Field Solutions of Parametric PDEs

Biswajit Khara[1], **Aditya Balu**[1], **Ameya Joshi**[2],
Adarsh Krishnamurthy [2], **Soumik Sarkar**[1], **Chinmay Hegde**[2],
Baskar Ganapathysubramanian[1]
[1] Department of Mechanical Engineering, Iowa State University
[2] Department of Computer Science and Engineering, New York University
bkhara@iastate.edu, baditya@iastate.edu, ameya.joshi@nyu.edu, adarsh@iastate.edu, soumiks@iastate.edu,
chinmay.h@nyu.edu, baskarg@iastate.edu

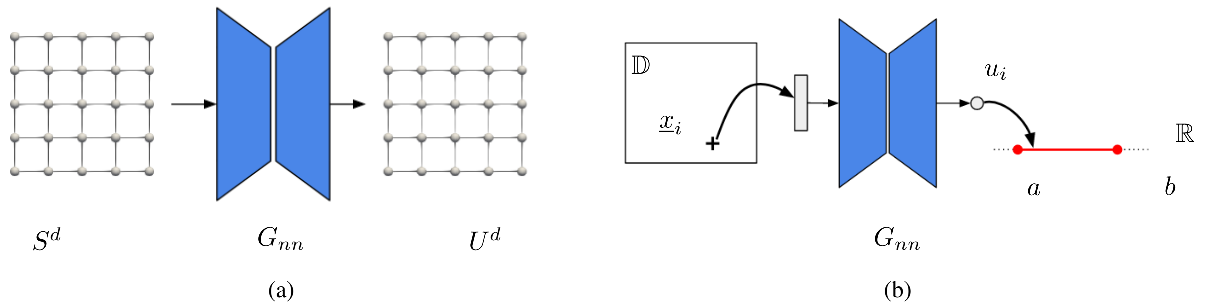

Figure 1: (a) In our DIFFNET approach, we train a neural network to produce a discretized field solution over a mesh. Such an approach offers a direct link to powerful PDE analysis techniques, at the cost of a larger network. (b) In contrast, neural methods (like (Lagaris, Likas, and Fotiadis 1998; Raissi, Perdikaris, and Karniadakis 2019)) are trained to produce point predictions: $G_{nn} : \mathbb{D} \rightarrow \mathbb{R}$, which are easier to train, but more difficult to analyze.

Abstract

We consider mesh based approaches for training neural network to produce field predictions to parametric partial differential equations (PDEs). This is in contrast to current approaches for 'neural PDE solvers' that employ collocation approaches to make point predictions of PDEs. Our approach has advantages of (a) easier handling of various boundary conditions, and (b) ease of invoking well developed PDE theory – including analysis of numerical stability and convergence – on discretized domains. On the other hand, an obvious disadvantage is the network size required for producing field solutions. We explore such a strategy using two loss functions based on (i) Finite Difference Method (FDM) and (ii) Finite Element Method (FEM) on two canonical parametric PDEs. While the FDM loss is closely related to losses used in recent PINN type approaches, the weighted galerkin loss (FEM loss) is akin to an energy functional that produces improved solutions, satisfies *a priori* mesh convergence, and can model Neumann boudary conditions. These results suggest that mesh based neural networks are promising approaches for parametric PDEs.

1 Introduction

Numerical methods – finite difference methods (FDM), finite element methods (FEM), spectral methods – for solving PDEs discretize the physical domain (into cells, elements, ect) and *approximate* the solution over this discretized domain using select families of basis functions (Hughes 2012; LeVeque 2007; Trefethen 2000). There is well developed and elegant theory that connects the discretization of the domain (in terms of element/cell dimension, h) and the properties of the basis functions (in terms of polynomial order, α) with the quality of the ensuing numerical solution to the PDE. In particular, numerical stability arguments and *a priori* error estimates allow users to judiciously reason about accuracy, robustness, and convergence (Brenner and Scott 2007; Larson and Bengzon 2013). Such theoretical arguments rely on the spatial discretization of the domain, and properties of the basis functions. Here, we formulate loss functions and networks that produce field solutions of PDEs (Fig. (1)(a)) – to be analogous with numerical PDE solvers – in contrast to current neural PDE solvers that predominantly produce point estimate (Fig (1)(b)).

In recent years data-driven (Rudy et al. 2019; Tompson et al. 2017) and data-free approaches (Raissi, Perdikaris, and Karniadakis 2019; Kharazmi, Zhang, and Karniadakis 2019; Sirignano and Spiliopoulos 2018; Yang, Zhang, and Karniadakis 2018; Pang, Lu, and Karniadakis 2019; Karumuri et al. 2020; Han, Jentzen, and Weinan 2018; Michoski et al. 2019; Samaniego et al. 2020) for solving PDEs have been proposed. Most of these models are designed for pointwise prediction, i.e., the networks in these cases simply take as input $\underline{x}$ and produces an output value of u, thereby calculating the value of u at one particular point. Some of these methods satisfy/apply the boundary conditions exactly (Lee and Kang

1990; Lagaris, Likas, and Fotiadis 1998; Malek and Beidokhti 2006) and others do that in an approximate manner (Lagaris, Likas, and Papageorgiou 2000; Raissi, Perdikaris, and Karniadakis 2019; Sirignano and Spiliopoulos 2018). Many of these methods do not require a mesh, and thus rely on collocating points from the domain randomly. The methods that approximately satisfies the boundary conditions do so by adding a loss function with respect to the specified boundary conditions. It has been shown by (van der Meer, Oosterlee, and Borovykh 2020) that these losses have to be carefully weighted, making this an non-trivial exercise in hyper parameter tuning. This underlines the difficulty of applying the boundary conditions in a neural network-based method (or simply neural method).

Contributions: We build upon some recent efforts that train networks to predict the full-field solution (Paganini, de Oliveira, and Nachman 2018; Botelho et al. 2020; Zhu et al. 2019). The specific contributions of this paper are :

1. We present an algorithm that is bridges traditional numerical methods with neural methods. The neural network is designed to map inputs to the discretized field solution u. However, the neural network is *not responsible* for ensuring the spatial differentiability of the solution. Rather, the discrete field solution relies on traditional numerical methods (and associated numerical differentiation and quadrature) for construction of the loss function. Such an approach allows natural incorporation of different boundary conditions, and allows *a priori* error estimates.
2. We define two loss functions based on FDM and FEM. By setting up the loss function this way, we create the function space with appropriate differentiability and also account for the "local" nature of the solution.
3. We demonstrate DIFFNET's performance on linear Poisson equation in 2D and steady heat transfer.

2 Formulations

Consider a bounded open (spatial) domain $D \in \mathbb{R}^n, n \geq 2$ with a Lipschitz continuous boundary $\Gamma = \partial D$. We will denote the domain variable as $\underline{x}$, where the underbar denotes a vector or tuple of real numbers. In $\mathbb{R}^n$, we have $\underline{x} = (x_1, x_2, \ldots, x_n)$; but for 2D and 3D domains, we will use the more common notation $\underline{x} = (x, y)$ and $\underline{x} = (x, y, z)$ respectively.

On this domain D, we consider an abstract PDE on the function $u : \Omega \to \mathbb{R}$ as:

$$\mathcal{N}[u; s(\underline{x}, \omega)] = f(\underline{x}), \quad \underline{x} \in D \tag{1a}$$

$$\mathcal{B}(u, \underline{x}) = g(\underline{x}), \quad \underline{x} \in \Gamma \tag{1b}$$

where $\mathcal{N}$ is a differential operator (possibly nonlinear) operating on a function u. The differential equation also depends on the data of the problem s which in turn is a function of the domain variable $\underline{x}$ and parameter ω. Thus $\mathcal{N}$ is essentially a family of PDE's parameterized by ω.

2.1 Neural approximation of the solution

Instead of seeking a mapping between the domain and an interval on the real line (Fig. (1)(b)), we seek a mapping between the input s and the full field solution u in the discrete spaces (Fig. (1)(a)). S^d denotes the discrete representation of the known quantity s. S^d could be either available only at discrete points (perhaps from some experimental data); or in many cases, s might be known in a functional form and thus S^d will be simply the values of s evaluated on the discrete points. Therefore, if we denote a DIFFNET (see Figure (1)(a)) network by G_{nn}, then G_{nn} takes as input a discrete or functional representation of s and outputs a discrete solution field U_θ^d, where θ denote the network parameters. For example, if we consider a PDE defined on a 2D bounded domain, then G_{nn} simply takes a 2D matrix containing the values of s and gives out the solution U_θ^d which is also a 2D matrix.

An untrained network, as expected, will produce a mapping that does not satisfy the discrete PDE; in fact, it will show a huge error. Our goal is to bring this error down to an acceptable level, and thereby reaching a solution that is "close enough" to the exact solution. And we do this by designing the loss function based on major ideas taken from the classical numerical methods. This is explained next.

2.2 Loss functions inspired by numerical methods

The design of the loss function, along with the choice of the neural mapping, forms the "heart" of our approach. As referenced before, the loss functions are based on either FDM and FEM discretization. We discuss both of these separately below.

FDM based loss function Suppose the set $\underline{X} = (\underline{x}_1, \underline{x}_2, .., \underline{x}_N) \in \mathbb{R}^{n \times N}$ denote a collection of points in $\mathbb{R}^n$ that produces a (uniform) discretization of D. Define $S_i = s(\underline{x}_i)$ and U_i an approximation of the unknown $u(\underline{x}_i)$. And as usual, we also define the vectors $\underline{S} = (S_1, S_2, .., S_N)$ and $\underline{U} = (U_1, U_2, .., U_N)$. On this regularly spaced stencil, the numerical derivatives can be approximately written in terms of finite difference formulas.

$$D^r u(x_i) \approx \sum_{j=-k_1}^{k_2} c_j(h) U_j \tag{2}$$

where D is a differential operator, r is the order or derivative, c's are known values (depends on the grid spacing h) and U_i as explained above. k_1 and k_2 are some integers that follows the choice of the stencil. Using the expressions of these finite difference formulas, we can then write the discrete PDE at each of the points in $\underline{X}$. By writing N such equations, we obtain an N-dimensional linear system. For a linear PDE, this is easily written as:

$$a_{i1}U_1 + a_{i2}U_2 + \ldots + a_{iN}U_N = f_i, \quad i = 1, 2, ..., N^{int} \tag{3a}$$

$$b_{i1}U_1 + b_{i2}U_2 + \ldots + b_{iN}U_N = g_i, \quad i = 1, 2, ..., N^{bc} \tag{3b}$$

Combining the interior and boundary equations, we can write down the matrix form,

$$\underline{\underline{A}}\,\underline{U} = \underline{f} \tag{4}$$

The residual vector $\underline{R}_{FDM}$ can then be defined as:

$$\underline{R}_{FDM} = \underline{\underline{A}}\,\underline{U} - \underline{f} \tag{5}$$

The loss is then defined as the 2-norm of the residual vector,

$$L_{FDM} = \|\underline{R}_{FDM}\|_2 \tag{6}$$

FEM based loss function The FEM loss involves the weakening of the PDE using an appropriate weighting functions. As in the FDM case, we conside a discretization of the domain D into a finite number of non-overlapping elements denoted by $Q_i, i = 1, 2, \ldots, n_{el}$ such that $\cup_i^{nel} Q_i = D$. The unknown solution can be approximated as:

$$u_\theta^h = \sum_{i=1}^{N} \phi_i(\underline{x})(U_i)_\theta \tag{7}$$

where ϕ_i are the finite element basis functions.

This approximation is plugged into the PDE, after which we invoke Galerkin's method. That is, multiply the PDE with a test function and reduce the differentiability requirement on u^h via integration by parts [1]:

$$\int_\Omega v \left[\mathcal{N}(u_\theta^h; s) - f\right] d\underline{x} = 0 \;\forall v \in V \tag{8}$$

Which results in this following (standard FEM) form

$$B(v, u_\theta^h) - L(v) = 0 \;\forall v \in V \tag{9}$$

where $B(v, u_\theta^h)$ is the bilinear form that encodes the PDE, while $L(v)$ is the linear form that encodes the load and boundary conditions. By choosing the test function to be the (unknown) solution, u_θ^h, we get an energy functional whose minima is the solution

$$J(u_\theta^h) = \frac{1}{2} B(u_\theta^h, u_\theta^h) - L(u_\theta^h) \tag{10}$$

This energy functional accounts for the PDE as well as all Neumann (and Robin) conditions. This serves as our loss function.

2.3 Applying boundary conditions

In DiffNet, the Dirichlet boundary conditions are applied exactly. The query result U_θ^d from the network pertains only to the interior of the domain. The boundary conditions need to be taken into account separately. There are two ways of doing this:

- Applying the boundary conditions exactly (this is possible only for Dirichlet conditions in FEM/FDM; and the zero-Neumann case in FEM)
- Taking the boundary conditions into account in the loss function, thereby applying them approximately.

[1]For completeness, we assume $u_\theta^h \in V \subset H^1(D)$ where $H^1(D)$ denotes the Hilbert space of functions on D that have square integrable first derivatives,

Algorithm 1 Algorithm for an instance of a PDE

```
Require: $S^d$, $(U_\theta^d)_{bc}$, $\alpha$ and TOL
1: Initialize $G_{nn}$
2: for epoch ← 1 to max_epoch do
3:     $(U_\theta^d)_{int} \leftarrow G_{nn}(S^d)$        ▷ "int" stands for interior nodes
4:     $(U_\theta^d) \leftarrow (U_\theta^d)_{int}\chi_{int} + (U_\theta^d)_{bc}\chi_b$
5:     $loss = L(U_\theta^d)$
6:     $\theta \leftarrow optimizer(\theta, \alpha, \underline{\nabla}_\theta(loss))$
7: end for
```

We take the first approach of applying the Dirichlet conditions exactly (subject to the mesh). Since the network architecture is well suited for 2d and 3d matrices (which serve as adequate representation of discrete field in 2d/3d on regular geometry), the imposition of Dirichlet boundary conditions amounts to simply padding the matrix by the appropriate values. A zero-Neumann condition can be imposed by taking the "edge values" of the interior and copying them as the padding. A nonzero Neumann condition is slightly more involved in the FDM case since additional equations need to be constructed; but if using FEM loss, this can be done with another surface integration on the relevant boundary.

2.4 Calculation of derivatives and integration

The derivatives in FDM are calculated by in-built convolution operations. DiffNet takes an image representation of the input field and returns another image for the solution. Thus, the vector U_θ^d is represented as a matrix (or image) in the code. There is an advantage of using this representation, which is that the derivative calculations can be made very fast using convolution operations. The stencil size determines the size of the kernel, and the FDM difference coefficients form the elements of the kernel. This convolution is only performed within the loss function, and thus has no relation to the convolutional neural network that is used to approximate the solution. See Figure (2) and (3) for example.

The same goes for the integration process in FEM. The full domain integration (i.e., $\int D$) is nothing but the simple sum of the integration over the individual elements (i.e., $\sum_{i=1}^{N_{el}} \int D_i$). This integration over an individual element is in turn the simple weighted sum of the integrand evaluated at the Gauss quadrature points. This evaluation at a single Gauss point can be represented as convolution. Thus, if there are 4 Gauss points in each element, then 4 convolution operations will evaluate the integrand at those points for each element. After that, we only need to sum across Gauss points first, then followed by a sum across elements. See Figure (4) and (5) for example.

2.5 Model architecture for DiffNet

Due to the structured grid representation of $\mathcal{S}^d$ and similar structured representation of U_θ^d, deep convolutional neural networks are a natural choice of network architecture. The spatial localization of convolutional neural networks helps

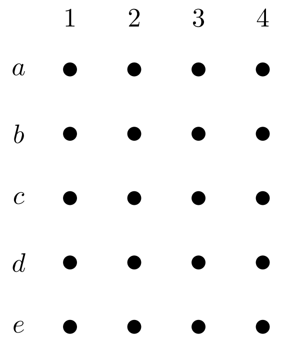

Figure 2: A 5×4 grid for applying finite difference method.

Algorithm 2 Algorithm for parametric PDE

```
Require: $S^d$, $(U^d_\theta)_{bc}$, $\alpha$ and TOL
 1: Initialize $G_{nn}$
 2: for epoch $\leftarrow$ 1 to max_epoch do
 3:     for mb $\leftarrow$ 1 to max_mini_batches do
 4:         Sample $S^d_{mb}$ from the set
 5:
 6:         $(U^d_\theta)_{int,mb} \leftarrow G_{nn}(S^d_{mb})$
 7:                                    ▷ "int" stands for interior nodes
 8:         $(U^d_\theta)_{mb} \leftarrow (U^d_\theta)_{int,mb}\chi_{int} + (U^d_\theta)_{bc}\chi_b$
 9:         $loss_{mb} = L(U^d_\theta)$
10:         $\theta \leftarrow optimizer(\theta, \alpha, \nabla_\theta(loss_{mb}))$
11:     end for
12: end for
```

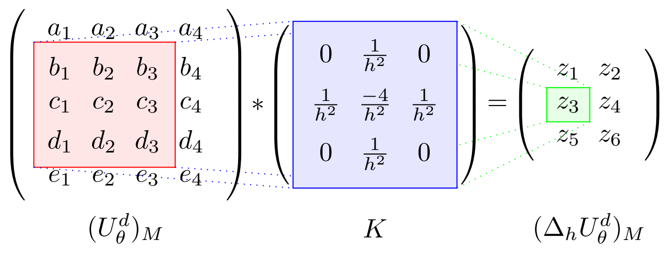

Figure 3: $(U^d_\theta)_M \in \mathbb{R}^{5\times4}$ is the matrix view of $U^d_\theta \in \mathbb{R}^{20}$, $K \in \mathbb{R}^{3\times3}$ is the 5-point Laplacian kernel; and $(\Delta_h U^d_\theta)_M \in \mathbb{R}^{3\times2}$ is the matrix view of the discrete Lapalacian $\Delta_h U^d_\theta \in \mathbb{R}^6$. This figure shows the calculation of the discrete derivatives using convolution operations. Specifically, it shows the calculation of the 5-pt Laplacian at the $(3,2)$ location of the grid shown in Figure (2). We have, $z_3 = +b_2\left(\frac{1}{h^2}\right) + c_1\left(\frac{1}{h^2}\right) + c_3\left(\frac{1}{h^2}\right) + d_2\left(\frac{1}{h^2}\right) - c_2\left(\frac{4}{h^2}\right)$ is the discrete Laplacian value calculated at $(3,2)$. The convolution results in a Laplacian matrix that is reduced in size. To keep the size same, padding can be applied on $(U^d_\theta)_M$. This is discussed in (2.3).

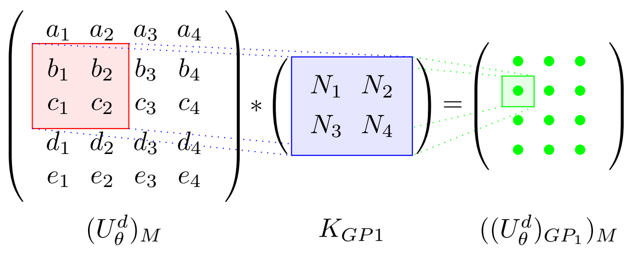

Figure 5: Quadrature quantity evaluation in FEM context. $(U^d_\theta)_M$ is the matrix view of the nodal values.K_{GP1} is kernel containing the basis function values at "gauss point - 1" (top left corner). This convolution results in the function values evaluated at the Gauss point "1" of each element (marked green). $(U^d_{\theta}{}_{GP1})_M$ is the matrix of this result. Function values (or their derivatives) evaluated at Gauss points can then be used in any integral evaluation. For example, $\int u^h dD = |J| \sum_{I\in M}\left[\sum_{i=1}^{4}(w_i(U^d_\theta)_{GPi})_M\right]$, where $|J|$ is the transformation Jacobian for integration and w are the quadrature weights.

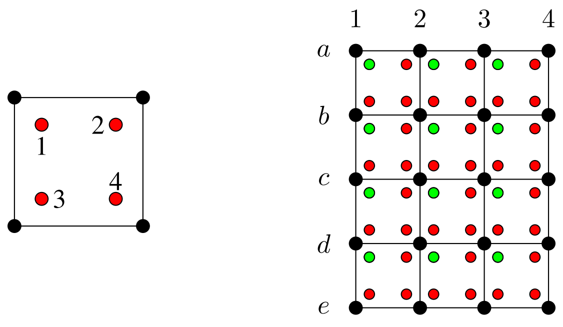

Figure 4: (Left) A single 2D element in FEM, with black dots denoting "nodes" and red dots denoting 2×2 Gauss quadrature points. (Right) A finite element mesh, with 4×3 linear elements and 5×4 nodes. Each of these elements contains Gauss points for integration to be performed within that element. Within each element, the "first" quadrature point (marked "1" on left) is marked green, and others red.

in learning the interaction between the discrete points locally. Since, the network takes an input of a discrete grid representation (similar to an image, possibly with multiple channels) and predicts an output of the solution field of a discrete grid representation (similar to an image, possibly with multiple channels), this is considered to be similar to a image segmentation or image-to-image translation task in computer vision. U-Nets (Ronneberger, Fischer, and Brox 2015; Çiçek et al. 2016) have been known to be effective for applications such as semantic segmentation and image reconstruction. Due to its success in diverse applications, we choose U-Net architecture for DIFFNET.

2.6 Training algorithm for DIFFNET

We provide two versions of the training algorithm. (i) an algorithm for computing the solution for an instance of a PDE and (ii) an algorithm for approximating the solution for a parametric PDE. The model architecture and the loss function remain the same for both. For solution for an instance, we use a simple approach as explained in Algorithm (1).

While sampling from a distribution of coefficients/forcing field for a parametric PDE, we employ the mini-batch based optimization approach as explained in Algorithm (2). The sampling of the known quantities is performed by using sobol sampling methodology. For training the neural network, we predict the solution field using sampled inputs and compute the loss using the FDM/FEM loss. We employ Adam optimizer with a learning rate of $1E-5$ for performing the optimization of the neural network parameters, θ using the gradients obtained from the FDM/FEM loss.

3 Results

We present results from a canonical PDE in two and three dimensions. Specifically, we consider the Poisson equation.

3.1 Linear Poisson's equation in 2D with forcing

For the first illustration, we consider a very simple problem of Poisson equation:

$$-\underline{\nabla} \cdot (\nu(\underline{x})\underline{\nabla} u) = f(\underline{x}) \text{ in } D \tag{11}$$

$$u|_{\partial D} = 0 \tag{12}$$

where $D = [0,1]^2$, a 2D square domain, ν is the *diffusivity* (or *permeability*) and is set to 1 for this problem. The forcing $f = f(\underline{x}) = f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$. The exact solution to this problem is given by $u_{ex}(x,y) = \sin(\pi x)\sin(\pi y)$. There are two ways to formulate this problem through DIFFNET architecture:

1. One way could be where we seek the mapping between the diffusivity field (which, in this case is just a matrix of 1's) and solution.
2. Or we could also seek the mapping between the forcing f and the solution.

In either case, the loss function (irrespective of the discretization scheme) remains exactly the same, written as follows:

- FDM based loss:

$$R = \sum_{i}^{N_{xy}} \|\underline{\nabla} \cdot [\nu(\underline{x}^i]\underline{\nabla} u(\underline{x}^i)) + f(\underline{x}^i)\|^2 \tag{13}$$

- weighted FEM based loss:

$$R = \frac{1}{2}\int \nu|\underline{\nabla} u|^2 d\underline{x} - \int uf d\underline{x} \tag{14}$$

In both cases, the Dirichlet boundary values are applied exactly by "padding" the 2D image/matrix representation of U^d_θ (see section (2.3)).

This problem is solved using DIFFNET according to the training process described in (2.6). The input data (S^d), discrete solution (U^d_θ) are shown in Figure (6) (the first two columns). This figure also contains a reference solution (column 3) calculated using a conventional (linear-system based) finite element method and the difference between the neural and numerical solution.

3.2 Poisson's equation with parametric log permeability

Our second illustration is a more practical one that is used as a model for simulating heat or mass transfer through an inhomogeneous media. The PDE and BC's are given by: The Poisson equation is given by

$$-\underline{\nabla} \cdot (\tilde{\nu}(\underline{x})\underline{\nabla} u) = 0 \text{ in } D \tag{15}$$

$$u(0,y) = 1 \tag{16}$$

$$u(1,y) = 0 \tag{17}$$

$$\frac{\partial u}{\partial n} = 0 \text{ on other boundaries} \tag{18}$$

where $D = [0,1]^2$ as in the example presented before; but $\tilde{\nu}$ is now a function of $\underline{x} = (x,y)$, rather than being constant.

In this case, do not have any forcing, so we really have only one set of data, which is $\tilde{\nu}$. So, for this problem we seek the mapping between $\tilde{\nu}$ and u.

Note: When the diffusivity is constant, the mapping between the forcing and the solution is linear (the previous example). But the mapping between u and $\tilde{\nu}$ is not linear. This can also be seen since $\tilde{\nu}$ and u are multiplied together in the PDE.

Here the diffusivity $\tilde{\nu}$ is parametric, and is represented by the following log permeability expression

$$\tilde{\nu}(\underline{x};\omega) = \exp\left(\sum_{i=1}^{m} \omega_i \lambda_i \xi_i(x)\eta_i(y)\right)$$

where ω_i is an m-dimensional parameter, λ is a vector of real numbers with monotonically decreasing values arranged in order; and ξ and η are functions of x and y respectively. We take $m = 4$, $\underline{\omega} = [-3,3]^4$ and $\lambda_i = \frac{1}{(1+0.25a_i^2)}$, where $\underline{a} = (1.72, 4.05, 6.85, 9.82)$. Also $\xi_i(x) = \frac{a_i}{2}\cos(a_i x) + \sin(a_i x)$ and $\eta(y) = \frac{a_i}{2}\cos(a_i y) + \sin(a_i y)$ The two sets of loss expressions can be written as:

- FDM based loss

$$R = \sum_{i}^{N_{xy}} \left\|\underline{\nabla} \cdot (\nu(\underline{x}^i)\underline{\nabla} u(\underline{x}^i)) - f(\underline{x}^i)\right\|^2 \tag{19}$$

- Integral based loss

$$R = \int \tilde{\nu}|\underline{\nabla} u|^2 d\underline{x} \tag{20}$$

In addition to Dirichlet conditions, we also have Neumann conditions applied in this problem. Dirichlet conditions are, once again, applied by padding the matrices. Neumann conditions can pose a slight challenge. In general, applying Neumann conditions in FDM results in an extra set of equations.. In FEM, especially in case of Poisson equation, Neumann conditions are naturally taken into the resulting discrete equations (no "extra" equations) in the form of surface/boundary integrals.

But since we are given "zero-Neumann" conditions, we take advantage of that, and apply it by replicating the values adjacent to the boundaries to the boundaries. In case of FDM

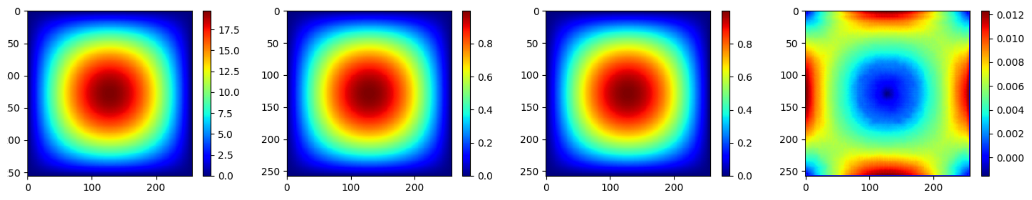

Figure 6: Solution to the linear Poisson's equation with forcing. From left to right: f, u_θ^h, u_{num} and $(u_\theta^h - u_{num})$. Here u_{num} is a conventional numerical solution obtained through FEM. Diffusivity $\nu = 1$

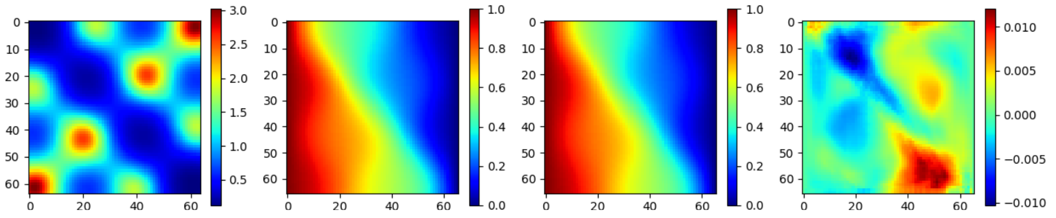

Figure 7: An example of solution corresponding to the coefficients $\underline{\omega} = (-0.26, -0.77, -0.37, -0.92)$ in the Poisson's equation with log permeability. From left to right: ν, u_θ^h, u_{num}, $(u_\theta^h - u_{num})$

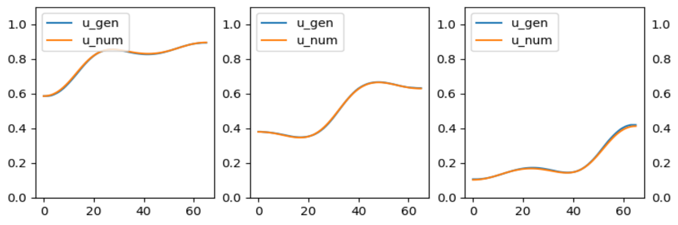

Figure 8: (Poisson's equation log permeability) Vertical line cuts of u_θ^h (u_gen) and u_{num}, at $x = 0.2, 0.5, 0.8$

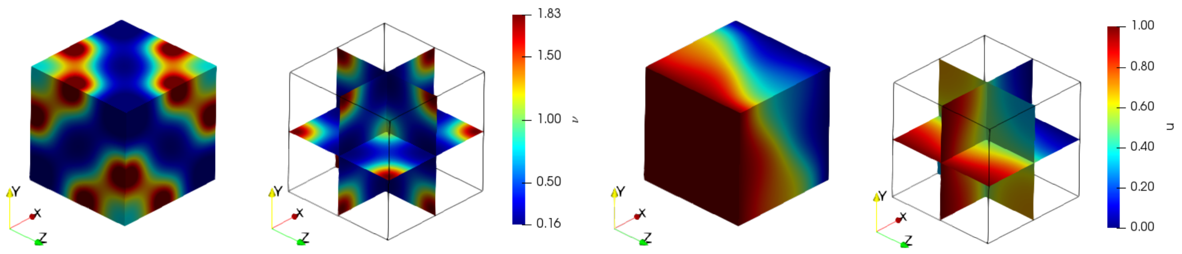

Figure 9: Example of $\nu(x, y, z)$ and the solution $u_\theta^h(x, y, z)$ to the 3D Poisson's problem with log permeability

loss, this strategy forms an approximate way of applying a zero-Neumann condition, since the optimization algorithm will have to "learn" this feature. But in case of FEM loss, this is an exact imposition of the zero-Neumann conditions.

The Dirichlet boundary conditions are applied exactly.

Instead of one instance of solution, we attempt to learn the distribution of the stochastic solution, given that the coefficients in the log permeability K-L sum come from a known range of values that depends on the parameter space ω.

Figure (7) shows $\tilde{\nu}$, u_θ^h, u_{num} (a solution from a numerical method, FEM in this case) and $u_\theta^h - u_{num}$. Figure (8) shows vertical line cuts for u_θ^h and u_{num}.

3-D Poisson's equation We finally show the ability to solve parametric PDE in 3D using the framework (Figure (9)).

4 Conclusions

In this paper, we integrate neural PDE methods with numerical methods by performing numerical differentiation (and integration, if needed) instead of using the spatial differentiability. We defined two loss functions (based on FDM and

FEM), for obtaining the neural approximate mapping between inputs and the discretized field solution. We demonstrate this framework's performance on poisson equation in 2D, steady heat transfer. We see that the results obtained from the neural PDE approximation (predicted) are match very closely with actual ground truth numerical solution.

References

Botelho, S.; Joshi, A.; Khara, B.; Sarkar, S.; Hegde, C.; Adavani, S.; and Ganapathysubramanian, B. 2020. Deep Generative Models that Solve PDEs: Distributed Computing for Training Large Data-Free Models. *arXiv preprint arXiv:2007.12792* .

Brenner, S.; and Scott, R. 2007. *The mathematical theory of finite element methods*, volume 15. Springer Science & Business Media.

Çiçek, Ö.; Abdulkadir, A.; Lienkamp, S. S.; Brox, T.; and Ronneberger, O. 2016. 3D U-Net: learning dense volumetric segmentation from sparse annotation. In *International conference on medical image computing and computer-assisted intervention*, 424–432. Springer.

Han, J.; Jentzen, A.; and Weinan, E. 2018. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences* 115(34): 8505–8510.

Hughes, T. J. 2012. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation.

Karumuri, S.; Tripathy, R.; Bilionis, I.; and Panchal, J. 2020. Simulator-free solution of high-dimensional stochastic elliptic partial differential equations using deep neural networks. *Journal of Computational Physics* 404: 109120.

Kharazmi, E.; Zhang, Z.; and Karniadakis, G. E. 2019. Variational physics-informed neural networks for solving partial differential equations. *arXiv preprint arXiv:1912.00873* .

Lagaris, I. E.; Likas, A.; and Fotiadis, D. I. 1998. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks* 9(5): 987–1000.

Lagaris, I. E.; Likas, A. C.; and Papageorgiou, D. G. 2000. Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks* 11(5): 1041–1049.

Larson, M. G.; and Bengzon, F. 2013. *The finite element method: theory, implementation, and applications*, volume 10. Springer Science & Business Media.

Lee, H.; and Kang, I. S. 1990. Neural algorithm for solving differential equations. *Journal of Computational Physics* 91(1): 110–131.

LeVeque, R. J. 2007. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM.

Malek, A.; and Beidokhti, R. S. 2006. Numerical solution for high order differential equations using a hybrid neural network—optimization method. *Applied Mathematics and Computation* 183(1): 260–271.

Michoski, C.; Milosavljevic, M.; Oliver, T.; and Hatch, D. 2019. Solving irregular and data-enriched differential equations using deep neural networks. *arXiv preprint arXiv:1905.04351* .

Paganini, M.; de Oliveira, L.; and Nachman, B. 2018. CaloGAN: Simulating 3D high energy particle showers in multilayer electromagnetic calorimeters with generative adversarial networks. *Physical Review D* 97(1): 014021.

Pang, G.; Lu, L.; and Karniadakis, G. E. 2019. fPINNs: Fractional physics-informed neural networks. *SIAM Journal on Scientific Computing* 41(4): A2603–A2626.

Raissi, M.; Perdikaris, P.; and Karniadakis, G. E. 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.

Ronneberger, O.; Fischer, P.; and Brox, T. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, 234–241. Springer.

Rudy, S.; Alla, A.; Brunton, S. L.; and Kutz, J. N. 2019. Data-driven identification of parametric partial differential equations. *SIAM Journal on Applied Dynamical Systems* 18(2): 643–660.

Samaniego, E.; Anitescu, C.; Goswami, S.; Nguyen-Thanh, V. M.; Guo, H.; Hamdia, K.; Zhuang, X.; and Rabczuk, T. 2020. An energy approach to the solution of partial differential equations in computational mechanics via machine learning: Concepts, implementation and applications. *Computer Methods in Applied Mechanics and Engineering* 362: 112790.

Sirignano, J.; and Spiliopoulos, K. 2018. DGM: A deep learning algorithm for solving partial differential equations. *Journal of computational physics* 375: 1339–1364.

Tompson, J.; Schlachter, K.; Sprechmann, P.; and Perlin, K. 2017. Accelerating eulerian fluid simulation with convolutional networks. In *International Conference on Machine Learning*, 3424–3433. PMLR.

Trefethen, L. N. 2000. *Spectral methods in MATLAB*. SIAM.

van der Meer, R.; Oosterlee, C.; and Borovykh, A. 2020. Optimally weighted loss functions for solving PDEs with Neural Networks. *arXiv preprint arXiv:2002.06269* .

Yang, L.; Zhang, D.; and Karniadakis, G. E. 2018. Physics-informed generative adversarial networks for stochastic differential equations. *arXiv preprint arXiv:1811.02033* .

Zhu, Y.; Zabaras, N.; Koutsourelakis, P.-S.; and Perdikaris, P. 2019. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *Journal of Computational Physics* 394: 56–81.