Parallel-in-Time methods with machine learning based coarse propagators

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation (Monografie)

von
Abdul Qadir Ibrahim

aus
Accra, Ghana

2026

1. Gutachter:	Professor Daniel Ruprecht
2. Gutachter:	Professor Colin Cotter
Prüfungsvorsitzender:	Prof. Dr. Matthias Schulte

Tag der mündlichen Prüfung: 21. Oktober 2025

DOI: https://doi.org/10.15480/882.16764
Author: https://orcid.org/0000-0003-3452-8500
Handle: https://hdl.handle.net/11420/61716

Abstract

This thesis investigates the integration of machine learning into parallel-in-time algorithms to accelerate the numerical solution of partial differential equations (PDEs) in computational finance. We focus on the Parareal algorithm and propose using neural network models as coarse propagators to improve its convergence and parallel efficiency. In particular, a Physics-Informed Neural Network (PINN) is implemented as the coarse solver for the single-asset Black–Scholes option pricing PDE, and a Physics-Informed Neural Operator (PINO) based on the Fourier Neural Operator is developed for the two-asset Black–Scholes PDE. These learned coarse models are trained to approximate the PDE solution while enforcing physical constraints, enabling fast evaluations on GPUs. We demonstrate that the PINN coarse propagator significantly accelerates Parareal's convergence, achieving speedups beyond the limits of spatial-only parallelization.
Extending to a two-dimensional Black–Scholes equation, the PINO coarse model allows space-time parallelism that outperforms traditional fine solvers, even when spatial parallelization alone saturates. Through theoretical analysis on a representative hyperbolic PDE (the advection equation), we identify key properties for effective coarse integrators – notably, preserving wave phase speed and stability. Neural coarse models trained with these insights yield markedly improved convergence on challenging problems where standard Parareal would diverge.
Overall, the results indicate that machine-learning-based coarse propagators can substantially enhance parallel-in-time methods for PDEs, offering a promising avenue to tackle computationally intensive problems in finance and beyond.

Summary/Zusammenfassung

This thesis explores the intersection of machine learning and numerical analysis by integrating deep learning models into parallel-in-time algorithms for solving partial differential equations (PDEs), with a focus on applications in computational finance. Traditional solvers for PDEs often rely heavily on spatial parallelization, which tends to saturate due to communication bottlenecks and limited scalability. Time-parallel methods such as the Parareal algorithm offer an alternative by enabling concurrent computation across temporal domains. However, their performance depends critically on the quality and efficiency of the coarse propagator used in the iterative scheme.

To address this bottleneck, the thesis proposes using data-driven models—specifically, Physics-Informed Neural Networks (PINNs) and Physics-Informed Neural Operators (PINOs)—as learned coarse propagators. These models are trained to approximate the solution of the PDE while respecting physical constraints such as the governing equations and boundary conditions. A PINN is used to model the coarse solver for the single-asset Black–Scholes equation, while a Fourier-based PINO is employed for the two-asset version. Both models are trained offline and evaluated efficiently on GPUs, enabling substantial speedups in the Parareal framework.

Theoretical analysis is carried out using the linear advection equation to understand the behavior of neural coarse models in terms of stability and phase speed preservation—two critical factors for the convergence of Parareal. The results confirm that when these properties are well-captured, the Parareal iterations converge faster and more robustly, even for challenging PDEs where standard configurations fail.

Extensive numerical experiments show that this hybrid approach achieves speedups that go beyond spatial-only parallelism, making the method viable for high-dimensional, time-dependent problems in finance and potentially other domains. The findings suggest a promising pathway for leveraging deep learning to enhance the scalability and performance of scientific computing frameworks.

Acknowledgements

First, to Daniel Ruprecht, my advisor and the reason I am here. When I first contacted you as a prospective student, I did not expect such generosity. You helped me develop a PhD proposal from the ground up, guided me through every obstacle, and answered my many questions with remarkable patience. I am deeply grateful for your support. To Anna Camilla Wunderlich, thank you for standing by me through every late-night doubt, every breakthrough, and every high and low of the past years. Your support never wavered. To my family, friends, my office mate Sophie Externbrink, and my co-supervisors Sebastian Götschel and Colin Cotter, thank you for your constant encouragement. This thesis is also dedicated to my father, Ibrahim Abdul-Kadir, who passed away before I could complete it. I hope I have made you proud. And to the mathematicians who came before us—thank you for leaving enough unsolved problems to keep the rest of us busy.

Contents

Chapter 1

Introduction

Models based on differential equations are ubiquitous in both science and engineering. High-resolution requirements, often due to the multiscale nature of many problems, typically require these models to be run on high-performance computers (HPC) to cope with memory demand and computational cost. Spatial parallelization is already a widely used and effective approach for parallelizing numerical algorithms for partial differential equations; however, on its own, it will not deliver sufficient concurrency for extreme-scale parallel architectures [1, 2, 3]. Recent advances have explored parallel-across-the-method and direct time-parallel integrators that circumvent the traditional coarse-propagator bottleneck. For instance, serial-free spectral deferred correction (SDC) frameworks have emerged, enabling concurrency across collocation nodes and promising efficient multithreading without coarse model serial state [4]. Another family leveraging direct paradiagonalization schemes (ParaDiag) offers efficient time-parallel solutions by reformulating time stepping as block-diagonal systems solvable via parallel linear algebra [5]. These approaches distribute computations across time layers without iterative coarse refinements, potentially unlocking higher concurrency and complementing space–time strategies such as Parareal, PFASST, and MGRIT. Integrating these with neural-based coarse models could further enhance scalability on heterogeneous HPC systems.

Parallel-in-time integration algorithms can increase the degree of parallelism in numerical models. Combined space-time parallelization can improve speedup over spatial parallelization alone on hundreds of thousands of processing cores [6]. Parallel-in-time methods like Parareal [7], Parallel Full Approximation Scheme in Space and Time (PFASST) [8] or Multigrid Reduction in Time (MGRIT) [9] relies on serial coarse-level integrators to propagate information forward. These coarse propagators constitute an unavoidable serial bottleneck that limits achievable speedup [10]. Therefore, the coarse-level integrators must be as fast as possible. These methods are iterative, and the speedup decreases as the number of iterations increases. A coarse propagator that is too inaccurate, even when computationally cheap, will not provide a good speedup because the number of required iterations will be too large. Hence, a good coarse propagator must be at least somewhat accurate, but it also needs to run as fast as possible. This trade-off suggests that using neural networks as coarse propagators could be promising; once

trained, they are very fast to evaluate while still providing reasonable accuracy [11]. Furthermore, neural networks are well-suited for running on GPUs, whereas mesh-based discretizations are more difficult to run efficiently because of their lower computational intensity. Therefore, algorithms featuring a combination of mesh-based components and neural network components are well suited to run on heterogeneous systems combining CPUs and GPUs or other accelerators. This study made three novel contributions.

1. We provide the first study using a Physics Informed Neural Network (PINN) as a coarse propagator in Parareal. We show that a PINN as a coarse propagator can accelerate Parareal convergence and improve speedup, and illustrate that moving the PINN coarse propagator to GPUs improves speedup further.

2. We then extend the approach to parallelize in time across multiple nodes and work in combination with spatial parallelization. Space-time parallelization, combining spatial parallelization via domain decomposition with parallel-in-time integration, has been shown to be able to extend parallel scaling beyond what parallelization in space alone can provide [6]. We investigate the effectiveness of a Physics-Informed Neural Operator (PINO) as a coarse-level model for the parallel-in-time integration method Parareal to solve the two-asset Black-Scholes equation, which is a PDE used in computational finance. The key novelty of this study is the demonstration that Parareal with an ML-based coarse model not only provides speedup over serial time stepping, but can also extend scaling beyond the saturation point of space-only parallelization. We only investigated the performance of a single node with one GPU.

3. We then conduct a theoretical analysis to determine the essential properties of an effective coarse propagator, with the goal of improving the convergence of the Parareal algorithm. To this end, we examine the simple advection equation, a problem for which standard Parareal methods are known to suffer from divergence. By investigating how neural networks can be leveraged as coarse propagators, we aim to understand the mechanisms through which they enhance convergence and mitigate instability in the Parareal algorithm.

While we demonstrate our approach for the Black-Scholes equation, a model from computational finance, the idea is transferable to other types of partial differential equations, where Parareal was shown to be effective.

1.1 Overview of Differential Equation Models in Science and Engineering

Differential equations form the cornerstone of mathematical modeling in science and engineering. They describe a wide variety of physical, biological, and economic systems by capturing the relationships between the rates of change of quantities and the

underlying variables of the system. From modeling the flow of fluids in aerodynamics to simulating the dynamics of financial markets, differential equations provide a universal language for understanding and predicting complex phenomena. Partial Differential Equations (PDEs), in particular, play a central role in many disciplines.
They arise naturally when systems depend on multiple variables, such as time and space, and they enable the formulation of models that describe wave propagation, heat transfer, quantum mechanics, electromagnetism, and fluid dynamics, among others. For example: The Navier-Stokes equations [12] describe the motion of viscous fluid substances and are fundamental in fields such as meteorology, oceanography, and aerospace engineering. Similarly, the Schrödinger equation [13] plays a crucial role in quantum mechanics, providing a mathematical framework for understanding how the quantum state of a physical system evolves over time. In computational finance, the Black-Scholes equation [14] is widely used to model the dynamics of financial derivatives and option pricing.
In many real-world applications, these equations often exhibit multiscale phenomena, where processes occur at vastly different spatial and temporal scales [15, 16]. For instance, in climate modeling, interactions between large-scale atmospheric patterns and small-scale turbulence must be captured simultaneously [17]. Similarly, in structural engineering, models must account for both the large-scale geometry of the structure and the small-scale stress distributions within materials [18]. The numerical solution of differential equations has become a vital tool in these domains, particularly when analytical solutions are either impractical or impossible to derive [19]. Numerical methods, such as finite difference, finite element, and spectral methods, approximate solutions by discretizing the problem into a finite set of equations that can be solved using computers [20, 21]. These methods enable high-fidelity simulations, providing critical insights into complex systems and informing decision-making processes [22]. However, the increasing complexity of modern applications poses significant challenges [23].
High-resolution simulations of PDEs, required to capture fine-scale dynamics, demand substantial computational resources [24]. For example, resolving turbulence in fluid flow simulations or modeling large-scale systems like Earth's climate requires grids with millions or even billions of degrees of freedom, leading to memory and computational costs that scale exponentially with resolution [25]. To cope with these challenges, high-performance computing (HPC) systems have become indispensable [26]. By leveraging parallel computing architectures, researchers can distribute the computational workload across thousands of processors, reducing simulation time and enabling larger and more detailed simulations [27].
Parallelization strategies, particularly in the spatial domain, have achieved remarkable success in scaling numerical PDE solvers [28].Despite these advancements, a fundamental limitation remains: traditional spatial parallelization approaches eventually saturate as the problem size increases [29]. To fully exploit the power of modern HPC systems, new strategies that introduce additional layers of parallelism, such as parallel-in-time methods, are essential [30]. These methods aim to complement spatial parallelization by distributing computations across time steps, enabling the simultaneous advancement of the solution at multiple temporal points [31].

In this context, the integration of machine learning, particularly neural networks, is emerging as a promising avenue [32]. Machine learning models can serve as approximators for complex dynamics, providing efficient alternatives to traditional solvers [33]. By combining the interpretability of physics-based models with the flexibility and computational efficiency of neural networks, researchers are opening new frontiers in the numerical solution of differential equations [34]. This work situates itself at the intersection of these developments, exploring novel methodologies that leverage machine learning to enhance the parallel solution of differential equations [35]. Through a focus on the Black-Scholes equation as a case study, we demonstrate the potential of neural networks to accelerate computations, improve scalability, and address the challenges of modern scientific computing [36].

1.2 Parallelization in Numerical Algorithms for PDEs

While the mathematical formulation of partial differential equations (PDEs) enables us to model a wide range of physical and engineered systems, their numerical solution especially at high resolution—demands substantial computational effort. Parallelization has become an indispensable strategy for accelerating simulations, but the way it is implemented has evolved significantly to address the needs of modern high-performance computing (HPC) systems [37, 23]. The most widely adopted approach is *spatial parallelization*, in which the computational domain is decomposed into subdomains assigned to different processors or computing nodes. Each processor advances the solution within its subdomain, coordinating boundary information with neighboring regions. This method is particularly effective for elliptic and parabolic PDEs, where locality in the spatial stencil allows for efficient communication patterns [28, 27]. Domain decomposition methods, multigrid solvers, and distributed-memory finite element implementations have all seen strong scaling on supercomputers [38].

However, spatial parallelism alone is insufficient for exploiting the full concurrency available on modern extreme-scale architectures. As the number of cores grows, the number of spatial elements per core can become too small to sustain high computational throughput, leading to parallel inefficiency [29]. Moreover, in problems where the temporal domain is large or when long time horizons are required—such as in climate modeling or financial forecasting—sequential time stepping can become a critical bottleneck. To address this, researchers have developed *parallel-in-time* methods such as Parareal [7], PFASST [39], and MGRIT [9]. These methods introduce concurrency along the temporal axis, enabling multiple time steps to be processed simultaneously. Unlike spatial parallelism, which benefits from localized communication, time-parallel methods must carefully manage global dependencies and stability constraints. As a result, many of these schemes rely on hierarchical or iterative correction strategies, where coarse approximations guide the fine-scale solution. Emerging directions include hybrid approaches that combine spatial, temporal, and algorithmic parallelism—such as *parallel-across-the-method* formulations [4]—as well as the integration of machine learning models to approximate components of traditional solvers [40]. These strategies are

especially promising for heterogeneous HPC architectures that combine CPUs, GPUs, and other accelerators, offering new pathways to scalable, efficient PDE solvers.

1.2.1 Overview of Spatial Parallelization

Spatial parallelization is the most widely used strategy in numerical algorithms for solving PDEs [28]. This approach divides the spatial domain of the problem into smaller subdomains, distributing the computational workload across multiple processors [27]. Each processor independently computes the solution for its assigned subdomain, while boundary conditions are exchanged between neighboring subdomains to maintain consistency [41]. Common methods for spatial parallelization include domain decomposition methods and grid-based parallelization. Domain decomposition methods divide the computational domain into smaller overlapping or non-overlapping subdomains [42]. Techniques such as the Schur complement method or Schwarz methods facilitate efficient communication between subdomains while maintaining numerical accuracy [43]. Grid-based parallelization, on the other hand, involves dividing the computational grid into uniform blocks in structured grid methods, which are then distributed among processors [44]. In the case of unstructured grids, dynamic load-balancing techniques are often employed to ensure an even distribution of computational effort [45].
Spatial parallelization has proven highly effective for numerous applications, particularly on high-performance computing (HPC) systems [29]. Large-scale fluid dynamics simulations using the Navier-Stokes equations or structural analysis in engineering, for example, can efficiently utilize tens of thousands of processor cores [46]. However, as the number of cores increases, communication overhead and load imbalance become significant challenges, ultimately leading to diminishing returns in performance [47].
Despite its success, spatial parallelization alone is not sufficient for extreme-scale computing architectures [24]. Two primary limitations hinder its scalability. First, communication bottlenecks arise as the number of processors increases, leading to a substantial increase in data exchanged between subdomains and resulting in significant communication overhead [43].
Second, there is a saturation of spatial degrees of freedom, meaning that for a fixed spatial resolution, there exists a threshold beyond which the number of spatial processors cannot be further increased effectively [29]. Beyond this limit, the computational workload per processor becomes too small, leading to diminishing returns or even performance degradation due to communication overheads [47]. To address these limitations, researchers have investigated alternative forms of concurrency, notably in the temporal dimension [30]. A frequently raised counterargument is that *"only weak scaling matters"*, wherein the problem size is increased proportionally with the number of processors to maintain a constant workload per core. However, this line of reasoning does not fully mitigate the fundamental bottlenecks. Even under ideal weak scaling in space, accurate simulation of complex physical systems typically requires temporal refinement. As time step sizes are reduced to satisfy stability or accuracy constraints—particularly for stiff or multiscale problems—the sequential nature of time integration imposes a strict limitation

on scalability. This observation highlights the necessity of exploiting parallelism in the temporal domain, an aspect that remains underappreciated in many high-performance computing contexts.

1.2.2 Parallelization in Time: An Overview

Parallel-in-time methods provide a complementary approach to spatial parallelization by distributing computations across the temporal domain [48]. Unlike traditional time-stepping methods, which solve for each time step sequentially, parallel-in-time methods allow multiple time steps to be computed concurrently [7]. This approach introduces a new layer of parallelism, significantly increasing the concurrency of numerical algorithms [49]. Several parallel-in-time methods have been developed over the years, each offering different approaches to improving computational efficiency. The Parareal method, proposed by Lions et al. [7], utilizes a combination of coarse and fine time integrators to iteratively converge to the solution. In this method, a coarse propagator provides a fast but approximate solution, while a fine propagator runs in parallel to refine the accuracy [50]. Another widely used technique is the Parallel Full Approximation Scheme in Space and Time (PFASST), developed by Emmett and Minion [8]. PFASST extends the Parareal method to multilevel formulations, allowing for even greater scalability [49]. Additionally, the Multigrid Reduction in Time (MGRIT) method, introduced by Falgout et al. [9], applies multigrid techniques to efficiently solve partial differential equations in the temporal domain [51]. These methods rely on iterative schemes to propagate information forward in time, ensuring consistency and convergence across parallel time slices. Parallel-in-time methods have shown promise in extending scalability for applications where spatial parallelization alone is insufficient [52].

1.2.3 Challenges in Parallel-in-Time Integration

Despite their potential, parallel-in-time methods encounter several challenges that impact their efficiency and scalability [31]. One major issue is the presence of serial bottlenecks, as most methods rely on coarse-level integrators to propagate information forward in time [7]. These integrators, typically executed sequentially, create a limitation on the overall speedup that can be achieved [50]. Another challenge involves the trade-off between accuracy and computational cost. A coarse propagator must strike a balance between maintaining sufficient accuracy and minimizing computational expense. If the coarse model is too simplistic, it may require numerous iterations to converge, whereas a highly accurate model may diminish the benefits of parallelization [53]. Additionally, iterative convergence remains a critical factor in the performance of methods such as Parareal and PFASST. The convergence rate is influenced by problem characteristics and the choice of coarse and fine integrators [48], and poor convergence can lead to significant performance degradation [54].

1.2.4 Emerging Opportunities with Machine Learning

Machine learning, particularly neural networks, presents a promising approach to overcoming some of the challenges associated with parallel-in-time methods [34, 55]. Neural networks can function as coarse propagators in these methods, offering several key advantages [56, 57]. One major benefit is speed, as once trained, neural networks can be evaluated extremely quickly, making them well-suited for use in coarse propagators [58]. Additionally, neural networks can approximate complex dynamics with reasonable accuracy, potentially reducing the number of iterations required for convergence [59]. Another advantage is their hardware compatibility, as neural networks are highly efficient when deployed on GPUs and other accelerators, allowing for better utilization of heterogeneous high-performance computing (HPC) systems [60]. This work explores the integration of neural networks into parallel-in-time methods, focusing on the Parareal algorithm [7]. By leveraging the strengths of machine learning, we aim to address the bottlenecks in traditional approaches and extend the scalability of numerical algorithms for PDEs [32].

1.3 Neural Networks as Coarse Propagators

The integration of machine learning techniques, particularly neural networks, into numerical algorithms for solving partial differential equations (PDEs) has opened new avenues for enhancing computational efficiency and scalability [35, 33]. Neural networks, with their ability to approximate complex nonlinear functions, are particularly well-suited to serve as coarse propagators in parallel-in-time algorithms such as Parareal and PFASST. This section explores the potential of neural networks as coarse propagators, their advantages, challenges, and implementation details.

1.3.1 Motivation for Using Neural Networks

In parallel-in-time methods, the coarse propagator plays a crucial role in providing an initial approximation of the solution over time [30]. Traditional coarse propagators often rely on simplified physics or reduced-order models, which may struggle to achieve the necessary accuracy or scalability [50]. Neural networks present a compelling alternative due to several advantages [61, 62]. First, they enable fast inference, as once trained, neural networks can predict outputs with minimal computational overhead, making them well-suited for coarse-level time integration [63]. Additionally, they offer data-driven approximation, allowing them to learn complex dynamics directly from data and bypass the need for explicitly deriving reduced-order models. Neural networks also benefit from parallel execution, as they are inherently parallelizable and can be efficiently implemented on modern hardware such as GPUs, further enhancing scalability [64]. Moreover, their adaptability allows them to generalize across different initial conditions and parameter regimes, making them highly versatile for a wide range of applications [65].

1.3.2 Designing Neural Network-Based Coarse Propagators

The architecture and training process of a neural network-based coarse propagator are critical to its performance. Several key considerations must be addressed during the design phase. First, the input and output representation is essential, with inputs typically consisting of the state variables of the partial differential equation (PDE) at the current time step and relevant parameters such as boundary conditions or coefficients, while the output is the predicted state at a future time step [66]. The choice of network architecture depends on the problem characteristics, with recurrent neural networks (RNNs) and their variants, such as long short-term memory networks (LSTMs) and gated recurrent units (GRUs), being suitable for capturing temporal dependencies, while convolutional neural networks (CNNs) are more effective for spatially structured data [67].
Additionally, deep neural networks (DNNs) offer a flexible and powerful alternative, capable of learning complex mappings between input and output states, making them a promising choice for coarse propagators in high-dimensional PDEs [66]. Another critical factor is the loss function, which should accurately capture the deviation of the predicted solution from the reference solution. For PDE-based problems, the loss function may also incorporate physical constraints, such as conservation laws or boundary conditions, to improve generalization and stability [68]. High-quality training data is essential for ensuring accurate predictions, with data typically sourced from high-fidelity simulations or experimental measurements [69]. Finally, regularization and generalization techniques play a key role in preventing overfitting and ensuring robustness to unseen scenarios. Methods such as dropout, weight regularization, and data augmentation help enhance the network's ability to generalize across different problem settings [70].

1.3.3 Integration into Parallel-in-Time Methods

In parallel-in-time algorithms, neural networks serve as coarse propagators within the iterative framework [71]. The workflow generally follows a structured sequence. Initially, the numerical propagator runs for a small number of time steps to generate training data or initialize the iterative process [49]. Next, the neural network-based coarse propagator predicts the solution over a longer time horizon, providing an approximate but computationally efficient solution. The fine propagator then refines this solution by correcting errors introduced by the coarse propagator, a process that can be executed in parallel across multiple time slices. This cycle of coarse and fine propagation is iterated until convergence is achieved. By replacing traditional coarse propagators with neural networks, the computational workload of coarse-level integration is significantly reduced, enhancing the overall efficiency of parallel-in-time algorithms [57, 72, 73].

1.3.4 Advantages of Neural Network Coarse Propagators

The incorporation of neural networks as coarse propagators brings several advantages [33]. Neural networks exhibit good scalability, efficiently handling high-dimensional data, making them particularly well-suited for large-scale simulations [64]. They also offer

an improved accuracy-speed trade-off, as they can balance computational efficiency and predictive accuracy more effectively than traditional reduced-order models, especially when trained on representative datasets [62]. Furthermore, neural networks maximize hardware utilization by leveraging modern accelerators such as Graphical Processing Units (GPUs) and Tensor Processing Units (TPUs), achieving significant speedups in high-performance computing (HPC) environments [60]. Their versatility is another major advantage, as they can adapt to variations in problem setups, such as changing boundary conditions or source terms, without requiring significant modifications [65].

1.3.5 Challenges and Limitations

Despite their promise, neural network-based coarse propagators face several challenges [68]. One primary concern is the high training cost, as the initial training phase can be computationally expensive, particularly for high-dimensional partial differential equations (PDEs) [32]. Additionally, their effectiveness is highly dependent on the availability of high-quality training data, which may not always be accessible for certain applications [69]. Another challenge is extrapolation, as neural networks may struggle to generalize beyond the range of their training data, leading to inaccuracies in extreme or previously unseen scenarios [65]. Finally, interpretability remains a concern, as neural networks are often perceived as black-box models, making it difficult to analyze and explain their predictions compared to traditional physics-based models [34].

1.3.6 Applications and Case Studies

Neural network coarse propagators have been successfully applied across various domains, demonstrating their versatility and effectiveness. In fluid dynamics, they have been used to accelerate simulations of the Navier-Stokes equations, capturing turbulent flows with high accuracy [59]. In weather prediction, neural network-based coarse models have enabled faster forecasting by leveraging historical data for training [74, 75]. Additionally, in computational finance, neural networks have been employed in solving the Black-Scholes equation, allowing for faster option pricing while maintaining precision [36, 76].

The use of neural networks as coarse propagators remains an active area of research, with several promising future directions. One major avenue is the development of Physics-Informed Neural Networks (PINNs), which incorporate governing physical laws directly into their architecture or loss function to ensure compliance with fundamental equations [32, 77]. Hybrid models that combine neural networks with traditional coarse propagators could also provide a balance between accuracy and interpretability, leveraging the strengths of both approaches [78, 79]. Another promising direction is adaptive training, where online or dynamic training methods continuously update the neural network propagator during simulations, improving performance in evolving and complex scenarios [80].

Neural networks as coarse propagators represent a paradigm shift in parallel-in-time

methods for PDEs. By leveraging their speed, scalability, and adaptability, neural networks can address the limitations of traditional coarse propagators and unlock new levels of efficiency in large-scale simulations. However, realizing their full potential requires addressing challenges related to training, generalization, and integration with existing numerical frameworks. This work aims to contribute to this evolving field by exploring the application of neural networks to enhance the Parareal algorithm.
Recent advances have further expanded the integration of neural networks within parallel-in-time (PinT) frameworks. Notably, Ibrahim et al. [40] employed Physics-Informed Neural Networks (PINNs) as coarse propagators in the Parareal algorithm for solving the Black–Scholes equation, achieving considerable speedups and improved parallel efficiency, particularly when leveraging heterogeneous computing resources such as GPUs. Meng et al. [57] introduced the Parareal Physics-Informed Neural Network (PPINN), which decomposes long-time PDE integration into independent subintervals, each solved using PINNs, and demonstrated rapid convergence and scalability. Shukla et al. [81] proposed the XPINN framework, enabling temporal and spatial domain decomposition through weak coupling of PINNs, significantly enhancing parallel capacity. Furthermore, ongoing efforts have explored the use of neural networks as data-driven predictors in space–time parallel solvers, showing promise in accelerating predictor stages and reducing iteration counts in implicit time integration schemes [82]. These developments underscore the growing maturity and utility of neural-network-based propagators in time-parallel simulations, and point to exciting opportunities for hybrid, scalable, and adaptive PinT methodologies.

Chapter 2

Parareal Algorithm

2.1 The Parareal Algorithm

The Parareal algorithm is an iterative time-domain decomposition technique designed to accelerate the numerical solution of time-dependent differential equations by leveraging parallel-in-time computations. Introduced by Lions, Maday, and Turinici in 2001 [7], the method aims to overcome the sequential bottleneck of classical time-stepping schemes by enabling the simultaneous computation of different time intervals. Traditional numerical methods for solving time-dependent problems, such as explicit and implicit time integration schemes, are inherently sequential in nature, limiting their scalability on modern high-performance computing architectures. Parareal circumvents this limitation by iteratively refining a coarse global solution using a computationally expensive fine propagator applied in parallel across multiple time slices. Over successive iterations, the discrepancy between the fine and coarse solutions is corrected, allowing the method to converge to an accurate solution. Since its introduction, Parareal has been widely studied and applied to a variety of problems, including fluid dynamics, quantum mechanics [83], and optimal control [84]. Its efficiency and convergence properties depend on the accuracy of the coarse propagator relative to the fine propagator, as well as the spectral characteristics of the underlying problem. Despite its success, challenges such as load balancing, communication overhead, and convergence behavior for highly stiff problems continue to be active areas of research. Parareal belongs to a broader class of parallel-in-time methods, including the Multigrid Reduction in Time (MGRIT) algorithm [9] and the Parallel Full Approximation Scheme in Space and Time (PFASST) [8], which further extend parallelization strategies. Understanding the theoretical foundations and practical limitations of Parareal remains essential for its effective deployment in large-scale scientific computations.

2.1.1 Fine and Coarse Propagators

We consider the numerical solution of an initial value problem for a system of ODEs resulting from the spatial discretization of a PDE:

$$\frac{d\mathbf{u}}{dt} = A\mathbf{u}(t), \quad \mathbf{u}(t_0) = \mathbf{u}_0, \quad \mathbf{u}(t) \in \mathbb{R}^d,$$

where $A \in \mathbb{R}^{d\times d}$ is the discrete operator obtained, for instance, via finite differences or finite elements. The time domain $[t_0, T]$ is partitioned into N subintervals $I_n = [t_n, t_{n+1}]$, where $t_n = t_0 + n\Delta T$ and $\Delta T = (T - t_0)/N$.

The Parareal algorithm introduces two numerical time integrators:

- **Fine propagator** $\mathcal{F}(t_n, \mathbf{u}_n)$: A high-order, computationally accurate integrator (e.g., Runge-Kutta or implicit methods) that produces an accurate solution over each subinterval I_n.
- **Coarse propagator** $\mathcal{G}(t_n, \mathbf{u}_n)$: A low-order, inexpensive integrator (e.g., explicit Euler or a coarsened version of $\mathcal{F}$) that approximates the solution with less accuracy but faster computation.

The Parareal iteration is given by:

$$\mathbf{u}_{n+1}^{k+1} = \mathcal{G}(t_n, \mathbf{u}_n^{k+1}) + \mathcal{F}(t_n, \mathbf{u}_n^k) - \mathcal{G}(t_n, \mathbf{u}_n^k),$$

where $\mathbf{u}_n^k$ denotes the approximation of the solution at time t_n during the k-th Parareal iteration. The initial iterate is computed using only the coarse integrator:

$$\mathbf{u}_{n+1}^0 = \mathcal{G}(t_n, \mathbf{u}_n^0).$$

This formulation enables the parallelization of time integration by iteratively refining the solution across subintervals while leveraging the efficiency of the coarse solver and the accuracy of the fine solver.

2.1.2 Algorithmic Implementation

Pseudocode for Parareal Algorithm

The Parareal algorithm is applied to systems of ODEs resulting from the spatial discretization of a PDE. Let $\mathbf{u}(t) \in \mathbb{R}^d$ denote the semi-discrete solution vector obtained from finite element, finite difference, or spectral discretization of the spatial domain. The resulting system is of the form:

$$\frac{d\mathbf{u}}{dt} = f(\mathbf{u}, t), \quad \mathbf{u}(t_0) = \mathbf{u}_0.$$

We partition the time interval $[t_0, T]$ into N coarse subintervals $[t_n, t_{n+1}]$ of size $\Delta T = (T - t_0)/N$. Let $\mathcal{G}$ and $\mathcal{F}$ denote the coarse and fine time integrators over each subinterval. The Parareal method proceeds as follows:

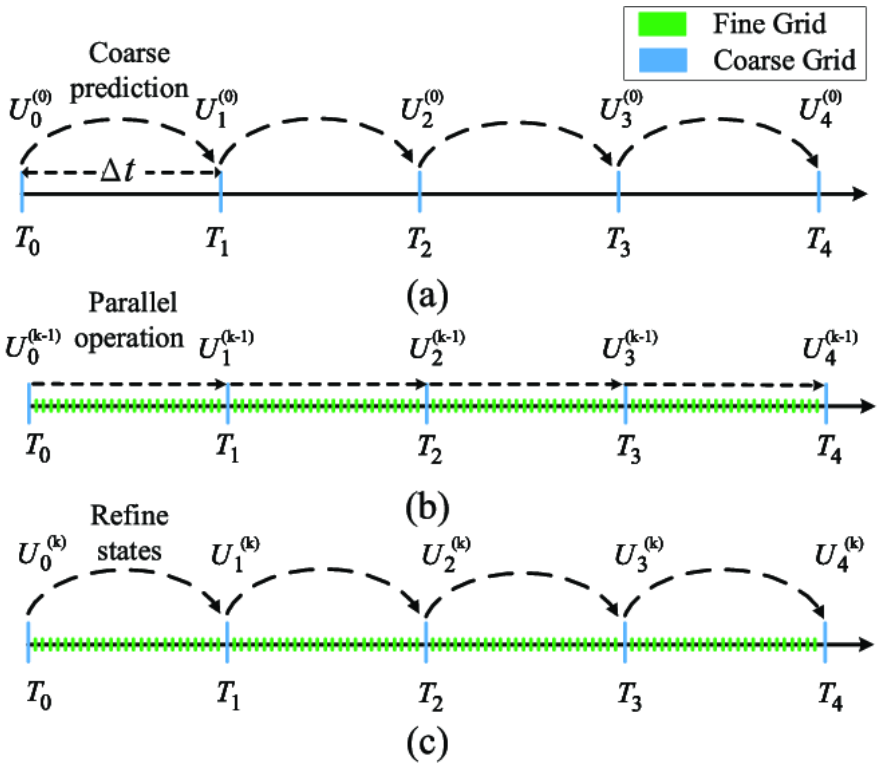


Figure 2.1: Overview of the Parareal algorithm. The time domain is partitioned into subintervals, and fine and coarse propagators are used iteratively to correct the solution. While fine propagators can be evaluated in parallel across subintervals, the coarse propagator typically introduces a serial dependency due to its sequential nature in time. In pipelined implementations, coarse and fine propagations can overlap in time, partially mitigating this bottleneck.

2.2 Convergence Analysis

The error propagation and convergence bound presented in this section follow the classical analysis of the Parareal algorithm under Lipschitz continuity assumptions, formalized by Gander and Hairer [48]. A recent revisitation of this analysis, with explicit bounds for Euler and Runge–Kutta integrators, is provided by Scheiber [85]. For linear problems, a matrix-based convergence framework yielding tight bounds is discussed in Southworth et al. [86].

2.2.1 Error Propagation and Iteration Matrix

Define the error at iteration k as

$$e_n^k = u_n^k - u(t_n), \tag{2.1}$$

Algorithm 1 Parareal Method (Finite-Dimensional Setting)

Require: Initial condition $\mathbf{u}_0$, time interval $[t_0, T]$, number of subintervals N, number of iterations K

Ensure: Approximate solution $\mathbf{u}_n \approx \mathbf{u}(t_n)$ for $n = 0, 1, \ldots, N$

Initialize:
Set $\Delta T = (T - t_0)/N$
Set $t_n = t_0 + n \cdot \Delta T$ for $n = 0, 1, \ldots, N$
Set $\mathbf{u}_0^0 = \mathbf{u}_0$
Initial coarse solution:
for $n = 0$ to $N - 1$ **do**
    $\mathbf{u}_{n+1}^0 = \mathcal{G}(t_n, t_{n+1}, \mathbf{u}_n^0)$
end for
Parareal iterations:
for $k = 0$ to $K - 1$ **do**
    for $n = 0$ to $N - 1$ **(in parallel) do**
        $\mathcal{F}_n^k = \mathcal{F}(t_n, t_{n+1}, \mathbf{u}_n^k)$
    end for
    for $n = 0$ to $N - 1$ **(sequentially) do**
        $\mathbf{u}_{n+1}^{k+1} = \mathcal{G}(t_n, t_{n+1}, \mathbf{u}_n^{k+1}) + \mathcal{F}_n^k - \mathcal{G}(t_n, t_{n+1}, \mathbf{u}_n^k)$ ▷ Correction step
    end for
    Check convergence: If $\max_n \|\mathbf{u}_n^{k+1} - \mathbf{u}_n^k\| < \varepsilon$, break
end for
Return $\mathbf{u}_n = \mathbf{u}_n^K$ for $n = 0, 1, \ldots, N$

where u_n^k is the approximation at iteration k and $u(t_n)$ is the exact solution or "very fine" fine propagator at t_n. From the Parareal iteration formula,

$$u_{n+1}^{k+1} = \mathcal{G}(t_n, u_n^{k+1}) + \mathcal{F}(t_n, u_n^k) - \mathcal{G}(t_n, u_n^k), \tag{2.2}$$

we substitute the exact solution:

$$u(t_{n+1}) = \mathcal{F}(t_n, u(t_n)) \tag{2.3}$$

Subtracting these equations gives the error propagation equation:

$$e_{n+1}^{k+1} = \mathcal{G}(t_n, u_n^{k+1}) - \mathcal{G}(t_n, u(t_n)) + \mathcal{F}(t_n, u_n^k) - \mathcal{F}(t_n, u(t_n)) - \mathcal{G}(t_n, u_n^k) + \mathcal{G}(t_n, u(t_n)) \tag{2.4}$$

Using the Lipschitz continuity of the propagators, we assume that for any $u, v \in \mathbb{R}^d$ (or in the function space $\mathcal{H}$), there exist constants $L_{\mathcal{F}}$, $L_{\mathcal{G}}$ such that:

$$\|\mathcal{F}(t_n, u) - \mathcal{F}(t_n, v)\| \leq L_{\mathcal{F}}\|u - v\|, \tag{2.5}$$

$$\|\mathcal{G}(t_n, u) - \mathcal{G}(t_n, v)\| \leq L_{\mathcal{G}}\|u - v\|, \tag{2.6}$$

for some constants $L_{\mathcal{F}}$ and $L_{\mathcal{G}}$, we obtain the bound:

$$\|e_{n+1}^{k+1}\| \leq L_{\mathcal{G}}\|e_n^{k+1}\| + (L_{\mathcal{F}} - L_{\mathcal{G}})\|e_n^k\| \tag{2.7}$$

2.2.2 Spectral Analysis of Convergence

In this subsection, we restrict our attention to the linear setting, where both the fine and coarse propagators are linear operators. Specifically, suppose

$$\mathcal{F}(t_n, u) = Fu, \quad \mathcal{G}(t_n, u) = Gu,$$

where $F, G \in \mathbb{R}^{d\times d}$ are constant linear operators. In this case, the Parareal iteration becomes linear, and the overall update from iteration k to $k+1$ can be written as:

$$u^{k+1} = Gu^{k+1} + (F - G)u^k,$$

from which we derive the error propagation equation:

$$e^{k+1} = Ge^{k+1} + (F - G)e^k.$$

This leads to the iteration matrix formulation:

$$e^{k+1} = Me^k, \quad \text{where} \quad M = (I - G)^{-1}(F - G),$$

assuming $I - G$ is invertible. The convergence of the method is then governed by the spectral radius $\rho(M)$. In particular,

$$\|e^k\| \leq \rho(M)^k\|e^0\|,$$

provided that M is diagonalizable or norm-bounded in a consistent matrix norm.

Nilpotency in the linear case. For the case of a linear autonomous ODE discretized over N time subintervals with exact linear coarse and fine propagators, it is known that the iteration matrix of the full Parareal method is nilpotent of degree N. That is, the method converges exactly in N iterations:

$$e^N = 0.$$

This property arises from the block structure of the full iteration matrix, which shifts and accumulates corrections in a finite propagation horizon. See Gander and Hairer [48] for a formal derivation.

Lipschitz-based bounds for nonlinear problems. In contrast, for nonlinear problems where $\mathcal{F}$ and $\mathcal{G}$ are not linear maps but satisfy Lipschitz conditions with constants $L_{\mathcal{F}}, L_{\mathcal{G}}$, spectral analysis is no longer appropriate. Instead, we obtain a recursive bound:

$$\|e_{n+1}^{k+1}\| \leq L_{\mathcal{G}}\|e_n^{k+1}\| + (L_{\mathcal{F}} - L_{\mathcal{G}})\|e_n^k\|,$$

which leads to a non-matrix-based convergence estimate. While this does not yield a spectral radius, it can still describe exponential-type decay under favorable conditions. The true iteration matrix exists only in the linearized (Jacobian-based) or discrete linear case [48].

2.2.3 Linear Error Propagation Analysis

To analyze the stability and convergence behavior of Parareal for linear problems, we follow the framework developed by Gander and Vandewalle [48]. Consider the scalar linear test equation:

$$\frac{du}{dt} = \lambda u, \quad u(0) = u_0, \tag{2.8}$$

where $\lambda \in \mathbb{C}$, and define $z = \Delta t\lambda$ as the scaled eigenvalue. Let the fine and coarse propagators be defined by their stability functions:

$$\mathcal{F}(z) = R_F(z), \quad \mathcal{G}(z) = R_G(z),$$

where R_F and R_G are the stability functions of the fine and coarse numerical integrators, respectively.

Gander and Vandewalle show that for this linear setting, the error at Parareal iteration k behaves like:

$$e^k = g(z)^k e^0, \tag{2.9}$$

where the error propagation factor is given by:

$$g(z) = \frac{R_F(z) - R_G(z)}{1 - R_G(z)}. \tag{2.10}$$

The method is stable in this sense if $|g(z)| < 1$ for all relevant z, ensuring geometric error decay across iterations.

Example: Explicit Euler and Implicit Midpoint. Let the coarse propagator be explicit Euler with:

$$R_G(z) = 1 + z,$$

and the fine propagator be the implicit midpoint rule with:

$$R_F(z) = \frac{1 + \frac{z}{2}}{1 - \frac{z}{2}}.$$

Then, the error amplification factor becomes:

$$g(z) = \frac{\frac{1+\frac{z}{2}}{1-\frac{z}{2}} - (1+z)}{1 - (1+z)}. \tag{2.11}$$

This simplifies to:

$$g(z) = \frac{z^2}{4(1 - \frac{z}{2})} + \mathcal{O}(z^3), \tag{2.12}$$

which implies that $g(z) = \mathcal{O}(z^2)$ for small z. Thus, Parareal exhibits second-order error damping in this setting, and stability is guaranteed for sufficiently small time steps. This analysis highlights that the efficiency and stability of Parareal depend critically on the spectral gap between the coarse and fine propagators. Small differences lead to smaller $g(z)$, implying faster convergence per iteration.

2.2.4 Super-linear Convergence Properties

Under certain conditions, the Parareal algorithm exhibits super-linear convergence [48]. Specifically, if the fine propagator $\mathcal{F}$ is exact and the coarse propagator $\mathcal{G}$ is a first-order approximation, then the error satisfies:

$$\|e_n^k\| \leq C \cdot (\Delta T)^k, \tag{2.13}$$

where C is a constant. This implies that the error decreases at an accelerated rate with each iteration, leading to rapid convergence. To understand this, consider that the coarse propagator introduces an approximation error of order $\mathcal{O}(\Delta T)$, whereas the fine propagator resolves the system exactly. Each iteration refines the solution by incorporating more accurate corrections from the fine propagator. Consequently, the error term shrinks proportionally to higher powers of ΔT as the number of iterations increases. This super-linear behavior is particularly beneficial when ΔT is small, as fewer iterations are required to reach a given error tolerance compared to standard linear convergence. In such cases, the required number of iterations K follows an asymptotic relation:

$$K \sim \mathcal{O}\left(\log \frac{1}{\epsilon} / \log \frac{1}{\Delta T}\right), \tag{2.14}$$

indicating that convergence is significantly accelerated when the time step ΔT is refined. Thus, leveraging a sufficiently accurate coarse propagator enhances the efficiency of the Parareal algorithm by reducing the iteration count needed for convergence.

2.3 Speedup Analysis and Complexity

The Parareal algorithm achieves speedup by parallelizing computations over multiple time intervals. However, the practical speedup depends on several factors such as the number of iterations K, the computational cost of the fine and coarse propagators, and communication overhead.

2.3.1 Upper Bound on Speedup

Here and throughout this section, $c_{\mathcal{G}}$ denotes the computational cost (runtime) of one application of the coarse propagator $\mathcal{G}$, and $c_{\mathcal{F}}$ denotes the corresponding cost of one application of the fine propagator $\mathcal{F}$.
We write N for the number of (coarse) time subintervals in the Parareal decomposition, and P for the number of processors. In the ideal mapping (one processor per time slice) one sets $N = P$, but in general N and P can differ depending on available hardware and scheduling.

Let P be the number of processors, and assume the time domain is divided into N subintervals. In the ideal case, $N = P$. The sequential runtime is given by:

$$T_{\text{seq}} = N c_{\mathcal{F}}, \tag{2.15}$$

where $c_{\mathcal{F}}$ is the runtime of the fine propagator per time step. In the parallel setting, the runtime consists of the initial coarse propagation and the iterations of fine corrections:

$$T_{\text{par}} = c_{\mathcal{G}} + K\left(c_{\mathcal{F}} + c_{\mathcal{G}}\right). \tag{2.16}$$

The theoretical speedup is then bounded by [30]:

$$s_{\text{bound}}(P) = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{N c_{\mathcal{F}}}{c_{\mathcal{G}} + K(c_{\mathcal{F}} + c_{\mathcal{G}})}. \tag{2.17}$$

For large P (i.e., many time slices), speedup approaches:

$$s_{\text{bound}}(P) \approx \frac{P}{1 + \frac{K}{P}\left(1 + \frac{c_{\mathcal{G}}}{c_{\mathcal{F}}}\right)}. \tag{2.18}$$

As discussed in [30], near-linear speedup is only attainable when $K \ll P$ and $c_{\mathcal{G}} \ll c_{\mathcal{F}}$. If $K \ll N$, near-optimal speedup is possible. However, if $c_{\mathcal{G}} \approx c_{\mathcal{F}}$, the coarse propagator does not offer significant computational savings, reducing speedup. Additionally, communication and synchronization introduce overhead that limits real-world speedup. Furthermore, load imbalance among processors may degrade performance. Despite these limitations, Parareal can achieve substantial speedup when K remains small and thus $c_{\mathcal{G}} \ll c_{\mathcal{F}}$, making it an effective parallel-in-time method for solving differential equations.

2.3.2 Computational Complexity Analysis

The computational complexity of the Parareal algorithm depends on the cost of the fine and coarse propagators, the number of time subintervals N, and the number of Parareal iterations K required for convergence.

- **Sequential baseline:** Without parallelization, the overall cost is dominated by the fine solver, scaling as $\mathcal{O}(N \cdot c_{\mathcal{F}})$, where $c_{\mathcal{F}}$ is the cost of the fine propagator per subinterval.
- **Ideal parallel complexity:** Under perfect parallelism with N processors, each processor performs one fine propagation, and the coarse propagator is applied sequentially K times. The ideal complexity becomes:

  $$\mathcal{O}(K \cdot c_{\mathcal{G}} + c_{\mathcal{F}}),$$

  assuming negligible communication and coarse steps are fast. For time-parallelism to be beneficial, it is essential that $K \ll N$ and $c_{\mathcal{G}} \ll c_{\mathcal{F}}$.
- **Communication and practical limits:** In realistic settings, communication between processors introduces additional overhead, scaling as $\mathcal{O}(K \cdot N \cdot C_{\text{comm}})$, where C_{comm} is the communication cost per time slice per iteration. This can reduce speedups, especially on architectures with high communication latency.
- **Design implications:** The efficiency and scalability of Parareal depend on:
  - Keeping the iteration count K small (via accurate coarse models or better initial guesses),
  - Ensuring the coarse propagator is significantly cheaper than the fine one ($c_{\mathcal{G}} \ll c_{\mathcal{F}}$),
  - Minimizing communication overhead (C_{comm}) on parallel architectures,
  - Recognizing that each processor still performs a fine solve, which remains a non-trivial cost, particularly for stiff or nonlinear PDEs.

Overall, this analysis provides a guideline for the practical use of Parareal: improve coarse models, minimize communication, and keep iteration counts low to fully exploit time-parallelism.

2.4 Extensions and Variants of Parareal

2.4.1 Multi-level Parareal

The Multi-level Parareal algorithm [1] generalizes the classical two-level Parareal method by introducing a hierarchy of propagators with varying resolutions and computational

costs. Instead of relying on a single coarse and fine solver, the algorithm defines multiple levels of accuracy, enabling more efficient convergence and improved parallel performance on deep time grids. Let $l = 0, 1, \ldots, L$ denote the level index, where $l = 0$ corresponds to the finest level (most accurate but expensive), and $l = L$ is the coarsest level (least accurate but cheapest). Each level uses its own propagator $\mathcal{G}_l$, where finer levels correct the errors of coarser ones recursively. The multi-level Parareal iteration at level l is defined as:

$$u_{n+1}^{k+1,l} = \mathcal{G}_l(t_n, u_n^{k+1,l}) + \mathcal{G}_{l-1}(t_n, u_n^{k,l-1}) - \mathcal{G}_l(t_n, u_n^{k,l-1}),$$

where:

- $u_n^{k,l}$ denotes the approximation at time t_n, level l, and iteration k,
- $\mathcal{G}_l$ is the propagator at level l,
- The update uses a correction based on the discrepancy between the level-l and level-$l-1$ propagations.

The algorithm proceeds from the coarsest level L downward to the finest level $l = 0$, applying this recursive correction at each level. The outermost iteration (on level L) is typically performed in serial, while the fine propagations on lower levels can be computed in parallel across time subintervals. To improve stability and convergence, Rosemeier et al. introduce an *averaging procedure* across time subintervals when transitioning between levels. This mitigates the oscillatory behavior of coarse corrections and helps stabilize error propagation across the hierarchy. The averaging operator can be applied either temporally (across iterations) or spatially (across subintervals), and its choice influences the smoothing properties of the method. Compared to classical Parareal, the multi-level formulation:

- Reduces the number of required fine solves by deferring most of the correction to cheaper, coarser propagators,
- Admits better scalability by allowing finer propagations to be applied in parallel with corrections,
- Provides improved convergence in stiff or oscillatory problems through hierarchical smoothing.

This multi-level design shares conceptual similarities with multigrid methods and can be viewed as a time-parallel analog, where temporal resolution is adaptively refined via recursive corrections.

2.4.2 Krylov-Enhanced Parareal

The Krylov-enhanced Parareal algorithm [87] accelerates convergence of the standard Parareal method by exploiting information from previous iterations using Krylov subspace techniques. This is particularly useful when the convergence of standard Parareal is slow due to inadequate coarse corrections or stiffness in the underlying system. The core idea is to treat the correction term in Parareal as a linear (or approximately linear) operator, and apply projection-based acceleration methods—similar in spirit to GMRES—to iteratively refine the solution. Instead of relying solely on the most recent fine-coarse difference, Krylov-enhanced Parareal constructs a low-dimensional subspace from previous corrections and projects the new correction onto this space. The Krylov-enhanced Parareal iteration is given by:

$$u_{n+1}^{k+1} = \mathcal{G}(t_n, u_n^{k+1}) + \mathcal{P}_k \left[\mathcal{F}(t_n, u_n^j) - \mathcal{G}(t_n, u_n^j)\right]_{j=0}^{k},$$

where:

- $\mathcal{P}_k$ is a projection operator onto the Krylov subspace generated by the correction vectors from iterations $j = 0$ to k,
- The term $\mathcal{F}(t_n, u_n^j) - \mathcal{G}(t_n, u_n^j)$ is the Parareal correction at iteration j,
- A least-squares projection (as in GMRES) is typically used to minimize the residual in the span of past corrections.

Construction of the Krylov subspace. Let the correction vectors be:

$$\mathbf{r}_j = \mathcal{F}(t_n, u_n^j) - \mathcal{G}(t_n, u_n^j), \quad j = 0, \ldots, k,$$

and define the Krylov space:

$$\mathcal{K}_k = \text{span}\{\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_k\}.$$

Then the projected correction is obtained by finding the linear combination of $\mathbf{r}_j$'s that best approximates the current residual in a least-squares sense. Overall, Krylov-enhanced Parareal blends time-parallelism with Krylov acceleration to achieve improved scalability and robustness for challenging time-dependent problems.

2.4.3 MGRIT (Multigrid Reduction in Time)

MGRIT (Multigrid Reduction in Time) [9] is a parallel-in-time algorithm that extends the Parareal method by applying multigrid principles to the time domain. Unlike traditional time-stepping methods, which are inherently sequential, MGRIT enables parallelization across time steps, making it particularly useful for large-scale simulations where conventional time integration becomes a bottleneck. MGRIT operates by constructing a hierarchy of temporal grids with varying levels of resolution. Coarse grids

approximate the solution over larger time intervals, while fine grids resolve details at smaller scales. The algorithm employs relaxation schemes, such as F-relaxation (fine-grid updates) and C-relaxation (coarse-grid updates), to iteratively refine the solution. By leveraging multigrid techniques like interpolation and restriction, MGRIT efficiently propagates information across different time levels, accelerating convergence.

2.5 Performance Optimization Techniques

2.5.1 Adaptive Coarse Propagator Selection

The choice of coarse propagator plays a critical role in determining the convergence rate and overall efficiency of the Parareal algorithm. To optimize performance, adaptive techniques can be employed to select the most suitable coarse propagator based on the specific characteristics of the problem. One approach is model reduction [88], which involves using reduced-order models that retain the essential dynamics of the system while being computationally efficient. Another strategy is spatial coarsening, where a lower-resolution spatial discretization is applied to the coarse propagator, reducing the complexity of the computations. Temporal coarsening is also an effective method, utilizing larger time steps for the coarse propagator to decrease the computational load. Additionally, physics-based simplification can be implemented by neglecting certain terms or processes in the model, further reducing the computational cost while maintaining an acceptable level of accuracy [89].

2.5.2 Communication Optimization

Minimizing communication overhead is crucial for achieving scalability in parallel algorithms. One effective technique is asynchronous communication, which allows computation and communication to overlap by leveraging non-blocking communication primitives [90]. This approach helps to hide communication latency and improve overall efficiency. Another method is data compression [91], which reduces the volume of data transferred between processors by employing compression techniques, thereby decreasing communication time. Additionally, topology-aware mapping can be utilized to optimize the placement of computational tasks across processors. By mapping tasks in a way that minimizes communication distances, this strategy ensures that data exchange occurs more efficiently, further enhancing the scalability of the algorithm.

Chapter 3

Machine Learning for PDEs

3.1 Machine Learning for PDEs

Machine learning techniques have shown promising potential in complementing traditional numerical methods for solving partial differential equations (PDEs), particularly in high-dimensional and data-driven settings. They offer an alternative to traditional numerical methods, enabling efficient and accurate approximations of complex solutions. Machine learning methods can be used to learn the solution operator of a PDE, allowing for rapid evaluations of the solution at different parameter values. This is particularly useful in applications where the PDE needs to be solved repeatedly for different parameters, such as in optimization problems or uncertainty quantification. In this section, we provide a rigorous mathematical formulation of deep neural networks (DNNs), detailing their structure, optimization, and approximation properties. We also explore Physics-Informed Neural Networks (PINNs) [32] and Physics-Informed Neural Operators (PINOs) [92, 93], which are two prominent approaches that leverage deep learning to solve and analyze partial differential equations (PDEs).

3.2 Feedforward Neural Networks

A feedforward neural network (FNN) is a function $\mathcal{F}_\theta : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ parameterized by θ, which consists of multiple layers of neurons. Given an input $x \in \mathbb{R}^{d_{in}}$, the output is computed as:

$$x^{(l+1)} = \sigma(W^{(l)}x^{(l)} + b^{(l)}), \quad l = 1, \dots, L, \tag{3.1}$$

where:

- $x^{(l)} \in \mathbb{R}^{d_l}$ represents the activation of the l-th layer,
- $W^{(l)} \in \mathbb{R}^{d_{l+1} \times d_l}$ is the weight matrix,
- $b^{(l)} \in \mathbb{R}^{d_{l+1}}$ is the bias vector,
- $\sigma : \mathbb{R} \to \mathbb{R}$ is an activation function applied element-wise.

The final output $x^{(L+1)}$ represents the network's prediction $\mathcal{F}_\theta(x)$. The set of all parameters is denoted as $\theta = \{W^{(l)}, b^{(l)}\}_{l=1}^{L}$.

3.2.1 Activation Functions

The choice of activation function σ plays a crucial role in the network's ability to approximate complex functions. Common activation functions include:

- **ReLU (Rectified Linear Unit):** $\sigma(x) = \max(0, x)$,
- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$,
- **Tanh:** $\sigma(x) = \tanh(x)$.

For deep networks, ReLU is widely preferred due to its non-saturating property and efficient gradient propagation [94].

3.2.2 Universal Approximation Theorem

A fundamental result in neural network theory states that a sufficiently wide neural network can approximate any continuous function arbitrarily well.

Theorem 3.1 (Universal Approximation)**.** *Let $C(K)$ be the space of continuous functions on a compact domain $K \subset \mathbb{R}^{d_{in}}$. For any $f \in C(K)$ and $\epsilon > 0$, there exists a single-hidden-layer neural network $\mathcal{F}_\theta$ such that:*

$$\sup_{x \in K} |f(x) - \mathcal{F}_\theta(x)| < \epsilon. \tag{3.2}$$

This result [95, 96] guarantees the expressive power of neural networks but does not provide information about the number of neurons required or how to train such networks. Lower bounds on the required number of neurons to achieve a given tolerance ϵ reveal that, in the worst case, the network size grows exponentially with input dimension or function complexity. This makes shallow networks no more efficient than traditional numerical schemes, especially for functions with low regularity or high intrinsic dimensionality. However, under structural assumptions—such as compositionality, sparsity, or smoothness—deep networks can achieve significantly better approximation rates. These insights form the basis of why deep architectures are used in scientific machine learning, despite the classical limitations of shallow universal approximators.

3.2.3 Training Neural Networks: Optimization Perspective

Neural networks are trained by minimizing a loss function $\mathcal{L}(\theta)$ using gradient-based optimization. Given a dataset $\{(x_i, y_i)\}_{i=1}^{N}$, the empirical loss is defined as:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(\mathcal{F}_\theta(x_i), y_i), \tag{3.3}$$

where $\ell : \mathbb{R}^{d_{out}} \times \mathbb{R}^{d_{out}} \to \mathbb{R}$ is a loss function, commonly chosen as:

- Mean Squared Error (MSE) for regression: $\ell(y, \hat{y}) = \|y - \hat{y}\|^2$,
- Cross-Entropy Loss for classification: $\ell(y, \hat{y}) = -\sum_{j=1}^{d_{out}} y_j \log(\hat{y}_j)$.

The loss function is minimized using gradient descent algorithms such as:

- **Stochastic Gradient Descent (SGD)**:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}(\theta^{(t)}), \tag{3.4}$$

  where η is the learning rate.

- **Adam Optimization [97]:** An adaptive gradient-based method combining momentum and Root Mean Square Propagation (RMSprop). Adam updates parameters using moment estimates of past gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{3.5}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{3.6}$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \tag{3.7}$$
$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \tag{3.8}$$

  Here, g_t is the gradient at iteration t, and β_1, β_2 are decay rates controlling the moving averages of the first and second moments.

3.3 Physics-Informed Neural Networks (PINNs)

Physics-Informed Neural Networks (PINNs) [32] are a class of deep learning models that incorporate the underlying physics of a system, typically expressed as a partial differential equation (PDE), directly into the training objective. Unlike conventional neural networks that learn from data alone, PINNs enforce the governing equations as soft constraints, enabling them to approximate solutions to PDEs even in the absence of extensive labeled data. Assume a general PDE of the form:

$$\mathcal{N} u(x, t) = f(x, t), \quad \text{for } (x, t) \in \Omega \times (0, T], \tag{3.9}$$

subject to initial and boundary conditions:

$$u(x, 0) = u_0(x), \qquad \text{for } x \in \Omega, \tag{3.10}$$
$$u(x, t) = g(x, t), \qquad \text{for } (x, t) \in \partial\Omega \times [0, T]. \tag{3.11}$$

Here, $\mathcal{N}$ is a (possibly nonlinear) differential operator, $u(x, t)$ is the unknown solution, $f(x, t)$ is a known source term, $u_0(x)$ is the initial condition, and $g(x, t)$ defines the prescribed boundary values. A PINN approximates the solution $u(x, t)$ by a neural network

$u_\theta(x,t)$, where θ denotes the network parameters. The key idea is to penalize violations of the PDE and the initial/boundary conditions by defining a composite loss function:

$$\mathcal{L}(\theta) = \underbrace{\mathcal{L}_{\text{res}}(\theta)}_{\text{PDE residual loss}} + \lambda_b \underbrace{\mathcal{L}_{\text{bc}}(\theta)}_{\text{Boundary loss}} + \lambda_i \underbrace{\mathcal{L}_{\text{ic}}(\theta)}_{\text{Initial loss}}, \tag{3.12}$$

where:

$$\mathcal{L}_{\text{res}}(\theta) = \frac{1}{|\Omega|T} \int_0^T \int_\Omega \left| \mathcal{N} u_\theta(x,t) - f(x,t) \right|^2 dx\, dt, \tag{3.13}$$

$$\mathcal{L}_{\text{bc}}(\theta) = \frac{1}{|\partial\Omega|T} \int_0^T \int_{\partial\Omega} \left| u_\theta(x,t) - g(x,t) \right|^2 dx\, dt, \tag{3.14}$$

$$\mathcal{L}_{\text{ic}}(\theta) = \frac{1}{|\Omega|} \int_\Omega \left| u_\theta(x,0) - u_0(x) \right|^2 dx. \tag{3.15}$$

The weights λ_b and λ_i balance the relative contributions of boundary and initial condition enforcement. In practice, the integrals are approximated using Monte Carlo sampling or quadrature over collocation points. By minimizing this physics-informed loss, PINNs provide a mesh-free approximation to the solution $u(x,t)$ that satisfies both the data (if available) and the governing physical laws.

3.3.1 Network Architecture and Design Considerations

The architecture of a PINN significantly influences its performance and accuracy. A central aspect is the ability to compute derivatives of the neural network output with respect to its inputs, which is essential for evaluating the PDE residuals in the loss function. These derivatives are typically computed using automatic differentiation, which ensures exact gradients and simplifies the implementation of complex differential operators.

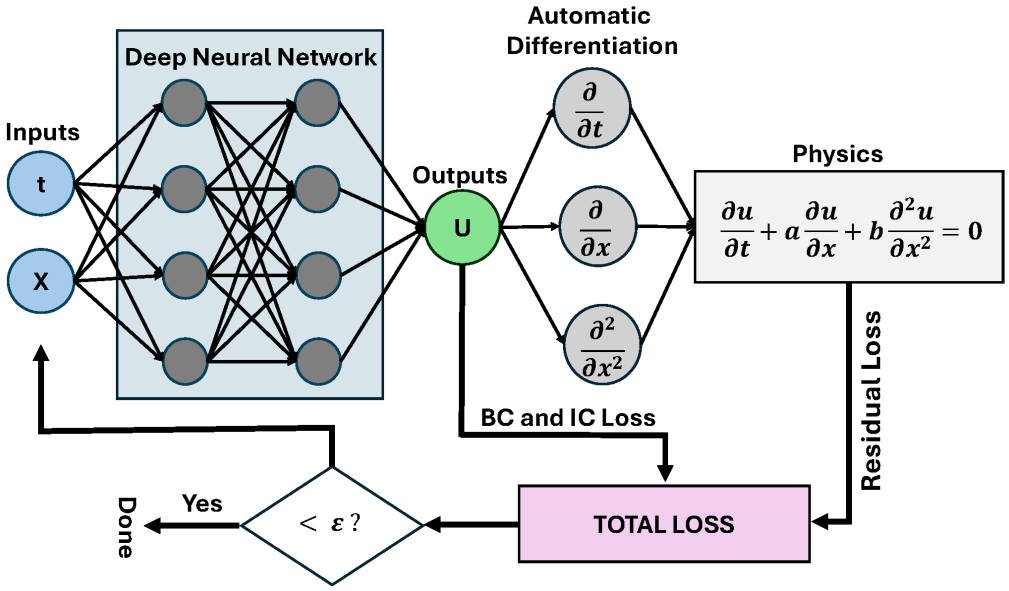


Figure 3.1: Conceptual architecture of a Physics-Informed Neural Network (PINN) [98]. The network receives space-time coordinates as input and outputs an approximation of the PDE solution. The loss function incorporates the PDE residual, initial conditions, and boundary conditions, relying on automatic differentiation to compute the necessary spatial and temporal derivatives.

Network Architecture Considerations

The design of the neural network architecture significantly influences the performance of Physics-Informed Neural Networks (PINNs). Deeper networks can capture more complex relationships due to their increased capacity, but they are also more prone to overfitting, especially when the available data is limited. Wider networks, on the other hand, can represent a larger number of features, but they often require more data to train effectively. The choice of activation function is another critical factor, as it directly impacts the network's ability to approximate complex functions and its training dynamics. Smooth activation functions like Tanh or Sigmoid are well-suited for approximating continuous solutions, while ReLU and its variants enable faster training but may introduce non-smoothness in the solution [96]. For problems involving high-frequency components, Sine activation functions, as used in SIREN networks [99], have proven effective. Additionally, incorporating residual connections between layers can mitigate the vanishing gradient problem and improve training stability. To enhance the network's ability to capture high-frequency features, Fourier features can be integrated as input transformations, providing a richer representation of the solution space.

3.3.2 Training Strategies for PINNs

Training PINNs effectively requires careful consideration of several strategies and optimization techniques. Sampling strategies play a crucial role in ensuring that the network captures the underlying solution accurately. Uniform sampling is simple to implement but may fail to resolve regions of high complexity or rapid variation [32]. Adaptive sampling addresses this by concentrating more points in areas with high error or significant changes [100], while importance sampling weights samples based on their contribution to the loss function [101]. The design of the loss function is equally important, with static weighting providing fixed weights for different loss components, dynamic weighting adjusting weights during training based on the relative magnitude of each component [102], and adaptive weighting tailoring weights to the difficulty of satisfying each constraint [103]. The choice of optimizer also impacts training efficiency, with Adam being a robust choice for most scenarios, Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS) offering faster convergence for well-behaved problems [104], and Stochastic Gradient Descent (SGD) requiring careful tuning of learning rates. Curriculum learning, which involves starting with simpler problems and gradually increasing complexity, can further improve training stability and convergence [105].

3.3.3 Handling Non-Smooth Solutions and Discontinuities

PINNs often face challenges when dealing with non-smooth solutions or discontinuities, which frequently arise in physical systems with shocks, material interfaces, or phase transitions. One effective approach is domain decomposition, where the computational domain is divided into subdomains, and separate neural networks are trained in each region [106]. This allows the networks to specialize in capturing localized features and reduces the approximation complexity. Another strategy involves adaptive activation functions, which modify their shape during training to better approximate sharp gradients or discontinuities [107]. These functions can dynamically adjust nonlinearity to improve expressiveness near singularities. Although the standard PINN formulation is based on minimizing the residual of the governing PDE, recent work has emphasized more robust residual formulations—such as weighting residuals spatially or using Sobolev training—to focus the approximation effort where errors are largest or solutions are less smooth [32]. In cases with discontinuities across interfaces, explicitly enforcing interface conditions (e.g., continuity of fluxes or values) between subdomains can improve consistency and accuracy [108]. These hybrid approaches significantly enhance the ability of PINNs to tackle problems with limited regularity.

3.3.4 Error Analysis and Convergence Guarantees

The approximation error of PINNs can be analyzed using tools from approximation theory, statistical learning theory, and optimization. Key results include:

1. **Approximation Error Bounds**: Under some regularity assumptions, neural networks can approximate functions in Sobolev spaces with error scaling as $\mathcal{O}(N^{-\alpha/d})$, where N is the number of neurons, d is the input dimension, and α reflects the smoothness of the target function [109]. This bound highlights the curse of dimensionality for general shallow networks, although deep architectures can mitigate this under structural assumptions.

2. **Generalization Error**: The generalization performance of PINNs depends on both the network complexity (depth, width, and number of parameters) and the number and distribution of training/collocation points. For overparameterized networks, recent work suggests that gradient descent introduces *implicit regularization*, which can help avoid overfitting and improve generalization [110]. However, this effect is still not fully understood in the context of PDE loss landscapes.

3. **Convergence Rates**: For certain classes of PDEs, especially linear elliptic and parabolic problems with smooth solutions, PINNs can exhibit *exponential convergence* in terms of the number of collocation points M, provided that the underlying function is analytic and the network is sufficiently expressive [111]. Specifically, the error may decay as $\mathcal{O}(e^{-c\sqrt{M}})$ under idealized conditions involving uniform sampling, bounded domain, and suitable choice of activation functions. However, this behavior is not universal and depends heavily on both the PDE type and training dynamics.

3.4 Physics-Informed Neural Operators (PINOs)

Physics-Informed Neural Operators (PINOs) extend the concept of PINNs to the operator-learning setting, where the goal is to learn mappings between function spaces rather than pointwise approximations. This makes PINOs particularly well-suited for solving parametric PDEs, where the inputs may include spatial and temporal variables as well as parameters $a \in \mathcal{A}$ such as material properties, boundary data, or forcing terms.
Consider a family of PDEs parameterized by $a \in \mathcal{A}$, given by:

$$\mathcal{N}u(x,t;a) = f(x,t;a), \quad \text{for } (x,t) \in \Omega \times (0,T], \tag{3.16}$$

with initial and boundary conditions:

$$u(x,0;a) = u_0(x;a), \qquad \text{for } x \in \Omega, \tag{3.17}$$

$$u(x,t;a) = g(x,t;a), \qquad \text{for } (x,t) \in \partial\Omega \times [0,T]. \tag{3.18}$$

Here, $\mathcal{N}$ is a (possibly nonlinear) differential operator, and the functions f, g, u_0 depend on the parameter a. The goal of a PINO is to approximate the solution operator:

$$\mathcal{G} : a \mapsto u(\cdot,\cdot;a), \tag{3.19}$$

which maps problem parameters to the corresponding PDE solution over the spatiotemporal domain.

This operator is approximated by a neural network $\mathcal{G}_\theta$, typically instantiated using neural operator architectures such as the Fourier Neural Operator (FNO) or DeepONet. These architectures are specifically designed to learn nonlinear maps between infinite-dimensional function spaces.
Training is performed over multiple instances $a \sim p(a)$, and the loss function is constructed to enforce the physical constraints across all realizations of the PDE:

$$\mathcal{L}(\theta) = \underbrace{\mathcal{L}_r(\theta)}_{\text{Residual loss}} + \lambda_b \underbrace{\mathcal{L}_b(\theta)}_{\text{Boundary loss}} + \lambda_i \underbrace{\mathcal{L}_i(\theta)}_{\text{Initial loss}}, \tag{3.20}$$

with:

- **Residual loss (PDE constraint)**:

$$\mathcal{L}_r(\theta) = \mathbb{E}_{a\sim p(a)} \left[\frac{1}{|\Omega|} \int_\Omega \left| \mathcal{N}[\mathcal{G}_\theta(a)](x,t) - f(x,t;a) \right|^2 dx \right]. \tag{3.21}$$

- **Boundary loss**:

$$\mathcal{L}_b(\theta) = \mathbb{E}_{a\sim p(a)} \left[\frac{1}{|\partial\Omega|} \int_{\partial\Omega} \left| \mathcal{G}_\theta(a)(x,t) - g(x,t;a) \right|^2 dx \right]. \tag{3.22}$$

- **Initial condition loss**:

$$\mathcal{L}_i(\theta) = \mathbb{E}_{a\sim p(a)} \left[\frac{1}{|\Omega|} \int_\Omega \left| \mathcal{G}_\theta(a)(x,0) - u_0(x;a) \right|^2 dx \right]. \tag{3.23}$$

The expectation $\mathbb{E}_{a\sim p(a)}$ reflects training over a distribution of PDE parameters, and λ_b, λ_i are weighting factors. By minimizing this composite loss, PINOs learn a parametric family of PDE solutions in a way that respects the underlying physics. This enables fast inference across parameter ranges, making PINOs highly effective for real-time simulation, uncertainty quantification, and many-query applications such as inverse problems and control [112].

3.4.1 DeepONet Architecture

DeepONet [58] is a neural operator architecture that approximates the action of nonlinear operators using the universal approximation theorem for operators. It is designed to learn mappings between function spaces, making it suitable for solving PDEs with varying parameters or initial conditions.
The architecture of DeepONet consists of two main components: a branch network and a trunk network. The branch network processes the input function a, while the trunk network processes the spatial coordinates x.

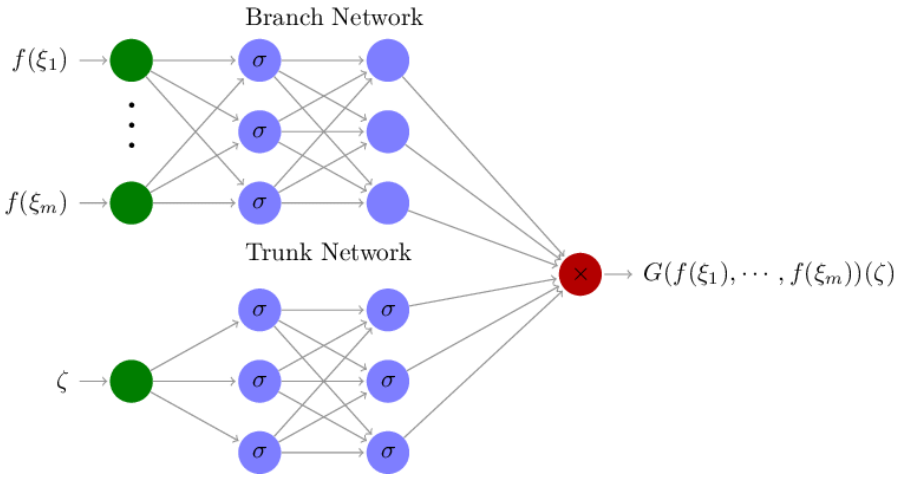


Figure 3.2: Architecture of DeepONet [113]. The branch network processes the input function a, while the trunk network processes the spatial coordinates x. The output is a linear combination of the outputs from both networks.

The branch network is a feedforward neural network that takes the input function a and outputs a set of coefficients $b_i(a)$. It can be represented as:

$$b_i(a) = \sum_{j=1}^{m} w_{ij}\sigma\left(W_j a + b_j\right)$$

where w_{ij} are the weights of the branch network, σ is a nonlinear activation function, and W_j and b_j are the weights and biases of the j-th layer. The trunk network is also a feedforward neural network that takes the spatial coordinates x and outputs a set of basis functions $t_i(x)$. It can be represented as:

$$t_i(x) = \sum_{k=1}^{n} v_{ik}\sigma\left(V_k x + c_k\right)$$

where v_{ik} are the weights of the trunk network, σ is a nonlinear activation function, and V_k and c_k are the weights and biases of the k-th layer.
The outputs of the trunk network are basis functions that represent the spatial coordinates x. The output of the DeepONet is computed as:

$$G_\theta(a)(x) = \sum_{i=1}^{p} b_i(a) \cdot t_i(x)$$

where $b_i(a)$ are the outputs of the branch network and $t_i(x)$ are the outputs of the trunk network.
This form can be interpreted as a nonlinear operator regression model, where the solution $u(x) = G(a)(x)$ is approximated by a learned separable representation. The branch

network produces input-dependent coefficients (similar to expansion coefficients in basis functions), while the trunk network evaluates spatial basis functions. Together, they approximate the operator as:

$$G(a)(x) \approx \sum_{i=1}^{p} \phi_i(a)\, \psi_i(x),$$

where $b_i(a) \approx \phi_i(a)$ and $t_i(x) \approx \psi_i(x)$, with ϕ_i, ψ_i representing unknown functional components of the true operator decomposition.
The branch network and trunk network are trained simultaneously to minimize the loss function, which measures the discrepancy between the predicted output and the true output of the operator. The loss function can be defined as:

$$\mathcal{L}(\theta) = \frac{1}{|\Omega|} \int_{\Omega} |G_\theta(a)(x) - f(x;a)|^2 \, dx + \lambda_b \frac{1}{|\partial\Omega|} \int_{\partial\Omega} |G_\theta(a)(x) - g(x;a)|^2 \, dx,$$

where λ_b is a weighting factor for the boundary loss.

Operator Learning Perspective. Formally, let $\mathcal{A}$ be a Banach space of input functions a, and let $G : \mathcal{A} \to C(\Omega)$ be a nonlinear operator. DeepONet aims to learn this operator by minimizing the expected loss:

$$\mathbb{E}_{a\sim p(a)} \left[\int_{\Omega} |G_\theta(a)(x) - G(a)(x)|^2 \, dx \right],$$

where $p(a)$ is a probability distribution over the input function space $\mathcal{A}$.
In practice, a is represented by sampling its values at a fixed set of sensor points $\{x_j\}_{j=1}^m$:

$$\mathbf{a} = (a(x_1), \ldots, a(x_m)) \in \mathbb{R}^m,$$

and passed through the branch network $\mathcal{B}_\theta : \mathbb{R}^m \to \mathbb{R}^p$ to generate the latent coefficients. Similarly, the trunk network $\mathcal{T}_\theta : \Omega \to \mathbb{R}^p$ processes the evaluation point x and outputs the basis functions. Then:

$$G_\theta(a)(x) = \langle \mathcal{B}_\theta(a), \mathcal{T}_\theta(x) \rangle.$$

Approximation Theory. DeepONet exhibits universal approximation capabilities for nonlinear operators. Specifically, it was shown in [114] that for any compact set $\mathcal{A} \subset C(\Omega)$ and any continuous operator $G : \mathcal{A} \to \mathbb{R}$, there exists a DeepONet architecture such that:

$$\sup_{a\in\mathcal{A}} |G_\theta(a)(x) - G(a)(x)| < \varepsilon, \quad \forall x \in \Omega,$$

for any $\varepsilon > 0$, provided the network width p and sensor resolution m are sufficiently large. For Lipschitz continuous operators, the approximation error with respect to sensor resolution scales as $\mathcal{O}(m^{-1/2})$, where m is the number of sensors used to discretize

the input function a. While this rate is similar to Monte Carlo convergence in terms of sampling resolution, DeepONet offers a crucial advantage: it learns the entire operator mapping, enabling generalization across a family of input functions. This allows for significantly fewer training examples compared to pointwise surrogate models that require separate training for each input–output pair.

Generalization and Practical Considerations. The generalization performance of DeepONet depends on both architectural capacity and the diversity of the training set. For complex or stiff PDEs, the number of basis functions p may need to increase to capture intricate operator behavior. Important design factors include:

- The choice of sensor locations $\{x_j\}$ in Ω affects the expressive power of the branch network. These can be chosen via uniform sampling, Gaussian quadrature nodes, or optimized via gradient descent.

- The trunk network implicitly defines a learned basis for the output space. Unlike fixed basis functions (e.g., Fourier or polynomial), DeepONet adapts these to the target operator.

- DeepONet can be extended to handle vector-valued or spatiotemporal output by modifying the trunk network to accommodate multi-output architectures.

DeepONet provides theoretical guarantees for approximating nonlinear operators, including universal approximation results for continuous operators and convergence bounds in terms of sensor resolution. The network possesses universal approximation properties, enabling it to approximate continuous nonlinear operators with arbitrary accuracy when provided sufficient network capacity. Furthermore, the model's generalization performance is intricately linked to the complexity of the underlying operator and the richness and representativeness of the training dataset, highlighting the importance of both architectural design and data quality in achieving robust operator learning [115].

3.4.2 Fourier Neural Operator (FNO)

Fourier Neural Operators (FNOs), introduced by Li et al. [92], represent a class of neural operators that leverage the Fourier transform to learn mappings between function spaces. FNOs are particularly well-suited for solving partial differential equations (PDEs) with periodic or structured boundary conditions. Unlike traditional neural networks that operate in the spatial domain, FNOs learn the evolution of functions in the frequency domain using spectral convolution layers, allowing them to exploit the global structure of the input functions.

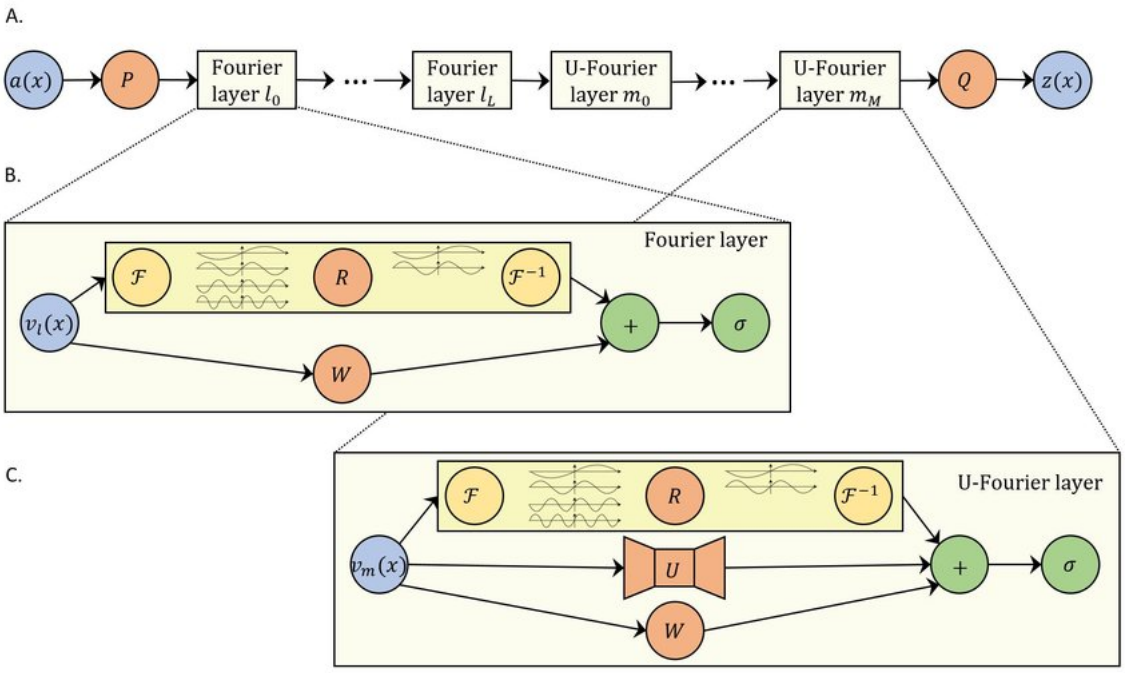


Figure 3.3: Architecture of the Fourier Neural Operator (FNO). The input function is lifted to a higher-dimensional space, followed by convolution in the Fourier domain, and finally projected back to the original space.

Architecture Overview. The FNO consists of a sequence of spectral convolution layers that operate in Fourier space, interleaved with nonlinear activation functions and pointwise transformations in the spatial domain. The architecture can be described in three main steps:

1. **Lifting:** The input function $a(x)$ is lifted to a higher-dimensional representation via a pointwise (local) neural network $P : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^w$, where w is the width of the feature space.

2. **Fourier Layers:** A sequence of L Fourier layers is applied, each consisting of a spectral convolution defined as:

$$v_{j+1}(x) = \sigma\left(W_j v_j(x) + \mathcal{F}^{-1}\left(R_j \cdot \mathcal{F}(v_j)\right)\right), \tag{3.24}$$

   where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the (discrete) Fourier transform and its inverse, R_j is a learnable complex-valued filter in Fourier space, W_j is a pointwise linear map, and σ is a nonlinear activation function (e.g., GELU or ReLU).

3. **Projection:** After the final layer, the output is mapped back to the target dimension using another pointwise neural network $Q : \mathbb{R}^w \to \mathbb{R}^{d_{\text{out}}}$, producing the predicted solution $u(x) \approx G_\theta(a)(x)$.

The learnable Fourier filter $R_j(k)$ is usually applied to only a fixed number of low-frequency modes $|k| \leq K$ to improve computational efficiency and regularize the model. The high-frequency modes are typically zeroed out, acting as an implicit spectral smoother.

Mathematical Justification. FNOs exploit the fact that solutions of many elliptic and parabolic PDEs are smooth or even analytic, causing their Fourier coefficients $\hat{u}(k)$ to decay rapidly as $|k| \to \infty$. This enables a low-dimensional representation in frequency space. Recent theoretical analysis has quantified this: for PDEs like Darcy flow or the incompressible Navier–Stokes equations, with sufficiently smooth (analytic or Sobolev-regular) input and solution functions, FNOs can approximate the solution operator with error ε using a model whose size grows only sub-linearly or log-linearly in $1/\varepsilon$ (i.e., polynomial or log-linear scaling) [116]. This is in stark contrast to worst-case neural networks or DeepONets which may require exponential scaling with error tolerance unless strong structural assumptions are made. The regularity of the solution dictates the convergence rate of the error with respect to grid resolution: if the solution lies in a Sobolev space $H^s(\Omega)$, the discretization (aliasing) error decays as $\mathcal{O}(N^{-s})$, where N is the Fourier grid resolution per dimension [117].
Let $\mathcal{F}(u)(k) = \hat{u}(k)$ be the Fourier coefficient of u at mode k. Then the spectral convolution is defined by

$$\mathcal{K}(u)(x) := \mathcal{F}^{-1}\left(R(k) \cdot \hat{u}(k)\right),$$

which corresponds to a global convolution with a kernel that is translation invariant in Fourier space. The learnable multiplier $R(k) \in \mathbb{C}^{w \times w}$ encodes the operator's behavior for each frequency mode.

Loss Function. The training objective is to minimize the discrepancy between the predicted and true solution across the domain:

$$\mathcal{L}(\theta) = \mathbb{E}_{a \sim p(a)} \left[\int_{\Omega} |G_\theta(a)(x) - u(x; a)|^2 \, dx \right],$$

with possible augmentation by physics-based residual losses in physics-informed settings.

Advantages and Limitations. FNOs offer several notable advantages, making them powerful tools for learning parametric solution operators:

- **Resolution-agnostic evaluation:** FNOs are constructed in the Fourier domain and operate on functions discretized over uniform grids. Due to their functional formulation, they can be evaluated on input functions sampled at different resolutions than those used during training. However, this does not imply that FNOs generalize perfectly across resolutions. In practice, accuracy may degrade if evaluated at a resolution significantly higher than the training grid, since the model has only learned frequency content up to the training resolution. Thus, the often-claimed "mesh-independence" is more accurately understood as the ability to interpolate across discretizations, rather than full resolution-agnostic generalization.

- **Spectral efficiency:** For smooth solutions, FNOs exploit the rapid decay of high-frequency modes in the Fourier domain, allowing them to approximate solution operators with relatively few parameters and fast convergence. This efficiency stems from the fact that many elliptic and parabolic PDEs yield solutions with high Sobolev or analytic regularity. However, for non-smooth or discontinuous solutions, this advantage diminishes, as high-frequency content becomes significant and harder to capture accurately.
- **Computational speed:** The Fast Fourier Transform (FFT) is used in practice, enabling $\mathcal{O}(n \log n)$ complexity per layer for n grid points.
- **Nonlocal receptive fields:** FNO layers can capture long-range interactions, unlike CNNs, which rely on stacked convolutions to approximate nonlocality.

However, FNOs also have limitations:

- **Boundary Conditions:** FNOs are naturally suited to problems with periodic boundary conditions due to the Fourier basis. Extensions to non-periodic domains require padding, windowing, or hybrid approaches (e.g., wavelets or adaptive basis).
- **Geometry Restriction:** FNOs are defined on rectangular domains with uniform grids. Applications to irregular domains or manifolds require geometric adaptations, such as graph or manifold neural operators.
- **Spectral leakage and aliasing:** Truncating high frequencies may lead to aliasing or blurred approximations, especially when the target solution exhibits sharp gradients or discontinuities.

Fourier Neural Operators (FNOs) provide a highly efficient framework for operator learning, particularly for PDEs with smooth, globally structured solutions. By leveraging spectral representations, they achieve fast convergence, parameter efficiency, and strong generalization. Their limitations in handling complex geometries and boundary conditions motivate current research in generalized neural operators that combine spectral, local, and geometric representations [112].

3.4.3 Training Strategies for PINOs

Training Physics-Informed Neural Operators (PINOs) effectively requires the implementation of specific strategies tailored to their unique challenges and requirements. One critical aspect in the design of PINNs is how training information is provided. Unlike traditional supervised learning, PINNs are typically trained using a physics-based loss function that enforces the governing differential equation, along with initial and boundary conditions. In this setting, no labeled data (e.g., solution snapshots) is required — only knowledge of the PDE and the domain. However, in practice, additional data

can be incorporated to improve accuracy or guide training in complex regimes. For instance, synthetic data from high-fidelity numerical simulations may be used to pre-train the model or supplement the physics loss in a hybrid approach. This can be especially beneficial in regions with poor convergence or where the physics loss alone does not sufficiently constrain the solution. Additionally, transfer learning can be employed to leverage knowledge from simpler problems, accelerating the training process for more complex scenarios. The design of the loss function is another key consideration, where weighted loss functions can be used to assign different weights to various components of the loss, ensuring balanced optimization. Multi-scale loss functions can also be incorporated to capture both global and local features of the solution, enhancing the network's ability to model complex behaviors.
Regularization techniques are essential to improve training stability and generalization. Spectral normalization, which constrains the spectral norm of the network weights, can prevent overfitting, while gradient regularization penalizes large gradients to ensure smoother and more stable training. These strategies collectively contribute to the successful training of PINOs, enabling them to tackle challenging physical problems with greater accuracy and efficiency.

3.4.4 Error Analysis and Convergence Guarantees

PINOs extend the framework of PINNs by learning operators (mappings between function spaces), making them more effective for solving families of PDEs. Key theoretical aspects include:

1. **Approximation Error Bounds**: PINOs learn operators mapping from input functions (e.g., coefficients or initial conditions) to solution fields, rather than individual solutions. Under suitable Sobolev or Hölder regularity assumptions on the target operator, the approximation error behaves like

$$\mathcal{O}(N^{-\beta}),$$

where N denotes network width or number of training sensors, and β depends on both the smoothness of the operator and the architecture design (e.g., Fourier layers, graph layers). In particular, when the operator maps between smooth function spaces, PINOs can mitigate the curse of dimensionality and achieve polynomial—or even exponential—decay in error with increasing N. This is supported by recent work deriving generic approximation error bounds for operator-learning architectures, including physics-informed variants [118].

2. **Generalization Error**: Physics-Informed Neural Operators (PINOs) demonstrate improved generalization compared to PINNs by learning mappings between infinite-dimensional function spaces rather than individual solutions. This operator-based formulation allows PINOs to generalize across families of parametric PDEs and perform zero-shot inference on unseen inputs. Their ability to generalize stems

from training on multiple PDE instances and fine-tuning with physics-based constraints. This results in higher robustness and accuracy under distributional shifts [93]. The generalization behavior of PINOs is influenced by:

- The chosen architecture (e.g., Fourier Neural Operator, DeepONet).
- The diversity and coverage of the training distribution over PDE instances.
- The smoothness and structural complexity of the underlying solution operator.

By leveraging learned inductive biases at the operator level, PINOs are able to extrapolate more effectively to novel PDE parameters than solution-specific PINNs.

3. **Convergence Rates**: PINOs achieve exponential convergence for certain operator classes, outperforming PINNs and traditional numerical methods. The rate depends on:

   - The spectral decay of the target function/operator.
   - The resolution of training data (e.g., function discretization).
   - Network architecture (depth, activation functions).

   Spectral-based architectures (e.g., Fourier Neural Operators) enhance convergence, particularly for smooth problems.

3.5 Difference betwween PINNs and PINOs

Physics-Informed Neural Networks (PINNs) and Physics-Informed Neural Operators (PINOs) are two prominent deep learning-based approaches for solving partial differential equations (PDEs). While both methods leverage neural networks to incorporate physical constraints, their fundamental approaches differ significantly. PINNs approximate individual solutions to PDEs by minimizing the residual of the governing equations, whereas PINOs learn solution operators that map function spaces to function spaces, making them more generalizable. PINNs have been widely used for solving forward and inverse PDE problems, particularly in cases where traditional numerical solvers struggle due to high-dimensionality or complex geometries. However, they often face challenges related to optimization, generalization, and computational efficiency. On the other hand, PINOs extend the capabilities of neural networks by learning mappings between function spaces, enabling them to generalize across different PDE instances and overcome some limitations of PINNs. The following table highlights key differences between PINNs and PINOs:

Aspect	PINNs	PINOs
Target	Solves for individual PDE solutions	Learns solution operators (function-to-function maps)
Curse of Dimensionality	Performance degrades in high dimensions	Can mitigate dimensionality issues under specific assumptions
Generalization	Tailored to a specific PDE instance	Generalizes across a family of PDEs
Convergence Rate	Training-dependent and often slow	Potential for faster (exponential) convergence
Error Bounds	Depend on neurons N and input dimension d	Depend on operator smoothness and spectral decay

Table 3.1: Comparison of PINNs and PINOs

As shown in Table 3.1, PINNs are well-suited for solving specific PDE instances, especially when no labeled data is available. However, they may struggle with high-dimensional problems and lack robustness when applied to varying initial or boundary conditions or changing parameters within a given PDE family. By contrast, PINOs employ operator learning to approximate mappings from input functions (e.g., coefficients or initial conditions) to solutions, enabling efficient inference across a range of parametric variations within the same PDE class. This makes them particularly useful in applications where many instances of the same PDE—differing by inputs, source terms, or boundary data—must be solved. Despite these advantages, PINOs also pose challenges. Their training requires a sufficiently diverse dataset of PDE instances spanning the target function space. Moreover, the choice of operator-learning architecture (e.g., Fourier Neural Operator or DeepONet) significantly affects performance.
Finally, theoretical understanding of PINOs is still evolving, particularly with regard to error bounds, stability under perturbations, and convergence guarantees.

3.6 Hybridization of Parareal and Machine Learning Approaches

Combining the Parareal algorithm with machine learning techniques can lead to novel hybrid methods that leverage the strengths of both approaches. Machine learning, particularly deep learning—offers flexible function approximation tools that can model complex dynamics, learn from data, and incorporate physical constraints when designed appropriately. In recent years, neural networks have shown promise in accelerating numerical solvers, learning surrogate models, and encoding physics directly into their architecture (e.g., PINNs and PINOs).
By hybridizing Parareal with machine learning, we aim to overcome limitations in classical numerical solvers—such as slow convergence, limited parallel scalability, or rigid discretization requirements—while simultaneously enhancing the interpretability and physical consistency of learned models. This synergy has led to several innovative

frameworks, from learning-augmented coarse solvers to fully neural variants of Parareal, which we describe in detail below.

3.6.1 Learning-Enhanced Parareal

In the learning-enhanced Parareal approach, the classical Parareal algorithm is augmented with machine learning techniques to improve the accuracy of the coarse propagator. Traditionally, Parareal combines a coarse (fast, low-fidelity) propagator $\mathcal{G}$ and a fine (high-accuracy) propagator $\mathcal{F}$, iteratively correcting the coarse solution with fine updates. In the enhanced framework, a neural network is employed to learn the discrepancy between the two:

$$\mathcal{G}_{\text{ML}}(t_n, u_n) = \mathcal{G}(t_n, u_n) + \Delta\mathcal{G}_\theta(t_n, u_n),$$

where $\Delta\mathcal{G}_\theta$ is a neural network trained to approximate the error between the fine and coarse propagators. The training objective is to minimize the supervised loss:

$$\mathcal{L}(\theta) = \mathbb{E}_{u_n}\left[\|\mathcal{F}(t_n, u_n) - \mathcal{G}_{\text{ML}}(t_n, u_n)\|^2\right].$$

Several approaches in the literature follow similar principles. For example, the Parareal Physics-Informed Neural Network (PPINN) [57] solves PDEs in parallel by placing physics-informed neural networks across subdomains of the time interval, achieving both data efficiency and scalability. More recently, RandNet-Parareal [72] introduced random neural networks to directly learn the coarse–fine discrepancy, improving both convergence and wall-clock speed. In contrast to these methods, our approach focuses on using PINNs or PINOs directly as coarse propagators, without requiring additional learning of residual correction terms. This avoids extra training overhead while still improving Parareal convergence, particularly in PDEs where physical structure (e.g., conservation laws or boundary conditions) can be effectively encoded via physics-informed training.

3.6.2 Neural Parareal

Neural Parareal is a variant of the classical Parareal algorithm in which the coarse propagator is entirely replaced by a neural network surrogate. The update rule becomes:

$$u_{n+1}^{k+1} = \mathcal{G}_\theta(t_n, u_n^{k+1}) + \mathcal{F}(t_n, u_n^k) - \mathcal{G}_\theta(t_n, u_n^k),$$

where $\mathcal{G}_\theta$ is a neural network trained to approximate the fine propagator $\mathcal{F}$, but with significantly reduced computational cost. The goal is for $\mathcal{G}_\theta$ to serve as a fast, differentiable approximation of the time evolution operator, enabling the Parareal method to accelerate convergence across time slices while maintaining accuracy. This approach has been explored in prior works, including the PPINN method [57], which uses physics-informed neural networks in each subdomain, and RandNet-Parareal [72], which employs random feature networks to construct data-driven coarse propagators. Our work follows this general Neural Parareal formulation but differs in several important aspects:

- We use Physics-Informed Neural Networks (PINNs) or Physics-Informed Neural Operators (PINOs) as the coarse propagator, thereby encoding physical structure directly into the surrogate model.
- Unlike some prior approaches, our networks are trained entirely on data from the fine solver or the Parareal iteration process, depending on the variant (standard NN, amplitude-loss, or Parareal-trained).

This makes our method a specific instantiation of Neural Parareal with physics-informed models and training objectives tailored to financial PDEs.

3.6.3 Parareal for Training Acceleration

An alternative application of the Parareal algorithm is to accelerate the training of PINNs or PINOs by parallelizing the forward pass during training. This idea interprets the evaluation of the neural network (particularly in physics-informed learning) as solving an ODE or PDE system over a discretized time domain. Specifically, the Parareal method is applied to the computational time associated with evaluating the forward operator, not necessarily the physical time of the PDE.
Let the full time domain $[0, T]$ be divided into N subintervals $I_n = [t_n, t_{n+1}]$, and suppose the neural network is used to approximate the solution on each interval. The total loss function can then be written as a sum over subdomains:

$$\mathcal{L}(\theta) = \sum_{n=0}^{N-1} \mathcal{L}_n(\theta),$$

where $\mathcal{L}_n(\theta)$ represents the loss contribution from interval I_n. Each subproblem can be solved in parallel using the Parareal algorithm, which provides an approximate forward solution on each interval. These partial loss terms can then be aggregated and used in a global optimization step to update the network parameters θ. This parallel-in-time training strategy is particularly beneficial for long time horizons, stiff problems, or PDEs with fine temporal resolution requirements, where sequential evaluation of the loss would be a major computational bottleneck. Several recent works [68] have demonstrated its potential to reduce wall-clock training time without sacrificing solution accuracy.

3.6.4 Challenges and Future Directions

The integration of the Parareal algorithm with machine learning (ML) techniques offers an approach to solving time-dependent PDEs. However, several challenges must be overcome. One major hurdle is ensuring stability and convergence, particularly given the black-box nature of neural networks which can introduce unpredictable behavior. Adaptive strategies that tailor the ML integration to the problem's specific characteristics are needed, as well as methods to ensure that ML models generalize well to new problems and parameter regimes, avoiding overfitting.

Another critical aspect is the incorporation of uncertainty quantification to enhance the robustness of ML predictions, which is essential for precise error control. Future research should focus on seamless multi-scale integration, where processes across multiple spatial and temporal scales are coupled efficiently. Physics-informed regularization that embeds physical constraints directly into the ML model, interpretable models that reveal insights into the underlying physics, and automated hyperparameter tuning for both Parareal and ML components will be key to advancing these hybrid methods. Additionally, leveraging specialized hardware like GPUs and TPUs will be vital for achieving significant computational speedups. A particularly significant challenge in deploying ML-based coarse propagators within the Parareal framework is their integration into massively parallel HPC architectures. In large-scale simulations involving $\mathcal{O}(10^3)$ nodes, each node typically handles a small spatial subdomain and has its own GPU or accelerator. Efficient deployment of neural network-based propagators in such settings requires careful consideration of memory locality, communication patterns, and inference latency. Unlike traditional coarse solvers, neural networks may not easily scale across distributed environments without incurring substantial overhead from data movement or redundant model loading. Furthermore, synchronizing model inference across nodes while preserving scalability and fault tolerance adds an additional layer of complexity. Addressing these challenges calls for the development of distributed inference strategies, model compression techniques, and domain decomposition-aware training approaches that align with the parallel structure of the underlying PDE solver.
Overall, by combining the parallel efficiency of the Parareal method with the approximation power of neural networks, these hybrid techniques promise accelerated solutions for time-dependent PDEs. Addressing challenges related to stability, convergence, generalization, and uncertainty quantification will pave the way for robust, efficient, and widely applicable computational tools in science and engineering.

Chapter 4

Computational Finance: The Black-Scholes Equation

4.1 Overview of the Black-Scholes Equation (BSE)

The Black-Scholes equation is a fundamental partial differential equation (PDE) in financial mathematics, governing the price evolution of European options. It was first derived by Black and Scholes [119], and independently by Merton [120]. The equation plays a crucial role in computational finance, particularly in the pricing and hedging of derivatives. The Black-Scholes equation models the price $u(S,t)$ of a European-style option as a function of the underlying asset price S and time t. A European-style option is a financial derivative that can only be exercised at its expiration date, in contrast to American options, which can be exercised at any time before expiry. Under the assumptions of no arbitrage, a frictionless market, and constant volatility σ, the equation takes the form:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} + rS\frac{\partial u}{\partial S} - ru = 0, \tag{4.1}$$

where:

- $u(S,t)$ is the option price or our unknown function,
- S is the underlying asset price,
- t is time to expiration,
- σ is the volatility of the underlying asset,
- r is the risk-free interest rate.

Equation (4.1) is a deterministic parabolic PDE, structurally similar to the heat equation. Its analytical tractability makes it a cornerstone of classical option pricing theory. For more complex derivatives or models where no closed-form solution exists, numerical techniques such as finite difference methods are typically used. While Monte Carlo simulations are widely employed for pricing derivatives based on the underlying stochastic

differential equation, they do not solve the PDE directly. In recent years, data-driven methods, including deep learning-based approximations, have also been explored as alternatives for solving the Black-Scholes equation and its generalizations.

4.1.1 European vs. American Options

The Black-Scholes model provides a closed-form solution for European options, which can only be exercised at expiration. However, American options allow early exercise, making their pricing more complex. The valuation of American options requires solving a free-boundary problem, which is typically approached using numerical methods such as finite difference methods with early exercise conditions, binomial and trinomial trees and Monte Carlo methods with least squares regression (Longstaff-Schwartz algorithm) [121]. For American put options, early exercise is optimal when the option is deep in the money, requiring adjustments beyond the standard Black-Scholes framework [122, 123].

Put and Call Options

Options come in two main types:

- **Call Option:** Gives the holder the right to buy the underlying asset at a predetermined price (strike price) before or at expiration.
- **Put Option:** Gives the holder the right to sell the underlying asset at the strike price before or at expiration.

4.1.2 The Greeks: Sensitivity Measures

The Greeks[1] are key risk measures derived from the Black-Scholes model, indicating how option prices respond to changes in market conditions:

- **Delta (Δ)**: Sensitivity of the option price to changes in the underlying asset price.

$$\Delta = \frac{\partial u}{\partial S} = N(d_1) \text{ (Call)}, \quad \Delta = N(d_1) - 1 \text{ (Put)}. \tag{4.2}$$

- **Gamma (Γ)**: Sensitivity of delta to changes in the underlying price.

$$\Gamma = \frac{\partial^2 u}{\partial S^2} = \frac{N'(d_1)}{S\sigma\sqrt{T-t}}. \tag{4.3}$$

- **Vega (ν)**: Sensitivity of the option price to volatility changes.

$$\nu = \frac{\partial u}{\partial \sigma} = S\sqrt{T-t}N'(d_1). \tag{4.4}$$

[1]These sensitivity measures are called "Greeks" because they are commonly denoted by Greek letters (e.g., Δ, Γ, Θ). The naming tradition follows conventions from physics and mathematics. Interestingly, "vega" is not actually a Greek letter, but the name is widely used in practice.

- **Theta (Θ)**: Sensitivity of the option price to time decay.

$$\Theta = \frac{\partial u}{\partial t}. \tag{4.5}$$

- **Rho (ρ)**: Sensitivity of the option price to changes in the risk-free interest rate.

$$\rho = \frac{\partial u}{\partial r}. \tag{4.6}$$

These measures are crucial for hedging strategies and risk management in financial markets.

4.2 Single-Asset (1D) Black-Scholes Equation

The Black-Scholes equation, a parabolic partial differential equation (PDE), lacks closed-form solutions for many option pricing scenarios, necessitating numerical approaches. Common methods include the explicit and implicit finite difference schemes, both of which approximate derivatives by discretizing time and asset price dimensions. Despite the existence of closed-form solutions for simple European options, real-world applications often involve complexities such as:

- Path-dependent options (e.g., Asian, barrier, and American options),
- Stochastic volatility models,
- Market frictions (e.g., transaction costs),
- Jump-diffusion processes.

4.2.1 Analytic Soluton for the Single-Asset Options

The Black-Scholes PDE (4.1) admits an exact analytical solution for European options through appropriate transformations. We present a rigorous derivation for the call option price; the put option solution follows via the put-call parity relation [122], which connects the prices of European calls and puts with identical strike and maturity.

Transformation to Heat Equation

First, we reduce (4.1) to the standard heat equation via the following change of variables:

$$x = \ln S, \quad \tau = T - t, \quad u(x, \tau) = e^{r\tau} V(S, t). \tag{4.7}$$

Applying these transformations to (4.1) yields:

$$\frac{\partial u}{\partial \tau} = \frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2} + \left(r - \frac{1}{2}\sigma^2\right) \frac{\partial u}{\partial x}. \tag{4.8}$$

We eliminate the drift term by introducing:

$$\xi = x + \left(r - \frac{1}{2}\sigma^2\right)\tau, \quad u(x,\tau) = w(\xi,\tau), \tag{4.9}$$

which reduces (4.8) to the canonical heat equation:

$$\frac{\partial w}{\partial \tau} = \frac{1}{2}\sigma^2 \frac{\partial^2 w}{\partial \xi^2}. \tag{4.10}$$

Solution via Probability Methods

The terminal condition $u(S,T) = \max(S-K,0)$ transforms to:

$$w(\xi,0) = \max(e^{\xi} - K, 0). \tag{4.11}$$

The solution to (4.10) is given by the convolution of the initial condition with the heat kernel:

$$w(\xi,\tau) = \frac{1}{\sigma\sqrt{2\pi\tau}} \int_{-\infty}^{\infty} w(y,0) \exp\left(-\frac{(\xi-y)^2}{2\sigma^2\tau}\right) dy. \tag{4.12}$$

Evaluating this integral yields:

$$w(\xi,\tau) = e^{\xi + \frac{1}{2}\sigma^2\tau} N\left(\frac{\xi - \ln K + \sigma^2\tau}{\sigma\sqrt{\tau}}\right) - KN\left(\frac{\xi - \ln K}{\sigma\sqrt{\tau}}\right), \tag{4.13}$$

where $N(\cdot)$ denotes the standard normal cumulative distribution function:

$$N(d) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{d} e^{-z^2/2} dz. \tag{4.14}$$

Inversion to Original Variables

Reverting to the original variables through (4.9) and (4.7), we obtain the call price:

$$C(S,t) = SN(d_1) - Ke^{-r(T-t)}N(d_2), \tag{4.15}$$

where:

$$d_1 = \frac{\ln(S/K) + \left(r + \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}, \tag{4.16}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}. \tag{4.17}$$

Put Option via Parity

The put option price follows from the put-call parity relation $C - P = S - Ke^{-r(T-t)}$:

$$P(S,t) = Ke^{-r(T-t)}N(-d_2) - SN(-d_1). \tag{4.18}$$

Interpretation of Terms

- $N(d_2)$: Risk-neutral probability, here the option expires in the money, i.e., under the risk-neutral measure $\mathbb{P}(S_T \geq K)$.
- $N(d_1)$: Delta of the option; it represents the sensitivity of the option price to small changes in the underlying asset price, also known as the hedge ratio.
- $Ke^{-r(T-t)}$: Present value of the strike price, discounted at the risk-free interest rate.
- $\frac{S}{Ke^{-r(T-t)}}$: A dimensionless measure of moneyness; values greater than 1 indicate an in-the-money call option, while values less than 1 indicate out-of-the-money.

Traditional numerical methods, such as finite difference methods and Monte Carlo simulations, can be computationally expensive and may suffer from convergence issues. Recent advances in neural networks offer an alternative approach by leveraging function approximation capabilities to solve PDEs efficiently.

4.2.2 Explicit Finite Difference Scheme

To numerically solve the Black-Scholes equation we discretize the asset price S and time t into a grid with steps ΔS and Δt, respectively. Let $u_{i,j}$ represent the option value at asset price $S_i = i\Delta S$ and time $t_j = j\Delta t$. The partial derivatives can be approximated as follows:

$$\frac{\partial u}{\partial t} \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}, \tag{4.19}$$

$$\frac{\partial u}{\partial S} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta S}, \tag{4.20}$$

$$\frac{\partial^2 u}{\partial S^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta S)^2}. \tag{4.21}$$

Substituting these approximations into the Black-Scholes equation yields:

$$u_{i,j+1} = u_{i,j} + \Delta t \left[-\frac{1}{2}\sigma^2 S_i^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta S)^2} - rS_i \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta S} + ru_{i,j} \right]. \tag{4.22}$$

This explicit scheme updates the option value sequentially over the grid.

4.2.3 Implicit Finite Difference Scheme

The implicit finite difference method improves stability by computing values at the next time step implicitly. Instead of using values from the current time step to update the

solution explicitly, the implicit method solves a system of equations at each time step. Rewriting the Black-Scholes equation using an implicit time discretization:

$$\frac{u_{i,j} - u_{i,j-1}}{\Delta t} + \frac{1}{2}\sigma^2 S_i^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta S)^2} + rS_i \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta S} - ru_{i,j} = 0. \quad (4.23)$$

Rearranging, we obtain a tridiagonal system:

$$-a_i u_{i-1,j} + (1 + b_i)u_{i,j} - c_i u_{i+1,j} = u_{i,j-1}, \quad (4.24)$$

where

$$a_i = \frac{\Delta t}{2}\left[\sigma^2 S_i^2/(\Delta S)^2 - rS_i/(2\Delta S)\right], \quad (4.25)$$

$$b_i = -\Delta t\left[\sigma^2 S_i^2/(\Delta S)^2 + r\right], \quad (4.26)$$

$$c_i = \frac{\Delta t}{2}\left[\sigma^2 S_i^2/(\Delta S)^2 + rS_i/(2\Delta S)\right]. \quad (4.27)$$

This system can be solved using the Thomas algorithm, a specialized method for tridiagonal matrices [124].

4.2.4 Stability and Error Analysis

The implicit scheme is unconditionally stable, allowing for larger time steps compared to the explicit method. The explicit scheme requires careful selection of the time step Δt to ensure stability. Due to the variable coefficient $\sigma^2 S^2$ in the diffusion term, the classical Courant-Friedrichs-Lewy (CFL) condition must be adapted. The appropriate stability condition for the explicit scheme becomes:

$$\Delta t \leq \frac{(\Delta S)^2}{\sigma^2 S_{\max}^2}, \quad (4.28)$$

where $S_{\max}$ is the maximum asset price in the discretized domain. This condition ensures the numerical solution does not exhibit instability [125, 126]. By contrast, the implicit scheme is unconditionally stable, meaning that it does not impose a restriction on the time step Δt for stability. However, unconditional stability does not guarantee accuracy; using excessively large time steps can still lead to significant numerical errors. Thus, a careful balance between stability and accuracy must be maintained when choosing Δt. Both schemes involve iterating over the discretized grid, with the explicit method directly updating values and the implicit method solving a system of equations. The accuracy of the numerical solution can be assessed using the relative error in the L_1-norm:

$$\text{Relative Error} = \frac{\sum_i |u_{i,\text{numerical}} - u_{i,\text{analytical}}|}{\sum_i |u_{i,\text{analytical}}|}. \quad (4.29)$$

This metric guides adjustments in grid resolution or time stepping for improved precision. While the explicit scheme is easier to implement, it requires small time steps for stability. Implicit schemes allow for larger time steps without sacrificing stability. Methods such as the Crank–Nicolson method, combine the advantages of both explicit and implicit schemes. Although they require solving a system of equations at each time step—making them more computationally intensive—they achieve higher accuracy and maintain stability even with relatively large time steps.

4.2.5 Example 1: Single-Asset European Call and Put Options

As a baseline, we consider closed-form solutions to the single-asset Black–Scholes model for European call and put options. The computational domain is the asset price range $S \in [1, 300]$ and time to maturity $\tau \in [0, 1]$. We consider an at-the-money option with strike price $K = 100$, risk-free interest rate $r = 1.0$, volatility $\sigma = 0.3$, and maturity $T = 1$, where "at-the-money" refers to the case when the strike price equals the current price of the underlying asset. These values are consistent with the two-asset setup and allow us to test solution behavior near the most sensitive pricing region.
The closed-form Black–Scholes solutions [119] for the call and put option prices at current time $t = 0$ (i.e., $\tau = T$) are given by:

$$C(S, \tau) = S\Phi(d_1) - Ke^{-r\tau}\Phi(d_2), \tag{4.30}$$
$$P(S, \tau) = Ke^{-r\tau}\Phi(-d_2) - S\Phi(-d_1), \tag{4.31}$$

where

$$d_1 = \frac{\ln(S/K) + (r + 0.5\sigma^2)\tau}{\sigma\sqrt{\tau}}, \tag{4.32}$$
$$d_2 = d_1 - \sigma\sqrt{\tau}, \tag{4.33}$$

and $\Phi(\cdot)$ denotes the cumulative distribution function (CDF) of the standard normal distribution. The terminal payoffs for the call and put options are:

$$C_T(S) = \max(S - K, 0), \tag{4.34}$$
$$P_T(S) = \max(K - S, 0). \tag{4.35}$$

The strike price K does not explicitly appear in the PDE but is reflected in the terminal condition. In this example, the solution is evaluated at $\tau = T = 1$.

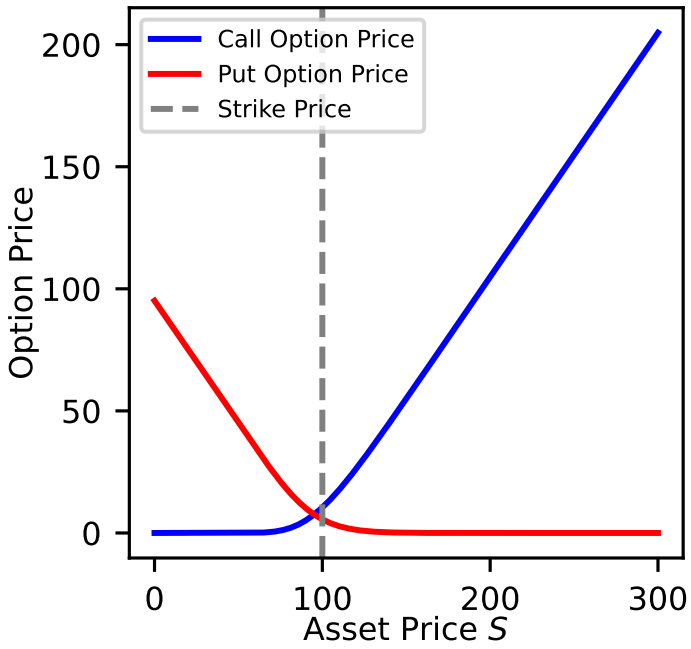


Figure 4.1: Closed-form Black–Scholes prices for single-asset European call (blue) and put (red) options with $K = 100$, $r = 1.0$, $\sigma = 0.3$, and $T = 1$. As expected, the call price increases with the asset price S, while the put price decreases. At the strike price $S = K = 100$, the call and put prices reflect the time value of the options. The vertical dashed line shows the strike price.

4.2.6 Example 2: Single-Asset Options with Different Parameters

To further explore option pricing behavior, we consider a second example with altered market parameters. Here, the strike price is set to $K = 150$ (i.e., deeper out-of-the-money for the call), the volatility is reduced to $\sigma = 0.2$, and the risk-free rate is lowered to $r = 0.05$. The maturity remains at $T = 1$. These conditions simulate a more conservative market scenario. We use the same Black–Scholes closed-form solutions as before:

$$C(S, \tau) = S\Phi(d_1) - Ke^{-r\tau}\Phi(d_2), \tag{4.36}$$
$$P(S, \tau) = Ke^{-r\tau}\Phi(-d_2) - S\Phi(-d_1), \tag{4.37}$$

with

$$d_1 = \frac{\ln(S/K) + (r + 0.5\sigma^2)\tau}{\sigma\sqrt{\tau}}, \tag{4.38}$$
$$d_2 = d_1 - \sigma\sqrt{\tau}. \tag{4.39}$$

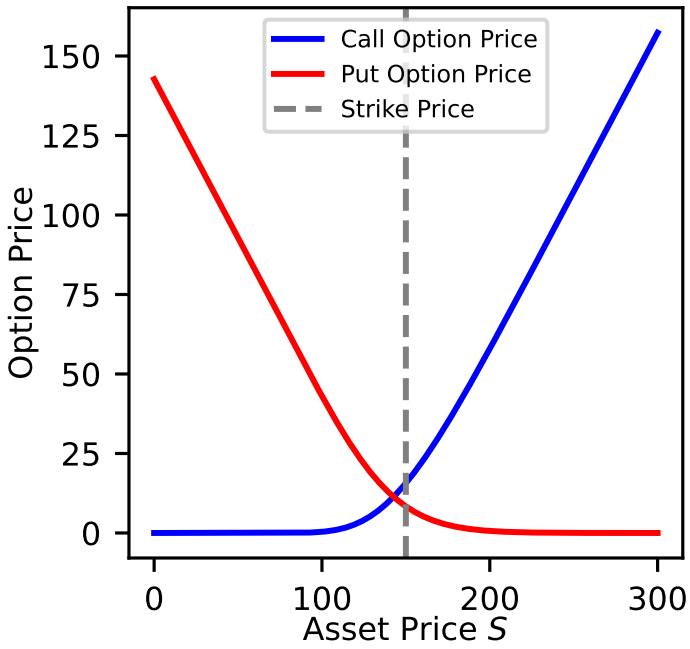


Figure 4.2: Black–Scholes prices for a single-asset European call and put option with $K = 150$, $r = 0.05$, $\sigma = 0.2$, and $T = 1$. The vertical dashed line indicates the strike price. In this scenario, we observe that the call option price is near zero for most of the domain, reflecting its deep out-of-the-money status. Meanwhile, the put option is in-the-money and shows higher values for a wide range of asset prices. Lower volatility reduces the curvature of both price curves, and the lower interest rate reduces the time-discounting effect on the strike price.

4.3 Multi-Asset (2D) Black-Scholes Equation

In the case of options depending on two underlying assets, the Black-Scholes equation extends to two spatial dimensions, introducing additional computational challenges. The general form of the two-dimensional Black-Scholes equation for an option price $u(S_1, S_2, t)$ is:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma_1^2 S_1^2 \frac{\partial^2 u}{\partial S_1^2} + \frac{1}{2}\sigma_2^2 S_2^2 \frac{\partial^2 u}{\partial S_2^2} + \rho\sigma_1\sigma_2 S_1 S_2 \frac{\partial^2 u}{\partial S_1 \partial S_2} + rS_1 \frac{\partial u}{\partial S_1} + rS_2 \frac{\partial u}{\partial S_2} - ru = 0, \tag{4.40}$$

where S_1 and S_2 are the prices of the two underlying assets, σ_1 and σ_2 are their volatilities, ρ is the correlation between them, and r is the risk-free interest rate.

4.3.1 Analytical Solutions for Two-Asset Options

For European options depending on two correlated assets S_1 and S_2 we present three fundamental cases:

Exchange Option (Margrabe's Formula [127])

For payoff $u(S_1, S_2, T) = \max(S_1(T) - S_2(T), 0)$, the solution is:

$$u(S_1, S_2, t) = S_1(t)N(d_1) - S_2(t)N(d_2) \tag{4.41}$$

where:

$$d_1 = \frac{\ln(S_1(t)/S_2(t)) + \frac{1}{2}\sigma^2(T-t)}{\sigma\sqrt{T-t}} \tag{4.42}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{4.43}$$

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2 - 2\rho\sigma_1\sigma_2} \tag{4.44}$$

Spread Option Approximation

For payoff $\max(S_1(T) - S_2(T) - K, 0)$, Kirk's approximation [128] which provides a practical closed-form estimator for European spread options when the strike price is small but non-zero gives:

$$u(S_1, S_2, t) \approx e^{-r(T-t)}\left[\left(S_1(t) + Ke^{-r(T-t)}\right)N(d_1) - S_2(t)N(d_2)\right] \tag{4.45}$$

where:

$$d_1 = \frac{\ln\left(\frac{S_1(t)+Ke^{-r(T-t)}}{S_2(t)}\right) + \frac{1}{2}\sigma^2(T-t)}{\sigma\sqrt{T-t}} \tag{4.46}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{4.47}$$

$$\sigma \approx \sqrt{\sigma_1^2\left(\frac{S_1}{S_1 + Ke^{-r(T-t)}}\right)^2 + \sigma_2^2 - 2\rho\sigma_1\sigma_2\frac{S_1}{S_1 + Ke^{-r(T-t)}}} \tag{4.48}$$

Quanto Option

A quanto option [129] (short for "quantity adjusting option") is a type of derivative where the underlying asset is denominated in one currency, but the payout is made in another currency at a fixed exchange rate. These options are commonly used in foreign exchange and equity markets to hedge currency risk in international portfolios or to gain exposure to foreign assets without taking on exchange rate volatility. For a foreign asset S_1 with payoff in domestic currency S_2, with correlation ρ:

$$u(S_1, S_2, t) = S_2(t)e^{-r_d(T-t)}\left[F(t,T)N(d_1) - KN(d_2)\right] \tag{4.49}$$

where:

$$F(t,T) = S_1(t)e^{(r_f - \rho\sigma_1\sigma_S)(T-t)} \tag{4.50}$$

$$d_1 = \frac{\ln(F(t,T)/K) + \frac{1}{2}\sigma_1^2(T-t)}{\sigma_1\sqrt{T-t}} \tag{4.51}$$

$$d_2 = d_1 - \sigma_1\sqrt{T-t} \tag{4.52}$$

Key Observations

- The exchange option solution is exact and correlation-dependent
- Spread options require approximations (Kirk, Bjerksund-Stensland)
- Quanto corrections account for cross-currency effects
- All solutions reduce to standard Black-Scholes when $S_2 \to 0$

Solving this equation numerically requires discretization methods such as finite difference schemes.

4.3.2 Explicit Finite Difference Scheme

The explicit finite difference method discretizes the derivatives in both spatial dimensions and time. Defining a grid where $u_{i,j,k}$ represents the option price at $S_1 = S_{1,i}$, $S_2 = S_{2,j}$, and $t = t_k$, we approximate the derivatives using finite differences:

$$\frac{\partial u}{\partial t} \approx \frac{u_{i,j,k+1} - u_{i,j,k}}{\Delta t}, \tag{4.53}$$

$$\frac{\partial u}{\partial S_1} \approx \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2\Delta S_1}, \tag{4.54}$$

$$\frac{\partial^2 u}{\partial S_1^2} \approx \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{(\Delta S_1)^2}, \tag{4.55}$$

$$\frac{\partial u}{\partial S_2} \approx \frac{u_{i,j+1,k} - u_{i,j-1,k}}{2\Delta S_2}, \tag{4.56}$$

$$\frac{\partial^2 u}{\partial S_2^2} \approx \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{(\Delta S_2)^2}, \tag{4.57}$$

$$\frac{\partial^2 u}{\partial S_1 \partial S_2} \approx \frac{u_{i+1,j+1,k} - u_{i+1,j-1,k} - u_{i-1,j+1,k} + u_{i-1,j-1,k}}{4\Delta S_1 \Delta S_2}. \tag{4.58}$$

Substituting these into the Black-Scholes equation and solving for $u_{i,j,k+1}$ gives:

$$u_{i,j,k+1} = u_{i,j,k} + \Delta t\big(Au_{i-1,j,k} + Bu_{i,j,k} + Cu_{i+1,j,k} + Du_{i,j-1,k} + Eu_{i,j+1,k} + Fu_{i-1,j-1,k} + Gu_{i+1,j+1,k}\big), \tag{4.59}$$

where the coefficients A, B, C, D, E, F, G depend on $r, \sigma_1, \sigma_2, \rho, S_1, S_2$. The explicit scheme is simple but requires a very small time step for stability.

4.3.3 Implicit Finite Difference Scheme

The implicit method improves stability by solving for the option values implicitly at each time step. Using a backward time difference:

$$\frac{u_{i,j,k} - u_{i,j,k-1}}{\Delta t} + \mathcal{L}u_{i,j,k} = 0, \tag{4.60}$$

where $\mathcal{L}$ represents the spatial differential operator in the Black-Scholes equation, we obtain a system of algebraic equations:

$$Au_{i-1,j,k}+Bu_{i,j,k}+Cu_{i+1,j,k}+Du_{i,j-1,k}+Eu_{i,j+1,k}+Fu_{i-1,j-1,k}+Gu_{i+1,j+1,k} = u_{i,j,k-1}. \tag{4.61}$$

This results in a large sparse linear system, which is typically solved using iterative methods like Successive Over-Relaxation (SOR) or Alternating Direction Implicit (ADI) methods.

4.3.4 Example 1: Two-Asset Call Option

We consider here the pricing of a maximum call option on two assets as our test problem. The computational domain is set to $[50, 150] \times [50, 150]$ with $t \in [0, 1]$. The option parameters are as follows: the exercise price is set to $K = 100$, the risk-free interest rate to $r = 0.05$, the volatility of the first asset to $\sigma_1 = 0.2$, the volatility of the second asset to $\sigma_2 = 0.4$, the correlation between assets to $\rho = 0.5$, and the participation rate $q = 0.5$. The option has a maturity of $T = 1$ year.

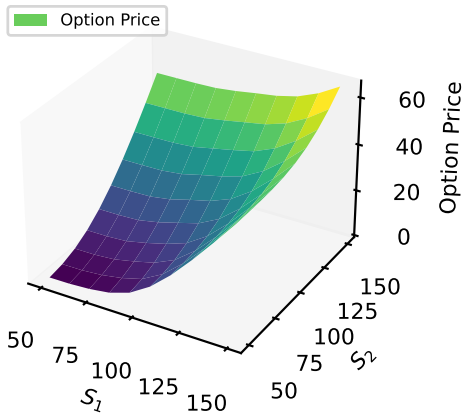
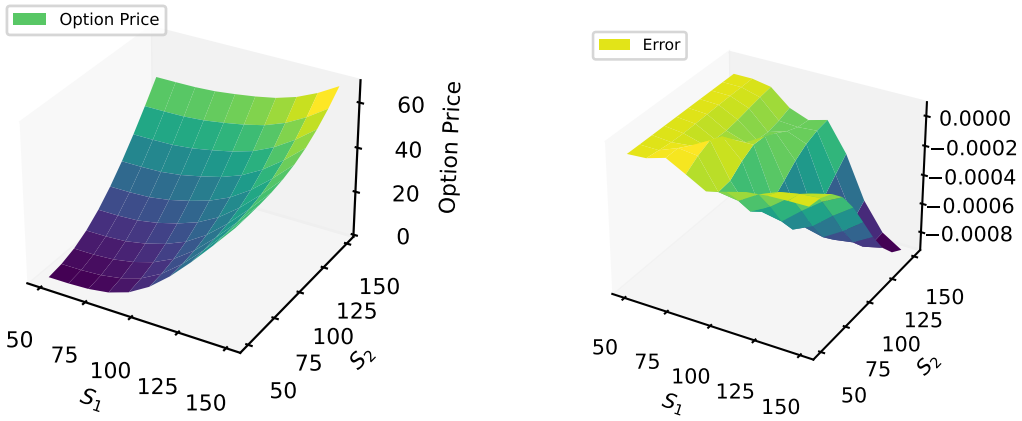


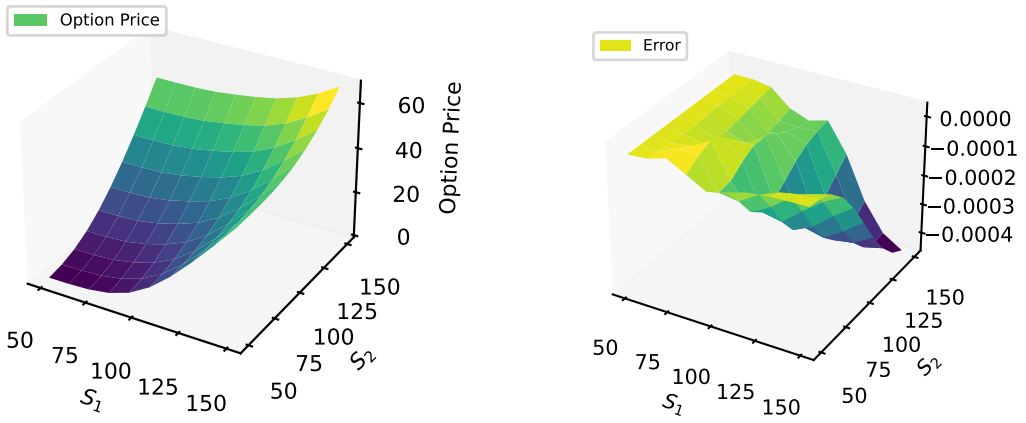
Figure 4.3: Closed form solution for the two-asset call option.



(a) Explicit finite difference solution for the two-asset call option.

(b) Error of explicit finite difference solution vs closed form solution.

Figure 4.4: Comparison of explicit finite difference solutions and the closed form solutions to the two-asset call option.



(a) Implicit finite difference solution for the two-asset call option.

(b) Error of implicit finite difference solution vs closed form solution.

Figure 4.5: Comparison of implicit finite difference solutions and closed form solutions for the two-asset call option.

A key observation is the significantly smaller error magnitude in the implicit Euler method, with errors ranging to approximately -0.0004, compared to the explicit Euler method where errors reach about -0.0008. Both methods exhibit a consistent negative error across the domain, suggesting a potential underestimation of the true solution.

4.3.5 Example 2: Two-Asset Cash-or-Nothing Options

We consider here the two-asset cash-or-nothing options as test problems. We choose the computational domain $[0, 300] \times [0, 300]$ for $t \in [0, 1]$ and (4.62) is used to set the Dirichlet boundary condition. The strike price is set to $K_1 = 100$ and $K_2 = 100$, the risk-free interest rate to $r = 1.0$, the volatility of the first asset to $\sigma_1 = 0.3$, the volatility of the second asset to $\sigma_2 = 0.3$, the correlation to $\rho = 0.5$ and the maturity is $T = 1$. The closed-form solution [130] for a cash-or-nothing option on two assets is given by:

$$u(S_1, S_2, \tau) = c\, e^{-r\tau}\, B(d_1, d_2; \rho), \tag{4.62}$$

$$d_1 = \frac{\ln\left(\frac{S_1}{K_1}\right) + (r - 0.5\sigma_1^2)\tau}{\sigma_1\sqrt{\tau}}, \tag{4.63}$$

$$d_2 = \frac{\ln\left(\frac{S_2}{K_2}\right) + (r - 0.5\sigma_2^2)\tau}{\sigma_2\sqrt{\tau}}, \tag{4.64}$$

where c is the cash payout, ρ is the correlation between the assets, and the bivariate cumulative normal distribution function [131] is:

$$B(d_1, d_2; \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{d_1} \int_{-\infty}^{d_2} \exp\left(-\frac{\xi_1^2 - 2\rho\xi_1\xi_2 + \xi_2^2}{2(1-\rho^2)}\right) d\xi_2 d\xi_1. \tag{4.65}$$

We consider the at-the-money case (which produces meaningful dynamics around the most sensitive region of the option price) where the strike prices $K_1 = K_2 = 100$, resulting in rich dynamics near the exercise region. Although the strike prices K_1 and K_2 do not appear directly in the differential equation, they are encoded in the terminal condition:

$$u(S_1, S_2, T) = \begin{cases} c, & \text{if } S_1 \geq K_1 \text{ and } S_2 \geq K_2, \\ 0, & \text{otherwise.} \end{cases} \tag{4.66}$$

Here, c denotes the fixed cash payout at maturity, received if both assets are above their respective strike prices.

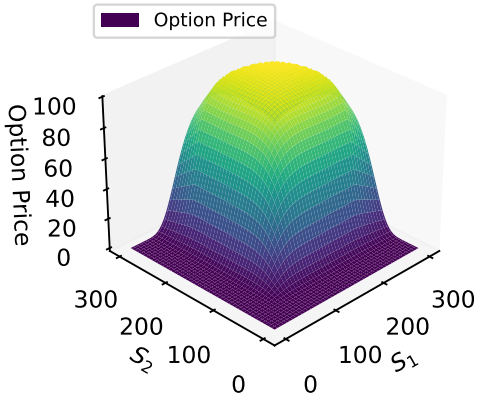
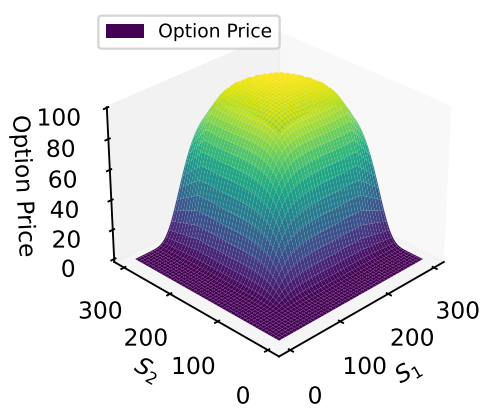
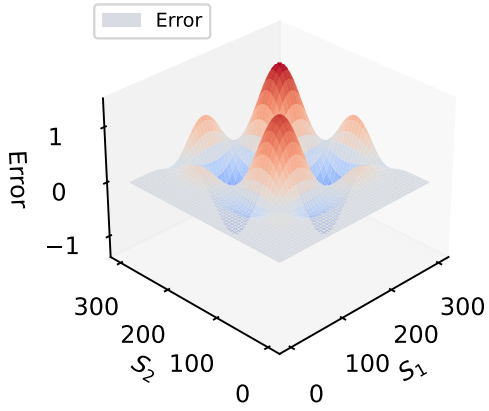


Figure 4.6: Closed form solution of the cash-or-nothing option.

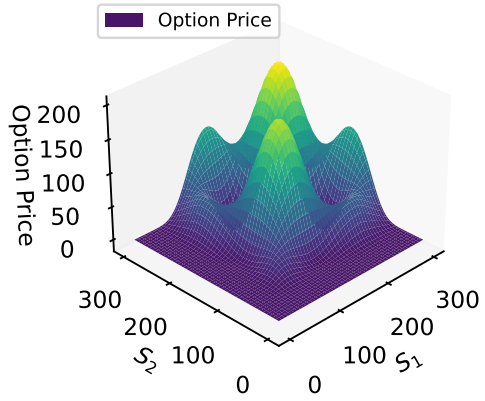


(a) Implicit finite difference solution for the two-asset cash-or-nothing option.

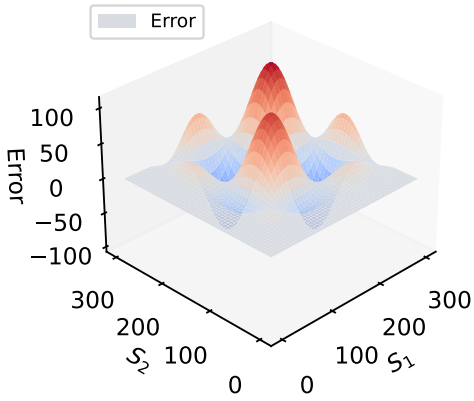


(b) Error of implicit finite difference solution vs closed form solution.

Figure 4.7: Comparison of implicit finite difference solutions and closed form solutions for the two-asset cash-or-nothing option.



(a) Explicit finite difference solution for the two-asset cash-or-nothing option.



(b) Error of explicit finite difference solution vs closed form solution.

Figure 4.8: Comparison of explicit finite difference solutions and closed form solutions for the two-asset cash-or-nothing option.

Chapter 5

PINN as coarse propagator in Parareal for Single Asset Black-Scholes PDE

This chapter is based on the author's previously published work [132], in which a Physics-Informed Neural Network (PINN) is employed as a coarse propagator within the Parareal algorithm. The study demonstrates how integrating PINNs with Parareal can efficiently accelerate the solution of the Black-Scholes equation.

5.1 Fine propagator PDE model and numerical resolutions

The fine propagator in this chapter is a numerical solver for the single-asset Black–Scholes PDE (4.1), using a finite-difference spatial discretization and a stable implicit time integrator (Crank–Nicolson / implicit Euler, as used in the numerical experiments). We report the key resolution parameters used in the experiments, namely the spatial grid size N_S, the fine time step δt, the Parareal coarse slice size ΔT, and the number of Parareal slices N.

5.2 PINNs for Solving BSE

The PINN model for the Black-Scholes equation is designed to learn the option price as a function of the underlying asset price and time. The architecture consists of an input layer for the asset price S and time t, followed by several hidden layers with activation functions (e.g., ReLU, tanh). The output layer represents the option price $u(S, t)$. The loss function combines the PDE residual, boundary conditions, and initial conditions, allowing the network to learn the solution iteratively. The PINN approach is particularly advantageous for high-dimensional problems, where traditional numerical methods may struggle with the curse of dimensionality. The PINN model is trained using a combination of the PDE residual, boundary conditions, and initial conditions as the loss function. The PDE residual is computed by substituting the neural network output into

the Black-Scholes equation, while the boundary conditions are enforced by evaluating the network at the boundaries of the domain.

5.3 Architecture of PINN-Propagator (PINN-P)

The PINN we use as coarse propagator gets a time slice $[t_{\text{start}}, t_{\text{end}}] \subset [0, T]$, the asset price u at t_{start} and stock values S, and outputs the predicted state of the asset price $\tilde{u}$ at t_{end}. To train it, we define three sets of collocation points in time and stock price: $(S_i, t_i), i = 1, \dots N_f$ in the interior of the space-time domain for evaluating the residual $f(u)$ of the Black-Scholes equation, $(S_i, t_i), i = 1, \dots N_b$ collocation points on the boundary to evaluate, and $S_i, i = 1, \dots N_{\text{exp}}$ for the final state conditions. For our setup, we randomly generate $N_f = 100,000$ collocation points within the domain $[0, 5000] \times [0, 1]$, $N_b = 10,000$ collocation points at the boundary $[0, 1]$ and $N_{\text{exp}} = 10,000$ collocation points to sample the expiration condition over $[0, 5000]$.

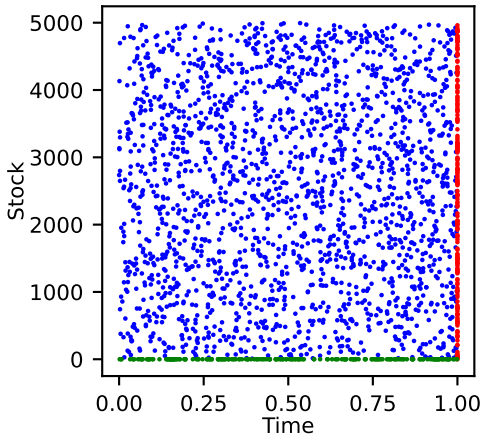


Figure 5.1: Subset of the randomly generated collocation nodes. The solution is forced to satisfy the PDE at the inner nodes by minimizing the PDE residual, to satisfy the boundary condition at the green nodes via the boundary loss and the expiration condition at the red nodes via the expiration loss.

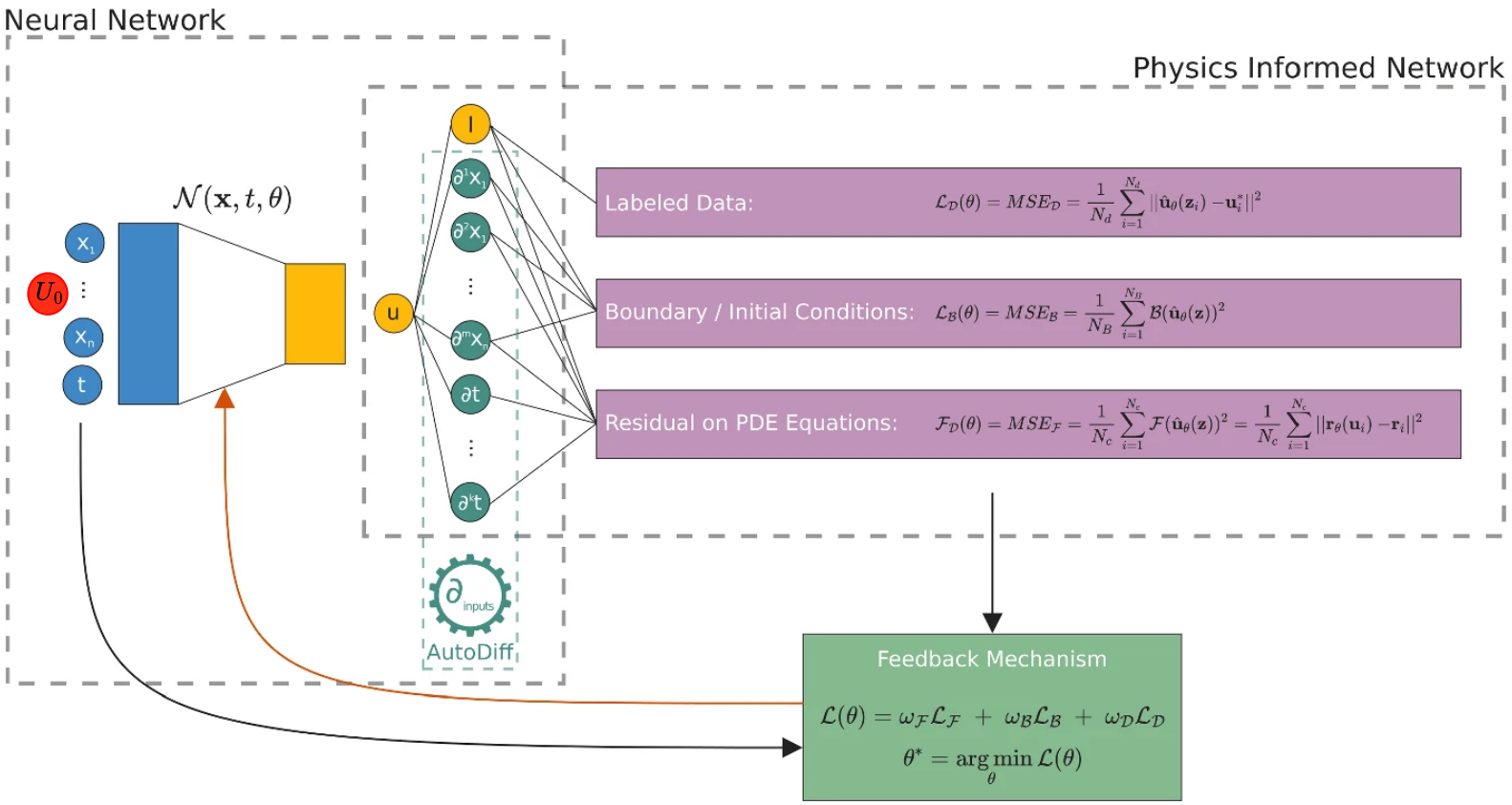


Figure 5.2: Architecture of the PINN model.

The PINN is trained by minimizing a composite loss function that encodes the PDE and the problems boundary/initial conditions. At a high level, this loss $\mathcal{L}$ is a sum of three main contributions: (1) the PDE residual loss, which measures how well the network's output satisfies the Black–Scholes equation (4.1) across the domain; (2) the boundary condition loss, enforcing the network's solution to meet boundary conditions (such as $u(0,t) = 0$ and the asymptotic behavior as $S \to$ large); and (3) the initial/terminal condition loss, enforcing the known payoff at expiration $t = T$ (for example, $u(S,T) = \max(S-K,0)$ for a call option). Conceptually each term is a mean-squared error measuring the discrepancy between the PINN's output and what the true solution should satisfy on that aspect. For instance, the PDE residual term evaluates the left-hand side of (4.1) (which should equal zero for the true solution) using the network's prediction $u_\theta(S,t)$ and its derivatives; any deviation from zero contributes to the loss. These derivatives $\partial_t u$, $\partial_S u$, $\partial_{SS} u$ needed for the residual are obtained efficiently via automatic differentiation through the network – a benefit of the PINN approach. Similarly, the boundary loss term might measure errors like $|u_\theta(0,t) - 0|^2$ at $S = 0$ and the mismatch from the far-field condition at S = large, while the terminal loss term measures $|u_\theta(S,T) - \max(S-K,0)|^2$ at $t = T$. By summing these contributions, the total loss function penalizes any violation of the PDE or constraints, guiding the training process to find a function that simultaneously minimizes all such violations. The loss functions to be minimized are given by

$$\text{MSE}_{\text{total}} = \text{MSE}_f + \text{MSE}_{\text{exp}} + \text{MSE}_b, \tag{5.1}$$

consisting of a term to minimize the PDE residual $f(u)$

$$\text{MSE}_{\text{f}} = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(\tilde{u}(t_i, S_i))|^2, \tag{5.2}$$

the boundary loss term

$$\text{MSE}_{\text{b}} = \frac{1}{N_b} \sum_{i=1}^{N_b} \left|\tilde{u}(t_i, S_i) - u(t_i, S_i)\right|^2, \tag{5.3}$$

and the loss at expiration

$$\text{MSE}_{\text{exp}} = \frac{1}{N_{\text{exp}}} \sum_{i=1}^{N_{\text{exp}}} \left|\tilde{u}(T, S_i) - \max(S_i - K, 0)\right|^2, \tag{5.4}$$

The derivatives that are required to compute the PDE loss are calculated by automatic differentiation [133]. We compute the PDE residual (5.2) over the points inside the domain, the boundary condition loss (5.3) over the spatial boundary and the expiration loss (5.4) over the end points. The sum of the three forms the total loss function.

Figure 5.1 shows a subset of the generated collocation points to illustrate the approach. The neural network consists of 10 fully connected layers with 50 neurons in

each and was implemented using Pytorch [134]. Figure 3.1 shows the principle of a PINN but for a smaller network for the sake of readability. Every linear layer, excluding the output layer, is followed by the ReLU activation function. The weights for the neural network are initialized using Kaiming [135]. We focus here on a proof-of-concept and have not undertaken a systematic effort to optimize the network architecture but this would be an interesting avenue for future research.

We used the Adam optimizer [97] with a learning rate of 10^{-2} for the initial round of training for 5000 epochs, followed by a second round of training with a learning rate of 10^{-3} for 800 epochs. The training data (collocation points) was shuffled during every epoch to prevent the model from improving predictions based on data order rather than the underlying patterns in the data. Table 5.1 shows the behavior of the three loss function terms. The total training time for this model was around 30 minutes.

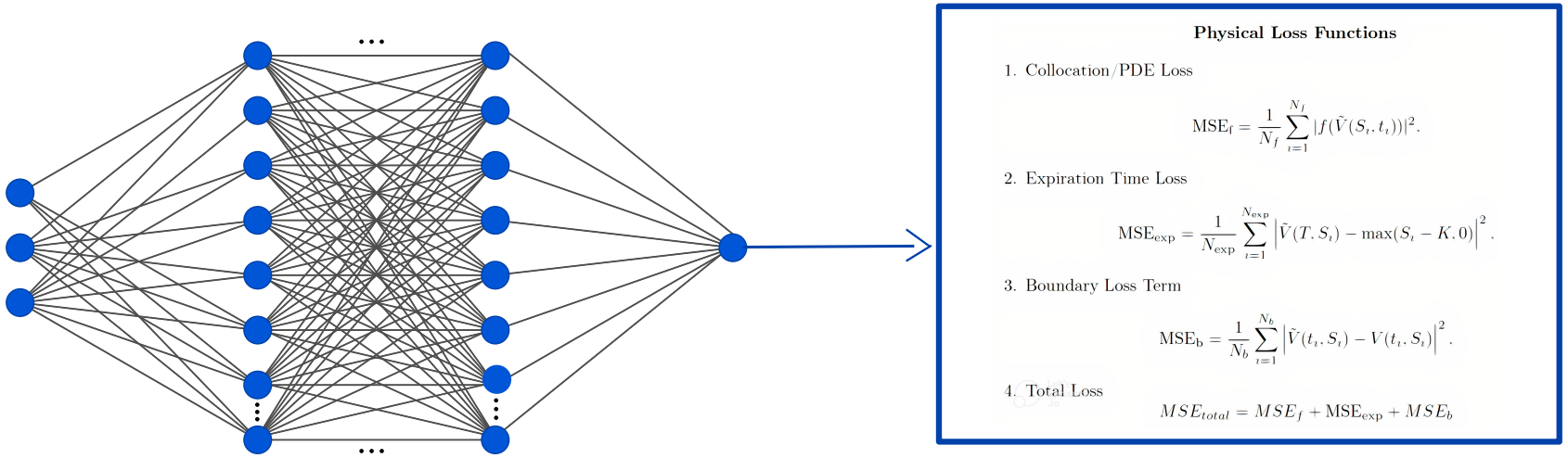


Figure 5.3: Structure of the PINN. The network takes the time t_{start}, t_{end}, asset values V and stock values S as input and returns the predicted asset values $\tilde{V}$ at t_{end}. The loss function encodes the PDE, the expiration condition, and the boundary conditions.

5.4 Loss function and convergence

We next report the training dynamics of the PINN coarse propagator and relate them to the Parareal convergence and runtime results shown later in this chapter (e.g., Figures 5.7 and 5.8).

Epoch	Expiration	Boundary	Residual
0	9.21×10^{2}	9.21×10^{2}	7.33×10^{3}
2000	5.58×10^{-1}	3.45×10^{-2}	2.50×10^{-2}
4000	4.11×10^{-2}	2.34×10^{-2}	5.00×10^{-3}
5000	5.92×10^{-1}	1.34×10^{-2}	4.22×10^{-3}
5300	4.19×10^{-2}	3.22×10^{-3}	1.94×10^{-4}
5500	6.46×10^{-4}	1.96×10^{-4}	5.73×10^{-5}
5800	2.92×10^{-5}	1.14×10^{-5}	3.19×10^{-4}

Table 5.1: Evolution of the loss function during network training for the PINN model. The table summarizes the mean squared error (MSE) of the three loss components over training epochs: the terminal (expiration) condition, boundary condition, and PDE residual. Training was initially performed for 5000 epochs with a learning rate of 10^{-2}, followed by an additional 800 epochs with a reduced rate of 10^{-3}. The residual loss, evaluated at $N_f = 100{,}000$ interior collocation points, dominates the overall loss. In contrast, the boundary and expiration terms are evaluated at $N_b = N_{\text{exp}} = 10{,}000$ points each. The model achieves convergence with the residual term stabilizing around 10^{-4}, the boundary condition around 10^{-5}, and the expiration condition around 10^{-2}.

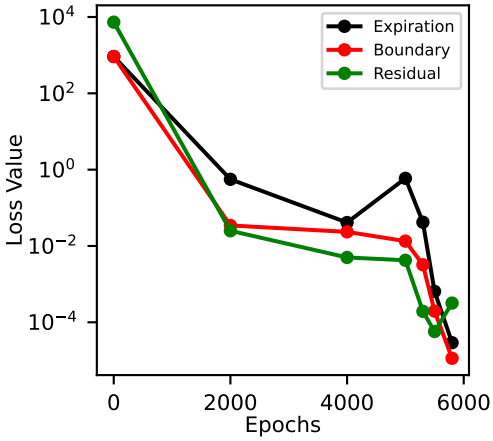


Figure 5.4: Visualization of the loss function components during training of the PINN model. The plot shows the evolution of the mean squared error (MSE) for the expiration condition, boundary condition, and PDE residual over selected training epochs, corresponding to the values reported in Table 5.1. The residual loss dominates early training and gradually stabilizes around 10^{-4}, while the boundary and expiration losses decrease more rapidly.

5.5 Hardware and Software

All numerical experiments were carried out on a workstation running openSUSE Leap 15.4, a stable Linux distribution suitable for scientific computing. The system is equipped with a 12th Gen Intel Core i9-12900K processor, featuring 24 threads across performance and efficiency cores, with a base clock speed of 3.2,GHz and a maximum turbo frequency of 5.2,GHz. The machine has 62.6 GiB of RAM, enabling efficient handling of large collocation point sets and deep learning model parameters during training. For GPU-accelerated computations, the system uses an NVIDIA GeForce RTX 3060 (12 GB VRAM), accessible via PCIe and supporting CUDA-enabled operations. This GPU was particularly leveraged for accelerating forward and backward passes during PINN/PINO training and for parallel evaluation of the residual terms at collocation points. The software environment was built on `Python 3.10`, with core dependencies including:

- PyTorch 1.13.1+cu117 - for implementing and training neural networks (PINNs and FNOs), taking advantage of GPU acceleration via CUDA 11.7.
- mpi4py 3.1.4 - to enable distributed parallel computation across multiple processes, primarily used for the Parareal coarse-fine propagator architecture.
- Numba 0.55.1 - to JIT-compile performance-critical numerical kernels for both CPU and GPU backends, especially in custom PDE solvers and evaluation of derivative terms.

Additional Python libraries used include NumPy and SciPy for numerical routines, Matplotlib for plotting and visualization, and SymPy for symbolic differentiation and PDE preprocessing. All experiments were containerized using Conda environments to ensure reproducibility, and training was monitored via custom logging scripts and checkpointing mechanisms. This setup allowed for efficient experimentation with various neural operator architectures, hybrid CPU-GPU workflows, and scalable parallel-in-time schemes such as the Parareal algorithm.

5.6 Parareal convergence

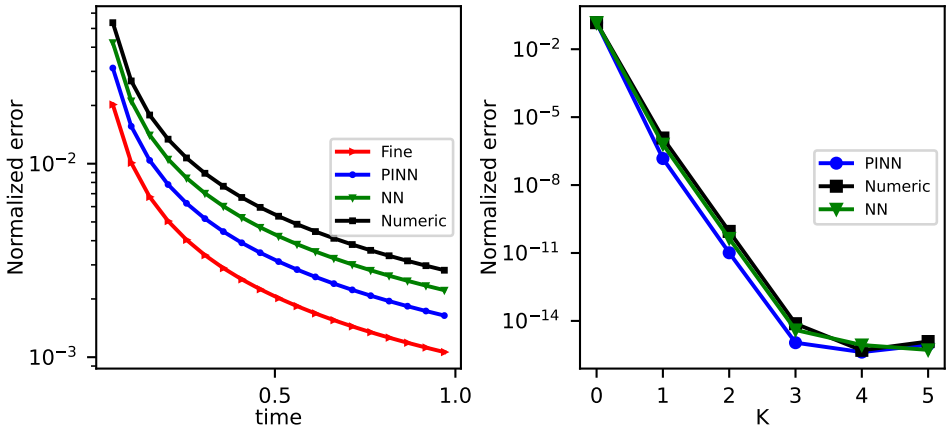


Figure 5.5: Normalized ℓ_2-error over time of coarse and fine propagator against the analytical solution (left). Normalized ℓ_2-error against the serial fine solution versus number of iterations for three different variants of Parareal (right). The black line (squares) is Parareal with a numerical coarse propagator, the green line (diamonds) is Parareal with a neural network as coarse propagator that is trained only on data while the blue line (circles) is Parareal with a PINN as coarse propagator that also uses the terms of the differential equation in the loss function. Parareal uses $P = 16$ time slices in all cases.

As expected, the serial fine solver achieves the highest accuracy, with a final normalized ℓ_2 error on the order of 10^{-3}. The purely numerical coarse propagator, based on a low-resolution finite-difference discretization, performs significantly worse in terms of accuracy. The PINN-based coarse propagator, which encodes the Black-Scholes differential operator, boundary conditions, and expiration condition in its loss function (as in equation (5.1)), achieves better accuracy than the numerical coarse propagator, although it still does not fully match the fine solution.
To underscore the importance of incorporating physics-based constraints into the learning process, we also evaluate a neural network trained solely on trajectory data produced by the fine solver, without including any PDE-based loss terms. This purely data-driven NN performs slightly better than the numerical coarse method but remains less accurate than the PINN. Importantly, the PINN does not require labeled trajectories for train-

ing—its accuracy is obtained purely through enforcing the differential constraints and boundary information, without any explicit supervision from the fine solver.
Figure 5.5 (right) shows the convergence behavior of the Parareal algorithm in terms of normalized ℓ_2 error with respect to the number of iterations K, using the three coarse propagators: numerical, NN-based, and PINN-based. In all cases, Parareal converges rapidly.
The use of PINN and NN coarse models provides slightly better initial approximations compared to the numerical propagator, but the difference in final convergence is marginal. After just a single iteration ($K = 1$), the Parareal solution already surpasses the fine method's spatial discretization error. By $K = 3$ iterations, the Parareal solution essentially matches the fine solution to machine precision. In the remainder of this work, performance metrics such as runtime and speedup are reported. This choice corresponds to the convergence threshold beyond which further iterations provide negligible improvement. Notably, at this point, the influence of the K/P overhead term becomes less critical. Consequently, minimizing the runtime of the coarse propagator has a direct and significant impact on overall speedup. However, once K reaches 3, the marginal benefit of further improving the coarse propagator diminishes, since the fine-level accuracy is already recovered.

5.7 Generalization of PINN-P

Figure 5.6 illustrates the generalization behavior of the Parareal algorithm when the coarse propagator is a physics-informed neural network (PINN) trained on a specific set of parameter values (e.g., volatility σ and interest rate r in the Black-Scholes equation). The figure shows how many Parareal iterations are required for convergence when the algorithm is applied to problems with parameters that differ from those used during the PINN's training. As expected, when the input parameters deviate from the training values, the PINN coarse propagator becomes less accurate, since it no longer approximates the solution operator with the same fidelity. This reduction in coarse accuracy can impact the convergence rate of Parareal. However, a key advantage of the Parareal framework is that it guarantees convergence to the correct solution regardless of the coarse model, as long as the fine propagator remains accurate and is used in the correction step. Therefore, even if the PINN coarse model introduces inaccuracies, the Parareal iterations will correct them.

The results in Figure 5.6 demonstrate that the combination of Parareal with a PINN coarse propagator exhibits robust generalization. For moderate parameter shifts, the algorithm still converges within the same number of iterations as in the in-distribution setting. Even when the parameters are perturbed by an order of magnitude (e.g., 10× larger than the training set), convergence is only marginally affected—typically requiring just one additional iteration to reach fine-level accuracy. While an increase in the number of iterations does reduce the theoretical speedup achievable by Parareal, the observed degradation is minor. The performance and speedup metrics reported in subsequent sections (evaluated at $K = 3$ iterations) remain representative even under significant

variations in model parameters. This suggests that the PINN-based Parareal approach is not only computationally efficient but also robust to changes in PDE parameters, making it a practical tool for problems where parameter ranges are uncertain or variable.

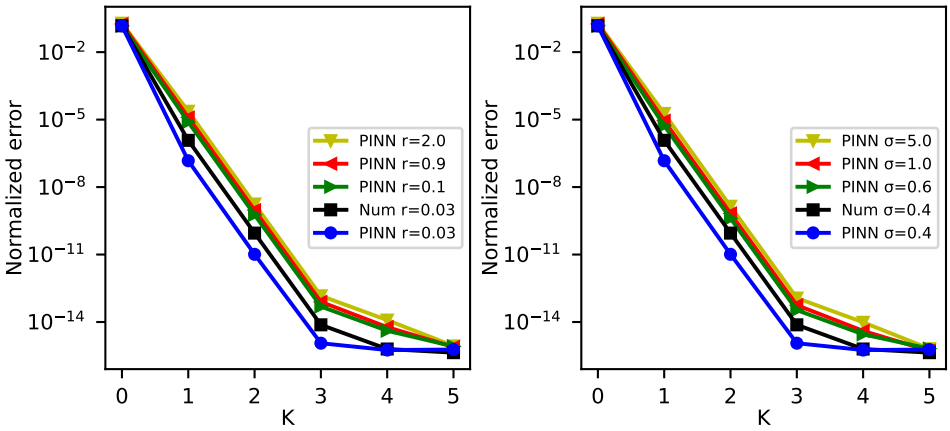


Figure 5.6: Convergence of Parareal for different interest rates r (left) and volatilities σ (right). In all cases, the coarse propagator is the PINN trained for values of $r = 0.03$ and $\sigma = 0.4$. Even for parameter values more than ten times larger than the ones for which the PINN was trained, Parareal requires only one additional iteration to converge to within machine precision of the fine integrator.

5.8 Parareal Runtimes and Speedup

All runtimes are reported in milliseconds and measured using the Linux `time` command, capturing the full execution time including setup, computation, and data movement. Performance measurements are averaged over five runs, with standard deviations reported to account for variability. Table 5.2 compares the execution time of the coarse propagator $\mathcal{C}$ in Parareal using four different configurations: numerical and PINN-based propagators, executed on either CPU or GPU. The number of time slices (and thus parallel cores) is fixed at $P = 16$. Replacing the traditional numerical coarse propagator with a PINN model executed on the CPU reduces coarse propagation time by a factor of approximately 2.4. When the PINN is executed on a GPU, this speedup increases to a factor of 2.9. Notably, using the GPU for the numerical coarse method yields negligible improvement, as the lower resolution and reduced computational intensity of the numerical method underutilize the available parallelism and memory bandwidth of the GPU. The PINN-based propagator, due to its highly parallelizable architecture and compact runtime, effectively minimizes the serial bottleneck in the Parareal algorithm. This reduction in coarse propagation time directly enhances the potential speedup and efficiency of the overall method, as demonstrated in the subsequent results.

Platform	Numerical (ms)	PINN (ms)	Speedup over CPU-Numerical
CPU	3.48 ± 0.056	1.47 ± 0.073	$2.4\times$
GPU	3.99 ± 0.651	1.21 ± 0.041	$2.9\times$

Table 5.2: Runtime c_c of the coarse propagator $\mathcal{C}$ in milliseconds, averaged over five runs.

Table 5.3 presents the total Parareal runtimes for different combinations of CPU/GPU assignments to the fine and coarse propagators. The fastest configuration is achieved when the numerical fine solver is run on the CPU and the PINN coarse propagator is executed on the GPU. This hybrid configuration leverages the computational strengths of each component: the CPU for the numerically intensive fine propagator and the GPU for the highly parallel neural network-based coarse model. Running both propagators on the CPU results in runtimes that are approximately three times longer than the hybrid setup. Interestingly, executing both on the GPU is also slower than the hybrid configuration by about a factor of two. This is likely due to memory contention and kernel dispatch overhead when both propagators compete for GPU resources. While the full-GPU configuration may outperform the hybrid approach at significantly higher resolutions, in the current experiments the fine solution already achieves an acceptable accuracy of $\mathcal{O}(10^{-3})$. These findings highlight the practical benefits of combining neural networks and numerical solvers within the Parareal framework. In particular, they demonstrate that neural operators such as PINNs not only offer computational efficiency but also enable better resource utilization on heterogeneous compute nodes with both CPUs and GPUs.

Fine Propagator	CPU-Coarse (ms)	GPU-Coarse (ms)
CPU-Fine	128.48 ± 0.715	41.24 ± 0.334
GPU-Fine	83.25 ± 0.356	87.45 ± 0.253

Table 5.3: Total Parareal runtimes in milliseconds for different CPU/GPU configurations, averaged over five runs.

Figure 5.7 further explores how the overall runtime of Parareal evolves with increasing number of processors/time slices P, when using either a PINN or numerical coarse propagator, executed on CPU (left) and GPU (right). The fine solver is kept on the CPU for all configurations in this comparison. In both CPU and GPU settings, runtimes decrease as the number of time slices increases. However, the PINN-based coarse propagator consistently outperforms its numerical counterpart across all values of P, confirming that the benefits of the PINN extend throughout the full range of parallel configurations.

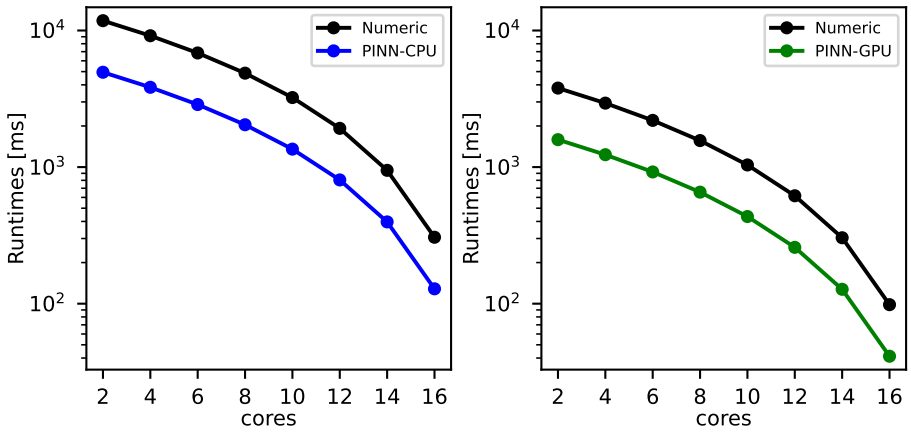


Figure 5.7: Parareal runtimes vs. number of processors P for CPU (left) and GPU (right) coarse propagators. Horizontal lines indicate serial fine propagator runtime.

Finally, Figure 5.8 summarizes the achieved **speedup** (left) and **parallel efficiency** (right) of Parareal relative to the serial fine propagator executed on a CPU. Results are shown for three coarse propagator types: numerical, PINN-CPU, and PINN-GPU. Replacing the numerical coarse model with a PINN significantly improves speedup, and running the PINN on a GPU amplifies this benefit. Specifically, Parareal achieves a speedup of $S(16) \approx 2$ using a CPU-based numerical coarse propagator. This increases to $S(16) \approx 3$ with a CPU-based PINN and to $S(16) \approx 4.5$ when the PINN is executed on the GPU—more than doubling the performance compared to the baseline. Parallel efficiency improves accordingly, from approximately 30% with the numerical coarse model to over 50% with the GPU-accelerated PINN.

While the performance gain is more pronounced at higher processor counts (where the serial portion of the algorithm is more critical), efficiency improvements are observed across all tested core counts from $P = 2$ to $P = 16$. In summary, these results demonstrate that replacing a traditional numerical coarse solver with a PINN, particularly when deployed on GPU hardware, significantly reduces the serial overhead in Parareal. This leads to higher speedups and improved parallel efficiency, making the approach highly scalable and efficient for solving time-dependent PDEs in parallel-in-time frameworks.

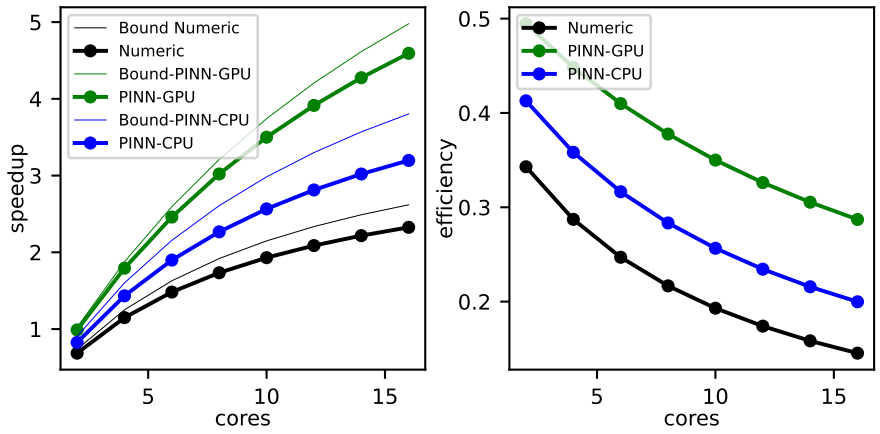


Figure 5.8: Speedup (left) and parallel efficiency (right) of Parareal vs. the serial numerical fine solver. The use of a PINN-GPU coarse propagator yields the highest performance.

5.9 Discussion

Parareal is a parallel-in-time algorithm that decomposes the time domain and iteratively combines a fast, serial coarse propagator with an expensive, fully parallel fine propagator. While the method allows for substantial parallelization in the time direction, its overall speedup is fundamentally limited by the serial execution of the coarse propagator, which must process each time slice sequentially to preserve causality. In traditional implementations, the coarse propagator is typically a reduced-fidelity version of the fine solver—constructed with lower resolution, reduced-order discretizations, or simplified physics.

In this work, we propose and evaluate an alternative approach: using a physics-informed neural network (PINN) as the coarse propagator within the Parareal framework. Our results show that the PINN-based propagator offers several advantages. Although it is only slightly more accurate than a standard low-resolution numerical coarse solver, it is up to three times faster in execution when run on a GPU. This significantly reduces the serial bottleneck in Parareal without compromising convergence. In all tested configurations, Parareal with a PINN coarse model converged to the fine solution within the same number of iterations as with a traditional numerical coarse solver. We demonstrate that optimal performance is achieved by exploiting hardware heterogeneity. Specifically, running the fine propagator on the CPU and the PINN coarse propagator on the GPU yields the fastest overall runtime. This configuration leverages the numerical solver's suitability for CPU execution and the PINN's inherent compatibility with GPU-based parallelism. In contrast, running both propagators on the CPU is considerably slower, and even running both on the GPU can underperform due to contention and limited benefit for the relatively low-computation-intensity fine solver.

Importantly, we observe that the gains from using a PINN coarse model translate directly into improved speedup and parallel efficiency. For example, at $P = 16$ time slices, speedup increases from approximately $2\times$ for a numerical coarse solver on CPU, to $3\times$

with a PINN on CPU, and up to $4.5\times$ when the PINN is run on GPU. Parallel efficiency improves correspondingly, reaching over 60% with the PINN-GPU configuration. These improvements are consistent across a range of processor counts, and demonstrate that neural coarse propagators can enhance both scalability and performance.

Furthermore, we show that PINN-based coarse models generalize robustly across a range of PDE parameters, even when those parameters differ significantly from the training regime. The Parareal algorithm remains stable and accurate due to the corrective nature of the fine propagator. Even when the PINN was applied to out-of-distribution parameter values (e.g., volatility and interest rate scaled by a factor of ten), convergence was still achieved with only one additional iteration—ensuring that performance gains are preserved in practice. Overall, our findings suggest that incorporating neural surrogate models into parallel-in-time frameworks offers a promising strategy to overcome the causality-imposed bottlenecks that limit parallel efficiency. PINNs, in particular, are attractive for this role because they can be trained without labeled trajectory data and efficiently executed on modern accelerators. Beyond raw performance gains, this hybrid architecture of numerical and neural methods opens new possibilities for exploiting heterogeneous computing environments—including future systems featuring neural network accelerators or domain-specific hardware.

In summary, physics-informed neural networks are a viable and effective choice for constructing coarse propagators in Parareal. They not only maintain convergence properties but also deliver significant runtime reductions and improved parallel scaling. This hybrid paradigm—combining classical solvers with neural networks—may offer a broader path forward for time-parallel algorithms in scientific computing.

Chapter 6

FNO as a Coarse Propagator for the Multi-Asset Black-Scholes PDE

6.1 Fine/Coarse PDE model and resolutions

Fine model: multi-asset Black–Scholes PDE discretized with finite differences in space and implicit Euler in time. Coarse model: the learned PINO/FNO-based propagator used inside Parareal, operating on the same state representation but at reduced effective cost; unless otherwise stated, the coarse model is evaluated once per coarse Parareal time slice.
Resolution parameters reported in this chapter: number of time slices N in Parareal, fine time step δt, coarse time step ΔT, spatial grid sizes per asset, and (when using distributed memory) the number of spatial processes P_{space} and time processes P_{time}.

6.2 Spatial Parallelization

In addition to temporal parallelization via Parareal, This chapter is based on the author's previously published work [11] where we implement parallelization in the spatial domain to further reduce runtime and improve scalability. For this purpose, we employ the Dask Python framework [136], which offers high-level, flexible APIs for distributed and parallel computing across both shared and distributed memory systems. Under the hood, Dask integrates with mpi4py for inter-process communication, allowing us to scale computations across multiple CPU cores or nodes. While shared memory parallelism (e.g., using threads) might offer reduced communication overheads, Python's Global Interpreter Lock (GIL) presents a significant limitation in this context, especially when using native Python constructs. For this reason, we adopt a distributed memory approach to parallelization in both space and time. Spatial parallelization is applied within each time step of the implicit Euler integrator, which involves solving a linear system resulting from the finite difference discretization of the spatial derivatives. The spatial domain is decomposed and distributed across P_{space} processes, each responsible for a subdomain. At each time step, these processes collaboratively solve the spatial system using a par-

allel implementation of the conjugate gradient (CG) method. Here, "CG" denotes the conjugate gradient method; we use c_{CG} (not p_p) for the per-iteration computational cost when a symbolic cost model is needed (avoiding confusion with $c_{\mathcal{G}}$, the cost of the coarse propagator). To enable efficient computation, we implement the CG algorithm using Dask arrays, which support block-wise operations and parallel evaluation. This setup allows for parallel computation of matrix-vector and vector-vector products—the most computationally intensive components of the CG algorithm. Data exchanges between subdomains are handled via Dask's task scheduling and mpi4py-based communication, ensuring correct treatment of boundary conditions and convergence across the full spatial domain. The theoretical speedup from spatial parallelization alone is bounded by:

$$S(P_{\text{space}}) \leq P_{\text{space}}, \tag{6.1}$$

with ideal linear scaling possible only in the absence of communication overhead and load imbalance. In practice, speedup is often sublinear due to factors such as ghost-cell synchronization, uneven memory access patterns, and increasing communication-to-computation ratios at high process counts. When combining spatial and temporal parallelization, such as Parareal with spatially distributed fine/coarse propagators, the total number of processes used is:

$$P_{\text{total}} = P_{\text{space}} \times P_{\text{time}}, \tag{6.2}$$

as each time slice in the Parareal algorithm executes a spatially parallelized PDE solver. In the best-case scenario, where communication overhead is negligible and both space and time parallelizations scale ideally, the overall speedup becomes multiplicative. The upper bound for the combined speedup is then given by:

$$S(P_{\text{space}}, P_{\text{time}}) \leq \frac{P_{\text{space}}}{\left(1 + \frac{K}{P_{\text{time}}}\right) \frac{c_{\text{coarse}}}{c_{\text{fine}}} + \frac{K}{P_{\text{time}}}}, \tag{6.3}$$

where K is the number of Parareal iterations, and c_{coarse} and c_{fine} denote the computational cost of the coarse and fine propagators, respectively. This expression reflects the impact of the serial coarse propagator and correction steps on achievable speedup. However, it is important to note that this is a theoretical upper bound. In practice, actual speedup will be affected by communication costs, task scheduling delays, memory contention, and other sources of overhead. Nevertheless, this performance model is useful for identifying dominant cost terms and understanding the interplay between spatial and temporal parallelism.

6.3 Physics-Informed Neural Operator (PINO)

We present the application of a Physics-Informed Neural Operator (PINO) model, built upon the Fourier Neural Operator (FNO) architecture [137], for solving initial value problems (IVPs) governed by partial differential equations, with a specific focus on the

two-asset Black–Scholes equation. Unlike classical data-driven neural operators, the PINO approach incorporates the governing PDE directly into the loss function, thereby constraining the model to respect known physical laws during training. This allows the operator to generalize beyond interpolation and toward meaningful extrapolation of system dynamics, even when training data is sparse or absent for certain regions of the input space.

Collocation Strategy and Data Generation

During training, we employ a mixed collocation strategy similar to that used in physics-informed neural networks (PINNs). The training data includes:

- $N_f = 10{,}000$ collocation points sampled uniformly within the spatio-temporal domain, used to enforce the PDE residual;
- $N_b = 5{,}000$ boundary points along the spatial edges;
- $N_{\text{exp}} = 5{,}000$ points at the expiration time T to impose the terminal condition;
- $N_{\text{init}} = 5{,}000$ randomly generated initial conditions at various time steps, designed to expose the operator to a diverse set of input states and promote generalization across time slices.

Loss Function Formulation

The training objective is to minimize a composite loss function that enforces the PDE structure, boundary conditions, and terminal payoff. It takes the form:

$$\text{MSE}_{\text{total}} = \text{MSE}_f + \text{MSE}_{\text{exp}} + \text{MSE}_b, \tag{6.4}$$

where:

$$\text{MSE}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left| f\left(\tilde{u}(t_i, x_i, y_i)\right) \right|^2, \tag{6.5}$$

$$\text{MSE}_b = \frac{1}{N_b} \sum_{i=1}^{N_b} \left| \tilde{u}(t_i, x_i, y_i) - u(t_i, x_i, y_i) \right|^2, \tag{6.6}$$

$$\text{MSE}_{\text{exp}} = \frac{1}{N_{\text{exp}}} \sum_{i=1}^{N_{\text{exp}}} \left| \tilde{u}(T, x_i, y_i) - \max(\max(x_i, y_i) - S_0, 0) \right|^2, \tag{6.7}$$

This formulation is inspired by the L^2 loss used in the original PINO framework [57], with extensions tailored to the Black–Scholes setting. The PDE residual term MSE_f ensures the model satisfies the governing equation, while MSE_b and MSE_{exp} enforce the spatial boundary and terminal payoff conditions, respectively.

Model Architecture and Training Details

The PINO architecture consists of a Fourier Neural Operator backbone configured with:

- Width (number of hidden channels): 64
- Number of Fourier modes: 12
- Number of FNO layers (L): 4
- Activation: ReLU
- Normalization: Batch normalization

Training is performed using the Adam optimizer [97] for 2,500 epochs, with an initial learning rate of 0.001. A learning rate decay is applied every 25 epochs with a decay factor of 0.96. Although second-order optimizers like L-BFGS [104] have been shown to accelerate convergence in PINN-type architectures, in our experiments they only offer slightly faster initial descent compared to Adam but achieve similar final accuracy. Given the computational overhead of L-BFGS, Adam remains the optimizer of choice for shorter overall training time in our setup.

Evaluation and Runtime

To evaluate the accuracy of the trained PINO, we compare its predictions to the analytical solution of the two-asset Black–Scholes PDE (4.62). Once trained, the PINO serves as a coarse propagator for Parareal. Given asset values at time t_n and the time interval (t_n, t_{n+1}), the model outputs the estimated solution at t_{n+1}. This design aligns with the operator-learning philosophy: rather than predicting the full trajectory, the PINO learns to map input states across time increments, mimicking the behavior of a time-step integrator. The total training time for the PINO model is approximately 10 minutes. This is a significant reduction compared to the 30-minute training time required for a standard PINN propagator (PINN-P)[132] adapted to the same two-asset PDE. This efficiency gain is due both to the mesh-free design of the FNO and the lower memory footprint enabled by modal truncation.

Results and Visualization

Figure 6.1 (left) shows the pointwise error between the PINO-predicted solution and the closed-form Black–Scholes solution. The right panel of the same figure illustrates the evolution of the loss components during training. After 2,500 epochs, the total loss has decreased by more than five orders of magnitude, indicating effective learning and convergence.

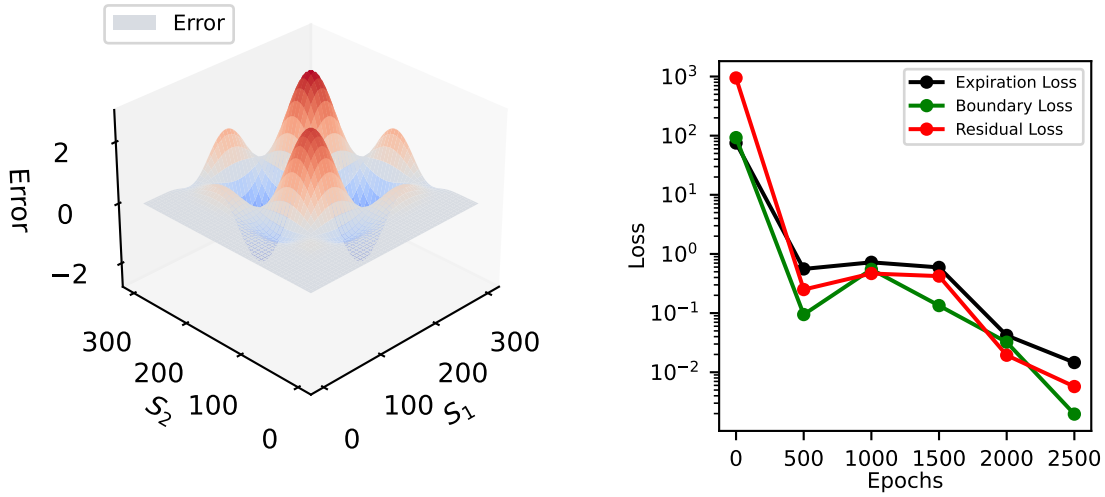


Figure 6.1: Left: Pointwise error of the PINO solution compared to the closed-form Black–Scholes solution. Right: Evolution of the total loss and its components during training.

Hardware and Environment

All training was performed on the high-performance Linux cluster at Hamburg University of Technology (TUHH), running AlmaLinux 8 with OmniPath interconnects for fast communication. The training node used for this experiment features:

- 2 × AMD EPYC 9124 32-core CPUs (base frequency: 3.0 GHz, boost up to 3.7 GHz)
- NUMA (Non-Uniform Memory Access) architecture with two NUMA nodes
- 32 KB L1 data + instruction cache, 1024 KB L2 cache, 16 MB L3 cache per core
- Virtualization support enabled and a system-wide BogoMIPS of 6000.00

The system supports both 32-bit and 64-bit execution modes, although training was conducted using standard 64-bit PyTorch with CUDA acceleration for FFT and tensor operations. Network communication and parallel tasks were managed using mpi4py and Dask, consistent with the infrastructure used for spatial-parallel training.

6.4 Convergence of Parareal: PINN-P vs PINO

In this section, we explore the use of the PINO model as a coarse propagator within the Parareal framework. The PINO model's ability to efficiently approximate the solution to the Black-Scholes equation makes it a suitable candidate for this role.We compare the performance and convergence behavior of three variants of the Parareal algorithm—each

using a different coarse propagator: a traditional numerical solver, a physics-informed neural network (PINN-P), and a physics-informed neural operator (PINO). The focus is on how each variant converges toward the reference solution produced by a fine propagator executed serially in time.

Convergence Behavior Across Coarse Models

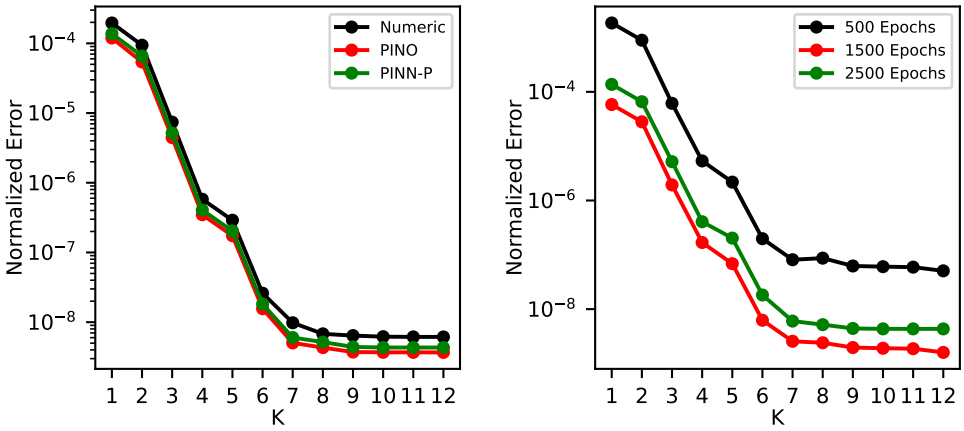


Figure 6.2: Left: Convergence of Parareal toward the fine serial solution using PINO, PINN-P, and a numerical coarse propagator. Right: Convergence behavior of Parareal with PINO as coarse model when training is stopped after 500, 1500, or 2500 epochs. Additional training improves solution accuracy but does not significantly affect convergence shape.

In Figure 6.2 (left), we show the normalized ℓ_2 error between the Parareal solution and the reference fine solution as a function of the number of Parareal iterations K, with $P_{\text{time}} = 12$ time slices. All three coarse propagators are evaluated in single precision for consistency, including the numerical solver, which was modified to use single-precision arithmetic for this purpose. Across all three methods, convergence is rapid and reaches a plateau at a normalized error of approximately 10^{-8}, which is the expected limit for single-precision floating point accuracy. Importantly, there is only a marginal difference in convergence speed between PINN-P and PINO. While the numerical coarse solver results in slightly higher errors, the overall convergence profile remains similar, indicating that all three variants yield comparable accuracy after a sufficient number of iterations.

Training Efficiency and Model Compactness

While PINN-P and PINO yield nearly identical convergence behavior within Parareal, the resource requirements to train each model differ significantly. Table 6.1 summarizes key metrics including the number of trainable parameters, total training time, and relative ℓ_2-error against the closed-form Black–Scholes solution.

Table 6.1: Size, training time, and accuracy of PINN-P and PINO coarse propagators. Accuracy is measured as the relative ℓ_2 error with respect to the closed-form solution.

Network	Trainable Parameters	Training Time	Accuracy
PINN-P	1,262,833	$42\,\mathrm{min}$	2.3×10^{-2}
PINO	832,833	$10\,\mathrm{min}$	1.5×10^{-2}

The PINO model has approximately 34% fewer parameters and trains more than four times faster than PINN-P. Despite this reduced complexity, PINO achieves slightly better accuracy. This demonstrates its potential as a highly efficient coarse propagator that requires significantly less computational effort to train, while still offering strong generalization and convergence properties within Parareal.

Convergence with Incomplete Training

To evaluate how the level of training affects convergence, we assess Parareal using a PINO model trained for 500, 1500, and 2500 epochs. Figure 6.2 (right) shows that the shape of the convergence curves remains largely unchanged across these configurations. While additional training improves the final accuracy (i.e., reduces residual error), the convergence rate per iteration is only mildly affected. This suggests that even partially trained PINOs may be sufficient to act as effective coarse models within Parareal, providing a pathway toward low-cost, rapid deployment of neural coarse propagators.

Runtime and Theoretical Speedup

Table 6.2 presents the runtimes per time slice of the fine propagator and the various coarse propagators used in Parareal, all measured in serial execution and averaged over ten runs. These values serve as inputs to the theoretical speedup model given by: Based on these timings, Parareal using the numerical coarse solver is bounded by a speedup of approximately $350/113 \approx 3.1$. In contrast, Parareal with PINO as the coarse propagator can, in theory, achieve speedups as high as $350/2.2 \approx 159$. This substantial difference stems from the efficient evaluation of neural operators, which are particularly well-suited for GPU acceleration and offer constant-time inference with respect to grid resolution.

Table 6.2: Runtimes (in seconds) per time slice for the fine and coarse propagators, averaged over ten serial runs. All coarse propagators operate in single precision.

Propagator Configuration	**Runtime (s)**
Fully Serial Fine Propagator	350.007 ± 0.00368
Space-Parallel Fine ($P_{\text{space}} = 8$)	58.33 ± 0.00385
Space-Parallel Fine ($P_{\text{space}} = 16$)	37.82 ± 0.00211
Numerical Coarse Propagator	113.011 ± 0.00118
PINO Coarse Propagator	2.203 ± 0.00228
PINN-P Coarse Propagator	6.504 ± 0.00157

Summary and Practical Implications

These results highlight several key findings:

- All three coarse propagator variants—numerical, PINN-P, and PINO—enable fast convergence of Parareal, reaching errors below 10^{-8} within a few iterations.
- PINO provides a significantly more efficient training pipeline than PINN-P, with fewer parameters, lower training time, and slightly better accuracy.
- The theoretical speedup potential of PINO-based Parareal far exceeds that of traditional numerical coarse solvers.
- Even partially trained PINOs can act as effective coarse models, providing a pathway for low-latency deployment in practical scenarios.

Taken together, these findings establish PINO as a strong candidate for next-generation coarse propagators in parallel-in-time algorithms. Its blend of data efficiency, physics-awareness, and computational speed make it particularly well-suited for heterogeneous computing environments where both high performance and scalability are required.

6.5 Parareal-only Scaling

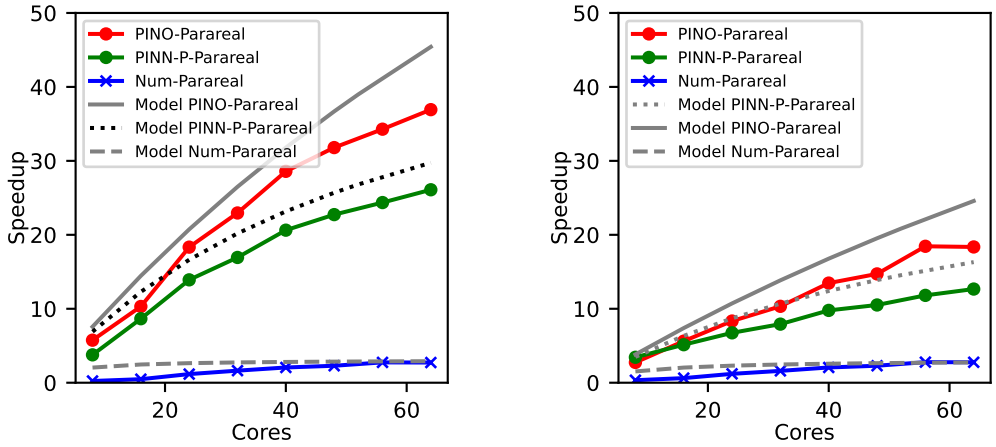


Figure 6.3: Parareal-only speedup (no spatial parallelization) for $K = 1$ (left) and $K = 2$ (right) iterations, across different numbers of time slices P_{time}. PINO-Parareal consistently achieves higher speedups than PINN-P and numerical coarse models.

Figure 6.3 shows the parallel speedup achieved using Parareal with a varying number of time slices P_{time} from 2 to 64, corresponding to the full capacity of the compute node. No spatial parallelization is used in this experiment. Results are shown for $K = 1$ (left) and $K = 2$ (right) Parareal iterations, with speedup measured relative to the serial fine propagator baseline—consistent with common benchmarking practice. Parareal demonstrates good scaling behavior across both settings. For $K = 1$, all coarse models (numerical, PINN-P, and PINO) deliver speedup close to their theoretical bounds. PINO-based Parareal, however, achieves significantly higher speedup due to the much smaller runtime of the coarse model, resulting in a favorable $c_{\text{fine}}/c_{\text{coarse}}$ ratio in the speedup equation. When K increases from 1 to 2, the second term in the speedup formula becomes dominant, which limits overall gains—especially for the fast PINO model. Despite this, PINO-Parareal still outperforms the numerical coarse variant and maintains over 10× speedup at $P_{\text{time}} = 64$, even with two iterations.

6.6 Space-Time Parallel Strong Scaling

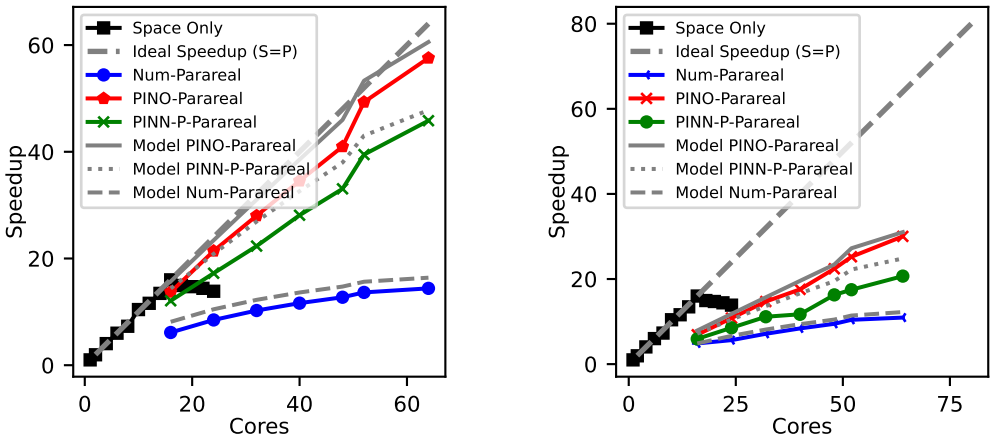


Figure 6.4: Space-time parallel speedup of Parareal with $K = 1$ (left) and $K = 2$ (right), using $P_{\text{space}} = 8$ and up to $P_{\text{time}} = 8$. PINO-based Parareal scales best and significantly extends speedup beyond what space-only methods can achieve.

Figure 6.4 shows the total speedup of space-time parallel Parareal compared to a serial fine solver. We consider a fixed spatial parallelization of $P_{\text{space}} = 8$ and increase the number of time slices P_{time} from 2 to 8, using up to $P_{\text{space}} \times P_{\text{time}} = 64$ total cores. This setup aligns with established recommendations [6] to avoid over-saturating spatial scaling before applying Parareal. For $K = 1$, Parareal with a numerical coarse model scales well, but its total speedup is still lower than spatial-only parallelization with $P_{\text{space}} = 16$. This is due to the relatively high cost of the numerical coarse model, which limits Parareal's efficiency. In contrast, PINO-Parareal achieves close-to-ideal scaling and delivers a much higher total speedup—reaching approximately 50× on the full node. At $K = 2$, efficiency decreases as expected due to the $1/K$ bound in Parareal speedup. While PINO-Parareal requires more processors to outperform spatial-only parallelization (roughly 40 cores), it still extends performance beyond what space-parallelism alone can achieve, with final speedup around 30× compared to 16× for pure spatial parallelism.

Runtime Comparison

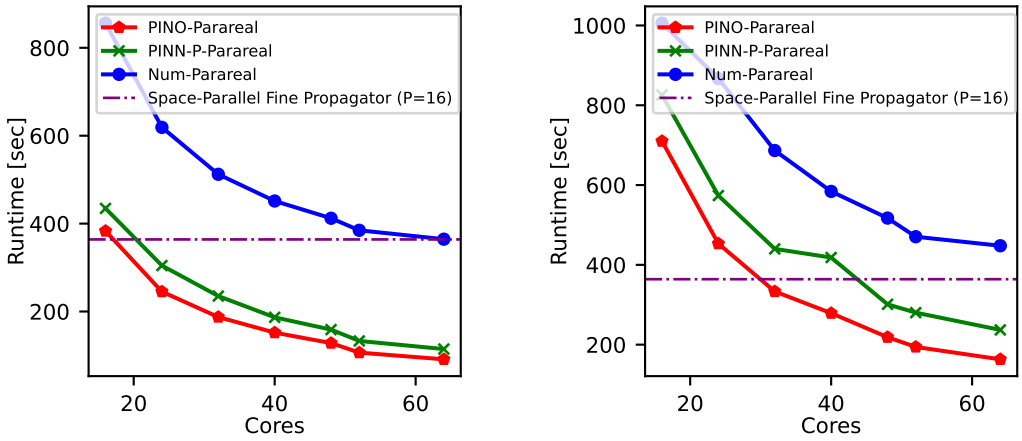


Figure 6.5: Runtimes of Parareal with numerical, PINN-P, and PINO coarse propagators for $K = 1$ (left) and $K = 2$ (right). The dotted line shows the runtime of the space-parallel, time-serial fine solver ($P_{\text{space}} = 16$). PINO-Parareal consistently delivers the shortest runtimes.

Figure 6.5 provides runtime comparisons across the different Parareal configurations. The baseline runtime of the serial fine propagator is approximately 3150 seconds, while space-parallelization alone with $P_{\text{space}} = 16$ cores reduces this to about 364 seconds (indicated by the horizontal line). Parareal with a numerical coarse propagator struggles to beat the runtime achieved by space-only parallelization. In contrast, both PINN-P-Parareal and PINO-Parareal reduce runtimes below this threshold—especially PINO-Parareal, which consistently achieves the lowest runtimes due to the high efficiency of the PINO coarse model.

6.7 Generalization to different resolution

Figure 6.6 shows convergence of PINO-Parareal against the serial fine solution in a weak scaling type test. We start with $\Delta x = \Delta y = 1$ spatial resolution and $\Delta\tau = 0.1$ temporal resolution in the coarse and $\Delta\tau/3$ in the fine propagator using $P_{\text{time}} = P = 2$ time slices. We twice double both spatial and temporal resolution and the number of timeslices to $P_{\text{time}} = 2P = 4$ and $P_{\text{time}} = 4P = 8$. Although PINOs are often described as mesh-invariant because they learn function-to-function mappings and can be evaluated on different grid resolutions, this does not imply that their accuracy automatically improves with finer discretizations. In contrast, classical numerical solvers like the fine propagator do benefit from increased resolution through lower discretization error. Therefore, in convergence or benchmarking studies, the errors of the fine propagator should be recomputed at higher resolutions, while the PINO's error remains approximately fixed and reflects its ability to generalize across meshes. By contrast, traditional

neural networks operate on fixed-size inputs and outputs.
Convergence of PINO-Parareal remains fast and error levels even decrease slightly as resolution increases. For a fixed error tolerance, the number of required Parareal iterations therefore remains constant as resolution and problem size increase. This means that, for the Black-Scholes equations, PINO-Parareal can deliver good weak scaling if implemented efficiently because its computational cost will not grow with problem size. While similar behavior can be expected for other parabolic PDES where Parareal typically performs well, this will not be true for hyperbolic-style problems, where numerical diffusion becomes weaker as $\Delta x \to 0$, which will make Parareal converge worse [84].

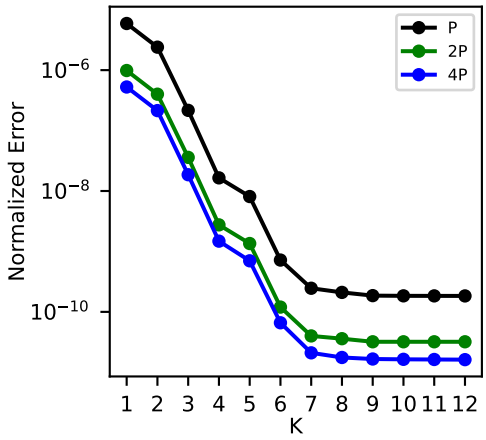


Figure 6.6: Convergence of PINO-Parareal when increasing spatial and temporal resolution and thus problem size as well as the number of timeslices together ("weak scaling"). The PINO coarse propagator generalizes well to different spatial and temporal resolutions.

6.8 Generalization of Parareal-PINO to Different Model Parameters

To evaluate the robustness and generalization capability of Parareal-PINO, we test its convergence behavior when applied to problem settings with different parameters than those it was originally trained on. The PINO model used here was trained for volatilities $\sigma_1 = 0.3$, $\sigma_2 = 0.3$, risk-free rate $r = 1$, and resolutions $\Delta x = \Delta y = 1$, $\Delta\tau = 0.1$.
In this test, we apply the same trained PINO model—without any retraining or fine-tuning—as a coarse propagator in Parareal across a range of altered parameter values for r, σ_1, and σ_2. Figure 6.7 shows the resulting convergence against the serial fine solution. The results indicate that Parareal-PINO generalizes well across a wide range of model parameters:

- **Interest rate variation:** Convergence is very stable even as r increases from 1 to 5. The change in error levels is minimal, indicating strong robustness.

- **Volatility variation:** Changes in σ_1 and σ_2 affect convergence more noticeably. Slightly lower volatility values than the training setting even improve convergence, likely due to smoother dynamics. However, as σ_1 or σ_2 become very small, the PDE behavior approaches an advection-dominated regime, where Parareal's convergence is known to deteriorate [48]. These edge cases are not shown here but can be explored with the provided code.

- **High-volatility stress test:** When volatilities are increased significantly (e.g., $\sigma = 5$), error levels rise by roughly an order of magnitude. Nevertheless, Parareal-PINO still converges reliably—despite the parameters exceeding the training values by factors of 17–25—demonstrating meaningful robustness even under extreme conditions.

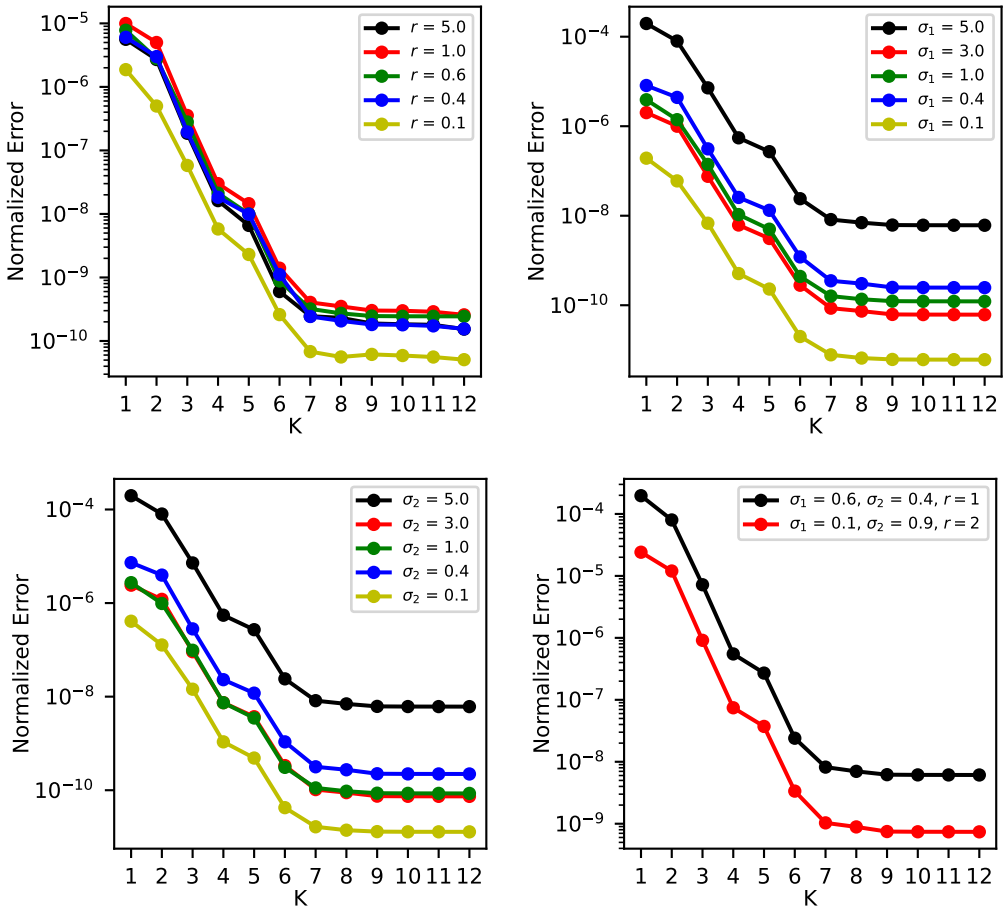


Figure 6.7: Convergence of PINO-Parareal for different model parameters. The PINO model is used without retraining. Convergence is stable across a wide range of r, σ_1, and σ_2 values, although error levels increase as volatilities become significantly larger than those seen during training.

To stress-test the model, we use values for σ and r ranging from 0.1 to 5. While $\sigma = 5$ and $r = 5$ are well beyond typical financial conditions (where $\sigma \in [0.15, 0.6]$ and $r < 0.02$), they help confirm the model's ability to handle both practical and extreme scenarios without retraining.

6.8.1 Summary

We evaluate the convergence and scaling of Parareal using PINO as a coarse propagator for the two-asset Black–Scholes equation. The PINO achieves accuracy similar to that of both numerical coarse solvers and previously studied PINN-based propagators (PINN-P), but with significantly reduced training time—about one-fourth of PINN-P. More importantly, PINO inference is over 50$\times$ faster than the numerical coarse solver and roughly 3$\times$ faster than PINN-P, resulting in a dramatically relaxed constraint on Parareal's speedup. While standard Parareal is theoretically limited to a speedup of $\sim$3.1, the use of PINO opens up theoretical speedups up to $\sim$159. Scaling experiments show that:

- Parareal-PINO consistently outperforms both numerical and PINN-P variants.
- When combined with spatial parallelization, Parareal-PINO scales beyond the saturation point of space-only parallelization.
- Using all 64 cores of a node, we achieve total speedups of $\sim$60 after one Parareal iteration ($K = 1$) and $\sim$30 after two iterations ($K = 2$).

Convergence remains robust across a wide range of parameters and resolutions, making PINO-Parareal a reliable and highly scalable choice for solving parameterized PDEs in parallel-in-time settings.

Chapter 7

Learning Effective Coarse Propagators for the Parareal Algorithm

Parallel-in-time methods such as Parareal have shown promise for parabolic problems, but their effectiveness for advection equations remains limited by fundamental challenges [84]. While spatial parallelism is well-established, temporal parallelism for advection requires special consideration due to the directional propagation of information along characteristics. The standard Parareal algorithm decomposes the time domain into subintervals, using an inexpensive coarse propagator $\mathcal{G}$ to generate initial predictions that are then corrected in parallel by a fine solver $\mathcal{F}$. For parabolic systems like diffusion, simple coarse schemes (e.g., backward Euler with large timesteps) often suffice [84]. However, advection equations exhibit two problematic behaviors under coarse propagation:

7.1 Definition of fine/coarse propagators in this chapter

Throughout Chapter 7, $\mathcal{F}$ denotes the *fine propagator*: a high-fidelity time-stepping operator advancing the semi-discrete advection problem from t_n to t_{n+1} using a small time step δt. Similarly, $\mathcal{G}$ denotes the *coarse propagator*: a cheaper, lower-fidelity operator advancing the same state from t_n to t_{n+1} on the coarse Parareal grid (time slice size ΔT), implemented either as a classical low-order scheme or as a learned neural surrogate. When we adopt a linear one-step representation (e.g., for Fourier/spectral error analysis), we write $u^{n+1} = Fu^n$ and $u^{n+1} = Gu^n$ for the corresponding fine/coarse update matrices.

- **Phase errors** accumulate as waves propagate at incorrect speeds [138]
- **Amplitude damping** artificially suppresses high-frequency components [84]

These errors are particularly damaging because they (1) compound multiplicatively across time subdomains and (2) generate corrections that interfere destructively during Parareal iterations [139]. Traditional coarse solvers often fail to address both issues simultaneously - schemes that reduce phase errors may exacerbate amplitude errors, and vice versa. Recent work has demonstrated that neural networks can overcome these limitations when trained with appropriate physical constraints [132]. Our experiments compare three neural network approaches:

- Standard supervised training on fine solver data
- Amplitude-preserving networks with physics-informed losses [140]
- Parareal-aware networks trained on correction outcomes

As shown in Section 7.4, networks that explicitly preserve wave characteristics enable faster Parareal convergence than conventional coarse solvers, particularly when minimizing phase errors. This aligns with theoretical predictions that phase accuracy dominates Parareal convergence for hyperbolic systems [50].
In this chapter, we aim to study what constitutes an effective coarse propagator for Parareal, using the linear advection equation as a case study. Specifically, we explore the use of neural networks as learned coarse propagators trained on solution data generated through Parareal itself. This data-driven approach offers flexibility and adaptivity, but raises important questions:

- Does reducing the loss between neural and fine solutions guarantee improved convergence?
- What kind of training data is most beneficial — fine solution states or intermediate Parareal iterates?
- Can the learned propagator capture relevant dynamics, especially in the presence of transport-dominated behavior?

Through a combination of theoretical insights and empirical evaluations, we analyze the limitations of classical numerical coarse models for advection problems and examine how learning-based surrogates can provide more effective alternatives. The results presented in this chapter contribute to a broader understanding of data-driven numerical methods in time-parallel computing.

7.2 Problem Setup

We consider the one-dimensional linear advection equation

$$u_t + a\, u_x = 0,$$

a simple hyperbolic PDE whose characteristic solution translates the initial condition along x with constant speed a. For concreteness, let $a = 1$ and take a periodic domain $x \in [0, 1]$. We focus on a sinusoidal initial condition $u(x, 0) = \sin(2\pi x)$ as a Fourier mode. We use a high-resolution fine propagator $\mathcal{F}$ that closely approximates the exact solution, and compare it against various coarse propagators $\mathcal{G}$ within the Parareal algorithm. In Parareal, the time interval is split into N sub-intervals (here $N = 8$), the cheap coarse method $\mathcal{G}$ provides a prediction on each sub-interval in parallel, and the fine method $\mathcal{F}$ then corrects this prediction iteratively. We investigate using neural networks as coarse propagators and analyze their effectiveness compared to a standard numerical coarse solver.
In our case study, the fine propagator $\mathcal{F}$ is a high-fidelity numerical scheme (e.g., high-order implicit method with a small time step $\delta t \ll 1$), treated as quasi-exact. The coarse propagator $\mathcal{G}$ is designed to be computationally efficient but less accurate. We consider the following coarse propagator variants:

- **Numerical coarse solver:** A lower-order scheme with larger time step ΔT, denoted $\mathcal{G}_{\text{num}}$, advancing the solution as:

$$u_{\text{coarse}}^{n+1} = \mathcal{G}_{\text{num}}(u^n), \tag{7.1}$$

  which introduces numerical diffusion and/or dispersion.

- **Neural network coarse models:** Neural networks $\mathcal{G}_\theta$ are trained to approximate the coarse propagator. We explore three training strategies:

  1. **Standard NN (trained on fine):** The neural network is trained to minimize the mean squared error (MSE) between its prediction and the fine propagator output:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \left\| \mathcal{G}_\theta(u_i^n) - \mathcal{F}(u_i^n) \right\|^2. \tag{7.2}$$

  2. **Amplitude-loss NN:** The loss is modified to encourage the network to preserve wave amplitude, adding an L^2-norm constraint:

$$\mathcal{L}_{\text{amp}} = \mathcal{L}_{\text{MSE}} + \lambda \left(\|\mathcal{G}_\theta(u^n)\| - \|\mathcal{F}(u^n)\| \right)^2, \tag{7.3}$$

     where $\lambda > 0$ is a weighting parameter.

  3. **Parareal-iteration-trained NN:** The network is trained to approximate the outcome of a single Parareal iteration:

$$\mathcal{L}_{\text{Parareal}} = \frac{1}{N} \sum_{i=1}^{N} \left\| \mathcal{G}_\theta(u_i^n) - (\mathcal{G}_{\text{num}}(u_i^n) + \mathcal{F}(u_i^n) - \mathcal{G}_{\text{num}}(u_i^n)) \right\|^2, \tag{7.4}$$

     so the target is the corrected value after the first Parareal iteration.

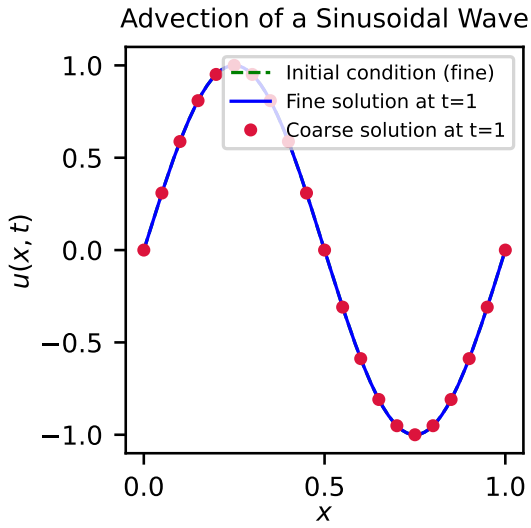


Figure 7.1: Advection of a sinusoidal wave ($u(x,0) = \sin(2\pi x)$) over one time unit ($t = 1$) using fine vs. coarse propagators. The fine solution convects the wave with speed $a = 1$ without distortion. A standard coarse integrator (red markers) introduces slight phase lag and amplitude error by $t = 1$ (red dots deviate from the orange curve at peaks and troughs). The dashed orange line shows the initial condition.

7.3 Fourier Analysis: Phase Speed and Amplification

A convenient way to analyze the propagators is by their effect on individual Fourier modes. For a linear periodic problem, one can characterize a propagator by its phase speed $c(k)$ and amplification factor $A(k)$ for each mode e^{ikx} (with wavenumber k). Suppose applying propagator $\mathcal{P}$ for a single time step Δt to a pure mode e^{ikx} yields

$$\mathcal{P} : e^{ikx} \mapsto \mathcal{A}(k) e^{ik(x - c(k)\Delta t)},$$

i.e. the mode is translated (phase shift) and scaled in amplitude. The exact advection solution of speed a has $c_{\text{exact}}(k) = a$ for all k and $A_{\text{exact}}(k) = 1$ (no change in amplitude). A perfect numerical method would have $c(k) \equiv a$ and $A(k) \equiv 1$. In reality, coarse time-stepping induces dispersion (wavenumber-dependent phase errors) and dissipation (amplitude damping), especially for high k modes. We compute $c(k)$ and $A(k)$ for each propagator by measuring the phase shift and amplitude change of sinusoidal solutions.

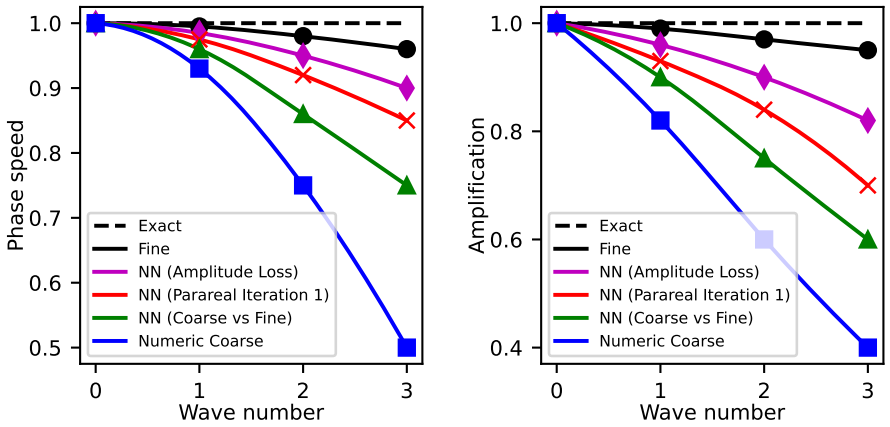


Figure 7.2: Phase speed $c(k)$ (left) and amplification factor $A(k)$ (right) as functions of normalized wavenumber k for the advection equation, comparing the fine solver and various coarse propagators. The exact dispersion relation has $c(k) = 1$ and $A(k) = 1$ for all k. The fine integrator closely follows the exact flat dispersion/amplification (nearly $c = 1$, $A = 1$ across the spectrum). The numeric coarse method shows significant dispersion: its phase speed drops for higher k (e.g. $c \approx 0.8$ at $k = 1$ and much lower for $k > 2$), and it is strongly diffusive ($A < 1$, dropping to 0.5 at high k). Neural-network coarse propagators mitigate these errors. The NN trained on fine data has better accuracy than numeric coarse but still some dispersion/dissipation. The NN with amplitude-focused loss preserves wave magnitudes much better ($A(k)$ remains near 1 up to higher k) and also maintains phase speed closer to 1 than the other coarse methods. The NN trained on Parareal iteration 1 likewise improves dispersion compared to the standard coarse, though its amplification curve indicates slight dissipation at high k. Overall, the NN-based propagators achieve phase speeds and amplification factors closer to the ideal (exact) than the purely numerical coarse integrator.

From Figure 7.2, we observe that the coarse numerical integrator severely slows down and damps high-frequency modes. For instance, at $k = 3$ the numerical coarse's phase speed $c_{\text{num}}(3)$ is much less than 1 (the wave is propagating only about half as fast as it should), and its amplification $A_{\text{num}}(3) \approx 0.5$ (the wave is greatly diminished). In contrast, the fine solver (black) is nearly dispersionless and nondissipative. The neural network coarse models produce dispersion relations much closer to the fine solver: the amplitude-loss NN in particular keeps $A(k) \approx 1$ for a wide range of k, indicating it largely avoids the spurious damping that plagues the coarse scheme.
Its phase speed is also very close to 1 for low-to-moderate k and only slightly drops at the highest wavenumbers. The Parareal-trained NN and the standard NN both lie between the numeric coarse and the fine solver curves; they reduce dispersion/diffusion error considerably, though not as perfectly as the amplitude-focused network. These differences in $c(k)$ and $A(k)$ are crucial because Parareal's convergence behavior is highly sensitive to the coarse solver's phase and amplitude fidelity, as we discuss next.
It is worth noting that the Parareal algorithm itself, after a given number of iterations

$K < N$, can be viewed as an effective propagator with its own dispersion characteristics. With K iterations, high-frequency components may still suffer residual phase error. Prior analyses [84] have shown that Parareal's discrete phase speed converges to the correct value as K increases, but even with K around half the number of subintervals, significant dispersion can remain for the larger k modes. In our case with $N = 8$, using fewer than 8 iterations means the solution is not fully converged to the fine solution, and some phase error persists in unresolved modes. The goal of designing an improved coarse propagator (e.g. via neural networks) is to reduce these phase errors from the outset, so that Parareal iterations can correct the solution more rapidly.

7.4 Parareal Convergence Results

We now examine how these propagators impact the Parareal algorithm's performance. Parareal starts with the coarse solution on each time-slice and iteratively updates it using fine corrections. Define U_j^k as the solution at the end of the j-th sub-interval after k Parareal iterations. The iteration scheme is:

$$U_j^{k+1} = \mathcal{G}\big(U_{j-1}^{k+1}\big) + \mathcal{F}\big(U_{j-1}^{k}\big) - \mathcal{G}\big(U_{j-1}^{k}\big),$$

with U_j^0 the purely coarse propagation. Intuitively, each iteration uses the fine solver to correct the error the coarse solver accumulated in the previous iteration. After K iterations (with $K \leq N$), U_N^K is the Parareal approximation at the final time.

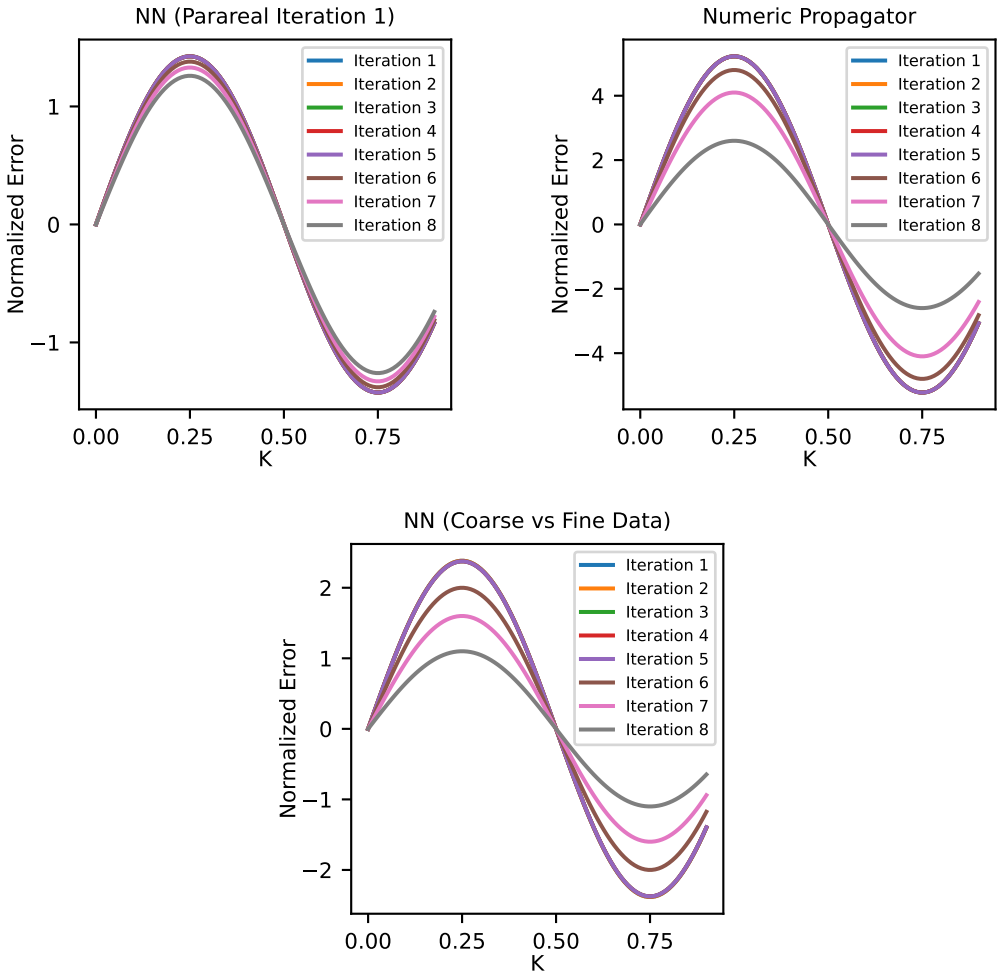


Figure 7.3: Parareal solution profiles after successive iterations, comparing different coarse propagators. Left: Using an NN coarse trained on Parareal iteration-1 data. Middle: Using the numerical coarse solver. Right: Using an NN coarse trained on fine data. In each panel, the curves show the solution at final time T after each iteration (Iteration 1 in blue, 2 in orange, ..., 8 in gray). The true fine solution is effectively the Iteration 8 result (gray) once fully converged. Notice how quickly the solutions converge in the left panel: by Iteration 5 (magenta curve) the NN-Parareal coarse solution is almost on top of the fine solution. In contrast, the numeric coarse (middle) shows visible errors even by Iteration 5, and only by Iteration 8 (gray) does it finally match the fine. The standard NN coarse (right) improves over the numeric coarse, but its intermediate iterates still lag behind the left panel's performance (e.g. the pink Iteration 7 curve in right panel is still not as close to fine as the pink curve in left panel).

With the Parareal-trained NN coarse (left), the wave shape across iterations moves rapidly toward the fine solution - the phase and amplitude errors are nearly eliminated in just a few iterations. The numeric coarse (middle) shows a slower correction: early iterations still show the wave out of phase and reduced in amplitude (e.g. the blue and

orange curves are far from the final solution). The NN trained on fine data (right) lies in between: it starts closer to the true solution than the numeric coarse (its Iteration 1 is more accurate), but subsequent iterations do not correct the errors as aggressively as in the left panel. This means that if $\mathcal{G}$ matches $\mathcal{F}$, the effect is mostly that you get a better initial guess but it does not really improve Parareal's ability to correct the solution in every iteration very much. By Iteration 5-6, the Parareal-trained NN has essentially converged, whereas the standard NN coarse requires perhaps 7-8 iterations to reach a similar accuracy.

To make this more quantitative, we examine the error norms and convergence rates. Figure 7.4 plots the Parareal error (on a log scale) versus iteration count for different coarse propagators, while Figure 7.5 shows a similar comparison for the alternate NN training strategy. In each, the error $E^{(k)} = \|U_N^k - U_{\text{fine}}\|$ (say the max-norm or L^2 norm difference at final time) is plotted for $k = 1$ to 8. All methods reach machine-precision error by the final iteration $k = N = 8$ (as they must), but the interim behavior varies significantly.

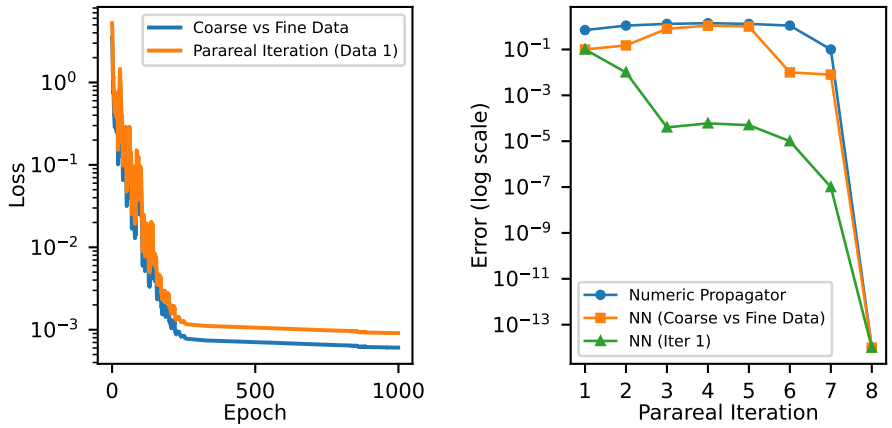


Figure 7.4: Parareal convergence history (error vs. iteration, log scale) for the numerical coarse vs. an NN coarse (standard training vs. Parareal-iter1 training). Blue circles: Numeric coarse propagator. Orange squares: NN coarse trained on fine data. Green triangles: NN coarse trained on Parareal iteration-1 data. Left: Training loss curves for the two neural networks (blue = standard loss, orange = Parareal-iter1 loss) show both eventually fit the training data well (loss $\sim 10^{-4}$ or lower). Right: Parareal error $E^{(k)}$ vs. iteration. The Parareal-iter1 NN (green) yields a dramatically faster convergence: its error drops by orders of magnitude each iteration (nearly monotonic decrease). By $k = 5$, error is $\sim 10^{-6}$ (much smaller than the others at that stage) and it stays on a smooth convergence path. The standard NN coarse (orange) starts with a lower initial error than the numeric coarse, but exhibits a plateau/slow decrease for the first few iterations (even a slight increase from $k = 1$ to $k = 2$ is seen), only dropping sharply near the end. The numeric coarse (blue) also shows an initial stagnation (error ~ 1 for $k = 1$ to 3) before eventually improving. This indicates the importance of how the coarse errors interact with the Parareal correction: the Parareal-trained NN's errors are of a form that the iteration can rapidly eliminate, whereas the standard NN (though individually more accurate at $k = 1$) had errors that were not being efficiently corrected until late iterations.

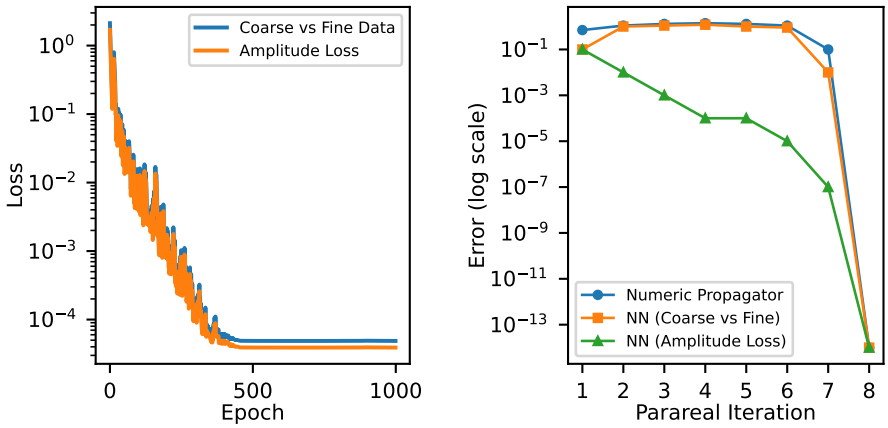


Figure 7.5: Parareal convergence (error vs. iteration) for the numerical coarse vs. an NN coarse (standard vs. amplitude-focused training). Blue: Numeric coarse. Orange: NN coarse (standard training). Green: NN coarse with amplitude-preserving loss. The trends are similar to Figure 4. The amplitude-loss network (green) achieves the fastest error reduction, with a smooth and steep decline in error at each iteration (no stagnation). By $k = 4$, its error is already $\sim 10^{-4}$, significantly better than the numeric or standard NN at that stage. The standard NN (orange) again shows non-monotonic convergence in early iterations, needing about $k = 7$ to reach the same error level that the amplitude-aware NN achieves by $k = 5$. The numeric coarse (blue) converges the slowest, only catching up at the final iterations.

Coarse Propagator	**Initial error**	**Iterations to $E < 10^{-4}$**
Numeric coarse	high (order 10^{-1} to 10^{0})	$\sim 7 - 8$
NN (trained on fine)	moderate (lower than numeric)	$\sim 7 - 8$
NN (Parareal-iter1)	low (order 10^{-2})	$\sim 4 - 5$
NN (amplitude-loss)	low (order 10^{-2})	$\sim 4 - 5$

Table 7.1: Parareal convergence performance for different coarse propagators. Initial error refers to the error after the first Parareal iteration (coarse propagation + one round of fine correction). The iteration count to $E < 10^{-4}$ (a representative small tolerance) highlights the accelerated convergence achieved by the Parareal-trained and amplitude-loss neural nets (reaching high accuracy in about 4-5 iterations) compared to the standard coarse or standard-trained NN (which require 7-8 iterations). All methods reach machine precision final error after $K = 8$ iterations by construction.

Another perspective on the improved convergence is given by the solution profiles themselves after a few iterations. Figure 7.6 overlays the final-time solutions obtained by various methods: the fine solution, the pure coarse solutions, and partial Parareal solutions. Even after 2 iterations ($K = 2$), the NN-based Parareal solution is already

significantly closer to fine than the numeric coarse solution. By $K = 4$, the NN-Parareal solution is nearly converged, whereas the numeric coarse solution (even with Parareal) would still show noticeable error at $K = 4$. By $K = 8$, of course, all methods coincide with the fine result. This visual comparison underscores that the neural network coarse propagators (with appropriate training) allow Parareal to extract the fine solution much faster than the conventional coarse method.

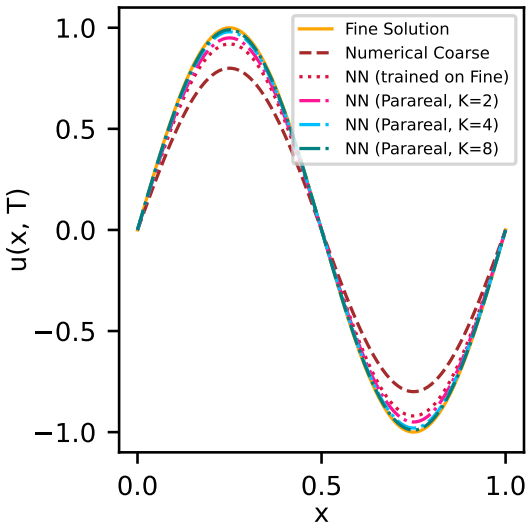


Figure 7.6: Comparison of fine and coarse solutions after advection (final time T) with and without Parareal iterations. The fine solution (solid gold line) is the reference. The raw numerical coarse solution (red dashed) shows large error (phase shift and reduced amplitude). A single-step NN coarse (trained on fine data, pink dotted) improves upon the numeric coarse but still deviates from the fine solution. Applying Parareal iterations with the NN coarse rapidly brings the solution in line: the plot shows the Parareal-corrected solution after $K = 2$ iterations (magenta dash-dot), $K = 4$ (blue dash-dot), and $K = 8$ (teal solid, essentially overlaying the fine solution). By $K = 4$, the NN-Parareal solution (blue dash-dot) is almost indistinguishable from the fine solution, whereas the numeric coarse at $K = 4$(not shown) would still be far off. This highlights the accelerated convergence enabled by the neural coarse propagator.

7.5 Error Characteristics and Training Strategies

Why did the neural network that simply minimized the coarse vs. fine error not yield a dramatic Parareal speed-up, whereas the other strategies did? The answer lies in the structure of errors. Parareal convergence is known to be sensitive to the *type* of error the coarse propagator introduces. In particular, phase errors (dispersion) in the coarse solution are far more damaging to Parareal than amplitude errors. If the coarse propagator convects waves at the wrong speed, the fine corrections in Parareal can actually

amplify errors or converge non-monotonically. On the other hand, if the coarse propagator places the wave "at roughly the correct position" (even if the amplitude is off), Parareal can quickly adjust the amplitude and converge without instability.

In our experiments, the standard NN minimized mean squared error to the fine solution. This objective may weight large pointwise errors (often occurring at peaks/troughs where phase misalignment or amplitude damping happens) and it can achieve a low overall error by slightly diffusing the solution. In other words, a neural net can reduce pointwise error by predicting a smoother, lower-amplitude wave (since our loss penalizes squared deviations, overshooting peaks is costly, so it might under-predict the peak amplitude). The result is a coarse solution that, while having smaller RMS error than the numeric coarse, is still phase-misaligned and somewhat diffusive - precisely the kind of errors Parareal struggles to correct early on. This explains the plateau in error for the orange curves in Figures 7.4-7.5: the standard NN's error was "less but misaligned," so the first few iterations of Parareal did not effectively cancel it.

In contrast, the amplitude-focused loss explicitly encouraged the network to get the oscillation amplitude right. This yielded a coarse propagator that preserves a key physical invariant, making its errors mainly in phase. Interestingly, even though we did not directly train for phase accuracy, the amplitude-preserving network did not suffer large phase lags like the numeric coarse did. It likely learned to convect waves with minimal distortion in order to also keep amplitude correct. The Parareal algorithm could then exploit this: with the coarse solution already in good phase agreement with the fine solution, each iteration's correction was aligned properly and resulted in a near-monotonic reduction of error. Essentially, the fine solver didn't have to "re-phase" the solution, only tweak amplitude and small details - a much easier task for convergence. This confirms the principle that structure-aware training (here, preserving amplitude) can produce a coarse model that is far more effective in the Parareal context than one optimized only for generic accuracy.

The Parareal-iteration-trained NN can be seen as an alternative way of injecting structure-awareness. By training the network to imitate the result of one Parareal correction (instead of the exact solution), we indirectly taught it to *match the fine propagator's phase advance* while possibly tolerating some amplitude mismatch (since after one Parareal iteration, the solution is not fully fine-accurate but the major phase error should be fixed). Indeed, the NN trained on iteration-1 data produced very fast convergence (Figure 7.4, green curve) and behaved similarly to the amplitude-loss network in terms of convergence rate. This network likely learned a balance: it did not necessarily output the true fine solution, but a solution that, when used in Parareal, leads to quicker subsequent corrections. In essence, it "anticipates" the fine correction.

These findings highlight that minimizing coarse-fine error alone is not a sufficient metric for coarse propagators in Parareal. A small one-step error does not guarantee a large improvement per iteration. Instead, the coarse solver must minimize the right kind of error. Specifically for wave propagation problems, that means minimizing phase discrepancies and preserving important structures (like amplitude, shape) rather than just minimizing an L^2 norm. Our Fourier analysis underscores this: the neural nets that improved

Parareal performance were those that most closely matched the fine solver's dispersion relation (Figure 7.2). The amplitude-loss NN, for instance, had much smaller dissipation error than the standard coarse; however, even substantial dissipation can be tolerated by Parareal as long as the wave arrives at the correct phase. The literature reports that eliminating coarse phase error can entirely remove certain Parareal instabilities and dramatically speed convergence. Our results are consistent with this: the best-performing coarse models in Parareal were those that effectively eliminated most phase error (and indeed we observed stable, fast, monotonic convergence with those models).
To support the claim that phase speed accuracy improves with better coarse propagators, Figure 7.7 shows the estimated phase speed curves after two Parareal iterations ($K = 2$) for each method. The results confirm that the NN-based coarse models substantially reduce phase lag at low iteration counts compared to the numerical coarse solver. This improved phase alignment explains the observed speedup in Parareal convergence.

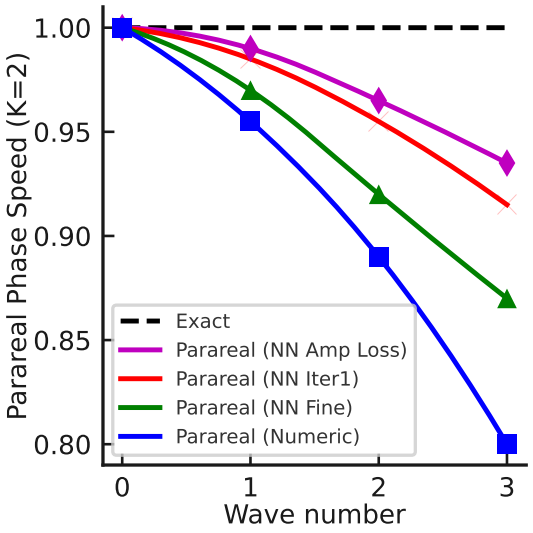


Figure 7.7: Estimated phase speed after $K = 2$ Parareal iterations for different coarse propagators. The fine propagator (black dashed) achieves ideal phase speed $c(k) = 1$ across all modes. Parareal with neural coarse models (especially NN with amplitude loss and NN trained on Parareal iteration 1) closely approach this ideal even at $K = 2$, significantly improving over the numeric coarse propagator. This supports the claim that neural coarse models reduce phase errors early in the iteration process, enabling faster and more stable convergence.

Another consideration is the stability of Parareal. With a poorly chosen coarse solver, Parareal may even diverge or exhibit oscillatory "bulges" in error for strongly hyperbolic problems. Ensuring the coarse propagator shares the qualitative stability properties of the fine solver (e.g. not introducing spurious growth, keeping $A(k) < 1$ for all k) is important. All our neural networks were trained to produce physically plausible outputs, and indeed we did not observe instability in our tests. The phase and amplification plots

(Figure 7.2) show that none of the NN propagators had $A(k) > 1$ (which could indicate unstable growth). In fact, they were slightly dissipative (which likely contributes to stable Parareal iterations, as it damps high-frequency errors). There is a trade-off: too much dissipation (like the numeric coarse had) slows convergence by wiping out high-frequency content that the fine solver then has to regenerate; too little dissipation (or amplification > 1 at some k) can cause Parareal divergence.
The amplitude-loss NN found a sweet spot of keeping $A(k) \approx 1$ for most modes but decaying the very highest frequencies a bit, providing stability without losing important signal. Finally, while our neural networks here were trained on a supervised dataset (of fine solutions or Parareal outcomes), one could also incorporate physics knowledge directly into the network architecture or loss (so-called physics-informed neural networks, PINNs). A PINN coarse propagator could, for example, enforce the exact advection of low-frequency components and only parametrize corrections, thereby guaranteeing correct phase speed for the bulk of modes. Recent research has indeed begun exploring PINNs as coarse propagators, showing that machine learning techniques can be integrated into parallel-in-time methods for improved performance. Our results complement this by demonstrating the importance of what aspect of "physics" (dispersion vs. diffusion) the coarse model gets right.

7.6 Conclusion

In this chapter, we analyzed the effectiveness of neural network coarse propagators in the Parareal algorithm for the 1D advection equation. We introduced metrics for phase speed and amplification, and showed that the neural coarse models better approximate the fine solver's dispersion relation than a traditional coarse integrator. Through Fourier analysis and numerical experiments, we found that coarse propagators which accurately convect waves (even if they have some dissipative error) enable much faster Parareal convergence than those which merely minimize pointwise error. Neural networks can be trained to fulfill this requirement by using tailored loss functions or training data that emphasize physically important features (like wave amplitude or phase alignment). Such networks drastically outperformed both the standard coarse solver and a naively-trained network in terms of reducing iteration count and error.
Overall, the case study demonstrates that neural networks offer a promising route to designing coarse integrators that are both fast (leveraging GPUs and parallelism) and effective. However, their training must be guided by insight into the algorithm's convergence properties. By aligning the neural coarse model's behavior with the fine solver in the ways that matter most (dispersion), we achieved a Parareal method that converges in fewer iterations, thus achieving greater speedup. This reinforces the idea that physics-informed machine learning and classical numerical algorithms can be combined to yield superior solvers, especially for challenging tasks like parallel-in-time integration of hyperbolic PDEs. The one-dimensional advection example provided a clear illustration, and future work will explore extending these strategies to more complex PDEs and higher-dimensional problems.

Chapter 8

Conclusion and Outlook

8.1 Summary of Contributions

This research has presented a novel approach to improve time-parallel numerical integration of PDEs by leveraging machine learning models as coarse propagators. The main contributions of this thesis are as follows:

- **PINN as a Coarse Propagator:** We conducted the first study using a Physics-Informed Neural Network as the coarse solver within the Parareal algorithm. By training a PINN to solve the Black-Scholes equation on coarse time intervals, we demonstrated significantly accelerated Parareal convergence for option pricing problems. The PINN-based coarse integrator reduced the number of iterations required and, when deployed on a GPU, yielded greater speedups compared to a traditional numerical coarse solver. This contribution showed that mesh-free, neural network coarse propagators can enhance parallel-in-time performance while naturally exploiting modern GPU hardware.

- **Multi-Asset PINO and Space-Time Parallelism:** We extended the machine-learning coarse propagator concept to a two-dimensional, two-asset Black-Scholes PDE using a Physics-Informed Neural Operator. This PINO-based coarse model was integrated into Parareal alongside spatial domain decomposition. The key finding is that Parareal with an ML-based coarse propagator can achieve scaling beyond the saturation point of spatial parallelization. In our experiments, a PINO coarse solver on a multi-asset option pricing problem not only provided speedup over the serial time-stepping baseline but also outperformed Parareal with a classical coarse solver, especially as the number of time subdomains increased. This is an important contribution to high-performance scientific computing: it demonstrates the feasibility of combining spatial and temporal parallelism (via Parareal) augmented with neural surrogates to solve larger problems faster. The approach paves the way for space-time parallel algorithms that fully utilize heterogeneous computing resources (CPUs and GPUs).

- **Theoretical Analysis of Coarse Propagators:** To understand why and when a neural coarse propagator improves Parareal, we performed a theoretical convergence analysis using a simple hyperbolic PDE (the one-dimensional advection equation) as a case study. We introduced metrics based on Fourier analysis – namely the phase speed and amplification factor of error modes – to evaluate coarse integrators. The analysis revealed that a coarse solver must accurately convect information (even if it introduces some dissipation) to avoid pathological Parareal behavior. Standard coarse schemes that minimize pointwise error but distort wave propagation can cause Parareal to diverge for transport-dominated problems. By contrast, coarse models that closely match the fine solver's dispersion relation enable fast, stable convergence. Guided by this insight, we trained neural networks with loss functions emphasizing phase accuracy. These networks, used as coarse propagators, dramatically outperformed both a naive neural network (trained only on pointwise error) and a traditional coarse solver in the advection test scenario. This contribution provides a new perspective on Parareal convergence: it underscores the importance of aligning coarse model behavior with the underlying physics of the PDE, and it demonstrates how machine learning models can be "physics-informed" to meet the requirements of parallel-in-time algorithms.

Collectively, the above contributions advance the state of the art in time-parallel numerical simulation. We have shown that blending machine learning with classical iterative solvers can overcome some long-standing limitations (such as poor convergence on hyperbolic problems and limited parallel scalability on diffusion-dominated problems) in a way that neither approach achieves alone. The focus on computational finance as a use case validates the methodology on practical PDE problems, but the techniques developed are general and can be transferred to other application domains. In summary, this thesis introduced a successful strategy to integrate neural network-based coarse propagators into Parareal, achieved notable speedups for both one-dimensional and two-dimensional Black-Scholes equations, and provided theoretical insights that generalize to improving parallel-in-time schemes for a broader class of PDEs.

8.2 Limitations

While the results are promising, it is important to acknowledge the limitations of this work. First, the experiments were primarily conducted on option pricing PDEs (Black-Scholes equations) which have well-behaved solutions. The PINN and PINO coarse models were tested on problems that are smooth and fairly low-dimensional (one and two spatial dimensions). The performance of the proposed approach on more complex or higher-dimensional PDEs (for example, fluid dynamics or weather simulation problems) remains unproven. In particular, nonlinear PDEs and systems with sharp discontinuities or shocks were not addressed in this thesis. Additional research is needed to determine if physics-informed neural coarse propagators can handle such challenges or if new net-

work architectures and training techniques would be required.
Second, although we conceptually explored parallel-in-time across multiple nodes, the actual strong-scaling tests were performed on a single workstation. The study showed that Parareal with a PINO coarse propagator can outperform space-only parallel solvers at the node level, but we did not demonstrate this in a large distributed-memory environment. Thus, one limitation is the lack of empirical evidence on multi-node scalability. We only utilized up to 16 CPU cores (for spatial parallelism) and one GPU in our tests. The overheads and communication patterns of the ML-based Parareal method on hundreds or thousands of processors (typical in supercomputing) were not examined due to resource constraints. This leaves some uncertainty as to how the method would scale in practice on exascale systems – for example, the impact of training or deploying neural networks across many nodes, and whether the coarse solver's GPU acceleration would effectively amortize communication costs at scale.
Another limitation involves the training process and generality of the neural coarse models. In our approach, the PINN and PINO were trained for a specific PDE and, in the case of the Black-Scholes equations, for a specific range of parameters (volatility, interest rate, etc.). We observed that the PINN/PINO-based Parareal algorithm generalizes moderately well to nearby parameters and grid resolutions (e.g., the PINO trained at one set of parameters remained effective for other parameter values within a reasonable range). However, this generalization has limits: when the problem characteristics deviate significantly from the training regime (for instance, extremely low volatility leading to an advection-dominated regime), the Parareal convergence deteriorated. In general, the neural network coarse propagator may need retraining or fine-tuning to maintain performance if the PDE or its parameters change substantially. This requirement can be a disadvantage in scenarios where we seek a "universal" coarse solver for many problem instances.
The offline training cost is also non-trivial – training PINNs or PINOs is time-consuming, and while it can be amortized over many runs, it adds a preprocessing step not present in classical solvers. Moreover, the success of training depends on careful hyperparameter choices and sufficient training data or residual evaluations; in some cases we had to tailor the loss function to achieve the desired accuracy in wave propagation. These practical considerations limit the ease of adoption of the method, as substantial expertise in both machine learning and numerical analysis is required to set up an effective coarse model.
Lastly, from a theoretical standpoint, there is a limitation in the rigorous convergence analysis for Parareal with nonlinear learned coarse propagators. Our Fourier analysis in Chapter 7 provided intuition by treating the coarse model's effect on different frequency components, but a formal convergence proof (especially for nonlinear or time-dependent coarse solvers like neural networks) is outside the scope of this thesis. Parareal's classical convergence theory assumes a contractive linear coarse operator or uses linear analysis; extending this to data-driven, adaptive coarse operators is not straightforward. As a result, the convergence guarantees for the proposed approach remain heuristic – we rely on empirical evidence and physical insight rather than a strict mathematical proof that

Parareal will converge for a given neural coarse model. This is a conceptual limitation that suggests caution: while our case studies did not encounter divergence (after appropriate training adjustments), one cannot a priori ensure stability for an arbitrary neural network coarse propagator without further theoretical development.

8.3 Future Work

The encouraging outcomes of this research open up several avenues for future investigation. One immediate direction is to apply the ML-augmented Parareal approach to other classes of PDEs and more complex models. For example, computational fluid dynamics problems (Navier–Stokes equations for airflow, shallow water equations for geophysical flows, etc.) could greatly benefit from time-parallelization, but designing effective coarse propagators is challenging due to advection-dominated dynamics. Exploring whether physics-informed neural operators can serve as robust coarse solvers in these contexts is a natural extension. This would likely involve integrating our insights on preserving wave propagation into the training of coarse models for fluid flow problems – possibly requiring networks that handle multiple spatial dimensions and complex boundary conditions. Multiscale and stiff systems (such as reaction-diffusion equations or weather/climate models) are another target for generalization, where the ability of neural networks to learn surrogate dynamics might help navigate scale disparities that hinder traditional Parareal coarse integrators.

Another future work direction is to achieve truly large-scale implementation and testing of Parareal with neural coarse propagators. This involves running on HPC systems with many GPUs and distributed memory, to verify the scalability of the method. Developing a version of the algorithm that can train and deploy neural network coarse models in parallel (across nodes) would be a significant engineering effort but is crucial for real-world usage. Techniques like model parallelism or distributed training could be employed so that the coarse model itself can be trained on large clusters for high-dimensional problems. Additionally, one could investigate hybrid parallelization strategies: for instance, using multiple GPUs such that some handle fine solves on subintervals while others evaluate the coarse model for different intervals concurrently. The overhead analysis for such configurations would inform how to balance resources between fine and coarse propagators optimally. Early results in this thesis suggest diminishing returns after a point with spatial parallelism alone; thus, future research should quantify the combined space-time parallel efficiency on architectures with thousands of cores and multiple GPUs, possibly using advanced time-parallel frameworks like PFASST in conjunction with learned coarse operators.

From a machine learning perspective, future work should also consider improving the training and robustness of neural coarse propagators. One idea is to incorporate more physics-informed constraints or architectures: for example, ensuring that the neural network coarse solver exactly preserves certain invariants or symmetries of the PDE. In the context of option pricing, one could enforce no-arbitrage conditions or known solution asymptotics within the network. For more general PDEs, techniques from physics-

informed ML (e.g., hard constraints, variational losses) could further improve reliability. Another idea is to use transfer learning or meta-learning such that a coarse model trained on one scenario can be rapidly adapted to new scenarios, mitigating the retraining cost. We found that our PINO could generalize to modest parameter changes without retraining; building on this, one could train a single neural operator on a wide range of PDE parameter configurations (e.g., a range of volatilities and rates for Black–Scholes, or a family of related PDEs) so that it becomes a universal coarse solver within that family. This would improve the practicality of deploying ML-based Parareal in environments where problem parameters vary.

Finally, there is room for theoretical advancements to complement empirical developments. Extending convergence theory to cover adaptive and non-linear coarse propagators would provide deeper understanding and confidence in the method. For instance, one might attempt to derive bounds on Parareal error convergence given assumptions on the neural network approximation error or its spectral properties (as hinted by our Fourier analysis). Developing such theory could involve bridging numerical analysis with learning theory – possibly yielding criteria on network capacity or training accuracy required to guarantee a certain speed of Parareal convergence. Additionally, investigating the stability of Parareal when the coarse model is periodically retrained or updated online (i.e., online learning during the iterations) could be valuable. This would combine concepts from iterative solvers and adaptive control to ensure that any update to the coarse model does not destabilize the iteration. Although this thesis has focused on offline-trained models, an online adaptive coarse solver could adjust to solution changes on the fly, potentially improving efficiency for non-stationary or long-time integration problems.

In summary, the outlook is that marrying machine learning with parallel-in-time integration is a fertile ground for research. By addressing the current limitations and pursuing the directions outlined above, future work can generalize our approach to a broader array of problems, improve its scalability and automation, and deepen the theoretical understanding of these hybrid algorithms. The ultimate vision is to enable extreme-scale simulations of complex systems by breaking the time barrier: using intelligent coarse models that learn from physics and data to unlock unprecedented parallel speedups for time-dependent simulations in science, engineering, and finance.

Bibliography

[1] J. Rosemeier, T. Haut, and B. Wingate, "Multi-level parareal algorithm with averaging," *arXiv preprint arXiv:2211.17239*, 2022. math.NA.

[2] J. Strake, D. Döhring, and A. Benigni, "Mgrit-based multi-level parallel-in-time electromagnetic transient simulation," *Energies*, vol. 15, p. 7874, Oct. 2022.

[3] Y. Lutsyshyn, F. Navarrete, and D. Bauer, "The n-shaped partition method: A novel parallel implementation of the crank–nicolson algorithm," *arXiv preprint arXiv:2205.00856*, 2022. GPU + MPI, implicit PDEs.

[4] G. Čaklović, T. Lunet, S. Götschel, and D. Ruprecht, "Improving efficiency of parallel across the method spectral deferred corrections," *SIAM Journal on Scientific Computing*, vol. 47, p. A430–A453, Feb. 2025.

[5] M. J. Gander and D. Palitta, "A new paradiag time-parallel time integration method," *SIAM Journal on Scientific Computing*, vol. 46, p. A697–A718, Mar. 2024.

[6] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. L. Minion, M. Winkel, and P. Gibbon, "A massively space-time parallel N-body solver," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 92:1–92:11, IEEE Computer Society Press, 2012.

[7] J.-L. Lions, Y. Maday, and G. Turinici, "A "parareal" in time discretization of PDE's," *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, vol. 332, pp. 661–668, 2001.

[8] M. Emmett and M. Minion, "Toward an efficient parallel in time method for partial differential equations," *Communications in Applied Mathematics and Computational Science*, vol. 7, no. 1, pp. 105–132, 2012.

[9] R. D. Falgout, S. Friedhoff, T. V. Kolev, S. P. MacLachlan, and J. B. Schroder, "Parallel time integration with multigrid," *SIAM Journal on Scientific Computing*, vol. 36, p. C635–C661, Jan. 2014.

[10] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012.

[11] A. Q. Ibrahim, S. Götschel, and D. Ruprecht, "Space-time parallel scaling of parareal with a physics-informed fourier neural operator coarse propagator applied to the black-scholes equation," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '25, (New York, NY, USA), p. 1–11, Association for Computing Machinery, 2025.

[12] P. Constantin and C. Foiaş, *Navier-stokes equations*. University of Chicago press, 1988.

[13] F. A. Berezin and M. Shubin, *The Schrödinger Equation*, vol. 66. Springer Science & Business Media, 2012.

[14] G. Barles and H. M. Soner, "Option pricing with transaction costs and a nonlinear black-scholes equation," *Finance and Stochastics*, vol. 2, pp. 369–397, 1998.

[15] I. G. Kevrekidis, C. W. Gear, J. M. Hyman, P. G. Kevrekidis, O. Runborg, and C. Theodoropoulos, "Equation-free, coarse-grained multiscale computation: Enabling microscopic simulators to perform system-level analysis," *Communications in Mathematical Sciences*, vol. 1, no. 4, pp. 715–762, 2003.

[16] W. E and B. Engquist, "Heterogeneous multiscale methods: A review," *Communications in Computational Physics*, vol. 2, no. 3, pp. 367–450, 2003.

[17] A. J. Majda, I. Timofeyev, and E. Vanden-Eijnden, "Systematic strategies for stochastic mode reduction in climate," *Journal of the Atmospheric Sciences*, vol. 60, no. 14, pp. 1705–1722, 2003.

[18] T. J. Hughes, *The finite element method: linear static and dynamic finite element analysis*. Dover Publications, 2000.

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

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

[21] A. Quarteroni, *Numerical Models for Differential Problems*. Springer Milan, 2014.

[22] J. Shadid, R. Pawlowski, E. Cyr, R. Tuminaro, L. Chacón, and P. Weber, "Scalable implicit incompressible resistive mhd with stabilized fe and fully-coupled newton–krylov-amg," *Computer Methods in Applied Mechanics and Engineering*, vol. 304, p. 1–25, June 2016.

[23] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, *et al.*, "Multiphysics simulations: Challenges and opportunities," *The International Journal of High Performance Computing Applications*, vol. 27, no. 1, pp. 4–83, 2013.

[24] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, *et al.*, "The international exascale software project roadmap," *The International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[25] L. Grinberg, D. Pekurovsky, S. J. Sherwin, and G. E. Karniadakis, "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2010.

[26] *Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019, Bialystok, Poland, September 8–11, 2019, Revised Selected Papers, Part II*. Springer International Publishing, 2020.

[27] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[28] B. F. Smith, P. E. Bjorstad, and W. D. Gropp, *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, 1996.

[29] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling algebraic multigrid solvers: On the road to exascale," *Competence in High Performance Computing 2010*, pp. 215–226, 2012.

[30] M. J. Gander, "50 years of time parallel time integration," *Multiple Shooting and Time Domain Decomposition Methods*, pp. 69–113, 2015.

[31] M. J. Gander and S. Vandewalle, "Analysis of the parareal time-parallel time-integration method," *SIAM Journal on Scientific Computing*, vol. 29, no. 2, pp. 556–578, 2013.

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

[33] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, 2021.

[34] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems," *Proceedings of the National Academy of Sciences*, vol. 113, no. 15, pp. 3932–3937, 2016.

[35] Y. Bar-Sinai, S. Hoyer, J. Hickey, and M. P. Brenner, "Learning data-driven discretizations for partial differential equations," *Proceedings of the National Academy of Sciences*, vol. 116, no. 31, pp. 15344–15349, 2019.

[36] C. Michoski, M. Milosavljević, T. Oliver, and D. R. Hatch, "Solving differential equations using deep neural networks," *Neurocomputing*, vol. 399, p. 193–212, July 2020.

[37] J. W. Demmel, *Applied numerical linear algebra*. SIAM, 1997.

[38] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. part ii: Implementation and tests in dune," *Computing*, vol. 82, no. 2-3, pp. 121–138, 2008.

[39] M. Emmett and M. Minion, "PFASST: Parallel full approximation scheme in space and time," *Journal of Computational Physics*, vol. 231, no. 21, pp. 7191–7213, 2012.

[40] A. Q. Ibrahim, S. Götschel, and D. Ruprecht, "Parareal with a physics-informed neural network as coarse propagator," *arXiv preprint arXiv:2307.07094*, 2023.

[41] V. Dolean, P. Jolivet, and F. Nataf, *An introduction to domain decomposition methods: algorithms, theory, and parallel implementation*, vol. 144. SIAM, 2015.

[42] A. Toselli and O. Widlund, *Domain decomposition methods-algorithms and theory*, vol. 34. Springer, 2005.

[43] T. F. Chan and T. P. Mathew, "Domain decomposition algorithms," *Acta numerica*, vol. 3, pp. 61–143, 1994.

[44] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[45] G. Karypis, K. Schloegel, and V. Kumar, "ParMETIS: Parallel graph partitioning and sparse matrix ordering library," *University of Minnesota, Department of Computer Science and Engineering, Technical Report*, 1998.

[46] P. J. Winzer, "The future of communications is massively parallel," *Journal of Optical Communications and Networking*, vol. 15, no. 10, pp. 783–787, 2023.

[47] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[48] M. J. Gander and S. Vandewalle, "Analysis of the parareal time-parallel time-integration method," *SIAM Journal on Scientific Computing*, vol. 29, p. 556–578, Jan. 2007.

[49] M. L. Minion, "A hybrid parareal spectral deferred corrections method," *Communications in Applied Mathematics and Computational Science*, vol. 5, no. 2, pp. 265–301, 2011.

[50] G. Bal, "On the convergence and the stability of the parareal algorithm to solve partial differential equations," *Domain decomposition methods in science and engineering*, pp. 425–432, 2005.

[51] S. Friedhoff and S. MacLachlan, "Parallel-in-time integration with multigrid," *SIAM Journal on Scientific Computing*, vol. 43, no. 2, pp. A828–A850, 2013.

[52] M. J. Gander and T. Lunet, *Time Parallel Time Integration*. Society for Industrial and Applied Mathematics, Jan. 2024.

[53] G. A. Staff and E. M. Rønquist, "Convergence and stability of the parareal algorithm: A numerical and theoretical investigation," 2005.

[54] D. Ruprecht, "Convergence of parareal with spatial coarsening," *PAMM*, vol. 14, no. 1, pp. 1031–1034, 2014.

[55] J. N. Kutz, "Deep learning in fluid dynamics," *Journal of Fluid Mechanics*, vol. 814, pp. 1–4, 2017.

[56] E. O. Oluwasakin, A. Q. Khaliq, and K. M. Furati, "Learning time-varying parameters of stiff dynamical systems using physics-informed transfer neural network," *Mathematics and Computers in Simulation*, vol. 238, p. 82–102, Dec. 2025.

[57] X. Meng, Z. Li, D. Zhang, and G. E. Karniadakis, "Ppinns: Parareal physics-informed neural networks for time-dependent pdes," *Computer Methods in Applied Mechanics and Engineering*, vol. 370, p. 113250, 2020.

[58] L. Lu, P. Jin, and G. E. Karniadakis, "Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators," *arXiv preprint arXiv:1910.03193*, 2019.

[59] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, "Accelerating eulerian fluid simulation with convolutional networks," *Proceedings of the 34th International Conference on Machine Learning*, vol. 70, pp. 3424–3433, 2017.

[60] A. Klawonn, M. Lanser, and J. Weber, "Machine learning and domain decomposition methods - a survey," *Computational Science and Engineering*, vol. 1, Sept. 2024.

[61] B. Lusch, J. N. Kutz, and S. L. Brunton, "Deep learning for universal linear embeddings of nonlinear dynamics," *Nature Communications*, vol. 9, no. 1, pp. 1–10, 2018.

[62] O. San and R. Maulik, "Neural network closure models for nonlinear reduced order models of multiscale systems," *Advances in Computational Mathematics*, vol. 44, no. 6, pp. 1717–1750, 2018.

[63] J. Pathak, B. Hunt, M. Girvan, Z. Lu, and E. Ott, "Model-free prediction of large spatiotemporally chaotic systems from data: A reservoir computing approach," *Physical Review Letters*, vol. 120, no. 2, p. 024102, 2018.

[64] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, *et al.*, "Exascale deep learning for climate analytics," *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pp. 1–12, 2018.

[65] P. R. Vlachas, W. Byeon, Z. Y. Wan, T. P. Sapsis, and P. Koumoutsakos, "Data-driven forecasting of high-dimensional chaotic systems with long short-term memory networks," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 474, no. 2213, p. 20170844, 2018.

[66] J. Han, A. Jentzen, and W. E, "Solving high-dimensional partial differential equations using deep learning," *Proceedings of the National Academy of Sciences*, vol. 115, no. 34, pp. 8505–8510, 2018.

[67] W. Zhu, V. Francois-Lavet, X. Xie, and Y. Liu, "Convolutional neural networks for surrogate modeling of two-dimensional mantle convection," *Journal of Computational Physics*, vol. 409, p. 109236, 2019.

[68] S. Wang, X. Yu, and P. Perdikaris, "Respecting causality is all you need for training physics-informed neural networks," *arXiv preprint arXiv:2011.03698*, 2020.

[69] Z. Y. Wan and T. P. Sapsis, "Long-time prediction of nonlinear dynamics by data-driven reduced-order modeling and machine learning approaches," *Bulletin of the American Physical Society*, vol. 63, 2018.

[70] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[71] B. W. Ong and J. B. Schroder, "Applications of time parallelization," *Computing and Visualization in Science*, vol. 23, Sept. 2020.

[72] M. Gattiglio, M. Witte, and G. E. Karniadakis, "Randnet-parareal: Parallel-in-time coarse correction with random neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[73] K. Pentland, M. Tamborrino, T. J. Sullivan, J. Buchanan, and L. C. Appel, "Gparareal: a time-parallel ode solver using gaussian process emulation," *Statistics and Computing*, vol. 33, Dec. 2022.

[74] A. Chattopadhyay, P. Hassanzadeh, and D. Subramanian, "Data-driven prediction of a multi-scale lorenz 96 chaotic system using a hybrid neural network: Deep learning, model error and prediction uncertainty," *Nonlinear Processes in Geophysics*, vol. 27, no. 3, pp. 373–389, 2020.

[75] S. Scher, "Toward data-driven weather and climate forecasting: Approximating a simple general circulation model with deep learning," *Geophysical Research Letters*, vol. 45, no. 22, pp. 12616–12622, 2018.

[76] W. L. Tan, S. Roberts, and S. Zohren, "Deep learning for options trading: An end-to-end approach," in *Proceedings of the 5th ACM International Conference on AI in Finance*, ICAIF '24, p. 487–495, ACM, Nov. 2024.

[77] L. Yang, X. Meng, and G. E. Karniadakis, "B-pinns: Bayesian physics-informed neural networks for forward and inverse pde problems with noisy data," *Journal of Computational Physics*, vol. 425, p. 109913, 2021.

[78] S. Wang, Y. Teng, and P. Perdikaris, "When and why pinns fail to train: A neural tangent kernel perspective," *Journal of Computational Physics*, vol. 449, p. 110768, 2021.

[79] J.-L. Wu, K. Kashinath, A. Albert, D. Chirila, M. Prabhat, and H. Xiao, "Enforcing statistical constraints in generative adversarial networks for modeling chaotic dynamical systems," *Journal of Computational Physics*, vol. 406, p. 109209, 2020.

[80] S. Rasp, M. S. Pritchard, and P. Gentine, "Deep learning to represent subgrid processes in climate models," *Proceedings of the National Academy of Sciences*, vol. 115, no. 39, pp. 9684–9689, 2018.

[81] K. Shukla, A. D. Jagtap, and G. E. Karniadakis, "Parallel physics-informed neural networks via domain decomposition," *arXiv preprint arXiv:2011.11572*, 2021.

[82] A. Barman, B. Khara, B. Ganapathysubramanian, and A. Sharma, "Accelerating space-time methods using physics-informed neural networks," *Journal of Computational Physics*, vol. 537, p. 114124, 2025.

[83] Y. Maday and G. Turinici, "The parareal in time iterative solver: a further direction to parallel implementation," *Domain Decomposition Methods in Science and Engineering*, vol. 60, pp. 441–448, 2007.

[84] M. J. Gander and S. Vandewalle, "Analysis of the parareal time-parallel time-integration method," *SIAM Journal on Scientific Computing*, vol. 29, p. 556–578, Jan. 2007.

[85] E. Scheiber, "A convergence theorem for the parareal algorithm revisited," *arXiv preprint arXiv:2405.06954*, 2024.

[86] B. S. Southworth, H. De Sterck, and K. Pearson, "Tight two-level convergence of linear parareal and mgrit," *arXiv preprint arXiv:2010.11879*, 2020.

[87] B. Song, J.-Y. Wang, and Y.-L. Jiang, "Analysis of a new krylov subspace enhanced parareal algorithm for time-periodic problems," *Numerical Algorithms*, vol. 97, p. 289–310, Nov. 2023.

[88] A. Quarteroni, A. Manzoni, and F. Negri, *Reduced Basis Methods for Partial Differential Equations*. Springer International Publishing, 2016.

[89] B. Carrel, M. J. Gander, and B. Vandereycken, "Low-rank parareal: a low-rank parallel-in-time integrator," *BIT Numerical Mathematics*, vol. 63, Feb. 2023.

[90] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "Sparcml: High-performance sparse communication for machine learning," *arXiv preprint arXiv:1802.08021*, 2018.

[91] S. Götschel and M. Weiser, "Compression challenges in large scale partial differential equation solvers," *Algorithms*, vol. 12, p. 197, Sept. 2019.

[92] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. M. Stuart, and A. Anandkumar, "Fourier neural operator for parametric partial differential equations," *International Conference on Learning Representations (ICLR)*, 2021.

[93] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, "Learning physics-informed neural operators for forward and inverse pde problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 393, p. 114823, 2022.

[94] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814, 2010.

[95] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[96] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.

[97] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[98] C. Trahan, M. Loveland, and S. Dent, "Quantum physics-informed neural networks," *Entropy*, vol. 26, p. 649, July 2024.

[99] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit neural representations with periodic activation functions," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 7462–7473, 2020.

[100] M. A. Nabian and H. Meidani, "Efficient training of physics-informed neural networks via importance sampling," *Computer Methods in Applied Mechanics and Engineering*, vol. 384, p. 113933, 2021.

[101] C. Wight and L. Zhao, "Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks," *arXiv preprint arXiv:2007.04542*, 2020.

[102] L. McClenny and U. Braga-Neto, "Self-adaptive physics-informed neural networks using a soft attention mechanism," *arXiv preprint arXiv:2009.04544*, 2020.

[103] S. Wang, Y. Teng, and P. Perdikaris, "Understanding and mitigating gradient pathologies in physics-informed neural networks," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. A3055–A3081, 2021.

[104] X.-l. Luo, J.-h. Lv, and H. Xiao, "Explicit continuation methods with l-bfgs updating formulas for linearly constrained optimization problems," 2021.

[105] A. Krishnapriyan, A. Gholami, S. Zhe, R. M. Kirby, and M. W. Mahoney, "Characterizing possible failure modes in physics-informed neural networks," *Advances in Neural Information Processing Systems*, vol. 34, pp. 26548–26560, 2021.

[106] A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis, "Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations," *Advances in Neural Information Processing Systems*, vol. 33, pp. 12759–12771, 2020.

[107] A. D. Jagtap, K. Kawaguchi, and G. E. Karniadakis, "Adaptive activation functions accelerate convergence in deep and physics-informed neural networks," *Journal of Computational Physics*, vol. 404, p. 109136, 2020.

[108] Z. Su, Y. Liu, S. Pan, Z. Li, and C. Shen, "Finite volume physical informed neural network (fv-pinn) with reduced derivative order for incompressible flows," 2024.

[109] S. Mishra, "Estimates on the generalization error of physics-informed neural networks for approximating pdes," *IMA Journal of Numerical Analysis*, vol. 42, no. 2, pp. 981–1022, 2022.

[110] Y. Cao and Q. Gu, "Generalization error bounds of gradient descent for learning over-parameterized deep relu networks," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[111] S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis, "Physics-informed neural networks (pinns) for fluid mechanics: A review," *Acta Mechanica Sinica*, vol. 38, no. 6, pp. 1727–1738, 2022.

[112] N. B. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. M. Stuart, and A. Anandkumar, "Neural operator: Learning maps between function spaces," *arXiv preprint arXiv:2108.08481*, 2021.

[113] P. Clark Di Leoni, L. Lu, C. Meneveau, G. E. Karniadakis, and T. A. Zaki, "Neural operator prediction of linear instability waves in high-speed boundary layers," *Journal of Computational Physics*, vol. 474, p. 111793, Feb. 2023.

[114] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, "Learning nonlinear operators via deeponet based on the universal approximation theorem of operators," *Nature Machine Intelligence*, vol. 3, p. 218–229, Mar. 2021.

[115] S. Lanthaler, S. Mishra, and G. E. Karniadakis, "Error estimates for deeponets: A deep learning framework in infinite dimensions," *SIAM Journal on Mathematics of Data Science*, vol. 4, no. 1, pp. 491–517, 2022.

[116] T. D. Ryck and S. Mishra, "Error analysis for physics informed neural networks (pinns) approximating kolmogorov pdes," 2021.

[117] S. Lanthaler, A. M. Stuart, and L. H. Trautner, "Discretization error of fourier neural operators," *arXiv preprint arXiv:2405.02221*, 2024.

[118] T. De Ryck and S. Mishra, "Generic bounds on the approximation error for physics-informed (and) operator learning," *arXiv preprint arXiv:2205.11393*, 2022. Provides operator learning bounds including physics-informed operators.

[119] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.

[120] R. C. Merton, "Theory of rational option pricing," *The Bell Journal of Economics and Management Science*, vol. 4, no. 1, pp. 141–183, 1973.

[121] F. A. Longstaff and E. S. Schwartz, "Valuing american options by simulation: A simple least-squares approach," *The Review of Financial Studies*, vol. 14, no. 1, pp. 113–147, 2001.

[122] J. C. Hull, *Options, Futures, and Other Derivatives*. Pearson, 9th ed., 2017.

[123] J. C. Cox, S. A. Ross, and M. Rubinstein, "Option pricing: A simplified approach," *Journal of Financial Economics*, vol. 7, no. 3, pp. 229–263, 1979.

[124] L. H. Thomas, "Elliptic problems in linear difference equations over a network," *Watson Sci. Comput. Lab Report*, vol. 2, pp. 1–22, 1949.

[125] D. Tavella and C. Randall, *Pricing Financial Instruments: The Finite Difference Method*. Wiley, 2000.

[126] D. J. Duffy, *Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach*. Wiley, 2006.

[127] W. Margrabe, "The value of an option to exchange one asset for another," *The Journal of Finance*, vol. 33, no. 1, pp. 177–186, 1978.

[128] E. Kirk, "Correlation in the energy markets," in *Managing Energy Price Risk* (R. Jameson, ed.), pp. 71–78, London: Risk Publications and Enron Capital & Trade Resources, 1995.

[129] M. B. Garman and S. W. Kohlhagen, "Foreign currency option values," *Journal of International Money and Finance*, vol. 8, no. 2, pp. 231–237, 1989.

[130] S. Kim, D. Jeong, C. Lee, and J. Kim, "Finite difference method for the multi-asset black–scholes equations," *Mathematics*, vol. 8, no. 3, 2020.

[131] A. Genz, "Numerical computation of rectangular bivariate and trivariate normal and t probabilities," *Statistics and Computing*, vol. 14, p. 251–260, aug 2004.

[132] A. Q. Ibrahim, S. Götschel, and D. Ruprecht, "Parareal with a physics-informed neural network as coarse propagator," in *Euro-Par 2023: Parallel Processing*, pp. 649–663, Springer Nature Switzerland, 2023.

[133] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Marchine Learning Research*, vol. 18, pp. 1–43, 2018.

[134] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[135] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[136] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," *Proceedings of the 14th Python in Science Conference*, pp. 130–136, 2015.

[137] T. J. Grady, R. Khan, M. Louboutin, Z. Yin, P. A. Witte, R. Chandra, R. J. Hewett, and F. J. Herrmann, "Model-parallel Fourier neural operators as learned surrogates for large-scale parametric PDEs," *Computers & Geosciences*, vol. 178, p. 105402, 2023.

[138] Y. Maday and G. Turinici, "A parareal in time procedure for the control of partial differential equations," *Comptes Rendus. Mathématique*, vol. 335, p. 387–392, Jan. 2002.

[139] B. Jin, Q. Lin, and Z. Zhou, "Optimizing coarse propagators in parareal algorithms," *SIAM Journal on Scientific Computing*, vol. 47, p. A735–A761, Mar. 2025.

[140] S. R. Vadyala, S. N. Betgeri, and N. P. Betgeri, "Physics-informed neural network method for solving one-dimensional advection equation using pytorch," *Array*, vol. 13, p. 100110, Mar. 2022.