Neural Network-Based Models for Physical Systems

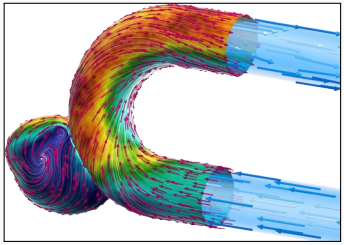
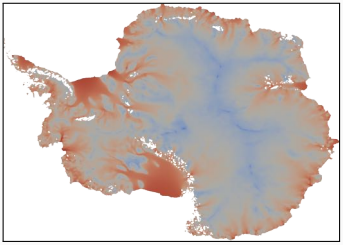
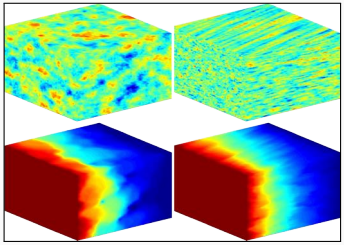
Analysis, Domain Decomposition, and Preconditioning

Alexander Heinlein[1]

International Conference on Latest Advances in Computational and Applied Mathematics 2025 (LACAM-25), IISER Thiruvananthapuram, Kerala, India, December 8-11, 2025

[1] Delft University of Technology

Scientific Computing and Machine Learning

Numerical methods

Based on physical models

+ Robust and generalizable
− Require availability of mathematical models

Machine learning models

Driven by data

+ Do not require mathematical models
− Sensitive to data, limited extrapolation capabilities

Scientific machine learning

Combining the strengths and **compensating the weaknesses** of the individual approaches:

numerical methods	**improve**	machine learning techniques
machine learning techniques	**assist**	numerical methods

Outline

1. Physics-informed neural networks – Adaptive sampling and localization via domain decompostion

   Based on joint work with

Damien Beecroft	(University of Washington)
Victorita Dolean	(Eindhoven University of Technology)
Bianca Giovanardi, **Coen Visser**	(Delft University of Technology)
Amanda A. Howard and **Panos Stinis**	(Pacific Northwest National Laboratory)
Siddhartha Mishra	(ETH Zürich)
Ben Moseley	(Imperial College London)

2. Deep operator networks – Error analysis and localization via domain decompostion

   Based on joint work with

Damien Beecroft	(University of Washington)
Amanda A. Howard and **Panos Stinis**	(Pacific Northwest National Laboratory)
Johannes Taraz	(Delft University of Technology)

Physics-informed neural networks – Adaptive sampling and localization via domain decompostion

Physics-Informed Neural Networks (PINNs)

In the **physics-informed neural network (PINN)** approach introduced by **Raissi et al. (2019)**, a **neural network** is employed to **discretize a partial differential equation**

$$\mathcal{N}[u] = f, \quad \text{in } \Omega.$$

PINNs use a **hybrid loss function**:

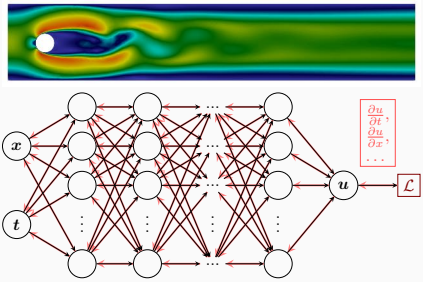
$$\mathcal{L}(\boldsymbol{\theta}) = \omega_{\text{data}}\mathcal{L}_{\text{data}}(\boldsymbol{\theta}) + \omega_{\text{PDE}}\mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}),$$

where ω_{data} and ω_{PDE} are **weights** and

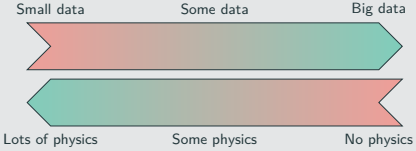
$$\mathcal{L}_{\text{data}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{data}}} \sum_{i=1}^{N_{\text{data}}} \left(u(\hat{\boldsymbol{x}}_i, \boldsymbol{\theta}) - u_i\right)^2,$$

$$\mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{PDE}}} \sum_{i=1}^{N_{\text{PDE}}} \left(\mathcal{N}[u](\boldsymbol{x}_i, \boldsymbol{\theta}) - f(\boldsymbol{x}_i)\right)^2.$$

See also **Dissanayake and Phan-Thien (1994); Lagaris et al. (1998)**.



Hybrid loss



- **Known solution values** can be included in $\mathcal{L}_{\text{data}}$
- **Initial and boundary conditions** are also included in $\mathcal{L}_{\text{data}}$

Advantages

- "Meshfree"
- Small data
- Generalization properties
- High-dimensional problems
- Inverse and parameterized problems

Drawbacks

- Training cost and robustness
- Convergence not well-understood
- Difficulties with scalability and multi-scale problems

Main Drivers of the Total Error of PINNs

Estimate of the total error (Mishra and Molinaro (2022))

The total error (or generalization error) satisfies

$$\mathcal{E}_G \leq C_{\text{PDE}} \mathcal{E}_T + C_{\text{PDE}} C_{\text{quad}}^{1/p} N^{-\alpha/p}$$

- $\mathcal{E}_G = \mathcal{E}_G(\boldsymbol{X}, \boldsymbol{\theta}) := \|\mathbf{u} - \mathbf{u}^*\|_V$ **general. error** (V Sobolev space, $\boldsymbol{X}$ training data set)
- $\mathcal{E}_T$ **training error** (l^p **loss** of the residual of the PDE)
- N **number of the training points** and α **convergence rate of the quadrature**
- C_{PDE} and C_{quad} **constants** depending on the **PDE**, **quadrature**, and **neural network**

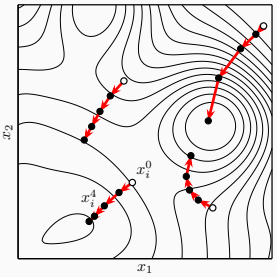
- **Training error**: optimization is **challenging** due to the **highly nonconvex** loss landscape of PINNs.
- **Sampling error**: unlike classical numerical methods, standard quadrature rules **poorly approximate** neural-network trial functions.

PACMANN – Point Adaptive Collocation Method for Artificial Neural Networks

In **Visser, Heinlein, and Giovanardi (subm. 2025; arXiv:2411.19632)**, the collocation points are updated by solving the **min-max problem**

$$\min_{\boldsymbol{\theta}} \left[\omega_{\text{data}} \mathcal{L}_{\text{data}}(\boldsymbol{\theta}) + \max_{\boldsymbol{X} \subset \Omega} \omega_{\text{PDE}} \mathcal{L}_{\text{PDE}}(\boldsymbol{X}, \boldsymbol{\theta}) \right].$$

This idea was already mentioned in **Wang et al. (2022)**. Different from other residual-based adaptive sampling methods, **existing collocation points are moved** using gradient-based optimizers such as **gradient ascent**, **RMSprop** (**Hinton (2018)**), or **Adam** (**Kingma, Ba (2017)**).



Algorithm 1: **PACMANN** with iteration counts P and T and stepsize s

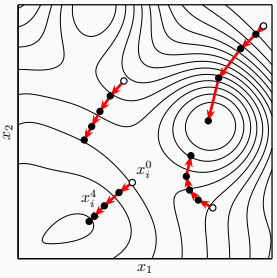
Sample a set $\boldsymbol{X}$ of N_{PDE} collocation points using a uniform sampling method;
while *stopping criterion not reached* **do**
  Train the PINN for P iterations;
  for $k = 1, \ldots, T$ **do**
    Compute squared residual $\mathcal{R}(\boldsymbol{x}_i) = (\mathcal{N}[u](\boldsymbol{x}_i, \boldsymbol{\theta}) - f(\boldsymbol{x}_i))^2$ for all $\boldsymbol{x}_i \in \boldsymbol{X}$;
    Compute gradient $\nabla_{\boldsymbol{x}} \mathcal{R}(\boldsymbol{x}_i)$ for all $\boldsymbol{x}_i \in \boldsymbol{X}$;
    Move the points in $\boldsymbol{X}$ according to the chosen optimization algorithm and stepsize s;
  end
  Resample points in $\boldsymbol{X}$ that moved outside Ω based on a uniform probability distribution;
end

PACMANN – Point Adaptive Collocation Method for Artificial Neural Networks

In **Visser, Heinlein, and Giovanardi (subm. 2025; arXiv:2411.19632)**, the collocation points are updated by solving the **min-max problem**

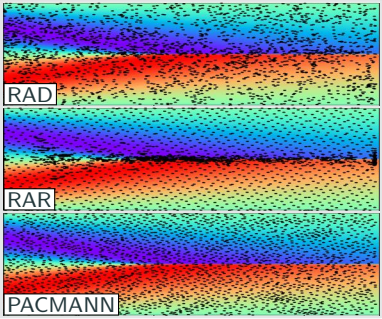
$$\min_{\boldsymbol{\theta}} \left[\omega_{\text{data}} \mathcal{L}_{\text{data}}(\boldsymbol{\theta}) + \max_{\boldsymbol{X} \subset \Omega} \omega_{\text{PDE}} \mathcal{L}_{\text{PDE}}(\boldsymbol{X}, \boldsymbol{\theta}) \right].$$

This idea was already mentioned in **Wang et al. (2022)**. Different from other residual-based adaptive sampling methods, **existing collocation points are moved** using gradient-based optimizers such as **gradient ascent**, **RMSprop** (**Hinton (2018)**), or **Adam** (**Kingma, Ba (2017)**).



Burgers' equation – Comparison against different methods

sampling method	L_2 relative error		mean
2500 coll. points	mean	1 SD	runtime [s]
uniform grid	25.9%	14.2%	425
Hammersley grid	0.61%	0.53%	443
random resampling	0.40%	0.35%	**423**
RAR	0.11%	**0.05%**	450
RAD	0.16%	0.10%	463
RAR-D	0.24%	0.21%	503
PACMANN–Adam	**0.07%**	**0.05%**	461

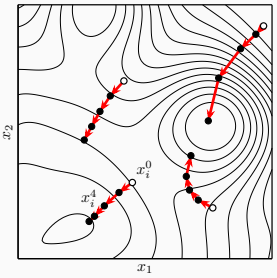


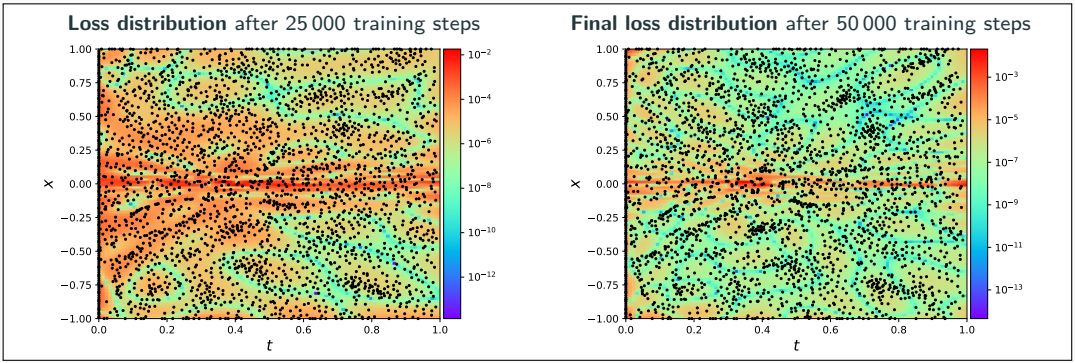
PACMANN – Point Adaptive Collocation Method for Artificial Neural Networks

In **Visser, Heinlein, and Giovanardi (subm. 2025; arXiv:2411.19632)**, the collocation points are updated by solving the **min-max problem**

$$\min_{\boldsymbol{\theta}} \left[\omega_{\text{data}} \mathcal{L}_{\text{data}}(\boldsymbol{\theta}) + \max_{\boldsymbol{X} \subset \Omega} \omega_{\text{PDE}} \mathcal{L}_{\text{PDE}}(\boldsymbol{X}, \boldsymbol{\theta}) \right].$$

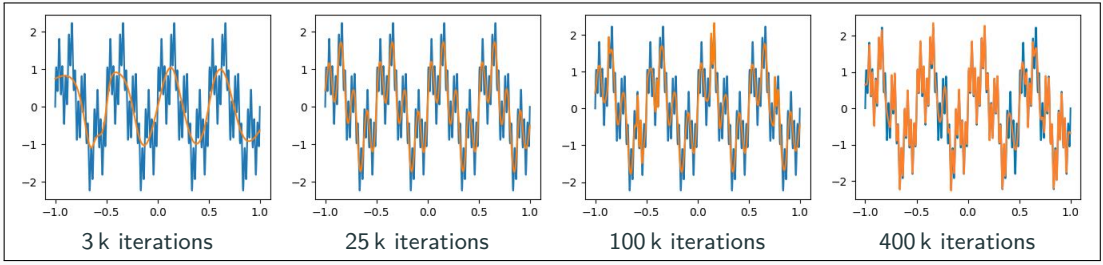
This idea was already mentioned in **Wang et al. (2022)**. Different from other residual-based adaptive sampling methods, **existing collocation points are moved** using gradient-based optimizers such as **gradient ascent**, **RMSprop** (**Hinton (2018)**), or **Adam** (**Kingma, Ba (2017)**).





Spectral Bias of Neural Networks

Rahaman et al. (2019) observed the **spectral bias of neural networks**: during training, they **learn low-frequency functions faster than high-frequency functions**; see also **Cao et al. (2021), Wang, et al. (2022), Hong et al. (arXiv 2022), Xu et al. (2024), ...**



This can be understood by interpreting the traning dynamics of neural networks as **gradient flow**

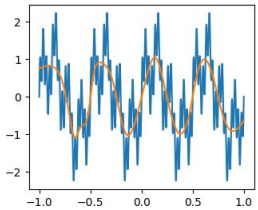
$$\theta_{k+1} = \theta_k - \eta_k \nabla_\theta \mathcal{C}(\eta_{\theta_k}) \quad \rightarrow \quad \frac{\mathrm{d}x}{\mathrm{d}t}(t) = -\nabla_x \mathcal{C}(x(t))$$

For **infinitely wide neural networks**, we obtain a **constant neural tangent kernel (NTK)** (**Jacot et al. (2018)**).
For mean squared error (MSE) loss $\mathcal{C}$, we obtain the **discretized** $\boldsymbol{K} = \left(\left\langle \frac{d\eta_{\theta(t)}}{d\theta}(x_i), \frac{d\eta_{\theta(t)}}{d\theta}(x_j) \right\rangle \right)_{ij}$:

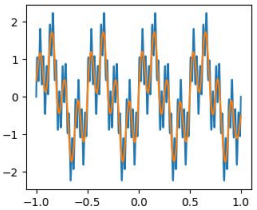
$$\frac{d\eta_{\theta(t)}}{dt} = -\frac{2}{n} \cdot \boldsymbol{K}(t) \left(\eta_{\theta(t)}(\mathbf{X}) - \mathbf{Y} \right) \quad \Rightarrow \quad \mathbf{U}^\top \left(\eta_{\theta(t)}(\mathbf{X}) - \mathbf{Y} \right) \approx e^{-t\frac{2}{n}\boldsymbol{\Lambda}} \mathbf{U}^\top \mathbf{Y}$$

Spectral Bias of Neural Networks

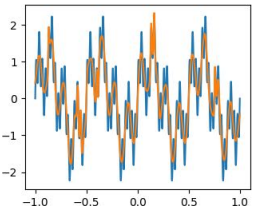
Rahaman et al. (2019) observed the **spectral bias of neural networks**: during training, they **learn low-frequency functions faster than high-frequency functions**; see also **Cao et al. (2021), Wang, et al. (2022), Hong et al. (arXiv 2022), Xu et al. (2024), ...**



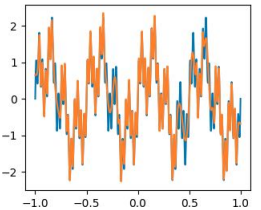
3 k iterations



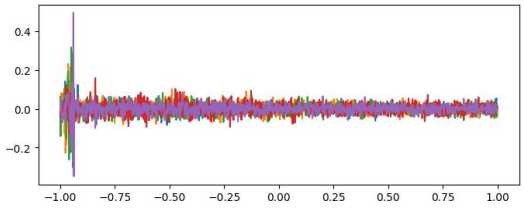
25 k iterations



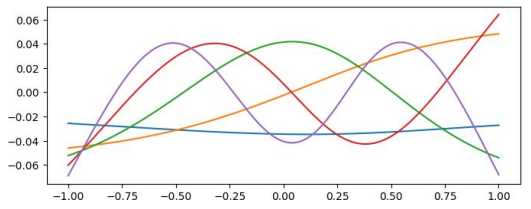
100 k iterations



400 k iterations



smallest eigenvalues (**slow training**)



largest eigenvalues (**fast training**)

Scaling of PINNs for a Simple ODE Problem

Solve

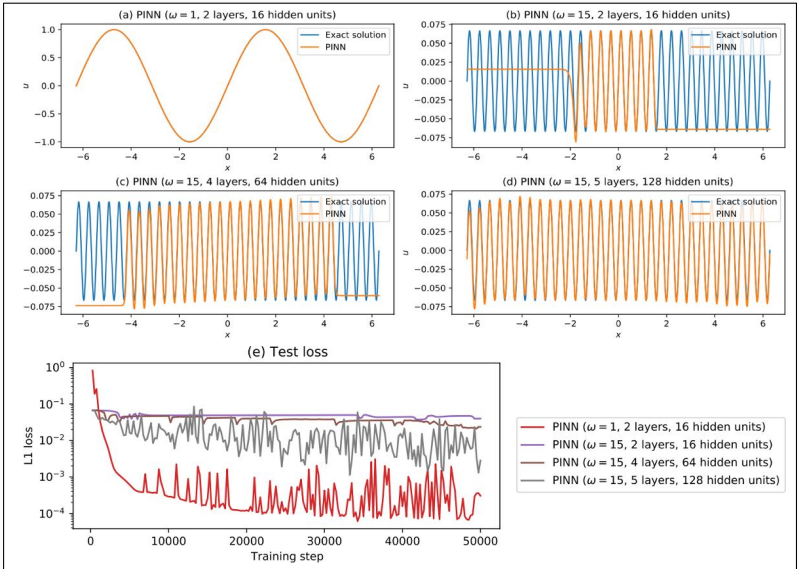
$$\begin{aligned} u' &= \cos(\omega x), \\ u(0) &= 0, \end{aligned}$$

for different values of ω using **PINNs with varying network capacities**.

Scaling issues

- Large computational domains
- Small frequencies

Cf. **Moseley, Markham, and Nissen-Meyer (2023)**



(a) 321 free parameters (d) 66 433 free parameters

Scaling of PINNs for a Simple ODE Problem

Solve

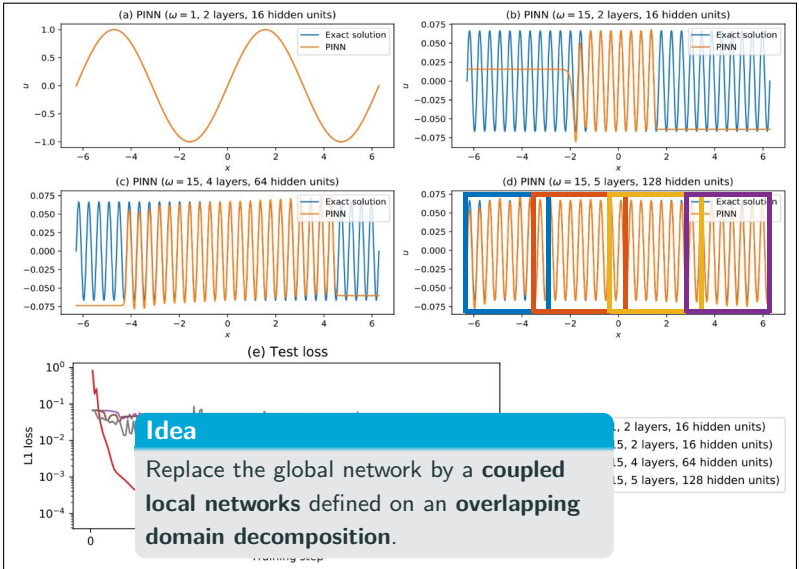
$$\begin{aligned} u' &= \cos(\omega x), \\ u(0) &= 0, \end{aligned}$$

for different values of ω using **PINNs with varying network capacities**.

Scaling issues

- Large computational domains
- Small frequencies

Cf. **Moseley, Markham, and Nissen-Meyer (2023)**



Idea

Replace the global network by a **coupled local networks** defined on an **overlapping domain decomposition**.

(a) 321 free parameters (d) 66 433 free parameters

Domain Decomposition Methods and Machine Learning – Literature

A **non-exhaustive literature overview**:

- **ML for adaptive BDDC, FETI–DP, AGDSW**: H., Klawonn, Lanser, Weber (2019, 2020, 2021, 2021, 2021, 2022); Klawonn, Lanser, Weber (2024, 2025)
- **cPINNs, XPINNs**: Jagtap, Kharazmi, Karniadakis (2020); Jagtap, Karniadakis (2020)
- **Classical Schwarz iteration for PINNs or DeepRitz:**: Li, Tang, Wu, and Liao (2019); Li, Xiang, Xu (2020); Mercier, Gratton, Boudier (arXiv 2021); Dolean, H., Mercier, Gratton (acc. 2025); Li, Wang, Cui, Xiang, Xu (2023); Sun, Xu, Yi (arXiv 2023, 2024); Kim, Yang (2023, 2024, 2024)
- **FBPINNs, FBKANs**: Moseley, Markham, Nissen-Meyer (2023); Dolean, H., Mishra, Moseley (2024, 2024); H., Howard, Beecroft, Stinis (2025); Howard, Jacob, Murphy, H., Stinis (arXiv 2024)
- **DD for randomized NNs**: Dong, Li (2021); Dang, Wang (2024); Sun, Dong, Wang (2024); Sun, Wang (2024); Chen, Chi, E, Yang (2022); Shang, H., Mishra, Wang (2025); Anderson, Dolean, Moseley, Pestana, (arXiv 2024); van Beek, Dolean, Moseley (arxiv 2025)
- **DD for Neural Operators and Surrogate Models**: H., Howard, Beecroft, Stinis (2025); Ramezankhani, Parekh, Deodhar, Birru (arXiv 2024); Wu, Kovachki, Liu (arXiv 2025); Pelzer, Verburg, H., Schulte (arXiv 2025); Klaes, Klawonn, Kubicki, Lanser, Nakajima, Shimokawabe, Weber (arXiv 2025); Howard, H., Stinis (in prep.)
- **DD for CNNs**: Gu, Zhang, Liu, Cai (2022); Lee, Park, Lee (2022); Klawonn, Lanser, Weber (2024); Verburg, H., Cyr (2025)

An overview of the state-of-the-art in 2024:

A. Klawonn, M. Lanser, J. Weber
Machine learning, domain decomposition methods – a survey
Computational Science and Engineering. 2024

Finite Basis Physics-Informed Neural Networks (FBPINNs)

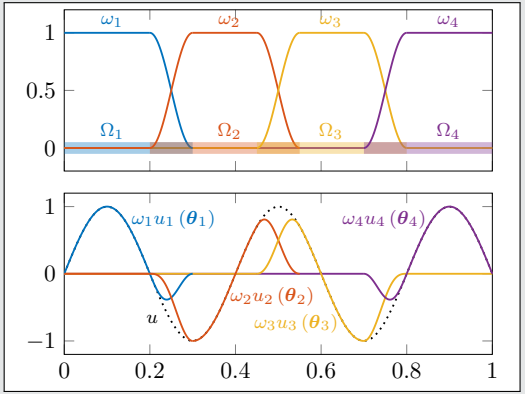
FBPINNs (Moseley, Markham, Nissen-Meyer (2023))

FBPINNs employ the **network architecture**

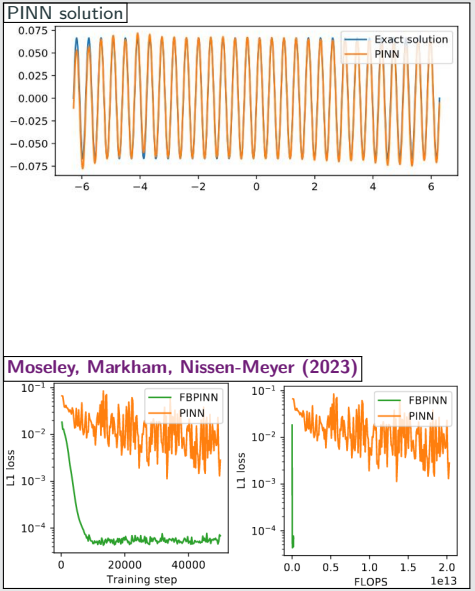
$$u(\theta_1, \dots, \theta_J) = \sum_{j=1}^{J} \omega_j u_j(\theta_j)$$

and the **loss function**

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left(\mathcal{N}[\sum_{\boldsymbol{x}_i \in \Omega_j} \omega_j u_j](\boldsymbol{x}_i, \theta_j) - f(\boldsymbol{x}_i) \right)^2.$$



1D single-frequency problem



Finite Basis Physics-Informed Neural Networks (FBPINNs)

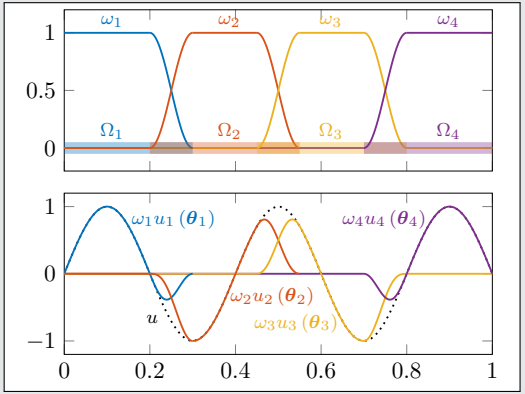
FBPINNs (Moseley, Markham, Nissen-Meyer (2023))

FBPINNs employ the **network architecture**

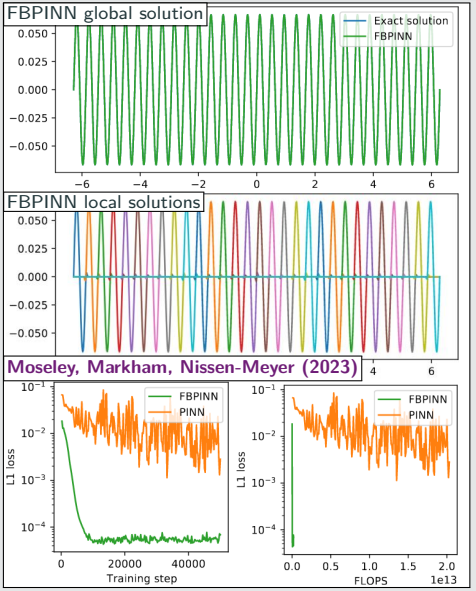
$$u(\theta_1, \ldots, \theta_J) = \sum_{j=1}^{J} \omega_j u_j(\theta_j)$$

and the **loss function**

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left(\mathcal{N}[\sum_{\boldsymbol{x}_i \in \Omega_j} \omega_j u_j](\boldsymbol{x}_i, \theta_j) - f(\boldsymbol{x}_i) \right)^2.$$



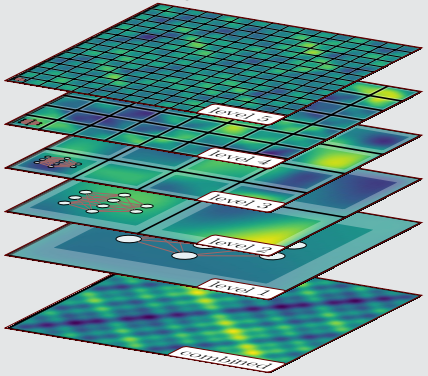
1D single-frequency problem



Multi-Level FBPINNs

Multi-level FBPINNs (ML-FBPINNs)

ML-FBPINNs (**Dolean, Heinlein, Mishra, Moseley (2024)**) are based on a **hierarchy of domain decompositions:**

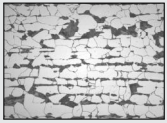


This yields the **network architecture**:

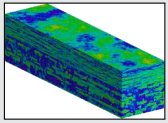
$$u(\theta_1^{(1)}, \dots, \theta_{J^{(L)}}^{(L)}) = \sum_{l=1}^{L} \sum_{i=1}^{N^{(l)}} \omega_j^{(l)} u_j^{(l)}(\theta_j^{(l)})$$

Multiscale problems . . .

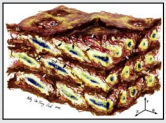
. . . appear in most areas of modern science and engineering:



Dual-phase steel; fig. courtesy of **J. Schröder**.



Groundwater flow; cf. **Christie & Blunt (2001)** (SPE10).



Arterial walls; cf. **O'Connell et al. (2008)**.

Multi-frequency problem

Consider the **multi-frequency Laplace problem**

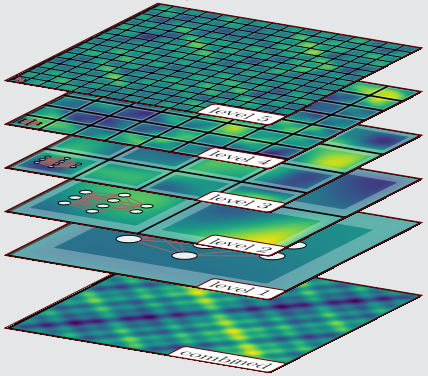
$$-\Delta u = 2 \sum_{i=1}^{n} (\omega_i \pi)^2 \sin(\omega_i \pi x) \sin(\omega_i \pi y),$$

with homogeneous Dirichlet boundary conditions and $\omega_i = 2^i$.

For increasing values of n, we obtain the **solutions**:

Multi-level FBPINNs (ML-FBPINNs)

ML-FBPINNs (**Dolean, Heinlein, Mishra, Moseley (2024)**) are based on a **hierarchy of domain decompositions:**

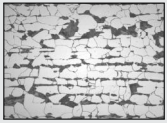


This yields the **network architecture**:

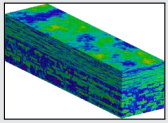
$$u(\theta_1^{(1)}, \dots, \theta_{J^{(L)}}^{(L)}) = \sum_{l=1}^{L} \sum_{i=1}^{N^{(l)}} \omega_j^{(l)} u_j^{(l)}(\theta_j^{(l)})$$

Multiscale problems . . .

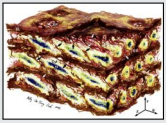
. . . appear in most areas of modern science and engineering:



Dual-phase steel; fig. courtesy of **J. Schröder**.



Groundwater flow; cf. **Christie & Blunt (2001)** (SPE10).



Arterial walls; cf. **O'Connell et al. (2008)**.

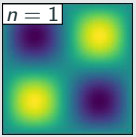
Multi-frequency problem

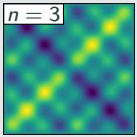
Consider the **multi-frequency Laplace problem**

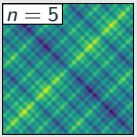
$$-\Delta u = 2 \sum_{i=1}^{n} (\omega_i \pi)^2 \sin(\omega_i \pi x) \sin(\omega_i \pi y),$$

with homogeneous Dirichlet boundary conditions and $\omega_i = 2^i$.

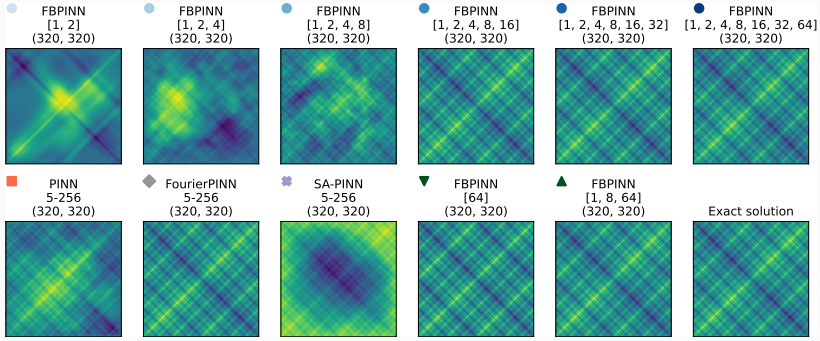
For increasing values of n, we obtain the **solutions**:

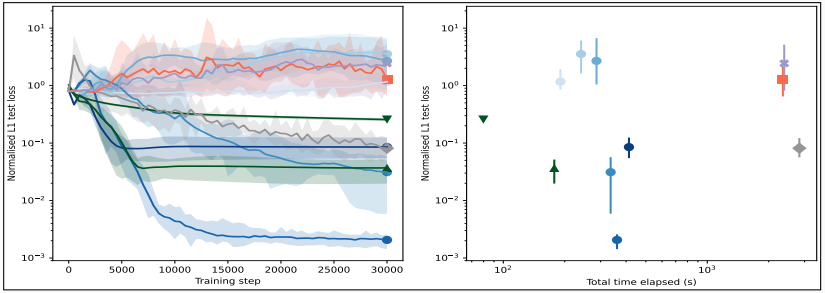


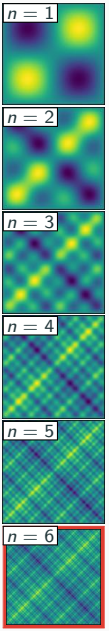




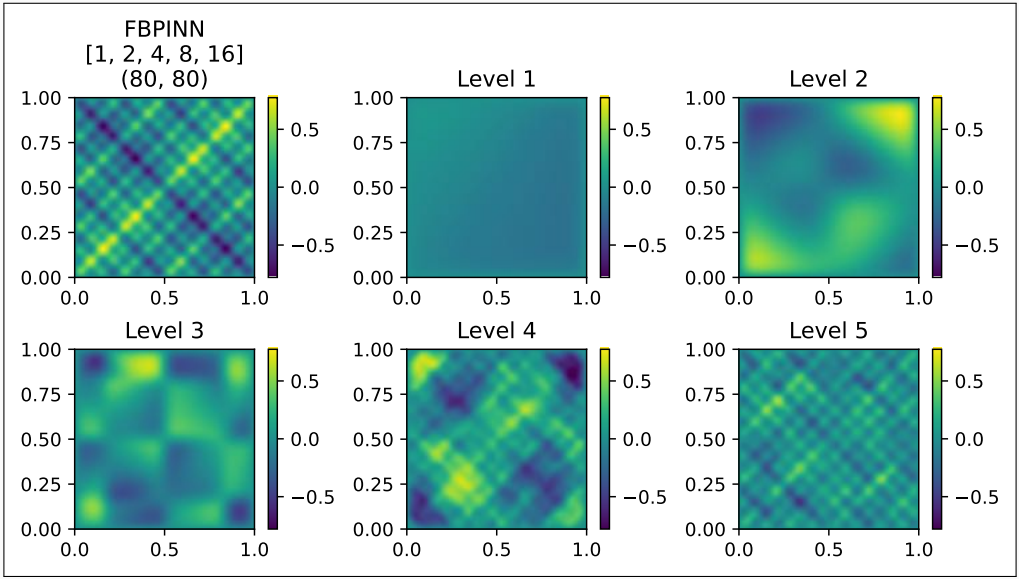
Multi-Level FBPINNs for a Multi-Frequency Problem – Strong Scaling







Multi-Frequency Problem – What the FBPINN Learns

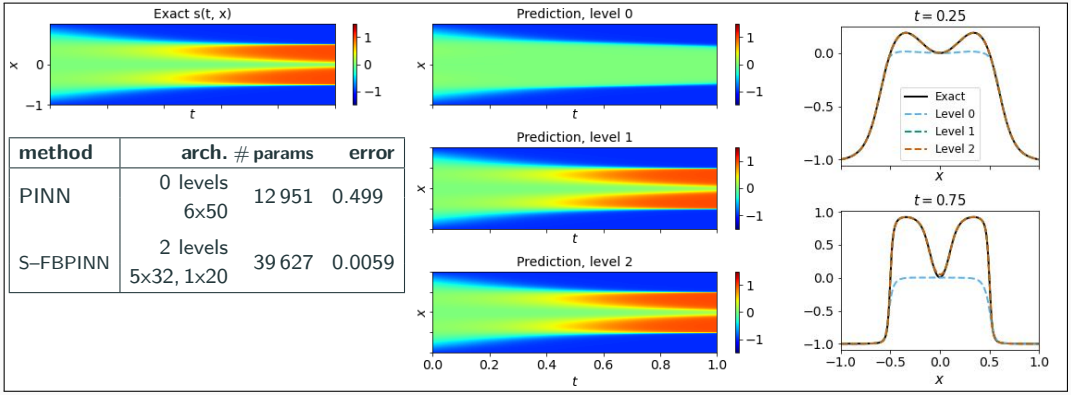


Cf. **Dolean, Heinlein, Mishra, Moseley (2024)**.

Multifidelity Stacking FBPINNs – Allen–Cahn Equation

Finally, we consider the **Allen–Cahn equation**, describing phase separation in multi-component alloy systems:

$$
\begin{aligned}
s_t - 0.0001 s_{xx} + 5s^3 - 5s = 0, \quad & t \in (0,1], x \in [-1,1], \\
s(x,0) = x^2 \cos(\pi x), \quad & x \in [-1,1], \\
s(x,t) = s(-x,t), \quad & t \in [0,1], x = -1, x = 1, \\
s_x(x,t) = s_x(-x,t), \quad & t \in [0,1], x = -1, x = 1.
\end{aligned}
$$



method	arch.	# params	error
PINN	0 levels 6x50	12 951	0.499
S–FBPINN	2 levels 5x32, 1x20	39 627	0.0059

PINN **gets stuck** at **fixed point of the dynamical system**; cf. **Rohrhofer et al. (2023)**.

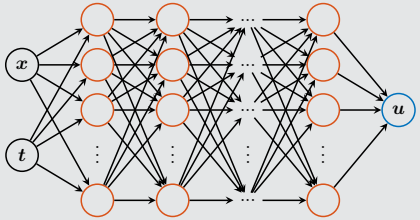
Domain Decomposition for Physics-Informed Randomized Neural Networks

Neural networks

A standard **multilayer perceptron (MLP)** with L hidden layers is a **parametric** model of the form

$$u(\boldsymbol{x}, \boldsymbol{\theta}) = F_{L+1}^{\boldsymbol{A}} \cdot F_{L}^{\boldsymbol{W}_L, \boldsymbol{b}_L} \circ \ldots \circ F_{1}^{\boldsymbol{W}_1, \boldsymbol{b}_1}(\boldsymbol{x}),$$

where $\boldsymbol{A}$ is linear, and the ith hidden layer is nonlinear $F_i^{\boldsymbol{W}_i, \boldsymbol{b}_i}(\boldsymbol{x}) = \sigma(\boldsymbol{W}_i \cdot \boldsymbol{x} + \boldsymbol{b}_i)$.



In order to optimize the loss function

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}),$$

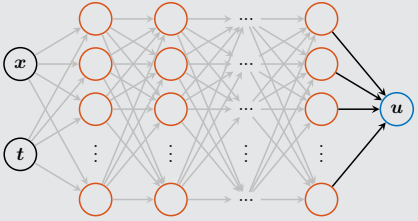
all parameters $\boldsymbol{\theta} = (\boldsymbol{A}, \boldsymbol{W}_1, \boldsymbol{b}_1, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_L)$ are **trained**.

Randomized neural networks

In **randomized neural networks (RaNNs)** as introduced by **Pao and Takefuji (1992)**,

$$u(\boldsymbol{x}, \boldsymbol{A}) = F_{L+1}^{\boldsymbol{A}} \cdot F_{L}^{\boldsymbol{W}_L, \boldsymbol{b}_L} \circ \ldots \circ F_{1}^{\boldsymbol{W}_1, \boldsymbol{b}_1}(\boldsymbol{x}),$$

the weights in the hidden layers are randomly initialized and **fixed**; only $\boldsymbol{A}$ is trainable.



The model is linear with respect to the trainable parameters $\boldsymbol{A}$, and the optimization problem reads

$$\min_{\boldsymbol{A}} \mathcal{L}(\boldsymbol{A}).$$

This can **simplify the training process**.

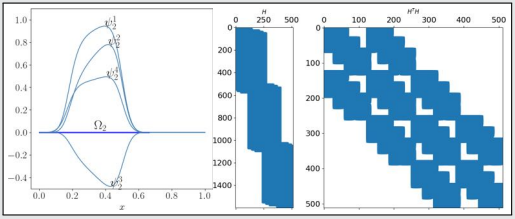
Domain Decomposition for Physics-Informed Randomized Neural Networks

Domain decomposition for RaNNs

We employ the FBPINNs approach; cf. **Shang, Heinlein, Mishra, Wang (2025)**. This is closely related to the **random feature method (RFM)** by **Chen, Chi, E, Yang (2022)**. In particular, we solve

$$\mathcal{A}[\sum_{j=1}^{J} \omega_j u_j(\boldsymbol{A}_j)](\boldsymbol{x}_i) = f(\boldsymbol{x}_i),$$

for $i = 1, \dots, N_{\text{PDE}}$; the boundary condtions are incorporated directly into the u_j.



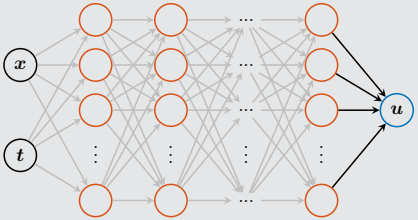
The hidden weights are randomly initialized, the resulting matrices $\boldsymbol{H}$ and $\boldsymbol{H}^\top \boldsymbol{H}$ are block-sparse.

Randomized neural networks

In **randomized neural networks (RaNNs)** as introduced by **Pao and Takefuji (1992)**,

$$u(\boldsymbol{x}, \boldsymbol{A}) = F_{L+1}^{\boldsymbol{A}} \cdot F_L^{W_L, b_L} \circ \dots \circ F_1^{W_1, b_1}(\boldsymbol{x}),$$

the weights in the hidden layers are randomly initialized and **fixed**; only $\boldsymbol{A}$ is trainable.



The model is linear with respect to the trainable parameters $\boldsymbol{A}$, and the optimization problem reads

$$\min_{\boldsymbol{A}} \mathcal{L}(\boldsymbol{A}).$$

This can **simplify the training process**.

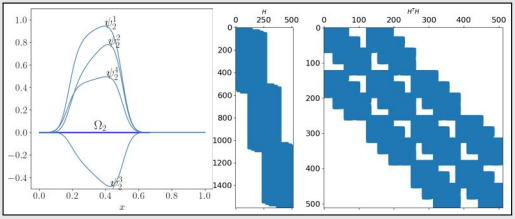
Domain Decomposition for Physics-Informed Randomized Neural Networks

Domain decomposition for RaNNs

We employ the FBPINNs approach; cf. **Shang, Heinlein, Mishra, Wang (2025)**. This is closely related to the **random feature method (RFM)** by **Chen, Chi, E, Yang (2022)**. In particular, we solve

$$\mathcal{A}[\sum_{j=1}^{J} \omega_j u_j(\boldsymbol{A}_j)](\boldsymbol{x}_i) = f(\boldsymbol{x}_i),$$

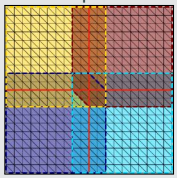
for $i = 1, \dots, N_{\text{PDE}}$; the boundary condtions are incorporated directly into the u_j.



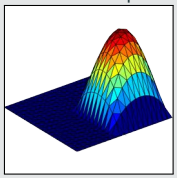
The hidden weights are randomly initialized, the resulting matrices $\boldsymbol{H}$ and $\boldsymbol{H}^\top \boldsymbol{H}$ are block-sparse.

One-level Schwarz preconditioner

Overlap $\delta = 2h$



Solution of local problem



$$\boldsymbol{M}_{\text{OS-1}}^{-1}\boldsymbol{K} = \sum_{i=1}^{N} \boldsymbol{R}_i^\top \boldsymbol{K}_i^{-1} \boldsymbol{R}_i \boldsymbol{K},$$

where $\boldsymbol{R}_i$ restriction operator to Ω_i', $\boldsymbol{K}_i := \boldsymbol{R}_i \boldsymbol{K} \boldsymbol{R}_i^\top$.

Singular Value Decomposition (SVD)

On each subdomain Ω_j, the RaNN reads

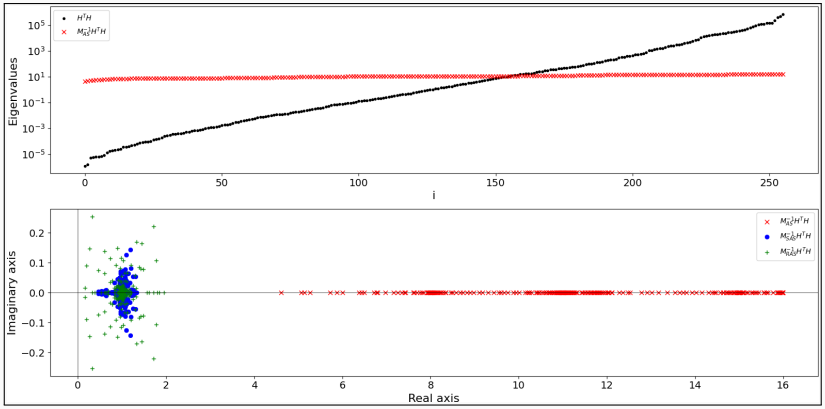
$$u_j(\boldsymbol{x}, \boldsymbol{A}_j) = \boldsymbol{A}_j \begin{bmatrix} \Phi_1(\boldsymbol{x}) & \cdots & \Phi_k(\boldsymbol{x}) \end{bmatrix}^\top,$$

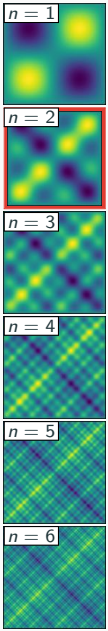
where $\Phi_1, \dots, \Phi_k$ are the randomized basis functions in the **last hidden layer**. Replace it by

$$\hat{u}_j(\boldsymbol{x}, \boldsymbol{A}_j) = \boldsymbol{A}_j \hat{\boldsymbol{V}}^\top \begin{bmatrix} \Phi_1(\boldsymbol{x}) & \cdots & \Phi_k(\boldsymbol{x}) \end{bmatrix}^\top,$$

where $\hat{\boldsymbol{V}}^\top$ **truncated right singular vectors** of Φ.

Results for the Multi-Frequency Problem (n=2)

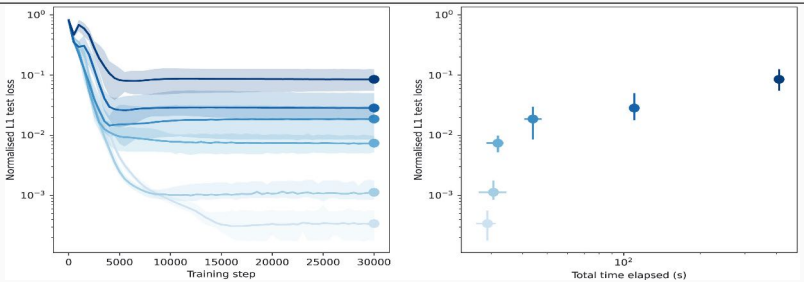




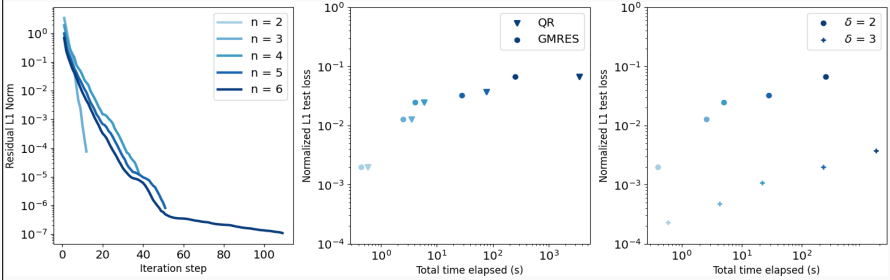
	$M^{-1} = I$		$M^{-1} = M_{AS}^{-1}$		$M^{-1} = M_{RAS}^{-1}$		$M^{-1} = M_{SAS}^{-1}$	
	iter	e_{L^2}	iter	e_{L^2}	iter	e_{L^2}	iter	e_{L^2}
CG	> 2000	$1.95{\cdot}10^{-2}$	8	$5.03{\cdot}10^{-3}$	—	—	—	—
CGS	> 2000	$2.63{\cdot}10^{-2}$	4	$5.04{\cdot}10^{-3}$	24	$5.03{\cdot}10^{-3}$	6	$5.04{\cdot}10^{-3}$
BICG	> 2000	$1.03{\cdot}10^{-2}$	8	$5.08{\cdot}10^{-3}$	32	$5.05{\cdot}10^{-3}$	11	$5.09{\cdot}10^{-3}$
GMRES	> 2000	$8.68{\cdot}10^{-2}$	13	$5.07{\cdot}10^{-3}$	31	$5.06{\cdot}10^{-3}$	11	$5.08{\cdot}10^{-3}$

4×4 subdomains; DoF $= 256$; $N = 1600$; $\boldsymbol{\theta}^0 \in \mathcal{U}(-1, 1)$; stop.: $\|\boldsymbol{M}^{-1}\boldsymbol{r}^k\|_{L^2} / \|\boldsymbol{M}^{-1}\boldsymbol{r}^0\|_{L^2} \leq 10^{-5}$

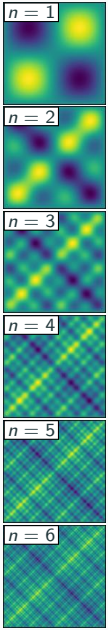
Results for the Multi-Frequency Problem



Multi-level FBPINNs; cf. Dolean, Heinlein, Mishra, Moseley (2024)



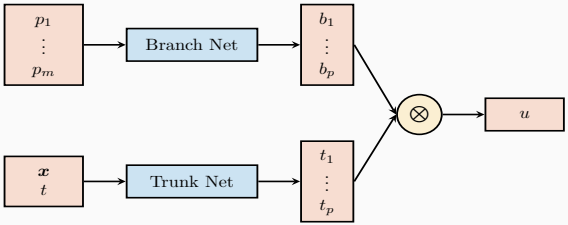
DD-PIRaNNs; cf. Shang, Heinlein, Mishra, Wang (2025)



Deep operator networks – Error analysis and localization via domain decompostion

Deep Operator Networks (DeepONets / DONs)

Neural operators learn operators between function spaces using neural networks. Here, we learn the **solution operator** of a initial-boundary value problem parametrized with $p_1, \ldots, p_m$ using **DeepONets** as introduced in **Lu et al. (2021)**.



Single-layer case

The DeepONet architecture is based on the **single-layer case** analyzed in **Chen and Chen (1995)**. In particular, the authors show **universal approximation properties for continuous operators**.

The architecture is based on the following ansatz for presenting the parametrized solution

$$u_{(p_1,\ldots,p_m)}(x,t) = \sum_{i=1}^{p} \underbrace{b_i(p_1,\ldots,p_m)}_{\text{branch}} \cdot \underbrace{t_i(x,t)}_{\text{trunk}}$$

Physics-informed DeepONets

DeepONets are **compatible with the PINN approach** but **physics-informed DeepONets (PI-DeepONets)** are challenging to train.

Other operator learning approaches

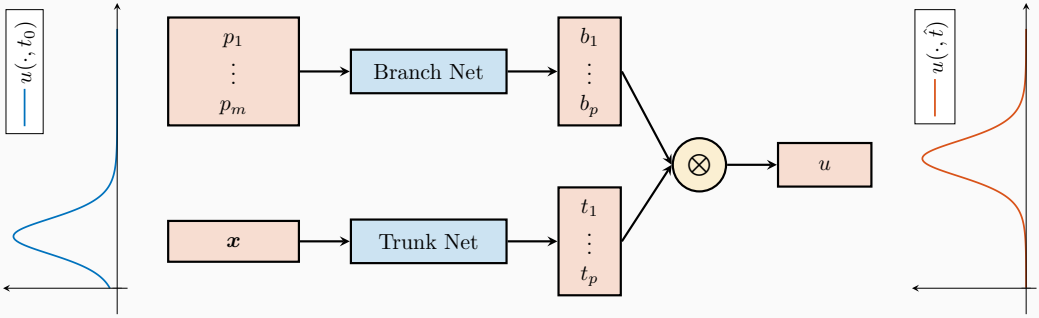
- **FNOs**: **Li et al. (2021)**
- **PCA-Net**: **Bhattacharya et al. (2021)**
- **Random features**: **Nelsen and Stuart (2021)**
- **CNOs**: **Raonić et al. (2023)**

How a DeepONet Maps Between Function Spaces

To illustrate how a DeepONet operates, we consider the **Korteweg–de Vries (KdV) equation**

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - 0.01\frac{\partial^3 u}{\partial x^3},$$

which models unidirectional waves in shallow water. Our goal is to train a DeepONet that predicts the wave profile at a future time $\hat{t}$ from the observed height profile $u(\cdot, t_0)$.



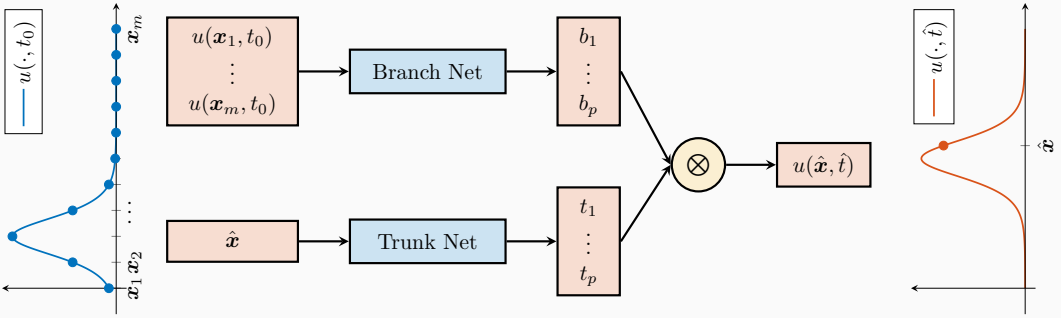
Here, the forecast time $\hat{t}$ is fixed to keep the learning task simple. A **more general neural operator** can take the target time as an additional input to the trunk network.

How a DeepONet Maps Between Function Spaces

To illustrate how a DeepONet operates, we consider the **Korteweg–de Vries (KdV) equation**

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - 0.01\frac{\partial^3 u}{\partial x^3},$$

which models unidirectional waves in shallow water. Our goal is to train a DeepONet that predicts the wave profile at a future time $\hat{t}$ from the observed height profile $u(\cdot, t_0)$.



Here, the forecast time $\hat{t}$ is fixed to keep the learning task simple. A **more general neural operator** can take the target time as an additional input to the trunk network.

DeepONet – Error Decomposition

Given **fixed parametrizations** $\mathscr{P}_1, \dots, \mathscr{P}_M$ and **evaluation points** $\hat{\boldsymbol{x}}_1, \dots, \hat{\boldsymbol{x}}_N$, the reference targets form the matrix

$$A = \begin{bmatrix} u_{\mathscr{P}_1}(\hat{\boldsymbol{x}}_1) & \cdots & u_{\mathscr{P}_M}(\hat{\boldsymbol{x}}_1) \\ \vdots & \ddots & \vdots \\ u_{\mathscr{P}_1}(\hat{\boldsymbol{x}}_N) & \cdots & u_{\mathscr{P}_M}(\hat{\boldsymbol{x}}_N) \end{bmatrix}$$

and the outputs of the DeepONet can be written as

$$\tilde{A} = \underbrace{\begin{bmatrix} t_1(\hat{\boldsymbol{x}}_1) & \cdots & t_m(\hat{\boldsymbol{x}}_1) \\ \vdots & \ddots & \vdots \\ t_1(\hat{\boldsymbol{x}}_N) & \cdots & t_m(\hat{\boldsymbol{x}}_N) \end{bmatrix}}_{=:T} \underbrace{\begin{bmatrix} b_1(\mathscr{P}_1) & \cdots & b_1(\mathscr{P}_M) \\ \vdots & \ddots & \vdots \\ b_m(\mathscr{P}_1) & \cdots & b_m(\mathscr{P}_M) \end{bmatrix}}_{=:B^\top} = \begin{bmatrix} t_1 & \cdots & t_m \end{bmatrix} \begin{bmatrix} b_1^\top \\ \vdots \\ b_m^\top \end{bmatrix},$$

where B and T are discretizations of the branch and trunk net outputs. Similar to **Lanthaler et al. (2022)**, we derive an **error decomposition into branch and trunk errors**

$$\mathscr{E} := \|A - \tilde{A}\|_F^2 = \underbrace{\|T(T^+A - B^\top)\|_F^2}_{=:\mathscr{E}_B} + \underbrace{\|(I - TT^+)A\|_F^2}_{=:\mathscr{E}_T},$$

where $\|\cdot\|_F$ is the **Frobenius norm** and T^+ is the **pseudoinverse** of T.

Analyzing the DeepONet Using the Singular Value Decomposition

A **singular value decomposition (SVD)** of the target matrix A (rank k) reads as follows:

$$A = \Phi \Sigma V^\top = \begin{bmatrix} \phi_1 & \cdots & \phi_k \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} \begin{bmatrix} v_1^\top \\ \vdots \\ v_k^\top \end{bmatrix} = \sum_{j=1}^{k} \sigma_j \phi_j v_j^\top,$$

where Φ and V are semi-orthogonal matrices. The **Eckart–Young theorem** states that the **best low-rank approximation of a matrix in the Frobenius norm is given by truncating its SVD**.

The structure is the same for the discretized DeepONet

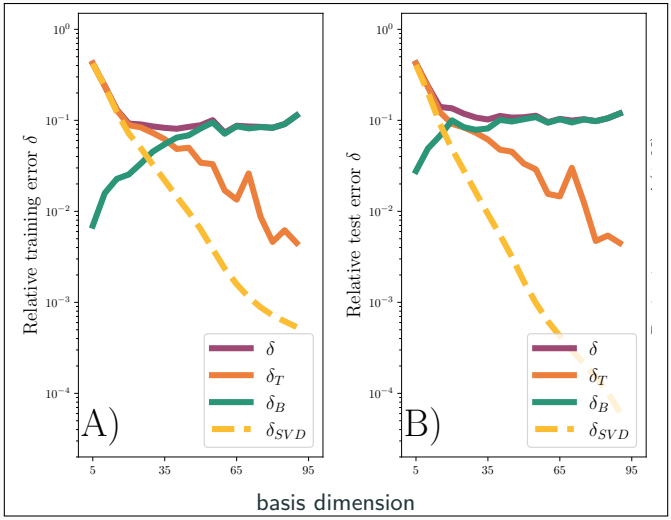
$$\tilde{A} = \begin{bmatrix} t_1 & \cdots & t_m \end{bmatrix} \begin{bmatrix} b_1^\top \\ \vdots \\ b_m^\top \end{bmatrix}.$$

Therefore, on the training data, we can minimize the discrete approximation error on the training data when choosing:

$$t_i = \phi_i \quad \text{and} \quad b_i = \sigma_i v_i.$$

DeepONet – Error Decomposition Results for the KdV Equation

Results for the **KdV equation** with $t_0 = 0.0$ and $\hat{t} = 0.2$. 900 training and 100 test configurations.



total error δ **trunk error** δ_T **branch error** δ_B **SVD truncation error** δ_{SVD}

DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

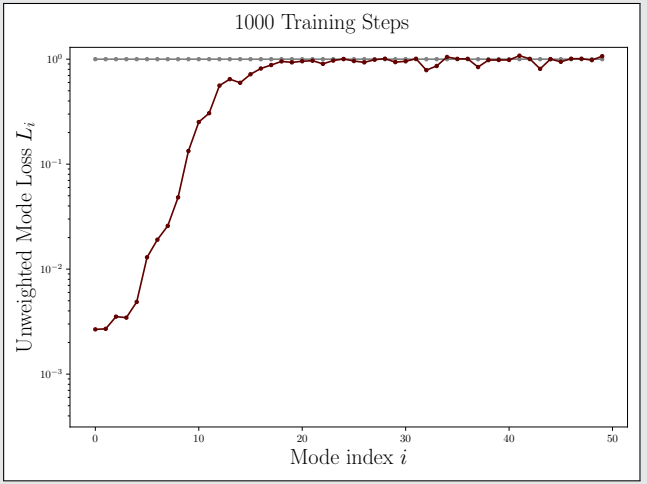
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

Unweighted mode loss



DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

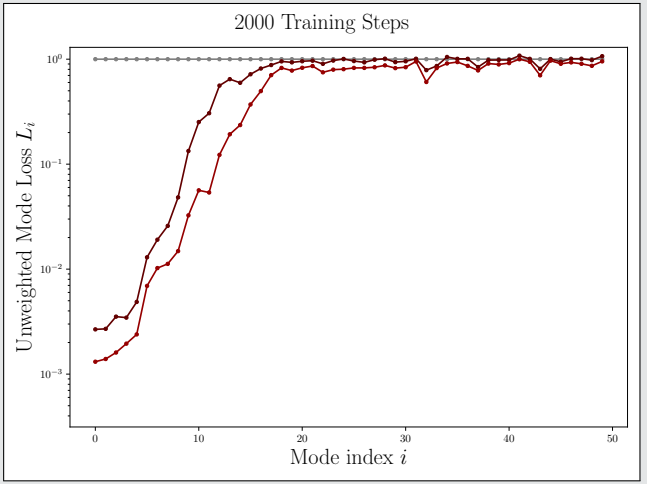
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

Unweighted mode loss



DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

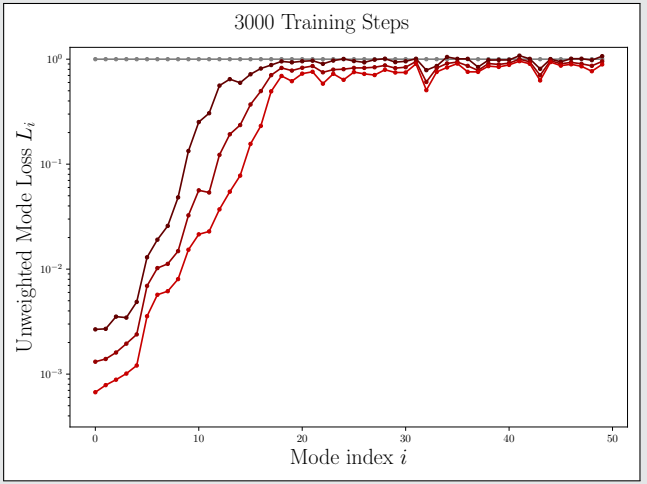
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

Unweighted mode loss



DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

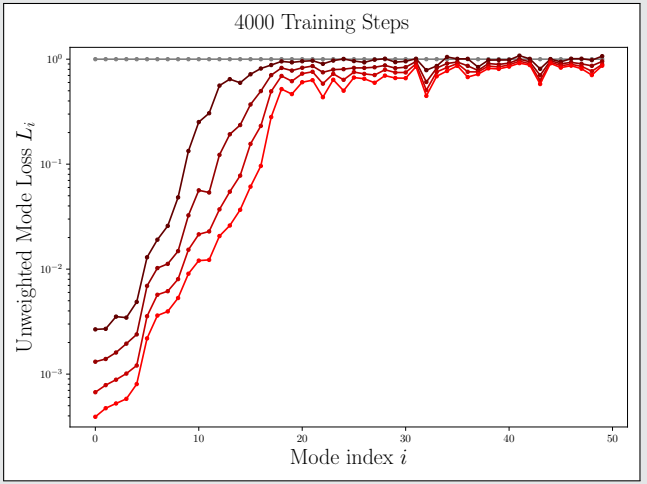
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

Unweighted mode loss



DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

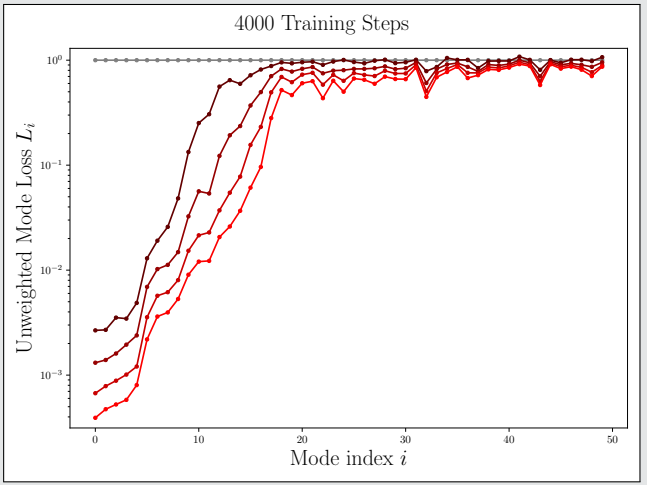
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

Unweighted mode loss



→ The coefficients of modes with **large singular values** are **learned best**. Errors for modes with **small singular values remain high**.

DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

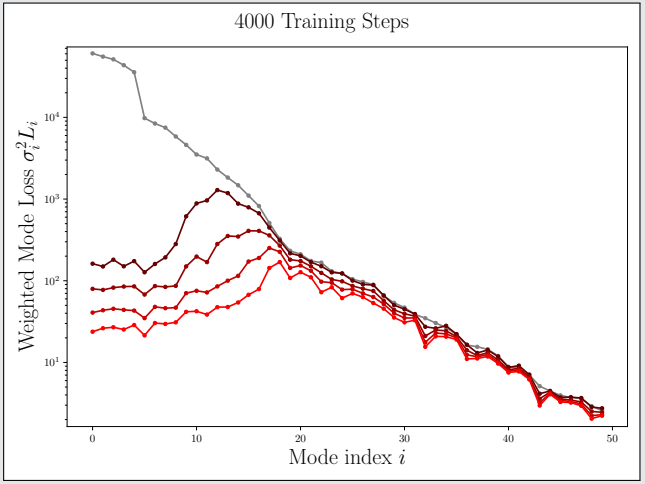
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

(Weighted) mode loss



DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

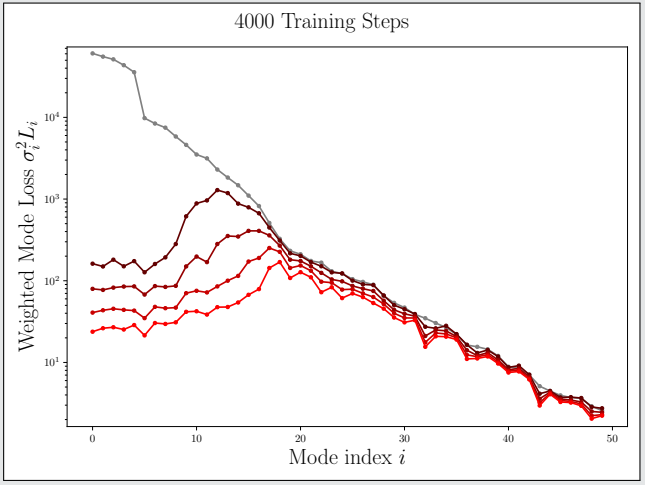
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

(Weighted) mode loss



→ Analyzing the actual error contributions, the **modes with medium singular values contribute most**.

DeepONet – Branch Error

Using the left singular vectors, the **branch error** becomes

$$\mathcal{E}_B = \sum_{i=1}^{m} \sigma_i^2 \underbrace{\|b_i - v_i\|_2^2}_{=:L_i}.$$

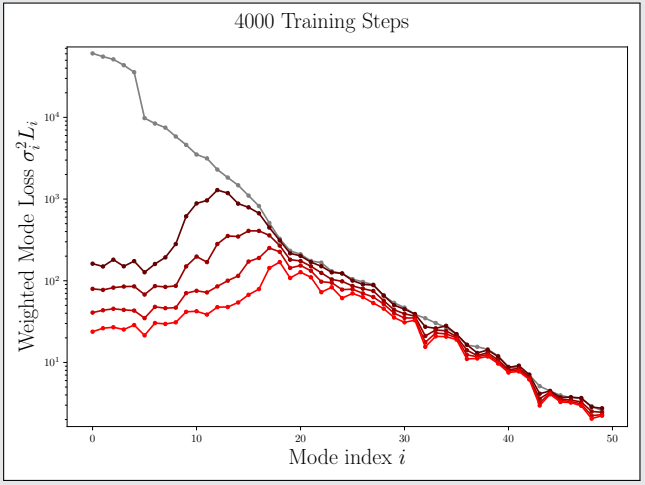
We call

$$\sigma_i^2 L_i$$

the **(weighted) mode loss** because it equals the loss contribution of the ith mode. Accordingly, L_i is the **unweighted mode loss**.

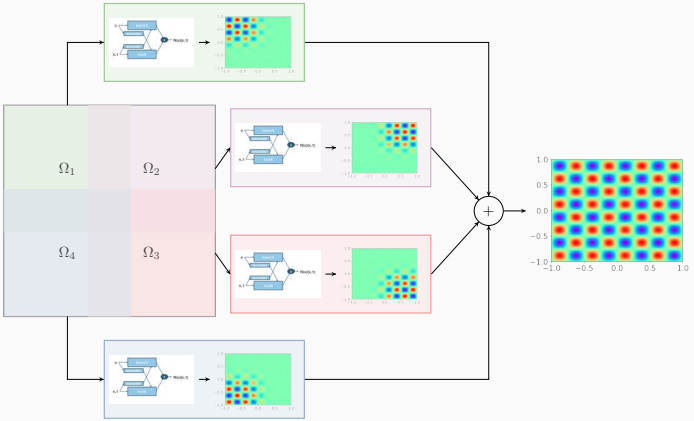
This choice of **left singular vectors as the trunk basis** is often denoted **POD-DeepONet** in **Lu et al. (2022)**.

(Weighted) mode loss



How to improve the performance on medium-sized singular value modes?

Finite Basis DeepONets (FBDONs)



Howard, Heinlein, Stinis (in prep.)

Variants:

Shared-trunk FBDONs (ST-FBDONs)

The trunk net learns spatio-temporal basis functions. In ST-FBDONs, we use the **same trunk network for all subdomains**.

Stacking FBDONs

Combination of the **stacking multifidelity approach** with FBDONs.

Heinlein, Howard, Beecroft, Stinis (2025)

FBDONs – Wave Equation

Wave equation

$$\frac{d^2 s}{dt^2} = 2\frac{d^2 s}{dx^2}, \qquad (x,t) \in [0,1]^2$$

$$s_t(x,0) = 0, x \in [0,1], \quad s(0,t) = s(1,t) = 0,$$

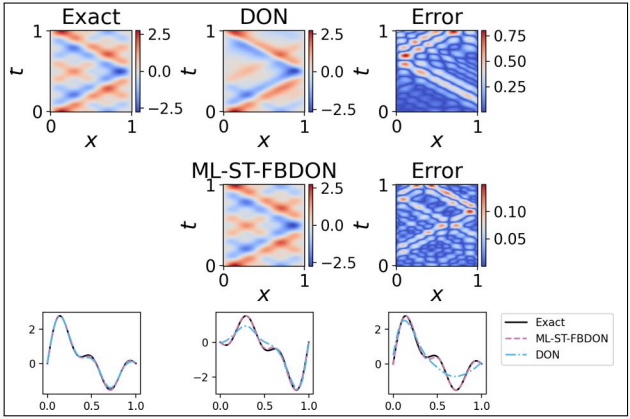
Solution: $s(x,t) = \sum_{n=1}^{5} b_n \sin(n\pi x)\cos\left(n\pi\sqrt{2}t\right)$

Parametrization

Initial conditions for s parametrized by $b = (b_1, \ldots, b_5)$ (normally distributed):

$$s(x,0) = \sum_{n=1}^{5} b_n \sin(n\pi x) \quad x \in [0,1]$$

Training on 1 000 random configurations.



Mean rel. l_2 error on 100 config.	
DeepONet	0.30 ± 0.11
ML-ST-FBDON ([1, 4, 8, 16] subd.)	0.05 ± 0.03
ML-FBDON ([1, 4, 8, 16] subd.)	0.08 ± 0.04

→ Sharing the trunk network does not only save in the number of parameters but even yields **better performance**

Cf. **Howard, Heinlein, Stinis (in prep.)**

Summary

Multilevel Finite Basis Physics Informed Neural Networks (ML-FBPINNs)

- **Schwarz domain decomposition architectures improve the scalability of PINNs** to **large domains** / **high frequencies**, **keeping the complexity of the local networks low**.
- As classical domain decomposition methods, **one-level FBPINNs** are **not scalable to many subdomains**; **multilevel FBPINNs enable scalability**.

Extensions to Stacking Multifidelity PINNs, RaNNs, and DeepONets

- **Multifidelity stacking PINNs with FBPINNs** improve **accuracy and efficiency** for **time-dependent problems**.
- **RaNNs** reduce computational cost but face **ill-conditioning**, mitigated by **Schwarz preconditioning** and **SVD**.
- **DeepONets** provide **efficient predictions** for **parametrized problems** but struggle with **multiscale problems**. **Domain decomposition improves scalability and performance**.

Thank you for your attention!

Topical Activity Group

Scientific Machine Learning