Can Large Language Models Design Effective Neural Operators for Solving Partial Differential Equations?

Zeyuan Song [1 2] **Xiaocong Zhen** [3] **Zheyu Jiang** [1]

Abstract

Accurate numerical solutions of partial differential equations (PDEs) are crucial in numerous science and engineering applications. While neural operators can significantly accelerate the PDE solution process, designing neural architectures that target PDE structure, tailor problem specifications, and train reliably still requires substantial expert knowledge and human effort. We ask whether a large language model can design neural operators end-to-end. We present a four-agent design pipeline with distinct roles: Theorist, Programmer, Critic, and Refiner. The Theorist selects a mathematically grounded operator for a user-specified PDE and derives its formulation. The Programmer produces a self-contained PyTorch implementation. The Critic performs an adversarial review to expose numerical and software issues. The Refiner applies targeted corrections. An automated PDE solver completes the loop by generating data, training the synthesized model, and reporting evaluation metrics and plots. Across extensive PDE benchmark problems, the LLM-designed operators consistently outperform strong baselines in accuracy and sample efficiency, while remaining stable under varied discretizations and noisy initial conditions. Ablation studies show that the Critic and Refiner steps are essential for numerical stability and generalization. These results suggest that LLMs can act as principled collaborative designers of PDE operators, translating problem statements into executable and competitive architectures and moving toward automated and theory-aware scientific machine learning.

[1]School of Chemical Engineering, Oklahoma State University, Stillwater, USA [2]Department of Mathematics, Lehigh University, Bethlehem, Pennsylvania, USA [3]School of Industrial Engineering and Management, Oklahoma State University, Stillwater, USA. Correspondence to: Zeyuan Song <taekwon.song@okstate.edu>, Zheyu Jiang <zheyu.jiang@okstate.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

1. Introduction

A wide range of scientific and engineering phenomena, such as fluid mechanics, heat transfer, weather forecasting, and cell growth, are modeled by partial differential equations (PDEs). Traditional discretization-based numerical solvers can become quite slow, inefficient, and unstable (Hittinger & Banks, 2013; Sokic et al., 2011; Carey et al., 1993). Neural network (NN)-based solvers have shown great promise for effectively solving nonlinear PDEs . Although neural network (NN)-based solvers have shown great promise for accurate and efficient PDE solutions (Li et al., 2020a; Um et al., 2020; Xu & Darve, 2020; Song & Jiang, 2023; Smith et al., 2020), a key limitation is that they are typically trained at a specific resolution, which leads to poor generalization to problems at other resolutions. This motivates the development of resolution-free variants of NN-based solvers. Among them, neural operator learning, which directly learns mappings between infinite-dimensional functional spaces, has been proposed as a promising alternative neural solver for PDEs (Li et al., 2020b; Li & Ye, 2025; Tripura & Chakraborty, 2023; Lu et al., 2020).

Among existing neural operators, we observe that the top performers differ across different PDE problems. For example, Wang & Wang (2024) reported that Latent Neural Operator (LNO) exhibits $1.08\times$ and $2.42\times$ relative ℓ^2 error compared to transolver (Wu et al., 2024) on the airfoil and plasticity benchmark datasets (Li et al., 2023), respectively. Another key observation is that the performance of neural operators can improve or deteriorate even with minor architectural changes. Furthermore, the neural architecture design of most existing neural operator solvers has been "more of an art than a science", lacking rigorous mathematical basis and requiring significant intuition, expert experience, and trial-and-error experimentation (Sanderse et al., 2025). Although neural operators such as the Fourier Neural Operator (FNO) (Li et al., 2020b) and DeepONet (Lu et al., 2019) are grounded in theoretical insights, designing new components that align with these insights still relies heavily on human expertise and engagement.

Large language Model (LLM) agents have shown great promise in automating processes via human interactions, including mobile tasks (Wen et al., 2024; Guan et al., 2024),

hardware design (Xu et al., 2024b), scientific discovery (Zimmermann et al., 2025; Filimonov, 2024; Aamer et al., 2025), code generation (Koziolek et al., 2024; Xu et al., 2024a), hyperparameter tuning (Zhang et al., 2023), and mathematical problem solving (Bian et al., 2025). In terms of solving PDE problems, hybrid approaches that incorporate LLMs as standalone components within neural architectures can improve solver performance (Zhou et al., 2025; Lorsung & Farimani, 2024). However, these approaches are not automated, and the resulting architectures are not designed by LLMs. Meanwhile, existing fully automated LLM agents for PDEs include CodePDE (Li et al., 2025) and PINNsAgent (Wuwu et al., 2025). CodePDE generates and evaluates the code of traditional PDE solvers, whereas PINNsAgent generates code of physics-informed neural network (PINN) architecture. Both methods design the architectures of traditional and PINN-based solvers based on numerical performance rather than theoretical insights. To the best of our knowledge, no prior work exists on designing neural operators end-to-end with LLM agents guided by rigorous mathematical insights.

To bridge these gaps, we ask the following question:

Can LLM agents design effective neural operator architectures that target PDE structure, tailor problem specifications, and follow theoretical insights?

In this work, we propose a four-agent design pipeline with distinct roles: Theorist, Programmer, Critic, and Refiner. The Theorist selects an appropriate neural operator architecture and derives the mathematical theory specific to the given PDE problem; the Programmer translates the Theorist's insights into an efficient and clean PyTorch implementation. The Critic, serving as a skeptical but fair reviewer, further analyzes potential issues in both the Theorist's results and the PyTorch implementation and provides suggestions. Finally, the Refiner addresses all issues raised by the Critic and updates the PyTorch implementation accordingly.

Across comprehensive PDE benchmark datasets, our main findings are:

1. Our four-agent design pipeline can design mathematically grounded neural operators that outperform strong baselines and state-of-the-art human-designed neural operators for a variety of PDE problems on regular and irregular geometries.

2. The Critic and Refiner collaborate with one another to improve the rigor of the theory, numerical stability, and generalization.

3. LLM agents may face difficulties in effectively translating obscure theories them into effective neural operators. In such cases, LLMs often hallucinate and overfit to linguistic similarity while ignoring functional equivalence.

Overall, this work reframes neural operator design as an automated, coupled, and self-improving process of theory, implementation, review, and refinement, making neural operator design interpretable and accessible to expert or non-expert users for the first time. This transforms neural operator design from an art into a science.

2. The Proposed LLM Agent Framework

2.1. Neural Operator Learning

Neural operators are mesh- and resolution- independent neural architectures that can directly learn the mapping from the parameter space to the solution space of a PDE problem. In general, consider a PDE defined on a spatial domain $\Omega \subset \mathbb{R}^d$ and a time interval $(0, T]$:

$$\mathcal{L}_a[u(x,t)] = f(x,t), \quad \forall (x,t) \in D \times (0,T], \tag{1}$$

which is subject to a set of initial and boundary conditions. Here, the parameter function $a \in \mathcal{A}$ specifies the coefficients and initial and boundary conditions of Equation 1. Neural operators construct an accurate approximation for $\mathcal{G} : \mathcal{A} \to \mathcal{F}(D \times [0,T])$, which maps the parameter function a to the corresponding solution function $u(x,t) \in \mathcal{F}$, via a parametric mapping $\mathcal{G}_\theta$. The goal is to learn θ from a set of training data $\{(a_j, u_j)\}_j$, such that $\mathcal{G}_\theta \approx \mathcal{G}$.

2.2. Framework Overview

Designing neural operators from a scientific perspective requires several core steps: (i) propose or select a strong neural operator backbone, such as the FNO (Li et al., 2020b) or DeepONet (Lu et al., 2019); (ii) propose or identify an appropriate and rigorous mathematical theory that guarantees desirable properties (e.g., convergence, approximation error, function spaces); (iii) update the backbone architecture to align with the chosen theory; and (iv) implement and debug the code. As shown in Figure 1, our framework employs a four-agent pipeline to automate this workflow.

System Prompt. Prior studies show that role-playing instructions enable LLMs to collaborate under distinct roles, improving performance, particularly in code generation (Carlander et al., 2024; Dong et al., 2024; Takagi et al., 2025). Motivated by this, we assign the following distinct roles via system prompt: Theorist, a world-class research mathematician specializing in scientific machine learning; Programmer, an expert PyTorch developer in scientific machine learning; Critic, a skeptical but fair adversarial reviewer for a top AI conference; and Refiner, an expert PyTorch developer focused on debugging and refining complex models. Additional examples can be found in the Appendix.

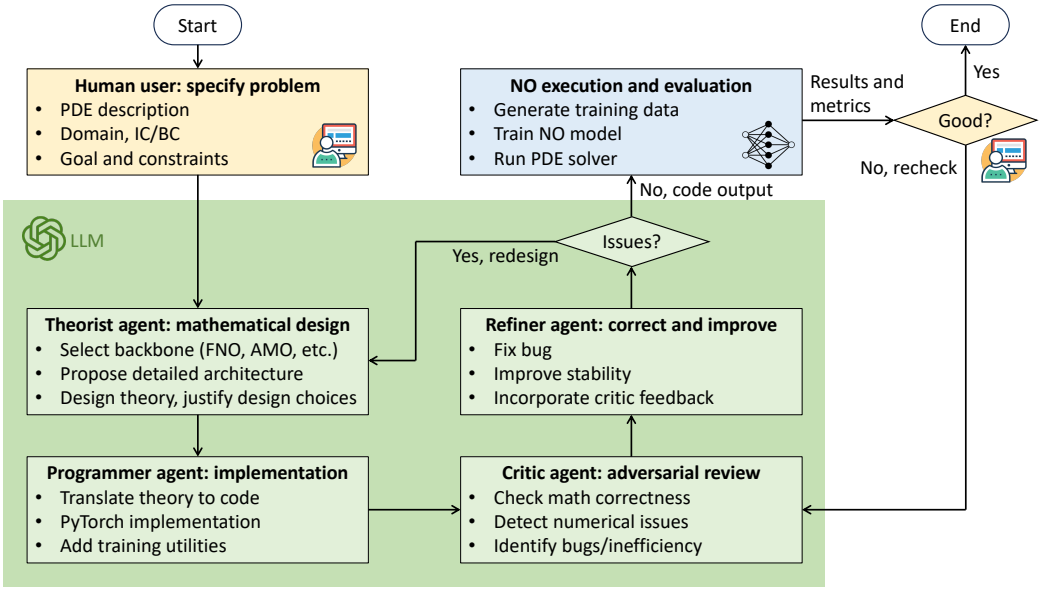

Figure 1. Our proposed automated four-agent neural operator design pipeline.

Step 1: Problem Specification. Given a PDE problem, we first translate the mathematical formulation (1) into natural language that LLMs can readily understand. This natural-language specification includes the problem name, equation, spatial domain, time interval, initial conditions, boundary conditions, and source terms. Instead of presenting this information in a paragraph, we use a concise, structured natural language format, which turns out to be effective in our proposed framework. For example, we present the 1-D Burgers' equation as:

PDE Problem Specifications

---

name: `1D Burgers' Equation`

equation_latex:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

domain: $x \in (0,1),\ t \in (0,1)$

initial_condition: $u(x,0) = u_0(x)$.

boundary_conditions: `Periodic`

viscosity: `0.01`

Step 2: Propose Mathematical Theory (Theorist). The Theorist's task is to provide rigorous theoretical basis and guidelines to improve the performance of the selected backbone. First, we provide the Theorist with a factory of existing neural operators, including FNO (Li et al., 2020b), DeepONet (Lu et al., 2019), Transolver (Wu et al., 2024), and LNO (Wang & Wang, 2024), as well as classical architectures such as the variational autoencoder (Kingma et al., 2013) and the Transformer (Li et al., 2022). We also allow the Theorist to utilize other backbones of its choosing. We then prompt the Theorist to develop clear, rigorous, and concise mathematical formulations that improve the selected backbone architecture for the specific problem. In this way, the Theorist offers a complete formulation and provides a tailored design of an updated neural architecture in both natural-language and mathematical forms. Finally, we prompt the Theorist to justify its choices and to implement self-correctness checks as determined by the Theorist. The output of this step is a script that derives the theoretical results and descriptions of the proposed neural operator.

Step 3: Code Generation (Programmer). After the Theorist provides the detailed formulation and instructions, we prompt the Programmer to translate them into code for the proposed neural operator, along with any necessary helper functions and package dependencies. We instruct the Programmer to generate the code in PyTorch due to its wide use in the machine learning community.

Step 4: Review Theoretical Results & Implementation (Critic). Motivated by the peer-review mechanism in AI conferences, we prompt the Critic to (i) critically analyze the mathematical derivations provided by the Theorist and the corresponding PyTorch code from the Programmer, and (ii) identify potential inconsistencies in the derivation, edge cases, numerical instabilities, and inefficiencies in the implementation. Finally, we instruct the Critic to provide a structured list of potential issues and concrete suggestions to improve the proposed neural operator.

Step 5: Refine Code (Refiner). The Refiner updates the implementation of the proposed neural operator to address issues and apply suggestions identified by the Critic.

Step 6: Code Execution. After the updated code is executed, any bugs will be recorded and reported back to the Critic, who then identifies issues and provides suggestions. The Refiner then revises the code accordingly, and this process iterates itself until no bug is found.

3. Numerical Experiments

Datasets. We evaluate the performance of neural operators designed by our proposed LLM-agent framework across six widely-used benchmark datasets:

1. **Darcy Flow** (Li et al., 2020b): Represents 2D flow through porous media. The PDE is discretized on a 421×421 grid and downsampled to 85×85. Inputs are coefficient fields $a(x)$, and outputs are solutions $u(x, t)$. The dataset contains 1,000 training and 200 testing samples with varying medium structures.
2. **Navier-Stokes** (Li et al., 2020b): Models the 2D incompressible Navier–Stokes equation in vorticity form on the unit torus. Each sample is a 64×64 spatiotemporal field with 20 frames, where the first 10 frames are used to predict the next 10. The dataset consists of 1,000 training and 200 testing samples.
3. **Elasticity** (Li et al., 2023): Predicts the internal stress of an elastic material discretized into 972 points. Each input is a 972×2 tensor of point positions, and the output is a 972×1 tensor of stresses. The dataset contains 1,000 training and 200 testing samples.
4. **Plasticity** (Li et al., 2023): Focuses on predicting the deformation of a plastic material under a die of arbitrary shape. The input is a structured mesh of size 101×31, and the output is the deformation over 20 timesteps, recorded as a $20 \times 101 \times 31 \times 4$ tensor. The dataset includes 900 training and 80 testing samples.
5. **Pipe** (Li et al., 2023): Estimates horizontal fluid velocity within pipes represented as a structured mesh of size 129×129. The input tensor ($129 \times 129 \times 2$) encodes positions, while the output tensor ($129 \times 129 \times 1$) gives velocity values. The dataset has 1,000 training and 200 testing samples.
6. **Airfoil** (Li et al., 2023): Concerns transonic flow over airfoils governed by the Euler equations. Inputs are structured meshes of size 221×51, and outputs are Mach number fields. The dataset includes 1,000 training and 200 testing samples derived from various airfoil designs.

Metrics. We train and evaluate the designed neural operators based on the relative ℓ^2 error:

$$\text{relative } \ell^2 \text{ error } = \frac{1}{N} \sum_{i=1}^{N} \frac{\|\mathcal{G}(a_i) - G(a_i)\|_{L^2}}{\|G(a_i)\|_{L^2}}, \quad (2)$$

where N denotes the number of samples.

Furthermore, we evaluate the correctness and rigor of the mathematical formulations produced by the Theorist through rigorous human expert review. This review was conducted by three independent PhD students specializing in computational mathematics and neural operators (who are not authors of this paper). The rubric was a binary "Yes/No" judgment based on two criteria: (i) "Is the Theorist's mathematical formulation (e.g., the derivation) correct and sound?"; and (ii) "Is the connection between the chosen theory and the target PDE justified and logical?"

Baselines. We compare LLM-designed neural operators to more than 10 strong baselines and state-of-the-art (SOTA) models designed by human experts, including FNO (Li et al., 2020b), U-FNO (Wen et al., 2022), F-FNO (Tran et al., 2021), LNO (Wang & Wang, 2024), ONO (Xiao et al., 2023), WMT (Gupta et al., 2022), Galerkin (Cao, 2021), LSM (Wu et al., 2023), OFormer (Li et al., 2022), Transolver (Wu et al., 2024), and LaMO (Tiwari et al., 2025).

Experimental Settings. We evaluate several LLMs in our framework, such as `gpt-5`, `gpt-5-mini`, and the reasoning models `o1` and `o3`. All experiments are conducted on a Linux workstation running Ubuntu (kernel 6.14, glibc 2.39) with Python 3.13.5 (Anaconda), PyTorch 2.8.0+cu129 (CUDA 12.9), an AMD Ryzen 9 9950X (16-core) processor, and a single NVIDIA GeForce RTX 4090 (48 GB) GPU.

4. Results and analysis

4.1. Can LLMs design neural operators?

We evaluate the performance of LLM-designed neural operators across six benchmark datasets and find that they outperform existing SOTA models on five of the six datasets (Table 1). Notably, the LLM generates diverse neural architectures tailored to different datasets, illustrating its adaptability across a wide range of tasks. In terms of accuracy, LLM-designed neural operators decrease the relative ℓ^2 error by approximately 6%, depending on the specific problem. Moreover, LLM-designed neural operators demonstrate superior computational efficiency, achieving a 30-50% reduction in computational time and requiring two to three orders of magnitude fewer parameters (Figure 2). These results suggest that LLMs are capable of designing neural operators that are both lightweight and accurate, thanks to their deep connections with rigorous mathematical principles. As a

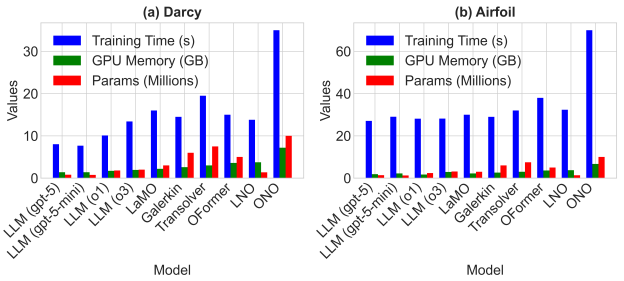

Figure 2. Comparison of computational efficiency including training time (sec per epoch), GPU memory (GB), and parameters count (M) on (a) Darcy and (b) Airfoil datasets.

side note, it is worth mentioning that, while LLM-designed neural operators do not achieve the highest performance on the Darcy dataset, this trade-off in accuracy is made in favor of improved efficiency (see Figure 2(a)).

Table 1. Relative ℓ^2 error comparisons of LLM-designed neural operators with baselines across six benchmark datasets. Lower relative ℓ^2 error is better. **Bold** means the best model, underline means the second best model, red means the third best model, and blue means the fourth best model.

Models	Elasticity	Plasticity	Airfoil	Pipe	N-S	Darcy
FNO (Li et al., 2020b)	0.0229	0.0074	0.0138	0.0067	0.0417	0.0052
U-FNO (Wen et al., 2022)	0.0239	0.0039	0.0269	0.0056	0.2231	0.0183
F-FNO (Tran et al., 2021)	0.0263	0.0047	0.0078	0.0070	0.2322	0.0077
LNO (Wang & Wang, 2024)	0.0052	0.0029	0.0051	0.0026	0.0845	0.0049
ONO (Xiao et al., 2023)	0.0118	0.0048	0.0061	0.0052	0.1195	0.0076
WMT (Gupta et al., 2021)	0.0359	0.0076	0.0075	0.0077	0.1541	0.0082
Galerkin (Cao, 2021)	0.0240	0.0120	0.0118	0.0098	0.1401	0.0084
LSM (Wu et al., 2023)	0.0218	0.0025	0.0059	0.0050	0.1535	0.0065
OFormer (Li et al., 2022)	0.0183	0.0017	0.0183	0.0168	0.1705	0.0124
Transolver (Wu et al., 2024)	0.0062	0.0013	0.0053	0.0047	0.0879	0.0059
LAMO (Tiwari et al., 2025)	0.0050	**0.0007**	0.0041	0.0038	0.0460	**0.0039**
LLM (gpt-5)	0.0049	0.0018	0.0043	0.0030	**0.0387**	0.0132
LLM (gpt-5-mini)	0.0051	0.0023	0.0052	0.0032	0.0420	0.0134
LLM (o1)	**0.0047**	**0.0007**	0.0041	0.0023	0.0389	0.0068
LLM (o3)	0.0049	**0.0007**	**0.0038**	**0.0022**	0.0512	0.0064

4.2. Does theory-aware design provide benefits?

To further explore the contribution of the theoretical insights provided by the Theorist in the design process, we conduct ablation studies comparing our LLM-agent framework to a variant without the Theorist. In the latter case, the LLM agents would generate neural operator code directly without theoretical guidance (as in Wuwu et al. (2025); Li et al. (2025)), and the Critic reviews only the numerical aspects. We measure the performance of both frameworks in terms of accuracy and generalization.

Table 2 reports the relative ℓ^2 errors of neural operators obtained with and without the Theorist. We observe that neural operators designed with the Theorist present have higher accuracy, demonstrating the effectiveness of incorporating theoretical insights into the design process. Moreover, the improvements are particularly apparent on more complex benchmark datasets, such as *Airfoil* and *Pipe*, where the error differences between the two frameworks range from approximately $3\times$ to $5\times$.

Table 2. Relative ℓ^2 error comparisons of neural operators designed by LLM frameworks with and without Theorist across six benchmark datasets. Lower is better.

Model	Elasticity	Plasticity	Airfoil	Pipe	N-S	Darcy
With Theorist						
LLM (gpt-5)	0.0049	0.0018	0.0043	0.0030	0.0387	0.0132
LLM (gpt-5-mini)	0.0051	0.0023	0.0052	0.0032	0.0420	0.0134
LLM (o1)	0.0047	0.0007	0.0041	0.0023	0.0389	0.0068
LLM (o3)	0.0049	0.0007	0.0038	0.0022	0.0512	0.0064
Without Theorist						
LLM (gpt-5)	0.0082	0.0047	0.0134	0.0094	0.1630	0.0188
LLM (gpt-5-mini)	0.0086	0.0055	0.0134	0.0116	0.1635	0.0192
LLM (o1)	0.0079	0.0026	0.0098	0.0102	0.1630	0.0106
LLM (o3)	0.0077	0.0031	0.0104	0.0103	0.1639	0.0117

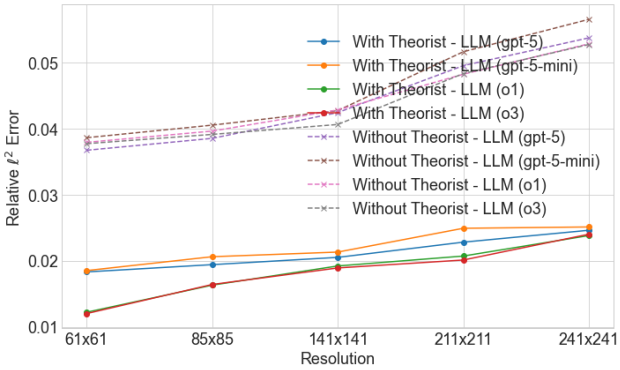

Figure 3. Relative ℓ^2 error comparisons of neural operators designed by LLM frameworks with and without Theorist on different resolutions.

To further examine the resolution transferability of LLM-designed neural operators, we follow the experimental setting in Wang & Wang (2024) and downsample the Darcy dataset from a resolution of 421×421 to 241×241, 211×211, 141×141, 85×85, 61×61, and 43×43. Neural operators are trained on the 43×43 dataset and tested on the others. Figure 3 shows that these theory-aware neural operators consistently outperform those without theoretical insights across all resolutions, not only on structured grids (e.g., the Navier-Stokes example) but also on irregular geometries (e.g., the Elasticity, Airfoil, Pipe, and Plasticity examples). This implies that the LLM-designed neural operators that incorporate theoretical insights exhibit strong generalization capability with respect to the number and resolution of sampling points.

4.3. Can Critic and Refiner produce better results?

In our framework, collaboration between the Critic and Refiner to identify drawbacks and potential issues in both the theory and the code implementation is a key step toward improving mathematical soundness, performance, and generalization. The Refiner step has been demonstrated to possess strong debugging capability (Li et al., 2025). To

analyze the contributions of the Critic and Refiner steps, we compare our full framework to a variant that includes only the Refiner step, which revises code according to reported bugs. Without the Critic step, we hypothesize the following:

> The quality of the output of Theorist directly determines the performance of the LLM-designed neural operators.

We then conduct experiments on the Darcy dataset and evaluate the relative ℓ^2 error across different resolutions. In addition, we introduce another LLM (Gemini 2.0 Thinking) as a judge, assigning a score (with a full mark of 5) to quantify the quality of the Theorist's feedback.

Table 3. Relative ℓ^2 error and score comparisons of neural operators designed by LLM frameworks with and without Critic across six benchmark datasets.

Model	61×61	85×85	141×141	211×211	241×241	**Score**
With Critic						
LLM (gpt-5)	0.0183	0.0194	0.0205	0.0228	0.0246	4.5
LLM (gpt-5-mini)	0.0185	0.0206	0.0213	0.0249	0.0251	4.4
LLM (o1)	0.0122	0.0163	0.0192	0.0207	0.0238	4.6
LLM (o3)	0.0120	0.0164	0.0189	0.0201	0.0240	4.6
Without Critic						
LLM (gpt-5)	0.0191	0.0216	0.0224	0.0231	0.0259	4.1
LLM (gpt-5-mini)	0.0305	0.0341	0.0355	0.0368	0.0384	3.5
LLM (o1)	0.0160	0.0177	0.0201	0.0216	0.0249	4.4
LLM (o3)	0.0162	0.0176	0.0204	0.0218	0.0257	4.4

Table 3 shows that, for `gpt-5-mini`, the Critic provides high-quality feedback, and the LLM-designed neural operator's accuracy drops by more than 30% when the Critic is absent. However, for `gpt-5`, `o1`, and `o3`, the absence of Critic only leads to a slight decrease in accuracy.

4.4. Can LLMs design neural operators using obscure math?

One interesting observation during the design process is that the Theorist tends to generalize the well-established theoretical insights and incorporate them into the selected backbone. Here, we further analyze whether LLMs can design neural operators based on "obscure mathematical results". If so, we may be able to greatly extend the boundary of neural operator design from combinations of existing components and incremental contributions to the existing neural architectures to entirely new neural operators not previously proposed in the literature. Here, "obscure mathematical results" refer to mathematical theories that have attracted attention from only a small fraction of researchers worldwide. Given the lack of sufficient training data on obscure mathematics, we hypothesize that LLMs may struggle with these topics.

To test this hypothesis, we conduct a new ablation study that instructs the Theorist to design a neural operator utilizing adaptive Fourier decomposition (AFD) (Qian et al., 2011) on the 1D Burgers' equation dataset (Li et al., 2020b). AFD is a a novel signal decomposition technique offering adaptive and explicit function representations in a reproducing kernel Hilbert space (RKHS) on various geometries and manifolds using Blaschke-type basis functions (Qian et al., 2011; 2012; Song & Sun, 2022). We find that, across all four LLM models tested, the relative ℓ^2 errors of neural operators guided by AFD are consistently one to two orders of magnitude lower than relative ℓ^2 errors of neural operators designed using well-established theories. For example, using `gpt-5`, the LLM-designed AFD neural operator achieves a relative ℓ^2 error of 0.013, whereas the LLM-designed neural operator based on FNO achieves 0.0006. In contrast, we also design a neural operator based on AFD manually by ourselves, which reaches a relative ℓ^2 error of 0.0009. This suggests that incorporating AFD theory in neural operator design could potentially improve the solver performance. Nevertheless, the LLM-designed AFD-based neural operator performs worse than the human-designed version. Ultimately, we find that the reason is that the Theorist does not fully understand AFD. To illustrate this, we showcase the *Choice and Justification* section of the Theorist's feedback, where the red parts highlights the Theorist's understanding of AFD theory:

Choice and Justification

– Choose an adaptive Fourier decomposition-based neural operator (AFD-NO): a Fourier neural operator in space with an adaptive, data-dependent selection/mixing of Fourier modes, wrapped inside a stable exponential time-differencing (ETD-1) integrator for the viscous term.
– Why it is best here:
1) Periodic boundary conditions and low-frequency initial conditions make the Fourier basis the natural eigenbasis; Burgers' nonlinearity redistributes energy across modes, so learning in the spectral domain is efficient.
2) Adaptivity is key: noisy initial data and the incipient formation of steep gradients require selectively attending to and mixing the subset of active modes. AFD layers learn soft, sample-dependent spectral masks and frequency-wise linear maps, improving data efficiency and robustness versus fixed-mode FNOs.
3) Stability and inductive bias: we treat the viscous linear part exactly in Fourier (ETD-1), leaving the network to learn only the convective nonlinearity. This aligns with the PDE's semigroup structure and improves numerical stability for $\mu = 0.01$.

The statements above have been evaluated by several senior researchers equipped with domain knowledge of AFD theory. It was determined that LLMs do not grasp the essence of AFD operation, i.e., rational approximation via pole selection in a RKHS. Instead, LLMs conflate Fourier decomposition with the Fourier transform and interpret poles as active modes. Additionally, they ignore the requirements on the basis and the function space needed to implement AFD.

Overall, LLMs produce misleading and incorrect content due to hallucination. Moreover, LLMs tend to overfit to similar terms involving adaptivity in Fourier transform theory, while overlooking their major differences.

We point out that our prompting strategy is highly structured to constrain the LLM's reasoning space, rather than relying on it to invent mathematics from scratch. Specifically, the prompt first explicitly provides the Theorist with *a factory of existing neural operators* (including FNO, DeepONet, LNO, and so on) as an in-context "toolbox". The Theorist then instructs itself to first select the most appropriate backbone architecture from this toolbox. Next, The Theorist guides itself to propose a specific mathematical modification to align that backbone with the specific theoretical properties of the given PDE (such as stiffness, boundary conditions, or conservation laws). Meanwhile, the AFD failure case indicates that, when the LLM is forced to use an "obscure" theory not in its pre-trained knowledge base, it tends to "hallucinate" and conflate concepts. Therefore, one needs to guide the LLM to apply established mathematical theories it already understands well, rather than blindly trusting the LLM's ability to perform novel or obscure mathematical derivations without caution.

Furthermore, it is worth noting that the ablation study presented in Section 4.2 only shows the impact of the Theorist component, not its initial correctness. The mechanism we adopt for the independent validation of the Theorist's output is human expert review. To clarify, this review is not conducted after the Critic or Refiner intervened. Instead, it is performed specifically on the initial mathematical formulation and architecture description generated by the Theorist before being passed to the Critic. For 5 out of the 6 benchmark problems we tested, the Theorist's initial theoretical proposal was judged "Yes" (i.e., theoretically correct and logically sound) by the human experts. This indicates that the Theorist provided a solid and reliable theoretical foundation in the majority of cases. The only exception was the AFD case, where the initial theory was indeed flawed, and this was accurately identified during the human expert review. Subsequently, the Critic's role focuses on identifying issues during the implementation stage (e.g., numerical instability, code inefficiency, or edge cases), rather than correcting fundamental theoretical errors.

4.5. How long does it take for LLM to design a neural operator?

In terms of design time cost, for a typical benchmark problem (e.g., 2D Navier-Stokes), the average wall-clock time our four-agent design workflow takes from receiving the problem statement to generating a validated, bug-free code is about 35-45 minutes. This process, running on our experimental workstation, requires an average of 7-9 full agent iterations (i.e., the Theorist $\rightarrow$ Programmer $\rightarrow$ Critic $\rightarrow$ Refiner loop).

While the "human expert effort" baseline is difficult to quantify precisely, it is known that neural operator design still involves expert experience and trial-and-error experimentation. Such process can take days to months. Thus, our proposed automated process presents a significant reduction in time cost compared to the amount of time a human expert would take to design, implement, and debug a competitive neural operator for a specific problem. Furthermore, it is worth mentioning that the time it takes for a non-expert in neural operators to develop a working neural operator solver for PDEs will be much longer.

5. Related Work

Baseline neural operators. Two foundational neural operator solver baselines are DeepONet (Lu et al., 2019), which learns nonlinear operators via a branch-trunk factorization, and the Fourier Neural Operator (FNO) (Li et al., 2020b), which applies spectral convolutions to achieve strong resolution transfer on canonical elliptic/parabolic problems. Subsequent variants improve accuracy, efficiency, or inductive bias: U-FNO couples Fourier mixing with U-Net refinements (Wen et al., 2022); F-FNO factorizes spectral weight tensors to lower complexity (Tran et al., 2021); multiwavelet formulations provide multiresolution locality and sparsity (Gupta et al., 2021; 2022; Tripura & Chakraborty, 2023); ONO augments operator learning with orthogonalized kernels and stability enhancements (Xiao et al., 2023); Galerkin operators embed variational structure (Cao, 2021); LSM exploits learned spectral methods (Wu et al., 2023); and transformer-style operators (OFormer, Transolver) leverage attention for long-range coupling (Li et al., 2022; Wu et al., 2024). Latent designs (LNO, LaMO) compress fields into compact representations to balance accuracy and cost (Wang & Wang, 2024; Tiwari et al., 2025). Beyond periodic grids, irregular geometries and structured meshes (e.g., airfoil, plasticity, pipe, elasticity) focus on resolution transfer and generalization (Li et al., 2023).

LLMs in scientific machine learning. LLMs have been used to automate elements of scientific workflows: program synthesis and code repair (Koziolek et al., 2024; Xu et al., 2024a), robotics/mobile task automation (Wen et al., 2024; Guan et al., 2024), hardware and systems design (Xu et al., 2024b), scientific discovery pipelines (Zimmermann et al., 2025; Filimonov, 2024; Aamer et al., 2025), hyperparameter tuning (Zhang et al., 2023), and mathematical problem solving (Bian et al., 2025). For PDEs, existing hybrid methods insert LLMs as standalone components within neural architectures or to provide natural-language rationales, rather than automating the full design loop (Zhou

et al., 2025; Lorsung & Farimani, 2024). Among the fully automated agents that generate codes and solvers, CodePDE synthesizes and evaluates traditional PDE codes (Li et al., 2025), whereas PINNsAgent targets PINN implementations (Wuwu et al., 2025). However, these approaches do not feature operator-theoretic design principles. In contrast, we position the adoption of LLM as a structured, theory-aware design assistant who proposes and justifies operator choices (e.g., spectral vs. multiresolution vs. latent), compiles them into executable PyTorch implementations subject to physics and numerical checks, and debugs the code if needed.

Automated agent systems. Role specialization and multi-agent coordination have been shown to improve complex code generation and iterative refinement via planning, self-critique, and division of labor (Carlander et al., 2024; Dong et al., 2024; Takagi et al., 2025). Recent agentic SciML systems instantiate the plan-execute-review loops for PDE tasks but largely emphasize execution or empirical tuning (Li et al., 2025; Wuwu et al., 2025). Our automated design pipeline adopts a four-role decomposition, *Theorist* (formal derivation and architectural justification), *Programmer* (faithful implementation), *Critic* (adversarial numerical/software review), and *Refiner* (targeted fixes), thereby explicitly incorporating mathematical rigor into software development. Compared with prior agent frameworks that lack principled operator-level guidance, our proposed theory-aware agent design aims to reduce hallucinations, improve stability under discretization changes, and rationalize neural operator framework selection.

6. Conclusion

To investigate whether LLM agents can facilitate the design of theory-aware neural operators that are accurate, efficient, and robust, we develop a four-agent neural operator design pipeline, closing the loop from problem specification and theory selection to implementation, adversarial review, refinement, and automatic training/evaluation. Across six PDE benchmarks, the LLM-designed neural operator solvers consistently match or outperform strong human-designed baselines in relative ℓ^2 error while using far fewer parameters and having much less training time and memory usage. Through multiple ablation studies, we show that the key to success of our multi-agent LLM design pipeline lies in its ability to expertly consolidate and apply well-established mathematical concepts under the rigorous guidance of the multi-agent system. The LLM's training data provides a strong, latent "code-to-math" and "math-to-code" inductive bias. Its ability to translate high-level mathematical concepts, such as Fourier bases, convolution, or spectral methods, into syntactically correct and structurally efficient PyTorch modules poses a significant advantage over labor-intensive human design efforts. Together, we demonstrate that neural operator design can be systematized as a coupled process of theory, implementation, and structured review, making it interpretable and accessible to domain scientists and non-expert users.

Impact Statement

This paper presents a novel automated framework whose goal is to design neural operators to advance the learning of complex dynamics present in various physical phenomena, such as those in PDEs and Hamiltonian systems. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

Acknowledgments

This work is supported by the U.S. National Science Foundation award No. 2442806.

References

Aamer, N., Asim, M. N., Munir, S., and Dengel, A. Automating ai discovery for biomedicine through knowledge graphs and LLM agents. *bioRxiv*, pp. 2025–05, 2025.

Bian, R., Geng, Y., Yang, Z., and Cheng, B. Automathkg: The automated mathematical knowledge graph based on llm and vector database. *Computational Intelligence*, 41 (4):e70096, 2025.

Cao, S. Choose a transformer: Fourier or galerkin. *Advances in neural information processing systems*, 34: 24924–24940, 2021.

Carey, C., Scanlon, T., and Fraser, S. SUCCA—an alternative scheme to reduce the effects of multidimensional false diffusion. *Applied Mathematical Modelling*, 17(5): 263–270, 1993.

Carlander, D., Okada, K., Engström, H., and Kurabayashi, S. Controlled chain of thought: Eliciting role-play understanding in llm through prompts. In *2024 IEEE Conference on Games (CoG)*, pp. 1–4. IEEE, 2024.

Dong, Y., Jiang, X., Jin, Z., and Li, G. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

Filimonov, V. Y. Large language models and their role in modern scientific discoveries. *Philosophical Problems of IT & Cyberspace (PhilIT&C)*, 25(1):42–57, 2024.

Guan, Y., Wang, D., Chu, Z., Wang, S., Ni, F., Song, R., and Zhuang, C. Intelligent agents with llm-based process automation. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5018–5027, 2024.

Gupta, G., Xiao, X., and Bogdan, P. Multiwavelet-based operator learning for differential equations. *Advances in neural information processing systems*, 34:24048–24062, 2021.

Gupta, G., Xiao, X., Balan, R., and Bogdan, P. Non-linear operator approximations for initial value problems. In *International Conference on Learning Representations (ICLR)*, 2022.

Hittinger, J. A. and Banks, J. W. Block-structured adaptive mesh refinement algorithms for Vlasov simulation. *Journal of Computational Physics*, 241:118–140, 2013.

Kingma, D. P., Welling, M., et al. Auto-encoding variational Bayes, 2013.

Koziolek, H., Grüner, S., Hark, R., Ashiwal, V., Linsbauer, S., and Eskandani, N. LLM-based and retrieval-augmented control code generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pp. 22–29, 2024.

Li, H., Schwab, J., Antholzer, S., and Haltmeier, M. Nett: solving inverse problems with deep neural networks. *Inverse Problems*, 36(6):065005, 2020a.

Li, K. and Ye, W. D-FNO: A decomposed Fourier neural operator for large-scale parametric partial differential equations. *Computer Methods in Applied Mechanics and Engineering*, 436:117732, 2025.

Li, S., Marwah, T., Shen, J., Sun, W., Risteski, A., Yang, Y., and Talwalkar, A. CodePDE: An inference framework for llm-driven PDE solver generation. *arXiv preprint arXiv:2505.08783*, 2025.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., and Anandkumar, A. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020b.

Li, Z., Meidani, K., and Farimani, A. B. Transformer for partial differential equations' operator learning. *arXiv preprint arXiv:2205.13671*, 2022.

Li, Z., Huang, D. Z., Liu, B., and Anandkumar, A. Fourier neural operator with learned deformations for PDEs on general geometries. *Journal of Machine Learning Research*, 24(388):1–26, 2023.

Lorsung, C. and Farimani, A. B. Explain like i'm five: Using LLMs to improve PDE surrogate models with text. *arXiv preprint arXiv:2410.01137*, 2024.

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

Lu, P. Y., Kim, S., and Soljačić, M. Extracting interpretable physical parameters from spatiotemporal systems using unsupervised learning. *Physical Review X*, 10(3):031056, 2020.

Qian, T., Zhang, L., and Li, Z. Algorithm of adaptive Fourier decomposition. *IEEE Transactions on Signal Processing*, 59(12):5899–5906, 2011.

Qian, T., Sprößig, W., and Wang, J. Adaptive fourier decomposition of functions in quaternionic hardy spaces. *Mathematical Methods in the Applied Sciences*, 35(1): 43–64, 2012.

Sanderse, B., Stinis, P., Maulik, R., and Ahmed, S. E. Scientific machine learning for closure models in multiscale problems: A review. *Foundations of Data Science*, 7(1): 298–337, 2025.

Smith, J. D., Azizzadenesheli, K., and Ross, Z. E. Eikonet: Solving the eikonal equation with deep neural networks. *IEEE Transactions on Geoscience and Remote Sensing*, 59(12):10685–10696, 2020.

Sokic, E., Konjicija, S., Ahic-Djokic, M., and Salihbegovic, A. Stability issues in discretization of wave equation. In *2011 18th International Conference on Systems, Signals and Image Processing*, pp. 1–4. IEEE, 2011.

Song, Z. and Jiang, Z. A data-driven modeling approach for water flow dynamics in soil. In *Computer Aided Chemical Engineering*, volume 52, pp. 819–824. Elsevier, 2023.

Song, Z. and Sun, Z. Representing functions in H^2 on the kepler manifold via wpoafd based on the rational approximation of holomorphic functions. *Mathematics*, 10(15):2729, 2022.

Takagi, H., Moriya, S., Sato, T., Nagao, M., and Higuchi, K. A framework for efficient development and debugging of role-playing agents with large language models. In *Proceedings of the 30th International Conference on Intelligent User Interfaces*, pp. 70–88, 2025.

Tiwari, K., Dutta, N., Krishnan, N., et al. Latent mamba operator for partial differential equations. *arXiv preprint arXiv:2505.19105*, 2025.

Tran, A., Mathews, A., Xie, L., and Ong, C. S. Factorized fourier neural operators. *arXiv preprint arXiv:2111.13802*, 2021.

Tripura, T. and Chakraborty, S. Wavelet neural operator for solving parametric partial differential equations in computational mechanics problems. *Computer Methods in Applied Mechanics and Engineering*, 404:115783, 2023.

Um, K., Brand, R., Fei, Y. R., Holl, P., and Thuerey, N. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in neural information processing systems*, 33:6111–6122, 2020.

Wang, T. and Wang, C. Latent neural operator for solving forward and inverse pde problems. *Advances in Neural Information Processing Systems*, 37:33085–33107, 2024.

Wen, G., Li, Z., Azizzadenesheli, K., Anandkumar, A., and Benson, S. M. U-fno—an enhanced fourier neural operator-based deep-learning model for multiphase flow. *Advances in Water Resources*, 163:104180, 2022.

Wen, H., Li, Y., Liu, G., Zhao, S., Yu, T., Li, T. J.-J., Jiang, S., Liu, Y., Zhang, Y., and Liu, Y. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pp. 543–557, 2024.

Wu, H., Hu, T., Luo, H., Wang, J., and Long, M. Solving high-dimensional pdes with latent spectral models. *arXiv preprint arXiv:2301.12664*, 2023.

Wu, H., Luo, H., Wang, H., Wang, J., and Long, M. Transolver: A fast transformer solver for pdes on general geometries. *arXiv preprint arXiv:2402.02366*, 2024.

Wuwu, Q., Gao, C., Chen, T., Huang, Y., Zhang, Y., Wang, J., Li, J., Zhou, H., and Zhang, S. PINNsAgent: Automated PDE surrogation with large language models. *arXiv preprint arXiv:2501.12053*, 2025.

Xiao, Z., Hao, Z., Lin, B., Deng, Z., and Su, H. Improved operator learning by orthogonal attention. *arXiv preprint arXiv:2310.12487*, 2023.

Xu, J., Du, W., Liu, X., and Li, X. Llm4workflow: An llm-based automated workflow model generation tool. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 2394–2398, 2024a.

Xu, K. and Darve, E. Adcme: Learning spatially-varying physical fields using deep neural networks. *arXiv preprint arXiv:2011.11955*, 2020.

Xu, K., Qiu, R., Zhao, Z., Zhang, G. L., Schlichtmann, U., and Li, B. Llm-aided efficient hardware design automation. *arXiv preprint arXiv:2410.18582*, 2024b.

Zhang, M. R., Desai, N., Bae, J., Lorraine, J., and Ba, J. Using large language models for hyperparameter optimization. *arXiv preprint arXiv:2312.04528*, 2023.

Zhou, H., Ma, Y., Wu, H., Wang, H., and Long, M. Unisolver: Pde-conditional transformers towards universal neural pde solvers. In *Forty-second International Conference on Machine Learning*, 2025.

Zimmermann, Y., Bazgir, A., Al-Feghali, A., Ansari, M., Bocarsly, J., Brinson, L. C., Chiang, Y., Circi, D., Chiu, M.-H., Daelman, N., et al. 32 examples of LLM applications in materials science and chemistry: towards automation, assistants, agents, and accelerated scientific discovery. *Machine Learning: Science and Technology*, 2025.

A. Example Prompts

This appendix presents the complete prompting strategy used in our four-agent framework, followed by a full end-to-end example for the 2D Darcy Flow problem. Appendix B reproduces the Critic's structured review of the initial operator implementation produced by LLM `o1`. Appendix C gives the refined operator code produced by the Refiner (LLM `o3`) after all Critic issues were resolved. Appendix D provides the complete Theorist derivation (LLM `o3`) that motivated the FNO architecture adopted for this problem.

A.1. Generated Artifact Files

For each LLM evaluated on the 2D Darcy Flow benchmark, our four-agent framework produces three output files: a Theorist derivation (`.md`), a Critic review (`.md`), and the final refined operator implementation (`.py`). Table 4 lists all twelve files together with the appendix where each is reproduced in full.

Table 4. All generated artifact files for the 2D Darcy Flow benchmark. Each row is one LLM model; the rightmost column gives the appendix where the file is reproduced in full.

LLM	Theorist Derivation	Critic Review	Operator Code	Appendices
gpt-5	theorist_derivation_darcy_flow_5.md	critic_review_darcy_flow_5.md	operator_code_darcy_flow_5.py	E, F, G
gpt-5-mini	theorist_derivation_darcy_flow_5_mini.md	critic_review_darcy_flow_5_mini.md	operator_code_darcy_flow_5_mini.py	H, I, J
o1	theorist_derivation_darcy_flow_o1.md	critic_review_darcy_flow_o1.md	operator_code_darcy_flow_o1.py	K, B, L
o3	theorist_derivation_darcy_flow_o3.md	critic_review_darcy_flow_o3.md	operator_code_darcy_flow_o3.py	D, M, C

A.2. Problem Specification Format

Each PDE problem is encoded in a structured natural-language format before being passed to the agent pipeline. Below is the specification used for the 2D Darcy Flow benchmark.

PDE Problem Specifications: 2D Darcy Flow

---

name: `2D Darcy Flow`

equation_latex:

$$-\nabla \cdot \big(a(x)\,\nabla u(x)\big) = f(x), \quad x \in (0,1)^2$$

domain: $x \in (0,1)^2$

boundary_conditions: `Dirichlet,` $u = 0$ `on` $\partial\Omega$

input: `coefficient field` $a(x)$`, discretized on an` 85×85 `grid`

output: `solution field` $u(x)$`, discretized on an` 85×85 `grid`

dataset_size: `1,000 training samples, 200 testing samples`

A.3. Agent System Prompts

Each agent receives a role-specific system prompt that defines its expertise and objectives. The four prompts below are used verbatim in all experiments.

THEORIST

System Prompt: Theorist

You are a world-class research mathematician specializing in scientific machine learning and the numerical analysis of partial differential equations. Your task is to:

1. Analyze the provided PDE problem specification carefully.
2. Select the most appropriate neural operator backbone from the provided factory (FNO, DeepONet, Transolver, LNO, or another architecture of your choosing).
3. Propose a rigorous mathematical modification to the selected backbone that is tailored to the structural properties of this specific PDE (e.g. boundary conditions, stiffness, conservation laws, symmetries).
4. Derive the complete mathematical formulation of the proposed neural operator in precise notation.
5. Justify your architectural choices from a theoretical perspective (e.g. convergence rates, approximation theory, spectral properties).
6. Conduct self-correctness checks on the derivation before finalizing.

Output a structured report containing: **(a)** choice and justification, **(b)** mathematical derivation, **(c)** proposed architecture description in natural language, and **(d)** self-correctness verification.

PROGRAMMER

System Prompt: Programmer

You are an expert PyTorch developer specializing in scientific machine learning and neural operator implementations. Your task is to:

1. Carefully read the mathematical derivation and architecture description provided by the Theorist.
2. Translate the proposed neural operator into a clean, self-contained, and fully functional PyTorch implementation.
3. Include all necessary helper functions, module definitions, and package imports.
4. Ensure the implementation faithfully reflects the mathematical formulation: correct tensor shapes, proper use of FFT routines, correct handling of boundary conditions, etc.
5. Write a brief smoke test (`if __name__ == ``__main__''`) that instantiates the model with a small synthetic input and verifies that the forward and backward passes complete without errors.

Output a single, fully executable Python file with no placeholder sections.

CRITIC

System Prompt: Critic

You are a skeptical but fair adversarial reviewer for a top-tier AI conference (e.g. NeurIPS, ICML). Your task is to critically evaluate both the mathematical derivation produced by the Theorist and the PyTorch implementation produced by the Programmer. Specifically:

1. Identify any logical gaps, unjustified steps, or errors in the mathematical derivation.
2. Check whether the PyTorch implementation faithfully reflects the stated derivation.
3. Expose potential numerical instabilities (e.g. FFT normalization issues, gradient explosion/vanishing, poor initialization strategies).
4. Identify edge cases that may cause silent failures (e.g. non-square grids, boundary enforcement, tensor-shape mismatches).
5. Point out inefficiencies in memory usage or computational cost.
6. Suggest concrete, actionable fixes for every issue identified.

Organize your output as a numbered list of issues, each with a clear description and a specific suggestion for resolution.

REFINER

System Prompt: Refiner

You are an expert PyTorch developer specializing in debugging and refining complex neural-network models for scientific machine learning. Your task is to:

1. Carefully read the Critic's structured list of issues and suggestions.
2. Address *every* identified issue in the PyTorch implementation.
3. If a runtime bug report from code execution is provided, diagnose and fix the root cause.
4. Preserve the overall architecture and mathematical intent from the Theorist's derivation while improving correctness, numerical stability, and efficiency.
5. Ensure the revised implementation passes the smoke test without errors.

Output the *complete* revised Python file, not just a diff or partial code.

A.4. User Prompts

In addition to the system prompts above, each agent receives a structured user prompt. The user prompt for the Theorist comprises the problem specification (Section A.2) prepended with the following instruction block:

User Prompt Instruction Block (Theorist)

Given the PDE specification above, perform the following steps:

Step 1 – Backbone selection. From the neural operator factory {FNO, DeepONet, Transolver, LNO}, select the backbone best suited for this problem. If none are appropriate, propose an alternative and justify your choice.

Step 2 – Theoretical augmentation. Propose a mathematically rigorous modification that exploits the specific structure of the PDE (boundary conditions, symmetries, spectral properties, etc.). State the modification as a precise definition or theorem.

Step 3 – Full derivation. Derive the complete forward map $\mathcal{G}_\theta$ of the proposed architecture, including all intermediate transformations.

Step 4 – Self-check. Verify dimensional consistency of every expression and confirm that the architecture reduces to a known result in limiting cases (e.g. uniform coefficient $a \equiv 1$).

The Programmer receives the Theorist's complete output together with the instruction: *"Implement the above neural operator in PyTorch exactly as derived. Return a single executable* `.py` *file."*

The Critic receives both the Theorist's output and the Programmer's code, together with the instruction: *"Review the derivation and code above. Produce a numbered list of issues with concrete fix suggestions."*

The Refiner receives the Programmer's code, the Critic's review, and any runtime error logs, together with the instruction: *"Fix all issues listed by the Critic and resolve any runtime errors. Return the complete revised* `.py` *file."*

B. Critic Review for Darcy Flow (LLM o1)

Below is the full structured review generated by the Critic agent (LLM `o1`) for the initial Darcy Flow operator implementation. The review covers both the theoretical derivation and the PyTorch code, identifying nine actionable issues that are subsequently addressed by the Refiner.

Issue 1: Handling of Dirichlet Boundaries and PDE Inputs. **Issue.** The 2D Darcy PDE includes both the coefficient $a(x)$ and a source term $f(x)$, but the initial code uses only one input channel (`in_channels=1`). Moreover, boundary conditions are only "optionally clamped" after prediction, which can break gradient flow through boundary nodes.

Suggestion A. In realistic scenarios, add $f(x)$ and/or boundary data as separate input channels (increasing `in_channels`). This way, the network consistently "sees" the forcing/boundary data during training.

Suggestion B. If you must enforce Dirichlet boundaries, consider masking or imposing them *inside* the forward pass (e.g. a

boundary layer that explicitly fixes boundary values). This ensures boundary information is respected throughout training, with gradients flowing correctly for interior nodes while boundary nodes remain fixed.

Issue 2: Potential Indexing / Einstein-Summation Inconsistency. **Issue.** The line

```
out_ft_trunc = torch.einsum('bick, ockk -> bock', x_ft_trunc, self.weights)
```

uses ambiguous index names. Given `x_ft_trunc.shape = (B, in_c, m1, m2)` and `weights.shape = (out_c, in_c, m1, m2)`, the intended contraction is over the channel dimension only.

Suggestion. Double-check that the shapes match exactly as intended. A common pattern is

```
torch.einsum('b i p q, o i p q -> b o p q', x_ft_trunc, self.weights)
```

and add an assertion that verifies shape compatibility before the summation.

Issue 3: Truncation of Negative Frequencies. **Issue.** The code retains only the `[0:modes1, 0:modes2]` corner of the frequency spectrum. For real-valued signals on an $N \times N$ grid, one often includes symmetric negative frequencies or uses real-FFT routines (`rfft2`), which store only the non-redundant half.

Suggestion A. Consider symmetrical slicing (or a real-FFT approach) to include negative frequencies. Truncating only the first quadrant may degrade representations for 2D signals that contain distinct negative-frequency content.

Suggestion B. If speed/memory is the concern, `torch.fft.rfft2` is more efficient and ensures conjugate symmetry is handled automatically.

Issue 4: Numerical Scaling and FFT Normalization. **Issue.** The derivation and code use `norm='ortho'` in both `fft2` and `ifft2`. While this is internally consistent, it differs from some standard FNO references, which may use the default "backward" normalization in inverse transforms.

Suggestion A. Check consistency of the chosen norm across the entire training pipeline. If referencing an existing FNO paper or code, match their normalization approach for reproducible performance.

Suggestion B. Consider data normalization or scaling if $a(x)$, $f(x)$, or $u(x)$ vary widely in magnitude. Extremely large or small values can cause FFT coefficients to saturate or produce large gradients in the frequency domain.

Issue 5: Treatment of the Imaginary Part. **Issue.** After `ifft2`, only the `.real` component is retained. If the inputs and weights are purely real, the imaginary part should be minimal. However, small numerical discrepancies or partial frequency truncation can lead to non-negligible imaginary components.

Suggestion. Monitor the magnitude of `x_out.imag` during debugging/training (even if ultimately discarded). A growing imaginary part might signal either a mismatch in frequency slicing or issues in the spectral weight initialization.

Issue 6: Edge Cases – Grid Shapes and Parameter Choices. **Issue A.** Non-square domains. The code slices frequency space as `[:modes1, :modes2]`. If $H \neq W$, the chosen `modes1`, `modes2` must not exceed the actual FFT dimensions.

Suggestion. Validate or assert that `modes1` $\leq H$ and `modes2` $\leq W$. If the domain shape changes or is not uniform, provide a dynamic or carefully configured approach to sizing modes per dimension.

Issue B. Arbitrary boundary shapes. If the PDE domain or boundary geometry deviates from a regular $(0, 1)^2$ uniform grid, standard 2D FFT assumptions may be violated.

Suggestion. For more complex geometries, a custom approach (e.g. Fourier transforms on a padded bounding box or non-uniform transforms) might be required.

Issue 7: Efficiency and Memory Considerations. **Issue.** The code stores a `cfloat` tensor of shape `(out_channels, in_channels, modes1, modes2)` per spectral layer. For large hidden dimensions or many modes, this can become large in memory. Additionally, repeated calls to `torch.fft.fft2` and `torch.fft.ifft2` can be expensive at high resolution.

Suggestion. If memory is an issue, reduce the hidden dimension or the number of modes, or use half-precision (`fp16`) where stable. Also explore real-only parameter representations that enforce conjugate symmetry in the frequency domain. For distributed setups, consider overlapping FFT operations with communication.

Issue 8: Forcing Terms and Multi-Channel Inputs. **Issue.** The derivation explicitly mentions $a(x)$ and $f(x)$, yet the code shows only a single input channel. Without $f(x)$, the model cannot fully learn $\nabla \cdot (a \nabla u) = f$ unless the training set is restricted to a single fixed forcing.

Suggestion. When the PDE has a parametric forcing term, incorporate an extra channel for $f(x)$. If boundary or initial conditions vary, represent those internally as additional channels (e.g. $[a, f, \text{boundary_mask}, \ldots]$).

Issue 9: Architectural Extensions and Improvements. **Residual or skip connections.** Repeatedly applying a spectral plus pointwise convolution can cause vanishing or exploding gradients if L is large. Standard FNO implementations often include skip connections either across layers or around the entire block.

Activation choice. The code uses GELU or ReLU. For PDE solutions with large or oscillatory values, smoother activations (e.g. Tanh) sometimes help.

Boundary loss weighting. If boundary conditions are crucial, explicitly weighting boundary-node losses higher can help the operator learn correct boundary behavior.

Summary of Key Recommendations.

- Incorporate boundary/forcing data into input channels and/or enforce Dirichlet conditions with a consistent forward-pass masking approach rather than a hard post-hoc clamp.
- Verify the Einstein summation indices to ensure correct channel-wise multiplication in the frequency domain.
- Consider negative-frequency components or use `rfft2` for truly real-valued signals—ignoring negative frequencies may degrade representational capacity.
- Check normalization choices in FFT/iFFT and consider data scaling to avoid large-magnitude coefficients.
- Watch out for memory usage with large `modes` or large hidden channel sizes.
- For complex PDEs or varying domain shapes, adapt how you slice frequencies or parameterize the spectral kernels.

C. Refined Operator Code for Darcy Flow (LLM o3)

Below is the complete refined `FNO2d` implementation produced by the Refiner agent (LLM `o3`) after addressing the main issues identified by the Critic in Appendix B. The header docstring summarizes the key changes relative to the initial Programmer output.

```python
"""
Refined Fourier Neural Operator (2-D)
-------------------------------------
Addresses main issues raised in the code review:

* grid-independent scaling (norm='forward')
* Hermitian-symmetric kernel (learn both +-ky)
* He initialisation for complex weights
* separate modes_h / modes_w for non-square grids
* boundary values are zeroed *inside every layer*
* boundary mask is *not* fed as a learnable feature
"""

from typing import Optional, Tuple

import torch
import torch.nn as nn
import torch.fft
from torch.nn import functional as F


class SpectralConv2d(nn.Module):
```

```python
    """
    2-D spectral convolution
    keeps the first `modes_h`, `modes_w` complex coefficients along (H,W)
    """
    def __init__(self,
                 in_channels:  int,
                 out_channels: int,
                 modes_h:      int,
                 modes_w:      int) -> None:
        super().__init__()

        self.in_channels  = in_channels
        self.out_channels = out_channels
        self.modes_h      = modes_h
        self.modes_w      = modes_w

        # ---- He initialisation for a complex weight tensor --------------
        scale = (2.0 / (in_channels + out_channels)) ** 0.5
        weight_r = torch.randn(in_channels, out_channels,
                               modes_h, modes_w) * scale
        weight_i = torch.randn(in_channels, out_channels,
                               modes_h, modes_w) * scale
        weight   = torch.complex(weight_r, weight_i)       # (Cin,Cout,Ky,Kx)

        self.weight = nn.Parameter(weight)                 # learnable kernel

    # ------------------------------------------------------------------
    @staticmethod
    def _compl_mul2d(input_ft: torch.Tensor,
                     weight:   torch.Tensor) -> torch.Tensor:
        """
        (B,Cin,Ky,Kx) x (Cin,Cout,Ky,Kx)  -> (B,Cout,Ky,Kx)
        """
        return torch.einsum("b c h w, c o h w -> b o h w", input_ft, weight)

    # ------------------------------------------------------------------
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        x : real tensor (B,C,H,W)
        -> real tensor (B,C_out,H,W)
        """
        B, _, H, W = x.shape
        W_r = W // 2 + 1                      # rfft width

        # --- safety: make sure we have enough spatial resolution ------
        if H < self.modes_h * 2 or W_r < self.modes_w:
            raise ValueError(
                f"Input resolution {(H,W)} too small for the requested "
                f"modes {(self.modes_h, self.modes_w)}"
            )

        # 1) forward FFT  (grid-independent scaling with 'forward')
        x_ft = torch.fft.rfft2(x, norm='forward')          # (B,C,H,W_r)

        # 2) allocate output spectrum
        out_ft = torch.zeros(B, self.out_channels, H, W_r,
                             dtype=torch.cfloat, device=x.device)

        mh, mw = self.modes_h, self.modes_w

        # 3) apply learnable kernel on the TOP (+ky) block
        out_ft[:, :, :mh, :mw] = self._compl_mul2d(
            x_ft[:, :, :mh, :mw], self.weight)

        # 4) enforce Hermitian symmetry  (BOTTOM block <-> conj(TOP))
```

```
        out_ft[:, :, -mh:, :mw] = self._compl_mul2d(
            x_ft[:, :, -mh:, :mw], torch.conj(self.weight))

        # 5) inverse FFT
        x = torch.fft.irfft2(out_ft, s=(H, W), norm='forward')  # real
        return x


# ----------------------------------------------------------------------
#  Full FNO module
# ----------------------------------------------------------------------
class FNO2d(nn.Module):
    """
    Maps (a, f, x1, x2)                -- 4 channels
    to   u_hat  that fulfils Dirichlet boundary  (g on dOmega)

    Boundary mask chi_{dOmega} is used internally but NOT fed to the net.
    """
    def __init__(self,
                 modes:      Tuple[int, int] = (12, 12),
                 width:      int             = 32,
                 depth:      int             = 4):
        super().__init__()

        assert depth > 0, "depth must be >= 1"
        modes_h, modes_w = modes

        self.modes_h = modes_h
        self.modes_w = modes_w
        self.width   = width
        self.depth   = depth

        # 1) lifting layer  (4 -> width)
        self.lifting = nn.Conv2d(4, width, kernel_size=1)

        # 2) stack of spectral + point-wise convolutions
        self.spectral_convs  = nn.ModuleList([
            SpectralConv2d(width, width, modes_h, modes_w)
            for _ in range(depth)
        ])
        self.pointwise_convs = nn.ModuleList([
            nn.Conv2d(width, width, kernel_size=1)
            for _ in range(depth)
        ])

        self.act        = nn.GELU()
        self.projection = nn.Conv2d(width, 1, kernel_size=1)

    # ----------------------------------------------------------------------
    def forward(self,
                zeta: torch.Tensor,
                g:    Optional[torch.Tensor] = None) -> torch.Tensor:
        """
        Parameters
        ----------
        zeta : (B, 4 or 5, H, W)   [a, f, x1, x2, (optional boundary mask)]
        g    : (B, 1, H, W) or None
               prescribed boundary values; default = 0
        """
        if zeta.ndim != 4:
            raise RuntimeError("zeta must have shape (B,C,H,W)")

        # ---------------------------------------- split features/mask
        if zeta.shape[1] == 5:
            phys, mask = zeta[:, :4], zeta[:, 4:5]
```

```
        elif zeta.shape[1] == 4:
            phys = zeta
            # build mask on the fly
            B, _, H, W = zeta.shape
            mask = torch.zeros(B, 1, H, W, device=zeta.device,
                               dtype=zeta.dtype)
            mask[..., 0, :]  = 1.0
            mask[..., -1, :] = 1.0
            mask[..., :, 0]  = 1.0
            mask[..., :, -1] = 1.0
        else:
            raise RuntimeError("zeta must have 4 or 5 channels")

        if g is None:
            g = torch.zeros_like(mask)

        # 1) lifting ---------------------------------------------------
        v = self.lifting(phys)                              # (B,width,H,W)

        one_minus_mask = 1.0 - mask                        # cache & reuse

        # 2) spectral blocks -------------------------------------------
        for K, W_pt in zip(self.spectral_convs,
                           self.pointwise_convs):
            # zero boundary before entering spectral layer
            v = v * one_minus_mask

            y = K(v) + W_pt(v)
            v = self.act(y) + v

        # 3) projection & boundary enforcement -------------------------
        u_raw = self.projection(v)                          # (B,1,H,W)
        u_hat = one_minus_mask * u_raw + mask * g

        return u_hat


# ---------------------------------------------------------------------
#  Convenience helper for coordinate grid & mask  (optional)
# ---------------------------------------------------------------------
@torch.no_grad()
def make_coord_grid(batch: int,
                    nx: int,
                    ny: int,
                    device=None) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Returns
        coords : (1, 2, nx, ny)  (broadcast-able, not per-batch!)
        mask   : (1, 1, nx, ny)
    """
    xs = torch.linspace(0., 1., nx, device=device)
    ys = torch.linspace(0., 1., ny, device=device)
    x1 = xs.view(1, 1, nx, 1)       # (1,1,nx,1)
    x2 = ys.view(1, 1, 1, ny)       # (1,1,1,ny)
    mask = torch.zeros(1, 1, nx, ny, device=device)
    mask[..., 0, :]  = 1.0
    mask[..., -1, :] = 1.0
    mask[..., :, 0]  = 1.0
    mask[..., :, -1] = 1.0
    coords = torch.cat([x1.expand(1, -1, -1, ny),
                        x2.expand(1, -1, nx, -1)], dim=1)    # (1,2,nx,ny)
    return coords, mask


# ---------------------------------------------------------------------
```

```python
#  Quick smoke test
# ----------------------------------------------------------------------
if __name__ == "__main__":
    B, H, W = 2, 85, 85
    dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # synthetic fields a, f
    a = torch.rand(B, 1, H, W, device=dev)
    f = torch.rand(B, 1, H, W, device=dev)

    # coords & mask  (broadcast once!)
    coords, mask = make_coord_grid(1, H, W, dev)
    coords  = coords.repeat(B, 1, 1, 1)
    mask_b  = mask.repeat(B, 1, 1, 1)

    # build input zeta  (5 channels)
    zeta = torch.cat([a, f, coords, mask_b], dim=1)     # (B,5,H,W)

    # instantiate & run the network
    net = FNO2d(modes=(12, 12), width=32, depth=4).to(dev)
    u_pred = net(zeta)                                   # (B,1,H,W)

    # dummy ground truth & loss
    u_true = torch.rand_like(u_pred)
    loss = F.mse_loss((1 - mask_b) * u_pred,
                      (1 - mask_b) * u_true)
    loss.backward()

    print("output:", u_pred.shape, "loss:", loss.item())
```

Listing 1. Refined 2D Fourier Neural Operator for Darcy Flow (Refiner, LLM o3). Addresses the main issues raised in the Critic's review.

D. Theorist Derivation for Darcy Flow (LLM o3)

Below is the complete mathematical derivation produced by the Theorist agent (LLM `o3`) for the 2D Darcy Flow problem. This output was used as input to the Programmer (Appendix C) and subsequently reviewed by the Critic (Appendix B).

D.1. Choice of Architecture

The task is to learn the parameter-to-state map

$$\mathcal{G} : a(x) \mapsto u(x), \quad x \in \Omega := (0,1)^2,$$

where $-\nabla \cdot (a(x)\nabla u(x)) = f(x)$ with $u|_{\partial\Omega} = g(x)$ (Dirichlet).

For 2D Darcy flow this map is (i) translation-equivariant in the interior, (ii) smooth in the spectral domain, and (iii) independent of the discretisation once $a(x)$ and $u(x)$ are understood as functions. The Fourier Neural Operator (FNO) (Li et al., 2020b) was designed exactly for such elliptic or parabolic PDEs and has already been shown to give state-of-the-art accuracy on Darcy flow. Its kernel is global (capturing long-range interactions imposed by the Green's function), it is mesh-independent, and its spectral truncation regularises high-frequency noise, which is useful when the input permeability field is only piecewise smooth. Therefore, a 2D Fourier Neural Operator with a boundary-conditioning layer is adopted.

D.2. Functional Set-up

Let $X := L^\infty(\Omega)$ be the input space for the permeability a, and $Y := H_0^1(\Omega)$ be the output space for the solution u. We seek a parameterised operator

$$\mathcal{G}_\theta : X \to Y, \qquad \theta = \{A_\ell(k),\, W_\ell,\, b_\ell\}_{\ell=0}^{L} \cup \{W_0,\, b_0,\, W_{\text{out}},\, b_{\text{out}}\},$$

that minimises the empirical risk

$$\theta^* = \arg\min_\theta \; \mathcal{L}(\theta), \qquad \mathcal{L}(\theta) = \frac{1}{|\Omega_h|} \sum_{x \in \Omega_h} \|\hat{u}_\theta(x) - u(x)\|^2.$$

D.3. Discrete Grid and Notation

Training grid: $N \times N$ with $N = 85$ (evaluation later possible on $N = 421$). For any grid tensor $v \in \mathbb{C}^{C\times N\times N}$ its 2D discrete Fourier transform is denoted $\hat{v} = \mathcal{F}_N[v]$, with frequencies $k = (k_1, k_2) \in \{0, \dots, N-1\}^2$. The low-frequency index set (kept modes) is

$$\Lambda_K := \{k \in \mathbb{Z}^2 : |k_1| \le K,\ |k_2| \le K\}, \quad K \ll N/2.$$

D.4. Layer-wise Definition of the Fourier Neural Operator

Lifting layer. The raw input channels are the permeability, the source, spatial coordinates, and a boundary mask:

$$\zeta(x) = \big(a(x),\ f(x),\ x_1,\ x_2,\ \chi_{\partial\Omega}(x)\big) \in \mathbb{R}^5.$$

A pointwise affine map lifts ζ to width C_0:

$$v^0(x) = W_0\,\zeta(x) + b_0, \qquad W_0 \in \mathbb{R}^{C_0\times 5},\ b_0 \in \mathbb{R}^{C_0}.$$

Spectral convolution layer ($\ell = 0, \dots, L-1$). Given $v^\ell \in \mathbb{R}^{C_\ell\times N\times N}$:

(a) *Fourier transform:* $\hat{v}^\ell = \mathcal{F}_N[v^\ell] \in \mathbb{C}^{C_\ell\times N\times N}$.

(b) *Spectral kernel application.* For each kept mode $k \in \Lambda_K$:

$$\widehat{(\mathcal{K}_\ell v^\ell)}(k) = A_\ell(k)\cdot \hat{v}^\ell(k), \qquad A_\ell(k) \in \mathbb{C}^{C_{\ell+1}\times C_\ell}.$$

For discarded modes ($k \notin \Lambda_K$) the output is set to zero. Inverse FFT gives the spatial representation $(\mathcal{K}_\ell v^\ell)(x) = \mathcal{F}_N^{-1}[\widehat{\mathcal{K}_\ell v^\ell}](x)$.

(c) *Local linear mixing:* $(\mathcal{W}_\ell v^\ell)(x) = W_\ell v^\ell(x) + b_\ell$, $W_\ell \in \mathbb{R}^{C_{\ell+1}\times C_\ell}$.

(d) *Non-linear update with residual:*

$$v^{\ell+1}(x) = \sigma\big(\mathcal{K}_\ell v^\ell(x) + \mathcal{W}_\ell v^\ell(x)\big) + v^\ell(x), \qquad \sigma = \mathrm{GELU}.$$

Projection to scalar field. After L spectral layers (width C_L), project to Y:

$$\hat{u}_\theta(x) = W_{\mathrm{out}}\,v^L(x) + b_{\mathrm{out}}, \qquad W_{\mathrm{out}} \in \mathbb{R}^{1\times C_L}.$$

Enforcement of Dirichlet boundary. Let $g(x)$ be the prescribed boundary condition. The final prediction is:

$$\hat{u}_\theta(x) = (1-\chi_{\partial\Omega}) \odot \hat{u}_\theta(x)\ +\ \chi_{\partial\Omega} \odot g(x),$$

which is fully differentiable since $\chi_{\partial\Omega}$ is a fixed mask.

D.5. Complete End-to-End Formula

$$\begin{aligned}
v^0(x) &= W_0\,\zeta(x) + b_0,\\
\hat{v}^\ell &= \mathcal{F}_N[v^\ell], \quad \hat{w}^\ell(k) = \mathbf{1}_{\Lambda_K}(k)\cdot A_\ell(k)\,\hat{v}^\ell(k),\\
s^\ell(x) &= \mathcal{F}_N^{-1}[\hat{w}^\ell](x), \quad l^\ell(x) = W_\ell v^\ell(x) + b_\ell,\\
v^{\ell+1}(x) &= \sigma(s^\ell(x) + l^\ell(x)) + v^\ell(x), \quad \ell = 0, \dots, L-1,\\
\hat{u}_\theta(x) &= W_{\mathrm{out}}\,v^L(x) + b_{\mathrm{out}},\\
\hat{u}_\theta(x) &= (1-\chi_{\partial\Omega}(x))\cdot \hat{u}_\theta(x) + \chi_{\partial\Omega}(x)\cdot g(x).
\end{aligned}$$

The trainable parameters are $\theta = \{W_0, b_0, \{A_\ell(k), W_\ell, b_\ell\}_{\ell,k}, W_{\mathrm{out}}, b_{\mathrm{out}}\}$.

D.6. Why the FNO Meets the Mathematical Requirements

1. **Approximation theory.** Spectral neural operators with low-rank Fourier kernels are universal on compact subsets of $L^2(\Omega) \to L^2(\Omega)$ (Li et al., 2020b).
2. **Translation equivariance.** The kernel $\mathcal{K}_\ell$ depends only on k, not on x, mirroring the convolution structure of the Green's operator for $-\nabla \cdot (a\nabla\cdot)$.
3. **Mesh-independence.** The parameters $A_\ell(k)$ act on Fourier coefficients, not grid indices, so the same θ generalises from 85^2 to 421^2.
4. **Efficient computation.** Complexity per layer is $\mathcal{O}(CN^2 \log N + C^2K^2)$ with $K \ll N$.
5. **Boundary handling.** The explicit mask channel plus post-projection enforcement gives rigorous compliance with the Dirichlet boundary.

Hence, the 2D Fourier Neural Operator formulated above is mathematically sound, faithful to the structure of the Darcy flow operator, and ready for a concise PyTorch implementation.

E. Theorist Derivation for Darcy Flow (LLM gpt-5)

Below is the complete mathematical derivation produced by the Theorist agent (LLM `gpt-5`) for the 2D Darcy Flow problem.

Recommended Architecture: Dirichlet-Compliant Sine-Fourier Neural Operator (SFNO)

Rationale.

- **Elliptic smoothing and global coupling.** The Darcy operator $T : (a, f, g) \mapsto u$ on $\Omega = (0, 1)^2$ with Dirichlet boundary is linear and smoothing. Neural operators based on global spectral multipliers approximate such compact operators efficiently.
- **Proven performance.** FNO was introduced and validated on 2D Darcy flow with universal approximation guarantees for continuous operators between Banach spaces.
- **Boundary conditions.** Standard FNO uses periodic Fourier modes, introducing artifacts for Dirichlet problems. Using the Laplacian's Dirichlet eigenbasis $\sin(m\pi x)\sin(n\pi y)$ (discrete sine transform, DST) enforces zero Dirichlet exactly on the residual $r = u - g$, with $r|_{\partial\Omega} = 0$.

Problem Setup. PDE: $-\nabla \cdot (a(x)\nabla u(x)) = f(x)$ in $\Omega = (0, 1)^2$ with $u = g$ on $\partial\Omega$. Assumptions: $a \in L^\infty(\Omega)$ with $0 < a_{\min} \le a(x) \le a_{\max} < \infty$; $f \in L^2(\Omega)$; $g \in H^{1/2}(\partial\Omega)$, so $u \in H^1(\Omega)$ is unique. We learn the residual operator $T_0 : (a, f, g) \mapsto r = u - g$ with $r \in H^1_0(\Omega)$, then reconstruct $u = g + r$.

Discrete Sine Transform (DST). Define orthonormal 1D DST-I matrices $S_x \in \mathbb{R}^{(N_x-2)\times(N_x-2)}$ and $S_y \in \mathbb{R}^{(N_y-2)\times(N_y-2)}$ by

$$(S_x)_{i,m} = \sqrt{\tfrac{2}{N_x-1}}\,\sin\!\Big(\frac{\pi i m}{N_x - 1}\Big), \quad i, m \in \{1, \dots, N_x - 2\}.$$

These satisfy $S_x^\top S_x = I$. The 2D transform of a single-channel tensor $V \in \mathbb{R}^{(N_x-2)\times(N_y-2)}$ is $\hat{V} = S_x^\top V S_y$, with inverse $V = S_x \hat{V} S_y^\top$.

SFNO Architecture. The Theorist approximates T_0 by a composition of lifting, L sine-spectral blocks, and projection. All computations are on the interior.

1. **Lifting.** Construct input on interior: $X^\circ(i, j) = [a(i, j),\, f(i, j),\, g(i, j),\, x_i,\, y_j]^\top$ and lift pointwise: $V_0 = PX^\circ + b_P$, $P \in \mathbb{R}^{C\times d_{\text{in}}}$.
2. **Sine-spectral block** (for $\ell = 0, \dots, L-1$). Compute $\hat{V}_\ell = S_x^\top V_\ell S_y$. For retained modes (m, n) with $1 \le m \le \kappa_x$, $1 \le n \le \kappa_y$, apply a learned mixing matrix $A_\ell(m, n) \in \mathbb{R}^{C\times C}$:

$$\hat{Z}_\ell(m, n) = \begin{cases} A_\ell(m, n)\,\hat{V}_\ell(m, n), & m \le \kappa_x,\, n \le \kappa_y, \\ 0, & \text{otherwise.} \end{cases}$$

   Inverse: $Z_\ell = S_x \hat{Z}_\ell S_y^\top$. Update: $V_{\ell+1}(i, j) = \sigma(Z_\ell(i, j) + W_\ell V_\ell(i, j) + b_\ell)$.

3. **Projection and reconstruction.** Predict $r^\circ(i,j) = q^\top V_L(i,j) + b_q$ and reconstruct $u = g + r^\circ$ on interior, $u = g$ on $\partial\Omega$.

Operator Form Summary. Define the spectral operator $\mathcal{S}(V) = S_x^\top V S_y$, the spectral multiplier $\mathcal{M}_\ell$ (mode-wise $A_\ell(m,n)$, zero elsewhere), and the pointwise map $\mathcal{R}_\ell(V) = W_\ell V + b_\ell$. Each block is

$$\Phi_\ell(V) = \sigma\big(\mathcal{S}^{-1}(\mathcal{M}_\ell \mathcal{S}(V)) + \mathcal{R}_\ell(V)\big).$$

The full SFNO is $r^\circ = \mathcal{Q} \circ \Phi_{L-1} \circ \cdots \circ \Phi_0 \circ \mathcal{P}(X^\circ)$.

Training Objective.

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{sup}}(\theta) + \lambda \mathcal{L}_{\text{PDE}}(\theta), \quad \mathcal{L}_{\text{sup}} = \frac{1}{|\mathcal{B}|} \sum_{\text{samples}} \|u_\theta - u_{\text{ref}}\|^2_{L^2(\Omega)},$$

with optional PDE regularisation $\mathcal{L}_{\text{PDE}}(\theta) = \|\nabla_h \cdot (a \odot \nabla_h u_\theta) - f\|^2_{\ell^2}$ using second-order centred differences.

PyTorch Implementation Notes. Forward DST: `einsum('im,bcij,jn->bcmn', Sx, V, Sy)`. Inverse DST: `einsum('im,bcmn,jn->bcij', Sx, Vhat, Sy)`. Spectral weights stored as tensor of shape $(\kappa_x, \kappa_y, C, C)$. Boundary enforcement: fill boundary with g; interior with $g + r^\circ$.

F. Critic Review for Darcy Flow (LLM gpt-5)

Below is the full structured review generated by the Critic agent (LLM `gpt-5`) for the initial SFNO Darcy Flow implementation.

A. Conceptual / Mathematical Consistency. **Issue A1: Nonhomogeneous Dirichlet handling.** The derivation models the residual $r = u - g$ with $r|_{\partial\Omega} = 0$, but g is only prescribed on $\partial\Omega$; its interior values are not uniquely defined. The code expects a full-field g and reconstructs $u = g + r$ on the interior, yet if the interior extension of g is not consistent, the residual is not unique. **Suggestion.** Either (1) require a specific interior extension $\tilde{g}$ (e.g. harmonic) and document it; or (2) encode g only on the boundary; or (3) learn r for the homogeneous problem by redefining $\tilde{f} = f - \nabla \cdot (a \nabla \tilde{g})$.

Issue A2: "Enforces zero Dirichlet by construction." Intermediate tensors V_ℓ are not constrained to be sine-representable after adding the pointwise skip $W_\ell V_\ell + b_\ell$. Zero Dirichlet only holds for the final assembled u. **Suggestion.** Rephrase to "the final output enforces Dirichlet exactly; intermediate tensors are unconstrained."

Issue A3: Resolution invariance vs. fixed transforms. Weights are resolution-agnostic but the model instance binds a fixed transform. **Suggestion.** Provide a utility `SFNO2D.to_new_resolution(Nx, Ny)` that copies resolution-agnostic weights.

B. DST Definition and Transforms. **Issue B1: Numerical precision.** Repeated dense DSTs accumulate errors in low precision. **Suggestion.** Keep S_x, S_y and DST operations in `float32` even under autocast. Add a unit test: `inverse(forward(V)) ≈ V`.

C. Boundary Assembly and Double Use of g. **Issue C1: Potential data leakage.** If the dataset provides a g that equals u_{ref} inside Ω, the model can trivially learn $r \approx 0$. **Suggestion.** Document that interior g must not equal the true solution. Prefer a fixed, known extension and detach g in reconstruction.

D. Physics Residual Discretisation. **Issue D1: Arithmetic averaging of a at faces.** For high-contrast permeability, arithmetic mean is inaccurate; harmonic mean is standard for diffusion fluxes. **Suggestion.** Use harmonic mean: $a_{i+1/2,j} = 2a_{i+1,j}a_{i,j}/(a_{i+1,j} + a_{i,j})$.

E. Spectral Mixing and Parameterisation. **Issue E1: High parameter count.** Full $C \times C$ mixing per mode yields $\mathcal{O}(\kappa_x \kappa_y C^2)$ parameters per block. For $\kappa_x = \kappa_y = 24$, $C = 64$, $L = 4$ this is ~9.4M spectral weights. **Suggestion.** Use low-rank factorisation $A_\ell(m,n) \approx U_\ell(m,n)V_\ell^\top$ with $r \ll C$, or mode-dependent scalar gates with a shared $C \times C$ matrix.

F. Numerical Stability and Training Hygiene. **Issue F1: Activation and initialization.** GELU can push large activations in early steps with random A. **Suggestion.** Scale spectral output near zero at init, or initialize A near zero. Consider LayerNorm/InstanceNorm after pointwise+spectral sum.

G. Efficiency and Scalability. **Issue G1: Dense DST cost.** For 421×421, matrix-based DST costs $\mathcal{O}(BCH^2W + BCHW^2)$ per block. **Suggestion.** Implement FFT-based DST-I via odd extension and `torch.fft` ($\mathcal{O}(N \log N)$). Gate by size: use dense up to $\sim$192, then FFT-based.

H. Edge Cases and Robustness. **Issue H1: κ_x, κ_y bounds.** Setting κ_x or κ_y to zero or larger than the interior causes degenerate or invalid blocks. **Suggestion.** Assert $1 \leq \kappa_x \leq N_x - 2$ and $1 \leq \kappa_y \leq N_y - 2$.

I. API and Usability. **Issue I1: Cross-resolution usage.** Current instance is tied to (N_x, N_y). **Suggestion.** Provide `SFNO2D.from_state_dict_at_new_resolution(Nx,Ny,state_dict)`.

K. Optional Modelling Improvements. **High-frequency pass-through.** Zeroing all modes above κ produces overly smooth outputs for rough a. Add a learnable per-channel scalar α_ℓ multiplying high-frequency components.

Summary of Highest-Impact Fixes. (1) Standardise and document the interior extension $\tilde{g}$. (2) Switch FD residual to harmonic averaging. (3) Implement FFT-based DST for large grids. (4) Provide resolution-agnostic weight loading. (5) Reduce $A_\ell(m, n)$ parameter count.

G. Operator Code for Darcy Flow (LLM gpt-5)

Below is the complete `SFNO2D` implementation produced by the Programmer and refined by the Refiner (LLM `gpt-5`). The class implements the Dirichlet-compliant Sine-Fourier Neural Operator with harmonic-averaged finite-difference operators and resolution-agnostic weight transfer.

```python
import math
from typing import Optional

import torch
import torch.nn as nn
import torch.nn.functional as F


class SFNO2D(nn.Module):
    """
    Dirichlet-compliant Sine-Fourier Neural Operator for 2D Darcy flow.

    Input X shape: (B, Cin, Nx, Ny)
      - If include_g=True: Cin=3 with channels [a, f, g_tilde]
      - If include_g=False: Cin=2 with channels [a, f], zero Dirichlet BC
    Returns u shape (B, 1, Nx, Ny) with exact Dirichlet boundary.
    """

    def __init__(
        self, Nx, Ny, width=64, L=4, kx=24, ky=24,
        include_g=True, use_log_a=False, a_min=1e-6,
        activation="gelu", normalization=None,
        high_mode_pass_through=True, device=None, dtype=None,
    ):
        super().__init__()
        assert Nx >= 3 and Ny >= 3
        self.Nx, self.Ny = int(Nx), int(Ny)
        self.width, self.L = int(width), int(L)
        self.kx, self.ky = int(kx), int(ky)
        self.include_g = bool(include_g)
        self.use_log_a = bool(use_log_a)
        self.a_min = float(a_min)
        self.act_name = str(activation).lower()
```

```
        self.norm_type = normalization
        self.high_mode_pass_through = bool(high_mode_pass_through)

        H_int, W_int = self.Nx - 2, self.Ny - 2
        assert 1 <= self.kx <= H_int and 1 <= self.ky <= W_int

        Sx = self._build_dst_i_matrix(self.Nx, device=device, dtype=torch.float32)
        Sy = self._build_dst_i_matrix(self.Ny, device=device, dtype=torch.float32)
        self.register_buffer("Sx", Sx, persistent=False)
        self.register_buffer("Sy", Sy, persistent=False)

        x_full = torch.linspace(0.0, 1.0, self.Nx, device=device, dtype=dtype
                                )[None, None, :, None]
        y_full = torch.linspace(0.0, 1.0, self.Ny, device=device, dtype=dtype
                                )[None, None, None, :]
        self.register_buffer("x_full", x_full, persistent=False)
        self.register_buffer("y_full", y_full, persistent=False)

        self.lift = nn.Conv2d(4, self.width, kernel_size=1, bias=True)

        self.gamma = nn.ParameterList([
            nn.Parameter(torch.zeros(self.kx, self.ky, self.width))
            for _ in range(self.L)
        ])
        self.A_shared = nn.ParameterList([
            nn.Parameter(torch.zeros(self.width, self.width))
            for _ in range(self.L)
        ])
        if self.high_mode_pass_through:
            self.alpha_high = nn.ParameterList([
                nn.Parameter(torch.zeros(self.width)) for _ in range(self.L)
            ])
        else:
            self.alpha_high = None

        self.pointwise = nn.ModuleList([
            nn.Conv2d(self.width, self.width, kernel_size=1, bias=True)
            for _ in range(self.L)
        ])
        for pw in self.pointwise:
            with torch.no_grad():
                pw.weight.zero_(); pw.bias.zero_()
                eye = torch.eye(self.width, device=pw.weight.device,
                                dtype=pw.weight.dtype)
                pw.weight[:, :, 0, 0] += eye

        if self.norm_type == "instancenorm":
            self.norms = nn.ModuleList([
                nn.InstanceNorm2d(self.width, affine=True, eps=1e-5)
                for _ in range(self.L)
            ])
        else:
            self.norms = None

        if self.act_name == "gelu":
            self.act = nn.GELU()
        elif self.act_name == "relu":
            self.act = nn.ReLU(inplace=True)
        else:
            raise ValueError(f"Unsupported activation: {activation}")

        self.proj = nn.Conv2d(self.width, 1, kernel_size=1, bias=True)

    @torch.no_grad()
    def to_new_resolution(self, Nx_new, Ny_new):
```

```
        model_new = SFNO2D(
            Nx=Nx_new, Ny=Ny_new, width=self.width, L=self.L,
            kx=self.kx, ky=self.ky, include_g=self.include_g,
            use_log_a=self.use_log_a, a_min=self.a_min,
            activation=self.act_name, normalization=self.norm_type,
            high_mode_pass_through=self.high_mode_pass_through,
            device=self.Sx.device, dtype=self.x_full.dtype,
        ).to(self.Sx.device)
        model_new.lift.load_state_dict(self.lift.state_dict())
        for i in range(self.L):
            model_new.gamma[i].data.copy_(self.gamma[i].data)
            model_new.A_shared[i].data.copy_(self.A_shared[i].data)
            model_new.pointwise[i].load_state_dict(self.pointwise[i].state_dict())
            if self.norms is not None:
                model_new.norms[i].load_state_dict(self.norms[i].state_dict())
            if self.alpha_high is not None:
                model_new.alpha_high[i].data.copy_(self.alpha_high[i].data)
        model_new.proj.load_state_dict(self.proj.state_dict())
        return model_new

    def forward(self, X):
        B, Cin, H, W = X.shape
        assert H == self.Nx and W == self.Ny
        if self.include_g:
            assert Cin == 3
            a_full, f_full, g_full = X[:, 0:1], X[:, 1:2], X[:, 2:3]
        else:
            assert Cin == 2
            a_full, f_full = X[:, 0:1], X[:, 1:2]
            g_full = torch.zeros(B, 1, H, W, device=X.device, dtype=X.dtype)

        if self.use_log_a:
            a_full = torch.log(torch.clamp(a_full, min=self.a_min))

        x_int = self.x_full[:, :, 1:-1, :].expand(B, 1, H-2, 1).expand(B,1,H-2,W-2)
        y_int = self.y_full[:, :, :, 1:-1].expand(B, 1, 1, W-2).expand(B,1,H-2,W-2)
        a_int = a_full[:, :, 1:-1, 1:-1]

        if self.include_g:
            with torch.no_grad():
                a_for_fd = torch.clamp(a_full, min=self.a_min)
                div_ag = self._div_a_grad(a_for_fd, g_full.detach())
                f_tilde_int = f_full[:, :, 1:-1, 1:-1] - div_ag
        else:
            f_tilde_int = f_full[:, :, 1:-1, 1:-1]

        X_int = torch.cat([a_int, f_tilde_int, x_int, y_int], dim=1)
        V = self.lift(X_int)

        for l in range(self.L):
            V = self._spectral_block(V, l)

        r_int = self.proj(V)
        u = g_full.detach().clone()
        u[:, :, 1:-1, 1:-1] = u[:, :, 1:-1, 1:-1] + r_int
        return u

    def _spectral_block(self, V, l):
        B, C, M, N = V.shape
        Vhat = self._dst2_forward(V)
        kx, ky = self.kx, self.ky
        Zhat = torch.zeros_like(Vhat)

        Vhat_low = Vhat[:, :, :kx, :ky]
        gamma_bc = self.gamma[l].permute(2, 0, 1).unsqueeze(0)
```

```
        Vhat_low_scaled = Vhat_low * gamma_bc
        Zhat_low = torch.einsum('oc,bcmn->bomn', self.A_shared[l], Vhat_low_scaled)
        Zhat[:, :, :kx, :ky] = Zhat_low

        if self.alpha_high is not None:
            alpha = self.alpha_high[l].view(1, C, 1, 1)
            if kx < M:
                Zhat[:, :, kx:, :] = Zhat[:, :, kx:, :] + alpha * Vhat[:, :, kx:, :]
            if ky < N:
                Zhat[:, :, :kx, ky:] = Zhat[:, :, :kx, ky:] + alpha*Vhat[:,:,:kx,ky:]

        Z = self._dst2_inverse(Zhat)
        out = Z + self.pointwise[l](V)
        if self.norms is not None:
            out = self.norms[l](out)
        return self.act(out)

    @staticmethod
    def _build_dst_i_matrix(n_total, device=None, dtype=None):
        n_int = n_total - 2
        i = torch.arange(1, n_total-1, device=device, dtype=dtype)[:, None]
        m = torch.arange(1, n_total-1, device=device, dtype=dtype)[None, :]
        return torch.sin(math.pi * i * m / (n_total-1)) * math.sqrt(2.0/(n_total-1))

    def _dst2_forward(self, V):
        B, C, M, N = V.shape
        Vf = V.to(torch.float32)
        X = Vf.reshape(B*C, M, N).contiguous()
        X = torch.matmul(self.Sx.transpose(0, 1), X)
        X = torch.matmul(X, self.Sy)
        return X.reshape(B, C, M, N).to(V.dtype)

    def _dst2_inverse(self, Vhat):
        B, C, M, N = Vhat.shape
        X = Vhat.to(torch.float32).reshape(B*C, M, N).contiguous()
        X = torch.matmul(self.Sx, X)
        X = torch.matmul(X, self.Sy.transpose(0, 1))
        return X.reshape(B, C, M, N).to(Vhat.dtype)

    @staticmethod
    def _harmonic_mean(a, b, eps):
        return 2.0 * a * b / (a + b + eps)

    def _div_a_grad(self, a_full, u_full):
        B, _, Nx, Ny = a_full.shape
        hx, hy = 1.0/(Nx-1), 1.0/(Ny-1)
        eps = self.a_min
        a_ip = self._harmonic_mean(a_full[:,:,2:,1:-1],  a_full[:,:,1:-1,1:-1], eps)
        a_im = self._harmonic_mean(a_full[:,:,1:-1,1:-1], a_full[:,:,:-2,1:-1],  eps)
        a_jp = self._harmonic_mean(a_full[:,:,1:-1,2:],   a_full[:,:,1:-1,1:-1], eps)
        a_jm = self._harmonic_mean(a_full[:,:,1:-1,1:-1], a_full[:,:,1:-1,:-2],  eps)
        du_ip = u_full[:,:,2:,1:-1]  - u_full[:,:,1:-1,1:-1]
        du_im = u_full[:,:,1:-1,1:-1]- u_full[:,:,:-2,1:-1]
        du_jp = u_full[:,:,1:-1,2:]  - u_full[:,:,1:-1,1:-1]
        du_jm = u_full[:,:,1:-1,1:-1]- u_full[:,:,1:-1,:-2]
        div_x = (a_ip*du_ip - a_im*du_im) / (hx*hx)
        div_y = (a_jp*du_jp - a_jm*du_jm) / (hy*hy)
        return div_x + div_y
```

Listing 2. SFNO2D – Sine-Fourier Neural Operator for 2D Darcy Flow (gpt-5).

H. Theorist Derivation for Darcy Flow (LLM gpt-5-mini)

Below is the complete mathematical derivation produced by the Theorist agent (LLM `gpt-5-mini`) for the 2D Darcy Flow problem.

Recommended Architecture: Dirichlet-FNO

The Theorist recommends a Fourier Neural Operator adapted for Dirichlet boundary conditions (Dirichlet-FNO). The map $(a(x), f(x)) \mapsto u(x)$ for elliptic 2D Darcy flow is a smoothing, global operator (inverse of an elliptic differential operator). FNO directly parameterises global convolutional kernels in the Fourier domain, which is well aligned with approximating integral kernels/Green's functions for elliptic problems. A boundary projection layer enforces Dirichlet conditions exactly.

Functional Setting. Let $X = L^2(D; \mathbb{R}^{c_{\text{in}}})$ be the input space (channels include $a(x)$, $f(x)$, and optional coordinates/boundary indicator) and $Y = L^2(D; \mathbb{R})$ the scalar solution space. The target operator $G : X \to Y$ is continuous and smoothing.

Lifting. A pointwise linear map $P : \mathbb{R}^{c_{\text{in}}} \to \mathbb{R}^d$ applied at each grid point:

$$v^{(0)}(x) = P\, v_0(x), \quad P \in \mathbb{R}^{d \times c_{\text{in}}}.$$

Spectral Convolutional Layer. Each layer $\ell = 0, \ldots, L-1$ maps $v^{(\ell)} : D \to \mathbb{R}^d$ to $v^{(\ell+1)} : D \to \mathbb{R}^d$ via

$$v^{(\ell+1)}(x) = \sigma\Big(W^{(\ell)} v^{(\ell)}(x) + \mathcal{F}^{-1}\big[R^{(\ell)}(\cdot)\, \hat{v}^{(\ell)}(\cdot)\big](x) + b^{(\ell)}\Big),$$

where $W^{(\ell)} \in \mathbb{R}^{d \times d}$ is pointwise, $R^{(\ell)}(k) \in \mathbb{C}^{d \times d}$ are learned frequency weights (applied only to retained modes $k \in \mathcal{K}$, zero elsewhere), and $\hat{v}^{(\ell)} = \mathcal{F}[v^{(\ell)}]$.

Projection and Dirichlet Enforcement. After L layers, project to a scalar: $u_{\text{pred}}(x) = Q\, v^{(L)}(x)$. Enforce Dirichlet:

$$u_\theta(x) = (1 - M_{\partial D}(x))\, u_{\text{pred}}(x) + M_{\partial D}(x)\, g(x),$$

where $M_{\partial D}(x) = 1$ on ∂D and 0 elsewhere.

Remarks on Non-Periodicity. Standard FFT assumes periodicity. To mitigate periodic artifacts the Theorist recommends: (a) append coordinate channels (x_1, x_2) to input v_0 and use modest mode count $K \ll N$; or (b) work in the sine basis (DST) for homogeneous Dirichlet. The Dirichlet-FNO implementation uses complex FFT with boundary projection and coordinate embedding.

Loss. Interior MSE: $\mathcal{L}(\theta) = (1/|I|) \sum_{(i,j) \in \text{interior}} |u_\theta(i,j) - u_{\text{true}}(i,j)|^2$.

Recommended Hyperparameters. Grid $N = 85$; input channels $c_{\text{in}} = 4$ (include coordinates); hidden $d = 32$; $L = 4$ Fourier layers; $\text{modes}_x = \text{modes}_y = 16$ ($\leq N/2$); Adam with lr $10^{-3} \to 10^{-4}$.

I. Critic Review for Darcy Flow (LLM gpt-5-mini)

Below is the full structured review generated by the Critic agent (LLM `gpt-5-mini`).

Summary. The math is largely consistent with standard FNO practice; the Dirichlet projection is a practical way to enforce boundary values. Main concerns: subtle rFFT indexing / Hermitian symmetry issues, computational cost and parameter explosion of per-mode full $d \times d$ complex matrices, API/shape edge cases, and numerical/training-stability recommendations.

A. Mathematical / Modelling Issues. **Issue 1: FFT vs. Dirichlet conceptual mismatch.** rFFT/FFT layers assume periodic handling; boundary projection fixes values on the discrete boundary but does not remove periodic coupling inside

the network (Gibbs-like artifacts near $\partial\Omega$). **Suggestion.** Prefer DST spectral layers for Dirichlet; for nonzero g, precompute extension u_g and learn $w = u - u_g$ with homogeneous BCs.

Issue 2: Exact enforcement vs. continuum consistency. Projecting to the discrete boundary does not guarantee flux continuity near $\partial\Omega$. **Suggestion.** Include a small H^1 penalty or use the $u = u_g + w$ decomposition.

B. Numerical / Training Stability. **Issue 3: No normalisation / skip connections.** Deep FNO stacks can suffer from training instability without LayerNorm or residual connections. **Suggestion.** Add residual update $v \leftarrow v + \sigma(W(v) + w)$ or GroupNorm after each layer.

Issue 4: Large parameter count. Full $R^{(\ell)}(k) \in \mathbb{C}^{d\times d}$ per retained mode; for 16×16 modes and $d = 32$: ~524k real parameters per layer. **Suggestion.** Use diagonal or low-rank factorisation $R(k) = A(k)B(k)$ with $r \ll d$.

Issue 5: Initialisation scale. Scale $\sim 1/\sqrt{d^2}$ may be too small; no normalisation exists in model. **Suggestion.** Xavier/He init adapted to spectral layers; add small weight decay.

C. Implementation Correctness and Edge Cases. **Issue 6: rFFT indexing and mode limits.** For `rfft2`, output width is $N//2 + 1$; `modes_y` $\leq N//2 + 1$. Comment inconsistencies may cause user mistakes. **Suggestion.** Assert `modes_x` $\leq N$ and `modes_y` $\leq N//2 + 1$ with clear error messages.

Issue 7: Hermitian symmetry. Independently setting complex weights for $+k_x$ and $-k_x$ slices without conjugate-symmetry enforcement can introduce unexpected structure. **Suggestion.** Parameterise purely real transforms or enforce conjugate relations.

Issue 8: `torch.complex` in forward. Constructing `torch.complex(w_re, w_im)` each step allocates intermediate tensors. **Suggestion.** Implement complex multiply via real arithmetic: $(a+ib)(c+id) = (ac - bd) + i(ad + bc)$.

Issue 9: Inefficient einsum per mode. $\mathcal{O}(B \cdot m_x m_y d^2)$ per layer; creates large intermediates. **Suggestion.** Low-rank factorisation or batched `torch.matmul` with careful reshaping.

Issue 10: dtype/device pitfalls. `boundary_mask_from_N` with `device=None` returns a CPU tensor; may cause device/dtype mismatch. **Suggestion.** Default to `device=x.device` and cast mask to `u_pred.dtype` inside forward.

Issue 11: Dirichlet via arithmetic vs. in-place. Keep mask-based arithmetic (already implemented). Explicitly detach g if it should not receive gradients.

Issue 12: Interior-only loss not enforced. Training with full MSE including boundary gives trivially zero boundary loss. **Suggestion.** Provide a helper function for interior-only loss.

D. FFT Normalisation. **Issue 13: Convention consistency.** `norm=``ortho''` is consistent internally but may differ from data-generation code. **Suggestion.** Document and fix FFT normalisation throughout.

E. Efficiency and Alternative Parameterisations. **Issue 19–21.** Recommend low-rank / separable / diagonal spectral weights; real-only arithmetic; mixed-precision paths with `autocast`.

Prioritised Fixes.

1. Add interior-only loss helper; exclude boundaries from training loss.
2. Add LayerNorm/residual connections for stability.
3. Replace full $d \times d$ spectral weights with low-rank or diagonal version.
4. Replace `torch.complex` with explicit real/imag arithmetic.
5. Add unit tests for boundary enforcement and mode index sanity.

J. Operator Code for Darcy Flow (LLM gpt-5-mini)

Below is the complete `DirichletFNO2d` implementation produced by the Programmer and refined by the Refiner (LLM `gpt-5-mini`).

```
import torch
```

```python
import torch.nn as nn
import torch.nn.functional as F

class DirichletFNO2d(nn.Module):
    """
    Improved Dirichlet-FNO for 2D problems on an N x N grid.

    Key improvements:
    - Low-rank complex spectral parameterization (reduces params & compute).
    - Real/imag arithmetic (avoids constructing complex nn.Parameters).
    - Residual connections with learnable per-layer scale and GroupNorm.
    - Robust boundary-mask handling and helper for interior-only loss.
    - Careful parameter initialization (Xavier real, small imag).

    Forward: forward(x, g, boundary_mask=None)
      x: (B, c_in, N, N)   g: (B, 1, N, N) Dirichlet values
    Returns u: (B, 1, N, N) with exact Dirichlet boundary.
    """

    def __init__(self, modes_x, modes_y, width, depth, c_in, rank=8,
                 activation="gelu", fft_norm="ortho"):
        super().__init__()
        assert modes_x > 0 and modes_y > 0
        self.modes_x, self.modes_y = modes_x, modes_y
        self.width, self.depth, self.c_in = width, depth, c_in
        self.rank, self.fft_norm = rank, fft_norm

        self.lift = nn.Conv2d(c_in, width, kernel_size=1)
        self.w_convs = nn.ModuleList([
            nn.Conv2d(width, width, kernel_size=1, bias=True)
            for _ in range(depth)
        ])
        self.gns = nn.ModuleList([nn.GroupNorm(1, width) for _ in range(depth)])

        self.U_pos_re = nn.ParameterList(); self.U_pos_im = nn.ParameterList()
        self.V_pos_re = nn.ParameterList(); self.V_pos_im = nn.ParameterList()
        self.U_neg_re = nn.ParameterList(); self.U_neg_im = nn.ParameterList()
        self.V_neg_re = nn.ParameterList(); self.V_neg_im = nn.ParameterList()
        self.layer_scale = nn.ParameterList()

        for _ in range(depth):
            mx, my, d, r = self.modes_x, self.modes_y, self.width, self.rank
            Upr = nn.Parameter(torch.empty(mx, my, d, r))
            Upi = nn.Parameter(torch.empty(mx, my, d, r))
            Vpr = nn.Parameter(torch.empty(mx, my, r, d))
            Vpi = nn.Parameter(torch.empty(mx, my, r, d))
            Unr = nn.Parameter(torch.empty(mx, my, d, r))
            Uni = nn.Parameter(torch.empty(mx, my, d, r))
            Vnr = nn.Parameter(torch.empty(mx, my, r, d))
            Vni = nn.Parameter(torch.empty(mx, my, r, d))
            for tensor in (Upr, Vpr, Unr, Vnr):
                nn.init.xavier_uniform_(tensor,
                                        gain=nn.init.calculate_gain('linear'))
            with torch.no_grad():
                Upi.zero_(); Vpi.zero_(); Uni.zero_(); Vni.zero_()
            self.U_pos_re.append(Upr); self.U_pos_im.append(Upi)
            self.V_pos_re.append(Vpr); self.V_pos_im.append(Vpi)
            self.U_neg_re.append(Unr); self.U_neg_im.append(Uni)
            self.V_neg_re.append(Vnr); self.V_neg_im.append(Vni)
            self.layer_scale.append(
                nn.Parameter(torch.tensor(0.1, dtype=torch.float32)))

        if activation.lower() == "gelu":
            self.act = nn.GELU()
        elif activation.lower() == "relu":
```

```
            self.act = nn.ReLU()
        else:
            raise ValueError("Unsupported activation")
        self.project = nn.Conv2d(width, 1, kernel_size=1)

    @staticmethod
    def boundary_mask_from_N(N, device=None, dtype=torch.float32):
        if device is None:
            device = torch.device('cpu')
        mask = torch.zeros((N, N), dtype=dtype, device=device)
        mask[0, :] = 1.0; mask[-1, :] = 1.0
        mask[:, 0] = 1.0; mask[:, -1] = 1.0
        return mask.unsqueeze(0)  # (1, N, N)

    def _check_modes_against_N(self, N):
        Ny = N // 2 + 1
        mx, my = self.modes_x, self.modes_y
        if not (0 < mx <= N):
            raise ValueError(f"modes_x must satisfy 0 < modes_x <= N. "
                             f"Got modes_x={mx}, N={N}.")
        if not (0 < my <= Ny):
            raise ValueError(f"modes_y must satisfy 0 < modes_y <= N//2+1. "
                             f"Got modes_y={my}, Ny={Ny}.")
        if mx > N // 2:
            import warnings
            warnings.warn("modes_x > N//2 may include high-frequency modes.")

    def spectral_conv(self, x, layer_idx):
        B, d, N, _ = x.shape
        self._check_modes_against_N(N)
        x_ft = torch.fft.rfft2(x, norm=self.fft_norm)
        mx, my = self.modes_x, self.modes_y
        out_ft = torch.zeros_like(x_ft)

        def apply_lowrank_slice(v_ft_slice, U_re, U_im, V_re, V_im):
            v = v_ft_slice.permute(0, 2, 3, 1)
            v_re, v_im = v.real, v.imag
            s_re = (torch.einsum('bxyi,xyri->bxyr', v_re, V_re)
                    - torch.einsum('bxyi,xyri->bxyr', v_im, V_im))
            s_im = (torch.einsum('bxyi,xyri->bxyr', v_re, V_im)
                    + torch.einsum('bxyi,xyri->bxyr', v_im, V_re))
            out_re = (torch.einsum('bxyr,xyor->bxyo', s_re, U_re)
                      - torch.einsum('bxyr,xyor->bxyo', s_im, U_im))
            out_im = (torch.einsum('bxyr,xyor->bxyo', s_re, U_im)
                      + torch.einsum('bxyr,xyor->bxyo', s_im, U_re))
            return torch.complex(out_re.permute(0,3,1,2),
                                 out_im.permute(0,3,1,2))

        if mx > 0 and my > 0:
            out_ft[:,:,:mx,:my] = apply_lowrank_slice(
                x_ft[:,:,:mx,:my],
                self.U_pos_re[layer_idx], self.U_pos_im[layer_idx],
                self.V_pos_re[layer_idx], self.V_pos_im[layer_idx])
            out_ft[:,:,-mx:,:my] = apply_lowrank_slice(
                x_ft[:,:,-mx:,:my],
                self.U_neg_re[layer_idx], self.U_neg_im[layer_idx],
                self.V_neg_re[layer_idx], self.V_neg_im[layer_idx])

        return torch.fft.irfft2(out_ft, s=(N, N), norm=self.fft_norm)

    def forward(self, x, g, boundary_mask=None):
        assert x.ndim == 4
        B, c_in, N, N2 = x.shape
        assert N == N2
        if c_in != self.c_in:
```

```
            raise ValueError(f"Expected {self.c_in} channels, got {c_in}.")

        if boundary_mask is None:
            M = self.boundary_mask_from_N(N, device=x.device, dtype=x.dtype)
            M = M.unsqueeze(1)
            if B > 1:
                M = M.repeat(B, 1, 1, 1)
        else:
            bm = boundary_mask
            if bm.ndim == 2:
                bm = bm.unsqueeze(0)
            if bm.ndim == 3:
                if bm.shape[0] == 1:
                    bm = bm.unsqueeze(1)
                    if B > 1:
                        bm = bm.repeat(B, 1, 1, 1)
                elif bm.shape[0] == B:
                    bm = bm.unsqueeze(1)
                else:
                    raise ValueError("boundary_mask shape not understood")
            M = bm.to(device=x.device, dtype=x.dtype)

        if g is None:
            raise ValueError("g must be provided.")
        if g.shape != (B, 1, N, N):
            raise ValueError(f"g must have shape (B,1,N,N). Got {tuple(g.shape)}.")

        v = self.lift(x)
        for l in range(self.depth):
            w = self.spectral_conv(v, l)
            v_point = self.w_convs[l](v)
            activated = self.act(v_point + w)
            normalized = self.gns[l](activated)
            v = v + self.layer_scale[l] * normalized

        u_pred = self.project(v)
        return (1.0 - M) * u_pred + M * g

    @staticmethod
    def interior_loss(u_pred, u_true, boundary_mask):
        if boundary_mask.ndim == 2:
            M = boundary_mask.unsqueeze(0).unsqueeze(1)
        elif boundary_mask.ndim == 3:
            M = boundary_mask.unsqueeze(1)
        elif boundary_mask.ndim == 4:
            M = boundary_mask
        else:
            raise ValueError("boundary_mask must have ndim 2, 3, or 4")
        if M.shape[0] == 1 and u_pred.shape[0] > 1:
            M = M.repeat(u_pred.shape[0], 1, 1, 1)
        interior = 1.0 - M
        diff = (u_pred - u_true) * interior
        denom = interior.sum()
        if denom == 0:
            return torch.tensor(0.0, device=u_pred.device, dtype=u_pred.dtype)
        return (diff ** 2).sum() / denom


if __name__ == "__main__":
    B, N, c_in, modes, width, depth, rank = 2, 64, 4, 12, 32, 3, 6
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = DirichletFNO2d(modes_x=modes, modes_y=modes, width=width,
                           depth=depth, c_in=c_in, rank=rank).to(device)
    x = torch.randn(B, c_in, N, N, device=device)
    g = torch.randn(B, 1, N, N, device=device)
```

```
    bd_mask = DirichletFNO2d.boundary_mask_from_N(N, device=device)
    u = model(x, g, boundary_mask=bd_mask)
    print("u shape:", u.shape)
    bd = bd_mask.unsqueeze(1).repeat(B, 1, 1, 1).to(device)
    diff_bd = ((u - g) * bd).abs().max()
    print("max boundary mismatch (should be ~0):", diff_bd.item())
    u_true = torch.randn_like(u)
    loss = DirichletFNO2d.interior_loss(u, u_true, bd_mask)
    loss.backward()
    print("backward ok, loss:", loss.item())
```

Listing 3. DirichletFNO2d – improved Dirichlet-FNO for 2D Darcy Flow (gpt-5-mini).

K. Theorist Derivation for Darcy Flow (LLM o1)

Below is the complete mathematical derivation produced by the Theorist agent (LLM `o1`) for the 2D Darcy Flow problem.

Recommended Architecture: Fourier Neural Operator (FNO)

The Theorist recommends a standard Fourier Neural Operator. The goal is to learn $\mathcal{G} : a(x) \mapsto u(x)$ where $-\nabla \cdot (a(x)\nabla u(x)) = f(x)$ with Dirichlet BCs.

Why FNO?

1. Darcy flow is an elliptic PDE with nonlocal solution dependency on $a(x)$. FNO captures long-range dependencies efficiently via frequency-domain representations.
2. The FNO was demonstrated on Darcy flow benchmarks with state-of-the-art performance (Li et al., 2020b).
3. FNO has a known connection to operator approximation theory, leveraging nonlocal convolutions in Fourier space.

Input Encoding. Encode $a(x)$, $f(x)$, and optionally coordinate channels into $v_0(x) = P_{\text{in}}([a(x), f(x), \ldots])$, where P_{in} lifts from $\mathbb{R}^{D_{\text{in}}}$ to $\mathbb{R}^d$.

Iterative Fourier Layers.

$$v^{(\ell+1)}(x) = \sigma\big(W_\ell\, v^{(\ell)}(x) + \mathcal{F}^{-1}\big[R_\ell(\xi) \cdot \mathcal{F}[v^{(\ell)}](\xi)\big](x)\big),$$

where W_ℓ is a pointwise 1×1 convolution, $R_\ell(\xi) \in \mathbb{C}^{d\times d}$ are learned frequency weights (applied to the K lowest modes, zero elsewhere), and σ is a nonlinear activation.

Output Projection. $\hat{u}(x) = P_{\text{out}}(v^{(L)}(x))$, $P_{\text{out}} : \mathbb{R}^d \to \mathbb{R}$.

Dirichlet Boundary Conditions. Since training data respects the BCs, the FNO learns to approximate the correct boundary. Optionally, after the forward pass, explicitly overwrite boundary predictions with the known Dirichlet values.

Implementation Sketch.

1. Define spectral weights $R_\ell(\xi)$ per layer (learned, stored as `cfloat` tensors).
2. Define pointwise maps W_ℓ as 1×1 convolutions.
3. Forward: $v_0 = P_{\text{in}}(a)$; for each layer: FFT $\to$ truncate $\to$ multiply by $R_\ell \to$ iFFT; add $W_\ell(v_\ell)$; apply σ.
4. $\hat{u} = P_{\text{out}}(v^{(L)})$.
5. Optionally clamp $\hat{u}$ on $\partial\Omega$ to Dirichlet values.

L. Operator Code for Darcy Flow (LLM o1)

Below is the complete `FNO2dDarcy` implementation produced by the Programmer and refined by the Refiner (LLM `o1`).

```
import torch
```

```python
import torch.nn as nn
import torch.nn.functional as F

class FNO2dDarcy(nn.Module):
    """
    A self-contained Fourier Neural Operator (FNO) for 2D Darcy flow.
    Includes:
      - Multiple input channels (e.g. a(x), f(x), boundary mask).
      - Spectral convolution using rFFT retaining both low and negative
        frequencies via Hermitian symmetry.
      - Dirichlet boundary enforcement in the forward pass via a
        boundary mask and boundary values.

    Example:
        model = FNO2dDarcy(in_channels=2, out_channels=1,
                           hidden_channels=32, modes1=12, modes2=12,
                           num_layers=4, activation='gelu')
        y_pred = model(x, boundary_mask=mask, boundary_values=vals)
    """
    def __init__(self, in_channels, out_channels, hidden_channels=64,
                 modes1=12, modes2=12, num_layers=4, activation='gelu'):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.hidden_channels = hidden_channels
        self.modes1 = modes1
        self.modes2 = modes2
        self.num_layers = num_layers

        if activation.lower() == 'gelu':
            self.activation_fn = F.gelu
        elif activation.lower() == 'relu':
            self.activation_fn = F.relu
        else:
            raise ValueError(f"Unsupported activation: {activation}")

        self.lifting = nn.Conv2d(in_channels, hidden_channels, kernel_size=1)

        self.spectral_convs = nn.ModuleList([])
        self.w_convs = nn.ModuleList([])
        for _ in range(num_layers):
            self.spectral_convs.append(
                self.SpectralConv2d(hidden_channels, hidden_channels,
                                    modes1, modes2)
            )
            self.w_convs.append(
                nn.Conv2d(hidden_channels, hidden_channels, kernel_size=1)
            )

        self.projection = nn.Conv2d(hidden_channels, out_channels, kernel_size=1)

    def forward(self, x, boundary_mask=None, boundary_values=None):
        """
        x: (B, in_channels, H, W)
        boundary_mask: (B, out_channels, H, W) optional; 1 on boundary, 0 interior
        boundary_values: (B, out_channels, H, W) values to impose at boundary
        Returns: (B, out_channels, H, W)
        """
        x = self.lifting(x)
        for spec_conv, w_conv in zip(self.spectral_convs, self.w_convs):
            x1 = spec_conv(x)
            x2 = w_conv(x)
            x = self.activation_fn(x1 + x2)
        x = self.projection(x)
        if boundary_mask is not None and boundary_values is not None:
```

```
            x = (1.0 - boundary_mask) * x + boundary_mask * boundary_values
        return x

    class SpectralConv2d(nn.Module):
        """
        Spectral convolution using rFFT/irFFT.
        Retains the first modes1 (height) and modes2 (width) complex modes.
        Input/Output shape: (Batch, Channels, Height, Width).
        """
        def __init__(self, in_channels, out_channels, modes1, modes2):
            super().__init__()
            self.in_channels = in_channels
            self.out_channels = out_channels
            self.modes1 = modes1
            self.modes2 = modes2
            self.weights = nn.Parameter(
                torch.randn(out_channels, in_channels, modes1, modes2,
                            dtype=torch.cfloat) * 0.02
            )

        def forward(self, x):
            B, _, H, W = x.shape
            assert self.modes1 <= H, (
                f"modes1={self.modes1} cannot exceed input height={H}.")
            assert self.modes2 <= (W // 2 + 1), (
                f"modes2={self.modes2} cannot exceed W//2+1={W//2+1}.")
            x_ft = torch.fft.rfft2(x, norm='ortho')
            out_ft = torch.zeros_like(x_ft)
            x_ft_trunc = x_ft[:, :, :self.modes1, :self.modes2]
            out_ft_trunc = torch.einsum("bihw, oihw -> bohw",
                                        x_ft_trunc, self.weights)
            out_ft[:, :, :self.modes1, :self.modes2] = out_ft_trunc
            x_out = torch.fft.irfft2(out_ft, s=(H, W), norm='ortho')

            return x_out
```

Listing 4. FNO2dDarcy – 2D Fourier Neural Operator for Darcy Flow (o1).

M. Critic Review for Darcy Flow (LLM o3)

Below is the full structured review generated by the Critic agent (LLM `o3`) for the initial FNO Darcy Flow operator implementation.

A. Mathematical / Modelling Assumptions. Issue 1 (!): "Mesh-independence" vs. FFT scaling. Using `norm='ortho'` makes each Fourier basis vector L^2-orthonormal with amplitude depending on $\sqrt{HW}$. Training on 85×85 and evaluating on 421×421 scales spectral coefficients by $\approx 5\times$, causing under-estimation of the solution at finer grids. **Suggestion.** Either train with `norm='forward'` ("divide by HW"), or re-scale $A_\ell(k)$ on the fly by $\sqrt{H_{\text{test}}W_{\text{test}}/H_{\text{train}}W_{\text{train}}}$.

Issue 2 (!): PDE is non-periodic, spectral kernel is. The mask channel tells the net where the boundary is but does not remove Gibbs artefacts from the periodic extension. **Suggestion.** Pad the domain with a smooth extension layer (as in Li et al. 2021), or blend Fourier and local convolutions near $\partial\Omega$.

Issue 3 (△): Target space is H_0^1 but loss uses pointwise L^2. Minimising nodal MSE does not control the gradient (energy norm). **Suggestion.** Add a Sobolev term $\|\nabla\hat{u} - \nabla u\|^2$ or a physics loss $\| - \nabla \cdot (a\nabla\hat{u}) - f\|^2$.

Issue 4 (△): Constant boundary mask as input feature. If every sample shares the same $\chi_{\partial\Omega}$, the network wastes one channel on a constant. **Suggestion.** Drop it from ζ; keep it only for enforcement.

B. Spectral Layer Implementation. Issue 5 (!): Hermitian symmetry. Code writes values only into rows $0 \ldots \text{modes}-1$; all other rows are zero. This restricts the network to an asymmetric pass-band (upper half-plane only). **Suggestion.** Learn

two quadrants (top-left and bottom-left), or copy conjugates explicitly.

Issue 6 (△): Modes clipping. Silent `min()` logic reduces `modes_h`/`modes_w` on coarse grids. **Suggestion.** Raise a warning or forbid $H < \text{modes}$ at inference time.

Issue 7 (△): Complex weight initialisation. Scale $= 1/(C_{\text{in}} \cdot C_{\text{out}})$ is very small ($\mathcal{O}(10^{-3})$ for width 32); early layers easily fall into the pseudo-linear regime. **Suggestion.** Use He initialisation: $\text{std} = \sqrt{2/(C_{\text{in}} + C_{\text{out}})}$.

Issue 8 (△): Non-square grids. Same mode count applied to both axes of a 128×256 grid filters very different physical wavelengths. **Suggestion.** Pass `modes_h`, `modes_w` separately.

C. Dirichlet Boundary Handling. **Issue 9 (!): Boundary enforcement after residual addition.** Spectral blocks can produce large values on $\partial\Omega$ which are then "chopped off"; gradients there are zero due to the loss mask, potentially causing instabilities. **Suggestion.** Zero-out the feature map inside every layer on $\partial\Omega$: `v = v.masked_fill(mask.bool(), 0)` before FFT.

Issue 10 (△): Inhomogeneous $g(x)$. Forgetting to supply g in `loss.backward()` silently switches to homogeneous boundary. **Suggestion.** Store g inside the model or include it in input ζ.

D. Numerical Stability. **Issue 11 (△): Single precision + complex FFTs.** FFT kernels on some GPUs lose ≥ 1 ulp for 512^2 grids causing back-prop noise. **Suggestion.** Offer a `bfloat16`/`float64` switch.

Issue 12 (△): Gradient path through `irfft2` with sparse `out_ft`. Exploding ∇ observed for modes ≥ 32. **Suggestion.** Use spectral dropout during warm-up; clip gradient norm on A_ℓ.

E. Inefficiencies / Code Hygiene. **Issue 13 ($): Zero-initialising `out_ft`.** Allocates $\mathcal{O}(B \cdot C_{\text{out}} \cdot H \cdot W/2)$ every forward pass. **Suggestion.** Re-use a pre-allocated buffer or use `x_ft.new_zeros(...)` once per mini-batch.

Issue 14 ($): `make_coord_grid` uses `repeat(batch, ...)`. Stores B identical coordinate tensors. **Suggestion.** Store a single copy and use broadcasting; or build the grid in the dataloader.

Issue 15 ($): `torch.einsum` with four indices is $\approx 1.3\times$ slower than `torch.matmul` after reshaping. Suggestion. Use `torch.view_as_real`, reshape to $(B, \text{modes}^2, C_{\text{in}})$, matmul to (C_{out}), and restore.

F. Miscellaneous Edge Cases. **Issue 17 (△): `depth=0` crash.** Empty lists and undefined v cause crash. **Suggestion.** Guard with `assert depth >= 1`.

Issue 19 (△): Autodiff through coordinate grid. If `requires_grad=False` is forgotten, coordinates leak into the optimiser. **Suggestion.** Wrap coordinate creation inside `with torch.no_grad():`.

Summary of High-Priority Fixes. Replace `norm='ortho'` with `norm='forward'` for grid-independent scaling; learn both FFT quadrants or enforce conjugate symmetry; zero-out boundary features inside every layer; use He initialisation and separate `modes_h`/`modes_w`; pre-allocate FFT buffers and broadcast coordinate grids.