Chapter 4

Case Study 1: Vibrating String Between Two Pulleys

4.1 Introduction and Problem Setting

The 1^{st} use case that we examined was the ***vibrating string (fixed at both ends) with axial speed*** or just ***axially moving vibrating string*** case. As said in Chapter 1, the main reasons we studied this problem were: it is a good research problem to see the potential/limitations of PINNs and Flanders Make had a client that worked with this kind of systems, making it a real-world industrial problem/use case.

In such a system, we wave a string vibrating between two pulleys, i.e. although the string is fixed at both ends, it has horizontal speed. Think, for example, that there is a motor pulling the string horizontally at one end. A diagram of such a system is shown in Fig. 4.1. Systems like this appear in many real-world cases: production lines, inside machines, elevators, and many more.

With only this simple system, we could study many PINNs related topics such as:

- Can we retrieve the PDE of the system from the data? Same to say, can we infer the string properties (linear density and horizontal speed) from measurements?
- And from noisy data?
- How do the properties of the strings influence the result?
- Can we correctly predict the string's position in the future?
- How much data do we need? And does the data resolution influence the result?

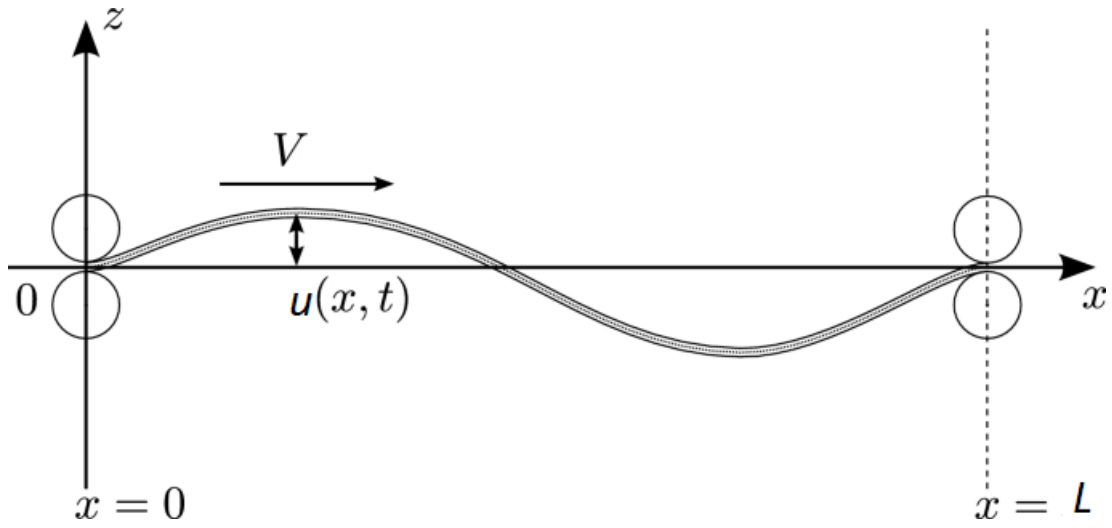


FIGURE 4.1: Diagram of an axially moving string between 2 pulleys. String has a horizontal velocity (v) and is fixed at both ends [30].

.

This system had the right complexity, not too complex (its physics are known and due to approximations can be considered linear) but at the same time not that simple. It is a slightly more complex system than the normal wave equation. And for that reason, we get to study a system that might occur in an industrial setting while maintaining the right level of complexity we were aiming. It was important to know beforehand the PDE of the system so that we could simulate data, so we couldn't choose a system whose PDE was completely unknown. The goal was to see if PINNs work on a simpler system before trying to tackle more complicated cases.

4.1.1 Physics of the System

A vibrating string, without horizontal speed, and fixed ends has the following PDE:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{4.1}$$

This is called the ***wave equation***. Here u, represents the y-position of the wire, also known as the ***displacement*** of the wire. We want to study the displacement of the wire as a function of time and the x-coordinate ($u(x,t)$), meaning we want to know how what is the y-coordinate at each point of the wire's length throughout a certain time window. The time variable is represented by t and the x-position by x. The variable c represents the ***wave speed***, and is given by:

$$c = \sqrt{T/\rho} \tag{4.2}$$

Where T is the applied tension to the wire and ρ is the linear density of the wire. The SI unit of each variable is the following:

- $[u] = m$
- $[t] = s$
- $[x] = m$
- $[c] = \left(\frac{[T]}{[\rho]}\right)^{\frac{1}{2}} = \left(\frac{kg \cdot m \cdot s^{-2}}{kg \cdot m^{-1}}\right)^{\frac{1}{2}} = m \cdot s^{-1}$

The string is fixed at both ends, which means $u(0,t) = u(L,t) = 0$; where L is the length of the wire. With these boundary conditions, the solution to the wave equation is straightforward and can be found knowing that it should be a superposition of a wave moving right $F(x+ct)$ and a wave moving left $G(x+ct)$ (or even knowing that the solution should be a wave given by $\Psi(x,t) = A\exp^{i(\pm kx \pm wt)}$ and insert this solution on the PDE). Nevertheless, the solution is a superposition of n vibration modes given by:

$$u(x,t) = \sum_{n=1}^{\infty} \left[\sin\left(\frac{n\pi x}{L}\right) \left[A_n \cos\left(\frac{n\pi ct}{L}\right) + B_n \sin\left(\frac{n\pi ct}{L}\right) \right] \right] \tag{4.3}$$

In our particular case, we also have a horizontal speed component v ($[v] = m \cdot s^{-1}$), which introduces some modifications to the wave equation. The PDE of the axially moving string is the following: [5]

$$\begin{aligned} &\frac{\partial^2 u}{\partial t^2} + 2v\frac{\partial^2 u}{\partial x \partial t} - (c^2 \pm v^2)\frac{\partial^2 u}{\partial x^2} = 0 \\ &\Leftrightarrow u_{tt} + 2v u_{xt} - (c^2 \pm v^2) u_{xx} = 0 \end{aligned} \tag{4.4}$$

The term $\pm v^2$ is positive if the speed is from right $\rightarrow$ left and negative otherwise. Similarly, as in the case of the pure wave equation (4.4), this equation has an analytical solution that can be obtained by knowing that the solution is a superposition of the same waves but with added horizontal speed: $F(x+(c+v)t)$ and $G(x+(c-v)t)$. The solution is the following [31]:

$$u(x,t) = \sum_{n=1}^{\infty} A_n \left[\sin\left(k_{bkwd,n}x + w_n t + \Psi_n\right) - \sin\left(k_{fwd,n}x - w_n t - \Psi_n\right) \right] \tag{4.5}$$

[5] Deriving both the PDE and its solution is outside the scope of this work. This paper [30] presents more than one way in which we can obtain both the axially moving string PDE and the solution to it.

Here, k and w_n were introduced to ease notation, but both are commonly used in wave equation solutions: k represents the wave number ($[k] = rad \cdot m^{-1}$), and w represents angular frequency ($[w] = rad \cdot s^{-1}$). The subscripts *fwd (forward)* and *bkwd (backward)* serve to ease notation and at the same time emphasise the fact that the solution is a sob-position of two waves: one travelling forward and one backwards. The phase Ψ_n was added to transform cosine into sine and simplify the expression. Each of these variables is defined as the following:

$$\begin{aligned} k_{reverse,n} &= \frac{n\pi L(c+v)}{c} \\ k_{forward,n} &= \frac{n\pi L(c-v)}{c} \\ w_n &= \frac{n\pi(c^2-v^2)}{Lc} \\ \Psi_n &= -\frac{2\pi(c+v)}{4c-\pi} \end{aligned} \tag{4.6}$$

Note that there are many possible ways to represent this expression (as shown in [31]): one can add phases to transform the sinusoidal functions, or express differently the argument of each sinusoidal function. By choosing an initial condition, $u(x,0)$, we can obtain the coefficients A_n. With c,v, and all the coefficients A_n, we have all the variables needed to fully generate the axially moving vibrating string using eq. 4.5.

It is important to note that both the wave equation and the vibrating string with axial speed are derived using some assumptions: the wire has linear density, the speed (v) is completely horizontal, we assume small strain and displacement, and there is no air resistance or heat dissipation. Of course, this is not exactly what happens in real-world scenarios, but it is a *good enough approximation*. With the analytical solution (4.5) to our problem, we can generate data to train our PINNs.

Finally, the PDE (4.4) can be numerically solved using a finite-difference scheme, just as the wave equation.

Simulated Data

By choosing some horizontal speed v, wave speed $c = \frac{T}{\rho}$, and initial condition (which is reflected in the coefficients A_n) we can generate solutions to the vibrating string with horizontal speed. Furthermore, we defined $L = 1$, for all cases, to further simplify things and remove one variable. Here are some examples of simulated axially moving strings:

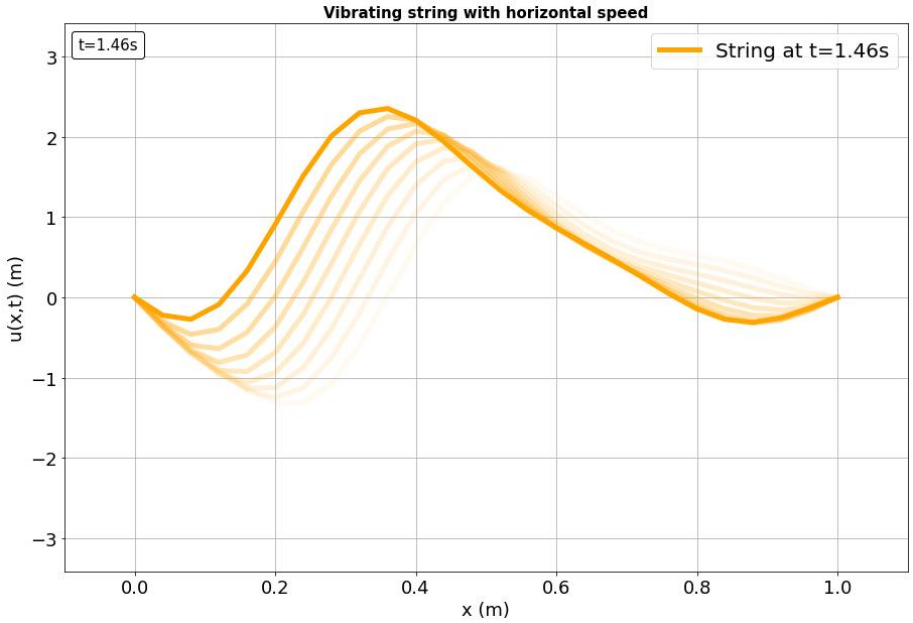


Figure 4.2: Example of an axially moving string with $c = 1ms^{-1}$, $v = 0.5ms^{-1}$,$dx = 0.04m$,$dt = 0.01s$ and 5 modes. Faded lines show previous time-steps position of the wire

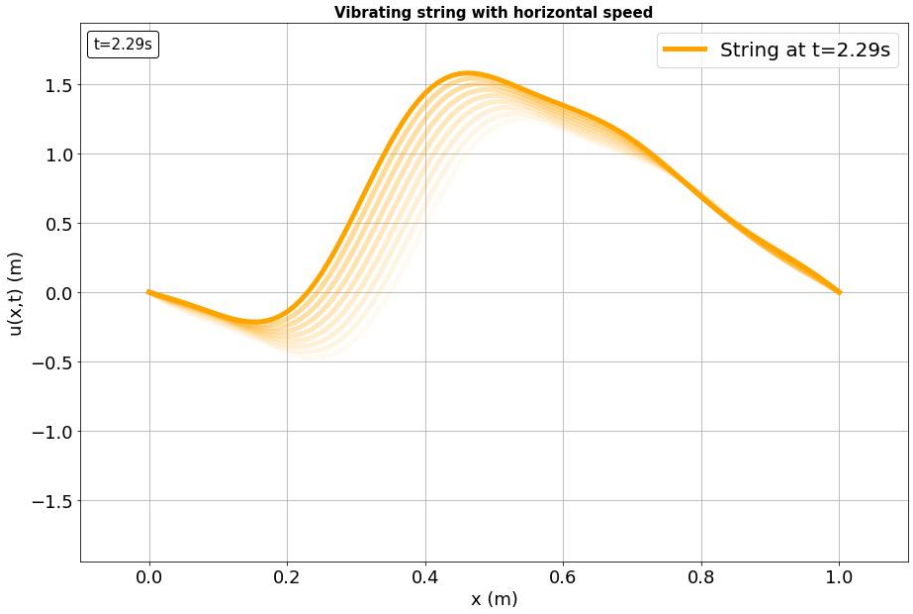


Figure 4.3: Example of an axially moving string with $c = 3ms^{-1}$, $v = 1ms^{-2}$,$dx = 0.005m$,$dt = 0.01s$ and 6 modes. Faded lines show previous time-steps position of the wire.

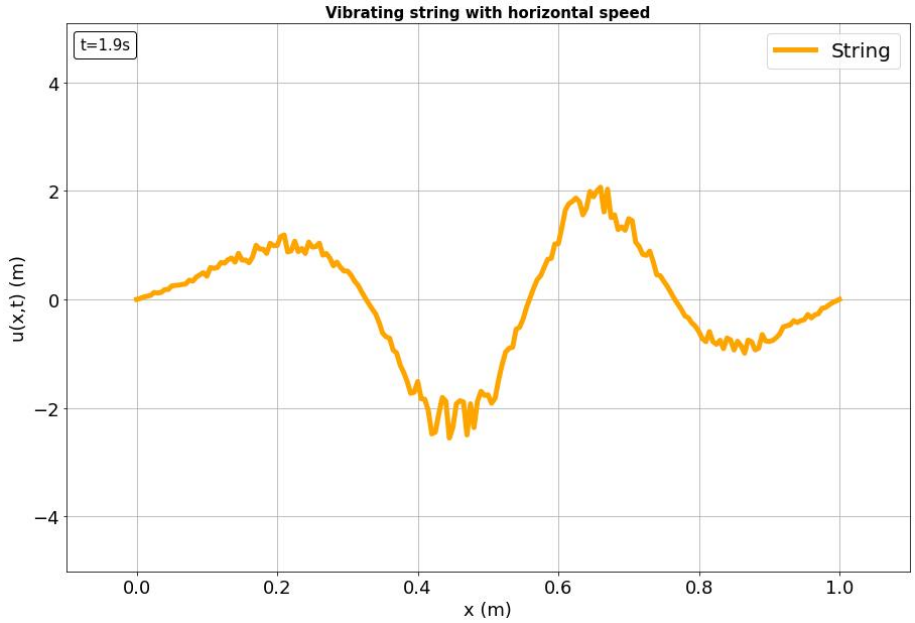


FIGURE 4.4: Example of a string with $c = 1ms^{-1}$, $v = 0.5ms^{-1}$, 3 modes, and 10% added noise.

Animated GIFs showing the behaviour of some string configurations are in the GitHub repository under the folder *String/GIFs*. In the GIFs, it is easier to notice the horizontal speed of the string, something that the faded lines intend to show.

The notebook ***String Data Generation.ipynb*** has the functions to generate data and save it as a GIF, image, and save the arrays in a file. We mainly worked with initial conditions being waves with constant random amplitude, but other initial conditions could have been used, e.g. a triangular wave, a rectangular wave, and much more. There is also the option to add Gaussian noise to the data, as to simulate real-world measurements.

Gaussian noise is defined as signal noise given by the Normal/Gaussian distribution $\mathcal{N}(\mu, \sigma)$ (where μ is the mean and σ the standard deviation). We decided to add Gaussian noise proportional to each "original" string function, thus we get a ***noise level***. Instead of adding an absolute value of noise, we add noise as a level/percentage of the original data. Formally, considering η as the noise level, we added noise by doing the following:

$$\begin{aligned} \sigma_{noise} &= \eta u_{original} \\ noise &= \sigma_{noise} \times \mathcal{N}(0,1) \\ u_{noisy} &= u_{original} + noise \end{aligned} \tag{4.7}$$

4.1.2 Goals and Baselines

We divide our goals into two main objectives: determine the underlying PDE coefficients and predict the wire's future behaviour. We wanted to simulate a real-world scenario where we did not know the string properties (c and v), so we first wanted to determine them, then use them to build the PDE and finally use the PDE to physically constrain the training process of a neural network.

Determining the Underlying Physics of the System (Inverse Problem)

As we have seen in the last chapter (3), PINNs can be used to solve inverse problems, i.e. problems in which we know some of the physics but not all. In this case, we know the general form of the PDE, but it could be the case that we do not know the horizontal speed v or even the linear density of the wire $\rho = T/c^2$. By using PINNs we can determine each PDE term coefficient. Let us recall how we use PINNs to solve inverse problems. First, we write:

$$u_{tt} = \Theta\xi \tag{4.8}$$

Where Θ is a matrix containing candidate functions and ξ is a vector containing each candidate scaling factor. In our case, by rewriting 4.4 as:

$$u_{tt} = \xi_1 u_{xx} - \xi_2 u_{xt} \tag{4.9}$$

And thus we get the following Θ:

$$\Theta = \begin{bmatrix} u_{xx}(\{x,t\})_0 & u_{xt}(\{x,t\})_0 \\ u_{xx}(\{x,t\})_1 & u_{xt}(\{x,t\})_1 \\ \vdots & \vdots \\ u_{xx}(\{x,t\})_N & u_{xt}(\{x,t\})_N \end{bmatrix} \tag{4.10}$$

In the end, if the vector ξ is correctly determined, we should obtain $\xi_1 = (c^2 \pm v^2)$ and $\xi_2 = 2v$. That is, we have determined v and c, which were the only physical unknowns, and thus determined the underlying physics of the system. Note that eq. $u_{tt} = \Theta\xi$ can be solved, for ξ, using, for example, the least squares method in matrix form.

In this case, we set the baseline as predicting c and v, with less than 3% of error. We generated solutions to the PDE with increasing error in c and v and then compare

them to the solutions without error. We thought that for coefficients with error in c and v greater than 3% the solution to the PDE became too different from the true solution. As such, we set the upper limit of acceptable error as being 3%. In 4.4, we show some examples of comparisons between true solutions and the solution with error.

Predicting the Wire's behaviour in the Future (Forward Problem)

Now, assuming that c and v are known, could we, given some data of the past positions of the wire, predict its position in the future? This is a textbook case where PINNs should perform better than a simple Neural Network. Recall that we can leverage the fact that we have physics knowledge and can compute $\mathscr{L}_{PDE}$, and thus make our network converge to physics adhering solutions. The goal is to train a PINN on the data that ranges from, $[0, t_1]$ which we will call *first domain* and denote by $\mathscr{D}_1$. After that, we will try to extrapolate on the *second domain* $\mathscr{D}_2 = [t_1, t_2]$. Fig. 4.5 shows the separation between these 2 domains. To help visualisation, imagine we had an experimental setting where it is possible to measure the wire's position over time with good resolution. But then, when the wire is used in a production setting, for whatever reason, you cannot measure the position of the wire, or you can only use low-resolution sensors. Our goal is to be able to accurately predict/determine the wire's position in the second domain, where we don't have any information about the wire position, other than at the fixed ends. Given the fact that we were capable to gather some data on the 1^{st} domain, we use it to learn the PDE of this particular wire and at the same time train a PINN on this data. Then with the trained PINN we predict the wire's position on the 2^{nd} domain.

Finally, in the forward problem, we defined two baselines: **a finite differences solution to the PDE** and **regular neural network**. Which advantages and disadvantages PINNs offer when compared to each of them?

4.1.3 Research Questions

This problem might seem simple and very well-defined, but there are many variables that could influence the end result. Here are some:

1. what is the minimum resolution (spatial and temporal) needed on the data in order to get acceptable results?

2. Can we emulate a real-world data acquisition system and access its performance?

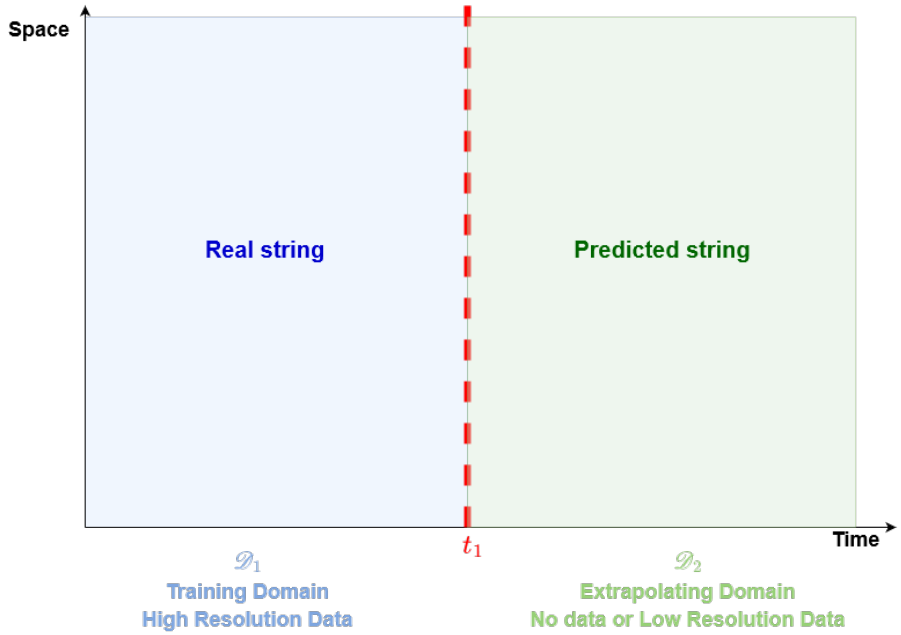


Figure 4.5: Diagram of the training and extrapolating domains.

3. What is the maximum noise level that the data can have? How long does training a PINN take?

4. What is the best training scheme? Does the number of nodes, horizontal speed, or even linear density of the wire influence the results?

5. How far can we predict the wire's behaviour? If we consider non-linear behaviour (like dissipation), do PINNs still work?

All these questions are extremely important in real-world applications, and they help define the limits of our approach and where to and not to use it. We tried to study all of these cases, but we did not have time to go into all of them; *especially, we did not study how do PINNs behave with extra non-linear factors.*

4.2 General Framework

In the end, we arrived at the following training scheme, which we use to solve both the inverse problem and the forward problem:

1. Train a regular neural network (NN_1) until convergence is achieved in the auto differentiation derivatives.

2. Train a PINN ($PINN_1$) to get the PDE coefficients (inverse problem).

3. Train a regular PINN ($PINN_2$) with known coefficients and try to extrapolate to never seen times (forward problem).

To get to this point, we went through many different schemes and this was an iterative and empirical process. I think it is important to highlight that much of PINNs literature only focus on end results; that is, usually the training scheme is not explained or even disclosed. It is hard to find a paper explaining what was done and why. So, to get to a point where the results were acceptable, we had to try many approaches. I will describe some and try to understand why they failed in later sections.

4.2.1 Train a (NN) until Convergence

We decided to start first by training a regular neural network (a neural network with only loss on the data) (NN_1) because we saw that the *derivatives, being computed using auto-diff, when training was still starting were completely off the real values*. We would then use these wrong derivatives to compute $\mathscr{L}_{PDE}$, resulting in the network being optimised to decrease something that was wrong. This would make training completely fail. We only understood that this was happening after we started to compare auto-differentiation derivatives with the analytical derivatives as training took place.

By giving time to the network to decrease the loss on the data, we ensure that the derivatives computed using auto-differentiation would be correct. Only when the network has ”learnt” the function of the data has it ”learnt” its derivative. To be sure that the derivatives were correct, we compared them, as training took place, to the analytical ones and stopped training when the error was sufficiently low (5-10%). Only then we introduced the PDE loss, thus physically constraining the neural network.

4.2.2 Train a PINN to determine the PDE coefficients.

Since we now know that the derivatives are being computed correctly, we introduce $\mathscr{L}_{PDE}$ to NN_1 and solve eq. 4.8 for ξ using the least squares method in matrix form. The value of each entry of ξ will quickly converge to a value, as we will see later, thus we obtain the complete PDE of the system, determining the unknowns c and v.

4.2.3 Train a PINN with known coefficients then extrapolate

Now, having the correct PDE we once again add $\mathscr{L}_{PDE}$ to NN_1, using eqs. 3.4, 3.2, and 3.3, and thus start training a PINN ($PINN_2$). The reason we start with an already "pre-trained" neural network (NN_1) is to get accurate (automatic differentiation) derivatives from the start. This way we are certainly constricting the neural network with the right physics (with the right derivative terms).

In this last phase, the validation set is the extrapolation domain, so time steps that the model hasn't "seen" before. So, we train on $T_{train} = [0, t_1]$ and evaluate on $T_{eval} = T_{extrapolate} = [t_1, t_2]$. Recalling that to compute $\mathscr{L}_{PDE}$ (3.2) we do not need to know the real values, only the derivatives computed using auto-differentiation. This allows us to compute $\mathscr{L}_{PDE}$ in the validation set, without knowing any targets, which makes extrapolation much easier. This is the *data augmentation* power of PINNs, since we, in fact, gain points inside the extrapolation domain where we can train our neural network. Although we do not know the targets inside the extrapolation domain, we can still "train" our neural network only on the PDE loss, thus achieving much better results than a regular neural network.

4.3 Details on Architecture, Activation Function, and others

In this section, we will detail the architectures, activation functions, some hyperparameters that were used, what were the most important parameters, and what range of values they usually took. As said before, this was the part that took the most time, since searching for good configurations is done empirically. Furthermore, what worked for a given string configuration (speed, density, number of modes, and noise) most of the time did not work for all the other configurations. Therefore, every time we changed string configuration, we had to find the right parameters again. Given all this, we preferred to present hyperparameter values in intervals rather than give a definitive answer. In the annex, there will be more concrete results for some parameters.

Architectures

Recall that we mainly used feed-forward neural networks. Therefore, the only parameters that changed in terms of architecture were: *depth* (number of hidden layers) and *width* (number of neurons per layer). For example, if a neural network is said to have 3

depth and 40 width it has an input x followed by 3 layer of 40 neurons each and then an output layer. This network is noted by: $(\boldsymbol{x})$-40-40-40-$(\hat{y})$.

Table 4.1 shows the typical depth and width used in this work. We found no significant differences in performance inside this interval.

Depth	Width
[3,5]	[20,70]

TABLE 4.1: Architecture parameters

Activation Functions

Recalling, from 2, that the activation function used should be n-differentiable, where n is the largest-order derivative term present in the PDE with which we were working. Because there are cases where we do not even know which concrete PDE we are studding, using an infinitely differentiable function solves all our problems. That is why we started by using tanh as activation function.

However, we had problems fitting our data while using tanh activations. The authors in [32] show that this happens because *"the difficulty in training PINNs stems from the model's strong initial bias towards flat outputs"*. This was exactly what we were observing, the predicted waveform of the wire would just collapse to a flat line after a few training steps. In the same paper, they propose changing the activation of the first layer to:

$$\sigma = \sin(2\pi \boldsymbol{z}) \tag{4.11}$$

Where $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, and thus the output of the first layer becomes:

$$thus, h(x) = \sin(2\pi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})) \tag{4.12}$$

By doing this, we accomplish two things:

1. *Embed Physically Relevant knowledge into PINNs* - We know that the solution to the wave function should be a sum of sinusoidal components; why not use that knowledge and try to restrict the possible solution space? This is what we are doing by having a sin activation function in the first layer: we transform all inputs into a sinusoidal space, and thus make all outputs of the neural network be sinusoidal functions as well.

2. Escape local minima - In [32], it is shown that having this sin transformation allows us to escape the flat output minimum.

We changed our approach and introduced 4.11 as activation of the 1st layer. Instantly, we started to achieve better results: faster loss minimisation and overall lower validation loss. Note that *all other activations stayed as* tanh (excluding the output layer, which is just a linear weighted sum without activation).

Weight Regularisation

As discussed in Chapter 2, regularising weight helps prevent overfitting and thus improves generalisation on never seen times. We used default values of λ in eq. 2.20, which are $\lambda \in [10^{-8}, 10^{-1}]$.

The major problem we faced was overfitting, so adding regularisation helped achieve better results when we wanted to predict the wire's behaviour in never seen times.

Balancing the Sub-Losses

How we weight both sub-losses in eq. 3.4, in particular how we chose, w_{PDE} is one, if not the most, important decisions when training a PINN. Note that it is advised to keep $w_{MSE} = 1$ and only adjust w_{PDE}, thus having one less parameter. Choose a "big" w_{PDE} and your network will not fit the data, and a "low" w_{PDE} and your network will not generalise. But the window between big and small is very short most of the time, and hard to find. There is no consensus on how to define w_{PDE}, even if it should be a dynamic parameter or a fixed one. We tried different strategies:

1. *Static Weighting* - Empirically choose a value, and it stays the same during all training. You could also compute both losses (without weighting) and use the ratio as w_{PDE}, this way we guarantee that when training starts, both sub-losses should be of the same magnitude after weighted.

2. *Dynamic Weighting* - Here we update the weights every training step. Making sure that the sub-losses are always of the same order. This seems to be the obvious answer, but convergence is not ideal most of the time, making the training process harder. Furthermore, some schemes are computationally intensive and not straightforward to implement. Some examples of dynamic balancing schemes are given in [22].

Most of the time we settled on the 1^{st} strategy, because it was simpler and the results did not differ greatly when comparing one to another. However, the second strategy seems more promising, although more complex.

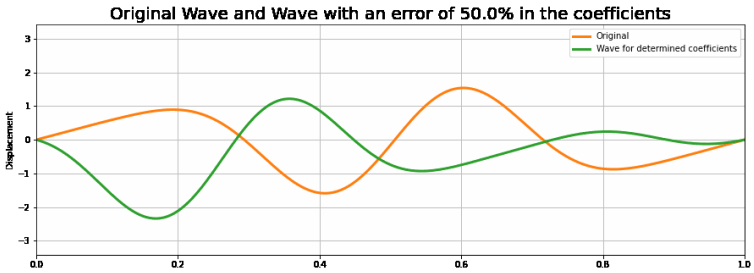
4.4 Results and Discussion

In this section, the results obtained plus explanation, comments, and discussion are presented.

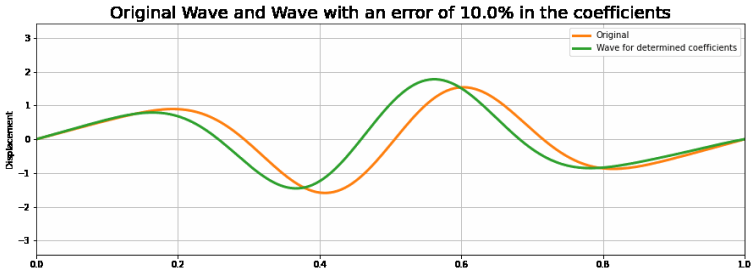
4.4.1 Obtaining the PDE Coefficients (Inverse Problem)

We managed to obtain the coefficients with an error up to 3% for a normal configuration (which we will see what is). Obviously, it varied between the string and hyperparameter configurations, noise level and data resolution. For a typical configuration, we had an error of 3%, and we could get around 1% error for ideal configurations. In the next sections, we will show how each variable influences the error on the determined coefficients. We studied the influence of each parameter by keeping all other variables constant.

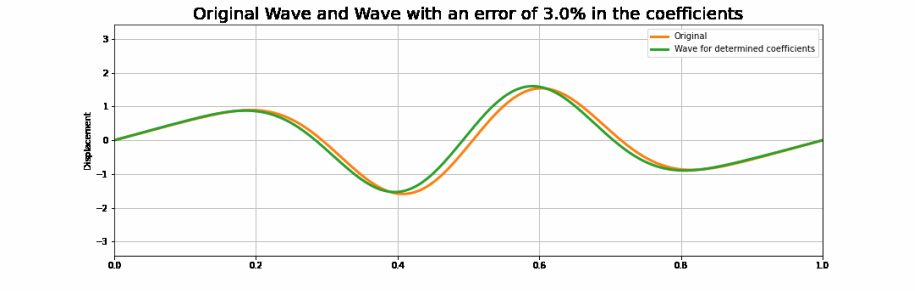
To get a clearer idea of how the coefficients of the PDE give way to different solutions, we have simulated data given by the same PDE with slightly different coefficient values. All GIFs are in the folder *String/GIFs/Coefficients*. We built this data by multiplying the original coefficients by $(1 - error)$, and thus get the new PDE coefficients. Here are some examples of frames from those GIFs:



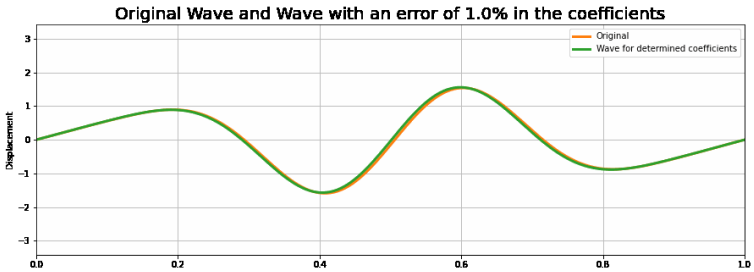
(A) Comparison of original wave and wave with 50% error on PDE coefficients



(B) Comparison of original wave and wave with 10% error on PDE coefficients



(C) Comparison of original wave and wave with 3% error on PDE coefficients



(D) Comparison of original wave and wave with 1% error on PDE coefficients

Figure 4.6: Frames for the same time step for different errors in the PDE coefficients.

It is clear that above 10% error in the coefficients we get a very different wave form. We aimed to have around 1-3% of error under optimal conditions. What were the conditions needed to achieve errors under 3%? Note that, when we say an error of, for example, 3%, this means that the determined PDE was:

$$u_{tt} + (1 \pm 0.03) \times 2v u_{xt} - (1 \pm 0.03)(c^2 \pm v^2) u_{xx} = 0 \tag{4.13}$$

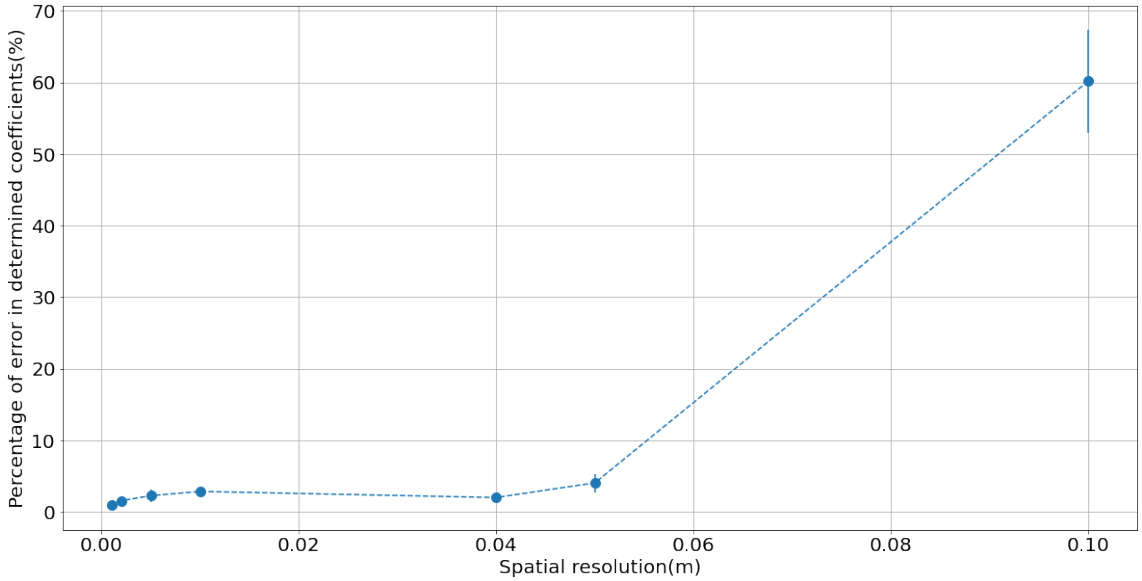
We now present the results of the influence of each parameter on the error of the determined PDE coefficients.

4.4.1.1 Data Resolution

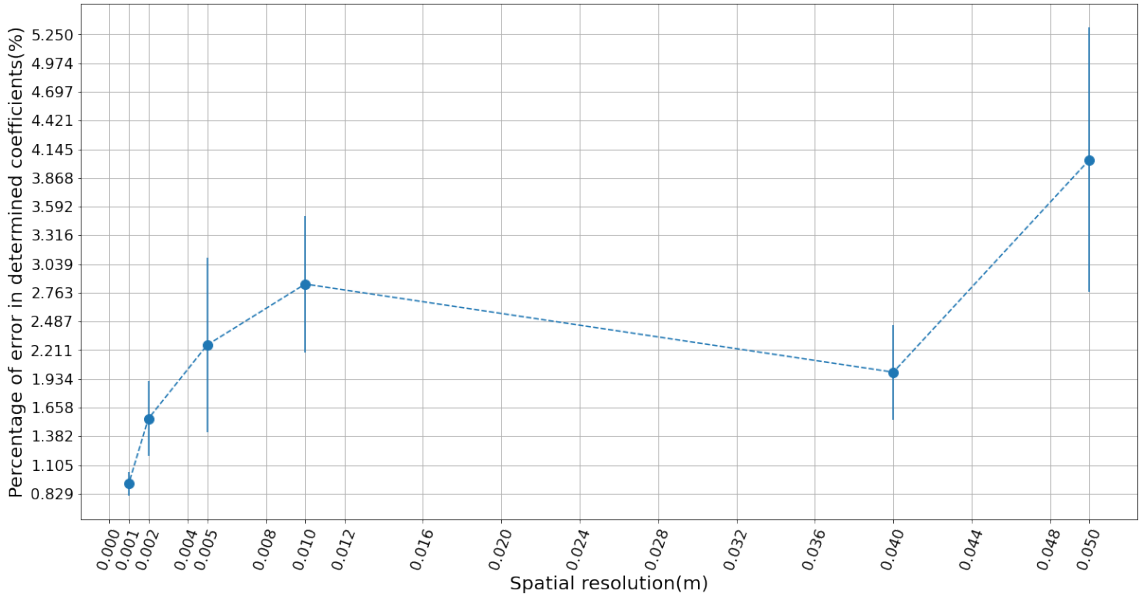
We started by studying the impact of resolution in the determined coefficients. We considered temporal resolutions dt of $dt \in [0.1, 0.001]s$. Normal cameras record at 24 frames per second (fps), which results in a resolution of $dt = \frac{1}{24} = 0.04s$, and high-resolution cameras record at 100 fps, that is $dt = 0.001s$. Our interval encompasses both normal and high (time) resolution cameras. When it comes to spatial resolution dx, we used an interval of $dx \in [10^{-1}, 10^{-3}]m$, which (considering that for sensing cameras the object space resolution ranges from about some millimetres to some microns) falls into the acceptable interval. We wanted to use parameters that stood as close to real-world ones, that is why we used these intervals.

It is important to note that we did not use spatial resolutions lower than $1mm$ because it would make training times very long, instead of training runs taking some hours, it would perhaps take some days. All string properties (c,v, noise level and initial condition) and neural network properties are described in Appendix A.

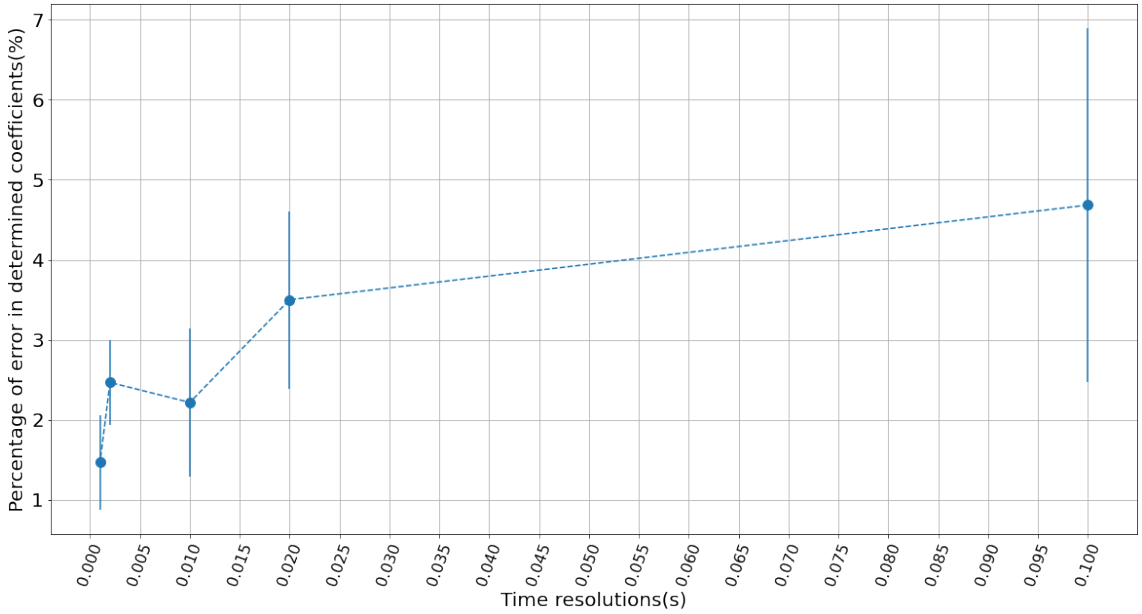
The results are shown in Fig. 4.7 and Table A.3. When varying dx, the time resolution was fixed at $dt = 0.01$, and when varying dt the spatial resolution was fixed at $dx = 0.005$. Both these values were in the middle of their respective intervals. Error bars show variance throughout different runs. Usually we did 5 runs for every configuration, but for higher resolutions ($dx = 0.001/2$ and $dt = 0.001$), given it could take 1 day to train a model, we only trained a model twice.



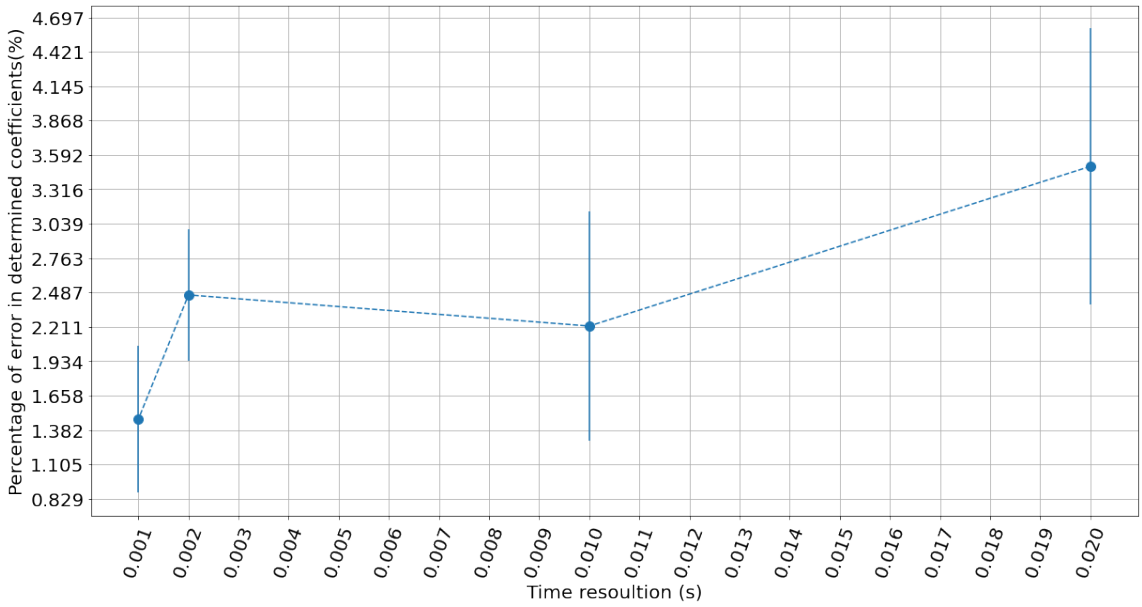
(A) Error on the coefficients as a function of spatial resolution.



(B) Error on the coefficients as a function of spatial resolution - zoomed.



(C) Error on the coefficients as a function of temporal resolution.



(D) Error on the coefficients as a function of temporal resolution - zoomed.

FIGURE 4.7: Results on the study of the effect of spatial/time resolution on the overall error on the determined coefficients.

It is clear that as (spatial and temporal) resolution increases, the percentage of error in the determined coefficients decreases.[6] In other words, higher resolution means better results, which was expected. Starting from $dt = 0.01s$ and $dt = 0.040m$ the error is less than 3%, which was our threshold. But at higher resolution values, for example for $dx = 0.001m$ (and dt fixed at $0.01s$) the error is less than 1%. As we suspected the results greatly depend on the resolution of the data, from the worse result ($dx = 0.1m$) to the

best result ($dx = 0.001m$) there is a difference of 59% of error in determined coefficients, which is a considerable disparity.

However, bear in mind that the number of training points increases proportionately with resolution, which means more training time and more computational resources. Therefore, to get a lower percentage of error, one needs to spend more time training. To get an idea, Fig. 4.8 shows a comparison between training runs, of 15,000 training steps, for different resolutions. From $dx = 0.04$ to $dx = 0.002$ there is a 20-fold increase in the number of training points. This difference in training points, in the hardware we used, translated into a difference of about 5 hours to reach 15k epochs, as we can see in Fig. 4.8. However, to obtain an error of 3% (the red dashed line), the difference time between the three resolutions was: $dx = 0.04 \rightarrow 48$ min $\rightarrow dx = 0.05 \rightarrow 48$ min $\rightarrow dx = 0.002$. In total, this represents a 96-minute difference between $dx = 0.04$ and $dx = 0.005$ training times. However, only the highest resolutions are capable of getting errors below $1/2\%$.

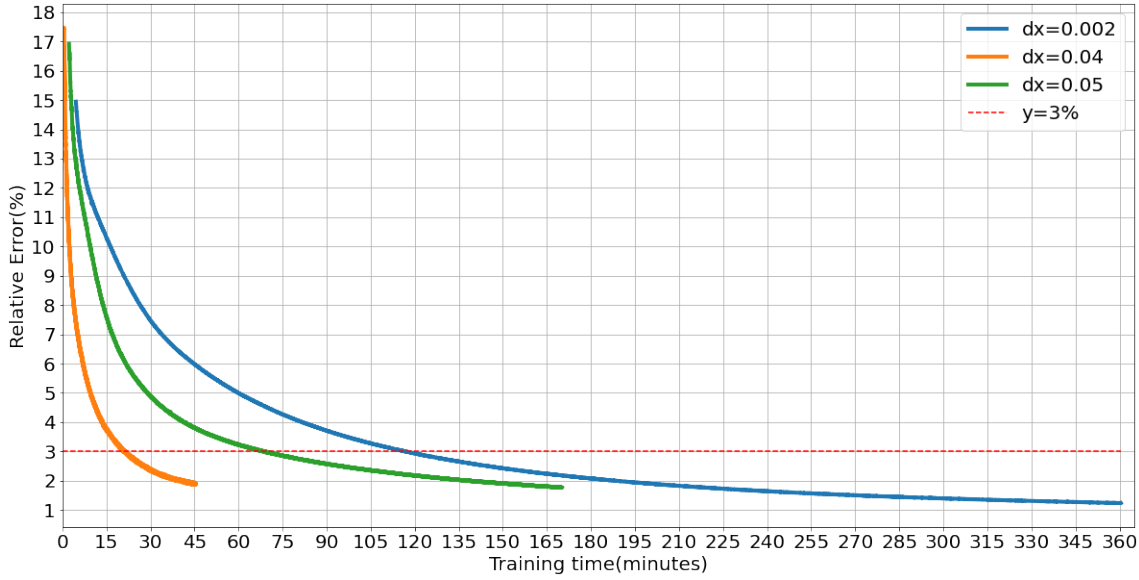


Figure 4.8: Error on the coefficients as a function of training time.

So, we should consider having lower resolution if we can afford to have slightly worse coefficients: for example, having around $dx = 0.01s$ and $dx = 0.04m$ should result in an error of about 3% while still having acceptable training times ($\sim$ 45 minutes). In contrast, if our application needs higher accuracy in the determined coefficients, we should consider having higher resolutions to the detriment of training duration. We are going to see, in the next section, what difference the coefficients make when we are extrapolating to never seen times.

[6] Higher resolution means smaller interval between points, hence smaller dt and dx.

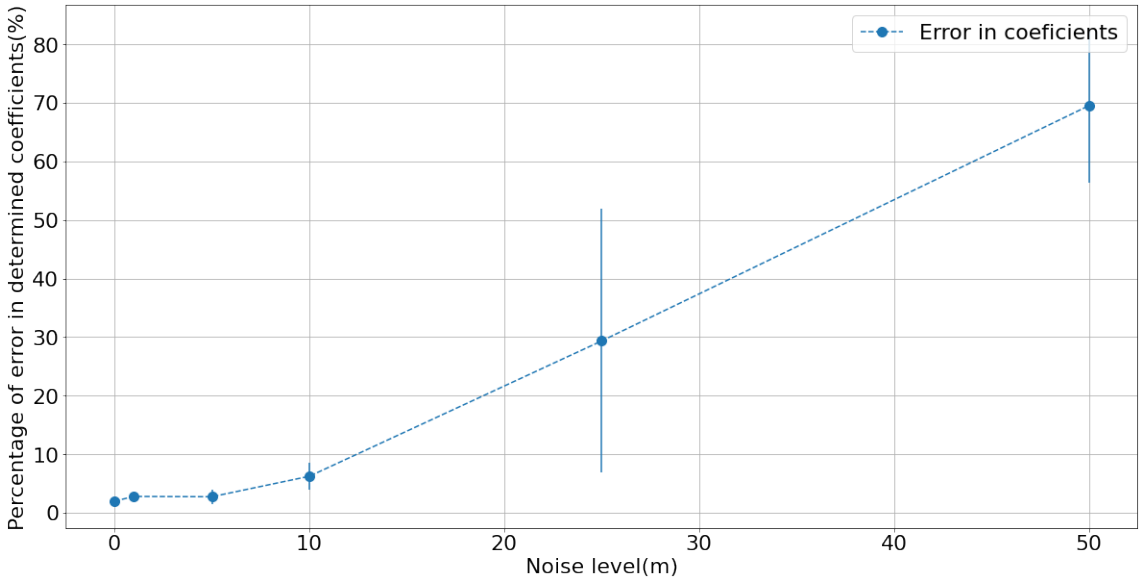
The main takeaway is that spatial resolutions lower than $0.04m$ and temporal resolutions equal to and lower than $0.01s$ should be used to get to the region of errors smaller than 3%. Using these two resolution values together gave an average error of $(2.0 \pm 0.5)\%$ of error.

A small note on the fact that the error as a function of spatial resolution (4.16a and 4.16b) has a point ($dx = 0.04m$) that breaks the trend of lower resolution corresponding to lower error. We can see that the error bar associated with this point is larger than that of the rest. Therefore, we can attribute this result to uncertainty.

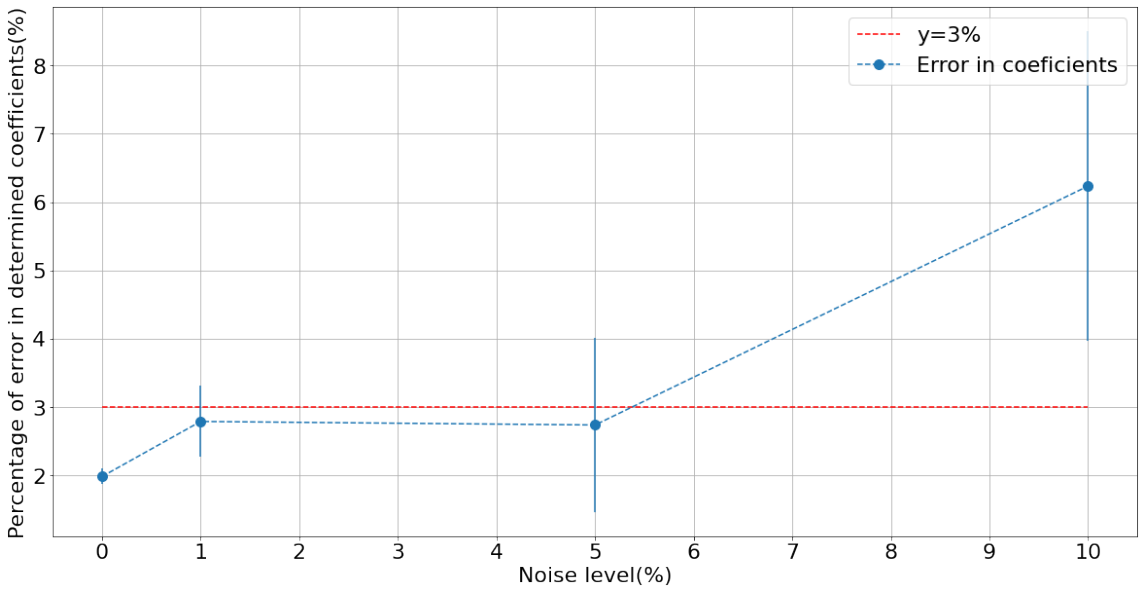
4.4.1.2 Noise Level

To study the impact of noise (present in the data) on the error of the determined coefficients, we decided to use $dx = 0.01s$ and $dx = 0.005m$, these values were a good compromise between training time and resolution-induced coefficient mismatch, as we have seen just before. Gaussian noise was added to obtain the noisy data. Once again, the properties of the string used are shown in A.

Fig. 4.9 shows the overall results. Here, we can see that with 25% of noise, the error becomes around 60%, which is completely off the 3% baseline we established. For 25%, we get an error of 20%, which is still bad. For the upper limit we set of 3-5% error, the maximum amount of noise is around 10%. These are exactly the results obtained in [33], more than 10% noise, and the error on the coefficients starts to get unreasonably large.



(A) Error on the coefficients as a function of noise percentage.



(B) Error on the coefficients as a function of noise percentage - zoomed.

Figure 4.9: Results on the study of the effect of noise on the overall error on the determined coefficients

These results allow us to say that ***PINNs successfully obtain the coefficients of the PDE from data with*** $noise < 1\%$ ***up to*** $2.7 \pm 0.4\%$ ***error***. However, if the noise level increases to 10%, the error increases and increases to around 6%, which, as we have seen, is high.

4.4.1.3 Horizontal and Wave Speeds

To better understand the role of the string parameters (density, applied tension, and horizontal/axial speed) we studied the error of the determined coefficients as a function of wave speed (c) and axial speed (v). Recall that, $c = \sqrt{\frac{lineardensity}{Tension}} = \sqrt{\rho/T}$. Knowing that, for example, the (non-linear!) density of steel is $7.85g \cdot cm^{-3}$, and if we consider the wire to be a cylinder of $1cm$ radius and $1m$ long, we get a linear density of $\rho = \frac{mass}{length} = \pi \times 0.785kg \cdot m^{-1}$. If, for example, we have a $1kg$ free weight pulling the string, resulting in $T = 10N$, we get the following wave speed: $c = \sqrt{\frac{10}{0.785}} \approx 2ms^{-1}$. For this reason, we decided to use $c \in [0.25, 2]ms^{-1}$. Furthermore, we assumed v should not differ much ,and always be slightly lower than c; as such we set $v \in [0.01, 1]$. This way we could get an idea if different c and v meant greater/smaller coefficients errors.

The results are shown in Fig. 4.10, Fig. 4.11 and Table A.5 in Appendix B. To obtain the second figure (4.11) we simply represented all errors above 5% with the same colour, making it possible to better distinguish lower errors. Note that for $c = v$: $c^2 - v^2 = 0$, and thus one of the coefficients of the PDE (4.4) is equal to 0. As such, all corresponding squares where c is equal to v are blank.

We can see that for both extreme values of v the error increases significantly, the same thing happens for the lowest value of c, 0.25. There is also a clear zone where the error is lower than the rest, for $c \in [0.75, 1.25]$ and for $v > 0.01$, where the error is smaller. And we think this is not a problem because we mainly used values on that range to optimise the hyperparameters of our neural networks (learning rate, weight decay, depth, width, activations, and more), and then used them to get these results. If we used other range of values, say higher values like $c > 2$, and optimised the training hyperparameters for those values, we could, almost certainly, achieve the same low errors as seen here. Nevertheless, we mainly used $c = 1$ and $v = 0.5$ to obtain all other results shown in this work to be sure we get reasonably stable/good results.

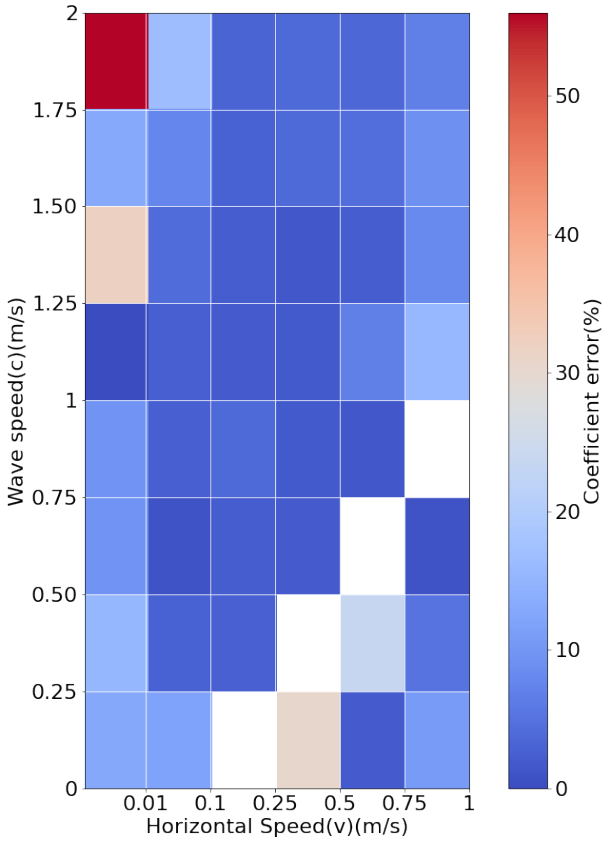


Figure 4.10: Coefficient error(%) as function of wave speed(c) and axial speed(v)

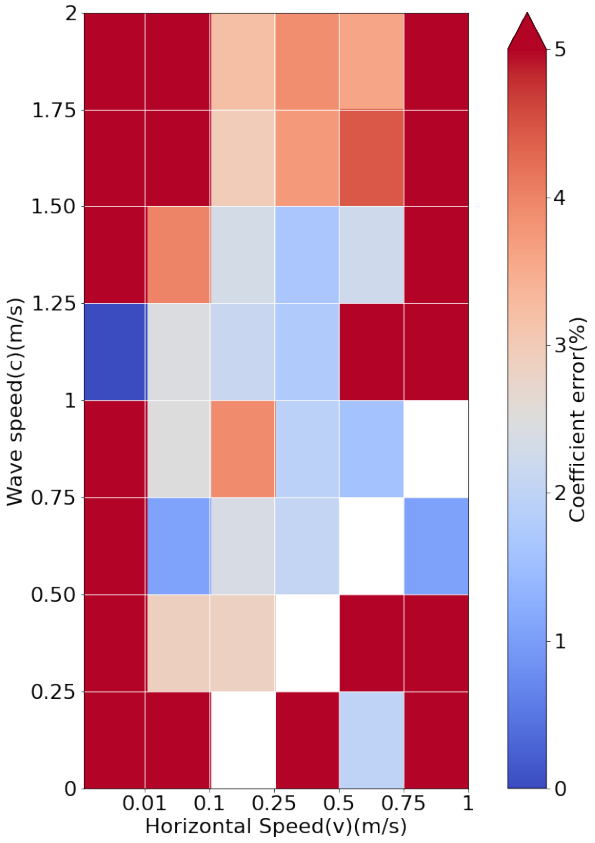


FIGURE 4.11: Coefficient error(%), thresholded at 5%, as function of wave speed(c) and axial speed(v)

4.4.2 Extrapolating (Forward Problem)

After determining the PDE, we can begin to physically constrain the neural network by adding the loss on the PDE to the total loss, as in 3.4. Note that we used the same coefficients obtained before to train these new PINNs, making our results depend on the previously obtained coefficients for each parameter configuration.

Recall that we only have training data (targets) in the first domain ($\mathscr{D}_1$), also called the training domain, and try to predict the position of the wire in the second domain ($\mathscr{D}_2$) where we do not have training data, also called the test domain. It is important to note that we know the string ends are fixed, meaning that $u(0,t) = u(L,t) = 0$. As we always know the position of the wire at the ends, we can use these two points as training

points as well. All neural network parameters and string properties are in A.2. All the GIFs of the results can be seen in *String/GIFs/Extrapolate* at the GitHub repository.

An example of an extrapolating prediction of a PINN for data with $dx = 0.005m$, $dt = 0.01s$ plus added 5% noise is shown in Fig. 4.12. In Fig. 4.13, we plot the absolute difference between the predicted wave and the true wave for the second domain.

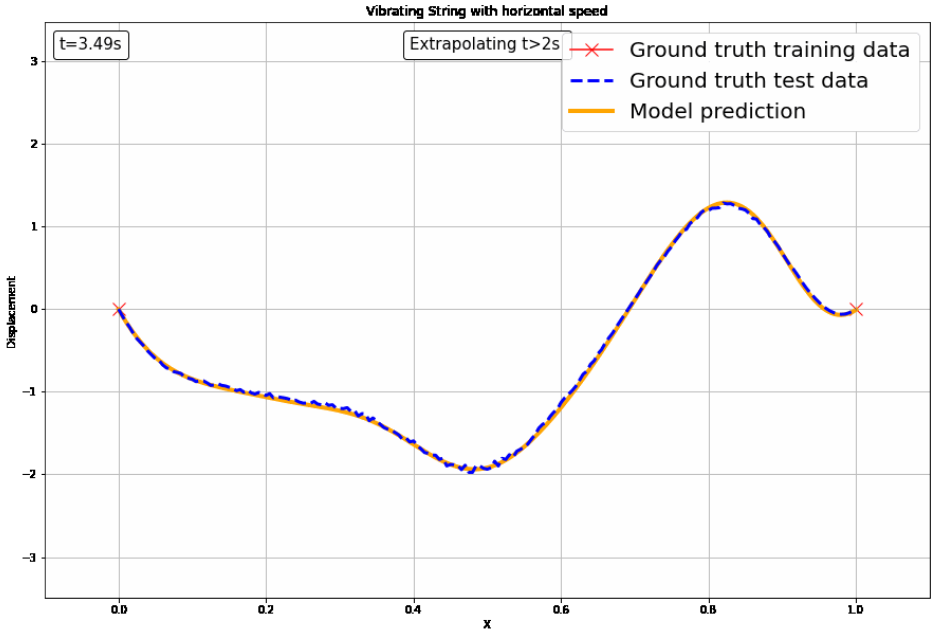


Figure 4.12: PINN predictions vs ground truth for $t = 3.49s$,data with $dx = 0.005m$, $dt = 0.01s$ plus added 5% noise.

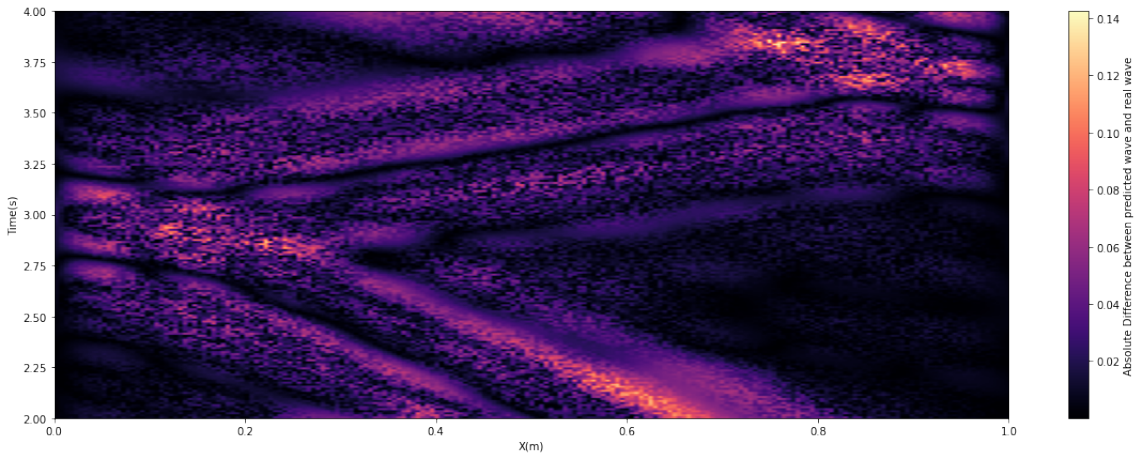


Figure 4.13: Absolute difference between PINN predictions and truth, on the second domain, for $t = 3.49s$,data with $dx = 0.005m$, $dt = 0.01s$ plus added 5% noise. X represents the horizontal position of the wire in meters.

For comparison, Fig. 4.14 and Fig. 4.19, are similar plots but for a regular feed-forward neural network (with the same parameters as the PINN before), with even higher resolution data and no added noise (optimal data conditions). We can clearly see the downgrade compared to the previous PINN. The regular neural network is only able to extrapolate near the fixed ends because those points are also training points; otherwise it cannot extrapolate in the second domain, as we can see by the error it is making in the middle of the length of the wire. This is the first time we clearly saw the improved performance of PINNs when compared to regular neural networks.

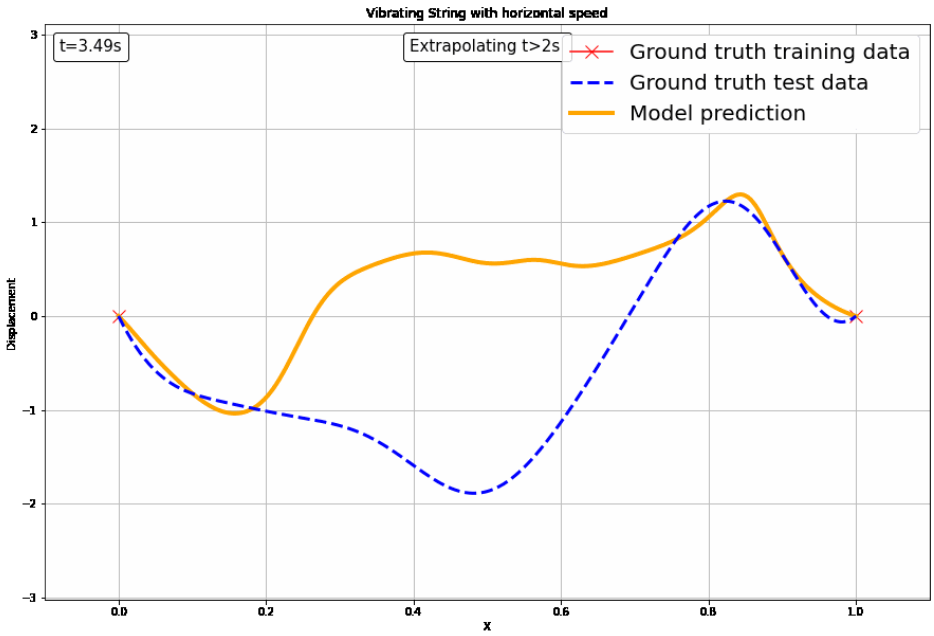


FIGURE 4.14: Regular neural network vs ground truth for $t = 3.49s$, data with $t = 3.49s$,data with $dx = 0.002m$, $dt = 0.01s$.

Obviously, once we remove the training data $\mathscr{D}_1$ and focus only on obtaining a low error on the PDE in$\mathscr{D}_2$ domain, the model will start to perform badly in $\mathscr{D}_1$ while decreasing the error in $\mathscr{D}_2$. In the GIFs one can clearly see this behaviour, when $t \geq 2$ the fit improves drastically, while for $t < 2$ the model just is not fitting the data (as we expected) because it is not trained to do so.

We measured the results by computing the MSE in $\mathscr{D}_2$, or, in other words, by computing the loss on the validation set. Again, as before, we cannot give a definitive result as the end result, since it depends on many factors. As with the coefficients, it was important to understand the effect of parameters such as resolution, noise, and string

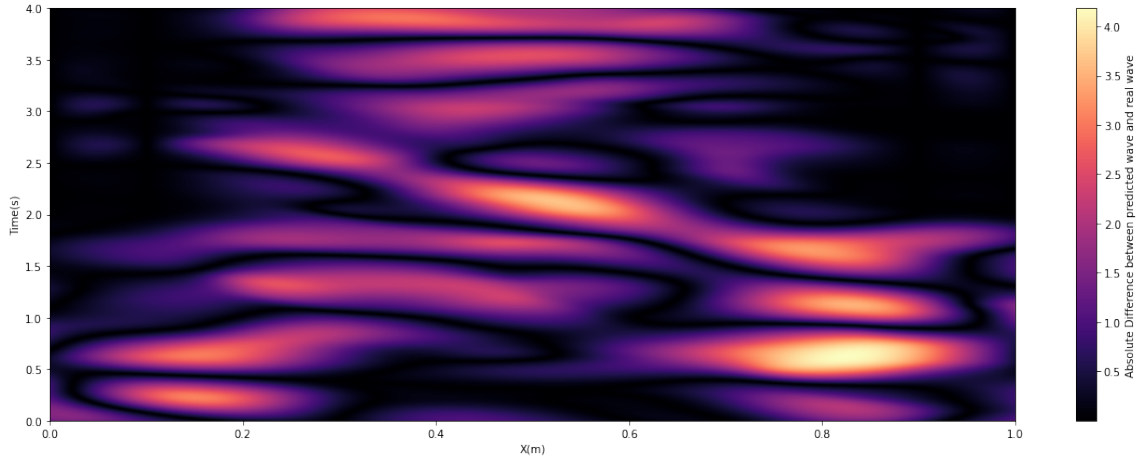
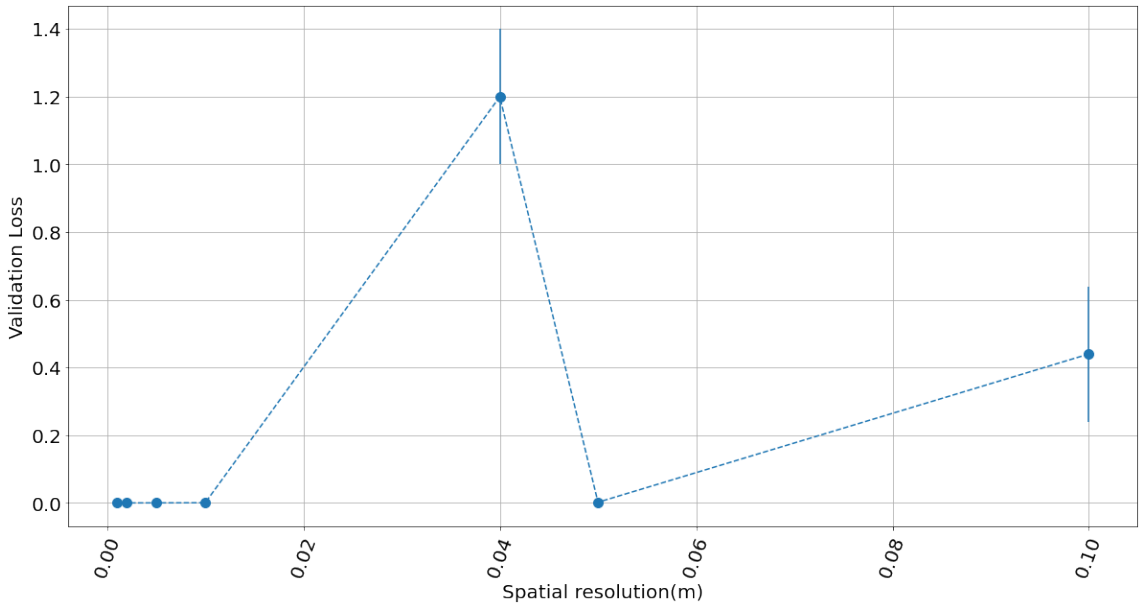


Figure 4.15: Absolute difference between model predictions and truth for$t = 3.49s$, data with $t = 3.49s$,data with $dx = 0.002m$, $dt = 0.01s$.

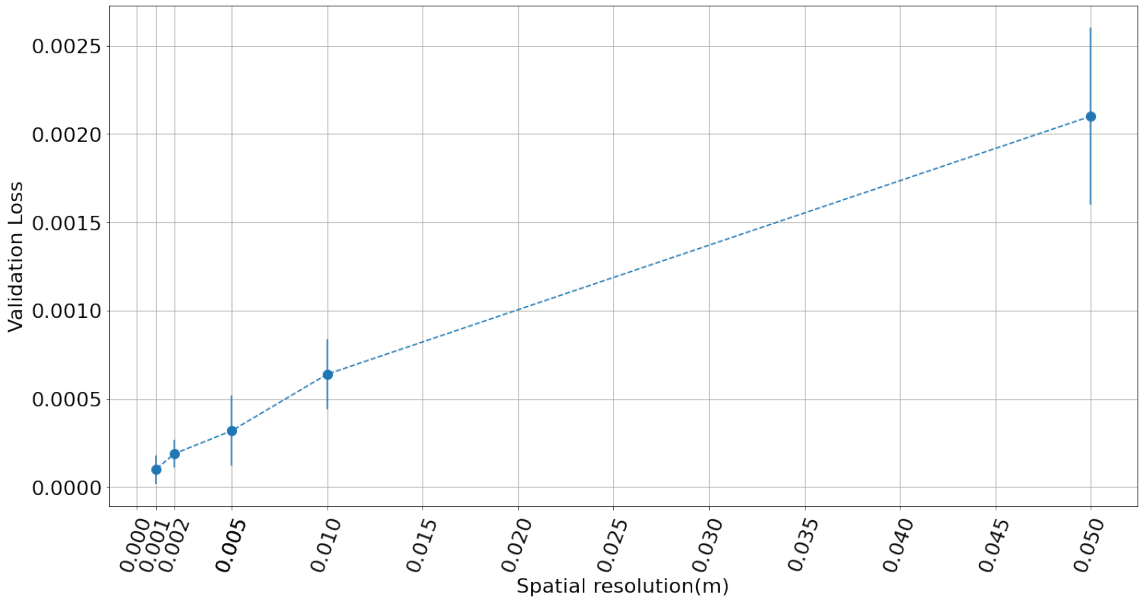
properties on the overall result. So, like before, we studied the influence of each parameter in the MSE on the second domain (or validation set). In the following sections, we present the results of the impact of resolution, noise, and string properties. All results, neural network parameters, and string properties are shown in A.2.

Resolution

Using the same values for spatial resolution (dx) and temporal resolution (dt) as in 4.4.1.1, we got the results shown in Fig. 4.16. As expected, when the resolution increased (dx and dt decreased), the fit in the validation set improved. Although for $dx = 0.04m$ we got surprisingly bad results, we also did not understand why that happened. Note that, Fig. 4.7d shows all the points present in Fig. 4.16a except the outlier at $dx = 0.04m$. This way, the downward trend is clearly seen.



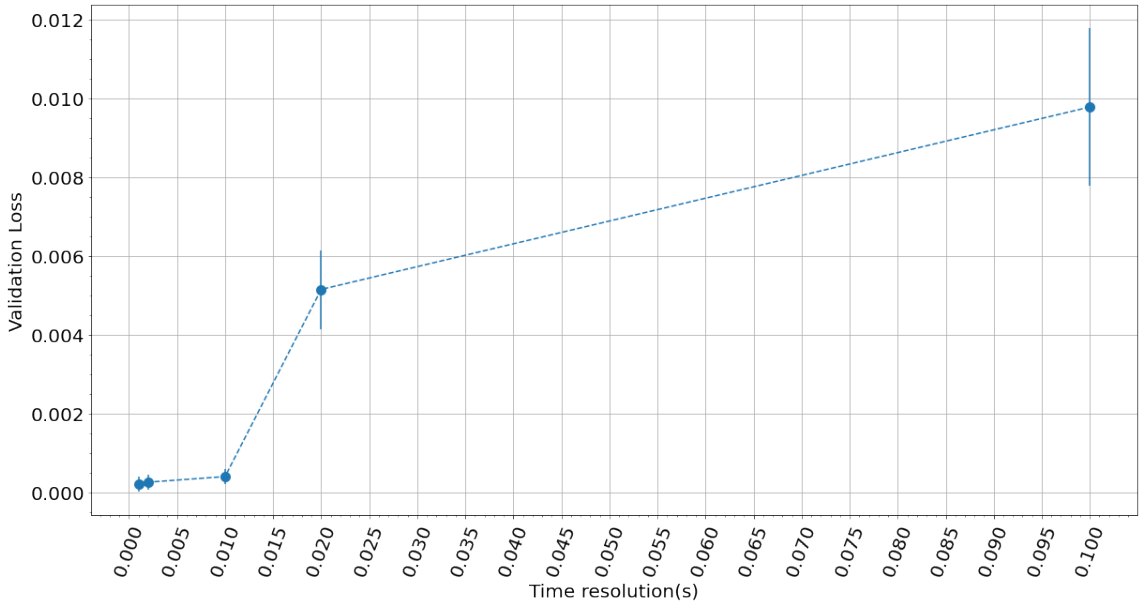
(A) MSE on $\mathscr{D}_2$ as a function of spatial resolution.



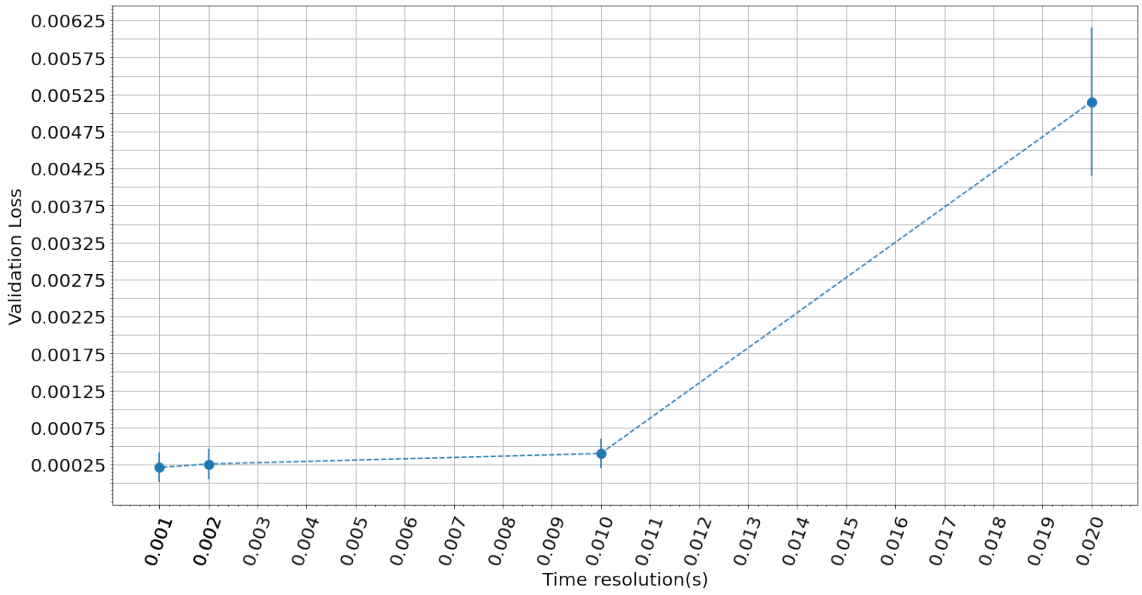
(B) MSE on $\mathscr{D}_2$ without the outlier at $dx = 0.04m$.

FIGURE 4.16: Results on the study of the effect of spatial resolution on the validation loss.

When it comes to time resolution, the same behaviour appears. The results can be seen in Fig. 4.17, and they show that as temporal resolution increases (dt becomes smaller), the validation loss decreases and as such we get better predictions on never seen time.



(A) MSE on $\mathscr{D}_2$ as a function of time resolution.



(B) MSE $\mathscr{D}_2$ without the outlier at $dt = 0.1s$

Figure 4.17: Results on the study of the effect of spatial resolution on the validation loss.

Noise Level

We got the following results for the validation loss as a function of the noise level:

Once again, we see that noise plays an important role in the overall result. Before, we saw that having 10% noise meant having an error of about 6% in the determined errors, opposed to around 3% when having 5% noise. But that difference in coefficients

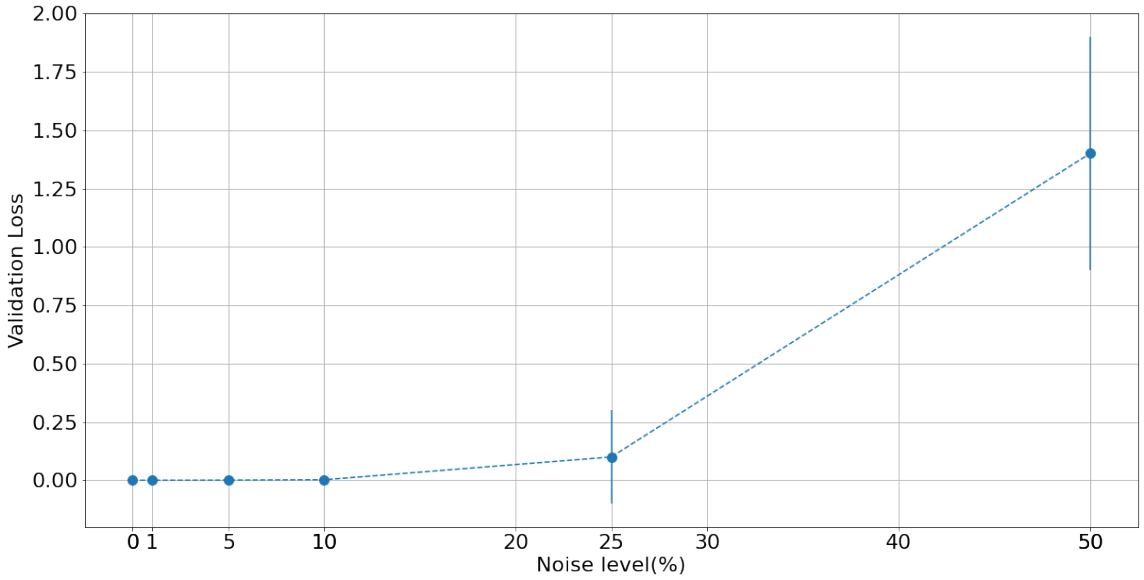


FIGURE 4.18: Validation Loss as a function of the Noise.

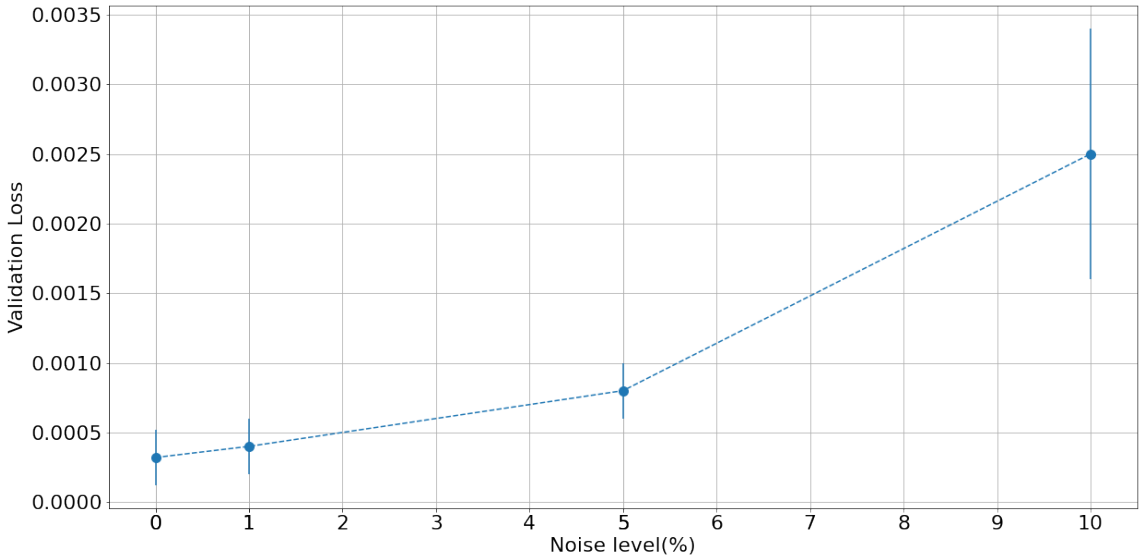


FIGURE 4.19: Validation Loss as a function of Noise only up to 10%.

determined meant that the validation loss when there is 10% of noise is almost double of when there is only 5%! This is a large downgrade in performance. Confirming our idea that having the right coefficients is crucial to extrapolate.

Wave and Axial Speeds

We thought it would be unnecessary to study the influence of the wave and axial speed when predicting the wire behaviour because, we have seen the influence each parameter has in the coefficient determination, and therefore we would be repeating the work done

before, as in principle, these parameters should not have influence in the PINN's capacity to extrapolate in never seen times.

4.4.2.1 Regular Feed Forward NNs

Because the MSE values do not give an idea of how good the predictions are, we will compare them with our baselines. A regular neural network achieved an MSE on the validation set of 1.1 while a PINN got an average of 0.0017 (for the same data of $dt = 0.010$ and $dx = 0.050$). This means that the PINN performance was about 600 times better than a regular NN. We did not test the performance of a regular neural network for every set of parameters we did for PINNs, but overall the performance difference was as in large as this case. To get an idea, the better result a regular feed-forward neural network got was on average 1, while for a PINN to get around that level of error, the data had to have 50% noise and normal resolution. PINNs almost outperform a regular neural network for data with no noise and high resolution, while having 50% of noise. This is a huge performance gap. We show that PINNs are able to comfortably beat a regular feed forward neural network when trying to solve a PDE on a domain outside its training data.

The difference in results is better perceived by seeing a GIF of a PINN predictions and then a GIF of a regular neural network predictions, one can instantly tell the difference and see the better fitting capacity of PINNs in the second domain.

4.4.2.2 Finite Differences

Solving 4.4 using finite differences is beyond the scope of this work. However, the truncation error in using a centred-finite difference method is $O(max(\delta x^2, \delta t^2))$ [34]. Table 4.2 shows the error as a function of resolution for the finite differences scheme.

dx(m) \ dt(m)	**0.1**	**0.02**	**0.01**	**0.002**	**0.001**
0.1	$1e^{-2}$	$1e^{-2}$	$1e^{-2}$	$1e^{-2}$	$1e^{-2}$
0.05	$1e^{-2}$	$25e^{-4}$	$25e^{-4}$	$25e^{-4}$	$25e^{-4}$
0.04	$1e^{-2}$	$16e^{-4}$	$16e^{-4}$	$16e^{-4}$	$16e^{-4}$
0.01	$1e^{-2}$	$4e^{-4}$	$1e^{-4}$	$1e^{-4}$	$1e^{-4}$
0.005	$1e^{-2}$	$4e^{-4}$	$1e^{-4}$	$25e^{-6}$	$25e^{-6}$
0.002	$1e^{-2}$	$4e^{-4}$	$1e^{-4}$	$4e^{-6}$	$1e^{-6}$
0.001	$1e^{-2}$	$4e^{-4}$	$1e^{-4}$	$4e^{-6}$	$1e^{-6}$

Table 4.2: Finite Differences truncation error as a function of dx and dt.

We can compare the 3^{rd} column of this table to Table A.8 and Fig. 4.16, previously obtained. Remembering that we measured validation loss in MSE, we have to apply a square root to all those values to be able to compared them. Therefore, we get the following table of results for fixed $dt = 0.01s$ validation set error for our PINN:

dx(m)	Validation Set Error
0.1	$6.6e^{-1}$
0.05	$4.5e^{-2}$
0.04	1.1
0.01	$2.5e^{-2}$
0.005	$1.7e^{-2}$
0.002	$1.4e^{-2}$
0.001	$1.2e^{-2}$

TABLE 4.3: Validation set error as a function of spatial resolution for fixed $dt = 0.01s$.

Finite differences vastly outperform PINNs, and that is normal when it comes to exactitude. And it was expected, finite differences methods are well-defined, studied, and understood. Their errors, drawbacks, and pitfalls are well known. Finite differences methods are "exact", we can be certain of the error, while PINNs still have uncertainty attached to them, it is not an exact method, training can not converge, our architecture could be wrong for the problem at hand, and much more.

Nevertheless, the biggest issue with PINNs when compared to finite difference methods seems to be stability/convergence problems. For example, finite differences have well-defined stability criteria ([30] and Stability of Finite difference methods), while PINNs are not guaranteed to converge and manual hyperparameter tuning still needs to be done to ensure PINNs are successfully trained.

All in all, we felt we did not need to make exact comparisons between PINNs and finite differences in this case study because, for now, those comparisons are mostly on sided towards finite differences; they require less computational time (vs training a PINN for about 15,000 epochs) and achieve better (or equal) accuracy. Nevertheless, there are three major advantages of using PINNs: solving the inverse problem, inference time, and when there are non-negligible extra physical effects. The first one is evidently clear, it is impossible to use finite differences schemes when one does not know the PDE. In that case, one could use a PINN to solve the inverse problem and then, when the PDE is determined, solve it using finite differences. So far, what we have seen points to this being the best scenario/method, because, as things stand, PINNs can not compete with finite differences (or even better numerical PDE solving methods).

When it comes to the second advantage, inference time, it is easy to see that it is more computational expensive to compute finite differences schemes over all the domain (spatial and temporal) than it is computing the forward pass of a neural network. This means that once we have a trained PINN is much faster computing predictions than when using finite differences because we have to solve the PDE for all the domain every time. This can be beneficial in industrial settings where having (almost) real-time predictions is crucial, but training sped/costs is not an important factor. If that is the case, a trained PINN should outperform finite difference schemes.

Finally, the last case when a PINN should outperform finite differences is when there are extra factors related to the physics of the system that we didn't expect or can not neglect, for example, if one requires precise determination of the wire position/velocity up to a point where we can not afford to ignore heat dissipation or air resistance. A PINN built with a good sparse library of terms should be able to "learn" that there are extra physics factors that appear in the data, and then make physical adhering predictions. While using finite differences, we might not know what are the important extra terms (which term influences the results and which do not

4.5 Conclusion

Regarding the task of determining the PDE coefficients, the PDE coefficients were successfully determined with $(0.9 \pm 0.1)\%$ error for the ideal configuration, where there is no noise, and we have high spatial and temporal resolutions. For a configuration with 5% noise and mid-level resolutions ($dx = 0.04m$ and $dt = 0.01s$), we got an error of $2.42 \pm 0.4\%$, which is below our maximum acceptable error of 3%.

It is important to mention that the error depends greatly on the setup, mainly on the 3 parameters we studied: resolution, noise, and string properties (c and v). Therefore, the results should be better interpreted with the help of plots and tables. It is hard to say that we got an overall error of $x\%$ because it depends on so many factors, and we choose only to study some.

Regarding extrapolation and predicting the wire's behaviour, our approach successfully improved the performance of a regular neural network, by introducing physical constrains, by a huge margin. It was verified that adding physical knowledge to the learning process of a neural network makes it generalise better, mainly because when it comes to extrapolate in domains where data is not available,PINNs can compute the

PDE loss and optimise the neural network to decrease it. Furthermore, we continue to see that resolution, noise, and string properties are important factors in the end result.

We also compared PINNs to finite differences methods and conclude that PINNs are still behind them when it comes to exactitude, total computation time (training and inference), and stability. But PINNs showed promise: Most importantly, they solved the inverse problem with great success, they provide much faster inference times, and they can provide greater flexibility when working with difficult geometries, boundaries, and PDEs where finite differences do not work[21].

Chapter 5

Case Study 2: Forecasting the Weather on an Offshore Wind-Farm

The 2^{nd} case study we tackled was forecasting the weather, mainly ***forecasting temperature and wind speed on a offshore wind-farm***. This study was inserted under the WaterEye project, which is an EU project that has the goal of improving sensing,monitoring,control, and diagnose of offshore wind farms. Knowing that weather conditions (mainly temperature, humidity, and wind speed) play an enormous role in wind turbine power generation and life cycle, knowing beforehand the weather conditions for the foreseeable future will help predict variables such as: power generation, degradation of materials, maintenance time frames, and much more. With that said, we were tasked with assessing the possibility of using PINNs to help forecast the weather on a long-term window, plus determining key physics of the system, for example, which localised phenomena were dominating and which were not important in that area.

Why use PINNs in this case? The reason is that adding physical knowledge to such a chaotic problem should, at least, help neural networks generalise better and get better predictions. Also, solving the inverse problem and determining the unknown physics should be enough to then use other established methods, such as finite differences and finite elements.

5.1 Goals and Baselines

The main goal was to long-term predict (years) wind speed and temperature on a particular offshore wind farm using PINNs. Obviously, if we could forecast the weather

accurately using PINNs, we could also obtain the underlying physics of the system, which was a prerequisite to achieve our goal.

We mainly had two baselines:

1. Predict tomorrow's temperature/wind-speed as being equal to today's temperature/wind-speed (naïve baseline)- A simple assumption, but a good enough baseline that sometimes it is hard to beat.
2. Predict tomorrow's temperature/wind speed as the mean of the last 5 days temperature/wind speed – Still a simple baseline, but even harder to beat.

As simple as these baselines are, they can be harder to beat, and they give us a good idea if our models are sufficiently good. For further comparisons with state-of-the-art weather forecasts, we intended to compare our results with other forecasting techniques, such as ARIMA models, auto-regressive models, or even the more recent Temporal Fusion Transformers [35].

5.1.1 Research Questions

There were important questions that we wanted to answer, apart from the two main ones of determining the underlying PDE system and of forecasting the weather, for example: Which is the longest time period we can accurately (up to a certain low-value error) forecast the weather? Does increasing the 2D grid size improve results or not? The main questions we wanted to investigate were the following:

- Can we just build a very large library of functions and determine the PDE without having an idea what it should be like?
- How does the error in the forecast increase as we try to forecast in longer time windows?
- Can we use CNNs with PINNs to leverage their power in learning 2D patterns?
- Is spatial and temporal resolution important in the end result?
- Should we instead use 3D data?

5.2 Framework

5.2.1 Physics of the System

The basis of modelling dynamic flow in the atmosphere are the Navier-Stokes equations. These equations govern the flow of fluids such as water,air and gases. Mathematically, a fluid can be represented by three functions: density, $\rho(\boldsymbol{x}, t)$, pressure $p(\boldsymbol{x}, t)$, and velocity $\boldsymbol{u}(\boldsymbol{x}, t)$. The density and pressure functions are scalar functions of the position vector $\boldsymbol{x} = (x_1, x_2, x_3) \in \Omega$, where Ω is the domain filled by the fluid. The velocity function is a vector function given by $\boldsymbol{u}(\boldsymbol{x}, t) = (u_1(\boldsymbol{x}, t), u_2(\boldsymbol{x}, t), u_3(\boldsymbol{x}, t))$.

It is out of the scope of this work to explain how to deduce them, but the incompressible Navier-Stokes equations are the following:

$$\begin{aligned} \nabla \cdot \boldsymbol{u} &= 0 \\ \frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nu \nabla^2 \boldsymbol{u} &= -\frac{1}{\rho}\nabla p + \boldsymbol{f} \end{aligned} \tag{5.1}$$

Where, f is a force density per unit volume and represents the acceleration of the body acting on the fluid (for example, gravity or inertial accelerations), ν is often called *viscosity* of the fluid, and rho is the density of the fluid (considered constant in this approximation).

These equations are non-linear because of the convective term $(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}$, and for that reason solving these equations analytically has proved impossible so far. In order to reach a solution, approximations are used, and each one of these approximations gives way to an approximate solution/model. Approximations are usually done to simplify the problem and reach solutions that make sense inside a particular problem, but cannot be used in many other situations.

In our case, we tried to test which model fitted our data best by solving the inverse problem using PINNs. For example, one of such models that we tested was *Rayleigh-Bénard convection model*[36], a model for turbulent convection between two horizontal layers.

It is important to note that weather behaviour is a chaotic phenomenon and very hard to model. Many people have spent their lives trying to improve current state-of-the-art models, and this is still an ongoing research area. Mainly, the ability to long-term forecast, say, longer than a 7-day range, is still limited. Currently, numerical weather prediction has become the central component of weather forecasting, and it is done on

supercomputers given the huge number of variables and the spatial domain involved. All this to say, weather forecast is far from a simple problem, by the contrary, it is probably one of the hardest physics-computational problems there is.

5.2.2 Dataset Description

We worked with an open source dataset [37] from the National Center for Atmospheric Research (USA). This is an *analysis* dataset, which means that it is composed of real observation data that is complemented with simulated data when it is not possible to obtain data from observations. The models used to infer the missing data are very advanced, combine mathematical and physical knowledge, have excellent results, and have been improved over the years. As such, we made the decision to use this dataset and consider it as ground truth data.

The particular subset of the dataset we used had the following characteristics:

- ***Localisation***: Gran Canarias offshore wind farm – centred around the coordinates (28 N,15 W).
- ***Time range***: 2008-2021 (approx. 13 years).
- ***Time resolution***: 1 day.
- ***Spatial Range***: 2 Dimensional grid (total side 5º) at a fixed height (x-y plane for a fixed z).
- ***Spatial resolution***: 1º on both sides of the grid.
- ***Variables available***: Temperature, Wind speed vector, Pressure, and Humidity.

5.2.3 Detailed Framework

We decided to start with the simpler case of selecting one point of the 2D grid and trying to forecast only the temperature at that point. See if it works, and only then work on the full 2D grid and wind speed. This way we would get a better sense of the dataset, on how PINNs work in this case, and if we should continue to the harder 2D case.

1D Weather/Fixed Point Forecasting

The framework we ended up using was the following:

1. Choose one point on the grid.
2. Separate the data into training(~ 80%) and validation(~ 20%) sets.

3. Normalise the temperature and the time variables.
4. Compute the Fourier Transform to determine the dominant frequencies.
5. Build a library of functions composed of sinusoidal functions having frequencies equal to the ones we previously established as being the dominant ones (using the Fourier Transform).
6. Use PINNs to determine the underlying PDE and forecast the temperature:
    - Train a PINN to learn the (approximate) PDE and extrapolate on never seen times.

The main difference from the previous problem, the string case, is that we decided to build a library of functions using sinusoidal functions after we saw that the temperature function was clearly periodically, and we could approximate it as a sum of sinusoidal functions. We will see it clearly in the results section. Otherwise, all other steps are common to the string case: train a PINN to determine the coefficients of the PDE and then use the determined PDE to train a PINN to extrapolate to never seen times.

2D Weather Forecasting

After we studied the 1D case, we tried to solve the 2D case first by having a large sparse library of functions, with many terms (derivative, polynomial, and cross-terms); and secondly by building libraries from known weather model approximations. The framework is similar to the 1D case:

- Get the 2D grid data for all the time period.
- Separate the data into training($\sim$ 80%) and validation($\sim$ 20%) sets.
- Normalise the coordinates, temperature, and time variables.
- Build one of two libraries of possible terms:
    1. Large sparse library with many terms.
    2. Library based on some weather model.
- Use PINNs to determine the underlying PDE.
- Use a PINN with the determined PDE to extrapolate to new times.

DeePyMoD

We used DeePyMoD [33], which is a *"modular framework for model discovery of PDEs and ODEs from noisy data"*. DeePyMoD follows the same structure of solving inverse problems, by posing it as a regression problem, as in 3.6, and then finding a vector ξ of PDE coefficients. In the string problem we knew exactly what the PDE was and what derivative terms should appear in it, but in the weather, as we have seen, we do not know with certainty the PDE system. Either we can make approximations and assumptions and get an approximate PDE system, or the library of candidate terms will have many candidate terms. Most of the candidate terms will not appear in the final PDE system, meaning that we are now trying to solve a sparse regression equation. DeePyMoD was built with the goal of having many options on how to solve this sparse equation (and enforce sparsity), so we thought we should use it to simplify things. It should not add more complexity; it is just a simpler way of solving sparse regression problems using PINNs.

5.2.4 Code in GitHub

All code is available on the GitHub repository in the *Weather* folder. The notebook ***Weather1D.ipynb*** handles the 1D case and the notebook ***Weather2D.ipynb*** the 2D case. Some examples of the images obtained are under the folder *Weather/Images*

5.3 Results

5.3.1 Fixed Point (or 1 dimentional) Case

We chose to study the evolution of temperature at the point (28N,15W), which was the centre of our grid. The plot of temperature as a function of day is as follows:

We clearly see periodic behaviour. We followed by computing the Fourier transform of this signal and determining the dominant frequencies. With those frequencies, we build the library of candidate terms, filling it with sinusoidal terms having frequencies equal to the dominant frequencies in the signal. Remember that we are solving eq. 3.6, so the library of functions Θ should be equal to the first derivative and not the signal itself. Then, for example, if one of the dominant frequencies is w_1 one of the terms in Θ could be $w_1 \sin(w_1 t)$.

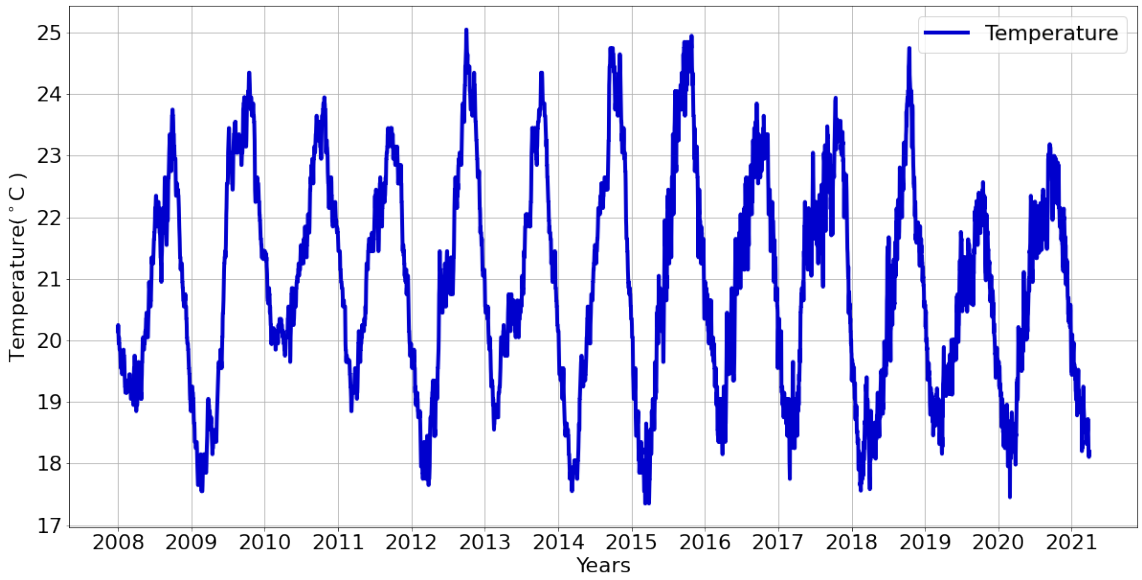
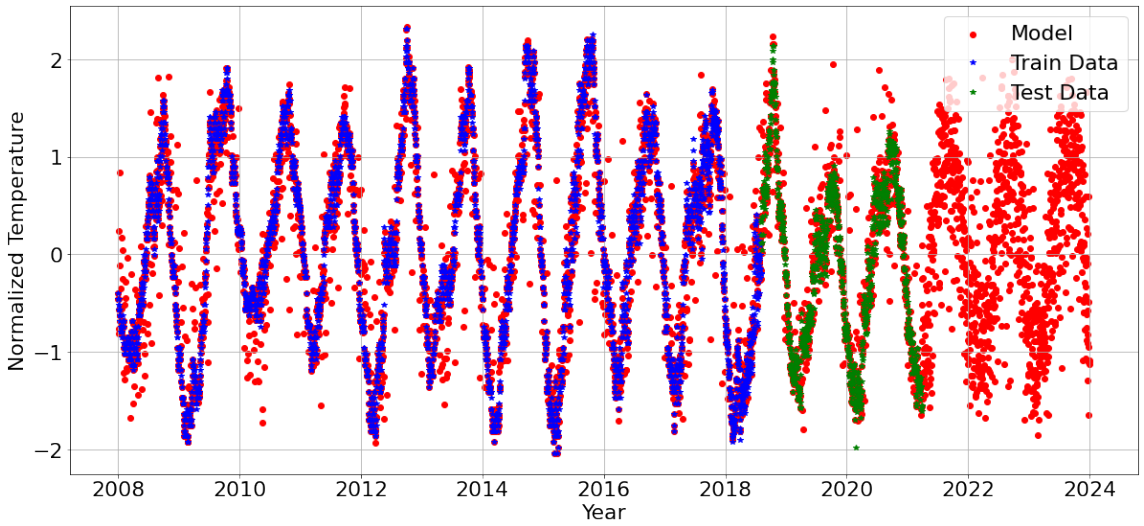
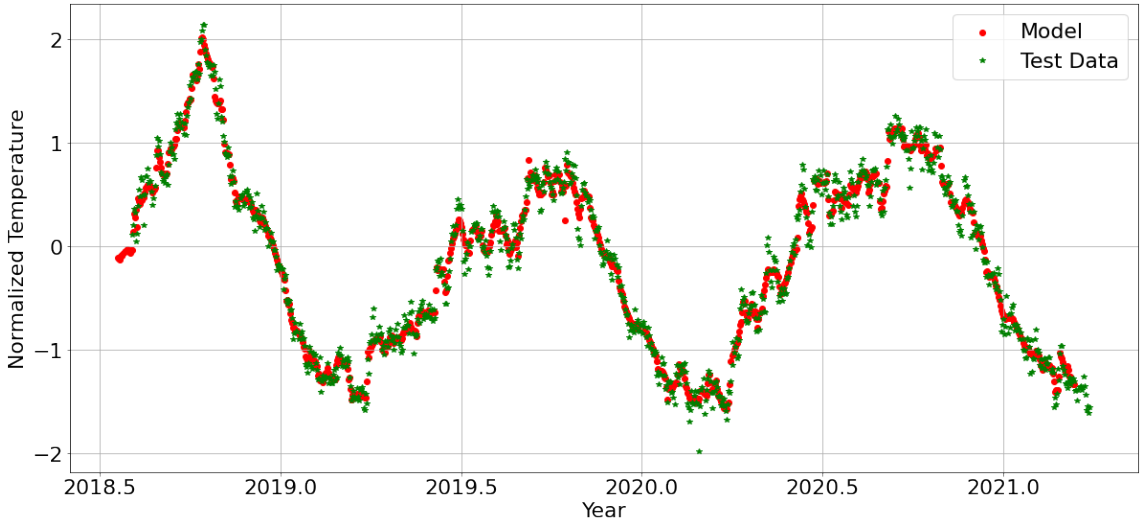


Figure 5.1: Temperature for the coordinates (28 N,15 W), starting in 2008 and ending in early 2021.

After training a PINN in 80% of the available data and testing/validation on the missing 20%, we got the predictions shown in Fig. 5.2. The second image only shows the testing interval, while the first shows training, test, and some points in even further in time.



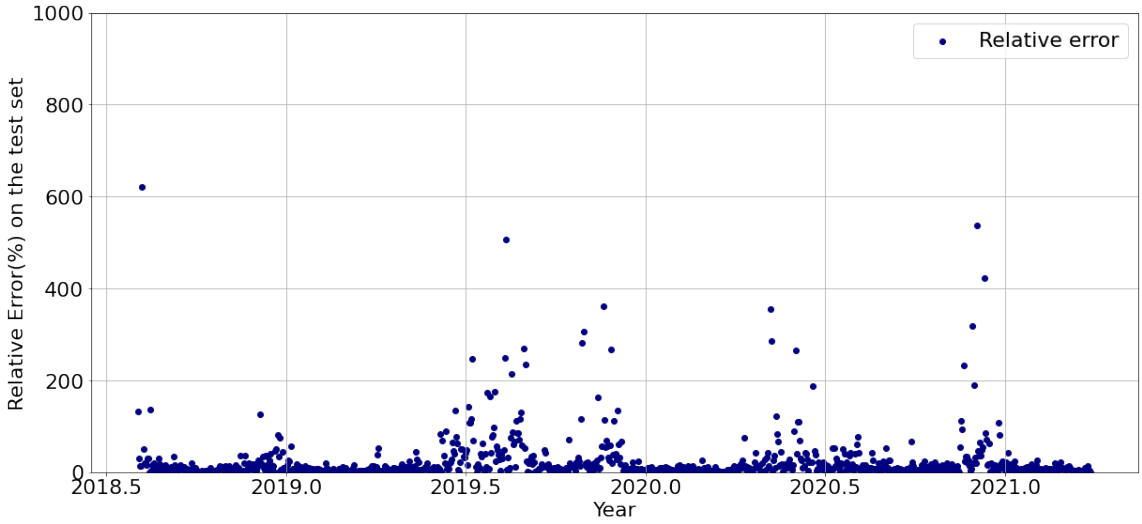
(A) Predictions of the PINN.



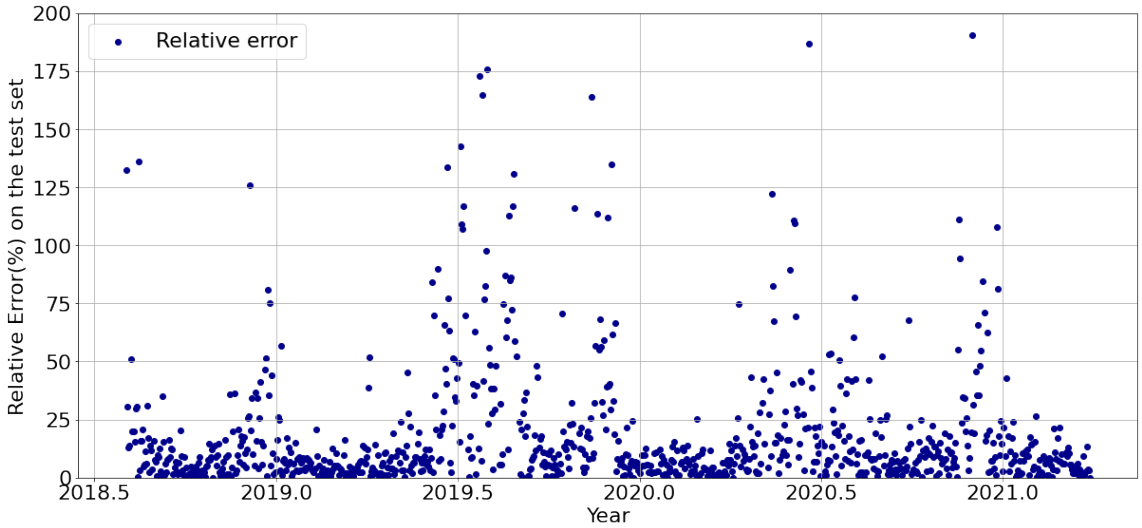
(B) Predictions of the model compared to ground truth on the test set.

FIGURE 5.2: PINN predictions compared to the ground truth on training and test sets.

We can see that the predictions seem very good. The predictions in the test set appear to be close to the real values, sometimes falling very close to the real value. Nevertheless, there are points that are lagging the ground truth; visually, they seem to be correct but come a day after, a day before, or 2 days after, 2 days before. Fig. 5.3 shows the errors for the test set, excluding the largest outliers. We can clearly see that there are still points that have a large error, mainly points where the inflection of the curve occurs. In those regions, it is harder for PINNs to compute exact derivatives, because they are very large (large slope) or near zero (inversion point) and larger errors appear. When comparing Fig. 5.3 with Fig. 5.2 it is easy to see that the zones of higher error match exactly with the zones where the ground truth temperature.



(A) Error on the coefficients as a function of noise percentage.



(B) Error on the coefficients as a function of noise percentage - zoomed.

Figure 5.3: Results on the study of the effect of noise on the overall error on the determined coefficients

Our PINN achieved ***MSE of 0.09 and Mean Absolute Error (MAE) of 0.07*** (both in the test set). Remembering that we normalised the temperature, converting MAE to degree Celsius gives a MAE of (0.12 ± 0.02) °C. That means that our predictions are incorrect on average 0.12 °C. We can compare the PINN result to the previously established baselines:

- ***PINN***: MAE = (0.12 ± 0.02) °C
- ***Today's temperature = Yesterday's temperature (naïve baseline)***: MAE = 0.14°C
- ***Regular feed-forward neural network***: MAE =(0.23 ± 0.04)°C

This means that PINNs represent an improvement of 8% when compared to the naïve baseline and an error of 77% when compared to the regular neural network. Adding

physics knowledge to the regular neural network made our model improve 77%. Still, beating the naïve baseline only by the small margin of 8% seems rather poor, and probably shouldn't be considered as a real improvement. Furthermore, we see that in the regions where temperature trends invert (the peaks), the error explodes; we are not able to model the temperature in those regions most certainly because computing derivatives in those regions is harder, and we are forcing our PINN to adhere to bad/wrong derivatives.

Another important aspect was determining the active terms in the PDE and each scalar coefficient. And here we got worse results.***The coefficients we got, from our library composed of sinusoidal terms, varied greatly from training run to training run***. We think that this is explained by the fact that there are many ways to express a periodic function as a sum of many other periodic functions. For example, on run 1 the neural network converges to express the signal as the sum of $1.5\Theta_1$ (first candidate function of Θ), plus $0.1\Theta_2$; but on run 2 it could be the case that it converges to express the temperature as a sum of $0.5\Theta_1 + 0.3\Theta_4 + 1.3\Theta_6$. *Both are accurate approximations* of the signal, different ways of expressing the same signal. That was what we think happened, and we could not find a way to steer the neural network to converge to one solution of coefficients. We decided to move on to the 2D case because that was where we really wanted to learn the PDE (not an approximation as we were doing here, in the 1D case), and add wind speed.

5.3.2 Two Dimentional Case

The temperature, horizontal wind speed, and vertical wind speed for the day of 07-08-2008 can be seen in Figs. 5.4, 5.5, and 5.6 respectively.

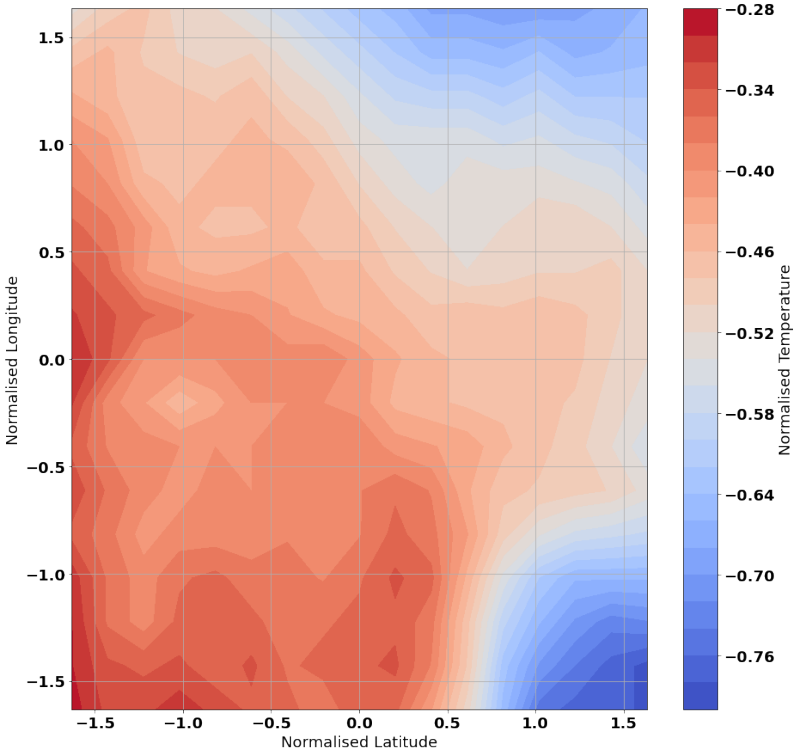


Figure 5.4: Normalised temperature speed over the 2D grid.

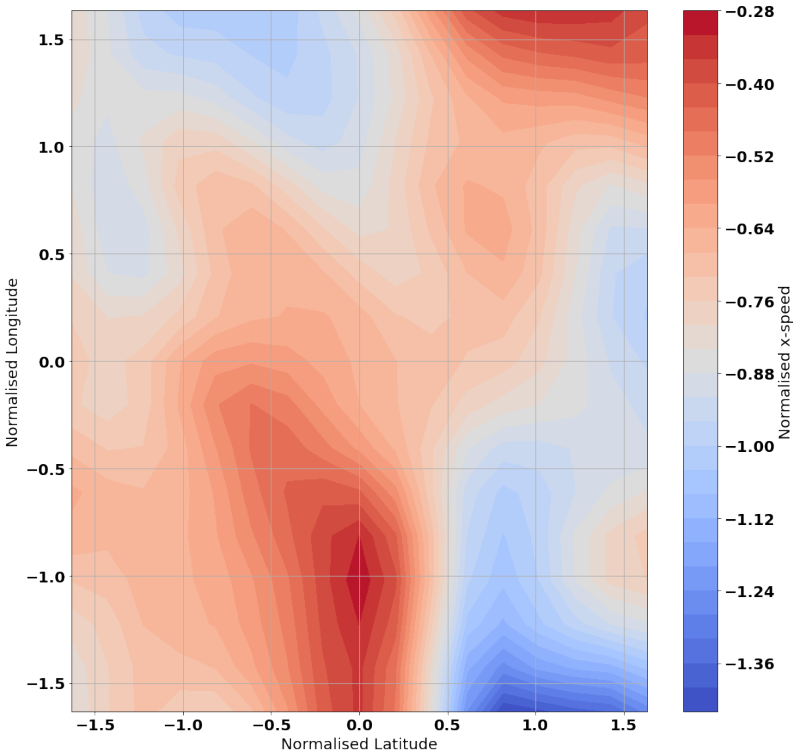


Figure 5.5: Normalised horizontal speed over the 2D grid.

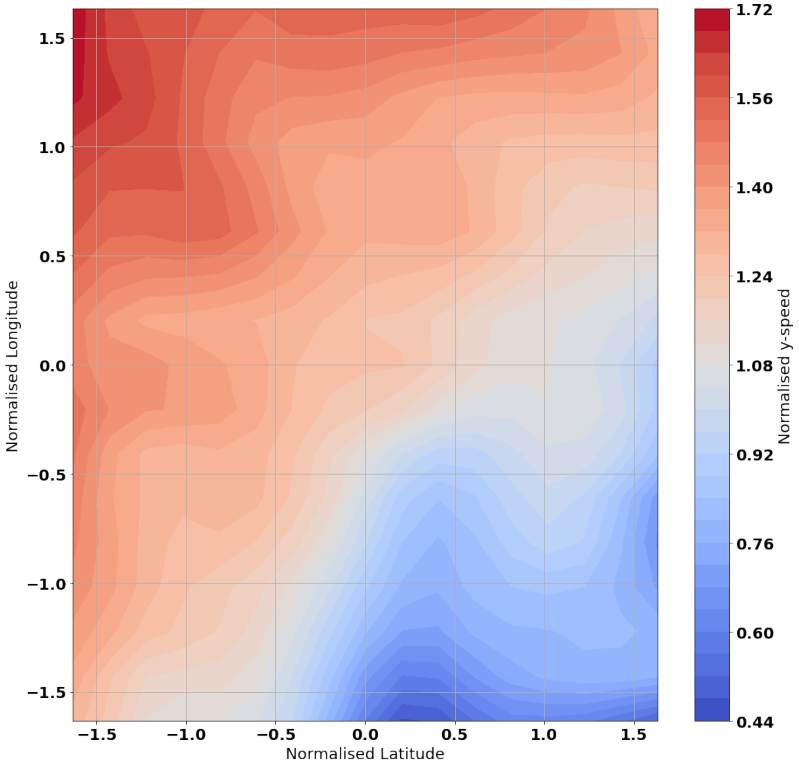


FIGURE 5.6: Normalised vertical speed over the 2D grid.

Opposed to the 1D case, now we can not approximate these fields as a periodic function because we are not looking at only one point but rather a full 2D grid. Because of that, we devised two possible strategies, differing in how we build the library of possible candidate terms Θ :

1. Θ is a mix of many possible terms (derivative terms, polynomial terms, cross terms).

2. Θ is built using one specific weather model, the Rayleigh-Bérnard convection model.

In 1) we leverage the fact that we can solve a sparse system to find which active terms are, in fact, in the underlying PDE. So in the end we would know which terms are in the PDE and their scalar weight. In 2) we would make the regression problem easier by reducing the number of terms in Θ, only using terms based on the Navier-Stokes system of equations.

Sparse Library

Filling the library of candidate terms Θ meant, for example, having:

$$\Theta = \begin{bmatrix} T_{xx}(\{x,t\})_0 & \boldsymbol{w}_{xt}(\{x,t\})_0 & \dots & \boldsymbol{w}_x T_t(\{x,t\})_0 \\ T_{xx}(\{x,t\})_1 & \boldsymbol{w}_{xt}(\{x,t\})_1 & \dots & \boldsymbol{w}_x T_t(\{x,t\})_1 \\ \vdots & \vdots & \vdots & \\ T_{xx}(\{x,t\})_N & \boldsymbol{w}_{xt}(\{x,t\})_N & \dots & \boldsymbol{w}_x T_t(\{x,t\})_N \end{bmatrix} \tag{5.2}$$

Where $T(x,t)$ and $\boldsymbol{w}(x,t)$ represent the temperature and velocity field at point x and time t respectively. We tried having many terms in Θ, sometimes having 20/30 possible terms.

We did not succeed in finding stable active terms, meaning that for different runs different terms would become active. This differed from the 1D case because before we knew we were determining an approximation of the PDE. This time, we were trying to determine the true PDE instead. We came up with two possible explanations for this problem:

1. The Sparse Least Squares problem is a hard one, and small variations in the second derivative (computed using auto differentiation) meant different solutions to the least squares problem.

2. We were not using enough variables needed to construct the Navier-Stokes equations.

We did not find a way to verify our suspicions about explanation 1), but our model was over-fitting the data, meaning derivatives were being correctly computed using auto-differentiation, but after each run we had different coefficients and active terms. Something was going wrong when computing the least-squares solution. Could it be that we had too many possible terms, and it was making the least-squares solver vary between solutions? Unfortunately, we did not manage to precisely find what was happening.

However, explanation 2) is easier to explain. We used only temperature and wind speed to try to model eq. 5.1, which also includes pressure. Therefore, we were, in fact, missing pressure, which is an important variable. We knew that this could happen, but we thought that adding another variable would make training much more complex and computationally expensive. So we tried only with wind speed and temperature at first, wanting to add pressure in the end to see if this corresponded to improvements. Unfortunately, we ran out of time.

Rayleigh-Bérnard Convection Model

The Rayleigh-Bérnard convection model [36] is a model of flow between two horizontal layers, with a horizontal layer of fluid heated from below so that the lower surface is at a higher temperature than the upper surface. Turbulent convection is a major feature of the atmosphere. This model is an idealised and very simplified model of atmosphere dynamics. We used it precisely because of its simplicity. The governing equations of such a system are the following:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{5.3a}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nu \nabla^2 \boldsymbol{u} = -\frac{1}{\rho}\nabla p + \boldsymbol{f} \tag{5.3b}$$

$$\frac{\partial T}{\partial t} + (\boldsymbol{u} \cdot \nabla)T = k\nabla^2 T \tag{5.3c}$$

We only used equations 5.3a and 5.3c to build our library of candidate terms, for the same reason as before: we did not want to include pressure (only after we tested with temperature and wind speed).

Using this approach, we still could not determine a stable PDE. Different training runs meant different active terms, and they always differed in value, never seeming to be converging towards one particular value.

While trying to understand what was wrong, we thought of computing equations 5.3 "manually" using finite differences and see if they were true for the data we had. And, they turned out not to hold, so the Rayleigh-Bérnard model did not really model the data we had. Meaning, we had to give up on this approach and focused only on trying to improve the sparse regression problem or even the neural fitting capacity.

5.3.3 Improvements and Further Results

After having bad results for both of our Library approaches, we tried other approaches, namely:

Using a CNN: leverage the power of CNNs to learn spatial patterns.

Use a different dataset:

- **Higher temporal resolution**: Previous dataset 1 day → New dataset 6 hours.

- **Higher spatial resolution**: Previous dataset 0.5°→ New dataset 0.25°.
- **Smaller grid size**: Previous dataset approx. 500 km x 500 km → approx. 450 km x 350 km.
- **Land location only**: before, our grid was located on a sea-land interface.

Recall that PINNs require that if we want to differentiate the target variable (T), with respect to some other variables (x, t), for example, if we want to compute $\frac{\partial T}{\partial x}$ or $\frac{\partial T}{\partial x}$, x, t must be inputs to the neural network. Otherwise, we cannot compute derivatives using backpropagation. In CNNs, the input is an image expressed as a 2D array, and in our case each entry would be the temperature/wind speed in the corresponding coordinates. Because x, t are not explicit inputs to a CNN, we computed derivatives using a finite-difference approach, based on the one in [38], which computes derivatives using a kernel/stencil.

Unfortunately, even after applying these changes, we could not determine the underlying PDE. We kept encountering the same problem of not being able to have a stable solution of the sparse least squares method. Failing to determine the underlying PDE system of the data means that our approach, of using physical knowledge (the PDE system itself) to constrain the neural network's optimisation process, does not work. So, it becomes irrelevant to present results on the performance of PINNs while extrapolating if they do not have a correct PDE to be optimised with respect with.

5.4 Conclusion

As for the simpler case, the fixed-point temperature evolution study, we managed to use PINNs to beat our baselines, adding physical knowledge to a regular feed-forward neural network improved its performance by 77%. We acknowledge that feed-forward neural networks are not the state-of-the-art when it comes to time-series forecasting. For that reason, we compared the PINN performance with the simple baseline of predicting today's temperature as being equal to yesterday's temperature, and PINNs only slightly outperformed it. Furthermore, we could not determine an exact PDE. We tried to determine an approximate PDE (solution to the PDE) as a sum of sinusoidal functions, but could not get a stable representation/solution. We think that was the case because the

same periodic function can be expressed by many different sums of other periodic functions, and the neural network optimisation (and initialisation) made the model converge to a different solution every time.

As for the more complex case, the forecast of temperature and wind speed over a 2D grid, the results were even worse. We could not find the underlying physics of the system, and without them, it is impossible to train a PINN. Ultimately, as the internship period came to an end, we were unable to find a way to solve this problem. Perhaps because we were missing key variables like pressure, humidity, etc., and given the complexity of trying to predict temperature over a 2D grid meant we were always going to encounter trouble in solving the sparse least squares method and optimisation (with respect to both losses) of the neural network at the same time. Perhaps, the data we had still had very high resolution, and we could not properly model the evolution of temperature on the 2D grid. Perhaps, using better architectures like CNNs, GANs, Transformers for 2D data was needed, and only when having a near perfect fit on the training data auto-differentiation meant derivatives were also near perfect which would result in always the same coefficients coming out of the sparse least squares (like in the string problem). Unfortunately, we did not have time to work out all these alternatives and probable solutions.

Probably trying to tackle such a complex problem, as weather, with PINNs, which are a new topic and still there is much research to be done in order to solve stability, convergence, and training issues, was optimistic. Nevertheless, while we did it, we learnt what are some drawbacks of PINNs, what is still to be tuned and improved: better stability/convergence criteria, better loss balancing schemes, better PINN architectures suited for each type of problem. One of the goals of this internship was to use PINNs in real-world problems, and even if we failed, we tried, and we hope that in the future we can build upon what we learnt and build a better framework to tackle this kind of problem.

Chapter 6

Concluding Remarks and Future Work

This work intended to present the work done during my internship at Flanders Make. The main goals was to understand what PINNs are and when/how they can be used to improve regular neural networks, or classic PDE numerical solvers. In addition, we intended to study the performance of PINNs in real-world scenarios.

In Chapter 2 we have seen the topics of Machine Learning and Deep Learning needed to understand PINNs. Then we saw what a PINN is, how they improve neural networks by adding physics knowledge to the training scheme, how they can improve a neural network's extrapolation capabilities, and even solve the inverse case.

In Chapter 4, we used PINNs to tackle our first use case: a vibrating string, fixed at both ends, with added horizontal speed. The goals were divided in two separate steps: 1) Determine the PDE coefficients; 2) Predicting the wire behaviour on never seen times. We show how the results, in both steps, depend on variables such as data resolution and noise level. We show that, for certain problem configurations, we can determine the PDE with less than 3% (even getting to less than 1% in ideal cases) and then accurately predict the wire's behaviour outperforming greatly regular neural networks. However, we show that the convergence of PINNs to the right coefficients greatly depends on the properties of the wire (c and v). Furthermore, while comparing PINNs to classic PDE solving algorithms like finite differences, we show that PINNs still struggle with training time, and stability.

In Chapter 5, we used PINNs to forecast temperature and wind speed on an offshore wind farm, as well as to determine the underlying PDE from data. First, we simplified

the problem by only working with one point in space (we called it fixed point case), and after that, we tried to predict weather/wind speed on a 2D spatial grid. For the fixed point case, we have shown that we were successful in forecasting the temperature, with a Mean Absolute Error of (0.12 ± 0.03) °C, which is an improvement of more than 70% compared to a regular feed-forward neural network.

However, we were unable to obtain stable PDE coefficients for both the fixed point and the 2D cases. In the fixed coordinates case (1D), we determined a PDE for each training run, which helped to improve the forecasts. But, given our approach of trying to model the temperature as a sum of other periodic functions, for each training run the learning process converged to a different way of expressing the temperature as a sum of such periodic functions. For that reason, we never determined one single PDE, but many ways of expressing an approximate PDE. For the 2D case, we show how we tackled the problem from many sides but did not succeed in determining an underlying PDE, and without it, it is pointless to use the PINN framework because we do not have the physical knowledge to constrain the learning process. For that reason, our PINN approach failed in its main goal of determining the underlying physics of weather/atmospheric behaviour and then using them to improve extrapolation.

In conclusion, in this internship we have seen how PINNs have potential to be PDE solvers, how they improve regular neural networks using physical knowledge, and how PINNs can be used in a real-world scenario in the vibrating string problem. At the same time, we encountered some drawbacks of PINNs, mainly stability, convergence, and training time. As for the weather forecasting case, trying to tackle such a complex problem with PINNs was perhaps too optimistic. In order to solve such a complex problem, a more robust framework is needed, proven by the fact that state-of-the-art weather forecasting models involve many forecasting methods and an enormous amount of data and variables as input. Such framework could also have been an improvement of PINNs, and there is where the future work lies, improving and extending the PINN framework we used. Some improvements could be solving problems such as stability/convergence, optimising the code to decrease training time. But in order to solve more complex problems, like weather modelling, a revamp of our framework is needed, perhaps changing the architecture of neural networks, from feed-forward to GANs, LSTMs or Transformers [39]; there are examples of many architectures and PINNs frameworks that have been used in complex problems. Perhaps improving the least squares regression method,

making it easier to converge to appropriate solutions. It is important to say that even if our PINN approach in weather forecasting failed, we were still able to learn and see by ourselves the drawbacks/shortcomings of PINNs and think how they should be overcome.

In summary, PINNs show great potential as both inverse problem and PDE solvers, but in order to achieve all that potential, there are still some problems to be solved. We talked about how PINNs should be adapted to easily be used with different types of architectures, mainly more recent and advanced networks like GANs and Transformers. And there are examples of research already doing precisely that ([25], [39], [21][40]), but there is still much to understand and improve. We still do not fully know how to best train a PINN: which are the best activations to use, how regularisation should be done, how should we choose the weights (w_{PDE} and w_{MSE}), should we pre-train a regular neural network and then introduce the PDE loss, and much more. Only with time, trial, and failure should all these questions be answered, and with the traction PINNs have been gathering, I hope they will be solved in the near future. Who knows if PINNs, in the future, will be used on edge cases (inverse problems, difficult geometries or boundaries, complex PDEs) in detriment of finite differences/elements methods?

Appendix A

Appendix A - Case 1

A.1 Coefficients (Inverse Problem)

Resolution

The following neural network parameters were used:

- ***Number of layers:*** 6
- ***Number of units per hidden layer***: 50
- ***Activations***: Sinusoidal in the 1^{st} layer, tanh in the hidden layers, and linear in the output layer.
- ***Optimiser***: Adam
- ***Weight decay***: 10^{-8}
- w_{kpde}: $[10^{-6}, 10^{-3}]$
- ***Number of epochs trained***: 15×10^3

The string/data properties were the following:

- $c = 1$
- $v = 0.5$
- ***Number of modes***: 4
- ***Initial condition***: Constant coefficients ($A_n = [0.08, 0.49, 0.35, 0.15, 0.08]$)
- ***Noise***: 0%
- ***L***: 1 m
- ***Training interval***: [0,2] s
- ***Extrapolating interval***: [2,4] s

For the spatial resolution study, we fixed the time resolution at $dt = 0.01s$. For the time resolution study, we fixed the spatial resolution at $dx = 0.04m$.

The results were the following:

TABLE A.1: Spatial results

dx(m)	Error
0.1	$(60 \pm 7)\%$
0.05	$(4.0 \pm 1.3)\%$
0.04	$(2.0 \pm 0.5)\%$
0.01	$(2.8 \pm 0.6)\%$
0.005	$(2.3 \pm 0.8)\%$
0.002	$(1.5 \pm 0.4)\%$
0.001	$(0.9 \pm 0.1)\%$

TABLE A.2: Temporal results

dt(s)	Error
0.1	$(4.7 \pm 2.0)\%$
0.02	$(3.5 \pm 1.1)\%$
0.01	$(2.2 \pm 0.9)\%$
0.002	$(2.4 \pm 0.5)\%$
0.001	$(1.4 \pm 0.4)\%$

TABLE A.3: Results for the study on the impact of resolution in coefficient determination.

Noise Level

The results are shown in Table A.4. The neural network used to obtain the results shown was the following:

- ***Number of layers***: 6
- ***Number of hidden units per layer***:30
- ***Activations***: Sinusoidal in the 1^{st} layer, tanh in the hidden layers, and linear in the output layer.
- ***Optimiser***: Adam
- ***Weight decay***: 10^{-8}
- w_{pde}: $[10^{-6}, 10^{-3}]$
- ***Number of epochs trained***: 15×10^3

The string/data properties were the following:

- $c = 1$
- $v = 0.5$
- ***Number of modes***: 4
- ***Initial condition***: Constant coefficients ($A_n = [0.08, 0.49, 0.35, 0.15, 0.08]$)
- ***dx***: 0.04m

- ***dt***: 0.01s
- ***L***: 1m
- ***Training interval***: [0,2]s
- ***Extrapolating interval***: [2,4]s

Noise(%)	Error
0	$(1.9 \pm 0.1)\%$
1	$(2.8 \pm 0.5)\%$
5	$(2.4 \pm 0.9)\%$
10	$(6.2 \pm 2.3)\%$
25	$(29 \pm 22)\%$
50	$(69 \pm 13)\%$

Table A.4: Error as a function of added noise percentage.

Wave and Horizontal Speeds

The results are shown in Table A.5. The neural network used to obtain the results shown was the following:

- ***Number of layers***: 6
- ***Number of hidden units per layer***:30
- ***Activations***: Sinusoidal in the 1^{st} layer, tanh in the hidden layers, and linear in the output layer.
- ***Optimiser***: Adam
- ***Weight decay***: 10^{-8}
- w_{pde}: $[10^{-6}, 10^{-3}]$
- ***Number of epochs trained***: 15×10^3

Note that when $c = v$, the physical solution makes no sense, therefore we did not study those configurations.

Axiall speed(m/s) / Wave speed(m/s)	0.01	0.1	0.25	0.50	0.75	1
0.25	12 ± 1	11 ± 2	-	31 ± 3	2 ± 0.3	11 ± 1
0.50	15 ± 2	2.9 ± 0.2	2.8 ± 0.1	-	24 ± 3	5.2 ± 0.5
0.75	9.8 ± 1.1	2.8 ± 0.5	2.3 ± 0.5	3.0 ± 0.8	–	$4 - 8 \pm 0.7$
1	9.7 ± 0.9	2.5 ± 0.2	2.1 ± 0.3	3.2 ± 0.4	3.5 ± 0.4	–
1.25	10 ± 1	2.9 ± 0.2	2.1 ± 0.3	1.7 ± 0.2	7.0 ± 0.8	16 ± 2
1.50	31 ± 8	4.0 ± 0.2	2.8 ± 0.2	5.2 ± 0.4	3.2 ± 0.2	8.2 ± 0.4
1.75	13 ± 3	7.8 ± 0.3	3.0 ± 0.2	3.7 ± 0.3	3.6 ± 0.3	7.0 ± 0.4
2	56 ± 10	16 ± 5	3.2 ± 0.2	3.9 ± 0.3	3.6 ± 0.2	7 ± 0.3

TABLE A.5: Coefficient error (%) as a function of wave speed(m/s) and axial speed(m/s).

A.2 Extrapolation (Forward Problem)

All network parameters are the same as the coefficients cases, as we wanted to re-use the determined coefficients. Training domain is $t = [0, 2]$ and extrapolating domain is $t = [2, 4]$ for all results shown.

A.2.1 Resolution

We used the same data, and the same obtained coefficients as in the section A.1. Meaning that the temporal resolution was fixed at $dt = 0.01s$ for the spatial resolution study, and the spatial resolution was fixed at $dt = 0.005m$ for the time resolution study. We got the following results:

dx(m)	Validation Loss (MSE)
0.1	$(4.4e^{-1} \pm 1e^{-1})$
0.05	$(2.1e^{-3} \pm 3e^{-4})$
0.04	$(1.2 \pm 2e^{-1})$
0.01	$(6.4e^{-4} \pm 3e^{-4})$
0.005	$(3.2e^{-4} \pm 9e^{-5})$
0.002	$(1.9e^{-4} \pm 5e^{-5})$
0.001	$(1.5e^{-4} \pm 5e^{-5})$

TABLE A.6: Validation set MSE as a function of spatial resolution for fixed $dt = 0.01s$.

dt(s)	Validation Loss (MSE)
0.1	$(9e^{-3} \pm 2e^{-3})\%$
0.02	$(5e^{-3} \pm 1e^{-3})\%$
0.01	$(4e^{-4} \pm 2e^{-4})\%$
0.002	$(2.4e^{-4} \pm 8e^{-5})\%$
0.001	$(2.0e^{-4} \pm 7e^{-5})\%$

TABLE A.7: Validation set MSE as a function of spatial resolution for fixed $dx = 0.005s$.

TABLE A.8: Validation set error as a function of spatial and temporal resolutions.

A.2.2 Noise

In this section, the same neural network and string parameters as in A.1 were used. The results are shown in Table A.9.

Noise level (%)	Validation Loss (MSE)
0	$(3e^{-4} \pm 7e^{-5})\%$
1	$(4e^{-4} \pm 7e^{-5})\%$
5	$(8e^{-4} \pm 1e^{-4})\%$
10	$(2.5e^{-3} \pm 9e^{-4})\%$
25	$(1.4 \pm 5e^{-1})\%$

Table A.9: Validation Loss (MSE) as a function of the noise level.

Appendix B

Appendix B - Case 2

DeePyMoD [33] has built-in tools to help PINNs converge faster and be more stable when dealing with sparse regression. We used some of them like: forcing sparsity after n-epochs, use PDE-FIND to solve the sparse regression problem instead of least using least squares. We saw no real difference in results when trying different combinations of tools.

B.1 Fixed Point (1D) case

For the 1D Weather problem we used the following feed-forward neural networks:

- ***Width*** : [30,60]
- ***Depth*** : [2,6]
- ***Activation Layers***:
  - Only tanh activations plus linear activation in the last layer
  - Only Sinusoidal activations plus linear in the last layer (Siren architecture [41])
- ***Optimiser***: Adam
- ***Weight Decay***: $[10^{-6}, 10^{-8}]$

We normalised temperature by doing the following:

$$T_{norm} = \frac{T - \mu_T}{T} \tag{B.1}$$

Where μ_T and σ_T are the mean and standard deviation of the Temperature (T).

To build the library of candidate terms we computed the Fourier Transform of the temperature and selected the 10 frequencies with higher coefficients. Then for each

dominant frequency w_i we computed the following terms: $-w_i \cos(w_i t)$, $w_i \sin(w_i t)$, $-w_i *^2 \sin(w_i t)$, and $-w_i^2 \cos(w_i t)$. These terms are just the first and second derivatives of the sinusoidal functions that compose the original signal.

B.2 2D Case

In this section the following neural network parameters were used:

- ***Width*** : [30,60]
- ***Depth*** : [2,6]
- ***Activation Layers***:
  - Only tanh activations plus linear activation in the last layer
  - Or only Sinusoidal activation's plus linear in the last layer (Siren architecture [41])
- ***Optimiser***: Adam
- ***Weight Decay***: $[10^{-6}, 10^{-8}]$

All data normalisation was done just like in Section B.1, plus in this section the coordinates were also normalised with respect to the distance to the centre of the grid.

Bibliography

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf [Cited on page 4.]

[2] W. F. et al., "Pytorch lightning," *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, vol. 3, 2019. [Cited on page 4.]

[3] F. Chollet, *Deep learning with python, second edition*, 2nd ed. Manning Publications, 2022. [Cited on pages xv, 7, 8, 9, 16, and 24.]

[4] (2022, Jun) Neural circuit. [Online]. Available: https://en.wikipedia.org/wiki/Neural_circuit [Cited on pages xv and 10.]

[5] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, p. 115–133, 1943. [Cited on page 10.]

[6] M. Iman, H. Arabnia, and R. Branchinst, "Pathways to artificial general intelligence: A brief overview of developments and ethical issues via artificial intelligence, machine learning, deep learning, and data science," 05 2020. [Cited on pages xv and 22.]

[7] S. Ruder, "An overview of gradient descent optimization algorithms," Jun 2017. [Online]. Available: https://arxiv.org/abs/1609.04747 [Cited on page 23.]

[8] (2022, Jun) Initializing neural networks. [Online]. Available: https://www.deeplearning.ai/ai-notes/initialization/ [Cited on pages 24 and 29.]

[9] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html [Cited on page 30.]

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015. [Online]. Available: https://arxiv.org/abs/1502.01852 [Cited on page 30.]

[11] M. Varadi, S. Anyango, M. Deshpande, S. Nair, C. Natassia, G. Yordanova, D. Yuan, O. Stroe, G. Wood, A. Laydon, and et al., "Alphafold protein structure database: Massively expanding the structural coverage of protein-sequence space with high-accuracy models," *Nucleic Acids Research*, vol. 50, no. D1, 2021. [Cited on page 33.]

[12] (2022, Jun) Energy based model. [Online]. Available: https://en.wikipedia.org/wiki/Energy_based_model [Cited on page 33.]

[13] M. M. Bronstein, J. Bruna, T. Cohen, and P. Velickovic, "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges," *CoRR*, vol. abs/2104.13478, 2021. [Online]. Available: https://arxiv.org/abs/2104.13478 [Cited on page 33.]

[14] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, p. 422–440, 2021. [Cited on pages 34 and 35.]

[15] J. Sirignano and K. Spiliopoulos, "Dgm: A deep learning algorithm for solving partial differential equations," *Journal of Computational Physics*, vol. 375, p. 1339–1364, 2018. [Cited on page 35.]

[16] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations," 2017. [Online]. Available: https://arxiv.org/abs/1711.10561 [Cited on pages 35 and 43.]

[17] S. H. Rudy, S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Data-driven discovery of partial differential equations," 2016. [Online]. Available: https://arxiv.org/abs/1609.06401 [Cited on page 38.]

[18] "Qr decomposition," Aug 2022. [Online]. Available: https://en.wikipedia.org/wiki/QR_decomposition [Cited on pages 40 and 41.]

[19] "Triangular matrix," May 2022. [Online]. Available: https://en.wikipedia.org/wiki/Triangular_matrix#Forward_and_back_substitution [Cited on page 41.]

[20] S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis, "Physics-informed neural networks (pinns) for fluid mechanics: A review," 2021. [Online]. Available: https://arxiv.org/abs/2105.09506 [Cited on page 43.]

[21] S. Cuomo, V. S. di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli, "Scientific machine learning through physics-informed neural networks: Where we are and what's next," 2022. [Online]. Available: https://arxiv.org/abs/2201.05624 [Cited on pages 43, 78, and 97.]

[22] R. Bischof and M. Kraus, "Multi-objective loss balancing for physics-informed deep learning," 2021. [Online]. Available: http://rgdoi.net/10.13140/RG.2.2.20057.24169 [Cited on pages 43 and 57.]

[23] B. Moseley, A. Markham, and T. Nissen-Meyer, "Solving the wave equation with physics-informed deep learning," 2020. [Online]. Available: https://arxiv.org/abs/2006.11894 [Cited on page 43.]

[24] A. D. Jagtap and G. E. Karniadakis, "Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations," *Communications in Computational Physics*, vol. 28, no. 5, pp. 2002–2041, 2020. [Cited on page 43.]

[25] Q. He and A. M. Tartakovsky, "Physics-informed neural network method for forward and backward advection-dispersion equations," *Water Resources Research*, vol. 57, no. 7, jul 2021. [Online]. Available: https://arxiv.org/abs/2012.11658 [Cited on pages 43 and 97.]

[26] C. J. Arthurs and A. P. King, "Active training of physics-informed neural networks to aggregate and interpolate parametric solutions to the navier-stokes equations," *Journal of Computational Physics*, vol. 438, p. 110364, 2021. [Cited on page 44.]

[27] L. Yang, D. Zhang, and G. E. Karniadakis, "Physics-informed generative adversarial networks for stochastic differential equations," *SIAM Journal on Scientific Computing*, vol. 42, no. 1, 2020. [Cited on page 44.]

[28] A. Mathews, M. Francisquez, J. W. Hughes, D. R. Hatch, B. Zhu, and B. N. Rogers, "Uncovering turbulent plasma dynamics via deep learning from partial observations," aug 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2009.05005 [Cited on page 44.]

[29] G. S. Misyris, A. Venzke, and S. Chatzivasileiadis, "Physics-informed neural networks for power systems," 2019. [Online]. Available: https://arxiv.org/abs/1911.03737 [Cited on page 44.]

[30] N. Banichuk, A. Barsuk, J. Jeronen, T. Tuovinen, and P. Neittaanmäki, "Stability of Axially Moving Strings, Beams and Panels," in *Stability of Axially Moving Materials*. Cham: Springer International Publishing, 2020, pp. 397–483. [Online]. Available: https://doi.org/10.1007/978-3-030-23803-2_7 [Cited on pages xv, 46, 47, and 76.]

[31] U. Zwiers and M. Braun, "Modelling and stability analysis of strings in axial motion." 12th IFToMM World Congress, 06 2007. [Cited on pages 47 and 48.]

[32] J. C. Wong, C. Ooi, A. Gupta, and Y.-S. Ong, "Learning in sinusoidal spaces with physics-informed neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2109.09338 [Cited on pages 56 and 57.]

[33] G.-J. Both, G. Vermarien, and R. Kusters, "Sparsely constrained neural networks for model discovery of pdes," 2020. [Online]. Available: https://arxiv.org/abs/2011.04336 [Cited on pages 64, 84, and 105.]

[34] H. P. Langtangen, "Analysis of the difference equations." [Online]. Available: http://hplgit.github.io/num-methods-for-PDEs/doc/pub/wave/html/._wave004.html#wave:pde1:analysis [Cited on page 75.]

[35] B. Lim, S. O. Arik, N. Loeff, and T. Pfister, "Temporal fusion transformers for interpretable multi-horizon time series forecasting," 2019. [Online]. Available: https://arxiv.org/abs/1912.09363 [Cited on page 80.]

[36] D. B. Chirila, "Towards lattice Boltzmann models for climate sciences : The GeLB programming language with applications," Mar. 2018, accepted: 2020-03-09T14:47:44Z Publisher: Universität Bremen. [Online]. Available: https://media.suub.uni-bremen.de/handle/elib/1402 [Cited on pages 81 and 92.]

[37] National Centers for Environmental Prediction, National Weather Service, NOAA, U.S. Department of Commerce, "NCEP GDAS/FNL 0.25 Degree Global Tropospheric Analyses and Forecast Grids," 2015, place: Boulder CO. [Online]. Available: https://doi.org/10.5065/D65Q4T4Z [Cited on page 82.]

[38] B. Moseley, A. Markham, and T. Nissen-Meyer, "Solving the wave equation with physics-informed deep learning," 2020. [Online]. Available: https://arxiv.org/abs/2006.11894 [Cited on page 93.]

[39] K. Kashinath, M. Mustafa, A. Albert, J.-L. Wu, C. Jiang, S. Esmaeilzadeh, K. Azizzadenesheli, R. Wang, A. Chattopadhyay, A. Singh, A. Manepalli, D. Chirila, R. Yu, R. Walters, B. White, H. Xiao, H. A. Tchelepi, P. Marcus, A. Anandkumar, P. Hassanzadeh, and n. Prabhat, "Physics-informed machine learning: case studies for weather and climate modelling," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 379, no. 2194, p. 20200093, Apr. 2021, publisher: Royal Society. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2020.0093 [Cited on pages 96 and 97.]

[40] L. Yang, D. Zhang, and G. E. Karniadakis, "Physics-Informed Generative Adversarial Networks for Stochastic Differential Equations," *SIAM Journal on Scientific Computing*, vol. 42, pp. A292–A317, 2020. [Online]. Available: https://doi.org/10.1137/18M1225409 [Cited on page 97.]

[41] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit neural representations with periodic activation functions," 2020. [Online]. Available: https://arxiv.org/abs/2006.09661 [Cited on pages 105 and 106.]

[42] J. Zegers and H. Van hamme, "Joint speaker segregation and recognition," Sep. 2020. [Online]. Available: http://fs.fish.govt.nz/Page.aspx?pk=7&sc=SUR