PHYSICS-DRIVEN MACHINE LEARNING MODELS COUPLING PYTORCH AND FIREDRAKE

Nacime Bouziani & David A. Ham
Department of Mathematics
Imperial College London, London, UK
{n.bouziani18, david.ham}@imperial.ac.uk

ABSTRACT

Partial differential equations (PDEs) are central to describing and modelling complex physical systems that arise in many disciplines across science and engineering. However, in many realistic applications PDE modelling provides an incomplete description of the physics of interest. PDE-based machine learning techniques are designed to address this limitation. In this approach, the PDE is used as an inductive bias enabling the coupled model to rely on fundamental physical laws while requiring less training data. The deployment of high-performance simulations coupling PDEs and machine learning to complex problems necessitates the composition of capabilities provided by machine learning and PDE-based frameworks. We present a simple yet effective coupling between the machine learning framework PyTorch and the PDE system Firedrake that provides researchers, engineers and domain specialists with a high productive way of specifying coupled models while only requiring trivial changes to existing code.

1 INTRODUCTION

Partial differential equations (PDEs) are ubiquitous in science and impact natural sciences and engineering, since most of the laws that govern the dynamics of physical systems are described by partial differential equations. This modelling is most effective when physical systems closely follow the, frequently idealised, asumptions used to derive the PDE. In more complex and realistic scenarios, the error induced by these assumptions can become large. PDE-based machine learning, or physics-driven machine learning, seeks to address this limitation by creating models informed both by the PDE and by observed data. Different discretisation techniques for the PDE part of the model are available. However, among these, the finite element method (FEM) uniquely combines a high degree of geometric and numerical flexibility with a high-level mathematical abstraction which is amenable to a differentiable programming approach. In many cases, the PDE is solved using the finite element method (FEM), a widely used method for solving PDEs, see (Berg & Nyström, 2021; Nguyen & Bui-Thanh, 2022; Costabal et al., 2023).

The use of efficient, composable, and high productivity environments relying on high-level domain specific languages, and code generation is prevalent in machine learning. On the other hand, only few scientific computing tools embrace this design approach. Examples include: PyTorch (Paszke et al., 2019), TensorFlow, (Abadi et al., 2016), and JAX (Bradbury et al., 2018) for machine learning, and Firedrake (Rathgeber et al., 2017), FEniCS (Logg et al., 2012), or FreeFem++ (Hecht, 2012) for finite element methods (FEM), to name but a few. In this work, we extend the high productivity and high performance capabilities of PyTorch and Firedrake to physics-driven machine learning models, enabling scientists and engineers to design, implement and run complex simulations coupling machine learning models implemented in PyTorch and partial differential equations implemented in Firedrake.

2 FIREDRAKE

Firedrake (Rathgeber et al., 2017) is an automated finite element system embedded in Python for the solution of partial differential equations. Firedrake uses the Unified Form Language (UFL) (Alnæs et al., 2014), a domain-specific language to provide high-level representations of finite element problems. The Firedrake system translates the symbolic specification of variational forms of PDEs expressed in UFL into low-level code for assembling the sparse matrices and vectors of the corresponding finite element problem. The dolfin-adjoint automatic differentiation package (Farrell et al., 2013; Mitusch et al., 2019) provides Firedrake with automatically derived tangent-linear and adjoint capabilities analogous to the forward- and back-propagated derivatives available in PyTorch.

3 DIFFERENTIABLE PROGRAMMING

In order to run hybrid models coupling partial differential equations and machine learning, we need to be able to couple the evaluation of both components but also to couple their differentiation as gradient calculation is critical in both machine learning and PDEs. Examples include the use of backpropagation for training neural networks in machine learning, or the use of Newton-type methods for solving PDEs. The use of automatic differentiation to automate the calculation of the derivatives of interest is widely employed in machine learning while only adopted by few PDE-based software. More specifically, there are two approaches considered: tangent-linear mode, also referred to as forward mode, and the adjoint mode, also referred to as reverse mode, of which backpropagation is a special case.

A central idea in differentiable programming is that the implementation of a given function f as a computer program can be represented as a directed acyclic graph whose nodes and edges represent the intermediate calculations to compute f. Another important idea which follows from chain rule is that the computation of a derivative in tangent-linear mode (resp. adjoint-mode) of f can be achieved by traversing that computational graph forward (resp. backward) and evaluate the derivative tangent linear model (resp. adjoint model) of each node on the fly.

For example, let V be a Hilbert space and let u be the solution of a PDE defined by:

$$F(u, m; v) = 0 \quad \forall v \in V \tag{1}$$

where F is the variational form of the PDE, and m is a known parameter. Then, backpropagating through the solution u is equivalent to compute the adjoint model of u, which can be written as:

$$\mathcal{J}_u^*(m; w) = -\frac{\partial F}{\partial m}^* \lambda \tag{2}$$

for all $w \in V^*$ and where $\lambda \in V$ is the solution of the adjoint equation defined as:

$$\frac{\partial F}{\partial u}^* \lambda = w \tag{3}$$

See appendix A.1 for more details. Firedrake and PyTorch are both equipped with AD modules to calculate the tangent-linear and adjoint models of a wide range of operations defined within both frameworks.

4 COUPLING PYTORCH AND FIREDRAKE

Let $\mathcal{F}$ be an operator representing a set of operations implemented in Firedrake (e.g. solving a PDE, or assembling a variational form). The use of the Firedrake finite element system for building physics-driven machine learning models within PyTorch necessitates the ability for PyTorch to go through the computational graph of $\mathcal{F}$ to evaluate it and automatically differentiate it. Note that the other side of the coupling, namely using PyTorch operations within Firedrake is already possible as introduced in Bouziani & Ham (2021). Also, similar work is conducted to compose Firedrake with JAX programs in Yashchuk (2022).

In this work, we build a custom Firedrake operator within *torch.autograd* (Paszke et al., 2017) to represent $\mathcal{F}$. The computational graph evaluation and differentiation is delegated to *torch.autograd* for PyTorch-based nodes, and to *dolfin-adjoint* for Firedrake-based nodes. This simple yet powerful high-level coupling, illustrated in figure 1, results in a composable environment that benefits from the full armoury of advanced features and AD capabilities both frameworks offer whilst maintaining separation of concerns.
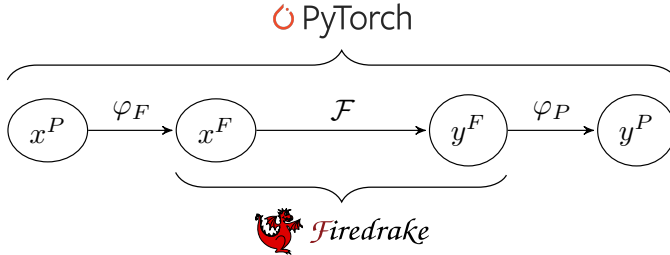

Figure 1: Subgraph of the PyTorch computational graph containing Firedrake operations of interest represented by $\mathcal{F}$, where P refers to PyTorch variables and F to Firedrake variables. φ_F and φ_P represent the casting of a PyTorch tensor to a Firedrake Function and vice versa.

This coupling is facilitated by the fact that the core data object in each case is merely a tensor of values. However, machine learning frameworks do not inherently attach further structure to this tensor. In contrast, FEM packages such as Firedrake explicitly associate state tensors with computational domain information. Consequently, the definition of Firedrake tensors induces the required mapping φ_F and φ_P to convert to and from PyTorch, see appendix A.2 for more details. Another consequence is that the Firedrake tensor representation facilitates the use of appropriate inner products, such as L^p and H^p, in addition to the ℓ^p-norm ubiquitous in machine learning. This is critical to guarantee mesh independence for PDE-constrained optimisation problems, see (Schwedes et al., 2017).

Listing 1 demonstrates how one can build physics-driven models using PyTorch and Firedrake. The fact that both libraries are embedded in Python facilitates the implementation of such models. More importantly, note that coupling both frameworks only requires a one-line change to existing codes as highlighted in listing 1 (cf.line 12), which makes the implementation straightforward from a user perspective. Line 9 defines the functional $\mathcal{F}$ as a function of given control(s), which enables to only traverse the relevant part of $\mathcal{F}$'s computational graph needed to differentiate $\mathcal{F}$ with respect to the given control(s). We also extend dolfin-adjoint to relax the assumption that it owns the final quantity of interest that is differentiated, thereby enabling backpropagation computation to start from a PyTorch tensor. The operator G defined in line 12 wraps the operator F to act on PyTorch tensors as illustrated in line 15 with x^P and y^P.

5 EXAMPLE: 2D HEAT CONDUCTIVITY INVERSE PROBLEM

We showcase our implementation on a simple heat conductivity problem modelled via the following time-independent heat equation: Find the temperature field $u \in V$ such that:

$$-\nabla \cdot (e^\kappa \nabla u) = f \quad \text{in } \Omega$$
$$u = 0 \quad \text{on } \partial\Omega \tag{4}$$

where $\Omega \subset \mathbb{R}^2$ is an open and bounded domain, $\kappa \in V$ is the conductivity, $f \in V$ the source term, and with V a suitable function space (i.e. $V = H_0^1(\Omega)$). We want to solve the inverse problem driven by equation 4 by learning the inverse operator:

$$\hat{\kappa} : u^{obs} \to \kappa \tag{5}$$

where u^{obs} refers to an observed temperature field and is modelled by: $u^{obs} = u(\kappa) + \varepsilon$, with $u(\kappa)$ the solution of equation 4 for a given κ, and ε the observation noise. We use a model-constrained

Listing 1: Outline of backpropagation through Firedrake using torch_operator

```python
import torch
import firedrake as fd
import firedrake.ml as fd_ml
import firedrake_adjoint as fda

...

# Defined reduced functional F with respect to control(s)
F = fda.ReducedFunctional(y_F, Control(x_F))

# Define the coupling operator: G := φ_P ∘ F ∘ φ_F
G = fd_ml.torch_operator(F)

# Apply the coupling operator to a torch.Tensor x^P
y_P = G(x_P)

# Backpropagate through G: calculate J*_{y^P}(x^P, w^P)
w_P = ...
y_P.backward(w_P)
```

deep neural network approach as introduced in Nguyen & Bui-Thanh (2022). More precisely, let κ_θ be a a neural network based model with θ the model parameters. We train κ_θ to learn the inverse operator defined in equation 5 by minimising the following loss function:

$$J(u^{obs}, \kappa^{exact}; \theta) = \frac{1}{2}\|\kappa_\theta(u^{obs}) - \kappa^{exact}\|^2_{L^2(\Omega)} + \frac{\alpha}{2}\|u(\kappa_\theta(u^{obs})) - u^{obs}\|^2_{L^2(\Omega)} \qquad (6)$$

where κ^{exact} refers to the exact conductivity for a given observable u^{obs}. In Nguyen & Bui-Thanh (2022), authors studied heat conductivity test case using a single hidden layer model, here we consider a convolutional neural network architecture (CNN) for learning the conductivity. Two Firedrake coupling operators are used during training: one for computing the PDE solution $u(\cdot)$, and the other for assembling the L^2-loss as the choice of norm is critical for mesh-independence. For training, we generate n random fields to form the training split: $\{\kappa_i^{exact}, u_i^{obs}\}_{1 \leq i \leq n}$, and average the loss defined in equation 6 across training samples. For evaluation, we generate data in the same way and average the error across test samples. More details about this test case can be found in appendix A.3. The exact version of Firedrake used in this paper is archived in The Firedrake developers (2023) while the code used to run this example is archived in Bouziani & Ham (2023).
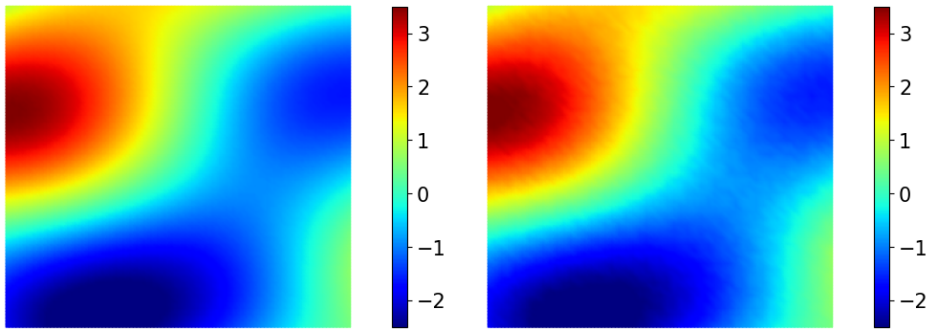

Figure 2: Heat conductivity as a function of position (x, y): exact conductivity (left), conductivity reconstructed from observed data $\kappa_\theta(u^{obs})$ by the CNN model (right).

REFERENCES

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2), Mar 2014. ISSN 0098-3500. doi: 10.1145/2566630. URL `https://doi.org/10.1145/2566630`.

Jens Berg and Kaj Nyström. Neural networks as smooth priors for inverse problems for PDEs. *Journal of Computational Mathematics and Data Science*, 1:100008, September 2021. ISSN 2772-4158. doi: 10.1016/j.jcmds.2021.100008. URL `https://www.sciencedirect.com/science/article/pii/S2772415821000043`.

Nacime Bouziani and David A. Ham. Escaping the abstraction: a foreign function interface for the Unified Form Language [UFL]. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, November 2021. URL `http://arxiv.org/abs/2111.00945`. arXiv: 2111.00945.

Nacime Bouziani and David A. Ham. Physics-driven machine learning models, March 2023. URL `https://github.com/nbouziani/physics-driven-ml`.

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

Francisco Sahli Costabal, Simone Pezzuto, and Paris Perdikaris. Δ-PINNs: physics-informed neural networks on complex geometries. February 2023. URL `https://openreview.net/forum?id=5P96KWeULzE`.

Patrick E. Farrell, David A. Ham, Simon F. Funke, and Marie E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, January 2013. ISSN 1064-8275, 1095-7197. doi: 10.1137/120873558. URL `http://arxiv.org/abs/1204.5577`. arXiv: 1204.5577.

Frédéric Hecht. New development in freefem++. *Journal of Numerical Mathematics*, 20(3-4):1–14, 2012. doi: 10.1515/jnum-2012-0013. URL `https://hal.sorbonne-universite.fr/hal-01476313`. Publisher: De Gruyter.

Michael Hinze, René Pinnau, Michael Ulbrich, and Stefan Ulbrich. *Optimization with PDE Constraints*, volume 23 of *Mathematical Modelling: Theory and Applications*. Springer Netherlands, Dordrecht, 2009. ISBN 978-1-4020-8838-4 978-1-4020-8839-1. doi: 10.1007/978-1-4020-8839-1. URL `http://link.springer.com/10.1007/978-1-4020-8839-1`. ISSN: 1386-2960.

Anders Logg, Kent-Andre Mardal, and Garth Wells (eds.). *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Lecture Notes in Computational Science and Engineering. Springer-Verlag, Berlin Heidelberg, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8. URL `https://www.springer.com/gp/book/9783642230981`.

Sebastian Mitusch, Simon Funke, and Jørgen Dokken. dolfin-adjoint 2018.1: automated adjoints for FEniCS and Firedrake. *Journal of Open Source Software*, 4(38):1292, June 2019. ISSN 2475-9066. doi: 10.21105/joss.01292. URL `http://joss.theoj.org/papers/10.21105/joss.01292`.

Hai V. Nguyen and Tan Bui-Thanh. TNet: A Model-Constrained Tikhonov Network Approach for Inverse Problems, September 2022. URL `http://arxiv.org/abs/2105.12033`. arXiv:2105.12033 [cs, math, stat].

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. *31st Conference on Neural Information Processing Systems (NIPS 2017)*, pp. 4, 2017.

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 *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://papers.nips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html`.

Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Transactions on Mathematical Software*, 43(3):1–27, January 2017. ISSN 0098-3500, 1557-7295. doi: 10.1145/2998441. URL `https://dl.acm.org/doi/10.1145/2998441`.

Tobias Schwedes, David A. Ham, Simon W. Funke, and Matthew D. Piggott. *Mesh Dependence in PDE-Constrained Optimisation: An Application in Tidal Turbine Array Layouts*. SpringerBriefs in Mathematics of Planet Earth, Mathematics of Planet Earth Collection. Springer International Publishing, 2017. ISBN 978-3-319-59482-8. doi: 10.1007/978-3-319-59483-5. URL `https://www.springer.com/gp/book/9783319594828`.

The Firedrake developers. Software used in 'Physics-driven machine learning models coupling PyTorch and Firedrake', feb 2023. URL `https://doi.org/10.5281/zenodo.7623009`.

Ivan Yashchuk. Bringing PDEs to JAX with forward and reverse modes automatic differentiation. June 2022. URL `https://openreview.net/forum?id=nEPNoiGsU3`.

A   APPENDIX

A.1   ADJOINT MODEL OF PDE SOLUTIONS

Let V and M be Hilbert spaces, and let $u \in V$ be the solution of the parametrised partial differential equation defined as:

$$F(u, m; v) = 0 \quad \forall v \in V \tag{7}$$

with F the variational form of the PDE, and $m \in M$ the control value. We assume that the PDE defined by F yields a unique solution for any control value $m \in M$. Hence, we can define the PDE solution operator $u(\cdot) : M \to V$ that maps any control value m to the corresponding solution $u(m)$ satisfying equation 7. Note that the PDE solution $u(m)$ is generally not known explicitly but can be approximated using the finite element method.

We assume that the linear form F is continuously Fréchet differentiable and that the linearised PDE operator $\frac{\partial F}{\partial u}$ is invertible. It follows, using implicit function theorem, that the solution operator $u(\cdot)$ is continuously Fréchet differentiable, see (Hinze et al., 2009, Sec. 1.4.2). Let V^* and M^* be the dual spaces of V and M, i.e. the spaces of all continuous linear functionals on V and M, respectively. Backpropagating through the PDE solution $u(m)$ equates to computing its adjoint model $\mathcal{J}_u^* : V^* \to M^*$ defined as:

$$\mathcal{J}_u^*(m; w) = \frac{du}{dm}^* w \quad \forall w \in V^* \tag{8}$$

where $\frac{du}{dm}^* \in \mathcal{L}(V^*, M^*)$ is the adjoint of the Gâteaux derivative of u with respect to m. While the expression of $\frac{du}{dm}^*$ is not known explicitly, given that $u(m)$ is the PDE solution, we can still compute it using equation 7. More precisely, given that we have $F(u(m), m; v) = 0$ for all $m \in M$, it follows that the derivative $\frac{dF}{dm}$ must be zero. Therefore, applying chain rule yields:

$$\frac{\partial F}{\partial u} \frac{du}{dm} + \frac{\partial F}{\partial m} = 0 \tag{9}$$

which leads to

$$\frac{du}{dm} = -\frac{\partial F}{\partial u}^{-1} \frac{\partial F}{\partial m} \tag{10}$$

since we assumed $\frac{\partial F}{\partial u}$ to be invertible. Combining equations 10 and 8, we finally obtain for all $w \in V^*$:

$$\mathcal{J}_u^*(m; w) = -\frac{\partial F}{\partial m}^* \lambda \tag{11}$$

where $\lambda \in V$ is the solution of the adjoint equation defined as

$$\frac{\partial F}{\partial u}^* \lambda = w \tag{12}$$

We refer the interested reader to (Schwedes et al., 2017) and (Hinze et al., 2009, Sec. 1.6) for more details.

A.2   CUSTOM MAPPINGS

Depending on the user case, different representations can be adopted to represent finite element functions in PyTorch. For example, one can feed a neural network with the values of a finite element function on a set of points, e.g on a uniform Cartesian grid for CNN-based architectures or a more general grid for graph neural networks. This representation is independent of the mappings φ_F and φ_P illustrated in figure 1 which merely casts tensors from one framework to the other without altering their shape. This consideration is rather an operation that happens either in Firedrake and/or PyTorch.

In particular, complex representations of finite element functions can be lifted at the level of the definition of the mesh and function spaces in Firedrake, as it inherently provides a richer space representation than PyTorch.

A.3 2D HEAT CONDUCTIVITY INVERSE PROBLEM: IMPLEMENTATION DETAILS

We generate a synthetic dataset $\{\kappa_i^{exact}, u_i^{obs}\}_{1 \leq i \leq n}$ for training and test using the following procedure:

- Randomly generate parameter of interest $\{\kappa_i\}_{1 \leq i \leq n}$
- Compute for each κ_i the corresponding solution of the forward problem (PDE): $u(\kappa_i)$
- Add noise to form the observables: $u_i^{obs} = u(\kappa_i) + \varepsilon_i \quad \forall i \in [|1, n|]$, e.g. $\varepsilon_i \in \mathcal{N}(0, 1)$.

This procedure is not specific to a given PDE and can be adapted to a wide range of inverse problems. For the heat conductivity example, we generate 500 training samples and 100 test samples. For training, we average the loss defined in equation 6 over the n training samples, i.e. we have:

$$\mathcal{L} = \frac{1}{2n} \sum_{i=1}^{n} \left(\|\kappa_\theta(u_i^{obs}) - \kappa_i^{exact}\|_{L^2(\Omega)}^2 + \alpha \|u(\kappa_\theta(u_i^{obs})) - u_i^{obs}\|_{L^2(\Omega)}^2 \right) \tag{13}$$

For evaluation, we adopt the evaluation metric used in Nguyen & Bui-Thanh (2022), i.e. we report the average relative error on the m test samples:

$$R = \frac{1}{m} \sum_{i=1}^{m} \frac{\|\kappa_\theta(u_i^{obs}) - \kappa_i^{exact}\|_{L^2(\Omega)}^2}{\|\kappa_i^{exact}\|_{L^2(\Omega)}^2} \tag{14}$$

The heat conductivity example is mostly designed for demonstrating the criticality of coupling PyTorch and Firedrake to design, implement, and run complex physics-driven machine learning models in a highly productive way. Despite neither having a baseline to compare with nor an available dataset to report the performance of our trained CNN architecture, we release the dataset we generated for this example for sake of future comparisons. On this synthetic dataset, the average relative L^2-error R on the test split is 17.68%. This result is not directly comparable with the analogous example presented in Nguyen & Bui-Thanh (2022) as the dataset and the problem formulation differ. However, given the close similarity between both test cases and the significant gap in performance in favour of our model, we argue that this can be explained by the fact that we have considered a more complex architecture.

We also provide the base implementation for implementing physics-driven machine learning models using PyTorch and Firedrake for arbitrarily defined PDE-based systems in Firedrake by providing support for data generation, training, and evaluation. The corresponding package can be found in Bouziani & Ham (2023).