AISE Projects

Final hand-in for the couse
401-4656-21L
AI in the Sciences and Engineering

by

Wu, You
Department of Mathematics
ETH Zürich
youwuyou@ethz.ch

Preword: I tried to report all projects adhere to the two-page limit.

- *project 1: an exact two-page report is included.*
- *project 2: one additional page serves for visualization.*
- *project 3: the report content itself spans over two pages, but I found it easier to read and follow if the visualizations are on-the-side; thus it ends up with 3 pages in total.*

Bonus: Both bonus tasks are attempted and can be seen after the main projects.

Appendix: This section mainly includes details that I personally found interesting and serves an educational purpose. It also provides additional information regarding model configuration and training parameters to ensure reproducibility.

Project 1: Training the FNO to Solve the 1D Wave Equation

In this project, we aim to resolve for the solution of the IBVP associated with the 1D wave equation. Instead of using a traditional approach, we aim to learn the underlying solution operator via Fourier Neural Operator (FNO), which is a neural architecture capable of learning mapping between infinite-dimensional function spaces. We refer readers to **Appendix P1.A and P1.B** to see more details about neural operators. Our FNO implementation follows the architecture proposed by Li et al. [1]. In our code, the `SpectralConv1d` class corresponds to the Fourier integral operator, whereas `FNO1d`[1] corresponds to the entire multi-layer FNO architecture as in (48). **The generic expression of the FNO architecture** can be seen in (1). We used `nn.Linear` for the implementation of both lifting and projection layers P, Q.

$$\mathcal{G}_\theta := \underbrace{Q}_{\text{Projecting}} \circ \mathcal{G}_\theta^{(L)} \circ \mathcal{G}_\theta^{(L-1)} \circ \dots \mathcal{G}_\theta^{(1)} \circ \underbrace{P}_{\text{Lifting}} \tag{1}$$

Each hidden layer $\mathcal{G}_\theta^{(l)}$ consists of a local linear map $W^{(l)}$, a non-local kernel-integral operator $K^{(l)}$, a bias function $b^{(l)}$, and a nonlinear activation function σ. For Fourier integral operator as we highlighted in equation (2), we used the real fast Fourier transform (FFT) `torch.fft.rfft` and `torch.fft.irfft` for inverse FFT back into the physical space in our Fourier layer for more cost-efficient computation. This is permissible because the provided datasets consist of real-valued data.

$$\mathcal{G}_\theta^{(l)}(u) := \sigma\big(W^{(l)} + K^{(l)} + b^{(l)}\big)(u) = \sigma\left(W^{(l)}(u) + \underbrace{\mathcal{F}^{-1}\big(R^{(l)} \circ \mathcal{F}\big)(u)}_{\text{Fourier Integral Operator}} + b^{(l)}\right), \text{where } l = 1, \dots, L \tag{2}$$

There are several parameters to test out for FNO model configuration. By trials and errors, we chose 30 Fourier modes, and set Fourier layer depth to 30, layer width to 16 for our FNO. Note that having sufficient Fourier modes is essential to capture the full-range of dynamics in our data. We selected the activation function σ to be GELU, which is a smoothened version of ReLU. ⚠️ **For input values:** since the wave equation (43) is second-order in time, we supplemented **two initial conditions** for the resulting evolution problem to be well-posed. We pass the initial data u_0 and the initial velocity v_0 concatenated with the 1D spatial grid in the forward pass to our FNO. Since the velocities are not explicitly provided, we use finite difference to approximate velocities.[2]

Task 1: One-to-One Training

The FNO model trained under the one-to-one training strategy will be used extensively throughout task 1-3, we include details of the parameters used in training, as well as the training history (Figure 20) in **Appendix P1.C**. For task 1, we report the **average relative L2 error** (3), which is defined over 128 trajectories for solutions of the evolution problem at $t = 1$. We can see that our FNO model performs fairly well on the `test_sol.npy` dataset, with an average relative L2 error of around $\sim 5\%$. For task 1 it is not yet obvious if passing the additional v_0 information brings any benefits. Nevertheless, we will consistently include v_0 in our input data throughout all tasks for consistency.

FNO (u_0)	FNO (u_0, v_0)	Error Formula
5.05%	5.21%	$\text{err} = \frac{1}{128}\sum_{n=1}^{128} \frac{\left\|u_{\text{pred}}^{(n)}(t=1.0) - u_{\text{true}}^{(n)}(t=1.0)\right\|_2}{\left\|u_{\text{true}}^{(n)}(t=1.0)\right\|_2}$ (3)

Table 1: Average relative L2 error on `test_sol.npy` for both passing and not passing initial velocity

Task 2: Testing on Different Resolutions

An operator that is capable of doing **zero-shot super-resolution** can evaluate inputs of a higher resolution in a zero-shot manner, despite the fact that it was trained on a lower resolution. To examine this property, we evaluate $\mathcal{G}_\theta$ on testing datasets of $128 \times 2 \times s$ resolution, with varying spatial resolution indicated by $s \in \{32, 64, 96, 128\}$.

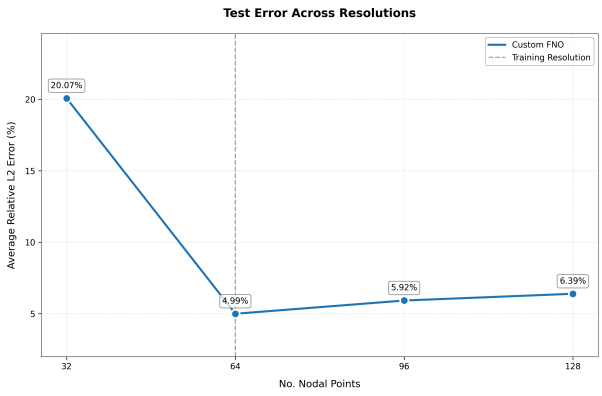


Figure 1: Average relative L2 error with equation (3) of our FNO implementation for 128 training trajectories. We marked the training spatial resolution with 64 nodal points with the grey dashed line.

- **Smallest Errors at Trained Resolution ($s = 64$):** In Figure 1, the lowest errors on test data can be found at the training resolution $s = 64$, which is indicated by the grey dashed line on the plot. Our implementation shows an average relative L2 error of 4.99%.
- **Success in Higher Resolutions ($s > 64$):** More importantly, for higher resolutions which is at $s \in \{96, 128\}$, we achieve accuracy of 5.92% and 6.39% respectively for our implementation. This demonstrates our implementation can perform zero-shot super-resolution, which is an important property of the FNO.
- **Failure in Lower Resolution ($s < 64$):** A notable observation is the failure of the FNO in resolving lower resolution. In Figure 1, we can see our model exhibits an error of 20.07% in the coarsest grid that consists of 32 nodal points.

The observation for **failure in lower resolution** is an expected behavior, reflecting the fact that the *FNO is not a Representation Equivalent Neural Operator (ReNO)*. The continuous-discrete equivalence (CDE) property is not preserved. This caveat is brought by the choice of nonlinear activation functions. In Bartolucci et al. [2], it was shown that nonlinear activation function drives us out of bandlimited function space, introducing aliasing error. For lower resolutions, the high-frequency pollution introduced by nonlinear activation functions strongly affect coarse grid resolution and we fall into the zone where there is no representation equivalence. Our custom implementation gives an 20.07% error.

[1] Model architecture implementation locates under `project_1/fno.py`

[2] Whether the initial velocity v_0 is needed is not obvious yet, but proved to be essential for all2all training later for task 4 & bonus.

To conclude task 2, we reported the average relative L2 error in Figure 1. **We verified our implementation can do zero-shot super-resolution** by showing it exhibits similar error magnitudes in $s \in \{96, 128\}$, compared to the training resolution where $s = 64$. However, we observed the failure of FNO in resolving lower resolutions, reflecting the fact that FNO is not a ReNO and the quality of approximated solutions is resolution-dependent.

Task 3: Testing on Out-of-Distribution (OOD) Dataset

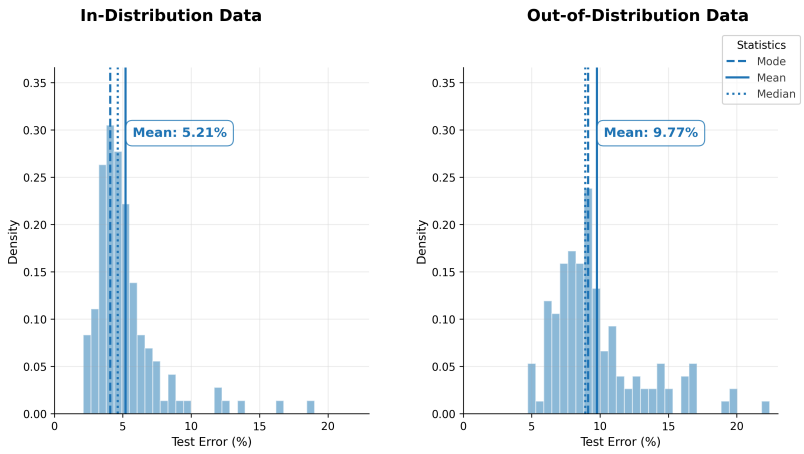


Figure 2: Histogram distributions of relative L2 test errors over 128 testing trajectories for our FNO implementation. Results show performance on both in-distribution **(task 1)** and out-of-distribution **(task 3)** test data, with vertical lines indicating the mode (dashed), mean (solid), and median (dotted) of each distribution.

In **task 3**, we test our final solution on OOD dataset that is of resolution $128 \times 2 \times 64$. Since no other spatial resolutions are provided, we will only compare our results on OOD dataset against the ones obtained from **task 1** throughout this task.

In Figure 2, we compare the relative L2 test errors for both the in-distribution testing dataset, and the OOD testing dataset. Both datasets consists of 64 spatial grid points. As reported in the error distribution plot at the left side, the average L2 test errors for in-distribution data are lower. Our implementation achieved 5.21% of average relative L2 error across 128 trajectories. For OOD data, we observed our models performed worse regardless of the implementation details. Both models exhibit similar errors, our implementation has an average L2 error of 9.77% in resolving the test data.

To conclude, compared to task 1, the average relative L2 error we obtained in OOD dataset is higher. This is matching our expectation because our model learned the particular data distribution μ during the training phase. However, the OOD initial data is chosen as $v \sim v$ with some other measure v such that $v \approx \mu$ but $v \neq \mu$. Thus, on unseen distribution, our model demonstrated worse performance.

Task 4: All2All Training

In order to account for time inputs as an extra dimension, we need to modify our `FNO1d` class, which corresponds to the multi-layer architecture in equation (48). We used the general-purpose conditioning FiLM layers [3] to embed the temporal information as additional variable in our model. If `time_dependent = True` is specified, the temporal information Δt will be passed through such time-conditional layer of the class `FILM`, which essentially performs an affine transformation on the input data, and scaled it with this additional Δt feature. The `SpectralConv1d` class for the Fourier integral operator is kept unchanged. ⚠️ **We again emphasized that we passed both u and its temporal derivative u_t (approximated via finite difference) into our model.** For training, we used same parameters as in table 1 & 2 for the model in task 1, with the only difference that we increased the width of our model from 16 to 32. We are interested in comparing results from **one-to-all training** and **all2all training** for time-dependent FNO, thus we trained two time-dependent FNO models with identical hyperparameters. We tested our trained models with two different training strategies on the `test_sol.npy` dataset, focusing **only** on predictions at $t = 1.0$.

Metric	One-to-All Training	All2All Training
Training Trajectories	64	64
Validation Trajectories	64	64
Sample Pattern	(t_0, t_i) sequential for $i \in [k]$	$\frac{k\times(k+1)}{2}$ combinations
Samples per Trajectory	5	$\frac{5\times6}{2} = 15$
Total Training Samples	$64 \times 5 = 320$	$64 \times 15 = 960$
Total Validation Samples	$64 \times 5 = 320$	$64 \times 15 = 960$
Total Samples	$320 + 320 = 640$	$960 + 960 = 1920$

Table 2: Number of samples for one-to-all (vanilla) and All2All strategies for $k = 5$ timesteps

The biggest difference between two strategies lie in the way how the training data is utilized. With **All2All**, we learn $\mathcal{G}_\theta$ starting from any intermediate distributions $u(\cdot, t)$, as long as t is one of the time points where snapshots of the training data are available. As a result, All2All allows us to utilize quadratic $O(k^2)$ samples per trajectory whereas the vanilla training only makes use of $O(k)$ training samples per trajectory. In the table below, we report the number of samples used for both strategies with the provided training dataset `training_sol.npy`. **For All2All approach, the total number of samples used for training are 1920, where 960 samples are dedicated for training and the other 960 for validation purpose.**

We also took the ordering of time points into account and ensured $t_{\text{in}} \leq t_{\text{out}}$.

Results

In Table 3, we report the average relative L2 error (3) as before over 128 trajectories, tested on the `test_sol.npy` dataset for $t = 1.0$ only. For conciseness, we do not include the training history here, but it can be found under the `project_1/checkpoints` folder for each model used.

One-to-One (Task 1)	One-to-All: Direct Inference	All2All: Direct Inference
5.21%	7.74%	8.20%

Table 3: Average relative L2 error with different training strategies, evaluated for the in-distribution data at $t = 1.0$.

As we can tell from table 5, models based on our time-dependent FNO implementation can resolve the solution of the PDE system at $t = 1.0$ at a rather high accuracy. **Time-dependent FNO results (7.74% and 8.20%) are indeed slightly worse than the time-independent model (5.21%) from task 1.** But this may due to the fact that we have a more complex loss function to minimize and our training procedure could be further improved. Furthermore, we do not observe a huge difference in results obtained from one-to-all (vanilla) or All2All strategy yet. Both time-dependent FNO models exhibit errors of similar magnitude at $t = 1.0$. In all, we conclude that our time-dependent training is successful. ⚠️ **This success is due to the fact that we approximate the velocity $u_t(\cdot, t)$ from $u(\cdot, t)$ via finite difference, and pass both data as initial conditions of the IBVP (44) when training our model.** Thinking about the fact that we have a wave equation which is second-order in time, providing two initial conditions appears to be very natural. This way we can ensure our problem is well-posed. **So what would happen if only $u(\cdot, t)$ is provided, without approximating $u_t(\cdot, t)$ and providing the second initial condition to a separate input channel?** We attempted this earlier and observed that, for the FNO that was trained on one-to-all time pairs, it had worse performance ($\sim 10\%$ of error), but it was still demonstrating correct behavior. However, for All2All, it was devastating and led to errors of very high magnitude ($\sim 30 - 40\%$). For more detailed information on how we approximate the velocities $u_t(\cdot, t)$, see **Appendix P1.F**.

Project 2: Reconstructing PDEs from Data Using PDE-Find

In PDE-Find [4], the main goal is to assemble a linear system of equations (LSE) from both the PDE solution data and approximation of derivatives of the solution, and then use sparse regression to solve the LSE for symbolic regression of the underlying equation. We consider a simplified setup, where all solution data is real-valued and the domain $\tilde{\Omega}$ is discretized into equidistant space-time grid of dimension $n \times m$, such that nodal values represents n spatial measurements at m time points. We can now rearrange the solution into a real-valued data matrix $U \in \mathbb{R}^{(n \cdot m) \times 1}$ which contains the discretization of the solution u across spatiotemporal domain. Similarly, the temporal derivative u_t is assembled into a column vector $U_t \in \mathbb{R}^{(n \cdot m) \times 1}$.

$$U_t = \Theta \cdot \xi \tag{4}$$

In equation (4), the first-order temporal derivative u_t of the discretized solution is set as the LHS of the equation. We call the matrix $\Theta \in \mathbb{R}^{(n \cdot m) \times D}$ the **feature library**, where D represents the number of candidate functions in the library. The feature candidates (column vectors of Θ) are discretized solution u and its mixed partial derivatives (u_t, u_x, uu_x …). With careful selection of feature candidate functions as column vectors in the matrix Θ, we can assume that the matrix is an overcomplete library such that the PDE can be represented by linear and nonlinear-combinations of the feature candidates.[3] Once we obtain the solution $\xi \in \mathbb{R}^{D \times 1}$, we can use its coefficients to recover the dynamic system, where each non-zero entry $\xi_j \neq 0, \forall j \in \{1, ..., D\}$ of the solution ξ represents a term in the PDE. This way, we can recover the governing equation using $u_t(x, y) \coloneqq \sum_j \Theta_j \xi_j$

Computing Derivatives

An essential part in the PDE-Find algorithm lies in computing the partial derivatives. In the original implementation of PDE-Find, simple **finite difference** is used for approximating the spatial and temporal derivatives on clean data. For noisy data, they utilized polynomial interpolation to overcome the additional difficulty. In project 2, **we initially wanted to train simple feedforward neural networks (FNNs) on all three provided datasets**, and then we can use the trained FNN to approximate the dataset better and then perform automatic differentiation to compute derivatives. However, due to the high training cost, **we only used FNN + automatic differentiation for the first two systems**. **For the third coupled system, we simply perform second-order accurate central differences method** with first order estimates at the boundaries via `torch.gradient` directly on the dataset.

1D Implementation (FNN + automatic differentiation)

For affordable 1D datasets, we trained simple **feed-forward neural networks (FNNs)** to approximate spatiotemporal solutions of the PDE, solely based on the provided `X.npz` dataset. For $X \in \{1, 2\}$, we trained a separate FNN as part of the feature library preparation process. For the **network architecture**, we used a simple feedforward neural network (FNN) with three linear layers of customable width, each linear layer is followed by an nonlinear activation. Even this construct is simple, it allows us to nicely approximate the solution $u(x, t)$ across time, thanks to the universal approximation property of FNN [5]. Once we have successfully trained the FNN and verified its expressiveness by comparing it to the original dataset, we use the **automatic differentiation** utility of the `PyTorch` library, provided by `torch.autograd.grad`, for computing derivatives up to arbitrary order.

2D Implementation (Direct `torch.gradient`)

For applying PDE-Find on the 2D dataset of system 3, we did not train a FNN, but performed the `torch.gradient` directly on the dataset. Under the hood, it performs second-order accurate central differences method with first order estimates at the boundaries. This is computationally much more affordable compared to the neural network based approach.

Building Libraries of Candidate Terms

To build the feature library Θ as in (4), we use the numerical approximation of derivatives as introduced in the previous section. We used automatic differentiation on FNN on 1D data to obtain derivatives, and direct finite difference on 2D data. Then for each system indexed with X, for $X \in \{1, 2, 3\}$ we need to make careful decision for selecting suitable candidates accordingly. Since it is give by project hand-out that the highest order of derivatives is 3, we set the upper limit to order of derivatives to 3. In **Appendix P2.A**, we explicitly list out candidates used for each system in our experiments. An example assembled LSE as shown below (5) is taken from the supplementary material of PDE-Find [6] for some spatio-temporal grid $\tilde{\Omega}$ with n spatial grid points and m time snapshots, and D indicates the number of candidates in the library.

$$\underbrace{\begin{bmatrix} u_t(x_0, t_0) \\ u_t(x_1, t_0) \\ u_t(x_2, t_0) \\ \vdots \\ u_t(x_{n-1}, t_m) \\ u_t(x_n, t_m) \end{bmatrix}}_{U_t \in \mathbb{R}^{(n \cdot m) \times 1}} = \underbrace{\begin{bmatrix} 1 & u(x_0, t_0) & u_x(x_0, t_0) & ... & u^5 u_{xxx}(x_0, t_0) \\ 1 & u(x_1, t_0) & u_x(x_1, t_0) & ... & u^5 u_{xxx}(x_1, t_0) \\ 1 & u(x_2, t_0) & u_x(x_2, t_0) & ... & u^5 u_{xxx}(x_2, t_0) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & u(x_{n-1}, t_m) & u_x(x_{n-1}, t_m) & ... & u^5 u_{xxx}(x_{n-1}, t_m) \\ 1 & u(x_n, t_m) & u_x(x_n, t_m) & ... & u^5 u_{xxx}(x_n, t_m) \end{bmatrix}}_{\Theta \in \mathbb{R}^{(n \cdot m) \times D}} \underbrace{\begin{bmatrix} \xi_0 \\ \xi_1 \\ \xi_2 \\ \vdots \\ \xi_D \end{bmatrix}}_{\xi \in \mathbb{R}^{D \times 1}} \tag{5}$$

Sparse Regression

Once we have built the candidate library $\Theta \in \mathbb{R}^{(n \cdot m) \times D}$ and computed the temporal derivative of the solution u_t, we can solve for the unknown ξ in the LSE (4). Note that since the resulting Θ **is not** a square matrix, instead, it is a *tall* rectangular system matrix with $(n \cdot m) > D$. For such **overdetermined system, where we have more equations than the unknowns**, we need to seek for the **least squares solution**. Moreover, we are motivated to use **sparse regression** because we know in advance that the solution vector ξ will have sparse coefficients. If we used plain least square methods without regularizations, we would end up in densely populated solution vector ξ, which is not desirable for what we aim for. Therefore, we implemented the *Algorithm 1. STRidge* with ridge regression. Recall that each ridge regression is to solve a linear least squares problem with l2 regularization, in our context, we aim to minimizes the following objective function:

$$\arg\min_{\xi \in \mathbb{R}^D} \|U_t - \Theta\xi\|_2^2 + \lambda \|\xi\|_2^2 \tag{6}$$

where λ is used to controls the regularization strength and penalize when the found coefficient ξ is too large. In our implementation, we use `sklearn.linear_model.Ridge` with `fit_intercept = False` and set `tol=1e-5` and `max_iter=500`. In addition to the ridge regression (6), **STRidge recursively performs ridge regression on the augmented LSE** and seek for some $\hat{\xi}$ that minimizes the regularized least squares problem. After each ridge regression solve, we augment the LSE to be solved by applying a predefined hard threshold τ and zero out all coefficients and set $\xi_i = 0$ in the solution vector for those that are larger than the threshold $\xi_i > \tau$. To gradually refine the selection of the best tolerance τ used in algorithm 1, we also implemented the *Algorithm 2. TrainSTRidge*. This algorithm performs the STRidge for a fixed number of time. Among all solves, it finds the optimal predictor ξ_{best} by optimizing the STRidge performance on a selected split of subset that serves as validation from the original dataset. Both algorithm 1 and 2 can be found in the supplementary material of the original PDE-Find paper [4], we also supplement them in **Appendix P2.B** for completeness.

[3] Under the assumption by the task hint, we assume the partial derivatives are only up to and include third order derivatives.

Results

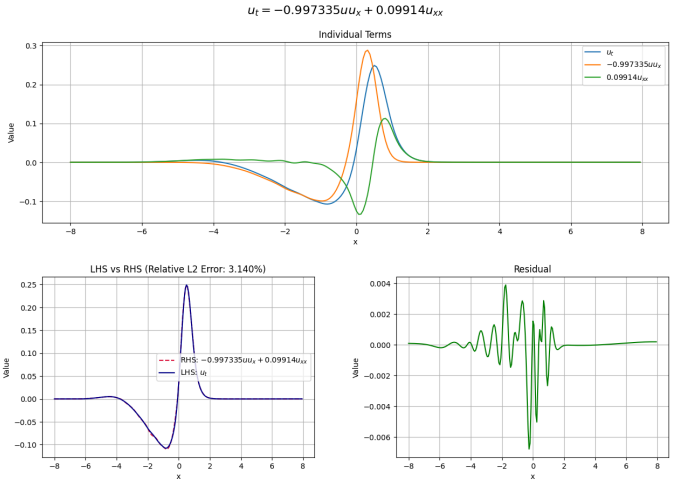
We present results on three different systems, using provided datasets named by `X.npz` for $X \in \{1, 2, 3\}$. For each system we report the discovered equation and the relative L2 error between the LHS and the RHS. The parameters used in sparse regression with `TrainSTRidge` are reported in the associated table for each dataset. For conciseness, we only report the number of candidate terms D used for each system. We again refer readers to **Appendix P2.A** to see the full expression of the used candidates, as well as the analysis of each least squares problem based on the condition number of the Θ matrix.

System 1 (FNN + automatic differentiation)

- *Discovered Equation:*

$$u_t = -0.997335 \cdot uu_x + 0.099140 \cdot u_{xx} \tag{7}$$

- *Relative L2 Error of (7) :* 3.140%



Library Size D	Ridge Penalty λ	STRidge Tolerance	STRidge Iterations
31 Terms	1e-6	5e-3	10
Ridge Iterations	**Train Split**	**Train Iterations**	**Train Penalty η**
500	0.5	50	1e-3

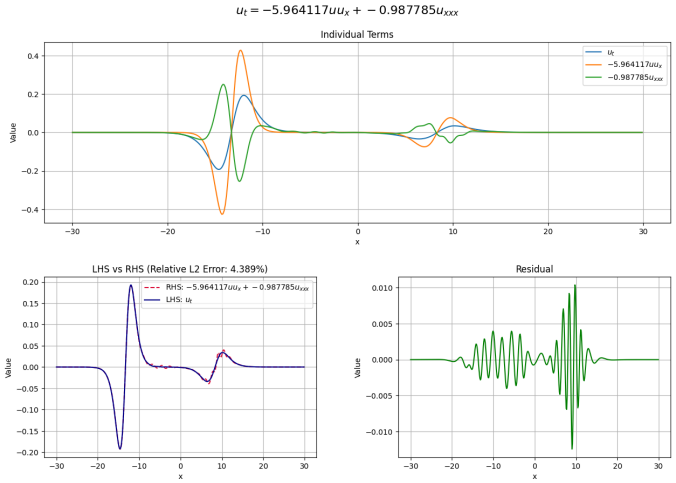
Table 4: Sparse regression parameter for system 1.

System 2 (FNN + automatic differentiation)

- *Discovered Equation:*

$$u_t = -5.964117 \cdot uu_x - 0.987785 \cdot u_{xxx} \tag{8}$$

- *Relative L2 Error of (8) :* 4.389%



Library Size D	Ridge Penalty λ	STRidge Tolerance	STRidge Iterations
13 Terms	1e-6	1e-2	10
Ridge Iterations	**Train Split**	**Train Iterations**	**Train Penalty η**
500	0.7	50	1e-3

Table 5: Sparse regression parameter for system 2.

Success & Failures

As shown above, we were not able to find an approximation of the governing equations, but the relative L2 error between the LHS and RHS is quite low $3 - 4\%$ for both 1D systems. **This first experiment indicates the success of our implementation.** However, unlike the straight forward success for system 1, where we were able to use 31 terms for the Θ library assembly, **we were not able to use more candidates for system 2.** As shown in Figure 4 and Figure 6, system 2 consists of two non-interacting traveling waves with different amplitudes to the KdV equation. **The similarity of the underlying physics to another system poses significant difficulty in symbolic regression**. For example, the nature of the one-way wave equation $u_t + cu_x = 0$ is highly similar with system 2. In the original paper, an approach for disambiguation is proposed in which the least squares problem is solved in a block-wise fashion combining two sets of observations. **We could potentially further improve our method by adding the disambiguation for system 2.** Additionally, we note that when the solution profile sharply rises or falls, the residuals are significant. Moreover, a greater percentage of errors are introduced by higher-order derivatives. In contrast to u_t and uu_x, u_{xxx} exhibits a significant oscillating pattern in the plot for system 2, which results in large errors at about $x = 10.0$.

System 3 (Direct `torch.gradient`)

- *Discovered system:*

$$u_t = 0.978845 \cdot u + 0.099731 \cdot u_{xx} + 0.139817 \cdot u_{yy} - 0.952589 \cdot u^3 + 0.993481 \cdot v^3 - 0.955152 \cdot u \cdot v^2 + 0.992257 \cdot u^2 \cdot v \tag{9}$$

$$v_t = -0.987519 \cdot u^3 + 1.175099 \cdot v + 0.106914 \cdot v_{xx} + 0.152016 \cdot v_{yy} - 1.159284 \cdot v^3 - 0.999191 \cdot u \cdot v^2 - 1.160521 \cdot u^2 \cdot v \tag{10}$$

with relative L2 Error of (9) : 11.802% and *of (10) :* 12.239%

Library Size D	Ridge Penalty λ	STRidge Tolerance	STRidge Iterations
23 Terms	1e-6	5e-3	10
Ridge Iterations	**Train Split**	**Train Iterations**	**Train Penalty η**
500	0.8	60	1e-3

Table 6: Sparse regression parameter for system 3.

Success & Failures

For system 3, we noticed that the sparse regression does not converge on the full LSE. We randomly select 10,000 rows of the Θ matrix and the corresponding rows of the RHS vector with the sampling operator $\mathcal{C}$. Thus, **we used a compressed library** $\mathcal{C}(\Theta)$ and the associated compressed RHS $\mathcal{C}(u_t)$ or $\mathcal{C}(v_t)$ respectively for system 3. **This effectively resolves the issue and gives reasonable results (with $\sim 11 - 12\%$ of relative L2 error) for the coupled system we obtained.** However, we also observed that the quality of the selected rows of Θ largely influence our final results. **In future, improvements could be done to select a high-quality sample collection which captures the most important dynamics of the system**, or one could optionally implement a routine similar to algorithm 2 that performs regression on a wide range of randomly selected $(\mathcal{C}(\Theta), \mathcal{C}(u_t))$ pairs and find the optimal indices for sample selection. Moreover, we also noticed that the 2D problem in system 3 exhibits larger errors compared to system 1 & system 2. We assume this is mainly due to the fact that there are more terms involved in the actual solution.

Visualization

FNN-Approximated Solutions

- for system 1 & system 2, we used FNN to approximate datasets in order to further perform automatic differentiation.

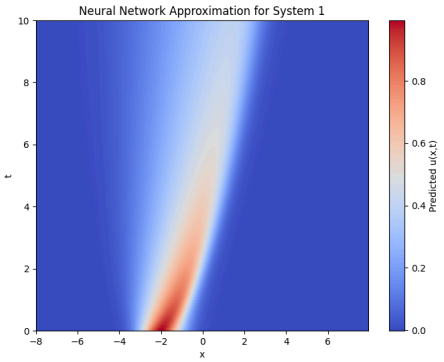


Figure 3: Heatmap of FNN-approximated solution across spatio-temporal grid for **system 1**

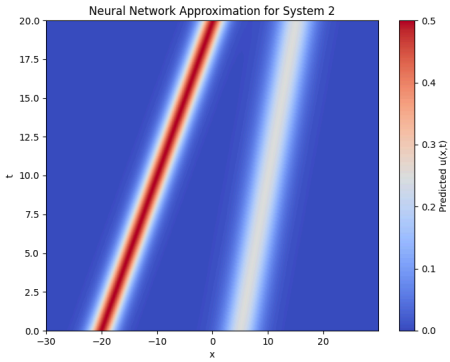


Figure 4: Heatmap of FNN-approximated solution across spatio-temporal grid for **system 2**

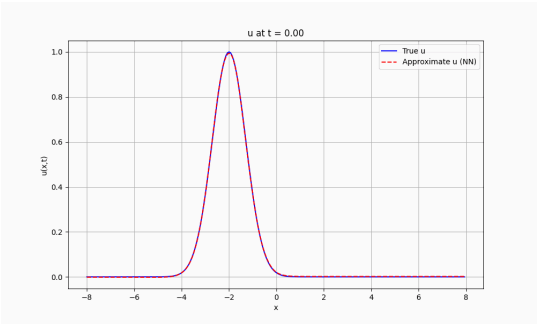


Figure 5: Comparison of ground truth and FNN-approximated solution of **system 1** at $t = 0$

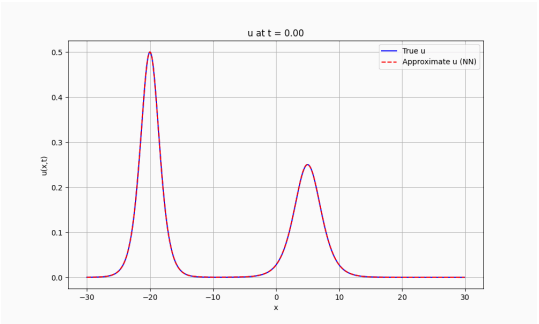


Figure 6: Comparison of ground truth and FNN-approximated solution of **system 2** at $t = 0$

Original 2D Data

- Following we present heatmap plots of the original 2D data u and v at final observation time. Note that we did not use FNN to approximate this dataset, but use `torch.gradient` to compute derivatives directly.

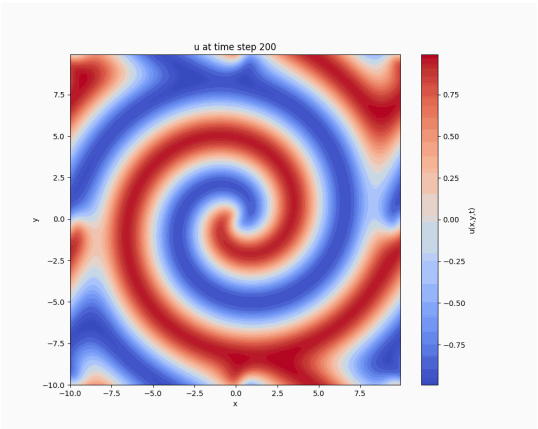


Figure 7: Heatmap plot of the **original** 2D data u at final time $t = T$

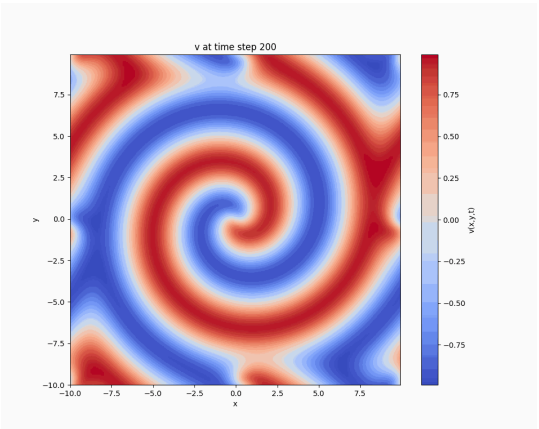


Figure 8: Heatmap plot of the **original** 2D data v at final time $t = T$

Project 3: Foundation Models for Phase-Field Dynamics

The Allen-Cahn equation describes the order-disorder transition in a many-particle system [7]. It is a nonlinear parabolic equation that models the physics of reaction-diffusion phenomena:

$$\frac{\partial u}{\partial t} = \Delta_x u - \frac{1}{\varepsilon^2} f(u) \tag{11}$$

where the parameter ε controls the width of transition layers between phases. The right-hand side function $f(u)$ is the derivative of a non-negative function $F \in C^1(\mathbb{R})$ with two minima $F(\pm 1) = 0$, meaning that the solution of the system u will evolve and converge to two stable equilibria $u \in \{-1, 1\}$ at its energy equilibrium. Here, we set $f(u) := u^3 - u$ and seek for the solution of (11) defined on the spatio-temporal domain $\tilde{\Omega} := \Omega \times [0, T]$, where $T > 0$ is some final observation time.

Data Generation

The Allen-Cahn equation (11) exhibits different behaviors depending on the parameter ε. In order to capture full range of dynamics, we experimented with a wide range of ε parameters, and finally selected ε to be 0.1, 0.05, and 0.01 in our training dataset. At *large* ε values like 0.1, the system shows **smooth, diffusion-dominated** behavior. Conversely, at *small* values when $\varepsilon \to 0$ like 0.01, **rapid phase separation occurs with sharp interfaces**, **nonlinear behaviors dominates over diffusive behavior**. The 0.05 is selected is an appropriate transition value showing intermediate dynamics. Notably, when ε is very small, the system becomes stiff. Therefore, to cope with potential issues with stiffness, we employed `scipy.integrate.solve_ivp` with the implicit integrator `Radau` for the implementation of the numerical solver, which can be seen under `data_generator.py`.

For generation of each trajectory across time, we fixed the spatial domain on the interval [−1.0, 1.0] with 128 grid points, as in the provided template code. To comply with the selected ε values, we used 5 uniformly spaced time points from 0.0 to 0.01, with a fixed timestep of $\Delta t = 0.0025$ for the temporal evolution. We emphasize that ε and Δt need to be selected in tandem. If we aim to capture the transient behavior of the system, but choose a timestep that is too large, we will not be able to capture enough dynamics after the initial snapshot. We used three different types of initial conditions $u_0 = u(x, 0)$ and implemented the corresponding `FunctionSampler` class: 1. Piecewise Linear (PL); 2. Gaussian Mixture (GM); 3. Fourier Series (FS). For each generated u_0, we ensured periodic boundary condition is enforced and nodal values $u_0 \in [-1, 1]$ holds. See **Appendix P3.A** for more details about each function class. **For the training dataset, we generated 1000 trajectories across spatio-temporal domain.**

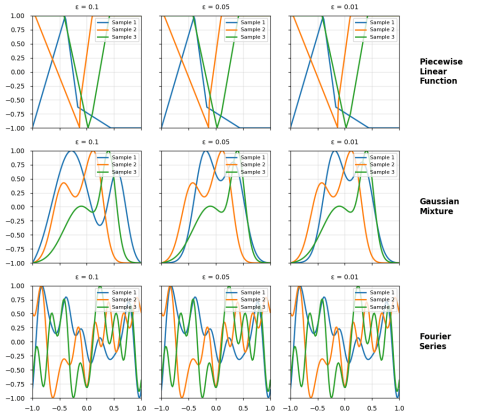


Figure 9: Selected three training trajectories at initial time $t = 0$. Periodic boundary condition is enforced and nodal values $u_0 \in [-1, 1]$ holds.

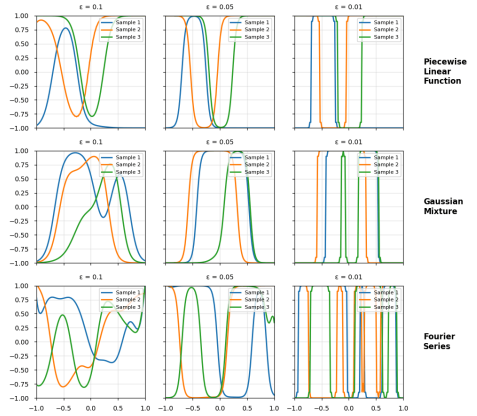


Figure 10: Selected three training trajectories at final observation time $T = 0.01$ after advancing 4 time steps of size $\Delta t = 0.0025$.

Generating Testing Data & OOD Data & ε-Test Data

We generated the standard testing set (`test_sol.npy`) in the same way we generated the training data, making it also in-distribution. Furthermore, we aim to evaluated our model's robustness through two more test sets. **The out-of-distribution (OOD) test set** (`test_sol_OOD.npy`) challenges the model with more complex patterns than seen during training: additional breakpoints in piecewise linear functions, more components in Gaussian mixtures, and higher-frequency terms in Fourier series. **To test if our model is capable of extra- interpolating ε parameters**, we generated another test set (`test_sol_eps.npy`) using the standard initial conditions but with epsilon values ranging from 10.0 to 0.006, extending well beyond our training range of 0.1 to 0.01. In the following figure, one can see the generated initial data at $t = 0$. **For each testing dataset, we generated 200 trajectories across spatio-temporal domain.**

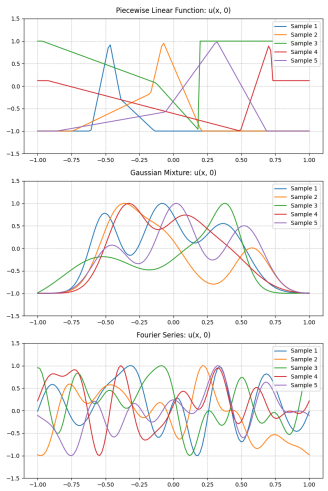


Figure 11: Testing Data (`test_sol.npy`)

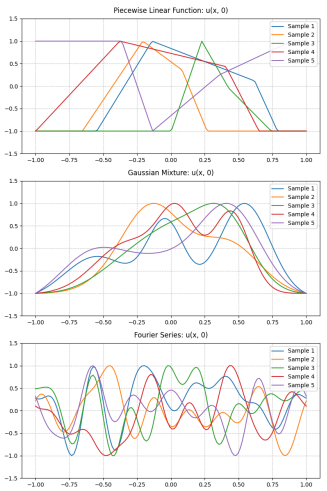


Figure 12: ε Testing Data (`test_sol_eps.npy`)

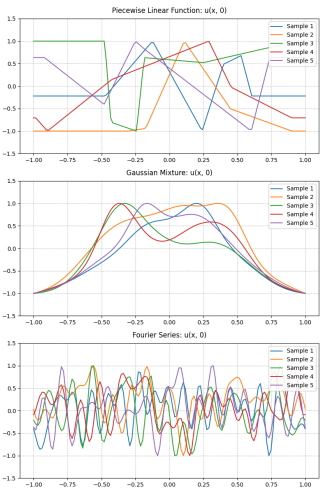


Figure 13: OOD Testing Data (`test_sol_OOD.npy`)

Neural Foundation Model

Our foundation model `AllenCahnFNO`[4] builds upon two important components. We used Fourier Neural Operator (FNO) [1] as the model backbone, and trainable general conditioning FiLM [3] layers are incorporated after each FNO block and an activation layer. This way we can **embed both the time-dependency, as well as the ε parameter** into our model. **Furthermore, our model is capable of handling different spatial and temporal resolutions.** In a forward pass through our model, we pass in the initial condition u_0, the associated ε value, and the temporal array $t := \{1\Delta t, 2\Delta t, 3\Delta t, 4\Delta t\}$ with some fixed equidistant timestep size Δt. We then automatically detect the number of spatial and temporal grid points from the shape of u_0 and t, respectively. The output is a 4×128 tensor storing time snapshots after advancing $i\Delta t$ in time from the provided u_0. Note that for both training and fine-tuning, we used all training samples at once, without subdividing them based on the initial condition used or the magnitudes of ε values. This is because *Curriculum Learning* [9] easily introduce biases for our `AllenCahnFNO` model, which we further elaborated in **Appendix P3.B**. For model configuration, we used 20 Fourier modes and a depth of 2 for each layer type, as well as a width of 64 and use GELU as our activation function.

Training

For training, we **only** trained **one base model** on the comprehensive `train_sol.npy` dataset across various initial conditions and ε values. Recall that we have generated 1000 trajectory across spatio-temporal domain for each data category by setting `n_train = 1000`, we have also three ε values where $\varepsilon \in \{0.1, 0.05, 0.01\}$ as well as three function classes where the initial condition u_0 belongs to. This gives us 9000 samples in all for the `train_sol.npy` dataset.[5]

Training Samples	Validation Samples	Maximal Number of Epochs	Early stopping
8100	900	800	After 50 epochs
Curriculum Learning	**Optimizer**	**Batch Size**	**Learning Rate**
False	AdamW	512	0.0001

Table 7: Training configuration for AllenCahnFNO.

Fine-Tuning

To examine if our `AllenCahnFNO` can serve as a foundation model, we **fine-tuned** the base model by passing it some few-shot samples before inference. Different from training the base model, **we freezed all other layers, but left the FiLM layers unfrozen**. This results in very lightweight fine-tuning in which **only 0.3%** of the parameters are trained. Fine-tuning epochs are set to 100 and only 20 fine-tune samples with a 16/4 split for training/validation are used.

Fine-Tune Samples	Total Model Parameters	Trainable parameters	Frozen parameters
20	387,009	**1,024 (0.3%)**	385,985 (99.7%)
Curriculum Learning	**Optimizer**	**Batch Size**	**Learning Rate**
False	AdamW	2	0.01

Table 8: Fine-tuning configuration for AllenCahnFNO.

Results

We report the average relative L2 error across 200 testing trajectories per testing category on datasets `test_sol.npy` and `test_sol_OOD.npy` in Table 11 & 12. The error is computed by comparing prediction against ground truth at each fixed time snapshot, and averaged over all trajectories. Note that for each dataset we have a fine-tuned model on that specific dataset. In all we have one base model named `base_model.pth` and three fine-tuned models named `finetuned_model_test_sol.pth`, `finetuned_model_test_sol_OOD.pth` and `finetuned_model_test_eps.pth`.

IC Type	ε	Zero-shot	Fine-tuned
PL	0.10	10.1411%	10.0359%
	0.05	8.1278%	8.0712%
	0.01	5.0139%	5.2801%
FS	0.10	26.8162%	26.7704%
	0.05	27.4113%	27.3574%
	0.01	16.3927%	16.0335%
GM	0.10	3.9864%	3.7835%
	0.05	6.4224%	6.3270%
	0.01	2.8208%	2.7567%

Table 9: Average relative L2 error between zero-shot and fine-tuned models on in-distribution test set (`test_sol.npy`)

IC Type	ε	Zero-shot	Fine-tuned
PL	0.10	16.5216%	16.3927%
	0.05	13.1615%	13.4098%
	0.01	3.6672%	4.2901%
FS	0.10	87.3301%	83.1482%
	0.05	47.4966%	45.9286%
	0.01	28.0266%	27.0400%
GM	0.10	3.7367%	4.7908%
	0.05	4.9008%	5.1821%
	0.01	2.0659%	2.5415%

Table 10: Average relative L2 error between zero-shot and fine-tuned models on OOD test set (`test_sol_OOD.npy`)

In tables above, we colored the fine-tuned results with green if it shows lower error percentage comparing to the zero-shot inference obtained directly via evaluating on the base model, and red if it becomes worse. We observe that, **our model is capable to resolve phase-field dynamics across different ε values**. Moreover, our model is also capable to capture nuances in different initial conditions. **We obtained less than 10% relative L2 errors on most subsets of PL and GM function classes.** For FS we obtained notably worse results, this may due to the fact that choosing `n_modes = 10` allows too many high-frequency features in our dataset, which behaves like an outlier, compared with other two initial condition types, that only mildly have around 4 breakpoints or components. **GM is remarkably the easiest category out of all function classes**, demonstrating errors $< 5\%$.

Furthermore, we highlight that our model is capable of transfer learning. For in-distribution dataset, all fine-tuned results, besides the $\varepsilon = 0.01$ for PL function class, improved after fine-tuning on only 20 samples. However, the OOD dataset worsens its results in both PL and GM datasets after fine-tuning, leaving only the FS category with some improvements. This again reflects our previous observation that **learning the observations from FS dataset is inherently harder due to its high-frequency features**. We increased the Fourier mode for OOD dataset from 10 to 20, this further leads to the inbalance in dataset difficulty. Interestingly, increasing the number of Gaussian components leads to better performance on the OOD dataset. This is due to the fact that the superposition of more Gaussian components give a more homogeneous profile of the initial condition.

[4] Previous work such as Subramanian et al. [8] has successfully shown the potentials of FNO to learn underlying physics for a wide range of downstream tasks. This further motivates our choice and has been proven successful in our experiments.

[5] Note that each sample can be further broken down into 5 time snapshots, but in our implementation, the model predicts 4 further time snapshots at once, thus we arrive at a train/validation split of 8100/900 samples.

Generalization Across Different ε Values

Next, we report results on the ε-test dataset (`test_sol_eps.npy`). This dataset consists of 6 unseen ε values for each IC type, where we tested ε values into three categories: The first category consists of ε values in $\{0.075, 0.025\}$, it tests the interpolation capability of our model. Next is $10.0, 0.5$, which tests the extrapolation to higher values. At last, $0.008, 0.006$ tests the extrapolation to lower values. As we can observe in Table 6, **our base model is capable of both interpolating, as well as extrapolating to lower values.** In both mentioned ε subsets, we got similar results as in the previous two datasets. **However, extrapolating ε to higher values poses a significant challenge for our models.** Our base model fatally failed to predict at higher unseen values $\varepsilon \in \{10, 0.5\}$.

Success of Lightweight Fine-Tuning

While our base model failed to predict higher ε values, the fine-tuned model gives remarkable results by learning from only 20 samples. For all IC classes, we significantly improve the quality of prediction. The GM class performs the best in which it improved from 12466.9521% and 310.5461% of average relative L2 error to 14.3580% ($\varepsilon = 10.0$) and 15.3666% ($\varepsilon = 0.5$). Similar trends can be seen for PL and FS classes too. **We highlight the fact that this success is obtained by activating only 1,024 parameters in fine-tuning, which is equivalent of 0.3% of the total parameters $387,009$ of our `AllenCahnFNO` model.** Despite the success, we also point out that fine-tuning does harm performance on other categories. For interpolated value such as $\varepsilon = 0.025$, our fine-tuned model downgraded its performance in which it consistently gives larger errors such as $5.7023\% \to 7.0492\%$ for PL class, $25.8429\% \to 31.3667\%$ for FS class and $5.9716\% \to 7.6806\%$ for GM class. For completeness, in Figure 16 we also reported sample trajectory at index 5 from each IC function class, the initial condition is reported in grey, and the predicted results are reported in dashed colored lines, and ground truth are marked by color-filled bold lines. The title reports the average relative L2 error in percentage. One can again see clearly how just fine-tuning FiLM layers is already powerful enough to give us reasonable predictions across time for all IC types.

IC Type	ε	Zero-shot	Fine-tuned
PL	10.000	9409.0869%	64.7127%
	0.500	233.5755%	26.3383%
	0.075	9.6927%	15.7740%
	0.025	5.7023%	7.0492%
	0.008	3.4510%	5.6850%
	0.006	3.8525%	5.7232%
FS	10.000	30882.6602%	61.9866%
	0.500	1394.6853%	53.6279%
	0.075	34.1827%	54.2362%
	0.025	25.8429%	31.3667%
	0.008	32.3054%	30.9287%
	0.006	30.1576%	28.5304%
GM	10.000	12466.9521%	14.3580%
	0.500	310.5461%	15.3666%
	0.075	6.4313%	13.3130%
	0.025	5.9716%	7.6806%
	0.008	2.9431%	6.3092%
	0.006	3.3061%	6.2021%

Table 11: Average relative L2 error between zero-shot and fine-tuned models on ε-test set (`test_sol_eps.npy`) showing extra- and interpolation capabilities.

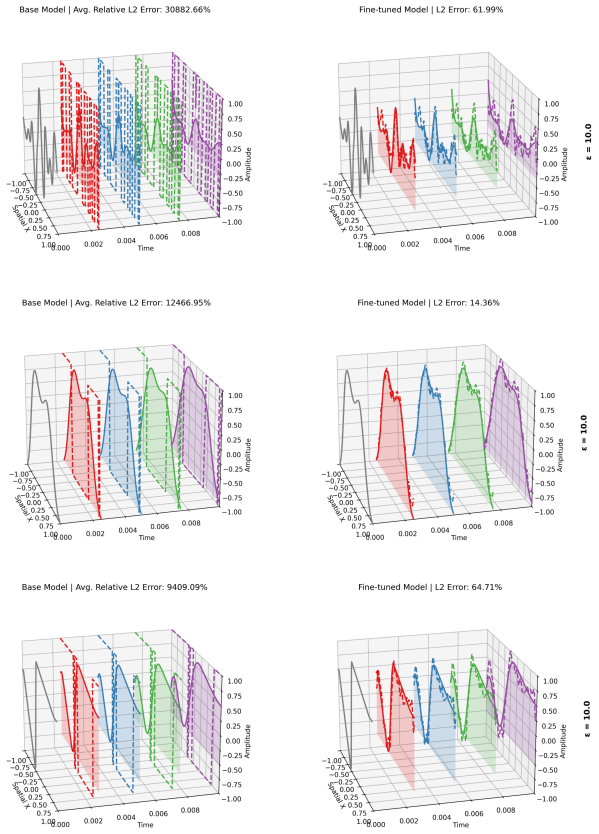


Figure 16: ε-test with all three functions classes. In this plot, we selects the 5-th trajectory of each I.C. type at $\varepsilon = 10.0$, which is an extrapolated parameter value that is hard for our model to approximate. We indicate the ground truth with bold, colored lines, and the predictions using our `AllenCahnFNO` model with dashed, colored lines. For each trajectory, we compared the prediction using base model, as well as with the fine-tuned model. Fine-tuned model shows consistently better performance than the base model, despite the fact that only FiLM layers (0.3% of total parameters) are trained on 20 samples for each dataset.

Observation

To summarize, we developed a lightweight yet effective model with 387,009 parameters for resolving a wide range of physics governed by Allen-Cahn equation (11). The most remarkable results from our model is that by fine-tuning only the FiLM layers that consist of 0.3% of model parameters, we were capable to reduce errors to reasonable range for all initial conditions. Moreover, zero-shot results from our base model are already quite good for most ε values and IC types. **However, our model is currently still limited in several aspects:**

1. Fine-tuning on in-distribution and OOD datasets is not as effective as fine-tuning for the ε test set.
2. We have not evaluated our model for other spatial grid sizes, presumably the inherent weakness of FNO will bring significant performance degrades for coarser grid sizes as it is not ReNO [2].
3. Our approach highly rely on the quality of data for both training and fine-tuning.
4. We did not fully make use of all time series generated, we only leveraged $O(n)$ sample pairs per trajectory

For future development, using more advanced architectural component such as scalable Operator Transformer (scOT) as backbone and training with All2All technique [10] could help us to overcome these challenges.

Bonus

We attempted both bonus tasks from project 1 & 3

⭐ Bonus 1 ⭐

In this section, we consider only the model trained using **All2All** strategy. In Figure 17, we report results at various time points using trained models from task 4 with **direct inference**. We report the **average relative L2 error** across 128 trajectories with the formula (3).

Overall, models trained via different data loading strategies show similar performance over time. However, when smaller timestep sizes are used in direct inference, the All2All model shows better performance with 1.96% and 3.68% of error for results at $t = 0.25$ and $t = 0.5$, about halved comparing to 3.91% and 5.46% of error obtained via the one-to-all model. For $t = 1.0$, the one-to-all model outperforms slightly, we have seen these results from task 4 already. The reason why the one-to-all model outperforms is probably due to the fact that it was targeted to perform mapping from $t = 0.0$ to the final observation time $T = 1.0$, and thus does a better job than the more versatile All2All model here.

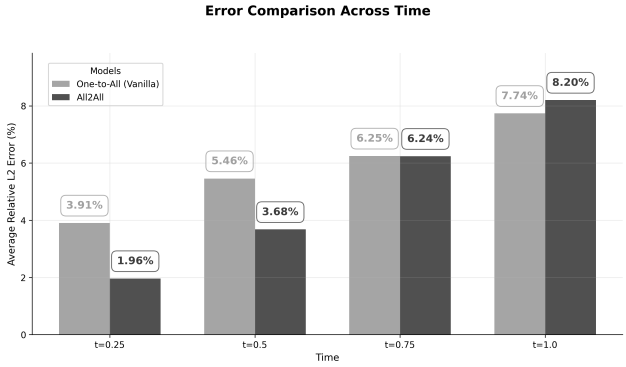


Figure 17: Comparison of average relative L2 errors obtained via time-dependent FNO trained by one-to-all and All2All strategies for 128 trajectories across time, where $t \in \{0.25, 0.5, 0.75, 1.0\}$. The direct inference is used.

For completeness, we report the space-time solution predicted by the All2All model via direct inference for the system of PDEs (44) from $t = 0$ to the final observation time $T = 1.0$. As shown in Figure 18, **our FNO is capable of resolving the nuances of the ground truth across time**, with largest error of only 1×10^{-2} in magnitude. We can also observe that the errors are larger at later time t. This is mainly due to the fact that the results above are obtained via direct inference. When the Δt used in direct inference is large, it may be hard for the model to directly predict the results because the solution deviates more from the initial state.

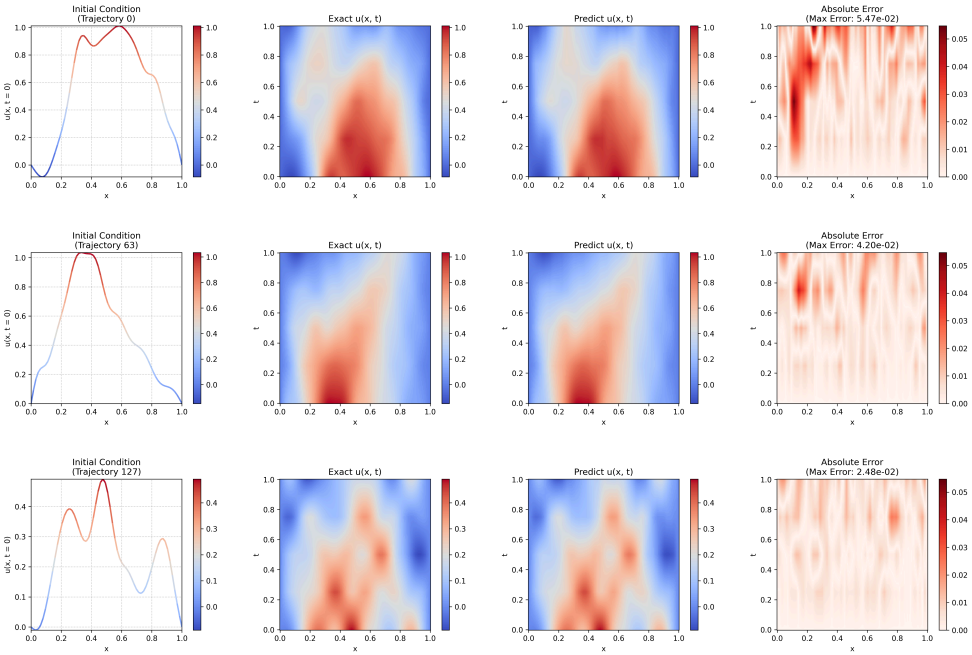


Figure 18: The trajectories with index 0, 63 and 127 in the `test_sol.npy` dataset are reported. The heatmap is plotted with linear interpolation across temporal resolutions. The predictions for intermediate time $t \in \{0.25, 0.5, 0.75, 1.0\}$ are obtained via direct inference using the all2all trained model.

OOD Testing

Next, we tested our model on the OOD dataset. As shown in the table below, both time-dependent models perform worse on the OOD dataset, showing $\mathbf{13.02\%}$ and $\mathbf{14.16\%}$ of error, almost doubled comparing to results 7.74% and 8.20% from the same model respectively from task 4 on the in-distribution dataset. This shows a consistent behavior as we observed before on the One-to-One trained model due to the limit of the FNO architecture itself to correctly capture unseen data.

Nevertheless, the results are not too bad because time-dependent FNOs only show slightly higher errors on the OOD data, compared to the 9.77% of error, which we obtained in task 3 using the plain FNO model, independent of time.

Dataset Type	One-to-One	One-to-All: Direct Inference	All2All: Direct Inference
In-Distribution	5.21% (Task 1)	7.74% (Task 4)	8.20% (Task 4)
OOD	9.77% (Task 3)	**13.02%**	**14.16%**

Table 12: Average relative L2 error for custom FNO implementations with different training strategies, evaluated for the in-distribution and out-of-distribution (OOD) datasets at $t = 1.0$. Both consist of 128 trajectories.

Comparison: Vanilla vs. All2All

Even though All2All strategy is computationally more expensive, but the fact that we can increase the number of samples from $O(k)$ to $O(k^2)$ samples per trajectory makes it attractive to try. We also expect a better performance from the model trained via All2All data loading strategy than the one-to-all (vanilla) strategy accordingly. However, in Table 3 of Task 4, it was shown that one-to-all and All2All exhibit errors of similar magnitude. Moreover, one-to-all strategy even had a lower error. One may ask, is All2All strategy really beneficial?

Recap: Inference Strategy

To better compare our time-dependent FNO models trained on different data loading strategies, we want to analyze the temporal stability of both models when performing **autoregressive rollouts**. Before we dive directly into the question, we would like to recap existing inference strategies and explain what the concept of autoregressive rollout means. In Figure 19, we demonstrate two inference strategies for evaluating our time-dependent model at a desired time t. The red arrow indicates the **direct inference** strategy. By evaluating $\mathcal{G}_\theta(t = 1, u_0)$ directly using the initial distribution $u_0 = u(\cdot, t = 0)$, we can directly obtain the solution with just one model inference. This approach has the advantage that we has low inference cost and there are no accumulative errors. However, it could lead to unstable and false results if there are large changes in the physics of the model problem across time.

On the other hand, the **autoregressive inference** as illustrated with blue arrows requires multiple inferences, but it may provide better predictions for the solution at the final time despite the fact that it also accumulates errors each time it performs an inference. For our specific example, after $k = 4$ successive applications of the trained FNO, we can approximate the solution operator at time $t = 1.0$ by:

$$\mathcal{G}(t = 1, u_0) \approx \mathcal{G}_\theta(t_4 - t_3, \mathcal{G}_\theta(t_3 - t_2, \mathcal{G}_\theta(t_2 - t_1, \mathcal{G}_\theta(t = 0, u_0)))) \tag{12}$$

where we defined $t_i := i\Delta t$ with $i \in \{0, 1, 2, 3, 4\}$, $\Delta t = 0.25$.

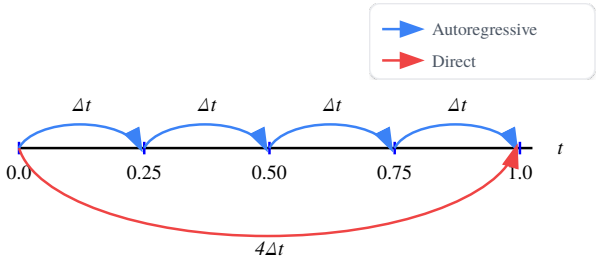


Figure 19: Comparison of different inference strategies

Results

We consider again the `test_sol.npy` dataset that consists of 5 time snapshots. In contrast to the direct inference in which we take $\Delta t = 1.0$ to reach the final time $t = 1.0$ directly, we will now perform autoregressive inference, in which we take several time steps. Each timestep taken is of the size that is some multiple of the base unit $\Delta t = 0.25$.

Timestep Sequence (No. of Δt)	One-to-All (Vanilla)	All2All
1+1+1+1	128.45%	78.45%
1+1+2	116.08%	61.26%
1+2+1	109.51%	62.60%
2+1+1	105.47%	67.04%
2+2	105.26%	54.56%
1+3	75.56%	65.89%
3+1	63.19%	57.04%
4 (Direct)	**7.74%**	**8.20%**

Table 13: The table reports the average relative L2 error at $t = 1.0$ between ground truth and our model predictions. We consider all possible combinations from autoregressive rollouts with $\Delta t = 0.25$, and report errors from two models trained with different data loading strategy, All2All and one-to-all (vanilla) here. The timestep sequence indicates the way how timesteps are taken to reach the final time $t = 1.0$.

In the table above, we reported results obtained from autoregressive evaluation considering all possible timestep combinations. First thing one can confirm is the fact that inference strategies are also of high importance and can significantly affect the quality of results we obtain. Overall, both time-dependent models, regardless of the data loading strategy, exhibit significant weaknesses on performing autoregressive evaluations. This may indicate FNO's weaknesses in capturing the true underlying physics of the model across time.

However, there is still something positive. We can confirm from the autoregressive results that the model trained on All2All strategy performs way better than the model trained on one-to-all strategy. The All2All strategy can indeed help us to maximize usage of temporal data.

Some Personal Thoughts

In fact, we could potentially further extend the All2All strategy in which we also allow timepairs $(t_{\text{in}}, t_{\text{out}})$ to have reversible ordering, meaning that we loosen the constraint and allow for also $t_{\text{in}} > t_{\text{out}}$. This is inspired by the fact that reversibility of the time-stepping method in traditional numerical simulations is a highly desirable property. If we allow our model to approximate solutions by going back-ward in time, we not only have more training data available, but can also potentially make our solution operator structure-preserving.

⭐ Bonus 2 ⭐

Theorem 1 (Allen-Cahn Stability): Let $u \in H^1([0,T];H^{-1}(\Omega)) \cap L^\infty([0,T];H^1(\Omega))$ be a weak solution of the Allen-Cahn equation with $|u| \le 1$ *almost everywhere in* $[0,T]\times\Omega$. Let $\tilde{u} \in H^1([0,T];H^{-1}(\Omega)) \cap L^2([0,T];H^1(\Omega))$ satisfy $|\tilde{u}| \le 1$ *almost everywhere in* $[0,T]\times\Omega$, and $\tilde{u}(0) = \tilde{u}_0, f \in H^1\big([0,T];W^{1,\infty}_{\text{loc}}(\Omega)\big)$ and solve

$$(\partial_t \tilde{u}, v) + (\nabla \tilde{u}, \nabla v) = -\varepsilon^{-2}(f(\tilde{u}), v) \tag{13}$$

for almost every $t \in [0,T]$, *all* $v \in H^1(\Omega)$. *Then we have*

$$\sup_{t\in[0,T]} \|u-\tilde{u}\|^2_{L^2(\Omega)} + \int_0^T \|\nabla(u-\tilde{u})\|^2 dt \le 2\|u_0-\tilde{u}_0\|^2 \exp\big((1+2c_f\varepsilon^{-2})T\big). \tag{14}$$

Hint: Consider the difference $w = u - \tilde{u}$ as a test function and analyze how the nonlinear term contributes to the energy estimate through appropriate bounds on f'.

Theorem 1: Allen-Cahn Stability

Proof.

Assume $u \in H^1([0,T];H^{-1}(\Omega)) \cap L^\infty([0,T];H^1(\Omega))$ is a weak solution of the Allen-Cahn equation (ACE) with $|u| \le 1$ *almost everywhere in* $[0,T]\times\Omega$, and $u(0) = u_0 \in H^1(\Omega)$. We know that for almost every $t \in [0,T]$ and all $v \in H^1(\Omega)$, u fulfills:

$$(\partial_t u, v) + (\nabla u, \nabla v) = -\varepsilon^{-2}(f(u), v) \tag{15}$$

Similarly, by Theorem 6.1 (Existence) in Bartels [11] we know that $\tilde{u}$ as defined in the task description is an approximate solution of ACE and fulfills:

$$(\partial_t \tilde{u}, v) + (\nabla \tilde{u}, \nabla v) = -\varepsilon^{-2}(f(\tilde{u}), v) \tag{16}$$

Now we consider the difference between two weak solutions $w := u - \tilde{u}$. Subtracting equation (16) from equation (15) and by utilizing the linearity of the inner product, we obtain:

$$(\partial_t w, v) + (\nabla w, \nabla v) = -\varepsilon^{-2}(f(u) - f(\tilde{u}), v) \tag{17}$$

By choosing the difference w as the test function $v := w$ in (17) and consider the L^2 inner product yields:

$$\underbrace{(\partial_t w, w)}_{\frac{1}{2}\frac{d}{dt}\|w\|^2_{L^2(\Omega)}} + \underbrace{(\nabla w, \nabla w)}_{\|\nabla w\|^2_{L^2(\Omega)} \ge 0} = -\varepsilon^{-2}(f(u) - f(\tilde{u}), w) \tag{18}$$

The RHS of (18) involves terms associated with the nonlinear function f. We will handle this nonlinearity with the Lipschitz estimate (40). Since we know that the gradient norm of w is strictly non-negative, the following equality holds:

$$\begin{aligned} \frac{1}{2}\frac{d}{dt}\|w\|^2_{L^2(\Omega)} &\le -\varepsilon^{-2}(f(u) - f(\tilde{u}), w) \\ \frac{d}{dt}\|w\|^2_{L^2(\Omega)} &\le -2\varepsilon^{-2}(f(u) - f(\tilde{u}), w) \end{aligned} \tag{19}$$

Now we will derive an important intermediate result which we will use later. Let's focus on the right-hand side: $-2\varepsilon^{-2}(f(u) - f(\tilde{u}), w)$, we can bound it by using the Cauchy-Schwarz inequality on the L^2 inner product and use the Lipschitz estimate (40):

$$\begin{aligned} |-2\varepsilon^{-2}(f(u) - f(\tilde{u}), w)| &\overset{\text{(CSI)}}{\le} 2\varepsilon^{-2}\|f(u) - f(\tilde{u})\|_{L^2(\Omega)}\|w\|_{L^2(\Omega)} \\ &\overset{L^2\text{-norm}}{=} 2\varepsilon^{-2}\left(\int_\Omega \underbrace{|f(u) - f(\tilde{u})|^2 dx}_{(\star)}\right)^{\frac{1}{2}}\left(\int_\Omega |w|^2 dx\right)^{\frac{1}{2}} \\ &\overset{\text{Lipschitz}}{\le} 2\varepsilon^{-2}\left(\int_\Omega c_f|w|^2 dx\right)^{\frac{1}{2}}\left(\int_\Omega |w|^2 dx\right)^{\frac{1}{2}} \\ &\overset{L^2\text{-norm}}{=} 2c_f\varepsilon^{-2}\|w\|^2_{L^2(\Omega)} \end{aligned} \tag{20}$$

here we apply Lipschitz estimate to $(\star)$ pointwise with $|f(u) - f(\tilde{u})| \le c_f|u - \tilde{u}| = c_f|w|$. Note that we have the Lipschitz constant defined as $c_f := \sup_{s\in[-1,1]}|f'(s)|$.

Thus

$$\begin{aligned} \frac{d}{dt}\|w\|^2_{L^2(\Omega)} &\le \big|-2\varepsilon^{-2}(f(u) - f(\tilde{u}), w)\big| \\ &\le 2c_f\varepsilon^{-2}\|w\|^2_{L^2(\Omega)} \end{aligned} \tag{21}$$

Note that (21) is a differential inequality is of the form $\frac{d}{dt}y(t) \le a(t)y(t)$:

$$\frac{d}{dt}\underbrace{\|w\|^2_{L^2(\Omega)}}_{y(t)} \le \underbrace{2c_f\varepsilon^{-2}}_{a=a(t)}\underbrace{\|w\|^2_{L^2(\Omega)}}_{y(t)} \tag{22}$$

Therefore, we can apply the Grönwall's inequality $y(t) \le y(0)\exp\big(\int_0^t a(s)ds\big)$ (42) on (21) and obtain:

$$\underbrace{\|w(t)\|^2_{L^2(\Omega)}}_{u(t)} \le \underbrace{\|w(0)\|^2_{L^2(\Omega)}}_{u(a)|_{a=0}} \cdot \underbrace{\exp\big(2c_f\varepsilon^{-2}t\big)}_{\exp\left(\int_{a=0}^t \beta(s)\,\mathrm{d}s\right)} \tag{23}$$

where $\beta \coloneqq 2c_f\varepsilon^{-2}$ is constant over time.[6]

Recall that we had the equality for the energy estimate (18) with L^2 inner product:

$$\frac{1}{2}\frac{d}{dt}\|w\|^2_{L^2(\Omega)} + \|\nabla w\|^2_{L^2(\Omega)} = -\varepsilon^{-2}(f(u) - f(\tilde{u}), w) \tag{24}$$

We integrate it over time with $\int_0^T \cdot\, dt$:

$$\int_0^T \frac{1}{2}\frac{d}{dt}\|w\|^2_{L^2(\Omega)}dt + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt = \int_0^T -\varepsilon^{-2}(f(u) - f(\tilde{u}), w)dt \tag{25}$$

After some simplification:

$$\frac{1}{2}\big[\|w(t)\|^2_{L^2(\Omega)}\big]_0^T + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt = \int_0^T -\varepsilon^{-2}(f(u) - f(\tilde{u}), w)dt \tag{26}$$

$$\frac{1}{2}\|w(T)\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt = \frac{1}{2}\|w(0)\|^2_{L^2(\Omega)} + \int_0^T -\varepsilon^{-2}(f(u) - f(\tilde{u}), w)dt \tag{27}$$

multiply both sides with a constant factor of 2, we have

$$\|w(T)\|^2_{L^2(\Omega)} + 2\int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt = \|w(0)\|^2_{L^2(\Omega)} + \int_0^T \underbrace{-2\varepsilon^{-2}(f(u) - f(\tilde{u}), w)}_{(\square)}dt \tag{28}$$

we know that we can bound the integrand of the RHS $(\square)$ with (20) and rewrite it into an inequality:

$$\|w(T)\|^2_{L^2(\Omega)} + 2\int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt \le \|w(0)\|^2_{L^2(\Omega)} + \int_0^T 2c_f\varepsilon^{-2}\underbrace{\|w\|^2_{L^2(\Omega)}}_{(\blacktriangle)}dt \tag{29}$$

we can further bound $(\blacktriangle)$ with $\|w(t)\|^2_{L^2(\Omega)} \le \|w(0)\|^2_{L^2(\Omega)} \cdot \exp\big(2c_f\varepsilon^{-2}t\big)$ (23) and obtain:

$$\|w(T)\|^2_{L^2(\Omega)} + 2\int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt \le \|w(0)\|^2_{L^2(\Omega)} + 2c_f\varepsilon^{-2}\|w(0)\|^2_{L^2(\Omega)} \cdot \int_0^T \exp\big(2c_f\varepsilon^{-2}t\big)dt \tag{30}$$

We evaluate the definite integral on the RHS of the inequality:

$$\begin{aligned}\|w(T)\|^2_{L^2(\Omega)} + 2\int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt &\le \|w(0)\|^2_{L^2(\Omega)} + \cancel{2c_f\varepsilon^{-2}}\|w(0)\|^2_{L^2(\Omega)} \cdot \frac{1}{\cancel{2c_f\varepsilon^{-2}}} \cdot \big[\exp\big(2c_f\varepsilon^{-2}t\big)\big]_{t=0}^{t=T} \\ &\le \|w(0)\|^2_{L^2(\Omega)} + \|w(0)\|^2_{L^2(\Omega)} \cdot \big(\exp\big(2c_f\varepsilon^{-2}T\big) - 1\big) \\ &\le \|w(0)\|^2_{L^2(\Omega)} \exp\big(2c_f\varepsilon^{-2}T\big))\end{aligned} \tag{31}$$

Thus we have

$$\|w(T)\|^2_{L^2(\Omega)} + 2\int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt \le \|w(0)\|^2_{L^2(\Omega)} \cdot \exp\big(2c_f\varepsilon^{-2}T\big) \tag{32}$$

To match the inequality as in the theorem, we subtract $\int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt$ on both sides and obtain:

$$\|w(T)\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt \le \|w(0)\|^2_{L^2(\Omega)} \cdot \exp\big(2c_f\varepsilon^{-2}T\big) - \underbrace{\int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt}_{\ge 0} \tag{33}$$

we again loosen the bound by replacing the subtraction with strictly positive terms

$$\begin{aligned}\|w(T)\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt &\le \|w(0)\|^2_{L^2(\Omega)} \cdot \exp\big(2c_f\varepsilon^{-2}T\big) + \underbrace{\|w(0)\|^2_{L^2(\Omega)} \cdot \exp(T)}_{\ge 0} \\ \|w(T)\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt &\le \|w(0)\|^2_{L^2(\Omega)} \underbrace{\big(\exp\big(2c_f\varepsilon^{-2}T\big) + \exp(T)\big)}_{\text{use Young's inequality}}\end{aligned} \tag{34}$$

from Young's inequality we know that $\exp(T) + \exp\big(2c_f\varepsilon^{-2}T\big) \le 2\exp\big(T + 2c_f\varepsilon^{-2}T\big)$:

$$\|w(T)\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt \le 2\|w(0)\|^2_{L^2(\Omega)} \exp\big(\big(1 + 2c_f\varepsilon^{-2}\big)T\big) \tag{35}$$

Since this bound holds for any T, it also holds for the supremum over $[0, T]$, giving us:

$$\sup_{t\in[0,T]} \|w\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w\|^2_{L^2(\Omega)}dt \le 2\|w(0)\|^2_{L^2(\Omega)} \cdot \exp\big(\big(1 + 2c_f\varepsilon^{-2}\big)T\big) \tag{36}$$

By plugging the definition of the difference $w = u - \tilde{u}$ back to the equation, we have:

$$\sup_{t\in[0,T]} \|u - \tilde{u}\|^2_{L^2(\Omega)} + \int_0^T \|\nabla(u - \tilde{u})\|^2_{L^2(\Omega)}dt \le 2\|u_0 - \tilde{u}_0\|^2_{L^2(\Omega)} \cdot \exp\big(\big(1 + 2c_f\varepsilon^{-2}\big)T\big) \tag{37}$$

[6] Note the exponential term at the RHS comes from the fact that $\beta \coloneqq 2c_f\varepsilon^{-2}$ are constant:

$\exp\big(\int_{a=0}^t \beta(s)\,\mathrm{d}s\big) = \exp\big(\int_{a=0}^t 2c_f\varepsilon^{-2}\,\mathrm{d}s\big) = \exp\Big(\big[2c_f\varepsilon^{-2}s\big]_0^t\Big) = \exp\big(2c_f\varepsilon^{-2}t\big)$ with $t \in [0, T]$

■

Used Theorems & Lemmas

For the proof the theorems and lemmas referenced above are taken from from Bartels [11] chapter 6 and Hiptmair [12].

Theorem 2 (Cauchy-Schwarz Inequality): If a is a symmetric positive semi-definite bilinear form on the real vector space V, then

$$|a(u,v)| \leq a(u,u)^{\frac{1}{2}} a(v,v)^{\frac{1}{2}} \tag{38}$$

Theorem 2: Cauchy-Schwarz Inequality

Corollary 2.1 (Cauchy-Schwarz for L^2 inner product):

$$\left| \int_\Omega u(\boldsymbol{x}) v(\boldsymbol{x}) d\boldsymbol{x} \right| \leq \left(\int_\Omega |u(\boldsymbol{x})|^2 d\boldsymbol{x} \right)^{\frac{1}{2}} \left(\int_\Omega |v(\boldsymbol{x})|^2 d\boldsymbol{x} \right)^{\frac{1}{2}} = \|u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} \tag{39}$$

Corollary 3: Cauchy-Schwarz for L^2 inner product

Lemma 3 (Lipschitz Estimate): Let $c_f := \sup_{s \in [-1,1]} |f'(s)|$, given $u, \tilde{u}$ being solutions with $|u|, |\tilde{u}| \leq 1$ almost everywhere in $[0,T] \times \Omega$, then we have:

$$|f(u) - f(\tilde{u})| \leq c_f |u - \tilde{u}| \tag{40}$$

Lemma 4: Lipschitz Estimate

Lemma 4 (Grönwall):

The Grönwall's lemma states that if a nonnegative function $y \in C([0,T])$ satisfies

$$y(T') \leq A + \int_0^{T'} a(t) y(t) dt, \forall T' \in [0,T] \tag{41}$$

with a nonnegative function $a \in L^1([0,T])$, then we have

$$y(T') \leq A \exp\left(\int_0^T a dt \right) \tag{42}$$

Lemma 5: Grönwall

Appendix

The appendix is where we keep all the fascinating details that deserve attention.

Appendix Project 1 (P1)

In this project, we consider the one-dimensional wave equation:

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = f(x,t) \tag{43}$$

We seek for the scalar field $u : \tilde{\Omega} \mapsto \mathbb{R}$ defined on the spatio-temporal domain $\tilde{\Omega} \coloneqq \Omega \times (0,T]$, with the final observation time at $T = 1$. The spatial domain spans over $\Omega = [0,1]$ with domain boundary $\partial\Omega = \{0,1\}$. In particular, we consider the special case when the equation is homogenous ($f \equiv 0$) and the superposition principle holds. By incorporating suitable initial and boundary conditions, we can formulate the following initial boundary value problem (IBVP):

$$\begin{cases} \frac{\partial^2 u}{\partial t^2} &= c^2 \frac{\partial^2 u}{\partial x^2} \quad (x,t) \in \Omega \times (0,T] \\ u(x,0) &= u_0(x) \quad x \in \Omega \\ \frac{\partial}{\partial t} u(x,0) &= 0 \quad x \in \Omega \\ u(x,t) &= 0 \quad (x,t) \in \partial\Omega \times (0,T] \end{cases} \tag{44}$$

Appendix P1.A: Neural Operator for Solving 1D Wave IBVP

We can approach the problem (44) in a generic setting, in which we consider the solution operator $\mathcal{G}$ to the system of PDEs (44). It is an infinite-dimensional mapping between some function spaces defined on bounded domain Ω and maps from the initial condition u_0 at the final observation time $u(t=1)$:

$$\mathcal{G} : u_0 \mapsto u(\cdot, t=1) \tag{45}$$

We aim to approximate $\mathcal{G}$ by constructing a parametric map $\mathcal{G}_\theta$ with parameters from the finite-dimensional space $\theta \in \mathbb{R}^p$ such that $\mathcal{G}_\theta \approx \mathcal{G}$. The approximation can be done by using a **neural operator** architecture and leverage the power of **operator learning** [13]. We write the generic form of such multi-layer operator learning architecture as follows:

$$\mathcal{G}_\theta \coloneqq \mathcal{G}_\theta^{(L)} \circ \mathcal{G}_\theta^{(L-1)} \circ ... \mathcal{G}_\theta^{(1)} \tag{46}$$

The architecture is a composition of linear and nonlinear operators $\mathcal{G}_\theta^{(l)}$ between function spaces. Each hidden layer $\mathcal{G}_\theta^{(l)}$ consists of a local linear map $W^{(l)}$, a non-local kernel-integral operator $K^{(l)}$, a bias function $b^{(l)}$, and a nonlinear activation function σ as in the following form:

$$\mathcal{G}_\theta^{(l)} \coloneqq \sigma\big(W^{(l)} + K^{(l)} + b^{(l)}\big), \text{where } l = 1, ..., L \tag{47}$$

Appendix P1.B: Fourier Neural Operator (FNO)

The FNO architecture is of the generic form as in equation (46) but with an additional add-on that it also includes a lifting layer P and a projection layer Q as follows:

$$\mathcal{G}_\theta \coloneqq \underbrace{Q}_{\text{Projecting}} \circ \mathcal{G}_\theta^{(L)} \circ \mathcal{G}_\theta^{(L-1)} \circ ... \mathcal{G}_\theta^{(1)} \circ \underbrace{P}_{\text{Lifting}} \tag{48}$$

The additional operators P and Q are used to lift the parameters into the large latent space and project them back to the physical space in the last forward pass. The key difference that distinguishes the FNO from other neural operators is the fact that the kernel-integral operator $K^{(l)}$ in each generic hidden-layer (47) is constrained to a translation-invariant convolution operation in the Fourier space:

$$K^{(l)}(u) \coloneqq \mathcal{F}^{-1}\big(R^{(l)} \circ \mathcal{F}\big)(u) \tag{49}$$

where $\mathcal{F}, \mathcal{F}^{-1}$ are the Fourier and inverse Fourier transforms respectively. In the discrete implementation, we learn the weights directly in the Fourier domain through the matrix $R^{(l)}$, and we use F, F^{-1} for discrete Fourier transform (DFT) and its inverse.

Learnable Parameters

The complete set of learnable parameters in the FNO architecture can be represented as:

$$\theta = \big[W^{(l)}, R^{(l)}, P, Q\big] \tag{50}$$

where $W^{(l)}$ represents the weights in the local linear map for residual connections, $R^{(l)}$ appears in the Fourier layer (49) as the complex Fourier multiplication matrix at each layer l. P, Q

Operator Learning

The mapping $\mathcal{G}_\theta$ that is parameterized by the set of learnable FNO parameters (50) can be learned by approximating the solution operator $\mathcal{G}$ from observations:

$$\Big\{u_0^{(i)}, \mathcal{G}\Big(u_0^{(i)}\Big)\Big\}_{i=1}^N \tag{51}$$

where $u_0^{(i)} \sim \mu, \forall 1 \le i \le N$ is an i.i.d. sequence drawn from some probability measure μ on the function space. In such discretized setting, we aim to minimize the following L2-norm:

$$J(\theta) \coloneqq \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} w_j \left\| \underbrace{\mathcal{G}\Big(u_0^{(i)}\Big)(y_j)}_{\text{Ground Truth}} - \underbrace{\mathcal{G}_\theta\Big(u_0^{(i)}\Big)(y_j)}_{\text{Approximation}} \right\| \tag{52}$$

For each discretization point y_j and quadrature weight w_j, we compute the L2-norm difference between the ground truth solution $\mathcal{G}\Big(u_0^{(i)}\Big)(y_j)$ and our neural operator approximation $\mathcal{G}_\theta\Big(u_0^{(i)}\Big)(y_j)$. Here, j indexes over M spatial domain points and i over N function space samples, with the loss averaged over all samples.

Appendix P1.C: One-to-One Training Strategy

We used the provided `train_sol.npy` dataset for training, which consists of 128 trajectories at all 5 time snapshots. Each trajectory at a time snapshot is reported by 64 equally-spaced nodal values that represent the solution of the system of PDEs (44) at a given time t. However, for **One-to-One training**, we will **only make use of the time snapshots at $t = 0$ and $t = 1.0$** to train our model for predicting $u_0 \mapsto u(\cdot, t = 1)$. We split the 128 trajectories into 64 trajectories for training, and the remainder 64 trajectories is kept to perform cross-validation. For convenience, we provide a `OneToOne`[7] class that provides data in form as we discussed above directly. It inherits from the abstract `Dataset` class from the `torch.utils.data` module.

In order to align with the nature of the wave equation (43) that is second-order in time, we provide both initial conditions explicitly by passing the initial velocity $v_0 := u_t(x, 0) = 0$ to our model in a separate input channel.[8] In Table 2, we listed out the relevant parameters used in the training process, we opted for an AdamW optimizer as it performed the best throughout testings. `StepLR` scheduler is used with multiplicative factor of learning rate decay `gamma = 0.1`.

Training Samples	Validation Samples	Early stopping	Maximal Number of Epochs
64	64	After 80 epochs	800
Batch Size	**Step size**	**Learning Rate**	**Optimizer**
5	300	$1 \cdot 10^{-3}$	AdamW

Table 14: Training configuration for FNO.

Figure 20: Training history of our FNO model when providing both u_0, v_0 by passing it to a separate channel.

Appendix P1.D: Super-Resolution Samples

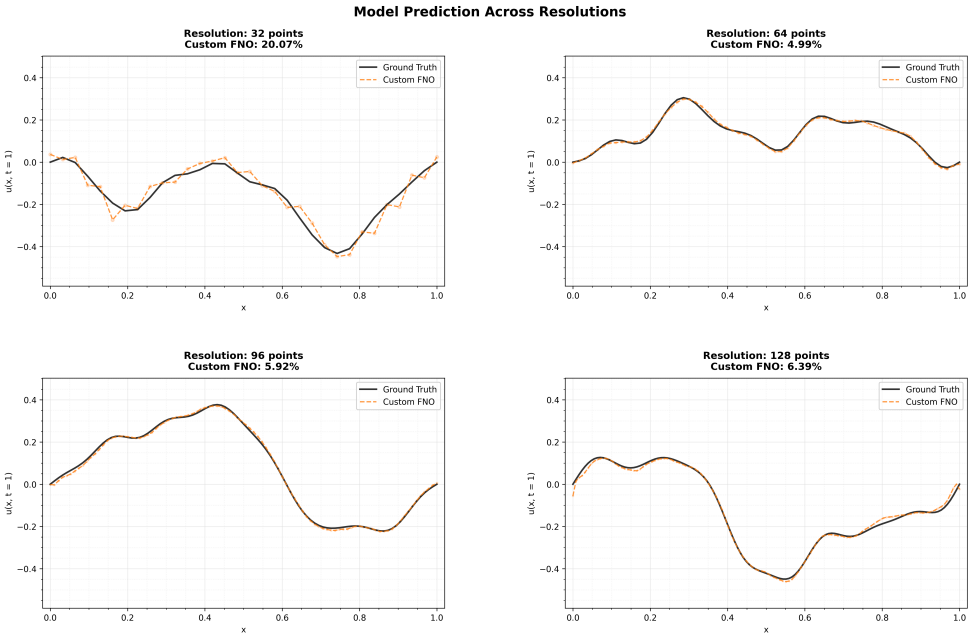


Figure 21: For completeness, we reported the model prediction for first trajectory of each `test_sol_res_{s}.npy` data set. (Task 2)

[7] locates under `project_1/dataset.py`

[8] For task 1-3, whether we explicitly provide v_0 does not make a big difference. However, for All2All training we find it to be necessary. Thus, we decide to stay consistent and use models with explicit v_0 throughout the report.

Appendix P1.E: Out-of-Distribution Generalization

Recall our implementation approach to resolve the 1D wave equation with its associated evolution problem: we considered the solution operator $\mathcal{G}$ that maps between infinite-dimensional function spaces, and approximated it with the trained Fourier Neural Operator (FNO), which we denoted as $\mathcal{G}_{\#\mu} \coloneqq \mathcal{G}_\theta$ in this task.

The aim is to learn $\mathcal{G}_{\#\mu}$ by approximating the solution operator $\mathcal{G}$ from observations:

$$\left\{u_0^{(i)}, \mathcal{G}\left(u_0^{(i)}\right)\right\}_{i=1}^{N} \tag{53}$$

where $u_0^{(i)} \sim \mu, \forall 1 \leq i \leq N$ is an i.i.d. sequence drawn from some probability measure μ on the function space.

In-Distribution Testing

So far, we have examined our operator $\mathcal{G}_{\#\mu}$, that has been trained on random testing samples drawn from μ on both task 1 and 2 on datasets `test_sol.npy` and `test_sol_res_{s}.npy` for $s \in \{32, 64, 96, 128\}$. These are in-distribution datasets, meaning that the initial data distribution of the test data was chosen as $u_0 \sim \mu$, which has been seen by $\mathcal{G}_{\#\mu}$ during pretraining. Our model has demonstrated in both tasks its ability to both resolution on the same spatial resolution as it was trained on, as well as to do super-resolution.

Out-of-Distribution (OOD) Testing

One question that may arise with using in-distribution dataset for testing is whether $\mathcal{G}_{\#\mu}$ has actually learned the underlying physics, instead of learning the particular data distribution μ that was trained on. To examine this, we utilize the OOD dataset `test_sol_OOD.npy`. Here we consider another input measure v, supported on the same function space, and choose the OOD initial data $v \sim v$ accordingly, such that $v \approx \mu$ but $v \neq \mu$.

Appendix P1.F: Training Strategy - Vanilla vs. All2All

In our implementation of the `All2All` class for loading training dataset, we provide two possible training strategies in relation to the utilization of temporal data. The first one is the **one-to-all training (vanilla)** where the aim is to approximate the solution operator of the system (44) for some final observation time t_{out}, where the system evolves from an initial distribution $u_0 = u(\cdot, t = 0)$. In this strategy, the starting time is fixed to $t = 0$ but the end time can vary.

An alternative that can better make use of the training data is the **All2All strategy** as proposed in Herde et al. [10]. In this strategy, both the starting time as well as the end time can be chosen freely. As long as it preserves the ordering of time $t_{\text{in}} \leq t_{\text{out}}$. It boils down to train for the following mapping with some fixed timestep size Δt, starting from t_{in} such that $t_{\text{out}} = t_{\text{in}} + \Delta t$. The system evolves from some intermediate distribution $u_{t_{\text{in}}}$:

$$\mathcal{G}_\theta : u_{t_{\text{in}}} \mapsto u(\cdot, t_{\text{in}} + \Delta t) \tag{54}$$

Approximating Velocities with Finite Difference

⚠️ Through trial and error, we found out that it is very essential in **All2All** training strategy to ensure that the velocity at the input time $v_{t_{\text{in}}}$ is explicitly provided to some input channel of the model. When using **One-to-One** or **one-to-all** training, the model can implicitly learn that $v_0 = 0$ if we consistently provide input data from the same starting point $t = 0$. However, when performing **All2All** training our input distribution $u(\cdot, t)$ may not necessarily be at any intermediate time point that the dataset spans over, and we need to explicitly provide the data such that our FNO can learn the underlying function of $u_t(\cdot, t)$. Therefore, for consistency and to better align with the nature of the underlying system, we approximate the velocities $v_i \coloneqq u_t(\cdot, t_i)$ for all $i \in \{0, 1, 2, 3, 4\}$ for 5 time snapshots with finite difference. The approximated velocities v_i are passed to an input channel of our FNO, together with u_i, Δt and an array representing the 1D discretization of the domain Ω.

Appendix Project 2 (P2)

In this project, we implemented the **PDE Functional Identification of Nonlinear Dynamics (PDE-Find)** algorithm [4], which discovers the relevant terms of the governing PDEs using sparse regression. It is an extension of the SINDy algorithm [14] in which PDE-Find considers also partial derivatives as part of its feature library. Within the scope of this report, we aim to recover the symbolic expression from the solution u of evolution problems, where the physics is governed by some unknown PDE(s). The spatiotemporal domain is defined as $\tilde{\Omega} := \Omega \times (0, T)$, where T is some final observation time.

Appendix P2.A: Selecting Library Candidates

Here we supplement information about the detailed selection of candidates used for building the library $\Theta \in \mathbb{R}^{(n \cdot m) \times D}$ for each system. We denote D as the size of the library, it equals the number of candidates in the library. n denotes the number of spatial points and m the number of time points on the $n \times m$ equidistant spatio-temporal grid $\tilde{\Omega}$. During our implementation, we found the supplementary materials of the original paper [4] to be fruitful and recognized that the provided data may be part of it. Moreover, by visualizing on the original data set, we also could identify the first system to emerge from Burgers' equation, second system to be the Korteweg–De Vries equation (KdV equation) and the third system to depict the reaction-diffusion process. Thus, during our experiments, we did not fully blindly select the candidates, but followed a Keplerian paradigm, **combining observations and assumptions with our modeling**.

System 1

- Size of library $D = 31$, we used 31 candidates to assemble the library Θ
- Generalized condition number of Θ is 653973.8125. Found optimal tolerance: 0.080.
- Domain consists of n = 256 spatial points and m = 101 time snapshots ($n \times m = 256 \times 101 = 25856$)
  - $\Theta \in \mathbb{R}^{25856 \times 31}$
  - $u_t \in \mathbb{R}^{25856 \times 1}$

$$\begin{aligned} \operatorname{col}(\Theta) := \operatorname{span}\{&1, u, u^2, u^2u_x, u^2u_x^2, u^2u_x^3, u^2u_{xx}, u^2u_{xx}^2, u^2u_{xx}^3, \\ &u^3, u^3u_x, u^3u_x^2, u^3u_x^3, u^3u_{xx}, u^3u_{xx}^2, u^3u_{xx}^3, \\ &uu_x, uu_x^2, uu_x^3, uu_{xx}, uu_{xx}^2, uu_{xx}^3, \\ &u_x, u_x^2, u_x^3, u_{xx}, u_{xx}^2, u_{xx}^3, \\ &u_{xxx}, u_{xxx}^2, u_{xxx}^3\} \end{aligned} \tag{55}$$

System 2

- Size of library $D = 13$, we used 13 candidates to assemble the library Θ
- Generalized condition number of Θ is 3415.815185546875. Found optimal tolerance: 0.500.
- Domain consists of n = 512 spatial points and m = 201 time snapshots ($n \times m = 512 \times 201 = 102912$)
  - $\Theta \in \mathbb{R}^{102912 \times 13}$
  - $u_t \in \mathbb{R}^{102912 \times 1}$

$$\Theta(u) := \begin{bmatrix} 1 & u & uu_t & uu_{tt} & uu_x & uu_{xx} & uu_{xxx} & u_tu_x & u_{tt} & u_{tt}u_{xx} & u_x & u_{xx} & u_{xxx} \end{bmatrix} \tag{56}$$

System 3

- Size of library $D = 23$, we used 23 candidates to assemble the library Θ
- Domain consists of $n = 256 \times 256 = 65536$ spatial points and $m = 201$ time snapshots ($n \times m = 65536 \times 201 = 13172736$)
  - $\Theta \in \mathbb{R}^{13172736 \times 13}$
  - $u_t \in \mathbb{R}^{13172736 \times 1}$

$$\begin{aligned} &\Theta(u, v) \\ &:= \begin{bmatrix} u & u_x & u_y & u_{xx} & u_{yy} & u_{xy} & u^4 & u^3 & u^2 & v & v_x & v_y & v_{xx} & v_{yy} & v_{xy} & v^4 & v^3 & v^2 & uv & uv^2 & u^2v & uv^3 & u^3v \end{bmatrix} \end{aligned} \tag{57}$$

Note that in system 3, we apply the sampling operator $\mathcal{C}$ on the candidate library Θ and the associated rows from RHS ξ and solve for a smaller least squares problem as follows:

$$\arg\min_{\xi \in \mathbb{R}^D} \|\mathcal{C}(u_t) - \mathcal{C}(\Theta)\xi\|_2^2 + \lambda \|\xi\|_2^2 \tag{58}$$

- Generalized condition number of $\mathcal{C}(\Theta)$ is 101.93910217285156.
- Found optimal tolerance for u_t: 0.098; optimal tolerance for v_t: 0.107
- $\mathcal{C}(\Theta) \in \mathbb{R}^{10000 \times 23}$
- $\mathcal{C}(u_t) \in \mathbb{R}^{10000 \times 1}$
- $\mathcal{C}(v_t) \in \mathbb{R}^{10000 \times 1}$

Appendix P2.B: Sparse Regression Algorithms

The following algorithms can be found in the supplementary materials of the original PDE-Find paper [4]. Here we include them in this appendix section just for completeness. There are a series of hyperparameters one could tune when using both algorithms, in the main report one can found the values used for each system in its associated parameter table.

Algorithm 1: STRidge (Sequential Threshold Ridge Regression)

function STRidge(Θ, U_t, λ, tol, iters)
  $\hat{\xi} \leftarrow \arg\min_{\xi} \|\Theta\xi - U_t\|_2^2 + \lambda\|\xi\|_2^2$
  bigcoeffs $\leftarrow \left\{j : |\hat{\xi}_j| \geq \text{tol}\right\}$
  $\hat{\xi}[\sim \text{bigcoeffs}] \leftarrow 0$
  $\hat{\xi}[\text{bigcoeffs}] \leftarrow \text{STRidge}(\Theta[:, \text{bigcoeffs}], U_t, \text{tol}, \text{iters} - 1))$
  return $\hat{\xi}$

Algorithm 2: TrainSTRidge

In algorithm 2, we scale the hyperparameter η proportionally to the condition number of the matrix Θ. In order to better target the linear least squares problems in the scope of our project, we use the generalized condition number of a matrix for monitoring the sensitivity of the systems to be resolved as in the lecture note by Hiptmair [12]. Given $A \in \mathbb{K}^{m,n}, m \geq n, \text{rank}(A) = n$, we define the **generalized Euclidean condition number** as:

$$\text{cond}_2(A) := \sqrt{\frac{\lambda_{\max}(A^H A)}{\lambda_{\min}(A^H A)}}. \tag{59}$$

where $\lambda_{\min}(A)$ represents the smallest eigenvalue of some matrix A, and $\lambda_{\max}(A)$ the largest eigenvalue on the contrary.

function TrainSTRidge(Θ, U_t, λ, d_{tol}, tol_iters, STR_iters)
  $\{\Theta^{\text{train}}, \Theta^{\text{test}}\} \leftarrow \{80/20 \text{ split of } \Theta\}$
  $\{U_t^{\text{train}}, U_t^{\text{test}}\} \leftarrow \{80/20 \text{ split of } U_t\}$
  $\eta \leftarrow 10^{-3}\kappa(\Theta)$
  $\xi^{\text{best}} \leftarrow (\Theta^{\text{train}})^{-1} U_t^{\text{train}}$
  $\text{error}_{\text{best}} \leftarrow \|\Theta^{\text{test}}\xi^{\text{best}} - u_t^{\text{test}}\|_2^2 + \eta \, \|\xi^{\text{best}}\|_0$
  $\text{tol} \leftarrow \text{dtol}$
  for $\text{iter} = 1, ..., \text{tol\_iters}$ **do**
    $\xi \leftarrow$ **TrainSTRidge** $(\Theta^{\text{train}}, U_t^{\text{train}}, \lambda, \text{tol}, \text{STR\_iters})$
    if $\text{error} \leq \text{error}_{\text{best}}$ **then**
      $\text{error}_{\text{best}} \leftarrow \text{error}$
      $\xi_{\text{best}} \leftarrow \xi$
      $\text{tol} \leftarrow \text{tol} + d_{\text{tol}}$
    else
      $\text{tol} \leftarrow \max([0, \text{tol} - 2d_{\text{tol}}])$
      $d_{\text{tol}} \leftarrow \frac{2d_{\text{tol}}}{\text{tol\_iters} - \text{iter}}$
      $\text{tol} \leftarrow \text{tol} + d_{\text{tol}}$
  return ξ^{best}

Appendix P2.C: Example Library Candidates

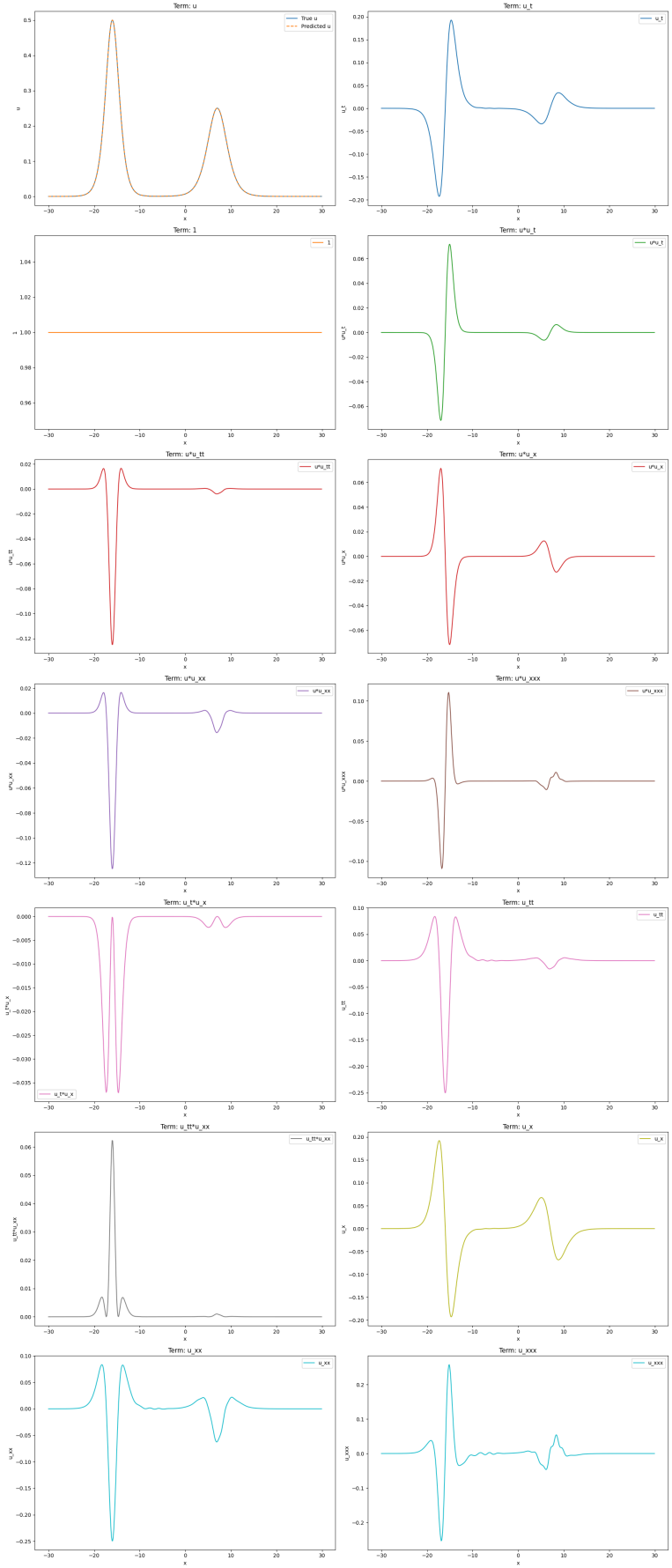


Figure 22: Visualization of u_t and all candidates used for building Θ of system 2. Computation of derivatives is done via automatic differentiation on a trained FNN for system 1 & 2. For system 3 direct finite difference with `torch.gradient` is used.

Appendix Project 3 (P3)

Appendix P3.A: Function Space Specifications

For the dataset generation, we ensured the periodic boundary condition is always enforced and the generated data span over x-axis from −1 to 1 and the nodal values $u_0 \in [-1, 1]$ as well. In all, we used the following three function classes for data generation. We distinguish between default samplers (used for `train_sol.npy`, `test_sol.npy` and `test_sol_eps.npy`) and special samplers (used for `test_sol_OOD.npy`).

Piecewise Linear Functions (PL)

Let $[a, b]$ be a closed interval and $\{x_0, x_1, ..., x_k\}$ be a partition where $a = x_0 < x_1 < ... < x_k = b$. A piecewise linear function $f : [a, b] \to \mathbb{R}$ is defined as:

$$f(x) = \begin{cases} \alpha_1 x + \beta_1 x \in [x_0, x_1] \\ \vdots \\ \alpha_k x + \beta_k x \in [x_{k-1}, x_k] \end{cases} \tag{60}$$

where $\alpha_i, \beta_i \in \mathbb{R}$ and continuity is enforced at breakpoints: $\alpha_i x_i + \beta_i = \alpha_{i+1} x_i + \beta_{i+1}$ for $i = 1, ..., k - 1$.

For **default samplers**, we set number of breakpoints to 4 and for the **OOD dataset** `test_sol_OOD.npy` we increased it to 6. It is worth to note that some initial data from the PL function class shows spurious oscillations during the solution process. This indicates that one needs to extra care when selecting proper training data for piecewise linear functions in the future to avoid learning solutions with inherent approximation errors in themselves.

Fourier Series (FS)

The Fourier series for an L-periodic function on $[0, L]$ is similarly given by:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos\left(\frac{2\pi k x}{L}\right) + b_k \sin\left(\frac{2\pi k x}{L}\right) \right), \tag{61}$$

with coefficients a_k and b_k given by $a_k = \left(\frac{2}{L}\right) \int_0^L f(x) \cos\left(\frac{2\pi k x}{L}\right) dx$ and $b_k = \left(\frac{2}{L}\right) \int_0^L f(x) \sin\left(\frac{2\pi k x}{L}\right) dx$.

Instead of taking an infinite summation, we used only 10 Fourier modes for **default samplers**, and for the **OOD dataset** `test_sol_OOD.npy` we increased the Fourier modes used to 20 in order to explore our model performance on high-frequency data.

Gaussian Mixture Model (GM)

For the GM class, we obtain the initial condition through the weighted sum of Gaussian components such that $\sum_{i=1}^{K} w_i \overset{!}{=} 1$:

$$u(x, 0) = \sum_{j=1}^{K} w_j \overbrace{\mathcal{N}(\boldsymbol{x}; \mu_j, \sigma_j)}^{\text{probability of j-th Gaussian}} \tag{62}$$

The number of Gaussian components K is determined randomly between 2 - 5 for **default sampler**. For the **OOD dataset** `test_sol_OOD.npy`, K is fixed to 10. We separated the interval depending on the number of components are used in the sample and then uniformly randomly draw mean μ and σ within the range of the subinterval.

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2} \right) \tag{63}$$

Appendix P3.B: Pitfalls of Curriculum Learning

We explored different training strategies for our Allen-Cahn model, with a particular focus on curriculum learning [9]. For curriculum trainings, we divided the first half of the training period into equal segments based on our ε values, introducing them sequentially from largest to smallest. On the contrary, for anti-curriculum training, we introduce new ε values by decreasing difficulty.

The **default training method**, which exposes the model to all ε values at once from the beginning, **obviously outperforms both curriculum alternatives** in our experimental results. The default method exhibits **smooth convergence** and a lower final validation loss as shown in Figure 23. Performance instability is evident in the *curriculum learning technique* as seen in Figure 24, with **noticeable jumps in loss whenever new ε values are introduced**. Similarly, bad performance was shown by the *anti-curriculum technique* in Figure 25, which started with the most difficult small ε values and showed extremely unstable training dynamics.

This comparison of validation loss of different training strategies indicates that the complexity of curriculum learning does not significantly improve the Allen-Cahn system over ordinary training techniques. **Default training strategy exhibits best convergence** in minimizing validation errors.

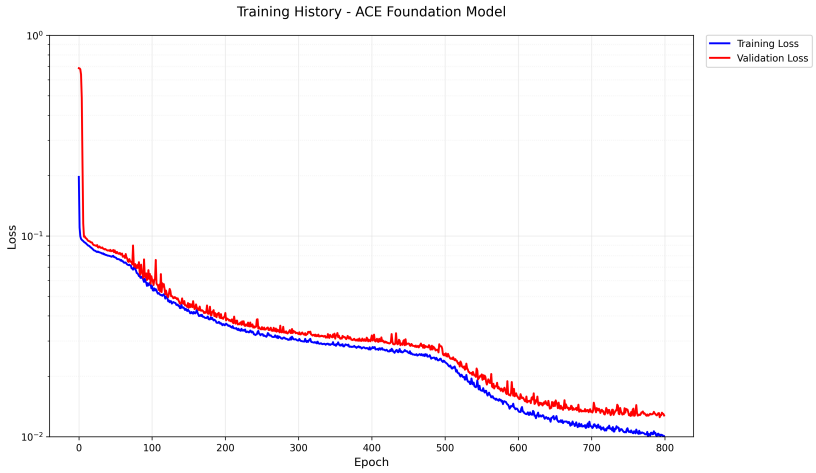


Figure 23: Default training strategy

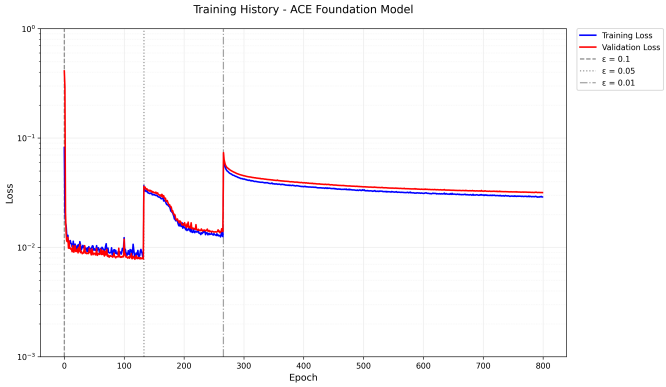


Figure 24: Training with curriculum learning (easiest samples first)

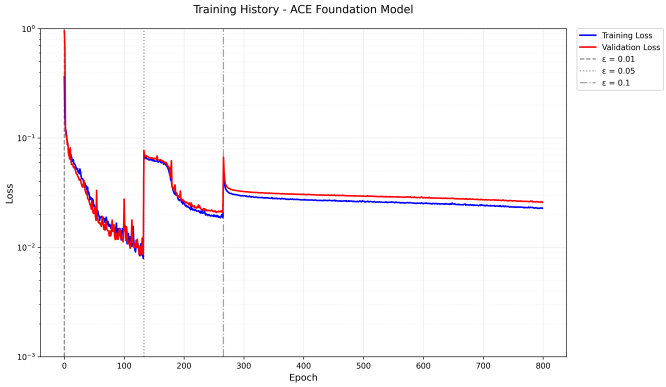


Figure 25: Training with anti-curriculum (hardest samples first)

Appendix P3.C1: In-Distribution Test Results (`test_sol.npy`)

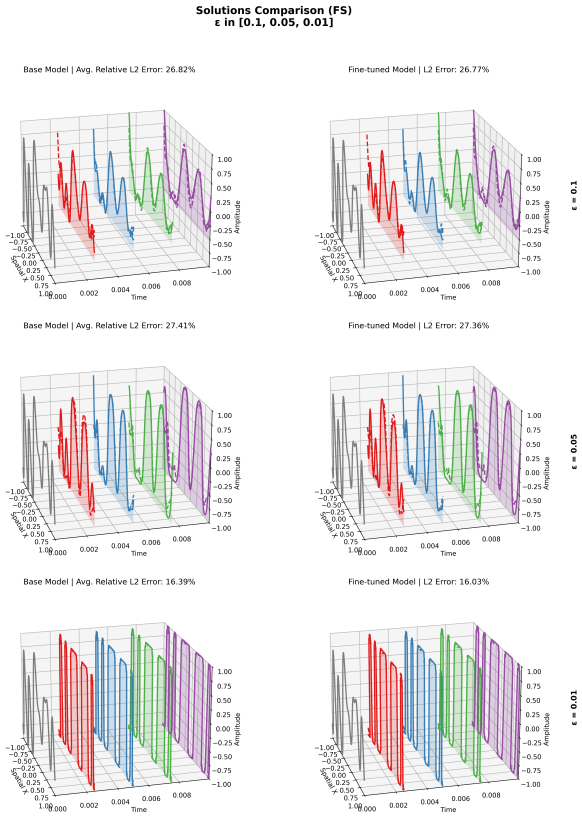


Figure 26: Fourier Series (FS)

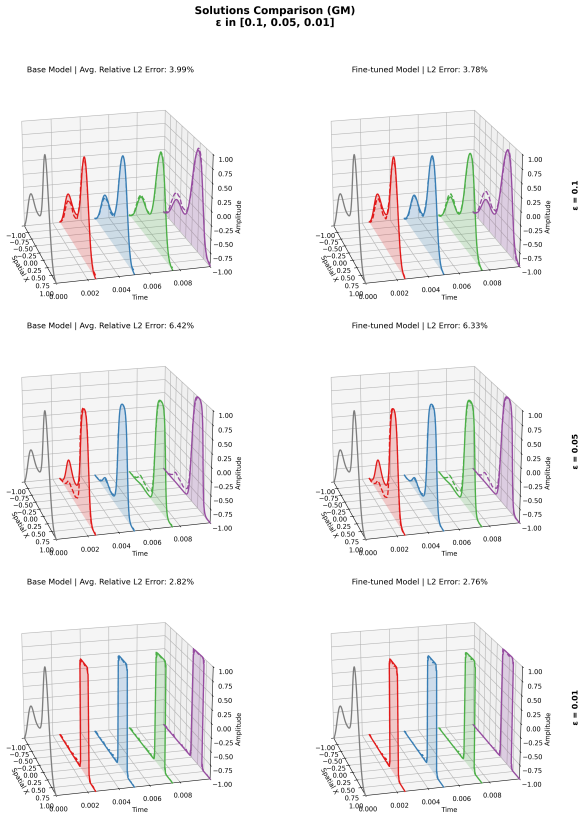


Figure 27: Gaussian Mixture (GM)

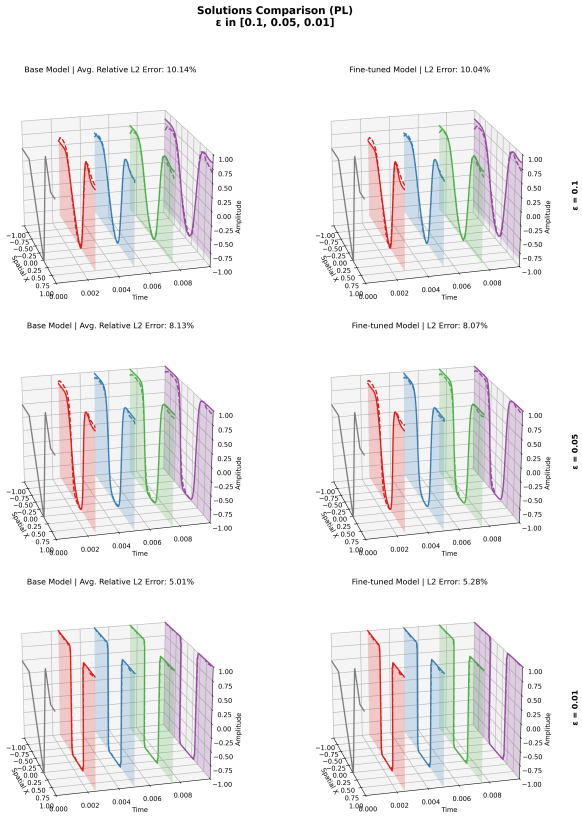


Figure 28: Piecewise Linear (PL)

Appendix P3.C2: Out-of-Distribution Test Results (`test_sol_OOD.npy`)

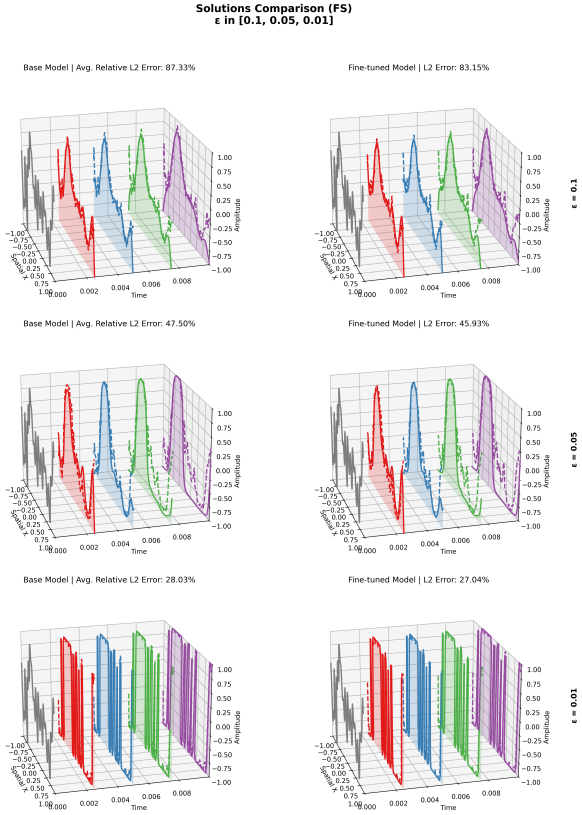


Figure 29: OOD Fourier Series (FS)

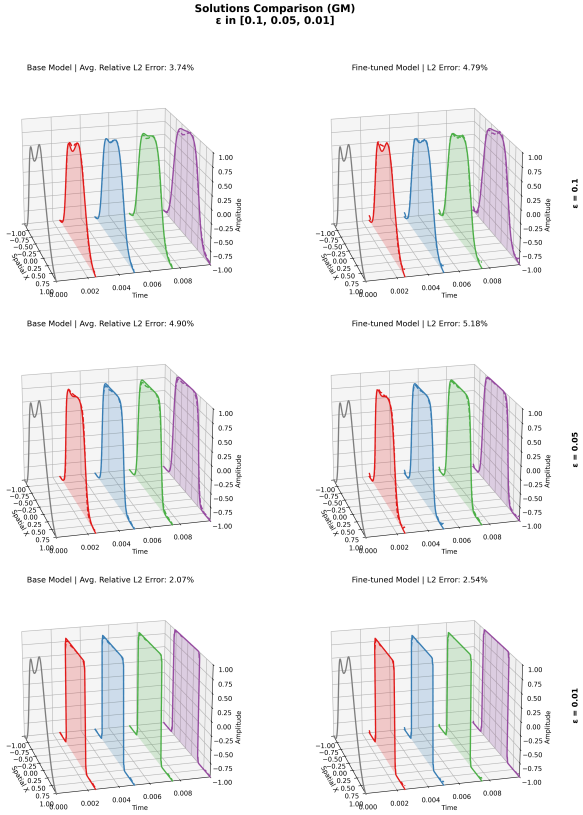


Figure 30: OOD Gaussian Mixture (GM)

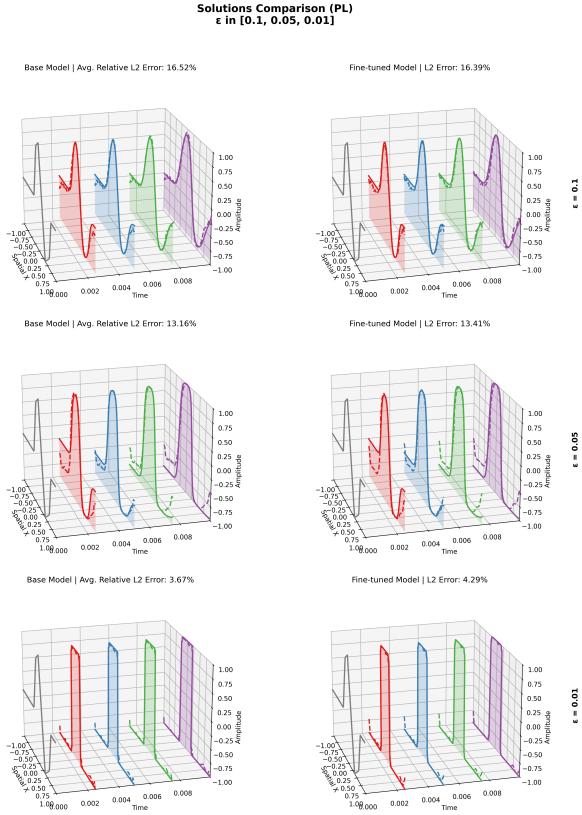


Figure 31: OOD Piecewise Linear (PL)

Appendix P3.C3: Epsilon Variation Test Results (`test_sol_eps.npy`)

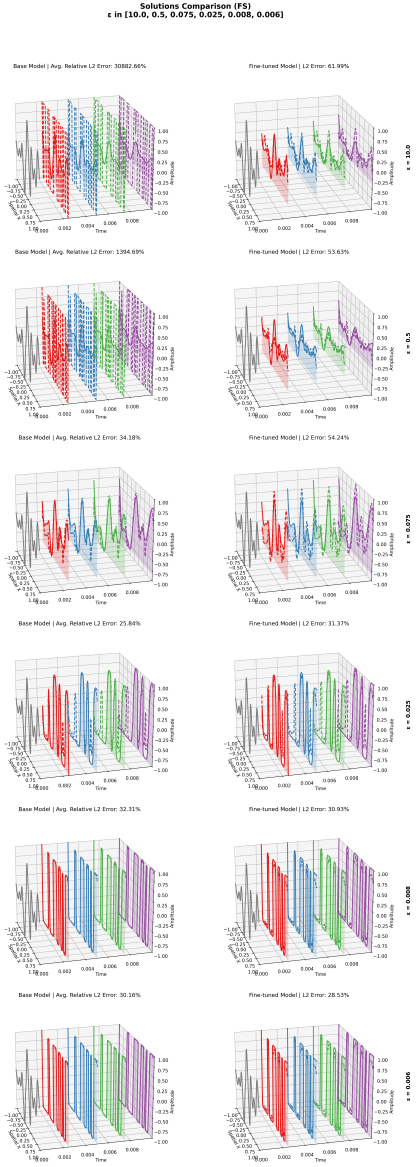


Figure 32: ε-Test Fourier Series (FS)

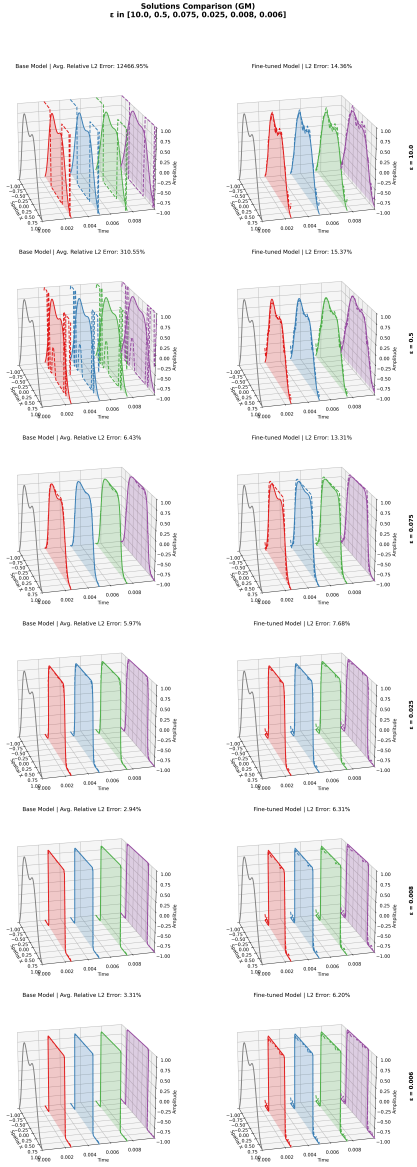


Figure 33: ε-Test Gaussian Mixture (GM)

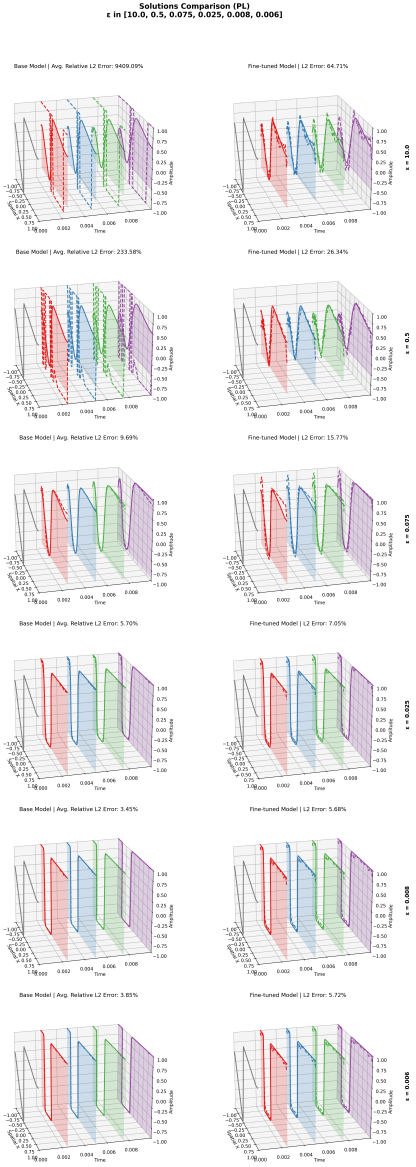


Figure 34: ε-Test Piecewise Linear (PL)

References

[1] Z. Li *et al.*, "Fourier Neural Operator for Parametric Partial Differential Equations," *ICLR 2021 - 9th International Conference on Learning Representations*, 2020, [Online]. Available: https://arxiv.org/abs/2010.08895v3

[2] F. Bartolucci, E. de Bézenac, B. Raonić, R. Molinaro, S. Mishra, and R. Alaifari, "Representation Equivalent Neural Operators: a Framework for Alias-free Operator Learning," *Advances in Neural Information Processing Systems*, vol. 36, 2023, [Online]. Available: https://arxiv.org/abs/2305.19913v2

[3] E. Perez, F. Strub, H. De Vries, V. Dumoulin, and A. Courville, "FiLM: Visual Reasoning with a General Conditioning Layer," *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 3942–3951, 2017, doi: 10.1609/aaai.v32i1.11671.

[4] S. H. Rudy, S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Data-driven discovery of partial differential equations," *Science Advances*, vol. 3, no. 4, 2017, doi: 10.1126/SCIADV.1602614/ASSET/B84AADFB-A619-444B-B64C-7FB89C5AA4E6/ASSETS/GRAPHIC/1602614-F3.JPEG.

[5] K. Hornik, M. Stinchcombe, and H. White, "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks," *Neural Networks*, vol. 3, no. 5, pp. 551–560, 1990, doi: 10.1016/0893-6080(90)90005-6.

[6] S. L. Brunton and J. N. Kutz, "Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control," *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*, pp. 1–472, 2019, doi: 10.1017/9781108380690.

[7] S. M. Allen and J. W. Cahn, "A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening," *Acta Metallurgica*, vol. 27, no. 6, pp. 1085–1095, 1979, doi: 10.1016/0001-6160(79)90196-2.

[8] S. Subramanian *et al.*, "Towards Foundation Models for Scientific Machine Learning: Characterizing Scaling and Transfer Behavior," *Advances in Neural Information Processing Systems*, vol. 36, 2023, [Online]. Available: https://arxiv.org/abs/2306.00258v1

[9] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," *ACM International Conference Proceeding Series*, vol. 382, 2009, doi: 10.1145/1553374.1553380.

[10] M. Herde *et al.*, "Poseidon: Efficient Foundation Models for PDEs," 2024, [Online]. Available: https://arxiv.org/abs/2405.19101v2

[11] S. Bartels, "Numerical Methods for Nonlinear Partial Differential Equations," vol. 47, 2015, doi: 10.1007/978-3-319-13797-1.

[12] R. Hiptmair, *Numerical Methods for Computational Science and Engineering*, 0th ed., vol. 0. Zürich: Seminar für Angewandte Mathematik, ETH Zürich, 2024.

[13] N. Kovachki *et al.*, "Neural Operator: Learning Maps Between Function Spaces," *Journal of Machine Learning Research*, vol. 23, pp. 1–97, 2021, doi: 10.5555/3648699.3648788.

[14] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data: Sparse identification of nonlinear dynamical systems," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 113, no. 15, pp. 3932–3937, 2015, doi: 10.1073/pnas.1517384113.