe at Re = 1000, for a subdomain of 190 x 200 x 210 in wall units. A local convective time unit of simulation was defined as T_c^+=L_x^+/$U(y)_{min}$ (L_x^+ was the domain size in streamwise direction) whose value was set at 12. 20000 collocation points were used along with 6644 boundary/initial points. The results have been illustrated in figure 2.17, which show that NSFNets are capable of sustaining turbulence with a good agreement with reference data.

One of the most important training strategies which is useful for turbulent flows is *transfer learning*. This is useful if one needs to change the Reynolds number of the flow for a fully trained neural network. It was found that by initialising the network on the trained weights for a particular flow conditions (Reynolds number) and further training them for a different Reynolds number is a computationally much faster process compared to training from scratch for each Reynolds number.

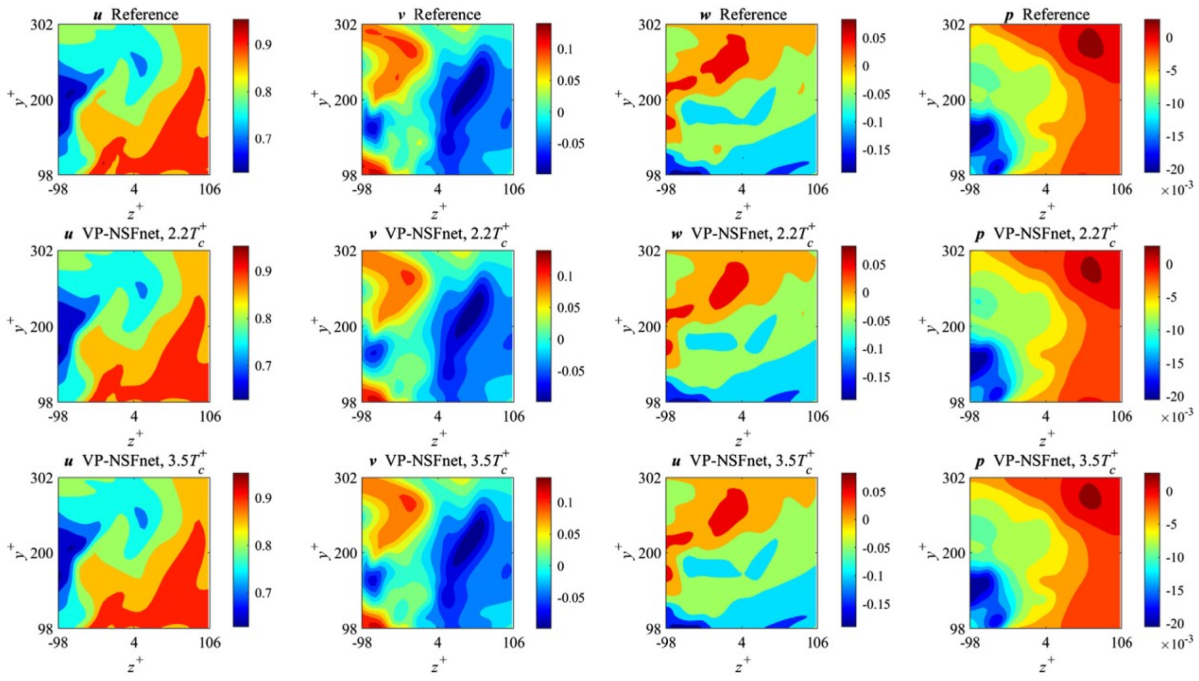


Figure 2.17: Results obtained by using NSFNets on turbulent Beltrami flow using velocity-pressure (VP) formulation of Navier Stokes equations
Image Source: Jin et al. [18]

Vortex Induced Vibrations

Vortex induced vibrations (VIV) are caused in a solid structure exposed to external fluid flow as a result of the shed vortices from the fluid. For example, a flow past a cylinder generates Von Karman vortex shedding, which in turn imposes resultant lift and drag forces, causing the cylinder to oscillate. This phenomenon typically occurs at a Strouhal number (fL/U, f being the frequency of vortex shedding, L being the characteristic length scale and U being the free stream velocity) of 0.2, and was investigated with the help of a PINN frame work by Raissi et al. [36] in the laminar flow regime. In this framework, one way fluid-structure interaction (FSI) was employed to study the resultant oscillations of a cylinder imposed by the flow field surrounding it. A 2D flow was studied over an elastically mounted cylinder, which was free to move only in the cross stream (y coordinate) direction, which was the primary direction for VIV. The flow field surrounding it was an incompressible flow governed by the Navier Stokes, with the streamwise direction being the x coordinate. This translated to a classical mass-spring-damper system with the cylinder displacement being given as η.

$$\rho\eta_{tt} + b\eta_t + k\eta = f_L, \tag{2.18}$$

where ρ, b and k are the mass, damping coefficient and stiffness respectively. f_L gives the fluid lift force which forces the VIV to occur. Four variables were evaluated namely u, v, p and η, therefore a system of four equations was needed to close the problem. This was achieved by tracking the transport of a given scalar quantity c in the flow field along with the Navier Stokes equations.

$$u_t + uu_x + vu_y = -p_x + Re^{-1}(u_{xx} + u_{yy}), \tag{2.19}$$

$$v_t + uv_x + vv_y = -p_y + Re^{-1}(v_{xx} + v_{yy}) + \eta_{tt}, \tag{2.20}$$

$$u_x + v_y = 0, \tag{2.21}$$

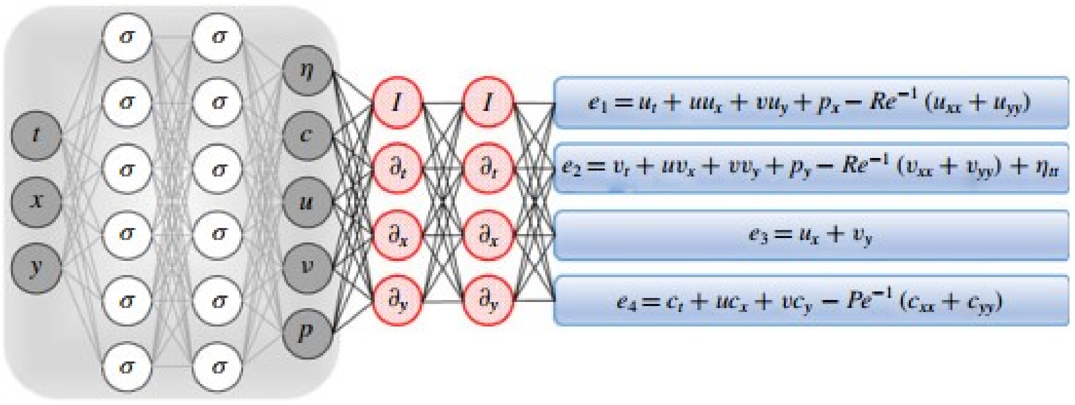
$$c_t + uc_x + vc_y = Pe^{-1}(c_{xx} + c_{yy}), \tag{2.22}$$

where Pe represents the Peclet number, which is defined as LU/D, where L is the characteristic length scale, U is the flow velocity and D is the mass-diffusion coefficient. Given the data for transport of c, the velocities and pressure were inferred by using a PINN framework. The lift and drag forces for the given velocity and pressure data can be evaluated as follows:

$$F_D = \oint \left(-pn_x + 2Re^{-1}u_x n_x + Re^{-1}(u_y + v_x)n_y\right) ds, \tag{2.23}$$

$$F_L = \oint \left(-pn_y + 2Re^{-1}v_y n_y + Re^{-1}(u_y + v_x)n_x\right) ds. \tag{2.24}$$

Here, (n_x, n_y) represent the outward normal for a cylinder at point (x, y), while ds is the arc length on the cylinder surface. The neural network shown in figure 2.18 used the 'sine' activation functions and the Adam optimizer for training. The dimensions of the neural network were chosen to be 10 hidden layers and 160 neurons per layer. The training data in the loss function consisted of initial and boundary conditions for the velocities and some measured data for the scalar concentrations c^n and the cylinder displacements η^n.

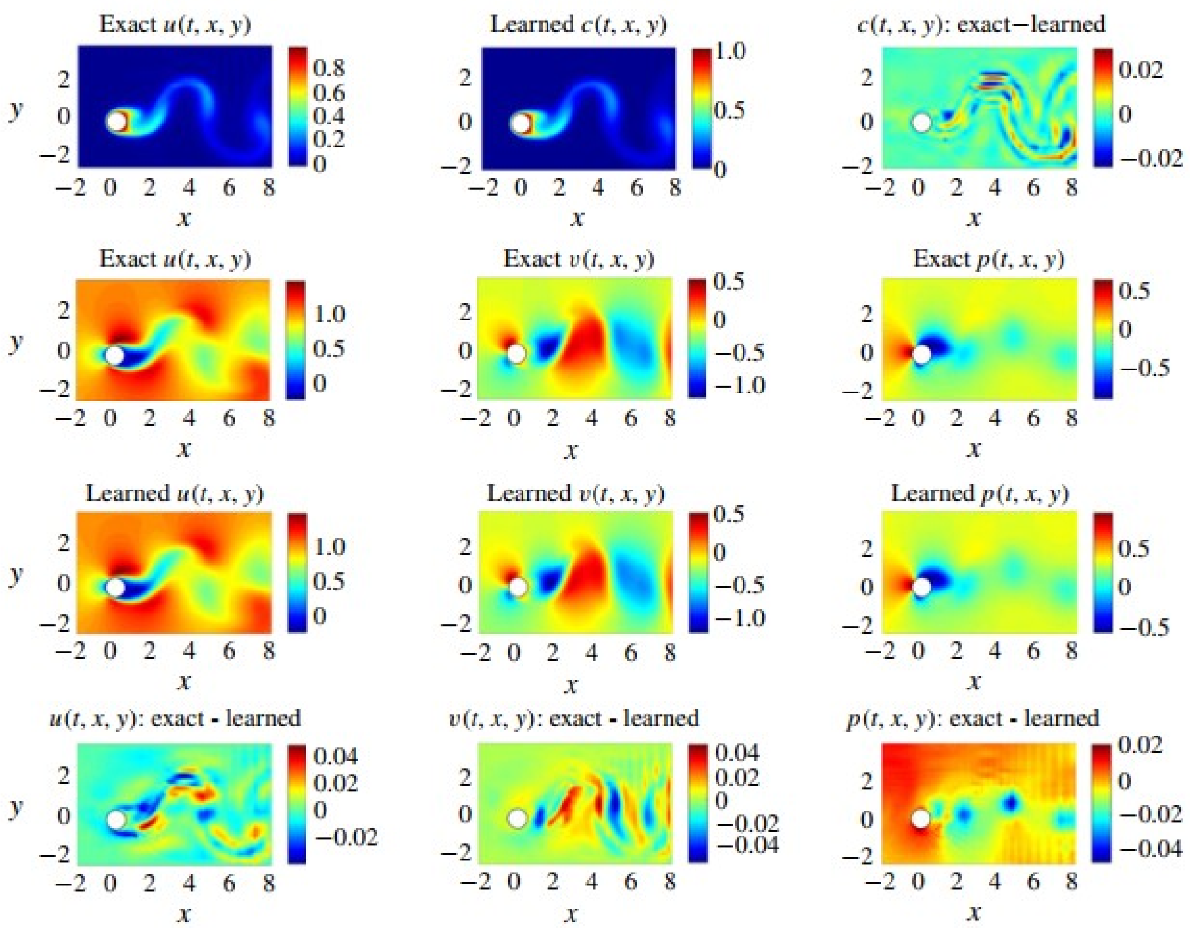


t
x
y
σ
η
c
u
v
p
I
∂_t
∂_x
∂_y
$e_1 = u_t + uu_x + vu_y + p_x - Re^{-1}(u_{xx} + u_{yy})$
$e_2 = v_t + uv_x + vv_y + p_y - Re^{-1}(v_{xx} + v_{yy}) + \eta_{tt}$
$e_3 = u_x + v_y$
$e_4 = c_t + uc_x + vc_y - Pe^{-1}(c_{xx} + c_{yy})$

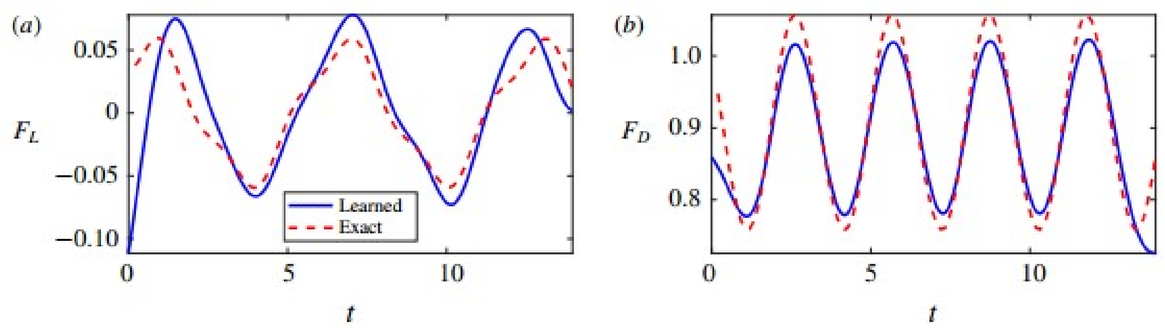
Figure 2.18: PINN framework for simulating vortex induced vibration
Image Source: Raissi et al. [36]

$$\begin{aligned} Loss &= \sum_{n=1}^{N} \left(|c(t^n, x^n, y^n) - c^n|^2 + |\eta(t^n) - \eta^n|^2\right) \\ &+ \sum_{m=1}^{M} \left(|u(t^m, x^m, y^m) - u^m|^2 + |v(t^m, x^m, y^m) - v^m|^2\right) + \sum_{i=1}^{4}\sum_{n=1}^{N} \left(|e_i(t^n, x^n, y^n)|^2\right). \end{aligned}$$

The results obtained after training the PINN have been summarized in figure 2.19. It must be noted that the accuracy of the results is not as high as expected but the main takeaway from this study is that PINNs can also be used for one-way coupled FSI simulations.



(a) Results for velocity contours



(b) Results for force predictions

Figure 2.19: PINN results for vortex induced vibrations
Image Source: Raissi et al. [36]

High Speed Flows
High speed aerodynamic flows can be modelled by using the Euler equations, which lead to discontinuities in the solution in the presence of shocks. A study for solution of such high speed flows using PINNs was performed by Mao et al. [27]. The governing equations for conservation of mass, momentum and energy for high speed compressible flows are as follows:

$$\partial_t U + \nabla . f(U) = 0, \quad x \in \Omega \subset \mathcal{R}^d, \quad d = 1, 2 \quad t = [0, T], \tag{2.25}$$

where for the 1D case,

$$U = \begin{pmatrix} \rho \\ \rho u \\ \rho E \end{pmatrix}, \qquad f(U) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(\rho E + p) \end{pmatrix},$$

and for the 2D case,

$$U = \begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho E \end{pmatrix}, \qquad f = (G_1, G_2), \quad G_i(U) = \begin{pmatrix} \rho u_i \\ \rho u_1 u_i + \delta_{i1} p \\ \rho u_2 u_i + \delta_{i2} p \\ u_i(\rho E + p) \end{pmatrix}, \quad i = 1, 2$$

. Here, ρ represents the density of the fluid, p gives the pressure, u_i gives the velocity component in each dimension, E gives the total energy and δ_{ij} represents the Kronecker delta. To close this system of equations, we require an equation of state relating pressure to total energy. This equation of state is one possibility given as follows:

$$p = (\gamma - 1)\left(\rho E - \frac{1}{2}\rho||\mathbf{u}||^2\right). \tag{2.26}$$

The aim of this study was to learn the density, velocity and pressure fields using a PINN framework by minimizing the loss function incorporating some training data (ICs/BCs and experimental/numerical data if available) and the governing Euler equations presented above.

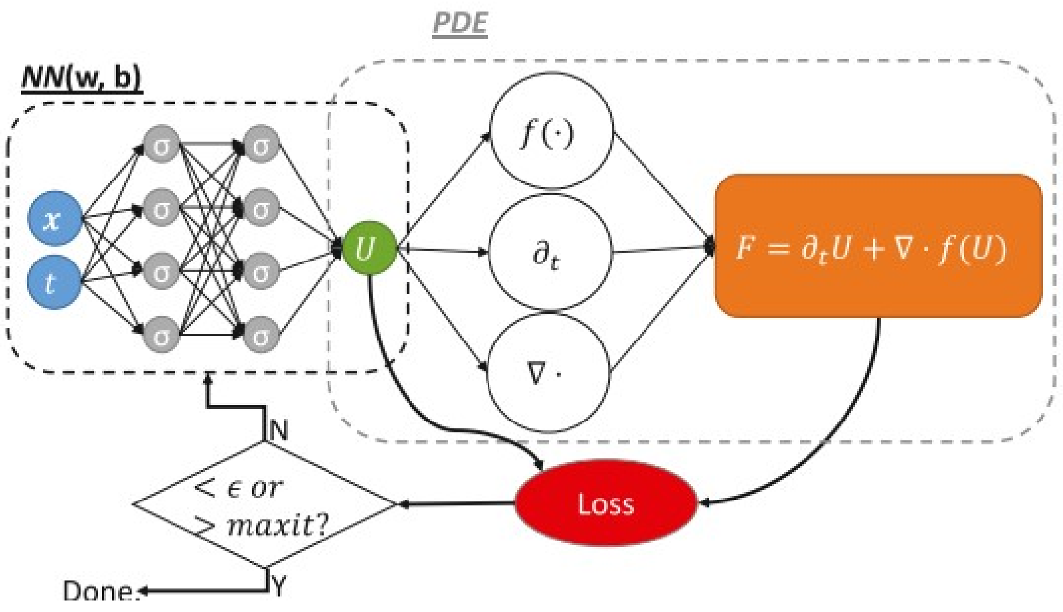
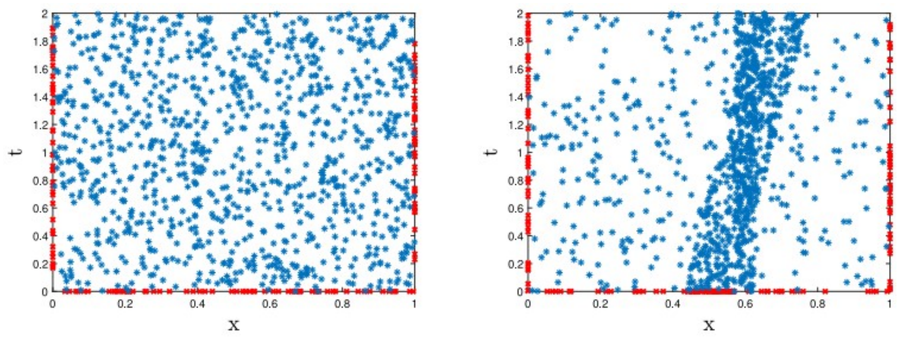
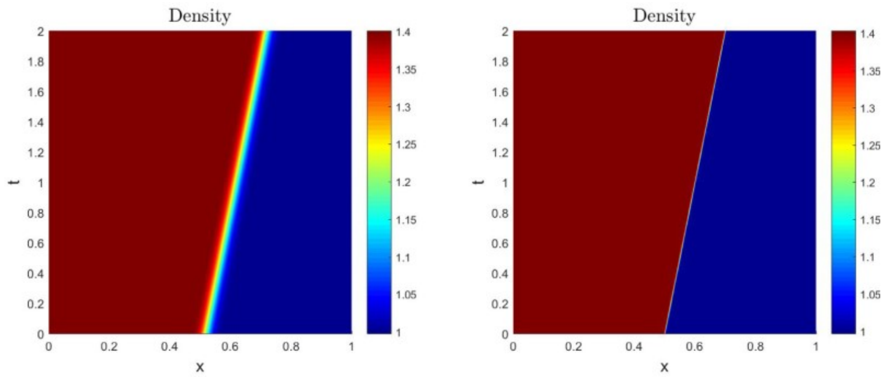


Figure 2.20: PINN framework for euler equations
Image Source: Mao et al. [27]

A simulation for the 1D unsteady Euler equation was performed using the above architecture, where collocation points N_F = 1000, boundary points N_{BC} = 60 and initial points N_{IC} = 60 were used. Two approaches were used for distribution of collocation points: random distribution and clustering near the expected shock as shown in figure 2.21. Two kinds of architectures were tested and compared which were named as HyperPa I: with 7 hidden layers and 20 neurons per layer. The second architecture named HyperPa II: with 4 hidden layers and 40 neurons per layer. tanh activation function was used for both the architectures. Two stage training strategy was adopted by using the Adam optimizer for the initial training and the L-BFGS optimizer for the final training. The computational domain was set to $[0, 1]$, with an initial shock being given at $x = 0.5$, and the left and right states being given by $(\rho_l, u_l, p_l) = (1.4, 0.1, 1.0))$ and $(\rho_r, u_r, p_r) = (1.0, 0.1, 1.0))$. Figure 2.21 depicts the results obtained after evaluating the solutions from the trained PINN.



(a) Collocation points distribution strategies



(b) Results obtained after training the PINN

Figure 2.21: PINN framework for high speed flow results using Euler equations
Image Source: Mao et al. [27]

It can be clearly seen that the clustering of the collocation points in the domain gives better results for the shock prediction, without any diffusive errors as opposed to randomly distributed collocation points. The results for the prediction of the desired variables, have been illustrated in figure 2.22. It can be inferred that indeed the performance of PINNs on the clustered domain was way better compared to the domain with random distribution of the collocation points. It was proposed that there was a need to develop a strategy for adaptive clustering of collocation points during the training process, near the region of occurrence of the shock. Development in this field for high speed flows would be useful as the position and orientation of shock is not always known a priori, such that the collocation points could be clustered beforehand accordingly. Two methodologies which the authors Mao et al. [27] believe could be useful in this regard are residual based adaptive refinement (RAR), where more points are iteratively added in the regions of high residual values, and addition of more points in regions of high gradients of solution.

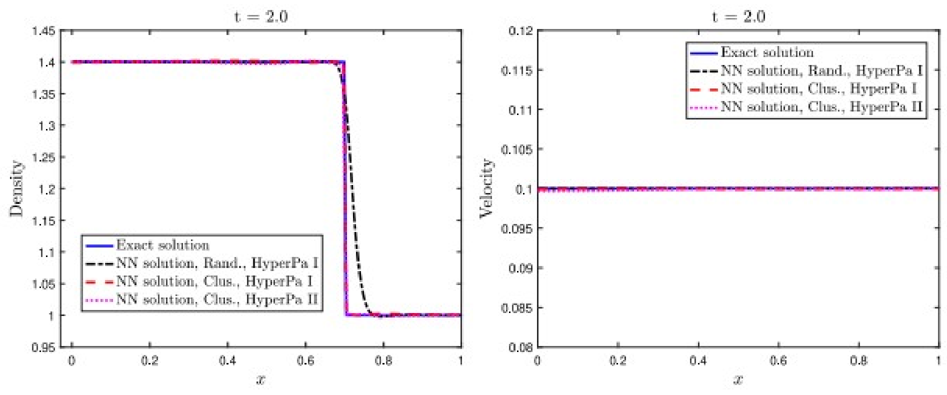


Figure 2.22: Data values obtained from the PINN trained for depicting the Euler equations
Image Source: Mao et al. [27]

2.2.4. Advanced Concepts and Solvers for PINNs

Conservative PINNs

From the previous section, it is evident that the regions of high gradients in the PINN solution need special treatment, such as increasing the density of collocation points in those regions, for better accuracy and easier convergence of the solution. Moreover, it is widely established in literature that deep neural networks (with a large number of hidden layers) are better suited for approximating complex functions with steep or rapidly changing gradients [17]. This gives rise to the need for domain decomposition while performing simulations with PINNs, where we can employ deeper neural networks in regions of steep/rapidly changing gradients and comparatively shallower networks in remaining regions. This approach is better than employing a single deep neural network for the entire domain, as deeper neural networks are in general harder to train due the *vanishing gradients* problem [29]. Therefore, to implement this idea of employing different neural networks to various subdomains of the desired domain, the concept of conservative PINNs (cPINNs) was put forth by Jagtap et al. [17]. This approach simply involves stitching of all the subdomains together by including a conservative flux term inside the loss function:

$$\mathcal{L} = W_{u_p} MSE_{u_p} + W_{F_p} MSE_{F_p} + W_{I_p}(MSE_{\text{flux}} + MSE_{\text{avg}}) \quad p = 1, 2, ..., N_{\text{sd}},$$

where,

$$MSE_{u_p} = \frac{1}{N_{u_p}} \sum_{i=1}^{N_{u_p}} |u^i - u(x^i_{u_p})|^2,$$

$$MSE_{F_p} = \frac{1}{N_{F_p}} \sum_{i=1}^{N_{F_p}} |F_p(x^i_{F_p})|^2,$$

$$MSE_{flux} = \frac{1}{N_{I_p}} \sum_{i=1}^{N_{I_p}} |f_p(u(x^i_{I_p})).\mathbf{n} - f_{p^+}(u(x^i_{I_p})).\mathbf{n}|^2,$$

$$MSE_{avg} = \frac{1}{N_{I_p}} \sum_{i=1}^{N_{I_p}} |u_p(x^i_{I_p}) - \{\{u(x^i_{I_p})\}\}|^2.$$

Here, N_{sd} is the number of subdomains (a sample representation has been given in figure 2.23), F is the residual of governing PDEs, W_{u_p}, W_{f_p} and W_{I_p} are the weights for boundary/initial condition loss, residual loss and interface loss respectively. $f_p.\mathbf{n}$ represents the interface flux for a subdomain p (and all surrounding domains being p^+). u_p, u_F and u_I represent the training points/BCs and ICs, collocation points and interface points. The average velocity is given by:

$$\{\{u\}\} = \frac{u_p + u_p^+}{2}$$

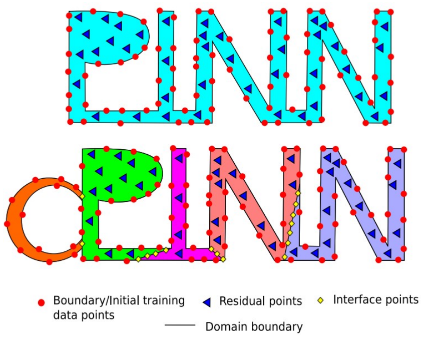


Figure 2.23: Illustration of a typical complex domain suited for cPINN formulation. The complex domain can be represented as a combination of distinct simpler domains represented in different colours and separated by interface points (yellow). A different neural network architecture can be used to train a constituent domain depending on the complexity of the expected solution in each domain
Image Source: Jagtap et al. [17]

NVIDIA SimNets

NVIDIA SimNets is a recent development by the team of NVIDIA which is quoted to be an 'AI accelerated multiphysics simulation framework'. It uses PINNs to solve multiphysics problems, transient problems as well as parameteric PDEs in a fully unsupervised manner, developed by Hennigh et al. [14]. It claims to be the first PINN framework that is capable of solving RANS turbulence equations using a zero equation turbulence model, which are usually hard to converge due to chaotic fluctuations in the flow field [14]. This has been demonstrated by solving a turbulent heat transfer problem coupled with fluid dynamics, for a Reynolds number Re = 1329. One way coupling is ensured by training two separate networks, one for fluid flow, which is trained first and one for heat transfer which is trained subsequently. The SimNet architecture and the results of this simulation are depicted in figure 2.25. The main advantage of NVIDIA SimNets is that it can be readily used as an opensource software (like OpenFoam for CFD) for performing simulations by the mechanism of PINNs in fully unsupervised manner for various kinds of physics and their combinations. The details of operating NVIDIA SimNets can be found in [37].

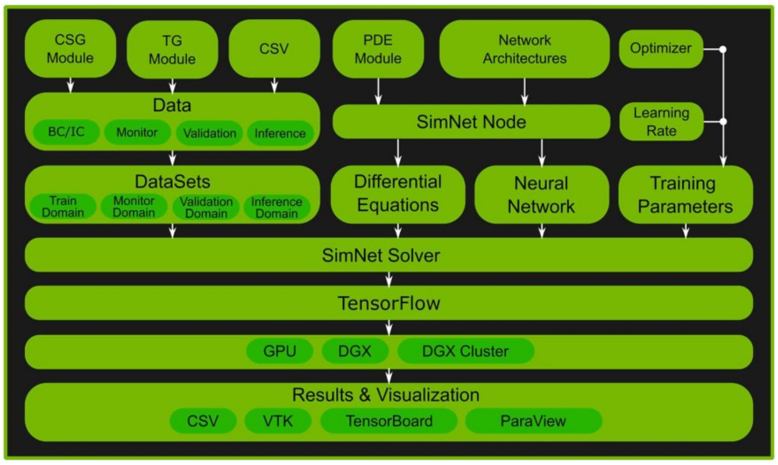


Figure 2.24: SimNets structure
Image Source: Hennigh et al. [14]

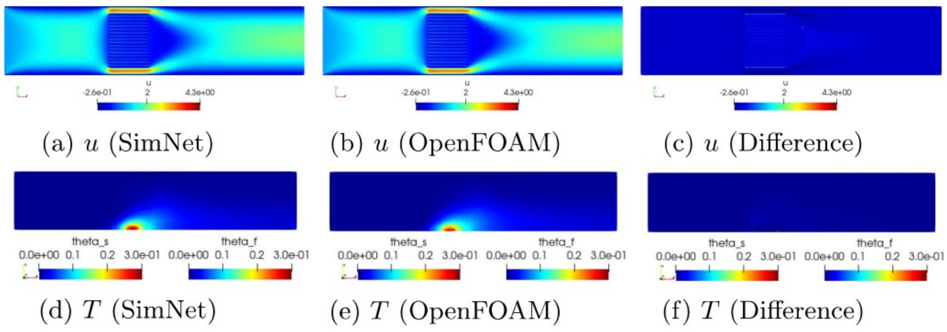


Figure 2.25: SimNet based solution for a turbulent multiphysics channel flow using Navier Stokes equations for a coupled fluid-dynamic and heat transfer problem. The velocity(u) contours and the Temperature (T) contours have been illustrated. A comparison of solution contours predicted by SimNets has been shown with respect to the results from numerical simulation performed using OpenFoam.
Image Source: Hennigh et al. [14]

Bayesian PINNs and Uncertainty Quantification

Uncertainty quantification is an important parameter to quantify whether the predictions made by a surrogate model (neural network mapping inputs to outputs) are trustworthy or not. Any trained neural network model has two fundamental types of uncertainties: *Aleatoric uncertainty* and *epistemic uncertainty*. Aleatoric uncertainy is due to noise in the training data, and cannot be eliminated by introducing more training data. This type of uncertainty is easy to estimate by analysing the data itself, but its elimination/minimization is a tedious process as it might require improving the accuracies of sensors and other hardware used for gathering the data. Epistemic uncertainty is the uncertainty of the trained neural network model itself. It is hard to predict but it can be minimized by introducing more and more training data. Liu et al. [41] have developed B-PINNs, standing for Bayesian physics-informed neural networks, which are capable of estimating the uncertainty of a surrogate model. The basic high level concept of B-PINNs is to replace all the deterministic individual weight values by some probability distributions, which are initialised as 'priors' $P(\theta)$. After introducing the training data in the form of 'evidence' for modifying the prior, we obtain the likelihood $P(D|\theta)$. After training, the probability distribution of the weights obtained is termed as 'posterior' distribution. Once trained, repeated evaluations are performed (sampling) by passing the input space through the neural network (Monte Carlo dropouts are also used [41]) and the mean and variance of the scattered output data is calculated. A detailed analysis of Bayesian PINNs is out of scope of this work but can be found in [41].

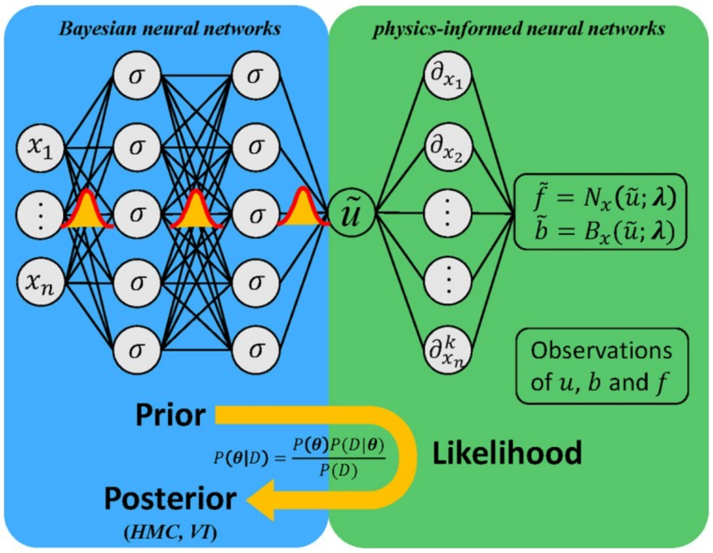

Bayesian neural networks
physics-informed neural networks
x_1
x_n
σ
$\tilde{u}$
∂_{x_1}
∂_{x_2}
$\partial^k_{x_n}$
$\tilde{f} = N_x(\tilde{u}; \boldsymbol{\lambda})$
$\tilde{b} = B_x(\tilde{u}; \boldsymbol{\lambda})$
Observations of u, b and f
Prior
$P(\boldsymbol{\theta}|D) = \frac{P(\boldsymbol{\theta})P(D|\boldsymbol{\theta})}{P(D)}$
Likelihood
Posterior
(HMC, VI)

Figure 2.26: Framework of bayesian PINNs
Image Source: Liu et al. [41]

Further, it is important to highlight various aspects which contribute to epistemic uncertainty, which can be classified to three broad categories of errors, namely the approximation error, generalization error and optimization error [25]. Approximation error highlights the difference between the actual output for a function u and the closest approximate output u_F which can be provided by the neural network formulation of that function, given the limited dimensions of the neural network. Here F denotes the family of all functions that can be represented by the neural network. Generalization error represents the capacity of this family F. Finally, the optimization error highlights the error between the true minima for the loss function compared to the minima achieved during the training process.

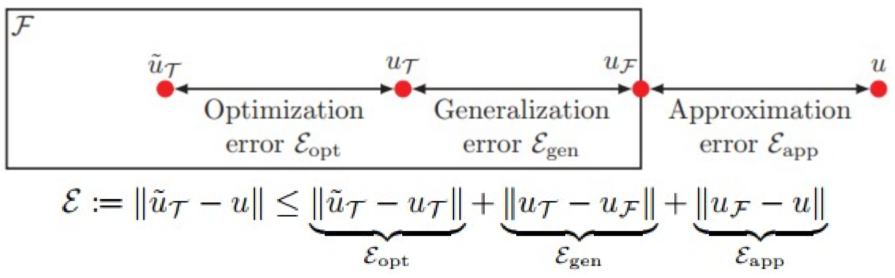

$\mathcal{F}$
$\tilde{u}_{\mathcal{T}}$
$u_{\mathcal{T}}$
$u_{\mathcal{F}}$
u
Optimization error $\mathcal{E}_{\text{opt}}$
Generalization error $\mathcal{E}_{\text{gen}}$
Approximation error $\mathcal{E}_{\text{app}}$
$\mathcal{E} := \|\tilde{u}_{\mathcal{T}} - u\| \le \underbrace{\|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\|}_{\mathcal{E}_{\text{opt}}} + \underbrace{\|u_{\mathcal{T}} - u_{\mathcal{F}}\|}_{\mathcal{E}_{\text{gen}}} + \underbrace{\|u_{\mathcal{F}} - u\|}_{\mathcal{E}_{\text{app}}}$

Figure 2.27: Errors contributing to epistemic uncertainty in PINNs
Image Source: Lu et al. [25]

2.2.5. Convergence of PINNs

Some important considerations which influence the convergence of PINNs are summarized as follows:

1. **Frequency Principle:** This principle states that neural networks learn solutions starting from lower frequencies to higher frequencies. Thereby, a neural network needs to be trained longer to obtain high frequency features in the solution. This phenomenon has been studied by Zhang et al. [45] and also pointed out by Markidis [29]. Lu et.al [25] concluded that existence of higher order derivatives in the governing equation might refute this principle and even higher frequencies are learnt almost simultaneously with the lower frequencies.
2. **Adaptive Activation Functions:** This concept was proposed by Jagtap et al. [16], where the convergence of the training process of a PINN can be accelerated by introducing two new hyper-parameters, a constant n with a fixed value greater than 1 (typically less than 5), and another trainable parameter a, which is initialised as $1/n$ and it's value is updated at every iteration during the training process by the optimizer, along with the weights of the neural network. The modified activation function switches from tanh(x) to tanh(nax). This enhances the representation power of the neural network as the activation functions are allowed the freedom to adjust their slope for more accurate prediction of the output. This idea was further extended by Jagtap et al. [15] to 'locally' adaptive functions, where each neuron is allowed to have a separate value of n and a. This further enhances the representation power and degree of freedom of the neural network but also increases the computational requirements manifold.
3. **Gradient Enhanced PINN:** This concept was proposed by Yu et al. [43], who conducted a study and concluded that the convergence rate for training a PINN could be increased by minimizing the gradient of the PDE residual along with the PDE residual itself in the loss function. For example, for training a Poisson equation, the loss function would contain an addition of mean squared errors of the PDE residual $\Delta u - f$ and the gradient of this PDE residual, that is $\nabla^3 u - \nabla f$.
4. **Dimensions of PINNs:** The number of hidden layers and neurons per layer are important hyperparameters which determine the successful convergence as well as accuracy of the PINN solutions. In general, from the author's understanding and experience, an optimal PINN architecture comprises a very large number of neurons per hidden layer (width) compared to the number of hidden layers used (depth). This can be attributed to two main observations that can be made from the available literature: deeper neural networks are harder to converge subject to the vanishing gradients problem [29] and deep neural networks with infinitely many neurons per layer converge to Gaussian processes (Wang et al. [40]), which helps in evaluating the convergence characteristics of PINNs. Moreover, an empirical relation of 32 neurons per output variable was proposed by Raissi et al. [36] whereas the number of hidden layers were kept limited to 10. Usually, the width of a PINN network is of the order of 100s of neurons, whereas the number of hidden layers are limited to less than 10.

3 Operator Learning

In the previous chapter, we have focused on the analysis of PINNs, which are capable of representing one particular instance of a PDE or a system of PDEs by learning the mappings between finite dimensional Euclidean spaces ($\mathbb{R}^{\text{input dim}}$) to ($\mathbb{R}^{\text{output dim}}$). Thereby, such frameworks are limited to one particular value of source function and boundary/initial conditions. On the contrary, if we could learn an operator itself, which can provide mapping between infinite dimensional function spaces, our surrogate model would not be limited to only one particular instance of PDE, it could rather represent a family of PDEs. For example, if we consider an example of Poisson equation, by using a PINN framework, we can learn an equation $\Delta u = f$, for one particular mapping between a given source function f and its solution u. On the other hand if we learn the Laplacian operator Δ itself, we can map any given function f to its corresponding solution output u by passing it through the Laplacian operator, thereby having the capability to represent a large family of PDEs. This concept has given rise to research in the field of *neural operators*. There are two main schools of thought for using neural operators: DeepONets given by Lu et al. [26] and Fourier Neural Operators given by Kovachki et al. [20] and further extended by Li et al. [21].

3.1. DeepONets

The core idea of DeepONet framework for operator learning was derived from the work of Chen and Chen [6], which proved that neural networks were capable for representing operators, which can be inferred from the following:

Theorem 1 - Universal Approximation Theorem for Operator: Suppose that X is a Banach space and σ is a continuous non-polynomial Banach space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively. V is a compact set in $C(K_1)$, G is a nonlinear continuous operator, which maps V into $C(K_2)$. Here $C(K)$ represents the Banach space of all continuous functions defined on K with norm $\|f\|_{C(K)} = \max_{x\in K}|f(x)|$. Then for any $\epsilon > 0$,there are positive integers n, p and m, constants c_i^k, ξ_{ij}^k, θ_i^k, $\zeta_k \in R$, $w_k \in R^d$, $x_j \in K_1$, $i = 1, ..., n$, $k = 1, ..., p$ and $j = 1, ..., m$, such that:

$$\left| G(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{\text{branch}} \underbrace{\sigma(w_k.\dot{y} + \zeta_k)}_{\text{trunk}} \right| < \epsilon,$$

holds for all $u \in V$ and $y \in K$. Here, $C(k)$ is the Banach space of all continuous functions defined on K with norm $||f||_{C(K)} = max_{x\in K}|f(x)|$.
The proof of theorem 1 can be found in [6]. An extension to this theorem (sharing the same notations) was given as theorem 2 by Lu et al. [24], and its proof can also be found in [24].

Theorem 2 - Generalized Universal Approximation Theorem for Operator: Suppose that X is a Banach space , $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively, V is a compact set in $C(K_1)$. Assume that $G : V \longrightarrow C(K_2)$ is a nonlinear continuous operator. Then for any $\epsilon > 0$, there exist positive integers m, p, continuous vector functions $\boldsymbol{g} : \mathbb{R}^m \longrightarrow \mathbb{R}^p$, $\boldsymbol{f} : \mathbb{R}^d \longrightarrow \mathbb{R}^p$ and $x_1, x_2, ..., x_m \in K_1$, such that

$$\left| G(u)(y) - \langle \underbrace{\mathbf{g}(u(x_1), u(x_2), ..., u(x_m))}_{\text{branch}}, \underbrace{f(y)}_{\text{trunk}} \rangle \right| < \epsilon,$$

holds for all $u \in V$ and $y \in K_2$, where $\langle \cdot, \cdot \rangle$ denotes the standard inner product in $\mathbb{R}^p$. Furthermore, the functions **g** and **f** can be chosen as diverse classes of neural networks, which satisfy the classical universal approximation theorem of functions, for example, (stacked/unstacked) fully connected neural networks, residual neural networks and convolutional neural networks.

On the basis of the Theorem 1 and Theorem 2 shown above, the neural network structure of DeepONets was framed as a juxtaposition of two neural networks, namely branch net, which acts as a functional mapping between function spaces as $G(u)$ (G is the operator and u is the input function), and a trunk net, which evaluates the real value for the function output of the branch net at a given input point y. The resulting architecture that was proposed for DeepONets has been illustrated in figure 3.1. The stacked DeepONet architecture was inspired by Theorem 1 having one trunk network and p branch networks, whereas Theorem 2 gave rise to the unstacked DeepONet architecture having one trunk network and correspondingly one branch network. DeepONets are trained using supervised learning, by applying mean squared error between the DeepONet output and the training data ([$u(x_1), u(x_2)..u(x_m)$],y) in the loss function.

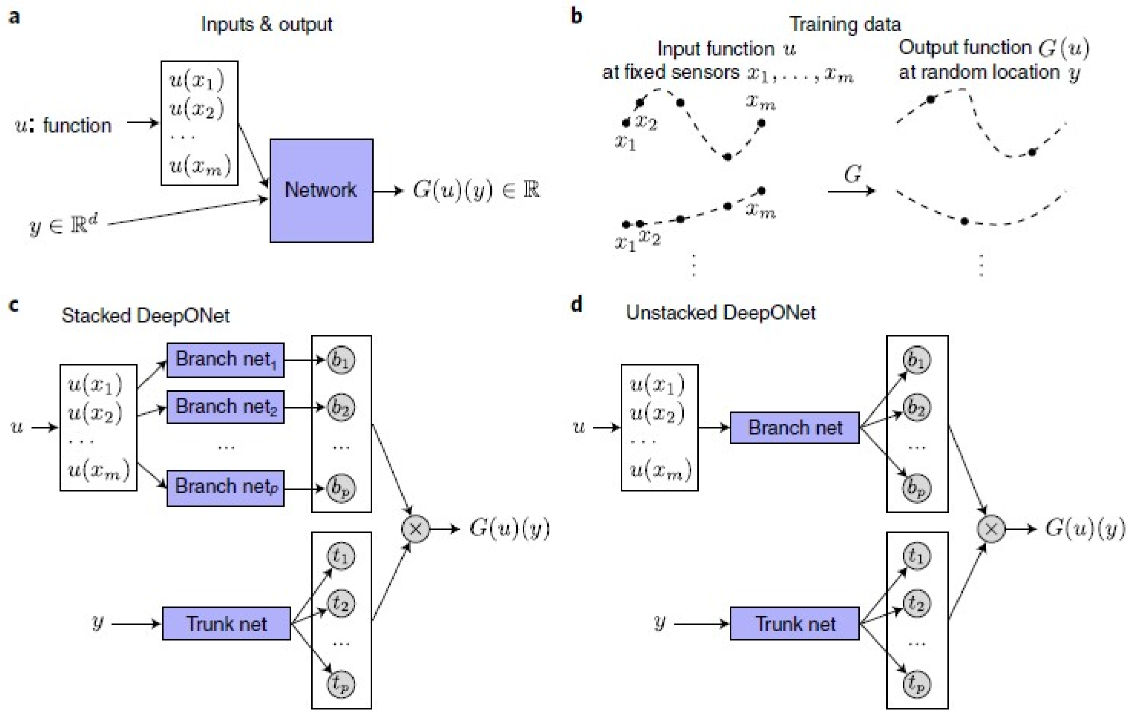


Figure 3.1: DeepONet framework
Image Source: Lu et al. [26]

DeepM&MNet: A DeepONet application
The concept of DeepM&MNets was given by Mao et al. [28], where M&M stands for multiphysics and multiscale problems. In this approach, multiple DeepONets are trained for different types and scales of physics, and combined together to generate the overall deep learning architecture. The problem taken up by Mao et al. to demonstrate this architecture pertained to hypersonic flows, where various kinds of dissociative reactions take place in the flow at the atomic level, which in turn influence the composition of the gas and subsequently its properties like viscosity, specific heat capacity (Cp) , ratio

of heat capacities (γ) etc. The change in these fluid properties also influence the flow properties at macroscopic level, particularly the velocity U and temperature T.

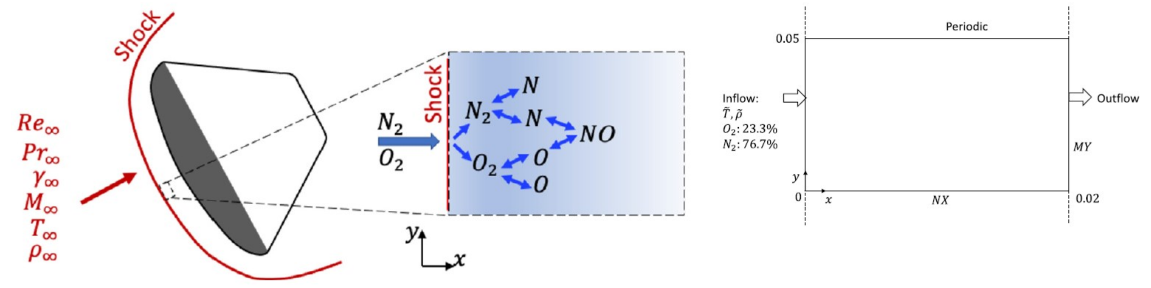
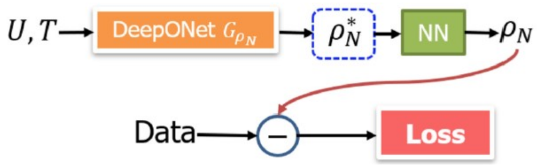
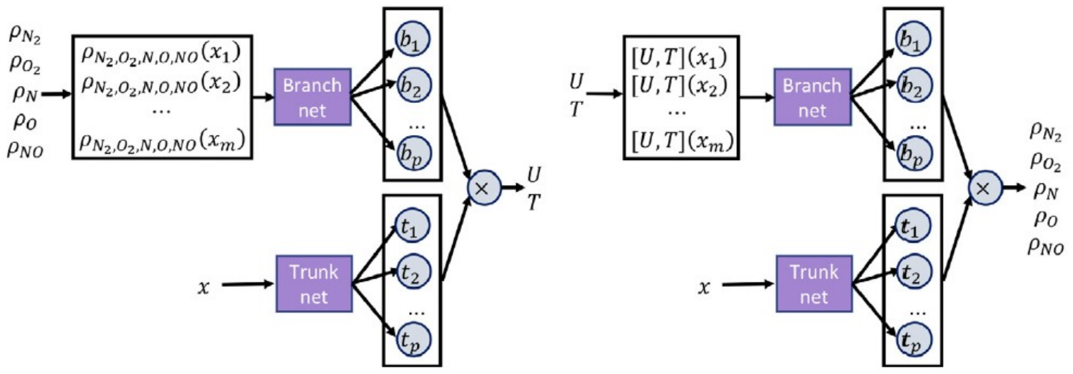


Figure 3.2: Hypersonics multiphysics coupling
Image Source: Mao et al. [28]

To study the coupled dynamics between fluid flow and chemical reactions, two kinds of DeepONets were trained independently: $G_{U,T} : (U,T) \longrightarrow \rho_i \;\; i = N, NO, O, N_2, O_2$ for evaluating fluid gas properties from flow parameters and $G_{\rho_i} : \rho_i \longrightarrow (U,T) \;\; i = N, NO, O, N_2, O_2$ for the opposite relation/mapping. These individual DeepONets can be used as building blocks to construct the desired coupled physics for hypersonic flows.



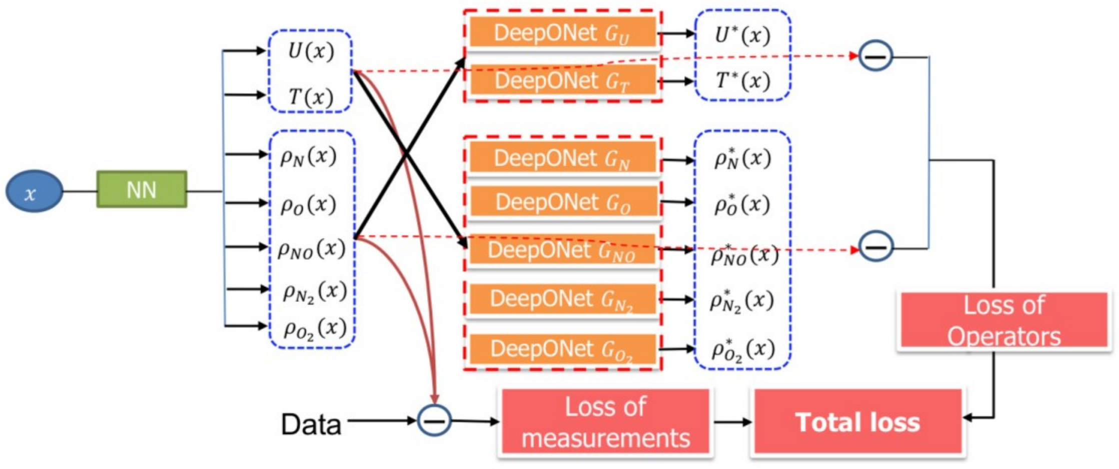
(a) Structure of individual DeepM&MNet network



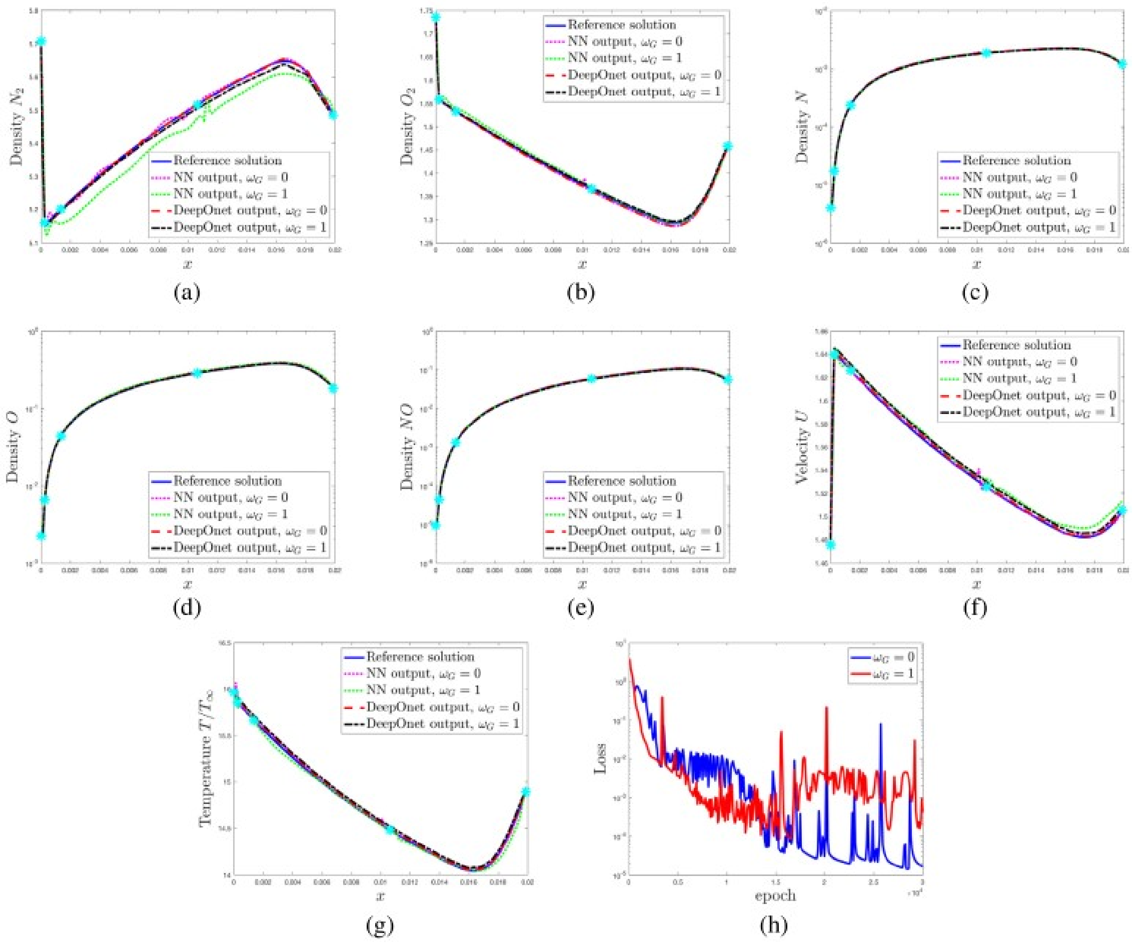
(b) Types of DeepM&MNets used

Figure 3.3: DeepM&MNet architectures trained for their respective uncoupled physics for flow properties as well as gas reactions for a hypersonic flow
Image Source: Mao et al. [28]

The constituent DeepONets were trained via supervised learning using mean squared error loss metric. Once trained, the DeepONets were combined to represent the complete coupled physics. The architecture used for combining DeepONets depends on the availability of training data. If data is available for densities as well as flow characteristics namely velocity and temperature (U,T), one may use a parallel combination of DeepONets. A more realistic hypothesis would be that only the flow data is available for training purpose (U,T) as these observations can be made easily at macroscopic level. This formulation would require the use of a series of combinations of DeepONets, where the densities predicted from $G_{U,T}$ are fed as inputs directly to G_{ρ_i}. In addition to the respective mean square losses, an additional term can be added to the loss which ensures continuity that is, $\rho U(x) =$ constant. This term may be turned on/off using it's weight coefficient w_G. The results for both the formulations have been illustrated in figures 3.4 and 3.5. It can be seen that the results generated by both formulations agree reasonably well with the real physics. The inherent disadvantage of using DeepONets in this case is the need for training data for several variables, which might be very expensive to generate (computationally for simulations as well financially for experiments) for hypersonic flows. A formulation to overcome this limitation is discussed in the next section.

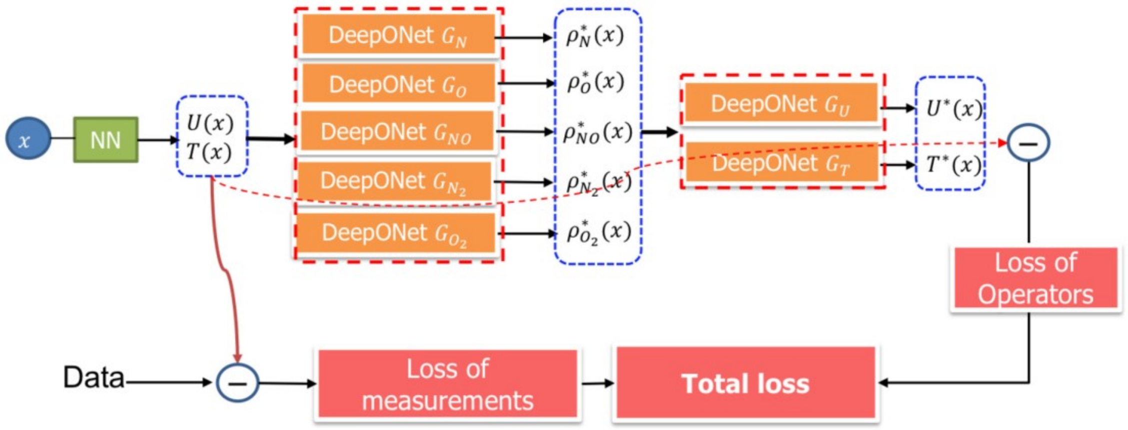


(a) Structure of parallel network for coupling of various physical phenomena (fluid flow and gas reactions) to represent hypersonic flows

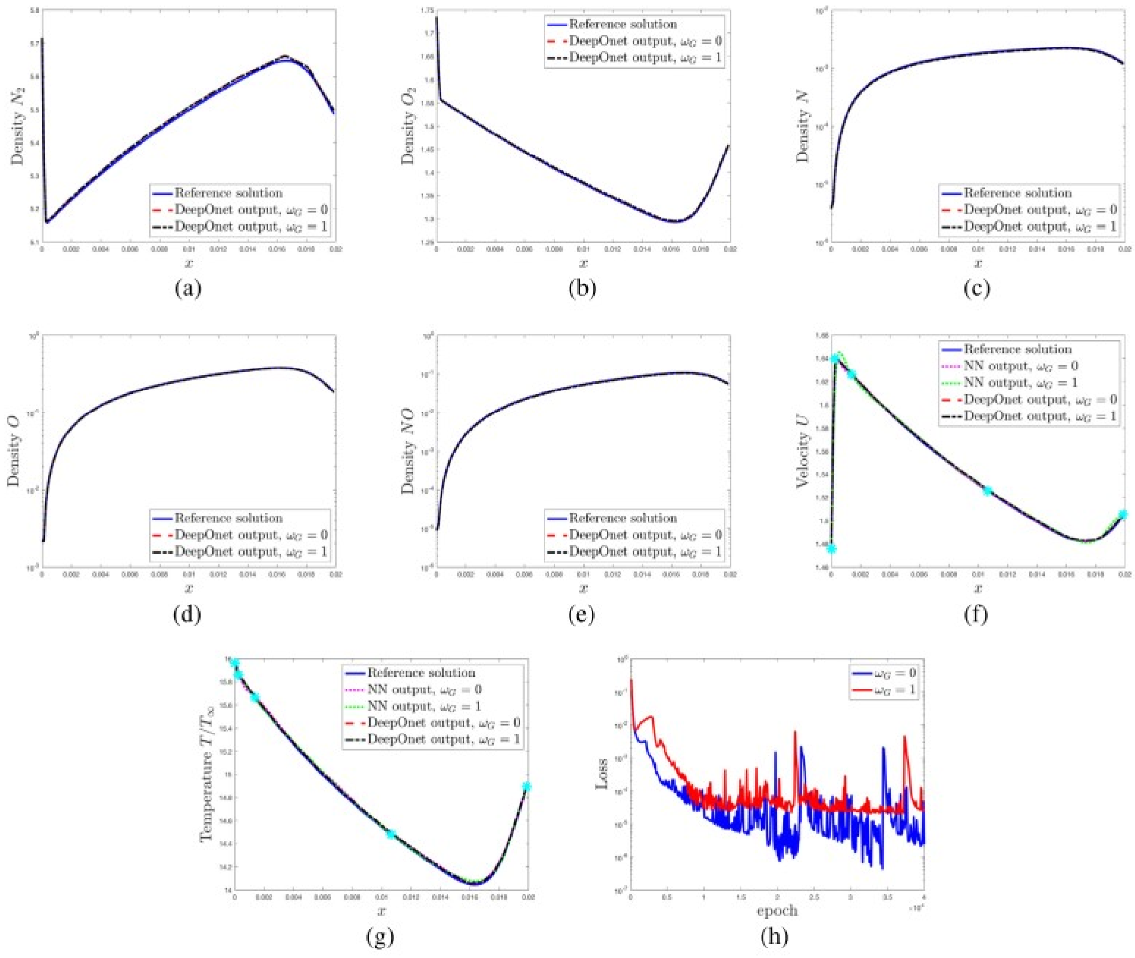


(b) Results from parallel network for coupled reactive flow physics for hypersonic flows

Figure 3.4: Parallel DeepM&Mnet architecture and results obtained for the target variables for a coupled reactive hypersonic flow
Image Source: Mao et al. [28]



(a) Structure of series network for coupling of various physical phenomena (fluid flow and gas reactions) to represent hypersonic flows



(b) Results from series network for coupled reactive flow physics for hypersonic flows

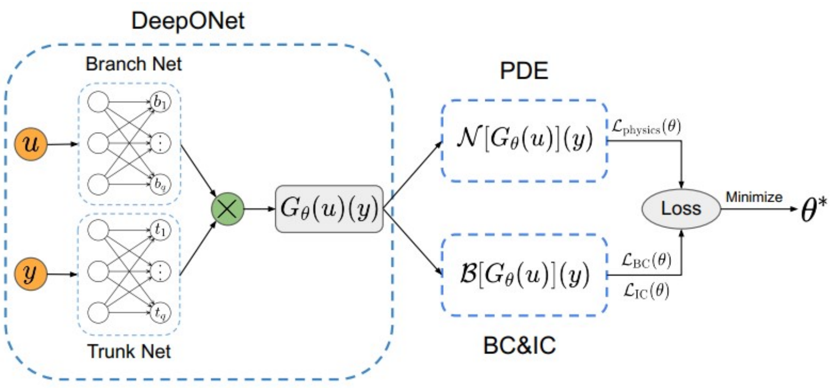
Figure 3.5: Series DeepM&Mnet architecture and results obtained for the target variables for a coupled reactive hypersonic flow
Image Source: Mao et al. [28]

3.2. Physics Informed DeepONets

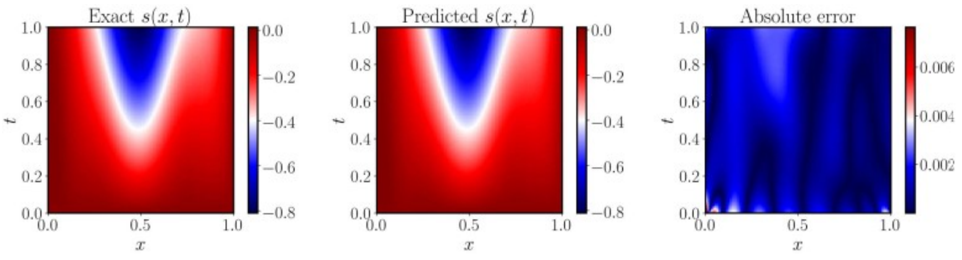
The concept of physics-informed DeepONets was introduced by Wang et al. [39] to overcome the limitation of DeepONets being solely dependent on supervised training. The fundamental principle which has driven this idea is the concept of automatic differentiation. The output of a DeepONet $G(u)(y)$ is differentiable with respect to it's inputs, therefore a PDE residual can be generated for the loss function for training the DeepONet. This makes DeepONets essentially similar to PINNs, only the neural network architecture between the inputs and the outputs being different. The loss function is similar to that of PINNs and comprises of boundary/initial condition loss with the given corresponding data labels, and the residual PDE loss. The use of this approach was experimented with the reaction diffusion equation:

$$\frac{\partial s}{\partial t} = \mathcal{D}\frac{\partial^2 s}{\partial^2 x} + ks^2 + u(X) \qquad (x,t) \in (0,1] \times (0,1], \tag{3.1}$$

where $\mathcal{D} = 0.01$ is the diffusivity,$k = 0.01$ is the reaction rate, $u(x)$ is the source function and $s(x)$ is the solution. The physics informed DeepONet architecture along with the results obtained for the above stated equation 3.1 are shown in figure 3.6. This methodology seems very promising as it seems to combine the best of both worlds pertaining to data independence of PINNs and better generalization ability of DeepONets' operator approach.


DeepONet
Branch Net
Trunk Net
u
y
b_1
b_q
t_1
t_q
$G_\theta(u)(y)$
PDE
$\mathcal{N}[G_\theta(u)](y)$
$\mathcal{L}_{\text{physics}}(\theta)$
Loss
Minimize
θ^*
$\mathcal{B}[G_\theta(u)](y)$
$\mathcal{L}_{\text{BC}}(\theta)$
$\mathcal{L}_{\text{IC}}(\theta)$
BC&IC

(a) Structure of physics informed DeepONet



(b) Results for reaction diffusion equation

Figure 3.6: Physics informed DeepONets
Image Source: Wang et al. [39]

3.3. Fourier Neural Operators

A second school of thought that put forth the idea of using neural operators for mapping function spaces in scientific simulations was given by Kovachki et al. [20]. The fundamental principle of this approach relied on establishing an analogy with classical neural networks which are used to map functions between Euclidean spaces, using linear algebraic equations inside nonlinear activation functions for a series of hidden layers. Such neural networks are capable of representing any non-linear functions as established by the Theorem 1 by Chen and Chen [6].

$$\hat{y} = \sigma(w_{i+1}(\sigma(w_i z_{i-1} + b_i) + b_{i+1})...$$

Similarly, it is possible to represent nonlinear operators (analogous to nonlinear functions) by stacking linear operators (analogous to linear algebraic equations) inside nonlinear activation functions σ. Some theoretical concepts about linear operators are presented in brief as follows. We assume $\mathcal{L}$ is a linear operator, for example Laplacian Δ, that maps a solution $u(x)$ to an output function/source term $f(x)$:

$$\mathcal{L}(u) = f,$$

then, to recover the solution u from the source function f, we can apply a reverse linear operator $\mathcal{L}^{-1}$ (if it exists at all, that is when there are no non-trivial solutions for the equation $\mathcal{L}(u) = 0$) on the source function f. This is again analogous to recovery of solution x from a system of equations $Ax = b$ by equating $x = A^{-1}b$ (or using pseudo inverse A^{+}). In linear operators, the process of solving for $\mathcal{L}^{-1}f$ can be substituted by performing a convolutional integral on the source function f with an appropriate Green's function G over the entire domain $\mathcal{D}$, provided the solution has homogeneous boundary conditions:

$$u = \mathcal{N}^{-1}f = \int_{\mathcal{D}} G(\cdot, y)f(y)\,dx, \tag{3.2}$$

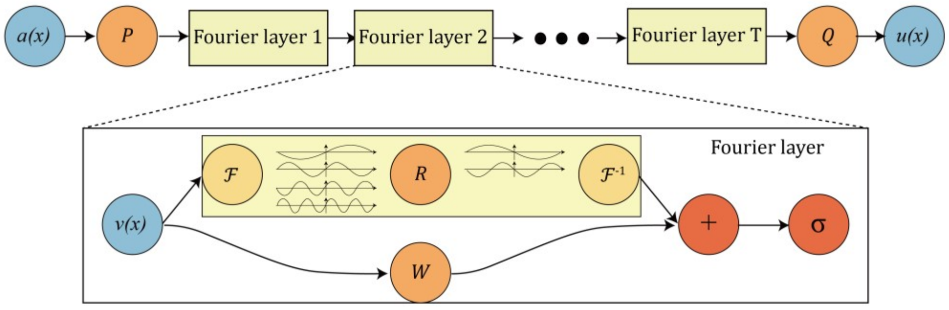
for example, for Poisson equation $-\Delta u = f$, with homogeneous boundary conditions in domain $x \in (0, 1)$, the corresponding Green's function is given as:

$$G = \frac{1}{2}(x + y - |y - x|) - xy \qquad (x, y) \in [0, 1]. \tag{3.3}$$

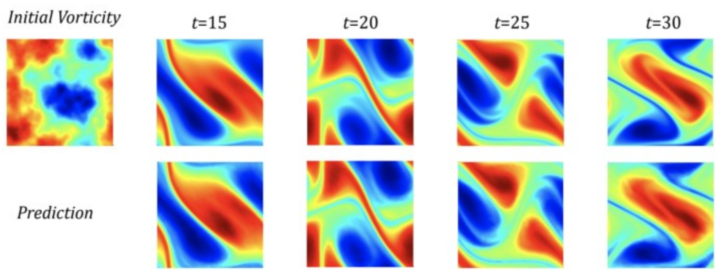
This Green's function can be substituted by a kernel operator in the neural network framework having weights θ according to Kovachki et al. [20]:

$$\mathcal{K}(f) = \int_{\mathcal{D}} \kappa_\theta(\cdot, y)f(y)\,dy.$$

Now, we also know that a convolution operation in a given space is equivalent to a simple multiplication operation inside the corresponding transformed Fourier space. Thereby, according to Kovachki et al. [20], the kernel κ_θ could be replaced by a simple matrix R_ϕ which can be multiplied to the Fourier transformed input (using fast Fourier transform FFT for discrete inputs) and subsequently one can apply inverse Fourier transform (iFFT) to get back to the original solution space. This gave rise to Fourier neural operators [21]. The structure of a single hidden Fourier layer is given in figure 3.7. An additional matrix W is added to map the inputs to outputs to increase the representation power of the Fourier layers such as for non-periodic solutions. Also the simulation results obtained by using Fourier neural operators have been presented for the vorticity variant of transient Navier Stokes. Also, to test whether the hypothesis of assuming that the neural network kernel κ_θ actually assumes the role of Green's function, the trained kernel for poisson equation has been compared to the Green's function (3.3) in figure 3.8.



(a) Structure of fourier neural network



(b) Results from fourier neural network for vorticity equation

Figure 3.7: Fourier Neural Operator Networks
Image Source: Li et al. [21]

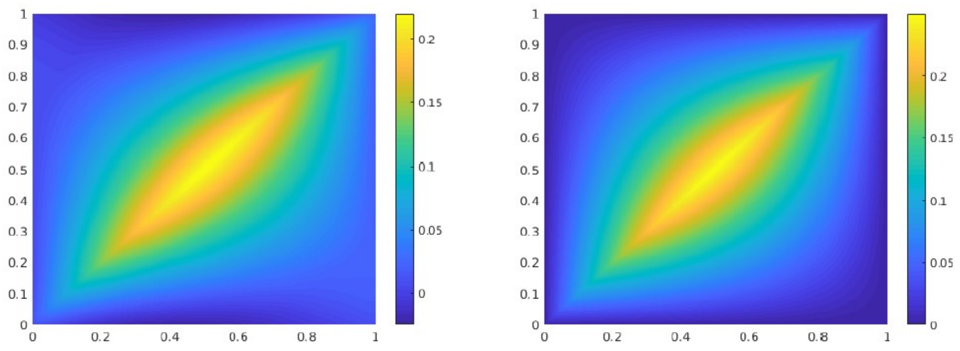


Figure 3.8: Validation of green's function hypothesis: **Left Image** represents the learned kernel function while the **Right Image** illustrates the analytical green's function.
Image Source: Kovachki et al. [20]

3.4. Physics Informed Neural Operator (PINO)

Fourier Neural Operators like conventional DeepONets also have the limitation of being dependent on supervised learning. This limitation was overcome by the work of Li et al. [22] by developing physics informed neural operators (PINOs). In this framework, a two stage training process was prescribed for embedding the knowledge of physics into the PINO framework. First, some training data or PDE loss (residual defined using automatic differentiation) was used to train the neural operator for representing a family of PDEs. Next, this trained neural operator was used as ansatz to further train for a particular PDE instance of interest. The second step enhanced the accuracy of the prediction but could be avoided as well. This formulation was tested on chaotic Kolmogorov flow (Re = 500) and lid driven cavity flow, the results for which have been summarized in the following figure 3.9

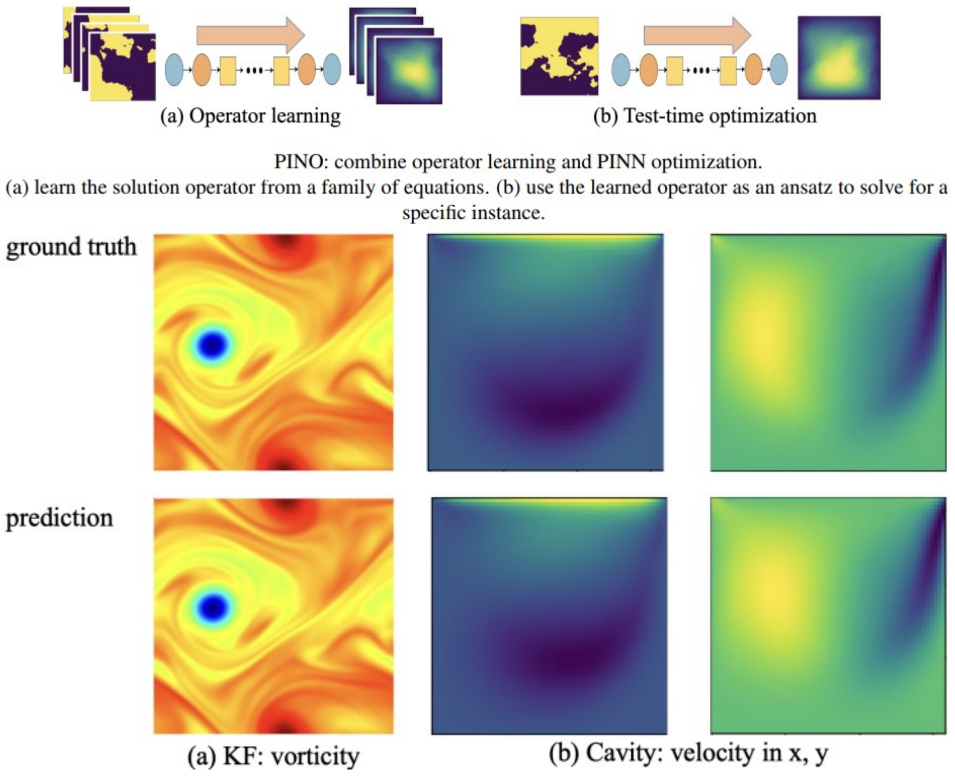

(a) Operator learning
(b) Test-time optimization
PINO: combine operator learning and PINN optimization.
(a) learn the solution operator from a family of equations. (b) use the learned operator as an ansatz to solve for a specific instance.
ground truth
prediction
(a) KF: vorticity
(b) Cavity: velocity in x, y

Figure 3.9: Physics Informed Neural Operators
Image Source: Li et al. [22]

4 Generative Modelling for Geometry

This domain of deep learning is used to generate low dimensional representations of various geometries into latent space vectors, thereby enabling the generation of a wide range of geometries solely by manipulating the latent space variables. Generative modelling can broadly be classified into two techniques: auto-encoders and generative adversarial networks.

4.1. Auto-encoders

Auto-encoders (AE): The vanilla formulation of auto-encoders essentially consists of two component networks namely an encoder and a decoder. An encoder maps an input of given dimensions onto a latent space of significantly less dimensions. The decoder on the other hand maps the latent space vector onto an output of same dimensions as the input, such that it mimicks the input. The training process can be carried using the conventional MSE (mean squared error) loss between input and output vector. After training the auto-encoder framework, the encoder is capable of generating the original input solely from the latent space, thereby making the low dimensional latent space fully capable of representing a significantly higher dimensional entity.

Variational Auto-encoders (VAE): The classic auto-encoder is not robust enough to be able to work on representations of the latent space which have not been seen during the training process. This limitation is overcome by introducing variational auto-encoders, where the deterministic weights of the latent layer are replaced by a probability distribution, represented by a mean μ and variance σ. This makes the auto-encoder more robust and capable of generating similar images based on similar latent space settings which have not been seen during the training process. For an input x, latent vector z and output $\hat{x}$, the encoder now represents a probability distribution $q_\phi(z|x)$ and the decoder represents the probability distribution $p_\theta(x|z)$ [1]. The loss function is now given as:

$$\text{Loss}(\theta, \phi, x) = \text{reconstruction loss} + \text{regularization loss},$$

where, reconstruction loss is the same as that used in classic auto-encoder (mean squared error) and regularization loss tries to match the probability distributions, the prior distribution $p(z)$ and encoder distribution $q_\phi(z|x)$. This can be achieved by minimizing the Kullback-Leibler (KL) divergence [1] between these probability distributions:

$$D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z)) = \sum_z q_\phi(z|x) log \left(\frac{q_\phi(z|x)}{p_\theta(z)} \right). \tag{4.1}$$

The KL divergence basically quantifies the difference in two probability distributions. For example, KL divergence between two Gaussian distributions of slightly different means and variances would be close to zero, whereas it would be a very large value when computed over a Gaussian and a uniform probability distribution.

Conditional Variational Auto-encoders (CVAE): These auto-encoders have additional nodes at encoder input as well as decoder input so that the images generated by the CVAEs conform to this additional information given in the form of labels. The use of auto-encoders for parametrization of airfoil geometries was studied by Yonekura et al. [42]. Yonekura et al. [42] demonstrated the use of CVAEs for the generation of auto-encoders where lift coefficients C_L were given as conditional labels

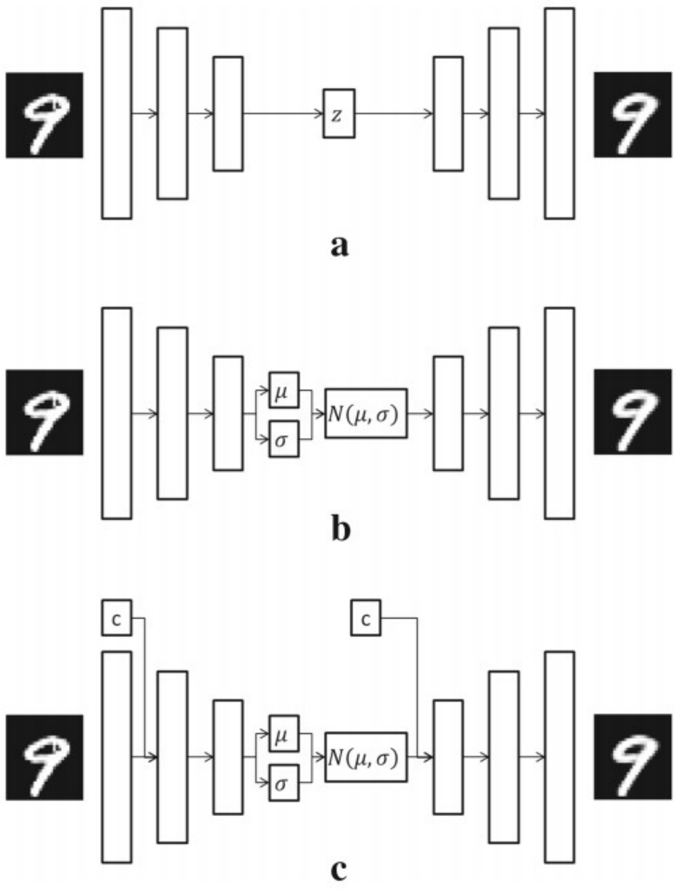
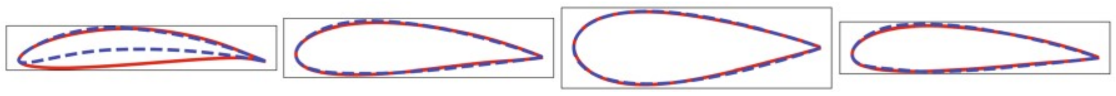
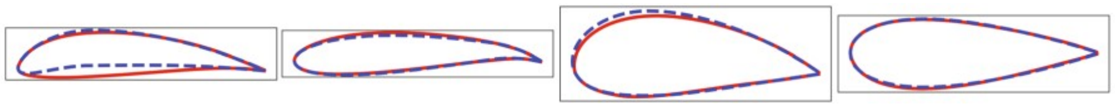


Figure 4.1: Illustration of different auto-encoder types
(a) Auto-encoder (b) Variational auto-encoder (c) Conditional variational auto-encoder
Image Source: Yonekura et al. [42]



(a) Training Phase



(b) Testing Phase

Figure 4.2: Blue dashed curves represent the inputs for training the auto-encoder, while the red curves are the corresponding outputs generated by the trained auto-encoder for airfoil geometry prediction.
Image Source: Yonekura et al. [42]

DeepSDF
DeepSDF is a ready to use library which can be used for generative modelling of complex geometries using auto-encoders. DeepSDF was developed by Park et al. [32]. The main advantage of using DeepSDF comes from its ability to work with very complex geometries by representing them as signed distance functions (SDFs). SDFs inherently use binary classification of coordinate points as lying outside the geometry (SDF > 0) or inside the geometry (SDF < 0). Thereby, the contour SDF = 0 gives the shape of the geometry. This approach can be easily extended to 2D as well as 3D geometries.

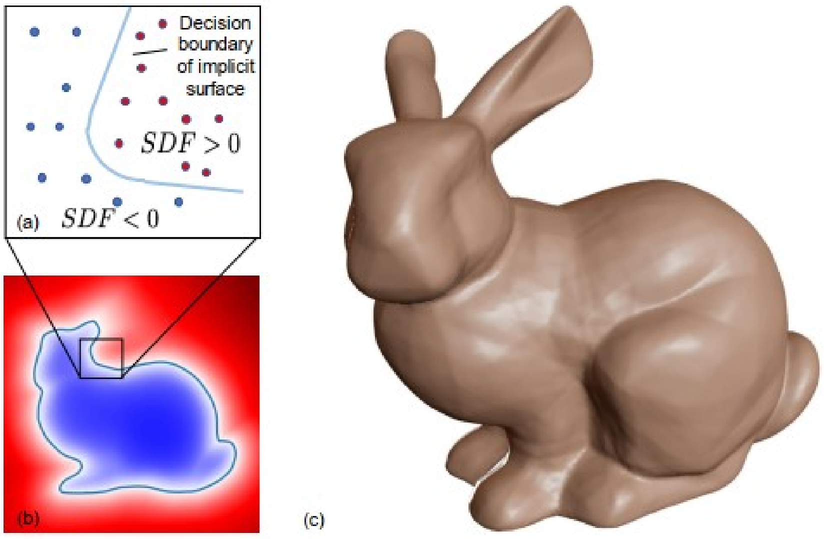


Figure 4.3: Representation of complex geometry using signed distance function (SDF)
Image Source: Park et al. [32]

Another interesting contribution that was given by Park et al. [32] was pertaining to auto-encoders, where they proposed to train an encoder less network called auto-decoder. In a conventional auto-encoder, the encoder is rendered useless after training as the decoder along with the latent space vector can successfully generate the desired outputs. Thereby, one can simply train the latent space vector with the decoder in a supervised manner by initialising the decoder weights and keeping them as constant.

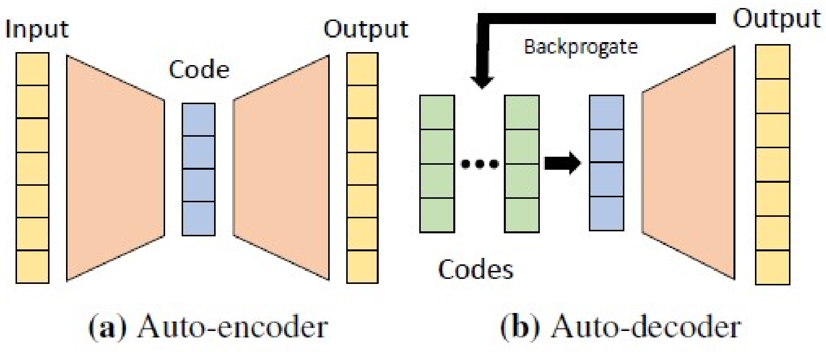


Figure 4.4: Architecture of auto-encoder and auto-decoder
Image Source: Park et al. [32]

4.2. Generative Adversarial Networks

Generative Adversarial Networks (GANs) were proposed by Goodfellow et al. [12] for generative modelling. This formulation involves two networks: generator $\mathcal{G}$ and discriminator $\mathcal{D}$. The role of the generator is to generate new data which should mimic a given reference dataset. The role of the discriminator is to successfully distinguish between the data generated by the generator and the actual reference dataset. Therefore, it can be concluded that the training has converged, when the generator is able to successfully fool the discriminator as described in figure 4.5. The objective function for training a generative adversarial network is given by evaluating the value function V as a function of the generator and the discriminator probability distributions $\mathcal{G}$ and $\mathcal{D}$ respectively :

$$\underset{\mathcal{G}}{min}\ \underset{\mathcal{D}}{max}\ V(\mathcal{D},\mathcal{G}) = \underbrace{\mathbb{E}_{x \approx p_{data}(x)}[log\ \mathcal{D}(x)]}_{\text{Discriminator's prediction on real data } x} + \underbrace{\mathbb{E}_{z \approx p_z(z)}[log(1-\mathcal{D}(G(z))))]}_{\text{Discriminator's prediction on fake data produced by generator } G(z)} . \tag{4.2}$$

The objective function represented in equation 4.2 is used for training generative adversarial networks. The goal of the discriminator is to maximize the given value function V, to be able to successfully distinguish between the real data and the fake generated data. On the other hand, the goal of the generator is to minimize the value function V so that it can successfully fool the discriminator into predicting that the fake data coming from the generator is a part of the actual real dataset.

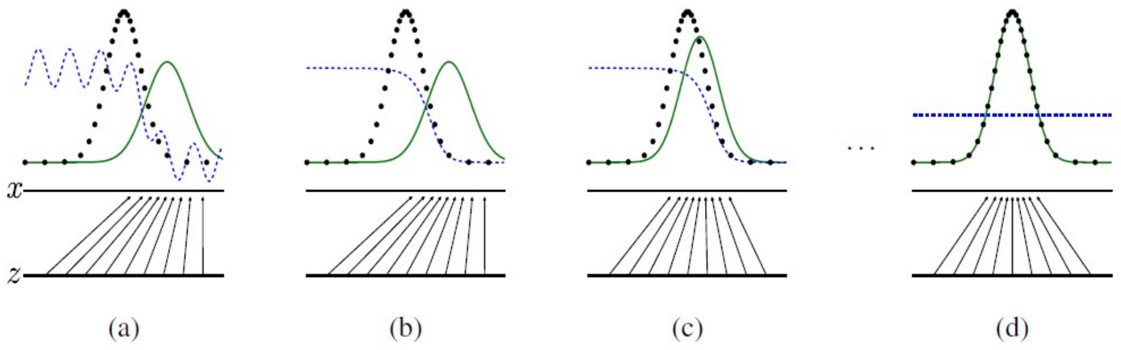


Figure 4.5: GANs training process where the discriminative distribution is given by the blue dashed line, the data distribution is given by the black dotted line and the generative distribution is the green solid line. As the training process proceeds from (a) to (d), the generative distribution moves closer to the data distribution and the discriminative distribution becomes a uniform distribution, making it hard for the discriminator to differentiate between the actual data and generated data, making the discriminative probability 1/2.
Image Source: Goodfellow et al. [12]

These GANs were successully used by Chen et al. [7] for generating airfoil shapes. The main shortcoming of using this approach is that it gives discrete outputs, as this framework was developed for working with pixelated images. This might lead to inappropriate shapes being generated for airfoils which are not continuous or sufficiently smooth. To overcome this shortcoming, one can use bezier curves [8], which are capable of generating sufficiently smooth functions subject to a set of given control points. Thereby, the discrete outputs generated from GANs can be fed into the bezier framework as control points, thereby ensuring a smooth and continuous geometry being generated for the resultant airfoils.

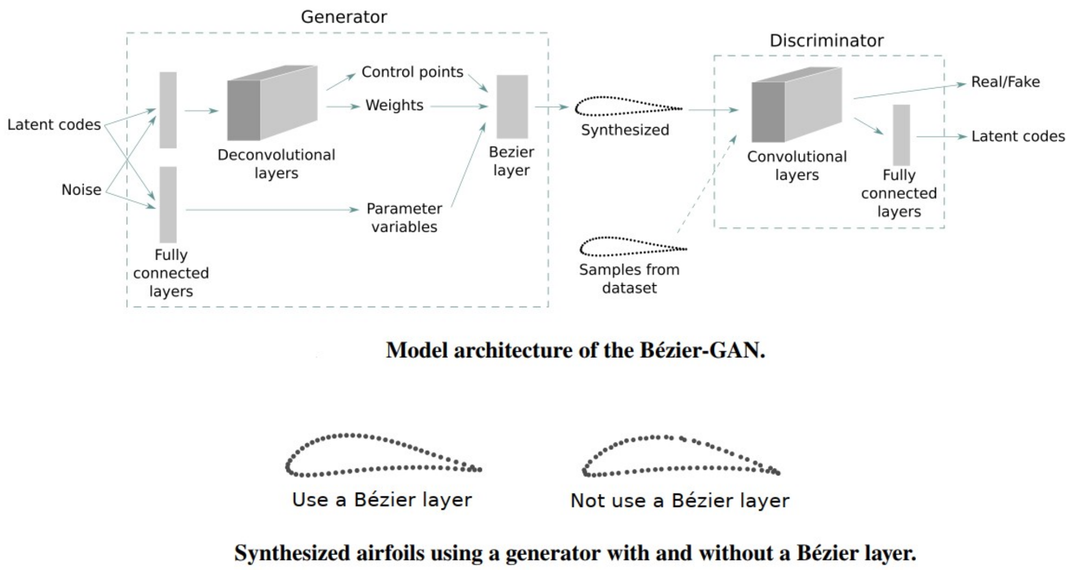


Figure 4.6: *Image Source: Chen et al.* [7]

5 Parametrized PINNs for the Poisson Equation

In the previous chapters 1,2,3 and 4, an extensive literature study was presented to summarize the work done by various researchers around the world in the fields of physics informed machine learning and generative modelling. From now on, that is in chapters 5, 6, 7 and 8, the details regarding the work carried out by the author as a part of the undertaken thesis have been given. The first case that was explored for solving a parametrized system using PINNs was that of the two dimensional Poisson equation, solved over a unit square domain in ξ, η coordinates,

$$\frac{\partial^2 u}{\partial \xi^2} + \frac{\partial^2 u}{\partial \eta^2} = f(\xi, \eta). \tag{5.1}$$

To attain a formulation with parametrized domain for the given Poisson equation, a coordinate transformation was performed, to convert the system domain from a unit square to an arbitrary rectangular domain, defined by its geometrical parameters namely the horizontal dimension (length) X_0 and the vertical dimension (height) Y_0. This process is visualised in figure 5.1.

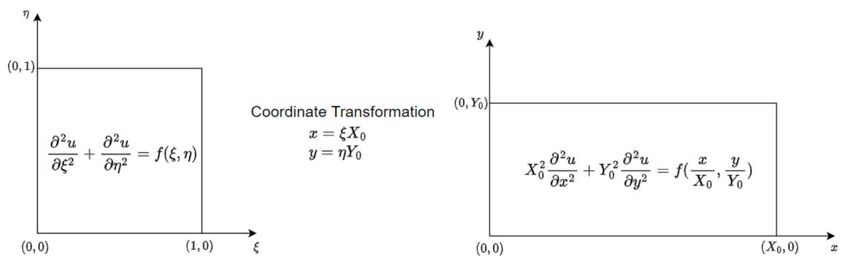
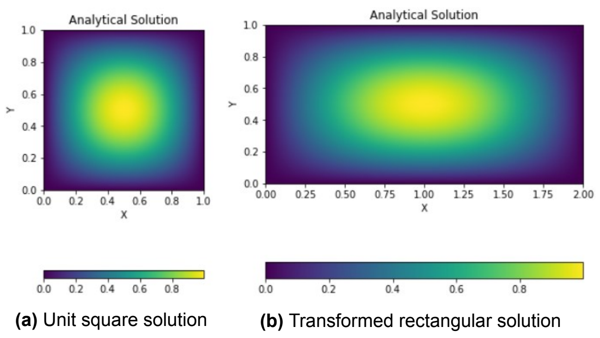

η
$(0,1)$
$\frac{\partial^2 u}{\partial \xi^2} + \frac{\partial^2 u}{\partial \eta^2} = f(\xi,\eta)$
$(0,0)$
$(1,0)$
ξ
Coordinate Transformation
$x = \xi X_0$
$y = \eta Y_0$
y
$(0, Y_0)$
$X_0^2 \frac{\partial^2 u}{\partial x^2} + Y_0^2 \frac{\partial^2 u}{\partial y^2} = f(\frac{x}{X_0}, \frac{y}{Y_0})$
$(0,0)$
$(X_0,0)$
x

Figure 5.1: Transformation scheme for converting the Poisson equation from the unit square coordinate system to a generalised rectangular coordinate system



(a) Unit square solution **(b)** Transformed rectangular solution

Figure 5.2: Analytical solutions for original unit square domain and transformed rectangular domain

Source Function

For a given variable $u(\xi, \eta)$ in the unit square domain $\hat{\Omega} = (0,1) \times (0,1)$, for the given Poisson equation (5.1), we assume the solution to be:

$$u(\xi, \eta) = \text{sin}(\pi\xi)\text{sin}(\pi\eta), \tag{5.2}$$

Thereby, the desired source function f to satisfy the given solution for the Poisson equation can be derived as follows:

$$\frac{\partial u}{\partial \xi} = \pi\text{cos}(\pi\xi)\text{sin}(\pi\eta),$$

$$\frac{\partial^2 u}{\partial \xi^2} = -\pi^2\text{sin}(\pi\xi)\text{sin}(\pi\eta),$$

similarly,

$$\frac{\partial^2 u}{\partial \eta^2} = -\pi^2\text{sin}(\pi\xi)\text{sin}(\pi\eta),$$

Therefore,

$$f(\xi, \eta) = \frac{\partial^2 u}{\partial \xi^2} + \frac{\partial^2 u}{\partial \eta^2} = -2\pi^2\text{sin}(\pi\xi)\text{sin}(\pi\eta) = -2\pi^2 u. \tag{5.3}$$

5.1. Coordinate Transformation

To convert the Poisson equation from the original unit square domain to any given rectangular domain of arbitrary dimensions defined by parameters X_0 and Y_0, we can perform the coordinate transformation as follows:

$$x = \xi X_0, \quad \xi \in [0,1], \tag{5.4}$$

$$y = \eta Y_0, \quad \eta \in [0,1]. \tag{5.5}$$

The derivatives can be converted between the two coordinate systems using the chain rule of derivatives:

$$\begin{aligned}
\frac{\partial u}{\partial x} &= \frac{\partial u}{\partial \xi}.\frac{\partial \xi}{\partial x} = \frac{\partial u}{\partial \xi}.\frac{1}{X_0}, \\
\frac{\partial^2 u}{\partial x^2} &= \frac{1}{X_0}.\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial \xi}\right) = \frac{1}{X_0}.\frac{\partial}{\partial \xi}\left(\frac{\partial u}{\partial x}\right), \\
&= \frac{1}{X_0}.\frac{\partial}{\partial \xi}\left(\frac{\partial u}{\partial \xi}.\frac{1}{X_0}\right) = \frac{1}{X_0^2}.\frac{\partial^2 u}{\partial \xi^2}, \\
\frac{\partial^2 u}{\partial \xi^2} &= X_0^2.\frac{\partial^2 u}{\partial x^2}.
\end{aligned}$$

Similarly,

$$\frac{\partial^2 u}{\partial \eta^2} = Y_0^2.\frac{\partial^2 u}{\partial y^2}.$$

Therefore, the corresponding governing equation in the transformed coordinates can be written as:

$$X_0^2\frac{\partial^2 u}{\partial x^2} + Y_0^2\frac{\partial^2 u}{\partial y^2} = f(\frac{x}{X_0}, \frac{y}{Y_0}), \tag{5.6}$$

with the corresponding solution being

$$u = \text{sin}(\pi\frac{x}{X_0})\text{sin}(\pi\frac{y}{Y_0}).$$

It should be noted that the assumed solution in both the coordinate systems inherently ensures a homogenous Dirichlet boundary condition, therefore it has not been explicitly mentioned.

5.2. Solution for a Single Geometry

The first formulation of the deep learning pipeline was developed to solve the given Poisson equation (5.6) on a single rectangular geometry, with the intention of subsequently being extended to accommodate variable geometries parametrized by their dimensions X_0 and Y_0, respectively. For this case, the geometry parameters were set to $(X_0, Y_0) = (2, 1)$.

Mesh

The mesh in this context refers to the set of input points that the neural network would receive during the training process for learning the desired Poisson equation. The mesh needs to be divided into two main groups of points, namely interior (or collocation) points and the boundary points. The interior points ensure that the neural network is able to learn the desired PDE throughout the interior of the domain, whereas the boundary points are provided to impose the desired boundary condition in addition to the interior PDE solution. The different groups of mesh points have been illustrated in figure 5.3. A standardised mesh resolution of $20X_0 \times 20Y_0$ collocation points and $50X_0 \times 50Y_0$ boundary points was used for all the test cases.

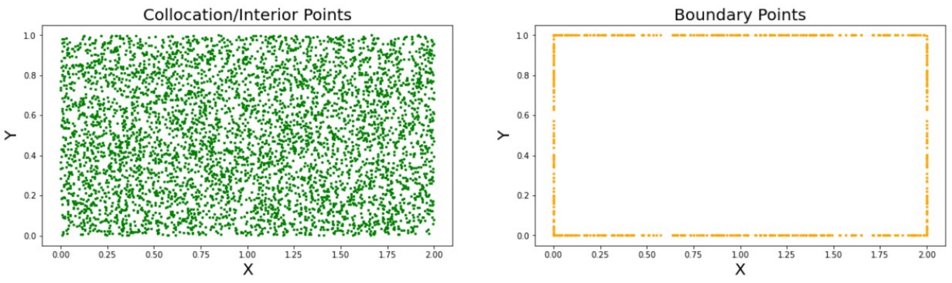


Figure 5.3: Distribution of interior and boundary points for the input mesh

Neural Network

As explained in chapter 2 and depicted in figure 2.7 for PINNs, the neural network for a PINN formulation consists of two parts, where the first part maps the input mesh points (x and y along with other inputs if necessary) to the desired solution variable (single variable u in the present case) and the second part formulates the desired PDE by taking analytical derivatives of the output variables, and formulating the loss function in the form of a PDE residual (explained in the upcoming section). The (representative) structure of the solution mapping part of the neural network is illustrated in figure 5.4. For capturing the solution over a single geometry, we simply need to map the coordinates (x and y) of the mesh points to the solution u. The network dimensions were finalised to be 8 hidden layers with 40 neurons per layer, obtained via a parametric study. 'Tanh' was used as the activation function to allow for the calculation of second order derivatives via automatic differentiation to define the PDE residual as loss function.

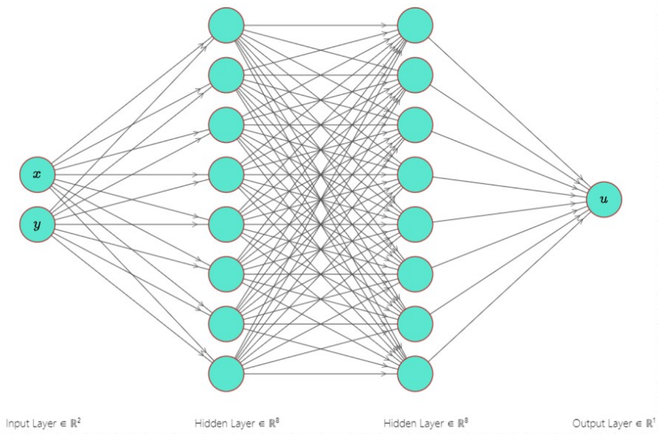


Figure 5.4: Solution mapping fully connected neural network architecture for a fixed geometry. This is a representative neural network to show the mapping from inputs to outputs, the actual neural network contained 8 hidden layers with 40 neurons per layer

Loss Function Formulation

Ideally the loss functions for physics informed neural networks should eliminate the need to using any sorts of training data, as such data for training neural networks to learn PDEs would be needed to be generated via classical numerical techniques, thereby defeating the entire purpose of attempting to replace classical numerical solver schemes with physics informed deep learning. The neural network has to decode two types of input data points, namely the interior/collocation points and the boundary points. The solution values at the collocation points need to be obtained by solving the desired PDE, whereas the boundary points come with their prescribed data values (assuming only Dirichlet boundary conditions throughout the scope of this thesis) which also need to be learnt and satisfied by the underlying neural network simultaneously with the PDE solution. Therefore, as described in chapter 2 the global loss function formulations for PINNs are usually a weighted superposition of two loss formulations, namely and PDE loss (for the interior points) and a data loss (for boundary points as well as some interior data points if training data is available). In our case, we assume the weights of PDE loss and data loss should add to unity. Therefore our loss formulation becomes a slightly modified version of equation 2.2

$$\mathcal{L}_{total} = w\mathcal{L}_{PDE} + (1-w)\,\mathcal{L}_{data}. \tag{5.7}$$

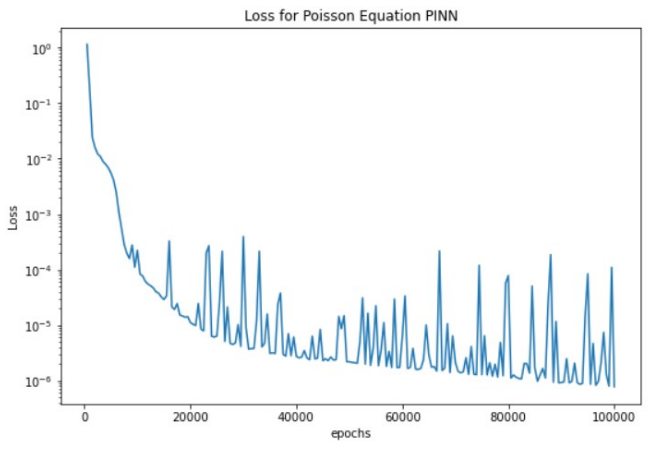
The PDE loss would simply be the residual of the governing equation 5.1.

$$\mathcal{L}_{\text{pde}} = \frac{1}{N_{\text{pde}}}\sum_{j=1}^{N_{\text{pde}}}\left(X_0^2\frac{\partial^2 u_j}{\partial x^2} + Y_0^2\frac{\partial^2 u_j}{\partial y^2} - f(\frac{x_j}{X_0}, \frac{y_j}{Y_0})\right)^2_{\text{interior points } j}, \tag{5.8}$$

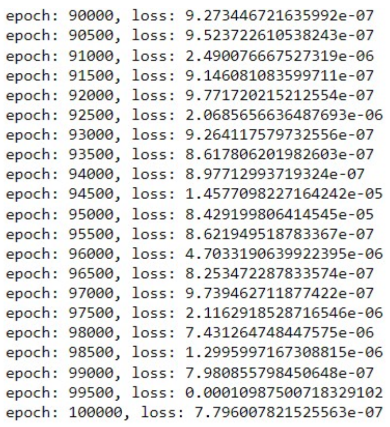
$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}}\sum_{i=1}^{N_{\text{data}}}(u(x_i, y_i) - u_i)^2_{\text{boundary points } i}. \tag{5.9}$$

Training of Neural Network

The weight w used for the loss function was set to the value of 0.01, considering that it is quite difficult for the PINN to satisfy the homogeneous Dirichlet boundary conditions along with the interior PDE solution, which may vary from case to case. The learning rate was set to be 0.0001. The total number of iterations was set to be 100000. Moreover, the set of input mesh points (interior and boundary points) were regenerated randomly after every 5 iterations, thereby increasing the overall resolution of the input mesh without increasing the computational load for a single iteration. It should be noted that this technique of super-resolution by periodic perturbation of the input mesh is a distinct advantage of neural network based techniques over classical numerical methods, which also help in convergence to very low loss functions values, along-with ensuring a regularisation mechanism against overfitting on a fixed mesh.



(a) Loss variation with iterations/epochs

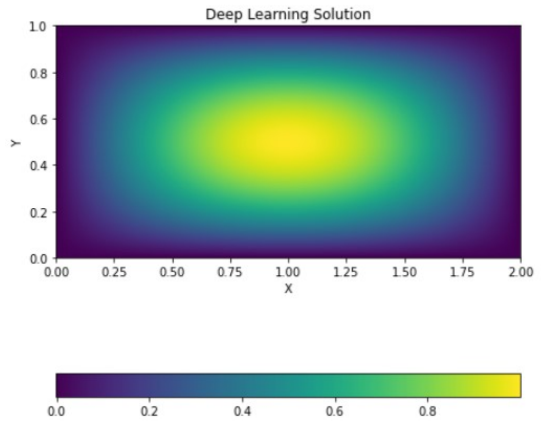


```
epoch: 90000, loss: 9.273446721635992e-07
epoch: 90500, loss: 9.523722610538243e-07
epoch: 91000, loss: 2.490076667527319e-06
epoch: 91500, loss: 9.146081083599711e-07
epoch: 92000, loss: 9.771720215212554e-07
epoch: 92500, loss: 2.0685656636487693e-06
epoch: 93000, loss: 9.264117579732556e-07
epoch: 93500, loss: 8.617806201982603e-07
epoch: 94000, loss: 8.97712993719324e-07
epoch: 94500, loss: 1.4577098227164242e-05
epoch: 95000, loss: 8.429199806414545e-05
epoch: 95500, loss: 8.621949518783367e-07
epoch: 96000, loss: 4.7033190639922395e-06
epoch: 96500, loss: 8.253472287833574e-07
epoch: 97000, loss: 9.739462711877422e-07
epoch: 97500, loss: 2.1162918528716546e-06
epoch: 98000, loss: 7.431264748447575e-06
epoch: 98500, loss: 1.2995997167308815e-06
epoch: 99000, loss: 7.980855798450648e-07
epoch: 99500, loss: 0.00010987500718329102
epoch: 100000, loss: 7.796007821525563e-07
```

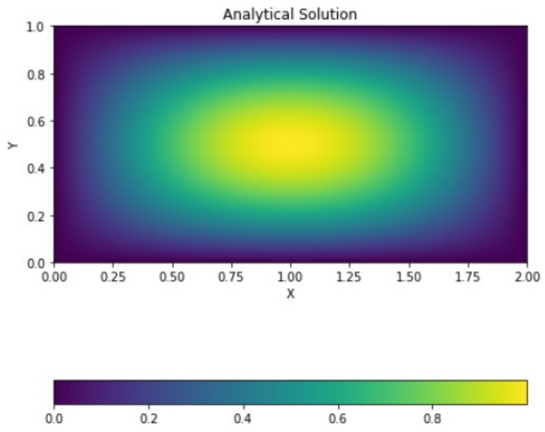
(b) Loss values attained in final result

Figure 5.5: Loss variation during training for Poisson equation on a fixed geometry

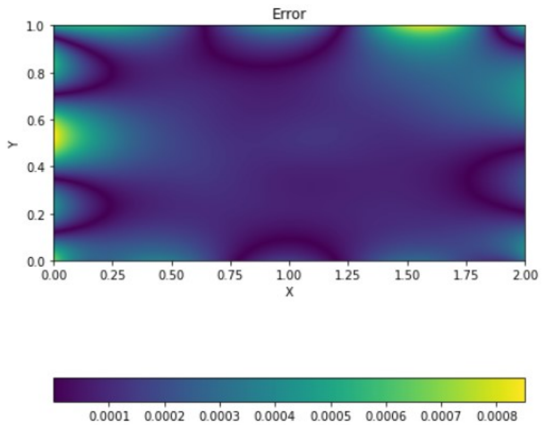
Results
As depicted in figure 5.5, the loss converged quite smoothly to a very low value of the order $\mathcal{O}(1e-7)$, during the training process, owing to the optimal selection of parameters (attained after repetitive hit and trial while varying the various hyperparameters such as loss weights, learning rate, width and depth of the network etc.) in addition to the perturbation strategy of mesh points. Therefore, the final results from the trained PINN were expected to be highly accurate, which have been illustrated in figure 5.6 as follows. It can be observed that the point wise errors never exceed $\mathcal{O}(1e-4)$, thereby making the predictions reasonably accurate and reliable.



(a) Deep learning based solution contours



(b) True solution contours



(c) Point wise error variation

Figure 5.6: PINN based results for Poisson equation on a single transformed domain

5.3. Solution for Parameterized Variable Geometry

After successfully obtaining more than satisfactory results for training a PINN on a Poisson equation for a fixed geometry, the next step was taken to build a deep learning pipeline capable of producing instantaneous results upon training for a given equation (Poisson equation) on a variable geometry, parametrized by its dimensions X_0 and Y_0, without the need for performing repeated simulations for each geometry as opposed to classical numerical techniques. It was decided to include the geometry parameters X_0 and Y_0 also in the input, expecting the neural network to be able to capture the relationship of the mesh points with their respective geometries. This approach/architecture shown in figure 5.7 turned out to be successful in capturing various geometries for the given Poisson equation, when used in conjunction with the same hyper-parameters and training strategy as used for the previous case with fixed geometry.

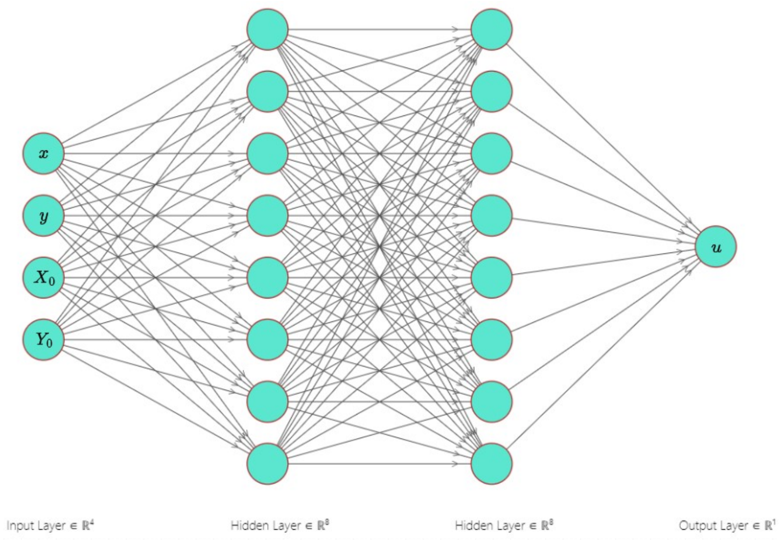


Figure 5.7: Solution mapping fully connected neural network architecture for parametrized varying geometry. This is a representative neural network to show the mapping from inputs to outputs, the actual neural network contained 8 hidden layers with 40 neurons per layer

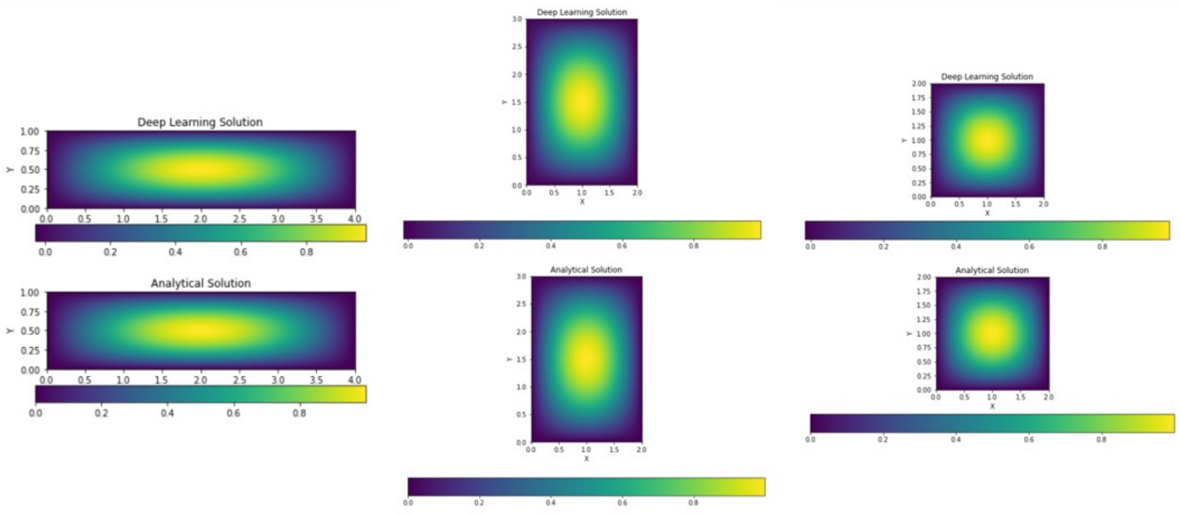
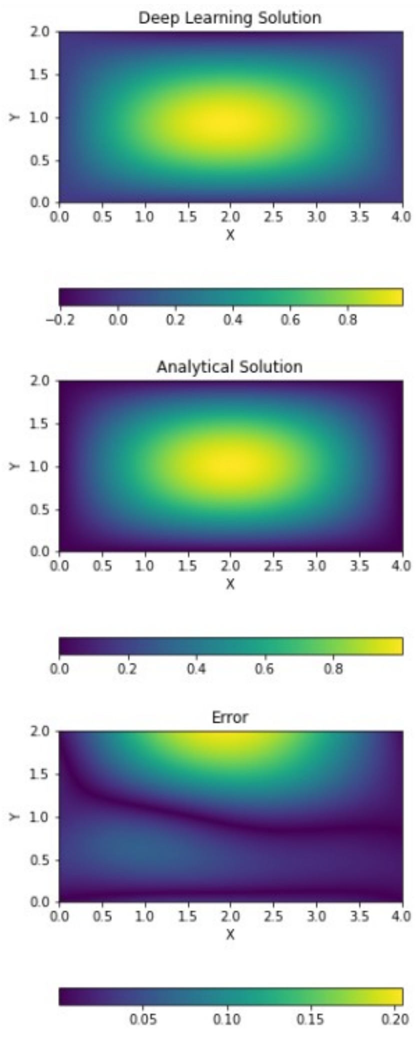


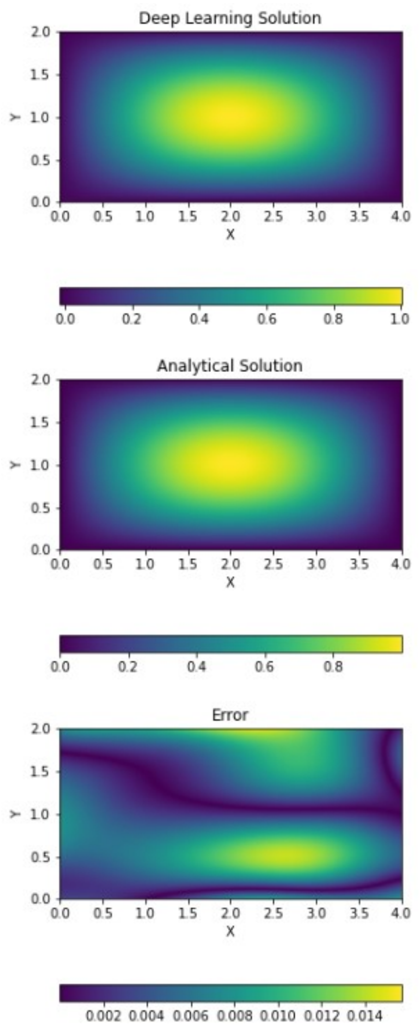
Figure 5.8: Results for trained PINN on new architecture (figure 5.7) for different geometry inputs without repetitive simulations

Test Cases

The geometries for input meshes were generated by randomly picking the dimensions X_0 and Y_0 during the perturbation steps within the training process as explained in the previous section. Both X_0 and Y_0 were independently assigned a dimension unit from a given selection of values, for example one of the sample sets being an integer among $\{1, 5, 9\}$, called the 'wide' sample set, as the given subsequent values differ by a margin of 4 units. Similarly, a 'narrow' sample set with the dimension values comprising $\{1, 3, 5, 7, 9\}$ was also tested. The goal of this study was to determine the impact of the frequency of the geometry parameters responsible for generating the training data points on the interpolation capability of the neural network, as ideally we would prefer to be able to train the neural network on limited geometry parameter values, without compromising with our interpolation accuracy of the trained PINN. The final result of the study was that the underlying PINN which was trained with the 'wide' sample set of parameter values performed better on domains which it had seen during the training (for example $(X_0, Y_0) = (5, 1)$), whereas the PINN trained with the 'narrow' sample set performed better during interpolation on domains which had not been seen during the training process by either of the PINNs. But we witness a compromise on the overall accuracy of the PINN results compared to the case for fixed geometry. This behaviour of the PINNs could be attributed to the fact that we are trying to accommodate more geometry-solution pairs on a given neural network of fixed dimensionality (weights and biases), thereby compromising the the accuracy of the solution for a single geometry.

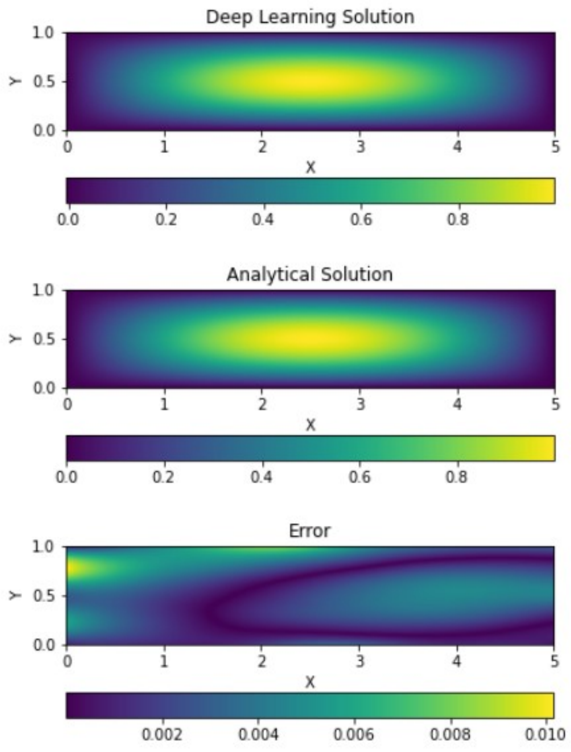


(a) Results for a geometry not seen during training on wide sample set

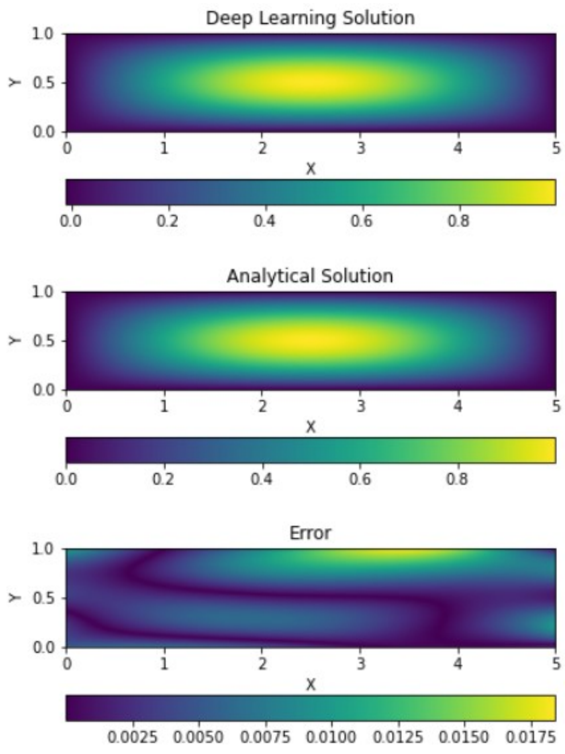


(b) Results for a geometry not seen during training on wide sample set

Figure 5.9: Comparison of results between PINNs trained on wide and narrow sample sets for a geometry not seen during training



(a) Results for a geometry seen during training on wide sample set



(b) Results for a geometry seen during training on wide sample set

Figure 5.10: Comparison of results between PINNs trained on wide and narrow sample sets for a geometry seen during training

6 Parameterized PINNs for the Stokes Equations

The Stokes equations (in two dimensions), ideally used for simulating very low Re ($U_{\text{ref}}L_{\text{ref}}/\nu$) flows, are given as follows:

$$\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)=\frac{\partial p}{\partial x}, \tag{6.1}$$

$$\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)=\frac{\partial p}{\partial y}, \tag{6.2}$$

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0. \tag{6.3}$$

It can be seen that equations (6.1) and (6.2) represent the x and y momentum terms respectively for the Stokes equations, which are derived by ignoring the inertial terms from Navier-Stokes. It must be noted that these equations are in dimensionless form as the diffusion terms have been normalised by the Reynolds number Re. It is significant to use dimensionless equations while training PINNs or any simple neural network to ensure better convergence and ease of training, which is ensured when the neural networks are required to map inputs and output values of a limited range ($[0,1] \mapsto [-1,1]$) [30]. In our case, to make the conversion between dimensional and dimensionless forms of the equations easy, we have chosen the free stream reference velocity (U_{ref}) as 1 m/s and the free stream reference pressure as gauge 0. The chord length of the airfoil (L_{ref}) being 1 m makes the Reynolds number the inverse of the kinematic viscosity $Re = 1/\nu$. The equation (6.3) is the continuity equation suitable for incompressible flows. Therefore, in this particular test case, our goal is to solve a system of three equations simultaneously using physics informed neural networks, first on a single NACA airfoil geometry (sample geometry in figure 6.1) and subsequently extend it to a collection of airfoil geometries parameterized by NACA 4 digit series formulation such as,

NACA CPTT

eg.

NACA 4812

Where,

1. The first digit C stands for the magnitude of maximum camber as a percentage of chord length.
2. The second digit P signifies the position of the maximum camber from the leading edge as a percentage of chord length.
3. The last two digits TT define the maximum thickness of the airfoil as a percentage of chord length.

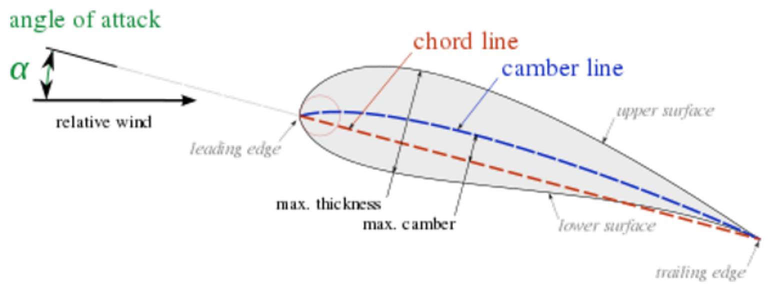


Figure 6.1: NACA 4 series airfoil naming convention
Image Source: en.wikipedia.org

6.1. Numerical Solution for Airfoil Geometry

The first test case for training a PINN architecture with the system of Stokes equations was applied to NACA 4812 airfoil. The numerical solution to aid as reference for the given equations (6.1), (6.2) and (6.3) was obtained via the FEniCS [23] finite element library in Python, and the code for this purpose was inspired from the tutorial given in [11]. The geometry used for solving the Stokes flow is given in figure 6.2.

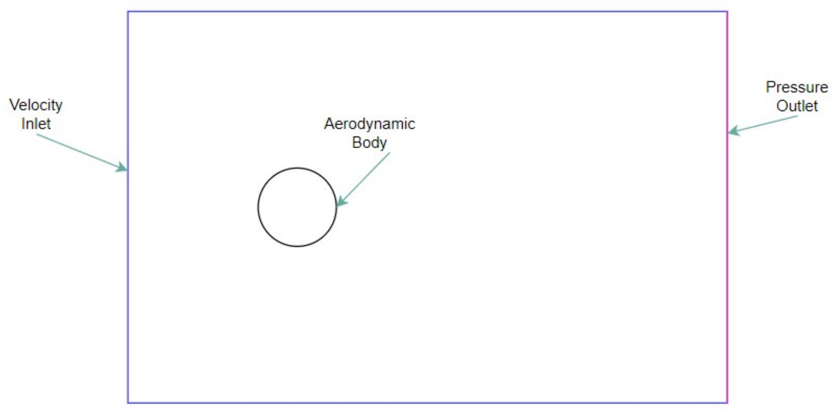


Figure 6.2: Simulation geometry for airfoil flow

Parameter	**Value/Strategy**:
Velocity Inlet Condition	Prescribed Dimensionless Velocity (1)
Pressure Outlet Condition	Prescribed Pressure $(0 \ gauge)$
Aerodynamic Body	Airfoil (no slip $\mathbf{u} = 0$)
Kinematic Viscosity (ν)	$1.5e-2 \ m^2/s$
Airfoil Chord Length	$1m$
Angle of Attack	$5 \ deg$
Reynolds Number	66.67

Table 6.1: Parameters used for numerical solution of airfoil flow using Stokes equations

By setting up the free stream velocity to 1 m/s and chord length to 1 m, the Reynolds number is simply the reciprocal of the kinematic viscosity, thereby the resulting equations are identical in dimensional as well as dimensionless format, which makes things easier for training as well as for post processing purposes. The Reynolds number has been kept very low to justify the use of the Stokes equations. The simulations for the single geometry case were performed for the airfoil NACA 4812, the results for which are illustrated in figure 6.3.

The FEniCS package provides numerical solutions for the desired PDEs using the finite element methodology, for which we need to convert the given system of equations into a weak formulation, which is obtained by introducing a weighting function and integrating the product of our desired PDE with the weighting function over each mesh element. The general formulation of a weak formulation is given by [11]:

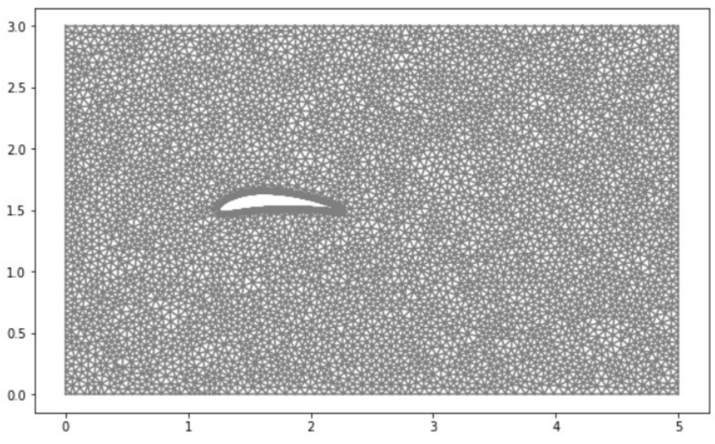
$$a((\mathbf{u},p),(\mathbf{v},q)) = L((\mathbf{v},q)), \tag{6.4}$$

where a is referred to as a bi-linear form and L is the linear form. The unknowns namely velocity vector **u** and pressure p, are referred to as trial functions (as they are assumed to vary linearly or via a defined polynomial between the element nodes) and $\mathbf{v}, q$ are the corresponding test/weight functions. The function space of the test functions used determines the type of finite element method that is formulated, for example Galerkin method or Ritz method etc. The corresponding weak formulation for the Stokes equation comes out to be:

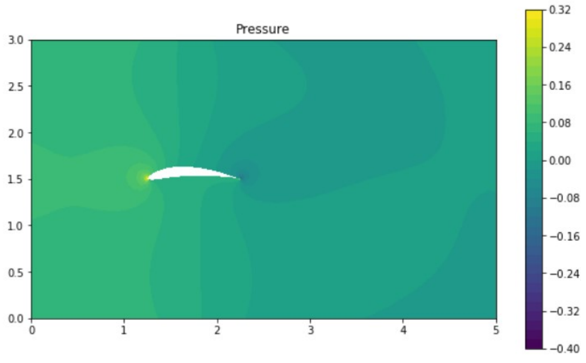
$$a((\mathbf{u},p),(\mathbf{v},q)) = \int_{\Omega} (\nabla\mathbf{u} : \nabla\mathbf{v} - p\nabla.\mathbf{v} + q\nabla.\mathbf{u})\, dV, \tag{6.5}$$

$$L((\mathbf{v},q)) = \int_{\Omega_p} p_0\mathbf{n}.\mathbf{v}\, dS = 0, \qquad (p_0 = 0). \tag{6.6}$$

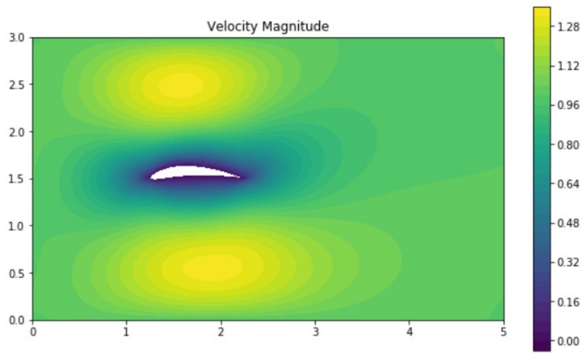
These equations along with the boundary conditions were coded into the FEniCS framework, the code for which can be referred from the appendix A section A.1. The numerical results for the fluid flow simulation using this FEM formulation have been illustrated in the following figure 6.3, where the pressure contours are given in figure 6.3b and the velocity contours are given in figure 6.3c. The reader can refer to [11] for more details on finite element method and its practical application in fluid flow simulations, as this is beyond the scope of the current thesis.



(a) FEniCS mesh for airfoil simulation



(b) FEniCS solution contours for pressure



(c) FEniCS solution contours for velocity

Figure 6.3: FEniCS based results for Stokes equations on a single airfoil domain

6.2. Deep Learning Solution for Single Airfoil Geometry

The deep learning formulation for solving the system of Stokes equations was first implemented to solve for a single airfoil geometry at a given angle of attack. The methodology used for developing such a pipeline was quite similar to the case of Poisson equations, requiring some modifications which have been highlighted as follows.

Mesh

As explained in the previous chapters, the mesh for a PINN formulation requires two sets of input points, namely the interior points and the prescribed boundary points (figure 6.4). However, when the case of Poisson equation was implemented using the same strategy as Poisson equation (that is fully unsupervised training), the neural network failed to converge. Therefore, it was decided to include some training data for ease of training and convergence. Therefore in the present case for solving Stokes equation via PINNs, the input mesh consists of the interior points as usual, and a set of 'data points', comprising both the boundary points as well as some mesh points in the interior of the domain with prescribed velocity and pressure values. Thereby, the training strategy in this case converts from fully unsupervised to semi-supervised, having both data loss and PDE loss within the global loss function. The input data mesh and the prescribed data values were imported from the numerical solver FEniCS. Therefore, the input mesh for this case was a combination of randomly generated collocation points with the resolution of $20X_0 \times 20Y_0$ along with the FEniCS mesh points imported as it is. Moreover, a for loop was implemented to generate the airfoil cavity by eliminating the collocation points lying within the airfoil contour.

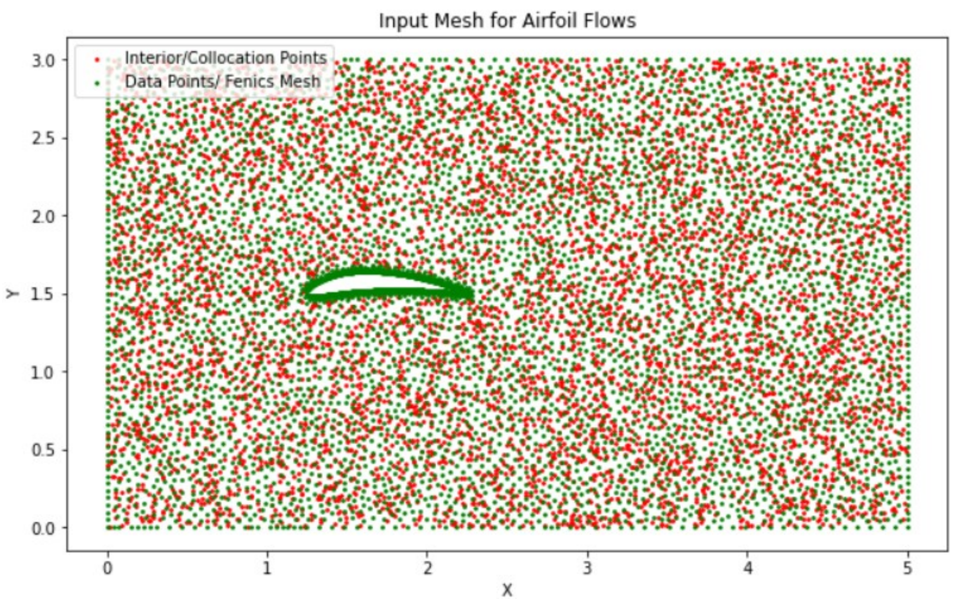


Figure 6.4: Distribution of interior and boundary points for the input mesh of airfoil Stokes flow

Neural Network

The neural network for this case was formulated on the same principles as for the single fixed geometry case for the Poisson equation, mapping the input mesh coordinate points x and y to the output values u (x velocity component), v (y velocity component) and pressure p. Since there are three outputs required in this neural network, the number of neurons per hidden layer were taken to be 100, whereas 4 hidden layers were generated within the neural network. The 'tanh' activation function was used as in the Poisson case.

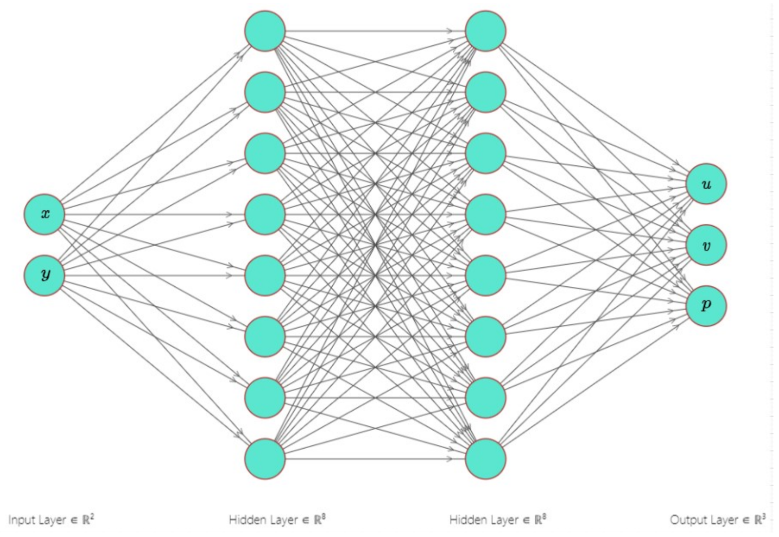


Figure 6.5: Neural network architecture for a fixed airfoil geometry. This is a representative neural network to show the mapping from inputs to outputs, the actual neural network contained 4 hidden layers with 100 neurons per layer

Loss Function Formulation
The loss function consisted of the PDE loss and the data loss as described in equation 5.7. The data loss was simply the mean squared error of the PINN output values with respect to the FEniCS solution values. The PDE loss for this case consisted of a combination of 3 PDE residuals as shown in the following equations. The weight function w to balance the PDE and data losses was kept at 0.6 (in favour of data loss, finalised after a parametric study).

$$e_1 = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial p}{\partial x}, \tag{6.7}$$

$$e_2 = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial p}{\partial y}, \tag{6.8}$$

$$e_3 = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}, \tag{6.9}$$

$$\mathcal{L}_{PDE} = e_1^2 + e_2^2 + e_3^2. \tag{6.10}$$

Training of the Neural Network
The neural network was trained following the same approach that was adopted for Poisson equation in chapter 5. The collocation points were regenerated randomly after every 5 iterations, to achieve faster convergence and attain lower loss values by using the principle of super-resolution. The learning rate was set to 0.0001 while the number of iterations were set to 1 million.

Results
The loss function after the completion of 1 million iterations came out to be of the order $\mathcal{O}(1e-4)$. It should be noted that the purpose of simulating Stokes flow over airfoils via PINNs was merely as a preparation for understanding the framework to implement Navier Stokes. The conclusions and learnings from the analysis of Stokes flow have been used to develop the algorithm for the implementation of Navier Stokes, and all the measures to improve accuracy and reduce the residual loss function are taken up while training those PINNs respectively. The results of the Stokes simulation for NACA 4812 have been illustrated in figure 6.6. It can be observed from the figures that the PINN estimations for velocity match really well with the expected numerical solution, although the pressure predictions are highly inaccurate, thereby generating a need for providing a stronger weight value in favour of the training data loss for more accurate predictions of the desired pressure contours.

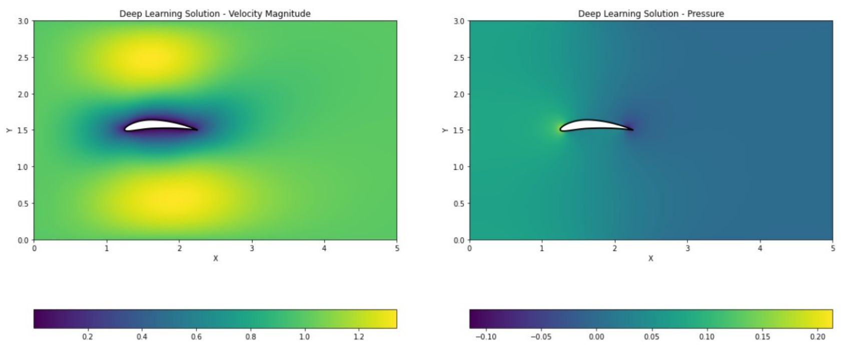


Figure 6.6: Velocity and pressure contours for the solution of Stokes equation via PINN

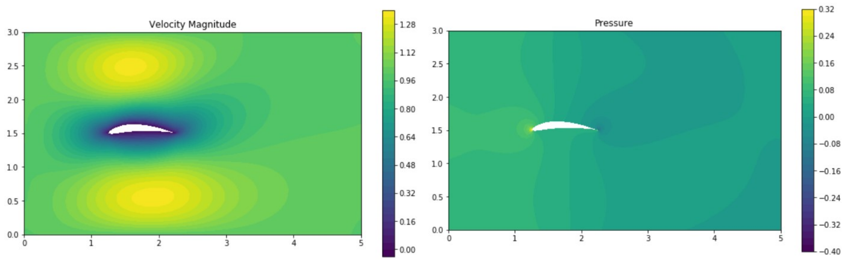


Figure 6.7: FEniCS based results for Stokes equations on a single airfoil domain

6.3. Deep Learning Solution for Variable Airfoil Geometry

After successfully developing a deep learning network for representing Stokes flow for a single airfoil geometry, the next step was to understand if the neural network would be able to learn the parameterized formulation of NACA airfoils and correspondingly be trained for representing the Stokes flow solution for multiple geometries. As discussed in the case for Poisson equation, additional input parameters were defined for the neural network, namely the camber, camber position, airfoil thickness and the angle of attack α, as shown in figure 6.8. For the current case, to keep things simple for the very first simulation of such kind, the study was performed only for symmetric airfoils where the airfoil thickness was varied as the geometry parameter. The formulation developed here would be used for parametric studies of more complex cambered airfoils along with varying angles of attack for the final case with Navier Stokes equations. The simulation was performed over NACA 0006, 0010, 0014 and 0018 at 5 degrees angle of attack. The loss function after training converged to the order $\mathcal{O}(1e-4)$.

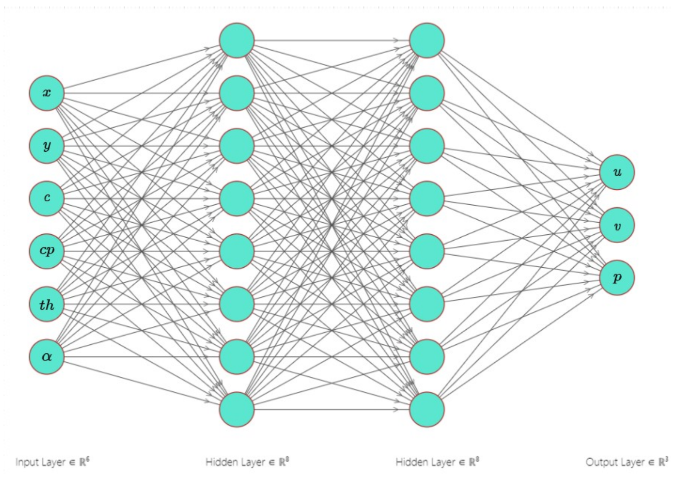

x
y
c
cp
th
α
u
v
p
Input Layer ∈ $\mathbb{R}^6$
Hidden Layer ∈ $\mathbb{R}^8$
Hidden Layer ∈ $\mathbb{R}^8$
Output Layer ∈ $\mathbb{R}^3$

Figure 6.8: Neural network architecture for a varying airfoil geometry.This is a representative neural network to show the mapping from inputs to outputs, the actual neural network contained 4 hidden layers with 100 neurons per layer

Results

After training the neural network for the parameterized formulation of Stokes equations, it was found that the neural network was able to learn the NACA parameterisation of symmetric airfoils. The results for both seen and unseen geometries are illustrated in the following figures.

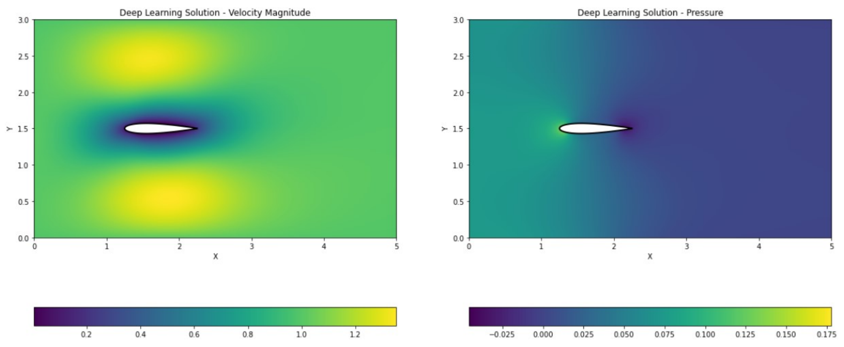


Figure 6.9: Velocity and pressure contours for the solution of Stokes equation via PINN for a seen geometry NACA 0014

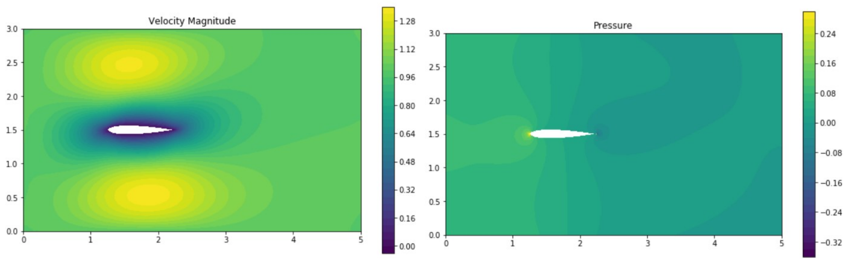


Figure 6.10: FEniCS based results for Stokes equations on NACA 0014

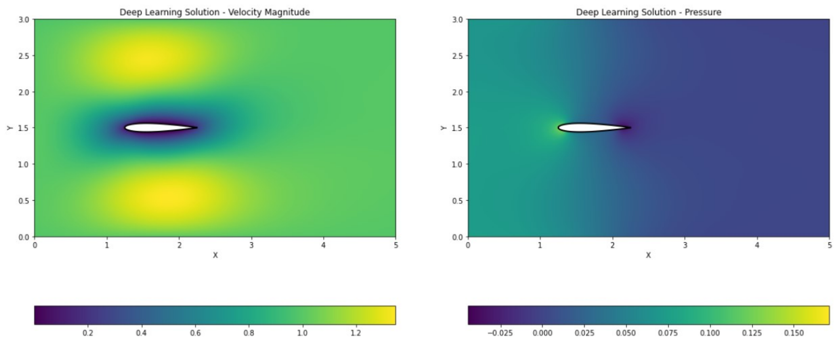


Figure 6.11: Velocity and pressure contours for the solution of Stokes equation via PINN for an unseen geometry NACA 0012

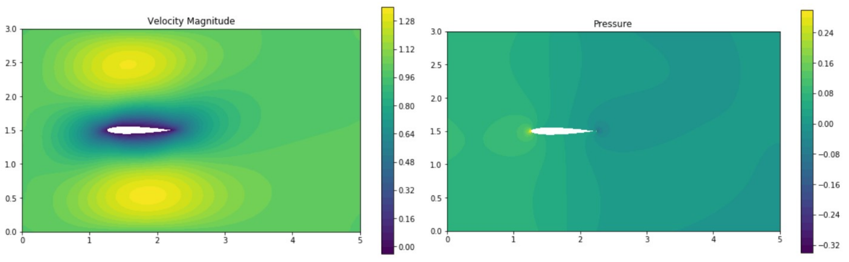


Figure 6.12: FEniCS based results for Stokes equations on NACA 0012

It can be observed that the PINN formulation has indeed learned to identify an airfoil geometry and adjust the solution accordingly. To further test if the PINN is actually sensitive to the geometry input parameters, we can perform a test by inputting a cambered airfoil (NACA 4812), for which this particular neural network was not trained. We can observe from figure 6.13 that the solution contours give nonphysical results upon receiving completely unusual geometry parameters, which is expected.

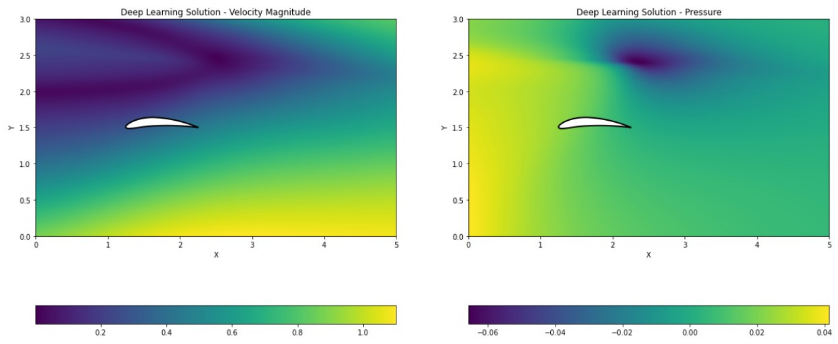


Figure 6.13: Velocity and pressure contours for the solution of Stokes equation via PINN for an unseen geometry NACA 4812

As a result, our formulation of parameterized PINN over the Stokes equations has provided the following observations and conclusions, which have been used for developing our final PINN formulation using Navier Stokes:

1. The major constraint of PINN visualization accuracy is posed by pressure contour results, therefore a higher weight towards data loss is required.
2. Longer training processes for more iterations need to be performed, to achieve lower (more accurate) residual loss values.
3. The PINN formulation is capable of learning NACA airfoil parameterization which are provided along with the mesh coordinates as additional inputs to the neural network.

7 Parameterized PINNs for the Navier Stokes Equations

After successfully developing the framework for representing Stokes equations using PINNs in chapter 6, we can now move on to simulate Navier Stokes equations using the same methodology by making some minor changes. The Navier Stokes equations are the fundamental governing equations for fluid flow simulations, which can be arrived at by including the inertial terms in the Stokes equations.

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial p}{\partial x}, \tag{7.1}$$

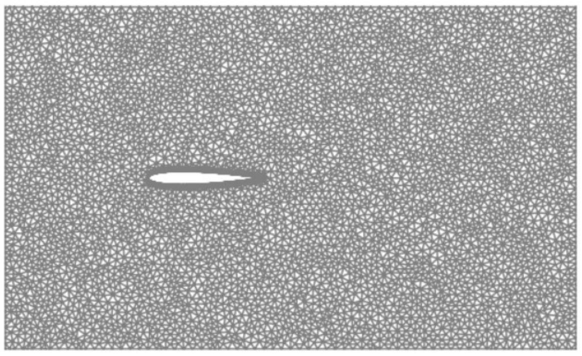
$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial p}{\partial y}, \tag{7.2}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{7.3}$$

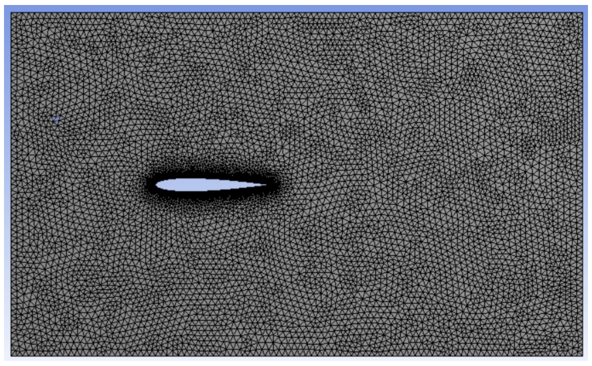
Our choice of parameters and boundary conditions is identical to those for Stokes equation as shown in table 6.1, making the equation identical in dimensional and dimensionless form (as the Reynolds number is the inverse of the kinematic viscosity ν). The only parameter that has been altered is the kinematic viscosity ν which has been changed to $1.5e-3$, to increase the Reynolds number by a factor of 10, making the new Re = 666.67. The corresponding numerical solution from FEniCS solver can also be obtained by adding the inertial term in the weak formulation of the governing equations for Stokes flow. The weak form for Navier Stokes equations is given as [11]:

$$\int_\Omega (\nabla \mathbf{u} : \nabla \mathbf{v} + (\mathbf{u}.\nabla \mathbf{u}).\mathbf{v} - p\nabla.\mathbf{v} + q\nabla.\mathbf{u})\, dV = 0. \tag{7.4}$$

Since, now the most common governing equation for CFD, i.e. incompressible Navier Stokes was simulated, we had the opportunity for validating our FEniCS code with a standardised commercial solver, like ANSYS Fluent in order to compare the force computation algorithm written in FEniCS (which was not present in the code for Stokes equations) with a standardised industrial solver. The same simulation parameters as stated in table 6.1 were used in ANSYS Fluent. An unstructured fine mesh was generated as shown in figure 7.1, without any additional features like boundary layer inflation as they were not available and hence never used for FEniCS simulation. As a result, the solution predictions in terms of velocity and pressure contours along with lift and drag were found to be similar for the FEniCS solver as well as ANSYS Fluent, as shown in figure 7.2 and figure 7.3. Since the Navier Stokes solver in the FEniCS framework is almost exactly similar to the Stokes solver as well, with the exception of the additional inertial term, we assume that since the FEniCS solver is producing accurate results for the Navier Stokes flow, the corresponding solution for the Stokes flow should also be accurate.

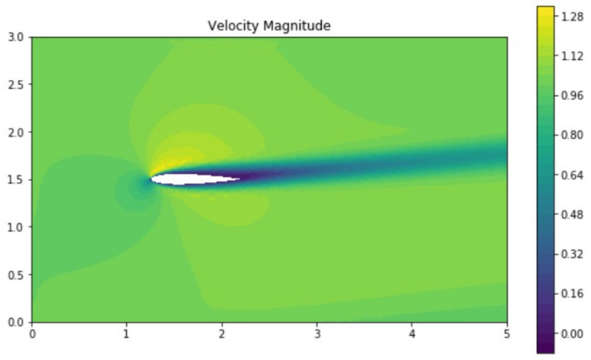


(a) FEniCS mesh

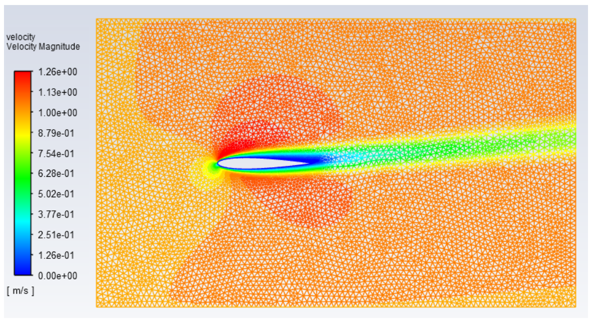


(b) Ansys mesh

Figure 7.1: Mesh comparison for Navier-Stokes simulation on NACA 0012

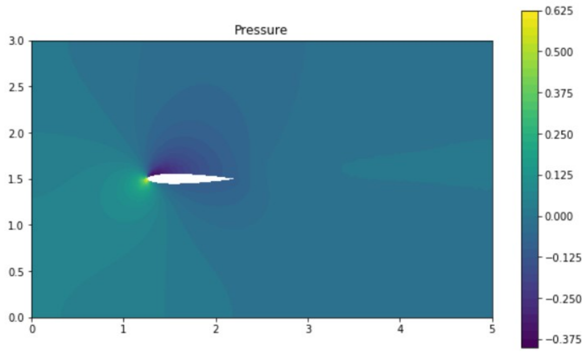


(a) FEniCS velocity contour

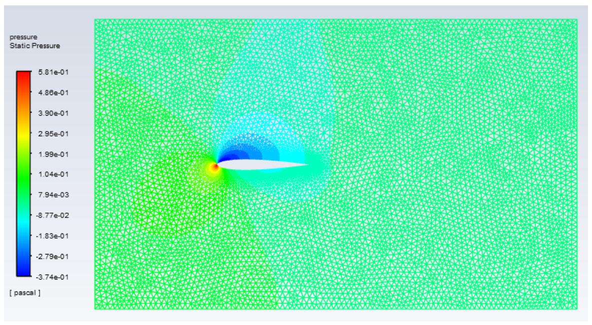


(b) Ansys velocity contour

Figure 7.2: Velocity contours' comparison for Navier-Stokes simulation on NACA 0012



(a) FEniCS pressure contour



(b) Ansys pressure contour

Figure 7.3: Pressure contours' comparison for Navier-Stokes simulation on NACA 0012

Parameter	Value ANSYS	Value FEniCS
Horizontal Force (F_x)	0.07135 N	0.07173 N
Vertical Force (F_y)	0.15376 N	0.15089 N

Table 7.1: Force values comparison between ANSYS Fluent and FEniCS

7.1. Solution for single airfoil geometry

The very first simulation for solving Navier Stokes equations via PINNs was setup for solving a fixed geomerty of airfoil NACA 4812, while deriving all the simulation strategies and parameters from the Stokes flow PINN as shown in chapter 6, including the same neural network architecture as in figure 6.5. The only difference was in terms of the Reynolds number being 10 times higher (Re = 666.67 and kinematic viscosity ν = 1.5e-3), the weight parameter between PDE and data loss being 0.8, in favour of the data loss, to accomodate for better prediction of pressure, and slight change in the PDE loss formulation as depicted in the following equations:

$$e_1 = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial p}{\partial x} - u\frac{\partial u}{\partial x} - v\frac{\partial u}{\partial y}, \tag{7.5}$$

$$e_2 = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial p}{\partial y} - u\frac{\partial v}{\partial x} - v\frac{\partial v}{\partial y}, \tag{7.6}$$

$$e_3 = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}, \tag{7.7}$$

$$\mathcal{L}_{PDE} = e_1^2 + e_2^2 + e_3^2. \tag{7.8}$$

The learning rate was set to 0.0001, and the maximum number of iterations were set to be 1 million. After training, the loss function reached to $\mathcal{O}(1e-5)$. The final results are portrayed in figures 7.4 and 7.5.

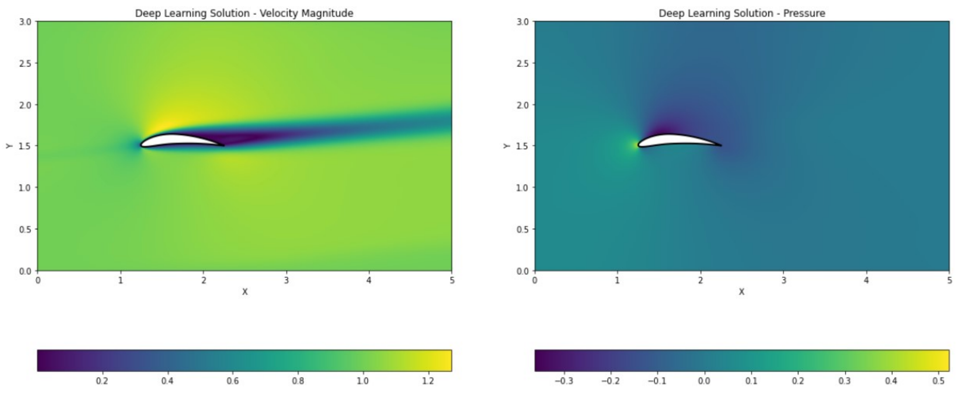


Figure 7.4: Velocity and pressure contours for the solution of Navier Stokes equation via PINN

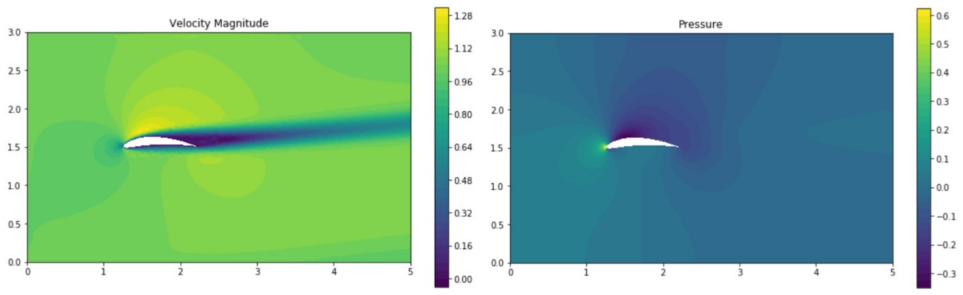


Figure 7.5: FEniCS based results for Navier Stokes equations on a single airfoil domain NACA 4812

It can be observed that the results attained for the velocity contours are almost identical, whereas the results obtained for the pressure contours are much better than seen in the case of Stokes flow. This could be attributed to the higher weight used in the favour of data loss and the global loss attaining a much lower value (by an order of magnitude) at the end of the training process for the Navier Stokes flow compared to Stokes flow.

7.2. Solution for symmetric airfoils of varying thickness

After successfully demonstrating the ability of PINNs to learn the solution to the Navier Stokes equation on a single cambered airfoil geometry, the next task was undertaken to test whether such neural network can learn the solutions for multiple airfoil geometries simultaneously and hence be able to predict the flow field of the Navier Stokes equations over similar unseen geometries. The airfoil geometries used for training this neural network were NACA 0006, 0010, 0014 and 0018. The angle of attack (AoA) was fixed at 5 degrees. The neural network architecture used for this case (and all the subsequent cases) was the one for Stokes equation for varying geometries, as shown in figure 6.8. The maximum number of iterations were now set to 4 million, to allow the neural network to converge to lower loss values. The learning rate was also set to adaptively reduce by a factor of 10 after every million iterations, starting with the value of our default learning rate of 0.0001. A slight modification was made for the loss function to improve the pressure prediction capabilities of the neural network. We introduce two weight parameters, one weight w_{data} for weighting the PDE loss and data loss, and the other weight w_{pressure} for weighting the respective root mean square loss between velocity data $\mathcal{L}_{\text{data}}(\text{Velocity})$ and the corresponding root mean square loss for pressure data $\mathcal{L}_{\text{data}}(\text{Pressure})$ after bifurcating the velocity and pressure data separately. The values of both the weights w_{data} and w_{pressure} were set to be 0.8 each in the favour of data loss and pressure data loss respectively.

$$\mathcal{L}_{\text{overall}} = (1 - w_{\text{data}}) * \mathcal{L}_{\text{PDE}} + w_{\text{data}}((1 - w_{\text{pressure}}) * \mathcal{L}_{\text{data}}(\text{Velocity}) + w_{\text{pressure}} * \mathcal{L}_{\text{data}}(\text{Pressure})).$$

After training, the final loss values were still of the order $\mathcal{O}(1e-5)$ but they were much lower compared to the single airfoil geometry case with non-adaptive learning rate and 1 million iterations. The results for various seen and unseen geometries predicted from the trained PINN have been illustrated in figures 7.6 and 7.7.

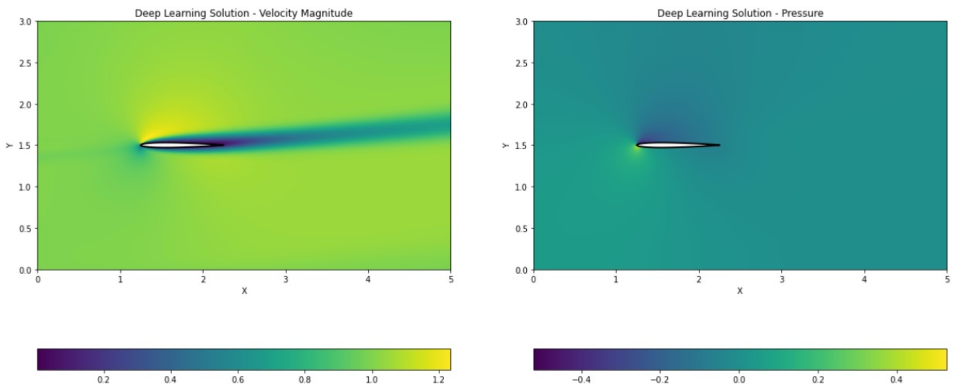


Figure 7.6: Navier Stokes solution via PINN on a seen thin airfoil NACA 0006

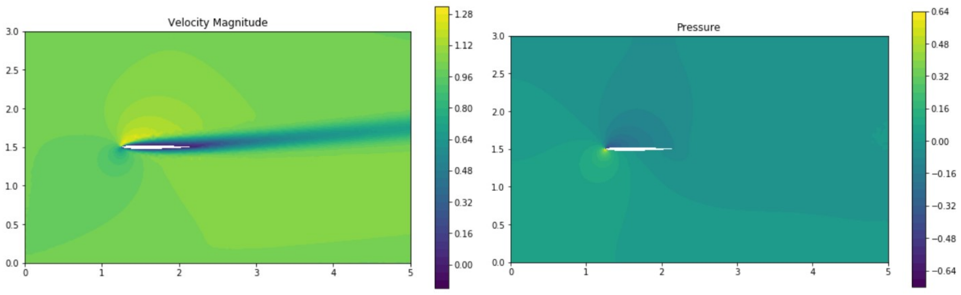


Figure 7.7: FEniCS based results for Navier Stokes equations on NACA 0006

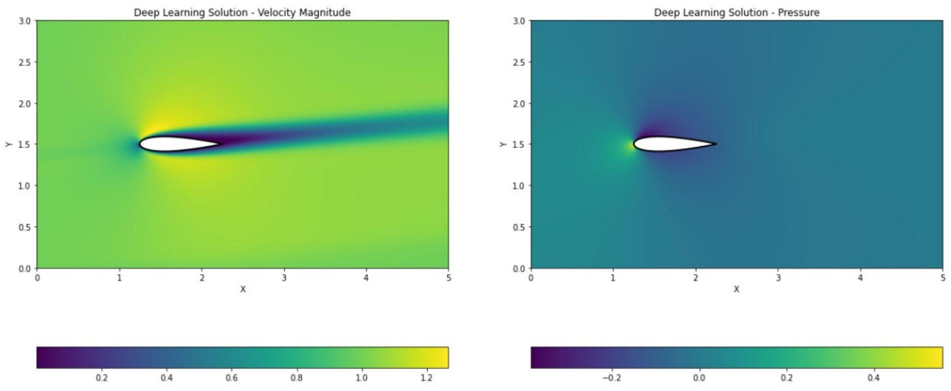


Figure 7.8: Navier Stokes equation via PINN on a seen thick airfoil NACA 0018

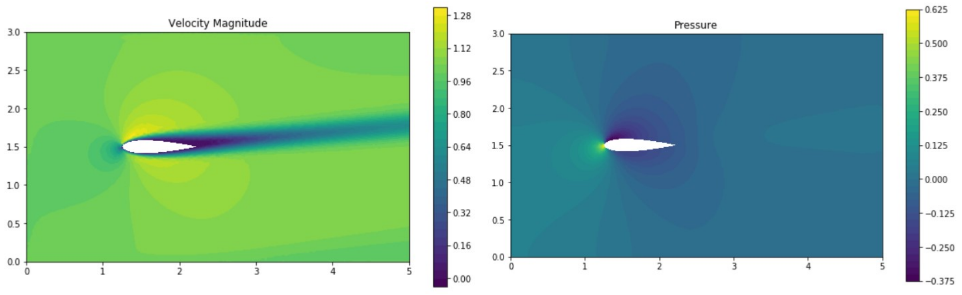


Figure 7.9: FEniCS based results for Navier Stokes equations on NACA 0018

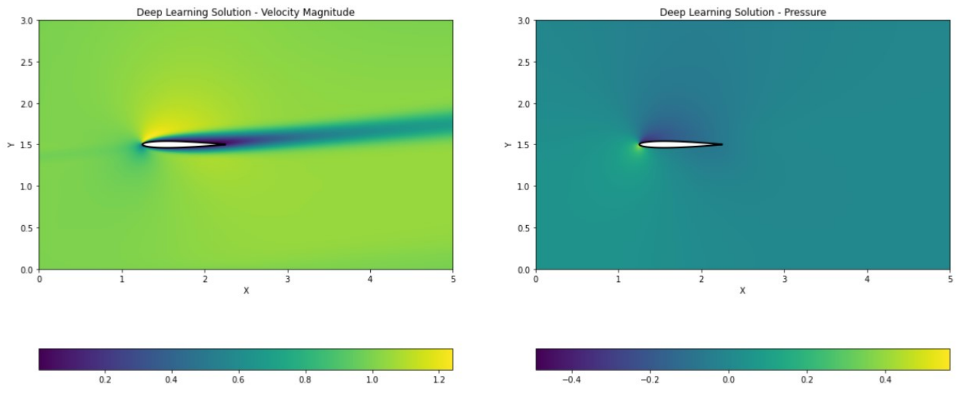


Figure 7.10: Navier Stokes solution via PINN on an unseen thin airfoil NACA 0008

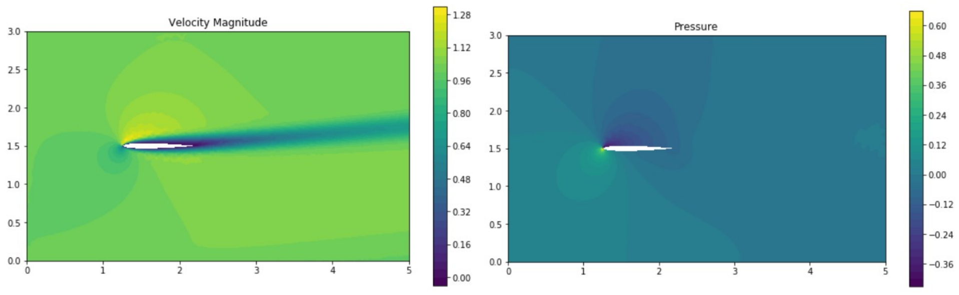


Figure 7.11: FEniCS based results for Navier Stokes equations on NACA 0008

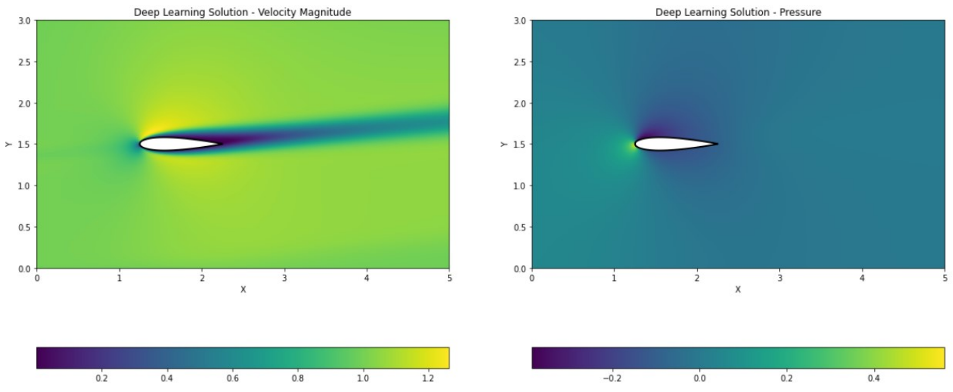


Figure 7.12: Navier Stokes equation via PINN on an unseen thick airfoil NACA 0016

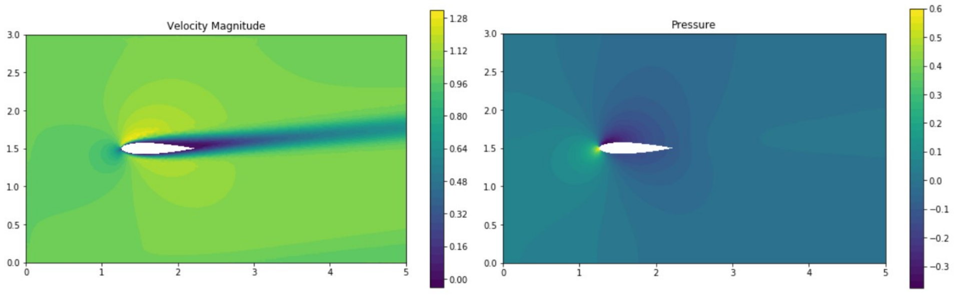


Figure 7.13: FEniCS based results for Navier Stokes equations on NACA 0016

It can be observed from the above results that the neural network has been successful in learning the Navier Stokes solutions for multiple symmetric airfoil geometries and is also capable of interpolating the results for unseen geometries. The pressure predictions for this case do not seem to be significantly more accurate than the previous case without data bifurcation, but this approach has been adopted as standard for future simulations nonetheless. Moreover, it can be observed that the pressure predictions become more accurate for thicker airfoils compared to thinner airfoils.

7.3. Solutions for further cases of parameterization

After attaining successful results for simulating a single cambered and parameterized symmetric airfoils on a single angle of attack with Navier Stokes equations using PINNs, we can now move forward to explore more aspects of parameterization which are possible. We explore three such cases for now:

1. Variation of angle of attack for a single airfoil (NACA 4812)
2. Variation of camber magntitude and camber positions (first two NACA digits) for an airfoil of same thickness (12%)
3. Variation of angle of attack and thickness of symmetric airfoils simultaneously, along with the study of validation loss.

The above mentioned studies were performed in order to understand the underlying requirements and challenges for training a PINN with different forms and levels of parameterization, and the outcomes of these studies are used to design the final deep learning pipeline capable of predicting the Navier Stokes solution for any shape of a NACA airfoil (symmetric as well as cambered) determined by an autoencoder, at any angle of attack before separation. It should be noted that the simulation parameters, boundary conditions, mesh resolution and training strategies are the same for the upcoming simulation studies as explained above for the case of symmetric airfoils with parameterized thickness.

7.3.1. Variation of Angle of Attack

An important parameterization for flow around an airfoil comes from varying the angle of attack of the airfoil. The analysis was carried on NACA 4812 airfoil for the angles of attack being 0, 4 and 8 degrees respectively. Training was performed for 4 million iterations, reaching the loss value of the order $\mathcal{O}(1e-5)$. The results obtained are given in figures 7.14, 7.15, 7.16 and 7.17.

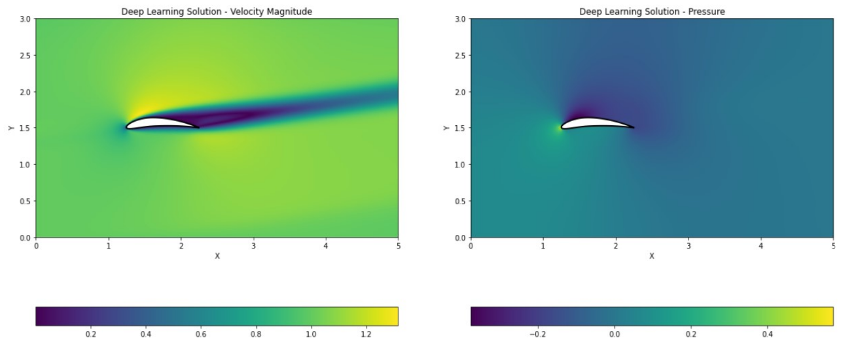


Figure 7.14: Navier Stokes equation via PINN on an angle of attack 8 degrees (as seen during the training) for NACA 4812

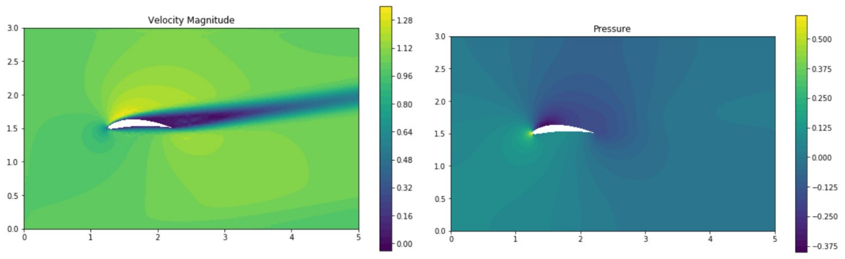


Figure 7.15: FEniCS based results for Navier Stokes equations on NACA 4812 angle of attack 8 degrees

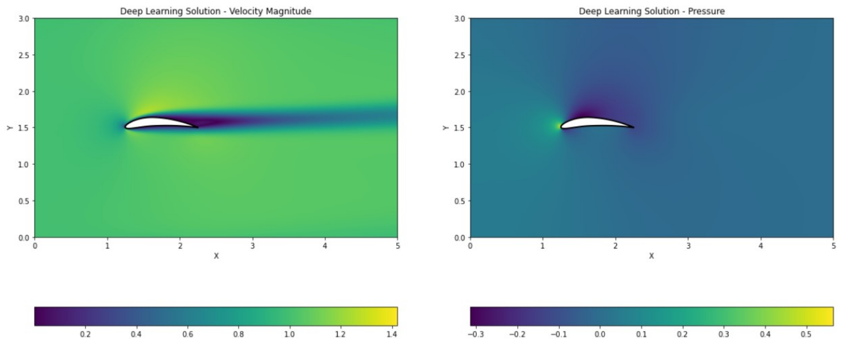


Figure 7.16: Navier Stokes equation via PINN on an unseen angle of attack 3 degrees for NACA 4812

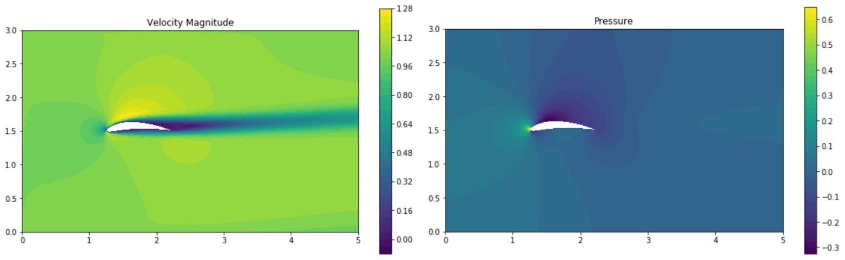


Figure 7.17: FEniCS based results for Navier Stokes equations on NACA 4812 angle of attack 3 degrees

It can be seen from the results in the figures 7.16 and 7.17 that the velocity predictions, in terms of the maximum velocity attained, is quite inaccurate for the case of 3 degrees angle of attack. On the other hand, the pressure contours for both the cases are quite well predicted. This might be a short coming of having a higher weightage for pressure loss in the bifurcated pressure and velocity data. Therefore, to ensure accurate flow field predictions for all possible angles of attack by the PINN, the number of training iterations should be increased further, which has been done in the final case study given in chapter 8.

7.3.2. Variation of camber parameters

After analysing the geometric variation of symmetric airfoils, we are now in a position to extend our study to cambered airfoils, where we can vary the first two digits of a NACA airfoil, namely the camber magnitude and the maximum camber position. The first case was setup in a very simple manner, where we trained the neural network for NACA airfoils 0012, 2412 and 4812 on 5 degrees angle of attack, to see how well it could capture the camber information from these limited number of cases. The training was performed for 4 million iterations. The neural network completely failed to provide results for unseen airfoils using this set of training data.

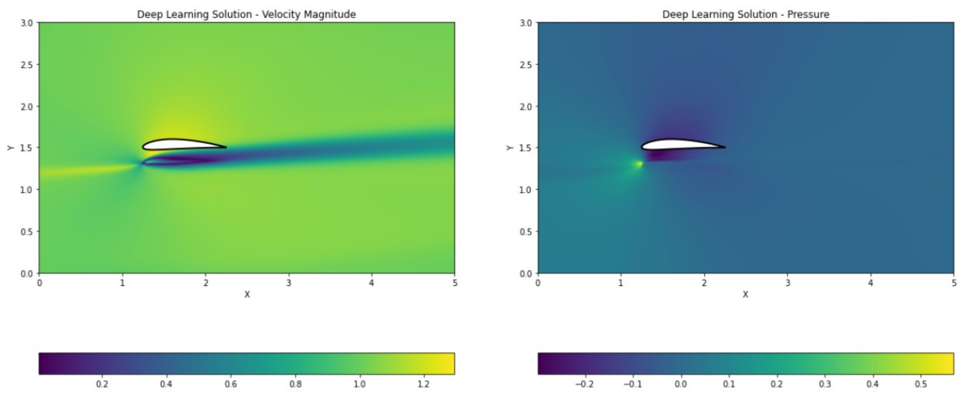


Figure 7.18: Navier Stokes equation via PINN on an unseen cambered airfoil NACA 4412

Therefore, a better training strategy needed to be evolved for representing cambered airfoils via PINNs. It was decided to use 6 different geometries of NACA airfoils, namely NACA 0012, 2412, 4412, 6412, 4812, 4212 and 4612, thereby ensuring multiple camber magnitudes for a given camber position and vice versa. This approach worked very well for training the PINN on unseen cambered airfoils, at a fixed angle of attack 5 degrees, the results for which have been visualised in figures 7.19 and 7.20.

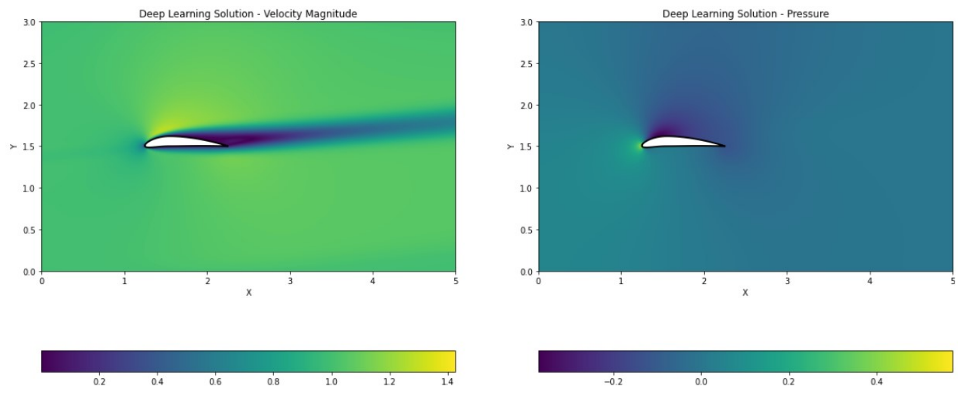


Figure 7.19: Navier Stokes equation via PINN on an unseen cambered geometry NACA 3612

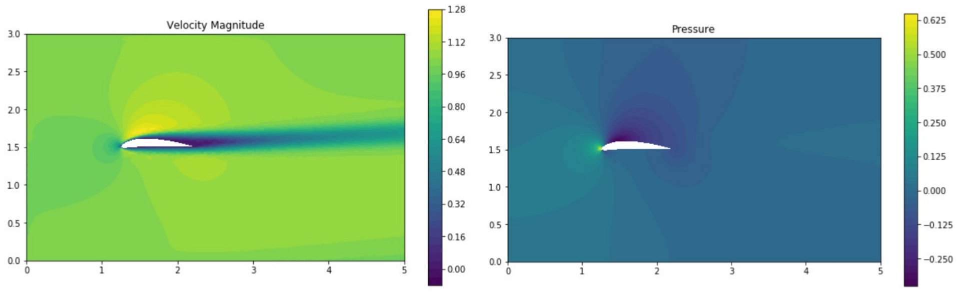


Figure 7.20: FEniCS based results for Navier Stokes equations on NACA 3612 angle of attack 5 degrees

It can be observed in this case that the results are not highly accurate but that can be solved by increasing the number of iterations, as will be demonstrated in chapter 8. The main takeaway from this experiment is that PINNs can be trained for cambered airfoils by providing sufficient number of training geometries, especially by having multiple camber values for a given camber position and vice versa.

7.3.3. Variation of angle of attack and thickness of symmetric airfoils

All the cases we explored till now involved the parameterization of either one component of the geometry (thickness or camber) or varying the angles of attack. In the present case, we tried to build a highly robust PINN framework that could be trained on variable geometry (thickness for symmetric airfoil) and the angle of attack. This study was supposed to act as a precursor towards our final goal of simulating cambered airfoils of complex shapes with varying angles of attack after coupling with the auto-encoder. All the learnings from the previous experiments have been used for setting up this formulation. Moreover, we introduced another parameter for studying the effectiveness of the training process, called the validation loss. The motivation behind studying the validation loss was to study the impact of physics-informed loss formulation in preventing the overfitting of the PINN. The phenomena of overfitting and underfitting were explained in chapter 2 under the section 2.1.2. Seeing the erroneous velocity profile results in some previous simulations as a result of weighted data loss bifurcation in favour of pressure data, the number of training iterations were raised from 4 million to 5 million. Moreover, in this case the training dataset is much larger compared to previous cases due to more levels of parameterization, thereby justifying the use of more iterations. The training data consisted of airfoils NACA 0010, 0014, 0018 and 0022, each with angles of attack ranging 0, 4, 8 and 12 degrees. The testing/validation dataset consisted of airfoils not seen during the training, namely NACA 0012, 0016 and 0020 at the angle of attack 5 degrees. The comparative plot of loss function variation and validation loss variation with iterations is illustrated in figure 7.21. The final value of loss function after training was attained to be of the order $\mathcal{O}(1e-5)$.

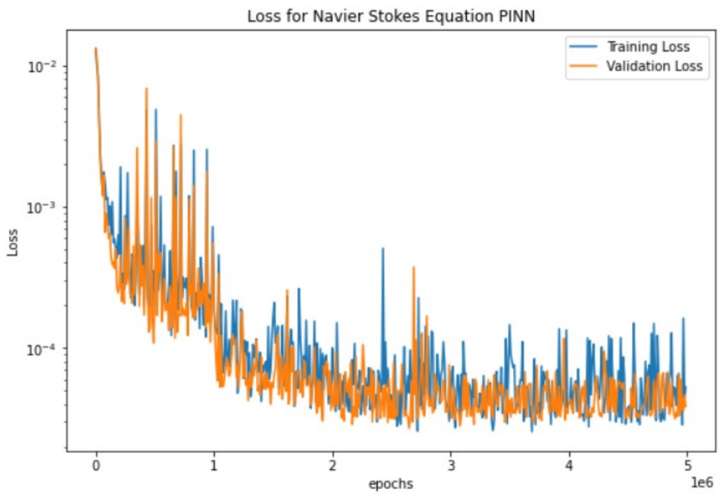


Figure 7.21: Loss variation with iterations for parameterized Navier Stokes

It can be inferred from figure 7.21 that the validation loss follows almost a similar variation as the training loss, thereby suggesting no overfitting during the training process. This could be attributed to the fact that as the training loss approaches zero, the PINN not only learns to fit the desired training data, but also the corresponding governing equations that have been used to generate the training as well as validation data. The simulation results for this case over an unseen airfoil geometry and angle of attack are illustrated in figures 7.22, 7.23, 7.24 and 7.25.

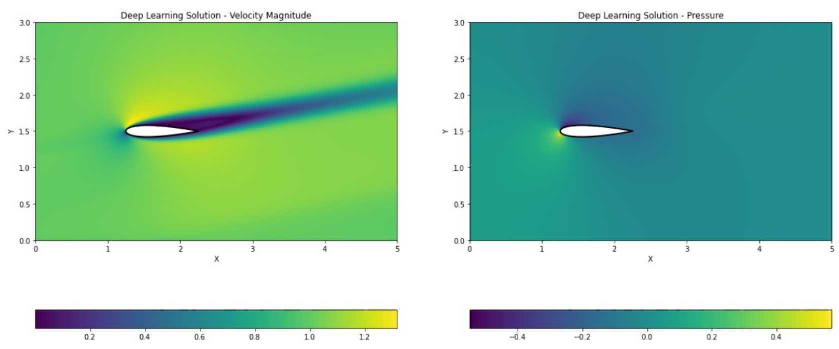


Figure 7.22: Navier Stokes equation via PINN on an unseen geometry NACA 0016 at 7 deg angle of attack

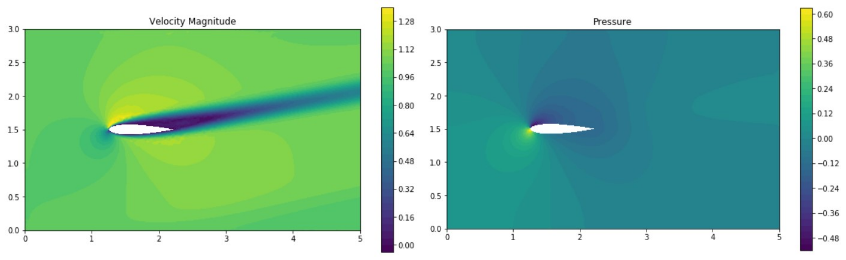


Figure 7.23: FEniCS based results for Navier Stokes equations on NACA 0016 at 7 deg angle of attack

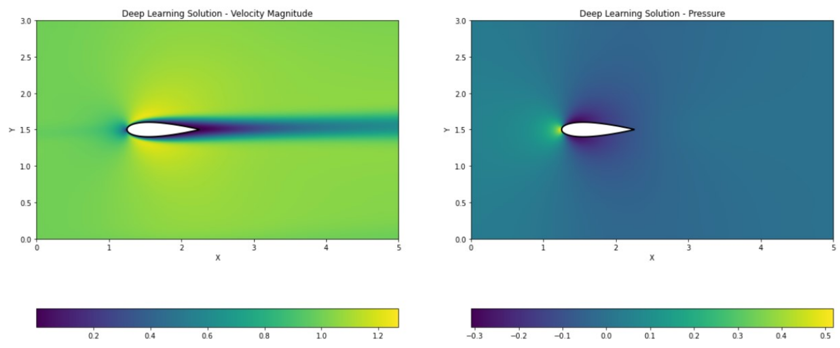


Figure 7.24: Navier Stokes equation via PINN on an unseen geometry NACA 0020 at 1 deg angle of attack

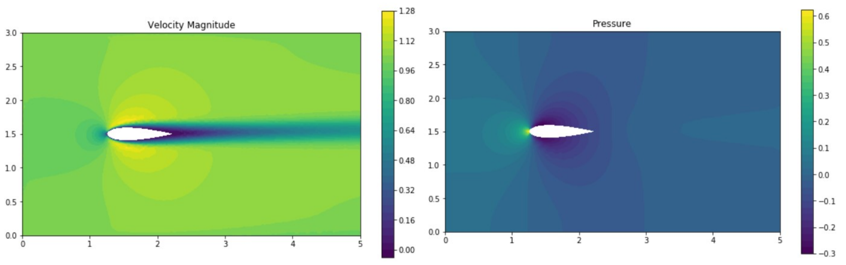


Figure 7.25: FEniCS based results for Navier Stokes equations on NACA 0020 at 1 deg angle of attack

It can be observed from these results that pressure prediction still continues to be inaccurate, whereas the velocity predictions seem to be quite accurate. Nonetheless, the PINN has successfully demonstrated its capability in terms of parameterized learning of PDEs, and this methodology would also be used for our final case on cambered airfoils by using PINNs coupled with auto-encoders.

8
Coupling of parameterized PINN and auto-encoder for the Navier Stokes Equation

In the previous chapters 5, 6 and 7, we used well defined parameters to describe our geometries of interest due to their simplicity, for example the dimensions of a rectangular domain (chapter 5) or the NACA airfoil parameters (chapters 6 and 7). However, in many use cases of practical interests, geometries cannot be parameterized via classical polynomials. Deep learning architectures called auto-encoders, discussed in chapter 4 on generative modelling can be used for compressing the geometric representations to a limited number of parameters. These parameters are stored in the latent space vector of the auto-encoder. The first part of the auto-encoder, called the encoder, bridges the actual geometry to the latent space vector, whereas the second part called the decoder, can be used to interpret the actual geometry defined by the latent space parameters. In the present case, we use this auto-encoder framework, developed by Monolith AI pvt. ltd., to parameterize the set of cambered NACA airfoils into three latent space parameters (as we already know that NACA airfoils require three defining parameters) to demonstrate the coupling capability of PINNs and auto-encoders for simulating complex geometry beyond classical parameterization techniques. The latent space parameters are called z1, z2 and z3 and they are fed as inputs to the PINN framework, as opposed to the camber, camber position and airfoil thickness while using the standard NACA parameterization. The neural network architecture used for the PINN is shown in figure 8.1.

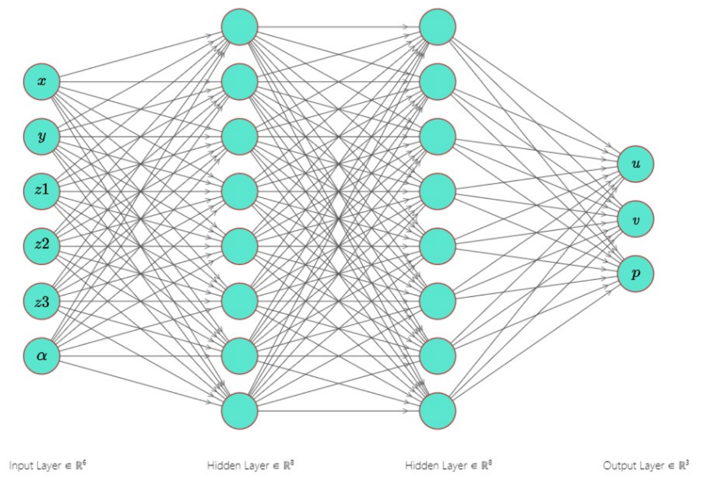
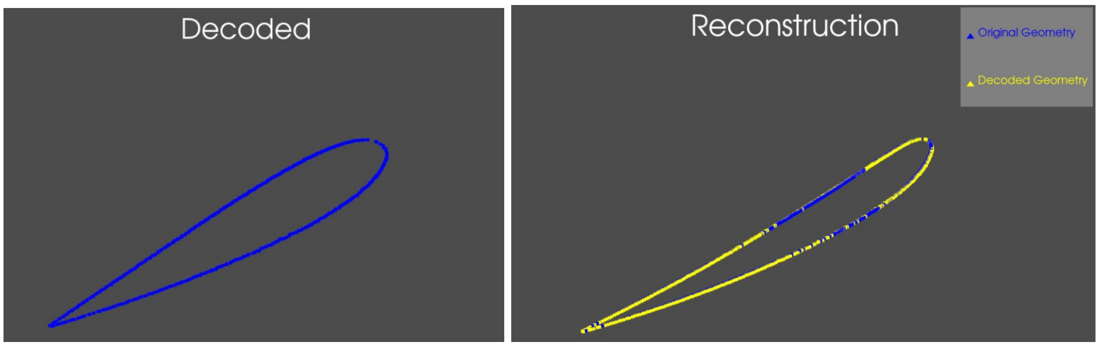

x
y
$z1$
$z2$
$z3$
α
u
v
p
Input Layer $\in \mathbb{R}^6$
Hidden Layer $\in \mathbb{R}^8$
Hidden Layer $\in \mathbb{R}^8$
Output Layer $\in \mathbb{R}^3$

Figure 8.1: Neural network architecture (PINN part) for the auto-encoder based parameterized airfoils.This is a representative neural network to show the mapping from inputs to outputs, the actual neural network contained 4 hidden layers with 100 neurons per layer.

The auto-encoder was trained on 500 randomly generated NACA airfoil geometries for latent space spanned by 3 parameters, namely z1, z2 and z3. The model converged upto a loss value of the order $\mathcal{O}(1e-6)$. After training, the encoder can be used to find the latent space parameterization of a desired NACA airfoil, whereas the decoder can be used to formulate the corresponding airfoil geometry of an arbitrary selection of latent space parameters. It must be noted that parameterization of NACA airfoils with the help of auto-encoders is done only to demonstrate the proof of concept for coupling PINNs and auto-encoders for complex geometries, otherwise NACA airfoils are self parameterized.



(a) Decoding of airfoil for [z1,z2,z3] = [0,0,0] **(b)** Reconstruction (encoding + decoding) of NACA 4612

Figure 8.2: Results for reconstruction and simple decoding of airfoils from the trained auto-encoder

For training the PINN framework on auto-encoder based parameterization of cambered airfoils, using our standard training data for conventional NACA airfoils, we need to generate a database which contains the latent space parameterization of our well defined NACA airfoils. It must be noted that during training the PINN, it is extremely important for the PINN to learn the latent space parameterization of the NACA airfoils, that is the decoder part of the corresponding auto-encoder, in order to be able to interpret the desired geometry correctly and solve the corresponding Navier Stokes equations around it. In the present case, the encoder was used to generate a database of z parameters corresponding to the NACA airfoils to be used for training and the corresponding data structure was exporting to the PINN training code, as opposed to actually coupling the PINN and the auto-encoder.

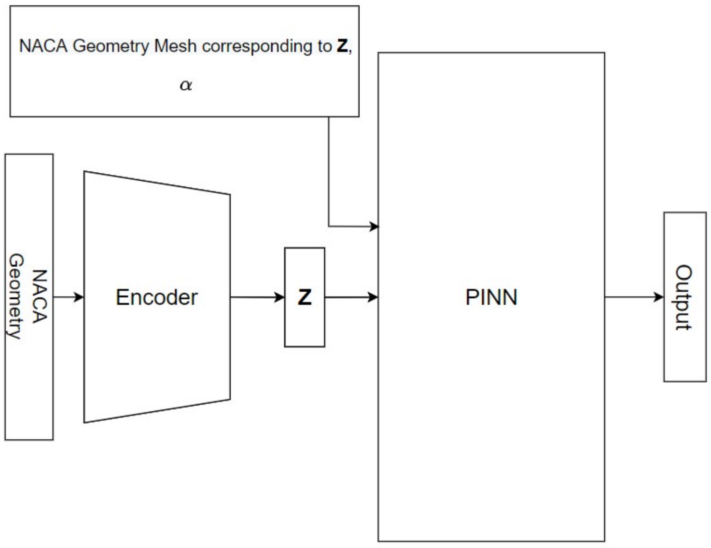


Figure 8.3: Training pipeline for auto-encoder parameterized airfoils

After training, we can either couple the whole auto-encoder (encoder + decoder) with the trained PINN to test its performance over a predefined NACA airfoil, or we can use only the decoder in case we want to generate solution data from PINNs for unconventional latent space parameterized airfoils. The results for this trained PINN for our standard NACA 4812 airfoil are illustrated in figure 8.4.

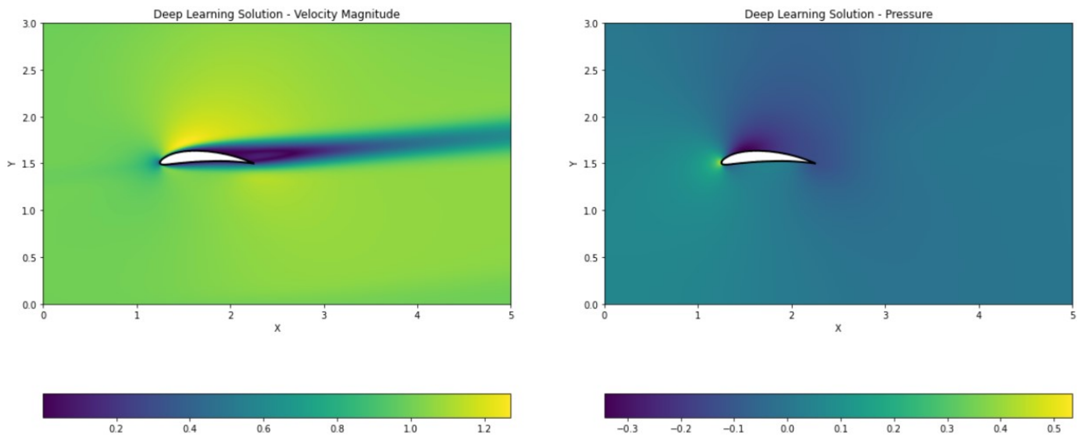


Figure 8.4: Velocity and Pressure contours for auto-encoder parameterized PINN on standard NACA 4812 airfoil at 5 deg angle of attack.

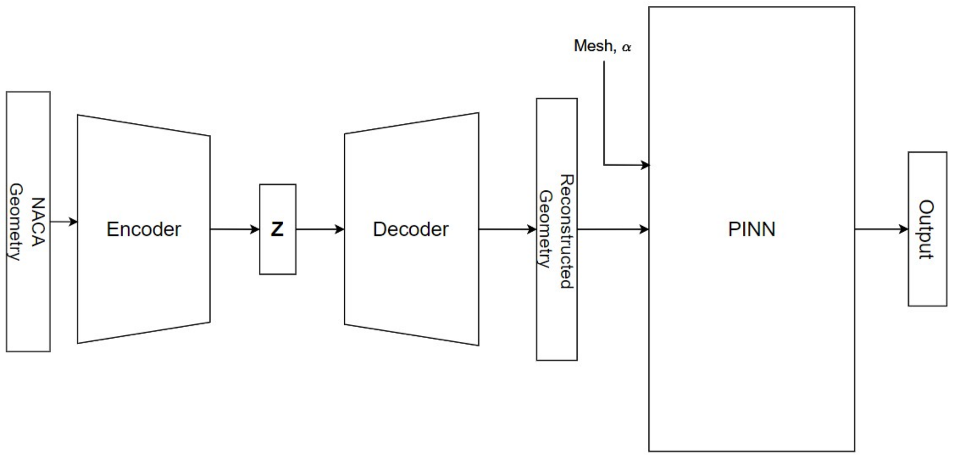


Figure 8.5: Architecture for evaluating results on NACA airfoil with latent space parameterized PINN

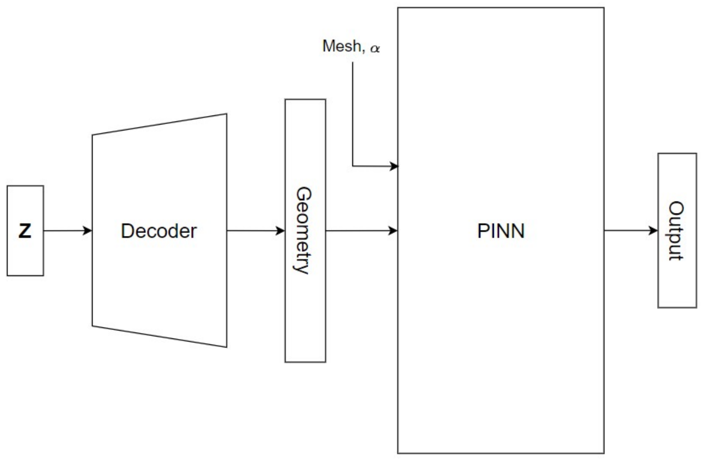
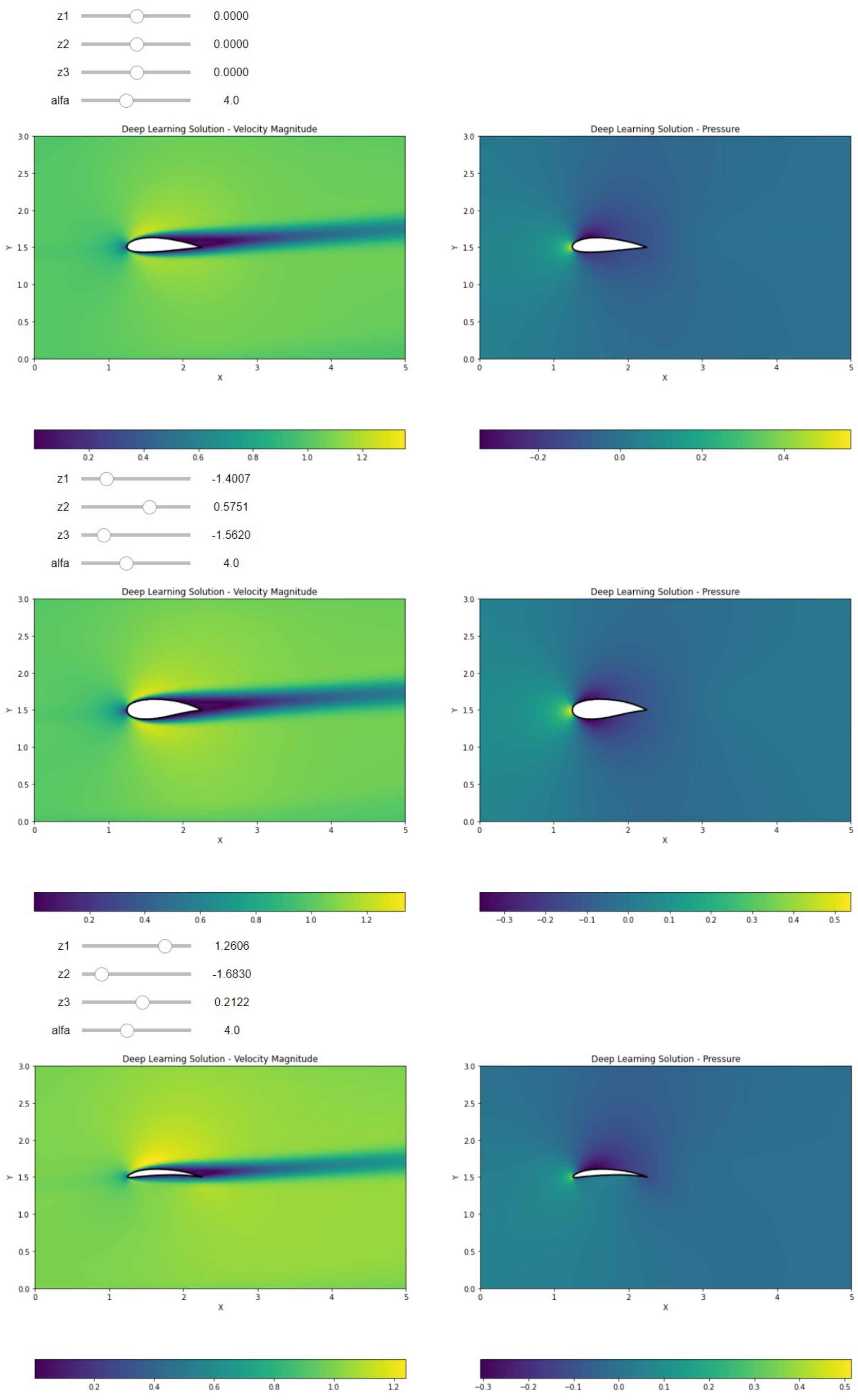


Figure 8.6: Architecture for evaluating results on latent space parameterized PINN

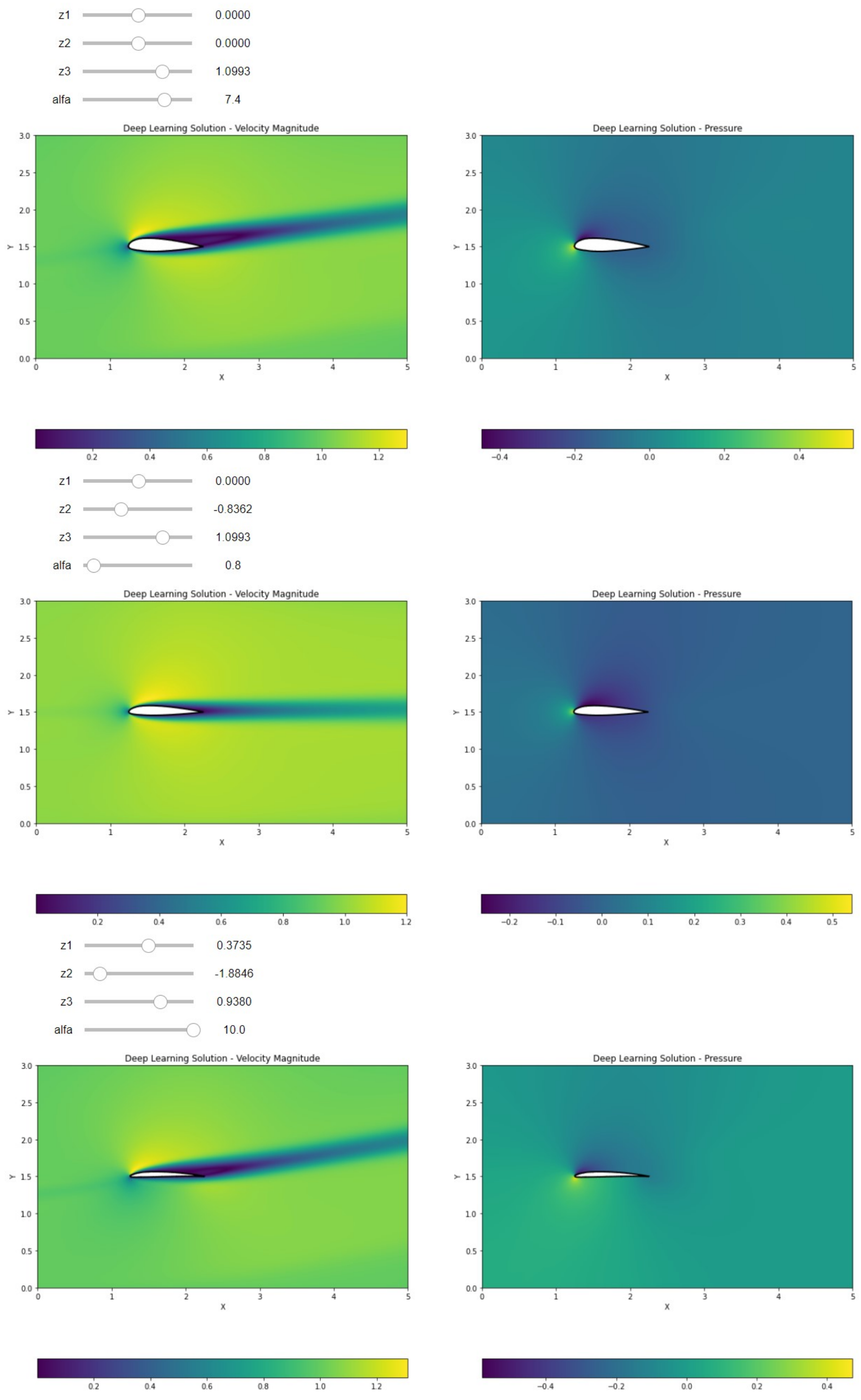
The final results for auto-encoder based parameterized cambered airfoil geometries for PINNs trained for Navier Stokes equations are depicted in figures 8.7 and 8.8.


z1 0.0000
z2 0.0000
z3 0.0000
alfa 4.0

z1 -1.4007
z2 0.5751
z3 -1.5620
alfa 4.0

z1 1.2606
z2 -1.6830
z3 0.2122
alfa 4.0
Deep Learning Solution - Velocity Magnitude
Deep Learning Solution - Pressure
X
Y

Figure 8.7: Results from coupling of PINN and decoder according to architecture in figure 8.6


z1 0.0000
z2 0.0000
z3 1.0993
alfa 7.4

z1 0.0000
z2 -0.8362
z3 1.0993
alfa 0.8

z1 0.3735
z2 -1.8846
z3 0.9380
alfa 10.0

Figure 8.8: Results from coupling of PINN and decoder according to architecture in figure 8.6

Lift and Drag Computation
The lift and drag computations over the airfoil results can be performed by making use of equations (2.23) and (2.24) as mentioned in chapter 2. It must be noted that the angle of attack variation was neglected for ease of computation and in this case the drag and lift simply mean the horizontal and vertical forces on the airfoil respectively, as we only needed a quantitative measure to compare the

force computation which could be compared even by neglecting the orientation of the components of the resultant overall force. It was found that although the trained PINN is able to provide reasonably accurate velocity and pressure contours for a wide range of airfoil geometries and angles of attack, it fails completely while computing the corresponding forces imparted by the fluid over the airfoils. This is due to the fact that unlike numerical methods, in deep learning formulations, the boundary conditions are supposed to be ’learnt’ by the neural network and cannot be prescribed exactly. For example over the airfoil surface, the no slip boundary condition would dictate zero velocity, but the velocity values learnt by the PINN are close to zero, for example 0.05 m/s or 0.001 m/s etc. This could explain why even though the contours provided by the PINN are reasonably accurate (as in a contour colour scheme there would not be much difference between 0 m/s and 0.001 m/s) but the force computations are completely unreliable. These observations suggest that the developed PINN-auto-encoder framework is useful for the quick qualitative analysis of geometries under fluid flow, but not (yet) mature enough for a quantitative analysis including force computations for lift and drag.

9 Conclusions and Recommendations

In the preceding chapters, an extensive study has been provided on the relatively new research direction of embedding first-principle physics into neural networks and coupling them with autoencoders to achieve the objectives of computational fluid dynamics on parameterized variable geometries while removing the redundancies associated with classical numerical analysis. It has been concluded that deep learning methods, at least with the formulation studied in this thesis, are probably only suitable to study the target variable contours (which also has its own significance in industrial, especially aerodynamic applications) but not reliable enough for force (drag and lift) computations. The main findings of this thesis are as follows:

- If the input mesh (boundary plus collocation points) is regenerated randomly after every few training iterations, the PINN converges easily as it gets to witness a large variety of sample points during the whole training process. This is probably the biggest advantage of neural networks that instead of defining a very fine mesh to enhance accuracy throughout all the iterations (as in numerical analysis), we can achieve the same objectives by providing the dense mesh points in batches, hence making the PINN more robust as well as computationally efficient. Moreover, we can train the PINN on a coarse mesh and evaluate the results on a fine mesh, which is referred to as 'super-resolution'.
- Pressure prediction is quite hard with the adopted formulation of the fluid flow PINNs (Stokes as well as Navier Stokes) even after heavy biasing in the loss function towards pressure data. Although no quantitatively conclusive remarks can be made in this regard but it might be worthwhile to experiment with different governing equation formulations or neural network architectures, for example a dedicate neural network for each variable such as velocity components $u,\ v$ and pressure p, respectively.
- Excessive training to achieve lower loss values does not lead to overfitting on training data due to the presence of the PDE residual in the loss function. It has been observed that by accommodating information regarding physics (governing differential equations), the loss corresponding to training data as well as that for validation data maintain the same order of magnitude throughout the training process. This might be explained from the fact that via PINN formulation, the neural network is not only learning the solution mapping for the given set of data points but also the desired instances of PDEs for both the training data as well as validation data set. The loss variation patterns that signify this fact have been illustrated in figure 9.1. For the case of physics informed learning, both the training and validation losses follow the same order of magnitude with the minimal values being attained of the $\mathcal{O}(1e-5)$. Whereas, for the case with purely data based training, the validation loss is an order of magnitude higher $\mathcal{O}(1e-5)$ than the training loss $\mathcal{O}(1e-6)$.
- Parameterization suffers from the curse of dimensionality, the required training data increases exponentially with the number of parameters, as the network learns to parameterize the system when provided with sufficient data for variation of each parameter in isolation while keeping all the others constant.
- The neural network in PINN should be capable of learning the decoder part of the auto-encoder for the successful coupling of PINNs and autoencoders. This might require deviating from simple

feed forward architectures for the PINN for dealing with parameterization of complex 3D geometries with the autoencoder, which are much more advanced than cambered NACA airfoils.

- Drag and lift or other force computations are not feasible on PINN results in our simulations due to their lack of accuracy in predicting the boundary data.

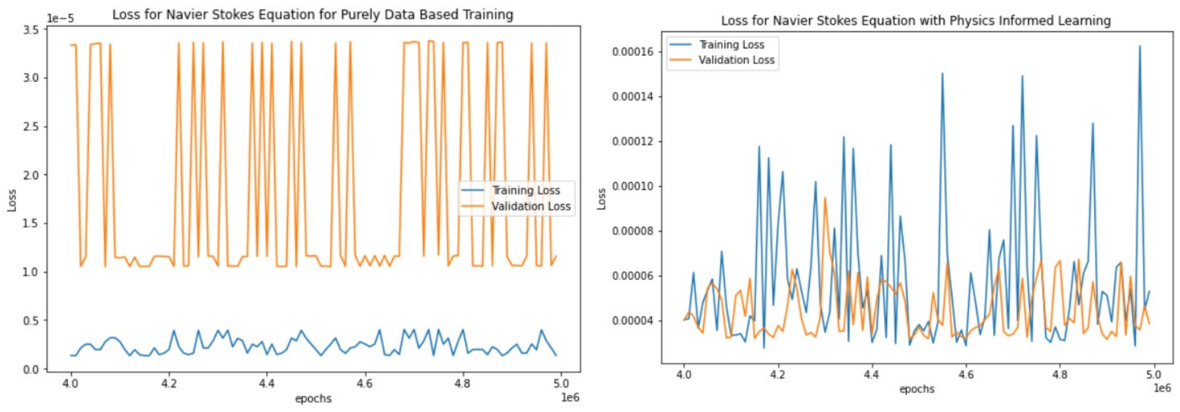


Figure 9.1: Comparison of training and validation data losses for the final 1 million training iterations

References

[1] Alexander Amini and Ava Soleimany. *MIT Deep Learning 6.S191, 2020*. URL: `introtodeeplearning.com`.

[2] Bengt Andersson et al. *Computational Fluid Dynamics for Engineers*. Cambridge University Press, 2011. DOI: `10.1017/CBO9781139093590`.

[3] Jack Y Araz, Juan Carlos Criado, and Michael Spannowsky. "Elvet–a neural network-based differential equation and variational problem solver". In: *arXiv preprint arXiv:2103.14575* (2021).

[4] Giovanni Calzolari and Wei Liu. "Deep learning to replace, improve, or aid CFD analysis in built environment applications: A review". In: *Building and Environment* 206 (Dec. 2021), p. 108315. ISSN: 0360-1323. DOI: `10.1016/J.BUILDENV.2021.108315`.

[5] Feiyu Chen et al. "NeuroDiffEq: A Python package for solving differential equations with neural networks". In: *Journal of Open Source Software* 5.46 (2020), p. 1931.

[6] T. Chen and H. Chen. "Approximations of continuous functionals by neural networks with application to dynamic systems". In: *IEEE Transactions on Neural Networks* 4.6 (1993), pp. 910–918. DOI: `10.1109/72.286886`.

[7] Wei Chen, Kevin Chiu, and Mark Fuge. "Aerodynamic design optimization and shape exploration using generative adversarial networks". In: *AIAA Scitech 2019 Forum*. 2019, p. 2351.

[8] J Austin Cottrell, Thomas JR Hughes, and Yuri Bazilevs. *Isogeometric analysis: toward integration of CAD and FEA*. John Wiley & Sons, 2009.

[9] Miles Cranmer et al. "Discovering symbolic models from deep learning with inductive biases". In: *arXiv preprint arXiv:2006.11287* (2020).

[10] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[11] Asger Bolet Gaute Linga. *PROJECT IN CONTINUUM MECHANICS: Simulating Fluid Flow in Complex Geometries using FEniCS*. 2016.

[12] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014).

[13] Ehsan Haghighat and Ruben Juanes. "Sciann: A keras/tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks". In: *Computer Methods in Applied Mechanics and Engineering* 373 (2021), p. 113552.

[14] Oliver Hennigh et al. "NVIDIA SimNet™: An AI-Accelerated Multi-Physics Simulation Framework". In: *International Conference on Computational Science*. Springer. 2021, pp. 447–461.

[15] Ameya D Jagtap, Kenji Kawaguchi, and George Em Karniadakis. "Locally adaptive activation functions with slope recovery for deep and physics-informed neural networks". In: *Proceedings of the Royal Society A* 476.2239 (2020), p. 20200334.

[16] Ameya D Jagtap, Kenji Kawaguchi, and George Em Karniadakis. "Adaptive activation functions accelerate convergence in deep and physics-informed neural networks". In: *Journal of Computational Physics* 404 (2020), p. 109136.

[17] Ameya D Jagtap, Ehsan Kharazmi, and George Em Karniadakis. "Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems". In: *Computer Methods in Applied Mechanics and Engineering* 365 (2020), p. 113028.

[18] Xiaowei Jin et al. "NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations". In: *Journal of Computational Physics* 426 (2021), p. 109951.

[19] George Em Karniadakis et al. “Physics-informed machine learning”. In: *Nature Reviews Physics* 3.6 (2021), pp. 422–440.

[20] Nikola Kovachki et al. “Neural operator: Learning maps between function spaces”. In: *arXiv preprint arXiv:2108.08481* (2021).

[21] Zongyi Li et al. “Fourier neural operator for parametric partial differential equations”. In: *arXiv preprint arXiv:2010.08895* (2020).

[22] Zongyi Li et al. “Physics-informed neural operator for learning partial differential equations”. In: *arXiv preprint arXiv:2111.03794* (2021).

[23] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

[24] Lu Lu, Pengzhan Jin, and George Em Karniadakis. “Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators”. In: *arXiv preprint arXiv:1910.03193* (2019).

[25] Lu Lu et al. “DeepXDE: A deep learning library for solving differential equations”. In: *SIAM Review* 63.1 (2021), pp. 208–228.

[26] Lu Lu et al. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature Machine Intelligence* 3.3 (2021), pp. 218–229.

[27] Zhiping Mao, Ameya D Jagtap, and George Em Karniadakis. “Physics-informed neural networks for high-speed flows”. In: *Computer Methods in Applied Mechanics and Engineering* 360 (2020), p. 112789.

[28] Zhiping Mao et al. “DeepM&Mnet for hypersonics: Predicting the coupled flow and finite-rate chemistry behind a normal shock using neural-network approximation of operators”. In: *arXiv preprint arXiv:2011.03349* (2020).

[29] Stefano Markidis. “The old and the new: Can physics-informed deep-learning replace traditional linear solvers?” In: *Frontiers in big Data* (2021), p. 92.

[30] Dr. Miguel Mendez. *Hands on Machine Learning for Fluid Dynamics*. 2022.

[31] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN: 9780387400655. URL: https://books.google.nl/books?id=VbHYoSyelFcC.

[32] Jeong Joon Park et al. “Deepsdf: Learning continuous signed distance functions for shape representation”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 165–174.

[33] Maziar Raissi. “Deep hidden physics models: Deep learning of nonlinear partial differential equations”. In: *The Journal of Machine Learning Research* 19.1 (2018), pp. 932–955.

[34] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[35] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. “Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for assimilating flow visualization data”. In: *arXiv preprint arXiv:1808.04327* (2018).

[36] Maziar Raissi et al. “Deep learning of vortex-induced vibrations”. In: *Journal of Fluid Mechanics* 861 (2019), pp. 119–137.

[37] *SimNet, A Neural Network Based Partial Differential Equation Solver, User Guide, Releasev21.06, June 2021*.

[38] Sifan Wang, Yujun Teng, and Paris Perdikaris. “Understanding and mitigating gradient flow pathologies in physics-informed neural networks”. In: *SIAM Journal on Scientific Computing* 43.5 (2021), A3055–A3081.

[39] Sifan Wang, Hanwen Wang, and Paris Perdikaris. “Learning the solution operator of parametric partial differential equations with physics-informed DeepOnets”. In: *arXiv preprint arXiv:2103.10974* (2021).

[40] Sifan Wang, Xinling Yu, and Paris Perdikaris. “When and why pinns fail to train: A neural tangent kernel perspective”. In: *arXiv preprint arXiv:2007.14527* (2020).

[41] Liu Yang, Xuhui Meng, and George Em Karniadakis. “B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data”. In: *Journal of Computational Physics* 425 (2021), p. 109913.

[42] Kazuo Yonekura and Katsuyuki Suzuki. “Data-driven design exploration method using conditional variational autoencoder for airfoil design”. In: *Structural and Multidisciplinary Optimization* 64.2 (2021), pp. 613–624.

[43] Jeremy Yu et al. “Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems”. In: *arXiv preprint arXiv:2111.02801* (2021).

[44] Aston Zhang et al. “Dive into Deep Learning”. In: *CoRR* abs/2106.11342 (2021). arXiv: `2106.11342`. URL: `https://arxiv.org/abs/2106.11342`.

[45] Tongtao Zhang et al. “Frequency-compensated PINNs for Fluid-dynamic Design Problems”. In: *arXiv preprint arXiv:2011.01456* (2020).

[46] Kirill Zubov et al. “NeuralPDE: Automating physics-informed neural networks (PINNs) with error approximations”. In: *arXiv preprint arXiv:2107.09443* (2021).

A

Source Codes

A.1. Fenics Codes (Numerical Codes)

A.1.1. Fenics Code Stokes Flow

```python
from dolfin import *
from mshr import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from airfoils import Airfoil

## Simulation Parameters
alfa = 5
u_stream = 1
chord = 1
kinematic_visc = 1.5e-2
Re = Constant(u_stream * chord / kinematic_visc)
p_right = Constant(0)


airfoil_name = '4812'
x_0 = 5
y_0 = 3
points = 200 # keep it even
foil = Airfoil.NACA4(airfoil_name, n_points=points)
data = foil.all_points.T
data[points:,:] = np.flip(data[points:,:],0)
data = np.flip(data,0)
domain_vertices = []

for i in range(2*points):
    domain_vertices.append(Point(data[i,0]+x_0/4,data[i,1]+y_0/2))
domain = Rectangle(Point(0, 0), Point(x_0, y_0)) - Polygon(domain_vertices)
mesh = generate_mesh(domain,60)

plt.figure(1, figsize = (2*x_0, 2*y_0))
plot(mesh)

# Making a mark dictionary
# Note: the values should be UNIQUE identifiers.
mark = {"generic": 0,
"wall": 1,
"left": 2,
"right": 3,
"airfoil" : 4
        }

subdomains = MeshFunction("size_t", mesh, 1)
subdomains.set_all(mark["generic"])

```

```
class Left(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 0)

class Right(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], x_0)

class Wall(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and (near(x[1], 0) or near(x[1], y_0))

class Airfoil(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[0]>x_0/6 and x[0]<2*x_0/3 and x[1]>y_0/4 and x[1]<3*y_0/4

left = Left()
left.mark(subdomains, mark["left"])

right = Right()
right.mark(subdomains, mark["right"])

wall = Wall()
wall.mark(subdomains, mark["wall"])

airfoil = Airfoil()
airfoil.mark(subdomains, mark["airfoil"])

# Define function spaces
V = VectorElement("CG", triangle, 2)
P = FiniteElement("CG", triangle, 1)
W = FunctionSpace(mesh, V*P)

# Define variational problem
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)

dx = Measure("dx", domain=mesh, subdomain_data=subdomains) # Volume integration
ds = Measure("ds", domain=mesh, subdomain_data=subdomains) # Surface integration

# Surface normal
n = FacetNormal(mesh)
alfa = np.deg2rad(alfa)
# Pressures. First define the numbers (for later use):
u_left = Expression(('cos(alfa)','sin(alfa)'),degree=2,alfa = alfa, u_stream = u_stream)



a = (1/Re)*inner(grad(u), grad(v))*dx - p*div(v)*dx + q*div(u)*dx
L = inner(Constant((0, 0)), v)*dx


noslip = Constant((0.0, 0.0))
slip_wall = Expression(('cos(alfa)','sin(alfa)'),degree=2,alfa = alfa, u_stream = u_stream)
bc_wall = DirichletBC(W.sub(0), slip_wall, subdomains, mark["wall"])
bc_cylinder = DirichletBC(W.sub(0), noslip, subdomains, mark["airfoil"])
bc_left = DirichletBC(W.sub(0), u_left, subdomains, mark["left"]) # Velocity Inlet BC
bc_right = DirichletBC(W.sub(1), p_right, subdomains, mark["right"])
bcs = [bc_wall,bc_cylinder,bc_left, bc_right]

# Compute solution
w = Function(W)
solve(a == L, w, bcs)

# Split using deep copy
(u, p) = w.split(True)

# Plot solution
plt.figure(2, figsize = (2*x_0, 2*y_0))
plot(u, title="Velocity")
plt.colorbar(plot(u))
```

```python

plt.figure(3, figsize = (2*x_0, 2*y_0))
plot(p, title="Pressure")
plt.colorbar(plot(p))

# Magnitude function
def magnitude(vec):
    return sqrt(vec**2)
plt.figure(4, figsize = (2*x_0, 2*y_0))
plot(magnitude(u), title="Speed", cmap = 'brg')
plt.colorbar(plot(magnitude(u)))
```

A.1.2. Fenics Code Navier Stokes Flow

```python
from dolfin import *
from mshr import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from airfoils import Airfoil

airfoil_name = '0012'
alfa = 5


u_stream = 1
chord = 1
kinematic_visc = 1.5e-3
Re = Constant(u_stream * chord / kinematic_visc)
p_right = Constant(0)


x_0 = 5
y_0 = 3
points = 200 # keep it even

foil = Airfoil.NACA4(airfoil_name, n_points=points)
data = foil.all_points.T
data[points:,:] = np.flip(data[points:,:],0)
data = np.flip(data,0)
domain_vertices = []

for i in range(2*points):
    domain_vertices.append(Point(data[i,0]+x_0/4,data[i,1]+y_0/2))


domain = Rectangle(Point(0, 0), Point(x_0, y_0)) - Polygon(domain_vertices)
mesh = generate_mesh(domain,60)
plt.figure(1, figsize = (2*x_0, 2*y_0))
plot(mesh)

# Making a mark dictionary
# Note: the values should be UNIQUE identifiers.
mark = {"generic": 0,
"wall": 1,
"left": 2,
"right": 3,
"airfoil" : 4
       }

subdomains = MeshFunction("size_t", mesh, 1)
subdomains.set_all(mark["generic"])

class Left(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 0)

class Right(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], x_0)
```

```

class Wall(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and (near(x[1], 0) or near(x[1], y_0))

class Airfoil_bd(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[0]>x_0/6 and x[0]<2*x_0/3 and x[1]>y_0/4 and x[1]<3*y_0/4

left = Left()
left.mark(subdomains, mark["left"])

right = Right()
right.mark(subdomains, mark["right"])

wall = Wall()
wall.mark(subdomains, mark["wall"])

airfoil = Airfoil_bd()
airfoil.mark(subdomains, mark["airfoil"])

# Define function spaces
V = VectorElement("P", mesh.ufl_cell(), 2)
P = FiniteElement("P", mesh.ufl_cell(), 1)
TH = MixedElement([V, P])
W = FunctionSpace(mesh, TH)

# Define variational problem
v, q = TestFunctions(W)
w = Function(W)
u, p = split(w)


dx = Measure("dx", domain=mesh, subdomain_data=subdomains) # Volume integration
ds = Measure("ds", domain=mesh, subdomain_data=subdomains, subdomain_id = mark["airfoil"]) #
    Surface integration

# Surface normal
n = FacetNormal(mesh)
alfa = np.deg2rad(alfa)
u_left = Expression(('cos(alfa)','sin(alfa)'),degree=2,alfa = alfa, u_stream = u_stream)


a = dot(dot(grad(u),u), v)*dx + (1/Re)*inner(grad(u), grad(v))*dx - p*div(v)*dx - q*div(u)*dx


noslip = Constant((0.0, 0.0))
slip_wall = u_left
bc_wall = DirichletBC(W.sub(0), slip_wall, subdomains, mark["wall"])
bc_cylinder = DirichletBC(W.sub(0), noslip, subdomains, mark["airfoil"])
bc_left = DirichletBC(W.sub(0), u_left, subdomains, mark["left"]) # Velocity Inlet BC
bc_right = DirichletBC(W.sub(1), p_right, subdomains, mark["right"])
bcs = [bc_wall,bc_cylinder,bc_left, bc_right]

# Compute solution
solve(a == 0, w, bcs, solver_parameters={"newton_solver":{"relative_tolerance":1e-8},"
    newton_solver":{"maximum_iterations":200}})

# Split using deep copy
(u, p) = w.split(True)


# Plot solution
plt.figure(2, figsize = (2*x_0, 2*y_0))
plot(u, title="Velocity")
plt.colorbar(plot(u))


plt.figure(3, figsize = (2*x_0, 2*y_0))
plot(p, title="Pressure")
plt.colorbar(plot(p))
```

```

# Magnitude function
def magnitude(vec):
    return sqrt(vec**2)
plt.figure(4, figsize = (2*x_0, 2*y_0))
plot(magnitude(u), title="Velocity Magnitude", cmap = 'brg')
plt.colorbar(plot(magnitude(u)))

## Force Computations
ds_airfoil = Measure("ds", subdomain_data=subdomains, subdomain_id=mark["airfoil"])
n = FacetNormal(w.function_space().mesh())
force = -p*n + kinematic_visc*dot(grad(u), n)
F_D = assemble(-force[0]*ds_airfoil)
F_L = assemble(-force[1]*ds_airfoil)

print(F_D)
print(F_L)
```

A.2. Deep Learning Codes

A.2.1. Unparameterized Poisson Equation Code

```python
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
from random import randint

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

## Hyperparameters

x_0 = 1
y_0 = 2
max_epochs = 1000000
num_variables = 2
num_inputs = 4
num_layers = 8
num_neurons = 40
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = 1
u_array = np.ndarray.tolist(u_array.astype(int))
nxp = 50
nyp = 50
nbxp = 100
nbyp = 100
nx = int(nxp * x_0)
ny = int(nyp * y_0)
nbx = int(nbxp * x_0)
nby = int(nbyp * y_0)
learning_rate = 0.0001
iteration = 0
lamda = 0.01


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

u_net.to(device)

## Load Model
#FILE = 'u_net_para_perturbed_low_BC.pth'
#FILE = 'u_net_hpc.pth'
#FILE = '/home/nfs/skakkar/model_data/poisson_init.pth'
#u_net.load_state_dict(torch.load(FILE))
#u_net.eval()

##########################################################


## Generating Data Points
def interior_points(nx, ny, x_0, y_0):
    xy = torch.rand(nx * ny, num_variables).float()
    xy[:, 0] *= x_0
    xy[:, 1] *= y_0

    x_0_array = x_0 * torch.ones((len(xy[:, 0]), 1)).float()
    y_0_array = y_0 * torch.ones((len(xy[:, 1]), 1)).float()

    xy = torch.from_numpy(np.hstack((xy, x_0_array)))
    xy = torch.from_numpy(np.hstack((xy, y_0_array))).to(device)

    return xy


## Generating Boundary Points
def bound_points(nbx, nby, x_0, y_0):
    xb = torch.rand(nbx, 1) * x_0
    yb = torch.rand(nby, 1) * y_0

    xy_b_1 = torch.hstack((xb, y_0 * torch.ones((nbx, 1))))
    xy_b_2 = torch.hstack((xb, torch.zeros((nbx, 1))))
    xy_b_3 = torch.hstack((x_0 * torch.ones((nby, 1)), yb))
    xy_b_4 = torch.hstack((torch.zeros((nby, 1)), yb))

    xy_b = torch.vstack((xy_b_1, xy_b_2, xy_b_3, xy_b_4)).float()

    x_0_array = x_0 * torch.ones((len(xy_b[:, 0]), 1)).float()
    y_0_array = y_0 * torch.ones((len(xy_b[:, 1]), 1)).float()

    xy_b = torch.from_numpy(np.hstack((xy_b, x_0_array)))
    xy_b = torch.from_numpy(np.hstack((xy_b, y_0_array))).to(device)

    return xy_b


xy = interior_points(nx, ny, x_0, y_0)
xy_b = bound_points(nbx, nby, x_0, y_0)
optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)
f_source = -2 * pi * pi * sin(pi * xy[:, 0]/x_0) * sin(pi * xy[:, 1]/y_0).float().to(device)


def f_pde(XY):
    x = torch.tensor(XY[:, 0], requires_grad=True).float().cpu()
    y = torch.tensor(XY[:, 1], requires_grad=True).float().cpu()
    x_0 = np.asscalar(XY[0, 2].cpu().numpy())
    y_0 = np.asscalar(XY[0, 3].cpu().numpy())
    xy_pde = torch.hstack((torch.reshape(x, (-1, 1)), torch.reshape(y, (-1, 1)))).float()
    x_0_array = x_0 * torch.ones((len(xy_pde[:, 0]), 1)).float()
    y_0_array = y_0 * torch.ones((len(xy_pde[:, 1]), 1)).float()

    xy_pde = torch.hstack((xy_pde, x_0_array))
    xy_pde = torch.hstack((xy_pde, y_0_array)).to(device)

    u = u_net(xy_pde)
    x.to(device)
    y.to(device)

    u_x = torch.autograd.grad(
        u, x,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0]
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True,
        create_graph=True
    )[0]
    u_y = torch.autograd.grad(
        u, y,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0]
    u_yy = torch.autograd.grad(
        u_y, y,
        grad_outputs=torch.ones_like(u_y),
        retain_graph=True,
        create_graph=True
    )[0]

    f = u_yy + u_xx
    return f.to(device)

def loss_func(XY, XY_b, f_rhs, weight):
    pde_loss = f_pde(XY) - f_rhs
    bc_loss = u_net(XY_b)
    loss_val = weight * torch.mean(pde_loss ** 2) + (1 - weight) * torch.mean(bc_loss ** 2)

    return loss_val


epoch = 0
loss = loss_func(xy, xy_b, f_source, lamda)

## Training Loop
while epoch <= max_epochs and loss.item() > 1e-6:
    # Perturbation
    if epoch % 5 == 0:

        xy = interior_points(nx, ny, x_0, y_0)
        xy_b = bound_points(nbx, nby, x_0, y_0)

        f_source = -2 * pi * pi * sin(pi * xy[:, 0]/x_0) * sin(pi * xy[:, 1]/y_0).float().to(
            device)

    # forward and loss
    loss = loss_func(xy, xy_b, f_source, lamda)

    # backward
    loss.backward()

    # update
    optimizer.step()

    if epoch % 500 == 0:
        print(f'epoch: {epoch}, loss: {loss.item()}', flush = True)

    optimizer.zero_grad()
    epoch += 1

###########################################################################

FILE = '/home/nfs/skakkar/model_data/u_net_unparametrized_long.pth'
#FILE = 'poisson_init.pth'
torch.save(u_net.state_dict(), FILE)
```

A.2.2. Parameterized Poisson Equation

```
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
from random import randint

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

## Hyperparameters

x_0 = 1
y_0 = 2
int_array = np.arange(1,11,2)
max_epochs = 1000000
num_variables = 2
num_inputs = 4
num_layers = 8
num_neurons = 40
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = 1
u_array = np.ndarray.tolist(u_array.astype(int))
nxp = 50
nyp = 50
nbxp = 100
nbyp = 100
nx = int(nxp * x_0)
ny = int(nyp * y_0)
nbx = int(nbxp * x_0)
nby = int(nbyp * y_0)
learning_rate = 0.0001
iteration = 0
lamda = 0.01


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

u_net.to(device)

## Load Model
#FILE = 'u_net_para_perturbed_low_BC.pth'
#FILE = 'u_net_hpc.pth'
#FILE = '/home/nfs/skakkar/model_data/poisson_init.pth'
#u_net.load_state_dict(torch.load(FILE))
#u_net.eval()

#########################################################


## Generating Data Points
def interior_points(nx, ny, x_0, y_0):
    xy = torch.rand(nx * ny, num_variables).float()
    xy[:, 0] *= x_0
    xy[:, 1] *= y_0

    x_0_array = x_0 * torch.ones((len(xy[:, 0]), 1)).float()
    y_0_array = y_0 * torch.ones((len(xy[:, 1]), 1)).float()

    xy = torch.from_numpy(np.hstack((xy, x_0_array)))
    xy = torch.from_numpy(np.hstack((xy, y_0_array))).to(device)

    return xy


## Generating Boundary Points
def bound_points(nbx, nby, x_0, y_0):
    xb = torch.rand(nbx, 1) * x_0
    yb = torch.rand(nby, 1) * y_0

    xy_b_1 = torch.hstack((xb, y_0 * torch.ones((nbx, 1))))
    xy_b_2 = torch.hstack((xb, torch.zeros((nbx, 1))))
    xy_b_3 = torch.hstack((x_0 * torch.ones((nby, 1)), yb))
```

```python
    xy_b_4 = torch.hstack((torch.zeros((nby, 1)), yb))

    xy_b = torch.vstack((xy_b_1, xy_b_2, xy_b_3, xy_b_4)).float()

    x_0_array = x_0 * torch.ones((len(xy_b[:, 0]), 1)).float()
    y_0_array = y_0 * torch.ones((len(xy_b[:, 1]), 1)).float()

    xy_b = torch.from_numpy(np.hstack((xy_b, x_0_array)))
    xy_b = torch.from_numpy(np.hstack((xy_b, y_0_array))).to(device)

    return xy_b


xy = interior_points(nx, ny, x_0, y_0)
xy_b = bound_points(nbx, nby, x_0, y_0)
optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)
f_source = -2 * pi * pi * sin(pi * xy[:, 0]/x_0) * sin(pi * xy[:, 1]/y_0).float().to(device)


def f_pde(XY):
    x = torch.tensor(XY[:, 0], requires_grad=True).float().cpu()
    y = torch.tensor(XY[:, 1], requires_grad=True).float().cpu()
    x_0 = np.asscalar(XY[0, 2].cpu().numpy())
    y_0 = np.asscalar(XY[0, 3].cpu().numpy())
    xy_pde = torch.hstack((torch.reshape(x, (-1, 1)), torch.reshape(y, (-1, 1)))).float()
    x_0_array = x_0 * torch.ones((len(xy_pde[:, 0]), 1)).float()
    y_0_array = y_0 * torch.ones((len(xy_pde[:, 1]), 1)).float()

    xy_pde = torch.hstack((xy_pde, x_0_array))
    xy_pde = torch.hstack((xy_pde, y_0_array)).to(device)

    u = u_net(xy_pde)
    x.to(device)
    y.to(device)

    u_x = torch.autograd.grad(
        u, x,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0]
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True,
        create_graph=True
    )[0]
    u_y = torch.autograd.grad(
        u, y,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0]
    u_yy = torch.autograd.grad(
        u_y, y,
        grad_outputs=torch.ones_like(u_y),
        retain_graph=True,
        create_graph=True
    )[0]

    f = u_yy + u_xx
    return f.to(device)

def loss_func(XY, XY_b, f_rhs, weight):
    pde_loss = f_pde(XY) - f_rhs
    bc_loss = u_net(XY_b)
    loss_val = weight * torch.mean(pde_loss ** 2) + (1 - weight) * torch.mean(bc_loss ** 2)

    return loss_val


```

```python
epoch = 0
loss = loss_func(xy, xy_b, f_source, lamda)

## Training Loop
while epoch <= max_epochs and loss.item() > 1e-6:
    # Perturbation
    if epoch % 5 == 0:

        if epoch % 100 == 0:
            x_0 = np.random.choice(int_array)
            y_0 = np.random.choice(int_array)
            nx = int(nxp * x_0)
            ny = int(nyp * y_0)
            nbx = int(nbxp * x_0)
            nby = int(nbyp * y_0)
        xy = interior_points(nx, ny, x_0, y_0)
        xy_b = bound_points(nbx, nby, x_0, y_0)

        f_source = -2 * pi * pi * sin(pi * xy[:, 0]/x_0) * sin(pi * xy[:, 1]/y_0).float().to(
            device)

    # forward and loss
    loss = loss_func(xy, xy_b, f_source, lamda)

    # backward
    loss.backward()

    # update
    optimizer.step()

    if epoch % 500 == 0:
        print(f'epoch: {epoch}, loss: {loss.item()}', flush = True)

    optimizer.zero_grad()
    epoch += 1

############################################################################

FILE = '/home/nfs/skakkar/model_data/u_net_unparametrized_long.pth'
#FILE = 'poisson_init.pth'
torch.save(u_net.state_dict(), FILE)
```

A.2.3. Poisson PINN Postprocessing Script

```python
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
import matplotlib.pylab as plt
from random import randint


## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

## Hyperparameters
x_0 = 4
y_0 = 2
num_variables = 2
num_inputs = 4
num_layers = 8 #4 for u_net_para, 8 for perturbed
num_neurons = 40 #50 for u_net_para, 20 for perturbed
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = 1
u_array = np.ndarray.tolist(u_array.astype(int))
nx = int(150*x_0)
```

```
ny = int(150*y_0)
nbx = int(250*x_0)
nby = int(250*y_0)


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out

u_net = neural_net(u_array).to(device)

## Load Model
#FILE = 'u_net_parallel_integer.pth'
#FILE = 'u_net_parallel_narrow_latest.pth'
FILE = 'u_net_parallel_wide_latest.pth'
u_net.load_state_dict(torch.load(FILE))
u_net.eval()

## Plotting
nx_plot = 200
ny_plot = 200
x = torch.linspace(0, x_0, nx_plot)
y = torch.linspace(0, y_0, ny_plot)


X, Y = np.meshgrid(x,y)

X_pred = torch.from_numpy(np.hstack((X.flatten()[:,None], Y.flatten()[:,None])))

x_0_array = x_0*torch.ones((len(X_pred[:,0]),1)).float()
y_0_array = y_0*torch.ones((len(X_pred[:,1]),1)).float()

X_pred = torch.from_numpy(np.hstack((X_pred, x_0_array)))
X_pred = torch.from_numpy(np.hstack((X_pred, y_0_array))).to(device)

# Deep Learning Solution
u_pred = u_net(X_pred)

u_numpy = np.reshape(u_pred.detach().cpu().numpy(), (nx_plot, ny_plot))

plt.ion()
plt.figure(figsize=(4*x_0, 4*y_0))
plt.clf()

plt.subplot(1,3,1)
plt.imshow(np.flip(u_numpy, 0), extent=[0, x_0, 0, y_0])
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar(orientation='horizontal')
plt.title('Deep Learning Solution')
plt.show()
#plt.savefig('poisson_rectangle.jpg')

# Analytical Solution
u_pred = sin(pi*X_pred[:,0]/x_0)*sin(pi*X_pred[:,1]/y_0)
u_actual = np.reshape(u_pred.detach().cpu().numpy(), (nx_plot, ny_plot))

#plt.ion()
plt.figure(figsize=(4*x_0, 4*y_0))
#plt.clf()

plt.subplot(1,3,2)
plt.imshow(np.flip(u_actual, 0), extent=[0, x_0, 0, y_0])
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar(orientation='horizontal')
plt.title('Analytical Solution')
plt.savefig('poisson_analytical_unit.jpg')
plt.show()

plt.figure(figsize=(4*x_0, 4*y_0))
plt.subplot(1,3,3)
plt.imshow(np.flip(abs(u_actual-u_numpy), 0), extent=[0, x_0, 0, y_0])
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar(orientation='horizontal')
plt.title('Error')
plt.show()
```

A.2.4. Unparameterized Airfoil Stokes Flow Code

```
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
from random import randint
from airfoils import Airfoil
import matplotlib.pyplot as plt
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

x_0 = 5
y_0 = 3
nxp = 20
nyp = 20
num_variables = 2
max_epochs = 1000000
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 4    # default 4
num_neurons = 20  # default 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))
nx = int(nxp * x_0)
ny = int(nyp * y_0)

airfoil_name = '4812'
```

```
mesh_file_name = '/home/nfs/skakkar/jupyter_codes/
    mesh_data_stokes_correctedpressure_53_4812_lowRe.npy'
sol_file_name = '/home/nfs/skakkar/jupyter_codes/
    solution_data_stokes_correctedpressure_53_4812_lowRe.npy'
alfa = 5 # degrees
u_stream = 1 # m/s
learning_rate = 0.0001   # default = 0.0001
data_lamda = 0.6  # default 0.6
chord = 1 # m
kin_viscosity = 1.5e-2 # m^2/s
Re = chord * u_stream / kin_viscosity


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

#FILE = 'airfoil_lowRe_31_stokes_myPC.pth'
#u_net.load_state_dict(torch.load(FILE))
#u_net.eval()

## Generating Data Points
def interior_points(nx, ny, x_0, y_0, airfoil_name, alfa):

    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])

    xy = np.random.rand(nx * ny, num_variables)
    xy[:, 0] *= x_0
    xy[:, 1] *= y_0

    indices = []

    for i in np.arange(len(xy[:,0])):
        if xy[i,0]>=x_0/4:
            if xy[i,0]<=x_0/4+1:
                if xy[i,1]>=foil.y_lower(xy[i,0]-x_0/4)+y_0/2:
                    if xy[i,1]<=foil.y_upper(xy[i,0]-x_0/4)+y_0/2:
                        indices.append(i)

    xy = np.delete(xy, indices, axis = 0)

    xy = torch.from_numpy(xy).float()

    cam_array = camber * torch.ones((len(xy[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy[:, 0]), 1)).float()

    xy = torch.from_numpy(np.hstack((xy, cam_array)))
    xy = torch.from_numpy(np.hstack((xy, cam_pos_array)))
    xy = torch.from_numpy(np.hstack((xy, thickness_array)))
    xy = torch.from_numpy(np.hstack((xy, alfa_array))).to(device)

    return xy

def mesh_points(file_name, airfoil_name, alfa):
    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])

    xy_data = np.load(file_name)

    cam_array = camber * torch.ones((len(xy_data[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_data[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_data[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_data[:, 0]), 1)).float()

    xy_data = torch.from_numpy(np.hstack((xy_data, cam_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, cam_pos_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, thickness_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, alfa_array))).float().to(device)

    return xy_data

# Mesh Generation
xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)
xy_data = mesh_points(mesh_file_name, airfoil_name, alfa)
uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)


## PINN Formulation
optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)

def interior_loss(XY, Re): # remember order of three outputs (u,v,p)
    x = torch.tensor(XY[:, 0], requires_grad=True).float().cpu()
    y = torch.tensor(XY[:, 1], requires_grad=True).float().cpu()
    camber = np.asscalar(XY[0, 2].cpu().numpy())
    camber_pos = np.asscalar(XY[0, 3].cpu().numpy())
    thickness = np.asscalar(XY[0, 4].cpu().numpy())
    alfa = np.asscalar(XY[0, 5].cpu().numpy())

    xy_pde = torch.hstack((torch.reshape(x, (-1, 1)), torch.reshape(y, (-1, 1)))).float()
    cam_array = camber * torch.ones((len(xy_pde[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_pde[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_pde[:, 1]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_pde[:, 1]), 1)).float()

    xy_pde = torch.hstack((xy_pde, cam_array))
    xy_pde = torch.hstack((xy_pde, cam_pos_array))
    xy_pde = torch.hstack((xy_pde, thickness_array))
    xy_pde = torch.hstack((xy_pde, alfa_array)).to(device)

    u = u_net(xy_pde)[:,0].to(device)
    v = u_net(xy_pde)[:,1].to(device)
    p = u_net(xy_pde)[:,2].to(device)
    x.to(device)
    y.to(device)

    u_x = torch.autograd.grad(
        u, x,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_y = torch.autograd.grad(
        u, y,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_yy = torch.autograd.grad(
        u_y, y,
        grad_outputs=torch.ones_like(u_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    v_x = torch.autograd.grad(
        v, x,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_xx = torch.autograd.grad(
        v_x, x,
        grad_outputs=torch.ones_like(v_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_y = torch.autograd.grad(
        v, y,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_yy = torch.autograd.grad(
        v_y, y,
        grad_outputs=torch.ones_like(v_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_x = torch.autograd.grad(
        p, x,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_y = torch.autograd.grad(
        p, y,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    #f1 = p_xx + p_yy
    f1 = u_x + v_y
    f2 = u_xx/(Re) + u_yy/(Re) - p_x
```

```python
    f3 = v_xx/(Re) + v_yy/(Re) - p_y

    loss1 = torch.mean(f1 ** 2)
    loss2 = torch.mean(f2 ** 2)
    loss3 = torch.mean(f3 ** 2)

    return (loss1 + loss2 + loss3)

def loss_func(XY, data_weight, Re, XY_data, UVP_data):


    data_supervised = u_net(XY_data) - UVP_data


    loss_val   = (1-data_weight)*interior_loss(XY, Re) + (data_weight)*torch.mean(
        data_supervised**2)

    return loss_val

epoch = 0
loss = loss_func(xy, data_lamda, Re, xy_data, uvp_data)

## Training Loop
while epoch <= max_epochs and loss.item() > 1e-6:
    # Perturbation
    if epoch % 5 == 0:

        xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)

    # forward and loss
    loss = loss_func(xy, data_lamda,  Re , xy_data, uvp_data)

    # backward
    loss.backward()

    # update
    optimizer.step()

    if epoch % 100 == 0:
        print(f'epoch: {epoch}, loss: {loss.item()}, data_lamda = {data_lamda}, lr = {
            learning_rate}', flush = True)

    optimizer.zero_grad()
    epoch += 1

FILE = '/home/nfs/skakkar/model_data/airfoil_stokes_correctedpressure_neur_20_53_4812.pth'
torch.save(u_net.state_dict(), FILE)
```

A.2.5. Parameterized Airfoil Stokes Flow Code

```python
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
import random
from airfoils import Airfoil
import matplotlib.pyplot as plt
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

x_0 = 5
y_0 = 3
nxp = 20
nyp = 20
num_variables = 2
```

```python
max_epochs = 1000000
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 8   # default was 8 layers in generalised case
num_neurons = 20   # default 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))
nx = int(nxp * x_0)
ny = int(nyp * y_0)

data_dict = {'mesh_data_stokes_correctedpressure_53_0006_lowRe.npy':'
    solution_data_stokes_correctedpressure_53_0006_lowRe.npy',\
             'mesh_data_stokes_correctedpressure_53_0010_lowRe.npy':'
                 solution_data_stokes_correctedpressure_53_0010_lowRe.npy',\
            'mesh_data_stokes_correctedpressure_53_0014_lowRe.npy':'
                solution_data_stokes_correctedpressure_53_0014_lowRe.npy',\
            'mesh_data_stokes_correctedpressure_53_0018_lowRe.npy':'
                solution_data_stokes_correctedpressure_53_0018_lowRe.npy'}

mesh_file_name, sol_file_name = random.choice(list(data_dict.items()))
airfoil_name = mesh_file_name[-14:-10]

mesh_file_name = '/home/nfs/skakkar/jupyter_codes/' + mesh_file_name
sol_file_name = '/home/nfs/skakkar/jupyter_codes/' + sol_file_name
alfa = 5 # degrees
u_stream = 1 # m/s
learning_rate = 0.0001  # default 0.0001
data_lamda = 0.6  # default 0.6
chord = 1 # m
p_outlet = 1 # scaled between 0-1
kin_viscosity = 1.5e-2 # m^2/s
Re = chord * u_stream / kin_viscosity


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

#FILE = 'airfoil_lowRe_31_stokes_myPC.pth'
#u_net.load_state_dict(torch.load(FILE))
#u_net.eval()

## Generating Data Points
def interior_points(nx, ny, x_0, y_0, airfoil_name, alfa):

    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])

    xy = np.random.rand(nx * ny, num_variables)
    xy[:, 0] *= x_0
    xy[:, 1] *= y_0

    indices = []

    for i in np.arange(len(xy[:,0])):
        if xy[i,0]>=x_0/4:
            if xy[i,0]<=x_0/4+1:
                if xy[i,1]>=foil.y_lower(xy[i,0]-x_0/4)+y_0/2:
                    if xy[i,1]<=foil.y_upper(xy[i,0]-x_0/4)+y_0/2:
                        indices.append(i)

    xy = np.delete(xy, indices, axis = 0)

    xy = torch.from_numpy(xy).float()

    cam_array = camber * torch.ones((len(xy[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy[:, 0]), 1)).float()

    xy = torch.from_numpy(np.hstack((xy, cam_array)))
    xy = torch.from_numpy(np.hstack((xy, cam_pos_array)))
    xy = torch.from_numpy(np.hstack((xy, thickness_array)))
    xy = torch.from_numpy(np.hstack((xy, alfa_array))).to(device)

    return xy

def mesh_points(file_name, airfoil_name, alfa):
    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])

    xy_data = np.load(file_name)

    cam_array = camber * torch.ones((len(xy_data[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_data[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_data[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_data[:, 0]), 1)).float()

    xy_data = torch.from_numpy(np.hstack((xy_data, cam_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, cam_pos_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, thickness_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, alfa_array))).float().to(device)

    return xy_data

# Mesh Generation
xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)
xy_data = mesh_points(mesh_file_name, airfoil_name, alfa)
uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)


```

```python
## PINN Formulation
optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)

def interior_loss(XY, Re): # remember order of three outputs (u,v,p)
    x = torch.tensor(XY[:, 0], requires_grad=True).float().cpu()
    y = torch.tensor(XY[:, 1], requires_grad=True).float().cpu()
    camber = np.asscalar(XY[0, 2].cpu().numpy())
    camber_pos = np.asscalar(XY[0, 3].cpu().numpy())
    thickness = np.asscalar(XY[0, 4].cpu().numpy())
    alfa = np.asscalar(XY[0, 5].cpu().numpy())

    xy_pde = torch.hstack((torch.reshape(x, (-1, 1)), torch.reshape(y, (-1, 1)))).float()
    cam_array = camber * torch.ones((len(xy_pde[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_pde[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_pde[:, 1]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_pde[:, 1]), 1)).float()

    xy_pde = torch.hstack((xy_pde, cam_array))
    xy_pde = torch.hstack((xy_pde, cam_pos_array))
    xy_pde = torch.hstack((xy_pde, thickness_array))
    xy_pde = torch.hstack((xy_pde, alfa_array)).to(device)

    u = u_net(xy_pde)[:,0].to(device)
    v = u_net(xy_pde)[:,1].to(device)
    p = u_net(xy_pde)[:,2].to(device)
    x.to(device)
    y.to(device)

    u_x = torch.autograd.grad(
        u, x,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_y = torch.autograd.grad(
        u, y,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_yy = torch.autograd.grad(
        u_y, y,
        grad_outputs=torch.ones_like(u_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    v_x = torch.autograd.grad(
        v, x,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_xx = torch.autograd.grad(
        v_x, x,
        grad_outputs=torch.ones_like(v_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_y = torch.autograd.grad(
        v, y,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_yy = torch.autograd.grad(
        v_y, y,
        grad_outputs=torch.ones_like(v_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_x = torch.autograd.grad(
        p, x,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_y = torch.autograd.grad(
        p, y,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    #f1 = p_xx + p_yy
    f1 = u_x + v_y
    f2 = u_xx/(Re) + u_yy/(Re) - p_x
    f3 = v_xx/(Re) + v_yy/(Re) - p_y

    loss1 = torch.mean(f1 ** 2)
    loss2 = torch.mean(f2 ** 2)
    loss3 = torch.mean(f3 ** 2)

    return (loss1 + loss2 + loss3)

def loss_func(XY, data_weight, Re, XY_data, UVP_data):


    data_supervised = u_net(XY_data) - UVP_data


    loss_val   = (1-data_weight)*interior_loss(XY, Re) + (data_weight)*torch.mean(
        data_supervised**2)

    return loss_val

epoch = 0
loss = loss_func(xy, data_lamda, Re, xy_data, uvp_data)

## Training Loop
while epoch <= max_epochs and loss.item() > 1e-6:
    # Perturbation
    if epoch % 5 == 0:

        mesh_file_name, sol_file_name = random.choice(list(data_dict.items()))
        airfoil_name = mesh_file_name[-14:-10]
        mesh_file_name = '/home/nfs/skakkar/jupyter_codes/' + mesh_file_name
        sol_file_name = '/home/nfs/skakkar/jupyter_codes/' + sol_file_name
        xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)
        xy_data = mesh_points(mesh_file_name, airfoil_name, alfa)
        uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)

    # forward and loss
    loss = loss_func(xy, data_lamda,  Re , xy_data, uvp_data)

    # backward
    loss.backward()

    # update
    optimizer.step()

    if epoch % 100 == 0:
        print(f'epoch: {epoch}, loss: {loss.item()}, data_lamda = {data_lamda}, lr = {
            learning_rate}', flush = True)

    optimizer.zero_grad()
    epoch += 1

FILE = '/home/nfs/skakkar/model_data/
    airfoil_stokes_generalised_correctedpressure_neur_20_53_06101418.pth'
torch.save(u_net.state_dict(), FILE)
```

A.2.6. Unparameterized Airfoil Navier Stokes Flow Code

```
#### HPC CODE ###############
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
from random import randint
from airfoils import Airfoil
import matplotlib.pyplot as plt
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

x_0 = 5
y_0 = 3
nxp = 20
nyp = 20
num_variables = 2
max_epochs = 1000000
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 8     # default 4
num_neurons = 100  # default 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))
nx = int(nxp * x_0)
ny = int(nyp * y_0)

airfoil_name = '4812'
mesh_file_name = '/home/nfs/skakkar/jupyter_codes/NS_mesh_data_53_4812_highRe.npy'
sol_file_name = '/home/nfs/skakkar/jupyter_codes/NS_solution_data_53_4812_highRe.npy'
alfa = 5 # degrees
u_stream = 1 # m/s
learning_rate = 0.0001   # default = 0.0001
data_lamda = 0.6  # default 0.6
chord = 1 # m
p_outlet = 1 # scaled between 0-1
kin_viscosity = 1.5e-3 # m^2/s
Re = chord * u_stream / kin_viscosity


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

#FILE = 'airfoil_lowRe_31_stokes_myPC.pth'
#u_net.load_state_dict(torch.load(FILE))
#u_net.eval()

## Generating Data Points
def interior_points(nx, ny, x_0, y_0, airfoil_name, alfa):

    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])

    xy = np.random.rand(nx * ny, num_variables)
    xy[:, 0] *= x_0
    xy[:, 1] *= y_0

    indices = []

    for i in np.arange(len(xy[:,0])):
        if xy[i,0]>=x_0/4:
            if xy[i,0]<=x_0/4+1:
                if xy[i,1]>=foil.y_lower(xy[i,0]-x_0/4)+y_0/2:
                    if xy[i,1]<=foil.y_upper(xy[i,0]-x_0/4)+y_0/2:
                        indices.append(i)

    xy = np.delete(xy, indices, axis = 0)

    xy = torch.from_numpy(xy).float()

    cam_array = camber * torch.ones((len(xy[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy[:, 0]), 1)).float()

    xy = torch.from_numpy(np.hstack((xy, cam_array)))
    xy = torch.from_numpy(np.hstack((xy, cam_pos_array)))
    xy = torch.from_numpy(np.hstack((xy, thickness_array)))
    xy = torch.from_numpy(np.hstack((xy, alfa_array))).to(device)

    return xy

def mesh_points(file_name, airfoil_name, alfa):
    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])
```

```python

    xy_data = np.load(file_name)

    cam_array = camber * torch.ones((len(xy_data[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_data[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_data[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_data[:, 0]), 1)).float()

    xy_data = torch.from_numpy(np.hstack((xy_data, cam_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, cam_pos_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, thickness_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, alfa_array))).float().to(device)

    return xy_data

# Mesh Generation
xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)
xy_data = mesh_points(mesh_file_name, airfoil_name, alfa)
uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)


## PINN Formulation
optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)

def interior_loss(XY, Re): # remember order of three outputs (u,v,p)
    x = torch.tensor(XY[:, 0], requires_grad=True).float().cpu()
    y = torch.tensor(XY[:, 1], requires_grad=True).float().cpu()
    camber = np.asscalar(XY[0, 2].cpu().numpy())
    camber_pos = np.asscalar(XY[0, 3].cpu().numpy())
    thickness = np.asscalar(XY[0, 4].cpu().numpy())
    alfa = np.asscalar(XY[0, 5].cpu().numpy())

    xy_pde = torch.hstack((torch.reshape(x, (-1, 1)), torch.reshape(y, (-1, 1)))).float()
    cam_array = camber * torch.ones((len(xy_pde[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_pde[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_pde[:, 1]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_pde[:, 1]), 1)).float()

    xy_pde = torch.hstack((xy_pde, cam_array))
    xy_pde = torch.hstack((xy_pde, cam_pos_array))
    xy_pde = torch.hstack((xy_pde, thickness_array))
    xy_pde = torch.hstack((xy_pde, alfa_array)).to(device)

    u = u_net(xy_pde)[:,0].to(device)
    v = u_net(xy_pde)[:,1].to(device)
    p = u_net(xy_pde)[:,2].to(device)
    x.to(device)
    y.to(device)

    u_x = torch.autograd.grad(
        u, x,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_y = torch.autograd.grad(
        u, y,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_yy = torch.autograd.grad(
        u_y, y,
        grad_outputs=torch.ones_like(u_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    v_x = torch.autograd.grad(
        v, x,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_xx = torch.autograd.grad(
        v_x, x,
        grad_outputs=torch.ones_like(v_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_y = torch.autograd.grad(
        v, y,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_yy = torch.autograd.grad(
        v_y, y,
        grad_outputs=torch.ones_like(v_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_x = torch.autograd.grad(
        p, x,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_y = torch.autograd.grad(
        p, y,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    #f1 = p_xx + p_yy
    f1 = u_x + v_y
    f2 = u_xx/(Re) + u_yy/(Re) - p_x - u*u_x - v*u_y
    f3 = v_xx/(Re) + v_yy/(Re) - p_y - u*v_x - v*v_y

    loss1 = torch.mean(f1 ** 2)
    loss2 = torch.mean(f2 ** 2)
    loss3 = torch.mean(f3 ** 2)

    return (loss1 + loss2 + loss3)

def loss_func(XY, data_weight, Re, XY_data, UVP_data):


    data_supervised = u_net(XY_data) - UVP_data


    loss_val   = (1-data_weight)*interior_loss(XY, Re) + (data_weight)*torch.mean(
        data_supervised**2)

    return loss_val

epoch = 0
loss = loss_func(xy, data_lamda, Re, xy_data, uvp_data)

## Training Loop
while epoch <= max_epochs and loss.item() > 1e-6:
    # Perturbation
    if epoch % 5 == 0:

        xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)

    # forward and loss
    loss = loss_func(xy, data_lamda,  Re , xy_data, uvp_data)

    # backward
    loss.backward()

    # update
    optimizer.step()

    if epoch % 100 == 0:
        print(f'epoch: {epoch}, loss: {loss.item()}, data_lamda = {data_lamda}, lr = {
            learning_rate}', flush = True)

    optimizer.zero_grad()
    epoch += 1

FILE = '/home/nfs/skakkar/model_data/airfoil_navier_stokes_highRe_layers_8_53_4812.pth'
torch.save(u_net.state_dict(), FILE)
```

A.2.7. Parameterized Airfoil Navier Stokes Flow Code

```
#### HPC CODE ###############
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
import random
from airfoils import Airfoil
import matplotlib.pyplot as plt
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

x_0 = 5
y_0 = 3
nxp = 20
nyp = 20
num_variables = 2
max_epochs = 4000000
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 4     # final value 4
num_neurons = 100  # final value 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))
nx = int(nxp * x_0)
ny = int(nyp * y_0)

data_dict = {'NS_mesh_data_BC_53_AOA_5_0012_highRe.npy':'
    NS_solution_data_BC_53_AOA_5_0012_highRe.npy',\
             'NS_mesh_data_BC_53_AOA_5_2412_highRe.npy':'
                 NS_solution_data_BC_53_AOA_5_2412_highRe.npy',\
             'NS_mesh_data_BC_53_AOA_5_4412_highRe.npy':'
                 NS_solution_data_BC_53_AOA_5_4412_highRe.npy',\
             'NS_mesh_data_BC_53_AOA_5_6412_highRe.npy':'
                 NS_solution_data_BC_53_AOA_5_6412_highRe.npy',\
             'NS_mesh_data_BC_53_AOA_5_4812_highRe.npy':'
                 NS_solution_data_BC_53_AOA_5_4812_highRe.npy',\
             'NS_mesh_data_BC_53_AOA_5_4212_highRe.npy':'
                 NS_solution_data_BC_53_AOA_5_4212_highRe.npy',\
            'NS_mesh_data_BC_53_AOA_5_4612_highRe.npy':'
                NS_solution_data_BC_53_AOA_5_4612_highRe.npy'}

mesh_file_name, sol_file_name = random.choice(list(data_dict.items()))
airfoil_name = mesh_file_name[-15:-11] # for lowRe, will become -15:-11 for highRe
alfa = int(mesh_file_name[-17]) # degrees for highRe

mesh_file_name = '/home/nfs/skakkar/jupyter_codes/' + mesh_file_name
sol_file_name = '/home/nfs/skakkar/jupyter_codes/' + sol_file_name

u_stream = 1 # m/s
learning_rate = 0.0001   # final value starting 0.0001
data_lamda = 0.8  # final value 0.2 changed to 0.8
pres_lamda = 0.8
chord = 1 # m
p_outlet = 1 # scaled between 0-1
kin_viscosity = 1.5e-3 # m^2/s
Re = chord * u_stream / kin_viscosity


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

#FILE = 'airfoil_lowRe_31_stokes_myPC.pth'
#u_net.load_state_dict(torch.load(FILE))
#u_net.eval()

## Generating Data Points
def interior_points(nx, ny, x_0, y_0, airfoil_name, alfa):

    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])

    xy = np.random.rand(nx * ny, num_variables)
```

```python
    xy[:, 0] *= x_0
    xy[:, 1] *= y_0

    indices = []

    for i in np.arange(len(xy[:,0])):
        if xy[i,0]>=x_0/4:
            if xy[i,0]<=x_0/4+1:
                if xy[i,1]>=foil.y_lower(xy[i,0]-x_0/4)+y_0/2:
                    if xy[i,1]<=foil.y_upper(xy[i,0]-x_0/4)+y_0/2:
                        indices.append(i)

    xy = np.delete(xy, indices, axis = 0)

    xy = torch.from_numpy(xy).float()

    cam_array = camber * torch.ones((len(xy[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy[:, 0]), 1)).float()

    xy = torch.from_numpy(np.hstack((xy, cam_array)))
    xy = torch.from_numpy(np.hstack((xy, cam_pos_array)))
    xy = torch.from_numpy(np.hstack((xy, thickness_array)))
    xy = torch.from_numpy(np.hstack((xy, alfa_array))).to(device)

    return xy

def mesh_points(file_name, airfoil_name, alfa):
    foil = Airfoil.NACA4(airfoil_name)
    camber = int(airfoil_name[0])
    camber_pos = int(airfoil_name[1])
    thickness = int(airfoil_name[2:])

    xy_data = np.load(file_name)

    cam_array = camber * torch.ones((len(xy_data[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_data[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_data[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_data[:, 0]), 1)).float()

    xy_data = torch.from_numpy(np.hstack((xy_data, cam_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, cam_pos_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, thickness_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, alfa_array))).float().to(device)

    return xy_data

# Mesh Generation
xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)
xy_data = mesh_points(mesh_file_name, airfoil_name, alfa)
uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)


## PINN Formulation
optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)

def interior_loss(XY, Re): # remember order of three outputs (u,v,p)
    x = torch.tensor(XY[:, 0], requires_grad=True).float().cpu()
    y = torch.tensor(XY[:, 1], requires_grad=True).float().cpu()
    camber = np.asscalar(XY[0, 2].cpu().numpy())
    camber_pos = np.asscalar(XY[0, 3].cpu().numpy())
    thickness = np.asscalar(XY[0, 4].cpu().numpy())
    alfa = np.asscalar(XY[0, 5].cpu().numpy())

    xy_pde = torch.hstack((torch.reshape(x, (-1, 1)), torch.reshape(y, (-1, 1)))).float()
    cam_array = camber * torch.ones((len(xy_pde[:, 0]), 1)).float()
    cam_pos_array = camber_pos * torch.ones((len(xy_pde[:, 1]), 1)).float()
    thickness_array = thickness * torch.ones((len(xy_pde[:, 1]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_pde[:, 1]), 1)).float()

    xy_pde = torch.hstack((xy_pde, cam_array))
    xy_pde = torch.hstack((xy_pde, cam_pos_array))
    xy_pde = torch.hstack((xy_pde, thickness_array))
    xy_pde = torch.hstack((xy_pde, alfa_array)).to(device)

    u = u_net(xy_pde)[:,0].to(device)
    v = u_net(xy_pde)[:,1].to(device)
    p = u_net(xy_pde)[:,2].to(device)
    x.to(device)
    y.to(device)

    u_x = torch.autograd.grad(
        u, x,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_y = torch.autograd.grad(
        u, y,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_yy = torch.autograd.grad(
        u_y, y,
        grad_outputs=torch.ones_like(u_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    v_x = torch.autograd.grad(
        v, x,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_xx = torch.autograd.grad(
        v_x, x,
        grad_outputs=torch.ones_like(v_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_y = torch.autograd.grad(
        v, y,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_yy = torch.autograd.grad(
        v_y, y,
        grad_outputs=torch.ones_like(v_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_x = torch.autograd.grad(
        p, x,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_y = torch.autograd.grad(
        p, y,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    #f1 = p_xx + p_yy
    f1 = u_x + v_y
    f2 = u_xx/(Re) + u_yy/(Re) - p_x - u*u_x - v*u_y
    f3 = v_xx/(Re) + v_yy/(Re) - p_y - u*v_x - v*v_y

    loss1 = torch.mean(f1 ** 2)
    loss2 = torch.mean(f2 ** 2)
    loss3 = torch.mean(f3 ** 2)

    return (loss1 + loss2 + loss3)

def loss_func(XY, data_weight,pres_weight, Re, XY_data, UVP_data):


    data_supervised_vel = u_net(XY_data)[:,0:2] - UVP_data[:,0:2]
    data_supervised_pres = u_net(XY_data)[:,2] - UVP_data[:,2]


    loss_val   = (1-data_weight)*interior_loss(XY, Re) + (data_weight)*(pres_weight*torch.
        mean(data_supervised_pres**2) +\
                                                               (1-pres_weight)*torch.mean(
                                                                   data_supervised_vel**2))

    return loss_val

epoch = 0
loss = loss_func(xy, data_lamda,pres_lamda, Re, xy_data, uvp_data)

## Training Loop
while epoch <= max_epochs and loss.item() > 1e-6:
    # Perturbation
    if epoch % 5 == 0:

        mesh_file_name, sol_file_name = random.choice(list(data_dict.items()))
        airfoil_name = mesh_file_name[-15:-11]  # for lowRe, change for highRe
        alfa = int(mesh_file_name[-17]) # degrees for highRe
        mesh_file_name = '/home/nfs/skakkar/jupyter_codes/' + mesh_file_name
        sol_file_name = '/home/nfs/skakkar/jupyter_codes/' + sol_file_name
        xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa)
        xy_data = mesh_points(mesh_file_name, airfoil_name, alfa)
        uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)

    if (epoch % 1000000 == 0) and (epoch != 0):
        learning_rate = learning_rate/10
        optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)


    # forward and loss
    loss = loss_func(xy, data_lamda,pres_lamda, Re , xy_data, uvp_data)

    # backward
    loss.backward()

    # update
    optimizer.step()

    if epoch % 100 == 0:
        print(f'epoch: {epoch}, loss: {loss.item()}, data_lamda = {data_lamda}, lr = {
            learning_rate}', flush = True)

    optimizer.zero_grad()
    epoch += 1

FILE = '/home/nfs/skakkar/model_data/
    airfoil_navier_stokes_cambered_finalised_highRe_53_00244864.pth'
torch.save(u_net.state_dict(), FILE)
```

A.2.8. Airfoil Navier Stokes Flow Post Processing Script

```python
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
from random import randint
from airfoils import Airfoil
import matplotlib.pyplot as plt
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

x_0 = 5
y_0 = 3

num_variables = 2
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 4
num_neurons = 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))


airfoil_name = '0020'
alfa = 1 # degrees
chord = 1 # m



## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

```

```python
if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

#FILE = '
    airfoil_navier_stokes_generalised_highRe_adaptive_long_wt_80_pres_part_long_53_18222630.
    pth'
#FILE = '
    airfoil_navier_stokes_generalised_highRe_adaptive_long_wt_80_pres_part_long_53_06101418.
    pth'
#FILE = 'airfoil_navier_stokes_generalised_highRe_adaptive_long_second_53_06101418.pth'
#FILE = 'airfoil_navier_stokes_generalised_highRe_adaptive_long_wt_80_53_06101418.pth'
#FILE = 'airfoil_stokes_corrected_BC_100_neur_4_layers_wt_60_53_4812.pth'
#FILE = 'airfoil_navier_stokes_aoa_wide_highRe_53_048.pth'
#FILE = 'airfoil_navier_stokes_cambered_varying_thickness_53_48_10141822.pth'
#FILE = 'airfoil_navier_stokes_cambered_highRe_53_002448.pth'
#FILE = 'airfoil_navier_stokes_cambered_finalised_highRe_53_00244864.pth'
FILE = 'airfoil_navier_stokes_fully_final_validation_long_53_10141822.pth'
#FILE = 'airfoil_stokes_correctedpressure_default_53_4812.pth'
#FILE = 'airfoil_stokes_generalised_correctedpressure_default_53_06101418.pth'
#FILE = 'airfoil_navier_stokes_highRe_wt80_53_4812.pth'
#FILE = 'airfoil_navier_stokes_fully_final_long_53_10141822.pth'
u_net.load_state_dict(torch.load(FILE))
u_net.eval()


# Plotting

nx_plot = 300
ny_plot = 300
x = torch.linspace(0, x_0, nx_plot)
y = torch.linspace(0, y_0, ny_plot)
foil = Airfoil.NACA4(airfoil_name)
camber = int(airfoil_name[0])
camber_pos = int(airfoil_name[1])
thickness = int(airfoil_name[2:])

X, Y = np.meshgrid(x,y)

X_pred = np.hstack((X.flatten()[:,None], Y.flatten()[:,None]))

X_pred = torch.from_numpy(X_pred).float()

cam_array = camber * torch.ones((len(X_pred[:, 0]), 1)).float()
cam_pos_array = camber_pos * torch.ones((len(X_pred[:, 0]), 1)).float()
thickness_array = thickness * torch.ones((len(X_pred[:, 0]), 1)).float()
alfa_array = alfa * torch.ones((len(X_pred[:, 0]), 1)).float()


X_pred = torch.from_numpy(np.hstack((X_pred, cam_array)))
X_pred = torch.from_numpy(np.hstack((X_pred, cam_pos_array)))
X_pred = torch.from_numpy(np.hstack((X_pred, thickness_array)))
X_pred = torch.from_numpy(np.hstack((X_pred, alfa_array))).to(device)

# Deep Learning Solution
u_pred = torch.sqrt(u_net(X_pred)[:,0]**2 + u_net(X_pred)[:,1]**2)
p_pred = u_net(X_pred)[:,-1]

u_numpy = np.reshape(u_pred.detach().cpu().numpy(), (nx_plot,ny_plot))
p_numpy = np.reshape(p_pred.detach().cpu().numpy(), (nx_plot, ny_plot))


points = 200
foil = Airfoil.NACA4(airfoil_name, n_points = points)
data = foil.all_points.T
data[points:,:] = np.flip(data[points:,:],0)
data = np.flip(data,0)
data[:,0] += x_0/4
data[:,1] += y_0/2

```

```

poly1 = patches.Polygon(data, facecolor = 'white', edgecolor = 'k', linewidth = 2)
poly2 = patches.Polygon(data, facecolor = 'white', edgecolor = 'k', linewidth = 2)

plt.ion()
plt.figure(figsize=(4*x_0, 4*y_0))
plt.clf()

plt.subplot(1,2,1)
plt.imshow(np.flip(u_numpy, 0), extent=[0, x_0, 0, y_0])
plt.gca().add_patch(poly1)
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar(orientation='horizontal')
plt.title('Deep Learning Solution - Velocity Magnitude')
#plt.show()
#plt.savefig('poisson_rectangle.jpg')

#plt.figure(figsize=(4*x_0, 4*y_0))
plt.subplot(1,2,2)
plt.imshow(np.flip(p_numpy, 0), extent=[0, x_0, 0, y_0])
plt.gca().add_patch(poly2)
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar(orientation='horizontal')
plt.title('Deep Learning Solution - Pressure')
plt.show()
```

A.2.9. Autoencoder Parameterized PINN for Navier Stokes

```
#### HPC CODE ###############
import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
import random
from airfoils import Airfoil
import matplotlib.pyplot as plt
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

x_0 = 5
y_0 = 3
nxp = 20
nyp = 20
num_variables = 2
max_epochs = 5000000
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 4     # final value 4
num_neurons = 100  # final value 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))
nx = int(nxp * x_0)
ny = int(nyp * y_0)

## Simulation Parameters
valid_airfoil_cambers = ['00','26', '22', '48']
valid_airfoil_thicknesses = ['12', '16', '20', '24']
valid_airfoil_names = []

for valid_airfoil_camber in valid_airfoil_cambers:
```

```python
    for valid_airfoil_thickness in valid_airfoil_thicknesses:
        valid_airfoil_name_val = valid_airfoil_camber + valid_airfoil_thickness
        valid_airfoil_names.append(valid_airfoil_name_val)

airfoil_cambers = ['00','24', '44', '42', '64', '46', '48']
airfoil_thicknesses = ['10', '14', '18', '22']
airfoil_names = []

for airfoil_camber in airfoil_cambers:
    for airfoil_thickness in airfoil_thicknesses:
        airfoil_name_val = airfoil_camber + airfoil_thickness
        airfoil_names.append(airfoil_name_val)

airfoil_name = random.choice(airfoil_names)
alfa = random.choice(['00','04','08'])

airfoil_latent_data =   {'0010': [-0.6596158742904663, -1.52833092212677,
    0.9524492621421814],
 '0012': [-0.7612241506576538, -1.2206069231033325, 0.8008520007133484],
 '0014': [-0.8675788640975952, -0.9109957218170166, 0.6440109610557556],
 '0016': [-0.9889535307884216, -0.5921140313148499, 0.47418004274368286],
 '0018': [-1.105587363243103, -0.28051847219467163, 0.3102298378944397],
 '0020': [-1.2086365222930908, 0.018362203612923622, 0.18606562912464142],
 '0022': [-1.304009199142456, 0.3306569755077362, 0.10021809488534927],
 '0024': [-1.3908438682556152, 0.6727849245071411, 0.026510173454880714],
 '2210': [0.11811462044715881, -1.3414424657821655, 1.1414552927017212],
 '2212': [0.038074791431427, -1.0323399305343628, 1.0091665983200073],
 '2214': [-0.046844109892845154, -0.7240248322486877, 0.8723058104515076],
 '2216': [-0.13896270096302032, -0.4172692596912384, 0.7273876070976257],
 '2218': [-0.24476927518844604, -0.10955093055963516, 0.5754779577255249],
 '2220': [-0.350607693195343, 0.21484485268592834, 0.45308536291122437],
 '2222': [-0.4248036742210388, 0.5542337894439697, 0.37878525257110596],
 '2224': [-0.4812098741531372, 0.9119169116020203, 0.3088846504688263],
 '2410': [0.8199985027313232, -1.1582105159759521, 1.2965266704559326],
 '2412': [0.7554594278335571, -0.8424240350723267, 1.1838085651397705],
 '2414': [0.6957219243049622, -0.5209964513778687, 1.0823055505752563],
 '2416': [0.6317368745803833, -0.21112176775932312, 0.9724076390266418],
 '2418': [0.5705732703208923, 0.08479166030883789, 0.8617393970489502],
 '2420': [0.5197797417640686, 0.39344969391822815, 0.7782306671142578],
 '2422': [0.4537320137023926, 0.7428445816040039, 0.6980561017990112],
 '2424': [0.397599995136261, 1.0913816690444946, 0.6249390244483948],
 '2610': [1.4744316339492798, -1.0117841958999634, 1.4661911725997925],
 '2612': [1.4168353080749512, -0.6963366270065308, 1.3756537437438965],
 '2614': [1.3596229553222656, -0.38328906893730164, 1.2880265712738037],
 '2616': [1.3085203170776367, -0.0814577266573906, 1.2024937868118286],
 '2618': [1.258837342262268, 0.2248709797859192, 1.1287785768508911],
 '2620': [1.2078857421875, 0.5453170537948608, 1.063645601272583],
 '2622': [1.1824116706848145, 0.8681461811065674, 0.9859014749526978],
 '2624': [1.1369116306304932, 1.2177963256835938, 0.9152663350105286],
 '4410': [0.5521025657653809, -1.416182041168213, 0.5152968764305115],
 '4412': [0.4818824529647827, -1.1100090742111206, 0.38052666187286377],
 '4414': [0.41211169958114624, -0.8039820194244385, 0.2461097538471222],
 '4416': [0.3441205620765686, -0.4986405670642853, 0.11174564808607101],
 '4418': [0.278931200504303, -0.2064802646636963, -0.013988763093948364],
 '4420': [0.21971362829208374, 0.06446453928947449, -0.12193141132593155],
 '4422': [0.16691040992736816, 0.37098559737205505, -0.19668009877204895],
 '4424': [0.11524598300457001, 0.7092867493629456, -0.2573535144329071],
 '4210': [-0.030042901635169983, -1.470795750617981, 0.7521668076515198],
 '4212': [-0.11681389808654785, -1.165541648864746, 0.6092405319213867],
 '4214': [-0.20626066625118256, -0.8594645261764526, 0.4649921655654907],
 '4216': [-0.3003275990486145, -0.5506353974342346, 0.3178010880947113],
 '4218': [-0.4089584946632385, -0.23580028116703033, 0.16623751819133759],
 '4220': [-0.5053760409355164, 0.04962416738271713, 0.04146712273359299],
 '4222': [-0.575164258480072, 0.3692358136177063, -0.03879170119762421],
 '4224': [-0.6272857189178467, 0.7224670052528381, -0.11474796384572983],
 '6410': [0.2077949047088623, -1.6443651914596558, -0.18043068051338196],
 '6412': [0.12505307793617249, -1.3446606397628784, -0.3285021483898163],
 '6414': [0.04139922186732292, -1.0458341836929321, -0.4760527014732361],
 '6416': [-0.042650021612644196, -0.7489229440689087, -0.6232290267944336],
 '6418': [-0.11326076090335846, -0.4711480438709259, -0.7427667379379272],
 '6420': [-0.16893139481544495, -0.22356154024600983, -0.8342057466506958],
 '6422': [-0.23072992265224457, 0.04906310886144638, -0.9354248046875],
 '6424': [-0.2911964356899 2615, 0.36464574933052063, -1.020395040512085],
 '4610': [1.060377836227417, -1.4079362154006958, 0.2931698262691498],
 '4612': [0.9925789833068848, -1.110345482826233, 0.16980434954166412],
 '4614': [0.9277621507644653, -0.8131468892097473, 0.05033652111887932],
 '4616': [0.8681806325912476, -0.517879843711853, -0.0674424022436142],
 '4618': [0.8196607828140259, -0.22987797856330872, -0.16609537601470947],
 '4620': [0.7808578014373779, 0.04395593702793121, -0.2566536068916321],
 '4622': [0.7552579641342163, 0.334025114774704, -0.3284453749656677],
 '4624': [0.7265974283218384, 0.6591168642044067, -0.385637104511261],
 '4810': [1.5547513961791992, -1.440292239189148, 0.10100625455379486],
 '4812': [1.498142123222351, -1.1450642347335815, -0.01623530685901642],
 '4814': [1.4510027170181274, -0.8445427417755127, -0.12679888308048248],
 '4816': [1.4071964025497437, -0.5509340763092041, -0.23745934665203094],
 '4818': [1.3673973083496094, -0.26776471734046936, -0.3373967409133911],
 '4820': [1.3424327373504639, 0.007963575422763824, -0.4251900017261505],
 '4822': [1.326838254928588 9, 0.29495349526405334, -0.4953930377960205],
 '4824': [1.314286231994629, 0.6019899845123291, -0.5496833324432373]}

mesh_file_name = '/home/nfs/skakkar/jupyter_codes/Database/' + 'NS_mesh_data_BC_53_AOA_' +
    alfa + '_' + airfoil_name + '_highRe.npy'
sol_file_name = '/home/nfs/skakkar/jupyter_codes/Database/' + 'NS_solution_data_BC_53_AOA_' +
     alfa + '_' + airfoil_name + '_highRe.npy'

alfa = int(alfa)

u_stream = 1 # m/s
learning_rate = 0.0001   # final value starting 0.0001
data_lamda = 0.99  # final value 0.8
pres_lamda = 0.5  # default 0.8
chord = 1 # m
p_outlet = 1 # scaled between 0-1
kin_viscosity = 1.5e-3 # m^2/s
Re = chord * u_stream / kin_viscosity


## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
```

```python
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)

#FILE = 'airfoil_lowRe_31_stokes_myPC.pth'
#u_net.load_state_dict(torch.load(FILE))
#u_net.eval()

## Generating Data Points
def interior_points(nx, ny, x_0, y_0, airfoil_name, alfa):

    foil = Airfoil.NACA4(airfoil_name)
    z1 = float(airfoil_latent_data[airfoil_name][0])
    z2 = float(airfoil_latent_data[airfoil_name][1])
    z3 = float(airfoil_latent_data[airfoil_name][2])

    xy = np.random.rand(nx * ny, num_variables)
    xy[:, 0] *= x_0
    xy[:, 1] *= y_0

    indices = []

    for i in np.arange(len(xy[:,0])):
        if xy[i,0]>=x_0/4:
            if xy[i,0]<=x_0/4+1:
                if xy[i,1]>=foil.y_lower(xy[i,0]-x_0/4)+y_0/2:
                    if xy[i,1]<=foil.y_upper(xy[i,0]-x_0/4)+y_0/2:
                        indices.append(i)

    xy = np.delete(xy, indices, axis = 0)

    xy = torch.from_numpy(xy).float()

    z1_array = z1 * torch.ones((len(xy[:, 0]), 1)).float()
    z2_array = z2 * torch.ones((len(xy[:, 0]), 1)).float()
    z3_array = z3 * torch.ones((len(xy[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy[:, 0]), 1)).float()

    xy = torch.from_numpy(np.hstack((xy, z1_array)))
    xy = torch.from_numpy(np.hstack((xy, z2_array)))
    xy = torch.from_numpy(np.hstack((xy, z3_array)))
    xy = torch.from_numpy(np.hstack((xy, alfa_array))).to(device)

    return xy

def mesh_points(file_name, airfoil_name, alfa):
    foil = Airfoil.NACA4(airfoil_name)
    z1 = float(airfoil_latent_data[airfoil_name][0])
    z2 = float(airfoil_latent_data[airfoil_name][1])
    z3 = float(airfoil_latent_data[airfoil_name][2])

    xy_data = np.load(file_name)

    z1_array = z1 * torch.ones((len(xy_data[:, 0]), 1)).float()
    z2_array = z2 * torch.ones((len(xy_data[:, 0]), 1)).float()
    z3_array = z3 * torch.ones((len(xy_data[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_data[:, 0]), 1)).float()

    xy_data = torch.from_numpy(np.hstack((xy_data, z1_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, z2_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, z3_array)))
    xy_data = torch.from_numpy(np.hstack((xy_data, alfa_array))).to(device)

    return xy_data

# Mesh Generation
xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa).float()
xy_data = mesh_points(mesh_file_name, airfoil_name, alfa).float()
uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)


## PINN Formulation
```

```python
optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)

def interior_loss(XY, Re): # remember order of three outputs (u,v,p)
    x = torch.tensor(XY[:, 0], requires_grad=True).float().cpu()
    y = torch.tensor(XY[:, 1], requires_grad=True).float().cpu()
    z1 = np.asscalar(XY[0, 2].cpu().numpy())
    z2 = np.asscalar(XY[0, 3].cpu().numpy())
    z3 = np.asscalar(XY[0, 4].cpu().numpy())
    alfa = np.asscalar(XY[0, 5].cpu().numpy())

    xy_pde = torch.hstack((torch.reshape(x, (-1, 1)), torch.reshape(y, (-1, 1)))).float()
    z1_array = z1 * torch.ones((len(xy_pde[:, 0]), 1)).float()
    z2_array = z2 * torch.ones((len(xy_pde[:, 1]), 1)).float()
    z3_array = z3 * torch.ones((len(xy_pde[:, 1]), 1)).float()
    alfa_array = alfa * torch.ones((len(xy_pde[:, 1]), 1)).float()

    xy_pde = torch.hstack((xy_pde, z1_array))
    xy_pde = torch.hstack((xy_pde, z2_array))
    xy_pde = torch.hstack((xy_pde, z3_array))
    xy_pde = torch.hstack((xy_pde, alfa_array)).to(device)

    u = u_net(xy_pde)[:,0].to(device)
    v = u_net(xy_pde)[:,1].to(device)
    p = u_net(xy_pde)[:,2].to(device)
    x.to(device)
    y.to(device)

    u_x = torch.autograd.grad(
        u, x,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_y = torch.autograd.grad(
        u, y,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    u_yy = torch.autograd.grad(
        u_y, y,
        grad_outputs=torch.ones_like(u_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    v_x = torch.autograd.grad(
        v, x,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_xx = torch.autograd.grad(
        v_x, x,
        grad_outputs=torch.ones_like(v_x),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_y = torch.autograd.grad(
        v, y,
        grad_outputs=torch.ones_like(v),
        retain_graph=True,
        create_graph=True
    )[0].to(device)
    v_yy = torch.autograd.grad(
        v_y, y,
        grad_outputs=torch.ones_like(v_y),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_x = torch.autograd.grad(
        p, x,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    p_y = torch.autograd.grad(
        p, y,
        grad_outputs=torch.ones_like(p),
        retain_graph=True,
        create_graph=True
    )[0].to(device)

    #f1 = p_xx + p_yy
    f1 = u_x + v_y
    f2 = u_xx/(Re) + u_yy/(Re) - p_x - u*u_x - v*u_y
    f3 = v_xx/(Re) + v_yy/(Re) - p_y - u*v_x - v*v_y

    loss1 = torch.mean(f1 ** 2)
    loss2 = torch.mean(f2 ** 2)
    loss3 = torch.mean(f3 ** 2)

    return (loss1 + loss2 + loss3)

def loss_func(XY, data_weight,pres_weight, Re, XY_data, UVP_data):


    data_supervised_vel = u_net(XY_data)[:,0:2] - UVP_data[:,0:2]
    data_supervised_pres = u_net(XY_data)[:,2] - UVP_data[:,2]


    loss_val   = (1-data_weight)*interior_loss(XY, Re) + (data_weight)*(pres_weight*torch.
        mean(data_supervised_pres**2) +\
                                                    (1-pres_weight)*torch.mean(
                                                        data_supervised_vel**2))

    return loss_val

epoch = 0
loss = loss_func(xy, data_lamda,pres_lamda, Re, xy_data, uvp_data)

## Training Loop
while epoch <= max_epochs and loss.item() > 1e-6:
    # Perturbation
    if (epoch % 5 == 0) and (epoch != 0):

        airfoil_name = random.choice(airfoil_names)
        alfa = random.choice(['00','04','08'])

        mesh_file_name = '/home/nfs/skakkar/jupyter_codes/Database/' + '
            NS_mesh_data_BC_53_AOA_' + alfa + '_' + airfoil_name + '_highRe.npy'
        sol_file_name = '/home/nfs/skakkar/jupyter_codes/Database/' + '
            NS_solution_data_BC_53_AOA_' + alfa + '_' + airfoil_name + '_highRe.npy'

        alfa = int(alfa)

        xy = interior_points(nx, ny, x_0, y_0, airfoil_name, alfa).float()
        xy_data = mesh_points(mesh_file_name, airfoil_name, alfa).float()
        uvp_data = torch.from_numpy(np.load(sol_file_name)).to(device)

    if (epoch % 1000000 == 0) and (epoch != 0):
        learning_rate = learning_rate/10
        optimizer = torch.optim.Adam(u_net.parameters(), lr=learning_rate)


    # forward and loss
    loss = loss_func(xy, data_lamda,pres_lamda, Re , xy_data, uvp_data)

    # backward
    loss.backward()

    # update
    optimizer.step()

    if (epoch % 100 == 0) and (epoch != 0):
        valid_airfoil_name = random.choice(valid_airfoil_names)
        valid_alfa = random.choice(['03', '05', '07'])
        valid_mesh_file_name = '/home/nfs/skakkar/jupyter_codes/Validation_Database/' + '
            NS_mesh_data_BC_53_AOA_' + valid_alfa + '_' + valid_airfoil_name + '_highRe.npy'
        valid_sol_file_name = '/home/nfs/skakkar/jupyter_codes/Validation_Database/' + '
            NS_solution_data_BC_53_AOA_' + valid_alfa + '_' + valid_airfoil_name + '_highRe.
            npy'
        valid_alfa = int(valid_alfa)
        valid_xy = interior_points(nx, ny, x_0, y_0, valid_airfoil_name, valid_alfa).float()
        valid_xy_data = mesh_points(valid_mesh_file_name, valid_airfoil_name, valid_alfa).
            float()
        valid_uvp_data = torch.from_numpy(np.load(valid_sol_file_name)).to(device)
        valid_loss = loss_func(valid_xy, data_lamda,pres_lamda, Re , valid_xy_data,
            valid_uvp_data)

        print(f'epoch: {epoch}, loss: {loss.item()}, airfoil = {airfoil_name}, valid_loss = {
            valid_loss}, valid_airfoil = {valid_airfoil_name} ,lr = {learning_rate}', flush =
             True)

    optimizer.zero_grad()
    epoch += 1

FILE = '/home/nfs/skakkar/model_data/
    airfoil_navier_stokes_fully_final_cambered_autoencoder_accurate_wt_99_53_10141822.pth'
torch.save(u_net.state_dict(), FILE)
```

A.2.10. Coupled Autoencoder and PINN Post Processing Script for Latent Space Defined Geometries

```
import pyvista as pv
import numpy as np
from tqdm import tqdm
from ds_models_tf.sk_api.models.structured.ae.sk_sae_pca import SKSAEPCA
import matplotlib.pyplot as plt
import ipywidgets as wgt

from ds_models_tf.sk_api.models.structured.ae.sk_sae import SKSAE
from ds_models_tf.sk_api.models.structured.ae.sk_sae_pca import SKSAEPCA
from ds_models_tf.sk_api.models.structured.ae.sk_wsae_pca import SKWSAEPCA

from airfoils import Airfoil

import joblib

import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
import random
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

#FILE_AE = 'naca_ae_sym.joblib'
```

```python
FILE_AE = 'naca_ae_cambered.joblib'
model = joblib.load(FILE_AE)

x_0 = 5
y_0 = 3

num_variables = 2
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 4
num_neurons = 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))

def airfoil_decoder(airfoil_name, model):
    foil = Airfoil.NACA4(airfoil_name, n_points=200)
    data = foil.all_points.T

    z_value = 0 #np.linspace(0,1,5)
    data = np.vstack((
    np.hstack((data,z_value*np.ones((len(data),1))))))

    original_mesh = pv.PolyData(data)
    latent_space_vals = model.encoder_predict([original_mesh])[0]

    return latent_space_vals

airfoil_name = '0012'
airfoil_latent_vals = airfoil_decoder(airfoil_name, model)
#z1 = float(airfoil_latent_vals[0]); z2 = float(airfoil_latent_vals[1]); z3 = float(
    airfoil_latent_vals[2])

z1_slider = wgt.FloatSlider(value = 0, min=-2.5, max=2.5, step=1e-4, description='z1',
    disabled=False,continuous_update=True,
                           orientation = 'horizontal', readout = True, readout_format = '.4f'
                               ,)
z2_slider = wgt.FloatSlider(value = 0, min=-2.5, max=2.5, step=1e-4, description='z2',
    disabled=False,continuous_update=True,
                           orientation = 'horizontal', readout = True, readout_format = '.4f'
                               ,)
z3_slider = wgt.FloatSlider(value = 0, min=-2.5, max=2.5, step=1e-4, description='z3',
    disabled=False,continuous_update=True,
                           orientation = 'horizontal', readout = True, readout_format = '.4f'
                               ,)
alfa_slider = wgt.FloatSlider(value = 4, min=0, max=10, step=0.1, description='alfa',disabled
    =False,continuous_update=True,
                           orientation = 'horizontal', readout = True, readout_format = '.1f'
                               ,)

#z1 = z1_slider; z2 = z2_slider ;z3 = z3_slider

alfa = alfa_slider # degrees
chord = 1 # m

## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)


FILE = 'airfoil_navier_stokes_fully_final_cambered_autoencoder_accurate_53_10141822.pth'

u_net.load_state_dict(torch.load(FILE))
u_net.eval()


# Plotting

def plotter(z1, z2, z3, alfa):
    nx_plot = 300
    ny_plot = 300
    x = torch.linspace(0, x_0, nx_plot)
    y = torch.linspace(0, y_0, ny_plot)
    #foil = Airfoil.NACA4(airfoil_name)

    X, Y = np.meshgrid(x,y)

    X_pred = np.hstack((X.flatten()[:,None], Y.flatten()[:,None]))

    X_pred = torch.from_numpy(X_pred).float()

    z1_array = z1 * torch.ones((len(X_pred[:, 0]), 1)).float()
    z2_array = z2 * torch.ones((len(X_pred[:, 0]), 1)).float()
    z3_array = z3 * torch.ones((len(X_pred[:, 0]), 1)).float()
    alfa_array = alfa * torch.ones((len(X_pred[:, 0]), 1)).float()


    X_pred = torch.from_numpy(np.hstack((X_pred, z1_array)))
    X_pred = torch.from_numpy(np.hstack((X_pred, z2_array)))
    X_pred = torch.from_numpy(np.hstack((X_pred, z3_array)))
    X_pred = torch.from_numpy(np.hstack((X_pred, alfa_array))).to(device)

    # Deep Learning Solution
    u_pred = torch.sqrt(u_net(X_pred)[:,0]**2 + u_net(X_pred)[:,1]**2)
    p_pred = u_net(X_pred)[:,-1]

    u_numpy = np.reshape(u_pred.detach().cpu().numpy(), (nx_plot,ny_plot))
    p_numpy = np.reshape(p_pred.detach().cpu().numpy(), (nx_plot, ny_plot))


    points = 200
    #foil = Airfoil.NACA4(airfoil_name, n_points = points)
    data = model.decoder_predict([[z1, z2, z3]])[0][:,0:2]
    data[points:,:] = np.flip(data[points:,:],0)
    data = np.flip(data,0)
    data[:,0] += x_0/4
    data[:,1] += y_0/2


    poly1 = patches.Polygon(data, facecolor = 'white', edgecolor = 'k', linewidth = 2)
    poly2 = patches.Polygon(data, facecolor = 'white', edgecolor = 'k', linewidth = 2)

    plt.ion()
    plt.figure(figsize=(4*x_0, 4*y_0))
    plt.clf()

    plt.subplot(1,2,1)
    plt.imshow(np.flip(u_numpy, 0), extent=[0, x_0, 0, y_0])
    plt.gca().add_patch(poly1)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.colorbar(orientation='horizontal')
    plt.title('Deep Learning Solution - Velocity Magnitude')
    #plt.show()
    #plt.savefig('poisson_rectangle.jpg')

    #plt.figure(figsize=(4*x_0, 4*y_0))
    plt.subplot(1,2,2)
    plt.imshow(np.flip(p_numpy, 0), extent=[0, x_0, 0, y_0])
    plt.gca().add_patch(poly2)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.colorbar(orientation='horizontal')
    plt.title('Deep Learning Solution - Pressure')
    plt.show()

    return

wgt.interact(plotter, z1 = z1_slider, z2 = z2_slider, z3 = z3_slider, alfa = alfa_slider)
```

A.2.11. Coupled Autoencoder and PINN Post Processing Script for NACA Geometries

```
import pyvista as pv
import numpy as np
from tqdm import tqdm
from ds_models_tf.sk_api.models.structured.ae.sk_sae_pca import SKSAEPCA
import matplotlib.pyplot as plt
import ipywidgets as wgt

from ds_models_tf.sk_api.models.structured.ae.sk_sae import SKSAE
from ds_models_tf.sk_api.models.structured.ae.sk_sae_pca import SKSAEPCA
from ds_models_tf.sk_api.models.structured.ae.sk_wsae_pca import SKWSAEPCA

from airfoils import Airfoil

import joblib

import torch
from torch import pi, sin, cos
import numpy as np
from collections import OrderedDict
import random
from matplotlib import patches

## CUDA support
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

#FILE_AE = 'naca_ae_sym.joblib'
FILE_AE = 'naca_ae_cambered.joblib'
model = joblib.load(FILE_AE)

```

```python
x_0 = 5
y_0 = 3

points = 200
num_variables = 2
num_variables = 2
num_inputs = 6 # x,y,alfa,airfoil name (3)
num_outputs = 3
num_layers = 4
num_neurons = 100
u_array = np.ones(num_layers + 2) * num_neurons
u_array[0] = num_inputs
u_array[-1] = num_outputs
u_array = np.ndarray.tolist(u_array.astype(int))

def airfoil_decoder(airfoil_name, model):
    foil = Airfoil.NACA4(airfoil_name, n_points=200)
    data = foil.all_points.T

    z_value = 0 #np.linspace(0,1,5)
    data = np.vstack((
    np.hstack((data,z_value*np.ones((len(data),1))))))

    original_mesh = pv.PolyData(data)
    latent_space_vals = model.encoder_predict([original_mesh])[0]

    return latent_space_vals

airfoil_name = '4812'
airfoil_latent_vals = airfoil_decoder(airfoil_name, model)
z1 = float(airfoil_latent_vals[0]); z2 = float(airfoil_latent_vals[1]); z3 = float(
    airfoil_latent_vals[2])

#z1 = z1_slider; z2 = z2_slider ;z3 = z3_slider

alfa = 5 # degrees
chord = 1 # m

## Neural Network
class neural_net(torch.nn.Module):
    def __init__(self, layers):
        super(neural_net, self).__init__()

        # parameters
        self.depth = len(layers) - 1

        # set up layer order dict
        self.activation = torch.nn.Tanh

        layer_list = list()
        for i in range(self.depth - 1):
            layer_list.append(
                ('layer_%d' % i, torch.nn.Linear(layers[i], layers[i + 1]))
            )
            layer_list.append(('activation_%d' % i, self.activation()))

        layer_list.append(
            ('layer_%d' % (self.depth - 1), torch.nn.Linear(layers[-2], layers[-1]))
        )
        layerDict = OrderedDict(layer_list)

        # deploy layers
        self.layers = torch.nn.Sequential(layerDict)

    def forward(self, x):
        out = self.layers(x)
        return out


u_net = neural_net(u_array).to(device)

```

```python
if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
    model = torch.nn.DataParallel(u_net)


FILE = 'airfoil_navier_stokes_fully_final_cambered_autoencoder_accurate_53_10141822.pth'

u_net.load_state_dict(torch.load(FILE))
u_net.eval()


# Plotting


nx_plot = 300
ny_plot = 300
x = torch.linspace(0, x_0, nx_plot)
y = torch.linspace(0, y_0, ny_plot)
#foil = Airfoil.NACA4(airfoil_name)

X, Y = np.meshgrid(x,y)

X_pred = np.hstack((X.flatten()[:,None], Y.flatten()[:,None]))

X_pred = torch.from_numpy(X_pred).float()

z1_array = z1 * torch.ones((len(X_pred[:, 0]), 1)).float()
z2_array = z2 * torch.ones((len(X_pred[:, 0]), 1)).float()
z3_array = z3 * torch.ones((len(X_pred[:, 0]), 1)).float()
alfa_array = alfa * torch.ones((len(X_pred[:, 0]), 1)).float()


X_pred = torch.from_numpy(np.hstack((X_pred, z1_array)))
X_pred = torch.from_numpy(np.hstack((X_pred, z2_array)))
X_pred = torch.from_numpy(np.hstack((X_pred, z3_array)))
X_pred = torch.from_numpy(np.hstack((X_pred, alfa_array))).to(device)

# Deep Learning Solution
u_pred = torch.sqrt(u_net(X_pred)[:,0]**2 + u_net(X_pred)[:,1]**2)
p_pred = u_net(X_pred)[:,-1]

u_numpy = np.reshape(u_pred.detach().cpu().numpy(), (nx_plot,ny_plot))
p_numpy = np.reshape(p_pred.detach().cpu().numpy(), (nx_plot, ny_plot))


data = model.decoder_predict([[z1, z2, z3]])[0][:,0:2]
data[points:,:] = np.flip(data[points:,:],0)
data = np.flip(data,0)
data[:,0] += x_0/4
data[:,1] += y_0/2


poly1 = patches.Polygon(data, facecolor = 'white', edgecolor = 'k', linewidth = 2)
poly2 = patches.Polygon(data, facecolor = 'white', edgecolor = 'k', linewidth = 2)

plt.ion()
plt.figure(figsize=(4*x_0, 4*y_0))
plt.clf()

plt.subplot(121)
plt.imshow(np.flip(u_numpy, 0), extent=[0, x_0, 0, y_0])
plt.gca().add_patch(poly1)
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar(orientation='horizontal')
plt.title('Deep Learning Solution - Velocity Magnitude')
#plt.show()
#plt.savefig('poisson_rectangle.jpg')

#plt.figure(figsize=(4*x_0, 4*y_0))
```

```
plt.subplot(122)
plt.imshow(np.flip(p_numpy, 0), extent=[0, x_0, 0, y_0])
plt.gca().add_patch(poly2)
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar(orientation='horizontal')
plt.title('Deep Learning Solution - Pressure')
plt.show()
```