Industrial Training

Navier-Stokes equations approximation through Physics Informed Neural Networks (PINN)

Final Internship Report

Submitted By: **Gaurav Bokil**

Student, Masters Programme in Computational Mechanics

Supervised by: **Dr. Arghir Dani Zarnescu**

Leader of Applied Analysis research group,
Basque Center for Applied Mathematics (BCAM)

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Barcelona, August 2021

Acknowledgement

I would like to extend my gratitude to **Dr. Arghir Dani Zarnescu**, Group leader of "Applied Analysis" team and **Dr. Luis Vega**, Group leader of "Linear and Non-linear Waves" team in the research area of "Analysis of PDEs" for providing me with the opportunity to learn and work with them as a Masters intern. It was due to their guidance and the help from my brilliant colleagues **Dr. Jamie Taylor**, **Razvan Ceuca** and **Dr. Sergei Lakunin** that I was able to grasp the complexities of the research project and thus successfully complete my internship.

I would also like to thank **Dr. Ricardo Rossi** and **Mrs. Lelia Zielonka** for providing me with the necessary support and assistance in fulfilling my internship.

To this end, I also thank **Universitat Politècnica de Catalunya (UPC)**, **International Centre for Numerical Methods and Engineering (CIMNE)** and **Basque Centre for Applied Mathematics (BCAM)** for making this practical experience possible.

Sincerely,

Gaurav Bokil

Student, MSc. in Computational Mechanics

CONTENTS

Objective

The goal of the undertaken project, was to approximate the Navier-Stokes equations in fluid dynamics using Physics Informed Neural Networks (PINNs).

Introduction

In Engineering and Physics, Partial Differential Equations (PDEs) are solved by numerical techniques like Finite Element, Finite Volume, Finite Difference methods, etc. A new method to solve PDEs is using an Artificial Neural Network (ANN) as a solution approximator. Physics Informed Neural Networks (PINNs) [1] are ANNs that are "trained" in order to find a solution that satisfies a specific physical law which is defined often by ODEs or PDEs. ANNs are generally used to find relations between input and output using obtained data. The output (u) of a feed forward Neural Network i.e., a Multilayer Perceptron (MLP) is defined as a composition of an activation function (σ) applied to the input (x) with W, b parameters multiplying and adding to (x) respectively as such:

$$u = \ldots \sigma(W_{k-2} * \sigma(W_{k-1} * \sigma(W_k * x + b_k) * x + b_{k-1}) * x + b_{k-2}) \ldots.$$

In this project, the solution is approximated using the PDE and boundary conditions, but not experimental data. Neural Networks have a special property which is described in the "Universal Approximation Theorem". The Universal Approximation Theorem indicates that under minor constraints on the activation function, every continuous function on a compact set can be approximated by a Multilayer Perceptron (MLP). Thus, NN can fundamentally approximate any continuous function which would be a solution of differential equations.

A NN is created using an activation function σ and hyperparameters W, b with inputs $X = [x, t]$, where $x = spatial\ dimensions\ and\ t = time$, in case of evolutionary equations. The output dimensions of the NN are the number of unknowns in the equations that are to be solved. In this work, PDEs are prominently considered. Consider a general evolutionary PDE given in the form,

$$\begin{aligned} &\left(\frac{\partial u}{\partial t}\right) + \mathcal{L}(u) = 0 && in\ \Omega && where, \\ &u_{t=0} = u_0 && in\ \Gamma_{\mathrm{IC}} && \Omega\ = computational\ domain \\ &u = g && in\ \Gamma_{\mathrm{D}} && \Gamma_{IC} = Boundary\ (T = 0) \\ & && && \Gamma_D\ = Dirichlet\ Boundary \end{aligned}$$

The approximate solution thus will be given by the NN output:

$$u_{NN}(X, \theta) = \ldots \sigma(W_{k-2} * \sigma(W_{k-1} * \sigma(W_k * X + b_k) * X + b_{k-1}) * X + b_{k-2}) \ldots.$$

$$where, k = 1,2,3, \ldots M\ layers$$

The goal is to obtain the hyperparameters $\theta = [W, b]$ such that that $u_{NN} \approx u$. The standard paradigm of *training* the neural network is to minimize the loss between the output of NN and the desired target to obtain θ using optimization methods. In this case since the solution itself is to be found and is an unknown, a different approach needs to be employed. The idea is to *find the hyperparameters of the Neural Network such that the "residual of the PDE, boundary conditions and initial conditions" (to be defined later) is zero.* Hence, using the derivatives of the NN, the following optimization problem is formulated to find the solution:

1. The Loss/Residual functional is computed using $\mathcal{L}^2$ norm which is a sufficient measure for this problem. The Loss functional is calculated on the collocation/sample points in the domain (N_D), on the initial condition space (N_{IC}) and on the boundaries (N_B) as such.

$$Loss_{PDE} = \frac{1}{N_D}\sum_{i=1}^{N_D}\left\|\left(\frac{\partial u_{NN}(x_i,\theta)}{\partial t}\right) + \mathcal{L}\left(u_{NN}(x_i,\theta)\right)\right\|^2 \quad on\ N_D$$

$$Loss_{Initial} = \frac{1}{N_{IC}}\sum_{i=1}^{N_{IC}}\left\|u_{NN}(x_i^0,\theta) - u_0\right\|^2 \quad on\ N_{IC}$$

$$Loss_{Boundary} = \frac{1}{N_B}\sum_{i=1}^{N_B}\|u_{NN}(\mathrm{x}_i^{\mathrm{B}},\theta) - g\|^2 \quad on\ N_B$$

$$Loss_{Total} = (\lambda_1 * L_{PDE}) + (\lambda_2 * L_{Initial}) + (\lambda_3 * L_{Boundary})$$

   Here, λ are weights applied to the loss functionals. These weights determine the importance of the respective loss. The choice of these weights depends on the problem under consideration and can either be a constant value or an adaptive variable which is optimized as the ANN training progresses.

2. The optimization problem is given by:

$$find \quad \theta = [W, b] \in \mathbb{R}$$

$$such\ that, \quad Loss_{total} < \epsilon_{tolerance}$$

The schematic diagram of PINN is shown below [2]:

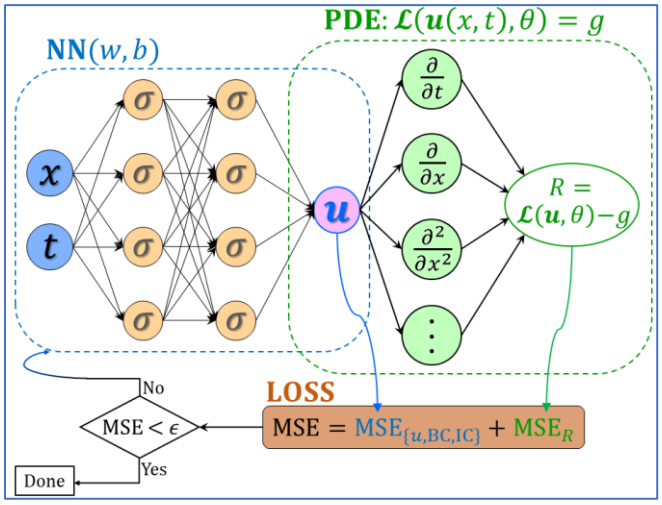

Figure 1: Physics Informed Neural Network framework

Numerical experiments

Before starting with the Navier-Stokes equations, PINNs were implemented on simpler PDEs to understand the working and estimating the behavior of the optimization process. All the simulations are implemented using Python 3.7 and "automatic differentiation" in TensorFlow is used to compute gradients for formulating the loss functional. All the simulations are run on Intel Core i7-9750H CPU 4.5 GHz 6 cores 12 threads.

1D Heat equation

The 1D Heat equation to be solved is defined on the domain as shown below,

$$Domain\ (D) = [0,1] \times [0,T]$$

$$\frac{\partial u}{\partial t} = c\frac{\partial^2 u}{\partial x^2} \qquad (x,t) \in D$$

$$c = 0.1$$
$$T = 2$$

$$u_{x=0} = u_{x=1} = 0$$

$$u(t=0) = \sin(\pi x)$$

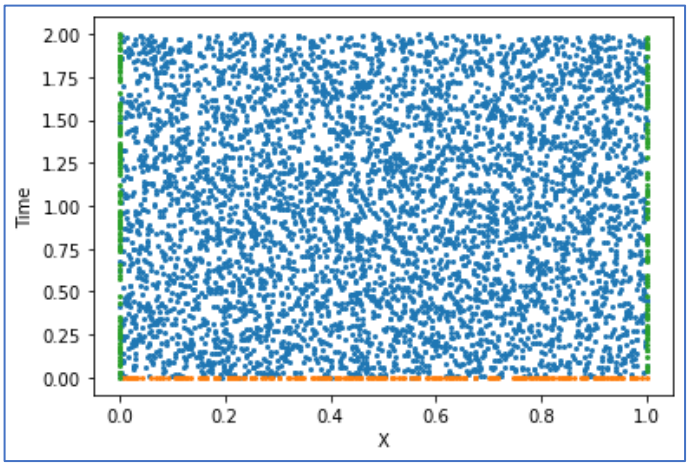

Figure 2: Collocation points in X-Time domain

Sample points are created on the domain and boundaries using uniform random distribution. The weights and biases of the neural network are generated randomly using normal distribution. The activation function σ for this problem is $tanh$. The model parameters were as follows:

N_D	N_{BC}	N_{IC}	Neural network	Optimizer	Iterations	Step size (h)
5000	200	200	[20 , 20 , 20 , 20]	Gradient Descent	50000	1e-3

The loss function $(all\ \lambda = 1)$ calculated on the respective samples points for this problem is,

$$Loss_{Tot} = \frac{1}{N_D}\sum_{i=1}^{N_D}\left\|\frac{\partial u_{NN}(x_i,t_i,\theta)}{\partial t} - 0.1\frac{\partial^2 u_{NN}(x_i,t_i,\theta)}{\partial x^2}\right\|^2 + \frac{1}{N_{IC}}\sum_{j=1}^{N_{IC}}\|u_{NN}(x_0,\theta) - \sin(\pi x_0)\|^2$$

$$+ \frac{1}{N_B}\sum_{k=1}^{N_B}\|u_{NN}(x=0,t_k,\theta)\|^2 + \frac{1}{N_B}\sum_{m=1}^{N_B}\|u_{NN}(x=1,t_m,\theta)\|^2$$

Using this loss function, the hyperparameters are updated every iteration using Gradient Descent algorithm with step size (h), as such:

$$\theta = \theta - \left(h * \frac{\partial(Loss_{Tot})}{\partial\theta}\right)$$

The neural network contained 1341 parameters, and the optimization was run for 50000 iterations which took 40 minutes. Following were the results:

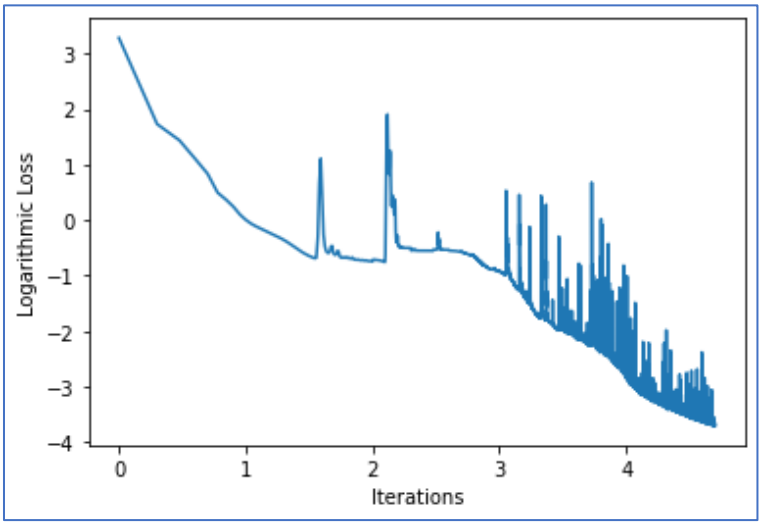

Figure 3: Log Loss vs iterations

Figure-3 shows the progression of the loss for every iteration. After 50000 iterations, the mean squared value of loss, also called as the "generalization error" was *2.1e-4*.

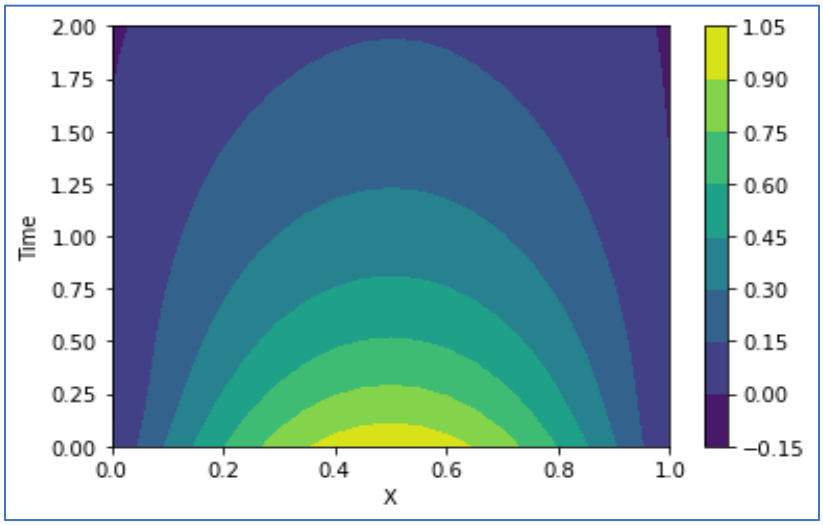

Figure 4: Solution (u) on X-Time domain

Figure-4 shows the result of the heat (u) propagation in time and space. The exact solution of the 1D Heat equation for this problem is,

$$u = \sin(\pi x) * e^{-\pi^2 ct}$$

For comparison, the following contour shows the squared error between the approximate and true solution i.e., $(u_{NN} - u)^2$:

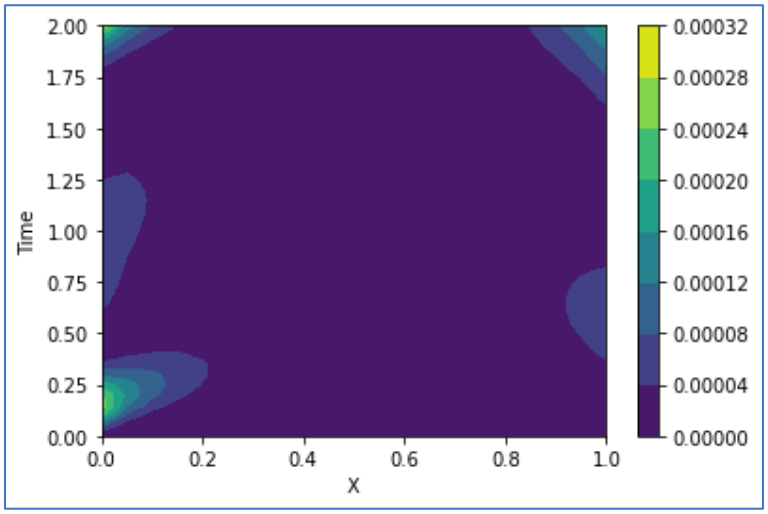

Figure 5: Square error in the solution (u)

The maximum L_2 error with the true solution is thus *3e-4* which is quite a good approximation.

Key discoveries

1. The minimization of the loss functional is oscillatory. This is due to the use of Gradient Descent (GD) algorithm which is a local optimizer. A global optimizer or an adaptive method would be much more efficient.
2. A large number of collocation/sample points are needed to get more accurate results if loss is calculated only on the sample points i.e., a Monte Carlo integration. Thus, a better approach would be to integrate the loss functional over the domain using Gauss quadrature.
3. Along with Gauss quadrature integration, adaptive sampling (similar to adaptive mesh refinement) can improve performance. It would be wise to add extra sample points at the locations where the loss is maximum, either at every iteration or after every certain number of iterations.

2D Incompressible Euler equations

For solving the Euler equations for incompressible inviscid flow, an interesting problem called "Taylor Vortex" is chosen. The Taylor vortex problem has periodic boundary conditions and multiple vortices in the domain. Since the solution is also periodic, the domain with only one vortex is considered for computation [3]. The incompressible Euler equations are defined here as,

$$D = [-6, 6]^2$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u}.\nabla)\boldsymbol{u} + \nabla p = 0 \qquad x \in D \times (0, T)$$

$$\nabla.\boldsymbol{u} = 0 \qquad x \in D \times (0, T)$$

$$\boldsymbol{u}(x, 0) = \boldsymbol{g}(x) \qquad x \in D$$

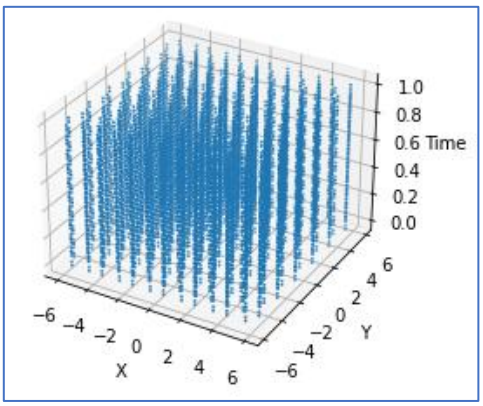

Figure 6: Collocation points in X-Y-Time domain

Figure-6 shows the sampling points in the domain for computation. The sampling points are actually Gauss quadrature integration points for integrating the loss function over the entire domain, as discussed in the key discoveries found in the experiment on the Heat equation. For integrating, the domain was divided into 6 divisions in X and Y direction each and 10 divisions in temporal direction and basically created gauss points in these subdomains. Now, for the actual Taylor vortex problem, the initial condition is:

$$X-comp: \qquad u_0(x,y) = -ye^{0.5*(1-x^2-y^2)} + a_x$$
$$Y-comp: \qquad v_0(x,y) = xe^{0.5*(1-x^2-y^2)} + a_y$$
$$with \quad a_x = 4 \quad \& \quad a_y = 0$$

The residual/loss functions for the Euler equations will be defined as,

$$Loss_{PDE} = \frac{1}{N_D}\sum_{i=1}^{N_D}\left\|\frac{\partial u_{NN}(x_i,t_i,\theta)}{\partial t} + (u_{NN}(x_i,t_i,\theta)\ .\ \nabla)u_{NN}(x_i,t_i,\theta) + \nabla p_{NN}(x_i,t_i,\theta)\right\|^2$$

$$Loss_{div} = \frac{1}{N_D}\sum_{i=1}^{N_D} \| \ \nabla\ .\ u_{NN}(x_i,t_i,\theta)\ \|^2$$

$$Loss_{IC} = \frac{1}{N_B}\sum_{i=1}^{N_B} \| \ u_{NN}(x_i,0,\theta) - g(x_i)\ \|^2$$

$$Loss_{Total} = Loss_{PDE} + Loss_{div} + Loss_{IC}$$

The most important detail of this problem, is that the boundary conditions are periodic. There are 2 ways to deal with it.

1. Imposing them weakly by adding them to the loss functional as done in the 1D Heat equation experiment and minimizing the boundary loss.
2. A more efficient way for periodic boundary conditions is to impose them strongly in the neural network architecture. To implement this, instead of passing the inputs (x, y, t) directly to the neural network, one layer is added in between as such:

$$\begin{bmatrix} x \\ y \\ t \end{bmatrix} \rightarrow \begin{bmatrix} \sin\left(\frac{2\pi x}{h}\right) \\ \cos\left(\frac{2\pi x}{h}\right) \\ \sin\left(\frac{2\pi x}{w}\right) \\ \cos\left(\frac{2\pi x}{w}\right) \\ t \end{bmatrix} \rightarrow [NN] \rightarrow \begin{bmatrix} u \\ v \\ p \end{bmatrix}$$

Here, h, w are the height and width of the domain. This inclusion of sin and cos layer makes the input to output map periodic and thus the solution periodic in the domain. This approach strongly imposes the periodic boundary conditions without the need of any additional computation and without any errors. The $softplus$ function was found to perform better than others and was used.

$$softplus\ (x) = \log_n(1 + e^x)$$

The model parameters were as follows:

N_D	N_{IC}	Neural network	Optimizer	Iterations	Step size (h)
10000	252	[20 , 20 , 20 , 20]	Nadam + Adam	10000	1e-2

Since the problem is fundamentally 3 dimensional (including time), the collocation points for computations are more. More importantly, since periodic boundary conditions are imposed strongly in the PINN architecture, collocation points on the boundaries are not needed. Points are sampled in $D \times (0, T)$ for computing PDE loss and sampled on $D \times (T_0)$ for initial condition t = 0. As the GD optimizer induced oscillations in the minimization, adaptive optimizers were chosen. Nesterov-Adam (Nadam) and Adam are built-in optimizers in TensorFlow that are actually modified versions of the Stochastic Gradient Descent algorithm. Nadam uses Nesterov-momentum during minimization to speed up the process. Although, Nadam tends to oscillate when the loss is small. Hence, Nadam was implemented for the first 2000 iterations in order to reduce the loss quickly and then the optimizer is switched to Adam which works smoothly when the loss value is close to zero. The optimization is run for 10000 iterations and the figure below shows the progression of the total mean loss for every iteration:

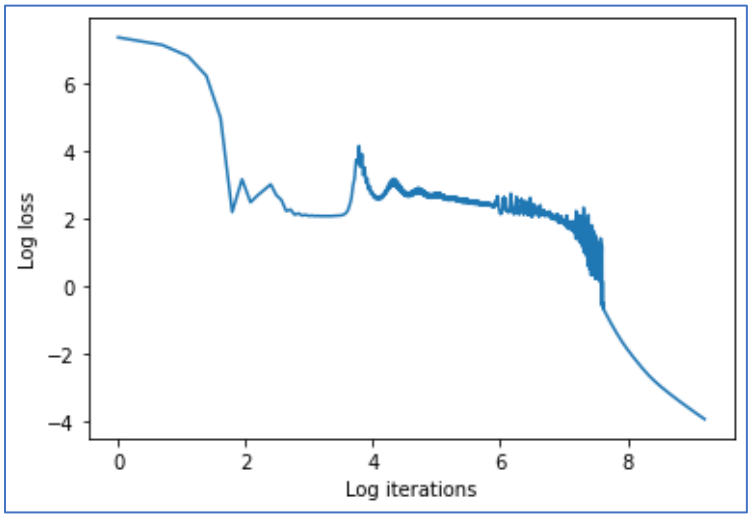

Figure 7: Log loss vs iterations

The minimization of the loss is much smoother compared to the Heat equation experiment due to the use of better optimizers i.e., Adam and Nadam. For the Taylor vortex, the vorticity vector is the main focus. After obtaining the optimized hyperparameters, the solution from the neural network (u, v, p) at (x, y, t) is obtained and then differentiated to obtain the vorticity as such:

$$\omega_{NN} = \frac{\partial v_{NN}}{\partial x} - \frac{\partial u_{NN}}{\partial y}$$

The following figures shows the vorticity contour at different times:

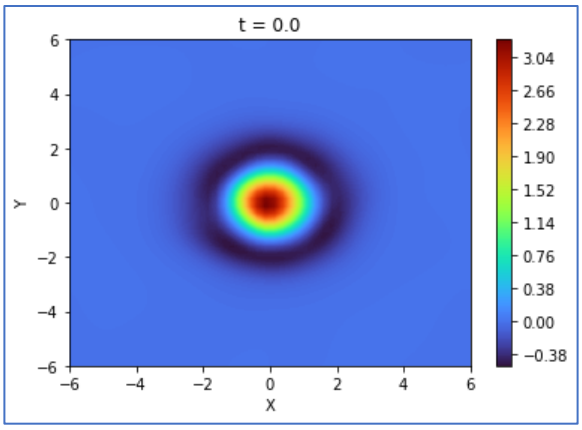

Figure 8: Vorticity on XY domain at t=0

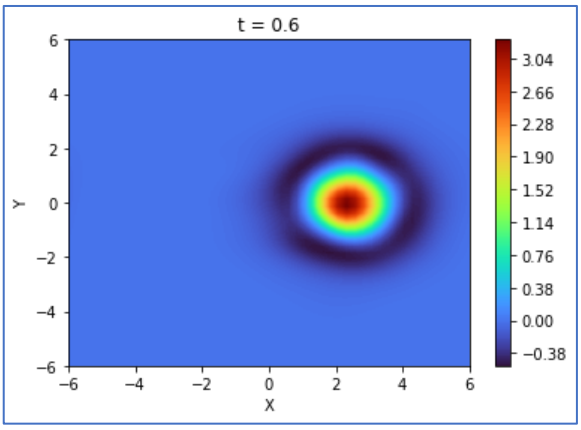

Figure 9: Vorticity on XY domain at t=0.6

At the initial condition, the Taylor vortex is at the center of the domain. As time progresses, the vortex shifts in X-direction. The simulation took 25 minutes and the mean generalization error (total loss) was 1.9e-2.

Since the boundary conditions are periodic, we can see in the result below that when the vortex passes through the boundary, another vortex appears from the opposite boundary.

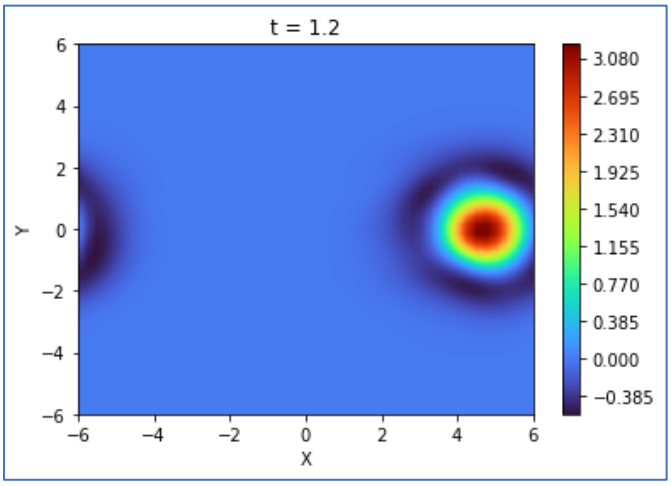

Figure 10: Vorticity on XY domain at t=1.2

The exact solution for the velocity is given by [3],

$$u(x,y,t) = -(y - a_y t)e^{0.5*\left[1-(x-a_x t)^2-(y-a_y t)^2\right]} + a_x$$
$$v(x,y,t) = (x - a_x t)e^{0.5*\left[1-(x-a_x t)^2-(y-a_y t)^2\right]} + a_y$$

However, the exact solution given in the referenced paper is not periodic. The analytical solution is correct as long as the values on the boundary are zero i.e., the vortex is completely inside the domain. The following contour shows the squared error between the PINN and analytical solution.

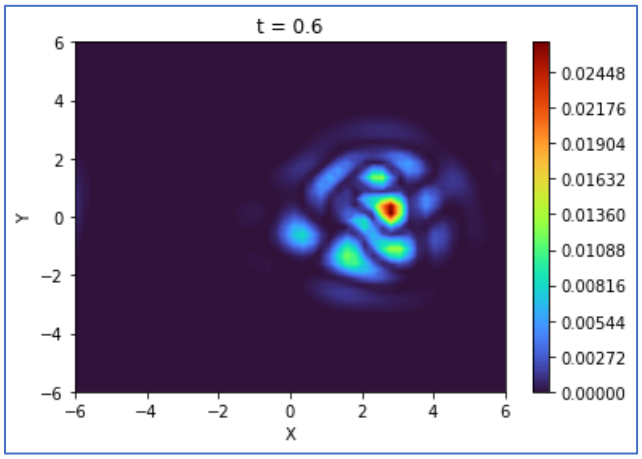

Figure 11: Squared error in the vorticity solution at t=0.6

The error in the X-component of velocity was 0.13 % and in Y-component of velocity was 4.7 %.

2D Navier-Stokes equations

After performing the numerical experiments, the working of PINN was understood along with the optimization procedure for the hyperparameters. Using the knowledge from previous simulations, a PINN was programmed for the 2D Navier Stokes equations.

The problem chosen for this simulation was the *2D Unsteady flow over a flat plate*, which is a benchmark problem in CFD. The plate is located on Γ_1 at $y = 0.0$ and the fluid inlet is located on Γ_2 *and* Γ_3. The problem is mathematically modelled as:

Domain:

$$D \in [0,1] \times [0,0.5] \times [0,T]$$

Boundary conditions:

$$[\,u \;\; v\,] = [\,1 \;\; 0\,] \quad in\ \Gamma_{2,3}$$
$$u = 0 \quad in\ \Gamma_1$$

Initial conditions:

$$[\,u_0 \;\; v_0\,] = [\,0 \;\; 0\,] \quad in\ D$$

Reynolds number = 100

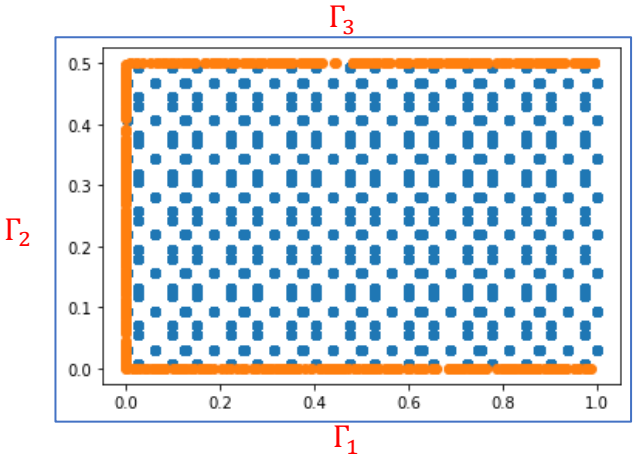

Figure 12: Collocation points on XY domain

N_D	N_{BC}	N_{IC}	Neural network	Optimizer	Iterations	Step size (h)
14336	1500	500	[20 , 20 , 20 , 20]	Adam	50000	Decay ($1e^{-2}$ to $1e^{-5}$)

Decaying step size ensures that the step size is less than the loss and optimization does not blow up. The hyperparameters were initialized using random distribution with mean 0.0 and variance 0.1.

Using this knowledge, a new activation function was proposed [4] called as the "Gaussian Error Linear Unit" (GELU). This function resembles the famous ReLU (Rectified Linear Unit) and the ELU (Exponential Linear Unit). ReLU is the most common activation function in data science. However, since ReLU is a bilinear function, its 2nd derivative is zero and is not useful here as 2nd order derivatives are needed to compute the PDE loss functional. After experimenting, it was found that the GELU performed much better than ELU. (Formulae in Appendix)

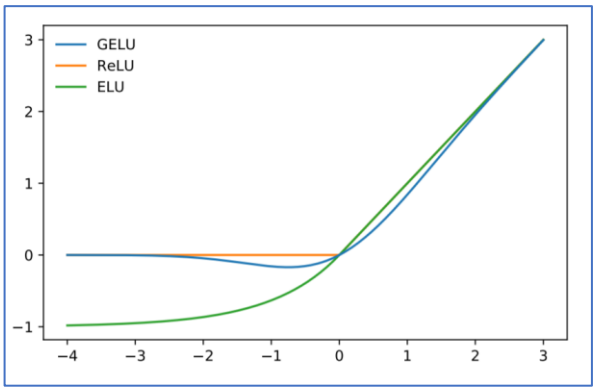

Figure 13: Activation functions comparison

Thus, the GELU activation function was chosen for this simulation. The total time considered for the simulation was T = 1. After running the optimization for 50000 iterations which took 50 minutes, the mean total loss value on the collocation points was *6e-3*. Following plot shows the loss at every iteration:

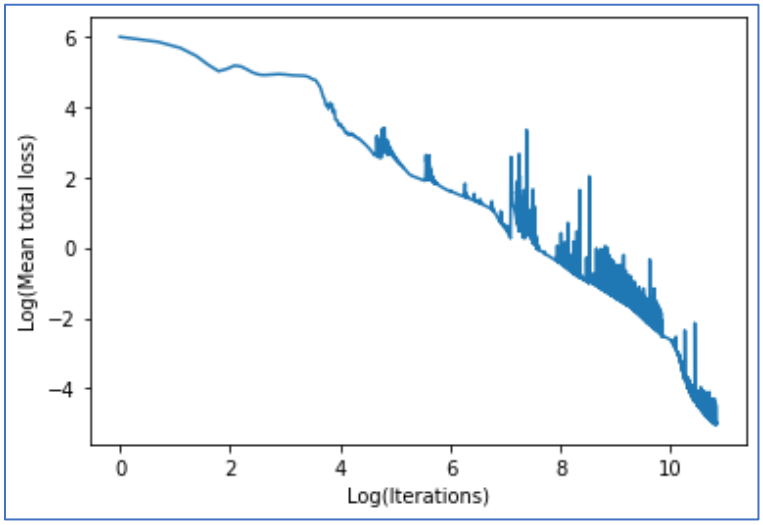

Figure 14: Log Loss vs Iterations

The target loss was 1e-7 and running the optimization for a large set of collocation points to obtain the desired loss would require hours of computation on a portable laptop. A GPU and parallel processing are thus advised for solving PDEs using PINNs. The following plot shows the velocity contour at t = 0.1.

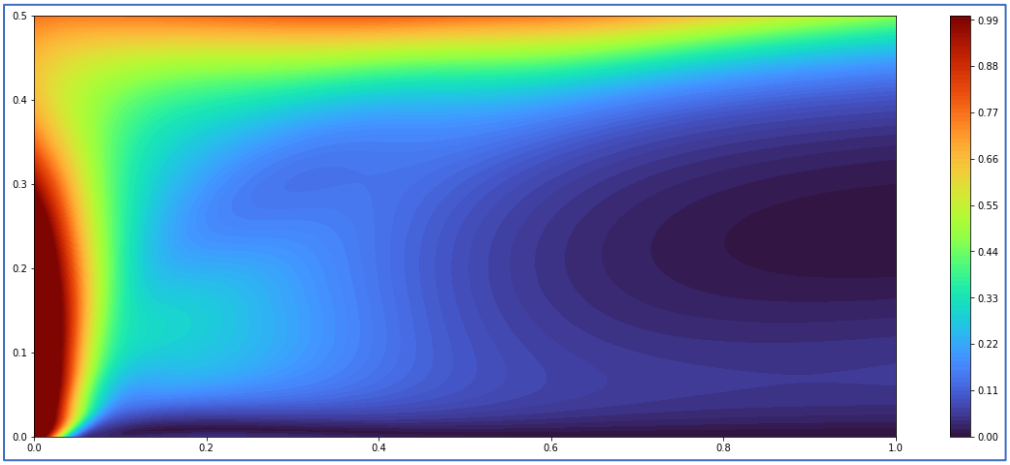

Figure 15: Velocity on XY domain at t=0

The boundary conditions and initial conditions are not exactly satisfied since they are weakly imposed in the loss function and the loss value is not approximately equal to 0 after optimization. For this transient problem, the steady state is reached at the end time (t = 1).

The figures below show the solution for pressure and velocity magnitude at the end time:

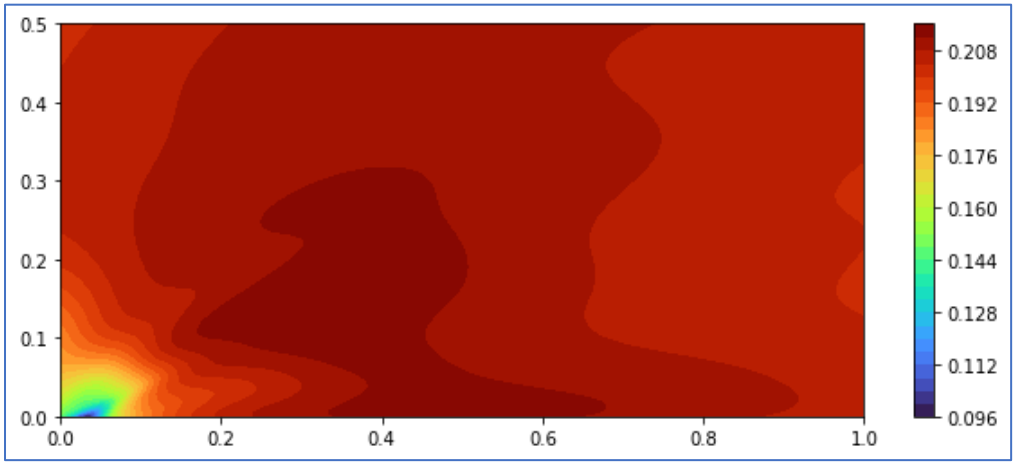

Figure 16: Pressure on XY domain at t=1

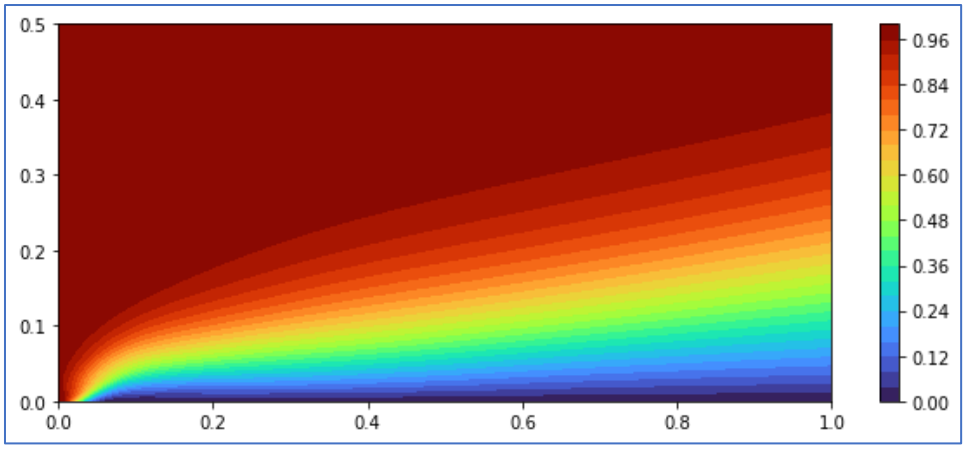

Figure 17: Velocity magnitude on XY domain at t=1

The formation of the boundary layer is clearly visible as expected. In order to validate the results, the "Boundary Layer App" from MATLAB created by Ye Cheng [5] was used. The solution from this MATLAB application is considered as a reference.

Following were the results obtained using the MATLAB Boundary Layer app:

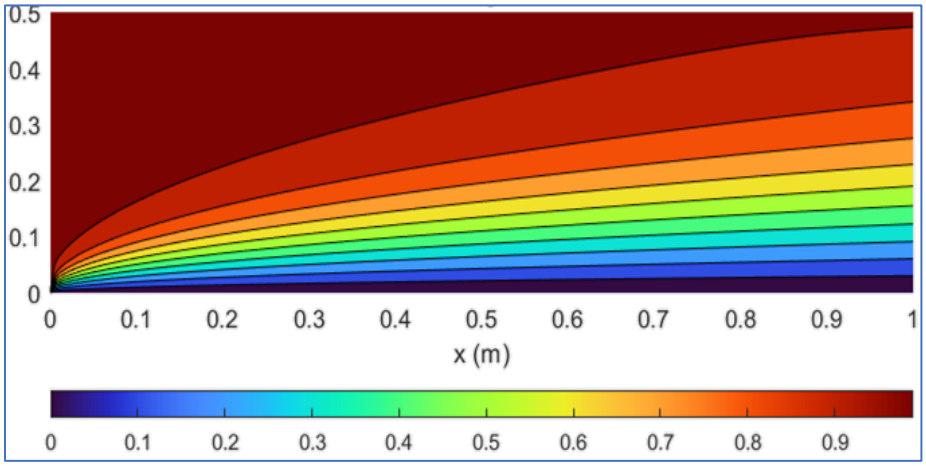

Figure 18: Velocity magnitude using Boundary Layer app

It's evident here that the PINN's solution matches precisely with the "true" solution. In order to obtain extremely accurate solution, more iterations and more sample points are needed. This computation needs to be done on a desktop or a GPU server which would be a future project. Various experiments were conducted for problems such as *2D Lid-driven cavity*, *2D jet flow*. However, an accurate solution was not obtained due to computational reasons.

Conclusions

1. Physics Informed Neural Networks (PINNs) are a brand-new way of solving PDEs and the ongoing research is only at its nascent stage.
2. The main advantage of PINNs is its property as a universal function approximator. This theoretically allows the ANN to approximate any continuous function which is a solution of the PDE. Also, PINNs do not need meshing/discretization of the domain.
3. The major disadvantage of PINNs or ANNs in general is the training/optimization time required to get the solution.
4. A physical advantage of PINNs is that it is not only capable of approximating one solution, but many solutions as well. Taking the example of Heat equation, there exists a constant "c" in the PDE. When training the PINN, along with space and time another dimension can be added which will be "c" and points will be sampled on a specific range for the values of "c". Thus, the PINN will be trained in the x-c-t domain. After minimizing the loss, the PINN now acts as a black box to give the solution "u" for every point in space, time and for every value of "c" in range. That is, the solution is a function of (x, c, t).
5. This approach can be extended to more interesting aspects. The solution can also be a function of its initial condition. Considering that the initial condition can be approximated by "P" order of polynomial and thus "P+1" coefficients, it is possible to add P+1 input dimensions to the PINN and perform the optimization for spatial + temporal + P + 1 dimensional inputs and one output being the solution. Once the training is completed, the NN will be a map from domain and initial condition to solution. That is, it will be possible to find solution to the same PDE for different initial conditions which can be approximated by P+1 polynomial coefficients. These experiments were performed and the results were close but not accurate due to computational limitations. This approach can also be applied for source/force function in the PDE. Then, it will be possible to get solution of the PDE for any source/force function which can be approximated using P+1 polynomial coefficients in the range of values used for training.
6. Moreover, ANNs are also being used for approximating the solution using experimental data along with the typical PINNs framework by adding an additional loss function which is the error between u_{NN} and $u_{experimental}$. This is not possible in traditional numerical methods.
7. Different variations of PINNs have been created like Fourier Neural Operator [6] where an entire family of PDEs is learnt i.e. operator approximation. Here, one single layer of alternate sin and cos is used which basically makes it a Fourier series and is a known basis for solution to PDEs.
8. PINNs cannot replace numerical methods like FEM/FDM but combining numerical methods with Neural Networks is opening new doors in computational mechanics.

Appendix

$$
\begin{aligned}
softplus\ (x) &= \log_n(1+e^x) \\
ReLU\ (x) &= \max(0,x) \\
ELU\ (x) &= \begin{cases} x & x > 0 \\ \alpha * (e^x - 1) & x \leq 0 \end{cases} \\
tanh\ (x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
sigmoid(x) &= \frac{1}{1+e^{-x}}
\end{aligned}
$$

References

[1] M. Raissi, P. Perdikaris, and G. E. Karniadakis, 'Physics Informed Deep Learning (Part I): Data-driven Discovery of Nonlinear Partial Differential Equations', no. Part II, pp. 1–19, 2017, [Online]. Available: http://arxiv.org/abs/1711.10566.

[2] X. Meng, Z. Li, D. Zhang, and G. E. Karniadakis, 'PPINN: Parareal physics-informed neural network for time-dependent PDEs', *Comput. Methods Appl. Mech. Eng.*, vol. 370, pp. 1–17, 2020, doi: 10.1016/j.cma.2020.113250.

[3] S. Mishra and R. Molinaro, 'Estimates on the generalization error of Physics Informed Neural Networks (PINNs) for approximating a class of inverse problems for PDEs', 2020, [Online]. Available: http://arxiv.org/abs/2007.01138.

[4] D. Hendrycks and K. Gimpel, 'Gaussian Error Linear Units (GELUs)', pp. 1–9, 2016, [Online]. Available: http://arxiv.org/abs/1606.08415.

[5] Y. Cheng, 'Ye Cheng (2021). Boundary Layer App (https://www.mathworks.com/matlabcentral/fileexchange/40680-boundary-layer-app), MATLAB Central File Exchange. Retrieved July 10, 2021'. .

[6] Z. Li *et al.*, 'Fourier Neural Operator for Parametric Partial Differential Equations', no. 2016, pp. 1–16, 2020, [Online]. Available: http://arxiv.org/abs/2010.08895.