Structure-preserving learning of embedded closure models

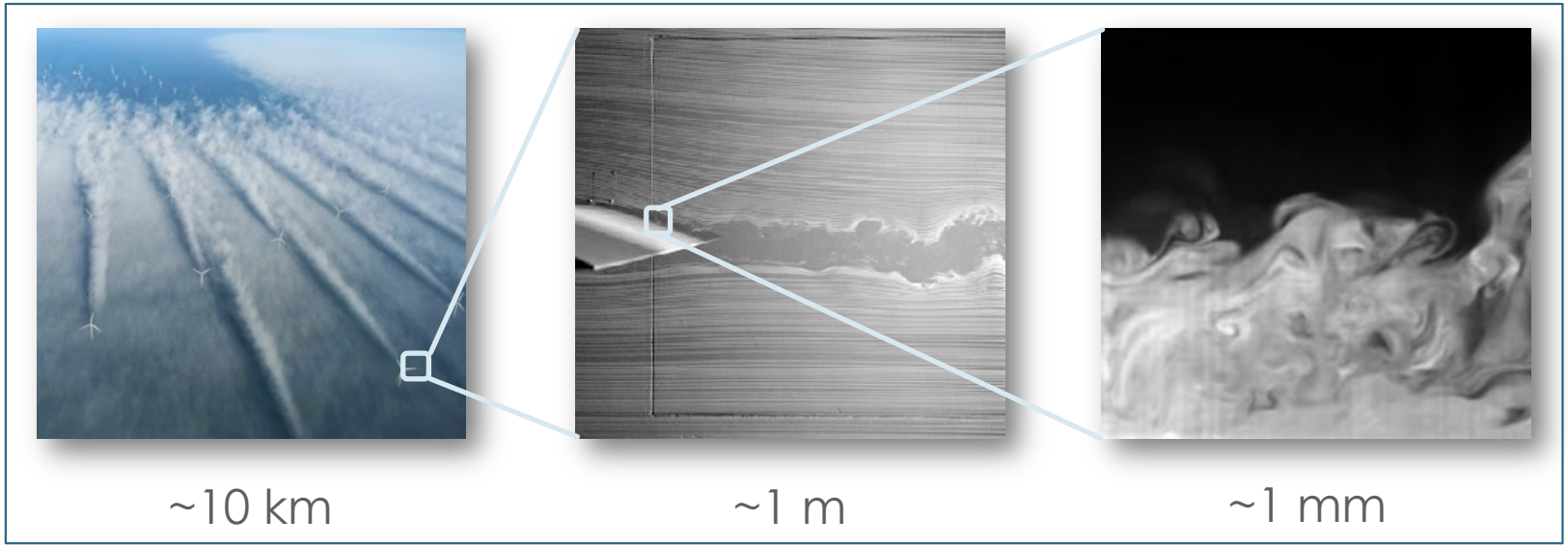
Benjamin Sanderse, Syver Agdestein, Toby van Gastelen, Henrik Rosenberger, Hugo Melchers

6th December 2023

Scientific Machine Learning Workshop, Amsterdam

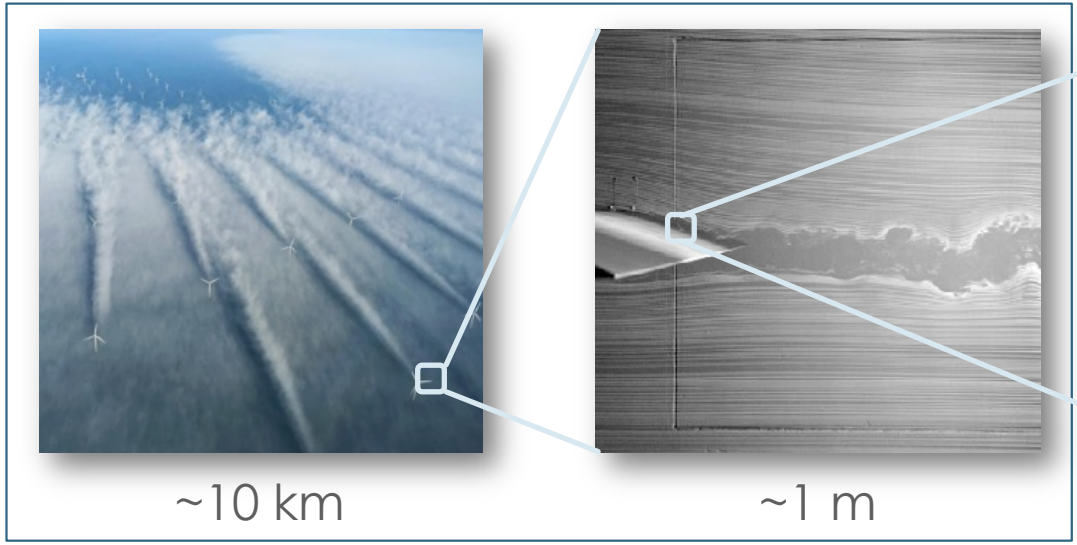
CWI

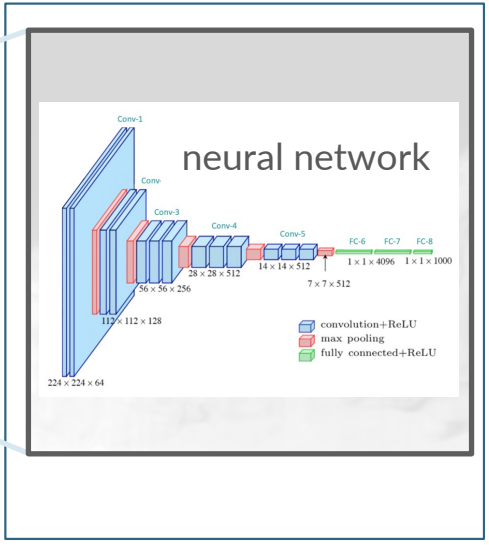
Multiscale problems are omnipresent



Simulating all scales with a computational model is unfeasible

Accurate and stable closure models needed

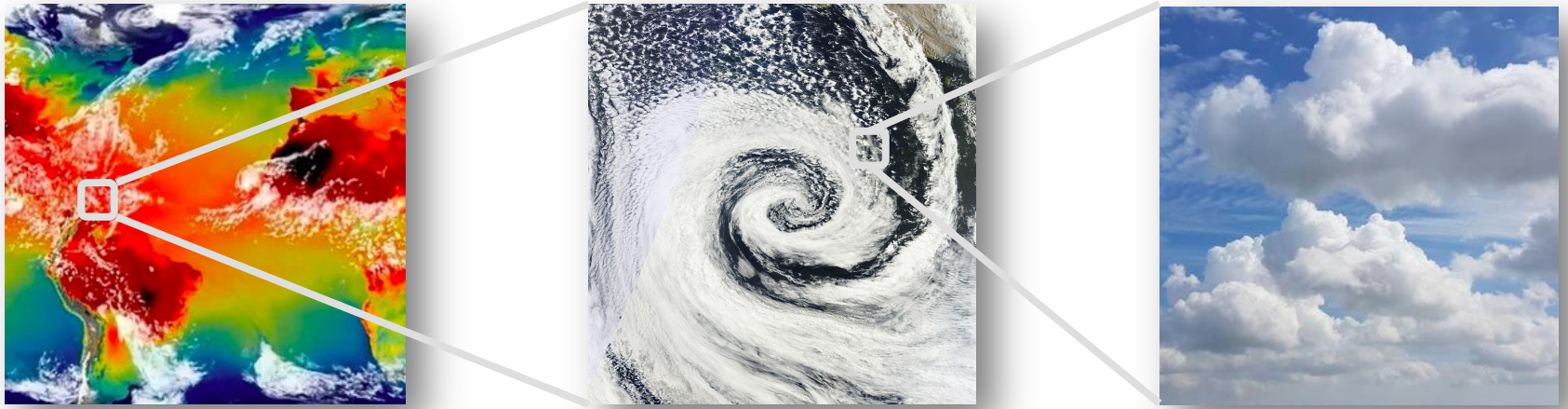




large scales $\bar{u}$

small scales u'

Closure problems occur in many fields

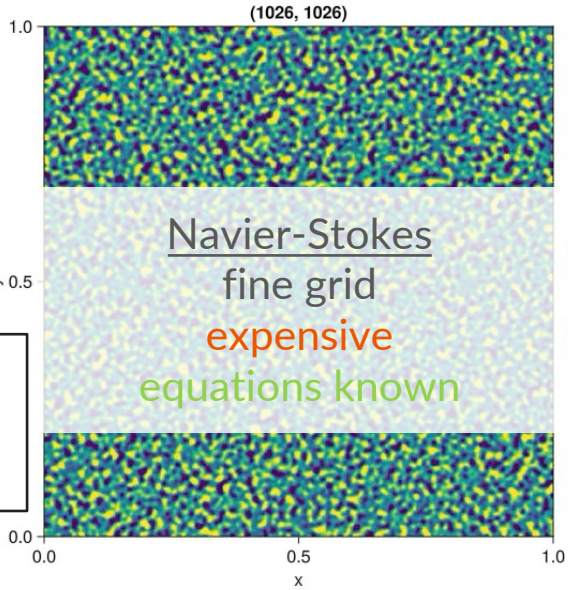


Resolving clouds in climate/weather models: “parameterization”

Filtering the Navier-Stokes equations

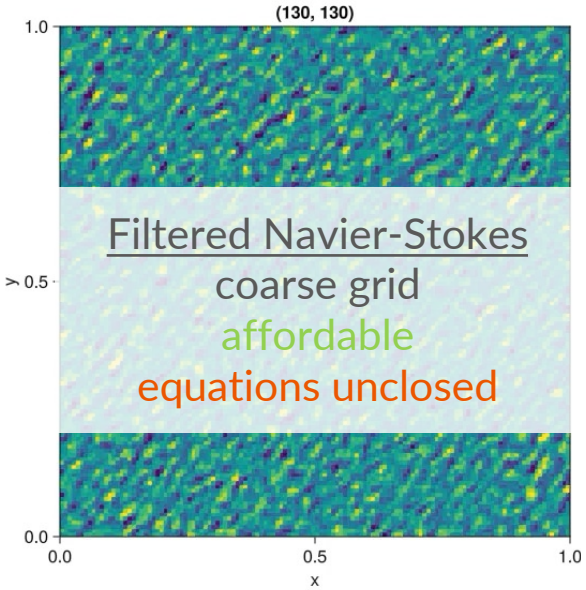
$\boldsymbol{u}$

Direct Numerical Simulation ("reference")



$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{f}(\boldsymbol{u})$$

$\bar{\boldsymbol{u}}$



$$\frac{\partial \bar{\boldsymbol{u}}}{\partial t} = \boldsymbol{f}(\bar{\boldsymbol{u}}) + \mathcal{C}(\bar{\boldsymbol{u}}; \boldsymbol{u})$$

Basics of closure modelling

- Consider PDEs describing many scales, e.g. the Navier-Stokes equations

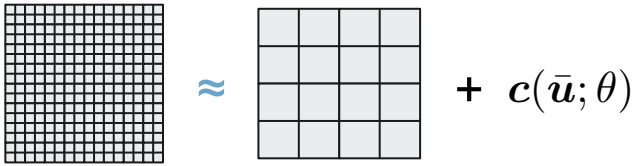
$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{f}(\boldsymbol{u}) \qquad \boldsymbol{f}(\boldsymbol{u}) := -\nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{u}) - \nabla p + \nu \nabla^2 \boldsymbol{u}$$

- NS describes (too) many scales of motion for small viscosity ν
- Reduce range of scales by a **filtering** operation:

$$\bar{\boldsymbol{u}} = \mathcal{A}(\boldsymbol{u}) \qquad \mathcal{A}(\boldsymbol{u}) = \int \boldsymbol{u}(\xi, t) G(x, \xi) \mathrm{d}\xi \qquad \boldsymbol{u}' = \boldsymbol{u} - \bar{\boldsymbol{u}}$$

- **Aim**: use coarser meshes and larger time steps when solving for $\bar{\boldsymbol{u}}$

Basics of closure modelling



- Problem: filter and PDE operator **do not commute**
- Art: find a **closure model** with parameters θ s.t.

$$\boldsymbol{c}(\bar{\boldsymbol{u}};\theta) \approx \mathcal{C}(\bar{\boldsymbol{u}},\boldsymbol{u}) = \overline{\nabla\cdot(\boldsymbol{u}\otimes\boldsymbol{u})} - \nabla\cdot(\bar{\boldsymbol{u}}\otimes\bar{\boldsymbol{u}})$$

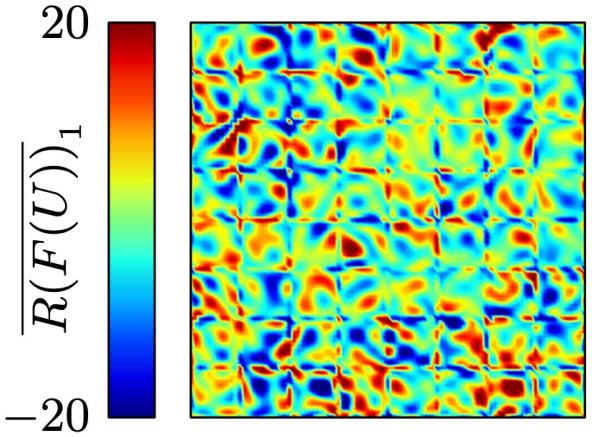
- Finding $\boldsymbol{c}(\bar{\boldsymbol{u}};\theta)$ is an inverse problem
- Common form: $\dfrac{\partial\bar{\boldsymbol{u}}}{\partial t} = \boldsymbol{f}(\bar{\boldsymbol{u}}) + \boldsymbol{c}(\bar{\boldsymbol{u}};\theta)$

Idea: use neural networks as closure model

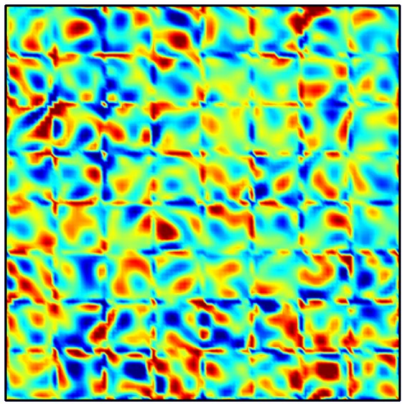
$$\boldsymbol{c}(\bar{\boldsymbol{u}};\theta) = \mathrm{NN}(\bar{\boldsymbol{u}};\theta)$$

$$\frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = f(\bar{\mathbf{u}}) + \mathrm{NN}(\bar{\mathbf{u}};\theta)$$

model discovery
(inverse problem)



$\mathcal{C}(\bar{\boldsymbol{u}}, \boldsymbol{u})$



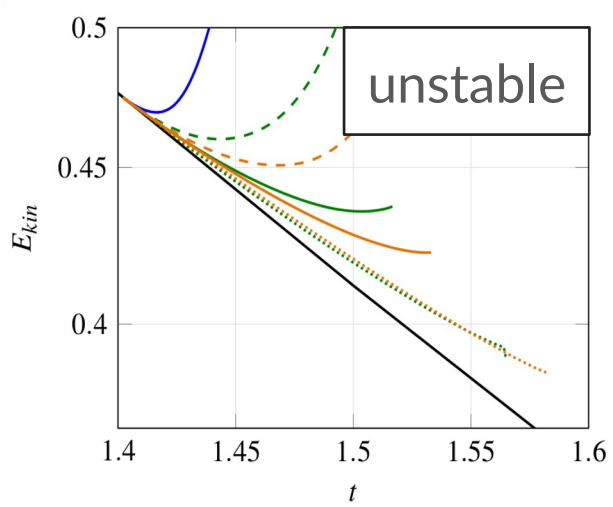
$\boldsymbol{c}(\bar{\boldsymbol{u}};\theta) = \mathrm{NN}(\bar{\boldsymbol{u}};\theta)$

Kurz & Beck, “A machine learning framework for LES closure terms”, 2021

Issue: NNs destabilize the dynamical system

- NN matches closure term, but **solution blows up**
- "**Model-data inconsistency**" (mismatch training environment and prediction environment)

Open questions:
- Stability
- Trainability, generalizability, interpretability
- Discrete or continuous
- Stochastic or deterministic



Our approach: "preserve structure" + "embedded learning"

Examples of preserving structure

- ODE formulation (“neural ODE”)
- Closure model form (“neural closure model”)
- Conservation form
- Translation invariance
- **Energy conservation**

$$\frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = \mathrm{NN}(\bar{\mathbf{u}};\theta)$$

$$\frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = f(\bar{\mathbf{u}}) + \mathrm{NN}(\bar{\mathbf{u}};\theta)$$

$$\frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = f(\bar{\mathbf{u}}) + \nabla \cdot \mathrm{NN}(\bar{\mathbf{u}};\theta)$$

CNN architecture

Energy conservation implies stability

- Many PDEs possess **secondary conservation laws** (energy, entropy), which give a **stability bound**

- **Example: Navier-Stokes**

$$\nabla \cdot \boldsymbol{u} = 0$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{u}) = -\nabla p + \nu \nabla^2 \boldsymbol{u}$$

$$\Longrightarrow \quad \frac{\mathrm{d}E}{\mathrm{d}t} = -\nu \int_\Omega \nabla \boldsymbol{u} : \nabla \boldsymbol{u} \, \mathrm{d}\Omega$$

$$E := \frac{1}{2} \int \boldsymbol{u} \cdot \boldsymbol{u} \, \mathrm{d}\Omega$$

Idea: use energy conservation to construct stable closure models

Energy conservation implies stability

- Many PDEs possess **secondary conservation laws** (energy, entropy), which give a **stability bound**

- **Example: Korteweg – de Vries**

$$\frac{\partial u}{\partial t} + 3\frac{\partial u^2}{\partial x} = -\frac{\partial^3 u}{\partial x^3} \quad \Longrightarrow \quad \frac{\mathrm{d}E}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\underbrace{\frac{1}{2}\int_\Omega u^2 d\Omega}_{=:E} = 0$$

$$E := \frac{1}{2}\int u^2 \,\mathrm{d}\Omega$$

Idea: use energy conservation to construct stable closure models

Korteweg - de Vries equation

- Shallow water waves, solitons:

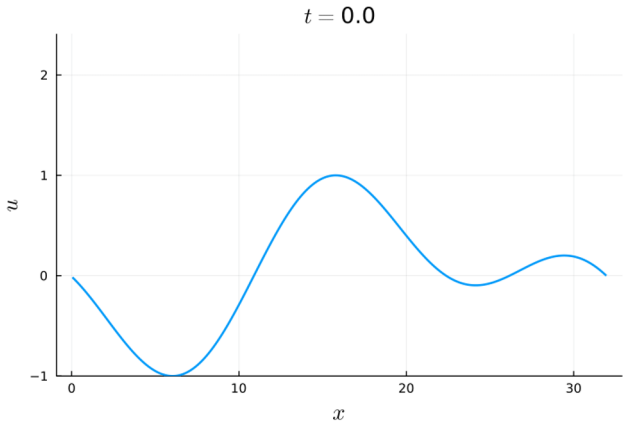
$$\frac{\partial u}{\partial t} + 3\frac{\partial u^2}{\partial x} = -\frac{\partial^3 u}{\partial x^3}$$

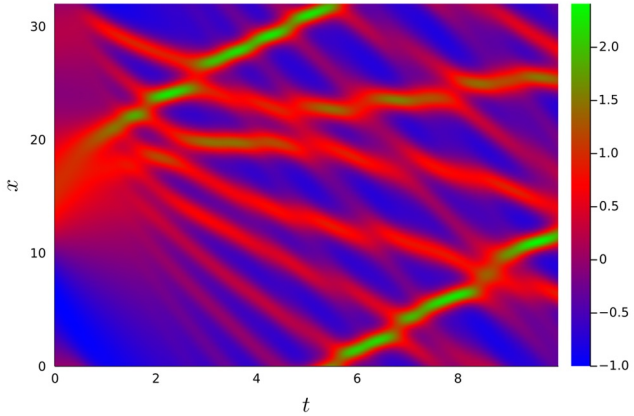
- Energy conservation (periodic BCs):

$$\frac{\mathrm{d}E}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\underbrace{\frac{1}{2}\int_\Omega u^2 d\Omega}_{=:E} = 0$$

- Discretized using skew-symmetric scheme:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = -3\mathbf{G}(\mathbf{u}) - \mathbf{D}_3\mathbf{u} \qquad (\mathbf{u}, \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}) = 0$$





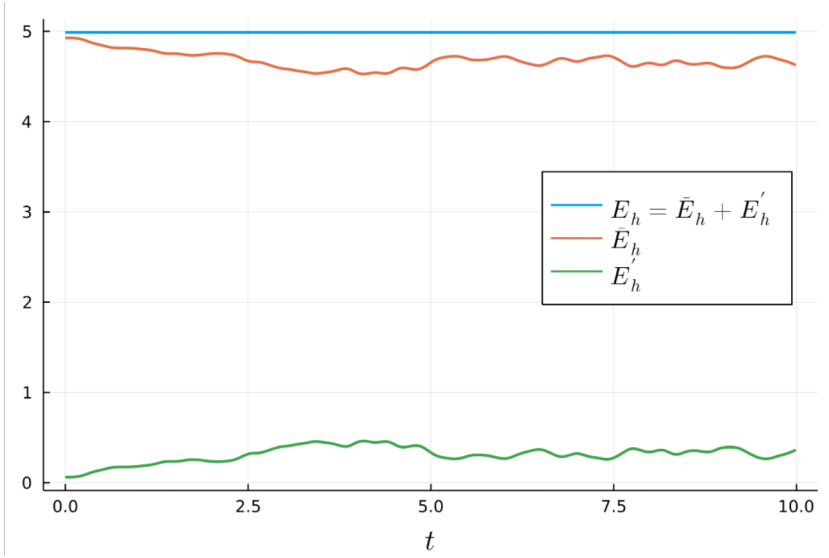
Energy decomposition

$$\frac{\mathrm{d}E_h}{\mathrm{d}t} = \frac{\mathrm{d}\bar{E}_h(\bar{\mathbf{u}})}{\mathrm{d}t} + \frac{\mathrm{d}E'_h(\mathbf{u}')}{\mathrm{d}t} = 0$$

- Decompose the energy:

$$E_h = \underbrace{\frac{1}{2}(\bar{\mathbf{u}}, \bar{\mathbf{u}})_\Omega}_{=:\bar{E}_h} + \underbrace{\frac{1}{2}(\mathbf{u}', \mathbf{u}')_\omega}_{=:E'_h}$$

- Energy of $\bar{\mathbf{u}}$ is not conserved!
- -> Need information about $\mathbf{u}'$ in order to use energy conservation



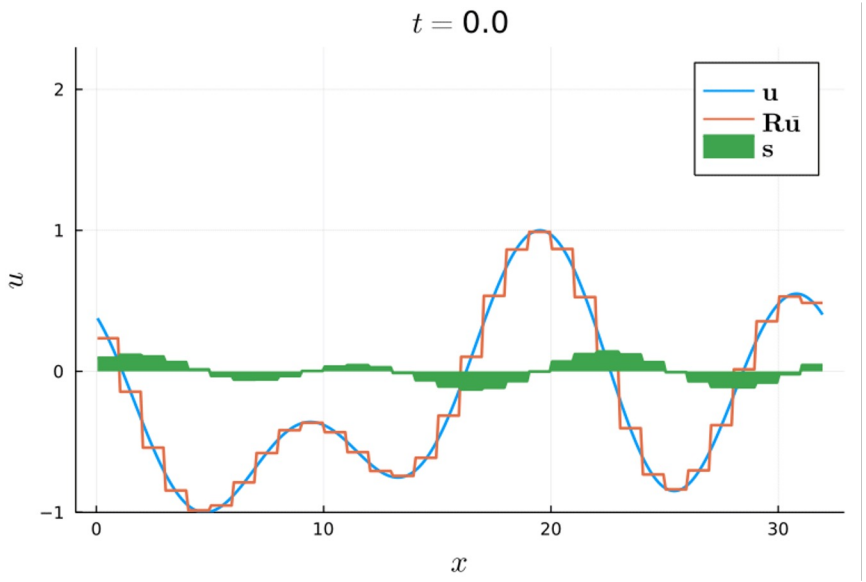
Compress the small scale variables

$$\mathbf{s} = \mathbf{T}\mathbf{u}'$$

T is learned such that

$$\frac{1}{2}(\mathbf{s},\mathbf{s}) \approx \frac{1}{2}(\mathbf{u}',\mathbf{u}')$$

coarse grid	fine grid
affordable	expensive



compressed subgrid variable identifies sharp gradients

New energy-conserving closure model

Filtered dynamics

Compressed small scales

Extended neural closure model

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\bar{\mathbf{u}}\\ \mathbf{s}\end{bmatrix} = \begin{bmatrix}f(\bar{\mathbf{u}})\\ \mathbf{0}\end{bmatrix} + \begin{bmatrix}c_u(\bar{\mathbf{u}},\mathbf{s};\theta_u)\\ c_s(\bar{\mathbf{u}},\mathbf{s};\theta_s)\end{bmatrix}$$

We propose: energy-conserving form

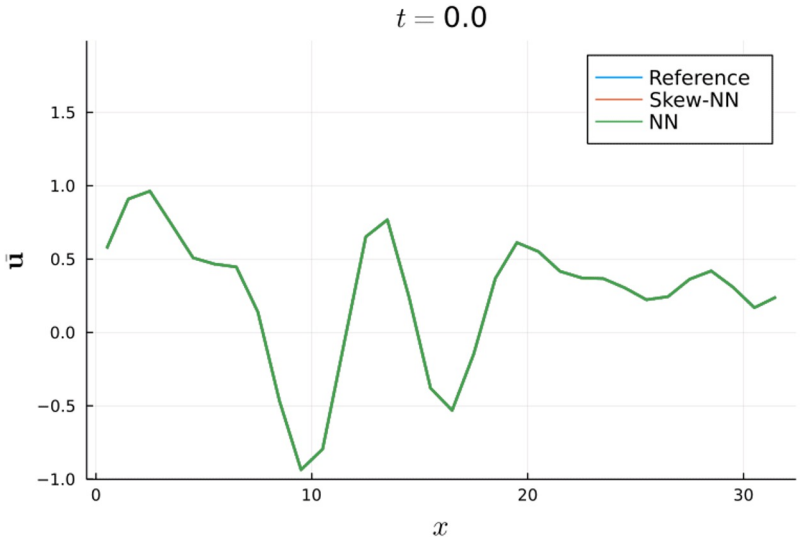
$$\begin{bmatrix}c_u(\bar{\mathbf{u}},\mathbf{s};\theta_u)\\ c_s(\bar{\mathbf{u}},\mathbf{s};\theta_s)\end{bmatrix} = \mathcal{K}(\bar{\mathbf{u}},\mathbf{s};\boldsymbol{\Theta})\begin{bmatrix}\bar{\mathbf{u}}\\ \mathbf{s}\end{bmatrix}$$

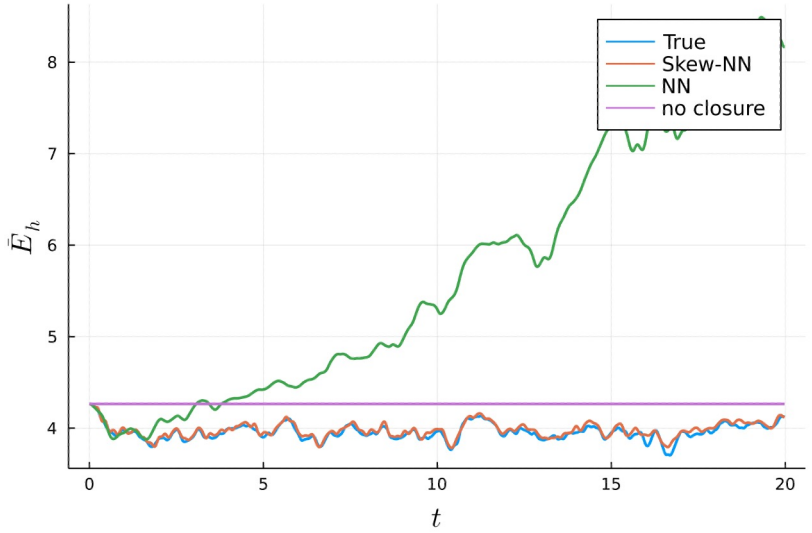
Skew-symmetric matrix $\mathcal{K}$ leads to exact energy conservation

$$\frac{\mathrm{d}\bar{E}_h(\bar{\mathbf{u}})}{\mathrm{d}t} + \frac{1}{2}\frac{\mathrm{d}(\mathbf{s},\mathbf{s})_\omega}{\mathrm{d}t} = 0$$

New closure model improves quality + stability

- Trained on different initial conditions, tested on unseen initial conditions
- Reduction from N = 600 to N = 30
- Compare to standard CNN



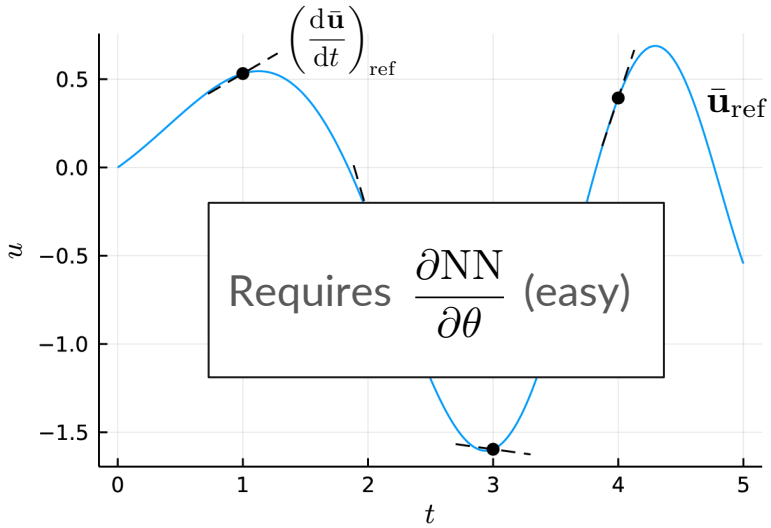


How to train neural closure models?

What is the loss function?

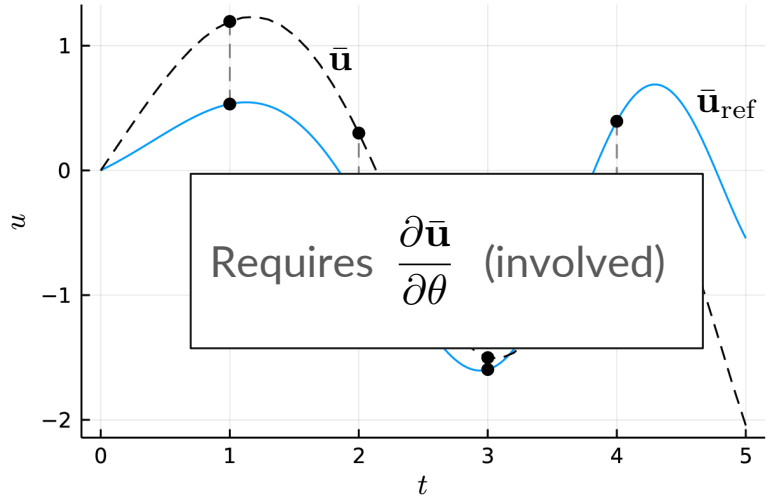
$$\frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = f(\bar{\mathbf{u}}) + \mathrm{NN}(\bar{\mathbf{u}};\theta)$$

Derivative fitting (a-priori)



$$\mathrm{Loss} = \left\| \left(\frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t}\right)_{\mathrm{ref}} - f(\bar{\mathbf{u}}_{\mathrm{ref}}) - \mathrm{NN}(\bar{\mathbf{u}}_{\mathrm{ref}};\theta) \right\|^2$$

Trajectory fitting (a-posteriori)



$$\mathrm{Loss} = \sum_{i=1}^{N_t} \left\| \bar{\mathbf{u}}_{\mathrm{ref}}(t_i) - \bar{\mathbf{u}}(t_i) \right\|^2, \text{ where } \frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = f(\bar{\mathbf{u}}) + \mathrm{NN}(\bar{\mathbf{u}};\theta)$$

$$\text{Loss} = \sum_{i=1}^{N_t} \|\bar{\mathbf{u}}_{\text{ref}}(t_i) - \bar{\mathbf{u}}(t_i)\|^2, \text{ where } \frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = f(\bar{\mathbf{u}}) + \text{NN}(\bar{\mathbf{u}}; \theta)$$

Trajectory fitting – computing $\frac{\mathrm{dLoss}}{\mathrm{d}\theta}$

1. **Discretise-then-optimise (DtO):**
   - Need differentiable solver
   - Preferably explicit

2. **Optimise-then-discretise (OtD):**
   - Solve adjoint equations[1]
   - Here: interpolating adjoint
   - Need dense output

$$\begin{cases} \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{y}^T = -\mathbf{y}^T \frac{\partial}{\partial \bar{\mathbf{u}}}(f(\bar{\mathbf{u}}) + \text{NN}(\bar{\mathbf{u}}; \theta)) \\ \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{z}^T = -\mathbf{y}^T \frac{\partial}{\partial \theta}(f(\bar{\mathbf{u}}) + \text{NN}(\bar{\mathbf{u}}; \theta)) \\ \frac{\mathrm{dLoss}}{\mathrm{d}\theta} = \mathbf{z}(0) \end{cases}$$

[1]Chen et al, Neural ordinary differential equations, NeurIPS 2018

Derivative fitting vs. trajectory fitting

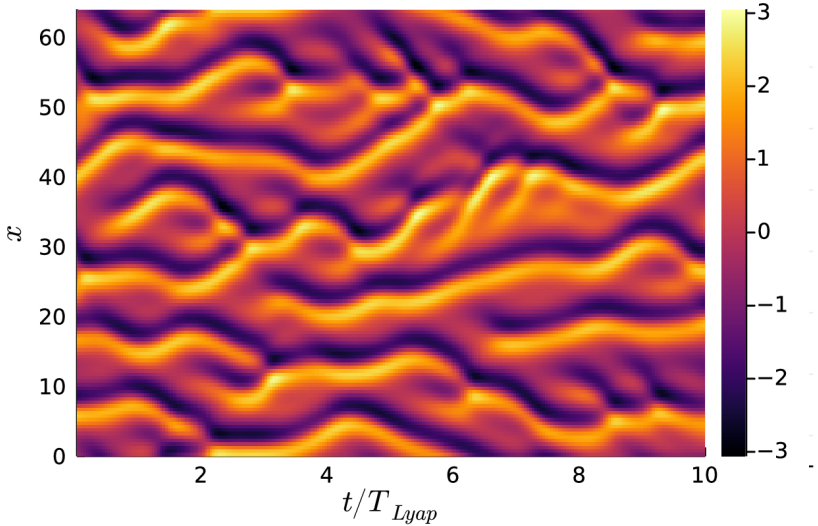
	Derivative fitting	Trajectory fitting	
		DtO	OtD
Differentiability required	NN	NN, f, ODE solver	NN, f
Accuracy of loss function gradients	Exact	Exact	Approximate
Learns long-term accuracy	No	Yes	Yes
Requires time-derivatives of training data	Yes	No	No
Computational cost	Low	High	High

Kuramoto-Sivashinsky equation

$$\frac{\partial u}{\partial t} = -\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right) - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4}$$

$$\frac{\mathrm{d}\bar{\mathbf{u}}}{\mathrm{d}t} = f(\bar{\mathbf{u}}) + \nabla \cdot \mathrm{NN}(\bar{\mathbf{u}}; \theta)$$

- **Chaotic:**
  - Trajectory lengths not too large
- **Stiff**:
  - OtD: impl/expl RK, KenCarp47[1]
  - DtO: expl ETDRK4 in Fourier domain[2]
- Filter **W**:
  - downsampling 1024 => 128



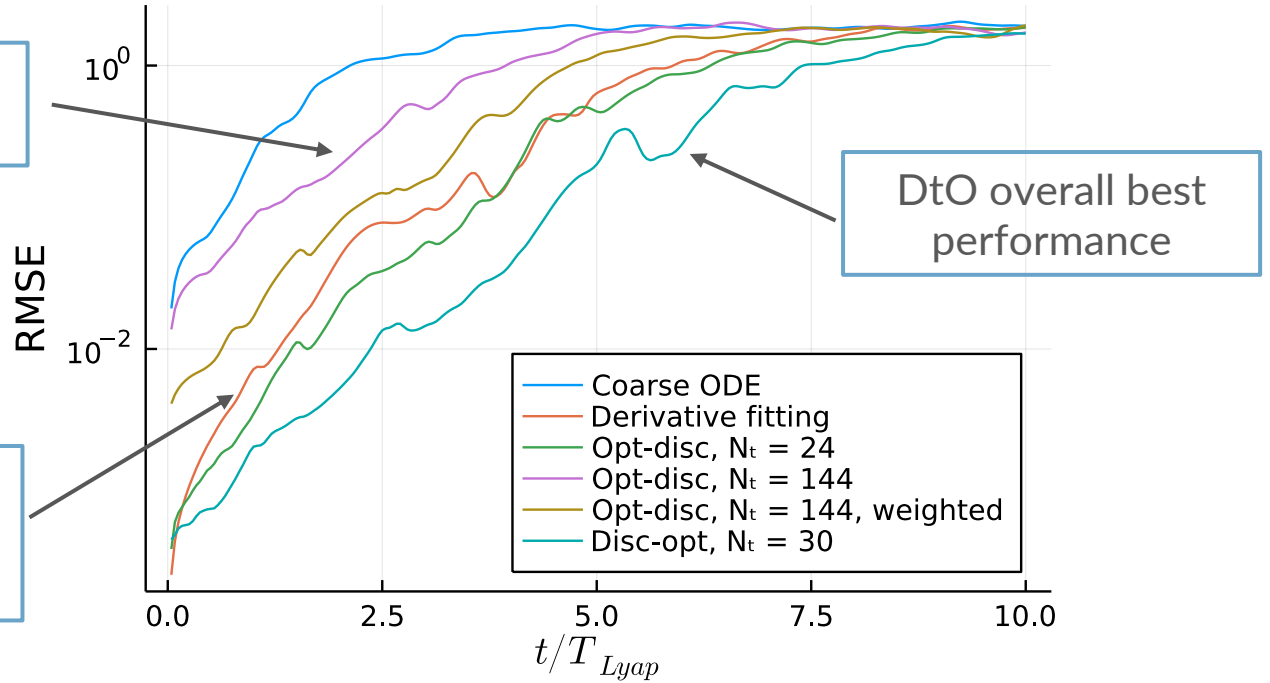
[1]Kennedy and Carpenter, Higher-order additive Runge-Kutta schemes for ODEs, Applied Numerical Mathematics, 2019.
[2]Kassam & Trefethen, Fourth-order time-stepping for stiff PDEs. SIAM Journal on Scientific Computing, 2005

DtO outperforms OtD and derivative fitting

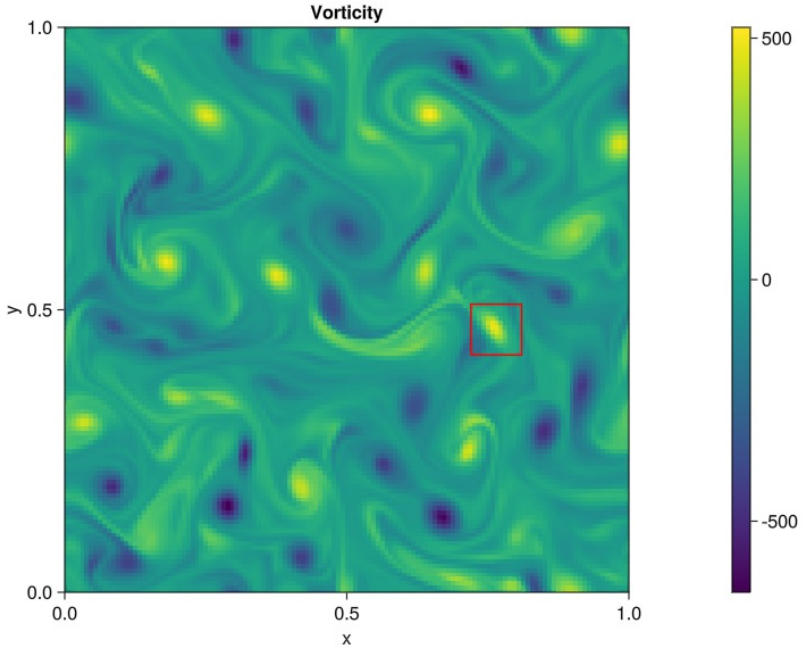
OtD sensitive to longer intervals

Derivative fitting OK, but testcase dependent

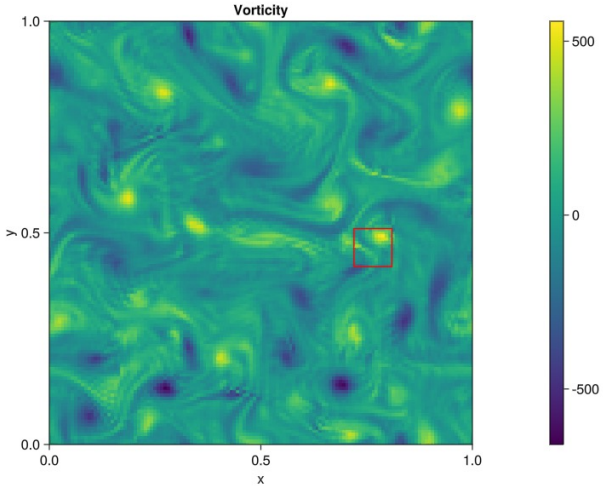
DtO overall best performance



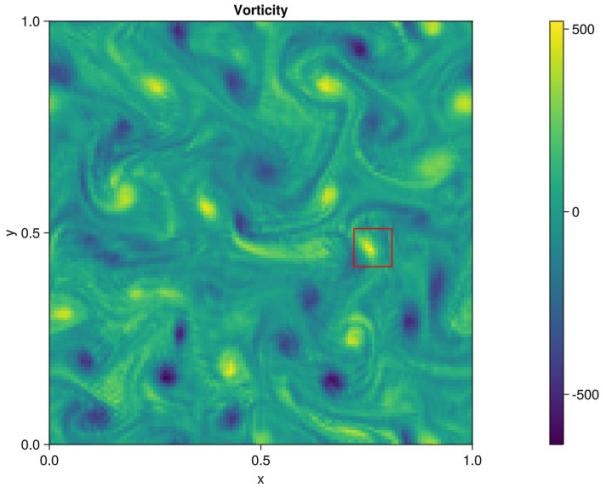
Example results (2D)



Filtered Navier-Stokes

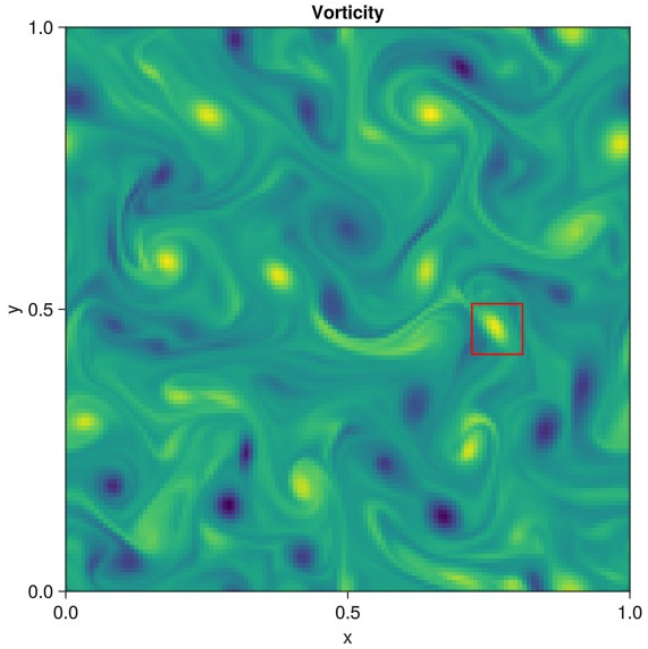


No closure model

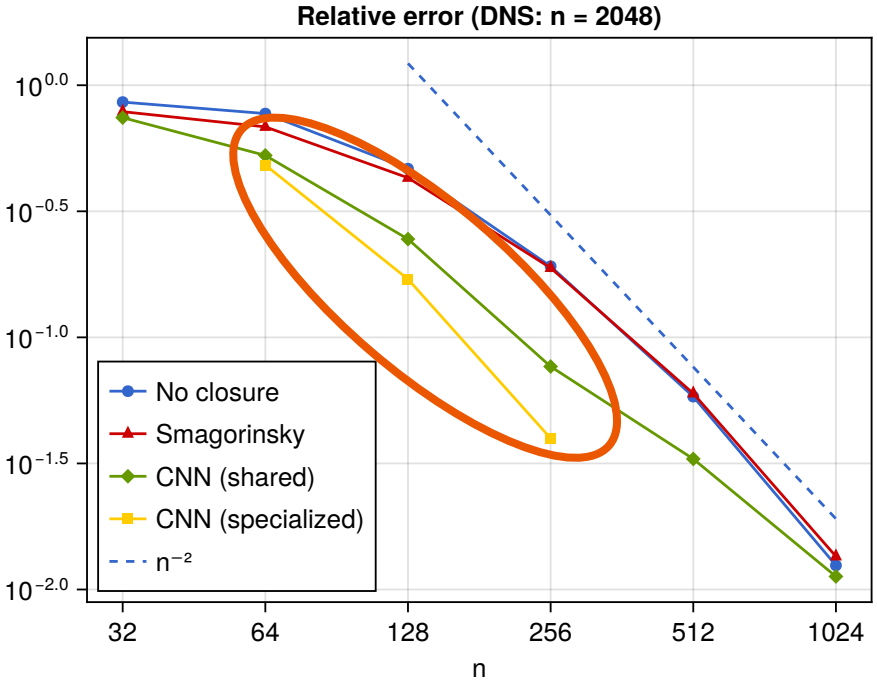


Fourier Neural Operator

First convergence results on LES with NNs

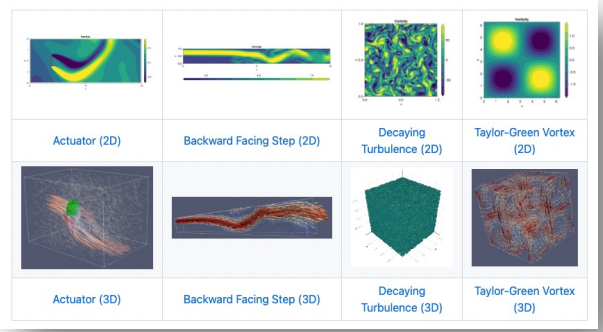


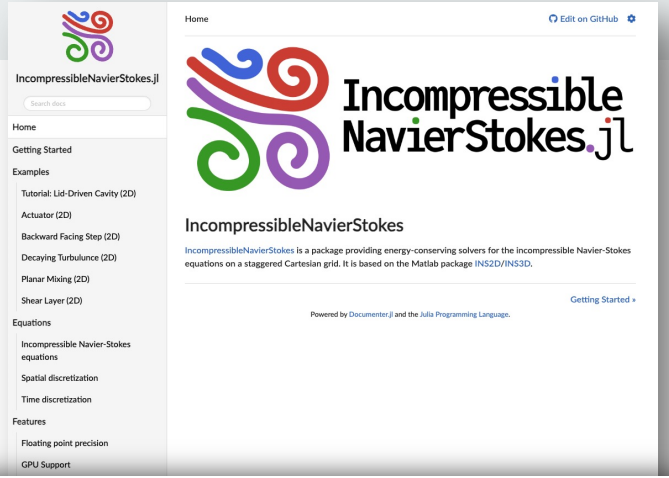
Filtered Navier-Stokes

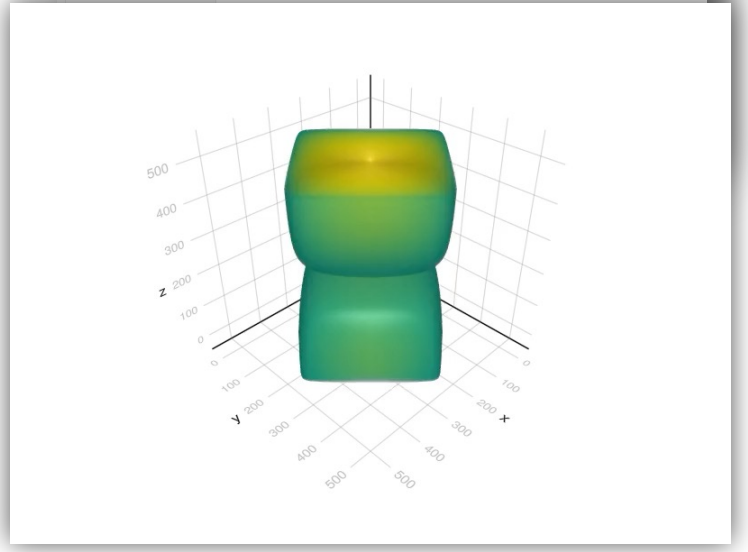


Software: fluid flow simulator

- Incompressible 2D/3D Navier-Stokes code written in Julia
- github.com/agdestein/IncompressibleNavierStokes.jl
- Automatic differentiation with Zygote
- CPU and GPU implementation

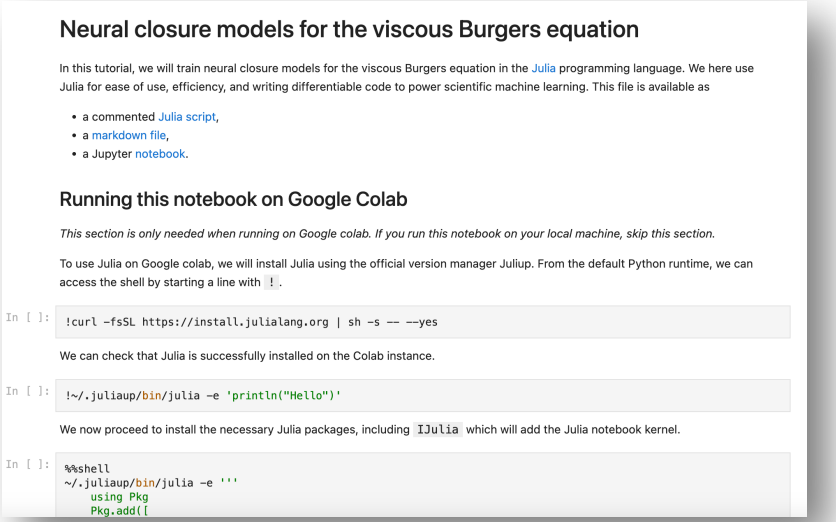






Software: neural closure

- Feel free to test our closure models
- Tutorial available at github.com/agdestein/NeuralClosure
- Can be directly run on Google Colab



Neural closure models for the viscous Burgers equation

In this tutorial, we will train neural closure models for the viscous Burgers equation in the Julia programming language. We here use Julia for ease of use, efficiency, and writing differentiable code to power scientific machine learning. This file is available as

- a commented Julia script,
- a markdown file,
- a Jupyter notebook.

Running this notebook on Google Colab

This section is only needed when running on Google colab. If you run this notebook on your local machine, skip this section.

To use Julia on Google colab, we will install Julia using the official version manager Juliup. From the default Python runtime, we can access the shell by starting a line with `!`.

In [ ]:
```
!curl -fsSL https://install.julialang.org | sh -s -- --yes
```

We can check that Julia is successfully installed on the Colab instance.

In [ ]:
```
!~/.juliaup/bin/julia -e 'println("Hello")'
```

We now proceed to install the necessary Julia packages, including `IJulia` which will add the Julia notebook kernel.

In [ ]:
```
%%shell
~/.juliaup/bin/julia -e '''
    using Pkg
    Pkg.add([
```

Conclusions

- **Multi-scale modelling leads to closure problems**
  Neural networks are a good candidate, but stability is a main issue

- **Preserve structure -> Stability**
  - Non-linear stability with energy-conserving methods

- **Embedded learning -> Accuracy**
  - Discretise-then-optimise with differentiable solvers