

# Hierarchical Deep Learning of Multiscale Differential Equation Time-Steppers

Yuying Liu<sup>†\*</sup>, J. Nathan Kutz<sup>†,‡</sup>, Steven L. Brunton<sup>†,‡</sup>

<sup>†</sup> Department of Applied Mathematics, University of Washington, Seattle, WA 98105

<sup>‡</sup>Department of Mechanical Engineering, University of Washington, Seattle, WA 98105

## Abstract

Nonlinear differential equations rarely admit closed-form solutions, thus requiring numerical time-stepping algorithms to approximate solutions. Further, many systems characterized by multiscale physics exhibit dynamics over a vast range of timescales, making numerical integration computationally expensive due to numerical stiffness. In this work, we develop a hierarchy of deep neural network time-steppers to approximate the flow map of the dynamical system over a disparate range of time-scales. The resulting model is purely data-driven and leverages features of the multiscale dynamics, enabling numerical integration and forecasting that is both accurate and highly efficient. Moreover, similar ideas can be used to couple neural network-based models with classical numerical time-steppers. Our multiscale hierarchical time-stepping scheme provides important advantages over current time-stepping algorithms, including (i) circumventing numerical stiffness due to disparate time-scales, (ii) improved accuracy in comparison with leading neural-network architectures, (iii) efficiency in long-time simulation/forecasting due to explicit training of slow time-scale dynamics, and (iv) a flexible framework that is parallelizable and may be integrated with standard numerical time-stepping algorithms. The method is demonstrated on a wide range of nonlinear dynamical systems, including the Van der Pol oscillator, the Lorenz system, the KuramotoSivashinsky equation, and fluid flow pass a cylinder; audio and video signals are also explored. On the sequence generation examples, we benchmark our algorithm against state-of-the-art methods, such as LSTM, reservoir computing, and clockwork RNN. Despite the structural simplicity of our method, it outperforms competing methods on numerical integration.

**Keywords:** Deep learning, Multiscale modeling, Numerical stiffness, Scientific computing, Dynamical systems

## 1 Introduction

Scientific computing has revolutionized nearly every scientific discipline, allowing for the ability to model, simulate, engineer, and optimize a complex system's design and performance. This capability has been especially important in nonlinear, multiscale systems where recourse to analytic and perturbation methods are limited. For instance, modern high-fidelity simulations enable researchers to design aircraft, simulate the evolution of galaxies, quantify atmospheric and ocean interactions for weather forecasting, and model high-dimensional neuronal networks of the brain. Thus, given a set of governing equations, typically spatio-temporal *partial differential equations* (PDEs), discretization in time and space form the foundational algorithmic structure of scientific computing [1–3]. Discretization is required to accurately resolve all relevant spatial and temporal scales in order to produce a high-fidelity representation of the dynamics. Such resolution can be prohibitively expensive, as resolving physics on fast time scales limits simulation times and

---

\*email: yliu814@uw.edu

the ability to model slow timescale processes, i.e. it results in well-known *numerical stiffness* [4, 5]. Time-stepping schemes are typically based on Taylor series expansions, which are local in time and have a numerical accuracy determined by the step size  $\Delta t$ . However, there is a rapidly growing effort to develop *deep neural networks* (DNNs) to learn time-stepping schemes unrestricted by local Taylor series constraints [6–9]. We build on the flow map viewpoint of dynamical systems [8, 10, 11] in order to learn *hierarchical time-steppers* (HiTSs) that explicitly exploit the multiscale flow map structure of a dynamical system over a disparate range of time-scales. In leveraging features at different timescales, we can circumvent numerical stiffness and produce an accurate and efficient computational scheme that can provide exceptional efficiency in long-time simulation/forecasting and that can be integrated with classical time-stepping algorithms.

Numerical discretization has been extensively studied since the earliest days of scientific computing. Numerical analysis has provided rigorous estimates of error bounds for the diversity of discretization schemes developed over the past few decades [1–3]. Spatial discretization predominantly involves finite element, finite difference, or spectral methods. Multigrid methods have been extensively developed in physics-based simulations where coarse grained models must be progressively refined in order to achieve a required numerical precision while remaining tractable [12, 13]. The resulting discretized dynamics may be generically represented as a nonlinear dynamical system of the form

$$\frac{d}{dt} \mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), t) \quad (1)$$

in terms of a state  $\mathbf{x} \in \mathbb{R}^D$  (typically  $D \gg 1$ ). The dynamics are then integrated with a time-stepping algorithm. As with spatial discretization, there is a wide range of techniques developed for time-stepping, including explicit and implicit schemes, which have varying degrees of stability and accuracy. These schemes approximate the discrete-time flow map [14, 15]

$$\mathbf{x}(t + \Delta t) = \mathbf{F}(\mathbf{x}(t), \Delta t) \triangleq \int_t^{t + \Delta t} \mathbf{f}(\mathbf{x}(\tau), \tau) d\tau, \quad (2)$$

often through a Taylor-series expansion. Runge-Kutta, for which the Euler method is a subset, is one of the standard time-stepping schemes used in practice. Generically, it takes the form

$$\mathbf{x}_{n+1} \approx \tilde{\mathbf{F}}_{\Delta t}(\mathbf{x}_n) \triangleq \mathbf{x}_n + \Delta t \sum_{j=1}^k b_j \mathbf{h}_j \quad (3)$$

where

$$\mathbf{h}_j = \mathbf{f} \left( \mathbf{x}_n + g \left( \sum_{k=1}^{j-1} \alpha_{j,k} \mathbf{h}_k \right), t_n + c_j \Delta t \right) \quad (4)$$

and  $\mathbf{x}_n = \mathbf{x}(t_n) = \mathbf{x}(n\Delta t)$ . Note that the contributions  $\mathbf{h}_j$  are hierarchically computed in the Runge-Kutta scheme. The weightings  $b_j$ ,  $c_j$  and  $\alpha_{j,k}$  are derived from Taylor series expansions in order to minimize error. For instance, the classic fourth-order Runge-Kutta scheme, for which  $k = 4$  above, has a local truncation error of  $\mathcal{O}(\Delta t^5)$ , which leads to a global time-stepping error of  $\mathcal{O}(\Delta t^4)$ . Euler stepping, for which  $k = 1$ , has local and global time-stepping errors of  $\mathcal{O}(\Delta t)$  and  $\mathcal{O}(\Delta t^2)$  respectively. Importantly, the error is explicitly related to the time-step  $\Delta t$ , making such time-discretization schemes *local* in nature.

In contrast to schemes such as Runge-Kutta, that approximate the flow map with a local Taylor series, it is possible to directly construct an approximate flow map  $\hat{\mathbf{F}}$  using DNN architectures. There are several approaches to modeling flow-map time-steppers using neural networks, which will be reviewed below. The approach taken in this work is to develop a hierarchy of approximate flow maps  $\hat{\mathbf{F}}_j(\mathbf{x}, \Delta t_j)$  to facilitate the accurate and efficient simulation of multiscale systems over a range of time-scales; similar flow map composition schemes have been demonstrated to be highly effective for simulating differential equations without neural networks [10, 11, 16]. Flow maps also provide a robust framework for model discovery of multiscale physics [17, 18]. Various existing DNN architectures can be integrated into this hierarchical framework. Flow map approximations based on the Taylor series are typically only valid for small time steps, as the flow map becomes arbitrarily complex for large time steps in chaotic systems. However, DNN

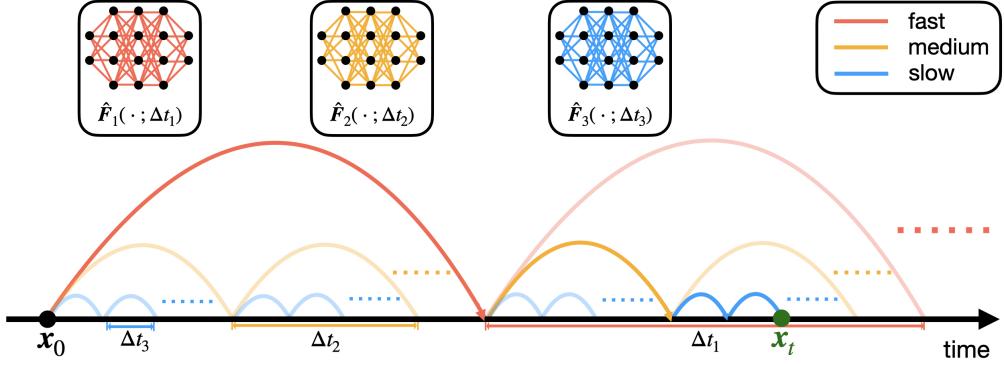


Figure 1: **Multiscale hierarchical time-stepping scheme.** Here, we employ neural network time-steppers over three time scales. The red model takes large steps, leaving the finer time-stepping to the yellow and blue models. The dark path shows the sequence of maps from  $x_0$  to  $x_t$ .

architectures are not limited by this small time step constraint, as they may be sufficiently descriptive to approximate exceedingly complex flow map functions [19].

Neural networks have been used to model dynamical systems for decades [20, 21]. They are computational models that are composed of multiple layers and are used to learn representations of data [22, 23]. Due to their remarkable performance on many data-driven tasks [24–29], various new architectures that favor interpretability and promote physical insight have been recently proposed, leading to many successful applications. In particular, it has been shown that neural networks may be used in conjunction with classical methods in numerical analysis to obtain discrete time-steppers [6–8, 30, 31]. Other applications include reduced order modeling [32, 33], multiscale modeling [34–37], scientific computing [38–40], coordinate transformations [41, 42], attractor reconstructions [43, 44], and forecasting [45–48]. Neural network models are increasingly popular for two reasons. First, the universal approximation theorem guarantees that arbitrary continuous functions can be approximated by neural networks with sufficiently many hidden units [49]. Second, neural networks themselves can be viewed as discretizations of continuous dynamical systems [50–56], which makes them suitable for studying dynamics. Among all architectures, *recurrent neural networks* (RNNs) are natural candidates for temporal sequence modeling; however, training has proven to be especially difficult due to the notorious exploding/vanishing gradient problem [57, 58]. To alleviate this problem, new architectures have been proposed [59–61], for example, augmenting the network with explicit memory using gating mechanism, resulting in the *long short term memory* (LSTM) algorithm [59], or adding skipped connections to the network, leading to *residual networks* (ResNet) [62].

In this work, we expand on [8] and employ a deep *residual network* (ResNet) as the basic building block for modeling the flow-map dynamics (2). Unconstrained by the typical form of a Taylor-series based time-stepping scheme (3), a multiscale modeling perspective is taken to strengthen the performance of our proposed multiscale, flow-map models. Our multiscale HiTS algorithm consists of ResNet models trained to perform hierarchical time-stepping tasks. The contributions of this work are summarized as follows:

- We propose a novel method to couple neural network HiTSs trained across different time scales, shown in Fig. 1, resulting in more accurate future state forecasts without losing computational efficiency.
- Neural network HiTSs may be coupled with classical numerical time-steppers. This hybrid time-stepping scheme can be naturally parallelized, accelerating classical numerical simulation algorithms.
- By coupling models across different scales, each individual model only need to be trained over a short period without being exposed to the exploding/vanishing gradient problem, enabling faster training.
- Despite the structural simplicity, the coupled model can still be used to capture long-term dependencies, achieving state-of-the-art performance on sequence generation.

The paper is organized as follows. We motivate the proposed approach and present the methodology in Sec. 2. Our approach is then tested on several benchmark problems in Sec. 3. In Sec. 4, we conclude and discuss future directions. Our code is publicly available at [https://github.com/luckystarufo/multiscale\\_HiTS](https://github.com/luckystarufo/multiscale_HiTS).

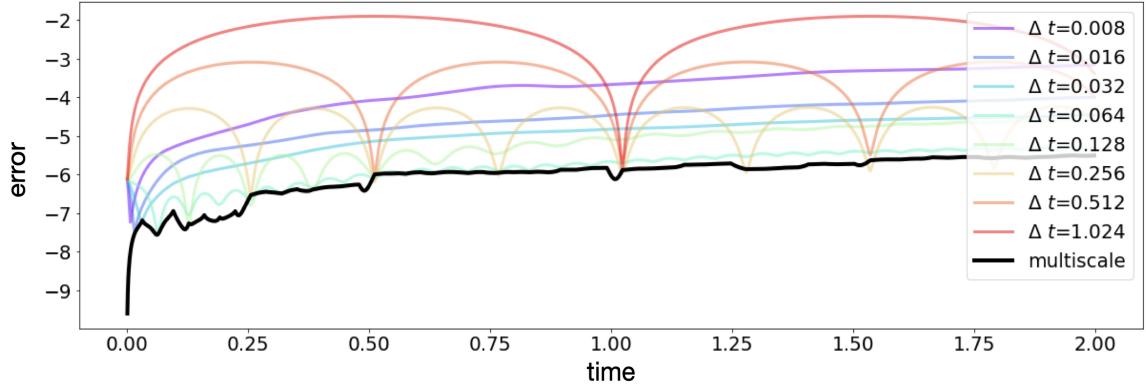


Figure 2: **Performance of multiscale HiTS on harmonic oscillator example.** This figure shows the time-stepping performance of different neural network time-steppers. 100 testing trajectories are used for benchmarking each time-stepper and the mean squared errors at each step are plotted in the base-10 logarithmic scale. The black curve represents our proposed multiscale scheme whereas other colors represent time-steppers at particular scales.

## 2 Multiscale Time-Stepping with Deep Learning

Here we outline our multiscale hierarchical time-stepping based on deep learning, illustrated in Fig. 1. Our approach constructs a hierarchy of flow maps,  $\hat{\mathbf{F}}_j(\mathbf{x}, \Delta t_j)$ , each approximated with a deep neural network. This enables accurate and efficient simulations with fine temporal resolution over long time scales, as compared in Fig. 2. We begin with a motivating example, followed by a summary of notation and description of the training data. Then we will introduce our multiscale hierarchical time-stepping scheme, including descriptions on how to vectorize operations and create hybrid time-steppers by combining with classical numerical methods.

### 2.1 Motivating example

To explore the effect of time step size on simulation performance, we consider the following simple linear differential equation for a harmonic oscillator

$$\dot{x} = y \tag{5a}$$

$$\dot{y} = -x. \tag{5b}$$

We individually train 8 neural networks with step sizes  $\Delta t$  of 0.008, 0.016, 0.032, 0.064, 0.128, 0.256, 0.512 and 1.024 on 500 sampled trajectories and test them on 100 new trajectories. Linear interpolation is used to estimate the state at time steps that are not directly obtained from the neural network time-stepping scheme. To evaluate the forecasting performance, we calculate the averaged error at each time step against the ground truth analytical solution. In this experiment, training and testing data are both sampled from the region  $\{(x, y) | x^2 + y^2 \leq 1\}$ , and we only train one step forward for each neural network time-stepper.

Results are plotted in Fig. 2, where it is clear that the proposed multiscale time-stepper outperforms all fixed step models. Networks equipped with small time steps offer accurate short-term predictions, although error accumulates at each step and quickly dominates. Networks with large time steps can handle long-term predictions, although they fail to provide information between steps. Time-steppers with some intermediate step sizes (eg.  $\Delta t = 0.064$ ) perform well in general as they balance these two factors. However, by leveraging all time steppers across each scale, it is possible to create a multiscale HiTS, given by the black curve, that is both accurate and efficient over long time scales and with fine temporal resolution. Details of the methodology will be presented in the remainder of this section.

## 2.2 Notation and training data

For training data, we collect  $n$  trajectories sampled at  $p$  instances with time step  $\Delta t$ :

$$S^{(i)} = \{(\mathbf{x}_t^{(i)}, \mathbf{x}_{t+\Delta t}^{(i)}, \mathbf{x}_{t+2\Delta t}^{(i)}, \dots, \mathbf{x}_{t+p\Delta t}^{(i)})\} \quad (6)$$

for  $i = 0, 1, \dots, n - 1$ . This data is used to train neural network flow map approximations,  $\hat{\mathbf{F}}_j(\mathbf{x}, \Delta t_j)$ , for  $j = 0, 1, \dots, m - 1$ .

We follow [8] and employ the residual network as our fundamental building block of our deep learning architecture. Residual networks were first proposed in [62] and have gained considerable prominence since. Specifically, our network only models the difference between time steps  $\mathbf{x}_{n+1}$  and  $\mathbf{x}_n$ , so that it is technically the flow map minus the identity:

$$\hat{\mathbf{x}}_{t+\Delta t_j} = \mathbf{x}_t + \hat{\mathbf{F}}_j(\mathbf{x}, \Delta t_j) \quad (7)$$

where

$$\hat{\mathbf{F}}_j(\mathbf{x}, \Delta t_j) = \sigma_M(\mathbf{A}_{M-1}(\dots \sigma_1(\mathbf{A}_0)\dots)) \quad (8)$$

is a feed-forward neural network. The network is parameterized by the linear operators  $\mathbf{A}_j$ , and  $\sigma_j$  are nonlinear activation functions that are chosen to be rectified linear units (ReLU). The extra addition creates a skipped connection from the inputs to the outputs. Here, the architecture  $\hat{\mathbf{F}}_j$  learns the increment in states between each  $\Delta t_j$  step. It is possible to compose the networks to take multiple steps forward in time:  $\hat{\mathbf{F}}_j^{(k)} = \underbrace{\hat{\mathbf{F}}_j \circ \hat{\mathbf{F}}_j \circ \dots \circ \hat{\mathbf{F}}_j}_{k \text{ times}}$ . Finally, we formulate our training objective function as

$$MSE = \frac{1}{np} \sum_{i=1}^n \sum_{k=1}^p (\hat{\mathbf{x}}_{t+k\Delta t_j}^{(i)} - \mathbf{x}_{t+k\Delta t_j}^{(i)})^2 \quad (9)$$

which is the classical mean squared loss function.

## 2.3 Multiscale hierarchical time-stepping scheme

Multiscale modeling is ubiquitous in modern physics-based simulation models. Computational challenges arise in these simulations since coarse-grained macroscale models are usually not accurate enough and microscale models are too expensive to be used in practice [63]. By coupling macroscopic and microscopic models, we hope to take advantage of the simplicity and efficiency of the macroscopic models, as well as the accuracy of the microscopic models [64]. Many efforts have been made towards this goal, resulting in many algorithms that exploit multiscale structure in space and time, including the multi-grid method [65], the fast multipole method [66], adaptive mesh refinement [67], domain decomposition [68], multi-resolution representation [69], multi-resolution dynamic mode decomposition [70], etc. Mathematical algorithms such as heterogeneous multiscale modeling (HMM) [71, 72] and the equation-free approach [73] attempt to develop general guidelines and provide principled methods for this field. In this work, however, we develop data-driven models using neural networks, which have different considerations than physics-based simulation models. Regardless, the goal is still to produce accurate and efficient computational models.

Coupling neural network time-steppers across different time scales is rather straightforward, as shown in Fig. 1. One can clearly see the time-steppers with small  $\Delta t$  are responsible for the accurate time-stepping results over short periods, while the models with larger  $\Delta t$  steps are used to 'reset' the predictions over longer periods, preventing error accumulations from the short-time models. There are additional benefits of this multiscale coupling in time. First, training each individual network is simpler, as it is possible to use trajectories with small  $p$  so that each model may focus on its own range of interest, circumventing the problem of exploding/vanishing gradients. Second, the framework is flexible, so that for forecasting it is possible to vectorize the computations or utilize parallel computing technologies, enabling fast time-stepping schemes. Moreover, it can be easily combined with classical numerical time-steppers, resulting in hybrid schemes, boosting the performance of simulation algorithms.

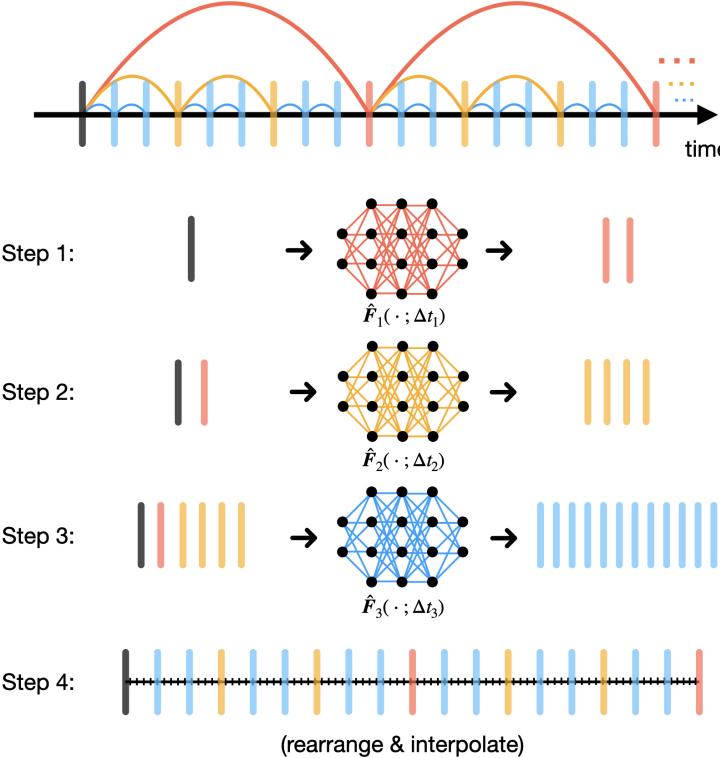


Figure 3: **Vectorized computation.** The three neural networks in this diagram are used sequentially, ordered by their associated step sizes from large to small. For each network, we stack all currently existing states and step forward (in the beginning, we only use the initial state), resulting in vectorized computations. These newly generated states are further fed to the next neural network in queue. Once we finish using all networks, the states will be rearranged in terms of chronological order and intermediate time steps will be obtained via interpolation.

The proposed multiscale coupling procedure may resemble those used in multiscale simulations (e.g., HMM). However, the data-driven microscopic cannot provide accurate long-time forecasts on its own, so the coupling here is not only for efficiency but also to improve the long-time fidelity of the model. It should also be noted that before coupling neural network time-steppers across different time scales, cross validation is used to filter the models. This is practically helpful because qualities of different models may vary and we want to couple the best set of models. Specifically, suppose we have  $m$  neural network models  $\{\hat{\mathbf{F}}_0, \hat{\mathbf{F}}_1, \dots, \hat{\mathbf{F}}_{m-1}\}$ , ordered by their associated step sizes. We first determine the upper bound index  $u$  so that ensembling  $\{\hat{\mathbf{F}}_0, \hat{\mathbf{F}}_1, \dots, \hat{\mathbf{F}}_u\}$  has the best time-stepping performance among  $\{\hat{\mathbf{F}}_0, \hat{\mathbf{F}}_1, \dots, \hat{\mathbf{F}}_k\}$  for all  $k$ . Next, we seek a lower bound index  $l$  so that  $\{\hat{\mathbf{F}}_l, \hat{\mathbf{F}}_{l+1}, \dots, \hat{\mathbf{F}}_u\}$  performs best among  $\{\hat{\mathbf{F}}_k, \hat{\mathbf{F}}_{k+1}, \dots, \hat{\mathbf{F}}_u\}$  for all  $k$ .

**Vectorization.** The diagram for vectorized computations is illustrated in Fig. 3. The basic idea is to start by using the time-steppers with the largest  $\Delta t$  step and generate the future states corresponding with this stepper. We then stack the new states with the original states and feed them to the next-level neural network time-stepper. After we proceed through all time-steppers, we rearrange the states and use interpolation to fill in the state at all intermediate time steps. Details are given in Algorithm 1.

**Hybrid time-steppers.** The flexibility of our multiscale coupling approach makes it possible to combine these time-steppers with classical numerical time-stepping algorithms. The algorithm is detailed in Algorithm 2 and the concept is illustrated in Fig. 4. This approach provides an innovation in the computational paradigm of numerical simulations. If one were to use classical numerical time-steppers (eg. Runge-Kutta method) alone, opportunities for vectorized computations or parallel computations are limited due to the

---

**Algorithm 1** Vectorized multiscale hierarchical time-stepping

---

**Input:** a set of neural network time-steppers  $\hat{\mathbf{F}}_s$   
**Output:** a list of predicted states  $X_s$  sorted in chronological order

```
1: function VECTORIZEDMULTISCALETIMESTEPPING( $\hat{\mathbf{F}}_s$ )
2:   Sort( $\hat{\mathbf{F}}_s$ );                                      $\triangleright$  models are ordered with decreasing step sizes
3:    $X_s = \text{List}();$                                  $\triangleright$  create an empty list
4:   Append( $\mathbf{x}_0, X_s$ );                          $\triangleright$  append initial state to the list
5:   for i in  $0, 1, \dots, m - 1$ : do
6:      $K_i :=$  the number of steps forward with  $\hat{\mathbf{F}}_i$ ;
7:      $X_{\text{cur}} = \text{Stack}(X_s);$                    $\triangleright$  stack together all states in  $X_s$ 
8:      $X_{\text{next}} = \hat{\mathbf{F}}_i.\text{Forward}(X_{\text{cur}}, K_i);$      $\triangleright$  step forward  $K_i$  steps with  $\hat{\mathbf{F}}_i$  and  $X_{\text{cur}}$ 
9:     Append( $X_{\text{next}}, X_s$ );                    $\triangleright$  update the list with new states
10:    end for
11:    Rearrange( $X_s$ );
12:    Interpolate( $X_s$ );
13:    return  $X_s$ 
14: end function
```

---

**Algorithm 2** Hybrid time-stepping scheme

---

**Input:** a set of neural network time-steppers  $\hat{\mathbf{F}}_s$   
**Output:** a list of predicted states  $X_s$  sorted in chronological order

```
1: function HYBRIDTIMESTEPPING( $\hat{\mathbf{F}}_s$ )
2:   Sort( $\hat{\mathbf{F}}_s$ ); (models are ordered with decreasing step sizes)
3:    $X_s = \text{List}();$                                  $\triangleright$  create an empty list
4:   // Use neural network time-steppers for large steps
5:   Append( $\mathbf{x}_0, X_s$ );                          $\triangleright$  append initial state to the list
6:   for i in  $0, 1, \dots, q - 1$ : do
7:      $K_i :=$  the number of steps forward with  $\hat{\mathbf{F}}_i$ ;
8:      $X_{\text{cur}} = \text{Stack}(X_s);$                    $\triangleright$  stack together all states in  $X_s$ 
9:      $X_{\text{next}} = \hat{\mathbf{F}}_i.\text{Forward}(X_{\text{cur}}, \hat{\mathbf{F}}_i);$      $\triangleright$  step forward  $K_i$  steps with  $\hat{\mathbf{F}}_i$  and  $X_{\text{cur}}$ 
10:    Append( $X_{\text{next}}, X_s$ );                    $\triangleright$  update the list with new states
11:   end for
12:   // Use classic numerical time-steppers for fine steps
13:    $K :=$  the number of steps forward with runge-kutta time stepper;
14:    $X_{\text{cur}} = \text{Stack}(X_s);$  (stack together all states in  $X_s$ )
15:    $X_{\text{next}} = \text{RK4}(X_{\text{cur}}, K);$      $\triangleright$  step forward  $K$  steps with runge-kutta time-stepper and  $X_{\text{cur}}$ 
16:   Append( $X_{\text{next}}, X_s$ );                    $\triangleright$  update the list with new states
17:   Rearrange( $X_s$ );
18:   return  $X_s$ 
19: end function
```

---

serialized nature: one cannot march forward to the future without knowledge about the past. Our hybrid scheme, on the other hand, bypasses the difficulty by utilizing large scaled neural network time-steppers, enabling vectorized computations (or parallel computations) at the bottom level where we use classical numerical time-steppers for accurate simulations.

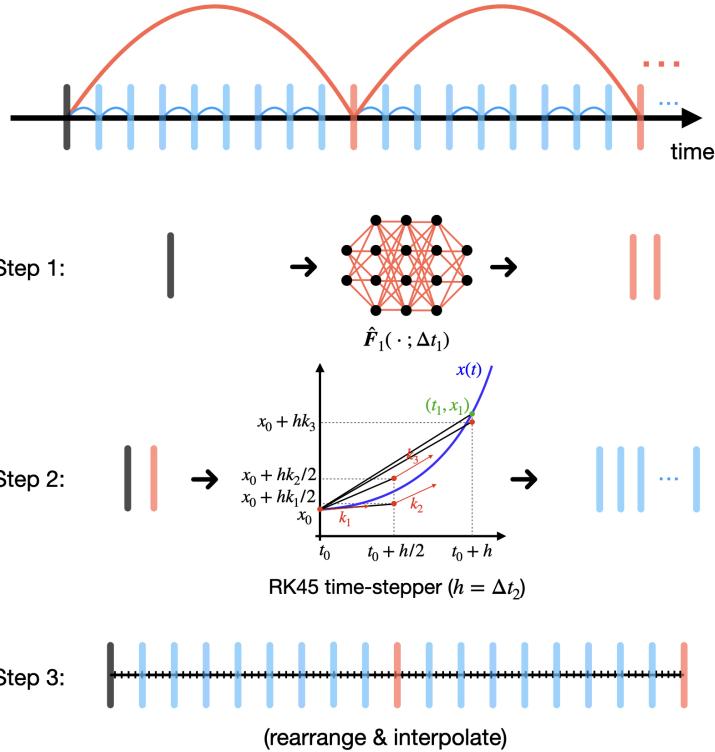


Figure 4: **Hybrid time-stepper.** A hierarchy of coarse neural network time-steppers generate states that are fed to a fourth order Runge-Kutta solver for fine-scale time-stepping.

### 3 Numerical Experiments

We will now provide a thorough exploration of our proposed multiscale hierarchical time-steppers. We begin by comparing against single time-scale neural network time-steppers and Runge-Kutta algorithms on simple dynamical systems. We then explore this approach on more sophisticated examples in spatiotemporal physics and sequence generation.

#### 3.1 Benchmark on time-stepping

We first benchmark the multiscale neural network HiTS against the single time-scale neural network time-steppers on five simple nonlinear dynamical systems: a nonlinear system with a hyperbolic fixed point, a damped cubic oscillator, the Van der Pol oscillator, a Hopf normal form, and the Lorenz system. For each example, we train 11 single time-scale neural network time-steppers with separate time steps, and then combine them into a multiscale neural network HiTS with the methods described in Section 2.3. More details about these numerical experiments can be found in Appendix A.1.

Fig. 5 shows that the multiscale scheme outperforms all single time-scale schemes in terms of accuracy, as shown by the black curve in the last column. On the sampled testing trajectory, in the third column, our multiscale scheme achieves nearly perfect time-stepping for a time period of 51.20 for the first four nonlinear systems. For the Lorenz system, discrepancy occurs in the forecast after about 5 time units, when the trajectory switches lobes. Indeed, the error plot suggests the time-stepper becomes unreliable even after a single time unit, due to the intrinsic chaotic dynamics; more details are discussed in Appendix B.4.

The trade-off between computational accuracy and efficiency are visualized in Fig. 6A. Here, we report the  $L_2$  error, averaged over all time steps and test trajectories. The single time-scale scheme curves have a “U” shape for each example, indicating that accuracy first improves and then deteriorates as we spend more time in the computation. This finding is consistent with [30], which states that there is a problem-dependent

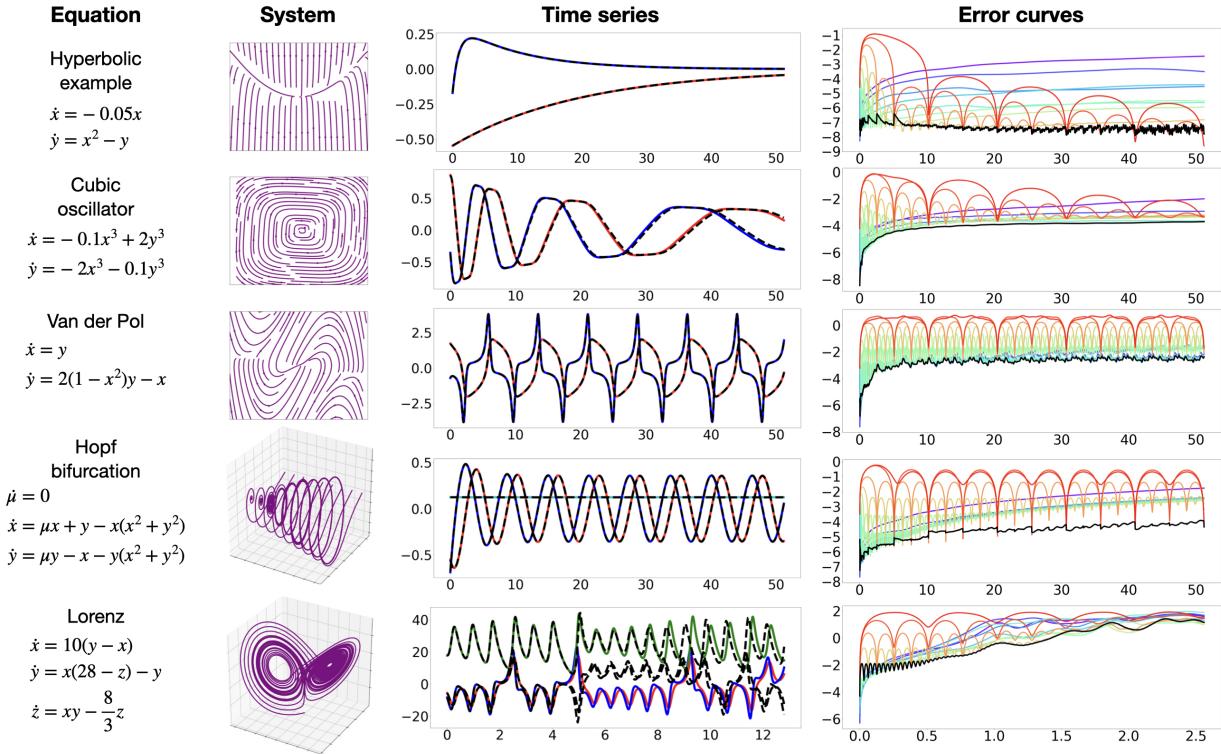
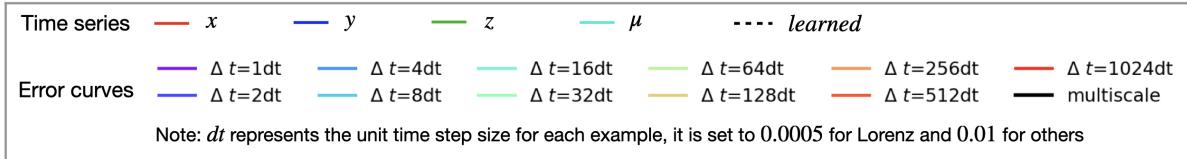


Figure 5: **Performance of multiscale HiTSs on nonlinear systems.** In the first two columns, systems of equations and phase portraits are visualized. In the third column, we visualize the predictions of multiscale time-steppers on a testing trajectory. In the last column, mean squared errors at each step are visualized in the base-10 logarithmic scale for different time-steppers.

sweet-spot for the hyperparameter  $\Delta t$ . Our multiscale HiTS always achieves the best accuracy, usually with a reasonable computational efficiency, due to the vectorized computations of array programming. For the cubic oscillator, Van der Pol oscillator, and Lorenz system, there seems to exist a single-scale neural network time-stepper with higher efficiency and competitive accuracy compared to our multiscale scheme. This is due to the greedy method we use for the cross validation process: we always prefer models with higher accuracy at the cost of efficiency. However, one can adjust the balance between accuracy and efficiency depending on the objective. The hyperparameter tuning of  $\Delta t$  is implicitly conducted in the cross validation process before coupling the various scales.

Though the multiscale scheme improves the computational efficiency of neural network time-steppers, they still cannot easily match the efficiency of most classic numerical algorithms (see Appendix B.3 for more details). Indeed, evaluating a neural network model (forward propagation) typically requires more computational effort than applying a classic discretization scheme that only involves a few evaluations of the known vector field. A hybrid time-stepper, on the other hand, may break this bottleneck by combining large-scale neural network time-steppers with classic numerical time-steppers to make the computations

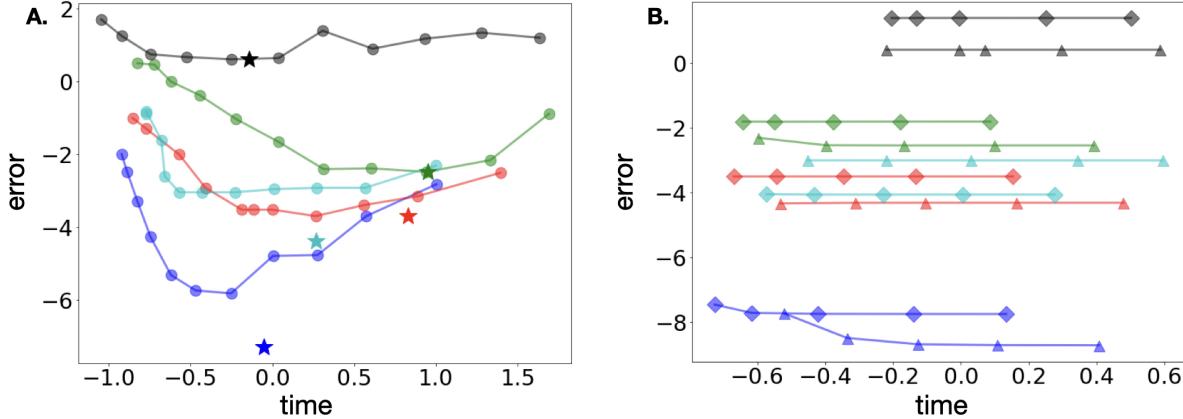


Figure 6: **Accuracy vs computational efficiency plot.** **A.** Comparison between multiscale neural network time-stepper and all single time-scale neural network time-steppers. **B.** Comparison between hybrid time-stepper and Runge-Kutta time-steppers with uniform step sizes. In both plots, horizontal and vertical axes represent time and integrated  $\mathcal{L}_2$  error respectively, visualized in the base-10 logarithmic scale.

inherently parallelizable. Therefore, we benchmark the hybrid time-steppers against the classic numerical time-steppers in Fig. 6B. In particular, we use a fourth-order RungeKutta integrator, with 7 different sizes of uniform time step. For simplicity, the hybrid time-steppers are constructed from the same fourth-order Runge-Kutta (RK4) time-steppers with one large-scale neural network time-stepper at the highest level. More details can be found in Appendix A.2.

Fig. 6B shows the trade-off between accuracy and efficiency for our hybrid time-stepper and the classic RK4 scheme. Our hybrid time-stepper offers an efficiency gain over the Runge-Kutta time-stepper with the same minimal step size. This suggests the marginal benefits of enabling vectorized computation is potentially greater than the costs of evaluating a neural network time-stepper. But the accuracy of hybrid time-steppers are usually slightly lower than the purely numerical time-steppers with rare exceptions (e.g., the Hopf normal form example), as the global error are usually dominated by the error of neural network time-steppers, which are model agnostic and purely data-driven, limiting their ability to produce high-fidelity simulations.

### 3.2 Benchmark on sequence generation

In addition to integrating simple low-dimensional dynamical systems, here we show that it is possible to forecast the state of more complex, high-dimensional dynamical systems. In the field of machine learning, this is often termed *sequence generation*. Importantly, we benchmark our architecture against state-of-the-art networks, including long short-term memory networks (LSTMs) [59], echo state networks (ESNs) [43], and clockwork recurrent neural networks (CW-RNNs). Here, our goal is to train different architectures that can generate the target sequence as accurately as possible. The sequences we explore include a simulated solution of the KuramotoSivashinsky (KS) equation, a music excerpt from Bach’s Fugue No. 1 In C Major, BWV 846, a simulation of fluid flow past a circular cylinder at Reynolds number 100 [74, 75], and a video frame of blooming flowers. Within each individual experiment, the various architectures have nearly the same

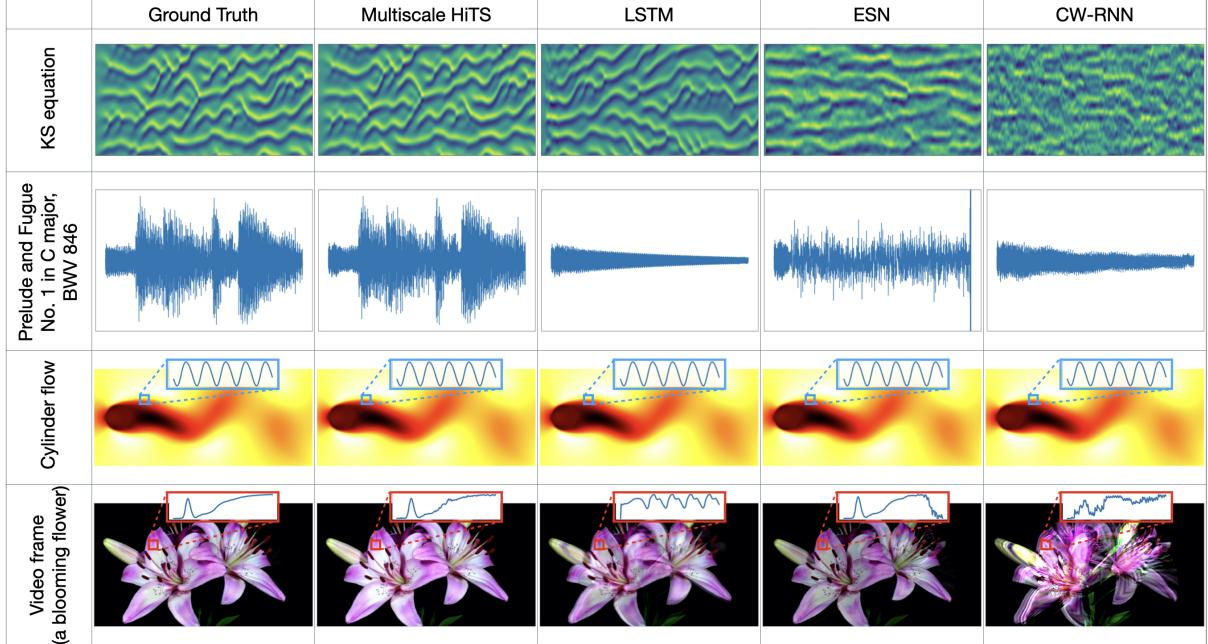


Figure 7: **Outputs of different network architectures (column) on each training sequence (row).** We use different visualization schemes to show the results: for the KS equation and the music excerpt, we plot the time series evolution, that is, the horizontal axes represent time; For the cylinder flow and the video frame, since each state is a 2D array, we choose to visualize the last frame of our reconstruction, however, we also visualize the time evolution of some states averaged over a small patch of pixel values. For a video that shows the performance, visit: <https://youtu.be/2psX5efLhCE>.

number of parameters; more details about the data preprocessing and choice of parameters are described in Appendix A.3.

From Fig. 7, one can visually see that the multiscale HiTS provides the best sequence generation results, and these results are confirmed by the integrated  $\mathcal{L}_2$  errors shown in Table 1. We also see that the LSTM and CW-RNN can learn the first few steps accurately, whereas the ESN tends to smooth the signals. It should be noted that our sequence generation task are very different from the tasks considered in [76], where they use observations of the systems past evolution to predict future states. An ESN is usually trained by finding a set of output weights through linear regression, if the reservoir is large enough, it can in principle reproduce the observed dynamics perfectly, which would make it an inappropriate benchmark. An ESN with the same number of parameters as the other architectures often cannot fully describe the dynamics, resulting in coarse or smoothed approximations.

In a nutshell, the competing architectures fail to capture the long-time behaviors, as error accumulation is inevitable for serialized computations. However, our proposed framework should not be viewed as a replacement for these state-of-the-art methods, as they take a different approach and philosophy for sequence generation. RNNs tend to uncover the full dynamics with the help of memory in their internal states, whereas our scheme performs reconstruction using the time-stepping schemes learned at different time scales, though these schemes may only be accurate for a few steps. Instead, our multiscale framework should be used to strengthen these existing approaches, as it can utilize data-driven models across different scales, avoiding local error accumulations and potentially boosting the accuracy and efficiency.

<b>Sequences</b>	<b>HiTS</b>	<b>LSTM</b>	<b>ESN</b>	<b>CW-RNN</b>
<b>Fluid flow</b>	1.23e-7	9.20e-8	2.22e-8	6.79e-7
<b>Video frame</b>	7.09e-5	1.44e-2	1.62e-3	8.05e-2
<b>Music data</b>	8.59e-7	4.65e-5	2.69e-4	4.70e-5
<b>KS equation</b>	1.04e-3	3.50e+0	3.51e+0	3.59e+0

Table 1: The integrated  $\mathcal{L}_2$  error between the generated sequence and the exact sequence.

## 4 Conclusions & Discussion

In this work, we have demonstrated an effective and general data-driven time-stepper framework based on synthesizing multiple deep neural networks for hierarchical time-steppers (HiTSs) trained at multiple temporal scales. Our approach outperforms neural networks trained at a single scale, providing an accurate and flexible approach for integrating nonlinear dynamical systems. We have carefully explored this multiscale HiTS approach on several illustrative dynamical systems as well as for a number of challenging high-dimensional problems in sequence generation. In the sequence generation examples, our approach outperforms state-of-the-art neural network architectures, including LSTMs, ESNs, and CW-RNNs. Our method explicitly takes advantage of dynamics on different scales by learning flow-maps for those different scales. The coupled model still maintains computational efficiency thanks to the vectorized computations of array programming. Moreover, exactly due to this coupling scheme, each individual network can focus on their intrinsic ranges of interest, bypassing the exploding/vanishing gradient problem for training recurrent neural networks. In addition, we demonstrate the joint use of our neural network time-steppers with the classical time-steppers, resulting in a new computational paradigm: numerical simulation algorithms are now parallelizable rather than serialized in nature, leading to performance boosts in computational speed.

This work highlights fundamental differences between physics-based simulation models and data-driven models. In the former, the error of the time-stepping constraints are determined strictly by Taylor series expansions which are local in nature and limit the time-step  $\Delta t$ . The latter is a more general flow-map construction that can be trained for any time step  $\Delta t$ . Thus the error is not limited by a local Taylor expansion, but rather by pairs of training data mapping the solution to a future  $\Delta t$ . However, some cautionary remarks are warranted (see Appendix B.4): as with all DNN architectures, obtaining reliable large scaled time-steppers comes at the cost of significant training. Specifically, one usually needs to acquire large enough data sets to train the appropriate deep NN architecture. In the end, we show our proposed scheme is capable of learning long-term dependencies on some more realistic data sets, achieving state-of-the-art performance.

This work also suggests a number of open questions that motivate further investigation. In particular, nearly every sub-field within numerical integration can be revisited from this perspective. For example, bringing the idea of adaptive step sizes into this framework may potentially lead to even more efficient and accurate time-stepping schemes. Although this important element may be naturally addressed, it is not yet built in to the proposed methodology in its present form. In addition, as mentioned in Appendix B.4, the increments of the flow maps mostly exhibit obvious multiscale features. Since deep learning is essentially an interpolation method [77], to leverage more effective training, new sampling strategies may be proposed to address this problem [18, 78]. This is of crucial importance for deep learning, as neural networks are data-hungry and the curse of dimensionality makes it impossible to generate largely enough data sets for high dimensional systems. It is also important to introduce hierarchical uncertainty quantification mechanisms, as in the real world, we often have different levels of confidence across different time scales.

## 5 Acknowledgements

SLB acknowledges funding support from the Army Research Office (W911NF-19-1-0045); SLB and JNK acknowledge funding support from the Air Force Office of Scientific Research (FA9550-19-1-0386). JNK acknowledges support from the Air Force Office of Scientific Research (FA9550-17-1-0329).

Systems	Number of samples (train / validate / test)	Sampled region $\mathcal{D}$ in state space	Network architectures	Cross-validated NNTSs
Hyperbolic	1600/320/320	$[-1, 1]^2$	[2, 128, 128, 128, 2]	3 - 10
Cubic oscillator	3200/320/320	$[-1, 1]^2$	[2, 256, 256, 256, 2]	1 - 4
Van der Pol	3200/320/320	$[-2, 2] \times [-4, 4]$	[2, 512, 512, 512, 2]	2 - 5
Hopf bifurcation	3200/320/320	$[-0.2, 0.6] \times [-1, 2] \times [-1, 1]$	[3, 128, 128, 128, 3]	2 - 11
Lorenz	6400/640/640	$[-0.1, 0.1]^3$	[3, 1024, 1024, 1024, 3]	6 - 8

Table 2: Parameters and setups for the neural network time-steppers.

## A Methods and data

### A.1 Multiscale neural network HiTS

The setup of our multiscale neural network HiTS relies on a sequence of successfully trained, single time-scale neural network time-steppers. Here, we document the working details. For all systems, we first specify a domain of interest  $\mathcal{D}$  in the state space, over which we uniformly sample the initial states of training, validating, and testing trajectories. For all systems except for the Lorenz system, the time steps of the single time-scale neural network time-steppers are chosen to be 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12 and 10.24. Validating and testing trajectories are different from the training trajectories and they last 51.20 time units. For the Lorenz system, the process is slightly different. We simulate 3 long trajectories to form training, validating, and testing data sets (one for each). For each of these 3 trajectories, we cut off the initial 5 time units to make sure the collected data are on the attractor. 11 neural network time-steppers are trained for the Lorenz system as well; however, they have time steps of 0.0005, 0.001, 0.002, 0.004, 0.008, 0.016, 0.032, 0.064, 0.128, 0.256 and 0.512 due to the inherently chaotic dynamics. The validating and testing trajectories last 2.56 time units. For convenience, we term these single time-scale neural network time-steppers NNTS 0 - NNTS 10 within each individual experiment. In our experiments, all trajectory data are simulated with the `odeint` function provided by the **Scipy** package and are considered as the ground truth. Upon training, the number of forward steps  $p$  is set to 5, and we use the Python API for the **PyTorch** framework and the Adam optimizer with a learning rate of  $1e - 3$ . The training ends when the maximum epoch is reached (which is set to 100000) or when the mean squared error on one-step prediction is lower than  $1e - 8$ . Upon evaluation, we use the procedure presented in Section 2.3, coupling the cross-validated HiTSs to perform testing. The number of training/validating/testing samples, regions of interest, network architectures, and indices of cross validated neural network time-steppers are shown in Table 2.

### A.2 Hybrid time-steppers

We benchmark our proposed hybrid time-stepper against a commonly used numerical time-stepper, a fourth-order Runge-Kutta integrator with uniform step size. For the first four nonlinear system examples, we simulate with step sizes of 0.01, 0.02, 0.04, 0.08 and 0.16. For the Lorenz system, we shrink the sizes accordingly to 0.0005, 0.001, 0.002, 0.004 and 0.008. For convenience, we term them RK 0 - RK 4 for each individual experiment. For simplicity, our hybrid time-steppers are constructed to be the same Runge-Kutta time-steppers (i.e. RK 0 - RK 4) associated with one single time-scale neural-network time-stepper. The step size of this neural network time-stepper is set to be 0.512 for Lorenz and 10.24 for others. Similarly, we term them Hybrid 0 - Hybrid 4 for convenience.

<b>Sequences</b>	<b>HiTS</b>	<b>LSTM</b>	<b>ESN</b>	<b>CW-RNN</b>
<b>Fluid flow</b>	15388	15408	15400	15622
<b>Video frame</b>	31216	30976	31360	31560
<b>Music data</b>	1020416	1018368	1024000	1042328
<b>KS equation</b>	4069760	4091232	4070400	4076232

Table 3: Number of parameters for different architectures of the sequence generation experiments.

<b>Sequences</b>	<b>Network architectures in multiscale HiTS</b>
<b>Fluid flow</b>	[22, 256, 22], [22, 64, 22], [22, 16, 22], [22, 4, 22]
<b>Video frame</b>	[64, 128, 64], [64, 64, 64], [64, 32, 64], [64, 16, 64]
<b>Music data</b>	[128, 2048, 128], [128, 1024, 128], [128, 512, 128], [128, 256, 128], [128, 128, 128]
<b>KS equation</b>	[512, 2048, 512], [512, 1024, 512], [512, 512, 512], [512, 256, 512], [512, 128, 512]

Table 4: Network architectures in multiscale HiTS for sequence generation experiments.

### A.3 Details for sequence generation examples

In Section 3.2, we benchmark our multiscale HiTS against long short-term memory networks (LSTMs), echo state networks (ESNs), and clockwork recurrent neural networks (CW-RNNs) on sequence generation tasks over four data sets: the KuramotoSivashinsky (KS) equation, a music snippet from Fugue No. 1 In C Major, BWV 846, an animation of fluid flow passing a cylinder, and a video frame of blooming flowers.

Information for these data sets and preprocessing steps are as follows:

- For the KS equation, 4001 snapshots on 512 evenly distributed spatial points over the interval  $(0, 16\pi)$  are simulated with a spectral method. These data are used to train different network architectures.
- For the music data, we sample at 1102 Hz for 2 seconds. Then, a time-delay embedding with 128 delays is applied to produce a richer feature space, resulting in a training tensor of size  $128 \times 2077$ .
- For the simulated fluid flow example, data is generated using the immersed boundary projection method (IBPM) [74, 75] at a Reynolds number of 100; details on the numerical simulation may be found in [79]. We apply principal component analysis (PCA) to the streamwise velocity field  $u$ , and use the dynamics on the first 22 modes to train the neural networks.
- The video frame of blooming flowers consists of 175 image snapshots of  $540 \times 960$  pixels and 3 channels (RGB). Similar to the fluid flow data, we preprocess it by applying PCA and work with its reduced representation in the first 64 PC dimensions.

For each sequence, we setup different types of neural network architectures so that they have about the same number of trainable parameters; see Table 3 for the summary of number of network parameters. The specific procedures are as follows:

- We first set up our multiscale neural network time-steppers. For the fluid flow and video frame examples, we use 4 neural network time-steppers with step sizes 1, 4, 16, and 64, respectively. For the music data, we use 5 time-steppers and the corresponding step sizes are 1, 5, 25, 125, and 625. For the KS example, we use 5 time-steppers with step sizes of 1, 6, 36, 216, and 1296. For each individual example, one step size equals to the time interval between two adjacent snapshots. For simplicity, all HiTSs only have one hidden layer, and detailed network architectures are shown in Table 4.
- Once the multiscale HiTSs are trained for each data set, we calculate the total number of parameters and use this information to create the other three types of architectures by carefully choosing the number of hidden units.
- For the CW-RNN, it has two other hyper-parameters: the number of internal modules and their associated clock rates. We set these parameters to match those in multiscale HiTSs for fair comparison.

Systems	Hyperbolic	Cubic oscillator	Van der Pol	Hopf bifurcation	Lorenz
NNTS 0	10.06s	24.74s	48.95s	9.98s	42.98s
NNTS 1	3.75s	7.68s	21.48s	3.69s	19.01s
NNTS 2	1.89s	3.62s	8.74s	1.87s	8.51s
NNTS 3	1.01s	1.84s	4.00s	1.03s	4.10s
NNTS 4	0.56s	1.00s	2.06s	0.59s	2.03s
NNTS 5	0.34s	0.77s	1.09s	0.37s	1.09s
NNTS 6	0.24s	0.65s	0.60s	0.27s	0.56s
NNTS 7	0.18s	0.39s	0.36s	0.22s	0.30s
NNTS 8	0.15s	0.27s	0.24s	0.21s	0.18s
NNTS 9	0.13s	0.17s	0.19s	0.17s	0.12s
NNTS 10	0.12s	0.14s	0.15s	0.17s	0.09s
multiscale	0.89s	6.73s	7.96s	1.84s	0.73s

Table 5: Computation time for multiscale and all single time-scale neural network time-steppers.

## B Additional results

### B.1 Computation time for neural network time-steppers

As a supplement to Fig. 6A, Table 5 shows the computational time for all single time-scale neural network time-steppers along with the timings for our multiscale time-steppers. It is not surprising to see that the computation accelerates as the step size grows. The multiscale scheme is more efficient than the finest scale neural network time-stepper in use (see Table 2 for the coupled NNTSs), which suggests the benefit of vectorized computation.

### B.2 Noisy measurements

In Table 6, Table 7, Table 8, and Fig. 8, we study the accuracy of neural network time-steppers with different temporal gaps and the robustness of our results with respect to noise in the observations of the system states. Gaussian random noise with different variances are independently applied to each component of the dynamics. The variances are set to be 0% (noise free), 1%, and 2% of the variance of that component averaged over all trajectories across the data sets. We observe that larger noise corruption levels generally lead to inferior accuracy, but our multiscale scheme consistently works better than any single time-scale NNTSs. Similar to the results shown in [30], neural networks with larger temporal gaps tend to show more robustness and the optimal value of the temporal gap is usually problem-dependent, indicating the temporal gap is a crucial parameter for data-driven modeling. By leveraging our proposed multiscale coupling strategy, it is automatically taken into account.

### B.3 More benchmark results against Runge-Kutta time-steppers

Fig. 9 shows the mean squared errors of different schemes versus simulation time. Again, they are plotted in a base-10 logarithmic scale. Our multiscale schemes provide competitive results compared to the Runge-Kutta schemes in terms of accuracy. This is especially notable for the purely data-driven, neural network-based multiscale time-stepping scheme: the true model of the underlying dynamics is unavailable, which is in sharp contrast to the classical simulation setting. In other words, neural network time-steppers can identify the dynamics and generate accurate predictions at the same time. Hybrid time-steppers, on the other hand, also provide competitive accuracy. And the accuracy level is dominated by either the numerical time-stepper or the neural network time-stepper, whichever is worse. Therefore, the hybrid time-steppers usually bear a slightly lower accuracy than the purely numerical time-steppers.

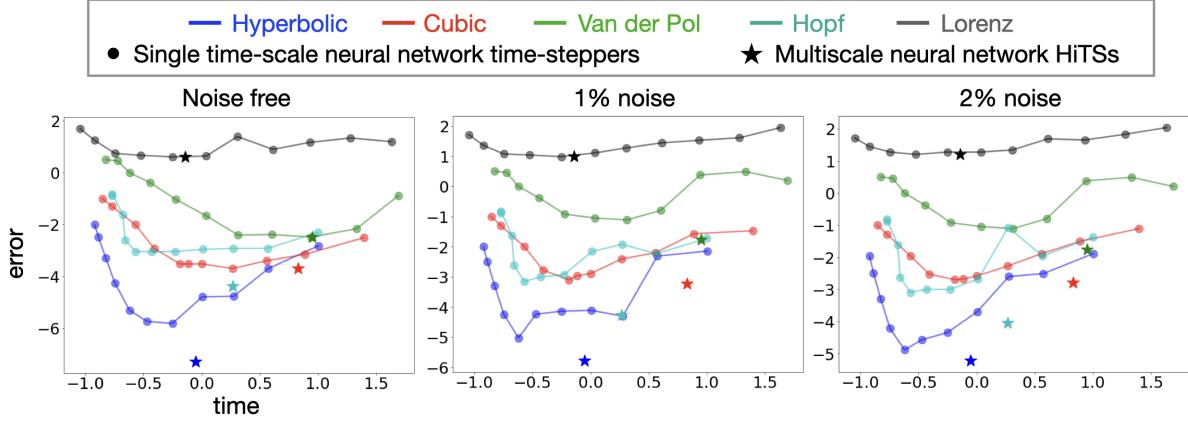


Figure 8: Accuracy and efficiency trade-offs under 0%, 1% and 2% noise corruptions

Systems	Hyperbolic	Cubic oscillator	Van der Pol	Hopf bifurcation	Lorenz
NNTS 0	$1.5e - 3$	$3.1e - 3$	$1.3e - 1$	$4.9e - 3$	$1.6e + 1$
NNTS 1	$2.0e - 4$	$7.0e - 4$	$6.9e - 3$	$1.2e - 3$	$2.2e + 1$
NNTS 2	$1.7e - 5$	$4.0e - 4$	$3.3e - 3$	$1.2e - 3$	$1.5e + 1$
NNTS 3	$1.6e - 5$	$2.0e - 4$	$4.1e - 3$	$1.1e - 3$	$8.0e + 0$
NNTS 4	$1.5e - 6$	$3.0e - 4$	$3.9e - 3$	$9.0e - 4$	$2.5e + 1$
NNTS 5	$1.8e - 6$	$3.0e - 4$	$2.2e - 2$	$9.0e - 4$	$4.4e + 0$
NNTS 6	$4.8e - 6$	$3.0e - 4$	$9.3e - 2$	$9.0e - 4$	$4.1e + 0$
NNTS 7	$5.4e - 5$	$1.2e - 3$	$4.2e - 1$	$2.5e - 3$	$4.7e + 0$
NNTS 8	$5.0e - 4$	$9.8e - 3$	$1.0e + 0$	$2.4e - 2$	$5.6e + 0$
NNTS 9	$3.2e - 3$	$5.0e - 2$	$2.9e + 0$	$1.5e - 1$	$1.8e + 1$
NNTS 10	$1.0e - 2$	$1.0e - 1$	$3.2e + 0$	$1.3e - 1$	$5.1e + 1$
multiscale	$5.1e - 8$	$2.0e - 4$	$3.2e - 3$	$4.1e - 5$	$4.1e + 0$

Table 6: Integrated  $\mathcal{L}_2$  between the predicted and the exact measurement with noise free data.

Systems	Hyperbolic	Cubic oscillator	Van der Pol	Hopf bifurcation	Lorenz
NNTS 0	$7.2e - 3$	$3.4e - 2$	$1.6e + 0$	$1.9e - 2$	$9.0e + 1$
NNTS 1	$4.9e - 3$	$2.7e - 2$	$3.1e + 0$	$5.9e - 3$	$4.1e + 1$
NNTS 2	$5.0e - 4$	$6.4e - 3$	$2.4e + 0$	$1.2e - 2$	$3.4e + 1$
NNTS 3	$7.8e - 5$	$3.9e - 3$	$1.6e - 1$	$7.0e - 3$	$2.8e + 1$
NNTS 4	$7.3e - 5$	$1.3e - 3$	$7.8e - 2$	$1.2e - 3$	$1.9e + 1$
NNTS 5	$5.8e - 5$	$1.1e - 3$	$9.0e - 2$	$1.0e - 3$	$1.3e + 1$
NNTS 6	$9.5e - 6$	$8.0e - 4$	$1.2e - 1$	$7.0e - 4$	$9.7e + 0$
NNTS 7	$5.7e - 5$	$1.7e - 3$	$4.2e - 1$	$2.4e - 3$	$1.1e + 1$
NNTS 8	$5.0e - 4$	$1.0e - 2$	$1.0e + 0$	$2.4e - 2$	$1.2e + 1$
NNTS 9	$3.2e - 3$	$5.0e - 2$	$2.9e + 0$	$1.5e - 1$	$2.3e + 1$
NNTS 10	$1.0e - 2$	$1.0e - 1$	$3.2e + 0$	$1.3e - 1$	$5.2e + 1$
multiscale	$1.7e - 6$	$6.0e - 4$	$1.7e - 2$	$5.4e - 5$	$9.8e + 0$

Table 7: Integrated  $\mathcal{L}_2$  between the predicted and the exact measurement with 1% Gaussian noise.

Systems	Hyperbolic	Cubic oscillator	Van der Pol	Hopf bifurcation	Lorenz
NNTS 0	$1.3e - 2$	$7.8e - 2$	$1.6e + 0$	$4.2e - 2$	$1.1e + 2$
NNTS 1	$3.1e - 3$	$3.1e - 2$	$3.1e + 0$	$1.1e - 2$	$6.8e + 1$
NNTS 2	$2.5e - 3$	$1.3e - 2$	$2.4e + 0$	$8.5e - 2$	$4.5e + 1$
NNTS 3	$2.0e - 4$	$5.3e - 3$	$1.6e - 1$	$2.1e - 3$	$4.9e + 1$
NNTS 4	$4.5e - 5$	$2.6e - 3$	$7.8e - 2$	$1.0e - 3$	$2.2e + 1$
NNTS 5	$2.7e - 5$	$2.1e - 3$	$9.0e - 2$	$1.0e - 3$	$1.9e + 1$
NNTS 6	$1.3e - 5$	$2.0e - 3$	$1.2e - 1$	$8.0e - 4$	$1.9e + 1$
NNTS 7	$6.1e - 5$	$3.0e - 3$	$4.2e - 1$	$2.4e - 3$	$1.6e + 1$
NNTS 8	$5.0e - 4$	$1.1e - 2$	$1.0e + 0$	$2.4e - 2$	$1.9e + 1$
NNTS 9	$3.2e - 3$	$5.2e - 2$	$2.9e + 0$	$1.5e - 1$	$2.8e + 1$
NNTS 10	$1.1e - 2$	$1.0e - 1$	$3.2e + 0$	$1.3e - 1$	$5.2e + 1$
multiscale	$6.1e - 6$	$1.6e - 3$	$1.7e - 2$	$9.0e - 5$	$1.6e + 1$

Table 8: Integrated  $\mathcal{L}_2$  between the predicted and the exact measurement with 2% Gaussian noise.

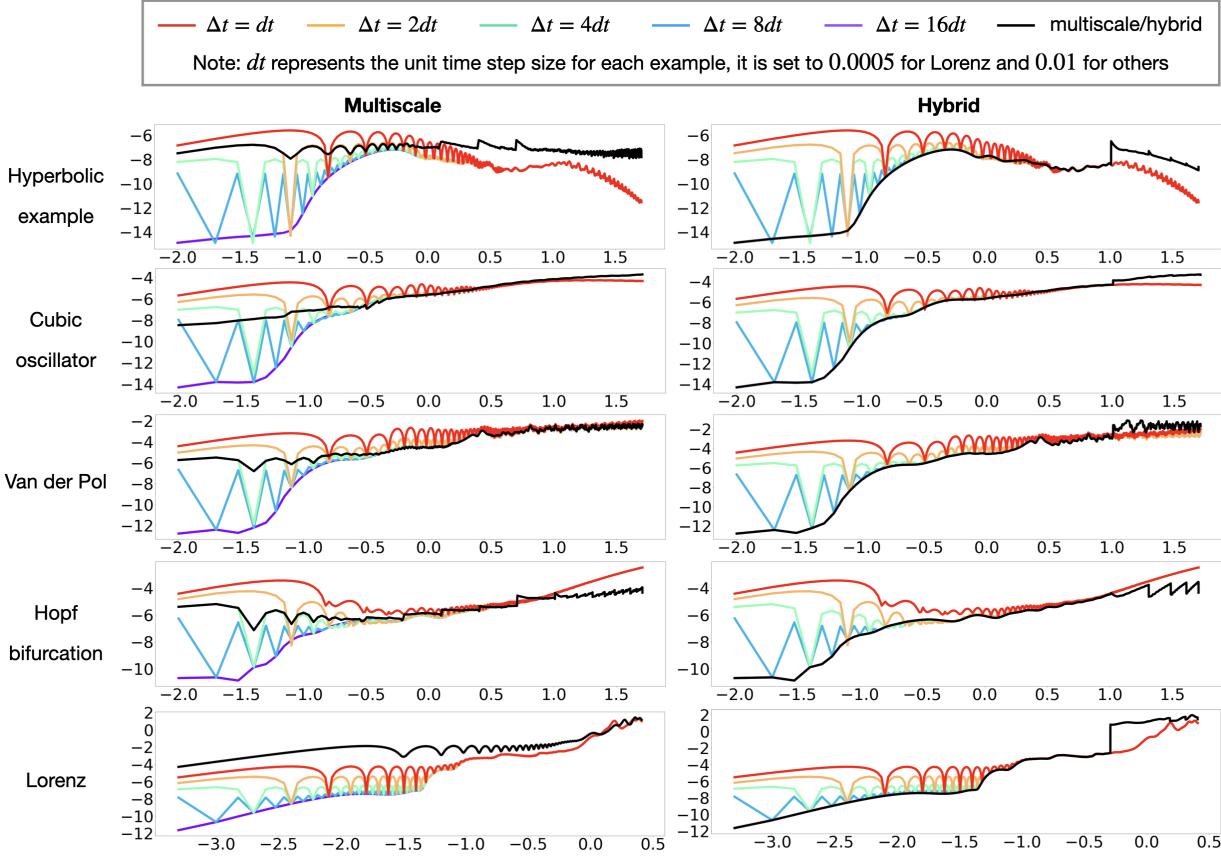
Systems	Hyperbolic	Cubic oscillator	Van der Pol	Hopf bifurcation	Lorenz
RK 0	$2.55s$	$3.18s$	$2.82s$	$3.93s$	$3.87s$
RK 1	$1.28s$	$1.60s$	$1.30s$	$2.21s$	$1.97s$
RK 2	$0.74s$	$1.01s$	$0.78s$	$1.07s$	$1.17s$
RK 3	$0.46s$	$0.64s$	$0.42s$	$0.60s$	$0.98s$
RK 4	$0.30s$	$0.43s$	$0.30s$	$0.35s$	$0.60s$
Hybrid 0	$1.36s$	$2.25s$	$1.37s$	$1.89s$	$3.17s$
Hybrid 1	$0.72s$	$0.96s$	$0.69s$	$1.01s$	$1.78s$
Hybrid 2	$0.37s$	$0.55s$	$0.42s$	$0.59s$	$0.99s$
Hybrid 3	$0.24s$	$0.30s$	$0.29s$	$0.37s$	$0.74s$
Hybrid 4	$0.19s$	$0.23s$	$0.22s$	$0.26s$	$0.62s$
multiscale	$0.89s$	$6.73s$	$7.96s$	$1.84s$	$0.73s$

Table 9: Computation time of various schemes on testing trajectories.

Table 9 shows the computational times of the various algorithms which is a supplement for Fig. 6. For Runge-Kutta time-steppers, computations speed up as the time step increases. Both our multiscale neural network hierarchical time-stepping scheme and hybrid time-stepping scheme provide efficient computations. In particular, the hybrid time-steppers outperform the corresponding RK integrators, as the vectorized computations accelerate these simulations. The hybrid time-steppers generally beat the multiscale neural network time-steppers in terms of the speed. This speed-up may stem from the computation on each individual step: evaluating a large neural network model (forward propagation) requires more computational effort than applying the Runge-Kutta scheme, which only involves four evaluations of the vector field.

#### B.4 Remarks on learning flow maps

Though there is evidence that our schemes can boost the performance of neural network time-steppers, as well as classical numerical simulation algorithms, each individual result heavily depends on the system under study. Theoretically, there is nothing preventing us from fitting the flow maps with arbitrarily long time periods; however, this is not always possible. For example, in the Lorenz system, one can clearly see the performance of NNTS 10 is unsatisfactory: the one-step prediction error is on the order of  $\mathcal{O}(1)$ . The reason could either be that the current architecture is not large enough to encode the map, or the training data set is not large enough to fully capture the complexity of the corresponding flow map. Chaotic dynamics will



**Figure 9: Stepwise error comparisons with Runge-Kutta integration.** For each plot, the horizontal axis represents simulation time and vertical axis represents  $\mathcal{L}_2$  error. Both axes are visualized in base-10 logarithmic scale. The left column shows the comparisons between multiscale time-steppers and Runge-Kutta time-steppers. The right column shows the comparisons between hybrid time-steppers and Runge-Kutta time-steppers. For both columns, mean squared errors at each time step are plotted.

generally present this challenge, as the flow map complexity grows exponentially with time. As presented in Section 2.2, we are essentially fitting the  $\Delta t$ -lag flow map minus the identity with the neural network

$$\hat{\mathbf{F}}(\mathbf{x}_t, \Delta t) \approx \mathbf{x}_{t+\Delta t} - \mathbf{x}_t. \quad (10)$$

To see what these increments look like, for the first four nonlinear systems, we have visualized  $\mathbf{x}_{j\Delta t} - \mathbf{x}_0$ , for  $j = 1, 2, \dots, 64$  with  $\mathbf{x}'_0$ s uniformly sampled from their associated region on interest  $\mathcal{D}$ , shown in Table 2. As for the Lorenz system, the process is similar but the initial states are sampled from the region  $[-9, -7] \times [6, 8] \times \{27\}$  because this region is close to the strange attractor, and  $\Delta t$  is set to 0.008 for Lorenz and 0.16 for others. Results are shown in Fig. 10.

In the examples of hyperbolic fixed point and Hopf bifurcation, the plots do not visually grow more complex as time proceeds. For the Van der Pol oscillator, complexity grows in the beginning stage but later it tends to maintain that level of complexity. For the Cubic oscillator example, the complexity keeps growing with time; however, there are clear structures, as more spirals are generated. The most interesting case is the Lorenz system: not only does the complexity grow, but earlier stripe patterns (period doubling) also disappear as time proceeds, leading to tremendous difficulties in training. To fully capture this growing complexity, one may need an exponentially growing data set and network architecture. In addition, one seemingly common feature from all plots is that the multiscale effects become more pronounced as time proceeds, leading to the emergence of patterns. Whether we can utilize this feature to perform more effective training may be a promising direction that motivates future work.

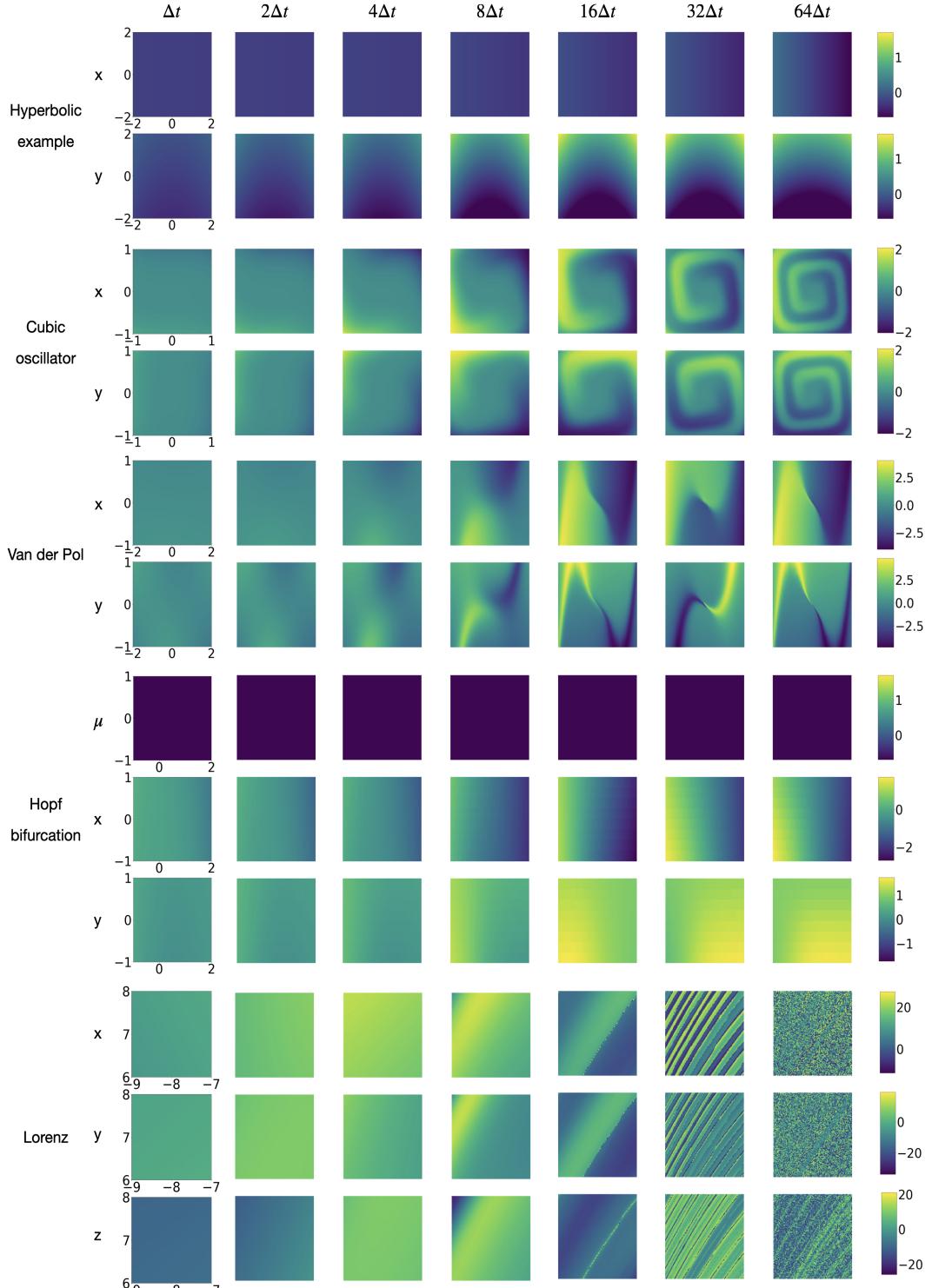


Figure 10: **Increments of flow maps.** We visualize  $\mathbf{x}_{j\Delta t} - \mathbf{x}_0$ , for  $j = 1, 2, \dots, 64$  where  $\mathbf{x}$  is the state of each example. For the Lorenz system, we set  $\Delta t = 0.008$  and the region we visualize is  $[-9, -7] \times [6, 8] \times \{27\}$  whereas for other examples,  $\Delta t = 0.16$  and the regions are in Table 2.

## References

- [1] F. B. Hildebrand, *Introduction to numerical analysis*. Courier Corporation, 1987.
- [2] S. D. Conte and C. De Boor, *Elementary numerical analysis: an algorithmic approach*. SIAM, 2017.
- [3] J. N. Kutz, *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013.
- [4] W. H. Enright, T. Hull, and B. Lindberg, “Comparing numerical methods for stiff systems of ode: s,” *BIT Numerical Mathematics*, vol. 15, no. 1, pp. 10–48, 1975.
- [5] G. D. Byrne and A. C. Hindmarsh, “Stiff ode solvers: A review of current and coming attractions,” *Journal of Computational physics*, vol. 70, no. 1, pp. 1–62, 1987.
- [6] E. J. Parish and K. T. Carlberg, “Time-series machine-learning error models for approximate solutions to parameterized dynamical systems,” *Computer Methods in Applied Mechanics and Engineering*, vol. 365, p. 112990, 2020.
- [7] F. Regazzoni, L. Dedè, and A. Quarteroni, “Machine learning for fast and reliable solution of time-dependent differential equations,” *Journal of Computational Physics*, vol. 397, p. 108852, 2019.
- [8] T. Qin, K. Wu, and D. Xiu, “Data driven governing equations approximation using deep neural networks,” *Journal of Computational Physics*, vol. 395, pp. 620–635, 2019.
- [9] H. Lange, S. L. Brunton, and N. Kutz, “From fourier to koopman: Spectral methods for long-term time series prediction,” *arXiv preprint arXiv:2004.00574*, 2020.
- [10] L. Ying and E. J. Candès, “The phase flow method,” *Journal of Computational Physics*, vol. 220, pp. 184–215, 2006.
- [11] S. L. Brunton and C. W. Rowley, “Fast computation of FTLE fields for unsteady flows: a comparison of methods,” *Chaos*, vol. 20, p. 017503, 2010.
- [12] S. F. McCormick, *Multigrid methods*. SIAM, 1987.
- [13] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*. Elsevier, 2000.
- [14] J. Guckenheimer and P. Holmes, *Nonlinear oscillations, dynamical systems, and bifurcations of vector fields*. Springer Science & Business Media, 2013, vol. 42.
- [15] S. Wiggins, *Introduction to applied nonlinear dynamical systems and chaos*. Springer Science & Business Media, 2003, vol. 2.
- [16] D. M. Luchtenburg, S. L. Brunton, and C. W. Rowley, “Long-time uncertainty propagation using generalized polynomial chaos and flow map composition,” *Journal of Computational Physics*, vol. 274, pp. 783–802, 2014.
- [17] J. J. Bramburger and J. N. Kutz, “Poincaré maps for multiscale physics discovery and nonlinear floquet theory,” *Physica D: Nonlinear Phenomena*, p. 132479, 2020.
- [18] J. J. Bramburger, D. Dylewsky, and J. N. Kutz, “Sparse identification of slow timescale dynamics,” *arXiv preprint arXiv:2006.00940*, 2020.
- [19] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
- [20] R. Gonzalez-Garcia, R. Rico-Martinez, and I. Kevrekidis, “Identification of distributed parameter systems: A neural net based approach,” *Computers & chemical engineering*, vol. 22, pp. S965–S968, 1998.
- [21] M. Milano and P. Koumoutsakos, “Neural network modeling for near wall turbulent flow,” *Journal of Computational Physics*, vol. 182, no. 1, pp. 1–26, 2002.
- [22] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.
- [24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [25] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler, “Joint training of a convolutional network and a graphical model for human pose estimation,” in *Advances in neural information processing systems*, 2014, pp. 1799–1807.
- [26] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Černocký, “Strategies for training large scale neural network language models,” in *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*. IEEE, 2011, pp. 196–201.
- [27] A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath *et al.*, “Deep neural networks for acoustic modeling in speech recognition,” *IEEE Signal processing magazine*, 2012.
- [28] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, “Natural language

- processing (almost) from scratch,” *Journal of machine learning research*, vol. 12, no. Aug, pp. 2493–2537, 2011.
- [29] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Advances in neural information processing systems*, 2014, pp. 3104–3112.
  - [30] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Multistep neural networks for data-driven discovery of nonlinear dynamical systems,” *arXiv preprint arXiv:1801.01236*, 2018.
  - [31] S. H. Rudy, J. N. Kutz, and S. L. Brunton, “Deep learning of dynamics and signal-noise decomposition with time-stepping constraints,” *Journal of Computational Physics*, vol. 396, pp. 483–506, 2019.
  - [32] S. Pan and K. Duraisamy, “Data-driven discovery of closure models,” *SIAM Journal on Applied Dynamical Systems*, vol. 17, no. 4, pp. 2381–2413, 2018.
  - [33] Z. Y. Wan, P. Vlachas, P. Koumoutsakos, and T. Sapsis, “Data-assisted reduced-order modeling of extreme events in complex dynamical systems,” *PloS one*, vol. 13, no. 5, 2018.
  - [34] J.-H. Jacobsen, E. Oyallon, S. Mallat, and A. W. Smeulders, “Multiscale hierarchical convolutional networks,” *arXiv preprint arXiv:1703.04140*, 2017.
  - [35] C. Wehmeyer and F. Noé, “Time-lagged autoencoders: Deep learning of slow collective variables for molecular kinetics,” *The Journal of chemical physics*, vol. 148, no. 24, p. 241703, 2018.
  - [36] C. C. Douglas, “Mgnet: a multigrid and domain decomposition network,” *ACM SIGNUM Newsletter*, vol. 27, no. 4, pp. 2–8, 1992.
  - [37] Y. Liu, C. Ponce, S. L. Brunton, and J. N. Kutz, “Multiresolution convolutional autoencoders,” *arXiv preprint arXiv:2004.04946*, 2020.
  - [38] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations,” *arXiv preprint arXiv:1711.10561*, 2017.
  - [39] Y. Bar-Sinai, S. Hoyer, J. Hickey, and M. P. Brenner, “Learning data-driven discretizations for partial differential equations,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 31, pp. 15 344–15 349, 2019.
  - [40] V. Sitzmann, J. N. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, “Implicit neural representations with periodic activation functions,” *arXiv preprint arXiv:2006.09661*, 2020.
  - [41] B. Lusch, J. N. Kutz, and S. L. Brunton, “Deep learning for universal linear embeddings of nonlinear dynamics,” *Nature communications*, vol. 9, no. 1, pp. 1–10, 2018.
  - [42] K. Champion, B. Lusch, J. N. Kutz, and S. L. Brunton, “Data-driven discovery of coordinates and governing equations,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 45, pp. 22 445–22 451, 2019.
  - [43] J. Pathak, Z. Lu, B. R. Hunt, M. Girvan, and E. Ott, “Using machine learning to replicate chaotic attractors and calculate lyapunov exponents from data,” *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 27, no. 12, p. 121102, 2017.
  - [44] Z. Lu, B. R. Hunt, and E. Ott, “Attractor reconstruction by machine learning,” *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 28, no. 6, p. 061104, 2018.
  - [45] S. Pan and K. Duraisamy, “Long-time predictive modeling of nonlinear dynamical systems using neural networks,” *Complexity*, vol. 2018, 2018.
  - [46] J. Pathak, A. Wikner, R. Fussell, S. Chandra, B. R. Hunt, M. Girvan, and E. Ott, “Hybrid forecasting of chaotic processes: Using machine learning in conjunction with a knowledge-based model,” *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 28, no. 4, p. 041101, 2018.
  - [47] P. R. Vlachas, W. Byeon, Z. Y. Wan, T. P. Sapsis, and P. Koumoutsakos, “Data-driven forecasting of high-dimensional chaotic systems with long short-term memory networks,” *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 474, no. 2213, p. 20170844, 2018.
  - [48] S. Wiewel, M. Becher, and N. Thuerey, “Latent space physics: Towards learning the temporal evolution of fluid flow,” in *Computer Graphics Forum*, vol. 38. Wiley Online Library, 2019, pp. 71–82.
  - [49] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.
  - [50] E. Weinan, “A proposal on machine learning via dynamical systems,” *Communications in Mathematics and Statistics*, vol. 5, no. 1, pp. 1–11, 2017.
  - [51] E. Weinan, J. Han, and Q. Li, “A mean-field optimal control formulation of deep learning,” *Research in the Mathematical Sciences*, vol. 6, no. 1, p. 10, 2019.
  - [52] C. Ma, L. Wu *et al.*, “Machine learning from a continuous viewpoint,” *arXiv preprint arXiv:1912.12777*, 2019.
  - [53] B. Chang, L. Meng, E. Haber, F. Tung, and D. Begert, “Multi-level residual networks from dynamical

- systems view,” *arXiv preprint arXiv:1710.10348*, 2017.
- [54] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, “Neural ordinary differential equations,” in *Advances in neural information processing systems*, 2018, pp. 6571–6583.
  - [55] E. Haber and L. Ruthotto, “Stable architectures for deep neural networks,” *Inverse Problems*, vol. 34, no. 1, p. 014004, 2017.
  - [56] B. Poole, S. Lahiri, M. Raghu, J. Sohl-Dickstein, and S. Ganguli, “Exponential expressivity in deep neural networks through transient chaos,” in *Advances in neural information processing systems*, 2016, pp. 3360–3368.
  - [57] S. Hochreiter, “Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München,” 1991.
  - [58] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.
  - [59] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
  - [60] H. Jaeger, *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the “echo state network” approach*. GMD-Forschungszentrum Informationstechnik Bonn, 2002, vol. 5.
  - [61] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *International conference on machine learning*, 2013, pp. 1310–1318.
  - [62] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
  - [63] W. E and J. Lu, “Multiscale modeling,” *Scholarpedia*, vol. 6, no. 10, p. 11527, 2011, revision #91540.
  - [64] E. Weinan and B. Engquist, “Multiscale modeling and computation,” *Notices of the AMS*, vol. 50, no. 9, pp. 1062–1070, 2003.
  - [65] A. Brandt, “Multi-level adaptive solutions to boundary-value problems,” *Mathematics of computation*, vol. 31, no. 138, pp. 333–390, 1977.
  - [66] L. Greengard and V. Rokhlin, “A fast algorithm for particle simulations,” *Journal of Computational Physics*, vol. 135, no. 2, pp. 280–292, 1997.
  - [67] M. J. Berger, P. Colella *et al.*, “Local adaptive mesh refinement for shock hydrodynamics,” *Journal of computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.
  - [68] A. Toselli and O. Widlund, *Domain decomposition methods-algorithms and theory*. Springer Science & Business Media, 2006, vol. 34.
  - [69] I. Daubechies, *Ten lectures on wavelets*. Siam, 1992, vol. 61.
  - [70] J. N. Kutz, X. Fu, and S. L. Brunton, “Multiresolution dynamic mode decomposition,” *SIAM Journal on Applied Dynamical Systems*, vol. 15, no. 2, pp. 713–735, 2016.
  - [71] E. Weinan, *Principles of multiscale modeling*. Cambridge University Press, 2011.
  - [72] E. Weinan, B. Engquist, X. Li, W. Ren, and E. Vanden-Eijnden, “Heterogeneous multiscale methods: a review,” *Communications in computational physics*, vol. 2, no. 3, pp. 367–450, 2007.
  - [73] I. G. Kevrekidis, C. W. Gear, J. M. Hyman, P. G. Kevrekidis, O. Runborg, C. Theodoropoulos *et al.*, “Equation-free, coarse-grained multiscale computation: Enabling macroscopic simulators to perform system-level analysis,” *Communications in Mathematical Sciences*, vol. 1, no. 4, pp. 715–762, 2003.
  - [74] K. Taira and T. Colonius, “The immersed boundary method: a projection approach.” *Journal of Computational Physics*, vol. 225, no. 2, pp. 2118–2137, 2007.
  - [75] T. Colonius and K. Taira, “A fast immersed boundary method using a nullspace approach and multi-domain far-field boundary conditions,” *Computer Methods in Applied Mechanics and Engineering*, vol. 197, pp. 2131–2146, 2008.
  - [76] J. Pathak, B. Hunt, M. Girvan, Z. Lu, and E. Ott, “Model-free prediction of large spatiotemporally chaotic systems from data: A reservoir computing approach,” *Physical review letters*, vol. 120, no. 2, p. 024102, 2018.
  - [77] S. Mallat, “Understanding deep convolutional networks,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150203, 2016.
  - [78] D. Dylewsky, M. Tao, and J. N. Kutz, “Dynamic mode decomposition for multiscale nonlinear physics,” *Physical Review E*, vol. 99, no. 6, p. 063311, 2019.
  - [79] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.