A Hybrid Framework for Accelerating Linear Solvers for Partial Differential Equations

by

Yuhan Wu

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday August 26, 2025 at 15:00.

Student number:	*******	
Project duration:	December 19, 2024 – August 26, 2025	
Thesis committee:	Dr. Alexander Heinlein,	TU Delft, supervisor
	Prof. dr. Victorita Dolean-Maini,	TU Eindhoven, supervisor
	Dr. Francesca Bartolucci,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Contents

1 Introduction

Partial differential equations (PDEs) are equations involving multivariate functions and corresponding partial derivatives. PDEs play a significant role in describing various phenomena across different disciplines of science and engineering, such as fluid dynamics [1], solid mechanics [2], materials science [3], and thermal engineering [4]. For traditional numerical methods, such as the finite element, finite differences, or spectral methods, the discretization of linear partial differential equations typically results in a system of linear equations as the last step:

$$\mathrm{A}u = f$$

where A is the stiffness matrix, u represents the solution, and f denotes the forcing term. When solving large-scale linear systems, direct solvers face memory limitations and are challenging to parallelize effectively. Consequently, iterative solvers are preferred due to their lower memory requirements and better parallel scalability for large-scale linear systems.

Classical iterative methods, such as the Jacobi method and the Gauss-Seidel method [5], are typically suitable for well-conditioned or diagonally dominant problems. However, these stationary iterative methods inherently suffer from slow convergence, particularly for low-frequency eigenmodes, and might diverge when dealing with non-symmetric and indefinite systems [6]. Challenges in eliminating low-frequency eigenmode errors have motivated the development of multigrid methods [7, 8] which utilize a hierarchy of fine-to-coarse discretizations to effectively reduce eigenmode errors ranging from low to high frequencies.

Krylov subspace methods, such as Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES) methods [5, 9, 10], are widely-used iterative solvers in solving large-scale and sparse linear systems. Compared to classical stationary methods, Krylov subspace methods typically exhibit faster convergence, especially in large-scale problems. However, the convergence of Krylov subspace methods strongly depends on the conditioning of the matrix and the distribution of eigenvalues. Therefore, an appropriate preconditioner is often critical for the convergence performance.

Recently, Scientific Machine Learning (SciML) [11, 12] has emerged and rapidly developed as a powerful framework for solving various problems in computational science and engineering. Recent research has shown that it can be advantageous to solve differential equations by machine learning techniques [11]. Specifically, Physics-Informed Neural Networks (PINNs) [13] have received substantial attention. In contrast to data-driven approaches [14, 15] which rely on sufficient simulation or experimental data, PINNs explicitly encode PDE constraints into the loss function to predict the solution to PDEs. These neural network-based surrogate models, which are trained offline, can efficiently produce approximate solutions to PDEs, without relying on traditional numerical solvers. In addition, neural operators, a framework for learning mappings between functions, have emerged as a tool for learning the solution operator associated with PDEs. Deep Operator Networks (DeepONets) [14] and Fourier Neural Operators (FNOs) [15] have been successfully employed to predict parameterized PDE solutions with high computational efficiency.

However, deep neural networks typically exhibit a spectral bias [16], learning low-frequency components more easily while struggling to capture high-frequency features during training. The spectral

bias exists in many deep learning-based approaches for solving PDEs [17, 18, 19], such as PINNs and DeepONets, making them efficient in approximating low-frequency solutions while inadequate in learning high-frequency features.

To address this limitation, Zhang et al. [19] recently proposed the hybrid iterative numerical transferable solver (HINTS), and its variant HINTS-MG which integrates HINTS with the multigrid method. The HINTS approach combines the DeepONet with classical relaxation methods (e.g., Jacobi or Gauss-Seidel). By alternating between neural networks and classical iterative methods, this hybrid method leverages complementary frequency preferences of neural networks and traditional iterative solvers, and achieves robust and effective error elimination across different frequency ranges.

However, the HINTS method still exhibits several issues such as the mismatch between training and practical data distributions and mid-frequency errors that slow convergence. This thesis aims to answer the question: ”How can the HINTS framework be further enhanced to improve its robustness, convergence, and generalizability?” In this thesis, the Poisson equation and the indefinite Helmholtz equation with homogeneous Dirichlet boundary conditions will be used as benchmark problems to evaluate the performance of HINTS.

To address this question, the thesis first analyzes the convergence behavior of HINTS and investigates the mechanisms underlying its late-stage slowdown. Motivated by these findings, two methodologies, Gradient-Enhanced HINTS (GE-HINTS) and HINTS-in-the-loop training approaches, are developed to restore rapid convergence. The effectiveness of these methods is substantiated through a series of systematic numerical experiments.

The remainder of this thesis is structured as follows: Chapter 2 provides an overview of classical iterative solvers, including the Jacobi and Gauss-Seidel methods, Krylov subspace methods, and multigrid methods; Chapter 3 introduces the fundamental concepts of neural networks, demonstrates concepts and examples of spectral bias, and details the architecture of DeepONet in solving PDEs; Chapter 4 provides a detailed introduction to HINTS and HINTS-MG methods and primarily validates their performance on the Poisson and Helmholtz equations; Chapter 5 presents a detailed convergence analysis of the HINTS framework, identifies the phenomenon of slow convergence plateau, investigates its underlying causes, and reinterprets DeepONet’s role within the HINTS framework in the context of neural preconditioning; Chapter 6 introduces the anti-frequency principle and proposes Gradient-Enhanced HINTS (GE-HINTS) to mitigate DeepONet’s spectral bias; Chapter 7 presents the ”HINTS-in-the-loop” training strategy to overcome the data distribution mismatch in the HINTS framework; Chapter 8 summarizes the key contributions of this work, discusses limitations of the proposed methodologies, and outlines promising directions for future research.

2

Numerical Methods for Solving Linear Systems

Using different discretization methods such as the finite element method, finite difference methods, or spectral methods, a linear partial differential equation with appropriate boundary conditions is typically discretized into a system of linear equations as follows:

$$\mathrm{A}x = b, \quad \mathrm{A} \in \mathbb{R}^{n\times n}, x, b \in \mathbb{R}^n \tag{2.1}$$

In this chapter, we provide a review of classical numerical methods for solving linear systems, such as the Jacobi and Gauss-Seidel methods, the multigrid method, and Krylov subspace methods.

2.1. Direct Methods

Direct methods, such as the LU decomposition (for nonsingular matrices) and the Cholesky decomposition (for symmetric positive definite matrices), typically solve the linear system exactly (within machine precision) by performing a matrix factorization to decompose the problem into simpler subproblems.

In the LU decomposition, a nonsingular matrix A can be factorized as:

$$\mathrm{A} = \mathrm{LU},$$

where matrices L and U are lower and upper triangular, respectively. If A is symmetric and positive definite, the Cholesky decomposition yields:

$$\mathrm{A} = \mathrm{L}\mathrm{L}^\top$$

where L is a lower triangular matrix with positive diagonal entries.

These factorizations convert the original problem in Eq. (2.1) into the following simpler subproblems with triangular matrices:

$$\begin{cases} \mathrm{L}y = b \\ \mathrm{U}x = y \ \ (\text{or } \mathrm{L}^\top x = y). \end{cases} \tag{2.2}$$

Eq. (2.2) can be solved by forward and backward substitution methods. For a dense matrix A, the factorization typically costs $\mathcal{O}(n^3)$ arithmetic operations and $\mathcal{O}(n^2)$ storage. This leads to poor scalability, as computational and memory costs increase rapidly as n grows.

In the discretization of a PDE, the matrix A typically exhibits sparsity. However, factorization in direct methods might cause significant fill-in [5], so that many zero entries in the original matrix A become nonzero in the factorized matrices, thereby introducing substantially increased storage and computational costs in the denser factorized matrices.

Over the years, many highly optimized libraries, such as MUMPS [20], UMFPACK [21], and Pardiso [22], have been developed for the direct solution of linear systems and have been widely applied in open-source and commercial simulation software such as COMSOL Multiphysics [23].

2.2. Stationary Iterative Methods

In contrast to direct methods, which yield the exact solution only in the final step, iterative methods generate a sequence of intermediate solution approximations and terminate when a prescribed tolerance is satisfied. Iterative methods are generally advantageous for large, sparse linear systems due to lower computational complexity and memory requirements.

Stationary iterative methods are a classical class of iterative solvers, which solve a linear system using a stationary operator that approximates the original one [5]. At each iteration, the current approximate solution is updated by solving a correction equation derived from the residual and the approximate operator. Although the fixed-point iteration scheme underlying stationary iterative methods is straightforward to implement, their convergence is typically slow and only guaranteed for certain types of matrices, such as diagonally dominant or symmetric positive definite matrices [24].

2.2.1. Richardson Method

The Richardson method is one of the simplest iterative methods for solving the system of linear equations in Eq. (2.1). The basic iteration scheme for the Richardson method is defined as:

$$x^{(k+1)} = x^{(k)} + \alpha r^{(k)}, \quad r^{(k)} = b - \mathrm{A}x^{(k)}, \quad k = 0, 1, 2, ... \tag{2.3}$$

The above iteration is equivalent to applying the gradient descent method in solving the linear system, with α representing the step size. Eq. (2.3) can also be written as:

$$x^{(k+1)} = (\mathrm{I} - \alpha\mathrm{A})x^{(k)} + \alpha b \tag{2.4}$$

Let $x^\star$ be the exact solution such that $\mathrm{A}x^\star = b$, for the residual in the $k+1$th iteration $r^{(k+1)} = b - Ax^{(k+1)}$, we have:

$$\begin{aligned} r^{(k+1)} &= b - \mathrm{A}\left[(\mathrm{I} - \alpha\mathrm{A})x^{(k)} + \alpha b\right] \\ &= b - \mathrm{A}x^{(k)} + \alpha\mathrm{A}^2x^{(k)} - \alpha\mathrm{A}b \\ &= (\mathrm{I} - \alpha\mathrm{A})r^{(k)} \end{aligned} \tag{2.5}$$

The general procedure of the Richardson method are shown in Algorithm 1.

Algorithm 1 Richardson Method [24]

Require: Matrix $\mathrm{A} \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$, initial guess $x^{(0)} \in \mathbb{R}^n$, step size α, tolerance ϵ, maximum iterations $k_{\max}$.
Ensure: Approximate solution $x^{(k)}$

```
for k = 0, 1, 2, ..., k_max do
    Compute residual: r^(k) = b − Ax^(k)
    if ‖r^(k)‖ ≤ ε then
        return x^(k)
    end if
    Update solution: x^(k+1) = x^(k) + α r^(k)
end for
```

2.2.2. Jacobi and Gauss-Seidel Methods

Matrix Splitting

The Jacobi and Gauss-Seidel methods, as two of the most commonly used iterative solvers, are based on the splitting of the matrix $\mathrm{A} = \mathrm{M} - \mathrm{N}$, with matrix M being an easily invertible approximation of the matrix A. Let $\mathrm{A} = \mathrm{M} - \mathrm{N}$ with $det(\mathrm{M}) \neq 0$, we have the iterative scheme as follows:

$$x^{(k+1)} = \mathrm{M}^{-1}\mathrm{N}x^{(k)} + \mathrm{M}^{-1}b = x^{(k)} + \mathrm{M}^{-1}r^{(k)}, \quad k = 0, 1, \cdots \tag{2.6}$$

Since the linear system with matrix M needs to be solved at low cost, this is typically satisfied with a diagonal or a triangular matrix. The idea of splitting motivates the Jacobi method and the Gauss-Seidel method as follows:

- Jacobi Method:
$$\mathrm{A} = \mathrm{L} + \mathrm{D} + \mathrm{U};\ \mathrm{M} = \mathrm{D};\ \mathrm{N} = -(\mathrm{L} + \mathrm{U})$$
where $\mathrm{D} = \mathrm{diag}(\mathrm{A})$ with $\mathrm{A}_{ii} \neq 0$
- Gauss-Seidel Method:
$$\mathrm{A} = \mathrm{L} + \mathrm{D} + \mathrm{U};\ \mathrm{M} = \mathrm{L} + \mathrm{D};\ \mathrm{N} = -\mathrm{U}$$
where L and U are the strictly lower and upper triangular components of A, with $\det(\mathrm{M}) \neq 0$.

More flexibility and improvement of convergence behavior might be obtained by introducing a relaxation parameter $\alpha > 0$ to the updating of Jacobi and Gauss-Seidel iterations:

$$x^{(k+1)} = (\mathrm{I} - \alpha \mathrm{M}^{-1}\mathrm{A})x^{(k)} + \alpha \mathrm{M}^{-1} b. \tag{2.7}$$

where α is the relaxation parameter, $\alpha = 1$ corresponds to the standard Jacobi and Gauss-Seidel iterations.

Iteration Matrix

From the iterative scheme in Eq. (2.7), let $x^\star$ be the true solution such that $\mathrm{A}x^\star = b$, and error $e^{(k)} = x^{(k)} - x^\star$, then we have:

$$\begin{aligned} x^{(k+1)} - x^\star &= \left(I - \alpha \mathrm{M}^{-1}\mathrm{A}\right) x^{(k)} + \alpha \mathrm{M}^{-1} b - x^\star \\ &= \left(I - \alpha \mathrm{M}^{-1}\mathrm{A}\right) (x^{(k)} - x^\star) \\ &= \left(I - \alpha \mathrm{M}^{-1}\mathrm{A}\right) e^{(k)}. \end{aligned}$$

The iteration matrix $\mathrm{T} = I - \alpha \mathrm{M}^{-1}\mathrm{A}$ directly governs the convergence behavior of Jacobi and Gauss-Seidel methods.

Pseudocode

Algorithm 2 Jacobi / Gauss-Seidel Solver with Relaxation [24]

Require: Matrix $\mathrm{A} \in \mathbb{R}^{n \times n}$, vector $b \in \mathbb{R}^n$, initial guess $x_0 \in \mathbb{R}^n$, relaxation coefficient $\alpha > 0$, tolerance ϵ
Ensure: Approximate solution x_k
1: Decompose Matrix $A = \mathrm{M} - \mathrm{N}$
2: **for** $k = 0, 1, ..., n_{\max}$ **do**
3: Compute residual: $r_k = b - \mathrm{A}x_k$
4: **if** $\|r_k\| \leq \epsilon$ **then**
5: **return** x_k
6: **end if**
7: Compute update direction: $e_k = \mathrm{M}^{-1}(\mathrm{N}x_k + b) - x_k$
8: Relaxed update: $x_{k+1} = x_k + \alpha e_k$
9: **end for**

Spectral Analysis and Error Modes

Assume that the iteration matrix T has n eigenvalues λ_j with corresponding eigenvectors v_j, then we can express the initial error $e^{(0)} = x^{(0)} - x^\star$ in terms of the eigenvector basis such that

$$e^{(0)} = \sum_{j=1}^{n} \gamma_j v_j$$

where γ_j are coefficients. Then, the error after k iterations becomes:

$$e^{(k)} = (\mathrm{M}^{-1}\mathrm{N})^k e^{(0)} = \sum_{j=1}^{n} \gamma_j \lambda_j^k v_j \tag{2.8}$$

The convergence of the Jacobi and Gauss-Seidel methods depends on the magnitude of eigenvalues of the iteration matrix T. In particular, the Jacobi and Gauss-Seidel methods converge if and only if the spectral radius satisfies:

$$\rho(\mathrm{T}) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of the iteration matrix } \mathrm{T}\} < 1. \tag{2.9}$$

For many elliptic partial differential equations, such as the Poisson equation, $\rho(\mathrm{T}) < 1$ is satisfied in most cases, thus ensuring convergence. However, for indefinite problems, such as the Helmholtz equation, some eigenvalues of the iteration matrix may exceed 1, resulting in the divergence of Jacobi or Gauss-Seidel methods.

In addition, Eq. (2.8) shows that the error modes with different eigenvalues exhibit distinct convergence behaviors. To illustrate the convergence behaviors clearly, we consider the standard one-dimensional Poisson equation on $[0, 1]$ with homogeneous Dirichlet boundary conditions:

$$-u''(x) = f(x), \quad u(0) = u(1) = 0. \tag{2.10}$$

The linear system derived from finite difference methods with a uniform mesh is shown below:

$$\mathrm{A}u = \mathrm{f}, \quad \text{where} \quad \mathrm{A} = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & 0 \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ 0 & & & -1 & 2 \end{bmatrix}_{n\times n}, \quad \mathrm{f} = \begin{bmatrix} f(h) \\ f(2h) \\ \vdots \\ f(nh-h) \\ f(nh) \end{bmatrix}_{n} \tag{2.11}$$

The eigenvalues and eigenvectors of A are given by [7]:

$$\lambda_k = \frac{2}{h^2}\left(1 - \cos\frac{k\pi}{n+1}\right), \quad (v_k)_i = \sin(\frac{ik\pi}{n+1}), \quad i, k = 1, 2, \dots, n. \tag{2.12}$$

where $(v_k)_i$ denotes the ith component of the eigenvector v_k, and k labels the eigenmodes. Eigenmodes with a small k correspond to low-frequency, smooth waves, while large values of k correspond to highly oscillatory waves.

For the Jacobi method with $M = D = \mathrm{diag}(A) = \frac{2}{h^2}I$ and relaxation coefficient $\alpha = 1$, the Jacobi iteration matrix T^J is:

$$T^J = I - D^{-1}A. \tag{2.13}$$

$D^{-1}A$ shares the same eigenvectors v_k of A and $D^{-1}A\,v_k = (1 - \cos\frac{k\pi}{n+1})\,v_k$, the eigenvectors and eigenvalues of T_J are

$$\lambda_k^J = 1 - (1 - \cos\frac{k\pi}{n+1}) = \cos\frac{k\pi}{n+1}, \quad (v_k^J)_i = \sin(\frac{ik\pi}{n+1}), \quad i, k = 1, 2, \dots, n. \tag{2.14}$$

For the Gauss–Seidel method with $M = L + D$ and relaxation coefficient $\alpha = 1$, the Gauss–Seidel iteration matrix T^{GS} is:

$$T^{GS} = I - (L + D)^{-1}A = -(L + D)^{-1}U. \tag{2.15}$$

The Gauss–Seidel iteration matrix has eigenvalues and eigenvectors [7]:

$$\lambda_k^{GS} = \cos^2\frac{k\pi}{n+1}, \quad (v_k^{GS})_i = \left[\cos\frac{k\pi}{n+1}\right]^i \sin\frac{ik\pi}{n+1}, \quad i, k = 1, 2, \dots, n. \tag{2.16}$$

Eigenvectors of T^{GS} do not coincide with the eigenvectors of A and T^J. Therefore, the eigenvalue λ_k^{GS} gives the convergence rate for the kth eigenvector of T^{GS} instead of the kth mode of A.

Fig. 2.1 demonstrates convergence behaviors of the Jacobi or Gauss-Seidel methods with three different pure eigenmode errors, using eigenvectors $v^{(k)},\ k \in \{1, 3, 6\}$ of the corresponding iteration matrix T.

Generally, the convergence rate of the Jacobi and Gauss-Seidel methods is highly related to the frequency of the error components. Low-frequency and smooth errors typically show slow convergence as their corresponding eigenvalues are close to one, whereas high-frequency and oscillatory errors usually can be effectively reduced due to their small associated eigenvalues in most cases. For indefinite problems (e.g., the Helmholtz equation), some low-frequency components may even be amplified when their corresponding eigenvalue $|\lambda_k| \geq 1$, resulting in instability or divergence [6].

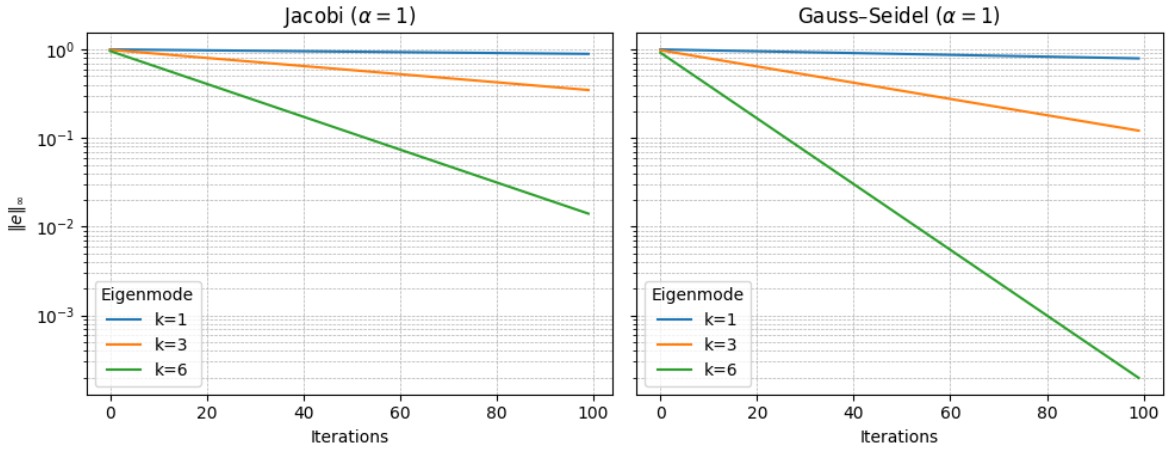


Figure 2.1: Convergence behavior of different eigenmodes using Gauss–Seidel and Jacobi methods ($n = 64, \alpha = 1$), where low-frequency eigenmodes (e.g., $k = 1$) decay slower than cases with higher frequency (e.g., $k = 6$) and Gauss–Seidel exhibits faster decay than Jacobi.

2.3. Multigrid Method

The multigrid method [7] is an efficient numerical algorithm extensively applied for solving large-scale linear systems. Stationary iterative methods such as the Jacobi or Gauss-Seidel methods effectively eliminate high-frequency components of errors but suffer from inefficiency in handling low-frequency components, resulting in a slow overall convergence rate. To address this inefficiency, the multigrid method utilizes a hierarchy of grids to accelerate the convergence.

In 1961, Federenko [8] proposed the first complete multigrid method by introducing an auxiliary grid to annihilate the low-frequency components of the error. The remaining low-frequency error components can be effectively eliminated on a coarse grid where they appear as higher-frequency components. This strategy enables efficient error elimination across all frequency components, resulting in fast convergence.

A classical implementation of the multigrid method is the V-cycle multigrid algorithm, including the following key procedures:

- Pre-smoothing: Applying classical iterative methods such as Jacobi or Gauss-Seidel method as a smoother on the current grid to reduce high-frequency errors.
- Residual computation: Computing the residual after the pre-smoothing.
- Restriction: Interpolating the residual from the fine grid to a coarser grid.
- Coarse-grid correction: Solving the error equation on the coarser grid to compute a correction term.
- Prolongation: Interpolating the coarse-grid correction from the coarse grid back to the finer grid and updating the solution.
- Post-smoothing: Performing additional smoothing on the finer grid to eliminate high-frequency errors again.

Algorithm 3 details the implementation of a V-cycle multigrid solver with Jacobi as the smoother for the linear system $\mathrm{A}^h\mathrm{u}^h = \mathrm{f}^h$:

Algorithm 3 Multigrid V-Cycle Solver with a Jacobi Smoother

Require: Matrix A^h, right-hand side f^h, number of iterations n_{max}, pre-smoothing steps n_{pre}, post-smoothing steps n_{post}, and relaxation coefficient α
Ensure: Approximate solution v^h
Initialize: $\mathrm{v}^h \leftarrow 0$
for $k = 1$ to n_{max} **do**
  $\mathrm{v}^h \leftarrow \text{V\_Cycle}(\mathrm{A}^h, \mathrm{f}^h, \mathrm{v}^h, n_{\text{pre}}, n_{\text{post}}, \alpha)$
  if convergence criterion is met **then**
    return v^h
  end if
end for
return v^h
function V_Cycle(A^h, f^h, v^h, n_{pre}, n_{post}, α)
  Pre-smoothing: $\mathrm{v}^h \leftarrow \mathrm{Jacobi}(\mathrm{A}^h, \mathrm{f}^h, \mathrm{v}^h, n_{\text{pre}}, \alpha)$
  if A^h is not on the coarsest grid **then**
    Residual computation: $\mathrm{r}^h \leftarrow \mathrm{f}^h - \mathrm{A}^h\mathrm{v}^h$
    Restriction: $\mathrm{r}^{2h} \leftarrow R\mathrm{r}^h,\ \mathrm{A}^{2h} \leftarrow R\mathrm{A}^h P$
    Recursive solve: $\mathrm{e}^{2h} \leftarrow \mathrm{V\_Cycle}(\mathrm{A}^{2h}, \mathrm{r}^{2h}, 0, n_{\text{pre}}, n_{\text{post}}, \alpha)$
    Prolongation: $\mathrm{e}^h \leftarrow P\mathrm{e}^{2h}$
    Solution Update: $\mathrm{v}^h \leftarrow \mathrm{v}^h + \mathrm{e}^h$
  else
    Direct solve: $\mathrm{A}^h\mathrm{e}^h = \mathrm{r}^h$
  end if
  Post-smoothing: $\mathrm{v}^h \leftarrow \mathrm{Jacobi}(\mathrm{A}^h, \mathrm{f}^h, \mathrm{v}^h, n_{\text{post}}, \alpha)$
  return v^h
end function

Here Jacobi(A, f, v, n, α) denotes n applications of the Jacobi algorithm defined in Algorithm 2

$$v = v + \alpha\, D^{-1}(f - Av), \quad D = \mathrm{diag}(A),$$

$R : \Omega_h \to \Omega_{2h}$ and $P : \Omega_{2h} \to \Omega_h$ denote the restriction and prolongation operators.

In addition to serving as a standalone solver for linear systems, the multigrid method is often used as a preconditioner in Krylov subspace methods, further enhancing overall convergence and efficiency.

2.4. Krylov Subspace Methods

Krylov subspace methods [5] form an important class of iterative methods for solving the linear system $\mathrm{A}x = b$, which rely on projections onto Krylov subspaces of A. Considering the Richardson iteration with a fixed step size α, as defined in Eq. (2.4):

$$\begin{aligned} r^{(k)} &= (\mathrm{I} - \alpha\mathrm{A})r^{(k-1)} \\ &= (\mathrm{I} - \alpha\mathrm{A})^k r^{(0)} \\ &= P_k(\mathrm{A})r^{(0)} \end{aligned}$$

where $r^{(0)} = b - \mathrm{A}x_0$ is the initial residual, and $P_k(\mathrm{A}) = (\mathrm{I} - \alpha\mathrm{A})^k$ is a matrix polynomial of degree k. Starting from an initial guess $x^{(0)}$, the Richardson iteration yields:

$$\begin{aligned} x^{(k)} &= x^{(0)} + \alpha(r^{(0)} + r^{(1)} + \cdots + r^{(k-1)}) \\ &= x^{(0)} + \alpha \sum_{j=0}^{k-1} P_j(\mathrm{A})r^{(0)} \\ &= x^{(0)} + v \end{aligned} \tag{2.17}$$

where $v = \alpha \sum_{j=0}^{k-1} P_j(\mathrm{A})r^{(0)} \in \mathrm{span}\{r^{(0)}, \mathrm{A}r^{(0)}, \cdots, \mathrm{A}^{k-1}r^{(0)}\}$. This k-dimensional space spanned by $\mathrm{A}^i r^{(0)}\ (0 \le i \le k-1)$ is known as the k-dimensional Krylov subspace of A with respect to $r^{(0)}$, denoted

by:

$$\mathcal{K}_k(\mathrm{A}, r^{(0)}) = \mathrm{span}\{r^{(0)}, \mathrm{A}r^{(0)}, \dots, \mathrm{A}^{k-1}r^{(0)}\}.$$

Thus, the approximate solution in Eq. (2.17) lies in the following affine space:

$$x^{(k)} \in x^{(0)} + \mathcal{K}_k(\mathrm{A}, r^{(0)}).$$

In the Richardson iteration, the polynomial $P_k(\mathrm{A})$ is fixed by the form $P_k(\mathrm{A}) = (\mathrm{I} - \alpha\mathrm{A})^k$ and depends only on the choice of α. This fixed polynomial in Richardson iterations limits its efficiency in reducing residuals, often resulting in slow convergence. In contrast, the Krylov subspace methods seek approximations $x^{(k)} \in x^{(0)} + \mathcal{K}_k(\mathrm{A}, r^{(0)})$ by selecting the update vector according to specific optimality principles such as residual or error norms, thereby significantly accelerating convergence.

2.4.1. Conjugate Gradient Method (CG)

The conjugate gradient (CG) method [10] applies when the matrix A is symmetric positive definite, which seeks approximate solutions in the affine Krylov space

$$x^{(k)} = x^{(0)} + \mathcal{K}_k(\mathrm{A}, r^{(0)}), \quad r^{(0)} = b - \mathrm{A}x^{(0)}$$

.

At each iteration, the approximate solution is chosen so that the residual is A-orthogonal to the current Krylov subspace

$$r^{(k)} \perp_A \mathcal{K}_k(\mathrm{A}, r^{(0)}),$$

which is equivalent to the following optimality condition, minimizing the A-norm of the error over the affine Krylov space:

$$\|x^\star - x^{(k)}\|_{\mathrm{A}} = \min_{z \in x_0 + \mathcal{K}_k(\mathbf{A}, r^{(0)})} \|x - z\|_{\mathrm{A}}, \tag{2.18}$$

where $x^{(k)}$ is the approximate solution at step k, $x^\star$ is the exact solution. The pseudocode for the conjugate gradient method is listed in Algorithm 4.

Algorithm 4 Conjugate Gradient Method (CG)

Require: SPD matrix $\mathrm{A} \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$, initial guess $x_0 \in \mathbb{R}^n$, tolerance ϵ, maximum iterations $n_{\max}$
Ensure: Approximate solution x_k

Set $r^{(0)} = b - \mathrm{A}x^{(0)}, \quad p_0 = r^{(0)}$
for $k = 0, 1, \dots, n_{\max} - 1$ **do**
  $\alpha_k = \dfrac{\langle r^{(k)},\, r^{(k)}\rangle}{\langle p_k,\, \mathrm{A}\, p_k\rangle}$
  $x^{(k+1)} = x^{(k)} + \alpha_k\, p_k$
  $r^{(k+1)} = r^{(k)} - \alpha_k\, \mathrm{A}\, p_k$
  if $\|r^{(k+1)}\| \le \epsilon$ **then**
    return $x^{(k+1)}$
  end if
  $\omega_k = \dfrac{\langle r^{(k+1)},\, r^{(k+1)}\rangle}{\langle r^{(k)},\, r^{(k)}\rangle}$
  $p_{k+1} = r^{(k+1)} + \omega_k\, p_k$
end for
return $x^{(n_{\max})}$

2.4.2. Generalized Minimal Residual Method (GMRES)

The Generalized Minimal Residual Method (GMRES) [9] is applicable when the matrix $A \in \mathbb{R}^{n\times n}$ is nonsingular. Let the search space $\mathcal{S}_k = \mathcal{K}_k(\mathrm{A}, r^{(0)})$, the constraint space $\mathcal{C}_k = \mathrm{A}\mathcal{S}_k$, if $r^{(0)} = b - \mathrm{A}x_0$ is of grade $d \ge 1$ with respect to the matrix A, then the GMRES method is well-defined at every step k until convergence at step d with $r_d = 0$. Algorithm 5 shows the pseudocode for the GMRES method [9].

The GMRES method seeks an approximate solution $x^{(k)}$ by minimizing the ℓ_2-norm of the residual over the affine space $x_0 + \mathcal{K}_k(\mathrm{A}, r^{(0)})$:

$$\|r^{(k)}\|_2 = \min_{z \in x_0 + \mathcal{K}_k(A, r^{(0)})} \|b - \mathrm{A}z\|_2 \tag{2.19}$$

where $r^{(k)} = b - \mathrm{A}x^{(k)}$.

This method is particularly suitable for solving general nonsingular linear systems, where the conjugate gradient method is not applicable.

Algorithm 5 Generalized Minimal Residual Method (GMRES)

Require: Nonsingular matrix $\mathrm{A} \in \mathbb{R}^{n \times n}$, vector $b \in \mathbb{R}^n$, initial guess $x^{(0)} \in \mathbb{R}^n$, tolerance ϵ, maximum iterations $n_{\max}$

Ensure: Approximate solution $x^{(k)}$

Set $r^{(0)} = b - \mathrm{A}\, x^{(0)}$

for $k = 0, 1, \ldots, n_{\max} - 1$ **do**

  Perform the kth step of Arnoldi to generate V_k and $\mathrm{H}_{k+1,k}$ such that

$$\mathrm{A}\,\mathrm{V}_k \;=\; \mathrm{V}_{k+1}\,\mathrm{H}_{k+1,k}$$

  Perform QR decomposition of the Hessenberg matrix $H_{k+1,k}$: $\mathrm{H}_{k+1,k} \;=\; \mathrm{Q}_{k+1}\,\mathrm{R}_{k+1,k}$

  Compute the residual norm

$$\|r^{(k)}\|_2 = \|r^{(0)}\|_2 (\mathrm{Q}_{k+1}^T e_1)_{k+1},$$

  if $\|r^{(k+1)}\| \leq \epsilon$ **then**

    Compute the vector with pseudoinverse of $\mathrm{H}_{k+1,k}$

$$t_k = H_{k+1,k}^{+}(\|r_0\|_2 e_1)$$

    Return the approximate solution $x^{(k+1)} = x^{(0)} + V_k\, t^{(k)}$

  end if

end for

return $x^{(n_{\max})}$

3

Introduction to Deep Operator Networks

Over the past decade, deep learning techniques have had a revolutionary impact on various disciplines in science and engineering, including bioengineering [25], computer vision [26], natural language processing [27]. In recent advances in scientific computing, neural network-based approaches for solving partial differential equations (PDEs) have attracted significant interest, such as physics-informed neural networks (PINNs) [13], neural operators [14, 15], and their variants. Among these algorithms, the Deep Operator Network (DeepONet) [14], proposed to learn solution operators of PDEs directly, has emerged as an efficient solver with strong generalization capabilities for parametric PDE problems.

The goal of this chapter is to provide a comprehensive overview of the theoretical background, structure, and implementation of neural networks and DeepONet, with an emphasis on the application of DeepONet to PDEs.

3.1. Neural Networks

Neural networks, inspired by the biological nervous system in the brain, are composed of interconnected neurons to learn complex nonlinear mappings between inputs and outputs. Traditional artificial neural networks (ANNs) [28] have been extensively utilized across various fields for tasks such as classification and regression.

3.1.1. Structure

The back-propagation (BP) neural network [29], first proposed by Rumelhart, Hinton, and Williams in 1986, remains one of the most fundamental and extensively adopted neural network architectures. A typical BP neural network consists of the following components:

- An input layer: receives the input data.
- One or more hidden layers: perform nonlinear transformations on the input data.
- An output layer: produces the final output.

As shown in Fig. 3.1, each layer in the network consists of multiple neurons, with each neuron fully connected to all neurons from the adjacent layers. A typical neuron in the hidden layer or output layer computes a weighted sum of outputs of neurons in the previous layer, and then applies a nonlinear activation function. The output $a_j^{(l)}$ of neuron j in layer l is given by:

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right), \quad \text{where } z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}, \tag{3.1}$$

where $a_i^{(l-1)}$ denotes the output of neuron i in the previous layer $l-1$, $w_{ji}^{(l)}$ is the trainable weight connecting neuron i in layer $l-1$ and neuron j in layer l, $b_j^{(l)}$ is the bias term in the current layer l, $z_j^{(l)}$ is the weighted sum of outputs from the previous layer $l-1$, and $\sigma(\cdot)$ is the nonlinear activation function.

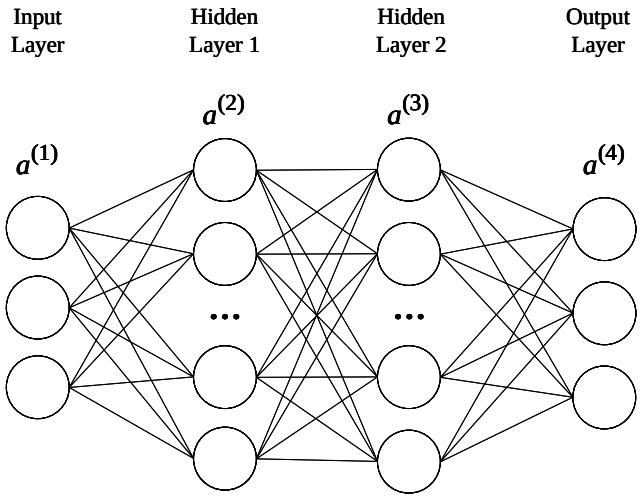
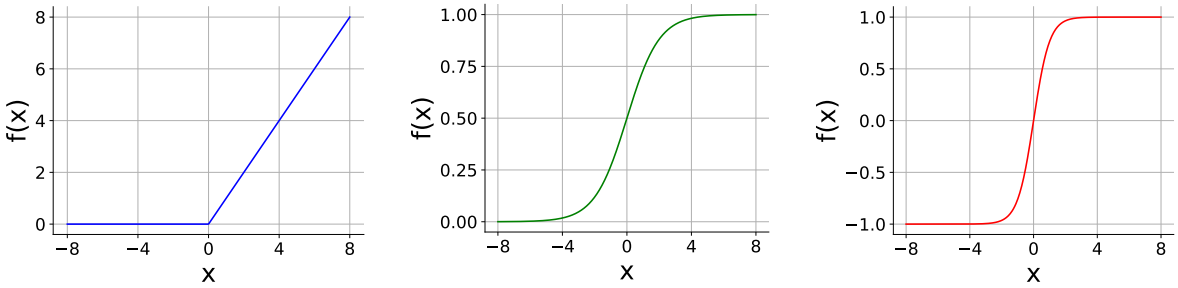


Figure 3.1: Structure of a typical artificial neural network

The nonlinear activation functions introduce nonlinearity to neural networks, preventing them from becoming naive linear models and enabling them to learn complex nonlinear relationships between inputs and outputs. Common nonlinear activation functions are illustrated in Fig. 3.2:



(a) ReLU $f(x) = \max(0, x)$ (b) Sigmoid $f(x) = \frac{1}{1+e^{-x}}$ (c) Tanh $f(x) = \tanh(x)$

Figure 3.2: Common nonlinear activation functions

3.1.2. Training

As shown in Eq. (3.1), a large number of trainable parameters, such as weights and biases of each layer, exist in a deep neural network. The process of iteratively adjusting these parameters to minimize the difference between the network predictions and the target outputs is called training.

In the initial stage of training, the parameters of a neural network are usually initialized by sampling from specific distributions, such as the common Xavier or Kaiming initialization strategies [30, 31]. After initialization, the neural network computes predictions through forward propagation. Then, the difference between the neural network output and the target output is quantified by a loss function. The choice of loss function depends on the specific task of the neural network.

In the context of solving PDEs with neural networks, two main categories of training losses are commonly used:

Data-driven loss Data-driven loss quantifies the discrepancy between the predicted solutions and the provided target values directly. A common example of data-driven loss in solution approximation is the mean squared error (MSE), defined as follows:

$$L_{\text{data}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \|u(x_i) - u_\theta(x_i)\|^2, \tag{3.2}$$

where $u(x_i)$ represents the target solution at x_i, $u_\theta(x_i)$ denotes the predicted solution of the neural network, and n is the number of samples.

Physics-informed loss Physics-informed loss leverages physical laws to measure the accuracy of the predicted solution by including partial differential equations (PDEs) as penalty terms.

Consider a general PDE of the form:

$$\mathcal{N}[u](x) = f(x), x \in \Omega,$$

where $\mathcal{N}$ represents the differential operator applied to the unknown solution $u(x)$. The physics-informed loss is typically defined as the MSE of the PDE residual over a set of collocation points $\{x_i\}_{i=1}^{n_f}$ in the PDE domain Ω:

$$L_{\text{PDE}}(\theta) = \frac{1}{n_f} \sum_{i=1}^{n_f} \left| \mathcal{N}[u_\theta](x_i) - f(x_i) \right|^2, \tag{3.3}$$

where u_θ is the predicted solution of the neural network parameterized by trainable parameters θ.

In many applications of physics-informed machine learning, supervised training data for solving PDEs are available [32, 12], which can be obtained from initial and boundary conditions, experimental measurements, or high-fidelity numerical simulations. Therefore, in many deep learning-based PDE solvers such as PINNs [12, 13], a weighted sum of physics-informed loss and data-driven loss can be adopted to balance data fidelity and physical consistency.

$$L(\theta) = w_{\text{data}} L_{\text{data}}(\theta) + w_{\text{PDE}} L_{\text{PDE}}(\theta) \tag{3.4}$$

Once the loss is computed, the network parameters θ (weights and biases) can be updated iteratively based on the gradients of the loss function with respect to themselves. This process, known as backpropagation [29], works in contrast to the forward propagation, such that forward propagation gives predictions by passing data from the input layer to the output layer, while backpropagation computes gradients inversely from the output layer to the input layer by the chain rule.

Optimization strategies such as gradient descent or its variants (e.g., Adam optimizer [33]), are widely used to update parameters in neural networks. A typical gradient descent update step for the network parameter θ is defined as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta L(\theta^{(t)}) \tag{3.5}$$

where η is known as the learning rate that controls the step size in each update, and $\nabla_\theta L(\theta^{(t)})$ represents the gradient of the total loss. The above process continues iteratively until some predefined convergence criteria are satisfied, indicating that the network has sufficiently learned the underlying patterns from the training data and physical constraints.

3.1.3. Universal Approximation Theorem

The theoretical underpinnings of neural networks largely rely on the mathematical guarantee provided by the universal approximation theorem, which states that a neural network with sufficient network complexity can approximate arbitrary continuous functions to a desired accuracy.

Theorem 3.1 (Universal Approximation Theorem [34, 35, 36])**.** *Let $C(X, \mathbb{R}^m)$ denote the space of continuous functions from a non-empty compact subset $X \subseteq \mathbb{R}^n$ to $\mathbb{R}^m$. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous non-polynomial activation function, applied component-wise. Then, for any function $f \in C(X, \mathbb{R}^m)$ and every $\varepsilon > 0$, there exist an integer k, a weight matrix $A \in \mathbb{R}^{k \times n}$, a bias vector $b \in \mathbb{R}^k$, and an output matrix $C \in \mathbb{R}^{m \times k}$ such that the neural network approximation*

$$g(x) = C\left(\sigma(Ax + b)\right)$$

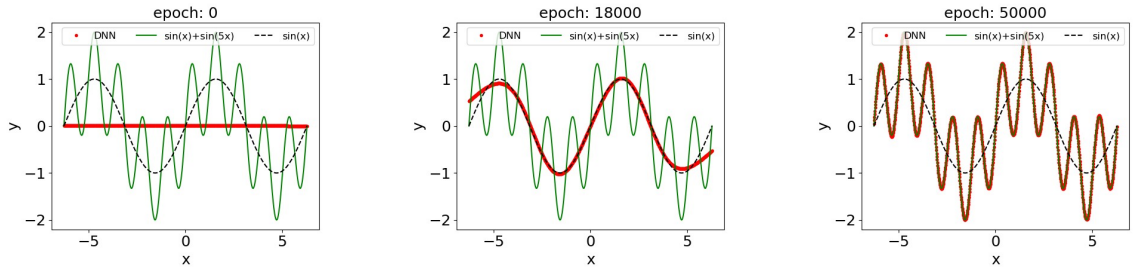
satisfies

$$\sup_{x \in X} \|f(x) - g(x)\| < \varepsilon.$$

The universal approximation theorem guarantees that, with a continuous, non-polynomial activation function and enough hidden units, a neural network can approximate any continuous function on a compact domain arbitrarily well. Therefore, neural networks also have the potential to approximate continuous PDE solutions, and thereby solve PDEs.

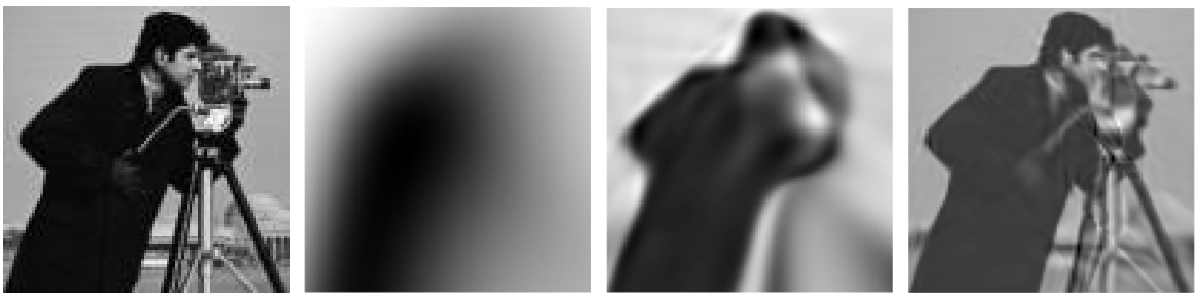
3.1.4. Spectral Bias

Spectral bias (or frequency principle) [18, 16, 17, 19] is a significant phenomenon observed during the training of deep neural networks; that is, neural networks preferentially fit target functions from low-frequency components to high frequencies. Xu et al. [16] demonstrated this phenomenon by approximating one-dimensional mixed frequency functions and image inpainting tasks in the review of spectral bias, as shown in Figs. 3.3 and 3.4. Their results suggest that neural networks effectively capture the smoother (low-frequency) component of the target function, but struggle to learn oscillatory (high-frequency) structures.



(a) Epoch 0 (b) Epoch 18000 (c) Epoch 50000

Figure 3.3: Spectral bias in 1D function approximation taken from Xu et al. [16]



(a) True (b) Step 80 (c) Step 2000 (d) Step 58000

Figure 3.4: Spectral bias in 2D image inpainting taken from Xu et al. [16]

It should be noted that the concept of ”frequency” in spectral bias does not simply refer to the spatial frequency of the input images or data itself, but the response frequency of the functional mapping from input to output; that is, whether a small change in the input will induce a large change in the output [16]. For function approximation tasks, this preference for low-frequency functions suggests that deep neural networks have better generalization capabilities for smooth functions, but may be inefficient in approximating high-frequency solutions.

Spectral bias also imposes high-frequency challenges on neural network-based methods for solving partial differential equations (PDEs). Sifan Wang et al. [17] and Zhang et al. [19] revealed that physics-informed neural networks (PINNs) and deep neural networks (DeepONet) suffer from spectral bias; that is, these methods for solving PDEs often demonstrate slow convergence to the high-frequency components of the target function. To enhance the learning of those PDE solutions containing high-frequency oscillations, special neural structures or training strategies usually need to be considered [17, 18].

3.2. Deep Operator Networks

PINNs approximate the PDE solution by incorporating the governing differential equations into the training loss. However, PINNs are typically tailored to specific physical systems. Solving a new PDE problem usually requires retraining or a new network architecture, which limits the flexibility and generalization of the model.

Unlike PINNs, Lu et al. [14] proposed the Deep Operator Network (DeepONet) based on the universal approximation theorem for operators. The DeepONet directly learns the nonlinear operator map-

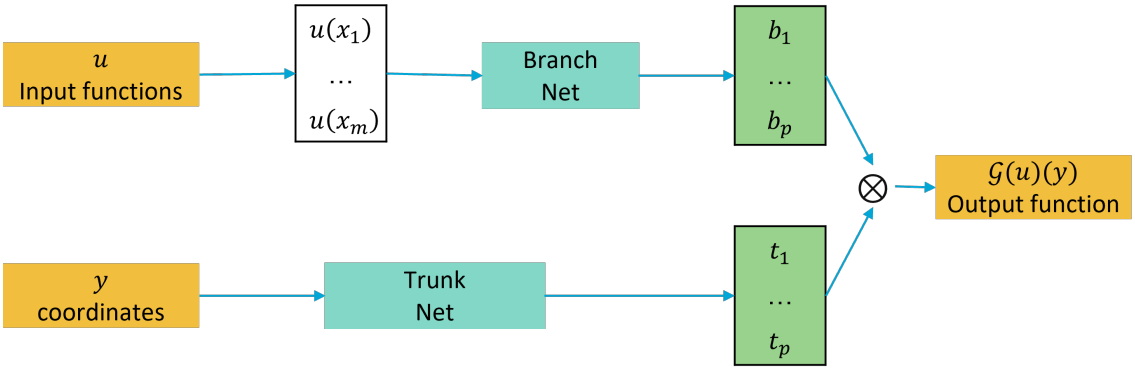


Figure 3.5: Architecture of a DeepONet approximating the nonlinear operator $\mathcal{G} : u \mapsto \mathcal{G}(u)$

ping from the input function space to the corresponding PDE solution space, instead of approximating a single PDE solution.

A trained DeepONet establishes a new paradigm for fast solvers of parametric PDEs. This section provides an overview of the DeepONet, which learns a nonlinear operator

$$\mathcal{G} : u \mapsto \mathcal{G}(u), \qquad \mathcal{G}(u)(y) \in \mathbb{R}^k, \tag{3.6}$$

where u is the input function taken by the operator $\mathcal{G}$, $\mathcal{G}(u)$ is the corresponding output function. For any point y in the domain of the output function $\mathcal{G}(u)$, $\mathcal{G}(u)(y) \in \mathbb{R}^k$ represents the value of the output function $\mathcal{G}(u)$ evaluated at y.

3.2.1. Universal Approximation Theorem for Operators

DeepONet relies on the operator approximation theorem proposed by Chen and Chen [37], which indicates that a neural network with a single hidden layer can accurately approximate both continuous nonlinear functional mappings from a function space to scalar values and complex operator mappings between distinct function spaces. This theorem extends the approximation capabilities of neural networks from functions to operators.

Theorem 3.2 (Universal Approximation Theorem for Operators [14, 37]). *Suppose that σ is a continuous non-polynomial function, X is a Banach space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively, V is a compact set in $C(K_1)$, $\mathcal{G}$ is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\varepsilon > 0$, there exist positive integers n, p, m and constants $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$, $i = 1, \cdots, n$, $k = 1, \cdots, p$, and $j = 1, \cdots, m$, such that*

$$\left| \mathcal{G}(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{\text{branch}} \underbrace{\sigma(w_k \cdot y + \zeta_k)}_{\text{trunk}} \right| < \varepsilon \tag{3.7}$$

holds for all $u \in V$ and $y \in K_2$. Here, $C(K)$ is the Banach space of all continuous functions defined on K with norm $\|f\|_{C(K)} = \max_{x \in K} |f(x)|$.

3.2.2. Network Architecture of DeepONet

Motivated by the universal approximation theorem for operators, a typical DeepONet architecture consists of two distinct subnetworks, the branch network and the trunk network. Fig. 3.5 illustrates the network structure of the DeepONet approach.

For solving PDEs, the DeepONet learns the solution operator $\mathcal{G}$ from two inputs: the PDE input function $u(x)$ and the spatial coordinate $y \in \mathbb{R}^d$ at which the solution $\mathcal{G}(u)(y)$ is evaluated. Specifically, the input function $u(x)$ is represented discretely by evaluating $u(x)$ at a sufficient but finite number of fixed sensor points $x_1, x_2, \cdots, x_m$. For every function $u(x)$ in the training dataset, all input functions $u(x)$ should be evaluated at the same sensor points. However, for the evaluation coordinates y of the PDE

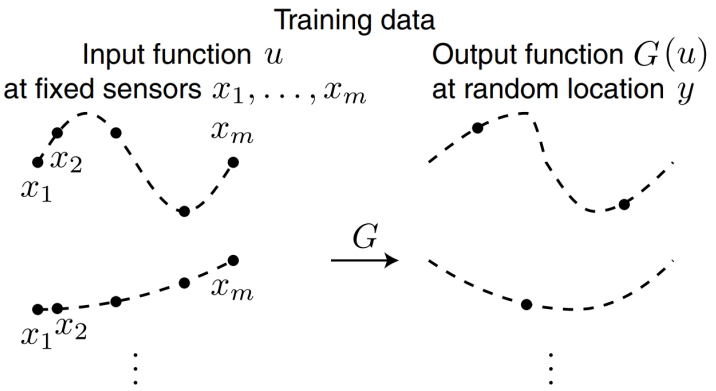


Figure 3.6: Illustration of training data in DeepONet taken from Lu et al. [14]

solutions across the training dataset, no constraints are imposed. Fig. 3.6 illustrates the sensors points and evaluation coordinates across the training samples.

The branch network outputs a vector $b(u) = [b_1(u), b_2(u), \cdots, b_p(u)] \in \mathbb{R}^p$, and the trunk network generates another vector $t(y) = [t_1(y), t_2(y), \ldots, t_p(y)] \in \mathbb{R}^p$. The predicted PDE solution at y is then given by the inner product:

$$\mathcal{G}(u)(y) \approx \sum_{k=1}^{p} b_k(u) t_k(y) \tag{3.8}$$

DeepONet is flexible regarding network types; various classical neural network architectures, such as fully connected neural networks (FCNs), recurrent neural networks (RNNs), convolutional neural networks (CNNs), or residual networks (ResNets), can be applied in the branch network and the trunk network. To improve generalization performance, DeepONet usually selects deep networks in practice, and global bias terms can also be introduced in the final inner product:

$$\mathcal{G}(u)(y) \approx \sum_{k=1}^{p} b_k(u) t_k(y) + b_0 \tag{3.9}$$

These designs ensure that the DeepONet is capable of learning nonlinear operators, thereby achieving high accuracy in solving a broad class of PDEs.

4

Introduction to HINTS

As discussed in Section 2.2.2, classical iterative solvers such as Jacobi and Gauss-Seidel methods are effective in eliminating high-frequency components of errors, but converge slowly or even diverge for low-frequency error modes. On the other hand, as discussed in Section 3.1.4, deep neural networks, including deep learning-based approaches for solving PDEs, suffer from the well-known spectral bias [16]: they tend to be proficient at learning low-frequency components but struggle to learn high-frequency and oscillatory components.

These different preferences across the error spectrum motivate the integration of deep learning-based solvers with classical iterative methods for PDEs [19, 38, 39, 40]. By combining their complementary strengths, the hybrid solvers have the potential to achieve both accuracy and efficiency across various error modes.

Among these approaches [19, 38, 39, 40], the Hybrid Iterative Numerical Transferable Solver (HINTS), proposed by Zhang et al. [19], blends classical relaxation methods such as Jacobi and Gauss-Seidel methods with the Deep Operator Network (DeepONet). By utilizing a well-trained DeepONet as an offline model, HINTS significantly improves the performance of classical iterative methods across a broad class of PDEs. This hybrid framework not only accelerates the convergence performance of classical iterative solvers in solving linear PDEs, but also maintains its effectiveness in indefinite problems where traditional methods might fail.

In this chapter, we first present the HINTS framework, detailing its integration with classical relaxation methods. Subsequently, we introduce an integration of HINTS with multigrid methods (HINTS-MG). We then validate the performance of these hybrid methods through numerical experiments conducted on the same representative PDEs considered by Zhang et al. [19].

4.1. Benchmark Problem Setup

For validation purposes, we adopt the same boundary value problems taken from Zhang et al. [19], which serve as benchmark problems throughout this chapter:

- Poisson equation:

$$\begin{aligned} -\nabla \cdot (k(x)\nabla u(x)) &= f(x), \quad x \in (0,1) \\ u(0) = u(1) &= 0 \end{aligned} \tag{4.1}$$

- Helmholtz equation:

$$\begin{aligned} \nabla^2 u(x) + k^2(x)u(x) &= f(x), \quad x \in (0,1) \\ u(0) = u(1) &= 0 \end{aligned} \tag{4.2}$$

4.2. The General Framework of HINTS

Consider the linear system from the FDM discretization of a linear PDE problem given in Eqs. (4.1) and (4.2), defined as:

$$\mathrm{A}^h \mathbf{u}^h = \mathrm{f}^h, \tag{4.3}$$

where u^h denotes the discrete solution on the grid. Let $\mathrm{v}^{h,(k)}$ represent the approximation at the kth iteration, and the corresponding residual is given by:

$$\mathrm{r}^{h,(k)} = \mathrm{f}^h - \mathrm{A}^h \mathrm{v}^{h,(k)} \tag{4.4}$$

The core idea of the HINTS approach is to rapidly eliminate high-frequency components with Jacobi and Gauss-Seidel methods and reduce low-frequency components with a pre-trained DeepONet model. At each step k, the HINTS approach updates the current approximation $\mathrm{v}^{h,(k)}$ as follows:

$$\mathrm{v}^{h,(k+1)} = \mathrm{v}^{h,(k)} + \Delta \mathrm{v}^{h,(k)} \tag{4.5}$$

where the correction term $\Delta \mathrm{v}^{h,(k)}$ alternates between stationary iterative methods (e.g., the Jacobi or Gauss-Seidel method) and DeepONet predictions. Let n_r denote the iteration interval for the DeepONet model such that it is applied every n_r iterations:

$$\Delta \mathrm{v}^{h,(k)} = \begin{cases} \text{Stationary Iterative method}, & \text{if } k \mod n_r \neq 0, \\ \text{DeepONet Prediction}, & \text{if } k \mod n_r = 0, \end{cases} \tag{4.6}$$

For the Jacobi or Gauss-Seidel method, as shown in Algorithm 2, the correction $\Delta \mathrm{v}^{h,(k)}$ is obtained by solving a simplified linear system derived from matrix splitting:

$$\begin{aligned} \mathrm{M} e^{(k)} &= \mathrm{r}^{h,(k)} \\ \Delta \mathrm{v}^{h,(k)} &= \alpha e^{(k)} \end{aligned} \tag{4.7}$$

where M corresponds to the easily invertible matrix from the splitting and α is the relaxation parameter.

For the DeepONet correction, consider the case for the Poisson and Helmholtz equations, a DeepONet model should be trained offline in advance, taking the function values of $k(x)$ and $f(x)$ from Eq. (4.1) or (4.2) at predefined sensor points as input. During a HINTS iteration where a DeepONet model is applied, the discrete residual vector $\mathrm{r}^{h,(k)} = \mathrm{f}^h - \mathrm{A}^h \mathrm{v}^{h,(k)}$, defined on the FDM grid, is mapped to the predefined sensor points of the DeepONet model by linear interpolation, producing a vector of function values $r(x)$. The vector $r(x)$ represents values of the residual on sensor points. To keep the scale of the input consistent, the residual vector $r(x)$ needs to be normalized, defined as:

$$\tilde{r}(x) = \begin{cases} \dfrac{r(x)}{\|r(x)\|_\infty}, & \|r(x)\|_\infty > 0, \\ 0, & \|r(x)\|_\infty = 0, \end{cases}$$

where the normalized residual function values $\tilde{r}(x)$, along with the parameter function values $k(x)$, serve as the input to the pre-trained DeepONet model to predict the correction term $\Delta \mathrm{v}^{h,(k)}$:

$$\begin{aligned} r(x) &\leftarrow \mathrm{f}^h - \mathrm{A}^h \mathrm{v}^{h,(k)} \\ \tilde{r}(x) &\leftarrow r(x) / \|r(x)\|_\infty \\ \Delta \mathrm{v}^{h,(k)} &\leftarrow \mathrm{DeepONet}(k(x), \tilde{r}(x)) \cdot \|r(x)\|_\infty \end{aligned} \tag{4.8}$$

Algorithm 6 provides the detailed implementation of the HINTS framework.

Algorithm 6 Hybrid Iterative Numerical Transferable Solver (HINTS)

Require: Parameter function $k(x)$, source function $f(x)$, PDE to be solved, number of iterations n_{it}, pre-trained DeepONet model, frequency n_r, (optionally) relaxation coefficient α
Ensure: Approximate solution v^h

Discretize the PDE to be solved by FDM: $\mathrm{A}^h, \mathrm{f}^h \leftarrow k(x), f(x), \text{PDE}$
function HINTS(A^h, f^h, v^h, $k(x)$, n_{it}, n_r, α)
  for $i = 1$ to n_{it} **do**
    Compute residual: $\mathrm{r}^h = \mathrm{f}^h - \mathrm{A}^h v^h$
    if $i \bmod n_r = 0$ **then**
      Linear interpolation: $r(x) \leftarrow \mathrm{r}^h$
      Normalization: $\tilde{r}(x) \leftarrow r(x)/\|r(x)\|_\infty$
      Predict increment: $\Delta v^h \leftarrow \mathrm{DeepONet}(k(x), \tilde{r}(x)) \cdot \|r(x)\|_\infty$
      Update: $v^h \leftarrow v^h + \Delta v^h$
    else
      Matrix splitting: $\mathrm{A}^h = \mathrm{M} - \mathrm{N}$
      Solve: $\mathrm{M}\Delta v^h = \mathrm{r}^h$
      Relaxed update: $v^h \leftarrow v^h + \alpha \Delta v^h$
    end if
  end for
  return v^h
end function

4.3. Introduction to the HINTS-MG

Multigrid methods are one of the most extensively applied solvers in solving large-scale linear systems, which use a hierarchy of grids to achieve significantly faster convergence. However, multigrid methods often suffer from divergence when applied to indefinite problems such as the Helmholtz equation with a large wavenumber. One of the reasons lies in the traditional smoother (e.g., the Jacobi or Gauss-Seidel method) used in the multigrid method, which may amplify the low-frequency components of the error for indefinite problems [6]. Developing an efficient multigrid method for the Helmholtz equation remains an open and challenging problem [41, 6, 42, 43].

By integrating the HINTS approach into multigrid methods, a new hybrid multilevel solver, HINTS-MG, is developed by Zhang et al. [19] to improve the performance of the classical multigrid method.

4.3.1. Algorithm

Algorithm 7 presents the detailed implementation of HINTS-MG with V-cycle. Instead of the traditional Jacobi or Gauss-Seidel smoother, the HINTS approach is applied in both pre- and post-smoothing stages at each level of the grid hierarchy. To guarantee that DeepONet is applied at least once per smoothing phase, the number of pre- and post-smoothing iterations, n_{pre} and n_{post}, must both exceed the activation frequency of DeepONet in the HINTS approach n_r.

This hybrid smoothing strategy ensures the efficient elimination of both high- and low-frequency error modes across multiple levels of grid, which is particularly advantageous for indefinite problems compared with the classical multigrid method, where low-frequency error components might be amplified in some cases.

Algorithm 7 Multigrid HINTS Solver with V-cycle Smoothing

Require: Parameter function $k(x)$, source f^h, PDE at level h, number of outer iterations n_{cycle}, smoothing steps n_{pre}, n_{post}, DeepONet frequency $n_r < \min(n_{\text{pre}}, n_{\text{post}})$, relaxation factor α
Ensure: Approximate solution v^h

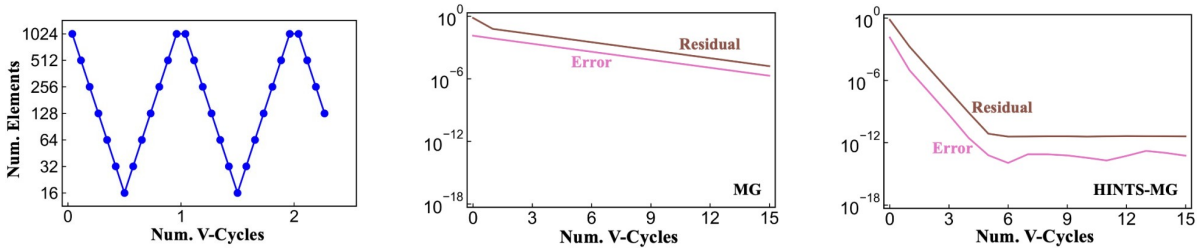
Discretize the PDE to be solved: $\mathrm{A}^h, \mathrm{f}^h \leftarrow \text{DISCRETIZE}(k(x), f(x), \text{PDE})$
Initialize: $v^h \leftarrow 0^h$
for $i = 1$ to n_{cycle} **do**
    $v^h \leftarrow \text{HINTS\_V\_CYCLE}(\mathrm{A}^h, \mathrm{f}^h, v^h, k(x), n_{\text{pre}}, n_{\text{post}}, n_r, \alpha)$
end for
return v^h

function HINTS_V_CYCLE(A^h, f^h, v^h, $k(x)$, n_{pre}, n_{post}, n_r, α)
    Pre-smoothing: $v^h \leftarrow \text{HINTS}(\mathrm{A}^h, \mathrm{f}^h, v^h, k(x), n_{\text{pre}}, n_r, \alpha)$
    if Ω^h is not the coarsest grid **then**
        Compute residual: $\mathrm{r}^h \leftarrow \mathrm{f}^h - \mathrm{A}^h v^h$
        Restrict: $\mathrm{r}^{2h},\ \mathrm{A}^{2h} \leftarrow \text{RESTRICTION}(\mathrm{r}^h),\ \text{RESTRICTION}(\mathrm{A}^h)$
        Recursive solve: $v^{2h} \leftarrow \text{HINTS\_V\_CYCLE}(\mathrm{A}^{2h}, \mathrm{r}^{2h}, 0^{2h}, k(x), n_{\text{pre}}, n_{\text{post}}, n_r, \alpha)$
        Correction: $v^h \leftarrow v^h + \text{PROLONGATION}(v^{2h})$
    end if
    Post-smoothing: $v^h \leftarrow \text{HINTS}(\mathrm{A}^h, \mathrm{f}^h, v^h, k(x), n_{\text{post}}, n_r, \alpha)$
    return v^h
end function

4.3.2. Coarsest-Grid Strategy

In the original implementation of the multigrid method and HINTS-MG by Zhang et al. [19], an uncommon strategy was applied on the coarsest grid: using only 10 iterations of the smoother to approximate the coarse-grid solution instead of solving the coarsest-grid problem exactly. This ”smoother-only” strategy might degrade the accuracy of the coarse-grid correction and could unfairly favor HINTS-MG in comparisons, as HINTS converges faster than the classical smoother on the coarsest level.

Fig. 4.1 presents the original results and the corresponding configuration from Zhang et al. [19] for the 1D Poisson problem. Although their findings show a clear performance gain for HINTS-MG, this advantage might be the result of the ”smoother-only” strategy instead of the HINTS smoother. To comprehensively evaluate the improvement of HINTS-MG over the MG method, the performance of both ”smoother-only” and ”direct solver” strategies on the coarsest grid will be investigated in the numerical experiments presented in Section 4.5.2.



(a) V-cycle configuration. Each blue dot represents 10 iterations of the smoother.

(b) Error and residual norms for MG with Gauss-Seidel relaxation.

(c) Error and residual norms for HINTS-MG with HINTS relaxation.

Figure 4.1: Configurations and results of HINTS-MG taken from Zhang et al. [19] on the 1D Poisson equation with $n = 1025$ grid points.

4.4. Training Procedure of DeepONet for HINTS

In the HINTS framework, the DeepONet is used to predict the low-frequency corrections in iterations, which should be trained offline in advance to its deployment in HINTS. To validate the performance of HINTS and HINTS-MG independently, we regenerated the training data and trained a DeepONet for the one-dimensional Poisson equation (Eq. (4.1)) and Helmholtz equation (Eq. (4.2)). The detailed

training process of DeepONet, including data generation, pre-processing, and network architecture, is presented in this section.

4.4.1. Data Generation

We use Gaussian Random Fields (GRFs) [44] to generate the parameter function $k(x)$ and the source function $f(x)$ in the training data and testing data, which are independently sampled from GRF realizations governed by:

$$\begin{aligned} k(x) &\sim \mathcal{GP}\left(k_0, \mathcal{K}_k(x_1, x_2)\right), \quad \mathcal{K}_k(x_1, x_2) = \sigma_k^2 \exp\left(-\frac{|x_1 - x_2|^2}{2l_k^2}\right), \\ f(x) &\sim \mathcal{GP}\left(0, \mathcal{K}_f(x_1, x_2)\right), \quad \mathcal{K}_f(x_1, x_2) = \sigma_f^2 \exp\left(-\frac{|x_1 - x_2|^2}{2l_f^2}\right), \end{aligned} \tag{4.9}$$

where each realization is discretized on a set of $n = 31$ uniformly distributed grid points in $[0, 1]$. To consider challenges in indefinite Helmholtz problems with a large wavenumber, a cutoff threshold k_{min} is imposed in the data generation, such that samples are retained only if they satisfy the condition $\min_x k(x) > k_{\text{min}}$. The configuration of k_0, k_{min}, σ_k, l_k, σ_f, l_f for Poisson and Helmholtz equations taken from Zhang et al. [19] is summarized in Table 4.1.

Table 4.1: Parameters for GRF-based data generation in DeepONet training taken from Zhang et al. [19].

Case	k_0	$k_{\min}$	σ_k	l_k	σ_f	l_f
1D Poisson	1.0	0.3	0.3	0.1	1.0	0.1
1D Helmholtz	8.0	3.0	2.0	0.2	1.0	0.1

To generate the target output $u(x)$ for supervised training, each PDE instance with generated $k(x)$ and $f(x)$ is numerically solved using the finite difference method (FDM) on the same uniform grid with spacing $h = 1/30$. The values of $k(x)$, $f(x)$, and $u(x)$ evaluated at these grid points $\{x_i\}_{i=1}^{31}$ are then used to train DeepONet to approximate the underlying solution operator.

4.4.2. Data Preprocessing for the Helmholtz Problem

Discretizing the Helmholtz equation with the finite-difference method (FDM) on a uniform grid of spacing h yields the linear system

$$A_h u_h = f_h.$$

Here $A_h = L_h + \text{diag}(k(x)^2)$, where L_h is the discrete Laplacian. Numerical instability may occur due to resonance. For example, when $k(x) \equiv k$ is constant,

$$A_h = L_h + k^2 I \quad \Rightarrow \quad \lambda_j(A_h) = \lambda_j(L_h) + k^2.$$

If k^2 is close to the negative of an eigenvalue of the discrete Laplacian $-\lambda_j(L_h)$ for some j, the smallest singular value of A_h becomes very small and the matrix A_h is nearly singular, which can significantly amplify numerical errors in the computed solution.

To avoid such ill-conditioned instances in the training set, the generated training data for the Helmholtz equation is filtered based on the condition number of the matrix A_h

$$\kappa_2(A_h) = \|A_h\|_2 \|A_h^{-1}\|_2$$

More specifically, cases with $\kappa_2(A_h)$ above the 80th percentile over all generated matrices are discarded to exclude numerically unstable samples.

After $k(x)$ and $f(x)$ are generated and filtered, we solve the resulting linear systems using a direct solver, which form the input-output pairs for the supervised learning of DeepONet:

$$(k(x), f(x)) \mapsto u(x). \tag{4.10}$$

4.4.3. Network Architecture Design

As demonstrated in Section 3.2.2, the DeepONet architecture consists of two sub-networks: the branch network and the trunk network.

The branch network accepts values of input functions $k(x)$ and $f(x)$ evaluated at predefined sensor points. In the training set, both $k(x)$ and $f(x)$ are discretized on a fixed uniform grid $\{x_i\}_{i=1}^{n_D}$ (e.g., $n_D = 31$). A fully connected neural network with multiple hidden layers is employed in the branch network, using ReLU as the activation function.

The trunk network receives spatial coordinates x at predefined sensor points, where tanh activation function is used in all layers. The final output of the DeepONet is given by the inner product of the output vectors from branch and trunk networks.

We use the same architecture for solving the 1D Poisson equation and the 1D Helmholtz equation. The detailed structure of the DeepONet for 1D Poisson and 1D Helmholtz equations is listed in Table 4.2.

Table 4.2: Layer-wise architecture of the DeepONet used. Each sub-network is a fully connected feedforward neural network, where "FCN" denotes the fully connected layer.

Layer	**Branch Network**	**Trunk Network**
Input	$[k(x_1), \dots, k(x_{n_D}), f(x_1), \dots, f(x_{n_D})]^T \in \mathbb{R}^{2n_D}$	$x \in \mathbb{R}$
Hidden Layer 1	FCN($2n_D \to 60$)	FCN($1 \to 80$)
Activation	ReLU	Tanh
Hidden Layer 2	FCN($60 \to 60$)	FCN($80 \to 80$)
Activation	ReLU	Tanh
Output Layer	FCN($60 \to 80$)	FCN($80 \to 80$)
Activation	None	Tanh

We kept this architecture fixed across all experiments to isolate solver effects. Preliminary trials indicated that deeper networks did not change conclusions.

4.4.4. Loss Function

For the Poisson equation, the mean squared error (MSE) loss is used.

$$\mathcal{L} = \frac{1}{Nn_D} \sum_{j=1}^{N} \sum_{i=1}^{n_D} (\hat{u}^{(j)}(x_i) - u^{(j)}(x_i))^2, \tag{4.11}$$

where $\hat{u}^{(j)}$ is the DeepONet prediction, $u^{(j)}$ is the reference solution obtained by a direct solver, N represents the total number of training samples, and $n_D = 31$ is the number of uniformly distributed grid points in the domain $\Omega = [0, 1]$.

For the Helmholtz equation, the amplitude of u varies markedly across the training data, compared to that of the Poisson equation. To avoid optimization being dominated by large-magnitude samples, we adopt the relative MSE used by Zhang et al. [19], which is defined as:

$$\mathcal{L} = \frac{1}{Nn_D} \sum_{j=1}^{N} \sum_{i=1}^{n_D} \frac{(\hat{u}^{(j)}(x_i) - u^{(j)}(x_i))^2}{\epsilon + |u^{(j)}(x_i)|^2}, \tag{4.12}$$

where $\epsilon = 0.01$ is a regularization constant.

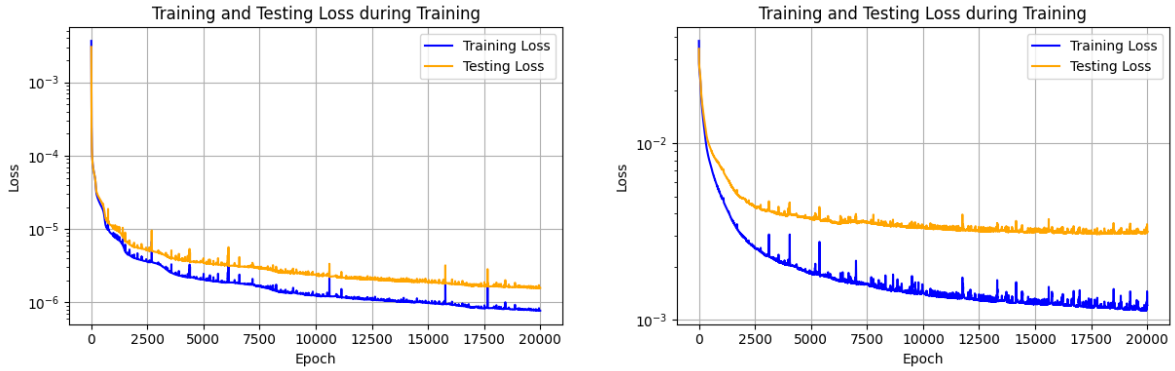
4.4.5. Training Configuration and Results

Two DeepONet models with the same architecture were trained separately for solving the Poisson and Helmholtz equations. As discussed in Section 4.4.1, The values of $k(x)$, $f(x)$, and $u(x)$ evaluated at the same grid points $\{x_i\}_{i=1}^{31}$ were used to train the DeepONet models. The key training hyperparameters are summarized in Table 4.3:

Table 4.3: Hyperparameters for DeepONet Training Process

Parameter	Value
Batch size	1,024
Number of epochs	20,000
Learning rate	0.001
DeepONet Grid Resolution	31
Training dataset size	8,500

Figs. 4.2a and 4.2b illustrate the evolution of training and validation losses over 20,000 epochs for the Poisson and Helmholtz equations, respectively. Both training and testing losses exhibit convergence during training. The model obtained at the final epoch is used in the subsequent numerical experiments, providing low-frequency error correction based on the residual.



(a) MSE loss for the Poisson equation.

(b) Relative MSE loss for the Helmholtz equation.

Figure 4.2: Training and testing losses in the training process of DeepONet

4.5. Numerical Results Validation and Discussion

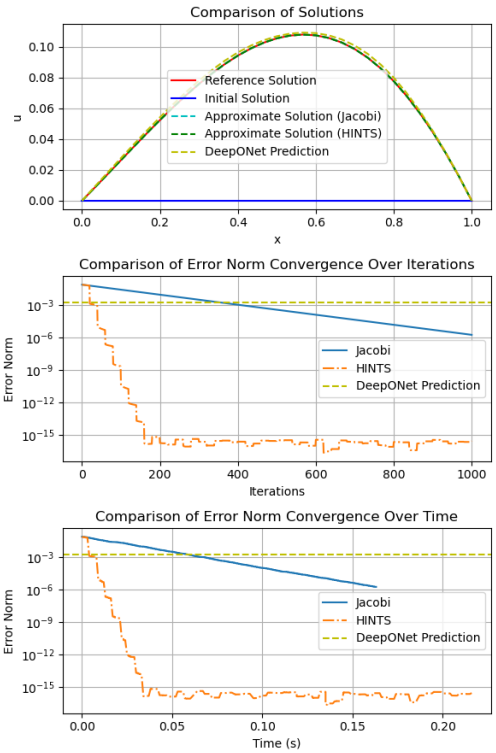
To demonstrate and independently validate the performance of these hybrid solvers, following the training procedure described in Section 4.4, numerical experiments on 1D Poisson and Helmholtz equations were conducted to compare HINTS and HINTS-MG against classical iterative solvers such as Jacobi and multigrid (MG).

4.5.1. Performance of HINTS

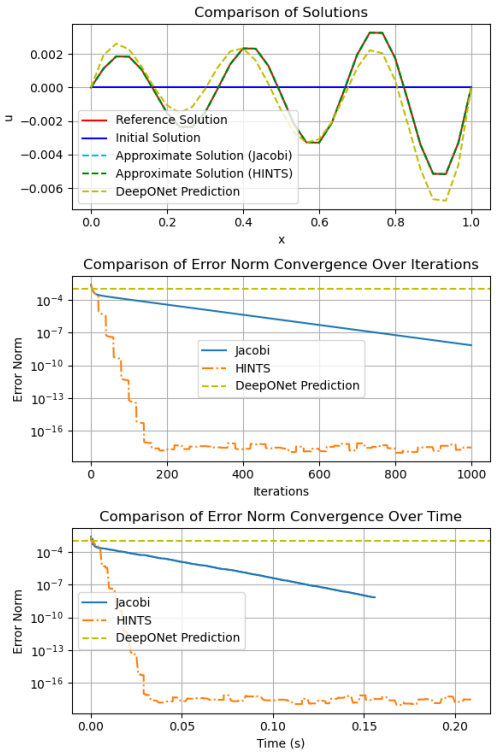
1D Poisson Equation

We first apply HINTS to the 1D Poisson equation discretized on $n = 31$ uniformly spaced grid nodes, matching the sensor points used in the DeepONet training process. Fig. 4.3 shows two representative samples: one with a low-frequency source term $f(x) = \sin(\pi x)$ and another with a slightly higher-frequency source term $f(x) = \sin(6\pi x)$. In both cases, HINTS converges to machine precision in about 200 iterations, outperforming the Jacobi method which converges slowly. In addition, a standalone DeepONet prediction is accurate for the low-frequency case but deteriorates for the higher-frequency case due to its spectral bias.

Fig. 4.4 shows the results of the same representative samples on a finer grid ($n = 1025$). HINTS still reduces the error rapidly in early iterations; however, after some iterations, its overall convergence rate rapidly slows down to that of the Jacobi method. One possible reason is the mid-range spectral components in the error, which are difficult to eliminate by either the Jacobi method or the DeepONet model. As the mesh is refined, the gap between the spectral ranges effectively handled by Jacobi (high-frequency) and DeepONet (low-frequency) becomes more significant, leaving mid-range error modes unresolved, which prevents further acceleration. A detailed analysis to validate this hypothesis is presented in Chapter 5.

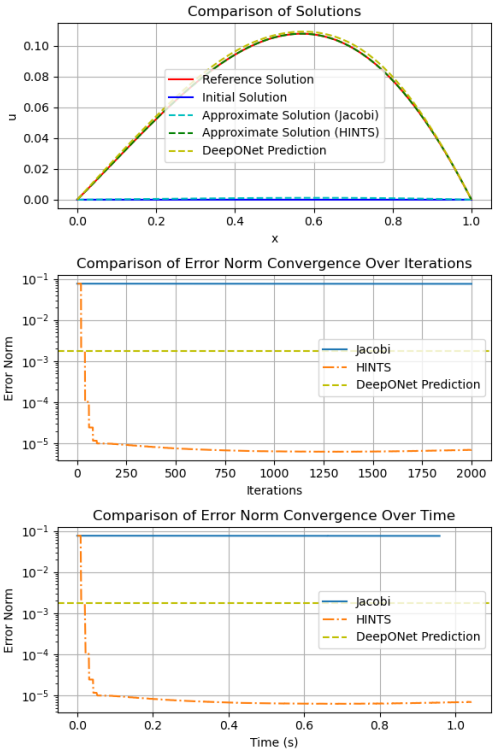


(a) Case 1: $k(x) = 1.5 - x;\ f(x) = \sin(\pi x)$

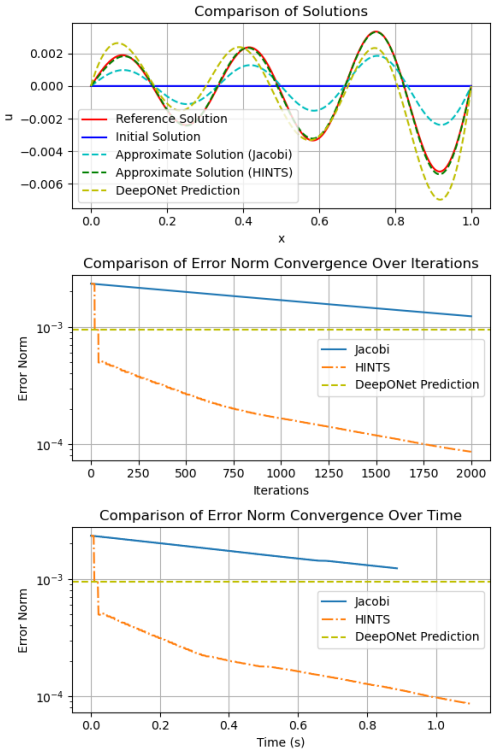


(b) Case 2: $k(x) = 1.5 - x;\ f(x) = \sin(6\pi x)$

Figure 4.3: Performance of HINTS vs. Jacobi on 1D Poisson Equation with $n = 31$ grid points. Each subfigure includes (top) approximate solution comparison, (middle) error norm over iterations, and (bottom) error norm over time.



(a) Case 1: $k(x) = 1.5 - x;\ f(x) = \sin(\pi x)$



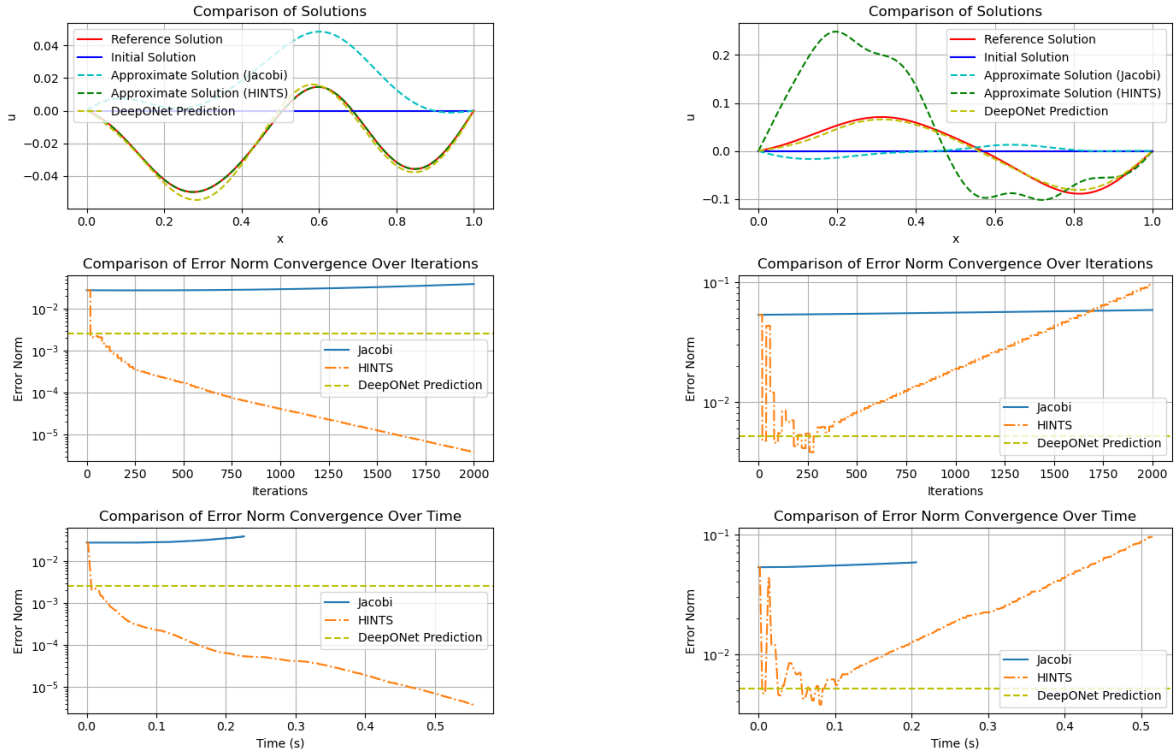
(b) Case 2: $k(x) = 1.5 - x;\ f(x) = \sin(6\pi x)$

Figure 4.4: Performance of HINTS vs. Jacobi on 1D Poisson Equation with $n = 1025$ grid points. Each subfigure includes (top) approximate solution comparison, (middle) error norm over iterations, and (bottom) error norm over time.

1D Helmholtz Equation

The performance of HINTS on the 1D Helmholtz equation (Eq. (4.2)) is investigated on a uniform grid with $n = 201$. This indefinite problem is significantly more challenging than the Poisson equation.

Experiments show that the performance of HINTS on the Helmholtz problem is not always robust. Fig. 4.5 presents representative results of HINTS on the Helmholtz problem. The Jacobi method diverges in both cases, while HINTS sometimes achieves stable and rapid convergence, as shown in Fig. 4.5a. However, in the case shown in Fig. 4.5b, the hybrid solver becomes unstable and diverges, even in the initial stage of iterations.



(a) Convergence case

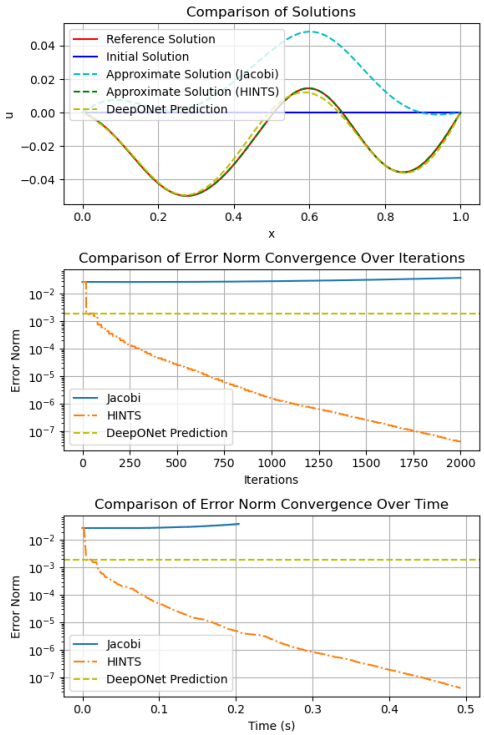
(b) Divergence case

Figure 4.5: Performance of HINTS vs. Jacobi on 1D Helmholtz Equation with $n = 201$ grid points. Each subfigure includes (top) approximate solution comparison, (middle) error norm over iterations, and (bottom) error norm over time.

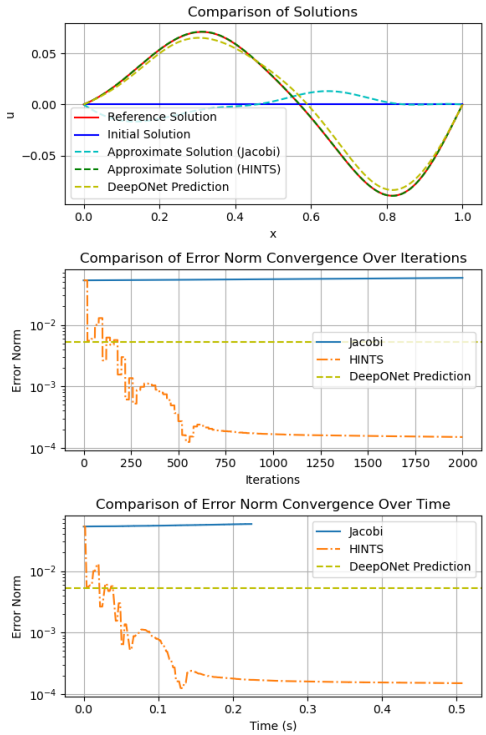
To further investigate the instability of HINTS for the Helmholtz equation, we tested the impact of training data. We trained a new DeepONet model with a more stringently filtered dataset, where samples with a matrix condition number $\kappa_2(A_h)$ above the 50th percentile were replaced with new data satisfying the criterion, instead of the 80th percentile used for the baseline model in Fig. 4.5.

Fig. 4.6 shows the performance of HINTS using the new model on the same examples as in Fig. 4.5. Compared to the performance of the baseline model shown in Fig. 4.5, the new DeepONet model achieves better performance, exhibiting faster convergence and avoiding divergence. However, for more challenging cases in Fig. 4.7, this new model still exhibits instability and divergence.

In conclusion, HINTS combines the complementary strengths of stationary iterative methods (Jacobi or Gauss-Seidel) and DeepONet to achieve faster and more robust convergence, particularly in some cases where stationary iterative methods fail to converge. Furthermore, the robustness of HINTS is highly related to the training data of the DeepONet model: filtering out samples with large matrix condition numbers $\kappa_2(A_h)$ improves the performance. Nevertheless, problems of divergence and robustness still exist in challenging cases.

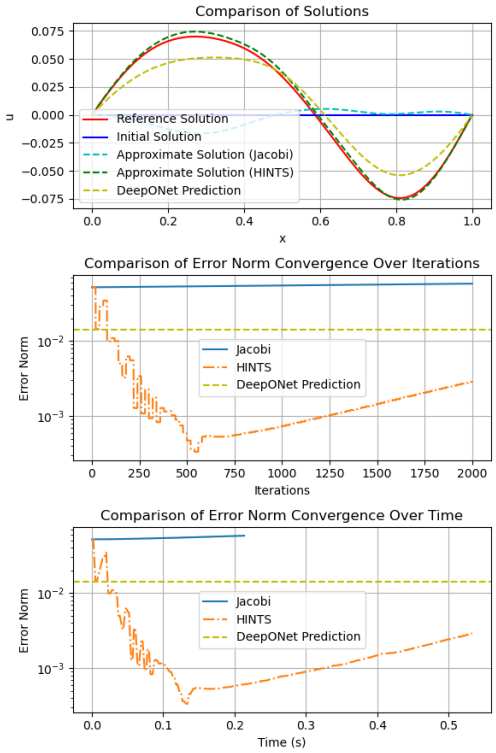


(a) Convergence case same with Fig. 4.5a

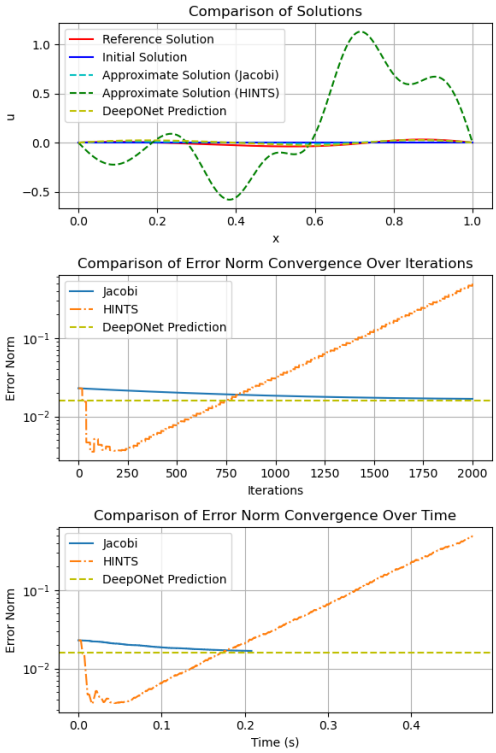


(b) Divergence case same with Fig. 4.5b

Figure 4.6: Performance of HINTS with the improved DeepONet (trained on 50th percentile filtered data) on the same samples from Fig. 4.5. Each subfigure includes (top) approximate solution comparison, (middle) error norm over iterations, and (bottom) error norm over time.



(a) Case 1



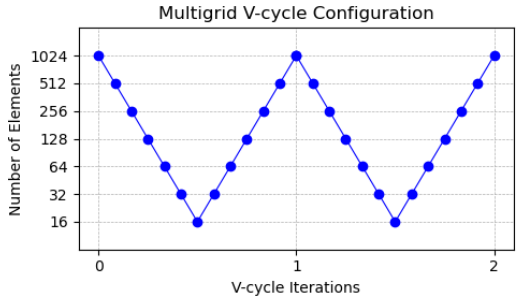
(b) Case 2

Figure 4.7: Challenging case where HINTS with the improved model still exhibits divergence or instability. Each subfigure includes (top) approximate solution comparison, (middle) error norm over iterations, and (bottom) error norm over time.

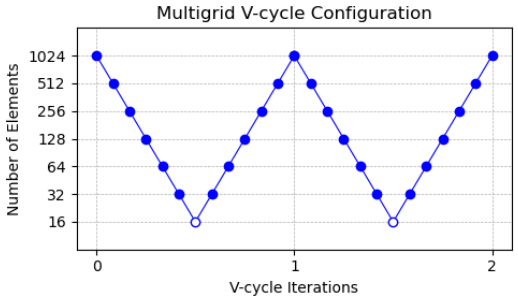
4.5.2. Performance of HINTS-MG

1D Poisson Equation

Similar to the configuration of HINTS-MG from Zhang et al. [19] on the 1D Poisson Equation, Fig. 4.8 illustrates the two V-cycle setups in our experiments on the 1D Poisson problem with a refined grid ($n = 1025$) where the HINTS approach struggles to converge. The corresponding representative results of MG and HINTS-MG are illustrated in Fig. 4.9.



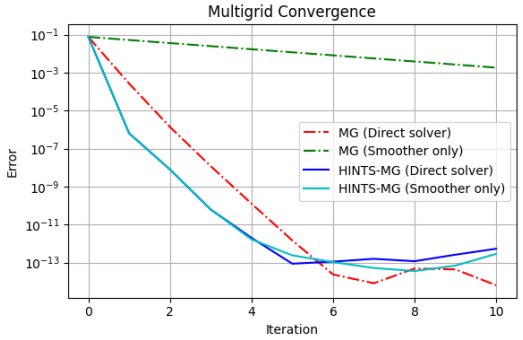
(a) "Smoother-only" V-cycle configuration.



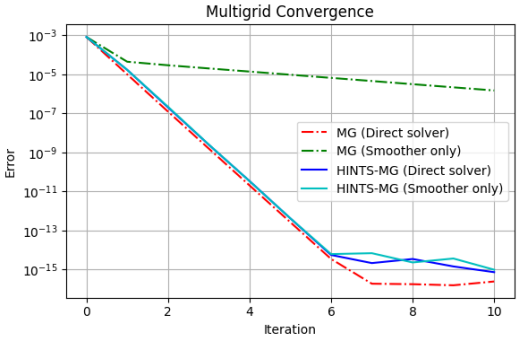
(b) "Direct-solver" V-cycle configuration.

Figure 4.8: Seven-level V-cycle configurations used in our experiments. Each solid dot represents 10 iterations of the smoother on that level, and a hollow dot denotes a direct solver. For HINTS-MG, $n_r = 10$.

Under the "smoother-only" strategy, HINTS-MG outperforms MG, reproducing the speed-up observed in Fig. 4.1. When both methods use an exact solver on the coarsest grid, HINTS-MG shows only a slight advantage for the smooth case and exhibits a similar convergence performance as MG for the oscillatory case. The results confirm that much of the performance gain of HINTS-MG in the 1D Poisson Equation reported by Zhang et al. [19] stems from the sub-optimal coarsest-grid strategy rather than the hybrid smoother itself.



(a) Smooth solution



(b) Oscillatory solution

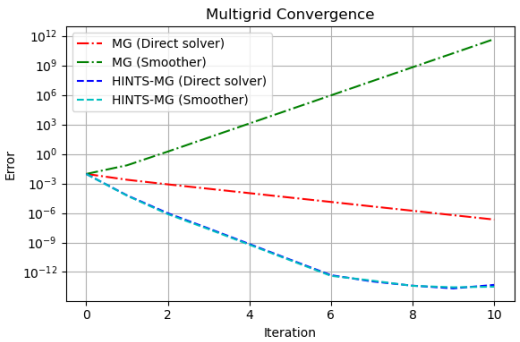
Figure 4.9: Error Norms for MG and HINTS-MG using direct solver and smoother on 1D Poisson Equation with $n = 1025$ grid points using the V-cycle configuration in Fig. 4.8.

1D Helmholtz Equation

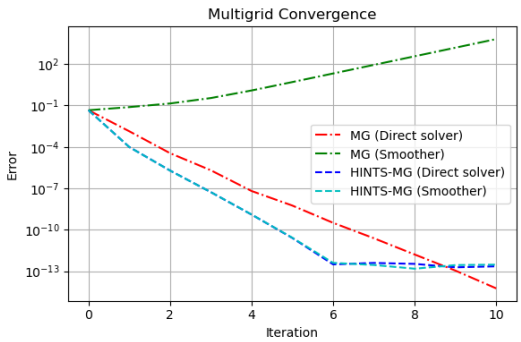
Using the V-cycle configurations in Fig. 4.8 (coarsest grid ≈15 nodes), 200 samples from the validation set of DeepONet were used to compare the performance of MG and HINTS-MG for the 1D Helmholtz equation, with the "smoother-only" and "direct solver" coarse-grid strategies. Two representative results of MG and HINTS-MG are shown in Fig. 4.10.

Under the "smoother-only" coarse-grid strategy, the classical MG diverges in most cases from the validation set due to inaccurate coarse-grid correction. In contrast, HINTS-MG converges rapidly but finally stagnates at an error norm around 10^{-13}. When both methods employ a direct solver on the coarsest grid, HINTS-MG retains the same performance, eventually stagnating at an error plateau of around 10^{-13} and outperforming the convergence rate of MG. However, in some cases, the final error

norm of MG can drop below that of HINTS-MG, but only at very low error levels after a sufficient number of iterations.



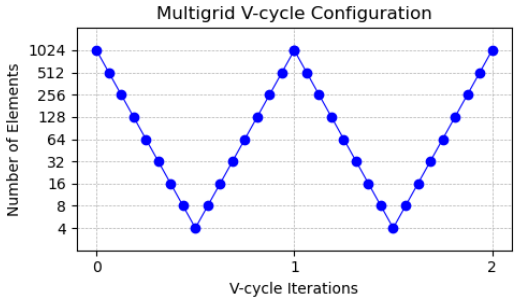
(a) HINTS-MG outperforms MG.



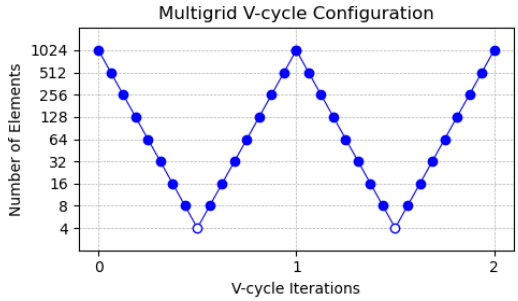
(b) MG outperforms HINTS-MG.

Figure 4.10: Performance comparison of MG and HINTS-MG on the 1D Helmholtz equation discretized with $n = 1025$ grid points using seven-level V-cycle configuration as shown in Fig. 4.8.

We then increase the number of levels so that only three nodes remain on the coarsest grid, as shown in Fig. 4.11, which is more challenging for MG and HINTS-MG as a coarsest grid with only three grid points makes it difficult to produce an accurate coarsest grid correction. In our experiment, we observe that MG diverges under both strategies, while HINTS-MG converges in approximately 86.5% of the tested samples. Representative convergence and failure cases are shown in Fig. 4.12.

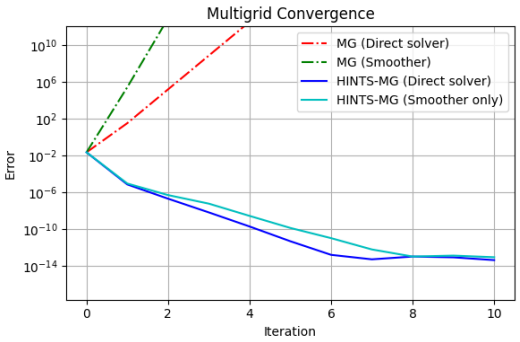


(a) "Smoother-only" V-cycle configuration.

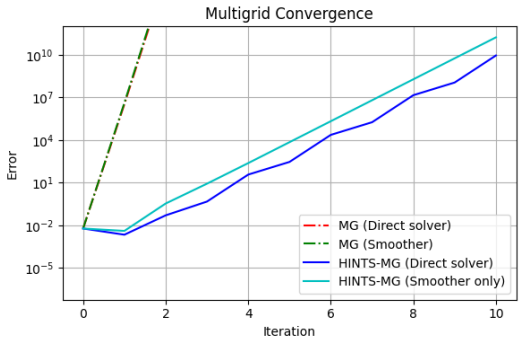


(b) "Direct-solver" V-cycle configuration.

Figure 4.11: Nine-level V-cycle configurations used in our experiments. Each solid dot represents 10 iterations of the smoother on that level, and a hollow dot denotes a direct solver. For HINTS-MG, n_r = 9.



(a) HINTS-MG converges; standard MG diverges.



(b) Both HINTS-MG and MG diverge.

Figure 4.12: Performance comparison of MG and HINTS-MG on the 1D Helmholtz equation discretized with $n = 1025$ grid points with Nine-level V-cycle configuration as shown in Fig. 4.11.

In conclusion, HINTS-MG integrates the hybrid HINTS smoother within a multigrid V-cycle. Experimental results show that employing a “smoother-only” strategy on the coarsest grid degrades both MG and HINTS-MG. When the coarsest grid is solved exactly, HINTS-MG offers only a slight advantage over the standard MG on the 1D Poisson problem. However, on indefinite Helmholtz equations, HINTS-MG substantially outperforms the standard MG, exhibiting faster and more robust convergence in most cases.

In addition, as a nonlinear operator, the DeepONet within the HINTS framework does not preserve properties of the system matrix A, such as symmetry, linearity, or positive definiteness. Consequently, unlike a standard MG method, the HINTS-MG or HINTS cannot be used as a preconditioner for Krylov subspace methods except for the flexible-GMRES (F-GMRES) [38].

5

Convergence Analysis of HINTS

This chapter investigates the convergence behavior of the HINTS method, highlighting its fast convergence at the initial stage and subsequent slowdown or even plateauing at a convergence rate comparable to stationary iterative methods. Specifically, we investigate why DeepONet works within the HINTS framework, explain why HINTS exhibits a significant slowdown in the convergence rate in later iterations, and provide spectral analysis to interpret DeepONet's effectiveness in HINTS.

5.1. Slow Convergence Plateau in HINTS

Taking the one-dimensional Poisson equation for illustration, Fig. 5.1 replots the convergence behavior of HINTS over the first 200 iterations (originally shown in Fig. 4.4 within 2000 iterations). The result reveals a sharp decline in the convergence rate after approximately 80 iterations, limiting overall acceleration in later stages.

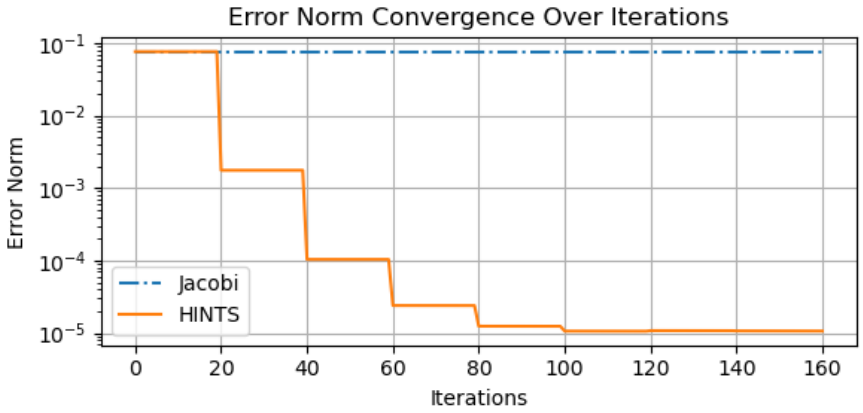


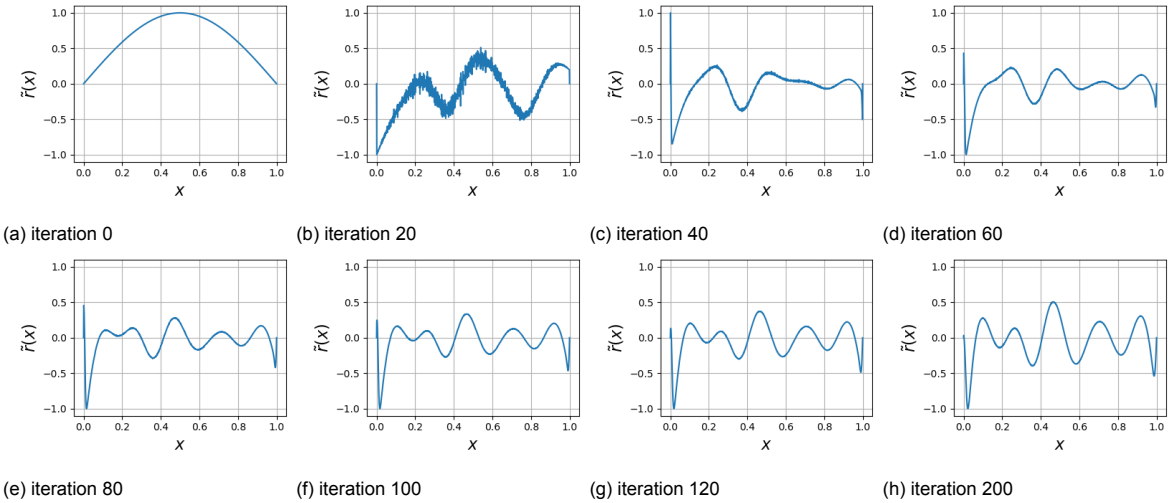
Figure 5.1: Error norm convergence over the first 200 iterations of HINTS compared to the Jacobi method.

To further understand this behavior, we analyze the input fed to the DeepONet model during each HINTS iteration. Within the HINTS framework, the residual function $r(x)$ is normalized before being passed to the DeepONet model. The normalized residual function, denoted by $\tilde{r}(x)$, is defined as:

$$\tilde{r}(x) = \begin{cases} \dfrac{r(x)}{\|r(x)\|_\infty}, & \|r(x)\|_\infty > 0, \\ 0, & \|r(x)\|_\infty = 0, \end{cases}$$

This normalization scales the residual function at each iteration to $[-1, 1]$, ensuring comparable input scales across different convergence stages and allowing for a clear comparison of the spatial profile independent of its decreasing magnitude. Fig. 5.2 displays normalized residual functions $\tilde{r}(x)$

at different stages (0, 20, 40, 60, 80, 100, 120, and 200th iterations). As observed in these figures, we found that the residual function stabilizes after iteration 80, maintaining an almost consistent shape thereafter. This stability corresponds directly to the sharp decrease in convergence speed near iteration 80. After iteration 80, the stabilized residual functions are difficult for both neural networks and stationary iterative solvers to address efficiently.



(a) iteration 0 (b) iteration 20 (c) iteration 40 (d) iteration 60

(e) iteration 80 (f) iteration 100 (g) iteration 120 (h) iteration 200

Figure 5.2: Normalized residual function $\tilde{r}(x)$ at selected iterations in the HINTS method.

5.2. The Principle of Superposition and Operator Approximation in HINTS

The capacity and limitation of the HINTS framework are rooted in the principle of superposition and how well the DeepONet model approximates them. can be understood through the principle of superposition.

Theorem 5.1 (Principle of Superposition [45]). *Let D and δ be linear differential operators acting on spatial variables $(x_1, \dots, x_n)$. For any constants c_1, c_2 and any source terms f_1, f_2:*

- *If u_1 solves the linear PDE $Du = f_1$ and u_2 solves $Du = f_2$, then $u = c_1u_1 + c_2u_2$ solves $Du = c_1f_1 + c_2f_2$.*
- *If u_1 satisfies the linear boundary condition $\delta u|_A = f_1|_A$ and u_2 satisfies $\delta u|_A = f_2|_A$, then $u = c_1u_1 + c_2u_2$ satisfies $\delta u|_A = c_1f_1|_A + c_2f_2|_A$.*

5.2.1. Error-Residual Equation in Linear PDEs

For a linear PDE operator P with exact solution $u^\star$ satisfying $Pu^\star = f$, let $\tilde{u}^{(k)}$ denote the approximate solution. The relationship between the error, defined as $e^{(k)} = u^\star - \tilde{u}^{(k)}$ and the residual, defined as $r^{(k)} = f - P\tilde{u}^{(k)}$, can be derived from Theorem 5.1:

$$Pe^{(k)} = r^{(k)}. \tag{5.1}$$

Ideally, if an inverse operator P^{-1} was available, a single iteration would yield the exact solution based on the residual

$$u^\star = \tilde{u}^{(k)} + e^{(k)} = \tilde{u}^{(k)} + P^{-1}r. \tag{5.2}$$

However, in many applications, it is challenging to explicitly find or apply the inverse operator P^{-1}. The HINTS approach approximates P^{-1} by combining two complementary operators: a fixed approximate inverse operator from classical stationary iterative methods, which effectively eliminates high-frequency errors, and a learned DeepONet model, which is trained to reduce low-frequency components.

By alternating between these two specialized operators, HINTS constructs a more effective approximation of the ideal inverse operator P^{-1}. However, since the DeepONet provides an approximation for its target components rather than the exact inverse operator P^{-1}, its effectiveness is subject to limitations such as spectral bias and distribution shift.

5.2.2. DeepONet as an Inverse Operator and Spectral Bias

DeepONet [14] approximates the PDE solution operator through supervised learning on a dataset of PDE solutions. In the HINTS framework, the DeepONet model approximates the inverse operator P^{-1} . Let DON_θ denote the trained DeepONet, the update at each iteration is:

$$\tilde{e}^{(k)} = \|r^{(k)}\|_\infty \, \mathrm{DON}_\theta(\tilde{r}^{(k)}), \qquad \tilde{u}^{(k+1)} = \tilde{u}^{(k)} + \tilde{e}^{(k)}, \qquad r^{(k)} = f - A\tilde{u}^{(k)}, \tag{5.3}$$

where $\tilde{u}^{(k)}$ denotes the current iterate, $r^{(k)}$ is the residual vector, $\tilde{r}^{(k)} := r^{(k)}/|r^{(k)}|_\infty$ is the normalized residual vector, A is the discrete PDE operator matrix, and f is the vector of the source term.

However, as discussed in Section 3.1.4, deep neural networks often exhibit difficulties in learning high-frequency features. Consequently, the predicted error $\tilde{e}^{(k)}$ produced by the DeepONet model might underfit high-frequency components of the true error $e^{(k)}$. When $e^{(k)}$ is dominated by high-frequency components, the update $\tilde{e}^{(k)}$ may degrade convergence, resulting in stagnation or even divergence.

5.2.3. Distribution Shift during Iterations

In addition to spectral bias, using DeepONet as an inverse operator also encounters challenges related to distribution shift. After iterative applications of smoothing steps (e.g., Jacobi) and DeepONet updates, residual functions $r(x)$ gradually shift toward stationary states dominated by mid-frequency modes, due to the different spectral preferences of neural networks and stationary iterative solvers.

As shown in Fig. 5.3, the discrete Fourier transform (DFT) of the residual function reveals a clear shift in spectral energy as the number of DeepONet updates increases within the HINTS framework. Specifically, the dominant frequency components of the residual migrate from low-frequency modes toward the mid-frequency range, where they gradually stabilize. This shift brings residual functions into frequency ranges where DeepONet has limited prior exposure.

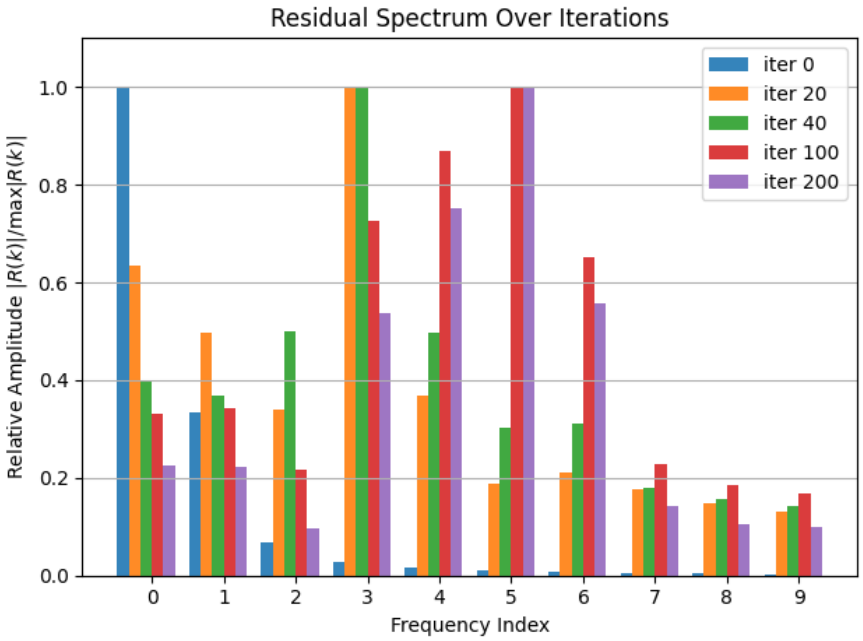


Figure 5.3: DFT amplitudes of the residual at selected iterations (iter = 0, 20, 40, 100, 200), where the amplitude $|R_t(k)|$ is normalized by its maximum $\max_| R_t(k)|$ for comparison across different iterations.

Our neural network is trained on initial source terms generated by GRFs, which are generally low-

frequency dominant (shown in blue in Fig. 5.3). During iterations, actual residual functions encountered differ significantly from training data distributions, causing a mismatch and further reducing the overall effectiveness of HINTS. Augmenting the training set with residual snapshots collected across HINTS iterations or making the DeepONet model ”aware” of the residual distributions it will actually encounter during inference will be helpful in mitigating this issue. Chapter 7 explores strategies for this data enrichment, including both offline and online methods.

5.3. Spectral Radius of the DeepONet-based Preconditioner

This section positions the DeepONet module used within the HINTS framework in the context of neural preconditioning research, which seeks to optimize the spectral properties of iterative solvers. In this section, we review common spectral losses used in neural preconditioning research, derived the iteration matrix associated with the HINTS approach, and provide insights into why DeepONet exhibits a preference for low-frequency components from the perspective of neural preconditioning.

5.3.1. Spectral Loss in Neural Preconditioning

A neural preconditioner can be viewed as a parametric mapping $P_\theta \approx A$ designed to improve the spectral properties of the preconditioned matrix $P_\theta^{-1}A$, thereby accelerating iterative solvers such as the conjugate gradient [46, 47, 48, 38].

Directly optimizing spectral properties such as the largest or smallest eigenvalues of the preconditioned matrix $P_\theta^{-1}A$ is challenging, due to high computational and memory costs in computing eigenvalues for large matrices. Hence, existing works on neural preconditioning avoid direct eigenvalue computation and instead employ inexpensive surrogate spectral losses, which are summarized as follows:

- **Frobenius loss**: Häusner et al. [47] proposed a graph neural network-based preconditioner that produces an explicit, easily invertible, and matrix-form preconditioner P_θ, and minimizes the difference between the given preconditioner P_θ^{-1} and the matrix A as below:

$$\mathcal{L}_{\mathrm{F}}(\theta) = \|P_\theta - A\|_F^2 \tag{5.4}$$

  This Frobenius loss is related to an upper bound on the largest singular value of the preconditioner matrix AP_θ

- **Stochastic Frobenius loss**: Häusner et al. [47] also use Hutchinson's trace estimator [49] to estimate the Frobenius norm in Eq. (5.4).

$$\mathcal{L}_{\mathrm{SF}}(\theta) = \mathbb{E}_w\|P_\theta w - Aw\|_2^2, \quad \mathbb{E}[ww^T] = I \tag{5.5}$$

  where the random vector w satisfying $E[ww^T] = I$ provides an unbiased estimation of the Frobenius norm.

- **Solution-weighted Frobenius loss**: Li et al. [48] introduced an inductive bias by modifying the distribution of vectors used for the expectation. Instead of random vectors, this loss uses a dataset of actual PDE solutions u and their corresponding right-hand sides $f = Au$, thus benefiting from the actual data distribution:

$$\mathcal{L}_{\mathrm{Data}}(\theta) = \mathbb{E}_{(u,f)}\|P_\theta u - f\|_2^2, \quad \mathbb{A}u = f \tag{5.6}$$

  where $u,\ f$ are from a precomputed dataset containing actual solution and actual source term.

- **Low-frequency-focused loss**: Trifonov et al. [46] derived a loss function similar to the Solution-weighted Frobenius loss (Eq. (5.6)) from the Frobenius loss (Eq. (5.4)) based on the inverse of system matrix A^{-1} such that:

$$\begin{aligned}\mathcal{L}_{\mathrm{LF}}(\theta) &= \|(P_\theta - A)A^{-1}\|_F^2 \\ &= \mathbb{E}_w\|(P_\theta - A)A^{-1}w\|^2, \quad \mathbb{E}[ww^T] = I \\ &= \mathbb{E}_w\|P_\theta A^{-1}w - w\|^2\end{aligned} \tag{5.7}$$

  Trifonov et al. [46] conjecture that optimizing Eq. (5.7) will primarily emphasize low-frequency modes in the preconditioned equation system, and have similar effects as Solution-weighted Frobenius loss.

The Frobenius loss and its stochastic counterpart aim for a globally accurate approximation of the original matrix A. In contrast, solution-weighted and low-frequency-focused losses introduce an inductive bias by sampling from the PDE solution space or A^{-1}, emphasizing low-frequency components where classical stationary iterative solvers struggle instead of creating a universal preconditioner.

5.3.2. DeepONet-based Preconditioner in HINTS

For a discretized linear PDE

$$Au = f, \quad A \in \mathbb{R}^{n\times n}, f \in \mathbb{R}^n, \tag{5.8}$$

a typical HINTS iteration is equivalent to alternating between two distinct preconditioners within a Richardson method (Algorithm 1):

- **Jacobi/Gauss-Seidel Preconditioner**

$$u^{(k+1)} = u^{(k)} + \alpha M^{-1}(f - Au^{(k)}) \tag{5.9}$$

  where M is the Jacobi or Gauss-seidel preconditioner of the matrix A, which effectively damps the high-frequency components of the error.

- **DeepONet-based Preconditioner**

$$u^{(k+1)} = u^{(k)} + \mathrm{DON}_\theta(f - Au^{(k)}) \tag{5.10}$$

  where D_θ is the DeepONet model, which approximates A^{-1} on the low-frequency band of the spectrum.

5.3.3. Spectral loss of DeepONet-based Preconditioner

The iteration matrix corresponding to the DeepONet preconditioner is given by:

$$T_\theta = I - \mathrm{DON}_\theta A \tag{5.11}$$

The convergence rate of the HINTS framework therefore depends on the spectral radius $\rho(T_\theta)$. However, due to the nonlinear and implicit nature of DeepONet, it is infeasible to explicitly compute eigenvalues directly. Instead, we optimize the upper bound of the spectral radius

$$\rho(T_\theta) \le \|T_\theta\|_2 \le \|T_\theta\|_F \tag{5.12}$$

As discussed in Section 5.3.1, stochastic estimations (biased and unbiased) can be applied to minimize the Frobenius norm $\|T_\theta\|_F$. Specifically, the following strategies are available for estimating and optimizing $\|T_\theta\|_F$:

- **Stochastic Frobenius loss**

$$\mathcal{L}_{\mathrm{SF}}(\theta) = \mathbb{E}_w\|T_\theta w\|_2^2 = \mathbb{E}_w\|w - \mathrm{D}_\theta A w\|_2^2, \quad \text{where } \mathbb{E}[ww^T] = I \tag{5.13}$$

- **Solution-weighted Frobenius loss**

$$\mathcal{L}_{\mathrm{Data}}(\theta) = \mathbb{E}_u\|u - \mathrm{D}_\theta A u\|_2^2 = \mathbb{E}_{(u,f)}\|u - \mathrm{D}_\theta f\|_2^2, \quad \text{where } \mathbb{A}u = f \tag{5.14}$$

- **Low-frequency-focused loss**

$$\mathcal{L}_{\mathrm{LF}}(\theta) = \|(P_\theta^{-1} - A)A^{-1}\|_F^2 = \mathbb{E}_w\|P_\theta^{-1}x - w\|^2, \quad \text{where } x = A^{-1}w,\ \mathbb{E}[ww^T] = I \tag{5.15}$$

Notably, the solution-weighted Frobenius loss Eq. (5.14) coincides with the standard mean square error loss Eq. (4.11) used during DeepONet training, revealing that conventional DeepONet training implicitly optimizes a biased estimation of $\rho(T_\theta)$. In practice, DeepONet training within HINTS is trained on GRF-generated, low-frequency-dominant residuals and their solutions. This biases DeepONet towards accurately approximating low-frequency modes while inadequately optimizing mid- and high-frequency components.

Conversely, the stochastic Frobenius loss, although theoretically considering all spectral bands, might overly amplify high-frequency components in the random vector during training. For the 1D Poisson equation with Dirichlet boundary conditions, the discretization A has eigenvalues

$$\lambda(A) = \frac{4}{h^2}\sin^2\Big(\frac{kh\pi}{2}\Big), \qquad k = 1, \dots, n,$$

which grows from $\approx \pi^2$ to $\approx 4/h^2$. Hence, for a random vector w, the training input–output ratio $\|Aw\|_2/\|w\|_2$ is typically large and dominated by high-frequency modes. In contrast, we use input and output pairs (f, u) during inference, with $u = A^{-1}f$. Since eigenvalues of A^{-1} range from $\approx h^2/4$ to $\approx 1/\pi^2$, the inference input–output ratio $\|f\|_2/\|A^{-1}f\|_2$ becomes much smaller. Figure 5.4 compares these two ratios, showing a significant magnitude mismatch between training $\frac{\|Aw\|}{\|w\|}$ and inference $\frac{\|f\|}{\|u\|}$. The scale mismatch during training and inference leads to unstable optimization and poor transferability from training to inference, making the stochastic Frobenius loss impractical for the HINTS.

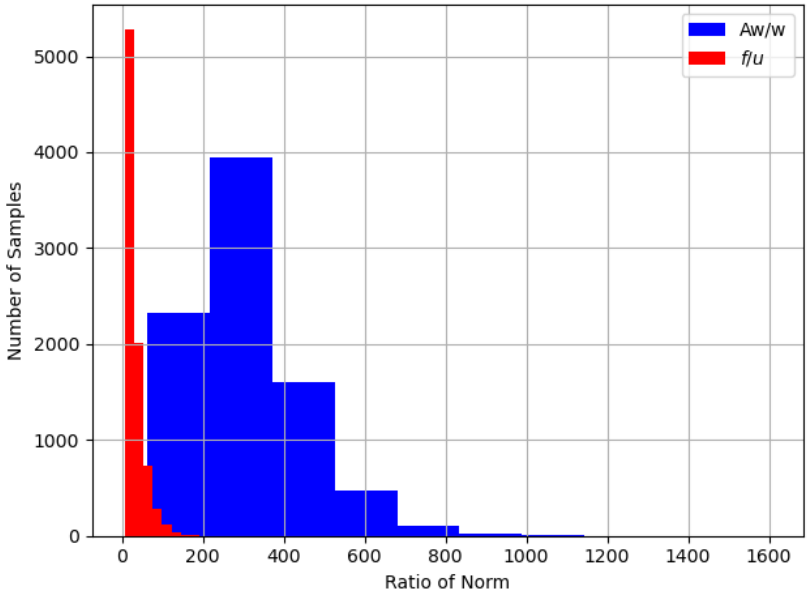


Figure 5.4: Ratio of Norm between Inputs and Expected Outputs of the DeepONet model using the stochastic Frobenius loss. Blue: training $\|Aw\|_2/\|w\|_2$ with random vector satisfying $\mathbb{E}[ww^T] = I$; red: inference $\|f\|_2/\|u\|_2$ with $u = A^{-1}f$.

5.3.4. Performance of Low-frequency-focused loss

Trifonov et al. [46] claimed that the low-frequency-focused loss $\mathcal{L}_{\mathrm{LF}}$ would exhibit similar effectiveness to the solution-weighted Frobenius loss $\mathcal{L}_{\mathrm{Data}}$, as both implicitly emphasize low-frequency modes in the preconditioned system. To empirically validate this hypothesis in HINTS, we conducted numerical experiments on the one-dimensional Poisson equation, comparing the performance of DeepONet models trained using the low-frequency-focused loss $\mathcal{L}_{\mathrm{LF}}$ and the conventional mean squared error (MSE) loss in Eq. (4.11).

Fig. 5.5 illustrates the convergence behavior of HINTS iterations under these two loss functions. The DeepONet model optimized using the low-frequency-focused spectral loss (HINTS-LF) slightly outperformed the baseline model trained with the MSE loss (HINTS-Baseline), demonstrating a slightly more rapid reduction in both error norm and residual norm.

We also embedded the trained DeepONet models into a multigrid (MG) framework, the performance of which is shown in Fig. 5.6. The model trained by the low-frequency-focused loss, HINTS-MG (LF), showed slightly faster convergence rates compared to the baseline model, HINTS-MG (Baseline), trained by a MSE loss, while both HINTS-MG models outperformed the classical multigrid method.

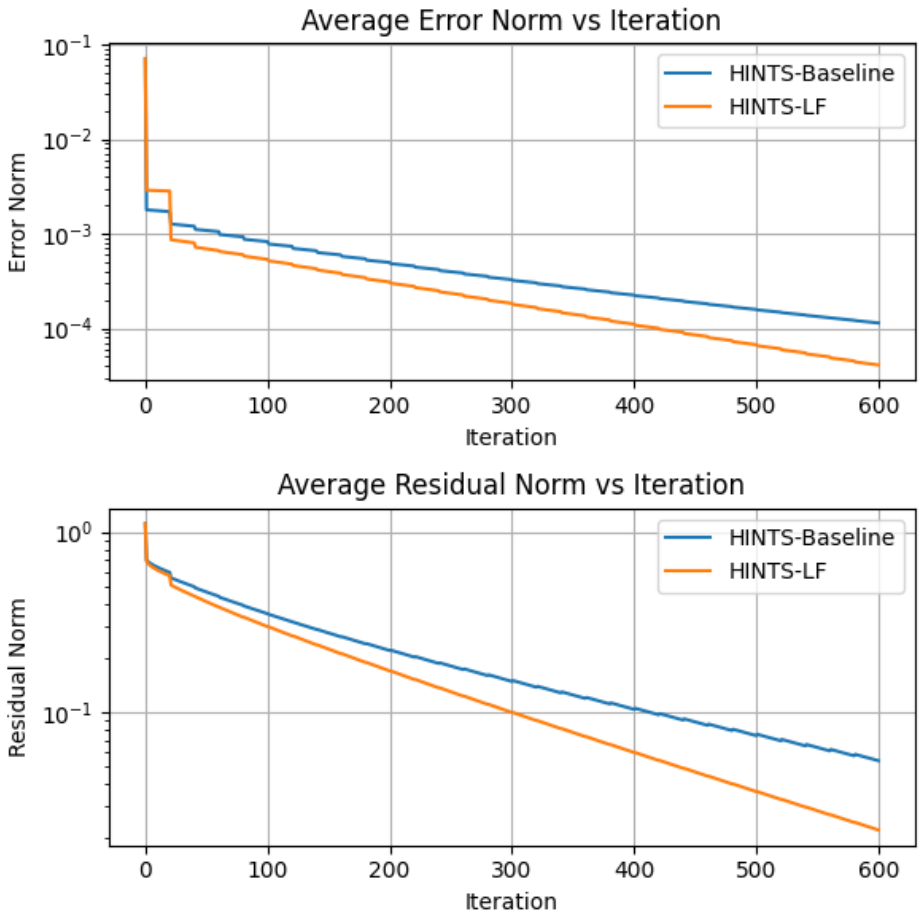


Figure 5.5: Comparison of convergence behavior between DeepONet models trained with low-frequency-focused loss, HINTS (LF), and mean squared error loss, HINTS (Baseline), within the HINTS framework.

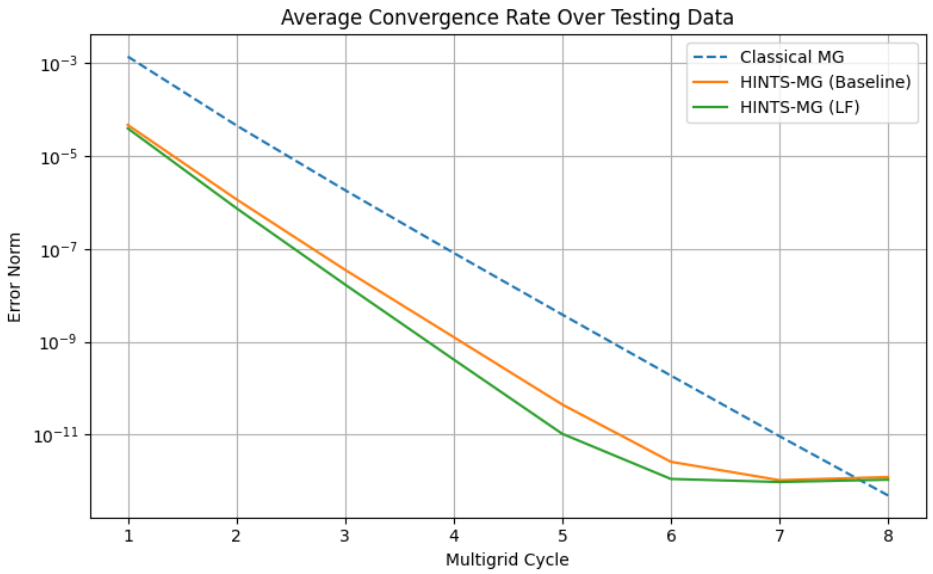


Figure 5.6: Comparison of average convergence rate over the testing data using classical multigrid (MG) and HINTS-MG with DeepONet models trained with the MSE and low-frequency-focused loss (Baseline & LF).

These experimental results indicate that DeepONet training implicitly optimizes a biased estimation of spectral properties. In addition, the choice of training data and associated loss functions influences the performance of the model. Specifically, the performance of the low-frequency-focused loss indicates that the trained DeepONet is able to effectively resolve low-frequency components, thus leading to a convergence performance comparable to MSE in HINTS, confirming the statement of Trifonov et al. [46].

6

Gradient-Enhanced HINTS (GE-HINTS)

This chapter extends the original HINTS framework by incorporating first-order derivative information into the training of the DeepONet model. We show that the resulting gradient-enhanced HINTS (GE-HINTS) mitigates the issue of spectral bias in the DeepONet model and therefore exhibits better performance.

6.1. Motivation and the Anti-Frequency Principle

The residual plateau in the standard HINTS, as discussed in Chapter 5, originates from the different spectral preferences between the DeepONet model (low frequency) and stationary iterative methods (high frequency). Due to spectral bias, deep neural networks tend to learn low-frequency components first and struggle with high-frequency ones [16, 19]. Hence, mitigating the spectral bias can narrow the gap between the frequency ranges of the DeepONet model and stationary iterative methods.

In contrast to the spectral bias that is common in training neural networks, the anti-frequency principle refers to cases in which high-frequency components have equal or higher priority in learning than low-frequency ones [16]. A common way to alleviate the effect of the spectral bias or even induce the anti-frequency principle is to impose a high priority on high-frequency components. Whether the anti-frequency principle can be observed is determined by the competition between the activation regularity and the loss function [16]. When the loss function has a higher priority for high-frequency components than the low-frequency priority induced by the activation function, the anti-frequency principle emerges.

A practical way to impose a high priority on high-frequency components is to add derivative terms to the loss [16]:

$$\mathcal{L}_{\mathrm{GE}} = \|u_\theta - u^\star\|^2 + \alpha \|\nabla u_\theta - \nabla u^\star\|^2, \tag{6.1}$$

where $\{x_j\}_{j=1}^N$ are evaluation points and $\alpha > 0$ is the weight that balances data and gradient terms. In the Fourier domain, the derivative is given by:

$$\mathcal{F}[\nabla u](\xi) = i\,\xi\,\mathcal{F}[u](\xi).$$

Hence, the mean squared error of the gradient contributes a frequency factor $|\xi|^2$ to each mode [16]. As a result, higher frequencies are penalized more with a higher weight and tend to converge earlier [50, 51]. When the weight of the gradient penalty is sufficiently large, it may mitigate or even overcome the network's spectral bias, exhibiting the anti-frequency principle [16]. Motivated by this, we designed a gradient-enhanced loss for the DeepONet model to mitigate its spectral bias and, consequently, the overall performance of HINTS.

6.2. Gradient-Enhanced Loss of DeepONet

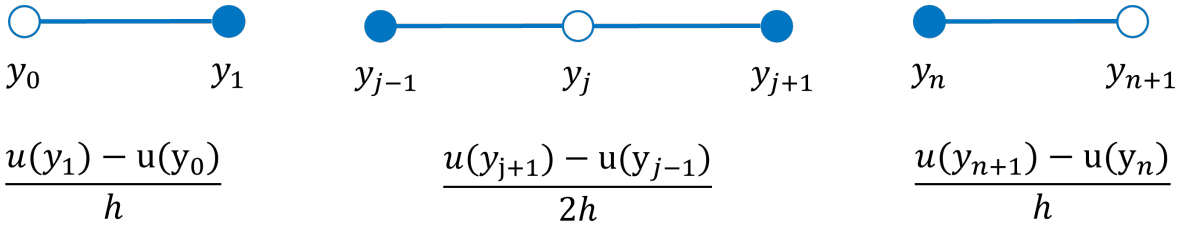
To incorporate first-order derivatives, labels for both the predicted solution u_θ and its first-order spatial derivatives are required during training. The following gradient-enhanced loss function can be adopted for the DeepONet model:

$$\mathcal{L}_{\mathrm{GE}}(\theta) \;=\; \frac{1}{N}\sum_{j=1}^{N}\Big(\big|u_\theta(y_j) - u^\star(y_j)\big|^2 \;+\; \alpha\,\big|\nabla u_\theta(y_j) - \nabla u^\star(y_j)\big|^2\Big). \tag{6.2}$$

where $\nabla u_\theta(y_j)$ is the gradient of the DeepONet model's output, which can be obtained by automatic differentiation, $\nabla u^\star(y_j)$ is the pre-computed reference gradient, which can be obtained by finite differences, and α denotes the weight of the derivative term. For a 1D PDE's solution, the reference gradient of the solution at y_j on $\Omega_h = \left\{\frac{j}{n+1}\right\}_{j=0}^{n+1}$ is computed as follows:

$$\nabla u^\star(y_j) \approx \begin{cases} \dfrac{u^\star(y_j+h)-u^\star(y_j)}{h}, & \text{if } j=0, \\ \dfrac{u^\star(y_j+h)-u^\star(y_j-h)}{2h}, & \text{if } 1\le j\le n, \\ \dfrac{u^\star(y_j)-u^\star(y_j-h)}{h}, & \text{if } j=n+1, \end{cases}$$

where $h = 1/(n+1)$ is the mesh size. Figure 6.1 provides a schematic of these forward, central, and backward difference stencils.



(a) Forward difference ($j=0$) (b) Central difference ($1\le j\le n$) (c) Backward difference ($j=n+1$)

Figure 6.1: Schematic of the finite difference stencils used to compute the reference gradient $\nabla u^\star$ at boundaries and interior points.

The gradient is required only for training. At the inference stage, since no labels or predictions of gradient are required, GE-HINTS behaves exactly like the original model and the cost per solver iteration remains the same.

6.3. Results & Discussion

In this section, we present numerical experiments to evaluate the effectiveness of the Gradient-Enhanced HINTS (GE-HINTS) framework. We consider the standard 1D Poisson equation and apply different grid resolutions of the DeepONet's sensor points (31, 51, 81). To investigate the influence of the gradient-loss weight parameter α, we conduct a parametric study with α set to 0.1, 0.2, and 0.5. Table 6.1 summarized the hyperparameters for training these DeepONet models.

Table 6.1: Hyperparameters for Gradient-Enhanced DeepONet

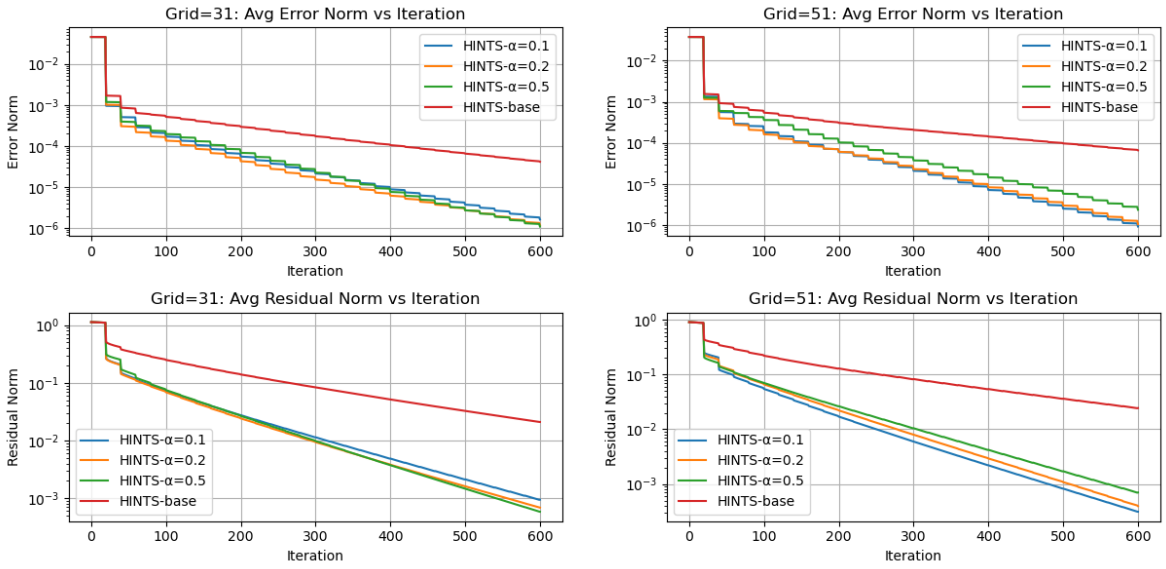
Parameter	Value
Batch size	1,024
Number of epochs	10,001
Learning rate	0.0001
DeepONet grid resolution	31, 51, 81
Training dataset size	8,500
Gradient-loss Weight α	0.1, 0.2, 0.5

6.3.1. Performance of Gradient-Enhanced HINTS

For each configuration, we compare GE-HINTS with the original HINTS method (denoted as HINTS-base).

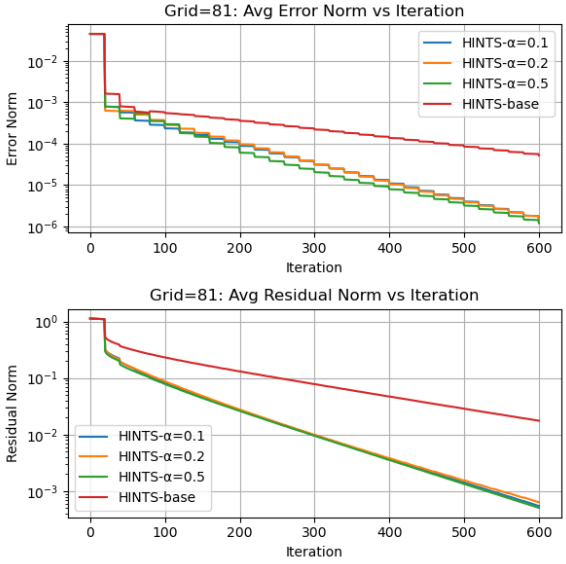
Figures 6.2 summarize the experimental results by plotting the convergence history of the error norm and residual norm against the iteration number when applying GE-HINTS to the 1D Poisson equation ($n = 201$). As depicted, the gradient-enhanced training significantly improves the convergence speed

of both error and residual norms compared to the HINTS-base. In addition, all tested values of the gradient-loss weight $\alpha(0.1, 0.2, and 0.5)$ yield a clear improvement, while the performance difference among them is relatively small.



(a) DeepONet grid resolution = 31

(b) DeepONet grid resolution = 51



(c) DeepONet grid resolution = 81

Figure 6.2: Convergence of error norm and residual norm for GE-HINTS with varying grid resolutions and gradient-loss weights α. HINTS-base denotes the baseline DeepONet model without the loss term based on gradient; HINTS-α is the GE-HINTS with the gradient-loss weight α; DeepONet Grid resolution represents the number of sensor points in the DeepONet model

Another observation is the robustness of GE-HINTS across varying grid resolutions. For different sizes of sensor points (DeepONet grid resolution = $31, 51, 81$), GE-HINTS consistently outperforms the baseline HINTS, demonstrating its effectiveness in enhancing DeepONet's capabilities and speeding up the convergence rate of the HINTS framework.

6.3.2. Performance of Gradient-Enhanced MG-HINTS

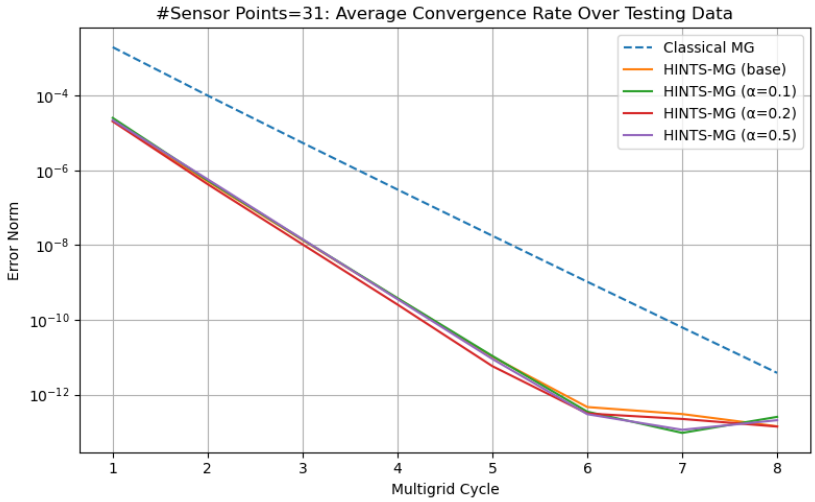
To further assess the generalizability of GE-HINTS and its compatibility with multilevel solvers, we embed the trained gradient-enhanced DeepONet preconditioners into the Multigrid-HINTS (MG-HINTS) framework. This hybrid solver leverages multigrid cycles to resolve different spectral components

across grid levels, while using learned DeepONet-based smoothers to accelerate convergence. For comparison, we also evaluate the performance of the classical multigrid (MG) method under the same test settings.

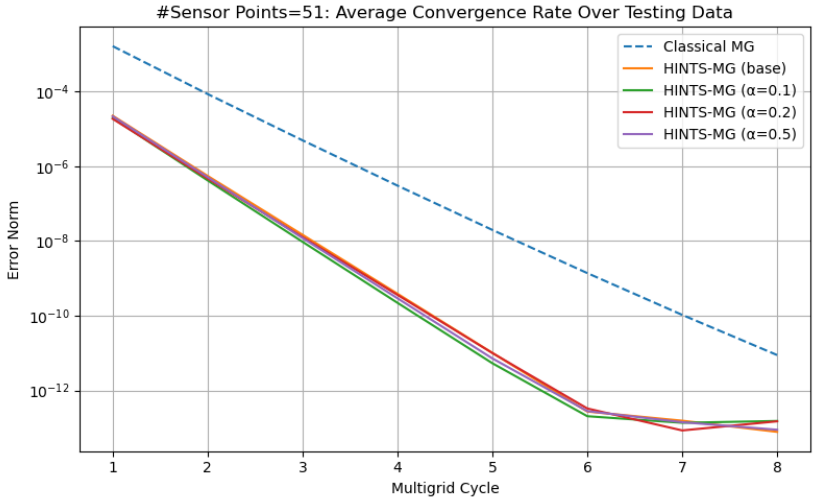
Figure 6.3 presents the convergence of the error norm across multigrid cycles for MG-HINTS using DeepONet models trained at different resolutions (31, 51, 81) and various gradient-loss weights α. All experiments are conducted on the 1D Poisson equation with $n = 1024$ grid points on the finest level.

Results demonstrate that regardless of the mesh on which the DeepONet corrector is trained, the resulting HINTS-MG solver converges faster than the classical MG method. Furthermore, for any grid resolution, the influence of gradient-enhanced loss and the choice of gradient-enhanced weight $\alpha \in 0.1, 0.2, 0.5$ on multigrid convergence is insignificant. This observation suggests that gradient-enhanced training is most advantageous in single-level HINTS, whereas its impact on HINTS-MG is negligible. The reason might be that the additional spectral coverage gained from gradient supervision becomes less critical when embedded into a multigrid scheme.

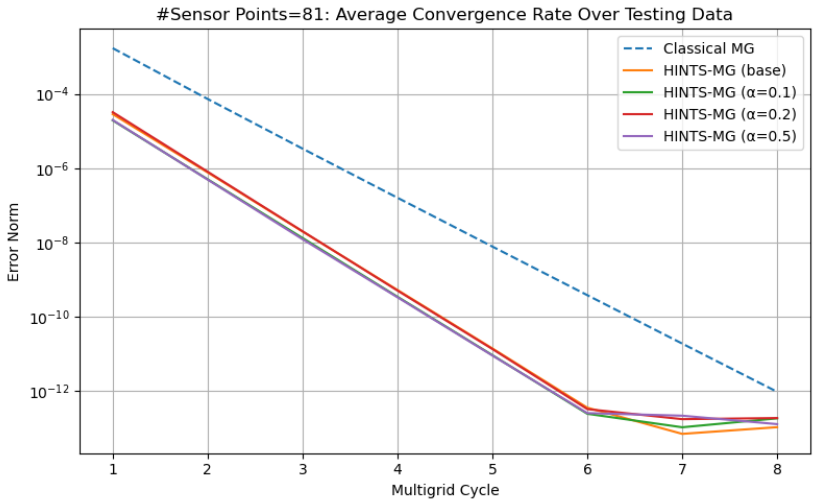
In summary, while gradient-enhanced loss significantly improves single-level HINTS solvers, its impact becomes less prominent in the multigrid context. Nonetheless, HINTS-MG remains competitive and surpasses classical MG among all tested resolutions, highlighting the effectiveness of DeepONet preconditioners.



(a) MG grid resolution = 31



(b) MG grid resolution = 51



(c) MG grid resolution = 81

Figure 6.3: Convergence performance of Gradient-Enhanced MG-HINTS for varying grid resolutions and gradient-loss weights α.

7

HINTS-in-the-loop

As a hybrid solver that incorporates a data-driven neural network model, the HINTS framework relies heavily on the similarity between the training data distribution and the actual residual functions encountered during the iterative solving process. However, discrepancies in these distributions can severely compromise the accuracy of the neural network's prediction as iterations progress, leading to a situation where the solver performs well only in the early stages of the iterative process, but stagnates or even diverges in later iterations.

This chapter aims to address the issue of data distribution shift discussed in Chapter 5. We introduce and evaluate two training strategies designed to mitigate the negative impact of distribution shift by making the DeepONet model "aware" of the actual residuals in the iterative process during training. Motivated by the "Solver-in-the-loop (SOL)" concept [52], two strategies were explored to mitigate the negative impact of distribution shift:

1. Offline pre-compute strategy: augments the training data with pre-calculated residual snapshots.

2. Online in-loop strategy: integrates the HINTS iterative process directly into a differentiable training pipeline.

Numerical validation is provided along with detailed discussions on their performance.

7.1. Offline Pre-compute Strategy

The offline pre-compute (PRE) strategy aims to alleviate the distribution shift through offline data resampling. The core idea is to simulate the HINTS iterative process using a pretrained DeepONet and collect actual residuals and corresponding target increments $(r^{(k)}, \Delta u^{(k)})$ along the iterative trajectories of HINTS. The newly collected dataset is then used to fine-tune the pretrained DeepONet model to further adapt it to the residual distribution encountered during inference.

7.1.1. Algorithm

Given an initial DeepONet model DON_{θ_0}, the process of PRE strategy is summarized in Algorithm 8:

Algorithm 8 Offline Pre-compute Strategy

Require: Pre-trained network DON_{θ_0}, original dataset $D_{\mathrm{ORI}} = \{(A_i, f_i, u_i^\star)\}_{i=1}^{N_f}$, look-ahead size K, Jacobi iterations per cycle n_r
Ensure: Fine-tuned network $\mathrm{DON}_{\theta_{\mathrm{PRE}}}$
Initialize the augmented dataset $\mathcal{D}_{\mathrm{PRE}} \leftarrow D_{\mathrm{ORI}}$
for each source term vector f_i and exact solution $u_i^\star$ in D_{ORI} **do**
  Set initial guess for the HINTS $u_{\mathrm{cur}} \leftarrow 0$.
  for $k = 1, \dots, K$ **do**
    for $j = 1, \dots, n_r - 1$ **do**
      Apply Jacobi: $u_{\mathrm{cur}} \leftarrow u_{\mathrm{cur}} + D^{-1}(f_i - A_i u_{\mathrm{cur}})$.
    end for
    Compute the residual that will be fed into DeepONet: $r^{(k)} \leftarrow f_i - A u_{\mathrm{cur}}$.
    Compute the target increment (true error): $\Delta u^{(k)} \leftarrow u_i^\star - u_{\mathrm{cur}}$.
    Collect data: $\mathcal{D}_{\mathrm{PRE}} \leftarrow \mathcal{D}_{\mathrm{PRE}} \cup \left(r^{(k)}, \Delta u^{(k)}\right)$
    Apply DeepONet: $u_{\mathrm{cur}} \leftarrow u_{\mathrm{cur}} + \mathrm{DON}_{\theta_0}(r^{(k)})$
  end for
end for
Fine-tune network: $\theta_{\mathrm{PRE}} \leftarrow \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{(r,\Delta u)\sim\mathcal{D}_{\mathrm{PRE}}} \left\|\mathrm{DON}_\theta(r) - \Delta u\right\|_2^2$
return $\mathrm{DON}_{\theta_{\mathrm{PRE}}}$

7.1.2. Limitations

Although the PRE strategy provides a resampled dataset to alleviate the gap of data distribution between training and inference, two major limitations still exist.

Drift after Parameter Update Fine-tuning on $\mathcal{D}_{\mathrm{PRE}}$ yields an updated DeepONet model $\mathrm{DON}_{\theta_{\mathrm{PRE}}}$. However, the updated DeepONet model will have a different residual trajectory in subsequent solves. As a result, the encountered residuals for the new network might have a distribution that lies outside the dataset $\mathcal{D}_{\mathrm{PRE}}$ on which it has recently been trained. This mismatch is inherent to the PRE strategy and is hard to be eliminated using a single offline fine-tuning step.

Network Model Prerequisite The data collection phase in Algorithm 8 requires a pre-trained DeepONet model DON_{θ_0} to generate residuals $r^{(k)}$ encountered in the HINTS iterative process. However, the entire procedure, including pre-training, data resampling, and fine-tuning, requires training the DeepONet model at least twice.

7.2. Online In-loop Strategy

Unlike the offline PRE strategy, the in-loop (IL) strategy incorporates the entire iterative HINTS process, including Jacobi and DeepONet iterations, into a differentiable computational graph. For each data sample, the algorithm simulates the HINTS iterative process for K cycles (denoted by the look-ahead size in Algorithm 9), starting from a zero initial guess $u^{(0)} = 0$. The loss is computed over all K HINTS cycles. Gradients are obtained by backpropagation through the computational graph, which are then used to update the neural network parameters. The IL strategy ensures that the residuals for training evolve as the model is updated, rather than relying on a static, pre-computed dataset. Furthermore, the IL strategy ensures that the model only needs to be trained once.

7.2.1. Algorithm

In the IL strategy, each HINTS cycle is a single differentiable step in the algorithm, which consists of two main stages:

1. **DeepONet Update**: $u^{(k+n_r)} = u^{(k)} + \mathrm{DON}_\theta(r^{(k+n_r)})$
2. **Jacobi Updates**: $u^{(k+i)} = u^{(k+i-1)} + D^{-1} r^{(k+i-1)}$, for $i = 1, \cdots, n_r - 1$

These steps form a differentiable computational graph, enabling end-to-end training. The network parameters are optimized by minimizing a loss function defined over future K steps (the "look-ahead" size), as detailed in Section 7.2.2. Alg. 9 summarizes the main process of an IL strategy using solution deviation as the loss function for the HINTS framework.

Algorithm 9 Online In-loop Strategy

Require: Initialized DeepONet model DON_θ; training set $\mathcal{D}_{\mathrm{ORI}} = \{(A_i, f_i, u_i^\star)\}_{i=1}^{N_f}$; Jacobi iterations per cycle n_r; look-ahead size K; epochs E; learning rate η
Ensure: Updated network DON_θ

for epoch $= 1$ **to** E **do**
  for each $(A_i, f_i, u_i^\star)$ in $\mathcal{D}_{\mathrm{ORI}}$ **do**
    Initialize solution: $u_{\mathrm{cur}} \leftarrow 0$.
    Initialize loss: $\mathcal{L}_{\mathrm{sol}}(\theta) = 0$
    for $k = 1$ **to** K **do**
      Apply DeepONet: $\Delta u \leftarrow \mathrm{DON}_\theta(f_i - A_i\, u_{\mathrm{cur}})$.
      Update solution: $u_{\mathrm{cur}} \leftarrow u_{\mathrm{cur}} + \Delta u$.
      Compute residual: $r^{(k)} \leftarrow f_i - A_i\, u_{\mathrm{cur}}$.
      Update loss: $\mathcal{L}_{\mathrm{sol}}(\theta) \leftarrow \mathcal{L}_{\mathrm{sol}}(\theta) + \|u_{\mathrm{cur}} - u_i^\star\|_2^2$
      for $j = 1, \ldots, n_r - 1$ **do**
        $u_{\mathrm{cur}} \leftarrow u_{\mathrm{cur}} + D^{-1}(f_i - A_i u_{\mathrm{cur}})$.
      end for
    end for
    Back Propagation: $\theta \leftarrow \theta - \eta\, \nabla_\theta \mathcal{L}_{\mathrm{sol}}(\theta)$
  end for
end for
return DON_θ

When the look-ahead size $K = 1$ and the initial guess $u^{(0)} = 0$, one IL cycle using solution deviation as the loss function is equivalent to the standard supervised training of a DeepONet. The IL strategy with the look-ahead size $K = 1$ and the loss function of solution deviation is:

$$u^{(1)} \;=\; u^{(0)} + \mathrm{DON}_\theta(f - A\, u^{(0)}) = \mathrm{DON}_\theta(f), \quad \mathcal{L}_{\mathrm{sol}}(\theta) = \|u^{(1)} - u^\star\|_2^2.$$

In this case, the training objective is identical to training a standard DeepONet designed to learn the mapping $(k, f) \mapsto u^\star$.

7.2.2. Loss Functions

To train the DeepONet model DON_θ with the IL strategy, we evaluated three different loss functions, each designed to optimize a different property of the HINTS solver: the accuracy of the solution, the magnitude of the residual, and the rate of convergence.

1. **Solution loss**

$$\mathcal{L}_{\mathrm{sol}}(\theta) = \frac{1}{K} \sum_{k=1}^{K} \left\| u^{(k)} - u^\star \right\|_2^2. \tag{7.1}$$

   The solution loss directly penalizes the deviation from the ground-truth solution $u^\star$. Because the reference solution $u^\star$ must be pre-computed, this loss is limited to settings where reference solutions are available.

2. **Residual loss**

$$\mathcal{L}_{\mathrm{res}}(\theta) = \frac{1}{K} \sum_{k=1}^{K} \left\| r^{(k)} \right\|_2^2. \tag{7.2}$$

   The residual loss minimizes the magnitude of the residual at each step of the trajectory. The residual is available during the iteration and therefore requires no reference solution.

3. **Convergence loss**

$$\mathcal{L}_{\text{con}}(\theta) = \frac{1}{K}\sum_{k=1}^{K}\frac{\|r^{(k)}\|_2}{\|r^{(k-1)}\|_2}^2, \tag{7.3}$$

The convergence loss is designed to directly optimize the speed of convergence, where $\|r^{(k)}\|_2/\|r^{(k-1)}\|_2$ represents the convergence rate per DeepONet update. The convergence loss can be interpreted as a weighted sum of residual norms $\|r^{(k)}\|_2$ with dynamic weights, which encourages faster convergence.

7.3. Results & Discussion

In this section, numerical experiments on the 1D Poisson equation were conducted to evaluate the effectiveness of pre-compute and in-loop strategies.

7.3.1. Offline Pre-Compute Strategy

In the pre-compute experiments, we reuse the baseline DeepONet from Section 4.5.1, collect residual–update pairs for the first five DeepONet calls, and fine-tune the model on this trajectory data. All tests are carried out on the 1D Poisson equation discretized by different meshes.

Training & Validation Loss

Figure 7.1 shows the loss curves during fine-tuning. The loss on the newly collected data (blue) decreases within the first 5000 epochs and then plateaus, indicating that the network has adapted to the in-loop residual distribution. The validation loss on the original dataset (orange) remains almost unchanged, demonstrating that the fine-tuned model retains its predictive capability on the original training data and does not overfit to the resampled residual trajectory.

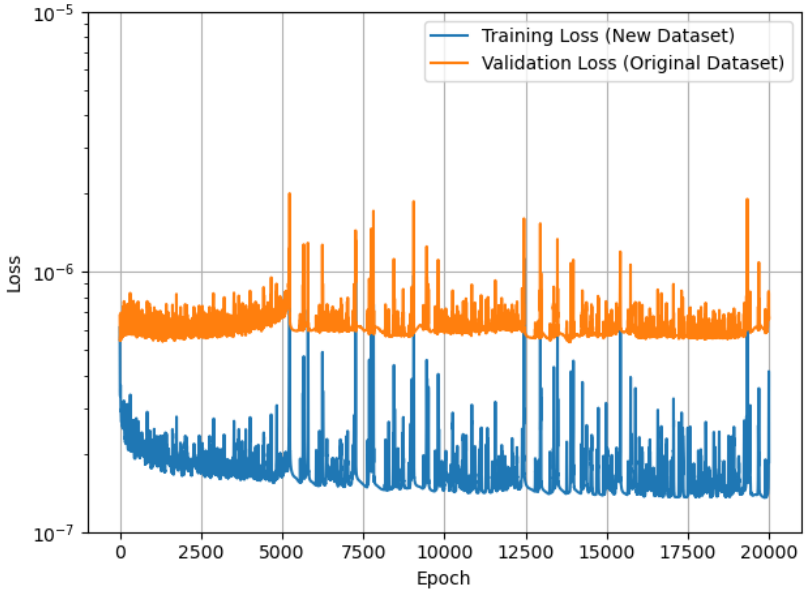
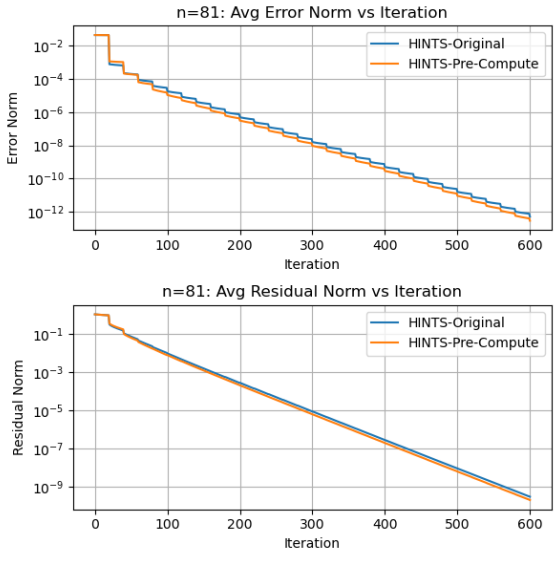


Figure 7.1: Fine-tuning loss on resampled data (blue) and validation loss on the original dataset (orange).

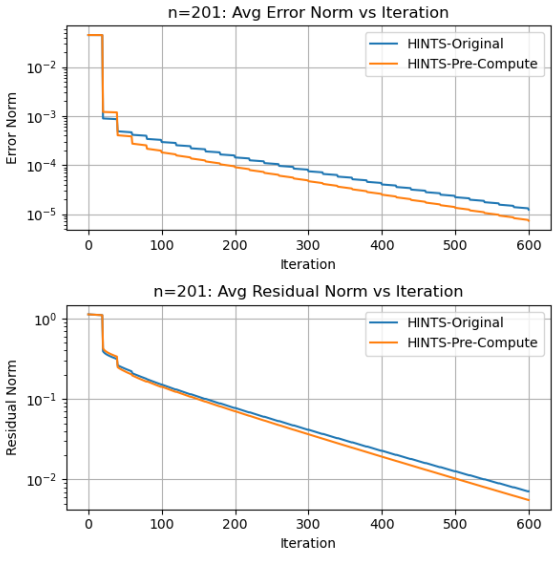
Impact on HINTS & HINTS-MG

Figure 7.2 compares the average convergence histories of HINTS with and without the pre-compute strategy on 1D Poisson equations with uniform grids of $n=81$, 201, 501, and 801. On the coarsest grid ($n = 81$) the two curves almost coincide. As the grid is refined, the performance gain of the pre-compute strategy can be observed. This might be because on the coarsest grid ($n = 81$), the gap between the frequency preferences of the DeepONet and stationary iterative solvers is not significant. The results

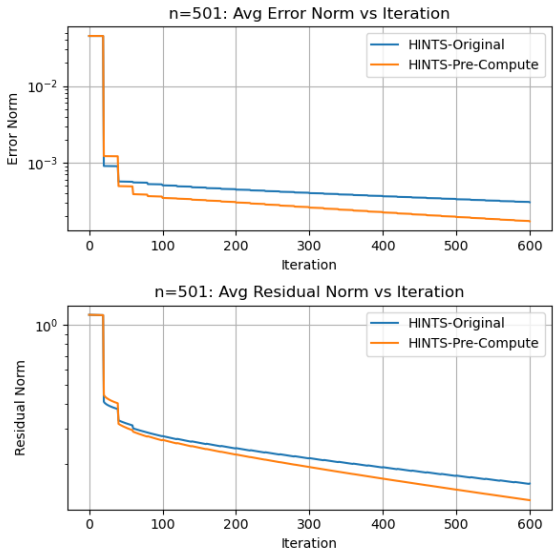
confirm that the precompute strategy can be advantageous in the case where a mid-range frequency plateau exists.



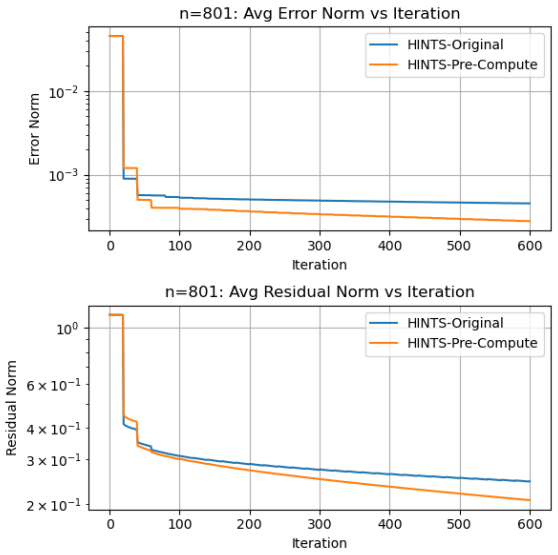
(a) n = 81



(b) n = 201



(c) n = 501



(d) n = 801

Figure 7.2: Error and residual norms of HINTS with (orange) and without pre-compute (blue) for different grid sizes.

We also embedded the fine-tuned DeepONet into a HINTS-MG framework and compared it with the baseline HINTS-MG and the classical multigrid (MG) method. Figure 7.3 shows the error norm versus multigrid cycles for grid size $n = 2001$. Results show that MG-HINTS substantially outperforms classical MG in all three cases. However, the extra performance gain from the pre-compute strategy is marginal.

In conclusion, the pre-compute strategy efficiently bridges the distribution gap for single-level HINTS, especially on finer grids, where the gap between the distribution of residual functions and the original training data becomes more significant. However, in the HINTS-MG framework, its effect is negligible, indicating that the pre-compute resampling strategy is redundant once the multilevel hierarchy is considered.

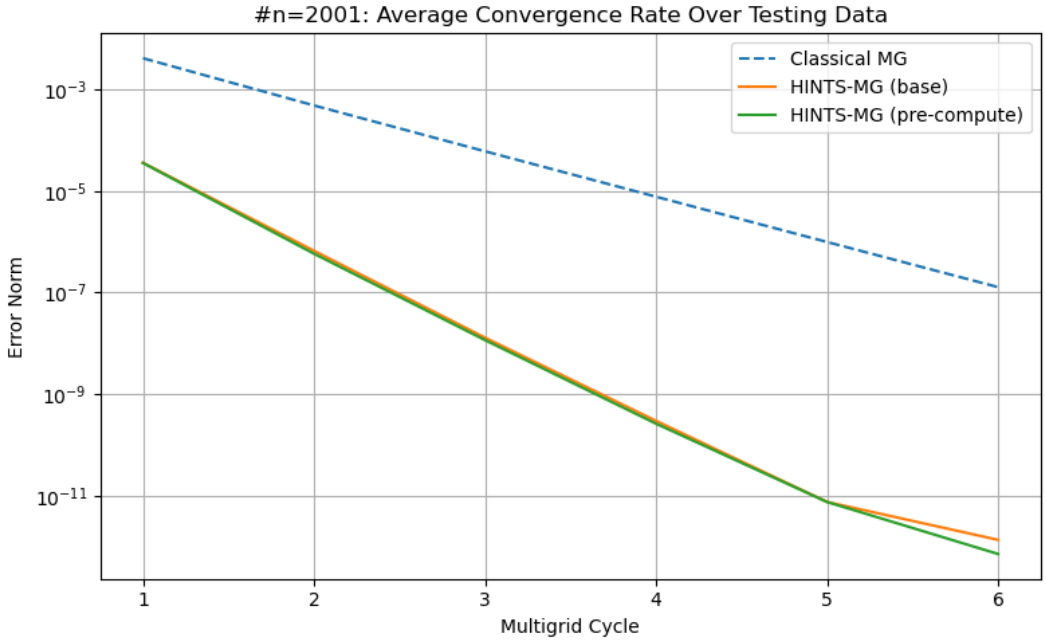


Figure 7.3: Comparison of performance of MG-HINTS with and without the pre-compute strategy on the 1D Poisson Equation with grid size $n = 2001$.

7.3.2. Online In-loop Strategy

In this section, we investigate the performance of the in-loop strategy, evaluating different loss functions and look-ahead sizes on 1D Poisson equations from the testing data.

Training and Validation Loss

We employ the same training dataset as in previous sections 4.5.1 and compare three distinct loss functions—solution loss, residual loss, and convergence loss—as described in Section 7.2.2 with various look-ahead sizes $K \in \{1, 3, 5\}$. Figure 7.4 reports the validation loss on the same validation set, defined as the MSE loss between the model prediction and the reference solution

$$\mathcal{L}_{\text{val}} = \frac{1}{N_{\text{val}}} \sum_{i=1}^{N_{\text{val}}} \|\text{DON}_\theta(f_i) - u_i^\star\|^2.$$

As observed in Figure 7.4, employing the solution loss in the in-loop strategy achieves a performance comparable to the baseline DeepONet model, with a slight improvement seen at a look-ahead size of 5. For the solution loss with a look-ahead size of 1, the DeepONet model is theoretically equivalent to the baseline DeepONet model, as they use the same data and loss function.

In contrast, models trained using residual or convergence losses exhibit higher validation losses, as they do not directly penalize the solution deviation. Larger look-ahead sizes ($K = 3, 5$) for the convergence loss improve the performance of solution prediction in the validation set, yet remain worse than the baseline model.

Training Cost

Table 7.1 reports the training cost of the in-loop strategy with different look-ahead sizes K, alongside the conventionally trained baseline DeepONet, where all runs are performed on a single NVIDIA GeForce RTX 4080 GPU.

The per-epoch training cost and peak memory of the in-loop strategy grows approximately linearly with the look-ahead size K (about 0.032 s and 4.35 MB per extra look-ahead step). The baseline DeepONet has a lower per-epoch cost and peak memory. For the inference stage, since the trained DeepONet will be applied separately, there is no difference among the baseline and in-loop strategies.

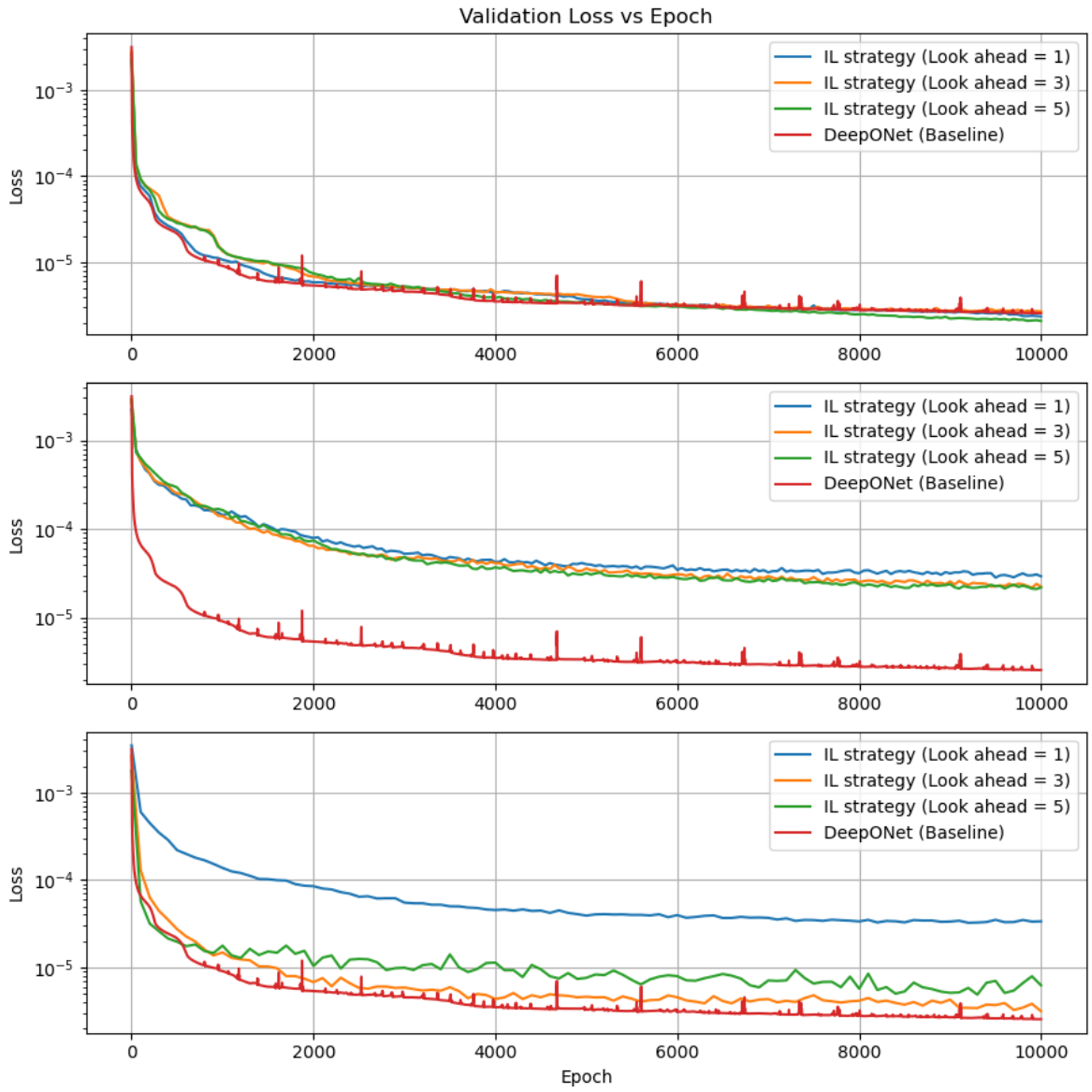


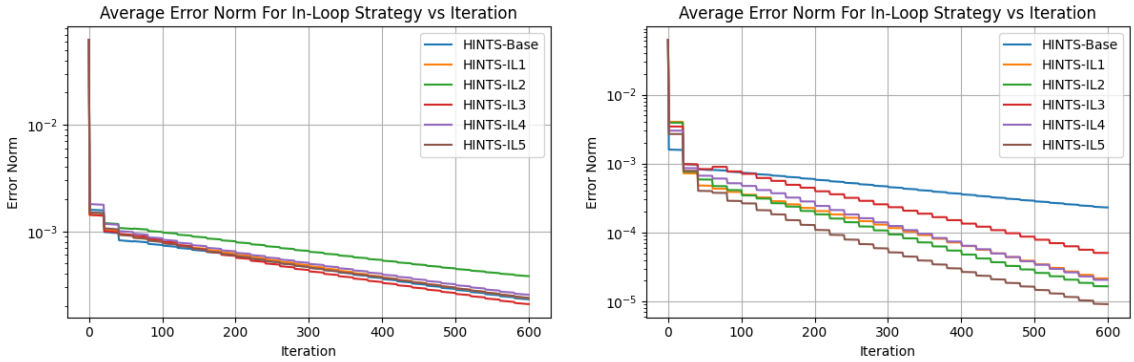
Figure 7.4: Losses on the validation set $\mathcal{L}_{\text{val}}$ of the in-loop strategy with different loss functions (from top to bottom: Solution, Residual, Convergence loss) and look-ahead sizes $K \in \{1, 3, 5\}$, compared with standard DeepONet training (red).

Table 7.1: Training cost of the in-loop strategy with different look-ahead sizes K.

Method	Time/epoch (s)	Peak GPU mem (MB)
In-loop (K = 1)	0.0510	31.1
In-loop (K = 2)	0.0869	34.8
In-loop (K = 3)	0.1205	39.3
In-loop (K = 4)	0.1552	43.9
In-loop (K = 5)	0.1808	48.5
Baseline DeepONet	0.01795	19.9

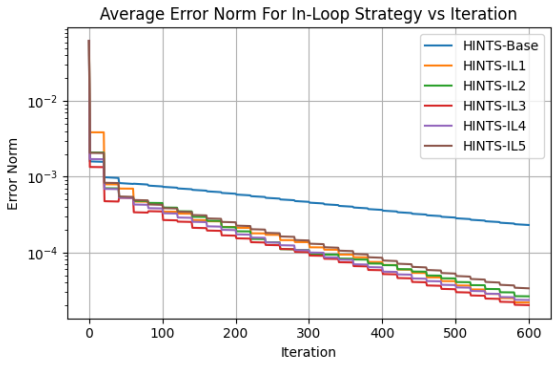
Impact on HINTS Performance

Figure 7.5 illustrates the convergence histories of HINTS adopting the IL strategy using solution, residual, and convergence losses for the 1D Poisson equation with grid size $n = 301$.

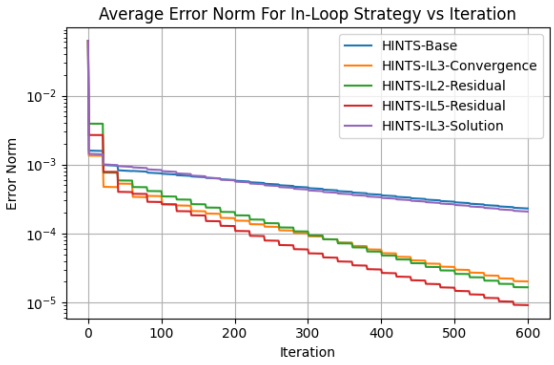


(a) Solution loss

(b) Residual loss



(c) Convergence loss



(d) Comparison of best cases

Figure 7.5: Comparison of convergence performance for HINTS using various in-loop training strategies (solution, residual, and convergence loss). HINTS-Base denotes the baseline HINTS model trained with the MSE loss (Eq. (4.11)) and without using the in-loop strategy; HINTS-ILk denotes the model using the in-loop strategy with a look-ahead size $K{=}k$. All experiments are conducted with the same grid ($n = 301$) and the same interval of DeepONet update $n_r = 20$

Solution loss When the DeepONet model is trained with the solution loss, all models trained with the IL strategies perform similarly to the baseline model. Among them, the model trained by the IL strategy with the look-ahead size $K = 3$ achieves the fastest decay, slightly faster than all other models.

Residual loss Training on the residual loss leads to a pronounced acceleration, with all models trained with the residual loss achieving faster decay than the baseline model. The model using the IL strategy with the look-ahead size $K = 5$ achieves the fastest decay and is significantly faster than the other models using smaller look-ahead sizes $K = 1, 2, 3, 4$.

Convergence loss Using the convergence loss for all look-ahead sizes K also outperforms the baseline model. The model using the IL strategy with the look-ahead size $K = 3$ achieves the best performance, while performance gains over all other models using the IL strategy are small.

Figure 7.5d reports the best-case comparison for models using different loss functions. The in-loop model trained with residual loss and look-ahead size $K = 5$ achieves the best performance, followed by the residual loss with $K = 2$ (”IL2-Residual”) and the convergence loss with $K = 3$ ”IL3-Convergence”. Although models trained with residual and convergence losses exhibit worse performance in predicting the solution in the validation set than models using the solution loss, these models significantly outperform models using the solution loss in the HINTS framework. Even ”IL1” models trained with the residual or convergence loss significantly outperform the baseline model. These results indicate that residual-based objectives are better aligned with the iterative goal of HINTS than the MSE of solutions, which further facilitates the IL strategy in achieving better performance.

Impact on HINTS-MG Performance

We further evaluate the in-loop strategy within the MG-HINTS framework on a 1D Poisson equation with grid size $n = 2001$. The corresponding results are shown in Figure 7.6. Similar to the pre-compute strategy, the performance gain introduced by the in-loop strategy in this experiment is limited when integrated into the HINTS-MG framework.

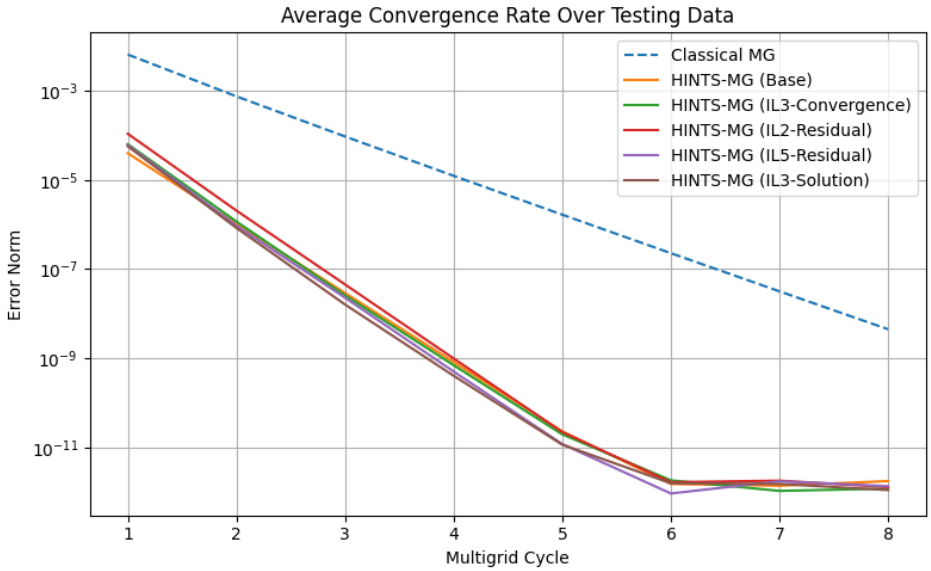


Figure 7.6: Performance comparison of MG-HINTS using different models trained by in-loop strategies and the classical MG method.

In summary, the in-loop strategy markedly improves the performance of single-level HINTS when using residual or convergence losses, reflecting its effectiveness in bridging the data distribution mismatch problem during the iterative solving stage. However, in the HINTS-MG context, these benefits become less pronounced.

8

Conclusion

This thesis investigated the HINTS framework [19], a novel, hybrid, and iterative solver that combines advantages from stationary iterative solvers and the Deep Operator Network (DeepONet) to achieve faster convergence in solving linear systems derived from partial differential equations (PDEs). Aiming to further enhance the HINTS framework, our research focused on the following four research questions with theoretical analysis and numerical experiments.

8.1. Research Questions

Research Question 1: *What is the actual performance of HINTS compared with classical stationary iterative methods?*

In Chapter 4, we validated the performance of HINTS through numerical experiments on benchmark problems. Compared with stationary iterative methods, the HINTS framework exhibits significantly faster convergence in solving Poisson equations. For challenging indefinite Helmholtz equations, although not robust, the HINTS framework maintained convergence on some problems where stationary iterative methods diverged and showed better convergence behavior. In addition, the thesis also validates the performance of HINTS-MG. Comparing the ”smoother-only” strategy with a direct solver on the coarsest grid , we verified its accelerated performance on Helmholtz equations.

Research Question 2: *Why is the HINTS method able to accelerate solving PDEs, and what are its current limitations?*

We explained the iterative process of HINTS from two perspectives: the superposition principle and neural preconditioning. Within the HINTS framework, the DeepONet can be viewed not only as an approximation of the solution operator but also as a preconditioner for the Richardson method acting on low-frequency components of the error. However, two main limitations exist for this approach:

- Spectral bias: As an approximation of the solution operator, the spectral bias of DeepONet prevents it from effectively eliminating error components beyond the low-frequency range. This leads to the gradual accumulation of mid-frequency error components that cannot be efficiently handled by either the stationary iterative method or the DeepONet, finally resulting in the ”slow convergence plateau” of HINTS in later iterations.
- Data distribution shift: The accumulation of mid-frequency components leads to a discrepancy between data distributions in the training and inference stages. The DeepONet is trained on low-frequency-dominated data, but has to deal with mid-frequency-dominated residuals in the iterative process of HINTS, resulting in a decline in network performance.

Research Question 3: *How can the spectral bias in the DeepONet be mitigated to further enhance the performance of HINTS?*

To mitigate the common issue of spectral bias in deep neural networks, motivated by the anti-frequency principle, we introduced a penalty on the first-order gradient term during training. This encourages the DeepONet to prioritize higher-frequency features, thereby improving the overall performance of HINTS.

Research Question 4: *How to mitigate the issue of residual distribution shift in the HINTS method?*

For the problem of data distribution shift, we introduced the ”HINTS-in-the-loop” training strategy, which allows the model to be aware of the actual distribution of residuals in the inference stage, thus improving the performance of HINTS. Two specific strategies were explored in this thesis:

- Pre-compute: An offline method where a pre-trained DeepONet model is applied in the HINTS framework and collect actual residual data to fine-tune the model.
- In-loop: An online approach embedding the whole HINTS iterative process as a differentiable, computational graph, which is then optimized over multiple HINTS steps.

Experiments demonstrated that both strategies can alleviate the problem of data distribution shift and improve the convergence speed of HINTS.

8.2. Future Work

This thesis has shown several limitations and improvements of the HINTS framework, yet several questions remain. This section will briefly discuss several directions and questions that were not explored in this report for future research.

Generalization to other problems The current research was primarily focused on one-dimensional Poisson equations on structured grids. The next step is to extend these strategies to more complex problems such as indefinite Helmholtz problems, two- and three-dimensional domains, and unstructured grids. Besides, for indefinite Helmholtz equations where this study only offered preliminary insights on the relationship between training data quality and robustness, enhancing the robustness of HINTS on indefinite Helmholtz equations is an important direction for future work.

Improvement of HINTS-MG While the proposed strategies significantly enhanced the performance of HINTS, their performance gains were limited when used in the HINTS-MG framework. Future research could investigate the robustness issues of HINTS-MG in the Helmholtz problem, analyzing the conditions that lead to divergence. Beyond improving the smoother within the multigrid framework, another promising direction is to use DeepONet to learn other components of the multigrid framework, such as the interpolation and restriction operators [38].

Adaptive Correction and Advanced Network Architectures: The current HINTS framework applies the DeepONet correction at a fixed interval, as in Zhang et al. [19]. However, as the residual shifts towards the mid-frequency range in later iterations, the standard DeepONet may become less effective and even have a negative impact. Future work should explore adaptive strategies to determine when to use the DeepONet model. This could be complemented by developing specialized networks for handling mid-frequency errors or by integrating alternative neural operators, such as the Fourier Neural Operator [15], to enhance performance.

Refinement of the Interpolation Scheme: Following Zhang et al. [19], this thesis adopted the same linear interpolation to map residuals to sensor points of the DeepONet. Since the accuracy of this projection directly impacts the network’s input, thus influencing the quality of the subsequent correction. A systematic investigation into the effects of different interpolation methods is warranted.

In conclusion, this thesis positions HINTS as a substantive bridge between classical solvers and operator-learning networks. By revealing its convergence plateau, recasting the DeepONet in a neural preconditioning view, and introducing GE-HINTS and HINTS-in-the-loop training strategies, this thesis provides both mechanistic understanding and practical recipes for accelerating linear PDE solvers. We hope these insights will aid researchers seeking effective hybrid iterative solvers and will contribute to further progress in this area.

A

Declaration

A.1. AI Disclosure Statement

During the preparation of this thesis, the author has utilized ChatGPT to assist with translating, grammar and spelling checks, and enhancing language. After using this tool, the author thoroughly reviewed, edited, and revised the content as needed. The author takes full responsibility for the content and conclusions presented in this thesis.

A.2. Code Availability Statement

The code used to reproduce all the numerical experiments and related algorithms in this thesis will be publicly available on GitHub at `https://github.com/HNU-WYH/HINTS_in_PDEs_Yuhan_Wu` before defense.

Bibliography

[1] Anthony T Patera. “A spectral element method for fluid dynamics: laminar flow in a channel expansion”. In: *Journal of computational Physics* 54.3 (1984), pp. 468–488.

[2] Singiresu S Rao. *The finite element method in engineering*. Elsevier, 2010.

[3] Michel Rappaz et al. *Numerical modeling in materials science and engineering*. Vol. 20. Springer, 2003.

[4] Theodore L Bergman. *Fundamentals of heat and mass transfer*. John Wiley & Sons, 2011.

[5] Henk A Van der Vorst. *Iterative Krylov methods for large linear systems*. 13. Cambridge University Press, 2003.

[6] Oliver G Ernst and Martin J Gander. “Why it is difficult to solve Helmholtz problems with classical iterative methods”. In: *Numerical analysis of multiscale problems* (2011), pp. 325–363.

[7] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.

[8] Radii Petrovich Fedorenko. “A relaxation method for solving elliptic difference equations”. In: *USSR Computational Mathematics and Mathematical Physics* 1.4 (1962), pp. 1092–1096.

[9] Youcef Saad and Martin H Schultz. “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems”. In: *SIAM Journal on scientific and statistical computing* 7.3 (1986), pp. 856–869.

[10] Jonathan Richard Shewchuk et al. “An introduction to the conjugate gradient method without the agonizing pain”. In: (1994).

[11] Christopher Rackauckas et al. “Universal differential equations for scientific machine learning”. In: *arXiv preprint arXiv:2001.04385* (2020).

[12] George Em Karniadakis et al. “Physics-informed machine learning”. In: *Nature Reviews Physics* 3.6 (2021), pp. 422–440.

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

[14] Lu Lu et al. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature machine intelligence* 3.3 (2021), pp. 218–229.

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

[16] Zhi-Qin John Xu, Yaoyu Zhang, and Tao Luo. “Overview frequency principle/spectral bias in deep learning”. In: *Communications on Applied Mathematics and Computation* (2024), pp. 1–38.

[17] Sifan Wang, Xinling Yu, and Paris Perdikaris. “When and why PINNs fail to train: A neural tangent kernel perspective”. In: *Journal of Computational Physics* 449 (2022), p. 110768.

[18] Sifan Wang, Hanwen Wang, and Paris Perdikaris. “Improved architectures and training algorithms for deep operator networks”. In: *Journal of Scientific Computing* 92.2 (2022), p. 35.

[19] Enrui Zhang et al. “Blending neural operators and relaxation methods in PDE numerical solvers”. In: *Nature Machine Intelligence* (2024), pp. 1–11.

[20] Patrick R Amestoy et al. “A fully asynchronous multifrontal solver using distributed dynamic scheduling”. In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[21] Timothy A Davis. “Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method”. In: *ACM Transactions on Mathematical Software (TOMS)* 30.2 (2004), pp. 196–199.

[22] Olaf Schenk and Klaus Gärtner. “Solving unsymmetric sparse systems of linear equations with PARDISO”. In: *Future Generation Computer Systems* 20.3 (2004), pp. 475–487.

[23] COMSOL BV and COMSOL OY. “COMSOL Multiphysics Reference Guide© COPYRIGHT 1998–2010 COMSOL AB.” In: (1998).

[24] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[25] John Jumper et al. “Highly accurate protein structure prediction with AlphaFold”. In: *nature* 596.7873 (2021), pp. 583–589.

[26] Peiyuan Jiang et al. “A Review of Yolo algorithm developments”. In: *Procedia computer science* 199 (2022), pp. 1066–1073.

[27] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[28] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[29] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[30] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[31] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[32] Ameya D Jagtap, Ehsan Kharazmi, and George Em Karniadakis. “Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems”. In: *Computer Methods in Applied Mechanics and Engineering* 365 (2020), p. 113028.

[33] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[34] George Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[35] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural networks* 2.5 (1989), pp. 359–366.

[36] Moshe Leshno et al. “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function”. In: *Neural networks* 6.6 (1993), pp. 861–867.

[37] Tianping Chen and Hong Chen. “Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems”. In: *IEEE transactions on neural networks* 6.4 (1995), pp. 911–917.

[38] Alena Kopaničáková and George Em Karniadakis. “Deeponet based preconditioning strategies for solving parametric linear systems of equations”. In: *SIAM Journal on Scientific Computing* 47.1 (2025), pp. C151–C181.

[39] Bar Lerer, Ido Ben-Yair, and Eran Treister. “Multigrid-augmented deep learning for the helmholtz equation: Better scalability with compact implicit layers”. In: *arXiv preprint arXiv:2306.17486* (2023).

[40] Chen Cui et al. “Fourier neural solver for large sparse linear algebraic systems”. In: *Mathematics* 10.21 (2022), p. 4014.

[41] Yogi A Erlangga, Cornelis Vuik, and Cornelis Willebrordus Oosterlee. “On a class of preconditioners for solving the Helmholtz equation”. In: *Applied Numerical Mathematics* 50.3-4 (2004), pp. 409–425.

[42] Vandana Dwarka and Cornelis Vuik. “Scalable convergence using two-level deflation preconditioning for the Helmholtz equation”. In: *SIAM Journal on Scientific Computing* 42.2 (2020), A901–A928.

[43] Jinqiang Chen, Vandana Dwarka, and Cornelis Vuik. “A matrix-free parallel two-level deflation preconditioner for two-dimensional heterogeneous Helmholtz problems”. In: *Journal of Computational Physics* 514 (2024), p. 113264.

[44] Matthias Seeger. “Gaussian processes for machine learning”. In: *International journal of neural systems* 14.02 (2004), pp. 69–106.

[45] Lawrence C Evans. *Partial differential equations*. Vol. 19. American Mathematical Society, 2022.

[46] Vladislav Trifonov et al. “Learning from linear algebra: A graph neural network approach to preconditioner design for conjugate gradient solvers”. In: *arXiv preprint arXiv:2405.15557* (2024).

[47] Paul Häusner, Ozan Öktem, and Jens Sjölund. “Neural incomplete factorization: learning preconditioners for the conjugate gradient method”. In: *arXiv preprint arXiv:2305.16368* (2023).

[48] Yichen Li et al. “Learning preconditioners for conjugate gradient PDE solvers”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 19425–19439.

[49] Michael F Hutchinson. “A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines”. In: *Communications in Statistics-Simulation and Computation* 18.3 (1989), pp. 1059–1076.

[50] Chao Ma, Lei Wu, et al. “Machine learning from a continuous viewpoint, I”. In: *Science China Mathematics* 63.11 (2020), pp. 2233–2266.

[51] L Lu et al. *A deep learning library for solving differential equations*. 2020.

[52] Kiwon Um et al. “Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers”. In: *Advances in neural information processing systems* 33 (2020), pp. 6111–6122.