AN EXTENSIBLE BENCHMARK SUITE FOR LEARNING TO SIMULATE PHYSICAL SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Time integration of models of physical systems is a core task of scientific computing. Recently, there has been a surge of interest in data-driven methods that learn from data a model of the physical system and then integrate it in time to make predictions. This work introduces benchmarks for evaluating data-driven methods on a variety of physical systems and proposes evaluation scenarios. The proposed benchmarks comprise three representative physical systems (spring, spring mesh, wave) and a collection of classical time integrators as baselines. For demonstration purposes, we apply several data-driven methods to the benchmarks and report accuracy and computational efficiency.

1 INTRODUCTION

Integrating models of physical systems in time is a core task in scientific computing. Traditional time integrators require knowledge of the physical model of the system of interest, which is given typically in the form of partial differential equations (PDEs). However, if the physical model is unknown, this classical approach is not applicable. Recently, there has been interest in methods that learn governing equations, and how to integrate them, from data sampled from the system of interest. A successful adoption of such data-driven methods into scientific computing pipelines requires a solid and exhaustive assessment of their performance — an increasingly challenging task given the large diversity of physical systems and corresponding data-driven solutions, combined with the lack of standardized sets of problems and comparison protocols and metrics.

In this work, we introduce an extensible benchmark pipeline, by proposing: (1) a set of simple, yet representative, physical models, with a range of training and evaluation boundary conditions, coefficients, and parameters, as well as reference, high-accuracy solutions which are used to evaluate data-driven methods, (2) reference implementations of traditional time integration algorithms, which are used as baselines for evaluation, and (3) implementations of widely used data-driven methods, including physics-agnostic multi-layer perceptions (MLPs), efficient kernel machines, and geometric deep learning models based on graph neural networks. Further, our benchmark suite is modular, permitting extensions with limited code changes. We focus on the setting where the physical model is unavailable during training, mimicking situations in computational science and engineering with ample data and a lack of models. Our preliminary numerical experiments show promising results for the considered data-driven methods and at the same time reveal opportunities for improvements.

2 RELATED WORK

The purpose of the proposed benchmarks is to enable comparisons of different learning-based methods in terms of their accuracy and efficiency. We briefly review several streams of learning methods for physical systems:

Models for learning physics. One line of work aims to understand how neural networks can be structured and trained to reproduce known physical system behavior, with the goal of designing general methods applicable in a variety of settings (Greydanus et al., 2019; Sanchez-Gonzalez et al., 2019; Chen et al., 2020; Sanchez-Gonzalez et al., 2018; Raissi et al., 2017; 2019; Lu et al., 2019; Haghighat et al., 2020; Tartakovsky et al., 2018).

Accelerating solving PDEs. Another line of research aims to develop a variety of techniques to accelerate solving PDEs. Typically, these methods are developed for specific PDEs and a specific restricted range of problems. For example, fluid dynamics problems (Ribeiro et al., 2020; Kim et al., 2019; Xie et al., 2018), with particular applications to cardiovascular modeling (Liang et al., 2020; Kissas et al., 2020) and aerodynamics (Umetani & Bickel, 2018); or solid mechanics simulation tasks, including stresses (Nie et al., 2020; Liang et al., 2018; Maso Talou et al., 2020; Khadilkar et al., 2019; Li et al., 2018; Lia et al.). In cases where the governing equations are not given, the learning task becomes approximating them from data (Packard et al., 1980; Crutchfield & Mcnamara, 1987; Antoulas & Anderson, 1986; Gustavsen & Semlyen, 1999; by Athanasios C. Antoulas et al., 2020; Nakatsukasa et al., 2018; Brunton et al., 2016; Schaeffer, 2017; Schaeffer et al., 2018; Schmid & J., 2008; Schmid, 2010; Tu et al., 2014; Qian et al., 2020).

Predicting quantities of interest. Yet another line of work is concerned with problems where one is interested solely in quantities of interest (e.g., compliance, drag), which typically are obtained via functionals of solutions, but the solution fields themselves are of less importance (Zhu et al., 2019; Baque et al., 2018; Umetani & Bickel, 2018; Papila et al., 2016; White et al., 2019; Sasaki & Igarashi, 2019; Lin et al., 2018; Liu et al., 2020; Hoole et al., 2020).

3 Background and problem setup

PDEs, dynamical systems, and time integration. Consider a time-dependent PDE of the form $\partial_t u = \mathcal{L}(u)$, where u is the solution function and $\mathcal{L}$ is a potentially nonlinear operator that includes spatial derivatives of u. By discretizing in space, one obtains a dynamical system such as the (first-order) system

$$\dot{x}(t) = f(x(t)) \tag{1}$$

with an N-dimensional state $x(t) \in \mathbb{R}^N$ at time $t \in [0, T]$. The function f is Lipschitz to ensure solution uniqueness and the initial condition is denoted as $x_0 \in \mathbb{R}^N$. If we have a second-order system $\ddot{q}(t) = f(q(t))$, then we consider its formulation via position q and velocity p as a first-order system $[\dot{q}(t); \dot{p}(t)] = [p(t); f(q(t))]$. To numerically integrate (1), we choose time steps $0 = t_0 < t_1 < \cdots < t_K = T$. Then, a time integration scheme (Süli & Mayers, 2003; Hairer et al., 2009; Hairer & Wanner, 2009) gives an approximation $x_k \approx x(t_k)$ of the state $x(t_k)$ at each time step $k = 1, \ldots, K$.

Problem setup and learning problems. Given M initial conditions $x_0^{(1)}, \ldots, x_0^{(M)} \in \mathbb{R}^N$ and the corresponding M trajectories $X^{(i)} = [x_0^{(i)}, \ldots, x_K^{(i)}] \in \mathbb{R}^{N \times (K+1)}, i = 1, \ldots, M$ obtained with a time integration scheme from dynamical system (1), we consider the problem of learning an approximation $\tilde{f}$ of the right-hand side function f (1). This gives an approximate $\dot{\tilde{x}}(t) = \tilde{f}(\tilde{x}(t))$ that is then numerically integrated to produce a trajectory $\tilde{X}$ for an initial condition $\tilde{x}_0$. The aim is that $\tilde{X}$ approximates well the true trajectory X obtained with f from (1) for the same initial condition.

To assess the learned models, we evaluate them on their ability to predict derivatives producing good approximate trajectories from randomly sampled initial conditions. During evaluation, we use initial conditions drawn independently from those used to produce training data, both from the same distribution as the training samples, as well as from a distribution with support outside the training range. We train networks on data sets of various sizes. For details, see Appendix B.

Data-driven time integration schemes. We consider a variety of common machine learning methods: a k-nearest neighbors (KNN) regressor which memorizes the input-output pairs from the training set, a neural network kernel, and two simple MLP architectures. Additionally, we consider a graph neural network derived from Pfaff et al. (2020). This network predicts accelerations for each system, and integrates these to produce predictions for the velocity which are then provided to each of the numerical integration methods when computing trajectories. The graph structure is selected to match a static mesh chosen for each of our systems. Architecture details are provided in Appendix B.

4 Benchmark systems

The core of our proposed benchmark are the physical systems which we use to generate ground truth data for training and subsequent evaluation. We selected three representative systems of increasing

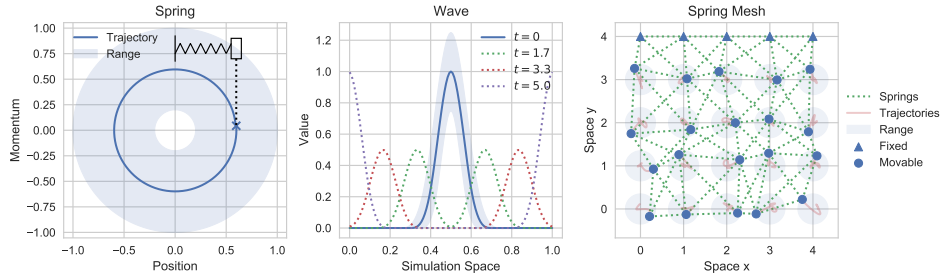

Figure 1: Representative initial conditions for the three systems. Each two state components: a position q and momentum p. Shaded blue regions denote the sampling range for initial states.

complexity (Appendix C). For each of these systems, training and evaluation trajectories are produced by backward (implicit) Euler integration with a very small timestep. Sample initial conditions for these systems are illustrated in Figure 1.

Spring. One of the simplest physical systems is a single spring with rest length zero oscillating in a space of one dimension: $[\dot{q}(t), \dot{p}(t)] = [p(t), -q(t)]$. States evolve in a one-dimensional space.

Spring Mesh. A direct extension of the single spring is a set of masses connected by springs. This system has a larger configuration space and allows testing the scalability of data-driven methods. Here, particles are arranged in a unit grid, with springs along the grid and across diagonals. Masses and states are two-dimensional.

Wave. Moving away from springs, we adopt a linear wave system closely following Peng & Mohseni (2016) to test time-integration of a more challenging PDE. On a space $[0, 1)$ with periodic boundary conditions we have $\partial_{tt} u = c^2 \partial_{xx} u$, where $c = 0.1$ is the wave speed. We represent this as a first-order system with

$$\begin{bmatrix} \dot{q}(t) \\ \dot{p}(t) \end{bmatrix} = \begin{bmatrix} 0 & I \\ c^2 D_{xx} & 0 \end{bmatrix} \begin{bmatrix} q(t) \\ p(t) \end{bmatrix} , \tag{2}$$

where $D_{xx} \in \mathbb{R}^{n \times n}$ corresponds to the three-point central difference approximation of the spatial derivative ∂_{xx}. Here, we discretize the one-dimensional space with $n = 125$ grid points.

5 Numerical Experiments

We compare the performance of each of the considered methods on our three benchmark systems. For each system, we select a time step size at which integration succeeds with acceptable error. We train the methods on trajectory snapshots from randomly sampled initial conditions, and measure the accuracy of the trajectories resulting from the learned approximation of the right hand side function with several integration schemes as well as the computational overhead of the learned methods.

Our results show that the learning methods successfully approximate these systems even without access to the underlying model. Potential improvements could address accuracy, data requirements, and computational overhead. In most cases the kernel method—though one of the simplest models—performs well. The graph network also demonstrates impressive stability, even producing good results outside of the distribution of training samples. This may be due to the encoding of the true mesh structure directly into the network architecture, demonstrating the potential of such techniques.

6 Conclusion

A standardized set of problems is valuable to provide consistent accuracy and performance measurements as more learning methods for data-driven time integration emerge. The focus of our benchmarks is on simplicity and the setting where training samples are available but access to the underlying model is not. In the future, we hope to extend this benchmark suite to include additional systems with different physical behavior and to cover a wider range of data-driven tasks in scientific computing.

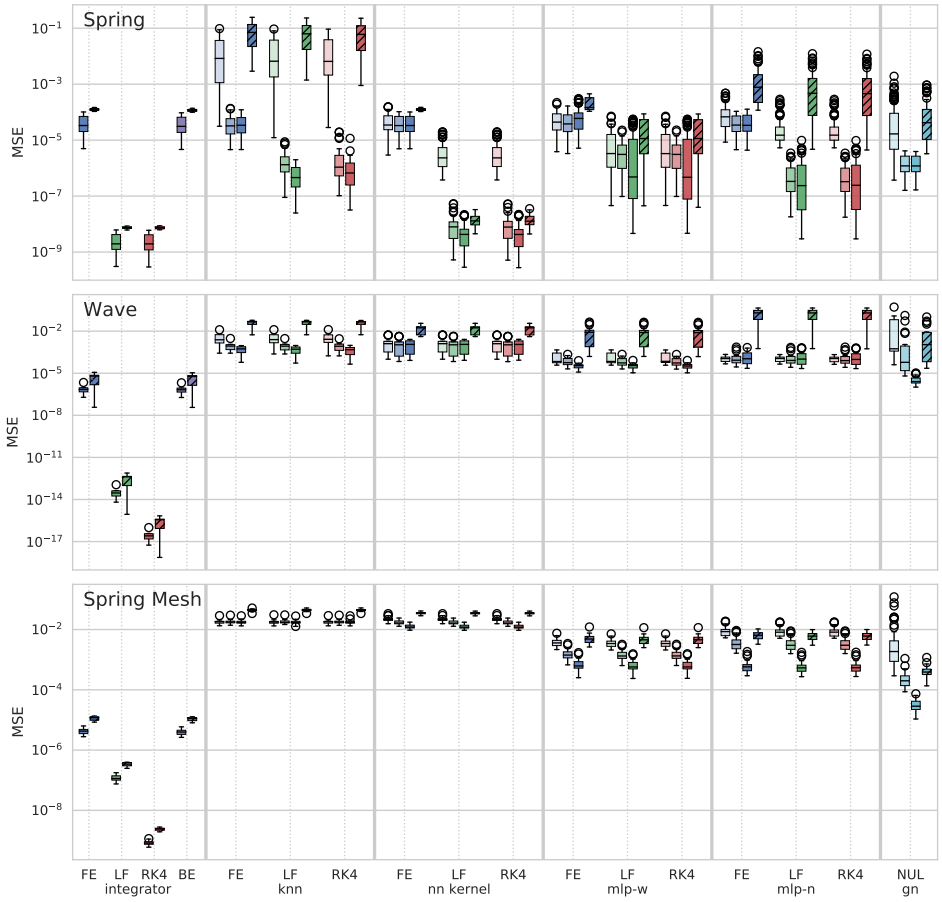

Figure 2: Evaluation error distribution for all systems. MSE is averaged across all evaluation snapshots from all evaluation trajectories, as well as three independent trainings for each neural network, showing mean error over time from different initial conditions and as a result of training. Darker colors represent larger training sets, and the final hatched boxes show out-of-distribution error. Integration schemes are forward Euler, leapfrog, RK4, and backward Euler.

Table 1: Relative slowdowns for equivalent error for learned method. For each base integrator, match error rates with the learned method in the column by increasing time step sizes. Each cell reports the resulting time slowdown factor and (after the slash) the time step size scaling necessary.

Base Integrator		mlp-w		mlp-n		nn-kernel		gn	knn	
		FE	RK4	FE	RK4	FE	RK4	NUL	FE	RK4
Spring	FE	19.9 / 1	19.9 / 1	23.2 / 1	23.2 / 1	19.2 / 1	19.2 / 1	376.2 / 1	106.8 / 1	106.8 / 1
	LF	29.9 / 32	29.9 / 32	34.7 / 32	34.7 / 32	28.9 / 32	28.9 / 32	506.7 / 4	160.2 / 32	160.2 / 32
	RK4	19.9 / 1	19.9 / 1	23.2 / 1	23.2 / 1	19.2 / 1	19.2 / 1	541.0 / 32	106.8 / 1	106.8 / 1
	BE	29.8 / 1	29.8 / 1	34.6 / 1	34.6 / 1	28.8 / 1	28.8 / 1	562.1 / 1	159.6 / 1	159.6 / 1
Wave	FE	69.8 / 4	69.8 / 4	50.7 / 8	50.7 / 8	104.7 / 4	104.7 / 4	873.6 / 2	318.8 / 8	318.8 / 8
	LF	103.2 / 128	103.2 / 128	39.5 / 128	39.5 / 128	154.7 / 128	154.7 / 128	2049.4 / 64	248.0 / 128	248.0 / 128
	RK4	123.3 / 128	123.3 / 128	47.1 / 128	47.1 / 128	184.9 / 128	184.9 / 128	1652.7 / 128	296.2 / 128	296.2 / 128
	BE	29.7 / 16	29.7 / 16	58.9 / 64	58.9 / 64	230.9 / 64	230.9 / 64	183.6 / 4	370.0 / 64	370.0 / 64
Spring Mesh	FE	3.5 / 8	3.5 / 8	2.8 / 8	2.8 / 8	4.0 / 16	4.0 / 16	34.2 / 8	102.7 / 16	102.7 / 16
	LF	3.5 / 8	3.5 / 8	2.8 / 8	2.8 / 8	4.0 / 16	4.0 / 16	34.2 / 8	102.7 / 16	102.7 / 16
	RK4	3.5 / 8	3.5 / 8	2.8 / 8	2.8 / 8	4.0 / 16	4.0 / 16	34.2 / 8	102.7 / 16	102.7 / 16
	BE	3.5 / 8	3.5 / 8	2.8 / 8	2.8 / 8	4.0 / 16	4.0 / 16	34.2 / 8	102.7 / 16	102.7 / 16

REFERENCES

A. C. Antoulas and B. D. O. Anderson. On the scalar rational interpolation problem. *IMA Journal of Mathematical Control & Information*, 3(2-3):61–88, 1986.

Pierre Baque, Edoardo Remelli, François Fleuret, and Pascal Fua. Geodesic convolutional shape optimization. *arXiv preprint arXiv:1802.04016*, 2018.

Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

by Athanasios C. Antoulas, Christopher Beattie, and Serkan Gugercin. *Interpolatory Methods for Model Reduction*. SIAM, 2020.

Zhengdao Chen, Jianyu Zhang, Martin Arjovsky, and Léon Bottou. Symplectic recurrent neural networks. In *International Conference on Learning Representations*, 2020.

James P. Crutchfield and Bruce S. Mcnamara. Equations of motion from a data series. *Complex Systems*, pp. 452, 1987.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32, pp. 15379–15389. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/26cd8ecadce0d4efd6cc8a8725cbd1f8-Paper.pdf.

B. Gustavsen and A. Semlyen. Rational approximation of frequency domain responses by vector fitting. *Power Delivery, IEEE Transactions on*, 14(3):1052–1061, Jul 1999.

Ehsan Haghighat, Maziar Raissi, Adrian Moure, Hector Gomez, and Ruben Juanes. A deep learning framework for solution and discovery in solid mechanics, 2020.

Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer, 2009.

Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, 2009.

Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. URL https://doi.org/10.1038/s41586-020-2649-2.

Joshua Hoole, Pia Sartor, Julian D Booker, Jonathan E Cooper, Xenofon Gogouvitis, and R Kyle Schmidt. Comparison of surrogate modeling methods for finite element analysis of landing gear loads. In *AIAA Scitech 2020 Forum*, pp. 0681, 2020.

Aditya Khadilkar, Jun Wang, and Rahul Rai. Deep learning–based stress prediction for bottom-up sla 3d printing process. *The International Journal of Advanced Manufacturing Technology*, 102(5):2555–2569, Jun 2019. ISSN 1433-3015. doi: 10.1007/s00170-019-03363-4. URL https://doi.org/10.1007/s00170-019-03363-4.

Byungsoo Kim, Vinicius C Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep fluids: A generative network for parameterized fluid simulations. In *Computer Graphics Forum*, volume 38, pp. 59–70. Wiley Online Library, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Georgios Kissas, Yibo Yang, Eileen Hwuang, Walter R Witschey, John A Detre, and Paris Perdikaris. Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4d flow mri data using physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 358:112623, 2020.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL https://doi.org/10.1145/2833157.2833162.

Yu Li, Hu Wang, Kangjia Mo, and Tao Zeng. Reconstruction of simulation-based physical field by reconstruction neural network method. *arXiv preprint arXiv:1805.00528*, 2018.

Yu Lia, Hu Wanga, Wenquan Shuaia, Honghao Zhangb, and Yong Pengb. Image-based reconstruction for the impact problems by using dpnns.

Liang Liang, Minliang Liu, Caitlin Martin, and Wei Sun. A deep learning approach to estimate stress distribution: a fast and accurate surrogate of finite-element analysis. *Journal of The Royal Society Interface*, 15(138):20170844, 2018. doi: 10.1098/rsif.2017.0844. URL https://royalsocietypublishing.org/doi/abs/10.1098/rsif.2017.0844.

Liang Liang, Wenbin Mao, and Wei Sun. A feasibility study of deep learning for predicting hemodynamics of human thoracic aorta. *Journal of Biomechanics*, 99:109544, 2020.

Qiyin Lin, Jun Hong, Zheng Liu, Baotong Li, and Jihong Wang. Investigation into the topology optimization for conductive heat transfer based on deep learning approach. *International Communications in Heat and Mass Transfer*, 97:103–109, 2018.

Xing Liu, Christos E Athanasiou, Nitin P Padture, Brian W Sheldon, and Huajian Gao. A machine learning approach to fracture mechanics problems. *Acta Materialia*, 2020.

Lu Lu, Xuhui Meng, Zhiping Mao, and George E Karniadakis. Deepxde: A deep learning library for solving differential equations. *arXiv preprint arXiv:1907.04502*, 2019.

Gonzalo D Maso Talou, Thiranja P Babarenda Gamage, Mark Sagar, and Martyn P Nash. Deep learning over reduced intrinsic domains for efficient mechanics of the left ventricle. *Frontiers in Physics*, 8:30, 2020.

Yuji Nakatsukasa, Olivier Sète, and Lloyd N. Trefethen. The aaa algorithm for rational approximation. *SIAM Journal on Scientific Computing*, 40(3):A1494–A1522, 2018.

Zhenguo Nie, Haoliang Jiang, and Levent Burak Kara. Stress field prediction in cantilevered structures using convolutional neural networks. *Journal of Computing and Information Science in Engineering*, 20(1), 2020.

Norman H. Packard, James P. Crutchfield, J. Doyne Farmer, and Robert S. Shaw. Geometry from a time series. *Physical Review Letters*, 45:712–716, 1980.

Nilay Papila, Wei Shyy, Lisa Griffin, and Daniel Dorney. Shape optimization of supersonic turbines using response surface and neural network methods. In *39th Aerospace Sciences Meeting and Exhibit*, pp. 1065, 2016.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Liqian Peng and Kamran Mohseni. Symplectic model reduction of hamiltonian systems. *SIAM Journal on Scientific Computing*, 38(1):A1–A27, 2016. URL https://doi.org/10.1137/140978922.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint*, 2020. URL https://arxiv.org/abs/2010.03409.

E. Qian, B. Kramer, B. Peherstorfer, and K. Willcox. Lift & learn: Physics-informed machine learning for large-scale nonlinear dynamical systems. *Physica D: Nonlinear Phenomena*, 2020.

Ali Rahimi, Benjamin Recht, et al. Random features for large-scale kernel machines. In *NIPS*, volume 3, pp. 5. Citeseer, 2007.

Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. arxiv. *arXiv preprint arXiv:1711.10561*, 2017.

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.

Mateus Dias Ribeiro, Abdul Rehman, Sheraz Ahmed, and Andreas Dengel. Deepcfd: Efficient steady-state laminar flow approximation with deep convolutional neural networks. *arXiv preprint arXiv:2004.08826*, 2020.

Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pp. 4470–4479. PMLR, 2018.

Alvaro Sanchez-Gonzalez, Victor Bapst, Kyle Cranmer, and Peter Battaglia. Hamiltonian graph networks with ODE integrators. *arXiv preprint*, 2019. URL https://arxiv.org/abs/1909.12790.

Hidenori Sasaki and Hajime Igarashi. Topology optimization accelerated by deep learning. *IEEE Transactions on Magnetics*, 55(6):1–5, 2019.

H. Schaeffer, G. Tran, and R. Ward. Extracting sparse high-dimensional dynamics from limited data. *SIAM Journal on Applied Mathematics*, 78(6):3279–3295, 2018.

Hayden Schaeffer. Learning partial differential equations via data discovery and sparse optimization. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2197): 20160446, 2017.

P. Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, 8 2010. ISSN 1469-7645.

P. Schmid and Sesterhenn J. Dynamic mode decomposition of numerical and experimental data. In *Bull. Amer. Phys. Soc., 61st APS meeting*, pp. 208. American Physical Society, 2008.

Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels*. MIT Press, 1998.

Endre Süli and David F. Mayers. *An Introduction to Numerical Analysis*, chapter Initial Value Problems for ODEs, pp. 349–353. Cambridge University Press, 2003.

Alexandre M Tartakovsky, Carlos Ortiz Marrero, Paris Perdikaris, Guzel D Tartakovsky, and David Barajas-Solano. Learning parameters and constitutive relationships with physics informed deep neural networks. *arXiv preprint arXiv:1808.03398*, 2018.

Jonathan H. Tu, Clarence W. Rowley, Dirk M. Luchtenburg, Steven L. Brunton, and J. Nathan Kutz. On dynamic mode decomposition: Theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014.

Nobuyuki Umetani and Bernd Bickel. Learning three-dimensional flow for interactive aerodynamic design. *ACM Transactions on Graphics (TOG)*, 37(4):1–10, 2018.

Daniel A White, William J Arrighi, Jun Kudo, and Seth E Watts. Multiscale topology optimization using neural network surrogate models. *Computer Methods in Applied Mechanics and Engineering*, 346:1118–1135, 2019.

You Xie, Erik Franz, Mengyu Chu, and Nils Thuerey. TempoGAN: A temporally coherent, volumetric GAN for super-resolution fluid flow. *ACM Transactions on Graphics (TOG)*, 37(4):1–15, 2018.

Yinhao Zhu, Nicholas Zabaras, Phaedon-Stelios Koutsourelakis, and Paris Perdikaris. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *Journal of Computational Physics*, 394:56–81, 2019.

A Numerical Integration Schemes

We briefly review the time integration schemes that we consider in this study: forward Euler (FE), leapfrog (LF), Runge-Kutta 4 (RK4), and backward Euler (BE). Other sources also discuss these integration schemes, for example Süli & Mayers (2003); Hairer et al. (2009); Hairer & Wanner (2009).

Time integration with the explicit Euler method leads to

$$x_k = x_{k-1} + \delta \mathrm{t}\, f(x_{k-1}),$$

where $\delta \mathrm{t} > 0$ is the time step size and f is the right-hand side function. The explicit Runge-Kutta 4 scheme is

$$x_k = x_{k-1} + \frac{\delta \mathrm{t}}{6}\left(h_1 + 2h_2 + 2h_3 + h_4\right),$$

where

$$\begin{aligned} h_1 &= f(x_{k-1}) & h_2 &= f(x_{k-1} + \delta \mathrm{t}\,/2h_1) \\ h_3 &= f(x_{k-1} + \delta \mathrm{t}\,/2h_2) & h_4 &= f(x_{k-1} + \delta \mathrm{t}\,/2h_3) \end{aligned}$$

for $k = 1, \ldots, K$. For leapfrog integration we separate the components of the state $x = (q, p)$ and $f(q_k, p_k) = (\dot{q}_k, \dot{p}_k)$ and compute:

$$\begin{aligned} p_{k+1/2} &= p_k + \frac{\delta \mathrm{t}}{2}\dot{p}_k \\ q_{k+1} &= q_k + \dot{q}(q_k, p_{k+1/2})\,\delta \mathrm{t} \\ p_{k+1} &= p_{k+1/2} + \frac{\delta \mathrm{t}}{2}\dot{p}(q_{k+1}, p_{k+1/2}) \end{aligned}$$

where the notation $\dot{q}(q_k, p_{k+1/2})$ denotes the $\dot{q}$ component of $f(q_k, p_{k+1/2})$ and analogously for $\dot{p}$.

We also consider the implicit Euler method, which is given by the potentially nonlinear equation

$$x_k - \delta \mathrm{t}\, f(x_k) = x_{k-1}$$

that is solved in each time step $k = 1, \ldots, K$.

B Learning Methods

In this section we provide details on the training and implementation of the learning methods tested in this work. The neural network methods used in this work are implemented in PyTorch (Paszke et al., 2019).

B.1 TRAINING

The learning methods considered in this work are trained to approximate the right hand side function for each system. That is, we train a function $\tilde{f}_\theta$ such that $\tilde{f}_\theta(q(t), p(t)) \approx (\dot{q}(t), \dot{p}(t))$. This training is conducted supervised on ground truth snapshots gathered from the underlying models. For each system we randomly sample initial conditions and each of these is then numerically integrated to produce a trajectory. Each trajectory includes state samples (q, p) as well as target derivatives $(\dot{q}, \dot{p})$ used for training.

Table 2: Training parameter values

System	# Train Trajectories	# Eval Trajectories	Time Step Size	# Steps
Spring	10, 500, 1000	30	0.00781, $\div 128$	805
Wave	10, 25, 50	6	0.00049, $\div 8$	10205
Spring Mesh	25, 50, 100	15	0.00781, $\div 128$	805

Table 2 lists the parameters used to generate trajectories for training and evaluation. Training sets of three sizes are generated, each containing the specified number of trajectories. The systems are integrated at the listed time step sizes, but the ground truth data is subsampled further by the factor shown after $\div$ in the table: the integration schemes are run at a smaller time step and intermediate computations are discarded.

Training is done with the Adam (Kingma & Ba, 2014) optimizer for all neural networks, except the kernel which uses standard stochastic gradient descent with learning rate 0.001 and weight decay 0.0001. With the Adam optimizer, no weight decay is used, and the graph network uses a learning rate of 1×10^{-4} while other networks use 1×10^{-3}. The number of training epochs varies based on the target system. On spring, wave, and spring mesh the graph networks were trained for 25, 25, and 150 epochs respectively. The other networks trained for 400, 250, and 800 epochs on the same systems. We train each neural network three times from random initialization, and average errors over all time steps of each evaluation trajectory, for each trained network.

B.2 KNN REGRESSOR

We use a k nearest neighbors regressor to predict the value of the state derivatives, using $k = 1$. With this method $\tilde{f}_\theta(q, p)$ finds the closest matching point in the training set, and uses that point's associated derivatives as its approximation for $(\dot{q}, \dot{p})$. We use the KNN implemented in scikit-learn (Pedregosa et al., 2011).

B.3 KERNEL METHODS

Kernel methods provide a nonparametric regression framework (Schölkopf & Smola, 1998). In this benchmark we consider dot-product kernels of the form $k(x, x') = \eta(\langle x, x' \rangle)$, which can be efficiently implemented using random feature expansions (Rahimi et al., 2007) via the representation

$$\begin{aligned} k(x, x') &= \mathbb{E}_{z \sim \nu}[\rho(\langle x, z \rangle)\rho(\langle x', z \rangle)] \\ &\approx \frac{1}{L} \sum_{l=1}^{L} \rho(\langle x, z_l \rangle)\rho(\langle x', z_l \rangle) \ , \end{aligned}$$

where ν is a rotationally-invariant probability distribution over parameters and $z_l \sim \nu$ iid. The resulting maps $x \mapsto \rho(\langle x, z_l \rangle)$ are *random features*, associated with a shallow neural network with 'frozen' weights. In our experiments, we use $\rho = \text{ReLU}$ and $L = 32768$ (for the single spring system we set $L = 4096$) random features and train using kernel ridge regression.

B.4 MLP

We apply simple MLP networks consisting of two architectures: one "narrow" and one "wide" abbreviated as "mlp-n" and "mlp-w" above. The wide network has two layers (one hidden, with a dimension of 2048), while the narrow network has three layers (two hidden, with dimension 200).

B.5 GRAPH NEURAL NETWORK

We use a graph neural network derived from Pfaff et al. (2020) which we implement using PyTorch Geometric (Fey & Lenssen, 2019). The graph network outputs accelerations for each particle which are used to update estimates for the system's velocity.

This network operates on graphs derived from meshes. We use different static mesh configurations for each system: the single spring system is a mesh consisting of three particles, two of them fixed and the third moving to compute the effects of the oscillation; the wave system is a cyclic graph where each discretized spatial coordinate is connected to its neighbors; the spring mesh system uses the graph defined by the masses and springs directly.

The network itself consists of a paired encoder/decoder pair which transforms the input states into values on the graph, and converts them back for output. These networks are both small MLPs with one hidden layer of dimension 128. On the graph structure another MLP is applied to the vertices and edges; these are again MLPs with one hidden layer of size 128. These graph-acting networks are applied fifteen times, recurrently, and this result is decoded by the decoder network.

C BENCHMARK SYSTEMS

We consider three physical systems: a single oscillating spring, a linear wave equation, and a mesh of damped springs. The ground truth models for each of these systems with classical time integrators are implemented using NumPy (Harris et al., 2020) and accelerated, where possible, with Numba (Lam et al., 2015). When generating ground truth training and evaluation snapshots, we randomly sample initial conditions. Representative initial condition states for each system are illustrated in Figure 1. The ground truth data consist of the values of the state variables (q, p) for each discrete time step, and the associated derivatives $(\dot{q}, \dot{p})$.

C.1 SPRING

We simulate a single one-dimensional oscillating spring. In this system, the spring has zero rest length, and both the oscillating mass and spring constant are set to 1. The spring then exerts a force inversely proportional to the position of the mass q: $\dot{p}(t) = -q$ and $\dot{q}(t) = p$.

The energy of the system is proportional to $q^2 + p^2$ which is the radius of the cycle in phase space. To sample initial conditions, we first sample a radius uniformly, then choose an angle theta uniformly. This produces a uniform distribution over spring system energy levels and starts at an arbitrary point in the cycle. Simulations of the spring system always run through one period. For "in-distribution" training values, the radius is selected in the range $(0.2, 1)$ and "out-of-distribution" radii are chosen from $(1, 1.2)$.

C.2 WAVE

This system closely follows the design in Peng & Mohseni (2016). Consider the wave equation with speed $c = 0.1$

$$\partial_{tt} u = c^2 \partial_{xx} u \,, \tag{3}$$

on a one-dimensional spatial domain $[0, 1)$ with periodic boundary conditions. We represent this second-order system as a first-order system with

$$\begin{bmatrix} \dot{q}(t) \\ \dot{p}(t) \end{bmatrix} = \begin{bmatrix} 0 & I \\ c^2 D_{xx} & 0 \end{bmatrix} \begin{bmatrix} q(t) \\ p(t) \end{bmatrix} , \tag{4}$$

where $D_{xx} \in \mathbb{R}^{n \times n}$ corresponds to the three-point central difference approximation of the spatial derivative ∂_{xx} and the matrices I and 0 are the identity and zero matrix, respectively, of appropriate size. We discretize space into $n = 125$ evenly spaced grid points and evolve the system following the dynamics described above.

Initial conditions are sampled with an initial pulse in the q component centered in at 0.5, in the middle of the space. All initial conditions have zero momentum. The initial pulse is produced by a spline

kernel as described in Peng & Mohseni (2016):

$$s(x) = \frac{10}{p_w} \cdot |x - 0.5| \tag{5}$$

$$h(s) = p_h \cdot \begin{cases} 1 - \frac{3}{2}s^2 + \frac{3}{4}s^3 & \text{if } 0 \le s \le 1 \\ \frac{1}{4}(2-s)^3 & \text{if } 1 < s \le 2 \\ 0 & \text{else} \end{cases} \tag{6}$$

where the width and height of the pulse are scaled by parameters p_w and p_h, respectively. The spline kernel pulse is then $h(s(x))$ for $x \in [0, 1)$, evaluated at the discretized grid points.

For "in-distribution" samples parameters p_w, p_h are both chosen uniformly in the range $(0.75, 1.25)$ and "out-of-distribution" runs sample uniformly from $(0.5, 0.75) \cup (1.25, 1.5)$. All trajectories are integrated until $t = 5$ when the wave has traveled through half a period.

C.3 Spring Mesh

This system manipulates a square grid of particles connected by springs, in a two dimensional space. The particles all have mass 1, and are arranged into a unit grid. Springs are added along the axis-aligned edges and diagonally across each grid square, with rest lengths selected so that the regularly-spaced particles are in a rest position.

In this work we use a 5×5 mesh where the top row of particles is fixed in place. Initial conditions are sampled by choosing a perturbation for the position of each non-fixed spring. These perturbations are chosen as uniform vectors inside a circle with radius 0.35. Out-of-distribution perturbations are chosen uniformly in a ring with inner radius 0.35 and outer radius 0.45. The sampled initial conditions all have zero momentum.

In this system, a spring between particles a and b exerts a force:

$$F_{ab} = -k \cdot \left(\|q_a - q_b\|_2 - \ell_{ab}\right) \frac{q_a - q_b}{\|q_a - q_b\|_2} - \gamma(\dot{q}_a - \dot{q}_b) \tag{7}$$

where ℓ_{ab} is the rest length of the spring, $\gamma = 0.1$ is a parameter controlling the magnitude of an underdamped velocity-based decay, and $k = 1$ is the spring constant.

D Out of Distribution

We also give the results from Table 1 for out of distribution samples in Table 3.

Table 3: Relative slowdowns for equivalent error for learned method. This table follows the same style as Table 1 but instead shows results for out of distribution samples.

Base Integrator		mlp-w		mlp-n		nn-kernel		gn	knn	
		FE	RK4	FE	RK4	FE	RK4	NUL	FE	RK4
Spring	FE	21.6 / 1	21.6 / 1	39.1 / 4	39.1 / 4	38.0 / 1	38.0 / 1	445.9 / 1	153.5 / 16	153.5 / 16
	LF	28.9 / 32	28.9 / 32	74.0 / 4	74.0 / 4	52.1 / 16	52.1 / 16	595.4 / 32	153.5 / 16	116.0 / 1
	RK4	28.9 / 32	28.9 / 32	74.0 / 4	74.0 / 4	52.1 / 16	52.1 / 16	595.4 / 32	153.5 / 16	116.0 / 1
	BE	29.3 / 1	29.3 / 1	74.0 / 4	74.0 / 4	51.5 / 1	51.5 / 1	603.7 / 1	153.5 / 16	116.0 / 1
Wave	FE	97.9 / 8	97.9 / 8	108.1 / 8	108.1 / 8	168.5 / 4	168.5 / 4	1602.9 / 8	383.5 / 8	383.5 / 8
	LF	118.4 / 128	118.4 / 128	130.6 / 128	130.6 / 128	385.4 / 128	385.4 / 128	1937.7 / 128	463.6 / 128	463.6 / 128
	RK4	103.2 / 128	103.2 / 128	113.8 / 128	113.8 / 128	335.9 / 128	335.9 / 128	1688.5 / 128	404.0 / 128	404.0 / 128
	BE	186.4 / 128	186.4 / 128	113.8 / 128	113.8 / 128	387.8 / 64	387.8 / 64	1688.5 / 128	404.0 / 128	404.0 / 128
Spring Mesh	FE	1.6 / 8	1.6 / 8	3.4 / 16	3.4 / 16	2.5 / 16	2.5 / 16	58.8 / 8	107.6 / 16	107.6 / 16
	LF	1.6 / 8	1.6 / 8	3.4 / 16	3.4 / 16	2.5 / 16	2.5 / 16	58.8 / 8	107.6 / 16	107.6 / 16
	RK4	1.6 / 8	1.6 / 8	3.4 / 16	3.4 / 16	2.5 / 16	2.5 / 16	58.8 / 8	107.6 / 16	107.6 / 16
	BE	1.6 / 8	1.6 / 8	3.4 / 16	3.4 / 16	2.5 / 16	2.5 / 16	58.8 / 8	107.6 / 16	107.6 / 16