When Differentiable Programming Meets Spectral PDE Solver

Qijia Jiang
Department of Statistics
UC Davis
`qjang@ucdavis.edu`

Abstract

We aim to combine data and physics for designing more accurate and faster PDE solvers. We reinterpret the data-driven machine learning approach of [16] through a dynamical system perspective and draw a connection to neural ODE and implicit layer neural network architectures. These in turn inspire a class of sample-efficient spectral PDE solvers (with an encoder - processor - decoder structure) that can be trained end-to-end in a memory-efficient way. The crucial benefit of the methods is that they are resolution-invariant and guaranteed to be consistent.

1 General Framework: Dynamical system perspective

We begin by giving a systematic perspective for [16] that operates in physical space, and will use the linear advection equation

$$\partial_t u(t,x) + c\nabla u(t,x) = 0, u(0,x) = u_0(x) \text{ for } (x,t) \in (0,1)\times(0,T), \tag{1}$$

as a running example. The dynamical system perspective entails that we try to learn the tunable controls to drive the system to state u^T at final time when initialized from state u^0, using training data from solved PDEs with a high-fidelity classical method.

Re-interpretation of the simplest set-up: On (1), the proposed three-point finite difference scheme takes the following form (with fixed u_1^n, u_J^n derived from periodic boundary condition):

$$\frac{u_j^{n+1}-u_j^n}{\Delta t} = \frac{c(1-g)}{\Delta x}(b_{-1}u_{j-1}^{n+1} + b_0 u_j^{n+1} + b_1 u_{j+1}^{n+1}) + \frac{cg}{\Delta x}(b_{-1}u_{j-1}^n + b_0 u_j^n + b_1 u_{j+1}^n)$$

where $u_j^n \approx U(x_j, t^n)$. With $b_0 = -1-2b_{-1}$ and $b_1 = 1 + b_{-1}$, g arbitrary, the scheme can be proved to be consistent and conservative $\sum_j u_j^{n+1} = \sum_j u_j^n$ [16]. Several fixed choices of g, b_{-1} in fact recover classical numerical methods such as Backward Euler and Crank-Nicolson. The machine-learned approach seeks to optimize over I data pairs $\{(u^{0,i}, u_{ref}^{N,i}) \in \mathbb{R}^J \times \mathbb{R}^J\}_{i=1}^I$ a loss of low-dimensional parameters:

$$E(g, b_{-1}) := \frac{\Delta x}{2}\sum_{i=1}^{I}\sum_{j=1}^{J} |u_j^{N,i}(g,b_{-1}) - u_{j,ref}^{N,i}|^2 \,. \tag{2}$$

Note that the parameters g, b_{-1} are shared across time and the loss only depends on the final time. If we substitute in the requirement on b_0, b_1, one ends up with for $j \in [J]$

$$u_j^{n+1} = u_j^n + \frac{c\Delta t b_{-1}}{\Delta x}[(1-g)u_{j-1}^{n+1} + g u_{j-1}^n] - \frac{c\Delta t(1+2b_{-1})}{\Delta x}[(1-g)u_j^{n+1} + g u_j^n] + \frac{c\Delta t(1+b_{-1})}{\Delta x}[(1-g)u_{j+1}^{n+1} + g u_{j+1}^n]$$

or

$$u_j^{n+1} + \frac{c\Delta t}{\Delta x}[(1+2b_{-1})(1-g)u_j^{n+1} - b_{-1}(1-g)u_{j-1}^{n+1} - (1+b_{-1})(1-g)u_{j+1}^{n+1}]$$

$$= u_j^n + \frac{c\Delta t}{\Delta x}[-(1+2b_{-1})gu_j^n + b_{-1}gu_{j-1}^n + (1+b_{-1})gu_{j+1}^n] \, ,$$

which means if we put all the J grid points into a vector u^n, u^{n+1}, we have for tri-diagonal A_θ, B_θ

$$u^{n+1} + \frac{\Delta t}{\Delta x}A_\theta u^{n+1} = u^n + \frac{\Delta t}{\Delta x}B_\theta u^n \tag{3}$$

where $\theta = \{b_{-1}, g\}$ is time-independent and $\Delta x, \Delta t$ are fixed a-priori. We recognize it as a dynamical system update in $\mathbb{R}^J$ from time t^n to t^{n+1}, and can be approximated as a continuous update

$$\lim_{\Delta t \to 0} \frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{\Delta x} \lim_{\Delta t \to 0} (B_\theta u^n - A_\theta u^{n+1})$$

giving for $u^t \in \mathbb{R}^J, t \in (0, T)$,

$$\partial_t u^t = \frac{1}{\Delta x}(B_\theta - A_\theta)u^t \, . \tag{4}$$

One would recognize this as a Neural ODE [7] evolving a J-dimensional vector: $\partial_t u(t) = f(u(t), t, \theta)$, where we learn the dynamics (parameterized by θ) that match the ground truth solution at time T. This can be thought of as an interpolation of the discrete dynamics, but the dependence on g is gone when taking the $\Delta t \to 0$ limit in (4). The nice thing about the neural ODE formulation is that one can solve an augmented ODE through adjoint method backward in time to efficient propagate the gradients needed for updating θ [7]: (let $a(t) := \frac{\partial E(u^T)}{\partial u^t}$)

$$\frac{\partial E(u^T;\theta)}{\partial \theta} = -\int_T^0 a(t)^\top \frac{\partial f(u(t), \theta)}{\partial \theta} \, dt \, , \quad \text{where } \frac{\partial a(t)}{\partial t} = -a(t)^\top \frac{\partial f(u(t), \theta)}{\partial u} \, . \tag{5}$$

General discrete dynamical system: More generally, the system can be modeled as a sequence of N implicit layers with *time-dependent* θ^n (see Example 1 below), since from (3) we are looking for

$$[I + \frac{\Delta t}{\Delta x}A_\theta^n]u^{n+1} - [I + \frac{\Delta t}{\Delta x}B_\theta^n]u^n = 0, \quad \forall n \in [N] \, . \tag{6}$$

Each implicit layer can be described as finding u^{n+1} such that $f(u^n, u^{n+1}(u^n, \theta^{n+1}), \theta^{n+1}) = 0$ [1], for which we can use *implicit function theorem* to directly compute gradient at the solution u_{n+1}^*:

$$\partial_{u^n} f + \partial_{u^{n+1}} f \frac{\partial u^{n+1}}{\partial u^n} = 0 \Rightarrow \frac{\partial u^{n+1}}{\partial u^n} = -[\partial_{u^{n+1}} f]^{-1} \partial_{u^n} f$$

$$\partial_{\theta^{n+1}} f + \partial_{u^{n+1}} f \frac{\partial u^{n+1}}{\partial \theta^{n+1}} = 0 \Rightarrow \frac{\partial u^{n+1}}{\partial \theta^{n+1}} = -[\partial_{u^{n+1}} f]^{-1} \partial_{\theta^{n+1}} f$$

which gives us (written for 1 sample for simplicity, generally will have $\sum_i$ for each $u^{N,i}$)

$$\frac{\partial E(u^N;\{\theta^n\})}{\partial \theta^n} = \frac{\partial E(u^N)}{\partial u^N} (\prod_{i=n}^N \frac{\partial u^i}{\partial u^{i-1}}) \frac{\partial u^n}{\partial \theta^n} = \frac{\partial E(u^N)}{\partial u^N} (\prod_{i=n}^N [\partial_{u^i} f]^{-1} \partial_{u^{i-1}} f)([\partial_{u^n} f]^{-1} \partial_{\theta^n} f), \tag{7}$$

where everything on the right now can be computed with "traditional" automatic differentiation. Above we work with a similar loss $\min \, E(g^n, b_{-1}^n)$ as (2), which is only a function of u^N at the final time. The gradient computation calculated with (7) is cheap compared to naive `autograd` since one doesn't have to store the intermediate states from the iterative forward solver such as CG or truncated Neumann series used in root-finding (6). The parameters θ^n are considered to be the "controls" at each step n that drives the system towards u_{ref}^N.

Example 1 (Adapted from [16])**.** *Consider the ODE $\partial_t u(t) = -cu(t), u(0) = u_0$ for $u \colon [0, T] \mapsto \mathbb{R}^d$. The analytical solution in this case is $u(t) = u_0 e^{-ct}$. If one were to deploy the three point BDF scheme $(1 + g^n)u^{n+2} - (1 + 2g^n)u^{n+1} + g^n u^n = -c\Delta t u^{n+2}$ with $u^0 = u_0, u^1 = e^{-c\Delta t}u_0$, then the optimal*

$$g_0^* = \arg\min_{g \in \mathbb{R}} \left| \left(\frac{(1+2g)e^{-c\Delta t}}{1 + g + c\Delta t} - \frac{g}{1 + g + c\Delta t} \right) - e^{-2c\Delta_t} \right|^2$$

depend on $\Delta t, c$ and more generally g^n should be time-varying, instead of universally fixed at $0/0.5$.

With this connection between the implicit layer / neural ODE and machine-learned numerical solver drawn, we are ready to build on top and propose a few additional (spectral-based) algorithms coming from this differentiable simulation perspective. We highlight that all proposed methods (1) guarantee convergence to the true solution; (2) can evaluate solution $u^T(x)$ at any position x and the accuracy is resolution-invariant beyond certain sampling rate in space.

[1] https://implicit-layers-tutorial.org/

2 Spectral Methodology I: Fixed basis with prior knowledge

The above approach is inherently tied to a fine grid $\{x_j\}$ and operate entirely in physical space, and while one can try to "learn" a better discretization grid in this differentiable programming pipeline, a better way (i.e., sample-efficient way) is to employ spectral method. The difference from the Spectral Neural Operator [8] is instead of mapping coefficients to coefficients in a non-parametric way we model a dynamical system in the spectral domain by exploiting problem structure.

2.1 Deterministic

The most commonly used "spectral encoder" is probably the Fourier basis, which is efficient to compute. DFT (and its inverse) is a linear operation but most often the PDE itself is nonlinear. In this part, we use viscous Burgers' equation

$$\partial_t u(t,x) + u(t,x)\nabla_x u(t,x) - \epsilon\partial_{xx}u(t,x) = 0, u(0,x) = u_0(x) \text{ for } (x,t) \in (0,1)\times(0,T) \quad (8)$$

with periodic boundary condition $u(\cdot,0) = u(\cdot,1), \partial_x u(\cdot,0) = \partial_x u(\cdot,1)$ as the running example. By exploiting orthogonality of the dictionary, the Fourier–Galerkin method prescribes that the solution $\{u(t,x)\}_t$ stays in the fixed subspace spanned by colspace$(A) := \{e^{ikx} : k = -K/2, \ldots, K/2-1\}$. Transforming the PDE evolution to the Fourier domain, one ends up with K coupled ODEs:

$$\partial_t \hat{u}_k = -\frac{ik}{2}\sum_p \hat{u}_p\hat{u}_{k-p} - \epsilon k^2\hat{u}_k \quad \forall t > 0,\ k = -K/2, \ldots, K/2-1\,.$$

If we employ the three-point BDF on the resulting K-dimensional ODE involving $\{\hat{u}_k(t)\}_k$,

$$(1+g^n)\hat{u}_k^{n+2} - (1+2g^n)\hat{u}_k^{n+1} + g^n\hat{u}_k^n = \Delta t[-\frac{ik}{2}\sum_p \hat{u}_p^{n+2}\hat{u}_{k-p}^{n+2} - \epsilon k^2\hat{u}_k^{n+2}], \quad \forall k$$

with $\hat{u}_k^n \approx \hat{u}_k(t^n) \approx [A^\top u^n]_k$. Now each implicit layer, parameterized by g^n, solves $f_{g^n}(\hat{u}^{n+2}) - (1+2g^n)\hat{u}^{n+1} + g^n\hat{u}^n = 0_K$ for $\hat{u}^{n+2} \in \mathbb{R}^K$. One can optimize over $\{g^n\}$ using a similar loss as (2) at time t^N with j, J replaced by k, K again summed over I samples $\{(\hat{u}^{0,i}, \hat{u}_{ref}^{N,i}) \in \mathbb{R}^K \times \mathbb{R}^K\}_{i=1}^I$. The gradient $\partial E(\hat{u}^N; \{g^n\})/\partial g^n$ needed for update follow easily from (7). Small ϵ in (8) poses challenge as the solution can develop shock waves so in fact a multi-scale wavelet maybe more appropriate for "compression". In practice we often know the mini-max optimal reconstruction through K-term truncation of frames / RKHS [4] so these can be exploited as well for function class approximation. Another benefit of working with Fourier basis is its rotation equivariance in SO(3), which means if our PDE has rotation symmetry ($u^0 \to u^T$ imply $Ru^0 \to Ru^T$), and our ODE solver $\mathcal{S}$ obeys $\mathcal{S}(R\hat{u}^0) = R\mathcal{S}(\hat{u}^0)$ for its output, the methodology above will guarantee an equivariant PDE solver. Such property is often desirable as it improves stability while being more data-efficient [9].

2.2 Randomized

Spectral methods based on hand-crafted deterministic basis such as Fourier, Chebyshev orthogonal polynomial encounter "Kolmogorov n-widths" bottleneck [12, 20] since they form linear approximation (i.e., the solution lies in a fixed, finite-dimensional subspace oblivious to the input). One way to get around is to employ (regularized) Random Features model [17]. We consider a general ansatz space: $u^t(x) = A(x)v^t$ for $v^t \in \mathbb{R}^J$ assumed K-sparse, where the matrix A is random as opposed to e.g., DFT matrix. For each input $u^{0,i}$, one option is to pick A^i as a random Fourier matrix, the other is to let it be NN with random weights:

$$(a)\ [A^i(x)]_{lj} = \exp(i\omega_j x_l), \omega_j \sim \mathcal{N}(0,1); \quad (b)\ [A^i(x)]_{lj} = \sigma(\omega_j u^0(x_l)), \omega_j \sim \mathcal{N}(0,1)\,. \quad (9)$$

With such construction, we randomly sampled L points, denoted as $\{u^0(x_l)\}_{l=1}^L$ and solve for $v^0 \in \mathbb{R}^J$ by ℓ_1 minimization such as Lasso, where $L \approx K \ll J$. Assuming I observations, we evolve these $\{v^{0,i}\}_{i=1}^I$ to get $\{v^{N,i}\}_{i=1}^I$ by noting that the spectral ODE for e.g., the linear advection equation in this case becomes (ω is re-sampled for each i so A^i differs):

$$A^i(x)\partial_t v^i(t) + c\nabla_x A^i(x)v^i(t) = 0_L\,,\ i = 1, \ldots, I\,. \quad (10)$$

Above $\nabla_x A^i(x)$ is analytical, and we optimize over the time marching parameters $\{g^n\}$ using three-point BDF on I samples similar to Section 2.1. Since the random projection matrix A^i is not trained, the loss is imposed in a reduced $\approx K$-dim spectral domain. To obtain $v_{ref}^{N,i}$ for training, we randomly sample $\{u_{ref}^{N,i}(y_l)\}_{l=1}^{L}$ and use a least square fit on the same support as $v^{0,i}$. Crucially the input/output can be sampled on different grids and distinct across the i's. The Lasso LP is only solved once for each data pair on $\{u^{0,i}(x_l)\}_l$ outside the differentiable programming pipeline. We note that the decoder is non-linear due to the randomness in A^i, hence the search is not over a fixed subspace.

We offer some intuitions on the choices behind the random embedding of u in (9): (a) is related to random Fourier features and can be thought of as an approximate kernel method [17]; (b) is a one-layer NN and is related to Nerual Tangent Kernel [11] in the lazy training regime where the weights don't move far from the random initialization. The algorithm has the structure of encoder - processor - decoder, where the encoding from infinite-dimensional $u^0(x)$ to finite-dimensional v^0 (resp. $u^N(y)$ and v^N) will be *lossless* under standard Compressed Sensing assumptions. The dynamics in the spectral domain leverages physics knowledge and is known to be consistent, while data is used to learn a better time-stepping scheme $\{g^n\}$ for solving (10).

3 Spectral Methodology II: Learn the basis with differentiable programming

One can generalize further and consider data-driven encoder-decoder pair to integrate these learning components to the dynamical system pipeline. The matrix A can still take the same form as (9) but with trainable parameters ω_j: the former setup now is akin to compressed-sensing off-the-grid [21] and the latter becomes a shallow NN. Assuming again $u^t(x) = A(x)v^t$ for $v^t \in \mathbb{R}^J$ K-sparse, the following algorithm can then be trained end-to-end with gradient updates: with randomly initialized A (i.e., ω_j) and fixed g^n that recovers classical numerical method (e.g., $g^n = 0.5$), repeat

- Randomly sample L points $\{u^0(x_l)\}$ instead of fixed collocation points to avoid aliasing [2]
- With the current projection matrix $A(x_l;\omega)_{l=1}^{L} \in \mathbb{R}^{L\times J}$ for $L \ll J$, solve Lasso to get $v^0 \in \mathbb{R}^J$: $\min \|v^0\|_1 \ s.t. \ Av^0 = u^0(x_l)_{l=1}^{L}$
- Transform the PDE dynamics to a coupled-ODE dynamics in J-dimension. After time discretization, the resulting numerical scheme involves solving a (nonlinear) system (parameterized by g^n) at each step for advancing time as $\{v^n\} \to \{v^{n+1}\}$ similar to Section 2
- After N steps, form the loss in physical domain with $\{u^N(y_l)\}_l = A(y_l;\omega)v^N$ by randomly evaluate at L points. The above procedure is done on each sample $i \in [I]$ with the loss aggregated over $\{u^{N,i}(y_l), u_{ref}^{N,i}(y_l)\}_{l,i}$ to update the free parameters $\{g^n\}, \{\omega_j\}$

To train the "representation matrix" A consisting of J parameters $\{\omega_j\}$, we only need to work out one additional derivative for back-propagation (written below for 1 sample):

$$\frac{\partial E(u^N;\{g^n\},A)}{\partial A}\frac{\partial A}{\partial \omega_j} = \frac{\partial E}{\partial u^N}\Big(\frac{\partial u_N}{\partial v_N}\prod_{i=0}^{N-1}\frac{\partial v^{i+1}}{\partial v^i}\frac{\partial v^0}{\partial A} + \frac{\partial u^N}{\partial A}\Big)\frac{\partial A}{\partial \omega_j} = \frac{\partial E}{\partial u^N}\Big(A\prod_{i=0}^{N-1}\frac{\partial v^{i+1}}{\partial v^i}[-e_1^\top F^{-1}\frac{\partial G}{\partial A}] + v^N\Big)\frac{\partial A}{\partial \omega_j}$$

Above $\partial v^0/\partial A$ involves differentiate through the convex optimization Lasso solver mapping $\{A(x_l;\omega), u^0(x_l)\}_{l=1}^{L}$ to v^0, which can be viewed as a layer solving the KKT system: (λ is the dual)

$$\partial\|v^0\|_1 - A^\top\lambda = 0, Av^0 - u^0 = 0 \ \Rightarrow \ G(v^{0*}(A,u^0), \lambda^*(A,u^0), A, u^0) = 0\,,$$

implying [1] (matrix F can be easily computed from the definition of G, so does $\partial G/\partial A$)

$$\begin{bmatrix}\frac{\partial G}{\partial v^{0*}} & \frac{\partial G}{\partial \lambda^*}\end{bmatrix}\begin{bmatrix}\frac{\partial v^{0*}}{\partial A}\\ \frac{\partial \lambda^*}{\partial A}\end{bmatrix} = -\frac{\partial G}{\partial A} \ \Rightarrow \ \frac{\partial v^{0*}}{\partial A} = -e_1^\top\begin{bmatrix}\frac{\partial G}{\partial v^{0*}} & \frac{\partial G}{\partial \lambda^*}\end{bmatrix}^{-1}\frac{\partial G}{\partial A} =: -e_1^\top F^{-1}\frac{\partial G}{\partial A}\,. \tag{11}$$

The benefit of (11) comes from not needing to back-propagate through the numerical LP solver and unroll the updates as naive `autograd` would (i.e., we only leverage the optimality condition at v^{0*}, λ^*), therefore significantly saving memory and compute. Since A is learned it's data-adaptive, at the price of solving multiple Lasso through the training passes. The random grids $\{x_l\}, \{y_l\}$ are not re-sampled after the first pass, but can differ across the I samples. Compared to the algorithm proposed in [16], one can think of it as performing super-resolution with sampling / updating only on a coarse grid / lower dimensional space. A pictorial comparison of the methods is shown in Appendix A, along with discussions of the current work in relation to previous work in the literature.

[2]For e.g., band-limited function, there is a duality between Fourier coefficients $\{\hat{u}_k\}_{k=-K}^{K}$, and values of the function on the uniform grid $\{u(\frac{k}{2K+1})\}_{k=-K}^{K}$, and these completely characterize the continuous function $u(t)$.

References

[1] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. *Advances in neural information processing systems*, 32, 2019.

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

[3] Francesca Bartolucci, Emmanuel de Bezenac, Bogdan Raonic, Roberto Molinaro, Siddhartha Mishra, and Rima Alaifari. Representation equivalent neural operators: a framework for alias-free operator learning. *Advances in Neural Information Processing Systems*, 36, 2023.

[4] Pau Batlle, Matthieu Darcy, Bamdad Hosseini, and Houman Owhadi. Kernel methods are competitive for operator learning. *Journal of Computational Physics*, 496:112549, 2024.

[5] Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural PDE solvers. *arXiv preprint arXiv:2202.03376*, 2022.

[6] Joan Bruna, Benjamin Peherstorfer, and Eric Vanden-Eijnden. Neural Galerkin schemes with active learning for high-dimensional evolution equations. *Journal of Computational Physics*, 496:112588, 2024.

[7] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[8] VS Fanaskov and Ivan V Oseledets. Spectral neural operators. In *Doklady Mathematics*, pages 1–7. Springer, 2024.

[9] Jacob Helwig, Xuan Zhang, Cong Fu, Jerry Kurtin, Stephan Wojtowytsch, and Shuiwang Ji. Group Equivariant Fourier Neural Operators for Partial Differential Equations. In *International Conference on Machine Learning*, pages 12907–12930. PMLR, 2023.

[10] Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. Learning Neural PDE Solvers with Convergence Guarantees. *International Conference on Learning Representations*, 2019.

[11] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31, 2018.

[12] Nikola B Kovachki, Samuel Lanthaler, and Andrew M Stuart. Operator learning: Algorithms and analysis. *arXiv preprint arXiv:2402.15715*, 2024.

[13] Zongyi Li, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. In *International Conference on Learning Representations*, 2021.

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

[15] Brek Meuris, Saad Qadeer, and Panos Stinis. Machine-learning-based spectral methods for partial differential equations. *Scientific Reports*, 13(1):1739, 2023.

[16] Siddhartha Mishra. A machine learning framework for data driven acceleration of computations of differential equations. *arXiv preprint arXiv:1807.09519*, 2018.

[17] Nicholas H Nelsen and Andrew M Stuart. The random feature model for input-output maps between banach spaces. *SIAM Journal on Scientific Computing*, 43(5):A3212–A3243, 2021.

[18] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. 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, 2019.

[19] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International conference on machine learning*, pages 8459–8468. PMLR, 2020.

[20] Jacob Seidman, Georgios Kissas, Paris Perdikaris, and George J Pappas. NOMAD: Nonlinear manifold decoders for operator learning. *Advances in Neural Information Processing Systems*, 35:5601–5613, 2022.

[21] Tang, Gongguo and Bhaskar, Badri Narayan and Shah, Parikshit and Recht, Benjamin. Compressed sensing off the grid. *IEEE transactions on information theory*, 59(11):7465–7490, 2013.

[22] Kiwon Um, Robert Brand, Yun Raymond Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 33:6111–6122, 2020.

A Related Work & Discussion

Compared to the traditional numerical PDE solvers, machine learning based methods such as Fourier Neural Operator [13], DeepONet [14] rely on data to learn a surrogate mapping between infinite-dimensional function spaces to amortize the cost. By incorporating appropriate problem structure instead of being fully-generic, our proposed data-driven methods hold promise for being more sample-efficient. On the contrary, methods such as PINN [18] explicitly use the PDE information but doesn't leverage the availability of data one might have on similar problems. Neural Galerkin [6] attempts to learn a "better" sampling measure over time in the spirit of active learning however also doesn't take as input a priori data about the PDE solutions.

Some additional recent work seeking to address data & physics incorporation include:

- [22, 10] aim to learn the residual / correction term from low-fidelity traditional numerical solver but is typically more expensive to train with long roll-outs compared to our method.
- [15] extracts a learned "basis" from DeepONet output for expansion in spectral domain which means it is not trained end-to-end. This prevents the error from the representation to interact with the error from the solver for optimizing the performance.
- [2] learns discretization stencil directly in the physical space and deviates from the "learning representation basis" approach that we take here, which can be more data-efficient.
- The works of [5] and [19] follow similar encoder-processor-decoder paradigm but leverage a different architecture, namely graph neural network (GNN), which is motivated by the fact that message-passing neural networks representationally contain several classical solvers. These reply on the fact that locally

  $$[\partial_x^{(n)} u]_i \approx \sum_{j \in \mathcal{N}(i)} \alpha_j^{(n)} u_j$$

  for train-able α_j's which essentially adopts a particle-based representation of the physical system. Such architecture choice maybe more conducive to enforcing equivariant properties.

In our work, we adopt the encoder-processor-decoder paradigm that a lot of the traditional numerical methods do, but put learning components into each of the 3 modules using available data.

Our method is an example of ReNO [3] which admits equivalence between neural operator and its discrete representation under compressed sensing conditions. By blending Harmonic Analysis with Differentiable Programming, we are able to propose a *resolution-invariant* and *memory-efficient* algorithm that holds promise in terms of both speed and accuracy. We plan to systematically evaluate them on large-scale problems in the future.

Remark 1. *One rationale behind such low-frequency Fourier basis in Section 2.1 is that if one assumes the initial condition $u(0,\cdot)$ is drawn according to a Gaussian random field, the Karhunen-Loève decomposition entails that approximation of the function would involve a few leading random Fourier coefficients.*

Remark 2. *We further remark that for the methodology in Section 2.2, it is possible to have an input-adaptive $(A^i, v^{0,i})$ for each $u^{0,i}$, while learning the $\{g^n\}$ across all I samples. This would involve formulating the convex optimization over the (infinite-dimensional) space of measures for each sample in the sense of*

$$\begin{aligned} \inf_{\mu} \quad & \|\mu\|_{TV} = \|v^0\|_1 \\ s.t. \quad & \Phi\mu = u^0(x_l)_{l=1}^L \end{aligned}$$

and solve a SDP as done in [21]. In the above

$$\phi(\beta) = [\sigma(\beta \cdot u^0(x_l))]_{l=1,\dots,L}, \; \mu(\beta) = \sum_{k=1}^K v_k^0 \cdot \delta_{\beta_k},$$

and

$$\Phi\mu = \int_{\mathbb{R}} \phi(\beta) d\mu(\beta) = [\sum_{k=1}^K v_k^0 \cdot \sigma(\beta_k \cdot u^0(x_l))]_{l=1,\dots,L}\,.$$

Put differently, we are trying to recover K spikes given L measurements of the continuous function u^0 randomly sampled at $\{u^0(x_l)\}_{l=1}^L$. Theory exists in this case supporting that $L \approx K$ suffices under mild assumptions. This program will be more computationally intensive compared to the Lasso LP solver employed in Section 2.2 for random features, although this is a one-time pre-processing cost (i.e., outside the differentiable programming pipeline). The advantage is having now $\beta^i \in \mathbb{R}^K$ being data-dependent instead of random, with non-linear encoder/decoder pair (note the basis ϕ depends on u^0 and the recovered $\{\beta_k^i\}$ will generally vary across the I samples). The training loss, posed in the spectral (i.e., μ) domain, can be optimized over $\{g^n\}$ to identify the best updates as done in Section 2.2.

Remark 3. *One can draw connection of our method in Section 3 to DeepONet [14] architecture $u^N(y) := \mathcal{G}(u^0)(y) = \langle g(u^0(x_1), \ldots, u^0(x_l)), f(y)\rangle$, where the two learned embeddings f, g depend on y and $\{u^0(x_l)\}$. In contrast we have $u^N(y) = A(y; \omega)v^N$, but (1) we place more constraints on the e.g., $\{u^0(x_l)\} \to v^0 \to v^N$ mapping by exploiting the PDE structure; (2) A in our case can depend on u^0 as well (c.f. option (b) in (9)) and not simply a function of y.*

Below in Figure 1 we also give a comparison of the methods discussed in the paper.

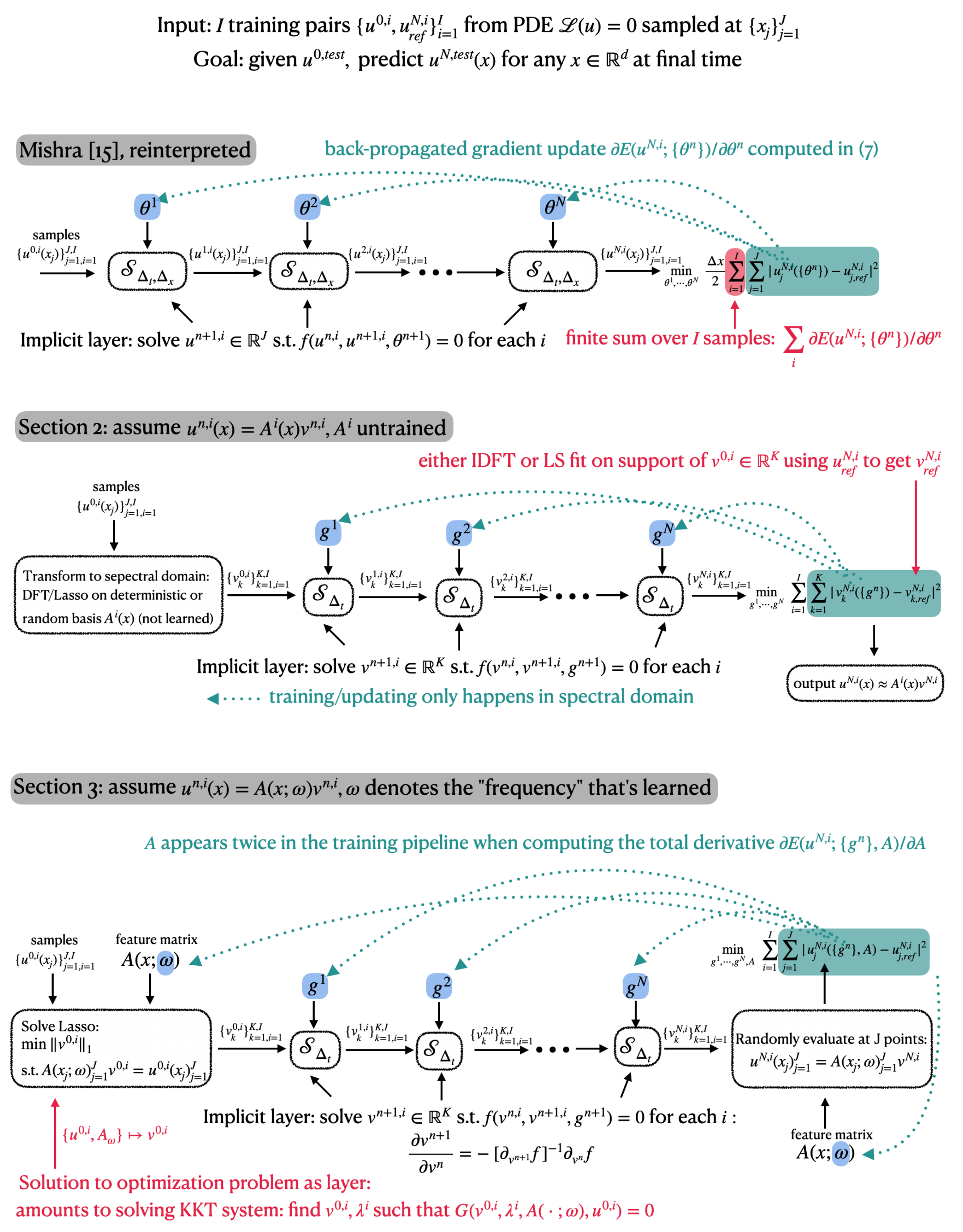
Input: I training pairs $\{u^{0,i}, u^{N,i}_{ref}\}_{i=1}^{I}$ from PDE $\mathscr{L}(u)=0$ sampled at $\{x_j\}_{j=1}^{J}$
Goal: given $u^{0,test}$, predict $u^{N,test}(x)$ for any $x \in \mathbb{R}^d$ at final time
Mishra [15], reinterpreted
back-propagated gradient update $\partial E(u^{N,i}; \{\theta^n\})/\partial\theta^n$ computed in (7)
samples $\{u^{0,i}(x_j)\}_{j=1,i=1}^{J,I}$
θ^1
θ^2
θ^N
$\mathcal{S}_{\Delta_t,\Delta_x}$
$\{u^{1,i}(x_j)\}_{j=1,i=1}^{J,I}$
$\{u^{2,i}(x_j)\}_{j=1,i=1}^{J,I}$
$\{u^{N,i}(x_j)\}_{j=1,i=1}^{J,I}$
$\min_{\theta^1,\cdots,\theta^N} \frac{\Delta x}{2} \sum_{i=1}^{I} \sum_{j=1}^{J} |u_j^{N,i}(\{\theta^n\}) - u^{N,i}_{j,ref}|^2$
Implicit layer: solve $u^{n+1,i} \in \mathbb{R}^J$ s.t. $f(u^{n,i}, u^{n+1,i}, \theta^{n+1}) = 0$ for each i
finite sum over I samples: $\sum_i \partial E(u^{N,i}; \{\theta^n\})/\partial\theta^n$
Section 2: assume $u^{n,i}(x) = A^i(x)v^{n,i}$, A^i untrained
either IDFT or LS fit on support of $v^{0,i} \in \mathbb{R}^K$ using $u^{N,i}_{ref}$ to get $v^{N,i}_{ref}$
samples $\{u^{0,i}(x_j)\}_{j=1,i=1}^{J,I}$
Transform to sepectral domain: DFT/Lasso on deterministic or random basis $A^i(x)$ (not learned)
$\{v_k^{0,i}\}_{k=1,i=1}^{K,I}$
$\{v_k^{1,i}\}_{k=1,i=1}^{K,I}$
$\{v_k^{2,i}\}_{k=1,i=1}^{K,I}$
$\{v_k^{N,i}\}_{k=1,i=1}^{K,I}$
g^1
g^2
g^N
$\mathcal{S}_{\Delta_t}$
$\min_{g^1,\cdots,g^N} \sum_{i=1}^{I} \sum_{k=1}^{K} |v_k^{N,i}(\{g^n\}) - v^{N,i}_{k,ref}|^2$
output $u^{N,i}(x) \approx A^i(x)v^{N,i}$
Implicit layer: solve $v^{n+1,i} \in \mathbb{R}^K$ s.t. $f(v^{n,i}, v^{n+1,i}, g^{n+1}) = 0$ for each i
training/updating only happens in spectral domain
Section 3: assume $u^{n,i}(x) = A(x;\omega)v^{n,i}$, ω denotes the "frequency" that's learned
A appears twice in the training pipeline when computing the total derivative $\partial E(u^{N,i}; \{g^n\}, A)/\partial A$
samples $\{u^{0,i}(x_j)\}_{j=1,i=1}^{J,I}$
feature matrix $A(x;\omega)$
Solve Lasso: $\min \|v^{0,i}\|_1$ s.t. $A(x_j;\omega)_{j=1}^{J} v^{0,i} = u^{0,i}(x_j)_{j=1}^{J}$
$\{v_k^{0,i}\}_{k=1,i=1}^{K,I}$
$\{v_k^{1,i}\}_{k=1,i=1}^{K,I}$
$\{v_k^{2,i}\}_{k=1,i=1}^{K,I}$
$\{v_k^{N,i}\}_{k=1,i=1}^{K,I}$
g^1
g^2
g^N
$\mathcal{S}_{\Delta_t}$
$\min_{g^1,\cdots,g^N,A} \sum_{i=1}^{I} \sum_{j=1}^{J} |u_j^{N,i}(\{g^n\}, A) - u^{N,i}_{j,ref}|^2$
Randomly evaluate at J points: $u^{N,i}(x_j)_{j=1}^{J} = A(x_j;\omega)_{j=1}^{J} v^{N,i}$
feature matrix $A(x;\omega)$
Implicit layer: solve $v^{n+1,i} \in \mathbb{R}^K$ s.t. $f(v^{n,i}, v^{n+1,i}, g^{n+1}) = 0$ for each i :
$\frac{\partial v^{n+1}}{\partial v^n} = -[\partial_{v^{n+1}} f]^{-1} \partial_{v^n} f$
$\{u^{0,i}, A_\omega\} \mapsto v^{0,i}$
Solution to optimization problem as layer:
amounts to solving KKT system: find $v^{0,i}, \lambda^i$ such that $G(v^{0,i}, \lambda^i, A(\cdot;\omega), u^{0,i}) = 0$

Figure 1: Algorithmic pipeline with end-to-end training (trainable components are shaded in blue). The crux of our efficient implementation comes from *implicit function theorem*, either by defining **implicit layer** or **convex optimization layer**.