Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Physics Informed Neural Networks in Fluid Dynamics

Naveen Kumar Subramanian

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Physics Informed Neural Networks in Fluid Dynamics

Author:	Naveen Kumar Subramanian
1st examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Assistant advisor:	Dr. Felix Dietrich, Dr.-Ing. Henning Sauerland (Hitachi)
Submission Date:	15 June 2021

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

15 June 2021 Naveen Kumar Subramanian

Acknowledgments

This Thesis has been a part of my studies of my master degree at the Technical University of Munich. The thesis work was done at Hitachi Europe GmbH.

In this context I want to thank Univ.-Prof. Dr. Hans-Joachim Bungartz from Scientific Computing chair as the first academic Supervisor at Technical University of Munich, as well as Dr.-Ing. Henning Sauerland as my advisor at Hitachi who gave a opportunity to work in this specific topic. Especially, I am very thankful to Dr. Felix Dietrich for being a continuous support and guidance throughout my thesis work.

Special acknowledgments are owed to my program coordinator's at Technical University of Munich for helping me through the organisation aspects of my thesis work and submission. Finally, I would like to thank my family for always supporting me during the time of this dissertation.

Abstract

Deep learning has a to lot of applications; however, its use in solving partial differential equations (PDEs) has been a trending topic recently. Recent works have shown that neural networks can be used to solve partial differential equations, which introduce us to a method called physics informed neural networks (PINNs).The purpose of this project is to have a clear understanding of these methods. Then we study about the tool developed in recent times to implement PINNs known as SimNet by NVIDIA. We elaborate the usage and customizability of SimNet and systematically study the concept of PINNs applied to Navier-Stokes equations as compared to conventional CFD methods such as Finite Volume Method (FVM). The sensitivity of PINNs with respect to parameter variations are studied based on the flow around the cylinder in the steady state regime. From the parameters studies, best parameters are chosen to demonstrate the application of SimNet for a parametrized simulation problem.

Contents

1 Introduction

1.1 Introduction

Advances in computing power and rapid growth of available data in recent years have revived the field of artificial intelligence and deep learning . In particular, deep learning methods have achieved exceptional results in a wide range of problems, including image recognition, natural language processing and reinforcement learning [1] [2] [3]. The most notable architecture within deep learning is the deep neural networks. The potential of deep learning methods stems from the approximation capabilities of deep neural networks, and the ability of training algorithms to find network parameters with which accurate approximations are achieved.

Typical applications of neural networks employ these networks to recover functions that are not directly available to the user. For example in image recognition, functions that map images to the corresponding classes are vastly complex and high dimensional functions, which convolutional neural networks are able to accurately recover [1]. Other problems where one is interested in finding unknown functions are given by partial differential equations. For more difficult partial differential equations, analytical solutions are typically unavailable, and one has to resort to numerically approximating the solution. Many state of the art numerical solvers divide the geometry of the problem into a mesh of points or simple geometric elements, and proceed to compute an approximation of the solution on this set of points or using the basis functions on these finite elements.

Neural networks can also be utilized as a kind of basis function. The study of [4] shows that there is a strong connection between finite element methods and neural networks with rectified linear units, as only two hidden layers are required for such neural networks to be able to express any output of a finite element method.

Particularly, the study of [5] motivates the use of neural networks in this context by stating that neural networks are closed form expressions, and thus provide information about the approximation anywhere in the relevant domain. This information includes derivatives of the solution, as neural networks with the appropriate activation functions are differentiable functions. Furthermore, they state that training neural networks is a highly parallelizable process. So deep neural networks may be the key to solving high dimensional partial differential equations since these methods are meshfree in contrast to most conventional numerical approaches.

1.2 Motivation of the thesis

Although most works [6] [7] that utilize deep neural networks to solve partial differential equations are similar in nature, these methods are still not well understood. The main goal of this thesis is to gain a deeper understanding of PINNs, and gain knowledge to implement the methods in NVIDIA's SimNet tool. In particular, we aim to understand how the approximation properties of PINNs can be influenced and how the obtained solution compares with traditional numerical approaches.

This thesis is structured as follows. In Section 2.1, we discuss the background theory that is required to understand the neural networks and basic definitions of terms used in deep neural networks. Then Section 2.2 and 2.3 covers the literature study about PINNs and the implementation of PINNs based on the SimNet tool. The aim is to approximate the solution of Navier Stokes equations for the case of flow around a cylinder in two spatial dimensions. We discuss optimization of hyperparameters present in the proposed methods, and some general properties of PINNs. In Section 3.1 and 3.2 ,we validate the results with conventional Computational Fluid Dynamics (CFD) methods like finite volume method (FVM) and elaborate computational aspects of PINNs and its advantages over conventional CFD methods. The final chapter concludes by giving a brief summary and discussion of the results, and some recommendations for further research.

2 State of the Art

2.1 Basics of Neural Networks

2.1.1 Architecture

Neural networks define complicated functions that are parametrized by the weights of the connections between neurons, and the biases of these neurons [8]. Let us denote these weights by W and the biases by b, and the joint set containing all parameters by $\theta = \{W, b\}$. Each of the neurons computes the weighted average of their input neurons and the corresponding weights, adds its bias to the result, and feeds the resulting value through a nonlinear activation function, which we label α. Let us denote the inputs by the vector x. With this notation, a neural network defines a function $f(x;\theta)$, where f has the same dimension as the number of neurons in the output layer, and may thus be vector-valued.

The simplest configuration of deep neural networks is the multilayer perceptron (MLP). For such neural networks, all neurons are arranged in ordered layers, and each neuron is connected to all neurons in neighbouring layers as shown in Figure 2.1. Under this configuration, it makes sense to label the outputs of the parameters belonging to each layer separately. Using vector notation, we denote the output of the i^{th} layer by f_i and the biases by b_i. We denote the weight connecting the $(i-1)^{th}$ layer to the i^{th} layer by the matrix W_i. The input layer is assigned the index 0, and the output layer is assigned the index k. Then, the output of the i^{th}layer is given by:

$$f_i(x) = \begin{cases} x & i\text{=0} \\ \sigma(W_i f_{i-1} x + b_i) & i\text{= 1 ,2,....}k\text{-1} \\ W_i f_{i-1}(x) + b_i & i\text{= } k \end{cases} \tag{2.1}$$

where σ is the vector containing the outputs of the activation function α applied element wise to the argument vector. The output of the network is thus given by $f(x;\theta) = f_k(x)$.

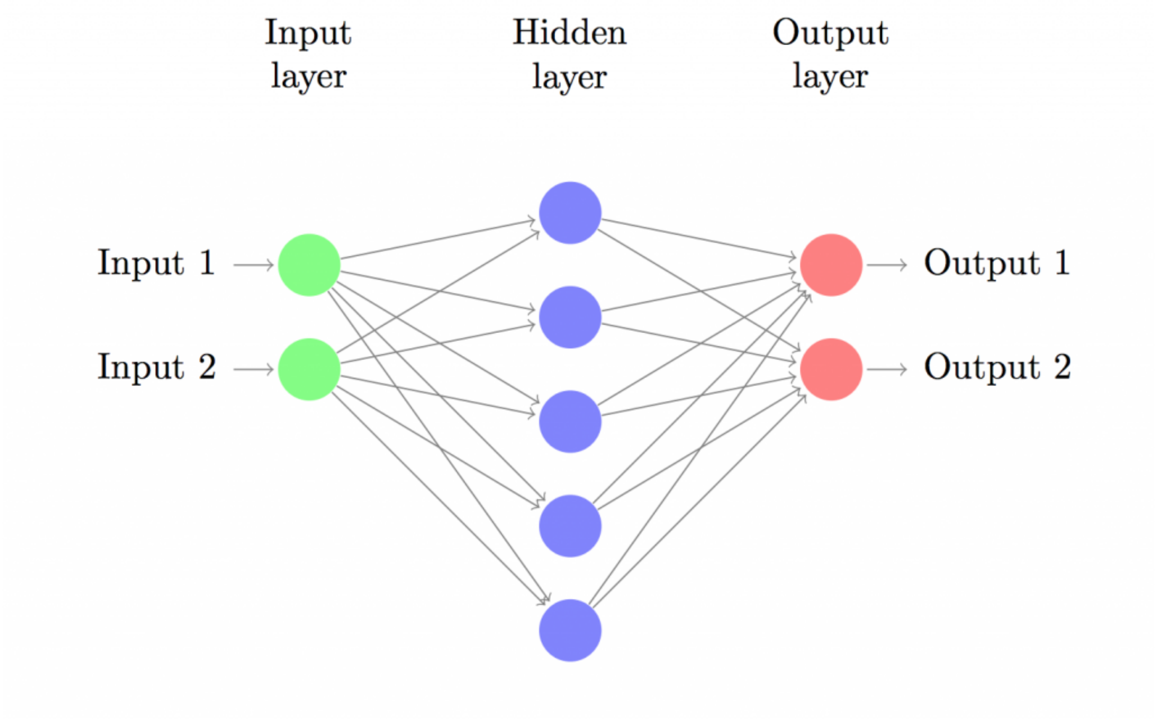


Figure 2.1: Graphical representation of feed forward Neural Networks [8]

It is of critical importance that these activation functions are nonlinear, since otherwise neural networks would only be able to describe linear functions, regardless of the configuration [9]. Common choices of activations functions are sigmoid functions such as the logistic function, the hyperbolic tangent and rectified linear unit (ReLU) is another popular choice.

2.1.2 Automatic Differentiation

There are four ways for computing the derivatives [10]:(1) hand-written analytical way to compute derivatives. (2) numerical methods like finite difference methods (3) symbolic differentiation. and (4) automatic differentiation (AD, also called algorithmic differentiation). In deep learning, the derivatives are evaluated using back propagation [10], a specialized technique of AD. We know neural network represent a compositional function, so AD makes use of the chain rule to compute the gradients or derivatives. AD can be performed in couple of steps , in which first step involves a forward pass to calculate the values of all variables and second one uses backward pass to calculate the gradients or derivatives. We will represent it in Figure 2.2.

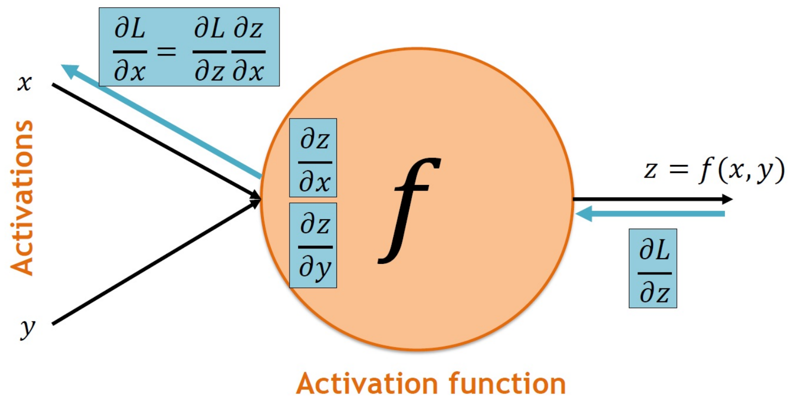

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
x
Activations
y
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
$z = f(x,y)$
$\frac{\partial L}{\partial z}$
Activation function

Figure 2.2: Overview of Forward and Backward pass in back propagation

2.1.3 Activation Functions

In the Section 1.3.1 we defined a vector containing the outputs of the activation function σ applied element wise to the argument vector. They are mostly non linear functions and we will see a brief explanation of some commonly used activation functions in the following section

ReLU

The default choice for an activation function in modern neural networks is the ReLU function implemented through the maximum function and the plot is in Figure 2.3.

$$g(z) = max(0, z) \tag{2.2}$$

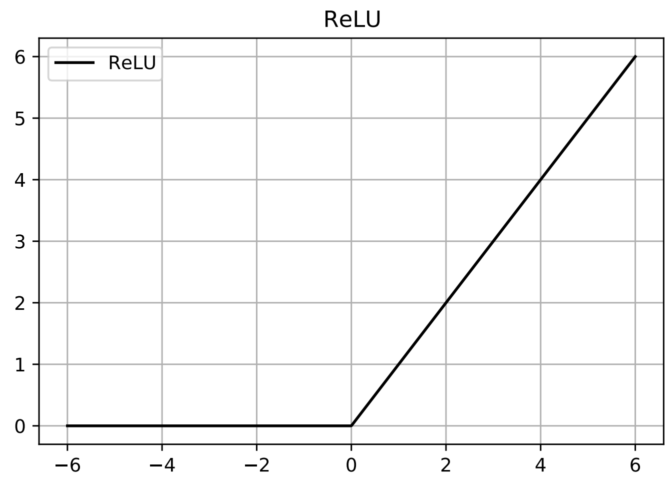


Figure 2.3: Rectified linear Unit Function

Parameters of the neural networks that use RELUs are optimized since the derivative is either 0 or a positive constant value through the domain. One drawback to ReLUs is that the parameters cannot learn via gradient-based methods on examples for which the activation is zero [9].

Sigmoid

The sigmoid function is used to represent a probability distribution over a binary variable and plotted in Figure 2.4. It is defined as:

$$\sigma(z) = \frac{1}{1 + exp(-z)} \tag{2.3}$$

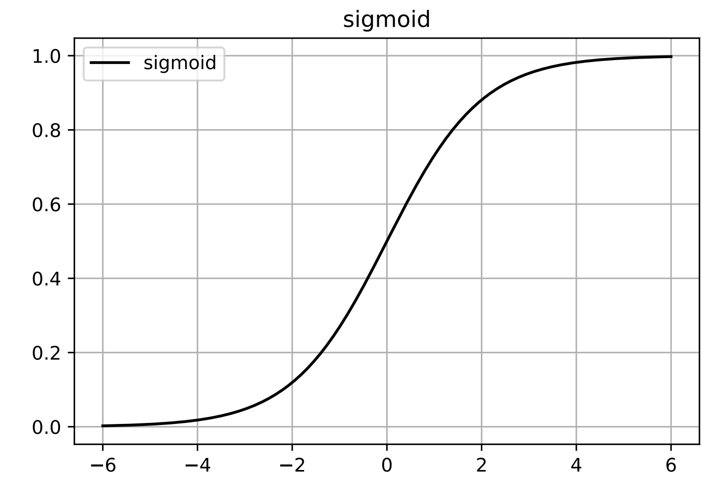


Figure 2.4: Sigmoid Function

Hyperbolic tan Function

The tanh non-linearity is shown on the Figure below. It squeezes a real-valued number to the range [-1, 1]. Very much like the sigmoid , the activations of tanh saturate, but its output is zero-centered. Hence, most of the times the tanh is chosen ahead of sigmoid due to non linearity [9].

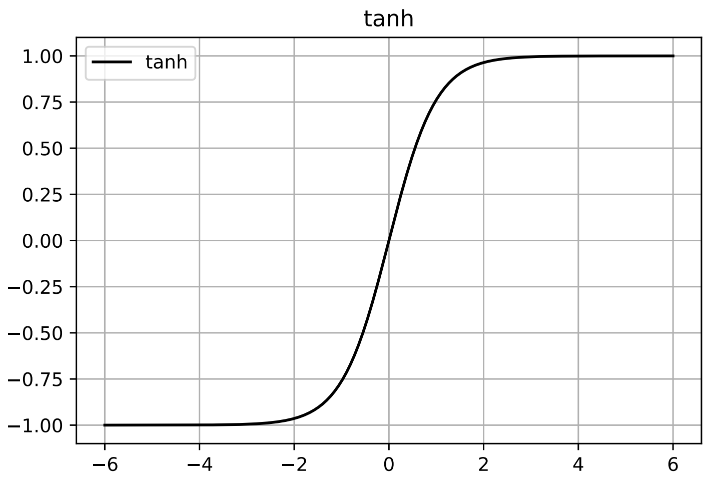


Figure 2.5: tanh Function

2.1.4 Loss Functions

Although neural networks have very nice properties in theory, it is not straightforward to make use of these in practice [9]. It has proven to be very challenging to find sets of weights with which accurate approximations are achieved. There seem to be no direct ways of finding such weights. However, by constructing a kind of distance measure between the parametrized function that a neural network defines and the function one aims to approximate, the problem of finding weights that are good for the neural networks can be modeled or written as a minimization problem. Such distance measures are generally referred to as loss functions. Loss functions do not need to be true distance measures; the only thing that matters is that their minima correspond to accurate approximations [8].

In most cases, parametric models define a distribution. Let $p_{model}(x;\theta)$ be a family of probability distributions over the same space indexed by θ that maps any configuration x to a real number that estimates the true probability $p_{data}(x)$. The maximum likelihood θ_{ML} is then defined as

$$\theta_{ML} = \text{argmax} p_{\text{model}}(\chi : \theta). \tag{2.4}$$

Other popular loss functions are quadratic functions, such as the mean-square-error (MSE) and the root-mean-square-error (RMSE). The Mean Square Error(MSE) is defined as :

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N}(f(x_i,\theta) - y_i)^2. \tag{2.5}$$

Minimizing such a loss function is called training in the context of neural networks. There are lot of training algorithms and we will discuss about some of them in the next section.

2.1.5 Training Algorithm

A neural network is trained by having a set of input data called a training set. The goal of the training is to minimize the loss function. The training process involves set of steps to obtain good weights and biases of the networks neurons. After training the network to learn the relationship between inputs and outputs, and it can produce outputs that is close to original output for the specified inputs.

Gradient Descent

Gradient descent is a iterative optimization function for finding a minimum of a function. Gradient descent iterates from a initial set of parameters to a set of parameters that minimizes the function. The local minimum is found by taking steps proportional to the negative of the gradient at the local point. The algorithm is generic and easy to implement but may result in a local minimum instead of the global minimum [11].

In gradient-based methods, one of the issue is that the parameters tends to not get optimized when the gradient size is big. To solve this problem, we multiply the gradient by a small constant (called the learning rate) in the parameter update rule. However, in order to improve learning efficiency, there are many methods for changing the value at each step, instead of using a constant learning rate like exponential-based learning rate schedule [12].

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is one of the most common training algorithms for neural networks [13]. It is very similar to gradient descent, which is a first order gradient-based optimization method that updates the variables subject to optimization in the direction of steepest descent. Applying gradient descent to neural networks is straightforward, because neural networks define closed form expressions which are differentiable almost everywhere. Many machine learning libraries that support neural networks also provide automatic differentiation tools, making the implementation of such algorithms managable.

Despite the simplicity of implementing gradient descent, stochastic gradient descent is generally preferred over gradient descent because it is less likely to get stuck at local minima or saddle points. This stochasticity is usually introduced by computing the gradients with respect to subsets of the data. Using a batch size of k, an iteration of SGD when minimizing the loss function L(θ) is given by

$$\theta_{i+1} = \theta_i - \sigma \nabla L(\theta_i) \tag{2.6}$$

where L is the loss function evaluated on a randomly selected batch i containing k samples ,i.e.

$$L(\theta) = \frac{1}{k} \sum_{j=1}^{k} (f(x_{i(j)}, \theta) - y_{i(j)})^2 \tag{2.7}$$

where $i(j)$ is taken between {1,2,..k} and {1,2,..N} where N is the total number of available data points.

Adaptive Gradient

The Adaptive Gradient (Adagrad) method is an adaptive method which changes the learning rate by dividing the learning rate by the L2-norm of the gradient at each step [12]. In the denominator, the square of the gradient at each step is added. Therefore, as the learning progresses, the denominator becomes larger.

RMSProp Method

Root Mean Square Propagation(RMSProp) [14] tries to resolve Adagrad's radically diminishing learning rates by using a moving average of the squared gradient, which utilizes the magnitude of recent gradient descents for normalization of the gradient. Therefore, with the increase of the learning rate, the algorithm used would move in a horizontal direction with larger steps converging faster.

Adam Optimizer

Adam realizes the benefits of both AdaGrad and RMSProp. What makes Adam different from gradient descent is that it takes more parameters into account when calculating these steps [15]. In gradient descent there is a constant learning rate, but in Adam the learning rate is changed with respect to both the first (the mean) and second (the uncentered variance) moments of the gradients.

2.2 Physics Informed Neural Networks (PINNs)

In the work done by Raissi et al. [16] [17] [18] , they named a approach for approximating the solutions of differential equation as the physics informed neural networks (PINNs). In many cases, the governing equations or empirically determined rules defining the problem are known a priori. For instance, an incompressible flow has to satisfy the law of conservation of mass and momentum. By incorporating this information, the solution space is drastically reduced and, as a result, less training data are needed to learn the solution. In particular, the goal is to solve problems which can be described by parameterized nonlinear partial differential equations of the form

$$u_t + N[u;\lambda] = 0, x \in \Omega, t \in [0,T] \tag{2.8}$$

Here, the solution $u(t,x)$ depends on time t and a spatial variable x, $N[u;\lambda]$ represents the nonlinear operator with parameter λ, and Ω refers to a space in R^D. This description covers a wide range of problems ranging from advection-diffusion-reaction of chemical or biological systems to the governing equations of continuum mechanics.

Furthermore, there was demonstration of PINNs on a variety of examples that are of interest in a physics and engineering context. The code for the demonstation was written in Python and utilizes the popular GPU-accelerated machine learning framework Tensorflow and it is is publicly available on github [19] allowing others to explore physics-informed neural networks and contribute to their development .

2.2.1 Algorithm for approximate solution of PDEs using PINNs

Let us consider the following PDE which can be parametrized by λ for the solution $u(x)$ with $x = (x_1, , x_2, ...x_d)$ taken on a domain $\Omega \subset R^d$ [20]:

$$f(x; \frac{\partial u}{\partial x_1}, ..., \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial {x_1}^2}, ... \frac{\partial^2 u}{\partial x_1 \partial x_d}; ...; \lambda) = 0, x \subset \Omega, \tag{2.9}$$

with boundary conditions : $B(u, x) = 0$ on $\partial\Omega$ where $B(u, x)$ can be any one of the Neumann, Dirchlet, Robin peroidic boundary conditions.

The PINNs algorithm for solving differential equations [20].

1. Construct a neural network $\hat{u}(x; \theta)$ with parameters θ
2. Specify the two training sets τ_f and τ_b for the equation and boundary or initial condition
3. Specify a loss function by summing the weighted L^2 norm of both the PDE equation and boundary condition residuals.
4. Train the neural network to find the best parameters θ^* by minimizing the loss function $L(\theta; \tau)$.

Constructing a Neural network

Here we construct a neural network $\hat{u}(x; \theta)$ to approximate solution of $u(x)$, which has the input x and outputs is a vector which has a dimension as u. Here $\theta = (W^l, b^l)_{(1<l<L)}$ represents a set of weight matrices and bias vectors in the neural network $\hat{u}$ and we can get the gradients of $\hat{u}$ with respect to x using Automatic Differentiation, which can be easily implemented with Tensorflow or PyTorch [20].

Specifying the training Sets

Here we restrict $\hat{u}$ on some points randomly distributed points in the domain, i.e., the training data $\tau = (x_1, x_2, .., x_{|\tau|})$. Here, τ consists of two sets $\tau_f \subset \Omega$ and $\tau_b \subset \partial\Omega$, that are the points in the domain and on the boundary. We denote τ_f and τ_b as the sets of residual points [20].

Specifying the Loss Functions

To quantify the difference between the neural network $\hat{u}$ and the constraints, we select the loss function as the weighted sum of the L^2 norm of residuals for the equation and boundary conditions [20]:

$$L(\theta; \tau) = w_f L_f(\theta; \tau_f) + w_b L_b(\theta; \tau_b) \tag{2.10}$$

where

$$L_f(\theta;\tau_f) = \frac{1}{|\tau_f|}\sum_{x\in\tau_f}\|f(x;\frac{\partial \hat{u}}{\partial x_1},...,\frac{\partial \hat{u}}{\partial x_d};\frac{\partial^2 \hat{u}}{\partial {x_1}^2},...\frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d};...;\lambda\|_2{}^2 \tag{2.11}$$

$$L_b(\theta;\tau_b) = \frac{1}{|\tau_b|}\sum_{x\in\tau_b}\|B(\hat{u},x)\|_2{}^2 \tag{2.12}$$

where w_f and w_b are weights and definition of weights is a topic of current research. Section 2.3.1 will cover how these weights are defined within SimNet.

Training the Neural Networks

At last, the way of searching for a better θ by minimizing the loss function $L(\theta;\tau)$ is called as training. We know the fact that the loss is nonlinear and non-convex with respect to θ, so we minimize the loss function by gradient-based optimizers as discussed above in Section 2.1.5 [20].

2.2.2 Schematic Representation of PINNs

Let us try to represent PINNs in a schematic way with a simple example which is a 1D-diffusion equation

$$\frac{\partial u}{\partial t} = \lambda\frac{\partial^2 u}{\partial x^2} \tag{2.13}$$

with the boundary conditions $u(x,t) = g_D(x,t)$ on $\Gamma_D \subset \partial\Omega$ and $\frac{\partial u}{\partial n}(x,t) = g_R(x,u,t)$ on $\Gamma_R \subset \partial\Omega$. Let $\hat{u}$ be the neural network with inputs (x,t) that approximates the function $u(x,t)$. We can define the loss function as mentioned in equation 2.10 where,

$$L_f(\theta;\tau_f) = \frac{1}{|\tau_f|}\sum_{x\in\tau_f}\|\frac{\partial \hat{u}}{\partial t} - \lambda\frac{\partial^2 \hat{u}}{\partial x^2}\|_2{}^2 \tag{2.14}$$

$$L_b(\theta;\tau_b) = \frac{1}{|\tau_b|}\sum_{x\in\tau_b}\|[\hat{u}(x,t) - g_D(x,t)] + [\frac{\partial \hat{u}}{\partial n}(x,t) - g_R(x,u,t)]\|_2{}^2 \tag{2.15}$$

where τ_f and τ_b represents the residual or sampling points for the PDE and boundary conditions respectively. The overall representation can be summarised as shown in the Figure 2.6.

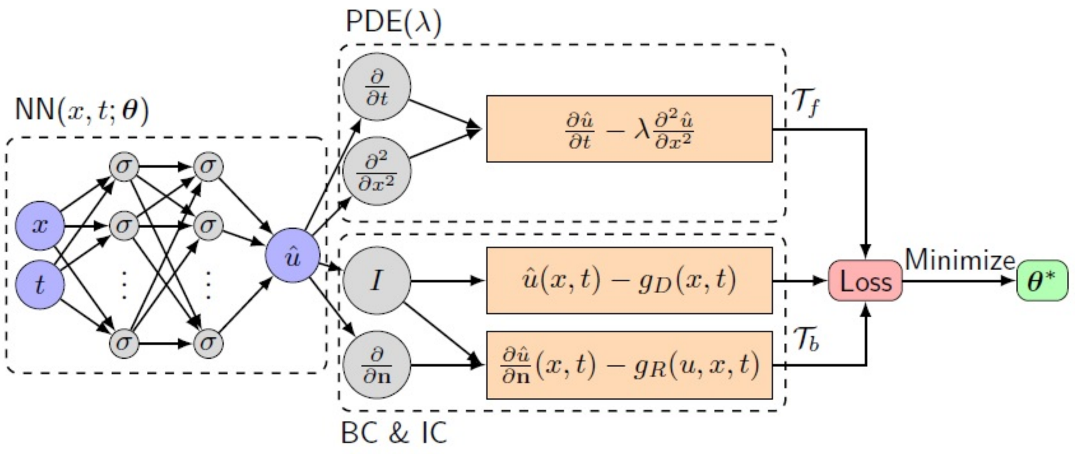

PDE(λ)
NN($x, t; \theta$)
$\frac{\partial}{\partial t}$
$\frac{\partial^2}{\partial x^2}$
$\frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2}$
$\mathcal{T}_f$
x
t
σ
$\hat{u}$
I
$\frac{\partial}{\partial n}$
$\hat{u}(x,t) - g_D(x,t)$
$\frac{\partial \hat{u}}{\partial n}(x,t) - g_R(u,x,t)$
$\mathcal{T}_b$
Loss
Minimize
θ^*
BC & IC

Figure 2.6: Schematic of PINN for Solving 1D diffusion [20]

2.3 SimNet -Tool for PINNs

In this section we are going to have a look into the AI-driven multi-physics simulation framework called SimNet which is developed by NVIDIA for Physics Informed Neural Networks [21].

2.3.1 Physics Informed Neural Networks in SimNet

SimNet is a neural network solver which is able to solve problems with complex geometries. SimNet has different way of formulating the losses than the conventional way explained in Section 2.2. The losses are formulated in integral form and the equation 2.14 can be written as,

$$L_f = \int_\Omega \| \frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2} \|_2{}^2 dx \approx (\int_\Omega dx) * \frac{1}{|\tau_f|} \sum_{x \in \tau_f} \| \frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2} \|_2{}^2 \tag{2.16}$$

Then the approximation of integral is done using Monte Carlo integration. Thereby, effectively the same formulation as equation 2.14 is obtained, but scaled by the area/volume of the domain. NVIDIA argues that this keeps the losses consistent across all domains. The boundary condition losses are treated similarly [22].

2.3.2 Modules in Simnet

SimNet is a Tensorflow based neural network solver and it provides APIs which allows us to make our own applications based on the existing modules. An overview of SimNet architecture is given in Figure 2.7. The modules that allow us to define a physical system are grouped in the geometry modules and PDEs modules. We can also chose network architecture , optimizers of our own choice , which allows us to build a neural network and train the model and minimize the loss function and compute gradients efficiently. The outputs are saved in form of CSV or VTK files and can be visualized using TensorBoard and ParaView. In the following section, we will discuss about all the modules in SimNet and also different network architectures. The training procedure can be done with the help of TensorFlow built-in functions on a single or cluster of GPUs.

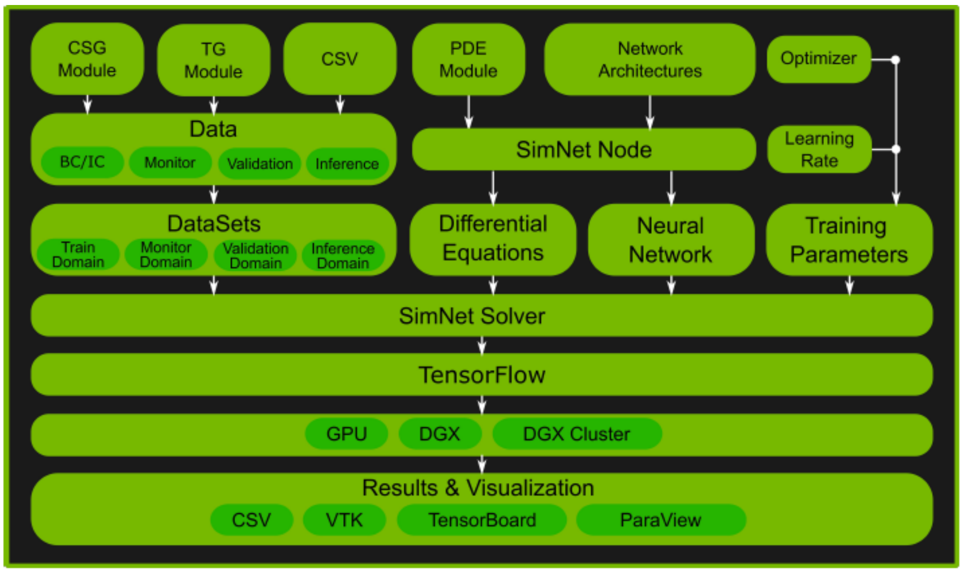


Figure 2.7: SimNet Architecture [21]

Geometry modules

In SimNet there are currently two geometry modules which are called as Constructive Solid Geometry (CSG) and Tessellated Geometry (TG) modules. Geometry primitives can be easily constructed using CSG module and we can perform boolean operations on them. This makes it easier for parameterization of broad range of different geometries and contruct various parameterized geometry for design optimization. The TG module allow us to imports STL, OBJ, and other tessellated geometries that make us to work with complex geometries.

PDE modules

The PDE module consists of common differential equations including the Navier-Stokes, diffusion, advection-diffusion, wave equations, and linear elasticity equations. SymPy module in Python has been also used in SimNet, so that PDE modules can be extended to define our own differential equations. The PDE module in SimNet also provides implementations of turbulence and exact continuity in the Navier-Stokes equations [21].

2.3.3 Activation Functions in SimNet

SimNet uses some other activations functions other than the ones that mentioned in the Section 1.3.3. We can see a brief introduction about those activation functions in this section.

Swish Function

A little modification is added to sigmoid, to define a swish function. ReLU produces zero as output for negative inputs and cannot be back-propagated. Herein, swish can partially handle this problem. The function is formulated as $sigmoid(x)$ multiplied by x shown in Figure 2.8.

$$swish(x) = \frac{x}{1 + exp(-x)} \tag{2.17}$$

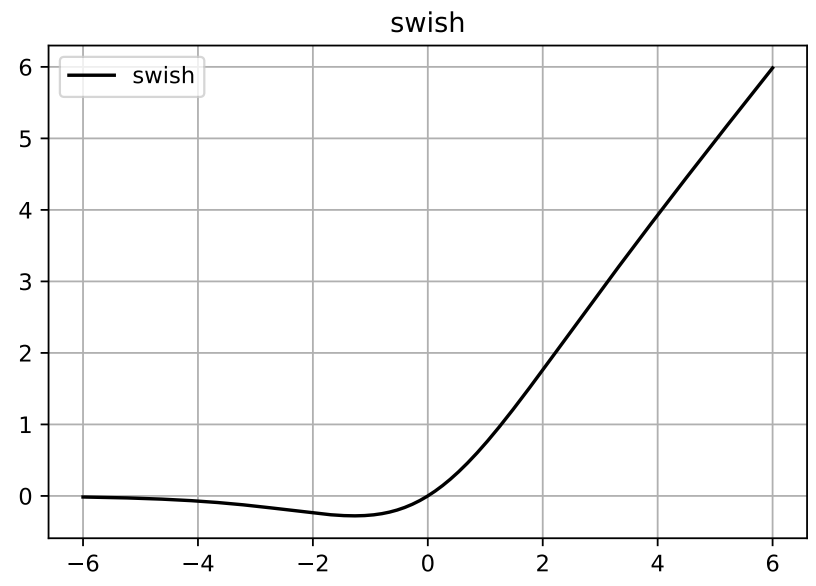


Figure 2.8: Swish Function

Exponential Linear Unit(ELU)

In ELU, if you input an x-value that is greater than zero, then it's the same as the ReLU which means the result will be a y-value equal to the x-value. But this time, if the input value x is less than 0, we get a value slightly below zero. It is defined as :

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

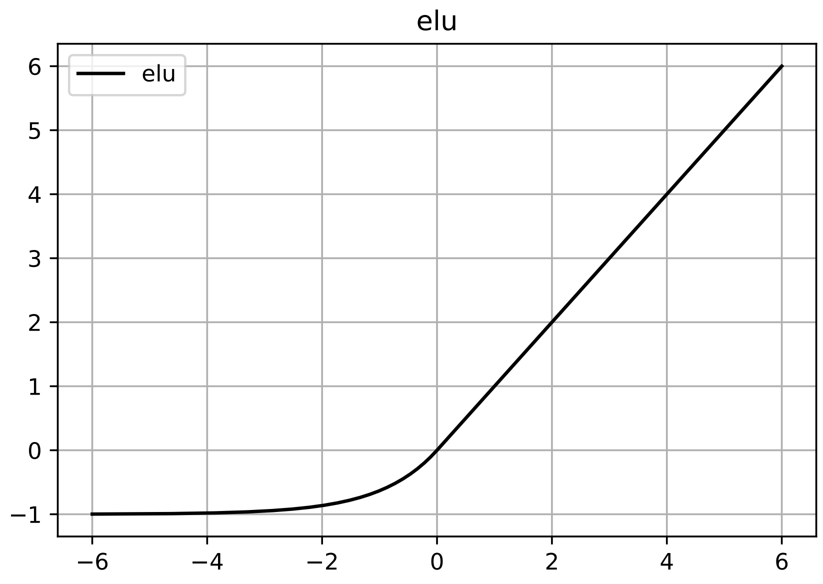


Figure 2.9: ELU activation function

The alpha controls the value to which an ELU function saturates for negative net inputs as described in Figure 2.9. ELU reduce the vanishing gradient effect [23]. The alpha value chosen here is 1.0 which is also the default value in Tensorflow.

Leaky ReLU

In SimNet ,Leaky ReLU takes the mathematical form as given below and can be viewed in Figure 2.10.

$$L - ReLU(x) = 0.55x + 0.45|x|$$

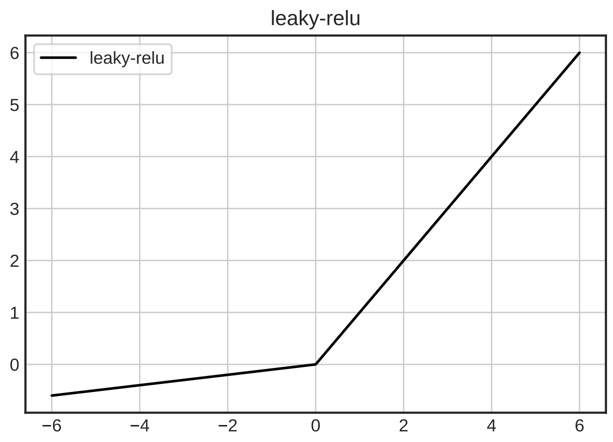


Figure 2.10: Leaky-ReLU

SeLU

In Scaled Exponential Linear Unit(SeLU) ,the input value x is greater than zero, the output value becomes x multiplied by λ. If the input value x is less than or equal to zero, we have a function that goes up to 0, which is our output y, when x is zero. Essentially, when x is less than zero, we multiply α with the exponential of the x-value minus the alpha value, and then we multiply by the λ value [24]. In SimNet $\lambda \approx 1.0507$ and $\alpha \approx 1.6733$ and plot is shown in Figure 2.11.

$$SeLU(x) = \lambda \begin{cases} x & \text{if } x > 0, \\ \alpha e^{x} - \alpha & \text{if } x \leq 0 \end{cases}$$

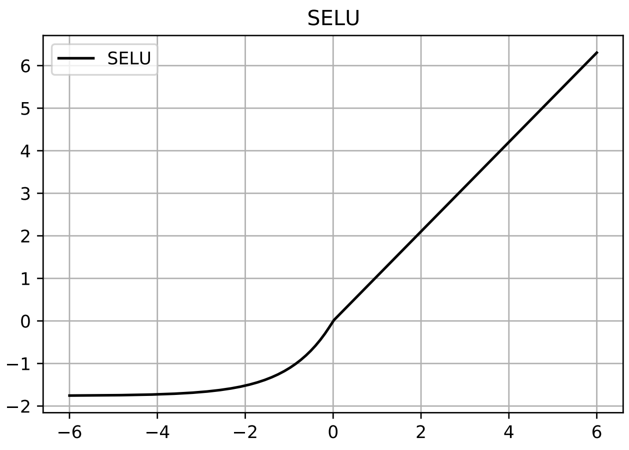


Figure 2.11: SELU

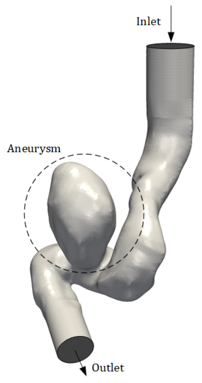
The concatenated versions of above activation functions and the commonly used activation functions discussed in Section 2.1.3 are also included in SimNet and some of these activation functions are included as a parameter to study which influences the convergences and accuracy of PINNs in the later sections.

2.3.4 Example in SimNet

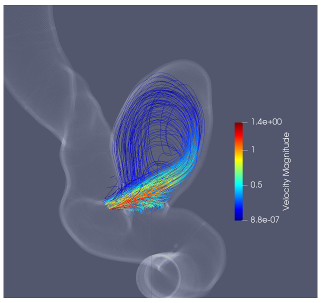
There are different types of examples in SimNet such as turbulent and multi-physics simulation, simulation with complex geometries, design optimization for a multi-physics system, and an inverse problems. For illustrative purposes one example is picked from the user guide [22] to demonstrate SimNet's capability to deal with complex flow problems.

Blood flow in an Intracranial Aneurysm

We can examine the ability of SimNet to work with STL geometries from a CAD system [21]. Using the SimNet's TG module, the simulation of the flow inside a patient specific geometry of an aneurysm is investigated as shown in the Figure 2.12a. The streamline plot in Figure 2.12b shows that SimNet captures the flow field. Then, Figure 2.13 also shows the good agreement of the SimNet prediction in comparison with a CFD simulation using OpenFoam.



(a) Aneurysm geometry



(b) Streamlines inside the aneurysm sac

Figure 2.12: SimNet results for the aneurysm problem [22]

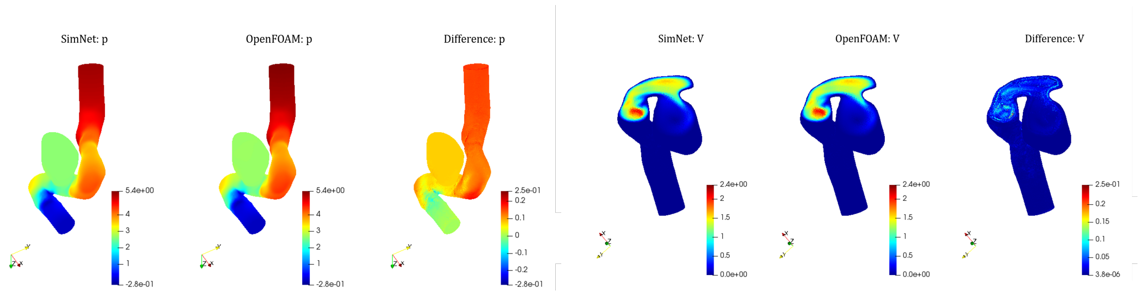


Figure 2.13: comparison between the SimNet and OpenFOAM results for the velocity magnitude and pressure [22]

2.3.5 Methods/Parameters to Improve Convergence

In the following section, we will have a brief introduction about some of the methods used in SimNet to improve the accuracy and convergence speed of PINNs results. These methods can come in handy when we use SimNet for larger scale problems and makes SimNet robust to use.

Global Adaptive Activation Functions

In Section 2.1.5, we discussed that minimizing the loss function is given by equation 2.6. In global adaptive activation [25], another parameter α is multiplied to the inputs to change the slope of activation functions and equation 2.6 can be rewritten as

$$\theta_i = \theta_{i-1} - \sigma(\alpha \nabla L(\theta_{i-1})) \tag{2.18}$$

Now minimizing the loss function is done by optimizing α together with the θ which is a set weights and biases , which can be described as

$$\alpha^*, \theta^* = \underset{\theta,\alpha}{\operatorname{argmin}}\, L(\theta, \alpha) \tag{2.19}$$

Here α is called as global adaptive activation parameter.

Integral Continuity Planes

For some of the fluid flow problems involving channel flow, it is observed that in addition to solving the Navier-Stokes equations in differential form, specifying the mass flow through some of the planes in the domain significantly speeds up the rate of convergence and gives better accuracy. Assuming there is no leakage of flow, the flow exiting the system must be equal to the flow entering the system. Also, by specifying such constraints at several other planes in the interior improves the accuracy further. For incompressible flows, one can replace mass flow with the volumetric flow rate [22].

Spatial Weighting of Losses(SDF)

Signed distance functions, or SDFs for short, when passed the coordinates of a point in space, return the shortest distance between that point and some surface. The sign of the return value indicates whether the point is inside that surface or outside (hence signed distance function). We can recall to equation 2.10 where the weights w_f and w_b impacts the convergence of the solution. We can rewrite the integral formulation from equation 2.16 as

$$L_f = \int_\Omega w_f(x) \| \frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2} \|_2{}^2 dx \tag{2.20}$$

The signed distance function (SDF) is used for $w_f(x)$ and hence the figure 2.14 shows the impact on how SDF improves the accuracy. The L2 error term will be explained more clearly in the following sections. This graph is to show why SDF is used as default [22].

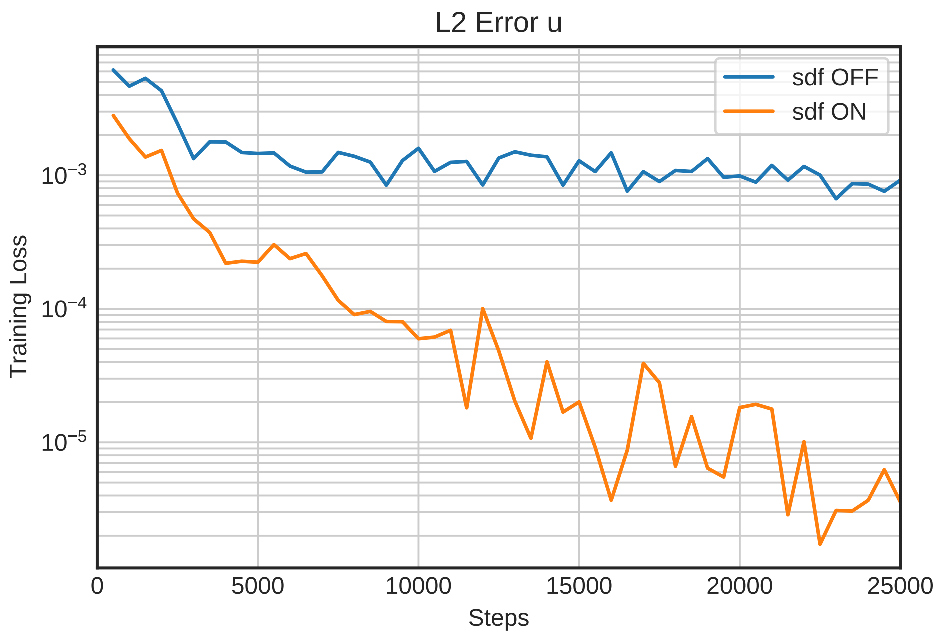


Figure 2.14: Improvements in convergence speed by weighting the equation residuals spatially

Quasi Random Sampling

Training points in SimNet are generated according to a random distribution by default. An other way is the quasi-random sampling, that gives us a way to generate training points with a low level of variation across the domain. There are other sequences like Halton sequences [26], the Sobol sequences and the Hammersley sets. SimNet uses Halton sequences. The Figure 2.15 shows how points are generated using random sampling and Halton sequences for an exemplary geometry example [22].

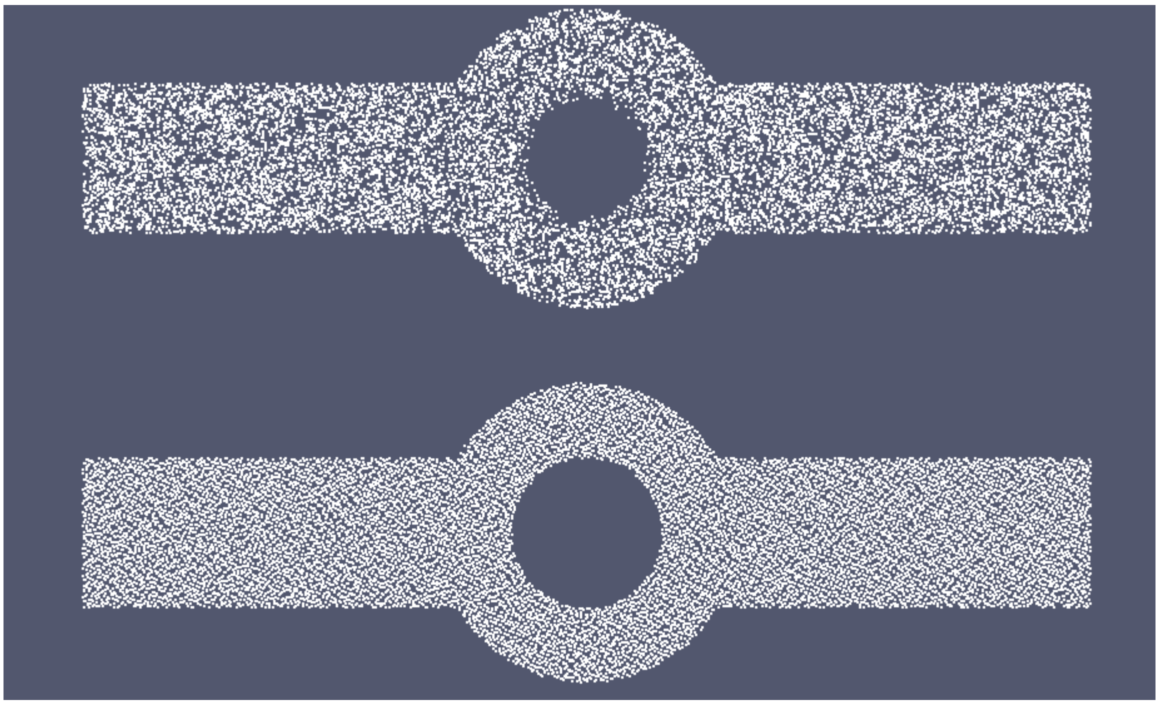


Figure 2.15: batch of training points generated using uniform sampling (top) and Halton sequences (bottom) for the annular ring example in tutorial [21]

2.4 Advanced Architectures

In addition to the feed-forward, fully connected networks, SimNet offers a number of more advanced architectures such as Fourier feature networks , Modified Fourier feature networks and Sin activation functions dubbed Sinusoidal Representation Networks(SiReNs). These architectures are not investigated in this work and it is therefore referred to [22] for more details.

3 Physics Informed Neural Networks (PINNs)

3.1 Problem Definition

3.1.1 2D flow around a Cylinder

In this section, we employ the proposed PINNs to model the steady flow passing a circular cylinder. For the entire neural network solvers in this case study, the architectures consist of 6 layers, each with 512 units, and Swish activation function as their default. We used the Adam optimizer with an initial learning rate of 10^{-4} [21].

Geometry and flow configuration

The underlying geometry is a channel with dimensions 2.2 x 0.41 with a circular cylinder with r = 0.05 and centre is at $(0.2, 0.2)$. The gravity is ignored. Taking the fluid density $\rho = 1.0$, the fluid is characterised by the Navier-stokes equation [27].

$$-\nu \triangle u + u \nabla u + \nabla p = 0 \tag{3.1}$$

$$\nabla . u = 0 \tag{3.2}$$

with u defining the velocity vector and p the pressure. The kinematic viscosity is taken as $\nu = 0.001$.

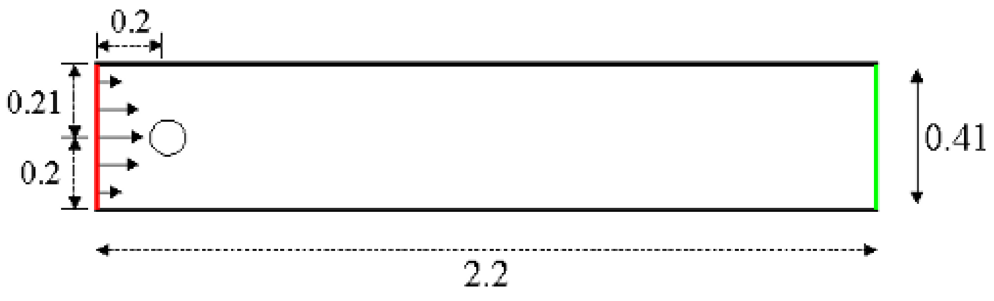


Figure 3.1: Flow around a cylinder

Boundary Conditions

For the lower and upper walls $\Gamma_1 = [0, 2.2] \times 0$ and $\Gamma_2 = [0, 2.2] \times 0.41$ as well as the boundary $S = \partial B_r(0.2, 0.2)$, no slip boundary conditions are defined,

$$u_{|\Gamma_1} = u_{|\Gamma_2} = u_{|S} = 0 \tag{3.3}$$

On the left edge $\Gamma_3 = 0 \times [0, 0.41]$, a parabolic inflow profile is prescribed by equation 3.4 with the maximum velocity $U = 0.3$. On the right edge , $\Gamma_4 = 2.2 \times [0, 0.41]$, do nothing boundary conditions define the outflow.

$$u(0, y) = (\frac{4Uy(0.41 - y)}{0.41^2}, 0) \tag{3.4}$$

3.1.2 Case Setup in Simnet

To set up case in SimNet , we can create a single python script containing the geometry , boundary conditions and network information. We will go through the steps in writing the python script and entire code will be available in the appendix part A.

$\boxed{\text{Steps to solve the Case Study in SimNet [22].}}$

1. Creating geometry using the geometry module in SimNet.

2. Defining training domain for training our Neural network.

3. Creating validation data

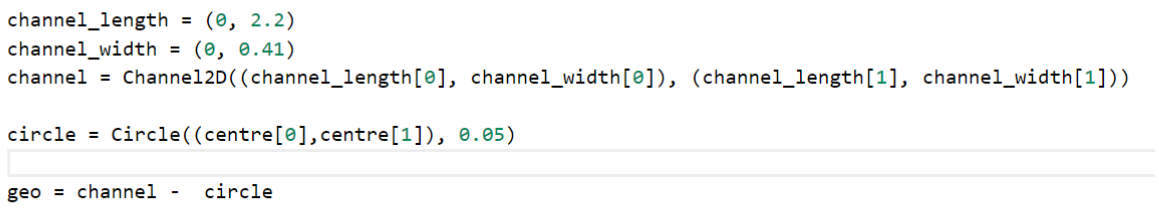
4. Making the network solver

5. Running the SimNet solver

6. Results and postprocessing in SimNet

Creating Geometry

Once we are done with importing all the required modules , we can generate the geometry of the case study using the geometry modules in SimNet. The geometry module has pre-defined 2D and 3D shapes to construct a geometry. In our case we are going to use the 2D channel and circle and construct by performing subtract operations shown in Figure 3.2.



```
channel_length = (0, 2.2)
channel_width = (0, 0.41)
channel = Channel2D((channel_length[0], channel_width[0]), (channel_length[1], channel_width[1]))

circle = Circle((centre[0],centre[1]), 0.05)

geo = channel -  circle
```

Figure 3.2: creating geometry in SimNet

Defining Training Domain

Here we define a training domain for training our neural network. As a result of this , a loss function is created that is a combination of boundary conditions in equation 3.3, 3.4 and equations in 3.1, 3.2 that a neural network should satisfy after training. These training points for boundary conditions and equations are defined as sample points on boundary of the geometry and the conservation equations of fluid mechanics are enforced on all the points on the interior of the geometry [22].

Creating validation data

We know, SimNet doesn't require training data from any other CFD/PDE solver to make predictions as it is a physics based neural network solver. But, in SimNet it is possible to add CFD data from a solver like OpenFoam and which can be used to compare the SimNet's result. The results from OpenFoam are added or imported into SimNet as a .csv file [22].

Making the Neural Network Solver

The train domain and validation domain are assigned here and the equation to be solved are specified here too. In the given case, the Navier-Stokes equations are readily defined in the PDE module of SimNet. Subsequently, the inputs and outputs of neural network is specified and nodes of architecture are made.

Results and Post Processing

The Tensorboard can be used to visualise the different losses at each step during the training. The Adam optimizer loss is the total loss computed by the network. The 'L2_continuity' , 'L2 _momentum_x' and 'L2 _momentum_y' tells us the L2 loss computed for the continuity and momentum equation in x and y direction respectively in equations 3.7 and 3.8. The 'L2 _u ' and 'L2 _v' determine how well the boundary conditions are satisfied and it is shown in equation 3.6. The 'L2 _relative-error _u' and 'L2 _relative-error _v' are relative error with respect to validation data and relative error in SimNet as calculated using the formula in equation in 3.5.

$$L2\_relative\_error = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}\|f_{true} - f_{pred}\|^2}}{variance(f_{true})} \tag{3.5}$$

$$L2\_Error = \frac{1}{N}\sum_{i=1}^{N}\beta * (True\_value_{bc} - Predicted\_value_{bc})^2 \tag{3.6}$$

$$L2\_Momentum\_loss\_x = \frac{1}{N}\sum_{i=1}^{N}(momentum_x(x_i, y_i) - momentum_x(\hat{x_i}, \hat{y_i}))^2 \tag{3.7}$$

$$L2\_Continuity\_loss\_x = \frac{1}{N}\sum_{i=1}^{N}(continuity(x_i, y_i) - continuity(\hat{x_i}, \hat{y_i}))^2 \tag{3.8}$$

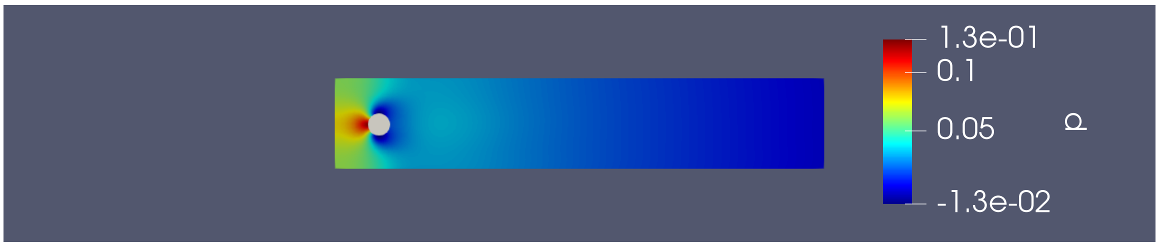
where β is the weighting used in Monte Carlo integration as discussed in Section 2.3.1 and N is the total number of sampling points. x_i, y_i are the true values and $\hat{x_i}$,$\hat{y_i}$ are the network predicted values. The true value for momentum and continuity in our case are zero.

3.1.3 Validation

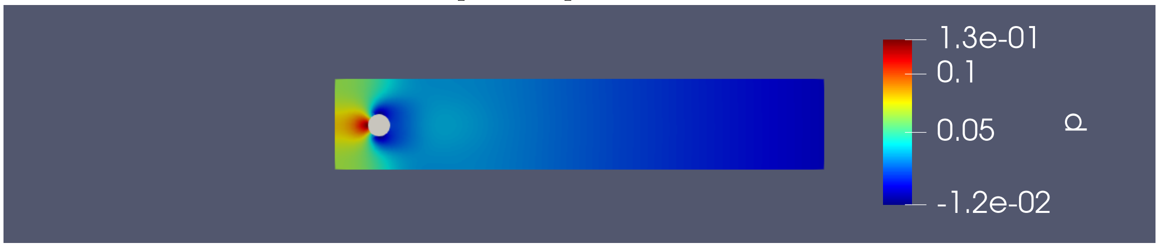
In the previous we saw a brief explanation of how to write a SimNet python code for our case study, some terminology used for different loss functions and the sample code can be seen in Section 5.1 of appendix A. Here, we will see the results we obtained after training the neural networks for 25k steps and compare it with OpenFoam solution and analyse how close it is to it. Below table depicts the number of sampling points used by default. The results after training and comparison is given in Figures 3.3 , 3.4 , 3.5.

geometry	No.of Sampling points
upper wall	99
lower wall	99
inflow	20
outflow	20
cylinder	54
interior	3611

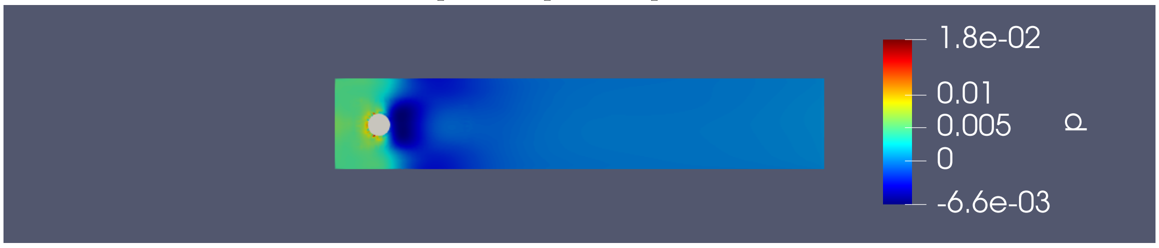
Table 3.1: Default Number of Sampling points



(a) pressure p from SimNet

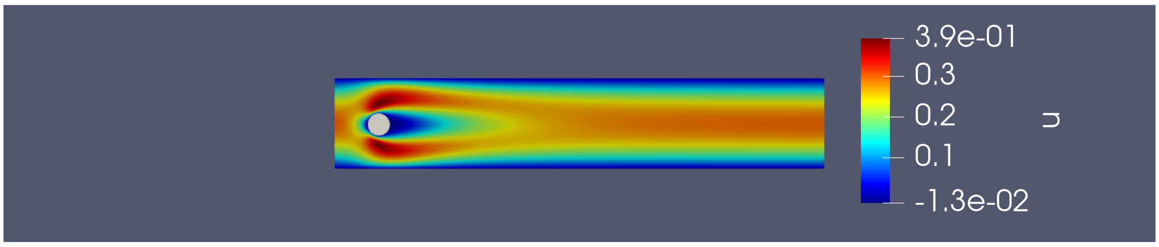


(b) pressure p from OpenFoam

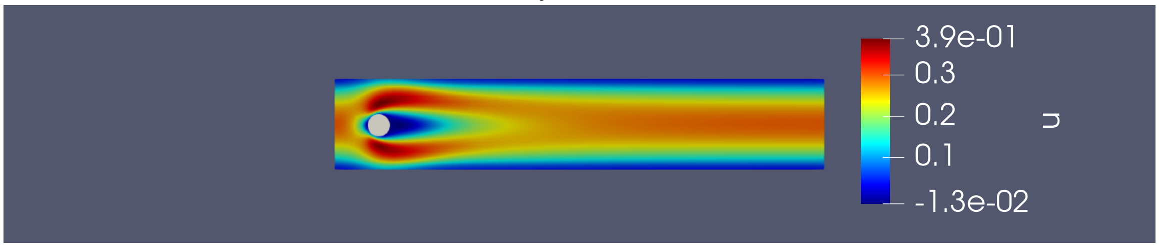


(c) p:difference

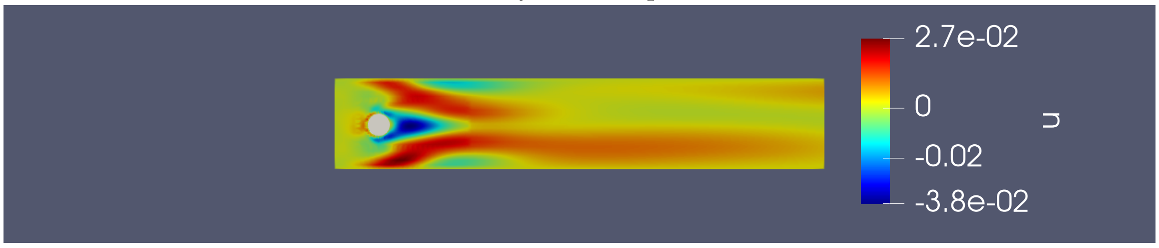
Figure 3.3: pressure distribution comparison



(a) velocity u from SimNet

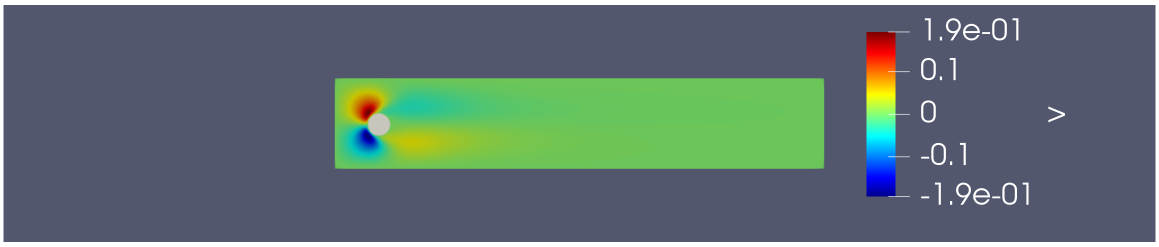


(b) velocity u from OpenFoam

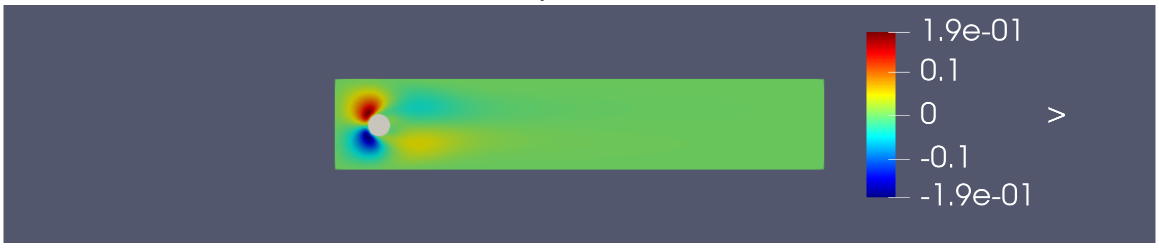


(c) u:difference

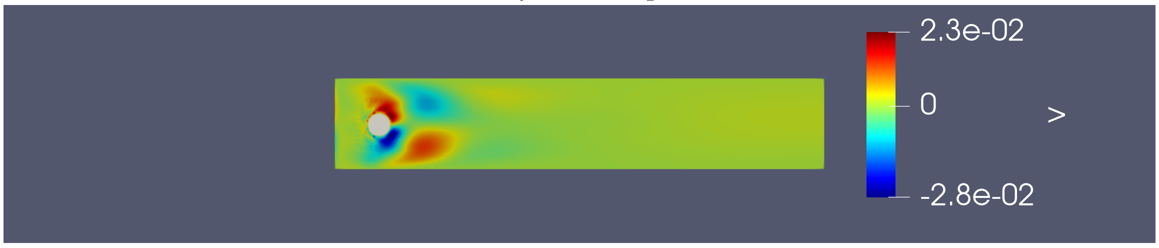
Figure 3.4: velocity magnitude comparison



(a) velocity v from SimNet



(b) velocity v from OpenFoam



(c) v:difference

Figure 3.5: Comparison of velocity(v) profiles

The above figures clearly depicts the SimNet results are in good accordance with Open-Foam results for the given set of sampling points and defaults parameters used. In the next section, defaults parameters will be changed separately and studied how these parameters influence the convergence and accuracy of PINNs in SimNet.

3.2 Study of Different Parameters

3.2.1 Activation Functions

Choice of activation function has a significant impact on the success of training PINNs to solve PDEs. During training following the backpropagation algorithm, the derivative of the loss function with respect to the weights of each layer must be calculated. In this calculation, the derivative of activation functions is multiplied by itself several times, equal to the layer distance from the output. For an activation function where the derivative is squeezed into a narrow range, training of a neural networks with many hidden layers may not be successful.

As we discussed before , swish is the default activation function used . But in literature study , we discussed about some activation functions that are available in SimNet and we are going to study and compare how these activation functions impact the convergence rate. We keep the default values same and just change the activation functions one by one and we plot the overall training loss in a separate plot to analyse the convergence rate. The plot is shown in the Figure 3.6.

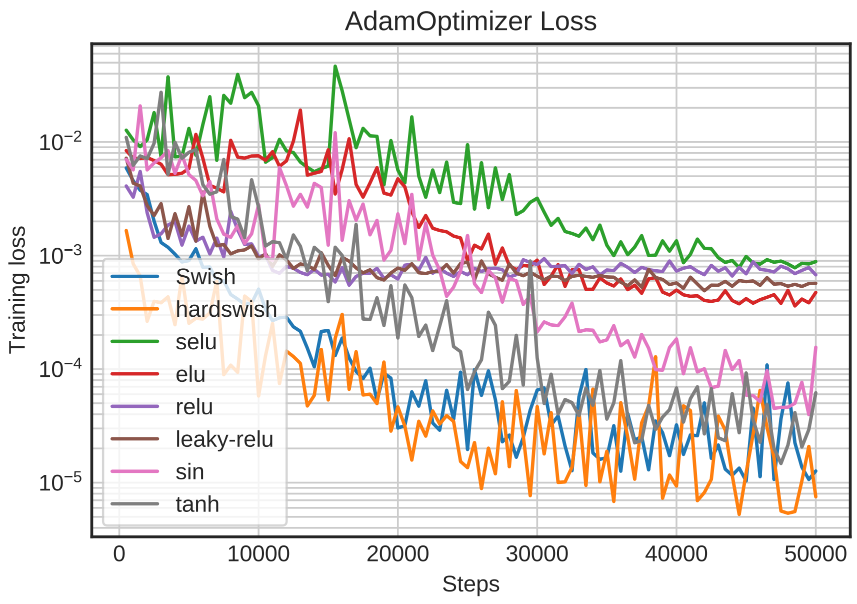
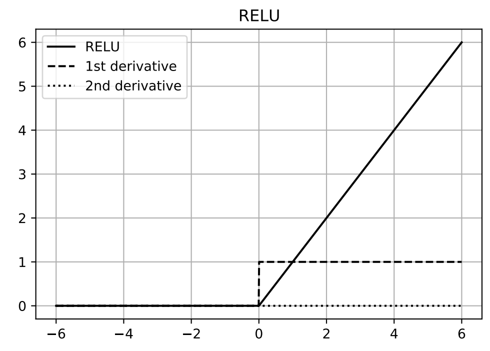


Figure 3.6: AdamOptimizer loss for Different Activation Functions

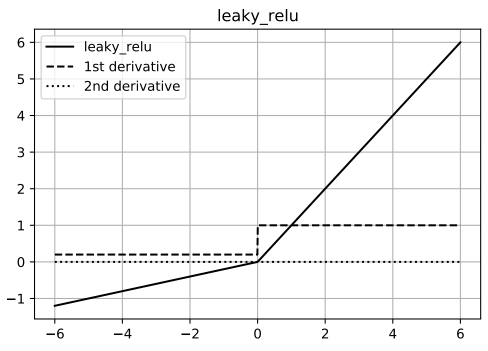
We group them into 4 different compare and analyse them separately in the following sections.

RELU Vs L-RELU

RELU has dying relu(ignores negative values and hence no weight updates in back propagation) problem which was overcomed by leaky relu [4]. For solving a PDE such as Navier-Stokes, it is necessary to take the first and second derivatives of the neural networks with respect to network inputs to calculate the loss function. However, second derivatives of ReLU and Leaky-RELU are equal to zero as shown in Figure 3.7. This makes the training process ineffective.



(a) RELU

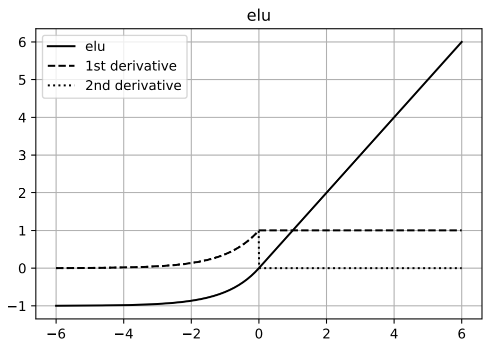


(b) Leaky-Relu

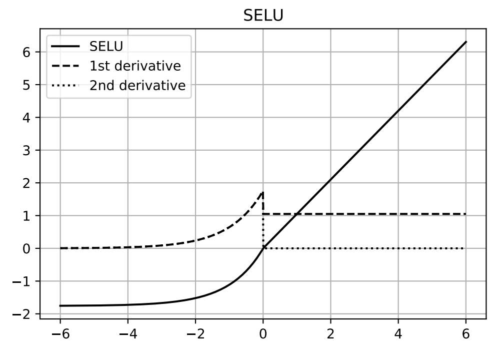
Figure 3.7: RELU Vs Leaky-RELU

ELU Vs SELU

In ELU, in the negative region, the curve is not a straight line because of the exponential term. Because of this term only, the negative values saturate to some level and as a result, the model is not impacted more by the noise which is the case in leaky-relu where we get considerable negative output for a very large negative input. The plots of these two functions are plotted along with their derivatives in the Figure 3.8. The second derivative of ELU is non-zero and also first derivative doesn't have peaks like SELU, and hence its loss value is lesser than other three activation functions.



(a) ELU

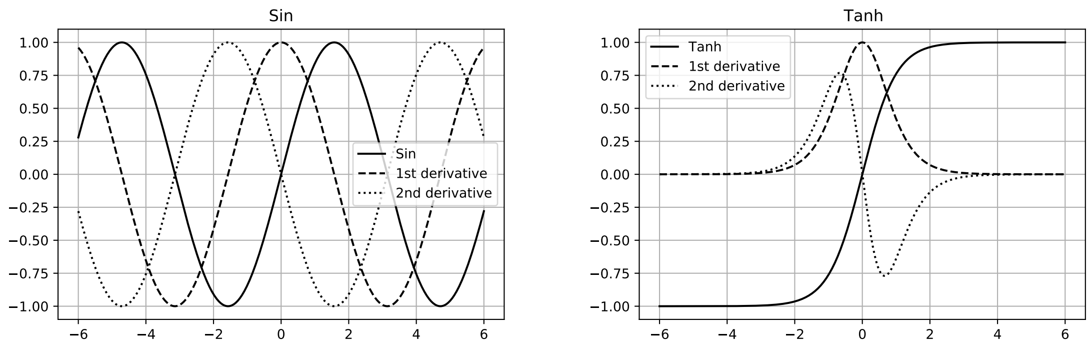


(b) SELU

Figure 3.8: ELU Vs SELU

Tanh Vs sin

sin and tanh are non linear functions and they also have non-zero second derivatives and they are zero centered as shown in Figure 3.9, hence perform better compared to previous four activation functions.



(a) sin (b) tanh

Figure 3.9: tanh Vs sin

Swish

Swish and its derivatives are a smooth function. This allows the optimizer to go through fewer oscillations which helps in faster convergence, effective optimization and generalization and details in Figure 3.10.

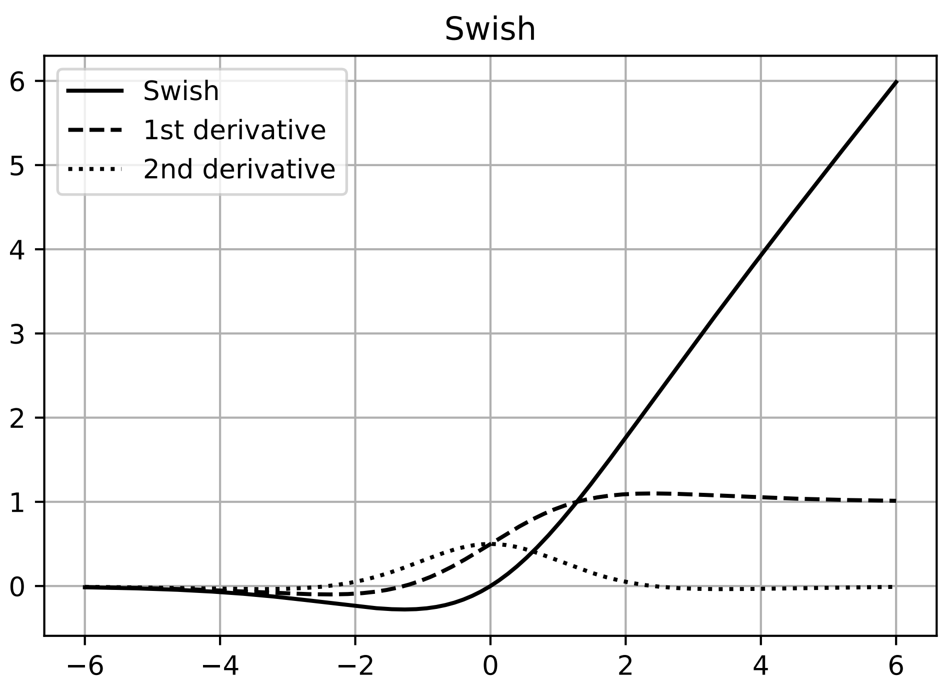


Figure 3.10: swish

In conclusion, we noticed the activation functions that have second derivatives close to zero perform badly or have bad convergence rate. Also we learn Swish perform better than all other activation function mentioned above because they are non-monotonic (RELU and others are either entirely non increasing or non decreasing). Hence, swish is chosen as default by SimNet.

Hard swish also performs similar to swish, but the formulation was different in SimNet and the details and plots are in Section 5.2 of Appendix A. Hence, hard swish is also considered in the following studies.

3.2.2 Influence of Point Sampling

In the Section 3.1.3, we did validation with respect to an OpenFoam solution and we used default number of sampling points as mentioned in the table 3.1. Here we are going to increase the number of sampling points and we study how this influences the convergence rate and solution accuracy. The sampling points that we are going to use are mentioned in the table 3.2 which is approximately close to the number of mesh points used in OpenFoam which is given in the table 3.3.

geometry	Sampling1(default)	Sampling2
upper wall	99	200
lower wall	99	200
inflow	20	40
outflow	20	40
cylinder	54	108
interior	3611	15285

Table 3.2: Different set of Sampling points

geometry	mesh1	mesh2
upper wall	100	200
lower wall	100	200
inflow	20	40
outflow	20	40
cylinder	54	108
interior	2374	10049

Table 3.3: OpenFoam Settings: Mesh points corresponding to their sampling points

Sampling1 vs Sampling2 - Improving Convergence

In the Section 2.2.4, we discussed briefly about the parameter that can be used to improve convergence. Here we will use parameters such as integral continuity planes, adaptive activations and quasirandom sampling both in Sampling1 and Sampling2 and study how the different losses like Overall loss , boundary condition loss , validation loss are influenced. We used 6 integral continuity planes as shown in the Figure 3.11.

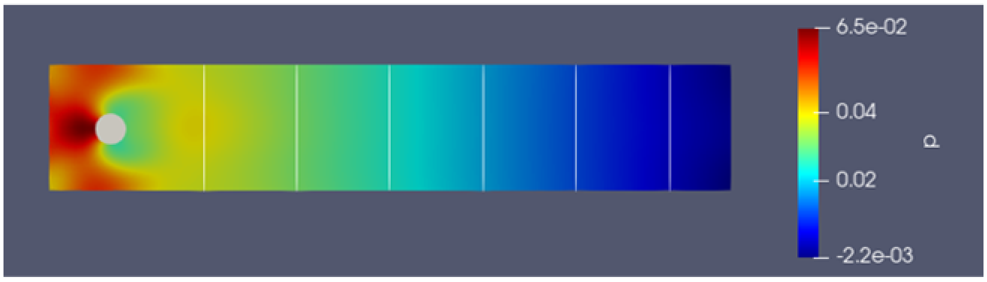
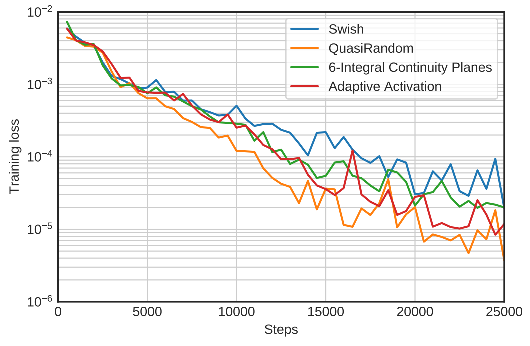
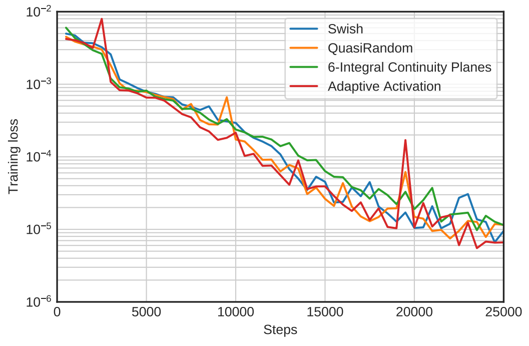


Figure 3.11: 6- integral continuity planes



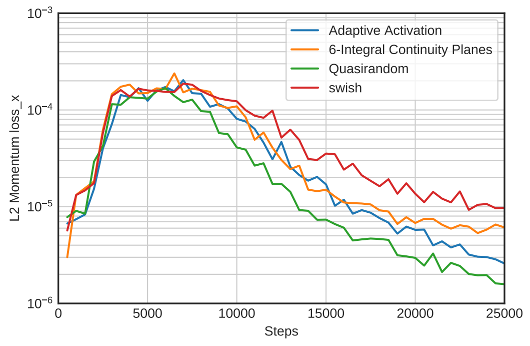
(a) Sampling1 - Overall loss



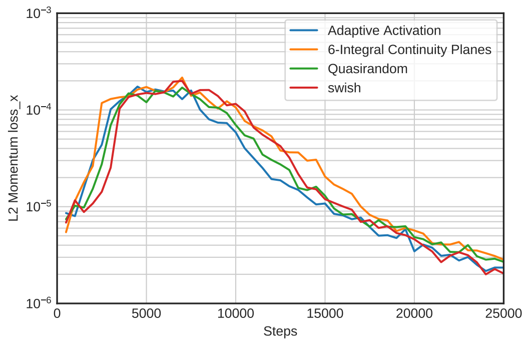
(b) Sampling2 - Overall loss

Figure 3.12: Sampling1 Vs Sampling2 - Overall loss for networks with two different training data sampling strategies

Here in the Figure 3.12 we can notice there is improved convergence with quasi random settings in Sampling1 compared to default setting which is plotted as swish in the graph, but for Sampling2 we see the convergences rates are similar. The similar kinds of trends are seen in the validation loss and momentum loss in x direction and also continuity loss which are clearly seen in Figures 3.13, 3.14, 3.15.

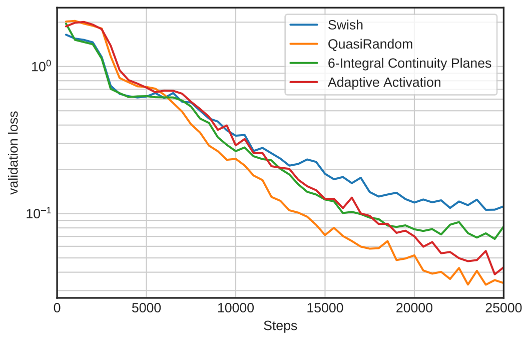


(a) Sampling1 -L2 Momentum loss

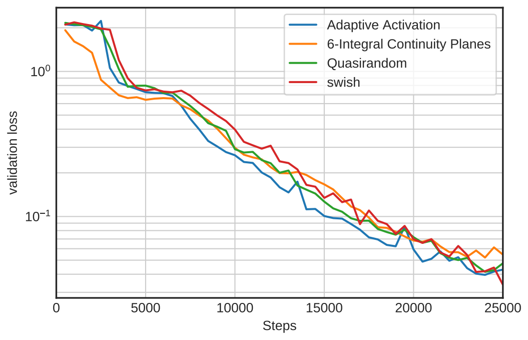


(b) Sampling2 - L2 Momentum loss

Figure 3.13: Sampling1 Vs Sampling2 -Momentum loss in x direction

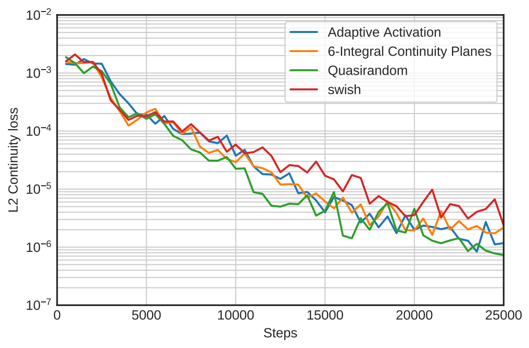


(a) Sampling1 - Validation loss in u

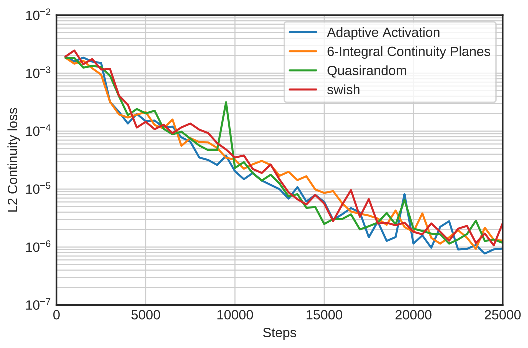


(b) Sampling2 - Validation loss in u

Figure 3.14: Sampling1 Vs Sampling2 -Validation loss in u



(a) Sampling1 - L2 Continuity loss



(b) Sampling2 - L2 Continuity loss

Figure 3.15: Sampling1 Vs Sampling2 -L2 Continuity loss

It is learned that in case of Sampling1 which has fewer training points, quasi random settings can improve convergence. In case of Sampling2 that has more training points, adaptive activation gives the minimum loss value as it is seen in the table 3.4.

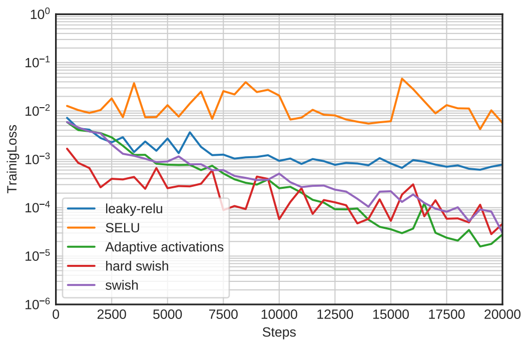
Training sets	Swish	Quasirandom	6-Integralplanes	Adaptive activations
Sampling1	1.96e-5	3.74e-6	2.00e-5	1.18e-5
Sampling2	9.5e-6	1.15e-5	1.14e-5	6.61e-6

Table 3.4: Overall Loss value after 25k steps for the two sampling sets

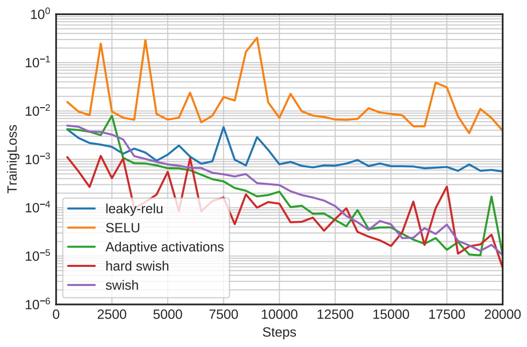
This section gives a brief explanation of how different sampling points and parameters that can be used to get to convergence faster. Also it is learned that when there are less sampling points, quasi random setting have a bigger influence but in other case integral continuity planes and quasi random settings does not have bigger impact.

Sampling1 vs Sampling2 - Different Activation functions

In this section , we compare Sampling1 and Sampling2 by taking two activation functions that showed slower convergence (SELU and Leaky RELU) and two activation functions showed faster convergence (Swish and hard Swish) as in Section 3.2.1 and also we take account of adaptive activation as per previous section and study different losses. Here we can see, irrespective of the sampling size, the validation loss, boundary condition loss and overall loss don't converge at all for SELU and Leaky-relu. For swish activation functions, the convergence is faster with increasing number of sampling points which can be seen clearly from the graphs 3.16, 3.17, 3.18, 3.19.

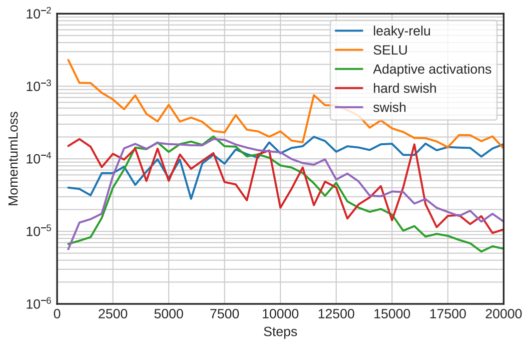


(a) Sampling1 - Overall Loss

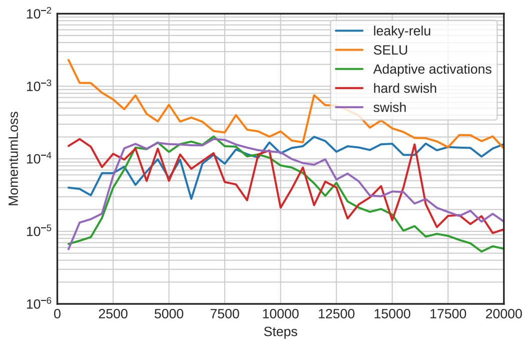


(b) Sampling2 - Overall Loss

Figure 3.16: Sampling1 Vs Sampling2 - Overall Loss for different activation functions

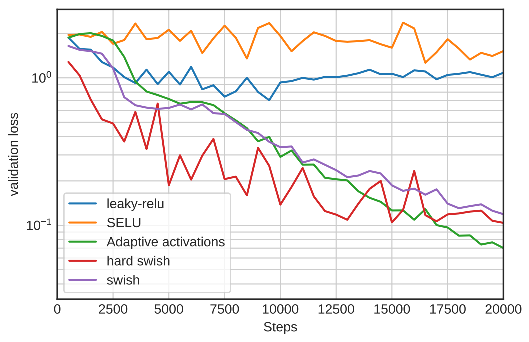


(a) Sampling1 - Momentum loss

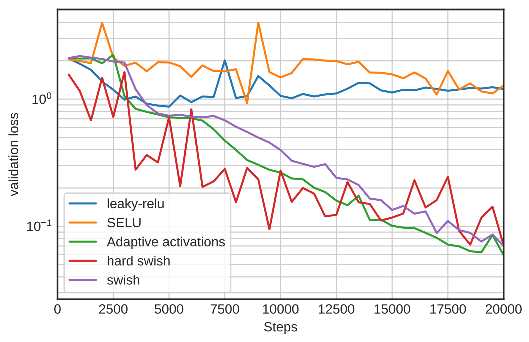


(b) Sampling2 - Momentum loss

Figure 3.17: Sampling1 Vs Sampling2 - L2 momentum loss in x direction for different activation functions

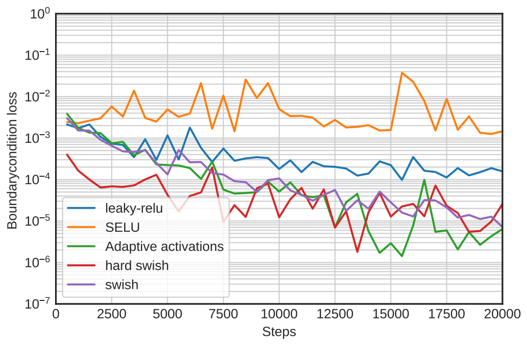


(a) Sampling1 - Validation loss in u

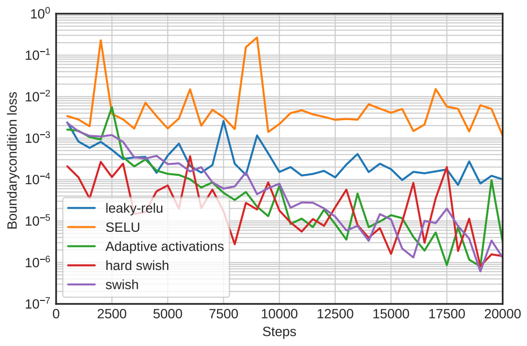


(b) Sampling2 - Validation loss in u

Figure 3.18: Sampling1 Vs Sampling2 -Validation loss in u for different activation



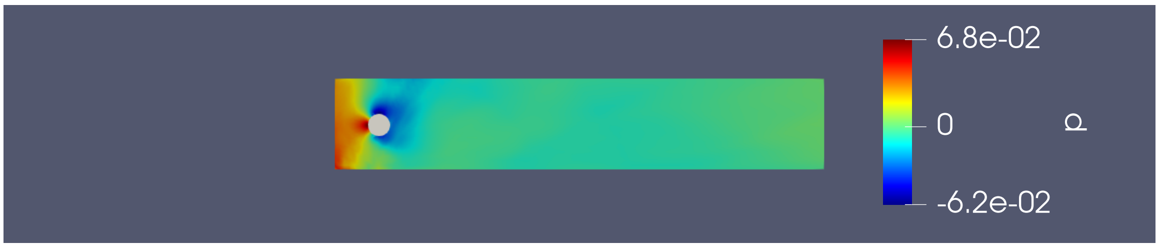
(a) Sampling1 - Boundary Condition loss in u



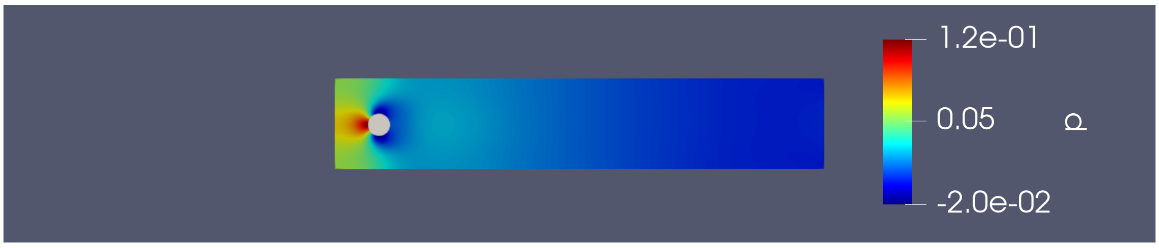
(b) Sampling2 - Boundary Condition loss in u

Figure 3.19: Sampling1 Vs Sampling2 -Boundary Condition loss in u for different activation

We will also plot the pressure distribution p and velocity magnitude for Sampling1 in the Figures 3.20 , 3.21 , 3.22. As you can see in the below plots, SELU gives unphysical results, while swish converges well and gives accurate results for same number of steps(20k).

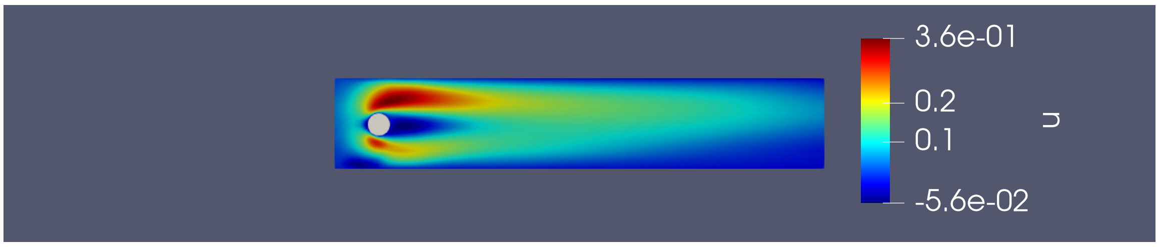


(a) Pressure distrbution with SELU activation Function

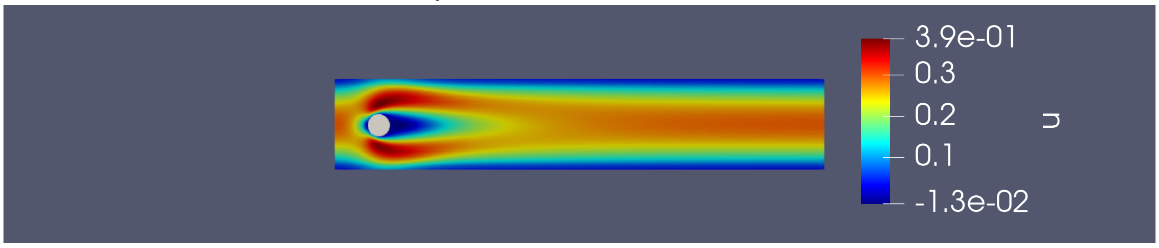


(b) Pressure distribution with Swish Activation Function

Figure 3.20: Pressure distrbution in Sampling1 - Swish Vs SELU

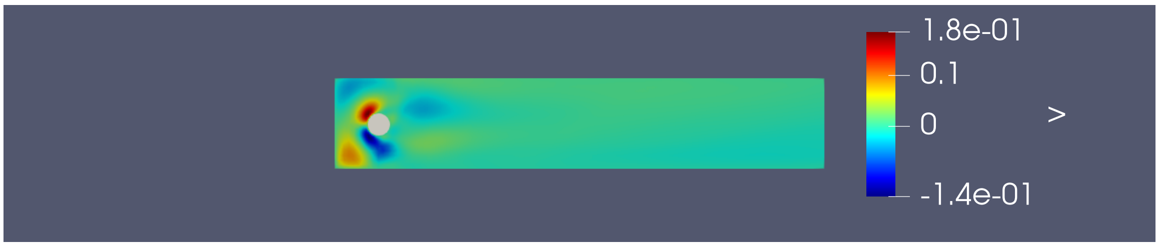


(a) Velocity u with SELU activation Function

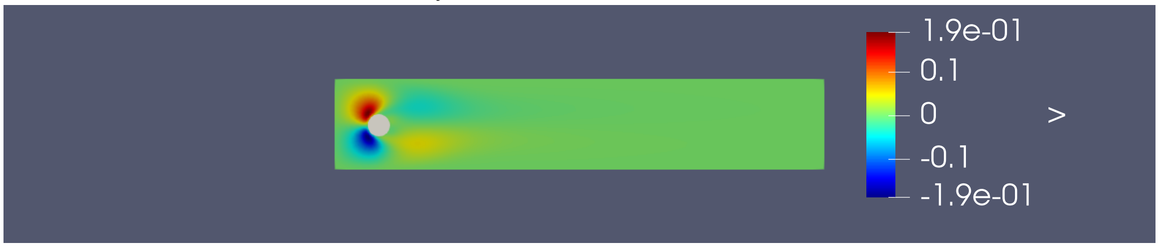


(b) Velocity u with Swish Activation Function

Figure 3.21: Velocity u in Sampling1 - Swish Vs SELU



(a) Velocity v with SELU activation Function



(b) Velocity v with Swish Activation Function

Figure 3.22: Velocity v in Sampling1 - Swish Vs SELU

In conclusion from the above observations,Sampling1 set of training points with quasi random setting and adaptive function with swish activation gives best convergence. This setting will be taken into account for our application problem in Section 3.3.

3.2.3 Number of Layers and Number of Nodes

In this section, we are going to analyse the different combinations of number of layers and number of nodes in each layer as depicted in the table 3.5 and see which gives better convergence rate and the hardware used for running the training is NVIDIA TITAN X.

number of nodes	Number of layers		
	3	6	12
256	3*256	6*256	12*256
512	3*512	6*512(default)	12*512
1024	3*1024	6*1024	12*1024

Table 3.5: Different combination to be analysed

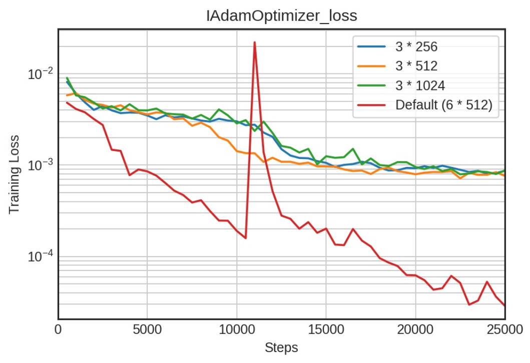
Different combinations	time taken for 25k steps(minutes)
3*256	16
3*512	28
3*1024	67
6*256	28
6*512	57
6*1024	147
12*256	52
12*512	111
12*1024	304

Table 3.6: Training time for different combinations

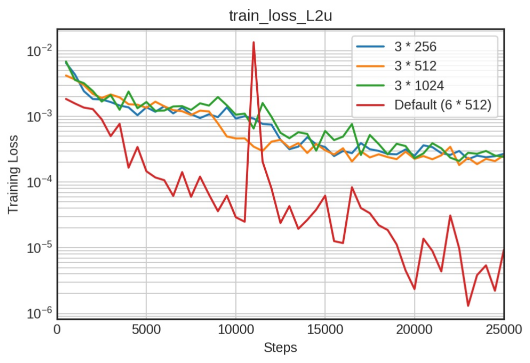
Increasing Number of Nodes

In this section, we will increase the number of nodes per layer by keeping the layer size constant. So as the result of it we get three different combination for each of the layer size and we will analyse how the different losses in comparison to the default combination 6 x 512. We will plot four different plots for each combination - validation loss in p, boundary condition loss in u, L2_continuity loss and overall loss. The time take for each runs for different combinations is given in the table 3.7.

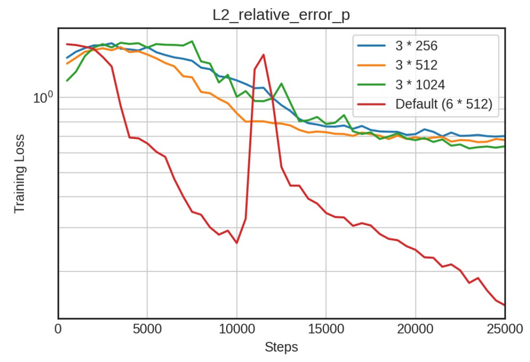
From the Figure 3.23 we see, irrespective of increasing the number of nodes per layer, for the layer size 3, the training converge as compared to the default combination or in other words it takes may take much longer to converge. So layer size 3, irrespective of number of nodes is not a good choice.



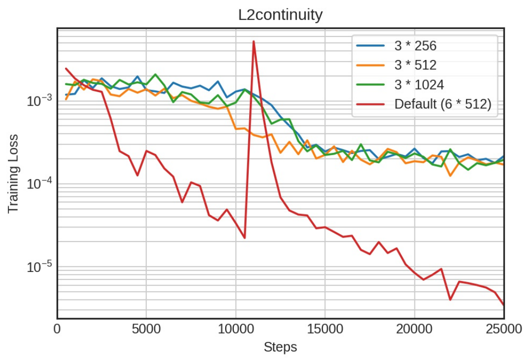
(a) Overall Loss



(b) Boundary condition loss in u



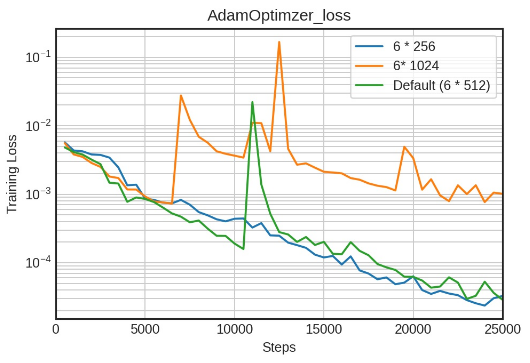
(c) validation loss in p



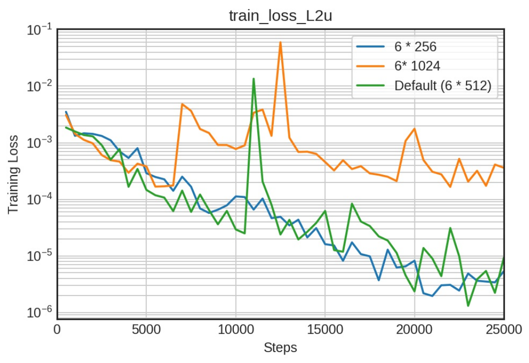
(d) L2 continuity loss

Figure 3.23: Increasing number of nodes for layer size 3

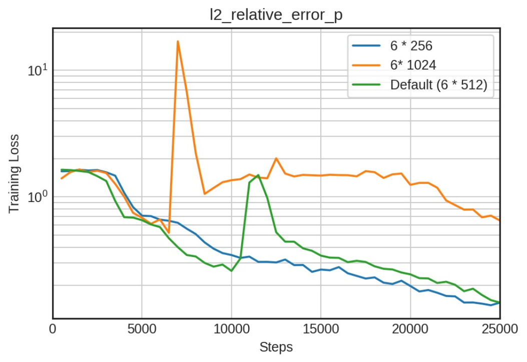
From the Figure 3.24 we see, increasing the number of nodes per layer from the default, doesn't converge or behaves in a unusual way when number of nodes is 1024. But when the number of nodes is 256 per layers, it gives very similar convergence to the default combination and also takes less time compared to default as show in the table 3.7.



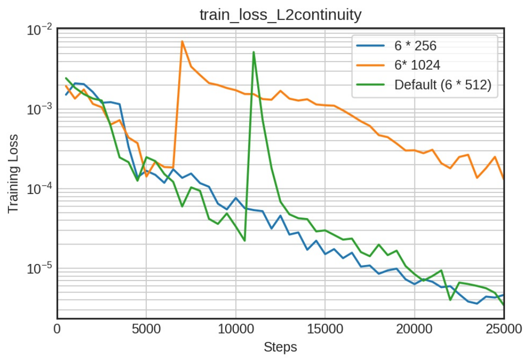
(a) Overall Loss



(b) Boundary condition loss in u



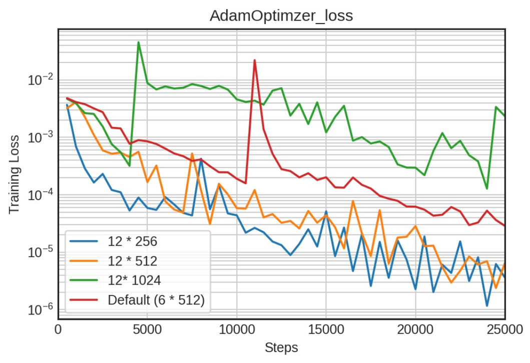
(c) validation loss in p



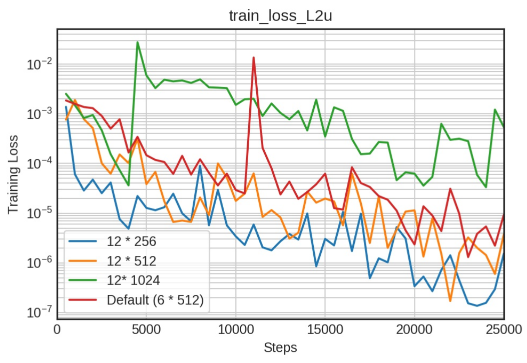
(d) L2 continuity loss

Figure 3.24: Increasing number of nodes for layer size 6

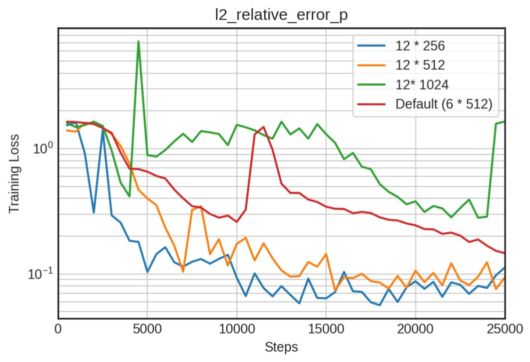
From the Figure 3.25 we see, increasing the layer size and decreasing the number of nodes (12 x 256), gives better convergence than default combination and it takes similar time. The combination 12* 512 also gives the better convergence, but the time taken is twice as much as default combination(6 x 512). Considering time efficient and convergence rate, 12 * 256 is best choice.



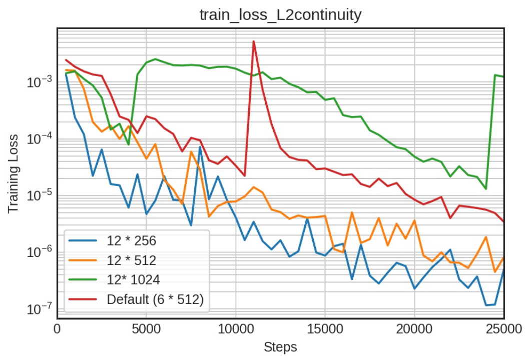
(a) Overall Loss



(b) Boundary condition loss in u



(c) validation loss in p

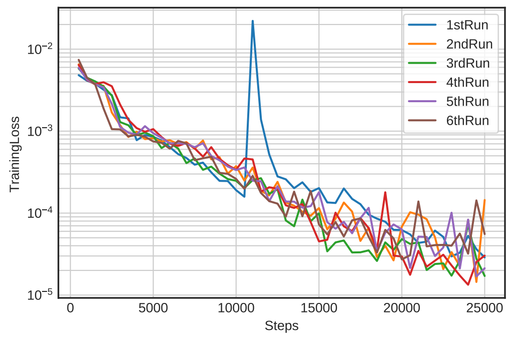


(d) L2 continuity loss

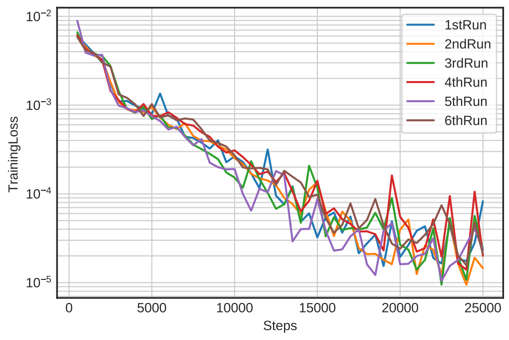
Figure 3.25: Increasing number of nodes for layer size 12

3.2.4 Impact of Random Initialisation

As SimNet uses random initialisation for weights and bias, we tried to run same training with default values multiple times and as you can see in Figure 3.26, there isn't much difference in trends and values, but there are sudden large spikes in Sampling1 and multiple small spikes in Sampling2. Hence we can come to theory that, random initialisation and low number of sampling points together may contribute to large spikes in the overall loss and thereby slow down convergence.



(a) Sampling1 -Multiple runs



(b) Sampling2 -Multiple runs

Figure 3.26: Sampling1 Vs Sampling2 -Random Initialisation study

After all the above studies, we conclude the hyper parameters for our application problem in SimNet. We use 12*256 size, sampling1 set of training points, adaptive activations with swish function and quasi-random settings for the next section.

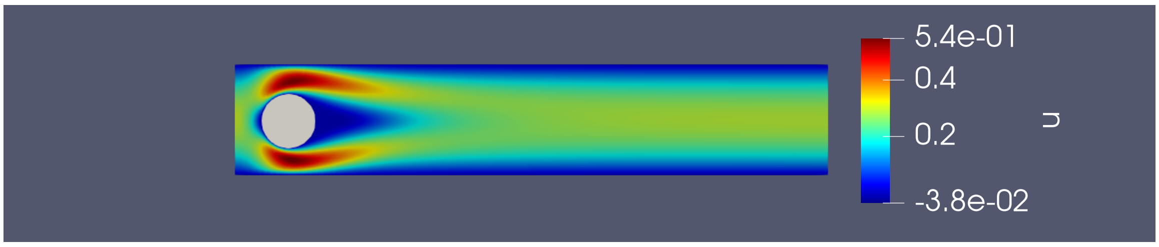
3.3 Application of SimNet

One important advantage of a neural network solver over traditional numerical methods is its ability to solve parameterized geometries [28]. In other words, SimNet allows us to solve problems for multiple design parameters in a single training. This implies that once the training is complete , it is possible to inference on several geometry/physical parameter combinations as a post processing step, without solving the forward problem again. To illustrate this concept we solved a parameterized version of our case study by defining the range as show in table 3.7.

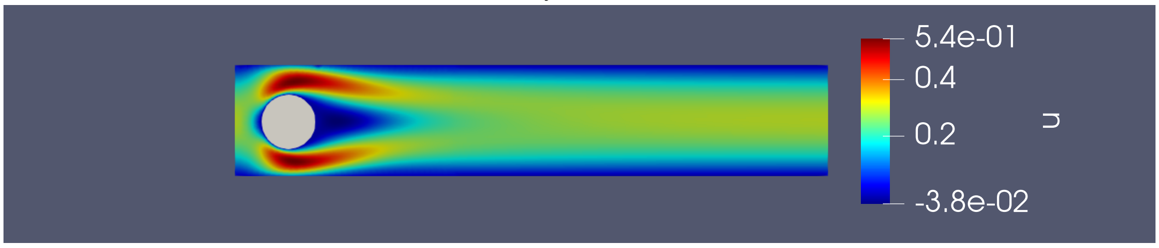
parameter	range	values chosen for validation
cylinder radius - r	0.025 to 0.1	0.025, 0.05, 0.1
Max.inlet velocity - U	0.3 to 0.6	0.3 , 0.45 , 0.6

Table 3.7: Parameter ranges

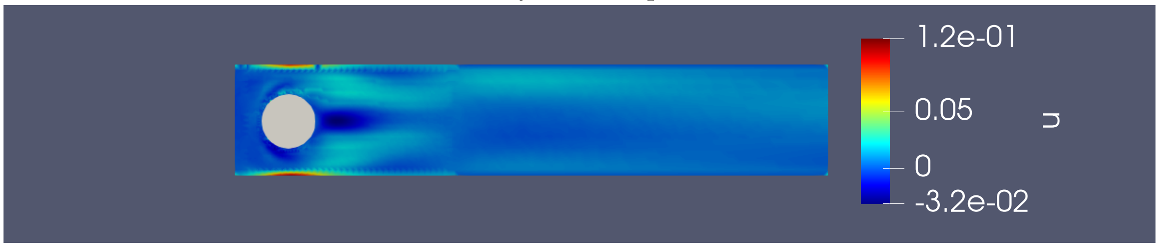
Once we have trained the parameter for a given range, three values are taken for validation. To illustrate the parametrized cylinder radius problem, velocity flow field is plotted for cylinder radius 0.1 in Figure 3.27. The complete plots are available in Section 5.3 of Appendix A.



(a) velocity u from simnet



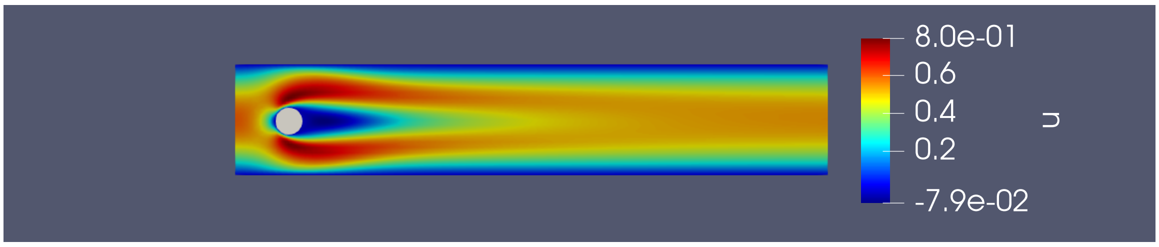
(b) velocity u from OpenFoam



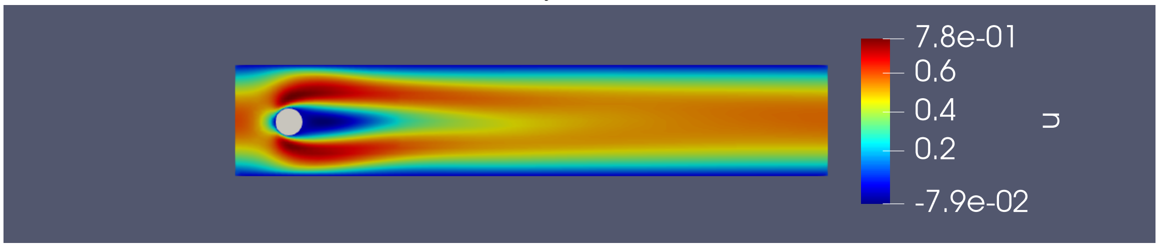
(c) u:difference

Figure 3.27: velocity Magnitude Comparison for radius 0.1

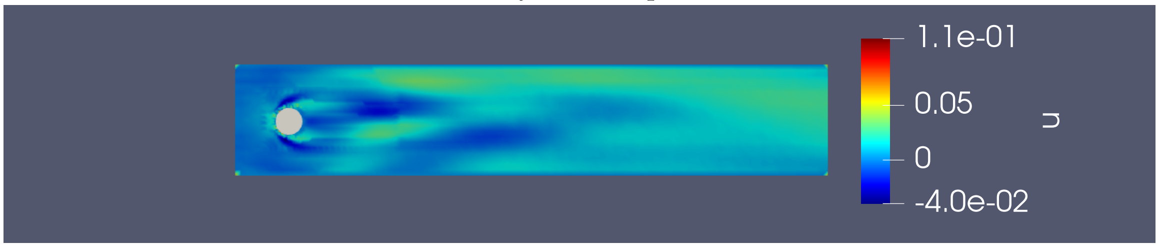
It was mentioned that it is possible to parametrize a physical parameter. So inlet velocity is trained in the range given in table 3.7. Similar to the previous case, velocity flow field for inlet velocity 0.6 is plotted in figure 3.28 to demonstrate the parametrized solution. The complete plots are available in Section 5.3 of Appendix A.



(a) velocity u from simnet



(b) velocity u from OpenFoam



(c) u:difference

Figure 3.28: velocity Magnitude Comparison for U = 0.6

The results obtained from SimNet for all other values chosen for validation are in good accordance with the OpenFoam results. As best hyperparameters are chosen from the analysis to solve our parametrized problem, now the study of convergence rate and time taken for a parametrized problem is compared with non parametrized problem with same hyper parameters.

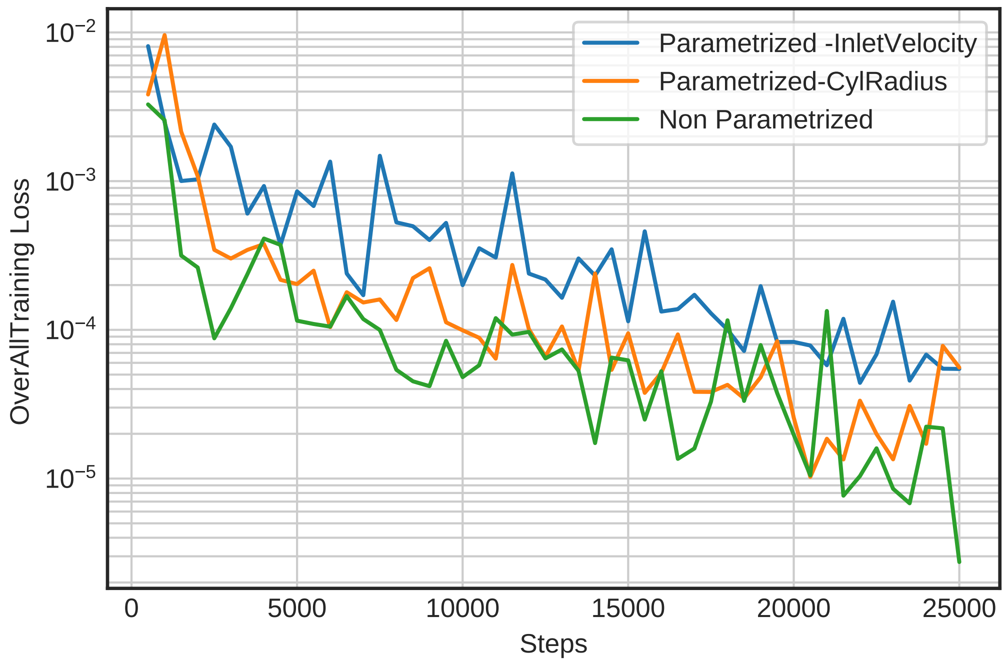


Figure 3.29: AdamOptimizer Loss - Comparison

It is seen from the Figure 3.29 that parametrized problem convergence rate comparatively slower than the non-parametrized problem. The table 3.8 gives the details about training time and loss value. It is understood that parametrized problem takes little more training time.

	Parametrized-r	Parametrized-U	Non-parametrized
training time(minutes)	66	64	57
Overall Loss	5.58e-5	5.45e-5	2.75e-6

Table 3.8: Training time and Loss values comparison

4 Conclusion

4.1 Discussion

Based on the case study on Navier stokes equation, Physics informed neural networks were constructed using NVIDIA's SimNet (version 20.12) and different parameters influencing the convergence are studied. It is found that Swish activation function along with adaptive activations work the best and also it is analysed that the combination of 12 * 256 of network size gives better convergence. Quasi random sampling helps to improve convergence for Sampling1 set of training points. Then all those best parameters are chosen for the parameterized problem and the results are compared with OpenFoam solutions and results are in good accordance with OpenFoam.

In this study, the training data used are randomly selected in the space domain, hence the PINNs does not need to consider the discretization of Navier stokes equation and can predict solutions from a little amount of data. But in solvers like OpenFoam which uses Finite Volume methods, discretization of equations are done. It is also learned that PINNs are mesh free and meshing cost can be avoided in case of complex geometries.

4.2 Conclusion and Outlook

Because SimNet tool is GPU integrated AI simulation, Multi GPU training can be done which enables us to reduce the time taken for training. Apart from the parametrised problem , SimNet has other advantages such a solving the inverse problems, creating geometry for real world problems to predict solutions and also solving design optimization problem [21].

The SimNet achieves good results due to the powerful function approximation ability incorporated using PINNs. In the future, physics-informed neural networks will create a big influence in the study of solving partial differential equations and even much more tools like SimNet will be developed on the field of scientific computing.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," 2017.

[2] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv:1909.11942*, 2019.

[3] J. Jiang, C. Dun, T. Huang, and Z. Lu, "Graph convolutional reinforcement learning," *arXiv:1810.09202*, Oct,2019.

[4] J. He, L. Li, J. Xu, and C. Zheng, "Relu deep neural networks and linear finite elements," July 2018.

[5] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *arXiv e-prints, page physics/9705023*, May 1997.

[6] C. Beck and A. Jentzen, "Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equation," *arXiv:1709.05963*, 2017.

[7] J. Sirignano and K. Spiliopoulos, "Dgm: A deep learning algorithm for solving partial differential equations," *arXiv:1708.07469*, 2017.

[8] D. Brittz, "http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/," *blog*, 2015.

[9] S. Course, "Cs231n: Convolutional neural networks for visual recognition," *Course Material*, 2017.

[10] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *The Journal of Machine Learning Research, 18*, (2017).

[11] D. Paper, "Data science fundamentals for python and mongodb," *Apress*, 2018.

[12] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research, vol. 12, pp. 2121–2159*, 2010.

[13] B. Ginsburg, P. Castonguay, O. Hrinchuk, O. Kuchaiev, R. Leary, V. Lavrukhin, J. Li, H. Nguyen, Y. Zhang, and J. M. Cohen, "Training deep networks with stochastic gradient normalized by layerwise adaptive second moments," *arXiv:1905.11286*, 2019.

[14] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The rprop algorithm," *IEEE International Conference on Neural Networks pp. 586–591, IEEE*, 1993.

[15] D. P. Kingma and J. B. Adam, "A method for stochastic optimization," *arXiv:1412.6980*, 2014.

[16] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part i): data-driven solutions of nonlinear partial differential equations," *arXiv preprint arXiv:1711.10561*, (2017).

[17] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part ii): data-driven solutions of nonlinear partial differential equations," *arXiv preprint arXiv:1711.10566*, (2017).

[18] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics, 378 , pp. 686–707*, (2019).

[19] M. rassi, "https://github.com/maziarraissi/ pinns,"

[20] L. LU, X. MENG, Z. MAO, and G. E. KARNIADAKIS, "Deepxde: A deep learning library for solving differential equations," *arXiv : 1907.04502v2*, (2019).

[21] O. Hennigh, S. Narasimhan, M. A. Nabian, A. Subramaniam, K. Tangsali, M. Rietmann, J. del Aguila Ferrandis, W. Byeon, Z. Fang, and S. Choudhry, "Nvidia simnet: An ai-accelerated multi-physics simulation framework," (DECEMBER 16, 2020).

[22] NVIDIA, "Simnet0.2 userguide," *SimNEt Documentation*, 2020.

[23] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv:1511.07289*, 2015.

[24] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiterr, "Self-normalizing neural networks," *arXiv:1706.02515*, 2017.

[25] A. D. Jagtap, K. Kawaguchi, and G. E. Karniadakis, "Adaptive activation functions accelerate convergence in deep and physics-informed neural networks," *Journal of Computational Physics, 404:109136*, 2020.

[26] J. H. Halton, "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numerische Mathematik,2(1):84–90*, 1960.

[27] S. Turek, "Benchmark computations of laminar flow around cylinder; in flow simulation with high-performance computers ii," *Notes on Numerical Fluid Mechanics 52, 547-566*, 1996.

[28] L. Sun, H. Gao, S. Pan, and J.-X. Wang, "Surrogate modeling for fluid flows based on physics constrained deep learning without simulation data," *Computer Methods in Applied Mechanics and Engineering 361:112732*, 2020.

[29] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," *arXiv:1905.02244*, 2019.

List of Figures

List of Tables

5 Appendix A

5.1 Sample Python Code

```python
from sympy import Symbol ,Eq
from simnet.pdes import PDES
from sympy import Symbol
import numpy as np
from simnet.csv_utils.csv_rw import csv_to_dict
from simnet.solver import Solver
from simnet.dataset import TrainDomain , ValidationDomain
from simnet.data import  Validation
from simnet.PDES import NavierStokes , IntegralContinuity
from simnet.sympy_utils.functions import parabola
from simnet.sympy_utils.geometry_2d import Circle ,Channel2D , Line
from simnet.controller import SimNetController

channel_length = (0, 2.2)
channel_width = (0, 0.41)
channel = Channel2D((channel_length[0], channel_width[0]),
                    (channel_length[1], channel_width[1]))
centre = (0.2,0.2)
inlet_vel = 0.3


circle = Circle((centre[0],centre[1]), 0.05)


geo = channel -  circle

inlet = Line((channel_length[0], channel_width[0]),
                (channel_length[0], channel_width[1]), 1)
outlet = Line((channel_length[1], channel_width[0]),
                (channel_length[1], channel_width[1]), 1)


```

```
# params for domain

# define sympy varaibles to parametize domain curves
 # sympy variables
x , y = Symbol('x') , Symbol('y')

class NavierTrain(TrainDomain):
  def __init__(self, **config):
    super(NavierTrain, self).__init__()


    #initial condition

    #boundary conditions

    # inlet
    parabola_sympy = parabola(y, inter_1=channel_width[0],
                                 inter_2=channel_width[1],
                                height=inlet_vel)
    inletBC = inlet.boundary_bc(outvar_sympy={'u': parabola_sympy ,'v':0},
                                 batch_size_per_area=49)
    self.add(inletBC, name="Inlet")

    outletBC = outlet.boundary_bc(outvar_sympy={'p': 0},batch_size_per_area=49)
    self.add(outletBC, name="Outlet")

    topWall = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                              batch_size_per_area=45,
                              lambda_sympy={'lambda_u': 1.0  ,
                                            'lambda_v': 1.0},
                              criteria=Eq(y,0.41)
                             )

    self.add(topWall, name="TopWall")

    # no slip
    bottomWall = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                                 batch_size_per_area=45,
                                 criteria=Eq(y,0))
    self.add(bottomWall, name="NoSlip")


    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0,
                                 'momentum_x': 0,
                                'momentum_y': 0},
                                bounds=
                                {x: (channel_length[0],channel_length[1]),
                               y: (channel_width[0],channel_width[1])},
                               lambda_sympy={'lambda_continuity': geo.sdf,
                               'lambda_momentum_x': geo.sdf,
                              'lambda_momentum_y': geo.sdf},
                                batch_size_per_area=4095

                                   )


    self.add(interior, name="Interior")

    ObstacleCont = circle.boundary_bc(outvar_sympy={'u': 0, 'v':0},
                                       batch_size_per_area=172,

                                       lambda_sympy={'lambda_u': 1.0 ,
                                       'lambda_v':1.0})

    self.add(ObstacleCont, name="Obstacle")

    outletCont = outlet.boundary_bc
                                (outvar_sympy={'integral_continuity': 0.082},
                               batch_size_per_area=45,
                              lambda_sympy=
                           {'lambda_integral_continuity': 0.1})
    self.add(outletCont, name="IntegralContinuity1")




#validation data
mapping = {'Points_0':'x','Points_1':'y','U_0':'u','U_1':'v','p':'p'}
openfoam_var = csv_to_dict('Cylinder_OpenFoam_mesh1.csv', mapping)
openfoam_invar_numpy = {key: value for key, value in openfoam_var.items()
                                                 if key in ['x', 'y']}
openfoam_outvar_numpy = {key: value for key, value in openfoam_var.items()
                                               if key in ['u', 'v','p']}
```

```

class NavierVal(ValidationDomain):
  def __init__(self, **config):
    super(NavierVal, self).__init__()
    val = Validation.from_numpy(openfoam_invar_numpy, openfoam_outvar_numpy)
    self.add(val, name='Val')

# Define neural network
class NavierSolver(Solver):
  train_domain = NavierTrain
  val_domain = NavierVal
  #inference_domain = NavierInference


  def __init__(self, **config):
    super(NavierSolver, self).__init__(**config)

    self.equations = NavierStokes(nu=0.001, rho=1.0, dim=2,
                                  time=False).make_node()
                                    + IntegralContinuity().make_node()


    heat_net = self.arch.make_node(name='heat_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    self.nets = [heat_net]

  @classmethod # Explain This
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_NavierMesh1',
        'max_steps': 50000,
        'decay_steps': 1000
        })

if __name__ == '__main__':
  ctr = SimNetController(NavierSolver)
  ctr.run()
```

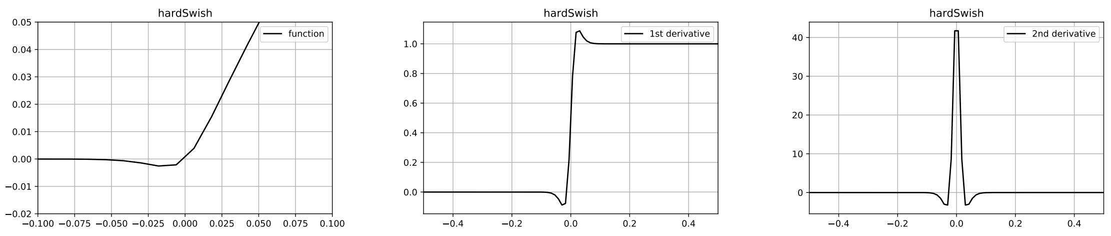
5.2 Hard Swish

Hard Swish is a type of activation function based on Swish, but replaces the computationally expensive sigmoid with a piecewise linear analogue [29] which can be written as

$$hard - swish(x) = x * \frac{RELU6(x+3)}{6}. \tag{5.1}$$

But in SimNet , the hard swish is formulated differently as shown in the equation below. The plots of functon and it derivatives are shown in Figure 5.1.

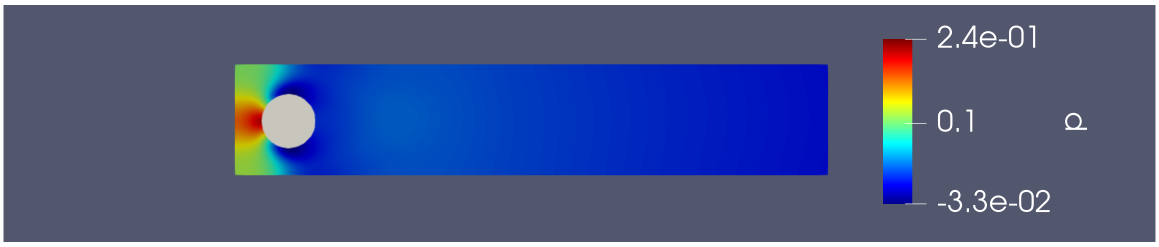
$$h - swish(x) = x * sigmoid(100 * x) \tag{5.2}$$



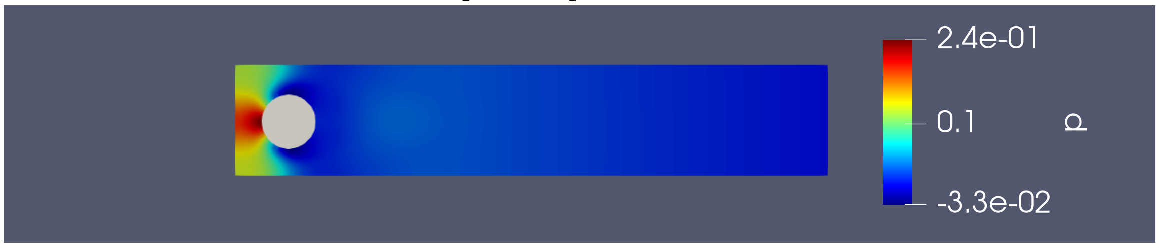
(a) Hard swish (b) Hard swish - 1st derivative (c) Hard swish - 2nd derivative

Figure 5.1: Hard swish and its derivatives in SimNet

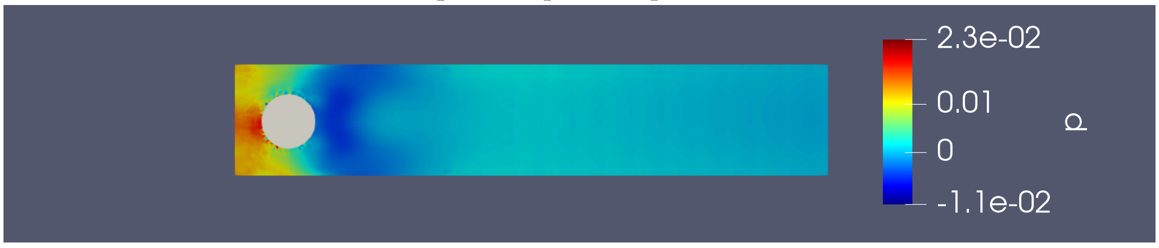
5.3 Parametrized Problem - solution plots



(a) pressure p from simnet

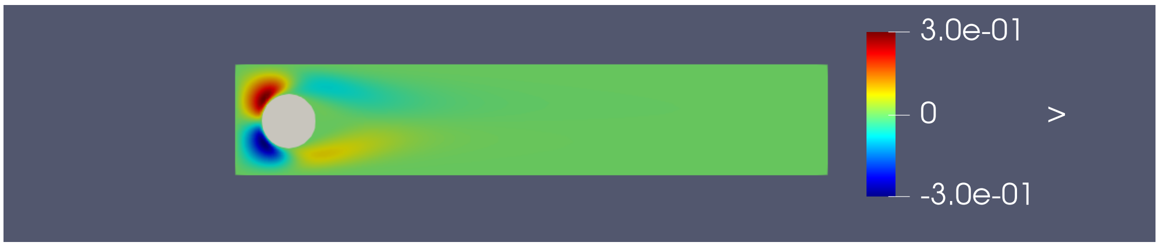


(b) pressure p from OpenFoam

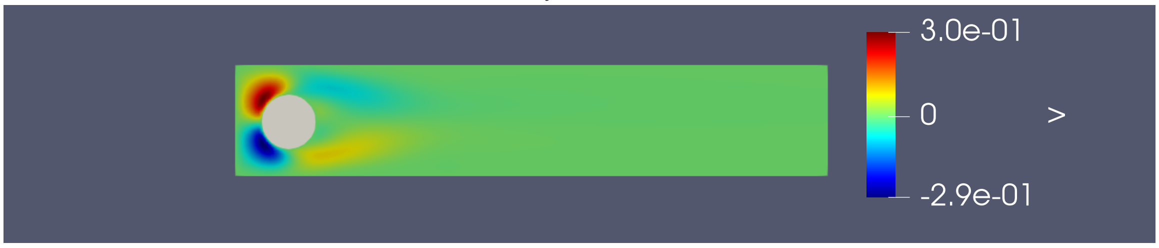


(c) p:difference

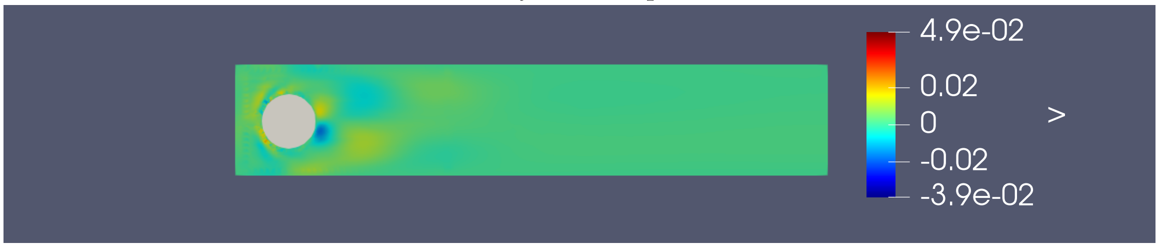
Figure 5.2: Pressure Comparison for radius 0.1



(a) velocity v from simnet

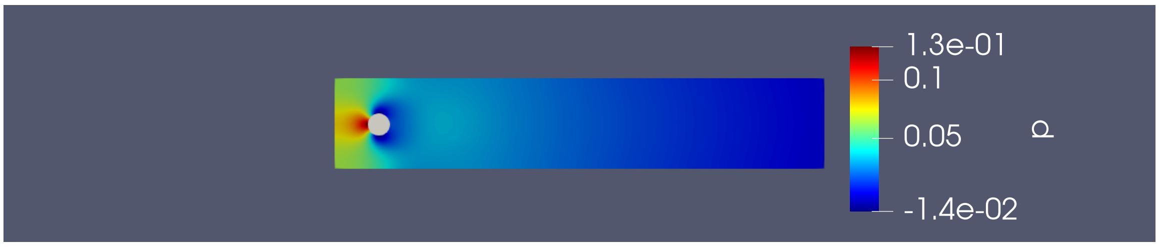


(b) velocity v from OpenFoam

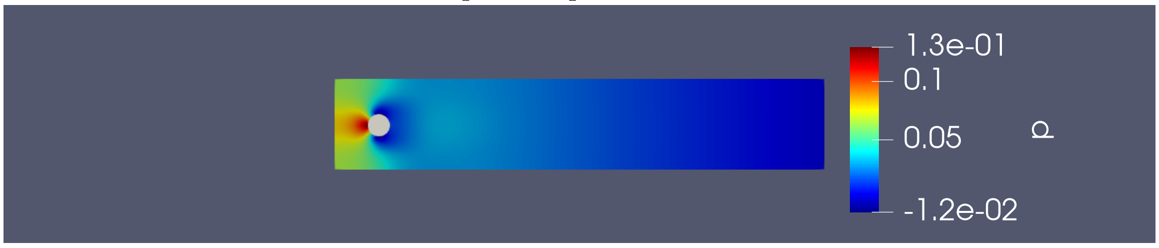


(c) v:difference

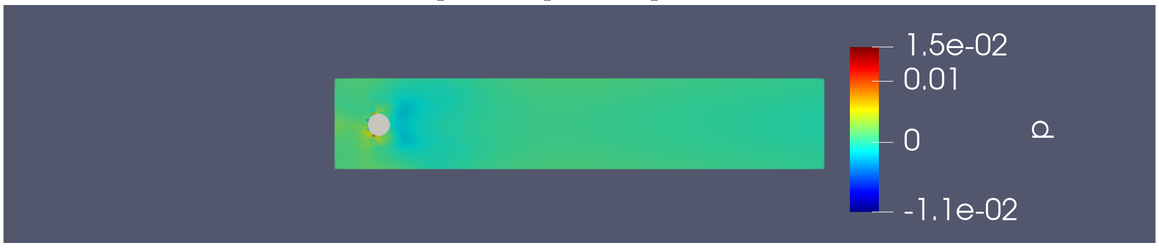
Figure 5.3: velocity v for radius 0.1



(a) pressure p from simnet

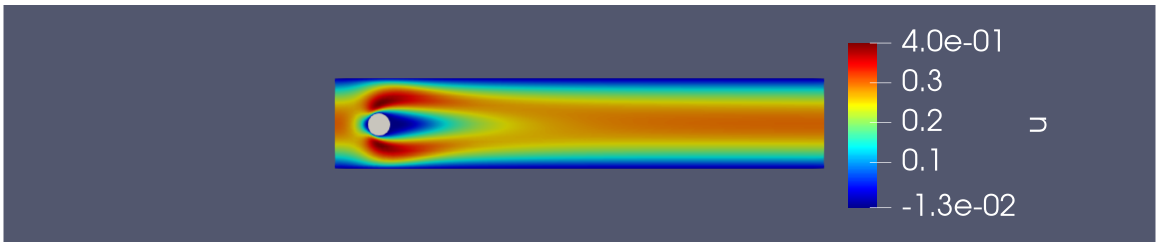


(b) pressure p from OpenFoam

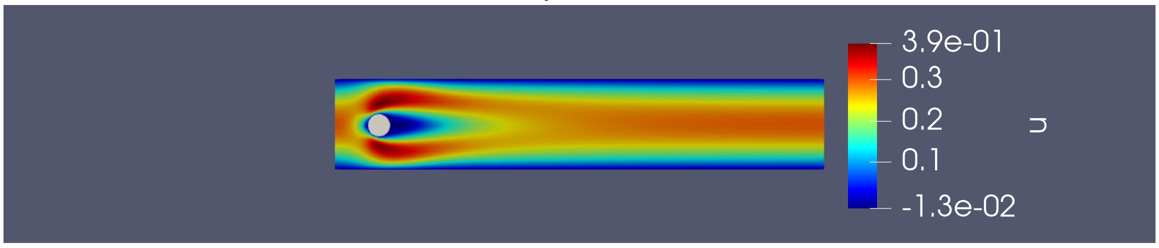


(c) p:difference

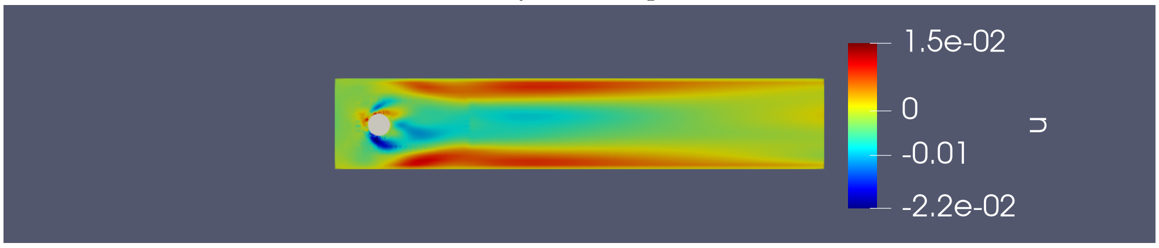
Figure 5.4: Pressure Comparison for radius 0.05



(a) velocity u from simnet

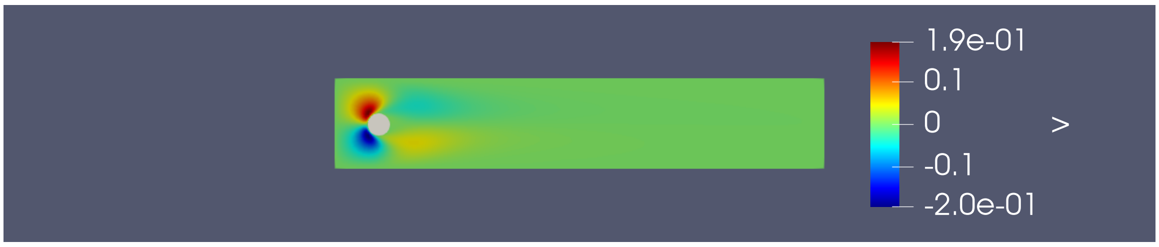


(b) velocity u from OpenFoam

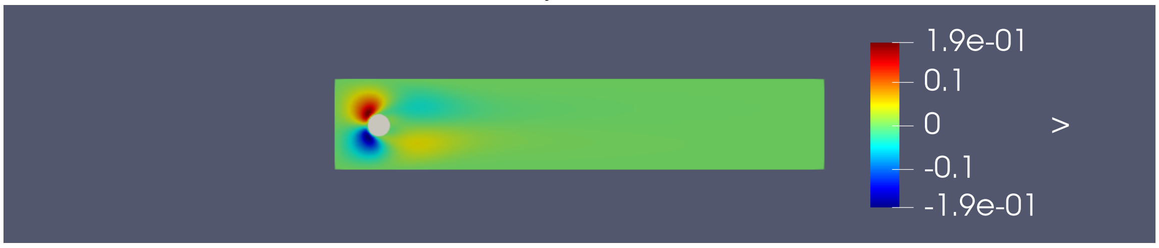


(c) u:difference

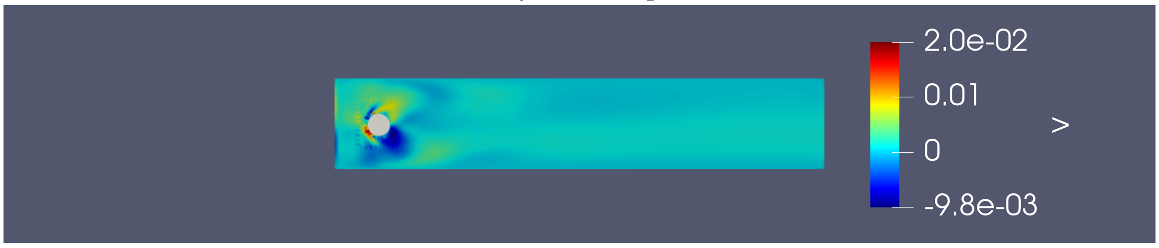
Figure 5.5: velocity Magnitude Comparison for radius 0.05



(a) velocity v from simnet

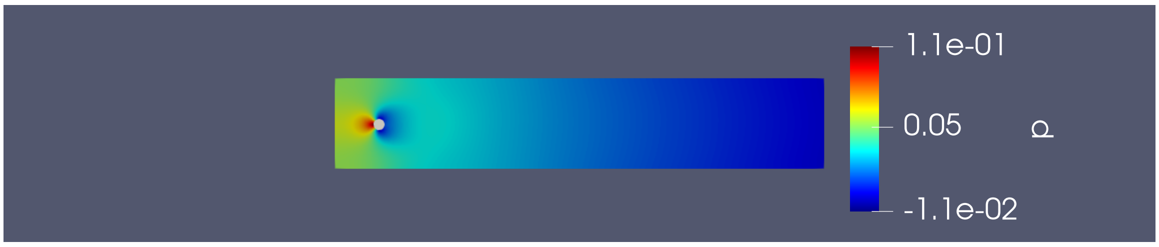


(b) velocity v from OpenFoam

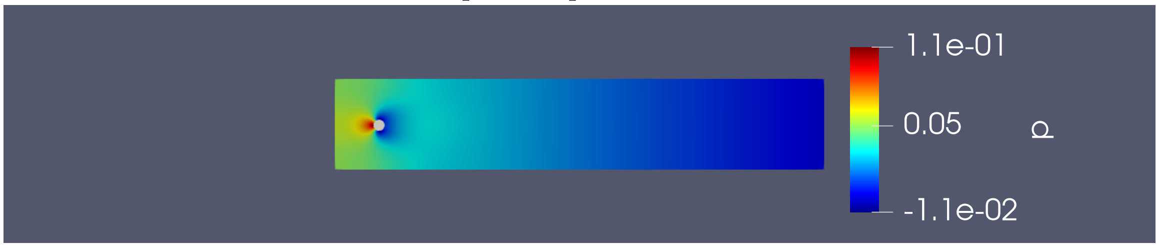


(c) v:difference

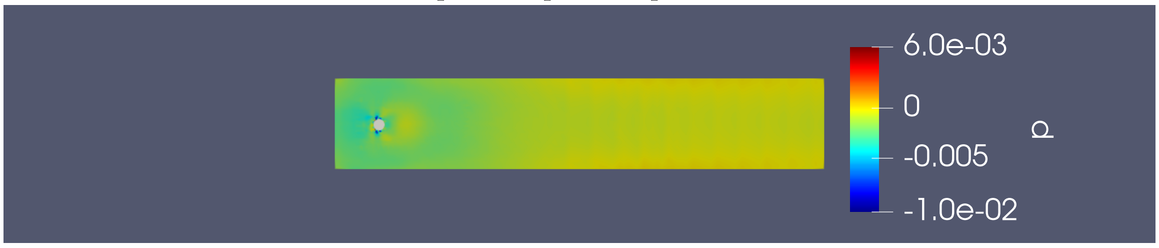
Figure 5.6: velocity v for radius 0.05



(a) pressure p from simnet

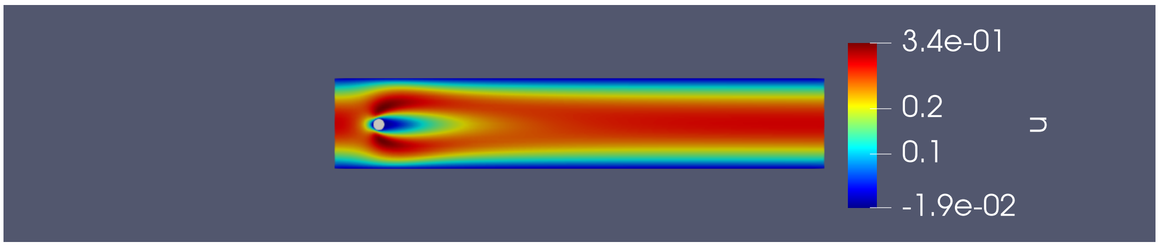


(b) pressure p from OpenFoam

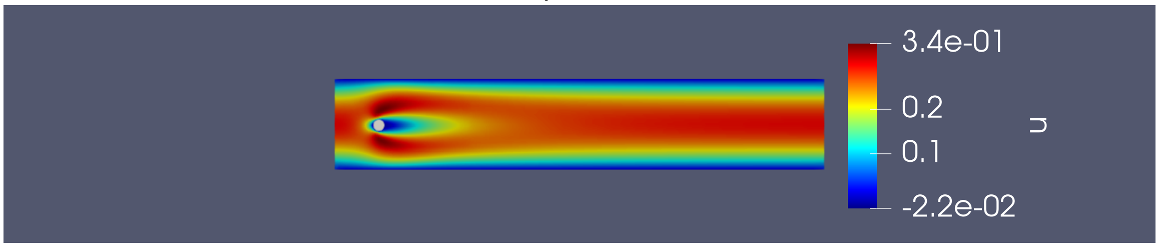


(c) p:difference

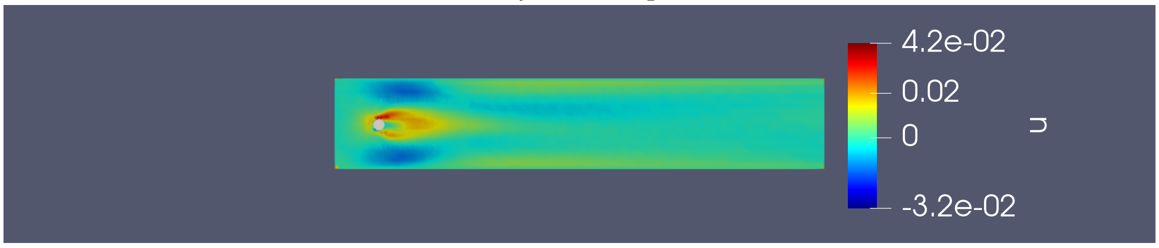
Figure 5.7: Pressure Comparison for radius 0.025



(a) velocity u from simnet

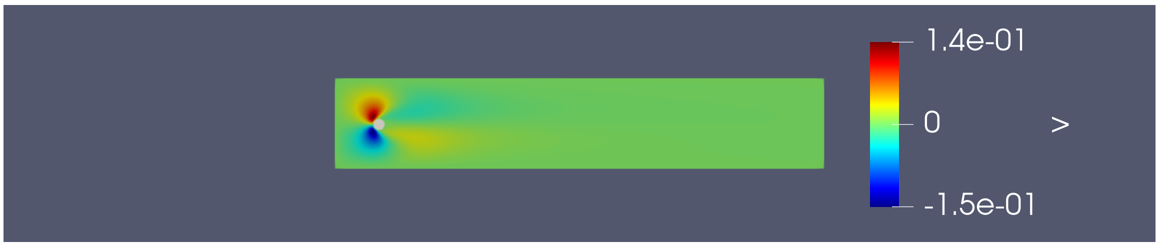


(b) velocity u from OpenFoam

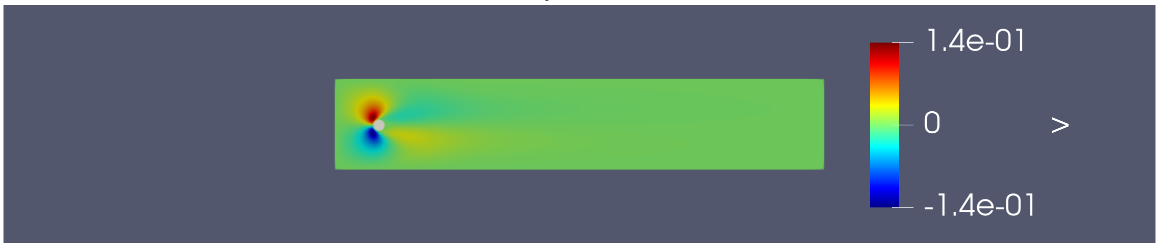


(c) u:difference

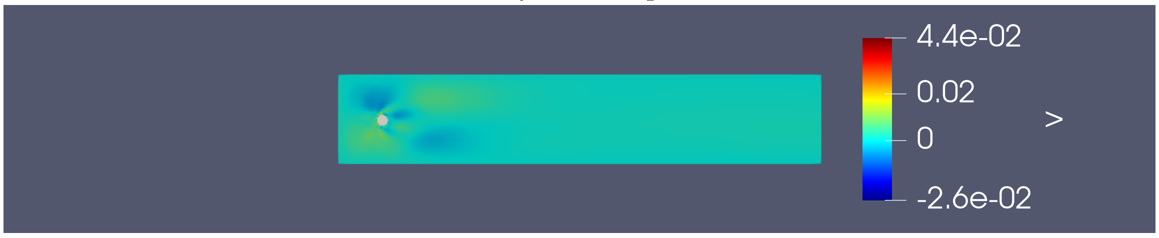
Figure 5.8: velocity Magnitude Comparison for radius 0.025



(a) velocity v from simnet

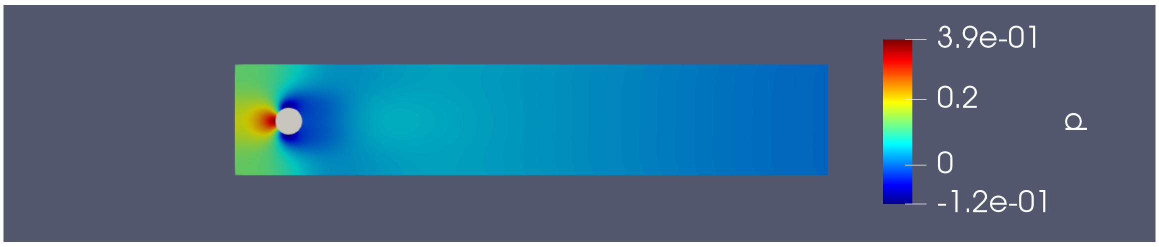


(b) velocity v from OpenFoam

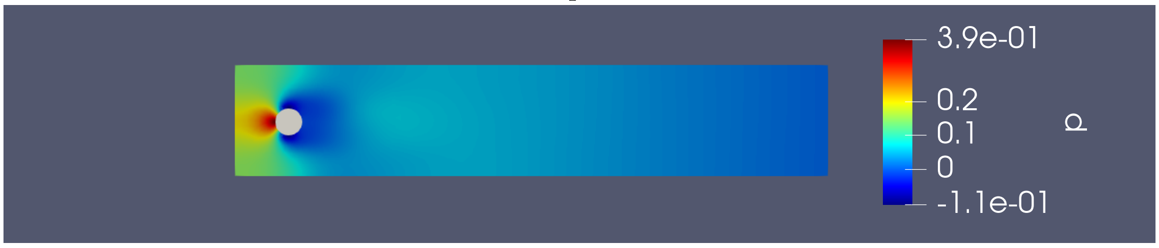


(c) v:difference

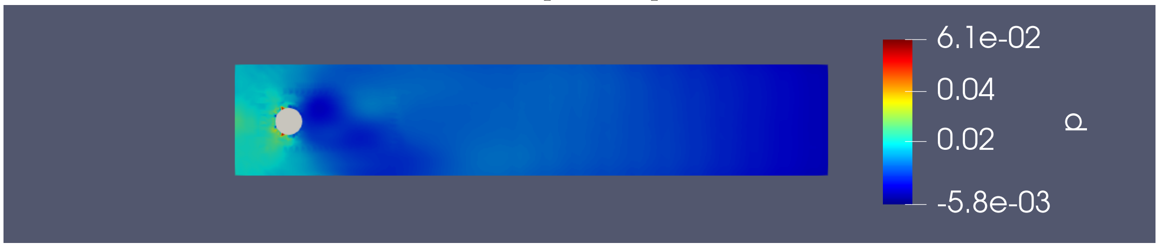
Figure 5.9: velocity v for radius 0.025



(a) Pressure p from simnet

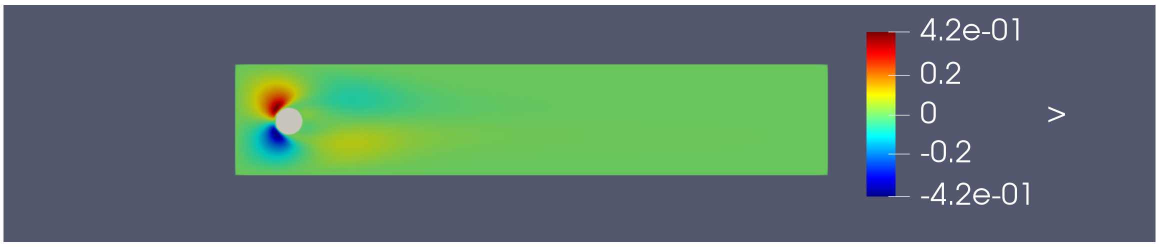


(b) Pressure p from OpenFoam

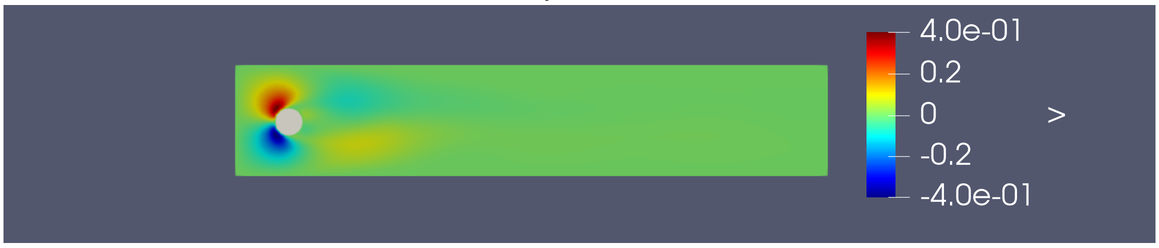


(c) p:difference

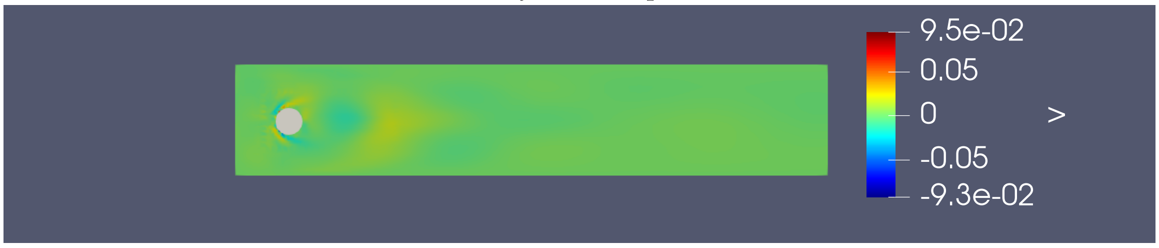
Figure 5.10: Pressure p Comparison for U = 0.6



(a) velocity v from simnet

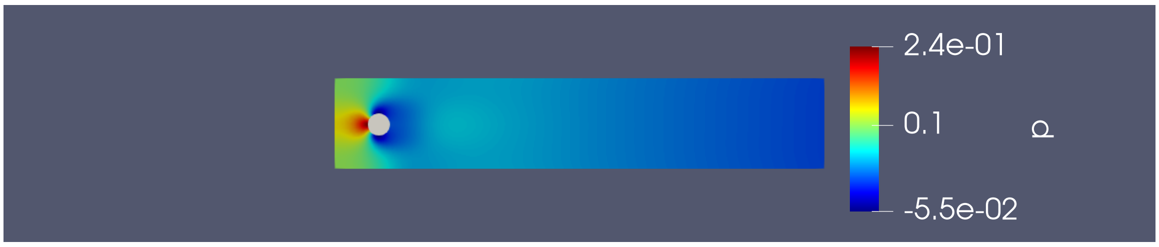


(b) velocity v from OpenFoam

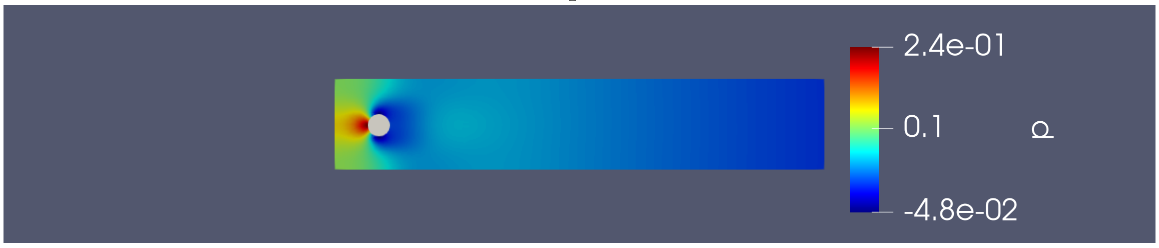


(c) v:difference

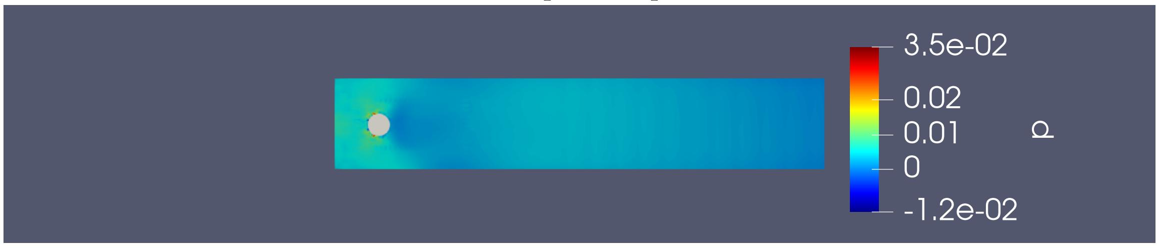
Figure 5.11: velocity v for U = 0.6



(a) Pressure p from simnet

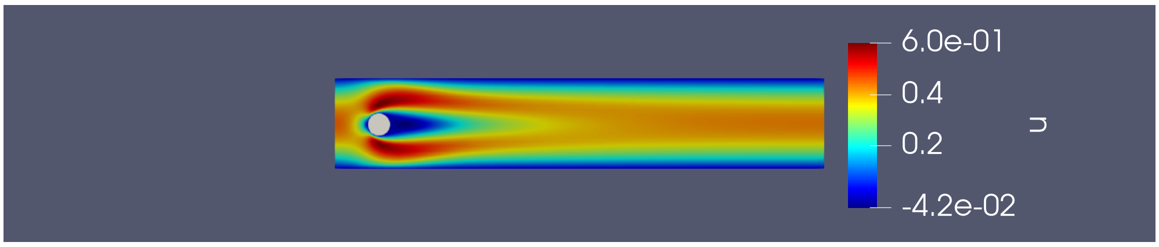


(b) Pressure p from OpenFoam

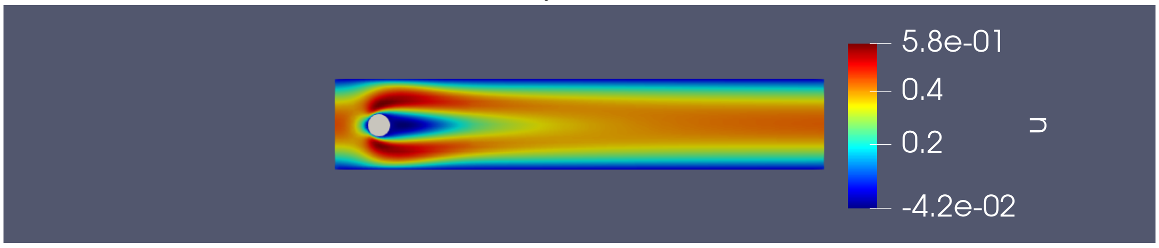


(c) p:difference

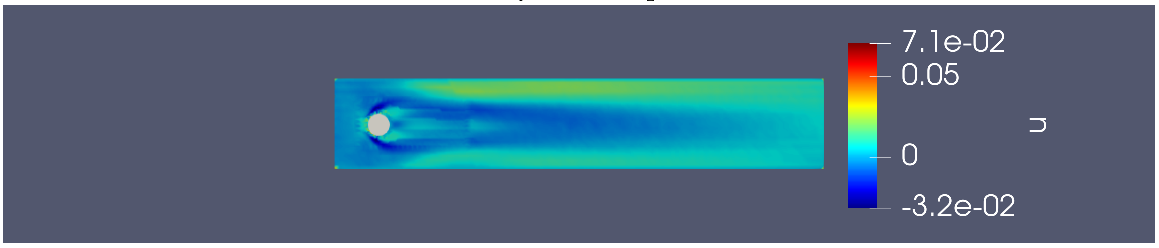
Figure 5.12: Pressure p Comparison for U = 0.45



(a) velocity u from simnet

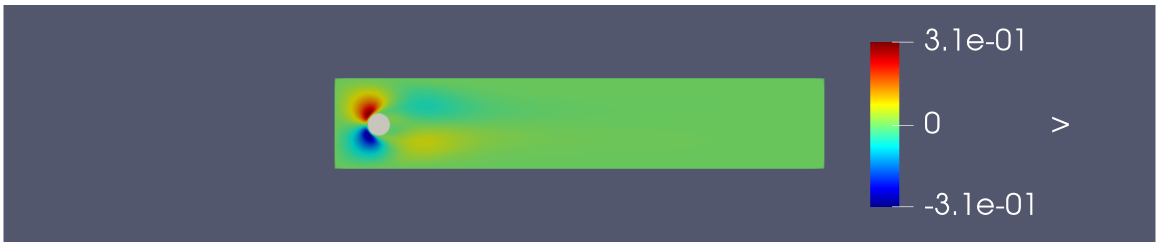


(b) velocity u from OpenFoam

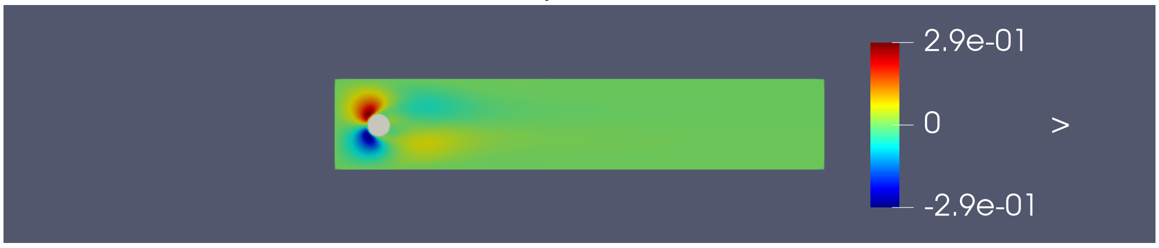


(c) u:difference

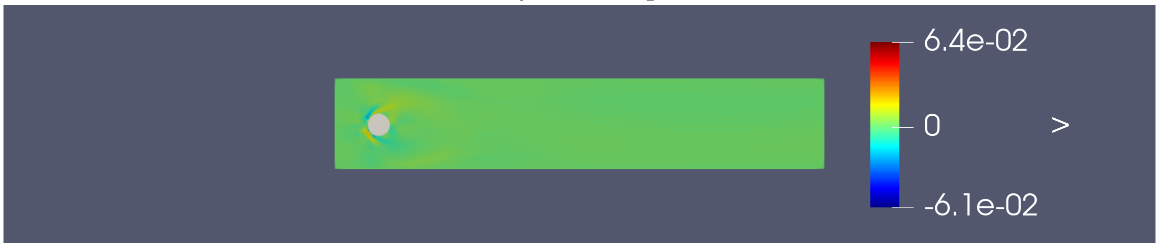
Figure 5.13: velocity Magnitude Comparison for U = 0.45



(a) velocity v from simnet

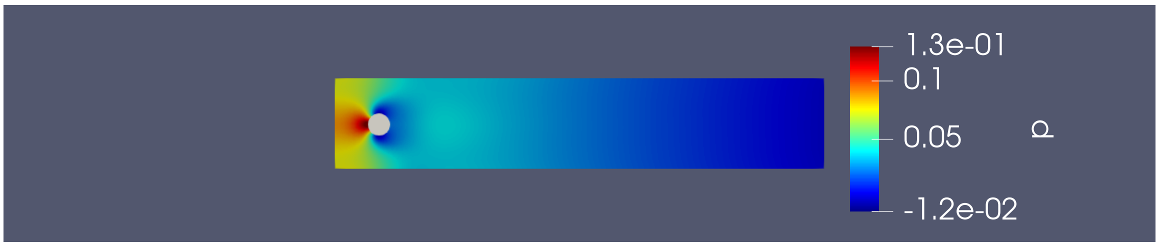


(b) velocity v from OpenFoam

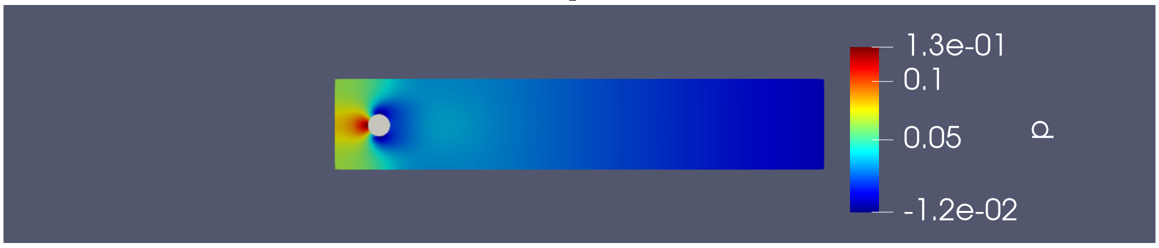


(c) v:difference

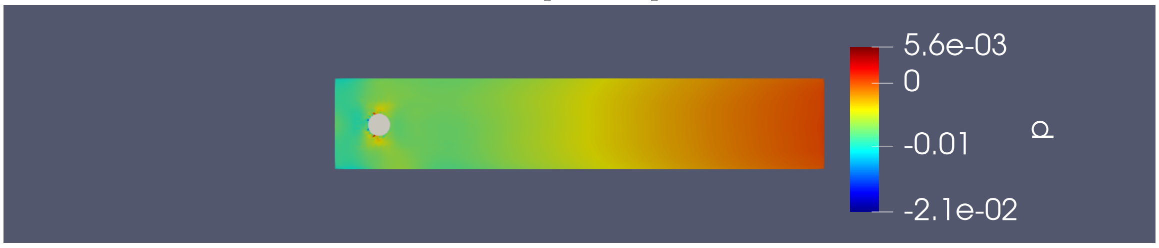
Figure 5.14: velocity v for U = 0.45



(a) Pressure p from simnet

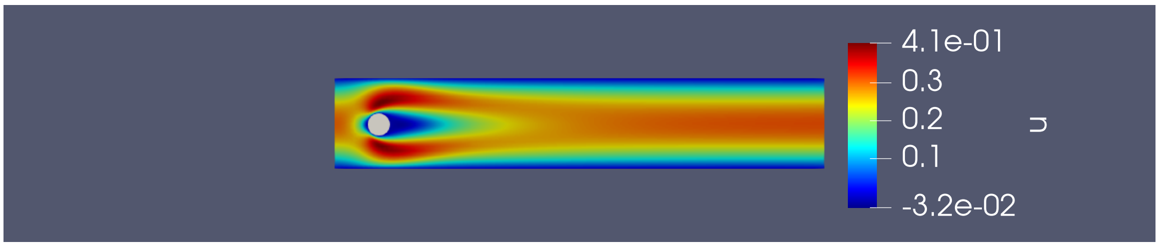


(b) Pressure p from OpenFoam

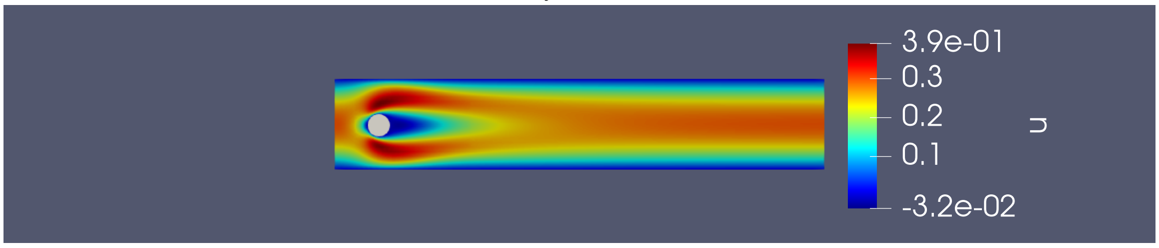


(c) p:difference

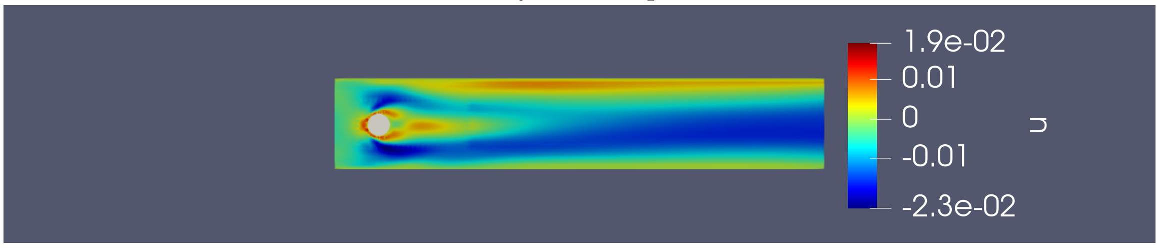
Figure 5.15: Pressure p Comparison for U = 0.3



(a) velocity u from simnet

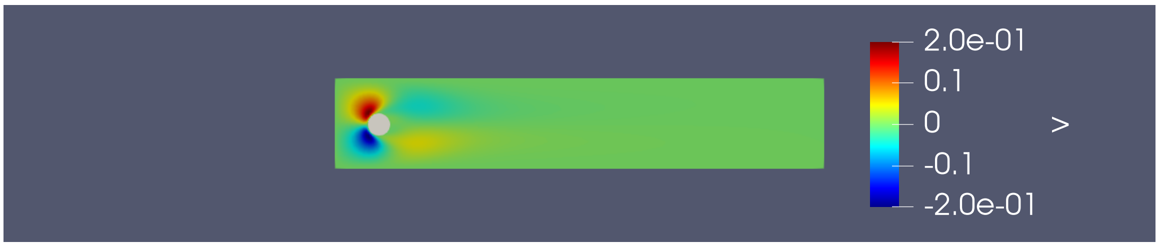


(b) velocity u from OpenFoam

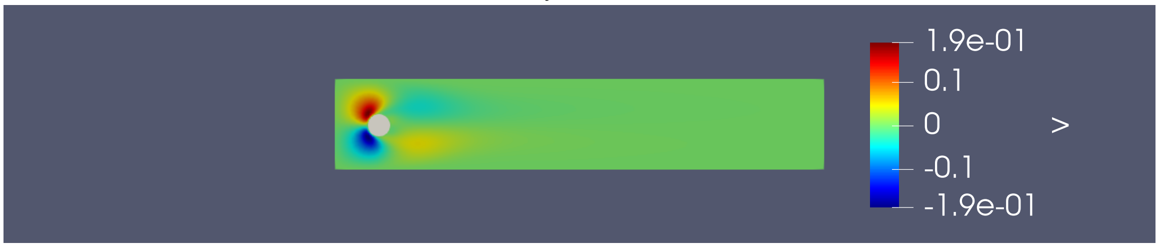


(c) u:difference

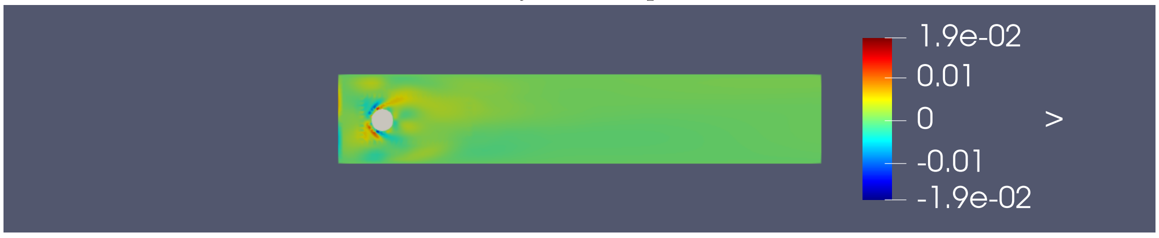
Figure 5.16: velocity Magnitude Comparison for U = 0.3



(a) velocity v from simnet



(b) velocity v from OpenFoam



(c) v:difference

Figure 5.17: velocity v for U = 0.3