FastVPINNs: Tensor-Driven Acceleration of VPINNs for Complex Geometries

Thivin Anandh [1]   THIVINANANDH@IISC.AC.IN
Divij Ghose [1]   DIVIJGHOSE@IISC.AC.IN
Himanshu Jain [2]   MS19026@IISERMOHALI.AC.IN
Sashikumaar Ganesan [1*]   SASHI@IISC.AC.IN

[1] *Department of Computational and Data Sciences*
Indian Institute of Science, Bangalore
Karnataka, India
[2] *Department of Physical Sciences*
Indian Institute of Science Education and Research, Mohali
Punjab, India
*Corresponding author

Abstract

Variational Physics-Informed Neural Networks (VPINNs) utilize a variational loss function to solve partial differential equations, mirroring Finite Element Analysis techniques. Traditional hp-VPINNs, while effective for high-frequency problems, are computationally intensive and scale poorly with increasing element counts, limiting their use in complex geometries. This work introduces FastVPINNs, a tensor-based advancement that significantly reduces computational overhead and improves scalability. Using optimized tensor operations, FastVPINNs achieve a 100-fold reduction in the median training time per epoch compared to traditional hp-VPINNs. With proper choice of hyperparameters, FastVPINNs surpass conventional PINNs in both speed and accuracy, especially in problems with high-frequency solutions. Demonstrated effectiveness in solving inverse problems on complex domains underscores FastVPINNs' potential for widespread application in scientific and engineering challenges, opening new avenues for practical implementations in scientific machine learning.

Keywords: Physics-informed neural networks, Variational physics-informed neural networks, Domain decomposition, hp-Variational physics-informed neural networks

1 Introduction

At present, the realm of applied mathematics is witnessing substantial progress as a result of incorporating deep learning approaches in solving partial differential equations (PDEs). These methods, collectively termed scientific machine learning (SciML) (Cuomo et al. (2022); Baker et al. (2019); Psaros et al. (2023)), often complement or replace traditional solvers. Despite the dominance of conventional numerical methods such as the finite element method (FEM), which boasts several open-source solvers (Wilbrandt et al. (2017); Bangerth et al. (2007); Ganesan and Shah (2020)), SciML has rapidly evolved. This evolution has led to a growing repertoire of accessible SciML libraries (Lu et al. (2021a); NVIDIA Modulus).

Since the advent of physics-informed neural networks (PINNs) (Lagaris et al. (1998); Raissi et al. (2019)), the application of such methods has surged. PINNs enhance the

typical data-driven loss function of neural networks by incorporating an additional term that minimizes the residual of the underlying PDE and enforces physics-based constraints. Features such as the ease of obtaining gradients through automatic differentiation, and the ability to train networks for both forward and inverse modeling, make PINNs superior to traditional methods (Abueidda et al. (2021); Lu et al. (2021b)). Consequently, PINNs have found widespread application in various fields, including solid mechanics (Haghighat et al. (2021); Zhang et al. (2022)), fluid mechanics (Mao et al. (2020); Cai et al. (2021); Eivazi et al. (2022)), and geophysics (Smith et al. (2021)).

An extension of PINNs, Variational PINNs (VPINNs), utilize the weak form of the PDE in their loss functions, mirroring the Petrov-Galerkin framework used in conventional FEM (Kharazmi et al. (2019); Khodayi-Mehr and Zavlanos (2020)). Despite their computational demand due to the necessity to numerically compute integrals in the loss function, VPINNs represent a significant step forward in applying neural networks to PDEs.

Based on VPINN, hp-VPINNs have been developed to improve accuracy through domain decomposition (h-refinement) or increasing polynomial order (p-refinement) (Kharazmi et al. (2021)). Although hp-VPINNs offer superior accuracy in various problems, they suffer from critical limitations that hinder their practical application. These limitations include their computational complexity, linear scaling of training time with element count, and difficulty in handling complex geometries with skewed elements. These issues are evidenced in both academic and practical scenarios and have been a barrier to greater adoption despite various attempts to deploy hp-VPINNs in different applications (Radin et al. (2023); Yang and Foster (2021)).

Furthermore, recent developments such as convolutional PINNs (cv-PINNs) (Liu and Wu (2023)) introduced convolutional operations to accelerate the training of hp-VPINNs. Despite claims of superior performance, cv-PINNs still exhibited linear increases in training time with element count in 1D studies, and their solutions have been limited to domains suitable for decomposition into regular quadrilateral cells.

In response to these challenges, this work introduces FastVPINNs, a novel framework designed to address the limitations of hp-VPINNs. The proposed FastVPINNs leverage tensor-based operations to compute the loss, substantially reducing training times and diminishing the impact of increasing element counts. This advancement enables the handling of complex geometries, significantly broadening the practical applicability of our approach.

2 Preliminaries

2.1 Governing Equations

Consider a two-dimensional steady-state convection-diffusion equation:

$$
\begin{aligned}
-\varepsilon \Delta u(x) + \mathbf{b} \cdot \nabla u(x) = f(x), &\quad \text{in } \Omega \subseteq \mathbb{R}^2, \\
u(x) = g(x), &\quad \text{on } \partial\Omega.
\end{aligned}
\tag{1}
$$

Here, $x \in \Omega$, ε, and $\mathbf{b}$ are the diffusion coefficient and convective velocity, respectively. In addition, $f(x)$ is a known source function with appropriate smoothness. The Dirichlet boundary condition $u(x) = g(x)$ is imposed on the domain boundary $\partial\Omega$. The two-dimensional steady-state Poisson equation can be obtained from (1) by substituting $\mathbf{b} = (0,0)^T$ and

$\varepsilon = 1$.

$$-\Delta u(x) = f(x), \quad \text{in } \Omega \subseteq \mathbb{R}^2,$$
$$u(x) = g(x), \quad \text{on } \partial\Omega. \tag{2}$$

For the remainder of our discussion in this section, we will be using (2) as a reference.

2.2 Physics Informed Neural Networks

The result generated by a deep neural network is typically structured as a parametric function of the variable x, denoted as $u_{\text{NN}}(x; W, b)$. In this context, W and b represent the weights and biases of the network. When the neural network consists of h hidden layers, with the ith layer containing n_i neurons, the mathematical representation of the function takes the following shape:

$$u_{\text{NN}}(x; W, b) = l \circ \text{T}^{(h)} \circ \text{T}^{(h-1)} \dots \circ \text{T}^1(x) \tag{3}$$

Here, $l : \mathbb{R}^{n_h} \to \mathbb{R}$ is a linear mapping in the output layer and $\text{T}^{(i)}(\cdot) = \sigma(W^{(i)} \times \cdot + b^{(i)})$ is a non-linear mapping in the ith layer $(i = 1, 2, \cdots, h)$, with the non-linear activation function σ and the weights $W^{(i)}$ and biases $b^{(i)}$ of the respective layers.

The neural network's output $u_{\text{NN}}(x; W, b)$ can serve as an approximation to the unknown solution $u(x)$ of the equation 2 by training the deep neural network. Physics-Informed Neural Networks aim to achieve this estimation by systematically reducing the residual arising from the governing equations specified in equation 2 with $u_{\text{NN}}(x; W, b)$. Specifically, we refer to the residual related to the partial differential equation (PDE) and the boundary conditions as $\mathcal{P}$ and $\mathcal{B}$, respectively, that is,

$$\mathcal{P}(x; W, b) = -\Delta u_{\text{NN}}(x; W, b) - f(x), \quad \text{in } \Omega,$$
$$\mathcal{B}(x; W, b) = u_{\text{NN}}(x; W, b) - g(x), \quad \text{on } \partial\Omega. \tag{4}$$

The loss function employed in the neural network architecture is designed to integrate two primary components: a Partial Differential Equation (PDE) loss and a boundary loss. These components are defined at particular training points within the domain to reduce inaccuracies associated with the PDE and boundary constraints. This division of the loss function allows for a focused reduction in the difference between the neural network's forecasts and the expected physical phenomena described by the governing equations and boundary constraints. In particular, we define

$$L_p(W, b) = \frac{1}{N_T} \sum_{t=1}^{N_T} |\mathcal{P}(x; W, b)|^2,$$

$$L_b(W, b) = \frac{1}{N_D} \sum_{d=1}^{N_D} |\mathcal{B}(x; W, b)|^2, \tag{5}$$

$$L_{\text{PINN}}(W, b) = L_p + \tau L_b,$$

where N_T is the total number of training points in the interior of the domain Ω, N_D is the total number of training points on the boundary $\partial\Omega$ and τ is a scaling factor applied to control the penalty on the boundary term.

2.3 hp-Variational Physics Informed Neural Network

In this section, we initially establish the variational form of the Poisson equation (2), followed by introducing the Variational Physics Informed Neural Network. Let $\mathrm{H}^1(\Omega)$ denote the conventional Sobolev space, and define

$$V := \left\{ v \in \mathrm{H}^1(\Omega) : v = 0 \quad \text{on} \quad \partial\Omega \right\}.$$

The subsequent procedure consists of taking the equation (2), multiplying it by $v \in V$, integrating over Ω, and then utilizing integration by parts on the second derivative term. For more detailed information, please refer to (Ganesan and Tobiska (2017)). Consequently, the variational representation of the Poisson equation can be formulated as:

Find $u \in V$ such that,

$$a(u, v) = f(v) \quad \text{for all } v \in V,$$

where

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v \, dx, \qquad f(v) := \int_\Omega fv \, dx. \tag{6}$$

The domain Ω is then divided into an array of non-overlapping cells, labeled as K_k, where $k = 1, 2, \ldots, \texttt{N\_elem}$, ensuring that the complete union $\bigcup_{k=1}^{\texttt{N\_elem}} K_k = \Omega$ covers the entire domain Ω. In this context, we define V_h as a finite-dimensional subspace of V, spanned by the basis functions $\phi_h := \{\phi_j(x)\}$, $j = 1, 2, \ldots, \texttt{N\_test}$, where $\texttt{N\_test}$ indicates the total number of basis functions in V_h. As a result, the discretized variational formulation related to equation (6) can be written as follows,

Find $u_h \in V_h$ such that,

$$a_h(u_h, v) = f_h(v) \quad \text{for all } v \in V_h, \tag{7}$$

where

$$a_h(u_h, v) := \sum_{k=1}^{\texttt{N\_elem}} \int_{K_k} \nabla u_h \cdot \nabla v \, dK, \qquad f_h(v) := \sum_{k=1}^{\texttt{N\_elem}} \int_{K_k} fv \, dK.$$

These integrals can be approximated by employing a quadrature rule, leading to

$$\int_{K_k} \nabla u_h \cdot \nabla v \, dK \approx \sum_{q=1}^{\texttt{N\_quad}} w_q \, \nabla u_h(x_q) \cdot \nabla v(x_q) \,,$$

$$\int_{K_k} fv \, dK \approx \sum_{q=1}^{\texttt{N\_quad}} w_q \, f(x_q) \, v(x_q) \,.$$

Here, $\texttt{N\_quad}$ is the number of quadrature points in a cell.

The hp-Variational Physics Informed Neural Networks (hp-VPINNs) framework, as presented by Kharazmi et al. (2021), utilizes specific test functions v_k, where k ranges from 1 to N_elem, that are localized and defined within individual non-overlapping cells across the domain.

$$v_k = \begin{cases} v^p \neq 0, & \text{over } K_k, \\ 0, & \text{elsewhere.} \end{cases} \tag{8}$$

Here, v^p represents a polynomial function of degree p. This selection of test and solution spaces results in a Petrov-Galerkin finite element method. Specifically, u_h is estimated by $u_{\mathrm{NN}}(x; W, b)$, which is the neural network solution, whereas the test function v_h is a predetermined polynomial function. By utilizing these functions, we establish the cell-wise residual of the variational form (7) with $u_{\mathrm{NN}}(x; W, b)$ as

$$\mathcal{W}_k(x; W, b) = \int_{K_k} \left(\nabla u_{\mathrm{NN}}(x; W, b) \cdot \nabla v_k \; - \; f \, v_k \right) dK. \tag{9}$$

Further, define the variational loss by

$$L_v(W, b) = \frac{1}{\texttt{N\_elem}} \sum_{k=1}^{\texttt{N\_elem}} |\mathcal{W}_k(x; W, b)|^2 \tag{10}$$

and the cost function of the neural network in hp-VPINN as

$$L_{\mathrm{VPINN}}(W, b) = L_v + \tau L_b. \tag{11}$$

Here, L_b is the Dirichlet boundary loss as expressed in (5) and τ is a scaling factor applied to control the penalty on the boundary term.

Remark: The only distinction between VPINNs and hp-VPINN resides in the selection of the test function. Within the framework of VPINNs, a polynomial that has global support throughout the domain is employed as a test function. Conversely, in the context of hp-VPINN, a compilation of polynomials, each offering support on an element-wise basis, is utilized. Figure 1 represents the schematic of VPINNs for a 2D Poisson problem.

3 Implementation of hp-VPINNs

This section focuses on the existing implementation of hp-VPINNs. Initially, we explain the current approach to calculating the variational loss in hp-VPINNs using numerical integration methods like Gauss-Lobatto or Gauss-Legendre to achieve accurate integral approximations. Next, we will examine two key limitations of the existing implementation. Firstly, we will explore why the current approach leads to longer training times. Secondly, we will discuss why it may not be suitable for problems involving complex shapes.

3.1 Overview of Current hp-VPINNs Implementation

To calculate the cost function related to hp-VPINNs, as specified in Equation 11, it is essential to estimate the integration outlined in Equation 9. This estimation is accomplished by utilizing numerical integration methods, with the Gauss-Lobatto and Gauss-Legendre quadrature methods being particularly effective. As a result of applying these numerical techniques to the integral over the cell K_k, $\mathcal{W}_k(x; W, b)$ becomes

$$\mathcal{W}_k(x; W, b) \approx \sum_{q=1}^{\texttt{N\_quad}} w_q \left(\frac{\partial u_{\mathrm{NN}}(x_q; W, b)}{\partial x} \frac{\partial v_k(x_q)}{\partial x} + \frac{\partial u_{\mathrm{NN}}(x_q; W, b)}{\partial y} \frac{\partial v_k(x_q)}{\partial y} - f(x_q) v_k(x_q) \right)$$

Here, `N_quad` denotes the number of quadrature points in a given element K_k.

Algorithm 1: hp-VPINNs Implementation in Kharazmi et.al

```
# VPINNs Existing Implementation
def train_step():
  for k in N_elem:
    u_NN        = model(quadrature_points_in_curr_element) # get NN output
    u_x, u_y    = model.get_grad(u_NN)              # get sol gradients NN
    v_kx, v_ky  = fem.get_grad(v_k)                 # get test fn grad
    j,j_x,j_y   = fem.get_jacobians(K_k)            # get jacobians
    w           = fem.get_quad_wt(v_k, K_k)         # get quad wt
    r[]         = 0

    # compute PDE loss
    for q in N_quad: # number of test functions
      for j in N_test: # number of quadrature points
        grad_x = (j[q]/j_x[q])*w[q]*u_x[q]*v_kx[q][j] # du/dx.dv/dx
        grad_y = (j[q]/j_y[q])*w[q]*u_y[q]*v_ky[q][j] # du/dy.dv/dy
        r[j]  += (grad_x + grad_y - F[q][j])

    # Var loss (F is precomputed outside train loop)
    variational_loss += reduce_mean(square(r))

  dirichlet_loss = reduce_mean(square(model(input_bound_pts) - bd_actual)
  total_loss = variational_loss + tau * dirichlet_loss  # Total loss
```

Nomenclature:
`v_kx, v_ky`: gradients of the test functions in the x and y directions at the k-th cell. `u_x, u_y`: gradients of `u_NN` in the x and y directions (computed using AutoDiff). `j[q]`: Jacobian of the element at the specified quadrature point. `j_x[q], j_y[q]`: Jacobian of the element at the specified quadrature point in the x and y directions.

Algorithm 1 describes the approach for quantifying the loss function in the context of hp-VPINNs, following the guidelines provided in the GitHub repository (Kharazmi (2023)). This approach involves an iterative procedure that covers each element of the computational domain to compute the local PDE loss. Local losses are then combined to obtain the overall PDE loss measure. Central to this method is the need for gradients and function values of the test functions, which are obtained from predefined basis functions such as Jacobi polynomials. Simultaneously, the neural network provides solutions and gradients at specified points within each element. After computing the local losses, the algorithm consolidates these losses from all elements and combines them with the prescribed boundary loss, as detailed in equation (9). This unified loss function is crucial for training the neural network, ensuring that the hp-VPINNs model is finely adjusted not only to the underlying physics described by the PDE but also to the specified boundary constraints.
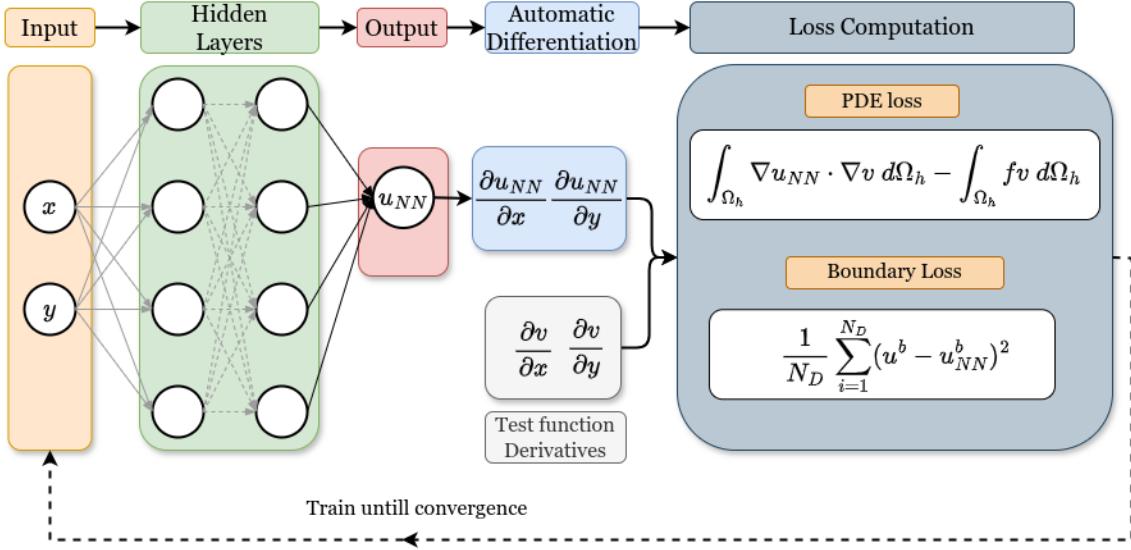
Figure 1: Schematics of Variational PINNs for a 2D Poisson problem.

3.2 Need for Efficient hp-VPINNs Implementation

hp-VPINNs employs h-refinement that restricts the test functions to smaller areas within the computational domain. This restriction of test functions allows hp-VPINNs to capture high frequency features in the solution. However, increasing the number of elements leads to a proportional increase in the time required to train the model.

- **Training time complexity**: The training time of hp-VPINNs increases linearly with the addition of more elements, even when the total number of quadrature points remains constant throughout the domain, as demonstrated in Figure 2. Consequently, employing h-refinement, which involves adding more elements to handle rapidly changing solutions, does not provide the anticipated advantage, as it substantially increases the computational cost.

- **Handling complex geometries**: The current version of hp-VPINNs is limited to operate with structured quadrilateral elements. This limitation stems from the current implementation using constant Jacobian values for each element to transform the gradients from the reference element to the actual element. However, practical applications often require skewed elements within complex domains as shown in Figure 3), where the Jacobian will not be constant within an element.

4 FastVPINNs Methodology

This section introduces FastVPINNs, a novel framework that significantly improves upon the existing hp-VPINNs. FastVPINNs tackles two critical challenges: handling complex geometries and achieving faster training times. We achieve these advancements through several key optimizations, including eliminating element-wise looping, removing redundant computations, and leveraging BLAS (Basic Linear Algebra Subprograms) routines. For
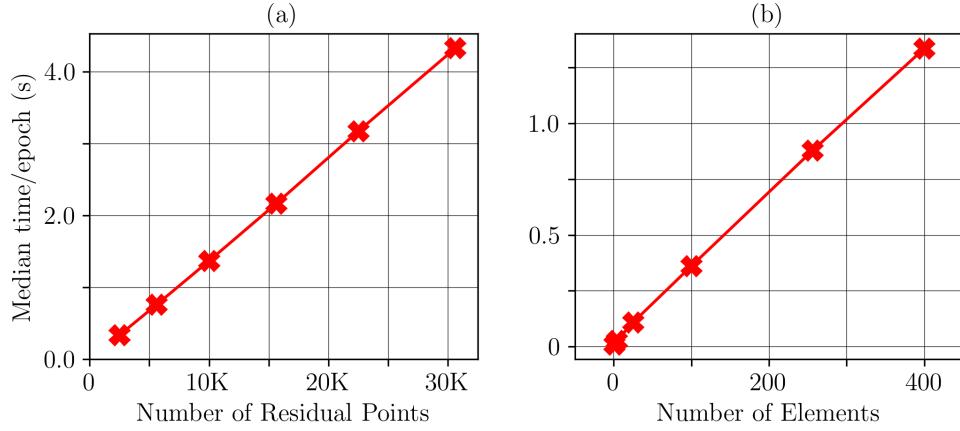
Figure 2: **hp-VPINNs: Effect of training time on Number of elements** (a) Number of residual points vs training time.(Each element has 25 quadrature points, so 30K quadrature points will have 1200 elements).(b) Number of elements vs training time(total number of quadrature points within the domain is kept constant at 6400).
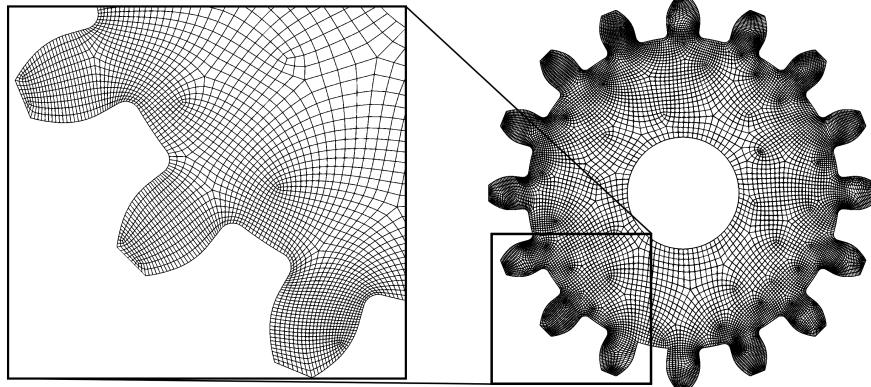

Figure 3: Mesh of spur gear with 14,000 quad elements.

handling complex geometries, we incorporate concepts of mapped finite elements by utilizing bilinear transformations.

4.1 Mapped hp-VPINNs

Complex geometric shapes frequently lead to the presence of irregular triangles or quadrilaterals in 2D, which pose challenges when it comes to accurately computing integrals and derivatives. To tackle this challenge, the FEM employs mapped finite elements. This approach converts all irregular elements in the actual domain into simpler shapes within a common reference element, as shown in Figure 4.

The adoption of mapped finite elements offers several advantages:

- The simplification of integrals and derivatives is achieved by moving these calculations to a reference element.

- The need to compute gradients multiple times for each element is eliminated, as the reference gradients are consistent across all elements in the domain.

In this work, the bilinear transformation is employed when dealing with quadrilateral elements. Further details on bilinear transformation can be found in the Appendix.
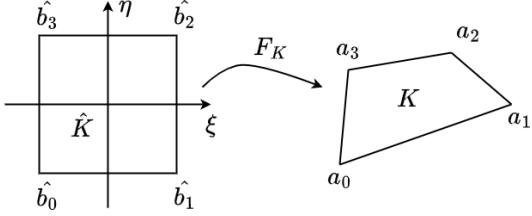
Figure 4: Bilinear transformation.

4.2 Optimization I: Enhancing Efficiency with Matrix-Vector Product Reformulation

Upon a detailed analysis of Algorithm 1, it is evident that its computational procedure consists mainly of iterative multiplication operations. These operations pertain to the derivatives of test functions and the quadrature weights utilized for numerical integration, which remain constant throughout the training cycle. The only component that undergoes variation at each training step is the predicted solution and its gradients derived from the neural network. Recognizing this recurrent pattern, we've devised an approach to modify the gradient computation, denoted as `grad_x` in Algorithm 1, to be expressed as a matrix-vector product. Initially, a matrix is prepared in advance, known as the precomputed test function matrix. This matrix, sized `N_test` $\times$ `N_quad`, comprises pre-computed values obtained by multiplying each shape function by the corresponding quadrature weights. During the training process, this matrix is multiplied by the solution of the neural network and a scaling factor known as Jacobian. This multiplication yields a vector of residuals, with dimensions `N_test` $\times$ 1. This has the following advantages.

- **Using BLAS for Faster Calculations:** By organizing our loss computations in a matrix-vector format, we can utilize the BLAS (Basic Linear Algebra Subprograms) routines. These optimized routines are designed to operate efficiently on GPUs, enhancing the speed of our calculations.

- **Elimination of Redundant Calculations:** Since the test function matrix is precomputed and remains unchanged during training, except for cases such as moving domain problems where the domain shape changes, it is not necessary to repeatedly compute the product of the derivatives of the shape functions and the quadrature weights at each training iteration. This efficient approach saves considerable time and

computational resources, thereby improving the speed and effectiveness of the training process.

4.3 Optimization II: Overcoming the Element Looping for regular elements

Despite the efforts made to reduce redundant computations and optimize matrix vector operations, our algorithm still requires the individual processing of each element for the calculation of loss within the computational domain. This means that during every training iteration, the neural network must calculate the solution `N_elem` times (known as forward passes) and then compute the gradients of these solutions another `N_elem` times (known as backward passes) to determine the loss. This sequential process is a major factor that contributes to the slowdown in the training of hp-VPINNs. For basic non-distorted quadrilateral elements (similar to those shown in Figure 5), the Jacobian values are constant for a given element. Instead of computing the gradients individually for each element, a single reference gradient matrix can be computed and later multiplied with the corresponding Jacobians to obtain the actual gradients for each element.

In our approach, we exploit these characteristics of a domain composed of regular elements to perform loss computation without the need to iterate through individual elements. The process involves gathering quadrature points, which are used for numerical integration, from all elements across the domain. These points are organized into a single input of dimensions (`N_quad` x `N_elem`, 2) and then supplied as input to the neural network.Consequently, the neural network produces an output in the form of a vector that can be reorganized into a matrix of size `N_quad` × `N_elem`. Each column of this matrix represents the predicted results at the quadrature points for each element. Subsequently, we proceed by taking each column of this matrix and multiplying it by the corresponding Jacobian. The loss calculation now involves multiplying the premultiplier matrix (reference gradient) with the gradient matrix from the neural network, which is scaled by the Jacobian. The result of this multiplication produces a final matrix with dimensions `N_test` × `N_elem`, where each column represents the residuals computed for each element within the actual domain.

The key advantages of this optimized approach include:

- **Single Backpropagation per Iteration:** Through the aggregation of inputs from all elements, we are able to compute the gradients for all elements simultaneously, requiring only a single forward pass and one backward pass (for gradient calculation) across the neural network.

- **Elimination of Element Looping:** By combining all inputs and using matrix multiplication, we avoid individually processing each element. This approach significantly accelerates our computations and enhances the overall computational efficiency.

Algorithm 2: Vectorised hp-VPINNs with elimination of element looping

---

```
# J_x, J_y: premultiplier jacobian matrices (N_quad, N_elem)
# F      : preassembled force function matrix (N_elem, N_test)
# V      : test function matrix (N_test, N_quad)
# V_x, V_y: test function gradient matrices (N_test, N_quad)
```
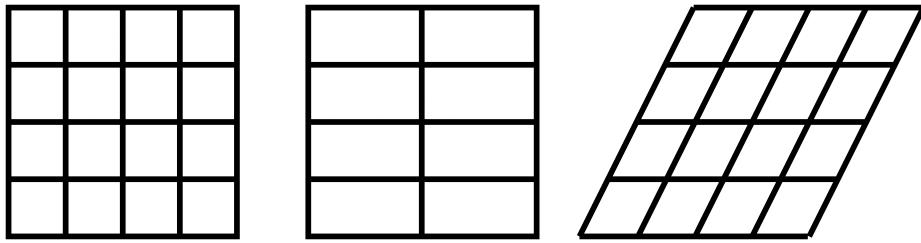
Figure 5: Quadrilateral meshes with elements having constant Jacobian.

```
# NOTE    : refer Appendix for pseudocode of pre-assembly matrices.
def train_step():
  u_NN = model(quadrature_points_in_curr_element)    # get NN output

  # get gradients and reshape them
  u_x, u_y = model.get_gradients().reshape(N_quad, N_elem)

  # Multiply jacobians of corresponding elements to sol gradients
  # Perform mat-mat multiplication
  grad_x = matmul(grad_x_mat, u_x * J_x) # int(du/dx.dv/dx),
  grad_y = matmul(grad_y_mat, u_y * J_y) # int(du/dy.dv/dy)

  # obtain mse for each element by row reduction (axis 0)
  residual_elements = reduce_mean(square(grad_x + grad_y - F), axis=0)
  variational_loss = reduce_sum(residual_elements) # sum all residuals
  total_loss = variational_loss + beta * dirichlet_loss   # Final loss
```

Algorithm 2 details a method for stacking input tensors to compute all gradients in a single backpropagation step, enabling efficient loss calculation for a domain composed of regular elements. The assembly process for the premultiplier matrices is described in the Appendix.

4.4 FastVPINNs: Generalized algorithm for Complex Geometries

Though the method we mentioned earlier results in notable speed enhancements, it operates under the premise that the Jacobian remains constant for every element. While this assumption is true for regular elements, it becomes less applicable when dealing with skewed quadrilaterals. In such scenarios, the Jacobian may vary at distinct integration points (quadrature points) and for each shape function.

To effectively tackle this challenge, it is essential to have a unique pre-multiplier matrix which stores the the gradients of the actual elements after transformation. However, using this strategy leads to a process that requires looping through each element to calculate loss, similar to the methodology outlined in Algorithm 1. This renders the previous optimization unfeasible for domains with skewed elements.

In this section, we present a novel approach known as FastVPINNs. This technique stacks the pre-multiplier matrices into a three-dimensional array, or third-order tensor, with dimensions $N\_elem \times N\_test \times N\_quad$. The neural network solutions are structured into a two-dimensional matrix sized $N\_quad \times N\_elem$. This configuration facilitates specific

11

operations using tensors, where each layer of the tensor (representing each element) is systematically multiplied by the corresponding column of the matrix of solution gradients. The result is a set of residual vectors, each of size `N_test` $\times$ 1 for every element. These vectors populate the columns of a final residual matrix sized `N_test` $\times$ `N_elem`, as illustrated in Figure 6. However, to calculate residuals element-wise using basic linear algebra subprograms (BLAS) such as tensor-matrix multiplication, it is necessary to transpose the dimensions of the solution gradients to `N_elem` $\times$ `N_quad`. This adjustment enables the utilization of TensorFlow's Abadi et al. (2015) `tf.linalg.matvec` function for efficient multiplication of tensor-matrix, resulting in a residual matrix of size `N_elem` $\times$ `N_test`. Subsequently, this matrix can be transposed to achieve the desired layout of `N_test` $\times$ `N_elem`.

The FastVPINNs approach offers several significant benefits:

- **Handles Complex Geometries:** Through the manipulation of tensors, this approach eliminates the necessity of iterating through individual elements, even when working with complex geometries.

- **Efficient Loss Computation with BLAS:** The loss computation, being formulated as tensor-based operations, is well-suited for GPU computations using BLAS routines. Additionally, GPUs typically have tensor cores designed specifically for tensor-based calculations, allowing these routines to leverage optimized hardware capabilities.

- **Requires Only One Backpropagation Pass:** In contrast to existing approaches that require multiple backpropagation to compute gradients, FastVPINNs accomplishes this in a single step even for complex geometries.
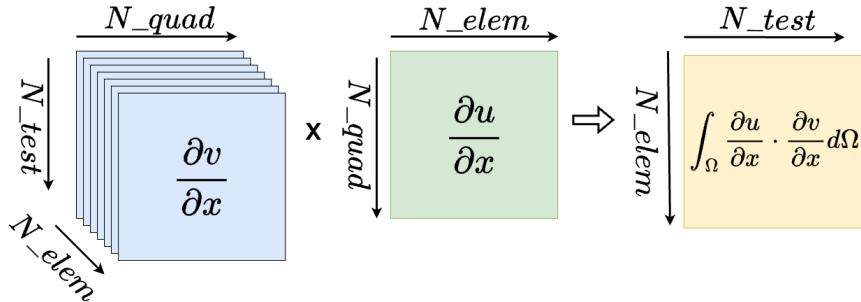

Figure 6: FastVPINNs Tensor schematic representation for residual computation.

Algorithm 3: Generalised Tensor-based FastVPINNs Algorithm

```
# Compute the prematrix multipliers for testFunction and gradients
# Obtain test fn and gradients in actual element and stack them
# to form a tensor of shape N_elem x N_test x N_quad
test_tensor   =  tf.stack(test_fn, axis=0)
grad_x_tensor =  tf.stack(grad_x_mat_list, axis=0)
grad_y_tensor =  tf.stack(grad_y_mat_list, axis=0)

# Compute Forcing Matrix similar to Algorithm 2

def train_step():
    # obtain solution for the quadrature points in entire domain
    sol = model(quad_points_in_domain)

    # obtain gradients and reshape them
    u_x, u_y = model.get_gradients().reshape(N_elem, N_quad)

    # Perform tensor-matrix mult to evaluate the integral
    grad_x = tf.transpose(tf.matvec(grad_x_tensor, u_x))
    grad_y = tf.transpose(tf.matvec(grad_y_tensor, u_y))

    # Subract the forcing matrix from diff to get residual
    residual_matrix = grad_x + grad_y - F

    # obtain mse error for each element in domain
    residual_elements =  reduce_mean(square(residual_matrix), axis=0)
    # sum residuals of all elem in domain
    variational_loss = reduce_sum(residual_elements)
    total_loss = variational_loss + beta * dirichlet_loss
```

Algorithm 3 summarizes the approach for FastVPINNs framework. In the next section, we assess the performance of the proposed FastVPINNs framework.

4.5 Experimental Setup and Design

In the present research, every experiment was conducted on a system that boasts an NVIDIA RTX A6000 GPU with 48 GB of device memory. The NVIDIA-Modulus library (NVIDIA, 2023) was used for physics-informed neural networks (PINN). To establish a baseline comparison with hp-VPINNs, we use the cutting-edge hp-VPINNs code available on GitHub (Kharazmi, 2023). To ensure uniformity in all our experiments, we opted for the same kind of Jacobi polynomials as outlined in (Kharazmi et al., 2021). These polynomials, denoted as P_n for degree n, follow the equation $P_n = P_{n+1} - P_{n-1}$. We also integrated the Gauss-Jacobi-Lobatto quadrature for numerical integration in our computations. The hp-VPINNs software generally functions using exttttf.float64 precision for floating-point computations. However, our code is compatible with both `tf.float64` and the slightly less accurate `tf.float32`. In all our experiments, we employed `tf.float32`, except when stated otherwise. The duration of each experiment was recorded using the `time.time()` function

in Python. In the following sections, the phrase "residual points" will refer to the collocation points in PINNs and the quadrature points used in hp-VPINNs and FastVPINNs..

Initially, we conduct a comparison between the accuracy of FastVPINNs and PINNs, and delve into the impact of h- and p-refinement on the convergence of FastVPINNs in Section 4.6.1. The duration of training of PINNs, hp-VPINNs, and FastVPINNs is compared in Section 4.6.2. The efficacy of PINNs and FastVPINNs in dealing with high-frequency solution problems is examined in Section 4.6.3. The capability of FastVPINNs to tackle forward problems in intricate geometries is exhibited in Section 4.6.4. Lastly, the performance of FastVPINNs in resolving inverse problems in intricate geometries is probed in sections 4.7.1 and 4.7.2.

4.6 Evaluation of FastVPINNs

Initially, we evaluate our code's efficacy on forward problems by solving the two-dimensional Poisson's equation within the unit square, using the specified forcing function

$$-\Delta u(x, y) = -2\omega^2 \sin(\omega x) \sin(\omega y) \quad \text{in} \in \Omega = (0, 1)^2,$$
$$u(0, \cdot) = u(\cdot, 0) = 0.$$

This problem has the exact solution,

$$u(x, y) = -\sin(\omega x) \sin(\omega y).$$

To evaluate the effectiveness of the suggested solver in different solution frequencies, we run the code with $\omega = 2\pi, 4\pi$ and 8π. The exact solutions for these cases are illustrated in Figure 7.
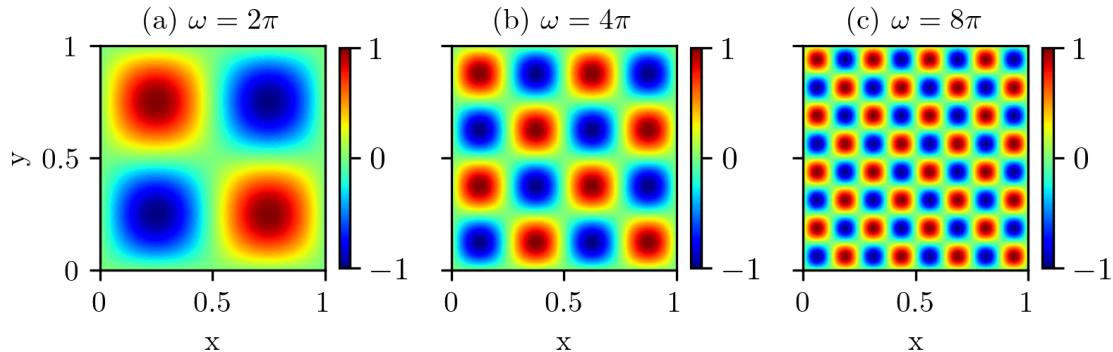

Figure 7: Exact solutions of the two-dimensional Poisson's equation for given test-cases.

4.6.1 ACCURACY TEST

In this section, we evaluate the accuracy of the proposed FastVPINNs framework by comparing it with the conventional PINNs technique on a problem where the parameter ω is set to 2π. This comparative analysis involves both methodologies, using neural networks

composed of three hidden layers, each containing 30 neurons. The FastVPINNs solution is derived using a grid of 2×2 elements, each element encompassing 40×40 quadrature points and 15 test functions per direction. The PINNs framework is trained using a total of 6400 collocation points, which corresponds to the total number of quadrature points employed in FastVPINNs. Both methodologies were trained for 100,000 iterations. The accuracy of their results was evaluated using a grid of 100×100 uniformly distributed points.
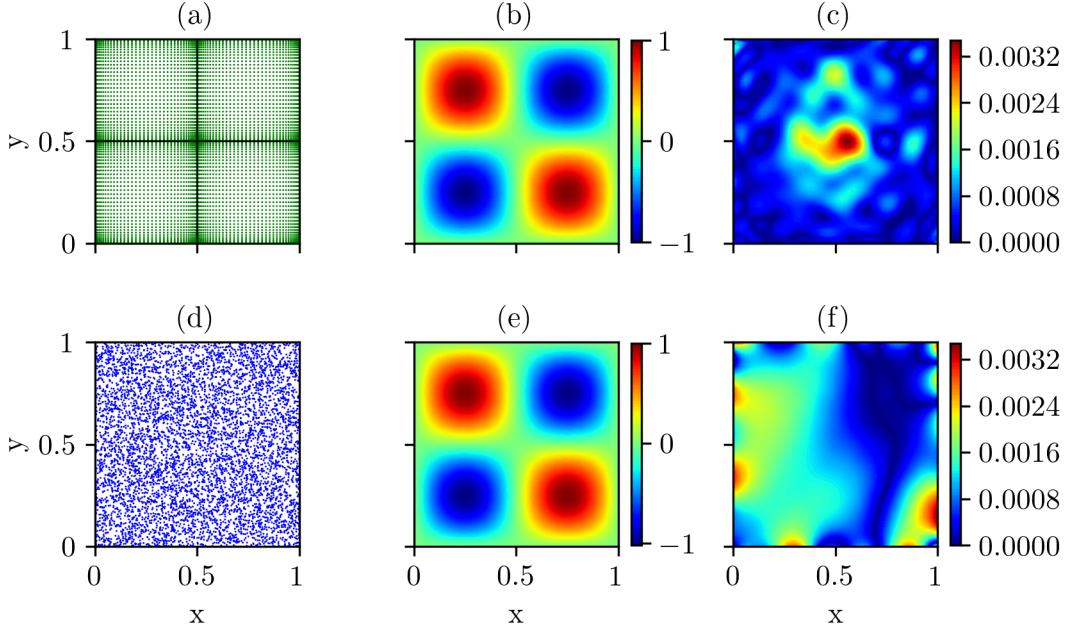
Figure 8: Accuracy comparison between PINNs and FastVPINNs. The quadrature points used to train FastVPINNs are shown in (a). The FastVPINNs prediction is shown in (b) and the point-wise error for the FastVPINNs solution is shown in (c). The PINNs is trained on collocation points as shown in (d). The PINNs solution and point-wise error are shown in (e) and (f), respectively.

As illustrated in Figure 8, the results show that FastVPINNs can reach a comparable level of accuracy as traditional PINNs. Furthermore, it is highlighted that the accuracy of FastVPINNs can be enhanced via h- and p-refinement.

The impact of h-refinement is examined using an example where $\omega = 4\pi$. The process begins with a single element, after which the domain is partitioned into 4×4 grids, and then into 8×8 grids, while keeping the number of quadrature points per element constant at 80×80 and using five test functions in each direction within each element. The model has difficulty accurately depicting the shape or magnitude of the solution when only one element is used. However, when the grid is expanded to 8×8 elements, the error is significantly reduced to approximately $\mathcal{O}(10^{-3})$, as illustrated in Figure 9(a). In the case of p-refinement, increasing the number of test functions in a 1×1 element grid from 5×5 to 20×20 reduces the error from $\mathcal{O}(10^{0})$ to $\mathcal{O}(10^{-2})$, as depicted in Figure 9(b), thereby proving the effectiveness

of p-refinement. A detailed error analysis for both h-refinement and p-refinement is provided in the Appendix B.1.
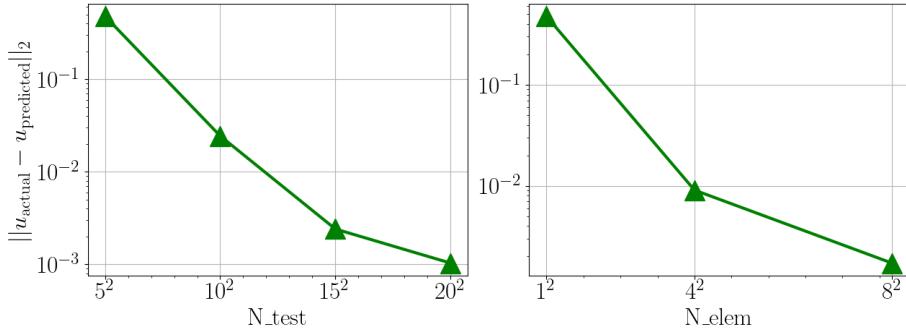

Figure 9: Left: Effect of p-refinement(left) and h-refinement(right) on FastVPINNs.

4.6.2 Efficiency Test: Comparison of FastVPINNs with PINNs and Hp-VPINNs

In this section, we assess the performance of our FastVPINNs framework by comparing it with two alternative methods: PINNs and the original version of hp-VPINNs. To measure computational proficiency, we recorded the duration needed to finish one training cycle for each framework. The evaluations were performed over 1,000 cycles to determine the median duration consumed. The results of the comparision are portrayed in Figure 10(a), where the x-axis indicates the count of residual points used for the calculations (collocation points for PINN and quadrature points for hp-VPINN) in addition to the total count of elements taken into account. For both hp-VPINNs and FastVPINNs, we applied five test functions on the x- and y-axes, culminating in a sum of 25 quadrature points per element and 25 test functions in total. This experimental arrangement facilitates a straightforward efficiency comparison between FastVPINNs and PINNs, using the count of collocation points (quadrature) as a shared factor. The results depicted validate that FastVPINNs surpass both PINNs and the original version of hp-VPINNs library in terms of training time, regardless of whether `tf.float32` or `tf.float64` precision is employed.

In contrast to the linear rise in training time observed with the increase in elements for the original version of hp-VPINNs, FastVPINNs demonstrate a nearly steady training time up to 400 elements, as depicted in Figure 10(b)

Section 4.8 provides additional understanding regarding the influence of different hyperparameters such as the count of test functions, quadrature points, and elements on the training duration of FastVPINNs.

4.6.3 Performance of FastVPINNs on higher frequency problems

Spectral bias is a phenomenon that describes the slower learning rate of neural networks for higher frequencies compared to lower frequency solutions during the training process Cao et al. (2020). The h-refinement technique can confine test functions to a more restricted area within our domain, enabling the network to detect higher frequency solutions in that
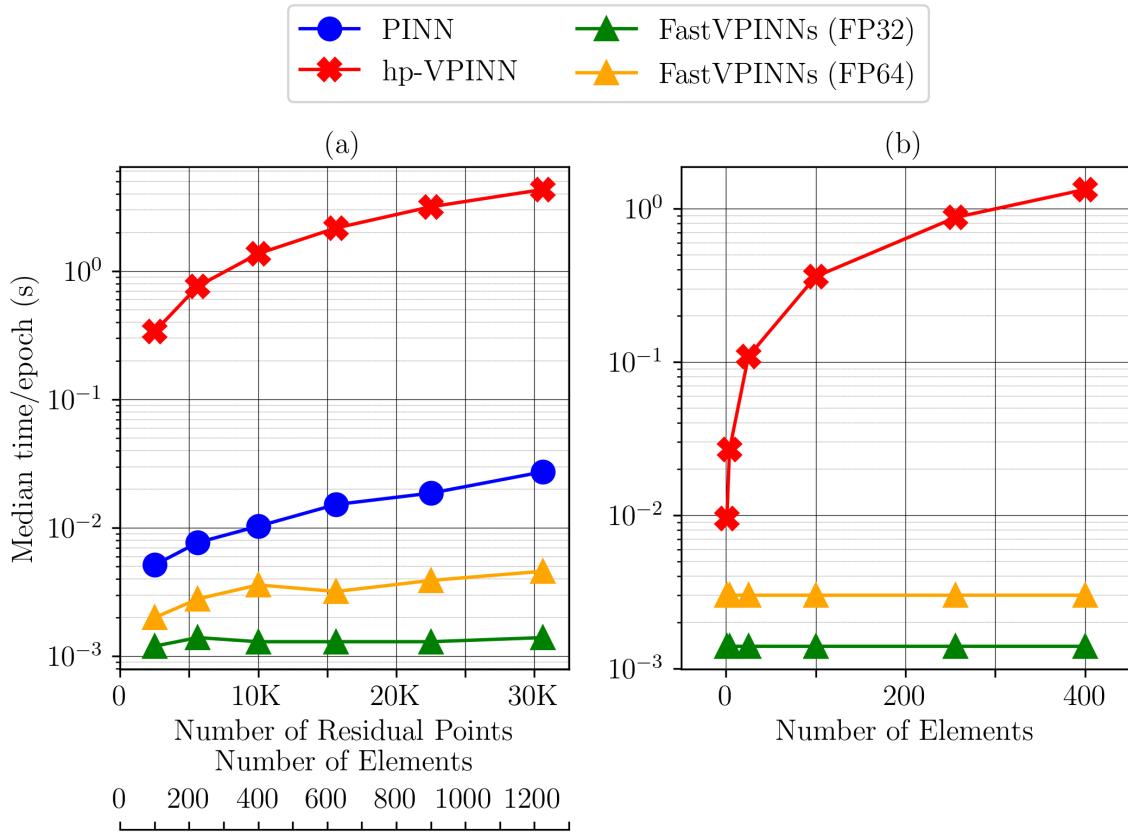
Figure 10: (a) Variation of computational time with the number of quadrature (residual) points, plotted against the median time taken per epoch; (b) Comparison of computational time between hp-VPINNs and FastVPINNs for varying numbers of elements.

specific area Kharazmi et al. (2021). In contrast to hp-VPINNs, which face longer training durations with an increased number of elements, the capability of FastVPINNs to sustain a consistent training time offers computational benefits for problems typified by higher-frequency solutions.

The aim of this section is to assess and contrast the efficacy of PINNs and FastVPINNs when dealing with solutions of varying frequencies, while keeping the number of residual points constant. In the FastVPINN scenario, the h-refinement will be adjusted according to the solution's frequency. For example, for a solution frequency of $\omega = 2\pi$, a 2x2 element set-up will be used, each comprising 40x40 quadrature points. If the frequency is $\omega = 4\pi$, the setup will be modified to 4x4 elements, each with 20x20 quadrature points, and so on. FastVPINNs uses five test functions in each direction. It is important to emphasize that the total count of quadrature points will remain fixed at 6,400 for all FastVPINNs configurations, equaling the number of collocation points used for PINNs training. Both models use the same parameters: 1000 Dirichlet boundary points, a neural network with

three hidden layers, each containing 30 neurons, trained over 100,000 iterations with a constant learning rate of 0.001 using the Adam optimizer Kingma and Ba (2014)
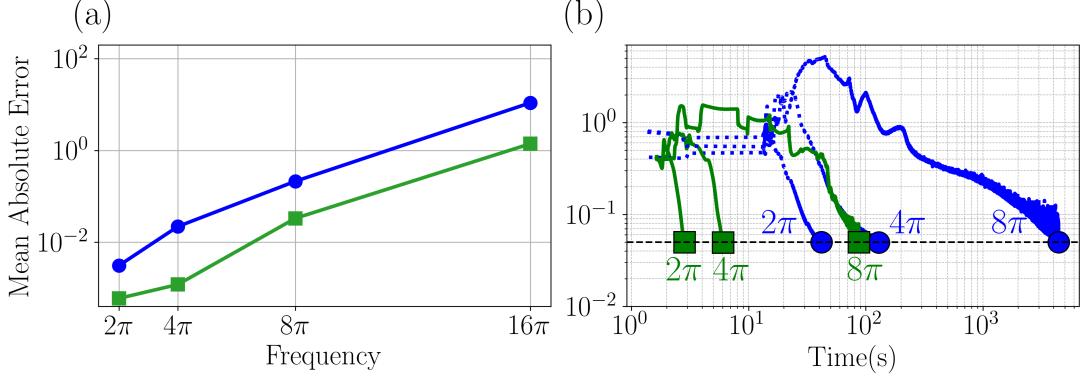

Figure 11: (a) Comparing MAE for PINNs and FastVPINNs over different frequencies. (b) time taken for PINNs and FastVPINNs to reach an MAE of 5×10^{-2} for different frequencies.

Figure 11(a) demonstrates that FastVPINNs consistently outperform PINNs in Mean Absolute Error (MAE) across all tested frequencies, emphasizing the benefits of h-refinement. Furthermore, FastVPINNs reach the MAE target threshold more rapidly than traditional PINNs, as illustrated in Figure 11(b), highlighting their superior speed and accuracy. This also underscores the importance of carefully selecting hyperparameters such as `N_elem`, `N_test`, and `N_quad` for FastVPINNs to optimize convergence and achieve efficiency and accuracy surpassing conventional PINNs.

4.6.4 FastVPINNs on Complex Geometries

In order to assess the proficiency of the FastVPINN framework in handling intricate domains with a large number of elements, we used it in a 2D convection-diffusion problem (12)on a spur gear domain (Ω_{gear}) depicted in Figure 3. The gear mesh, made up of 14,192 quadrilateral cells, was created using Gmsh(Geuzaine and Remacle (2009)).

$$
\begin{aligned}
-\varepsilon \Delta u + \mathbf{b} \cdot \nabla u &= f, \quad \mathbf{x} \in \Omega_{\text{gear}}, \\
u &= 0, \quad \mathbf{x} \in \partial\Omega_{\text{gear}},
\end{aligned}
\tag{12}
$$

where

$$
f = 50 \times \sin(x) + \cos(x); \quad \epsilon = 1, \quad \mathbf{b} = [0.1, \ 0]^T.
$$

During this experiment, we used four testing functions per direction and assigned 25 quadrature points to each component. This setup resulted in a total of 354,800 quadrature points, plus an extra 6,096 points specifically for the Dirichlet boundary ($\partial\Omega_{\text{gear}}$). The neural network was made up of 3 layers, each layer containing 50 neurons, and was launched with an initial learning rate of 0.005. This rate was adjusted through a method that reduced

18

it by a factor of 0.99 for every 1,000 training iterations. The model was trained for a total of 150,000 iterations.
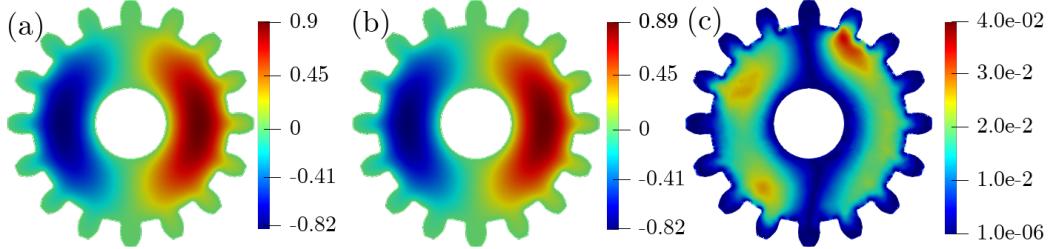

Figure 12: (a) Exact Solution obtained using FEM. (b) Predicted solution-FastVPINNs (c) Pointwise absolute error.

Figure (12) shows the exact and predicted solution along with the absolute point error. It should be noted that the model training was remarkably quick, finishing in less than 35 minutes, and each iteration took approximately 13 milliseconds. This highlights the efficiency of FastVPINNs in solving problems in domains with larger element counts and skewed elements, which are challenging for the original version of hp-VPINNs. The comparison of prediction times between FEM and FastVPINNs for varying numbers of degrees of freedom (DOFs) is discussed in Appendix B.2.

4.7 Investigating Inverse Problems with FastVPINNs

In PINNs, inverse problems involve the use of neural networks to infer unidentified PDE parameters, aided by observed solutions at scattered data points, usually in complex physical systems. By enhancing the loss function with data from sensor points scattered across the 2D domain, the network is capable of simultaneously ascertaining the solution to the Poisson equation and the value of the parameter ϵ. Consequently, the loss function for this issue is formulated as follows.

$$L_{\text{VPINN}}(W, b) = L_v + \tau L_b + \gamma L_s,$$

In this context, L_s denotes the loss term that arises from the discrepancy between observed solutions at sensor points and their corresponding predictions from the neural network. The terms L_v and L_b represent the variational loss and the boundary loss, respectively, as indicated in (11). The hyperparameters τ and γ are adjusted to suit the specific problem and the chosen architecture. This study investigates two separate categories of inverse problems related to predicting diffusion parameters: one involves uniform diffusion parameter prediction in a Poisson-2D problem, and the other involves space-dependent diffusion parameter prediction in a Convection-Diffusion-2D problem.

4.7.1 ESTIMATING CONSTANT DIFFUSION PARAMETERS

In the context of uniform parameter inverse problems, we consider a two-dimensional (2D) Poisson equation as given in Eq. (13). The objective is to predict the constant diffusion parameter, denoted as ϵ, and the corresponding solution $u(x)$.
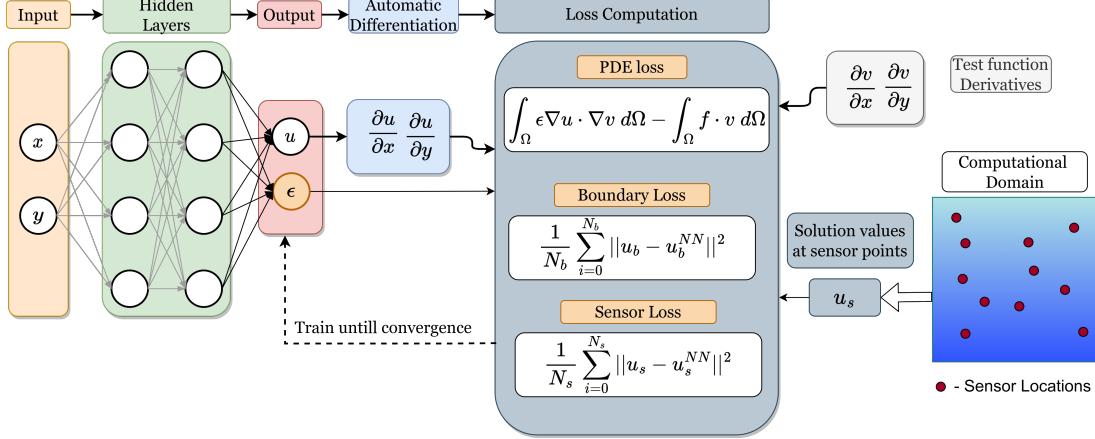
19

Figure 13: Schematics of VPINNs for space dependent inverse parameter modelling.

$$-\epsilon \Delta u(x) = f(x), \quad x \in \Omega = (-1,1)^2 \tag{13}$$

Suppose the epsilon (ϵ) is spatially invariant, we can incorporate it as a single trainable variable within the FastVPINNs framework. To demonstrate this approach, we built a FastVPINN with a 30×3 neural network architecture, 50 randomly distributed sensor points, a 2×2 element domain discretization, and 40×40 quadrature points per element. The exact solution to this problem is $u(x,y) = 10\sin(x)\tanh(x)e^{-\epsilon x^2}$. The initial guess of epsilon ($\epsilon_{\text{initial}}$) is taken as 2, and the actual value of epsilon (ϵ_{actual}) is 0.3. Training is carried out until it reaches the convergence criteria of $|\epsilon_{\text{predicted}} - \epsilon_{\text{actual}}| < 10^{-5}$. The network converged in 8909 epochs with a mean absolute error of the solution at 6.6×10^{-2}. The total training time was close to 18 s with a mean time per epoch of 2 ms. Figure 14 presents the exact and predicted values of both the solution and ϵ.

4.7.2 Predicting Space-Dependent Diffusion Parameters

This section addresses the more challenging case of predicting space-dependent parameters in partial differential equations (PDEs). As illustrated in Figure 13, the architecture of the neural network is modified to output both the solution and the space-dependent diffusion parameter at each point, enabling their simultaneous prediction. The network is guided by sensor point data, which are substituted here with the exact solution to accurately determine both the solution and its corresponding diffusion parameter. In this example, we consider a two-dimensional (2D) convection-diffusion equation as shown in Equation (14), solved on a circular domain using 1024 quadrilateral elements.

$$-\left(\frac{\partial}{\partial x}\left(\epsilon(x,y)\frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y}\left(\epsilon(x,y)\frac{\partial u}{\partial y} \right) \right) + b_x \frac{\partial u}{\partial x} + b_y \frac{\partial u}{\partial y} = f, \quad \mathbf{x} \in \Omega \tag{14}$$

where

$$f = 10; \quad \epsilon_{\text{actual}}(x,y) = 0.5 \times (\sin x + \cos y); \quad b_x = 1.0; \quad b_y = 0.0.$$
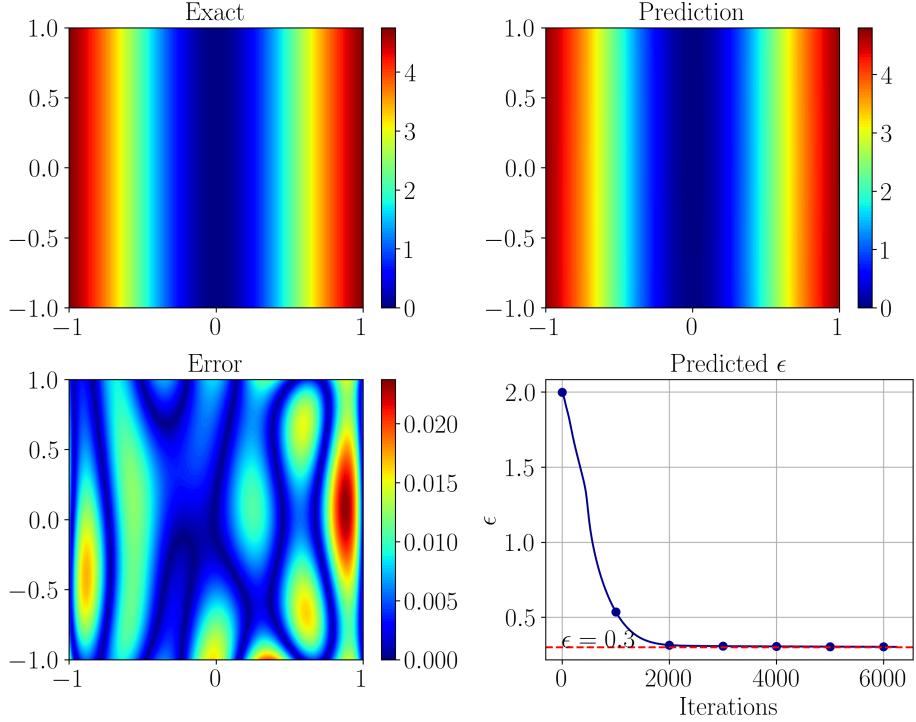
20

Figure 14: Inverse problem on FastVPINNs with constant diffusion parameter.

15 shows the reference FEM solution, FastVPINNs solution and the corresponding pointwise error for both solution and ϵ. We observe that the FastVPINNs framework can reasonably predict both the solution and the spatially varying diffusion parameter with errors of the order of $\mathcal{O}(10^{-2})$

4.8 Impact of Hyperparameters on FastVPINNs' Performance

To conclude our evaluation of FastVPINNs, we investigated the impact of three critical hyperparameters on training time using a Poisson2D problem as an example. These hyperparameters are: N_quad (quadrature points per element), N_test (shape functions per element), and N_elem (number of elements). All experiments utilized a neural network architecture consisting of 30 layers with 3 neurons each, and conducted over 1000 training iterations. The median time per epoch was used for data visualization. Figure 16(a) illustrates the relationship between N_test and N_quad for a fixed N_elem of 1, showing that the training time per epoch remains relatively constant regardless of the number of N_test functions, up to 10,000 quadrature points. Beyond this threshold, an increase in training time becomes noticeable, which depends on the parameters involved. Figure 16(b) shows the impact of varying N_test and N_elem with a fixed N_quad of 100 per element. Here, the execution time remains stable across different N_test functions until the number of elements reaches 100, beyond which it begins to increase. Figure 16(c) shows the effects of varying N_quad and N_elem while maintaining a constant N_test of 100 per element. Figures suggest that while the overall memory occupied by the tensors might not change for certain
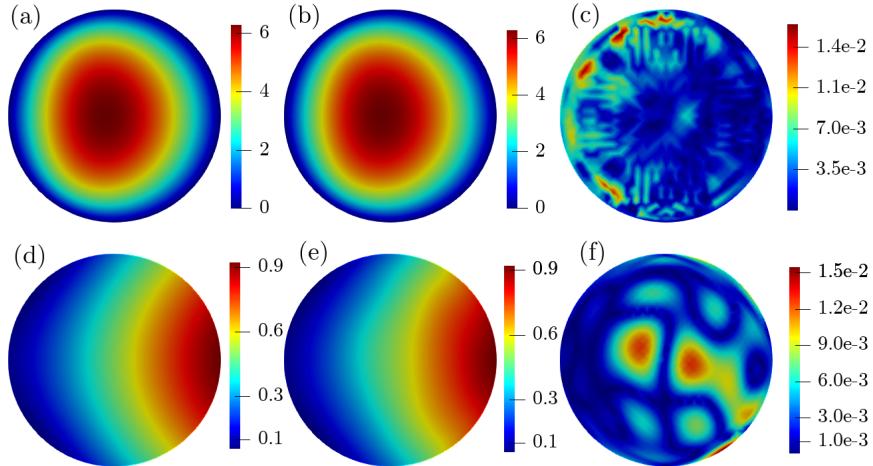
Figure 15: (a) Exact solution(u) obtained from FEM(ParMooN). (b) Predicted Solution-FastVPINNs. (c) Absolute error: FEM vs FastVPINNs solution. (d) Exact solution diffusion parameter(ϵ_{actual}). (e) Predicted diffusion parameter by FastVPINNs($\epsilon_{\text{predicted}}$) (f) Absolute error: ϵ_{actual} vs $\epsilon_{\text{predicted}}$.

combinations of N_test and N_quad, the number of quadrature points (N_quad) has a more pronounced impact on training time compared to the number of test functions (N_test). This observation can be attributed to the tensor-based loss computation, which performs a reduction operation along the N_quad dimension, thereby underscoring the importance of N_quad in influencing training time.

5 Conclusion

In this work, we addressed the existing challenges of hp-VPINNs, which excel in capturing high-frequency solutions but struggle with the extended training times associated with problems involving a large number of elements and complex geometries. We introduced FastVPINNs, a novel framework that employs tensor-based computations to significantly reduce training time dependence on the number of elements and to efficiently handle complex meshes. We demonstrated a 100x speedup compared to the existing hp-VPINNs implementation. Additionally, with proper hyperparameter selection, FastVPINNs can surpass state-of-the-art PINN codes in both speed and accuracy. Our method's ability to manage complex geometries with a large number of elements was showcased by solving a forward problem on a 14,000-element gear quad mesh. We successfully solved inverse problems in complex domains, as evidenced by estimating a diffusion parameter in a circular domain with 1,024 elements. This task was completed in less than 200 seconds for 100,000 epochs, which would have taken hours with the existing hp-VPINNs implementation. We also examined how hyperparameters, such as the number of test functions (N_test), quadrature
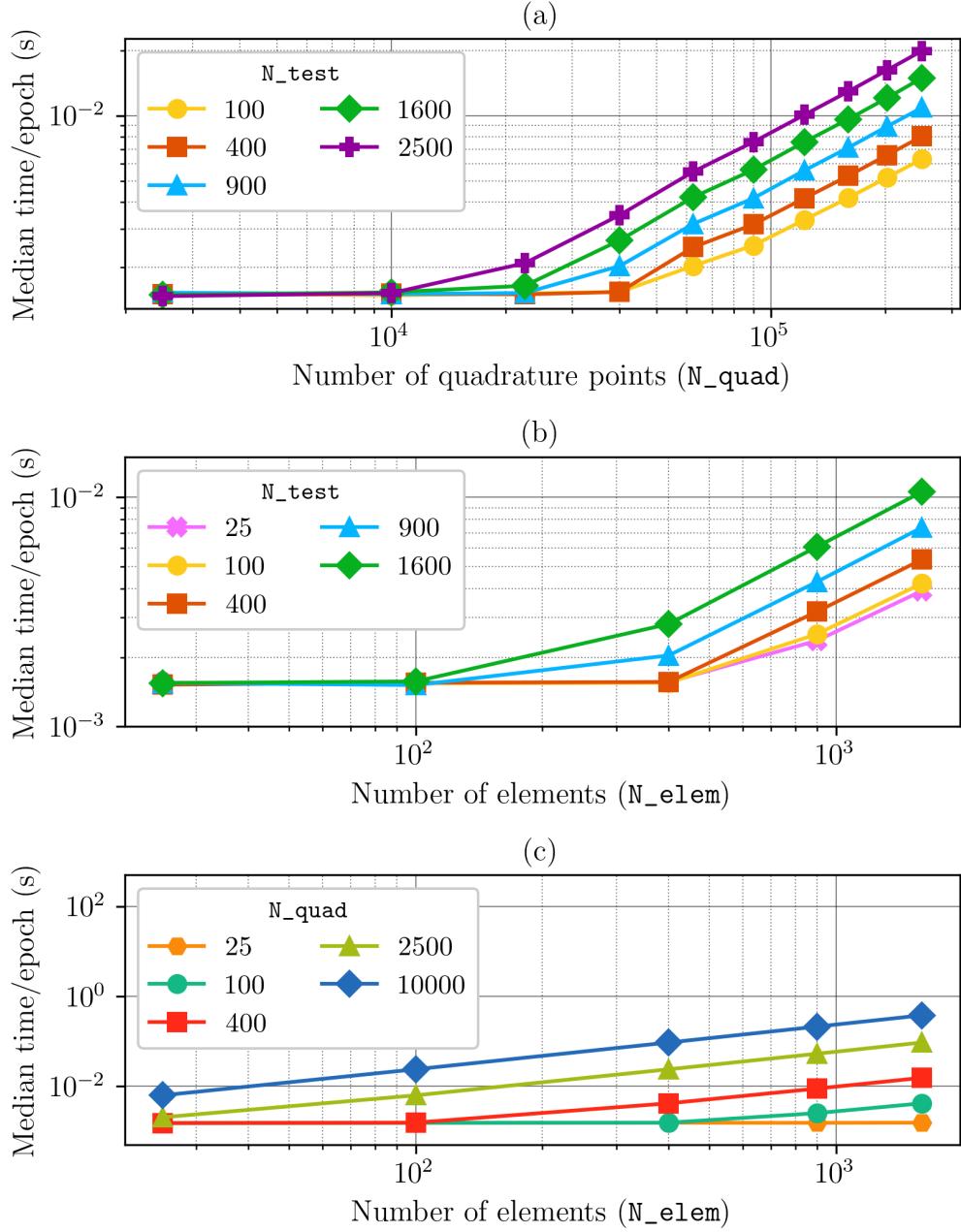
Figure 16: Comparative analysis of the median training time per epoch for different hyperparameters: (a) Number of test functions (N_test) vs Number of quadrature points (N_quad) with (N_elem=1), (b) Number of Elements (N_elem) vs Number of test functions (N_test) with quadrature points fixed at 10x10 (N_quad=10*10), (c) Number of elements (N_elem) vs Number of quadrature points (N_quad) with fixed number of test functions (N_test=10*10).

points (`N_quad`), and element count (`N_elem`), influence the training time. This analysis offers valuable insights for selecting the most suitable hyperparameters for specific problems. The versatility of FastVPINNs unlocks its potential for real-world applications in fields such as fluid dynamics, where complex geometries and massive datasets are prevalent. Looking ahead, we aim to expand the capabilities of FastVPINNs to include triangular elements and 3D domains, further broadening its impact in scientific and engineering fields.

Acknowledgments and Disclosure of Funding

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

D. W. Abueidda, Q. Lu, and S. Koric. Meshless physics-informed deep learning method for three-dimensional solid mechanics. *International Journal for Numerical Methods in Engineering*, 122(23):7182–7201, 2021.

N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee. Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence. 2 2019. doi: 10.2172/1478744. URL `https://www.osti.gov/biblio/1478744`.

W. Bangerth, R. Hartmann, and G. Kanschat. deal. II—a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4): 24–es, 2007.

S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis. Physics-informed neural networks (PINNs) for fluid mechanics: A review. *Acta Mechanica Sinica*, 37(12):1727–1738, 2021.

Y. Cao, Z. Fang, Y. Wu, D.-X. Zhou, and Q. Gu. Towards understanding the spectral bias of deep learning, 2020.

S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.

H. Eivazi, M. Tahani, P. Schlatter, and R. Vinuesa. Physics-informed neural networks for solving Reynolds-averaged Navier–Stokes equations. *Physics of Fluids*, 34(7), 2022.

S. Ganesan and M. Shah. SParSH-AMG: A library for hybrid CPU-GPU algebraic multigrid and preconditioned iterative methods. *arXiv preprint arXiv:2007.00056*, 2020.

S. Ganesan and L. Tobiska. *Finite elements: Theory and algorithms*. Cambridge University Press, 2017.

C. Geuzaine and J.-F. Remacle. Gmsh: A 3–D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009. doi: https://doi.org/10.1002/nme.2579. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2579`.

E. Haghighat, M. Raissi, A. Moure, H. Gomez, and R. Juanes. A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 379:113741, 2021.

E. Kharazmi. hp-VPINNs: High-performance variational physics-informed neural networks, 2023. URL `https://github.com/ehsankharazmi/hp-VPINNs`. Accessed: 2023-12-01.

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

E. Kharazmi, Z. Zhang, and G. E. Karniadakis. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, 374:113547, 2021.

R. Khodayi-Mehr and M. Zavlanos. Varnet: Variational neural networks for the solution of partial differential equations. In *Learning for Dynamics and Control*, pages 298–307. PMLR, 2020.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `https://api.semanticscholar.org/CorpusID:6628106`.

I. Lagaris, A. Likas, and D. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998. doi: 10.1109/72.712178.

C. Liu and H. Wu. cv-PINN: Efficient learning of variational physics-informed neural network with domain decomposition. *Extreme Mechanics Letters*, 63:102051, 2023. ISSN 2352-4316. doi: https://doi.org/10.1016/j.eml.2023.102051. URL `https://www.sciencedirect.com/science/article/pii/S2352431623000974`.

L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM review*, 63(1):208–228, 2021a.

L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo, and S. G. Johnson. Physics-informed neural networks with hard constraints for inverse design. *SIAM Journal on Scientific Computing*, 43(6):B1105–B1132, 2021b.

Z. Mao, A. D. Jagtap, and G. E. Karniadakis. Physics-informed neural networks for high-speed flows. *Computer Methods in Applied Mechanics and Engineering*, 360:112789, 2020.

NVIDIA. NVIDIA Modulus Documentation. `https://docs.nvidia.com/deeplearning/modulus/modulus-sym/index.html`, 2023. Accessed: 2023-03-10.

NVIDIA Modulus. `https://developer.nvidia.com/modulus`. Last accessed Jan 01, 2024.

A. F. Psaros, X. Meng, Z. Zou, L. Guo, and G. E. Karniadakis. Uncertainty quantification in scientific machine learning: Methods, metrics, and comparisons. *Journal of Computational Physics*, 477:111902, 2023.

N. Radin, S. Klinkel, and O. Altay. Effects of variational formulations on physics-informed neural network performance in solid mechanics. *PAMM*, page e202300222, 2023.

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

J. D. Smith, Z. E. Ross, K. Azizzadenesheli, and J. B. Muir. HypoSVI: Hypocentre inversion with Stein variational inference and physics informed neural networks. *Geophysical Journal International*, 228(1):698–710, 08 2021. ISSN 0956-540X. doi: 10.1093/gji/ggab309. URL `https://doi.org/10.1093/gji/ggab309`.

U. Wilbrandt, C. Bartsch, N. Ahmed, N. Alia, F. Anker, L. Blank, A. Caiazzo, S. Ganesan, S. Giere, G. Matthies, et al. ParMooN—a modernized program package based on mapped finite elements. *Computers & Mathematics with Applications*, 74(1):74–88, 2017.

M. Yang and J. T. Foster. hp-Variational Physics-Informed Neural Networks for Nonlinear Two-Phase Transport in Porous Media. *Journal of Machine Learning for Modeling and Computing*, 2(2), 2021.

E. Zhang, M. Dao, G. E. Karniadakis, and S. Suresh. Analyses of internal structures and defects in materials using physics-informed neural networks. *Science advances*, 8(7): eabk0644, 2022.

Appendix A. FastVPINNs Methodology

A.1 Bilinear Transformation

Let $b_0(-1, -1)$, $b_1(1, -1)$, $b_2(1, 1)$, $b_3(-1, 1)$ be the vertices of the reference element $\hat{K}$, see Figure 4. For any function $u(X)$, we denote $u(X) = u(F_k(\hat{X})) = \hat{u}(\hat{X})$. Further, the derivatives of the function, $\partial u/\partial x$ and $\partial u/\partial y$ on the original element can be obtained in terms of the derivatives defined on the refernce element $\partial\hat{u}/\partial\xi$ and $\partial\hat{u}/\partial\eta$ as follows:

$$\hat{u}(\hat{X}) = \hat{u}(F_k^{-1}(X)) = u(X), \tag{15}$$

$$\frac{\partial\hat{u}}{\partial\xi} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial\xi} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial\xi}, \tag{16}$$

$$\frac{\partial\hat{u}}{\partial\eta} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial\eta} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial\eta}. \tag{17}$$

we can express this relation as,

$$\begin{bmatrix} \frac{\partial\hat{u}}{\partial\xi} \\ \frac{\partial\hat{u}}{\partial\eta} \end{bmatrix} = \begin{bmatrix} (x_{c1} + x_{c3}\eta) & (y_{c1} + y_{c3}\eta) \\ (x_{c2} + x_{c3}\xi) & (y_{c2} + y_{c3}\xi) \end{bmatrix} \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix}.$$

where,

$$x_{c0} = \frac{(x_0 + x_1 + x_2 + x_3)}{4}, \qquad x_{c1} = \frac{(-x_0 + x_1 + x_2 - x_3)}{4},$$

$$x_{c2} = \frac{(-x_0 - x_1 + x_2 + x_3)}{4}, \qquad x_{c3} = \frac{(x_0 - x_1 + x_2 - x_3)}{4},$$

$$y_{c0} = \frac{(y_0 + y_1 + y_2 + y_3)}{4}, \qquad y_{c1} = \frac{(-y_0 + y_1 + y_2 - y_3)}{4},$$

$$y_{c2} = \frac{(-y_0 - y_1 + y_2 + y_3)}{4}, \qquad y_{c3} = \frac{(y_0 - y_1 + y_2 - y_3)}{4},$$

Finally, we have Finally, we have

$$\begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix} = \frac{1}{D} \begin{bmatrix} (y_{c2} + y_{c3}\xi) & -(y_{c1} + y_{c3}\eta) \\ -(x_{c2} + x_{c3}\eta) & (x_{c1} + x_{c3}\xi) \end{bmatrix} \begin{bmatrix} \frac{\partial\hat{u}}{\partial\xi} \\ \frac{\partial\hat{u}}{\partial\eta} \end{bmatrix},$$

where, D is the determinant of the Jacobian matrix

A.2 Algorithm : Premultiplier Assembly for Algorithm 2

Algorithm : hp-VPINNs vectorised code - Premultiplier Assembly

```
# Compute the prematrix multipliers for shapefunctions and gradients
# Shape: (N_test, N_quad)
for j in range(N_test):
  for q in range(N_quad):      '
    V_k[j][q]  = quad_wt[q] * v_k[j][q]   # Premult. shape mat
    V_x[j][q]  = quad_wt[q] * v_kx[j][q]  # Premult. grad x mat
    V_y[j][q]  = quad_wt[q] * v_ky[j][q]  # Premult. grad y mat

# Compute Forcing Matrix
# Shape (N_Elem, N_Test)
for n in range(N_elem):
  for j in range(N_test):
    for q in range(N_quad):
      # xi and eta are quadrature co-ordinates in reference domain
      x,y = fe.get_actual_coordinates(n,xi[q],eta[q])
      F[n][j] += j[n][q] * qw[q] * force_fn(x,y) * test[j][q]

# Collect the jacobian terms and stack them
# Shape: (N_quad, N_elem)
for q in range(N_quad):
  for n in range(N_elem):
    J_x[q][n] = j[n][q] / j_x[n][q] # Jacobian x matrix
    J_y[q][n] = j[n][q] / j_y[n][q] # Jacobian y matrix
```

Appendix B. Results
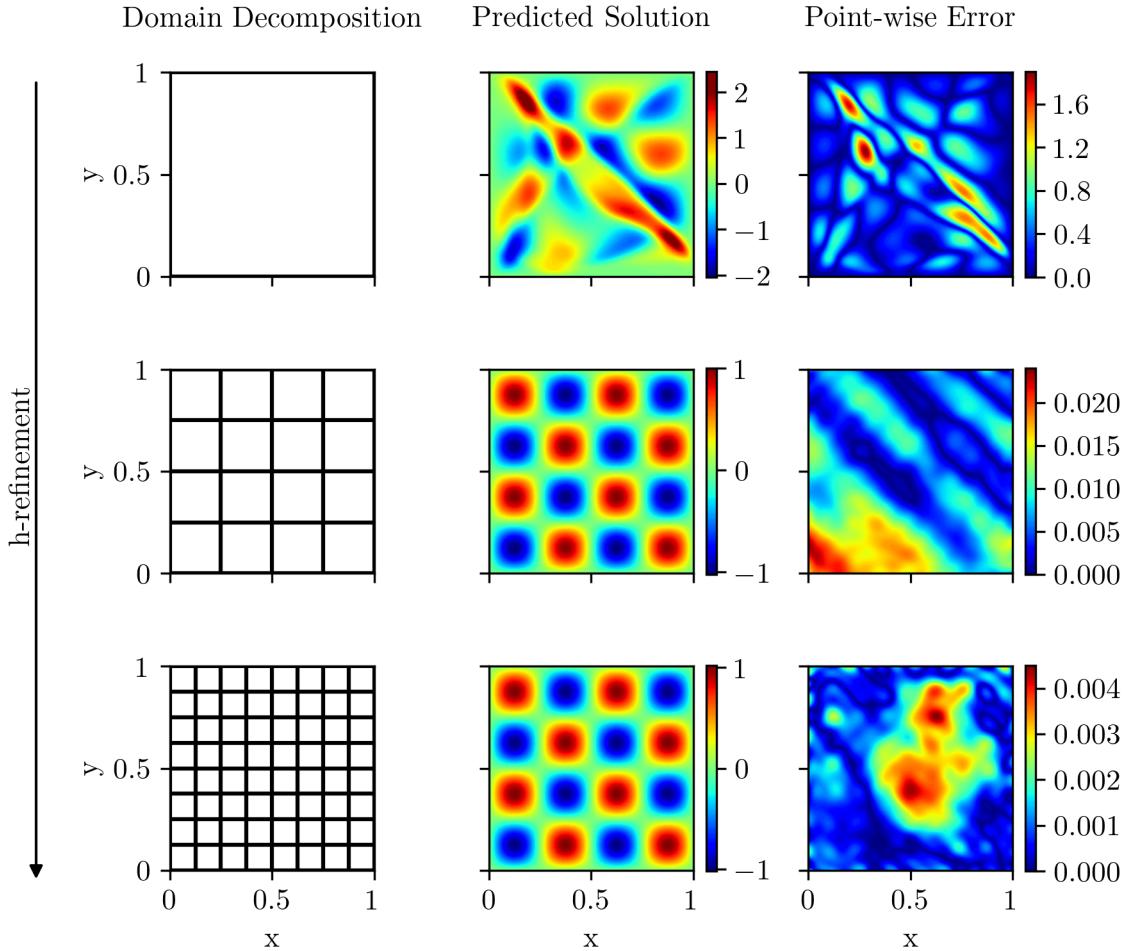
B.1 Effect of h- and p-refinement in FastVPINNs

Figure 17: Effect of h-refinement on the accuracy of solving the two-dimensional Poisson equation. The first, second, and third columns are the domain decomposition, FastVPINNs solution, and pointwise test error, respectively. From top to bottom: N_elem = 1, N_elem = 16 and N_elem = 64. For each element, we use 80×80 quadrature points and 5 test functions in each direction.
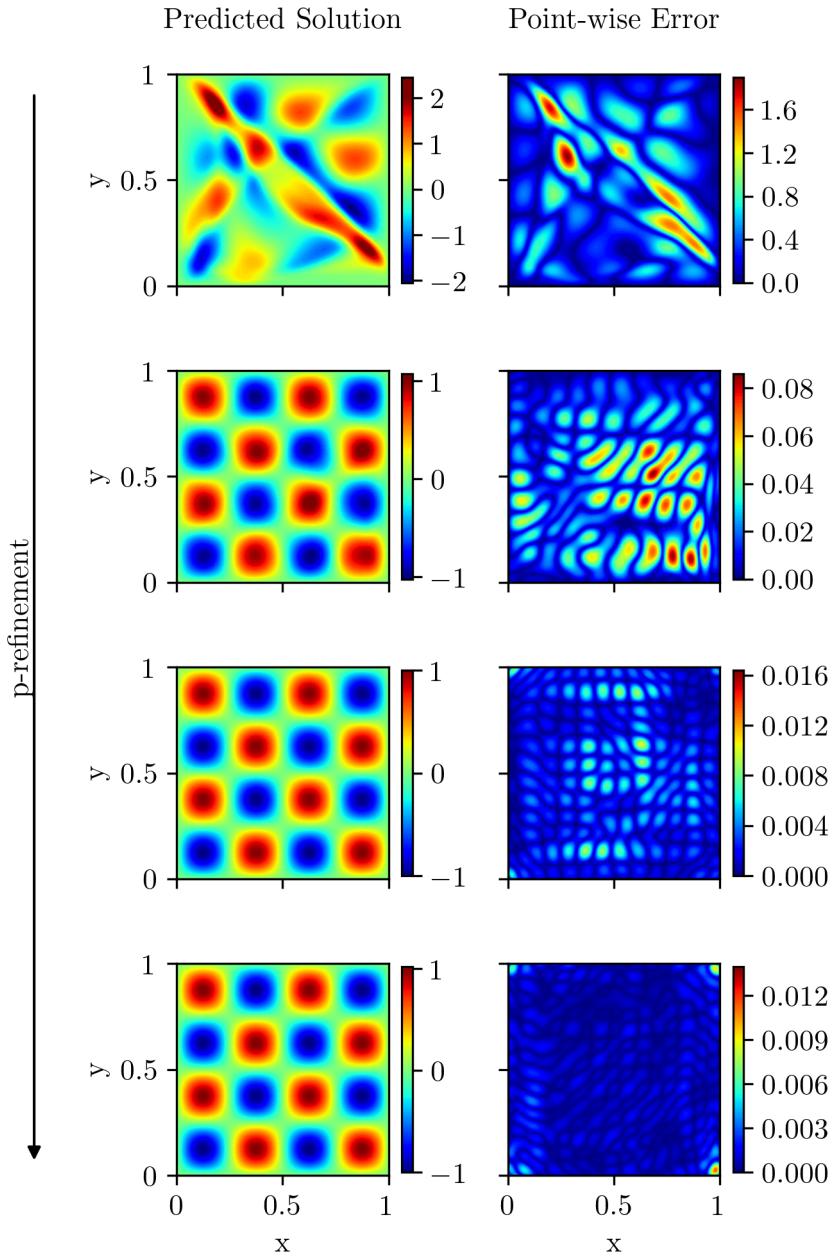
Figure 18: Effect of p-refinement on the accuracy of solution of two-dimensional Poisson's equation on 1 element. The first and second columns are the FastVPINNs solution and point-wise test error, respectively. From top to bottom: `N_test` = 5×5, `N_test` = 10×10, `N_test` = 15×15 and `N_test` = 20×20. The element has 80×80 total quadrature points.

Table 1: Time Taken (in s) for Prediction using FEM and PINNs

N_DOF	FEM	PINNs
29302	2.638	7.807e-4
115868	13.0352	7.978e-4
259698	32.220	1.777e-3
460792	50.680	3.139e-3
719150	81.784	4.890e-3
1034772	172.967	7.023e-3

B.2 Time Comparison - FastVPINNs vs FEM

The Table 1 and Figure 19 below represents the time required for the prediction of the solution using PINNs and FEM. It's important to note that for FastVPINNs, this time does not include the training time.
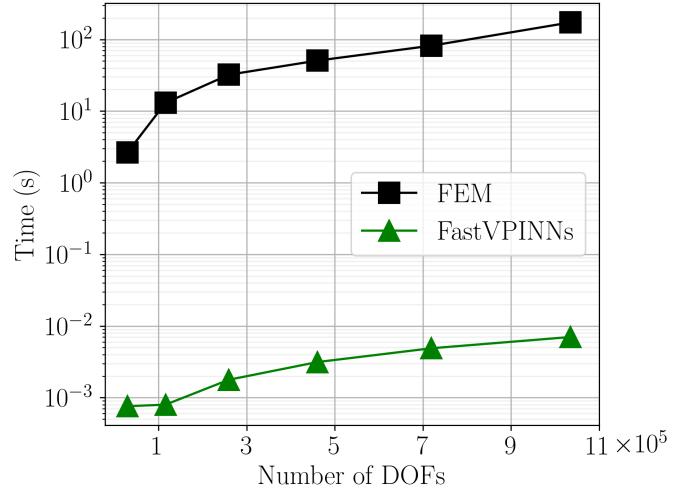

Figure 19: Comparision of time taken for prediction by FEM and PINNs for varying number of quadrature points.