Physics Informed Neural Networks for Inverse Advection-Diffusion Problems

Ketil Fagerli Iversen

A Master's Thesis
Assignment for MAB399
Department of Applied Mathematics

November 22, 2021

Acknowledgements

I would like to thank my supervisors Guttorm Alendal and Anna Oleynik for following up on the project, helping me, giving constructive criticism and encouraging me. They hosted weekly on- and off-line meetings which were valuable in keeping everyone updated, and it kept me driven to always have at least something done by the end of each week.

Many thanks to Lu Lu, an assistant professor at the University of Pennsylvania and one of the developers behind DeepXDE [26, 27]. He gave me invaluable insight which lead me to overcoming a hurdle that I had been stuck on for an unspecified number of weeks. Additionally, thanks to Pierre Jacquier for graciously providing his TensorFlow 2.0 conversion code [19] free-of-charge for private and commercial use. This recognition also extends to all the researchers and developers of PINNs who made their codes open to the public. Their work was instrumental in making this thesis possible.

Id also like to acknowledge the assistance of Chris Rackauckas and Magnus Svärd, for taking the time to answer questions and for helping me locate some relevant studies.

Finally, I would also like to extend my deepest gratitude to my family and significant other. They have given me relentless support and kept me motivated throughout the whole process.

Ketil Fagerli Iversen
Bergen, 2021

Abstract

In this study, we will address the problem of localising a source of pollutant given a sparse set of noisy data. By sparse we mean spatially separated time-series measurements in space. To this end, we will be adopting a machine learning algorithm, the Physics Informed Neural Network (PINN) [43]. PINNs are neural networks which aim to be possible alternatives to conventional forward and inverse PDE solvers. PINNs are quite generalisable as the same method can be applied to a wide array of problems. Furthermore, PINNs are not restricted to computational meshes, which gives us the freedom to use sparse and incomplete data-sets.

Consequently, we wished to leverage the generality and power of this novel method on a transport equation which models pollutant transport, the advection-diffusion equation. We will be investigating how we can localise the source by solving the advection-diffusion equation in an inverse manner. It is the inverse problem which will be the main goal of the study, as forward numerical solvers of linear PDEs are already robust, fast and accurate. However, the forward problem will also be looked at as a preliminary to the construction of the PINN. We compute the reference solutions of the transport equation using traditional numerical methods, which gives us training data to be fed to the PINN. Then we employ the PINNs on different test cases involving time-dependent transport equations in one- and two-dimensional spatial domains. Our results show that the PINN is indeed able to solve for these problems and localise sources with high accuracy in certain situations.

Chapter Outline

In chapter 1 we describe the motivation behind this thesis, derive the relevant partial differential equations (PDEs), and define the problem statement that will be worked on. In section 1.6 we briefly describe some related work on similar problems but using different methods. Next, we go through how we solved for the relevant PDEs using numerical methods in chapter 2. Relevant proofs and arguments are provided for well-posedness and stability. This is to show that the PDEs have a unique solution and that the solution we generate, should be unique and accurate. The generated solutions will be used as training data for our PINN, and as reference solutions to estimate errors.

In chapter 3, we describe how the Physics Informed Neural Network (PINN) works in general terms, and how it can solve for a time-dependent PDE in both the forward and inverse sense. We also discuss the pros and cons with the PINNs method over traditional numerical methods. We also go through the underlying mechanics of how a densely connected neural network operates and trains and discuss the choice of the network architecture we will use in chapter 4. Furthermore, in chapter 5, we describe in specifics how we generate the solutions and how we implement the PINNs in code.

Then in chapter 6, we go over the results we achieved by using a basic implementation of PINNs, the Baseline-PINN, and we discuss its limitations. In chapter 7, we discuss some mitigating strategies to alleviate some of the problems we faced. Thereafter, in chapter 8 we implement some of these strategies to gain better results. We also implement the improved PINN on very sparse data to simulate a more realistic scenario. At the end of chapter 8 we propose a method which improves the performance further.

Next, we follow up with some conclusions and potential future work in chapter 9. Finally, we end the thesis with some final comments by the author in 10. Here we discuss how the project could have been handled differently.

The appendices A, B and C include relevant mathematical derivations and analysis which were too lengthy to include in the main chapters of the thesis. Appendices D and E include information on where to find the code that was used and developed for this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

This study will focus on the application of PINNs on a transport equation. Specifically, the advection-diffusion equation (ADE), sometimes referred to as the convection-diffusion equation, describes how a passive scalar, such as a pollutant, is transported and diffused in a given velocity field. The velocity field can, for example, be obtained by solving the Navier-Stokes equation. As the pollutant is assumed to be passive, it does not influence the density of the fluid, and hence does not alter the velocity field. Therefore, the solutions to the velocity field are not coupled to the ADE. And thus the velocity can be solved independently.

The ADE is of interest as it can be used to determine the time evolution of a pollutant in a fluid or gas environment. Specifically we are interested in discovering and localising the source terms contained within the ADE, as they will tell us where the pollutant is generated from. Determining the location of a source can be of great importance. For example in the field of environmental forensics, locating sources of undesirable pollutants (such as crude oils, plastics, hazardous chemicals, greenhouse gasses) in the environment is central. In particular the ACTOM project [2] aims to advance the tools of offshore monitoring of stored CO2. One of the many goals behind ACTOM is to develop tool-kits for detecting subtle leaks of CO2 storage in the environment, something that PINNs may be able to accomplish. This is of particular interest for societal and environmental reasons, as CO2 capture, utilisation and storage (CCUS) is being funded as a part of an international initiative to combat climate change.

For the application of pollutant source identification, it is necessary to solve for the ADE in an inverse manner. The inverse problem makes use of measurements of physical variables to discover the relevant parameters of the governing equations. We will employ PINN method as both a forward and an inverse solver on the ADE to do this. For real-life applications, measurements of the physical variables may be noisy or prohibitively expensive. As such we may be limited in the amount of and resolution of the data. Due to how PINNs operate, we are not restricted to computational grids, and we can deal with very sparse data-sets, as such PINNs may be used for this purpose. The PINN will be discussed in full detail in chapter 3 and 4.

1.2 The Advection-Diffusion Equation (ADE)

The ADE can be derived from a conservation law. A conservation law specifies how some quantity is conserved in some volume, and by which mechanisms the quantity can increase or decrease. In the case of a pollutant, we want that it can only enter or exit the control volume Ω via two mechanisms; sources and boundary fluxes. A conservation law which obeys these attributes can be written as:

$$\frac{\partial}{\partial t}\iint_{\Omega} u d\mathbf{x} = \iint_{\Omega} F(\mathbf{x},t;P)d\mathbf{x} + \int_{\partial\Omega} \mathbf{J}\cdot\mathbf{n}dS. \tag{1.1}$$

Here, $u(\mathbf{x},t)$ is a scalar field which describes the concentration of pollutant at any point in space and time, $F(\mathbf{x},t;P)$ is the source term for which positive values describe a source and negative values describe a sink. The parameter P within the source term, represents variables which describe attributes of the source. $\mathbf{J}$ is the flux, and it accounts for the ways pollutant can enter the control volume through the boundary. $\mathbf{n}$ is the outward unit normal vector on the surface of the control volume.

The flux of pollutant across the surface consists of two parts. For one, pollutant can advect through the surface due to transport imposed by a velocity field $\mathbf{v}(\mathbf{x},t)$. Second, pollutant can diffuse through the surface. In the case of advection, the amount of pollutant that can enter through a unit surface area of the control volume $\partial\Omega$, will depend on the amount of pollutant and the velocity infinitesimally close to the surface. Thus it makes sense to assert that the flux due to advection can be written as the following.

$$\mathbf{J}_{Adv} = -u\mathbf{v}.$$

The choice of sign will come apparent soon.

In the case of the diffusion, we can use Fick's first law of diffusion [9]. It is derived from the fact that pollutant will diffuse from regions with high concentration to regions of low concentration. Fick's law of diffusion postulates that the diffusion flux points in the direction from high concentration to low concentration, and that the rate is directly proportional to the gradient of said concentration,

$$\mathbf{J}_{Diff} = D\nabla u.$$

Here, D is the diffusion rate. It determines how rapidly a peak of concentration, where ∇u is large, disperses and flattens towards an average.

Now let the total flux $\mathbf{J}$ be the sum of these two contributions $\mathbf{J} = \mathbf{J}_{Adv} + \mathbf{J}_{Diff}$. Then by equation (1.1) we arrive at the following conservation law.

$$\iint_{\Omega} \frac{\partial u}{\partial t} d\mathbf{x} = \iint_{\Omega} F(\mathbf{x},t;P)d\mathbf{x} - \int_{\partial\Omega} u\mathbf{v}\cdot\mathbf{n}dS + \int_{\partial\Omega} D\nabla u\cdot\mathbf{n}dS.$$

To explain the choice of signs, note the following. If the velocity at the surface is pointing in the same direction as the outward unit normal vector then $\mathbf{v}\cdot\mathbf{n} \geq 0$. If we assume $D = 0$ and $F = 0$, then all pollutant inside Ω must be moving out from the domain. Therefore, we would expect the concentration inside Ω to be decreasing in this

instance. That is why we have the negative sign for the second term on the right-hand side.

Similarly, note what happens when we only look at the contribution from diffusion, $\mathbf{v} = 0$, and $F = 0$. If there is a higher concentration outside the domain Ω than inside, then we would expect there to be a net inflow to the domain. In this case, the gradient ∇u would be pointing in the same direction as $\mathbf{n}$, which does give a positive value corresponding to increasing concentration inside Ω.

Now, we apply Gauss' theorem on the two surface integrals to transform them into volume integrals. We gain the following:

$$\iint_\Omega \frac{\partial u}{\partial t} d\mathbf{x} = \iint_\Omega F(\mathbf{x},t;P) d\mathbf{x} - \iint_\Omega \nabla \cdot (u\mathbf{v}) d\mathbf{x} + \iint_\Omega \nabla \cdot (D\nabla u) d\mathbf{x}.$$

Since all terms are valid under the same domain, we can transition over to the differential form of the conservation law. This allows us to get rid of the integral terms.

$$\frac{\partial u}{\partial t} + \nabla \cdot (u\mathbf{v}) = \nabla \cdot (D\nabla u) + F(\mathbf{x},t;P) \tag{1.2}$$

Equation (1.2) is the most general version of the ADE. It can be further expanded as,

$$\frac{\partial u}{\partial t} + \mathbf{v} \cdot \nabla u + u\nabla \cdot \mathbf{v} = D\nabla \cdot \nabla u + \nabla D \cdot \nabla u + F(\mathbf{x},t;P). \tag{1.3}$$

There are some simplifications that can be made, depending on the types of problems we are looking at. For example, if the diffusion rate is independent of space, then $\nabla D = \mathbf{0}$ and the second term on the right-hand side disappears. When dealing with incompressible fluids, which is often the case, we can make another simplification. Incompressible fluids has an extra condition imposed on the flow, namely $\nabla \cdot \mathbf{v} = 0$. In this case, the third term on the left-hand side disappears.

For our studies, we assert that $\nabla D = \mathbf{0}$, as we assume that the diffusion rate is only dependent on the fluid and material properties of the pollutant. We will also be assuming an incompressible flow. Thus, we finally arrive at the ADE which will be used in this study as shown in equation (1.4). Note how $\Delta u = \nabla \cdot \nabla u$.

$$\frac{\partial u}{\partial t} + \mathbf{v} \cdot \nabla u = D\Delta u + F(\mathbf{x},t;P), \quad \mathbf{x} \in \Omega, \quad t \geq 0 \tag{1.4}$$

By specifying a velocity field $\mathbf{v}$, a source term F, diffusivity rate D and with a sufficient set of initial and boundary conditions, one should be able to find a unique solution for u. The velocity field can be arbitrarily set by specifying it as a space-time dependent function or by using a velocity-field data-set when solving the PDE numerically in the discretized case.

There are two main types of boundary conditions applied to this problem, they are the Dirichlet and Neumann boundary conditions. The former can be written as follows,

$$u(\mathbf{x},t)|_{\mathbf{x}\in\partial\Omega} = f(\mathbf{x},t), \quad t > 0.$$

This forces u to satisfy some known function f at the boundary. If say $f = 0$, then this would represent walls which absorb pollutant such that the concentration is negligent next to the walls.

The latter condition, the Neumann boundary condition, can be written as,

$$\mathbf{n} \cdot \nabla u|_{\mathbf{x} \in \partial\Omega} = f(\mathbf{x}, t), \quad t > 0.$$

Here $\mathbf{n}$ is the outward unit normal vector to $\partial\Omega$. In this case, it is the directional gradient of u that is forced to satisfy the known function f at the boundary. This on the other hand, describe the fluxes of u at the boundary. If $f = 0$, then the walls would be reflective and pollutant would stay within the domain. If $f > 0$ at any point on the boundary, then there would be a net positive flux of pollutant into the domain there and vice versa for $f < 0$.

For the sake of simplicity, the homogeneous Dirichlet condition was chosen for our case studies. In other words, we chose the Dirichlet boundary condition for when $f = 0$. Though this is the less realistic scenario out of the two options, it does simplify the derivation of well-posedness for the PDE, the numerical computation for generating data in MATLAB [47] and it simplifies the treatment of the PINN. Though it should be noted that PINNs have been shown to be able to deal with Neumann boundary conditions as well [27]. In addition, homogeneous initial condition was chosen, $u(\mathbf{x}, 0) = 0$.

In summary, with these choices, the problem describes how pollutant generated by a source F develops in time under the influence of a velocity field $\mathbf{v}$. It is known that the concentration is zero at the boundaries and that the leak starts at $t = 0$. Thus, the only possible way for pollutant to enter the domain would be from any sources within the domain, specified by F.

1.3 Choice of Source

In our studies, we wanted to look at the advection of some pollutant u being generated from a time-independent leak located at $\mathbf{x}_s$. This can be achieved by letting $F \geq 0$ be a forcing function such that it is non-zero at $\mathbf{x}_s$ and vanishing away from $\mathbf{x}_s$. The simplest way of achieving this would be to set F to the Kronecker-delta function.

$$F(\mathbf{x}) = \delta(\mathbf{x} - \mathbf{x}_s) \tag{1.5}$$

However, due to limitations in how PINNs operate, we could not define F as a delta function. The problem is that the form of F in equation (1.5) is not continuous and differentiable. Therefore, we chose the following N-dimensional forcing function,

$$F(\mathbf{x}; P) = \lambda_{N+1} e^{-S(\mathbf{x} - \mathbf{x}_s)^2}, \tag{1.6}$$

for which,

$$P = [\lambda_1, \ldots, \lambda_{N+1}]^T, \quad \mathbf{x}_s = [\lambda_1, \ldots, \lambda_N]^T.$$

Here $\mathbf{x}_s$ is the location of the source, λ_{N+1} is the strength of the source and S is how concentrated the leak is. P is used as a shorthand term for all of the lambda values.

This choice of $F(\mathbf{x};P)$ will mimic the results given by the delta function (1.5), as the functional values for (1.6) converge rapidly towards zero away from $\mathbf{x}_s$ as long as $S > 0$ is kept large enough. Yet the form in equation (1.6) is still differentiable and continuous such that it can be used in PINNs.

Therefore, when we are referring to the ADE, we mean the ADE of the form:

$$\begin{aligned} &\frac{\partial u}{\partial t} + \mathbf{v}\cdot\nabla u = D\Delta u + F(\mathbf{x};P), \quad \mathbf{x}\in\Omega\subset\mathbb{R}^N, \quad t\in[0,T], \\ &F(\mathbf{x};P) = \lambda_{N+1} e^{-S(\mathbf{x}-\mathbf{x}_s)^2}, \quad P=[\lambda_1,\ldots,\lambda_{N+1}]^T, \quad \mathbf{x}_s=[\lambda_1,\ldots,\lambda_N]^T, \\ &u(\mathbf{x},t) = 0|_{\mathbf{x}\in\partial\Omega}, \quad u(\mathbf{x},0)=0, \\ &\mathbf{v}_i = V_i(t). \end{aligned} \tag{1.7}$$

Figure 1.1 shows an example of interplay between the ADE solution u and the velocity field $\mathbf{v}$. It shows how the the pollutant generated at a source F at the point $\mathbf{x}_s = [-1.45, 0.0]^T$ varies in intensity and how it is advected and diffused downstream. The solution shown here was computed in MATLAB using the methods discussed in chapter 2. The specific velocity-field used was generated by a Navier-Stokes solver provided by *Shibata* [45].

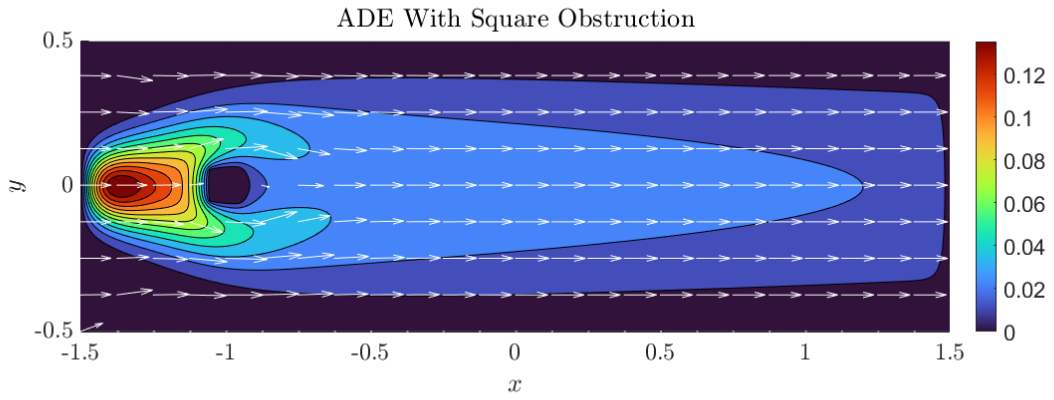

Figure 1.1: ADE problem solved for the case where flow wraps around square obstruction in the domain. Arrows show the direction of the velocity field. The flow is laminar as the Reynolds number was kept low.

1.4 Problem formulation

Now that we have discussed the form of relevant transport equation (1.7), we can now discuss what scenarios we will be solving for. We will be looking at two different

scenarios, the forward and inverse problems.

1.4.1 The Forward Problem

The forward problem consists of solving for u given the ADE and the initial and boundary conditions as specified in equation (1.7). Note that P in equation (1.7) will be fully specified here. To this end, we will employ both traditional numerical methods and the PINN for solving the ADE. The numerical methods will provide us with a reference solution u^* which we can use to compare with the solution u provided by the PINN. The goal is for the PINN to construct an accurate approximation to the reference solution, one which satisfies the ADE in all interior points and satisfies the homogeneous boundary- and initial-conditions. At the end of training, the neural network will be a continuous function $u(\mathbf{x},t)$ which approximately solves for the ADE. Figure 1.2 visualises the idea behind the forward problem.

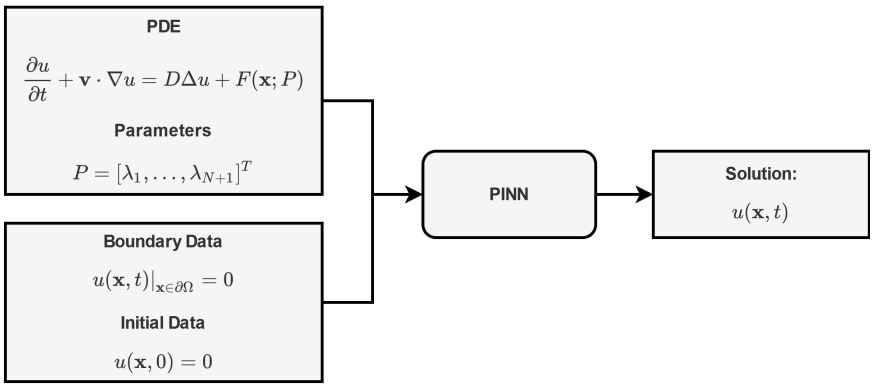

Figure 1.2: Pipeline for the forward problem. The relevant PDE, boundary- and initial-conditions are provided, which the PINN uses to generate a continuous solution.

1.4.2 The Inverse Problem

The inverse problem is what this study mainly be focused on, it consists of discovering unknown coefficients or terms in the ADE by looking at measured samples of the reference solution u^*.

For the scenario we will be studying, the parameters in equation (1.7) such as the velocity field $\mathbf{v}$ and the diffusive rate D will be known, but the location $\mathbf{x}_s = [\lambda_1, \ldots, \lambda_N]^T$ and strength λ_{N+1}, will be unknown. The goal in this case, is for the PINN to return approximations to the unknown parameters P in the source function F as specified by equation (1.6). This will in other words discover where the leak is and how strong it is. The training data-set will be the reference solution u^* generated by traditional numerical methods.

Due to how the inverse PINN works, we also get a continuous extrapolated solution $u(\mathbf{x},t)$ for free. Similar to what we had in the forward case. This may sound redundant as we already have the reference solution u^* as training data. But, for some applications, this data-set might be quite sparse and noisy. Therefore, this extrapolated solution

can come in handy. The PINN is able to use the imposed physics to infer a continuous solution over the entire domain, and patch any holes in the data-set. Figure 1.3 visualises the idea behind the inverse problem.

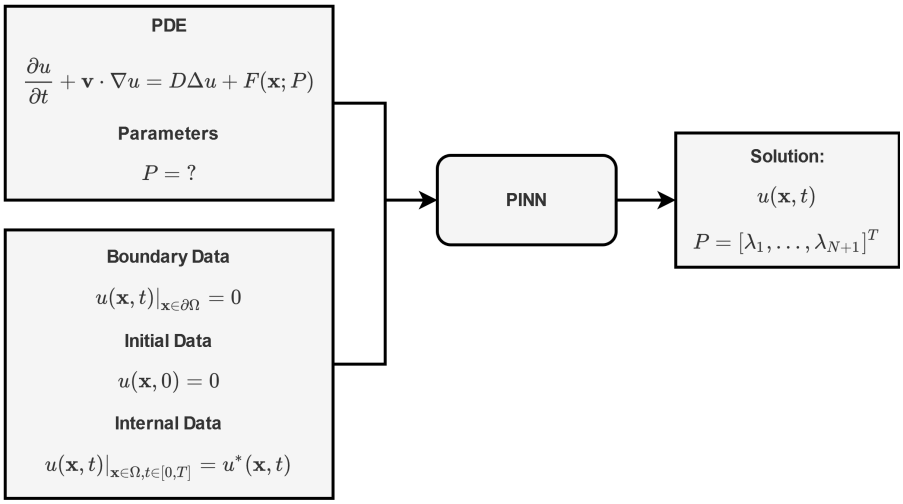

Figure 1.3: Pipeline for the inverse problem. A sparse collection of measurements of the solution u^* are provided, which the PINN uses to generate a continuous solution and discover unknown parameters

1.5 Test-Cases

We wish to explore the capabilities of PINNs to solve both the forward and inverse cases described above. To this end we will be looking at 1D and 2D test cases involving the ADE (1.7). The 1D cases were performed mainly as a first step to ensure our solvers worked, before moving to the more complicated 2D cases. The 1D and 2D forms of the ADE can be derived directly by substituting the component forms of the ∇u and Δu operators in equation (1.7). We will be working in a Cartesian coordinate system.

1.5.1 The 1D Test-Case

When discussing the 1D ADE we will be referring to the following instance,

$$\begin{aligned} &\frac{\partial u}{\partial t} + V_x \frac{\partial u}{\partial x} = D\frac{\partial^2 u}{\partial x^2} + \lambda_2 e^{-S(x-\lambda_1)^2}, \quad x \in \Omega \subset \mathbb{R}, \quad t > 0, \\ &u(x,t) = 0|_{x \in \partial\Omega}, \quad u(x,0) = 0, \\ &V_x = 1, \quad D = \frac{1}{4\pi}, \quad S = 100, \\ &\lambda_1 = -0.8, \quad \lambda_2 = 5, \quad P = [\lambda_1, \lambda_2]^T \\ &\Omega = [-1,1], \quad t \in [0,1]. \end{aligned} \tag{1.8}$$

This is a simple case where all coefficients are constant in space and time. These coefficients were chosen mostly arbitrary, in accordance with creating good looking plots,

and they hold no physical significance.

Whenever we are dealing with an inverse problem, we will leave λ_1 and λ_2 as unknowns, and we will assemble them into a parameter $P = [\lambda_1, \lambda_2]^T$. These unknowns will be left for the PINN to discover using the available data.

1.5.2 The 2D Test-Case

When discussing the 2D ADE we will be referring to,

$$\begin{aligned}
\frac{\partial u}{\partial t} + V_x\frac{\partial u}{\partial x} + V_y\frac{\partial u}{\partial y} &= D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \lambda_3 e^{-S\left((x-\lambda_1)^2+(y-\lambda_2)^2\right)}, \quad \mathbf{x} \in \Omega \subset \mathbb{R}^2, \quad t > 0, \\
u(\mathbf{x},t) &= 0|_{\mathbf{x}\in\partial\Omega}, \quad u(\mathbf{x},0) = 0, \\
V_x &= 2, \quad V_y = 3cos(3\pi t) + 1, \quad D = 0.2, \quad S = 4, \\
\lambda_1 &= -1.4, \quad \lambda_2 = -1.0, \quad \lambda_3 = 1.0, \quad P = [\lambda_1, \lambda_2, \lambda_3]^T \\
\Omega &= [-2,2] \times [-2,2], \quad t \in [0,2].
\end{aligned} \tag{1.9}$$

This is a more complicated problem setup, as the vertical velocity component V_y is now time-dependent. This form was chosen in accordance with creating nice looking plots, but also to create a semi-realistic scenario in which the flow periodically changes direction. Such as the case when dealing with tides.

Whenever we are dealing with an inverse problem, we will leave λ_1, λ_2 and λ_3 as unknowns, and we will assemble them into a parameter $P = [\lambda_1, \lambda_2, \lambda_3]^T$. These unknowns will be left for the PINN to discover using the available data

1.6 Other Related Work

In the following section we discuss related work on the problem of finding source locations by solving the inverse ADE. This is to show the alternate methods which can be used to achieve similar goals.

1.6.1 Gradient Methods for Source Identification Problems

The problem statement in this thesis written by *Banshoya* [4] is similar to ours. *Banshoya* wished to discover the location of a contaminant source given sparse measurements of the ADE. Their test case consists of a 2D square domain with a single point source inside with a uniform constant velocity-field.

They are able to localise the source by setting the problem up as a minimisation problem. In their case, their objective function was of the form,

$$J(u,\mathbf{d}) = \frac{1}{2}\sum_{m=1}^{N_m}\int_0^T \frac{(u(\mathbf{x}_m,t;\mathbf{d}) - u_m^*(t))^2}{\omega_m^2}dt, \tag{1.10}$$

with,

$$F = \sum_{i=1}^{N_s} a_i \delta(\mathbf{x} - \varepsilon_i). \tag{1.11}$$

Here, $\mathbf{d} = \{a_i, \varepsilon_i\}$ is a parameter containing the source strengths and source locations respectively. Unlike our continuous source given by (1.6), they were able to adopt point sources in the form of delta functions (1.11). This gives their method an advantage in that prior knowledge in leak 'size' is not required, as every source is assumed a point source. Though, it should be mentioned that there is nothing about the formulation of PINNs that limits us from letting S in equation (1.7) be yet another unknown to be solved for.

$u_m^*(t)$ corresponds to continuous time-series measurement data of the ADE model, while u is the numerical solution. Thus, minimisation of their objective function corresponds to finding what numerical solution u with parameters $\mathbf{d}$ which most closely corresponds to what the available measurements u^* are. To this end, they used a conjugate gradient method to minimise the objective function J. Once the minimum is found, the parameters $\mathbf{d}$ can be interpreted as the source intensities and locations.

Measurements were gathered from $N_m = 30$ locations in the domain, to give 30 different time-series plots. We will explore how the PINN performs on a very similar data-set in chapter 8. The main difference in the problem setup, is that they use a constant uniform velocity-field for most of their test cases. This was due to how their PDE solver was not able to handle time-dependent velocities. Thus their method has not been shown to work when the velocities can vary over time. Our test-cases in 2D however had a time-dependent velocity-field.

Their method was successful in localising the sources, however only for when the initial guesses where set near the true answer. An initial guess too far away would lead to diverging solutions.

1.6.2 Inverse Source Problem in a 2D Linear Evolution Transport Equation: Detection of Pollution Source

In the study by *Hamdi* [17], they aimed to find the intensity and location of a contaminant source using data generated by the ADE. Although, their ADE model includes a reaction term which we have not dealt with. In their test case, the contaminant is assumed to be generated from a location inside a rectangular domain. The rectangular domain is bounded by four boundaries: an inlet boundary, an outlet boundary and two parallel walls. Intuitively, this situation can be thought of as a rectangular tube where the fluid flows from one end to the other.

What makes this scenario unique is that the measurement stations are set only at the inlet and outlet boundaries, but not in the interior. In addition, the source intensity varies with time. Remarkably, they are able to discover not only the time evolution of the

source intensity, but also the location of the source with good accuracy. This was done with only four measurement stations on the inlet and outlet boundaries. Their results were also stable with respect to noisy data.

This method has the clear advantage of being able to localise the source with measurements taken only at the boundaries. In addition, it has the advantage of discovering the time history of source intensity, which our implementation of PINNs cannot do in a simple-to-implement manner. We suspect that the PINN may be able to do predictions on time-dependent sources if we assume that our source term $F(\mathbf{x},t;P)$ can be expressed as a linear combination of simple functional terms such as a sum of sinusoidal waves. However, we have not attempted this, so we cannot confidently confirm this claim.

Again their method was not shown to be able to deal with time-dependent velocities, though this case was supposed to simulate the stream down a section of a rectangular tube, where the mean velocity can, in a simple case, be assumed to be uniform and constant in time.

1.6.3 Point Source Identification in Non-Linear Advection-Diffusion-Reaction Systems

In the study by *Mamonov and Tsai* [28], the problem statement is again similar to our own. *Mamonov and Tsai* are dealing with a rectangular domain, and the source is located somewhere within this domain. Here they use a similar adjoint gradient method as described in section 1.6.1. In this study however, they showed that their method is robust with respect to obstacles in the domain, noisy data, multiple sources, and time-dependent source intensities. However, it is unknown if their velocity-fields were constant, or space-time dependent.

Mamonov and Tsai ran multiple test cases, varying the number of measurements, in a range of $N_m = 20$ down to $N_m = 3$. They found that their method required a low number of data to get good accuracy. Their study also includes a complimentary algorithm which can determine the optimal locations to place new measurement stations in case more refined results are desirable. The great advantage to their model is that their algorithm includes a heat-map of source location probabilities, where the maximum of this heat-map constitutes the predicted source location. This heat-map gives a measure on how certain the prediction is, and it gives alternative possible locations.

Chapter 2

Numerical Solutions of PDE

Differential equations are one of the cornerstones of modern physics. A vast set of real-life physical problems can be accurately modelled by rephrasing the physical setup into the mathematical language of differential equations and then constructing their solutions. For any given differential equation, one would ideally construct an analytical closed form solution. Analytical solutions give us not just arbitrary precision, but also valuable insight into how solutions behave with changes of parameters. Simple ODEs and PDEs can in some instances be solved analytically by using clever mathematical tricks. Unfortunately, this is not possible for most instances. For any given differential equation, one cannot be guaranteed that an analytical closed form solution exists. And in many real-world cases the differential equations are sufficiently complex such that only approximate solutions can be obtained. And this can only be achieved using various numerical methods. Though numerical methods can in principle reach arbitrary accuracy, they may be prohibitively expensive to compute. Regardless, numerical methods are still an essential tool for whenever one wants to solve and study complex real-life problems via the use of differential equations.

Before arriving at the method of PINNs, we will discuss how we generated reference solutions u^* for the ADE in 1D and 2D as given by equation (1.8) and (1.9) respectively. As we wished to have the maximum amount of control of the problem setup, we generated the data ourselves by designing our own numerical solver. To this end, we need to discuss the following: Well-posedness, the discretization scheme and stability. Well-posedness asks if the problem posed by the PDE is solvable. The discretization scheme gives us a way to solve the PDEs numerically, and stability ensures that the numerical scheme used to solve the problem converges on the real solution.

2.1 Well-Posedness

Well-posedness of PDEs determine whether solutions actually exist, are unique and if the solutions change continuously with changes in initial- and boundary-conditions. Well-posedness also show that solutions do not explode to infinite values within finite time. The proof of well-posedness, for the general n-dimensional ADE considered in equation (1.7), is quite lengthy. Therefore we place the proof in appendix A. Regardless, well-posedness means that the solution of our PDE satisfies the following

inequality.

$$
\begin{aligned}
&||u(\cdot,t)|| \leq Ke^{\alpha t}\left(||u(\cdot,0)|| + \int_0^t (||F(\cdot,\tau)|| + |g(\tau)|^2)d\tau\right) \\
&||u(\cdot,t)|| = \iint_\Omega u(\mathbf{x},t)^2 d\mathbf{x}
\end{aligned}
\tag{2.1}
$$

Where $K \in \mathbb{R}$ is a real number independent of F, g and $u(\cdot,0)$.

This inequality prohibits any growth to infinity in finite time. However, it does allow for exponentially growing solutions with an arbitrary growth constant $\alpha \in \mathbb{R}$. This condition also disallows the use of any unbounded initial data $||u(\cdot,0)||$ and disallows unbounded growth in the source function $||F(\cdot,\tau)||$ and boundary data $|g(\tau)|$.

Working under the assumption that the ADE as described in equation (1.7) is well-posed and has a unique solution, as proven in appendix A, we can continue to the next step.

2.2 Solving PDEs With Finite Differences

A common way to compute the solution of PDEs is the method of finite differences. The main idea behind finite differences is to approximate the derivative terms of a PDE by a finite difference approximation. Thus we reduce derivative evaluations into function evaluations.

We computed the solution of our PDEs using finite differences via semi-discretization. As such, we approximate the true solution $u^*(\mathbf{x},t)$ by grid-functions $u_i(t)$ which approximates the solution u^* at the grid-points $\mathbf{x}_i$ at time t.

We construct the grid-function such that it takes the form $\mathbf{u}(t) = [u_1(t),\ldots,u_N(t)]^T$. In other words, it takes the form of a column-vector for which each component i corresponds to the approximate solution $u_i(t)$ at the grid-point $\mathbf{x}_i$. The number N here denotes the number of grid-points that constitute the grid which is superimposed on the domain Ω.

This process then transforms the PDE into a large system of coupled ODEs, in which each ODE corresponds to solving for each component $u_i(t)$. This ODE system can be solved by using for example the fourth order Runge-Kutta time integrator. In appendix B we derive the central difference approximations for the first and second order derivatives on a general function $f(x)$. Then we show their order of accuracy, and then apply them on the ADE (1.7). We then end up with finite difference equations for our test cases as shown in equation (B.5) and (B.10).

2.2.1 Dealing With Boundary- and Initial-Conditions

As we have discussed in section 1.2, we settled on the homogeneous Dirichlet boundary- and initial-conditions. For the finite difference scheme to obey the boundary conditions, we need to consider how we deal with the boundary points.

In the 1D case as shown in appendix B, we divided the x-axis into N_x points such that $x_i \in [a,b]$, $i = 1,\ldots,N_x$. Due to how the difference operators are constructed, we need some way to deal with the fact that the points x_{-1} and x_{N_x+1} are unavailable to us. The simplest option is to set the solution u_1 and u_{N_x} at the boundary points x_1 and x_{N_x} to obey the boundary data. This is a method called injection.

Consequently, it becomes unnecessary to solve for these boundary points using finite differences, because they are already known. The finite difference scheme can be used as-is for all interior points, excluding boundary points.

The same idea holds for the 2D case. In all computations, the injection method was used. As such, any components of $\mathbf{u}(t)$ along the boundary grid-points were always set to obey the homogeneous Dirichlet boundary condition. When it comes to obeying the homogeneous initial condition, we simply initialise the grid-function $\mathbf{u}(0)$ with zeroes.

2.2.2 Finite Difference Method Applied to the N-Dimensional Linear ADE

Now, let the solution $u^*(\mathbf{x},t)$ of the N-dimensional ADE (1.7) be approximated with a grid-function of the form $\mathbf{u}(t) = [u_1(t),\ldots,u_N(t)]^T$. Also let the source $F(\mathbf{x})$ be approximated as a grid-function of the form $\mathbf{F} = [F_1,\ldots,F_N]^T$. Thanks to the method of finite differences as discussed in appendix B, we can approximate the derivative terms ∇u and Δu using finite difference operators $\mathcal{D}_1(\mathbf{u})$ and $\mathcal{D}_2(\mathbf{u})$ respectively. Therefore, we can construct a finite difference scheme of the form,

$$\frac{\partial \mathbf{u}}{\partial t} = D\mathcal{D}_2(\mathbf{u}) - \mathbf{v}(t)\cdot\mathcal{D}_1(\mathbf{u}) + \mathbf{F}. \tag{2.2}$$

Each term on the right-hand side of (2.2) contains only the components of the grid-functions $\mathbf{u}$ and $\mathbf{F}$. In addition, $\mathcal{D}_1(\cdot)$ and $\mathcal{D}_2(\cdot)$ are linear operators, therefore we can rephrase the difference scheme in (2.2) using a matrix-vector product,

$$\frac{\partial \mathbf{u}}{\partial t} = A(t)\mathbf{u}(t) + \mathbf{F}. \tag{2.3}$$

Here A is a sparse, diagonally-banded and possibly time-dependent matrix. In the 1D case this matrix will have three bands, corresponding to the three grid-points used in the derivative stencil, while in the 2D case, this matrix will have five bands corresponding to its five-pointed stencil.

We could solve the difference scheme by writing the scheme explicitly as in equation (B.5) and (B.10). However, the matrix-vector form as shown in equation (2.3) unifies

the time marching scheme between the 1D and 2D case. Hence, the same solver can be used without any modification between the two cases. In addition, it allows us to more readily show stability.

What remains now is to show how we transfer the functions $u(\mathbf{x},t)$ and $F(\mathbf{x})$ into the corresponding grid-functions $\mathbf{u}$, $\mathbf{F}$ and the matrix A for the two cases. And we will show that indeed the form in equation (2.3) can be constructed equivalent to (B.5) and (B.10).

Figure 2.1 and 2.2 show how we can construct the grid-function $\mathbf{u}(t)$ in the 1D and 2D instances respectively.

2.3 The Finite Difference Method Applied to the 1D ADE

Let the domain Ω, consisting of the x-axis be divided into N_x grid-points. Then let the solution $u(x,t)$ for the 1D ADE as described in equation (1.8) be approximated by a grid-function $\mathbf{u}(t)$ of N_x elements such that $\mathbf{u}(t) = [0, u_2(t), \ldots, u_{N_x-1}(t), 0]^T$. Here the elements u_i correspond to each grid-point i on the x-axis, see figure 2.1. Also let the continuous source $F(x)$ be approximated by a grid-function $\mathbf{F} = [0, F_2, \ldots, F_{N_x-1}, 0]^T$, such that it obeys the boundary conditions. Then, the finite difference scheme in as derived in equation (B.5), can be rewritten in the following form:

$$\frac{\partial}{\partial t}\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_i \\ \vdots \\ u_{N_x-1} \\ u_{N_x} \end{bmatrix} = \begin{bmatrix} 0 & & & & & & \\ \alpha_x & \beta_x & \gamma_x & \ldots & & & \\ & & \ddots & & & & \\ & \ldots & \alpha_x & \beta_x & \gamma_x & \ldots & \\ & & & & \ddots & & \\ & & & \ldots & \alpha_x & \beta_x & \gamma_x \\ & & & & & & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_i \\ \vdots \\ u_{N_x-1} \\ u_{N_x} \end{bmatrix} + \begin{bmatrix} 0 \\ F_2 \\ \vdots \\ F_i \\ \vdots \\ F_{N_x-1} \\ 0 \end{bmatrix}, \tag{2.4}$$

where

$$\alpha_x = \frac{D}{\Delta x^2} + \frac{V_x}{2\Delta x}, \quad \beta_x = -\frac{2D}{\Delta x^2}, \quad \gamma_x = \frac{D}{\Delta x^2} - \frac{V_x}{2\Delta x}.$$

The matrix here may be readily derived, by looking at equation (B.5). This is easy to see, as equation (B.5) essentially tells us the following:

$$\frac{\partial u_i}{\partial t} = \alpha_x u_{i-1} + \beta_x u_i + \gamma_x u_{i+1} + F_i, \quad i = 2, \ldots, N_x - 1,$$

which is indeed equivalent to the matrix-vector form in (2.4). Notice how this formulation of the numerical scheme automatically obeys the homogeneous boundary conditions, as u_1 and u_{N_x} are never updated. This is because their corresponding derivatives remain at zero. Also note that none of the terms in A are time-dependent. This is because our 1D ADE as described in equation (1.8) has no time-dependent coefficients. As such, we have simplified the difference scheme (B.5) into the desired form of (2.3).

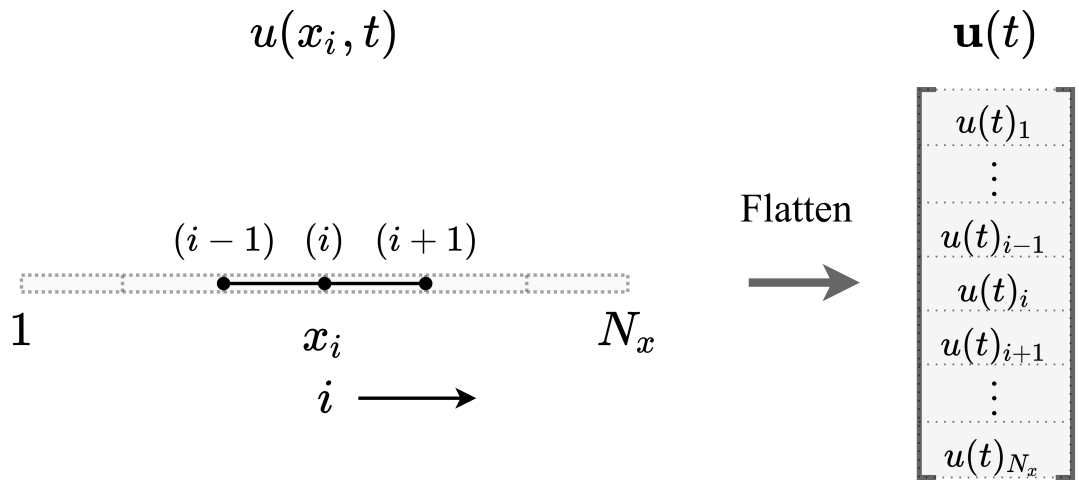

Figure 2.1: The one-dimensional line is divided into N_x grid-points. We assemble the solution $u(x_i,t)$ into a vector according to the scheme shown in the figure. The difference scheme stencil is superimposed on the grid.

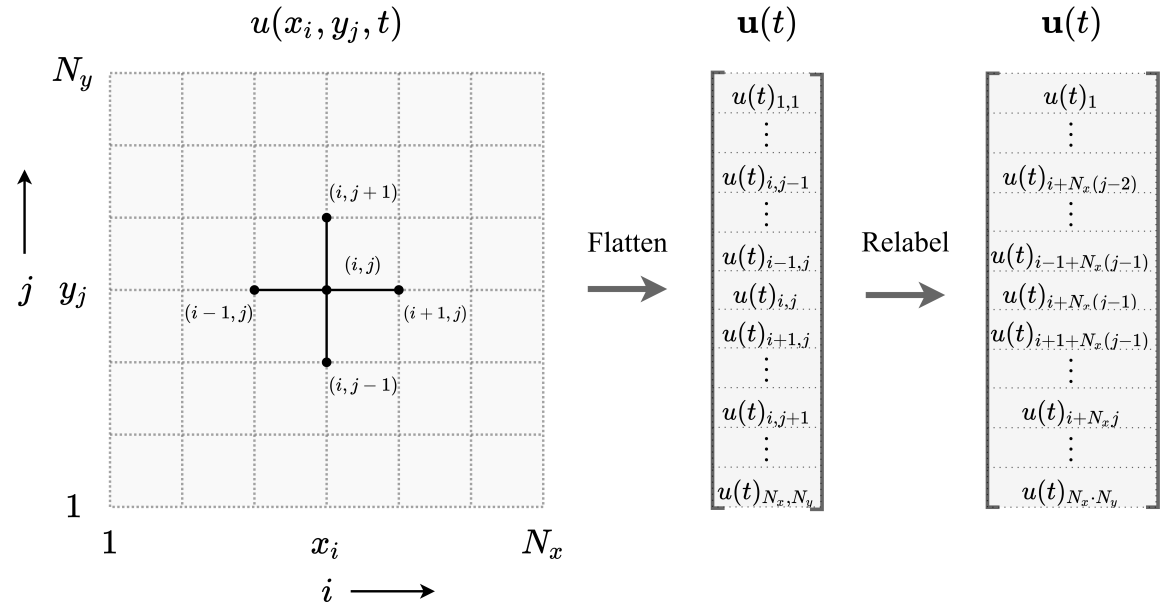

Figure 2.2: The two-dimensional plane is divided into $N_x \cdot N_y$ grid-points. We assemble the solution $u(x_i,y_i,t)$ into a vector according to the scheme shown in the figure. The difference scheme stencil is superimposed on the grid.

2.4 The Finite Difference Method Applied to the 2D ADE

The matrix A in the 2D case is a bit more involved to derive. One has to flatten the two dimensional function $u(x_i, y_j, t) = u(t)_{i,j}$ into a vector $\mathbf{u}(t) = [u_1, \ldots, u_{N_x \cdot N_y}]^T$ with $N_x \cdot N_y$ elements. We found that the following procedure,

$$u(t)_{i,j} = \mathbf{u}_{i+N_x(j-1)},$$

was sufficient to do this. But, this has to be done in such a way that A obeys the difference scheme and each of the four homogeneous boundary conditions. The full derivation of A is shown in appendix B. Regardless, we were able to simplify the difference scheme into the desired form of equation (2.3). In this case however, $A(t)$ was time dependent. This was because the 2D ADE as described in equation (1.9) contains time-dependent velocity components.

2.5 The Fourth Order Runge-Kutta Method

We decided to use the explicit fourth order Runge-Kutta (RK4) time integrator for our time marching scheme. This is because the RK4 method has a larger and more favourable domain of stability in comparison to other many other explicit schemes such as the forward Euler scheme. This becomes helpful when ensuring stability without requiring very small time-steps dt. Also, since RK4 explicit, it is trivial to implement.

The fourth order Runge-Kutta method solves for ODE systems of the form,

$$\frac{\partial \mathbf{u}}{\partial t} = f(t, \mathbf{u}), \tag{2.5}$$

where f describes the discretization scheme on the right-hand side of the finite difference equation (2.3). Let $\mathbf{u}^n = \mathbf{u}(t^n)$ be the solution of the grid-function $\mathbf{u}$ at time-step t^n. Marching forward in time is then accomplished by computing $\mathbf{u}^{n+1}$ at the next time-step $t^{n+1} = t^n + \Delta t$. To do so, we compute (2.6) and then step forward by equation (2.7).

$$\begin{aligned} \mathbf{k}_1 &= f(t^n, \mathbf{u}^n) \\ \mathbf{k}_2 &= f(t^n + \Delta t/2, \mathbf{u}^n + \Delta t \mathbf{k}_1/2) \\ \mathbf{k}_3 &= f(t^n + \Delta t/2, \mathbf{u}^n + \Delta t \mathbf{k}_2/2) \\ \mathbf{k}_4 &= f(t^n + \Delta t, \mathbf{u}^n + \Delta t \mathbf{k}_3) \end{aligned} \tag{2.6}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \frac{1}{6}\Delta t(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \tag{2.7}$$

We know from before that we can rephrase the finite difference schemes in matrix-vector form. As such, when solving for the ADE with RK4, we set f by,

$$f(t, \mathbf{u}) = A(t)\mathbf{u}(t) + \mathbf{F}. \tag{2.8}$$

To ensure that the solutions generated by the matrix-vector form (2.3) indeed give us the same results as the explicit finite difference equations (B.5) and (B.10), we computed

the solutions of both. We found that we do get the same solution down to machine precision.

2.6 Consistency, Stability and Convergence of the Numerical Scheme

To know that our numerical scheme actually approximates the well-posed PDE, we need to show that the scheme is consistent and stable. If this is the case, then it can be shown by the Lax-Richtmyer equivalence theorem [6], that the scheme will converge to the true solution of the PDE as the step sizes go to zero.

Consistency requires that the finite difference equations approximates the differential equation as the step sizes go to zero. As we have shown in appendix B, the first order central difference scheme approximates $f'(x)$ with order $\mathcal{O}(h^2)$ and the second order central difference scheme approximates $f''(x)$ with order $\mathcal{O}(h^2)$ as well. Clearly, as the step-size $h \to 0$ we have that the error shrinks as h^2 and the finite differences converge towards the actual derivatives contained in the PDE. Ergo, the numerical schemes (B.5) and (B.10) approximate their respective ADEs and thus satisfy the consistency condition.

As for stability, one would typically design the scheme using SBP-SAT operators. Then the scheme would satisfy stability by construction. Or one could apply the energy method to the semi-discrete scheme to get an energy estimate, which mimics the well-posedness results. If it can be shown that the energy does not blow up in finite time, we can assert that the scheme is stable.

However, to leverage the SBP-SAT properties we would have to rephrase our numerical schemes. This would become further complicated when promoting the system from 1D to 2D. To avoid the tedious mathematical legwork, we instead decided to use a different method. The method we chose let us leverage the power of computers to establish the stability of our schemes.

In short, the method is based on computing the eigenvalues of $A(t)$ in equation (2.3), and plotting them against the stability regions of the time-marching scheme. We go through the the stability and convergence analysis in further detail in appendix C. We conclude that our difference schemes for the 1D and 2D ADE are both stable.

Furthermore, in appendix C we show the convergence rate for both schemes as the number of grid-points increases. We find that the schemes converge with approximate order $\mathcal{O}(dx^2)$. To balance performance and accuracy, we make a compromise and choose a number of grid-points such that the relative L^2-norm error as given by equation (5.1) was around 0.1%.

Chapter 3

Physics Informed Neural Networks (PINNs)

Now that we have discussed how we arrive at the solution for the ADE via traditional numerical methods, we can now discuss the method that will be the focus in this thesis, the physics informed neural network.

3.1 Intro to PINNs

Neural networks require a large amount of training data to perform well. For example, the image classification problem can only really be solved effectively by gathering sufficiently many labelled image examples for the neural network to train on in supervised learning. With today's powerful and efficient computers, training the neural network does not constitute a huge hurdle anymore, even with large data-sets and large networks.

However, for many scientific and engineering applications, there can be a great difficulty and expense in gathering such great amount of data. Especially for large scale systems such as the weather or the oceans or in systems in which it is challenging to place measurement devices such as underground or inside living organisms. In those cases, data may be quite sparse in both space and time and data may be corrupted by noise. Hence, it may be prohibitively expensive to gather dense and accurate data for the neural network to sufficiently train on.

Being motivated by solving PDEs in the forward and inverse sense in regimes where data is sparse, there has been research put into combining the generality and power of machine-learning techniques with physics and domain knowledge. To this end, physics informed neural networks (PINNs) were developed. The method of PINNs is based on embedding prior knowledge about the physical systems into a neural network in such a way to constrict the space of possible solutions to only encompass the types of solutions which we are interested in. As such, we magnify the information content that the model obtains from the available data. This is done through a regulariser in the form of the governing equations which enforces the physical laws into the solution. As such, the model is not only much more information efficient with the data that it is dealt, but also able to infer the solution by using the physical laws. This is especially useful in the instances where data acquisition is expensive.

In short, the PINN uses the universal function approximator capability of neural networks [18], to express solutions of time-dependent linear and non-linear PDEs of the form,

$$\begin{aligned} \frac{\partial u}{\partial t} + \mathcal{N}[u;P] = 0, \quad \mathbf{x} \in \Omega \subset \mathbb{R}^d, \quad t \in [0,T], \\ u(\mathbf{x},t) = g(\mathbf{x},t), \quad \mathbf{x} \in \partial\Omega, \quad t \in [0,T], \\ u(\mathbf{x},0) = h(\mathbf{x}), \quad \mathbf{x} \in \Omega, \quad t = 0. \end{aligned} \tag{3.1}$$

Here $\mathcal{N}[u;P]$ is a differential operator acting on u with a set of parameters P. In the general sense P are PDE parameters, but we will treat them as in equation (1.7).

Following the work of *Raissi et al.* [43], we approximate the solution u with a neural network $u(\mathbf{x},t;\theta) = NN(\mathbf{x},t;\theta)$ parametrized by θ and with input variables $(\mathbf{x},t)$. The form of this parametrized function is directly related to the architecture of the network and the activation functions used, as will be discussed in further detail in chapter 4.

As with any regular neural network, one trains it by feeding it some training data, computing the fitness metric via an objective loss function $\mathcal{L}$ and then adjusting the network to improve the fit. The loss tells us how well the neural network performs given some objective. During training, the parameters θ are updated with the goal of minimising this loss function. For PINNs, the loss is typically divided into two parts: The residual loss $\mathcal{L}_f$ and the supervised loss $\mathcal{L}_u$.

So, if we want our network to approximate some differential equation, we require that for every point $\{\mathbf{x}_i,t_i\}_{i=1}^{N_f} \in \Omega \times [0,T]$ that we feed to the network, it should approximately satisfy the differential equation we wish to study, i.e, equation (3.1). To accomplish this task, one defines the residual of the PDE, f as the right-hand side in (3.1),

$$f(\mathbf{x},t;\theta,P) = \frac{\partial}{\partial t}u(\mathbf{x},t;\theta) + \mathcal{N}[u(\mathbf{x},t;\theta);P]. \tag{3.2}$$

The partial derivatives $\partial u/\partial t$ and $\mathcal{N}[u;P]$ of the network u are computed using AutoDiff methods [5] discussed in chapter 4. If the network $u(\mathbf{x},t;\theta)$ exactly satisfies the PDE in equation (3.1), then we should expect this residual f to be zero. Note that any deviation of u from the true solution u^* would lead the residual to be nonzero, as it would no longer exactly solve for the PDE. We want the residual to be zero or a positive value as small as possible. Therefore, we can define the residual part of the objective loss function $\mathcal{L}_f$ as the mean squared error of the residual f on a set of points $\{\mathbf{x}_i,t_i\}_{i=1}^{N_f} \in \Omega \times [0,T]$. These points will henceforth be called collocation points. They represent any space-time coordinates that may be sampled from the space-time domain of interest $\Omega \times [0,T]$ without the need of knowing the reference solution u^*. This residual loss $\mathcal{L}_f$ is given as,

$$\mathcal{L}_f(\theta, P) = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(\mathbf{x}_i, t_i; \theta, P)|^2, \quad \{\mathbf{x}_i, t_i\}_{i=1}^{N_f} \in \Omega \times [0, T]. \tag{3.3}$$

Thus, if the network $u(\mathbf{x}, t; \theta)$ approximately satisfies the PDE in (3.1), then we expect $\mathcal{L}_f$ to be a positive value near zero. Equation (3.3) is then a regularizer which reinforces the desired physics into the network. Note that the residual loss $\mathcal{L}_f(\theta, P)$ is a function of both θ and P here, but in the forward case, we have that P is fully specified. Therefore P is only relevant for the inverse case.

Note that since the derivatives $\partial u/\partial t$ and $\mathcal{N}[u; P]$ are computed using AutoDiff on the network, we are no longer tied to a computational grid. We do not require a sufficiently dense grid to compute these derivatives, as we did with finite differences. As such, we have in a sense uncoupled the computational mesh from the PDE. Why is that? The derivatives are instead dependent on the computational structure of the network. This gives us the freedom to use any space-time coordinates $\{\mathbf{x}_i, t_i\}_{i=1}^{N_f}$ that we want for training and evaluation.

As in the case of any PDE solver, the enforcement of the governing equations is not enough to guarantee a unique solution. We need to also specify initial and boundary conditions. To enforce the initial and boundary conditions into the network, we define an additional supervised loss function $\mathcal{L}_u$ as the mean squared error between the true solution of (3.1), $u^*(\mathbf{x}, t)$, and the neural network prediction $u(\mathbf{x}, t, \theta)$ on a set of points $\{\mathbf{x}_i, t_i, u_i^*\}_{i=1}^{N_u} \in \Omega \times [0, T]$ in which the true solution u_i^* is known. These points will hereafter be called data-points, as they include the sampled data of the true solution u^* at the coordinate $(\mathbf{x}, t)$. The supervised loss takes the form,

$$\mathcal{L}_u(\theta) = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(\mathbf{x}_i, t_i; \theta) - u_i^*|^2, \quad \{\mathbf{x}_i, t_i, u_i^*\}_{i=1}^{N_u} \in \Omega \times [0, T]. \tag{3.4}$$

Note that the data-points require that we know the reference solution u^* at certain points. This is not a problem in either the forward or inverse cases. In the forward case, we simply provide the data that is given by the initial- and boundary-conditions, while in the inverse case we can provide any data that is available inside the space-time domain, but also including the initial- and boundary-conditions.

Typically, when training PINNs, we have seen that it is common to select collocation points via a Latin-Hyper-cube sampling method. This ensures that the points are uniformly distributed in the space-time domain. But how do we determine the number of points we need to ensure that the PINN sufficiently approximates the PDE? *Chris Rackauckas* touched on this point in his talk [10]. One can show that if the collocation- and data-points are of a sufficient number and if they sufficiently cover the domain, then minimising equation (3.3) is equivalent to solving the PDE (3.1). This proof was provided by *Shin et al.* [46] for elliptic and parabolic PDE. But this proof is concerned with the limit as the number of points increases to infinity. Other studies conducted by *Mishra and Molinaro* [34, 35] showed that the generalisation error can be bounded by

the training loss and the number of training points.

The goal for the PINN algorithm then, is to find the parameters θ (and P in the inverse case) which minimises the following objective loss function,

$$\mathcal{L}(\theta, P) = f_m \mathcal{L}_f(\theta, P) + u_m \mathcal{L}_u(\theta). \tag{3.5}$$

Here $f_m, u_m \in \mathbb{R}$ are scalars which scale the contributions from the residual part (3.3) and the supervised part (3.4).

This is where our formulation for PINNs differ from the one provided in the original paper by *Raissi et al.* [43]. We will refer to their formulation as the Baseline-PINN. In their studies, they had set $f_m = u_m = 1$, and thus they assume that both parts should contribute equally to training. However, as *Raissi* suggests in his presentation, proper scaling between these two parameters is not fully understood, and there is a lot of open questions left to be answered as to how to best scale these terms for optimal performance [41].

The optimisation problem can be stated as,

$$[\theta^*, (P^*)]^T = \min_{\theta,(P)} \mathcal{L}(\theta, P). \tag{3.6}$$

Without going into too much detail about training, the minimisation procedure can be accomplished using back-propagation and gradient descent as will be discussed in chapter 4. The basic idea is to differentiate the loss $\mathcal{L}$ with respect to θ (and P in the inverse case). And then use these gradients to update the parameters iteratively. Note that the parameters θ are shared between $u(\mathbf{x}, t; \theta)$ and $\mathcal{L}(\theta, P)$, thus when the optimal θ which minimise $\mathcal{L}$ are found, then we also have the optimal solution u which most closely solve for the problem as stated in equation (3.1). This general formulation of PINNs can be used both as a forward solver of the PDE, but also as a data-driven solver in use for inverse modelling and discovery of unknown parameters P.

Figure 3.1 shows the basic pipeline for how a PINN operates for both forward inverse problems. Note that for any given problem, only one optimiser will be acting on the PINN.

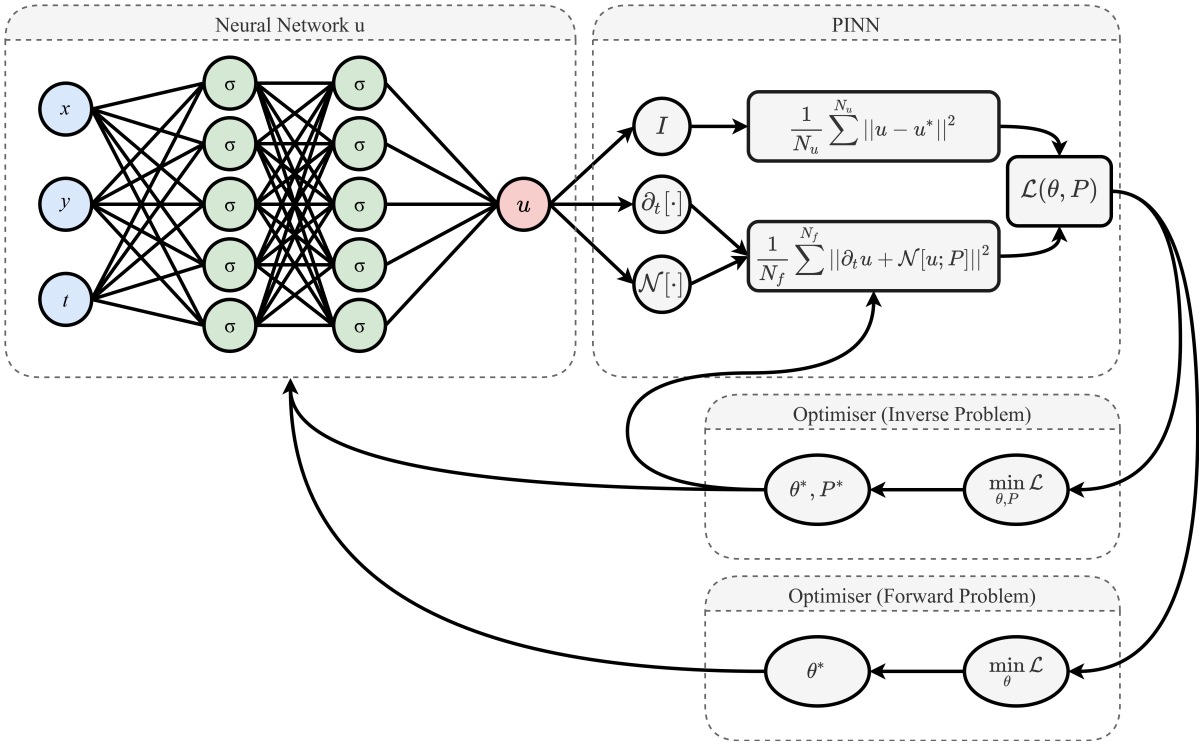

Figure 3.1: The basic pipeline of how the PINN operates divided into three parts: the network u, the physics informed part and the optimiser. Both the forward and inverse optimisers are shown here, but only one will be active for any given problem..

Note that all the constraints put on the neural network via $\mathcal{L}$ in equation (3.5) are only soft constraints [24]. As such, it is unlikely for the solution generated by the PINN, $u(\mathbf{x},t;\theta)$, to exactly solve for the conditions in equation (3.1). This is because it is generally not feasible to find the global optimum of the optimisation problem posed by equation (3.6). And even if we could find the global optimum, the data may be noisy or limited, which would add a bound on how accurate the model can get. This is opposed to how some numerical solvers may impose strong constraints such that certain conditions are always exactly satisfied, as for example, the method of injection for the finite difference scheme.

This does not inhibit us from imposing strong constraints on our PINN however. One can for example define the network u in the following manner:

$$u(\mathbf{x},t;\theta) = t \cdot NN(\mathbf{x},t;\theta).$$

As such, the homogeneous initial condition would be satisfied by construction. We have seen this method used on occasion, but this is not in accordance with the Baseline-PINN as provided by *Raissi et al.* [43], so we will not be using it in our studies.

3.2 Forward vs. Inverse Formulation of PINNs

The forward and inverse PINNs wish to solve for two very different problems, but there are only very slight differences between how the two formulations are constructed. Re-

call that the forward problem is concerned with solving for u when the PDE and initial/boundary conditions are fully specified. The inverse problem however, is concerned with using data measured from u^* and then estimating the unknown parameters P in the PDE. The main difference comes from how the supervised loss $\mathcal{L}_u$ in equation (3.4) is defined, and which parameters we set as we trainable variables.

3.2.1 The Forward PINN

The forward problem wishes to solve for u given only the PDE and the initial- and boundary conditions. We assume here that the parameters P are known, such that we are solving for a well-posed problem. By equation (3.1), we have that the true solution u^* is known only on the spatial boundaries $\mathbf{x} \in \partial\Omega$ as $u^*(\mathbf{x},t) = g(\mathbf{x},t)$ and at $t = 0$ as $u^*(\mathbf{x},0) = h(\mathbf{x})$. Hence, we define the residual loss $\mathcal{L}_u$ by equation (3.9).

$$\mathcal{L}_{u_b} = \frac{1}{N_b}\sum_{i=1}^{N_b} |u(\mathbf{x}_i,t_i;\theta) - g(\mathbf{x}_i,t_i)|^2, \quad \{\mathbf{x}_i,t_i\}_{i=1}^{N_b} \in \partial\Omega \times [0,T], \tag{3.7}$$

$$\mathcal{L}_{u_0} = \frac{1}{N_0}\sum_{i=1}^{N_0} |u(\mathbf{x}_i,0;\theta) - h(\mathbf{x}_i)|^2, \quad \{\mathbf{x}_i\}_{i=1}^{N_0} \in \Omega, \tag{3.8}$$

$$\mathcal{L}_u = \mathcal{L}_{u_b} + \mathcal{L}_{u_0}. \tag{3.9}$$

Here, $\mathcal{L}_b$ is the loss with regards to how well the network u satisfy the boundary conditions, while $\mathcal{L}_0$ is the loss regards to how well u satisfy the initial conditions. Note, that any deviation of u from u^* on the points specified by N_0 and N_b, will lead to $\mathcal{L}_u$ being non-zero. Thus, if the network approximately satisfies the boundary and initial conditions as defined by (3.9), then we know $\mathcal{L}_u$ will be a positive value near zero.

The residual loss $\mathcal{L}_f$ is defined precisely as in equation (3.3). Then all that is left to do is to define the total loss $\mathcal{L}$ by equation (3.5) and then find the minima by gradient descent with respect to the trainable variables θ. As the parameters P are known, we do not need to minimise with respect to them. The gradient descent algorithm for the forward PINN takes the form given in equation (3.10). Where α is the learning rate and $\mathbb{G}[\cdot]$ is some operator specified by the optimiser, which acts on $\nabla_\theta\mathcal{L}$ to create a gradient step. How a network trains and the use of gradient descent will be explored further in chapter 4. Therefore the gradient descent algorithm will take the form,

$$\theta_{n+1} = \theta_n + \alpha \cdot \mathbb{G}[\nabla_\theta\mathcal{L}]. \tag{3.10}$$

At the end of training, one should end up with a network $u(\mathbf{x},t;\theta)$ which acts as an approximator to the PDE in (3.1).

3.2.2 The Inverse PINN

In the inverse case however, we have access to the functional value of u^* in the interior as well as on the boundaries. But we now have unknown parameters P in our PDE which remain to be solved for. Here we assume that the data u_i^* has been sampled from

the boundaries, the initial condition and inside the domain. Here, we can define the supervised loss as,

$$\mathcal{L}_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(\mathbf{x}_i, t_i; \theta) - u_i^*|^2, \quad \{\mathbf{x}_i, t_i\}_{i=1}^{N_u} \in \Omega \times [0,T]. \tag{3.11}$$

Notice that the form in equation (3.11) differs from the one in equation (3.9) in that the data-points may lie inside the domain, and not just on the boundaries. This means we are using all the information available to us to solve the problem. Additionally, since P is now unknown, the residual loss (3.3) becomes a function of P,

$$\mathcal{L}_f(\theta, P) = \frac{1}{N_f}\sum_{i=1}^{N_f} |f(\mathbf{x}_i, t_i; \theta, P)|^2 = \frac{1}{N_f}\sum_{i=1}^{N_f} |\frac{\partial}{\partial t}u(\mathbf{x}_i, t_i; \theta) + \mathcal{N}[u(\mathbf{x}_i, t_i; \theta); P]|^2, \tag{3.12}$$
$$\{\mathbf{x}_i, t_i\}_{i=1}^{N_f} \in \Omega \times [0,T].$$

With equation (3.11) and (3.12), we can assemble the total loss $\mathcal{L}$ with equation (3.5).

To accompany the change of objective from the forward to inverse case, the optimiser needs to be modified slightly. This modification involves appending the unknown parameters P in 3.1 as extra trainable parameters to the network. In other words if θ is some data-structure such that $\theta = [\theta_1, \ldots, \theta_N]^T$, then we can append the unknown parameters P to the end, $\theta \to [\theta, P]^T$. The optimiser then computes the gradient of the loss $\mathcal{L}$ with respect to not only the parameters of the network θ, but also the unknown variables P. This differentiation process is done using back-propagation. Similarly to how the parameters θ are adjusted during training to optimise the loss, so are the unknowns P. This is how the inverse formulation can discover unknown parameters. As a result, they are treated as trainable parameters of the network and are thus subject to the optimisation algorithm used for training. More specifically, the gradient descent optimiser will take the form,

$$\begin{bmatrix}\theta \\ P\end{bmatrix}_{n+1} = \begin{bmatrix}\theta \\ P\end{bmatrix}_{n} + \alpha \cdot \mathbb{G}\begin{bmatrix}\nabla_\theta \mathcal{L} \\ \nabla_P \mathcal{L}\end{bmatrix}. \tag{3.13}$$

Once the loss is minimised, one will obtain guesses for the unknown parameters P. Additionally, similar to the forward case, one will end up with a network $u(\mathbf{x}, t; \theta)$ as an approximate solution to the PDE.

As mentioned in section 1.4 this may sound redundant as we already had the true solution u^* as training data to train the inverse PINN. However, this feature of inverse PINNs becomes very useful in reconstructing a solution when only very sparse and noisy data are available for training. As $u(\mathbf{x}, t; \theta)$ is a continuous function defined for any arbitrary input $(\mathbf{x}, t)$, we are free to reconstruct the solution on any grid or structure that we want.

3.3 PINNs Applied to the ADE

The PDE we will be working with is described in its general form in equation (1.7). By following the summary in section 3.1, we need to construct a residual f of the form given in equation (3.2). Assuming that all derivative terms of u in equation (1.7) can be computed at each collocation-point $\{\mathbf{x}_i, t_i\}_{i=1}^{N_f}$, then we can define the following residual:

$$f(\mathbf{x}_i, t_i; \theta, P) = \left(\frac{\partial u}{\partial t}\right)_i + \mathbf{v}_i \cdot \big(\nabla u\big)_i - D\big(\Delta u\big)_i - F(\mathbf{x}_i, t_i; P).$$

Here the subscript i denotes the functional values for each term at each of the collocation-points. Thus, the physics informed, or residual part of the loss $\mathcal{L}_f$ for the ADE takes the form,

$$\mathcal{L}_f(\theta, P) = \frac{1}{N_f}\sum_{i=1}^{N_f}\left|\left(\frac{\partial u}{\partial t}\right)_i + \mathbf{v}_i \cdot \big(\nabla u\big)_i - D\big(\Delta u\big)_i - F(\mathbf{x}_i, t_i; P)\right|^2, \quad \{\mathbf{x}_i, t_i\}_{i=1}^{N_f} \in \Omega \times [0, T]. \tag{3.14}$$

We also define the supervised loss $\mathcal{L}_u$ on the data-points $\{\mathbf{x}_i, t_i, u_i^*\}_{i=1}^{N_u}$ as,

$$\mathcal{L}_u = \frac{1}{N_u}\sum_{i=1}^{N_u}|u(\mathbf{x}_i, t_i; \theta) - u_i^*|^2, \quad \{\mathbf{x}_i, t_i, u_i^*\}_{i=1}^{N_u} \in \Omega \times [0, T]. \tag{3.15}$$

Hence, the complete objective loss function (3.5) of the ADE described in equation (1.7) becomes,

$$\mathcal{L}(\theta, P) = \frac{f_m}{N_f}\sum_{i=1}^{N_f}\left|\left(\frac{\partial u}{\partial t}\right)_i + \mathbf{v}_i \cdot \big(\nabla u\big)_i - D\big(\Delta u\big)_i - F(\mathbf{x}_i, t_i; P)\right|^2 + \frac{u_m}{N_u}\sum_{i=1}^{N_u}|u(\mathbf{x}_i, t_i; \theta) - u_i^*|^2. \tag{3.16}$$

The loss $\mathcal{L}$ is a function of the parameters of the network θ (and P in the inverse case). It is now clear that if the neural network u satisfies the ADE and the data u^* as specified by 1.7 closely, then the loss should be small.

Also, recall the scaling factors u_m and f_m. They will be manipulated later in section 7.1 to elevate the performance from the Baseline-PINN.

3.4 Advantages and Disadvantages of the PINN as a PDE Solver

PINNs have a lot of advantages over conventional methods for solving partial differential equations. Methods such as the finite difference, finite volume and finite element schemes, which are commonly used as numerical solvers for real-life applications. One of the greatest advantages of PINNs is that it is agnostic to grids and boundary treatment schemes. In other words, one is not required to grid or tessellate the domain into cells or nodes as a part of discretization. In fact, for PINNs, discretization is not needed at all to compute derivatives. Differentiation is instead done by AutoDiff [5] on the network. This means that one can use an arbitrary set of points inside of, and on the surface of, the domain. Thus, one skips over any complicated meshing step.

For simple domain geometries, meshing is not usually a big deal for traditional methods. And with a rectangular domain, one can often get away with a regular rectangular grid. However, for some applications, such as those of the fluid dynamics of an aerofoil, proper meshing becomes important. However, there is nothing about the formulation of PINNs as described in section 3.1 that limits us from solving PDEs on irregular domains. We were not able to explicitly showcase this advantage, due to time-limitations. Instead, we can refer to examples of PINNs running on complicated domain geometries. *Raissi et al.* for example, showed the inverse PINN solving for the Navier-Stokes equations on irregular shapes, and even on the CT-scan of a brain aneurysm [41, 42]. To this end there has been efforts to increase the robustness of PINNs on complex domains, such as the development of XPINNs [20] or DeepXDE [26, 27].

In addition, for inverse problems, the neural network is agnostic to initial and boundary conditions. So, there is no issue if the boundary and initial conditions for the problem are unknown [42], uniqueness is ensured by the reference data u^* that is available [41]. As such, the PINN can solve for ill-posed problems as well.

Due to grid-agnosticism, PINNs are able to handle sparse data-sets. This is because the enforced physics allow the PINNs to patch holes of missing data in the domain. One can therefore train the PINN on a coarse set of points, then later evaluate the problem for a more refined set. This specific aspect was exploited in our studies, particularly in the sparse 2D inverse case. We will show that the PINN can extrapolate the solution for the whole domain, despite only being trained on a handful of points in space.

The methodology behinds PINNs is also generalisable. For example, if one wanted to solve for the non-linear Burger's equation instead of the ADE, one would simply have to replace $\mathbf{v}_i$ in equation (3.16) with u_i. And then retrain the model. The non-linearity of the Burger's equation does not seem to impose any problems either as *Raissi et al.* showed that the Baseline-PINN could handle the shock that develops very well [43]. This is unlike the finite differences formulation of the Burger's equation. In which one must take into careful consideration which spatial discretization one applies. The central difference will lead to oscillatory solutions as the shock forms, whereas an upwind scheme will not. But even the upwind scheme will not be completely accurate, as it will

act as if there was a damping term present. As *Raissi et al.* suggests, one only needs to ensure well-posedness of the PDE, for there to be an optimal solution for the PINN [43].

One of the main drawbacks however, is that convergence of PINNs is not fully understood [41], but there have been studies conducted which aim to find a bound on the generalisation error for PINNs on various PDE for both forward and inverse problems [34, 35]. But it is still an open question which hyper-parameters should be used for accurate solutions. Also, as *Markidis* [29] suggests, PINNs are still too slow and inefficient to be of any practical use as forward solvers. In large part, this is due to the fact that PINNs require a lot of data. Despite the fact that the physics informed regularisation is supposed to make the neural network more data efficient, they still need a large amount of data to achieve good accuracy. But, as *Markidis* showed, the use of PINNs can be further optimised via the use of transfer learning [29]. This makes it computationally faster to solve for several similar problem cases without having to redo training from an initialised neural network each time. PINNs are therefore not as useful as forward solvers in their current state. However, the use of PINNs seems much more promising for use with inverse problems as the inverse problem can be solved with the same computational complexity and efficiency as the forward problem [34].

Another drawback is that despite PINNs being generalisable, they may not always succeed in finding a good solution. This issue seems to be heavily problem specific, meaning that PINNs may work well for some problems, but poorly for others. A lot of contemporary research on PINNs is focused on mitigating these issues [24, 31, 49, 50]. We will mention some of these strategies later on in section 7.1, 7.2 and 7.3.

Chapter 4

PINNs Background

4.1 Feed Forward Neural Networks

To understand PINNs in proper detail, we need to understand the simple feed forward neural network (FNN). The Baseline-PINN as formulated by *Raissi et al.* in [43] is an FNN. The basic structure of the FNN we will be working with is shown in figure 4.1.

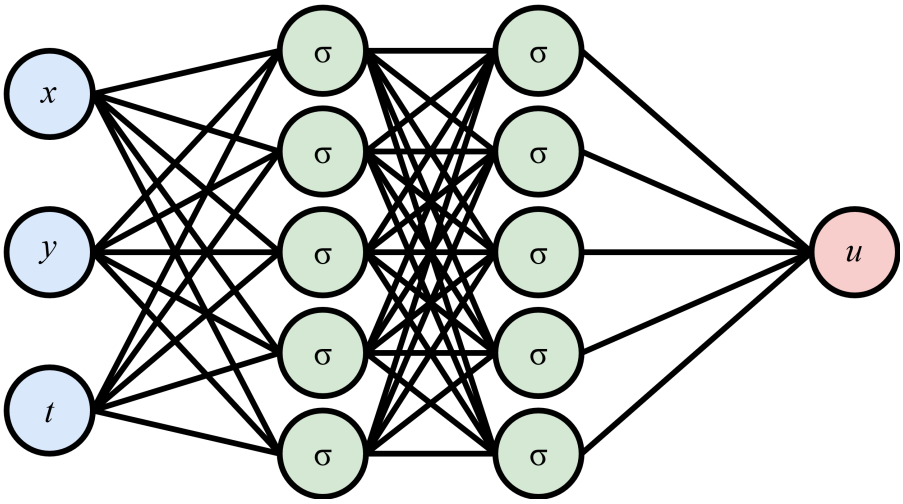

Figure 4.1: Basic structure of a feed forward neural network. Input nodes coloured in blue, hidden nodes coloured in green and the output node coloured in red. Information propagates from left to right. This example depicts an FNN with three input nodes, two hidden layers with 5 nodes each, and one output node.

The network consists of nodes, organised into groups of layers, which are interconnected by a web of weights $W^l_{i,j} \in \mathbb{R}$. A node computes a weighted sum of its inputs and returns some output based on the activation function σ. An activation function is simply a predefined function $\mathbf{X} \to \sigma(\mathbf{X}) \in \mathbb{R}^d$ that determines the output given some input $\mathbf{X} \in \mathbb{R}^d, d = N+1$. Here we assume that the inputs of the network have the same form as the collocation-points in section 3.1 $\mathbf{X}_i = [\mathbf{x}_i, t_i]^T$.

The activation function is there to introduce non-linearity to the network, as it allows the FNN to estimate non-linear functions. It can be shown that without the non-linearity introduced by the activation functions, a multilayered network would act equivalent to

a single layered network [14]. This is due the fact that a set of successive linear transformations is still linear. Therefore, activation functions are essential for constructing multi-layered networks.

The weights, are as the name suggests, variables which determine the weighting of connections between each node pair in adjacent layers. Biases $b_j^l \in \mathbb{R}$ are introduced in each node as constants that offset each activation function by some amount. In the general case for when the PINN u is a densely connected FNN, we can express the network using nested functions and matrix multiplication.

$$u(\mathbf{X}) = \sigma^L((W^L)^T\sigma^{L-1}((W^{L-1})^T\sigma^{L-2}(\dots\sigma^0((W^0)^T\mathbf{X}+b^0)+b^{L-2})+b^{L-1})+b^L) \tag{4.1}$$

Where $L+1$ is the total number of layers of the network, including the input and output layers. The superscript $l \in \{0,\dots,L\}$ denotes the specific layer we are referencing. The number N_l describes how many nodes are contained in layer l. In this convention, we start counting from zero such that $l = 0$ describes the input layer. As such, $L = 3$ in figure 4.1. The weight matrix $W^l = (w_{i,j})^l$ is an $N_{l-1} \times N_l$ matrix containing the weights between layer $l-1$ and l, where index (i,j) corresponds to the weight connecting node i of layer $l-1$ with node j of layer l. Also b^l is a $N_l \times 1$ vector containing each of the biases in layer l. Finally, σ^l describe the activation function used in layer l.

4.2 Evaluation and Training

Training then consists of tuning the W^l and b^l parameters in a certain way such that the output $u(\mathbf{X})$ matches some given desired objective given by the loss $\mathcal{L}$. This parameter tuning happens by the method of gradient descent (4.2) on the objective function $\mathcal{L}$. The loss that we are concerned with is described in its general form in equation (3.5). Training is thus an optimisation problem in which one must find what combination of weights and biases minimises the loss $\mathcal{L}(\theta)$. Here $\theta = \{W^l, b^l\}_{l=0}^L$ are all the weights and biases from each layer of the network.

$$\theta_{n+1} = \theta_n - \alpha \cdot \mathbb{G}[\nabla_\theta \mathcal{L}] \tag{4.2}$$

Training consists of two parts, a forward pass and a backwards pass. In the forward pass, the network u and loss $\mathcal{L}$ are evaluated. The network is evaluated by feeding training data of the form $\{\mathbf{X}_i\}_{i=1}^{N_f}$ to the input nodes. While the loss is evaluated by constructing $\mathcal{L}$ as given in equation (3.5). To evaluate the supervised part we require data of the form $\{\mathbf{X}_i, u_i^*\}_{i=1}^{N_u}$. While for the physics informed part, we assume that the derivatives of u with respect to the input $\mathbf{X}$, can be computed using AutoDiff [5].

Once the gradients are computed, then they can be fed into an optimiser which will update the weights and biases via gradient descent. This updating scheme is completed several times by iterating over the data-set $\{\mathbf{X}_i\}_{i=1}^{N_f}$ and $\{\mathbf{X}_i, u_i^*\}_{i=1}^{N_u}$ until one converges

on an optimal solution for θ. Each iteration over the entire data-set constitutes one epoch of training.

The Forward Pass

The formulation of the FNN in equation (4.1) may be cumbersome to work with directly as it requires us to write the network in full as a set of function compositions. We can instead describe the network in an iterative fashion. Let z^l be the weighted sum of inputs into each of the nodes in layer l. z^l takes the form of a $N_l \times 1$ vector. Also define the output (activation) of each node from layer l as a^l. Then we get the set of equations shown in (4.3). The left column shows the matrix-vector notation, while the right column shows the element notation for clarity. Here $a^{-1} = \mathbf{X}$ is the input fed into the network,

$$\begin{aligned} a^l &= \sigma^l(z^l), & a^l_j &= \sigma^l(z^l_j), & j &= 1,\dots,N_l \\ z^l &= (W^l)^T a^{l-1} + b^l, & z^l_j &= \sum_i W^l_{ij} \cdot a^{l-1}_i + b^l_j, & j &= 1,\dots,N_l. \end{aligned} \tag{4.3}$$

The forward pass propagates the input data $\mathbf{X}$ through each layer of the network until we reach the final layer $a^L = u$. In other words, the forward pass is evaluating the network $u(\mathbf{X})$. For computational purposes, this formulation is advantageous. In addition to computing $u(\mathbf{X})$, we occupy each of the intermediate values a^l and z^l numerical values. These intermediate values are stored to be used in the backward pass. The forward pass can be expressed as the set of iterative computations shown in (4.4). Note that this iterative formulation is equivalent to (4.1). In fact, this order of operations is exactly how the FNN is evaluated in code.

$$\begin{aligned} a^{-1} &= \mathbf{X} \\ z^0 &= (W^0)^T a^{-1} + b^0 \\ a^0 &= \sigma^0(z^0) \\ z^1 &= (W^1)^T a^0 + b^1 \\ &\vdots \\ a^{L-2} &= \sigma^{L-2}(z^{L-2}) \\ z^{L-1} &= (W^{L-1})^T a^{L-2} + b^{L-1} \\ a^{L-1} &= \sigma^{L-1}(z^{L-1}) \\ z^L &= (W^L)^T a^{L-1} + b^L \\ a^L &= \sigma^L(z^L) = u \end{aligned} \tag{4.4}$$

The Backward Pass

As mentioned earlier, the goal of the backward pass is to compute the gradient of $\mathcal{L}$ with respect to θ. The issue is that we require some algorithm to compute the derivatives of the loss function $\mathcal{L}$ with respect to the weights W^l and biases b^l. There are a

couple options for computing derivatives. Finite differences or symbolic differentiation for example.

Finite differences could in theory be used here, however it has several disadvantages. The two main disadvantages are that finite differences require two or more distinct points which lie relatively close to each other in parameter space in order to compute an approximate derivative. This is the reason why finite differences require us to define a regular computational grid. Additionally, this method becomes computationally inefficient for large networks. This is as each parameter increases the dimensionality of the problem. However, finite differences are sometimes used for validating results [14]. In addition, for PINNs we will are required to compute the gradient of u with respect to the inputs $\mathbf{X}$ as well. If we were using finite differences, then PINNs would no longer be agnostic to grids. And finally, finite differences suffer from computational instabilities as step sizes become small.

Another option is symbolic differentiation. This is feasible for small networks with very few nodes and layers. However, the functional complexity of the derivatives grows exponentially as one adds more layers and nodes [5], and thus is not ideal for our use. Neural networks may consist of thousands or even millions of nodes. And each node of will have several weights associated with them. Symbolic differentiation will simply not be practical in such cases either. Another option however, AutoDiff, is meant to sidestep these issues.

Automatic differentiation or AutoDiff (AD) [5] is the backbone of how neural networks are trained. The main idea comes from applying the chain rule of calculus successively through each layer in the network. In particular, it is a subclass of AutoDiff, the back-propagation algorithm, that is most commonly used to train neural networks. This is because the back-propagation algorithm is well suited for functions with many independent variables [5], as is the case for neural networks with thousands of weights and biases to be optimised. AD gives numerical values for derivatives similar to finite differences. This is opposed to giving functional expressions for the derivatives, such as what symbolic differentiation does. One might assume then, that AD is some approximate numerical method. However, it is not, as it will give the exact numerical values for the derivatives down to machine precision. As is the case to evaluation of symbolic differentiation. Back-propagation lets us compute the gradients accurately and efficiently.

Here we define the order of operations which back-propagation performs to compute the desired gradients. Define $\partial\mathcal{L}/\partial z_j^l = \delta_j^l$, as the rate of change of the loss $\mathcal{L}$ with respect to the inputs of node j in layer l. Now, notice from (4.3) that for $l < L$, we have that z_k^{l+1} is dependent on all the inputs from z_j^l. Therefore, we must sum the contribution from all these connections. By the chain-rule we get the following:

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l}.$$

From (4.3), we have,

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = W_{jk}^{l+1} \cdot \frac{\partial a_j^l}{\partial z_j^l} = W_{jk}^{l+1} \cdot (\sigma^l)'(z_j^l).$$

Hence, we arrive at,

$$\delta_j^l = \sum_k \delta_k^{l+1} \cdot W_{jk}^{l+1} \cdot (\sigma^l)'(z_j^l).$$

This formulation is cumbersome to work with, as such we convert it to vector form,

$$\delta^l = W^{l+1}\delta^{l+1} \odot (\sigma^l)'(z^l), \quad \delta^l \in [N_l, 1]. \tag{4.5}$$

Where $\odot$ denotes element-wise multiplication. We thus have a way of computing δ^l for all $l < L$. Now, we can derive $\partial \mathcal{L}/\partial W_{ij}^l$. By the chain-rule we have,

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^l} = \frac{\partial \mathcal{L}}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial W_{ij}^l}.$$

By looking at (4.3) we get,

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^l} = \delta_j^l \cdot a_i^{l-1}.$$

We transfer this into vector form to gain,

$$\frac{\partial \mathcal{L}}{\partial W^l} = a^{l-1}(\delta^l)^T, \quad \frac{\partial \mathcal{L}}{\partial W^l} \in [N_{l-1}, N_l]. \tag{4.6}$$

Finally, we need to compute $\partial \mathcal{L}/\partial b_j^l$. By looking at (4.3), it can be shown that,

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \frac{\partial \mathcal{L}}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l.$$

Hence, in vector form we gain,

$$\frac{\partial \mathcal{L}}{\partial b^l} = \delta^l, \quad \frac{\partial \mathcal{L}}{\partial b^l} \in [N_l, 1]. \tag{4.7}$$

We now have a way of computing the derivative of $\mathcal{L}$ with respect to the weights W^l and biases b^l in an iterative fashion by starting from layer $l+1$ and computing derivatives backwards to layer l. For $l = L$ however, we need to compute $\delta^L = \partial \mathcal{L}/\partial z^L$ directly. This is because δ^L is problem specific, as it entirely depends on the form of $\mathcal{L}$. This is immediately evident by looking at (3.5). Different PDEs will give different forms of the loss. Let us assume we know how to compute δ^L for now, such that we can continue.

When defining the loss $\mathcal{L}$ for PINNs, we will also need to compute $\partial u/\partial \mathbf{X}$. In other words, differentiate the network with respect to our input variables. We can replicate what we did for $\mathcal{L}$ by defining a new variable. Let μ^l be the rate of change of u with respect to the inputs z^l to layer l.

$$\mu^l = \frac{\partial u}{\partial z^l}.$$

It is trivial to show by (4.3) that $\mu^L = (\sigma^L)'(z^L)$. Thus, similarly to δ^l we get,

$$\mu^l = W^{l+1}\mu^{l+1} \odot (\sigma^l)'(z^l), \quad \mu^l \in [N_l, 1]. \tag{4.8}$$

We can then compute μ^0 by working backwards from μ^L. This is done via the iterative formula (4.8). Finally, we need to compute $\partial z^0/\partial a^{-1}$. From (4.3) we get,

$$\frac{\partial z^0}{\partial a^{-1}} = (W^0)^T.$$

Finally, we have that,

$$\frac{\partial u}{\partial \mathbf{X}} = \frac{\partial u}{\partial z^0}\frac{\partial z^0}{\partial a^{-1}} = (W^0)^T\mu^0. \tag{4.9}$$

As such, we have shown how we can compute $\partial u/\partial \mathbf{X}$. Similarly $\partial^2 u/\partial \mathbf{X}^2$ can be obtained. These derivatives can then be used to construct $\mathcal{L}$ according to the PDE we want to solve.

Once δ^L has been computed by $\delta^L = \partial \mathcal{L}/\partial z^L$, then all gradients of $\mathcal{L}$ follow by the iterative equations in (4.10).

$$\begin{aligned} \delta^l &= W^{l+1}\delta^{l+1} \odot (\sigma^L)'(z^l), \quad l \in L-1,\dots,1 \\ \frac{\partial \mathcal{L}}{\partial W^l} &= a^{l-1}(\delta^l)^T, \quad l \in L-1,\dots,1 \\ \frac{\partial \mathcal{L}}{\partial b^l} &= \delta^l, \quad l \in L-1,\dots,1 \end{aligned} \tag{4.10}$$

This is how back-propagation operates. First, intermediate values are occupied by numerical values in a forward pass (4.4). Then the loss can be numerically evaluated as (3.5). Then gradients of this loss can be computed with the backward pass as described in (4.10).

Notice that the only derivatives we are required to know symbolically are $\partial \mathcal{L}/\partial z^L = \delta^L$ and $(\sigma^l)'(\cdot)$. As $\sigma^l(\cdot)$ are predetermined activation functions, we can assume that we also know their derivatives. On the other hand, $\partial \mathcal{L}/\partial z^L = \delta^L$ can vary between problems. Also notice, that all of the intermediate values such as a^l and z^l which are needed to compute the iterative formula in (4.10), are already known to us. This is because we already occupied these variables with numerical values in the forward step 4.2. This is the main computational advantage of the back-propagation algorithm. The gradients for each layer can be derived iteratively, using values which have already been used in the forward pass.

For multiple training data, we can simply add the contribution from every point,

$$\begin{aligned} \delta^l &= \frac{1}{N}\sum_{i=1}^{N}\left(\delta^l\right)_i, \\ \frac{\partial \mathcal{L}}{\partial W^l} &= \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial \mathcal{L}}{\partial W^l}\right)_i, \\ \frac{\partial \mathcal{L}}{\partial b^l} &= \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial \mathcal{L}}{\partial b^l}\right)_i. \end{aligned} \tag{4.11}$$

Hence, we have shown how to compute the gradient of $\mathcal{L}$ with respect to the weights W^l and biases b^l by (4.11). These gradients can be assembled into a ragged tensor in TensorFlow $\nabla_\theta \mathcal{L} = \left[\partial\mathcal{L}/\partial W^l, \partial\mathcal{L}/\partial b^l\right]^T = \left[\partial\mathcal{L}/\partial\theta\right]^T$, which can be applied in the gradient descent method (4.2). Additionally, we have shown how to compute the gradient of u with respect to the inputs $\mathbf{X}$ by (4.9) for use in PINNs.

4.3 Initialisation, Network-Structure and Activation Functions

As previously stated, activation functions introduce non-linearities into the neural network. Common activation functions include *tanh*, *ReLU*, *Swish* and *Sigmoid*. Figure 4.2 show some of them.

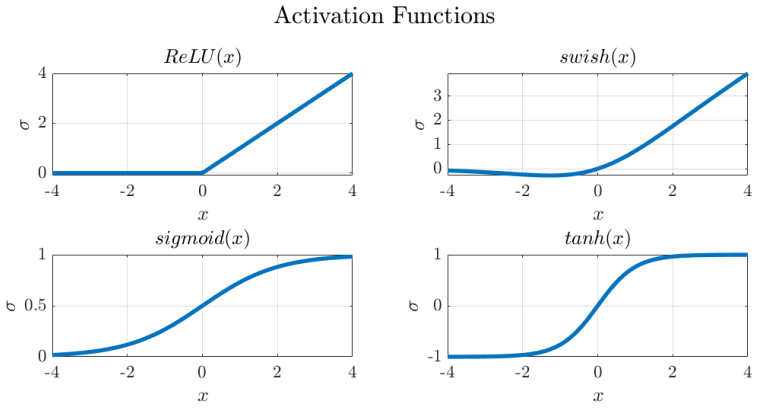

Figure 4.2: Common activation functions used in neural networks.

Before a network can be trained, one must first specify the activation functions $\sigma^l(\cdot)$ and initialise the parameters θ. But how do we choose what activation functions to use, and what should θ be initialised as? This is what we will discuss now.

When neural networks were first conceived, they often suffered from poor convergence. One of the reasons for this turned out to be how the weights and biases of the neural networks were initialised prior to training. But also from a poor choice of activation functions [14]. As it turns out, to get proper convergence, one must initialise the weights and biases in a specific manner which corresponds to the appropriate activation functions. One of the initialisation methods that were developed to this end was Glorot

initialisation also called Xavier initialisation [12].

Glorot initialisation, specifically normal Glorot initialisation, is based on initialising the weights by drawing from a normal distribution with a variance of σ and a mean $\mu = 0$. In other words $W^l_{i,j} \sim \mathcal{N}(\mu, \sigma^2_{stddev})$. For the use in PINNs, we found that *Raissi et al.* had used normal Glorot initialisation in their codes [43]. The variance used in normal Glorot initialisation is given by,

$$\sigma^2_{stddev} = \frac{2}{N_{in} + N_{out}}. \tag{4.12}$$

Here, N_{in} describes the number of connections entering the given node and N_{out} describes the number of connections exiting the node. By following *Raissi et al.*'s work, we found that the biases on the other hand, were initialised with zeros [43]. As such, when initialising the PINN we also used normal Glorot initialisation as specified by (4.12).

The proper choice of initialisation is important, because one tends to run into the problem of vanishing gradients whenever initialisation is done poorly. Vanishing gradients occur when activations become saturated and thus these nodes no longer contribute to training. This happens when the inputs x to the activation function $\sigma(x)$ become so small, or large that $\sigma'(x)$ becomes vanishingly small. As the *Glorot and Bengio* in [12] suggest, Glorot initialisation makes it so that the variance of gradients is maintained for all layers of the network. Therefore, one reduces the chance that any activations become saturated. One might ask why one must initialise the weights with a random distribution. Why can we not initialise all variables with the same value? The idea is that if all variables are initialised identically, then back-propagation will affect each node for every layer identically, this leads to the model acting as if each layer only has one node. Random initialisation will break the symmetry and allow for better utilisation of multiple nodes [14].

For many machine learning applications, *ReLU* is typically used as it works well empirically [14]. In the PINN case however, we are required to use activation functions which can be differentiated more than once. This is due to the fact that any arbitrary PDE might contain derivatives of higher order than one. If we were to use *ReLU*, we would be limited to first order derivatives in our PDE. In the case of the ADE (1.7), we have second order derivatives, thus we cannot use *ReLU*. In all contemporary papers we had found on PINNs [21, 24, 29, 31, 42, 43, 49, 51] the *tanh* activation function was used. Additionally, *Markidis* in [29] state that PINNs with *ReLU* and other non-smooth activation functions, are not convergent methods. In the limit of an infinite training data-set a well-trained PINN with *ReLU*-like activation functions, the solution does not converge to the exact solution [29, 34].

As *Glorot and Bengio* in [12] suggest, their proposed normalised Glorot initialisation scheme can be helpful in networks which use the *tanh* activation function. This is because magnitudes of activations and gradients are preserved between layers both in forward and backward propagation. Hence, we also decided to settle on the *tanh* activation function with Glorot initialisation.

The structure of the network is also an important aspect to be determined. One needs to specify how many hidden layers there should be and how many nodes there should be for each of the hidden layers individually. Unfortunately, there are not any strict guidelines one can adhere to for optimal performance in this regard [14]. *Raissi et al.* gave the following piece of advice: For PINNs to perform well, it is at a minimum, required for the network to have sufficient complexity to express the possible solutions one would expect, given the PDE in question [43]. He argues that the network needs to be given sufficient approximation capacity in accordance with the complexity of the solution. However *Raissi* admits in his talk [41] that the interplay between required architecture and the complexity is still poorly understood.

Generally, it holds that the more hidden nodes and layers there are, the more expressive the network becomes. One also gets a higher efficiency by adding more hidden layers versus adding more nodes per hidden layer [14]. Thus, it is beneficial to construct deeper neural networks instead of very wide ones. Though, more layers and neurons come at an obvious computational cost. Hence, this is a balancing act that needs to be performed. A network too small will not be able to express the nature of the solution, while a network too large will require a greater amount of time and memory to train.

Instead of choosing a structure blindly. We instead, decided to use what we had observed to work for other problems. We chose the same neural network structure as was used in [43] for the inverse 2D Navier-Stokes problem. Namely, a densely connected FNN with 8 hidden layers with 20 neurons each. The Navier-Stokes equation is a non-linear version of the ADE, and as such, typically more difficult to solve for. Therefore, we argue that if this specific architecture was sufficient to express the solution for the Navier-Stokes equation, then it should be sufficient for the ADE. This may not be a perfect argument, however *Krishnapriyan et al.* [24] suggest that the failure of a PINNs convergence is often not due to insufficient expressivity, but due to problems with optimisation. This claim will be validated later on in chapter 8. As such, we feel confident that this network should be expressive enough.

Deciding how many input and output nodes there are however, is trivial. The input nodes for a PINN will simply correspond to each of the input variables available. For our case, spatial and temporal coordinates. For a one dimensional time-dependent problem, it necessary to have two input nodes, one for x the spatial variable and one for t the temporal variable. For a two dimensional time-dependent problem, one simply adds another input node corresponding to the spatial variable y. As we will only predict one quantity, namely the concentration u, we only need one output node. The network will have the same form as shown in figure 4.1, but with more hidden nodes and more hidden layers.

Before we finish this section, we need to look at normalisation of the inputs. The $tanh$ activation function becomes saturated for inputs tending towards large absolute values. This becomes evident by looking at the derivative of $tanh$ given by $sech^2$ as shown in figure 4.3.

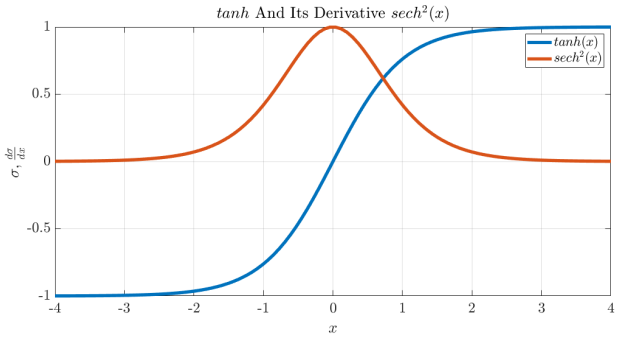

Figure 4.3: Plot of $tanh(x)$ versus its derivative $sech^2(x)$.

As we see, the derivative tends rapidly towards zero as the inputs lie further and further away from the origin. As such, we need to ensure that the inputs to our network lie in this 'effective range' to avoid vanishing gradients. To this end, *Raissi et al.* in [43] used a normalisation scheme applied to the inputs before being fed to the input layer as given by,

$$\mathbf{X} \rightarrow \frac{2(\mathbf{X}-\mathbf{lb})}{\mathbf{ub}-\mathbf{lb}} - 1. \tag{4.13}$$

Here, $\mathbf{X} = [\mathbf{x}, t]^T$. **ub** is the component-wise upper bounds of $\mathbf{X}$ while **lb** is the component-wise lower bounds of $\mathbf{X}$. The division occurs component-wise.

If $\mathbf{X} = \mathbf{ub}$, in other words, each element of $\mathbf{X}$ take their maximum values, then the input becomes normalised to one, while if $\mathbf{X} = \mathbf{lb}$, then the input becomes normalised to negative one. As such, no matter the range of inputs $\mathbf{X}$, we always have the inputs to the network be normalised to the range $[-1, 1]$ to most effectively utilise the *tanh* activation function gradients.

4.4 Minimising the Loss $\mathcal{L}$

The training of a neural network is at its core an optimisation problem. The goal is to find the parameters θ which give the optimal value for the objective loss function $\mathcal{L}$. To this end, there has been many different optimisation algorithms developed for use with neural network optimisation. We will discuss training with full-batching and mini-batching and two of the common optimisers used to train PINNs in the following sections.

Full-Batch Gradient Descent

The full-batch gradient defines the loss $\mathcal{L}$ using the entire data-set. Here N_u and N_f constitute the entire data-set for the collocation-points $\{\mathbf{x}_i, t_i, u_i^*\}_{i=1}^{N_u}$ and data-points $\{\mathbf{x}_i, t_i\}_{i=1}^{N_f}$ respectively. Then θ can be updated using the optimiser $\mathbb{G}$. One step with the optimiser is then considered an epoch as we have used the entire data-set in one update. Let the loss $\mathcal{L}$ be given as equation (3.5), then:

$$\mathcal{L}(\theta) = \frac{u_m}{N_u}\sum_{i=1}^{N_u}|u_i(\mathbf{x}_i,t_i;\theta)-u_i^*|^2 + \frac{f_m}{N_f}\sum_{i=1}^{N_f}|f(\mathbf{x}_i,t_i;\theta)|^2,$$

$$\theta_{n+1} = \theta_n - \alpha\cdot\mathbb{G}[\nabla_\theta\mathcal{L}].$$

The main issue with this method is that it may be slow to compute and memory intensive, particularly when the data-set is large. It can also tend to get stuck in local minima. This could lead to cases where the loss will quickly stagnate despite the model underfitting on the data.

Despite these disadvantages, we found that full-batch gradient descent was by far the most common option for training PINNs, we could only find one example where mini-batching was explicitly used [22].

Mini-Batch Gradient Decent

The mini-batch gradient descent method is meant to alleviate some of the issues of full-batch gradient descent. This is done by computing the gradient of $\mathcal{L}$ for a random subset of training points data-set $N_u^{rnd} \subset N_u$ and $N_f^{rnd} \subset N_f$. This noisy 'stochastic gradient' is then applied to the network several times for each epoch until each data-point has been used. Here each epoch may consist of multiple updates. Let the loss $\mathcal{L}$ be given as equation (3.5), then:

$$\mathcal{L}(\theta) = \frac{u_m}{N_u^{rnd}}\sum_{i=1}^{N_u^{rnd}}|u_i(\mathbf{x}_i,t_i;\theta)-u_i^*|^2 + \frac{f_m}{N_f^{rnd}}\sum_{i=1}^{N_f^{rnd}}|f(\mathbf{x}_i,t_i;\theta)|^2,$$

$$\theta_{n+1} = \theta_n - \alpha\cdot\mathbb{G}[\nabla_\theta\mathcal{L}].$$

One of the main advantages of mini-batch gradient descent is that the gradient $\nabla_\theta\mathcal{L}$ does not necessarily point in the true direction of greatest descent. This is due to the randomness in the sampling of N^{rnd}. This means that with every step taken, one moves only approximately towards the optimum. Therefore the optimiser may be able to escape local minima. Despite only computing approximate descent directions, networks still tend to converge close to the optimal solution [14]. A smaller amount of training data for each update also helps to alleviate with the memory issues encountered with full-batch gradient descent. The downsides is that one needs to specify how many training points that go into each mini-batch N^{rnd}. A range from 1 to 512 training points work well empirically [14]. *Kadeethum et al.* [22] found a similar range that works well for PINNs on non-linear Biot's equations. We did conduct tests with mini-batch gradient descent, but found that mini-batching was neither significantly beneficial or detrimental to performance.

The Regular Gradient Descent Optimiser

The regular gradient descent method follows the form shown in equation (4.14).

$$\theta_{n+1} = \theta_n - \alpha\nabla_\theta\mathcal{L}(\theta_n) \tag{4.14}$$

It implies that the gradient operator $\mathbb{G}[\cdot]$ in (4.2) is the unitary operator. This is the most simplistic form of the gradient descent. The main issue with this optimiser is that the learning rate α is global and affects each of the gradients equally. This may not always be optimal, as some parameters may be better suited to smaller or larger steps. We found that none of the implementations of PINNs in the literature used this form of gradient descent, as such we did not use it either.

The ADAM Optimiser

The ADAM optimiser is a popular adaptive optimiser used for training neural networks. Adam stands for adaptive moment estimation. It has been shown that for a variety of different machine learning problems, the ADAM optimiser is a computationally efficient and versatile optimisation method [14, 23]. The advantages to the ADAM optimiser are that each parameter of the neural network are updated according to individual adaptive learning rates. Specifically, each parameter will update at different rates depending on the parameters 'Signal-to-Noise-ratio' (SNR). This ratio tells the optimiser how certain an update to a parameter is aligned with the true gradient or not. If the SNR is high, then the gradient is robust to changes in the data or noise. In such an instance the parameters will be updated with a large learning rate. If the SNR is low, then noise dominates, and the parameters will not be updated as heavily. The SNR can also be low in instances where the parameters are close to an optimum. In which case, we would want the step sizes to be small as we would not want to walk further away from the optimal values.

Regardless, the learning rate is approximately bounded by the global learning rate α, such that updates cannot grow arbitrarily large. This effectively gives each parameter its own adaptive learning rate. ADAM then has the advantage that it is not as sensitive to learning rate tuning. The ADAM optimiser can be summarised by equations (4.15 - 4.18). These equations were taken from the original publication of ADAM [23].

The first moment estimate m_{n+1} of the gradient is computed with the exponential moving average as shown in equation (4.15). Similarly, the second moment estimate v_{n+1} is computed with another exponential moving average as shown in equation (4.16). Next, the first and second moments are corrected via a bias-correction term (4.17) which is dependent on the iteration number n.

$$m_{n+1} = \beta_1 m_n + (1-\beta_1)\cdot \nabla_\theta \mathcal{L}(\theta_n) \tag{4.15}$$

$$v_{n+1} = \beta_2 v_n + (1-\beta_2)\cdot \left(g_{n+1} \odot g_{n+1}\right) \tag{4.16}$$

$$\begin{aligned} \hat{m}_{n+1} &= \frac{m_{n+1}}{1-\beta_1^{n+1}} \\ \hat{v}_{n+1} &= \frac{v_{n+1}}{1-\beta_2^{n+1}} \end{aligned} \tag{4.17}$$

Finally, the weights and biases of the network θ are updated via the gradient descent update in equation (4.18). The ratio $\hat{m}_n/\hat{v}_n$ corresponds to the SNR.

$$\theta_{n+1} = \theta_n - \alpha \cdot \frac{\hat{m}_{n+1}}{\sqrt{\hat{v}_{n+1}} + \varepsilon} \tag{4.18}$$

The value ε is there for numerical stability, as one would run into issues if the second moment estimate $\hat{v}$ ever went to zero. By default in TensorFlow, the parameters are set as $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-7}$. Empirically the default values set by TensorFlow should work well for most problems [14]. And indeed we found that *Raissi et al.* had not changed these values either in their codes when implementing the Baseline-PINN [43].

The ADAM optimiser is the main optimiser we used in our application of PINNs. We found that all studies focusing on PINNs utilised this optimiser as well. However, it seems that ADAM is not always sufficient to achieve optimal accuracy. Therefore, ADAM is often supplemented with another optimiser, namely L-BFGS.

The L-BFGS Optimiser

In the canonical literature for PINNs, it is commonplace to supplement the ADAM method with the L-BFGS optimiser. This optimiser is a second order optimiser, meaning that it uses the Hessian of $\mathcal{L}$ with respect to the parameters θ. This is as opposed to ADAM which is a first order optimiser as it only uses the first order gradients.

Generally for PINNs, one trains the network with ADAM first, then one switches over to L-BFGS optimisation as the loss begins to stagnate and little is gained with any further training with ADAM. More often than not, one tends to get vastly better performance if L-BFGS optimisation is used in tandem with ADAM. One typically does not train with L-BFGS alone as L-BFGS has a higher likelihood to get stuck in local minima [29]. Therefore, ADAM is used to guide the network approximately towards an optimal minimum, where L-BFGS takes care of the rest. As claimed by *Bishop* [8], full-batch optimisation using simple gradient descent is actually a poor algorithm. For full-batch optimisation, quasi-Newton methods such as L-BFGS is much preferred. Moreover, *Markidis* [29] claims that L-BFGS is essential for the PINN to succeed.

In our studies, we performed test cases both with and without the addition of L-BFGS optimisation. We found that yes, L-BFGS does improve performance by a significant margin, especially with regards to accuracy on parameter discovery in the inverse problems. As such, we will also be using L-BFGS.

The discussion of how L-BFGS operates goes outside the scope of our studies, for more information see [40] or [25].

4.5 Network Summary

We have now explained how a network is evaluated and trained in general terms. We have also explained the choice of initialisation, network-architecture and activation functions for PINNs. We found that PINNs require a specific set of treatments to work properly. At first glance, these choices seemed ambiguous to us, however the more we investigated them, the more they made sense. This is why we decided to keep the network in line with the Baseline-PINN in [43].

In summary, we chose a network with 8 hidden layers and 20 nodes each. Each hidden layer is assigned the *tanh* activation function, while the first and last layers are linear. The input layer is scaled by the normaliser in (4.13). The parameters of the network are initialised according to the normal Glorot scheme (4.12). We then train the networks first using ADAM, then by using LBFG-S. This is all in accordance with how the Baseline-PINN is implemented in [43].

Chapter 5

Methodology

5.1 Generating Data with MATLAB

To train the PINN for forward and inverse problems, one needs training data. As discussed in chapter 2, the finite difference method with the RK4 time-marching scheme was used to generate training and testing data u^*. Specifically, MATLAB [47] was used to implement the numerical solver in code. The finite difference schemes for the 1D ADE (B.5) and 2D ADE (B.10) were converted into matrix-vector form as shown in equation (2.3), then solved by using equation (2.7).

As shown in chapter 2, the PDEs underlying the discretizations are well-posed. We also know that the discretizations themselves are stable under the RK4 time-marching scheme. Therefore, we know that for a sufficiently fine grid, the solutions we generated should be close to the real solution. And we can be reasonably assured that solutions will not explode or exhibit any growing oscillatory solutions.

We generated the solutions on sufficiently fine regular rectangular grids on the domains of interest $\Omega \times [0,T]$ such that the relative L^2-norm error (5.1) was below 0.1%. Figure 5.1 and figure 5.2 show the reference solutions u^* we will be working with in 1D and 2D respectively.

5.2 Implementing PINNs in Python

When constructing PINNs, there are a couple of different options available to do so. The TensorFlow [1] library offers a wide variety of in-built functions and model structures, which make many canonical machine learning problems such as image classification simple plug-and-play affairs. However, the loss function used in PINNs (3.5) cannot be constructed using these in-built functions directly. It must instead be constructed using lower-level functionality.

To this end, one can either code the PINN using these lower-level functionalities, or one can use established PINNs libraries such as DeepXDE [26, 27], SciANN [15, 16] or TensorDiffeq [32, 33]. These libraries offer simple yet powerful implementations of PINNs, without having to deal with the underlying machinery of TensorFlow. The

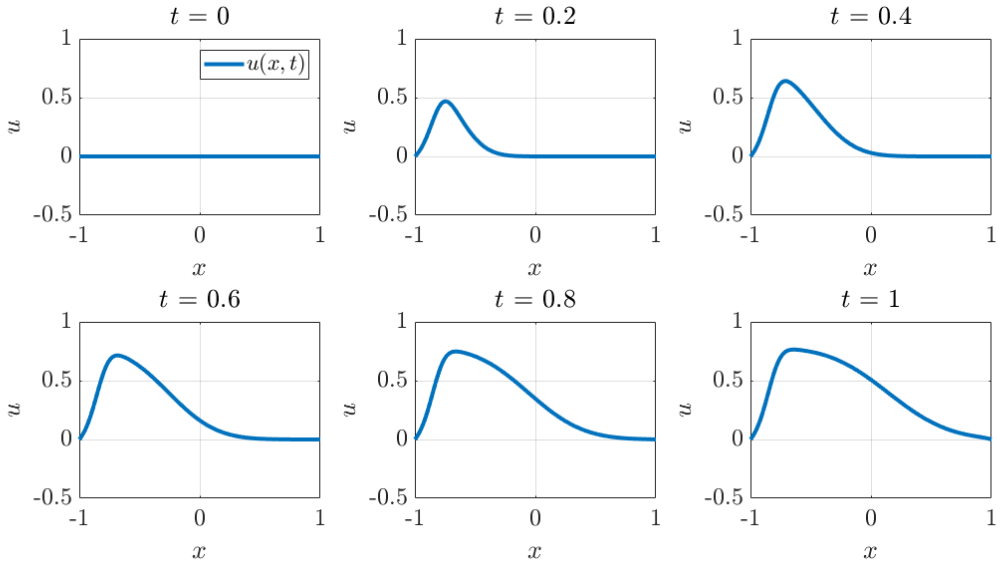

Figure 5.1: Data generated by the finite difference method in MATLAB for the 1D ADE as specified by equation (1.8)

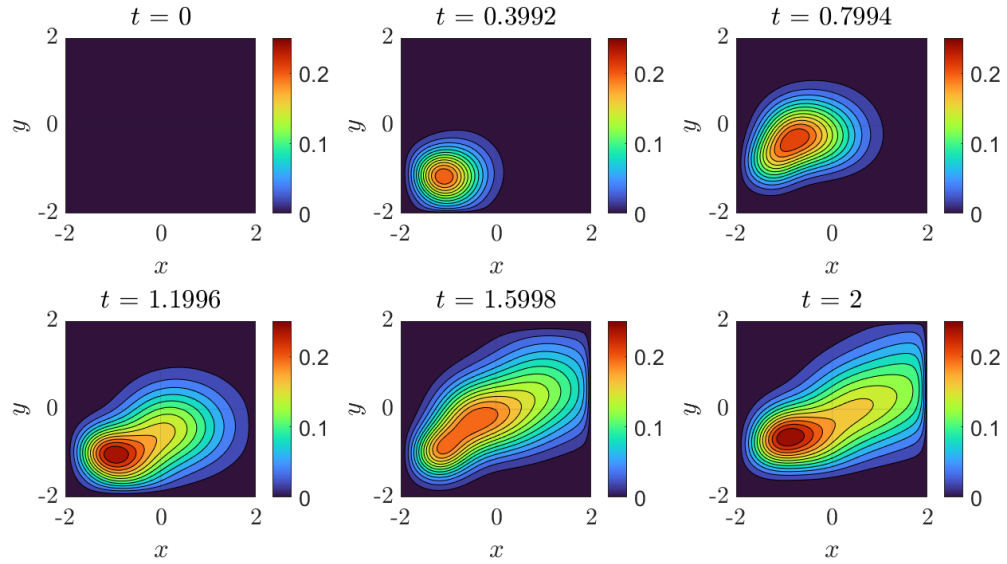

Figure 5.2: Data generated by the finite difference method in MATLAB for the 2D ADE as specified by equation (1.9)

DeepXDE library in particular comes with the powerful ability to define any spatial domain Ω as the union and intersection of simple shapes. With this feature, one can solve for PDEs on complicated domains in a straight forward manner. DeepXDE also has implemented other experimental training schemes which may help with improving the accuracy of PINNs.

Despite these advantages, we decided not to use them; to both avoid using the PINN as a black box, and to have full control over how the code functions. Also, these PINN libraries are still under development at the time of writing this thesis, with varying quality of documentation. Thus, whenever a problem was encountered, it became prohibitively difficult to debug them. Therefore we decided to code PINNs directly using the TensorFlow library instead. In the literature, it appears that it is most common to use TensorFlow 1.0 (TF1) for PINNs. However, as there is less up-to-date documentation for TF1, we instead decided on using TensorFlow 2.0 (TF2). This comes with the advantage of the syntax being easier to understand and with a greater amount of support for it. But with the disadvantage that a bit of extra legwork was required to construct custom training loops via low-level functions in TF2. This is because TF1 and TF2 operate quite differently. Hence, any code published by other researchers in TF1 could not be transferred to our code without heavy tweaking and workarounds.

The code that was used in this study was originally made by *Jacquier* [19]. It is a TF2 conversion of the TF1 Burger's equation PINN made by *Raissi et al.* [43]. This code was then altered to solve the ADE and then modified further to handle 2D problems. We also added many additional features to it. Such as a way to monitor the history of parameter prediction during training, the addition of mini-batch training and customised training-data pre-processing. Since we had full control of the training pipeline, we could add experimental algorithms which were meant to improve convergence, as will be discussed in chapter 7.

In short, the network $u(\mathbf{x}, t; \theta)$ was constructed as outlined in chapter 4. This was done using the snippet of Python code shown below. Note that the for-loop iterates over every hidden layer and applies the activation function according to the Python variable **activation_func**. However, in the code provided by *Jacquier* [19], we found a discrepancy. In his code, the final layer is also assigned an activation function. We believe that this may have been a mistake because this does not align with that was done by *Raissi et al.* for the Baseline-PINN [43]. Hence, we decided to correct this inconsistency. Thus, the activation function in the final layer was made linear. As consistent with the formulation of the Baseline-PINN.

Listing 5.1: u - model

```
1 self.u_model = tf.keras.Sequential()
2 self.u_model.add(tf.keras.layers.InputLayer(input_shape=(layers[0],)))
3 self.u_model.add(tf.keras.layers.Lambda(lambda X: 2.0*(X - lb)/(ub - ...
      lb) - 1.0))
4 for width in layers[1:-1]:
5     self.u_model.add(tf.keras.layers.Dense(
```

```
        width, activation=activation_func,
        kernel_initializer='glorot_normal'))
self.u_model.add(tf.keras.layers.Dense(
  layers[-1], activation=None,
  kernel_initializer='glorot_normal'))
```

The residual f to be fed into the residual loss (3.14), was computed using AutoDiff [5] on the network **self.u_model** with the **tf.GradientTape** paradigm. The following snippet of Python code shows how the residual f is computed for the 2D ADE. Here, **l1, l2, l3** are the unknown parameters $P = [\lambda_1, \lambda_2, \lambda_3]^T$ to be discovered by the PINN in the inverse 2D case. The variable **X_u** consists of data-points of the form $\{x_i, y_i, t_i, u_i^*\}$.

Listing 5.2: f - model

```
def __f_model(self, X_u):
    l1, l2, l3 = self.get_params()
    x_f = tf.convert_to_tensor(X_u[:, 0:1], dtype=self.dtype)
    y_f = tf.convert_to_tensor(X_u[:, 1:2], dtype=self.dtype)
    t_f = tf.convert_to_tensor(X_u[:, 2:3], dtype=self.dtype)

    with tf.GradientTape(persistent=True) as tape:
        tape.watch(x_f)
        tape.watch(y_f)
        tape.watch(t_f)
        X_f = tf.stack([x_f[:,0], y_f[:,0], t_f[:,0]], axis=1)

        u = self.u_model(X_f)
        u_x = tape.gradient(u, x_f)
        u_y = tape.gradient(u, y_f)

    u_xx = tape.gradient(u_x, x_f)
    u_yy = tape.gradient(u_y, y_f)
    u_t = tape.gradient(u, t_f)

    del tape

    PDE = u_t + 2.0*u_x + (3.0*tf.math.cos(3.0*np.pi*t_f)+1.0)*u_y - ...
        0.2*u_xx - 0.2*u_yy - 13*tf.math.exp(-4.0*tf.math.square(x_f ...
        - l1)-4.0*tf.math.square(y_f - l2))

    return PDE
```

As will be discussed later, the loss may be noisy and we might not always decrease the loss with an iteration of our optimiser. To avoid the risk of ending up with a large loss at the end of training, we decided to employ a model save-state-algorithm. When iterating over our epochs, the algorithm will compare the current loss (3.5) with the lowest loss achieved. If the current loss is higher, the algorithm will continue training. If not, the current state of the network, θ, will be saved before training continues. At the very end of training, the network will recover this saved state and re-apply the parameters θ to the network. This makes sure that by the end of training, the network will perform optimally. This may sound ill-advised for general neural network problems. As typically we want to avoid over-fitting. But here we are mostly interested in lowering the loss as much as possible. This is to get the most accurate results on the training data.

PINNs require a long time to train and they can be computationally and memory intensive. For large networks and data-sets, graphics processing units (GPUs) are typically required for training within reasonable time-frames. Our GPU unfortunately did not meet the memory requirements for training. Therefore, we could at worst be faced with training times of 10 hours. As training was very slow without a GPU, we had to resort to cloud computing services. We used Google Colab [13], an online Jupyter notebook environment specialised for prototyping machine learning models. Thanks to their GPUs, we reduced training times significantly, down to the order of an hour at worst. This was invaluable for whenever we wanted to test and debug code multiple times.

Figure 5.3 shows a bare bones flowchart describing the implementation of the Baseline-PINN in pseudo-code.

5.3 Solving the ADE and Determining Accuracy

We generated reference solutions of the ADE via the methods discussed in 5.1. The PINN then used the reference solutions as training data. When training was completed, the PINN was be able to generate its own predictions. The reference solutions was denoted as $\mathbf{u}^*$ while the predictions given by the PINN as $\mathbf{u}$.

We wanted to look at how well the PINN performs as a forward and inverse solver. When it comes to the determining the accuracy of our solutions u, we used the relative L^2-norm error. It is here defined as,

$$E_{Rel} = \sum_{i=1}^{N} \frac{||\mathbf{u}_i - \mathbf{u}_i^*||_2}{||\mathbf{u}_i^*||_2}. \tag{5.1}$$

Here u is the solution generated by the PINN, while u^* is the reference solution generated by MATLAB. N is the total number of points on the grid we are evaluating on.

When it comes to the inverse problem, we used the relative error of the unknown parameters. Let the true solution be denoted as λ_i^* and the predicted solution as λ_i, then this error takes the form,

$$E_{\lambda_i} = \frac{|\lambda_i - \lambda_i^*|}{|\lambda_i^*|}. \tag{5.2}$$

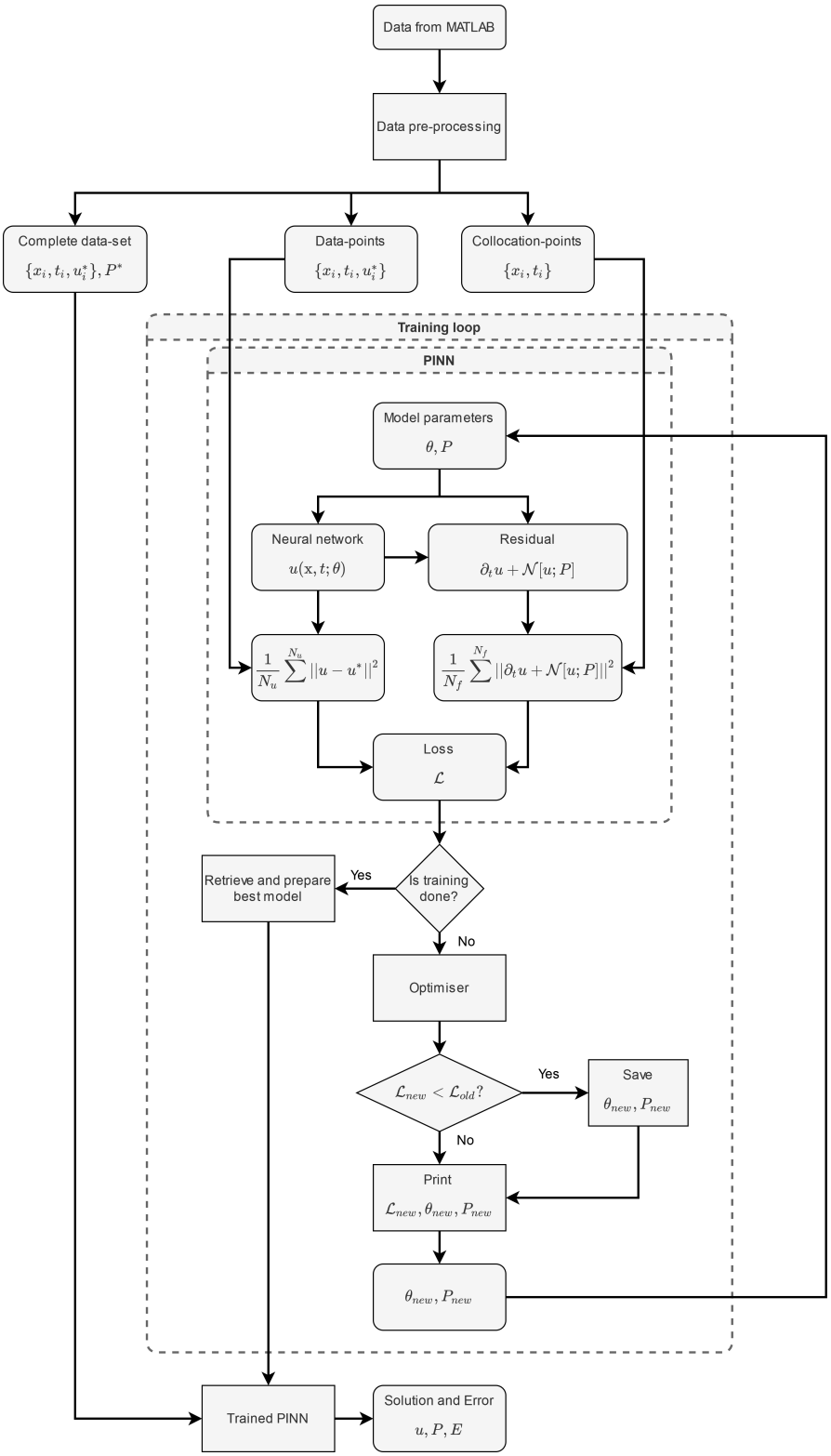

Figure 5.3: Bare bones flowchart of how the Baseline-PINN is implemented in our code.

Chapter 6

Using the Baseline-PINN

This chapter focuses on how the Baseline-PINN as described by *Raissi et al.* [43], performs for the 1D and 2D ADE in the forward and inverse case.

6.1 The Baseline-PINN on the 1D Advection-Diffusion Equation

6.1.1 The 1D Forward Problem

This first section showcases how the Baseline-PINN can approximate the solution for the 1D ADE problem as specified by the problem-statement in equation (1.8).

In the forward case, the PINN only requires the reference solution u^* at the initial state and at the boundaries. This data is specified by the initial- and boundary-conditions. These conditions are imposed on the PINN via $N_u = 400$ randomly sampled data-points on the spatio-temporal boundaries of the form $\{x_i, t_i, u_i^*\}_{i=1}^{N_u}$. Additionally, the PINN will train on $N_f = 5000$ randomly sampled collocation-points $\{x_i, t_i\}_{i=1}^{N_f}$ within the space-time domain. The training-data is shown in figure 6.1, the horizontal axis represents the temporal dimension while the vertical axis represents the spatial dimension. Note that P is fully specified here.

After optimising θ with ADAM and L-BFGS, we end up with a final objective function loss $\mathcal{L}$ of 6.1×10^{-6}. The predicted solution given by the network $u(x,t;\theta)$ is shown in figure 6.2. We end up with a relative L^2-norm error (5.1) of 0.87%.

By figure 6.2 see that the solution given by the PINN u accurately mimics the dynamics of the system. There are no visible discrepancies when compared with the reference solution u^* generated by MATLAB, and indeed the L^2-norm error is small.

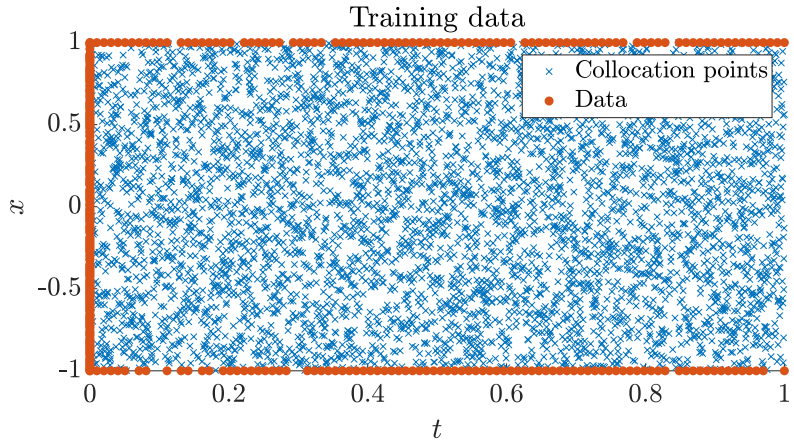

Figure 6.1: Dense training data. Orange dots represent data-points sampled from boundary and initial conditions. Blue crosses represent collocation-points.

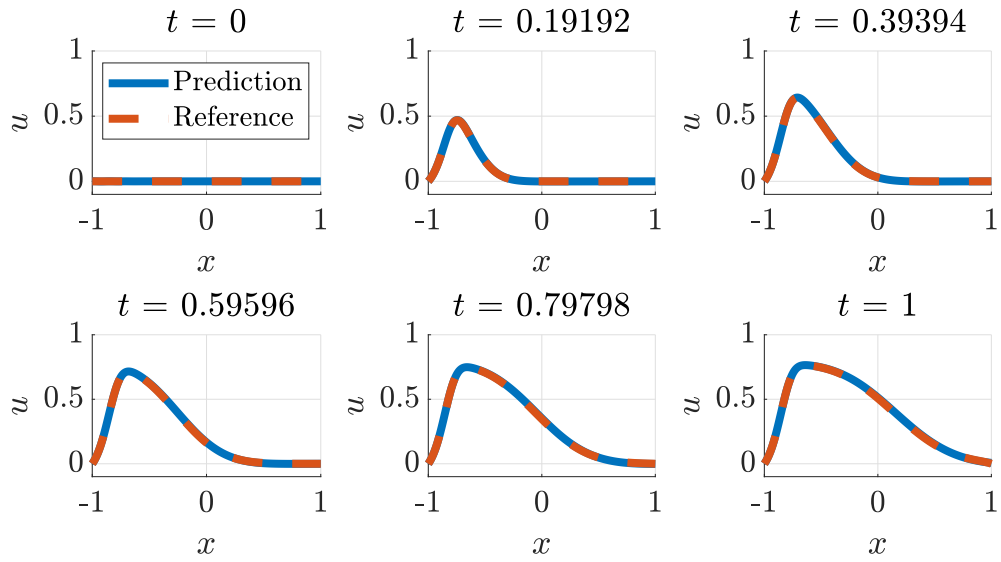

Figure 6.2: The predicted solution u generated by the forward Baseline-PINN versus the reference solution u^* of the 1D ADE as specified by equation (1.8) at different snapshots in time.

6.1.2 The 1D Inverse Problem

In the following section we show how the Baseline-PINN can accurately discover the location and strength the source in the 1D ADE specified by (1.8), i.e, λ_1 and λ_2. These two parameters were left as unknown variables for the PINN to discover.

In the inverse case, we feed the PINN N_u randomly sampled data-points from the reference solution u^* over the entire domain. The data-set looks similar to 6.1, but here collocation points are exchanged with data-points $\{x_i, t_i, u_i^*\}_{i=1}^{N_u}$. We kept the total number of points the same.

Again, we trained the network with ADAM then L-BFGS. Figure 6.3 shows the prediction histories of the two unknown variables and the corresponding loss over each

epoch of training. The dashed horizontal lines in figure 6.3 show the corresponding target values (the true λ^* values used to generate the reference data u^*). We see that the unknowns λ_1 and λ_2 move away from their initial guesses and eventually converge onto their target values. Figure 6.4 shows the reconstructed solution given by the PINN. However, disappointingly the inverse PINN struggles to satisfy the initial condition at $t = 0$ as there is now a visible deviation from the reference solution near the source. Despite this discrepancy at the initial time, this reconstruction is overall more accurate than the forward case as the relative L^2-norm error was here 0.50% according to equation (5.1).

By looking at figure 6.3 we can see a clear benefit of using L-BFGS. Particularly in the top panel, we see that λ_2 takes a large number of epochs to converge with ADAM, and it eventually stagnates just below the true value. However, once L-BFGS takes over, it quickly jumps to the target value. Table 6.1 summarises our results.

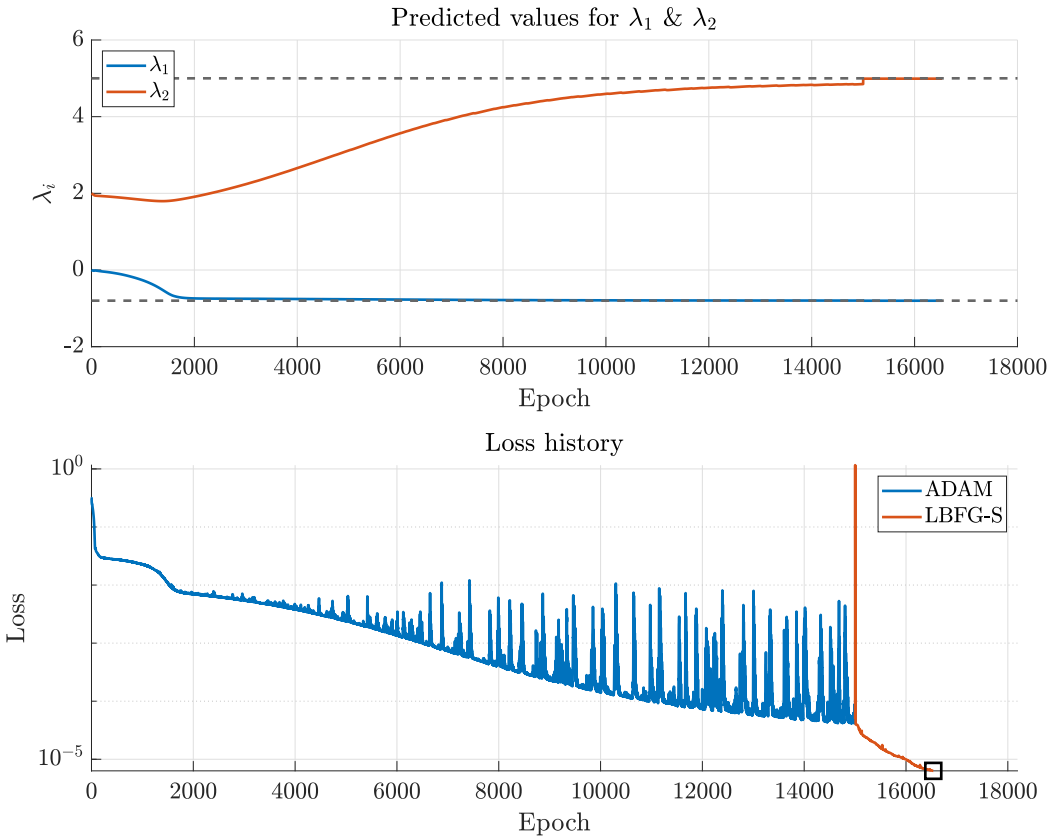

Figure 6.3: Training histories for the Baseline-PINN on dense data in the inverse case for the problem specified by equation (1.8). Top: The predicted values for λ_1 and λ_2 over each epoch of training. Bottom: The Objective function loss over each epoch of training. Lowest achieved loss given by the black square. Notice how loss decreases significantly once LBFG-S training is engaged.

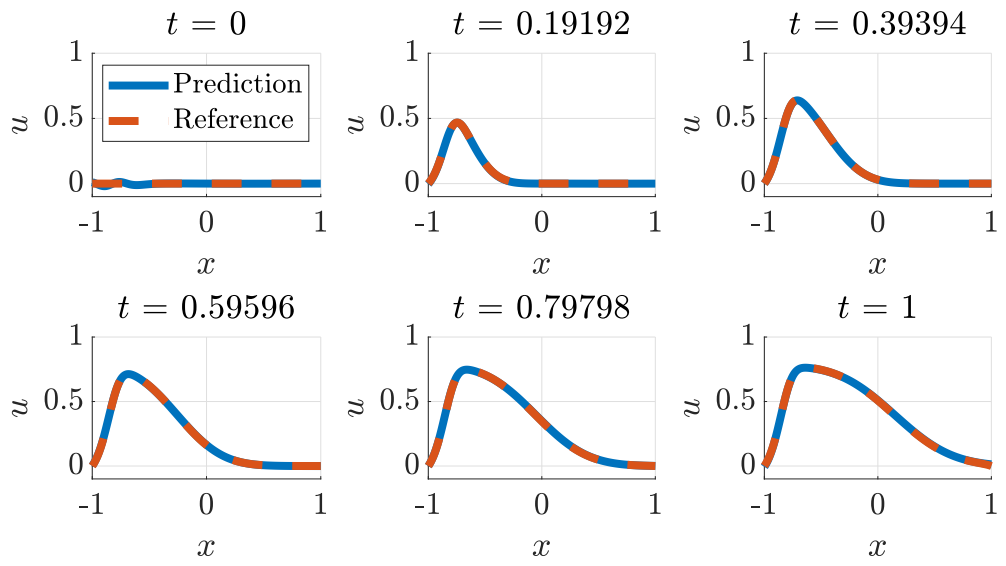

Figure 6.4: The predicted solution u generated by the inverse Baseline-PINN versus the reference solution u^* of the 1D ADE as specified by equation (1.8) at different snapshots in time.

Baseline-PINN					
N_u	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_u =$ 5400	λ_1	-0.8010	-0.8000	0.13%	$E_{Rel} = 0.5\%$
	λ_2	4.9912	5.0000	0.18%	

Table 6.1: Results generated by the Baseline-PINN versus the true target values for the unknown parameters λ_1 and λ_2 of the 1D ADE as specified by (1.8).

6.2 The Baseline-PINN on the 2D Advection-Diffusion Equation

The tests in the previous sections 6.1.1 and 6.1.2 were conducted to show that the Baseline-PINN employed in chapter 5 works as expected on simple advection-diffusion scenarios. We will now promote the system into 2D, and now include time-dependent velocities.

6.2.1 The 2D Forward Problem

This next section showcases that the previous forward Baseline-PINN 6.1.1 can also handle the 2D time dependent ADE. In this case we solved for the ADE as specified by (1.9).

In this case we gave the network $N_f = 40000$ collocation-points of the form $\{x_i, y_i, t_i\}_{i=1}^{N_f}$ and $N_u = 10000$ data-points on the boundaries and at the initial time of the form $\{x_i, y_i, t_i, u_i^*\}_{i=1}^{N_u}$. The training data is shown in figure 6.5, note that though this data-set consists of a dense cloud, it does not constitute every available data-point. Only about 0.3% of the available 2×10^7 data-points were used. The predicted solution given by

the network u is shown in figure 6.6. Again, ADAM was used in tandem with L-BFGS to optimise the loss until we reached stagnation.

To more clearly show the magnitudes of error between the the predicted u and reference solutions u^*, we show the absolute point-wise error between the two in figure 6.7. The L^2-norm error was 4.3%.

By figure 6.6 we see that the solution given by the PINN, gives a good qualitative description of the dynamics of the system, albeit not perfectly accurate as we can see in figure 6.7. In large part, the errors are the most prevalent by the incorrect prediction of the initial condition, which also affects later predictions. This may be alleviated by introducing more collocation-points at the initial instance, or by some of the fixes which will be discussed in section 7.1.

Regardless, the tests shown here and in section 6.1.1 demonstrate that PINNs can be used to solve PDEs without having to deal with complicated grid algorithms or time-stepping procedures. This is because it solves for the entire domain at the same time. One drawback however, is that we cannot be guaranteed that the solution we get is within the accuracy tolerances we want, with more training. Despite loss stagnating, the accuracy still leaves a lot to be desired. Accuracy may be improved by increasing the number of training data as the following studies [34, 35, 46] suggests.

On the other hand, these tests validate one of the claims we made in chapter 4. We claimed that the network architecture we chose should be able to express the solution of the 2D ADE, and indeed it does.

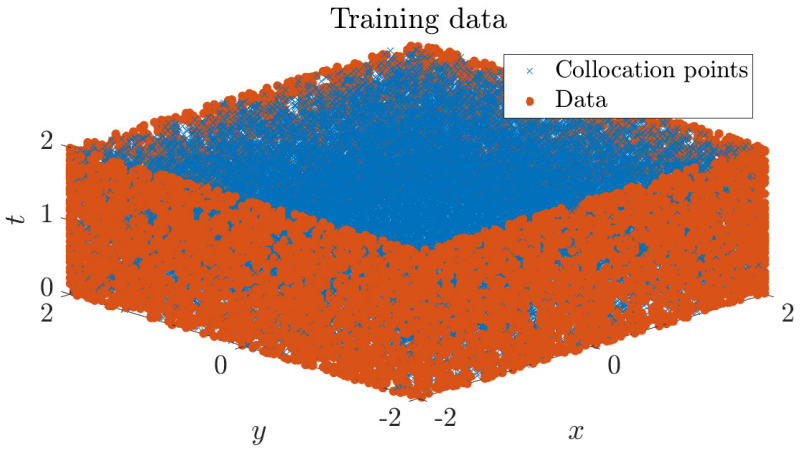

Figure 6.5: Dense training data. Orange dots represent data-points sampled from boundary and initial conditions. Blue crosses represent collocation-points.

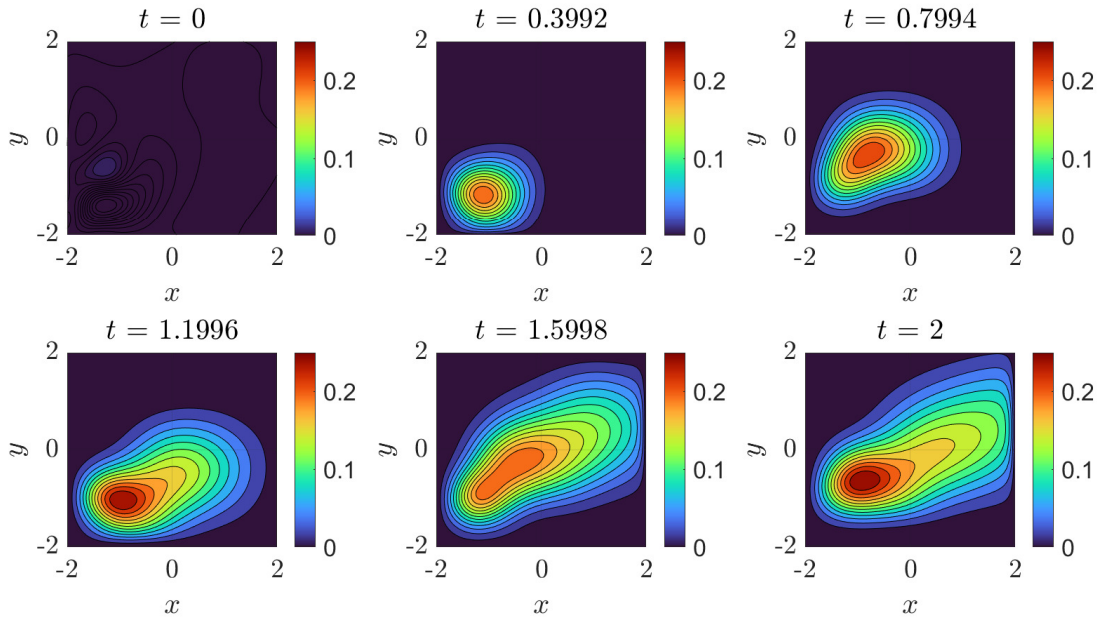

Figure 6.6: The predicted solution u generated by the forward Baseline-PINN of the 2D ADE as specified by equation (1.9) at different snapshots in time.

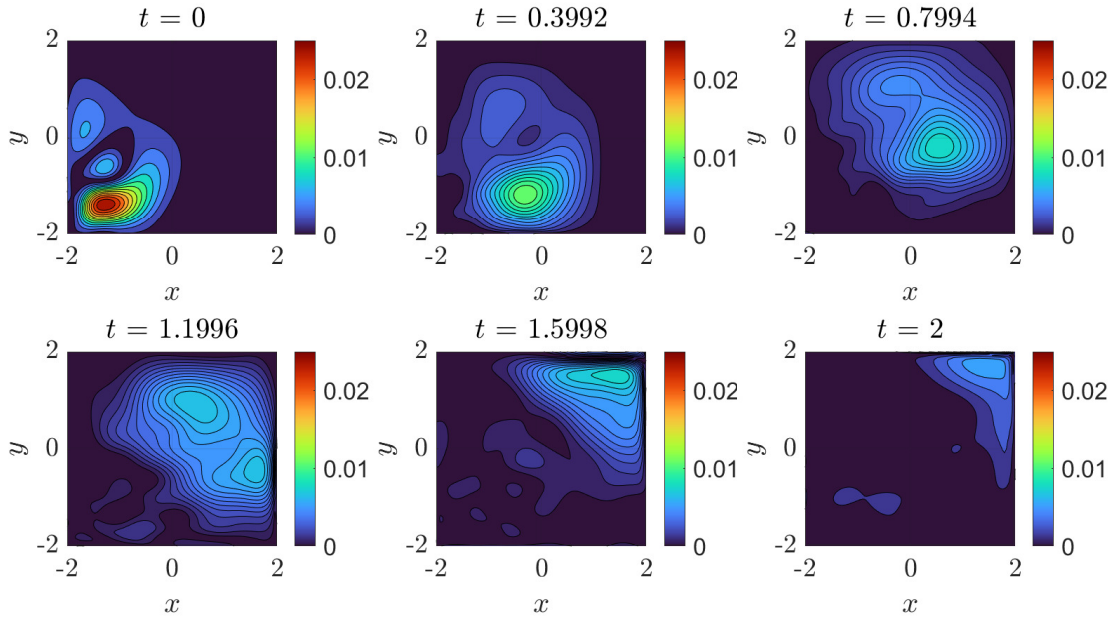

Figure 6.7: Absolute point-wise error between the predicted solution u generated by the 2D forward Baseline-PINN and the reference solution u^* for different snapshots in time.

6.2.2 The 2D Inverse Problem

Sections 6.1.1 and 6.1.2 showcased that the Baseline-PINN we implemented operates as expected, while section 6.2.1 showed that the network architecture is sufficient for the solutions we will be looking at. Now we will move over to problem at focus in this study, the 2D inverse problem.

Up until now, we have shown that the Baseline-PINN algorithm as described by *Raissi et al.* [43], with $f_m = u_m = 1$, is sufficient for solving the 1D forward, 1D inverse and the 2D forward test cases. In all three instances we got acceptable results without any hyper-parameter tweaking. Despite our successes so far with this method, we will now demonstrate that we run into issues in the following 2D inverse case.

Here, we want the Baseline-PINN to discover the 2D ADE as specified by (1.9) with three unknowns λ_1, λ_2 and λ_3. The unknowns specify the x, y coordinates of the source and the strength respectively. The training data consisted of $N_u = 52000$ data-points of the form $\{x_i, y_i, t_i, u_i^*\}_{i=1}^{N_u}$ randomly sampled in the domain and on the boundaries. The data-set is similar to what is shown in figure 6.5, but with collocation points replaced with data-points. Again we only use about 0.3% of the available data. Figure 6.8 and 6.9 show the results given by the PINN after training terminated. Despite giving the model ample data, we see that the Baseline-PINN very poorly regresses the solution, and the unknown parameters λ_i do not converge to their expected values at all.

Baseline-PINN With the Dense Data-set					
N_u	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_u = 52000$	λ_1	-0.7460	-1.4000	47%	$E_{Rel} = 55\%$
	λ_2	0.4043	-1.0000	140%	
	λ_3	-0.0121	1.0000	101%	

Table 6.2: Results generated by the Baseline-PINN versus the true target values for the unknown parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

Despite the loss decreasing to the lower half of 10^{-5}, we see that the the parameters λ_i wander around aimlessly without ever converging to their expected values. Additionally by looking at figure 6.9, we see that the reconstructed solution u was very inaccurate. These sorts of issues were the main setback we faced when working with PINNs on this thesis. This failure was observed regardless of hyper-parameter tuning and massaging of data. The PINN failed to produce the expected results no matter what was done to mitigate the issue.

We spent a large amount of time conducting a countless amount of independent tests, implementing common fixes for convergence problems with neural networks. For each test we would vary hyper-parameters, change training algorithms, vary network architecture and try different activation functions.

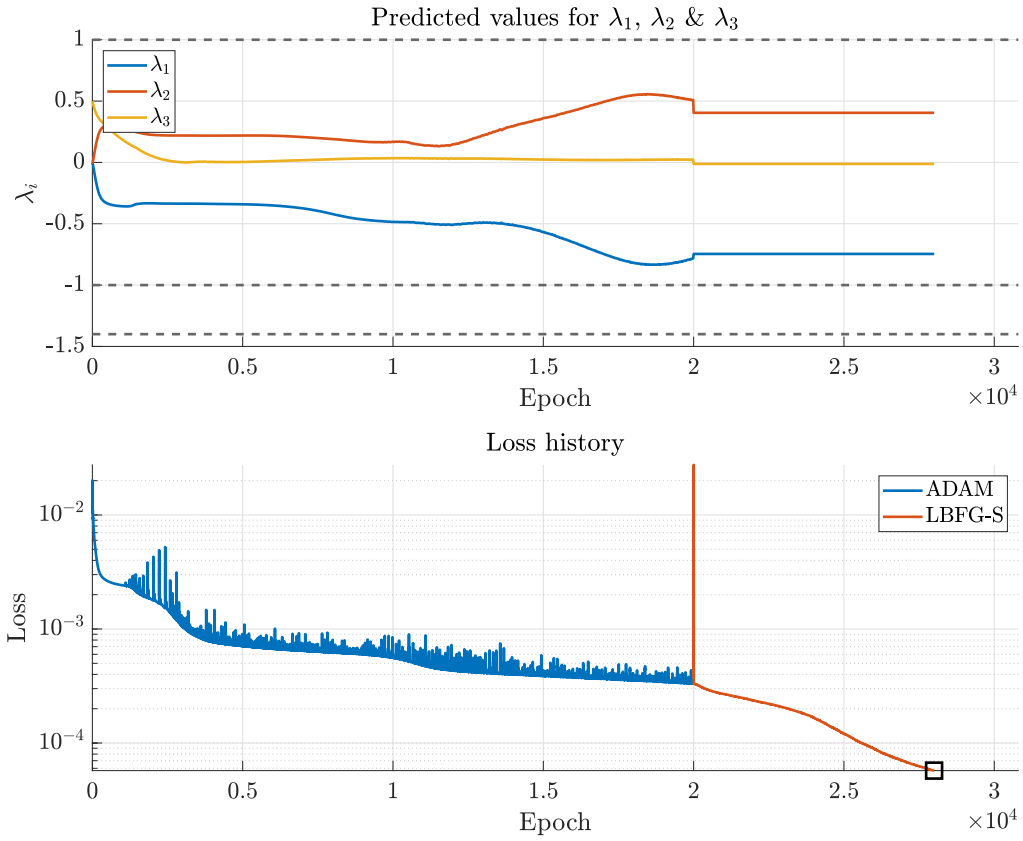

Figure 6.8: Training histories for the Baseline-PINN on dense data in the inverse case for the problem specified by equation (1.9). Top: The predicted values for λ_1, λ_2 and λ_3 over each epoch of training. Bottom: The Objective function loss over each epoch of training. Lowest achieved loss given by the black square.

Among these changes, we attempted to expand the network to contain more layers and more nodes. This change did not improve performance, and only extended training time considerably. The fact that this did not help makes sense. We already know that the network as implemented in section 5.2 and discussed in 4.3 should be expressive enough to convey the solution. We know this as we have shown that the PINN can solve the forward case in section 6.2.1. As such, the problem should not come from an insufficiency in the network.

We also tried training for considerably longer, but unfortunately there seemed to be a hard limit to the fit regardless of how much the network was trained. The loss would stagnate at a certain point, and then never improve. We also experimented with other training algorithms such as regular gradient descent, SGD, Adadelta and AdaMax. However, ADAM and L-BFGS remained the best performing optimisation algorithms out of all of them. Another aspect that was tried, was to increase the amount of training data significantly. This would also only serve to increase training time, with no benefit to performance.

Mini-batch training as discussed in chapter 4 was also an avenue we explored. At a

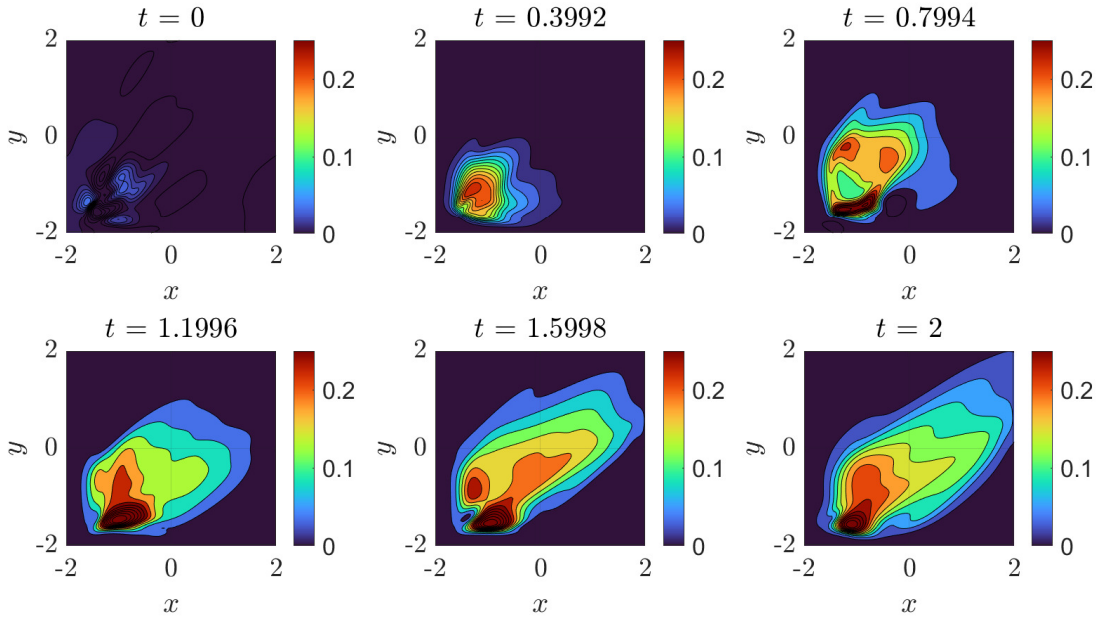

Figure 6.9: The predicted solution u generated by the Baseline-PINN of the 2D ADE as specified by equation (1.9) at different snapshots in time.

brief period of time, we thought that perhaps we were getting stuck in local minima. However, mini-batch training also failed to produce any good results. Finally, we investigated different activation functions. For example, we tried *Swish* [44], as seen in figure 4.2. This activation function has gathered interest from machine learning researchers as it seems to perform as well or better than *ReLU* for some problems [44]. As the *Swish* function is continuous and infinitely differentiable, we thought that we may leverage the form of *ReLU*-like activation functions with PINNs. However, again very little to no improvement was observed. Ultimately, none of our tests would give any satisfying results.

We also considered the possibility of there being some hidden bug in our code, which would hinder the Baseline-PINN from working correctly. Sections 6.1.1 and 6.1.2, were meant to discredit this possibility, but we felt that perhaps something happened in the transition from 1D to 2D. To ensure that the problem was not with a bug, we decided to create two separate codes implemented with DeepXDE [26, 27] and SciANN [15, 16] respectively. We wanted to avoid using these libraries as they act closer to black boxes, but they would rule out any coding misconducts on our end. Again, we found that the Baseline-PINN was simply unable to solve for the inverse 2D ADE on the same data-set. As such, we figured that there has to be a limitation to how far the Baseline-PINN can take us on this specific problem.

Note that for every test we conducted for this case, the results would vary by a large amount, even between runs with same hyper-parameters. As such, figure 6.8 and 6.9 only show one representative example of the failures we experienced. The specific example we have highlighted show a failure for which the regressed solution is poor and the unknowns λ_i wander aimlessly. We could for example also experience that the

λ_i values would grow in the positive or negative directions endlessly until training was concluded.

Chapter 7

Going Beyond the Baseline-PINN

Chapter 6 shows that the Baseline-PINN we implemented worked as expected for the 1D forward, 1D inverse and the 2D forward problems. But the 2D inverse problem remained unresolved, despite pursuing various changes and tests. We now will discuss methods which may elevate performance beyond what the Baseline-PINN can provide.

7.1 Improving Performance by Mitigating Gradient Pathologies With the Weighted-PINN

The major issue that has yet to be addressed, is the scaling of the objective function. Specifically the u_m and f_m parameters in the objective loss functions (3.5). *Raissi et al.* [43] had set these parameters to ones, which therefore assumes that the residual and supervised parts of the loss contribute equally to training.

Once we had run out of things to try, we contacted one of the developers of DeepXDE, *Lu et al.* [26, 27] for some advice. We were told that the relation between u_m and f_m is critically important for some PDEs. In all test cases up until this point, we had set $u_m = f_m = 1$ as was done for the Baseline-PINN in [43]. We were informed however, that the loss should ideally be scaled such that each of the two parts of the loss (3.4) and (3.3) contribute equally to training. If one part dominates heavily, then the PINN will not be training efficiently as it will only conform to the contributions from one part of the loss.

With this insight in mind, we took the naive approach of brute force. We conducted a large number of trial runs, varying f_m and u_m until the PINN could consistently solve for the inverse problem in section 6.2.2 accurately. By testing many different combinations of values, we settled on $u_m = 10.0$ and $f_m = 0.01$ as suitable loss weights. This particular setup of parameters seemed to strike a good balance between the two objective function terms in (3.5). This choice consistently lead to accurate results, even when varying the problem setup, such as source location or the velocity fields. Figure 7.2 and 7.1 shows that this Weighted-PINN with $u_m = 10.0$ and $f_m = 0.01$ can now solve for the inverse problem as specified by (1.9). As we see, the solution is accurately regressed and all three parameters λ_1, λ_2 and λ_3 now converge onto their expected target values. Note that we are dealing with a dense data-set as described in section 6.2.2.

Table 7.1 shows how the predicted values for the three unknowns λ_1, λ_2 and λ_3 compare with their true target values. We see that the predicted values turn out to be very accurate. Furthermore the relative L^2-norm error was 1.3%.

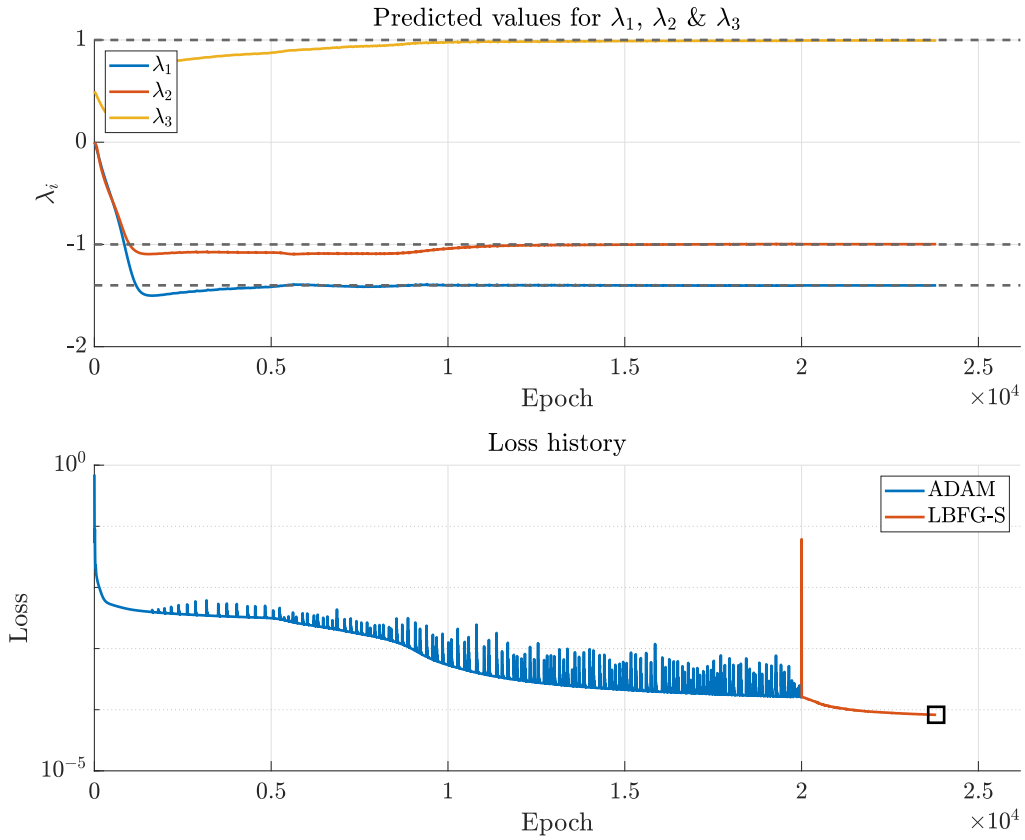

Figure 7.1: Training histories for the Weighted-PINN on dense data in the inverse case for the problem specified by equation (1.9). Top: The predicted values for λ_1, λ_2 and λ_3 over each epoch of training. Bottom: The Objective function loss over each epoch of training. Lowest achieved loss given by the black square.

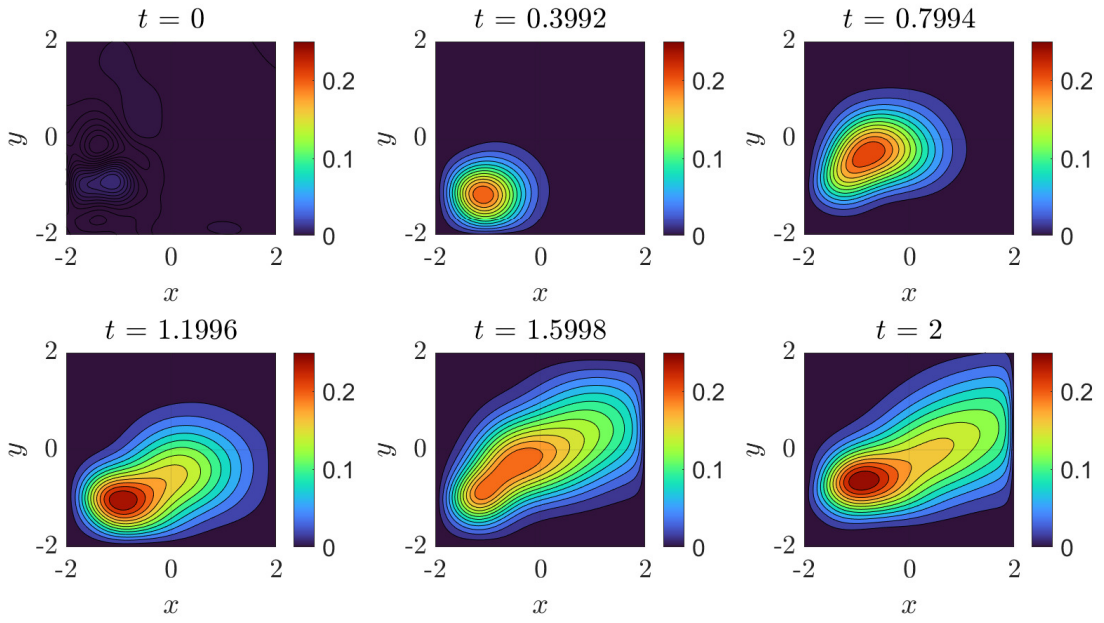

Figure 7.2: The predicted solution u generated by the Weighted-PINN of the 2D ADE as specified by equation (1.9) at different snapshots in time.

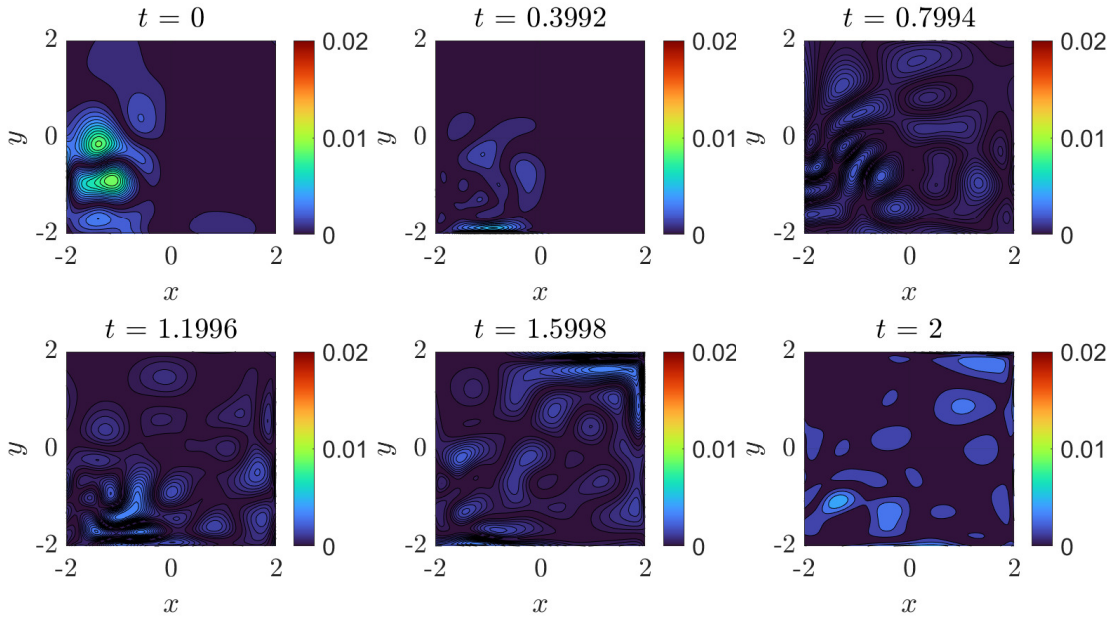

Figure 7.3: Absolute point-wise error between the predicted solution u generated by the Weighted-PINN and the reference solution u^* for different snapshots in time.

Weighted-PINN With the Dense Data-set					
N_u	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_u = 52000$	λ_1	-1.4010	-1.4000	0.071%	$E_{Rel} = 1.3\%$
	λ_2	-0.9975	-1.0000	0.25%	
	λ_3	0.9951	1.0000	0.49%	

Table 7.1: Results generated by the Weighted-PINN versus the true target values for the unknown parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

7.2 Automatic Weighting With the Adaptive-PINN

We have now shown that the PINN method is indeed able to solve for the inverse 2D ADE as given by (1.9), but only after careful tweaking of u_m and f_m in (3.5). This only further validates our claim in chapter 4. As it turns out, the original formulation of PINNs as described by *Raissi et al.* [43] seems to be insufficient to solve this specific problem according to our tests. As *Wang et al.* in [49] suggests, this may be because of improper scaling of the two variables u_m and f_m. In fact, they categorise this inherent issue with PINNs as a gradient pathology. They showed that in their test cases, as training progresses, the gradients $\nabla_\theta \mathcal{L}_u$ tend to vanish. In other words, they tend towards zero and thus do not contribute to any training benefits. This is critical as $\mathcal{L}_u$ regularises the network to obey uniqueness of the solution. And for inverse problems, if we are not letting the data regulate the solution, then we are essentially solving for an ill-posed PDE and we have no chance of finding the true values λ^*.

One of the main issues is that proper scaling required for convergence is problem specific. Such that different PDEs or even the same PDE with different parameters may require a different set of scaling values for good convergence. Therefore, there is no one-size-fits-all set of parameters which will guarantee sufficient results. Without any guiding principles on how to tune these parameters for good convergence, one will run into the issue of having to tune the parameters f_m and u_m with trial and error until good convergence is achieved, as was done in section 7.1.

This is even more of a problem when we have no prior knowledge of what the solution is supposed to be. We tuned our loss parameters until the PINN converged to the true solution, but if we do not know the solution prior, then we will have no idea whether or not the PINN is predicting accurately. This issue has proven to be a major setback to the PINNs method, and much of the contemporary research on PINNs has been put into identifying and mitigating convergence problems [3, 24, 30, 31, 39, 49, 50].

In addition, one may have noticed so far that the loss histories for the ADAM regime exhibit oscillatory behaviour, as shown in figure 6.3 and 7.1. This is not be due to stochasticity, because we are indeed using full-batch gradient descent and a dense data-set here. We wondered for long time what the cause of this behaviour was. As it turns out, this comes from the way the loss is formulated with PINNs, and how the gradient descent algorithm steps forward in the parameter space θ. Specifically it is the residual loss $\mathcal{L}_f$ which gives us trouble. *Wang et al.* looked at the Hessian of the residual loss

(3.3), and they found that it may develop a large gap between the largest and smallest eigenvalues, leading to the gradient descent algorithm essentially becoming a stiff ODE [3, 49]. The gradient descent algorithm is essentially a forward Euler ODE stepping algorithm, and it is known that forward Euler is inherently unstable for stiff problems [10].

Hence, even with full-batch gradient descent, the gradient descent step is no longer guaranteed to decrease the loss monotonically, unless we use a sufficiently small learning rate. Therefore, the optimiser can get trapped in limit cycles [3]. This is most likely why we are observing oscillations in the loss here as well, as their loss histories look strikingly similar to ours, with an oscillatory nature. This finding seems to be corroborated by *Krishnapriyan et al.* [24], as they found that the loss landscape for PINNs can become very noisy and chaotic, and in such a case, the optimiser may struggle to find a suitable optima.

Wang et al. [49] indeed found that the tuning of u_m and f_m was very important for mitigating these issues and for improving the convergence and accuracy of PINNs. Inspired by how ADAM adaptively tunes the learning rate, they developed an algorithm to dynamically tune u_m during training at regular intervals using an exponential walking average.

As mentioned earlier, the main problem is that the gradients of $\nabla_\theta \mathcal{L}_u$ vanish and training becomes dominated by $\nabla_\theta \mathcal{L}_f$. Their algorithm aim to scale $\mathcal{L}_u$ in such a manner that these gradients are approximately on the same order throughout training. This is essentially a way to dynamically change the learning rates between the two loss terms to optimise training. This automatic weighting scheme is meant to mitigate issues above, and should in theory let us avoid tedious manual tweaking and trial and error procedures.

Unfortunately, the algorithm they provided comes in two different forms. We will describe both of them here. First, they set $f_m = 1$ in equation (3.5), then the intermediate value $\hat{u}_m$ is computed by either equation (7.1) or (7.2),

$$\hat{u}_m = \frac{\mathbf{max}(|\nabla_\theta \mathcal{L}_f|)}{\mathbf{mean}(|\nabla_\theta \mathcal{L}_u|)}, \tag{7.1}$$

$$\hat{u}_m = \frac{\mathbf{max}(|\nabla_\theta \mathcal{L}_f|)}{\mathbf{mean}(|\nabla_\theta u_m^i \mathcal{L}_u|)}. \tag{7.2}$$

In equation (7.1) and (7.2) $\mathbf{max}(\cdot|)$ signifies the maximum absolute value, similarly $\mathbf{mean}(|\cdot|)$, signifies the arithmetic mean of absolute values. The u_m^i values are updated using an exponential walking average as shown in equation (7.3), where $\alpha \in [0,1]$ determines how strongly the update will depend on $\hat{u}_m$. We found that α was typically kept low to decrease the amount of stochasticity in u_m.

$$u_m^{i+1} = (1-\alpha)u_m^i + \alpha \hat{u}_m, \tag{7.3}$$

then the loss in iteration $i+1$ is given by the adaptive loss,

$$\mathcal{L} = \mathcal{L}_f + u_m^{i+1}\mathcal{L}_u. \tag{7.4}$$

Confusingly, *Wang et al.* are not consistent with their choice of the intermediate values $\hat{u}_m$. In their paper [49], the intermediate variable $\hat{u}_m$ was given as equation (7.1), while in their accompanying video presentation on the study [3], it is given by (7.2).

To add to the confusion, *Jin et al.* [21] and *Wang and Perdikaris* [48] refer to the algorithm provided in the paper (7.1) [49], but then write equation (7.2) in their own papers.

We suspect that this is perhaps a printing error in their paper or that perhaps we have an outdated version of the study. We examined the codes provided by [21, 48, 49], and found that equation (7.2) seems more consistent. We tried both versions, and found that the form in equation (7.2) also performed more consistently.

Therefore, when we refer to the Adaptive-PINN, we are referring to the PINN with the loss given by equation (7.4), with the updating scheme given by equation (7.3) and (7.2).

Regardless, this adaptive scheme has proved to be popular among PINN researchers trying to solve for PDEs for which the Baseline-PINN is not sufficient [21, 48, 51]. This adaptive scheme, as described above, has the advantage of finding the proper scaling automatically, we refer to it as the Adaptive-PINN. We will be implementing this Adaptive-PINN in chapter 8.

The arguments given by *Wang et al.* [48] give us a reason as to why our weighting of the parameters by $u_m = 10.0$ and $f_m = 0.01$ worked in section 7.1. The Baseline-PINN worked well enough for the 1D forward, 1D inverse and 2D forward case, but it seems that the 2D inverse case was the breaking point. By scaling these terms sufficiently, we were able to alleviate the issues of vanishing gradients and thus the PINN could converge onto the correct solution u in correspondence with the fed data u^*. This static weighting scheme been referred to static- or non-adaptive weighting in various literature [31, 51], but we refer to it as the Weighted-PINN.

7.3 Other Mitigative Training Schemes

In addition to adaptive schemes, there are other training schemes that have been proposed to alleviate the weaknesses of the Baseline-PINN.

NTK Weighting

NTK weighting is another adaptive scheme which aims to update both f_m and u_m in tandem. This algorithm was also developed by *Wang et al.* [50]. Unfortunately, at the time of writing this thesis, their codes were not available publicly and therefore we did

not get to implement this specific method. Their algorithm however, seems to be a further development of the scheme in [49]. *Wang et al.* claim that the previous method works well in practice, but it does not have theoretical justification. This new adaptive scheme however, comes with theoretical framework which shows how it may benefit convergence. We remain hopeful that this alternate adaptive scheme will perform even better than the one described above.

Self-Adaptive PINNs

McClenny and Braga-Neto propose a method for which each collocation- and data-point are assigned a trainable weight Λ_i that is to be optimised and tuned during training. These weights will behave similarly to the scalars u_m and f_m, but here they are unique to each training point. The minimisation problem in equation (3.6) can obviously be solved by letting $\Lambda_i = 0$, but this is a trivial solution. As such, they instead propose a min-max procedure for which the loss $\mathcal{L}(\theta, \Lambda)$ is to be minimised with respect to θ, but maximised with respect to Λ. This forces the PINN to train more heavily on points with high loss and less so on points with lower loss.

Curriculum Training

Krishnapriyan et al. in [24] suggested that PINNs train better by introducing the network to simpler problems first. For instance, they showed that if one wished to solve the 1D advection equation in the forward sense, it became increasingly difficult for the PINN to accurately predict the solution as the advection rate increased. This corresponds with increasing absolute values of $\mathbf{v}$ in equation (1.7).

They demonstrated that this problem could be alleviated by incrementally increasing the advection rate during training until the desired rate was reached. They referred to this type of training as **curriculum training**. This scheme proved successful at improving the performance of PINNs in the forward sense.

Seq2seq Training

Krishnapriyan et al. also suggested a **seq2seq** training scheme, which is based on dividing the training data into discrete segments in time. Then the PINNs are trained on data from each segment individually. This is opposed to how the Baseline-PINN is generally trained, where the PINN learns the solution over the entire domain at the same time.

Krishnapriyan et al. argue that PINNs perform poorly at solving time-dependent problems over larger time-scales when trained with the whole data-set at once. This is something that we can also attest to in our studies. Typically, neural networks perform better when fed more data, thus we had hoped we could gain higher accuracy when we increased the time horizon as this would only increase the amount of available data to the PINN. However, counter-intuitively, we once again suffered poor performance in this case, similarly to what is shown in section 6.2.2.

As *Krishnapriyan et al.* suggests, PINNs perform better over shorter time-scales, so if one wants to use PINNs over a larger time-scale, it may help to train it using their suggested **seq2seq** scheme. *Wight and Zhao* [51] propose an alternative version of **seq2seq** in which one uses an ensemble of PINNs, each of which specialise on one specific segment in time, for which the initial data for the next PINN is based on the final predictions made by the previous PINN.

Adaptive Re-sampling

In a similar vein, there are yet other methods that attempt to elevate the performance of the Baseline-PINN. For example, adaptive re-sampling. *Wight and Zhao* [51] proposed a method in which one adaptively re-samples the data-points based on the magnitude of the residual loss (3.3). As such the PINN will train more heavily in areas with high error. This method showed to be effective on the Allen-Cahn equation, but it does increase the amount of data during training and thus increases computation time.

Chapter 8

Results

Now that we know several strategies to improve the performance of PINNs, we employ some of them on the remaining 2D inverse problems.

8.1 The Adaptive-PINN on the 2D Inverse Problem With Dense Data

We decided to put the Adaptive-PINN as described in section 7.1 to the test. We set $f_m = 1.0$, and then let u_m be adaptively updated for every training epoch with ADAM using the equation (7.3) with $\alpha = 0.01$. This choice of α smoothens the update history and reduces the amount of noise.

The data-set and all other relevant hyper-parameters were left the same as the previous case in section 7.1. The data-set consists of N_u data-points and they are densely distributed, similar to what is shown in figure 6.5. Figure 8.1 shows the prediction histories of the three unknown parameters λ_1, λ_2 and λ_3, and the corresponding loss over each epoch of training. We can see that the Adaptive-PINN is also able to discover the unknown parameters. Figure 8.2 shows the reconstructed solution given by the PINN. Figure 8.4 shows how u_m changes with each epoch of training. Table 8.1 summarises the results.

Again the predicted results are very accurate, we end up with a relative L^2-norm error of 1.1%. The results mostly remain the same as shown in section 7.1, with a marginal increase of accuracy in the L^2-norm error and λ_3, but a marginal decrease for λ_1 and λ_2. With the variance that we get in-between runs however, we can assert that this discrepancy is well within the margin of error.

The Adaptive-PINN has shown to be able to converge onto the expected solution on its own, without any manual tweaking of f_m and u_m. Additionally, as shown in the bottom panel of figure 8.1, the adaptive scheme seems to have reduced the oscillatory behaviour of the loss which was prevalent in the Weighted-PINN 7.1. This is likely due to the fact that the PINN now finds the optimal loss scaling to optimise training and consequently reduces the stiff behaviour of the gradient descent optimiser as discussed in [3, 49].

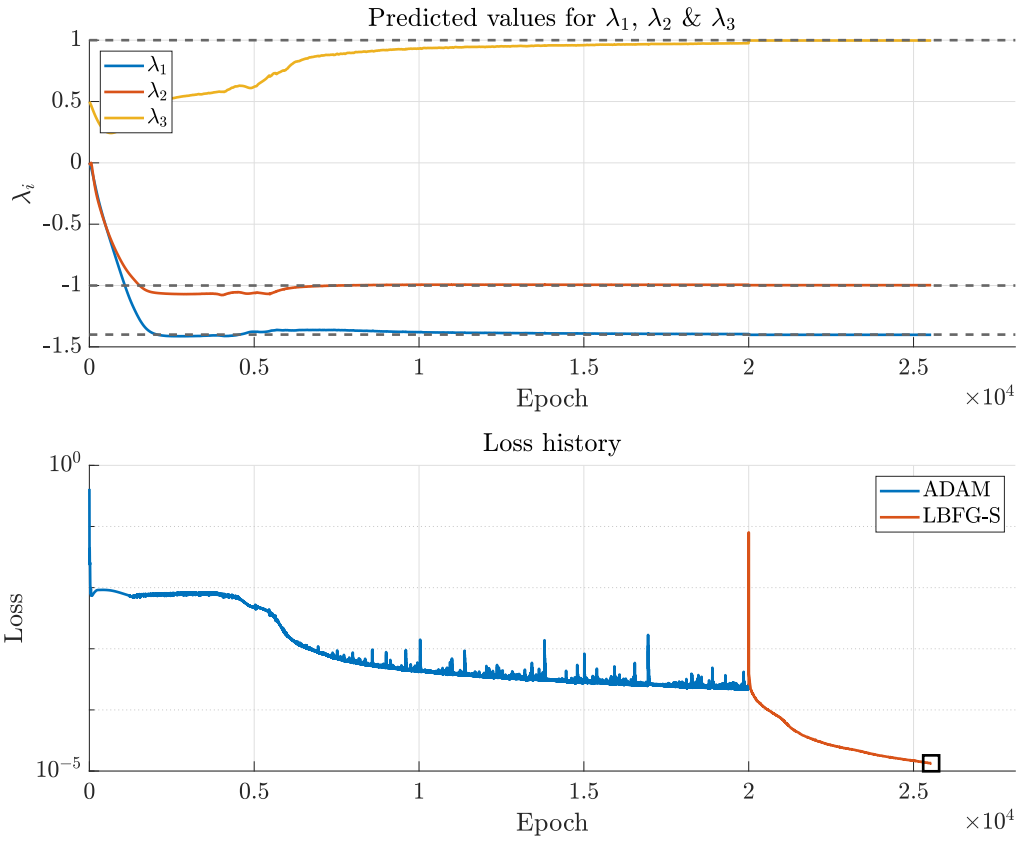

Figure 8.1: Training histories for the Adaptive-PINN on dense data in the inverse case for the problem specified by equation (1.9). Top: The predicted values for λ_1, λ_2 and λ_3 over each epoch of training. Bottom: The Objective function loss over each epoch of training. Lowest achieved loss given by the black square. Notice how loss decreases significantly once LBFG-S training is engaged.

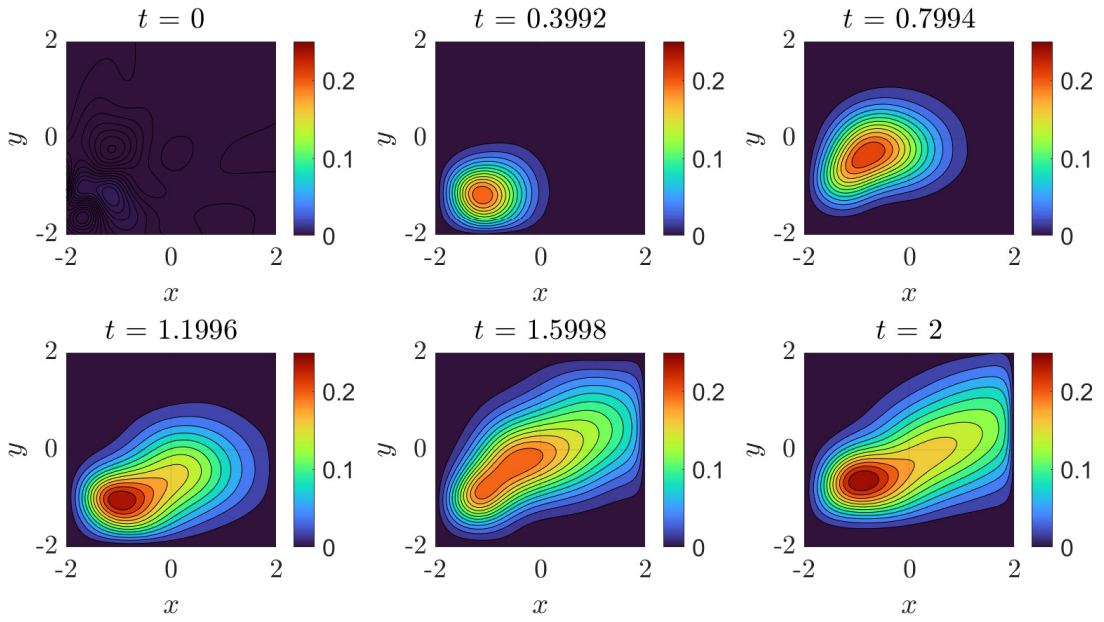

Figure 8.2: The predicted solution u generated by the Adaptive-PINN of the 2D ADE as specified by (1.9) at different snapshots in time.

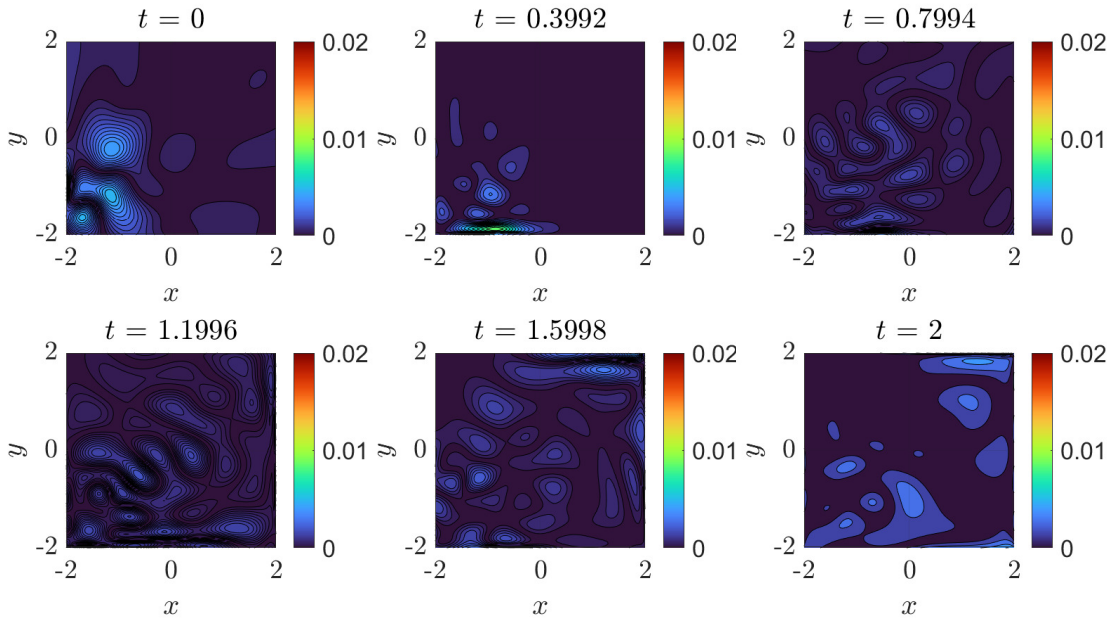

Figure 8.3: Absolute point-wise error between the predicted solution u generated by the Adaptive-PINN and the reference solution u^* for different snapshots in time.

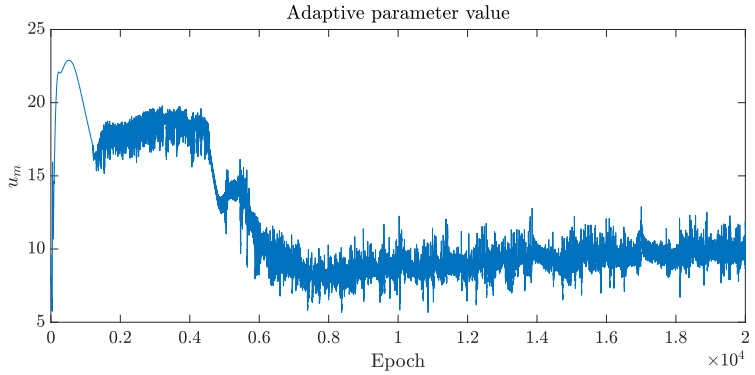

Figure 8.4: The history of the adaptive parameter u_m over each epoch of training with ADAM.

Adaptive-PINN With the Dense Data-set					
N_u	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_u = 52000$	λ_1	-1.4017	-1.4000	0.13%	$E_{Rel} = 1.1\%$
	λ_2	-0.9970	-1.0000	0.30%	
	λ_3	0.9973	1.0000	0.27%	

Table 8.1: Results generated by the Adaptive-PINN versus the true target values for the unknown parameters λ_1 λ_2 λ_3 of the 2D ADE as specified by (1.9).

8.2 The Weighted-PINN on the 2D Inverse Problem With Sparse and Noisy Data

The results from sections 7.1 and 8.1 look promising, but ultimately the results are not particularly realistic for most applications. For the previous inverse problems 6.1.2, 6.2.2, 7.1 and 8.1, we have been required to gather a dense cloud of N_u data-points of the form $\{x_i, y_i, t_i, u_i^*\}_{i=1}^{N_u}$ which covers the entire space-time domain as shown in figure 6.1 and 6.5. This may not be sensible for some situations. A more realistic scenario would be if we had a coarse set of measurement stations which probe the solution u^* at set coordinates continuously in time. In other words, if we have a set of time-series measurements at a set of discrete locations, can we still reconstruct an accurate global solution and predict the location of the source?

This section demonstrates how the PINN performs on this more realistic scenario. Again we want to discover the source pollutant its strength. However, now there are a limited number of measurement stations that are installed. We assume that the measurement stations remain stationary in space, and that they measure the pollutant u^* continuously in time. We also assume that we know the velocity $\mathbf{v}$, at the stations. In the following example $N_m = 26$ measurement stations were placed in a near uniform staggered grid pattern in the domain, as shown in figure 8.5. This number of stations N_m including boundary- and initial-data constituted approximately 62500 data-points.

The distribution of measurement stations is almost uniform, and as such we imply no prior knowledge of the true solution u^* and the true source location. Moreover,

Gaussian noise of 5% was added to the data u^*. More specifically, the time-series of measurements gathered from each measurement station is shown in blue in figure 8.7. Note the change in u-axis range for each plot, which makes the noise look substantially larger for some of them. It is natural that some measurements are dominated by noise and are not as useful for training. This may happen whenever stations are too far away from the source to receive any significant signal, as we can see is the case for station 20, 23 and 26.

This problem-setup gives the trains the PINN only on the values of u^* at a set number of locations in the $\mathbf{x} \in \Omega$ space. This will present the PINN to the challenge of accurately extrapolating the solution on the untrained areas between the stations.

Despite the seeming success of the Adaptive-PINN in section 8.1, we were not able to solve the inverse 2D ADE reliably with the Adaptive-PINN for this sparse data-set, we again got unsatisfactory results similar to those shown in figure 6.8 and 6.9. Thus, we returned to the Weighted-PINN as described in section 7.1 here, as we found that it performed more reliably for these problems.

Figure 8.6 shows the prediction histories of the three unknowns λ_1, λ_2 and λ_3 and the corresponding loss over each epoch of training, while figure 8.8 shows the reconstructed solution u given by the PINN. We can see that the three unknown parameters quickly converge close to their true values. The predicted solution u looks accurate to the true solution; however, it is clear that the PINN is not as accurate as in the dense case. The relative L^2-norm error was 7.0%, significantly larger than in the dense case as shown in table 7.1 and 8.1, despite having a similar number of training data.

This is likely due to the fact that the PINN is not trained on spatial locations in-between the measurement stations. As such, the PINN has to extrapolate over large distances. But, we do expect the solution u to be accurate at or near the measurement stations. This is evident by looking at figure 8.7. We see that the PINN very accurately reproduces the underlying signal that is received at each of the stations. Thus, we know that the predicted solution u is at its most accurate near where measurements were taken.

We see that despite there being a clear degradation in accuracy of u, the model still accurately discovers the unknown variables λ_1, λ_2 and λ_3. The final results are shown in table 8.2.

Weighted-PINN With the Sparse Data-set					
N_m	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_m = 26$	λ_1	-1.3627	-1.4000	2.7%	$E_{Rel} = 7.0$
	λ_2	-0.9945	-1.0000	0.55%	
	λ_3	1.0791	1.0000	7.9%	

Table 8.2: Results generated by the Weighted-PINN versus the true target values for the unknown parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

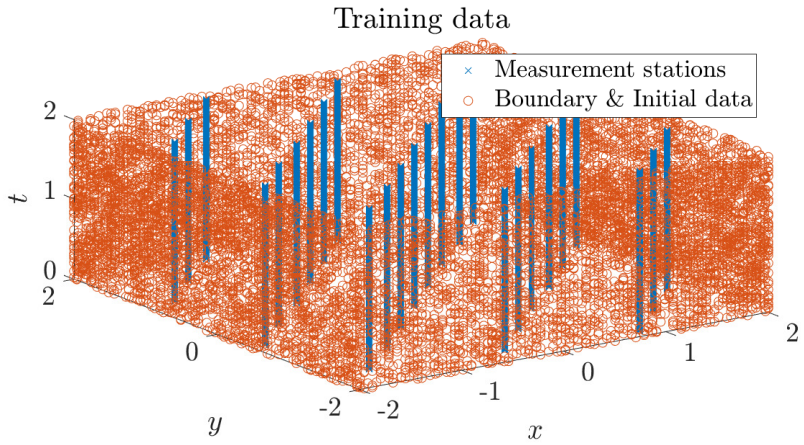

Figure 8.5: Sparse training data. Orange dots represent data-points sampled from boundary and initial conditions. Blue crosses represent data-points from the $N_m = 26$ measurement stations. Measurement stations lie on a near uniform staggered grid.

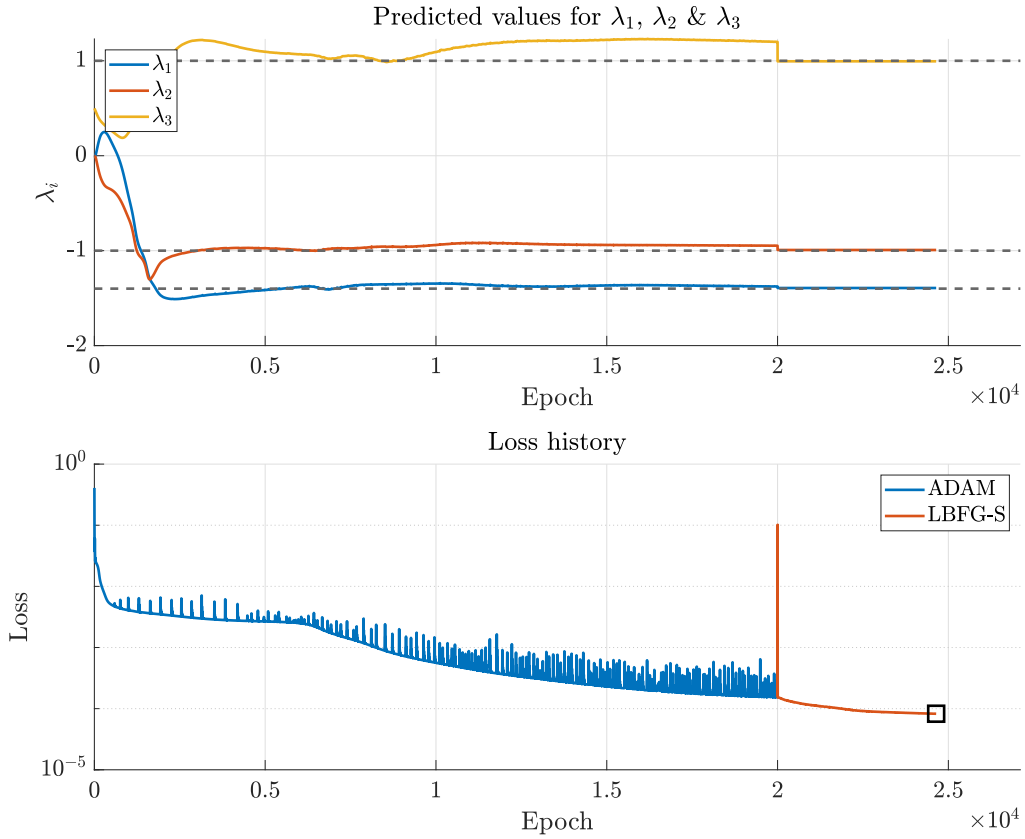

Figure 8.6: Training histories for the Weighted-PINN on sparse data in the inverse case for the problem specified by equation (1.9). Top: The predicted values for λ_1, λ_2 and λ_3 over each epoch of training. Bottom: The Objective function loss over each epoch of training. Lowest achieved loss given by the black square.

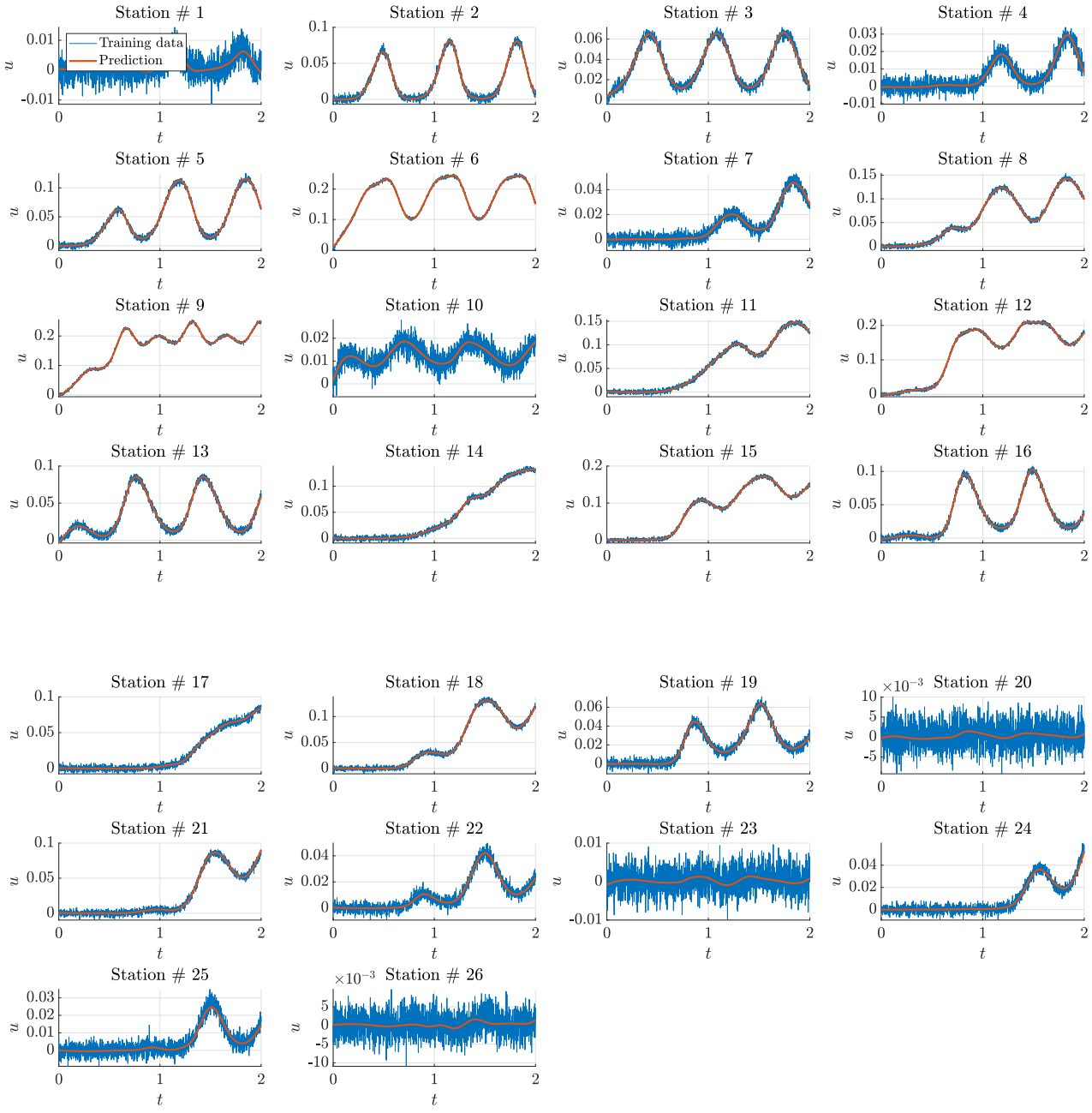

Figure 8.7: The predicted solution u generated by the Weighted-PINN versus the noisy training data u^* given as time-series from measurement stations.

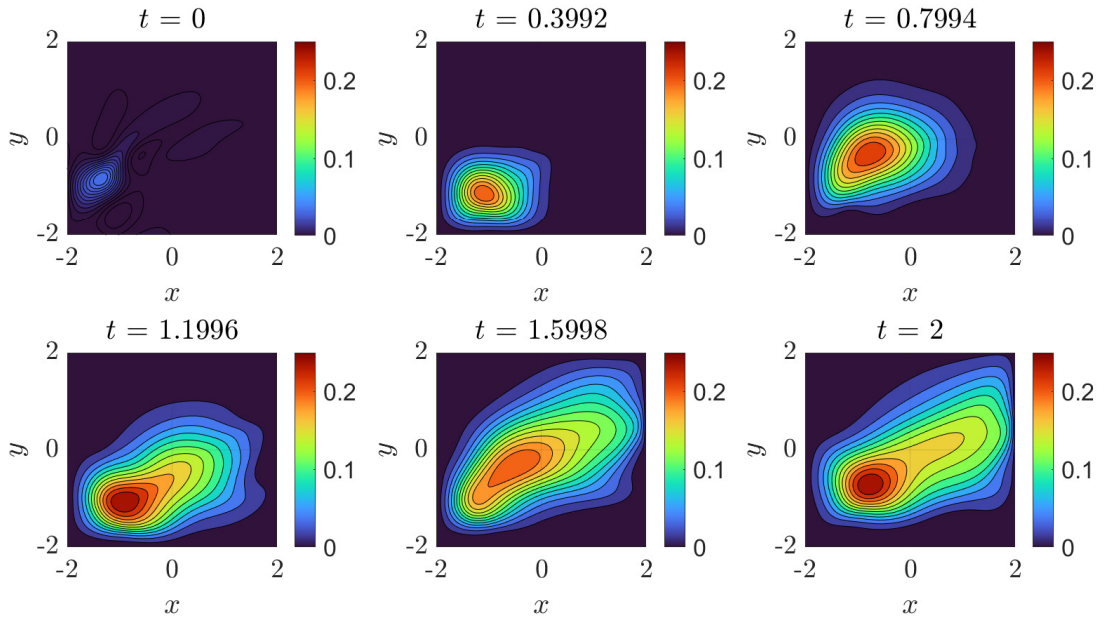

Figure 8.8: The predicted solution u generated by the Weighted-PINN of the 2D ADE as specified by (1.9) at different snapshots in time.

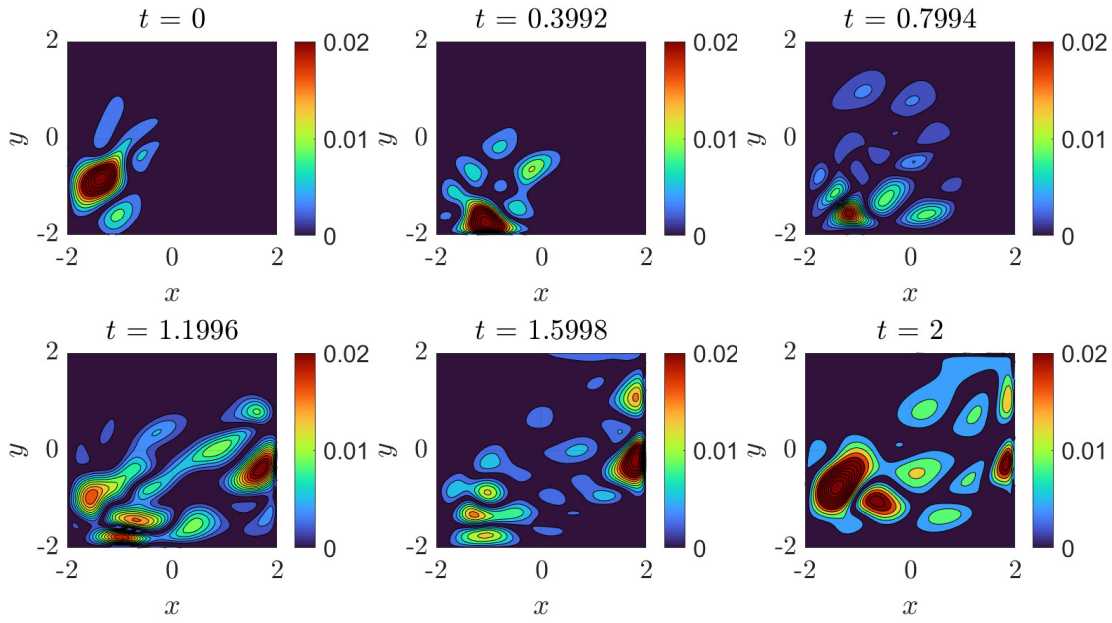

Figure 8.9: Absolute point-wise error between the predicted solution u generated by the Weighted-PINN and the reference solution u^* for different snapshots in time.

8.3 The Weighted-PINN on the 2D Inverse Problem With Randomly Distributed Sparse and Noisy Data

This test-case is the similar to the previous one in section 8.2. However, now the measurement stations are randomly placed in the domain. As we will see, the Weighted-PINN still managed to converge to a reasonable result. The data-set is shown below in figure 8.10.

In figure 8.12 we see that in comparison to the previous runs in sections 7.1 8.1 and 8.2, the Weighted-PINN does not reconstruct the solution particularly accurately with this data-set. There is a clear degradation of accuracy in u when the measurement stations are placed randomly as the L^2-norm error is 19%. Nevertheless, it does accurately discover the parameters λ_i we were looking for. This shows that we need not be very particular in where we place our measurement stations, as it still performs reasonably with a random distribution.

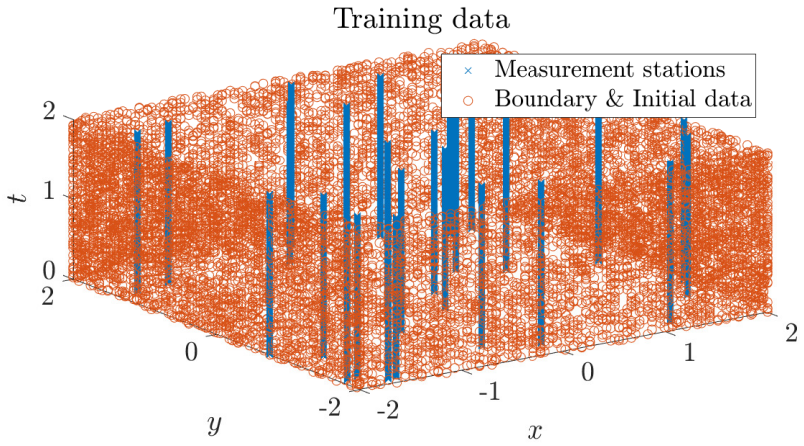

Figure 8.10: Sparse and randomly distributed training-data. Orange dots represents data-points sampled from boundary- and initial-conditions. Blue crosses represents data-points from the $N_m = 26$ measurement stations. Measurement stations lie randomly placed in the spatial domain.

Weighted-PINN With the Sparse and Randomly Distributed Data-set					
N_m	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_m = 26$	λ_1	-1.3964	-1.4000	0.26%	$E_{Rel} = 19\%$
	λ_2	-0.9989	-1.0000	0.11%	
	λ_3	0.9883	1.0000	1.2%	

Table 8.3: Results generated by the Weighted-PINN versus the true target values for the parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

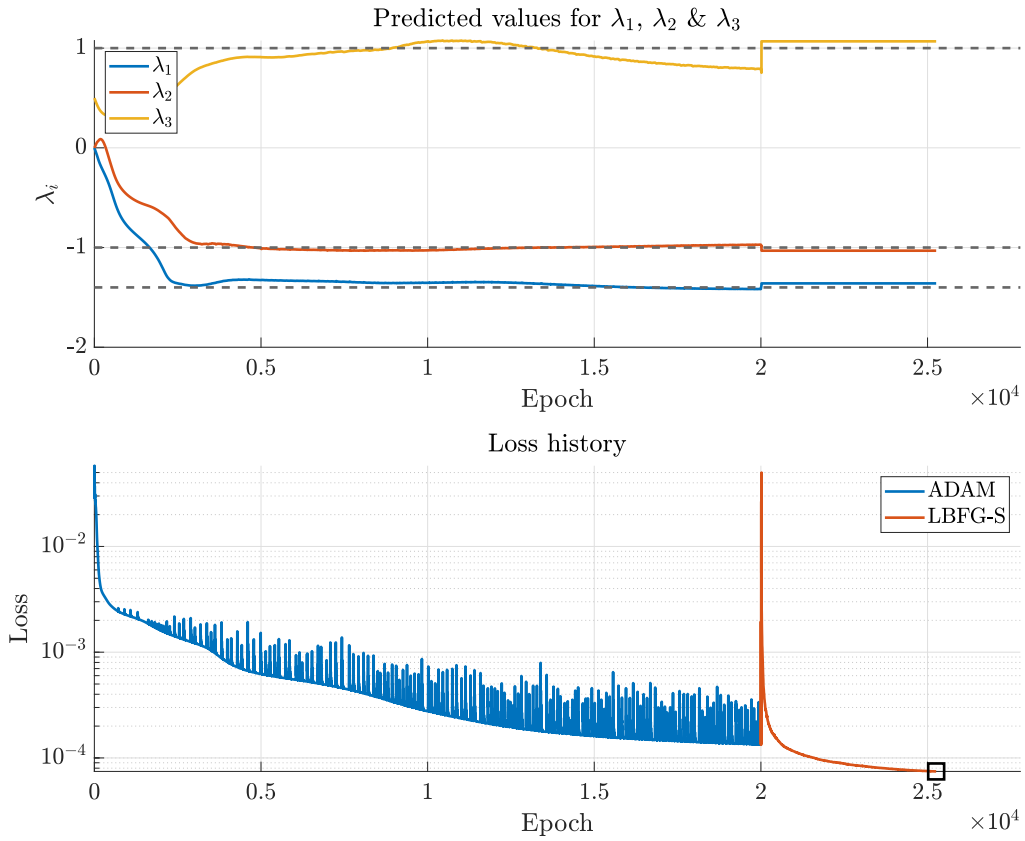

Figure 8.11: Training histories for the Weighted-PINN on sparse and random data in the inverse case for the problem specified by equation (1.9). Top: The predicted values for λ_1, λ_2 and λ_3 over each epoch of training. Bottom: The objective function loss over each epoch of training. The lowest achieved loss is given by the black square.

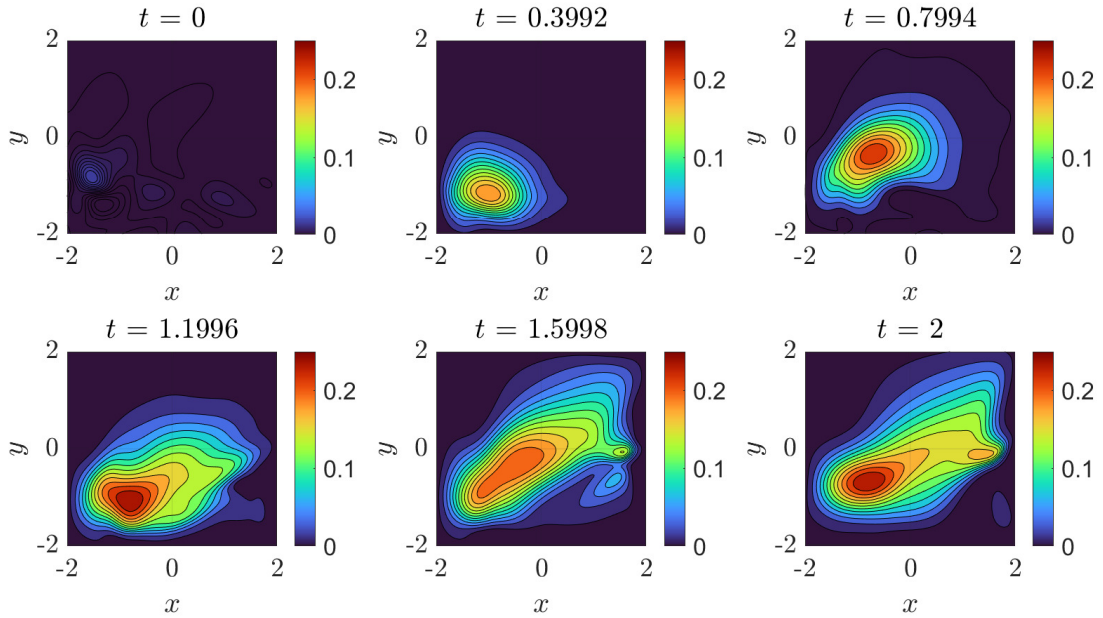

Figure 8.12: The predicted solution u generated by the Weighted-PINN of the 2D ADE as specified by (1.9) at different snapshots in time.

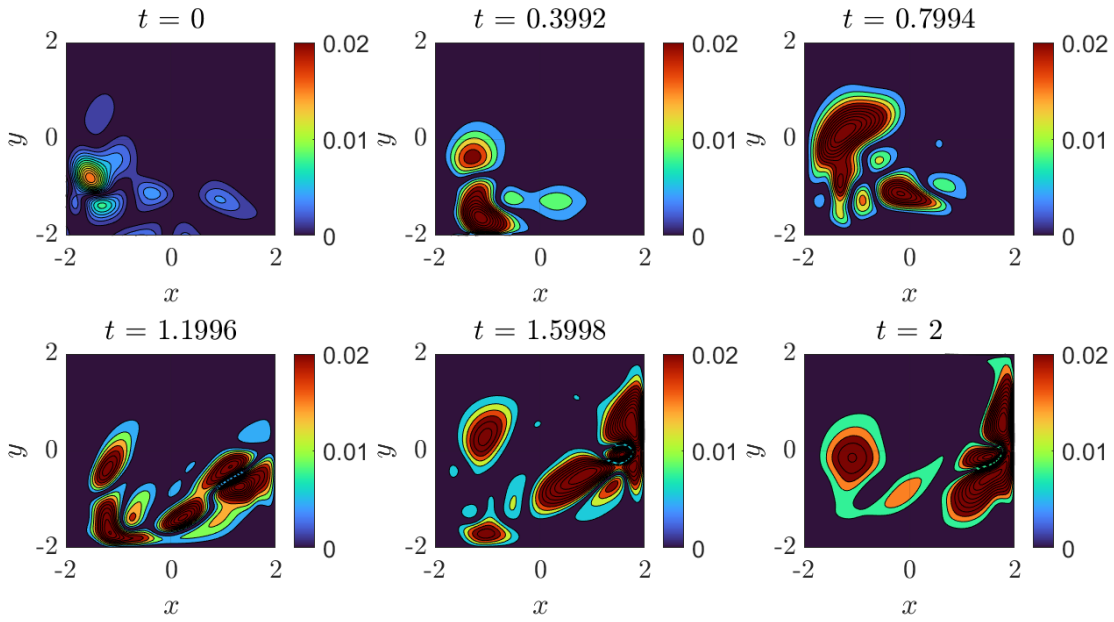

Figure 8.13: Absolute point-wise error between the predicted solution u generated by the Weighted-PINN and the reference solution u^* for different snapshots in time.

8.4 Stability With Respect to Initial Guesses

One thing that remains unexplored is the choice of initial guesses of λ_i. In all of the 2D test cases considered, the initial guesses were set at $P = [\lambda_1, \lambda_2, \lambda_3]^T = [0.0, 0.0, 0.5]^T$. This seems reasonable as the location corresponds to the middle of the domain.

Changing this initial guess and performing further tests revealed to us that the Weighted-PINN will again suffer from poor convergence if the initial guess is too far away from the true solution. This is not too much of a problem in this case, as the middle of the domain seems to be the most reasonable first guess to make in the absence of any data. However, if we are dealing with a larger or more complicated domain, this initial guess will be more important to determine beforehand.

We aimed to establish how reliable and stable the results in section 8.2 are. The initial guess for the two parameters λ_1, λ_2, determining location, were changed to different points in the plane while λ_3 was kept the same. Then several training sessions were run on the same sparse data-set as shown in figure 8.5.

If the result is reproducible and stable, we should expect most of the PINNs to converge near to the true solution. The traced paths of which each PINN take is shown in the left plot in figure 8.14. Figure 8.14 shows the paths each prediction took over each epoch. We see that the results are mostly stable with changes in initial guesses. Most of the initial guesses given by the circles eventually converge near the source. There were however three cases, coloured in bright red, which did not converge. Their final guesses did not end up anywhere near the true location. Therefore, it seems that the basin of attraction for the system is of finite size and one must choose a sufficient ini-

tial guess for the PINN to converge correctly.

Something that can be done to mitigate this issue, is to simply select the initial guess at the measurement station with the highest overall concentration. If the stations cover the domain sufficiently, then this measurement station is likely to be somewhat close to the source, and qualify for a good initial guess.

We consolidated all of the predicted values for $\lambda_{1,2,3}$ using the arithmetic mean. The variation of these predictions were computed by the standard deviation. Figure 8.15 shows the evolution of the all predictions over each training epoch, ignoring the three instances that did not converge. We see that when the Weighted-PINN is initialised properly, the predictions are fairly stable and we will end up with the same prediction with changes in initial guesses. We summarise the results in table 8.4.

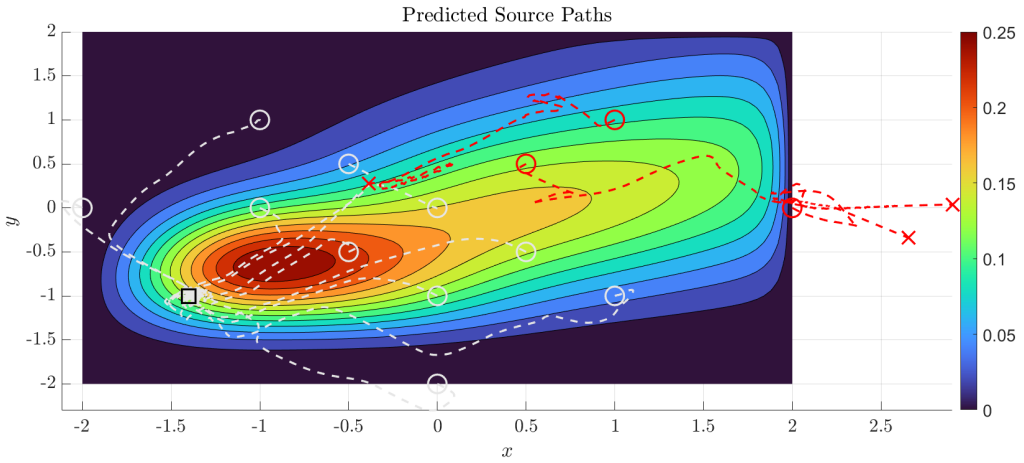

Figure 8.14: Predicted source paths over multiple runs with different initial guesses. Initial guesses shown by circles, final guesses given by the crosses and the true location given by the black square. The three runs in red represent runs that did not converge.

Weighted-PINNs With the Sparse Data-set				
N_m	Parameter	Predicted λ $(\mu \pm \sigma)$	True λ^*	Relative Error λ
$N_m = 26$	λ_1	-1.3896 $\pm$0.0088	-1.4000	$\pm$1.0%
	λ_2	-0.9854 $\pm$0.0060	-1.0000	$\pm$2.0%
	λ_3	0.9933 $\pm$0.0283	1.0000	$\pm$3.5%

Table 8.4: Results generated by the Weighted-PINNs versus the true target values for the parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

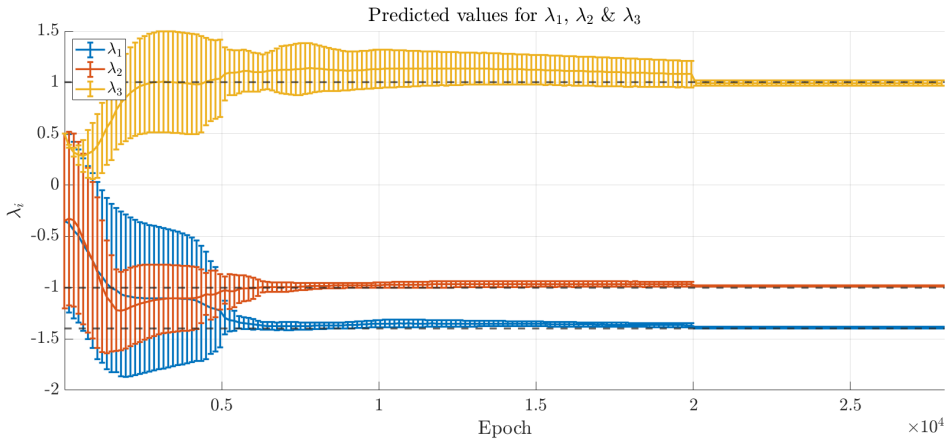

Figure 8.15: Consolidated histories for the three unknown parameters λ_1, λ_2 and λ_3. The standard-deviation for each prediction is given by the error bars. We see that the variation decreases with training as every PINN home in on the same values. Notice that when L-BFGS kicks in, the variation drops significantly.

8.5 Improving Performance Further and Reducing the Amount of Measurement Stations

In sections 8.2, 8.3 and 8.4 we took use of $N_m = 26$ measurement stations placed in the domain. This was the lowest number of stations we could achieve with the Weighted-PINN without the L^2-norm error becoming very high. But can we do better than this? This is still a very large number of measurements, and it becomes hard to justify PINNs in for practical purposes when the methods discussed in sections 1.6.2 and 1.6.3 suffice with less data.

Note that in sections 8.2, 8.3 and 8.4 we assumed that our stations measured the concentration of pollutant u^* but also the velocities $\mathbf{v}$ at that point. But what if we know the velocity-field globally in the domain? This may be feasible by using an independent Navier-Stokes solver or some extrapolation algorithm. As discussed in section 1.1, we assume that the pollutant is passive, so the solution u does not affect the velocity-field $\mathbf{v}$. Therefore, it can be solved for separately. Why does this matter? By looking at figure 8.8 and 8.12 we can see there is a clear degradation of the solution u in the sparse-data cases. Despite the PINN using the physics to extrapolate the solution on points it was not trained on, it struggles to do so with the little information it has available. If we could train the PINN on locations aside from only at the measurement stations, then perhaps we could improve performance on solution u and the predicted parameters λ_i.

We propose the following hybrid training scheme: We train the inverse PINN on the data-points of the form $\{\mathbf{x}_i, t_i, u_i^*\}_{i=1}^{N_u}$ given by the measurement stations, exactly as was done in section 8.2. However, now we also supplement the residual loss as described by equation (3.3) with additional collocation-points $\{\mathbf{x}_i, t_i\}_{i=1}^{N_f}$ which are randomly sampled in the domain. This does not require any additional information on u^*, we only require that we know $\mathbf{v}$ at the collocation-points. This is because these additional collocation-points will only be fed to the residual loss as given by (3.14), not the

supervised loss (3.15). As such, we are employing a hybrid data-set which is a mixture of sparse data-points and dense collocation-points, as such combining the ideas behind the forward and inverse PINN. With this idea in mind, we aim to lower the required number of measurement stations N_m and increase the accuracy in u and λ_i. Therefore, we hope to localise the source with even less data.

To this end, we employ the Adaptive-PINN once more. Recall that we mentioned in section 8.2 that the Adaptive-PINN could not solve for the data-set in figure 8.10. However, we will now show that with our proposed training scheme, the Adaptive-PINN was able to solve for the sparse data-set. For each test, we let N_m vary as $N_m = \{26, 13, 8, 5\}$. For the Adaptive-PINN we supplement the data-set with $N_f = 50000$ collocation-points as to make a hybrid data-set. The results for the Adaptive-PINN on the hybrid data-set are shown in table 8.7. For comparison, we show how the Weighted-PINN performs on the sparse data-set on the same number of measurement stations N_m, see table 8.5 for the results.

In table 8.5 we see that we have an acceptable predictive capacity on λ_i for $N_m = 26$ and $N_m = 13$, however as N_m lowers even further, the accuracy falters to the point of becoming useless. Additionally, the relative L^2-norm error as described by equation (5.1) on u is considerably high for all four scenarios, starting at 7.0% and peaking at an abysmal 73%.

As stated in section 8.2, we were unable to get the Adaptive-PINN to behave well with the sparse data-set. In table 8.6 we have the corresponding results for the Adaptive-PINN trained exclusively on the sparse data-set. We see that the errors are unsatisfactory across the board.

On the other hand, in table 8.7 we have the analogous results for the Adaptive-PINN trained on the hybrid data-set. We see that the predicted λ_i values have a relative error at or below 1% for every scenario except for $N_m = 5$ with a maximum of 5.2% error. In particular for when $N_m = 8$, the Adaptive-PINN only has only access to the solution u^* at 8 locations in space, yet it is able to predict the λ_i values with less than 1% error! In addition, the L^2-norm error remains below 3% for all scenarios except for the one outlier at $N_m = 5$. In fact, the L^2-norm error with $N_m = 5$ for the Adaptive-PINN on the hybrid data-set is lower for the $N_m = 26$ case for the corresponding Weighted-PINN.

These results show us that we can drastically reduce the amount of information on u^* to localise the source, while also improving performance as long as we can provide the PINN additional collocation-points to train on. In our case, this is only possible if we know the velocity-field $\mathbf{v}$ on the entire domain. This finding is very promising as it makes the PINN method seem more viable again, as $N_m = 26$ is not particularly practical, but $N_m = 8$ and $N_m = 5$ sound more reasonable. And furthermore, since this was accomplished on the Adaptive-PINN, we remain hopeful that changes in PDE parameters such as diffusion rate D, velocity-field $\mathbf{v}$, and leak size S in equation (1.7) will be accounted for automatically, without having to tediously tune f_m and u_m in equation (3.5) for proper convergence.

Weighted-PINN With the Sparse Data-set					
N_m	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_m = 26$	λ_1	-1.3627	-1.4000	2.7%	$E_{Rel} = 7.0$
	λ_2	-0.9945	-1.0000	0.55%	
	λ_3	1.0791	1.0000	7.9%	
$N_m = 13$	λ_1	-1.4024	-1.4000	0.17%	$E_{Rel} = 21\%$
	λ_2	-0.9799	-1.0000	2.0%	
	λ_3	1.0139	1.0000	1.4%	
$N_m = 8$	λ_1	-0.9937	-1.4000	29%	$E_{Rel} = 45\%$
	λ_2	-1.0534	-1.0000	5.3%	
	λ_3	0.5044	1.0000	50%	
$N_m = 5$	λ_1	-0.7216	-1.4000	48%	$E_{Rel} = 73\%$
	λ_2	-0.6645	-1.0000	34%	
	λ_3	0.6222	1.0000	38%	

Table 8.5: Results generated by the Weighted-PINN on the sparse data-set versus the true target values for the parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

Adaptive-PINN With the Sparse Data-set					
N_m	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_m = 26$	λ_1	-1.3465	-1.4000	3.8%	$E_{Rel} = 23\%$
	λ_2	-0.8348	-1.0000	17%	
	λ_3	0.4852	1.0000	51%	
$N_m = 13$	λ_1	-1.2476	-1.4000	11%	$E_{Rel} = 74\%$
	λ_2	-0.9614	-1.0000	3.8%	
	λ_3	1.3455	1.0000	35%	
$N_m = 8$	λ_1	0.6246	-1.4000	145%	$E_{Rel} = 64\%$
	λ_2	-0.0083	-1.0000	99%	
	λ_3	0.0215	1.0000	98%	
$N_m = 5$	λ_1	-0.7892	-1.4000	44%	$E_{Rel} = 70\%$
	λ_2	-0.6011	-1.0000	40%	
	λ_3	0.4228	1.0000	58%	

Table 8.6: Results generated by the Adaptive-PINN on the sparse data-set versus the true target values for the parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

Adaptive-PINN With the Hybrid Data-set					
N_m	Parameter	Predicted λ	True λ^*	Relative Error λ	Relative Error L^2
$N_m = 26$	λ_1	-1.4029	-1.4000	0.20%	$E_{Rel} = 2.1\%$
	λ_2	-0.9947	-1.0000	0.53%	
	λ_3	0.9982	1.0000	0.18%	
$N_m = 13$	λ_1	-1.3902	-1.4000	0.70%	$E_{Rel} = 2.9\%$
	λ_2	-0.9829	-1.0000	1.7%	
	λ_3	0.9894	1.0000	1.1%	
$N_m = 8$	λ_1	-1.3907	-1.4000	0.66%	$E_{Rel} = 1.6\%$
	λ_2	-1.0034	-1.0000	0.34%	
	λ_3	0.9914	1.0000	0.86%	
$N_m = 5$	λ_1	-1.3275	-1.4000	5.2%	$E_{Rel} = 6.5\%$
	λ_2	-0.9773	-1.0000	2.3%	
	λ_3	0.9523	1.0000	4.8%	

Table 8.7: Results generated by the Adaptive-PINN on the hybrid data-set versus the true target values for the parameters λ_1, λ_2 and λ_3 of the 2D ADE as specified by (1.9).

Figure 8.16 shows the training histories corresponding to the Adaptive-PINN trained on the hybrid data-set with $N_m = 8$. We see that the oscillatory behaviour in the loss history has been greatly reduced, and as expected the unknowns λ_i converge to their target velus. By figure 8.18 and 8.19 we see that solutions for u are now more accurate than in the case for the Weighted-PINN in section 8.2. Figure 8.17 shows the noisy reference data u^* that was available for the PINN to train on.

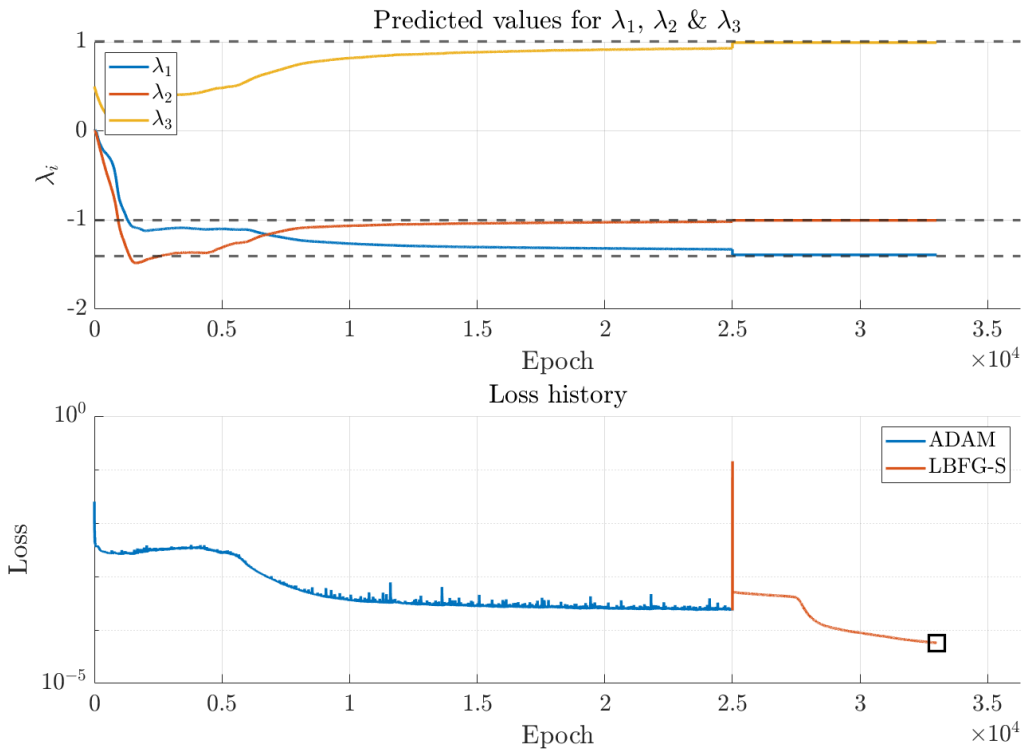

Figure 8.16: Training histories for the Adaptive-PINN on hybrid data in the inverse case for the problem specified by equation (1.9). Top: The predicted values for λ_1, λ_2 and λ_3 over each epoch of training. Bottom: The Objective function loss over each epoch of training. Lowest achieved loss given by the black square.

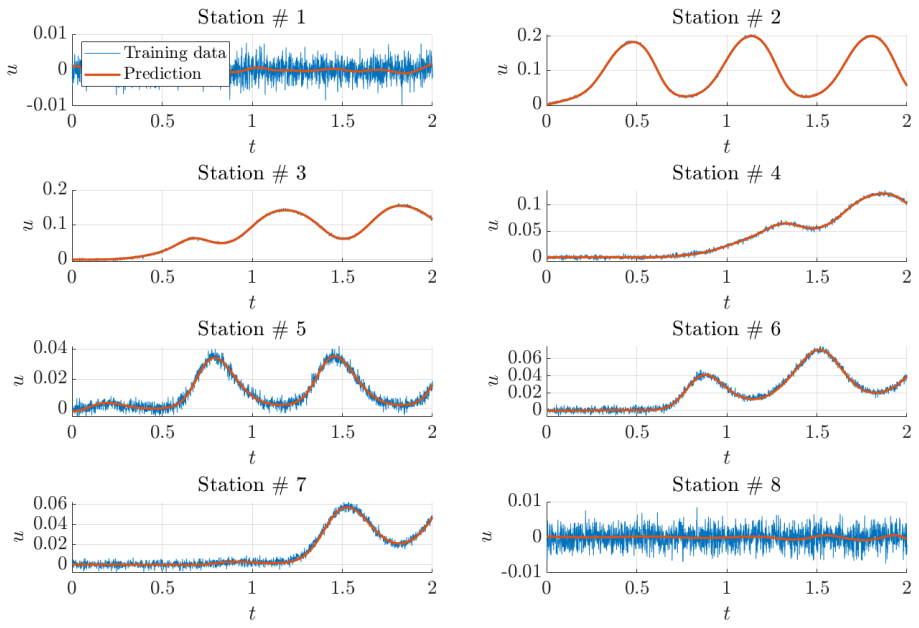

Figure 8.17: The predicted solution u generated by the Adaptive-PINN versus the noisy training data u^* given as time-series from measurement stations.

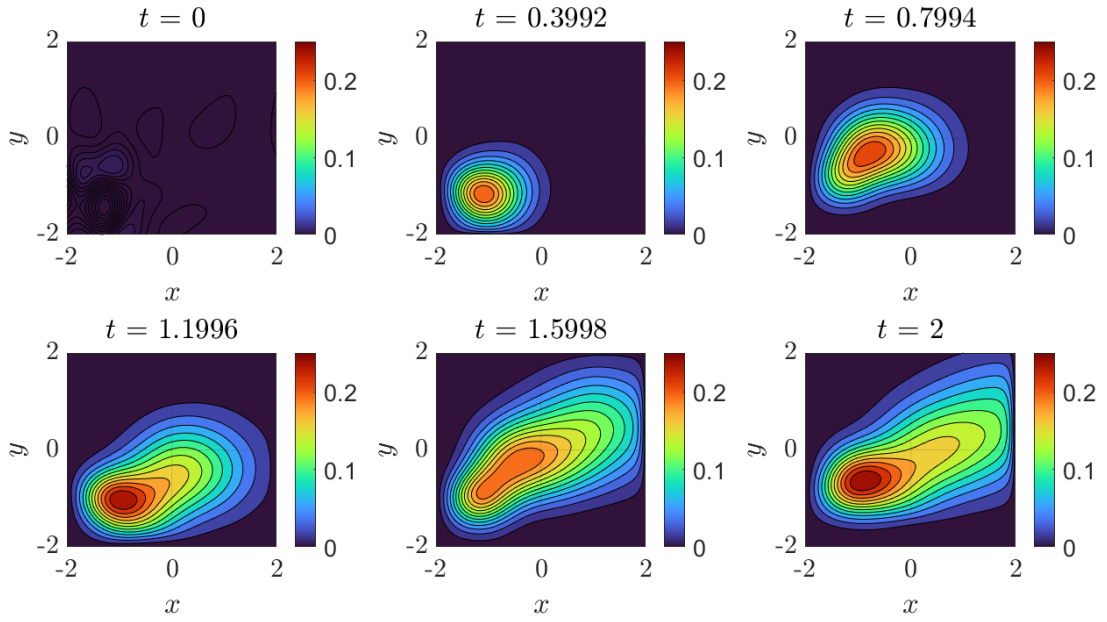

Figure 8.18: The predicted solution u generated by the Adaptive-PINN of the 2D ADE as specified by (1.9) at different snapshots in time.

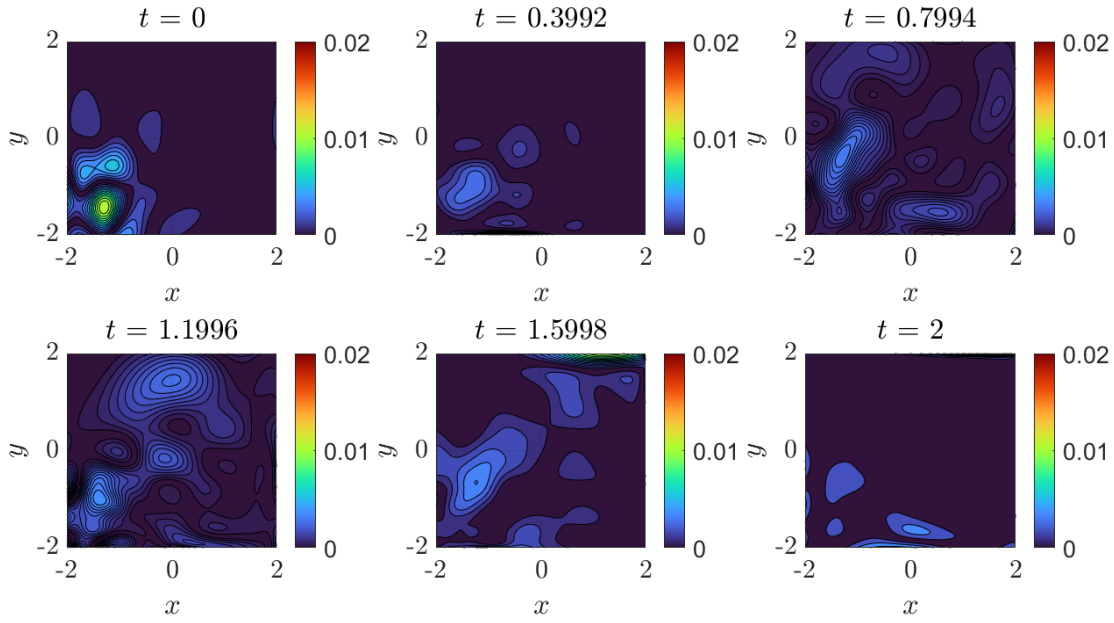

Figure 8.19: Absolute point-wise error between the predicted solution u generated by the Adaptive-PINN and the reference solution u^* for different snapshots in time.

Chapter 9

Conclusions and Future Work

We have seen how the PINN can be applied to a variety of different advection-diffusion problems. We ran the PINN on series of test-cases in one and two dimensions in both the forward and inverse sense. The main takeaways are the following:

The densely connected network containing 8 hidden layers and 20 neurons per layer as discussed in section 4.3 is sufficient enough to express the solutions for both the 1D and 2D ADEs specified by (1.8) and (1.9). This is evident, simply by the fact that the PINN could indeed approximate the solutions for u in the forward sense, as shown in section 6.1.1 and 6.2.1.

However, the Baseline-PINN would run into convergence issues whenever faced with the 2D inverse ADE problem as shown in section 6.2.2, as the Baseline-PINN can express gradient pathologies which inhibit convergence. But as *Wang et al.* found, this can be alleviated by scaling the objective loss function by the Weighted- or Adaptive-PINN. The Adaptive-PINN is generalisable and can account for changes in problem formulation, and it proved to be effective in solving the problem in section 8.1 with dense data. But we found that it was not able to solve for the sparse data-sets in section 8.2 and 8.3 effectively. As such, the Weighted-PINN had to be used here instead. We found that the Weighted-PINN could accurately predict the source locations with these sparse data-sets (see tables 8.2 and 8.3 in comparison to table 7.1), with a compromised accuracy for the regressed solution u.

We also investigated the stability of solutions with respect to changes in initial guesses. We found that the basin of attraction for the initial guesses covers a significant part of the domain, and if the initial guess is sufficient, then the PINN will reliably converge onto the expected results. However, if the initial guesses are sufficiently far away from the true values, the PINN will not converge, even with the strategies discussed in 6.2.2, 7.1 and 7.2. Therefore, it may be important to determine good initial guesses prior to applying the PINN on an inverse problem.

In sections 8.2 and 8.3 we showed that the Weighted-PINN trained on sparse and noisy data had the capacity to discover and localise sources very accurately. However, $N_m = 26$ measurement stations is not particularly practical. And furthermore, the 2D test case we were working with as specified by equation (1.9) is not particularly

complex, as the velocity field is uniform and smooth. We can imagine that problems would further compound if we were to use turbulent velocity fields. It puts into question how PINNs would fair against complicated velocity-fields without an unrealistic number of measurement stations. Furthermore, these results were only achieved on the Weighted-PINN, which works well for the specific test case in equation (1.9), but it may require manual re-tuning of u_m and f_m if faced with a different test case.

Therefore, we aimed to improve the accuracy of the Adaptive-PINN and make it more robust. In section 8.5 we proposed a hybrid training data-set which allows the Adaptive-PINN to train on points other than those at the measurement stations. We found that the Adaptive-PINN with this hybrid training set outperforms the Weighted-PINN on the dense data-set by a significant margin. In term we can reduce the required amount of measurement stations, by the supplement of additional collocation-points, from $N_m = 26$ down to $N_m = 8$ or $N_m = 5$ depending on the desired accuracy. This result strengthens the robustness of the PINN method, and it shows that we may be able to implement the Adaptive-PINN on more complicated problems in the future without having to go through the tedious work of finding suitable scaling values for u_m and f_m with the Weighted-PINN.

There is nothing in the formulation of PINNs that would impede us from expanding to 3D problems, problems on complicated domains, and problems with non-uniform velocity fields. However, the fact that the Baseline-PINN did not work very well, implies that the method is not robust in its basic form. We therefore cannot fully trust that the results would reflect the ground truth. But, the Weighted- and Adaptive-PINNs worked very reliably, even on sparse data-sets. In particular we had notably high accuracy on the Adaptive-PINN trained on the hybrid data-set. We therefore conclude that the PINN methodology indeed has the capability of localising and discovering sources for inverse 2D advection-diffusion problems, as long as gradient pathologies are taken care of and they are configured correctly for the specific problem at hand. As for possible future work, we can make the following remarks:

Further studies need to be conducted on how to make PINNs more robust and efficient. There are still many questions that remain: Since the method is not fully robust, to what degree can we trust the results? What are the best hyper-parameters for PINNs? The Weighted- and Adaptive-PINNs work well, but are there other and more effective ways of scaling the objective loss function for optimal performance? How do we select the best initial guesses for inverse problems? Are there other strategies that can be employed to make PINNs more robust and less prone to convergence issues? We discussed other possible mitigating strategies in section 7.3, and they could provide an interesting avenue for further studies. Could these strategies we discussed increase performance on the our test cases even further? Moreover, how does the Adaptive-PINN with the hybrid data-set as described in section 8.5 compare to the alternative methods discussed in sections 1.6.1, 1.6.2 and 1.6.3 in terms of accuracy, required data and optimisation times? In section 1.6.2 we mentioned that our implementation cannot deal with time-dependent source intensities directly, but we gave a suggestion on one possible way to do so. Can the PINN be modified to deal with such problems? And lastly, it possible to make the PINN able to solve for an unknown number of sources?

Finally, a word of warning; If one aims to solve problems with PINNs, one should be prepared to do a lot of trial and error to get them to work properly. As proclaimed by *Bilionis* when discussing PINNs in his lecture: '... What is the catch? The catch is that when you try to apply PINNs to your problem, it is not going to work.' [7]. We feel that this quote articulates our position on PINNs. Before we figured out that gradient pathologies were the culprit behind our convergence issues, we were led down an endless rabbit hole of parameter tuning which ultimately lead to nothing useful. And even when we had an idea of what was going wrong thanks to *Lu et al.* [26, 27], we still had to do a lot of tuning to get the Weighted-PINN to converge properly. The theory behind PINNs is not fully understood, and one must be ready to encounter many setbacks when implementing them for your own problems. Let sections 7.1, 7.2 and 7.3 be guidelines for established strategies to try.

Chapter 10

Final Comments

If I could, I would have done many things differently. Mainly, I would have tried to get a wider and more nuanced perspective of PINNs before attempting to implement them. I started implementing PINNs almost immediately after reading about them in the studies by *Raissi et al.* [42, 43]. These two papers in particular showed that PINNs were very successful in solving a wide array of problems. And because of this success, it gave me a skewed perspective of how robust and powerful the Baseline-PINN really is. This perspective lead me to get stuck in a line of thinking that inhibited me from trying other ideas. I would always blame the fault of the PINN on my own implementation. Whether that be possible bugs in the code or ill-suited hyper-parameters. Early on, I never considered that the Baseline-PINN itself could be insufficient, as it had been used to such great success.

It was only until I asked *Lu et al.* [27] for advice and found examples of PINNs failing, that i realised that there is more to the story. Reading up on a wider selection of papers relating to PINNs in the beginning, would have helped me realise the Baseline-PINN does have inherent issues which can prevent proper convergence. As such, I could have avoided several weeks of trial and error which ultimately lead to no real improvements, and instead focus on the mitigative strategies that were mentioned in section 7.1, 7.2 and 7.3. I had a gut feeling about loss scaling for a long time, but I did not have a proper justification for why it is important. This justification was apparent to me only after reading the paper by *Wang et al.* [49].

If I had resolved the issues discussed in 6.2.2 at an earlier date, I may have been able to work on the 2D inverse problem exclusively. Instead I became unsure if I would ever resolve my issues, so I spent time on solving for the easier problems. As such, I did not have the time to implement PINNs on the irregular domains and non-uniform velocity fields. In fact, figure (1.1) shows one of the problem scenarios with an irregular domain and non-uniform velocity field that I was prototyping, but ultimately had to be scrapped.

Appendix A

Well-Posedness Proof for the ADE

The following section shows the well-posedness proof for the ADE in this study. It is used to show that the solution exists and does not blow up within finite time.

A.1 Well-Posedness of the ADE

The ADE in n dimensions can be written according to equation (1.7). We write it again here,

$$\frac{\partial u}{\partial t} + \mathbf{v} \cdot \nabla u = D\Delta u + F(\mathbf{x}, t). \tag{A.1}$$

Here $\mathbf{x} = x_i \mathbf{e}_i$ for $i = 1, 2, \ldots, n$ is the position in the n dimensional space. $\mathbf{v} = V_i(\mathbf{x}, t)\mathbf{e}_i$ are the corresponding velocity components which may depend on space and time. D is the diffusion rate, and F is a source/forcing term. Here we will interpret it as a source for some contaminant leak which can be localised in space. In this specific case we let it follow the form in equation (1.6),

$$F(\mathbf{x}, t) = Ae^{-B((\mathbf{x}-\mathbf{x}_s)^2)}.$$

The constant $\mathbf{x}_s$ can be tuned to change the location of the contaminant leak in the domain. While A and B determine the strength and size of the leak respectively.

To prove well posedness of the PDE, we need to show that the following inequality holds:

$$||u(\cdot, t)|| \leq Ke^{\alpha t}\left(||u(\cdot, 0)|| + \int_0^t (||F(\cdot, \tau)|| + |g(\tau)|^2)d\tau\right). \tag{A.2}$$

Here $g(t)$ are potential boundary data corresponding to different boundary conditions. This condition is sufficient to prove that the solution will not grow without bound for finite time and that a unique solution exists [6].

We apply the energy method on the ADE by multiplying each term by u, then integrating over the spatial domain. First we define the L^2-norm energy as the following:

$$||u(\cdot,t)||^2 = \iint_\Omega u(\mathbf{x},t)^2 d\mathbf{x}. \tag{A.3}$$

Now we multiply each term in the ADE by u and integrate. We look at the first term to start,

$$\iint_\Omega u u_t d\mathbf{x} = \frac{1}{2}\iint_\Omega \frac{\partial}{\partial t} u^2 d\mathbf{x} = \frac{1}{2}\frac{\partial}{\partial t}\iint_\Omega u^2 d\mathbf{x} = \frac{1}{2}\frac{\partial}{\partial t}\big(||u(\cdot,t)||^2\big). \tag{A.4}$$

We see that the first term readily simplifies into the time-derivative of the L^2-norm energy. The second term can be dealt with via integration by parts,

$$\iint_\Omega u(\mathbf{v}\cdot\nabla u)d\mathbf{x} = \int_{\partial\Omega} u^2 \mathbf{v}\cdot\mathbf{n} dS - \iint_\Omega u\nabla\cdot(u\mathbf{v})d\mathbf{x}. \tag{A.5}$$

The third term can be simplified with the help of Green's first identity,

$$\iint_\Omega Du\Delta u d\mathbf{x} = D\int_{\partial\Omega} u(\nabla u\cdot\mathbf{n})dS - D\iint_\Omega (\nabla u\cdot\nabla u)d\mathbf{x}. \tag{A.6}$$

And for the final term, we use the Cauchy-Schwartz inequality,

$$\iint_\Omega uF d\mathbf{x} \le \iint_\Omega |uF| d\mathbf{x} \le ||u||\cdot||F|| \le \frac{1}{2}||u||^2 + \frac{1}{2}||F||^2. \tag{A.7}$$

The last inequality comes from the fact that $|ab| \le \frac{1}{2}a^2 + \frac{1}{2}b^2$, for all real numbers a and b. Assembling equations (A.4), (A.5), (A.6) and (A.7) all together according to (A.1), gives us the following inequality:

$$\begin{aligned}\frac{1}{2}\frac{\partial}{\partial t}\big(||u(\cdot,t)||^2\big) \le &-\int_{\partial\Omega} u^2\mathbf{v}\cdot\mathbf{n}dS + \iint_\Omega u\nabla\cdot(u\mathbf{v})d\mathbf{x} \\ &+ D\int_{\partial\Omega} u(\nabla u\cdot\mathbf{n})dS - D\iint_\Omega(\nabla u\cdot\nabla u)d\mathbf{x} \\ &+ \frac{1}{2}||u||^2 + \frac{1}{2}||F||^2.\end{aligned}$$

The easiest way of guaranteeing well-posedness is to set $u = 0$ at the boundaries, thus the boundary terms are guaranteed to disappear,

$$\frac{\partial}{\partial t}\big(||u(\cdot,t)||^2\big) \le 2\iint_\Omega u\nabla\cdot(u\mathbf{v})d\mathbf{x} - 2D\iint_\Omega(\nabla u\cdot\nabla u)d\mathbf{x} + ||u||^2 + ||F||^2. \tag{A.8}$$

Notice how $-2D\iint_\Omega(\nabla u\cdot\nabla u)d\mathbf{x} = -2D||\nabla u||^2 \le 0$, so this term is always negative and it does therefore not contribute to any growth and it can be ignored for our proof. We still need to bound the integral term containing the velocity field components $\mathbf{v}$. We have the following relations:

$$\iint_\Omega u\nabla\cdot(u\mathbf{v})d\mathbf{x} = \iint_\Omega u^2\nabla\cdot\mathbf{v} + u\nabla u\cdot\mathbf{v}d\mathbf{x} = \iint_\Omega u^2\nabla\cdot\mathbf{v}d\mathbf{x} + \iint_\Omega u\nabla u\cdot\mathbf{v}d\mathbf{x}. \tag{A.9}$$

Now we need to show that we can bound these terms. We have the following relation for the first term of the last equality in (A.9):

$$\iint_\Omega u^2 \nabla \cdot \mathbf{v} d\mathbf{x} \leq \max_{\mathbf{x},t} (\nabla \cdot \mathbf{v}(\mathbf{x},t)) \iint_\Omega u^2 d\mathbf{x} = \max_{\mathbf{x},t} (\nabla \cdot \mathbf{v}(\mathbf{x},t)) ||u||^2.$$

Here, $\max_{\mathbf{x},t} (\nabla \cdot \mathbf{v}(\mathbf{x},t))$ represents the maximum value $\nabla \cdot \mathbf{v}$ takes in the entire domain Ω for all times considered. For the sake of keeping clutter to a minimum, let $M_\nabla = \max_{\mathbf{x},t} (\nabla \cdot \mathbf{v}(\mathbf{x},t))$.

Now, for the second term of the last inequality in (A.9), define $\mathbf{E} = \sum_i \mathbf{e}_i$ where $\mathbf{e}_i$ are the basis vectors for the space. $\mathbf{E}$ represents an n dimensional vector with all entries occupied by 1. We can make the following upper bound:

$$\iint_\Omega u \nabla u \cdot \mathbf{v} d\mathbf{x} \leq \max_{\mathbf{x},t} (\mathbf{v}(\mathbf{x},t)) \iint_\Omega u \nabla u \cdot \mathbf{E} d\mathbf{x} = \max_{\mathbf{x},t} (\mathbf{v}(\mathbf{x},t)) \iint_\Omega \nabla u \cdot (u\mathbf{E}) d\mathbf{x}. \quad \text{(A.10)}$$

Similarly to before, $\max_{\mathbf{x},t} (\mathbf{v}(\mathbf{x},t))$ represents the maximum value that any of the components $|V_i|$ of $\mathbf{v}$ takes in the entire domain Ω for all times considered. Again, name $M = \max_{\mathbf{x},t} (\mathbf{v}(\mathbf{x},t))$. Let $\mathbf{U} = u\mathbf{E}$. Now by integration by parts on (A.10), we get the following:

$$\iint_\Omega u \nabla u \cdot \mathbf{v} d\mathbf{x} \leq M \iint_\Omega \nabla u \cdot \mathbf{U} d\mathbf{x} = M \left[\int_{\partial\Omega} u\mathbf{U} \cdot \mathbf{n} dS - \iint_\Omega u \nabla \cdot \mathbf{U} d\mathbf{x} \right]. \quad \text{(A.11)}$$

It can be shown that $\nabla u \cdot \mathbf{U} = u \nabla \cdot \mathbf{U}$ in the last term of (A.11). It is easiest to see this in component form. We have the following two component form expansions:

$$\nabla u \cdot \mathbf{U} = \frac{\partial u}{\partial x_i} \mathbf{e}_i \cdot u \sum_j \mathbf{e}_j = \sum_j \frac{\partial u}{\partial x_i} u \delta_{ij} = \sum_j \frac{\partial u}{\partial x_j} u, \quad \text{(A.12)}$$

and,

$$u \nabla \cdot \mathbf{U} = u \frac{\partial}{\partial x_i} \mathbf{e}_i \cdot \sum_j u \mathbf{e}_j = \sum_j u \frac{\partial u}{\partial x_i} \delta_{ij} = \sum_j \frac{\partial u}{\partial x_j} u. \quad \text{(A.13)}$$

Equations (A.12) and (A.13) are indeed equivalent. Thus, we can assert the following from (A.11):

$$M \iint_\Omega \nabla u \cdot \mathbf{U} d\mathbf{x} = \frac{1}{2} M \left[\int_{\partial\Omega} u\mathbf{U} \cdot \mathbf{n} dS \right] = 0.$$

Where the last equality is due to our assertion that $u = 0$ on the boundaries $\partial\Omega$.

Hence, from equation (A.11) we have the following:

$$\iint_\Omega u \nabla u \cdot \mathbf{v} d\mathbf{x} \leq 0.$$

This term will also not lead to growth and can accordingly be ignored for our proof. Therefore, we are now left with the following simplified form of (A.8),

$$\frac{\partial}{\partial t}\left(||u(\cdot,t)||^2\right) \leq 2M_{\nabla}||u||^2 + ||u||^2 + ||F||^2.$$

This is an ODE of $||u||^2$. Solving for $||u(\cdot,t)||^2$ gives the following inequality:

$$||u(\cdot,t)||^2 \leq e^{(1+2\max_{\mathbf{x},t}(\nabla\cdot\mathbf{v}(\mathbf{x},t))t}\left(||u(\cdot,0)||^2 + \int_0^t e^{-(1+2\max_{\mathbf{x},t}(\nabla\cdot\mathbf{v}(\mathbf{x},t))\tau}||F(\cdot,\tau)||^2 d\tau\right).$$

Since $(1+2\max_{\mathbf{x},t}(\nabla\cdot\mathbf{v}(\mathbf{x},t))) \geq 0$, we can further assert the following:

$$||u(\cdot,t)||^2 \leq e^{(1+2\max_{\mathbf{x},t}(\nabla\cdot\mathbf{v}(\mathbf{x},t))t}\left(||u(\cdot,0)||^2 + \int_0^t ||F(\cdot,\tau)||^2 d\tau\right).$$

Now let the growth constant α be set such that it obeys the following inequality.

$$\alpha \geq (1+2\max_{\mathbf{x},t}(\nabla\cdot\mathbf{v}(\mathbf{x},t)).$$

Therefore,

$$||u(\cdot,t)||^2 \leq e^{\alpha t}\left(||u(\cdot,0)||^2 + \int_0^t ||F(\cdot,\tau)||^2 d\tau\right).$$

Now we have shown that the L^2-norm energy of u (A.3) obeys the inequality in (A.2). Hence, for the boundary conditions $u = 0|_{\mathbf{x}\in\partial\Omega}$ and by assuming that the derivatives of the velocity components are bounded, $\max_{\mathbf{x},t}(\nabla\cdot\mathbf{v}(\mathbf{x},t)) \leq C < \infty$ we have that the PDE is well posed! We now have hopes of solving a well defined problem which would result in a unique solution. Hence, when solving for the ADE, we mean solving for the following problem,

$$\frac{\partial u}{\partial t} + \mathbf{v}\cdot\nabla u = D\Delta u + F(\mathbf{x},t), \quad \mathbf{x}\in\Omega,$$

$$u(\mathbf{x},t) = 0|_{\mathbf{x}\in\partial\Omega}, \quad u(\mathbf{x},0) = 0,$$

$$\mathbf{v}_i = V_i(\mathbf{x},t).$$

A.2 Uniqueness of the solution

Showing uniqueness when assuming the problem is well-posed, is simple. Suppose there are two solutions of the linear PDE in question, u and v, both of which obviously satisfy the same boundary and initial data.

$$\begin{aligned} ||u(\cdot,0)|| &= ||v(\cdot,0)||, \\ ||F_u(\cdot,\tau)|| &= ||F_v(\cdot,t)||, \\ ||g_u(\tau)|| &= ||g_v(\tau)||. \end{aligned}$$

Let the difference between these two solutions be $e = u - v$. This difference must then satisfy the linear PDE with the following energies,

$$\begin{aligned} ||e(\cdot,0)|| &= ||u(\cdot,0)|| - ||v(\cdot,0)|| = 0, \\ ||F_e(\cdot,\tau)|| &= ||F_u(\cdot,\tau)|| - ||F_v(\cdot,t)|| = 0, \\ ||g_e(\tau)|| &= ||g_u(\tau)|| - ||g_v(\tau)|| = 0. \end{aligned}$$

However, as both u and v are assumed to be well-posed, then by linearity e is also well-posed. So, by the well-posedness inequality (A.2), we are lead to the following equality:

$$||e(\cdot,t)|| = 0.$$

But, this could only be true if $u = v$ for all t. Hence, we have shown uniqueness.

Appendix B

Finite Differences Derivations

In this appendix we deal with defining the difference schemes which we use to generate our reference data.

B.1 Finite Differences and Order of Accuracy

Let f be a real continuous function defined on $x \in \mathbb{R}$. Then by the definition of the derivative, we have the following relation,

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

On a computer, we cannot simply set $h = 0$ to compute our derivatives. But we can get a reasonable approximation by letting h be some small value.

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}, \quad 0 < h << 1$$

This finite difference approximation for the first derivative is commonly called the forward difference approximation. How small do we need to make h to reach some accuracy threshold? By Taylor expanding f we get,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(C), \quad x < C < x+h.$$

Then, by inserting this into our approximation we get,

$$\frac{f(x+h) - f(x)}{h} = \frac{f(x) + hf'(x) + \frac{h^2}{2} f''(C) - f(x)}{h} = f'(x) + \frac{h}{2} f''(C) = f'(x) + \mathcal{O}(h).$$

As long as the second derivative of f is bounded such that $|f''(C)| \leq M < \infty$ for $x < C < x+h$, then our finite difference approximation has an error of order $\mathcal{O}(h)$. This means that the error decreases proportionally to h. However, we can do better than this. By Taylor expanding $f(x+h)$ and $f(x-h)$ we have the following two expansions,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(C_1), \quad x < C_1 < x+h,$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(C_2), \quad x-h < C_2 < x.$$

Then by taking the difference of these two expansions, we get cancelling terms,

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{6}[f'''(C_1) + f'''(C_2)].$$

Now, dividing by $2h$, we get another finite difference approximation of f',

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{12}[f'''(C_1) + f'''(C_2)] = f'(x) + \mathcal{O}(h^2). \tag{B.1}$$

In this case, we find a finite difference approximation with order of accuracy $\mathcal{O}(h^2)$. This specific approximation is commonly called the central difference scheme.

As the central difference scheme is symmetric around x and it has higher order accuracy than the forward difference scheme, we will use it over the forward difference scheme.

It can be similarly shown that the second order derivative f'' has the central difference approximation of the following form,

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + \mathcal{O}(h^2). \tag{B.2}$$

We see that it also has an order of accuracy of $\mathcal{O}(h^2)$.

To compute these derivatives on the computer, one will have to transfer the continuous function $f(x)$ into a grid-function f_i. Where the index i denotes the position x_i of the input. Let $f_i = f(x_i)$. In this notation, the first and second order central difference schemes in equation (B.1) and (B.2) respectively become,

$$\begin{aligned} f_i' &= \mathcal{D}_1 f_i = \frac{f_{i+1} - f_{i-1}}{2h} + \mathcal{O}(h^2) \\ f_i'' &= \mathcal{D}_2 f_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + \mathcal{O}(h^2) \end{aligned} \tag{B.3}$$

The same general idea can be applied to partial derivatives of multi-variable functions as well.

One can argue that the choice of the central difference scheme for the ADE is poor. This is because it can be shown that the central difference can suffer from growing oscillatory solutions when applied to the advection equation [11]. In fact, it is generally advised to use an upwind scheme to avoid oscillatory solutions. However, as we wanted to create a universal solver that can handle any advection direction, we decided to stick with the central difference scheme. Regardless, as long as the diffusive rate D is large enough and dt small enough, we can expect oscillatory solutions to be damped before they become too much of a problem. In fact the authors of [11] argue that in the 1D instance, if the the Péclet number,

$$P_e = \frac{\Delta x V_x}{D}, \tag{B.4}$$

satisfies $P_e < 2$, then oscillatory solutions do not occur. In our studies we had chosen our ADE coefficients such that the Péclet number was kept well below one. Accordingly, we did not expect any oscillatory solutions.

B.2 Finite Difference Approximation of the 1D ADE

Given the 1D instance of the ADE in equation (1.7) we have the following expanded differential equation,

$$\begin{aligned} \frac{\partial u}{\partial t} + V_x \frac{\partial u}{\partial x} &= D\frac{\partial^2 u}{\partial x^2} + F(x;P), \quad x \in [a,b], \\ F(x;P) &= \lambda_2 e^{-S(x-\lambda_1)^2}, \quad P = \lambda_i, \\ u(a,t) &= u(b,t) = 0, \quad u(x,0) = 0. \end{aligned}$$

We divide the x axis into N_x grid-points. Let $u(t)_i = u(x_i,t)$ with $x_i = a + (i-1)\Delta x$, $\Delta x = (b-a)/(N_x - 1)$ and $i = \{1,\ldots,N_x\}$, then by the equations in (B.3), we get,

$$\begin{aligned} \left(\frac{\partial u}{\partial x}\right)_i &\approx \frac{u(t)_{i+1} - u(t)_{i-1}}{2\Delta x}, \\ \left(\frac{\partial^2 u}{\partial x^2}\right)_i &\approx \frac{u(t)_{i+1} - 2u(t)_i + u(t)_{i-1}}{\Delta x^2}. \end{aligned}$$

Thus, the semi-discrete finite differences approximation can be written in the following form,

$$\begin{aligned} \left(\frac{\partial u}{\partial t}\right)_i = D\frac{u(t)_{i+1} - 2u(t)_i + u(t)_{i-1}}{\Delta x^2} - V_x \frac{u(t)_{i+1} - u(t)_{i-1}}{2\Delta x} + \lambda_2 e^{-S(x_i-\lambda_1)^2}, \\ u(t)_1 = u(t)_{N_x} = 0, \quad u(0)_i = 0, \\ i = \{1,\ldots,N_x\}. \end{aligned} \tag{B.5}$$

B.3 Matrix-Vector Form of the 1D ADE Finite Difference Scheme

The finite difference equation for the 1D ADE is given by equation (B.5). We want to transfer this equation into vector form given as,

$$\frac{\partial \mathbf{u}}{\partial t} = A\mathbf{u} + \mathbf{F}.$$

To do this, we let the solution $\mathbf{u}(t)$ be a vector of the form $\mathbf{u}(t) = [u_1(t), \ldots, u_i(t), \ldots, u_{N_x}(t)]^T$. Similarly we also construct $\mathbf{F} = [F_1, \ldots, F_{N_x}]^T$.

The matrix A is constructed such that row i follows the same form as in (B.5). As the difference scheme has a stencil consisting of three points, the matrix A will be a tri-diagonal matrix. In particular, we get the following equations corresponding to the different rows of $A\mathbf{u}$,

$$\begin{aligned} A_{1j}u_j &= 0, \\ A_{ij}u_j &= \left(\frac{D}{\Delta x^2} - \frac{V_x}{2\Delta x}\right)u_{i+1} - \frac{2D}{\Delta x^2}u_i + \left(\frac{D}{\Delta x^2} + \frac{V_x}{2\Delta x}\right)u_{i-1}, \\ A_{N_x j}u_j &= 0, \end{aligned} \tag{B.6}$$

and similarly for the corresponding rows of $\mathbf{F}$,

$$\begin{aligned} F_1 &= 0, \\ F_i &= \lambda_2 e^{-S(x_i-\lambda_1)^2}, \\ F_{N_x} &= 0. \end{aligned} \tag{B.7}$$

The first and last rows of A and $\mathbf{F}$ are zero, as to obey the homogeneous boundary conditions via injection.

By studying equation (B.6) we end up with a matrix A as given in equation (B.8) with coefficients as given in (B.9).

$$A = \begin{bmatrix} 0 & & & & & & \\ \alpha_x & \beta_x & \gamma_x & \cdots & & & \\ & & \ddots & & & & \\ & \cdots & \alpha_x & \beta_x & \gamma_x & \cdots & \\ & & & & \ddots & & \\ & & & \cdots & \alpha_x & \beta_x & \gamma_x \\ & & & & & & 0 \end{bmatrix} \tag{B.8}$$

$$\alpha_x = \frac{D}{\Delta x^2} + \frac{V_x}{2\Delta x}, \quad \beta_x = -\frac{2D}{\Delta x^2}, \quad \gamma_x = \frac{D}{\Delta x^2} - \frac{V_x}{2\Delta x}, \tag{B.9}$$

With the definitions given in equations (B.7), (B.8) and (B.9) we can readily construct the finite difference scheme of the desired form in equation (2.3), which can be solved as described in section 2.5.

B.4 Finite Difference Approximation of the 2D ADE

Given the 2D instance of the ADE in equation (1.7) we have the following expanded differential equation,

$$\begin{aligned}
\frac{\partial u}{\partial t} + V_x\frac{\partial u}{\partial x} + V_y(t)\frac{\partial u}{\partial y} &= D\frac{\partial^2 u}{\partial x^2} + D\frac{\partial^2 u}{\partial y^2} + F(x,y;P), \quad x \in [a,b], \quad y \in [c,d], \\
F(x,y;P) &= \lambda_3 e^{-S\left((x-\lambda_1)^2+(y-\lambda_2)^2\right)}, \quad P = \lambda_i, \\
u(a,y,t) = u(b,y,t) = u(x,c,t) = u(x,d,t) &= 0, \quad u(x,y,0) = 0.
\end{aligned}$$

We divide the x,y rectangular plane into $N_x \cdot N_y$ grid-points. Let $u(t)_{i,j} = u(x_i, y_j, t)$ where $x_i = a + (i-1)\Delta x$, $y_j = c + (j-1)\Delta y$, $\Delta x = (b-a)/(N_x - 1)$, $\Delta y = (d-c)/(N_y - 1)$ and $i = \{1,\ldots,N_x\}$, $j = \{1,\ldots,N_y\}$.

Then by the equations in (B.3), we get,

$$\begin{aligned}
\left(\frac{\partial u}{\partial x}\right)_{i,j} &\approx \frac{u(t)_{i+1,j} - u(t)_{i-1,j}}{2\Delta x}, \\
\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} &\approx \frac{u(t)_{i+1,j} - 2u(t)_{i,j} + u(t)_{i-1,j}}{\Delta x^2}, \\
\left(\frac{\partial u}{\partial y}\right)_{i,j} &\approx \frac{u(t)_{i,j+1} - u(t)_{i,j-1}}{2\Delta y}, \\
\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} &\approx \frac{u(t)_{i,j+1} - 2u(t)_{i,j} + u(t)_{i,j-1}}{\Delta y^2}.
\end{aligned}$$

Thus, the semi-discrete finite differences approximation can be written as,

$$\begin{aligned}
\left(\frac{\partial u}{\partial t}\right)_{i,j} &= D\left(\frac{u(t)_{i+1,j} - 2u(t)_{i,j} + u(t)_{i-1,j}}{\Delta x^2} + \frac{u(t)_{i,j+1} - 2u(t)_{i,j} + u(t)_{i,j-1}}{\Delta y^2}\right) \\
&\quad - V_x\frac{u(t)_{i+1,j} - u(t)_{i-1,j}}{2\Delta x} - V_y(t)\frac{u(t)_{i,j+1} - u(t)_{i,j-1}}{2\Delta y} + F_{i,j}, \\
F_{i,j} &= \lambda_3 e^{-S\left((x_i-\lambda_1)^2+(y_j-\lambda_2)^2\right)}, \\
u(t)_{1,j} = u(t)_{N_x,j} = u(t)_{i,1} = u(t)_{i,N_y} &= 0, \quad u(0)_{i,j} = 0, \\
i = \{1,\ldots,N_x\}, &\quad j = \{1,\ldots,N_y\}.
\end{aligned} \tag{B.10}$$

The solution can then be assembled as a matrix $U(t)$ of the form,

$$u(t) = \left[[u(t)_{1,1},\ldots,u(t)_{N_x,1}]^T,\ldots,[u(t)_{1,N_y},\ldots,u(t)_{N_x,N_y}]^T\right].$$

B.5 Matrix-Vector Form of the 2D ADE Finite Difference scheme

The finite difference equation for the 2D ADE is given by equation (B.10). We want to transfer this equation into vector form given as,

$$\frac{\partial \mathbf{u}}{\partial t} = A\mathbf{u} + \mathbf{F}.$$

This is not as straight forward as in the 1D instance as shown in section B.3. This is because we have treated the 2D grid-functions $u_{i,j}$ and $F_{i,j}$ as matrices. We need some way to flatten them into vectors of the form $\mathbf{u} = [u_{1,1}, \ldots, u_{N_x,N_y}]^T$ and $\mathbf{F} = [F_{1,1}, \ldots, F_{N_x,N_y}]^T$. We decided to flatten the matrices via the following transformations,

$$u_{i,j} = \mathbf{u}_{i+N_x(j-1)},$$

and,

$$F_{i,j} = \mathbf{F}_{i+N_x(j-1)}.$$

Now that $\mathbf{u}$ and $\mathbf{F}$ are $N_x \cdot N_y \times 1$ vectors, we need to construct the $N_x \cdot N_y \times N_x \cdot N_y$ matrix A. As this difference scheme is dependent on five points, the matrix will be a five banded diagonal matrix. In addition, we need some rows of A to be zero as we want to obey the homogeneous boundary conditions by injection.

We need to take into careful consideration the fact that boundaries consist of sparse sections of points since we flattened $u_{i,j}$. By trial and error, we figured out the following pieces of information about A: A consists of $N_x + 1$ rows of zeros, then $N_y - 2$ sections of alternating $N_x - 2$ non-zero rows followed by 2 rows of zeroes. This pattern repeats until we get $N_x - 1$ rows of zeroes for the last rows.

The non-zero rows are made up of five bands which correspond to each of the nodes used in the difference scheme. This information was enough to write an algorithm which could generate A for any size given by N_x and N_y. However, the form of this matrix becomes quite a challenge to write down. Instead of showing it in its general form, we show a simple example when $N_x = N_y = 4$.

By following our empirical algorithm, we need $N_x + 1 = 5$ rows of zeroes, then we have $N_y - 2 = 2$ sections with $N_x - 2 = 2$ rows non-zero followed by 2 rows with zeroes. Followed up with $N_x + 1 = 5$ rows of zeroes. We get,

$$A(t) = \begin{bmatrix}
0 & & & & & & & & & & & & & & & \\
& 0 & & & & & & & & & & & & & & \\
& & 0 & & & & & & & & & & & & & \\
& & & 0 & & & & & & & & & & & & \\
& & & & 0 & & & & & & & & & & & \\
\alpha_y & & & \alpha_x & \beta & \gamma_x & & & \gamma_y & & & & & & & \\
& \alpha_y & & & \alpha_x & \beta & \gamma_x & & & \gamma_y & & & & & & \\
& & & & & & 0 & & & & & & & & & \\
& & & & & & & 0 & & & & & & & & \\
& & & & \alpha_y & & & \alpha_x & \beta & \gamma_x & & & \gamma_y & & & \\
& & & & & \alpha_y & & & \alpha_x & \beta & \gamma_x & & & \gamma_y & & \\
& & & & & & & & & & 0 & & & & & \\
& & & & & & & & & & & 0 & & & & \\
& & & & & & & & & & & & 0 & & & \\
& & & & & & & & & & & & & 0 & & \\
& & & & & & & & & & & & & & 0 & \\
& & & & & & & & & & & & & & & 0
\end{bmatrix}, \tag{B.11}$$

with the following coefficients,

$$\begin{aligned}
\alpha_x &= \frac{D}{\Delta x^2} + \frac{V_x}{2\Delta x}, \quad \beta_x = -\frac{2D}{\Delta x^2}, \quad \gamma_x = \frac{D}{\Delta x^2} - \frac{V_x}{2\Delta x}, \\
\alpha_y &= \frac{D}{\Delta y^2} + \frac{V_y(t)}{2\Delta y}, \quad \beta_y = -\frac{2D}{\Delta y^2}, \quad \gamma_y = \frac{D}{\Delta y^2} - \frac{V_y(t)}{2\Delta y}, \\
\beta &= \beta_x + \beta_y.
\end{aligned} \tag{B.12}$$

Note that the gap in $A(t)$ between β and α_y and γ_y is of size N_x. This is because, to get entry $u_{i,j\pm1}$ one must travel $\pm N_x$ entries in the in the vector $\mathbf{u}$. One should attempt to validate that the matrix-vector form does indeed give us back the finite difference scheme for when $N_x = N_y = 4$ and when $u_{1,j} = u_{4,j} = u_{i,1} = u_{i,4} = 0$ for all $i, j \in \{1, ..., 4\}$.

Appendix C

Stability and Convergence of The Difference Scheme

C.1 Stability Analysis

Assume that the difference scheme can be written in the form in equation (2.3). Now let $\mathbf{v}(t) = \mathbf{u}(t) + \mathbf{e}(t)$ be a solution slightly perturbed from the true solution $\mathbf{u}(t)$ by an error $\mathbf{e}(t)$. Then, by (2.3),

$$\frac{\partial \mathbf{v}}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{e}}{\partial t} = A(\mathbf{u}+\mathbf{e}) + \mathbf{F} = A\mathbf{u} + \mathbf{F} + A\mathbf{e}.$$

As $\partial \mathbf{u}/\partial t = A\mathbf{u} + \mathbf{F}$, we can assert the following,

$$\frac{\partial \mathbf{e}}{\partial t} = A\mathbf{e}.$$

The perturbation $\mathbf{e}(t)$ therefore obeys the homogeneous difference scheme.

It is known that for semi-discrete schemes, if the spectrum of scaled eigenvalues $\lambda \cdot dt$ for A lie within the domain of stability for the time-marching scheme, then the scheme is stable. If they do not, then the errors $\mathbf{e}$ may grow without bound and the scheme will be unstable [36, 38, 52].

To determine the stability of the numerical scheme, we needed to compute the eigenvalues of A and find where they lie in the complex plane. However, these matrices become quite large as N_x and N_y grow. Thus, we were required to compute the eigenvalues using the $eig()$ function in MATLAB [47]. We computed the eigenvalues λ of A for the 1D and 2D ADE finite difference schemes as given in equations (B.8) and (B.11) respectively. Then we plotted the scaled eigenvectors $\lambda \cdot dt$ against the stability regions of different Runge-Kutta and the forward Euler schemes. The stability regions of the RK4, RK3 and RK2 methods were provided from code made by *Mostafa* [37]. The results are shown in figure C.1 and figure C.2.

As we can see, the eigenvalues indeed sit comfortably inside the region of stability of the RK4 method. Thus the numerical schemes are stable for the chosen parameter values. The eigenvalues can be brought closer to the origin by decreasing dt which would

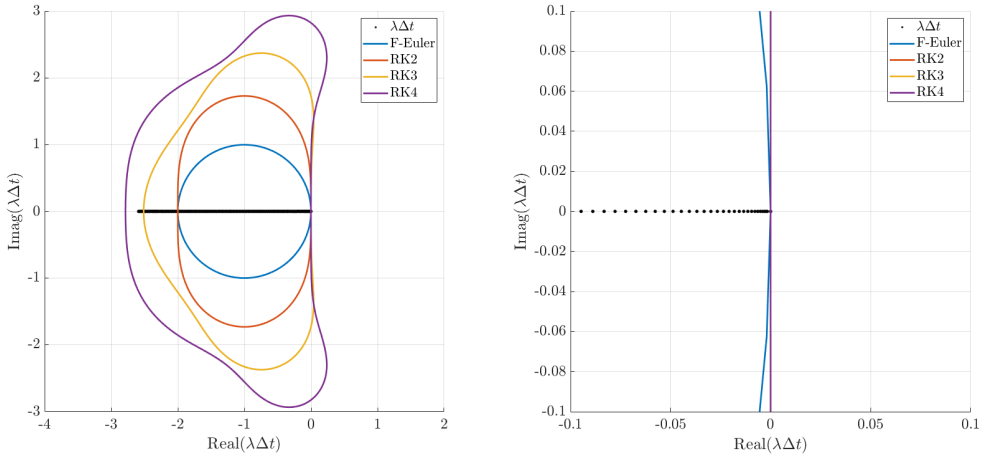

Figure C.1: Scaled eigenvalues $\lambda \cdot dt$ of A for 1D ADE finite difference scheme. Plotted against the stability regions for the explicit RK4, RK3, RK2 and the forward Euler schemes. Left: View of the entire region. Right: Enhanced plot showing that no eigenvalues lie to the right of the origin.

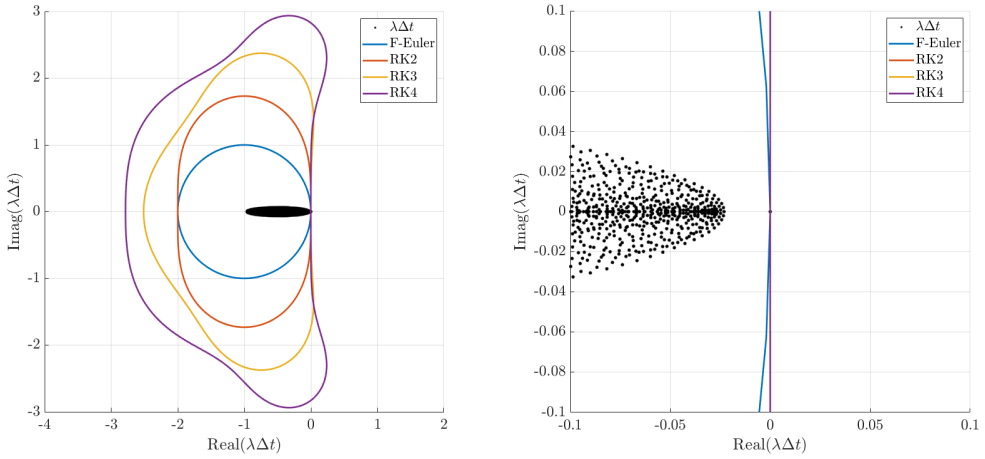

Figure C.2: Scaled eigenvalues $\lambda \cdot dt$ of A for 2D ADE finite difference scheme. Plotted against the stability regions for the explicit RK4, RK3, RK2 and the forward Euler schemes. Left: View of the entire region. Right: Enhanced plot showing that no eigenvalues lie to the right of the origin.

also increase accuracy for the scheme. However, in this instance it was not necessary.

Note that in the 2D case however, we are dealing with a time-dependent matrix $A(t)$. Hence, we decided to apply the principle of frozen coefficients. The principle puts a necessary but not sufficient condition on $A(t)$ at time t. For the scheme with time-dependent variables $A(t)$ to be stable, we require that the same scheme is stable for all values of $A(t)$ at all times $t \in [0,T]$ [6].

We cannot reasonably consider all times $t \in [0,T]$, so instead we checked the eigenvalues of $A(t)$ for the t for which the coefficients take their extrema values. Then, we assert stability, as long as none of the eigenvalues wander out of the stability region. Luckily, we observed practically no changes in the placement of eigenvalues for any change in $A(t)$ over time. All eigenvalues sat comfortably inside of the stability domain. This result was also verified by running several simulations with varying ranges for $A(t)$. The solution for the 2D ADE behaved nicely, even over extended time-periods. Thus we can confidently assert that finite difference scheme (B.10) is also stable. This is not a solid proof as we cannot reasonably test every possible value of $A(t)$ over time. But, we consider it a reasonable argument for stability here. Especially as the numerical experiments suggest the solution behaves nicely even for values far outside the range of values we were interested in.

The ADE as specified by equation (1.7) is a linear PDE. As such, the Lax-Richtmyer equivalence theorem [6] can be applied. The theorem states that for a linear well-posed PDE, as long as the scheme is consistent and stable, then the approximate solution will converge to the true solution as the step sizes become more refined. Well-posedness has been shown for our ADE (see appendix A), consistency has been shown (see section 2.6) and stability has been shown for our scheme. Therefore, we can assert that our numerical methods will converge to the true solution of the PDE, given a fine enough grid.

C.2 When the Numerical Scheme is Not Stable

For the sake of example. We show what happens whenever the eigenvalues land outside of the domain of stability. We increased the time step dt slightly for the 1D finite difference scheme as shown in figure C.1, such that some of the eigenvectors landed outside the stability domain. Then We attempt to plot the solution of the 1D ADE using this time-step dt. The results are shown in figure C.3 and C.4.

As we see, that the solution rapidly develops wild oscillations which blow up. This is why stability needs to be considered, as unstable schemes may give erroneous and non-physical solutions.

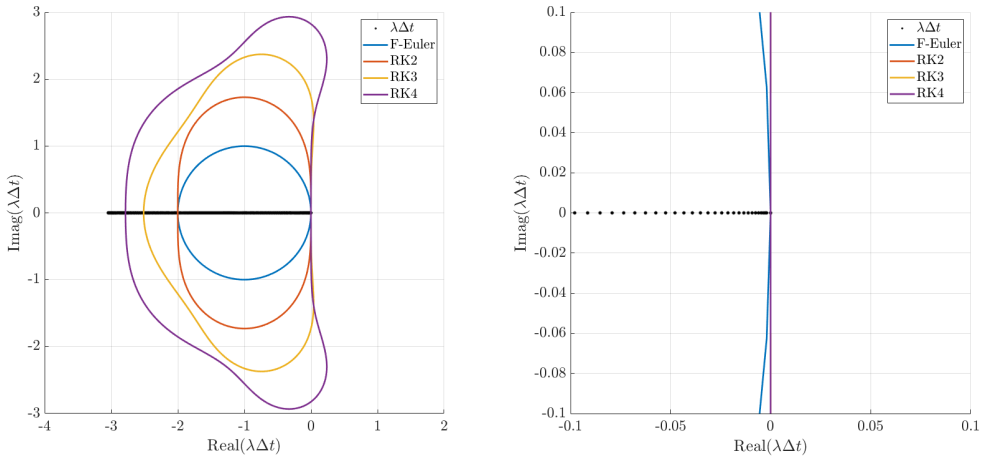

Figure C.3: Scaled eigenvalues $\lambda \cdot dt$ of A for 1D ADE finite difference scheme. Here, dt is increased slightly such that some of the eigenvalues poke outside the stability region of RK4 on the far left. Left: View of the entire region. Right: Enhanced plot showing that no eigenvalues lie to the right of the origin.

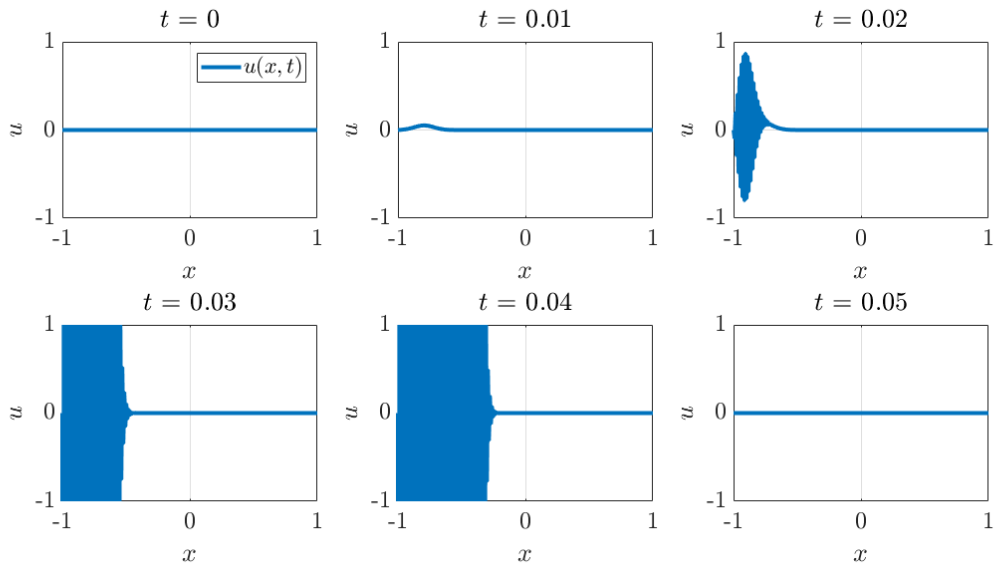

Figure C.4: Corresponding time-series solution for the 1D ADE for when the eigenvalues of A lie outside the stability region of RK4. This shows how the solution develops oscillations which blow up to infinity.

C.3 Convergence Analysis

Now that we know that the schemes are convergent as the step sizes go to zero, i.e., $N_x, N_y \to \infty$, we can determine convergence rate and an estimate of error.

We let dx and dy be defined by the following equations:

$$\Delta x = \frac{L_x}{N_x - 1}$$
$$\Delta y = \frac{L_y}{N_x - 1}$$

Here L_x, L_y are the side-lenghts of the rectangular domain along the x- and y-axis respectively. We conducted several runs of the 1D and 2D finite difference solvers and we reduced Δx and Δy between runs. dt was kept sufficiently small such that the eigenvalues of $A(t)$ obey the conditions discussed in section C.1.

Then, we performed an error estimation by computing the relative error between the solution $\mathbf{u}$ for consecutive runs. This will tell us how the error converges towards zero as the grids become more refined. Figure C.5 and C.6 show for the relative L^2-norm error as defined by equation (5.1) decreases as Δx diminishes for the 1D and 2D case respectively. We see that both schemes converge approximately of order $\mathcal{O}(\Delta x^2)$. In a balance between performance and accuracy, we found it acceptable to settle on a relative L^2-norm error of 0.05% and 0.13% for the 1D and 2D case respectively. Note that in the 2D case, we assumed a square domain with $N_x = N_y$, therefore $\Delta x = \Delta y$.

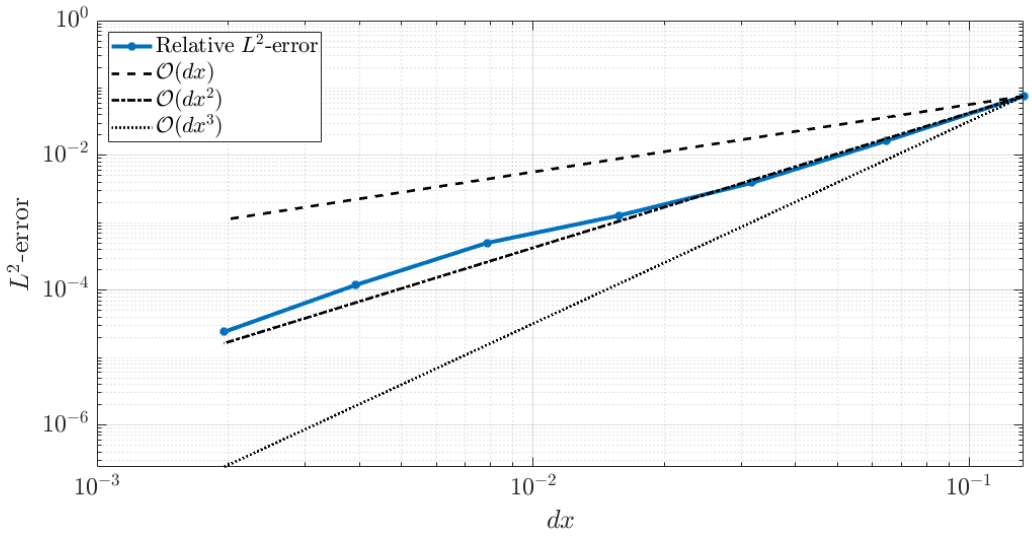

Figure C.5: Convergence rate for the finite difference scheme of the 1D ADE (B.5).

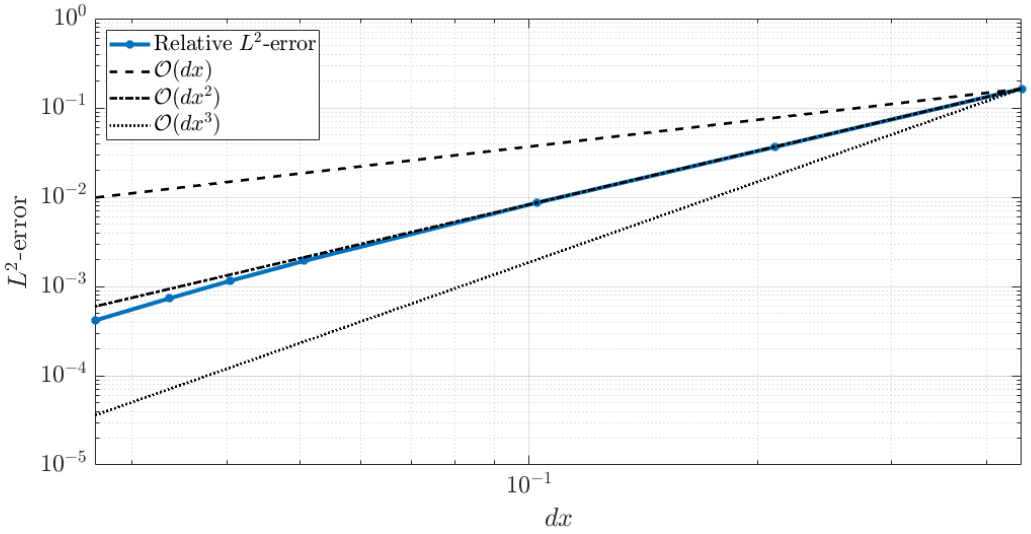

Figure C.6: Convergence rate for the finite difference scheme of the 2D ADE (B.10).

Appendix D

MATLAB Codes

In total there are 10 main MATLAB codes that were used in this project. We will provide them here:

https://github.com/KetilIversen/ADE-PINN-Masters-Thesis-Code.

All MATLAB codes should be able to run as-is. There are also 2 data-sets provided, corresponding to the reference solutions u^* in 1D and 2D which can be used as training-data in the TensorFlow codes.

Appendix E

Tensorflow Codes

There are in total 6 Python Notebooks that were used in this project. They will be provided here:

https://github.com/KetilIversen/ADE-PINN-Masters-Thesis-Code

It is recommended that they are run with Google Colab as that is the platform we developed them on. The path to the training data may need to be given manually by the user.

Bibliography

[1] Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng (2015), TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org. 5.2

[2] ACT (2021), Actom project summary, http://www.act-ccs.eu/actom, accessed: 2021-11-11. 1.1

[3] AI, M. C., and M. with Physics Sciences (2020), Understanding and mitigating gradient flow pathologies in pinn by paris perdikaris, available at: https://www.youtube.com/watch?v=OFUkRgCI6No. 7.2, 7.2, 8.1

[4] Banshoya (2020), Gradient Methods for Source Identification Problems. 1.6.1

[5] Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (2018), Automatic differentiation in machine learning: a survey. 3.1, 3.4, 4.2, 4.2, 5.2

[6] Bertil Gustafsson, J. O., Heinz-Otto Kreiss (2013), *Time Dependent Problems and Difference Methods*, WILEY. 2.6, A.1, C.1

[7] Bilionis, I. (2021), A hands-on introduction to physics-informed machine learning, available at: https://www.youtube.com/watch?v=o9JaZGWekWQ. 9

[8] Bishop, C. M. (2006), *Pattern Recognition and Machine Learning*, Springer. 4.4

[9] Cantor, B. (2020), *The Equations of Materials*, Oxford University Press. 1.2

[10] Chris Rackauckas, J. M. (2021), Physics-informed neural networks (pinns) - chris rackauckas | podcast 42, available at: https://www.youtube.com/watch?v=OmySUTFwh2g. 3.1, 7.2

[11] Ewing, R., and H. Wang. (2001), A summary of numerical methods for time-dependent advection- dominated partial differential equations, *Journal of Computational and Applied Mathematics*, pp. p.423–445. B.1

[12] Glorot, X., and Y. Bengio (2010), Understanding the difficulty of training deep feedforward neural networks, *Journal of Machine Learning Research - Proceedings Track*, *9*, 249–256. 4.3, 4.3

[13] Google, LLC (2021), Google colab, https://colab.research.google.com/, accessed: 2021-05-10. 5.2

[14] Géron, A. (2019), *Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow*, OReilly Media. 4.1, 4.2, 4.3, 4.3, 4.4, 4.4, 4.4

[15] Haghighat, E., and R. Juanes (2021), Sciann, available at: https://www.sciann.com/. 5.2, 6.2.2

[16] Haghighat, E., and R. Juanes (2021), Sciann: A keras/tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks, *Computer Methods in Applied Mechanics and Engineering*, *373*, 113,552, doi:10.1016/j.cma.2020.113552. 5.2, 6.2.2

[17] Hamdi, A. (2011), Inverse source problem in a 2d linear evolution transport equation: Detection of pollution source, *Inverse Problems in Science and Engineering - INVERSE PROBL SCI ENG*, *20*, 1–21, doi:10.1080/17415977.2011.637207. 1.6.2

[18] Hornik, K., M. Stinchcombe, and H. White (1989), Multilayer feedforward networks are universal approximators, *Neural Netw.*, *2*(5), 359366. 3.1

[19] Jacquier, P. (2019), Pinns-tf2.0, available at: https://github.com/pierremtb/PINNs-TF2.0. (document), 5.2

[20] Jagtap, A., and G. Karniadakis (2020), 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*, *28*, 2002–2041, doi:10.4208/cicp.OA-2020-0164. 3.4

[21] Jin, X., S. Cai, H. Li, and G. E. Karniadakis (2021), Nsfnets (navier-stokes flow nets): Physics-informed neural networks for the incompressible navier-stokes equations, *Journal of Computational Physics*, *426*, 109,951, doi:10.1016/j.jcp.2020.109951. 4.3, 7.2

[22] Kadeethum, T., T. M. Jørgensen, and H. M. Nick (2020), Physics-informed neural networks for solving inverse problems of nonlinear biot's equations: Batch training. 4.4, 4.4

[23] Kingma, D. P., and J. Ba (2017), Adam: A method for stochastic optimization. 4.4

[24] Krishnapriyan, A. S., A. Gholami, S. Zhe, R. M. Kirby, and M. W. Mahoney (2021), Characterizing possible failure modes in physics-informed neural networks. 3.1, 3.4, 4.3, 7.2, 7.3, 7.3

[25] Liu, D. C., and J. Nocedal (1989), On the limited memory bfgs method for large scale optimization, *Mathematical Programming*, *45*(1-3), 503528, doi:10.1007/bf01589116. 4.4

[26] Lu, L., X. Meng, Z. Mao, and G. E. Karniadakis (2021), Deepxde, available at: https://deepxde.readthedocs.io. (document), 3.4, 5.2, 6.2.2, 7.1, 9

[27] Lu, L., X. Meng, Z. Mao, and G. E. Karniadakis (2021), Deepxde: A deep learning library for solving differential equations, *SIAM Review*, *63*(1), 208228, doi: 10.1137/19m1274067. (document), 1.2, 3.4, 5.2, 6.2.2, 7.1, 9, 10

[28] Mamonov, A. V., and Y.-H. R. Tsai (2013), Point source identification in nonlinear advectiondiffusionreaction systems, *Inverse Problems*, *29*(3), 035,009, doi: 10.1088/0266-5611/29/3/035009. 1.6.3

[29] Markidis, S. (2021), The old and the new: Can physics-informed deep-learning replace traditional linear solvers? 3.4, 4.3, 4.4

[30] McClenny, L. (2021), Self-adaptive physics-informed neural networks using a soft attention mechanism by levi mcclenny, available at: https://www.youtube.com/watch?v=UTC6cccEEnM. 7.2

[31] McClenny, L., and U. Braga-Neto (2020), Self-adaptive physics-informed neural networks using a soft attention mechanism. 3.4, 4.3, 7.2, 7.2, 7.3

[32] McClenny, L. D., M. A. Haile, and U. M. Braga-Neto (2021), Tensordiffeq, available at: https://docs.tensordiffeq.io/. 5.2

[33] McClenny, L. D., M. A. Haile, and U. M. Braga-Neto (2021), Tensordiffeq: Scalable multi-gpu forward and inverse solvers for physics informed neural networks. 5.2

[34] Mishra, S., and R. Molinaro (2021), Estimates on the generalization error of physics informed neural networks (pinns) for approximating pdes. 3.1, 3.4, 4.3, 6.2.1

[35] Mishra, S., and R. Molinaro (2021), Estimates on the generalization error of physics informed neural networks (pinns) for approximating a class of inverse problems for pdes. 3.1, 3.4, 6.2.1

[36] MIT OpenCourseWare (2021), Eigenvalue stability of finite difference methods, available at: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-90-computational-methods-in-aerospace-engineering-spring-2014/numerical-methods-for-partial-differential-equations/eigenvalue-stability-of-finite-difference-methods/1690r-matrix-stability-for-finite-difference-methods/. C.1

[37] Mostafa, G. M. F. B. (2021), Domain of stability for runge kutta method, https://www.mathworks.com/matlabcentral/fileexchange/76688-domain-of-stability-for-runge-kutta-method, accessed: 2021-09-12. C.1

[38] O. Kreiss, H., and L. Wu (1993), On the stability definition of difference approximations for the initial boundary value problem, *Applied Numerical Mathematics*, *12*(1-3), 213–227, doi:10.1016/0168-9274(93)90119-C. C.1

[39] Perdikaris, P. (2021), Ddps | "when and why physics-informed neural networks fail to train" by paris perdikaris, available at: https://www.youtube.com/watch?v=xvOsV106kuA. 7.2

[40] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2007), *Numerical Recipes The Art of Scientific Computing Third Edition*, Cambridge University Press. 4.4

[41] Raissi, M. (2019), Hidden physics models: Machine learning of non-linear partial differential equations, available at: https://www.youtube.com/watch?v=GXZq2_WYRjo. 3.1, 3.4, 4.3

[42] Raissi, M., A. Yazdani, and G. E. Karniadakis (2018), Hidden fluid mechanics: A navier-stokes informed deep learning framework for assimilating flow visualization data. 3.4, 4.3, 10

[43] Raissi, M., P. Perdikaris, and G. Karniadakis (2019), Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, *Journal of Computational Physics*, *378*, 686–707, doi:10.1016/j.jcp.2018.10.045. (document), 3.1, 3.1, 3.1, 3.4, 4.1, 4.3, 4.3, 4.3, 4.4, 4.5, 5.2, 6, 6.2.2, 7.1, 7.2, 10

[44] Ramachandran, P., B. Zoph, and Q. V. Le (2017), Searching for activation functions. 6.2.2

[45] Shibata, R. (2017), Flow around a square cylinder at low speed limit, source code and report available at: https://www.rickshibata.com/projects/cfd/. 1.3

[46] Shin, Y., J. Darbon, and G. E. Karniadakis (2020), On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type pdes, *Communications in Computational Physics*, *28*(5), 20422074, doi:10.4208/cicp.oa-2020-0193. 3.1, 6.2.1

[47] The Mathworks, Inc. (2021), *MATLAB version R2021a*, Natick, Massachusetts. 1.2, 5.1, C.1

[48] Wang, S., and P. Perdikaris (2021), Deep learning of free boundary and stefan problems, *Journal of Computational Physics*, *428*, 109,914, doi:10.1016/j.jcp.2020.109914. 7.2

[49] Wang, S., Y. Teng, and P. Perdikaris (2020), Understanding and mitigating gradient pathologies in physics-informed neural networks. 3.4, 4.3, 7.2, 7.2, 7.3, 8.1, 9, 10

[50] Wang, S., X. Yu, and P. Perdikaris (2020), When and why pinns fail to train: A neural tangent kernel perspective. 3.4, 7.2, 7.3

[51] Wight, C. L., and J. Zhao (2020), Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks. 4.3, 7.2, 7.3, 7.3

[52] Zingg, D. W., and M. Lederle (2005), On linear stability analysis of high-order finite-difference methods, doi:10.2514/6.2005-5249. C.1