underPINN: A Modular and Scalable JAX-Based Framework for Physics-Informed Machine Learning

Anonymous authors
Paper under double-blind review

Abstract

Physics-informed neural network (PINN) libraries have proliferated across PyTorch, JAX, and Julia, and cross-framework speed comparisons between them are increasingly common. We show that these comparisons are surprisingly fragile. Framework defaults that never appear in a problem's configuration file — weight normalization applied to every hidden layer, renormalization of the network's time input, dimension-restricted differential operators — silently violate the "same architecture, same physics" assumption such comparisons rest on. Separately, we find that a single-run PINN ablation can reverse its conclusion under an identical random seed and identical code, as GPU floating-point non-determinism compounds through adaptive-collocation training dynamics.

We present `underPINN`, an open-source JAX-native PIML framework, and use it as a controlled instrument for rigorously matched comparison. `underPINN` compiles each problem's optimization step — forward evaluation, automatic differentiation, and the optimizer update — once with `jax.jit` and reuses that executable every epoch; an optional `jax.lax.scan`-fused path yields no measurable further speedup once the step is compiled, which we report. Against eager-mode PyTorch on three matched reimplementations the compiled path is 1.03–$17.4\times$ faster, with the speedup tracking the ratio of per-epoch dispatch overhead to per-step compute rather than problem complexity. Against NVIDIA PhysicsNeMo Sym (eight matched problems) and `jinns` (six), `underPINN` is faster on every problem — but only after the confounds above are identified and corrected, which we document as guidance for the field rather than a criticism of any one library.

Through one configuration interface the framework also exposes Fourier-feature embeddings, gated MLPs, FBPINN domain decomposition, residual-adaptive collocation, a deterministic DEIM-inspired collocation selector (QR-DEIM-R), and a Levenberg–Marquardt-damped Gauss–Newton step. We test these across multiple seeds rather than assume they help, and report what the tests return. Residual-adaptive collocation improves accuracy only where a localized, uncapped feature exists to exploit, not on smooth or dissipation-capped problems. QR-DEIM-R does not reproduce the gains published for QR-DEIM-based collocation on any of the six problems we run it on, and tuned Adam matches or beats the Gauss–Newton step on both accuracy and wall-clock. Our nine problems span ODEs, elliptic and parabolic PDEs, 3-D pipe flow, and 1-/2-D compressible shock flows, and we report the failures alongside the successes — including a 1-D blast wave whose relative L^2 error plateaus near 33%. The compressible cases are solved data-free, from the governing equations and boundary conditions alone with no reference solution supplied during training: among them a Mach 3 viscous shock/boundary-layer-interaction ramp, and Toro Test 3, whose 10^5 initial pressure ratio sits four orders of magnitude above the Sod- and Lax-type Riemann problems usual in the PINN shock literature.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Physics-Informed Neural Networks (PINNs) embed the governing PDEs and their boundary and initial conditions in a network's loss (Raissi et al., 2019; Karniadakis et al., 2021; Cuomo et al., 2022), and targeted fixes now exist for their known failure modes: Fourier features against spectral bias (Rahaman et al., 2019; Wang et al., 2021b), gated architectures against conflicting residual and boundary gradients (Wang et al., 2021a; 2022), FBPINN domain decomposition for large or complex domains (Moseley et al., 2023), and residual-adaptive collocation (RAR, RAD, RBA) (Lu et al., 2019; Wu et al., 2023; Anagnostopoulos et al., 2024), alongside learned-density (Tang et al., 2023) and QR-DEIM-based (Celaya et al., 2025) point selection. Appendix A.1 reviews this work in more detail.

As PINN methodology has matured, supporting software has proliferated – DeepXDE (Lu et al., 2019), NVIDIA PhysicsNeMo (NVIDIA Corporation, 2024), NeuralPDE.jl (Zubov et al., 2021), `jinns` (Gangloff & Jouvin, 2024) – making cross-framework comparison routine, but two difficulties recur. First, *dispatch overhead*: a PINN step is a cheap, high-order-derivative computation repeated for tens of thousands of epochs, and dispatching each one through Python control flow can dominate wall-clock time. JAX/XLA (Sabne, 2020) offers a route around this, and several JAX PINN codebases already take it (`jinns`, JAX-PI (Wang et al., 2023), PINNs-JAX (Akbarian Bafghi & Raissi, 2024), DeepXDE's JAX backend); what we did not find among them was one framework combining a compiled training loop with domain decomposition, gated architectures and adaptive collocation as configuration-selected components, which is the gap `underPINN` fills. Second, matched cross-framework comparison is itself error-prone: frameworks configured with the "same" network can differ in defaults that never surface in a configuration file. Section 3.3 finds one in PhysicsNeMo (weight normalization on every hidden layer), Section 3.4 two in `jinns` (time-input renormalization; a convective-term operator hard-coded to two dimensions), and Section 3.8 shows a single-run adaptive-collocation ablation reversing which method looks best under an identical seed and identical code, through GPU floating-point non-determinism.

We present `underPINN`, an open-source, JAX-native framework (Bradbury et al., 2018) that decouples the compiled execution engine from the PDE, geometry and network definitions, and use it as a controlled instrument for matched comparison. `jax.jit` traces each problem's optimization step once and reuses it every epoch; against eager-mode PyTorch (Paszke et al., 2019) on three matched reimplementations this is 1.03–$17.4\times$ faster, largest where per-epoch cost is dispatch-bound. That isolates eager-versus-compiled execution, not JAX-versus-PyTorch: against `torch.compile` (Ansel et al., 2024) the margin narrows substantially but does not close, and an optional `jax.lax.scan`-fused path helps only where per-step work is small ($1.15\times$ on Pipe Flow) and is slower on all three problems tested once its compilation is counted (Section 3.2). The framework also exposes FBPINN decomposition, Fourier features, gated MLPs, RAR-D/RAD/RBA collocation, QR-DEIM-R collocation and Gauss–Newton training through one configuration interface; we test the last two against their established baselines (Sections 3.8, 3.9) and find that neither improves on them. Table 4 (Appendix A.2) lists the nine benchmark problems, from 1-D to 3-D, steady and unsteady, smooth and shock-dominated.

2 FRAMEWORK ARCHITECTURE

Many open-source PINN implementations couple the PDE definition, network forward pass, and optimization loop tightly, simplifying canonical setups but making it harder to add custom logic (adaptive resampling, dynamic loss weighting, domain decomposition) without risking broken JIT compilation or reintroduced dispatch overhead. We instead separate `underPINN` into five loosely coupled components (Appendix A.2, Figure 2): configuration/resource management, domain formulation and physical models, network architectures, solver execution, and output/post-processing – keeping geometric sampling, governing-equation definitions, and network architecture independent of the compiled execution engine, so changing the physics or the network needs no change to the optimization loop.

2.1 COMPILED EXECUTION: THE JIT-FUSED TRAINING STEP

The main cost this design targets is host-GPU dispatch latency: Python-level for-loops issue a separate dispatch per epoch, which can dominate wall-clock time when the gradient step itself is cheap and repeated for thousands of epochs. We address this with JAX's functional transformations: the forward pass, physics-informed loss, gradient (*jax.value_and_grad*), and optimizer update (Optax) are expressed as one stateless functional block, traced and compiled once via *jax.jit*, so every subsequent epoch reuses the same compiled executable and the host issues one lightweight dispatch per epoch rather than retracing. This per-epoch `jax.jit` path is the default for every evaluator in this paper and accounts for essentially all of the wall-clock speedup reported later.

We also provide an optional *jax.lax.scan*-fused path, wrapping the same block in `jax.lax.scan` so the compiler lowers an entire multi-epoch budget into one kernel, removing host dispatch for the whole run rather than once per epoch. On the three problems tested against it (Section 3.2) its steady-state step is indistinguishable from per-epoch `jax.jit` on the two compute-heavier problems and $1.15\times$ faster on Pipe Flow, whose per-step work is small enough that the residual launch cost of each already-compiled call is a visible fraction of it. The dominant overhead, per-epoch dispatch and retracing, is already removed by `jax.jit`; what `jax.lax.scan` removes beyond that is small, and its one-time compilation more than repays it within a single $5{,}000$-epoch run on all three problems.

Fusion also has a structural limit that matters for everything in Section 2.3. Adaptive resampling (RAR-D, RAD, QR-DEIM-R) evaluates candidate pools and draws new points in host-side NumPy/SciPy, outside any compiled boundary, and changes the collocation arrays the step consumes. A `jax.lax.scan` loop can therefore fuse at most the stretch of epochs *between* two resampling events, never across one; for a configuration with static collocation it can fuse the whole budget. The per-epoch `jax.jit` path has no such restriction, which is a second reason, beyond the measurements above, that it is the default.

2.2 NETWORK ARCHITECTURES AND TRAINING STRATEGIES

Beyond a standard $\tanh$-activated MLP, `underPINN` provides GatedMLP (gate-blended dual input encoders re-injected at every layer, against gradient pathologies in stiff PDEs, following Wang et al. (2022)), FourierMLP (trainable-scale random Fourier input embedding against spectral bias, following Wang et al. (2021b)), FBPINN domain decomposition (overlapping subdomains via partition-of-unity windows, following Moseley et al. (2023)), and a lower-parameter HybridAttention/SimpleGate modulation block (no isolating ablation presented here). It also supports windowed time-marching (warm-started temporal windows for long-horizon 3-D unsteady problems, integrated with checkpointing below); trainable artificial viscosity ($\epsilon = \mathrm{softplus}(\eta)$, $\eta \in \mathbb{R}$ trainable jointly optimized, as an alternative to a fixed global dissipation coefficient – Section 3.8 tests it directly: without a regularizer on ϵ it makes Toro Test 3 worse); log-parameterization for positivity-constrained inverse discovery; and a Levenberg–Marquardt-damped Gauss–Newton step (Section 3.9). Appendix A.7 gives the full motivation and mechanism for each.

2.3 ADAPTIVE COLLOCATION, CHECKPOINTING, AND RESTART

Static collocation sampling (e.g. Latin Hypercube) can under-resolve transient features or evolving boundary layers. `underPINN` supports RAR-D/RAD (periodic residual-magnitude-proportional resampling, following Wu et al. (2023)), RBA (element-wise residual-magnitude loss weighting, following Anagnostopoulos et al. (2024)), and QR-DEIM-R, a deterministic alternative to RAD's magnitude-proportional draw that combines the column-pivoted-QR selection rule from Q-DEIM (Drmač & Gugercin, 2016) with leverage-score-weighted sampling (Drineas et al., 2006). It sits in the same family as Celaya et al.'s QR-DEIM collocation strategies (Celaya et al., 2025) but is not a reimplementation of either of them; we test it here rather than adopt it on the strength of those reported gains. Section 3.8 evaluates RAD and QR-DEIM-R against uniform sampling on six problems; resampling runs via NumPy/SciPy outside the compiled `jax.jit` boundary, called periodically rather than every epoch. Separately, a configuration-hash-based restart mechanism (MD5 fingerprint of the full YAML configuration) resumes an interrupted run from its last checkpoint when re-invoked unchanged, and starts fresh on a mismatch (untested under an actual multi-node/preemption workload); XLA is configured for on-demand rather than pre-allocated GPU memory so concurrent

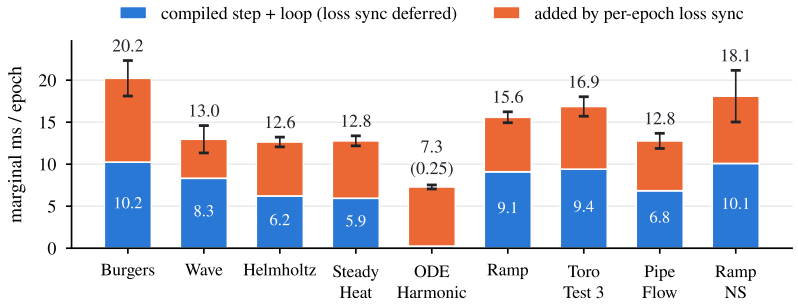


Figure 1: Per-epoch training cost per evaluator (three seeds; marginal cost from 500 to 3,000 epochs, excluding compilation). Bar height: the loop as run, reading the loss back to the host every epoch (error bar: std). Blue: the same loop with that read-back deferred, final losses unchanged (ODE Harmonic's in parentheses). Orange: the difference.

jobs can share a device. Appendix A.7 gives QR-DEIM-R's construction and the checkpointing mechanism in full.

3 Experimental Benchmarks

We evaluate `underPINN` on the nine problems of Table 4 on a single NVIDIA GB10 GPU, using the per-epoch `jax.jit` path of Section 2.1 throughout.

3.1 Compiled Execution and Computational Throughput

Figure 1 shows per-epoch cost across all nine problems, and splits it in two. As the training loops run by default, cost ranges from 7.3 ± 0.2 ms/epoch (ODE Harmonic) to 20.2 ± 2.1 (Burgers). Much of that is not the compiled step: every loop reads each epoch's loss back to the host, which blocks until the GPU finishes, so the host cannot queue the next step meanwhile. Deferring that read-back to the end of training (final losses unchanged) removes 36–53% of the per-epoch cost on eight problems and 97% on ODE Harmonic, whose compiled step takes only 0.25 ms. What remains tracks computational cost rather than dimensionality: cheapest are Steady Heat and Helmholtz (5.9, 6.2 ms/epoch), dearest Burgers (large collocation pool) and Ramp NS (second-order viscous terms) at 10.2 and 10.1. The per-epoch read-back is a property of the library's current training loops, not of compilation, and removing it is an improvement we have not yet made; the cross-framework timings of Sections 3.2–3.4 come from separate matched scripts that read the loss back only once, after the loop, so this overhead does not enter them.

3.2 Comparison Against Eager-Mode PyTorch

To test the eager-vs-compiled comparison directly, we implemented three representative problems – 1-D Burgers, 3-D Pipe Flow, 2-D Ramp NS (SBLI) – independently in eager-mode PyTorch, JAX `jax.jit`, and JAX `jax.lax.scan`, matched to the corresponding evaluator and three-seed-verified (Appendix A.5, Table 8). Speedup tracks the ratio of dispatch overhead to per-step compute, not problem complexity: $\sim 1.03\times$ for the already compute-bound Burgers, up to 15–17$\times$ for the kernel-launch-latency-dominated Pipe Flow, $\sim 14\times$ for Ramp NS between the two; PyTorch eager's own seed-to-seed std (0.18–0.71 ms) is 3–40$\times$ larger than either JAX variant's (0.01–0.07 ms), and its *between-session* drift is larger still (Appendix A.5). `jax.lax.scan`'s steady-state step is $1.15\times$ faster than plain `jax.jit` on Pipe Flow and indistinguishable on the other two, but slower on all three once its compilation is counted, so `jax.jit` is the suite default (Section 2.1). A natural objection is that eager PyTorch is a weak baseline, since `torch.compile` (Ansel et al., 2024) targets the same dispatch overhead. On Burgers it does not close the gap, for a reason worth stating: `torch.compile` cannot capture the `autograd.grad` double backward a PINN residual requires, and the `torch.func` rewrite it *can* capture is itself slower than eager PyTorch's classic path (Appendix A.5).

3.3 COMPARISON AGAINST NVIDIA PHYSICSNEMO SYM

To compare against an independently engineered framework rather than our own reimplementations, we ran NVIDIA PhysicsNeMo Sym (NVIDIA Corporation, 2024) on eight of the nine problems (all but the inviscid Ramp), matching physics, network depth and width, collocation counts and per-step batching as closely as its constraint-based API allows, and scoring both frameworks against the same reference solution (configuration audit in Appendix A.9).

Table 1: `underPINN` vs. PhysicsNeMo Sym: training throughput and final accuracy, same GPU, same reference solution per problem. underPINN throughput/accuracy are mean ± std over five seeds (Burgers, Toro Test 3, Ramp NS) or three seeds (Wave, Steady Heat, Helmholtz, ODE Harmonic); PhysicsNeMo is mean ± std over three seeds on every problem, each run after one discarded warm-up run; underPINN's Pipe Flow row is single-run. **Bold** marks the more accurate framework where the two differ by more than their seed-to-seed spread; on Burgers they do not.

Problem	underPINN ms/ep	PhysicsNeMo ms/ep	Speedup	underPINN rel. L^2	PhysicsNeMo rel. L^2
1-D Burgers (5,000 ep)	8.50 ± 0.06	26.71 ± 0.11	$3.14\times$	0.219 ± 0.076	0.225 ± 0.063
1-D Wave (5,000 ep)	0.64 ± 0.00	5.99 ± 0.10	$9.35\times$	$\mathbf{0.011 \pm 0.001}$	0.405 ± 0.043
2-D Steady Heat (5,000 ep)	0.42 ± 0.00	5.95 ± 0.15	$14.2\times$	$\mathbf{0.0025 \pm 0.0003}$	0.0158 ± 0.0010
2-D Helmholtz (10,000 ep)	1.17 ± 0.03	5.52 ± 0.06	$4.72\times$	$\mathbf{(5.9 \pm 0.2) \times 10^{-4}}$	1.085 ± 0.003
ODE Harmonic (3,000 ep)	0.49 ± 0.02	6.70 ± 0.15	$13.7\times$	0.771 ± 0.060	$\mathbf{0.318 \pm 0.217}$
3-D Pipe Flow (70,000 ep)	20.44	26.47 ± 0.04	$1.29\times$	–	0.179 ± 0.006 / 0.309 ± 0.011 (u/p)
1-D Toro Test 3 (5,000 ep)	10.63 ± 0.52	13.12 ± 0.08	$1.23\times$	$\mathbf{0.445 \pm 0.016}$	0.674 ± 0.071
2-D Ramp NS (30,000 ep)	11.33 ± 0.32	18.44 ± 0.05	$1.63\times$	$\mathbf{0.068 \pm 0.001}$	0.0707 ± 0.0013

Matching the architecture exposed a confound: PhysicsNeMo's `FullyConnectedArch` defaults to weight normalization on every hidden layer (Salimans & Kingma, 2016), which our first case scripts missed, so every early "plain MLP vs. plain MLP" comparison was silently mismatched. Every number in Table 1 uses `weight_norm=False`; that correction is itself not neutral for accuracy (below). Every PhysicsNeMo run also uses `jit=false`, because its default nvFuser activation fusion fails to compile on this GPU, so its timings are for its unfused path and would be expected to be lower where the default compiles (Appendix A.9).

`underPINN` is faster on all eight problems (Table 1), by $1.23\times$ (Toro Test 3) to $14.2\times$ (Steady Heat), least on the three problems with the heaviest per-step work (Toro Test 3, Pipe Flow, Ramp NS), consistent with the dispatch-overhead account of Section 3.2. Multi-seeding PhysicsNeMo mattered for throughput too: an earlier single-run version of this table had it about $1.5\times$ slower on Toro Test 3 and Ramp NS, on identical code and configuration, overstating those two speedups ($1.82\times$ and $2.47\times$); the other six reproduced to within 2%.

Speed is only half the table. On six of the seven problems scored both ways, `underPINN` is faster *and* at least as accurate; on five of them the accuracy gap exceeds both frameworks' seed-to-seed spread, and on Burgers it does not (0.219 ± 0.076 vs. 0.225 ± 0.063). The margin is largest on Wave and Helmholtz, where PhysicsNeMo's matched configuration does not converge well (relative L^2 of 0.40/1.08) while `underPINN`'s does, and the two failures have different causes. Helmholtz's is our own doing: it is caused by the weight-normalization standardization described above, and restoring PhysicsNeMo's native `weight_norm=True` improves it $69\times$ (to 0.016) at a 4% throughput cost (Appendix A.9), leaving `underPINN` about $27\times$ more accurate rather than three orders of magnitude. Wave's is not: restoring weight normalization leaves it unchanged (0.41), so that failure belongs to PhysicsNeMo's matched configuration rather than to our standardization. The seventh, ODE Harmonic, is the opposite: $13.7\times$ faster per epoch but relative $L^2 = 0.771 \pm 0.060$ against PhysicsNeMo's 0.318 ± 0.217 at the shared $3{,}000$-epoch budget. PhysicsNeMo's own result on this problem is strongly seed-dependent (0.011, 0.457 and 0.485 over three seeds; the single run an earlier version of this table reported, 0.043, was close to its best case), but `underPINN` trails its mean regardless. This is not a setup mismatch but `underPINN`'s untuned default Adam (10^{-3}) failing to converge an oscillatory solution spanning under two periods ($T{=}5$, period $2\pi/\omega \approx 3.14$) in this few epochs, the failure mode Section 3.9 documents and resolves with a higher learning rate; we did not retune here, to keep the configuration identical to PhysicsNeMo's. `underPINN`'s Pipe Flow accuracy at this matched scale ($100{,}000$ points, $70{,}000$ epochs) was not measured; Section 3.5 reports it on `underPINN`'s own configuration, which is not directly comparable.

3.4 A PRELIMINARY COMPARISON AGAINST JINNS

We also ran `jinns` (Gangloff & Jouvin, 2024) (version 1.10.0) on six problems and DeepXDE's JAX backend (Lu et al., 2019) (version 1.15.0) on four of them, matching physics, network depth and width, collocation counts, boundary-condition semantics and optimizer (plain Adam with a cosine schedule, rather than `jinns`' natural-gradient optimizer, to avoid confounding the comparison with an optimizer choice), and scoring all against the same closed-form solutions. Appendix A.10 gives the configuration audit, including two `jinns` API details that would have silently broken the comparison had we trusted docstrings over source.

Table 2: `underPINN` vs. jinns vs. DeepXDE's JAX backend vs. PhysicsNeMo: training throughput and accuracy on six matched problems, same GPU. underPINN, jinns and DeepXDE columns are mean ± std over three seeds (five for underPINN's Burgers); on Pipe Flow, at reduced scale, jinns is three-seed and `underPINN` a single run; PhysicsNeMo is three-seed (Table 1). **Bold** marks the fastest and the lowest-mean accuracy per row — *lowest mean*, which at these seed counts is not always a distinguishable win: several accuracy intervals overlap, and the text says which. DeepXDE has no collocation minibatching (its own `Model.train` documentation directs users not to pass `batch_size` with `PDE`/`TimePDE`), so its `num_domain` is matched to the *per-step* point count the other frameworks draw rather than to their larger pool; per-step work is therefore matched and the accuracy column is obtained from a fixed point set (Appendix A.10).

Problem	underPINN ms/ep	jinns ms/ep	DeepXDE (JAX) ms/ep	PhysicsNeMo ms/ep	underPINN rel. L^2	jinns rel. L^2	DeepXDE (JAX) rel. L^2	PhysicsNeMo rel. L^2
1-D Burgers (5,000 ep)	**8.50±0.06**	11.19 ± 0.12	9.56 ± 0.03	26.71 ± 0.11	0.219 ± 0.076	$\mathbf{0.134 \pm 0.102}$	0.268 ± 0.036	0.225 ± 0.063
2-D Steady Heat (5,000 ep)	**0.42±0.00**	0.99 ± 0.01	0.86 ± 0.01	5.95 ± 0.15	0.0025 ± 0.0003	0.0013 ± 0.0001	$\mathbf{0.00102 \pm 0.00008}$	0.0158 ± 0.0010
1-D Diffusion (5,000 ep)	**0.48±0.03**	1.11 ± 0.28	0.75 ± 0.04	–	0.013 ± 0.004	$\mathbf{0.011 \pm 0.004}$	0.015 ± 0.002	–
2-D Heat Unsteady (5,000 ep)	**1.01±0.03**	2.33 ± 0.21	1.44 ± 0.02	–	0.070 ± 0.013	0.054 ± 0.032	$\mathbf{0.021 \pm 0.006}$	–
ODE Harmonic (3,000 ep)	**0.49±0.02**	0.53 ± 0.01	n/a†	6.70 ± 0.15	0.771 ± 0.060	$\mathbf{0.177 \pm 0.118}$	n/a†	0.318 ± 0.217
3-D Pipe Flow (30,000 ep, reduced scale)	**8.16**	17.25 ± 0.08	–	–	–	0.067 ± 0.010*	–	–

*jinns' own axial-velocity error (three seeds); its pressure error is 0.127 ± 0.019. underPINN's Pipe Flow accuracy is not measured on this reduced scale (Appendix A.10), so no comparison is drawn for this row. †Not expressible under DeepXDE's JAX backend: $u'' + \omega^2 u = 0$ needs both $u(0)$ and $u'(0)$, and while `IC` imposes the value, the derivative condition must go through `OperatorBC`, which under this backend receives the output array rather than the `(value, apply_fn)` pair `dde.grad` requires (Appendix A.10). "–" marks a problem we did not run for that framework, as distinct from one it cannot express.

`underPINN` is faster than jinns on all six problems (Table 2), 1.07–2.36×, and faster than DeepXDE's JAX backend on all four we ran against it, 1.12–2.06×; both JAX frameworks are in turn faster than PhysicsNeMo throughout. A third compiled-JAX framework, PINNs-JAX (Akbarian Bafghi & Raissi, 2024), gives the same picture on Burgers at 11.23 ± 0.22 ms/epoch against `underPINN`'s 8.50 ± 0.06 (1.32×); we report throughput only for it, because PINNs-JAX reads its initial- and boundary-condition values from a reference solution on a mesh while the other three impose them from closed-form functions, which makes per-step cost comparable but final accuracy not (Appendix A.10). The four compiled-JAX implementations therefore sit within 1.12–1.32× of each other on this problem, against 3.14× for eager-mode PhysicsNeMo — which is the appropriate reading of the larger cross-framework numbers in Table 1: most of that gap is compiled-versus-interpreted execution rather than a framework-quality difference, and the residue attributable to `underPINN` specifically is a small multiple.

Accuracy inverts the ordering, though less firmly than a single run suggested: an earlier single-run version of this table had `jinns` ahead on Burgers, Steady Heat and Heat Unsteady, but its accuracy is highly seed-sensitive (7.1× and 4.1× best-to-worst spreads on Burgers and Heat Unsteady), and on those two its lower mean now overlaps `underPINN`'s (Appendix A.10).

What survives is narrower and better supported. On 2-D Steady Heat jinns is genuinely and repeatably more accurate (0.0013 ± 0.0001, a 1.3× spread, against `underPINN`'s 0.0025 ± 0.0003), and DeepXDE is more accurate still (0.00102 ± 0.00008); DeepXDE also leads on 2-D Heat Unsteady (0.021 ± 0.006). On ODE Harmonic both `jinns` (0.177 ± 0.118) and PhysicsNeMo (0.318 ± 0.217) beat `underPINN` (0.771 ± 0.060; Section 3.9's learning-rate finding again), and at three seeds the two are not distinguishable from each other. `underPINN` is the lowest-mean framework on none of the five rows. DeepXDE reaches its two results from a *fixed* collocation set, since it cannot minibatch collocation points, so that gap is not attributable to point coverage. On Steady Heat, the

one distinguishable `jinns` advantage, a direct test attributes about half the gap to `jinns`' initializer and none of it to its minibatch cycling; the remaining half is unexplained (Appendix A.10).

The seed sensitivity does not reach throughput: `jinns`' per-epoch cost varies by 1.03–1.24× between seeds while its accuracy varies by 1.3–7.1×, so a single-seed baseline is a defensible shortcut for a speed table but not for an accuracy one. Pipe Flow runs at a reduced, `jinns`-matched scale at which `underPINN`'s accuracy was not measured, so that row draws no accuracy comparison.

3.5 Convergence and Solution Accuracy

A concern when relying on a compiled, largely un-inspected training loop is whether the underlying optimization remains numerically well-behaved. Relative L^2 error decreases roughly monotonically for the lower-cost, smoother problems (1-D Wave, 2-D Helmholtz, 2-D Steady Heat, ODE Harmonic) by one to two orders of magnitude over their tested budgets; the blast-wave and shock problems converge markedly more slowly, tracking discontinuity severity more than spatial dimensionality, and Toro Test 3's relative L^2 error plateaus around 33% ($32.9\% \pm 0.2\%$, three seeds) even at $60{,}000$ epochs (Appendix A.11, Figure 8).

Loss trajectories across all nine problems (Appendix A.12, Figure 9) are largely monotonic, with no divergence, including for the stiffer conservation-law problems (Toro Test 3, Ramp NS). The visible transient spikes, on ODE Harmonic and Wave, occur in problems that never resample, so they are optimizer transients rather than resampling artifacts, and each recovers within a few tens of epochs; the two problems that do resample (Ramp, Ramp NS) show no comparable jumps. This is descriptive of the single runs shown, not a stability guarantee.

3.6 Spatial Resolution of Canonical PDEs

Predicted fields for Burgers, Wave, Steady Heat and Helmholtz track the exact solutions (Appendix A.2, Figure 3): Burgers' steepening front is resolved without non-physical oscillation, and the largest errors sit where expected, in a thin band around the Burgers front, along the Wave characteristics and near Helmholtz's boundaries (Appendix A.11).

3.7 Compressible Flows and Shock Resolution

We also evaluated compressible Euler/Navier–Stokes shock problems, a harder multi-scale-resolution test than the smooth problems above. The 1-D Toro Test 3 problem (a blast wave with a five-decade pressure jump) is the hardest case in the suite: Figure 4a shows the network qualitatively tracks the coupled (ρ, u, p) profiles' step-changes and rarefaction fan (fixed $\nu_{av} = 0.001$), but the post-shock density spike and contact discontinuity keep relative L^2 error plateaued around 33% ($32.9\% \pm 0.2\%$, three seeds) at $60{,}000$ epochs (Appendix A.8), consistent with the well-documented difficulty of resolving strong, unsteady 1-D shocks with a single global PINN.

This persistent error reflects compounding factors, detailed in Appendix A.8, rather than a training instability: spectral bias interacting with a fixed (not spatially-adaptive) artificial-viscosity layer width, the problem's genuinely unsteady multi-discontinuity structure, and a metric that, after non-dimensionalization, is dominated by the post-shock density spike the network under-resolves (Table 5 gives the definition and weighting).

Figures 4b and 5a (Appendix A.2) show the 2-D inviscid and viscous (Navier–Stokes) Ramp benchmarks, both resolving an oblique shock from supersonic flow over a compression ramp ($M_\infty = 3$). The inviscid case ($\theta = 10^\circ$, $\nu_{av} = 0.001$) tracks the analytic post-shock Mach state, reaching $\sim 1\%$ relative L^2 by $20{,}000$ epochs. The viscous case ($\theta = 15^\circ$, $Re = 10^4$) additionally resolves a genuine shock-boundary-layer interaction with no closed-form reference; Figure 5a shows its Mach field (theoretical inviscid shock angle overlaid) and the RAR-D collocation pool's migration toward the shock over training.

3.8 When Does Adaptive Collocation Help? A Six-Problem Ablation

Adaptive collocation is widely assumed to improve PINN accuracy, but the conditions under which it does are rarely isolated. We ran a four-arm ablation across six problems, five seeds each – uniform

(never resampled); RAD (Wu et al., 2023) and QR-DEIM-R (Section 2.3) at *full replacement*; and RAD at an $80/20$ *hybrid* (20% of the pool resampled, the rest frozen) – plus a fifth arm, QR-DEIM-R's own $80/20$ hybrid, run five seeds each on the two shock problems only (Toro Test 3, Ramp NS; Tables 5–6), across four regimes: two smooth (2-D steady Poisson/Steady Heat, 1-D Diffusion), one with a sharp viscous layer (1-D Burgers), two compressible shocks (Toro Test 3, Ramp NS), and one recirculating flow with corner singularities (lid-driven cavity, $Re = 100$, `examples/LDC`, added for this ablation). Five seeds per arm, resampling every 500 epochs.

Table 3: Relative L^2 error (mean over five seeds; see the text and Tables 5–6 for std and further metrics) for each resampling strategy, by problem. QR-DEIM-R's own $80/20$ hybrid (last column) was run, five seeds each, only on the two shock problems (Tables 5–6); "–" marks the four problems where it was not run, not a null result. **Bold** marks an arm that beats uniform on every one of the five seeds (a paired win, not just a lower mean). "Uniform" is the never-resampled baseline pool for each problem; on Ramp NS that pool is not plain-uniform but already blends boundary-layer clustering with uniform coverage (Appendix A.3), which is why Table 6 labels the same arm "static." RAD helps in proportion to how under-resolved a localized, uncapped feature is; QR-DEIM-R does not win a single row. Budgets differ by row and are listed in the Epochs column; each arm within a row shares its budget. ‡The Poisson and diffusion rows use the Table 2 problem configurations (2-D Steady Heat, i.e. Poisson, and 1-D Diffusion) at their $5{,}000$-epoch budget; their uniform-arm errors agree with that table (0.013 for diffusion; 0.0028 over five seeds here vs. 0.0025 over three there for Poisson).

Problem	Regime	Epochs	Uniform	RAD (full)	RAD (hyb.)	QR-DEIM-R (full)	QR-DEIM-R (hyb.)
2-D Poisson	smooth	5,000‡	0.0028	0.0028	0.0030	0.0034	–
1-D diffusion	smooth	5,000‡	0.013	**0.012**	**0.013**	0.012	–
1-D Burgers	viscous layer	5,000	0.219	**0.072**	0.161	0.367	–
Lid-driven cav.	corners, recirc.	10,000	0.165	**0.143**	0.144	0.175	–
Toro Test 3	shock (visc.-capped)	5,000	0.445	0.473	0.461	0.559	0.477
Ramp NS	shock (visc.-capped)	30,000	0.068	0.068	0.068	0.070	0.070

RAD helps in proportion to how under-resolved a localized, uncapped feature is. On 1-D Burgers, RAD with full replacement cuts relative L^2 from 0.219 ± 0.076 (uniform) to 0.072 ± 0.028 – a $3\times$ reduction with lower variance, beating uniform on every seed. Burgers develops a genuine thin internal layer at $x = 0$ that uniform sampling under-resolves, and unlike the shock problems below nothing else caps how sharply the network can represent it, so concentrating points there directly buys accuracy – reproducing the behaviour RAD was introduced for (Wu et al., 2023). The benefit is not monotone in layer sharpness: at the thinner $\nu = 0.01/\pi$ layer Wu et al. use, RAD's edge shrinks to $1.15\times$ and its seed-to-seed variance roughly triples, becoming seed-dependent once the layer is thin enough – a smaller-scale echo of the single-run fragility below. On the two smooth problems the effects are small. On Poisson every arm lands within noise of uniform. On diffusion both RAD arms beat uniform on every one of the five seeds, but by only about 9% and 2% (0.0117 and 0.0127 against 0.0129), a consistent gain far smaller than on Burgers.

The lid-driven cavity shows the same mechanism at smaller scale: its singular corners are a weak localized feature, and RAD improves velocity error by 13% ($0.165 \to 0.143$) on all five seeds (Appendix A.3).

On the shock problems, a fixed dissipation coefficient caps every arm. On both Toro Test 3 and Ramp NS, by domain-averaged relative L^2 and localized shock-sharpness metrics, no arm improves on uniform beyond seed noise (Appendix A.3, Tables 5–6): no Toro Test 3 arm recovers more than about 60% of the exact density peak or 8% of the reference peak $|\partial_x \rho|$, and Ramp NS's five arms sit within a 0.068–0.070 band, the only difference beyond seed noise being QR-DEIM-R's, about 2% worse than the rest. The fixed, non-adaptive artificial-viscosity coefficient (Section 3.7) caps how sharp any arm's shock can be regardless of point placement, so there is no headroom for a resampling *strategy* to exploit – and making that coefficient *trainable* rather than fixed (Appendix A.3) makes Toro Test 3 worse, since nothing in the objective penalizes a large learned coefficient.

QR-DEIM-R does not work. The deterministic DEIM-inspired selector we tested underperforms RAD on every one of the six problems and has a higher mean error than uniform on five (Table 3). Its full-replacement form is the least stable arm in the suite: $1.7\times$ worse than uniform on Burgers, and

on Toro Test 3 its density-peak recovery ranges 0.25–0.57 across seeds, the shock gradient nearly collapsing on its worst seed (Appendix A.3). We keep it as an option but do not recommend it – the opposite of an earlier, single-run version of this paper's claim for it. This also runs against Celaya et al. (2025), who report accuracy gains from QR-DEIM-based collocation on their own benchmark PDEs. We do not read this as contradicting them: QR-DEIM-R is a variant in that family, not a reimplementation of their strategies, and the problems, budgets and baselines differ. The benefit they report did not transfer to this construction on these six problems; a controlled reimplementation of their strategies is the test that would settle it, and we did not run it.

Single-run ablations of this kind are fragile. On Toro Test 3 the four closest arms have seed-to-seed spreads (0.016–0.029) as large as the gaps between their means (at most 0.032), so a single seed can rank them in any order; and two runs at an *identical* seed and code, differing only in GPU floating-point non-determinism, disagreed on whether hybrid QR-DEIM-R tied uniform or lost outright (Appendix A.3). This is a caution about single-run PINN ablations generally, including ones we ran only once elsewhere in this paper.

What this ablation does not cover. It spans residual-magnitude reweighting of a candidate pool and deterministic subset selection from it; learned-density samplers such as DAS-PINNs (Tang et al., 2023) are untested, for reasons given in Appendix A.3.

3.9 ODE and 3-D Boundary Value Problems

The ODE harmonic oscillator reaches relative L^2 error $(2.3 \pm 0.4) \times 10^{-4}$ (three seeds; Appendix A.2, Figure 6).

Gauss–Newton vs. Adam on the harmonic oscillator. On a small, 481-parameter network, we compared Levenberg–Marquardt-damped Gauss–Newton (Section 2.2) against Adam over five seeds: at `underPINN`'s default LR Adam does not converge to $100{,}000$ epochs, but tuned it matches or beats Gauss–Newton on both accuracy and wall-clock (Appendix A.4, Table 7).

On the 3-D Hagen–Poiseuille pipe flow the predicted radial velocity profile matches the analytic parabola (Appendix A.2, Figure 7), with relative L^2 error $(7.8 \pm 0.7) \times 10^{-4}$ at $60{,}000$ epochs (three seeds).

4 Conclusion and Future Work

We presented `underPINN`, a JAX-native PINN framework built around a compiled training step, and used it as an instrument for matched comparison. Against eager PyTorch the compiled step is 1.03–$17.4\times$ faster (Section 3.2). Against PhysicsNeMo Sym it is faster on all eight matched problems (1.23–$14.2\times$) and at least as accurate on six of seven, less accurate on ODE Harmonic through an untuned learning rate (Section 3.3). Matching the two exposed a hidden weight-normalization default whose removal costs PhysicsNeMo $69\times$ in accuracy on Helmholtz and nothing on Wave (Appendix A.9). Against the compiled JAX frameworks the margin is a small multiple (1.07–$2.36\times$ over `jinns`, 1.12–$2.06\times$ over DeepXDE's JAX backend), so most of the larger cross-framework gap is compiled-versus-eager execution rather than framework quality, and on accuracy the order inverts: `underPINN` has the lowest mean error on none of the five scored rows (Section 3.4).

Multi-seeding changed conclusions on both sides: it dissolved two of `jinns`' three single-run accuracy wins, made PhysicsNeMo's Burgers result a tie, shrank its ODE Harmonic advantage and corrected two of its timings, without changing any speed ranking. RAD helps where a localized, uncapped feature exists; QR-DEIM-R and Gauss–Newton do not improve on their baselines (Sections 3.8, 3.9). We regard the data-free benchmark suite, including Toro Test 3 and Ramp NS, and the account of what went wrong in matching frameworks as more central to this paper's contribution than either tested method's outcome.

Limitations. A few results are single-run (`underPINN`'s Pipe Flow comparison rows, loss curves, field plots), and not every component is isolated (Appendix A.6). Planned: neural operators, `pmap`/`pjit`, Bayesian UQ. `underPINN` is open-source.

AI USE STATEMENT

In this work, we used a generative AI coding assistant (Claude, via Claude Code) as a tool during manuscript revision: drafting and iterating on prose under our direction; implementing, running, and debugging the multi-seed ablation scripts and cross-framework comparison scripts described in Sections 3.3–3.8; and, in the course of that work, identifying and fixing a pre-existing seeding bug in `underPINN`'s own benchmark harness (Appendix A.11). We did not use generative AI to design the framework's core algorithms or to generate the underlying numerical results themselves beyond running the code we wrote and reviewed – every reported number comes from an executed training run, not a generated one. We reviewed the AI-drafted text and AI-generated code changes for correctness before incorporating them, re-ran the reported experiments ourselves, and take full responsibility for the final content of this work, including all text, claims, and artifacts.

REPRODUCIBILITY STATEMENT

`underPINN` is released as open-source software under a public repository that includes every evaluator, ablation script, and configuration file referenced in this paper. All benchmark problems use fixed, disclosed hyperparameters and collocation counts (Table 4 names the evaluator behind each); all multi-seed results report the exact seed set used (Sections 3.8, 3.9) and can be regenerated with the corresponding script under `benchmarks/rebuttal/`. Cross-framework comparisons (Sections 3.3, 3.4) document every configuration divergence we could not close, in the accompanying repository's README rather than repeated in the main text, so a reader auditing a specific number can trace it to the exact script and commit that produced it.

ETHICS STATEMENT

This work benchmarks a numerical PINN training framework on standard synthetic and manufactured physics problems (blast waves, shock flows, canonical PDEs); it does not involve human subjects, personal data, or dual-use physical experimentation, and we do not identify a specific ethical concern requiring further discussion beyond the standard research-integrity commitments (Reproducibility Statement above; the AI use statement above).

REFERENCES

Reza Akbarian Bafghi and Maziar Raissi. Comparing PINNs across frameworks: JAX, tensorflow, and pytorch. In *ICLR 2024 Workshop on AI4DifferentialEquations In Science*, 2024. URL https://openreview.net/forum?id=BPFzolSSrI.

Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998. doi:10.1162/089976698300017746.

Sokratis J. Anagnostopoulos, Juan Diego Toscano, Nikolaos Stergiopulos, and George Em Karniadakis. Residual-based attention in physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 421:116805, 2024. doi:10.1016/j.cma.2024.116805.

Cosmin Anitescu, Elena Atroshchenko, Naif Alajlan, and Timon Rabczuk. Artificial neural network methods for the solution of second order boundary value problems. *Computers, Materials & Continua*, 59:345–359, 01 2019. doi:10.32604/cmc.2019.06641.

Jason Ansel, Edward Yang, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024. doi:10.1145/3620665.3640366.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, 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/jax-ml/jax.

Adrian Celaya, David Fuentes, and Beatrice Riviere. Adaptive collocation point strategies for physics informed neural networks via the qr discrete empirical interpolation method. *arXiv preprint arXiv:2501.07700*, 2025. doi:10.48550/arXiv.2501.07700.

Saifon Chaturantabut and Danny C. Sorensen. Nonlinear model reduction via discrete empirical interpolation. *SIAM Journal on Scientific Computing*, 32(5):2737–2764, 2010. doi:10.1137/090766498.

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics-informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92, 2022. doi:10.1007/s10915-022-01939-z. URL https://doi.org/10.1007/s10915-022-01939-z.

Petros Drineas, Michael W. Mahoney, and S. Muthukrishnan. Sampling algorithms for ℓ_2 regression and applications. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1127–1136, 2006.

Zlatko Drmač and Serkan Gugercin. A new selection operator for the discrete empirical interpolation method—improved a priori error bound and extensions. *SIAM Journal on Scientific Computing*, 38(2):A631–A648, 2016. doi:10.1137/15M1019271.

Hugo Gangloff and Nicolas Jouvin. jinns: Physics informed neural network with JAX, 2024. URL https://gitlab.com/mia_jinns/jinns.

Zhongkai Hao, Jiachen Yao, Chang Su, Hang Su, Ziao Wang, Fanzhi Lu, Zeyu Xia, Yichi Zhang, Songming Liu, Lu Lu, and Jun Zhu. PINNacle: A comprehensive benchmark of physics-informed neural networks for solving PDEs. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2306.08827.

George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3:422–440, 2021. doi:10.1038/s42254-021-00314-5. URL https://doi.org/10.1038/s42254-021-00314-5.

Prashant Kumar and Rajesh Ranjan. Evaluation of physics-informed machine learning approach for computation of fluid flows. In *Proceedings of Fluid Mechanics and Fluid Power (FMFP) 2023, Vol. 2*, pp. 429–440, Singapore, 2025. Springer Nature Singapore. ISBN 978-981-97-6783-0.

Prashant Kumar and Rajesh Ranjan. A robust data-free physics-informed neural network for compressible flows with shocks. *Computers & Fluids*, 308:106975, 2026a. ISSN 0045-7930. doi:https://doi.org/10.1016/j.compfluid.2026.106975. URL https://www.sciencedirect.com/science/article/pii/S0045793026000174.

Prashant Kumar and Rajesh Ranjan. Multiscale physics-informed neural network for complex fluid flows with long-range dependencies, 2026b. URL https://arxiv.org/abs/2604.05652.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Lu Lu, Xuhui Meng, Zhiping Mao, and George E. Karniadakis. Deepxde: A deep learning library for solving differential equations. *CoRR*, abs/1907.04502, 2019. URL http://arxiv.org/abs/1907.04502.

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*, 3:218–229, 2021. doi:10.1038/s42256-021-00302-5. URL https://doi.org/10.1038/s42256-021-00302-5.

Ben Moseley, Andrew Markham, and Tarje Nissen-Meyer. Finite basis physics-informed neural networks (fbpinns): a scalable domain decomposition approach for solving differential equations. *Advances in Computational Mathematics*, 49, 2023. doi:10.1007/s10444-023-10065-9. URL https://doi.org/10.1007/s10444-023-10065-9.

Johannes Müller and Marius Zeinhofer. Achieving high accuracy with pinns via energy natural gradient descent. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

NVIDIA Corporation. NVIDIA PhysicsNeMo, 2024. URL https://github.com/NVIDIA/physicsnemo.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International conference on machine learning*, pp. 5301–5310. PMLR, 2019.

M. Raissi, P. Perdikaris, and G.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. ISSN 0021-9991. doi:https://doi.org/10.1016/j.jcp.2018.10.045. URL https://www.sciencedirect.com/science/article/pii/S0021999118307125.

Amit Sabne. Xla : Compiling machine learning for peak performance. 2020.

Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 29, 2016.

Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Daniel MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. PDEBench: An extensive benchmark for scientific machine learning. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2022. URL https://github.com/pdebench/PDEBench.

Kejun Tang, Xiaoliang Wan, and Chao Yang. DAS-PINNs: A deep adaptive sampling method for solving high-dimensional partial differential equations. *Journal of Computational Physics*, 476: 111868, 2023. doi:10.1016/j.jcp.2022.111868.

Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021a. doi:10.1137/20M1318043. URL https://doi.org/10.1137/20M1318043.

Sifan Wang, Hanwen Wang, and Paris Perdikaris. On the eigenvector bias of fourier feature networks: From regression to solving multi-scale pdes with physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 384:113938, 2021b. ISSN 0045-7825. doi:10.1016/j.cma.2021.113938. URL http://dx.doi.org/10.1016/j.cma.2021.113938.

Sifan Wang, Shyam Sankaran, and Paris Perdikaris. Respecting causality is all you need for training physics-informed neural networks, 2022. URL https://arxiv.org/abs/2203.07404.

Sifan Wang, Shyam Sankaran, Hanwen Wang, and Paris Perdikaris. An expert's guide to training physics-informed neural networks, 2023. URL https://arxiv.org/abs/2308.08468.

Chenxi Wu, Min Zhu, Qinyang Tan, Yadhu Kartha, and Lu Lu. A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 403:115671, 2023. ISSN 0045-7825. doi:https://doi.org/10.1016/j.cma.2022.115671. URL https://www.sciencedirect.com/science/article/pii/S0045782522006260.

Kirill Zubov, Zoe McCarthy, Yingbo Ma, Francesco Calisto, Valerio Pagliarino, Simone Azeglio, Luca Bottero, Emmanuel Lujan, Valentin Sulzer, Ashutosh Bharambe, Nand Vinchhi, Kaushik Balakrishnan, Devesh Upadhyay, and Chris Rackauckas. Neuralpde: Automating physics-informed neural networks (pinns) with error approximations. 07 2021. doi:10.48550/arXiv.2107.09443.

A Appendix

A.1 Related Work, in Full

Physics-Informed Neural Networks (PINNs) integrate the governing PDEs, initial conditions, and boundary constraints of a continuous dynamical system directly into a neural network's loss function, following Raissi et al. (2019), providing a mesh-free optimization framework for forward and inverse problems across fluid dynamics, heat transfer, and parameter discovery (Karniadakis et al., 2021; Cuomo et al., 2022; Kumar & Ranjan, 2025). Neural operators (DeepONets (Lu et al., 2021), Fourier Neural Operators (Li et al., 2020)) address the related but distinct problem of learning function-space mappings for parameterized PDE families; collocation-based PINNs remain the more common choice for the single-instance, data-sparse boundary value problems this paper considers. Standard MLP PINNs exhibit well-documented failure modes the field has developed targeted fixes for: spectral bias toward low-frequency content (Rahaman et al., 2019), addressed by Fourier feature embeddings (Wang et al., 2021b) and multiscale architectures (Kumar & Ranjan, 2026b); conflicting PDE-residual vs. boundary/initial-condition gradients (Wang et al., 2021a), addressed by gated architectures re-injecting input coordinates at every layer (Wang et al., 2022); and single-network optimization stalling on large or complex domains, addressed by Finite Basis PINNs (FBPINNs) (Moseley et al., 2023) decomposing the domain into overlapping, partition-of-unity-combined subdomains.

Collocation-point distribution matters as much as architecture: static schemes (e.g. Latin Hypercube) under-resolve localized features. RAR (Lu et al., 2019; Anitescu et al., 2019) periodically adds points where the residual is largest; RAD (Wu et al., 2023) resamples proportional to a powered residual magnitude $|R|^k$, used to concentrate points on shock fronts in compressible-flow PINNs (Kumar & Ranjan, 2026a); RBA (Anagnostopoulos et al., 2024) reweights loss terms per point instead. A further line replaces the candidate pool altogether, drawing points from a learned density: DAS-PINNs (Tang et al., 2023) fit a normalizing flow to the residual and sample from it, with the largest reported gains on high-dimensional, low-regularity problems. A separate line selects subsets deterministically: DEIM (Chaturantabut & Sorensen, 2010) and its column-pivoted-QR variant Q-DEIM (Drmač & Gugercin, 2016), developed in reduced-order modeling to pick a well-conditioned, mutually independent row subset rather than sampling by magnitude alone. Celaya et al. (2025) bring this principle to PINN collocation directly, proposing two QR-DEIM-based selection strategies and reporting improved accuracy over existing adaptive methods on benchmark PDEs. Section 3.8 tests a variant in that family (QR-DEIM-R, Section 2.3) against RAD on six problems and does not reproduce that benefit: it does not improve on RAD or uniform sampling on any of them, while RAD itself helps in proportion to how under-resolved a localized, uncapped feature is (Section 3.9 similarly evaluates a Gauss–Newton alternative to Adam (Amari, 1998; Müller & Zeinhofer, 2023)).

As PINN methodology has matured, supporting software has proliferated – DeepXDE (Lu et al., 2019) (TensorFlow/PyTorch/JAX), NVIDIA PhysicsNeMo (NVIDIA Corporation, 2024) (industrial multi-GPU PyTorch), NeuralPDE.jl (Zubov et al., 2021) (Julia), `jinns` (Gangloff & Jouvin, 2024) (JAX) – making cross-framework comparison routine, but two difficulties recur. First, *dispatch overhead*: a PINN step is a cheap, high-order-derivative computation repeated for tens of thousands of epochs, and dispatching each one individually through Python control flow (eager execution) can dominate wall-clock time; JAX/XLA (Sabne, 2020) offer a route around this, and several released JAX PINN codebases already exploit it — `jinns` (Gangloff & Jouvin, 2024), JAX-PI (Wang et al., 2023), PINNs-JAX (Akbarian Bafghi & Raissi, 2024), and DeepXDE's JAX backend (Lu et al., 2019) — with Akbarian Bafghi & Raissi (2024) reporting XLA speedups over TensorFlow and PyTorch that, as here, shrink once per-step compute grows. What we did not find among them was one framework combining a compiled training loop with domain decomposition, gated architectures, and adaptive collocation as configuration-selected components, which is the specific gap `underPINN` fills; the claim is about that combination, not about JAX PINN frameworks being scarce. Second, matched cross-framework comparison is itself error-prone: two frameworks configured with the "same" MLP depth, width, and activation can still differ in defaults that never surface in a problem's configuration file. Section 3.3 identifies one such default in PhysicsNeMo (weight normalization on every hidden layer, changing both the trained model and its per-step compute cost), and Section 3.4 identifies two more in `jinns` (time-input renormalization; a convective-term operator hard-coded to two dimensions). Compounding this, Section 3.8 shows a single-run adaptive-collocation ablation can reverse which method looks best under an identical seed and identical code, as GPU floating-point non-determinism compounds nonlinearly through training dynamics.

A.2 Result Figures

Referenced from Sections 2–3.7; moved here for space, not because they are less load-bearing than the numbers already given in the main text.

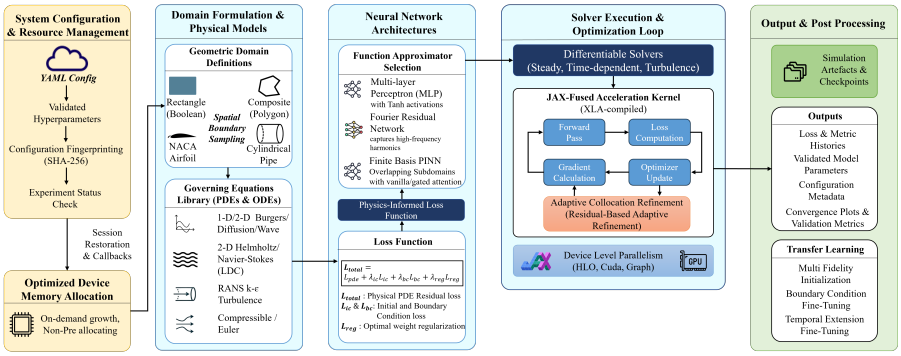
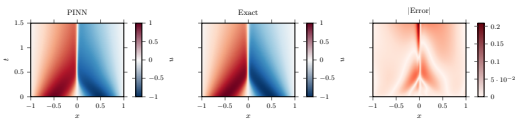


Figure 2: High-level schematic of the `underPINN` framework's five main components (Section 2).

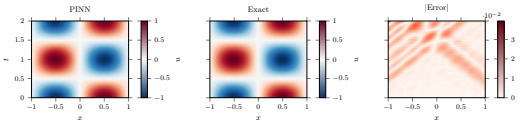
Table 4: The nine benchmark problems evaluated in this paper. Each row describes the benchmark evaluator that produced Figures 1 and 8 (`underPINN/benchmark_utils/evaluators.py` in the repository); a runnable example of each problem is at `examples/PINNs/<name>/`, `<name>` ∈ {`burgers`, `wave`, `helmholtz`, `heat`, `ode`, `ramp`, `toro3`, `pipe_flow`, `ramp_ns`} in row order. The cross-framework comparisons (Tables 1–2) use separately matched plain-MLP configurations described with those tables.

Problem	PDE / Governing equation	Network	Domain
1-D Burgers (ν=0.01)	$u_t + uu_x = \nu u_{xx}$	MLP + FBPINN, RBA	1-D, unsteady
1-D Wave (c=1)	$u_{tt} = c^2 u_{xx}$	FourierMLP	1-D, unsteady
2-D Helmholtz (k=1)	$\Delta u + k^2 u = f$	FourierMLP	2-D, steady
2-D Steady Heat (Poisson)	$\nabla^2 u = -2\pi^2 \sin \pi x \sin \pi y$	MLP	2-D, steady
ODE Harmonic (ω=2, $t \in [0, 3]$)	$u'' + \omega^2 u = 0$	FourierMLP (σ=ω)	0-D spatial, unsteady
2-D Ramp – Oblique Shock (M=3, θ=10°)	Steady compressible Euler	MLP, RAR-D	2-D, steady
1-D Toro Test 3 (blast wave)	1-D compressible Euler	MLP (static pool)	1-D, unsteady
3-D Pipe Flow (Re=10)	Steady 3-D incompressible N–S	MLP	3-D, steady
2-D Ramp NS – Viscous SBLI (M=3, Re=10^4)	Steady compressible Navier–Stokes	MLP, RAR-D	2-D, steady

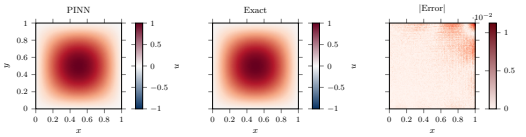
`examples/PINNs/<name>/` configurations are production setups and differ from the evaluators Table 4 describes: Toro Test 3's and Ramp NS's `config.yaml` defaults run 90,000 and 300,000 epochs, and Toro Test 3's example adds RAR resampling every 3,000 epochs. Every number in this paper instead calls the corresponding `underPINN.benchmark_utils` evaluator directly, at the budget stated in that number's caption, reproducible from the scripts under `benchmarks/rebuttal/`.



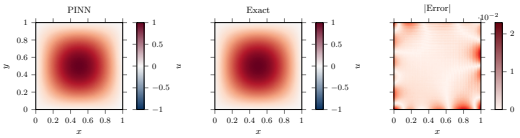
(a) Burgers equation.



(b) 1-D Wave equation.

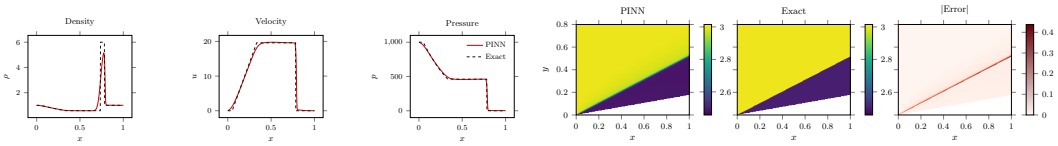


(c) Steady Heat equation.



(d) Helmholtz equation.

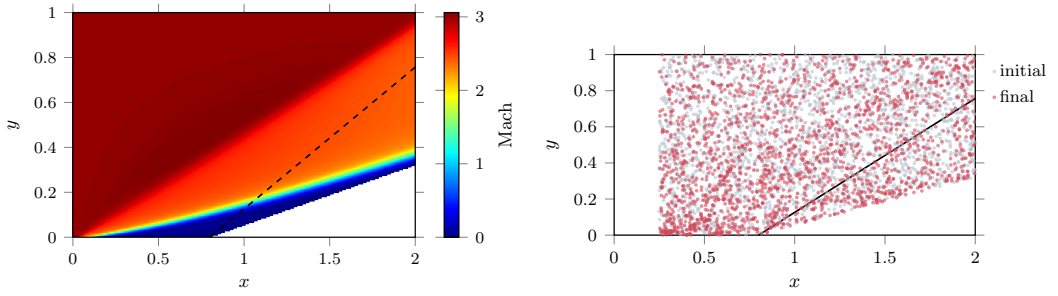
Figure 3: Predicted vs. exact spatial fields for four canonical PDEs (Section 3).



(a) Toro Test 3, $\nu_{av} = 0.001$.

(b) Inviscid Ramp, $\nu_{av} = 0.001$.

Figure 4: Solution fields for the two 1-/2-D shock problems (Section 3.7).



(a) Mach field, viscous ramp SBLI.

(b) RAR-D collocation migration.

Figure 5: Ramp NS ($M_\infty = 3$, $\theta = 15°$, $Re = 10^4$): Mach field with the theoretical inviscid shock angle overlaid, and the collocation pool's initial-to-final migration toward the shock (Section 3.7).

A.3 Adaptive Collocation: Per-Metric and Per-Problem Detail

This appendix gives the full per-metric breakdown Section 3.8 summarizes: Toro Test 3's four-metric table, Ramp NS's table and fixed-viscosity sweep, the trainable-viscosity result, and the single-run reproducibility episode.

Learned-density sampling, untested. The arms of Section 3.8 span one family of adaptive collocation: residual-magnitude reweighting of a fixed candidate pool (RAD, RAR-D) and deterministic subset selection from it (QR-DEIM-R). A second family replaces the pool itself with samples drawn from a learned distribution — DAS-PINNs (Tang et al., 2023) fits a KRnet normalizing flow to the residual, treated as an unnormalized density, and draws new collocation points from it, reporting the largest gains on high-dimensional, low-regularity problems. We do not test it, and the reason is worth stating rather than leaving as an omission. A fair test would have to reimplement the method *inside* `underPINN`, as every arm of Section 3.8 is, since the reference implementation is TensorFlow-based and running it directly would compare execution engines rather than sampling strategies — the confound Sections 3.3–3.4 are about. Porting KRnet faithfully is a larger undertaking than this ablation's scope, and a hasty approximation carrying the DAS name would be worse than a stated gap. The regime is also mismatched: DAS targets high dimension and low regularity, whereas five of our six ablation problems are 1-D or 2-D with smooth or viscosity-capped structure, so this suite would be a weak test of its central claim even implemented correctly. We note separately that Celaya et al. (2025)'s QR-DEIM strategies, discussed above, are the published point of comparison closest to QR-DEIM-R, and that a controlled reimplementation of those is the more directly relevant missing experiment.

Lid-driven cavity result. The lid-driven cavity shows the same mechanism at smaller scale: its two singular corners are a real but weak localized feature, and RAD improves full-field velocity error from 0.165 ± 0.034 to 0.143 ± 0.030 (13%) on all five seeds (full replacement) and four of five (hybrid) – a small, consistent effect, hence a paired win in Table 3 despite overlapping marginal distributions.

Lid-driven cavity configuration. The cavity is added for this ablation only and is absent from Table 4, so its setup is given here. Steady incompressible flow at $Re = 100$ on the unit square, driven by a unit-velocity lid. Network: FBPINN with a SimpleGate block over a single subdomain,

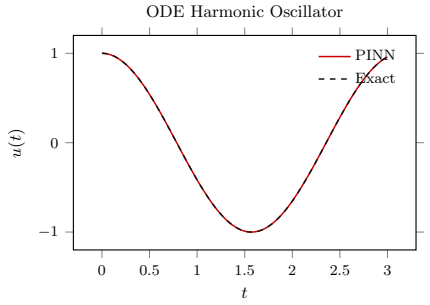


Figure 6: Solution for the ODE Harmonic Oscillator, $u'' + \omega^2 u = 0$, $\omega = 2$ (Section 3).

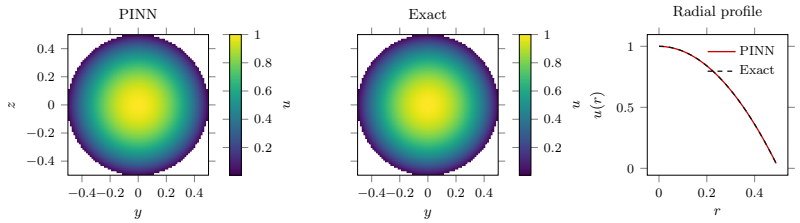


Figure 7: Results for the 3-D Hagen–Poiseuille Pipe Flow ($Re = 10$): PINN and exact velocity cross-sections, and the radial velocity profile (Section 3).

$[2, 64, 64, 64, 64, 3]$ mapping $(x, y) \mapsto (u, v, p)$, mirroring `examples/LDC/run_ldc.py`. Collocation: a 10,000-point interior pool and 300 boundary points per wall; 10,000 epochs, five seeds per arm. The corner singularities where the moving lid meets the stationary walls receive no special treatment — no lid-velocity tapering or corner regularization — so the discontinuous lid velocity enters the boundary loss as specified. There is no closed-form solution; each arm is scored against a Fluent reference field at $Re = 100$ on a 201×201 grid (`examples/LDC/re100.csv`) by the full-field relative L^2 of the velocity magnitude, with the two Ghia-style centreline profiles (u along $x = 0.5$, v along $y = 0.5$) as a secondary check.

Table 5: Toro Test 3, five seeds, 5,000 epochs (mean $\pm$ std). Relative L^2 is taken over the concatenated non-dimensional state $q = (\rho/\rho_{\text{ref}},\ u/u_{\text{ref}},\ p/p_{\text{ref}})$ at t_{final} on a uniform 800-point grid, $\|\hat{q} - q\|_2/\|q\|_2$ with the norm summed over all three components ($\rho_{\text{ref}} = 1$, $p_{\text{ref}} = 1000$, $u_{\text{ref}} = \sqrt{p_{\text{ref}}/\rho_{\text{ref}}}$), against the exact Riemann solution. In these units density carries $\sim$82% of $\|q\|_2^2$, pressure $\sim$11% and velocity $\sim$8%, so the metric is density-dominated; Figure 8 uses the same definition on a 400-point grid; the last three columns are localized shock-capture ratios against that same solution sampled on a common grid — the fraction of the exact peak density recovered, and peak $|\partial_x \rho|$ and $|\partial_x p|$ as a fraction of the exact (grid-limited) value, so higher is sharper and 1.0 would resolve the jump as sharply as the reference at this resolution.

Arm	Relative L^2	$\rho_{\max}$ frac.	$\lvert\partial_x \rho\rvert$ frac.	$\lvert\partial_x p\rvert$ frac.
Uniform (no resampling)	$\mathbf{0.445 \pm 0.016}$	$\mathbf{0.597 \pm 0.025}$	0.071 ± 0.012	0.117 ± 0.017
RAD, full replacement	0.473 ± 0.028	0.573 ± 0.020	0.067 ± 0.009	0.113 ± 0.012
RAD, hybrid 80/20	0.461 ± 0.022	0.590 ± 0.031	$\mathbf{0.072 \pm 0.012}$	$\mathbf{0.126 \pm 0.019}$
QR-DEIM-R, hybrid 80/20	0.477 ± 0.029	0.565 ± 0.025	0.063 ± 0.010	0.115 ± 0.014
QR-DEIM-R, full replacement	0.559 ± 0.061	0.423 ± 0.111	0.033 ± 0.023	0.062 ± 0.037

Static has the lowest domain-averaged error and recovers the most of the density peak; the RAD arms stay within about one standard deviation of it, full replacement at the edge of that (0.473 ± 0.028), and no adaptive arm beats it. Concentrating points on the moving shock – what adaptive resampling is for – does not help here, because the fixed, non-adaptive artificial-viscosity coefficient (Section 3.7) already caps how sharp any arm's captured shock can be, independent of where the points sit: every arm recovers at most about 60% of the exact density peak (full-replacement QR-DEIM-R only 42%)

and resolves peak $|\partial_x \rho|$ at under 8% of the reference, i.e. smears the front over roughly an order of magnitude more width than the grid.

Table 6: Ramp NS (viscous SBLI), $30{,}000$ epochs, resampled every 500, five seeds (mean $\pm$ std). Relative L^2 is the outer-flow Mach error against the analytic oblique-shock field; peak $|\nabla \rho|$ near the shock line is a localized sharpness measure, not scored against a reference. No arm is reliably best on either column.

Arm	Relative L^2	Peak $\lvert\nabla\rho\rvert$ near shock
Static (no resampling)	$\mathbf{0.0682 \pm 0.0009}$	20.72 ± 0.67
RAD, full replacement	0.0683 ± 0.0011	20.56 ± 0.66
RAD, hybrid $80/20$	0.0682 ± 0.0013	20.60 ± 0.58
QR-DEIM-R, full replacement	0.0695 ± 0.0008	20.59 ± 0.56
QR-DEIM-R, hybrid $80/20$	0.0695 ± 0.0006	20.95 ± 0.71

Ramp NS reproduces this pattern with less drama: every arm's outer-flow Mach error sits in a 0.068–0.070 band, and the localized peak-$|\nabla\rho|$ measure around the analytic shock line separates the arms by under 2% against a seed-to-seed standard deviation several times that – no arm resolves the shock measurably sharper than static uniform sampling. The one effect that survives five seeds is small and in the wrong direction for the method: both QR-DEIM-R variants are about 2% worse on domain-averaged error than static or RAD. We attribute the overall flatness to the static baseline here not being plain-uniform sampling to begin with – it already blends a boundary-layer-clustered pool on top of uniform coverage – leaving little headroom for a resampling *strategy* choice to move the aggregate error.

Separately, we swept the fixed artificial-viscosity coefficient on Ramp NS's static-baseline arm (one seed; the same fixed-ν_{av} limitation Section 3.7 raises for Toro Test 3) from the evaluator's default 2×10^{-3} down to 1×10^{-5}, three orders of magnitude, without finding an instability threshold: relative L^2 improves monotonically to a plateau around 2×10^{-4} ($0.069 \to 0.060$, a real $\sim 13\%$ reduction) and peak $|\nabla\rho|$ continues rising with diminishing returns to 22.3 at the lowest value tested, with no Gibbs-type oscillation visible in the field at any tested value. This does not implement the trainable, spatially localized dissipation Appendix A.8 identifies as the more principled fix, but it does show that on this problem, at least, the evaluator's fixed default was leaving real accuracy on the table for the cost of a single constant.

Trainable viscosity, tested directly, makes Toro Test 3 worse. Appendix A.8 raises `underPINN`'s trainable-dissipation formulation ($\epsilon = \mathrm{softplus}(\eta)$, $\eta \in \mathbb{R}$ trainable, jointly optimized with the network) as the more principled alternative to a hand-fixed ν_{av}; we tested it directly on Toro Test 3, holding collocation strategy fixed (static, as the uniform arm above) so the viscosity effect is isolated from the point-placement effect. Over five seeds it is worse on every metric: relative L^2 rises from 0.445 ± 0.016 (fixed $\nu_{av} = 10^{-3}$) to 0.498 ± 0.013, and peak $|\partial_x \rho|$ falls from 7.1% to 2.2% of the exact value – a *less* sharp shock, not a sharper one. The learned ϵ converges to 0.0056 ± 0.0007 on every seed, roughly $5\times$ the fixed default, moving in exactly the wrong direction: nothing in the training objective penalizes ϵ directly, so gradient descent can trivially lower the PDE-residual loss by smoothing the problem (raising ϵ) rather than by fitting a sharper true solution, and it consistently takes that route. This is the opposite of what the artificial-viscosity sweep just above found for a *fixed* coefficient, and the two results are consistent, not contradictory: manually lowering ν_{av} helps because it removes dissipation the optimizer was not otherwise motivated to add, whereas leaving ϵ free lets the optimizer add more of it. A regularization term pulling ϵ toward zero (or an explicit annealing schedule, rather than a free parameter) looks like a precondition for trainable viscosity to help on this problem, not an optional refinement; we did not test either here and leave both to future work, with this result attached as a concrete reason to expect they are needed.

The single-run reproducibility episode, in full. On Toro Test 3, two runs at an *identical* seed and identical code, differing only in GPU floating-point non-determinism (JAX/XLA matmul and reduction ops are not guaranteed bit-reproducible run to run), disagreed on whether hybrid QR-DEIM-R was tied with uniform or the single worst arm (0.4548 vs. 0.5314). A small difference in the residual read back into NumPy at the first resampling event selects measurably different points, and

this compounds nonlinearly over the following epochs rather than staying a bounded perturbation. Tables 5–6 report five-seed means rather than any single run for this reason.

A.4 Gauss–Newton vs. Adam, in Full

Section 3.9 summarizes; this appendix gives the full comparison. On a small network (1–20–20–1, 481 parameters, chosen so an explicit Gauss–Newton system is cheap per step), we compared the Levenberg–Marquardt-damped Gauss–Newton update (Section 2.2) against Adam over five seeds (Table 7). At `underPINN`'s default learning rate (10^{-3}) Adam does not converge this oscillatory problem at any budget to 100,000 epochs; tuned to 10^{-2} it does (3×10^{-2} trains erratically, 10^{-1} diverges), and then matches or beats Gauss–Newton on both accuracy and wall-clock.

Table 7: Harmonic oscillator, relative L^2 vs. $\cos(\omega t)$, five seeds (mean $\pm$ std), 481-parameter network. Wall-clock on one GPU. Adam's learning rate is tuned to 10^{-2}; Gauss–Newton has no learning rate but carries its own damping schedule ($\lambda_0 = 10^{-2}$, $\times 3$ on a rejected step, $\times 0.5$ on an accepted one). Gauss–Newton converges in far fewer epochs but does not reach a better accuracy/wall-clock point than tuned Adam on this problem.

Method (budget)	Relative L^2	Wall (s)
Adam (10^{-2}), 50,000 ep	$2.4 \times 10^{-2} \pm 1.5 \times 10^{-2}$	~ 5
Adam (10^{-2}), 100,000 ep	$6.1 \times 10^{-3} \pm 3.4 \times 10^{-3}$	~ 9
Adam (10^{-2}), 200,000 ep	$\mathbf{3.0 \times 10^{-3} \pm 1.1 \times 10^{-3}}$	~ 19
Gauss–Newton, 1,000 ep	$4.6 \times 10^{-3} \pm 1.7 \times 10^{-3}$	~ 34
Gauss–Newton, 2,000 ep	$4.4 \times 10^{-3} \pm 1.3 \times 10^{-3}$	~ 49

Gauss–Newton reaches 4.6×10^{-3} in 1,000 epochs with no learning rate to set, but Adam at 10^{-2}/2×10^5 epochs is both slightly more accurate and $\sim 2.5\times$ faster (per-step cost ~ 0.1 ms vs. Gauss–Newton's ~ 30 ms dense linear solve). We report Gauss–Newton as an option that replaces a learning-rate search with a damping schedule that adapts on its own — not one that is free of hyperparameters, and not a faster or more accurate optimizer. One problem, one network size; we do not claim it generalizes.

Damping schedule and cost. Each step solves the damped least-squares problem $\min_\delta \|J\delta - r\|_2^2 + \lambda\|\delta\|_2^2$ with $J \in \mathbb{R}^{M\times P}$ the residual Jacobian, as the augmented system $\left[\begin{smallmatrix} J \\ \sqrt{\lambda} I \end{smallmatrix}\right]\delta = \left[\begin{smallmatrix} r \\ 0 \end{smallmatrix}\right]$ via `jnp.linalg.lstsq`, which avoids squaring the condition number as forming $J^\top J + \lambda I$ would. The damping is adapted Levenberg–Marquardt style: a step that lowers the loss is accepted and $\lambda \leftarrow \max(0.5\lambda, 10^{-10})$; one that does not is rejected, leaving the parameters unchanged, and $\lambda \leftarrow \min(3\lambda, 10^{10})$. Table 7 starts from $\lambda_0 = 10^{-2}$ (the library default is 10^{-3}); the growth and shrink factors are the library defaults. On this problem $M = 202$ residual rows (200 collocation points plus two initial-condition terms) against $P = 481$ parameters, so J is wider than tall and the undamped problem is rank-deficient — the damping is what makes each step well-posed, not merely a stabilizer.

J is formed by forward-mode differentiation (P Jacobian–vector products) and the augmented system solved densely, at $O\big((M+P)P^2\big)$ time and $O\big((M+P)P\big)$ memory per step. That is $\sim 1.6 \times 10^8$ floating-point operations here, which is why the step costs ~ 30 ms. The same dense step applied to the networks elsewhere in this paper would be roughly 10^{13} operations and 2.4 GiB for the matrix alone on the Burgers MLP of Table 1 ($P \approx 1.7 \times 10^4$, $M \approx 2.1 \times 10^4$), and roughly 10^{15} operations and 57 GiB on Pipe Flow ($P \approx 6.7 \times 10^4$, $M = 1.6 \times 10^5$), before any solver workspace. The implementation is therefore restricted to small diagnostic networks. Scaling second-order training to production PINNs would need a matrix-free solver — Krylov iterations such as LSMR or conjugate gradients built from Jacobian–vector and vector–Jacobian products, or a structured curvature approximation — none of which `underPINN` currently implements.

A.5 EAGER-MODE PYTORCH COMPARISON, IN FULL

Section 3.2 summarizes; this appendix gives the full comparison. To test the eager-vs-compiled comparison directly, we implemented three representative problems – 1-D Burgers, 3-D Pipe Flow, 2-D Ramp NS (SBLI) – independently in (i) eager-mode PyTorch, (ii) JAX with per-epoch `jax.jit`, and (iii) JAX with the loop fused via `jax.lax.scan`, matching network, collocation counts, loss weighting, and derivative-computation approach to the corresponding evaluator elsewhere in this paper: Pipe Flow and Ramp NS's PyTorch/JAX residuals were checked numerically against the production JAX PDE classes on shared random points before any timing was trusted (max abs. difference $\sim 5 \times 10^{-4}$ against values ~ 0.1–0.9, consistent with float32 AD-order noise between `jax.jacfwd` and `torch.func.jacrev`, not a formula mismatch); Ramp NS uses static collocation only (uniform + boundary-layer pools, no RAR-D resampling) for a fair three-way comparison, since RAR-D's non-jittable NumPy step cannot be fused into `jax.lax.scan` on any framework regardless (Appendix A.7). All three variants run for $5{,}000$ epochs on the same GPU, three seeds each; for `jax.lax.scan` we report both a first call (one-time trace/compile included) and a second, identical-shape call (steady-state only), so compilation overhead appears explicitly rather than folded into the headline number.

Table 8: Measured wall-clock training throughput: PyTorch (eager and `torch.compile`) vs. JAX (`jit` and `lax.scan`), 5,000 epochs, mean $\pm$ std over three seeds. Speedups are relative to the PyTorch eager baseline (mean/mean). The `torch.compile` column necessarily uses the `torch.func` formulation, since `torch.compile` cannot capture the classic `autograd.grad` double backward the eager column uses; it runs on the `inductor` backend for Ramp NS and falls back to `aot_eager` for Burgers and Pipe Flow (Appendix A.5). Burgers is the one problem where that formulation is slower than eager to begin with, so compiling it does not recover the eager baseline.

Problem	PyTorch eager (ms/epoch)	PyTorch `torch.compile` (ms/epoch, ×)	JAX `jit` (ms/epoch, ×)	JAX `scan`, steady-state (ms/epoch, ×)	JAX `scan`, incl. compile (ms/epoch, ×)
1-D Burgers	14.13 ± 0.18	26.62 ± 0.28 (0.53×)	13.71 ± 0.06 (1.03×)	13.66 ± 0.02 (1.03×)	14.05 ± 0.03 (1.01×)
3-D Pipe Flow	12.76 ± 0.42	9.36 ± 0.40 (1.36×)	0.84 ± 0.01 (15.20×)	0.73 ± 0.01 (17.39×)	1.62 ± 0.02 (7.88×)
2-D Ramp NS (SBLI)	61.18 ± 0.71	35.60 ± 0.09 (1.72×)	4.44 ± 0.07 (13.77×)	4.44 ± 0.02 (13.79×)	6.17 ± 0.05 (9.91×)

Speedup tracks the ratio of dispatch overhead to per-step compute, not problem complexity (same GPU for both frameworks): Burgers is already compute-bound, so compilation helps least ($\sim 1.03\times$); Pipe Flow's small, fragmented per-epoch point sets are dominated by kernel-launch latency, the regime `jax.jit` helps most (15–17×, the largest measured – from this deliberately small, cross-framework-matched batch configuration, not the 7–20 ms/epoch Figure 1 reports for `underPINN`'s own production batch sizes); Ramp NS lands between the two ($\sim 14\times$). PyTorch eager's own seed-to-seed std (0.18–0.71 ms) is 3–40× larger than either JAX variant's (0.01–0.07 ms) – dispatch-bound eager execution is measurably noisier run to run than a compiled step, a second-order finding this three-seed pass surfaces that a single run could not have. The same sensitivity shows up between sessions, and more sharply: rerunning this table on an otherwise idle GPU moved every *eager* number down ($18.09 \to 14.13$, $15.93 \to 12.76$, $65.48 \to 61.18$ ms/epoch) while leaving both JAX variants within a few percent, on identical code, commit, PyTorch build and GPU. We report the idle-machine numbers, which are the tighter-variance ones, and note that they *lower* this paper's own headline speedup; a dispatch-bound baseline's wall-clock is partly a property of the machine it shared, which is worth knowing for anyone reproducing a throughput comparison of this kind. Comparisons against compiled JAX PINN frameworks are in Section 3.4; `torch.compile` is reported below. `jax.lax.scan`'s steady-state step beats plain `jax.jit` only on Pipe Flow (0.73 vs. 0.84 ms/epoch, $1.15\times$, well outside seed noise) and matches it on Burgers and Ramp NS; counting its one-time compilation it is slower on all three, at 1.02–1.93× `jax.jit`'s time for a single $5{,}000$-epoch run. `jax.jit` is therefore the suite default – both because of this, and because Ramp NS's RAR-D resampling calls non-jittable NumPy outside any compiled boundary regardless (Appendix A.7 gives the full compilation-overhead accounting).

This is a revised, independently reimplemented version of an earlier single-run pass: the original PyTorch/JAX scripts that produced that pass's numbers were not preserved (an ephemeral, uncommitted scratchpad artifact, as with several other single-run figures in this paper's first draft), so rather than leave the comparison single-run, we wrote new matched implementations from the corresponding evaluator's own architecture, collocation, and loss specification, verified as above, and report the

resulting numbers here in place of the originals – they differ from the first-draft values by roughly 1–40% depending on problem and variant (largest on Pipe Flow's PyTorch-eager number), consistent with ordinary reimplementation and multi-seed noise rather than a change in the qualitative pattern (`jax.jit` and `jax.lax.scan` both beat eager PyTorch on every problem, by a wide margin on Pipe Flow and Ramp NS and only marginally on the compute-bound Burgers, and `lax.scan`'s steady-state step beats plain `jax.jit` only on Pipe Flow, and not once compilation is counted). The numbers reported here are from a later rerun of those same scripts on an idle machine, for the reason given in the preceding paragraph.

Compiled PyTorch (`torch.compile`). Eager PyTorch is not the strongest baseline available: PyTorch 2's `torch.compile` (Ansel et al., 2024) captures graphs through TorchDynamo and generates fused kernels through TorchInductor, targeting the same per-epoch dispatch overhead that `jax.jit` removes. If that mechanism were the whole story, a compiled PyTorch baseline should close most of the gap in Table 8. We therefore measured it on all three problems, three seeds each, at the same 5,000 epochs and matched architecture, collocation counts, loss weighting, and optimizer. It narrows the gap substantially on two of the three, and closes it on none.

The first finding is structural rather than numerical: `torch.compile` cannot capture the standard PINN formulation at all. Computing a PDE residual via `torch.autograd.grad(..., create_graph=True)` requires double backward, which `aot_autograd` does not currently support. We verified this on both the `inductor` and `aot_eager` backends, and with `torch.compile` applied to the loss function and to the module separately. Compiling the classic formulation is therefore not a configuration we neglected to try, but one PyTorch does not presently offer. The compiled column of Table 8 consequently uses the `torch.func` (`jacrev`/`hessian`/`vmap`) formulation throughout — PyTorch's structural analogue of the JAX transforms `underPINN` uses, and the only one `torch.compile` accepts.

Where the per-step work is dispatch-bound, compiling helps materially: Pipe Flow improves from 12.76 to 9.36 ms/epoch (1.36×) and Ramp NS from 61.18 to 35.60 ms/epoch (1.72×). This is the effect one would expect from graph capture, and it is real. It is nonetheless not sufficient: on those same two problems `jax.jit` runs at 0.84 and 4.44 ms/epoch, so `underPINN` remains 11.1× and 8.0× faster than the *compiled* PyTorch baseline, not merely the eager one. Burgers runs the other way: its `torch.func` formulation is slower than classic eager autograd to begin with (28.49 vs. 14.13 ms/epoch) and compiling recovers only part of that (26.62), so no available compiled configuration beats eager there — though Burgers is also the problem where JAX's own margin is smallest (1.03×), so little turns on it either way.

Two caveats bound this result. `torch.compile` ran on `inductor` for Ramp NS but fell back to `aot_eager` for Burgers and Pipe Flow, on a reproducible ZeroTensor-immutability codegen bug in the generated backward kernel; those two numbers therefore reflect graph capture *without* Inductor's kernel generation and should be read as lower bounds. That Ramp NS compiled cleanly under Inductor indicates the bug is specific to the `vmap(jacrev)` + `vmap(hessian)` composition rather than general to this workload class. Separately, `torch.jit.script` could not be measured on this machine at all, since nvrtc rejected the GB10 architecture. We claim only that compiled PyTorch narrowed but did not close the gap in these configurations, not that no compiled PyTorch configuration could close it.

A.6 Limitations, in Full

Section 4 summarizes; this appendix gives the full discussion, unabridged, with the specific numbers behind each point.

First, most results in this paper are now multi-seed, but some remain single-run (`underPINN`'s Pipe Flow comparison rows, the loss curves and field plots, the Ramp NS viscosity sweep, and the initializer check on Burgers); Section 3.8 shows why that matters – across five seeds on Toro Test 3 the four closest arms (uniform, both RAD arms, hybrid QR-DEIM-R) have seed-to-seed spreads of 0.016–0.029 against gaps between their means of at most 0.032, so a single-run ranking of them is not meaningful, and we separately observed an identical-seed, identical-code rerun reverse which arm looked best (Appendix A.3).

Second, we did not present ablations isolating the individual contribution of every architectural component (gated MLPs, Fourier embeddings, FBPINN decomposition) on a shared benchmark problem, though Section 3.8 and Section 3.9 now provide that ablation specifically for adaptive collocation (QR-DEIM-R vs. RAD vs. uniform, five arms, five seeds, six problems) and for second-order training (Gauss–Newton vs. Adam, five seeds on one problem); QR-DEIM-R did not improve on its established baseline on any problem, while RAD did clearly on the two problems with a localized, uncapped feature to exploit (Burgers, the lid-driven cavity), marginally but on every seed on diffusion (about 9%), and not on the rest, a distinction the single-run version of this ablation could not have drawn reliably. That ablation is also confined to one family of adaptive collocation — residual-magnitude reweighting and deterministic subset selection over a fixed candidate pool. Generative approaches that resample the pool from a learned residual density, DAS-PINNs (Tang et al., 2023) in particular, are untested here, as are Celaya et al. (2025)'s published QR-DEIM strategies, which are the closest prior art to QR-DEIM-R and the more directly relevant of the two omissions (Section 3.8). Our per-problem results elsewhere show the remaining components work together on their respective problems, but do not by themselves establish which component is responsible for a given improvement.

Third, our `jinns` comparison (Section 3.4) covers six problems, narrower in scope than the eight-problem PhysicsNeMo comparison; both are now three-seed per problem. Multi-seeding mattered on both sides: it dissolved two of `jinns`' three apparent accuracy advantages (Section 3.4), and it shrank PhysicsNeMo's apparent ODE Harmonic advantage (a single-run 0.043 became 0.318 ± 0.217) and turned Burgers into a tie (Section 3.3); `underPINN` is faster per epoch than `jinns` on all six ($1.07\times$–$2.36\times$), Pipe Flow at a reduced, disclosed scale (20,000 points, 30,000 epochs, plain MLP on both sides) relative to Table 1's full configuration, since a from-scratch jinns port at the full 100,000-point/GatedMLP/70,000-epoch scale was not tractable in this pass. Burgers' result is itself a reversal from an earlier version, where jinns measured faster before a later fix to `underPINN`'s own residual code (Section 3.4) – a reminder that a library-level speed fix needs checking against every downstream comparison script separately, not assumed to propagate automatically. We audited this comparison for a hidden default of the kind (`weight_norm`) that materially changed the PhysicsNeMo accuracy results, checking precision, loss aggregation, per-step batch sizes, boundary-condition semantics, and Adam's hyperparameters; all matched except weight/bias initialization scheme and a minibatching-semantics difference (jinns cycles a shuffled pool in chunks, `underPINN` draws a fresh random sample every step), neither of which changes per-step compute cost.

We benchmark against three compiled JAX frameworks — `jinns`, DeepXDE's JAX backend (Lu et al., 2019), and PINNs-JAX (Akbarian Bafghi & Raissi, 2024) — but not against JAX-PI (Wang et al., 2023), which remains untested here. They agree with each other in a way worth stating: `underPINN` is faster than all three on every problem measured, by 1.07–$2.36\times$, and is the most accurate framework on none of the five scored rows (Section 3.4). The throughput result should therefore not be read as a general claim about the JAX ecosystem — against compiled JAX peers the margin is a small multiple, not the 1.23–$14.2\times$ obtained against eager-mode PhysicsNeMo.

Two of those three comparisons are narrower than the `jinns` one, and for stated reasons. DeepXDE covers four of the six problems: the ODE harmonic oscillator is not expressible under its JAX backend (Table 2), and we did not port Pipe Flow. PINNs-JAX is throughput-only and on Burgers alone, because it is architected around a reference solution on a mesh — its `Mesh` requires a `root_dir` and `read_data_fn`, and its initial- and boundary-condition samplers read their target values out of that array — so it trains with solution values present on the IC/BC manifold where the other three impose those conditions from closed-form functions. Per-step cost is unaffected by where a boundary value came from, so throughput remains comparable; final accuracy does not, and we do not report it rather than presenting a number that would flatter or penalize one framework for a difference in what it was shown. Relatedly, we report no memory measurements for any framework: peak and steady-state device memory are not instrumented anywhere in this paper, so scaling behaviour under memory pressure is uncharacterized for `underPINN` and for every baseline.

The repository ships 22 example cases, of which we present the nine of Table 4 for benchmarking. The others include cases with complex geometry and further physics: a patient-specific 3-D intracranial aneurysm whose geometry is built entirely from STL surfaces (watertight interior, inlet and outlet caps, vessel wall, and a mid-stream plane carrying an integral mass-flow constraint), an abdominal aortic aneurysm with Carreau rheology, an airfoil, a k–ϵ closure, and inverse-problem and neural-operator paths.

Two community suites would test generalization beyond a hand-configured suite more directly, and we use neither. PINNacle (Hao et al., 2024) is the closer fit: over twenty PDEs selected for complex geometry, multi-scale behaviour, nonlinearity and high dimensionality, with a common evaluation protocol, and built on DeepXDE — against whose JAX backend we already benchmark (Section 3.4), so the comparison harness is partly in place. PDEBench (Takamoto et al., 2022) overlaps less than its name suggests: it is organized around large pre-generated simulation datasets for training data-driven surrogates, with PINNs as one baseline among FNO and U-Net, so its central contribution — the data — is not something a data-free collocation framework consumes. Its PDE specifications and initial/boundary conditions would still be usable as problem definitions. We flag PINNacle as the more direct answer to whether these results generalize beyond a hand-configured suite.

Fourth (Section 3.7), the 1-D Toro Test 3 blast wave retained a relative L^2 error around 33% even at $60{,}000$ epochs, and the viscous Ramp NS shock-boundary-layer problem required substantially more training than the smoother diffusion and wave problems of comparable dimensionality. We view this as a limitation of the tested configuration – a fixed, non-adaptive artificial-viscosity coefficient and a single global network for a multi-discontinuity unsteady problem – rather than a claim that these classes of problems are unresolvable with the architecture: a direct viscosity sweep on Ramp NS (Appendix A.3) found real, if plateauing, headroom in the fixed-coefficient setting itself, and a direct test of the framework's global trainable-dissipation option on Toro Test 3 (Appendix A.3) made the solution worse rather than better, for a specific, fixable reason (no regularization on ϵ) rather than a dead end – the FBPINN-decomposition option remains untested on either shock problem.

A.7 Framework Feature Detail

Sections 2.2–2.3 summarize; this appendix gives the full motivation and mechanism for each feature.

`jax.lax.scan` compilation overhead (Section 3.2). `jax.lax.scan`'s one-time compilation cost (about 2–9 s across the three matched problems, from Table 8's "incl. compile" and "steady-state" columns) is non-trivial relative to a single $5{,}000$-epoch run: for Pipe Flow it exceeds the entire steady-state run time, so a one-shot run of that problem completes at $7.88\times$ over PyTorch rather than $17.39\times$ (Table 8's "incl. compile" vs. "steady-state" columns).

Network architectures (Section 2.2). Gated MLP (GatedMLP): following Wang et al. (2022), this architecture uses two input encoders (U and V) that are gate-blended into every hidden layer, so that the original spatial coordinates are re-injected throughout the network. This is intended to reduce gradient attenuation in deep networks trained on stiff PDEs (e.g., 3-D Navier–Stokes formulations). **Fourier feature MLP (FourierMLP):** following Wang et al. (2021b), input coordinates are mapped through random Fourier embeddings before the standard MLP, intended to mitigate the low-pass spectral bias of MLPs on oscillatory targets (e.g., Helmholtz or high-Reynolds-number flows). The scaling factor (σ) of the embedding is trainable rather than fixed, allowing it to be adapted during optimization rather than tuned by hand. **Domain decomposition (FBPINN):** following Moseley et al. (2023), the spatial domain is partitioned into overlapping subdomains via smooth, differentiable sigmoid partition-of-unity window functions, with an independent network assigned to each subdomain. This is intended to prevent high-gradient behavior in one region (e.g., a shock) from degrading optimization in the rest of the domain, and to make it feasible to train on larger or more geometrically complex domains than a single global network. **Hybrid attention and SimpleGate modulation:** `underPINN` also includes a gated-residual attention block (HybridAttention), which can be deployed inside individual FBPINN subdomains, and a SimpleGate module that performs an element-wise multiplication of the hidden state by a learnable gate. These are intended as a lower-parameter-count alternative to standard dense layers for modulating representational capacity near sharp transitions; we do not present an ablation isolating their individual effect in this paper.

Training and parameterization strategies (Section 2.2). Windowed time-marching: training a single network across the full time domain of a long-horizon, 3-D unsteady PDE (e.g., pulsatile flow) can exceed GPU memory and slow convergence. We provide an automated windowed time-marching mode: the temporal domain is split into sequential windows, and the network for each window is warm-started from the parameters of the previous window, with the previous window's end-state enforced as the initial condition for the next. We integrate this with the checkpointing mechanism below so that each window can be restarted independently. **Trainable artificial viscosity:** for

compressible Euler formulations with discontinuities (e.g., oblique shocks, Sod-type shock tubes), we support a global Laplacian dissipation term ($-\epsilon\nabla^2 U$) on the conserved variables, following standard artificial-viscosity regularization practice. Rather than fixing ϵ by hand, it can optionally be parameterized as $\epsilon = \mathrm{softplus}(\eta)$, with $\eta \in \mathbb{R}$ a trainable scalar, and learned jointly with the network weights. Left unregularized this is ill-posed: nothing in the residual penalizes a large ϵ, so gradient descent can lower the PDE loss by smoothing the problem rather than by fitting a sharper solution, and on Toro Test 3 it does exactly that (Appendix A.3). A penalty such as $\mathcal{L}_{\mathrm{total}} = \mathcal{L}_{\mathrm{PINN}} + \lambda_\epsilon\,\epsilon^2$, or an annealing schedule on ϵ, is a precondition for the option to help; neither is implemented or tested here. **Log-parameterization for inverse discovery:** for inverse problems (e.g., discovering a thermal diffusivity α from sparse sensor data), the framework supports log-parameterization ($\log\alpha$), a standard technique for enforcing positivity of a discovered physical parameter without a non-differentiable constraint. **Gauss–Newton natural-gradient training:** as an alternative to first-order Adam updates, we support a Levenberg–Marquardt-damped Gauss–Newton step, which forms an augmented least-squares system from the per-point residual Jacobian rather than the explicit Gram matrix $J^\top J$, avoiding the squared condition number that an explicit-normal-equations formulation would incur. It is a dense solver, $O\big((M+P)P^2\big)$ per step, and practical only for small networks; Appendix A.4 gives the damping schedule and the cost at this paper's network sizes.

QR-DEIM-R, formally (Section 2.3). Given the current parameters θ, a target pool size n and a candidate count $n_c \geq n$ (the same $5n$ RAD uses), one resampling call proceeds as follows (`underPINN.utils.sampling.qr_deim_resample`).

1. Draw $x_1, \ldots, x_{n_c}$ uniformly over the domain and evaluate the residual $r_i = r(x_i;\theta) \in \mathbb{R}^m$, where m is the number of residual components (m=1 for Burgers, 3 for Toro Test 3, 4 for Ramp NS).
2. Form the feature matrix $U \in \mathbb{R}^{n_c \times r_0}$ with rows $U_{i,:} = \big[\, r_i^\top,\ \|r_i\|_2\, x_i^\top \,\big]$, so $r_0 = m + d$ for d input coordinates. The coordinate columns let two points of similar residual but different location be told apart. Coordinates enter in each problem's own non-dimensional units, scaled only by the residual magnitude; no further normalization is applied, so the relative weight of the coordinate columns depends on the domain's extent.
3. *Anchors.* Take a column-pivoted QR of $U^\top \in \mathbb{R}^{r_0 \times n_c}$, $U^\top \Pi = QR$, and keep the first $a = \min(r_0, n)$ pivot indices — the classical Q-DEIM selection rule (Drmač & Gugercin, 2016). There is no rank-truncation criterion: all r_0 pivots are kept.
4. *Fill.* Take a thin QR $U = \tilde{Q}\tilde{R}$ with $\tilde{Q} \in \mathbb{R}^{n_c \times r_0}$ and form leverage scores $\ell_i = \|\tilde{Q}_{i,:}\|_2^2$, which sum to $\mathrm{rank}\, U \leq r_0$. Draw the remaining $n - a$ indices i.i.d. *with replacement* from $p_i = \ell_i / \sum_j \ell_j$ (Drineas et al., 2006).
5. Return the n points indexed by the anchors and the fill.

Cost is $O(n_c r_0^2)$ time and $O(n_c r_0)$ memory, never forming an $n_c \times n$ matrix.

Two consequences follow directly from this specification and bear on how Section 3.8's result should be read. First, the deterministic DEIM step contributes only r_0 points — 3 on Burgers, 5 on Toro Test 3, 6 on Ramp NS — out of the 6,000–40,000 it selects, at most 0.1% of the pool. In practice QR-DEIM-R is therefore almost entirely leverage-score sampling on a residual-plus-residual-weighted-coordinate basis, and the ablation tests that far more than it tests DEIM selection. Second, leverage measures how far a point's feature vector lies outside the span of the others, not how large its residual is: a high-residual point in a direction the basis already spans can receive little weight. That is one plausible reason it underperforms RAD's direct magnitude weighting, though we have not isolated it.

Checkpointing mechanism (Section 2.3). Configuration fingerprinting: each experiment is defined by a YAML configuration. On initialization, `underPINN` computes an MD5 hash of the full configuration state and stores it alongside the run's checkpoint artifacts. (We use MD5 here for configuration fingerprinting, not for any security-sensitive purpose; a cryptographic hash such as SHA-256 could be substituted without changing the mechanism.) **Restart on re-invocation:** if a run is interrupted, re-running the same command causes `underPINN` to compare the current configuration hash against the stored one. If they match, the `RestartManager` restores the optimizer state, epoch counter, and loss history from the last checkpoint, and training resumes from

that point. If the configuration was changed (e.g., a modified learning rate), the hash mismatch causes the framework to start a fresh run rather than resume. **On-demand device memory allocation:** by default, XLA pre-allocates a large fraction ($\sim$90%) of available GPU memory, which can prevent several lightweight models from sharing a single device. We configure XLA for on-demand memory growth instead, so that multiple concurrent jobs (e.g., a hyperparameter sweep) can share a single GPU.

A.8 Why Toro Test 3's Error Plateaus Around 33%

Section 3.7 summarizes; the persistent error is consistent with several compounding factors rather than a training instability. This section's title number itself changed during revision: an earlier single-run pass reported this plateau as "above 45%"; a three-seed rerun at the same $60{,}000$-epoch budget gives a consistent $32.9\% \pm 0.2\%$ instead (Section 3.5). We flag this prominently rather than quietly, since it is the same lesson Section 3.8 draws from the adaptive-collocation ablation – a single PINN training run, even for a paper's own headline number, is not by itself a reliable measurement – caught here in the number itself, not only in a comparison between resampling strategies. First, spectral bias: a $\tanh$-activated MLP is a smooth, band-limited basis, whereas the true solution contains genuine jump discontinuities (via the Rankine–Hugoniot conditions) at both the shock and the contact surface; the network can only approximate a jump by compressing a continuous transition into a layer whose width is set by the artificial-viscosity coefficient ν_{av}, not by network capacity, so the residual error is structurally concentrated in – and dominated by – a thin band around each discontinuity rather than reflecting a global fitting failure. Second, Toro Test 3 fixes $\nu_{av} = 0.001$ rather than using `underPINN`'s trainable-dissipation formulation (Section 2.2), so the dissipation width is not adaptively matched to local shock strength as training progresses. Third, the difficulty is compounded by the problem being genuinely unsteady: the 2-D Ramp benchmark uses the same fixed $\nu_{av} = 0.001$ yet reaches roughly 1% error by $20{,}000$ epochs (Figure 4b) because it only needs to resolve a single stationary shock in space, whereas Toro Test 3 requires one global network to simultaneously represent a moving shock, a moving contact discontinuity, and an expanding rarefaction fan – each propagating at a different characteristic speed – across the entire (x, t) domain. Finally, the metric weights the three fields very unequally. It is computed on the non-dimensional state, in which the pressure's five-decade range is compressed to $\tilde{p} \leq 1$ while the post-shock density spike reaches $\tilde{\rho} = 6$; density therefore carries about 82% of the exact solution's norm, pressure 11% and velocity 8%. The plateau is thus mostly a density error, and specifically the narrow density spike of which no arm recovers more than about 60% (Table 5) — not, as an earlier version of this paragraph argued, an artifact of the pressure field's dynamic range. A per-field metric would weight the three equally and could report a quite different number. Closing this gap likely requires explicit shock-fitting or characteristic tracking rather than a purely diffusive penalty, an FBPINN-style decomposition (Section 2.2) that isolates each discontinuity into its own subdomain, or a *spatially localized* trainable dissipation coefficient in place of a single fixed global ν_{av} – directions we still leave to future work, though not naively: Section 3.8 tests the simpler *global* trainable-ν_{av} version directly and finds it makes this problem worse, because nothing in the objective penalizes a large ϵ, so unregularized training inflates it rather than sharpening the solution.

A.9 PhysicsNeMo Comparison: Cost of the Weight-Normalization Standardization

Configuration audit. We installed NVIDIA PhysicsNeMo Sym (NVIDIA Corporation, 2024) (formerly Modulus) on the same machine and ran it on eight of the nine problems in Table 4 — all except the 2-D inviscid Ramp (1-D Burgers, 1-D Wave, 2-D Steady Heat, 2-D Helmholtz, ODE Harmonic, 3-D Pipe Flow, 1-D Toro Test 3, 2-D Ramp NS) — matching physics, network depth/width, collocation counts, and per-step batching as closely as PhysicsNeMo's constraint-based API allows, and scoring both frameworks against the same exact or manufactured reference solution. Full case-by-case configuration notes, including every point of divergence we could not close (PhysicsNeMo's default exponential learning-rate schedule versus our cosine schedule; architecture choices with no PhysicsNeMo equivalent, such as FourierMLP and GatedMLP), are recorded case by case in the accompanying benchmark repository.

We caught, rather than initially avoided, a real architecture confound worth reporting: every case script's first version built PhysicsNeMo's network as a plain `FullyConnectedArch(...)`, doc-

umented as "the same architecture as `underPINN`'s MLP." That claim was wrong – PhysicsNeMo's `FullyConnectedArchCore` defaults to `weight_norm=True` (Salimans & Kingma, 2016), applying weight normalization to every hidden layer unless explicitly disabled, which changes both the trained model and per-step compute cost, directly relevant to Table 1's throughput claim; every "plain MLP vs. plain MLP" comparison was therefore silently architecture-mismatched. We found this only after being asked directly why the two frameworks' outputs differed on ostensibly identical networks; every number in Table 1 is the corrected (`weight_norm=False`) rerun, and we report it as a caution about architecture-matching claims generally, not a criticism specific to PhysicsNeMo. The correction is not free, and we did not at first measure its price: matching the architecture for a fair *throughput* comparison removes a default that PhysicsNeMo's accuracy depends on for one problem, by $69\times$ on Helmholtz (Table 9) — a second instance of the hidden-default confound, this time one we introduced ourselves while correcting for the first – a default this consequential is easy to miss precisely because it appears in neither side's problem-specific configuration. We also audited weight/bias initialization, rescaling, and activation directly against PhysicsNeMo's source; all matched except weight initialization scheme, left uncorrected since it does not affect per-step compute cost. One execution setting also differs from PhysicsNeMo's default, and it bears on throughput: every PhysicsNeMo run here uses `jit=false`. Its default (`jit=true`) fuses activation functions through TorchScript's nvFuser, which fails to compile on this GPU (Blackwell, `sm_121`; NVRTC rejects the target architecture, the same failure that rules out `torch.jit.script`, Appendix A.5). PhysicsNeMo's timings in Table 1 are therefore for its unfused path; on hardware where its default path compiles, they would be expected to be lower. Burgers and Helmholtz's `underPINN` figures additionally reflect a later residual-code fix (three independent AD transforms fused into one shared `jax.vjp`/`jax.jvp` pass, verified under the actual training-step gradient rather than an isolated forward call, which can mislead – see `benchmarks/rebuttal/README.md`, §6, in the accompanying repository).

Section 3.3 sets PhysicsNeMo's `weight_norm=False` so both frameworks run the same plain MLP, which is the right choice for comparing per-epoch cost. It is not neutral for accuracy, and Table 9 measures how it moves accuracy, on four of the problems.

Table 9: PhysicsNeMo relative L^2 with its weight normalization disabled (the Table 1 runs) and at its native default, three seeds each (mean $\pm$ std), identical seeds, step budgets (Table 1: 5,000; Helmholtz 10,000; ODE Harmonic 3,000) and settings otherwise. Both arms use `jit=false` (Section 3.3). "Effect" is the disabled arm's error relative to the native one.

Problem	`weight_norm=False`	`weight_norm=True` (native)	Effect of disabling
2-D Steady Heat	0.0158 ± 0.0010	0.0271 ± 0.0086	$1.7\times$ better
1-D Wave	0.405 ± 0.043	0.413 ± 0.016	none
2-D Helmholtz	1.085 ± 0.003	0.0157 ± 0.0026	$69\times$ worse
ODE Harmonic	0.318 ± 0.217	0.530 ± 0.054	none within noise

Disabling weight normalization is therefore costly on one of the four problems, Helmholtz, where it accounts for PhysicsNeMo's failure to converge; it is neutral on Wave and ODE Harmonic and helps on Steady Heat. It buys 2–5% per-epoch throughput. We keep the disabled configuration as Table 1's headline because that table's claim is about throughput at matched architecture; its Helmholtz accuracy column should be read with this appendix beside it.

An earlier version of this appendix reported a different outcome (disabling weight normalization costing accuracy on three of four problems, by 2–$46\times$) together with a large cold-start effect on PhysicsNeMo's timing. Both came from a harness error, not from PhysicsNeMo. PhysicsNeMo names its output directory after the command-line overrides only, and resumes from any checkpoint it finds there; our earlier seeds shared one directory, so every run after the first in each arm restored the finished checkpoint and trained for zero steps, re-scoring the first seed's model and timing an empty loop. Each run in the table above writes to its own directory, and its training log confirms it trained from step 0. We note the error because it is an instance of this paper's theme: a default that appears in no configuration file silently changed what was measured.

A.10 JINNS COMPARISON: CONFIGURATION AUDIT DETAIL

Section 3.4 summarizes; this appendix gives the full configuration audit.

Multi-seed history. Accuracy inverts the ordering, but how firmly depends on how many seeds the comparison rests on — and multi-seeding the jinns baseline changed two of its three apparent wins. An earlier version of Table 2 carried jinns as a single run, which put it ahead of `underPINN` on Burgers (0.038 vs. 0.219), 2-D Steady Heat and 2-D Heat Unsteady (0.024 vs. 0.070). Across three seeds jinns' accuracy turns out to be highly seed-sensitive: 0.134 ± 0.102 on Burgers (a $7.1\times$ spread between its best and worst seed) and 0.054 ± 0.032 on Heat Unsteady ($4.1\times$). Its mean is still the lower one on both, but the intervals now overlap `underPINN`'s, so neither is a win this experiment can distinguish. The single-run value happened to be jinns' best seed on three of the five problems — not a selection effect, simply what one draw gives.

We flag the seed sensitivity rather than absorb it, because it is this paper's own thesis landing on its own comparison table: Section 3.8 argues that single-run PINN evaluation can reverse a conclusion, and here it did. The throughput comparison is unaffected — across the same runs jinns' per-epoch cost varies by 1.03–$1.24\times$ between seeds while its accuracy varies by 1.3–$7.1\times$, which is what one expects when per-step cost is fixed by the computation graph and accuracy by where the optimizer lands. A single-seed baseline is therefore a defensible shortcut for a speed table and not for an accuracy one. Pipe Flow runs at reduced scale (20,000 pts/30,000 ep vs. Table 1's 100,000/70,000); underPINN's own accuracy at *this* reduced, jinns-matched scale is unmeasured, so that row draws no cross-framework comparison (Section 3.5 separately reports Pipe Flow's accuracy on `underPINN`'s own production configuration, at yet another budget, $60{,}000$ epochs). This covers six problems, three seeds each on every framework.

Two `jinns` API details would have silently broken the comparison had we trusted docstrings over source: `jinns` normalizes the network's time input to $[0, 1]$ and takes T_{max} as a separate multiplier inside the residual for time-dependent problems (a convention we *conformed to*, building each domain over $\tilde{t} \in [0, 1]$ and passing T_{max}, rather than one we disabled — unlike PhysicsNeMo's weight normalization, nothing was removed from `jinns`), and its convective-term operator (`_u_dot_nabla_times_u_rev`) is hard-coded to 2-D inputs, unusable for the 3-D Pipe Flow case, whose convective term is therefore hand-written via `jax.jacfwd` while jinns' own dimension-general `vectorial_laplacian_rev`/`divergence_rev` operators still handle its diffusion/continuity terms. `jinns.loss.LossODE`'s built-in initial-condition mechanism also supports only a value constraint, not the ODE Harmonic problem's derivative condition $u'(0) = 0$; that comparison and Pipe Flow's (whose cylindrical geometry does not fit jinns' box-shaped boundary-condition machinery) both bypass `jinns.solve` for a hand-written training loop around jinns' PINN class and dynamic-loss extension points – genuine jinns machinery throughout, just not its top-level solver where the API does not fit. Every custom residual was verified before being trusted: Pipe Flow's against the exact analytic Hagen–Poiseuille solution (available here since the nonlinear term vanishes exactly for fully-developed flow), returning exactly zero at five random interior points; the ODE case's hand-derived T_{max}^2 scaling (a second-order-in-time equation scales differently than the first-order cases) by confirming the trained network converges to a solution tracking $\cos(\omega t)$, which a sign or scaling error would have prevented outright.

Burgers' comparison illustrates why a residual-computation fix needs checking end-to-end, not just at the library level: Table 2's underPINN figure is a rerun after the derivative-fusion fix described in Appendix A.9; `jinns` initially measured faster (10.27 vs. the pre-fix 14.62 ms/epoch), a result the fix reversed once applied consistently.

The settings audit checked numerical precision (float32 on both sides), loss-term aggregation and per-step minibatch sizes (matched, with one exception below), the boundary condition (a soft penalty on both sides, not a hard constraint as one might guess from "Dirichlet"), and Adam's hyperparameters (standard optax defaults, mathematically equivalent compositions), all directly against `jinns`' source. Two real settings differences survived this audit. First, weight/bias initialization: `jinns`' layers (`eqx.nn.Linear`) draw weights *and* biases from $\mathcal{U}(-1/\sqrt{\text{fan-in}}, 1/\sqrt{\text{fan-in}})$, while `underPINN`'s Flax `Dense` layers use LeCun-normal weights and zero biases. Second, minibatch cycling. Both frameworks draw $2{,}048$ of $5{,}000$ interior points per step without replacement *within* the step; they differ *across* steps. `underPINN` draws each step's batch independently, whereas

`jinns` keeps a shuffled pool and a cursor and reshuffles only when the pool is spent. Because its slice clamps at the end of the pool, a `jinns` pass is three chunks covering every point (about $1{,}100$ of them twice) rather than strictly non-overlapping ones. Neither difference changes per-step compute cost.

We tested both directly on 2-D Steady Heat, the one problem on which `jinns` remains distinguishably more accurate after multi-seeding (Section 3.4), by rerunning `underPINN`'s own Table 2 configuration with each `jinns` convention swapped in and nothing else changed (Table 10). The unmodified arm reproduces the reported 0.0025 ± 0.0003, which is what makes the others interpretable.

Table 10: 2-D Steady Heat, three seeds (mean $\pm$ std): `underPINN`'s Table 2 configuration with `jinns`' initializer and/or its cyclic minibatching substituted. The cyclic arm replicates `jinns.data` exactly on the interior pool and applies the same scheme to the flattened boundary pool (`jinns` batches 2-D boundaries per face, which this does not model).

Configuration	Relative L^2	Share of gap to `jinns` closed
`underPINN` as reported	0.00242 ± 0.00032	—
+ `jinns` initializer	0.00182 ± 0.00033	$\sim 54\%$
+ `jinns` cyclic minibatching	0.00271 ± 0.00065	none
+ both	0.00188 ± 0.00026	$\sim 49\%$
`jinns`	0.0013 ± 0.0001	

The initializer accounts for roughly half of the gap, and on two of the three seeds rather than all of them. The cyclic minibatching accounts for none of it; the expectation that epoch-style sampling without replacement should converge faster is not borne out here. With both conventions applied, about half the gap remains, so at least one further difference we have not identified is at work — `jinns`' per-face boundary batching is one candidate we did not model. The initializer's effect is also problem-dependent: the same substitution on Burgers, run once earlier in this study, moved `underPINN`'s error from 0.219 ± 0.076 to 0.326, the opposite direction. We therefore do not attribute `jinns`' accuracy advantage to initialization in general. On Steady Heat it explains about half; on Burgers there is no distinguishable gap left to explain (Section 3.4).

A.11 Convergence and Accuracy, in Full

Section 3.5 summarizes; this appendix gives the accuracy-vs-budget figure and its full discussion.

The three-seed pass behind Figure 8 also caught a bug in the benchmark harness: `ODEHarmonicEvaluator.train()` had ignored its own `seed` argument, silently repeating one run three times. It is fixed, and the affected runs were repeated.

Spatial fields. To check that the predicted fields track the underlying physics rather than only satisfying boundary constraints, we compare predicted spatial fields against exact analytical solutions for problems where such solutions are available (Appendix A.2, Figure 3). The non-linear Burgers' equation is a common stress test for PINNs because of its steepening wavefront; the network resolves this sharp, shock-like transition without severe, non-physical oscillations, with absolute error concentrated in a thin band around the shock and staying below 0.21 over the full (x, t) domain. For the 1-D wave equation, predicted and exact solutions agree closely, including the characteristic criss-crossing propagation pattern; absolute error stays below 4×10^{-2} and is concentrated along the sharp characteristic fronts, where the network is somewhat less accurate than in smooth regions. For Steady Heat and Helmholtz, predicted fields agree closely with the exact solutions, with absolute error roughly two orders of magnitude smaller than the field values themselves (peak 1.1×10^{-2} and 2.3×10^{-2} respectively, against fields normalized to $[0, 1]$); Helmholtz is a further test of resolving high-frequency content, since standard MLPs are known to struggle with oscillatory fields due to spectral bias, and its largest errors are concentrated near the domain boundaries.

Figure 8 reports relative L^2 error at each problem's largest budget tested *in its own production configuration* from Table 4 (e.g. Burgers' MLP+FBPINN+RBA), rather than the plain-MLP configuration Tables 1–2 use for cross-framework matching – the two are not the same run, and where

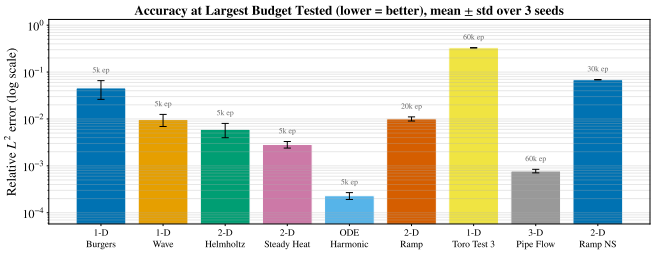


Figure 8: Relative L^2 error summary across benchmark problems.

they diverge it reflects that architecture or budget difference rather than a discrepancy. Burgers at 5,000 epochs is 0.219 ± 0.076 on the matched plain MLP (Tables 1–2) versus $4.6\% \pm 2.0\%$ here on the FBPINN+RBA production configuration; Helmholtz likewise appears here at its 5,000-epoch production budget, whereas Table 1's matched run was carried to 10,000 epochs and reaches a correspondingly lower $(5.9 \pm 0.2) \times 10^{-4}$. All nine are now three-seed-verified (mean $\pm$ std): the lower-cost problems at 5,000 epochs – ODE Harmonic $0.023\% \pm 0.004\%$ (the most accurate case in the suite), Steady Heat $0.29\% \pm 0.05\%$, Helmholtz $0.60\% \pm 0.20\%$, Wave $0.97\% \pm 0.28\%$, Burgers $4.6\% \pm 2.0\%$ – and the shock/3-D problems at their own wider budgets: Ramp $1.01\% \pm 0.10\%$ (20,000 ep), Pipe Flow $0.078\% \pm 0.007\%$ (60,000 ep), Ramp NS $6.87\% \pm 0.07\%$ (30,000 ep), and Toro Test 3 $32.9\% \pm 0.2\%$ (60,000 ep). Toro Test 3's under-resolved post-shock density spike, which dominates its density-weighted error measure (Table 5), leaves it the least accurate problem in the suite by a wide margin, a case discussed further in Appendix A.8 – this three-seed rerun revises that number down substantially from an earlier single-run estimate above 45%, the same single-run fragility Section 3.8 documents for the adaptive-collocation ablation, here caught in the paper's own headline accuracy claim rather than a comparison of arms. Excluding Toro Test 3, every problem reaches sub-percent-to-single-digit-percent error, with accuracy tracking discontinuity severity (Toro Test 3, Ramp NS) more than spatial dimensionality (3-D Pipe Flow is among the most accurate) or raw epoch count.

A.12 Training Loss Curves

Loss trajectories are largely monotonic across the tested systems: the combined loss, balancing the PDE residual against boundary and initial-condition terms, decreases over the allocated budgets. The visible transient spikes, on ODE Harmonic and Wave, occur in evaluators that never resample, so they are optimizer transients rather than resampling artifacts, and each recovers within a few tens of epochs; the two evaluators that do resample (Ramp, Ramp NS) show no comparable jumps. We did not observe severe gradient chattering or divergence in these runs, including for the stiffer, non-linear conservation-law problems (1-D Toro Test 3, 2-D Ramp NS), though we report loss curves from a single training run per problem rather than an average or spread over multiple seeds, so these observations should be read as descriptive of the runs shown rather than as a general stability guarantee.

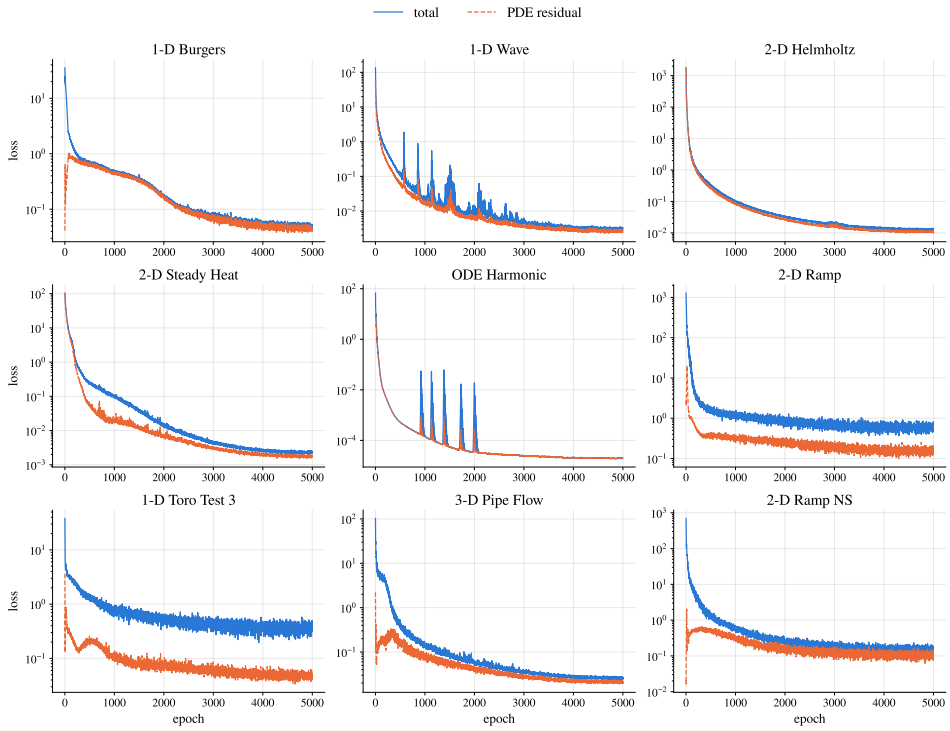


Figure 9: Training loss histories across all 9 benchmark problems, one training run each (first 5,000 epochs): the solid line is the total loss, the dashed line its PDE-residual term; the gap between them is the weighted boundary/initial-condition part.