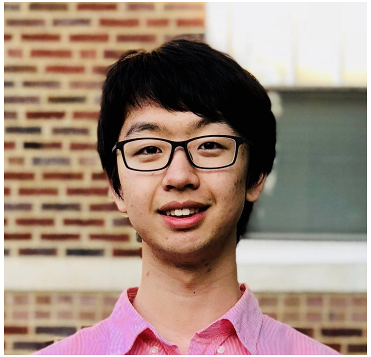
Neural Operator For Parametric PDEs

Nov, 2020

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu,
Kaushik Bhattacharya, Andrew Stuart, Anima Anandkumar

Caltech



Zongyi Li

Nikola Kovachki

Burigede Liu

Kamyar Azizzadenesheli

Anima Anandkumar

Andrew Stuart

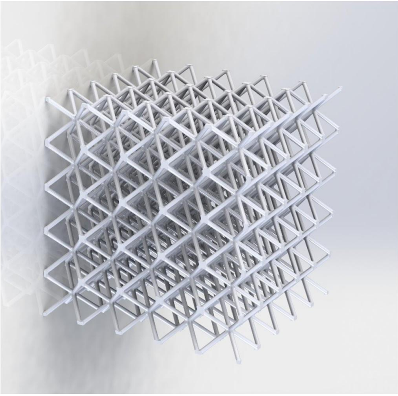
Kaushik Bhattacharya

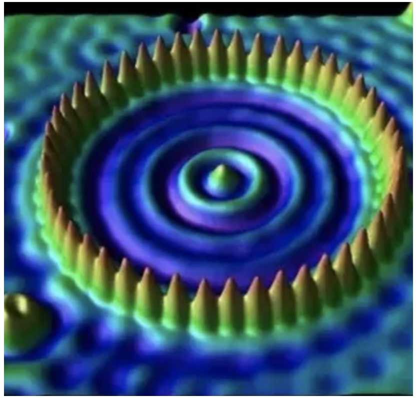
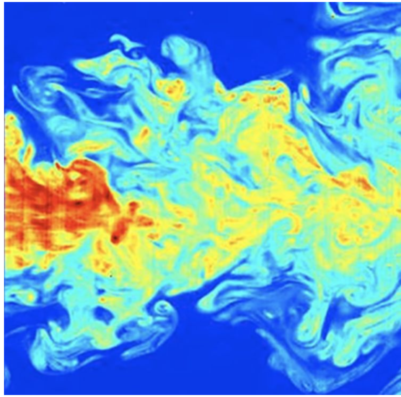
Overview

1. Introduction
   a. Neural operator vs FDM/FEM
   b. Neural operator vs CNN
2. Neural operator
   a. Intuition: Green’s function
   b. Formulation
3. Graph-based operator
4. Fourier neural operator
5. Experiments
6. Future work

1. Introduction

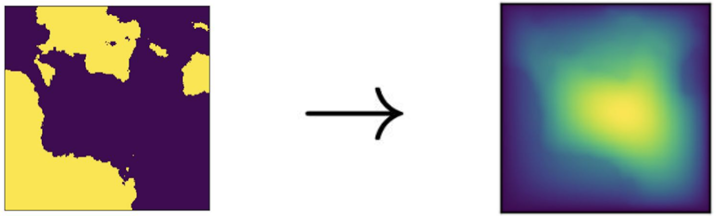
Problems in science and engineering reduce to PDEs.



Introduction

- Learning parametric PDE:
  Given the a set of coefficients/boundary conditions
  Find the solution functions

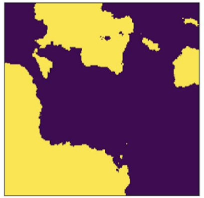


Input: coefficient → Output: solution

Problem Setting

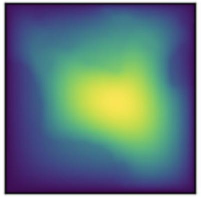
Second order elliptic equation:

$$\begin{aligned} -\nabla \cdot (a(x)\nabla u(x)) &= f(x), \quad & x \in D \\ u(x) &= 0, \quad & x \in \partial D \end{aligned}$$



Input: a(x)

$\rightarrow$



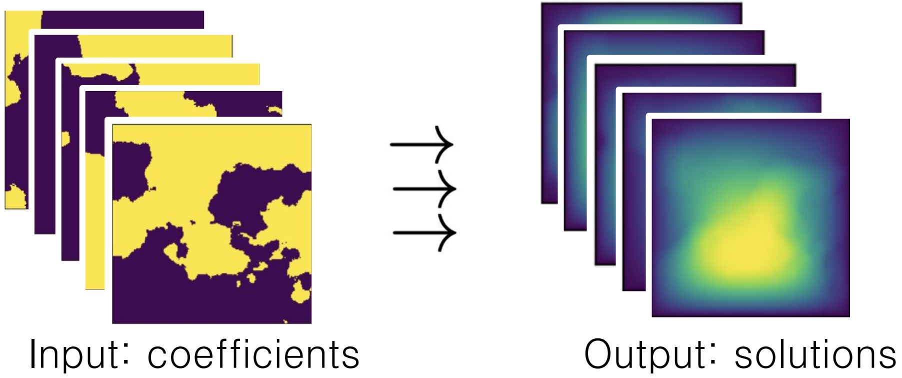
Output: u(x)

$$\mathcal{F} : \mathcal{A} \times \Theta \rightarrow \mathcal{U}$$

Operator learning

Solving PDEs is slow.
Learn the mapping from data (coefficients & solutions pairs).

- Fix an equation
- Multiple training instances
- Learn the mapping

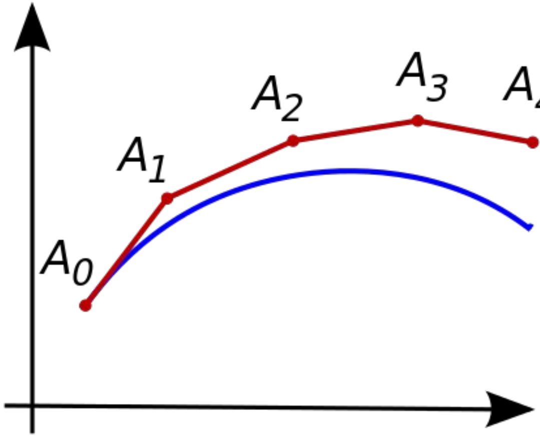


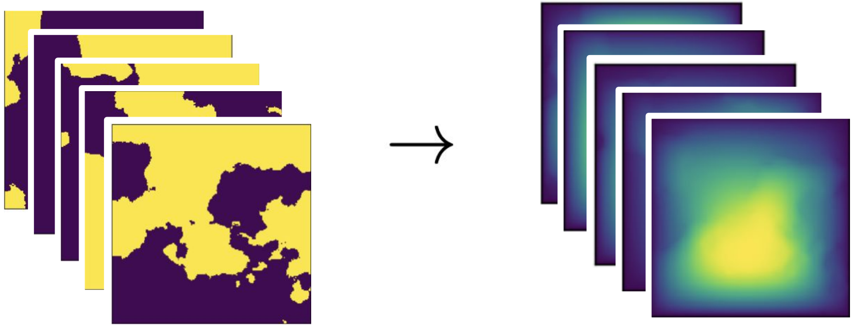
Slow to train. Fast to evaluate. $\mathcal{F} : \mathcal{A} \times \Theta \to \mathcal{U}$

Solve vs learn

Conventional methods:
Solve the equation
By approximation on a mesh

Data-driven methods:
Learn the trajectory
From a distribution





Input: coefficients

Output: solutions

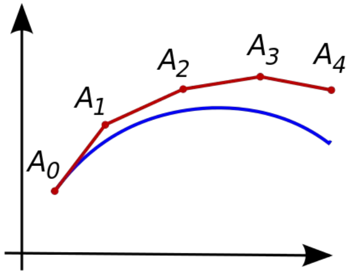
Solve vs learn

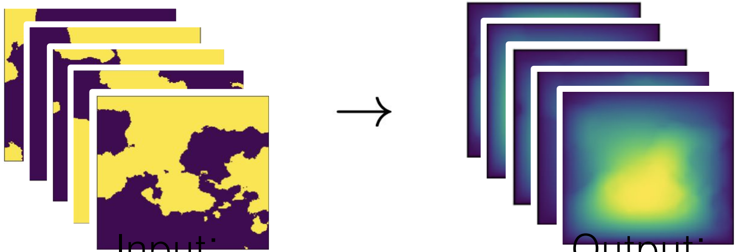
Conventional methods:

- Solve one instance
- Require the explicit form
- trade-off on resolution
- Slow on fine grids; fast on coarse grids

Data-driven methods:

- Learn a family of PDE
- Black-box, data-driven
- Resolution-invariant, mesh-invariant
- Slow to train; fast to evaluate





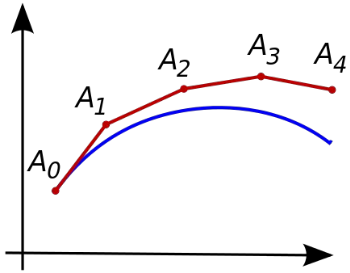
Solve vs learn

Conventional methods:

- Changing parameter
- Changing boundary
- Changing initial condition

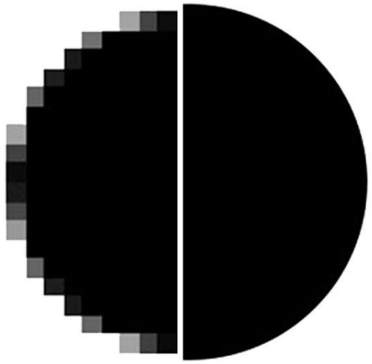
Data-driven methods:

- From a distribution
- Large distribution require more data
- Data could be slow to generate



Operator learning

- Not vector-to-vector mapping.
- But function-to-function mapping.



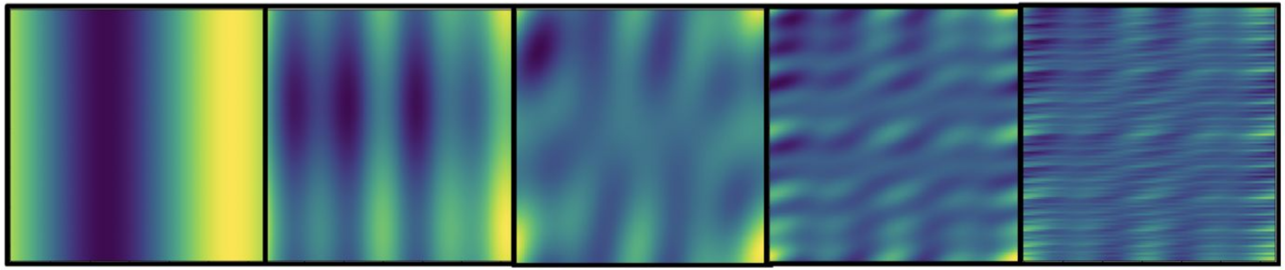
Discretized vector

Continuous function

Operator learning

Key idea: represent function & operator in mesh-invariant way

Filters in CNN



Fourier Filters

2. Neural operator

$$u = (K_l \circ \sigma_l \circ \cdots \circ \sigma_1 \circ K_0)\, v$$

Problem Setting

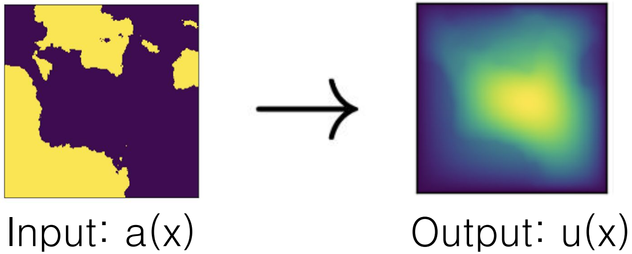
Second order elliptic equation:

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x), \quad x \in D$$
$$u(x) = 0, \quad x \in \partial D$$

Given $\{a_j, u_j\}_{j=1}^N$ pairs of functions

Want to learn the **operator**

$$\mathcal{F} : \mathcal{A} \times \Theta \to \mathcal{U}$$



Intuition: kernel method

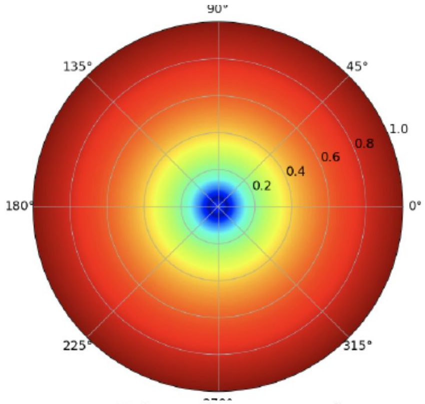
$$-\nabla \cdot (a(x)\nabla u(x)) = f(x), \quad x \in D$$
$$u(x) = 0, \quad x \in \partial D$$

Inverse of differential operator can be written in form of kernel

$$u(x) = \int_D G_a(x,y) f(y)\, dy.$$

Where G is the green function

$$u(x) = \int_D G_a(x,y)[f(y) + (\Gamma_a u)(y)]\, dy.$$



$G_a(x,y).$

Integral Operator

Idea: Approximate the kernel by a **neural network** κ_ϕ

$$u(x) = \int_D G_a(x,y)[f(y) + (\Gamma_a u)(y)]\, dy.$$

$$\int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y)\, \nu_x(dy)$$

Iterative solver: stack layers

$$u(x) = \int_D G_a(x,y) f(y)\, dy.$$

$$\int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y)\, \nu_x(dy)$$

Add iterations for t = 1,…,T, like an implicit method

$$K : v_t \mapsto v_{t+1}$$

$$v_{t+1}(x) = \sigma\left(W v_t(x) + \int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y)\, \nu_x(dy) \right)$$

Neural operator

$$u = (K_l \circ \sigma_l \circ \cdots \circ \sigma_1 \circ K_0)\, v$$

W are linear non-local integral operator
σ are non-linear local activation functions

Neural operator

$$u = Q\left(K_l \circ \sigma_l \circ \cdots \circ \sigma_1 \circ K_0\right) P\, v$$

P, Q are local network (encoder, decoder)

P lifts the input to a high dimensional channel space.
Q projects the representation back to the original space

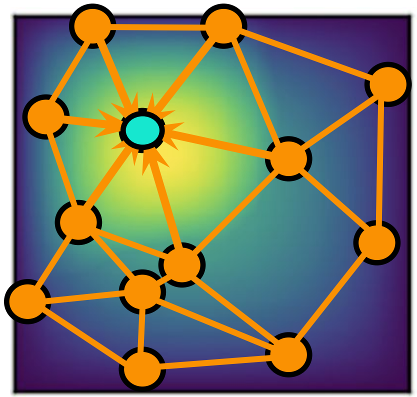
Neural operator

$$\int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y)\, \nu_x(dy)$$

Four variations:

1. Graph neural operator
2. Multipole graph neural operator
3. Low-rank neural operator
4. Fourier neural operator

3. Graph-based Neural operator

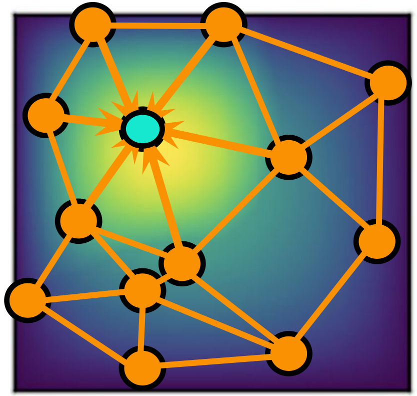


Kernel convolution as message passing on graph

$$v_{t+1}(x) = \sigma\left(W v_t(x) + \int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y)\, \nu_x(dy) \right)$$

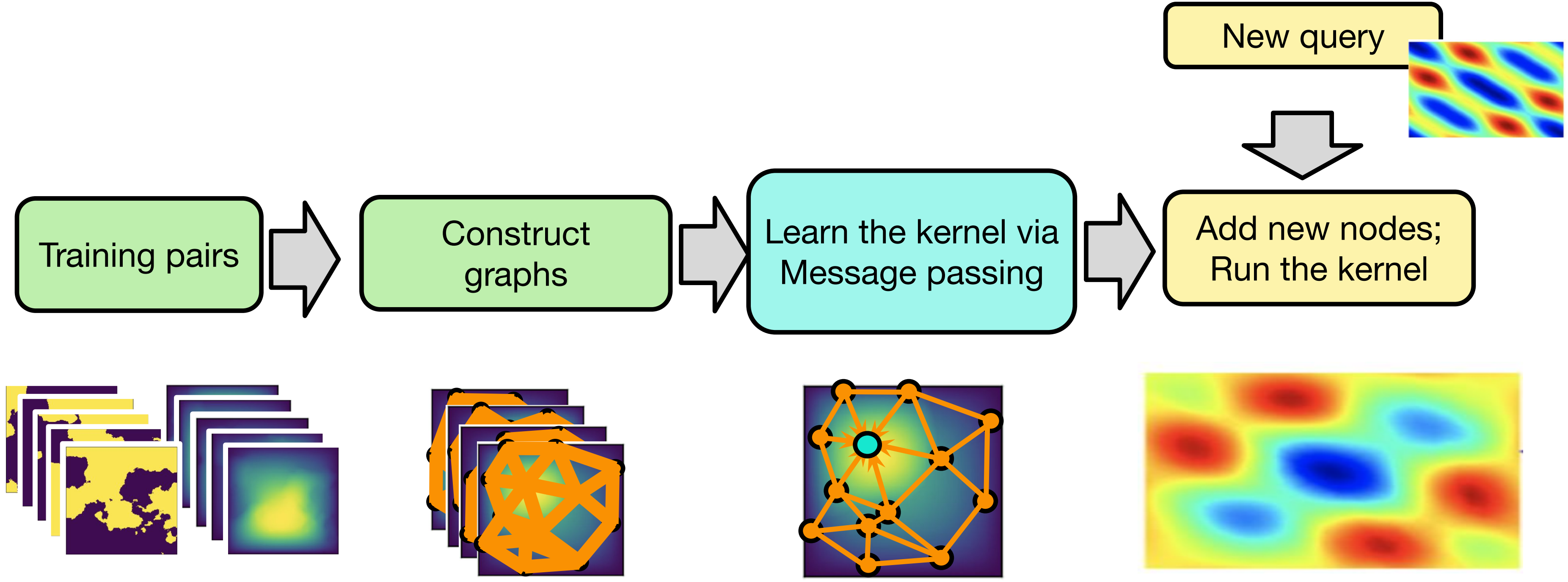
$$v_{t+1}(x) = W v_t(x) + \sum_{y \in N(x)} \kappa_\phi\big(e(x, y)\big) v_t(y)$$

Graph neural network

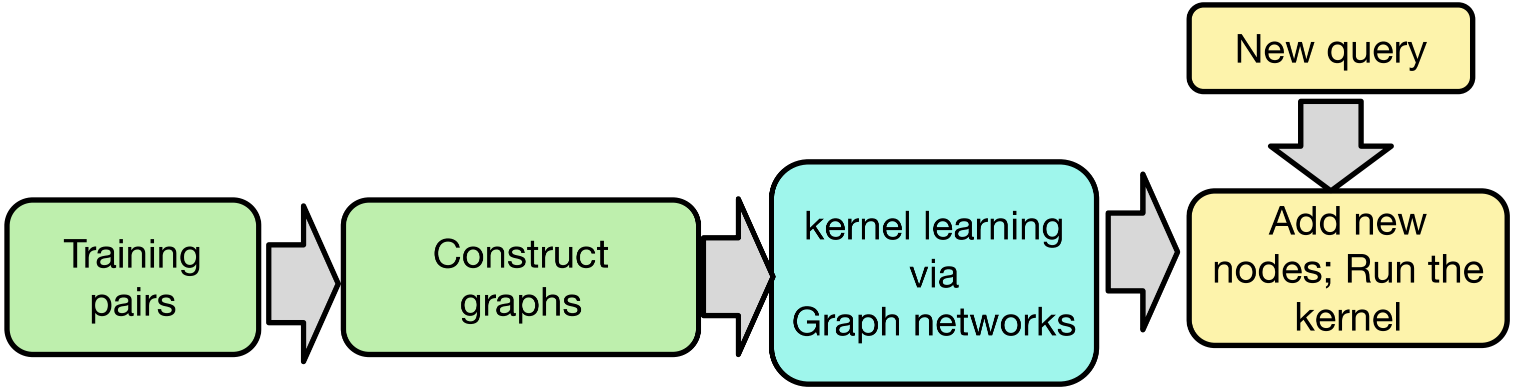


- Adjacency matrix = kernel matrix.
- Kernel integration = message passing

Graph Kernel Operator



Training and Testing



Training:

- for each training pair (a, u), sample several random graphs.
- Learn a universal kernel.

Testing:

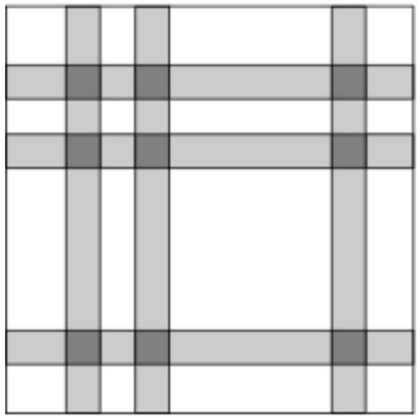
- To evaluate at a specific location, simply add a node at this location.
- No interpolation needed.

Nystrom Approximation

Computation scales with the number of edges.
On an s-by-s grid, O(E) = O(K^2) = O(s^4)

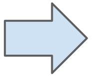
Nystrom Approximation:
Sample a small number of nodes (m).

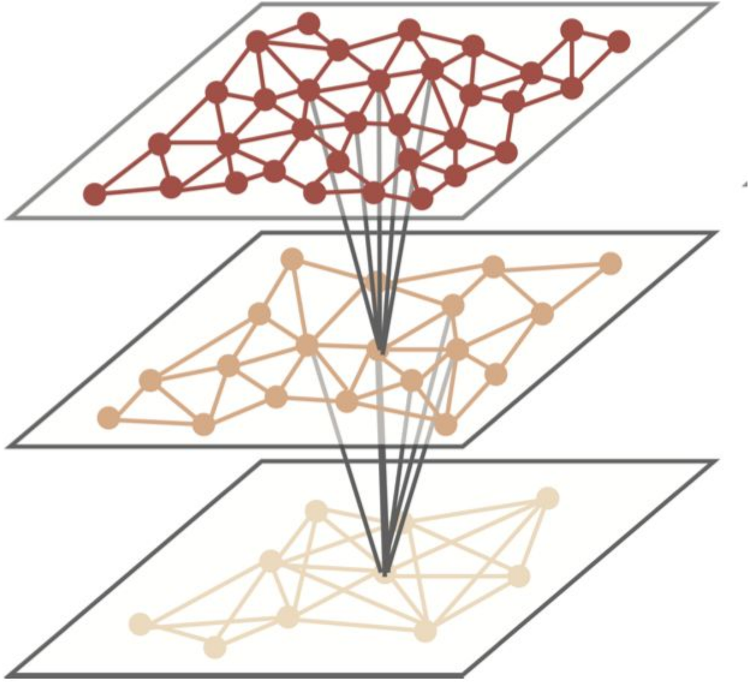
No need to sample too many nodes!
In practice, m ~ 200 is sufficient,
invariant of the resolution s.



Multipole graph method

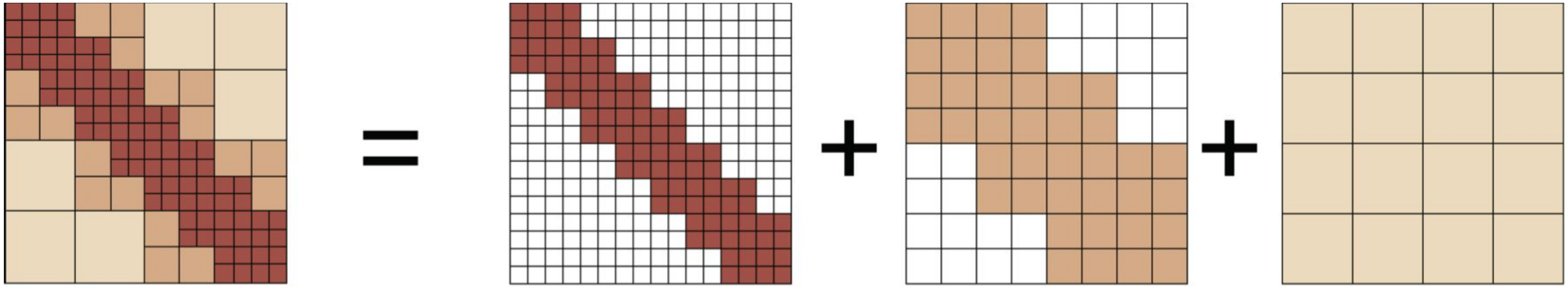
- Construct multi-level of graphs
- Long-term interaction captured by coarser-level graphs

 Multipole method



Hierarchical matrix

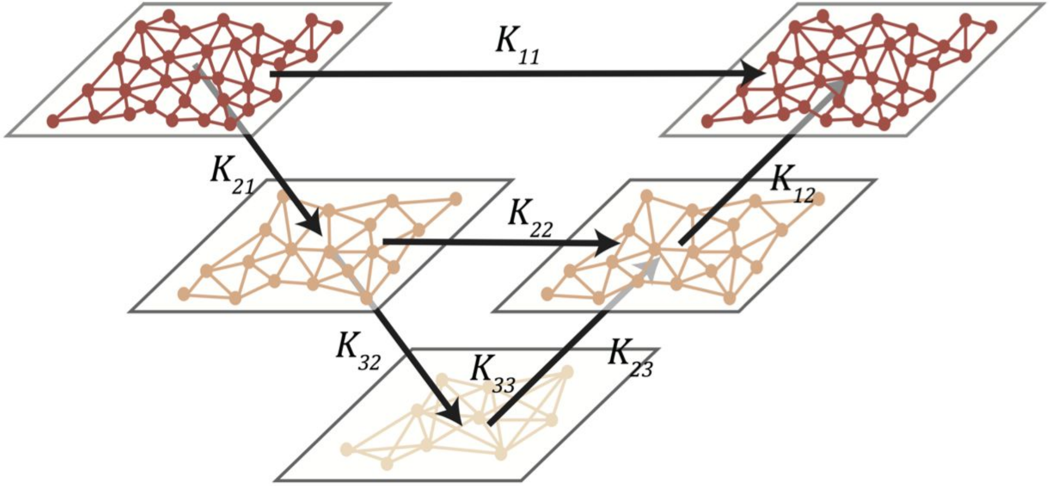
- Insight: the long-range interaction is usually smooth
- Decompose the interaction into different ranges
  - Short-range matrix is sparse
  - Long-range matrix is low-rank



$$K = K_1 + K_2 + \ldots + K_L$$

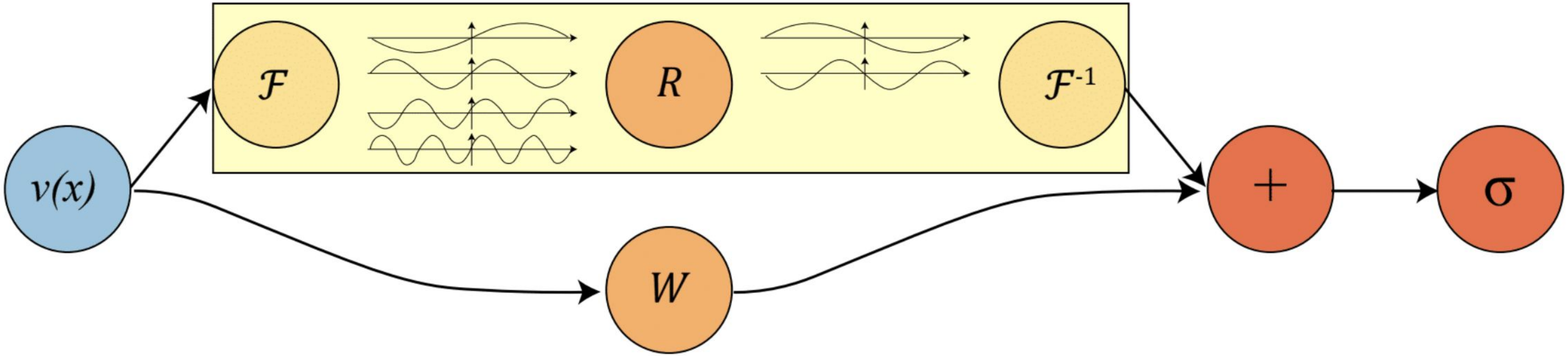
Multi-resolution decomposition

- Recursive low-rank structure = multi-resolution decomposition
- Equivalent to message passing via V-cycle algorithm



$$K \approx K_{1,1} + K_{1,2}K_{2,2}K_{2,1} + K_{1,2}K_{2,3}K_{3,3}K_{3,2}K_{2,1} + \cdots$$

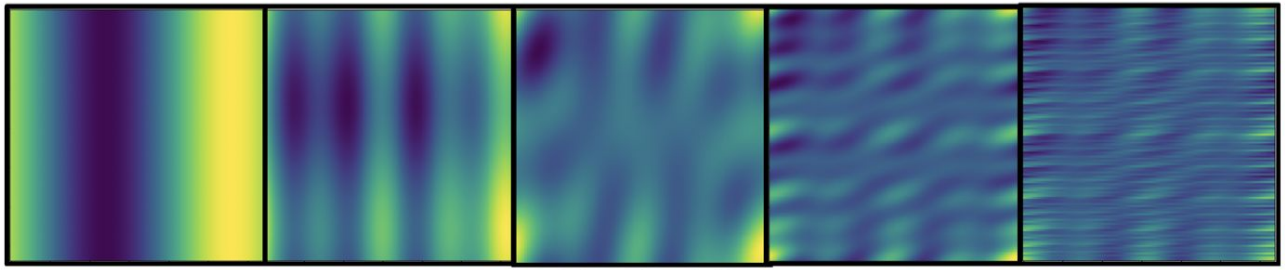
4. Fourier neural operator



Fourier filters

Fourier representation is more efficient than CNN.

Filters in CNN



Fourier Filters

Fourier layer

Use convolution as the integral operator
and implement with Fourier transform

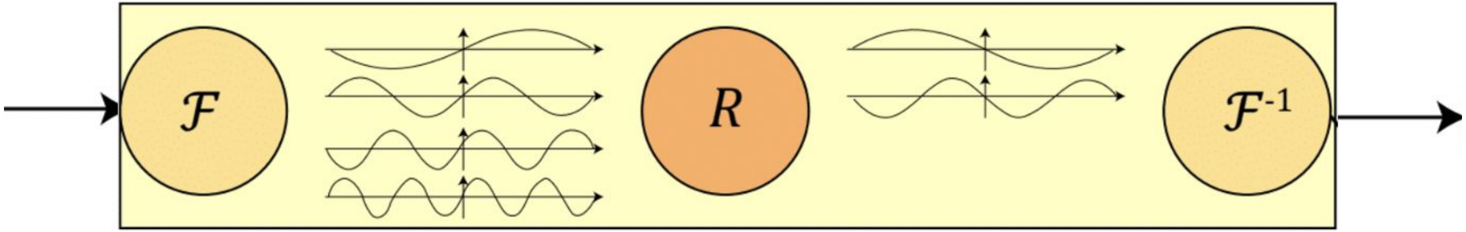
$$\big(\mathcal{K}(a;\phi)v_t\big)(x) := \int_D \kappa\big(x, y, a(x), a(y); \phi\big)v_t(y)\mathrm{d}y,$$

$$\big(\mathcal{K}(\phi)v_t\big)(x) = \mathcal{F}^{-1}\Big(R_\phi \cdot (\mathcal{F}v_t)\Big)(x)$$

Fourier layer

1. Fourier transform
2. Linear transform
3. Inverse Fourier transform

$$(\mathcal{K}(\phi)v_t)(x) = \mathcal{F}^{-1}\Big(R_\phi \cdot (\mathcal{F}v_t)\Big)(x)$$



Fourier layer

```
def forward(self, x):
    batchsize = x.shape[0]
    #Compute Fourier coeffcients up to factor of e^(- something constant)
    x_ft = torch.rfft(x, 2, normalized=True, onesided=True)

    # Multiply relevant Fourier modes
    out_ft = torch.zeros(batchsize, self.in_channels,  x.size(-2), x.size(-1)//2 + 1, 2, device=x.device)
    out_ft[:, :, :self.modes1, :self.modes2] = \
        compl_mul2d(x_ft[:, :, :self.modes1, :self.modes2], self.weights1)
    out_ft[:, :, -self.modes1:, :self.modes2] = \
        compl_mul2d(x_ft[:, :, -self.modes1:, :self.modes2], self.weights2)

    #Return to physical space
    x = torch.irfft(out_ft, 2, normalized=True, onesided=True, signal_sizes=( x.size(-2), x.size(-1)))
    return x
```

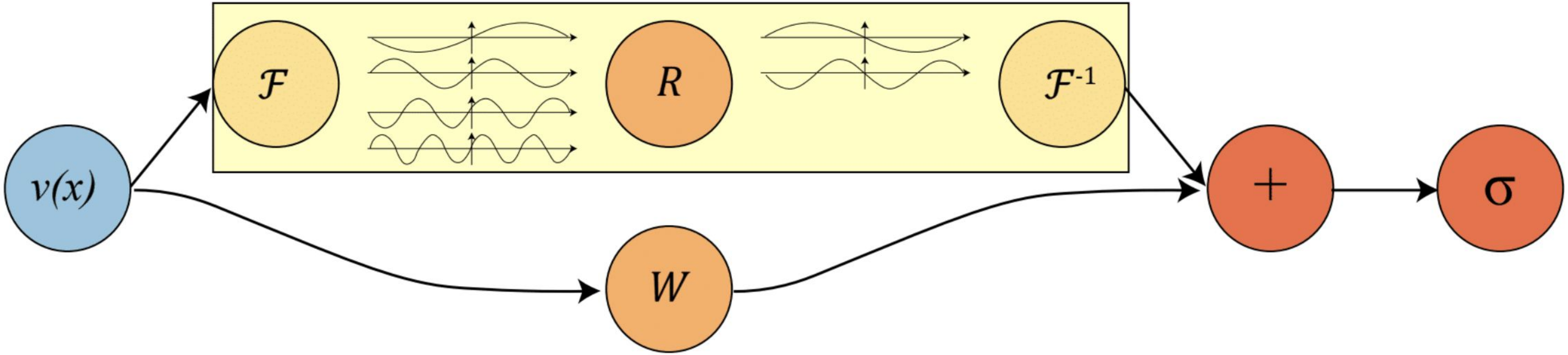
Fourier layer

Encoding & decoding
Activation function on the spatial domain

Recover high frequency modes

Fourier layer

The linear transform *W* outside keep the track of the location information (x) and non-periodic boundary



$$v_{t+1}(x) = \sigma\left(W v_t(x) + \int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y)\, \nu_x(dy) \right)$$

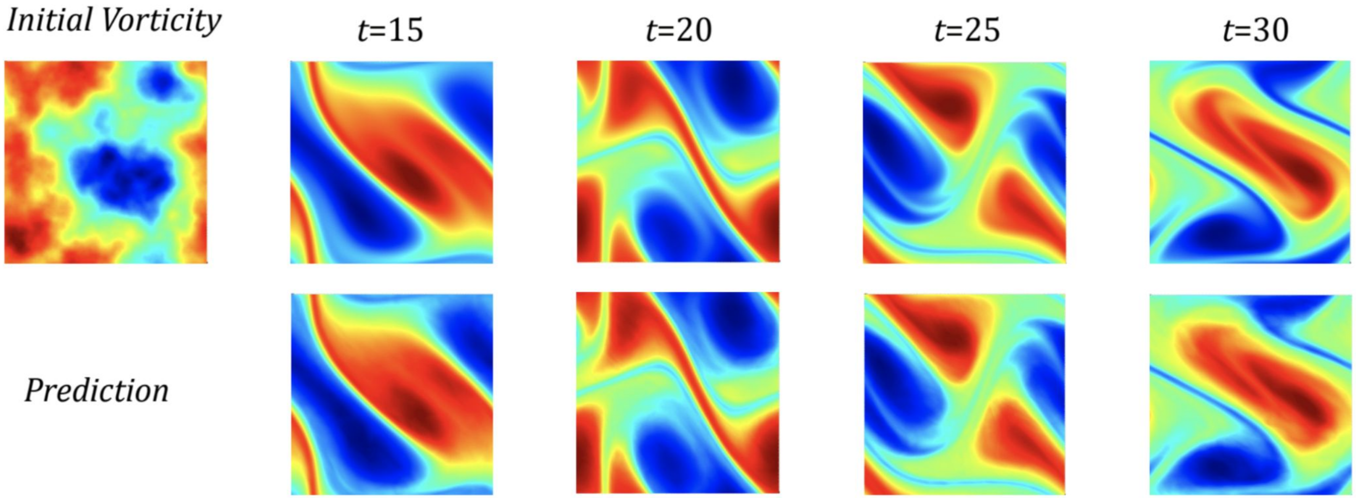
Fourier layer

Complexity:

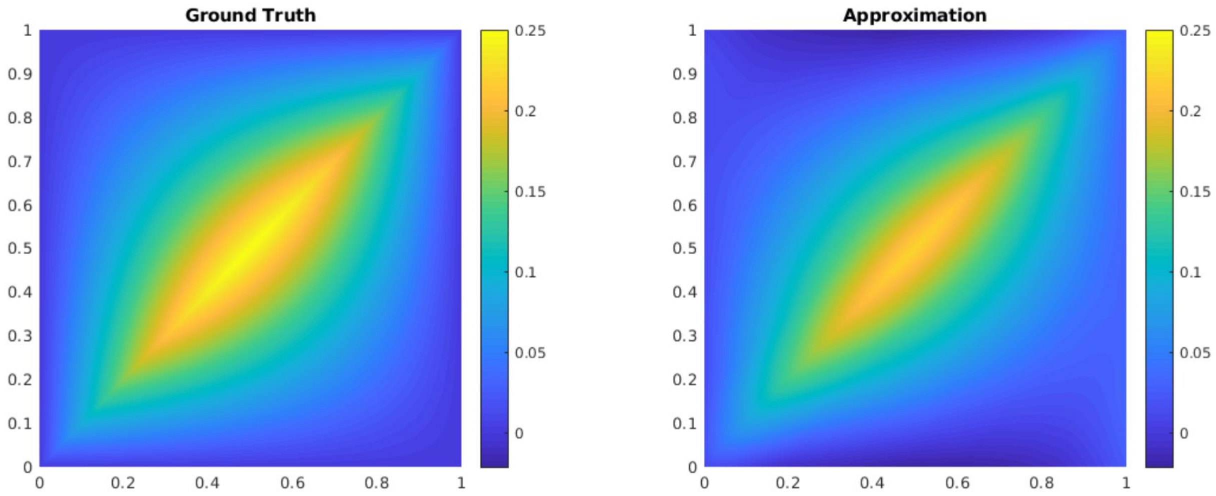
- Fourier transform O(k n)
- FFT O(nlogn)
- Linear O(n)

Resolution-invariant
Mesh-invariant

5. Experiments



Example 1: 1d-Poisson



Sanity check: the learned neural network kernel is very closed to the true analytic kernel

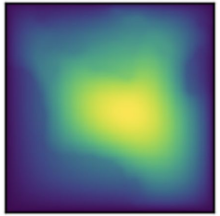
Example 2: 2d Darcy Flow

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x) \qquad x \in (0,1)^2$$

$$u(x) = 0 \qquad x \in \partial(0,1)^2$$

Input: coefficient

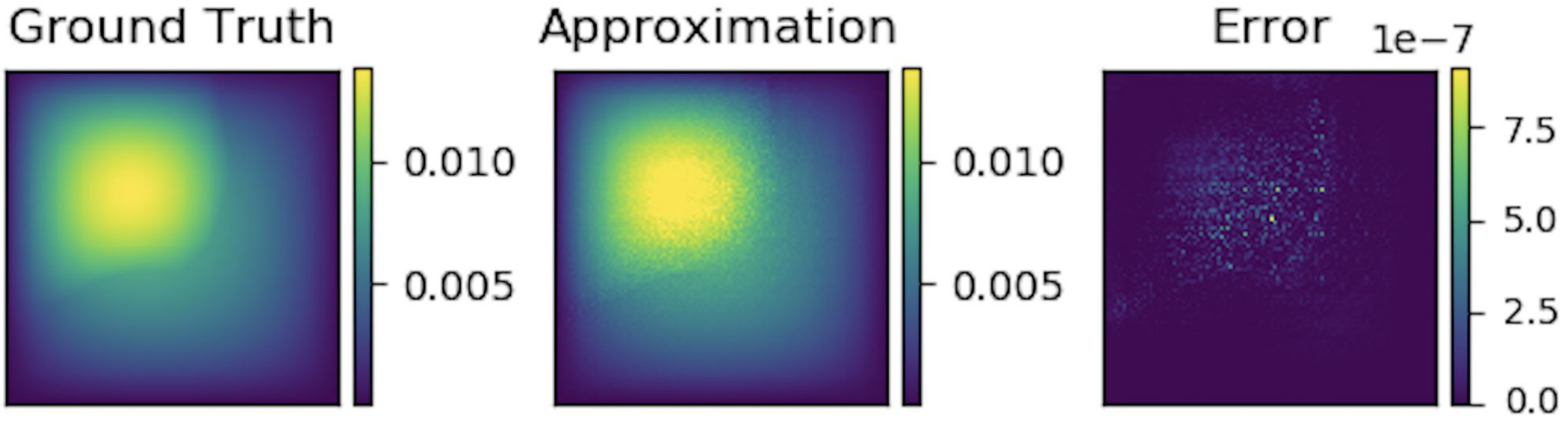
$\longrightarrow$



Output: solution

$$a \sim \mu \text{ where } \mu = \psi_{\#}\mathcal{N}(0, (-\Delta + 9I)^{-2})$$

Train on 16*16, test on 241*241



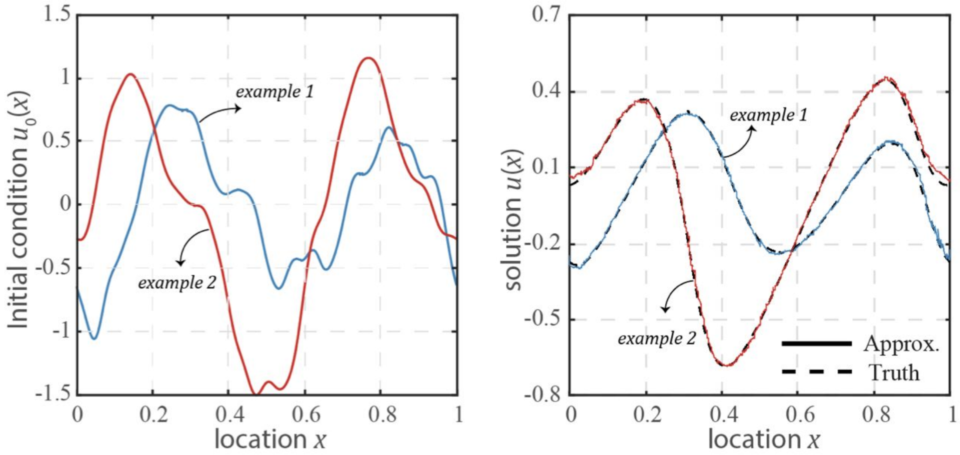
(Plot for the absolute squared error.
Average relative l2 error ~ 0.05)

Graph kernel network does super-resolution

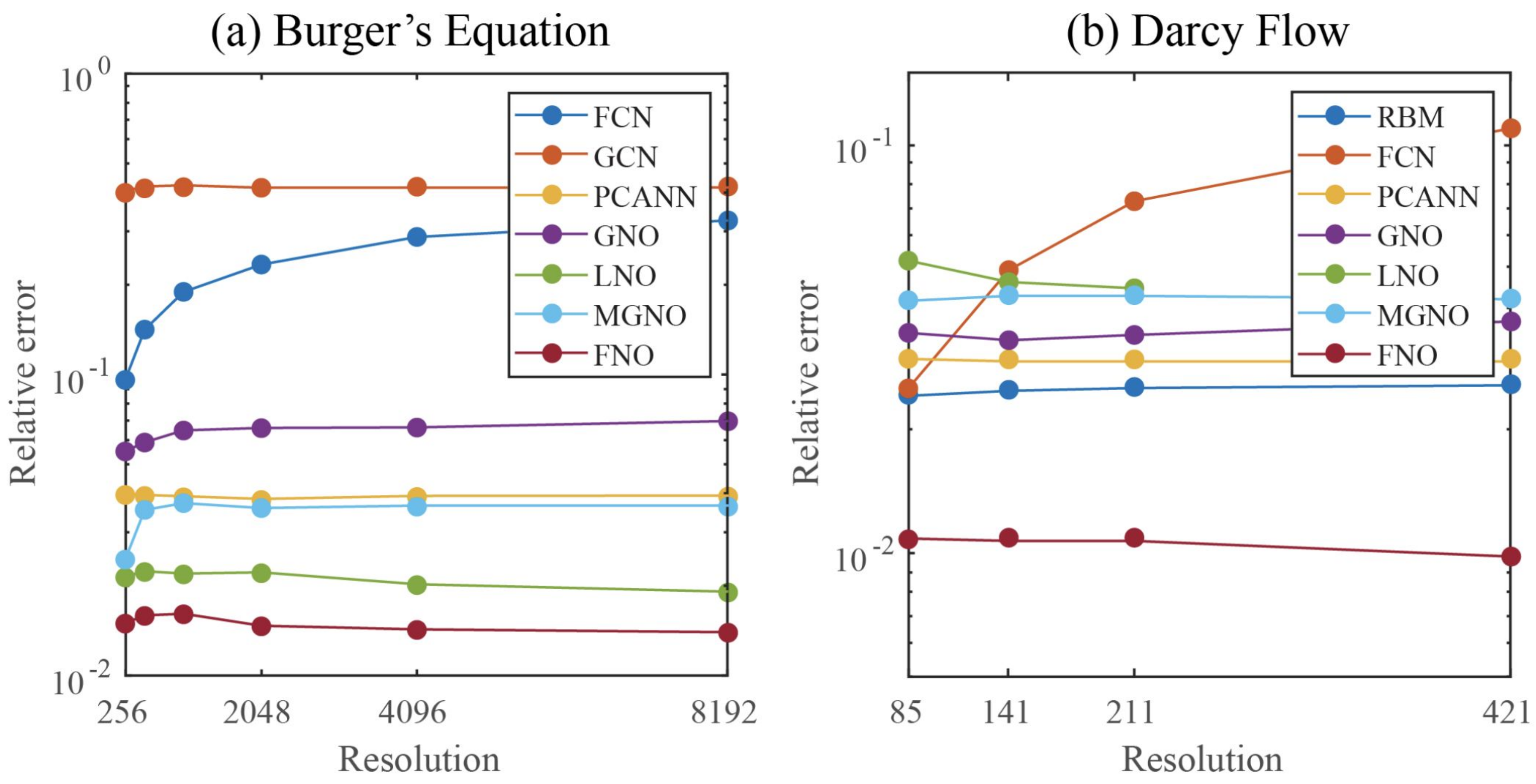
Example 3: 1d Burgers

$$\partial_t u(x,t) + \partial_x(u^2(x,t)/2) = \nu \partial_{xx} u(x,t), \qquad x \in (0,1), t \in (0,1]$$

$$u(x,0) = u_0(x), \qquad x \in (0,1)$$



$$u_0 \sim \mu \text{ where } \mu = \mathcal{N}(0, 625(-\Delta + 25I)^{-2})$$



(a) Burger's Equation
(b) Darcy Flow
Relative error
Resolution
FCN
GCN
PCANN
GNO
LNO
MGNO
FNO
RBM
256
2048
4096
8192
85
141
211
421

Example 4: Navier-Stokes

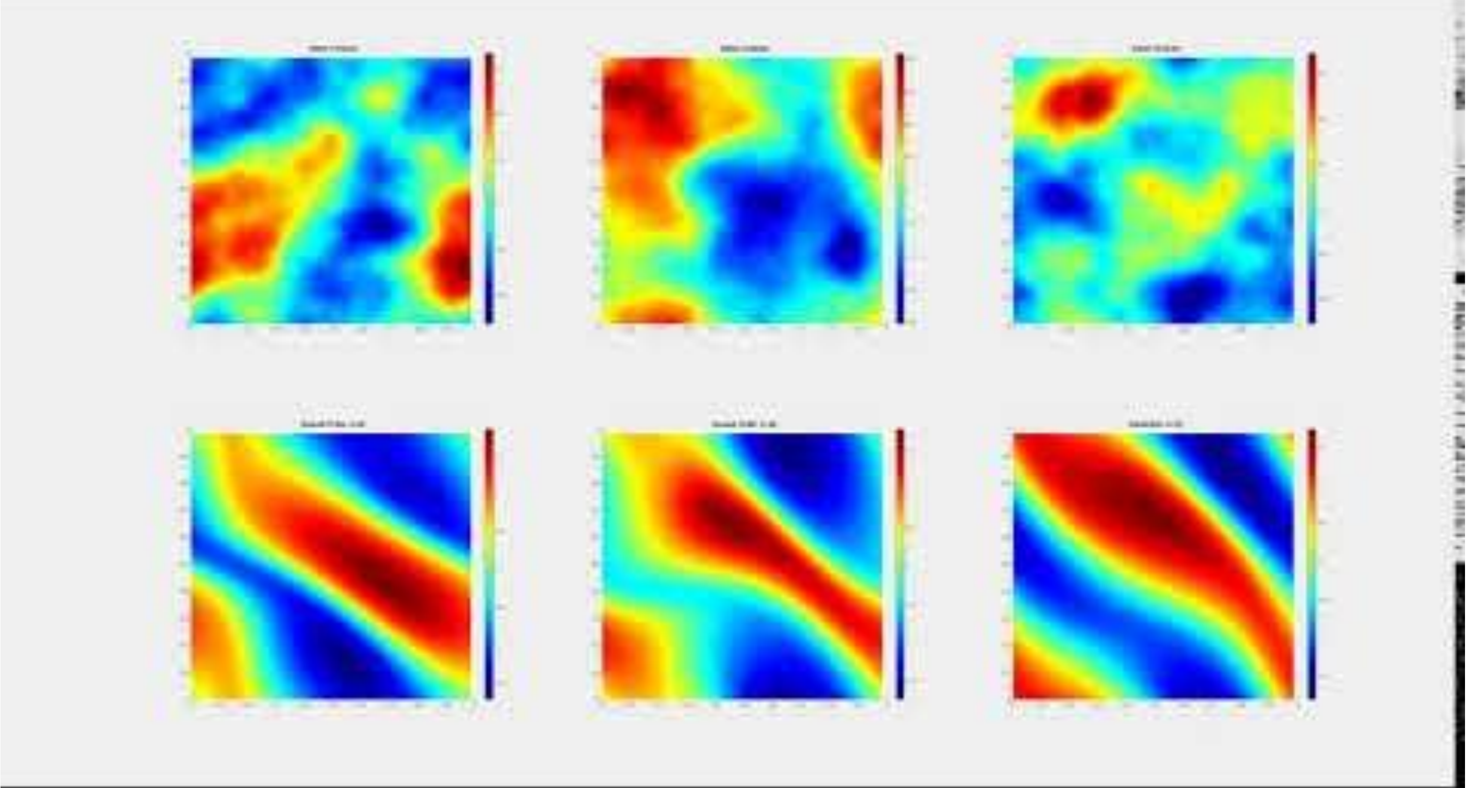
$$\begin{aligned}
\partial_t w(x,t) + u(x,t)\cdot\nabla w(x,t) &= \nu\Delta w(x,t) + f(x), && x\in(0,1)^2, t\in(0,T] \\
\nabla\cdot u(x,t) &= 0, && x\in(0,1)^2, t\in[0,T] \\
w(x,0) &= w_0(x), && x\in(0,1)^2
\end{aligned}$$

$$f(x) = 0.1(\sin(2\pi(x_1+x_2)) + \cos(2\pi(x_1+x_2)))$$

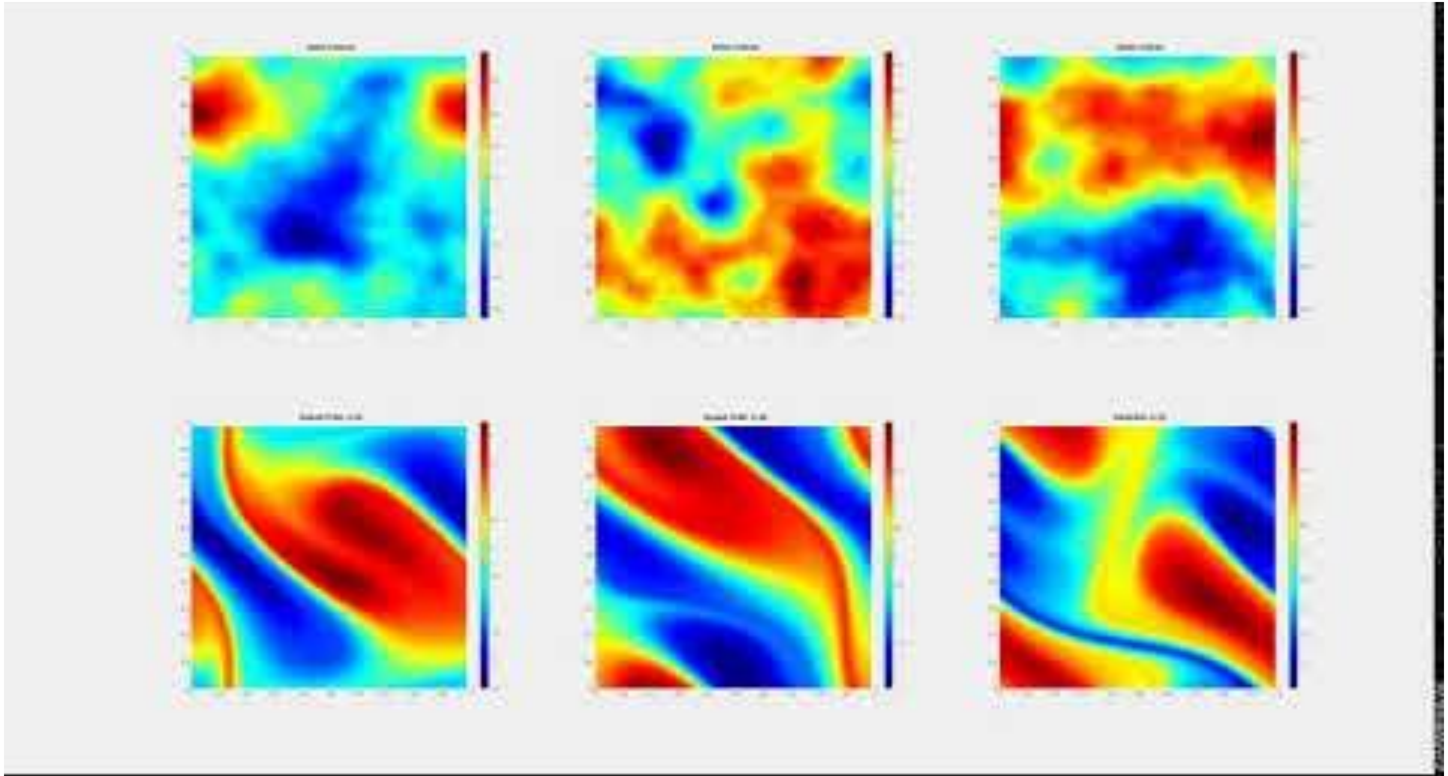
$$w_0 \sim \mu \text{ where } \mu = \mathcal{N}(0, 7^{3/2}(-\Delta + 49I)^{-2.5})$$

$$\text{viscosities } \nu = 1\text{e}{-3}, 1\text{e}{-4}, 1\text{e}{-5}$$

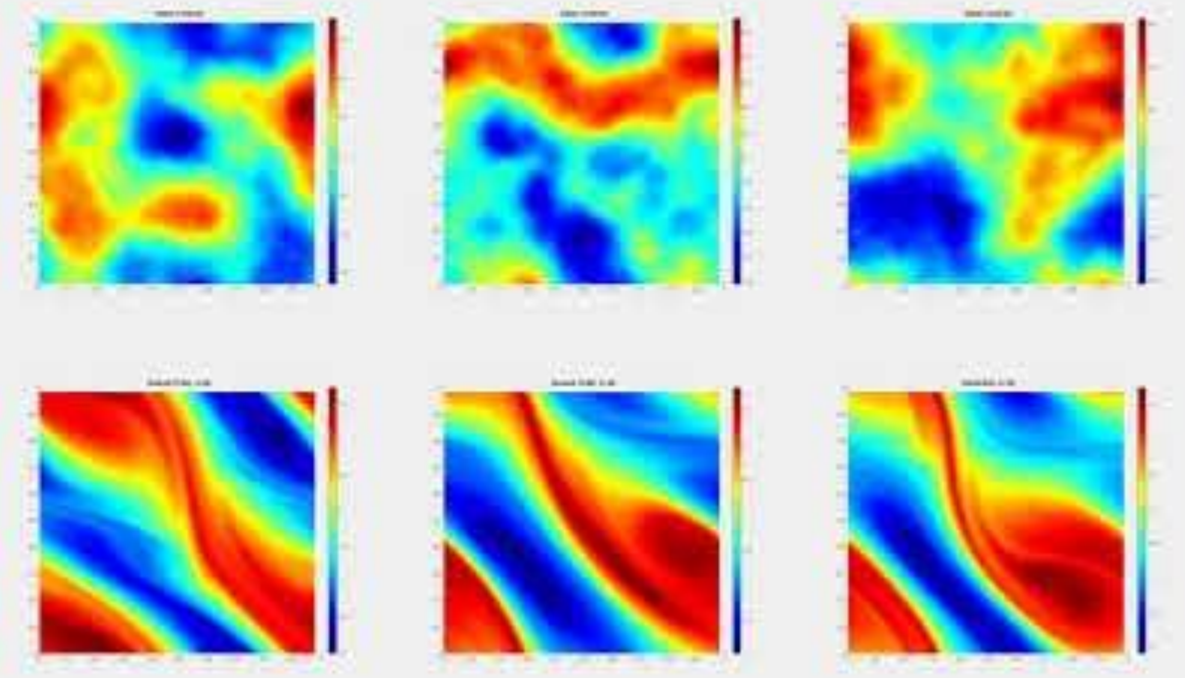
V=1e-3 (Re~1e+3)



V=1e-4 (Re~1e+4)



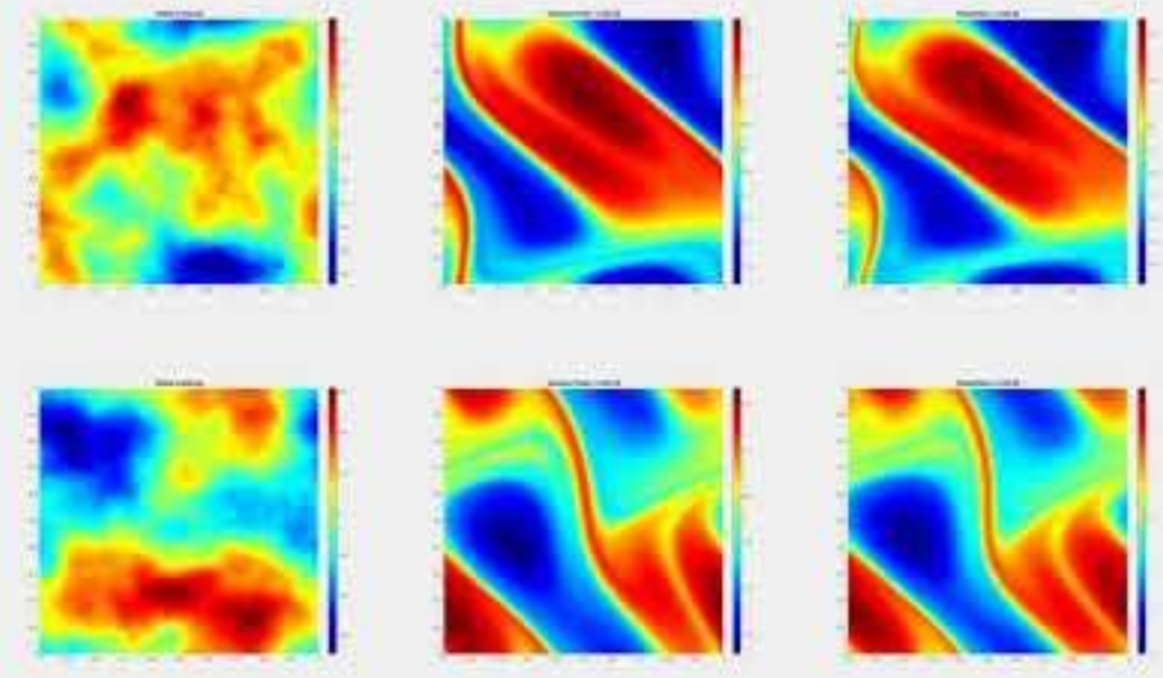
V=1e-5 (Re~1e+5)



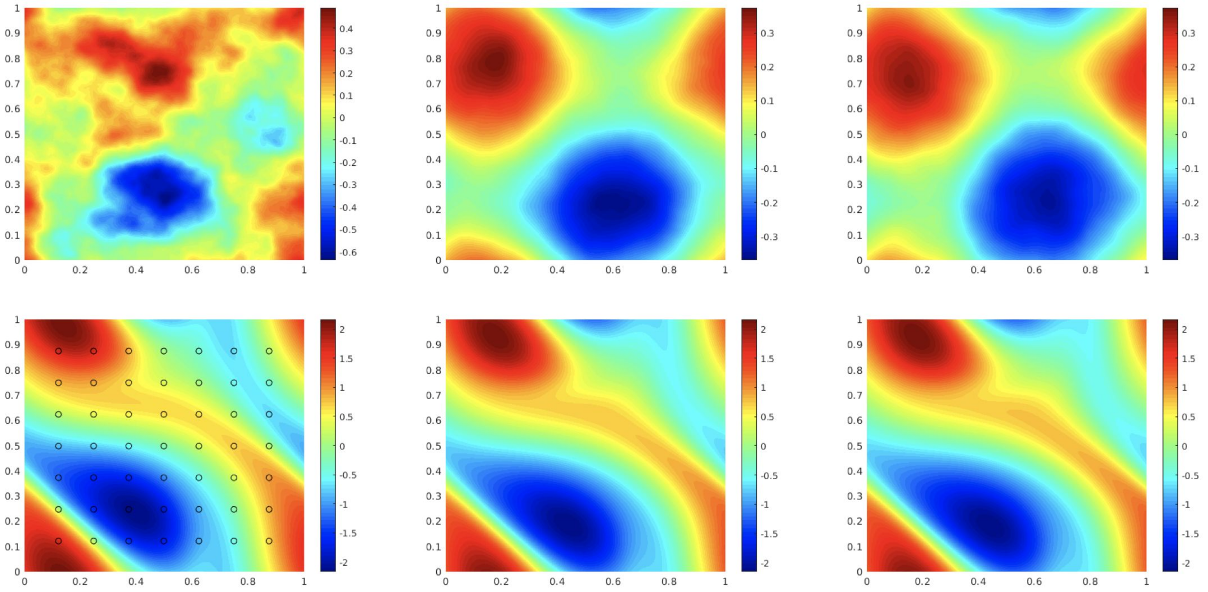
Example 4: Navier-Stokes

Config	**Parameters**	**Time per epoch**	$\nu = 1\text{e}{-3}$ $T = 50$ $N = 1000$	$\nu = 1\text{e}{-4}$ $T = 30$ $N = 1000$	$\nu = 1\text{e}{-4}$ $T = 30$ $N = 10000$	$\nu = 1\text{e}{-5}$ $T = 20$ $N = 1000$
FNO-3D	$6,558,537$	$38.99s$	**0.0086**	0.1918	**0.0820**	0.1893
FNO-2D	$414,517$	$127.80s$	0.0128	**0.1559**	0.0973	**0.1556**
U-Net	$24,950,491$	$48.67s$	0.0245	0.2051	0.1190	0.1982
TF-Net	$7,451,724$	$47.21s$	0.0225	0.2253	0.1168	0.2268
ResNet	$266,641$	$78.47s$	0.0701	0.2871	0.2311	0.2753

V=1e-4, zero-shot super-resolution

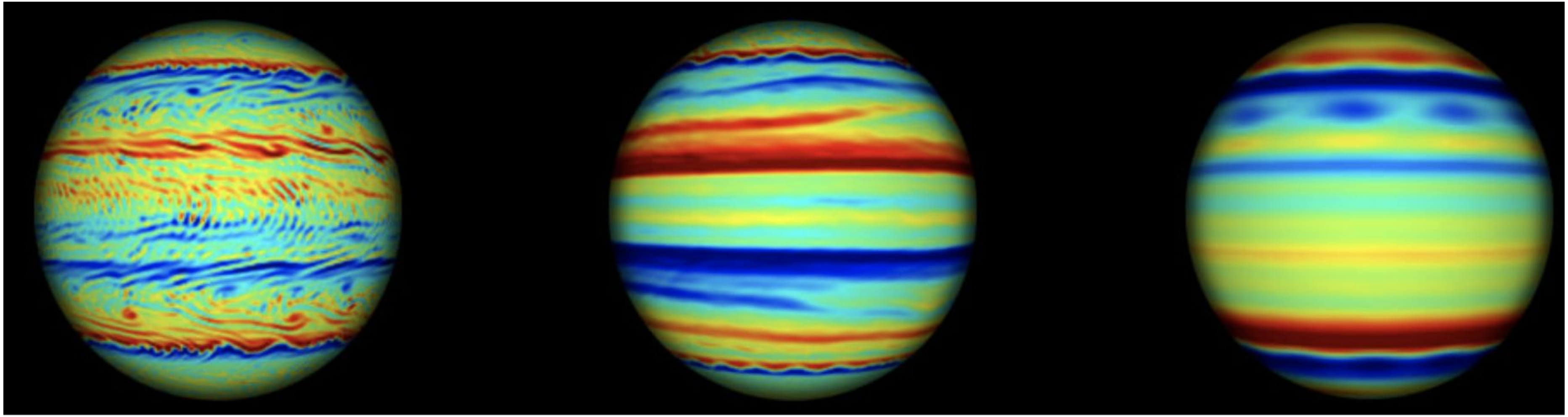


Bayesian inverse problem:



We a MCMC method, sampling initial conditions and evaluating them with the traditional solver and Fourier operator. The Fourier operator takes **0.005s** to evaluate each initial condition, while the traditional solver takes **2.2s**.

6. Future work



Future work

1. Combine with solvers:

F: u(t) -> u(t+delta_t) or F: u|[t1,t2] -> u|[t2,t3]

- For easier case, we can directly do u(0) -> u(50)
- For hard case, smaller delta_t

Augment coarse-grid solver

- Coarser spatial grid
- Larger time-step

Future work

2. Combine with PINNs

- Out a “context grid” (Meshfreeflownet)
- Helps PINNs parametrize the solution

Takeaway

1. Data-driven method: learn the equation
2. Operator-learning: parameterize the mesh-invariant operator
3. Fourier method: efficient for continuous inputs and outputs
4. Results: accurate than other deep learning method, faster than conventional solvers
5. Future work: combine with solvers. Scale up.

Reference

Arxiv:
https://arxiv.org/abs/2003.03485
https://arxiv.org/abs/2006.09535
https://arxiv.org/abs/2010.08895

Code:
https://github.com/zongyi-li/graph-pde
https://github.com/zongyi-li/fourier_neural_operator

Blog posts:
https://zongyi-li.github.io/blog/2020/graph-pde/
https://zongyi-li.github.io/blog/2020/fourier-pde/