Efficient PINNs and Natural Gradients in JAX

ML & AD in JAX for SC, Strasbourg

Nilo Schwencke

OCKHAM team (LIP – ENS Lyon • INRIA Lyon • CNRS)
Formerly (PhD): TAU team (LISN – Université Paris-Saclay • INRIA Saclay • CNRS)

June 11, 2026

Outline

1. Parametric models and Natural Gradient in a nutshell
2. Parametric models in JAX
3. Essential tools for PINNs implementation
4. Effective PINNs training implementation
5. Natural Gradient Implementation
6. Empirical motivation

Parametric models and Natural Gradient in a nutshell

Parametric model

$$u : \begin{cases} \mathbb{R}^P & \to & \mathcal{H} \\ \theta & \mapsto & u_\theta \end{cases} ; \mathcal{H} \text{ Hilbert space}$$

- $\mathcal{M} := \operatorname{Im} u = \{u_\theta \,:\, \theta \in \mathbb{R}^P\}$
- $T_\theta\mathcal{M} := \operatorname{Im} \mathrm{d}u_\theta = \operatorname{span}(\partial_p u_\theta)$

Quadratic regression

$$\mathcal{L}(u) := \frac{1}{2} \|u - f\|^2_{\mathrm{L}^2(\Omega)} ,$$

$$\mathrm{d}\mathcal{L}|_u(h) = \langle \underbrace{u - f}_{\nabla\mathcal{L}|_u}, h \rangle_{\mathrm{L}^2(\Omega)}.$$

Induces the gradient flow:

$$\begin{cases} u_0 \in \mathrm{L}^2(\Omega) \\ \dot{u}_t = -\nabla\mathcal{L}|_{u_t} = f - u_t \end{cases}$$

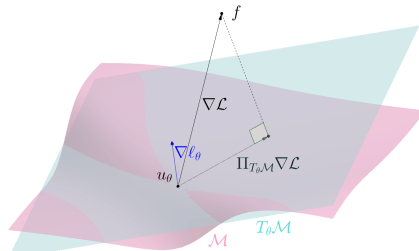
Solution: $u_t = f - e^{-t}(f - u_0)$.

Natural gradient in functional space

But u_θ lives on the manifold $\mathcal{M}$.
$\Rightarrow$ Project $\nabla\mathcal{L}$ onto $T_\theta\mathcal{M}$.

Then (Amari and Douglas1998):

$$\theta_{t+1} \leftarrow \theta_t - \eta \, \mathrm{d}u^\dagger_{\theta_t} \left(\Pi^\perp_{T_{\theta_t}\mathcal{M}} \nabla\mathcal{L}|_{u_{\theta_t}} \right),$$



$$\theta_{t+1} \leftarrow \theta_t - \eta \, G^\dagger_\theta \nabla_\theta \ell(\theta); \quad G_{\theta_{pq}} = \langle \partial_p u_\theta, \partial_q u_\theta \rangle_{\mathcal{H}} ;$$
$$\ell(\theta) := \mathcal{L}(u_\theta).$$

Natural Gradient of PINNs

Key remark

The only difference between the losses:

$$\mathcal{L}_{D,B}(u) = \int_\Omega \|D[u] - f\|^2 + \int_{\partial\Omega} \|B[u] - g\|^2$$

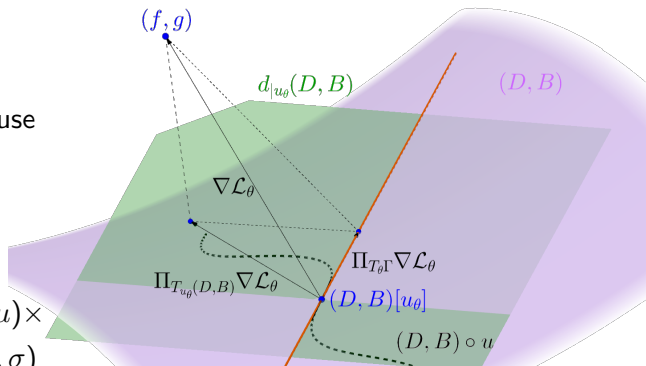
and $\mathcal{L}(u) = \frac{1}{2}\|u - f\|^2_{\mathrm{L}^2(\Omega)}$ is the use of the operators D and B.

Proposition

PINNs are a quadratic regression problem with model: $(D, B) \circ u$:

$$\begin{cases} \mathbb{R}^P & \to & \mathcal{H} & \to & \mathrm{L}^2(\Omega \to \mathbb{R}, \mu) \times \mathrm{L}^2(\partial\Omega \to \mathbb{R}, \sigma) \\ \theta & \mapsto & u_\theta & \mapsto & (D[u_\theta], B[u_\theta]) \end{cases}$$

Figure: Illustration of PINNs Natural Gradient



Parametric models in JAX

Implementing a parametric model in JAX

Parametric model

$$u : \begin{cases} \mathbb{R}^P & \to & \mathcal{H} \\ \theta & \mapsto & u_\theta \end{cases} ; \mathcal{H} \text{ Hilbert space}$$

Functional pov: matches JAX spirit!

Linear example

```
1  def linear_model_factory(function_basis):
2      def u(theta, x):
3          return jnp.sum([jnp.dot(a, f(x)) for
           ↪  a, f in zip(theta,
           ↪  function_basis)])
4      return u
```

Abstract implementation

From a JAX perspective, a parametric model is a function u with signature:

```
u: [PyTree, Array] -> Array #type Pmodel
```

where the first argument contains the trainable parameters and the second argument contains the evaluation points.

Typical usage for evaluation

```
1  # Given:
2  #  -theta (PyTree)
3  #  -samples (Array)
4  v_u = jax.vmap(u, (None, 0))
5  u_eval = v_u(theta, samples)
```

Differentiating a parametric model in JAX

Differential of a parametric model

$$\mathrm{d}u_\theta : \begin{cases} \mathbb{R}^P & \to & \mathcal{H} \\ h & \mapsto & \mathrm{d}u_\theta(h) \end{cases} ; \mathcal{H} \text{ Hilbert space}$$

How to compute $\mathrm{d}u_\theta$? **with AD !**

How to do it in practice?

Use `jax.linearize`:

```
# Given:
#  -u (Pmodel)
#  -theta (PyTree)
#  -samples (Array)
_, du_theta = jax.linearize(lambda p:
↪  jax.vmap(u, (None, 0))(p, samples), theta)
```

Given any PyTree `h` with the same structure as `theta`,

$$\texttt{du\_theta(h)} = \Big(\mathrm{d}u_\theta(h)(x)\Big)_{x \in \texttt{samples}}$$

Implementation remarks

Only efficient if we differentiate in a given direction h (not for the full Jacobian; see later)

Essential tools for PINNs implementation

Continuous problem

$$\begin{cases} \mathcal{D}[u] = f & \text{in } \Omega, \\ \mathcal{B}[u] = g & \text{on } \partial\Omega. \end{cases}$$

$$\mathcal{L}(u) = \frac{1}{2} \|\mathcal{D}[u] - f\|^2_{L^2(\Omega)} + \frac{\lambda}{2} \|\mathcal{B}[u] - g\|^2_{L^2(\partial\Omega)}$$

Discretization of the losses

Given a parametric model u:

$$\begin{aligned} \widehat{\mathcal{L}}(\theta) =& \frac{1}{2} \frac{1}{N_\Omega} \sum_i \Big(\mathcal{D}[u_\theta] - f\Big)(x_i)^2 \\ &+ \frac{\lambda}{2} \frac{1}{N_{\partial\Omega}} \sum_j \Big(\mathcal{B}[u_\theta] - g\Big)(z_j)^2. \end{aligned}$$

Implementation ingredients

- θ : `PyTree` of parameters;
- (x_i), (z_j): `Arrays` of samples;
- $\mathcal{D}, \mathcal{B}$ built by autodiff (see later).

We start with an abstract notion of Operator (that encompasses $\mathcal{D}, \mathcal{B}$)

Definition

```
Field = Array -> Array #type
op: Field -> Field #type Operator
```

How to insert a `Pmodel u` ?

```
def op_u_factory(u: Pmodel, op:
↪  Operator):
    def op_u(theta: PyTree, x: Array):
        return op(lambda z: u(theta,
        ↪  z))(x)
    return op_u
```

Key fact: `op_u` is also a `Pmodel`!

- Can be vectorized (`jax.vmap`)
- Can be auto-differentiated

What if `op_u` has parameters ?

```
p_op: [Field, PyTree] -> Field #type POperator
```

- Offline

```
# Given alpha (PyTree)
op: Operator = lambda v: p_op(v, alpha)
```

- Online

```
def p_op_up_factory(u: Pmodel, op: Operator):
    def p_op_u(theta: PyTree, x: Array, alpha:
    ↪  PyTree):
        return op(lambda z: u(theta, z),
        ↪  alpha)(x)
    return p_op_u
```

Computing differential operators in JAX

Partial derivatives by AD

Computation of $\partial_i v$, $v : \mathbb{R}^n \to \mathbb{R}^m$:

```
def dd(v: Field, i) -> Field:
  def dv_i(x):
    e_i = jnp.zeros_like(x).at[i].set(1.)
    _, dv_ix = jax.jvp(v, (x,), (e_i,))
    return dv_ix
  return dv_i
```

Higher orders by induction

Computation of $\partial_{ij} v$:

```
# Given v a Field
v_ij = dd(dd(v,j), i)
```

Example: heat operator

$$\mathcal{D}[u](t,x) = \partial_t u(t,x) - \kappa\, \partial_{xx} u(t,x).$$

```
def heat_operator(v: Field, kappa) -> Field:
  v_t = dd(v, 0)
  v_xx = dd(dd(v, 1), 1)

  def heat_v(tx):
      return v_t(tx) - kappa * v_xx(tx)
  return heat_v
```

Computational remarks

- Chaining `dd` may become expensive. `jax.experimental.jet` is an appealing alternative.
- We may also use more efficient techniques for some PDEs (e.g., the Laplacian, cf. practical session)

Effective PINNs training implementation

PINNs loss: $\widehat{\mathcal{L}}(\theta) = \frac{1}{2}\frac{1}{N_\Omega}\sum_i \Big(\mathcal{D}[u_\theta] - f\Big)(x_i)^2 + \frac{\lambda}{2}\frac{1}{N_{\partial\Omega}}\sum_j \Big(\mathcal{B}[u_\theta] - g\Big)(z_j)^2$

is a special case of an abstract loss: $\widehat{\mathcal{L}}_e(\theta) := \sum_{s=1}^{e}\sum_{i=1}^{N_s}\frac{w_s}{2N_s}\Big(\mathcal{O}_s[u_\theta](x_i^s) - f_k(x_i^s)\Big)^2$

Implementation

```
def loss_factory(u: Pmodel, ops: Iterable[Operators], fs: Iterable[Fields]):
  v_op_us = [jax.vmap(op_u_factory(u, op), (None, 0)) for op in ops]
  v_fs = [jax.vmap(f, (0,)) for f in fs]
  def loss(theta: PyTree, samples: Array, weights: Iterable[float]):
    cumulative_loss = 0.
    for v_op_u, v_f, sample, weight in zip(v_op_us, v_fs, samples, weights):
      op_u_eval = v_op_u(theta, sample)
      f_eval = v_f(sample)
      residual = op_u_eval - f_eval
      loss_op = jnp.mean(residual ** 2)
      cumulative_loss += weight * loss_op
    return cumulative_loss / 2
 return loss
```

Simply apply `jax.grad`

```
# Given:
# -loss (obtained with loss_factory)
# -theta (PyTree)
# -samples (Array)
# -weights (Iterable[float])
grad_loss = jax.grad(loss)
grad_loss_eval = grad_loss(theta, samples, weights) #PyTree
```

Last issue: update theta

Use `jax.flatten_util.ravel_pytree`
(should be imported separately!)

```
from jax.flatten_util import ravel_pytree
flat_theta, unravel_theta = ravel_pytree(theta)
```

- `flat_theta`: `Array` representing `theta` (arithmetic is applicable)

... and its gradient

- `unravel_theta: Array -> PyTree`, such that

```
unravel_theta(flat_theta) == theta #True
```

We can now train our PINN !

```
# minimal training loop
# Given:
# -etas (Iterable): learning rates
# -nsteps (int): number of training steps
for i in range(nsteps):
  eta = etas[i]
  grad_loss_eval = grad_loss(theta, samples, weights)
  flat_theta, unravel_theta = ravel_pytree(theta)
  flat_grad, _ = ravel_pytree(grad_loss_eval)
  theta = unravel_theta(theta - eta * flat_grad)
```

Natural Gradient Implementation

Computing the Jacobian for Natural Gradient

What we need?

Given a parametric model u,

$$J_\theta = \left(\partial_{\theta_p} u_\theta(x_i)\right)_{p,i}.$$

Then:

$$G_\theta = \frac{1}{N} J_\theta J_\theta^\top, \qquad \nabla_{\mathrm{NG}} \widehat{\mathcal{L}} = G_\theta^\dagger \nabla_\theta \widehat{\mathcal{L}}.$$

When using (abstract) operators

Compute for each operator $\mathcal{O}_s$:

$$J_\theta^s = \left(\partial_{\theta_p} \mathcal{O}_s[u_\theta](x_i)\right)_{p,i}; \quad G_\theta^s = \frac{1}{N_s} J_\theta^s J_\theta^{s\top}.$$

Then the full Gram reads:

$$G_\theta := \sum_s \lambda_s G_\theta^s.$$

Forward-mode implementation

```
1  def jac_linearize_factory(u):
2    def jac(theta: PyTree, samples: Array):
3      flat_theta, unravel =
       ↪  ravel_pytree(theta)
4
5      def batch_u(p):
6        return jax.vmap(u, (None,
         ↪  0))(unravel(p), samples)
7      _, tangent_fn = jax.linearize(batch_u,
       ↪  flat_theta)
8
9      # P forward passes
10     can_basis = jnp.eye(flat_theta.size)
11     return jax.vmap(tangent_fn)(can_basis)
12   return jac
```

Computing the Jacobian for Natural Gradient

Efficient reverse-mode version

```python
def jacrev_factory(u):
  def jac(theta: PyTree, samples: Array):
    flat_theta, unravel = ravel_pytree(theta)

    def u_flat(p, x):
      return model(unravel(p), x)

    jac_x = jax.jacrev(u_flat, argnums=0)
    J = jax.vmap(jac_x, (None, 0))(flat_theta,
    ↪  samples)

    return J.reshape((flat_theta.size, -1))
  return jac
```

Running time comparison

```
fwd mod: 67.1 ms ± 3.32 µs per loop
rev mod: 256  µs ± 7.15 µs per loop
```

Warning

Do not apply `jacrev` after `vmap`:

```python
jax.jacrev(lambda p: vmap_model(p,
↪  xs))(flat_p)
```

This materializes the whole batched Jacobian at once and may trigger:

```
RESOURCE_EXHAUSTED: Out of memory
```

Empirical motivation

1+1 D Heat equation

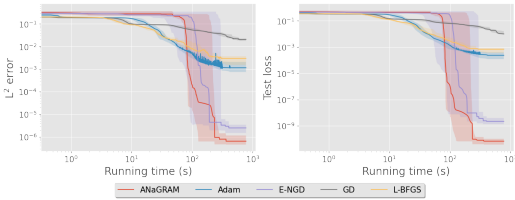


Figure: Performance comparison w.r.t running time for Heat equation in 1+1 D:

$$\begin{cases} \partial_t u - \frac{1}{4}\partial_{xx} u = 0 & \text{in } [0,1]^2 \\ u = 0 & \text{on } [0,1] \times \{0,1\} \\ u = \sin(\pi x) & \text{on } \{0\} \times [0,1] \end{cases}$$

1+1 D Allen-Cahn equation

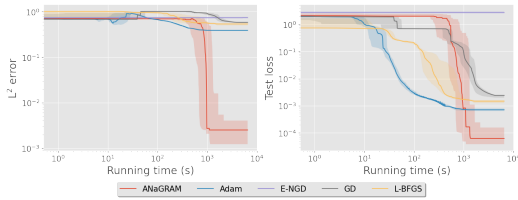


Figure: Performance comparison w.r.t running time for Allen-Cahn equation in 1+1 D:

$$\begin{cases} \partial_t u - 10^{-3}\,\partial_{xx} u = 5(u - u^3) & \text{in } \Omega = [0,1] \times [-1,1] \\ u = -1 & \text{on } \partial\Omega_b = [0,1] \times \{-1,1\} \\ u(0,x) = x^2 \cos(\pi x) & \text{on } \partial\Omega_0 = \{0\} \times [-1,1] \end{cases}$$

Note: E-NGD refers to Müller and Zeinhofer2023). ANaGRAM to Schwencke and Furtlehner2025).

Experiment	Train Loss		L_2 Error	
	AMStraMGRAM †	ANaGRAM	AMStraMGRAM †	ANaGRAM
Heat Equation	**6.29e-29 ± 6.78e-30**	8.56e-11 ± 7.05e-11	**2.32e-14 ± 1.14e-14**	1.28e-06 ± 1.75e-06
Laplace 2D	**1.46e-28 ± 1.87e-29**	4.27e-13 ± 4.66e-13	**2.24e-15 ± 2.52e-16**	3.49e-09 ± 3.58e-09
Laplace 5D	**2.04e-08 ± 1.16e-08**	6.37e-08 ± 7.01e-08	**2.12e-05 ± 8.15e-06**	4.00e-05 ± 2.93e-05
Allen–Cahn	**3.19e-11 ± 2.37e-11**	2.19e-04 ± 4.16e-04	**5.87e-05 ± 6.25e-06**	4.32e-03 ± 5.93e-03

Experiment	Train Loss		L_2 Error	
	AMStraMGRAM †	SSBroyden *	AMStraMGRAM †	SSBroyden *
Burgers (1+1 D)	**2.99e-12 ± 9.26e-13**	2.92e-10 ± 1.45e-10	**1.5e-06 ± 9.43e-7**	1.59e-06 ± 1.02e-6
Non-Linear Poisson	**8.51e-24 ± 2.24e-24**	3.03e-16 ± 3.82e-16	6.81e-10 ± 1.41e-09	**9.29e-12 ± 5.85e-12**
Allen–Cahn	3.19e-11 ± 2.37e-11	**6.42e-12 ± 5.52e-12**	5.87e-05 ± 6.25e-06	**3.94e-06 ± 1.72e-06**

* refers to the order two method of Urbán et al.2025), with adaptive sampling and hard constraint enforcement on boundary conditions.

† refers to Schwencke et al.2025)

Takeaways

1. Always think from a functional perspective (matches both the theory and effective implementation)
2. Defining abstract coding concepts (`Pmodel`, `Operator`, `Field`) helps the implementation.
3. AD is an efficient way of defining the differential operators.
4. Always vectorize!
5. Use Natural Gradient!

Not covered (see practical session):

- regularization
- linesearch

Thank you for your attention ! Questions welcome.

ScimBa

References I

Amari, S.-I. and S. C. Douglas (1998): "Why Natural Gradient?" in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, IEEE, vol. 2, 1213–1216.

Müller, J. and M. Zeinhofer (2023): "Achieving High Accuracy with PINNs via Energy Natural Gradient Descent," in *International Conference on Machine Learning*, PMLR, 25471–25485.

Schwencke, N. and C. Furtlehner (2025): "ANaGRAM: A Natural Gradient Relative to Adapted Model for Efficient PINNs Learning," in *The Thirteenth International Conference on Learning Representations*.

Schwencke, N., C. Rousselot, A. Shilova, and C. Furtlehner (2025): "AMStramGRAM: Adaptive Multi-Cutoff Strategy Modification for ANaGRAM," in *arXiv Preprint*, arXiv Preprint.

Urbán, J. F., P. Stefanou, and J. A. Pons (2025): "Unveiling the Optimization Process of Physics Informed Neural Networks: How Accurate and Competitive Can PINNs Be?" *Journal of Computational Physics*, 523, 113656.