Simulating Incompressible Flow with Physics-Informed Neural Networks

Lasse Köster, Claas Nitschke, and Lauritz Timm

Kiel University, Kiel, Germany
lauritz.timm@stu.uni-kiel.de

Abstract. The simulation of complex systems, like climate, is an important application of numerical methods to model differential equations. The motion of viscous fluids is describes by Navier–Stokes Equations. In this paper, we look into the possibility to train an neural network to predict the velocity and pressure field of flow inside a pipe. Based on the work of Raissi et al. we introduce an approach of using Physics-Informed Neural Networks (PINN) for unsupervised learning of the flow function in a two-dimensional space. Therefore, we propose a simple framework in PyTorch, enhancing re-usability and applicability of PINN to similar problems described by differential equations. We compare our results with the state-of-the art tool OpenFoam to evaluate the accuracy of our simulation.

Keywords: Navier-Stokes-Equations · PINN · PyTorch.

1 Introduction

Climate simulation and weather prediction are important topics of ongoing research [10]. With the recent developments in machine learning [5], we aim at improving existing numerical methods, by using the properties of neural networks to approximate differential equations, particularly incompressible flow modeled through a Navier-Stokes Equation (NSE) [11]. We investigate, whether it is possible to provide a simple setup to predict a complex system.

1.1 Goal of this Paper

With this paper we want to show, that we can use physics-informed neural networks (PINN) to predict flow functions of viscous fluids with similar accuracy compared to state-of-the-art (SOTA) tools. Therefore, we determine optimal training parameters to train a shallow neural network unsupervised to satisfy NSE at all spacial coordinates. In addition we investigate convergence of gradient descent and optimization of such a PINN. Since we are mainly interested to provide an estimation of the flow properties, we do not aim at a perfect prediction, but rather a trade-off between required training and accuracy.

1.2 Structure

In the following section, we will define some underlying foundations and concepts of the physical model and numerical simulation. Next, we will discuss our implementing of a PINN compared to a similar approach by Raissi et al. [8] and discuss how we differ regarding the unsupervised approach. After we determined hyper-parameters of our simulation, we compare several experiments against a reference model realized in OpenFoam [6]. We will conclude our work with discussing these results, threats to validity and possible future work, as well as possible extensions of our approach.

1.3 Research Questions

To guide this paper, we introduce the following research questions:

- RQ1 Can we reproduce the results by Raissi et al. with a simplified PyTorch implementation?

- RQ2 Can we predict a stream function with an unsupervised learning approach?
  - RQ2.1 How well can we predict velocity and pressure field compared to a SOTA tool?

- RQ3 What are the capabilities of using machine learning to simulate complex systems?
  - RQ3.1 How does our model compare to SOTA tools in respect to performance and accuracy?

2 Background and Related Work

In this section, we will introduce equations and concepts providing the underlying foundation for this paper.

2.1 Navier-Stokes Equation for Incompressible Flow

The Navier-Stokes equations describe the motion of viscous fluids such as liquids and gases. These equations are part of fluid dynamics and can be used to model a wide range of physical systems. For our case we are interested in the partial differential equation (PDE) for incompressible flow [11].

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\,\mathbf{u} = \nu \nabla^2 \mathbf{u} - \frac{1}{\rho} \nabla p \tag{1}$$

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y} \right) \qquad \nabla^2 f = \Delta f = \left(\frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y} \right) \tag{2}$$

Note that ∇ is the Nabla-Operator, whereas ∇^2 is the Laplacian Δ, as defined in (2) on two-dimensional Euclidean space $\mathbb{E}^2$. A two-dimensional flow function can be defined as ψ, so that we derive u and v as velocities in x and y direction [11]. Since we will only predict the niveous of such a flow function, we do not need to consider conservation of mass further [11].

$$u = \frac{\partial \psi}{\partial x} \qquad v = -\frac{\partial \psi}{\partial y} \tag{3}$$

Assume the density of a fluid is uniform $\rho = 1$. We introduce a Reynolds Number (Re) as characteristic of a system with viscosity ν, directional velocity u, and characteristic length L. Note, that ν is originally the dynamic viscosity, but since we set $\rho = 1$, we can ignore the dynamic part.

$$\mathrm{Re} = \frac{uL}{\nu} \tag{4}$$

We split the flow velocity vector field $\mathbf{u}$ in (1) into two components that denote the flow in the x and y direction denoted by u and v respectively. Applying this transformation to (1) we get two equations describing the physical properties of a fluid [11].

$$\begin{aligned}\frac{\partial u}{\partial t} + \left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) &= \nu\left(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y}\right) - \frac{\partial p}{\partial x}\\ \frac{\partial v}{\partial t} + \left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) &= \nu\left(\frac{\partial^2 v}{\partial^2 x} + \frac{\partial^2 v}{\partial^2 y}\right) - \frac{\partial p}{\partial y}\end{aligned} \tag{5}$$

We get Burgers' equation from (1) for a time-dependent one-dimensional flow using the respective operators in $\mathbb{E}^1$ and ignoring spatial changes of pressure [11].

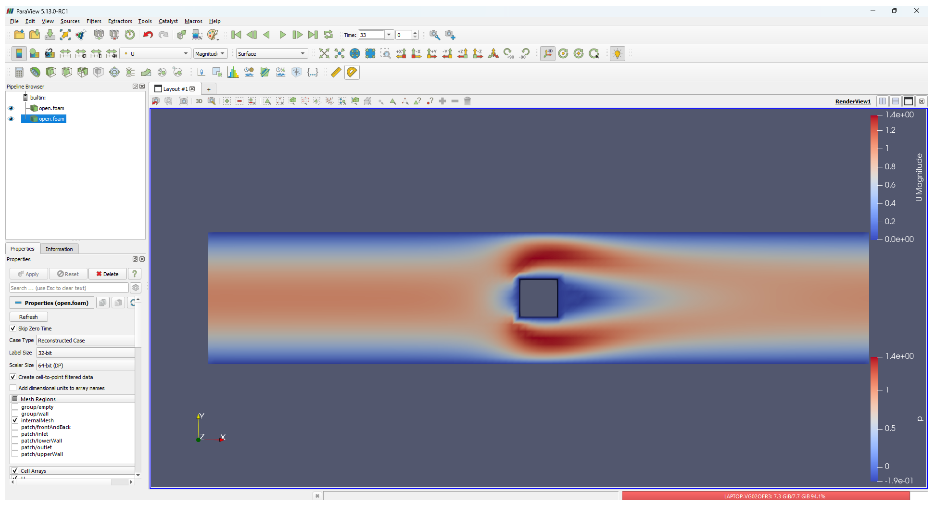
$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{6}$$

Note, that since BE only considers the velocity change in one dimension, a single equation suffices to describe the whole physical system.

2.2 OpenFOAM

OpenFOAM (Open Source Field Operation and Manipulation) [6] is a popular open source software used for computational fluid dynamics and continuum mechanics. It provides a flexible, extensible platform for developing and analyzing models of fluid flow, heat transfer, and other related physical processes. We use it for the computation and modeling of our reference setup.

OpenFOAM uses numerical solvers. They discretize the Navier-Stokes equations over a mesh of finite volumes, allowing for an iterative computation over time. We run the simulations until we reach a stationary solution, which can then be compared to the prediction of our PINN. Since our PINN directly produces a stationary solution, other time steps are not relevant for our comparison. The SimpleFoam solver is one of the methods of solving OpenFOAM using the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm, described by Patankar [7]. It is designed for steady-state problems, where the flow does not change over time. The fluid is assumed to be incompressible. SimpleFoam can work with and without turbulence models. Results of simulations can be visualised using ParaView [3], a popular tool used in conjunction with OpenFOAM to display the simulation data, shown in Fig. 1.



✓ Skip Zero Time
Case Type Reconstructed Case
Label Size 32-bit
Scalar Size 64-bit (DP)
✓ Create cell-to-point filtered data
Add dimensional units to array names
Mesh Regions
group/empty
group/wall
✓ internalMesh
patch/frontAndBack
patch/inlet
patch/lowerWall
patch/outlet
patch/upperWall
✓ Cell Arrays

Fig. 1: *An* OpenFOAM *simulation of the flow around a cube, visualised in* ParaView.

2.3 Artificial neural networks

For the scope of this work, we only look into shallow feed-forward neural networks, but there exists a variety of different architectures, like deep, convolutions, and recurrent networks [12]. Such a feed-forward network can be represented as a directed graph, as shown in Fig. 2. We see that we have an input layer, which takes the input vector, for example a coordinate (x, y), and forwards it through the network resulting in an output vector, in this case the values of a stream-function and pressure field is received. In addition, each output of a hidden layer might have an additional bias and activation function. [2].

Moreover, a network can be interpreted as a mapping $\mathcal{N}$, which, in this example, computes the value of $(\psi(x, y), p(x, y))$ at (x, y).

$$\mathcal{N} : (x, y) \mapsto \begin{pmatrix} \psi(x, y) \\ p(x, y) \end{pmatrix} \tag{7}$$

In algebraic terms, $\mathcal{N}$ is a vector field, mapping each input vector to its respected output vector. As we might have an non-linear, for

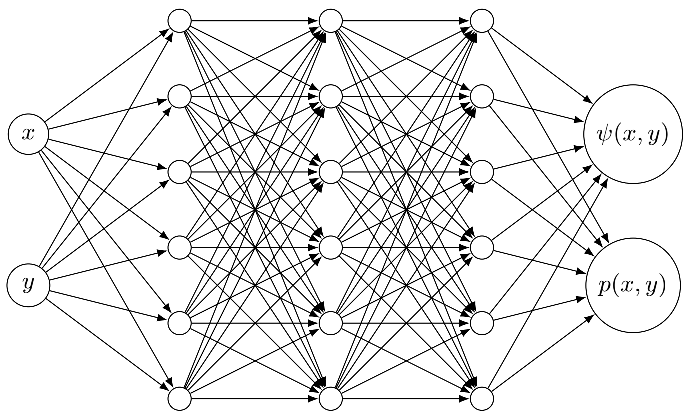


Fig. 2: *Fully-connected feed-forward neural-network with input / output of size 2 and three hidden layers, each with six artificial neurons. In this example, an singular coordinate* (x, y) *is forwarded through the network, resulting in the prediction of stream-function* $\psi(x, y)$ *and pressure* $p(x, y)$.

example hyperbolic, activation function applied to each layer, $\mathcal{N}$ inherits this property as well.

To improve the accuracy of $\mathcal{N}$, we apply stochastic gradient descent with a heuristic / loss function $\mathcal{L}$, measuring how far a prediction $\mathcal{N}(x)$ for some input x differs from an expected results. Through back propagation of this loss value, we can adjust the parameters / weights of the network to fit it on the expected values [2].

PINN are a sub-type of neural networks, that incorporate physical laws into the learning process. They use a loss function derived from differential equations, ensuring that the predictions comply with the physical properties of a system [8].

3 Simulating Incompressible Flow

In this section, we give an overview of the software and tools used for our approach. Next, we introduce our test setup, including experiments, evaluation and trainings mesh.

3.1 PyTorch

We use the PyTorch library in Python to implement the general layout of a shallow neural-network. In contrast to the implementation of Tensorflow of Raissi et al. [8], we only use default modules from PyTorch, as outlined in Fig. 3, and only need to provide an additional callback to calculate the loss. This heuristic / loss value of the current state of the network is calculated in `callback` by evaluating the network on a fixed set of (boundary) values and applying a metric, which will be discussed in the following sections. Note, that the only difference between a supervised and an unsupervised approach are the values used during this comparison. To learn a supervised set of given values, one might simply calculate the error to these in a supervised manner, while we approach an unsupervised method by fixing only boundary values. Therefore, we might try to minimize the loss, so that the network satisfies a physical condition or, in this case, differential equation, like (6) or (5).

```python
import torch
from torch import nn

model = nn.Sequential()
model.append(nn.Linear(2, 150, bias=True))
model.append(nn.Tanh())
model.append(nn.Linear(150, 150, bias=True))
...
model.to('cuda')
optimizer = torch.optim.LBFGS(model.parameters())

def callback():
    optimizer.zero_grad()
    ... = model.forward(...)
    loss = ...
    loss.backward()
    return loss

model.train()
optimizer.step(callback)
```

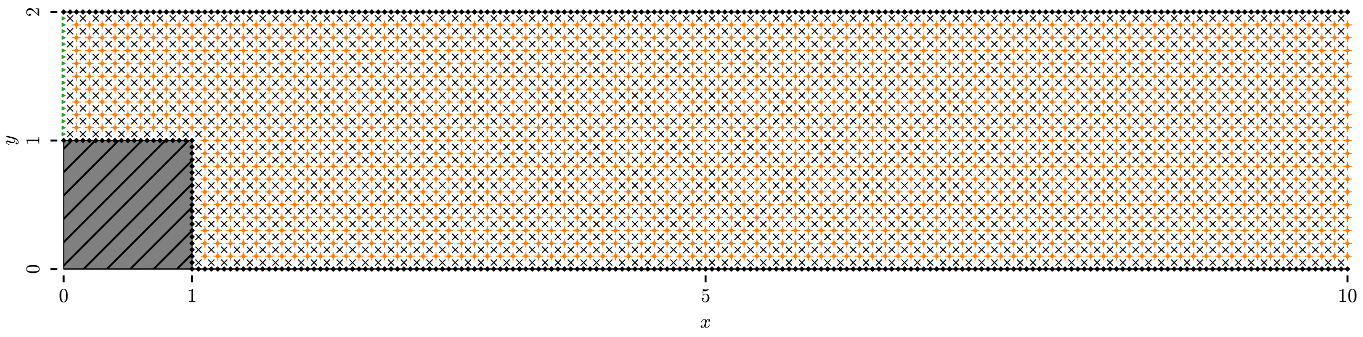
Fig. 3: *Building and training a shallow neural network with* PyTorch. *Implementation-details are omitted, but full source code is available on* GitHub.

Note, that the limited-memory BFGS optimization algorithm is directly available in PyTorch and does not require an additional import.

3.2 Mesh

Every of our test cases takes place in a rectangular area with an inlet on the left side, and an outlet on the other end. Depending on the experiment, different obstacles are placed in-between.

Fig. 4 visualizes the structure of `step` with points for training, boundary and evaluation. The experiments are discretized to a mesh of 20 times 100 squares. All squares are uniform in shape and size. During training, we evaluate the model on the intersections of those squares, but omit points inside obstructions or outside the pipe. Additional points, for a higher density and purposely accuracy, are placed on boundaries of the pipe and obstructions. For evaluation, we predict values at the center of each box, identical



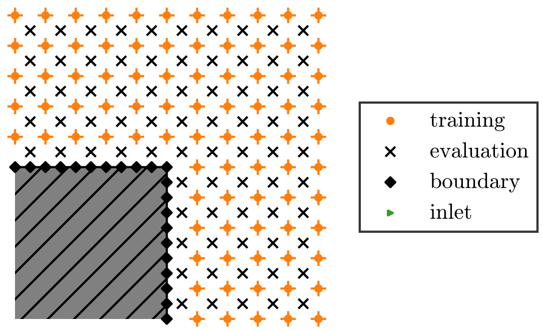


Fig. 4: *Layout of* `step` *experiment (top) with a zoom in near the step to better show the points for training, evaluation and boundary (bottom).*

to the finite-elements mesh of OpenFOAM, making the resulting velocities directly comparable.

3.3 Burgers-Equation

Similarly to the work of Raissi et al. [8], we use a PINN to predict the velocity u at coordinate x and time t. Similar to (7), the vector field is described by $\mathcal{N}^*$.

$$\mathcal{N}^* : (t, x) \mapsto u(t, x) \tag{8}$$

In addition, we define the mean squared error σ.

$$\sigma(a, b) = \frac{1}{n} \sum_{i=1}^{n} (a_i - b_i)^2 \tag{9}$$

From (6), we derive a term f^*, which must be satisfied, making it a candidate to be used in the loss function $\mathcal{L}^*$ modeling this system.

$$f^* := \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial^2 x} \overset{!}{=} 0 \tag{10}$$

We calculate $\mathcal{F}^*$ as the mean-squared error (9) of f^* over all mesh coordinates. $\varDelta\hat{u}^*$ is the mean squared error at boundary points over space and time. $\mathcal{L}^*$ is the sum of those values.

$$\mathcal{L}^* = \mathcal{F}^* + \varDelta\hat{u}^* \tag{11}$$

3.4 Navier-Stokes Equation

While 3.3 refers to the preexisting work of Raissi et al. [8], we want to improve on the one-dimensional use-case by applying unsupervised learning to the two-dimensional NSE. Raissi et al. showed, that a a simulation can be trained in a supervised manner [8] yielding a high accuracy prediction of the flow around a cylinder. For the scope of this work we only look into non-time dependent solutions, so that we can ignore the time gradient of (5).

Similar to 3.3, we need to formulate a heuristic for the accuracy of the simulation. The value of the the loss $\mathcal{L}$ indicates how good our network is approximating (5). From (5) we can derive two functions, $f(x,y)$ and $g(x,y)$, which both should be to zero for all points (x,y), so that $\mathcal{N}$ satisfies the properties inherited from (5). Since we use a discrete mesh, we calculate $\mathcal{F}$ and $\mathcal{G}$ by the mean-squared error (9) of f and g over all mesh coordinates.

$$\begin{aligned} f &:= \left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) - \nu\left(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y}\right) + \frac{\partial p}{\partial x} \overset{!}{=} 0 \\ g &:= \left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) - \nu\left(\frac{\partial^2 v}{\partial^2 x} + \frac{\partial^2 v}{\partial^2 y}\right) + \frac{\partial p}{\partial y} \overset{!}{=} 0 \end{aligned} \tag{12}$$

We define $\varDelta\hat{u}$ and $\varDelta\hat{v}$ as the mean squared errors (9) of predicted velocities compared to expected values at boundary coordinates. Therefore, we get an formulation of the loss function $\mathcal{L}$.

$$\mathcal{L} = \mathcal{F} + \mathcal{G} + \varDelta\hat{u} + \varDelta\hat{v} \tag{13}$$

3.5 Experiments

For testing our implementation, we created several interesting layouts to try to predict flow with different obstructions in a pipe.

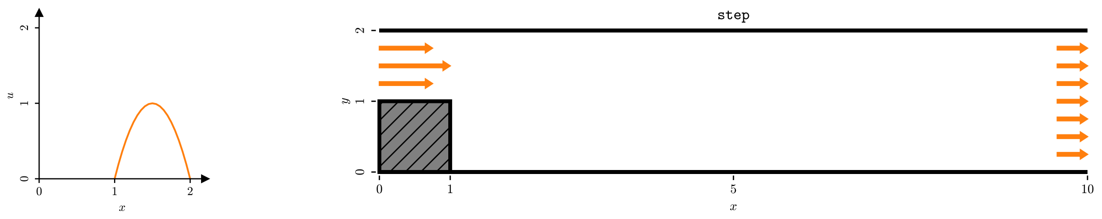


Fig. 5: *Geometry of* `step` *experiment (right) with obstruction, indicated by hatched area, and parabolic inlet-function (left).*

Exemplary, Fig. 5 shows the two-dimensional geometry of `step` with its parabolic inlet function. We choose to model the inflow as parabola instead of a constant value to reduce potential side-effect of at the intersection of inlet and border through non-differentiable behavior, since this function also assumes zero-velocity at those points. The same modeling is used by Visconti et al. in their book [11]. We sill use `step` to investigate hyper-parameters of our simulation due to its geometrical simplicity and well-researched flow function.
For calculating the loss $\mathcal{L}$ during training we use the mesh proposed in 3.2 with setting border and inlet accordingly. Since we use an arbitrary inlet function, we denote the maximal value of u over all inlet coordinates as u_{in}. For the comparison against OpenFOAM, we use the evaluation grid respectively. To get the Reynolds number of an experiment, we use the width of the inlet as characteristic length L. Note, that most experiments have $L = 2$, so that we have to scale ν and u_{in} accordingly.

The border and obstructions are modeled as a fine grained line of static flow (zero velocities in all directions) boundary points, which are compared against in the loss function.

For Simulating Burgers' Equation, we use the same setup, but replace (x, y) with (t, x) in the geometry of `empty` and normalize the length in both directions.

4 Evaluation and Results

In this chapter, we visualize and investigate the properties of our simulation.

4.1 Reference Model

To help us grasp the efficacy of our model, we compare it to the same setup simulated in OpenFOAM 2.2. In OpenFOAM the area gets divided into a number of blocks, modeling the different experiment setups. Tis is shown in Fig. 6. Those Blocks are than further subdivided into the final grid, giving us a mesh of 20x100 finite elements, identical to the evaluation grid of our PINN. Fig. 7 shows the finite elements of the OpenFOAM simulation in ParaView. For a higher simulation resolution, this subdivision can be increased. Our experiments called for an inlet with a parabolic shape, allowing a more natural inlet flow compared to a uniform inlet. We achieve this in OpenFOAM by utilising a precomputed set of inlet values. For simulating the experiments, the SimpleFOAM solver is used without a a turbulence model to ensure a stable solution. While ParaView work great for visualizing the simulation results, we cannot compare simulation and prediction inside the tool. Therefore we found it necessary to read the data directly from OpenFOAM and convert them to the same plotting format used in our model evaluation.

OpenFOAM runs on a *Intel Core i7-1065G7* under *Microsoft Windows 11.*

4.2 First Steps

While we originally planed to build upon the implementation of Raissi et al. [8], we quickly noticed, that we were unable to adapt the code to a more recent version of Tensorflow or run it on a GPU due to issues with the library. Instead, we opted to re-implement the network using PyTorch based on other adaption of this approach [1]. To our astonishment, on one hand PyTorch is significant faster on a CPU compared to Tensorflow, while it only requires an available CUDA-compiler to run on a GPU. Therefore, computation-intensive tasks, like optimization of a neural network,

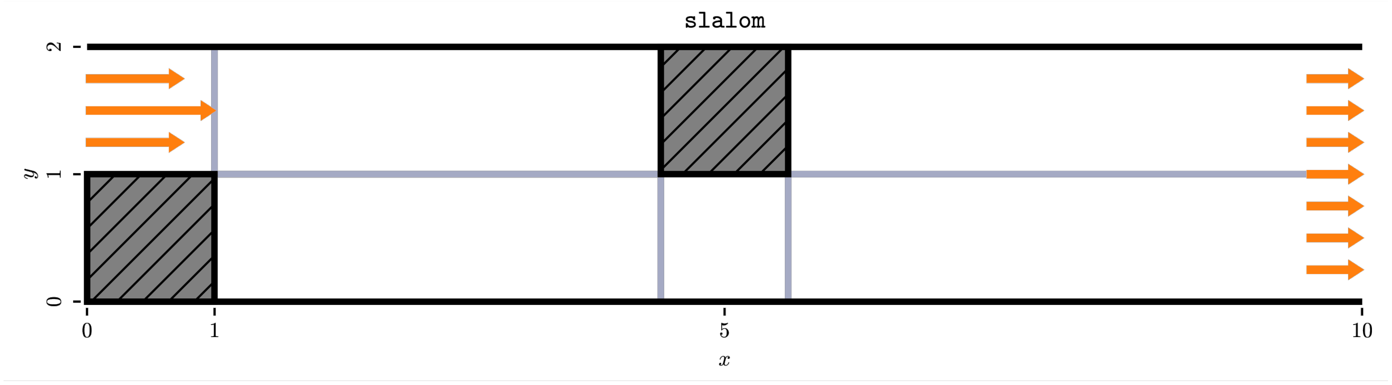


Fig. 6: *Division of* `slalom` *into in number of blocks, as used by Open-FOAM.*

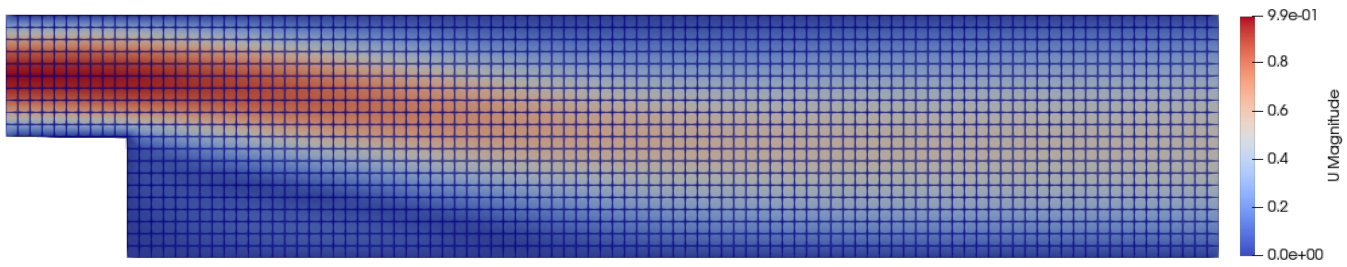


Fig. 7: *A visualisation of the finite element mesh of* `step` *for Open-FOAM, shown in ParaView.*

can be run on specialized hardware. The evaluations in this and following sections are are exclusively run on a *NVIDIA GeForce RTX 3080* and an *AMD Ryzen 9 5950X* under *Microsoft Windows 10*.

First, we focus on the replication process of the supervised Physics-Informed Neural Networks (PINNs) approach from Raissi et al. using PyTorch as presented in Fig. 8. We set up a boundary values similar to Raissi et al. [8] with a sinus fucntion as inletand use the same number of layers and nodes.

The results are very similar to the values obtained by the reference paper, making our approach viable to be further investigated.

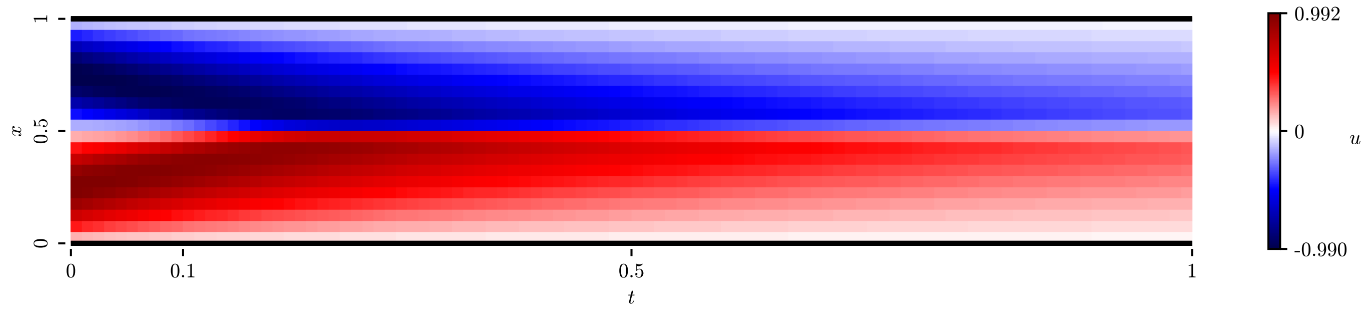


Fig. 8: *Prediction of Burgers' equation.*

4.3 Hyper-Parameters

To get an understanding of the properties of our network, we tried to correctly estimate the flow function of `step` for Re = 100 ($\nu = 0.01$, $u_{in} = 1$), compared to the prediction of OpenFOAM, while changing the hyper-parameters of our simulation. The evaluation matrix consists of different layer sizes of 25 to 200 and two to four hidden layers. We tried to optimize each simulation until the loss function converges to a change of only 5e-12 between two consecutive optimization steps. Note, that we use 64-bit floating point numbers, otherwise this precision would no be achievable.

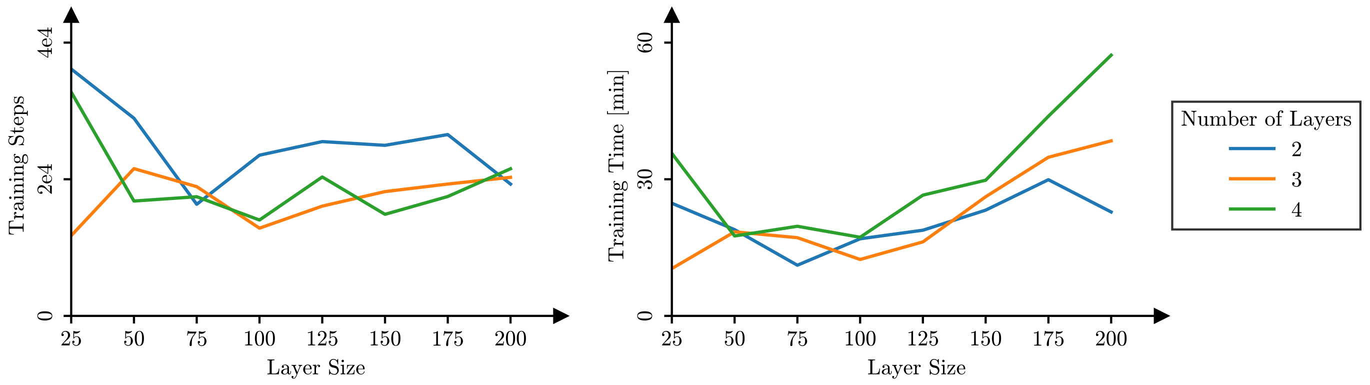


Fig. 9: *Optimization steps and training time until convergence for different layer sizes and network depths.*

We plotted the required steps and time to train the network in Fig. 9. As expected, increasing layer size and depth of the network leads to an increase in training-time, but interestingly does not yield a big impact on the number of optimization steps.

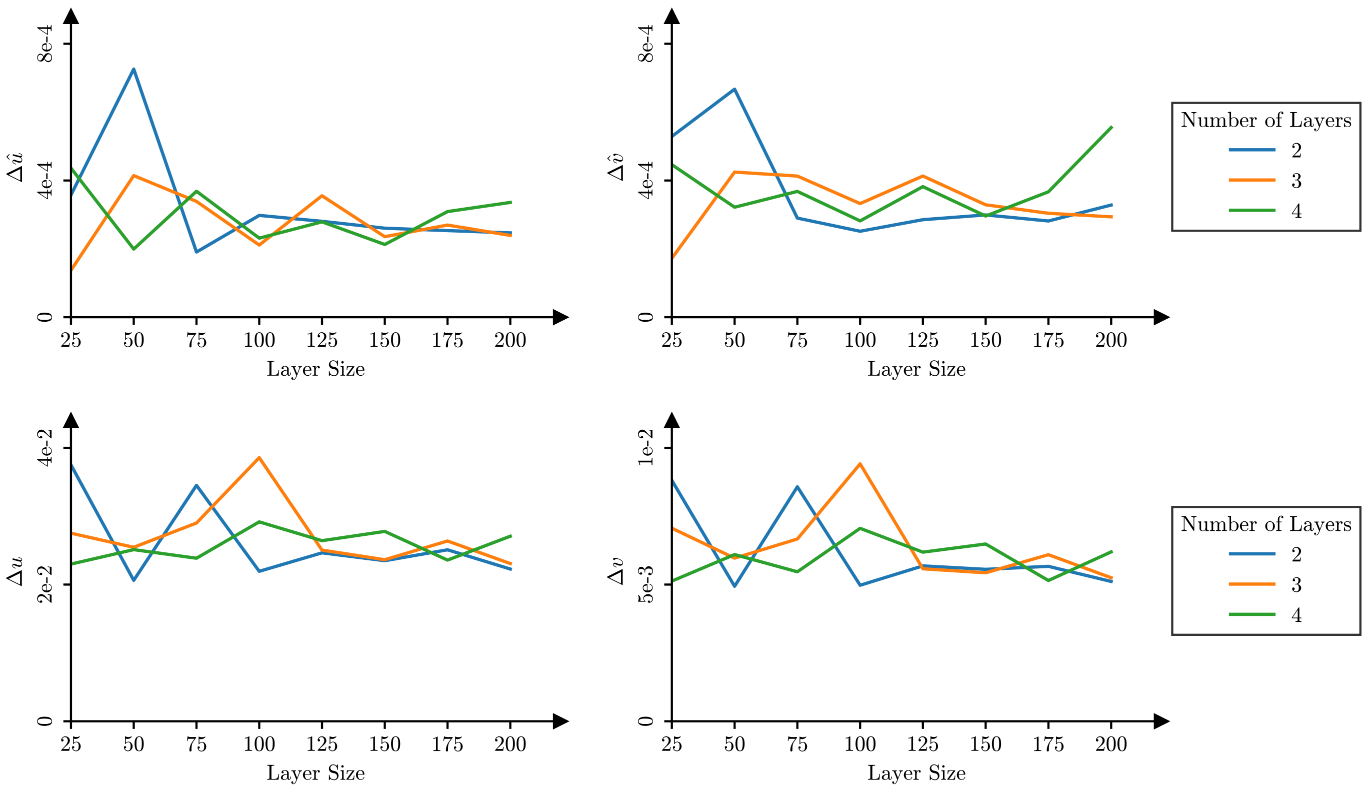


Fig. 10: *Absolute error at evaluation and boundary points of horizontal and vertical velocity, denoted by Δu, $\Delta\hat{u}$ and Δv, $\Delta\hat{v}$ respectively for different layer sizes and network depths. The predictions are compared against the expected value from an* OpenFOAM *simulation with the same setup.*

Plotting the comparison to OpenFOAM for boundary values and mesh points in Fig. 10, we observe, that until 125 nodes per layer, we see fluctuating accuracy for different numbers of layers. For even larger layers, this effect decreases, suggesting, that the overall higher number of weights allows the network to fit to the problem more efficiently.

To choose an optimal candidate to further investigate the capabilities of our simulation, we are looking into how errors compare to maximal training time. The cloud of simulations with different hyper-parameters presented in Fig. 11 allows us to determine, which setup has the most robust capabilities.

We saw in Fig. 10 better predictions with more than 125 nodes, which can also be concluded from Fig. 11 if we look into which simulations compare best in regard to robustness against over-fitting. A minimal training time of a singular evaluation, while the network performs significantly worse with different numbers of layers, is considered not robust. We determined the threshold at 125 nodes per layer and chose a network with three layers of size 150 nodes for further investigation and to be used in the next part of this paper. The predicted streamlines of this network are shown in Fig. 12, as well as the absolute difference in velocities compared to OpenFOAM in Fig. 13. We see some border artifacts, especially above the step, probably resulting from the way we model the boundary. Similar to the conclusion of the supervised approach [8], we are also able to model the correct pressure field, but focus on the velocity field for the sake of easier comparability and relation to the flow function.

4.4 Predict Incompressible Flow

Based on the results from the previous part, we decided to use a feed-forward neural network with three hidden layers, each with 150 nodes, and 46,052 trainable parameters overall. To determine the capabilities of our setup to predict the stream function in a general context, we evaluate several unique experiments. The full list can be found as appendix to this paper and on GitHub. We compared each experiment given five different values of u_{in} and ν with the predicted velocities from OpenFOAM.

In addition, we looked into how the prediction improves during training of the network. For this aspect, we looked into the error after only 1000 optimization steps. The results are presented in Tab. 1.

We observe, that we are able to predict simple geometry, like `empty` and `step`, within a small margin of error. On the other hand,

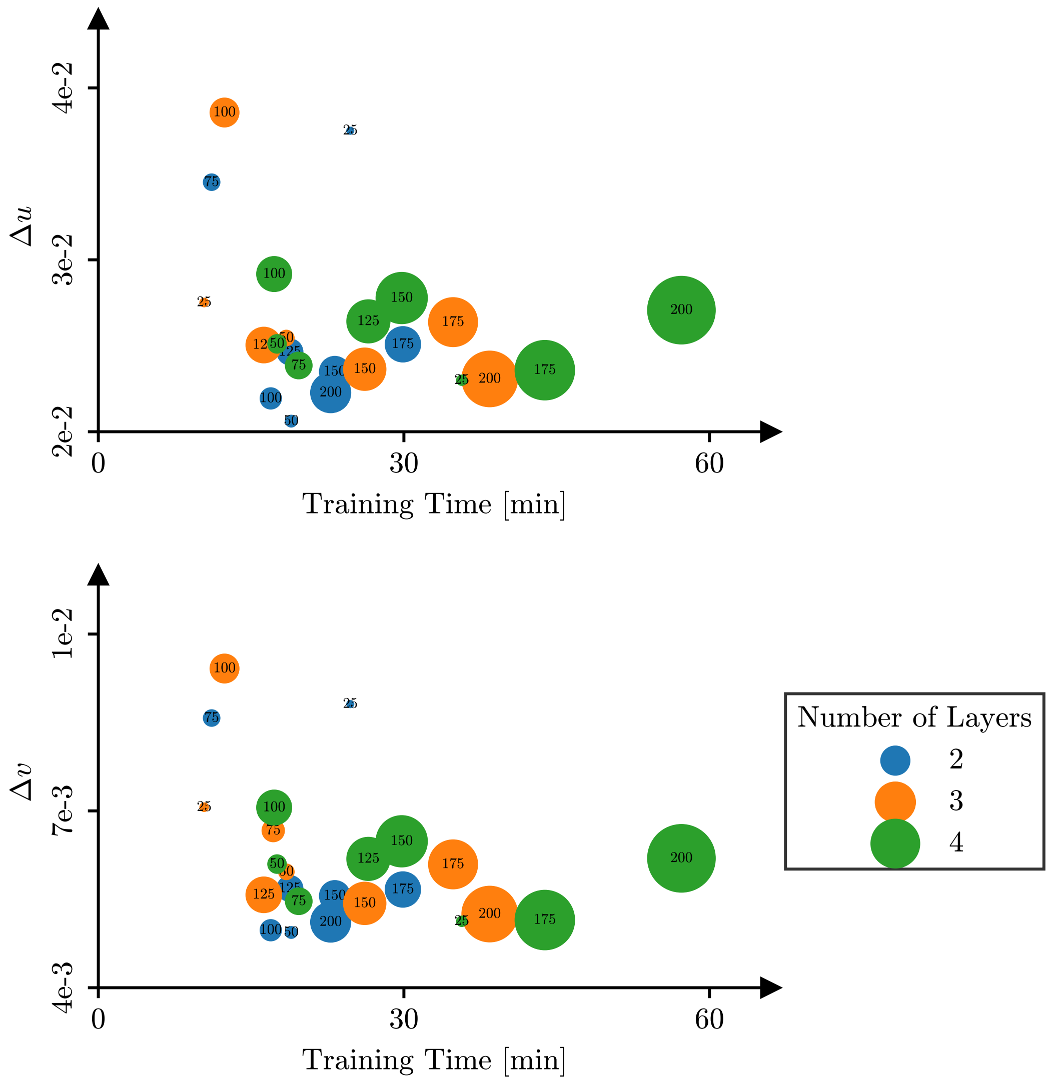


Fig. 11: *Prediction errors against training time. Sizes of nodes indicate number of parameters of corresponding neural network, including weights and biases.*

Experiment	ν	u_{in}	L	Re	Optimization Steps	Training Time	Δu	Δv
empty	0.02	0.5	2	50	1000	0:01:23	2.274e-3	2.559e-4
					1491	0:02:05	2.305e-3	1.865e-4
	0.01	1	2	200	1000	0:01:22	6.264e-3	5.952e-4
					2662	0:03:39	6.008e-3	2.838e-4
	0.02	1	2	100	1000	0:01:23	4.793e-3	4.449e-4
					2806	0:03:51	4.208e-3	2.339e-4
	0.02	2	2	200	1000	0:01:23	1.096e-2	3.324e-3
					5379	0:07:20	7.533e-3	6.784e-4
	0.04	1	2	50	1000	0:01:23	3.347e-3	3.922e-4
					4116	0:05:32	3.183e-3	1.630e-4
step	0.01	0.5	1	50	1000	0:01:23	1.583e-2	5.005e-3
					8942	0:12:12	8.702e-3	2.623e-3
	0.005	1	1	200	1000	0:01:22	1.272e-1	1.854e-2
					16666	0:24:14	6.675e-2	1.211e-2
	0.01	1	1	100	1000	0:01:22	6.449e-2	1.493e-2
					18192	0:26:09	2.364e-2	5.431e-3
	0.01	2	1	200	1000	0:01:23	2.433e-1	3.665e-2
					34455	0:50:22	9.690e-2	1.768e-2
	0.02	1	1	50	1000	0:01:22	3.117e-2	9.728e-3
					15876	0:23:19	1.751e-2	5.720e-3
slalom	0.01	0.5	1	50	1000	0:01:08	7.998e-2	1.707e-2
					31474	0:35:58	1.866e-2	6.698e-3
	0.005	1	1	200	1000	0:01:09	1.627e-1	5.421e-2
					49172	0:58:15	2.295e-1	6.369e-2
	0.01	1	1	100	1000	0:01:11	1.665e-1	4.151e-2
					26413	0:31:10	1.436e-1	3.732e-2
	0.01	2	1	200	1000	0:01:10	4.118e-1	1.098e-1
					96022	1:51:06	5.347e-1	1.320e-1
	0.02	1	1	50	1000	0:01:09	2.283e-1	4.430e-2
					30121	0:34:32	6.244e-2	1.724e-2
block	0.02	0.5	2	50	1000	0:01:22	4.325e-1	5.319e-2
					48201	1:06:29	2.994e-2	1.038e-2
	0.01	1	2	200	1000	0:01:25	8.058e-1	1.178e-1
					48675	1:07:46	2.351	2.329e-1
	0.02	1	2	100	1000	0:01:24	8.997e-1	1.108e-1
					168226	3:54:18	4.173e-2	1.771e-2
	0.02	2	2	200	1000	0:01:24	1.840	2.455e-1
					347741	8:05:43	3.034e-1	7.857e-2
	0.04	1	2	50	1000	0:01:22	9.592e-1	1.135e-1
					105224	2:27:25	5.483e-2	2.259e-2

Tab. 1: *Results of evaluation series to determine capabilities of our setup in regard to the accuracy of predicted flow within different geometries. Outliers are highlighted.*

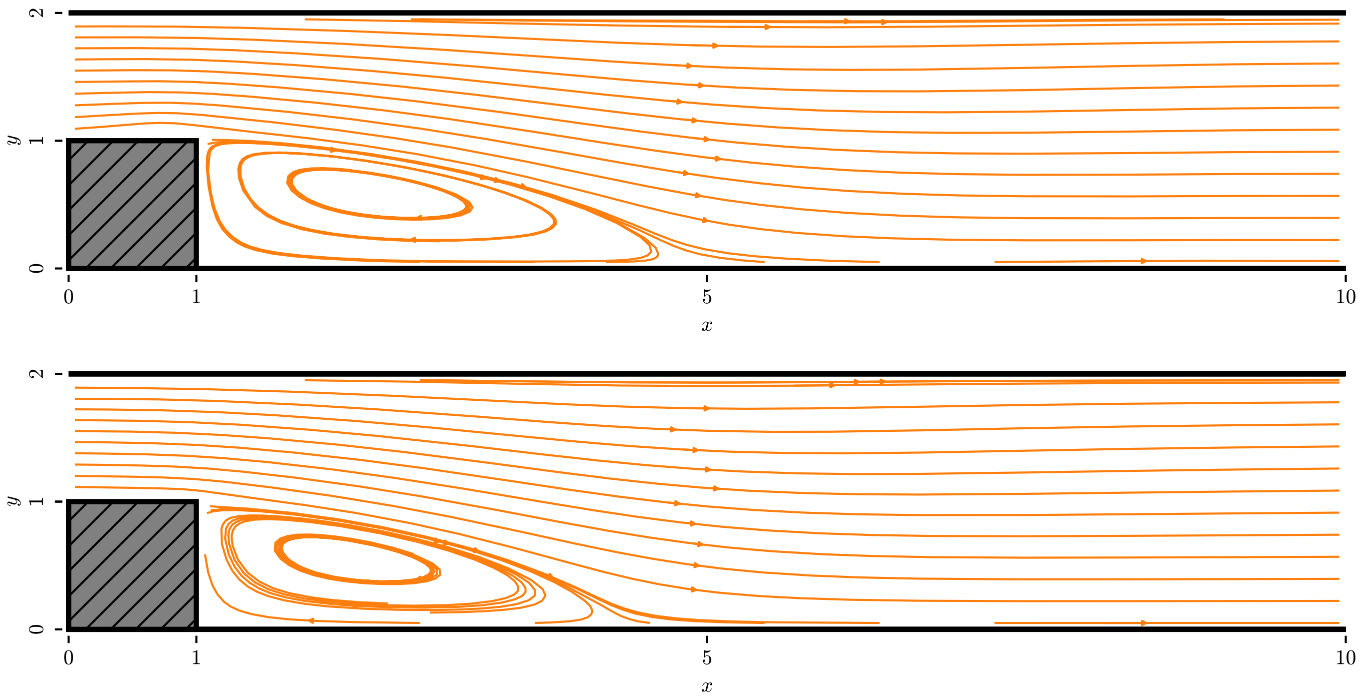


Fig. 12: *Comparison of predicted streamlines. The top image shows our prediction of* ***`step`*** *with Re* $= 100$ *(*$u_{in} = 1$*,* $\nu = 0.01$*). The bottom picture is the reference generated with* OpenFOAM *on the same parameters.*

more complex obstructions, like `slalom` and `block`, increase the error compared to OpenFOAM and may diverge and result in failure to predict meaningful results altogether. In contrast, using OpenFOAM we were able to predict the velocities for each case within seconds. Regarding our training time, we notice, that the time until the optimization converges is roughly proportional to u_{in} This is explainable, since the initial loss is larger for higher velocities, as we calculate the absolute error of those values compared to zero for most mesh points, increasing the loss value, thus making it harder to converge.

However, we noticed, that the expected error after only 1000 iterations is already small and in a similar order of magnitude of the final value, which can be observed in Fig. 14. Plotting the loss on a

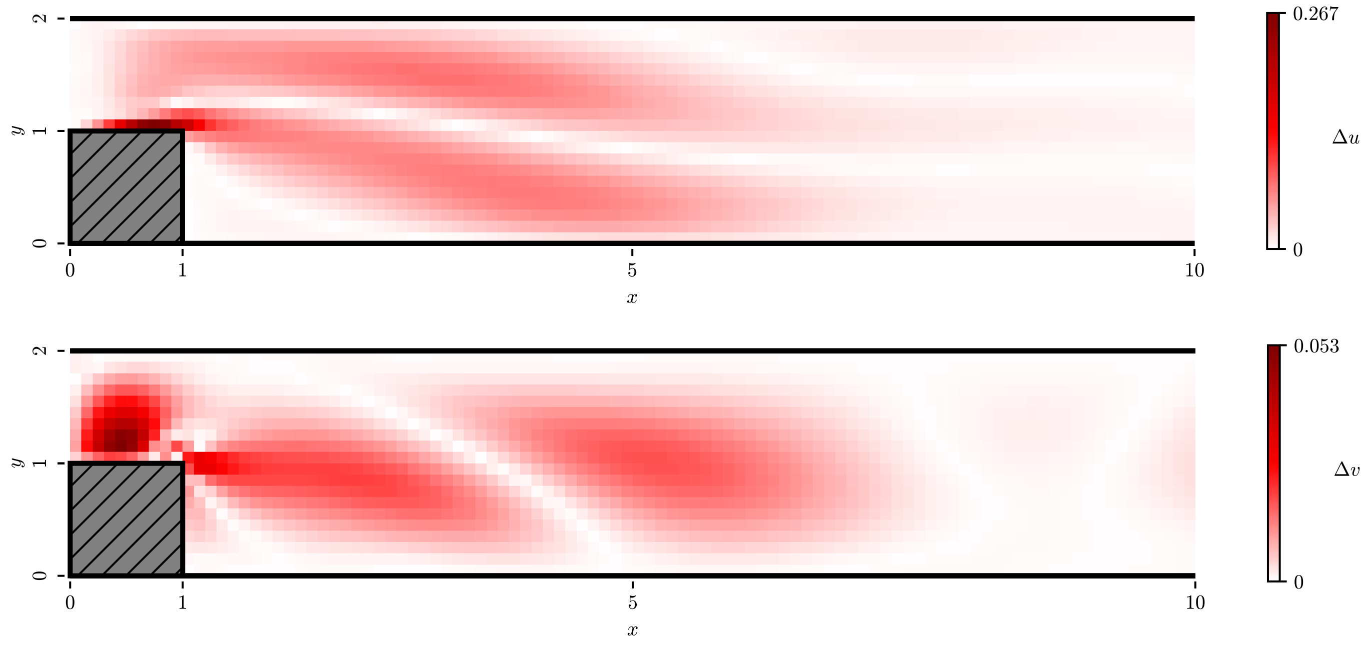


Fig. 13: *Difference between prediction and reference for the flow in horizontal direction u (top) and vertical direction v (bottom). We simulated $Re = 100$ ($u_{in} = 1$, $\nu = 0.01$).*

logarithmic scale, in the beginning the loss changes dramatically, but only slightly improves after those initial optimization of the network. Changing viscosity only has no predictable impact on the speed of convergence of $\mathcal{L}$, but low-velocity fluid might be harder to simulate.

The resulting streamlines of `block` shown in Fig. 15 using $u_{in} = 1$ and $\nu = 0.01$, which means, the simulation of a low-viscosity fluid ($\mathrm{Re} = 200$) results in a totally wrong prediction. In general terms, we see relatively good prediction for $\mathrm{Re} = 100$ for all experiments, while lower and especially higher viscosity lead to worse approximations. Unfortunately, Δu and Δu can not be solely used as metric for the accuracy of a prediction, but give an indication of the archived accuracy. Accommodating, we have to evaluate the graphical plots of streamlines and velocities individually to determine, whether a simulation succeeded in predicting the physical properties of flow in the system. This is evident for `block`, which is only well predictable for

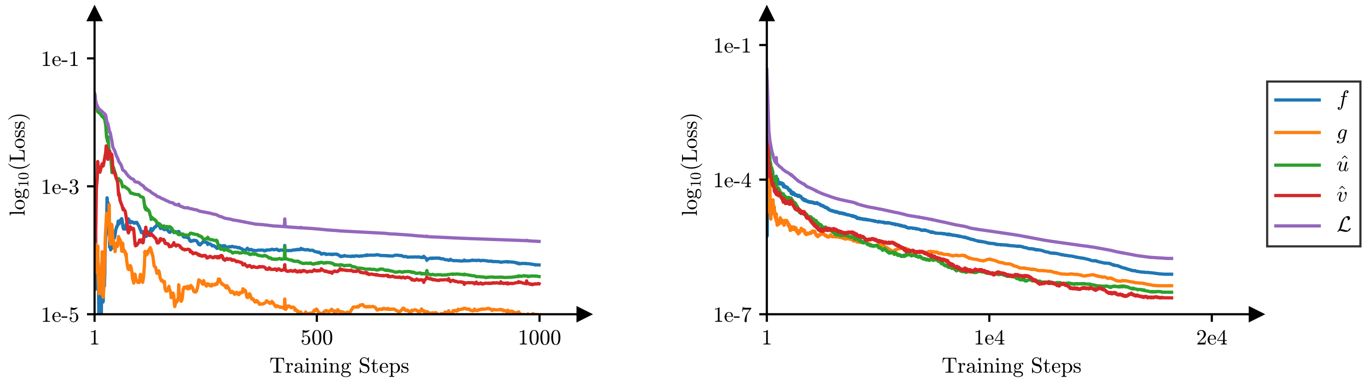


Fig. 14: *Loss. Optimization converges after 18192 steps.*

Re = 100, if compared to OpenFOAM. For `block` we also notice, that the reflow is slightly shifted to the right, which can be observed in Fig. 12, so that the prediction seems stretched along the direction of the flow.

To close this section on a positive note, we show the general capabilities of the simulation for the common use-cases of flow around a cylinder and a NACA airfoil [4]. Fig. 16 shows our predicted streamlines, which seem close to the expected results [11][12].

5 Discussion

Asking whether we can reproduce the results of Raissi et al. [8] for RQ1, we have shown in 4.2, that we can predict Burgers' equation in a similar manner. We were also able to scale down the complexity of the required implementation by using default modules of PyTorch, while maintaining high accuracy of the simulation.

For RQ2, we extended the setup to look into the prediction of a stationary flow function for several interesting problems using an unsupervised learning approach. We observed in 4.3, that for simple geometries, like `step`, the accuracy of our prediction is very high compared to SOTA tools such as OpenFOAM. Therefore, we can answer RQ2.1, such that we can generalize the unsupervised learning approach to two-dimensional problems and simulate the velocity (and pressure) field for selected experiments. Unfortunately, we no-

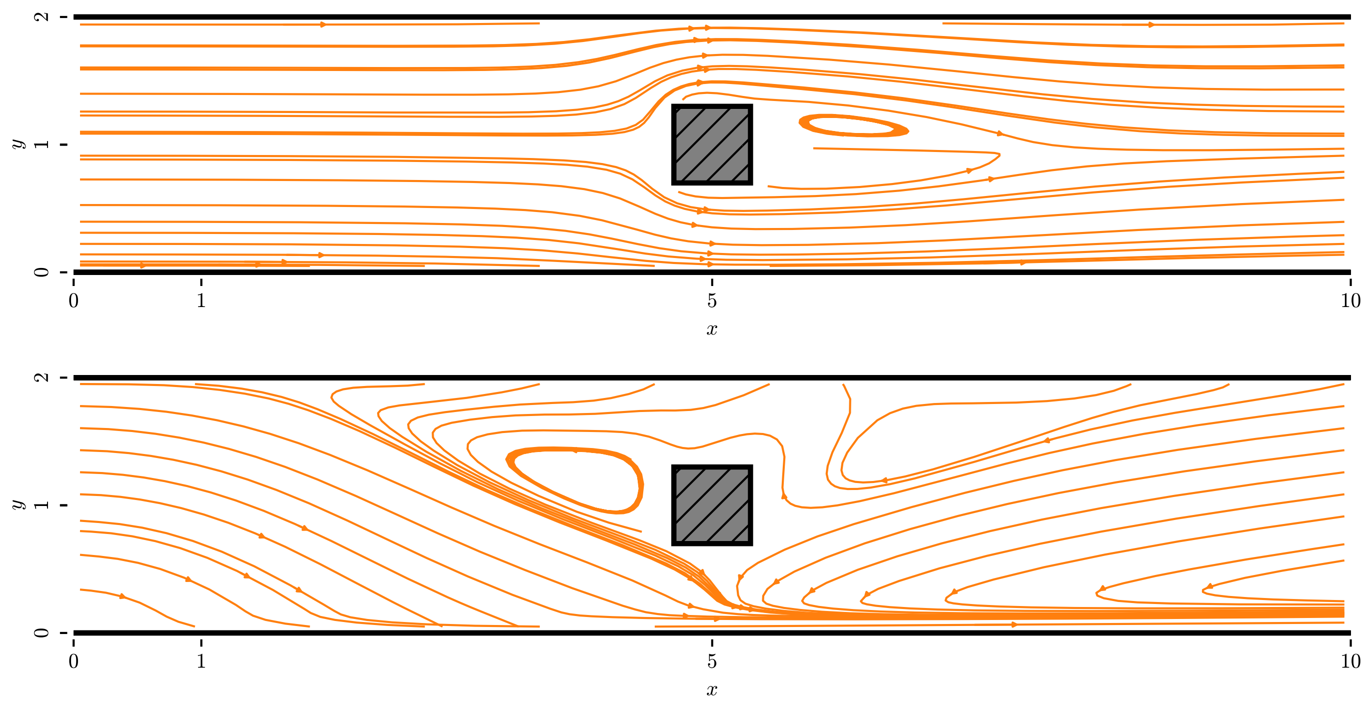


Fig. 15: *Successful (top) and failed (bottom) prediction of* `block` *for* $Re = 200$*. Higher velocity* $u_{in} = 2$ *and viscosity* $\nu = 0.02$ *yield result, while* $u_{in} = 1$ *and* $\nu = 0.01$ *fail.*

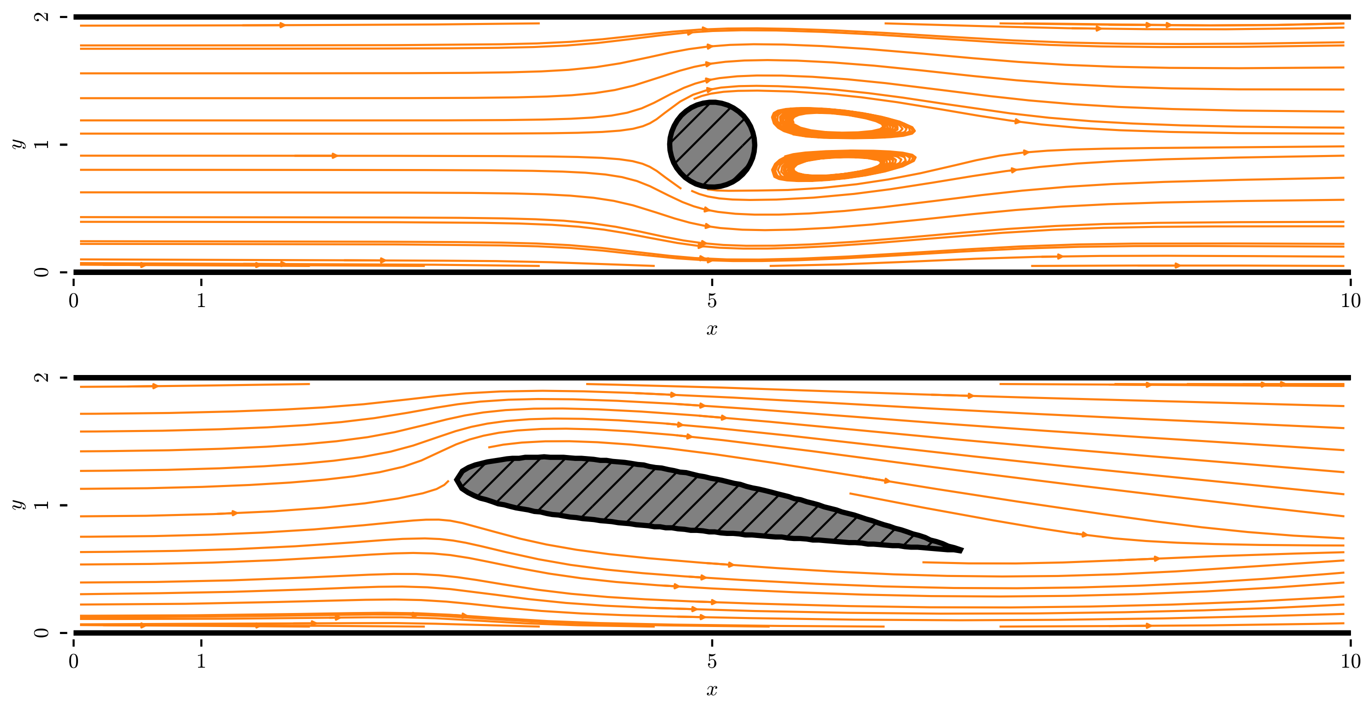


Fig. 16: *Predicted streamlines around amorphous (non-orthogonal) geometry. We simulated* **`cylinder`** *(top) and* **`wing`** *(bottom), both with* $u_{in} = 1$ *and* $\nu = 0.02$ *(Re* $= 100$*).*

ticed that our model of inlet, boundary values and mesh is prone to artifacts, and must be improved upon needing further investigation. As our simulation can be trained on any PDE with slight modifications, we see this as an achievement regarding RQ3, so that machine learning, especially shallow neural networks, are fully capable of being used for approximating PDE and problems from physics, chemistry and similar fields of research. In hindsight, SOTA tools, like OpenFOAM, are at this point far more capable with respect to performance and robustness than our approach, having been developed for decades. However, since Machine Learning and specialized hardware are evolving at a rapid rate, we can expect significant improvements upon our approach and the use of PINN in an unsupervised manner might become competitive [12]. Therefore, we can answer RQ3.1, such that while SOTA tools have vastly superior performance characteristics, we are able to achieve the same accuracy for selected experiments, as shown in 4.4, We are confident, that tuning the hyper-parameters and heuristics will lead to better performance. An import aspect of the fitted network is, that it models a (continuous) vector field, whereas other approaches need a space and time discretization.

5.1 Threats to Validity

As our entire comparison is based on the accuracy of the reference model of OpenFOAM. Therefore, should the reference model be proven to be obsolete or faulty, the accuracy of our model can be called into question. Due to OpenFOAM's reputation and standing as SOTA software, the quite unlikely, but user error on our part can not be completely ruled out.

6 Conclusion and Outlook

Concluding our findings of the previous chapter, we can summarize that we successfully recreated the experiment setup used by [8] for a unsupervised approach. We were able to predict the velocity field for several use-cases and provided a simple framework, which can be used in further research.

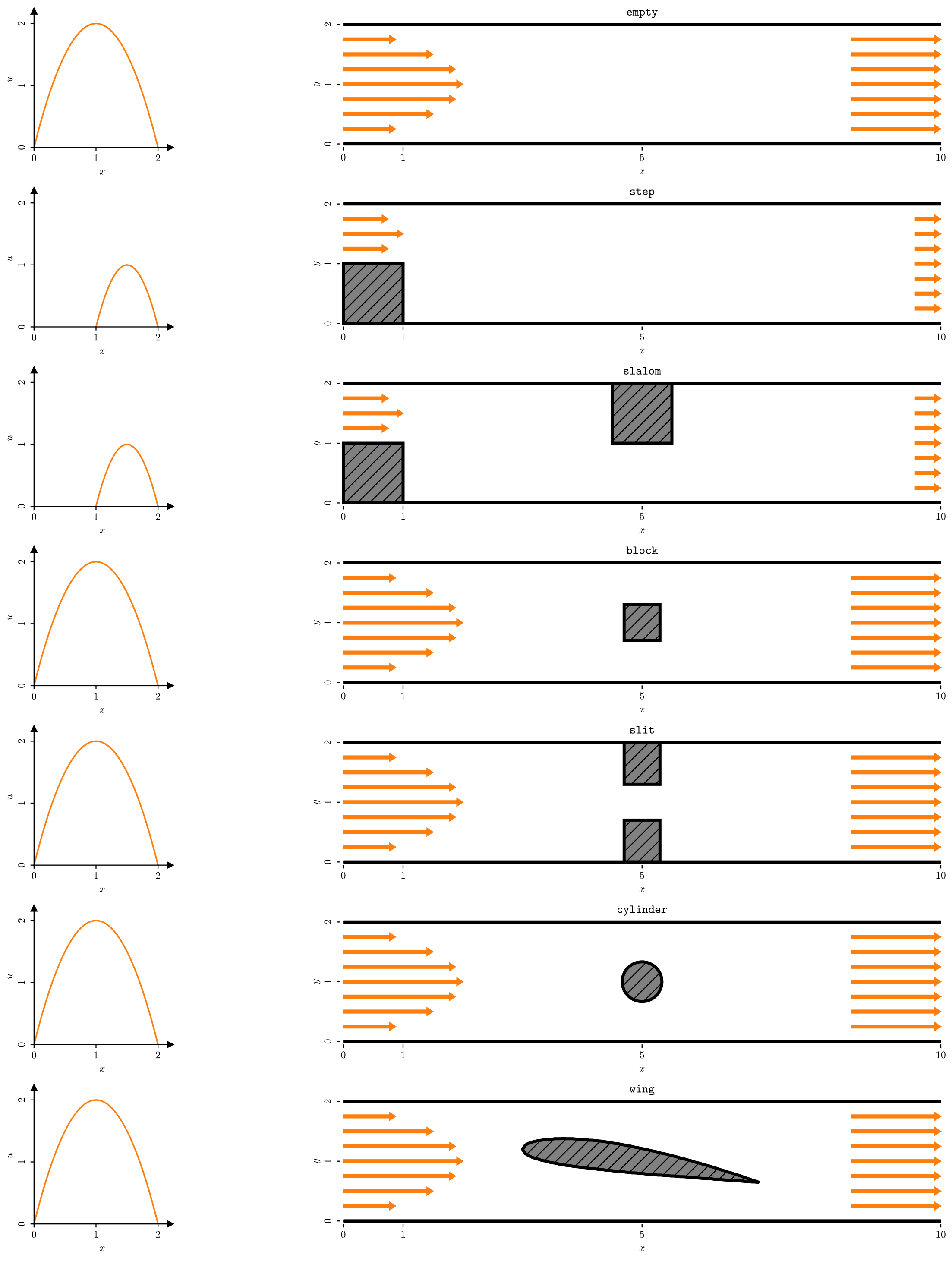
As mentioned in 5, the modeling of boundary conditions needs further investigation and improvement. For this work, we limited the mesh to a relatively coarse grid, whereas normally one would try to use a higher resolution. Unfortunately we noticed that the prediction of OpenFOAM is quite different for a finer grid, making a comparison impossible. An interesting idea would be to use a finer grid while training a similar PINN and look into whether the result change as well.

In addition, it would be interesting to improve our simulation by incorporating a turbulence model. Moreover, since the scope of our results is not limited to PINN and NSE, we look forward to important applications of machine learning in other fields of research [9].

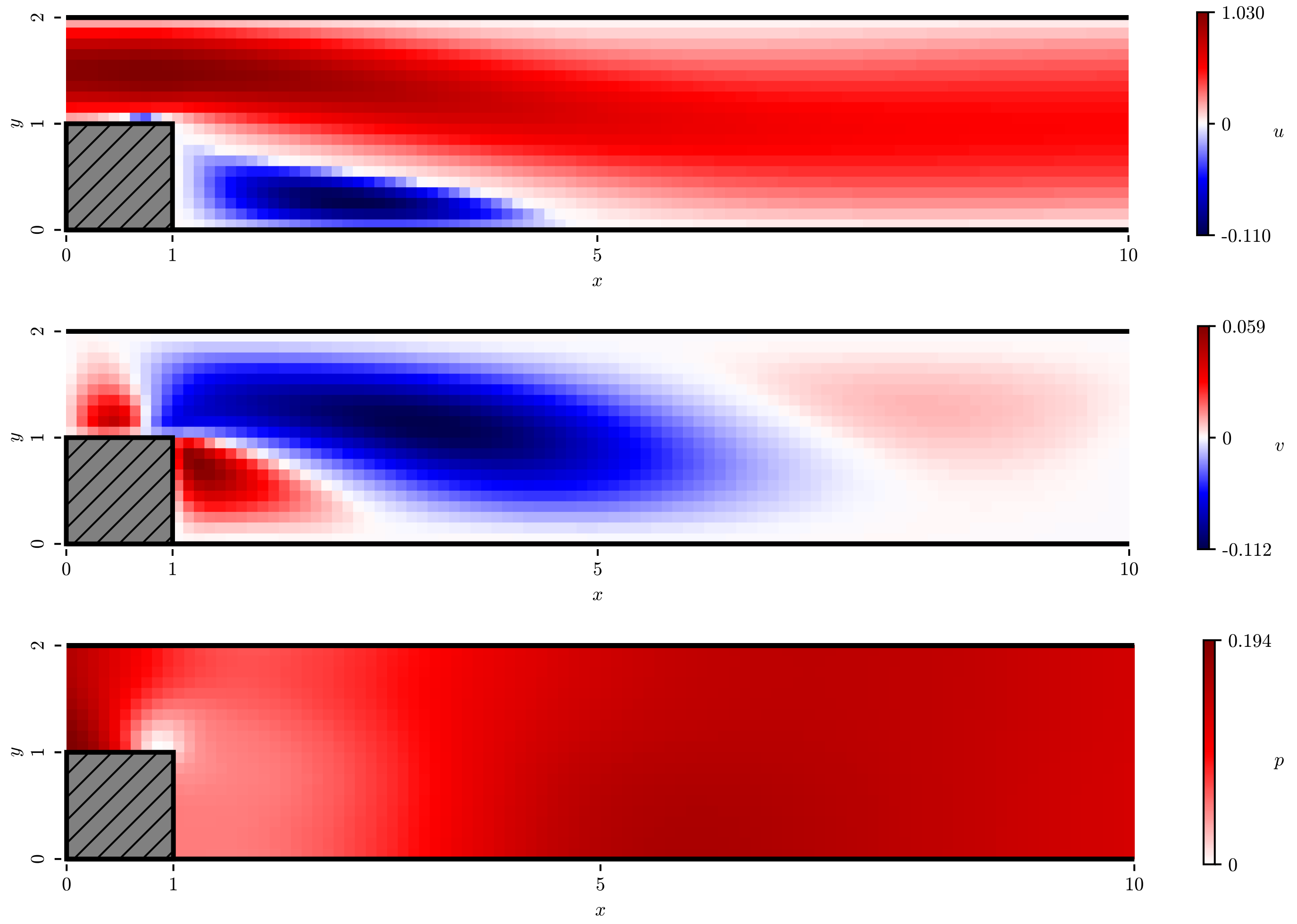
References

1. Bafghi, R.A., Raissi, M.: PINNs-TF2: Fast and User-Friendly Physics-Informed Neural Networks in TensorFlow V2 (Nov 2023)
2. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg (2006)
3. Kitware Inc.: ParaView (2002), `https://www.paraview.org/`, last accessed 23 Oct 2024
4. NASA: NACA Airfoils (2017), `https://www.nasa.gov/image-article/naca-airfoils/`, last accessed 23 Oct 2024
5. NVIDIA: Earth-2 (2024), `https://www.nvidia.com/en-us/high-performance-computing/earth-2/`, last accessed 23 Oct 2024
6. OpenCFD Ltd: OpenFOAM (2004), `https://www.openfoam.com/`, last accessed 23 Oct 2024
7. Patankar, S.: Numerical heat transfer and fluid flow. CRC press (2018)
8. Raissi, M., Perdikaris, P., Karniadakis, G.: Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics **378**, 686–707 (Feb 2019). `https://doi.org/10.1016/j.jcp.2018.10.045`
9. The Nobel Prize: The Nobel Prize in Physics 2024 (2024), `https://www.nobelprize.org/prizes/physics/2024/press-release/`, last accessed 23 Oct 2024
10. Thunberg, G.: The Climate Book. Penguin Random House (2022)
11. Visconti, G., Ruggieri, P.: Fluid Dynamics: Fundamentals and Applications. Springer International Publishing (2020). `https://doi.org/10.1007/978-3-030-49562-6`
12. Zuo, K., Ye, Z., Bu, S., Yuan, X., Zhang, W.: Fast simulation of airfoil flow field via deep neural network (2023). `https://doi.org/10.48550/ARXIV.2312.04289`

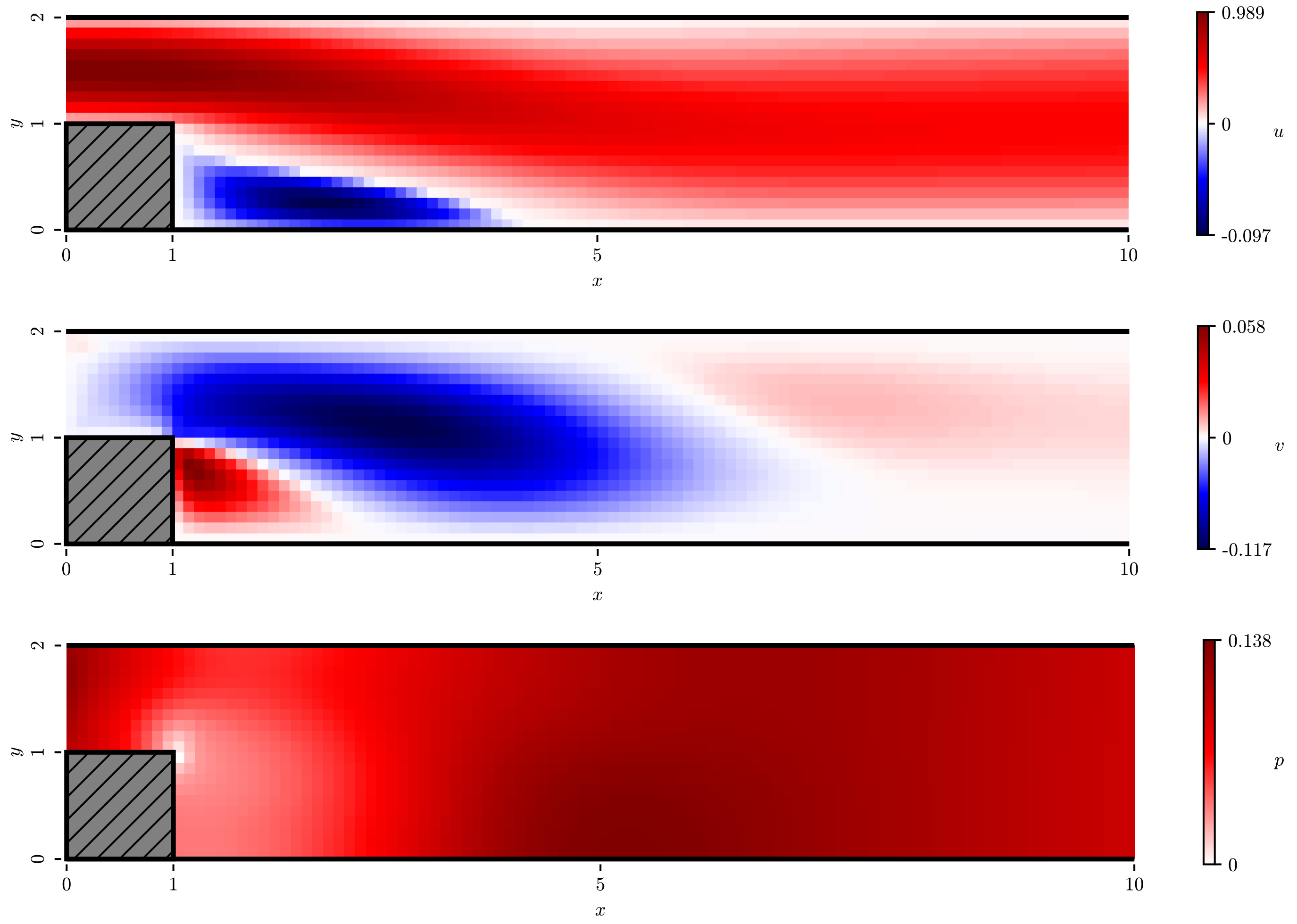
Source code available on GitHub.



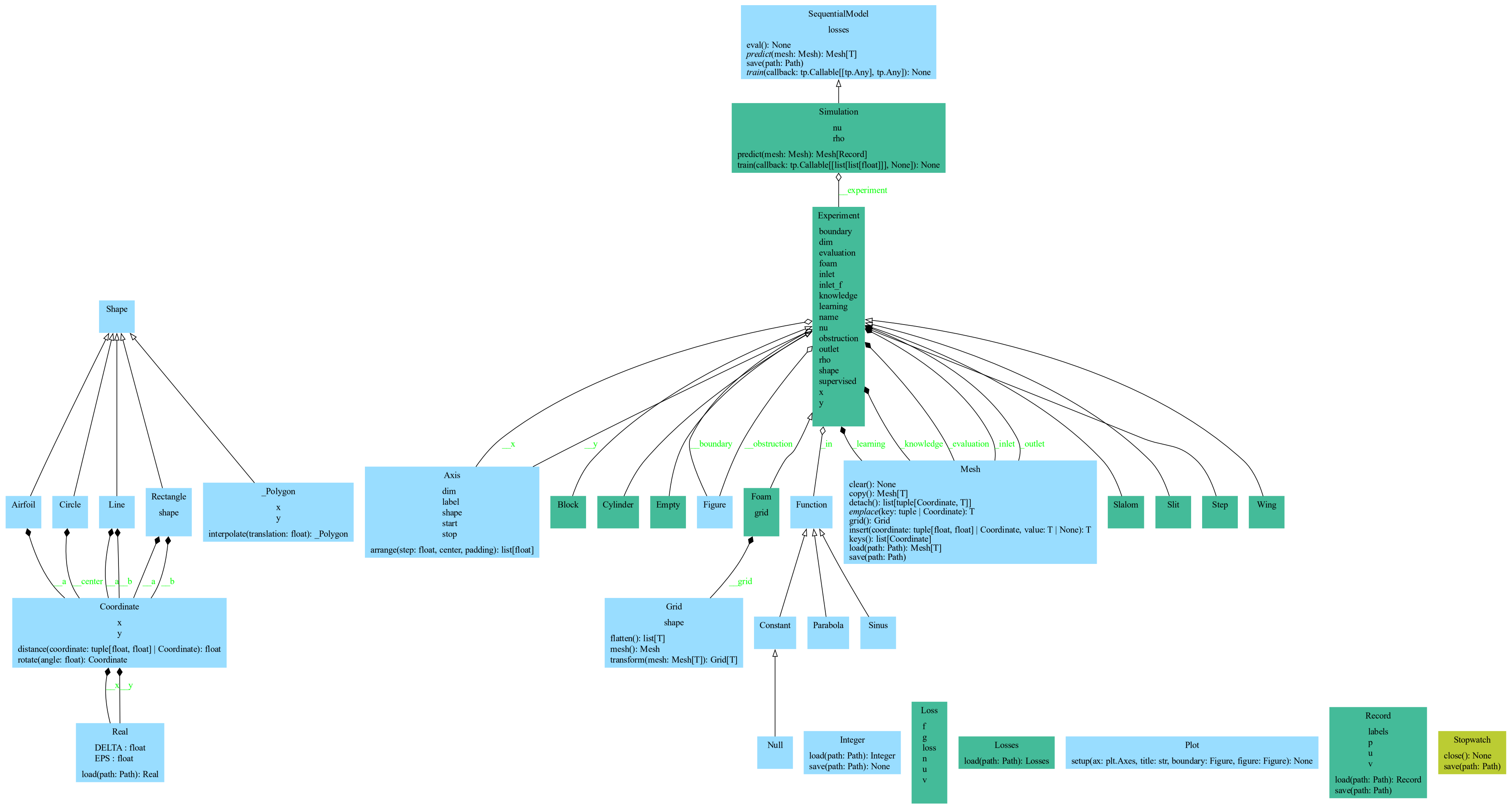
App. 1: *List of experiments with inlet function and schematic view of geometry.*



App. 2: *Prediction of* `step` *with* $Re = 100$ $(u_{in} = 1,\ \nu = 0.01)$.



App. 3: *Reference of* ***step*** *with* $Re = 100$ $(u_{in} = 1,\ \nu = 0.01)$.



App. 4: *UML Class Diagram of* Python *implementation. We only include public method and parameters. A full diagram is displayed on* GitHub.