# Test-Time Control Over Accuracy-Cost Trade-Offs in Neural Physics Simulators via Recurrent Depth

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Accuracy-cost trade-offs are a fundamental aspect of scientific computing. Classical numerical methods inherently offer such a trade-off: increasing resolution, order, or precision typically yields more accurate solutions at higher computational cost. Inspired by adaptive-compute language models, we introduce the **Recurrent-Depth Simulator (RDS)**, an architecture-agnostic, plug-and-play framework that enables explicit test-time control over accuracy-cost trade-offs. By setting the number of recurrent steps $K$, users can generate fast, less-accurate simulations for exploratory runs or real-time control loops, or increase $K$ for more-accurate simulations in critical applications or offline studies. We validate RDS on several fluid-dynamics benchmarks, including Burgers, Korteweg-De Vries, and Kuramoto Sivashinsky, and demonstrate 1) physically faithful simulations over long horizons, even in low compute settings; 2) superior accuracy-cost trade-offs compared to alternative adaptive-compute models, including Deep Equilibrium and diffusion-based models. We further validate the recurrent-depth simulator on the challenging task generating three-dimensional turbulent compressible Navier-Stokes simulations, where we demonstrate a 0.8B parameter model with a single recurrent-depth Fourier layer attains lower mean-squared error than a 1.6B parameter counterpart with six Fourier layers, while matching computational resources and utilizing 13.5% less memory during training.

## 1 Introduction

Simulations are fundamental to science and engineering. They enable scientists to study and predict the behavior of complex systems, and engineers to quickly iterate and optimize designs, without the need for expensive or impractical experiments. Early scientific computing, limited by computational resources, produced crude simulations with limited practical value. Today, with the wide availability of enormous computers, simulations have led to breakthroughs across different domains, including numerical weather prediction, fluid and particle flows, and drug and materials design. Still, even with today's computational resources, less accurate but fast simulations are essential for early-stage studies and prototyping.

In scientific computing, techniques for explicit control over accuracy-cost trade-offs are well-established. Heuristic search methods, such as genetic algorithms and simulated annealing, can balance desired accuracy against available computational resources by controlling the size of the search space. For instance, genetic algorithms obtain better solutions with larger population sizes or by running more generations. Similarly, numerical methods, which underpin practically all simulations, have inherent accuracy-cost trade-offs: using finer discretizations, higher-order methods, and lower tolerances yields more accurate solutions but requires more computational resources. For high-dimensional or large-scale problems, this trade-off becomes extremely unfavorable, rendering many real-world problems computationally intractable.

Machine learning provides a promising avenue to overcome this trade-off. Unlike numerical methods, which rely on explicitly defined models or heuristics, machine learning methods are general-purpose learners that learn directly from the vast amounts of available measurement and observational data, and are capable of generating simulations for a wide range of problems, geometries, discretizations, and boundary conditions. Machine learning methods also benefit from hardware and software advancements specifically developed for machine learning, including GPU acceleration and parallelization. Perhaps most notably, machine learning methods can improve simulation accuracy and efficiency: given a desired accuracy, machine learning-based simulations use fewer computational resources compared to numerical methods, or, equivalently, given a computational budget, deliver greater accuracy. Several works have demonstrated these advantages in applications such as atmosphere and weather modeling and automotive design [Price et al., 2025, Bleeker et al., 2025].

At train-time, there are a number of tunable knobs available for controlling the accuracy-cost trade-off. Generally, allocating more computational resources during training leads to more accurate predictions. Whether that is by increasing the training dataset through data acquisition, data augmentation, or synthetic data; by increasing the model size through stacking more layers or using wider layers; or by improving the optimization process through more advanced optimizers, higher numerical precision, or training for more steps. Each of these adjustments directly affect the train-time accuracy-cost trade-off.

At test-time, there are fewer tunable knobs. The Deep Equilibrium model can go through more computational resources by increasing the iteration limit or by lowering the tolerance [Bai et al., 2019]. Diffusion models can make use of additional denoising steps or more advanced samplers to generate higher-quality outputs at greater cost [Ho et al., 2020, Lu et al., 2022]. Recent natural language processing research proposes reasoning models that spend more “thinking” on hard inputs and finish early on easy ones [Wei et al., 2022].

In this work, we present Recurrent-Depth Simulator (RDS), a framework that enables explicit test-time control over accuracy-cost trade-offs, with a simple implementation (see Algorithm 1 and 2). Our approach enables adaptive-depth inference without retraining or architectural redesign. By setting a small number of recurrent steps K, the model is able to generate fast, less-accurate simulations for exploratory runs or real-time control loops. Increasing K generates more accurate simulations for critical applications or offline studies. We validate the recurrent-depth simulator on several fluid-dynamics benchmarks, including Burgers’, Korteweg-De Vries, and Kuramoto Sivashinsky and demonstrate physically faithful simulations over long horizons and superior accuracy-cost trade-offs compared to alternative adaptive models, including Deep Equilibrium and diffusion-based models. We further validate RDS on the challenging task of generating three-dimensional turbulent compressible Navier-Stokes simulations, a 0.8B parameter RDS with a single recurrent-depth Fourier layer attains lower mean-squared error than a 1.6B parameter standard Fourier neural operator architecture with six Fourier layers, while matching computational resources and utilizing 13.5% less memory during training.

## 2 Background

**Partial Differential Equations.** We consider time-dependent partial differential equations of the form

$$\mathbf{u}_t + \mathcal{N}(t, \mathbf{x}, \mathbf{u}, \mathbf{u}_{\mathbf{x}}, \mathbf{u}_{\mathbf{xx}}, \dots) = 0,$$

where $t \in [0, T]$ represents the temporal dimension, $\mathbf{x} \in \mathcal{X}$ represents the (possibly multiple) spatial dimension(s), and $\mathbf{u}(t, \mathbf{x}) : [0, T] \times \mathcal{X} \to \mathbb{R}^n$ represents the state at $(t, \mathbf{x})$. Here, $\mathcal{N}$ is a non-linear operator that governs the systems’ dynamics, describing the interactions among the different variables and their derivatives. We consider initial conditions given by $\mathbf{u}(0, \mathbf{x}) = \mathbf{u}_0(\mathbf{x})$, and unless otherwise specified, assume periodic boundary conditions.

Discretizing the partial differential equations transforms the continuous equations into a discrete form, yielding a sequence of states at discrete time steps $\{\mathbf{U}_n\}_{n=0}^{N}$, where $N = T/\Delta t$ is the number of time steps $\Delta t$. This discretization induces an evolution operator $\mathcal{G}$, which maps the state at any given time step to the state at the subsequent time step $\mathcal{G} : \mathbf{U}_n \to \mathbf{U}_{n+1}$.

**Neural Simulators.** A neural (physics) simulator approximates the evolution operator $\mathcal{G}$ with a learned operator $\mathcal{G}_\theta$ by minimizing the one-step loss $\mathcal{L} = ||U_{t+1} - \mathcal{G}_\theta(U_t)||_2^2$ using data from high-fidelity simulations or real-world measurements. Repeated application of $\mathcal{G}_\theta$ generates a trajectory. Because the one-step loss does not measure trajectory performance, accuracy is typically quantified by the *trajectory error*:

$$\sum_{n=0}^{N} \left|\left| U_n - \mathcal{G}_\theta^{(n)} \left( U_0 \right) \right|\right|_2^2 ,$$

where $\mathcal{G}_\theta^{(n)}$ denotes the $n$-fold application of the neural simulator. However, for chaotic systems, the trajectory error is unreliable. Instead, let

$$\tau_\alpha = \min \left\{ t = n\Delta t \middle| \rho \left( \mathbf{U}_n, \mathcal{G}_\theta^{(n)} \left( \mathbf{U}_0 \right) \right) < \alpha \right\} ,$$

denote the earliest time at which the Pearson correlation coefficient $\rho$ between the true and predicted state falls below a specified threshold $\alpha \in (0, 1)$. Computing $\tau_\alpha$ for every test trajectory yields (i) the *average correlation horizon*, obtained by averaging all $\tau_\alpha$ values, and (ii) the *worst-cast correlation horizon*, obtained by selecting the minimum $\tau_\alpha$. Together, the trajectory error and correlation horizons capture both long-term accuracy and stability.

**Related Work.** Training with only a one-step loss can lead to a distribution shift between training states and those encountered during unrolling. To mitigate this issue, Brandstetter et al. [2022] propose the push-forward trick where the neural simulator is unrolled for two steps, but errors are backpropagated only from the second step. Subsequent studies have explored training with longer unrolling and backpropagation. Koehler et al. [2024] demonstrate that unrolling and backpropagating though 50 steps significantly improves long-term accuracy at the cost of worsened short-term accuracy and linear growth in computational and memory demands.

A wide range of architectures have been explored. For regular domains, convolutional-based architectures such as the Residual Network (ResNet [He et al., 2016]) and the U-shaped Encoder-Decoder (UNet [Ronneberger et al., 2015]) effectively capture local interactions, whereas spectral-based architectures, such as the Fourier Neural Operator (FNO [Li et al., 2020]) and its factorized variant (F-FNO [Tran et al., 2021]), leverage global frequency-domain features. For irregular domains, Brandstetter et al. [2022] propose a message-passing graph neural network, while Li et al. [2023a] extend the FNO architecture with a geometry encoder and decoder, deforming an irregular mesh into a uniform latent space suitable for FNO application, and subsequently reversing this deformation. Pokle et al. [2022] propose FNO-DEQ, a Deep Equilibrium Model (DEQ [Bai et al., 2019]) variant with Fourier layers, to solve steady-state PDEs, showing improvements in accuracy and robustness to noise over baselines with four times as many parameters. Kohl et al. [2023] demonstrated that diffusion models are viable for turbulent flow simulation. Their results show that diffusion models outperform, in terms of long-term accuracy and stability, more efficient (and more commonly used) neural simulators. Recently, transformer-based architectures have gained prominence. Alkin et al. [2024] introduce the Universal Physics Transformer, a unified Eulerian-Lagrangian framework capable of handling large-scale simulations. Separately, McCabe et al. [2023] show that a single transformer pre-trained on multiple physics tasks can match or exceed task-specific baselines without additional fine-tuning.

Modern neural simulators achieve state-of-the-art predictive accuracy and computational efficiency across complex domains. Kochkov et al. [2021] apply neural simulators to model two-dimensional turbulence, achieving comparable errors to numerical solvers while operating at 8–10 times finer resolutions, resulting in 40-80-fold speedups. Similarly, Stachenfeld et al. [2021] show that neural simulators trained at low spatial and temporal resolutions outperform traditional numerical methods at equivalent resolutions and successfully capture turbulent dynamics usually resolved by numerical methods only at significantly high resolutions. In weather forecasting, Aurora [Bodnar et al., 2024], a foundation model for the Earth system, outperforms the Integrated Forecasting System (IFS)—the state-of-the-art numerical forecasting model—with roughly a 5,000-fold speedup, running forecasts in approximately 1.1 seconds per forecast hour on a single A100 GPU compared to approximately 5720 seconds per forecast hour on a high-end CPU node. Similar results have also been reported in aerodynamics, plasma physics, and various other scientific domains Galletti et al. [2025], Li et al. [2023b], Catalani et al. [2024].

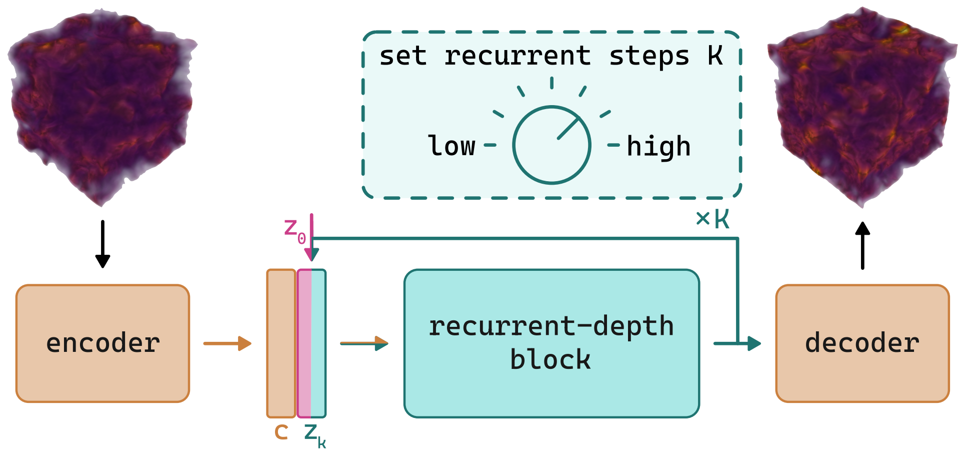

Figure 1: Schematic of the **Recurrent-Depth Simulator**. The framework consists of three main components: an encoder, a recurrent-depth block, and a decoder. At test-time, the user is able to control the accuracy-cost trade-off by setting the number of recurrent steps $K$.

## 3 Recurrent-Depth Simulator

**Overview.** The proposed **Recurrent-Depth Simulator (RDS)** consists of three main components: an encoder, a recurrent-depth block, and a decoder (see Figure 1). The encoder transforms the input state $\mathbf{x}$ into a conditioning vector $\mathbf{c}$. An initial latent $\mathbf{z}_0$ is drawn from a fixed distribution $p(\mathbf{z})$. For a user-chosen number of recurrent steps $K$, the recurrent-depth block $\mathcal{R}(\cdot, \theta_{\mathcal{R}})$—conditioned on $\mathbf{c}$—recursively updates the latent:

$$\mathbf{z}_k = \mathcal{R}\left(\left[\mathbf{c}, \mathbf{z}_{k-1}\right], \theta_{\mathcal{R}}\right), \qquad k = 1, \ldots, K.$$

After the final recurrent step, the decoder maps $\mathbf{z}_K$ to the predicted state $\hat{\mathbf{y}}$.

**Training.** RDS is trained end-to-end. For each training sample, a number of recurrent steps $K$ is drawn from a distribution $p(K)$; the recurrent-depth block is applied for that many steps, a supervised loss is evaluated, and gradients are back-propagated through the computation (see Algorithm 1). Sampling $K$ across a wide range encourages the recurrent block to contract toward a fixed point.

Large $K$ values inflate memory because every intermediate activation must ordinarily be stored. To bound the memory footprint, we use truncated backpropagation-through-time with a fixed *backpropagation window* $B$ [Williams and Peng, 1990]. Gradients are propagated through at most the last $B$ recurrent steps, while earlier steps are treated as constants. This caps memory at $O(B)$ regardless of $K$ and has proved sufficient for optimization. Empirical results for different backpropagation windows $B$ are explored in Appendix D.

**Inference.** At test-time, the user is free to choose $K$ according to their desired accuracy and available computational resources (see Algorithm 2). Small $K$ values generate fast, less-accurate simulations ideal for exploratory runs, or real-time control loops. Large $K$ values generate more-accurate slow simulations suitable for critical applications or offline studies. Empirically, the first few recurrent steps make the largest adjustments to the latent vector $\mathbf{z}_k$; subsequent steps contribute progressively smaller, yet still beneficial, adjustments. This behavior mirrors numerical methods, such as fixed-point and Newton methods, giving RDS a strong inductive bias that is well-suited for physical simulation tasks.

**Modularity.** The RDS framework is modular: each of, the encoder, recurrent-depth block, and decoder may be instantiated with the architecture primitive best suited to the problem—e.g., convolutional layers for Eulerian simulations or graph-convolutional layers for Lagrangian simulations—without altering the training or inference algorithms. The entire pipeline remains a standard end-to-end, supervised model with no custom losses, schedulers, or tricks—so adoption is essentially plug-and-play.

**Algorithm 1** Recurrent-Depth Simulator Training

```
Input: training data $\mathbf{x}, \mathbf{y}$
Output: model parameters $\boldsymbol{\theta}_\mathcal{E}, \boldsymbol{\theta}_\mathcal{R}, \boldsymbol{\theta}_\mathcal{D}$
repeat
    for $i \in \mathcal{B}$ do                                  ▷ for every training example index in batch
        $\mathbf{c} \leftarrow \mathcal{E}(\mathbf{x}_i, \boldsymbol{\theta}_\mathcal{E})$        ▷ compute conditioning vector
        $\mathbf{z}_0 \sim p(\mathbf{z})$                       ▷ sample initial latent representation
        $K \sim p(K)$                                           ▷ sample number of recurrent steps
        for $k = 1$ to $K$ do                                   ▷ unroll $K$ recurrent steps
            $\mathbf{z}'_{k-1} \leftarrow [\mathbf{c}, \mathbf{z}_{k-1}]$   ▷ concatenate conditioning and latent representation
            $\mathbf{z}_k \leftarrow \mathcal{R}(\mathbf{z}'_{k-1}, \boldsymbol{\theta}_\mathcal{R})$   ▷ apply recurrent block
        end for
        $\hat{\mathbf{y}}_i \leftarrow \mathcal{D}(\mathbf{z}_K, \boldsymbol{\theta}_\mathcal{D})$   ▷ decode latent representation
        $l_i \leftarrow ||\mathbf{y}_i - \hat{\mathbf{y}}_i||$  ▷ compute individual loss
    end for
    accumulate losses for batch and take gradient step
until converged
```

**Algorithm 2** Recurrent-Depth Simulator Inference

```
Input: input state $\mathbf{x}$, number of steps $K$, model parameters $\boldsymbol{\theta}_\mathcal{E}, \boldsymbol{\theta}_\mathcal{R}, \boldsymbol{\theta}_\mathcal{D}$
Output: output state $\mathbf{y}$
$\mathbf{c} \leftarrow \mathcal{E}(\mathbf{x}, \boldsymbol{\theta}_\mathcal{E})$        ▷ compute conditioning vector
$\mathbf{z}_0 \sim p(\mathbf{z})$                                   ▷ sample initial latent representation
for $k = 1$ to $K$ do                                               ▷ unroll $K$ recurrent steps
    $\mathbf{z}'_{k-1} \leftarrow [\mathbf{c}, \mathbf{z}_{k-1}]$   ▷ concatenate conditioning and latent representation
    $\mathbf{z}_k \leftarrow \mathcal{R}(\mathbf{z}'_{k-1}, \boldsymbol{\theta}_\mathcal{R})$   ▷ apply recurrent block
end for
$\mathbf{y} \leftarrow \mathcal{D}(\mathbf{z}_K, \boldsymbol{\theta}_\mathcal{D})$   ▷ decode latent representation to predicted state
```

**Initial Latent Distribution.** The initial latent vector $\mathbf{z}_0$ is drawn from a standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$. A Gaussian distribution is a natural default, and is widely used in diffusion models, yet other choices are possible. For example, replacing the Gaussian with a Student-$t$ prior to better capture heavy-tailed behavior [Pandey et al., 2025], or learning a fixed latent vector directly [Jaegle et al., 2021]. The RDS framework is agnostic to this choice; any prior that suits target problem can be substituted without changing the rest of the pipeline.

**Recurrent Step Distribution.** The number of recurrent steps $K$ is drawn from a Poisson log-normal distribution:

$$\upsilon \sim \mathcal{N}\left(\log \bar{K} - \frac{1}{2}\sigma^2, \sigma\right),$$
$$K \sim \text{Poisson}\left(e^{\upsilon}\right) + 1,$$

where $\bar{K} + 1$ is the desired mean. This distribution exposes the model to a broad spectrum of compute budgets during training: it is positively skewed with most draws landing near $\bar{K}$, but occasional very small and very large values are sampled, encouraging the recurrent-block to remain stable across both shallow and deep rollouts. Unless noted otherwise, we use $\bar{K} = 32$ and $\sigma = 0.5$—alternative values are explored in Appendix E.

**Merging Conditioning and Latent Vectors.** At each recurrent step, the conditioning vector $\mathbf{c}$ must be merged with the current latent vector $\mathbf{z_k}$. The simplest scheme is plain addition: $\mathbf{z}'_k = \mathbf{c} + \mathbf{z}_k$. A slightly richer variant introduces learnable scalar weights: $\mathbf{z}'_k = \alpha\mathbf{c} + \beta\mathbf{z}_k$. The weights can be made element-wise: $\mathbf{z}'_k = \boldsymbol{\alpha} \odot \mathbf{c} + \boldsymbol{\beta} \odot \mathbf{z}_k$. Alternatives include point-wise projection, or concatenating and passing the result through a width-halving layer. All variants are drop-in replacements and are explored in Appendix F.

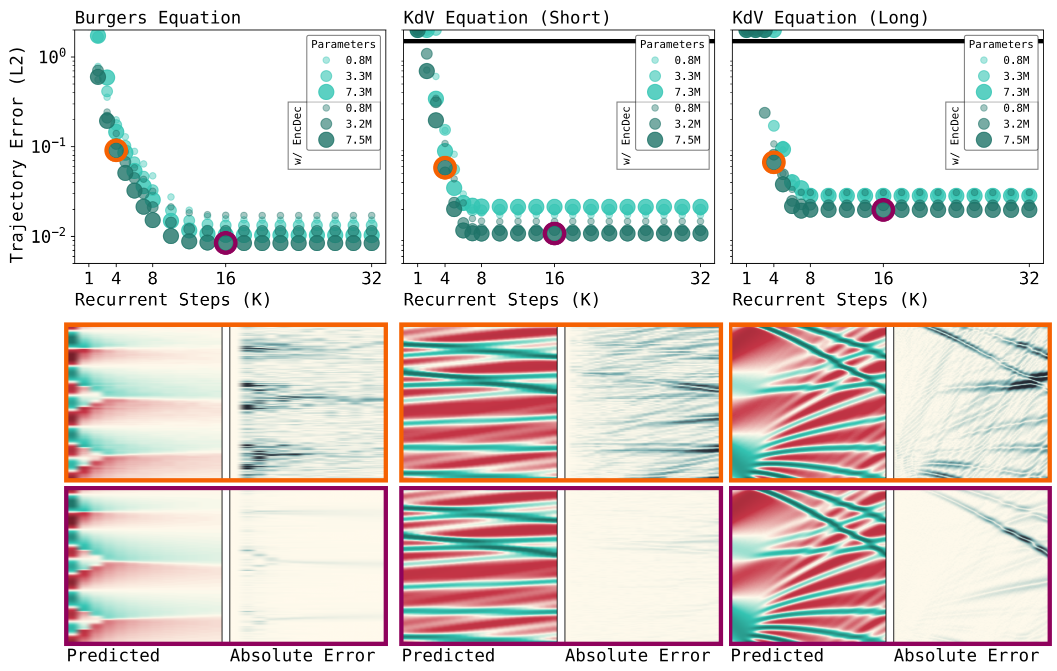

Figure 2: **Top:** Trajectory Error (L2) versus Recurrent Steps (K) for the Burgers (left), short-horizon KdV (middle), and long-horizon KdV (right). **Bottom:** Trajectories at $K = 4$ (orange) and $K = 16$ (purple) (highlighted above). Increasing $K$ sharpens shocks in Burgers and aligns soliton crests in KdV, illustrating how recurrent depth controls the accuracy–cost trade-off.

## 4 Results

Full specifications of the hardware, data acquisition, data generation, preprocessing pipelines, along with training hyper-parameters, are given in Appendix A-C. Unless noted, the main experiments use the Recurrent-Depth Simulator with Fourier layers—denoted $\text{RDS}_{\text{Fourier}}$—whose infinite receptive field simplifies analysis for depth-varying models. Other variants are explored in Appendix G.

### 4.1 Experiment: Accuracy-Cost Trade-Off

Commonly used neural simulators are trained for a single accuracy-cost setting: once the model is trained, every forward pass delivers the same expected accuracy and incurs the same cost. RDS, on the other hand, has a tunable knob for controlling the accuracy-cost setting (the number of recurrent steps $K$). The purpose of this experiment is to empirically demonstrate whether rolling out the trajectory across values of recurrent steps $K$ is viable.

**Experimental Setup.** We conduct experiments on three datasets: Burgers, short-horizon KdV, and long-horizon KdV. Two instantiations of RDS are benchmarked. The first variant ($\text{RDS}_{\text{Fourier}}$ `wo/ EncDec`) lifts the input with a point-wise operation, recursively applies a recurrent-depth block with a single Fourier layer, and projects back to physical space; the second variant ($\text{RDS}_{\text{Fourier}}$ `w/ EncDec`) inserts an additional Fourier layer in, both, the encoder and decoder. For each variant, we target three parameter budgets ($\sim 1.0\text{M}, 3.5\text{M}, 7.5\text{M}$), yielding six models per dataset. We use $\bar{K} = 32$ and $B = 4$. After convergence, we generate trajectories for every $K \in \{1, \ldots, 32\}$ and measure the trajectory error. All experiments are repeated with three seeds and averaged.

**Results.** Across all three datasets, both variants show the same qualitative accuracy-cost curve (Figure 2), but $\text{RDS}_{\text{Fourier}}$ `w/ EncDec` achieves consistently lower trajectory error. As $K$ increases, the trajectory error falls steadily and plateaus around $K = 16$ for Burgers and $K = 8$ for both, short- and long-horizon KdV ; further steps neither help nor harm. For each dataset, we plot low-compute

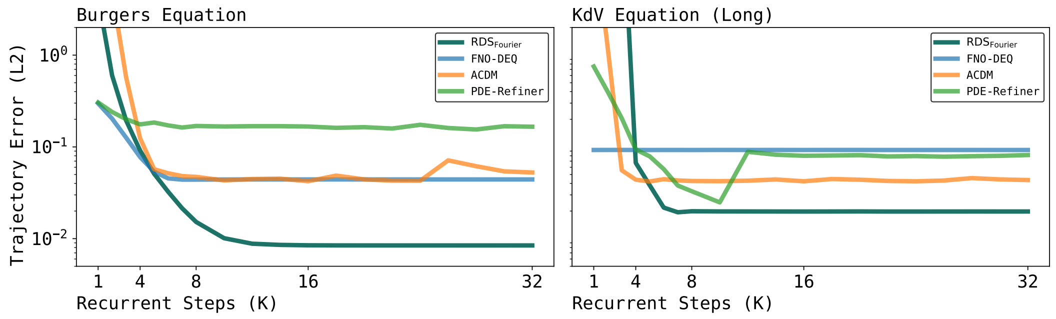

Figure 3: Trajectory Error (L2) versus Recurrent Steps (K) for the Burgers (left), and long-horizon KdV (right). Curves compare RDS$_{\text{Fourier}}$ (teal), FNO-DEQ (blue) [Marwah et al., 2023], ACDM (orange) [Kohl et al., 2023], and PDE-Refiner (green) [Lippe et al., 2023]. Across both tasks, RDS$_{\text{Fourier}}$ achieves the best accuracy-cost curve and reaches the lowest plateau.

($K = 4$) and high-compute ($K = 16$) trajectories. In Burgers, the two settings reproduce the same shock patterns, with the low-compute run showing slightly larger absolute error around the fronts. In both KdV datasets, the low-compute run already recovers the full soliton train; the absolute error is almost entirely a small amplitude and/or phase offset, visible as narrow streaks along the soliton trajectories. Increasing to $K = 16$ sharpens the shocks and aligns the soliton crests. These results demonstrate that RDS$_{\text{Fourier}}$ delivers physically faithful simulations over a range of accuracy-cost settings. Extended results are presented in Appendix G.

### 4.2 Experiment: Alternatives

There are a few recent neural simulators that have test-time controllable knobs. FNO-DEQ is a Deep Equilibrium Model with Fourier layers whose runtime is set by a maximum number of iterations or a minimum update. ACDM—an autoregressive conditional diffusion model—is able to adjust the prediction quality by varying the number and schedule of denoising steps. PDE-Refiner applies the same diffusion principle in a direct prediction and refinement process. In this experiment, we benchmark RDS$_{\text{Fourier}}$ against the three alternatives under identical data and training setups.

**Experimental Setup.** We conduct experiments on three datasets: Burgers, long-horizon KdV, and long-horizon Kuramoto-Sivashinsky. For RDS, we carry over the best variant from the previous set of experiments: point-wise lift + Fourier layer encoder, a recurrent-depth block with one Fourier layer, Fourier layer with point-wise projection decoder—configured with $\sim 7.5$M parameters, $\bar{K} = 32$, and $B = 4$ steps. FNO-DEQ follows the setup of Pokle et al. [2022], with its width scaled to match a parameter count of $\sim 7.5$M. ACDM and PDE-Refiner use a modern UNet backbone from their original implementations [Kohl et al., 2023, Lippe et al., 2023]. In early tests, both diffusion-based models proved parameter-inefficient and could not rollout beyond a few steps, so we train them with $\sim 15$M parameters for Burgers and KdV, and $\sim 50$M parameters for KS (the scale used by Lippe et al. [2023]). After convergence, we generate trajectories for every $K \in \{1, \ldots, 32\}$ (where $K$ is equal to the recurrent steps for RDS, iterations for FNO-DEQ, and denoising steps for ACDM and PDE-Refiner). On Burgers and KdV, we measure and report the trajectory error. Since the KS equation produces chaotic behavior, we measure the average and worst-case correlation horizon over a sweep of 30 thresholds ($\alpha = 0.7$-$0.99$ in increments of $0.01$).

**Results.** On Burgers, FNO-DEQ, ACDM, and PDE-Refiner all plateau by $K \approx 4$ (see Figure 3 (left)); PDE-Refiner gains practically nothing beyond its second refinement step. RDS$_{\text{Fourier}}$, by contrast, continues to improve until $K \approx 16$, while using half the parameters of the diffusion-based models. On KdV, FNO-DEQ exhibits the convergence limitation reported by Sittoni and Tudisco [2024]—the latent representation oscillates around, rather than converges to, the fixed point—so additional iterations provide no improvement. The ten-fold larger training dataset helps the diffusion-based models, however, once again, ACDM plateaus near $K \approx 4$. PDE-Refiner improves up to $K = 11$ before degrading because larger $K$ values are out-of-distribution. RDS$_{\text{Fourier}}$ delivers the best

| **Model** | **Params** | **Training Memory** | **Training Epochs** | **Training GFLOPs** | **MSE** $\times 10^{-2}$ **Density** | **MSE** $\times 10^{-2}$ **Pressure** | **MSE** $\times 10^{-2}$ **Velocity** |
|---|---|---|---|---|---|---|---|
| FNO | 0.5B | 38 GB | 100 | $1 \times 10^7$ | 9.60 | 9.59 | 9.55 |
| FNO | 1.0B | 57 GB | 100 | $2 \times 10^7$ | 7.83 | 7.79 | 7.82 |
| FNO | 1.6B | 73 GB | 100 | $3 \times 10^7$ | 7.61 | 7.59 | 7.62 |
| $\text{RDS}_{\text{Fourier}}$ | 0.8B | 64 GB | **82** | $3 \times 10^7$ | **7.57** | **7.51** | **7.53** |
| $\text{RDS}_{\text{Fourier}}$ | 0.8B | 64 GB | 100 | $5 \times 10^7$ | **7.37** | **7.33** | **7.36** |

Table 1: Performance comparison between FNO and $\text{RDS}_{\text{Fourier}}$. We report the total number of learnable parameters, the peak of GPU memory during training, total training epochs and training GFLOPs. We also report the trajectory MSE, i.e. the MSE between the ground truth trajectory and the predicted trajectory. We calculate it over 3 different channels, velocity, pressure, and density fields.

accuracy-cost curves and lowest trajectory errors. On KS (see Appendix G), where the diffusion-based models have 7-fold the amount of parameters as $\text{RDS}_{\text{Fourier}}$, ACDM plateaus early, and PDE-Refiner shows erratic worst-case correlation horizons. Taken together, $\text{RDS}_{\text{Fourier}}$ consistently outperforms alternatives while using fewer parameters.

### 4.3 Experiment: Large-Scale Compressible Navier-Stokes

**Models Details.** We train five different variations of Fourier Neural Operator (FNO), each with $64$ channels and $20$ modes. Three of these models are FNO with various depths, specifically with $2, 4, 6$ layers. The remaining two are $\text{RDS}_{\text{Fourier}}$ (`w/ EncDec`) models, each using a single Fourier layer in the encoder, decoder, and recurrent block, with a backpropagation window of $4$. The main architectural difference lies in the value of $\bar{K}$. In one model, we set $\bar{K} = 16$, and we use it to match the number of training steps of the other three FNO models. In contrast, the second model uses $\bar{K} = 8$. It is chosen to match the training FLOPs of the 6-layer FNO. We did not compare against DEQ, PDE-Refiner, or diffusion models. Training DEQ is notoriously slow and becomes impractical for a problem of this scale. As for PDE-Refiner and diffusion models, implementing a U-Net with 3D convolutional layers that fits within the same GPU memory budget would result in a model that is too shallow, with a limited receptive field. For these reasons, we chose not to include these baselines.

Training is run for $100$ epochs across all models, except for the FLOPs-matched recurrent model, which is trained for $82$ epochs to match the FLOPs of the deepest baseline.

**Results.** As shown in table 1, both $\text{RDS}_{\text{Fourier}}$ models consistently achieve a lower trajectory MSE compared to their non-recurrent FNO counterparts. Remarkably, the $\text{RDS}_{\text{Fourier}}$ with $\bar{K} = 8$, which is constrained to match the 6 layers FNO's total training FLOPs, still achieves lower trajectory MSE than all the standard models. Furthermore, $\text{RDS}_{\text{Fourier}}$ require substantially less GPU memory, approximately $13.5\%$ compared to the 6 layer FNO and uses half of the parameters. The encoder and decoder of $\text{RDS}_{\text{Fourier}}$ have the same number of parameters as the smallest FNO model. As shown in table 1, incorporating the recurrent block leads to a $22\%$ improvement in test performance. Similarly, the mid-size FNO can be interpreted as having a single layer in both the encoder and decoder, with a 2-layer middle block. Despite having only half as many parameters in the recurrent-depth block, $\text{RDS}_{\text{Fourier}}$ consistently outperforms the mid-size FNO.

**Conclusion.** We introduce the Recurrent-Depth Simulator (RDS), a simple and general methodology for dynamically adjusting the computational budget at test time. We describe how architectural primitives can be integrated into RDS, outline the training procedure, and discuss strategies for managing the accuracy–cost trade-off. We demonstrate that RDS achieves superior accuracy–efficiency trade-offs compared to state-of-the-art alternatives, including Deep Equilibrium and diffusion-based models. The experiments for three-dimensional simulation suggest that recurrent-depth is a viable and scalable mechanism for improving neural simulators.

## References

Benedikt Alkin, Andreas Fürst, Simon Schmid, Lukas Gruber, Markus Holzleitner, and Johannes Brandstetter. Universal physics transformers: A framework for efficiently scaling neural operators. *Advances in Neural Information Processing Systems*, 2024.

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993, 2021.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. *Advances in neural information processing systems*, 32, 2019.

Shaojie Bai, Vladlen Koltun, and J Zico Kolter. Multiscale deep equilibrium models. *Advances in neural information processing systems*, 33:5238–5250, 2020.

Maurits Bleeker, Matthias Dorfer, Tobias Kronlachner, Reinhard Sonnleitner, Benedikt Alkin, and Johannes Brandstetter. Neuralcfd: Deep learning on high-fidelity automotive aerodynamics simulations. *arXiv preprint arXiv:2502.09692*, 2025.

Cristian Bodnar, Wessel P Bruinsma, Ana Lucic, Megan Stanley, Johannes Brandstetter, Patrick Garvan, Maik Riechert, Jonathan Weyn, Haiyu Dong, Anna Vaughan, et al. Aurora: A foundation model of the atmosphere. *arXiv preprint arXiv:2405.13063*, 2024.

Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural pde solvers. *arXiv preprint arXiv:2202.03376*, 2022.

Charles G Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of computation*, 19(92):577–593, 1965.

Giovanni Catalani, Siddhant Agarwal, Xavier Bertrand, Frédéric Tost, Michael Bauerheim, and Joseph Morlier. Neural fields for rapid aircraft aerodynamics simulations. *Scientific Reports*, 14 (1):25496, 2024.

Nanxin Chen, Yu Zhang, Heiga Zen, Ron J Weiss, Mohammad Norouzi, and William Chan. Wavegrad: Estimating gradients for waveform generation. *arXiv preprint arXiv:2009.00713*, 2020.

Prafulla Dhariwal and Alexander Nichol. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems*, 34:8780–8794, 2021.

Gianluca Galletti, Fabian Paischer, Paul Setinek, William Hornsby, Lorenzo Zanisi, Naomi Carey, Stanislas Pamela, and Johannes Brandstetter. 5d neural surrogates for nonlinear gyrokinetic simulations of plasma turbulence. *arXiv preprint arXiv:2502.07469*, 2025.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *arXiv preprint arXiv:2502.05171*, 2025.

Zhengyang Geng, Xin-Yu Zhang, Shaojie Bai, Yisen Wang, and Zhouchen Lin. On training implicit models. *Advances in Neural Information Processing Systems*, 34:24247–24260, 2021.

Zhengyang Geng, Ashwini Pokle, and J Zico Kolter. One-step diffusion distillation via deep equilibrium models. *Advances in Neural Information Processing Systems*, 36:41914–41931, 2023.

Dobrik Georgiev, Joseph Wilson, Davide Buffelli, and Pietro Liò. Deep equilibrium algorithmic reasoning. *Advances in Neural Information Processing Systems*, 37:33638–33667, 2024.

Davis Gilton, Gregory Ongie, and Rebecca Willett. Deep equilibrium architectures for inverse problems in imaging. *IEEE Transactions on Computational Imaging*, 7:1123–1133, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. Perceiver: General perception with iterative attention. In *International conference on machine learning*, pages 4651–4664. PMLR, 2021.

Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. Elucidating the design space of diffusion-based generative models. *Advances in neural information processing systems*, 35:26565–26577, 2022.

Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21):e2101784118, 2021.

Felix Koehler, Simon Niedermayr, Nils Thuerey, et al. Apebench: A benchmark for autoregressive neural emulators of pdes. *Advances in Neural Information Processing Systems*, 37:120252–120310, 2024.

Georg Kohl, Li-Wei Chen, and Nils Thuerey. Benchmarking autoregressive conditional diffusion models for turbulent flow simulation. *arXiv preprint arXiv:2309.01745*, 2023.

Zhifeng Kong, Wei Ping, Jiaji Huang, Kexin Zhao, and Bryan Catanzaro. Diffwave: A versatile diffusion model for audio synthesis. *arXiv preprint arXiv:2009.09761*, 2020.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Zongyi Li, Miguel Liu-Schiaffini, Nikola Kovachki, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Learning dissipative dynamics in chaotic systems. *arXiv preprint arXiv:2106.06898*, 2021.

Zongyi Li, Daniel Zhengyu Huang, Burigede Liu, and Anima Anandkumar. Fourier neural operator with learned deformations for pdes on general geometries. *Journal of Machine Learning Research*, 24(388):1–26, 2023a.

Zongyi Li, Nikola Kovachki, Chris Choy, Boyi Li, Jean Kossaifi, Shourya Otta, Mohammad Amin Nabian, Maximilian Stadler, Christian Hundt, Kamyar Azizzadenesheli, et al. Geometry-informed neural operator for large-scale 3d pdes. *Advances in Neural Information Processing Systems*, 36: 35836–35854, 2023b.

Zongyi Li, Hongkai Zheng, Nikola Kovachki, David Jin, Haoxuan Chen, Burigede Liu, Kamyar Azizzadenesheli, and Anima Anandkumar. Physics-informed neural operator for learning partial differential equations. *ACM/JMS Journal of Data Science*, 1(3):1–27, 2024.

Phillip Lippe, Bas Veeling, Paris Perdikaris, Richard Turner, and Johannes Brandstetter. Pde-refiner: Achieving accurate long rollouts with neural pde solvers. *Advances in Neural Information Processing Systems*, 36:67398–67433, 2023.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017a.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017b. URL `https://arxiv.org/abs/1608.03983`.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL `https://arxiv.org/abs/1711.05101`.

Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. Dpm-solver++: Fast solver for guided sampling of diffusion probabilistic models. *arXiv preprint arXiv:2211.01095*, 2022.

Harris Abdul Majid, Pietro Sittoni, and Francesco Tudisco. Solaris: A foundation model for the sun. In *Neurips 2024 Workshop Foundation Models for Science: Progress, Opportunities, and Challenges*, 2024. URL `https://openreview.net/forum?id=FUZryaIpV2`.

Tanya Marwah, Ashwini Pokle, J Zico Kolter, Zachary Lipton, Jianfeng Lu, and Andrej Risteski. Deep equilibrium based neural operators for steady-state pdes. *Advances in Neural Information Processing Systems*, 36:15716–15737, 2023.

Michael McCabe, Bruno Régaldo-Saint Blancard, Liam Holden Parker, Ruben Ohana, Miles Cranmer, Alberto Bietti, Michael Eickenberg, Siavash Golkar, Geraud Krawezik, Francois Lanusse, et al. Multiple physics pretraining for physical surrogate models. *arXiv preprint arXiv:2310.02994*, 2023.

Alex Nichol, Prafulla Dhariwal, Aditya Ramesh, Pranav Shyam, Pamela Mishkin, Bob McGrew, Ilya Sutskever, and Mark Chen. Glide: Towards photorealistic image generation and editing with text-guided diffusion models. *arXiv preprint arXiv:2112.10741*, 2021.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025.

Kushagra Pandey, Jaideep Pathak, Yilun Xu, Stephan Mandt, Michael Pritchard, Arash Vahdat, and Morteza Mardani. Heavy-tailed diffusion models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=tozlOEN4qp`.

Ashwini Pokle, Zhengyang Geng, and J Zico Kolter. Deep equilibrium approaches to diffusion models. *Advances in Neural Information Processing Systems*, 35:37975–37990, 2022.

Ilan Price, Alvaro Sanchez-Gonzalez, Ferran Alet, Tom R Andersson, Andrew El-Kadi, Dominic Masters, Timo Ewalds, Jacklynn Stott, Shakir Mohamed, Peter Battaglia, et al. Gencast: Diffusion-based ensemble forecasting for medium-range weather. *arXiv preprint arXiv:2312.15796*, 2023.

Ilan Price, Alvaro Sanchez-Gonzalez, Ferran Alet, Tom R Andersson, Andrew El-Kadi, Dominic Masters, Timo Ewalds, Jacklynn Stott, Shakir Mohamed, Peter Battaglia, et al. Probabilistic weather forecasting with machine learning. *Nature*, 637(8044):84–90, 2025.

Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3, 2022.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

Chitwan Saharia, William Chan, Huiwen Chang, Chris Lee, Jonathan Ho, Tim Salimans, David Fleet, and Mohammad Norouzi. Palette: Image-to-image diffusion models. In *ACM SIGGRAPH 2022 conference proceedings*, pages 1–10, 2022a.

Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, et al. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in neural information processing systems*, 35:36479–36494, 2022b.

Chitwan Saharia, Jonathan Ho, William Chan, Tim Salimans, David J Fleet, and Mohammad Norouzi. Image super-resolution via iterative refinement. *IEEE transactions on pattern analysis and machine intelligence*, 45(4):4713–4726, 2022c.

Arne Schneuing, Charles Harris, Yuanqi Du, Kieran Didi, Arian Jamasb, Ilia Igashov, Weitao Du, Carla Gomes, Tom L Blundell, Pietro Lio, et al. Structure-based drug design with equivariant diffusion models. *Nature Computational Science*, 4(12):899–909, 2024.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472, 2022.

Pietro Sittoni and Francesco Tudisco. Subhomogeneous deep equilibrium models. In *International Conference on Machine Learning*, pages 45794–45812. PMLR, 2024.

Jascha Sohl-Dickstein. The boundary of neural network trainability is fractal. *arXiv preprint arXiv:2402.06184*, 2024.

Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. pmlr, 2015.

Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. *Advances in neural information processing systems*, 32, 2019.

Kimberly Stachenfeld, Drummond B Fielding, Dmitrii Kochkov, Miles Cranmer, Tobias Pfaff, Jonathan Godwin, Can Cui, Shirley Ho, Peter Battaglia, and Alvaro Sanchez-Gonzalez. Learned coarse models for efficient turbulence simulation. *arXiv preprint arXiv:2112.15275*, 2021.

Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Daniel MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. Pdebench: An extensive benchmark for scientific machine learning. *Advances in Neural Information Processing Systems*, 35:1596–1611, 2022.

Eleuterio F Toro, Michael Spruce, and William Speares. Restoration of the contact surface in the hll-riemann solver. *Shock waves*, 4(1):25–34, 1994.

Alasdair Tran, Alexander Mathews, Lexing Xie, and Cheng Soon Ong. Factorized fourier neural operators. *arXiv preprint arXiv:2111.13802*, 2021.

Bram Van Leer. Towards the ultimate conservative difference scheme. *Journal of computational physics*, 135(2):229–248, 1997.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

Sherry Yang, KwangHwan Cho, Amil Merchant, Pieter Abbeel, Dale Schuurmans, Igor Mordatch, and Ekin Dogus Cubuk. Scalable diffusion for materials generation. *arXiv preprint arXiv:2311.09235*, 2023.

Xiaohui Zhong, Lei Chen, Xu Fan, Wenxu Qian, Jun Liu, and Hao Li. Fuxi-2.0: Advancing machine learning weather forecasting model for practical applications. *arXiv preprint arXiv:2409.07188*, 2024.

## A Hardware Details

For the one-dimensional Burgers, Korteweg-De Vries, and Kuramoto-Sivashinsky equations, we generated the data using an AMD 7950X processor (16 cores/32 threads). Each example trajectory in the Burgers equation and Korteweg-De Vries equation datasets took approximately 10 and 20 minutes to generate, respectively. The entire datasets, with 600 examples (500 training examples and 100 testing examples), took approximately 6000 and 12000 minutes to generate, respectively. Each training example in the Kuramoto-Sivashinsky equation dataset took approximately 15 minutes to generate. The testing examples were twice as long, and took approximately 30 minutes to generate. The entire dataset, with 500 training examples and 100 testing examples, took approximately 10500 minutes to generate. All together, the three one-dimensional datasets took approximately 28500 minutes (475 hours) to generate.

All one-dimensional models were trained on a single NVIDIA A100 (40GB) GPU per run, with average training times ranging from 15-300 minutes per model—smaller models on the Burgers dataset took 15 minutes, whereas larger models trained on Korteweg-De Vries or Kuramoto-Sivashinsky datasets, which contained 10 times longer trajectories, took closer to 300 minutes. We trained approximately 1000 models for exploratory experiments (e.g., tuning hyperparameters, evaluating alternative architectures) and final experiments, and estimate a total of 1000 NVIDIA A100 (40GB) GPU hours.

The three-dimensional models were much larger. Under our experimental setup, only the smallest Fourier Neural Operator with two layers managed to fit on a single NVIDIA A100 (40GB) GPU. This model did not perform well (approximately 25-30% higher MSE compared to its six layer variant). So all three-dimensional experiments were trained on a single NVIDIA A100 (80GB) GPU. On average, each training run took 1200-1500 minutes to complete. We trained approximately 10 models for exploratory experiments and final experiments, and estimate a total of 225 NVIDIA A100 (80GB) GPU hours.

## B Data Details

### B.1 Equations

**Burgers Equation.** The Burgers equation is a second-order nonlinear partial differential equation derived to model convective steepening and diffusive smoothing. Its one-dimensional variant can be expressed as:

$$u_t + uu_x = \nu u_{xx}.$$

Here, $\nu$ plays the role of kinematic viscosity. Setting $\nu = 0$ yields the inviscid form $u_t + uu_x = 0$, whose solutions develop finite-time shock discontinuities; the viscous term $\nu u_{xx}$ regularises these shocks but introduces extremely thin internal layers that remain numerically stiff. Machine learning methods must learn to represent sharp gradients, moving shocks and the delicate interplay between nonlinearity and diffusion.

**Korteweg-De Vries Equation.** The Korteweg-De Vries (KdV) is a third-order nonlinear partial differential equation derived to model weakly nonlinear, weakly dispersive unidirectional waves. Its one-dimensional variant can be expressed as:

$$u_t + \alpha uu_x + u_{xxx} = 0.$$

Here, $\alpha$ (often set to $\pm 1$ or $\pm 6$) controls nonlinear steepening while the third-order derivative $u_{xxx}$ introduces dispersion. The exact balance of these effects produces solitary-wave solutions (solitons) that preserve their shape and speed and undergo only phase shifts upon interaction –small amounts of artificial dissipation can destroy these very structures making KdV an ideal candidate for evaluating whether machine learning methods can maintain accuracy, stability and conservation over long horizons.

**Kuramoto-Sivashinsky Equation.** The Kuramoto-Sivashinsky (KS) equation is a fourth-order nonlinear partial differential equation derived to model diffusive-thermal instabilities in laminar flame fronts. Its one-dimensional variant can be expressed as:

$$u_t + u_{xx} + u_{xxxx} + uu_x = 0.$$

Here, the fourth-order derivative $u_{xxxx}$ and the nonlinear term $uu_x$ contribute to complex and chaotic behavior which present a challenge for traditional numerical solvers. The challenges and the wide applicability of the KS equation make it an ideal candidate for evaluating machine learning methods.

**Compressible Navier-Stokes Equations.** The three-dimensional Compressible Navier-Stokes (CNS) equations model complex phenomena such as shock wave formation and propagation. They are widely used across various engineering and physics applications, including aircraft wing aerodynamics and the formation of interstellar gases. The equations can be expressed as:

$$\begin{aligned}
&\partial_t \rho + \nabla \cdot (\rho \mathbf{v}) = 0, \quad \rho(\partial_t \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v}) = -\nabla p + \eta \Delta \mathbf{v} + (\zeta + \eta/3)\nabla(\nabla \cdot \mathbf{v}), \\
&\partial_t(\epsilon + \rho \mathbf{v}^2/2) + \nabla \cdot \left[(p + \epsilon + \rho \mathbf{v}^2/2)\mathbf{v} - \mathbf{v} \cdot \sigma'\right] = 0,
\end{aligned}$$

where $\rho$ is the mass density, $\mathbf{v}$ is the fluid velocity, $p$ is the pressure, and $\epsilon$ is the internal energy determined by the equation of state. The term $\sigma'$ denotes the viscous stress tensor, while $\eta$ and $\zeta$ represent the shear and bulk viscosities, respectively. In this case, using a classical numerical solver to approximate the fluid flow is particularly challenging due to strict stability constraints, high computational cost, and the need for accurate yet robust numerical schemes that handle shocks, dissipation, and grid adaptivity in large-scale domains. Even though machine learning can overcome several of the challenges posed by traditional solvers, training a neural simulator on three-dimensional data comes with considerable engineering complexity. The primary limitation arises from storing the activations during training, increasing the memory requirement compared to smaller dimensions problems.

## B.2 Data Generation

For the one-dimensional Burgers and Korteweg-De Vries equations, we set $T = 10$ and $T = 100$, respectively (for both training and testing datasets). For the one-dimensional Kuramoto-Sivashinsky equation, we set $T = 100$ for the training dataset and $T = 200$ for the training dataset. For all three equations, we set $\Delta t = 0.2$. The spatial domain was set to $\mathcal{X} = [0, 2\pi]$ for the Burgers equation with $\Delta x = 2\pi/8192$, $\mathcal{X} = [0, 128]$ for the Korteweg-De Vries equation with $\Delta x = 128/1024$, and $\mathcal{X} = [0, 64]$ for the Kuramoto-Sivashinsky equation with $\Delta x = 64/4096$. For each equation, the spatial step $\Delta x$ was chosen to be as small as possible while maintaining trajectory generation under a pre-specified computational budget. All three domains had periodic boundaries. The initial conditions were sampled from a distribution over the truncated Fourier series with random coefficients $A_k \sim U(A_l, A_r)$, $l_k \sim \{l_a, l_b, l_c, l_d\}$, and $\phi_k \sim (\phi_l, \phi_r)$:

$$u_0(x) = \sum_{k=1}^{10} A_k \sin\left(\frac{2\pi l_k x}{L} + \phi_k\right),$$

where $L$ is the length of the spatial domain. Each trajectory was generated using the method of lines with the spatial derivatives computed using the pseudo-spectral method. For each equation, we selected a time-stepping method that balances accuracy and cost: RK23 for the Burgers equation, RK45 for the Korteweg-De Vries equation, and LSODA for the Kuramoto-Sivashinsky equation. See Table 2 for details.

| **Equation** | **Train $T$** | **Test $T$** | $\boldsymbol{\Delta t}$ | $\boldsymbol{\mathcal{X}}$ | $\boldsymbol{\Delta x}$ | $\{\mathbf{A_l}, \mathbf{A_r}\}$ | $\{\mathbf{l_a}, \mathbf{l_b}, \mathbf{l_c}, \mathbf{l_d}\}$ | $\{\boldsymbol{\phi}_\mathbf{l}, \boldsymbol{\phi}_\mathbf{r}\}$ | **Time-Stepping** |
|---|---|---|---|---|---|---|---|---|---|
| Burgers | 10 | 10 | 0.2 | $[0, 2\pi]$ | $2\pi/8192$ | $\{-0.5, 0.5\}$ | $\{3, 4, 5, 6\}$ | $\{0, 2\pi\}$ | RK23 |
| KdV | 100 | 100 | 0.2 | $[0, 128]$ | $128/1024$ | $\{-0.5, 0.5\}$ | $\{1, 2, 3, -\}$ | $\{0, 2\pi\}$ | RK45 |
| KS | 100 | 200 | 0.2 | $[0, 64]$ | $64/4096$ | $\{-0.5, 0.5\}$ | $\{1, 2, 3, -\}$ | $\{0, 2\pi\}$ | LSODA |

Table 2: Data generation settings.

We construct two additional datasets, short-horizon Korteweg-De Vries and short-horizon Kuratmoto-Sivashinsky, by considering the first 400 time steps to be part of a *warmup phase* and subsequently discarding them. See Table Table 3 for details.

For each of the one-dimensional equations, we generate 500 training trajectories and 100 testing trajectories. The data was initially generated using double-precision floating-point format (`float64`) and then converted into single-precision floating-point formation (`float32`) for our experiments.

| Equation | Warm-Up Steps | Train $T$ | Test $T$ |
|---|---|---|---|
| Short-Horizon KdV | 400 | 20 | 20 |
| Short-Horizon KS | 400 | 20 | 120 |
| Long-Horizon KdV | 0 | 100 | 100 |
| Long-Horizon KS | 0 | 100 | 200 |

Table 3: Short-horizon and long-horizon settings.

**Three-dimensional compressible Navier-Stokes dataset**

We use the three-dimensional compressible Navier-Stokes turbulence dataset provided by Takamoto et al. [2022]. This dataset consists of 600 trajectories, each containing 21 time steps, with 90% of the trajectories used for training and the remaining 10% reserved for testing. The turbulence initial condition considers turbulent velocity with uniform mass density and pressure. The initial velocity is defined as

$$\mathbf{v}(\mathbf{x}, t=0) = \sum_{i=1}^{4} A_i \sin(\mathbf{k}_i \cdot \mathbf{x} + \phi_i),$$

where the amplitude coefficients are

$$A_i = \frac{\bar{v}}{|\mathbf{k}_i|^2},$$

and the characteristic velocity $\bar{v} = c_s M$ is determined by the Mach number $M$ and the speed of sound

$$c_s = \sqrt{\frac{\Gamma p}{\rho}}.$$

To reduce compressibility effects, the compressible component of the velocity field is removed using a Helmholtz decomposition in Fourier space, resulting in a divergence-free velocity field that preserves turbulent structures while minimizing artificial acoustic modes.

The flow parameters are set to

$$(\eta, \zeta, M) = \left(10^{-2}, 10^{-2}, 1.0\right),$$

where $\eta$ and $\zeta$ are the shear and bulk viscosity coefficients, respectively, and $M$ is the initial Mach number.

The data are simulated using a second-order accurate HLLC Toro et al. [1994] scheme for the inviscid terms, the MUSCL Van Leer [1997] method for spatial reconstruction, and a central difference scheme for the viscous terms.

Each time step is composed by five channels: the three velocity components, pressure, and density, and each time steps is represented on a $64^3$ grid, resulting in $5 \times 64^3 = 5 \times 262,144 \approx 1.31 \times 10^6$ data points per step. The whole dataset size is 62 GB, indeed, due to memory constraints, training is performed by loading sub-batches of 32 samples directly from the hard disk where the dataset was stored. While this approach slows down training, it is necessary given the large dataset size.

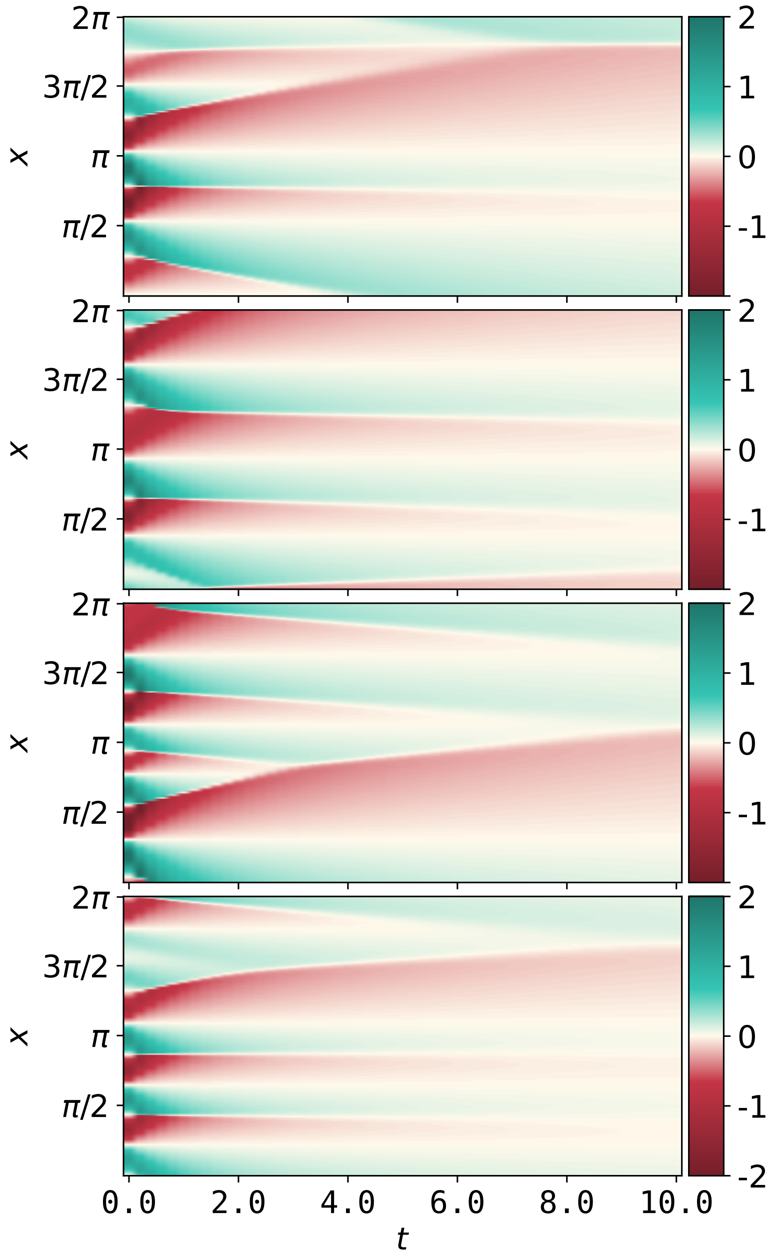

Figure 4: Example trajectories from the Burgers dataset. Train and test datasets share the same $T$.

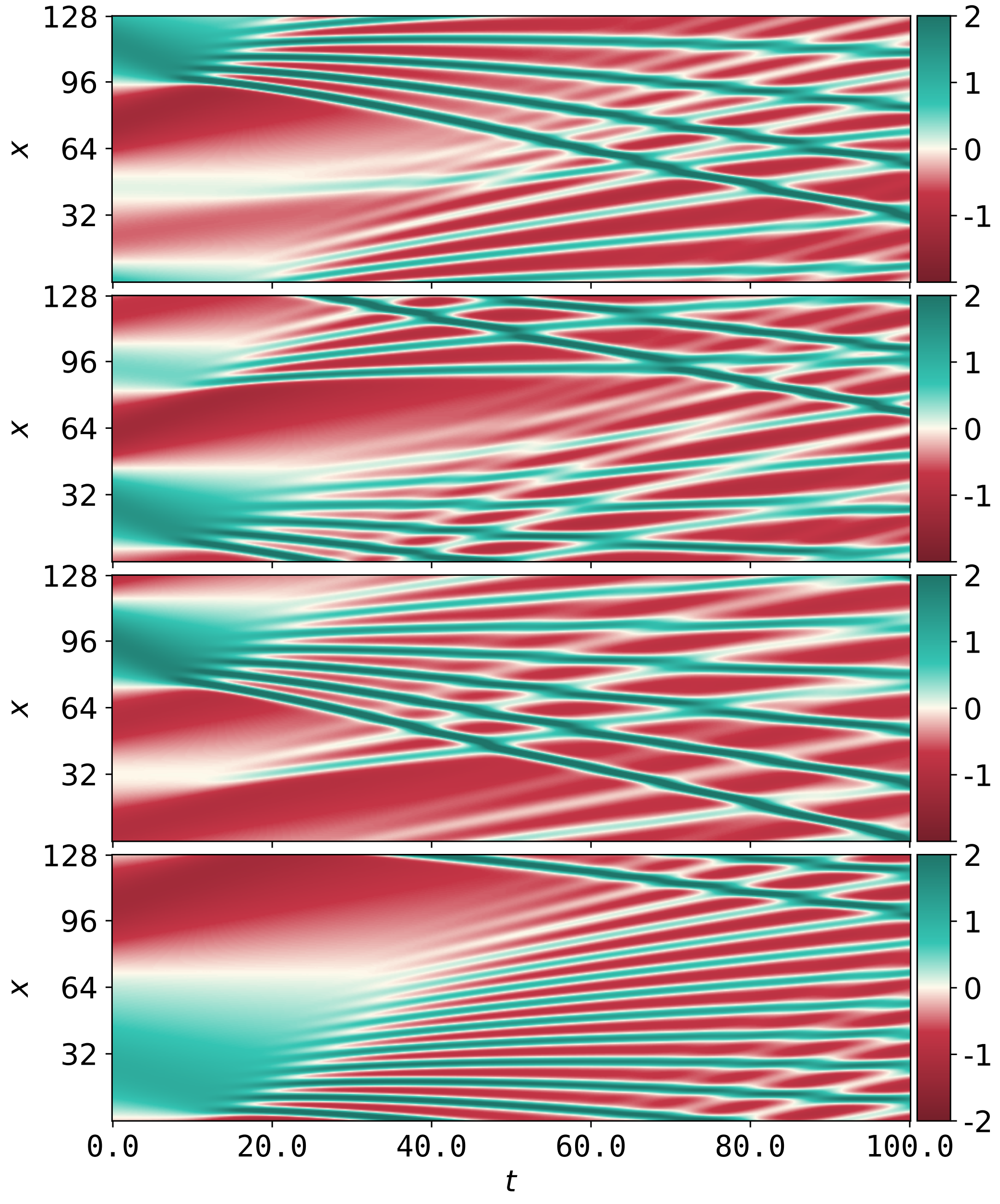

Figure 5: Example trajectories from the Korteweg-de Vries dataset. Train and test datasets share the same $T$.

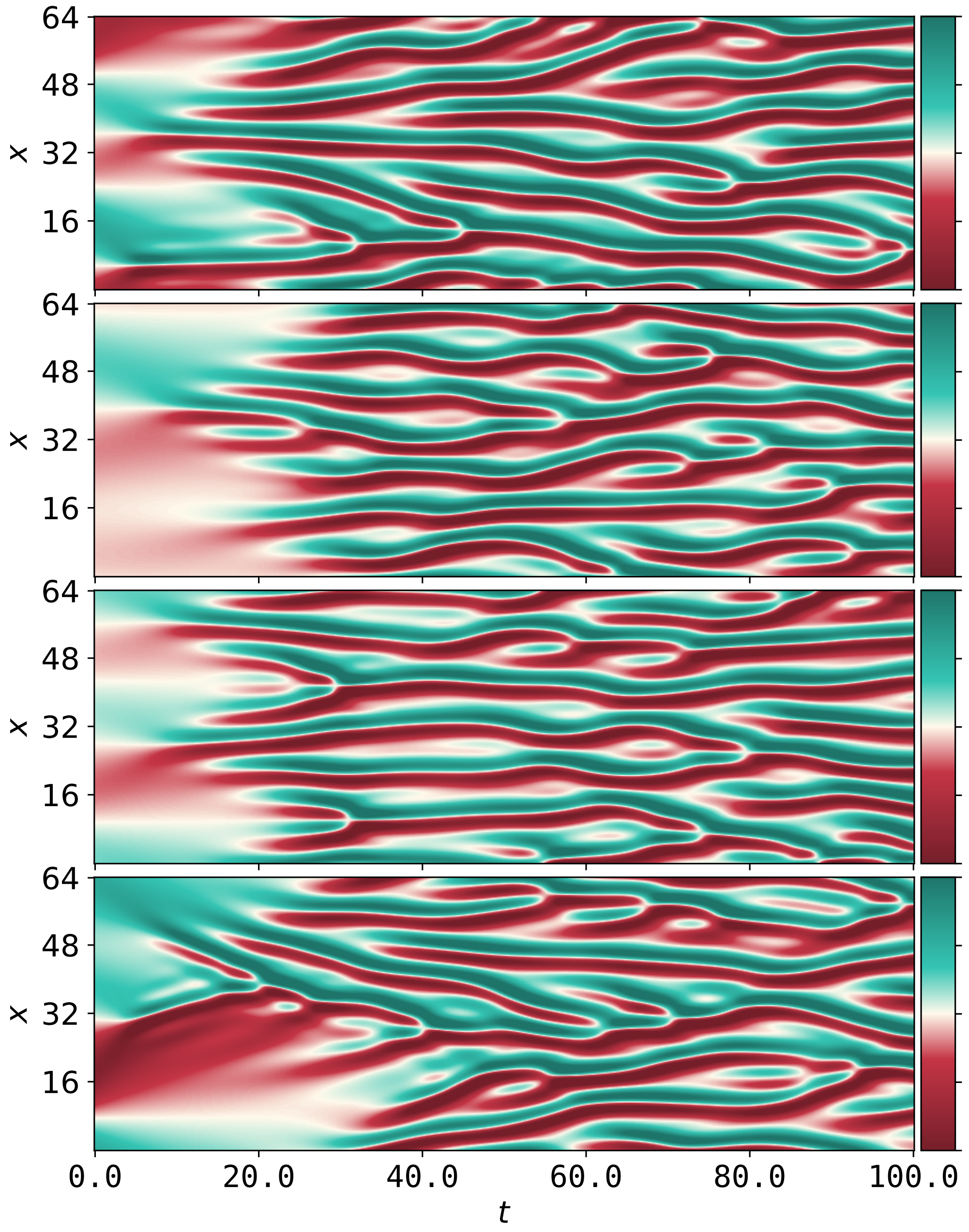

Figure 6: Example trajectories from the Kuramoto-Sivashinsky training dataset. The training dataset has $T = 100$, whereas the testing dataset has $T = 200$.

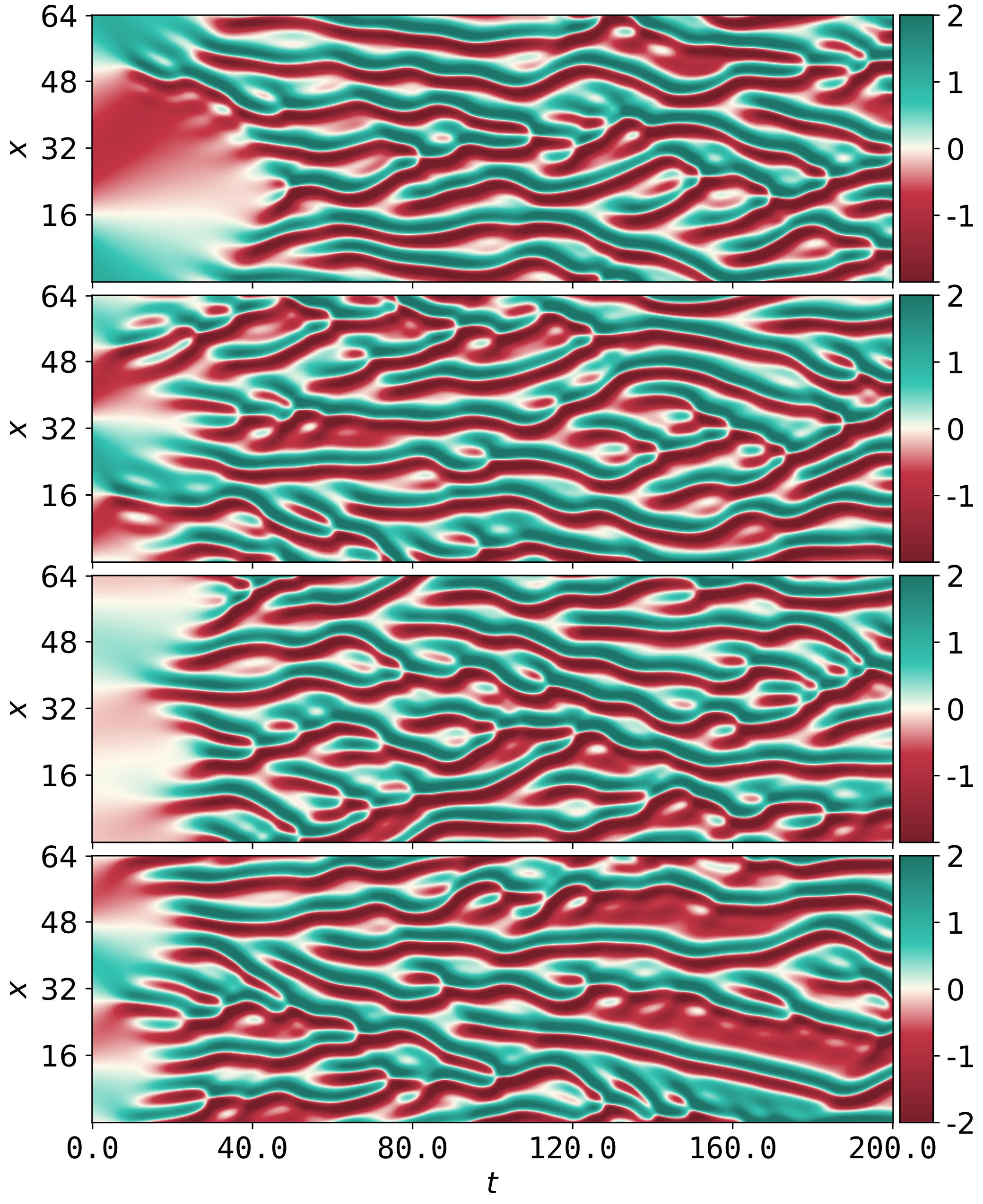

Figure 7: Example trajectories from the Kuramoto-Sivashinsky testing dataset. The training dataset has $T = 100$, whereas the testing dataset has $T = 200$.

## C Training Details

**Data Preparation.** To minimize the one-step loss $\mathcal{L} = ||U_{t+1} - \mathcal{G}_\theta(U_t)||_2^2$, we require input-output pairs. Consistent with prior work [Li et al., 2021], we set the prediction step size $\Delta t_p = 0.8$ and use residual prediction ($\mathcal{G}_\theta(\mathbf{U}_n) \approx \mathbf{U}_{n+1} - \mathbf{U}_n$) to balance short-term (one-step loss) and long-term (trajectory) performance. We also spatially downsample to 256 points. We scale each target by dividing by the maximum value across all trajectories, time steps, and spatial points; we found this to perform marginally better than normalizing to unit standard deviation.

**Neural Simulator Architectures.**

**Fourier Layer.** The Fourier layer transforms the input into the frequency domain using a fast Fourier transform (FFT), applies a truncated linear transformation to selected Fourier modes, and then maps the result back to the spatial domain via an inverse FFT. This spectral transformation is typically combined with a skip connection consisting of a point-wise convolution, a bias term, and an activation function. Formally, for an input $\mathbf{x} \in \mathbb{R}^n$, the layer computes:

$$\mathrm{F}(\mathbf{x}) = \sigma(\mathcal{F}^{-1}(R \cdot \mathcal{F}(\mathbf{x})) + W\mathbf{x} + \mathbf{b}),$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the FFT and inverse FFT respectively, $R : \mathbb{R}^n \to \mathbb{R}^{n'}$ is a learned linear transformation in frequency space, $W : \mathbb{R}^n \to \mathbb{R}^{n'}$ represents a point-wise convolution, and $\mathbf{b}$ is a bias term.

Several variations of the Fourier layer have been proposed. One such variant [Tran et al., 2021] modifies the layer by introducing a residual connection and a two-layer feedforward network, while omitting the point-wise convolution and bias term:

$$\mathrm{F}(\mathbf{x}) = \mathbf{x} + \sigma(W_2\sigma(W_1\mathcal{F}^{-1}(R \cdot \mathcal{F}(\mathbf{x})) + \mathbf{b}_1) + \mathbf{b}_2).$$

In our early experiments, this modification did not yield noticeable improvements. We also explored simply adding a skip connection without the feedforward block and inserting normalization layers at various points in the architecture, but these did not result in noticeable improvements.

**RDS$_{\text{Fourier}}$.** The Fourier Neural Operator is made of a point-wise lifting layer, followed by a sequence of Fourier layers, and then a point-wise projection. The Recurrent Depth Simulator (RDS) with Fourier layers can be interpreted in two ways: 1) RDS$_{\text{Fourier}}$ `wo/ EncDec`, a point-wise lifting layer encoder, followed by a sequence of Fourier layers (that make up the recurrent-depth block), and then a point-wise projection layer decoder, or 2) RDS$_{\text{Fourier}}$ `w/ EncDec`, where the first Fourier layer is part of the encoder and the last Fourier layer is part of the decoder. We find that RDS$_{\text{Fourier}}$ `w/ EncDec` often leads to more consistent and superior performance.

**RDS.** The Recurrent Depth Simulator is a highly flexible framework. Each component—the encoder, recurrent-depth block, and decoder—may be instantiated with any layer(s) depending on the task. For example, in problems with periodic boundaries and a requirement of parameter efficiency, where the Fourier Neural Operator would typically shine, Fourier layers can be used. On the other hand, if the goal is to develop a foundation model for physics on irregular meshes, where one might use a graph-based encoder, with an attention-based bottleneck, and a graph-based decoder, the RDS framework can be configured accordingly. With just a few additional lines of code, RDS enables explicit control over the accuracy-cost trade-off (see Appendix H for pseudocode).

Fourier- and attention-based layers are well-suited for recurrent-depth blocks due to their ability to model infinite receptive fields. In contrast, convolutional-based layers have a fixed receptive field that grow with the depth. For example, a standard convolutional layer in `PyTorch` with `kernelsize=3`, `dialation=1`, and $stride = 1$ has a receptive field of size 3. Stacking two such layers increases he receptive field to 5—capturing the center point and two neighboring point on each side. More generally, the receptive field after stacking $L$ such layers is given by $L \cdot (\texttt{kernelsize}/2) + 1$. To achieve a receptive field of size 64, to effectively model the Burgers equation, one would need to stack 63 layers. In RDS, where $K = 1$ could be sampled, 63 layers would need to be distributed across the encoder, recurrent-depth block, and decoder. To mitigate this, some alternatives can be considered to expand the receptive field more efficiently: increasing the kernel size, incorporating attention-based layers, or adding downsampling blocks.

**FNO-DEQ.** Similarly to Marwah et al. [2023], we use Anderson acceleration with a maximum of 16 iterations. For the backward pass of the DEQ layer, we follow the phantom gradient approach proposed by Geng et al. [2021], using parameters $s = 3$ and $\tau = 0.8$. To match the parameter count of $\text{RDS}_{\text{Fourier}}$, we employ a 1D FNO with 8 layers and 120 channels.

**ACDM.** We follow the original setup from Kohl et al. [2023], using a linear scheduler and training with a maximum of 50 diffusion steps. For conditioning, we concatenate the snapshot from the previous time step, i.e., the solution $u_t$ when predicting $u_{t+1}$. To ensure a fair comparison, we condition only on $u_t$ and do not include earlier time steps.

**PDE-Refiner** We use the same scheduler proposed by Lippe et al. [2023], with $\sigma^2_{\text{min}} = 2 \cdot 10^{-7}$ and $K = 10$. Following a similar approach to Kohl et al. [2023], we implement the following algorithm from scratch:

**Algorithm 3** PDE-Refiner: Training and Inference Procedures

```
1:  procedure TRAINSTEP($u_t, u_{\text{prev}}$)
2:      $k \leftarrow$ random integer in $[0, \text{num\_steps}]$
3:      if $k = 0$ then
4:          pred $\leftarrow$ NeuralOperator(zeros_like($u_t$), $u_{\text{prev}}, k$)
5:          target $\leftarrow u_t$
6:      else
7:          noise_std $\leftarrow$ min_noise_std$^{k/\text{num\_steps}}$
8:          noise $\leftarrow$ randn_like($u_t$)
9:          $u_{t,\text{noised}} \leftarrow u_t +$ noise $\cdot$ noise_std
10:         pred $\leftarrow$ NeuralOperator($u_{t,\text{noised}}, u_{\text{prev}}, k$)
11:         target $\leftarrow$ noise
12:     end if
13:     loss $\leftarrow$ MSE(pred, target)
14:     return loss
15: end procedure

16: procedure PREDICTNEXTSOLUTION 1($u_{\text{prev}}$)
17:     $u_{\hat{t}} \leftarrow$ NeuralOperator(zeros_like($u_{\text{prev}}$), $u_{\text{prev}}, 0$)
18:     for $k = 1$ to num_steps do
19:         noise_std $\leftarrow$ min_noise_std$^{k/\text{num\_steps}}$
20:         noise $\leftarrow$ randn_like($u_t$)
21:         $u_{\hat{t},\text{noised}} \leftarrow u_{\hat{t}} +$ noise $\cdot$ noise_std
22:         pred $\leftarrow$ NeuralOperator($u_{\hat{t},\text{noised}}, u_{\text{prev}}, k$)
23:         $u_{\hat{t}} \leftarrow u_{\hat{t},\text{noised}} -$ pred $\cdot$ noise_std
24:     end for
25:     return $u_{\hat{t}}$
26: end procedure
```

Algorithm 3 is taken from Lippe et al. [2023], and the number of inference num_steps is fixed at test time. To adapt the original algorithm, we investigated two variations: Algorithm 4 and 5. When $\bar{K} =$ num_steps, both methods recover the original procedure proposed in Lippe et al. [2023].

The first variation, Algorithm 4, adjusts the noise scheduler based on the number of inference steps. However, this strategy only performs well when the number of steps matches the training setup. To address this limitation, we introduce Algorithm 5, which retains the noise scheduler from training while allowing the number of inference steps to vary. This consistency in noise levels enhances stability and performance by preserving the distribution the network was trained on.

**Algorithm 4** Predict Next Solution 1

```
procedure PREDICTNEXTSOLUTION($u_{\text{prev}}$)
    $u_{\hat{t}} \leftarrow \text{NeuralOperator}(\text{zeros\_like}(u_{\text{prev}}), u_{\text{prev}}, 0)$
    for $k = 1$ to K do
        noise_std ← min_noise_std$^{k/\mathbf{K}}$
        noise ← randn_like($u_t$)
        $u_{\hat{t},\text{noised}} \leftarrow u_{\hat{t}} + \text{noise} \cdot \text{noise\_std}$
        pred ← NeuralOperator($u_{\hat{t},\text{noised}}, u_{\text{prev}}, k$)
        $u_{\hat{t}} \leftarrow u_{\hat{t},\text{noised}} - \text{pred} \cdot \text{noise\_std}$
    end for
    return $u_{\hat{t}}$
end procedure
```

**Algorithm 5** Predict Next Solution 2

```
 1: procedure PREDICTNEXTSOLUTION($u_{\text{prev}}$)
 2:     $u_{\hat{t}} \leftarrow \text{NeuralOperator}(\text{zeros\_like}(u_{\text{prev}}), u_{\text{prev}}, 0)$
 3:     for $k = 1$ to K do
 4:         noise_std ← min_noise_std$^{k/\text{num\_steps}}$
 5:         noise ← randn_like($u_t$)
 6:         $u_{\hat{t},\text{noised}} \leftarrow u_{\hat{t}} + \text{noise} \cdot \text{noise\_std}$
 7:         pred ← NeuralOperator($u_{\hat{t},\text{noised}}, u_{\text{prev}}, k$)
 8:         $u_{\hat{t}} \leftarrow u_{\hat{t},\text{noised}} - \text{pred} \cdot \text{noise\_std}$
 9:     end for
10:     return $u_{\hat{t}}$
11: end procedure
```

**Optimzation.** All optimization hyperparameters are listed in Table 4 and remain fixed across all experiments, except where explicitly stated. We train each model for 100 epochs using the AdamW optimizer [Loshchilov and Hutter, 2019], starting with a learning rate of $3 \times 10^{-4}$ and a weight decay of $1 \times 10^{-5}$. A cosine annealing schedule is applied to gradually reduce the learning rate to $3 \times 10^{-6}$ [Loshchilov and Hutter, 2017b]. In early experiments, we observed that using a higher initial learning rate (e.g., $1 \times 10^{-3}$) led to less consistent performance, though it occasionally improved performance [Sohl-Dickstein, 2024].

| **Hyperparameter** | **Value** |
|---|---|
| Epochs | 100 |
| Batch Size | 256 [1] |
| Optimizer | AdamW |
| Starting Learning Rate | $3 \times 10^{-4}$ |
| Weight Decay | $1 \times 10^{-5}$ |
| Scheduler | Cosine Annealing |
| Ending Learning Rate | $3 \times 10^{-6}$ |

Table 4: Optimization hyperparameters used in all experiments.

[1]For the three-dimensional experiments, we use a batch size of 32, and perform gradient accumulation to have an effective batch size of 256.

## D Backpropagation Window

During training, the recurrent-depth block is repeated $K$ times in the forward pass, after which gradients are propagated backward through the same computation. If $K$ is large, which could happen because $K$ is drawn from a long-tailed distribution, the backward pass must retain every intermediate activation, quickly exhausting GPU memory. To cap the memory usage, we use truncated backpropagation-through-time with a fixed backpropagation window $B$: gradients are backpropagated through at most the last $B$ steps, and earlier steps are treated as constants. This bounds memory at $O(B)$ independent of $K$. In this experiment, we study it truncated backpropagation-through-time is viable and the effect of different backpropagation windows.

**Experimental Setup.** We conduct experiments on three datasets: Burgers, long-horizon KdV, and long-horizon KS. We train a recurrent depth simulator with a point-wise lifting layer, a recurrent-depth block with a single Fourier layer, and a point-wise projection layer with $\sim$ 1M parameters. We set $\bar{K} = 32$, and the backpropagation window is swept over $B \in \{1, 2, 4, 16, 32\}$. With $B = 1$ the compute for the forward pass is equivalent to a Fourier layer with 33 layers, but the backward pass stores only a single activation; with $B = 32$ the backward pass stores every activation whenever $K \leq 32$ and the last 32 when $K > 32$. This would be infeasible for higher-dimensional problems.

**Results.** We report the trajectory errors in Table 5. Across all equation $B = 1$ performs worst and moving from $B = 1$ to $B = 2$ yields the largest gain, and improvements largely saturate by $B = 4$. Beyond $B = 4$, larger windows offer only marginal benefit while reinstating a substantial memory cost. Note that although trajectory error is not the preferred metric for KS, the same saturation is evident. Based on these results, and to balance accuracy and memory, we set $B = 4$ in all main experiments.

| **Backpropagation Window $B$** | **Burgers** | **Korteweg-De Vries** | **Kuramoto-Sivashinsky** |
|---:|---:|---:|---:|
| 1 | 0.0849 | 0.1046 | 1.6341 |
| 2 | 0.0315 | 0.0522 | 1.4097 |
| 4 | 0.0199 | 0.0317 | 1.3972 |
| 16 | 0.0181 | 0.0302 | 1.3960 |
| 32 | 0.0178 | 0.0298 | 1.3910 |

Table 5: Impact of the back-propagation window $B$ on trajectory error. Accuracy improves sharply up to $B = 4$ and then plateaus.

## E Distribution Parameter $\bar{\mathbf{K}}$

The distribution parameter $\bar{K}$ controls the expected number of recurrent steps during training. Setting $\bar{K}$ too low shortens training time but may leave the model under-exposed to large $K$ values during inference; setting it too high increases training time. In this experiment, we wish to identify the optimal $\bar{K}$.

**Experimental Setup.** We conduct experiments on three datasets: Burgers, long-horizon KdV, and long-horizon KS. We train a recurrent depth simulator with a point-wise lifting layer, a recurrent-depth block with a single Fourier layer, and a point-wise projection layer with $\sim$ 1M parameters. The backpropagation window is fixed at $B = 4$, and $\bar{K}$ is swept over $\{1, 2, 4, 8, 16, 32, 64, 128\}$. Doubling $\bar{K}$ roughly doubles the forward cost, yet backward memory remains capped by $B$; for instances, $\bar{K} = 8$ matches the forward FLOPS of an 8-layer FNO but the truncated-backpropagation-through-time keeps the backward pass FLOPs as cheap as a 4-layer FNO. After training, each model is evaluated at all values $K \in [1, 2\bar{K}]$ and we report the lowest trajectory error achieved.

**Results.** Figure 8-10 plot trajectory error as a function of $\bar{K}$. Increasing $\bar{K}$ consistently lowers the best achievable trajectory error, but we observe diminishing returns beyond $\bar{K} \approx 32$. We also notice that models trained with larger $\bar{K}$ underperform with small $K$ values (see Figure 15). In other words, the additional training compute shifts the accuracy-cost curve to the right and gains appear only once $K$ is allowed to grow. Based on these results, we set $\bar{K} = 32$ in our main experiments as it captures the bulk of the benefit of high-compute settings while leaving the model competitive in low-compute settings.

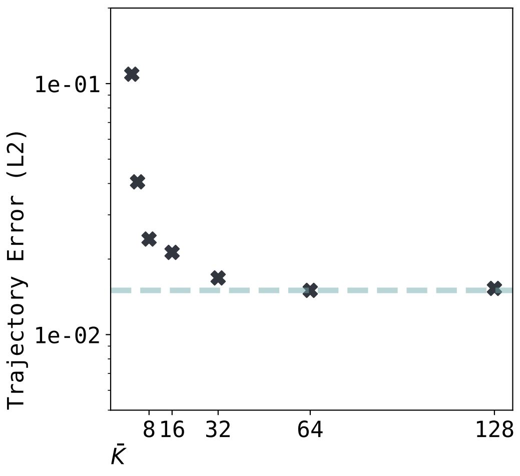

Figure 8: Choosing the distribution parameter $\bar{K}$ on the Burgers dataset.

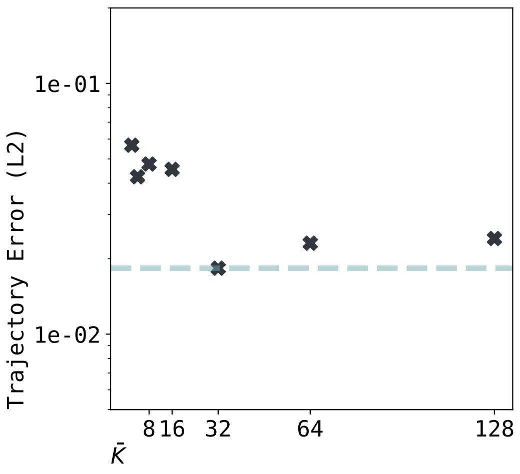

Figure 9: Choosing the distribution parameter $\bar{K}$ on the KdV dataset.

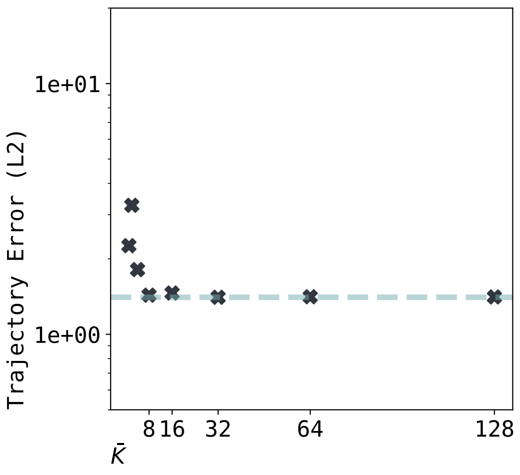

Figure 10: Choosing the distribution parameter $\bar{K}$ on the KS dataset.

## F Merging

At each recurrent step, the recurrent depth simulator must merge the condition vector $\mathbf{c}$ with the latent vector $\mathbf{z}_k$. We consider six merging methods of increasing capacity. **Add** simply sums the two vectors. **Add$_\mathbf{s}$** introduces two learnable parameters $\alpha$ and $\beta$ ($\mathbf{z}'_k = \alpha\mathbf{c} + \beta\mathbf{z}_k$). **Add$_\mathbf{e}$** generalizes this to element-wise vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ ($2 \times$ `hiddenchannels` additional trainable parameters). **Projection** concatenates $[\mathbf{c}, \mathbf{z}_k]$ and applies a point-wise linear map ($2 \times$ `hiddenchannels` $\times$ `hiddenchannels` additional trainable parameters); **Projection$_\mathbf{I}$** uses the same layer but is initialized with 1s along the diagonals and 0s everywhere else, so that it is equivalent to **Add$_\mathbf{e}$** at initialization but with increased capacity. **Concat** feeds the raw concatenation into the first layer (in the recurrent-depth block), doubling its input channels, and thus, trainable parameters. In this experiment, our goal is to test these merging methods.

**Experimental Setup.** All experiments run on the one-dimensional Burgers equation. The base architecture is fixed—a point-wise lift, a single Fourier layer encoder, a one-layer Fourier recurrent block, and a Fourier decoder with point-wise projection—trained with $\bar{K} = 32$ and back-propagation window $B = 4$. We sweep five parameter budgets $\{0.2\text{M}, 0.5\text{M}, 1.0\text{M}, 2.0\text{M}, 4.0\text{M}\}$ by scaling channel width, and implement each of the six merging methods at every budget. After training, each model is evaluated at all values $K \in [1, 2\bar{K}]$ and we report the lowest trajectory error achieved.

**Results.** Table Table 6 reports the lowest trajectory error for every configuration. The three addition variants perform almost identically and improve monotonically with parameter count. The **Projection** variant lags behind, but when initialized with 1s along the diagonals (**Projection$_\mathbf{I}$**), it matches or exceeds the additional family. **Concat** attains the lowest error overall, but at the price of $\sim 33\%$ extra parameters in the recurrent-block's first layer; we hypothesis that part of its gain stems from increased model size rather than a superior merging mechanism.

| **Parameters** | **Add** | **Add$_\mathbf{s}$** | **Add$_\mathbf{e}$** | **Projection** | **Projection$_\mathbf{I}$** | **Concat** |
|---|---|---|---|---|---|---|
| $\sim$ 0.2M | 0.0234 | 0.0230 | *0.0229* | 0.0240 | 0.0240 | **0.0214** |
| $\sim$ 0.5M | 0.0176 | 0.0173 | 0.0172 | 0.0223 | **0.0135** | *0.0146* |
| $\sim$ 1.0M | 0.0129 | *0.0126* | *0.0126* | 0.0169 | **0.0101** | 0.0151 |
| $\sim$ 2.0M | 0.0116 | 0.0115 | 0.0115 | 0.0094 | *0.0093* | **0.0090** |
| $\sim$ 4.0M | 0.0100 | *0.0098* | 0.0099 | 0.0110 | 0.0100 | **0.0083** |

Table 6: Trajectory error on Burgers for six merging methods across five parameter budgets. Best result in each row is **bold**, second-best *italic*.

704 ## G More Experiments

705 ### G.1 Experiment: Accuracy-Cost Trade-Off (Extended)

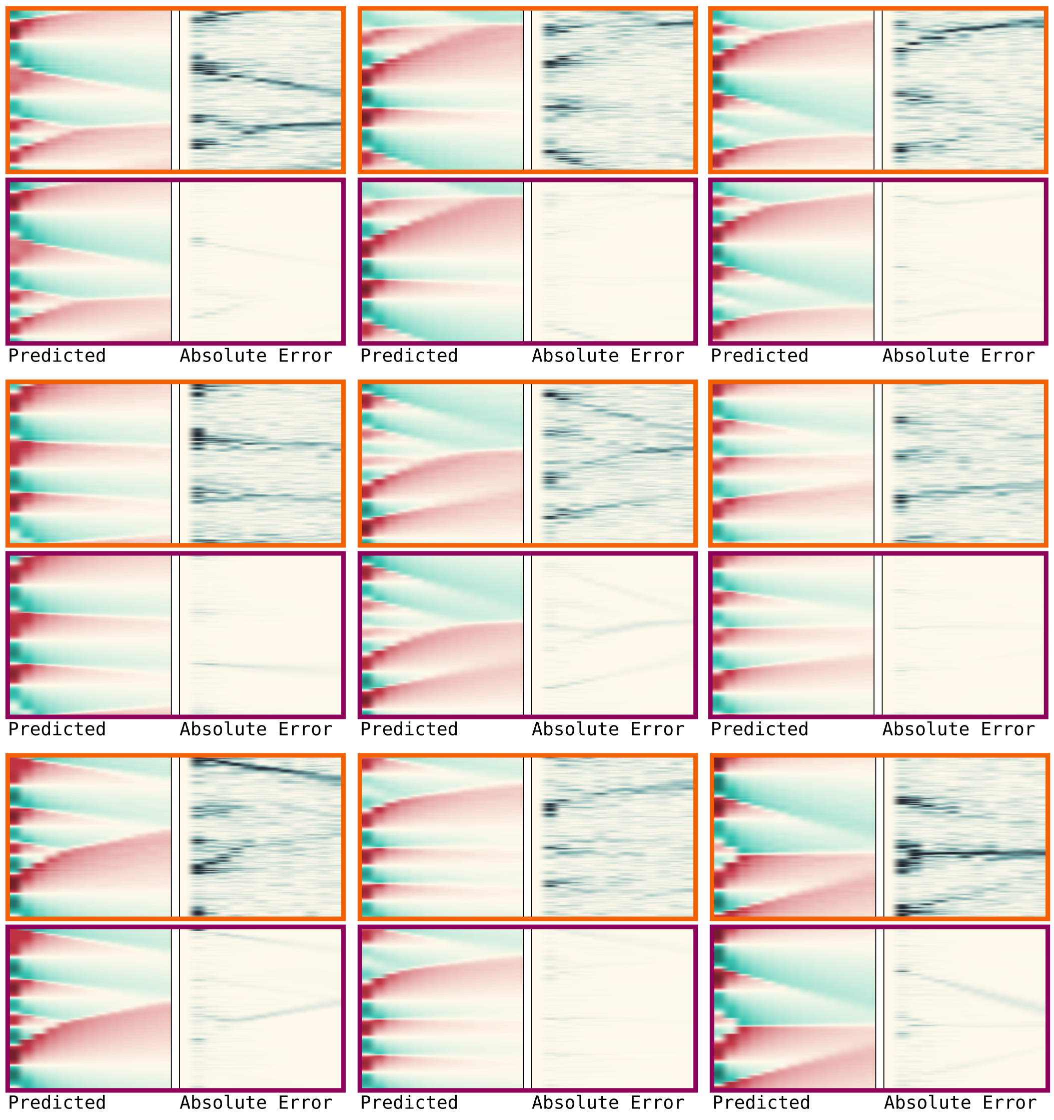

Figure 11: Burgers: Trajectories at $K = 4$ (orange) and $K = 16$ (purple).

706

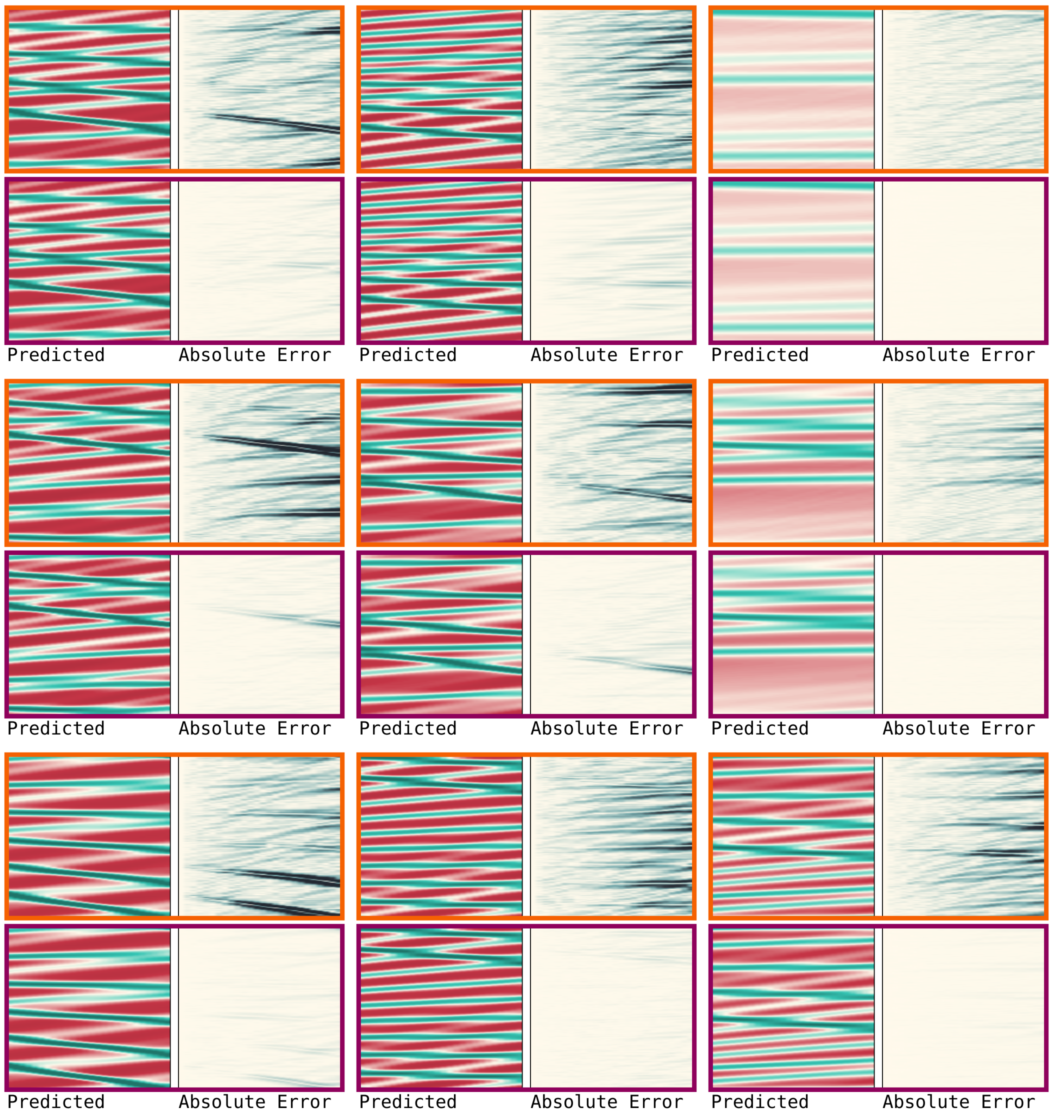

Figure 12: Short-Horizon KdV: Trajectories at $K = 4$ (orange) and $K = 16$ (purple).

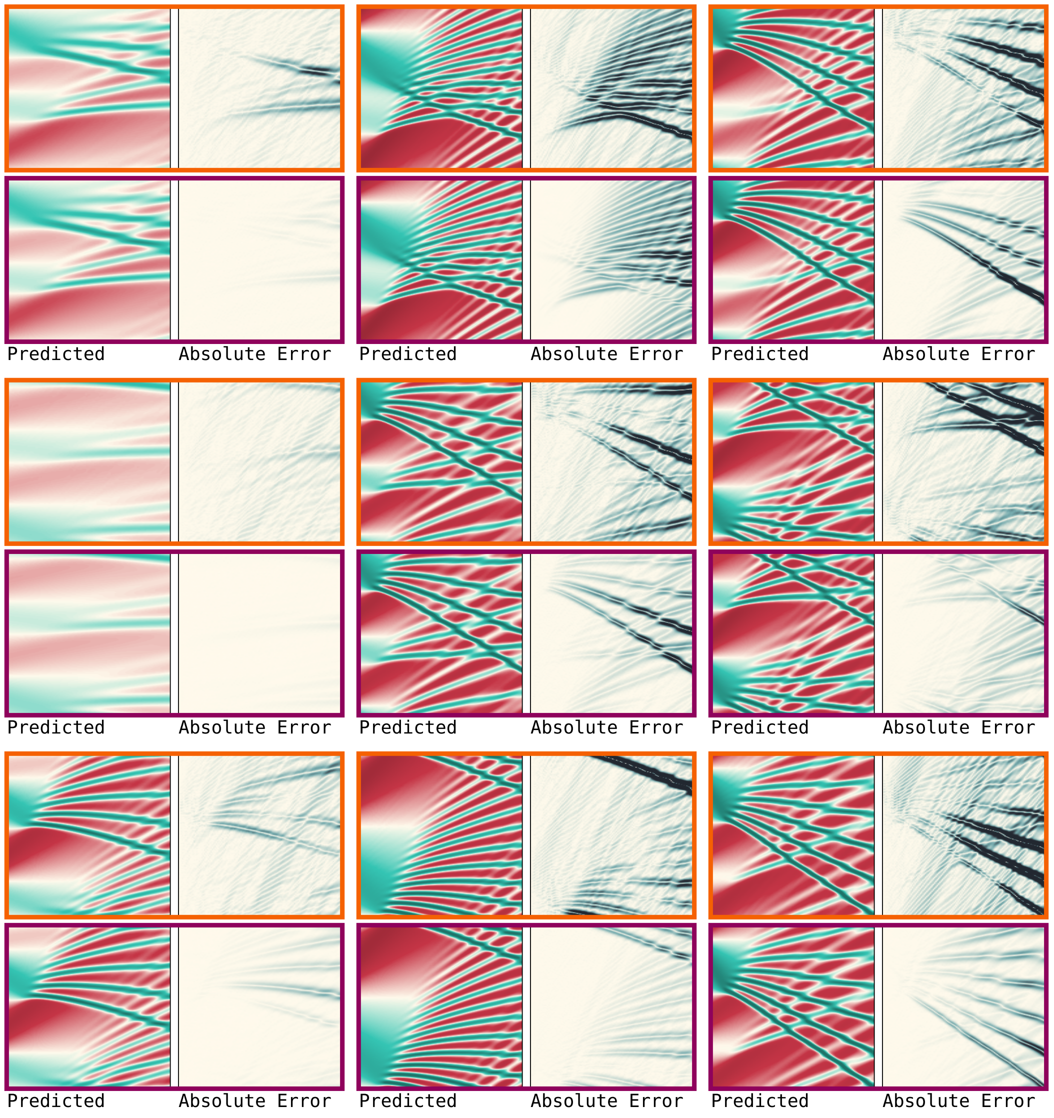

Figure 13: Long-Horizon KdV: Trajectories at $K = 4$ (orange) and $K = 16$ (purple).

## G.2 Experiment: Alternatives (Extended)

On the chaotic Kuramoto-Sivashinsky dataset we replace trajectory error with the average and worst-case correlation horizon metrics. Figure 14 shows the behavior of the four adaptive-compute simulators across 30 correlation thresholds ($\alpha = 0.70 - 0.99$) and all inference depths $K \in \{1, \dots, 16\}$. $\text{RDS}_{\text{Fourier}}$ (first column) shows the desired monotone pattern: both the average and the worst-cast correlation horizons rise steadily with $K$. FNO-DEQ delivers flat surfaces—its iterations leave the horizon essentially unchanged—so it cannot expliot extra compute. ACDM begins with short horizons, improves up to $K \approx 4$, and then flattens; only a narrow band of $K$ values is usable, limiting its test-time flexibility. PDE-Refiner gains up to $K \approx 8$ but then oscillates, making it hard to pick a reliable stopping point. Across both average and worst-case statistics RDS attains the longest horizons and is the only model whose accuracy scales predictably with additional compute, confirming its advantage for controllable accuracy-cost trade-offs in chaotic regimes.

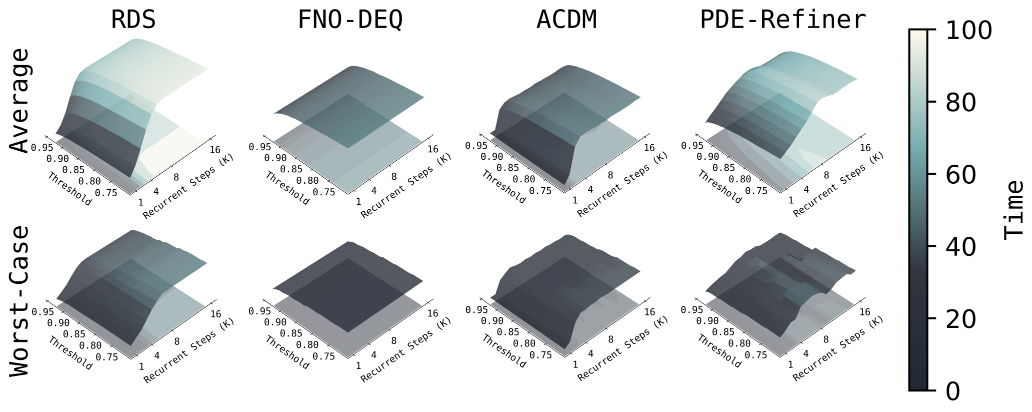

Figure 14: Kuramoto–Sivashinsky: average (top) and worst-case (bottom) correlation horizons and threshold $\alpha$ versus inference depth $K$.

## G.3 Experiment: Large-Scale Compressible Navier-Stokes (Extended)

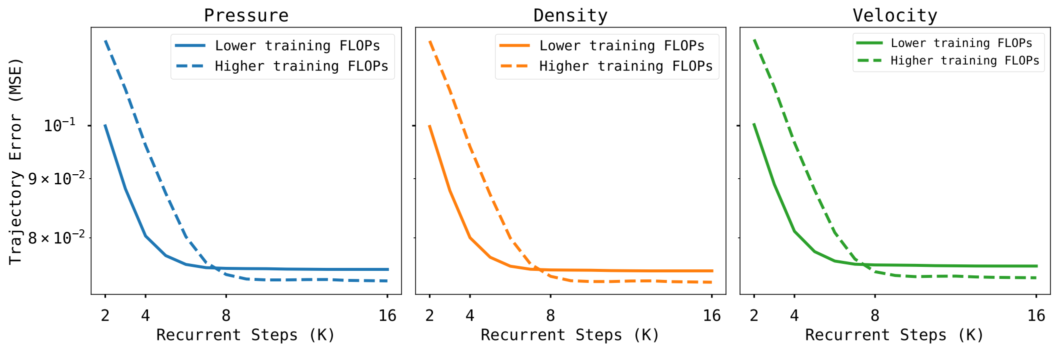

Figure 15: Trajectory error (MSE) over the number of recurrent steps $K$ for two $\text{RDS}_{\text{Fourier}}$ models, trained with lower and higher FLOPs budgets, respectively.

As shown in fig. 15, the two models present distinct trade-offs. When the number of recurrent steps during inference exceeds 8, the model trained with a higher FLOPs budget and a higher $\bar{K}$ yields significantly lower MSE. In contrast, for fewer than 8 recurrent steps, the model trained with a lower computational budget performs better.

## H Pseudocode

```
class Network(Module):
    def __init__(self):
        super().__init__()
        # Encoder Layer
        self.encoder = Layer()

        # Collect L Intermediate Layers
        layers = []
        for _ in range(L):
            layers.append(Layer())

        # Decoder Layer
        self.decoder = Layer()

    def forward(self, x):
        # Apply Encoder
        z = self.encoder(x)

        ######################
        ##### Main Block #####

        # Apply L Intermediate Layers
        for layer in self.layers:
            z = layer(z)

        ##### Main Block #####
        ######################



























        # Apply Decoder
        x = self.decoder(z)
        return x
```

Listing 1: Pseudocode of a *standard* neural simulator. The neural simulator contains an encoder or lifting layer (`self.encoder`), L intermediate layers of any type (residual layers, Fourier layers, etc.), and an decoder or projection layer (`self.decoder`).

```
class Network(Module):
    def __init__(self):
        super().__init__()
        # Encoder Layer
        self.encoder = Layer()

        # Collect L Intermediate Layers
        layers = []
        for _ in range(L):
            layers.append(Layer())

        # Decoder Layer
        self.decoder = Layer()

    def forward(self, x, K=None):
        # Apply Encoder
        c = self.encoder(x)

        ######################
        ##### Main Block #####

        # Sample Noise \w 'shape=x.shape'
        z = sample_noise()

        # During Inference:
        if not self.training:
            # Loop K Times
            for _ in range(K):
                # Concatenate x and z
                z = cat([c, z], dim=1)
                # Apply L Intermediate Layers
                for layer in self.layers:
                    z = layer(z)

        # During Training:
        if self.training:
            # Do Not Use Grad
            with no_grad():
                # Sample K (Using K_bar)
                K = sample_K()
                # Loop K - B Times
                for _ in range(K - B):
                    z = cat([c, z], dim=1)
                    for layer in self.layers:
                        z = layer(z)
            # Loop Remaining B Times
            for _ in range(B):
                z = cat([c, z], dim=1)
                for layer in self.layers:
                    z = layer(z)

        ##### Main Block #####
        ######################

        # Apply Decoder
        x = self.decoder(z)
        return x
```

Listing 2: Pseudocode of the Recurrent Depth Simulator—fewer than 20 new lines compared to a *standard* neural simulator. During inference, we apply the intermediate layers `K` times. During training, we apply the intermediate layers `K - B` times without gradient, and `B` times with gradient. Nothing else needs to change.

## I Extended Related Work

**Deep Equilibrium Models.** Deep Equilibrium Models (DEQs), introduced by Bai et al. [2019], are implicit, infinite-depth, weight-tied neural networks. A DEQ directly solves for the fixed point of a nonlinear transformation using any black-box root-finding algorithm and instead of backpropagating through each layer, which can be infeasible due to memory and numerical stability, the DEQ makes use of the Implicit Function Theorem to compute the gradients at the equilibrium—this approach has a constant memory requirement regardless of depth. Although the existence of the fixed point, or convergence to the fixed point, is not guaranteed; on large-scale language modeling tasks, Bai et al. [2019] demonstrated that DEQs can achieve performance comparable with state-of-the-art while using significantly less memory. Later, Bai et al. [2020] extended DEQs to large-scale computer vision tasks, showing similar performance and memory benefits. Subsequent research explored DEQs for various applications. Pokle et al. [2022] represent the entire sampling process in denoising diffusion implicit models as a single fixed-point system. Geng et al. [2023] distill diffusion models, directly from initial noise to the final image, into a DEQ. In inverse problems, Gilton et al. [2021] model a, potentially infinite, iterative reconstruction scheme as a DEQ. For partial differential equations, Pokle et al. [2022] propose FNO-DEQ, a DEQ variant with Fourier layers, to solve steady-state PDEs, showing improvements in accuracy and robustness to noise over baselines with four times as many parameters.

**Denoising Diffusion Models.** First introduced by Sohl-Dickstein et al. [2015], diffusion models are probabilistic models with an iterative forward diffusion process and a learned reverse diffusion process. The forward process gradually adds noise to data until only noise remains, and the reverse process gradually removes noise to restore the original data. New samples are generated by sampling a noise vector and passing it through the reverse process. Ho et al. [2020] presented high-quality image synthesis results using diffusion models. Dhariwal and Nichol [2021] and Karras et al. [2022] made further progress leading to state-of-the-art results and widespread adoption. Diffusion models have been applied to image generation [Nichol et al., 2021, Ramesh et al., 2022, Saharia et al., 2022b], image inpainting and outpainting [Saharia et al., 2022a], super-resolution [Saharia et al., 2022c], audio generation [Chen et al., 2020, Kong et al., 2020], text generation [Austin et al., 2021], including large language (diffusion) models [Nie et al., 2025]. In scientific domains, diffusion models have been applied to medium-range weather forecasting [Price et al., 2023], structure-based drug design [Schneuing et al., 2024], and stable materials generation [Yang et al., 2023]. Kohl et al. [2023] demonstrated that diffusion models are viable for turbulent flow simulation. Their results show that diffusion models outperform, in terms of long-term accuracy and stability, more efficient (and more commonly used) neural simulators. Kohl et al. [2023] also compared against PDE-Refiner [Lippe et al., 2023], a diffusion-based multi-step refinement process, but found that PDE-Refiner is highly sensitive to hyperparameters, and in some cases, generated substantially worse results compared to other methods.

## J Extended Discussion

To our knowledge, this is the first work to study neural simulators in terms of *test-time control of accuracy-cost trade-offs*. Since the performance varies with the chosen number of recurrent steps $K$, a scalar metric is no longer adequate; our experiments therefore focus on full accuracy-cost curve, and correlation-horizon surfaces. Across all tasks, the Recurrent-Depth Simulator provides a smooth, monotone trade-off, demonstrating that adaptive compute is possible, and we hope these results stimulate further work along this new axis.

Although the main experiments concentrate on RDS instantiated with Fourier layers—chosen for their infinite receptive field (see Appendix C)—preliminary tests with convolutional blocks yield qualitatively similar results. We also use a recurrent-block with a single-layer for clarity: it delivers the most predictable behavior, however, deeper blocks also showed strong performance. Exploring richer blocks and alternative layer types under this controllable-compute paradigm remains a promising direction for future research.