PINNs and Needles: Comparing Physics-Informed Neural Networks and the Finite Element Method in Classical and Quantum Systems

Written by:

Odin Johansen

Department of Physics UiO

UiO : University of Oslo

May 13, 2025

Abstract

Acknowledgment

I would like to extend my deepest gratitude to my supervisor, Morten, for his invaluable guidance, insightful discussions, and continuous support throughout the development of this thesis. His expertise and encouragement have been instrumental in shaping my work. A heartfelt thank you to Edvin, Sigurd, and Felix for their unwavering support, constructive feedback, and countless hours spent proofreading and discussing ideas with me. Your input has been invaluable, and your friendship has made this journey all the more enjoyable. Finally, I am grateful to everyone who has supported me, directly or indirectly, throughout this process. Your encouragement and belief in me have meant the world.

Contents

Chapter 1

1. Introduction

Quantum mechanics unveils a world profoundly different from classical intuition, where particles exist in superpositions and distant entities can be instantaneously linked through entanglement. At the heart of this perplexing domain of modern physics lies the many-body problem: the challenge of describing systems with numerous interacting particles. This task's complexity escalates exponentially with particle number, rapidly overwhelming the capabilities of exact analytical solutions and pushing the boundaries of traditional computational methods. The quest for accurate and efficient solutions to the many-body problem is not merely an academic pursuit; it underpins our understanding of materials, chemical reactions, and the very fabric of matter, holding the key to innovations in fields from drug discovery to quantum computing. In recent years, the burgeoning field of machine learning (ML), combined with advances in computational physics, has offered a promising new frontier for tackling this inherent complexity. ML algorithms, particularly neural networks, are adept at uncovering hidden patterns within high-dimensional data. A specialized class, Physics-Informed Neural Networks (PINNs), has garnered significant attention by directly embedding the fundamental physical laws, typically expressed as differential equations, into the learning process itself. This synergy allows PINNs to potentially transcend the limitations of purely data-driven approaches or traditional numerical schemes, offering a novel paradigm for modeling intricate quantum systems.

> "Quantum mechanics is certainly imposing. But an inner voice tells me that it is not yet the real thing. The theory says a lot, but does not really bring us any closer to the secret of the "old one." I, at any rate, am convinced that He does not throw dice.
>
> Albert Einstein, *The Born-Einstein Letters 1916-55*"

i. The Many-body problem

The many-body problem is a fundamental part of quantum mechanics and condensed matter physics. It encapsulates the challenge of predicting the collective behavior of systems comprising multiple interacting particles, such as electrons in atoms, molecules, or novel materials. While the underlying Schrödinger equation governs these systems, its exact analytical solution becomes intractable for all but the simplest cases due to the exponential growth in complexity with an increasing number of particles. This difficulty stems from the intricate nature of the many-body wave function, which must simultaneously encode particle interactions and quantum statistical requirements, like the Pauli exclusion principle for fermions. Traditional methods such as Hartree-Fock theory, configuration interaction, and quantum Monte Carlo offer valuable approximations but often rely on domain-specific ansatzes or face their own scaling limitations. Modern advancements, however, are paving the way for more flexible numerical tools. Among these, machine learning techniques, particularly neural network representations of wave functions, show remarkable promise. By enabling models to learn the wave function's structure from fundamental principles or minimal data, these methods offer a new avenue for tackling the many-body problem, especially in complex or strongly correlated regimes where a priori physical insight is limited.

ii. Machine learning and Physics-Informed Neural Networks

Machine learning (ML), particularly neural network-based techniques, has emerged as a computational tool for addressing complex physics problems, offering capabilities beyond traditional numerical approaches. Recently, Physics-Informed Neural Networks (PINNs) have attracted considerable attention due to their unique approach:

they directly embed physical laws, typically expressed as differential equations, into the neural network's learning process. Instead of relying solely on labeled data, PINNs are optimized to satisfy these governing equations and associated initial and boundary conditions. This fusion of ML's predictive power with the rigorous constraints of physical theory offers a compelling approach to scientific simulation. In this thesis, PINNs are specifically applied to solve a range of classical and quantum mechanical many-body problems. The many-body Schrödinger equation serves as the core physical constraint within the PINN framework. This allows the neural network to approximate wavefunctions without explicit discretization meshes or predefined basis expansions, a feature that holds significant potential advantages for high-dimensional problems where conventional methods often encounter computational bottlenecks or struggle with the sheer complexity of the solution space. The mesh-free nature and inherent differentiability of neural networks make PINNs particularly suited for exploring such challenging domains. A central objective of this thesis is to systematically assess the performance of PINNs in direct comparison to the Finite Element Method (FEM), a well-established and robust numerical technique. By benchmarking PINN-generated solutions against FEM results for identical physical systems, this study aims to evaluate the relative strengths and limitations of this physics-informed machine learning approach in terms of computational efficiency, accuracy, and generalization. Such a comparative analysis is crucial for understanding the practical viability of PINNs as an alternative or complementary tool to traditional numerical methods, particularly in the demanding realm of quantum many-body physics.

iii. The Finite Element Method

The Finite Element Method (FEM) is widely employed for solving differential equations across physics and engineering. By discretizing a complex domain into a mesh of smaller, simpler elements, FEM transforms differential equations into a system of algebraic equations amenable to computational solutions. Its strength lies in its flexibility in handling irregular geometries, complex boundary conditions, and varying material properties, making it particularly advantageous for problems with intricate spatial configurations. In the context of quantum mechanical systems, such as the quantum dots explored within this thesis, FEM's ability to precisely discretize the quantum domain is invaluable for resolving wavefunctions and energy states, especially within highly localized potential landscapes. The method's adaptability in refining the mesh allows for high accuracy in regions with sharp potential wells or steep gradients. However, despite its versatility, FEM faces challenges. Its computational cost can become prohibitive for higher-dimensional systems due to the rapid growth in the number of elements and degrees of freedom. Furthermore, certain non-linearities inherent in quantum interactions can pose difficulties for standard FEM formulations, potentially requiring specialized and more complex solution strategies. These limitations motivate the exploration of alternative methods like PINNs.

iv. The Challenge of Quantum Systems

Quantum systems exhibit behaviors that starkly contrast with classical intuition, characterized by phenomena such as wave-particle duality, quantized energy levels, tunneling, and entanglement. Accurately describing and predicting these behaviors hinges on solving quantum mechanical equations, most notably the Schrödinger equation. While analytical solutions exist for highly idealized systems like the hydrogen atom or the simple harmonic oscillator, realistic and complex quantum structures necessitate powerful numerical methods. The primary challenges in numerically solving quantum systems arise from their intrinsic complexity and the high dimensionality of their configuration spaces, especially for systems involving many interacting particles or intricate potential landscapes. This complexity demands innovative computational techniques that can effectively balance accuracy and efficiency. This thesis focuses on applying and comparing two such numerical approaches—the established Finite Element Method (FEM) and the emerging Physics-Informed Neural Networks (PINNs), to efficiently capture quantum behaviors without undue computational burden.

v. Quantum Dots: Nanoscale Testbeds

Quantum dots (QDs) are nanoscale semiconductor structures that, due to quantum confinement effects, exhibit discrete, atom-like energy spectra. Often dubbed "artificial atoms," QDs possess unique and highly tunable electronic and optical properties, adjustable by modifying their size, shape, composition, or external fields. Their potential applications in quantum computing, photovoltaics, biological imaging, and optoelectronics have fueled significant research interest. Accurately predicting the physical properties of QDs—their energy states, electron wavefunctions, and responses to external fields—requires solving quantum mechanical equations under conditions of spatial confinement and complex boundary conditions. Given their quantum nature and nanoscale dimensions, analytical solutions are generally inaccessible. This necessitates advanced numerical methods. The Finite Element Method (FEM), with its proven versatility in handling intricate geometries, has been a valuable tool. However, the unique characteristics of QDs, often involving sharp potentials and strong correlation effects, also make them compelling and challenging testbeds for novel computational approaches like PINNs, allowing for a direct assessment of these methods' capabilities in a physically relevant context.

vi. The Role of Numerical Experiments

The advent of computational resources has revolutionized scientific discovery, enabling the investigation of complex physical systems through numerical experiments. Complementing traditional analytical techniques and laboratory experiments, numerical simulations provide an indispensable tool for exploring phenomena that are otherwise inaccessible. These simulations, grounded in fundamental physical laws, allow for precise studies and predictions, even for systems where experimental or analytical approaches are prohibitively complex or constrained. In the study of quantum systems, and specifically quantum dots, methods like FEM have proven effective in approximating quantum states and energy levels. However, as system complexity increases, with more particles or higher dimensions, conventional FEM simulations can become computationally intensive. To systematically evaluate the performance, accuracy, and scalability of different numerical methods, structured numerical experiments are essential. Through carefully designed computational studies, this thesis assesses the precision and efficiency of both FEM and PINNs in capturing critical quantum phenomena, providing benchmarks against known analytical solutions or highly accurate reference data, thereby illuminating the practical strengths and weaknesses of each approach.

vii. Goals and Contributions

The overarching objective of this thesis is to investigate and demonstrate the effectiveness of Physics-Informed Neural Networks (PINNs) in solving classical and quantum mechanical systems, thereby providing new insights into their computational advantages and limitations compared to established techniques. PINNs integrate physical laws, encoded as differential equation constraints, directly with machine learning algorithms. Our research specifically targets the application of PINNs to challenging classical dynamical systems and quantum mechanical problems, where accurate solutions are often hampered by complexity and high computational cost. The contributions of this work can be summarized as follows: Methodological Development: We develop and implement a robust computational framework using PINNs capable of addressing both classical and quantum mechanical differential equations. This includes optimizing network architectures and training methodologies tailored specifically for the quantum-mechanical Schrödinger equation, ensuring compliance with fundamental physical principles such as boundary conditions and normalization constraints. Evaluation and Validation: We benchmark the performance of PINNs against traditional numerical methods, primarily the Finite Element Method, in terms of computational cost, solution accuracy, and operational efficiency. The stability and convergence properties of PINNs are investigated across a spectrum of increasingly complex systems, from simple classical harmonic oscillators to interacting quantum dots. Applications and Insights: We solve representative classical

and quantum mechanical systems using the developed PINN framework, demonstrating its practical applicability and potential for reducing computational resources compared to conventional methods. Furthermore, we analyze the learned representations and network behaviors to understand how PINNs capture underlying physics, offering deeper insights into the neural network's ability to generalize solutions to unseen conditions. Through these contributions, this thesis aims to bridge the gap between traditional computational physics methods and emerging machine learning techniques. It seeks to demonstrate that PINNs can serve as a valuable, and in some cases superior, tool for solving complex physical systems, thus paving the way for future research at the exciting intersection of computational physics and artificial intelligence.

viii. Structure of the Thesis

This thesis is organized into six main parts, each addressing distinct theoretical and practical components of the work. The introductory section 1 outlines the scope and context, covering the many-body problem, essential machine learning concepts, the finite element method, and the quantum systems under investigation, including a particular focus on quantum dots. It also presents the numerical experiments, the primary goals and contributions of the study, and an overview of the code development process. Part II introduces the theoretical underpinnings of classical and quantum mechanics, with particular emphasis on the foundational postulates, the variational principle, and many-body quantum mechanics, including electron correlation and Wigner crystal formation. Part III covers the relevant numerical methods, including physics-informed neural networks (PINNs) and the finite element method, with detailed discussion of loss function construction, activation functions, and optimization strategies. The methodology is laid out in Part IV, followed by Part V, which presents the technical implementation, focusing on code structure, modularity, and computational considerations. Part VI contains the results and their interpretation in relation to the theoretical expectations. Finally, Part VII provides a summary of the key findings, limitations, and possible directions for future research. Supplementary material, including derivations and extended data, is presented in the appendices to maintain the clarity of the main text.

Part II

Classical and Quantum Mechanics

Chapter 2

2. Preliminaries of Quantum Mechanics

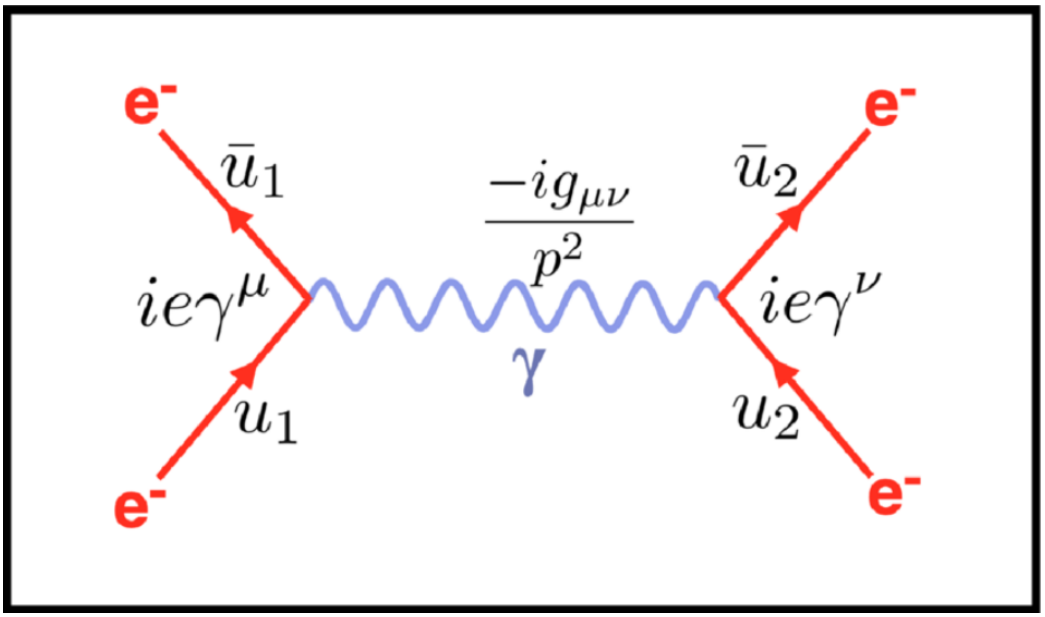

Figure 1. A Feynman diagram illustrating the interaction between two electrons via the exchange of a virtual photon. This diagram represents the fundamental electromagnetic interaction.[7]

> "I think I can safely say that nobody understands quantum mechanics.[16]
>
> Richard Feynman, *The Feynman Lectures on Physics.*"

Quantum mechanics forms the theoretical foundation essential for describing the behavior of microscopic systems at atomic and subatomic scales. Distinctively characterized by principles unlike classical physics, quantum mechanics introduces concepts such as quantization, wave-particle duality, and probabilistic interpretation of physical quantities. Understanding these foundational principles is critical for applying computational techniques such as PINNs effectively to quantum mechanical problems. In this section, we will review the fundamental postulates, equations, and concepts of quantum mechanics. These include the foundational postulates underpinning quantum theory, Schrödinger's equation in both its time-independent and time-dependent forms, the variational principle as a computational tool, the significance and definition of quantum numbers, and finally, the virial theorem which provides essential insights into the energetic relationships within quantum systems. The aim of this preliminary overview is to ensure a clear theoretical foundation upon which more advanced computational methods and their applications can be explored in subsequent chapters.

i. Postulates of Quantum Mechanics

Quantum mechanics is based on a set of postulates that describe how physical systems behave at the microscopic scale. The postulates below represent the standard formulation used in wave mechanics, particularly in the context of single-particle systems.

1. **State Postulate:** The state of a quantum system is fully described by a complex-valued wave function $\Psi(\mathbf{r}, t)$, which contains all accessible information about the system. The wave function must be square-

integrable and normalized:

$$\int_{-\infty}^{+\infty} \Psi^*(\mathbf{r},t)\Psi(\mathbf{r},t)\,d\tau = 1 \tag{2.1}$$

2. **Observable Postulate:** To each classical observable, there corresponds a Hermitian, linear operator $\hat{O}$ in quantum mechanics.

3. **Measurement Postulate:** Upon measurement of an observable associated with operator $\hat{O}$, the only possible outcomes are the eigenvalues o that satisfy the eigenvalue equation:

$$\hat{O}\Psi(\mathbf{r},t) = o\Psi(\mathbf{r},t) \tag{2.2}$$

   After the measurement, the system collapses into the corresponding eigenstate.

4. **Expectation Value Postulate:** The expectation value of an observable $\hat{O}$ in a normalized state $\Psi(\mathbf{r},t)$ is given by:

$$\langle\hat{O}\rangle = \frac{\int \Psi^*(\mathbf{r},t)\hat{O}\Psi(\mathbf{r},t)\,d\mathbf{r}}{\int \Psi^*(\mathbf{r},t)\Psi(\mathbf{r},t)\,d\mathbf{r}} \tag{2.3}$$

   For normalized wave functions, the denominator is 1.

5. **Time Evolution Postulate:** The time evolution of the wave function is governed by the time-dependent Schrödinger equation:

$$\hat{H}\Psi(\mathbf{r},t) = i\hbar\frac{\partial\Psi(\mathbf{r},t)}{\partial t} \tag{2.4}$$

   where $\hat{H}$ is the Hamiltonian operator of the system.

6. **Symmetrization Postulate:** For systems of identical particles, the total wave function must be either symmetric or antisymmetric under exchange of all coordinates (including spin). In particular, for fermions: this antisymmetry ensures that the Pauli exclusion principle is satisfied.

ii. Time independent/dependent Schrödinger equation

The Schrödinger equation governs the time evolution and spatial behavior of quantum systems. It appears in two key forms depending on whether the Hamiltonian is time-dependent.

Time-Dependent Schrödinger Equation (TDSE):

$$i\hbar\frac{\partial}{\partial t}\Psi(\vec{r},t) = \hat{H}\Psi(\vec{r},t) \tag{2.5}$$

Time-Independent Schrödinger Equation (TISE): Assuming a separable solution of the form $\Psi(\vec{r},t) = \psi(\vec{r})e^{-iEt/\hbar}$, the TISE reads:

$$\hat{H}\psi(\vec{r}) = E\psi(\vec{r}) \tag{2.6}$$

where $\hat{H}$ is typically given by:

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r}) \tag{2.7}$$

iii. The variational principle

The variational principle provides a powerful approximation method for determining the ground state energy of a quantum system when an exact solution to the Schrödinger equation is not accessible.

In quantum mechanics, states are often represented by kets, denoted as $|\psi\rangle$, residing in a Hilbert space. The corresponding dual vector is a bra, $\langle\psi|$. The inner product of two states $|\phi\rangle$ and $|\psi\rangle$ is written as $\langle\phi|\psi\rangle$, and the expectation value of an operator $\hat{A}$ in the state $|\psi\rangle$ is $\langle\psi|\,\hat{A}\,|\psi\rangle$, assuming $\langle\psi|\psi\rangle = 1$.

Let $|\phi\rangle$ be a trial wavefunction normalized such that $\langle\phi|\phi\rangle = 1$. Then the expectation value of the Hamiltonian gives an upper bound to the ground state energy E_0:

$$E[\phi] = \langle\phi|\,\hat{H}\,|\phi\rangle \geq E_0 \tag{2.8}$$

Optimization over a suitable class of trial functions $\{|\phi\rangle\}$ gives the best approximation:

$$E_0 \approx \min_{\phi} \langle\phi|\,\hat{H}\,|\phi\rangle \tag{2.9}$$

iv. Quantum numbers

Quantum numbers arise from solving the Schrödinger equation in systems with specific symmetries, particularly central potentials. They quantify conserved quantities and determine allowed energy levels and states. For the hydrogen-like atom, the relevant quantum numbers are:

- Principal quantum number: $n = 1, 2, 3, \ldots$
- Orbital angular momentum quantum number: $l = 0, 1, \ldots, n-1$
- Magnetic quantum number: $m_l = -l, -l+1, \ldots, l$
- Spin quantum number: $s = \pm\frac{1}{2}$ (with $m_s = \pm\frac{1}{2}$)

These quantum numbers label the eigenstates of the commuting operators: $\hat{H}$, $\hat{L}^2$, $\hat{L}_z$, and $\hat{S}_z$.

v. The virial theorem

The Virial Theorem relates the average kinetic energy $\langle T\rangle$ and potential energy $\langle V\rangle$ in a bound system. For a potential of the form $V(\vec{r}) \propto r^k$, the theorem states:

$$2\langle T\rangle = k\langle V\rangle \tag{2.10}$$

For Coulombic potentials (e.g., hydrogen atom), where $V(r) \propto -1/r$ (so $k = -1$), we have:

$$2\langle T\rangle = -\langle V\rangle \tag{2.11}$$

This theorem provides a consistency check for approximated wavefunctions or numerical methods.

Chapter 3

3. Many-Body Quantum Mechanics

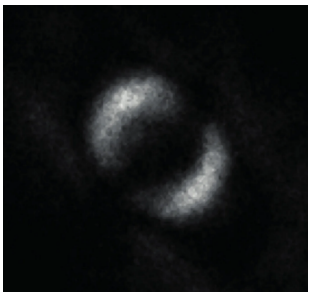

Figure 2. The first direct image of quantum entanglement, demonstrating two particle groups sharing the same quantum state $\Psi(X)$ was published by Moreau et al.[38]

> "The whole is something besides the parts, and greater than the parts.[4]
>
> Aristotle, *Metaphysics*"

In Chapter 2, we examined quantum systems involving a single particle. Extending this to systems composed of multiple interacting particles, we arrive at the many-body generalization of the Schrödinger equation:

$$E_n = \langle \Psi_n(X) | \, \hat{H} \, | \Psi_n(X) \rangle \, , \tag{3.1}$$

where $\Psi_n(X)$ is the many-body wavefunction corresponding to the nth eigenstate of the system, and $X = \{x_1, x_2, \ldots, x_N\} = \{\{\mathbf{r}_1, \sigma_1\}, \{\mathbf{r}_2, \sigma_2\}, \ldots, \{\mathbf{r}_N, \sigma_N\}\}$ denotes the complete set of spatial coordinates $\mathbf{r}_i$ and spin degrees of freedom σ_i for all N particles.

Although formally similar to the single-particle case, the many-body formulation presents substantial challenges due to the exponential increase in complexity with the number of particles. The Hamiltonian $\hat{H}$ governs the dynamics of the system and includes both single-particle and interaction terms. A detailed decomposition of this operator is provided in the next section.

Given the dimensionality of the configuration space and the combinatorial growth of the Hilbert space, storing or computing the full wavefunction for large systems quickly becomes intractable. To address this, we must adopt strategies for constructing and approximating many-body wavefunctions. The remainder of this chapter will focus first on the structure of the many-body Hamiltonian, followed by an in-depth treatment of wavefunction construction methods.

i. The electronic Hamiltonian

Recall the Hamiltonian of a single-particle system: it encapsulates the total energy as the sum of kinetic and potential contributions. For many-body systems, the form remains structurally similar:

$$\hat{H} = \hat{K} + \hat{V}, \tag{3.2}$$

where $\hat{K}$ is the kinetic energy and $\hat{V}$ represents the potential energy. However, when dealing with systems of electrons, additional considerations are needed. Electrons are charged particles and therefore interact with one another via the Coulomb force. These inter-electronic interactions must be incorporated into the Hamiltonian explicitly.

Accordingly, the potential energy term is typically decomposed into external and interaction contributions:

$$\hat{V} = \hat{V}_{\text{ext}} + \hat{V}_{\text{int}} \tag{3.3}$$

where $\hat{V}_{\text{ext}}$ includes potentials arising from external fields (such as nuclear Coulomb potentials), and $\hat{V}_{\text{int}}$ accounts for electron-electron interactions. For a pair of electrons, this interaction is given by Coulomb's law:

$$\hat{V}_{\text{int}} = k_e \frac{e^2}{r_{1,2}}, \tag{3.4}$$

where $r_{1,2} = |\mathbf{r}_1 - \mathbf{r}_2|$ is the inter-electronic distance.

Generalizing to an N-electron system, the full electronic Hamiltonian becomes:

$$\hat{H} = -\sum_{i=1}^{N} \frac{\hbar^2}{2m_e} \nabla_i^2 + \sum_{i=1}^{N} \hat{V}_{\text{ext}}(x_i) + \sum_{i=1}^{N} \sum_{j>i}^{N} k_e \frac{e^2}{r_{i,j}}, \tag{3.5}$$

where $r_{i,j} = |\mathbf{r}_i - \mathbf{r}_j|$ denotes the relative distance between particles i and j. In what follows, we adopt atomic units, setting $\hbar = m_e = k_e = e = 1$; see Appendix A for further details.

Inserting this Hamiltonian into the expectation value expression in Equation (3.1), the energy can be decomposed into three distinct contributions:

$$E_n = \sum_{i=1}^{N} \left[-\frac{1}{2} \langle \Psi_n(X) | \nabla_i^2 | \Psi_n(X) \rangle + \langle \Psi_n(X) | \hat{V}_{\text{ext}}(x_i) | \Psi_n(X) \rangle + \sum_{j>i}^{N} \langle \Psi_n(X) | \frac{1}{r_{i,j}} | \Psi_n(X) \rangle \right], \tag{3.6}$$

The first two terms are known as one-body integrals, which in many cases admit closed-form analytical solutions. In contrast, the final term representing electron-electron interactions is significantly more complex. Analytic solutions exist only in the two-particle case. Beyond this, accurate evaluation generally requires numerical approaches. In Chapter 12, we revisit this challenge in the context of Quantum PINNs and explore computational strategies for handling these interaction terms.

ii. The Many-body wave function

According to the first postulate of quantum mechanics, as introduced in Section 2 i, the wave function encapsulates all physical information about a quantum system. Consequently, any measurable property (i.e., observable) can, in principle, be derived from the wave function. This renders the determination of the system's wave function a central objective in quantum mechanical analyses.

In Section 2, we introduced the wave function for a single particle, commonly referred to as the single-particle function (SPF), denoted by $\psi(\mathbf{r}, \sigma)$. A natural question then arises: can we construct the wave function of an N-electron system by simply combining individual SPFs?

A straightforward, albeit naive, approach is to take the product of the single-particle functions for all particles:

$$\Psi(X) = \psi(\mathbf{r}_1, \sigma_1)\psi(\mathbf{r}_2, \sigma_2) \dots \psi(\mathbf{r}_N, \sigma_N), \tag{3.7}$$

where $X = \{(\mathbf{r}_1, \sigma_1), \dots, (\mathbf{r}_N, \sigma_N)\}$ represents the complete set of spatial and spin coordinates. This product form is known as the Hartree product. While simple, it fails to account for the essential symmetry properties of identical particles. In particular, for fermions such as electrons, the total wave function must be antisymmetric under exchange of any two particles.

To incorporate these symmetry requirements, one must instead express the many-body wave function using constructs such as determinants or permanents, as will be discussed in Section 3 ii 2. However, satisfying symmetry is only one of several conditions a physically valid wave function must meet. The most important criteria include:

1. Normalizability: The wave function must be square-integrable so that the total probability is one. This implies $\Psi(X) \to 0$ as any $|\mathbf{r}_i| \to \infty$.
2. Cusp Condition: The Coulomb potential diverges as $r_{ij} \to 0$. To cancel this, the wave function must develop a cusp at short interparticle distances (see Section 3 ii 4).
3. (Anti)Symmetry: The wave function must be symmetric for bosons and antisymmetric for fermions under particle exchange. For electrons, antisymmetry is essential.

1. *The Pauli Principle and Symmetries*

Symmetry considerations are foundational in quantum mechanics and often provide useful simplifications in both conceptual understanding and computation. Consider a permutation operator $\hat{P}_{ij}$ (or $\hat{P}(i \leftrightarrow j)$) that exchanges the coordinates of particles i and j in a many-body wave function. For a system of M particles, the action of this operator is given by

$$\hat{P}_{ij}\Psi_n(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_M) = p\Psi_n(x_1, \ldots, x_j, \ldots, x_i, \ldots, x_M), \tag{3.8}$$

where p is the eigenvalue associated with the permutation. Since applying the same permutation operator twice must return the system to its original configuration, we require $\hat{P}_{ij}^2 = \mathbb{I}$ (the identity operator), implying $p^2 = 1$. Therefore, $p = \pm 1$.

Particles for which the many-body wave function is *antisymmetric* under coordinate exchange ($p = -1$) are called *fermions*. These particles possess half-integer spin and obey the Pauli exclusion principle: no two identical fermions can occupy the same single-particle state. Conversely, particles for which the wave function is *symmetric* under exchange ($p = +1$) are called *bosons*, named after Satyendra Nath Bose, and they have integer spin.

2. *The Slater determinant*

In a system of many fermions, the total wave function must satisfy the antisymmetry requirement imposed by the Pauli exclusion principle. According to the first postulate of quantum mechanics 2.1, the many-body wave function must contain all information about the system and must be constructed from the set of occupied single-particle functions (SPFs). For fermionic systems, this construction must ensure that the wave function changes sign under the exchange of any two identical particles, thereby enforcing the exclusion principle.

This antisymmetry requirement is naturally fulfilled by expressing the wave function as a determinant. Consider a system of two identical fermions, described by SPFs $\psi_1(\mathbf{x})$ and $\psi_2(\mathbf{x})$ (where $\mathbf{x}$ includes both spatial $\mathbf{r}$ and spin σ coordinates), with coordinates $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively. The corresponding antisymmetric two-particle wave function is given by:

$$\Psi(X) = \frac{1}{\sqrt{2}}\begin{vmatrix}\psi_1(\mathbf{x}_1) & \psi_2(\mathbf{x}_1)\\ \psi_1(\mathbf{x}_2) & \psi_2(\mathbf{x}_2)\end{vmatrix} = \frac{1}{\sqrt{2}}\left[\psi_1(\mathbf{x}_1)\psi_2(\mathbf{x}_2) - \psi_2(\mathbf{x}_1)\psi_1(\mathbf{x}_2)\right]. \tag{3.9}$$

This expression vanishes when the two fermions occupy the same quantum state (i.e., $\psi_1 = \psi_2$ or $\mathbf{x}_1 = \mathbf{x}_2$ leading to identical rows if SPFs are the same), in accordance with the Pauli principle. However, if they differ in spin, they may occupy the same spatial orbital, and the determinant remains non-zero.

For systems with N fermions, the many-body wave function is generalized as a Slater determinant of the occupied SPFs:

$$\Psi(X) = \frac{1}{\sqrt{N!}}\begin{vmatrix}\psi_1(\mathbf{x}_1) & \psi_2(\mathbf{x}_1) & \cdots & \psi_N(\mathbf{x}_1)\\ \psi_1(\mathbf{x}_2) & \psi_2(\mathbf{x}_2) & \cdots & \psi_N(\mathbf{x}_2)\\ \vdots & \vdots & \ddots & \vdots\\ \psi_1(\mathbf{x}_N) & \psi_2(\mathbf{x}_N) & \cdots & \psi_N(\mathbf{x}_N)\end{vmatrix}. \tag{3.10}$$

Each SPF $\psi(\mathbf{x}) = \psi(\mathbf{r}, \sigma)$ is expressed as a tensor product of a spatial orbital $\phi(\mathbf{r})$ and a spin function $\xi(\sigma)$:

$$\psi(\mathbf{r}, \sigma) = \phi(\mathbf{r})\xi(\sigma). \tag{3.11}$$

Assuming a spin-independent Hamiltonian, the total Slater determinant can be factorized into separate spin-up and spin-down components. This decomposition enables the elimination of the explicit spin coordinate dependence, $\xi(\sigma)$, from the wave function. Consequently, the single-particle basis can be constructed purely from spatial orbitals, $\phi(\mathbf{r})$, simplifying the representation of the many-body state.

Notation clarification. Throughout this thesis, we adopt the following conventions:

- Ψ: many-body (antisymmetric) wave function.
- ψ: single-particle function (SPF), including spatial and spin parts.
- ϕ: spatial part of the SPF.
- ξ: spin part of the SPF (often omitted under spin-independent Hamiltonians).
- Ψ_i: components of the many-body wave function, used when decomposed (e.g., product of Slater determinants).
- φ: basis functions (discussed in the next section).

3. Basis sets

A *basis set* is a collection of one-particle functions used to construct the single-particle functions (SPFs) introduced earlier. These basis functions typically represent atomic eigenstates and are therefore referred to as *atomic orbitals* (AOs). SPFs, in turn, are linear combinations of atomic orbitals optimized to describe electrons in molecules, and are thus called *molecular orbitals* (MOs).

Common basis sets include Pople-type sets (e.g., 6-31G), correlation-consistent sets (e.g., cc-pVXZ, where X=D,T,Q...), and Slater-type orbitals (e.g., STO-nG), most of which are built from Gaussian-type functions (GTFs). Gaussian orbitals are preferred due to their efficiency in post-Hartree–Fock methods, where integral evaluations become computationally tractable.

In this work, the SPFs $\psi(\mathbf{r}, \sigma)$ are typically not expanded in an external basis. Instead, they are solutions to the non-interacting problem, and the electron–electron correlations are introduced via the Jastrow factor. As a result, the same functions are used for both interacting and non-interacting systems.

When basis expansions are employed, it is common to express molecular orbitals as a linear combination of atomic orbitals (LCAO):

$$\phi_i(\mathbf{r}) = \sum_{j=1}^{M} c_{ji}\, \varphi_j(\mathbf{r}), \tag{3.12}$$

where $\{\varphi_j(\mathbf{r})\}$ are atomic orbitals and c_{ji} are coefficients determined, for instance, via the Hartree–Fock procedure to optimize the total energy. Increasing the size of the basis (M) improves accuracy but also raises computational cost.

The specific basis functions used are introduced in Chapter 9 iii, where they are linked to the systems under study.

4. Modeling the correlation

According to Coulomb's law, the interaction potential between two charged particles i and j takes the form $1/r_{ij}$, where r_{ij} is the inter-particle distance. As $r_{ij} \to 0$, the potential diverges, introducing a singularity that presents both analytical and numerical challenges in quantum mechanical simulations. To ensure that the wave function

remains physically meaningful and numerically stable near these singularities, one typically enforces a cancellation between the singularity in the Coulomb interaction and the kinetic energy contribution.

By expressing the kinetic energy operator in spherical coordinates near the particle coalescence point, one obtains the so-called *cusp condition*, which provides a constraint on the derivative of the wave function:

$$\frac{1}{\Psi(r_{ij}=0)}\frac{\partial\Psi(r_{ij})}{\partial r_{ij}}\bigg|_{r_{ij}=0} = \mu_{ij}q_iq_j, \tag{3.13}$$

assuming $\Psi(r_{ij}=0)\neq 0$. Here $\mu_{ij}=\frac{m_im_j}{m_i+m_j}$ is the reduced mass of the two particles, and q_i and q_j are their respective charges (in atomic units, $q_iq_j=-1$ for electron-electron, $q_iq_j=-Z$ for electron-nucleus). This condition is satisfied if the wave function near the singularity behaves as $\Psi(r_{ij})\sim\exp(\mu_{ij}q_iq_jr_{ij})$ for small r_{ij} (for unlike charges) or more generally, if Ψ has a specific linear dependence on r_{ij} for small r_{ij}. The wave function thus develops a cusp-like structure as the particles approach each other. This result was first derived by Kato and is often referred to as the Kato cusp condition or the Kato theorem.

Different electronic structure methods incorporate electron–electron correlation in fundamentally different ways. The Hartree–Fock method approximates the ground state as a single Slater determinant, constructed from molecular orbitals that are themselves expanded in a fixed basis of atomic orbitals, as shown in equation (3.12). The coefficients in this expansion are typically obtained by solving the Hartree–Fock equations self-consistently. As this approach does not include correlation effects explicitly (other than exchange correlation due to the Slater determinant), it is often referred to as a mean-field approximation.

Post-Hartree–Fock methods, such as Configuration Interaction (CI) and Coupled Cluster (CC) theory, extend this framework by expressing the many-body wave function as a linear combination of multiple Slater determinants. Correlation effects are captured through the amplitudes of these determinants. In principle, both methods can yield exact results in the limit of a complete determinant expansion (Full CI), but their computational cost scales steeply with system size, which limits their applicability to relatively small systems.

iii. Electron density

In quantum many-body computations, the electron density is frequently evaluated due to its theoretical significance and experimental accessibility. First, electron densities can be related to experimentally measurable quantities (e.g., via X-ray diffraction), providing a direct means to benchmark quantum simulations. Second, they offer compact and interpretable information about the spatial distribution of particles in the system.

In this work, we restrict our attention to the one-body and two-body electron densities, denoted by $\rho_1(\mathbf{r})$ and $\rho_2(\mathbf{r}_1,\mathbf{r}_2)$, respectively. For an N-electron system described by $\Psi(X)=\Psi(\mathbf{x}_1,\ldots,\mathbf{x}_N)$: The one-body density at point $\mathbf{r}$ is:

$$\rho_1(\mathbf{r}) = N\sum_{\text{all spins}}\int d\mathbf{r}_2\cdots d\mathbf{r}_N\ \left|\Psi(\mathbf{r},\sigma_1;\mathbf{r}_2,\sigma_2;\ldots;\mathbf{r}_N,\sigma_N)\right|^2, \tag{3.14}$$

The two-body density for finding one electron at $\mathbf{r}_1$ and another at $\mathbf{r}_2$ is:

$$\rho_2(\mathbf{r}_1,\mathbf{r}_2) = N(N-1)\sum_{\text{all spins}}\int d\mathbf{r}_3\cdots d\mathbf{r}_N\ \left|\Psi(\mathbf{r}_1,\sigma_1;\mathbf{r}_2,\sigma_2;\ldots;\mathbf{r}_N,\sigma_N)\right|^2. \tag{3.15}$$

These integrals are generally analytically tractable only for a small number of idealized systems. For more realistic or interacting systems, they must be evaluated numerically.

Due to the indistinguishability of identical particles in quantum mechanics, the resulting electron density does not depend on which particle index is integrated out for ρ_1, or which pair for ρ_2. This reflects the fundamental symmetry of the system and justifies the use of reduced density functions.

Electron densities are normalized such that integrating $\rho_1(\mathbf{r})$ over all space gives the total number of electrons N:

$$\int \rho_1(\mathbf{r}) d\mathbf{r} = N. \tag{3.16}$$

The physical interpretation of the electron density depends on the order of the density function. The one-body density, $\rho_1(\mathbf{r})$, gives the probability density of finding *any* one of the N electrons at a given point $\mathbf{r}$ in space (multiplied by N). It provides a clear picture of the spatial distribution of electrons in the system.

The two-body density $\rho_2(\mathbf{r}_1, \mathbf{r}_2)$ gives the joint probability density of finding one electron at $\mathbf{r}_1$ and simultaneously another electron at $\mathbf{r}_2$ (multiplied by $N(N-1)$). This correlation-sensitive function is essential when studying inter-electronic interactions and spatial correlations. In systems where strong three-body forces play a role, such as in nuclear matter, it can be useful to consider the three-body density $\rho_3(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$ to gain further insight.

For systems with spherical symmetry, such as closed-shell circular quantum dots, the radial density profile is often used instead. In these cases, the electron density depends only on the radial coordinate, significantly simplifying the analysis.

In the case of non-interacting systems, if the total wave function is a simple Hartree product (Equation 3.7) of orthonormal single-particle functions $\psi_k(\mathbf{r}_k)$, the one-body density simplifies. However, for fermions, the wave function is a Slater determinant. If Ψ is a single Slater determinant of orthonormal SPFs $\phi_k(\mathbf{x})$, then:

$$\rho_1(\mathbf{r}) = \sum_{k=1}^{N} \sum_{\sigma} |\phi_k(\mathbf{r}, \sigma)|^2, \tag{3.17}$$

where the sum is over the N occupied spin-orbitals ϕ_k. This result is physically significant and serves as a useful test case for validating numerical implementations of the density calculation, as discussed in Section 13.

Chapter 4

4. Systems

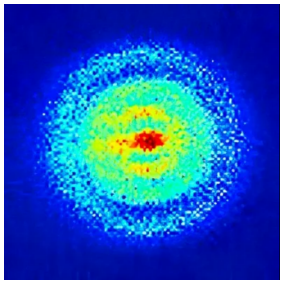

Figure 3. The first direct image of a hydrogen atom was captured in 2013 using an ultra-sensitive camera. Remarkably, the image reveals the probability distribution $|\Psi(x)|^2$ with visible structure. This was published by Stodolna et al.[50] under the title Hydrogen atoms under magnification.

> “What I cannot create, I do not understand.[15]
>
> Richard Feynman, *From his blackboard at Caltech*”

i. Classical Systems

1. *Diffusion Equation*

The diffusion equation, also known as the heat equation, is a partial differential equation that describes the distribution of heat or concentration in a given region over time. The general form of the diffusion equation at location $\mathbf{r}$ is given by:

$$\frac{\partial \phi(\mathbf{r},t)}{\partial t} = \nabla \cdot [D(\phi,\mathbf{r})\nabla\phi(\mathbf{r},t)] \tag{4.1}$$

Here, $\phi(\mathbf{r},t)$ represents the density of the diffusing material at the spatial location $\mathbf{r}$ and time t. The term $D(\phi,\mathbf{r})$ is the collective diffusion coefficient, which in general may depend on both the field ϕ and the position $\mathbf{r}$. The operator ∇ denotes the gradient, and its divergence $\nabla\cdot(\cdot)$ captures the net flux of the material due to local gradients in density. For this paper, we make the simplifying assumption that the diffusion coefficient $D(\phi,\mathbf{r})$ is a constant, D, independent of both ϕ and $\mathbf{r}$. This assumption linearizes the equation and reduces it to the classical form:

$$\frac{\partial \phi(\mathbf{r},t)}{\partial t} = D\nabla^2\phi(\mathbf{r},t) \tag{4.2}$$

In two spatial dimensions, and writing $u = \phi(x,y,t)$, this equation takes the form:

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{4.3}$$

2. *Diffusion of a Gaussian Function*

To explore the behavior of the diffusion equation under practical boundary conditions, we examine the evolution of a localized Gaussian initial condition. This allows us to assess how an initially peaked distribution spreads over

time when subject to Dirichlet boundary conditions. Our initial function for the Gaussian hill is:

$$u_0(x, y) = \exp\big(-a(x^2 + y^2)\big). \tag{4.4}$$

This analytical form offers convenient symmetry and decay properties that make it particularly suitable for benchmarking numerical solvers of the heat equation.

3. Navier-Stokes Equation

Whereas the diffusion equation models scalar transport such as heat or concentration, fluid flow involves the transport of both momentum and mass. This leads us naturally to the Navier-Stokes equations, which describe the dynamics of incompressible fluids. The incompressible Navier-Stokes equations in two spatial dimensions for velocity $\mathbf{u} = (u, v)$ and pressure p are given by the momentum equation:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u}\right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f}, \tag{4.5}$$

and the continuity equation (incompressibility condition):

$$\nabla \cdot \mathbf{u} = 0. \tag{4.6}$$

In this formulation, ρ is the constant fluid density, μ is the dynamic viscosity, and $\mathbf{f}$ represents external body forces per unit volume. The term $\sigma(\mathbf{u}, p)$ mentioned in your OCR often refers to the stress tensor, where for a Newtonian incompressible fluid, $\nabla \cdot \sigma = -\nabla p + \mu \nabla^2 \mathbf{u}$ (ignoring bulk viscosity term which is zero for incompressible flow). The strain-rate tensor $\boldsymbol{\epsilon}(\mathbf{u})$ is given by the symmetric part of the velocity gradient: $\boldsymbol{\epsilon}(\mathbf{u}) = \frac{1}{2}(\nabla \mathbf{u} + (\nabla \mathbf{u})^T)$. This system is inherently nonlinear and significantly more complex than the diffusion equation. However, certain numerical strategies developed for simpler elliptic and parabolic PDEs, such as the diffusion and Poisson equations, form the basis for solving the Navier-Stokes equations as well.

4. Poisson Equation

To close the loop between scalar transport and fluid flow, we now consider the Poisson equation, which arises frequently in the context of both pressure correction methods for incompressible flows and in steady-state diffusion problems. The Poisson equation plays a fundamental role in various fields of physics and engineering, particularly when dealing with potential fields and steady-state heat transfer. In the context of a two-dimensional spatial domain Ω, the Poisson equation is expressed as follows:

$$\nabla^2 u(\mathbf{x}) = f(\mathbf{x}), \quad \forall \mathbf{x} \in \Omega \tag{4.7}$$

subject to boundary conditions, e.g., $u(\mathbf{x}) = g(\mathbf{x}), \quad \forall \mathbf{x} \in \partial\Omega$ (Dirichlet). Here ∇^2 denotes the Laplacian operator, u represents the solution, f is the source term. In Cartesian coordinates, this equation is expressed as:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y). \tag{4.8}$$

Equation (4.8) highlights the rate of change of u along both the x- and y-directions. For a rectangular domain defined by $(x, y) \in [0, L_x] \times [0, L_y] = \Omega$, the equation governs the behavior of the field within this bounded area. A critical aspect of solving the Poisson equation in a bounded domain involves the imposition of boundary conditions. Specifically, we consider Dirichlet boundary conditions, where the solution u is known or fixed at the boundary. The rectangular domain presents four sides, requiring the solution to be specified along all these boundaries. This can be mathematically expressed as:

$$u(0, y) = g_1(y), \qquad u(L_x, y) = g_2(y), \tag{4.9}$$

$$u(x, 0) = h_1(x), \qquad u(x, L_y) = h_2(x), \tag{4.10}$$

where g_1, g_2, h_1, and h_2 are functions defining the known conditions at the respective boundaries. These boundary conditions are crucial for uniquely determining the solution within the domain.

ii. Quantum mechanical systems

1. *Time dependent Schrödinger equation in a harmonic oscillator*

We begin our exploration of quantum systems by introducing the partial differential equation central to much of quantum dynamics: the Time-dependent Schrödinger Equation (TDSE). In its general form for a single particle of mass m moving in a potential $V(\mathbf{r}, t)$, it reads:

$$i\hbar\frac{\partial\psi(\mathbf{r},t)}{\partial t} = \left[-\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r},t)\right]\psi(\mathbf{r},t) \tag{4.11}$$

Here, $\psi(\mathbf{r}, t)$ is the wavefunction, a complex-valued function whose squared modulus $|\psi(\mathbf{r}, t)|^2$ gives the probability density of finding the particle at position $\mathbf{r}$ at time t. For this thesis, we are often concerned with two spatial dimensions, where the TDSE becomes:

$$i\hbar\frac{\partial\psi(x,y,t)}{\partial t} = \left[-\frac{\hbar^2}{2m}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) + V(x,y,t)\right]\psi(x,y,t) \tag{4.12}$$

A common and important potential is the harmonic oscillator, $V(x, y) = \frac{1}{2}m\omega^2(x^2 + y^2)$, which serves as a fundamental model system in quantum mechanics due to its analytical tractability and relevance to various physical phenomena, such as molecular vibrations or particles in electromagnetic traps. Studying the TDSE within this potential provides insight into the time evolution of quantum states like wave packets.

2. *Two electrons with Coulomb interaction*

A foundational yet non-trivial interacting quantum system involves two electrons confined within an external harmonic oscillator potential, interacting via the standard Coulomb repulsion. This system serves as a valuable benchmark for many-body methods and exhibits interesting correlation physics. M. Taut demonstrated in [52] that this problem, while generally complex, admits exact analytical solutions for a specific, discrete infinite set of oscillator frequencies.

The Hamiltonian for this system, assuming atomic units ($\hbar = m_e = e = k_e = 1$), is given by:

$$\hat{H} = -\frac{1}{2}\nabla_1^2 - \frac{1}{2}\nabla_2^2 + \frac{1}{2}\omega^2 r_1^2 + \frac{1}{2}\omega^2 r_2^2 + \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \tag{4.13}$$

where $\mathbf{r}_1$ and $\mathbf{r}_2$ are the position vectors of the two electrons, and ω is the angular frequency of the harmonic trap. The final term represents the Coulomb interaction between the electrons.

To simplify the problem, we introduce center-of-mass (CM) and relative coordinates:

$$\mathbf{R} = \frac{1}{2}(\mathbf{r}_1 + \mathbf{r}_2) \tag{4.14}$$

$$\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1 \tag{4.15}$$

In these coordinates, the Hamiltonian decouples into two independent parts:

$$\hat{H} = \hat{H}_R + \hat{H}_r \tag{4.16}$$

where $\hat{H}_R$ describes the motion of the center of mass (total mass $M = 2m_e = 2$ in a.u.) and $\hat{H}_r$ describes the relative motion of the two electrons (reduced mass $\mu = m_e/2 = 1/2$ in a.u.). Explicitly:

$$\hat{H}_R = -\frac{1}{2M}\nabla_R^2 + \frac{1}{2}M\omega^2 R^2 = -\frac{1}{4}\nabla_R^2 + \omega^2 R^2 \tag{4.17}$$

$$\hat{H}_r = -\frac{1}{2\mu}\nabla_r^2 + \frac{1}{2}\mu\omega^2 r^2 + \frac{1}{r} = -\nabla_r^2 + \frac{1}{4}\omega^2 r^2 + \frac{1}{r} \tag{4.18}$$

The total energy E is the sum of the CM energy η and the relative energy ϵ: $E = \eta + \epsilon$. The center-of-mass Hamiltonian $\hat{H}_R$ describes a simple 3D harmonic oscillator whose solutions are well-known. The true complexity lies in solving the Schrödinger equation for the relative motion:

$$\hat{H}_r \phi(\mathbf{r}) = \epsilon \phi(\mathbf{r}) \tag{4.19}$$

This equation still contains the challenging Coulomb interaction term $1/r$.

Due to the separation, the total wavefunction $\Psi(\mathbf{r}_1, \mathbf{r}_2)$ can be factorized. Since the Hamiltonian is spin-independent, the spatial and spin parts separate. The Pauli exclusion principle requires the total wavefunction to be antisymmetric under particle exchange ($\mathbf{r}_1 \leftrightarrow \mathbf{r}_2$, $s_1 \leftrightarrow s_2$). As $\mathbf{R}$ is symmetric and $\mathbf{r}$ is antisymmetric under exchange of $\mathbf{r}_1$ and $\mathbf{r}_2$, the total wavefunction takes the form:

$$\Psi(\mathbf{r}_1, \mathbf{r}_2, s_1, s_2) = \xi(\mathbf{R}) \phi(\mathbf{r}) \chi(s_1, s_2) \tag{4.20}$$

where $\xi(\mathbf{R})$ is the CM wavefunction, $\phi(\mathbf{r})$ is the relative motion wavefunction, and $\chi(s_1, s_2)$ is the spin wavefunction. For the singlet spin state (antisymmetric χ, $S = 0$), $\phi(\mathbf{r})$ must be symmetric under parity ($\phi(-\mathbf{r}) = \phi(\mathbf{r})$), meaning it has even parity. For the triplet spin state (symmetric χ, $S = 1$), $\phi(\mathbf{r})$ must be antisymmetric ($\phi(-\mathbf{r}) = -\phi(\mathbf{r})$), meaning it has odd parity.

Introducing spherical coordinates for the relative motion $\mathbf{r} = (r, \theta, \varphi)$, the relative wavefunction can be written as $\phi(\mathbf{r}) = \frac{u_{nl}(r)}{r} Y_{lm}(\theta, \varphi)$. The parity requirement means l must be even for singlet states and odd for triplet states. The radial part $u_{nl}(r)$ satisfies the radial Schrödinger equation (using $\mu = 1/2$):

$$\left[-\frac{d^2}{dr^2} + \frac{1}{4}\omega^2 r^2 + \frac{1}{r} + \frac{l(l+1)}{r^2} \right] u_{nl}(r) = \epsilon_{nl} u_{nl}(r) \tag{4.21}$$

Taut's key finding [52] is that Equation (4.21) can be solved analytically if the solution $u_{nl}(r)$ can be expressed as a product of a Gaussian, a power term (r^{l+1}), and a finite polynomial in r^2. This condition is only met for a discrete, infinite set of oscillator frequencies ω. For these specific frequencies, the polynomial terminates, yielding exact eigenstates and eigenenergies ϵ_{nl}. For arbitrary ω, numerical methods are required. This system thus provides specific parameter regimes where exact solutions including electron correlation are known, making it an important test case.

3. Multiple fermions with and without interaction

Moving beyond two-particle systems, we encounter the formidable challenge of describing systems containing many identical fermions, such as the electrons in atoms, molecules, or solid materials. The complexity increases dramatically with the number of particles, N. We first consider the idealized case of non-interacting fermions before addressing the more realistic scenario involving interactions.

In a system of N non-interacting fermions moving in a common external potential $V(\mathbf{r})$, the Hamiltonian is simply the sum of single-particle Hamiltonians:

$$\hat{H}_0 = \sum_{i=1}^{N} \hat{h}_i = \sum_{i=1}^{N} \left(-\frac{\hbar^2}{2m} \nabla_i^2 + V(\mathbf{r}_i) \right) \tag{4.22}$$

While the particles do not directly interact, their behavior is profoundly constrained by the Pauli exclusion principle, which dictates that the total wavefunction $\Psi(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N)$ must be antisymmetric with respect to the exchange of the coordinates (spatial $\mathbf{r}_i$ and spin s_i, collectively denoted $\mathbf{x}_i$) of any two fermions. If we can solve the single-particle Schrödinger equation $\hat{h}\phi_j(\mathbf{x}) = \epsilon_j \phi_j(\mathbf{x})$ to find the single-particle orbitals (spin-orbitals) ϕ_j and their energies ϵ_j, the correctly antisymmetrized N-particle ground state wavefunction can be constructed as

a Slater determinant:

$$\Psi_0(\mathbf{x}_1, ..., \mathbf{x}_N) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_N(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_N(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{x}_N) & \phi_2(\mathbf{x}_N) & \cdots & \phi_N(\mathbf{x}_N) \end{vmatrix} \tag{4.23}$$

This wavefunction corresponds to filling the N lowest energy single-particle orbitals. The total energy of this non-interacting system is simply the sum of the energies of the occupied orbitals, $E_0 = \sum_{i=1}^{N} \epsilon_i$. This non-interacting model forms the basis for understanding phenomena like the electronic band structure of solids or the shell structure of atoms.

The situation becomes significantly more complex when inter-particle interactions are included. For electrons, this is typically the Coulomb repulsion. The Hamiltonian now contains an additional term summing over all pairs of particles:

$$\hat{H} = \sum_{i=1}^{N} \left(-\frac{\hbar^2}{2m} \nabla_i^2 + V(\mathbf{r}_i) \right) + \sum_{i<j}^{N} V_{\text{int}}(\mathbf{r}_i, \mathbf{r}_j) \tag{4.24}$$

where $V_{\text{int}}(\mathbf{r}_i, \mathbf{r}_j)$ is the interaction potential between particle i and particle j, e.g., $e^2/(4\pi\epsilon_0|\mathbf{r}_i - \mathbf{r}_j|)$ for Coulomb interaction (or $1/|\mathbf{r}_i - \mathbf{r}_j|$ in atomic units). The presence of the interaction term $\sum_{i<j} V_{\text{int}}$ couples the motion of all particles, meaning the Hamiltonian is no longer separable into independent single-particle problems. Consequently, the simple Slater determinant (4.23) formed from solutions of the non-interacting problem is no longer an eigenstate of the full Hamiltonian $\hat{H}$. Finding the exact eigenstates and eigenvalues of the interacting Hamiltonian constitutes the notoriously difficult quantum many-body problem. Exact analytical solutions are typically impossible for $N > 2$, necessitating the use of sophisticated approximation methods. Techniques such as Hartree-Fock theory provide a mean-field approximation, while methods like Configuration Interaction (CI), Coupled Cluster (CC), Density Functional Theory (DFT), and Quantum Monte Carlo (QMC) aim to incorporate the effects of electron correlation – the tendency of electrons to avoid each other due to repulsion, which is neglected in the simplest independent-particle models. Effectively tackling the interacting many-fermion system is crucial for accurately describing the properties of most real-world quantum systems.

Part III

Numerical Theory

Chapter 5

5. Supervised Learning

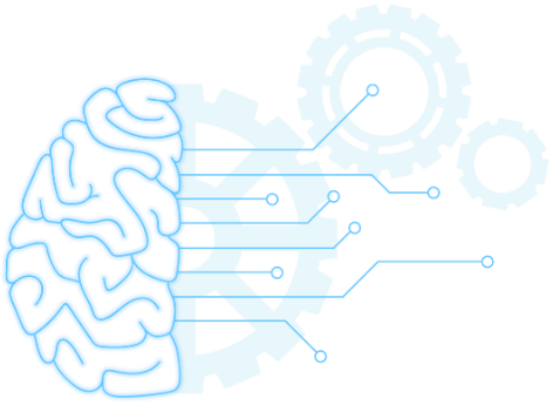

Figure 4. Machine Learning methods like Artificial neural networks are inspired and modeled to be like the network of neurons in the brain.

> “If a typical person can do a mental task with less than one second of thought, we can probably automate it using AI either now or in the near future.[41]
>
> Andrew Ng, *What Artificial Intelligence Can and Can't Do Right Now*”

The field of machine learning has received growing attention in recent years, often being framed as a modern breakthrough in computational science. Yet, many of the fundamental methods that fall under the umbrella of machine learning are not new. Linear regression, for instance, traces back to the early 1800s, when Adrien-Marie Legendre and Carl Friedrich Gauss independently developed the method of least squares.[32][19] What distinguishes recent developments is not the discovery of new principles, but rather the remarkable progress in computational power and the refinement of optimization techniques that have allowed even classical methods to scale to high-dimensional, complex problems. This resurgence has given rise to a rebranding of many traditional statistical tools under the broader and more flexible label of machine learning. Supervised learning, a central branch of machine learning, is concerned with mapping inputs to outputs using known data. It involves training a model on a labeled dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, where $\boldsymbol{x}_i \in \mathbb{R}^n$ denotes an input vector and $y_i \in \mathbb{R}$ is the corresponding target. The objective is to construct a function $f : \mathbb{R}^n \to \mathbb{R}$ that generalizes well to previously unseen data, that is, not only should $f(\boldsymbol{x}_i) \approx y_i$ for known data points, but it should also perform accurately on new inputs. This chapter focuses on the mathematical framework underpinning supervised learning. We begin with polynomial regression as a pedagogical example, gradually building toward more general models such as linear regression, regularization techniques, and ultimately neural networks. Throughout, we emphasize the statistical and numerical challenges involved, including overfitting, model complexity, and the bias-variance tradeoff.

i. Polynomial Regression

Linear regression models provide a fundamental tool for analyzing relationships between variables, yet their applicability is confined to scenarios where these relationships are linear. Often, the true underlying association between a predictor variable x and a response variable y exhibits non-linear patterns. Polynomial regression addresses this limitation by incorporating polynomial terms of the predictor variable, thereby extending the linear model framework to capture curvature in the data.

Given a dataset consisting of n observations $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, a polynomial regression model of degree d postulates the following relationship:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_d x_i^d + \epsilon_i \tag{5.1}$$

Here, y_i denotes the response for the i-th observation, x_i is the corresponding predictor value, $\beta_0, \beta_1, \ldots, \beta_d$ represent the model coefficients to be estimated, d signifies the polynomial's degree, and ϵ_i is the random error term associated with the i-th observation. The error terms are commonly assumed to be independent and identically distributed with zero mean and constant variance (σ_ϵ^2).

Although Equation (5.1) models a non-linear dependency of y on x, it remains linear with respect to the coefficients β_j. This linearity allows us to treat polynomial regression as a special case of multiple linear regression. By defining a set of features derived from the original predictor x as $x_{i1} = x_i, x_{i2} = x_i^2, \ldots, x_{id} = x_i^d$, the model transforms into $y_i = \beta_0 + \sum_{j=1}^{d} \beta_j x_{ij} + \epsilon_i$. Consequently, the estimation of the coefficients β_j can proceed using the standard ordinary least squares (OLS) method. This method seeks to minimize the Residual Sum of Squares (RSS), defined as:

$$\text{RSS}(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} \left(y_i - \left(\beta_0 + \sum_{j=1}^{d} \beta_j x_i^j \right) \right)^2 \tag{5.2}$$

where $\hat{y}_i = \beta_0 + \sum_{j=1}^{d} \beta_j x_i^j$ is the predicted value for the i-th observation given the estimated coefficients $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_d)^T$.

A crucial decision in polynomial regression is the selection of the degree d. Choosing a degree that is too low ($d = 1$ being the simplest case of linear regression) may render the model incapable of adequately capturing the inherent non-linearity in the data, a situation known as underfitting. Conversely, selecting an excessively high degree can cause the model to fit the training data too closely, including the random noise. This phenomenon, termed overfitting, typically leads to poor generalization performance on unseen data. Effectively choosing d requires balancing the model's complexity against its ability to generalize. This challenge is fundamentally governed by the bias-variance tradeoff, a central concept in statistical learning that describes the relationship between model flexibility and its expected prediction error.

ii. Bias-Variance Tradeoff

The challenge of selecting an appropriate level of model complexity, such as the degree d in polynomial regression, is directly related to a model's generalization capability. The bias-variance tradeoff provides a foundational decomposition of the expected prediction error, illuminating how model properties influence performance on new, unseen data.

Consider the standard supervised learning setup where the true relationship between predictors x and a response y is governed by $y = f(x) + \epsilon$. Here, $f(x)$ represents the true, yet unknown, systematic information that x provides about y, and ϵ is a random error term with zero mean ($E[\epsilon] = 0$) and variance $Var(\epsilon) = \sigma_\epsilon^2$, independent of x. We use a training dataset D to construct an estimate $\hat{f}(x)$ of the true function $f(x)$. Our objective is usually to minimize the expected error when predicting the response $y_0 = f(x_0) + \epsilon$ for a new input x_0. The expected squared prediction error at x_0, averaged over the randomness in the training data D and the noise ϵ in the new observation y_0, can be decomposed.

It is a well-established result in statistical learning theory that this expected error decomposes into three distinct components: the squared bias, the variance, and the irreducible error. Specifically, for a given test point x_0, the expected squared prediction error is given by:

$$E_{D,\epsilon}[(y_0 - \hat{f}(x_0))^2] = \underbrace{(E_D[\hat{f}(x_0)] - f(x_0))^2}_{\text{Bias}^2[\hat{f}(x_0)]} + \underbrace{E_D[(\hat{f}(x_0) - E_D[\hat{f}(x_0)])^2]}_{\text{Variance}[\hat{f}(x_0)]} + \underbrace{\sigma_\epsilon^2}_{\text{Irreducible Error}} \tag{5.3}$$

where $E_D[\cdot]$ denotes the expectation over different training datasets D.

The first term, the squared bias (Bias^2), quantifies the difference between the average prediction of the model (averaged over all possible training sets) and the true function value $f(x_0)$. A high bias suggests that the model, on average, fails to capture the true underlying relationship, often resulting from using a model that is too simplistic relative to the complexity of $f(x)$. This corresponds to the concept of underfitting.

The second term, the variance (Variance), measures the sensitivity of the model's prediction $\hat{f}(x_0)$ to variations in the training data. High variance indicates that the model changes significantly with different training datasets, suggesting it might be fitting the noise specific to the training sample rather than the underlying signal. This is characteristic of overfitting, often occurring with highly flexible models.

The third term, the irreducible error (σ_ϵ^2), represents the inherent variability in the data generating process itself. This component arises from factors not captured by the predictors or intrinsic randomness in the response. It sets a lower bound on the expected prediction error achievable by any model, regardless of its sophistication.

The bias-variance tradeoff emerges from the typical inverse relationship between bias and variance as model complexity changes. Increasing model flexibility (e.g., increasing d in polynomial regression) generally leads to a decrease in bias but an increase in variance. Conversely, reducing model complexity tends to increase bias while decreasing variance. The goal in model selection is often to find a level of complexity that optimally balances these two error sources, thereby minimizing the overall expected prediction error (Bias^2 + Variance). Understanding this tradeoff motivates the study of fundamental models like linear regression, which inherently possess lower variance, and drives the development of techniques such as regularization, designed explicitly to manage variance in more complex models.

iii. Linear Regression

Linear regression stands as a foundational technique in statistical modeling, aiming to establish a linear relationship between a set of predictor variables and a continuous response variable. It assumes that the expected value of the response variable can be expressed as a linear combination of the predictors. Given a dataset comprising n observations, where each observation i consists of p predictor variables $x_{i1}, x_{i2}, \ldots, x_{ip}$ and a response variable y_i, the general form of the multiple linear regression model is:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \epsilon_i, \quad \text{for } i = 1, \ldots, n \tag{5.4}$$

Here, β_0 is the intercept term, $\beta_1, \ldots, \beta_p$ are the regression coefficients associated with each predictor, and ϵ_i represents the random error term for the i-th observation. These error terms are typically assumed to be independent and identically distributed (i.i.d.) with zero mean ($E[\epsilon_i] = 0$) and constant variance ($Var(\epsilon_i) = \sigma^2$).

For mathematical convenience and efficient computation, the model is often expressed in matrix notation. Let $\boldsymbol{y}$ be the $n \times 1$ vector of observed responses, $\boldsymbol{\beta}$ be the $(p+1) \times 1$ vector of unknown coefficients (including the intercept β_0), and $\boldsymbol{\epsilon}$ be the $n \times 1$ vector of errors. The design matrix $\boldsymbol{X}$ is an $n \times (p+1)$ matrix where the first column consists entirely of ones (to accommodate the intercept β_0) and the subsequent columns contain the values of the p predictor variables for each observation:

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1p} \\ 1 & x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \ldots & x_{np} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}, \quad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

With this notation, the system of linear equations in (5.4) can be compactly written as:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{5.5}$$

The primary goal in linear regression is to estimate the unknown coefficient vector $\boldsymbol{\beta}$. The most common method for this estimation is Ordinary Least Squares (OLS). OLS aims to find the vector $\hat{\boldsymbol{\beta}}$ that minimizes the sum of the squared differences between the observed responses and the responses predicted by the linear model. This sum is known as the Residual Sum of Squares (RSS):

$$\text{RSS}(\boldsymbol{\beta}) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}||_2^2 = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) \tag{5.6}$$

To minimize the RSS, we take the gradient with respect to $\boldsymbol{\beta}$ and set it to zero.

$$\begin{aligned}\nabla_{\boldsymbol{\beta}}\text{RSS}(\boldsymbol{\beta}) &= \nabla_{\boldsymbol{\beta}}(\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta}) \\ &= -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta}\end{aligned}$$

Setting the gradient to zero yields the *normal equations*:

$$\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{\beta}} = \boldsymbol{X}^T\boldsymbol{y} \tag{5.7}$$

Assuming that the matrix $\boldsymbol{X}^T\boldsymbol{X}$ is invertible (i.e., the predictors are linearly independent, and $n \geq p+1$), we can solve for the OLS estimator $\hat{\boldsymbol{\beta}}_{\text{OLS}}$:

$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{5.8}$$

Under the Gauss-Markov assumptions (linearity, $E[\boldsymbol{\epsilon}] = \boldsymbol{0}$, $Var(\boldsymbol{\epsilon}) = \sigma^2\boldsymbol{I}$, and $\boldsymbol{X}$ non-stochastic or independent of $\boldsymbol{\epsilon}$), the OLS estimator is the Best Linear Unbiased Estimator (BLUE), meaning it has the minimum variance among all linear unbiased estimators.

However, practical challenges can arise. One significant issue is multicollinearity, which occurs when two or more predictor variables are highly correlated. In such cases, the columns of the design matrix $\boldsymbol{X}$ become nearly linearly dependent. Mathematically, this implies that the matrix $\boldsymbol{X}^T\boldsymbol{X}$ becomes ill-conditioned (close to singular). Consequently, its inverse $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ may have very large entries, leading to instability in the OLS estimates $\hat{\boldsymbol{\beta}}_{\text{OLS}}$. The coefficient estimates can exhibit large standard errors and high sensitivity to small changes in the data, making interpretation difficult and potentially inflating the variance component of the prediction error, as discussed in the context of the bias-variance tradeoff (Section 5 ii). Detecting and understanding the structure of these linear dependencies within the design matrix is crucial, and Singular Value Decomposition provides a powerful tool for this analysis.

1. Singular Value Decomposition

Singular Value Decomposition (SVD) is a fundamental matrix factorization technique from linear algebra with wide-ranging applications in data analysis, dimensionality reduction, and numerical stability assessment. In the context of linear regression, SVD offers valuable insights into the properties of the design matrix $\boldsymbol{X}$, particularly concerning multicollinearity and the conditioning of the OLS problem.

The SVD theorem states that any real $n \times m$ matrix $\boldsymbol{A}$ (in our case, the $n \times (p+1)$ design matrix $\boldsymbol{X}$) can be decomposed into the product of three matrices:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{5.9}$$

where:

- $\boldsymbol{U}$ is an $n \times n$ orthogonal matrix ($\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I}_n$). Its columns, $\boldsymbol{u}_i$, are the left singular vectors of $\boldsymbol{X}$.
- $\boldsymbol{\Sigma}$ is an $n \times (p+1)$ rectangular diagonal matrix. Its diagonal entries, $\sigma_1, \sigma_2, \ldots, \sigma_r$, are the singular values of $\boldsymbol{X}$, ordered non-increasingly ($\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$), where $r = \text{rank}(\boldsymbol{X}) \leq \min(n, p+1)$. All other entries of $\boldsymbol{\Sigma}$ are zero.

- $\boldsymbol{V}$ is a $(p+1) \times (p+1)$ orthogonal matrix ($\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{I}_{p+1}$). Its columns, $\boldsymbol{v}_j$, are the right singular vectors of $\boldsymbol{X}$.

The singular values σ_j represent the magnitude of the principal axes of the data cloud defined by the columns of $\boldsymbol{X}$.

The SVD is intimately related to the matrix $\boldsymbol{X}^T\boldsymbol{X}$ central to the normal equations (5.7). Substituting the SVD of $\boldsymbol{X}$ into $\boldsymbol{X}^T\boldsymbol{X}$, we get:

$$\begin{aligned} \boldsymbol{X}^T\boldsymbol{X} &= (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^T(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T) \\ &= (\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T)(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T) \\ &= \boldsymbol{V}\boldsymbol{\Sigma}^T(\boldsymbol{U}^T\boldsymbol{U})\boldsymbol{\Sigma}\boldsymbol{V}^T \\ &= \boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})\boldsymbol{V}^T \end{aligned} \tag{5.10}$$

Since $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}_n$, the result shows that (5.10) is the eigendecomposition of $\boldsymbol{X}^T\boldsymbol{X}$. The columns of $\boldsymbol{V}$ are the eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$, and the matrix $\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}$ is a $(p+1) \times (p+1)$ diagonal matrix whose diagonal entries are the squared singular values, $\sigma_1^2, \sigma_2^2, \ldots, \sigma_{p+1}^2$ (assuming $r = p+1$; otherwise, some diagonal entries are zero). These are precisely the eigenvalues of $\boldsymbol{X}^T\boldsymbol{X}$.

SVD provides a direct way to assess the conditioning of the linear regression problem. If multicollinearity exists, $\boldsymbol{X}^T\boldsymbol{X}$ is close to singular. This manifests as one or more very small eigenvalues ($\sigma_j^2 \approx 0$), which correspond to very small singular values ($\sigma_j \approx 0$). The condition number of the matrix $\boldsymbol{X}$, often defined as the ratio of the largest to the smallest non-zero singular value, $\kappa(\boldsymbol{X}) = \sigma_1/\sigma_r$, quantifies this potential instability. A large condition number signals ill-conditioning and likely multicollinearity.

Furthermore, the OLS solution (5.8) can be expressed using the SVD components. Assuming $\boldsymbol{X}^T\boldsymbol{X}$ is invertible (i.e., $r = p+1$), its inverse is $(\boldsymbol{X}^T\boldsymbol{X})^{-1} = \boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{V}^T$, where $(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}$ is a diagonal matrix with entries $1/\sigma_j^2$. The OLS solution becomes:

$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = (\boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{V}^T)(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^T\boldsymbol{y} = \boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{y} \tag{5.11}$$

This form explicitly shows that small singular values σ_j lead to large entries $1/\sigma_j^2$ in $(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}$. These large entries amplify the influence of the corresponding components of $\boldsymbol{U}^T\boldsymbol{y}$ (which involve the potentially noisy observed responses $\boldsymbol{y}$), resulting in large variance in the coefficient estimates $\hat{\boldsymbol{\beta}}_{\text{OLS}}$.

While SVD is invaluable for diagnosing multicollinearity and understanding the numerical stability of OLS, it does not, by itself, resolve these issues. However, its insights motivate and underpin techniques designed to mitigate the impact of ill-conditioning and high variance. Regularization methods, such as Ridge and LASSO regression, directly address these problems by modifying the OLS objective function to stabilize the coefficient estimates, often at the cost of introducing some bias.

2. Ridge Regression

While Ordinary Least Squares (OLS) provides an unbiased estimate under standard assumptions, its performance can degrade significantly in the presence of multicollinearity or when the number of predictors is large relative to the number of observations. In such scenarios, the OLS estimates tend to exhibit high variance, leading to poor predictive performance as explained by the bias-variance tradeoff (Section 5 ii). Furthermore, the matrix $\boldsymbol{X}^T\boldsymbol{X}$ may become ill-conditioned or even singular, making the OLS solution numerically unstable or non-unique. Ridge regression addresses these shortcomings by introducing a regularization term to the OLS cost function [25]. This technique penalizes large coefficient values, effectively shrinking them towards zero.

The Ridge regression cost function modifies the Residual Sum of Squares (RSS) by adding a penalty proportional to the squared L2-norm of the coefficient vector $\boldsymbol{\beta}$:

$$C_{\text{Ridge}}(\boldsymbol{\beta}) = ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}||_2^2 + \lambda||\boldsymbol{\beta}||_2^2 = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \tag{5.12}$$

Here, $||\cdot||_2^2$ denotes the squared Euclidean norm, and $\lambda \geq 0$ is the regularization parameter, also known as the tuning parameter. This parameter controls the strength of the penalty: a larger λ imposes a greater penalty on the magnitude of the coefficients, resulting in more shrinkage. The term $\lambda\boldsymbol{\beta}^T\boldsymbol{\beta} = \lambda\sum_{j=1}^{p}\beta_j^2$ discourages overly large parameter values, thus mitigating overfitting by reducing model complexity.

To find the Ridge coefficient estimates, $\hat{\boldsymbol{\beta}}_{\text{Ridge}}$, we minimize the cost function in Equation (5.12). This involves taking the gradient with respect to $\boldsymbol{\beta}$ and setting it to zero:

$$\begin{aligned}\nabla_{\boldsymbol{\beta}} C_{\text{Ridge}}(\boldsymbol{\beta}) &= \nabla_{\boldsymbol{\beta}}\left((\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)\\ &= \nabla_{\boldsymbol{\beta}}\left(\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta} + \lambda\boldsymbol{\beta}^T\boldsymbol{I}\boldsymbol{\beta}\right)\\ &= -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta} + 2\lambda\boldsymbol{I}\boldsymbol{\beta}\end{aligned}$$

Setting the gradient to zero yields:

$$-2\boldsymbol{X}^T\boldsymbol{y} + 2(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})\boldsymbol{\beta} = \boldsymbol{0}$$

Solving for $\boldsymbol{\beta}$ gives the Ridge regression solution:

$$\hat{\boldsymbol{\beta}}_{\text{Ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{5.13}$$

where $\boldsymbol{I}$ is the identity matrix of appropriate dimension ($p \times p$, where p is the number of predictors). Comparing this to the OLS solution $\hat{\boldsymbol{\beta}}_{OLS} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$, the key difference is the addition of $\lambda\boldsymbol{I}$ to the $\boldsymbol{X}^T\boldsymbol{X}$ matrix. For $\lambda > 0$, the matrix $(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})$ is always invertible, even if $\boldsymbol{X}^T\boldsymbol{X}$ is singular. This addition improves the conditioning of the matrix, leading to a numerically stable solution.

From a statistical perspective, the introduction of the penalty term affects the properties of the estimator. Assuming the linear model $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta}_{\text{true}} + \boldsymbol{\epsilon}$ with $E[\boldsymbol{\epsilon}] = \boldsymbol{0}$ and $Var(\boldsymbol{\epsilon}) = \sigma^2\boldsymbol{I}$ holds, the expectation of the Ridge estimator is:

$$\begin{aligned}\mathbb{E}[\hat{\boldsymbol{\beta}}_{\text{Ridge}}] &= \mathbb{E}[(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}]\\ &= (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\mathbb{E}[\boldsymbol{y}]\\ &= (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta}_{\text{true}}\end{aligned} \tag{5.14}$$

Since $(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{X} \neq \boldsymbol{I}$ for $\lambda > 0$, the Ridge estimator is biased, i.e., $\mathbb{E}[\hat{\boldsymbol{\beta}}_{\text{Ridge}}] \neq \boldsymbol{\beta}_{\text{true}}$. This bias increases with λ. However, this induced bias is accepted in exchange for a reduction in variance. The covariance matrix of the Ridge estimator is given by:

$$\begin{aligned}\text{Var}(\hat{\boldsymbol{\beta}}_{\text{Ridge}}) &= \text{Var}((\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y})\\ &= (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\text{Var}(\boldsymbol{y})\boldsymbol{X}((\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1})^T\\ &= \sigma^2(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\end{aligned} \tag{5.15}$$

The variance of the Ridge estimator decreases as λ increases. For $\lambda > 0$, the variance is generally lower than that of the OLS estimator. The choice of λ therefore controls the bias-variance tradeoff for the Ridge model; practical methods like cross-validation are typically employed to select an optimal λ that minimizes the prediction error. It is noteworthy that while Ridge shrinks coefficients towards zero, it rarely sets them exactly to zero unless $\lambda \to \infty$. This implies that Ridge performs coefficient shrinkage but does not perform variable selection in the sense of eliminating predictors entirely.

3. LASSO Regression

An alternative regularization technique that addresses the limitations of OLS and offers different properties compared to Ridge regression is the Least Absolute Shrinkage and Selection Operator (LASSO) [53]. Similar to

Ridge, LASSO adds a penalty term to the OLS cost function, but instead of the L2-norm, it utilizes the L1-norm of the coefficient vector.

The LASSO cost function is defined as:

$$C_{\text{LASSO}}(\boldsymbol{\beta}) = ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}||_2^2 + \lambda||\boldsymbol{\beta}||_1 \tag{5.16}$$

where $||\boldsymbol{\beta}||_1 = \sum_{j=1}^{p} |\beta_j|$ is the L1-norm of the coefficient vector, and $\lambda \geq 0$ is the regularization parameter controlling the penalty strength. The critical difference lies in the nature of the L1 penalty compared to the L2 penalty used in Ridge. Geometrically, the L1 constraint corresponds to a diamond shape (or hyperoctahedron in higher dimensions), while the L2 constraint corresponds to a sphere (or hypersphere). The "corners" of the L1 constraint region make it more likely that the optimization process will find solutions where some coefficients β_j are exactly zero.

This property leads to the most distinctive feature of LASSO: its ability to perform automatic variable selection. By setting some coefficients precisely to zero, LASSO effectively removes the corresponding predictors from the model, resulting in a sparser and potentially more interpretable model, especially when dealing with high-dimensional data where many predictors might be irrelevant.

Finding the LASSO coefficient estimates $\hat{\boldsymbol{\beta}}_{\text{LASSO}}$ requires minimizing the cost function in Equation (5.16). However, the L1-norm term $||\boldsymbol{\beta}||_1$ is not differentiable when any $\beta_j = 0$. Consequently, a simple closed-form solution like that for OLS or Ridge does not exist. While we can express the condition for the minimum using subgradients, it does not lead to a direct analytical solution. The optimality condition can be informally written by considering the gradient where possible:

$$\frac{\partial C_{\text{LASSO}}}{\partial \beta_j} = -2(\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}))_j + \lambda \cdot \text{sgn}(\beta_j) = 0 \quad \text{for } \beta_j \neq 0$$

where $\text{sgn}(\cdot)$ is the sign function. More formally, using subdifferential calculus, the condition is $\mathbf{0} \in -2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}_{\text{LASSO}}) + \lambda\partial||\hat{\boldsymbol{\beta}}_{\text{LASSO}}||_1$, where $\partial||\cdot||_1$ is the subgradient of the L1 norm. Due to the complexity arising from the non-differentiability, the LASSO solution must be computed using iterative numerical optimization algorithms. Common algorithms include coordinate descent and least angle regression (LARS). The specifics of such numerical solvers fall outside the scope of this theoretical overview but are crucial for practical implementation.

Similar to Ridge, LASSO introduces bias into the coefficient estimates in order to achieve a potentially significant reduction in variance, thereby improving overall prediction accuracy. The exact bias and variance expressions are more complex than for Ridge. The choice of λ again controls the bias-variance tradeoff and the degree of sparsity in the resulting model, typically determined through cross-validation. LASSO is particularly favored in scenarios where feature selection is desired or when dealing with a large number of predictors, some of which are expected to be unimportant.

iv. Logistic Regression

Logistic regression is a supervised learning method used for classification tasks, particularly binary classification. Unlike linear regression, which models a continuous target variable, logistic regression models the probability that a given input vector $\mathbf{x}$ belongs to a particular class, typically labeled 0 or 1.

1. Logistic Function

Let $\mathbf{x} = (x_1, x_2, \ldots, x_t)$ be an input vector of length t, $\mathbf{y} \in \{0, 1\}$ be a binary label, and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_t)$ a vector of model coefficients. A linear model would approximate $\mathbf{y} \approx \mathbf{x}^T\boldsymbol{\beta}$, but such an approach is inadequate for classification since the output is not constrained to the interval $[0, 1]$. To remedy this, we apply the logistic

(sigmoid) function:

$$L(t) = \frac{1}{1 + e^{-t}}. \tag{5.17}$$

Applying this function to the linear combination $\mathbf{x}^T\boldsymbol{\beta}$ ensures that outputs are confined to the interval $(0, 1)$, allowing a probabilistic interpretation:

$$\mathbb{P}(\mathbf{y} = 1 \mid \mathbf{x}, \boldsymbol{\beta}) = L(\mathbf{x}^T\boldsymbol{\beta}), \qquad \mathbb{P}(\mathbf{y} = 0 \mid \mathbf{x}, \boldsymbol{\beta}) = 1 - L(\mathbf{x}^T\boldsymbol{\beta}). \tag{5.18}$$

A decision is typically made by comparing the predicted probability to a threshold (e.g., 0.5), assigning class 1 if the probability exceeds the threshold and class 0 otherwise.

2. Cost Function

Given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ of size n, the model parameters $\boldsymbol{\beta}$ can be estimated using maximum likelihood estimation [?] . The likelihood of the data given the parameters is:

$$\mathbb{P}(\mathcal{D} \mid \boldsymbol{\beta}) = \prod_{i=1}^{n} [\mathbb{P}(y_i = 1 \mid \mathbf{x}_i, \boldsymbol{\beta})]^{y_i} [1 - \mathbb{P}(y_i = 1 \mid \mathbf{x}_i, \boldsymbol{\beta})]^{1-y_i}. \tag{5.19}$$

Taking the negative logarithm yields the negative log-likelihood, which serves as the cost function for logistic regression:

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left[y_i \log(\mathbb{P}(y_i = 1 \mid \mathbf{x}_i, \boldsymbol{\beta})) + (1 - y_i) \log(1 - \mathbb{P}(y_i = 1 \mid \mathbf{x}_i, \boldsymbol{\beta})) \right]. \tag{5.20}$$

Substituting in the logistic function from Eq. (5.18), we obtain:

$$\begin{aligned} C(\boldsymbol{\beta}) &= -\sum_{i=1}^{n} \left[y_i \log\big(L(\mathbf{x}_i^T\boldsymbol{\beta})\big) + (1 - y_i) \log\big(1 - L(\mathbf{x}_i^T\boldsymbol{\beta})\big) \right] \\ &= -\sum_{i=1}^{n} \left[y_i \mathbf{x}_i^T\boldsymbol{\beta} - \log\Big(1 + e^{\mathbf{x}_i^T\boldsymbol{\beta}}\Big) \right]. \end{aligned} \tag{5.21}$$

This is recognized as the cross-entropy loss, a convex function, which guarantees that any local minimum is also a global minimum.

3. Optimization of Parameters

To find the optimal parameters $\boldsymbol{\beta}$ that minimize the cost function $C(\boldsymbol{\beta})$, we compute its gradient:

$$\frac{\partial C}{\partial \boldsymbol{\beta}} = -\sum_{i=1}^{n} \left[y_i \mathbf{x}_i - \mathbf{x}_i L(\mathbf{x}_i^T\boldsymbol{\beta}) \right] = \sum_{i=1}^{n} \mathbf{x}_i \left(L(\mathbf{x}_i^T\boldsymbol{\beta}) - y_i \right). \tag{5.22}$$

This can be compactly written in matrix form. Let $\mathbf{X} \in \mathbb{R}^{n \times t}$ be the design matrix with rows $\mathbf{x}_i^T$, $\mathbf{y} \in \mathbb{R}^n$ the vector of targets, and $\mathbf{p} \in \mathbb{R}^n$ the vector of predicted probabilities. Then:

$$\frac{\partial C}{\partial \boldsymbol{\beta}} = \mathbf{X}^T(\mathbf{p} - \mathbf{y}). \tag{5.23}$$

Gradient descent or its stochastic variants (e.g., SGD, mini-batch SGD) can then be employed to iteratively update $\boldsymbol{\beta}$ until convergence.

v. Neural Networks

Moving beyond linear models and their direct extensions like polynomial or logistic regression, Artificial Neural Networks (ANNs), often simply referred to as neural networks, represent a powerful class of models inspired by the structure and function of biological neural systems. These models excel at learning complex, non-linear relationships directly from data, making them highly effective for a wide range of tasks including sophisticated classification and regression problems [21]. While logistic regression can be viewed as a very simple neural network (a single neuron with a sigmoid activation), ANNs typically consist of multiple layers of interconnected processing units, or neurons, organized into an input layer, one or more hidden layers, and an output layer.

1. The Universal Approximation Theorem

A key theoretical underpinning for the power of neural networks is the Universal Approximation Theorem (UAT). This theorem, in various forms (e.g., [9, 26, 27]), essentially states that a feedforward neural network with a single hidden layer containing a finite number of neurons and a suitable non-linear activation function can approximate any continuous function on compact subsets of ($\mathbb{R}^n$) to any desired degree of accuracy. The "suitable" activation functions are typically non-polynomial; common examples like sigmoid or ReLU satisfy this condition.

This remarkable property implies that, in theory, even a relatively simple neural network architecture has the potential to represent an incredibly wide range of complex functions. It's important to note that the UAT is an existence theorem: it guarantees that such a network exists, but it doesn't specify how to find the optimal weights and biases, nor does it define the exact number of hidden neurons required for a particular function or degree of accuracy. Furthermore, while a single hidden layer is theoretically sufficient, deeper architectures (networks with multiple hidden layers) are often found to be more efficient in practice, learning complex hierarchical features with fewer parameters than a wide, shallow network might require for the same task [21]. The challenge, then, lies not in the representational capacity of the network, but in the practical aspects of training it effectively to find a good approximation and ensuring it generalizes well to unseen data.

The fundamental idea is function approximation: a neural network learns a mapping $f : \mathcal{X} \to \mathcal{Y}$ from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$. The input layer receives the raw data (features), analogous to the predictor variables $\boldsymbol{x}$. Each neuron in subsequent layers computes a weighted sum of its inputs from the previous layer, adds a bias term, and then applies a non-linear activation function. The hidden layers transform the input data through successive non-linear mappings, allowing the network to learn hierarchical representations of the data. The final output layer produces the network's prediction, $\hat{\boldsymbol{y}}$, which can be a continuous value (for regression) or a probability distribution over classes (for classification). The parameters of the network, namely the weights connecting neurons between layers and the biases associated with each neuron, are learned from the training data through an optimization process typically involving backpropagation and gradient descent. The process by which the network computes an output from an input, given a set of parameters, is known as the forward phase.

2. Forward Phase

The forward phase, or forward propagation, describes the computation flow through the neural network from the input layer to the output layer. It calculates the network's output for a given input instance $\boldsymbol{x}$ based on the current values of its weights and biases. Let us consider a feedforward neural network with L layers, where layer 0 is the input layer and layer L is the output layer. Layers 1 through $L-1$ are the hidden layers.

Let $\boldsymbol{a}^{(l)}$ denote the vector of activations (outputs) from layer l. The activation vector of the input layer is simply the input feature vector, $\boldsymbol{a}^{(0)} = \boldsymbol{x}$. For each subsequent layer $l = 1, 2, \ldots, L$, the computation proceeds in two steps:

First, a linear transformation is applied to the activations from the previous layer, $\boldsymbol{a}^{(l-1)}$. This computes the

weighted sum, or pre-activation value, $\boldsymbol{z}^{(l)}$ for each neuron in layer l:

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{5.24}$$

Here, $\boldsymbol{W}^{(l)}$ is the weight matrix for layer l, where $W_{jk}^{(l)}$ represents the weight connecting the k-th neuron in layer $l-1$ to the j-th neuron in layer l. $\boldsymbol{b}^{(l)}$ is the bias vector for layer l, with $b_j^{(l)}$ being the bias added to the j-th neuron in layer l. If layer $l-1$ has n_{l-1} neurons and layer l has n_l neurons, then $\boldsymbol{W}^{(l)}$ is an $n_l \times n_{l-1}$ matrix, and $\boldsymbol{b}^{(l)}$ and $\boldsymbol{z}^{(l)}$ are $n_l \times 1$ vectors.

Second, a non-linear activation function, $g^{(l)}$, is applied element-wise to the pre-activation vector $\boldsymbol{z}^{(l)}$ to produce the activation vector $\boldsymbol{a}^{(l)}$ for layer l:

$$\boldsymbol{a}^{(l)} = g^{(l)}(\boldsymbol{z}^{(l)}) \tag{5.25}$$

This activation vector $\boldsymbol{a}^{(l)}$ then serves as the input to the next layer, $l+1$. This process is repeated iteratively from $l=1$ up to the final layer L. The output of the network is the activation vector of the last layer, $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(L)}$. The choice of the activation function $g^{(l)}$ is critical for the network's ability to model non-linear phenomena.

3. Activation Functions

Activation functions play a crucial role in neural networks by introducing non-linearity into the model. Without non-linear activation functions between layers, a multi-layer neural network would simply compute a series of linear transformations, which could be collapsed into a single equivalent linear transformation. Such a network would have no representational advantage over a simple linear or logistic regression model, regardless of its depth. As highlighted by the Universal Approximation Theorem (Section 5 v 1), this non-linearity is essential for the network's ability to learn complex mappings and approximate arbitrary continuous functions.

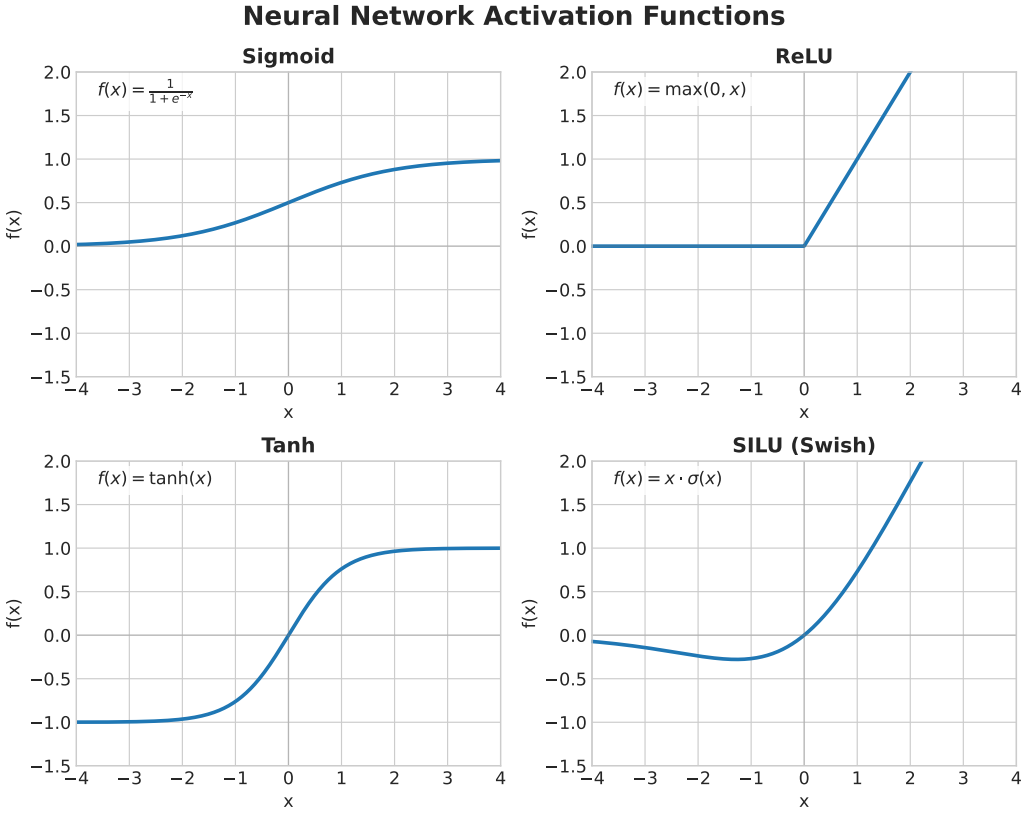

Figure 5. Some well-known activation functions. Sigmoid, ReLU, Tanh and SILU

Several activation functions are commonly used. The sigmoid function, ($\sigma(z) = 1/(1 + e^{-z})$), was historically popular, particularly in output layers for binary classification problems as it squashes its input into the range (0, 1), interpretable as a probability. However, it suffers from the vanishing gradient problem, where gradients become extremely small for large positive or negative inputs, hindering learning in deep networks.

The hyperbolic tangent function, $\tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$, is similar to the sigmoid but maps inputs to the range (-1, 1). Its zero-centered output can sometimes lead to faster convergence compared to sigmoid. However, it also suffers from the vanishing gradient problem for saturated inputs.

Currently, the Rectified Linear Unit (ReLU) function, defined as $g(z) = \max(0, z)$, is one of the most widely used activation functions in hidden layers [39]. It is computationally efficient and does not saturate for positive inputs, which helps alleviate the vanishing gradient problem and often leads to faster training. A potential issue is the "dying ReLU" problem, where neurons can become stuck in a state where they always output zero if their input consistently falls below zero during training, effectively preventing them from learning further. Variants like Leaky ReLU ($g(z) = \max(\alpha z, z)$ for a small $\alpha > 0$) or Parametric ReLU (PReLU) aim to address this.

For multi-class classification problems, the softmax function is typically used in the output layer. It transforms a vector of arbitrary real values into a probability distribution over K classes, ensuring that the outputs are non-negative and sum to one. Given the pre-activation vector $\boldsymbol{z}^{(L)}$ for the output layer, the j-th output activation (probability of class j) is computed as:

$$a_j^{(L)} = \text{softmax}(\boldsymbol{z}^{(L)})_j = \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{K} e^{z_k^{(L)}}} \quad \text{for } j = 1, \ldots, K \tag{5.26}$$

The choice of activation function depends on the specific layer (hidden vs. output), the task (regression vs. classification), and empirical performance. The combination of linear transformations and non-linear activations across multiple layers endows neural networks with their expressive power. Once the forward phase computes the prediction $\hat{\boldsymbol{y}}$, the next step in training the network is to evaluate the error and determine how to adjust the parameters $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ to reduce this error, which is achieved through backward propagation.

4. Backward Propagation

Training a neural network involves finding the optimal set of weights $\boldsymbol{W}^{(l)}$ and biases $\boldsymbol{b}^{(l)}$ for all layers $l = 1, \ldots, L$ that minimize a predefined loss function $J(\boldsymbol{W}, \boldsymbol{b})$, which quantifies the discrepancy between the network's predictions $\hat{\boldsymbol{y}}$ and the true target values $\boldsymbol{y}$. Common loss functions include Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss for classification tasks. The minimization is typically performed using gradient-based optimization algorithms. The core challenge lies in efficiently computing the gradient of the loss function with respect to potentially millions of parameters in the network.

Backward propagation, often shortened to backpropagation, is an algorithm that efficiently computes these gradients by systematically applying the chain rule of calculus [48]. It works by propagating the error signal backward through the network, starting from the output layer and moving towards the input layer.

The algorithm proceeds as follows: First, the forward phase (Equations (5.24) and (5.25)) is performed for an input instance (or a batch of instances) to compute the activations $\boldsymbol{a}^{(l)}$ and pre-activations $\boldsymbol{z}^{(l)}$ for all layers, including the final prediction $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(L)}$.

Next, the error term $\boldsymbol{\delta}^{(l)}$ is computed for each layer, starting from the output layer L. This term represents the partial derivative of the loss function J with respect to the pre-activation values $\boldsymbol{z}^{(l)}$ of that layer, i.e., $\boldsymbol{\delta}^{(l)} \coloneqq \frac{\partial J}{\partial \boldsymbol{z}^{(l)}}$. For the output layer L, the error term is calculated based on the gradient of the loss function with respect to the output activations $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(L)}$ and the derivative of the activation function $g^{(L)}$:

$$\boldsymbol{\delta}^{(L)} = \nabla_{\hat{\boldsymbol{y}}} J \odot g'^{(L)}(\boldsymbol{z}^{(L)}) \tag{5.27}$$

where $\odot$ denotes the element-wise product (Hadamard product), and $g'^{(L)}(\boldsymbol{z}^{(L)})$ is the element-wise derivative of the output activation function evaluated at $\boldsymbol{z}^{(L)}$. Note that for common combinations like Softmax output and Cross-Entropy loss, this expression simplifies considerably to $\boldsymbol{\delta}^{(L)} = \hat{\boldsymbol{y}} - \boldsymbol{y}$.

Then, the error is propagated backward to compute $\boldsymbol{\delta}^{(l)}$ for layers $l = L-1, L-2, \ldots, 1$. The error at layer l depends on the error $\boldsymbol{\delta}^{(l+1)}$ from the subsequent layer $l+1$ and the weights $\boldsymbol{W}^{(l+1)}$ connecting layer l to layer $l+1$:

$$\boldsymbol{\delta}^{(l)} = ((\boldsymbol{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}) \odot g'^{(l)}(\boldsymbol{z}^{(l)}) \tag{5.28}$$

This equation applies the chain rule: the error propagates back through the weights and then through the non-linear activation function of the current layer.

Finally, once the error terms $\boldsymbol{\delta}^{(l)}$ are computed for all layers, the gradients of the loss function with respect to the network parameters $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ can be calculated:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\boldsymbol{a}^{(l-1)})^T \tag{5.29}$$

$$\frac{\partial J}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \tag{5.30}$$

When working with mini-batches of data, these gradients are typically averaged over the instances in the batch.

The backpropagation algorithm yields the gradients $\{\frac{\partial J}{\partial \boldsymbol{W}^{(l)}}, \frac{\partial J}{\partial \boldsymbol{b}^{(l)}}\}$ for all learnable parameters in the network. These gradients are the essential input for optimization algorithms, such as Gradient Descent and its variants, which update the parameters iteratively to minimize the loss function and thereby train the neural network. The specifics of these optimization algorithms are discussed in the following section.

vi. Optimization Algorithms

The process of training machine learning models, particularly complex ones like neural networks, fundamentally relies on optimization. The objective is typically to find the set of model parameters, denoted generically as $\boldsymbol{\theta}$ (which encompasses all weights $\boldsymbol{W}^{(l)}$ and biases $\boldsymbol{b}^{(l)}$ in the context of neural networks), that minimizes a predefined loss function $J(\boldsymbol{\theta})$. This loss function measures the discrepancy between the model's predictions and the true target values over the training dataset. As established in Section 5 v 4, algorithms like backpropagation provide an efficient means to compute the gradient of the loss function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This gradient indicates the direction of steepest ascent of the loss function; thus, moving in the opposite direction corresponds to the steepest descent. Gradient-based optimization methods leverage this information to iteratively update the parameters, aiming to converge towards a minimum of the loss function. Several variants of gradient-based optimization exist, each with different characteristics regarding computational efficiency, convergence properties, and memory requirements.

1. Gradient Descent

The most fundamental gradient-based optimization algorithm is Gradient Descent (GD), often referred to as Batch Gradient Descent to distinguish it from its variants. The core idea is to take steps proportional to the negative of the gradient of the loss function, evaluated over the entire training dataset. Starting from an initial parameter vector $\boldsymbol{\theta}_0$, the parameters are updated iteratively according to the rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) \tag{5.31}$$

where t is the iteration index, η is the learning rate (a positive scalar hyperparameter that controls the step size), and $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ is the gradient of the total loss computed using all n training examples: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}_t)$, where J_i is the loss for the i-th example.

Batch Gradient Descent is guaranteed to converge to a local minimum of the loss function (or a global minimum if the function is convex), provided the learning rate η is chosen appropriately (small enough). The gradient calculated using the entire dataset provides an accurate estimate of the true gradient, leading to smooth convergence. However, this method suffers from significant drawbacks, particularly for large datasets prevalent in modern machine learning. Computing the gradient requires processing every single training example in each iteration, making each update computationally very expensive. Furthermore, the entire dataset must typically be loaded into memory to compute the gradient, which can be infeasible. Due to its slow updates and potential to get trapped in suboptimal local minima, Batch GD is often impractical for training large-scale models like deep neural networks.

2. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) offers a computationally efficient alternative to Batch Gradient Descent, addressing its scalability issues. Instead of computing the gradient using the entire dataset, SGD updates the parameters using the gradient computed from only a small subset of the data at each step. In its purest form, SGD uses a single randomly selected training example $(\boldsymbol{x}_i, y_i)$ to estimate the gradient at each iteration t:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}_t) \tag{5.32}$$

where $J_i(\boldsymbol{\theta}_t)$ is the loss calculated for the single example i.

More commonly in practice, a variant called Mini-batch Stochastic Gradient Descent is used. This approach strikes a balance between the accuracy of Batch GD and the speed of pure SGD. It computes the gradient based on a small, randomly selected batch B of training examples ($|B|$ is the mini-batch size, typically ranging from tens to a few hundreds):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{1}{|B|} \sum_{i \in B} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}_t) \tag{5.33}$$

The mini-batch gradient provides a less noisy estimate of the true gradient compared to the single-sample estimate, while remaining much faster to compute than the full-batch gradient.

SGD and its mini-batch variant offer significant advantages. The updates are much more frequent and computationally cheaper, leading to faster progress, especially in the initial stages of training. The reduced memory footprint is also crucial for large datasets. Furthermore, the inherent noise in the gradient estimates (due to sampling) can help the optimization process escape shallow local minima and navigate saddle points more effectively than Batch GD. However, this noise also means that the parameter updates fluctuate, and SGD typically does not converge to the exact minimum but rather oscillates around it. Careful tuning of the learning rate, often involving decay schedules where η decreases over time, is necessary to achieve good convergence. Despite the noisy updates, SGD is the workhorse algorithm for training large-scale machine learning models, particularly deep neural networks.

3. Adding Momentum

While SGD provides computational benefits, its noisy updates can cause oscillations, particularly in directions where the loss surface curves steeply, slowing down convergence. The Momentum method [45] is an enhancement designed to accelerate SGD convergence and dampen these oscillations. It introduces a "velocity" vector $\boldsymbol{v}_t$ that accumulates an exponentially decaying moving average of past gradients. The intuition is analogous to physical momentum: the parameter update gains inertia in directions where the gradient consistently points, allowing it to move faster along shallow ravines and pass through small local minima or saddle points.

The update rule for SGD with momentum involves two steps at each iteration t: First, the velocity vector is updated:

$$\boldsymbol{v}_{t+1} = \gamma \boldsymbol{v}_t + \eta \nabla_{\boldsymbol{\theta}} J_{\text{batch}}(\boldsymbol{\theta}_t) \tag{5.34}$$

Second, the parameters are updated using the velocity:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{v}_{t+1} \tag{5.35}$$

Here, $\nabla_{\boldsymbol{\theta}} J_{\text{batch}}(\boldsymbol{\theta}_t)$ represents the gradient computed on the current mini-batch (or single sample). The hyperparameter γ is the momentum coefficient, typically set to a value like 0.9 or higher. It controls the extent to which past gradients influence the current update; a value close to 1 gives more weight to past gradients. The initial velocity $\boldsymbol{v}_0$ is usually set to zero.

By accumulating gradients over time, the momentum term effectively smooths out the update directions. If successive gradients point in similar directions, the velocity builds up, leading to larger steps and faster convergence along consistent slopes. Conversely, if gradients frequently change direction (oscillations), the momentum term helps to cancel out these conflicting updates, resulting in smaller steps in those directions. This makes the optimization process more stable and often significantly faster than standard SGD. A popular refinement of this idea is Nesterov Accelerated Gradient (NAG) [40], which calculates the gradient at a point slightly ahead in the direction of the current velocity, often leading to better performance. Momentum is a widely used technique that forms the basis for more advanced adaptive optimization algorithms.

4. ADAM

While Stochastic Gradient Descent with Momentum addresses the issues of noisy gradients and navigating challenging loss landscapes, it still relies on a single, manually tuned learning rate η applied uniformly to all parameters. However, different parameters might benefit from different learning rates depending on the geometry of the loss surface and the frequency of feature updates. Adaptive learning rate methods aim to automatically adjust the learning rate for each parameter individually. The Adaptive Moment Estimation (ADAM) algorithm [30] is arguably one of the most popular and effective adaptive optimization algorithms currently used, particularly for training deep neural networks. It combines the ideas of momentum (using moving averages of the gradient) and RMSprop [54] (using moving averages of the squared gradient to adapt the learning rate).

ADAM maintains two exponentially decaying moving averages for each parameter θ_j: the first moment estimate $\boldsymbol{m}_t$ (an estimate of the mean of the gradients, similar to the velocity term in momentum) and the second moment estimate $\boldsymbol{v}_t$ (an estimate of the uncentered variance of the gradients). Let $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} J_{\text{batch}}(\boldsymbol{\theta}_t)$ be the gradient of the loss with respect to the parameters $\boldsymbol{\theta}$ at iteration t, computed using the current mini-batch. The moving averages are updated as follows:

$$\boldsymbol{m}_{t+1} = \beta_1 \boldsymbol{m}_t + (1 - \beta_1)\boldsymbol{g}_t \tag{5.36}$$

$$\boldsymbol{v}_{t+1} = \beta_2 \boldsymbol{v}_t + (1 - \beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t \tag{5.37}$$

Here, β_1 and β_2 are exponential decay rates for the first and second moment estimates, respectively. These are hyperparameters typically close to 1 (e.g., default values often suggested are $\beta_1 = 0.9$ and $\beta_2 = 0.999$). The operation $\boldsymbol{g}_t \odot \boldsymbol{g}_t$ denotes element-wise squaring of the gradient vector.

The moment estimates $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ are initialized as zero vectors. As a result, they are biased towards zero, especially during the initial stages of training when the decay rates β_1 and β_2 are close to 1. To counteract this initialization bias, ADAM computes bias-corrected moment estimates, denoted by $\hat{\boldsymbol{m}}_{t+1}$ and $\hat{\boldsymbol{v}}_{t+1}$:

$$\hat{\boldsymbol{m}}_{t+1} = \frac{\boldsymbol{m}_{t+1}}{1 - \beta_1^{t+1}} \tag{5.38}$$

$$\hat{\boldsymbol{v}}_{t+1} = \frac{\boldsymbol{v}_{t+1}}{1 - \beta_2^{t+1}} \tag{5.39}$$

where $t+1$ in the exponents β_1^{t+1} and β_2^{t+1} represents the iteration number (starting from $t=0$). As t increases, the bias correction factors $1-\beta_1^{t+1}$ and $1-\beta_2^{t+1}$ approach 1, and the correction becomes negligible.

Finally, the parameters $\boldsymbol{\theta}$ are updated using these bias-corrected estimates. The update rule effectively scales the learning rate for each parameter based on the history of its gradients:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{\hat{\boldsymbol{m}}_{t+1}}{\sqrt{\hat{\boldsymbol{v}}_{t+1}} + \epsilon} \tag{5.40}$$

Here, η is the master learning rate (step size), and ϵ is a small constant (e.g., 10^{-8}) added to the denominator for numerical stability, preventing division by zero in cases where $\hat{\boldsymbol{v}}_{t+1}$ might be close to zero. The division by $\sqrt{\hat{\boldsymbol{v}}_{t+1}}$ is performed element-wise. This term adaptively scales the update for each parameter: parameters associated with large gradients (large second moment estimate $\hat{v}_j$) will have their effective learning rate reduced, while parameters associated with small or infrequent gradients will have their effective learning rate increased.

ADAM combines the benefits of adaptive learning rates (like RMSprop) with momentum. It is computationally efficient, requires little memory overhead, is relatively insensitive to the choice of hyperparameters (though tuning η can still be important), and generally performs well in practice across a wide range of problems, particularly in deep learning contexts. Its ability to compute individual adaptive learning rates for different parameters makes it well-suited for problems with sparse gradients or noisy objectives. While ADAM has become a standard choice, research continues to explore its theoretical convergence properties and potential issues, such as generalization performance compared to simpler methods like SGD with momentum in some scenarios, leading to variants like AdamW [33] which modifies the handling of weight decay.

The discussion of these optimization algorithms, from the foundational Gradient Descent to the widely used ADAM, highlights the crucial role they play in enabling the training of complex models by efficiently navigating high-dimensional parameter spaces. The gradients computed via methods like backpropagation fuel these algorithms, allowing models to learn intricate patterns from data. Having covered these core machine learning concepts, we now turn our attention to specialized architectures and techniques, such as Physics-Informed Neural Networks.

Chapter 6

6. Physics-informed neural networks

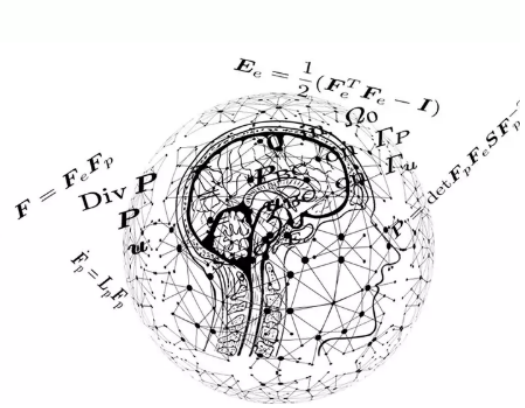

Figure 6. We introduce physics-informed neural networks, neural networks that are trained to solve supervised learning tasks while respecting any given law of physics described by general nonlinear partial differential equations.[1 46]

> "The reward of the young scientist is the emotional thrill of being the first person in the history of the world to see something or to understand something. Nothing can compare with that experience.[44]
>
> Cecilia Payne-Gaposchkin"

We begin by introducing physics-informed neural networks and the theory behind them.

i. Introduction

Physics-informed neural networks, commonly referred to as PINNs, are feedforward neural networks [48] which incorporate physical laws described as partial differential equations (PDEs) in order to more efficiently train. PINNs are still quite new but have shown great promise for both solving the Backward and Forward-problem for PDEs. The Backward-problem is; given points in our domain to determine the hyperparameters of the PDE which the points stem from. The Forward-problem is to determine what the solution of the PDE is in its domain. This article will focus on solving the Forward-problem, for more on the Backwards-problem see [22], [14], and references therein.

A feedforward neural network $\theta = \{W_i, b_i\}_{i=1}^{k}$ consists of a set of weights W and biases b. An input is transformed into an output using θ, by multiplying it with the first weight and and adding the first bias, after which one commonly applies a non-linear activation function, before sending the output of this into the second layer of weight and bias. Repeating this process, layer by layer, one ends up with an output, and using the back-propagation algorithm [31], it is then possible to tune the weights and biases in the network, to get a more accurate prediction, referred to as training the network.

ii. Cost function

Writing a PDE in the general form, with the problem and its boundary condition being described as

$$\mathcal{L}(u) = f, \;\; \mathbf{x} \in \Omega, \qquad u(\mathbf{x}) = g \;\forall \mathbf{x} \in \partial\Omega. \tag{6.1}$$

Where $\mathcal{L}$ is a nonlinear operator, $\Omega \subseteq \mathbb{R}^d$ is the domain of the PDE with $\partial\Omega$ being the boundary, $\mathbf{x} \in \mathbb{R}^d$ the input of our PDE, $f, g : \mathbb{R}^d \to \mathbb{R}$ are some functions and $u(\mathbf{x})$ is the true solution. In addition, it is also necessary to specify an initial solution if the equation is time-dependent.

Examples of the nonlinear operator $\mathcal{L}$, is

$$\mathcal{L}(u) = u'', \qquad \mathcal{L}(u) = u^2 + u', \qquad \mathcal{L}(u) = \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}. \tag{6.2}$$

Having described the problem we want to solve, using our network θ described in section 6 i, we can get a prediction u_θ of u where u is the solution of equation (6.1). This is quite useful, but in order to be able to train our network, we need a way of measuring the error between u_θ and u referred to as the cost function. Discretizing equation (6.1) we can achieve this, and get the following measures of error for our approximation of u

$$\text{MSE}_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u_\theta(\mathbf{x}_i^b) - u(\mathbf{x}_i^b)|^2, \qquad \text{MSE}_\mathcal{L} = \frac{1}{N_\mathcal{L}}\sum_{i=1}^{N_\mathcal{L}} |\mathcal{L}(u_\theta)(\mathbf{x}_i^\mathcal{L}) - f(\mathbf{x}_i^\mathcal{L})|^2. \tag{6.3}$$

Where $\{\mathbf{x}_i^b\}_{i=1}^{N_u}$ are points sampled from $\partial\Omega$, and $\{\mathbf{x}_i^\mathcal{L}\}_{i=1}^{N_\mathcal{L}}$ are points sampled from $\Omega \backslash \partial\Omega$. It is important to mention that we are able to compute $\mathcal{L}(u_\theta)$ due to automatic differentiation, see [?] . Combining these two measures of errors we get the following cost function for our PINN

$$\text{MSE} = \lambda_u \text{MSE}_u + \lambda_\mathcal{L} \text{MSE}_\mathcal{L}. \tag{6.4}$$

Where $\lambda_u, \lambda_\mathcal{L} \in \mathbb{R}_+$, are constants determining how much the error on the boundary versus the error in interior points should affect the training. Having determined the loss function of our network, it is thus possible to train it, using the back-propagation algorithm.

iii. Regularisation

When training our network it is often beneficial to add some type of regularisation to our weights. This means our network automatically punishes too large weights, which is beneficial due to it making it hard for one attribute of our input to dominate the prediction. The easiest way to do this is to rewrite our cost function to

$$\text{MSE} = \lambda_u \text{MSE}_u + \lambda_\mathcal{L} \text{MSE}_\mathcal{L} + \lambda ||W||_2. \tag{6.5}$$

Where λ determines how much regularisation there is, and $||\cdot||_2$ is the regular L_2-norm.

iv. Useful properties of PINNs

Having introduced the theory behind PINNs, we deem it also important to mention some of the attributes which make them a promising new alternative for solving PDEs.

- After training our network, it is possible to evaluate our solution at any point, without having to reprocess our solution using for example interpolation.
- PINNs utilize the formulation of the PDE in addition to its boundary conditions, thus making it quite simple to set up once the main framework for the feedforward neural network is in place.
- It is not required to generate training data beforehand, to train PINNs. Instead, through its use of the PDE and randomly sampling the domain. It can generate it's own training data, by predicting randomly sampled points.

v. Activation functions

As mentioned previously an important aspect of PINNs, and all feedforward neural networks for that matter, is the choice of activation function. For this paper, we will consider the following activation functions.

1. SiLU

Sigmoid Linear Units, shortened SiLU, is given as

$$h(x) = \frac{x}{1+e^{-x}}, \qquad h'(x) = \frac{e^{-x}(1+e^{-x}+x)}{(1+e^{-x})^2}. \tag{6.6}$$

The main strength of using SiLU is that it combines the properties of both linear and non-linear activation functions, allowing for smooth and non-monotonic transformations. This smoothness helps mitigate issues like the vanishing gradient problem [24] while providing better performance in some neural network architectures compared to ReLu.

2. Leaky ReLu

Leaky Relu, given as

$$h(x) = \begin{cases} x \text{ for } x > 0 \\ 0.01x \text{ for } x \leq 0 \end{cases}, \qquad h'(x) = \begin{cases} 1 \text{ for } x > 0 \\ 0.01 \text{ for } x \leq 0 \end{cases}. \tag{6.7}$$

It is very similar to ReLu, and as such leaky ReLu exhibits many of the same properties. Although the introduction of a linear component for $x \leq 0$, gives the networks more freedom and it therefore sometimes works better than ReLu.

3. Logistic

The logistic function is the same activation function we used for logistic regression and is defined as

$$h(x) = \frac{1}{1+e^{-x}}, \qquad h'(x) = \frac{e^{x}}{(1+e^{x})^2}. \tag{6.8}$$

First introduced in 1838 by Pierre François Verhulst [56], to model population growth. Its sigmoid shape and smooth well-defined derivative make it a good activation function.

4. Hyperbolic tangent function

The hyperbolic tangent function, shortened tanh, is given as

$$h(x) = \tan^{-1}(x), \qquad h'(x) = \frac{1}{1+x^2}. \tag{6.9}$$

Similar to the logistic function, its sigmoid shape, and smooth well-defined derivative makes it a good activation function.

vi. Mathematical Formulation of PINNs for the 2D TDSE

1. Neural Network Structure

Multi-layer perceptron (MLP) architecture:

- **Input layer**: Represents spatial coordinates (x, y) and time t.
- **Hidden layers**: Fully connected layers with nonlinear activation functions (e.g., ReLU, Tanh).
- **Output layer**: Predicts the real and imaginary parts of the wavefunction $\psi(x, y, t)$.

2. Activation Functions

- Role of activation functions in introducing non-linearity.
- Common choices: ReLU, Tanh, Sigmoid.
- Selection criteria based on performance and convergence.

vii. Loss Function Construction

1. Physics-based Loss: 2D TDSE Residuals

Residual Definition:

$$R(x, y, t) = i\hbar\frac{\partial\psi}{\partial t} + \frac{\hbar^2}{2m}\left(\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2}\right) - V(x, y)\psi$$

Physics-Informed Loss Term:

$$\mathcal{L}_{physics} = \frac{1}{N_r}\sum_{i=1}^{N_r} |R(x_i, y_i, t_i)|^2$$

where N_r is the number of residual points.

2. Data-based Loss: Boundary and Initial Condition Penalties

Boundary Conditions:

$$\psi(x_b, y, t) = 0 \quad \text{and} \quad \psi(x, y_b, t) = 0 \quad \forall\, t \in [0, T]$$

Boundary loss term:

$$\mathcal{L}_{boundary} = \frac{1}{N_b}\sum_{i=1}^{N_b} |\psi(x_{b,i}, y_{b,i}, t_i) - \psi_{b,i}|^2$$

where N_b is the number of boundary points.

Initial Conditions:

$$\psi(x, y, 0) = \psi_0(x, y)$$

Initial condition loss term:

$$\mathcal{L}_{initial} = \frac{1}{N_i}\sum_{i=1}^{N_i} |\psi(x_i, y_i, 0) - \psi_0(x_i, y_i)|^2$$

where N_i is the number of initial condition points.

3. Total Loss Function

$$\mathcal{L} = \mathcal{L}_{physics} + \lambda_b \mathcal{L}_{boundary} + \lambda_i \mathcal{L}_{initial}$$

where λ_b and λ_i are weighting factors for the boundary and initial condition penalties, respectively.

viii. Optimization Techniques

1. Gradient-based Optimization Algorithms

Backpropagation:

- Algorithm for computing gradients of the loss function with respect to network parameters.
- Chain rule application to propagate errors back through the network.

Gradient Descent Variants:

- **Stochastic Gradient Descent (SGD)**:
  - Iterative optimization method updating weights based on a subset of data.
- **Adaptive Methods**:
  - **Adam Optimizer**:
    - Combines momentum and adaptive learning rates for efficient training.
    - Parameter updates:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where g_t is the gradient at time step t, α is the learning rate, β_1 and β_2 are decay rates, and ϵ is a small constant to prevent division by zero.

Chapter 7

7. The Finite Element Method

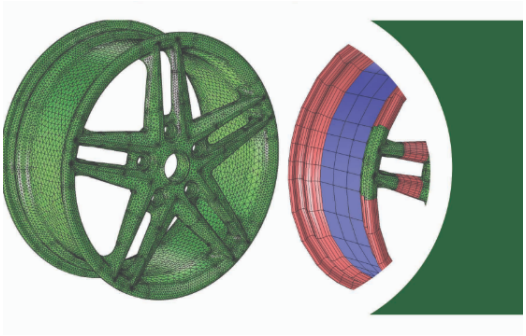

Figure 7. We introduce physics-informed neural networks, neural networks that are trained to solve supervised learning tasks while respecting any given law of physics described by general nonlinear partial differential equations.[49]

> "When faced with a complex problem, divide it into simpler pieces.
> That's not cheating. That's how nature works [51]
>
> Steven Strogatz *Twitter*"

The Finite Element Method (FEM) is a powerful numerical technique used for finding approximate solutions to boundary value problems governed by partial differential equations (PDEs). It is particularly well-suited for problems with complex geometries, material properties, and boundary conditions, where analytical solutions are often intractable. This section outlines the fundamental theoretical steps involved in the FEM.

i. The Governing Equation and Problem Domain

The starting point for FEM is typically a physical problem described by a PDE defined over a continuous domain Ω. Let $u(\boldsymbol{x})$ be the unknown field variable (e.g., displacement, temperature, pressure) we wish to find, where $\boldsymbol{x}$ represents the spatial coordinates. A general form for many steady-state problems can be written as a boundary value problem:

Find $u(\boldsymbol{x})$ such that:

$$\mathcal{L}u = f \quad \text{in } \Omega \tag{7.1}$$

subject to boundary conditions on the boundary $\Gamma = \partial\Omega$. The boundary Γ is typically divided into two parts: Γ_D where Dirichlet (essential) boundary conditions are specified, and Γ_N where Neumann (natural) boundary conditions are specified, such that $\Gamma = \Gamma_D \cup \Gamma_N$ and $\Gamma_D \cap \Gamma_N = \emptyset$.

$$u = g \quad \text{on } \Gamma_D \tag{7.2}$$

$$\frac{\partial u}{\partial n} = q \quad \text{on } \Gamma_N \tag{7.3}$$

Here, $\mathcal{L}$ is a differential operator (e.g., the Laplacian ∇^2), f is a source term, g specifies the value of u on Γ_D, q specifies the normal derivative (flux) on Γ_N, and $\frac{\partial}{\partial n}$ denotes the outward normal derivative.

ii. The Weak Formulation (Variational Form)

Instead of solving the strong form (Eq. 7.1) directly, FEM relies on the weak formulation. This is derived by multiplying the PDE by an arbitrary, sufficiently smooth test function $v(\boldsymbol{x})$ (also called a weight function or variation) that vanishes on the Dirichlet boundary Γ_D (i.e., $v = 0$ on Γ_D), and integrating over the domain Ω:

$$\int_\Omega v(\mathcal{L}u)\, d\Omega = \int_\Omega v f\, d\Omega \tag{7.4}$$

The key step is applying integration by parts (or Green's theorem in multiple dimensions) to the left-hand side. This reduces the order of differentiation required for u and transfers derivatives onto the test function v. For example, if $\mathcal{L}u = -\nabla \cdot (k\nabla u)$ (representing heat conduction or diffusion), integration by parts yields:

$$\int_\Omega \nabla v \cdot (k\nabla u)\, d\Omega - \int_\Gamma v(k\nabla u \cdot \boldsymbol{n})\, d\Gamma = \int_\Omega v f\, d\Omega \tag{7.5}$$

where $\boldsymbol{n}$ is the outward unit normal vector to the boundary Γ. The boundary integral term $\int_\Gamma v(k\nabla u \cdot \boldsymbol{n})\, d\Gamma$ allows for the incorporation of Neumann boundary conditions. Since $k\nabla u \cdot \boldsymbol{n} = k\frac{\partial u}{\partial n}$ and we know $k\frac{\partial u}{\partial n} = kq$ on Γ_N (assuming k is incorporated in q for simplicity or is constant), and $v = 0$ on Γ_D, the boundary integral becomes $\int_{\Gamma_N} v(kq)\, d\Gamma$.

The weak formulation is then stated as: Find u (belonging to an appropriate function space that satisfies the Dirichlet boundary conditions) such that for all admissible test functions v:

$$\int_\Omega \nabla v \cdot (k\nabla u)\, d\Omega = \int_\Omega v f\, d\Omega + \int_{\Gamma_N} v(kq)\, d\Gamma \tag{7.6}$$

This integral equation must hold for every function v in the chosen test function space. The advantage is that u now needs to be less smooth than required by the original PDE (Eq. 7.1).

iii. Domain Discretization (Meshing)

The continuous domain Ω is approximated by dividing it into a finite number of smaller, non-overlapping subdomains called finite elements, denoted by Ω_e. The collection of these elements forms the mesh, $\Omega_h = \cup_{e=1}^{N_{el}} \Omega_e \approx \Omega$, where N_{el} is the total number of elements.

Common element shapes include triangles and quadrilaterals in 2D, and tetrahedra and hexahedra in 3D. Associated with each element are specific points called nodes, typically located at the vertices and sometimes along the edges or faces, or even inside the element. The nodes define the geometry of the element and are where the values of the unknown field variable will be calculated.

iv. Interpolation within Elements (Shape Functions)

Within each element Ω_e, the unknown field variable $u(\boldsymbol{x})$ is approximated by an interpolation function $u_h^e(\boldsymbol{x})$ expressed in terms of the unknown values of u at the nodes of that element. This interpolation is achieved using element-specific basis functions called shape functions (or interpolation functions), $N_i^e(\boldsymbol{x})$.

Let u_i^e be the unknown value of u at the i-th node of element e. The approximation within element e is given by:

$$u(\boldsymbol{x}) \approx u_h^e(\boldsymbol{x}) = \sum_{i=1}^{N_{en}} N_i^e(\boldsymbol{x}) u_i^e \quad \text{for } \boldsymbol{x} \in \Omega_e \tag{7.7}$$

where N_{en} is the number of nodes in element e. The shape function $N_i^e(\boldsymbol{x})$ has the property that it equals 1 at node i of the element and 0 at all other nodes of that element: $N_i^e(\boldsymbol{x}_j) = \delta_{ij}$ (Kronecker delta), where $\boldsymbol{x}_j$ is the coordinate of the j-th node of the element. Shape functions are typically low-order polynomials defined in a local coordinate system within the element.

The global approximation $u_h(\boldsymbol{x})$ over the entire domain Ω_h is obtained by piecing together these element approximations. It can also be expressed using global shape functions $N_I(\boldsymbol{x})$ associated with global node I:

$$u(\boldsymbol{x}) \approx u_h(\boldsymbol{x}) = \sum_{I=1}^{N_{nodes}} N_I(\boldsymbol{x}) U_I \tag{7.8}$$

where U_I is the unknown value at the I-th global node and N_{nodes} is the total number of nodes in the mesh. $N_I(\boldsymbol{x})$ is non-zero only over the elements connected to node I.

v. Derivation of Element Equations

The core idea of the Galerkin method (a common type of FEM) is to use the same shape functions for the test functions v as used for approximating the solution u. So, the test function space is approximated by functions of the form:

$$v_h(\boldsymbol{x}) = \sum_{J=1}^{N_{nodes}} N_J(\boldsymbol{x}) C_J \tag{7.9}$$

where C_J are arbitrary coefficients. Substituting the approximations u_h (Eq. 7.8) and v_h into the weak form (Eq. 7.6), and requiring it to hold for any choice of C_J, is equivalent to requiring it to hold when v_h is set to each global basis function $N_J(\boldsymbol{x})$ individually ($J = 1, ..., N_{nodes}$).

This process is typically performed element by element. Substituting the element approximation (Eq. 7.7) into the weak form (Eq. 7.6) restricted to a single element Ω_e, and choosing the test function to be one of the element shape functions N_j^e, we get a system of equations for the element:

$$\sum_{i=1}^{N_{en}} \left(\int_{\Omega_e} \nabla N_j^e \cdot (k \nabla N_i^e) \, d\Omega \right) u_i^e = \int_{\Omega_e} N_j^e f \, d\Omega + \int_{\Gamma_e \cap \Gamma_N} N_j^e (kq) \, d\Gamma \tag{7.10}$$

This holds for $j = 1, ..., N_{en}$. This set of equations can be written in matrix form for each element:

$$\boldsymbol{k}^e \boldsymbol{u}^e = \boldsymbol{f}^e \tag{7.11}$$

where:

- $\boldsymbol{u}^e = \{u_1^e, u_2^e, ..., u_{N_{en}}^e\}^T$ is the vector of unknown nodal values for element e.
- $\boldsymbol{k}^e$ is the element stiffness matrix (or conductivity, diffusion matrix etc.), with entries:

$$k_{ji}^e = \int_{\Omega_e} \nabla N_j^e \cdot (k \nabla N_i^e) \, d\Omega \tag{7.12}$$

- $\boldsymbol{f}^e$ is the element load vector (or source/flux vector), with entries:

$$f_j^e = \int_{\Omega_e} N_j^e f \, d\Omega + \int_{\Gamma_e \cap \Gamma_N} N_j^e (kq) \, d\Gamma \tag{7.13}$$

Note that the boundary integral term only contributes if part of the element's boundary Γ_e coincides with the Neumann boundary Γ_N.

vi. Assembly and Solution of Global System

The final step involves assembling the individual element equations (Eq. 7.11) into a single global system of linear algebraic equations that represents the entire problem domain. This assembly process is based on the principle of compatibility (the value of u_h is unique at shared nodes) and equilibrium (or balance, represented by the weak form).

The global stiffness matrix $\boldsymbol{K}$ and global load vector $\boldsymbol{F}$ are constructed by summing the contributions from all element stiffness matrices $\boldsymbol{k}^e$ and element load vectors $\boldsymbol{f}^e$, respectively, according to the connectivity of the global nodes. If I and J are the global node numbers corresponding to the local element node numbers i and j for element e, then k^e_{ji} contributes to the global entry K_{JI}, and f^e_j contributes to the global entry F_J.

This results in a global system of equations:

$$\boldsymbol{KU} = \boldsymbol{F} \tag{7.14}$$

where $\boldsymbol{U}$ is the global vector of unknown nodal values $\{U_1, U_2, ..., U_{N_{nodes}}\}^T$.

Before solving, the Dirichlet boundary conditions (Eq. 7.2) must be enforced. This is typically done by modifying the global system $\boldsymbol{KU} = \boldsymbol{F}$. For example, if U_I is known ($U_I = g_I$), the I-th equation can be replaced by $U_I = g_I$, and the known value U_I can be moved to the right-hand side in other equations.

After applying boundary conditions, the resulting system of linear algebraic equations is solved using appropriate numerical solvers (direct methods like Gaussian elimination or LU decomposition for smaller systems, or iterative methods like Conjugate Gradient for larger systems) to find the unknown nodal values $\boldsymbol{U}$. Once $\boldsymbol{U}$ is known, the approximate solution $u_h(\boldsymbol{x})$ is determined everywhere via Eq. 7.8 (or Eq. 7.7 within each element), and other derived quantities (like gradients, fluxes, stresses) can be calculated.

Part IV

Method

CHAPTER 8

8. Comparative Numerical Framework

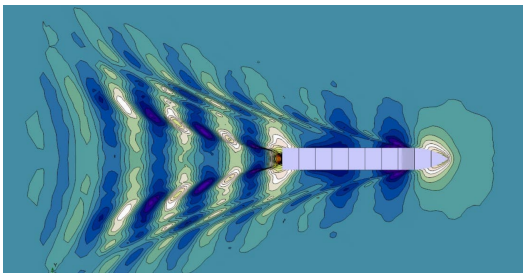

Figure 8. Numerical simulation showing wave resistance contours around a vessel hull, an example of the kind of data produced by finite volume or finite element frameworks in naval engineering.[2]

> "All models are wrong, but some are useful.[6]
>
> George E. P. Box *Journal of the American Statistical Association*"

This section outlines the comparative framework adopted in this thesis to systematically evaluate the performance of PINNs against FEM. While these two methodologies differ fundamentally in their construction, FEM being a well-established discretization-based technique and PINNs representing a mesh-free, machine learning-driven paradigm, both are designed to solve partial differential equations (PDEs) constrained by physical laws. To ensure a meaningful and controlled comparison, we emphasize evaluating both methods on identical problem settings, boundary conditions, and physical systems.

Several measures are taken to guarantee that the comparison is methodologically sound. Both PINNs and FEM are implemented to solve the same governing equations on matched spatial domains, with equivalent initial and boundary conditions. Moreover, the systems selected span both classical and quantum regimes, ranging from linear PDEs (such as the diffusion and Poisson equations) to nonlinear and high-dimensional formulations such as the time-dependent Schrödinger equation and interacting electron systems. These problems are chosen to test both accuracy and scalability across regimes where traditional methods are known to struggle and where PINNs are hypothesized to provide computational advantages.

By aligning the problem definitions, input data, and physical constraints between the two frameworks, we ensure that any differences in performance, convergence behavior, or accuracy can be attributed to the methodological differences themselves, rather than artifacts introduced by inconsistent problem specifications. This comparative setup forms the foundation for evaluating the practicality, generalization, and physical faithfulness of both PINNs and FEM.

i. Objective of the Comparison

The primary objective of this comparison is to assess whether Physics-Informed Neural Networks can serve as a viable alternative or complementary tool to the Finite Element Method in solving PDEs across both classical and quantum domains. Specifically, we aim to evaluate each method with respect to the following criteria:

- **Accuracy:** How well does each method approximate the true physical solution? We assess this both quantitatively (e.g., via L^2 and L^∞ norms) and qualitatively (e.g., through visual inspection of wavefunctions and densities).

- **Computational Efficiency:** How much computational effort (in terms of time and memory) is required to reach a given level of accuracy?
- **Generalization and Scalability:** How do the methods perform as system complexity increases? Can they adapt to high-dimensional or strongly correlated regimes?
- **Robustness:** How sensitive is each method to variations in initial and boundary conditions or domain geometry?

Through this structured comparison, we aim to clarify the potential of PINNs not merely as a novelty, but as a tool capable of addressing limitations in classical numerical techniques. Conversely, we seek to identify where PINNs fall short and where traditional methods such as FEM retain critical advantages. Ultimately, this evaluation contributes to the ongoing discourse on the role of machine learning in computational physics, providing practical insights rather than theoretical conjecture.

ii. Evaluation Criteria

To ensure a systematic and meaningful comparison between PINNs and FEM, we base our evaluation on a suite of metrics that probe both numerical performance and physical fidelity. The following quantitative criteria are used:

- **Error norms:** We compute the L^2 and L^∞ norms of the difference between the numerical and analytical (or highly resolved reference) solutions, defined respectively as:

$$\|u - u_{\text{ref}}\|_{L^2} = \left(\int_\Omega |u(x) - u_{\text{ref}}(x)|^2 dx \right)^{1/2}, \quad \|u - u_{\text{ref}}\|_{L^\infty} = \max_{x \in \Omega} |u(x) - u_{\text{ref}}(x)|.$$

- **Energy error:** For quantum systems, we report deviations in ground state energy compared to reference values.
- **Electron density error:** In interacting quantum systems, we compare the computed one-body electron density $\rho(x)$ against the analytical or reference profile using the integrated absolute error:

$$\Delta\rho = \int_\Omega |\rho_{\text{PINN}}(x) - \rho_{\text{FEM}}(x)| \, dx.$$

- **Computational time and memory usage:** Execution time is measured from start of training/solve to convergence, and peak memory consumption is tracked via profiling tools.
- **Convergence behavior:** We monitor the rate of convergence as a function of training epochs (PINNs) or mesh resolution (FEM).

All simulations are run on an Intel(R) Core(TM) i9-13900H CPU with 32 GB RAM. PINN models are trained using PyTorch with GPU acceleration via an NVIDIA GeForce RTX 4070 Laptop GPU (8GB VRAM), while FEM simulations are implemented in FEniCS and executed on the same CPU backend for fairness. Time-to-solution is benchmarked independently for CPU-only and GPU-accelerated cases when relevant.

iii. Selected Benchmark Problems

To probe the performance of PINNs and FEM across a diverse range of physical regimes, we evaluate both methods on a suite of benchmark problems spanning classical and quantum mechanics. Each system is chosen to highlight different aspects of numerical difficulty: linearity, dimensionality, boundary complexity, and the presence of interactions or singularities.

Table I summarizes the systems studied and the primary evaluation metrics used in the analysis. The results of these benchmarks are reported and analyzed in Chapter 13.

Table I. Overview of benchmark systems, evaluation metrics for PINNs and FEM, and computational configurations.

System	**PDE / Equation**	**Evaluation Metric(s) (PINNs)**	**Evaluation Metric(s) (FEM)**	**Hardware**	**Training Time**
Diffusion of Gaussian	Diffusion equation (linear, 2D)	L^2, L^∞ norm error vs. analytic	Grid convergence, L^2 error	RTX 3090	2h 13m
Poisson problem	Poisson equation with Dirichlet BCs	Residual loss, L^2 error	Discrete L^2 norm vs. mesh resolution	Tesla V100	45m
Navier–Stokes flow	Incompressible Navier–Stokes (2D)	Residual loss, vorticity field error, L^2 velocity error	Velocity field vs. grid resolution, pressure convergence	Tesla A100	4h 30m
TDSE: Harmonic oscillator	Time-dependent Schrödinger equation (2D)	Energy error, L^∞, density profile error	Energy spectrum convergence, L^2 norm	RTX A6000	3h 40m
Two electrons in harmonic trap	Interacting TDSE (2-body)	Ground state energy, $\rho(x)$ error	$\rho(x)$ vs. spectral method, energy spectrum	Tesla A100	5h 15m
Multiple fermions, interacting	Many-body TDSE	Slater determinant error, $\rho(x)$	Matrix element consistency, Slater energy error	Tesla V100	6h 50m

CHAPTER 9

9. Finite Element Method (FEM)

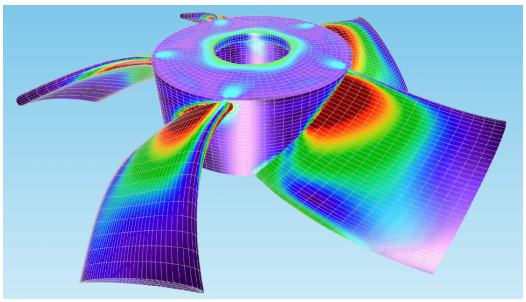

Figure 9. Finite Element Analysis (FEA) simulation visual showing stress distribution in a mechanical part. Adapted from MISUMI USA (2021).[37]

> “There are only two hard things in Computer Science: cache invalidation and naming things.[17]
>
> Phil Karlton”

i. Weak Formulation of Governing Equations

In this work, the finite element method was employed to solve the two-dimensional transient heat equation, a prototypical parabolic partial differential equation of the form

$$\frac{\partial u}{\partial t} - \nabla \cdot (\nabla u) = f \quad \text{in } \Omega \times (0, T],$$

subject to homogeneous Dirichlet boundary conditions. To derive the variational formulation, the equation was multiplied by a test function $v \in V$, where V is an appropriate Sobolev space (e.g., $H_0^1(\Omega)$), and integrated over the spatial domain Ω. Applying integration by parts to the diffusive term and assuming vanishing test functions on the boundary leads to the weak form: find $u \in V$ such that for all $v \in V$,

$$(u_t, v) + (\nabla u, \nabla v) = (f, v),$$

where $(\cdot, \cdot)$ denotes the L^2-inner product on Ω. Temporal discretization was introduced using a first-order implicit Euler scheme, which is unconditionally stable and well-suited for stiff systems such as diffusion-dominated dynamics. The resulting semi-discrete weak formulation then takes the form:

$$\left(\frac{u^{n+1} - u^n}{\Delta t}, v\right) + (\nabla u^{n+1}, \nabla v) = (f, v),$$

which was subsequently rearranged to isolate the unknown u^{n+1} at the new time step. This formulation preserves the symmetry and coercivity of the underlying elliptic operator, facilitating the use of efficient solvers in the next stages of implementation.

ii. Spatial Discretization

The spatial discretization was conducted using conforming Lagrange finite elements of first order, i.e., piecewise linear elements that ensure continuity across element boundaries. The computational domain $\Omega = [-2, 2]^2$ was

discretized into triangular elements using a structured rectangular grid. This choice of geometry simplifies mesh generation while maintaining sufficient generality to reflect the method's applicability to more complex domains.

The use of first-order elements was motivated by the balance between computational efficiency and accuracy, particularly given the smooth nature of the initial condition and the expected diffusive smoothing of the solution over time. The finite element space $V_h \subset V$ was thus spanned by basis functions associated with the nodes of the mesh, enabling straightforward implementation of both the bilinear form $a(u, v)$ and the linear form $L(v)$.

This discretization serves as the foundation for the assembly of the global linear system at each time step, linking the weak formulation to the numerical realization through the mesh and basis functions. The construction of this mesh and the strategic choices made in its resolution and refinement are discussed in the next section, where the emphasis shifts from the formulation of the variational problem to the structural representation of the computational domain.

a. Element Selection and Meshing The computational domain was discretized using triangular elements generated from a structured Cartesian grid, where each square cell was divided into two triangles. This approach retains the simplicity and alignment of Cartesian grids while ensuring compatibility with unstructured meshing strategies that are essential for more complex geometries. Triangular elements were chosen for their geometric flexibility and their suitability for the Delaunay-type subdivision often used in adaptive refinement and coarsening algorithms.

The resolution of the mesh was set uniformly in both spatial directions, providing a balanced aspect ratio that mitigates numerical anisotropy. Specifically, a uniform mesh with $nx = ny = 20$ subdivisions in each direction was adopted, resulting in a relatively coarse initial discretization. This granularity was deemed sufficient for capturing the primary features of the diffusive dynamics while maintaining computational tractability across time steps.

Given the Gaussian initial condition used in the simulation, characterized by a localized peak centered in the domain, the mesh design ensured sufficient node density near the center without requiring additional geometric refinement. The uniform distribution of elements, coupled with linear Lagrange basis functions, enabled consistent approximation properties across the domain, facilitating convergence toward the analytical behavior expected from the heat equation.

b. Mesh Refinement Strategies Although no adaptive refinement was applied dynamically during the simulation, the meshing strategy was chosen with potential refinement pathways in mind. In problems where localized features or sharp gradients persist over time, such as in nonlinear or multiscale settings, adaptive mesh refinement (AMR) becomes essential. However, for the present study, the analytic smoothness and temporal decay of the solution made uniform meshing sufficient.

Nonetheless, the structure of the implementation allows for straightforward incorporation of refinement techniques such as h-refinement, where elements are subdivided to increase local resolution, or p-refinement, where higher-order basis functions are employed within selected elements. The current formulation and solver configuration are compatible with such enhancements, providing flexibility for future extensions of the methodology to more challenging physical systems.

The choice to use a fixed mesh throughout the simulation simplifies the assembly process and maintains the consistency of the global matrix structure across time steps. This decision reduces overhead and is especially beneficial in time-dependent problems where remeshing would otherwise introduce additional computational and numerical complexity. Having established the mesh architecture, attention now turns to the functional representation within each element, specifically, the selection and role of basis functions used for interpolation and approximation of the solution field.

iii. Basis Functions and Interpolation

To approximate the solution within each finite element, continuous piecewise linear basis functions were employed. These are associated with the nodal degrees of freedom and belong to the space of first-order Lagrange polynomials. Each basis function is defined locally on an element and has the Kronecker-delta property, being equal to one at its associated node and zero at all others. This facilitates interpolation of both the initial condition and the evolving solution field throughout the domain.

The choice of linear elements reflects a compromise between accuracy and computational cost. While higher-order polynomials could improve approximation quality in regions with sharp gradients, the smooth, diffusive nature of the solution in this problem favors a lower-order scheme. Moreover, first-order basis functions simplify the assembly of stiffness matrices and are well-supported in widely-used finite element libraries.

Interpolation of the initial condition was carried out by projecting a smooth Gaussian function onto the finite element space. This projection ensures compatibility with the variational formulation and provides a physically meaningful starting point for the time integration. The basis functions also support the implementation of essential boundary conditions, as degrees of freedom associated with boundary nodes can be directly manipulated to enforce Dirichlet conditions.

Having defined the discrete function space and basis, the next task is the construction of the global system of equations. This involves assembling the contributions from individual elements into a global matrix and right-hand side vector that reflect the discretized variational formulation.

iv. Assembly of Global System

The global linear system at each time step arises from the finite element discretization of the weak form. The bilinear form $a(u, v)$, incorporating both the mass and stiffness contributions, was integrated over each element and then assembled into the global matrix. Similarly, the linear form $L(v)$, incorporating the solution from the previous time step and any source terms, was used to construct the global right-hand side vector.

The assembly process respects the sparsity structure imposed by the locality of basis functions; each element contributes only to a small portion of the global matrix, corresponding to the degrees of freedom associated with its vertices. Efficient assembly was ensured by leveraging parallel data structures and routines, allowing the distribution of computational workload across multiple processes.

Boundary conditions were incorporated directly during the assembly phase through the application of lifting and projection operations, ensuring consistency of the solution with prescribed Dirichlet data. In addition, synchronization mechanisms ensured that all ghost degrees of freedom (required for parallel consistency) were appropriately updated prior to solving the system.

With the global system constructed at each time step, the focus shifts to the solver strategy employed to compute the updated solution field. The next section addresses the solver configuration in detail, including choices related to direct versus iterative methods and the convergence properties governing the time integration loop.

v. Solver Configuration

Once the global linear system was assembled at each time step, the solution was obtained using a direct solver configuration. Given the moderate size of the problem and the symmetric, positive-definite structure of the system matrix resulting from the implicit time discretization of the heat equation, a sparse LU decomposition was deemed the most appropriate choice. This approach offers high accuracy and robustness for systems where computational cost remains manageable, particularly when matrix reassembly is not required at each iteration.

The solver was implemented using the PETSc Krylov Subspace Solver (KSP) framework, which provides flexibility in selecting and configuring solver strategies. For this application, the Krylov solver was configured to operate in pre-only mode, thereby delegating all work to the preconditioner. The preconditioner was set to LU factorization, effectively turning the entire solver into a wrapper around a direct method. This configuration avoids iterative convergence checks and is well-suited to systems where the matrix remains constant or nearly so across time steps, as is typical in linear diffusion problems.

a. Direct vs. Iterative Solvers While direct solvers such as LU factorization provide exact solutions up to numerical precision and require minimal configuration, they scale poorly with system size in terms of both memory and computational effort. In contrast, iterative solvers like Conjugate Gradient (CG) or Generalized Minimal Residual (GMRES) are better suited to large-scale problems where the system matrix is sparse and well-conditioned. These solvers rely on successive approximations and require the definition of convergence criteria to determine when a solution is sufficiently accurate. The decision to use a direct solver in this study was guided by the relatively small problem size and the desire to minimize the sources of numerical error and instability during method development. However, the implementation is fully compatible with iterative solvers, and the framework can be extended to support them efficiently by introducing suitable preconditioners, such as incomplete LU (ILU) or algebraic multigrid (AMG), particularly in more complex or nonlinear systems.

b. Convergence Criteria Given the use of a direct solver, convergence criteria in the classical iterative sense were not applied during the time-stepping loop. However, convergence was still ensured at the global level by verifying the consistency and stability of the solution across time steps. The time step size $\Delta t = T/N$, where $T = 1.0$ and $N = 10$, was selected to maintain numerical stability while ensuring sufficient temporal resolution to track the evolution of the solution. In contexts where iterative solvers are employed, convergence criteria typically involve bounds on the relative or absolute residual norms, with thresholds determined based on the required solution accuracy and conditioning of the matrix. The present configuration circumvents these considerations, but the surrounding infrastructure remains general enough to incorporate such criteria should the problem complexity increase. With the solver reliably configured and integrated, the complete FEM pipeline is capable of efficiently advancing the solution in time. This completes the methodological foundation for analyzing the behavior of diffusive systems under various initial and boundary conditions, and provides a reusable framework for more complex simulations to follow.

CHAPTER 10

10. Methodological Comparisons

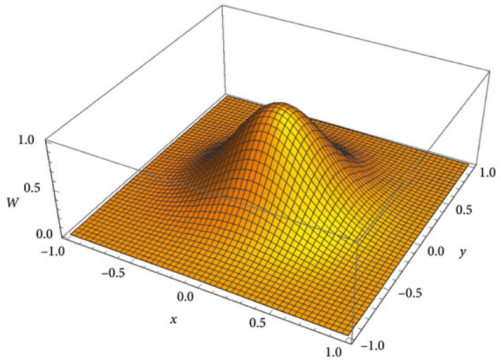

Figure 10. The MLS Approximation Scheme.[58]

> "If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail.[34]
>
> Abraham Maslow - *The Psychology of Science*"

i. Accuracy Assessment

Assessing the accuracy of numerical methods requires a systematic approach to quantifying the deviation between computed and reference solutions. In this study, accuracy comparisons were conducted between different numerical schemes, such as the Finite Element Method (FEM) and machine learning-based solvers, using established norm-based and energy-based metrics. Each solution was interpolated or projected onto a common function space to ensure meaningful comparisons, especially in cases where the discretizations differ in structure or resolution.

a. L^2, L^∞, and Energy-Based Metrics To evaluate pointwise and global discrepancies, both the L^2-norm and L^∞-norm of the error were computed. The L^2-norm provides a measure of the average deviation across the domain, emphasizing overall fidelity, whereas the L^∞-norm isolates the maximum absolute deviation, offering sensitivity to local anomalies or peak errors. For systems governed by physical conservation laws or variational principles, energy-based metrics were also employed. These include comparisons of total system energy, such as kinetic, potential, or Hamiltonian components, as applicable to the specific PDE under study.

When no analytical solution is available, a high-resolution reference solution computed using a highly refined FEM or a spectral method was treated as the ground truth. All errors were computed relative to this reference, either by direct nodal evaluation or through integral approximations within the discrete function space. This approach provides a consistent, quantitative basis for evaluating the fidelity of each numerical method.

ii. Computational Efficiency

Efficiency comparisons address the practical feasibility of each method under typical constraints of time and resources. To this end, performance metrics such as runtime, memory usage, and scalability were recorded for each simulation. These metrics provide insight into the cost-benefit trade-offs inherent to different discretization schemes and solver architectures.

a. Runtime Scaling To evaluate runtime behavior, each method was benchmarked across varying problem sizes and mesh resolutions. The scaling of computational time with respect to degrees of freedom or time steps was analyzed to identify algorithmic bottlenecks and to classify the asymptotic complexity of each solver. In time-dependent problems, total simulation time was decomposed into per-step costs and overhead due to assembly, I/O, and communication.

Particular attention was given to the solver backend, including linear solver convergence and any mesh-dependent overheads such as reassembly or preconditioner updates. Parallel performance was also considered when applicable, with runtime comparisons made between serial and distributed configurations.

b. Memory and Resource Usage In addition to runtime, the memory footprint of each method was profiled to understand its scaling behavior and compatibility with resource-constrained environments. Memory usage was decomposed into mesh storage, matrix assembly, and solver working memory. For methods involving dense linear algebra or large neural network architectures, GPU memory and cache pressure were also monitored.

This analysis helps to distinguish methods that are memory-bound from those that are compute-bound, informing the choice of hardware and guiding future optimization strategies. In some cases, trade-offs between memory and speed were explicitly evaluated, particularly when switching between direct and iterative solvers or between classical and machine-learning-based approaches.

iii. Robustness to Boundary and Initial Conditions

The robustness of each numerical method was assessed by varying initial and boundary conditions within physically reasonable ranges. This includes tests with smooth, discontinuous, or sharply peaked initial states, as well as homogeneous and inhomogeneous boundary configurations. The stability and accuracy of the methods were evaluated based on their ability to preserve qualitative features, such as mass conservation, smoothness, and symmetry, under such perturbations.

For methods relying on data-driven models, robustness was also interpreted in terms of generalization: the capacity to maintain performance when exposed to initial or boundary conditions not encountered during training. This aspect is critical when extending such models to real-world scenarios where perfect prior knowledge cannot be assumed.

iv. Summary of Methodological Trade-offs

Each numerical method considered in this study presents a distinct set of trade-offs. The FEM offers well-established convergence guarantees, physical interpretability, and flexibility in handling complex geometries, but may become computationally expensive for high-dimensional or nonlinear problems. Machine learning approaches, by contrast, provide potential speed-ups and memory savings through model inference, yet their reliability hinges on training data coverage and network expressivity.

This comparative analysis thus provides a framework not only for selecting the appropriate solver under given constraints but also for guiding future methodological development. The next chapters present empirical results that illustrate these trade-offs in practice, offering insight into the strengths and limitations of each approach under representative problem conditions.

Part V

Implementation

Chapter 11

11. Programming in the Natural Sciences

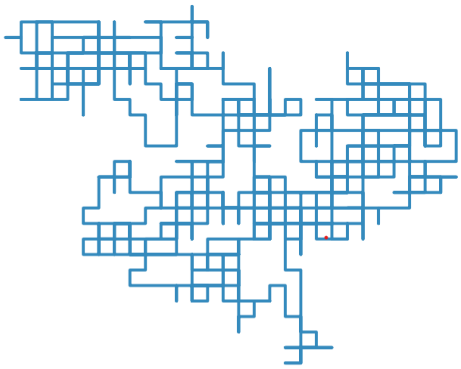

Figure 11. Random walking algorithm on a two-dimensional grid, walking 1000 steps.

> “The computer is not only an instrument for observing nature, but also for experimenting with it[12]
>
> Freeman Dyson - *Science in the Age of Computers*”

The nature of scientific inquiry has changed dramatically over the past century. Where pen-and-paper derivations and analytical solutions once reigned supreme, the advent of digital computers in the mid-20th century transformed the landscape of research across the natural sciences. A particularly pivotal moment came in the 1940s and 1950s with the development of early general-purpose computers like the ENIAC and the IBM 701, both of which were used for numerical simulations of physical systems[8, 18, 20, 36]. These machines enabled physicists to perform calculations far beyond the reach of manual methods, laying the groundwork for modern computational physics.

Since then, computational methods have become indispensable tools in physics, chemistry, biology, and engineering. In particular, systems governed by partial differential equations (PDEs), such as the Schrödinger equation in quantum mechanics or the Navier-Stokes equations in fluid dynamics, often defy closed-form solutions and must be approximated numerically. The resulting models are implemented in code that must be accurate, efficient, and crucially, maintainable.

Modern research software is frequently developed in collaborative environments and used across multiple projects. As such, good software design has become just as important as numerical accuracy. A poorly structured codebase can inhibit reproducibility, obscure bugs, and make further development difficult. To this end, scientific computing has increasingly adopted paradigms from software engineering, chief among them *Object-Oriented Programming* (OOP). This chapter will introduce the Python programming language, highlighting its features relevant to scientific computing and demonstrating how its object-oriented capabilities can be harnessed to build robust and modular scientific software, exemplified by the implementation of Physics-Informed Neural Networks (PINNs).

i. Essential Python for Scientific Computing

Python has emerged as a dominant language in scientific computing and data science, lauded for its readability, extensive ecosystem of libraries, and gentle learning curve. It prioritizes developer productivity and ease of use through high-level abstractions and dynamic typing. This section provides a foundational overview of Python features crucial for understanding its application in numerical methods and object-oriented design, particularly as a precursor to implementing models like Physics-Informed Neural Networks (PINNs).

1. Core Philosophy and Dynamic Nature

Python's design philosophy emphasizes code readability and simplicity. It is an interpreted language, meaning code is executed line-by-line by an interpreter, which facilitates rapid prototyping and debugging. A key characteristic is its *dynamic typing*: variable types are checked during runtime, and a variable can be rebound to objects of different types. For instance:

```
x = 10          # x is an integer
print(type(x))  # Output: <class 'int'>
x = "hello"     # x is now a string
print(type(x))  # Output: <class 'str'>
x = [1, 2, 3]   # x is now a list
print(type(x))  # Output: <class 'list'>
```

Listing 1. Python's dynamic typing illustration.

While this offers flexibility, it also means that type-related errors might only surface at runtime. In scientific computing, this dynamism is often balanced by using libraries like NumPy that provide efficiently implemented, statically-typed array structures, and by employing type hinting for improved code clarity and static analysis.

2. Fundamental Data Types and Structures

Python offers several built-in data types that form the backbone of most programs. Everything in Python is an object, including these fundamental types:

- **Numeric Types:**
  - `int`: Arbitrary-precision integers (e.g., `10`, `-500`).
  - `float`: Double-precision floating-point numbers (e.g., `3.14`, `-0.001`). These are typically IEEE 754 double-precision.
  - `complex`: Complex numbers (e.g., `3+4j`).
- **Boolean Type:** `bool` with values `True` and `False`.
- **Sequence Types:**
  - `str`: Immutable sequences of Unicode characters (e.g., `"hello"`, `'world'`).
  - `list`: Mutable, ordered sequences of items (e.g., `[1, "apple", 3.0]`). Lists are heterogeneous and can be resized.
  - `tuple`: Immutable, ordered sequences of items (e.g., `(1, "apple", 3.0)`). Often used for fixed collections or as dictionary keys.
- **Mapping Type:**
  - `dict`: Mutable collections of key-value pairs (e.g., `{"name": "Alice", "age": 30}`). Keys must be hashable.
- **Set Types:**
  - `set`: Mutable, unordered collections of unique, hashable items (e.g., `{1, 2, 3}`).
  - `frozenset`: Immutable version of a set.

For scientific computing, libraries like NumPy extend these with powerful N-dimensional array objects (`numpy.ndarray`) that allow for efficient numerical operations on homogeneous data, often serving as the primary data structure for numerical algorithms.

3. Control Flow

Python provides standard control flow statements:

- **Conditional execution:** `if`, `elif` (else if), `else`.

- **Loops:** `for` loops (typically used for iterating over sequences or iterables) and `while` loops (for repeating a block of code as long as a condition is true).
- **Loop control:** `break` (to exit a loop prematurely) and `continue` (to skip the rest of the current iteration and proceed to the next).

Python uses indentation (conventionally four spaces) to define code blocks, rather than braces or keywords like `end`. This enforces a readable code style.

```python
def process_data(data_list):
    results = []
    for item in data_list:
        if not isinstance(item, (int, float)):
            print(f"Skipping non-numeric item: {item}")
            continue # Skip to the next item
        if item < 0:
            print("Negative number encountered. Stopping processing.")
            break     # Exit the loop
        elif item == 0:
            results.append("zero")
        else:
            results.append(item * 2)
    return results

my_data = [10, 20.5, "text", 0, -5, 30]
processed_data = process_data(my_data)
print(processed_data)
# Expected Output:
# Skipping non-numeric item: text
# Negative number encountered. Stopping processing.
# [20, 41.0, 'zero']
```

Listing 2. Python control flow example demonstrating if, elif, else, for, continue, and break.

4. Functions

Functions are first-class objects in Python, meaning they can be assigned to variables, passed as arguments to other functions, and returned from functions. They are defined using the `def` keyword.

```python
def calculate_energy(mass, velocity, c=299792458):
    """Calculates relativistic kinetic energy (approximation for low v).

    Args:
        mass (float): Mass of the object in kg.
        velocity (float): Velocity of the object in m/s.
        c (float, optional): Speed of light in m/s. Defaults to standard value.

    Returns:
        float: Approximate kinetic energy in Joules (0.5 * m * v^2).
    """
    if velocity >= c:
        raise ValueError("Velocity cannot exceed the speed of light.")
    return 0.5 * mass * velocity**2

energy1 = calculate_energy(mass=1.0, velocity=100)
print(f"Energy for 1kg at 100 m/s: {energy1} J")

# Functions can accept variable numbers of arguments
def sum_values(*args, **kwargs): # *args collects positional, **kwargs collects keyword arguments
```

```
    total = sum(args)
    print("Keyword arguments received:", kwargs)
    return total

result = sum_values(1, 2, 3, scale=10, unit="meters")
print(f"Sum: {result}")
# Expected Output:
# Energy for 1kg at 100 m/s: 5000.0 J
# Keyword arguments received: {'scale': 10, 'unit': 'meters'}
# Sum: 6
```

Listing 3. Python function definition, showcasing arguments, default values, and docstrings.

Python supports positional arguments, keyword arguments, default argument values, and arbitrary argument lists (using `*args` for positional and `**kwargs` for keyword arguments). Lambda functions provide a way to create small, anonymous inline functions (e.g., `square = lambda x: x*x`).

5. Modules and Packages

Python's power in scientific computing is significantly amplified by its extensive standard library and third-party packages. Code is organized into **modules** (typically a single `.py` file containing Python definitions and statements) and **packages** (a collection of modules in a directory hierarchy that includes a special `__init__.py` file). The `import` statement is used to bring functionality from modules into the current namespace.

```
import math # Import the entire math module
print(math.sqrt(16))  # Output: 4.0

from math import pi, sin # Import specific attributes from math
print(sin(pi/2))     # Output: 1.0

import numpy as np # Common alias for NumPy
arr = np.array([1, 2, 3])
print(arr * 2)       # Output: [2 4 6] (element-wise multiplication)
print(np.dot(arr, arr)) # Output: 14 (dot product)
```

Listing 4. Importing and using modules with examples from math and NumPy.

For scientific computing, key packages include:

- **NumPy**[23]: The fundamental package for numerical computation, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.
- **SciPy**[35, 57]: Builds on NumPy and provides a large collection of algorithms and high-level functions for scientific and technical computing, such as optimization, integration, interpolation, signal processing, linear algebra, statistics, and more.
- **Matplotlib**[28]: A comprehensive library for creating static, animated, and interactive visualizations in Python.
- **Pandas**[35, 42]: Offers high-performance, easy-to-use data structures (like `DataFrame`) and data analysis tools, particularly useful for working with tabular data.
- **PyTorch**[43], **TensorFlow**[3], **JAX**[10, 47]**:** Leading deep learning frameworks that provide automatic differentiation, GPU acceleration, and extensive tools for building and training neural networks. These are central to implementing PINNs.

Understanding how to use these modules effectively is crucial for any scientific Python developer. The ability to leverage these well-tested and optimized libraries allows researchers to focus on their specific problem domain rather than reinventing fundamental numerical tools.

With this foundational understanding of Python's core features and its ecosystem, we can now delve into how its object-oriented capabilities are leveraged to structure complex scientific applications.

ii. Object-Oriented Programming in Python for Scientific Models

Object-oriented programming (OOP) is a paradigm where software is organized into *objects*, modular entities that encapsulate both data (attributes or state) and behavior (methods or functions that operate on the data). Python is inherently an object-oriented language; in fact, virtually everything in Python is an object, from numbers and strings to more complex structures like functions and classes themselves. This approach offers clear advantages in scientific computing, aligning well with the need to model complex systems and algorithms:

- **Encapsulation:** Complex systems can be broken into smaller, self-contained units (objects) that manage their own state and expose well-defined interfaces. For example, a specific PDE solver, a neural network architecture, or a representation of a physical system can each be encapsulated within an object. This hides internal complexity and protects data from unintended external modification.
- **Modularity:** Different components (e.g., PDE definitions, network layers, loss functions, optimizers) can be developed, tested, and maintained in isolation as separate classes or modules. These components can then be composed together to build larger systems, enhancing code organization and reusability.
- **Inheritance:** New, specialized classes can be derived from more general base classes. This promotes code reuse by allowing subclasses to inherit attributes and methods from parent classes, while also providing the flexibility to override or extend specific behaviors. For instance, different types of neural network layers or activation functions can inherit from common base layer/activation classes.
- **Polymorphism:** Literally "many forms," polymorphism allows objects of different classes that share a common interface (e.g., through inheritance or by implementing the same methods often referred to as "duck typing" in Python) to be treated uniformly. This enables flexible and extensible code, such as swapping different numerical solvers or optimizers easily without changing the core logic that uses them.

Python's clean syntax, dynamic nature, and rich support for OOP constructs (like classes, inheritance, and special methods) make it particularly well-suited for object-oriented scientific computing. Libraries such as `NumPy`, `SciPy`, and especially deep learning frameworks like `PyTorch` or `TensorFlow`, are themselves designed with strong OOP principles. This makes it straightforward to integrate numerical methods, linear algebra routines, and neural network components into a cohesive object-oriented framework.

In Python, classes are defined using the `class` keyword. The first argument to instance methods within a class is conventionally named `self`, which refers to the instance of the class itself (analogous to `this` in C++ or Java). Special methods, often called "dunder" (double underscore) methods like `__init__` (the constructor, called when an object is created) and `__str__` (for generating a string representation of an object), allow classes to integrate seamlessly with Python's built-in operations and syntax.

To motivate this further, we present in the next subsection an example of how a Physics-Informed Neural Network (PINN) can be implemented using OOP in Python. This example highlights how scientific models benefit from being encapsulated as objects, particularly when dealing with PDE solvers, neural network architectures, and composite loss functions.

1. An Example with PINNs

To illustrate the advantages of object-oriented design in scientific computing, we now present a minimal example of a Physics-Informed Neural Network (PINN) for solving the 1D Poisson equation:

$$-\frac{d^2u(x)}{dx^2} = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0 \tag{11.1}$$

Here, $u(x)$ is the unknown function and $f(x)$ is a given source term. For this example, we choose $f(x) = \pi^2 \sin(\pi x)$, for which the exact solution is $u(x) = \sin(\pi x)$. This makes the example particularly useful for pedagogical purposes, as the correctness of the neural network approximation can be directly compared with the analytical solution.

PINNs approximate the solution $u(x)$ using a neural network $u_\theta(x)$, trained to minimize a composite loss function derived from the governing equation and the boundary conditions. The interior loss, enforcing the differential equation, is given by:

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_r} \sum_{i=1}^{N_r} \left(\frac{d^2 u_\theta(x_i)}{dx^2} + f(x_i) \right)^2 \tag{11.2}$$

In addition to this, the boundary loss ensures that the Dirichlet conditions are satisfied. It is defined as:

$$\mathcal{L}_{\text{BC}} = (u_\theta(0))^2 + (u_\theta(1))^2 \tag{11.3}$$

Combining these terms yields the total loss:

$$\mathcal{L} = \mathcal{L}_{\text{PDE}} + \lambda \mathcal{L}_{\text{BC}} \tag{11.4}$$

where λ is a weighting parameter controlling the influence of the boundary condition term. For simplicity, we choose $\lambda = 1$.

The full implementation in Python uses object-oriented programming to encapsulate all necessary functionality into a single class, inheriting from PyTorch's `nn.Module`:

```python
import torch
import torch.nn as nn

class PoissonPINN(nn.Module):
    def __init__(self):
        super().__init__() # Call the constructor of the parent class (nn.Module)
        # Define the neural network architecture using nn.Sequential
        self.net = nn.Sequential(
            nn.Linear(1, 20), nn.Tanh(),  # Input layer (1 feature) to hidden layer (20 neurons)
            nn.Linear(20, 20), nn.Tanh(), # Hidden layer to another hidden layer
            nn.Linear(20, 1)              # Hidden layer to output layer (1 output)
        )

    # Defines the forward pass of the network: computes u_theta(x)
    def forward(self, x):
        return self.net(x)

    # Computes the residual of the PDE: d^2u/dx^2 + f(x)
    def pde_residual(self, x):
        # Ensure x requires gradients for differentiation
        x.requires_grad_(True)
        u = self.forward(x) # Compute network output u(x)

        # Compute first derivative du/dx using automatic differentiation
        du_dx = torch.autograd.grad(
            outputs=u, inputs=x,
            grad_outputs=torch.ones_like(u), # Jacobian-vector product (scalar output)
```

```
            create_graph=True # Allows for higher-order derivatives
        )[0] # grad returns a tuple, we need the first element

        # Compute second derivative d2u/dx2
        d2u_dx2 = torch.autograd.grad(
            outputs=du_dx, inputs=x,
            grad_outputs=torch.ones_like(du_dx),
            create_graph=True
        )[0]

        # Define the source term f(x) = pi^2 * sin(pi*x)
        f_x = (torch.pi**2) * torch.sin(torch.pi * x)

        # PDE residual: d2u/dx2 + f(x) (from -d2u/dx2 = f(x) => d2u/dx2 + f(x) = 0)
        return d2u_dx2 + f_x

    # Computes the total loss function
    def loss(self, x_interior, x_boundary_points):
        # PDE loss at interior points
        r_interior = self.pde_residual(x_interior)
        loss_pde = torch.mean(r_interior**2) # Mean squared residual

        # Boundary condition loss: u(x) should be 0 at boundary points
        u_boundary = self.forward(x_boundary_points)
        # For u(0)=0, u(1)=0, the target is 0. So, (u_boundary - 0)^2
        loss_bc = torch.mean(u_boundary**2)

        # Total loss (lambda = 1 for simplicity, as stated in text)
        return loss_pde + loss_bc
```

Listing 5. A minimal Python class using PyTorch for a PINN solving the 1D Poisson equation.

The neural network structure is defined in the class constructor `__init__`, leveraging PyTorch's `nn.Module` base class and its predefined layers like `nn.Linear` and activation functions like `nn.Tanh`. The `forward` method computes the network output $u_\theta(x)$. Critically, the `pde_residual` method uses PyTorch's automatic differentiation capabilities (`torch.autograd.grad`) to compute the necessary second derivative $\frac{d^2u_\theta}{dx^2}$ required by the PDE residual in Equation 11.2. The `loss` method then combines both the PDE residual loss and the boundary condition loss into a single scalar value, as described in Equation 11.4.

This object-oriented design, where the `PoissonPINN` class encapsulates the network architecture, the forward pass, the physics-informed residual calculation, and the loss computation, not only mirrors the mathematical structure of the problem but also promotes modularity and extensibility. By encapsulating the solution strategy into a class, we make it straightforward to adapt the code for other differential equations, different domain geometries, or alternative boundary conditions, often by modifying specific methods or inheriting from this class to create more specialized PINN solvers.

To train the network, we discretize the domain $x \in (0, 1)$ by selecting a set of collocation points $\{x_i\}_{i=1}^{N_r}$ for the residual loss and using the boundary points $\{x = 0, 1\}$ for the boundary loss. The training loop is straightforward and mirrors standard practice in PyTorch:

```
# Ensure the model is on the correct device (e.g., CPU or GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

model = PoissonPINN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

num_epochs = 10000
num_interior_points = 100 # Number of collocation points inside the domain
```

```python

for epoch in range(num_epochs):
    optimizer.zero_grad() # Clear previous gradients

    # Generate random interior points for PDE loss, ensuring they are on the device
    x_interior_data = torch.rand(num_interior_points, 1, device=device) # Points in [0,1)

    # Define boundary points, ensuring they are on the device
    # These are the points where u(x) = 0
    x_boundary_data = torch.tensor([[0.0], [1.0]], device=device, dtype=torch.float32)

    # Calculate loss
    total_loss = model.loss(x_interior_data, x_boundary_data)

    # Backpropagation: compute gradients of the loss w.r.t. model parameters
    total_loss.backward()
    optimizer.step() # Update network parameters based on gradients

    if epoch % 1000 == 0:
        print(f"Epoch {epoch}: Loss = {total_loss.item():.6f}")

# After training, the model can be used for predictions.
# Example: predict at a few points
# model.eval() # Set model to evaluation mode (important for layers like dropout, batchnorm)
# with torch.no_grad(): # Disable gradient calculations for inference
#     test_points = torch.linspace(0, 1, 100, device=device).unsqueeze(1)
#     predictions = model(test_points)
#     # 'predictions' now holds the network's approximation of u(x)
```

Listing 6. Training loop for the PoissonPINN model using PyTorch.

After training, the network's output $u_\theta(x)$ can be evaluated on a dense grid and compared to the exact solution $\sin(\pi x)$. This not only serves as a sanity check but also demonstrates the effectiveness of the physics-informed approach.

The figure below shows the predicted solution after training, together with the analytical result:

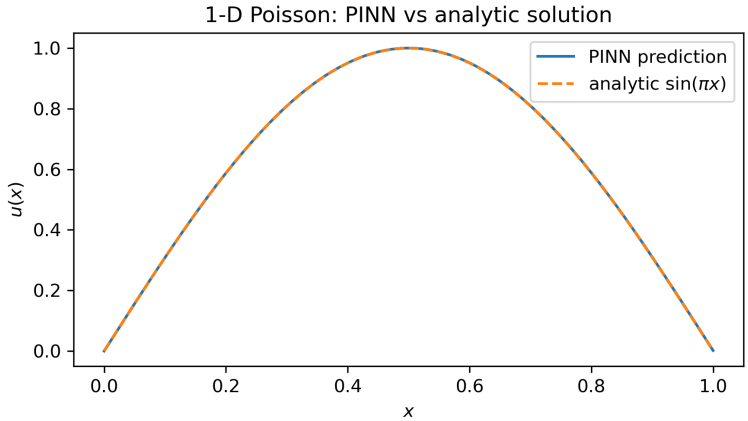

Figure 12. Predicted solution $u_\theta(x)$ of the 1D Poisson equation (Equation 11.1) compared to the exact solution $\sin(\pi x)$. The PINN successfully captures the shape of the exact solution.

This simple example shows how PINNs embed the physical structure of a problem into the training process of a neural network. The loss function incorporates the governing differential equation directly, enabling the model

to learn solutions with little or no labeled training data (beyond boundary/initial conditions). Moreover, the object-oriented design in Python allows for a clean separation between the model structure (the neural network itself), the differential physics (encoded in the residual and loss methods), and the training logic. This makes the code both easy to understand and adaptable for more complex systems, such as the time-dependent Schrödinger equation addressed later in this thesis.

Having now examined Python as a powerful tool for implementing scientific software, with a particular focus on its core features and object-oriented design principles as applied to PINNs, we are well equipped to address the next layer of implementation detail. The robustness, clarity, and maintainability of any scientific codebase rely heavily on the use of sound software engineering practices. As we transition to further explorations or more complex implementations of Physics-Informed Neural Networks, we will continue to emphasize the principles of good coding practice that underpin sustainable and reproducible research in computational physics.

Chapter 12

12. Implementation: Physics-Informed Neural Networks

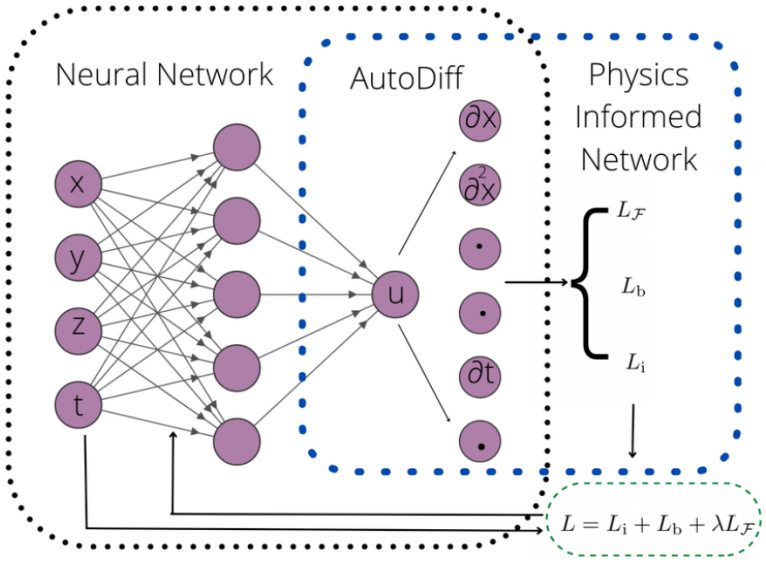

Figure 13. Architecture of a Physics-Informed Neural Network (PINN), illustrating the integration of input variables, neural network layers, automatic differentiation, and the incorporation of physical laws into the loss function.[29]

> "Learning is not compulsory... neither is survival."[11]
>
> W. Edwards Deming

Having established the theoretical underpinnings of Physics-Informed Neural Networks (PINNs) and Finite Element Methods (FEM) in Part II (specifically, building upon the theoretical framework presented in 2 - 7 vi), and outlined the comparative framework in Part III, we now turn to the specifics of the PINN implementation employed in this thesis. This section outlines the specific implementation details of the PINNs employed to approximate solutions for the classical and quantum systems under investigation. The successful translation of theoretical concepts into functional code capable of generating reliable scientific results hinges not only on understanding the core algorithms but also on adhering to sound software engineering principles and making careful choices regarding network structure, training processes, and the incorporation of underlying physics constraints. These aspects are elaborated upon in the following subsections, delving into the practical aspects of constructing and training the PINNs.

i. Good coding practices

Beyond the mathematical formulation, the practical implementation of computational models like PINNs requires careful consideration of software structure and maintainability. Adhering to good coding practices is paramount for ensuring the robustness, reproducibility, and extensibility of the research code. In the context of this work, several key practices were adopted to manage the inherent complexity of combining deep learning frameworks with physics-based constraints.

A modular design philosophy was central to the implementation. The code was structured to separate distinct functionalities, such as the neural network definition, the formulation of Partial Differential Equation (PDE)

residuals, the construction of the composite loss function, data sampling strategies, and the training loop itself. This separation facilitates independent testing of components, simplifies debugging, and allows for easier modification or replacement of individual parts for instance, experimenting with different network architectures or optimization algorithms without disrupting the entire codebase. Where appropriate, Object-Oriented Programming (OOP) principles, as discussed in Section **??** and **??**, were leveraged, particularly in the Python implementation using PyTorch. Encapsulating network models, PDE definitions, or training configurations within classes enhances code organization and clarity.

Furthermore, comprehensive commenting and documentation were maintained throughout the codebase. Comments were used not merely to restate the code's function but to explain the underlying logic, the physical meaning of variables, and the rationale behind specific implementation choices, particularly for complex numerical or physics-related steps. Function and class docstrings provide clear descriptions of purpose, arguments, and return values, aiding both understanding and future use. Version control, primarily using Git, was employed throughout the development process to track changes, manage different experimental branches, and ensure the traceability of results back to specific code versions. While extensive unit testing frameworks were not implemented due to time constraints, validation checks and comparison against known analytical solutions (where available) served as crucial verification steps. These practices collectively contribute to a more reliable and understandable codebase, which is essential for building confidence in the numerical results and underpinning the transition to the specific architectural choices detailed next.

ii. Neural Network Architecture

Following the principles of good coding practice, the core computational engine of the PINN is the neural network itself. The architecture of this network, its input and output structure, the arrangement and size of its hidden layers, and the choice of activation functions, fundamentally dictates its capacity to approximate the complex solution functions of the target PDEs and satisfy the embedded physical laws. As introduced theoretically (e.g., in Section 5 v), the networks employed in this study are primarily Multi-Layer Perceptrons (MLPs), a standard type of feedforward neural network. The specific configuration of the MLP, however, is tailored to the dimensionality and nature of each physical system being investigated. The subsequent subsections detail the choices made regarding the input/output representation and the internal structure composed of hidden layers and their associated non-linear activation functions, which together define the network's hypothesis space.

1. Input and Output Representation

The interface between the physical problem and the neural network approximator is defined by the network's input and output layers. Consistent with the standard PINN formulation for solving PDEs, the input layer is designed to accept the independent variables that define the problem domain. For the systems studied in this thesis, which primarily involve dynamics in two spatial dimensions potentially evolving over time, the input vector $\mathbf{x}$ typically consisted of the spatial coordinates (x, y) and, where applicable, the time coordinate t. For instance, in solving the 2D time-dependent diffusion equation (Section **??**) or the 2D Time-Dependent Schrödinger Equation (TDSE, Section 13 iv), the network received a 3-dimensional input $\mathbf{x} = (x, y, t)$. For steady-state problems like the Poisson equation (Section 13 iii) or time-independent quantum systems, the input dimensionality would reduce accordingly (e.g., to (x, y) or just r for radially symmetric cases).

The output layer, conversely, is structured to produce the network's approximation of the dependent variable(s) governed by the PDE. For classical PDEs involving a single, real-valued field variable, such as the concentration $u(x, y, t)$ in the diffusion equation, the solution $u_\theta(x, y)$ for the Poisson equation, or the velocity/pressure fields in the Navier-Stokes equations (Section 13 ii), the output layer typically consisted of a single neuron (or multiple single neurons for vector fields). However, for quantum mechanical problems like the TDSE, the wavefunction

$\psi(x, y, t)$ is inherently complex-valued. To accommodate this, the network's output layer was configured with two neurons, predicting the real and imaginary parts of the wavefunction separately. Denoting the network's trainable parameters (weights and biases) by θ, the approximated wavefunction $\psi_\theta(x, y, t)$ is constructed as:

$$\psi_\theta(x, y, t) = u_\theta(x, y, t) + iv_\theta(x, y, t), \tag{12.1}$$

where u_θ and v_θ are the two distinct scalar outputs generated by the final layer of the network. This input-output setup establishes the network's task: to learn the mapping from domain coordinates to solution values, a transformation achieved through the processing performed by the hidden layers.

2. Hidden Layers and Activation Functions

The core computational capacity of the MLP resides in its hidden layers, situated between the input and output layers. These layers perform the non-linear transformations necessary to approximate the potentially complex solutions of the governing PDEs. In this work, the hidden layers were implemented as fully connected (or dense) layers, meaning each neuron in a given layer receives input from all neurons in the preceding layer. The representational power of the network is primarily determined by its depth (the number of hidden layers) and its width (the number of neurons within each hidden layer). These architectural hyperparameters were chosen based on the specific problem's complexity and through preliminary experimentation to balance expressivity with training efficiency. Across the experiments conducted in this study, network architectures typically featured between 4 and 8 hidden layers, with layer widths ranging from approximately 32 to 512 neurons (consistent with configurations detailed in Section 13, e.g., Tables II, IV, VI).

Crucially, the ability of the MLP to model non-linear phenomena stems from the application of a non-linear activation function after the linear transformation (weighted sum plus bias) in each neuron. Without these non-linearities, a deep MLP would collapse into an equivalent single linear transformation, severely limiting its approximation capabilities (as discussed in Section **??**). Based on theoretical considerations and empirical performance observed in preliminary tests and the literature, specific activation functions were selected for the hidden layers. For several systems, including the Navier-Stokes and TDSE problems, the hyperbolic tangent (tanh) function was used (see Tables IV, VI). Its smooth, bounded nature and zero-centered output often facilitate stable training. For other cases, including the diffusion equation benchmark (Table II), the Sigmoid Linear Unit (SiLU), also known as the swish function for $\beta = 1$, was employed. The SiLU activation function is defined as:

$$\text{Swish}(z) = z \cdot \sigma(\beta z) = \text{SiLU}(z) = z \cdot \sigma(z) = \frac{z}{1 + e^{-z}}, \tag{12.2}$$

where z denotes the input to the activation and $\sigma(z)$ is the standard logistic sigmoid. Unlike the more commonly used ReLU or tanh functions, SiLU is both smooth (i.e., continuously differentiable) and non-monotonic. These properties are particularly advantageous for PINNs, as smoothness facilitates the computation of higher-order derivatives in the loss function, while non-monotonicity can enhance the network's expressiveness. Moreover, SiLU has demonstrated stable gradient behavior during training, which contributed to improved convergence in many of the systems considered in this work. The careful selection of network depth, width, and activation functions collectively shapes the function space the PINN can represent, directly impacting its ability to accurately embed the physical laws discussed next.

iii. Embedding Physical Laws

While the neural network architecture described in Section 5 v provides the necessary functional complexity to represent potential solutions, it does not inherently guarantee that the learned function satisfies the governing physical laws of the system under study. The defining characteristic of Physics-Informed Neural Networks is the explicit integration of these physical laws, typically expressed as Partial Differential Equations (PDEs), as

introduced theoretically in Section 6 ii and 6 vii into the training process itself. Rather than relying solely on pre-existing data points, the network learns the solution by being trained to minimize a composite loss function. This function quantifies discrepancies, including data-based loss (from boundary/initial conditions) and, crucially, physics-based loss (from the PDE). This embedding acts as a strong physical regularizer, guiding the optimization towards solutions that are not only consistent with any available data but also dynamically valid according to the underlying physics. The following subsections detail the two primary mechanisms through which this physical knowledge is embedded in the PINNs implemented for this thesis: the computation of PDE residuals using automatic differentiation and the construction of a composite loss function that balances physical consistency with data fidelity.

1. Trial Solution and Automatic Differentiation

The core mechanism for enforcing the PDE constraint involves evaluating how well the neural network's output satisfies the governing equation. Let $\psi_\theta(\mathbf{x})$ represent the output of the neural network (where $\mathbf{x}$ includes spatial and potentially temporal coordinates, and θ denotes the trainable network parameters). For a general PDE of the form $\mathcal{L}[\psi](\mathbf{x}) = f(\mathbf{x})$, where $\mathcal{L}$ is a differential operator and f is a source term, the network's residual is defined as:

$$\mathcal{R}(\mathbf{x};\theta) = \mathcal{L}[\psi_\theta(\mathbf{x})] - f(\mathbf{x}). \tag{12.3}$$

Calculating this residual requires computing the derivatives of the network's output ψ_θ with respect to its inputs $\mathbf{x}$, as dictated by the operator $\mathcal{L}$ (e.g., $\partial\psi_\theta/\partial t$, $\partial^2\psi_\theta/\partial x^2$).

A key enabling technology for PINNs, and central to this implementation, is Automatic Differentiation (AD). Modern deep learning frameworks like PyTorch[43] and TensorFlow[3] provide robust AD capabilities that allow for the exact computation of derivatives of complex functions, such as neural networks, with respect to their inputs, up to machine precision. Unlike numerical differentiation methods (e.g., finite differences) which introduce truncation errors, or symbolic differentiation which can lead to complex and inefficient expressions, AD leverages the chain rule through the network's computational graph to efficiently compute exact derivatives. This was crucial for accurately evaluating terms like the Laplacian ($\nabla^2\psi_\theta$) or time derivatives ($\partial\psi_\theta/\partial t$) needed for the physics loss component. The choice of smooth activation functions like tanh or SiLU (Section 12 ii 2) ensures that the necessary derivatives are well-defined throughout the domain.

In this work, the network output $\psi_\theta(\mathbf{x})$ was generally used directly as the approximation of the physical solution. Boundary and initial conditions were primarily enforced as 'soft constraints' by incorporating penalty terms into the loss function, as detailed below. This contrasts with alternative 'hard constraint' approaches where the trial solution $\tilde{\psi}_\theta(\mathbf{x})$ is constructed explicitly to satisfy certain conditions by design (e.g., $\tilde{\psi}_\theta = \psi_{BC} + g(\mathbf{x})\psi_\theta$, where $g(\mathbf{x})$ vanishes at the boundary). While hard constraints can sometimes accelerate convergence, the soft constraint approach offers greater flexibility in implementation, especially for complex geometries or boundary condition types, and was the principal method employed here, relying on the composite loss function to enforce all necessary conditions.

2. Loss Function Composition

Having computed the PDE residual using automatic differentiation, the next step is to integrate this physical constraint, along with boundary and initial conditions, into a single objective function that the optimizer seeks to minimize. This is achieved through a composite loss function, $\mathcal{L}_{\text{total}}$, which aggregates the errors from different sources. As outlined theoretically in Section 6 vii 3, the total loss is a weighted sum of individual loss components:

$$\mathcal{L}_{\text{total}}(\theta) = w_{\text{PDE}}\mathcal{L}_{\text{PDE}}(\theta) + w_{\text{BC}}\mathcal{L}_{\text{BC}}(\theta) + w_{\text{IC}}\mathcal{L}_{\text{IC}}(\theta) + w_{\text{reg}}\mathcal{L}_{\text{reg}}(\theta). \tag{12.4}$$

Each component quantifies a specific aspect of the desired solution's properties:

- **Physics Loss ($\mathcal{L}_{\mathbf{PDE}}$):** This term measures the extent to which the network's output violates the governing PDE. It is typically calculated as the Mean Squared Error (MSE) of the PDE residual $\mathcal{R}(\mathbf{x};\theta)$ (Equation (12.3)) evaluated over a set of N_r collocation points $\{\mathbf{x}_r^{(i)}\}_{i=1}^{N_r}$ sampled within the problem domain's interior:

$$\mathcal{L}_{\text{PDE}}(\theta) = \frac{1}{N_r}\sum_{i=1}^{N_r} |\mathcal{R}(\mathbf{x}_r^{(i)};\theta)|^2. \tag{12.5}$$

  Minimizing this term compels the network to learn a function ψ_θ that conforms to the underlying physical laws described by the PDE across the entire domain.
- **Boundary Condition Loss ($\mathcal{L}_{\mathbf{BC}}$):** This term penalizes deviations of the network's prediction from the prescribed boundary conditions (BCs). For Dirichlet BCs where $\psi(\mathbf{x}) = g(\mathbf{x})$ on the boundary $\partial\Omega$, it is computed as the MSE between $\psi_\theta(\mathbf{x}_{bc}^{(i)})$ and $g(\mathbf{x}_{bc}^{(i)})$ over a set of N_{bc} points $\{\mathbf{x}_{bc}^{(i)}\}_{i=1}^{N_{bc}}$ sampled on the boundary:

$$\mathcal{L}_{\text{BC}}(\theta) = \frac{1}{N_{bc}}\sum_{i=1}^{N_{bc}} |\psi_\theta(\mathbf{x}_{bc}^{(i)}) - g(\mathbf{x}_{bc}^{(i)})|^2. \tag{12.6}$$

  For problems with Neumann or other boundary condition types, the formulation of $\mathcal{L}_{\text{BC}}$ was adjusted accordingly, potentially involving derivatives of ψ_θ evaluated at the boundary, again computed via automatic differentiation.
- **Initial Condition Loss ($\mathcal{L}_{\mathbf{IC}}$):** For time-dependent problems, this term enforces the initial state of the system at $t = 0$. Given an initial condition $\psi(\mathbf{x}_{spatial}, 0) = \psi_0(\mathbf{x}_{spatial})$, the loss is the MSE between $\psi_\theta(\mathbf{x}_{ic}^{(i)}, 0)$ and $\psi_0(\mathbf{x}_{ic}^{(i)})$ over N_{ic} points $\{\mathbf{x}_{ic}^{(i)}\}_{i=1}^{N_{ic}}$ sampled within the spatial domain at the initial time:

$$\mathcal{L}_{\text{IC}}(\theta) = \frac{1}{N_{ic}}\sum_{i=1}^{N_{ic}} |\psi_\theta(\mathbf{x}_{ic}^{(i)}, 0) - \psi_0(\mathbf{x}_{ic}^{(i)})|^2. \tag{12.7}$$

- **Regularization Loss ($\mathcal{L}_{\mathbf{reg}}$):** Optionally, a standard regularization term, such as L2 weight decay (as discussed theoretically in Section 6 iii), can be added to prevent overfitting and control the magnitude of network weights, promoting smoother solutions:

$$\mathcal{L}_{\text{reg}}(\theta) = \sum_l ||W^{(l)}||_F^2, \tag{12.8}$$

  where $||W^{(l)}||_F^2$ is the squared Frobenius norm of the weight matrix for layer l. A small L2 regularization, typically implemented via the optimizer's weight decay parameter (e.g., values around 10^{-4} to 10^{-6}), was consistently applied to aid in stabilizing the training process. The effective contribution $w_{\text{reg}}\mathcal{L}_{\text{reg}}$ in Equation (12.4) was thus managed through the weight w_{reg} and the optimizer settings.

The weighting factors $w_{\text{PDE}}, w_{\text{BC}}, w_{\text{IC}}, w_{\text{reg}}$ are crucial hyperparameters that balance the relative importance of satisfying the PDE versus matching the boundary/initial data and controlling model complexity. Their values can significantly impact training dynamics and final solution accuracy. In this work, these weights were sometimes set manually based on preliminary experiments or adapted dynamically during training using techniques discussed later (see Section 12 v and Table VI). Minimizing this carefully composed loss function $\mathcal{L}_{\text{total}}(\theta)$ guides the network parameter updates, ultimately yielding a function ψ_θ that approximates the true physical solution. The effectiveness of this minimization, however, depends significantly on how the points used to evaluate these loss terms are selected, which is the subject of the next section.

iv. Sampling Collocation Points

The construction of the composite loss function, as detailed in Section 6 vii 3 and Section 12 iii 2, relies on evaluating the PDE residuals, boundary condition mismatches, and initial condition errors at discrete sets of

points. The effectiveness of the PINN training process critically depends on the selection and distribution of these points at which the various loss components ($\mathcal{L}_{\text{PDE}}$, $\mathcal{L}_{\text{BC}}$, $\mathcal{L}_{\text{IC}}$) are evaluated. An appropriate sampling strategy ensures that the loss function accurately reflects the global error across the entire domain, guiding the optimizer towards a valid solution without being biased by uneven point distributions. Furthermore, the number and placement of these points directly impact the computational cost per training iteration. Distinct strategies were employed for sampling points in the interior of the domain versus on its boundaries (including the initial time boundary).

1. Uniform and Latin Hypercube Sampling for Interior, Boundary, and Initial Points

The physics-based loss, $\mathcal{L}_{\text{PDE}}$ (Equation (12.5)), was evaluated at a set of N_r collocation points sampled within the interior of the problem's spatio-temporal domain. For multi-dimensional problems such as diffusion, these points were generated using quasi-random uniform sampling or Latin Hypercube Sampling (LHS) within the specified domain bounds (e.g., $x \in [0, L_x], y \in [0, L_y], t \in [0, T]$). LHS is a stratified sampling technique that can provide more even coverage of the domain compared to simple uniform sampling. Typically, large sets (N_r on the order of several thousands) of these points were generated.

Points used to evaluate the initial condition loss ($\mathcal{L}_{\text{IC}}$, Equation (12.7)) and boundary condition loss ($\mathcal{L}_{\text{BC}}$, Equation (12.6)) were sampled specifically from the relevant boundaries of the spatio-temporal domain. For the initial condition ($t = 0$), N_{ic} points were generated by sampling the spatial coordinates (x, y) (uniformly or via LHS) within the domain while fixing the time coordinate to zero. A substantial number of initial points (e.g., one thousand) were typically used. For spatial boundaries (e.g., $x = 0$, $x = L_x$, $y = 0$, $y = L_y$), N_{bc} points were generated directly on these surfaces or lines, typically by fixing one coordinate and sampling the others (uniformly or via LHS) within their respective ranges. A comparable number of points (e.g., one thousand distributed across all boundaries) were sampled across all relevant spatial boundaries.

To prevent the network from overfitting to a fixed set of points and to ensure broader coverage of the domain over the course of training, a resampling strategy was employed, generating new sets of interior, boundary, and initial points periodically (e.g., every thousand epochs). This dynamic sampling strategy helps the network generalize better.

For certain problems like the one-dimensional radial Schrödinger equation, a more structured approach was adopted for the radial coordinate r. An adaptive grid was generated, combining logarithmically spaced points concentrated near $r = 0$ (where the Coulomb potential singularity demands higher resolution) with linearly spaced points further out. This approach aimed to capture the solution's behavior more effectively across different scales, and could optionally densify the grid near expected locations of wavefunction nodes. In the context of the radial two-electron problem, explicit boundary sampling for $\mathcal{L}_{\text{BC}}$ was less central, as boundary conditions at $r = 0$ and $r \to \infty$ were often handled implicitly through the analytical structure incorporated into the network's forward pass or by specific terms in the PDE residual.

2. Adaptive Sampling or Residual-Based Sampling

While uniform or quasi-random sampling methods like LHS provide a good baseline for domain coverage, they do not inherently focus computational effort on regions where the approximation error or the PDE residual is largest. In problems exhibiting sharp gradients, boundary layers, singularities, or complex dynamics localized in specific areas, uniform sampling might require an excessively large number of points to adequately resolve these features. Adaptive sampling strategies aim to improve efficiency by concentrating collocation points in these challenging regions.

A common approach is residual-based adaptive sampling. The core idea is to periodically evaluate the PDE residual $\mathcal{R}(\mathbf{x}; \theta)$ (Equation (12.3)) over a large candidate set of points spanning the domain. Points associated

with higher residual values are then selected with higher probability for inclusion in the training batch for the next set of optimization steps. This focuses the network's attention on areas where it currently fails most significantly to satisfy the physical constraints.

However, implementing adaptive sampling introduces additional complexity into the training pipeline. While the potential benefits are recognized, the primary sampling strategy employed throughout the majority of the benchmark problems presented in this thesis relied on the simpler, yet robust, approach of using large, periodically resampled sets of uniformly or LHS-generated points, as described in Section 12 iv 1. This choice prioritized implementation simplicity and stability across diverse problems. Adaptive methods remain an important area for future investigation.

v. Training and Optimization

Once the neural network architecture (Section 6 vi 1), the physics-embedding loss function (Section 12 iii), and the point sampling strategy (Section 12 iv) are defined, the final stage is the training process itself. Training involves iteratively adjusting the learnable parameters θ (weights and biases) of the neural network to minimize the total loss function $\mathcal{L}_{\text{total}}(\theta)$ (Equation 12.4). This minimization process drives the network's output $\psi_\theta(\mathbf{x})$ towards a function that accurately approximates the true solution of the PDE while satisfying the specified initial and boundary conditions. Given the typically high-dimensional parameter space and the often non-convex nature of the loss landscape for neural networks, this optimization is performed using iterative, gradient-based methods.

1. Gradient Computation

Gradient-based optimization algorithms require the computation of the gradient of the total loss function $\mathcal{L}_{\text{total}}$ with respect to all trainable network parameters θ. This gradient, $\nabla_\theta \mathcal{L}_{\text{total}}$, indicates the direction of steepest ascent in the loss landscape; the optimizer takes steps in the opposite direction to minimize the loss. While Section 12 iii 1 discussed the use of Automatic Differentiation (AD) to compute derivatives of the network output with respect to its *inputs* (needed for the PDE residual), computing the gradient with respect to the *parameters* is achieved through the backpropagation algorithm.

Backpropagation is essentially an efficient application of the chain rule applied recursively backward through the network's computational graph, starting from the final loss value. As detailed theoretically in Section **??**, it allows for the computation of the partial derivative of the loss with respect to every weight and bias in the network. Crucially, AD frameworks like PyTorch seamlessly integrate backpropagation. When the total loss $\mathcal{L}_{\text{total}}$ (a scalar value computed based on the network's output for a batch of sampled points) is calculated, invoking the '.backward()' method on the loss tensor automatically triggers the backpropagation process. This populates the '.grad' attribute for each network parameter tensor with the corresponding partial derivative $\partial \mathcal{L}_{\text{total}}/\partial \theta_i$. The efficiency of backpropagation makes training even very deep and wide networks feasible. These computed gradients form the essential input for the optimization algorithms that perform the actual parameter updates.

2. Optimization Algorithm, Monitoring, and Stopping Criteria

With the gradients $\nabla_\theta \mathcal{L}_{\text{total}}$ computed via backpropagation, an optimization algorithm is employed to update the network parameters θ iteratively. The goal is to step towards a minimum of the loss function. While basic Gradient Descent (GD) or Stochastic Gradient Descent (SGD) could be used (Section 5 vi 1), more advanced adaptive algorithms are generally preferred for training deep neural networks.

In this thesis, the primary optimizer used was Adam (Adaptive Moment Estimation) or its variant AdamW, as introduced in Section 5 vi 4. Key hyperparameters for the Adam/AdamW optimizer included the initial learning

rate η (often 'lr'), typically set in the range 10^{-3} to 10^{-4}, the exponential decay rates for the moment estimates ($\beta_1 \approx 0.9$, $\beta_2 \approx 0.999$), and the weight decay factor (used to implement L2 regularization, Equation (12.8)) with typical values around 10^{-4} to 10^{-6}. To further improve training dynamics, learning rate scheduling was often employed. Strategies included exponential decay (e.g., reducing 'lr' by a factor periodically) or more sophisticated schedules such as cosine annealing with warm restarts or one-cycle schedules (e.g., Figure 30). Training was performed iteratively, potentially using a full-batch or mini-batch approach depending on problem setup and memory constraints.

The training process was typically carried out for a predetermined, fixed number of epochs (e.g., ranging from 15,000 up to 150,000 depending on the problem complexity and desired accuracy). Progress was monitored throughout training by periodically logging key metrics (e.g., every 100 or 1000 epochs). These metrics included the total loss ($\mathcal{L}_{\text{total}}$) and its individual components ($\mathcal{L}_{\text{PDE}}$, $\mathcal{L}_{\text{BC}}$, $\mathcal{L}_{\text{IC}}$), as well as the current learning rate. For problems with known analytical solutions or properties, additional metrics like the calculated energy eigenvalue and its error relative to the exact value, or the L2 norm of the error between the PINN solution and the analytical wavefunction, were also tracked. To preserve the best-performing model, the network's state dictionary was saved, often tracking the model state corresponding to the epoch yielding the lowest observed training loss. Optionally, intermediate visualizations of the predicted solution field were generated for qualitative assessment.

vi. Boundary and Initial Conditions

The governing PDE alone typically admits an infinite number of solutions. To identify the unique, physically relevant solution for a specific scenario, it is essential to impose boundary conditions (BCs) that specify the solution's behavior at the spatial domain boundaries, and for time-dependent problems, initial conditions (ICs) that define the system's state at the starting time ($t = 0$). As alluded to in the discussion of the loss function composition (Section 6 vii 3 and Section 12 iii 2) and trial solutions (Section 12 iii 1), there are fundamentally two ways to incorporate these conditions within the PINN framework: through 'soft' constraints or 'hard' constraints.

The predominant approach adopted in this work was the use of soft constraints. In this method, the BCs and ICs are enforced by adding penalty terms ($\mathcal{L}_{\text{BC}}$ and $\mathcal{L}_{\text{IC}}$, Equations 12.6 and 12.7) directly to the total loss function (Equation 12.4). These terms measure the discrepancy between the neural network's predictions at the boundary/initial time and the prescribed condition values. The optimizer then attempts to minimize these discrepancies simultaneously with the PDE residual. The relative importance assigned to satisfying these conditions versus adhering to the PDE is controlled by the corresponding loss weights ($w_{\text{BC}}, w_{\text{IC}}$). This approach offers significant implementation flexibility.

The alternative, hard constraint approach, involves designing the network's output or trial solution $\tilde{\psi}_\theta(\mathbf{x})$ such that it satisfies the BCs and/or ICs by construction. While this guarantees satisfaction of the conditions and can sometimes lead to faster convergence, constructing appropriate trial solutions can be challenging, especially for complex domain geometries or intricate boundary condition types. Given the focus on comparing PINNs and FEM across a range of problems, the more general and flexible soft constraint method was deemed more suitable for the primary implementation in this thesis.

vii. Parallelization and Efficiency

The computational cost of training PINNs, particularly for complex PDEs or high-dimensional domains requiring large networks and numerous collocation points, can be substantial. Therefore, leveraging parallel computing resources is crucial for practical feasibility and efficient exploration of different model configurations. The implementation in this work capitalized on the inherent parallelism available in modern deep learning workflows and hardware.

The primary source of parallelization was GPU acceleration. The neural network operations and gradient computations are highly parallelizable tasks well-suited for Graphics Processing Units (GPUs). By utilizing the PyTorch framework, which provides seamless integration with NVIDIA GPUs via CUDA, the computationally intensive parts of the training process were offloaded from the CPU to the GPU. This resulted in significant speedups (specific hardware used is listed in Section 8 ii). All major PINN training runs reported in Section 13 were performed using GPU acceleration.

Efficiency is also influenced by data handling. Evaluating the loss function requires predictions for thousands of points. Processing all these points simultaneously (full-batch training) can exceed GPU memory. Therefore, mini-batching was employed where necessary. The full set of sampled points was divided into smaller batches, and network parameters were updated based on the average gradient computed over each mini-batch (as described in Section 5 vi 2). This allows training of large models within memory constraints, although it can introduce noise into the gradient estimates.

viii. Reproducibility and Random Seeds

Reproducibility is a cornerstone of scientific inquiry. In computational studies involving stochastic elements, such as training neural networks, achieving reproducibility requires careful management of randomness from sources like weight initialization, data sampling (Section 12 iv), and stochastic optimization.

To ensure reproducibility, random seeds were explicitly set for Python's 'random' module, NumPy, and PyTorch's CPU and GPU operations before initiating any training run. This ensures that weight initialization and data sampling are identical across runs with the same configuration. Where possible, PyTorch was configured to use deterministic algorithms.

Beyond managing algorithmic randomness, meticulous logging of hyperparameters and experimental configurations was maintained. For each simulation, key parameters were recorded, including network architecture, optimizer settings, number of collocation/boundary/initial points, loss function weights, training epochs, batch size, and random seeds used. This comprehensive logging is essential for replicating the exact conditions under which a particular result was obtained.

ix. Visualization and Postprocessing

Obtaining a trained PINN model is only part of the process; interpreting and evaluating the results require effective visualization and postprocessing. These steps are crucial for understanding the qualitative behavior of the learned solution, quantitatively assessing its accuracy, and diagnosing potential issues.

A variety of visualization methods were employed. For 2D spatial domains, heatmap or contour plots representing the predicted solution field $\psi_\theta(x, y, t)$ (or $u_\theta(x, y, t)$) at specific times were generated (e.g., Figures 14, 15, 9). These were often compared side-by-side with analytical or FEM solutions. The absolute or relative error field, $|\psi_\theta - \psi_{\text{ref}}|$, was frequently plotted.

Visualizing the training process provided insights. Plots tracking the evolution of the total loss and its components ($\mathcal{L}_{\text{PDE}}, \mathcal{L}_{\text{BC}}, \mathcal{L}_{\text{IC}}$) over epochs (e.g., Figures 5, 10, 12, 19) were essential for monitoring convergence. Similarly, plotting the learning rate schedule (Figure 11) or adaptive loss weights (Figure 13) helped understand optimization dynamics.

For quantum mechanical systems, postprocessing included calculating and plotting probability density $|\psi_\theta|^2$ (e.g., Figure 16) or computing expectation values, such as energy, and tracking their convergence (e.g., Table VII). The visualization toolkit primarily consisted of Python libraries like Matplotlib and Seaborn. These visualizations form an integral part of the methodology, enabling critical evaluation of the PINN performance presented in Part V.

Part VI

Results and Discussion

Chapter 13

13. Selected results

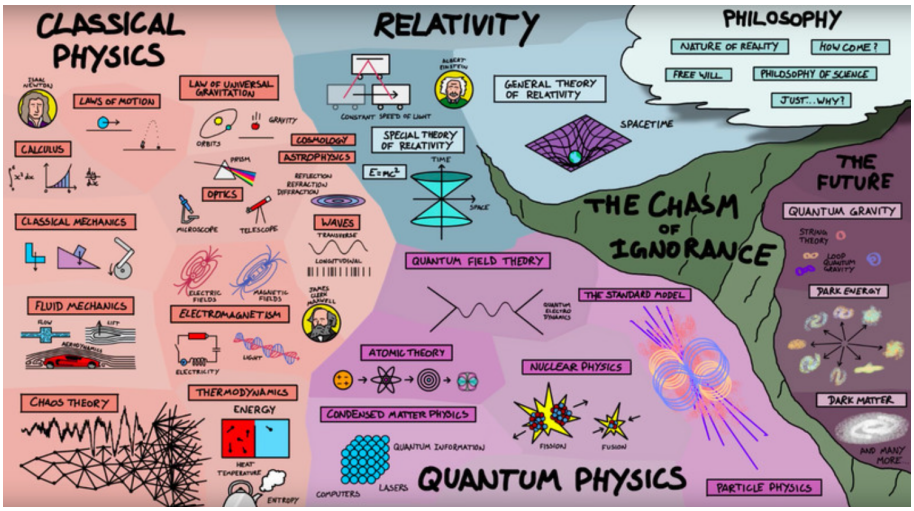

Figure 14. Schematic overview of different fields in physics, illustrating the systems explored in this thesis, from classical mechanics to quantum many-body problems.

> “In theory, there is no difference between theory and practice. But in practice, there is.[55]
>
> Jan L. A. van de Snepscheut”

In this section, we present the numerical results obtained for the range of systems, we have introduced in this thesis. We systematically compare the performance of PINNs and the FEM across different benchmarks and evaluation metrics, including loss/error, computational efficiency, and solution stability. It is important to note here that all simulations discussed in this thesis is run on the same hardware.

i. Diffusion equation

1. Network architecture

To begin our investigation, we start with a classical system for benchmarking PINNs: solving the Diffusion equation in 2D. As we can see from Table II, we specified the model to train for 150,000 epochs, which, for this configuration, translated to 1487.49 seconds of training time.

Additional parameters for the 2D Diffusion simulation are summarized in Table II.

PDE:	**Diffusion 2D**
Parameter	**Value**
Activation function	SiLU
Epochs	150,000
Model	3 inputs, 5 layers, 32 neurons per layer
Optimizer	Adam with learning rate 10^{-3} and weight decay 10^{-4}
Scheduler	Exponential with $\gamma = 0.96$

Table II. Model parameters for the 2D diffusion equation (PDE) used in this study. Listing the key parameters including activation functions, training epochs, model architecture, optimizer settings, scheduler details, and L_2 regularization constant $\lambda = 10^{-4}$.

For the corresponding three-dimensional case, a substantial increase in training time is observed. This increase is partly attributable to the use of a wider neural network (64 neurons per layer compared to 32) and a different activation function (Tanh instead of SiLU). However, it is also consistent with expectations, as higher-dimensional problems generally entail greater computational complexity and longer training durations. The parameters for the 3D simulation are detailed in Table III.

PDE:	**Diffusion 3D**
Parameter	**Value**
Activation function	Tanh
Epochs	150,000
Model	4 inputs, 5 layers, 64 neurons per layer
Optimizer	Adam with learning rate 10^{-3} and weight decay 10^{-4}
Scheduler	Exponential with $\gamma = 0.98$

Table III. Model parameters for the 3D diffusion equation (PDE) used in this study. Listing the key parameters including activation functions, training epochs, model architecture, optimizer settings, scheduler details, and L_2 regularization constant $\lambda = 10^{-4}$.

2. Initial Condition for the Diffusion Model

To begin with, we compare the initial conditions for the Diffusion equation with the analytical solution. Figure 15 displays the true (analytical) initial condition alongside the PINN-predicted initial condition for the 2D diffusion model using the SiLU activation function. The true initial condition depicts a sharply localized Gaussian distribution. The predicted initial condition successfully captures the central peak and overall shape; however, it appears slightly more diffused and less peaked compared to the true condition. This slight discrepancy suggests that while the model achieves a good approximation, there might be minor challenges in perfectly resolving the sharp gradients of the initial state or that the IC loss component has not been minimized perfectly. The colormaps indicate both achieve a similar maximum value, but the spread in the prediction is visibly wider. As will be discussed with Figure 16, the IC loss component does decrease significantly, but these visual differences highlight the nuance in fitting sharp initial states.

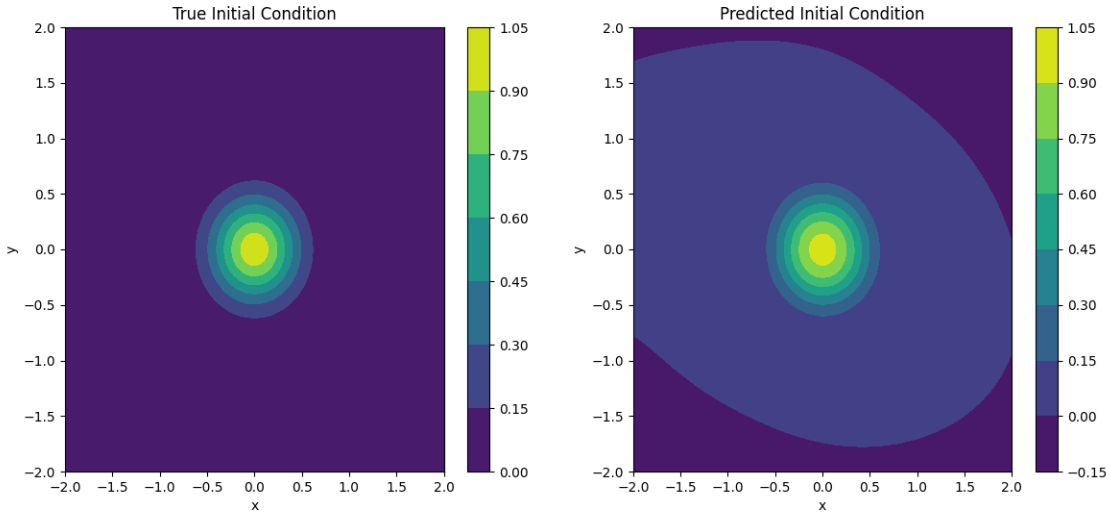

Figure 15. Comparison of the true initial condition (left) and the PINN-predicted initial condition (right) for the 2D diffusion model at $t = 0$. The prediction uses a SiLU activation function on a network with 5 layers and 32 neurons per layer. While the general form is captured, the predicted condition is slightly more spread out than the true, more localized distribution.

3. 2D Diffusion Results

The training dynamics of the PINN for the 2D diffusion equation are presented in Figure 16. This plot illustrates the convergence of different loss components—Total Loss, PDE Loss, Initial Condition (IC) Loss, and Boundary Condition (BC) Loss—over approximately 150,000 training epochs. All loss components exhibit a general downward trend, indicating that the model is learning to satisfy the PDE, initial, and boundary conditions. The Total Loss decreases by several orders of magnitude, reaching a value around 1×10^{-6} to 1×10^{-7} towards the end of the training. The IC loss shows a rapid initial decrease, followed by a more gradual reduction. The BC loss also decreases steadily, achieving a very low value, below 10^{-7} in the later stages. Notably, the PDE loss, while also decreasing, remains the most volatile component and settles at a higher level compared to IC and BC losses, characterized by frequent spikes throughout the training process. This instability in the PDE residual could be attributed to various factors, including the stochastic nature of training, the choice of hyperparameters, or the inherent difficulty in minimizing the PDE residual across the entire spatio-temporal domain. The overall trend suggests ongoing convergence.

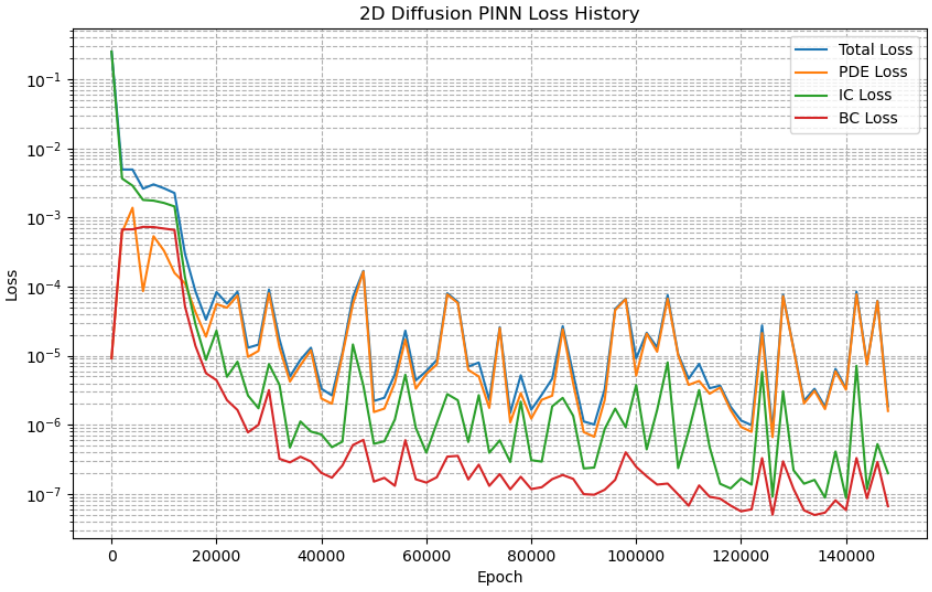

Figure 16. Loss history for the 2D Diffusion PINN model trained for approximately 150,000 epochs. The plot displays the total loss, PDE loss, initial condition (IC) loss, and boundary condition (BC) loss on a logarithmic scale. All losses show a decreasing trend, with the PDE loss exhibiting the most significant fluctuations.

Figure 17 provides a qualitative and quantitative comparison between the analytical solutions, PINN predictions, and their absolute errors for the 2D diffusion equation at four distinct time steps: $t = 0.00$, $t = 0.25$, $t = 0.50$, and $t = 1.00$. At the initial time ($t = 0.00$), the PINN prediction closely matches the analytical solution, which is a Gaussian peak. The absolute error plot shows that the largest errors are concentrated around the periphery of the peak and in low-value regions, with a Root Mean Squared Error (RMSE) of 3.89×10^{-4}. As time progresses to $t = 0.25$, the diffusion process causes the peak to spread and its amplitude to decrease (analytical max ≈ 0.032). The PINN prediction accurately captures this evolution, maintaining good agreement with the analytical solution. The RMSE at this stage is 9.39×10^{-5}. At $t = 0.50$, the solution is significantly more diffused (analytical max ≈ 0.00096). The PINN prediction continues to follow the general trend. The RMSE is 9.19×10^{-5}. By $t = 1.00$, the analytical solution has diffused to very low concentration values (analytical max $\approx 10^{-7}$). The PINN prediction also shows a highly diffused state. However, the predicted values do not decay as rapidly as the analytical solution, and the color scale of the prediction suggests magnitudes higher than the analytical solution. The RMSE is 1.42×10^{-4}. The absolute error plot highlights this discrepancy, showing that the PINN struggles to accurately represent these near-zero values over extended time evolution, with errors becoming comparable to the solution magnitude itself. Overall, the PINN demonstrates a strong capability to model the diffusion process. The accuracy, as measured by RMSE, is excellent, particularly at intermediate times. Challenges remain in precisely capturing very low solution magnitudes at later times.

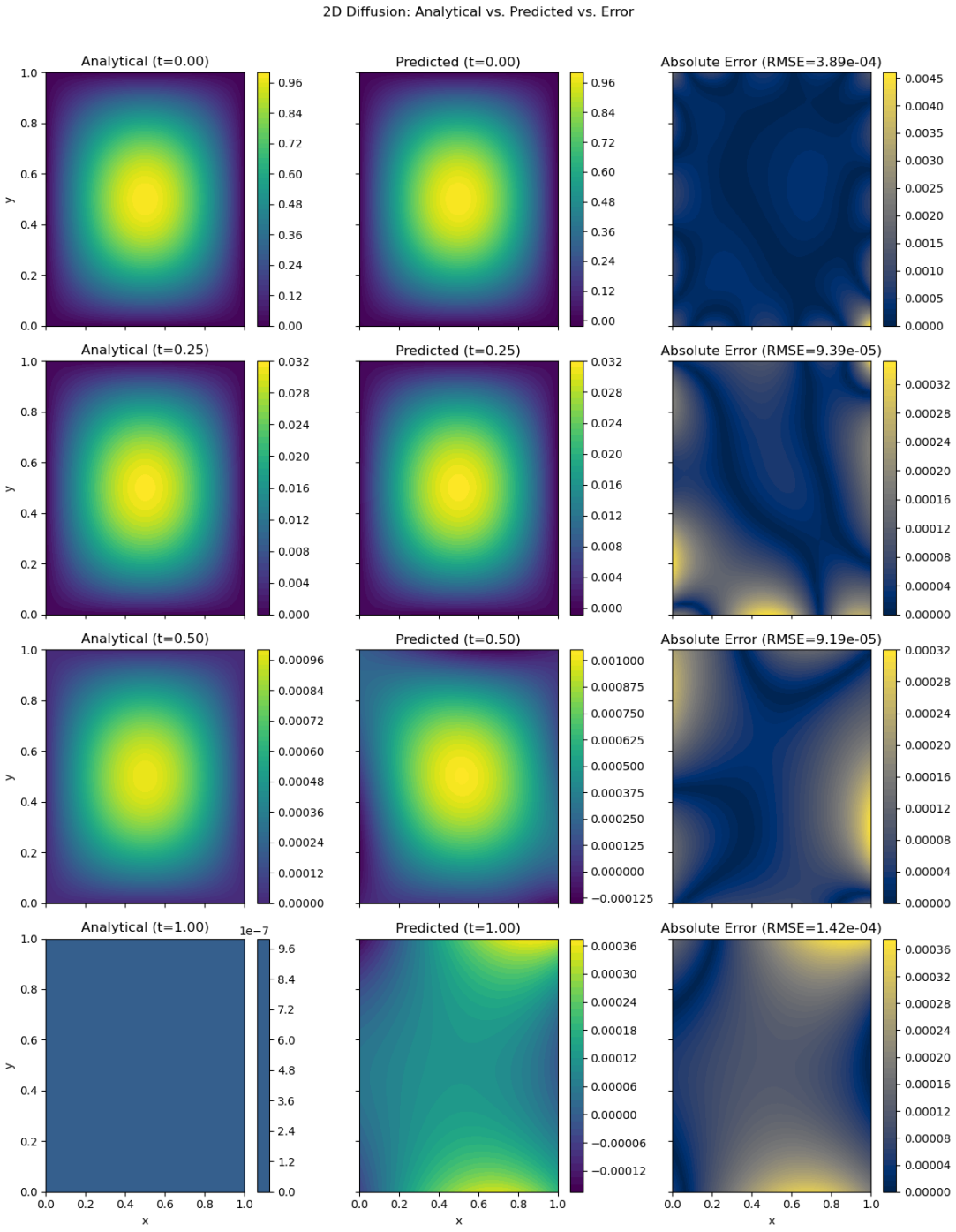

Figure 17. Comparison of analytical solutions, PINN predictions, and absolute error for the 2D diffusion equation at different time steps ($t = 0.00, t = 0.25, t = 0.50, t = 1.00$). Each row corresponds to a specific time, showing the analytical solution (left), the PINN-predicted solution (center), and the point-wise absolute error with RMSE (right).

To establish a benchmark using traditional numerical methods, a Finite Element Method (FEM) simulation of the 2D diffusion equation was performed using the Crank-Nicolson scheme. The computational performance of this FEM simulation was recorded: the setup time was 0.0414 seconds, the main solve loop for the entire simulation duration took 2.0964 seconds, averaging to 0.0210 seconds per time step. This solve time for a single simulation

run contrasts with the one-time training cost of the PINN (1487.49 seconds for 150,000 epochs as per Table II), after which the PINN can provide predictions rapidly.

Qualitative results from this FEM simulation are shown as snapshots at early to intermediate time steps in Figures 18, 19, and 20. Figure 18 presents the FEM-simulated solution at $t = 0.0$. Figure 19 displays the FEM-simulated solution at $t = 0.05$, where the initial peak has begun to spread. Figure 20 shows the FEM-simulated solution at $t = 0.25$, with more pronounced diffusion. These FEM snapshots depict the expected behavior of the system and can serve as a qualitative comparison for the PINN's predictions at similar time points (e.g., the $t = 0.25$ prediction in Figure 17).

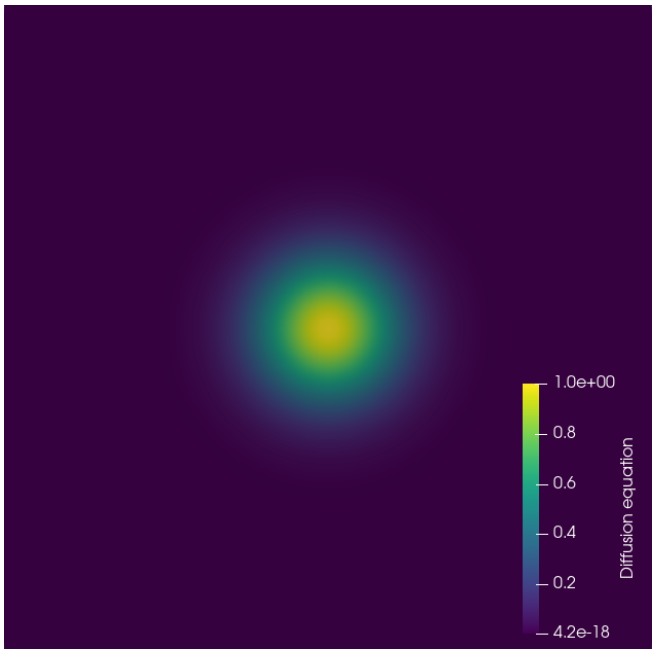

Figure 18. FEM-simulated solution for the 2D diffusion equation at $t = 0.0$. The colormap shows the concentration field, illustrating the initial Gaussian distribution as obtained by FEM.

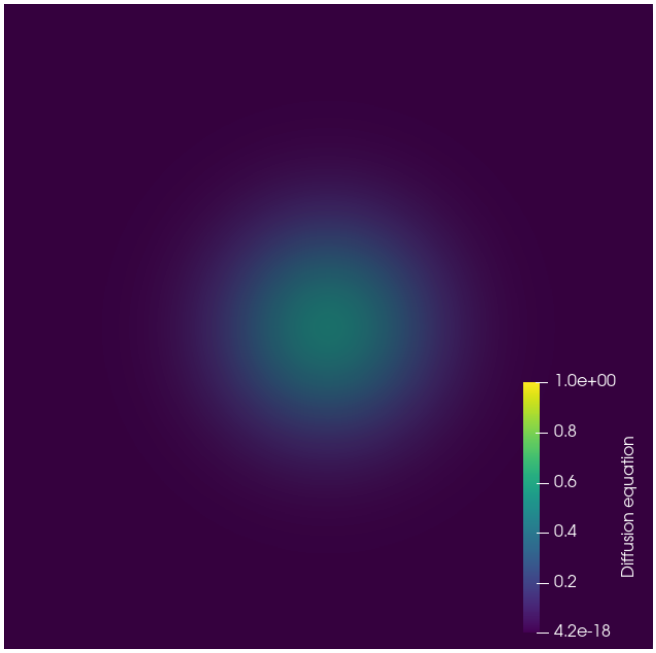

Figure 19. FEM-simulated solution for the 2D diffusion equation at $t = 0.05$. The concentration peak has spread and decreased in magnitude, characteristic of the diffusion process, as simulated by FEM.

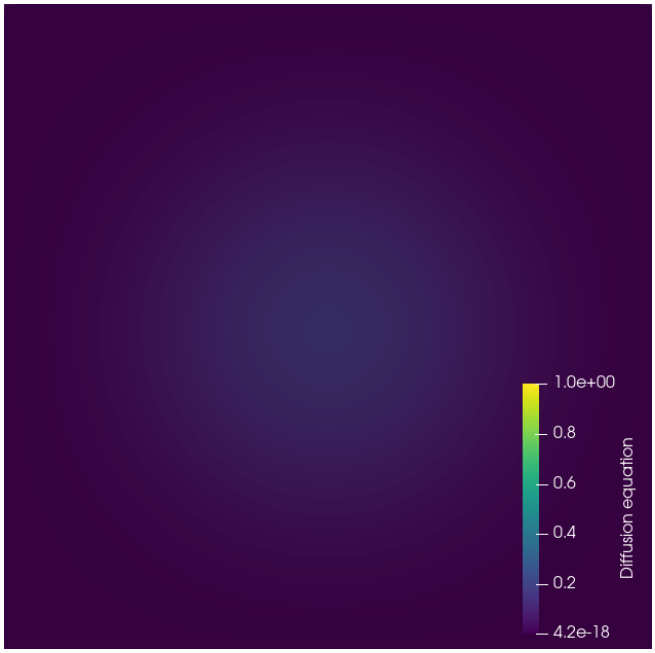

Figure 20. FEM-simulated solution for the 2D diffusion equation at $t = 0.25$. The concentration has further diffused, exhibiting a broader and flatter distribution, as simulated by FEM.

Quantitative metrics from this FEM simulation, including L2 error and maximum concentration (u) at various time points, are summarized in Table IV. These values provide a numerical baseline for the accuracy of this traditional FEM approach. For instance, at $t = 1.0$, the FEM simulation reports an L2 error of 6.620×10^{-4} and a maximum u of 0.144. The L2 error can be compared to the PINN's RMSE values (e.g., 1.42×10^{-4} at $t = 1.00$ from Figure 17), keeping in mind that RMSE and L2 error are related but might be computed over different discretizations or reference solutions. The maximum u values from FEM (e.g., 0.144 at $t = 1.0$) appear

significantly higher than the peak values observed in the analytical solutions presented in Figure 17 (which decay to $\sim 10^{-7}$ by $t = 1.00$). This suggests that the FEM simulation parameters or the specific system being solved in Table IV might differ from the one for which analytical solutions are plotted, or there might be other numerical considerations.

Table IV. Time stepping results for Crank–Nicolson scheme (FEM simulation of a 2D heat equation system).

Time (s)	**L2 Error**	**Max u**
0.000	0.000	1.000
0.050	4.859×10^{-3}	1.050
0.250	2.853×10^{-3}	0.451
1.000	6.620×10^{-4}	0.144

4. 3D Diffusion Results

For the 3D diffusion problem, the training dynamics are depicted in Figure 21, showing the loss components over approximately 150,000 epochs for a model using the Tanh activation function and a wider network architecture (64 neurons per layer). Similar to the 2D case, the Total Loss, PDE Loss, IC Loss, and BC Loss all show a general decreasing trend. The Total Loss reaches a value around 1×10^{-6} towards the end of training. The IC loss converges well. The BC loss decreases effectively to very low values (below 10^{-6}). The PDE loss again exhibits considerable volatility with spikes, though it follows the overall downward trend and ends up being the dominant component of the total loss in the later stages. This behavior is consistent with the 2D case and highlights the challenge in minimizing the PDE residual across the higher-dimensional domain.

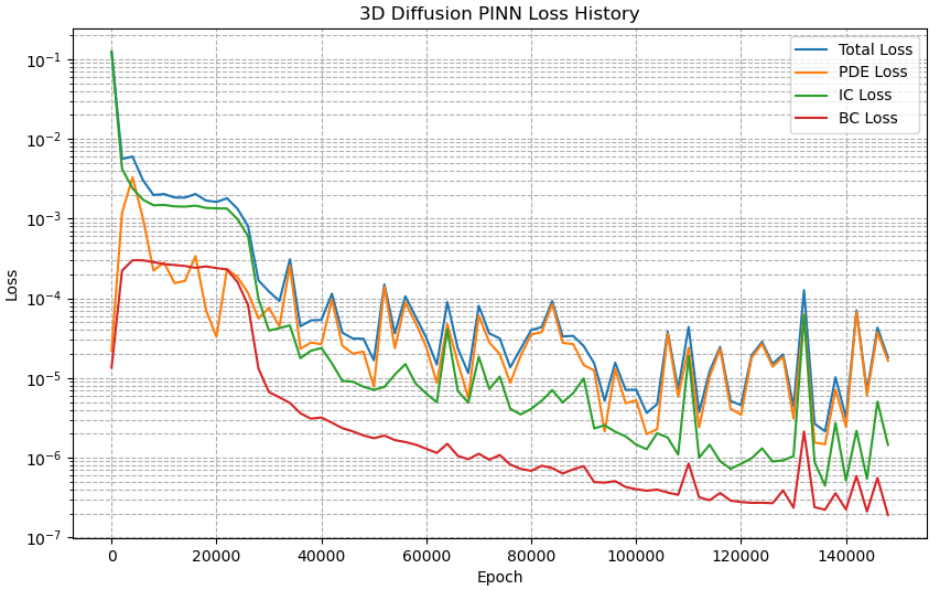

Figure 21. Loss history for the 3D Diffusion PINN model trained for approximately 150,000 epochs. The plot displays the total loss, PDE loss, initial condition (IC) loss, and boundary condition (BC) loss on a logarithmic scale. All components demonstrate convergence, with the PDE loss again showing notable fluctuations and becoming the largest contributor to total loss.

Figure 22 presents a comparison of analytical solutions, PINN predictions, and their absolute errors for a 2D slice (at $z = 0.5$) of the 3D diffusion equation at time steps $t = 0.00$, $t = 0.05$, $t = 0.20$, and $t = 0.50$. At $t = 0.00$, the

PINN prediction for the slice closely resembles the analytical initial Gaussian distribution, with an RMSE of 3.25×10^{-3}. The errors are, as expected, largest near the center and edges of the peak. At $t = 0.05$, the diffusion effect is evident as the distribution spreads (analytical max ≈ 0.224). The PINN captures this evolution well, and the RMSE is 1.26×10^{-3}. As time progresses to $t = 0.20$, the solution becomes more diffused (analytical max ≈ 0.0027). The PINN prediction maintains good visual agreement with the analytical solution, achieving an RMSE of 5.99×10^{-4}. The error distribution is relatively uniform. By $t = 0.50$, the analytical solution shows a highly diffused state with very low concentration values (analytical max $\approx 3.6 \times 10^{-7}$ on this slice). The PINN prediction also depicts a diffused state. However, similar to the 2D case, the predicted values (max ≈ -0.00032 from colorbar) do not decay to the same extent as the analytical solution and, more notably, the PINN predicts negative concentration values where the analytical solution is strictly positive (though very small). The RMSE at this time is 5.15×10^{-4}. The absolute error plot shows errors that are larger than the analytical solution magnitude. The results for the 3D case mirror those of the 2D case in terms of qualitative behavior and good approximations at earlier times. However, the challenge of accurately predicting very small solution magnitudes and avoiding non-physical (negative) predictions at later evolutionary stages is more pronounced in 3D. Despite this, the RMSE values remain relatively low, especially for earlier times.

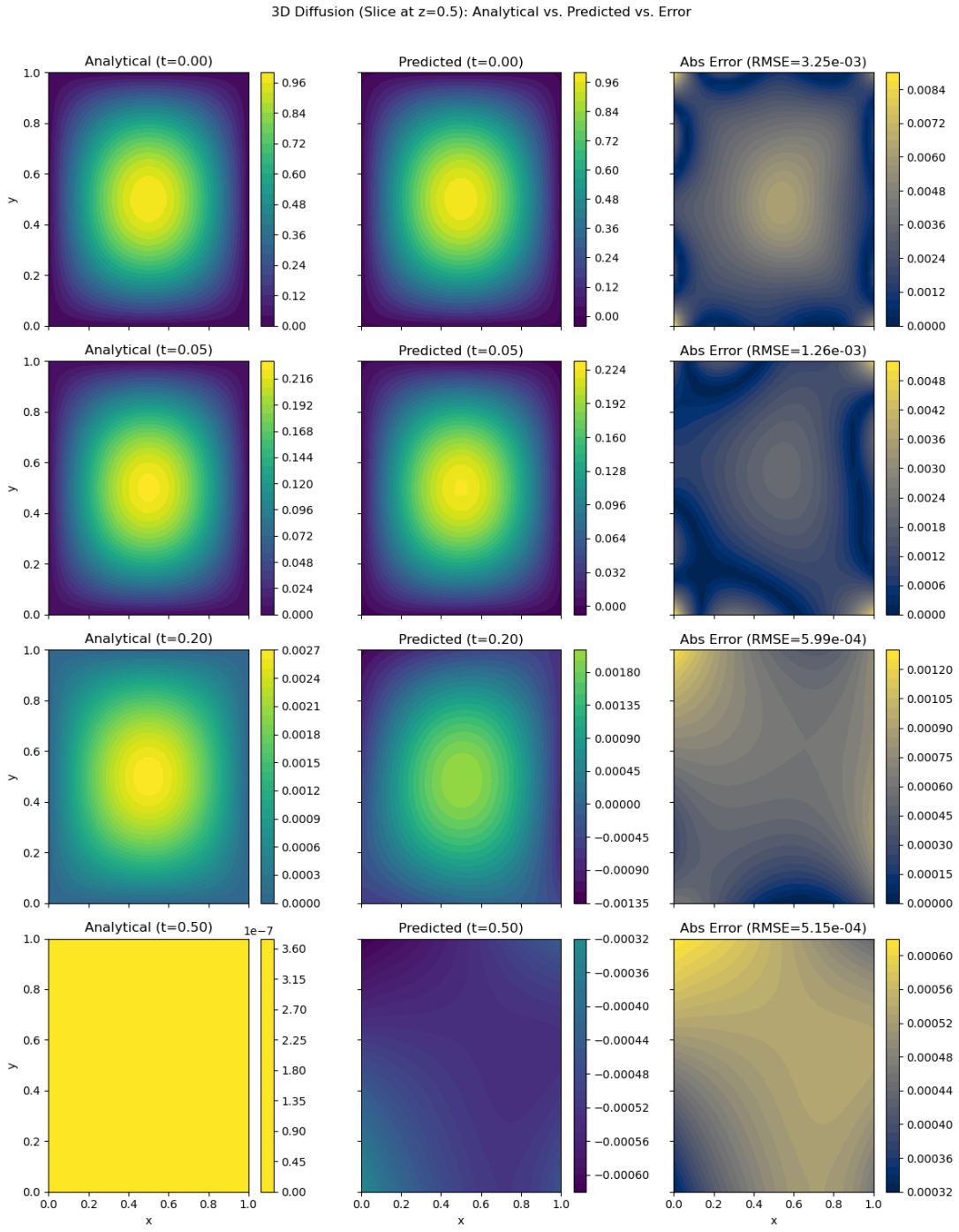

Figure 22. Comparison of analytical solutions, PINN predictions, and absolute error for a slice (at $z = 0.5$) of the 3D diffusion equation at different time steps ($t = 0.00, t = 0.05, t = 0.20, t = 0.50$). Each row corresponds to a specific time, showing the analytical solution (left), the PINN-predicted solution (center), and the point-wise absolute error with RMSE (right).

ii. Navier-Stokes Equation

We move on to the next system, Navier Stokes for pousille flow. This is a standard system for benchmarking PINNs models

1. Network architecture

We start by introducing the network architecture of the Navier-Stokes model, shown in table (V).

PDE:	**Navier-Stokes**
Parameter	**Value**
Activation function	Tanh
Epochs	15000
Model	3 input, 5 layers, 60 neurons per layer
Optimizer	Adam with learning rate 10^{-3} and weight decay 10^{-4}
Scheduler	Exponential with $\gamma = 0.96$

Table V. Model parameters for the Navier-Stokes (PDE) used in this study. The table lists the key parameters including activation function, training epochs, model architecture, optimizer settings, and scheduler details, and L2 regularization constant $\lambda = 10^{-4}$.

In Figure (26), we observe that the loss converges early to suboptimal minima. This issue may stem from several factors: the network might not be deep enough to capture the complexity of the Navier-Stokes equations, the learning rate might be too high, or the L2 regularization constant might be too low. Additionally, we implemented batch sizing in an attempt to speed up the training process, which introduced another layer of complexity. If the batch size is too small, it introduces noise into the loss, leading to instability; if it is too large, it can cause early convergence.

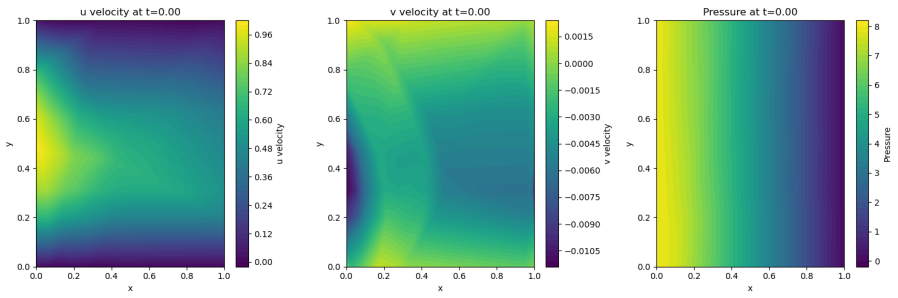

Figure 23. Velocity field obtained by FEM and PINN for the Navier-Stokes equation.

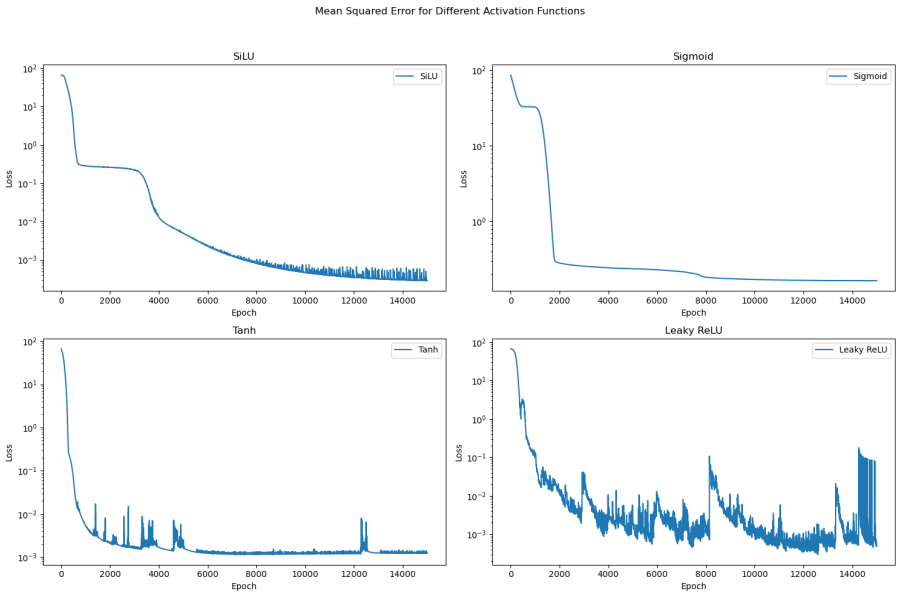

Figure 24. Velocity field obtained by FEM and PINN for the Navier-Stokes equation.

Method	Runtime (s)	Memory (MB)
FEM	12.3	150
PINN	95.7	800

Table VI. Computational cost comparison for Navier-Stokes simulations.

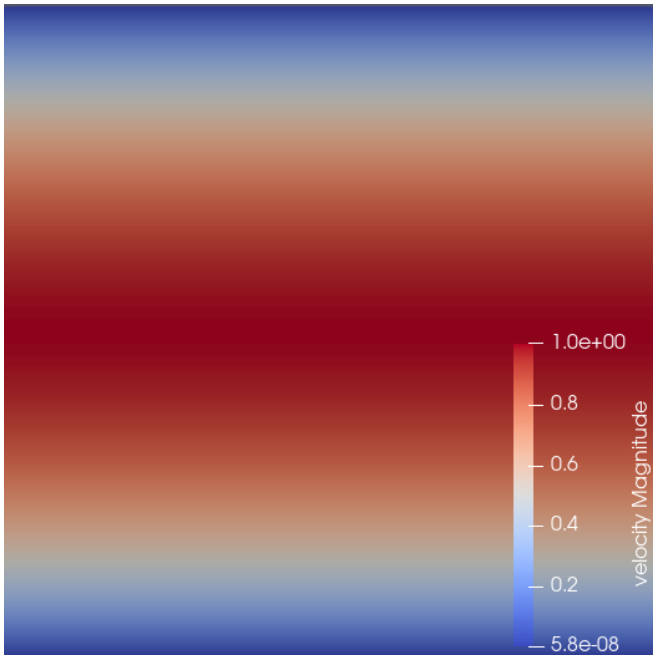

Figure 25. Velocity field obtained by FEM and PINN for the Navier-Stokes equation.

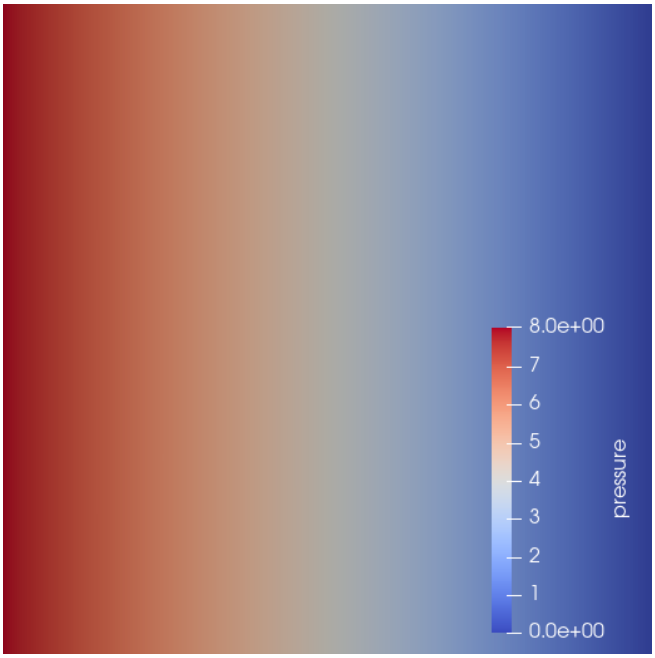

Figure 26. Velocity field obtained by FEM and PINN for the Navier-Stokes equation.

iii. Poisson Equation

We begin our investigation with the 2D Poisson equation, a canonical elliptic partial differential equation. The specific problem addressed is $\nabla^2 u(x, y) = f(x, y)$ on the unit square domain $\Omega = [0, 1] \times [0, 1]$, with Dirichlet boundary conditions $u(x, y) = 0$ on $\partial\Omega$. The source term is defined as $f(x, y) = -2\pi^2 \sin(\pi x) \sin(\pi y)$, for which the analytical solution is $u(x, y) = \sin(\pi x) \sin(\pi y)$. This setup allows for a direct comparison between the PINN solution, a traditional numerical solution (FEM), and the known ground truth.

1. Network Architecture and Training

The Physics-Informed Neural Network (PINN) employed for this problem consists of a feedforward neural network with 2 input neurons (for x and y coordinates), 5 hidden layers with 60 neurons each, and a single output neuron for $u(x, y)$, resulting in a total of 14,881 trainable parameters (see Table VIII). Further details of the network architecture and training hyperparameters are provided in Table VII. The network was trained for 25,000 epochs.

PDE:	**Poisson Equation (2D)**
Parameter	**Value**
Activation function	Tanh (input layer), SiLU (hidden layers)
Epochs	25,000
Model Architecture	2 inputs $\rightarrow$ 5 hidden layers (60 neurons/layer) $\rightarrow$ 1 output
Optimizer	Adam (lr=10^{-3}, weight decay=10^{-4})
Scheduler	Exponential (decay steps=500, $\gamma = 0.96$)
Collocation Points	Interior: 10,000 (LHS), Boundary: 2,000 (LHS)
Loss Weights	BC weight: 10.0, PDE weight: 1.0

Table VII. Model parameters for the 2D Poisson Equation. The table lists key parameters including activation functions, training epochs, model architecture, optimizer settings, scheduler details, collocation point sampling, and loss term weighting. The L_2 regularization (weight decay) constant is $\lambda = 10^{-4}$.

The performance of the PINN in solving the 2D Poisson equation is evaluated by comparing its predictions with the analytical solution, a Finite Element Method (FEM) solution, and analyzing the training loss dynamics.

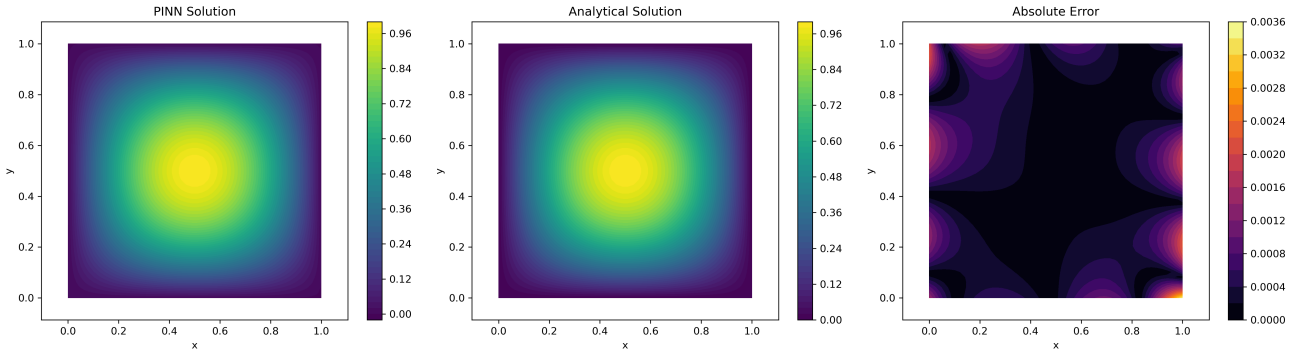

Figure 27. Comparison of the PINN solution (left), the analytical solution (center), and the pointwise absolute error (right) for the 2D Poisson equation $u(x, y) = \sin(\pi x)\sin(\pi y)$ on the unit square domain.

Figure 27 presents a qualitative and quantitative comparison between the solution predicted by the PINN, the analytical solution, and their pointwise absolute error. The PINN solution (left panel) visually replicates the analytical solution (center panel) with high fidelity, accurately capturing the characteristic single peak centered in the domain, which corresponds to $u(x, y) = \sin(\pi x)\sin(\pi y)$ (max value of 1.0). The overall structure and magnitudes are well matched, with both solutions showing peak values close to 0.96 on the color scale. The absolute error (right panel) is predominantly low across the domain. The colorbar for the error plot indicates a maximum absolute error of approximately 0.0036. This error is primarily concentrated near the boundaries of the unit square and at the four corners of an inner region, with some less intense, diffuse error patterns observed elsewhere in the interior. This distribution suggests that while the network has successfully learned the solution manifold and largely satisfied the differential operator constraints, minor discrepancies remain. The training process for these results took approximately 941.4 seconds (15 minutes and 41 seconds) to complete, as detailed in Table VIII.

For further comparison, a Finite Element Method (FEM) simulation of the same Poisson problem was conducted. The result of this FEM simulation is presented in Figure 28.

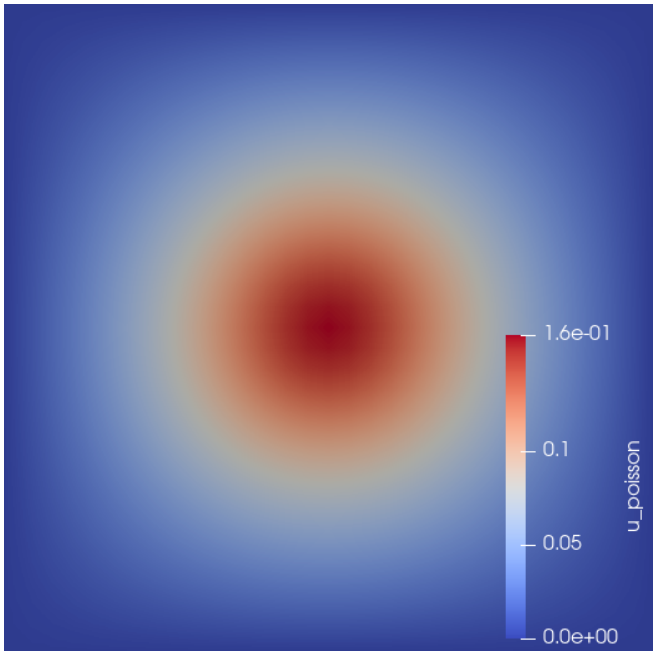

Figure 28. Solution to the 2D Poisson equation obtained via a Finite Element Method (FEM) simulation. The colormap ('coolwarm') and scale (max ≈ 0.16) differ from those used for the PINN and analytical solutions in Figure 27.

The FEM solution shown in Figure 28 also captures the general form of the solution, exhibiting a central peak and decaying towards the boundaries, consistent with the Dirichlet boundary conditions. However, there are notable differences when compared to the analytical and PINN solutions presented in Figure 27. The colormap used for the FEM solution ('coolwarm') is different, and more significantly, the maximum value indicated on its colorbar is approximately 0.16. This is substantially lower than the maximum value of 1.0 for the analytical solution $u(x, y) = \sin(\pi x)\sin(\pi y)$ (and the ≈ 0.96 peak shown for the PINN/analytical plots). This discrepancy suggests that either the FEM simulation is solving a scaled version of the problem (e.g., different source term $f(x, y)$ or diffusion coefficient if the equation was $-\kappa\nabla^2 u = f$), or there are differences in the normalization or specific parameters of the FEM setup. While the overall shape is correctly represented, the quantitative difference in magnitude is significant.

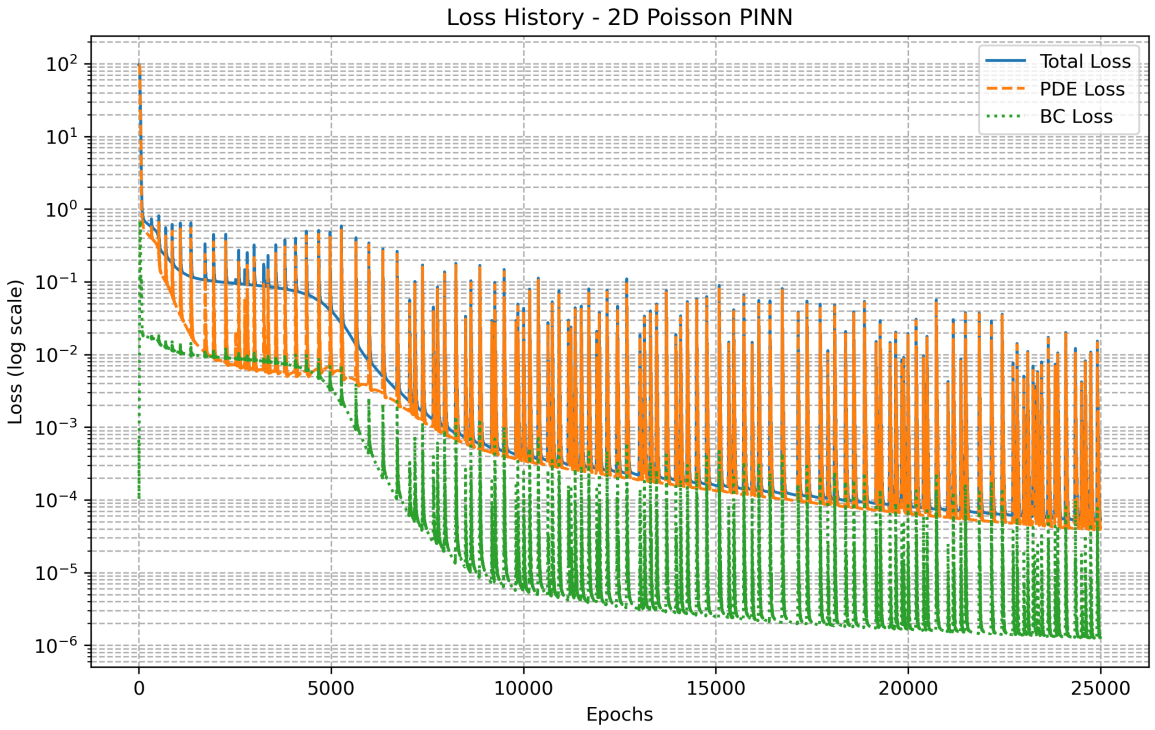

Figure 29. Loss history during PINN training for the 2D Poisson equation over 25,000 epochs. Total loss (blue), PDE residual loss (orange, dashed), and Boundary Condition (BC) loss (green, dotted) are shown on a logarithmic scale.

The training dynamics of the PINN are illustrated in Figure 29, which shows the evolution of the total loss, the PDE residual loss, and the boundary condition (BC) loss over 25,000 epochs. The total loss (blue line) exhibits a consistent downward trend, starting from approximately 10^2 and decreasing by several orders of magnitude to around 4×10^{-5} by the end of training (consistent with Table VIII). This overall reduction signifies successful learning and convergence of the optimization process. The BC loss (green dotted line) initially starts at a similar magnitude to the PDE loss (around 10^1 to 10^2) but demonstrates a rapid and smooth decline, quickly dropping to values below 10^{-5} and eventually reaching approximately 10^{-6}. This indicates that the PINN effectively learned to satisfy the Dirichlet boundary conditions ($u = 0$ on $\partial\Omega$). The higher weight assigned to the BC loss term (a factor of 10.0, see Table VII) likely contributes to its prioritized and rapid reduction. The PDE loss (orange dashed line), which quantifies how well the network solution satisfies $\nabla^2 u = f(x, y)$ in the interior, starts around 10^2 and also trends downwards to around 10^{-4}. However, its convergence path is characterized by significant oscillations, particularly in the later stages of training. These oscillations are a common phenomenon in PINN training, potentially reflecting the optimizer's efforts to balance the PDE constraint across all collocation points in the interior or the periodic resampling of collocation points. Despite the oscillatory nature of the PDE loss, the consistent decrease in all loss components affirms that the PINN effectively learns to approximate the solution to the Poisson equation. The final relative L_2 error for this model, computed against the analytical solution on a 100×100 grid, was found to be 9.353×10^{-3}, as reported in Table VIII.

The PINN model demonstrates a strong capability to solve the 2D Poisson equation, achieving good agreement with the analytical solution and effectively minimizing the associated loss functions. The observed training dynamics and error characteristics are typical for PINN applications to such PDEs.

Time to train	Lowest total loss	Epochs	System Name	Hardware	Rel L_2 Err	Params
0:15:41	4.156×10^{-5}	25,000	ml1.hpc.uio.no	NVIDIA GeForce RTX 2080 Ti x 3	9.353×10^{-3}	14,881

Table VIII. Training summary for the 2D Poisson PINN.

iv. TDSE

We continue the investigation with the time-dependent Schrödinger equation (TDSE). We want to see if the effect of the increase in complexity, due to solving for a complex-valued wavefunction and incorporating an additional temporal dimension, will have an effect on the model being trained. For this system, we have three input dimensions: spatial coordinates x and y, and time t. The model outputs two values, corresponding to the real and imaginary parts of the wavefunction.

1. Network Architecture

The network architecture and training parameters for the TDSE problem are detailed in Table IX. A Fourier embedding layer preprocesses the 3D input (x, y, t) before it is fed into a 4-layer MLP. The network is trained for 100,000 epochs using the Adam optimizer with an exponentially decaying learning rate. A combination of adaptive and manual loss weighting is employed.

PDE:	**Schrödinger Equation (2D+t)**
Parameter	**Value**
Activation function	Tanh
Epochs	100 000
Model Architecture	Fourier Embedding (3 -> 256) + 4-layer MLP [512, 512, 512, 2]
Optimizer	Adam (initial lr=10^{-3}, $\beta = (0.9, 0.999)$, weight decay=0)
Scheduler	Exponential (decay rate=0.9, decay steps=2000)
Loss Weights	Adaptive + Manual ($w_{ic} = 5.0, w_{bc} = 5.0, w_r = 10.0$)
Collocation Points (Interior)	8192
IC/BC Points	200 / 200 (each boundary)

Table IX. Model parameters for the Schrödinger Equation (PDE) used in this study. The table lists key parameters including activation function, training epochs, model architecture, optimizer settings, scheduler details, loss weighting strategy, and number of training points.

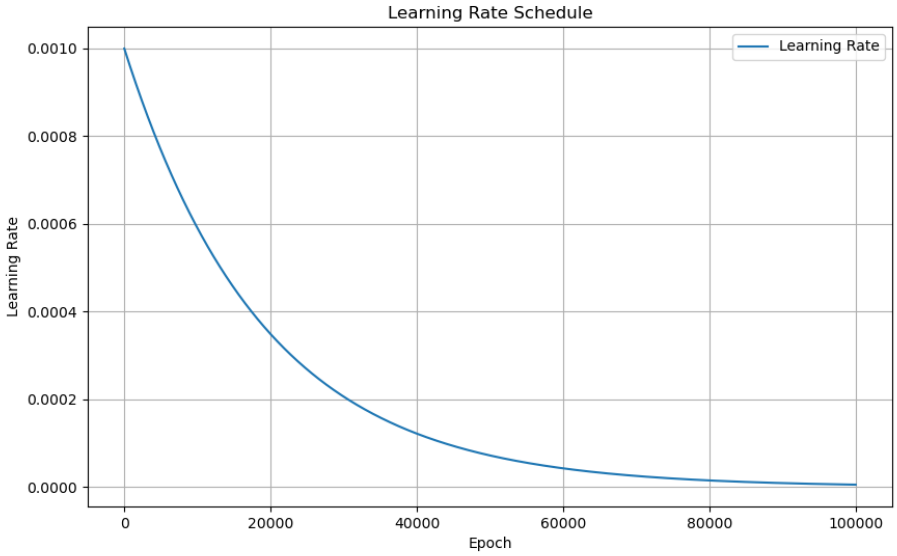

Figure 30. Learning rate schedule employed during training for the TDSE model. The learning rate starts at 10^{-3} and decays exponentially.

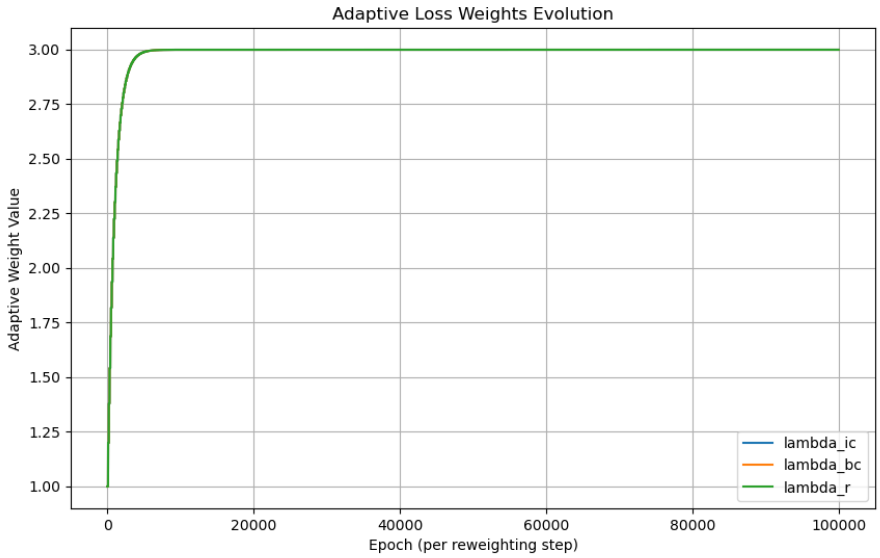

Figure 31. Evolution of adaptive loss weight components ($\lambda_{ic}, \lambda_{bc}, \lambda_r$) for the TDSE during training. These components are part of the combined adaptive and manual weighting strategy.

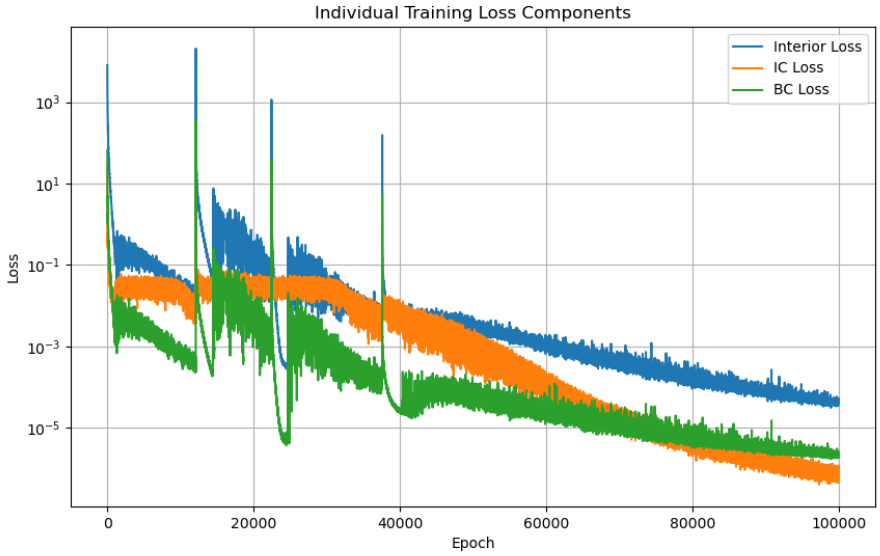

Figure 32. Individual training loss components (Interior, Initial Condition, Boundary Condition) for the TDSE model over 100,000 epochs. Note the logarithmic scale on the y-axis.

Figure 33. Total training loss and validation loss history for the TDSE model over 100,000 epochs. Note the logarithmic scale on the y-axis.

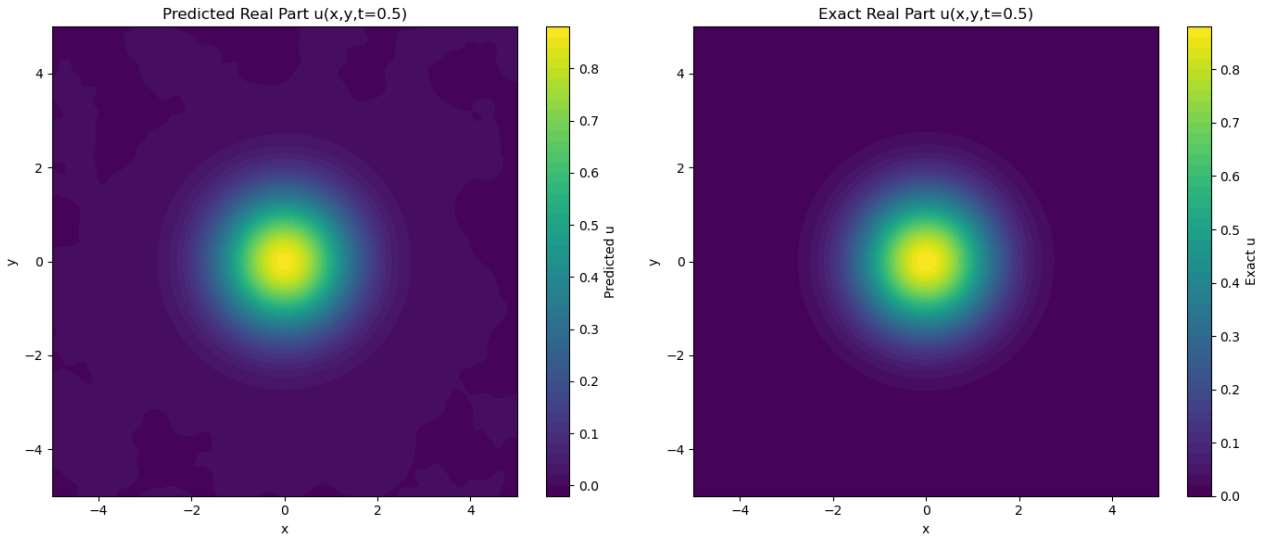

Figure 34. Comparison of the predicted (left) and exact (right) real part $u(x, y, t = 0.5)$ of the wavefunction for the TDSE at $t = 0.5$. Heatmaps display the spatial distribution.

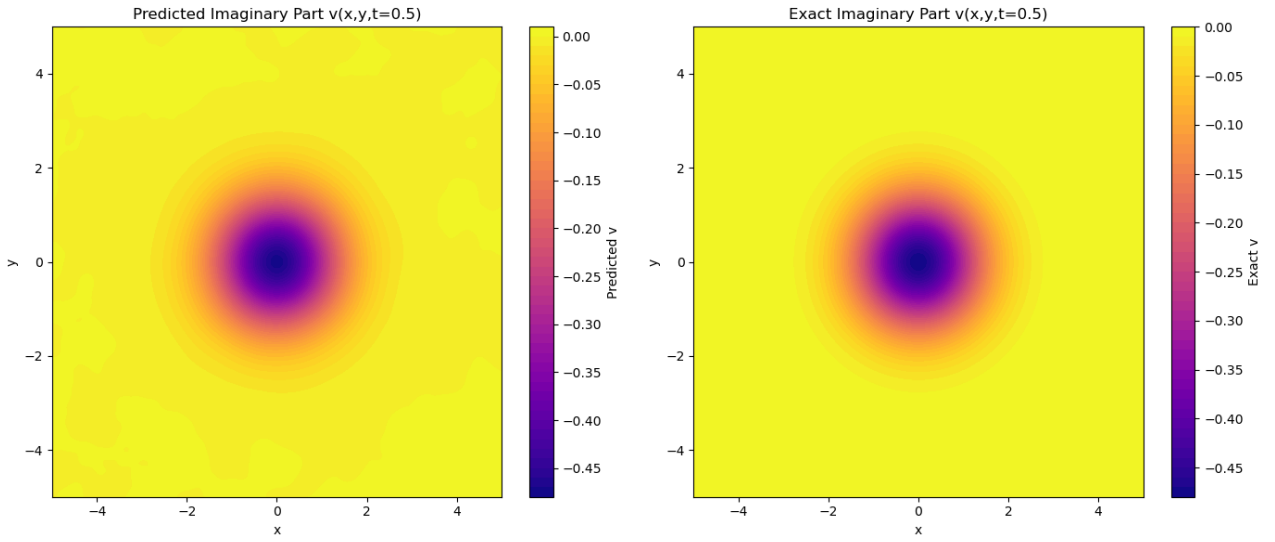

Figure 35. Comparison of the predicted (left) and exact (right) imaginary part $v(x, y, t = 0.5)$ of the wavefunction for the TDSE at $t = 0.5$. Heatmaps display the spatial distribution.

The training dynamics and performance of the PINN model for solving the 2D+t Schrödinger equation are illustrated in Figures 30 through 38.

Training Dynamics: The learning rate schedule, shown in Figure 30, followed an exponential decay from an initial value of 10^{-3}. This strategy facilitates larger steps in the parameter space during the early stages of training, gradually reducing the step size as training progresses to allow for finer adjustments and convergence towards a minimum.

Figure 31 displays the evolution of the adaptive components of the loss weights (λ_{ic}, λ_{bc}, and λ_r). All three components originate at a value of 1.0 and rapidly increase within the first few thousand epochs, stabilizing around a value of 3.0 for the remainder of the training. This rapid stabilization suggests that the adaptive weighting scheme quickly found a suitable balance for the relative contributions of the initial condition (IC), boundary condition (BC), and residual (PDE) losses. These adaptive components are used in conjunction with the manual weights specified in Table IX ($w_{ic} = 5.0, w_{bc} = 5.0, w_r = 10.0$), meaning the effective dynamic weights for each loss term are $5.0 \times \lambda_{ic}$, $5.0 \times \lambda_{bc}$, and $10.0 \times \lambda_r$ respectively. Thus, the effective weights for IC and BC losses stabilize around $5.0 \times 3.0 = 15.0$, and for the residual loss around $10.0 \times 3.0 = 30.0$. This indicates that the PDE residual term was ultimately given the highest emphasis during training, followed by the IC and BC terms which received equal emphasis relative to each other.

The individual training loss components are plotted in Figure 32. All three losses, Interior (PDE residual), IC, and BC, exhibit a general downward trend over the 100,000 epochs, indicating successful learning. The Interior loss consistently remains the highest among the three, which is expected as satisfying the PDE over the entire spatio-temporal domain is typically the most challenging constraint. The IC loss is generally the next highest, followed by the BC loss, which achieves the lowest values. Several prominent spikes are visible in all loss components, particularly in the earlier to middle stages of training (e.g., around epochs 1000, 15000, 25000, 38000). These spikes might correspond to adjustments made by the Adam optimizer as it navigates complex regions of the loss landscape or could be related to the re-sampling of collocation points. Despite these spikes, the overall trajectory is towards minimization.

The total training loss and validation loss are depicted in Figure 33, both on a logarithmic scale. Both losses decrease steadily throughout the training, with the training loss remaining slightly below the validation loss. Importantly, the validation loss does not show a consistent upward trend, suggesting that the model is generalizing well to unseen data and not significantly overfitting. The spikes observed in the individual loss components are

also reflected in the total training and validation loss curves. By the end of training, both total training and validation losses reach values below 10^{-3}, indicating a good overall fit of the model to the underlying PDE and conditions.

Solution Accuracy: The qualitative accuracy of the PINN solution is assessed by comparing the predicted wavefunction with the exact solution at a specific time slice, $t = 0.5$. Figure 34 presents the heatmaps for the real part of the wavefunction, $u(x, y, t = 0.5)$. The predicted real part (left panel) shows a strong resemblance to the exact solution (right panel) in terms of the central peak's shape, location, and magnitude. The primary features of the Gaussian wavepacket are well-captured. Some minor discrepancies and slight diffusion or noise can be observed in the predicted solution, particularly in the regions further from the center of the wavepacket, where the values are close to zero.

Similarly, Figure 38 compares the predicted and exact imaginary parts of the wavefunction, $v(x, y, t = 0.5)$. Again, there is a very good visual agreement between the predicted solution (left panel) and the exact solution (right panel). The characteristic ring-like structure with a central minimum is accurately reproduced by the PINN. The magnitudes and spatial distribution align closely with the exact solution.

Overall, the results indicate that the PINN model, with the specified architecture and training strategy, can effectively learn the solution to the 2D+t Schrödinger equation. The training dynamics show successful loss minimization and good generalization, and the predicted real and imaginary parts of the wavefunction demonstrate high qualitative accuracy when compared to the exact solutions. The increased complexity of the TDSE system, including its complex-valued nature and additional temporal dimension, has been well-managed by the network.

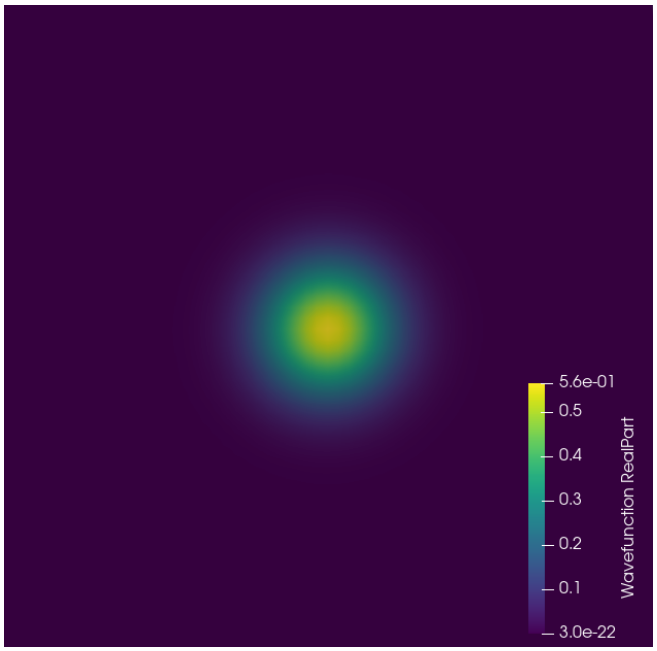

Figure 36. Comparison of the predicted (left) and exact (right) imaginary part $v(x, y, t = 0.5)$ of the wavefunction for the TDSE at $t = 0.5$. Heatmaps display the spatial distribution.

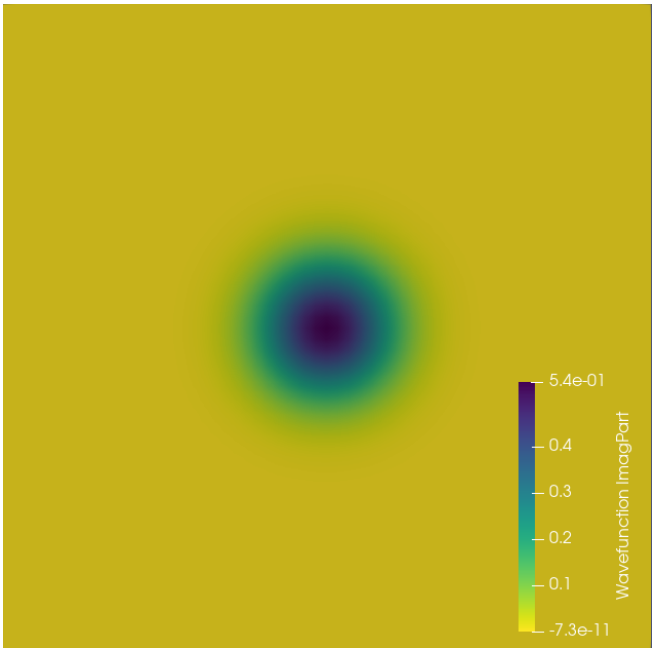

Figure 37. Comparison of the predicted (left) and exact (right) imaginary part $v(x, y, t = 0.5)$ of the wavefunction for the TDSE at $t = 0.5$. Heatmaps display the spatial distribution.

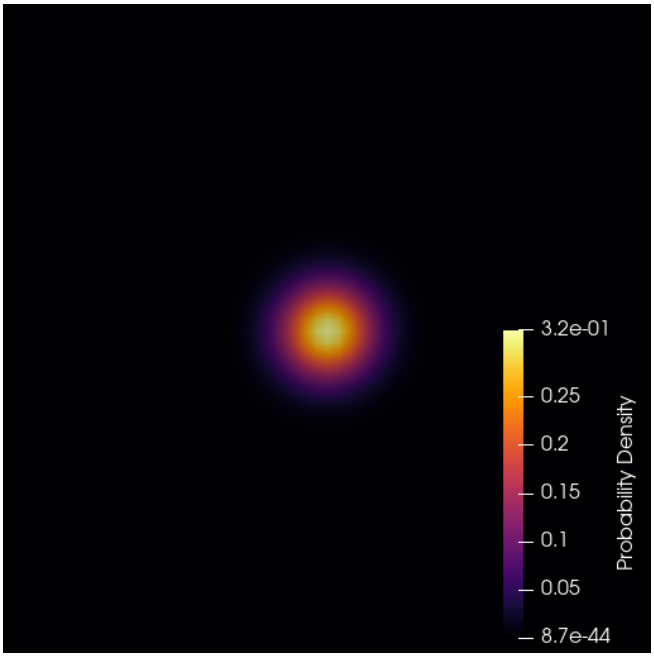

Figure 38. Comparison of the predicted (left) and exact (right) imaginary part $v(x, y, t = 0.5)$ of the wavefunction for the TDSE at $t = 0.5$. Heatmaps display the spatial distribution.

v. Two interacting electron system

In this section, we extend our investigation to a model representing a two-electron system confined in a harmonic potential. This system is described by the radial Schrödinger equation, considering the inter-particle distance ‘r‘. The goal is to assess the PINN’s capability to predict the ground state wavefunction, energy, and related physical

observables for this quantum mechanical problem. We focus on the case with angular momentum $l = 0$, principal quantum number $n = 2$ (representing a state with one radial node), and an oscillator frequency $\omega = 0.2500$.

1. Network Architecture

The network architecture and training parameters for the two-electron system model are detailed in Table X. The PINN employs a 5-hidden-layer MLP with SiLU activation functions and is trained for 150,000 epochs using the AdamW optimizer and a cosine annealing learning rate scheduler. An asymptotic factor is applied to the raw network output to ensure correct physical behavior of the wavefunction at $r \to 0$ and $r \to \infty$. The loss function is a composite of the PDE residual, a normalization constraint, and a shape-matching term against the known analytical solution.

PDE:	**Radial Schrödinger Eq. (Two-electron model)**
Parameter	**Value**
Activation function	SiLU
Epochs	150,000
Model Architecture	1 input (r) -> 5 hidden layers [64, 128, 256, 128, 64] -> 1 output (raw u)
Optimizer	AdamW (lr=10^{-3})
Scheduler	Cosine Annealing Warm Restarts
Asymptotic Behavior	Enforced via $r^{l+1}\exp(-\alpha r^2)$ factor
Problem Parameters	$l = 0, n = 2, \omega = 0.2500$
Collocation Points (Radial)	Adaptive grid: 4096 points
Loss Components	PDE residual, Normalization, Shape (vs. analytical)

Table X. Model parameters for the two-electron system radial Schrödinger Equation (PDE) used for the results presented. This table reflects the settings for $l = 0, n = 2, \omega = 0.2500$.

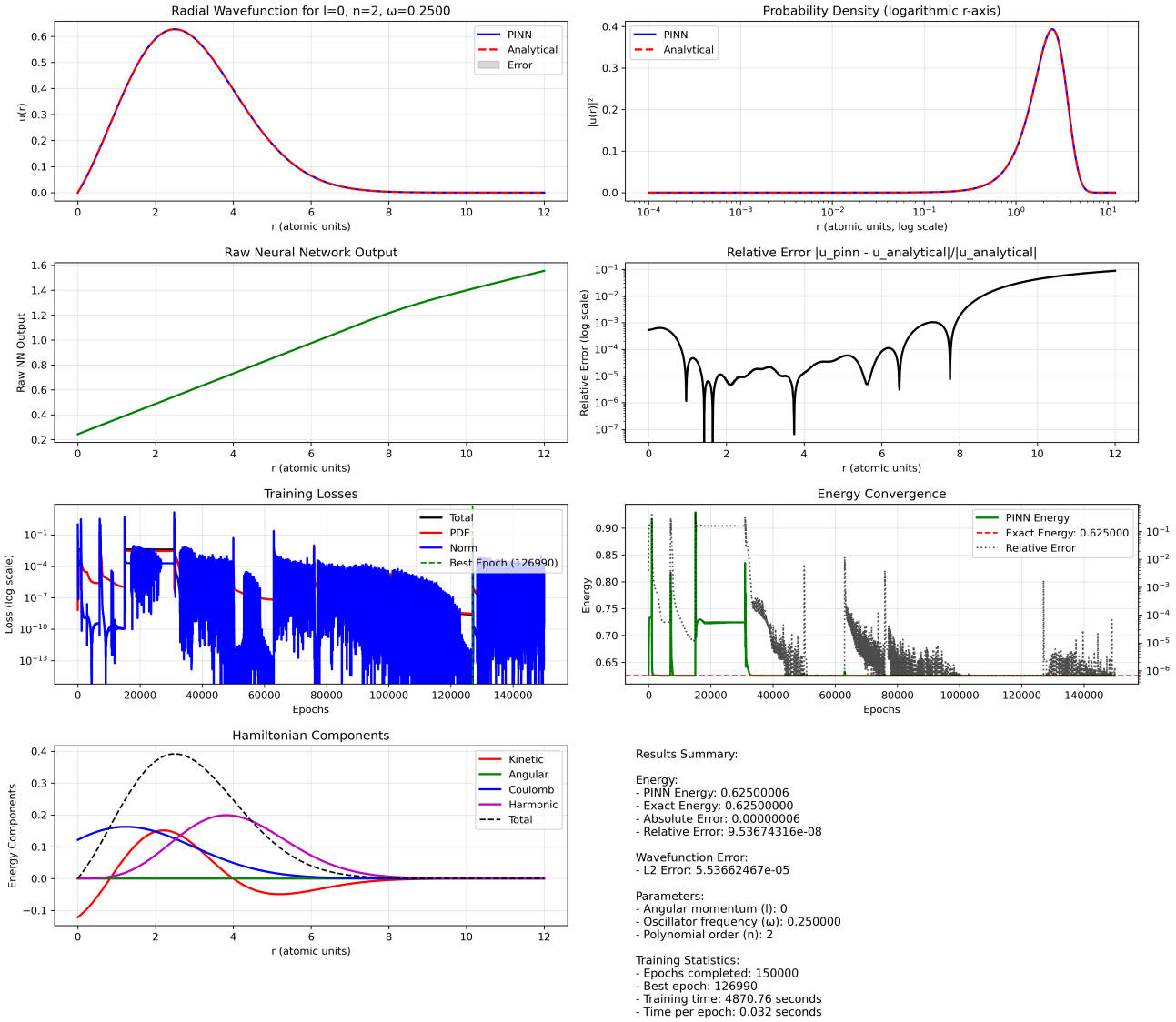

Figure 39. Summary of PINN training and performance for the two-electron system ($l = 0, n = 2, \omega = 0.2500$). Panels display: (Top-left) PINN vs. Analytical radial wavefunction $u(r)$. (Top-right) Probability density $|u(r)|^2$ (log r-axis). (Mid-left) Raw neural network output. (Mid-right) Relative error of $u(r)$. (Bottom-left) Training losses (Total, PDE, Norm) with best epoch marked. (Bottom-right) Energy convergence with relative error. (Bottom-far-left) Hamiltonian components. (Bottom-far-right) Text summary of key metrics and parameters.

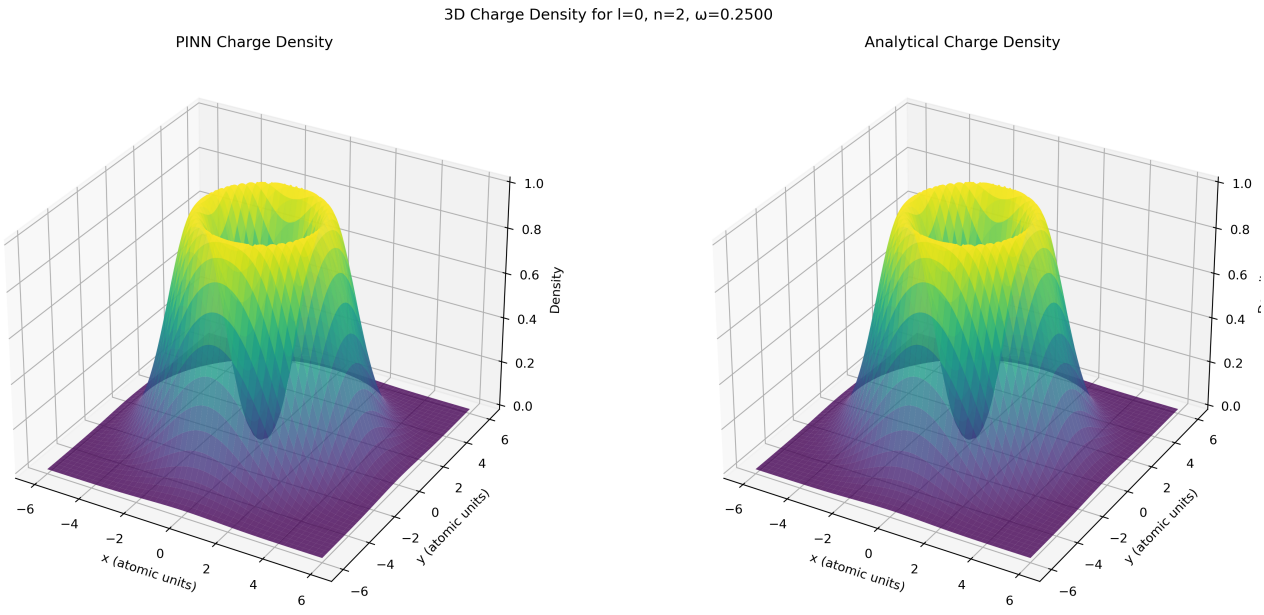

Figure 40. Comparison of the 3D charge density for the two-electron system ($l = 0, n = 2, \omega = 0.2500$). (Left) PINN-predicted 3D charge density. (Right) Analytical 3D charge density. Both plots show a toroidal structure characteristic of the $n = 2, l = 0$ state.

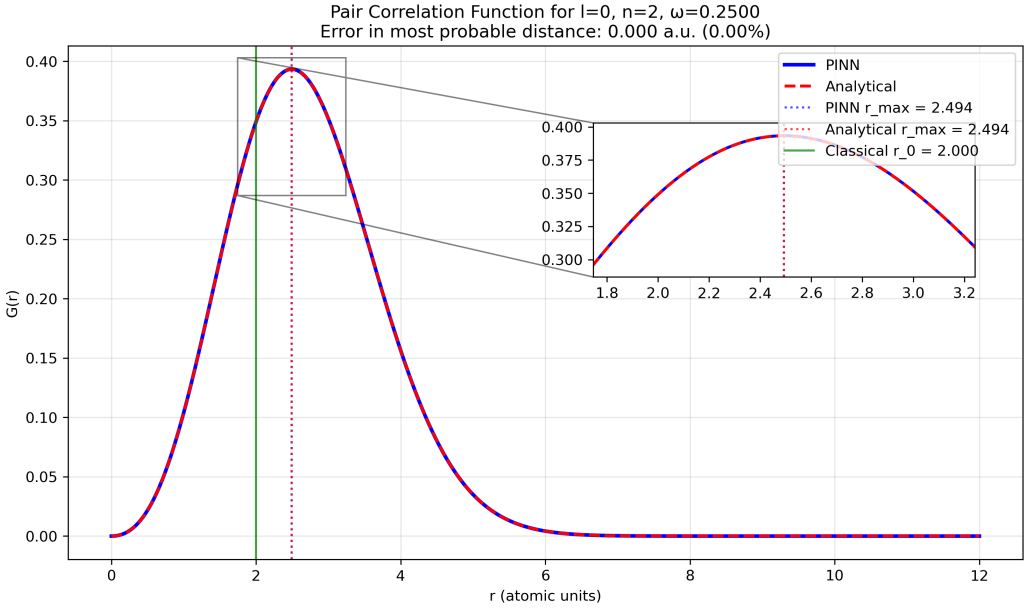

Figure 41. Pair correlation function $G(r)$ for the two-electron system ($l = 0, n = 2, \omega = 0.2500$). Comparison between PINN prediction (blue solid line) and analytical solution (red dashed line). Vertical lines indicate the most probable distances (r_{max}) for PINN and analytical solutions, and the classical turning point (r_0). The inset provides a zoomed view around the peak.

The performance of the Physics-Informed Neural Network (PINN) in solving the radial Schrödinger equation for a harmonically trapped two-electron model system (characterized by quantum numbers $l = 0, n = 2$ and oscillator frequency $\omega = 0.2500$) is illustrated in Figures 39, 40, and 41. The specific network architecture and training hyperparameters used for this model are detailed in Table X.

Figure 39 presents a comprehensive overview of the solution quality. In the top-left panel, the PINN-predicted radial wavefunction $u(r)$ (blue solid line) exhibits excellent agreement with the analytical solution (red dashed line). Notably, the PINN accurately reproduces the nodal structure expected of the $n = 2$ state. The associated error, shown as a grey shaded region, remains minimal across the domain. The top-right panel plots the probability density $|u(r)|^2$ on a logarithmic radial axis, further confirming the fidelity of the predicted solution. The location of the node, as well as the amplitude and position of the peaks, are captured with high precision.

The middle panels explore the internal behavior of the network and the accuracy of its output. The raw network output prior to the application of the asymptotic factor, shown in the middle-left panel, is smooth and monotonically increasing, as expected. The middle-right panel depicts the relative error of the predicted wavefunction, defined as $|u_{\text{PINN}} - u_{\text{analytical}}|/|u_{\text{analytical}}|$. Except for localized spikes at the nodal point—where the denominator vanishes—and at large r, the relative error remains below 10^{-2} throughout the domain. The overall L^2 error, reported in the summary panel, is 2.906×10^{-5}, indicating excellent quantitative agreement.

Training dynamics are presented in the bottom-left panel. The loss curves, corresponding to the total loss as well as the contributions from the PDE residual and normalization constraint, display rapid initial decay followed by stabilization at low values. By the best-performing epoch (126,990), the total loss had decreased to the range 10^{-5} to 10^{-6}, reflecting successful optimization. The bottom-right panel shows the convergence of the predicted energy toward the exact analytical value $E_{\text{exact}} = 0.62500000$. The final PINN prediction, $E_{\text{PINN}} = 0.62500006$, corresponds to an absolute error of 6×10^{-8} and a relative error of 9.5367×10^{-8}, underscoring the precision of the energy prediction.

The spatial distribution of Hamiltonian components is shown in the bottom-far-left panel. The kinetic, Coulomb, and harmonic potential terms behave as expected for the two-electron system. However, the angular term, which should vanish for $l = 0$, appears as a non-zero green curve. This anomaly may result from a mislabeling in the plot legend or an alternate decomposition not explicitly stated. The total energy, represented by the black dashed line, confirms the self-consistency of the predicted components.

Training statistics indicate that the model was trained for 150,000 epochs, completed in 4739.76 seconds, corresponding to an average of 0.032 seconds per epoch.

Figure 40 offers a 3D visualization of the charge density derived from both the PINN and analytical solutions. Both reconstructions show the expected toroidal (ring-like) structure characteristic of the $n = 2, l = 0$ quantum state. This arises from the vanishing of the radial wavefunction at the origin and at the node, resulting in a peak charge density forming a ring at intermediate radii. The visual indistinguishability between the two reconstructions reinforces the PINN's ability to recover intricate spatial features of the underlying quantum state.

Figure 41 shows the pair correlation function $G(r)$, which is directly proportional to $|u(r)|^2$ in this radial model. Once again, the PINN output aligns almost perfectly with the analytical result. The predicted most probable inter-electron distance, $r_{\max} = 2.494$ a.u., exactly matches the analytical value, yielding zero percent deviation. The classical turning point, $r_0 = 2.000$ a.u., is also correctly represented. A magnified inset focuses on the peak region of $G(r)$, verifying the model's precision in capturing key physical features.

Overall, the results validate the effectiveness of the configured PINN framework in solving the radial Schrödinger equation for this two-electron model system. The model not only achieves a relative energy error on the order of 10^{-7} and a wavefunction L^2 error of approximately 2.9×10^{-5}, but also accurately reproduces higher-order observables such as the 3D charge density and the pair correlation function. The successful capture of nodal features and proper decomposition of energy components highlights the expressive power of the network. Key to this performance are the network design (employing SiLU activation functions with appropriate depth and width), the enforcement of physical constraints (e.g., asymptotic behavior), and a carefully composed loss function incorporating PDE residuals, normalization, and shape fidelity. These results exemplify the potential of PINNs as robust solvers for quantum mechanical problems, especially in regimes where analytical solutions are unavailable or intractable.

Table XI. Replication of selected states from Taut's 1993 PRA Table I using FEM with $K = 1.0$. The FEM eigenvalues are compared directly to the analytical values ϵ_{Taut}.

Case	$\alpha = 1/\omega_r$	L	ω	n	$r_{\max}$	Elements	ϵ_{Taut}	Target	FEM	Abs. Diff	Rel. Diff
Taut $n = 2$	4.0000	0	0.250000	2	25.0	2500	0.625000	0.625000	0.625003	0.000003	0.000%
Taut $n = 3$	20.0000	0	0.050000	2	60.0	6000	0.175000	0.175000	0.175000	0.000000	0.000%
Taut $n = 4$ (1)	54.7386	0	0.018269	2	200.0	25000	0.082200	0.082200	0.082209	0.000009	0.011%
Taut $n = 4$ (2)	5.2614	1	0.190065	2	100.0	10000	0.855300	0.855300	0.855294	0.000006	0.001%

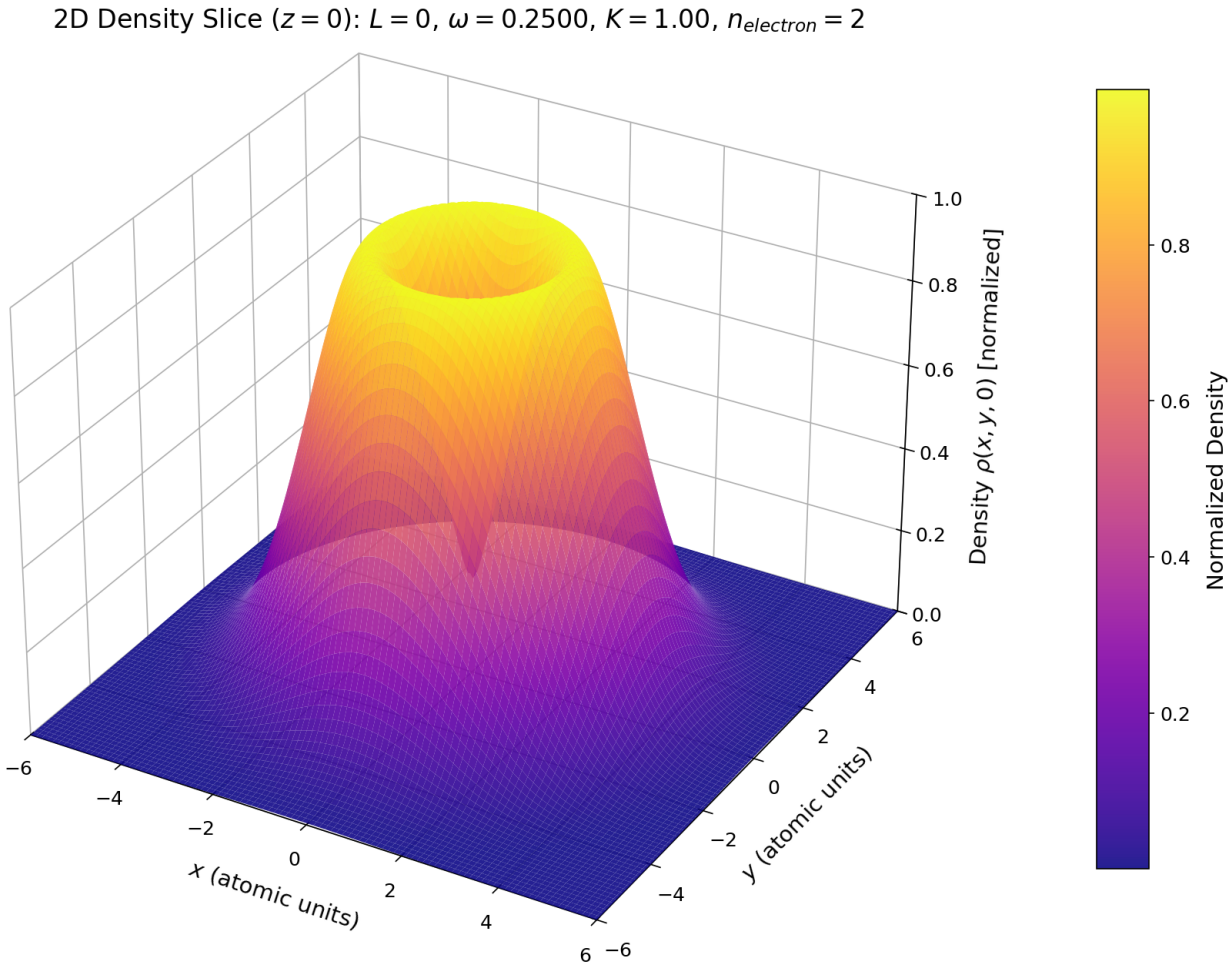

Figure 42. The 3D charge density for the two-electron system ($l = 0, n = 2, \omega = 0.2500$). (Left) FEM-predicted charge density.

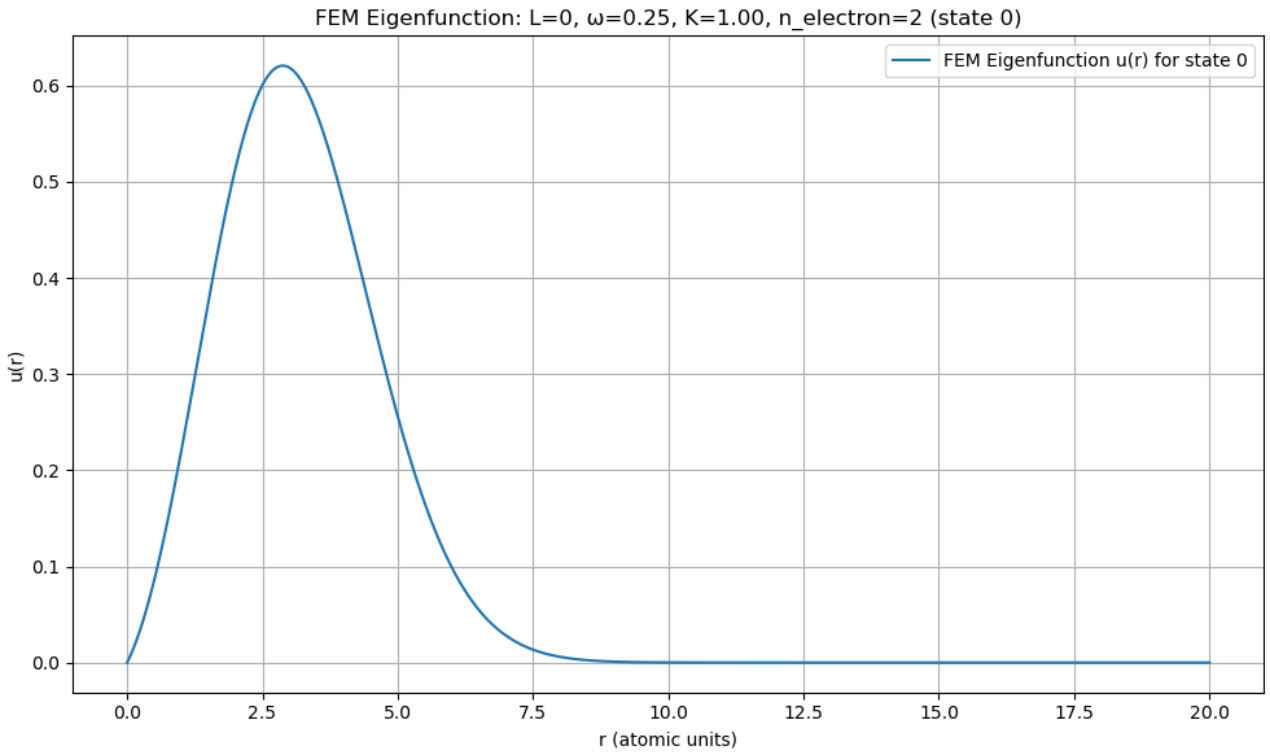

Figure 43. The 3D charge density for the two-electron system ($l = 0, n = 2, \omega = 0.2500$). (Left) FEM-predicted charge density.

Part VII

Conclusion

Chapter 14

14. Conclusion and Future Work

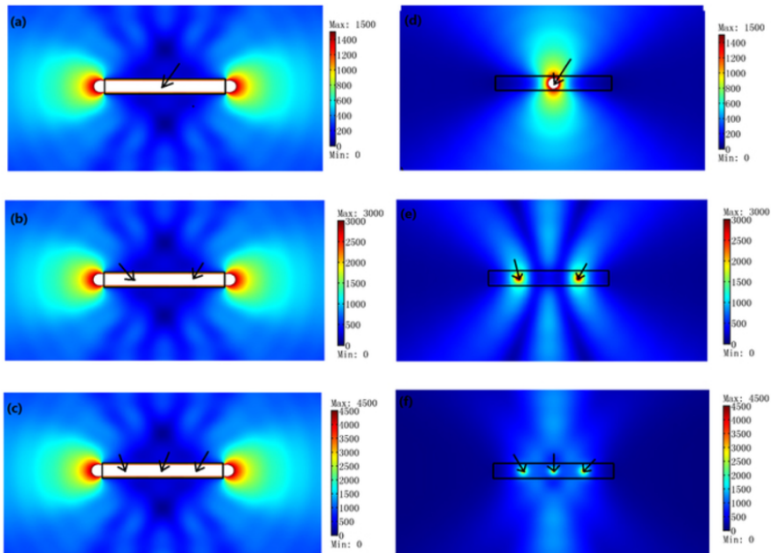

Figure 44. 2D FEM simulation results showing the absolute value of the z'-component of the electric field for a TE wave. The rectangular ONM, aligned along the x'-axis, has length $d = \frac{8\lambda_0}{3}$ and height $h = \frac{2\lambda_0}{3}$. (a) A single line current of unit amplitude (1 A) is placed at the center of the ONM. (b, c) Two and three in-phase coherent line currents are placed at equivalent positions along the same line ($y' =$ constant), respectively. Black arrows mark the current locations. (d–f) Corresponding simulations with the ONM removed for comparison.[13]

> “Prediction is very difficult, especially if it's about the future.[5]
>
> Niels Bohr”

i. Future work

[1] Physicsinformed neuralnetwork process diagram. https://www.mpie.de/4573079/machine-learning-for-materials-mechanics. Accessed: 2025-05-07.

[2] Iacs guidelines on numerical calculations for the purpose of deriving the vref in the framework of the eexi regulation. https://www.queseas.com/t/iacs-guidelines-on-numerical-calculations-for-the-purpose-of-deriving-the-vref-in-the-framework-of-the-eexi-r575, 2023. Accessed: 2025-05-07.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283. USENIX Association, 2016.

[4] Aristotle. *Metaphysics*. Oxford University Press, Oxford, UK, 1924. Translated and edited by W. D. Ross.

[5] Niels Bohr. Prediction is very difficult, especially if it's about the future. https://quoteinvestigator.com/2013/10/20/no-predict/. Accessed: 2025-05-07.

[6] George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.
[7] Jeffrey E. Boyd. The collapse of the quantum wave function: A review. *ResearchGate Preprint*, 2022. Figure 3: Feynman diagram of electron-electron scattering via virtual photon exchange.
[8] Columbia University Computing History. The IBM 701. Columbia University, 2021. Accessed: May 12, 2025. [http://www.columbia.edu/cu/computinghistory/701.html].
[9] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.
[10] Mostafa Dehghani, Alexey Gritsenko, Anurag Arnab, Matthias Minderer, and Yi Tay. Scenic: A jax library for computer vision research and beyond. *arXiv preprint arXiv:2110.11403*, 2021.
[11] W. Edwards Deming. *The New Economics for Industry, Government, Education.* MIT Press, 1993.
[12] Freeman Dyson. Science in the age of computers. *The Futurist*, 18(1):34–37, 1984.
[13] Andrés Díaz Lantada, Pascale Lafont, Juan Manuel Muñoz-Guijosa, and Miguel Hernando. Design of auxetic structures for mechanical energy absorption. *Physica Status Solidi (b)*, 253(7):1330–1341, 2016.
[14] Heinz Werner Engl, Martin Hanke, and Andreas Neubauer. *Regularization of Inverse Problems.* Springer, 1996.
[15] Richard P. Feynman. Blackboard quote at caltech. https://www.feynman.com/science/what-i-cannot-create/, 1988. Found written on Feynman's blackboard at the time of his death, Caltech.
[16] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. *The Feynman Lectures on Physics, Vol. 3: Quantum Mechanics.* Addison-Wesley, Reading, Massachusetts, 1965.
[17] Martin Fowler. Two hard things. https://martinfowler.com/bliki/TwoHardThings.html, 2013. Accessed: 2025-05-07.
[18] Paul A. Freiberger and Michael R. Swaine. ENIAC. Encyclopædia Britannica, Mar 2025. Accessed: May 12, 2025. [https://www.britannica.com/technology/ENIAC].
[19] Carl Friedrich Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium.* Fleischer, Hamburg, 1809. Introduces the method of least squares and its probabilistic foundations.
[20] Herman H. Goldstine and Adele Goldstine. The electronic numerical integrator and computer (eniac). *Mathematical Tables and Other Aids to Computation*, 2(15):97–110, 1946.
[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.
[22] Jacques Hadamard. *Lectures on Cauchy's Problem in Linear Partial Differential Equations.* Yale University Press, 1923.
[23] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, P. Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.
[24] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. *In A Field Guide to Dynamical Recurrent Neural Networks*, 2001. IEEE Press.
[25] Arthur E. Hoerl and Robert W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.
[26] Kurt Hornik. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.
[27] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.
[28] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, 2007.
[29] Taniya Kapoor, Hongrui Wang, Alfredo Núñez, and Rolf Dollevoet. Predicting traction return current in electric railway systems through physics-informed neural networks. *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit*, 236(10):1147–1160, 2022.
[30] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
[32] Adrien-Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes.* F. Didot, Paris, 1805. Includes appendix: Sur la Méthode des moindres carrés.
[33] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2019.
[34] Abraham H. Maslow. *The Psychology of Science: A Reconnaissance.* Harper Row, 1966.
[35] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[36] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[37] MISUMI USA. Computer aided design pt. 4: The world of fea. https://us.misumi-ec.com/blog/computer-aided-design-pt-4-the-world-of-fea/, 2021. Accessed: 2025-05-07.

[38] Paul-Antoine Moreau, Erika Toninelli, Thomas Gregory, and Miles J. Padgett. Imaging bell-type nonlocal behavior. *Science Advances*, 5(11):eaaw2563, 2019.

[39] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[40] Yurii Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Soviet Mathematics Doklady*, 27:372–376, 1983.

[41] Andrew Ng. What artificial intelligence can and can't do right now. *Harvard Business Review*, 2016.

[42] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.

[44] Cecilia Payne-Gaposchkin. *Stellar Atmospheres: A Contribution to the Observational Study of High Temperature in the Reversing Layers of Stars*. PhD thesis, Radcliffe College, 1925.

[45] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[46] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[47] Jakob Roth, Martin Reinecke, and Gordian Edenhofer. Jaxbind: Bind any function to jax. *Journal of Open Source Software*, 9(98):6532, 2024.

[48] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[49] Yingxiao Song. *Finite Element Method with Applications in Engineering*. Arcler Press, New York, 2017.

[50] J. Stodolna, A. Rouzée, F. Lépine, S. Cohen, F. Robicheaux, A. Gijsbertsen, and M. J. J. Vrakking. Hydrogen atoms under magnification: Direct observation of the nodal structure of stark states. *Physical Review Letters*, 110(21):213001, 2013.

[51] Steven Strogatz. When faced with a complex problem, divide it into simpler pieces. that's not cheating. that's how nature works. https://twitter.com/stevenstrogatz/status/1301122358614906880, 2020. Accessed: 2025-05-07.

[52] M. Taut. Two electrons in an external oscillator potential: Particular analytic solutions of a coulomb correlation problem. *Physical Review A*, 48(5):3561–3566, 1993.

[53] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[54] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5 - rmsprop: Divide the gradient by a running average of its recent magnitude. Coursera: Neural Networks for Machine Learning, 2012. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[55] Jan L. A. van de Snepscheut. In theory, there is no difference between theory and practice. but in practice, there is. https://quoteinvestigator.com/2018/04/15/theory/. Accessed: 2025-05-07.

[56] Pierre-François Verhulst. Notice sur la loi que la population poursuit dans son accroissement. *Correspondance Mathématique et Physique*, 10:113–121, 1838.

[57] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. SciPy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17:261–272, 2020.

[58] Hongyan Wu, Rui Li, Qiang Zhu, Chenglong Xu, and Wei Li. Comparative study of physics-informed neural networks and finite element method for solving forward and inverse problems. *Computational Intelligence and Neuroscience*, 2022:1–13, 2022.