Scientific Machine Learning: solving PDEs with FEM and PINNs.

Chiara Giraudo[a], Claudia Fabris[b], Agnieszka Seremak[b], Ana Anzulović[c]

[a]Department of Mathematics, University of Oslo, Norway
[b]Department of Chemistry, University of Oslo, Norway
[c]Department of Geosciences, University of Oslo, Norway

Abstract

We aim to approximate the steady-state solution of the 2D Darcy equation, which is a second-order linear elliptic partial differential equation (PDE), using two different methods: the Finite Element Method (FEM) and Physics-Informed Neural Networks (PINNs). FEM is a classical numerical method commonly used for solving PDEs, whereas PINNs leverage machine learning techniques to approximate PDE solutions. Our objective is to thoroughly analyse these two methods, examining not only their apparent similarities and dissimilarities but also their underlying ones. To compare the results obtained from the two numerical methods, we consider constant and piecewise constant diffusion coefficient values. We will discuss the strengths and weaknesses associated with each approach when applied to the 2D Darcy equation. Furthermore, we provide a link to a repository that contains all the codes required to solve the 2D Darcy equation using the aforementioned methods.

1 Introduction

In recent years, *Machine Learning* (ML) has experienced significant advancements, showcasing remarkable performance across various applications. At the same time, there has been growing interest in integrating this modern discipline with the longstanding field of Scientific computing. This convergence has given rise to a new interdisciplinary field known as Scientific Machine Learning (SciML) [1]. SciML is an emerging multidiscipline which leverages expertise from applied and computational mathematics, computer science, and physical sciences.

The objective of this report is to investigate the integration of Machine Learning and Numerical methods for solving Partial Differential Equations (PDEs). In particular, we will implement Physics-Informed Neural Networks (PINNs) [2] and compare their performance with a classical approach, such as the Finite Element Method (FEM). Although PDEs and neural networks may seem distinct, the significant technological advancements in recent years has facilitated the convergence of these fields.

Neural networks have demonstrated immense value in extracting information from large datasets. However, they possess a drawback in their inability to capture the inherent knowledge of the governing physical laws that describe the systems they aim to model. Specifically, the trained neural network model often remains a "black-box", making it challenging to verify whether their performance is a result of identifying the correct causal relationships or merely unstable correlations. Conversely, models based on physical laws (described by PDEs) have a long history of application and development, leading to a strong theoretical foundation and the advancement of numerical methods necessary for approximating their solutions (see [3]). Nevertheless, constructing a PDE model and finding its solution using appropriate numerical methods typically require explicit assumptions or computationally demanding computations. Moreover, these PDE models face limitations when learning from observational inputs, as a new model must be devised to address this challenge.

In this report, we are interested in studying the steady-state solution of the 2D Darcy equation, which is a second-order linear elliptic partial differential equation given by:

$$\begin{cases} -\nabla \cdot (\alpha(x,y)\nabla u(x,y)) = s, & \text{in } \Omega \\ u(x,y) = 0, & \text{on } \partial\Omega. \end{cases} \tag{1}$$

Here, $\alpha \in L^\infty\left((0,1)^2;\mathbb{R}_+\right)$ represents the diffusion coefficient, $s \in L^2\left((0,1)^2;\mathbb{R}\right)$ is the source or forcing function, and the domain is the unit square $\Omega = [0,1]^2$. Our objective is to find an approximate solution to problem (1) for different values of the diffusion coefficient, using two methods: Finite Element Method (FEM) and Physics-Informed Neural Networks (PINNs).

For this study, we fix the forcing function to $s = 1$ and consider a specific distribution class for the diffusion coefficient. We divide the domain into four regular subdomains and assume the diffusion coefficient to be piecewise constant, with a constant value within each subdomain.

In the Finite Element Method and Physics-Informed Neural Networks, we obtain an approximation of the solution by solving a single instance of the PDE for different values of α. Specifically, we vary the diffusion term within each subdomain to obtain the approximate solutions. Figure 2 shows how the domain is divided in subdomains. We will begin by providing a detailed description of these two methods, followed by the presentation of the obtained results. Finally, we will proceed with a comparative analysis of the results.

2 Methodologies

2.1 Finite Element Method

The majority of the laws that describe the dynamics of a system may be represented by partial differential equations. Due to their complexity, these equations can not be solved analytically and the use of numerical methods is required. Moreover, when trying to accurately reconstruct a complex function u with a measurement loss as the only constraint, large quantities of data are required, which are usually not available when dealing with physical systems, as data may be sparse or noisy.

The finite element method (FEM) is a commonly used method for numerically solving PDEs. A system of consideration is divided into smaller parts called the finite elements. This is realized by a process of discretization in the space dimensions, achieved with generating a mesh of studied object. Therefore, FEM requires a mesh to discretize the domain. This mesh is usually constructed with polyhedrons (rectangular and triangular finite elements in 2D). Depending on the complexity of studied problem the number of elements in the mesh can vary, which directly impact the method accuracy and the computational costs of such approximations. Creating a proper mesh is of particular importance when applying FEM and substantial attention should be given to this step in order to assure accurate results and small errors.

The heart of this method is turning the PDE into a variational problem. This can be done by multiplying the PDE with a *test function*, denoted v, then integrating the results over the domain (Ω) and finally performing integration by parts of terms with second-order derivatives. The canonical notation for problem (1) can be expressed as: find u $\in V := H_0^1(\Omega)$ such that

$$a(u, v) = S(v) \quad \forall v \in V, \tag{2}$$

where the bilinear and linear forms are given by

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x}, \tag{3}$$

$$S(v) = \int_\Omega s v \, d\mathbf{x}. \tag{4}$$

It is important to note that the boundary integral is absent in the formulation since it is equal to zero, due to the homogeneous Dirichlet boundary conditions. The Hilbert space $H_0^1(\Omega)$ is, in fact, defined as:

$$H_0^1(\Omega) := \left\{ v \in L^2(\Omega) \, : \, Dv \in L^2(\Omega), \, \gamma v = 0 \right\}. \tag{5}$$

Here, D represents the distributional derivative, and γ denotes the trace operator on the boundary of the domain, $\partial\Omega$.

Let us consider a mesh of triangles covering the domain denoted as $\mathscr{T}_h$. The approximate solution $u_h \in V_h$ is obtained by solving a system of equations of the form:

$$A u_h = s_h. \tag{6}$$

Here, A represents a matrix, while u_h and s_h are vectors. The space V_h is a finite-dimensional subspace of V. Specifically, the matrices A and the vector s_h are derived from the discretization of the bilinear form (3) and the linear form (4), respectively. The discretization process relies on both the mesh structure and the chosen approximation space V_h. In the Galerkin method, the finite-dimensional space V_h is the space of continuous piecewise polynomials of degree less than or equal to d (usually 1 or 2) on the mesh, *i.e.*, it belongs to the space:

$$V_h := X_h^d \cap H_0^1(\Omega), \tag{7}$$

where

$$X_h^d := \left\{ v_h \in C^0(\overline{\Omega}) \, : \, v_h|_K \in \mathbb{P}_d(K) \quad \forall K \in \mathscr{T}_h \right\}, d \geq 1. \tag{8}$$

Here d is the degree of the polynomial and $\mathbb{P}_d(K)$ is the space of polynomials of degree less than or equal to d in the domain K.

This method only requires the boundary conditions of the problem, and not any additional data. We will implement it using FEniCSx [5] in Python. The way a boundary-value Darcy equation can be solved is described with bullet points below:

- Identify the domain, the PDE and its boundary conditions along with the source function.
- Express PDE in variational form.
- Write a Python code using FEniCS abstraction which defines the domain, the variational problem, boundary conditions and source function.
- Finally, call FEniCS to solve the boundary-value problem.

This traditional solver has the advantage of well developed theory, which in practice means that it can provide the upper bounds for the error. We mentioned before that FEM is mesh dependent and it only requires the boundary conditions of the problem. In our study we impose homogeneous Dirichlet boundary condition.

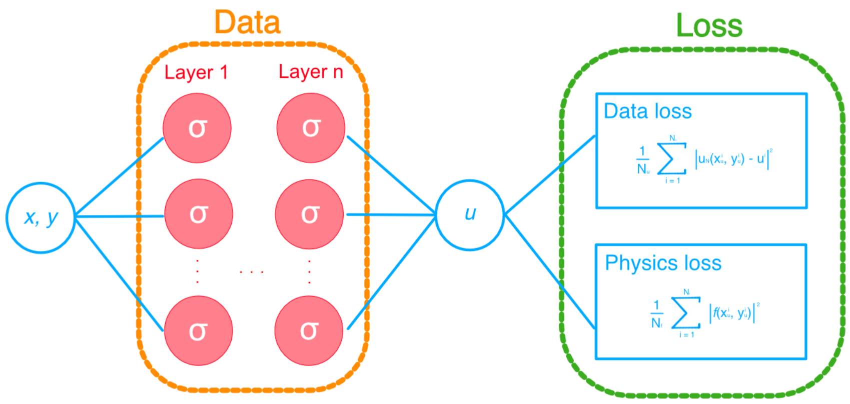

Data
Layer 1
Layer n
σ
x, y
u
Loss
Data loss
$\frac{1}{N_u}\sum_{i=1}^{N_u}|u_N(x_u^i, y_u^i) - u^i|^2$
Physics loss
$\frac{1}{N_f}\sum_{i=1}^{N_f}|f(x_u^i, y_u^i)|^2$

Figure 1: *Graphical representation of a PINN where σ is the activation function, u is the exact solution, f is the neural network, N_u is the number of training points and N_f is the number of collocation points. Replicated from [4].*

2.2 Physics Informed Neural Networks

Physics Informed Neural Networks (PINNs) [2] are universal function approximators trained to solve supervised learning assignments incorporating the knowledge of physical laws governing a given dataset in the learning process. Since both the training data and the governing equations must be satisfied, such kind of neural network does not require a large and complete set of starting data [6]. Additionally, unlike the FEM, the PINNs method does not require space discretization (*i.e.* a mesh). Instead of dividing the domain into finite elements, PINNs use a set of collocation points scattered throughout the domain and a set of training points to enforce the differential equation and the boundary conditions, respectively.

PINNs method is designed to approximate the solution of a partial differential equation using only boundary condition data, *i.e.* the values of the exact solution on the domain boundary. The fundamental concept behind PINNs is to utilize a Neural Network (NN) to approximate the exact solution u, resulting in a predicted solution u_N represented by a deep neural network. The neural network is then trained to minimize the error between the predicted solution and the exact solution at the training points. Additionally, a regularization term is included to ensure that the predicted solution satisfies the PDE. The graphical representation of a NN in the PINN method is shown in Figure 1. Now, let's delve into the details of this method by implementing it for the 2D steady Darcy equation (1). The differential equation in (1), can be rewritten as:

$$\frac{\partial}{\partial x}\left(\alpha\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\alpha\frac{\partial u}{\partial y}\right) = 1, \tag{9}$$

or

$$(\alpha u_x)_x + (\alpha u_y)_y = 1, \tag{10}$$

where α is the diffusion coefficient. Consequently, the Physics-Informed Neural Network $f(x, y)$, employed to solve our problem, can be defined as follows:

$$f(x, y) := (\alpha(u_N)_x)_x + (\alpha(u_N)_y)_y - 1. \tag{11}$$

It is important to note that since the predicted solution u_N is a neural network (NN), the function f is also a neural network. Furthermore, u_N and f share the same set of parameters, although f may employ different activation functions. The derivation of the neural network $f(x, y)$ can be achieved through automatic differentiation.

The shared parameters are learned by minimizing the mean squared error (MSE) loss function, given by:

$$MSE := MSE_u + MSE_f. \tag{12}$$

This loss function is composed of two parts: a data loss and a physics loss. The data loss represents the supervised learning aspect of the method, as it relies on the available data, specifically the values of the exact solution on the domain boundary. Mathematically, it is defined as:

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |\, u_N(x_u^i, y_u^i) - u^i\,|^2, \tag{13}$$

with $\{(x_u^i, y_u^i), u^i\}_i^{N_u}$ being the boundary training data. On the other hand, the physics loss ensures that the neural network u_N satisfies the PDE at a finite set of collocation points located inside the domain. This loss term relies on the Physics-Informed Neural Network f defined in (11), and it can be expressed as:

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f} |\, f(x_f^i, y_f^i)\,|^2, \tag{14}$$

where $\{(x_f^i, y_f^i)\}_i^{N_f}$ are the collocation points. A physics loss function is used to enforces adherence to the physical law. This is achieved by expressing the PDE as a residual at N_f collocation points, with the number of collocation points usually being significantly larger than the number of training points, i.e. $N_f \gg N_u$.

The MSE is then minimized at both the training and collocation points, and the gradients are backpropagated. To implement PINNs, we use the open-source TensorFlow library [7].

3 Results and Discussion

To narrow down the implementation to a specific class of 2D steady Darcy equations, we set the source function to $s = 1$ and select the diffusion coefficients α from a distribution of piecewise constant functions. In particular, we divide the domain into four equal subdomains, as depicted in Figure 2, and assume that α remains constant within each subdomain. Moreover, we enforce the constraint $0.1 < \alpha < 1$ to further refine the parameter space and enable the visualization of the solutions through graphical representations. In the following subsections, we present the results obtained using the FEM and PINNs methods, followed by a comparison between the two approaches. It should be noted that these methods could be studied by comparing the Mean Squared Errors computed from the exact solutions. However, in our case, as is often the case in real-world applications, the exact solution is unknown. Nevertheless, we have knowledge from [3] that the FEM method converges for the Poisson problem, which corresponds to the Darcy problem with a constant diffusion coefficient. Additionally, the code for the Poisson problem, implemented in [5], has also been shown to converge. As for the PINNs method, we have evidence from [2] that the Burger code converges, and the Darcy problem is considered relatively easier. For this reason we decided to analyse this methods by visualizing the approximate solutions as would happen in real applications.

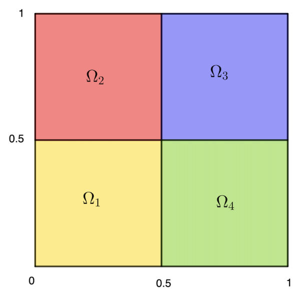


Figure 2: *Graphical representation of the subdomains of the domain Ω.*

3.1 FEM

The finite element method is implemented using the open-source computing platform FEniCSx, editing the 2D Poisson tutorial code in [5]. We use a Conda environment with the following versions for the main packages:

- *Python*: version 3.7.12
- *FEniCS*: version 2019.1.0
- *Numpy*: version 1.21.6

Since the domain Ω corresponds to the unit square, the mesh can be easily constructed using the following FEniCS command:

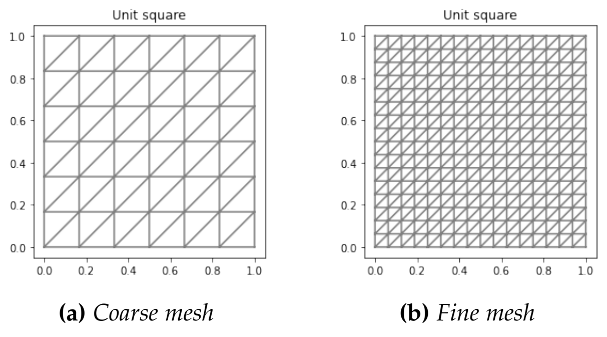
```
from fenics import *
mesh = UnitSquareMesh(Nx, Ny)
```

Here, Nx and Ny represent the number of subintervals in the x and y axes, respectively. As the mesh is a triangulation of the domain, the total number of elements in the mesh is equal to $(2 \times Nx \times Ny)$, and the mesh sizes are

$$h_{max} = \sqrt{(Nx^{-2} + Ny^{-2})} \tag{15}$$

$$h_{min} = min\left(Nx^{-1}, Ny^{-1}\right). \tag{16}$$

The mesh must satisfy the following property:
If an element of the mesh intersects the boundary of a subdomain, this intersection must coincide with an edge of the element.
This assumption holds true when both Nx and Ny are even numbers. Figure 3 illustrates two examples of mesh. The first example, shown in Figure 3a, is a coarse mesh with $Nx = Ny = 6$, while the second example, depicted in Figure 3b, represents a finer mesh with $Nx = Ny = 16$.



(a) *Coarse mesh* **(b)** *Fine mesh*

Figure 3: *Examples of meshes of the unit square.* ***(a)*** *coarse mesh: $Nx = 6$, $Ny = 6$, number of elements = 72.* ***(b)*** *fine mesh: $Nx = 16$, $Ny = 16$, number of elements = 512.*

Once the mesh has been created, the finite-dimensional space, where we approximate the exact solution, is defined using Continuous Galerkin (CG) spaces. These spaces consist of continuous and piecewise polynomial functions (see Equation 8). In FEniCS, we define this space as follows:

```
from fenics import *
V = FunctionSpace(mesh, 'CG', degree)
```

Here, the degree parameter represents the maximum degree of the polynomials in the space.

The linear system of equations resulting from the discretization of the variational formulation is solved using a sparse LU decomposition (Gaussian elimination). This method is the default choice in FEniCS programs, and we opted not to change it due to its high robustness and effectiveness in handling 2D problems with a few thousand unknowns, which aligns with the scale of our specific problem.

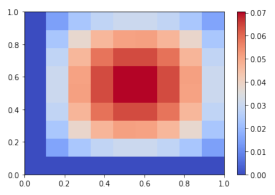
As mentioned earlier, the Finite Element Method has a well-developed theory, and under regularity conditions, we can establish an *a priori* error estimate. Let u_e denote the exact solution and u_h the approximated solution. The parameter h represents the mesh size, defined in Equation (15). In the setting of our problem, when the diffusion coefficient remains constant throughout the domain, the following error estimate holds true:

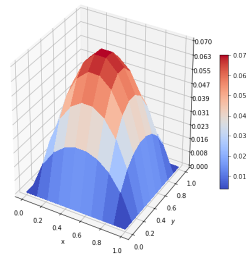
$$\|u - u_h\|_{H^1(\Omega)} \leq Ch\|u\|_{H^2(\Omega)}. \tag{17}$$

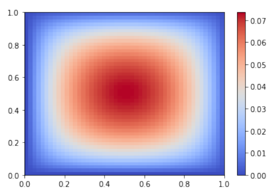
This estimate holds for any polynomial degree in the Galerkin space. Consequently, choosing a polynomial degree n greater than one, i.e., degree $= n$ for any $n \in \mathbb{N}$ with $n > 1$, does not yield a higher convergence rate than selecting degree $= 1$. For this reason in our implementation we fixed degree $= 1$. It's worth noting that a finer mesh corresponds to a smaller mesh size h, leading to a smaller error norm. When the diffusion coefficient is piecewise constant, it is still possible to establish convergence of the Galerkin method. However, the convergence rate is no longer known *a priori*. Consequently, it becomes impossible to prove *a priori* estimates such as (17). For the proofs of the convergence estimate and a more comprehensive description of the finite element method, we refer readers to [3].

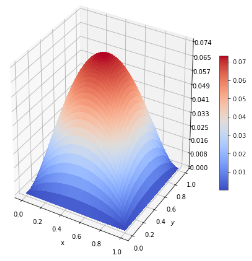
Figure 4 illustrates the approximate solutions for three different meshes. The first column presents a 2D representation, while the second column displays a 3D representation. Throughout the entire domain, the diffusion coefficient is fixed at $\alpha = 1$. The mesh sizes vary as follows: in figures 4a and 4b, $h = 0.14$; in figures 4c and 4d, $h = 0.028$; and in figures 4e and 4f, $h = 0.014$. Since we do not have access to the exact solution, we are unable to compute the approximation error for different meshes. Consequently, the graphical representation is the only means to observe the higher accuracy of the approximate solution with a finer mesh.

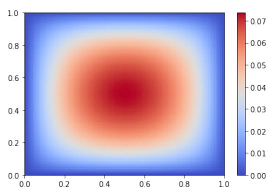
We report the approximate solutions for the Darcy problem (1) for two different values of the piecewise constant diffusion coefficient. The results displayed


(a) *FEM. 2D plot: 10^2 mesh element*


(b) *FEM. 3D plot: 10^2 mesh element*


(c) *FEM. 2D plot: 50^2 mesh element*


(d) *FEM. 3D plot: 50^2 mesh element*


(e) *FEM. 2D plot: 100^2 mesh element*

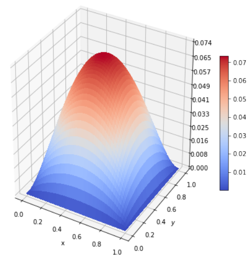
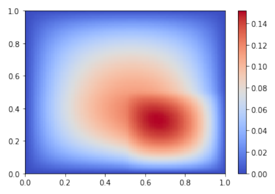

(f) *FEM. 3D plot: 100^2 mesh element*

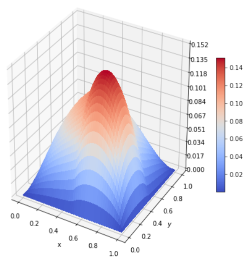
Figure 4: *FEM. Approximate solutions for different mesh size: 2D and 3D representation. Fixed hyperparameters: $\alpha = 1.0$ and degree = 1.* ***(a), (b)*** *number of mesh element = 10^2, $h = 0.14$.* ***(c), (d)*** *number of mesh element = 50^2, $h = 0.028$.* ***(e), (f)*** *number of mesh element = 100^2, $h = 0.014$.*

here were obtained by fixing the mesh size to $h = 0.014$ and the degree to 1. The first two figures, 5a and 5b, correspond to $\alpha = [0.64, 0.93, 0.88, 0.20]$, while the last two figures, 5c and 5d, correspond to $\alpha = [0.25, 0.74, 0.26, 0.18]$. Referring to Figure 2, we assign $\alpha[0]$ in Ω_1, $\alpha[1]$ in Ω_2, $\alpha[2]$ in Ω_3, and $\alpha[3]$ in Ω_4. The implementation time for the first problem is 0.078, while for the second problem, it is 0.077, indicating that the method is efficient. It is worth noting that for smaller values of α, the approximate solution tends to have higher magnitudes.

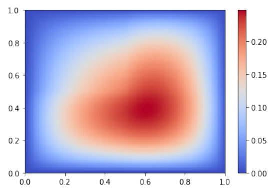
Finally, we aim to demonstrate that the Finite Element Method (FEM) is capable of finding an approximate solution in a relatively short time, even when the diffusion coefficient varies significantly. Figure 6 illustrates the results obtained by fixing the mesh size to $h = 0.014$, the degree to 1, and setting the



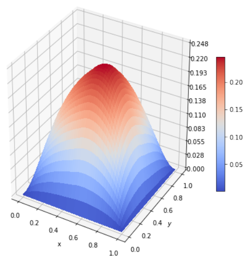
(a) *FEM. 2D plot:* α = [0.64, 0.93, 0.88, 0.20]



(b) *FEM. 3D plot:* α = [0.64, 0.93, 0.88, 0.20]



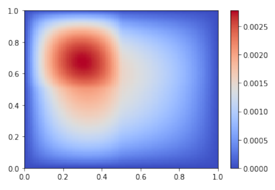
(c) *FEM. 2D plot:* α = [0.25, 0.74, 0.26, 0.18]



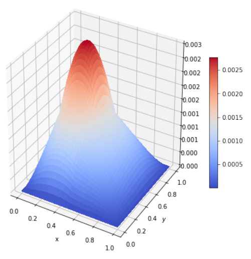
(d) *FEM. 3D plot:* α = [0.25, 0.74, 0.26, 0.18]

Figure 5: *FEM. Approximate solutions for different piecewise constant diffusion coefficient: 2D and 3D representation. Fixed hyperparameters:* $h = 0.014$ *and degree = 1.* ***(a), (b)*** $\alpha = [0.64, 0.93, 0.88, 0.20]$. ***(c), (d)*** $\alpha = [0.25, 0.74, 0.26, 0.18]$.

diffusion coefficient as $\alpha = [30, 10, 60, 100]$. The implementation time for this case is 0.095, which is slightly longer compared to the previous solutions shown, but still considered fast.



(a) *FEM. 2D plot:* $\alpha = [30, 10, 60, 100]$



(b) *FEM. 3D plot:* $\alpha = [30, 10, 60, 100]$

Figure 6: *FEM. Approximate solution for piecewise constant diffusion coefficient with vast differences in the constant parts: 2D and 3D representation. Fixed hyperparameters:* $h = 0.014$ *and degree = 1,* $\alpha = [30, 10, 60, 100]$

3.2 PINNs

The implementation of the physics informed neural networks utilizes the open source library TensorFlow. Starting from the PINNs code of the Burger equation in this GitHub repository, we implement a PINNs code for the 2D steady Darcy equation. The main packages utilized within a conda environment include:

- *Python*: version 3.5.5
- *TensorFlow*: version 1.3.0
- *Numpy*: version 1.11.3
- *pyDOE*: version 0.3.8

These specific versions were employed during the implementation process.

As previously mentioned, PINNs methods do not require a mesh. However, the loss function is evaluated at a finite set of points, which includes both the training and collocation points. The training points are located on the boundary of the domain and are utilized in the calculation of MSE_u, as defined in (13). On the other hand, the collocation points are situated within the domain and contribute to the evaluation of MSE_f, as defined in (14). In our experiments, we fix the number of training points to $N_u = 400$ and the number of collocation points to $N_f = 10520$. Specifically, there are 100 training points on each edge of the boundary, 2500 collocation points within each subdomain plus 520 collocation points on the boundary of the domain. The subdomains correspond to the ones depicted in Figure 2. It is important to note that N_u represents the total number of training data and is relatively small.

The network architecture is predefined, comprising of a total of 9 layers. It includes one input layer with 2 neurons, followed by 8 hidden layers, each containing 20 neurons. Lastly, there is one output layer with a single neuron. It is worth noting that the input layer is not considered in the count since it does not have tuning parameters. For the optimization method, weight/bias initialization, and activation function, we follow the original network specification proposed by Raisse et al. [2]. Specifically, the loss function is optimized using L-BFGS, which is a quasi-Newton, full-batch gradient-based optimization algorithm [8]. The weights are initialized from a normal distribution with zero mean and a standard deviation equal to $\sqrt{2/(in\_dim + out\_dim)}$, where in_dim and out_dim are the dimensions of the preceding and succeeding layers, respectively. In our implementation, the standard deviation for the first hidden layer is $\sqrt{2/22}$, for the central hidden layers it is $\sqrt{2/40}$, and for the last hidden layer it is $\sqrt{2/21}$. The biases are initialized to 0, and the activation function used is the hyperbolic tangent (tanh). Moreover, the input layer is scaled to ensure that the input data fall within the interval $[-1, 1]$.

Before analyzing the behavior of Physics-Informed Neural Networks with a piecewise constant coefficient function, we examine the influence of the number of training and collocation points, as well as the neural network depth, on the predicted solution. To conduct this analysis, we utilize a simpler problem with a constant diffusion coefficient by setting $\alpha = 1$ throughout the domain. We begin by investigating the role of training and collocation points in three different cases:

1. $N_u = 100$, $N_f = 2628$
2. $N_u = 400$, $N_f = 10520$
3. $N_u = 1600$, $N_f = 42000$.

For each case, the distribution of training points is such that one-fourth of the training points lies on each edge. Within each subdomain, we generate 625, 2500, and 10000 collocation points, respectively, using a Latin Hypercube Sampling strategy. Additionally, we include 128, 520, and 2000 collocation points on the boundary of the domain for each corresponding case.

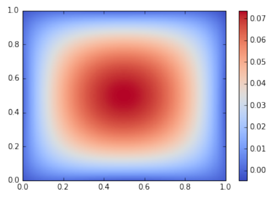
Figure 7 shows the predicted solution obtained by varying the number of total points. Specifically, Figures 7a and 7b correspond to Case 1, Figures 7c and 7d to Case 2 and Figures 7e and 7f to Case 3. The neural network architecture is fixed to 9 layers, with each hidden layer containing 20 neurons. Table 1 reports the loss value and training time for each case. It is important to note that in all cases, after training the neural network, the plots are generated by computing the predicted solution on a uniform grid of size 500×500. Consequently, the resolution of the plots in Figures 7 remains consistent, despite the NNs being trained with varying numbers of total points. Although it is difficult to discern significant differences between the three cases based on the graphical representations in Figures 7, it is evident that, in all three cases, the boundary conditions are not well satisfied. This discrepancy is apparent as the predicted solutions exhibit wave-like behavior at the boundary, whereas the exact solution is identically zero on the entire boundary. Based on the performance

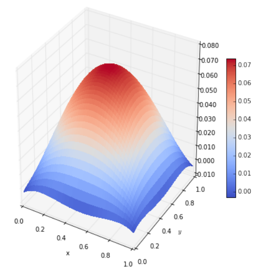
PINNs. Performance evaluation for different number of total points. Diffusion coefficient $\alpha = 1$

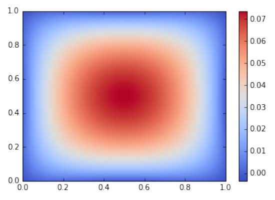
N_u	100	400	1600
N_f	2628	10520	42000
training time	37.32	80.55	254.89
MSE	$2.85\,10^{-6}$	$2.65\,10^{-6}$	$3.02\,10^{-6}$
MSE_u	$1.16\,10^{-6}$	$1.18\,10^{-6}$	$1.72\,10^{-6}$
MSE_f	$1.68\,10^{-6}$	$1.46\,10^{-6}$	$1.29\,10^{-6}$

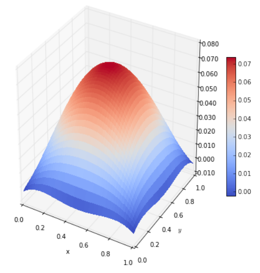
Table 1: *Performance of the PINNs method with respect to number of the total points. Fixed hyperparameters: NN with 9 hidden layers and 20 neurons for each hidden layers, $\alpha = 1.0$. Here MSE is the loss function, MSE_u is the data loss and MSE_f is the physics loss.*

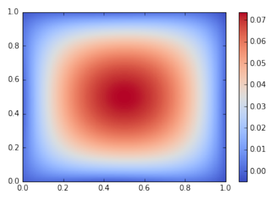
results presented in Table 1, it becomes evident that


(a) *PINNs. 2D plot:* $N_u = 100$, $N_f = 2628$


(b) *PINNs. 3D plot:* $N_u = 100$, $N_f = 2628$


(c) *PINNs. 2D plot:* $N_u = 400$, $N_f = 10520$


(d) *PINNs. 3D plot:* $N_u = 400$, $N_f = 10520$


(e) *PINNs. 2D plot:* $N_u = 1600$, $N_f = 42000$

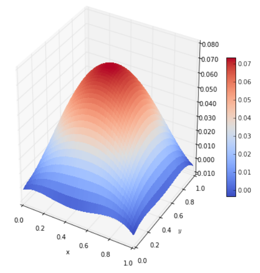

(f) *PINNs. 3D plot:* $N_u = 1600$, $N_f = 42000$

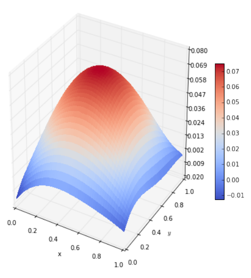
Figure 7: *PINNs. Approximate solutions for different numbers of total points: 2D and 3D representation. Fixed hyperparameters: NN with 8 hidden layers and 20 neurons for each hidden layers, $\alpha = 1.0$. **(a), (b)** $N_u = 100$ and $N_f = 2628$. **(c), (d)** $N_u = 400$ and $N_f = 10520$. **(e), (f)** $N_u = 1600$ and $N_f = 42000$. Here N_u is the number of training points and N_f is the number of collocation points.*

increasing the number of total points does not necessarily lead to a smaller mean square error (MSE) loss function. Conversely, the training time experiences a significant increase. Additionally, it is noteworthy that the data loss (MSE_u) and the physics loss (MSE_f) exhibit comparable values across all three cases. Finally, in each case, the method terminates within less than 3000 iterations, as the convergence criterion is satisfied. The convergence criterion depends on the gradient of the loss function.

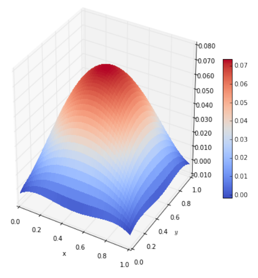
Based on these observations, we decided to set the number of training points to $N_u = 400$ and the number of collocation points to $N_f = 10520$ (Case 2) for the subsequent analysis. Proceeding with the investigation of the NN's depth and its impact on the method's performance, we consider four cases:

1. Shallow NN: 1 hidden layer
2. 4 hidden layer
3. 8 hidden layer
4. 12 hidden layer.

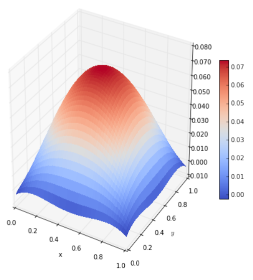
In each case, the input layer consists of 2 neurons, the hidden layers consist of 20 neurons, and the output layer consists of 1 neuron. Figure 8 displays the predicted solutions obtained by varying the depth of the NN. Since the 2D representations do not provide sufficient information, we present only the 3D representations. Specifically, Figure 8a represents the predicted solution for 1 hidden layer, Figure 8b for 4 hidden layers, Figure 8c for 8 hidden layers, and Figure 8d for 12 hidden layers. It is important to note that the number of training and collocation points remains fixed at $N_u = 400$ and $N_f = 10520$, respectively. Table 2 provides the loss values and training times for each case.



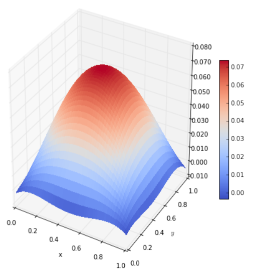
(a) *PINNs. 1 Hidden Layer.*



(b) *PINNs. 4 Hidden Layers.*



(c) *PINNs. 8 Hidden Layers*



(d) *PINNs. 12 Hidden Layers*

Figure 8: *PINNs. 3D representation of the approximate solutions varying the depth of the NN. Fixed hyperparameters: training and collocation points, $N_u = 400$ and $N_f = 10520$, and $\alpha = 1.0$.* **(a)** *Shallow NN: 1 hidden layer.* **(b)** *4 hidden layers.* **(c)** *8 hidden layers.* **(c)** *12 hidden layers. Here N_u is the number of training points and N_f is the number of collocation points.*

Upon examining the graphical representations in Figures 8, it becomes apparent that the shallow neural network fails to capture the boundary conditions. This observation is further supported by Table 2,

PINNs. Performance evaluation for different depths of the neural network.
Diffusion coefficient $\alpha = 1$

hidden layers	1	4
training time	14.11	31.06
MSE	$3.44\,10^{-5}$	$1.54\,10^{-6}$
MSE_u	$1.50\,10^{-5}$	$1.01\,10^{-6}$
MSE_f	$1.94\,10^{-5}$	$5.36\,10^{-7}$
hidden layers	8	16
training time	80.76	97.59
MSE	$2.65\,10^{-6}$	$3.50\,10^{-6}$
MSE_u	$1.18\,10^{-6}$	$2.01\,10^{-6}$
MSE_f	$1.46\,10^{-6}$	$1.48\,10^{-6}$

Table 2: *Performance of the PINNs method with respect to the depth of the NN. Fixed hyperparameters: training and collocation points, $N_u = 400$ and $N_f = 10520$, and $\alpha = 1.0$. Here MSE is the loss function, MSE_u is the data loss and MSE_f is the physics loss.*

where the shallow NN exhibits the highest data loss (MSE_u). For the remaining three cases, a comparison based on performance values from Table 2 is more appropriate, as the 3D representations in Figures 8b, 8c, and 8d appear very similar. It is worth noting that all predicted solutions exhibit a wave-like behavior at the boundary. Regarding the deep neural networks, the performance measurements in Table 2 clearly indicate that, as the number of hidden layers increases, both the loss (measured by MSE) and the training time also increase. This behavior is also evident in both the data loss (MSE_u) and the physics loss (MSE_f). However, the values of the loss functions remain comparable, with all of them being on the order of 10^{-6}. Based on these observations, although the NN with 4 hidden layers demonstrates the best behavior, we have chosen to focus on studying the 2D steady Darcy problem with a piecewise constant diffusion coefficient using an NN with 8 hidden layers. This decision is driven by the fact that the solution of the PDE with a non-constant diffusion coefficient exhibits a more complex behavior. In such case, a 9-layer NN provides a better approximation of the solution compared to a 5-layer NN.

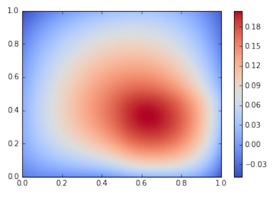
We conduct a similar analysis for the PINNs method as we have done for the Finite Element method, investigating how well it approximates the solution of the Darcy problem (1) for three different values of the piecewise constant diffusion coefficient. The results presented here were obtained by fixing the number of training and collocation points at $N_u = 400$ and $N_f = 10520$, respectively, and employing a neural network architecture with 9 layers, each hidden layer containing 20 neurons. The first two figures, 9a and 9b, correspond to $\alpha = [0.64, 0.93, 0.88, 0.20]$, the central two figures, 9c and 9d, correspond to $\alpha = [0.25, 0.74, 0.26, 0.18]$, and finally the last two figures, 9e and 9f, correspond to $\alpha = [30, 10, 60, 100]$. Referring to Figure 2, we assign $\alpha[0]$ to Ω_1, $\alpha[1]$ to Ω_2, $\alpha[2]$ to Ω_3, and $\alpha[3]$ to Ω_4.

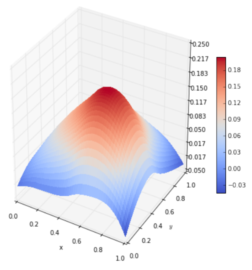
Figure 9 clearly demonstrates that the PINNs method performs poorly when the diffusion coefficient varies throughout the domain. The capability of the PINNs method to accurately predict a solution diminishes as the diffusion coefficient exhibits significant variation, as illustrated in Figure 9e and 9f, where $\alpha = [30, 10, 60, 100]$. This observation is supported by the performance evaluation presented in Table 3. However, the training time does not vary significantly for different values of the diffusion coefficient. To properly comment the obtained loss function values, it is necessary to consider the range of the function's image. For instance, a loss function on the order of 10^{-3} for a solution within the interval [0.0, 0.007] should be regarded as large, indicating that the method does not perform well. Once again, the neural networks appear to struggle in learning the boundary conditions. It is worth noting that for higher values of α, the approximated solution tends to have smaller magnitudes.

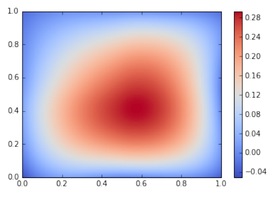
PINNs. Performance evaluation for different values of the diffusion coefficient α.

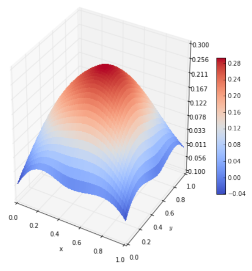
diffusion coefficient	α_1	α_1	α_3
training time	62.42	82.73	76.10
MSE	$8.06\,10^{-3}$	$4.78\,10^{-3}$	$9.80\,10^{-2}$
MSE_u	$4.34\,10^{-4}$	$1.80\,10^{-4}$	$1.22\,10^{-5}$
MSE_f	$7.62\,10^{-3}$	$4.60\,10^{-3}$	$9.80\,10^{-2}$

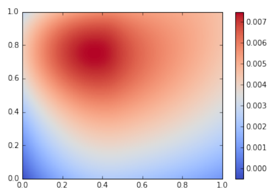
Table 3: *Performance of the PINNs method with respect to the piecewise constant diffusion coefficient α. Fixed hyperparameters: training and collocation points, $N_u = 400$ and $N_f = 10520$, NN with 9 layers with 20 neurons for each hidden layer. Here MSE is the loss function, MSE_u is the data loss and MSE_f is the physics loss. $\alpha_1 = [0.64, 0.93, 0.88, 0.20]$, $\alpha_2 = [0.25, 0.74, 0.26, 0.18]$ and $\alpha_3 = [30, 10, 60, 100]$*


(a) *PINNs. 2D plot:* $\alpha = [0.64, 0.93, 0.88, 0.20]$


(b) *PINNs. 3D plot:* $\alpha = [0.64, 0.93, 0.88, 0.20]$


(c) *PINNs. 2D plot:* $\alpha = [0.25, 0.74, 0.26, 0.18]$


(d) *PINNs. 3D plot:* $\alpha = [0.25, 0.74, 0.26, 0.18]$


(e) *PINNs. 2D plot:* $\alpha = [30, 10, 60, 100]$

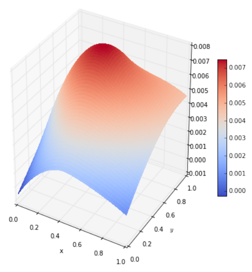

(f) *PINNs. 3D plot:* $\alpha = [30, 10, 60, 100]$

Figure 9: *PINNs. Approximate solution for different piecewise constant diffusion coefficient: 2D and 3D representation. Fixed hyperparameters: training and collocation points, $N_u = 400$ and $N_f = 10520$, and NN with 9 layers with 20 neurons for each hidden layer.* ***(a), (b)*** $\alpha = [0.64, 0.93, 0.88, 0.20]$. ***(c), (d)*** $\alpha = [0.25, 0.74, 0.26, 0.18]$. ***(e), (f)*** $\alpha = [30, 10, 60, 100]$.

In conclusion, based on our implementations, we can deduce that PINNs is capable of capturing the general behavior of the solution to the 2D steady Darcy equation (1) within a relatively short time. However, the method lacks precision, particularly when it comes to handling the boundary conditions. One potential approach to address this issue is to modify the optimizer for the loss function. In our implementations, we consistently utilized the L-BFGS method. It is worth noting that in all the examples presented in this section, the optimization method converged in less than 3500 iterations, suggesting that the loss function became trapped in a local minimum. It would be interesting to explore the implementation of the PINNs method with the ADAM optimizer. Nonetheless, this is not a straightforward task, as ADAM is not implemented in TensorFlow 1. One potential solution is to migrate the code from TensorFlow 1 to TensorFlow 2. Furthermore, improvements to the network could be achieved by considering that the loss function had equal weighting among the different loss components, therefore ignoring any gradient contribution from each of the terms. Such weights might be tuned while training the neural network [9].

To ensure reproducibility of the results, we utilized fixed seeds by incorporating the following code:

```
import numpy as np
import tensorflow as tf

np.random.seed(1234)
tf.set_random_seed(1234)
```

Additionally, the optimization method variables were selected as follows:

```
method = L-BFGS-B,
options = maxiter: 50000,
          maxfun: 50000,
          maxcor: 50,
          maxls: 50,
          ftol: 1.*np.finfo(float).eps
```

These measures aim to achieve consistent results and maintain the integrity of the experimentation process.

3.3 FEM vs PINNs

Finite Element Method and Physics Informed Neural Networks are two different approaches used for solving partial differential equations and obtaining numerical solutions. Here, we would like to elaborate on the key differences between FEM and PINNs.

Considering that in order to use FEM, the domain must be divided into smaller finite elements using a mesh, the quality of the mesh plays a crucial role in achieving accurate solutions. For a domain with simple geometry, as is the one used in this study (Figure 2), it is straightforward to generate a high-quality mesh. On the other hand, for irregular or complex domains generating such a mesh while obtaining a satisfactory level of accuracy becomes more challenging. In contrast, PINNs don't rely on the space discretization or a predefined mesh but on collocation points. This means that regularity of the domain shape is less critical and irregular and complex domains are handled more easily because they do not explicitly depend on mesh quality. The collocation points can be placed strategically based on the problem's characteristics, and the network can learn to approximate the solution effectively. Considering we only used a simple geometry domain, we can't directly evaluate the differences, but we conclude that regarding the domain shape, PINNs are more flexible and easier to apply for irregular or complex domains.

The results for the two methods are compared for constant and piecewise constant diffusion coefficient values. Selected diffusion coefficient distribution simplifies the coding process and allows for easier modification, especially since there are only four fixed regular subdomains involved. Implementing a more complex distribution of the diffusion coefficient would pose greater difficulty. It is important to note that if the shape or number of subdomains change, a new code is required for the Finite Element Method implementation. Additionally, creating a mesh that accounts for highly irregular subdomain divisions can be extremely challenging, if not impossible. Even though Physics-Informed Neural Networks do not require a mesh, incorporating an irregular distribution of the piecewise diffusion coefficient can still present a formidable task. However, drawing inspiration from the Fourier Neural Operator paper [10], one might attempt to address this issue by employing a Convolutional Neural Network (CNN) in the PINNs implementation.

A weak solution satisfies the variational formulation of the PDE, which involves multiplying the differential equation by a test function and integrating it over the entire domain. On the other hand, a strong solution directly solves the partial differential problem (1). It is important to note that every strong solution is also a weak solution, but the opposite is not always true. In FEM, the approximate solution is a continuous piecewise polynomial function of degree less than or equal to 1, whereas in PINNs, the predicted solution is obtained through a neural network. Additionally, FEM aims to approximate the weak solution of the PDE, while PINNs approximate the strong solution, which is evaluated pointwise across the entire domain.

4 Conclusions

FEM and PINNs performance were studied and compared for the 2D steady Darcy problem, considering both a constant and piecewise constant diffusion coefficient on the unit square domain. In both cases, the FEM solutions appeared to be more accurate than those obtained with PINNs. Furthermore, even with significant variations in the diffusion coefficient, the Finite Element Method proved to be a reliable option for efficiently solving the problem. Unfortunately, the same cannot be said for PINNs, which struggled to deliver comparable results.

Moreover, FEM implementation demonstrated faster computational speed compared to PINNs. Based on our results for this specific problem, it can be inferred that classical numerical methods outperformed machine learning techniques. However, PINNs were still able to capture the general behavior of the solution within a relatively short time. Their main challenge lay in accurately mimicking the boundary conditions. It is important to note that FEM has been studied for a century while, PINNs, being a relatively new method suggested only a decade ago, still has significant room for improvement. Additionally, we analyzed these methods in the context of a regular domain with regular subdomain divisions, which is an ideal scenario for FEM. It is likely that with more complex domain shapes and subdomain divisions, the two methods would exhibit different behaviors and performance compared to what was presented here.

Acknowledgements

All contributors to this report are part of the CompSci doctoral program which is managed by the Faculty of Mathematics and Natural Sciences at the University of Oslo (UiO). The program is partly funded by the EU Horizon 2020 under the Marie Skłodowska-Curie Action (MSCA) - Co-funding of Regional, National and International Programmes (COFUND).

Competing interests

The authors have no competing interests to declare.

References

[1] Nathan Baker et al. "Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence". In: (Feb. 2019). DOI: `10.2172/1478744`. URL: `https://www.osti.gov/biblio/1478744`.

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

[3] Alfio Quarteroni and Alberto Valli. *Numerical approximation of partial differential equations*. Vol. 23. Springer Science & Business Media, 2008.

[4] In: (). URL: `%5Curl%7Bhttps://www.youtube.com/watch?v=zYi8KO4rLwg%7D`.

[5] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Springer, 2017. ISBN: 978-3-319-52461-0. DOI: `10.1007/978-3-319-52462-7`.

[6] Scott T. M. Arzani Amirhossein; Dawson. "Data-driven cardiovascular flow modelling: examples and opportunities". In: *Journal of the Royal Society Interface* 18 (2021). URL: `https://doi.org/10.1098/rsif.2020.0802`.

[7] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[8] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* 45.1-3 (1989), pp. 503–528.

[9] Perdikaris P. Wang S. Teng Y. "Understanding and mitigating gradient flow pathologies in physics-informed neural networks". In: *J. Sci. Comput.* (2021).

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

Supplementary Files

The code can be found at This GitHub repository in the folder *Projects 2 and 3*.

Appendix

Notation	Meaning
Darcy equation	
Ω	domain
$\partial\Omega$	boundary of the domain
s	forcing/source function
α	diffusion coefficient
FEM	
$a(\cdot,\cdot)$	bilinear form
$S(\cdot)$	linear form
V	Hilbert space $H_0^1(\Omega)$
$\mathscr{T}_h$	triangulation of the domain (mesh)
V_h	finite-dimension space
$v \in V$	test function
$u \in V$	solution
$u_h \in V_h$	approximate solution
PINNs	
f	physics informed NN
MSE	mean square error (loss)
MSE_f	physics loss
MSE_u	data loss
N_f	number of collocation points
N_u	number of training points

Table 4: *List of notations used in this work.*