JAXPINNs: Extended Physics-Informed Neural Networks implemented with JAX.

Femtehjell, Winsvold, Steeneveldt, Johannessen and Hu

Department of Mathematics

(Dated: March 2024)

We develop a framework in order to simplify the implementation of extended physics-informed neural networks (XPINNs). The proposed implementation aims to simplify the process of setting up an XPINN, through automating the domain decomposition and initial set up of the networks. This allows the user to focus on the problem specific details of a given issue, rather than the underlying details of how everything is calculated and communicated. With this, we were able to utilize XPINNs for solving a number of different partial differential equations (PDEs) without having to worry about errors in the underlying network. We compared our results on the advection, Poisson and Navier-Stokes equations with that of a singular physics-informed neural network (PINN), getting worse, but comparable, results. As XPINNs decompose the domain, each sub-PINN is trained on a portion of the total points available, and without implementing parallelization, we were not able to increase the number of points without also increasing the total computational time.

CONTENTS

I. INTRODUCTION

The resolution of Partial Differential Equations (PDEs) underpins many scientific and engineering challenges. PDEs model the behaviour of a system of multiple variables with respect to its derivatives, where the problem at hand is to determine the function, rather than a number. It is often cumbersome, or impossible, to determine the analytical solution of a PDE arising from real-world conditions, which has led to the development of numerous numerical methods. A benefit of these numerical methods, is that the theoretical groundwork behind them has been extensively explored, making them more analytically transparent. However, the computational and analytical burdens associated with implementing such methods are often costly. As neural networks have a proven capacity of being able to act as universal approximators of functions (Cybenko, 1989), it is therefore natural to explore whether they are capable of solving PDEs.

A large part of the recent advancements in machine learning can be contributed to a higher degree of data availability and increased computational resources. With problems involving PDEs however, data acquisition can be an issue in and of itself. Due to this, recent methods combat this through incorporating the underlying knowledge we have about the system directly into the training of the neural network, in a framework aptly named Physics-Informed Neural Networks (PINNs) in Raissi *et al.* (2019). With these networks, we gain a greater degree of certainty regarding the accuracy of the predictions, as they inherently attempt to abide by the physical laws of the system.

This is primarily done through punishing the PINN in the training stage for not adhering to the PDE conditions. However, there are other facets of a problem one might concentrate on in order to increase predictive power. One of these stem from the inherent discontinuities present in certain problems, or the wildly differing behaviours across the spatial- or temporal dimension(s). A number of extensions of PINNs have therefore been proposed, including but not limited to, Conservative PINNs (cPINNs) (Jagtap

et al., 2020), Extended PINNs (XPINNs) (Jagtap and Karniadakis, 2020) and Augmented PINNs (APINNs) (Hu *et al.*, 2023). These methods primarily focus on various forms of domain decomposition.

We focus our attention in this report on Extended Physics-Informed Neural Networks. In this method, the domain is decomposed into subdomains consisting of hard boundaries, with a PINN trained on each subdomain. The goal of this method is to capture more homogeneous behaviour, in order to reduce the necessary complexity of the network and increase convergence speeds. We attempt to reproduce some of the results from Hu *et al.* (2022).

In order to achieve this, we provide a framework for the implementation of such problems, with a focus on modularity in order to accommodate for a wide variety of problems. We therefore utilize `JAX` (Bradbury *et al.*, 2018) in order to achieve good performance in `Python`, with `Optax` (DeepMind *et al.*, 2020) in order to perform the optimization steps.

We begin by covering the necessary preliminaries of neural networks and PDEs, before covering how these topics combine in PINNs. Next, we introduce XPINNs, before explaining how automatic differentiation functions. Following, we cover the necessary methodology used within our implementation, such as utilizing Adam as our optimizer. Finally, we test our implementation on the advection, Poisson and Navier-Stokes equations, comparing the performance against a typical PINN.

II. THEORY

A. Neural Networks

Neural networks are built upon a simplified understanding of how the neurons in the brain function, wherein electrical impulses are passed in a chain reaction throughout the brain. Feed-forward neural networks typically model this by passing an input through a composition of linear transformations and non-linear activation functions. In a so-called *feed-forward pass*, the input values $\boldsymbol{x} \in \mathbb{R}^{N_0}$ are sent to each neuron in the next layer, multiplied by a predetermined weight along the way with a value added called the bias $\boldsymbol{b} \in \mathbb{R}^{N_1}$. We represent this through the function $\mathcal{A} : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ defined by

$$\mathcal{A}_i(\boldsymbol{x}) = W_i \boldsymbol{x} + \boldsymbol{b}_i, \tag{1}$$

where $W \in \mathbb{R}^{N_i \times N_{i-1}}$ is the weight matrix, where the N_i is the number of neurons in layer i. The resulting values are denoted $\boldsymbol{z}^i$.

The result of this is then passed through an activation function $\sigma_i : \mathbb{R} \to \mathbb{R}$, which we apply element-wise by $\boldsymbol{a}^i_j = \sigma_i(\boldsymbol{z}^i_j)$. To simplify notation, we simply notate this as $\boldsymbol{a}^i = \sigma_i(\boldsymbol{z}^i)$. This continues throughout each layer, culminating in the output layer, which in our case will simply have a the identity function as its activation function. This process is illustrated in Figure 1.

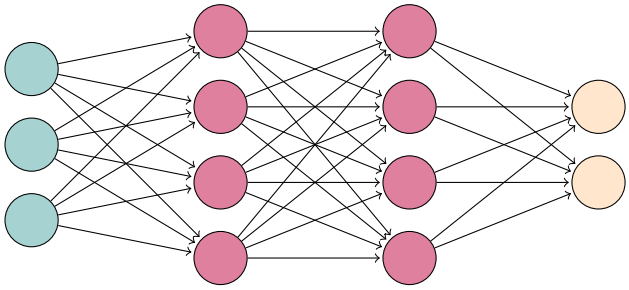

FIG. 1: Illustration of a fully connected feed-forward neural network with an input layer (teal), two hidden layers (purple) and an output layer (beige).

Let $\boldsymbol{\theta} = \{W_i, \boldsymbol{b}_i\}_{i=1}^{L}$ represent the trainable parameters of the network in the parameter space $\mathcal{V}$, and $\mathcal{N}_{\boldsymbol{\theta}} : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ denote the realization of a neural network with L layers, defined through the composition

$$\begin{aligned} \mathcal{N}_{\boldsymbol{\theta}} &= \sigma_L \circ \mathcal{A}_L \circ \sigma_{L-1} \circ \ldots \circ \sigma_1 \circ \mathcal{A}_1 \\ &= \bigcirc_{i=1}^{L} \sigma_i \circ \mathcal{A}_i. \end{aligned} \tag{2}$$

We call the parameters trainable, as we typically initialize them to random values, *training* them through optimization with the help of the backpropagation algorithm. We do this by measuring how well our model is able to predict a set of values with a cost function $\mathcal{C}$, which often gives the problem the form of finding

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta} \in \mathcal{V}} \mathcal{C}\left(\mathcal{N}_{\boldsymbol{\theta}}, \boldsymbol{x}, \hat{\boldsymbol{y}}\right), \tag{3}$$

where $\hat{\boldsymbol{y}} \in \mathbb{R}^{N_L}$ are a set of target values.

A typical choice of $\mathcal{C}$ in a regression problem is the Mean Squared Error (MSE), defined by

$$\mathcal{C}(\mathcal{N}_\theta, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n} \|\mathcal{N}_\theta(\boldsymbol{x}) - \boldsymbol{y}\|_2^2. \tag{4}$$

One of the main benefits of this measure is its ease of differentiation, lightening the computational load of training the network.

Through the Universal Approximation theorem, first proven by Cybenko (1989) and since extended, neural networks have a proven capacity to approximate any given function. It does not state however how one is to find the architecture and parameters which solve a given problem. The theorem does however guarantee the validity of at least attempting to utilize neural networks in a number of fields.

B. Partial Differential Equations

A Partial Differential Equation (PDE) is defined to be an equation containing a function with at least two different variables, as well as some degree of partial derivatives

to these variables. PDEs are ubiquitous in describing the properties, behavior, and evolution of physical systems.

An important concept when discussing PDEs is differential operators. A differential operator is defined as a function performing some sort of differentiation on the function it is applied to. It is indeed a function from a function space $\mathcal{F}_1$ to another function space $\mathcal{F}_2$. In the case of a function of n variables $u(x_1, x_2, \ldots, x_n)$ the usual differential operator is defined as

$$D^\alpha u = \frac{\partial u^{|\alpha|}}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \cdots \partial x_n^{\alpha_n}},$$

where $|\alpha| = \alpha_1 + \alpha_2 + \ldots + \alpha_n$. When considering a function $u(x, y)$, we will denote $\frac{\partial u}{\partial x}$ as u_x, $\frac{\partial^2 u}{\partial x^2}$ as u_{xx}, and so on. We will when discussing general PDEs denote the differential operator characterizing the PDE by $\mathcal{L}$.

An example of a PDE is Poisson's equation, which is characterized by

$$u_{xx} + u_{yy} = f \text{ on } \Omega,$$

where the residual f is independent of u, on the domain $\Omega \in \mathbb{R}^d$, where d is the number of dimensions. In the 2D case, we have $\mathcal{L}u = u_{xx} + u_{yy}$, so that the equation can be denoted succinctly as $\mathcal{L}u = f$.

We will also come across the differential operator ∇. For a problem in three physical dimensions, ∇ is defined as $\nabla = \frac{\partial}{\partial x}\boldsymbol{x} + \frac{\partial}{\partial y}\boldsymbol{y} + \frac{\partial}{\partial z}\boldsymbol{z}$, where $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ are the three-dimensional Euclidean unit vectors.

When considering time-dependent PDEs for physical problems, we frequently require initial and boundary conditions. Initial conditions are conditions for values at $t = 0$, and boundary conditions are conditions on the physical boundaries of the problem. Thus, both conditions are defined on the boundary $\partial\Omega$ of our spatial-temporal domain, and we treat them similarly. In this paper we encounter Dirichlet-type boundary conditions, where the value of the function on the boundary is given. We return to the Poisson equation as an example; we have for the function $u(x, y)$, $(x, y) \in [0, 1] \times [0, 1]$, the boundary conditions $u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0$. Neumann boundary conditions are also ubiquitous. This is when the boundary conditions determine the first order derivative in the direction normal to the boundary.

To capture these boundary conditions, we introduce a boundary value operator $\mathcal{B}$ such that we can write $\mathcal{B}u = g$ on $\partial\Omega$ for all our boundary conditions, for some function g on the boundary $\partial\Omega$. Given the PDE operator $\mathcal{L}$ and the boundary value operator $\mathcal{B}$, we present the general form of a PDE as

$$\begin{cases} \mathcal{L}u = f & \text{in } \Omega, \\ \mathcal{B}u = g & \text{in } \partial\Omega. \end{cases} \tag{5}$$

C. Physics-Informed Neural Networks

Physics-Informed Neural Networks (PINNs) incorporate the inherent physical constraints of a problem directly into the network. One such way is to bake in the PDE conditions of a problem directly into the loss function of the network (Raissi *et al.*, 2019). In doing so, we coerce the network into predicting results which adhere to the inherent physical laws. In this way, we reduce the need for labeled data, and are able to be more confident in the predictions of our network. Our neural network $\mathcal{N}_{\boldsymbol{\theta}}$ can then be regarded as an approximation of the hidden function u, denoting it as u^*.

We then measure the loss through

$$\text{MSE} = \text{MSE}_d + \text{MSE}_0 + \text{MSE}_u + \text{MSE}_f, \tag{6}$$

where

$$\begin{aligned}
\underbrace{\text{MSE}_d}_{\text{data}} &= \frac{w_d}{N_d} \sum_{i=0}^{N_d} \left(u^*(x_d^i) - u_d^i\right)^2, \\
\underbrace{\text{MSE}_0}_{\text{Initial}} &= \frac{w_0}{N_0} \sum_{i=0}^{N_0} \left(u^*(x_0^i) - u_0^i\right)^2, \\
\underbrace{\text{MSE}_u}_{\text{Boundary}} &= \frac{w_u}{N_u} \sum_{i=0}^{N_u} \left(u^*(x_u^i) - g(x_u^i)\right)^2, \\
\underbrace{\text{MSE}_f}_{\text{Residual}} &= \frac{w_f}{N_f} \sum_{i=0}^{N_f} \left(\mathcal{L}u^*(x_f^i) - f(x_f^i)\right)^2.
\end{aligned}$$

In this case, $(x_d^i, u_d^i)_{i=1}^{N_d}$ is a sample of data, u_0^i is our initial condition, and N_d, N_0, N_u, N_f is the number of data-, initial-, boundary-, and residual points on which the loss is calculated respectively. w_d, w_0, w_u, w_f are the respective weights given to each loss. With this baseline, one can extend the loss function through appending more terms, if there are other conditions, such as periodicity, one would like to enforce. Note that in MSE_f, we are taking the derivative of the network itself with respect to its output. This is done through the use of automatic differentiation, in much the same way the parameters are optimized, described further in section II.E.

The MSE_d can also be dropped, leading to unsupervised training. However, unsupervised PINNs have been documented to be slower than traditional numerical methods for solving PDEs (Ang *et al.*, 2023). Nevertheless, with a framework already set up, the process of setting up a PINN is quite simple, in contrast to developing a numerical algorithm.

The PINN framework therefore presents a simple methodology to incorporating knowledge about the systems we wish to solve into our neural network training. Moreover, PINNs also retain the data-based learning from machine learning models, allowing the user to train using

existing data. This approach has made PINNs particularly suited for inverse problems. Inverse problems involve determining some parameters of our system using data, and are notoriously difficult for traditional methods (Garay *et al.*, 2023; Jagtap *et al.*, 2022).

D. Extended Physics-Informed Neural Networks

Extended Physics-Informed Neural Networks (XPINNs) is a generalized space-time domain decomposition algorithm for PINNs initially proposed in Jagtap and Karniadakis (2020). The key idea is to decompose the domain Ω into several subdomains $(\Omega_s)_s^N$ such that $\bigcup_{s=1}^{N_{\text{sd}}} \Omega_s = \Omega$, as illustrated in Figure 2. The domain interfaces are given by $\Omega_i \cap \Omega_j = \partial\Omega_{ij}$, $i \neq j$. Each subdomain is then equipped with its own sub-PINN u_s^*, and the training and test points associated with its subdomain.

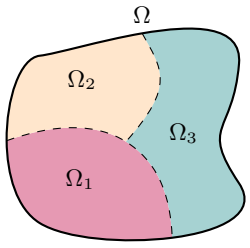

FIG. 2: Domain decomposition of Ω into three subdomains Ω_i, with dashed lines representing the interfaces $\partial\Omega_{ij}$. Each subdomain is equipped with its own unique PINN.

Given such a domain decomposition, we now consider the following XPINN solution to our PDE problems:

$$u^*(x) = \sum_{s=1}^{N_{\text{sd}}} u_s^*(x) \mathbb{1}_{\Omega_s}(x), \tag{7}$$

where $\mathbb{1}_{\Omega_s}(x)$ is the indicator function given by:

$$\mathbb{1}_{\Omega_s}(x) = \begin{cases} 0 & \text{If } x \notin \Omega_s \\ 1 & \text{If } x \in \Omega_s \backslash \text{common interface in } \Omega_s \\ \frac{1}{S} & \text{If } x \in \text{common interface in } \Omega_s, \end{cases} \tag{8}$$

where S is the number of subdomains intersecting in the common interface.

A classical problem when applying domain decomposition is ensuring sufficient communication between the different subdomains such that their solutions reflect a global solution. Ideally, the domain decomposition solution should be "blind" to the decomposition. For classical solvers, we frequently introduce *ghost layers* to facilitate the communication between neighboring subdomains.

However, as PINNs are mesh-independent we do not require ghost layers to communicate between PINNs. XPINNs instead proposes to amend an additional *interface loss* term MSE_I to the loss function (6) of each PINN to ensure that the PINNs agree on the subdomain interfaces. For instance, we might wish to enforce continuity in our solution across an interface. For a single PINN, we amend the loss function with

$$\underbrace{\text{MSE}_I}_{\text{Interface}} = \frac{1}{N_I} \sum_{i=0}^{N_I} \left(u^*(x_I^i) - \{u_I^i\}\right)^2, \tag{9}$$

where $\{u_I^i\}$ is the average predicted value at the interface from each PINN, and N_I is the number of points on the interface.

Like traditional domain decomposition algorithms, XPINNs introduce an intuitive parallelization by separating the domain into several subdomains with their own respective neural network where the subdomain solutions can be computed on independent processors. Thus, XPINNs might improve the computational efficiency through improved utilization of the processor architecture in a computer.

Moreover, Jagtap and Karniadakis (2020) argue that XPINNs also introduce architectural flexibility into our problem. For instance, we might employ a deep network in a subdomain with complex solution, whereas we can apply a shallow network for a subdomain with relatively simple and smooth solutions.

However, the domain decomposition introduces a trade-off for generalization. On the one hand, we can leverage the architectural flexibility of XPINNs to decompose a complex PDE solution into several simple parts, decreasing the problem complexity and boosting generalization. On the other hand, decomposing leads to each subdomain having access to less training data, which might cause overfitting and worse performance overall. However, in an unsupervised setting, the computational load can be balanced through introducing more points in a previously lighter subdomain. For a more rigorous analysis of when XPINNs improve generalization, we refer the reader to (Hu *et al.*, 2022).

Furthermore, the interface conditions also introduce complications to the loss function, changing the loss landscape. Thus, it is easy to imagine instances where for instance the loss landscapes of interface loss and boundary loss are different, resulting in an optimization dilemma of which condition to prioritize. Moreover, changing the loss function through interface conditions might also correspond to altering the PDE problem equation. To the extent of the authors' knowledge, there is not a clear theoretical understanding of how the interface conditions alter the solution to our PDE problem.

E. Automatic Differentiation

The backpropagation algorithm (Rumelhart *et al.*, 1986) forms the backbone of training a neural network. It is the method described in (13), wherein we take the derivative of the cost function $\mathcal{C}$ with respect to the parameters $\boldsymbol{\theta}$.

$\mathcal{C}$ is a deeply nested function as it involves the realization of the neural network $\mathcal{N}$. Differentiating this with respect to each parameter of the network is therefore quite a costly task.

There are in essence three methods for calculating this derivative, namely numerical, symbolic or automatic differentiation. Numerical differentiation relies on the method of finite differences, through approximations such as

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h}, \quad (10)$$

for small h. This method can be quite prone to numerical errors, however in a machine learning context this is not a major concern. What is concerning however, is the fact that we would need to calculate the derivative with respect to each parameter of the network. As we are potentially working with a huge amount of parameters, this is not feasible

Symbolic, or algorithmic, differentiation refers to a process much like one you would use when differentiating by hand. The derivatives of elementary functions are hard coded, such that complex expressions can be differentiated through the chain rule. The reason we don't utilize this method, is that the expressions have a tendency to grow in size. Consider as an example the function $f(x) = g(x) \cdot h(x)$, with derivative $f'(x) = g'(x)h(x) + g(x)h'(x)$. As a neural network is essentially a deeply nested composite function as described by (2), this quickly becomes expensive to compute.

Automatic differentiation tackles this issue through keeping track of which terms are repeated throughout the computation. Consider an extremely simple neural network, with one input and no hidden layer. The output y is then simply computed as

$$y = \sigma\left(x \cdot w + b\right), \quad (11)$$

for an input x, weight w, bias b, and activation function σ. The computational graph associated with this is illustrated in Figure 3, with $\alpha, \beta, \gamma, \varepsilon$ representing the intermediate values.

Say we are interested in computing the derivative of y with respect to the bias b, $\frac{\partial y}{\partial b}$. As y is not directly dependent on b, we compute the derivatives backward, as illustrated in Figure 4, through extensive utilization of the chain rule. We compute the derivatives with respect to each intermediate function through the chain rule, as

$$\frac{\partial \varepsilon}{\partial b} = \frac{\partial \beta}{\partial b}\frac{\partial \varepsilon}{\partial \beta} = \frac{\partial \beta}{\partial b}\frac{\partial \gamma}{\partial \beta}\frac{\partial \varepsilon}{\partial \gamma}. \quad (12)$$

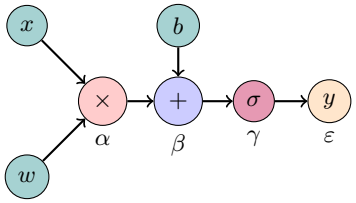

FIG. 3: Computational graph of a feed forward pass of a simple neural network as in (11), adapted from (Mehta, 2021).

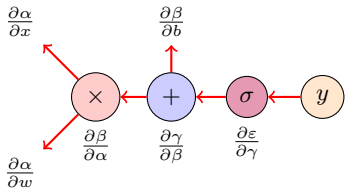

FIG. 4: Automatic differentiation of a simple neural network as in (11), adapted from (Mehta, 2021).

For one derivative, this is not particularly impressive.

However, if we are next interested in computing the derivatives with respect to the other variables, we already have $\frac{\partial \varepsilon}{\partial \beta}$ computed. This memoization is the reason automatic differentiation is able to compute derivatives efficiently, especially for functions with a larger number of inputs or outputs, e.g. a neural network with a large number of parameters.

III. METHOD

A. Domain Decomposition

We implement our domain decomposition framework with flexibility in mind, in order to allow for easy experimentation, designed to integrate seamlessly with our XPINN implementation. The computational domain, representing the physical and temporal space over which our problem is defined, is partitioned into multiple subdomains.

The subdomains themselves are composed of additive and subtractive polygons and circles, allowing for a wide variety of geometric boundaries. The decomposition generated inherently defines the number of sub-PINNs, as well as the boundaries, meaning this does not need to be specified again later, other than loading in the generated points. The goal of domain decomposition is, among other aspects, division of the total workload required, increasing computational efficiency. It also offers an avenue for multiprocessing, which can quickly become cumbersome with traditional neural networks.

B. Adam

The problem at the heart of machine learning is, in effect, solving (3) given a network architecture detailing the sizes and number of hidden layers and the activation functions. This is done iteratively through the backpropagation algorithm, wherein the gradient of each layer with respect to the weights are calculated. The weights can then be adjusted through e.g. gradient descent, with

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \gamma \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}, \tag{13}$$

where $\gamma \in \mathbb{R}_+$ is the step size or learning rate.

It can be shown that for small enough γ we have $\mathcal{C}(\boldsymbol{\theta}^i) \geq \mathcal{C}(\boldsymbol{\theta}^{i+1})$, leading to convergence to a local minima. With convex functions, this is sufficient, as a local minima is then also a global minima. However, as permuting a layer of neurons and weights in a neural network leads to the same output, they are not convex. A number of different optimization algorithms have been devised, often with a focus on tackling the non-convex nature of optimizing the weights of a neural network.

One such method is the Adaptive Moment Estimation (Adam) (Kingma and Ba, 2017) algorithm, which has grown incredibly popular in recent years. The algorithm is defined by

$$\begin{aligned} g_t &\leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{C}(\boldsymbol{\theta}_t) \\ m_{t+1} &\leftarrow \frac{\beta_1 \cdot m_t + (1-\beta_1) \cdot g_t}{1-\beta_1^{t+1}} \\ v_{t+1} &\leftarrow \frac{\beta_2 \cdot v_t + (1-\beta_2) g_t^{\circ 2}}{1-\beta_2^{t+1}} \\ \boldsymbol{\theta}_{t+1}^{(n)} &\leftarrow \boldsymbol{\theta}_t^{(n)} - \frac{\gamma}{\sqrt{\delta + v_{t+1}^{(n)}}} m_{t+1}^{(n)}, \end{aligned} \tag{14}$$

where m, v are the 1st and 2nd moment vectors, initialized to $\mathbf{0}$, $\beta_1, \beta_2 \in [0, 1)$ are decay rates for the moment estimates, $x^{(n)}$ denotes the nth component of the vector, $g^{\circ 2}$ denotes the elementwise square $g \odot g$ and $\delta > 0$ is a numerical stabilizer.

This algorithm has effectively become the de facto standard for optimizing neural networks, however it is not without its problems. In fact, it can fail to converge even for simple convex functions (Reddi *et al.*, 2019), in contradiction to what was claimed in (Kingma and Ba, 2017). However, Adam has empirically performed well, which is why it grew popular in the first case. This might have had an effect on the development of new methods within machine learning, as new methods are typically tested using Adam as the optimizer. One might then question if promising methods have been discarded simply because they didn't perform well with Adam, as it might then have been assumed that the method itself was without merit. However, discussing this issue in depth falls outside the scope of this paper.

Nevertheless, we utilize Adam in our implementation, using the `Optax` (DeepMind *et al.*, 2020) optimization library for `JAX` (Bradbury *et al.*, 2018).

C. JAX technicalities

In implementing, we focused on achieving good performance while keeping a certain degree of modularity. This simplifies the process of changing key parameters like the subdomain decomposition, or the problem at hand. The natural choice for implementing this was then to utilize `JAX` (Bradbury *et al.*, 2018) within `Python`. High-Performance Computing (HPC) and Object-Oriented Programming (OOP) are two programming paradigms which are often at odds with each other, stemming from the obfuscation of the underlying computations necessary to allow for a simple interface. We also lose out on possible problem-specific optimizations, as the program has to work in a wide variety of settings.

`JAX` serves as an important component, as it allows for efficient matrix / vector operations, which are the most computationally heavy aspect of most machine learning, through just-in-time (jit) compilation. It also implements automatic differentiation through `grad`, allowing us to compute the necessary derivatives. Note that `JAX` utilizes 32-bit floating points, which give us a little over 7 digits of precision. This limits our possible lowest loss to around 10^{-7}, rather than the 10^{-16} expected with 64-bit numbers.

D. Weight initialization

We utilize a variant of Xavier initialization (Glorot and Bengio, 2010), in order to initialize the weights of the neural networks. For $W \in \mathbb{R}^{N_i \times N_{i-1}}$ as in (1), we draw

$$W_{ij} \sim N\left(0, \frac{2}{N_i + N_{i-1}}\right), \tag{15}$$

in order to alleviate certain issues. If the weights are too large, errors propagate quickly through out the network, while if they are too small we run the risk of vanishing gradients. As these issues stem from the relation between the number of neurons in a layer, and their weights, it intuitively makes sense to initialize the weights according to this relation.

It is designed to work well with tanh (Katanforoosh and Kunin, 2019), which is our chosen activation function for our implementation. It is defined by

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{16}$$

IV. COMPUTATIONAL EXPERIMENTS

A. Advection Equation

1. Problem Formulation

We first consider the one-dimensional linear advection equation, with discontinuous residual f. The problem has a simple analytical solution, serving as a sanity check for whether our implementation of the method has any merit. The problem is defined by (17) below.

$$\begin{cases} u_t + \alpha \cdot u_x = 0 & \text{for } (x,t) \in [-1,1] \times (0,1] \\ u(x,0) = u_0(x) = f & \text{for } x \in [-1,1], \end{cases} \tag{17}$$

where $\alpha = 0.5$, and the initial condition f is defined by

$$f(x) = \begin{cases} 1 & \text{if } x \in [-0.2, 0.2] \\ 0 & \text{else.} \end{cases}$$

By the method of characteristics, the analytical solution is

$$u(x,t) = u_0(x - \alpha t) = \begin{cases} 1 & \text{if } x - \alpha t \in [-0.2, 0.2], \\ 0 & \text{else.} \end{cases} \tag{18}$$

Thus, the problem defines a wave travelling rightwards without diffusion with a speed α. The analytical solution also predicts two discontinuities. For our XPINN formulation, we decompose the domain according to the predicted discontinuities as in Figure 5.

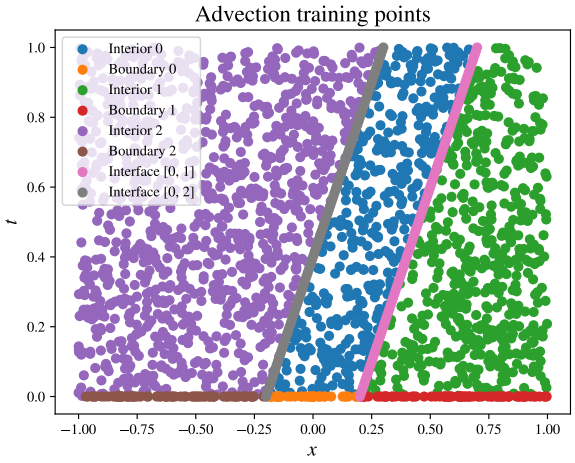

FIG. 5: Training points for the advection equation with the domain decomposed into three subdomains.

In order to verify that our method is not simply learning the decomposition itself, we also verify that the wave is able to travel across the interfaces by setting $\alpha = -1$ while keeping the same decomposition. Finally we compare our XPINN model with a single PINN training on the entire domain without any subdomains.

Our network is composed of 6 hidden layers, each with 20 nodes, using tanh as our hidden activation function and no activation for the output layer. The learning rate is set to 10^{-4}, with Adam as the optimizer. We utilize 2000 points for the interior, and 200 evenly spread around the boundary and interfaces.

2. Results

We run the models for 10 000 epochs each. The resulting predictions are seen in Figure 6, closely fitting the true solution. The absolute errors against the true solution can be seen in Figure 7, where most of the error is found along the interfaces. This is unsurprising, given the discontinuities along the boundaries.

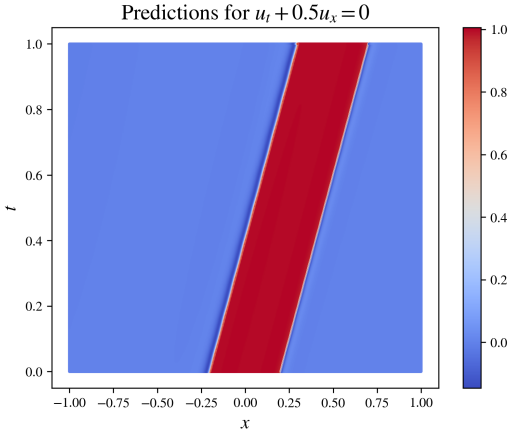

FIG. 6: XPINN Predictions for the advection equation.

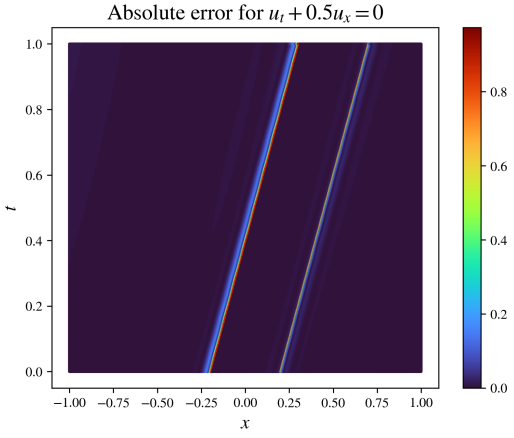

FIG. 7: Absolute error against the true solution, XPINN with $\alpha = 0.5$.

The loss per epoch is shown in Figure 8. It is interesting to note how the different sub-PINNs adjust in accordance with each other, wherein the reduction in the loss of

one sub-PINN often leads to an increase in the others, while the total loss is reduced. Do note that the model seems to still be improving after 10 000 epochs, and that the learning rate needs further tuning. We chose to not optimize this problem fully, as we were more interested in learning more about the behaviour of XPINNs, and seeing if our implementation had any merit.

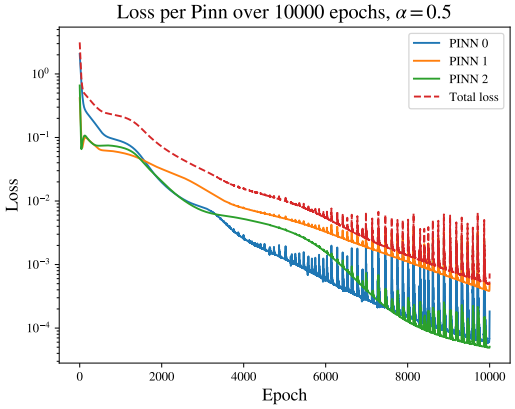

FIG. 8: Loss per epoch for the advection equation with an XPINN, with $\alpha = 0.5$.

Comparing against how a single PINN on the entire domain performs, we see better and more stable results from running the same number of epochs. The predicted values are shown in Figure 9, with the errors in Figure 10. XPINNs then probably do not have a lot of merit for such a simple problem, especially given the simplicity of deriving the analytical solution.

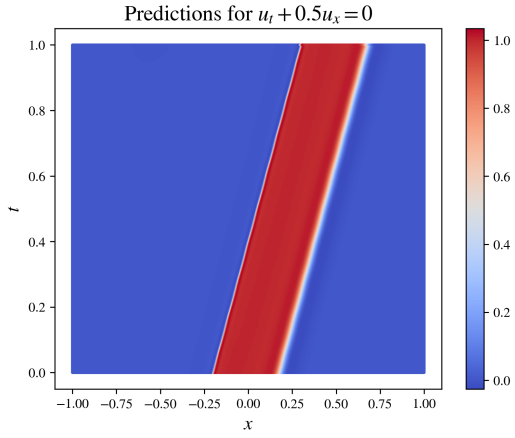

FIG. 9: Predicted values with a PINN.

We stress test our model by changing the equation in question by setting $\alpha = -1$, while keeping the same domain decomposition. This is done in order to ensure that the XPINN is not simply learning the domain decomposition, but rather the behaviour we are expecting. The wave is now able to traverse across the subdomain

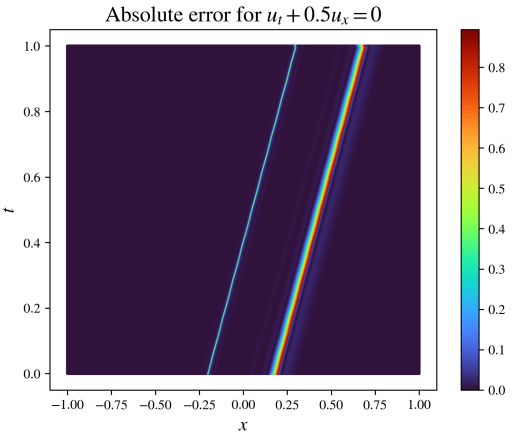

FIG. 10: Absolute errors with a PINN.

decomposition, as in Figure 11, however with worse errors in Figure 12 than we saw previously. This is a promising result before moving on to more difficult problems.

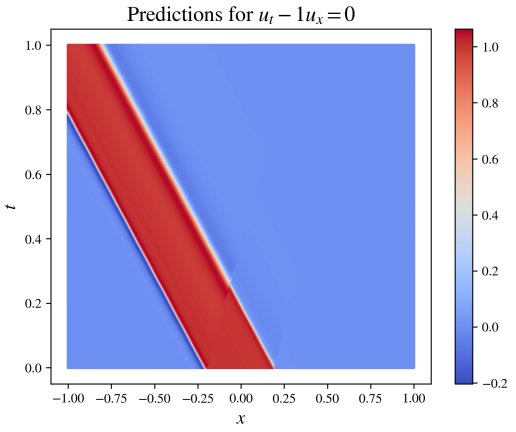

FIG. 11: XPINN predictions with $\alpha = -1$.

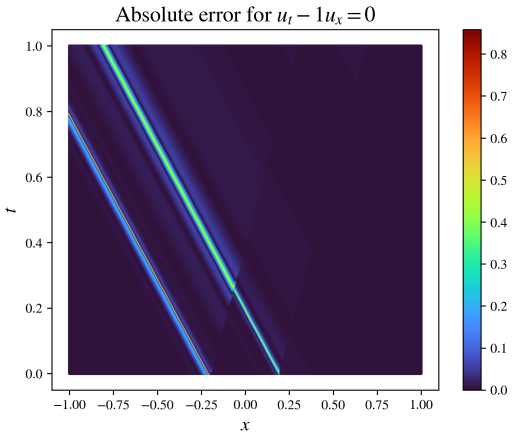

FIG. 12: Absolute error with $\alpha = -1$.

B. Poisson Equation

1. Problem Formulation

In this subsection, we consider a two-dimensional inhomogeneous Poisson equation with residual. The problem is a second order linear PDE, and is defined as

$$\begin{cases} u_{xx} + u_{yy} = f & \text{on } \Omega \\ u(x, y) = 0 & \text{on } \partial\Omega, \end{cases} \tag{19}$$

where $\Omega = [0, 1] \times [0, 1]$. For the Poisson equation, we consider both a continuous and a discontinuous residual.

a. Continuous Residual: The first residual is given by

$$f_{\text{cont}}(x, y) = -2\pi^2 \sin(\pi x) \sin(\pi y), \tag{20}$$

which yields the known analytical solution

$$u_{\text{true}}(x, y) = \sin(\pi x) \sin(\pi y),$$

which is displayed in Figure 13.

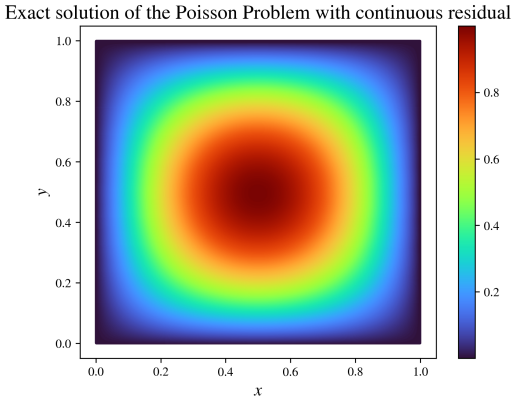

FIG. 13: The exact analytical solution to the Poisson equation with continuous residual.

Our experimental setup follows that of Müller and Zeinhofer (2023). We consider the problem for both a single PINN and for an XPINN decomposition shown in Figure 14. For the PINN, we train using 900 interior points and 120 boundary points distributed uniformly. For the XPINN, we also add 80 uniformly distributed interface points. Thus, the total amount of training points used for the XPINN exceeds that for the PINN. We apply unitary weights to the different losses. All PINNs have the same architecture of a 3-layered neural network with 64 hidden units, and we use the tanh activation function. Both solvers are trained using the ADAM optimizer with an exponentially decreasing learning rate. The initial learning rate is 10^{-3}. After 1.5×10^4 steps, we decrease the learning rate by a factor of 10^{-1} every 10^4 steps until a minimum learning rate of 10^{-7}. We train both cases for 2×10^5 iterations. For our testing, we consider a uniform grid with 1 000 000 points.

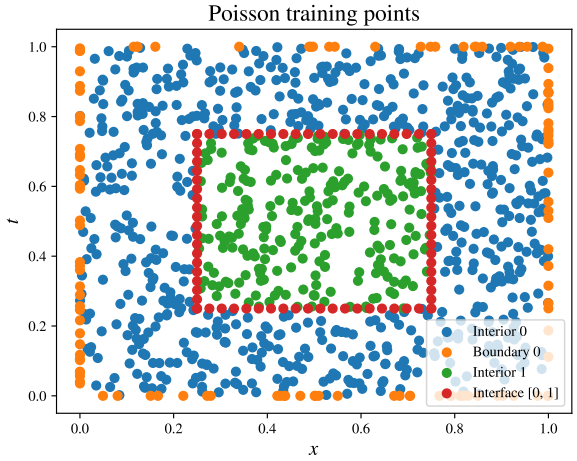

FIG. 14: Training points with the XPINN decomposition for the Poisson equation. The PINN is given the same points, without the interface points.

b. Discontinuous Residual: The second residual is due Hu *et al.* (2022), where they compare the generalizeability of PINNs and XPINNs. The residual is given as a discontinuous form

$$f_{disc}(x, y) = \begin{cases} 1 & \text{if } (x, y) \in [0.25, 0.75] \times [0.25, 0.75], \\ 0 & \text{else.} \end{cases}$$

The reference solution is computed numerically using the `poissonpy` package (Chao, 2022) on a uniform test mesh with 1 000 000 points, displayed in Figure 15. We apply a 9 layer neural network with 20 hidden units, using the tanh activation function. Again, we apply 900 internal, 120 boundary and 80 interface points for training the XPINN model, as in Figure 14. Additionally, we apply 20 weights to the boundary conditions for the PINN, and 20 for boundary and interface conditions for our XPINN. ADAM is again used, and the solvers are run for 2×10^5 iterations. The exponential decay rate is the same as in the continuous experiment, although the minimum learning rate is set as 10^{-5}.

Note that the paper by Hu *et al.* (2022) uses the LBFGS optimizer, however this method is not implemented in our chosen optimization library `optax`. LBFGS is available in `JAXopt` (Blondel *et al.*, 2021). However, `JAXopt` is currently being merged into `optax`. Adapting our implementation to be compatible with `JAXopt` falls outside the scope of this project.

2. Results

a. Continuous Residual: Table I shows the minimum, median, and maximum relative L^2 errors of the single PINN

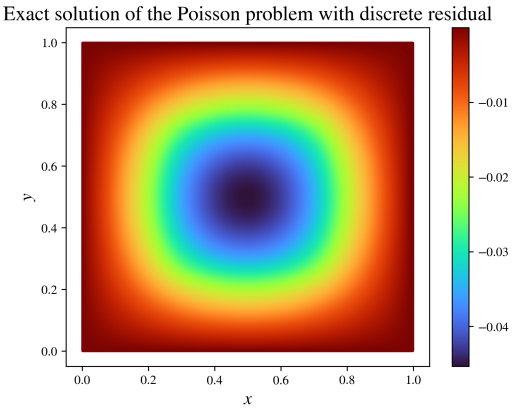

FIG. 15: Numerically computed reference solution to the Poisson equation with discontinuous residual.

and XPINN solvers for the continuous residual Poisson equation using the ADAM optimizer, across ten iterations. Note that the training mesh was re-generated between each iteration.

Figure 16 displays the L^2 error evolution against the iterations. From the figure, we see that the L^2 error seems to stabilize. Although of similar orders of magnitude, we see that the PINN model frequently outperforms our XPINN discretization. This is as expected by the argument of Hu *et al.* (2022), as the domain decomposition does not reduce the target function complexity in the subdomains, whereas each subdomain trains on fewer points than the PINN.

TABLE I: Median, minimum and maximum of the relative L^2-errors for the Poisson equation with continuous residual achieved by the PINN and XPINN.

	Median	Minimum	Maximum
PINN	$1.37 \cdot 10^{-3}$	$8.4 \cdot 10^{-4}$	$2.16 \cdot 10^{-3}$
XPINN	$2.82 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	$5.42 \cdot 10^{-3}$

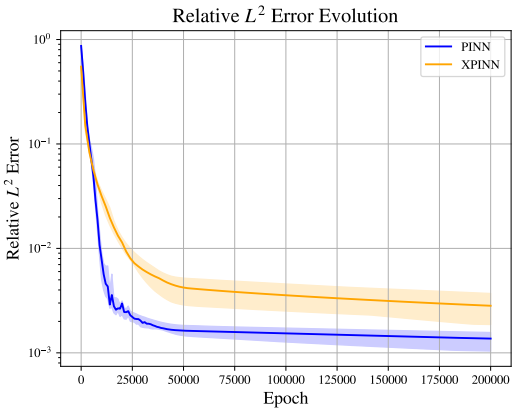

FIG. 16: Relative L^2 error with median and interquartile range highlighted, for continuous residual.

The PINN and XPINN predictions are shown in Figure 17 and Figure 18, and although they are hard to distinguish visually, the errors displayed in Figure 19 and Figure 20 tell a different story. Note that the error figures are created with the same colorbar, in order to be comparable. We note that the significant sources of error for the PINN appear to be surrounding the boundaries. The XPINN solution also shows boundary error, but the interface error seems to dominate. Optimizing for the different weighting hyper-parameters might lower the interface error, potentially at the cost of increased boundary error.

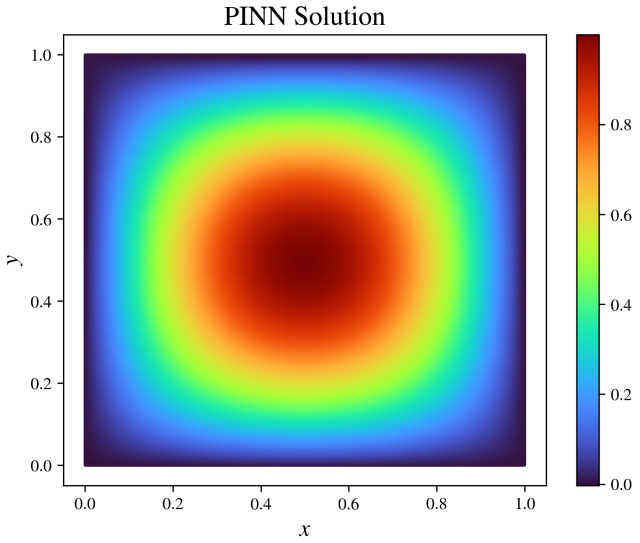

FIG. 17: PINN prediction with continuous residual.

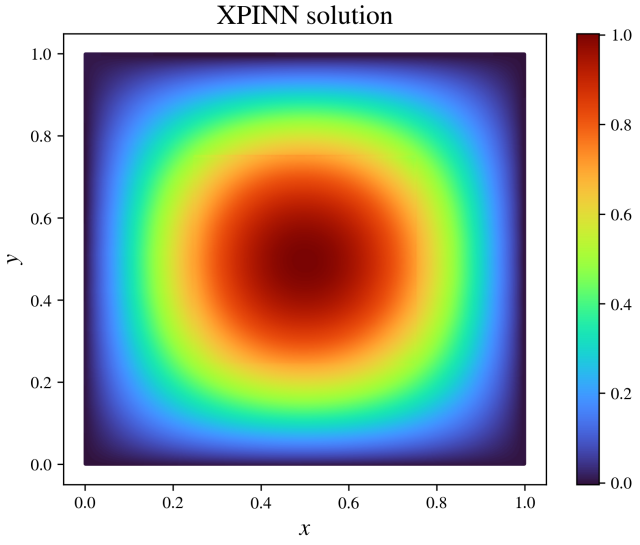

FIG. 18: XPINN prediction with continuous residual.

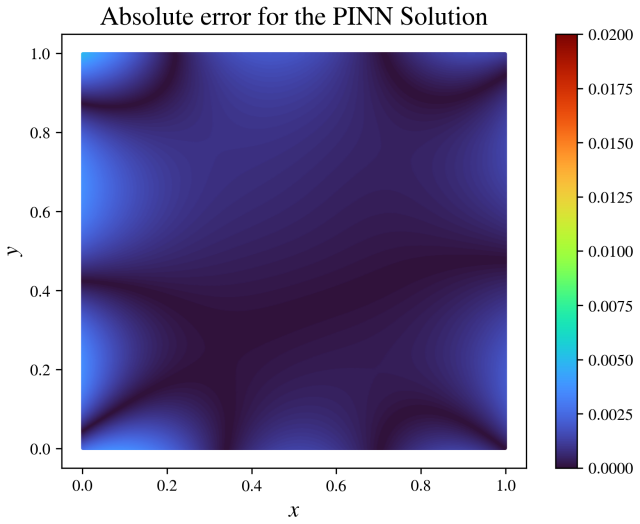

FIG. 19: PINN absolute error with continuous residual.

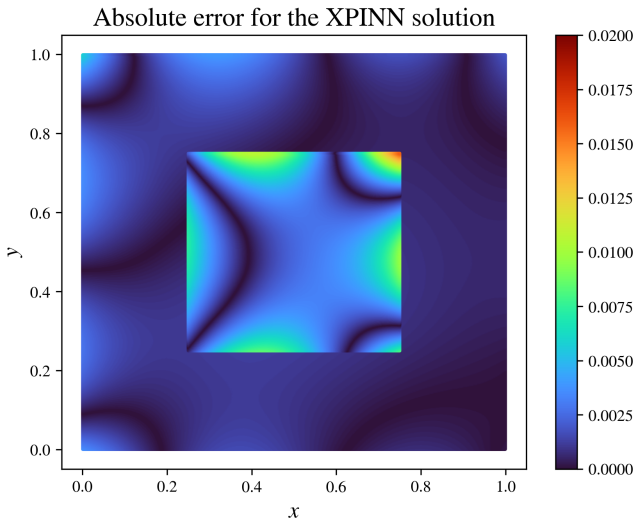

FIG. 20: XPINN absolute error with continuous residual.

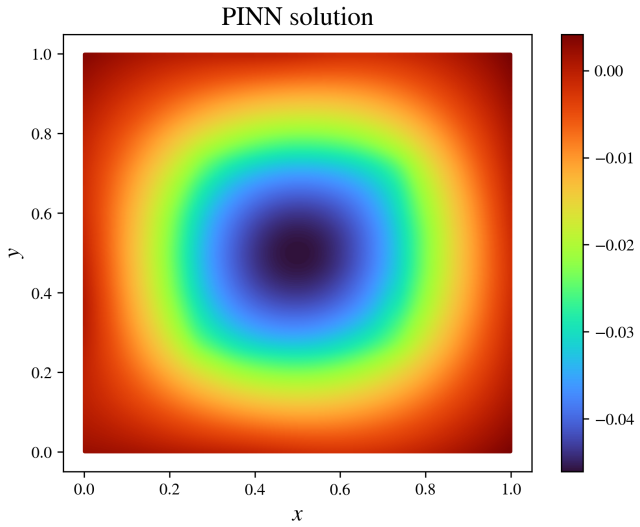

FIG. 21: PINN prediction with discontinuous residual.

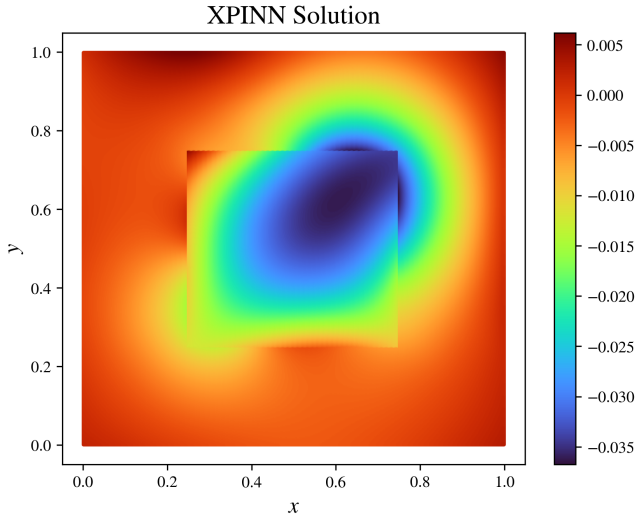

FIG. 22: XPINN prediction with continuous residual.

b. Discontinuous Residual: Figure 21 and Figure 22 display the PINN and XPINN solutions. Compared with the continuous residual, it is now much clearer where the subdomain boundaries are. Interestingly, the outer sub-PINN visually looks similar to the solution of the PINN. The error are shown in Figure 23 and Figure 24, this time with different colorbars as the maximum error of the XPINN is significantly higher than that of the PINN. Both solvers seem to perform worse on this problem. Notably, the XPINN solver appears to perform significantly worse, which is according to expectations (Hu *et al.*, 2022).

Wondering whether this discrepancy is due a lack of training, we investigated the loss per epoch, shown in Figure 25. As we see, the loss only sees marginal improvements after 125 000 iterations. The same is seen with the relative L^2 error in Figure 26, wherein the PINNs stabilize, while the XPINNs remains higher. We therefore believe this is partly due to the reduced number of training points. The points are drawn from a uniform distribution, and as the center subdomain is $[0.25, 0.75] \times [0.25, 0.75]$, it receives on average just 25% of the total points. In addition, given the symmetric nature of the problem, one can argue for whether evenly spaced points might outperform our choice of uniformly random points.

The relative L^2 errors are also summarized in Table II. The XPINNs give a median error about an order of magnitude higher than that of the PINN over 10 runs. These results are consistent with what was found in (Hu *et al.*, 2022).

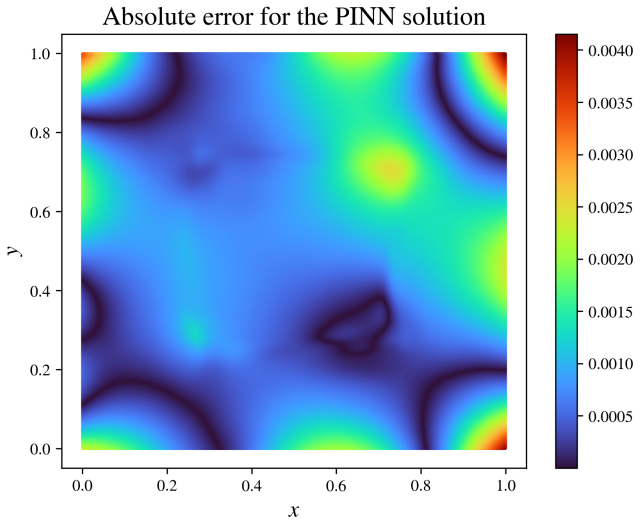

FIG. 23: PINN absolute error with discontinuous residual.

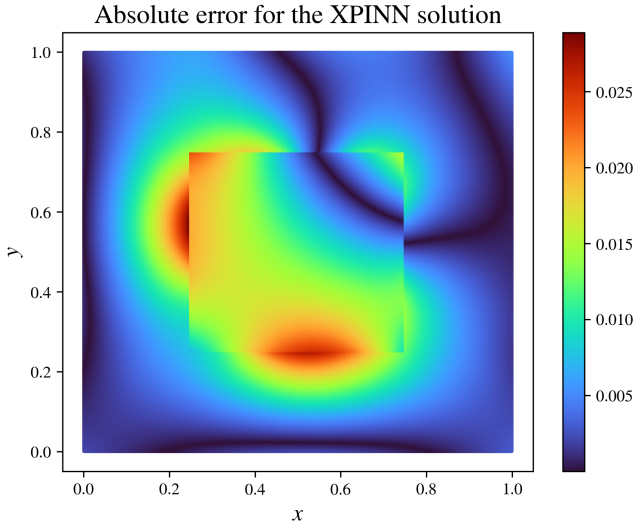

FIG. 24: XPINN absolute error with discontinuous residual.

TABLE II: Median, minimum and maximum of the relative L^2-errors for the Poisson equation with discontinuous residual achieved by the PINN and XPINN.

	Median	Minimum	Maximum
PINN	$3.73 \cdot 10^{-2}$	$3.11 \cdot 10^{-2}$	$6.05 \cdot 10^{-2}$
XPINN	$4.69 \cdot 10^{-1}$	$2.36 \cdot 10^{-1}$	$1.00 \cdot 10^{0}$

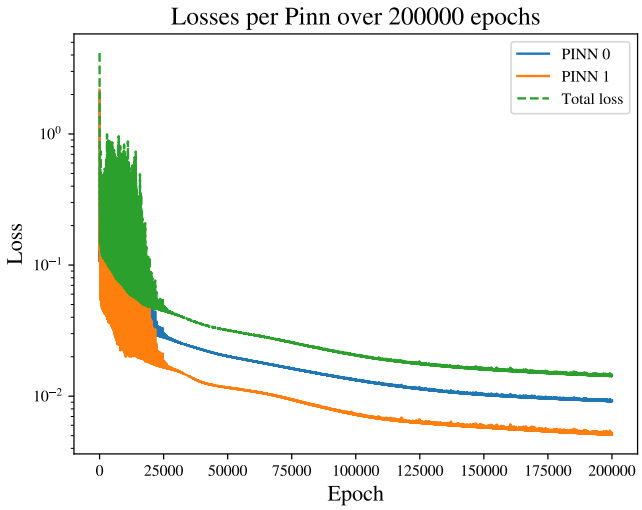

FIG. 25: Loss per epoch for XPINN on the Poisson equation with discontinuous residual.

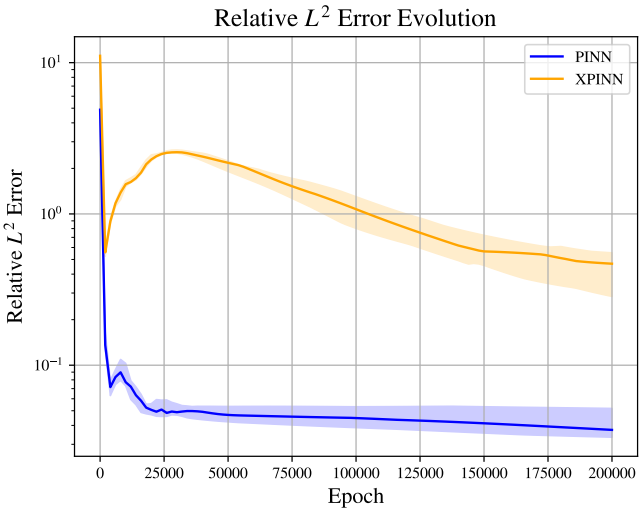

FIG. 26: Relative L^2 error with median and interquartile range highlighted, for PINN and XPINN with discontinuous residual.

C. Navier-Stokes Equation

1. Problem Formulation

Finally, we consider the steady Navier-Stokes equation at Reynolds number $Re = 20$ as described in the FEATFLOW benchmark suite (FEATFLOW, 2011). The problem is a simulation of incompressible 2D fluid flow past a cylinder in a rectangular domain. The steady, incompressible Navier Stokes equation can be written as

$$-\nu\nabla^2\mathbf{u} + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p = \mathbf{0}, \quad \nabla\cdot\mathbf{u} = 0, \qquad (21)$$

where $\nu = 0.001$. In our notation, we will isolate the x- and y-flow as u and v respectively, writing $\mathbf{u} = (u, v)^\intercal$.

From this, we can express the PDEs for the second-order spatial derivatives, as well as the equation for zero convergence due to incompressibility as

$$\begin{cases} -\nu(u_{xx}+u_{yy})+uu_x+vu_y+p_x=0, \\ -\nu(v_{xx}+v_{yy})+uv_x+vv_y+p_y=0, \\ u_x+v_y=0. \end{cases} \tag{22}$$

The domain of the problem is given by

$$\Omega=[0,2.2]\times[0,0.41]\setminus B_r(0.2,0.2), \quad r=0.05.$$

In order to impose the zero divergence condition, we consider the stream function ψ characterizing incompressible flow. u and v can then be obtained through $u=\psi_y$ and $v=-\psi_x$. Our network $\mathcal{N}_\theta$ thus predicts the stream function ψ and the pressure p, with

$$\mathcal{N}_\theta:\mathbb{R}^2\to\mathbb{R}^2, \quad (x,y)\to(\psi,p). \tag{23}$$

This problem includes three different boundary conditions. First, we have the boundary for the upper and lower walls, as well as around the edge of the cylinder. In the DFG benchmark 2D-1 (FEATFLOW, 2011), the lower and upper walls are notated as $\Gamma_1=[0,2.2]\times 0$ and $\Gamma_3=[0,2.2]\times 0.41$ respectively, and the cylinder as $S=\partial B_r(0.2,0.2)$. On these walls, we enforce impermeability and the no-slip conditions, leading to zero-flow on these boundaries. The no-slip boundary condition is defined as

$$\begin{aligned} u_{|\Gamma_1}=u_{|\Gamma_3}=u_{|S}=0, \\ v_{|\Gamma_1}=v_{|\Gamma_3}=v_{|S}=0. \end{aligned} \tag{24}$$

On the left boundary we enforce polynomial inflow. The left boundary is $\Gamma_2=0\times[0,0.41]$ and the polynomial inflow is given by

$$u=\frac{4Uy(0.41-y)}{0.41^2}, \quad v=0, \tag{25}$$

where $U=0.3$, and the mean inlet velocity is $U_{\text{mean}}=0.2$. Finally, on the right boundary we have the do-nothing boundary condition. The boundary is $\Gamma_4=2.2\times[0,0.41]$, and the condition is given by

$$\nu u_x-p=0, \quad \nu u_y=0. \tag{26}$$

Hence, we confirm our target Reynolds number using the cylinder diameter as the characteristic length $L=0.1$. U_{mean} is used as the characteristic velocity. The Reynolds number then becomes

$$Re=\frac{U_{\text{mean}}L}{\nu}=\frac{0.2\cdot 0.1}{0.001}=20. \tag{27}$$

2. Results

Achieving qualitative results required a fair deal of hyperparameter tuning, especially the weighting of the losses in (6), as well as the model architecture. This search was not carried out exhaustively, but rather through a heuristic approach of trial and error. Given this, it should be recognized that there may still be potential to achieve better results.

The PINN was structured with 25 hidden layers, each consisting of 30 nodes, with the tanh activation function and unitary weighting of the loss function. We utilized exponential decay in order to tune the learning rate of Adam, leading to a rapid, chaotic, decrease in the measured loss in the early epochs, stabilizing, as in Figure 27. The loss did not fully plateau, meaning there could be room for improvement, although that might also be indicative of overfitting.

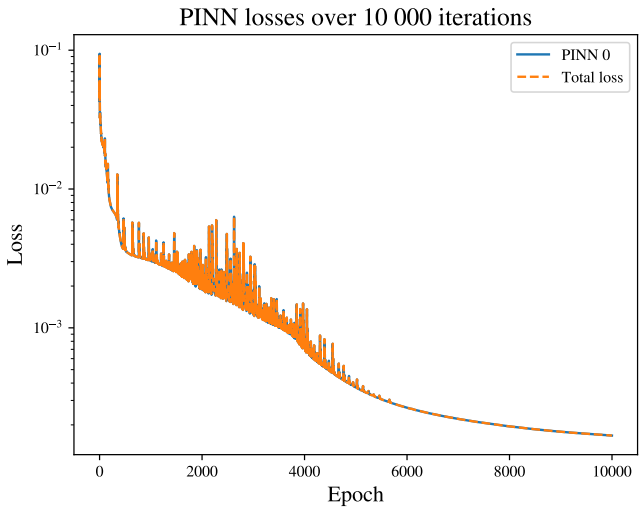

FIG. 27: Loss per epoch for PINN on the Navier-Stokes equation.

The domain decomposition for the XPINN consisted of

$$\Omega_2=[0.8,2.2]\times[0,0.41] \quad \text{and} \quad \Omega_1=\Omega\setminus\Omega_2. \tag{28}$$

The XPINN was structured with 25 hidden layers nodes for Ω_1 and 10 hidden layers for Ω_2, all with 30 nodes each. As the XPINN encountered difficulties on the interface, we imposed Neumann interface condition on ψ in order to increase performance. This involved including a continuity loss on the flow field, with

$$\frac{1}{N_I}\sum_{i=0}^{N_I}\left(\nabla\psi(x_I^i)-\{\nabla\psi_I^i\}\right)^2, \tag{29}$$

where $\{\nabla\psi_I^i\}$ is the average gradient predicted at the interface, allegoric to (9).

The loss for the XPINN seems to stabilize earlier in Figure 28, although do note that the loss is not directly comparable with Figure 27, as in addition to the Neumann interface condition, we applied a heavy weighting of 20 on the inflow boundary, and a weighting of 40 on

the left-to-right interface. Even with this weighting, the interface is still clearly visible in Figure 32. To further enhance performance at the interface, we could explore additional adjustments, such as introducing more continuity conditions, e.g. Robin continuity.

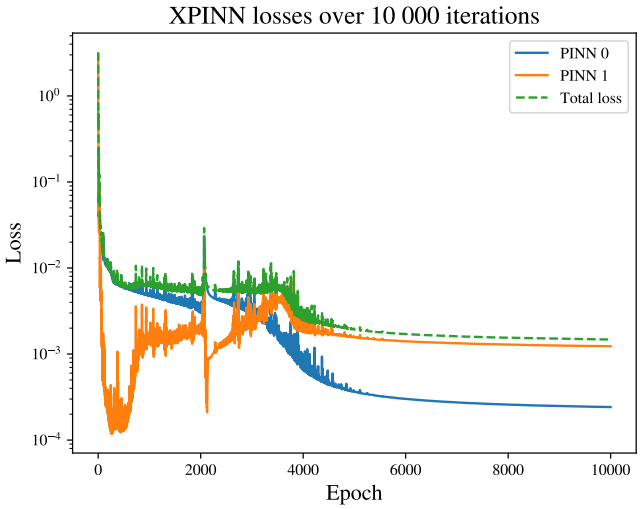

FIG. 28: Loss per epoch for XPINN on the Navier-Stokes equation.

In general, the PINN produced better results than the XPINN. However, with the reduced complexity of the XPINN, we were able to increase the computational speed significantly. Having said that, both our models successfully replicated the flow plots from the DFG benchmark (FEATFLOW, 2011), as illustrated in Figures 29 through 34. It should be noted on comparison with the benchmark that slightly different colorings were used.

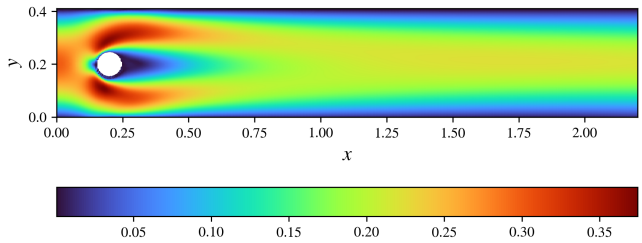

FIG. 29: Velocity magnitude of the PINN.

Along with the flow figures, the benchmark also provides drag and lift coefficients, as well as the pressure difference in front of and behind the cylinder. Our approximation of these results are seen in Table III.

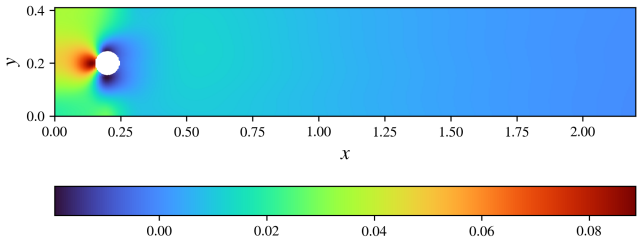

FIG. 30: Pressure prediction of the PINN.

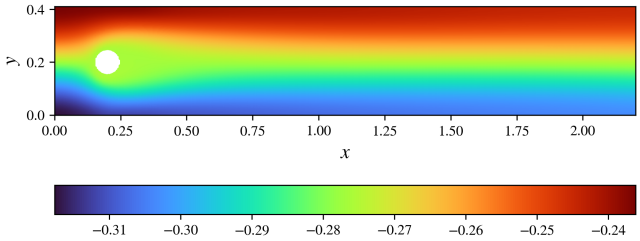

FIG. 31: Predicted streamfunction of the PINN.

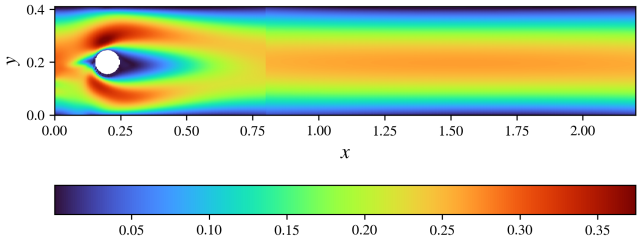

FIG. 32: Velocity magnitude of the XPINN.

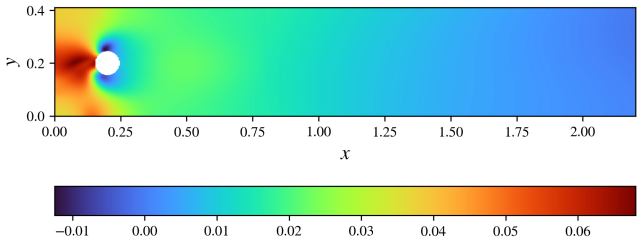

FIG. 33: Predicted pressure of the XPINN.

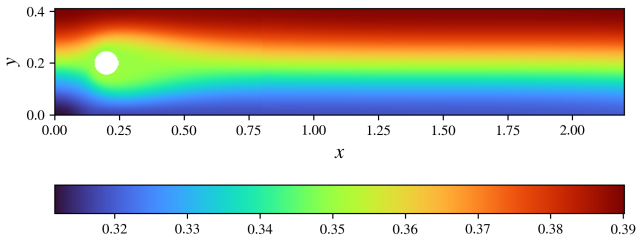

FIG. 34: Predicted streamfunction of the XPINN.

TABLE III: Comparison with reference results.

	Drag	Lift	Pressure difference
DFG	5.5795	0.0106	0.1175
PINN	4.0687	-0.2542	0.0891
XPINN	2.7482	-0.3368	0.0557

V. CONCLUSION

From the results on the advection equation we demonstrated that XPINNs can capture the wave's propagation, including handling the predicted discontinuities, which suggest they can reliably find solutions close to the analytical benchmarks. They are still slightly outperformed by a less computationally costly single PINN, however our aim of this demonstration was to show that they give comparable results. XPINNs are evidently superfluous for simple problems, but have adequate performance to be considered for more complex problems.

In the case of the Poisson equation, our findings indicate that while XPINNs can manage both continuous and discontinuous residuals. As expected the XPINN underperforms for solving the Poisson equation. We also get quantitatively worse results than Hu *et al.* (2022), however this can likely be contributed to the fact that we use ADAM as our optimizer while they utilize LBFGS, which is a quasi-Newton method expected to perform better. Our results are qualitatively not too far off from the analytical solution, however they are again outperformed by a singular PINN.

One of the primary benefits of XPINNs is the inherent capacity for parallelization. However, we were not able to implement this, meaning we were not able to get a computational performance boost while training the XPINNs, other than the reduced complexity utilized with Navier-Stokes. In addition, introducing interface losses might complicate the loss landscape entirely, such that more powerful optimization algorithms are necessary. These methods include Energy Natural Gradient Descent (Müller and Zeinhofer, 2023), which yielded orders of magnitude lower errors, and learning rate free algorithms (Sharrock *et al.*, 2023), which alleviate the need to tune the learning rate as a parameter while performing remarkably well.

This leads us to another challenge we ran into with the XPINN, wherein we introduce a number of new hyperparameters through the MSE in (6), which needs to be balanced against the interface losses. This issue is unavoidable with XPINNs, leading to a more complex tuning stage. Hu *et al.* (2023) attempts to alleviate this through the introduction of Augmented PINNs (APINNs), which present a natural evolution of XPINNs. Here, the domain is decomposed through the use of gating functions, rather than hard boundaries. Thus, interface losses are eliminated entirely, in addition to each sub-PINN gaining access to all of the available training data. We believe further work should be focused on exploring and utilizing APINNs, rather than optimizing XPINNs further.

Another avenue for exploration is within the realm of how the weights of the network are set. Recently, 1.58bit models (Ma *et al.*, 2024) have been developed, where in each weight W_{ij} of the networks in constrained to be in $\{-1, 0, 1\}$, rather than the 32-bit or 64-bit floating point numbers used today. This greatly reduce the physical size of networks, while seemingly performing similarly. It also opens an avenue for a reduction in the energy usage, as with a new computational architecture, the costly multiplication operations can be eliminated entirely. In an era where cutting edge models are trained in huge data centers, this represents a vital development within sustainable computing.

Machine learning seems to be increasingly developing towards a direction where models are hand-tuned based on the problem at hand, rather than being brute force approximators. XPINNs serve as one such specialization focused on the resolution of PDEs, representing an ever complex machine learning landscape.

REFERENCES

E. H. W. Ang, G. Wang, and B. F. Ng (2023), "Physics-informed neural networks for low reynolds number flows over cylinder," Energies **16** (12), 10.3390/en16124558.

M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert (2021), "Efficient and modular implicit differentiation," arXiv preprint arXiv:2105.15183.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang (2018), "JAX: composable transformations of Python+NumPy programs," .

B. Chao (2022), "poissonpy," .

G. V. Cybenko (1989), "Approximation by superpositions of a sigmoidal function," Mathematics of Control, Signals and Systems **2**, 303–314.

DeepMind, I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, A. Dedieu, C. Fantacci, J. Godwin, C. Jones, R. Hemsley, T. Hennigan, M. Hessel, S. Hou, S. Kapturowski, T. Keck, I. Kemaev, M. King, M. Kunesch, L. Martens, H. Merzic, V. Mikulik, T. Norman, G. Papamakarios, J. Quan, R. Ring, F. Ruiz, A. Sanchez, L. Sartran, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, M. Stanojević, W. Stokowiec, L. Wang, G. Zhou, and F. Viola (2020), "The DeepMind JAX Ecosystem," .

FEATFLOW (2011), *DFG flow around cylinder benchmark 2D-1, laminar case Re=20*, Lehrstuhl III, Angewandte Mathematik und Numerik, Technische Universität Dortmund, Vogelpothsweg 87, Dortmund, available at `https://wwwold.mathematik.tu-dortmund.de/~featflow/en/benchmarks/cfdbenchmarking/flow/dfg_benchmark1_re20.html`, accessed 31. march 2024.

J. Garay, J. Dunstan, S. Uribe, and F. S. Costabal (2023), "Physics-informed neural networks for blood flow inverse problems," arXiv:2308.00927 [cs.CE].

X. Glorot, and Y. Bengio (2010), "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 9, edited by Y. W. Teh, and M. Titterington (PMLR, Chia Laguna Resort, Sardinia, Italy) pp. 249–256.

Z. Hu, A. D. Jagtap, G. E. Karniadakis, and K. Kawaguchi (2022), "When do extended physics-informed neural networks (xpinns) improve generalization?" SIAM Journal on Scientific Computing **44** (5), A3158–A3182, https://doi.org/10.1137/21M1447039.

Z. Hu, A. D. Jagtap, G. E. Karniadakis, and K. Kawaguchi (2023), "Augmented physics-informed neural networks (apinns): A gating network-based soft domain decomposition methodology," Engineering Applications of Artificial Intelligence **126**, 107183.

A. D. Jagtap, and G. E. Karniadakis (2020), "Extended Physics-informed Neural Networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations," Communications in Computational Physics .

A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis (2020), "Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems," Computer Methods in Applied Mechanics and Engineering **365**, 113028.

A. D. Jagtap, Z. Mao, N. Adams, and G. E. Karniadakis (2022), "Physics-informed neural networks for inverse problems in supersonic flows," Journal of Computational Physics **466**, 111402.

K. Katanforoosh, and D. Kunin (2019), "Initializing neural networks," Accessed on March 24, 2024.

D. P. Kingma, and J. Ba (2017), "Adam: A method for stochastic optimization," arXiv:1412.6980 [cs.LG].

S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei (2024), "The era of 1-bit llms: All large language models are in 1.58 bits," arXiv:2402.17764 [cs.CL].

R. Mehta (2021), "Automatic differentiation in neural nets," Accessed on March 22, 2024.

J. Müller, and M. Zeinhofer (2023), "Achieving high accuracy with pinns via energy natural gradient descent,".

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

S. J. Reddi, S. Kale, and S. Kumar (2019), "On the convergence of Adam and beyond," arXiv:1904.09237 [cs.LG].

D. E. Rumelhart, G. E. Hinton, and R. J. Williams (1986), "Learning representations by back-propagating errors," Nature **323**, 533–536.

L. Sharrock, L. Mackey, and C. Nemeth (2023), "Learning rate free sampling in constrained domains," arXiv:2305.14943 [stat.ML].

Appendix A: Source code

The code is available at `https://github.com/augustfe/FYS5429`, along with a number of figures left out of this paper.