POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Physics Informed Neural Networks for Fluid Dynamics

Project Report
Numerical Analysis for Partial Differential Equations

Giulia Mescolini, Luca Sosta

Professor:
Prof. Alfio Quarteroni

Tutors:
Dr. Stefano Pagani
Dr. Francesco Regazzoni

Academic Year:
2020-2021

Abstract: In this project we apply Physics Informed Neural Networks for the numerical solution of fluid dynamics problems when few noisy data are available. The integration of the physical laws within the Neural Network has ensured better results for the reconstruction of pressure with respect to the fitting task, specially in the case of noisy boundary data. Moreover, an accurate output is obtained even with just few data (low-data regime), and this witnesses the ability of PINNs in reconstructing velocity and pressure fields in a condition of shortage of data, as it can happen in the medical framework.

Key-words: PINNs, Navier-Stokes, Poiseuille Flow, Colliding Flows, Lid-Driven Cavity, Coronary Flow

1. Introduction

In this work, we propose an approach based on Physics Informed Neural Networks (PINNs) to fluid dynamics problems, with the goal of reconstructing solutions in a regime of shortage of measurements.
In section 2 we introduce fundamental Machine Learning concepts about Neural Networks; we will deepen into their structure, their approximation power and the algorithm through which the model's parameters are learned by the net.
In section 3 we present PINNs and the minimization problem that is solved when the information coming from partial differential equations is exploited.
In section 4 we apply PINNs to two fluid dynamic test cases in which the analytical solution is known, and we introduce the main techniques adopted in the project, in terms of noise management, data normalization and losses implementation.
In section 5 we consider the *Lid-Driven Cavity problem* both in the stationary and in the non-stationary scenario.
In section 6 we apply the method developed in a medical setting: the flow of blood in an artery affected by a *stenosis*.
The software and libraries exploited in the project are presented in Appendix B.
The core part of the code implemented, such as the loss functions and utilities for data management, is reported in Appendix C and Appendix D.

2. An overview on Neural Networks

With the term *Artificial Neural Network* (ANN), we refer to a computing system inspired by the biological neural networks constituting animal brains. These tools were first introduced in the second half of the XX century, and nowadays they are witnessing an amazing popularity.
They fall into the framework of Deep Learning, which, in turn, is the most advanced specialization of Machine Learning, and they enable to process the information in a surprisingly complete way. The main strength of ANNs is that, while linear predictions (e.g. linear regression) work well only when the features (i.e. the inputs) are good, they can directly learn the features from the data.

2.1. The neuron

In order to understand how these tools work, we first introduce their basis components: **neurons**.
They consists in processing units which:

1. Take as input a finite number of signals
2. Compute a weighted sum with weights $w_{i,j}$
3. Subtract a threshold by adding a bias term b_j
4. Produce one single output applying a non-linear activation function ϕ (hyperbolic tangent in case of Figure 1). More in general, activation functions should behave similarly to the Heaviside function; other popular choices are the sigmoid function σ and the ReLU $(\cdot)_+$:

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad (x)_+ = max\{0, x\}$$

   It is fundamental that the activation function is *non-linear*; else, the whole Neural Network would be only a highly factorized linear function of the input data.

In *Fully Connected Neural Networks*, as the one involved in our project, each neuron is connected to all the neurons of the next and the previous layer.

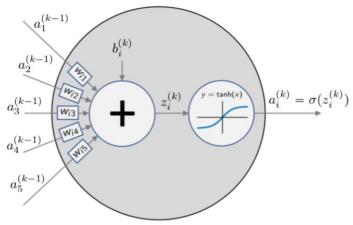


Figure 1: The neuron: the basic component of a Neural Network

2.2. General Structure

Now, let us define the structure of a Neural Network (NN), which enables the understanding of the already described process of features learning from data. As Figure 2 shows, the three main section are:

- **Input Layer**, receiving a d-dimensional input.
- **Hidden Layers**: L layers, each one with K_l nodes. Their task is to find suitable features.
- **Output Layer**, performing the desired task (which may be regression or classification) using the L^{th} layer's output as input.

Using this notation, the output of the neuron j in layer l is:

$$x_j^{(l)} = \phi\left(\sum_{i=1}^{K_{l-1}} w_{i,j} x_i^{(l-1)} + b_j^{(l)}\right)$$

2.3. Approximation Power

A possible rising question could be: *what functions can be approximated by Neural Networks?*
The comforting answer is given by Barron's Approximation Result.

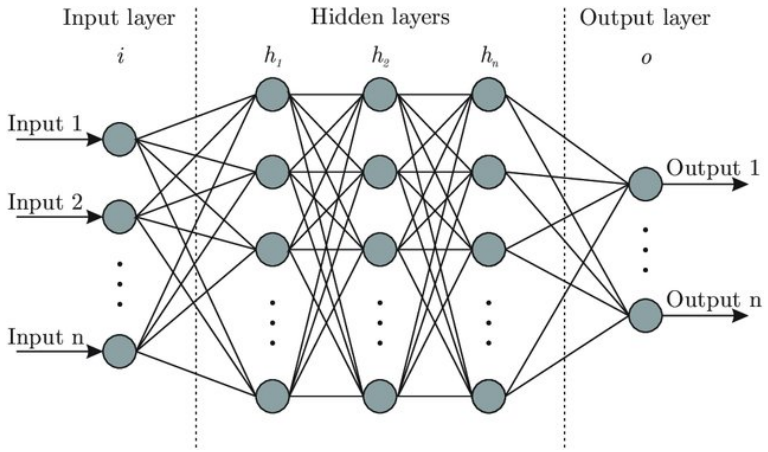


Figure 2: General structure of a Neural Network

Theorem 2.1. *Given a function $f : \mathbb{R}^D \to \mathbb{R}$, indicate with $\hat{f}$ its Fourier Transform. If*

$$\exists C > 0 \text{ such that } \int_{\mathbb{R}^D} |\omega||\hat{f}(\omega)|d\omega \leq C$$

Then

$$\forall n \geq 1 \quad \exists f_n \text{ of the form } f_n(x) = \sum_{j=1}^{n} c_j \phi(x^T w_j + b_j) + c_0 \text{ so that } \int_{|x|\leq r} |f(x) - f_n(x)|^2 dx \leq \frac{(2Cr)^2}{n}$$

.

This is equivalent to state that every ANN with a single hidden layer and a sigmoid-like activation function ϕ can approximate with arbitrarily small error (in the sense of the L^2 norm any continuous function on a compact set, provided that a sufficient number of hidden neurons are employed.
There exist a similiar result also for the pointwise approximation error, obtained by Shekhtman; in this case, the activation function is the ReLU.

2.4. Backpropagation

First of all, we define the Loss function $L(f)$, which provides a quantitative measure of the performance of the NN by expressing the distance between the output estimated by the model $f(x_n)$ and its true value y_n. A classical choice for approximation problems is the Mean Squared Error:

$$\frac{1}{N} \sum_{n=1}^{N} (y_n - f(x_n))^2 \tag{1}$$

The optimal solution is obtained by minimizing the Loss function, with respect to all weights and biases:

$$\min_{w_{i,j}^{(l)}, b_i^{(l)}} L(f)$$

To perform this task, we have first to choose the *optimizer*. i.e. the optimization algorithm. There exist several possibilities, with different memory consumption and time needed to reach convergence.

Optimization can be done, for example, with Stochastic Gradient Descent:

Algorithm 1 Stochastic Gradient Descent

1: Sample uniformly n
2: Compute the gradient of $L_n = \frac{1}{2}(y_n - f(x_n))^2$
3: Update the parameters: $(w_{i,j}^{(l)})_{t+1} = (w_{i,j}^{(l)})_t - \gamma \frac{\partial L_n}{\partial w_{i,j}^{(l)}}$ and $(b_i^{(l)})_{t+1} = (b_i^{(l)})_t - \gamma \frac{\partial L_n}{\partial b_i^{(l)}}$

Among the first order optimizers, a popular choice is *Adam*, an empowered version of Stochastic Gradient Descent.

Its peculiarity is that it is an *adaptive learning rate method*. This means that *Adam* is able to learn an optimal learning rate, which represents the width of displacements for each step of gradient descent.
We report in Algorithm 2 its general scheme, using the following notation:

- $Q(w)$ is the function to minimize
- m, v are the first and second moment of the gradient
- $\hat{m}$, $\hat{v}$ are unbiased estimators of m,v
- β_1, β_2 and ϵ are fixed parameters

Algorithm 2 Adam Optimizer

1: Initialization of w_0
2: **while** iteration $t <$ number of iterations **do**
$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla Q(w_t)$
$v_{t+1} = \beta_2 v_t + (1-\beta_2)(\nabla Q(w_t))^2$
$\hat{m} = \frac{m_{t+1}}{1-\beta_1^{t+1}}$
$\hat{v} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$
$w_{t+1} = w_t - \eta \frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}$
$t = t+1$
3: **end while**

Although in the framework of big data first order optimizers are the most popular, we present a second order one too, that is included in our NN: the Broyden–Fletcher–Goldfarb–Shanno (BFGS) Algorithm.
The logic behind this algorithm is the same of the Newton method: we want to minimize a convex function $minf(\mathbf{x})$ using a sequence of $\{\mathbf{x}_k\}$ second-order approximation of f, so we start from:

$$f(\mathbf{x}_k + t) \approx f(\mathbf{x}_k) + t\nabla f(\mathbf{x}_k) + \frac{t^2}{2}\nabla^2 f(\mathbf{x}_k)$$

We want that $\mathbf{x}_{k+1} = \mathbf{x}_k + t^*$, where t^* is the argmin of the right-hand side; hence, denoting the Hessian matrix by $H(\mathbf{x_k}) = \nabla^2 f(\mathbf{x}_k)$, we have:

$$t^* = -H^{-1}(\mathbf{x_k})\nabla f(\mathbf{x}_k)$$

The BFGS method provides a faster way to compute this value, using a sequence $\{\mathbf{H}_k\}$, an iterative approximation of the real Hessian matrix.

Algorithm 3 BFGS Optimizer

1: Initialization of $\mathbf{x}_0$, $\mathbf{B}_0$
2: **while** iteration $t <$ number of iterations **do**
$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{H}_k^{-1}\nabla f(\mathbf{x}^{(k)})$
$\mathbf{s}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$
$\mathbf{y}^{(k)} = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)})$
$\mathbf{H}_{k+1} = \mathbf{H}_k - \frac{\mathbf{H}_k \mathbf{s}^{(k)} (\mathbf{s}^{(k)})^T \mathbf{H}_k}{\mathbf{s}^{(k)} \cdot \mathbf{H}_k \mathbf{s}^{(k)}} + \frac{\mathbf{y}^{(k)} (\mathbf{y}^{(k)})^T}{\mathbf{y}^{(k)} \cdot \mathbf{s}^{(k)}}$
$t = t+1$
3: **end while**

We note that, whatever algorithm we choose, the computation of $\frac{\partial L}{\partial w_{i,j}^l}$ and $\frac{\partial L}{\partial b_i^l}$ is needed, $\forall i, j, l$.
This means that the number of derivatives that must be computed is of the order of $\mathcal{O}(K^2 L)$, supposing that we have K nodes in each of the L layers for the sake of simplicity; therefore, finding an efficient manner to compute the derivatives jointly becomes fundamental.
To illustrate the backpropagation algorithm, let us introduce the following notations:

- **Weight matrices:** $\mathbf{W}^{(l)}$, where $\mathbf{W}_{i,j}^{(l)} = w_{i,j}^{(l)}$.
  Note that $\mathbf{W^{(1)}} \in \mathbb{R}^{D\times K}$, $\mathbf{W^{(l)}} \in \mathbb{R}^{K\times K}$ $\forall\ 2 \le l \le L$, $\mathbf{W^{(L+1)}} \in \mathbb{R}^K$.
- **Bias vectors:** $\mathbf{b}^{(l)}$, whose i-th component is $b_i^{(l)}$. Note that $\mathbf{b^{(l)}} \in \mathbb{R}^K$ $\forall\ 1 \le l \le L$, $\mathbf{b^{(L+1)}} \in \mathbb{R}$.

With this notation,

$$x^{(1)} = f^{(1)}(x^{(0)}) = \Phi((\mathbf{W}^{(1)})^T x^{(0)} + \mathbf{b}^{(1)})$$
$$x^{(2)} = f^{(2)}(x^{(1)}) = \Phi((\mathbf{W}^{(2)})^T x^{(1)} + \mathbf{b}^{(2)})$$

$$\vdots$$

$$x^{(L+1)} = f^{(L+1)}(x^{(L)}) = \Phi((\mathbf{W}^{(L+1)})^T x^{(L)} + \mathbf{b}^{(L+1)})$$

Therefore, writing all in a more compact form, the output of the Neural Network is:

$$y = f(x^{(0)}) \text{ with } f = f^{L+1} \circ f^L \circ \cdots \circ f^2 \circ f^1$$

The loss function has the following form; we remark that it is a function of all weight matrices and bias vectors.

$$L = \frac{1}{2N} \sum_{n=1}^{N} (y_n - f^{(L+1)} \circ f^L \circ \cdots \circ f^2 \circ f^1(x_n))^2$$

Applying Stochastic Gradient Descent, we sample uniformly $n \in \mathbb{N}$, and we just need to compute

$$L_n = \frac{1}{2} (y_n - f^{(L+1)} \circ f^L \circ \cdots \circ f^2 \circ f^1(x_n))^2$$

The quantities we are interested on are the partial derivatives $\frac{\partial L_n}{\partial w_{i,j}^{(l)}}$ and $\frac{\partial L_n}{\partial b_i^{(l)}}$ $\forall (i,j,l)$.
Using the chain rule for the derivative computation for compound fractions and defining

$$z^{(l)} := (\mathbf{W}^{(l)})^T x^{(l-1)} + b^{(l)} = (\mathbf{W}^{(l)})^T \Phi(z^{(l-1)}) + b^{(l)}$$

We get that:

$$\frac{\partial L_n}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{K} \frac{\partial L_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial L_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial L_n}{\partial z_j^{(l)}} x_i^{(l-1)} = \delta_j^{(l)} x_i^{(l-1)}$$

$$\frac{\partial L_n}{\partial b_j^{(l)}} = \sum_{k=1}^{K} \frac{\partial L_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \frac{\partial L_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)}$$

$\delta_j^{(l)}$ can be easily computed in the following way:

$$\delta_j^{(l)} = \frac{\partial L_n}{\partial z_j^{(l)}} = \sum_{k=1}^{K} \frac{\partial L_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_k^{(l)}} = \sum_{k=1}^{K} \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_k^{(l)}}$$

Since $\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \Phi'(z_j^{(l)}) w_{j,k}^{(l+1)}$ (because $\frac{\partial w_{i,k}^{(l+1)}}{\partial z_j^{(l)}} = 0$ if $i \neq j$), we finally get:

$$\delta_j^{(l)} = \sum_{k=1}^{K} \delta_k^{(l+1)} \Phi'(z_j^{(l)}) w_{j,k}^{(l+1)}$$

Therefore, we could split the computation of the derivatives into two distinct phases:

- **Forward Pass**: the network is crossed from the input layer to the output layer.

$$x^{(0)} = x_n \in \mathbb{R}^d$$

$$z^{(l)} = (\mathbf{W}^{(l)})^T x^{(l-1)} + b^{(l)} \quad l = 1, \ldots, L+1$$

$$x^{(l)} = \Phi(z^{(l)}) \quad l = 1, \ldots, L+1$$

- **Backward Pass**: now the network is crossed in the opposite direction; note that in this step the initialization depends upon the choice of the activation function, for which possible options are presented in section 2.5.

$$\delta^{(L+1)} = -2(y_n - x^{(L+1)}) \Phi'(z^{(L+1)})$$

$$\delta^{(l)} = (\mathbf{W}^{(l+1)} \delta^{(l+1)}) \odot \Phi'(z^{(l)}) \ [1]$$

[1] Hadamard Product, i.e. elementwise multiplication

2.5. Activation Functions

In this section we present some common activation functions.

- **Sigmoid**:
$$\sigma(x) = \frac{1}{1+e^{-x}}$$
This function is smooth everywhere, but it presents a weak point: $|\sigma'(x)| \ll 1$ for $|x| \gg 1$, therefore the gradient is *vanishing*, and the learning can be slow for deep NNs.
- **Hyperbolic Tangent**:
$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$
This is the activation function chosen in the project; it is a balanced version of the Sigmoid.
- **ReLU**:
$$(x)_+ = max\{0, x\}$$
In this case, the gradient is not vanishing for $x > 0$, since the derivative is 1, but the function is not differentiable at $x = 0$ and for $x < 0$ the derivative is 0.
- **Leaky ReLU**:
$$f(x) = max\{\alpha x, x\}$$
This slightly modified version of ReLU solves the issue of the 0-gradient for $x < 0$.

2.6. Training and Testing

It is common practice in Machine Learning to partition the dataset into *training* and *test* set.
For the evaluation of the model performance, indeed, it is not fair to use the same points used to learn it (called *training* points), because the model may *overfit*, learning also the noise in the training data and therefore provide inaccurate estimates on freshly extracted points.
The general procedure for analyzing the performance of the model includes therefore the computation of the loss over points extracted only for the purpose of model evaluation, called *test* points.
This means that we cannot use the whole dataset for training, but we should reserve a part of them (typically smaller than 20%) for testing. This could be penalizing in Machine Learning as it common to wish for working with the most data possible, but in our application, as we will state in section 3, we rely more on the physical information from partial differential equations rather than on big data, hence we did not encounter problems of shortage of data.

2.7. Structure of our Neural Network

The architecture of the Neural Network employed in the project is a *feed-forward fully connected* Neural Network, having 3 hidden layers with 32 nodes, sketched in Figure 3.
The activation function chosen for hidden layers is the *hyperbolic tangent*.

3. Physics Informed Neural Networks

In this section we present Physics Informed Neural Networks, which represent a deep learning framework for solving problems involving partial differential equations.
They have been introduced in the last 5 years, and in their development Karniadakis, Perdikaris and Raissi have been pioneers. Nowadays, these tools are being empowered even outside academia, as it can be seen from solutions such as NVIDIA Modulus (see here).
PINNs consist in Neural Networks that are trained to solve *supervised learning* tasks - i.e., tasks when the outputs y_n are given and can be used for training, while respecting laws of physics governed by partial differential equations.
In standard Machine Learning, Neural Networks are common in "big data" frameworks, when plenty of observations are available to train our net, but in some applications data may be extremely costly and difficult to evaluate, such as in the medical field. Therefore, the exploitation of the physical knowledge that we have on the phenomena, represented by differential models, plays a key role, since it enables us to work even in a "small data" regime.

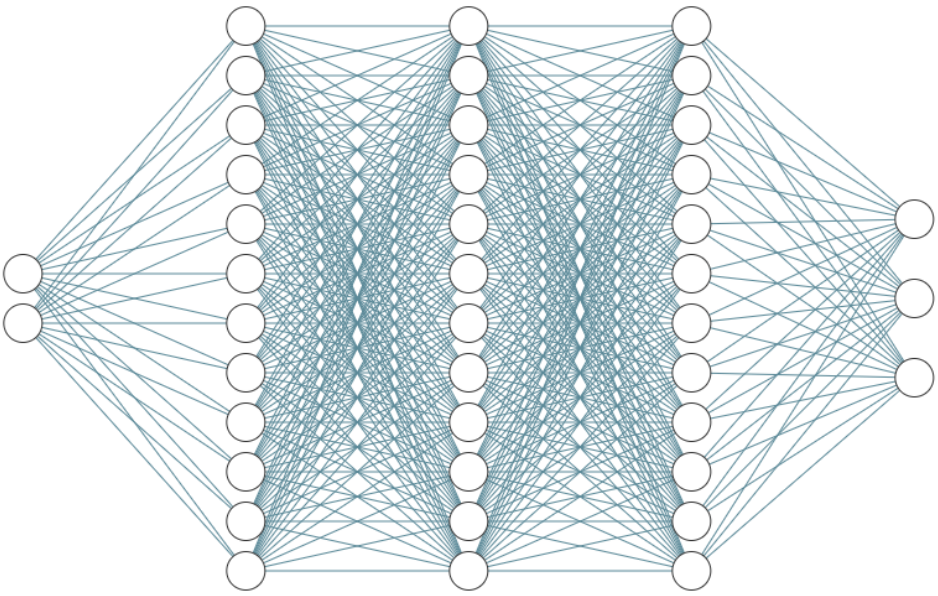


Figure 3: Architecture of the neural network (case with 2D input).

3.1. PINNs and Optimization Problems

In this work we are solving a direct problem: we follow the traditional scheme which starts from some data x, applies a model and obtains the desired output.
We deal with a PDE-constrained optimization problem such as:

$$\begin{cases} \min_u \mathcal{J}(u) \\ s.t.\ \dfrac{\partial u}{\partial t} + \mathcal{L}(u) = 0 \text{ in } \Omega \times [0,T] \quad (+ \text{ b.c. on } \partial\Omega) \\ \text{where } \mathcal{J}(u) = \lambda_{obs}||u(\mathbf{x},t) - y(\mathbf{x},t)||^2 + \lambda_{res} R(u) \end{cases}$$

where:

- $u(\mathbf{x},t)$ is the solution of the forward problem
- $(\mathbf{x},t)$ are the space and temporal variables defined in the input space $\Omega \times [0,T]$
- $\mathcal{L}$ is the Stokes or Navier-Stokes (nonlinear) differential operator
- $\lambda_{obs}, \lambda_{res} \in \mathbb{R}_+$ are hyper-parameters, balancing the effects of both terms of the functional
- $R(u)$ is a suitable regularization term.

PINN training can be formulated as an optimization problem, with unknowns represented by the weights and the biases of the network, summarized in the variable $\mathbf{W}$:

$$\min_{\mathbf{W}} \mathcal{J}(\mathbf{W})$$

Through PINNs we reformulate the optimization problem using a Neural Network as model for u.
Now the functional $\mathcal{J}$ depends both on the solution u and on the set of parameters $\mathbf{W} = [\mathbf{w}, \mathbf{b}]$, including weights and biases, and the constraint imposes that u is the output of our model.

$$\begin{cases} \min_{u,\mathbf{W}} \mathcal{J}(u, \mathbf{W}) \\ s.t.\ u(\mathbf{x},t) = \mathcal{NN}(\mathbf{x}, t; \mathbf{W}) \\ \text{where } \mathcal{J}(u,\mathbf{W}) = \dfrac{\lambda_{obs}}{N_{obs}} \displaystyle\sum_{i=1}^{N_{obs}} (u(\mathbf{x}_i, t_i) - y_i)^2 + \lambda_{res} R(u, \mathbf{W}) \end{cases} \tag{2}$$

Note that the interpolation error is replaced by its empirical mean, computed on a set of observation also called *dataset*, and the regularization term will take the burden to fulfill the previous physical constraint.
More details on the explicit formulation of the regularization term and on the dataset will be provided in the following sections. A further generalization can be done applying the above described method to an inverse problem in a trivial way: in fact, it is only needed to include the set of physical parameters of interest $\boldsymbol{\lambda}$ among the set of parameters $\mathbf{W}$, leading to the same formulation as Equation 2 with $\mathbf{W} = [\mathbf{w}, \mathbf{b}, \boldsymbol{\lambda}]$.

3.2. Structure of a PINN

We now illustrate the general structure of Physics-Informed Neural Networks.

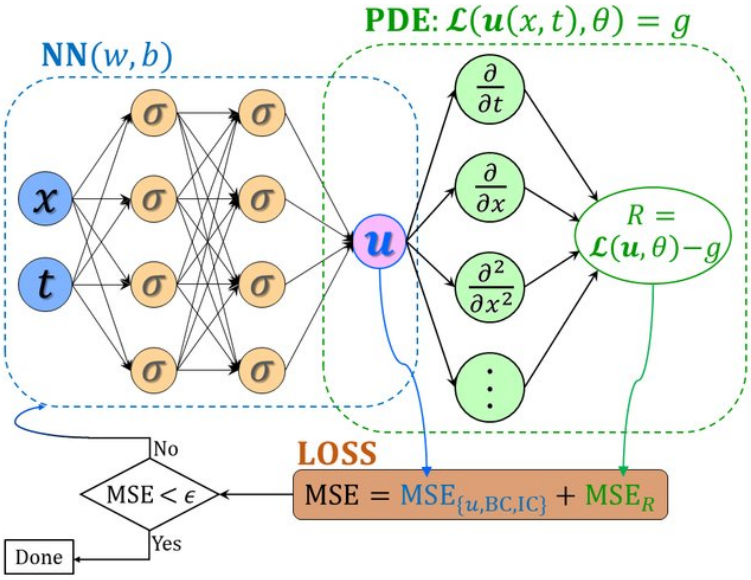


Figure 4: Structure of a PINN.

Given a generic partial differential equation (linear/nonlinear, steady/unsteady ...), we introduce the following notation:

- x and t are, respectively, space and time variable; $x \in \mathbb{R}^n$, $t \in \mathbb{R}$.
- $\mathcal{L}$ is the differential operator.
- $\mathbf{u}(\mathbf{x}, \mathbf{t})$ is the solution, which can depend in general on space and time.
- θ represents the physical parameters of the problem.
- Ω is a bounded domain in $\mathbb{R}^n$

The analytical formulation of the problem to solve is therefore:

$$\begin{cases} \mathcal{L}(\mathbf{u}(x,t)) = g \text{ in } \Omega \\ + \text{ B.C. on } \partial\Omega \end{cases} \tag{3}$$

In the framework of the forward problem our aim is to find the solution $\mathbf{u}$ to Equation 3 knowing θ, so knowing the operator $\mathcal{L}$.

In Figure 4 it is shown how PINNs work in this case: a prediction $\mathbf{u}$ is computed by a forward pass through the Neural Network, then, after computing partial derivatives with automatic differentiation (in this project, the reference library is `nisaba`, presented in Appendix B.1) we evaluate the losses associated to the residuals of the PDEs.

We can distinguish three important datasets within the *training* dataset:

- **Collocation Points**, being the points inside the domain Ω where we impose the PDE constraints, by requiring to minimize the residual of the equation.
  Referring to Equation 3, our residual is:

  $$\mathcal{R}(x,t) = \mathcal{L}(\mathbf{u}(x,t)) - g$$

  and using MSE loss function (Equation 1) we aim at minimizing the loss:

  $$\mathcal{J}_{col} = \frac{1}{N_{col}} \sum_{n=1}^{N_{col}} \mathcal{R}(x_n, t_n)^2$$

  with $\{(x_n, t_n)\}_{n=1}^{N_{col}}$ denoting the elements of the set of Collocation points.
- **Boundary Points**, on which we impose the exact value when we have a Dirichlet Boundary condition, or the minimization of the residual of the equation describing the Neumann condition.
  For example, with a Dirichlet Boundary Condition of the form

  $$u = g \text{ on } \Gamma_D$$

we add the loss

$$\mathcal{J}_{bd} = \frac{1}{N_{BCD}} \sum_{n=1}^{N_{BCD}} (u(x_n, t_n) - g(x_n, t_n))^2$$

- **Fitting Points**, i.e. points inside the domain Ω where we impose a value for the solution (obtained by the known analytical expression or, for example, from numerical simulations).
  In some test cases we employed few *hints* to help the Neural Network reconstructing a solution, but clearly the goal is to reduce them as much as possible.
  However, we remark that they are necessary in some situations, such as when pressure is determined up to an additive constant: in this cases, we used $n = 1$ fitting point for pressure to fix a solution referring to a specific constant.
  The loss to add is in this case:

$$\mathcal{J}_{fit} = \frac{1}{N_{fit}} \sum_{n=1}^{N_{fit}} (u(x_n, t_n) - u_{ex}(x_n, t_n))^2$$

3.2.1. Combination of the Losses

The final loss function that we want the net to minimize includes the effect of all the losses introduced above, with a weighted combination of them. For example, with M losses $\mathcal{J}_i$, we have

$$\mathcal{J} = \sum_{i=1}^{M} q_i \mathcal{J}_i$$

with q_i indicating the weight of the i-th loss.
Note that $\mathbf{q}$ is an hyperparameter of the net, and its choice will be crucial for the performance (as we will see in section 5).

4. First Test Cases

After a first attempt on a toy problem, the *Poisson Problem* reported in Appendix A, we started analyzing physical-based test cases.
We first introduce the *Navier Stokes Equations*, which govern fluid-dynamics problems: given a domain $\Omega \subset \mathbb{R}^n$ (with $n = 2, 3$ in physical applications), find a velocity field $\mathbf{u} \in \mathbb{R}^n$ and a pressure field $p \in \mathbb{R}$ such that:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 & \text{in } \Omega \\ \rho \left(\dfrac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} \right) + \nabla p - \mu \Delta \mathbf{u} = \mathbf{f} & \text{in } \Omega \\ + \text{ B.C. on } \partial\Omega \end{cases} \tag{4}$$

This formulation is the most general possible and can describe complex phenomena; however, in many applications, some terms can be neglected and this leads to a simplification of the problem.
For example, when the problem is *steady* and the *Reynolds Number* Re[2] is small, we can neglect the convection term as diffusion prevails over inertia and obtain the *Stokes Equations*:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 & \text{in } \Omega \\ \nabla p - \mu \Delta \mathbf{u} = \mathbf{f} & \text{in } \Omega \\ + \text{ B.C. on } \partial\Omega \end{cases} \tag{5}$$

Note that it is common to work with the *dimentionless* formulation of the Navier-Stokes Equations, which enables an easy generalization of the results to problems characterized by different orders of magnitude, but with the same ratios between scales.
Indeed, taking a reference velocity U and length L, we may obtain dimensionless variables and operators as:

$$\mathbf{u}^* = \frac{\mathbf{u}}{U}, \quad x^* = \frac{x}{L}, \quad p^* = \frac{p}{\rho U^2}, \quad t^* = \frac{t}{L/U}$$

$$\frac{\partial}{\partial t^*} = \frac{L}{U} \frac{\partial}{\partial t}, \quad \nabla^* = L\nabla, \quad \Delta^* = L^2 \Delta$$

[2] Defined as $Re = \frac{UL}{\nu}$, being ν the kynematic viscosity of the fluid and U and L a characteristic velocity and length of the problem

$$\begin{cases} \nabla^* \cdot \mathbf{u}^* = 0 & \text{in } \Omega \\ \dfrac{\partial \mathbf{u}^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*)\mathbf{u}^* + \nabla^* p^* - \frac{1}{Re}\Delta^* \mathbf{u}^* = \mathbf{f} & \text{in } \Omega \\ + \text{ B.C. on } \partial\Omega \end{cases} \tag{6}$$

We have first applied a Physics Informed Neural Network to test cases in which an analytical solution is known; we report in this section two meaningful examples of fluid-dynamics problems: *Poiseuille Flow* and *Colliding Flows*.

4.1. Poiseuille Problem

4.1.1. Goal

The first fluid-dynamic test case analyzed was a steady-state, laminar flow in a channel.
It enabled us to approach the study of differential systems with PINNs and to implement losses associated to partial differential equations and boundary conditions of type Dirichlet and Neumann, which appeared in the following more complex test cases.

4.1.2. Problem Setting

We chose to model the motion of lava, which is involved in slow and regular fluxes; therefore, the physical parameters ρ and μ are:

$$\rho = 3100 \frac{kg}{m^3} \qquad \mu = 890 \frac{kg}{m \cdot s}$$

leading to a $Re \sim 3$.
The domain under observation, shown in Figure 5, is a channel, with $L = 1m$, $H = 0.1m$, $\delta = H/2 = 0.05m$. What matters is clearly the scale between length and height, so these results can be easily replicated for larger domains and solve physical problems; moreover, with RANS simulations on the software PHOENICS, we assessed that the length of the channel is such that the flow reaches the fully developed conditions, that means that the solution is independent from the coordinate x.
We represented a test case in which the pressure at the beginning of the channel is $p_{str} = 10^6 Pa$, while the one at its ending is $p_{end} = 0Pa$.

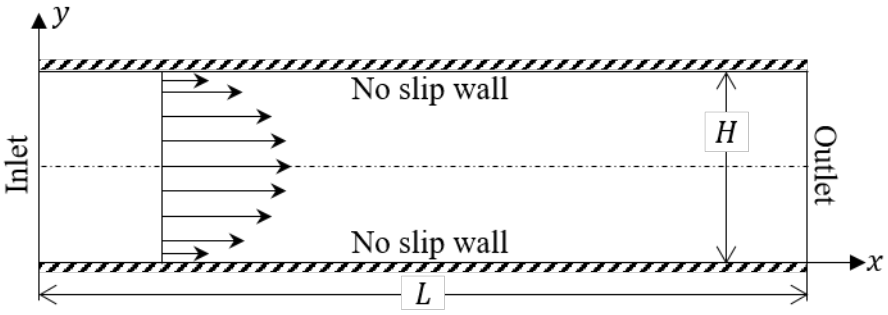


Figure 5: Geometry of the Poiseuille flow problem.

The equations governing the problem, named after the physicist Poiseuille, are the following:

$$\begin{cases} \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 & \text{in } \Omega \\ \rho\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) + \frac{\partial p}{\partial x} - \mu\Delta u = 0 & \text{in } \Omega \\ \rho\left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) + \frac{\partial p}{\partial y} - \mu\Delta v = 0 & \text{in } \Omega \\ u = v = 0 & \text{on } \Gamma_{D1} \\ u = u_{ex}, v = v_{ex} & \text{on } \Gamma_{D2} \\ \rho\frac{\partial u}{\partial x} + p = p_{end} & \text{on } \Gamma_N \\ \frac{\partial v}{\partial x} = 0 & \text{on } \Gamma_N \end{cases} \tag{7}$$

Where $\Omega = (0,1) \times (0, 2\delta)$, $\Gamma_{D1} = (0,1) \times \{0, 2\delta\}$, $\Gamma_{D2} = \{0\} \times (0, 2\delta)$ and $\Gamma_N = \{1\} \times (0, 2\delta)$.

The exact solution (in the fully developed region, where velocities do not depend on the coordinate x) is:

$$p_{ex} = \frac{p_{end} - p_{str}}{L} x + p_{str} \qquad u_{ex} = -\frac{\partial p}{\partial x} \frac{\delta}{2} y \left(2 - \frac{y}{\delta}\right) \qquad v_{ex} = 0$$

4.1.3. Training and test points

For each test case, we had to chose the number of training and test points and, more in detail, the sample size of each category of training points, referring to the distinction made in section 3 among collocation, boundary and fitting points.
In a `.txt` file named `simulation_options.txt`, the user can specify the epochs to perform and the number of points to use for training (divided into the different categories) and for testing.
Then, a subset of points is extracted randomly (according to a uniform distribution) in the given domain with the desired numerosity for each group.

4.1.4. Data Normalization

We introduced in all test cases normalization to let the network work with data in a more suitable range: from the Neural Network theory, indeed, we know that normalization techniques such as *Batch Normalization* can improve the performance.
In particular, the strategy adopted was to divide velocity and pressure by their spread, i.e. the difference between the maximum and the minimum value of the exact solution.

$$q \to q_{norm} = \frac{q}{q_{max} - q_{min}}$$

with q denoting our quantity of interest.
For the velocities, we kept the same normalization constant for u and v, by choosing the maximum among the spreads.
Note that we did not normalize the input of the Neural Network, as in all the test cases analyzed the range of the domain (both in space and time) was already in an acceptable range for the network.
A side-effect of this strategy is that the loss can now be interpreted also as squared percentage error.

4.1.5. Physical Losses

We now introduce the losses associated to the residuals of Navier-Stokes equations (stationary case).
To impose *mass conservation*, we included a loss minimizing the residual of the first equation of Equation 7:

$$\mathcal{R}_{mass}(x, y) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

For what concerns the *momentum conservation* equations, we want to minimize

$$\mathcal{R}_{mom_x}(x, y) = \rho \left(u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} \right) + \frac{\partial p}{\partial x} - \mu \Delta u \qquad \mathcal{R}_{mom_y}(x, y) = \rho \left(u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} \right) + \frac{\partial p}{\partial y} - \mu \Delta v$$

Following the definition of collocation losses given in section 3, we associated to the above defined residuals a loss function, defined as:

$$\mathcal{J}_{col} = \frac{1}{N_{col}} \sum_{n=1}^{N_{col}} [\mathcal{R}_{mass}(x_n, y_n)^2 + \mathcal{R}_{mom_x}(x_n, y_n)^2 + \mathcal{R}_{mom_y}(x_n, y_n)^2]$$

.

4.1.6. Boundary Losses

We now introduce the losses associated to the boundary conditions:

- **Dirichlet Boundary Condition**: the residual to minimize reads as

$$\mathcal{R}_{dir_x}(x, y) = u(x, y) - u_{ex} \text{ on } \Gamma_D$$

$$\mathcal{R}_{dir_y}(x, y) = v(x, y) - v_{ex} \text{ on } \Gamma_D$$

  denoting by $\Gamma_D = \Gamma_{D1} \cup \Gamma_{D2}$, and by u_{ex}, v_{ex} the boundary value imposed in Equation 7.

- **Neumann Boundary Condition**: the residual to minimze reads as:

$$\mathcal{R}_{neu}(x,y) = \rho\frac{\partial u}{\partial x} + p - p_{end} \text{ on } \Gamma_N$$

We included the boundary conditions through the loss function

$$\mathcal{J}_{bd} = \frac{1}{N_{BCD}} \sum_{n=1}^{N_{BCD}} [\mathcal{R}_{dir_x}(x_n,y_n)^2 + \mathcal{R}_{dir_y}(x_n,y_n)^2 + \mathcal{R}_{neu}(x_n,y_n)^2]$$

.

4.1.7. Fitting Losses

We now introduce the fitting losses, since from now on we will rely on their exploitation.
With the fitting losses, our aim is to impose the exact value, known analytically or from a numerical solution, of the PINN solution in a given point set.
The fitting residuals are:

$$\mathcal{R}_{fit_u}(x,y) = u(x,y) - u_{ex}(x,y)$$
$$\mathcal{R}_{fit_v}(x,y) = v(x,y) - v_{ex}(x,y)$$
$$\mathcal{R}_{fit_p}(x,y) = p(x,y) - p_{ex}(x,y)$$

and we associated to them a loss

$$\mathcal{J}_{fit} = \frac{1}{N_{fit}} \sum_{n=1}^{N_{fit}} [\mathcal{R}_{fit_u}(x_n,y_n)^2 + \mathcal{R}_{fit_v}(x_n,y_n)^2 + \mathcal{R}_{fit_p}(x_n,y_n)^2]$$

.
Note that fitting losses are, in terms of code, extremely similar to Dirichlet Boundary Losses: for this reason we decided to define a unique function, `dir_loss` for this family of losses, whose implementation can be found in Appendix C.
Indeed, both types of losses quantify the deviation of the computed function from given values on a set of points and do not involve derivative computation.
This strategy enabled us to reduce significantly *code reuse*, as the same function can be applied for test losses.
Note that in this test case we did not use at all fitting points for pressure, as the Neumann Boundary Condition makes it uniquely determined, while we integrated the information coming from the PDE with 10 measures for the velocity field.

4.1.8. Problem Formulation

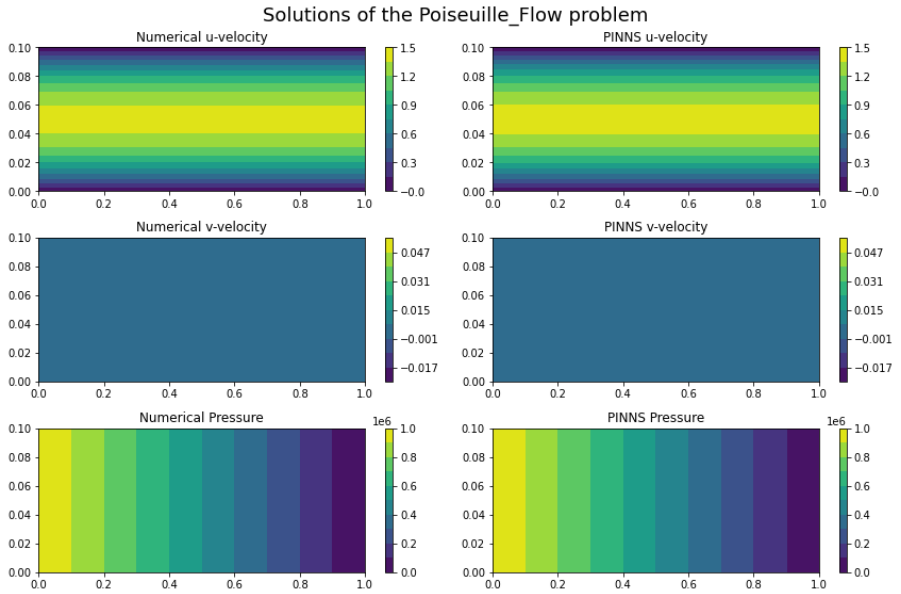
Having introduced the above mentioned residuals, the optimization problem associated to the PINN reads as:

$$\begin{cases} \min\limits_{u,\mathbf{W}} \mathcal{J}(u,\mathbf{W}) \\ s.t.\ u(\mathbf{x},t) = \mathcal{NN}(\mathbf{x},t;\mathbf{W}) \\ \text{where } \mathcal{J}(u,\mathbf{W}) = \dfrac{\lambda_{obs}}{N_{obs}} \sum\limits_{i=1}^{N_{obs}} (u(\mathbf{x}_i,t_i) - y_i)^2 + \lambda_{res} R(u,\mathbf{W}) \end{cases} \tag{8}$$

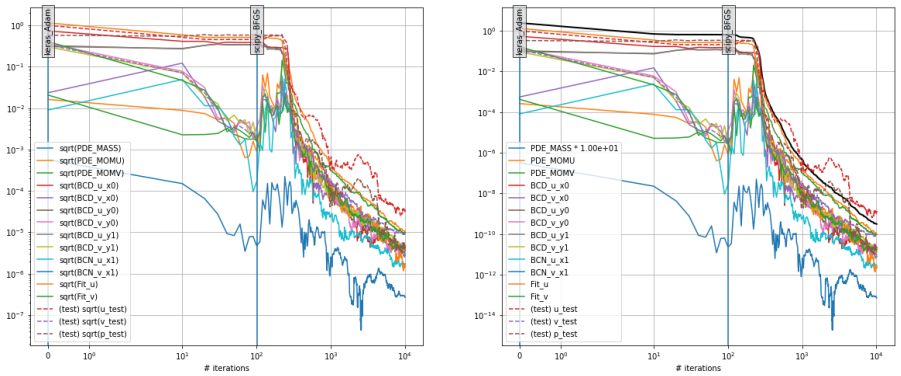
being $R(u,\mathbf{W}) = \mathcal{J}_{col} + \mathcal{J}_{bd} + \mathcal{J}_{fit}$.

4.1.9. Results

We managed to reconstruct an accurate solution, as it can be assessed by comparing the PINNs solution with the exact one (see Figure 6a) and by checking the test loss trends in Figure 6b, which stablilize themselves below 10^{-8} after 10.000 epochs.



(a) Exact vs PINN solution.



(b) Losses trend.

Figure 6: Poiseuille Flow results.

4.2. Colliding Flows

4.2.1. Goal

In this test case, whose analytical solution is known, we introduced noise in order to mimic a situation in which noisy measurements are available.
Moreover, we faced the issue of determining pressure uniquely since it is not fixed by the boundary conditions, and we implemented two strategies for pressure reconstruction.

4.2.2. Problem Setting

This test case represents two colliding flows in a square $\Omega = (-1, 1) \times (-1, 1)$.
In this case and in section 5, we will focus less on the physical parameters of the problem, in order to provide more general results.
The equations governing the problem are:

$$\begin{cases} \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 & \text{in } \Omega \\ \frac{\partial p}{\partial x} - \Delta u = 0 & \text{in } \Omega \\ \frac{\partial p}{\partial y} - \Delta v = 0 & \text{in } \Omega \\ u = 20xy^3 & \text{on } \partial\Omega \\ v = 5x^4 - 5y^4 & \text{on } \partial\Omega \end{cases} \tag{9}$$

Since the problem is *fully Dirichlet*, pressure is involved only in the partial differential equations, where it appears through its gradient; this means that in general it is defined up to a constant.
The exact solutions are:

$$p_{ex} = 60x^2y - 20y^3 + C \quad u_{ex} = 20xy^3 \quad v_{ex} = 5x^4 - 5y^4$$

If, for example, we want our Neural Network to reconstruct one specific solution for pressure, let us say the one with $C = 0$ (and therefore $\int_\Omega p dx = 0$), we need to insert some additional information inside the PINN.
For this purpose, we developed two strategies:

- **One fitting point for pressure**
  This strategy consists in fixing the pressure value in a point inside the domain, assigning it with the exact solution, possibly with some noise.
- **Pressure Mean Imposition Strategy**
  In this case we implemented a new loss function, which penalizes deviations from zero of the mean value of pressure.

4.2.3. Fitting Losses

The first strategy relies on the usage of fitting data for pressure; hence, we added a fitting loss $\mathcal{J}_{fit}$ for pressure according to the definition provided in 4.1.7.

4.2.4. Pressure Mean Imposition Loss

We then implemented the loss associated to the second strategy, which corresponds to minimizing the residual:

$$\mathcal{R}(x, y) = \int_\Omega p \, dxdy$$

The loss reads as:

$$\mathcal{J}_{pres} = \frac{1}{N_{pres}} \sum_{n=1}^{N_{pres}} p(x_n, y_n)$$

This technique results less invasive than the former, as we do not need to impose the exact value at any point. We evaluated the above loss on a sample of $N_{pres} = 100$ points randomly extracted; note that in this case we do not have to try to employ the least number possible of points, since we are imposing a weak condition which does not require the exact measure of pressure in any point.

4.2.5. Noisy Problem

In this test case, we started the study of the performance of our Network with respect to noisy measures; in particular way, we generated a *gaussian noise*, with $\sigma = 1$ and $\mu = 0$. It can be then scaled by a `factor`, that we generally set to be $1 - 5\%$ of the input. Note that this corresponds to a change of variance of the noise, by the property:

$$X \sim \mathcal{N}(0, 1) \Rightarrow aX \sim \mathcal{N}(0, a^2) \quad \forall a \in \mathbb{R}$$

We used this noise to distort both boundary conditions and fitting values, in order to simulate a real case in which measures are affected by noise.

4.2.6. Problem Formulation

For the first strategy for pressure management, the optimization problem associated to the PINN reads as:

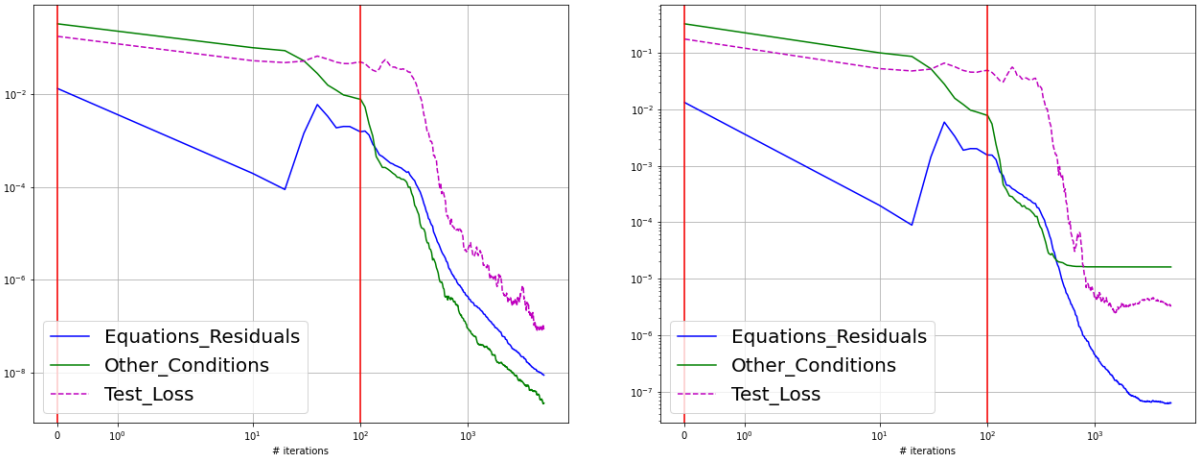
$$\begin{cases} \min_{u,\mathbf{W}} \mathcal{J}(u, \mathbf{W}) \\ s.t. \;\; u(\mathbf{x}, t) = \mathcal{NN}(\mathbf{x}, t; \mathbf{W}) \\ \text{where } \mathcal{J}(u, \mathbf{W}) = \dfrac{\lambda_{obs}}{N_{obs}} \displaystyle\sum_{i=1}^{N_{obs}} (u(\mathbf{x}_i, t_i) - y_i)^2 + \lambda_{res} R(u, \mathbf{W}) \end{cases} \tag{10}$$

being $R(u, \mathbf{W}) = \mathcal{J}_{col} + \mathcal{J}_{bd} + \mathcal{J}_{fit}$.
For the case of pressure mean imposition, we have:

$$\begin{cases} \min\limits_{u,\mathbf{W}} \mathcal{J}(u, \mathbf{W}) \\ s.t.\ u(\mathbf{x}, t) = \mathcal{NN}(\mathbf{x}, t; \mathbf{W}) \\ \text{where } \mathcal{J}(u, \mathbf{W}) = \dfrac{\lambda_{obs}}{N_{obs}} \displaystyle\sum_{i=1}^{N_{obs}} (u(\mathbf{x}_i, t_i) - y_i)^2 + \lambda_{res} R(u, \mathbf{W}) \end{cases} \tag{11}$$

being $R(u, \mathbf{W}) = \mathcal{J}_{col} + \mathcal{J}_{bd} + \mathcal{J}_{pres}$.

4.2.7. Results with fitting strategy



(a) Losses trend - non noisy case.

(b) Losses trend - noisy case.

Figure 7: Colliding Flows with Fitting strategy.

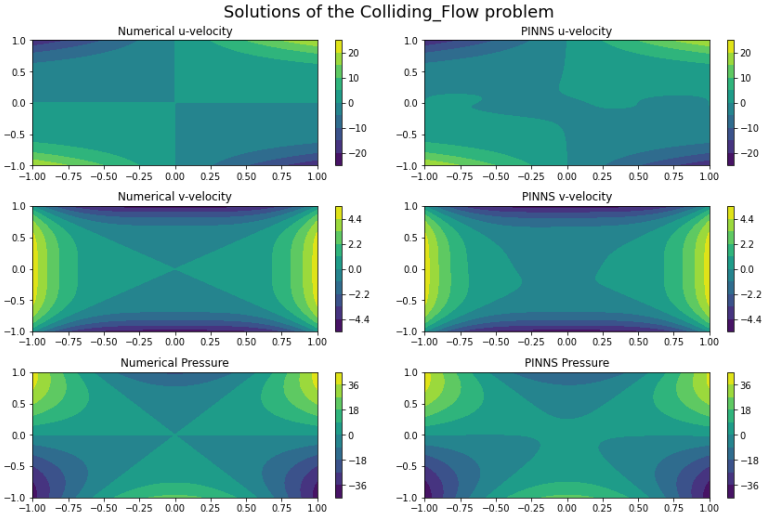
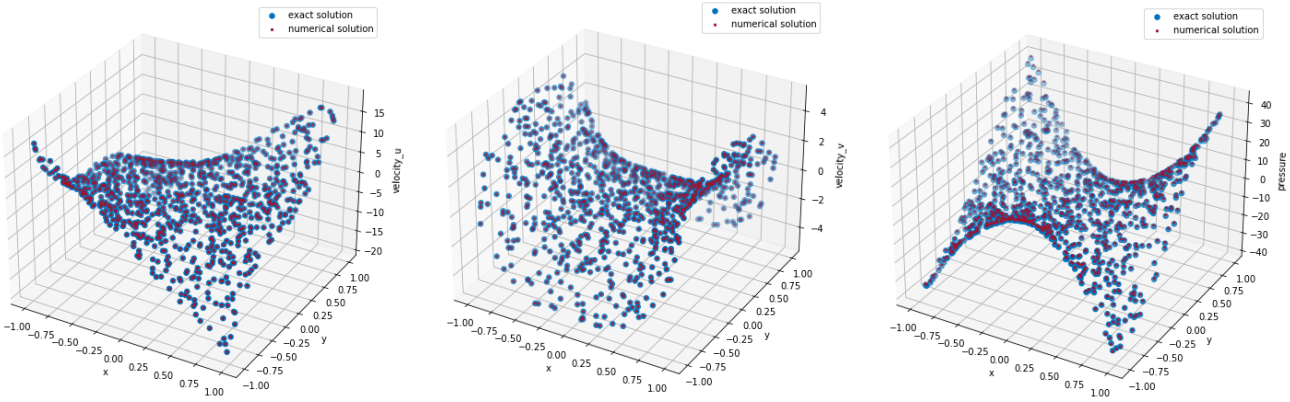


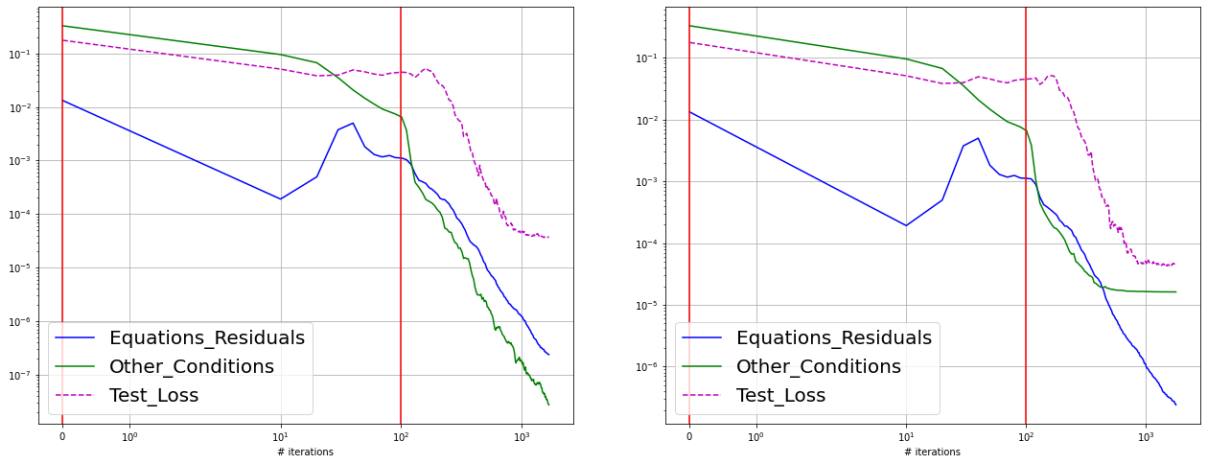
Figure 8: Exact vs PINN solution.

4.2.8. Results with Pressure Mean Imposition strategy



(a) u-velocity. (b) v-velocity. (c) Pressure.

Figure 9: Colliding Flows with Pressure Mean Imposition strategy - Results.



(a) Losses trend - non noisy case. (b) Losses trend - noisy case.

Figure 10: Colliding Flows with Pressure Mean Imposition strategy strategy - Losses.

Note that in the noisy case, for both strategies, the boundary and test losses remain stuck (at a value that still ensures a pretty accurate reconstruction of the solution, as it can be seen from Figure 8 and Figure 9).
In the case without noise, the boundary loss does not show the plateau, while the test loss still does; this means that the performance of the net is nearly the same in both cases, and it is slightly better using the first strategy, as it can be assessed by comparing the value reached by the test loss (which is clearly the same loss function in both cases).

5. Navier-Stokes - The Lid-Driven Cavity

5.1. Goal

In this section, we deal with a test case whose solution is not known analytically in two different settings: first in a *stationary* version, then we introduced time evolution.
We used this more complex test case to build an interface between the PINN and numerical solutions, in a setting in which we include noise.
Moreover, we adapted our method to non-stationary problems, by developing a strategy to consider the time variable.

5.2. Problem Setting

The object of this study is the *Lid-Driven Cavity* Flow, describing the motion of a fluid inside a square cavity, subjected to an upper moving wall with velocity U.
We denote by Γ_{D1} the bottom, right and left boundary, while with Γ_{D2} the upper one, and, as already done in subsection 4.2, we study a case with $Re = 1$.
The equations describing the problem in the general non stationary setting read as:

$$\begin{cases} \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 & \text{in } \Omega \\ \frac{\partial u}{\partial t} + \left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} \right) + \frac{\partial p}{\partial x} - \Delta u = 0 & \text{in } \Omega \\ \frac{\partial v}{\partial t} + \left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} \right) + \frac{\partial p}{\partial y} - \Delta v = 0 & \text{in } \Omega \\ u = v = 0 & \text{on } \Gamma_{D1} \\ u = U, v = 0 & \text{on } \Gamma_{D2} \\ + \text{ initial condition: } u, v, p = 0 \text{ in } \Omega \end{cases} \tag{12}$$

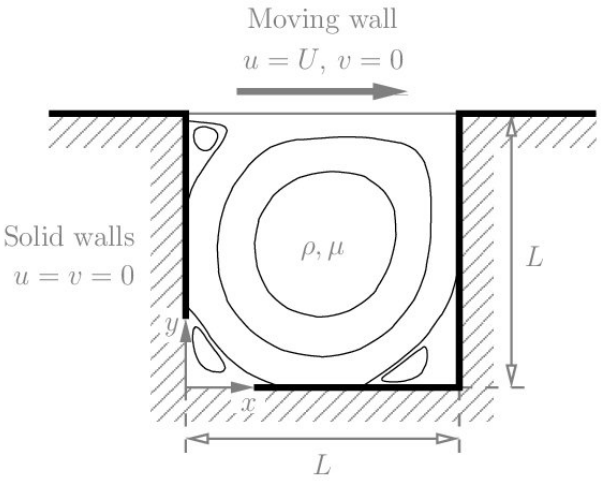


Figure 11: Geometry of the problem.

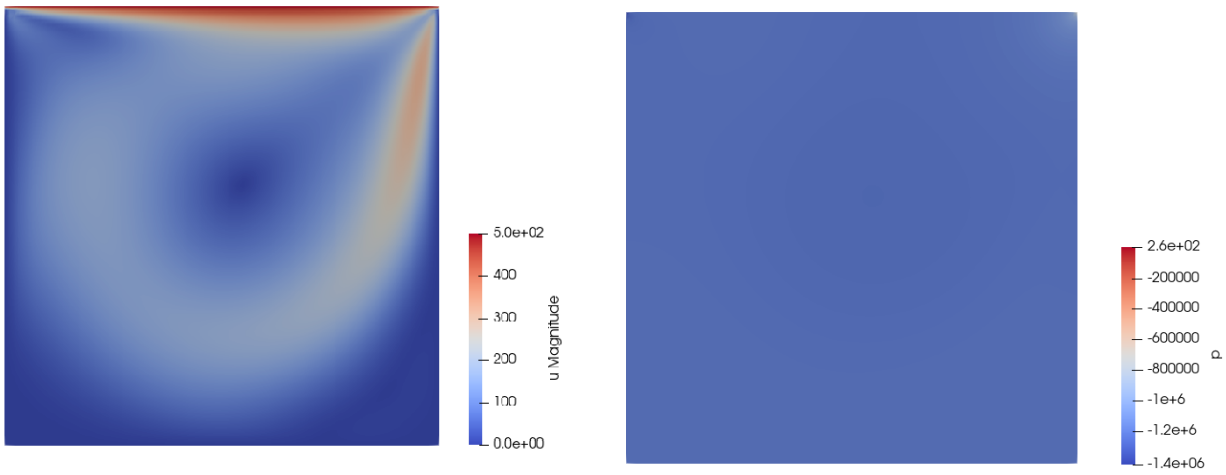
Note that, in this problem, pressure appears only through its derivatives in the equations, but we also have an initial condition on it: hence, for the non-stationary case it is not necessary to set its level with a fitting point. For testing and for fitting we needed a reference solution, that could not be the exact one as we do not have an analytical expression in this case.
We relied on the numerical solution generated with `FeNiCS`, a `Python` library for Finite Elements; the workflow of these auxiliary scripts employed in the project is described in Appendix B.2.

5.3. Steady Case

For the steady case, we set $U = 500$, and solved Navier-Stokes equations with a Gaussian noise of 1% both on fitting data and on boundary data.

5.3.1. Point sets construction

From the point of view of the implementation of physical losses, nothing changed with respect to the ones presented in subsection 4.1, because we are solving the same PDEs.
The geometry is regular, hence we could generate a regular domain grid, with the same strategy adopted in the other test cases. However, it should be remarked that this time we need to identify the position of each point in the `FeNiCS` output, because, for fitting and for testing, we need to find the corresponding numerical solution. An easy way to solve this problem is extracting the mesh directly from `FeNiCS` output: the solution is stored in a `.h5` file, containing both the values of numerical velocity and pressure and the geometrical information on the domain, hence we could read the triplets (x, y, sol), being sol the desired velocity component or the pressure, directly from the `.h5` file.
However, we noted that the points in the mesh were stored in the same order as in a regular grid built with `np.linspace`, so we constructed the domain manually and recovered the corresponding value of the solution.



(a) Velocity magnitude (steady case). (b) Pressure (steady case).

Figure 12: Lid-Driven Cavity - Steady: numerical solution.

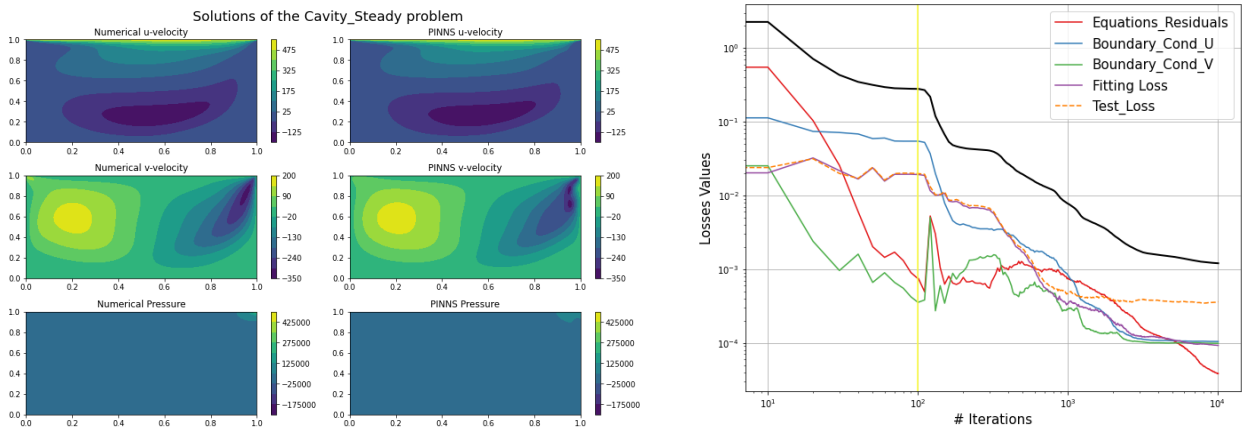
5.3.2. Problem Formulation

Using the notation introduced in the subsection 4.1, the problem formulation reads as:

$$\begin{cases} \min\limits_{u,\mathbf{W}} \mathcal{J}(u,\mathbf{W}) \\ s.t.\ \ u(\mathbf{x},t) = \mathcal{NN}(\mathbf{x},t;\mathbf{W}) \\ \text{where } \mathcal{J}(u,\mathbf{W}) = \dfrac{\lambda_{obs}}{N_{obs}} \displaystyle\sum_{i=1}^{N_{obs}} (u(\mathbf{x}_i,t_i) - y_i)^2 + \lambda_{res} R(u,\mathbf{W}) \end{cases} \tag{13}$$

being $R(u,\mathbf{W}) = \mathcal{J}_{col} + \mathcal{J}_{bd} + \mathcal{J}_{fit}$.

5.3.3. Results



(a) Comparison between numerical and PINN solution. (b) Trend of the losses.

Figure 13: Steady Lid-Driven Cavity.

The reconstruction of the numerical solution is accurate, with just 1 noisy measure for the pressure inside the domain.
However, we noticed that in this case the amount of noisy measures for the velocity required for the velocity was pretty high (100).
Specially, it was higher than the one of the test case in section 6, although the complexity of the geometry is smaller. We supposed that this is due to the choice of a high-speed bulk flow ($500\frac{m}{s}$) and to the less regular velocity profile (which can be assessed by comparing Figure 12a and Figure 17).

5.4. Unsteady Case

In this case, we chose a smaller velocity of the moving wall, namely $U = 1$.
We focused on a test case with a Gaussian noise with $\sigma = 5\%$, both on boundary data and on fitting data.

5.4.1. The third dimension: time

In this case, we got to find a strategy for the inclusion of the time variable in the problem.
The technique adopted consisted in considering a domain grid whose coordinates are time and space: we therefore assembled a matrix where each row corresponds to a triplet (t, x, y). In this way, we were able to generalize techniques for automatic differentiation, as time derivatives could be extracted as the first component of the gradient with respect to (t, x, y). There is just a stylistic downside: we could not, for example, exploit `divergence` or `laplacian` operators, as they included time derivatives as well, but we had to extract partial derivatives in each direction.

5.4.2. Problem Formulation

$$\begin{cases} \min\limits_{u,\mathbf{W}} \mathcal{J}(u, \mathbf{W}) \\ \quad s.t.\ u(\mathbf{x}, t) = \mathcal{NN}(\mathbf{x}, t; \mathbf{W}) \\ \quad \text{where } \mathcal{J}(u, \mathbf{W}) = \dfrac{\lambda_{obs}}{N_{obs}} \displaystyle\sum_{i=1}^{N_{obs}} (u(\mathbf{x}_i, t_i) - y_i)^2 + \lambda_{res} R(u, \mathbf{W}) \end{cases} \tag{14}$$

being $R(u, \mathbf{W}) = \mathcal{J}_{col} + \mathcal{J}_{bd} + \mathcal{J}_{fit}$.
Note that in this case $\mathcal{J}_{col}$ is slightly different, as the time derivative appears in the equations, too. Indeed:

$$\mathcal{R}_{mom_x}(x, y) = \frac{\partial u}{\partial t} + \rho \left(u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} \right) + \frac{\partial p}{\partial x} - \mu \Delta u \qquad \mathcal{R}_{mom_y}(x, y) = \frac{\partial v}{\partial t} + \rho \left(u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} \right) + \frac{\partial p}{\partial y} - \mu \Delta v$$

and

$$\mathcal{J}_{col} = \frac{1}{N_{col}} \sum_{n=1}^{N_{col}} [\mathcal{R}_{mass}(x_n, y_n)^2 + \mathcal{R}_{mom_x}(x_n, y_n)^2 + \mathcal{R}_{mom_y}(x_n, y_n)^2]$$

5.4.3. Management of pressure behavior

With the postprocessing of the numerical solution on `ParaView` we noted that pressure exhibited jumps from one time step to another, resulting in difficulties for the net even for the pure interpolation task.
In order to improve the performance, we subtracted at each time step the mean value.

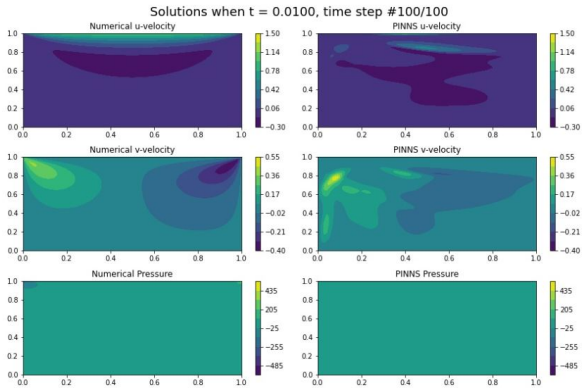
5.4.4. Results

We performed trials for increasing weight given to the physical losses, as for the first trial performed we noticed that too many fitting points were necessary in order to get a good performance.
At the beginning, by setting:

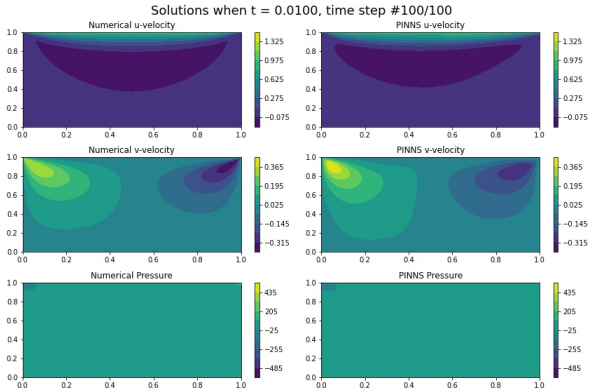
Loss	Weight
`PDE_MASS`	10^{-2}
`PDE_MOM_X`	10^{-3}
`PDE_MOM_Y`	10^{-3}

Table 1: Small weights for physical losses.

We did not obtain satisfactory results, as it can be seen in Figure 14a, even using 50 fitting points for velocity and 1 for pressure.



(a) Small `weight` for PDE losses.



(b) Higher `weight` for PDE losses.

Figure 14: Solution at the final time instant for unsteady Lid-Driven Cavity.

Then, by increasing `weights`, we managed to obtain the solution in Figure 14b, which is close to the numerical one with the options in Table 2:

Loss	Weight
PDE_MASS	10^1
PDE_MOM_X	10^0
PDE_MOM_Y	10^0

Table 2: Increased weights for physical losses.

Note that, we were also able to avoid the usage of fitting points, because the solution of the problem is not anymore determined by a constant, since the initial conditions fix the pressure initial value and the equations regulate its evolution.
In this case, we also analyzed the loss trend:

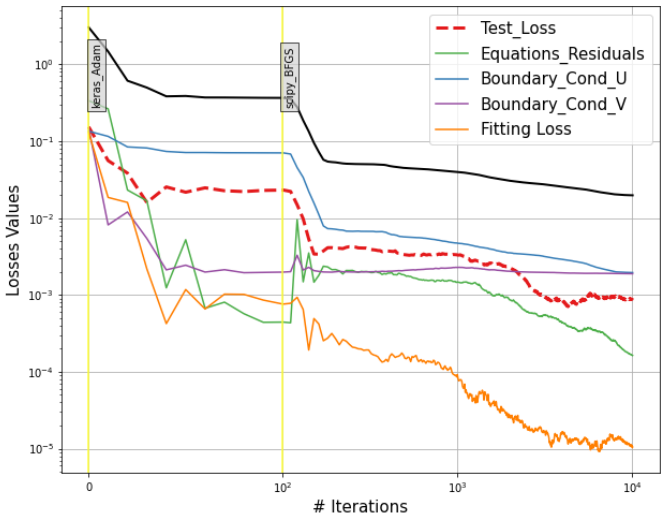


Figure 15: Losses Trend - Unsteady Lid-Driven Cavity.

After 10.000 epochs, the test loss stabilizes itself at $\sim 10^{-3}$, and the reconstruction of the numerical solution is pretty accurate.

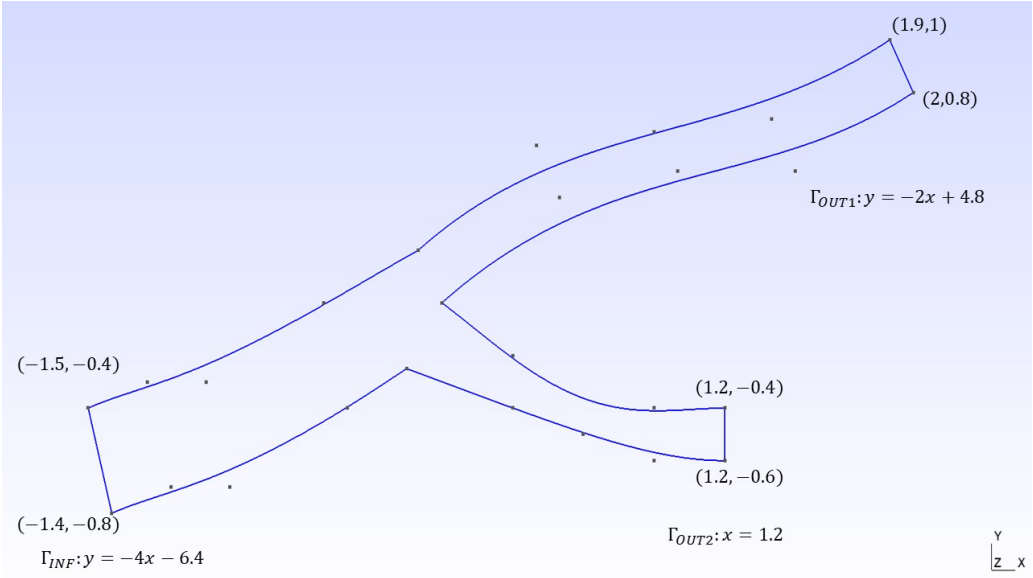


Figure 16: Coronary Flow - Geometry of the problem with specification of boundaries.

6. An Application - The Coronary Flow

6.1. Goal

The goal of this test case is to apply the method introduced to a real-based situation, characterized by a complex geometry, unavailability of the analytical solution and noisy measures.

6.2. Problem Setting

The case under study is the flow of blood in an artery with a vessel bifurcation and a significant deformation in one of the branches, which can represent a disease such as a *stenosis*.
This time, giving meaning to the physical solution is crucial, hence we dropped the simplification of $Re = 1$ adopted in the previous cases and approached the problem with physical-based data.
The geometry of the problem is in the range $[-1.5, 2]$ for the x direction and $[-0.8, 1]$ for the y direction. The units, in order to mimic a real artery of medium size, should be centimeters, as the diameter of the largest vessel (in our case, inflow) is $\sim 0.4cm$.
We introduce the quantities:

- $H = \sqrt{(-0.4 + 0.8)^2 + (-1.5 + 1.4)^2} cm = 0.41cm$, being the diameter of the principal vessel.
- $\theta = atan\left(\frac{1}{4}\right)$, being the angle between the horizontal axis and the inflow.

The equations describing the problem read as:

$$\begin{cases} \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 & \text{in } \Omega \\ \rho\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) + \frac{\partial p}{\partial x} - \mu\Delta u = 0 & \text{in } \Omega \\ \rho\left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) + \frac{\partial p}{\partial y} - \mu\Delta v = 0 & \text{in } \Omega \\ u = v = 0 & \text{on } \Gamma_{NOSL} \\ u = Ucos(\theta)\frac{\sqrt{((x-x_0)^2+(y-y_0)^2)}}{H}\left(1 - \frac{\sqrt{((x-x_0)^2+(y-y_0)^2)}}{H}\right) & \text{on } \Gamma_{INF} \\ v = Usin(\theta)\frac{\sqrt{((x-x_0)^2+(y-y_0)^2)}}{H}\left(1 - \frac{\sqrt{((x-x_0)^2+(y-y_0)^2)}}{H}\right) & \text{on } \Gamma_{INF} \\ \nu\frac{\partial \mathbf{u}}{\partial \mathbf{n}} + p\mathbf{n} = 0 & \text{on } \Gamma_{OUT1} \\ \nu\frac{\partial \mathbf{u}}{\partial \mathbf{n}} + p\mathbf{n} = 0 & \text{on } \Gamma_{OUT2} \end{cases} \tag{15}$$

where the labels $\Gamma_{INF}, \Gamma_{OUT1}, \Gamma_{OUT2}$ are the segments in Figure 16, $\Gamma_{NOSL} = \partial\Omega \setminus (\Gamma_{INF} \cup \Gamma_{OUT1} \cup \Gamma_{OUT2})$ and $(x_0, y_0) = (-1.4, -0.8)$.

6.2.1. Boundary Conditions

It is noteworthy to analyze the boundary conditions:

- Inflow: on Γ_{INF}, we imposed the parabolic profile which is the analytical solution of the Poiseuille flow (same as in subsection 4.1). Note that we imposed it on the coordinate system rotated by an angle θ with respect to the horizontal direction, so we needed to go back to the cartesian system with a projection.
- Non-slip boundary: on the lateral edges of the vessel we imposed null velocity.
- Outflows: on the two outflows $\Gamma_{OUT1}, \Gamma_{OUT2}$ we imposed natural Neumann conditions, as already done for the outflow in subsection 4.1.
  In the case of Γ_{OUT2}, the implementation is trivial as the normal versor is $\mathbf{n} = (1, 0)^T$, being the edge vertical. For Γ_{OUT1}, we computed the normal versor to this edge $\mathbf{n} = (2, 1)^T$ from the analytical expression of the boundary edge $y = -2x + 4.8$.

6.3. Numerical solution

First of all, we adapted the `FeNiCS` script used for the Lid-Driven Cavity to solve the Navier-Stokes equations on the coronary mesh. Apart from the above mentioned boundary conditions, the most relevant modification was the introduction of the physical quantities (through the kynemicatic viscosity ν) into the equations.

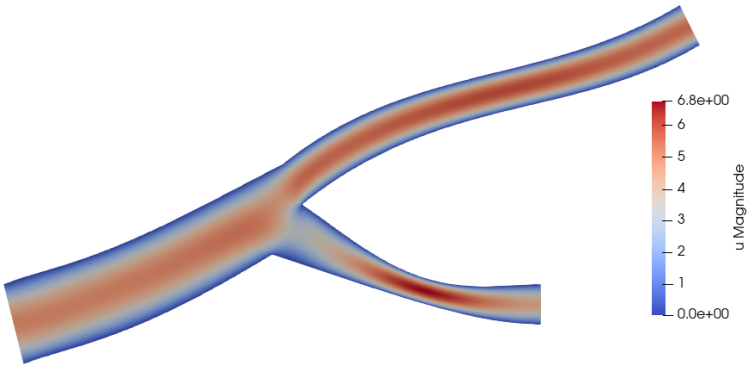


Figure 17: Numerical solution - Velocity magnitude.

From the visual inspection of the numerical solution, we can note that in the lower branch, affected by the *stenosis*, there is an increase in the velocity magnitude, coherently with the shrinkage of the vessel.

Figure 18: Numerical solution - Pressure.

6.4. PINN solution

6.4.1. Point sets construction

Being the domain irregular and not straightforwardly parametrizable, this time we had to extract the point dataset from the mesh.
For what concerns the boundaries, the segments $\Gamma_{INF}, \Gamma_{OUT1}, \Gamma_{OUT2}$ could be easily constructed, while there is no analytical expression for Γ_{NOSL}.

Therefore, we generated in the `FeNiCS` script the file `bpoints.npy`, marking with a flag $(1, 2, 3$ or $4)$ points on the boundary depending on the Γ_i of membership.

6.5. Probelm Formulation

6.5.1. Losses

From the implementation point of view, we relied on the already introduced codes for physical losses and boundary conditions.
Therefore, we report here only the loss associated to Neumann Boundary conditions, which changes significantly with respect to the ones presented in subsection 4.1 and Appendix A.

$$\begin{cases} \min\limits_{u,\mathbf{W}} \mathcal{J}(u, \mathbf{W}) \\ s.t.\ u(\mathbf{x}, t) = \mathcal{NN}(\mathbf{x}, t; \mathbf{W}) \\ \text{where } \mathcal{J}(u, \mathbf{W}) = \dfrac{\lambda_{obs}}{N_{obs}} \displaystyle\sum_{i=1}^{N_{obs}} (u(\mathbf{x}_i, t_i) - y_i)^2 + \lambda_{res} R(u, \mathbf{W}) \end{cases} \tag{16}$$

being $R(u, \mathbf{W}) = \mathcal{J}_{col} + \mathcal{J}_{bd} + \mathcal{J}_{fit}$.

6.6. Results

We present the results for the noisy case (1% both on boundary data and on velocity measures), after 30.000 epochs of training, since in this case we saw that the losses did not assess themselves on a constant value after 10.000 epochs, as can be checked in Figure 19.

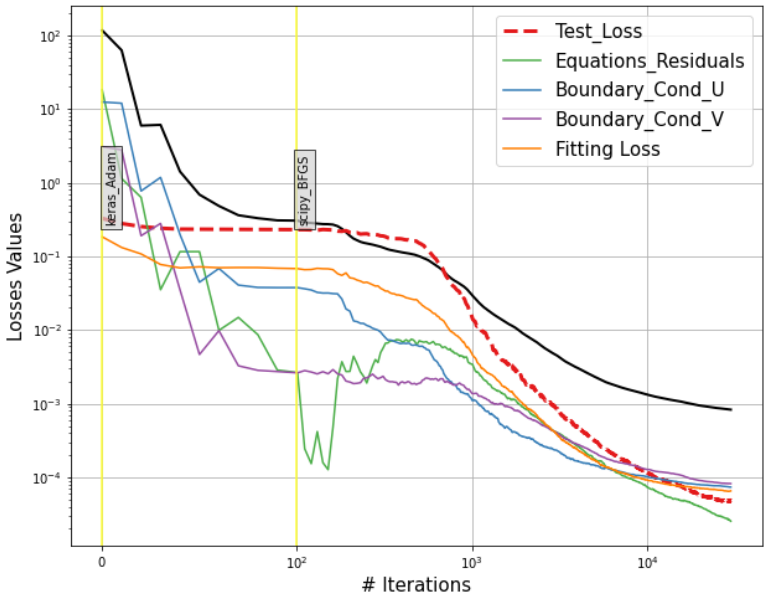


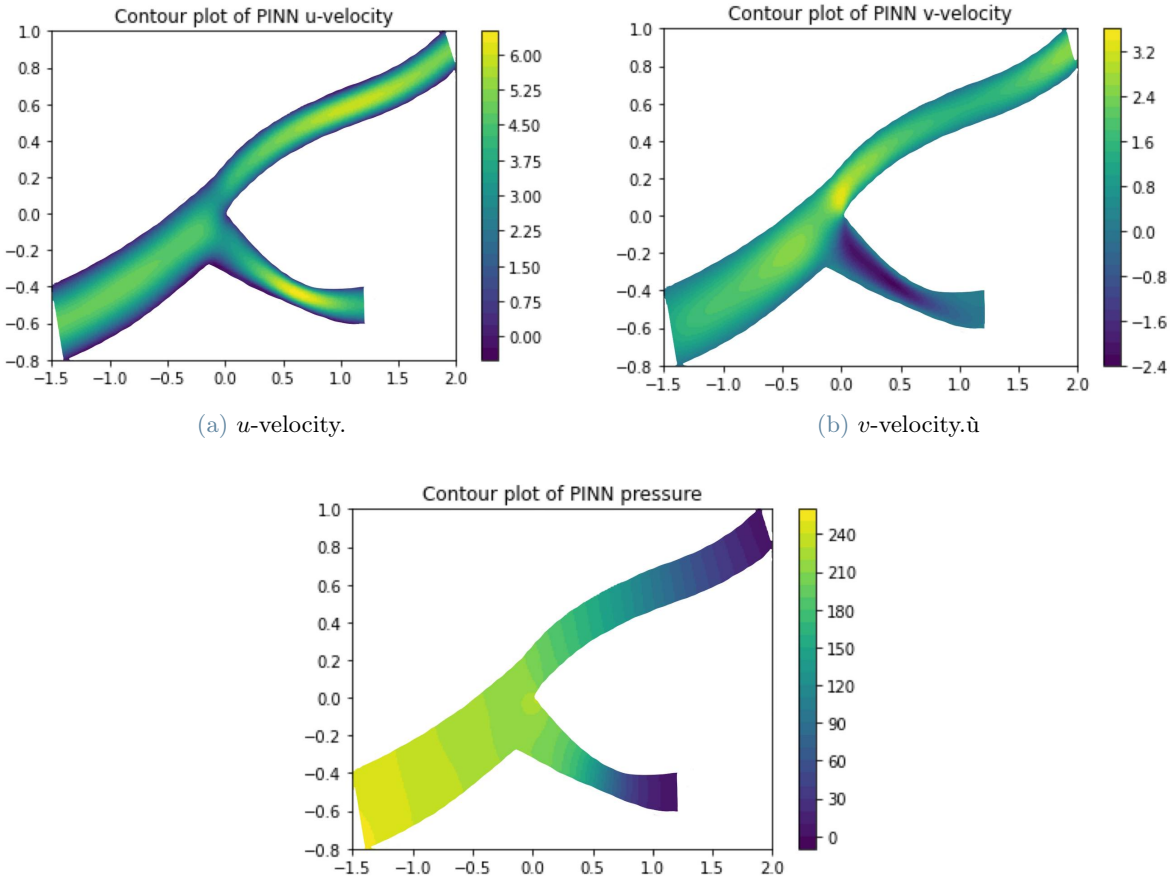
Figure 19: Trend of the losses - Coronary Steady Problem.

We managed to obtain an accurate reconstruction of the numerical solution with 10 fitting points for velocity and no fitting points for pressure. Indeed, the Neumann Conditions enabled us to fix uniquely pressure without asking for measures inside the domain.
The PINN solutions are shown in Figure 20.

7. Conclusions

In this project, we developed methods based on Physics Informed Neural Networks to analyze several test cases from fluid dynamics.
Starting from problems whose analytical solution is known, we assessed the PINN's reconstruction power in a

regime where noisy measures are available, then we dealt with cases in which the analytical solution is unknown, and compared the PINN's performance with the numerical solution.
More specifically, we tried to estimate pressure, which is in general the most difficult quantity to measure, without employing too many data on it; we managed to reconstruct it accurately using no information other than the boundary conditions and the Navier-Stokes equations in the test cases presented in subsection 4.1, subsection 5.4 and section 6, while we had to use one noisy measure of it or a condition on its mean value in subsection 4.2 and subsection 5.3.



(a) u-velocity.

(b) v-velocity.ù

(c) Pressure.

Figure 20: PINN solution for Coronary Steady case.

References

[1] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (PINNs) for fluid mechanics: A review, 2021.

[2] Donald E. Knuth. Machine Learning and Deep Neural Networks Applications in Coronary Flow Assessment. *Journal of Thoracic Imaging*, 35:S66–S71, 2020.

[3] Alfio Quarteroni. *Modellistica Numerica per Problemi Differenziali.* Springer, 2012.

[4] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part I): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[5] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part II): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.

[6] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. Hidden fluid mechanics: A Navier-Stokes Informed Deep Learning Framework for Assimilating Flow Visualization Data, 2018.

[7] Yinlin Ye, Yajing Li, Hongtao Fan, Xinyi Liu, and Hongbing Zhang. Deep neural network methods for solving forward and inverse problems of time fractional diffusion equations with conformable derivative, 2021.

A. Poisson Problem

A.1. Problem Formulation

In this section of the appendix we report the results obtained for the trivial test case of the Poisson Problem. The problem reads as:

$$\begin{cases} -\Delta u = 2sin(x)sin(y) & \text{in } \Omega = (0, 2\pi)^2 \\ u = 0 & \text{on } \Gamma_D := (0, 2\pi) \times \{0, 2\pi\} \\ u_x = sin(y) & \text{on } \Gamma_N := \{0, 2\pi\} \times (0, 2\pi) \end{cases} \tag{17}$$

This is indeed a toy problem, where there is only one scalar unknown and the analytical solution is known:

$$u_{ex}(x, y) = sin(x)sin(y)$$

A.2. Results

In Figure 21 we report the trend of the losses: after 10.000 training epochs, the test loss stabilizes itself at a value close to 10^{-7}. Note that the loss value is not *exactly* the error value, but it is a measure of the error, since it corresponds to the Mean Squared Error on the test set.

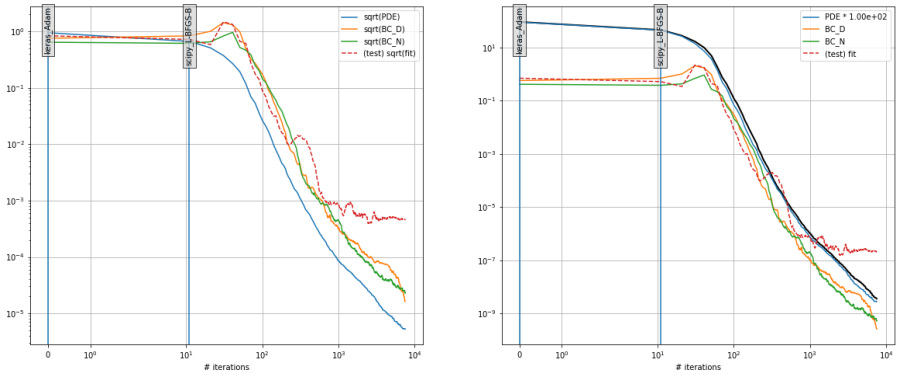
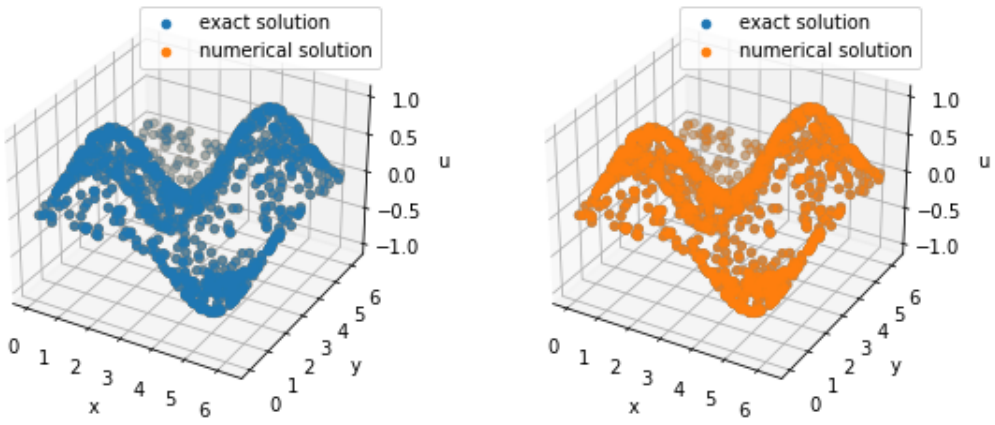


Figure 21: Loss trends for the Poisson test case.

In Figure 22 we show a comparison between the analytical and the numerical (PINN) solution. The accuracy of the results obtained can be assessed even considering the $||\cdot||_\infty$ norm of the error:

$$||u - u_{ex}||_\infty = 0.002$$



(a) Exact solution. (b) PINN solution.

Figure 22: Exact and PINN solution for the Poisson problem with mixed boundary conditions.

B. Technical Notes

B.1. Nisaba

`nisaba` is a Python library built on top of TensorFlow, exploited in Machine Learning for scientific computing. It is the reference library for handling derivatives in this project, as the functions used to compute gradients, divergences and laplacians are taken from the module `nisaba.experimental.physics.tens_style`, renamed as `operator` in our code for the sake of clearness.
We also remark the necessity to call `tf.GradientTape` each time we compute derivatives inside a physical loss: indeed, it is needed to record operations for automatic differentiation.

B.2. Fenics and Docker

In section 5 and section 6 we exploited numerical solutions to train and test our PINN.
They were obtained using the Python package for Finite Elements `FeNiCS` to generate `.h5` files with the numerical solutions (with a Windows architecture, this can be done for example inside a `Docker` container with `FeNiCS` installed).
We now draw the general structure of a `FeNiCS` script:

1. Loading or creation of a Mesh: for simple geometries such as cavity, we can work directly inside `FeNiCS`, while for complicated ones, such as the coronary, it is necessary to generate them with a software such as `GMSH` and then import it to `FenNiCS`.
2. Definition of domain boundaries and of Dirichlet Boundary conditions.
3. Definition of the weak formulation of the problem: *Trial* Space, *Test* Space, bilinear form and right-hand side, to pass as input to `FeNiCS` solver.
4. Calling to the solver; if the problem is time-dependent, we should first discretize it in time with finite differences, then call the solver at each time step.

B.3. ParaView

As a visualization tool for numerical solutions, we exploited `ParaView` to read `.xdmf` files; for time-dependent solutions, it enabled us to create an animation of the simulation in the desired time range, too.

C. Losses Implementation

In this section, we report the code snippets associated to the several type of losses mentioned in the project.

C.1. PDE Losses

We report below the implementation of the losses associated to the partial differential equations of the Navier-Stokes system.

- The loss associated to the *mass conservation equation* reads as:

```
def PDE_MASS():
    x = tf.gather(dom_grid, idx_set["PDE"])
    with ns.GradientTape(persistent=True) as tape:
        tape.watch(x)
        u_vect = model(x)[:,0:2]
    return divergence(tape, u_vect, x, dim)
```

- The *momentum conservation* loss, denoting by k the component to consider, reads as:

```
def PDE_MOM(k):
    x = tf.gather(dom_grid, idx_set["PDE"])
    with ns.GradientTape(persistent=True) as tape:
        tape.watch(x)
        u_vect = model(x)
        p    = u_vect[:,2] * norm_pre
        u_eq = u_vect[:,k] * norm_vel

        grad_eq = gradient(tape, u_eq, x)
```

```
        dp    = gradient(tape, p, x)[:,k]
        deqx = grad_eq[:,0]
        deqy = grad_eq[:,1]
        lapl_eq = laplacian(tape, u_eq, x, dim)

        unnormed_lhs = rho * (u_vect[:,0] * deqx + u_vect[:,1] * deqy)
                     - mu * (lapl_eq) + dp
        norm_const = 1/max(norm_pre,norm_vel)

    return unnormed_lhs*norm_const
```

C.2. Dirichlet-Style Losses

We report the code used for all losses aimed at imposing an exact value on a set of points.
With this general implementation, we were able to avoid code reusing by exploiting the same function for:

- Dirichlet Losses
- Fitting Losses
- Test Losses

```
def dir_loss(points, component, rhs):
    uk = model(points)
    uk = uk[:,component]
    return uk - rhs
```

C.3. Neumann Boundary Losses

The implementation of the Neumann Boundary Conditions slightly varies among the test cases.

C.3.1. Poiseuille Problem

```
def BC_N():
    with ns.GradientTape(persistent = True) as tape:
        tape.watch(x_BC_N)
        u = model(x_BC_N)
        u_x = operator.gradient_scalar(tape, u, x_BC_N)[:,0]
    return u_x - g
```

C.3.2. Coronary Flow

```
def neu_loss(edge,k,rhs):
    x = bnd_pts[edge]
    n = tf.constant([[2.0],[1.0]]) if edge == 'OUT1' else  tf.constant([[1.0],[0.0]])
    with ns.GradientTape(persistent=True) as tape:
        tape.watch(x)
    u_eq = model(x)[:,k] * norm_vel
    p_eq = model(x)[:,2] * norm_pre
    grad = gradient(tape, u_eq, x)[:,0:2]
    return nu * tf.linalg.matmul(grad,n) - p_eq * n[k] - rhs
```

C.4. Pressure imposition Loss

We report the implementation of the loss for strategy of imposing null pressure mean in subsection 4.2.

```
def PRESS_0(x):
    uk = model(x)[:,2]
    uk_mean = tf.abs(tf.math.reduce_mean(uk))
    return uk_mean
```

D. Code snipptes - Utilities

D.1. Point Sets Construction

We report the code for the extraction of the mesh from the `.h5` file (section 5 and section 6):

```
mesh_h5 = h5py.File(f'{folder_h5}/steady_coronary_steady.h5', "r")['Mesh']['0']
          ['mesh']['geometry']

x_vec = mesh_h5[:,0]
y_vec = mesh_h5[:,1]
```

D.2. Generation of the domain grid (subsection 5.4)

The code reads as:

```
x_vec = np.linspace(Le_x, Ue_x, n1+1) if uniform_mesh else np.random.uniform(Le_x,
    Ue_x, n1+1)
y_vec = np.linspace(Le_y, Ue_y, n2+1) if uniform_mesh else np.random.uniform(Le_y,
    Ue_y, n2+1)
time_vec = np.arange(0.0, T, step = dt)

dom_grid = tf.convert_to_tensor([(t,i,j) for t in time_vec for j in y_vec for i in
    x_vec])
```

D.3. Noise Generation

```
def generate_noise(n_pts, factor = 0, sd = 1.0, mn = 0.0):
    noise = tf.random.normal([n_pts], mean=mn, stddev=sd, dtype= np.double)
    return noise * factor
```