AQTIVATE

# Automatic Differentiation and Machine Learning for Numerical Simulations

*AQTIVATE ML Workshop*

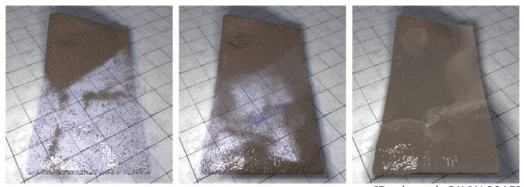
**Kiwon Um**

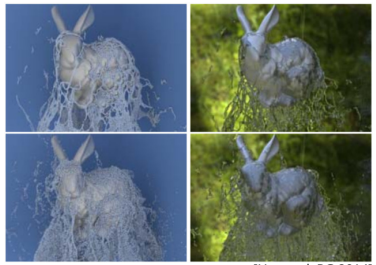
TELECOM Paris

IP PARIS

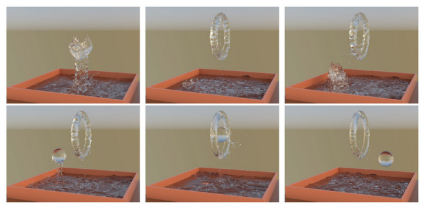
# Personal background

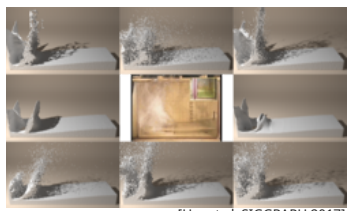
- Computer science and engineering


[Baek et al. CAVW 2015]


[Um et al. PG 2014]

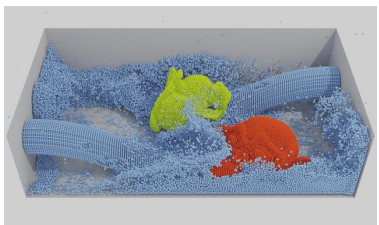
- Physics-based simulations, particularly for computer graphics


[Um et al. CASA 2017]


[Um et al. SIGGRAPH 2017]

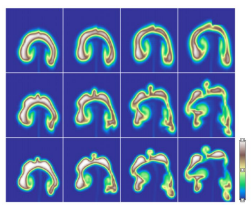
- Fluid simulations

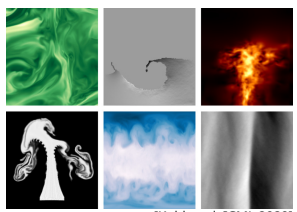

[Kee et al. SIGGRAPH 2021]

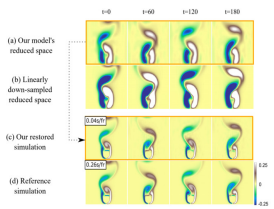

[Kee et al. EG 2023]

- Soft-body simulations


[Um et al. SCA 2018]


[Um et al. NeurIPS 2020]


[Kohl et al. ICML 2020]



[Paliard et al. VMV 2023]

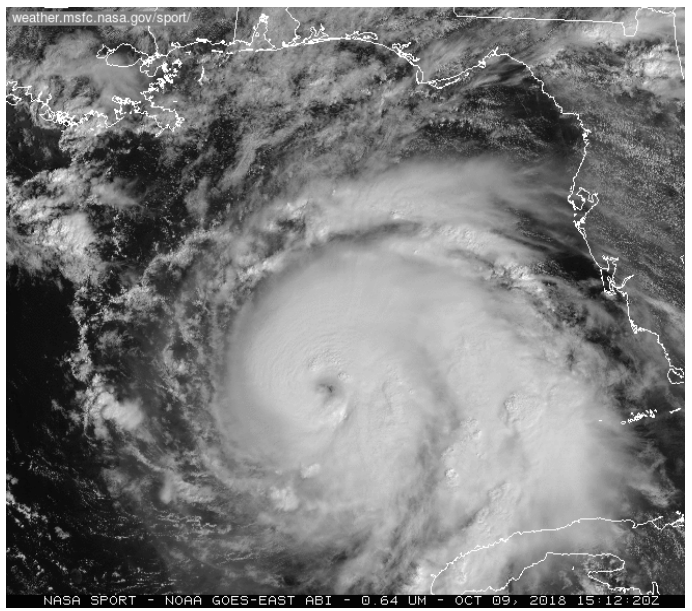
- Deep learning

# My research interests

- Understanding natural phenomena and reproducing them
- Modeling the dynamics, e.g., using PDEs
- Solving PDEs - *numerical or learning-based or hybrid methods*

[Wikipedia]

[Wikipedia]




[NOAA/ NASA MSFC, SPoRT]

# In this talk ...

# Automatic differentiation

- A technical term
  - Refer to a specific family of techniques
  - Compute derivatives through the accumulation of values
  - Execute code to generate numerical **derivative evaluations** rather than **derivative expressions**
- Shortly, *"Autodiff"*
- Also called *"Algorithmic differentiation"*
- Versus
  - Manual differentiation
  - Numerical differentiation
  - Symbolic differentiation
- ... a nice survey
  - [2018, Baydin et al., Automatic differentiation in machine learning: a survey]

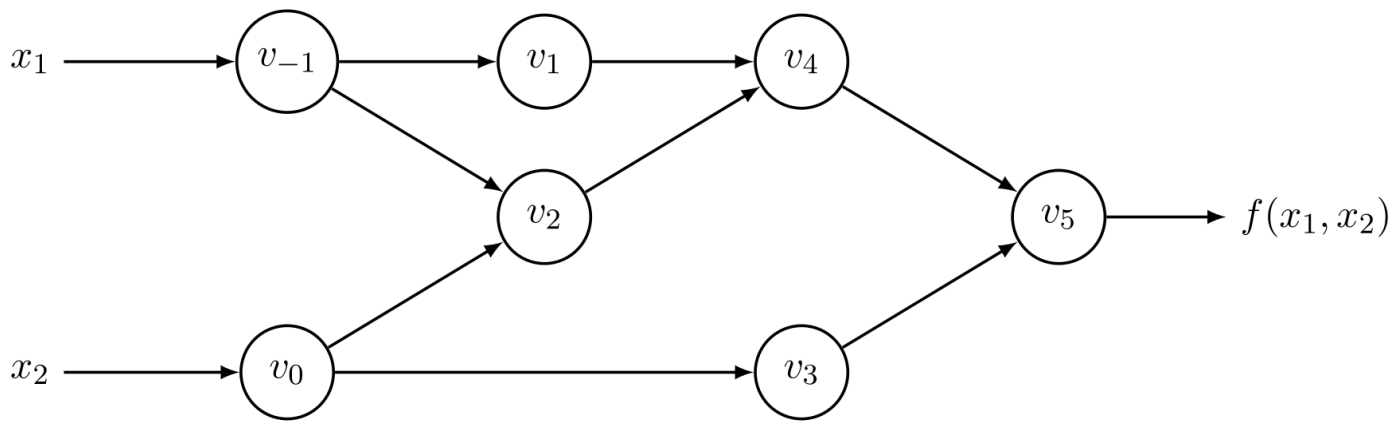
# Forward mode

- An example function:

$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) = y$$

- Want to evaluate $\dot{y}$ for **a given input**, e.g., $(x_1 = 2, x_2 = 5)$

- Computational graph
  - Trace of elementary operations

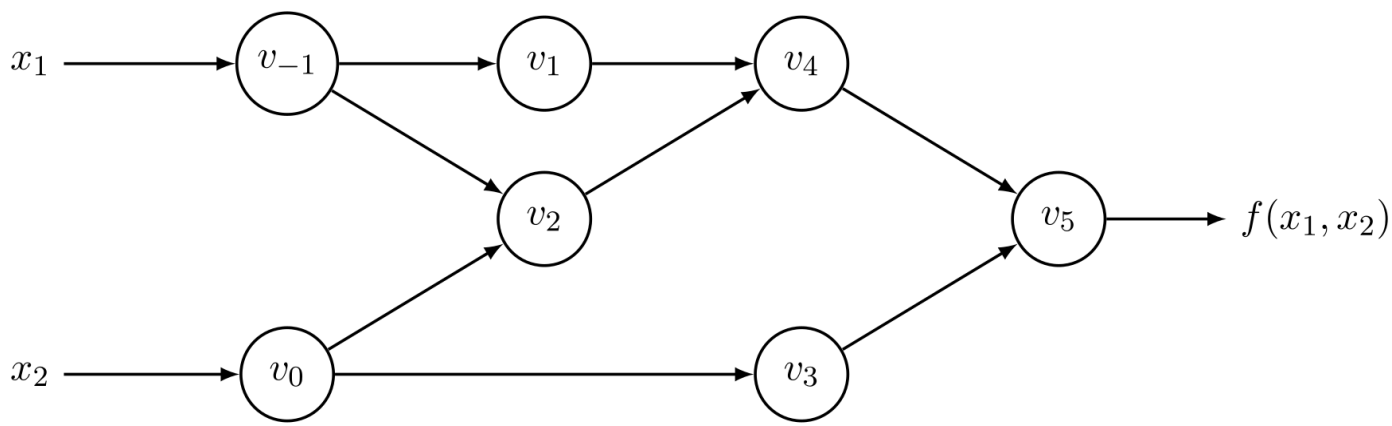


# Forward mode (cont'd)

- Forward primal and tangent (derivative) traces for $\partial y / \partial x_1$

| | | | |
|---|---|---|---|
| $v_{-1}$ | $= x_1$ | $= 2$ | |
| $v_0$ | $= x_2$ | $= 5$ | |
| $v_1$ | $= \ln(v_{-1})$ | $= \ln(2)$ | $= 0.693$ |
| $v_2$ | $= v_{-1} \times v_0$ | $= 2 \times 5$ | $= 10$ |
| $v_3$ | $= \sin(v_0)$ | $= \sin(5)$ | $= -0.959$ |
| $v_4$ | $= v_1 + v_2$ | $= 0.693 + 10$ | $= 10.693$ |
| $v_5$ | $= v_4 - v_3$ | $= 10.693 + 0.959$ | $= 11.652$ |
| $y$ | $= v_5$ | $= 11.652$ | |

| | | | |
|---|---|---|---|
| $\dot{v}_{-1}$ | $= \dot{x}_1$ | $= 1$ | |
| $\dot{v}_0$ | $= \dot{x}_2$ | $= 0$ | |
| $\dot{v}_1$ | $= \dot{v}_{-1} / v_{-1}$ | $= 1/2$ | $= 0.5$ |
| $\dot{v}_2$ | $= \dot{v}_{-1} \times v_0 + v_{-1} \times \dot{v}_0$ | $= 1 \times 5 + 0 \times 2$ | $= 5$ |
| $\dot{v}_3$ | $= \dot{v}_0 \times \cos v_0$ | $= 0 \times \cos(5)$ | $= 0$ |
| $\dot{v}_4$ | $= \dot{v}_1 + \dot{v}_2$ | $= 0.5 + 5$ | $= 5.5$ |
| $\dot{v}_5$ | $= \dot{v}_4 - \dot{v}_3$ | $= 5.5 - 0$ | $= 5.5$ |
| $\dot{y}$ | $= \dot{v}_5$ | $= 5.5$ | |



# Forward mode (cont'd)

- Simple to evaluate
- Can be generalized to computing the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$
  - Require $n$ evaluations for each independent variable $x_i$
- In general, when $n < m$,
  - Very efficient to evaluate the derivative
- Unfortunately, however,
  - It's the other way around for machine learning.
  - I.e., we need to evaluate the derivative *with respect to* a lot of trainable parameters.

# Reverse mode

- Also called *adjoint* or *cotangent linear* mode
- Trace the **sensitivity** of an output $y_j$ with respect to changes in intermediate variables $v_i$

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$

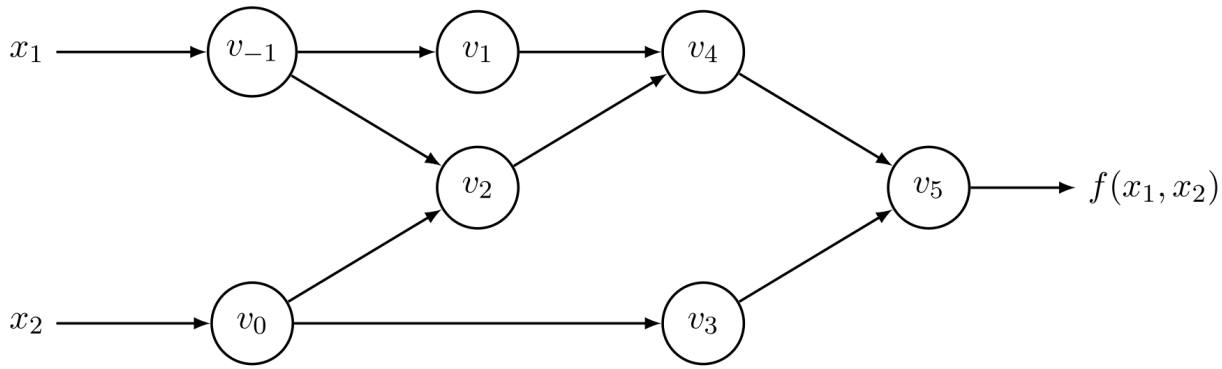
- Two-phase process
  - The 1st phase: forward
    - Populating intermediate variables $v_i$ and recording the dependencies
  - The 2nd phase: reverse
    - Propagating adjoints $\bar{v}_i$ from the outputs to the inputs
- A generalized *backpropagation* algorithm
  - *Remind that ML practice principally involves the gradient of a scalar-valued objective with respect to a large number of parameters.*

# Reverse mode (cont'd)

- Forward primal trace and reverse adjoint (derivative) trace

| | | | |
|---|---|---|---|
| $v_{-1}$ | $= x_1$ | $= 2$ | |
| $v_0$ | $= x_2$ | $= 5$ | |
| $v_1$ | $= \ln(v_{-1})$ | $= \ln(2)$ | $= 0.693$ |
| $v_2$ | $= v_{-1} \times v_0$ | $= 2 \times 5$ | $= 10$ |
| $v_3$ | $= \sin(v_0)$ | $= \sin(5)$ | $= -0.959$ |
| $v_4$ | $= v_1 + v_2$ | $= 0.693 + 10$ | $= 10.693$ |
| $v_5$ | $= v_4 - v_3$ | $= 10.693 + 0.959$ | $= 11.652$ |
| $y$ | $= v_5$ | $= 11.652$ | |

| | | | |
|---|---|---|---|
| $\bar{x}_1$ | $= \bar{v}_{-1,1} + \bar{v}_{-1,2}$ | $= 0.5 + 5$ | $= 5.5$ |
| $\bar{x}_2$ | $= \bar{v}_{0,2} + \bar{v}_{0,3}$ | $= 2 - 0.284$ | $= 1.716$ |
| $\bar{v}_{-1,1}$ | $= \bar{v}_{1,4} \frac{\partial v_1}{\partial v_{-1}} = \bar{v}_{1,4} / v_{-1}$ | $= 1/2$ | $= 0.5$ |
| $\bar{v}_{0,2}$ | $= \bar{v}_{2,4} \frac{\partial v_2}{\partial v_0} = \bar{v}_{2,4} \times v_{-1}$ | $= 1 \times 2$ | $= 2$ |
| $\bar{v}_{-1,2}$ | $= \bar{v}_{2,4} \frac{\partial v_2}{\partial v_{-1}} = \bar{v}_{2,4} \times v_0$ | $= 1 \times 5$ | $= 5$ |
| $\bar{v}_{0,3}$ | $= \bar{v}_{3,5} \frac{\partial v_3}{\partial v_0} = \bar{v}_{3,5} \times \cos(v_0)$ | $= -1 \times 0.284$ | $= -0.284$ |
| $\bar{v}_{2,4}$ | $= \bar{v}_{4,5} \frac{\partial v_4}{\partial v_2} = \bar{v}_{4,5} \times 1$ | $= 1 \times 1$ | $= 1$ |
| $\bar{v}_{1,4}$ | $= \bar{v}_{4,5} \frac{\partial v_4}{\partial v_1} = \bar{v}_{4,5} \times 1$ | $= 1 \times 1$ | $= 1$ |
| $\bar{v}_{3,5}$ | $= \bar{v}_{5,y} \frac{\partial v_5}{\partial v_3} = \bar{v}_{5,y} \times (-1)$ | $= 1 \times -1$ | $= -1$ |
| $\bar{v}_{4,5}$ | $= \bar{v}_{5,y} \frac{\partial v_5}{\partial v_4} = \bar{v}_{5,y} \times 1$ | $= 1 \times 1$ | $= 1$ |
| $\bar{v}_{5,y}$ | $= \bar{y}$ | $= 1$ | |



# Automatic differentiation in TensorFlow

- Perform the **reverse mode** differentiation
- *Gradient tapes*: TensorFlow API for automatic differentiation
  - `tf.GradientTape`
    - *Record* relevant operations executed inside this context onto a "tape"
  - `GradientTape.gradient(target, sources)`
    - Calculate the gradient

```
x = tf.Variable(3.0)
with tf.GradientTape() as tape:
  y = x**2
```

```
dy_dx = tape.gradient(y, x)
dy_dx.numpy()
```

```
6.0
```

- The "best" material for the details
  - https://www.tensorflow.org/guide/autodiff

# Gradients w.r.t. a model in TensorFlow

- In most cases
  - We want to calculate gradients with respect to a model's trainable variables.
  - *Why?*
  - Update the model's parameters: **Backpropagation**!

```
layer = tf.keras.layers.Dense(2, activation='relu') # 2 outputs
x = tf.constant([[1., 2., 3.]])                     # 1 x 3 inputs

with tf.GradientTape() as tape:
  # Forward pass
  y = layer(x)                  # x:[1x3] . w:[3x2] + b:[1x2] = y:[1x2]
  loss = tf.reduce_mean(y**2)

# Calculate gradients with respect to every trainable variable
grad = tape.gradient(loss, layer.trainable_variables)

for var, g in zip(layer.trainable_variables, grad):
  print(f'{var.name}, shape: {g.shape}')
```

```
dense/kernel:0, shape: (3, 2)
dense/bias:0, shape: (2,)
```

# What can we do with this?

# Numerical simulations

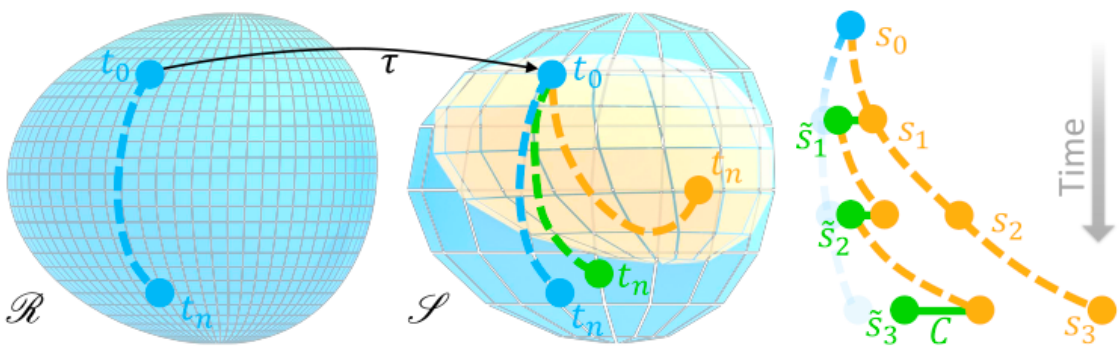
- Numerical methods $\longrightarrow$ errors
  - Discrete simulation of PDEs
  - Higher-order methods: preferable but difficult in practice
  - As numerical solutions are integrated over time, error often grows exponentially.

- Our target:
  - A discretized PDE solver $\mathcal{P}$
    solving for discrete time steps $\Delta t$
    using a regular grid

- Each subsequent step depends on any number of previous steps.
  - E.g., $\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathcal{P}(\mathbf{u}(\mathbf{x}, t), \mathbf{u}(\mathbf{x}, t - \Delta t), \mathbf{u}(\mathbf{x}, t - 2\Delta t), \cdots)$
    - $\mathbf{x} \in \Omega \subset \mathbb{R}^d$
    - $t \in \mathbb{R}^+$

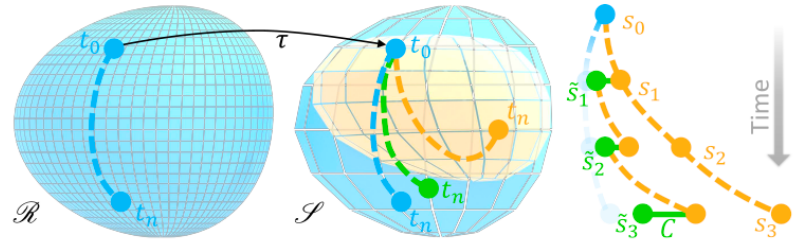
# Different numerical solvers

- Two different solvers of the same PDE
  - $\mathcal{P}_R$: **accurate** solver
    - Solutions $\mathbf{r} \in \mathscr{R}$ from *reference* manifold
  - $\mathcal{P}_S$: **efficient** solver
    - Approximated solutions $\mathbf{s} \in \mathscr{S}$ from *source* manifold
- Reference sequence: $\{\mathbf{r}_t, \mathbf{r}_{t+\Delta t}, \cdots, \mathbf{r}_{t+k\Delta t}\}$
- Source sequence: $\{\mathbf{s}_t, \mathbf{s}_{t+\Delta t}, \cdots, \mathbf{s}_{t+k\Delta t}\}$
- Mapping operator $\mathcal{T}$
  - Transform a phase space point from one solution manifold to a suitable point in the other manifold
  - E.g., $\mathbf{s} = \mathcal{T}\mathbf{r}$



# Problem statement

- Evaluating $\mathcal{P}_R$ for $\mathscr{R}$
  - $\mathbf{r}_{t+\Delta t} = \mathcal{P}_R(\mathbf{r}_t)$
  - ... $\mathbf{r}_{t+k\Delta t}$ ... simply $\mathbf{r}_{t+k}$



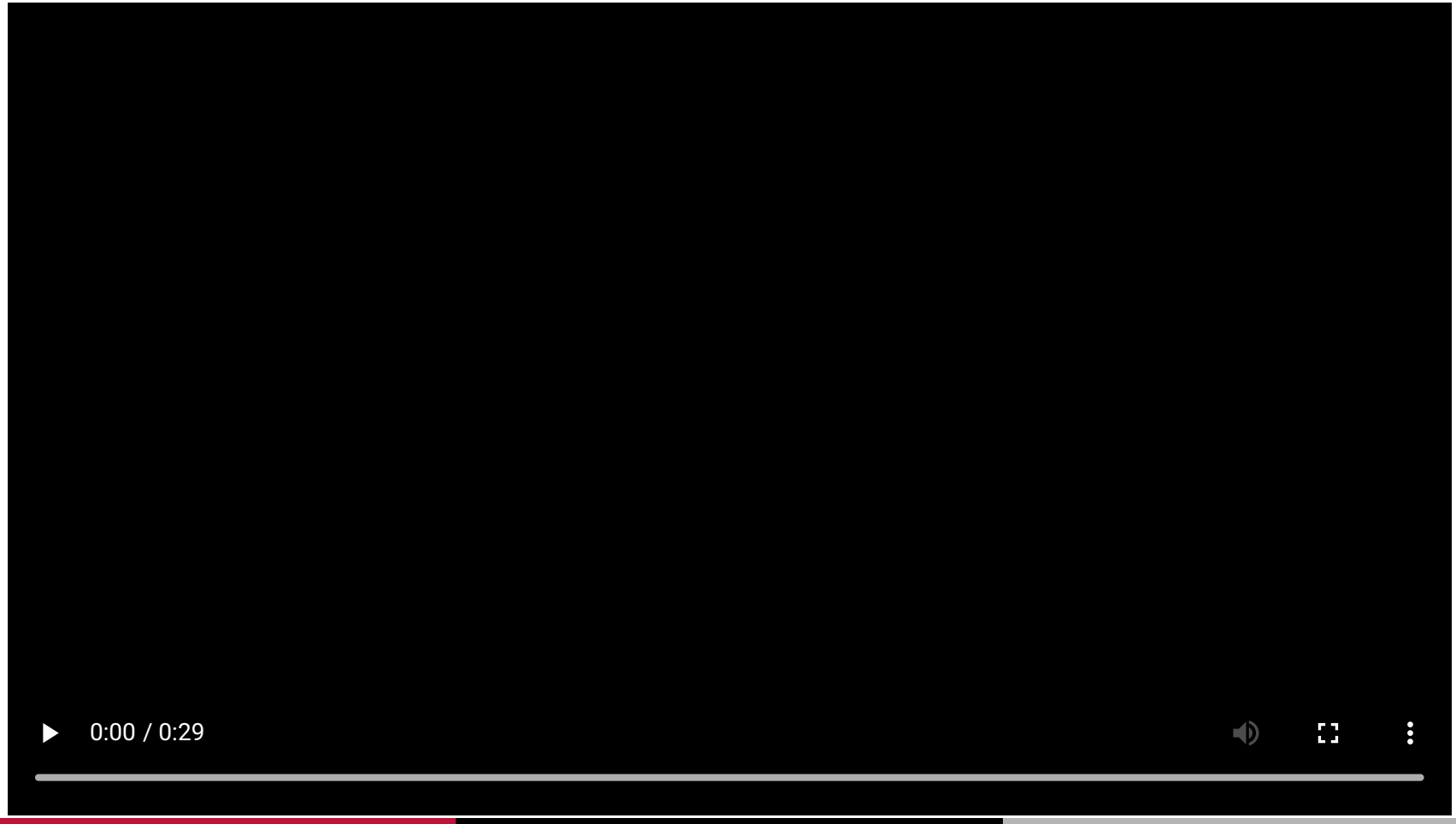

- Different numerical approximations: $\mathcal{T}\mathbf{r}_{t+1} \neq \mathcal{P}_S(\mathcal{T}\mathbf{r}_t)$
- Deviations, $l^2$-norm: $\mathcal{L}(\mathbf{s}_t, \mathcal{T}\mathbf{r}_t) = \|\mathbf{s}_t - \mathcal{T}\mathbf{r}_t\|_2$
- Goal: find a correction operator $\mathcal{C}(\mathbf{s})$
  - $\mathcal{L}(\mathcal{C}(\mathcal{P}_S(\mathcal{T}\mathbf{r}_t)), \mathcal{T}\mathbf{r}_{t+1}) < \mathcal{L}(\mathcal{P}_S(\mathcal{T}\mathbf{r}_t), \mathcal{T}\mathbf{r}_{t+1})$
- **Deep neural network model**: $\mathcal{C}(\mathbf{s}|\theta)$
- Recursive application of solver
  - $\mathbf{s}_{t+n} = \mathcal{P}_S(\mathcal{P}_S(\cdots\mathcal{P}_S(\mathcal{T}\mathbf{r}_t)\cdots)) = \mathcal{P}_S^n(\mathcal{T}\mathbf{r}_t)$
  - $\tilde{\mathbf{s}}_{t+n} = \mathcal{C}(\mathcal{P}_S(\mathcal{C}(\mathcal{P}_S(\cdots\mathcal{C}(\mathcal{P}_S(\mathcal{T}\mathbf{r}_t))\cdots)))) = \mathcal{P}_S\mathcal{C}^n(\mathcal{T}\mathbf{r}_t)$

# Related work

- Negiar et al., *Learning differentiable solvers for systems with hard constraints*, ICLR 2023
- Kochkov et al., *Machine learning–accelerated computational fluid dynamics*, PNAS 2021
- Holl et al., *Learning to control PDEs with differentiable physics*, ICLR 2020
- Bar-Sinai et al., *Learning data-driven discretizations for partial differential equations*, PNAS 2019
- Raissi and Karniadakis, *Hidden physics models: Machine learning of nonlinear partial differential equations*, JCP 2018
- Chen et al., *Neural ordinary differential equations*, NeurIPS 2018
- Long et al., *PDE-Net: Learning PDEs from data*, ICML 2018
- ...

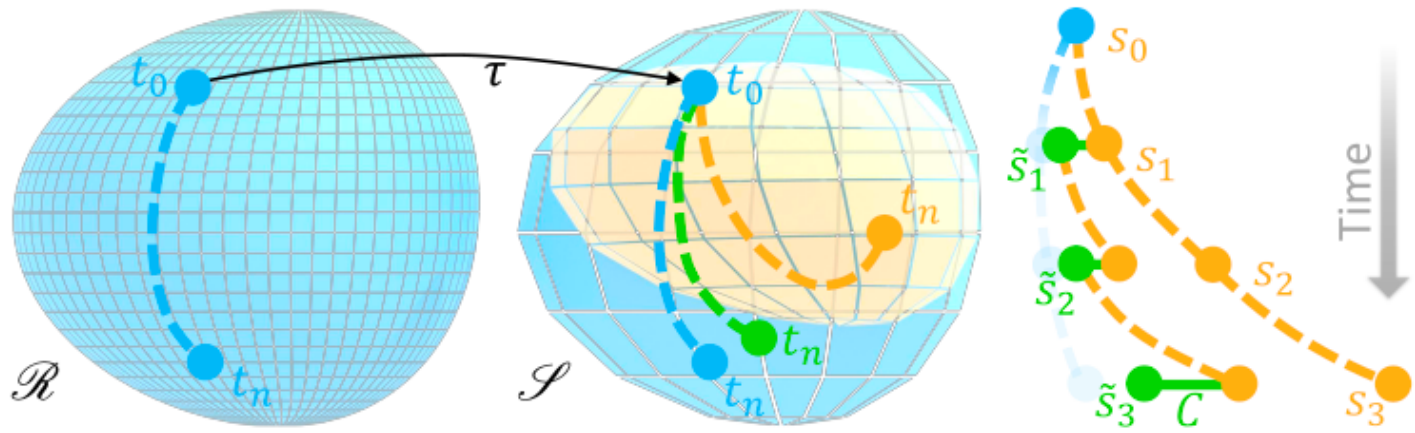
# Highlight

- Efficiency & accuracy
  - Use low-resolution setups, i.e., less amount of resources
  - Generate solutions as accurate as possible



# Models

- Non-interacting model, **NON**
- Pre-computed interaction model, **PRE**
- Solver-in-the-loop model, **SOL**

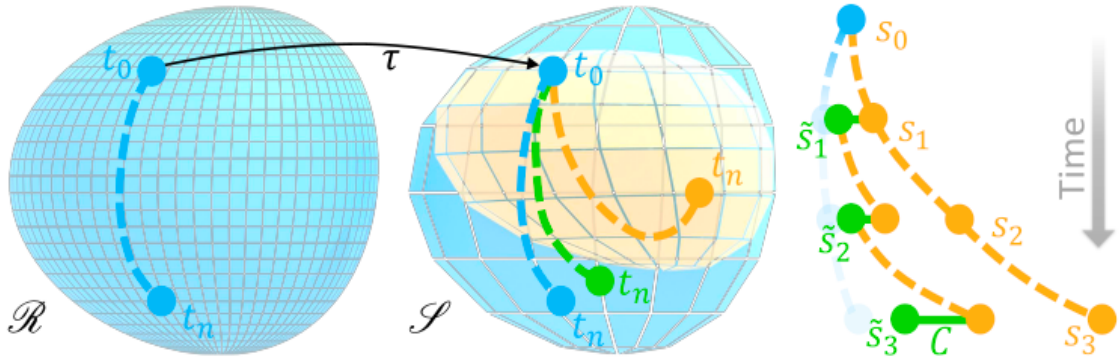


# Non-interacting (NON)

- Learn from the unaltered PDE trajectories
  - $\mathbf{s}_{t+k} = \mathcal{P}_S^k(\mathcal{T}\mathbf{r}_t)$

- Model will not see any states deviated from the original solutions.
  - Learn *difference between **orange** and **blue** trajectories*

- $\arg\min_\theta \sum_{i=0}^{n} \left\| \mathbf{s}_{t+i} + \mathcal{C}(\mathbf{s}_{t+i}|\theta) - \mathcal{T}\mathbf{r}_{t+i} \right\|^2$

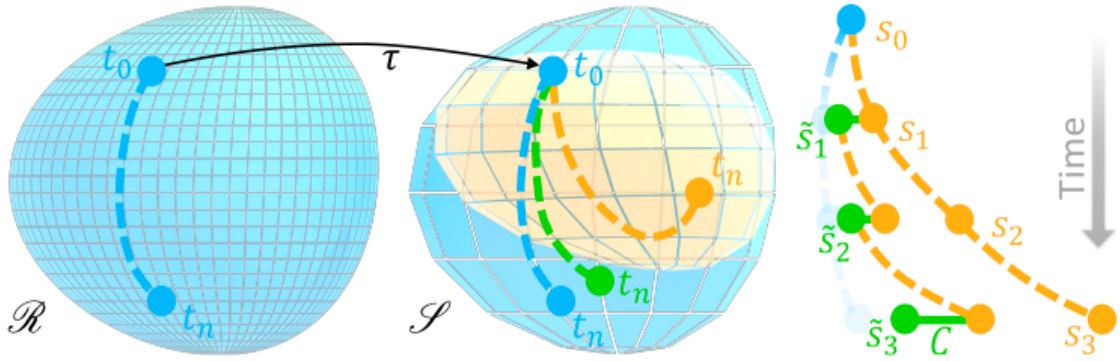


# Pre-computed interaction (PRE)

- Learn from the pre-computed correction function, $\mathcal{C}_{\text{pre}}$
  - $\tilde{\mathbf{s}}_{t+k} = (\mathcal{P}_S \mathcal{C}_{\text{pre}})^k (\mathcal{T}\mathbf{r}_t)$
  - *How to pre-compute? … coming soon*

- Model will see the corrected states, which are not influenced by the learning process.

- $\arg\min_\theta \sum_{i=0}^{n} \|\mathbf{c}_{t+i} - \mathcal{C}(\tilde{\mathbf{s}}_{t+i}|\theta)\|^2$



# Training data for PRE model

- Assume the *perfect* correction $\mathbf{c}_R$
- Given a high-res. correction $\mathbf{c}_R$, solve for $\mathbf{c}_S$ (on low-res.):

$$\underset{\mathbf{c}_S}{\operatorname{argmin}} \|\mathbf{W}\mathbf{c}_S - \mathbf{c}_R\|^2 \quad \text{subject to} \quad \mathbf{M}\mathbf{c}_S = 0$$

  - $\mathbf{W}$: transform $\mathscr{S}$ to $\mathscr{R}$, e.g., *interpolation*
  - $\mathbf{M}$: constraints, e.g., *divergence* $\nabla\cdot$ *for the Navier-Stokes scenario*
- Linear system with Lagrange multiplier: $\lambda$

$$\begin{bmatrix} \mathbf{W}^\top\mathbf{W} & -\mathbf{M} \\ -\mathbf{M}^\top & 0 \end{bmatrix} \begin{bmatrix} \mathbf{c}_S \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{W}^\top\mathbf{c}_R \\ 0 \end{bmatrix}$$

- Simplify with the Schur complement

$$\mathbf{M}^\top(\mathbf{W}^\top\mathbf{W})^{-1}\mathbf{M}\lambda = \mathbf{M}^\top(\mathbf{W}^\top\mathbf{W})^{-1}\mathbf{W}^\top\mathbf{c}_R$$
$$\mathbf{c}_S = (\mathbf{W}^\top\mathbf{W})^{-1}(\mathbf{W}^\top\mathbf{c}_R - \mathbf{M}\lambda)$$

- *In our example,* $\mathbf{c}_t = (\mathbf{W}^\top\mathbf{W})^{-1}(\mathbf{W}^\top(\mathbf{r}_t - \mathbf{W}\mathbf{s}_t) - \mathbf{M}\lambda)$

# Temporal regularization

- Very chaotic corrections despite smoothly changing inputs
  - Need to make the corrections smooth as well
  - *Why?* ... hoping the model can learn better

- Updated objective function:

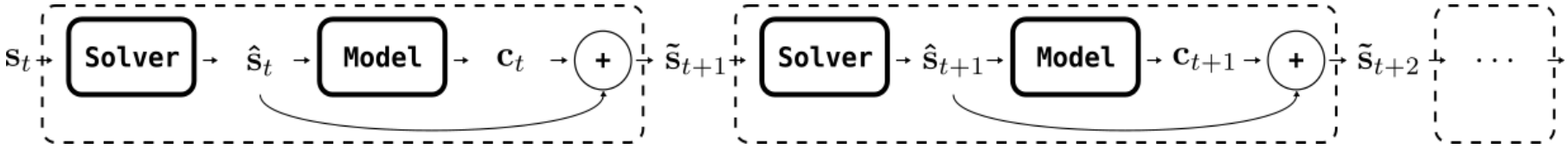
$$\underset{\mathbf{c}_S}{\operatorname{argmin}} \left( \|\mathbf{W}\mathbf{c}_S - \mathbf{c}_R\|^2 + \beta \| \frac{d\mathbf{c}_S}{dt} \|^2 \right) \quad \text{subject to} \quad \mathbf{M}\mathbf{c}_S = 0$$

- Linear system with the regularization

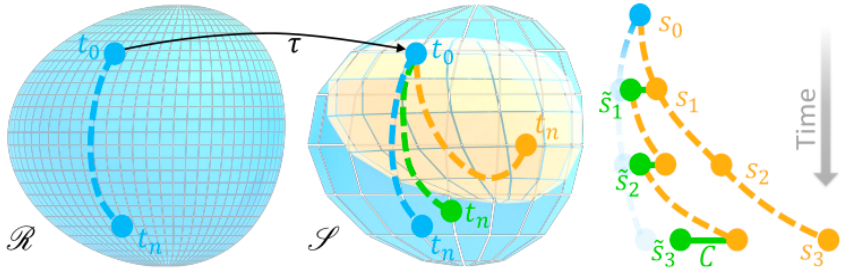
$$\begin{bmatrix} \mathbf{W}^\top\mathbf{W} + \beta\frac{2}{\Delta t}\mathbf{I} & -\mathbf{M} \\ -\mathbf{M}^\top & 0 \end{bmatrix} \begin{bmatrix} \mathbf{c}_S \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{W}^\top\mathbf{c}_R + \beta\frac{2}{\Delta t}\mathbf{c}_S^{t-1} \\ 0 \end{bmatrix}$$

# Solver-in-the-loop (SOL)

- The learning process interacts with the correction function, $\mathcal{C}$.
  - $\tilde{\mathbf{s}}_{t+k} = (\mathcal{P}_S \mathcal{C})^k (\mathcal{T} \mathbf{r}_t)$
  - Similar to PRE, but what differences?
    - ... $\mathcal{C}$ interacts with the physical system.
  - Need a *differentiable physics solver!*

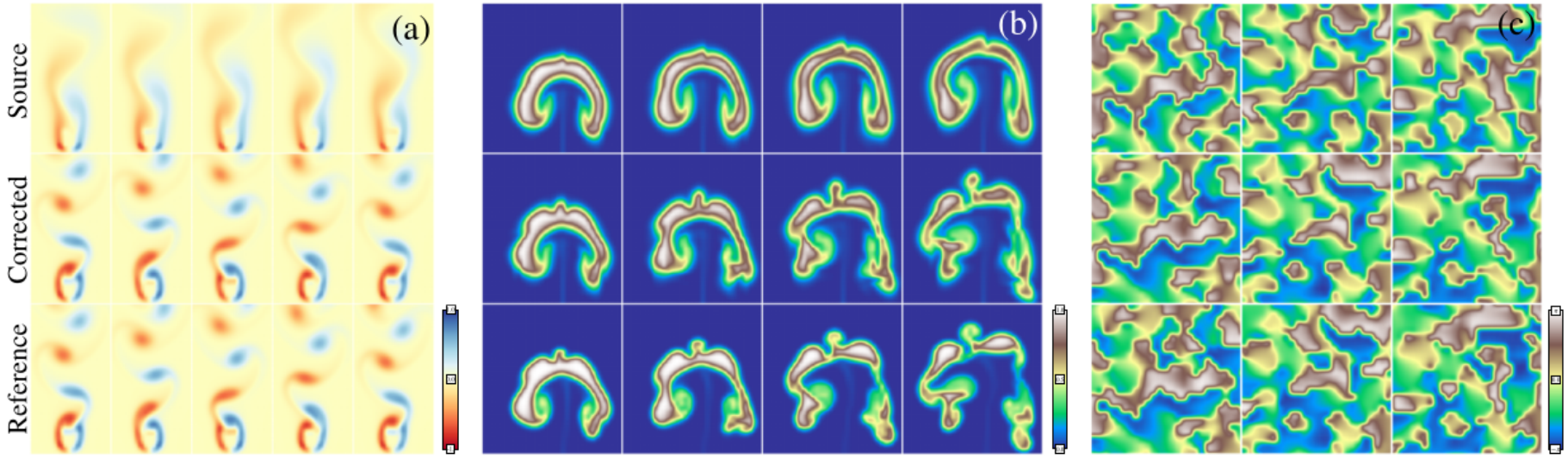



- Model will see the corrected states and influence the evolution of trajectories.
- $\arg\min_\theta \sum_{i=0}^{n-1} \| \mathcal{P}_S(\tilde{\mathbf{s}}_{t+i}) + \mathcal{C}(\mathcal{P}_S(\tilde{\mathbf{s}}_{t+i}) | \theta) - \mathcal{T} \mathbf{r}_{t+i+1} \|^2$



# Experiments

- 5 complex examples
  - Unsteady wake flow[(a)]
  - Buoyancy-driven flow[(b)]
  - Forced advection-diffusion[(c)]
  - Conjugate gradient solver
  - Three-dimensional fluid flow



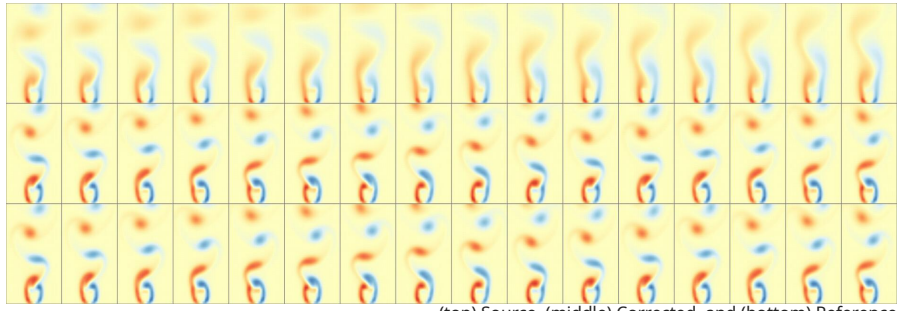
# Unsteady wake flow

- Incompressible Navier-Stokes equations for Newtonian fluids:

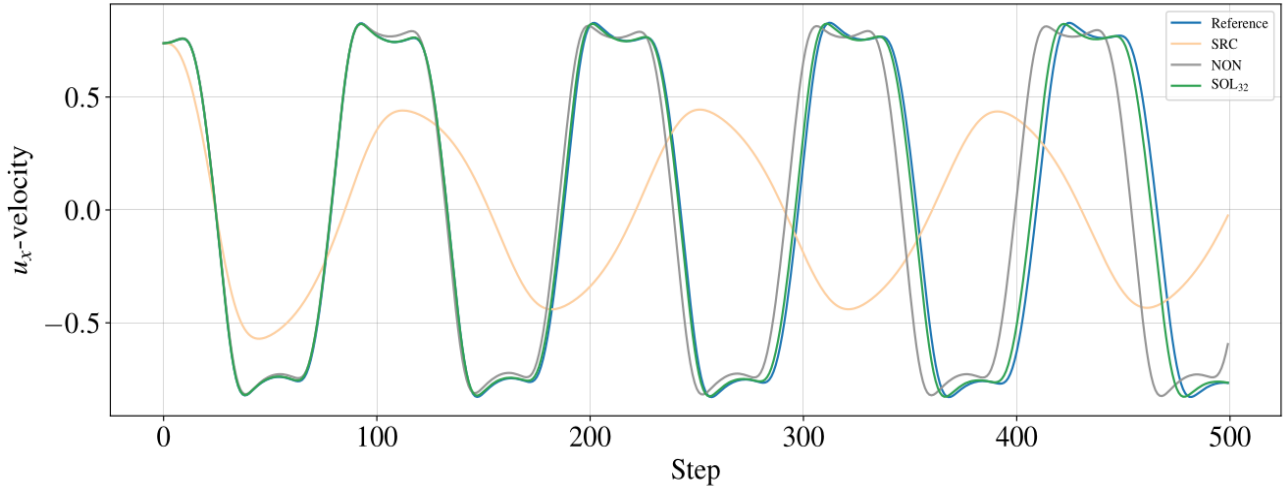
$$\frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x = -\frac{1}{\rho}\nabla p + \nu \nabla \cdot \nabla u_x$$

$$\frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y = -\frac{1}{\rho}\nabla p + \nu \nabla \cdot \nabla u_y$$

$$\text{subject to} \quad \nabla \cdot \mathbf{u} = 0$$



(top) Source, (middle) Corrected, and (bottom) Reference

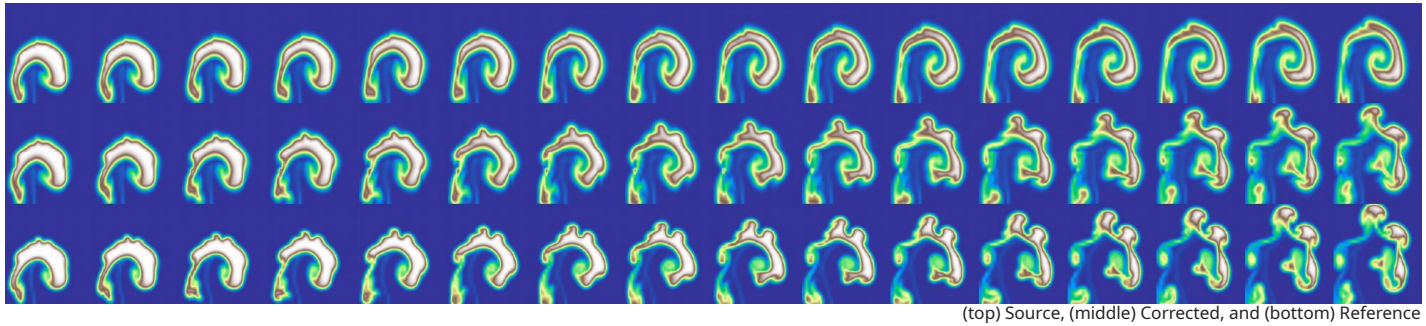


# Buoyancy-driven flow

- Incompressible N.-S. equations with a marker density field:

$$\frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x = -\frac{1}{\rho}\nabla p, \qquad \frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y = -\frac{1}{\rho}\nabla p + \eta d \quad \text{subject to} \quad \nabla \cdot \mathbf{u} = 0$$

$$\frac{\partial d}{\partial t} + \mathbf{u} \cdot \nabla d = 0$$



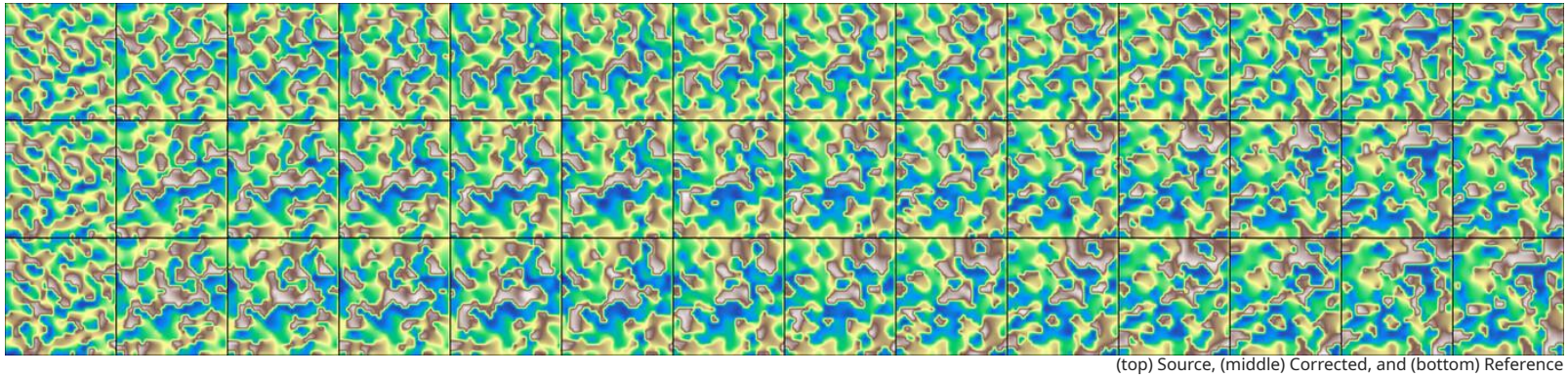
(top) Source, (middle) Corrected, and (bottom) Reference

| **Quantity** | **MAE Velocity**, Mean (std. dev.) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | SRC | NON | PRE | $SOL_2$ | $SOL_{16}$ | $SOL_{32}$ | $SOL_{64}$ | $SOL_{128}$ |
| Velocity | 1.590 (1.032) | 1.079 (0.658) | 1.373 (0.985) | 1.027 (0.656) | 0.859 (0.539) | 0.775 (0.482) | 0.695 (0.420) | 0.620 (0.389) |
| Marker $d$ | 0.677 (0.473) | 0.499 (0.336) | 0.579 (0.409) | 0.484 (0.325) | 0.430 (0.281) | 0.419 (0.277) | 0.401 (0.262) | 0.391 (0.253) |

# Forced advection-diffusion

- Burgers' equation

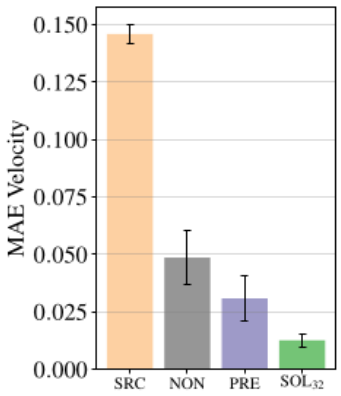
$$\frac{\partial u_x}{\partial t} + \mathbf{u} \cdot \nabla u_x = \nu \nabla \cdot \nabla u_x + g_x(t), \quad \frac{\partial u_y}{\partial t} + \mathbf{u} \cdot \nabla u_y = \nu \nabla \cdot \nabla u_y + g_y(t)$$



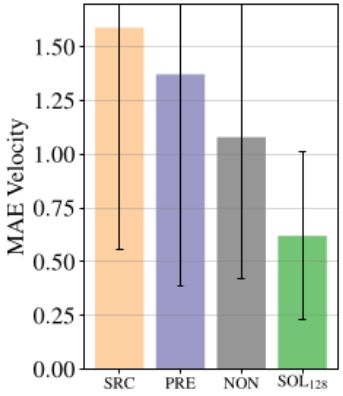
(top) Source, (middle) Corrected, and (bottom) Reference

| **MAE Velocity**, Mean (std. dev.) | | | | | | |
|---|---|---|---|---|---|---|
| With forcing | SRC | PRE | NON | $SOL_2$ | $SOL_4$ | $SOL_8$ |
| | 0.248 (0.019) | 0.218 (0.017) | 0.159 (0.015) | 0.148 (0.016) | 0.152 (0.015) | 0.158 (0.017) |
| Without forcing | SRC | NON | $SOL_4$ | $SOL_8$ | $SOL_{16}$ | $SOL_{32}$ |
| $(\times 100)$ | 0.306 (0.020) | 0.272 (0.028) | 0.276 (0.037) | 0.277 (0.040) | 0.268 (0.030) | 0.253 (0.020) |

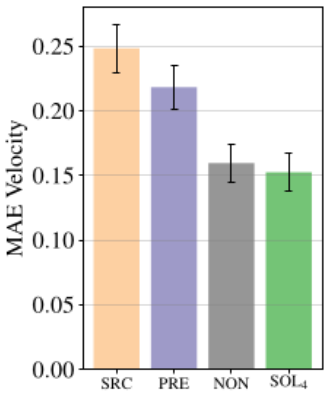
# Errors and improvements



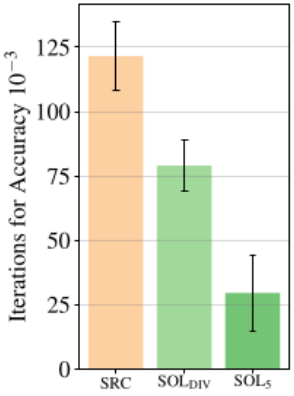

(a) Unsteady wake



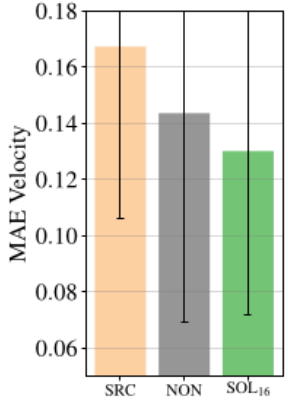

(b) Buoyancy-driven



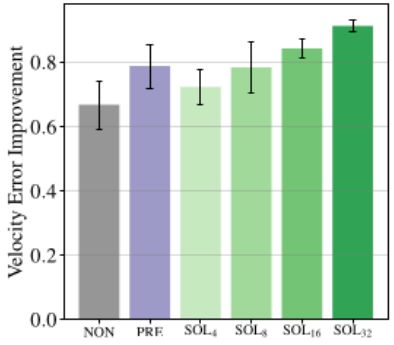

(c) Advection-diffusion



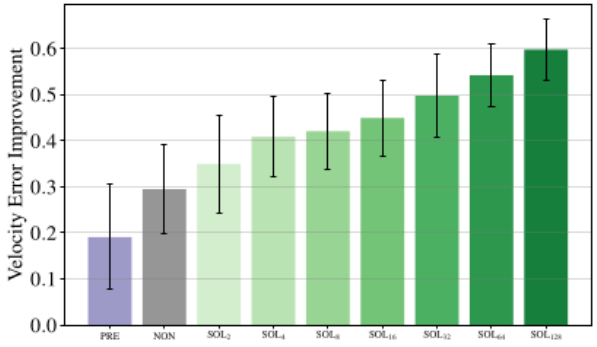

(d) CG solver



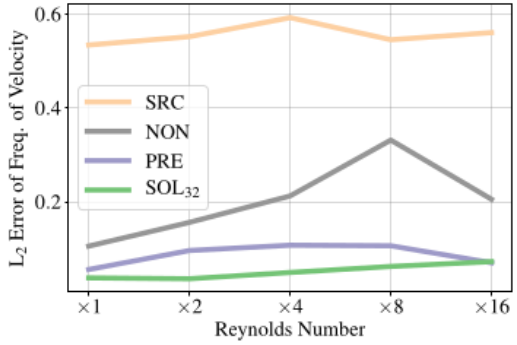

(e) 3D wake




(f) Unsteady wake, look-ahead




(g) Buoyancy-driven, look-ahead




(h) Unsteady wake, frequency error

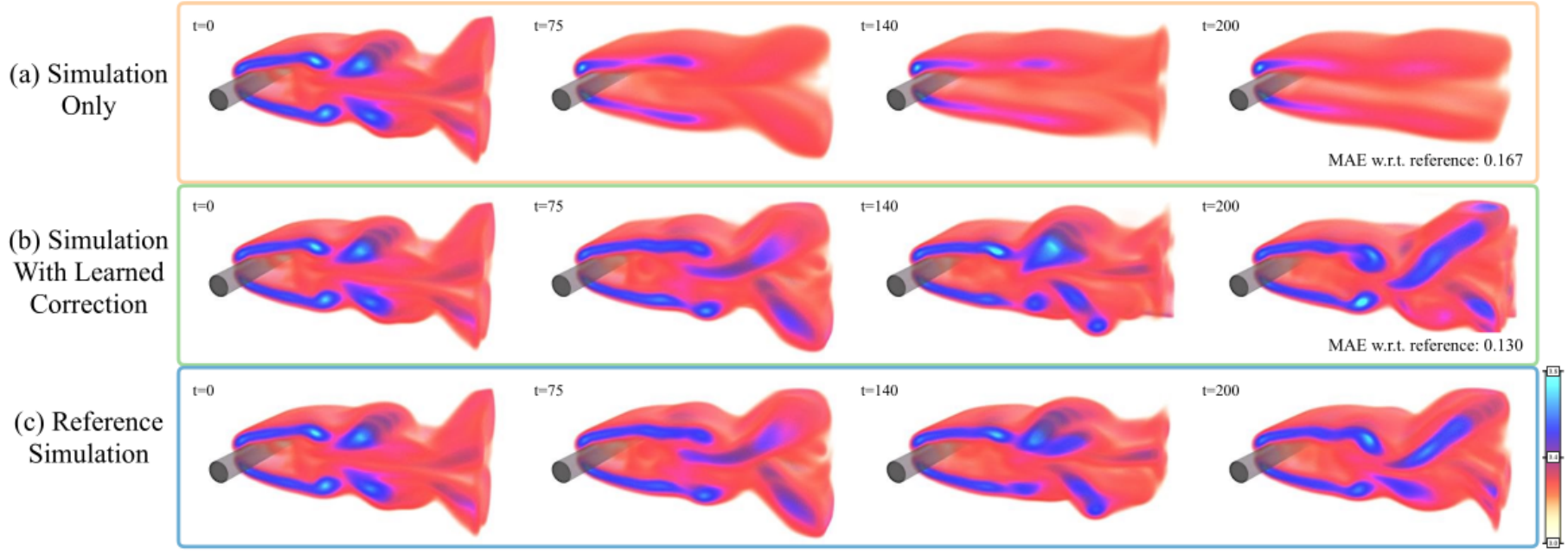
# Quantitative summary

| Exp. | Mean absolute error of velocity | | | | | Rel. improvement | | | |
|---|---|---|---|---|---|---|---|---|---|
| | SRC | PRE | NON | $SOL_s$ | SOL | PRE | NON | $SOL_s$ | SOL |
| Wake Flow | 0.146±0.004 | 0.031±0.010 | 0.049±0.012 | 0.041±0.009 | 0.013±0.003 | 79% | 67% | 72% | 91% |
| Buoyancy | 1.590±1.033 | 1.373±0.985 | 1.080±0.658 | 0.944±0.614 | 0.620±0.390 | 19% | 29% | 41% | 60% |
| Adv.-diff. | 0.248±0.019 | 0.218±0.017 | 0.159±0.015 | 0.152±0.015 | 0.158±0.017 | 12% | 36% | 39% | 36% |
| *CG Solver | 121.6±13.44 | - | - | 79.03±10.02 | 29.59±14.83 | - | - | 35% | 76% |
| 3D Wake | 0.167±0.061 | - | 0.144±0.074 | - | 0.130±0.058 | - | 14% | - | 22% |

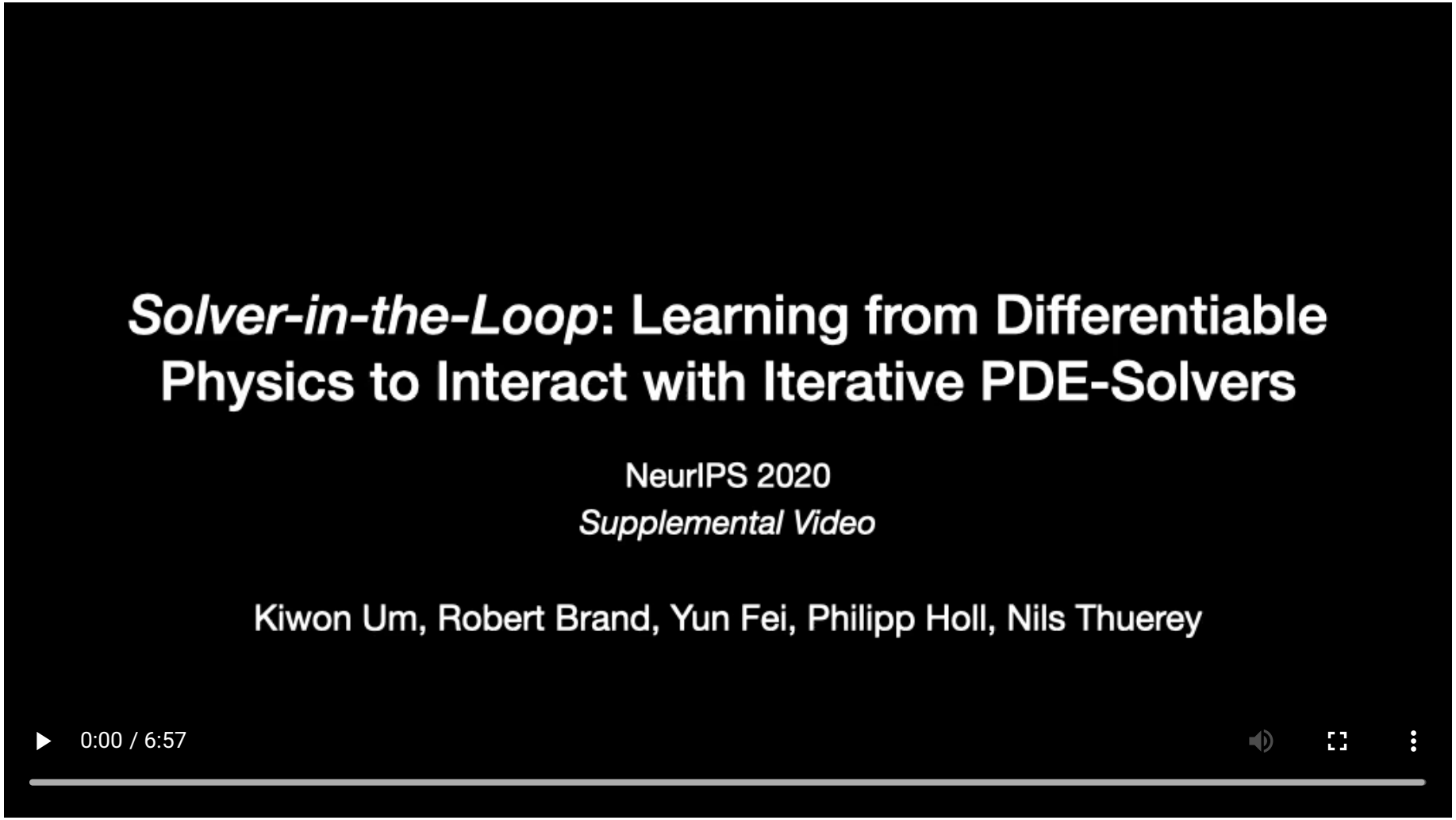
- $SOL_S$: smaller model
- * Iterations to reach an accuracy of 0.001

# Performance

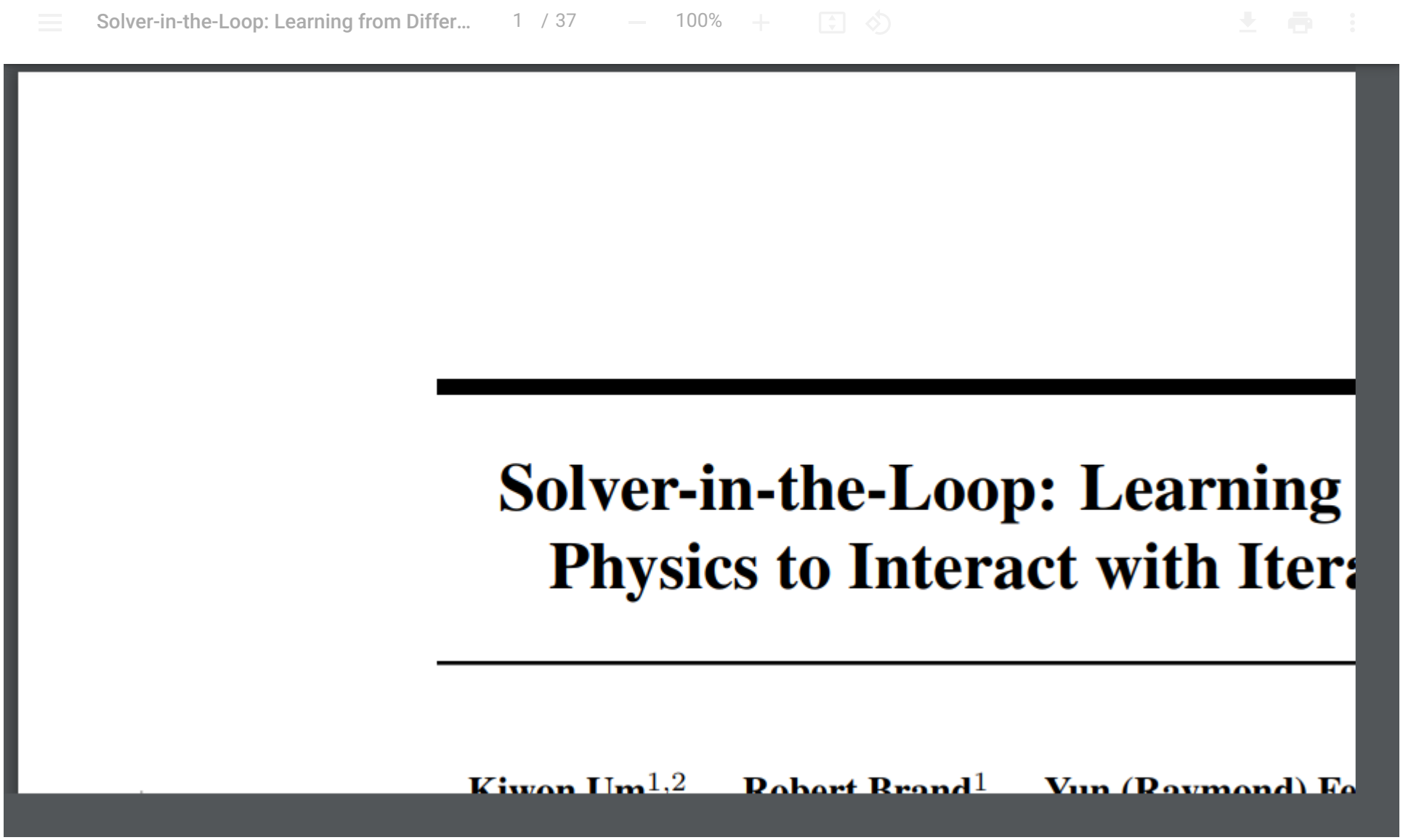
- 3D Wake Flow: average cost for 100 time steps
- Reference simulation (CPU): 913.2 seconds
- Source simulation with the trained model: 13.3 seconds
- **Speed-up of ca. 68**×



# Performance (cont'd)



# More details

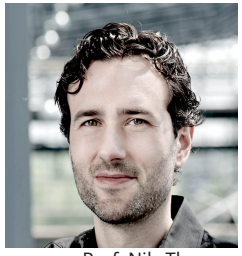


# Thank you! Questions?

The online book, *Physics-based Deep Learning*, and the codes are available!

https://www.physicsbaseddeeplearning.org/


Special thanks to my collaborators:


Prof. Nils Thuerey


Philipp Holl