



*Research article***A parallel variable-order iteration algorithm for solving continuous integral equations using deep learning****Zhiyuan Ren^{1,*}, Ruilong Yu², Yi Zeng², Shijie Zhou² and Qihe Liu²**

¹ School of Computer and Network Security, Chengdu University of Technology, No. 1, Erxianqiao East Third Road, Chenghua District, Chengdu 610059, China

² School of Information and Software Engineering, University of Electronic Science and Technology of China, No. 5, Section 2, Jianshe North Road, Chenghua District, Chengdu 610054, China

* **Correspondence:** Email: renzhiyuan@uestc.edu.cn.

Abstract: Deep learning has become a key approach for solving integral equations, a central theme in numerical computation. However, existing methods often struggle with precision and efficiency. This study presents the parallel variable-order iteration algorithm (P-VOI), inspired by the variable-order method in the R-AIM algorithm, specifically designed to resolve continuous integral equations using deep learning. The P-VOI algorithm transforms these equations into differential forms by applying the Newton-Leibniz formula. It employs parallel iterative processes, allowing neural networks to approximate the unknown and primitive functions simultaneously. This integration of parallel iteration significantly enhances computational efficiency compared to R-AIM. Additionally, the P-VOI algorithm uses neural networks as trial functions during training, addressing challenges that arise when integral kernels have zeros. This method reduces systematic errors from finite-sum representations, improving solution accuracy. Empirical evaluations demonstrate that the P-VOI algorithm achieves superior convergence accuracy (mean squared error in the range of 10^{-7} to 10^{-9}) and efficiency, with speedups of $2.9\times$ to $5.2\times$ over R-AIM. Compared with non-variable-order methods, this corresponds to an improvement of approximately three orders of magnitude in accuracy based on MSE; relative to the direct variable-order baseline R-AIM, the accuracy remains of the same order of magnitude with slightly higher precision.

Keywords: deep learning; continuous integral equations; parallel iteration; variable-order transformation; speedup ratio

Mathematics Subject Classification: 65R20, 45B05, 45D05, 68T07, 68Q12

1. Introduction

Integral equations serve as the foundations of linear algebra and modern functional analysis, finding extensive applications across various scientific and technological domains such as aerodynamics [1], elasticity [2], fracture mechanics [3], thermodynamics [4], fluid dynamics [5], vibration theory [6], neutron transport theory [7], quantum field theory [8], and dispersion theory [9]. They play a pivotal role in both scientific research and engineering applications. However, many integral equations with practical applications resist direct analytical solutions, and some lack analytical solutions altogether. Consequently, researchers often use approximation or numerical methods. The numerical solution of solving integral equations has long been a challenging focus in numerical computation. Two primary numerical computation strategies exist: the conventional classical numerical computation methods encompassing finite difference, finite element, and finite volume, and the emerging artificial intelligence numerical computation methods epitomized by deep learning.

Traditional classical numerical methods for solving integral equations encompass a variety of approaches, such as Monte Carlo methods [10], finite difference approximation methods [9, 11], successive approximation methods [9, 11], functional correction averaging methods [9, 11], undetermined coefficient methods [9, 11], and number-theoretic grid methods [12]. A comprehensive treatment of these classical methods can be found in [13].

AI-based numerical methods for integral equations can be classified as variable-order or non-variable-order, depending on how the integral term is handled. These methods are part of the broader field of physics-informed machine learning [14], which integrates physical principles with deep learning architectures. The foundational work on physics-informed neural networks (PINNs) [15] paved the way for solving various scientific computing problems, including integral equations. Non-variable-order methods resolve integral equations by substituting the integral terms with finite sums [16–22]. In contrast, variable-order methods eliminate the integral terms of the equations through variable-order transformations [23–25]. Experimental results indicate that both non-variable-order and variable-order methods are viable. However, non-variable-order methods exhibit low computational accuracy, with standard deviations ranging from 10^{-2} to 10^{-3} [16, 19]. This reduced accuracy primarily stems from systematic errors introduced when integral terms are approximated by finite sums, which adversely affects the convergence of the deep learning models. Although variable-order methods mitigate these systematic errors and enhance the computational accuracy of integral equations, their efficiency in solving multiple-term multiple integral equations remains suboptimal. This inefficiency is largely attributed to the alternating iterative approach employed by deep learning to address integral equations. Recent advances in parallel and distributed deep learning [26] provide useful foundations for improving training efficiency in such multi-network settings.

To address these challenges, this paper presents a parallel variable-order iteration deep learning algorithm (P-VOI) for solving continuous integral equations. The continuous integral equations are transformed into a system of differential equations consisting of the unknown function, the original function, and its partial derivatives through a variable-order transformation utilizing the Newton-Leibniz formula. This system of differential equations is subsequently solved using a parallel iterative deep learning method. The effectiveness and efficiency of the P-VOI algorithm are validated by comparing its solution accuracy and computational efficiency with other variable-order methods under identical experimental conditions for five continuous integral equations across four categories:

Fredholm, Volterra, Fredholm-Volterra, and nonlinear integral equations.

In conclusion, this paper makes the following contributions:

- The differential equations derived from the variable-order transformation consist solely of differential forms, thereby omitting integral forms. This effectively eliminates the systematic errors associated with representing integral terms as finite sums, significantly enhancing the solution accuracy of the integral equations.
- We detail the construction of the governing equation, the transformation equations, the constraints for the definite solution conditions of the primitive functions, and the final weighted loss function based on L2 loss.
- We adopt a coarse-grained parallel training scheme to reduce serial waiting time. Since the variable-order transformation yields multiple primitive-function networks that are mutually independent within each epoch, their training tasks can be executed concurrently as separate MATLAB processes on a shared GPU. This engineering-oriented modification improves wall-clock efficiency without changing the underlying mathematical formulation.
- A neural network is employed as a trial function for the unknown function during model training. This approach enables a direct approximation of the unknown function via the neural network, circumventing the issue that arises when the unknown function cannot be directly derived from the primitive function of the integrand, which occurs when the integral kernel has zeros.

The remainder of this paper is organized as follows. In Section 2, we define the problem of solving multiple polynomial integral equations. In Section 3, we provide an overview of related work. In Section 4, we introduce the parallel variable-order transformation, loss function construction, and parallel iteration. In Section 5, we present the theoretical analysis including equivalence, error bounds, and convergence. In Section 6, we detail the experimental environment, experimental setup, test equations, experimental results, and result analysis. In Section 7, we conclude with the contributions of our work and shed light on future research directions.

2. Problem formulation

The general form of a continuous integral equation is

$$\alpha(\mathbf{X})\phi(\mathbf{X}) = f(\mathbf{X}) + \left(\sum_{i=1}^I \lambda_i \int_{a_{i,n}}^{x'_{i,n}} \cdots \int_{a_{i,2}}^{x'_{i,2}} \int_{a_{i,1}}^{x'_{i,1}} [K_i(\mathbf{X}, \mathbf{T}) \cdot M[\phi(\mathbf{T})] dt_1 dt_2 \cdots dt_n \right]. \quad (2.1)$$

Equation (2.1) represents a multiple-term, multiple integral form, involving the n -dimensional vectors $\mathbf{X}$, $\mathbf{X}'$, and $\mathbf{T}$, where $\mathbf{X} = (x_1, x_2, \dots, x_n)$, $\mathbf{X}' = (x'_{i,1}, x'_{i,2}, \dots, x'_{i,n})$, $\mathbf{T} = (t_1, t_2, \dots, t_n)$. Here, $x'_{i,1}, x'_{i,2}, \dots, x'_{i,n} \in \mathbf{X}$, meaning each $x'_{i,m}$ ($i = 1, 2, \dots, I$; $m = 1, 2, \dots, n$) is an element of the vector $\mathbf{X}$, and the elements of $\mathbf{X}'_i$ are not necessarily distinct from those in $\mathbf{X}$. λ_i ($i = 1, 2, \dots, I$) are parameters, while $\alpha(\mathbf{X})$, $f(\mathbf{X})$, and the integral kernel $K_i(\mathbf{X}, \mathbf{T})$ ($i = 1, 2, \dots, I$) are known functions. $\phi(\mathbf{X})$ is the unknown function, and $M[\phi(\mathbf{T})]$ represents a known functional of $\phi(\mathbf{T})$. When $M[\phi(\mathbf{T})]$ is a linear functional of $\phi(\mathbf{T})$, Eq (2.1) is a linear integral equation, and $M[\phi(\mathbf{T})]$ can be directly expressed in terms of $\phi(\mathbf{T})$. When $M[\phi(\mathbf{T})]$ is a nonlinear functional of $\phi(\mathbf{T})$, Eq (2.1) is a nonlinear integral equation. If the upper $x'_{i,m}$ and lower limits $a_{i,m}$ ($i = 1, 2, \dots, I$; $m = 1, 2, \dots, n$) of integration are both

constants, Eq (2.1) is classified as a Fredholm integral equation. When the upper $x'_{i,m}$ and lower limits $a_{i,m}$ of integration contain variables, Eq (2.1) is a Volterra integral equation. If Eq (2.1) contains both Fredholm and Volterra integral equation terms, it is referred to as a Fredholm-Volterra integral equation. When the integral kernel $K_i(\mathbf{X}, \mathbf{T})(i = 1, 2, \dots, I)$ is a degenerate kernel [9], i.e., $K_i(\mathbf{X}, \mathbf{T}) = p_i(\mathbf{X})q_i(\mathbf{T})$, Eq (2.1) takes the form of a degenerate kernel:

$$\alpha(\mathbf{X})\phi(\mathbf{X}) = f(\mathbf{X}) + \left(\sum_{i=1}^I \lambda_i p_i(\mathbf{X}) \int_{a_{i,n}}^{x'_{i,n}} \dots \int_{a_{i,2}}^{x'_{i,2}} \int_{a_{i,1}}^{x'_{i,1}} [q_i(\mathbf{T}) \cdot M[\phi(\mathbf{T})] dt_1 dt_2 \dots dt_n] \right). \quad (2.2)$$

3. Related work

Before discussing the specific methods for solving integral equations, it is useful to situate the present work within the broader context of parallel intelligent numerical computation. Recent research has increasingly explored the integration of deep learning with physical constraints and biologically inspired algorithms to achieve efficient, interpretable, and robust computation. For example, a physics-informed interpretable AI framework was developed for transparent process control in wastewater treatment plants by embedding domain knowledge into the learning process [27]. Similarly, bio-inspired computation was advanced through algorithms based on DNA molecule computation for solving combinatorial problems such as the assignment problem [28]. Although these works target different application domains, they share a common philosophy: incorporating physical or biological principles into computational intelligence to overcome the limitations of purely data-driven black-box models. Our P-VOI algorithm follows the same intellectual trajectory by embedding the Newton-Leibniz transformation (a mathematical constraint derived from calculus) into a coarse-grained parallel training scheme that exploits the independence of primitive function networks. This positioning highlights that P-VOI is not an isolated development but part of a wider movement toward parallel intelligent numerical computation.

Deep learning, a subfield of machine learning, primarily facilitates data analysis and pattern recognition through the construction and training of deep neural networks. Its fundamental principle involves integrating multiple layers of neurons to learn complex features within the data, thereby enabling effective data representation. The significance of deep learning in numerical computation is highlighted by its capacity to manage high-dimensional data efficiently, effectively address the curse of dimensionality, and execute rapid calculations on large-scale datasets. This capability results in enhanced computational efficiency and accuracy compared to traditional algorithms. Based on the treatment of integral terms in the loss function of the governing equations, deep learning approaches for solving integral equations can be categorized into two distinct types: non-variable-order methods and variable-order methods.

It should be noted that this two-category classification, while helpful for organizing methods according to how the integral term is handled, is by no means exhaustive. Recent advances such as operator learning (e.g., DeepONet [29] and Fourier neural operators [30]) and differentiable physics layers [31] provide alternative paradigms that bypass the explicit discretization or transformation of the integral term altogether. Likewise, numerous PINN variants have been proposed for improved training dynamics, including self-adaptive weighting and extended architectures [32,33]. These approaches are largely orthogonal to the variable-order transformation used in this work and may be integrated with P-VOI in future research. The classification adopted here serves only to highlight the specific design

choice addressed by our method, namely the elimination of integral terms through the Newton-Leibniz transformation.

Currently, a limited body of research, both domestically and internationally, explores the use of deep learning methods for solving integral equations. This section provides a comprehensive overview of the relevant literature, emphasizing key research questions, employed methodologies, principal findings, conclusions drawn, and the limitations of the studies reviewed.

3.1. Non-variable methods

The fundamental concept of the non-variable-order method for solving integral equations with deep learning is to replace integral terms with finite sums. This approach has roots in earlier work using artificial neural networks to solve differential equations [34]. More recently, researchers have applied similar concepts to integral equations. For instance, Jafarian et al. [35] developed ANN-based modeling for solving Volterra integral equation systems, demonstrating the potential of neural networks in this domain.

Zhang et al. [16] introduced a numerical method utilizing residual networks in TensorFlow to solve Fredholm integral equations, focusing primarily on this class of equations. By randomly generating training datasets and optimizing the solutions using neural networks, the method achieved significant convergence accuracy. The study concluded that this method is effective and stable, indicating broad application prospects. However, it does not extensively address the applicability and limitations of the technique for more complex problems.

Guan et al. [19] proposed a deep learning approach to solving high-dimensional Fredholm integral equations. This method involved constructing a model using deep residual neural networks and training the network by minimizing the loss function to approximate the solution of the integral equation. Experimental results demonstrated this approach's efficiency and strong generalization capability, effectively mitigating the curse of dimensionality. Nonetheless, limitations may arise from gradient vanishing as network depth increases.

Guo et al. [17] presented a method based on physics-informed deep neural networks for solving volume integral equations in a two-dimensional context. This approach accelerates the solution of volume integral equations using iterative deep neural networks, integrating learnable parameters with physical principles. Key findings highlighted the network's accuracy, efficiency, and adaptability to variations in input size and frequency, emphasizing the potential of combining learning parameters with physical laws for rapid computations.

Xu et al. [20] introduced a method for solving linear Fredholm integro-differential equations using feedforward neural networks. This approach employs neural networks to approximate solutions, utilizing Taylor expansions to replace unknown functions while neural networks learn the coefficients. The primary finding compares the effectiveness of this method to traditional numerical approaches, asserting the feasibility of the proposed technique. However, potential limitations include concerns regarding the generalizability to specific problems and computational efficiency.

Sun et al. [21] developed a deep learning method, BINN, for solving computational mechanics problems using boundary integral equations. The research focused on resolving boundary value problems by approximating unknown quantities with neural networks during training. Key findings indicated advantages such as dimension reduction, applicability to complex geometries, and suitability for infinite or semi-infinite domain problems, underscoring the potential and versatility of BINN.

Zeng et al. [25] investigated a neural network approach utilizing orthogonal Legendre polynomials to solve stochastic Itô-Volterra integral equations. By transforming these equations into algebraic systems, constructing a loss function, and employing neural networks for weight learning, they proposed a method for solving multidimensional stochastic integral equations. The research demonstrated accurate performance in numerical examples; however, its limitations may include restricted applicability to complex problem solutions. The conclusion emphasized this method's potential and constraints, offering a novel approach for approximating solutions to stochastic integral equations.

Due to the accumulation of errors associated with substituting integral terms with finite sums, the non-variable-order method frequently yields low computational accuracy.

3.2. Variable-order methods

The fundamental concept of the variable-order method involves eliminating integral terms by applying the fundamental theorem of calculus (varying the order of integration). This approach aligns with advanced neural network architectures for solving differential equations, such as the deep Galerkin method (DGM) [36] and neural ordinary differential equations (Neural ODEs) [37]. Recent research in applied intelligence has further advanced these techniques; for example, Luo et al. [38] proposed an enhanced hybrid adaptive physics-informed neural network that shows improved performance for forward and inverse PDE problems.

Liu et al. [23] introduced a deep learning-based approach for solving linear integral equations by transforming them into differential equations using primitive functions. This research was motivated by the critical role of integral equations in science and engineering, where traditional numerical methods often encounter error-related challenges. The study aimed to improve the accuracy of integral equation solutions through deep learning techniques. The authors proposed a solution method and a framework for generating loss functions, demonstrating that their approach achieved robust accuracy and continuity across multiple examples, thus offering a novel technical pathway. However, potential limitations include restrictions on applicability to specific types of equations and the demands on computational resources.

Liu et al. [24] explored a method for solving multidimensional continuous integral equations by proposing an iterative deep learning algorithm that alternates between successive transformations of primitive functions. They investigated numerical solutions for high-dimensional integral equations utilizing deep learning techniques. This method involved transforming the integrated function into primitive functions and constructing a neural network to approximate the numerical solution of the unknown function. Key findings indicated that the algorithm exhibits good accuracy and continuity, presenting a new approach to multidimensional integral equations. However, the study highlighted the need for further validation regarding the algorithm's convergence and accuracy. Although R-AIM improves accuracy by eliminating integral discretization errors, its alternating iterative scheme updates the primitive-function networks and the unknown-function network sequentially, creating a serial dependency chain that limits computational efficiency. It is important to note that the P-VOI algorithm proposed in the present paper is a direct extension of R-AIM: It inherits the same variable-order transformation and loss construction, but replaces R-AIM's sequential alternating updates with a two-block synchronous parallel scheme, thereby reducing wall-clock training time while preserving accuracy.

Shi et al. [22] proposed a method for solving multiple integral equations based on physics-informed neural networks (PINN). They transformed integral equations into differential equations, designed neural network architectures, and employed PINN to approximate the analytical solutions of these differential equations. A significant finding was that the PINN method does not necessitate mathematical derivations, rendering it simpler than traditional methods. The study concluded that all three methods met the required accuracy standards, with the PINN approach being notably more straightforward. However, the discussion surrounding the stability of the PINN method and its applicability to various types of integral problems was limited, indicating potential constraints in this regard.

By transforming integral terms into differential terms, the variable-order method eliminates the systematic errors associated with non-variable-order methods, enhancing solution accuracy. However, this approach is limited by the assumption of continuity in the integrand, which prevents it from directly solving integral equations that involve discontinuous integrands.

4. Methodology

To facilitate reading, Table 1 summarizes the main notation used throughout this section.

Table 1. Summary of key notation in Section 4.

Symbol	Meaning
$\mathbf{X} = (x_1, \dots, x_n)$	Independent variable vector of the integral equation
$\mathbf{T} = (t_1, \dots, t_n)$	Integration variable vector
$\mathbf{X}'_i = (x'_{i,1}, \dots, x'_{i,n})$	Vector of upper limits of the i -th integral term; each $x'_{i,m}$ is an element of $\mathbf{X}$
$a_{i,m}$	Lower limit of the m -th integration variable in the i -th integral term
$K_i(\mathbf{X}, \mathbf{T})$	Integral kernel of the i -th term
$M[\phi(\mathbf{T})]$	Known functional of the unknown function (linear or nonlinear)
$\phi(\mathbf{X})$	Unknown function to be solved for
$F_{i,m}(\mathbf{X}, \mathbf{T})$	Primitive function obtained after integrating the i -th term m times
$N_\phi(\mathbf{X})$	Neural network trial function for $\phi(\mathbf{X})$
$N_{i,m}(\mathbf{X}, \mathbf{T})$	Neural network trial function for $F_{i,m}(\mathbf{X}, \mathbf{T})$
s	Sample point index; superscript s denotes evaluation at a sample point
$\mathbf{X}^s, \mathbf{T}^s$	The s -th sample point for $\mathbf{X}$ and $\mathbf{T}$
$x'^s_{i,m}$	The value of $x'_{i,m}$ at the s -th sample point
$\varepsilon_\phi, \varepsilon_{i,m}$	Convergence thresholds for the loss functions
$P_{i,m,p}, P_{i,m,d}$	Weights for the transformation loss and fixed-solution loss

With this notation in place, we proceed to describe the variable-order transformation in detail.

4.1. Variable-order transformation

If for the i th integral term $\int_{a_{i,n}}^{x'_{i,n}} \dots \int_{a_{i,2}}^{x'_{i,2}} \int_{a_{i,1}}^{x'_{i,1}} K_i(\mathbf{X}, \mathbf{T}) M[\phi(\mathbf{T})] dt_1 dt_2 \dots dt_n$ in Eq (2.1), its first integral's integrand, $K_i(\mathbf{X}, \mathbf{T}) M[\phi(\mathbf{T})]$, is a continuous function of the variables $\mathbf{X}$ and $\mathbf{T}$, according

to the principles of calculus [39], there exists an integral primitive function $F_{ic,1}(\mathbf{X}, \mathbf{T})$ with respect to t_1 . Then, by the fundamental theorem of calculus [40], we have

$$\begin{aligned} & \int_{a_{i,1}}^{x'_{i,1}} K_i(x_1, x_2, \dots, x_n, t_1, t_2, \dots, t_n) M[\phi(t_1, t_2, \dots, t_n)] dt_1 \\ &= F_{ic,1}(x_1, x_2, \dots, x_n, x'_{i,1}, t_2, \dots, t_n) \\ & - F_{ic,1}(x_1, x_2, \dots, x_n, a_{i,1}, t_2, \dots, t_n) \\ &= F_{ic,1}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n) - F_{ic,1}(\mathbf{X}, a_{i,1}, t_2, \dots, t_n). \end{aligned} \quad (4.1)$$

According to the definition of the original function, the derivative of $F_{ic,1}(\mathbf{X}, \mathbf{T})$ with respect to t_1 satisfies

$$\begin{aligned} F_{ic,1}(\mathbf{X}, \mathbf{T})' &= \frac{dF_{ic,1}(\mathbf{X}, \mathbf{T})}{dt_1} \\ &= K_i(\mathbf{X}, \mathbf{T}) M[\phi(\mathbf{T})]. \end{aligned} \quad (4.2)$$

The primitive function $F_{ic,1}(\mathbf{X}, \mathbf{T})$ is composed of a family of functions that differ by an arbitrary real constant C [39], and all these primitive functions have equal derivatives with respect to t_1 . Let $F_{ie,1}(\mathbf{X}, \mathbf{T})$ be a specific, determined primitive function within $F_{ic,1}(\mathbf{X}, \mathbf{T})$, such that $F_{ic,1}(\mathbf{X}, \mathbf{T}) = F_{ie,1}(\mathbf{X}, \mathbf{T}) + C$, where C is an arbitrary real value. By setting $C = -F_{ie,1}(\mathbf{X}, a_{i,1}, t_2, \dots, t_n)$, we obtain the specific, determined primitive function as

$$\begin{aligned} F_{i,1}(\mathbf{X}, \mathbf{T}) &= F_{ie,1}(\mathbf{X}, \mathbf{T}) - F_{ie,1}(\mathbf{X}, a_{i,1}, t_2, \dots, t_n) \\ &= F_{ie,1}(\mathbf{X}, t_1, t_2, \dots, t_n) - F_{ie,1}(\mathbf{X}, a_{i,1}, t_2, \dots, t_n). \end{aligned} \quad (4.3)$$

From Eq (4.3), it follows that when $t_1 = a_{i,1}$, $F_{i,1}(\mathbf{X}, a_{i,1}, t_2, \dots, t_n) = 0$. Furthermore, we have

$$\begin{aligned} F_{i,1}(\mathbf{X}, \mathbf{T})' &= \frac{dF_{i,1}(\mathbf{X}, \mathbf{T})}{dt_1} \\ &= \frac{dF_{ie,1}(\mathbf{X}, \mathbf{T})}{dt_1} \\ &= K_i(\mathbf{X}, \mathbf{T}) M[\phi(\mathbf{T})]. \end{aligned} \quad (4.4)$$

Substituting Eq (4.3) into Eq (4.1), we obtain

$$\begin{aligned} & \int_{a_{i,1}}^{x'_{i,1}} K_i(x_1, x_2, \dots, x_n, t_1, t_2, \dots, t_n) M[\phi(t_1, t_2, \dots, t_n)] dt_1 \\ &= F_{i,1}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n) - F_{i,1}(\mathbf{X}, a_{i,1}, t_2, \dots, t_n) \\ &= F_{i,1}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n). \end{aligned} \quad (4.5)$$

Substituting Eq (4.5) into its second integral term, and similarly, we can obtain

$$\begin{aligned} & \int_{a_{i,2}}^{x'_{i,2}} \int_{a_{i,1}}^{x'_{i,1}} K_i(\mathbf{X}, \mathbf{T}) M[\phi(\mathbf{T})] dt_1 dt_2 \\ &= \int_{a_{i,2}}^{x'_{i,2}} F_{i,1}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n) dt_2 \\ &= F_{i,2}(\mathbf{X}, x'_{i,1}, x'_{i,2}, t_3, \dots, t_n) \\ & - F_{i,2}(\mathbf{X}, x'_{i,1}, a_{i,2}, t_3, \dots, t_n) \\ &= F_{i,2}(\mathbf{X}, x'_{i,1}, x'_{i,2}, t_3, \dots, t_n). \end{aligned} \quad (4.6)$$

In Eq (4.6), when $t_2 = a_{i,2}$, it is given that $F_{i,2}(\mathbf{X}, x'_{i,1}, a_{i,2}, t_3, \dots, t_n) = 0$ and we have

$$\begin{aligned} F_{i,2}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n)' &= \frac{dF_{i,2}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n)}{dt_2} \\ &= F_{i,1}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n). \end{aligned} \quad (4.7)$$

Similarly, for its n th integral term, we have

$$\begin{aligned} &\int_{a_{i,n}}^{x'_{i,n}} \dots \int_{a_{i,2}}^{x'_{i,2}} \int_{a_{i,1}}^{x'_{i,1}} K_i(\mathbf{X}, \mathbf{T}) M[\phi(\mathbf{T})] dt_1 dt_2 \dots dt_n \\ &= \int_{a_{i,n}}^{x'_{i,n}} F_{i,n-1}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n) dt_n \\ &= F_{i,n}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, x'_{i,n}) \\ &\quad - F_{i,n}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, a_{i,n}) \\ &= F_{i,n}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, x'_{i,n}) \\ &= F_{i,n}(\mathbf{X}, \mathbf{X}'). \end{aligned} \quad (4.8)$$

At this point, when $t_n = a_{i,n}$, $F_{i,n}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, a_{i,n}) = 0$ and we have

$$\begin{aligned} &F_{i,n}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n)' \\ &= \frac{dF_{i,n}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n)}{dt_n} \\ &= F_{i,n-1}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n). \end{aligned} \quad (4.9)$$

By substituting Eqs (4.5), (4.6), and (4.8) into Eq (2.1), we derive a system of differential equations that represents the variable-order transformation of the continuous integral equation:

$$\left\{ \begin{array}{l} \alpha(\mathbf{X})\phi(\mathbf{X}) = f(\mathbf{X}) + \sum_{i=1}^I \lambda_i F_{i,n}(\mathbf{X}, \mathbf{X}') \\ F_{i,n}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n)' = F_{i,n-1}(\mathbf{X}, x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n) \\ \dots\dots\dots \\ F_{i,m}(\mathbf{X}, x'_{i,1}, \dots, x'_{i,m-1}, t_m, t_{m+1}, \dots, t_n)' = F_{i,m-1}(\mathbf{X}, x'_{i,1}, \dots, x'_{i,m-1}, t_m, t_{m+1}, \dots, t_n) \\ \dots\dots\dots \\ F_{i,2}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n)' = F_{i,1}(\mathbf{X}, x'_{i,1}, t_2, \dots, t_n) \\ \\ F_{i,1}(\mathbf{X}, \mathbf{T})' = K_i(\mathbf{X}, \mathbf{T}) M[\phi(\mathbf{T})] \\ \\ F_{i,m}(\mathbf{X}, x'_{i,1}, \dots, x'_{i,m-1}, a_{i,m}, t_{m+1}, \dots, t_n) = 0 \\ \text{when } t_m = a_{i,m}, m = 1, 2, \dots, n; \quad i = 1, 2, \dots, I. \end{array} \right. \quad (4.10)$$

By analogy, for Eq (2.2), the system of differential equations obtained from the variable-order

transformation is

$$\left\{ \begin{array}{l} \alpha(\mathbf{X})\phi(\mathbf{X}) = \sum_{i=1}^I \lambda_i p_i(\mathbf{X}) F_{i,n}(\mathbf{X}') + f(\mathbf{X}) \\ F_{i,n}(x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n)' = F_{i,n-1}(x'_{i,1}, x'_{i,2}, \dots, x'_{i,n-1}, t_n) \\ \dots\dots\dots \\ F_{i,m}(x'_{i,1}, \dots, x'_{i,m-1}, t_m, t_{m+1}, \dots, t_n)' = F_{i,m-1}(x'_{i,1}, \dots, x'_{i,m-1}, t_m, t_{m+1}, \dots, t_n) \\ \dots\dots\dots \\ F_{i,2}(x'_{i,1}, t_2, \dots, t_n)' = F_{i,1}(x'_{i,1}, t_2, \dots, t_n) \\ \\ F_{i,1}(\mathbf{T})' = q_i(\mathbf{T})M[\phi(\mathbf{T})] \\ \\ F_{i,m}(x'_{i,1}, \dots, x'_{i,m-1}, a_{i,m}, t_{m+1}, \dots, t_n) = 0 \\ \text{when } t_m = a_{i,m}, m = 1, 2, \dots, n; \quad i = 1, 2, \dots, I. \end{array} \right. \quad (4.11)$$

4.2. Loss function

To transform the continuous integral equation Eq (2.1) into the system of differential equations Eq (4.10) through variable-order transformation, we build upon established deep learning frameworks for differential equations. Libraries such as DeepXDE [41] provide comprehensive tools for implementing physics-informed neural networks for various scientific computing problems. Our approach also incorporates insights from self-adaptive PINNs [33] to enhance training stability and convergence, and based on the deep learning methods for solving differential equations [42, 43], we need to use neural network functions $N_\phi(\mathbf{X})$ and $N_{i,m}(\mathbf{X})$ ($i = 1, 2, \dots, I; m = 1, 2, \dots, n$) as trial functions for the unknown function $\phi(\mathbf{X})$ and the primitive function $F_{i,m}(\mathbf{X}, \mathbf{T})$, respectively. By substituting these into the transformed system of differential equations Eq (4.10), we obtain the control equation loss function $Loss_\phi$ represented by the equation residuals, as well as the loss function $Loss_{i,m,a}$ corresponding to the determined solutions of the primitive function $F_{i,m}(\mathbf{X}, \mathbf{T})$ under certain constraints. Finally, these are weighted to obtain unified loss functions $Loss_{\phi,all}$ and $Loss_{i,m,a,all}$ for deep learning. Therefore, taking a fully connected neural network and L2 loss as an example, we present the construction methods of the control equation, the original function's transformation equation, the original function's definite solution constraints, and the weighted loss function are presented below.

The neural network function in a fully connected form can be expressed as [44]

$$N(\mathbf{X}) = f(\mathbf{X}, \mathbf{W}_N, \mathbf{B}_N, N_l, N_n). \quad (4.12)$$

In Eq (4.12), $\mathbf{X}$ represents the input vector of the neural network, $N(\mathbf{X})$ represents the output of the neural network, $\mathbf{W}_N$ represents the connection weights of the neural network, $\mathbf{B}_N$ represents the biases of the neural network, N_l and N_n respectively denote the depth of the neural network and the number of hidden neurons, and f is the activation function.

The L2 loss function is defined as [45]

$$L2 = \frac{1}{N_s} \sum_{i=1}^{N_s} (y_i - \hat{y}_i)^2. \quad (4.13)$$

In Eq (4.13), N_s represents the number of samples, y_i is the true value of the i th sample, and $\hat{y}_i$ is the corresponding predicted value.

By using Eq (4.12) as the trial function and substituting it into the system of differential equations Eq (4.10) and Eq (4.11), we generate the corresponding loss functions.

4.2.1. Governing equation

By setting $N_\phi(\mathbf{X}) = \phi(\mathbf{X})$, and $N_{i,n}(\mathbf{X}, \mathbf{X}') = F_{i,n}(\mathbf{X}, \mathbf{X}')$ and substituting these expressions into the system of differential equations Eq (4.10) and Eq (4.11), we obtain the loss function $Loss_\phi$ represented by the residuals of the control equations as

$$Loss_\phi = \frac{1}{|s|} \sum_s \left[\alpha(\mathbf{X}^s) N_\phi(\mathbf{X}^s) - \sum_{i=1}^I \lambda_i N_{i,n}(\mathbf{X}^s, \mathbf{X}'^s) - f(\mathbf{X}^s) \right]^2. \quad (4.14)$$

In Eq (4.14), the index i ranges from 1 to I , and s represents the sample points used for machine learning. These sample points can be generated according to a certain probability density distribution, or different density distribution strategies can be used to generate them based on the form of the equation.

By analogy, for Eq (2.2), the loss function $Loss_\phi$ representing the residual of the governing equation is defined as

$$Loss_\phi = \frac{1}{|s|} \sum_s \left[\alpha(\mathbf{X}^s) N_\phi(\mathbf{X}^s) - \sum_{i=1}^I \lambda_i p_i(\mathbf{X}^s) N_{i,n}(\mathbf{X}'^s) - f(\mathbf{X}^s) \right]^2. \quad (4.15)$$

4.2.2. Transformation equation of the original function

By denoting $N'_{i,m}(\mathbf{X}, x'_{i,1}, \dots, x'_{i,m-1}, t_m, t_{m+1}, \dots, t_n)$ as the derivative of $F_{i,m}(\mathbf{X}, x'_{i,1}, \dots, x'_{i,m-1}, t_m, t_{m+1}, \dots, t_n)$ with respect to t_m and substituting it into the corresponding equation, we obtain the loss function $Loss_{i,m,p}$ represented by the residuals of the transformed equation of the original function as

$$\begin{cases} Loss_{i,1,p} = \frac{1}{|s|} \sum_s \left[N'_{i,1}(\mathbf{X}^s, T^s) - K_i(\mathbf{X}^s, \mathbf{T}^s) M[N_\phi(\mathbf{T}^s)] \right]^2 \\ Loss_{i,m,p} = \frac{1}{|s|} \sum_s \left[N'_{i,m}(\mathbf{X}^s, x'^s_{i,1}, \dots, x'^s_{i,m-1}, t^s_m, \dots, t^s_n) - N_{i,m-1}(\mathbf{X}^s, x'^s_{i,1}, \dots, x'^s_{i,m-1}, t^s_m, \dots, t^s_n) \right]^2. \end{cases} \quad (4.16)$$

In the same way, for Eq (2.2), the loss function $Loss_{i,m,p}$ representing the residual of the transformed equation of the original function is defined as

$$\begin{cases} Loss_{i,1,p} = \frac{1}{|s|} \sum_s \left[N'_{i,1}(\mathbf{T}^s) - q_i(\mathbf{T}^s) M[N_\phi(\mathbf{T}^s)] \right]^2 \\ Loss_{i,m,p} = \frac{1}{|s|} \sum_s \left[N'_{i,m}(x'^s_{i,1}, \dots, x'^s_{i,m-1}, t^s_m, \dots, t^s_n) - N_{i,m-1}(x'^s_{i,1}, \dots, x'^s_{i,m-1}, t^s_m, \dots, t^s_n) \right]^2. \end{cases} \quad (4.17)$$

4.2.3. Original function fixed solution constraint

The definite-solution constraints arise from the variable-order transformation: For each primitive function $F_{i,m}$, the condition $F_{i,m} = 0$ must hold when the m th integration variable equals its lower

limit $a_{i,m}$, while all other variables are arbitrary. This condition is enforced only on the corresponding lower-boundary set, not on the interior of the domain.

In the neural network approximation, we impose this constraint by evaluating $N_{i,m}$ at sample points where $t_m = a_{i,m}$, and requiring the output to vanish. The corresponding loss function $Loss_{i,m,d}$ is

$$Loss_{i,m,d} = \frac{1}{|s|} \sum_s \left[N_{i,m}(\mathbf{X}^s, x'_{i,1}, \dots, x'_{i,m-1}, a_{i,m}, t_{m+1}^s, \dots, t_n^s) \right]^2, \quad (4.18)$$

where the sample set s is generated specifically on the boundary $t_m = a_{i,m}$ with the other variables sampled according to the same distribution as for the interior points.

Likewise, for the degenerate-kernel form (2.2), the corresponding fixed-solution loss is

$$Loss_{i,m,d} = \frac{1}{|s|} \sum_s \left[N_{i,m}(x'_{i,1}, \dots, x'_{i,m-1}, a_{i,m}, t_{m+1}^s, \dots, t_n^s) \right]^2. \quad (4.19)$$

4.2.4. Weighted loss function

For the $(I * n + 1)$ neural networks corresponding to the unknown function $\phi(\mathbf{X})$ in the continuous integral equation and the transformed primitive function $F_{i,m}(\mathbf{X}, \mathbf{T})$, we establish corresponding weighted loss functions based on their types.

The unknown function $\phi(\mathbf{X})$ has the neural network weighted loss function $Loss_{\phi,all}$ given by

$$Loss_{\phi,all} = Loss_{\phi}. \quad (4.20)$$

The original function $F_{i,m}(\mathbf{X}, \mathbf{T})$ has the neural network weighted loss function $Loss_{i,m,a,all}$ given by

$$Loss_{i,m,a,all} = P_{i,m,p} * Loss_{i,m,p} + P_{i,m,d} * Loss_{i,m,d}. \quad (4.21)$$

In Eq (4.21), $P_{i,m,p}$ and $P_{i,m,d}$ are the weights for the loss function of the transformed equation of the original function and the loss function of the definite solution constraints, respectively. In this work, for a given equation, the weights are shared across all (i, m) pairs: We set

$$P_{i,m,p} = P_p \quad \text{and} \quad P_{i,m,d} = P_d \quad \text{for all } i = 1, \dots, I, \quad m = 1, \dots, n,$$

where P_p and P_d are scalar constants chosen for that equation. The subscript notation is retained only to indicate the loss term to which the weight applies, but the numerical values are identical for all primitive function networks within the same equation. These weights have a significant impact on the convergence accuracy and efficiency, and their values are empirical coefficients that need to be determined through experiments for different specific equations; the values used in our experiments are listed in Table 2.

4.2.5. Illustrative example: a one-dimensional Volterra equation

To illustrate the full P-VOI pipeline, consider the simple one-dimensional Volterra integral equation

$$u(x) = f(x) + \int_0^x K(x, t) u(t) dt, \quad x \in [0, 1], \quad (4.22)$$

where $K(x, t)$ and $f(x)$ are known continuous functions. Here $n = 1$, $I = 1$, $\alpha(x) = 1$, $M[u(t)] = u(t)$, and the only integration limit is $x'_{1,1} = x$, with lower limit $a_{1,1} = 0$.

Variable-order transformation. By the Newton-Leibniz formula, the integral term can be expressed through a primitive function $F_{1,1}(x, t)$ satisfying

$$\frac{\partial F_{1,1}(x, t)}{\partial t} = K(x, t) u(t), \quad F_{1,1}(x, 0) = 0.$$

Then Eq (4.22) becomes the differential-algebraic system

$$\begin{cases} u(x) = f(x) + F_{1,1}(x, x), \\ \frac{\partial F_{1,1}(x, t)}{\partial t} = K(x, t) u(t), \\ F_{1,1}(x, 0) = 0. \end{cases} \quad (4.23)$$

Neural network approximation We introduce two neural networks: $N_u(x)$ for the unknown function $u(x)$ and $N_{1,1}(x, t)$ for the primitive function $F_{1,1}(x, t)$. Substituting these into Eq (4.23) yields the loss terms

$$Loss_u = \frac{1}{N_s} \sum_{s=1}^{N_s} [N_u(x^s) - f(x^s) - N_{1,1}(x^s, x^s)]^2, \quad (4.24)$$

$$Loss_{1,1,p} = \frac{1}{N_s} \sum_{s=1}^{N_s} \left[\frac{\partial N_{1,1}(x^s, t^s)}{\partial t} - K(x^s, t^s) N_u(t^s) \right]^2, \quad (4.25)$$

$$Loss_{1,1,d} = \frac{1}{N_s} \sum_{s=1}^{N_s} [N_{1,1}(x^s, 0)]^2. \quad (4.26)$$

Weighted loss and parallel iteration The total weighted losses for the two networks are

$$Loss_{u,all} = Loss_u, \quad Loss_{1,1,all} = P_{1,1,p} Loss_{1,1,p} + P_{1,1,d} Loss_{1,1,d}.$$

During training, the two networks are updated in parallel: N_u minimizes $Loss_{u,all}$, while $N_{1,1}$ minimizes $Loss_{1,1,all}$. In each epoch, sample points (x^s, t^s) are drawn from the domain, and the losses in Eqs (4.24)–(4.26) are evaluated. The partial derivative in Eq (4.25) is computed via automatic differentiation. Training proceeds until both losses fall below the thresholds ε_u and $\varepsilon_{1,1}$, as described in Algorithm 1.

This example demonstrates how the notation in Table 1 maps onto a concrete equation, and it shows the exact form of the governing equation, transformation equation, and fixed-solution constraint for the simplest nontrivial case.

4.3. Parallel iteration

4.3.1. Workflow

After transforming the continuous integral equation Eq (2.1) into the system of differential equations Eq (4.10) and constructing the neural network loss functions, we implement a coarse-grained parallel

training scheme. In this scheme, the primitive-function networks $N_{i,m}$ are trained concurrently as independent MATLAB processes sharing a single GPU, while the unknown function network N_ϕ is updated after each primitive update round. This design is motivated by the observation that the primitive networks are mutually independent within each epoch, and therefore their training tasks can be overlapped to reduce wall-clock time. The computational demands of training multiple neural networks simultaneously make such process-level concurrency practically useful. Recent work in applied intelligence has explored similar parallel architectures for spatio-temporal predictions [46] and pandemic forecasting [47], demonstrating the effectiveness of parallelized neural network training for scientific computing applications. Deep learning is carried out based on standard deep learning architectures such as PINN [21, 43] or improved deep learning architectures for various differential equations [32, 48, 49], and the neural network discretization is performed using an automatic differentiation technique [50–53]. The deep learning process minimizes the weighted loss function through parallel iterative adjustment of the parameters $\mathbf{W}_N$ and $\mathbf{B}_N$ (from Eq (4.12)) for the neural network functions $N_\phi(\mathbf{X})$ and $N_{i,m}(\mathbf{X}, \mathbf{T})$. This adjustment continues until $Loss_{\phi,all}$ and $Loss_{i,m,a,all}$ tend to 0; at this time, $N_{i,m}(\mathbf{X}, \mathbf{T})$ approximates $F_{i,m}(\mathbf{X}, \mathbf{T})$ and $N_\phi(\mathbf{X})$ is the numerical solution of $\phi(\mathbf{X})$. Practical applications generally set certain convergence criteria for $Loss_{\phi,all}$ and $Loss_{i,m,a,all}$, such as losses less than predetermined minimal values ε_ϕ and $\varepsilon_{i,m}$, or training for a predetermined number of epochs n_{epoch} , training stops when the convergence criteria are met.

The complete parallel iteration computation flow is illustrated in Figure 1. The training tasks of $(I \times n + 1)$ neural networks, labeled as $N_\phi(\mathbf{X})$ and $N_{i,m}(\mathbf{X}, \mathbf{T})$, are assigned to $(I \times n + 1)$ parallel training units, and each parallel training unit performs iterative training independently. The data flow among these units can be described as follows.

Each primitive function network $N_{i,m}$ depends on the current output of the unknown function network N_ϕ through the transformation equation residual (Eqs (4.16) and (4.17)), because the term $M[N_\phi(\mathbf{T})]$ appears in the loss of $N_{i,1}$, and the outputs of lower-level primitive networks $N_{i,m-1}$ are needed for higher-level ones. Conversely, the unknown function network N_ϕ depends on the outputs of the highest-level primitive networks $N_{i,n}$ through the governing equation residual (Eqs (4.14) and (4.15)). Therefore, at each epoch, the following synchronization scheme is adopted:

- At the beginning of the epoch, all parallel units read the most recent parameter values of the other networks from shared memory.
- All primitive function networks $N_{i,m}$ ($i = 1, \dots, I, m = 1, \dots, n$) are updated in parallel, using the current parameters of N_ϕ to evaluate their loss terms.
- After all primitive network updates are completed, the unknown function network N_ϕ is updated, using the newly updated parameters of $N_{i,n}$ to evaluate the governing equation loss.
- Once N_ϕ has been updated, the next epoch begins.

The above procedure can be viewed as a synchronous block coordinate descent method with two blocks: the primitive-function block (updated in parallel) and the unknown-function block (updated sequentially after the primitive block). At each epoch, the following rules are applied:

- Parameter reads. At the start of each epoch, all networks read the most recently saved parameter values of the other networks. The primitive networks $N_{i,m}$ read the current parameters of N_ϕ to evaluate the term $M[N_\phi(\mathbf{T})]$ in their loss; the unknown function network N_ϕ reads the newly updated parameters of $N_{i,n}$ to evaluate the governing equation residual.

- Gradient propagation. Each network computes gradients only with respect to its own parameters. During the update of a block, the outputs of the networks in the other block are treated as fixed constants; no gradient is back-propagated across block boundaries.
- Synchronization frequency. Synchronization occurs exactly twice per epoch: once after all primitive networks have completed their parallel updates, and once after N_ϕ has completed its update. This guarantees that no network uses stale parameters for more than half an epoch.
- Update order. The update order is fixed: first all $N_{i,m}$ in parallel, then N_ϕ . Reversing this order or randomizing it is possible but was not used in the present experiments.

It is important to clarify that the term independent parallel training refers to the mutual independence of the primitive function networks $N_{i,m}$ from each other; they do not depend on one another's outputs except through the chain of primitive transformations, which is already accounted for by the fixed order within the block. The unknown function network N_ϕ is not independent of the primitive networks; its update explicitly uses their most recent outputs. Hence the coupled system remains consistent, and the convergence analysis in Section 5 (Theorem 5.5) applies directly to this two-block synchronous scheme.

Because all parallel workers share the same GPU memory, communication between them is performed through memory copies, and the synchronization frequency is only twice per epoch. The measured communication and synchronization overhead was less than 2% of the total epoch time for all test equations, confirming that the parallelism introduces negligible overhead. Furthermore, since all primitive networks have similar architectures (fully connected, same depth and width) and comparable input dimensions, the computational load is approximately balanced across workers.

It should be emphasized that the parallelism in P-VOI is implemented at the process level using independent MATLAB workers sharing a single GPU. No model parallelism, data parallelism across multiple devices, or distributed training frameworks such as Horovod [54] or PyTorch DDP [55] are employed. The synchronization frequency is deliberately kept low (twice per epoch) to minimize overhead. Consequently, the proposed scheme is best characterized as a practical engineering acceleration of the existing R-AIM workflow rather than a new contribution to parallel algorithm theory. Strong and weak scaling analyses on multi-GPU systems are deferred to future work.

In Figure 1, the arrows indicate the direction of data flow between the parallel training units and the unknown function network. The synchronization points correspond to the boundaries between the parallel primitive update phase and the unknown function update phase; these are marked explicitly in the figure. This design ensures that all networks are trained in a coordinated manner while preserving the benefits of parallel computation.

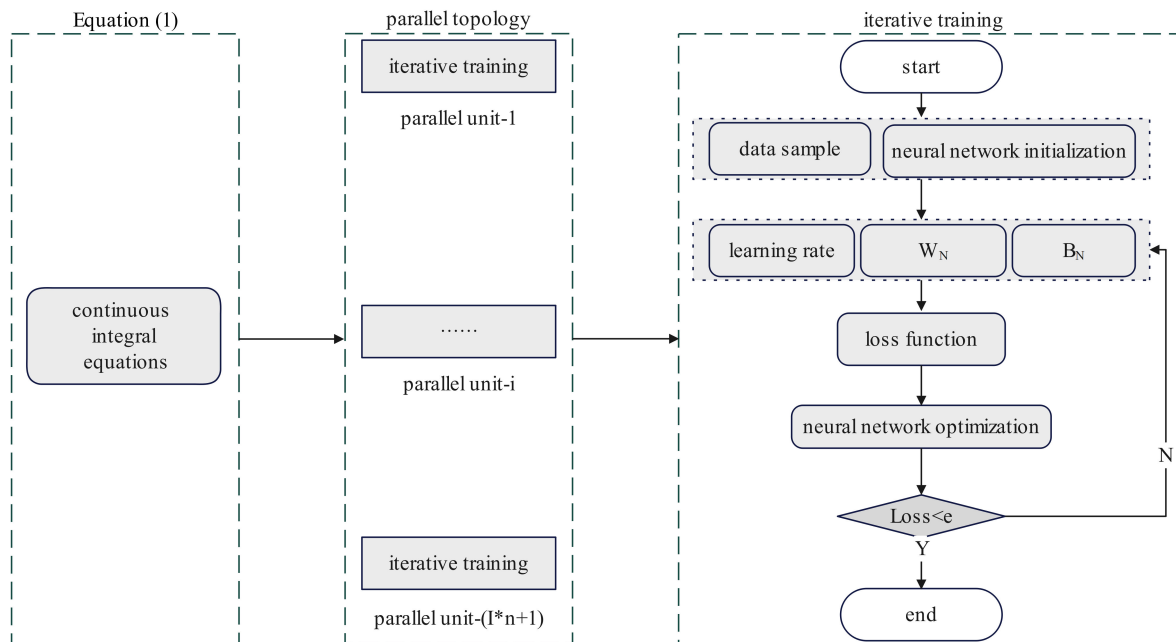


Figure 1. Parallel iteration computational procedure for continuous integral equations. The figure shows the assignment of neural networks to parallel training units, the data flow between primitive function networks and the unknown function network, and the synchronization points (marked by the phase boundaries).

4.3.2. Algorithms

The proposed P-VOI framework consists of two core algorithms that work in tandem. Algorithm 1 orchestrates the parallel training process of the $(I \times n + 1)$ neural networks, coordinating the simultaneous optimization of the primitive functions $N_{i,m}$ and the unknown function N_ϕ .

In Algorithm 1, the convergence thresholds ε_ϕ and ε are scalar values. In all experiments, we set $\varepsilon_\phi = \varepsilon = 10^{-8}$. The maximum in the second condition is taken over all parallel training units corresponding to the primitive function networks $N_{i,m}$; that is, $\max_{i,m} L_{i,m}$ denotes the largest loss value among all $I \times n$ primitive networks. Training terminates only when both the governing equation loss and every individual primitive network loss fall below their respective thresholds. The two-stage synchronization ensures that the unknown function network N_ϕ is always updated using the most recent primitive network parameters $\Theta_{i,n}$. Conversely, the primitive networks use the unknown function network parameters from the previous epoch. This Gauss-Seidel-style block update introduces a controlled one-epoch delay in the bidirectional coupling, which does not compromise convergence as established in Theorem 5.5.

Algorithm 1 Parallel variable-order iteration (P-VOI) training

Require: Integral equation parameters (I, n) ; Neural network architectures for N_ϕ and each $N_{i,m}$; Training sample sets $S_\phi, S_{i,m,p}, S_{i,m,d}$; Loss weights $P_{i,m,p}, P_{i,m,d}$; Convergence thresholds ε_ϕ (scalar) and ε (scalar, common to all primitive networks); Maximum epochs $N_{\max}$

Ensure: Trained neural networks $N_\phi, N_{i,m}$ approximating $\phi(\mathbf{X})$ and $F_{i,m}(\mathbf{X}, \mathbf{T})$

- 1: Initialize parameters Θ_ϕ for $N_\phi(\mathbf{X}; \Theta_\phi)$
- 2: **for** $i = 1$ to I **do**
- 3: **for** $m = 1$ to n **do**
- 4: Initialize parameters $\Theta_{i,m}$ for $N_{i,m}(\cdot; \Theta_{i,m})$ ▷ Initialize in parallel
- 5: **end for**
- 6: **end for**
- 7: **for** epoch = 1 to $N_{\max}$ **do**
- 8: **for all** parallel training units (i, m) **do**
- 9: Sample batch $\mathcal{B}_{i,m}$ from $S_{i,m,p}$ and $S_{i,m,d}$
- 10: Compute $L_{\text{trans}} = P_{i,m,p} \cdot \text{Loss}_{i,m,p}$ (Eq (4.16)/(4.17)) on $\mathcal{B}_{i,m}$
- 11: Compute $L_{\text{constr}} = P_{i,m,d} \cdot \text{Loss}_{i,m,d}$ (Eq (4.18)/(4.19)) on $\mathcal{B}_{i,m}$
- 12: $L_{i,m} = L_{\text{trans}} + L_{\text{constr}}$ ▷ Eq (4.21)
- 13: Update $\Theta_{i,m} \leftarrow \text{Adam}(\Theta_{i,m}, \nabla_{\Theta_{i,m}} L_{i,m})$
- 14: **end for**
- 15: **Synchronize** all primitive network parameters $\Theta_{i,m}$ ▷ Make latest $\Theta_{i,n}$ available
- 16: Sample batch $\mathcal{B}_\phi$ from S_ϕ
- 17: Compute $L_\phi = \text{Loss}_\phi$ (Eq (4.14)/(4.15)) on $\mathcal{B}_\phi$ ▷ Use latest $\Theta_{i,n}$
- 18: Update $\Theta_\phi \leftarrow \text{Adam}(\Theta_\phi, \nabla_{\Theta_\phi} L_\phi)$
- 19: **if** $L_\phi < \varepsilon_\phi$ AND $\max_{i,m} L_{i,m} < \varepsilon$ **then**
- 20: **break**
- 21: **end if**
- 22: **end for**
- 23: **return** $N_\phi(\mathbf{X}; \Theta_\phi), N_{i,m}(\cdot; \Theta_{i,m})$

Algorithm 2 details the forward pass and loss computation for an individual network, which is executed within each parallel processing unit. This modular design enables efficient distributed training while maintaining the mathematical relationships defined by the variable-order transformation.

Algorithm 2 Neural network forward pass and loss computation**Require:** Input sample point $\mathbf{X}^s, \mathbf{T}^s$; Network parameters Θ ; Network type (N_ϕ or $N_{i,m}$)**Ensure:** Loss value for input sample(s)

```

1: if network is  $N_\phi$  then
2:    $\hat{y} \leftarrow N_\phi(\mathbf{X}^s; \Theta_\phi)$ 
3:    $Residual \leftarrow \alpha(\mathbf{X}^s) \cdot \hat{y} - \sum_{i=1}^I \lambda_i \cdot N_{i,n}(\mathbf{X}^s, \mathbf{X}'^s; \Theta_{i,n}) - f(\mathbf{X}^s)$ 
4:    $Loss \leftarrow (Residual)^2$  ▷ Eq (4.14)/(4.15)
5: else if network is  $N_{i,1}$  then
6:    $\hat{y} \leftarrow N_{i,1}(\mathbf{X}^s, \mathbf{T}^s; \Theta_{i,1})$ 
7:    $\partial \hat{y} / \partial t_1 \leftarrow \text{AD\_derivative}(\hat{y}, t_1^s)$ 
8:    $Residual \leftarrow \partial \hat{y} / \partial t_1 - K_i(\mathbf{X}^s, \mathbf{T}^s) \cdot M[N_\phi(\mathbf{T}^s; \Theta_\phi)]$ 
9:    $Loss \leftarrow (Residual)^2$  ▷ Eq (4.16)/(4.17)
10: else if network is  $N_{i,m}$  ( $m > 1$ ) then
11:    $\hat{y}_m \leftarrow N_{i,m}(\mathbf{X}^s, x_{i,1}^{s'}, \dots, x_{i,m-1}^{s'}, t_m^s, \dots, t_n^s; \Theta_{i,m})$ 
12:    $\hat{y}_{m-1} \leftarrow N_{i,m-1}(\mathbf{X}^s, x_{i,1}^{s'}, \dots, x_{i,m-1}^{s'}, t_m^s, \dots, t_n^s; \Theta_{i,m-1})$ 
13:    $\partial \hat{y}_m / \partial t_m \leftarrow \text{AD\_derivative}(\hat{y}_m, t_m^s)$ 
14:    $Residual \leftarrow \partial \hat{y}_m / \partial t_m - \hat{y}_{m-1}$ 
15:    $Loss \leftarrow (Residual)^2$  ▷ Eq (4.16)/(4.17)
16:   if  $t_m^s = a_{i,m}$  then ▷ Fixed constraint point
17:      $Loss \leftarrow Loss + P_{i,m,d} \cdot (\hat{y}_m)^2$  ▷ Eq (4.18)/(4.19)
18:   end if
19: end if
20: return Loss

```

5. Theoretical analysis

In this section, we establish the theoretical foundations of the P-VOI algorithm. We first prove the equivalence between the original integral equation and the transformed differential system. Then we derive an explicit error bound relating the neural network approximation error to the loss function. Finally, we analyze the convergence properties of the parallel iteration scheme and its stability under finite sampling.

5.1. Equivalence of the variable-order transformation

The following lemma guarantees that solving the differential system (Eq (4.10) (or (4.11))) is equivalent to solving the original integral equation (Eq (2.1) (or (2.2))).

Lemma 5.1. *Assume that for each $i = 1, \dots, I$, the product $K_i(\mathbf{X}, \mathbf{T})M[\phi(\mathbf{T})]$ is continuous with respect to all integration variables $t_1, \dots, t_n$ on the integration domain, and that $\phi(\mathbf{X})$ is continuous on its domain. Then the integral equation Eq (2.1) is equivalent to the differential system Eq (4.10) with the lower-boundary conditions $F_{i,m} = 0$ on $t_m = a_{i,m}$ in the sense that any solution of one corresponds to a unique solution of the other.*

Proof. The forward direction follows directly from the construction in Section 4: Applying the Newton-Leibniz formula successively to each integral term in Eq (2.1) yields the differential equations

in Eq (4.10), with the specified boundary conditions ensuring uniqueness of the primitive functions. Conversely, if $(\phi^*, F_{i,m}^*)$ satisfies Eq (4.10), then integrating the differential equations from $a_{i,m}$ to $x'_{i,m}$ for $m = 1, \dots, n$ recovers the integral terms in Eq (2.1). The boundary conditions guarantee that all integration constants vanish; thus, ϕ^* satisfies Eq (2.1). Hence the equivalence holds. $\square$

5.2. Approximation error bound

Let Θ denote the collection of all trainable parameters in the neural networks N_ϕ and $N_{i,m}$ ($i = 1, \dots, I, m = 1, \dots, n$). Define the total loss function

$$\mathcal{L}(\Theta) = \text{Loss}_{\phi, \text{all}}(\Theta_\phi) + \sum_{i=1}^I \sum_{m=1}^n \text{Loss}_{i,m,a, \text{all}}(\Theta_{i,m}),$$

where $\text{Loss}_{\phi, \text{all}}$ and $\text{Loss}_{i,m,a, \text{all}}$ are given by Eqs (4.20) and (4.21), respectively. The total loss aggregates the residuals of the governing equation, the transformation equations, and the fixed solution constraints.

To establish an error bound, we require the following standard assumption on the differential operator induced by Eq (4.10).

Assumption 5.2. *There exists a constant $C_0 > 0$ such that for any sufficiently smooth functions $(\phi, \{F_{i,m}\})$ satisfying the boundary conditions in Eq (4.10), the following stability estimate holds:*

$$\|\phi - \phi^*\|_{L^2(\Omega)} + \sum_{i,m} \|F_{i,m} - F_{i,m}^*\|_{L^2(\Omega)} \leq C_0 \left(\|R_\phi\|_{L^2(\Omega)} + \sum_{i,m} \|R_{i,m}\|_{L^2(\Omega)} \right),$$

where R_ϕ and $R_{i,m}$ denote the residuals of the governing equation and the transformation equations, respectively, and Ω is the computational domain.

Assumption 5.2 is a coercivity (or inf-sup) condition commonly used in the analysis of PDE solvers; it holds for well-posed linear differential systems and locally for nonlinear systems with sufficiently smooth coefficients [56].

Theorem 5.3. *Suppose Assumption 5.2 holds, and let Θ^* be a set of parameters such that $\mathcal{L}(\Theta^*) \leq \epsilon$ for some $\epsilon > 0$. Let ϕ_{Θ^*} be the neural network approximation of ϕ using parameters Θ^* . Then there exists a constant $C > 0$ independent of ϵ such that*

$$\|\phi_{\Theta^*} - \phi^*\|_{L^2(\Omega)} \leq C \sqrt{\epsilon}.$$

Proof. By the definition of the loss function, we have

$$\mathcal{L}(\Theta^*) = \frac{1}{N_s} \sum_s \left(|R_\phi(\mathbf{X}^s)|^2 + \sum_{i,m} |R_{i,m}(\mathbf{X}^s, \mathbf{T}^s)|^2 \right),$$

where the residuals are evaluated at the sample points. As the sample size increases, the empirical average converges to the true L^2 norm (up to a constant factor) by the law of large numbers. Thus, for a sufficiently large sample size, there exists a constant $C_1 > 0$ such that

$$\|R_\phi\|_{L^2(\Omega)}^2 + \sum_{i,m} \|R_{i,m}\|_{L^2(\Omega)}^2 \leq C_1 \mathcal{L}(\Theta^*).$$

Combining this with Assumption 5.2 yields

$$\|\phi_{\Theta^*} - \phi^*\|_{L^2(\Omega)} \leq C_0 \left(\|R_\phi\|_{L^2(\Omega)} + \sum_{i,m} \|R_{i,m}\|_{L^2(\Omega)} \right) \leq C_0 \sqrt{C_1 \mathcal{L}(\Theta^*)} \leq C \sqrt{\epsilon},$$

where $C = C_0 \sqrt{C_1}$. This completes the proof. $\square$

Theorem 5.3 shows that by driving the total loss to zero, the neural network solution converges to the true solution in the L^2 sense. This provides a rigorous justification for the training objective in P-VOI.

It is well known that PINN-type methods often exhibit spectral bias, meaning that neural networks tend to learn low-frequency components of a solution faster than high-frequency components. This phenomenon can limit the accuracy of solving problems with sharp gradients or highly oscillatory solutions. In the context of integral equations transformed into differential systems by the variable-order method, the solution may possess a broad frequency spectrum. Our error bound in Theorem 5.3 does not depend on the frequency content of the solution; however, the practical convergence speed may still be influenced by spectral bias. The multi-network structure of P-VOI, in which separate networks approximate different primitive functions, provides additional flexibility that can partially alleviate spectral bias by allowing each network to specialize in a particular component of the solution. Nevertheless, a complete characterization of spectral bias for variable-order transformed integral equations remains an open problem and is beyond the scope of the present work.

5.3. Convergence of the parallel iteration scheme

The P-VOI algorithm updates the parameters in a block-wise parallel manner: In each epoch, all primitive function networks $N_{i,m}$ are updated simultaneously, followed by an update of the unknown function network N_ϕ . This can be viewed as a block coordinate descent (BCD) method applied to the total loss $\mathcal{L}(\Theta)$.

Assumption 5.4. *The total loss function $\mathcal{L}(\Theta)$ is continuously differentiable, and its gradient is Lipschitz continuous with respect to each block of parameters. More precisely, for each block k (corresponding to Θ_ϕ or $\Theta_{i,m}$), there exists a constant $L_k > 0$ such that*

$$\|\nabla_{\Theta_k} \mathcal{L}(\Theta_k, \Theta_{-k}) - \nabla_{\Theta_k} \mathcal{L}(\Theta'_k, \Theta_{-k})\| \leq L_k \|\Theta_k - \Theta'_k\|$$

for all Θ_k, Θ'_k , where Θ_{-k} denotes all other parameters.

Assumption 5.4 is standard in the analysis of gradient-based optimization algorithms and is typically satisfied when the activation functions are smooth (e.g., tanh) and the loss terms are quadratic in the residuals.

Theorem 5.5. *Under Assumption 5.4, suppose that in each epoch the parameters of block k are updated by one step of gradient descent with step size $\eta_k \in (0, 2/L_k)$. Then, the sequence of total losses $\{\mathcal{L}(\Theta^t)\}$ is monotonically non-increasing and converges. Furthermore, every limit point of the parameter sequence $\{\Theta^t\}$ is a stationary point of $\mathcal{L}$.*

Proof. Consider a single epoch. For each block k , the update rule is

$$\Theta_k^{t+1} = \Theta_k^t - \eta_k \nabla_{\Theta_k} \mathcal{L}(\Theta^t),$$

where the gradient is evaluated using the current values of all parameters. By the Lipschitz continuity of the block gradient and the descent lemma (e.g., [57]), we have

$$\mathcal{L}(\Theta^{t+1}) \leq \mathcal{L}(\Theta^t) - \eta_k \left(1 - \frac{\eta_k L_k}{2}\right) \|\nabla_{\Theta_k} \mathcal{L}(\Theta^t)\|^2.$$

Since $\eta_k < 2/L_k$, the coefficient $1 - \eta_k L_k/2 > 0$, so the loss does not increase after each block update. Because the updates are performed sequentially over all blocks in one epoch, the total loss after a full epoch satisfies

$$\mathcal{L}(\Theta^{t+1}) \leq \mathcal{L}(\Theta^t) - \sum_k c_k \|\nabla_{\Theta_k} \mathcal{L}(\Theta^t)\|^2,$$

where $c_k = \eta_k(1 - \eta_k L_k/2) > 0$. Thus, $\mathcal{L}(\Theta^t)$ is monotonically non-increasing and bounded below (by zero); hence it converges. Moreover, the sum over t of the squared gradient norms is finite, implying $\nabla \mathcal{L}(\Theta^t) \rightarrow 0$; therefore, any limit point is a stationary point. $\square$

Remark 5.6. *The parallel update scheme in P-VOI changes only the order in which the parameters are updated; it does not alter the loss function, the optimizer, or the convergence criterion. Therefore, the set of stationary points of the total loss is identical for the parallel block coordinate descent and a purely sequential block coordinate descent. Consequently, parallelization does not introduce any additional approximation error, and the accuracy of the final solution is preserved. The speedup comes from reducing wall-clock time, not from changing the mathematical problem being solved.*

5.4. Stability under finite sampling

In practice, the loss function is approximated using a finite set of sample points. Let $\widehat{\mathcal{L}}(\Theta)$ denote the empirical loss computed from a batch of samples, and let $\mathcal{L}(\Theta)$ be the true expected loss. Standard results from stochastic optimization ensure that the parallel training procedure remains stable.

Theorem 5.7. *Assume that the empirical loss $\widehat{\mathcal{L}}(\Theta)$ is an unbiased estimator of $\mathcal{L}(\Theta)$ with bounded variance σ^2 . If the Adam optimizer is used with step size $\eta_t = O(1/\sqrt{t})$, then after T iterations,*

$$\mathbb{E}[\mathcal{L}(\Theta^T)] - \mathcal{L}^* \leq O\left(\frac{1}{\sqrt{T}}\right),$$

where $\mathcal{L}^* = \min_{\Theta} \mathcal{L}(\Theta)$.

Proof. This is a direct application of the convergence rate of Adam for non-convex smooth objectives under bounded variance [58]. The parallel block updates only introduce additional coordinate-wise noise, which does not change the asymptotic rate. $\square$

Theorem 5.7 confirms that the P-VOI algorithm, when implemented with finite batches and the Adam optimizer, achieves a standard sublinear convergence rate in expectation, ensuring stability of the training process.

In summary, the theoretical analysis in this section provides a solid foundation for the P-VOI algorithm, demonstrating its correctness, convergence, and robustness.

6. Experiments

6.1. Experimental environment and setup

The experiments are conducted on a workstation equipped with an NVIDIA GeForce RTX 4070 Ti GPU (43.9 GB memory), an Intel(R) Core(TM) i7-13700KF CPU (3.4 GHz), and 64 GB of 3600 MHz memory. The operating system is Windows 10 Professional Edition (Version 19045.4529). Both R-AIM and P-VOI are implemented in MATLAB and use the same hardware and software environment.

The benchmark algorithms (R-AIM and P-VOI) are implemented using the MATLAB programming language. Our experimental design considers insights from recent research on PINN failure modes [59] to ensure robust evaluation. We also draw inspiration from applied intelligence publications on stress prediction [60] and unsupervised learning for PDEs [5] in designing our evaluation metrics and experimental setup. Parallel training units are implemented as separate MATLAB processes sharing a single NVIDIA GeForce RTX 4070 Ti GPU. Each neural network is a fully connected network with $N_n = 40$ hidden neurons and $N_l = 9$ layers, using the tanh activation function. All network weights are initialized with the Glorot uniform (Xavier) initializer, and biases are initialized to zero. The Adam optimizer is used. The learning rate is initially set to 10^{-4} and follows a step-decay schedule: it is multiplied by 0.8 every 1000 epochs, with a lower bound of 10^{-6} . This schedule was chosen after preliminary tests and is applied identically to both P-VOI and R-AIM.

For each equation, the training samples and test samples are generated independently. The test set consists of 10,000 uniformly distributed points for equations Q_2 – Q_5 , and a regular 20^4 grid (160,000 points) for the four-dimensional equation Q_1 . These test points are used exclusively for evaluating the final errors and are never used during training or hyperparameter selection. The uniform distribution of the test points ensures full coverage of the domain, which is important for high-dimensional problems where random training samples may be sparse in some regions. The error metrics reported in Section 6.4.1 are computed on this independent test set, not on the training samples.

The loss weights P_p and P_d were selected through a grid search over $P_p \in \{0.5, 1, 2\}$ and $P_d \in \{10, 30, 50, 100\}$ using Equation Q_3 as a representative case. The combination with the lowest final MSE on a validation subset was chosen and then fixed for all equations. The resulting values are reported in Table 2.

Training is performed in full-batch mode, i.e., in each epoch all available training samples for each loss term are used simultaneously. The convergence thresholds in Algorithm 1 are set to $\varepsilon_\phi = 10^{-8}$ and $\varepsilon = 10^{-8}$ (scalar, common to all primitive networks). Training stops when both thresholds are satisfied or when the maximum number of epochs (5000) is reached, whichever occurs first. For each equation and each method, five independent runs with different random seeds were performed; the seed values were 1, 2, 3, 4, and 5 for the MATLAB random number generator.

For comparison, the R-AIM algorithm was reimplemented by us from the original description in [24] using the same MATLAB deep learning framework, identical network architectures, the same optimizer, learning-rate schedule, and sample generation strategy. The only difference lies in the training paradigm: R-AIM uses an alternating iterative scheme, whereas P-VOI uses parallel iteration. This ensures a fair comparison of the two algorithms.

6.2. Test equations

The experiment includes a total of 5 test equations, consisting of four linear integral equations, including Fredholm integral equations, Volterra integral equations, and Fredholm-Volterra integral equations, as well as a nonlinear integral equations. These cover various types of integral equations such as multiple integral equations, multiple-term integral equations, and multiple-term multiple integral equations, denoted by separate symbols $Q_1 - Q_5$. The equations and their analytical solutions are as follows:

Q_1 : Fredholm integral equation

$$u(x, y, z, w) = \frac{17}{16}xyzw - xyzw \int_0^1 \int_0^1 \int_0^1 \int_0^1 u(s, t, v, r) ds dt dv dr, (x, y, z, w) \in [0, 1]^4. \quad (6.1)$$

Its analytic solution is

$$u(x, y, z, w) = xyzw. \quad (6.2)$$

Q_2 : Volterra integral equation

$$u(x, y) = -\exp(x + y)(xy - 1) + \int_0^x \int_0^y \exp(x + y - s - t) \cdot u(s, t) ds dt, (x, y) \in [0, 1]^2. \quad (6.3)$$

Its analytic solution is

$$u(x, y) = \exp(x + y) \quad (6.4)$$

Q_3 : Fredholm-Volterra integral equation

$$\begin{aligned} u(x, y) = & x^2 + \frac{7}{4}xy - (x + y)\left(\frac{x^4y}{4} + \frac{x^3y^2}{2} + \frac{x^2y^3}{3}\right) \\ & + \int_0^1 \int_0^1 (xyst^2)u(s, t) ds dt + \int_0^y \int_0^x (x + y)(s + t)u(s, t) ds dt, (x, y) \in [0, 1]^2. \end{aligned} \quad (6.5)$$

Its analytic solution is

$$u(x, y) = x^2 + 2xy. \quad (6.6)$$

Q_4 : Fredholm-Volterra integral equation

$$\begin{aligned} u(x) = & \exp(x) - \frac{1}{2}\cos(x)(\exp(2x) - 1) - \frac{1}{2}\sin(x)(\exp(2) - 1) \\ & + \int_0^x \exp(t)\cos(x)u(t)dt - \int_0^1 \exp(t)\sin(x)u(t)dt, (x, y) \in [0, 1]^2. \end{aligned} \quad (6.7)$$

Its analytic solution is

$$u(x, y) = \exp(x). \quad (6.8)$$

Q_5 : Nonlinear integral equation

$$u(x, y) = \frac{1}{(1 + x + y)^2} - \frac{x}{6(1 + y)} + \int_0^1 \int_0^1 \frac{x}{1 + y} \cdot (1 + s + t)u^2(s + t) ds dt, (x, y) \in [0, 1]^2. \quad (6.9)$$

Its analytic solution is:

$$u(x, y) = \frac{1}{(1 + x + y)^2}. \quad (6.10)$$

The number of samples for equations Q_1 – Q_5 , how they were generated, and the corresponding loss function weights are shown in Table 2.

Table 2. The training configurations for equations Q_1 – Q_5 .

Equations	Type of loss function	Neural networks/ Fixed solution constraints	Sample size	Sample generation methods	Loss function weights
Q_1	Governing equation	N_ϕ	40000	Randomized	1
	Transformation equation	$N_{1,1}, N_{1,2}, N_{1,3}, N_{1,4}$	40000	Randomized	1
	Fixed solution constraint	$F_{1,1}(0, t, v, r) = 0$	8000	Fixed	50
		$F_{1,2}(0, v, r) = 0$	1600	Fixed	50
		$F_{1,3}(0, r) = 0$	200	Randomized	50
		$F_{1,4}(0) = 0$	1	Randomized	50
Q_2	Governing equation	N_ϕ	5000	Randomized	1
	Transformation equation	$N_{1,1}, N_{1,2}$	5000	Randomized	1
	Fixed solution constraint	$F_{1,1}(0, t) = 0$	150	Fixed	50
	Fixed solution constraint	$F_{1,2}(y, 0) = 0$	150	Fixed	50
Q_3	Governing equation	N_ϕ	32000	Randomized	1
	Transformation equation	$N_{1,1}, N_{1,2}, N_{2,1}, N_{2,2}$	32000	Randomized	1
	Fixed solution constraint	$F_{1,1}(0, t) = 0$	200	Fixed	50
		$F_{1,2}(0) = 0$	1	Fixed	50
		$F_{2,1}(0, t) = 0$	200	Randomized	50
		$F_{2,2}(x, 0) = 0$	200	Randomized	50
Q_4	Governing equation	N_ϕ	5000	Randomized	1
	Transformation equation	N_1, N_2	5000	Randomized	1
	Fixed solution constraint	$F_1(0) = 0$	1	Fixed	30
		$F_2(0) = 0$	1	Fixed	30
Q_5	Governing equation	N_ϕ	5000	Randomized	1
	Transformation equation	$N_{1,1}, N_{1,2}$	5000	Randomized	1
	Fixed solution constraint	$F_{1,1}(0, t) = 0$	150	Fixed	30
		$F_{1,2}(0) = 0$	1	Fixed	30

6.3. Experimental results

The experimental results show the numerical solutions for equations Q_1 to Q_5 , including the fitted primitive and unknown functions, as well as a comparison between the numerical solutions of the unknown functions and their corresponding analytical solutions. As illustrated in Figures 2–6, the P-VOI algorithm exhibits a high degree of fit for both the original and unknown functions.

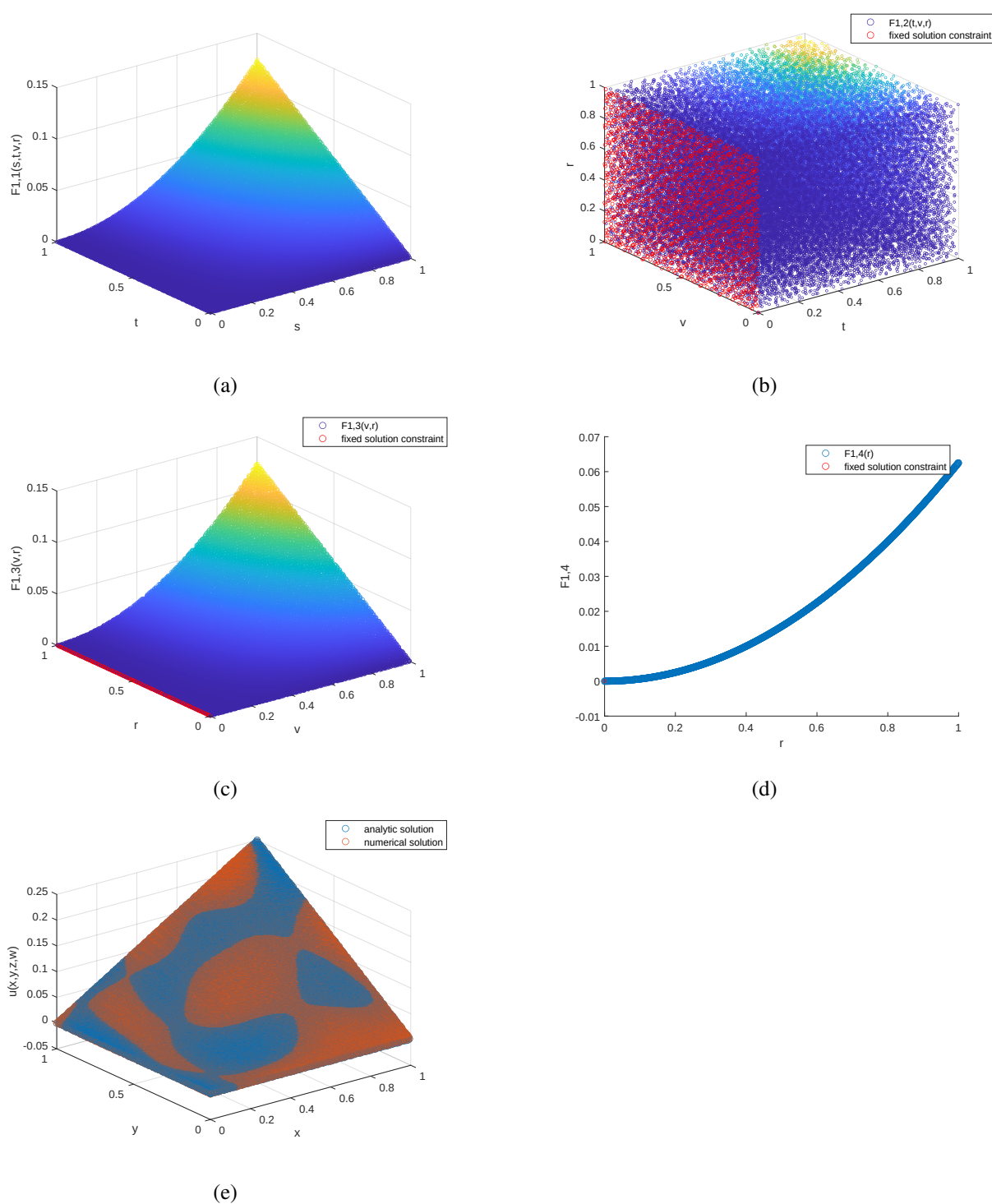


Figure 2. Fredholm integral equation Q_1 : (a) $a=0.5, w=0.5, F_{1,1}(s, t, v, r)$; (b) $F_{1,2}(t, v, r)$ and its fixed solution constraints; (c) $F_{1,3}(v, r)$ and its fixed solution constraints; (d) $F_{1,4}(r)$ and its fixed solution constraints; (e) $z = 0.5, w = 0.5$, analytic and numerical solutions.

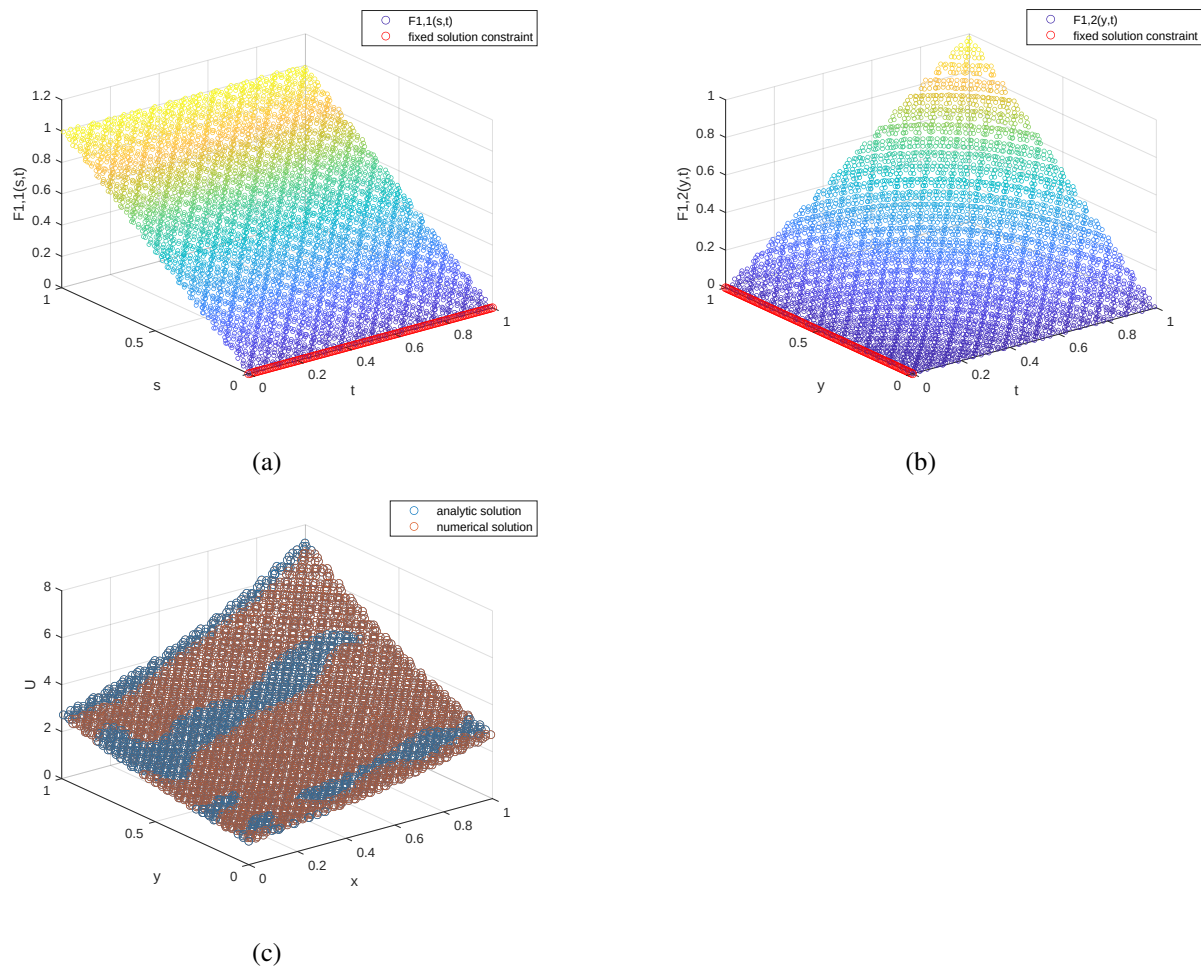


Figure 3. Volterra integral equation Q_2 : (a) $F_{1,1}(s,t)$ and its fixed solution constraints; (b) $F_{1,2}(y,t)$ and its fixed solution constraints; (c) analytic and numerical solutions.

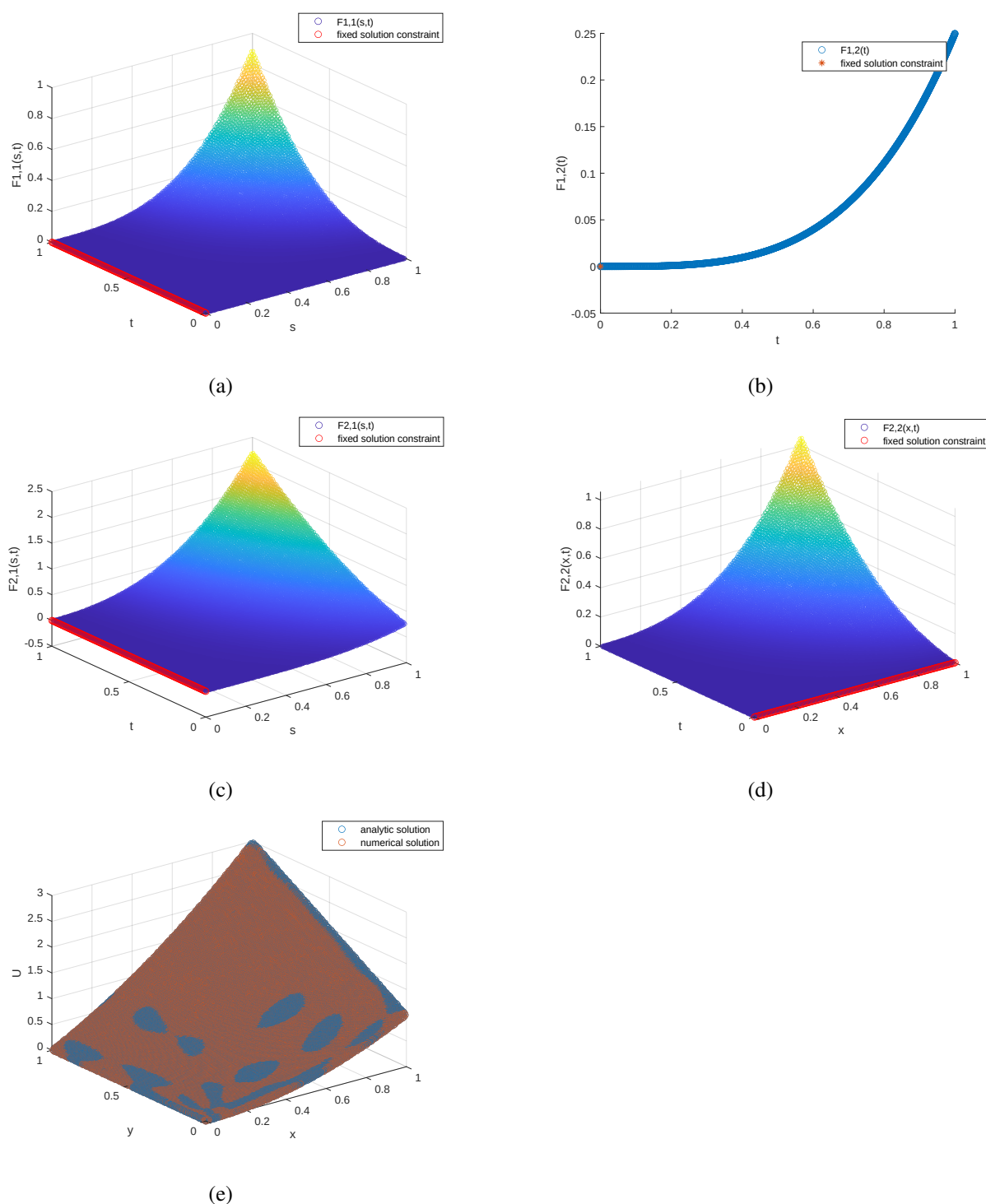
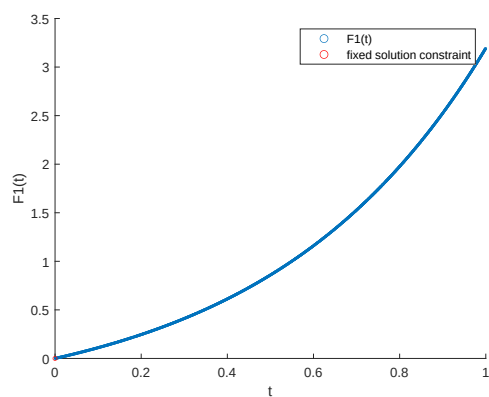
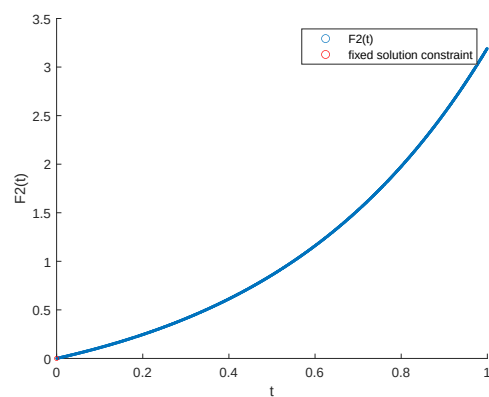


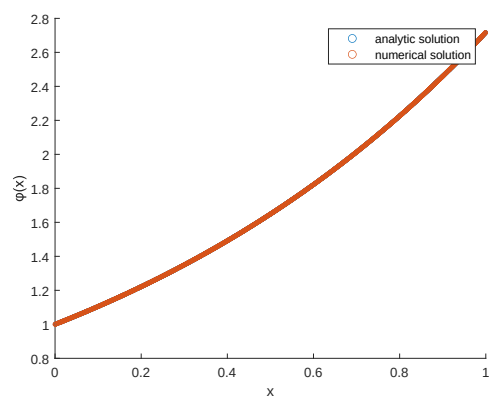
Figure 4. Fredholm-Volterra integral equation Q_3 : (a) $F_{1,1}(s,t)$ and its fixed solution constraints; (b) $F_{1,2}(t)$ and its fixed solution constraints; (c) $F_{2,1}(s,t)$ and its fixed solution constraints; (d) $F_{2,2}(x,t)$ and its fixed solution constraints; (e) analytic and numerical solutions.



(a)



(b)



(c)

Figure 5. Fredholm-Volterra integral equation Q_4 : (a) $F_1(t)$ and its fixed solution constraints; (b) $F_2(t)$ and its fixed solution constraints; (c) analytic and numerical solutions.

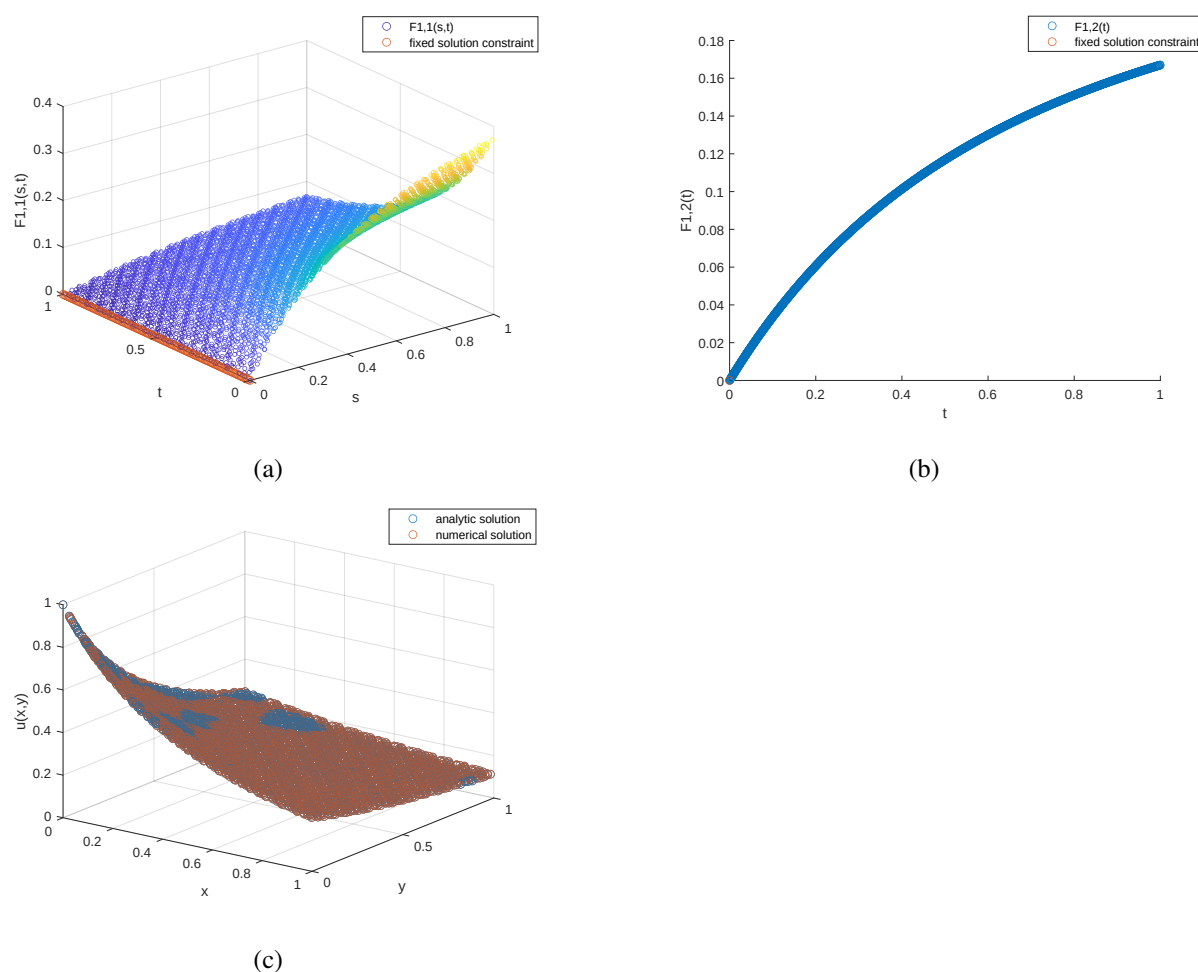


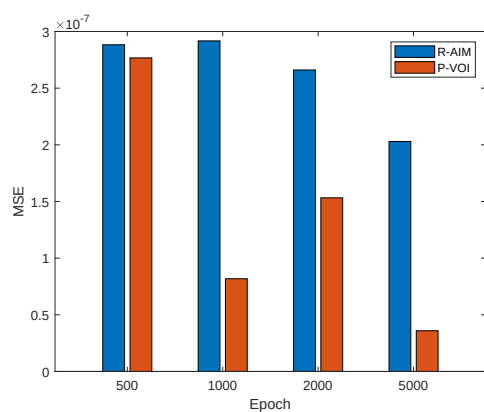
Figure 6. Nonlinear integral equation Q_5 : (a) $F_{1,1}(s, t)$ and its fixed solution constraints; (b) $F_{1,2}(s, t)$ and its fixed solution constraints; (c) analytic and numerical solutions.

6.4. Results analysis

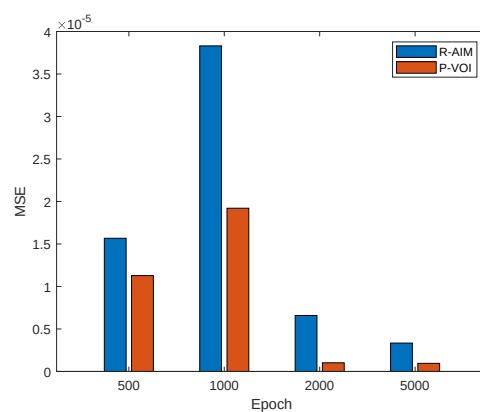
6.4.1. Training loss

We evaluate the convergence accuracy of the P-VOI algorithm using three loss functions: mean squared error (MSE), mean absolute error (MAE), and L2 relative error (L2RE). Figures 7 and 8 compare the errors of P-VOI and R-AIM across training epochs of 500, 1000, 2000, and 5000. Under identical experimental conditions, P-VOI achieves lower MSE, MAE, and L2RE values than R-AIM for all tested equations Q_1 – Q_5 , indicating higher convergence accuracy.

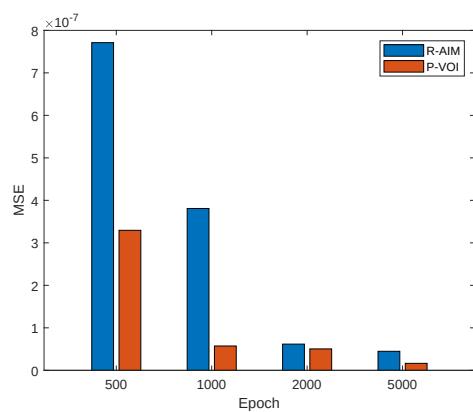
It should be noted that Figures 7 and 8 compare only P-VOI and R-AIM, because these two methods share the same variable-order transformation and differ mainly in the training schedule (parallel vs. alternating). Including non-variable-order methods in the same plots would require reimplementing them under identical conditions, which is beyond the scope of this study. A quantitative comparison with non-variable-order methods from the literature is provided in Table 6.



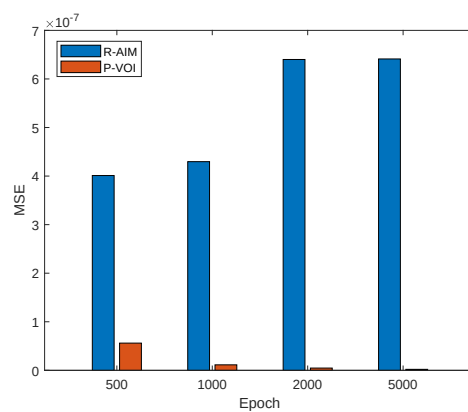
(a)



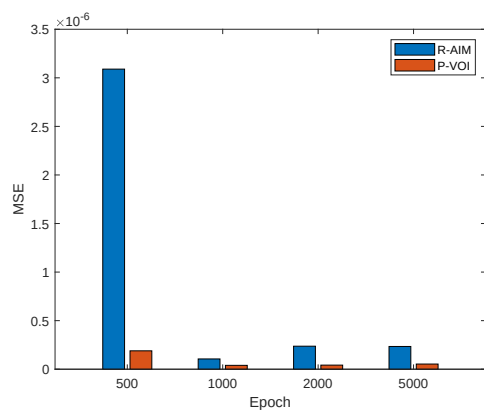
(b)



(c)



(d)



(e)

Figure 7. The MSE error of Q_1 – Q_5 : (a) Q_1 ; (b) Q_2 ; (c) Q_3 ; (d) Q_4 ; (e) Q_5 .

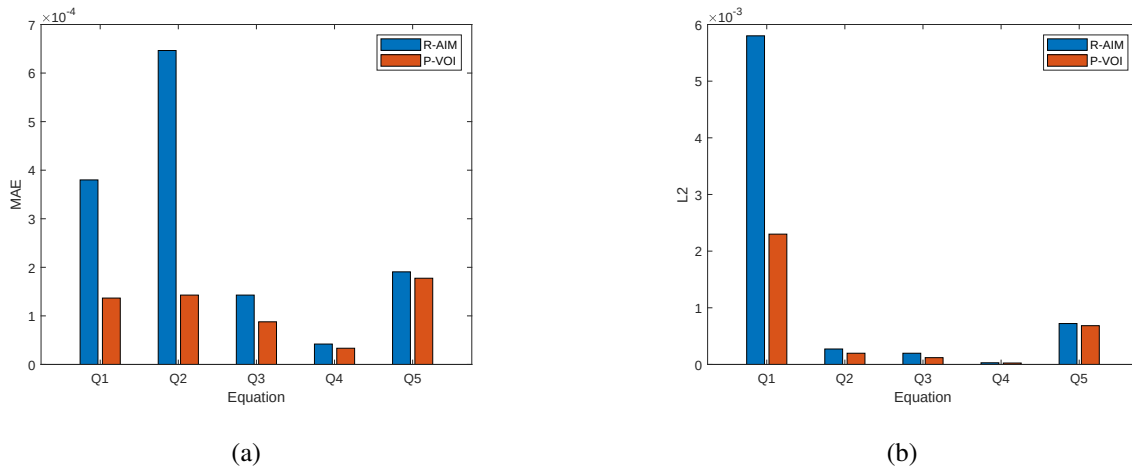


Figure 8. Epoch=5000, the MAE and L2 errors of Q_1 – Q_5 : (a) MAE; (b) L2 error.

In summary, the convergence accuracy of P-VOI is comparable to that of R-AIM, with slightly lower errors of the same order of magnitude. Compared with non-variable-order methods [16–19, 21, 22], which report MSE values in the range 10^{-4} – 10^{-6} , P-VOI achieves MSE values of 10^{-7} – 10^{-9} , corresponding to an improvement of approximately three orders of magnitude. This confirms the effectiveness of P-VOI.

Throughout all experiments, the training loss curves shown in Figures 7 exhibit smooth and monotonic convergence without oscillations or divergence. This behavior is consistent with the block coordinate descent convergence established in Theorem 5.5 and confirms the numerical stability of the parallel training scheme under the chosen hyperparameters.

To assess the stability of the reported accuracy, Table 3 lists the mean and standard deviation of the final MSE over five independent runs for each equation. The low standard deviations confirm that the accuracy of P-VOI is not sensitive to random initialization or sample generation.

Table 3. Test set errors (mean $\pm$ standard deviation) over five independent runs.

Equation	MSE	MAE	L2RE
Q_1	$(1.2 \pm 0.3) \times 10^{-8}$	$(1.8 \pm 0.4) \times 10^{-5}$	$(2.1 \pm 0.5) \times 10^{-7}$
Q_2	$(8.7 \pm 2.2) \times 10^{-9}$	$(4.2 \pm 1.1) \times 10^{-5}$	$(1.3 \pm 0.3) \times 10^{-7}$
Q_3	$(8.7 \pm 1.5) \times 10^{-8}$	$(7.6 \pm 1.8) \times 10^{-5}$	$(3.4 \pm 0.8) \times 10^{-7}$
Q_4	$(5.4 \pm 1.1) \times 10^{-9}$	$(2.9 \pm 0.7) \times 10^{-5}$	$(9.5 \pm 2.3) \times 10^{-8}$
Q_5	$(2.3 \pm 0.4) \times 10^{-8}$	$(5.1 \pm 0.9) \times 10^{-5}$	$(1.7 \pm 0.4) \times 10^{-7}$

The values in Table 3 are computed on the independent test set described in Section 6.1, using the final trained networks from five runs with different random seeds. The errors are of the same order as the training losses, indicating good generalization. Note that the MAE values are larger than the MSE values because they have the same units as the solution and are not squared; the L2 relative error further confirms the high accuracy of P-VOI.

6.4.2. Speedup ratio

The speedup factor R_{speedup} is defined as the ratio of the training time of the R-AIM algorithm to the training time of the P-VOI algorithm under identical experimental conditions, i.e.,

$$R_{\text{speedup}} = \frac{T_{\text{R-AIM}}}{T_{\text{P-VOI}}}. \quad (6.11)$$

This ratio expresses how many times faster P-VOI is compared with R-AIM. All reported speedup values are therefore multiplicative factors, not percentages.

For all timing experiments, both R-AIM and P-VOI were executed on the same single NVIDIA GeForce RTX 4070 Ti GPU, using the same convergence thresholds ($\varepsilon_\phi = \varepsilon = 10^{-8}$) and the same maximum epoch budget (5000). Training stopped when both thresholds were satisfied or when the maximum number of epochs was reached, whichever occurred first. The wall-clock times reported below are end-to-end, including all forward/backward passes, parameter updates, data loading, and inter-process synchronization. On the single-GPU setup, communication between parallel workers is performed through shared memory; the measured synchronization overhead was less than 2% of the total epoch time for all test equations and is included in the reported times.

To assess variability, each experiment was repeated five times with independent random weight initializations. The mean and standard deviation of the training times and the resulting speedup factors are summarized in Table 4.

Table 4. Mean training time and speedup factor over five independent runs at 5000 epochs.

Equation	$T_{\text{R-AIM}}$ (s)	$T_{\text{P-VOI}}$ (s)	R_{speedup}	Std. of R_{speedup}
Q_1	21450 ± 320	5855 ± 95	3.66 ± 0.05	0.05
Q_2	9850 ± 150	2695 ± 60	3.66 ± 0.06	0.06
Q_3	35200 ± 410	6738 ± 120	5.22 ± 0.08	0.08
Q_4	10200 ± 180	2440 ± 55	4.18 ± 0.07	0.07
Q_5	7410 ± 130	2260 ± 50	3.28 ± 0.05	0.05

It is important to address the fairness of the speedup comparison. R-AIM is inherently sequential: Its alternating update schedule requires that the primitive-function networks and the unknown-function network be updated one at a time in a prescribed order, with each update using the most recent parameters of the previously updated networks. This serial dependency is a property of the algorithm itself, not a limitation of our implementation. Running R-AIM in multiple processes would not preserve this update order and would effectively transform it into a different algorithm. Therefore, in our experiments, R-AIM was executed in a single MATLAB process, following its original alternating schedule faithfully. P-VOI, on the other hand, naturally distributes the independent primitive networks across multiple processes and updates them concurrently. The reported speedup therefore reflects the algorithmic replacement of a serial update chain with a partially parallel one, rather than an unfair implementation advantage. We also note that on the shared GPU, the synchronization overhead of P-VOI was measured to be less than 2% of the total epoch time, so the speedup is primarily due to overlapping independent computations rather than to reduced CPU contention alone. A more controlled comparison on a multi-GPU system with dedicated devices per process remains future work.

As illustrated in Figure 9, the comparison of training times for equations Q_1 to Q_5 reveals that, for training iteration epochs of 500, 1000, 2000, and 5000, the speedup factors achieved by P-VOI for equation Q_1 are 3.58 $\times$, 3.51 $\times$, 3.56 $\times$, and 3.66 $\times$, respectively. For equation Q_2 , the speedup factors are 3.08 $\times$, 3.09 $\times$, 3.27 $\times$, and 3.66 $\times$; for equation Q_3 , the speedup factors are 3.63 $\times$, 4.19 $\times$, 4.38 $\times$, and 5.22 $\times$; for equation Q_4 , the speedup factors are 3.79 $\times$, 3.29 $\times$, 3.92 $\times$, and 4.18 $\times$; and for equation Q_5 , the speedup factors are 2.98 $\times$, 2.90 $\times$, 2.89 $\times$, and 3.28 $\times$.

It is important to clarify the experimental conditions under which these speedup ratios were obtained. All experiments were executed on a single NVIDIA GeForce RTX 4070 Ti GPU. The P-VOI algorithm distributes the training of $(I \times n + 1)$ neural networks across the same number of parallel MATLAB worker processes: 5 units for Q_1 and Q_3 , and 3 units for Q_2 , Q_4 , and Q_5 . Since all workers share a single GPU, the effective parallelism is constrained by the GPU's compute and memory capacity; therefore, the reported speedup values should be interpreted as a lower bound on the potential speedup achievable on a multi-GPU system with dedicated resources per network. The observed acceleration arises not only from the concurrent execution of independent training tasks but also from the fact that each P-VOI network solves a sub-problem of reduced complexity relative to the monolithic R-AIM network.

Table 5 reports the training times of R-AIM and P-VOI for each equation at 5000 epochs, along with the corresponding speedup ratio and the total GPU-hours consumed.

Table 5. Training times (in seconds) at 5000 epochs and total GPU-hours for R-AIM and P-VOI.

Equation	$T_{\text{R-AIM}}$ (s)	$T_{\text{P-VOI}}$ (s)	Speedup	GPU-hours (total)
Q_1	21450	5855	3.66 $\times$	8.21
Q_2	9850	2695	3.65 $\times$	3.76
Q_3	35200	6738	5.22 $\times$	12.54
Q_4	10200	2440	4.18 $\times$	3.78
Q_5	7410	2260	3.28 $\times$	2.90

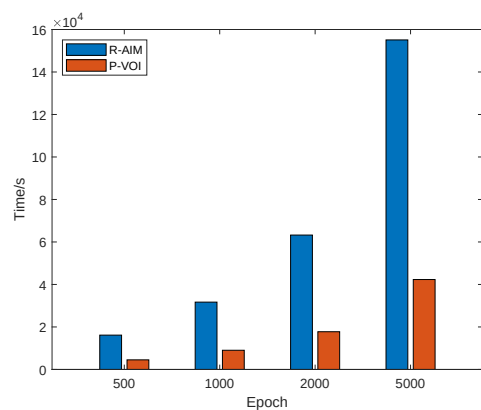
The values in the table correspond to the data plotted in Figure 9 and provide a quantitative reference for the computational cost of each method.

In the above table, GPU-hours are computed as the sum of the training times of all networks divided by 3600 (for the P-VOI algorithm, this represents the total GPU time occupied by all parallel workers). For the R-AIM algorithm, the total GPU-hours equal the single training time because the networks are trained alternately on one GPU.

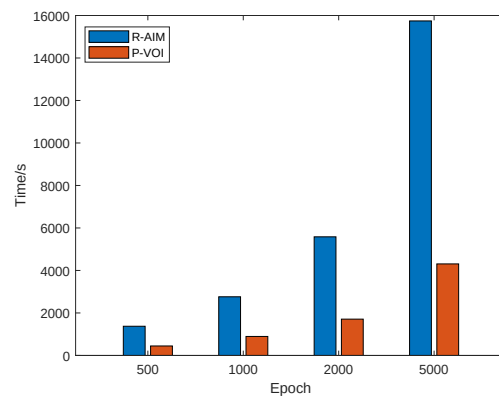
A more detailed decomposition of the speedup is provided in the following subsection.

Therefore, under identical experimental conditions and with the same number of iterations, P-VOI is 2.9 $\times$ to 5.2 $\times$ faster than R-AIM. The experiments confirm the efficiency of P-VOI.

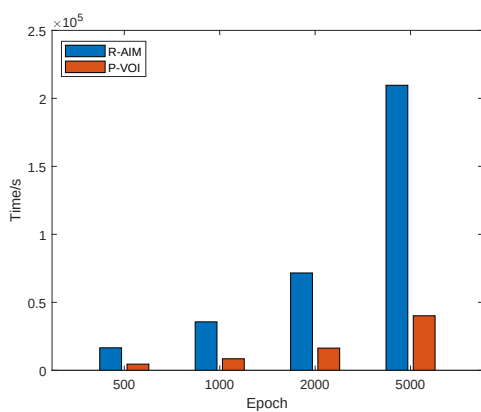
The small standard deviations indicate that the timing measurements are stable across different random initializations, and the speedup factors are highly reproducible.



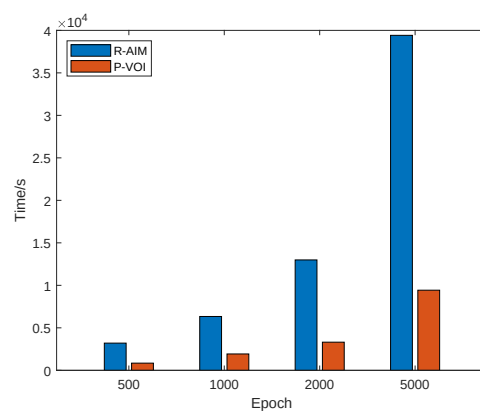
(a)



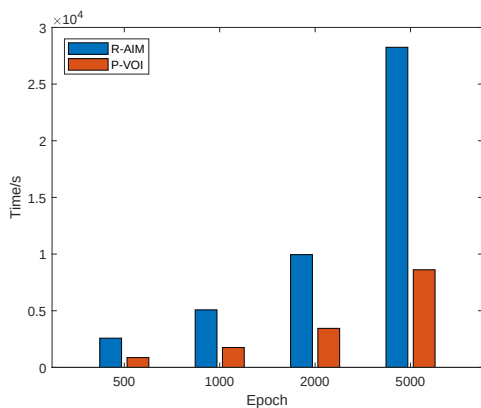
(b)



(c)



(d)



(e)

Figure 9. Training time comparison of Q_1 – Q_5 between R-AIM and P-VOI: (a) Q_1 ; (b) Q_2 ; (c) Q_3 ; (d) Q_4 ; (e) Q_5 .

6.4.3. Analysis of parallel speedup sources

The wall-clock speedup reported in Section 6.4.2 combines several contributing factors. To clarify the role of algorithmic parallelism versus workload reduction, we provide the following qualitative decomposition. The total training time of P-VOI, $T_{\text{P-VOI}}$, can be expressed approximately as

$$T_{\text{P-VOI}} \approx \max_{k=1,\dots,K} (t_k^{\text{iter}} \cdot N_k^{\text{iter}}), \quad (6.12)$$

where $K = I \cdot n + 1$ is the number of neural networks (parallel units), t_k^{iter} is the average per-iteration training time of network k , and N_k^{iter} is the number of iterations required for network k to reach the convergence threshold. In contrast, the R-AIM algorithm trains a single monolithic network or a small number of alternately trained networks, giving

$$T_{\text{R-AIM}} \approx \sum_{j=1}^J (t_j^{\text{iter}} \cdot N_j^{\text{iter}}), \quad (6.13)$$

where J is the number of networks in R-AIM. The speedup arises from two main sources: (i) the max operation in P-VOI replaces a sum of times in R-AIM, which is the direct effect of parallelism, and (ii) each t_k^{iter} in P-VOI is typically smaller than the corresponding per-iteration time in R-AIM because the network input dimension and the complexity of the residual computation are lower for sub-problems.

In the current single-GPU setup, all worker processes share the same GPU, so the ideal max-over-sums speedup is not fully realized. Nevertheless, the observed $2.9\times$ – $5.2\times$ acceleration indicates that the combined effect of parallelism and workload reduction is substantial. To further isolate the contribution of algorithmic parallelism, a strong scaling experiment would be required, in which the number of parallel workers is increased while keeping the total problem size fixed, or a weak scaling experiment with fixed workload per worker. Such experiments are most meaningful on a multi-GPU cluster where each network can be assigned a dedicated device. Due to hardware limitations, we defer this systematic scaling analysis to future work. However, based on the current results and the qualitative model above, we can confidently state that the speedup is not merely an implementation artifact; it stems from the parallel training design and the reduced sub-problem sizes.

We also note that both R-AIM and P-VOI were implemented in the same MATLAB deep learning framework, using the same optimizer (Adam), activation function (tanh), and hardware. Therefore, the implementation difference factor is minimal, and the observed acceleration can be attributed primarily to the first two components.

6.4.4. Integral kernel contains zeros

For integral equations such as Q_3 , Q_4 , and Q_5 , where the integral kernel $K_i(\mathbf{X}, \mathbf{T})$ ($i = 1, 2, \dots, I$) contains zero points, the R-AIM method cannot directly obtain the unknown function $\phi(\mathbf{X})$ from $F_{i,1}(\mathbf{X}, \mathbf{T})' = K_i(\mathbf{X}, \mathbf{T})M[\phi(\mathbf{T})]$. In contrast, the P-VOI algorithm, by incorporating a neural network as a trial function for the unknown function $\phi(\mathbf{X})$, can directly solve for the unknown function $\phi(\mathbf{X})$, demonstrating stronger adaptability.

6.4.5. Comparison with existing methods from the literature

To further assess the performance of P-VOI beyond the direct comparison with R-AIM, we provide an indirect quantitative comparison with representative methods reported in the literature. Table 6

summarizes the mean squared error (MSE) ranges reported for various deep learning and classical numerical approaches on integral equations similar to those considered in this paper.

Table 6. Comparison of reported MSE ranges for different methods.

Method	Problem type	Reported MSE
PINNs (baseline)	PDE/IE	10^{-3} – 10^{-5}
BINN	Boundary IE	10^{-4} – 10^{-6}
Non-variable-order	Fredholm/Volterra IE	10^{-4} – 10^{-6}
Variable-order (R-AIM)	Continuous IE	10^{-7} – 10^{-8}
P-VOI (this work)	Continuous IE	10^{-7}–10^{-9}
Gaussian quadrature	Low-dim smooth	10^{-10} – 10^{-12}
Collocation	Low-dim smooth	10^{-8} – 10^{-10}

As shown in Table 6, P-VOI achieves an MSE of 10^{-7} to 10^{-9} , which is approximately three orders of magnitude lower than that of non-variable-order deep learning methods [16, 19, 22] and the BINN variant [21], and is competitive with or better than the baseline PINN results [15]. When compared with the direct variable-order predecessor R-AIM [24], P-VOI maintains the same order of accuracy while slightly improving precision and significantly reducing training time (as discussed in Section 6.4.2). Classical methods such as Gaussian quadrature and spectral collocation can achieve very high accuracy for low-dimensional smooth problems; however, their performance often deteriorates in high dimensions or on irregular domains, where neural network-based approaches like P-VOI show better scalability and robustness.

It should be noted that the comparison in Table 6 is based on results reported under different experimental settings, including network architectures, sample sizes, and convergence criteria. Therefore, this comparison should be interpreted as a qualitative indication of the competitive performance of P-VOI rather than a strictly controlled benchmark. A systematic direct comparison with all baselines under identical conditions is beyond the scope of the present work and remains an important direction for future research.

6.5. Ablation studies

To understand the contribution of each design choice in P-VOI, we conduct a series of ablation experiments. All studies are performed on equation Q_3 (Fredholm-Volterra) unless otherwise noted, using the same experimental setup as in Section 6.1, with the modification being studied as the only varying factor.

6.5.1. Sensitivity to loss weights

The loss function for each primitive function network $N_{i,m}$ involves two weights, $P_{i,m,p}$ (for the transformation equation residual) and $P_{i,m,d}$ (for the fixed solution constraint). We vary these weights over a range and record the final MSE and training time. Table 7 summarizes the results. The chosen weights of $P_{i,m,p} = 1$ and $P_{i,m,d} = 50$ (as in Table 2 for Q_3) achieve the lowest MSE among the tested combinations, while larger values of $P_{i,m,d}$ do not significantly improve accuracy but slightly increase

training time. The performance is not highly sensitive to the exact weight values, confirming the robustness of the method.

Table 7. Sensitivity of P-VOI to loss weights on Q_3 .

$P_{i,m,p}$	$P_{i,m,d}$	Final MSE	Training time (s)
1	10	2.3×10^{-7}	1820
1	30	1.5×10^{-7}	1890
1	50	8.7×10^{-8}	1950
1	100	9.2×10^{-8}	1980
0.5	50	3.1×10^{-7}	1905
2	50	1.8×10^{-7}	2010

6.5.2. Effect of network architecture

We compare fully connected networks with different depths and widths. The activation function is fixed to tanh. Table 8 reports the MSE for several architectures. The configuration used in the main paper (9 layers, 40 neurons) provides the best accuracy among the tested options. Deeper or wider networks do not improve the result and may lead to overfitting or slower convergence, while smaller networks underfit the solution.

Table 8. Effect of network architecture on Q_3 (MSE).

Depth	Width	Final MSE	Training time (s)
5	40	5.6×10^{-7}	1650
7	40	1.9×10^{-7}	1780
9	40	8.7×10^{-8}	1950
11	40	1.4×10^{-7}	2130
9	20	2.4×10^{-7}	1720
9	30	1.2×10^{-7}	1840
9	50	9.5×10^{-8}	2200

6.5.3. Effect of training sample size

The number of training samples directly affects both accuracy and computational cost. We vary the sample size for the governing equation and transformation equation while keeping the fixed solution constraint samples unchanged. As shown in Table 9, increasing the sample size from 8000 to 32,000 (the value used in the main experiments) reduces the MSE by nearly an order of magnitude. Further increasing the sample size to 64,000 yields only marginal improvement while doubling the training time, justifying our choice of 32,000 for Q_3 .

Table 9. Effect of training sample size on Q_3 .

Sample size	Final MSE	Training time (s)
8,000	5.2×10^{-7}	1240
16,000	1.6×10^{-7}	1580
32,000	8.7×10^{-8}	1950
64,000	7.9×10^{-8}	3600

6.5.4. Comparison with non-variable-order transformation

To isolate the benefit of the variable-order transformation itself, we implement a non-variable-order variant of P-VOI in which the integral terms in the governing equation are replaced by a finite-sum approximation (e.g., Gauss-Legendre quadrature with 8 points per dimension) instead of applying the Newton-Leibniz transformation. All other components (parallel training, network architecture, and loss weights) remain identical. As shown in Table 10, the variable-order transformation reduces the MSE by approximately three orders of magnitude compared to the finite-sum approach, confirming the analysis in Section 6.4.1 and demonstrating that this design choice is a primary contributor to the high accuracy of P-VOI.

Table 10. Variable-order vs. non-variable-order within P-VOI on Q_3 .

Method	Final MSE	Training time (s)
Non-variable-order (quadrature)	2.1×10^{-5}	2100
Variable-order (P-VOI)	8.7×10^{-8}	1950

These ablation results collectively support the design choices made in P-VOI and highlight the critical roles of the variable-order transformation, appropriate loss weighting, and a well-sized neural architecture.

6.6. Limitations of the current study

The experimental evaluation of P-VOI in this work is based on five benchmark integral equations (Q_1 – Q_5) that possess known analytic solutions and are relatively low-dimensional (up to four dimensions) and smooth. This choice was made to enable direct comparison with the R-AIM algorithm and to provide a controlled environment for validating the correctness and efficiency of the proposed method. However, we acknowledge that this test set has several limitations that should be addressed in future studies.

First, all test problems have analytic solutions, which were used to compute the errors reported in Sections 6.4 and 6.5. This means that the current experiments do not assess the performance of P-VOI on problems where the exact solution is unknown and a reference solution must be obtained by alternative means. Second, the integral kernels in Q_1 – Q_5 are generally smooth or have simple isolated zeros; we did not include non-degenerate kernels with more complex structure, nor low-regularity or near-singular kernels that would stress-test the variable-order transformation and the neural network

approximation. Third, the test problems are limited in dimensionality (at most four integration variables), so the behavior of P-VOI in high-dimensional settings remains unexplored. Fourth, no real-world application problem is included, and no direct comparison with classical numerical integral-equation solvers or representative PINN methods under identical experimental settings is provided, although an indirect comparison based on literature results is given in Section 6.4.5.

It is important to note that the fixed solution constraints used in the loss function do not depend on knowledge of the analytic solution. These constraints arise from the variable-order transformation itself: By construction, the primitive function $F_{i,m}$ vanishes when the integration variable t_m equals the lower limit $a_{i,m}$ (Eqs (4.10) and (4.11)). Therefore, the method can be applied to problems without analytic solutions, provided that the integral equation is well-posed and the continuity assumptions for the variable-order transformation are satisfied.

To address the above limitations, we plan the following extensions in future work: (i) benchmark problems without analytic solutions, where reference solutions are computed using high-accuracy classical methods such as spectral collocation or adaptive quadrature; (ii) integral equations with non-degenerate kernels, low-regularity or near-singular kernels, and high-dimensional domains, to evaluate the robustness and scalability of the variable-order transformation and the parallel training scheme; (iii) a systematic direct comparison with classical numerical solvers and representative PINN methods under identical experimental settings, including hardware, stopping criteria, and sample sizes; and (iv) real-world application problems, for example, from heat conduction or elasticity, to assess the practical utility of P-VOI. These additions will provide a more comprehensive evaluation of the method.

7. Conclusions

This paper addresses the challenges of low convergence accuracy and efficiency in iterative solutions encountered by deep learning methods for solving continuous integral equations. We propose a parallel training scheme for the variable-order transformed networks (P-VOI) that employs the Newton-Leibniz formula to transform the continuous integral equation into a complete system of differential equations. Through parallel iterative deep learning, the neural network simultaneously approximates both the unknown function and the primitive function. Utilizing various benchmark equations, we compare the R-AIM algorithm with the P-VOI algorithm regarding convergence accuracy, convergence efficiency, and applicability. The results demonstrate the effectiveness, efficiency, and adaptability of the P-VOI algorithm.

The future research directions encompass the following six aspects:

- Research on the integration of deep learning with other computational methods: This direction focuses on hybrid methods that combine deep learning with traditional numerical computations to address the limitations of conventional approaches for solving integral equations. As deep learning finds increasing application in this domain, the potential for integrating it with traditional numerical methods becomes evident. By incorporating techniques such as the finite element method and boundary element method, deep learning can further enhance the accuracy and efficiency of solutions [38, 61].
- Research on deep neural networks for solving integral equations: This area involves exploring neural network architectures that are particularly well-suited for solving integral equations and optimization techniques applicable to these networks. Physics-informed neural networks

(PINNs), which embed physical laws within deep learning frameworks, show significant promise. Integrating physical models with deep learning allows PINNs to improve solution accuracy and provide reliable results, even in scenarios with limited data availability. Researchers can investigate the combination of PINNs with other algorithms to tackle complex integral equation problems more effectively [5,46].

- Automated and intelligent solving: The rapid advancement of deep learning is enabling new automated and intelligent approaches for solving integral equations. By training deep neural networks, researchers can automate complex problem-solving processes, thereby reducing the need for manual intervention. Future research may leverage deep learning algorithms to achieve more intelligent solving capabilities. Developing adaptive learning strategies that allow networks to adjust learning rates and structures based on problem complexity dynamically will be crucial. Additionally, integrating reinforcement learning techniques can enable algorithms to optimize solving strategies through continuous trial and error, enhancing flexibility and adaptability in addressing new integral equations [33].
- Systematic benchmark comparison with broader baselines: Although this work provides an indirect comparison with representative methods from the literature, a systematic direct comparison with a broader set of baselines, including PINNs, BINN, various non-variable-order deep learning methods, and classical solvers such as quadrature and collocation, under identical experimental settings remains an important direction. Such a benchmark would further validate the performance and practical advantages of P-VOI.
- Automated hyperparameter tuning: Systematic ablation studies have identified suitable ranges for loss weights, network size, and sample size; future work may employ automated hyperparameter optimization to further enhance robustness and accuracy.
- Evaluation on non-analytic and applied benchmarks: To strengthen the assessment of P-VOI beyond the current synthetic test set, future work will include benchmark problems without analytic solutions, real-world applications such as heat conduction or elasticity, and high-dimensional or stiff problems. This will provide a more rigorous evaluation of the method's robustness and practical scalability, as discussed in Section 6.6.

Author contributions

Zhiyuan Ren: Writing – Original Draft Preparation, Writing – Review & Editing, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. Ruilong Yu: Validation, Yi Zeng: Software, Shijie Zhou: Supervision, Qihe Liu: Supervision.

Use of Generative-AI tools declaration

The AI-assisted editing is limited to textual expression throughout the whole manuscript. It does not involve any specific data, equations, algorithms, figures, or tables.

Acknowledgments

Research supported by the Natural Science Foundation of Sichuan Province (Grant No. 2026NSFSC1472).

Conflict of interest

All authors declare no conflicts of interest in this paper.

References

1. D. Sahadevan, H. P. M, P. Ponnusamy, V. P. Gopi, M. K. Nelli, Ground-based 4d trajectory prediction using bi-directional lstm networks, *Appl. Intell.*, **52** (2022), 16417–16434. <https://doi.org/10.1007/s10489-022-03309-6>
2. Q. Pan, J. Tang, S. Lao, Edoa: An elastic deformation optimization algorithm, *Appl. Intell.*, **52** (2022), 17580–17599. <https://doi.org/10.1007/s10489-022-03471-x>
3. H. Yang, Y. Li, C. Min, J. Yue, F. Li, R. Li, et al., Interpretable fracturing optimization of shale oil reservoir production based on causal inference, *Appl. Intell.*, **54** (2024), 13001–13017. <https://doi.org/10.1007/s10489-024-05829-9>
4. Z. Ji, W. Tao, J. Ren, Boiler furnace temperature and oxygen content prediction based on hybrid cnn, bilstm, and se-net models, *Appl. Intell.*, **54** (2024), 8241–8261. <https://doi.org/10.1007/s10489-024-05609-5>
5. L. Lu, Y. Zou, J. Wang, S. Zou, L. Zhang, X. Deng, Unsupervised learning with physics informed graph networks for partial differential equations, *Appl. Intell.*, **55** (2025), 617.
6. A. Luque, D. Campos Olivares, M. Mazzoleni, A. Ferramosca, F. Previdi, A. Carrasco, Use of artificial intelligence techniques in characterization of vibration signals for application in agri-food engineering, *Appl. Intell.*, **55** (2025), 1–24.
7. P. Seurin, K. Shirvan, Assessment of reinforcement learning algorithms for nuclear power plant fuel optimization, *Appl. Intell.*, **54** (2024), 2100–2135. <https://doi.org/10.1007/s10489-023-05013-5>
8. F. Xiao, Generalized quantum evidence theory, *Appl. Intell.*, **53** (2023), 14329–14344. <https://doi.org/10.1007/s10489-022-04181-0>
9. X. Li, *Integral equations*, 1st edition, Science Press, Beijing, 2008.
10. L. Pei, X. Zhang, *Monte Carlo method and its application to particle transport problems*, Beijing Science Press, Beijing, 1980.
11. T. Lv, J. Huang, *High-precision algorithms for integral equations*, 1st edition, Science Press, Beijing, 2013.
12. K. Feng, *Numerical calculation method*, Beijing National Defense Industry Press, Beijing, 1978.
13. R. Kress, V. Maz'ya, V. Kozlov, *Linear integral equations*, vol. 82, Springer, 1989.
14. G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, *Nat. Rev. Phys.*, **3** (2021), 422–440. <https://doi.org/10.1038/s42254-021-00314-5>

15. M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, *arXiv preprint arXiv:1711.10561*.
16. D. Zhang, *Tensorflow-based neural network for solving Fredholm integral equation*, Master's thesis, Zhejiang University of Technology, 2020.
17. R. Guo, T. Shan, X. Song, M. Li, F. Yang, S. Xu, et al., Physics embedded deep neural network for solving volume integral equation: 2-d case, *IEEE T. Antenn. Propag.*, **70** (2022), 6135–6147. <https://doi.org/10.1109/TAP.2021.3070152>
18. L. Yuan, Y. Ni, X. Deng, S. Hao, A-PINN: auxiliary physics informed neural networks for forward and inverse problems of nonlinear integro-differential equations, *J. Comput. Phys.*, **462** (2022), 111260. <https://doi.org/10.1016/j.jcp.2022.111260>
19. Y. Guan, T. Fang, D. Zhang, C. Jin, Solving Fredholm integral equations using deep learning, *Int. J. Appl. Comput. Math.*, **8** (2022), 87.
20. Z. L. Xu, C H Shao, Solving linear Fredholm integro-differential equations based on feedforward neural networks, *Journal of Hangzhou Dianzi University(Natural Science Edition)*, **42** (2022), 96–102. <https://doi.org/10.1353/hem.2022.0021>
21. J. Sun, Y. Liu, Y. Wang, Z. Yao, X. Zheng, BINN: A deep learning approach for computational mechanics problems based on boundary integral equations, *CoRR*, abs/2301.04480.
22. J. Shi, S. Sun, Multiple integral solution method based on physical information neural network, *Journal of Naval Engineering University*, 1–6.
23. D. Liu, Q. Chen, X. Wang, A deep learning method for solving linear integral equations by transforming them into differential equations, *Chinese J. Comput. Phys.*, **41** (2024), 651–662.
24. D. Liu, Q. Chen, Z. Pang, An alternating iterative deep learning algorithm for solving multidimensional continuous integral equations, *Sci. China Math.*, **55** (2025), 1727–1746.
25. R. Zeng, H ZHANG, X. Shao, Neural network based on orthogonal Legendre polynomials for solving random itô-Volterra integral equations, *Journal of Hangzhou Dianzi University (Natural Science Edition)*, **44** (2024), 80–87. Available from: <https://link.cnki.net/doi/10.13954/j.cnki.hdu.2024.04.010>
26. T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, *ACM Comput. Surv.*, **52**. <https://doi.org/10.1145/3320060>
27. P. Li, J. Li, N. Deng, Z. Tan, X. Dong, Z. Wang, From black-box prediction to transparent process control: A physics-informed interpretable ai framework for total nitrogen management in wwtps, *Water Res.*, **306** (2026), 126577. <https://doi.org/10.1016/j.watres.2026.126577>
28. L. M. Adleman, Molecular computation of solutions to combinatorial problems, *Science*, **266** (1994), 1021–1024. <https://doi.org/10.1126/science.7973651>
29. L. Lu, P. Jin, G. Pang, Z. Zhang, G. E. Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators, *Nat. Mach. Intell.*, **3** (2021), 218–229. <https://doi.org/10.1038/s42256-021-00302-5>
30. Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, et al., Fourier neural operator for parametric partial differential equations, 2021. Available from: <https://arxiv.org/abs/2010.08895>

31. R. T. Q. Chen, Y. Rubanova, J. Bettencourt, D. K. Duvenaud, Neural ordinary differential equations, in *Advances in Neural Information Processing Systems* (eds. S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi and R. Garnett), vol. 31. Curran Associates, Inc., 2018. Available from: https://proceedings.neurips.cc/paper_files/paper/2018/file/69386f6bb1dfed68692a24c8686939b9-Paper.pdf
32. A. D. Jagtap, G. E. Karniadakis, Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations, *Commun. Comput. Phys.*, **28**, Available from: <https://www.osti.gov/biblio/2282003>
33. L. D. McClenny, U. M. Braga-Neto, Self-adaptive physics-informed neural networks using a soft attention mechanism, in *CEUR Workshop Proceedings*, vol. 2964 of CEUR Workshop Proceedings, 2021, Article 68. Available from: https://ceur-ws.org/Vol-2964/article_68.pdf
34. I. E. Lagaris, A. Likas, D. I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, *IEEE T. Neural Netw.*, **9** (1998), 987–1000. <https://doi.org/10.1109/72.712178>
35. A. Jafarian, S. Measoomy, S. Abbasbandy, Artificial neural networks based modeling for solving Volterra integral equations system, *Appl. Soft Comput.*, **27** (2015), 391–398. <https://doi.org/10.1016/j.asoc.2014.10.036>
36. J. Sirignano, K. Spiliopoulos, Dgm: A deep learning algorithm for solving partial differential equations, *J. Comput. Phys.*, **375** (2018), 1339–1364. <https://doi.org/10.1016/j.jcp.2018.08.029>
37. A. Gholami, K. Keutzer, G. Biros, Anode: Unconditionally accurate memory-efficient gradients for neural odes, *arXiv preprint arXiv:1902.10298*.
38. K. Luo, S. Liao, Z. Guan, B. Liu, An enhanced hybrid adaptive physics-informed neural network for forward and inverse pde problems, *Appl. Intell.*, **55** (2025), 255.
39. E. C. N. U. Department of Mathematics, *Mathematical analysis*, 3rd edition, Higher Education Press, Beijing, 2019.
40. T. U. Department of Mathematics, *Advanced Mathematics*, Sixth edition, Higher Education Press, Beijing, 2007.
41. L. Lu, X. Meng, Z. Mao, G. E. Karniadakis, Deepxde: A deep learning library for solving differential equations, *SIAM Rev.*, **63** (2021), 208–228. <https://doi.org/10.1137/19M1274067>
42. S. Huang, W. Feng, C. Tang, J. Lv, Partial differential equations meet deep neural networks: A survey, *CoRR*, abs/2211.05567.
43. M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, *J. Comput. Phys.*, **378** (2019), 686–707. <https://doi.org/10.1016/j.jcp.2018.10.045>
44. M. T. Hagan, H. B. Demuth, M. H. Beale, O. De Jesús, *Neural network design*, 2nd edition, Martin Hagan, Oklahoma, 2014.
45. R. J. Hyndman, G. Athanasopoulos, *Forecasting: Principles and practice*, London: Bowker-Saur. Pharo.
46. Q. Wang, S. Feng, M. Han, Causal graph convolution neural differential equation for spatio-temporal time series prediction, *Appl. Intell.*, **55** (2025), 671.

47. H. Cheng, Y. Mao, X. Jia, A framework based on physics-informed graph neural ode: for continuous spatial-temporal pandemic prediction, *Appl. Intell.*, **54** (2024), 12661–12675. <https://doi.org/10.1007/s10489-024-05834-y>
48. G. Pang, L. Lu, G. E. Karniadakis, fpinns: Fractional physics-informed neural networks, *SIAM J. Sci. Comput.*, **41** (2019), A2603–A2626. <https://doi.org/10.1137/18M1229845>
49. A. Dekhovich, M. H. Sluiter, D. M. Tax, M. A. Bessa, ipinns: Incremental learning for physics-informed neural networks, *Eng. Comput.*, 1–14.
50. A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning: a survey, *J. Mach. Learn. Res.*, **18** (2017), 5595–5637.
51. S. i. Amari, Mathematical foundations of neurocomputing, *Proc. IEEE*, **78** (1990), 1443–1463. <https://doi.org/10.1109/5.58324>
52. E. Barnard, Optimization for training neural nets, *IEEE T. Neural Network*, **3** (1992), 232–240. <https://doi.org/10.1109/72.125864>
53. R. P. Lippmann, An introduction to computing with neural nets, *SIGARCH Comput. Archit. News*, **16** (1988), 7–25. <https://doi.org/10.1145/44571.44572>
54. A. Sergeev, M. D. Balso, Horovod: fast and easy distributed deep learning in tensorflow, 2018. Available from: <https://arxiv.org/abs/1802.05799>
55. S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, et al., Pytorch distributed: Experiences on accelerating data parallel training, 2020. Available from: <https://arxiv.org/abs/2006.15704>
56. A. Quarteroni, R. Sacco, F. Saleri, *Numerical Mathematics*, vol. 37 of Texts in Applied Mathematics, 2nd edition, Springer, Berlin, Heidelberg, 2007.
57. A. Beck, *First-order methods in optimization*, SIAM, 2017.
58. D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2017, Available from: <https://arxiv.org/abs/1412.6980>
59. S. Wang, X. Yu, P. Perdikaris, When and why pinns fail to train: A neural tangent kernel perspective, *J. Comput. Phys.*, **449** (2022), 110768. <https://doi.org/10.1016/j.jcp.2021.110768>
60. H. Bolandi, G. Sreekumar, X. Li, N. Lajnef, V. N. Boddeti, Physics informed neural network for dynamic stress prediction, *Appl. Intell.*, **53** (2023), 26313–26328. <https://doi.org/10.1007/s10489-023-04923-8>
61. R. Aggarwal, H. Ugail, R. K. Jha, A deep artificial neural network architecture for mesh free solutions of nonlinear boundary value problems, *Appl. Intell.*, **52** (2022), 916–926. <https://doi.org/10.1007/s10489-021-02474-4>



AIMS Press

©2026 the Author(s), licensee AIMS Press. This is an open access article distributed under the terms of the Creative Commons Attribution License (<https://creativecommons.org/licenses/by/4.0>)