Supplementary material
PDE-Refiner: Achieving Accurate Long Rollouts with Neural PDE Solvers

Table of Contents

A Broader Impact

Neural PDE solvers hold significant potential for offering computationally cheaper approaches to modeling a wide range of natural phenomena than classical solvers. As a result, PDE surrogates could potentially contribute to advancements in various research fields, particularly within the natural sciences, such as fluid dynamics and weather modeling. Further, reducing the compute needed for simulations may reduce the carbon footprint of research institutes and industries that rely on such models. Our proposed method, PDE-Refiner, can thereby help in improving the accuracy of these neural solvers, particularly for long-horizon predictions, making their application more viable.

However, it is crucial to note that reliance on simulations necessitates rigorous cross-checks and continuous monitoring. This is particularly true for neural surrogates, which may have been trained on simulations themselves and could introduce additional errors when applied to data outside of its original training distribution. Hence, it is crucial for the underlying assumptions and limitations of these surrogates to be well-understood in applications.

B Reproducibility Statement

To ensure reproducibility, we report the used model architectures, hyperparameters, and dataset properties in detail in Section 4 and Appendix D. We additionally include pseudocode for our proposed method, PDE-Refiner, in Appendix C. All experiments on the KS datasets have been repeated for five seeds, and three seeds have been used for the Kolmogorov Flow dataset. Plots and tables with quantitative results show the standard deviation across these seeds.

As existing software assets, we base our implementation on the PDE-Arena [21], which implements a Python-based training framework for neural PDE solvers in PyTorch [60] and PyTorch Lightning [14]. For the diffusion models, we use the library diffusers [63]. We use Matplotlib [33] for plotting and NumPy [85] for data handling. For data generation, we use scipy [81] in the public code of Brandstetter et al. [8] for the KS equation, and JAX [6] in the public code of Kochkov et al. [42], Sun et al. [75] for the 2D Kolmogorov Flow dataset. The usage of these assets is further described in Appendix D. Since our code is proprietary, we include pseudocode in Appendix C, and will release the full code alongside the datasets in this paper upon publication.

In terms of computational resources, all experiments have been performed on NVIDIA V100 GPUs with 16GB memory. For the experiments on the KS equation, each model was trained on a single NVIDIA V100 for 1 to 2 days. For the 2D Kolmogorov Flow dataset, we parallelized the models across 4 GPUs, with a training time of 2 days. The speed comparison for the 2D Kolmogorov Flow were performed on an NVIDIA A100 GPU with 80GB memory. Overall, the experiments in this paper required roughly 250 GPU days, with additional 400 GPU days for development, hyperparameter search, and the supplementary results in Appendix E.

C PDE-Refiner - Pseudocode

In this section, we provide pseudocode to implement PDE-Refiner in Python with common deep learning frameworks like PyTorch [60] and JAX [6]. The hyperparameters to PDE-Refiner are the number of refinement steps K, called `num_steps` in the pseudocode, and the minimum noise standard deviation σ_{min}, called `min_noise_std`. Further, the neural operator `NO` can be an arbitrary network architecture, such as a U-Net as in our experiments, and is represented by `MyNetwork` / `self.neural_operator` in the code.

The dynamics of PDE-Refiner can be implemented via three short functions. The `train_step` function takes as input a training example of solution $u(t)$ (named `u_t`) and the previous solution $u(t - \Delta t)$ (named `u_prev`). We uniformly sample the refinement step we want to train, and use the classical MSE objective if $k = 0$. Otherwise, we train the model to denoise $u(t)$. The `loss` can be used to calculate gradients and update the parameters with common optimizers. The operation `randn_like` samples Gaussian noise of the same shape as `u_t`. Further, for batch-wise inputs, we sample k for each batch element independently. For inference, we implement the function `predict_next_solution`, which iterates through the refinement process of PDE-Refiner. Lastly, to generate a trajectory from an initial condition `u_initial`, the function `rollout` autoregressively predicts the next solutions. This gives us the following pseudocode:

```python
class PDERefiner:
    def __init__(self, num_steps, min_noise_std):
        self.num_steps = num_steps
        self.min_noise_std = min_noise_std
        self.neural_operator = MyNetwork(...)

    def train_step(self, u_t, u_prev):
        k = randint(0, self.num_steps + 1)
        if k == 0:
            pred = self.neural_operator(zeros_like(u_t), u_prev, k)
            target = u_t
        else:
            noise_std = self.min_noise_std ** (k / self.num_steps)
            noise = randn_like(u_t)
            u_t_noised = u_t + noise * noise_std
            pred = self.neural_operator(u_t_noised, u_prev, k)
            target = noise
        loss = mse(pred, target)
        return loss

    def predict_next_solution(self, u_prev):
        u_hat_t = self.neural_operator(zeros_like(u_prev), u_prev, 0)
        for k in range(1, self.num_steps + 1):
            noise_std = self.min_noise_std ** (k / self.num_steps)
            noise = randn_like(u_t)
            u_hat_t_noised = u_hat_t + noise * noise_std
            pred = self.neural_operator(u_hat_t_noised, u_prev, k)
            u_hat_t = u_hat_t_noised - pred * noise_std
        return u_hat_t

    def rollout(self, u_initial, timesteps):
        trajectory = [u_initial]
        for t in range(timesteps):
            u_hat_t = self.predict_next_solution(trajectory[-1])
            trajectory.append(u_hat_t)
        return trajectory
```

As discussed in Section 3.1, PDE-Refiner can be alternatively implemented as a diffusion model. To demonstrate this implementation, we use the Python library diffusers [63] (version 0.15) in the pseudocode below. We create a DDPM scheduler where we set the number of diffusion steps to the number of refinement steps and the prediction type to `v_prediction` [67]. Further, for simplicity, we set the betas to the noise variances of PDE-Refiner. We note that in diffusion models and in diffusers, the noise variance σ_k^2 at diffusion step k is calculated as:

$$\sigma_k^2 = 1 - \bar{\alpha}_k = 1 - \prod_{\kappa=k}^{K}(1-\beta_\kappa) = 1 - \prod_{\kappa=k}^{K}(1-\sigma_{\text{min}}^{2\kappa/K})$$

Since we generally use few diffusion steps such that the noise variance falls quickly, i.e. $\sigma_{\text{min}}^{2k/K} \gg \sigma_{\text{min}}^{2(k+1)/K}$, the product in above's equation is dominated by the last term $1-\sigma_{\text{min}}^{2k/K}$. Thus, the noise variances in diffusion are $\sigma_k^2 \approx \sigma_{\text{min}}^{2k/K}$. Further, for $k=0$ and $k=K$, the two variances are always the same since the product is 0 or a single element, respectively. If needed, one could correct for the product terms in the intermediate variances. However, as we show in Appendix E.6, PDE-Refiner is robust to small changes in the noise variance and no performance difference was notable. With this in mind, PDE-Refiner can be implemented as follows:

```python
from diffusers.schedulers import DDPMScheduler

class PDERefinerDiffusion:
    def __init__(self, num_steps, min_noise_std):
        betas = [min_noise_std ** (k / num_steps)
                 for k in reversed(range(num_steps + 1))]
        self.scheduler = DDPMScheduler(num_train_timesteps=num_steps,
                                       trained_betas=betas,
                                       prediction_type='v_prediction')
        self.num_steps = num_steps
        self.neural_operator = MyNetwork(...)

    def train_step(self, u_t, u_prev):
        k = randint(0, self.num_steps + 1)
        noise_factor = self.scheduler.alphas_cumprod[k]
        signal_factor = 1 - noise_factor
        noise = randn_like(u_t)
        u_t_noised = self.scheduler.add_noise(u_t, noise, k)
        pred = self.neural_operator(u_t_noised, u_prev, k)
        target = (noise_factor ** 0.5) * noise - (signal_factor ** 0.5) * u_t
        loss = mse(pred, target)
        return loss

    def predict_next_solution(self, u_prev):
        u_hat_t_noised = randn_like(u_prev)
        for k in range(self.num_steps + 1):
            pred = self.neural_operator(u_hat_t_noised, u_prev, k)
            u_hat_t_noised = self.scheduler.step(pred, k, u_hat_t_noised)
        u_hat_t = u_hat_t_noised
        return u_hat_t

    def rollout(self, u_initial, timesteps):
        trajectory = [u_initial]
        for t in range(timesteps):
            u_hat_t = self.predict_next_solution(trajectory[-1])
            trajectory.append(u_hat_t)
        return trajectory
```

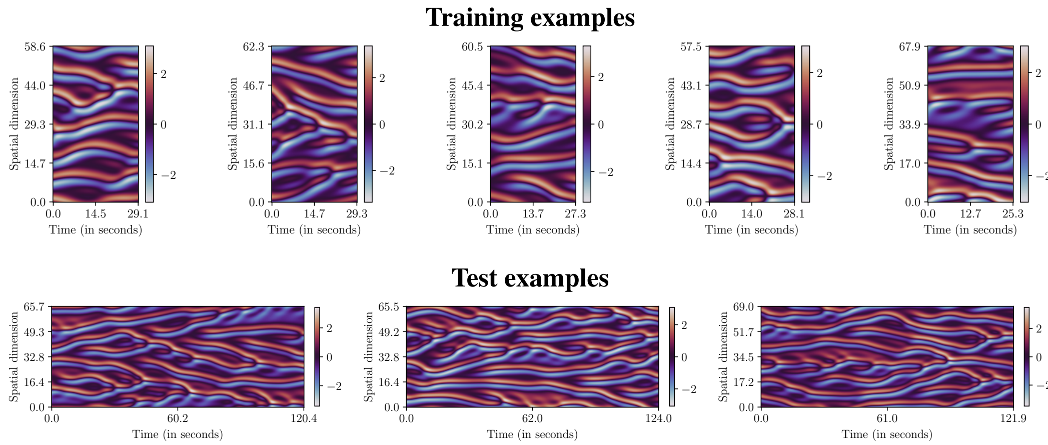


Figure 7: Dataset examples of the Kuramoto-Sivashinsky dataset. The training trajectories are generated with 140 time steps, while the test trajectories consist of 640 time steps. The spatial dimension is uniformly sampled from $[0.9 \cdot 64, 1.1 \cdot 64]$, and the time step in seconds from $[0.18, 0.22]$.

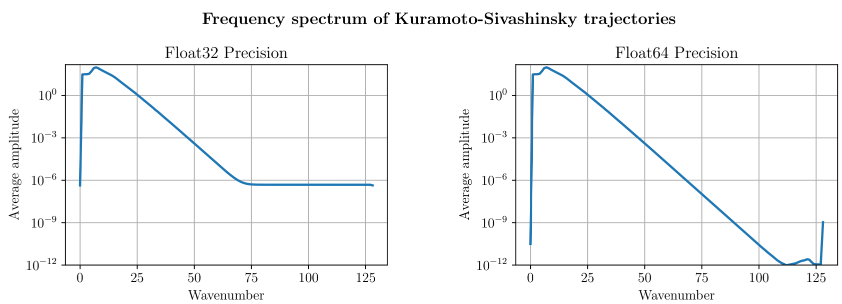


Figure 8: Frequency spectrum of the Kuramoto-Sivashinsky dataset under different precisions. Casting the input data to `float32` precision removes the high frequency information due to adding noise with higher amplitude. Neural surrogates trained on `float64` did not improve over `float32`, showing that it does not affect models in practice.

D Experimental details

In this section, we provide a detailed description of the data generation, model architecture, and hyperparameters used in our three datasets: Kuramoto-Sivashinsky (KS) equation, parameter-dependent KS equation, and the 2D Kolmogorov flow. Additionally, we provide an overview of all results with corresponding error bars in numerical table form. Lastly, we show example trajectories for each dataset.

D.1 Kuramoto-Sivashinsky 1D dataset

Data generation. We follow the data generation setup of Brandstetter et al. [8], which uses the method of lines with the spatial derivatives computed using the pseudo-spectral method. For each trajectory in our dataset, the first 360 solution steps are truncated and considered as a warmup for the solver. For further details on the data generation setup, we refer to Brandstetter et al. [8].

Our dataset can be reproduced with the public code[3] of Brandstetter et al. [8]. To obtain the training data, the data generation command in the repository needs to be adjusted by setting the number of training samples to 2048, and 0 for both validation and testing. For validation and testing, we increase the rollout time by adding the arguments `--nt=1000 --nt_effective=640 --end_time=200`, and setting the number of samples to 128 each. We provide training and test examples in Figure 7.

[3] `https://github.com/brandstetter-johannes/LPSDA#produce-datasets-for-kuramoto-shivashinsky-ks-equation`

Table 2: Detailed list of layers in the deployed modern U-Net. The parameter *channels* next to a layer represents the number of feature channels of the layer's output. The U-Net uses the four different channel sizes c_1, c_2, c_3, c_4, which are hyperparameters. The skip connection from earlier layers in a residual block is implemented by concatenating the features before the first GroupNorm. For the specifics of the residual blocks, see Figure 9.

Index	Layer
	Encoder
1	Conv(kernel size=3, channels=c_1, stride=1)
2	ResidualBlock(channels=c_1)
3	ResidualBlock(channels=c_1)
4	Conv(kernel size=3, channels=c_1, stride=2)
5	ResidualBlock(channels=c_2)
6	ResidualBlock(channels=c_2)
7	Conv(kernel size=3, channels=c_2, stride=2)
8	ResidualBlock(channels=c_3)
9	ResidualBlock(channels=c_3)
10	Conv(kernel size=3, channels=c_3, stride=2)
11	ResidualBlock(channels=c_4)
12	ResidualBlock(channels=c_4)
	Middle block
13	ResidualBlock(channels=c_4)
14	ResidualBlock(channels=c_4)
	Decoder
15	ResidualBlock(channels=c_4, skip connection from Layer 12)
16	ResidualBlock(channels=c_4, skip connection from Layer 11)
17	ResidualBlock(channels=c_3, skip connection from Layer 10)
18	TransposeConvolution(kernel size=4, channels=c_3, stride=2)
19	ResidualBlock(channels=c_3, skip connection from Layer 9)
20	ResidualBlock(channels=c_3, skip connection from Layer 8)
21	ResidualBlock(channels=c_2, skip connection from Layer 7)
22	TransposeConvolution(kernel size=4, channels=c_3, stride=2)
19	ResidualBlock(channels=c_2, skip connection from Layer 6)
20	ResidualBlock(channels=c_2, skip connection from Layer 5)
21	ResidualBlock(channels=c_1, skip connection from Layer 4)
22	TransposeConvolution(kernel size=4, channels=c_3, stride=2)
23	ResidualBlock(channels=c_1, skip connection from Layer 3)
24	ResidualBlock(channels=c_1, skip connection from Layer 2)
25	ResidualBlock(channels=c_1, skip connection from Layer 1)
26	GroupNorm(channels=c_1, groups=8)
27	GELU activation
28	Convolution(kernel size=3, channels=1, stride=1)

The data is generated with `float64` precision, and afterward converted to `float32` precision for storing and training of the neural surrogates. Since we convert the precision in spatial domain, it causes minor artifacts in the frequency spectrum as seen in Figure 8. Specifically, frequencies with wavenumber higher than 60 cannot be adequately represented. Quantizing the solution values in spatial domain introduce high-frequency noise which is greater than the original amplitudes. Training the neural surrogates with `float64` precision did not show any performance improvement, besides being significantly more computationally expensive.

Model architecture. For all models in Section 4.1, we use the modern U-Net architecture from Gupta et al. [21], which we detail in Table 2. The U-Net consists of an encoder and decoder, which are implemented via several pre-activation ResNet blocks [24, 25] with skip connections between encoder and decoder blocks. The ResNet block is visualized in Figure 9 and consists of Group Normalization [86], GELU activations [26], and convolutions with kernel size 3. The conditioning parameters Δt and Δx are embedded into feature vector space via sinusoidal embeddings, as for

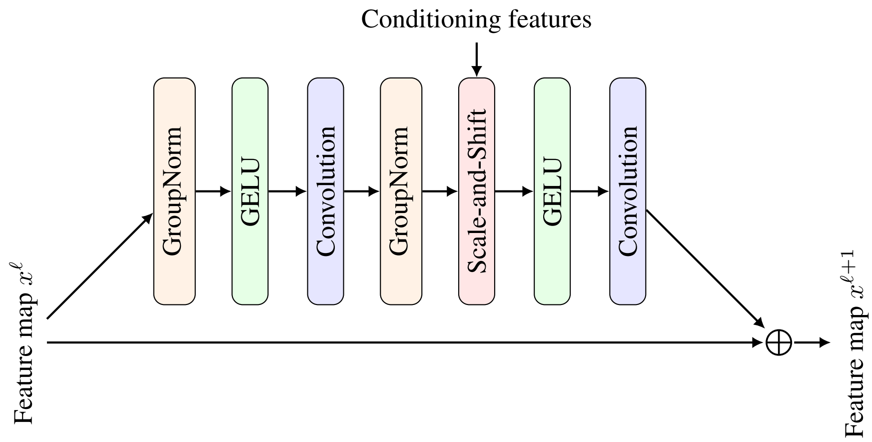


Figure 9: ResNet block of the modern U-Net [21]. Each block consists of two convolutions with GroupNorm and GELU activations. The conditioning features, which are Δt, Δx for the KS dataset and additionally ν for the parameter-dependent KS dataset, influence the features via a scale-and-shift layer. Residual blocks with different input and output channels use a convolution with kernel size 1 on the residual connection.

Table 3: Hyperparameter overview for the experiments on the KS equation. Hyerparameters have been optimized for the baseline MSE-trained model on the validation dataset, which generally worked well across all models.

Hyperparameter	**Value**
Input Resolution	256
Number of Epochs	400
Batch size	128
Optimizer	AdamW [50]
Learning rate	CosineScheduler(1e-4 $\rightarrow$ 1e-6)
Weight Decay	1e-5
Time step	0.8s / 4Δt
Output factor	0.3
Network	Modern U-Net [21]
Hidden size	$c_1 = 64$, $c_2 = 128$, $c_3 = 256$, $c_4 = 1024$
Padding	circular
EMA Decay	0.995

example used in Transformers [80]. We combine the feature vectors via linear layers and integrate them in the U-Net via AdaGN [59, 62] layers, which predicts a scale and shift parameter for each channel applied after the second Group Normalization in each residual block. We represent it as a 'scale-and-shift' layer in Figure 9. We also experimented with adding attention layers in the residual blocks, which, however, did not improve performance noticeably. The implementation of the U-Net architecture can be found in the public code of Gupta et al. [21].[4]

Hyperparameters. We detail the used hyperparameters for all models in Table 3. We train the models for 400 epochs on a batch size of 128 with an AdamW optimizer [50]. One epoch corresponds to iterating through all training sequences and picking 100 random initial conditions each. The learning rate is initialized with 1e-4, and follows a cosine annealing strategy to end with a final learning rate of 1e-6. We did not find learning rate warmup to be needed for our models. For regularization, we use a weight decay of 1e-5. As mentioned in Section 4.1, we train the neural operators to predict 4 time steps ahead via predicting the residual $\Delta u = u(t) - u(t - 4\Delta t)$. For better output coverage of the neural network, we normalize the residual to a standard deviation of about 1 by dividing it with 0.3.

[4] `https://github.com/microsoft/pdearena/blob/main/pdearena/modules/conditioned/twod_unet.py`

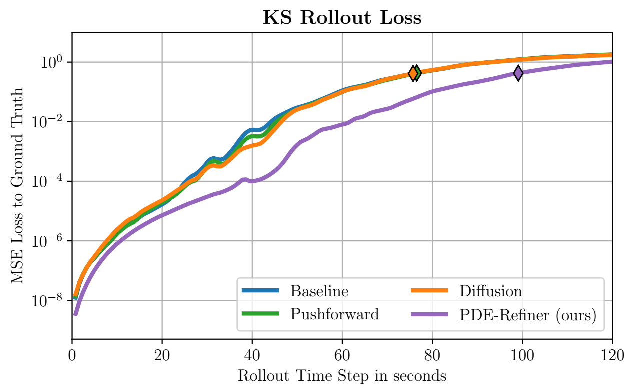


Figure 10: Visualizing the average MSE error over rollouts on the test set for four methods: the baseline MSE-trained model (blue), the pushforward trick (green), the diffusion model with standard cosine scheduling (orange), and PDE-Refiner with 8 refinement steps. The markers indicate the time when the method's average rollout correlation falls below 0.8. The y-axis shows the logarithmic scale of the MSE error. While all models have a similar loss for the first 20 seconds, PDE-Refiner has a much smaller increase of loss afterwards.

Thus, the neural operators predict the next time step via $\hat{u}(t) = u(t - 4\Delta t) + 0.3 \cdot \texttt{NO}(u(t - 4\Delta t))$. We provide an ablation study on the step size in Appendix E.2. For the modern U-Net, we set the hidden sizes to 64, 128, 256, and 1024 on the different levels, following Gupta et al. [21]. This gives the model a parameter count of about 55 million. Crucially, all convolutions use circular padding in the U-Net to account for the periodic domain. Finally, we found that using an exponential moving average (EMA) [40] of the model parameters during validation and testing, as commonly used in diffusion models [29, 37] and generative adversarial networks [15, 87], improves performance and stabilizes the validation performance progress over training iterations across all models. We set the decay rate of the moving average to 0.995, although it did not appear to be a sensitive hyperparameter.

Next, we discuss extra hyperparameters for each method in Figure 3 individually. The history 2 model includes earlier time steps by concatenating $u(t - 8\Delta t)$ with $u(t - 4\Delta t)$ over the channel dimension. We implement the model with $4\times$ parameters by multiplying the hidden size by 2, i.e. use 128, 256, 512, and 2048. This increases the weight matrices by a factor of 4. For the pushforward trick, we follow the public implementation of Brandstetter et al. [9][5] and increase the probability of replacing the ground truth with a prediction over the first 10 epochs. Additionally, we found it beneficial to use the EMA model weights for creating the predictions, and rolled out the model up to 3 steps. We implemented the Markov Neural Operator following the public code[6] of Li et al. [49]. We performed a hyperparameter search over $\lambda \in \{0.2, 0.5, 0.8\}, \alpha \in \{0.001, 0.01, 0.1\}, k \in \{0, 1\}$, for which we found $\lambda = 0.5, \alpha = 0.01, k = 0$ to work best. The error correction during rollout is implemented by performing an FFT on each prediction, setting the amplitude and phase for wavenumber 0 and above 60 to zero, and mapping back to spatial domain via an inverse FFT. For the error prediction, in which one neural operator tries to predict the error of the second operator, we scale the error back to an average standard deviation of 1 to allow for a better output scale of the second U-Net. The DDPM Diffusion model is implemented using the diffusers library [63]. We use a DDPM scheduler with `squaredcos_cap_v2` scheduling, a beta range of 1e-4 to 1e-1, and 1000 train time steps. During inference, we set the number of sampling steps to 16 (equally spaced between 0 and 1000) which we found to obtain best results while being more efficient than 1000 steps. For our schedule, we set the betas the same way as shown in the pseudocode of Appendix C. Lastly, we implement PDE-Refiner using the diffusers library [63] as shown in Appendix C. We choose the minimum noise variance $\sigma^2_{\text{min}} = $ 2e-7 based on a hyperparameter search on the validation, and provide an ablation study on it in Appendix E.6.

Results. We provide an overview of the results in Figure 3 as table in Table 4. Besides the high-correction time with thresholds 0.8 and 0.9, we also report the one-step MSE error between the

[5] `https://github.com/brandstetter-johannes/MP-Neural-PDE-Solvers/`

[6] `https://github.com/neuraloperator/markov_neural_operator/`

Table 4: Results of Figure 3 in table form. All standard deviations are reported over 5 seeds excluding *Ensemble*, which used all 5 baseline model seeds and has thus no standard deviation. Further, we include the average one-step MSE error of each method on the test set. Notably, lower one-step MSE does not necessarily imply longer stable rollouts (e.g. History 2 versus baseline).

Method	**Corr.** > 0.8 **time**	**Corr.** > 0.9 **time**	**One-step MSE**
	MSE Training		
Baseline	75.4 $\pm$ 1.1	66.5 $\pm$ 0.8	2.70e-08 $\pm$ 8.52e-09
History 2	61.7 $\pm$ 1.1	54.3 $\pm$ 1.8	1.50e-08 $\pm$ 1.67e-09
4$\times$ parameters	79.7 $\pm$ 0.7	71.7 $\pm$ 0.7	1.02e-08 $\pm$ 4.91e-10
Ensemble	79.7 $\pm$ 0.0	72.5 $\pm$ 0.0	5.56e-09 $\pm$ 0.00e+00
	Alternative Losses		
Pushforward [9]	75.4 $\pm$ 1.1	67.3 $\pm$ 1.7	2.76e-08 $\pm$ 5.68e-09
Sobolev norm $k = 0$ [49]	71.4 $\pm$ 2.9	62.2 $\pm$ 3.9	1.33e-07 $\pm$ 8.70e-08
Sobolev norm $k = 1$ [49]	66.9 $\pm$ 1.8	59.3 $\pm$ 1.5	1.04e-07 $\pm$ 3.28e-08
Sobolev norm $k = 2$ [49]	8.7 $\pm$ 0.9	7.3 $\pm$ 0.5	7.84e-04 $\pm$ 9.30e-05
Markov Neural Operator [49]	66.6 $\pm$ 1.0	58.5 $\pm$ 2.1	2.66e-07 $\pm$ 1.08e-07
Error correction [56]	74.8 $\pm$ 1.1	66.2 $\pm$ 0.9	1.46e-08 $\pm$ 1.99e-09
Error Prediction	75.7 $\pm$ 0.5	67.3 $\pm$ 0.6	2.96e-08 $\pm$ 2.36e-10
	Diffusion Ablations		
Diffusion - Standard Scheduler [29]	75.2 $\pm$ 1.0	66.9 $\pm$ 0.7	3.06e-08 $\pm$ 5.24e-10
Diffusion - Our Scheduler	88.9 $\pm$ 1.0	79.7 $\pm$ 1.1	2.85e-09 $\pm$ 1.65e-10
	PDE-Refiner		
PDE-Refiner - 1 step (ours)	89.8 $\pm$ 0.4	80.6 $\pm$ 0.2	3.14e-09 $\pm$ 2.85e-10
PDE-Refiner - 2 steps (ours)	94.2 $\pm$ 0.8	84.2 $\pm$ 0.4	5.24e-09 $\pm$ 1.54e-10
PDE-Refiner - 3 steps (ours)	97.5 $\pm$ 0.5	87.0 $\pm$ 0.9	5.80e-09 $\pm$ 1.65e-09
PDE-Refiner - 4 steps (ours)	98.3 $\pm$ 0.8	87.8 $\pm$ 1.6	5.95e-09 $\pm$ 1.95e-09
PDE-Refiner - 8 steps (ours)	98.3 $\pm$ 0.1	89.0 $\pm$ 0.4	6.16e-09 $\pm$ 1.48e-09
PDE-Refiner - 3 steps mean (ours)	98.5 $\pm$ 0.8	88.6 $\pm$ 1.1	1.28e-09 $\pm$ 6.27e-11

prediction $\hat{u}(t)$ and the ground truth solution $u(t)$. A general observation is that the one-step MSE is not a strong indication of the rollout performance. For example, the MSE loss of the history 2 model is twice as low as the baseline's loss, but performs significantly worse in rollout. Similarly, the Ensemble has a lower one-step error than PDE-Refiner with more than 3 refinement steps, but is almost 20 seconds behind in rollout.

As an additional metric, we visualize in Figure 10 the mean-squared error loss between predictions and ground truth during rollout. In other words, we replace the correlation we usually measure during rollout with the MSE. While PDE-Refiner starts out with similar losses as the baselines for the first 20 seconds, it has a significantly smaller increase in loss afterward. This matches our frequency analysis, where only in later time steps, the non-dominant, high frequencies start to impact the main dynamics. Since PDE-Refiner can model these frequencies in contrast to the baselines, it maintains a smaller error accumulation.

Speed comparison. We provide a speed comparison of an MSE-trained baseline with PDE-Refiner on the KS equation. We time the models on generating the test trajectories (batch size 128, rollout length $640\Delta t$) on an NVIDIA A100 GPU with a 24 core AMD EPYC CPU. We compile the models in PyTorch 2.0 [60], and exclude compilation and data loading time from the runtime. The MSE model requires 2.04 seconds (± 0.01), while PDE-Refiner with 3 refinement steps takes 8.67 seconds (± 0.01). In contrast, the classical solver used for data generation requires on average 47.21 seconds per trajectory, showing the significant speed-up of the neural surrogates. However, it should be noted that the solver is implemented on CPU and there may exist faster solvers for the 1D Kuramoto-Sivashinsky equation.

D.2 Parameter-dependent KS dataset

Data generation. We follow the same data generation as in Appendix D.1. To integrate the viscosity ν, we multiply the fourth derivative estimate u_{xxxx} by ν. For each training and test trajectory, we uniformly sample ν between 0.5 and 1.5. We show the effect of different viscosity terms in Figure 11.

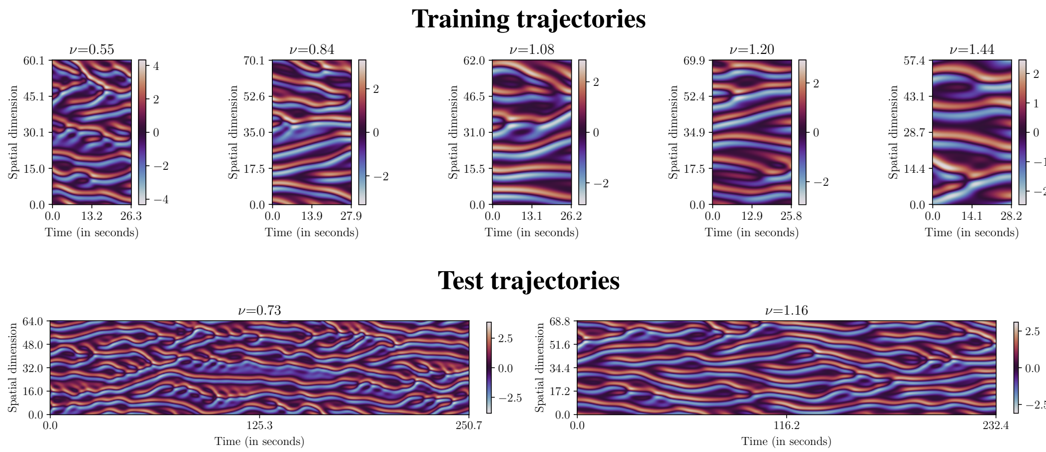


Figure 11: Dataset examples of the parameter-dependent Kuramoto-Sivashinsky dataset. The viscosity is noted above each trajectory. The training trajectories are 140 time steps, while the test trajectories are rolled out for 1140 time steps. Lower viscosities generally create more complex, difficult trajectories.

Table 5: Results of Figure 6 in table form. All standard deviations are reported over 5 seeds.

Method	**Viscosity**	**Corr.** > 0.8 **time**	**Corr.** > 0.9 **time**
MSE Training	$[0.5, 0.7)$	41.8 ± 0.4	35.6 ± 0.6
	$[0.7, 0.9)$	57.7 ± 0.6	50.7 ± 1.3
	$[0.9, 1.1)$	73.3 ± 2.3	66.0 ± 2.5
	$[1.1, 1.3)$	88.0 ± 1.5	76.7 ± 2.2
	$[1.3, 1.5]$	97.0 ± 2.7	85.5 ± 2.2
PDE-Refiner	$[0.5, 0.7)$	53.1 ± 0.4	46.7 ± 0.4
	$[0.7, 0.9)$	71.4 ± 0.3	64.3 ± 0.6
	$[0.9, 1.1)$	94.5 ± 0.6	84.9 ± 0.6
	$[1.1, 1.3)$	112.2 ± 0.9	98.5 ± 1.5
	$[1.3, 1.5]$	130.2 ± 1.5	116.6 ± 0.7

Model architecture. We use the same modern U-Net as in Appendix D.1. The conditioning features consist of Δt, Δx, and ν. For better representation in the sinusoidal embedding, we scale ν to the range $[0, 100]$ before embedding it.

Hyperparameters. We reuse the same hyperparameters of Appendix D.1 except reducing the number of epochs to 250. This is since the training dataset is twice as large as the original KS dataset, and the models converge after fewer epochs.

Results. We provide the results of Figure 6 in table form in Table 5. Overall, PDE-Refiner outperforms the MSE-trained baseline by 25-35% across viscosities.

D.3 Kolmogorov 2D Flow

Data generation. We followed the data generation of Sun et al. [75] as detailed in the publicly released code[7]. For hyperparameter tuning, we additionally generate a validation set of the same size as the test data with initial seed 123. Afterward, we remove trajectories where the ground truth solver had NaN outputs, and split the trajectories into sub-sequences of 50 frames for efficient training. An epoch consists of iterating over all sub-sequences and sampling 5 random initial conditions from each. All data are stored in `float32` precision.

Model architecture. We again use the modern U-Net [21] for PDE-Refiner and an MSE-trained

[7] https://github.com/Edward-Sun/TSM-PDE/blob/main/data_generation.md

Table 6: Hyperparameter overview for the experiments on the Kolmogorov 2D flow.

Hyperparameter	**Value**
Input Resolution	64×64
Number of Epochs	100
Batch size	32
Optimizer	AdamW [50]
Learning rate	CosineScheduler(1e-4 $\rightarrow$ 1e-6)
Weight Decay	1e-5
Time step	0.112s / $16\Delta t$
Output factor	0.16
Network	Modern U-Net [21]
Hidden size	[128, 128, 256, 1024]
Padding	circular
EMA Decay	0.995

baseline, where, in comparison to the model for the KS equation, we replace 1D convolutions with 2D convolutions. Due to the low input resolution, we experienced that the model lacked complexity on the highest feature resolution. Thus, we increased the initial hidden size to 128, and use 4 ResNet blocks instead of 2 on this level. All other levels remain the same as for the KS equation. This model has 157 million parameters.

The Fourier Neural Operator [48] consists of 8 layers, where each layer consists of a spectral convolution with a skip connection of a 1×1 convolution and GELU activation [26]. We performed a hyperparameter search over the number of modes and hidden size, for which we found 32 modes with hidden size 64 to perform best. This models has 134 million parameters, roughly matching the parameter count of a U-Net. Models with larger parameter count, e.g. hidden size 128 with 32 modes, did not show any improvements.

Hyperparameters. We summarize the chosen hyperparameters in Table 6, which were selected based on the performance on the validation dataset. We train the models for 100 epochs with a batch size of 32. Due to the increased memory usage, we parallelize the model over 4 GPUs with batch size 8 each. We predict every 16th time step, which showed similar performance to models with a time step of 1, 2, 4, and 8 while being faster to roll out. All models use as objective the residual $\Delta u = u(t) - u(t - 16\Delta t)$, which we normalize by dividing with its training standard deviation of 0.16. Thus, we predict the next solution via $\hat{u}(t) = u(t - 16\Delta t) + 0.16 \cdot \texttt{NO}(...)$. Each model is trained for 3 seeds, and the standard deviation is reported in Table 1.

Results. We include example trajectories and corresponding predictions by PDE-Refiner in Figure 12. PDE-Refiner is able to maintain accurate predictions for more than 11 seconds for many trajectories.

Speed comparison. All models are run on the same hardware, namely an NVIDIA A100 GPU with 80GB memory and an 24 core AMD EPYC CPU. For the hybrid solvers, we use the public implementation in JAX [6] by Kochkov et al. [42], Sun et al. [75]. For the U-Nets, we use PyTorch 2.0 [14]. All models are compiled in their respective frameworks, and we exclude the compilation and time to load the data from the runtime. We measure the speed of each model 5 times, and report the mean and standard deviation in Section 4.3.

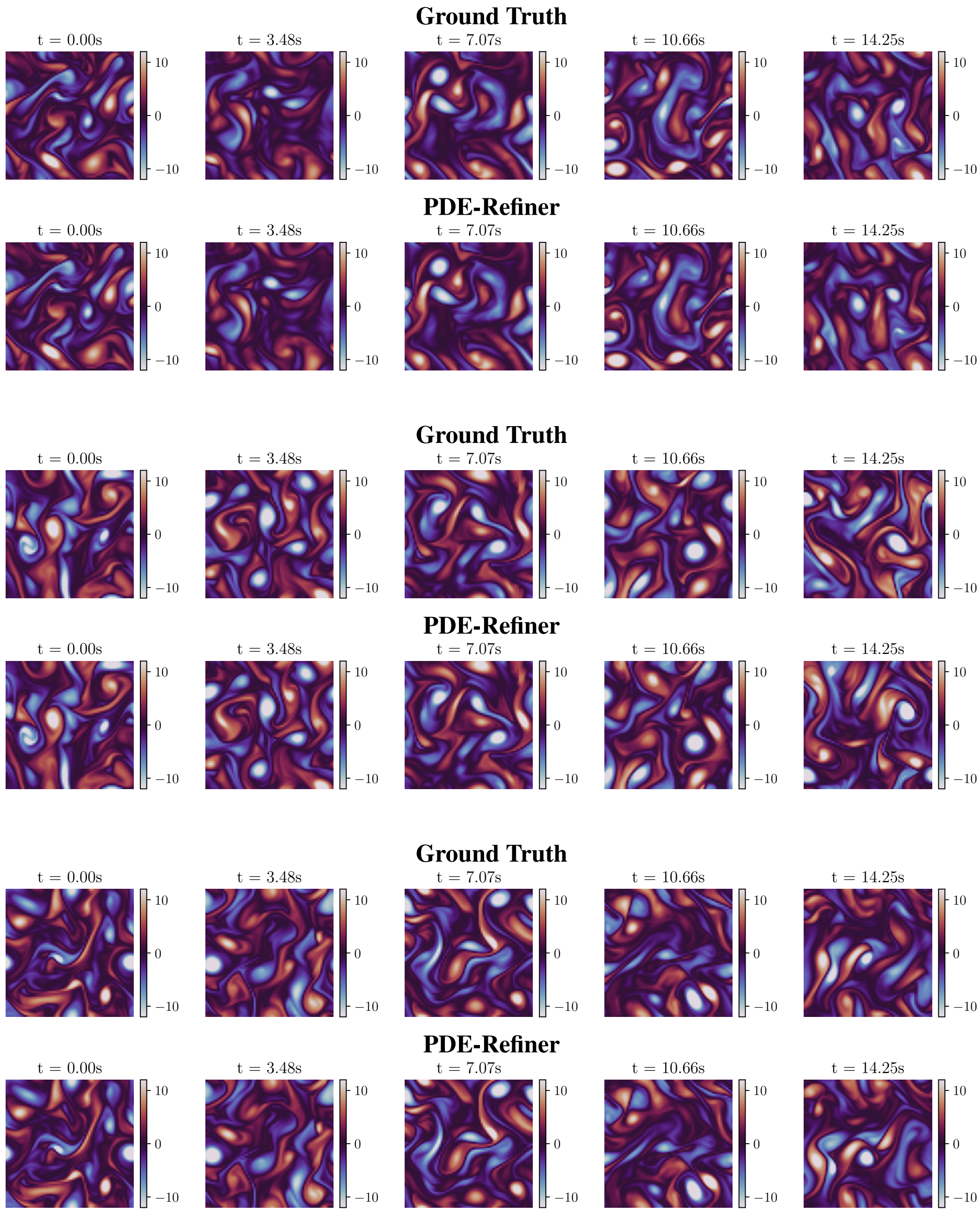


Figure 12: Visualizing the vorticity of three example test trajectories of the 2D Kolmogorov flow, with corresponding predictions of PDE-Refiner. PDE-Refiner remains stable for more than 10 seconds, making on minor errors at 10.66 seconds. Moreover, many structures at 14 seconds are still similar to the ground truth.

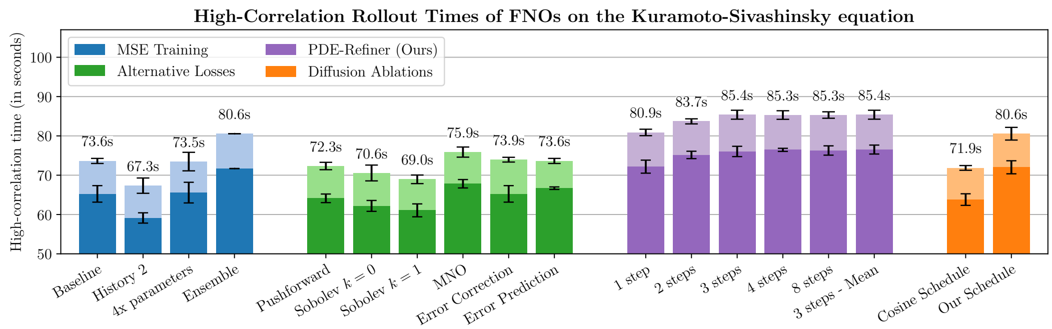


Figure 13: Experimental results of Fourier Neural Operators on the Kuramoto-Sivashinsky equation. All methods from Figure 3 are included here. FNOs achieve similar results as the U-Nets for the baselines. For PDE-Refiner and Diffusion, FNOs still outperforms the baselines, but with a smaller gain than the U-Nets due to the noise objective.

E Supplementary Experimental Results

In this section, we provide additional experimental results on the Kuramoto-Sivashinsky equation and the 2D Kolmogorov flow. Specifically, we experiment with Fourier Neural Operators as an alternative to our deployed U-Nets. We provide ablation studies on the predicted step size, the history information, and the minimum noise variance in PDE-Refiner on the KS equation. For the Kolmogorov flow, we provide the same frequency analysis as done for the KS equation in the main paper. Finally, we investigate the stability of the neural surrogates for very long rollouts of 800 seconds.

E.1 Fourier Neural Operator

Fourier Neural Operators (FNOs) [48] are a popular alternative to U-Nets for neural operator architectures. To show that the general trend of our results in Section 4.1 are architecture-invariant, we repeat all experiments of Figure 3 with FNOs. The FNO consists of 8 layers, where each layer consists of a spectral convolution with a skip connection of a 1×1 convolution and a GELU activation [26]. Each spectral convolution uses the first 32 modes, and we provide closer discussion on the impact of modes in Figure 14. We use a hidden size of 256, which leads to the model having about 40 million parameters, roughly matching the parameter count of the used U-Nets.

MSE Training. We show the results for all methods in Figure 13. The MSE-trained FNO baseline achieves with 73.6s a similar rollout time as the U-Net (75.4s). Again, using more history information decreases rollout performance. Giving the model more complexity by increasing the parameter count to 160 million did not show any improvement. Still, the ensemble of 5 MSE-trained models obtains a 7-second gain over the individual models, slightly outperforming the U-Nets for this case.

Alternative losses. The pushforward trick, the error correction and the error predictions again cannot improve over the baseline. While using the Sobolev norm losses decrease performance also for FNOs, using the regularizers of the Markov Neural Operator is able to provide small gains. This is in line with the experiments of Li et al. [49], in which the MNO was originally proposed for Fourier Neural Operators. Still, the gain is limited to 3%.

PDE-Refiner. With FNOs, PDE-Refiner again outperforms all baselines when using more than 1 refinement step. The gains again flatten for more than 3 steps. However, in comparisons to the U-Nets with up to 98.5s accurate rollout time, the performance increase is significantly smaller. In general, we find that FNOs obtain higher training losses for smaller noise values than U-Nets, indicating the modeling of high-frequent noise in PDE-Refiner's refinement objective to be the main issue. U-Nets are more flexible in that regard, since they use spatial convolutions. Still, the results show that PDE-Refiner is applicable to a multitude of neural operator architectures.

Diffusion ablations. Confirming the issue of the noise objective for FNOs, the diffusion models with standard cosine scheduling obtain slightly worse results than the baseline. Using our exponential noise scheduler again improves performance to the level of the one-step PDE-Refiner.

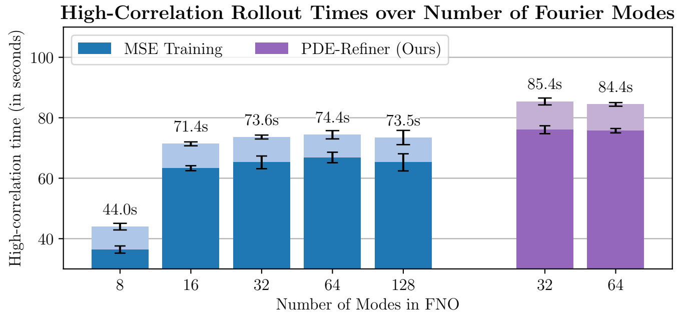


Figure 14: Investigating the impact of the choosing the number of modes in FNOs. Similar to our analysis on the resolution in the U-Nets (Figure 5), we only see minor improvements of using higher frequencies above 16 in the MSE training. Removing dominant frequencies above 8 significantly decreases performance. Similarly, increasing the modes of FNOs in PDE-Refiner has minor impact.

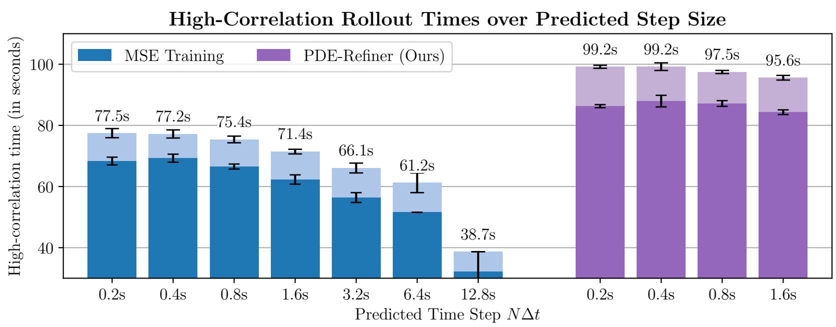


Figure 15: Comparing the accurate rollout times over the step size at which the neural operator predicts. This is a multiple of the time step Δt used for data generation (for KS on average 0.2s). For both the MSE Training and PDE-Refiner, lower step size provides longer stable rollouts, where very large time steps show a significant loss in accuracy. This motivates the need for autoregressive neural PDE solvers over direct, long-horizon predictions.

Number of Fourier Modes. A hyperparameter in Fourier Neural Operators is the number of Fourier modes that are considered in the spectral convolutions. Any higher frequency is ignored and must be modeled via the residual 1×1 convolutions. To investigate the impact of the number of Fourier modes, we repeat the baseline experiments of MSE-trained FNOs with 8, 16, 32, 64, and 128 modes in Figure 14. To ensure a fair comparison, we adjust the hidden size to maintain equal number of parameters across models. In general, we find that the high-correlation time is relatively stable for 32 to 128 modes. Using 16 modes slightly decreases performance, while limiting the layers to 8 modes results in significantly worse rollouts. This is also in line with our input resolution analysis of Figure 5, where the MSE-trained baseline does not improve for high resolutions. Similarly, we also apply a 64 mode FNOs for PDE-Refiner. Again, the performance does not increase for higher number of modes.

E.2 Step Size Comparison

A key advantage of Neural PDE solvers is their flexibility to be applied to various step sizes of the PDEs. The larger the step size is, the faster the solver will be. At the same time, larger step sizes may be harder to predict. To compare the effect of error propagation in an autoregressive solver with training a model to predict large time steps, we repeat the baseline experiments of the U-Net neural operator on the KS equation with different step sizes. The default step size that was used in Figure 5 is 4-times the original solver step, being on average 0.8s. For any step size below 2s, we model the residual objective $\Delta u = u(t) - u(t - \Delta t)$, which we found to generally work better in this range. For any step size above, we directly predict the solution $u(t)$.

High-correlation time. We plot the results step sizes between 0.2s and 12.8s in Figure 15. We find that the smaller the step size, the longer the model remains accurate. The performance also decreases

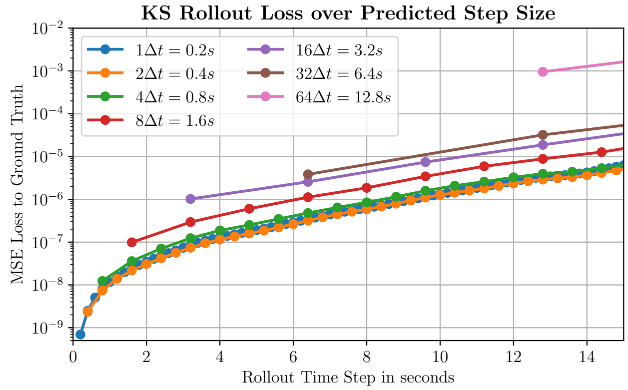


Figure 16: Visualizing the MSE error of MSE-trained models with varying step sizes over the rollout. The models with a step size of $1\Delta t$, $2\Delta t$, and $4\Delta t$ all obtain similar performance. For $8\Delta t$, the one-step MSE loss is already considerably higher than, e.g. rolling out the step size $1\Delta t$ model 8 times. For larger time steps, this gap increases further, again highlighting the strengths of autoregressive solvers.

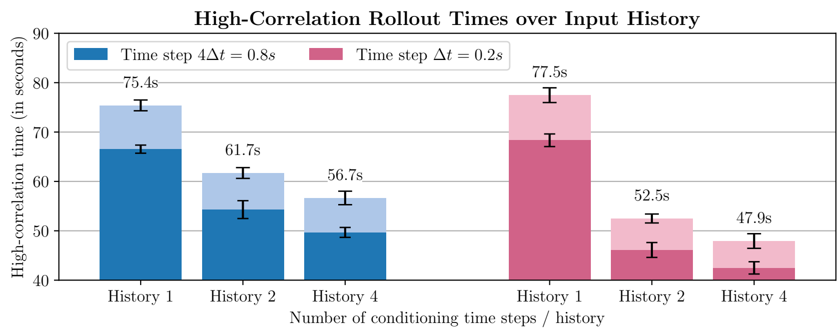


Figure 17: Investigating the impact of using more history / past time steps in the neural operators, i.e., $\hat{u}(t) = \texttt{NO}(u(t - \Delta t), u(t - 2\Delta t), ...)$, for $\Delta t = 0.8$ and $\Delta t = 0.2$. Longer histories decrease the model's accurate rollout time. This drop in performance is even more significant for smaller time steps.

faster for very large time steps. This is because the models start to overfit on the training data and have difficulties learning the actual dynamics of the PDE. Meanwhile, very small time steps do not suffer from autoregressive error propagation any more than slightly larger time steps, while generalizing well. This highlights again the strength of autoregressive neural PDE solvers. We confirm this trend by training PDE-Refiner with different step sizes while using 3 refinement steps. We again find that smaller time steps achieve higher performance, and we obtain worse rollout times for larger time steps.

MSE loss over rollout. To further gain insights of the impact of different step sizes, we plot in Figure 16 the MSE loss to the ground truth when rolling out the MSE-trained models over time. Models with larger time steps require fewer autoregressive steps to predict long-term into the future, preventing any autoregressive error accumulation for the first step. Intuitively, the error increases over time for all models, since the errors accumulate over time and cause the model to diverge. The models with step sizes 0.2s, 0.4s and 0.8s all achieve very similar losses across the whole time horizon. This motivates our choice for 0.8s as default time step, since it provides a 4 times speedup in comparison to the 0.2s model. Meanwhile, already a model trained with step size 1.6s performs considerable worse in its one-step prediction than a model with step size 0.2s rolled out 8 times. The gap increases further the larger the time step becomes. Therefore, directly predicting large time steps in neural PDE solvers is not practical and autoregressive solvers provide significant advantages.

E.3 History Information

In our experiments on the KS equation, we have observed that using more history information as input decreases the rollout performance. Specifically, we have used a neural operator that took as input the

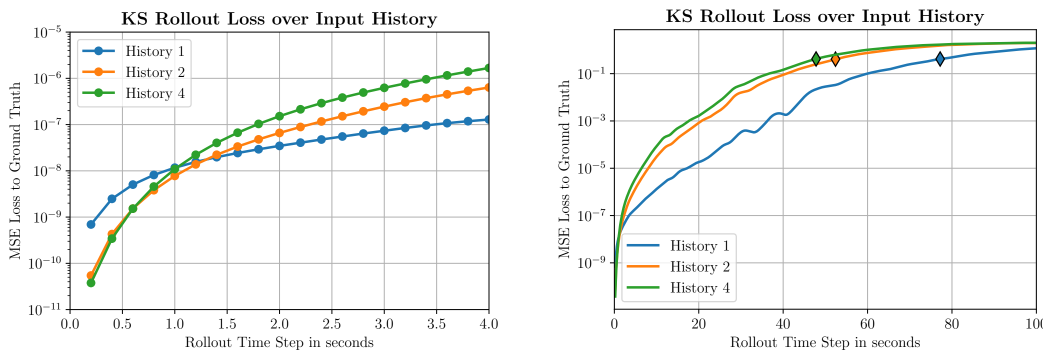


Figure 18: Comparing models conditioned on different number of past time steps on their MSE loss over rollouts. Note the log-scale on the y-axis. The markers indicate the time when the average correlation of the respective model drops below 0.8. The left plot shows a zoomed-in version of the first 4 seconds of the whole 100 second rollout on the right. While using more history information gives an advantage for the first ∼5 steps, the error propagates significantly faster through the models. This leads to a significantly higher loss over rollout.

Table 7: Comparing the uncertainty estimate of PDE-Refiner to Input Modulation [5, 68] and Model Ensemble [44, 68] on the MSE-trained models. The metrics show the correlation between the estimated and actual accurate rollout time in terms of the R^2 coefficient of determination and the Pearson correlation. PDE-Refiner provides more accurate uncertainty estimates than Input Modulation while being more efficient than an Model Ensemble.

Method	R^2 coefficient	Pearson correlation
PDE-Refiner	0.857 ± 0.027	0.934 ± 0.014
Input Modulation [5, 68]	0.820 ± 0.081	0.912 ± 0.021
Model Ensemble [44, 68]	0.887 ± 0.012	0.965 ± 0.007

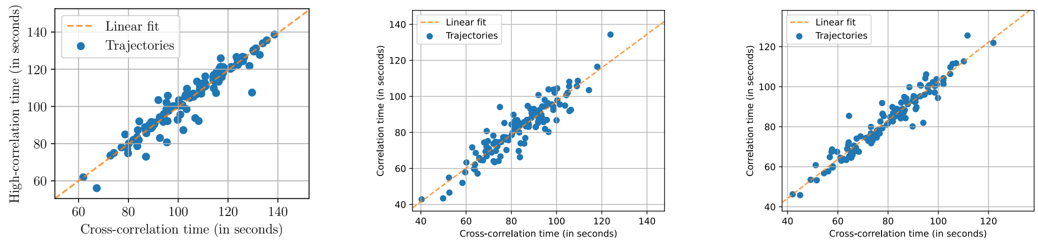
past two time steps, $u(t-\Delta t)$ and $u(t-2\Delta t)$. To confirm this trend, we repeat the experiments with a longer history of 4 past time steps and for models with a smaller step size of 0.2s in Figure 17. Again, we find that the more history information we use as input, the worse the rollouts become. Furthermore, the impact becomes larger for small time steps, indicating that the autoregressive error propagation becomes a larger issue when using history information. The problem arising is that the difference between the inputs $u(t-\Delta t) - u(t-2\Delta t)$ is highly correlated with the model's target $\Delta u(t)$, the residual of the next time step. The smaller the time step, the larger the correlation. This leads the neural operator to focus on modeling the second-order difference $\Delta u(t) - \Delta u(t-2\Delta t)$. As observed in classical solvers [35], using higher-order differences within an explicit autoregressive scheme is known to deteriorate the rollout stability and introduce exponentially increasing errors over time.

We also confirm this exponential increase of error by plotting the MSE error over rollouts in Figure 18. While the history information improves the one-step prediction by a factor of 10, the error of the history 2 and 4 models quickly surpasses the error of the history 1 model. After that, the error of the models continue to increase quickly, leading to an earlier divergence.

E.4 Uncertainty Estimation

We extend our discussion on the uncertainty estimation of Section 4.1 by comparing PDE-Refiner to two common baselines for uncertainty estimation of temporal forecasting: Input Modulation [5, 68] and Model Ensemble [44, 68]. Input Modulation adds small random Gaussian noise to the initial condition $u(0)$, and rolls out the model on several samples. Similar to PDE-Refiner, one can determine the uncertainty by measuring the cross-correlation between the rollouts. A Model Ensemble compares the predicted trajectories of several independently trained models. For the case here, we use 4 trained models. For both baselines, we estimate the uncertainty of MSE-trained models as usually applied.

We evaluate the R^2 coefficient of determination and the Pearson correlation between the estimated



(a) PDE-Refiner (Ours) (b) Input Modulation (c) Model Ensemble

Figure 19: Qualitative comparison between the uncertainty estimates of PDE-Refiner, Input Modulation, and the Model Ensemble. Both PDE-Refiner and the Model Ensemble achieve an accurate match between the estimated and ground truth rollout times.

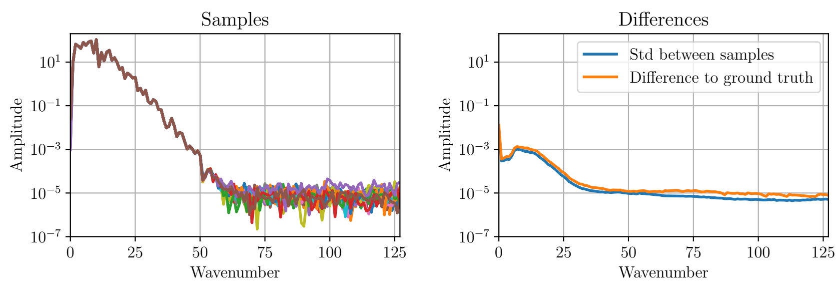


Figure 20: Investigating the spread of samples of PDE-Refiner. The left plot shows the frequency spectrum of 16 samples (each line represents a different sample), with the right plot showing the average difference to the ground truth and to the mean of the samples. The deviation of the samples closely matches the average error, showing that PDE-Refiner adapts its samples to the learned error over frequencies.

stable rollout times and the ground truth rollout times in Table 7. We additionally show qualitative results in Figure 19. PDE-Refiner's uncertainty estimate outperforms the Input Modulation approach, showing that Gaussian noise does not fully capture the uncertainty distribution. While performing slightly worse than using a full Model Ensemble, PDE-Refiner has the major advantage that it only needs to be trained, which is particularly relevant in large-scale experiments like weather modeling where training a model can be very costly.

To investigate the improvement of PDE-Refiner over Input Modulation, we plot the standard deviation over samples in PDE-Refiner in Figure 20. The samples of PDE-Refiner closely differs in the same distribution as the actual loss to the ground truth, showing that PDE-Refiner accurately models its predictive uncertainty.

E.5 Frequency Analysis for 2D Kolmogorov Flow

We repeat the frequency analysis that we have performed on the KS equation in the main paper, e.g. Figure 4, on the Kolmogorov dataset here. Note that we apply a 2D Discrete Fourier Transform and show the average frequency spectrum. We perform this over the two channels of $u(t)$ independently. Figure 21 shows the frequency spectrum for the ground truth data, as well as the predictions of PDE-Refiner and the MSE-trained U-Net. In contrast to the KS equation, the spectrum is much flatter, having an amplitude of still almost 1 at wavenumber 32. In comparison, the KS equation has a more than 10 times as small amplitude for this wavenumber. Further, since the resolution is only 64×64, higher modes cannot be modeled, which, as seen on the KS equation, would increase the benefit of PDE-Refiner. This leads to both PDE-Refiner and the MSE-trained baseline to model all frequencies accurately. The slightly higher loss for higher frequencies on channel 0 is likely due to missing high-frequency information, i.e., larger resolution, that would be needed to estimate the frequencies more accurately. Still, we find that PDE-Refiner improves upon the MSE-trained model on all frequencies.

In Figure 22, we additionally plot the predictions of PDE-Refiner at different refinement steps. Similar

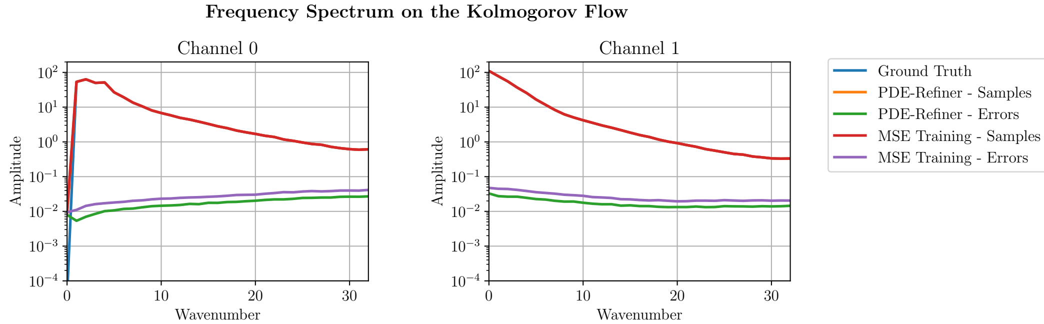


Figure 21: Frequency spectrum on the Kolmogorov Flow. The two plots show the two channels of the Kolmogorov flow. Since the data has a much more uniform support over frequencies than the KS equation, both the MSE-trained model and PDE-Refiner model the ground truth very accurately. Thus, the Ground Truth (blue), PDE-Refiner's prediction (orange) and the MSE-trained prediction (red) overlap in both plots. Plotting the error reveals that PDE-Refiner provides small gains across all frequencies.

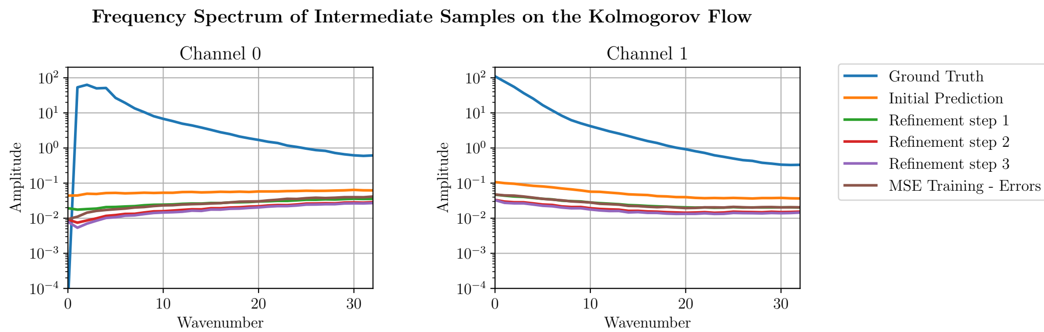


Figure 22: Frequency spectrum of intermediate samples in the refinement process of PDE-Refiner, similar to Figure 4 for the KS equation. The refinement process improves the prediction of the model step-by-step. For the last refinement step, we actually see minor improvements for the lowest frequencies of channel 0. However, due to flatter frequency spectrum, the high frequencies do not improve as much as on the KS equation.

to the KS equation, PDE-Refiner improves its prediction step by step. However, it is apparent that no clear bias towards the high frequencies occur in the last time step, since the error is rather uniform across all frequencies. Finally, the last refinement step only provides minor gains, indicating that PDE-Refiner with 2 refinement steps would have likely been sufficient.

E.6 Minimum noise variance in PDE-Refiner

Besides the number of refinement step, PDE-Refiner has as a second hyperparameter the minimum noise variance σ^2_{min}, i.e., the variance of the added noise in the last refinement step. The noise variance determines the different amplitude levels at which PDE-Refiner improves the prediction. To show how sensitive PDE-Refiner is to different values of σ^2_{min}, we repeat the experiments of PDE-Refiner on the KS equation while varying σ^2_{min}. The results in Figure 23 show that PDE-Refiner is robust to small changes of σ^2_{min} and there exist a larger band of values where it performs equally well. When increasing the variance further, the performance starts to decrease since the noise is too high to model the lowest amplitude information. Note that the results on Figure 23 show the performance on the test set, while the hyperparameter selection, in which we selected $\sigma^2_{\text{min}} = 2\text{e-}7$, was done on the validation set.

In combination with the hyperparameter of the number of refinement steps, to which PDE-Refiner showed to also be robust if more than 3 steps is chosen, PDE-Refiner is not very sensitive to the newly introduced hyperparameters and values in a larger range can be considered.

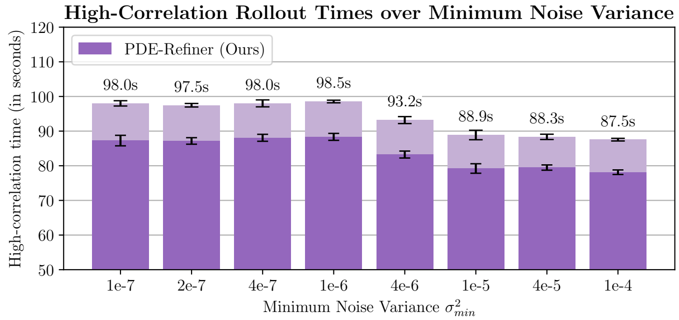


Figure 23: Plotting performance of PDE-Refiner over different values of the minimum noise variance σ^2_{min}. Each PDE-Refiner is robust to small changes of σ^2_{min}, showing an equal performance in the range of $\left[10^{-7}, 10^{-6}\right]$. Higher standard deviations start to decrease the performance, confirming our analysis of later refinement steps focusing on low-amplitude information. For the experiments in Section 4.1, we have selected σ^2_{min} =2e-7 based on the validation dataset.

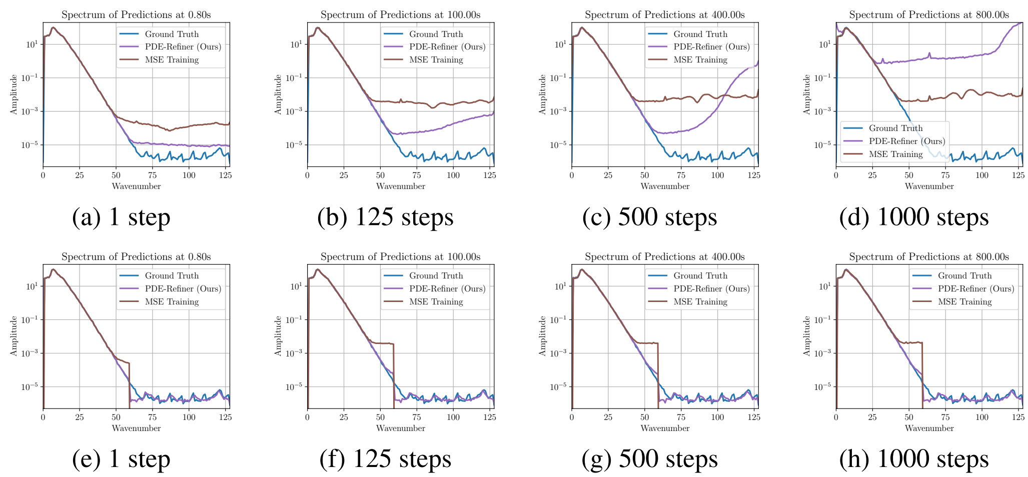


Figure 24: Evaluating PDE solver stability over very long rollouts (800 seconds, corresponding to 1000 autoregressive prediction steps). **(a-d)** The frequency spectrum of predictions of an MSE-trained model and PDE-Refiner. Over time, the MSE baseline's overestimation of the high frequencies accumulates. In comparison, PDE-Refiner shows to have an increase of extremely high frequencies, which is likely caused by the continuous adding of Gaussian noise. **(e-h)** When we apply the error correction [56] on our models by setting all frequencies above 60 to zero, PDE-Refiner remains stable even for 1000 steps and does not diverge from the ground truth frequency spectrum.

E.7 Stability of Very Long Rollouts

Besides accurate rollouts, another important aspect of neural PDE solvers is their stability. This refers to the solvers staying in the solution domain and not generating physically unrealistic results. To evaluate whether our solvers remain stable for a long time, we roll out an MSE-trained baseline and PDE-Refiner for 1000 autoregressive prediction steps, which corresponds to 800 seconds simulation time. We then perform a frequency analysis and plot the spectra in Figure 24. We compare the spectra to the ground truth initial condition, to have a reference point of common frequency spectra of solutions on the KS equation.

For the MSE-trained baseline, we find that the high frequencies, that are generally overestimated by the model, accumulate over time. Still, the model maintains a frequency spectrum close to the ground truth for wavenumbers below 40. PDE-Refiner maintains an accurate frequency spectrum for more than 500 steps, but suffers from overestimating the very high frequencies in very long rollouts. This is likely due to the iterative adding of Gaussian noise, that accumulates high-frequency errors. Further, the U-Net has a limited receptive field such that the model cannot estimate the highest frequencies properly. With larger architectures, this may be preventable.

However, a simpler alternative is to correct the predictions for known invariances, as done in McGreivy et al. [56]. We use the same setup as for Figure 3 by setting the highest frequencies to zero. This stabilizes PDE-Refiner, maintaining a very accurate estimation of the frequency spectrum even at 800 seconds. The MSE-trained model yet suffers from an overestimation of the high-frequencies.

In summary, the models we consider here are stable for much longer than they remain accurate to the ground truth. Further, with a simple error correction, PDE-Refiner can keep up stable predictions for more than 1000 autoregressive rollout steps.

References

[1] Troy Arcomano, Istvan Szunyogh, Alexander Wikner, Jaideep Pathak, Brian R Hunt, and Edward Ott. 2022. A Hybrid Approach to Atmospheric Modeling That Combines Machine Learning With a Physics-Based Numerical Model. *Journal of Advances in Modeling Earth Systems*, 14(3):e2021MS002712.

[2] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P Brenner. 2019. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349.

[3] David Berthelot, Arnaud Autef, Jierui Lin, Dian Ang Yap, Shuangfei Zhai, Siyuan Hu, Daniel Zheng, Walter Talbot, and Eric Gu. 2023. TRACT: Denoising Diffusion Models with Transitive Closure Time-Distillation. *arXiv preprint arXiv:2303.04248*.

[4] Saakaar Bhatnagar, Yaser Afshar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. 2019. Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics*, 64(2):525–545.

[5] Neill E. Bowler. 2006. Comparison of error breeding, singular vectors, random perturbations and ensemble Kalman filter perturbation strategies on a simple model. *Tellus A: Dynamic Meteorology and Oceanography*, 58(5):538–548.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. 2018. Software URL: http://github.com/google/jax.

[7] Johannes Brandstetter, Rianne van den Berg, Max Welling, and Jayesh K Gupta. 2023. Clifford Neural Layers for PDE Modeling. In *The Eleventh International Conference on Learning Representations*.

[8] Johannes Brandstetter, Max Welling, and Daniel E Worrall. 2022. Lie Point Symmetry Data Augmentation for Neural PDE Solvers. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 2241–2256. PMLR.

[9] Johannes Brandstetter, Daniel E. Worrall, and Max Welling. 2022. Message Passing Neural PDE Solvers. In *International Conference on Learning Representations*.

[10] Steven L Brunton and J Nathan Kutz. 2023. Machine Learning for Partial Differential Equations. *arXiv preprint arXiv:2303.17078*.

[11] Ashesh Chattopadhyay and Pedram Hassanzadeh. 2023. Long-term instabilities of deep learning-based digital twins of the climate system: The cause and a solution. *arXiv preprint arXiv:2304.07029*.

[12] Prafulla Dhariwal and Alexander Nichol. 2021. Diffusion models beat gans on image synthesis. *Advances in Neural Information Processing Systems*, 34:8780–8794.

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*.

[14] William Falcon and The PyTorch Lightning team. PyTorch Lightning. 2019. Software URL: `https://github.com/Lightning-AI/lightning`.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

[16] Jonathan Gordon, Wessel P Bruinsma, Andrew YK Foong, James Requeima, Yann Dubois, and Richard E Turner. 2019. Convolutional conditional neural processes. In *International Conference on Learning Representations*.

[17] Daniel Greenfeld, Meirav Galun, Ronen Basri, Irad Yavneh, and Ron Kimmel. 2019. Learning to Optimize Multigrid PDE Solvers. In *International Conference on Machine Learning (ICML)*, pages 2415–2423.

[18] Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. 2020. Hippo: Recurrent memory with optimal polynomial projections. *Advances in neural information processing systems*, 33:1474–1487.

[19] Albert Gu, Karan Goel, and Christopher Re. 2022. Efficiently Modeling Long Sequences with Structured State Spaces. In *International Conference on Learning Representations*.

[20] Xiaoxiao Guo, Wei Li, and Francesco Iorio. 2016. Convolutional neural networks for steady flow approximation. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 481–490.

[21] Jayesh K Gupta and Johannes Brandstetter. 2022. Towards Multi-spatiotemporal-scale Generalized PDE Modeling. *arXiv preprint arXiv:2209.15616*.

[22] Ernst Hairer and Gerhard Wanner. 1996. Solving ordinary differential equations. II, volume 14 of Springer Series in Computational Mathematics.

[23] Jiequn Han, Arnulf Jentzen, and Weinan E. 2018. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510.

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

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Computer Vision – ECCV 2016*, pages 630–645, Cham. Springer International Publishing.

[26] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

[27] Philipp Hennig, Michael A Osborne, and Hans P Kersting. 2022. *Probabilistic Numerics: Computation as Machine Learning*. Cambridge University Press.

[28] Jonathan Ho, William Chan, Chitwan Saharia, Jay Whang, Ruiqi Gao, Alexey Gritsenko, Diederik P Kingma, Ben Poole, Mohammad Norouzi, David J Fleet, et al. 2022. Imagen video: High definition video generation with diffusion models. *arXiv preprint arXiv:2210.02303*.

[29] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising Diffusion Probabilistic Models. In *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851. Curran Associates, Inc.

[30] Jonathan Ho, Chitwan Saharia, William Chan, David J Fleet, Mohammad Norouzi, and Tim Salimans. 2022. Cascaded Diffusion Models for High Fidelity Image Generation. *J. Mach. Learn. Res.*, 23(47):1–33.

[31] Emiel Hoogeboom and Tim Salimans. 2023. Blurring Diffusion Models. In *The Eleventh International Conference on Learning Representations*.

[32] Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. 2019. Learning Neural PDE Solvers with Convergence Guarantees. *arXiv preprint arXiv:1906.01200*.

[33] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95. Software URL: `https://github.com/matplotlib/matplotlib`.

[34] James M. Hyman and Basil Nicolaenko. 1986. The Kuramoto-Sivashinsky equation: A bridge between PDE'S and dynamical systems. *Physica D: Nonlinear Phenomena*, 18(1):113–126.

[35] Arieh Iserles. 2009. *A first course in the numerical analysis of differential equations*. 44. Cambridge university press.

[36] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. 2021. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440.

[37] Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. 2022. Elucidating the Design Space of Diffusion-Based Generative Models. In *Advances in Neural Information Processing Systems*.

[38] Ryan Keisler. 2022. Forecasting Global Weather with Graph Neural Networks. *arXiv preprint arXiv:2202.07575*.

[39] Ioannis G. Kevrekidis, Basil Nicolaenko, and James C. Scovel. 1990. Back in the Saddle Again: A Computer Assisted Study of the Kuramoto–Sivashinsky Equation. *SIAM Journal on Applied Mathematics*, 50(3):760–790.

[40] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[41] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

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

[43] Yoshiki Kuramoto. 1978. Diffusion-induced chaos in reaction systems. *Progress of Theoretical Physics Supplement*, 64:346–367.

[44] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6405–6416, Red Hook, NY, USA. Curran Associates Inc.

[45] Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Alexander Pritzel, Suman Ravuri, Timo Ewalds, Ferran Alet, Zach Eaton-Rosen, et al. 2022. GraphCast: Learning skillful medium-range global weather forecasting. *arXiv preprint arXiv:2212.12794*.

[46] Sangyun Lee, Hyungjin Chung, Jaehyeon Kim, and Jong Chul Ye. 2022. Progressive deblurring of diffusion models for coarse-to-fine image synthesis. *arXiv preprint arXiv:2207.11192*.

[47] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. 2020. Neural operator: Graph kernel network for partial differential equations. *arXiv preprint arXiv:2003.03485*.

[48] Zongyi Li, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Burigede liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. 2021. Fourier Neural Operator for Parametric Partial Differential Equations. In *International Conference on Learning Representations*.

[49] Zongyi Li, Miguel Liu-Schiaffini, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. 2022. Learning Chaotic Dynamics in Dissipative Systems. In *Advances in Neural Information Processing Systems*.

[50] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[51] Lu Lu, Pengzhan Jin, and George Em Karniadakis. 2019. DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*.

[52] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. 2021. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229.

[53] Lu Lu, Xuhui Meng, Shengze Cai, Zhiping Mao, Somdatta Goswami, Zhongqiang Zhang, and George Em Karniadakis. 2022. A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data. *Computer Methods in Applied Mechanics and Engineering*, 393:114778.

[54] Hao Ma, Yuxuan Zhang, Nils Thuerey, Xiangyu Hu, and Oskar J Haidn. 2021. Physics-driven learning of the steady Navier-Stokes equations using deep convolutional neural networks. *arXiv preprint arXiv:2106.09301*.

[55] James M. McDonough. 2007. *Lectures in Computational Fluid Dynamics of Incompressible Flow: Mathematics, Algorithms and Implementations*. 4. Mechanical Engineering Textbook Gallery.

[56] Nick McGreivy and Ammar Hakim. 2023. Invariant preservation in machine learned PDE solvers via error correction. *arXiv preprint arXiv:2303.16110*.

[57] Jonas Mikhaeil, Zahra Monfared, and Daniel Durstewitz. 2022. On the difficulty of learning chaotic dynamics with RNNs. In *Advances in Neural Information Processing Systems*, volume 35, pages 11297–11312. Curran Associates, Inc.

[58] Tung Nguyen, Johannes Brandstetter, Ashish Kapoor, Jayesh K Gupta, and Aditya Grover. 2023. ClimaX: A foundation model for weather and climate. *arXiv preprint arXiv:2301.10343*.

[59] Alexander Quinn Nichol and Prafulla Dhariwal. 2021. Improved denoising diffusion probabilistic models. In *International Conference on Machine Learning*, pages 8162–8171. PMLR.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc. Software URL: `https://github.com/pytorch/pytorch`.

[61] Jaideep Pathak, Shashank Subramanian, Peter Harrington, Sanjeev Raja, Ashesh Chattopadhyay, Morteza Mardani, Thorsten Kurth, David Hall, Zongyi Li, Kamyar Azizzadenesheli, Pedram Hassanzadeh, Karthik Kashinath, and Animashree Anandkumar. 2022. FourCastNet: A Global Data-driven High-resolution Weather Model using Adaptive Fourier Neural Operators. *arXiv preprint arXiv:2202.11214*.

[62] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. 2018. FiLM: Visual Reasoning with a General Conditioning Layer. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press.

[63] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, and Thomas Wolf. 2022. Diffusers: State-of-the-art diffusion models. Software URL: `https://github.com/huggingface/diffusers`.

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

[65] Stephan Rasp and Nils Thuerey. 2021. Data-driven medium-range weather prediction with a resnet pretrained on climate simulations: A new model for weatherbench. *Journal of Advances in Modeling Earth Systems*, 13(2):e2020MS002405.

[66] 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. 2022. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in Neural Information Processing Systems*, 35:36479–36494.

[67] Tim Salimans and Jonathan Ho. 2022. Progressive Distillation for Fast Sampling of Diffusion Models. In *International Conference on Learning Representations*.

[68] Sebastian Scher and Gabriele Messori. 2021. Ensemble Methods for Neural Network-Based Weather Forecasts. *Journal of Advances in Modeling Earth Systems*, 13(2).

[69] Jacob H Seidman, Georgios Kissas, George J Pappas, and Paris Perdikaris. 2023. Variational Autoencoding Neural Operators. *arXiv preprint arXiv:2302.10351*.

[70] G.I. Sivashinsky. 1977. Nonlinear analysis of hydrodynamic instability in laminar flames—I. Derivation of basic equations. *Acta Astronautica*, 4(11):1177–1206.

[71] Yiorgos S. Smyrlis and Demetrios T. Papageorgiou. 1991. Predicting Chaos for Infinite Dimensional Dynamical Systems: The Kuramoto-Sivashinsky Equation, A Case Study. *Proceedings of the National Academy of Sciences of the United States of America*, 88(24):11129–11132.

[72] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. 2015. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning*, pages 2256–2265. PMLR.

[73] Casper Kaae Sønderby, Lasse Espeholt, Jonathan Heek, Mostafa Dehghani, Avital Oliver, Tim Salimans, Shreya Agrawal, Jason Hickey, and Nal Kalchbrenner. 2020. Metnet: A neural weather model for precipitation forecasting. *arXiv preprint arXiv:2003.12140*.

[74] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. 2021. Score-Based Generative Modeling through Stochastic Differential Equations. In *International Conference on Learning Representations*.

[75] Zhiqing Sun, Yiming Yang, and Shinjae Yoo. 2023. A Neural PDE Solver with Temporal Stencil Modeling. *arXiv preprint arXiv:2302.08105*.

[76] Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Dan MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. 2022. PDEBench: An Extensive Benchmark for Scientific Machine Learning. In *36th Conference on Neural Information Processing Systems (NeurIPS 2022) Track on Datasets and Benchmarks*.

[77] Nils Thuerey, Philipp Holl, Maximilian Mueller, Patrick Schnell, Felix Trost, and Kiwon Um. 2021. Physics-based Deep Learning. *arXiv preprint arXiv:2109.05237*.

[78] Jakub M Tomczak. 2022. *Deep generative modeling*. Springer.

[79] Kiwon Um, Robert Brand, Yun (Raymond) Fei, Philipp Holl, and Nils Thuerey. 2020. Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers. In *Advances in Neural Information Processing Systems*, volume 33, pages 6111–6122. Curran Associates, Inc.

[80] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[81] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272. Software URL: `https://github.com/scipy/scipy`.

[82] Sifan Wang and Paris Perdikaris. 2023. Long-time integration of parametric evolution equations with physics-informed deeponets. *Journal of Computational Physics*, 475:111855.

[83] Sifan Wang, Hanwen Wang, and Paris Perdikaris. 2021. Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. *Science advances*, 7(40):eabi8605.

[84] Daniel Watson, William Chan, Jonathan Ho, and Mohammad Norouzi. 2022. Learning Fast Samplers for Diffusion Models by Differentiating Through Sample Quality. In *International Conference on Learning Representations*.

[85] Jonathan A Weyn, Dale R Durran, and Rich Caruana. 2020. Improving data-driven global weather prediction using deep convolutional neural networks on a cubed sphere. *Journal of Advances in Modeling Earth Systems*, 12(9):e2020MS002109.

[86] Yuxin Wu and Kaiming He. 2018. Group Normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[87] Yasin Yazıcı, Chuan-Sheng Foo, Stefan Winkler, Kim-Hui Yap, Georgios Piliouras, and Vijay Chandrasekhar. 2019. The Unusual Effectiveness of Averaging in GAN Training. In *International Conference on Learning Representations*.