Hes·so VALAIS WALLIS

Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

BACHELOR THESIS

Guillaume Bessard

Physics Informed Neural Networks

Informatique et Systèmes de communication (ISC)

ISC-ID-25-4

University of Applied Sciences and Arts Western Switzerland
Faculty of Engineering and Architecture
Haute École d'Ingénierie de Sion

Guillaume Bessard

Physics Informed Neural Networks

Thesis submitted to the Bachelor degree programme
Informatique et Systèmes de communication (ISC) – Data engineering major
Haute École d'Ingénierie de Sion

Supervisor: **PROF. DR. FLORIAN DESMONS**
Co-supervisor: **PROF. DR. LOUIS LETTRY**

Expert: **DR. ORESTIS MALASPINAS**

Submitted on: July 30, 2025

Abstract

In recent years, neural networks have become an important part of the Computer Science field of study. Their apparent ability to learn most situations given enough time and context led to their application in a wide array of different problems. This thesis will look at a recent development in machine learning and neural networks: **Physics Informed Neural Networks** (PINNs). The aim of PINNs is to learn a task while respecting given laws of physics, described as Partial Differential Equations (PDEs). In this thesis, we wanted to explore the use of PINNs in the context of fluid mechanics and compare the results we got to a more conventional approach. The goal was to train a first model for a steady-state **Poiseuille flow** and **Von Karman instability**. We then wanted to train a model to simulate an unsteady-state for the Von Karman instability. The final goal was to compare the results we got from PINNs and the conventional analytical simulation and understand the advantages and disadvantages of PINNs. In the end, we were unable to achieve any of these targets, but the different iterations of the model developed are a great testament to the problems that can arise when working with PINNs, the solutions we used to try to solve them, and the effect they had.

Keywords : Engineering, Machine learning, PINN, Navier-Stokes

Résumé

Depuis quelques années maintenant, les Réseaux Neuronaux sont devenus un sujet immanquable dans l'ingénierie informatique. Leur capacité d'apprentissage dans de nombreux domaines, avec suffisamment de temps et de contexte, a motivé leur utilisation dans un large éventail de problèmes différents. Ce travail de bachelor a pour but de se pencher sur un développement récent de l'apprentissage par ordinateur et des réseaux neuronaux, les **Réseaux Neuronaux Informé par la Physique** (PINNs). Leur fonctionnement est similaire à celui des réseaux neuronaux classiques, mais leur apprentissage est guidé par des équations de la physique, données sous forme d'équations différentielles partielles (PDEs). Dans cette thèse, nous voulions analyser l'utilisation de PINNs dans le contexte de la mécanique des fluides, et comparer les résultats obtenus avec les méthodes conventionnelles dans ce domaine. Le but était tout d'abord d'entrainer un modèle pour simuler un **écoulement de Poiseuille** et une **instabilité de Von Karman** en régime stationnaire. Ensuite, il était question de simuler une instabilité de Von Karman en régime instationnaire. Suite à ces simulations, le but était de comparer les résultats obtenus avec les solutions analytiques conventionnelles, afin de comprendre les avantages et inconvénients des PINNs. Au final, nous n'avons pas été en mesure d'atteindre ces objectifs, mais les différentes itérations des modèles développés montrent directement les problèmes pouvant apparaître lors de l'utilisation de PINNs, les solutions que nous avons essayé de mettre en place et les effets de ces dernières sur les résultats.

Keywords : Engineering, Machine learning, PINN, Navier-Stokes

Table of Contents

1 – Introduction 11
2 – State of the art 13
3 – Development and Methodology 15
3.1 – Problem setup 15
3.2 – Model setup 15
3.3 – Domain definition 16
3.4 – Navier-Stokes 17
3.5 – Deployment on computing server 18
4 – Results and Discussion 19
4.1 – Domain 19
4.2 – Local optimal 19
4.3 – Trying to avoid the local minimum 23
4.4 – Curriculum Regularization 23
4.5 – Best results 26
5 – Conclusion 29
Bibliography 31
Appendix 33

Chapter 1 – Introduction

Recent years have seen a boom in the use of neural networks in a wide area of domains. They are now mainstream in a lot of fields, from LLMs to image classification models and more. Their successfulness reside in the fact that they are able to utilize some computing devices — GPUs — astonishingly efficiently. When looking at that fact, one might ask themself: what are some computationally expensive problems that could benefit from this? A team at the University of Pennsylvania managed to apply this to the world of physics simulation, by developing neural networks "guided" by physics equations: **Physics Informed Neural Networks** [1]

This can avoid a lot of costly computation in many applications of physics simulations. In our case, we will focus on incompressible flow simulation. The aim of the thesis is to compare PINNs against the conventional analytical methods to solve a few problems:

- Steady-state Poiseuille Flow
- Steady-state Von Karman instability
- Unsteady-state Von Karman instability

For each problem, we then want to analyze the results by comparing them to results from the analytical methods to understand the advantages and disadvantages of PINNs

PINNs take advantage of the capacity of deep neural networks to be universal function approximators [2]. In turn, this means that we have to use the PINN to approximate the physical equations for our case of study. For us, dealing with incompressible flow, we will use the Navier-Stokes equations to guide our neural network. Specifically, we have to use the partial differential equations (PDEs) from Navier-Stokes in the loss function of our model

Another important part of training a PINN are the boundary conditions (BCs). In order to have a well defined domain of simulation, the model needs to be informed about how to act on the simulation's boundaries. These conditions will define the inlet, outlet and walls of the simulation.

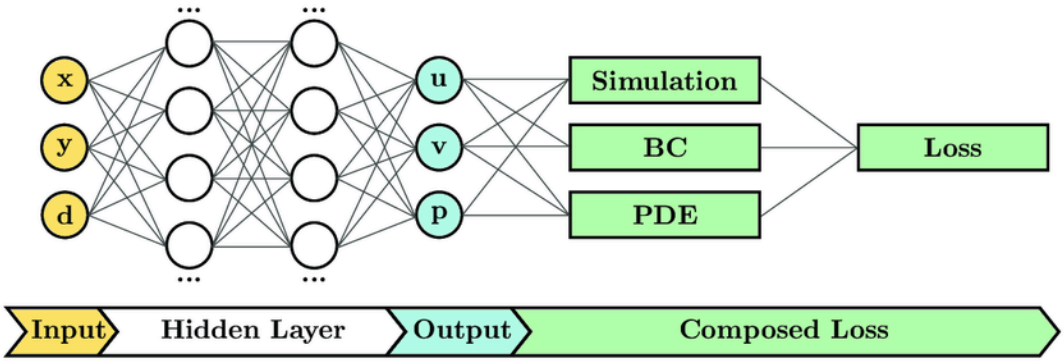


Figure 1 - Example architecture of a PINN [3]

Figure 1 represents a simple PINN architecture. It shows the use of the BC and PDE in the loss function of the model. This figure also shows the possibility of adding a data driven loss, with data collected from previous simulations or observations[1].

[1] In our case, we will base the model only on the PDE and BC, this schema is here used to show the global architecture of a PINN.

Chapter 2 – State of the art

PINNs are a relatively new idea in the world of machine learning. This thesis is based principally on the 2019 paper by M. Raissi *et al.*[1] The motivation for their work was to be able to resolve nonlinear problems directly, without having to resort to usual analytical workarounds. They also exploited the recent development in automatic differentiation[4] to differentiate the neural networks. This is the important reason why PINNs work. As neural networks are themselves differentiable, it allows us to use automatic differentiation when computing the physics loss and training the model.

In the paper, the team applies PINNs successfully to different problems, including a Karman vortex street. This means we know that PINNs works well for the problem we're trying to solve. Because of that, we don't have to worry about if we can get correct results for our tasks but how to refine he model to get the right results.

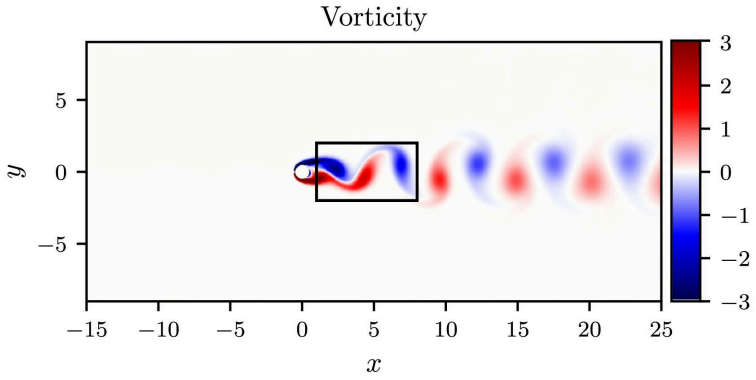


Figure 2 - PINN result from the 2019 paper: Karman vortex street, M. Raissi *et al.*[1]

Despite being recent, a lot of research and popularization has been made around PINNs. While gathering information, we found a paper about the usual failure modes of PINNs and possible fixes[5]. This might be very helpful in case we have trouble making our model learn the situation. The Youtube video[6] by user "Steve Brunton" was also a good introduction resource to first get into what PINNs are. These resources along with explanations from Prof. Lettry helped to understand what would have to be implemented to solve the different problems we have.

One of the central part of this project are the Navier-Stokes equations. Thankfully, these equations are very famous for being one of the Millennium Prize Problems, meaning they are often discussed and popularized. Resources like the Wikipedia page for the Navier-Stokes equations, youtube popularization videos like this one[7] from user "vcubingx" or explanations from Prof. Desmons helped a lot when trying to understand what these equations mean.

Chapter 3 – Development and Methodology

3.1 – Problem setup

Poiseuille flow is the first problem we have to tackle. We want to simulate a non-turbulent flow for an incompressible fluid between two infinitely long parallel plates, and with a constant pressure gradient. We have to set the parameters right for flow to be non-turbulent, and we can do that by looking at the Reynolds Number (Re). According to https://www.simscale.com/docs/simwiki/numerics-background/what-is-the-reynolds-number/, the flow should be laminar with a Reynolds number under 2300.

We set up the parameters for a low Reynolds number and the physical properties of water, and tweak the flow speed and channel height to get an adequate Re number:

$$\mathrm{Re} = \frac{\rho \cdot u \cdot 2h}{\mu}$$

Where:
- ρ is the density of the fluid [kg/m^3]
- u is the average flow velocity [m/s]
- h is the half-height of the channel [m]
- μ is the dynamic viscosity of the fluid [Pa · s]

We use the following values:

- $\rho = 1000$ [kg/m^3]
- $u = 0.01$ [m/s]
- $h = 0.01$ [m]
- $\mu = 1e-3$ [Pa · s]

Substituting the values into the equation:

$$\mathrm{Re} = \frac{\rho \cdot u \cdot 2h}{\mu} = \frac{1000 \cdot 0.01 \cdot 2 \cdot 0.01}{0.001} = \frac{0.2}{0.001} = 200$$

Hence, the Reynolds number is **200**, way under 2300, which guarantees non-turbulent flow

3.2 – Model setup

We use the PyTorch library to create and train our neural network. PyTorch has become the go-to library for neural network in python. PyTorch also includes an implementation of the automatic differentiation we talked about previously (autograd). The neural network was created with 5 interconnected hidden layers of 100 neurons at first, as suggested in the 2019 paper by M. Raissi *et al.* We don't know for sure if this is also valid in our situation, so the shape of the PINN might change during the testing. A very important part of PINNs is the neurons' activation functions. As we have second derivatives in the loss function (from the Navier-Stokes PDE), we need to be sure that the activation function is differentiable two times, and meaningfully.

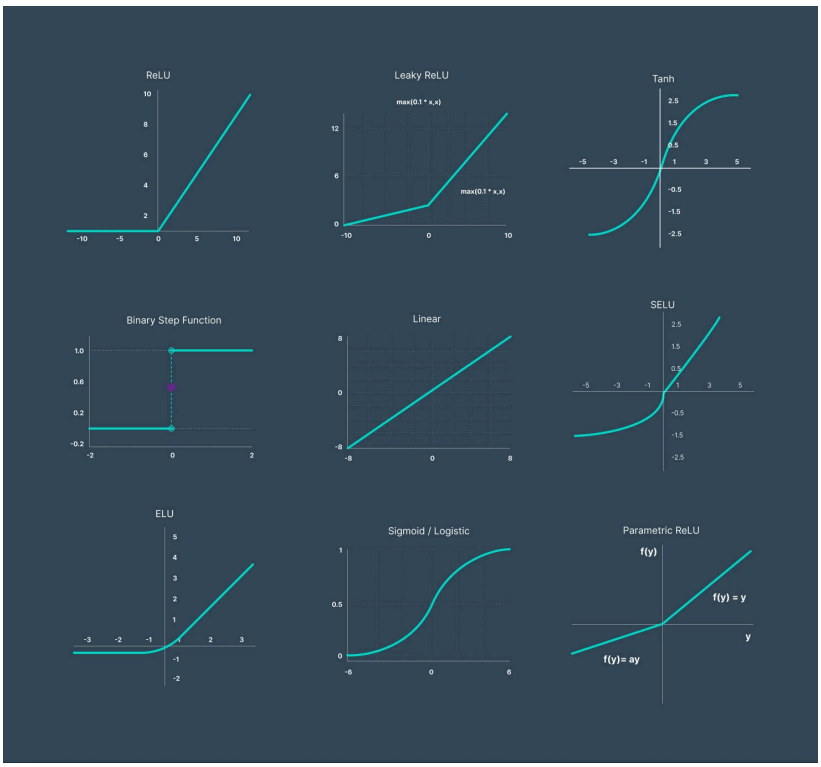


Figure 3 - Common activation functions for neural networks, from v7labs

On Figure 3, we can see a couple different activation functions. We can understand right away that functions like the ReLU of leaky ReLU ones wouldn't work in our case, because they are not differentiable. We also have to be careful about the result of the second derivative. For example, the linear activation function would just die out after the first derivative. The tanh activation function seems to be a good fit for PINNs, and it's what was used in the 2019 paper, so we will start with this.

For the optimizer, we choose an ADAM optimizer with a learning rate of 1e-4 to begin. learning rate decay is not fixed as we need to see how fast the system learns.

3.3 – Domain definition

For the learning domain, we need a long enough channel for the flow to develop itself fully, and we need to incorporate all the necessary boundary conditions. As we saw, the half height of the channel was set to 0.01 [m]. A domain length of 0.4 [m] was chosen, which should be long enough for the flow to develop, but not too long in order to lower training time.

Boundary condition were implemented as zones surrounding the flow channel. The "walls" span the complete length of the simulation and are 0.01 [m] wide. The inlet spans the height of the channel and is 0.01 [m] long, same for the outlet.

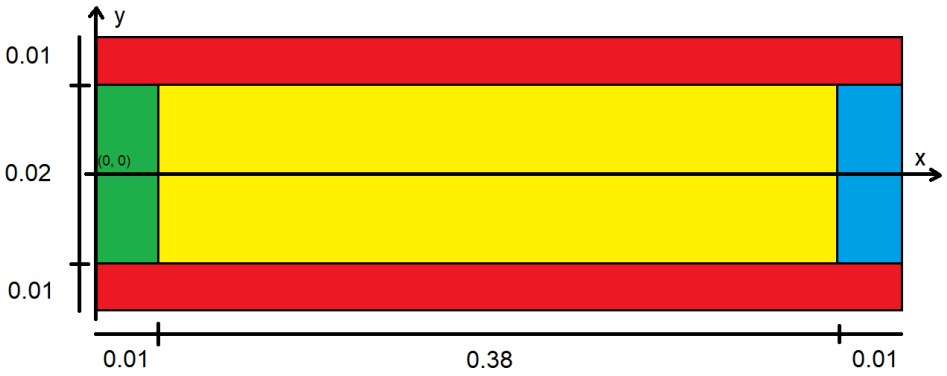


Figure 4 - Visualization of the domain with zone sizes in meters. Red zones are the walls, green is the inlet, blue is the outlet and yellow is the channel

Once we have defined the domain, we need to generate points in every zone. We will start with the same points density in every zones in the domain. We can then later generate different densities if necessary or meaningful. The code is made so that the density can be changed later.

The code relating to the domain definition and points generation can be found in the appendix at (fill: blue) [Listing 1].

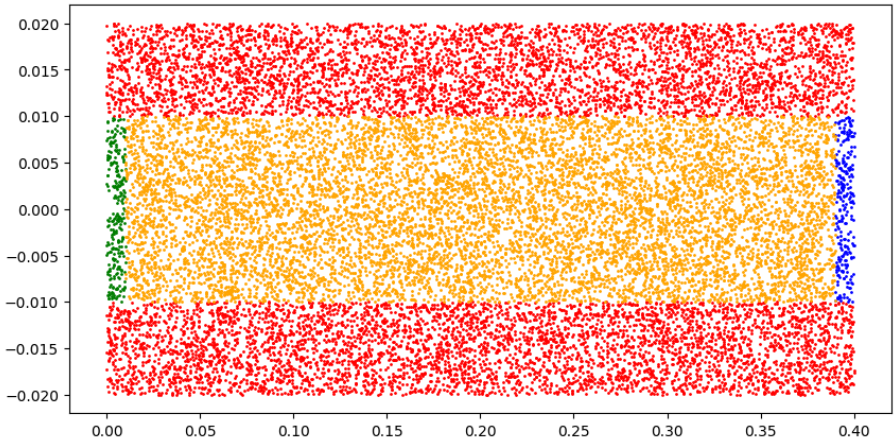


Figure 5 - Visualization of the points generated by the code, with the same density on the whole domain (20000 points)

Each of these point represents a x and y speed value, as well as a pressure value, and this is how the state of the system is represented.

With the defined zones, we can now start to build our loss functions. So the boundary condition will be:

- Green zone : flow must be equal to target velocity in x direction, and pressure must be 0
- Red zone: flow speed must be 0 in x and y direction (no slip condition)
- Blue zone: Gradient in the y direction must be 0 (fully developed flow)
- Yellow zone: see Section 3.4

3.4 – Navier-Stokes

The Navier-Stokes equations are partial differential equations which describe the motion of viscous fluid substances. Mathematically, they express both momentum and conservation of mass for a fluid, and they arise from applying Isaac Newton's second law to fluid motion. These equations look like this:

$$\rho \frac{dV}{dt} = \rho g \cdot \nabla p \cdot \mu \nabla^2 V$$

$$\nabla \cdot V = 0$$

Where V is the velocity vector field and p is the pressure. This is the complete form taking g, the gravitation, into account, but four our purpose we can ignore it. The first equation is our conservation of momentum, and the second one is our conservation of mass. We now want to rewrite the equations using the x and y direction flow velocity.

$$\rho\left(\frac{\delta u}{\delta t} + u\frac{\delta u}{\delta x} + v\frac{\delta u}{\delta y}\right) = -\frac{\delta p}{\delta x} + \mu\left(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}\right)$$

$$\rho\left(\frac{\delta v}{\delta t} + u\frac{\delta v}{\delta x} + v\frac{\delta v}{\delta y}\right) = -\frac{\delta p}{\delta y} + \mu\left(\frac{\delta^2 v}{\delta x^2} + \frac{\delta^2 v}{\delta y^2}\right)$$

$$\frac{du}{dx} + \frac{du}{dy} = 0$$

And these are our PDEs, and the terms we have to minimize in the loss function four our model to be physically accurate. We can see it implemented at (fill: blue)[Listing 2], with the PDEs

3.5 – Deployment on computing server

Training the model on a personal computer would take a very long time, so we asked access to the computing server of the HES-SO in order to speed things up. To run tasks on the server, we have to use SLURM[2]. For our script to be executed with SLURM, we will have to containerize it using Apptainer[3] The Apptainer definition file and the SLURM script can be found in the appendix, under Listing 3 and (fill: blue) [Listing 4].

The code for this thesis can be found at https://github.com/GuilBess/TB_PINN

[2]SLURM is a workload manager, used to share resources between different tasks from different users, see https://slurm.schedmd.com/documentation.html

[3]Apptainer is a container system, used a lot in HPC, see https://apptainer.org/

Chapter 4 – Results and Discussion

As stated previously, we did not manage to get physics accurate results while trying to solve Poiseuille flow. This next section will go over the results we got, analyze in which way it fails to yield correct simulations and explains what was done to improve the model. During the development and refining of the model, a few major problems were spotted and we will go over them in the next section.

4.1 – Domain

In the first iterations of the project, the aim was to build a model that was running and implemented all the different losses. We focused on the PINN part so that we could have a base on which to improve. The main improvement were made on the simulation domain. Here is what the domain looked like before.

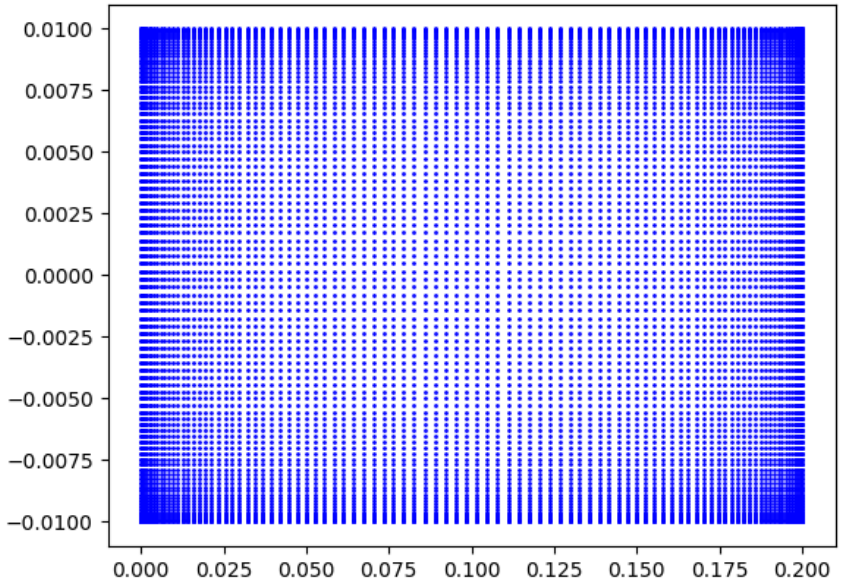


Figure 6 - First domain generation

The idea here was to have higher points density around the edges of the channel, where the model should have a more difficult time to learn the solution. In this iteration, the boundary conditions were not yet zones but were defined as the points at the edges of the channel. The channel was also half as long as described previously.

This domain could pose problems for the model in some ways. The imbalance of points in each zone, and the fact that they each have a different loss function assigned can make it hard to balance the weights of the losses correctly. This is why we later expanded the zones in our domain.

The point generation made it so the model would always see the same points, which makes learning more difficult, and the part that we are interested in to get the results (halfway point in the channel length) is where the lowest point density is, which doesn't make sense. If we want the comparison measure to be made there, this part of the simulation should be as accurate as possible, so we should increase the density of points if necessary.

From this domain, and after discussion with Prof. Desmons and Prof. Lettry, we built the new domain presented in Section 3.3. The improvements include the larger BC zones, random points generation and point density control.

4.2 – Local optimal

The next big problem was getting out of local optimal solutions. During testing, we found that our model really wants to either "average" the flow on the whole domain, or make the flow quickly dies out, and get

to 0 velocity in the channel in the x and y directions. These solutions are a lesser evil for the model when it can't find the global optimal: In both case, the physics loss is well respected on a big part of the domain. When in a constant, "averaged" flow or when the flow is 0 on most of the domain, we get a low physics loss. This is because the physic is respected on most of the domain in both cases. The transition between these two cases is linked to the weight on the wall boundary condition.

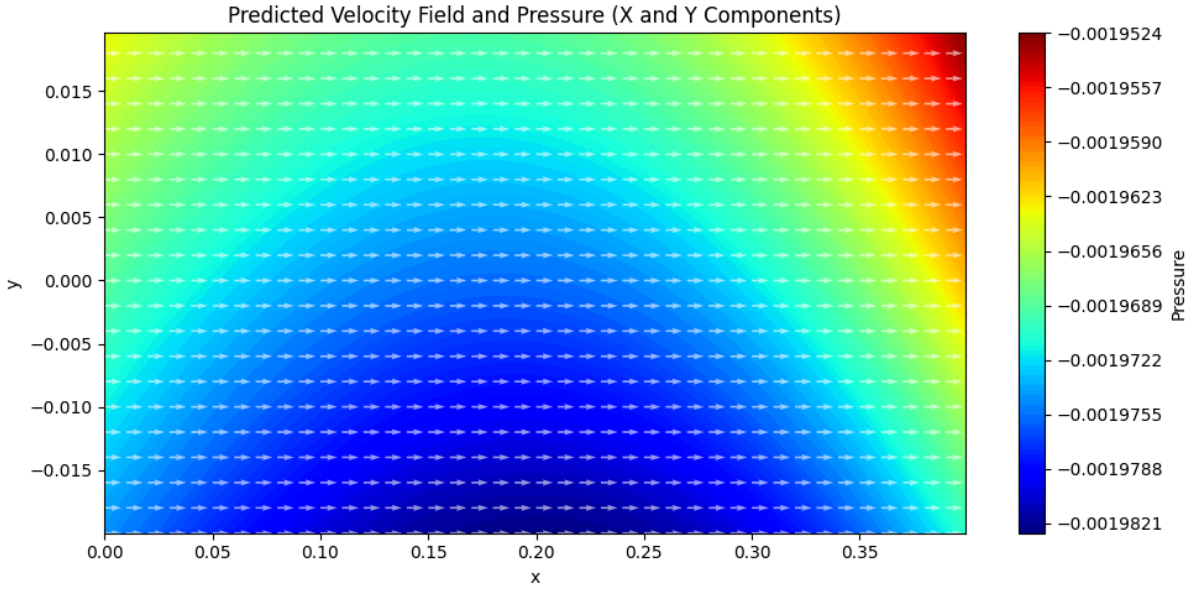


Figure 7 - Averaged situation, velocity and pressure

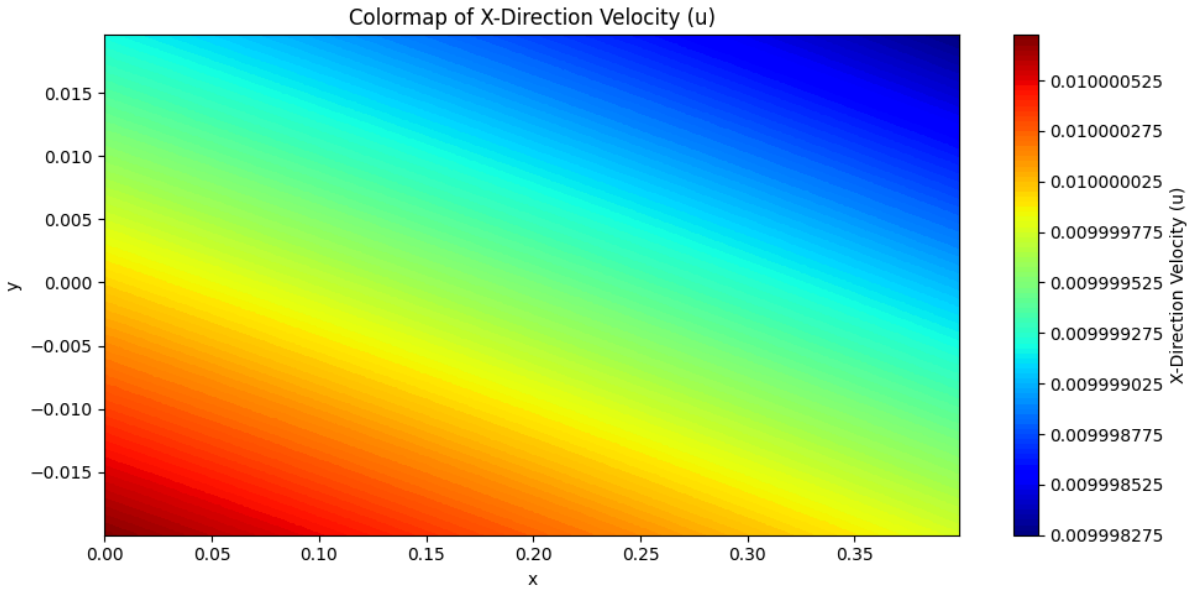


Figure 8 - Averaged situation, velocity colormap

In the Figure 7 and Figure 8, we see the flow speed being very constant on the whole domain, ignoring the walls. We can also see weird behavior in the pressure gradient. For a constant flow like this, we expect constant pressure over the whole domain. This was later explain by an error in the outlet boundary condition.

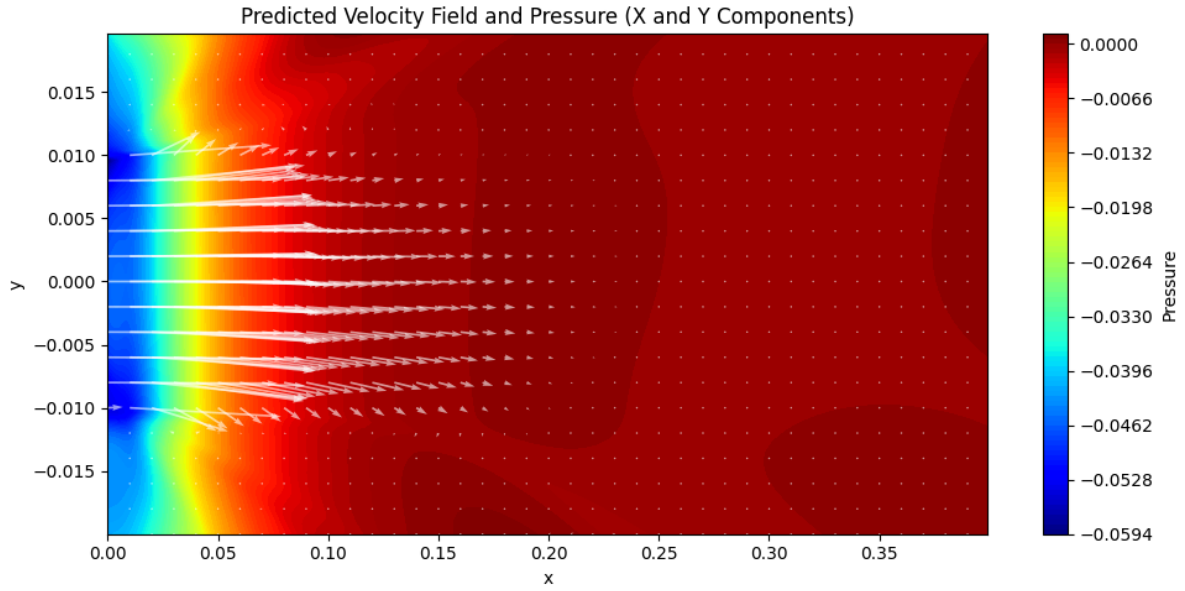


Figure 9 - dying flow situation, velocity and pressure

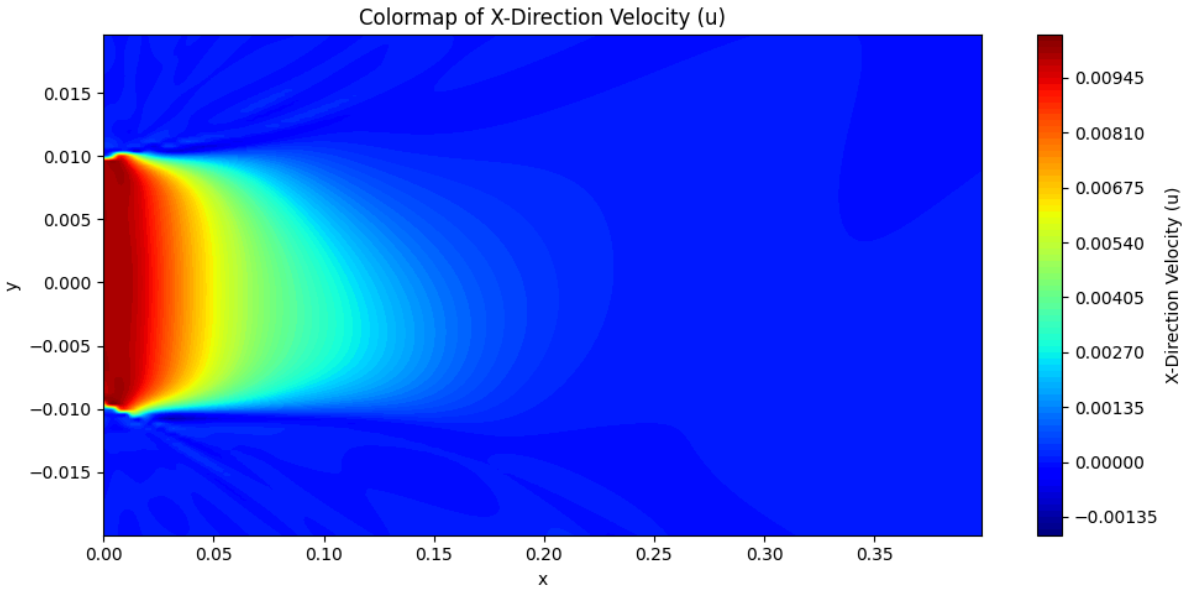


Figure 10 - dying flow situation, velocity colormap

In this case, we can see that the flow has trouble propagating. The part where the flow is 0 yields a low loss for the physics part, which makes it difficult for the model to learn the correct global behavior. Both these situations are due to the model being trapped in a local optimal. We can also still see that the pressure is wrong, this time the gradient looks more correct but is reversed. We expect the higher pressure to be at the inlet.

In order to fit a model, the neural network optimizer searches for the lowest point in the landscaped created by out loss function. This can be a problem if we have other local minimum that don't satisfy the situation perfectly. These might "trap" our model in a suboptimal state, with no way to explore other solutions.

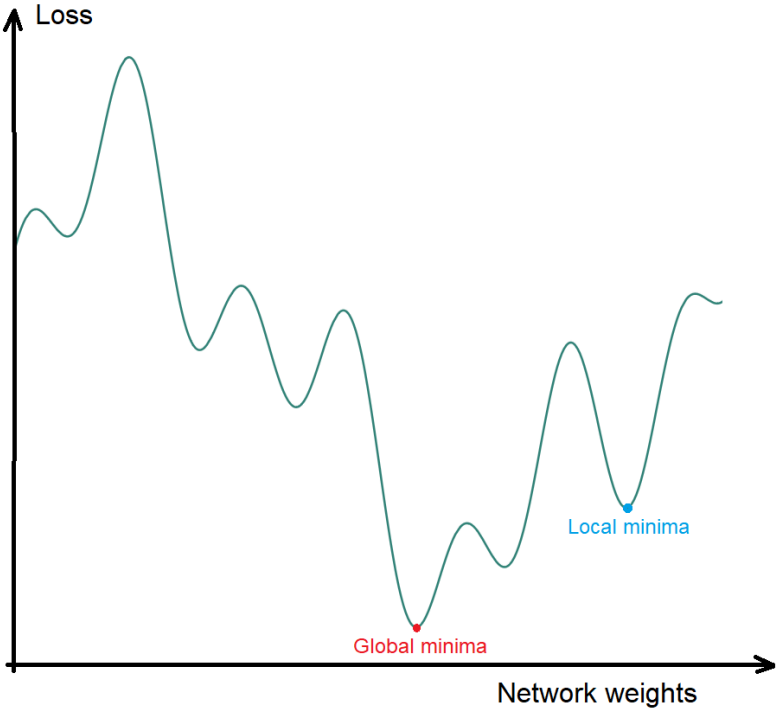


Figure 11 - Loss landscape example with local and global minimum

Figure 11 is a representation of a loss landscape. In this situation, we can see a deep local minimum, in blue, that could be hard to get out of for the optimizer. In this case, if we're at the right of the local minimum, we might end up trapped in it and not able to get out. In this example, the local minimum could be the situation where the flow is dying out. We might never reach the red global minimum, which would be the correct solution.

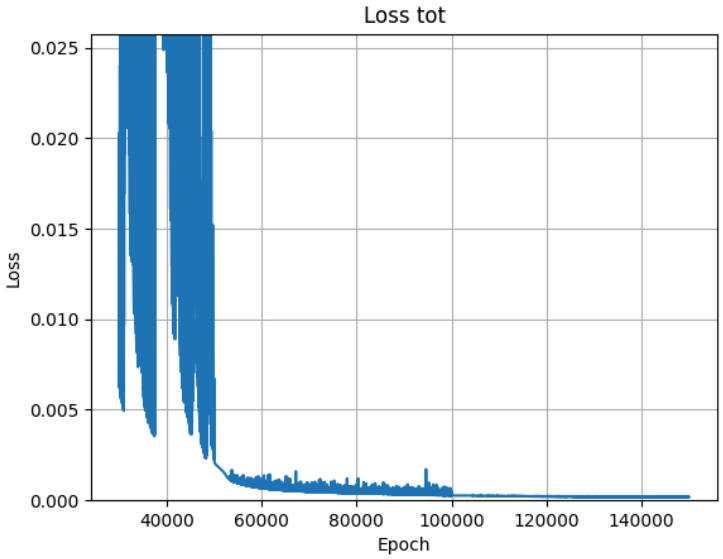


Figure 12 - Evolution of loss for the dying flow case

On Figure 12, we can see that the loss reaches a plateau and doesn't manage to find a better solution. As this solution is not correct, we can deduce that we are trapped in a local minimum. We also notice that the loss is very unstable before 50′000 epochs. The unstable loss was present throughout the development, as we didn't manage to fix it. In this case, we reduced the learning rate at 50′000 and 100′000 epochs. As seen on the graph, reducing the learning rate also reduces the instability of the loss, but it still stays noisy. This

might be a sign of a complex loss landscape too. We can see in the local loss of the interior zone where our error is the greatest.

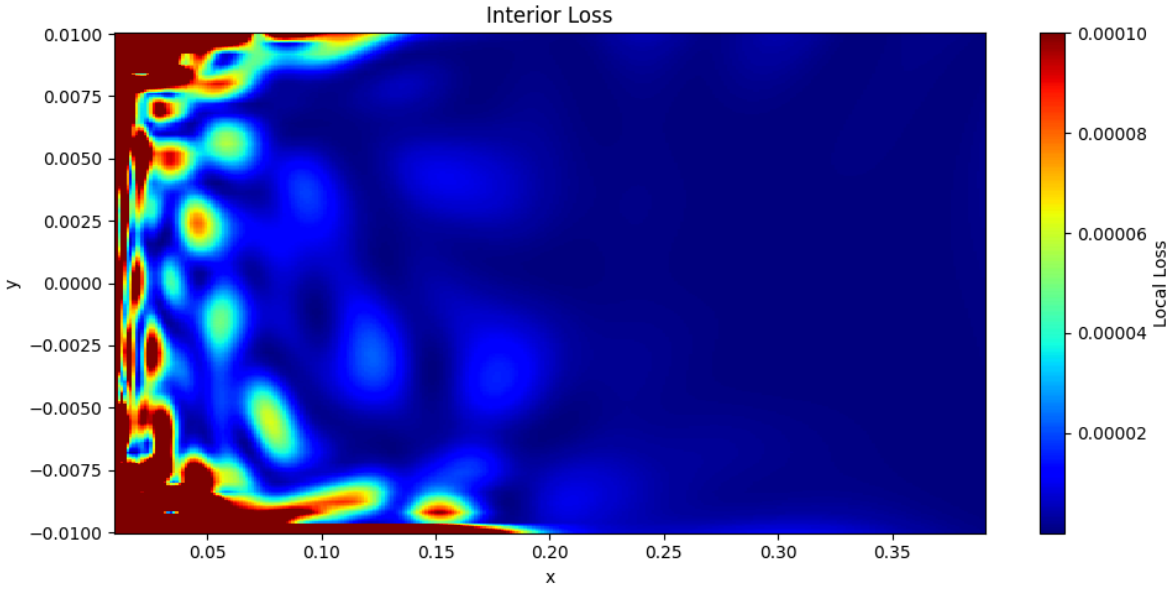


Figure 13 - Local loss for the interior zone, dying flow situation

In Figure 13, we can see that most of the interior loss is low, as this is the part where there is no flow, meaning the physics is right. We see a large error (which was clamped dow to 1e-4 for readability) near the other zones, which is expected as these are the most difficult point to learn in the domain, where 3 different losses interact near each other. We also see that the part where the flow is dying has a higher loss, but overall the average loss is very low, explaining why the model has a hard time getting out of this situation.

4.3 – Trying to avoid the local minimum

This mode of failure for a PINN is common, and the paper about failure modes by A. S. Krishnapriyan *et al.* [5] cited previously explains that usual regularization methods can lead to less smooth landscapes, making the problem harder to optimize.

This paper gives two way to regulate PINNs which work well. The first one is called *Curriculum Regularization* and the second one is *Sequence-to-sequence learning*

In *Curriculum regularization*, the goal is to, as said in the paper, “warm start” the PINN. By having the model learning a solution that is close to the true solution, we can put it in a good starting state. We can think of it as starting the PINN on the left of the global minimum on Figure 11, rather than on the right of the blue local minimum.

Sequence-to-sequence learning consists in training the model sequentially, on parts of the domain. The paper applies this method to time, but in our case we could apply it to vertical slices of the domain. This approach might help in the dying flow case, a it might help with flow propagation. This is because if the transition between flow and no flow take proportionally more place in the domain, as it would in a smaller slice, the loss would be higher and the model would avoid this situation.

4.4 – Curriculum Regularization

In order to have our net in a better starting configuration, we have to pick a starting case that is close to the result we expect. In this case, we want to avoid having the flow getting to a speed of 0. Our initial condition will be to force a constant flow of the same speed as the input in the channel. This will be done by adding a loss term that enforces the flow speed. Then over time the weight of this loss will be decreased slowly. We have to decrease the rate slowly for the optimizer to work properly.

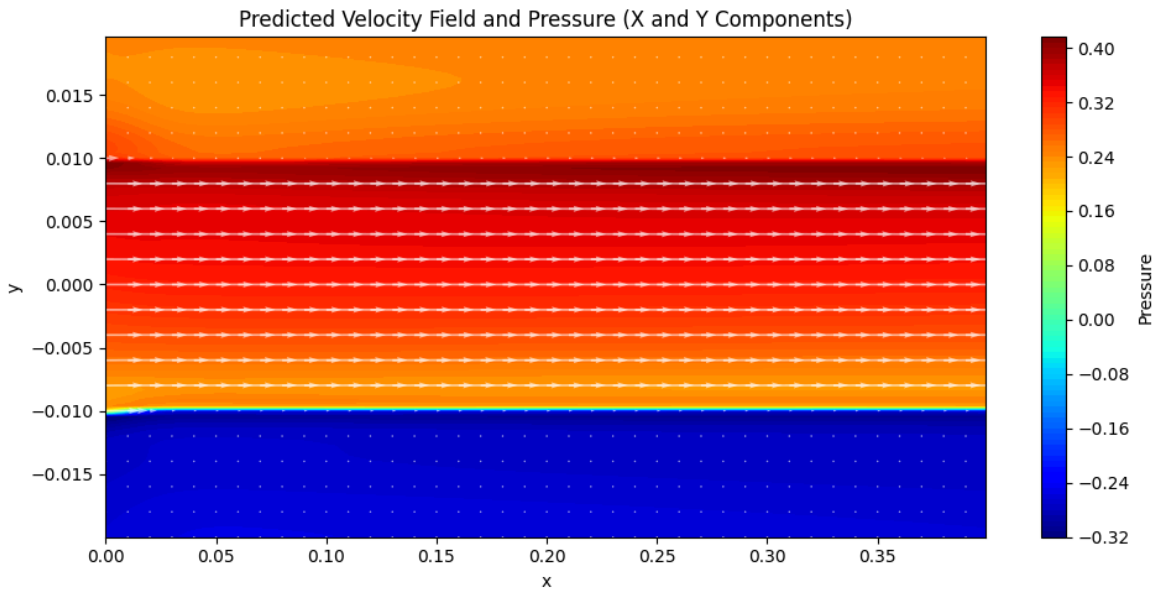


Figure 14 - Only initialization and BCs, velocity and pressure

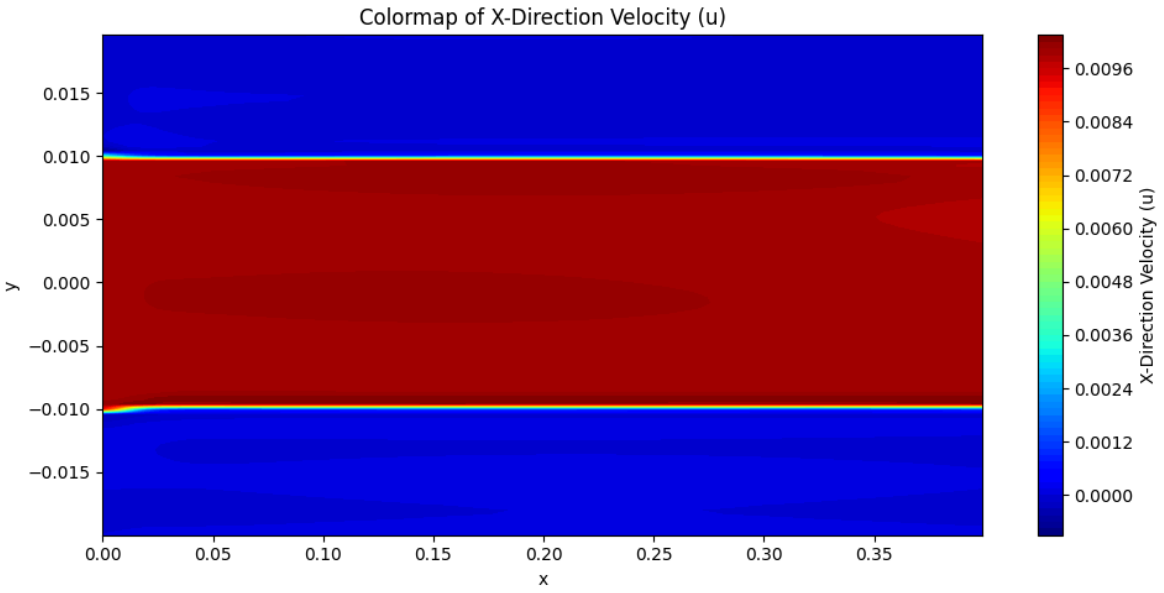


Figure 15 - Only initialization and BCs, velocity colormap

On Figure 14 and Figure 15 we can see that the model has no problem to produce a result with a constant channel flow speed and no wall flow speed. This test also tells us that it is possible to set the model in the situation we wanted. In Figure 14, we see that the pressure colormap is wrong, but this is because we don't fit the model on the pressure at all in this case, as the physics loss is not used. We now need to add the physics back.

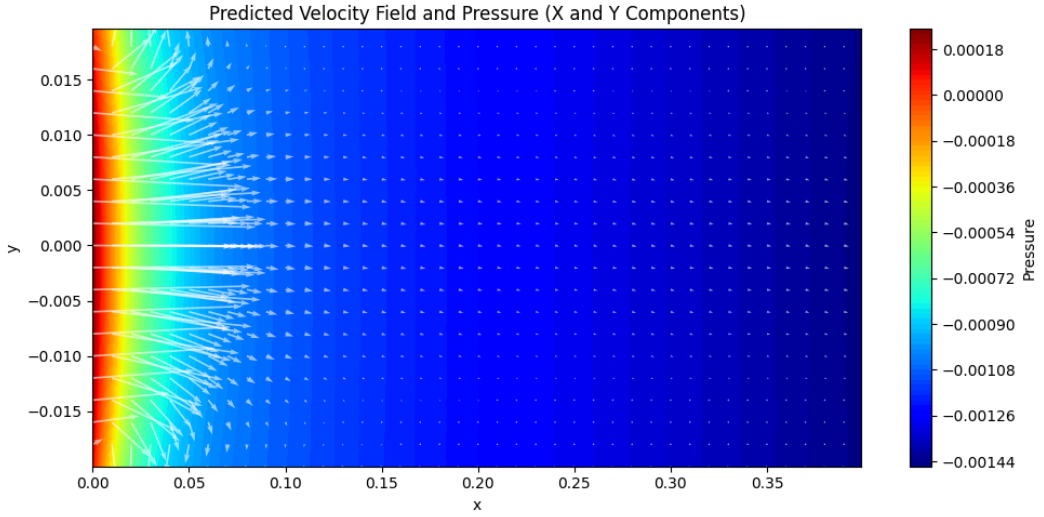


Figure 16 - Initialization start and physics, velocity and pressure

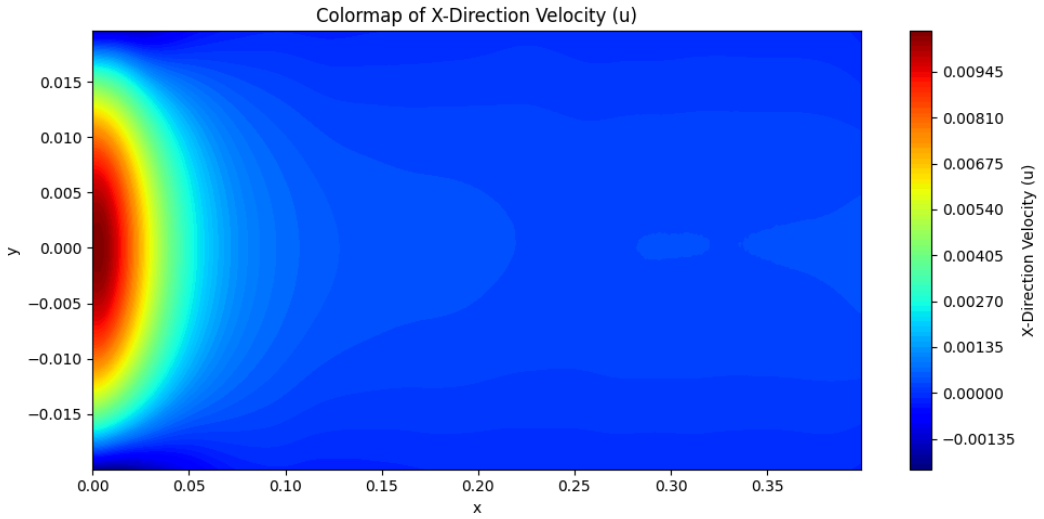


Figure 17 - Initialization start and physics, velocity colormap

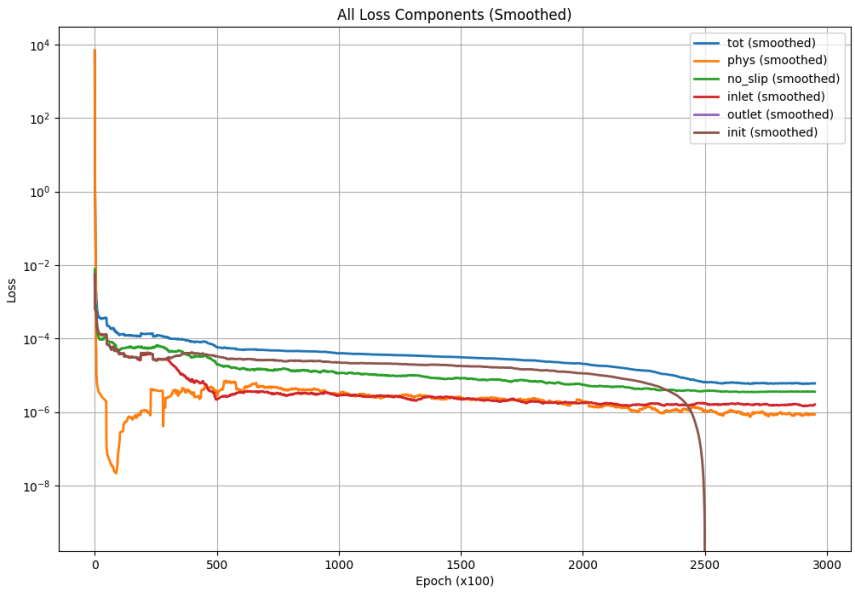


Figure 18 - Plot of the losses evolutions

We can see that even with the initialization, we do not manage to find the global minimum. In this case, the initialization weight is decreased linearly from 1 to 0 on the first 250'000 epochs. Although the walls are ignored, we do nonetheless notice that the model seems to act in a somewhat physical way in the left of the channel in Figure 16. The pressure gradient is also not reversed anymore.

4.5 – Best results

This part will present the best results we got thus far. During one of the earlier tests, we got one good run. Despite trying to set the model to the same hyperparameters again, we didn't manage to replicate this test. This may be due to a lucky random initialization that put us close to the global minimum in the loss landscape.

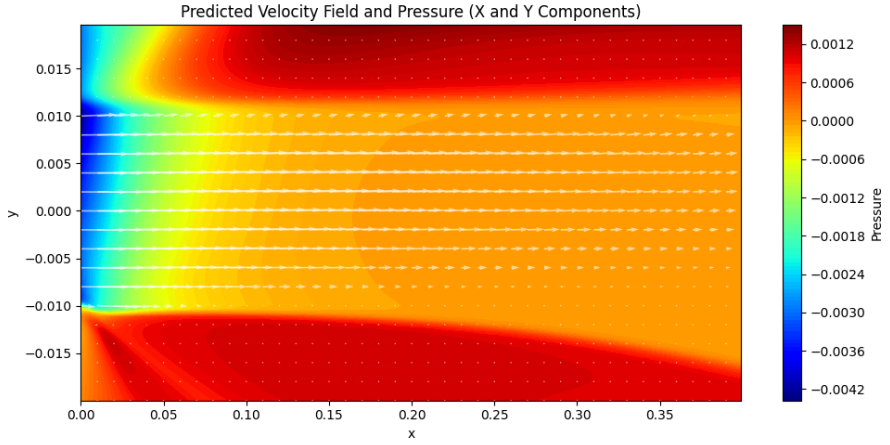


Figure 19 - Best found results, velocity and pressure

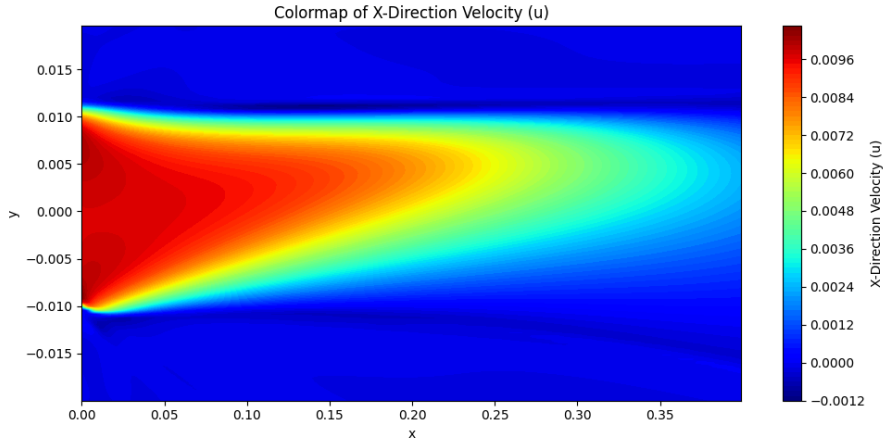


Figure 20 - Best found results, velocity colormap

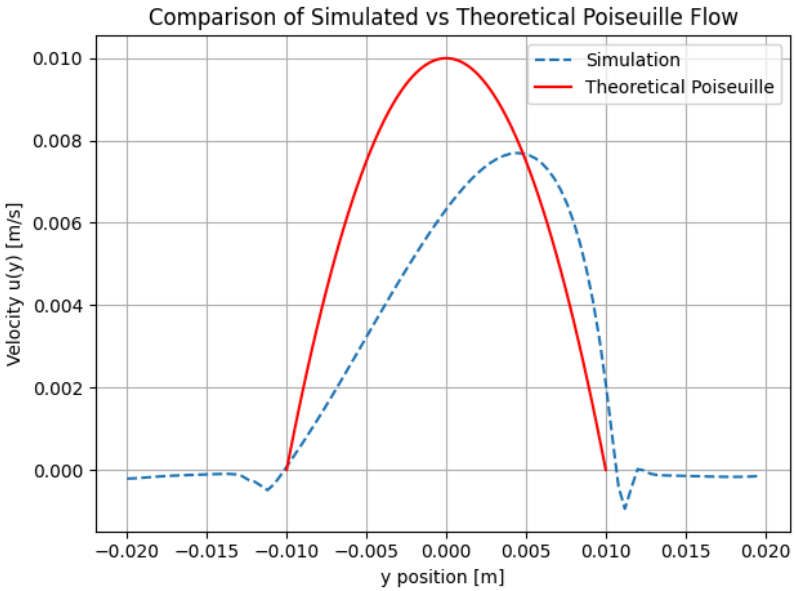


Figure 21 - Comparison of theoretical Poiseuille and simulation results at x=0.2

A few things here are wrong due to the fact that this was an early test. We see the pressure reversed again, and the flow is skewed, wich should not happen. But otherwise, this is the simulation acting the closest to what we expected.

Chapter 5 – Conclusion

In the end, we didn't manage to get a correct result for the Poiseuille flow. There are a couple reasons why the project didn't go as far as we wanted. We saw a couple of errors or assumptions made throughout the project which slowed down the development. First, we had to rethink the domain definition. Along with the modification of the BCs, relating to the pressure problem, this meant some of the tests were rendered useless. We also made the assumption that the only part of the domain that should fit on the physics was the channel part. In fact, with only the channel part, the physics could not learn the correct Poiseuille behavior, as it does not include any wall or inlet, meaning it would have no context.

A major part that slowed the project down was the testing. Running the model usually took between 30 minutes to 1 hour, and sometimes there would be a queue on the computing server, which is often running multiple tasks. Testing was also done in a suboptimal way. It's easy to run a test and wanting to quickly change just one parameter to see if it will work better, without really thinking about the results or noting down what we are changing. That led to a unmethodical approach to testing, with numerous test runs but limited documentation on what was being tested. This is the biggest lesson in this thesis for me, and an error I won't do again. This is clearly one of the reasons why the progress was slowed down.

A competent model isn't that far away, as we saw the physics and BCs being respected individually. The problem most likely is to find a good balance in the weights and to fix the physics domain to have an overlap with the BCs. With that, implementing the sequence-to-sequence learning principle discussed in Section 4.3.

The world of PINNs is vast and complex, tackling the newest technologies in machine learning as well as complex physical problems such as the Navier-Stokes equations. I want to thank Prof. Desmons and Prof. Lettry to have trusted me and given me the opportunity to work on such an interesting and new technology, and for their constant support and advice.

Bibliography

[1] M. Raissi, P. Perdikaris, and G. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,” *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019, doi: https://doi.org/10.1016/j.jcp.2018.10.045.

[2] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989, doi: https://doi.org/10.1016/0893-6080(89)90020-8.

[3] F. Pioch, J. H. Harmening, A. M. Müller, F.-J. Peitzmann, D. Schramm, and O. el Moctar, “Turbulence Modeling for Physics-Informed Neural Networks: Comparison of Different RANS Models for the Backward-Facing Step Flow,” *Fluids*, vol. 8, p. , 2023, doi: 10.3390/fluids8020043.

[4] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey.” [Online]. Available: https://arxiv.org/abs/1502.05767

[5] A. S. Krishnapriyan, A. Gholami, S. Zhe, R. M. Kirby, and M. W. Mahoney, “Characterizing possible failure modes in physics-informed neural networks,” *CoRR*, 2021, [Online]. Available: https://arxiv.org/abs/2109.01050

[6] S. Brunton, “Physics Informed Neural Networks (PINNs) [Physics Informed Machine Learning].” [Online]. Available: https://www.youtube.com/watch?v=-zrY7P2dVC4

[7] vcubingx, “The million dollar equation (Navier-Stokes equations).” [Online]. Available: https://www.youtube.com/watch?v=Ra7aQlenTb8

Appendix

Table of Acronyms

ADAM : Adaptive Moment Estimation
BC : Boundary Conditions
GPU : Graphics Processing Unit
LLM : Large Language Model
PDE : Partial Differential Equation
PINN : Physics Informed Neural Network
Re : Reynolds Number

List of Figures

Figure 1 Example architecture of a PINN [3] . . . 11
Figure 2 PINN result from the 2019 paper: Karman vortex street, M. Raissi *et al.*[1] . . . 13
Figure 3 Common activation functions for neural networks, from v7labs . . . 16
Figure 4 Visualization of the domain with zone sizes in meters. Red zones are the walls, green is the inlet, blue is the outlet and yellow is the channel . . . 17
Figure 5 Visualization of the points generated by the code, with the same density on the whole domain (20000 points) . . . 17
Figure 6 First domain generation . . . 19
Figure 7 Averaged situation, velocity and pressure . . . 20
Figure 8 Averaged situation, velocity colormap . . . 20
Figure 9 dying flow situation, velocity and pressure . . . 21
Figure 10 dying flow situation, velocity colormap . . . 21
Figure 11 Loss landscape example with local and global minimum . . . 22
Figure 12 Evolution of loss for the dying flow case . . . 22
Figure 13 Local loss for the interior zone, dying flow situation . . . 23
Figure 14 Only initialization and BCs, velocity and pressure . . . 24
Figure 15 Only initialization and BCs, velocity colormap . . . 24
Figure 16 Initialization start and physics, velocity and pressure . . . 25
Figure 17 Initialization start and physics, velocity colormap . . . 25
Figure 18 Plot of the losses evolutions . . . 25
Figure 19 Best found results, velocity and pressure . . . 26
Figure 20 Best found results, velocity colormap . . . 26
Figure 21 Comparison of theoretical Poiseuille and simulation results at x=0.2 . . . 27

Code Appendix

```python
def generate_points(N):
    # Define zones as (x_min, x_max, y_min, y_max, weight)
    zones = [
        (0, 0.4, -0.02, 0.02, 0.01),           # Base zone
        (0, 0.4, 0.01, 0.02, 0.00),          # Upper horizontal band
        (0, 0.4, -0.02, -0.01, 0.00),        # Lower horizontal band
        (0.19, 0.21, -0.01, 0.01, 0.0)           # Center square
    ]

    # Compute area and weighted area of each zone
    weighted_areas = []
    total_weighted_area = 0
    for x_min, x_max, y_min, y_max, weight in zones:
        area = (x_max - x_min) * (y_max - y_min)
        w_area = area * weight
        weighted_areas.append(w_area)
        total_weighted_area += w_area

    # Determine number of points per zone
    points_per_zone = [int((w_area / total_weighted_area) * N) for w_area in
weighted_areas]

    # Sample points in each zone
    all_points = []
    for i, (x_min, x_max, y_min, y_max, _) in enumerate(zones):
        count = points_per_zone[i]
        xs = np.random.uniform(x_min, x_max, count)
        ys = np.random.uniform(y_min, y_max, count)
        all_points.append(np.column_stack((xs, ys)))

    # Combine all points into a single array
    points = np.vstack(all_points)

    return points
```

Listing 1 - Code for points generation on domain

```python
# PDE computing for Navier-Stokes
def compute_PDE_full(xy, y_pred):
    u = y_pred[:, 0:1]
    v = y_pred[:, 1:2]
    p = y_pred[:, 2:3]

    # Gradient function
    grads = lambda out, inp: torch.autograd.grad(
        out, inp,
        grad_outputs=torch.ones_like(out),
        retain_graph=True,
        create_graph=True,
        only_inputs=True
    )[0]

    # First derivatives for ALL points
    grads_u = grads(u, xy)
    grads_v = grads(v, xy)
    grads_p = grads(p, xy)

    dudx, dudy = grads_u[:, 0:1], grads_u[:, 1:2]
    dvdx, dvdy = grads_v[:, 0:1], grads_v[:, 1:2]
    dpdx, dpdy = grads_p[:, 0:1], grads_p[:, 1:2]

```

```
    # Second derivatives for ALL points
    d2udx2 = grads(dudx, xy)[:, 0:1]
    d2udy2 = grads(dudy, xy)[:, 1:2]
    d2vdx2 = grads(dvdx, xy)[:, 0:1]
    d2vdy2 = grads(dvdy, xy)[:, 1:2]

    return dudx, dudy, dpdx, dpdy, d2udx2, d2udy2, dvdx, dvdy, d2vdx2, d2vdy2


# Navier-Stokes equations (dimensionless form)
Re = (rho * u_avg * 2*h) / mu
navier_x = (u_int * dudx + v_int * dudy +
            dpdx/(rho*(u_avg**2)) - (d2udx2 + d2udy2)/Re)
navier_y = (u_int * dvdx + v_int * dvdy +
            dpdy/(rho*(u_avg**2)) - (d2vdx2 + d2vdy2)/Re)
continuity = dudx + dvdy
```

Listing 2 - The PDE computing and loss function with Navier-Stokes

```
Bootstrap: docker
From: python:3.10-slim

%files
    poiseuille_script.py /opt/poiseuille_script.py

%post
    pip install numpy
    pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
    pip install tqdm
    pip install matplotlib
    pip install random-fourier-features-pytorch

%environment
    export PATH=/usr/local/bin:$PATH
    export PYTHONUNBUFFERED=1

%runscript
    exec python /opt/poiseuille_script.py "$@"
```

Listing 3 - Container definition for apptainer

```
#!/bin/bash
#SBATCH --job-name=test_PINN    # create a short name for your job
#SBATCH --nodes=1               # node count
#SBATCH --ntasks=1              # total number of tasks across all nodes
#SBATCH --cpus-per-task=1       # cpu-cores per task (>1 if multi-threaded tas>
#SBATCH --mem-per-cpu=2G        # memory per cpu-core (4G per cpu-core is defa>
#SBATCH --time=12:00:00         # total run time limit (HH:MM:SS)
#SBATCH --gres=shard:24
#SBATCH --output=output.txt # Standard output file
#SBATCH --error=error.txt # Standard error file
#module purge

apptainer run --nv mycontainer.sif
```

Listing 4 - SLURM script to run on computing server