UNIVERSITETET I OSLO

Department of Physics

FYS-STK4155 - Applied Data Analysis and Machine Learning

Numerical And Computational Methods For Partial Differential Equations

Author:
Mathias Svoren & Jonas Semprini Næss

Date: 20th December 2023

Table of Contents

List of Figures

List of Tables

1 Abstract

This study investigates methods for solving the one-dimensional heat equation under specific initial and boundary conditions. The FTCS (Forward Time-Centered Space) method, a combination of the Forward-Euler method for explicit time and the centered-difference method for spatial derivatives, is employed with spatial step sizes (Δx) of 0.1 and 0.01, and a fixed time step size of $(\Delta x)^2/2$.

Additionally, a Physics-Informed Neural Network (PINN) is implemented, producing a continuous and differentiable function that approximates the heat equation. Comparative analysis against the FTCS method reveals worse performance, especially with $\Delta x = 0.01$. Despite a slightly higher mean squared error, the neural network's continuous and differentiable flexibility makes it a favorable choice for implementation on more data sparse problems.

In summary, the study explores the application of neural networks for solving differential equations, demonstrating promise for simpler problems.

2 Introduction

Mathematics and computers have long been used to understand and predict things happening in the real world. One important tool for this are differential equations, which help describe the complex movements in various scientific areas. As we started studying more complicated systems, we needed a more advanced type of equation called partial differential equations (PDEs). These equations are crucial for explaining processes like fluid flow, heat transfer, and quantum mechanics, where things change in both space and time.

Solving PDEs using traditional methods becomes especially intricate when dealing with complex shapes and behaviors, making many problems impossible to solve analytically. That is why scientists turned to numerical methods – a practical way to find solutions by approximating them instead of solving them exactly.

In recent years, there has been a big change in how we approach these problems. We have started using artificial intelligence, especially neural networks, as part of our computational toolkit. This is particularly useful for PDEs that do not have neat analytical solutions, like the challenging Navier-Stokes equations that describe fluid dynamics. Neural networks are great at recognizing patterns and adapting to complicated relationships, helping us find solutions by learning from data.

This paper explores different ways we use computers to solve partial differential equations. We will stretch our scope from well-documented mathematical methods to the more recent use of numbers and PINNs (Physics-Informed Neural Networks). The goal is introduce and elucidate on how these computational methods can play a crucial role in helping us understand the complexities of physical systems described by partial differential equations, while comprehending the basics of implementation.

3 Theory

This study focuses on solving partial differential equations (PDEs) numerically. The theoretical section will cover both the foundations of PDEs as well as motivations for using Physics-informed neural networks (PINNs) to obtain sufficient results. Moreover, will there be elaborations on known numerical solvers to capture a broader spectrum of possible implementations.

3.1 Partial Differential Equations

3.1.1 Overview

A partial differential equation (PDE) constitutes an equation wherein an undetermined function of two or more variables and its partial derivatives concerning these variables are involved. The order of a partial differential equation corresponds to the highest order of its derivatives. Illustratively, the equation:

$$x\frac{\partial u}{\partial x} - y\frac{\partial u}{\partial y} = u \quad \text{with} \quad u = x^2 \quad \text{on} \quad y = x,\ 1 \leq y \leq 2 \tag{1}$$

exemplifies a first-order partial differential equation.

A solution to a partial differential equation is defined as any function that consistently satisfies the equation. A general solution, in turn, refers to a solution in which a set of arbitrary independent functions, the count of which, aligns with the order of the equation. Conversely, a particular solution derives when specific selections of these arbitrary functions are made from the general solution. For instance, consider the general solution of the partial differential equation (Equation 1):

$$u(x, y) = x\sqrt{xy} \tag{2}$$

A singular solution is one that cannot be derived through a specific selection of arbitrary functions within the framework of a general solution.

A boundary value problem associated with a partial differential equation aims to identify all solutions that adhere to specified conditions termed boundary conditions. In cases where time is a variable, the solution u (or $\frac{\partial u}{\partial t}$, or both) is required to meet initial conditions at $t = 0$.

The wave equation is a frequently encountered scenario of this, expressed in its one-dimensional form as follows:

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial^2 u}{\partial t^2}$$

Here, A is a constant, and the solution u is contingent on both spatial and temporal variables, denoted as $u(x, t)$. In the two-dimensional context, the solution becomes $u(x, y, t)$.

This equation finds application in various contexts, such as acoustic waves, seismic waves, heat conduction, optics and electromagnetic fields (Hjorth-Jensen 2015). For instance, in the case of electromagnetic waves, the constant A is represented as c^2, where c signifies the speed of light. Extending this equation to two or three dimensions is a straightforward process. In two dimensions, it can be expressed as:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = A\frac{\partial^2 u}{\partial t^2}.$$

In this project, our main focus is on another fundamental equation widely used in the natural sciences—the diffusion equation.

3.1.2 The Diffusion Equation (1D)

Diffusion equations serve as models for alterations in the concentration of a variable within a defined region concerning spatial and temporal variables. If the value of this variable at a specific space-time point is represented by a continuous function $u : V \times \mathbb{R}^+ \to \mathbb{R}$ (where $V \in \mathbb{R}^3$ signifies

the volume of interest), the evolution of this variable is governed by the partial differential equation:

$$\frac{\partial u}{\partial t} + \nabla \cdot \Gamma = 0 \tag{3}$$

where Γ is the flux of the diffusing substance or material. The one-dimensional diffusion equation, also known as the heat equation, is hence given by:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \tag{4}$$

where $u(x,t)$ is the temperature distribution, D is the diffusion coefficient, t is time, and x is the spatial coordinate. Equation (4) readily derives from (3) when integrated with respect to Fick's first law. Fick's first law posits that the flux of diffusing material within the system is proportional to the local density gradient, expressed as:

$$\Gamma = -D\nabla u(\mathbf{r}, t).$$

Analytical Solution

Consider an IVP for the diffusion equation in one dimension:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \tag{5}$$

on the interval $x \in [0, L]$ with initial condition

$$u(x, 0) = \sin(\pi x), \quad \forall x \in (0, L)$$

and Dirichlet boundary conditions

$$u(0, t) = u(L, t) = 0 \quad \forall t > 0$$

To find the analytical solution, we use separation of variables. Assume a solution of the form:

$$u(x, t) = X(x)T(t)$$

Substitute this into the heat equation and rearrange terms to get:

$$\frac{1}{D}\frac{T'}{T} = \frac{X''}{X} = -\lambda$$

This results in two ordinary differential equations (ODEs). The time component equation:

$$T'(t) + \lambda D T(t) = 0$$

with the solution:

$$T(t) = Ce^{-\lambda D t}$$

The spatial component equation:

$$X''(x) + \lambda X(x) = 0$$

with the solution:

$$X(x) = A\sin(\sqrt{\lambda}x) + B\cos(\sqrt{\lambda}x)$$

Applying the initial condition $u(x, 0) = \sin(\pi x)$, we find that $B = 0$ and $\sqrt{\lambda} = \pi$, so $\lambda = \pi^2$. Therefore, the spatial component solution becomes:

$$X(x) = A_n \sin(\pi x)$$

The coefficients A_n for the Fourier series expansion of the initial condition $u(x,0) = \sin(\pi x)$ are given by:

$$A_n = \frac{2}{L} \int_0^L u(x,0) \sin\left(\frac{n\pi x}{L}\right) dx$$

In this case, L is the length of the rod, and $u(x,0) = \sin(\pi x)$. Let's compute A_n for the given initial condition:

$$A_n = 2 \int_0^1 \sin(\pi x) \sin(n\pi x)\, dx$$

The integral is non-zero only when $n = 1$, and the formula simplifies to:

$$\begin{aligned} A_1 &= 2 \int_0^1 \sin(\pi x) \sin(\pi x)\, dx \\ &= 2 \int_0^1 \sin^2(\pi x)\, dx \\ &= 2 \left[\frac{x}{2} - \frac{\sin(2\pi x)}{4\pi}\right]_0^1 \\ &= 1 \end{aligned}$$

Therefore, the coefficient A_n is 1 when $n = 1$ and is zero for all other n. The Fourier series expansion for the initial condition is:

$$u(x,0) = \sin(\pi x) = A_1 \sin(\pi x)$$

The general solution is then given by the sum of the spatial and temporal components:

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x) e^{-n^2\pi^2 Dt}$$

where A_n are the coefficients we calculated. In this specific case, the solution is:

$$u(x,t) = \sin(\pi x) e^{-\pi^2 Dt} \tag{6}$$

3.2 Numerical Solvers

3.2.1 Forward-Euler

The forward Euler method, a foundational numerical technique in computational mathematics, is a straightforward and explicit algorithm for approximating solutions to ordinary differential equations (ODEs). This method is characterized by its simplicity and ease of implementation, making it a widely employed tool in various scientific and engineering disciplines. By iteratively advancing the solution in small time increments, the forward Euler method provides a computationally efficient approach for approximating the evolution of dynamic systems described by differential equations.

Given an ODE of the form $\frac{du}{dt} = f(u,t)$, the method proceeds by updating the solution u at discrete time points t_n using the recurrence relation:

$$u_{n+1} = u_n + hf(u_n, t_n)$$

Here, h represents the chosen time step. The method's formulation derives from a first-order Taylor expansion, expressing the solution at t_{n+1} based on the information at t_n. Precisely, the expansion

yields:

$$u(t_{n+1}) = u(t_n) + h\frac{du}{dt}\Big|_{t_n} + \frac{h^2}{2}\frac{d^2u}{dt^2}\Big|_{t_n} + \cdots \tag{7}$$

The local or truncation error introduced at each step is proportional to h^2, of the method's first-order accuracy/convergence. Meaning the local truncation error can be deduced from first two terms being precisely equal to the right-hand side of the Forward Euler Method formula. The magnitude of the remaining terms can be observed to scale as $\mathcal{O}(h^2)$ meaning the error is of quadratic complexity.

While the Forward Euler method's simplicity and computational efficiency make it a popular choice, careful consideration of the time step is crucial for maintaining stability and accuracy, particularly when presented wih stiff ODEs.

3.2.2 Runge-Kutta

The Runge-Kutta method is a powerful numerical technique widely used for solving ordinary differential equations (ODEs). It offers improved accuracy compared to simpler methods like the forward-Euler. By employing a multi-stage process to calculate intermediate values, Runge-Kutta provides a precise and efficient approach to approximating the evolution of dynamic systems described by ODEs. Its versatility and accuracy make it a popular choice in various scientific and engineering applications.

In the forward Euler approach, the solution at the next time step is estimated by utilizing information about the slope or derivative of u at the current time step. This method yields a Local Truncation Error (LTE) of $\mathcal{O}(h^2)$, classifying it as a first-order numerical technique (i.e covered in the previous section). Contrarily, Runge-Kutta methods constitute a class of techniques that systematically utilises the information of slopes from multiple points to project the solution into the subsequent time step.

To elucidate and illustrate its purposes we start out by deriving the second-order Runge-Kutta (RK) method, where the LTE is enhanced to $\mathcal{O}(h^3)$. Considering the Initial Value Problem (IVP) defined in Equation (7), a time step h, and the solution u_n at the n-th time step, we aim to compute u_{n+1} as follows: Consider the first-order ordinary differential equation (ODE) of the form:

$$\frac{du}{dt} = f(t, u)$$

with an initial condition $u(t_0) = u_0$. The goal is to derive the second-order Runge-Kutta (RK) method. The general form of the RK method is given by:

$$\begin{aligned} k_1 &= hf(t_n, u_n) \\ k_2 &= hf(t_n + \alpha h, u_n + \beta k_1) \\ u_{n+1} &= u_n + \gamma k_1 + \delta k_2 \end{aligned}$$

where $\alpha, \beta, \gamma, \delta$ are constants to be determined.

Now, let us compute k_2 by substituting the expressions for t_n, u_n, and k_1 into the ODE:

$$\begin{aligned} k_2 &= hf(t_n + \alpha h, u_n + \beta k_1) \\ &= hf(t_n + \alpha h, u_n + \beta h f(t_n, u_n)) \end{aligned}$$

To obtain a second-order accurate method, we want to choose $\alpha, \beta, \gamma, \delta$ such that the method is consistent with the Taylor series expansion and has a local truncation error of $\mathcal{O}(h^3)$.

Now, let us express k_2 as a Taylor series expansion up to the third order:

$$\begin{aligned} k_2 &= hf(t_n + \alpha h, u_n + \beta h f(t_n, u_n)) \\ &= h\left[f(t_n, u_n) + \frac{\partial f}{\partial t}\alpha h + \frac{\partial f}{\partial u}\beta h f(t_n, u_n)\right] + \mathcal{O}(h^3) \end{aligned}$$

where we have used the chain rule and neglected terms of order $\mathcal{O}(h^2)$ and higher since the method is exact for u to the h^2 term. Now, let's substitute this expression for k_2 into the RK method:

$$\begin{aligned} u_{n+1} &= u_n + \gamma k_1 + \delta k_2 \\ &= u_n + (\gamma + \delta)hf(t_n, u_n) + \delta h^2(\alpha\frac{\partial f}{\partial t} + \beta\frac{\partial f}{\partial u})(t_n, u_n) + \mathcal{O}(h^3) \end{aligned}$$

To match the terms in the Taylor expansion and achieve $\mathcal{O}(h^3)$ LTE, we set $\delta + \gamma = 1$, $\delta\beta = \frac{1}{2}$ and $\delta\alpha = \frac{1}{2}$. Solving these equations simultaneously yields the coefficients for the second-order RK method:

$$\alpha = 1, \quad \beta = 1, \quad \gamma = \frac{1}{2}, \quad \delta = \frac{1}{2}$$

Thus, the second-order Runge-Kutta method is given by:

$$\begin{aligned} k_1 &= hf(t_n, u_n) \\ k_2 &= hf(t_n + h, u_n + hk_1) \\ u_{n+1} &= u_n + \frac{k_1 + k_2}{2} \end{aligned}$$

This method yields $\mathcal{O}(h^3)$ error complexity and provides a more accurate approximation compared to the forward Euler method.

Likewise, higher-order Runge-Kutta methods can be systematically derived. Notably, the fourth-order Runge-Kutta (RK4) technique stands out as extensively applied for solving initial value problems (IVPs). Having error of order $\mathcal{O}(h^5)$, makes RK4 represent a robust numerical approach. The method is given below:

$$\begin{aligned} k_1 &= hf(t_n, u_n), \\ k_2 &= hf(t_n + \frac{h}{2}, u_n + \frac{k_1}{2}), \\ k_3 &= hf(t_n + \frac{h}{2}, u_n + \frac{k_2}{2}), \\ k_4 &= hf(t_n + h, u_n + k_3), \\ u_{n+1} &= u_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \end{aligned}$$

3.2.3 Forward time-centered space (FTCS)

The Forward Time Centered Space (FTCS) scheme is a numerical method used for solving the heat equation and, more generally, parabolic partial differential equations. In this approach, we approximate spatial derivatives at the current time step and the time derivative between the current and new time steps. Meaning

$$t = t_0 + n\Delta t, \quad x = x_0 + i\Delta x, \tag{8}$$

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}, \tag{9}$$

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}, \quad \frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} \tag{10}$$

Where, u_i^{n+1} represents the solution at spatial position i and time step n.

The FTCS scheme yields first-order accuracy in time and up to second-order accuracy in space. It can be consistently reformulated in the following expression:

$$u^{n+1} = u_i^n + \Delta t f(u_i^n, u_{i+1}^n, u_{i-1}^n) \tag{11}$$

Numerical Stability:

The FTCS scheme is an explicit method, where the new time step relies solely on the current time step. Despite its quick implementation and execution per time step, a significant drawback is its requirement for very small Δt to maintain numerical stability. In explicit schemes like FTCS, any effect can only propagate by a maximum of one spatial grid block in one time step, leading to stability challenges. For instance, in a purely 1st-order system, FTCS is unconditionally unstable (Imperial-College-London 2023).

$$\frac{\partial u}{\partial t} = -\mu \frac{\partial u}{\partial x} \quad \rightarrow \quad u_i^{n+1} = u_i^n - \Delta t \mu \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} \tag{12}$$

3.3 Neural Network

3.3.1 Overview

A neural network, is a foundational element in artificial intelligence, which emulates human brain functions to process data. Operating as a variety of machine learning known as deep learning, neural networks utilize layered structures of interconnected nodes, resembling the organization of the human brain. This configuration establishes an adaptive system, enabling computers to learn from errors and progressively enhance their performance through the given models. In such manner they can be employed to solve various tasks from facial recognition to regression analysis and classification problems, which is what this study is chiefly regarding.

This partial section was gathered from the paper: "**Classification and Regression, From Linear and Logistic Regression to Neural Networks**" by the authors (Svoren et al. 2023), where a more comprehensive analysis and framework was implemented to study a feed-forward neural network.

3.3.2 General setup and Layers

The first model of artificial neurons were introduced in 1943 to investigate brain signal processing (Hjorth-Jensen 2023a). The concept emulates the neural networks in the human brain, where billions of neurons communicate via electrical signals. Each neuron accumulates incoming signals, requiring them to surpass an activation threshold for output. Failure to meet the threshold results in neuron inactivity, signifying zero output. This behavior has motivated a basic mathematical model for an artificial neuron, which could be presented as follows

$$y = f\left(\sum_{i=0}^{n} w_i x_i\right)$$

where y is the output of that neuron, with corresponding signals $x_0, x_1, \ldots, x_n$ and weights $w_0, w_1 \ldots, w_n$. The function f is often defined as the activation function or threshold function.

Now extending the concept of a neuron, we can look at multiple neurons interacting through layers, as illustrated in Figure 1.

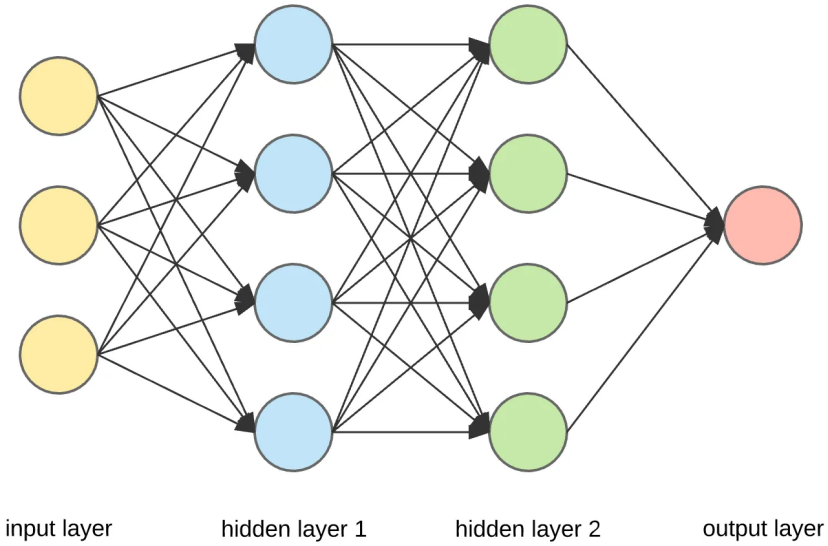


Figure 1: A visual of how a simple Artifical Neural Network looks in concept, with two hidden layers.

For each node i in the inital hidden layer, we compute the weighted sum z_i^1 of the input coordinates x_j, such that

$$z_i^1 = \sum_{j=1}^{M} w_{i,j}^1 x_j + b_j^1$$

where b_i is known as the bias which typically is required when activation weights or inputs are zero. The value z_i^1 serves as the argument for the activation function f_i of each node i. M represents all potential inputs to a node i in the first layer. The collective output y_i^1 of all neurons in layer 1 is then defined as:

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{i,j}^1 x_j + b_j^1\right)$$

where we have assumed identical activation functions for all nodes within the same layer, denoted as f. If on the other hand there happens to be different or several activation functions in different hidden layers then we can identify the output as:

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right)$$

where N_l is the node count in layer l. After computing the initial hidden layer nodes' output, the subsequent values of later layers are determined iteratively until obtaining the final output.

Considering this, we can further extend the network by examining the output of neuron i in layer 2, described as follows:

$$\begin{aligned} y_i^2 &= f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \\ &= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \end{aligned}$$

Generalizing this to an Artifical Neural Network (ANN) with l hidden layers, then yields the comprehensive functional form

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^2\right) + b_1^3\right]. \tag{13}$$

The functional form 13 encompasses what is known as a Feed-Forward Neural Network (FFNN), which is a type of neural network that considers uni-directional flow of information.

3.4 Schedulers

Various optimizer algorithms exhibit distinct strengths and weaknesses, and the optimal choice depends on the specific problem. In the context of our Physics-informed Neural Network (PINN) model, two noteworthy optimizers are **ADAM** and **Limited-memory BFGS** (L-BFGS). The subsequent sections elaborate on the details of these two optimization algorithms.

3.4.1 ADAM

The Adam Optimizer or simply ADAM is a gradient descent optimization algorithm known for its efficiency in handling large datasets and numerous parameters, requiring minimal memory. Conceptually, it combines aspects of the 'gradient descent with momentum' and 'RMSprop' algorithms.

ADAM maintains an exponentially decaying average of past squared gradients and past gradients, akin to momentum, which can be pictured as adding friction to a rolling object down an incline plane.(Algorithm 8.7 from Goodfellow et al. 2016)

Algorithm 1: The ADAM optimizer/algorithm

Data: Step size $\boldsymbol{\epsilon}$ (Suggested default: 0.001
Data: Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$ (Suggested defaults: 0.9 and 0.999 respectively)
Data: Small constant δ, somewhere in the region of 10^-8 to account for numerical stability
Data: Initial parameters $\boldsymbol{\theta}$
Initialize 1.st and 2.nd moment variables $\boldsymbol{s} = \boldsymbol{r} = 0$;
Initialize time step $t = 0$;
while termination criterion not met **do**
  Sample a minibatch of m examples from the training set $\left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\right\}$ with corresponding targets $\boldsymbol{y}^{(i)}$;
  $t \leftarrow t + 1$;
  Update biased first moment estimate: $\boldsymbol{s} \leftarrow \boldsymbol{s}\rho_1 + (1 - \rho_1)\boldsymbol{g}$;
  Update biased second moment estimate: $\boldsymbol{r} \leftarrow \boldsymbol{r}\rho_2 + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$;
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L((f(\boldsymbol{x}^{(i)}; \theta),\ \boldsymbol{y}^{(i)})$;
  Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$;
  Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$;
  Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{\epsilon}\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$ (operations applied element-wise) ;
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

3.4.2 Limited-memory BFGS (L-BFGS)

The L-BFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) algorithm serves as a Quasi-Newton method tailored for optimizing convex and differentiable objective functions. It operates iteratively, approximating solutions until convergence. Some authors, such as (Raissi et al. 2019), favor L-BFGS for Physics-informed Neural Network (PINN) applications.

Before getting into the full functionality of L-BFGS, foundational concepts such as line search and the Wolfe condition require definition. The complete BFGS algorithm will be presented thereafter, and for a more in-depth exploration, readers can refer to (Wright and Nocedal 1999).

Line search, a method in optimization, efficiently determines a suitable step size (α) along the search direction vector ($\mathbf{p}$) for an iterative optimization algorithm. The iteration for step $k+1$ follows $x_{k+1} = x_k + \alpha_k \mathbf{p}_k$. Line search is inexact, exploring a sequence of candidate values for α_k and terminating upon satisfying certain conditions.

The Wolfe conditions, essential for Quasi-Newton methods, consist of two conditions:

1. Sufficient Decrease Condition: Ensures the objective function f decreases by a specified amount for a given step size, indicating progress toward the optimal solution. Mathematically, this condition is expressed as:

$$f\left(x_{k+1}\right) = f\left(x_k + \alpha p_k\right) \leq f\left(x_k\right) + c_1 \alpha \nabla f_k^T p_k$$

For $c_1 \in (0, 1)$, a commonly employed value for L-BFGS is $c_1 = 10^{-4}$. Geometrically, this choice positions the objective value $f\left(x_{k+1}\right)$ below the slope of the line $\varkappa(\alpha)$ as illustrated in Figure 2.

However, relying solely on the sufficient decrease condition proves inadequate since it holds for all sufficiently small α, as evident from Figure 2. To address excessively short steps, a complementary requirement, termed the curvature condition, is introduced.

2. The Curvature Condition: This condition assesses the gradients of f_k and f_{k+1}, imposing the following inequality constraint:

$$\nabla f\left(x_{k+1}\right)^T p_k = \nabla f\left(x_k + \alpha_k p_k\right)^T p_k \geq c_2 \nabla f_k^T p_k$$

Here, $c_2 \in (c_1, 1)$, with a standard value for L-BFGS being $c_2 = 0.9$.

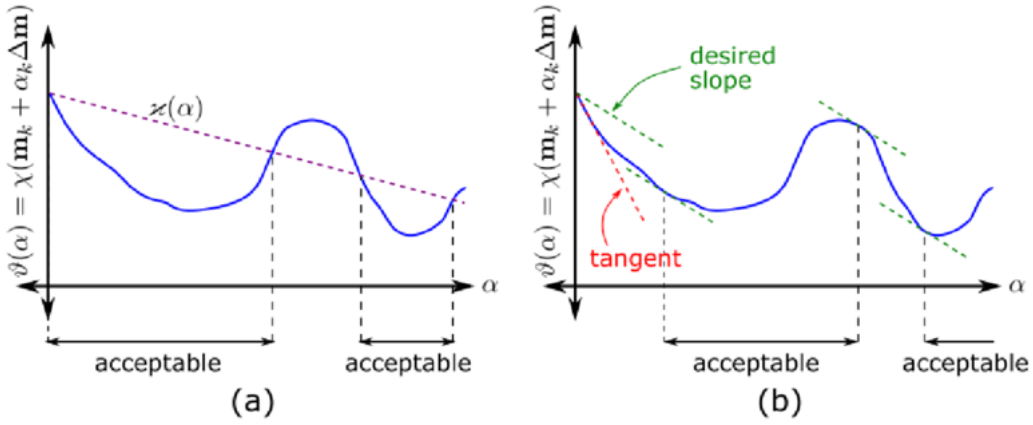


Figure 2: Sufficient decrease condition (a) and the curvature condition (b).

While supplementing the Wolfe conditions, one can further observe that excessive movement could lead to a positive gradient. To address this, line search can be constrained by limiting positive values. This is achieved by considering the absolute values of the gradient in the curvature condition. This set of constraints is commonly known as the strong Wolfe conditions.

$$f\left(x_k + \alpha_k p_k\right) \leq f\left(x_k\right) + c_1 \alpha_k \nabla f_k^T p_k \tag{14}$$

$$\left|\nabla f\left(x_k + \alpha_k p_k\right)^T p_k\right| \leq c_2 \left|\nabla f_k^T p_k\right|, \tag{15}$$

For $c_1 \in (0, 1)$ and $c_2 \in (c_1, 1)$. The PyTorch implementation of L-BFGS incorporates the strong Wolfe condition, utilizing default values of $c_1 = 10^{-4}$ and $c_2 = 0.9$.

Having discussed the step size α, we now explore the computation of the direction vector $\mathbf{p}$. BFGS employs an approximation B_k of the true Hessian matrix H, describing the curvature of the objective function. The search direction is the negative of the steepest descent direction, denoted by the equation $p_k = -B_k^{-1}\nabla f_k$, where ∇f_k is the gradient.

Notation-wise, the following vectors are defined:

$$s_k = x_{k+1} - x_k = \alpha_k p_k, \quad y_k = \nabla f_{k+1} - \nabla f_k$$

The BFGS update rule for B_{k+1}, shown to be (Wright, Nocedal et al. 1999), is:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

This update rule forms the basis for the subsequent algorithm.

Algorithm 2: BFGS Method Algorithm

Data: Starting point x_0, convergence tolerance $\epsilon > 0$, Hessian approximation B_0; $k \leftarrow 0$
while $\|\nabla f_k\| > \epsilon$ **do**
    Compute search direction: $p_k = -B_k^{-1}\nabla f_k$;
    Set $x_{k+1} = x_k + \alpha_k p_k$, where α_k is computed by line search satisfying Wolfe conditions;
    Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$;
    Compute B_{k+1} by the BFGS update rule;
    $k \leftarrow k + 1$;

Each iteration involves $\mathcal{O}\left(n^2\right)$ arithmetic operations, in addition to the costs associated with function and gradient evaluations.

The algorithm is robust and exhibits a superlinear convergence rate, making it suitable for practical purposes. Despite Newton's method achieving quadratic convergence (in general), its per-iteration cost is usually higher due to the computation of second derivatives and solving linear systems.

The BFGS algorithm has a limited-memory variant, L-BFGS, which retains only the m most recent correction pairs (s_i, y_i) while discarding others.

The core idea behind L-BFGS is that curvature information from recent iterations is most relevant to the current behavior of the Hessian. The algorithm's efficiency is improved by discarding information from earlier iterations. The choice of m depends on the problem, with values between 3 and 20 often yielding satisfactory results. However, L-BFGS tends to converge slowly on ill-conditioned problems (Wright and Nocedal 1999).

3.5 Physics-informed neural networks (PINN)

Neural networks typically demand substantial data for effective training, posing challenges in scientific and engineering applications where such extensive datasets may be scarce. Physics-informed Neural Networks (PINNs) offer a solution by embedding prior knowledge of physical laws into the network. This constrains the solution space to align with known principles, allowing for effective learning with reduced data. For instance, informing the network about the law of energy conservation in pendulum motion serves as a practical illustration of this approach.

3.5.1 Mathematical Modeling and Computational Approaches

A generic nonlinear partial differential equation is expressed as:

$$u_t + N[u; \lambda] = 0, \quad x \in \Omega, \quad t \in [0, T]$$

Where, $u(t, x)$ signifies the solution, $N[;\lambda]$ represents a parameterized nonlinear operator denoted by λ, and Ω is a subset of $\mathbb{R}^D$. This broad formulation covers various challenges in mathematical physics, such as conservative laws, diffusion processes, and more (Raissi et al. 2019)

When dealing with noisy measurements from a dynamic system described by the equation mentioned earlier, Physics-Informed Neural Networks (PINNs) can be strategically employed to address two distinct problem classes:

1. Data-driven solution
2. Data-driven discovery

These classes involved to find solutions from observed data and to discover the underlying partial differential equations, respectively.

3.5.2 Data-Driven Solution of Partial Differential Equations

The data-driven solution of partial differential equations (PDEs) involves the computation of the latent state $u(t, x)$ of a system based on given boundary data or measurements y and fixed model parameters λ. The main equation to model is (Raissi et al. 2019):

$$u_t + N[u] = 0, \quad x \in \Omega, \quad t \in [0, T]$$

This method defines the residual $r(t, x)$ as:

$$r := u_t + N[u] = 0$$

By approximating $u(t, x)$ with a deep neural network, which can be differentiated using automatic differentiation, the parameters of $u(t, x)$ and $r(t, x)$ are learned by minimizing the composite loss function $\mathcal{C}_t$:

$$\mathcal{C}_t = \mathcal{C}_u + \mathcal{C}_r$$

Here, $\mathcal{C}_u = \|u - y\|_{L_p}$ [1] quantifies the error between the Physics-Informed Neural Network (PINN) $u(t, x)$ and the set of boundary conditions and measured data defined on its point space or domain $\mho$.[2] Additionally, $L_r = \|r\|_{L_p}$ represents the mean-squared error of the residual function. The latter term helps the PINN capture the underlying structure of the partial differential equation as it undergoes training.

This approach has demonstrated efficacy in generating computationally efficient physics-informed surrogate models, finding applications in forecasting physical processes, predictive control modeling, multi-physics and multi-scale modeling, and simulation.

3.5.3 Cost function

For a Physics-Informed Neural Network (PINN) to learn effectively, it is necessary to create a cost function for optimizing the network, aiming to minimize the overall cost. The desired cost function is the mean sum of losses across all data points, where each loss corresponds to a singular data point.

For PINNs, the loss function comprises two principal components: one addressing the network's predictive error concerning concentration data at (x, y, t), and the other accounting for the residual

[1] The $\|\cdot\|_{L_p}$ corresponds to the L_p norm
[2] Most commonly $\mathbb{R}^d$

error in the diffusion process at a collocation point. Collocation points, pivotal in approximating solutions to partial differential equations (PDEs), specifically evaluate the PDE's residual at defined locations within its domain, referred to as PDE points. Carefully choosing these points has a big impact on how accurate the solution is. Using methods that focus on residuals involves removing less relevant points and adding more relevant ones. This strategy aims to improve how information is incorporated, and this refining process happens step by step.

Ultimately, the two aforementioned loss terms are aggregated, potentially scaled with a weight factor to adjust their relative significance. The initial step involves utilizing the diffusion equation to delineate the residual function $r(x, y, t)$ as:

$$r(x, y, t) = \frac{\partial c}{\partial t} - D\nabla^2 c \tag{16}$$

If D is accurately predicted, the residual becomes zero. The corresponding residual cost function is then defined by computing the L_p norm and taking the mean sum over all PDE points.

$$\mathcal{C}_r = \frac{1}{\mho_r} \sum_{i,j,k}^{\mho_r} \|r(x_i, y_j, t_k)\|_{L_p} \tag{17}$$

where $\mho_r$ is the number of PDE-points. The concentration data loss is subsequently defined as follows:

$$\mathcal{C}_c = \frac{1}{\mho_c} \sum_{i,j,k}^{\mho_c} \|c_{nn}(x_i, y_j, t_k) - c_d(x_i, y_j, t_k)\|_{L_p} \tag{18}$$

In this expression, $c_{nn}(x_i, y_j, t_k)$ represents the neural network-predicted concentration at (x_i, y_j, t_k), and $c_d(x_i, y_j, t_k)$ denotes the actual data sample concentration at the corresponding point. $\mho_c$ signifies the total number of data points.

Thus are we able to express the total cost of our model as

$$\mathcal{C}_T = \mathcal{C}_c + \mathcal{C}_r \tag{19}$$

where in many cases weights $w_c, w_r \in \mathbb{R}$ are multiplied to the terms respectively to scale in accordance to importance or measures of correction.

Using this cost function, we can set up an optimization problem to learn the parameters of the neural networks $c_{nn}(x, y, t)$ by minimizing the overall cost.

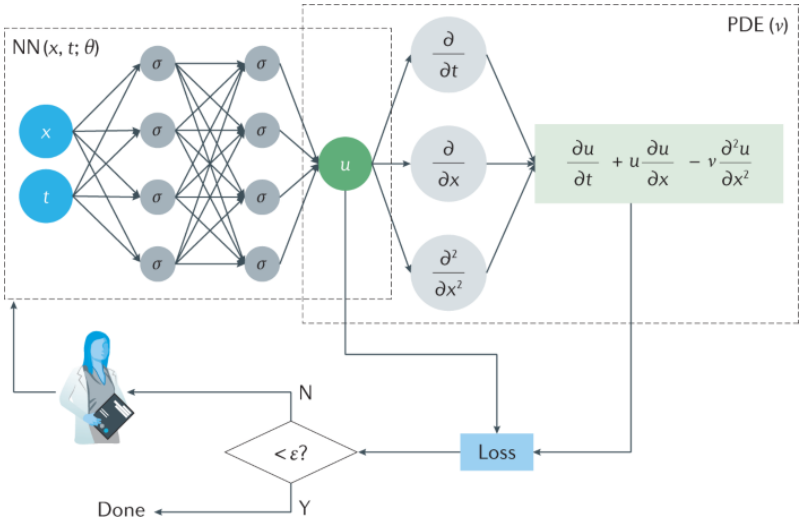


Figure 3: PINN architecture of a fully connected neural network

4 Method

In this section, we explore different methods for modeling and approximating the analytical solution of the presented 1D diffusion equation. Additionally, we provide insights into specific nuances and dependencies that might be needed in individual implementation.

4.1 Forward time-centered space: "Numerical Approach"

We presented the physical problem we aim to address in Section (3.1.2), focusing on modeling the temperature gradient in a rod with length $L = 1$. This involves solving a one-dimensional diffusion equation.

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t},\ t > 0,\ x \in (0, L) \tag{20}$$

with initial conditions

$$u(x, 0) = \sin(\pi x) \quad 0 < x < L, \tag{21}$$

and according boundary conditions

$$u(0, t) = 0\ t \geq 0, \tag{22}$$

and

$$u(L, t) = 0\ t \geq 0. \tag{23}$$

We can also write the difference equation as

$$u_{xx} = u_t \tag{24}$$

Where the function $u(x, t)$ can be the temperature gradient of a rod. As time increases, the velocity approaches a linear variation with x. We can introduce different step lengths for the space-variable x and the time-variable t through the step length for x given as

$$\Delta x = \frac{1}{n+1} \tag{25}$$

and the step length for time is Δt. So then the position after i steps and j steps in time is given as

$$t_j = j\Delta t,\ j \geq 0$$
$$x_i = i\Delta x,\ 0 \leq i \leq n+1$$

We can then rewrite u_{xx} and u_t with the derivative approximation.

We focus on the explicit forward Euler algorithm, utilizing discretized time via a forward formula and centered difference in space:

$$u_t \approx \frac{u(x, t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t} \tag{26}$$

$$u_{xx} \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta, t_j)}{\Delta x^2} \tag{27}$$

The corresponding difference equation is expressed as:

$$\frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t} = \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta, t_j)}{\Delta x^2} \tag{28}$$

The explicit forward Euler algorithm is defined as:

$$y_{n+1} = y_n + \Delta t f(t_n, y_n). \tag{29}$$

The algorithm for solving the difference equation is:

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha u_{i+1,j} \tag{30}$$

where $\alpha = \Delta t/\Delta x^2$.

Our objective is to implement the explicit scheme algorithm and conduct tests for $\Delta x = 0.1$ and $\Delta x = 0.01$. The stability limit of the explicit scheme dictates

$$\Delta t/\Delta x^2 \leq 1/2 \Rightarrow \Delta t \leq \Delta x^2/2 \tag{31}$$

4.2 Forward time-centered space: "Computational approach"

The method starts off by computing a stable time step (Δt) as a product of a predefined factor (0.50) and the square of the spatial step size $(\Delta x)^2$. Arrays `x` and `t` are then generated to discretize spatial and temporal grids. The spatial array (`x`) spans from `0` to the rod´s length (L) with intervals of size (Δx), while the temporal array (`t`) ranges from `0` to the maximum time (`T_max`) with intervals of size (Δt). [3]

Subsequently, `Nx` and `Nt` represent the number of spatial and temporal grid points, respectively, and arrays `u` and `u_prev` are initialized to store the solution at the current and previous time steps. The initial condition for `u_prev` is set using the initial condition $u(x, 0) = \sin(\pi x)$. The coefficient (α) for the finite difference scheme is set to $\frac{\Delta t}{(\Delta x)^2}$ (i.e Equation 30).

The method runs a nested loop structure where the outer loop iterates through time steps (`Nt` iterations), and the inner loop traverses spatial grid points (`Nx` iterations). Within this structure, the Forward Euler scheme updates the solution at each spatial point based on neighboring points and the previous time step.

The solution `u` at the current time step is updated using the Forward Euler formula (i.e Equation 30) including values from the previous time step (`u_prev`). Following the completion of a time step, the current solution becomes the previous solution for the next iteration.

Ultimately, the method returns the numerical solution (`u`) and the spatial grid (`x`).

4.3 Physics-Informed Neural Network (PINN)

The PINN was implemented in Python 3.11 as a class inheriting from the PyTorch `nn.Module` parent base-class module. The class is designed to request the number of hidden nodes, layers, and desired activation function, determining the network's density. `nn.Linear` modules are used to establish fully connected layers, with two input nodes and a single output node. The forward pass utilizes `nn.Sequential` to activate the entire layered network as a single module. To aid initialization, a **Normal Xavier Initialization** method is applied to weight each layer (Glorot and Bengio 2010). Biases are initialized to zero.

For cost calculation, a trial solution $g_t = h_1(x,t) + h_2(x, N(x,t,P))$ [4] is implemented (Hjorth-

[3] We adapt the notation of Δt and Δx here to be consistent with Section 4.1 and Section 3.1.2. Note that this is changed to dt and dx in the code respectively.

[4] $g_t = (1-t)\sin(\pi x) + x(1-x)tN(x,t)$ where $N(x,t)$ represents the forward pass in our neural network for values x, t

Jensen 2023b), easing the calculation of derivatives:

$$u_t = \frac{\partial g_t}{\partial t} \tag{32}$$

$$u_x = \frac{\partial g_t}{\partial x} \tag{33}$$

$$u_{xx} = \frac{\partial^2 g_t}{\partial x^2} \tag{34}$$

Resulting in:

$$\mathcal{C}_{PINN} = u_t - u_{xx} \Rightarrow \ell_{PINN} = \text{MSE}(\mathcal{C}_{PINN}) \tag{35}$$

Here, ℓ_{PINN} represents the loss in our prediction.

4.4 Data

We deemed it unnecessary to split our data into training and test data. For this problem we construct 40 uniformly spaced values for both $x \in [0, 1]$ and $t \in [0, 1]$. We then create a meshgrid of our x and t values which we use for both training and testing. Due to the linearity of the constructed data, we expect no over-fitting to training data. Conducting some testing yielded insignificant differences in mean square error between training and test data, confirming our initial decision.

4.4.1 Automatic Differentiation (PyTorch)

PyTorch's autograd package, featuring automatic differentiation, is essential for computing gradients during the backward propagation of a neural network. Autograd functions, tracking operations in the forward pass, efficiently calculate gradients during the backward pass. This automation streamlines the training process, contributing to accurate and optimized learning in neural networks.

5 Results

After exploring implementation methods in accordance to the theoretical framework, we proceed to present more rigorous results. We assess model performance by comparing it to the analytical solution (Equation 6), depicted in Figure (4). The section is segmented into two parts, focusing on explicit scheme and PINN results individually.

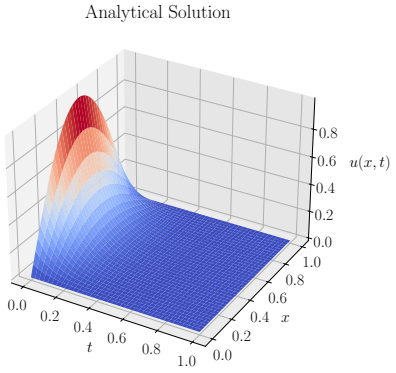
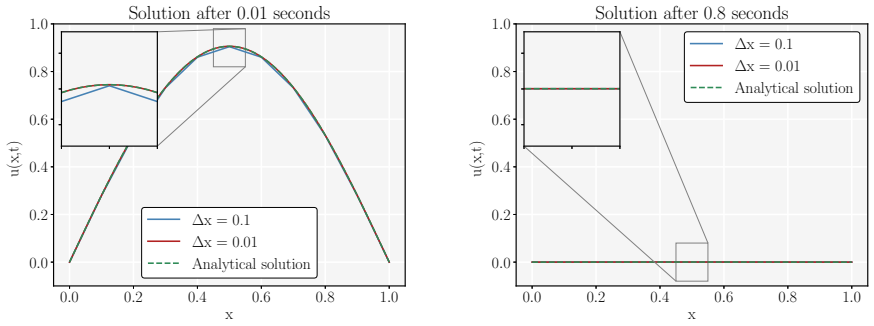


Figure 4: 3D surface plot of the analytical solution to the 1D diffusion equation.

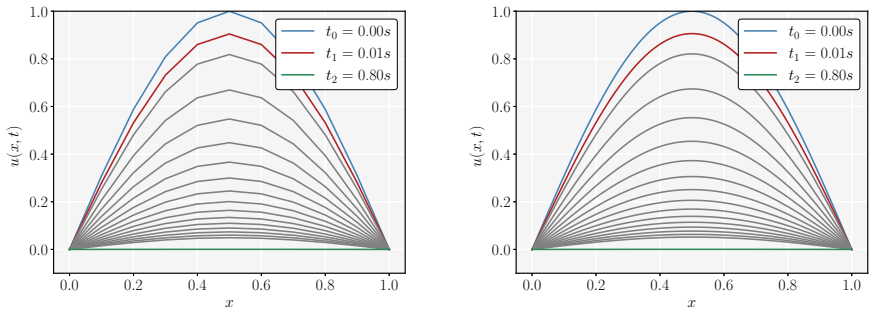
5.1 Forward time-centered space (FTCS)

We study $t_1 = 0.01s$ and $t_2 = 0.80s$, where $u(x, t_1)$ is smooth and significantly curved, and $u(x, t_2)$ is at the stationary state. Figure 5 compares the methods with the analytical result, revealing that using $\Delta x = 0.01$ yiels a better approximation to the analytical solution. In Figure 6 we get a two-dimensional overview of the evolution of $u(x, t)$ as t increases.



(a) Solution after $t_1 = 0.01$ seconds. (b) Solution after $t_2 = 0.8$ seconds.

Figure 5: 2D-plots of comparison between the analytical and approximated solution to the 1D diffusion equation

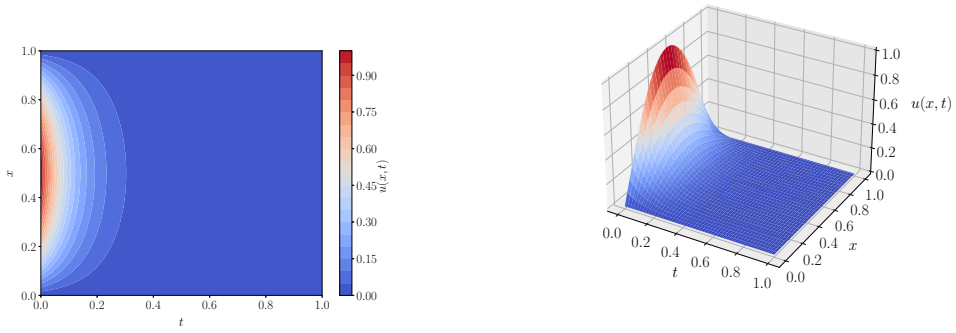


(a) (b)

Figure 6: Evolution of the 1D diffusion equation through the FTCS scheme.

For $\Delta x = 0.01$ our iterative method achieved a mean squared error of $3.4 \cdot 10^{-10}$ compared to the analytical result.

Figure 7 shows the contour plot and a three-dimensional surface plot of our diffusion equation.



(a) (b)

Figure 7: (a) Contour plot of predicted values (FTCS) (b) 3D surface plot of FTCS approximation.

5.2 Physics-Informed Neural Network (PINN)

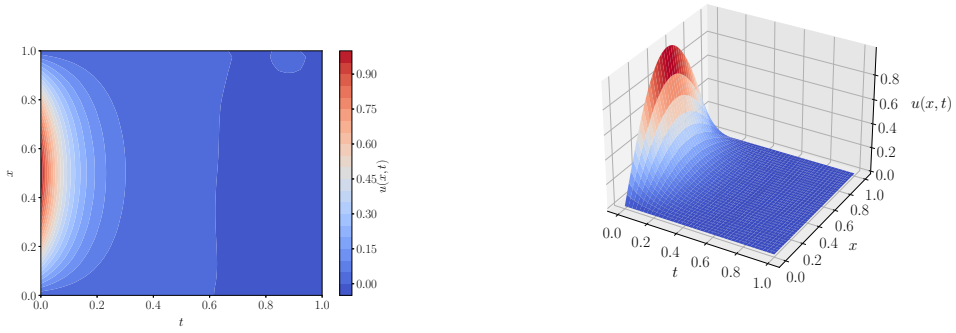
5.2.1 Grid-Search

Our optimal model, detailed in Table 1, came from a thorough grid-search analysis, the outcomes of which are provided in Appendix B. The grid-search involved exploring learning rates in the range of $[1.0 \times 10^{-4}, 1.0 \times 10^{-1}]$ and weight decay in $[1.0 \times 10^{-4}, 1.0]$ (Figure 9). Additionally, we scrutinized hidden layers (range: $[1, 5]$) and nodes per layer (range: $[10, 100]$) in Figure 10. Figure 11 presents the outcomes of the grid-search focusing on epochs ($[100, 500, 1000, 2000, 50000]$) and activation functions [Tanh, Sigmoid, ReLU, LeakyReLU, ELU].

Table 1: Parameters for best PINN model.

	PINN
Learning rate	0.01
Weight decay	0.0001
Layers	4
Nodes	80
Activation function	Tanh
Epochs	5000

The top-performing model achieves a loss of 3.9×10^{-4}, with a mean squared error of 1.4×10^{-7} compared to the analytical solution on the same dataset. Figure 8 presents contour and 3D surface plots illustrating the predicted values from the best model.



(a) (b)

Figure 8: (a) Contour plot of predicted values (PINN) (b) 3D surface plot of PINN prediction

6 Discussion

6.1 Explicit Scheme vs Neural Network

The examination of mean squared error outcomes derived from our explicit numerical scheme and the Physics-Informed Neural Network (PINN) demonstrates a notable efficiency advantage in favor of the explicit scheme, which exhibits a speed enhancement of approximately 425-fold. Moreover, the computational runtime of the explicit scheme markedly surpasses that of the neural network training process. Finally, the explicit scheme stands out for its remarkable ease of implementation.

These results are not unexpected since the Forward time-centered space (FTCS) algorithm was constructed to solve the heat equation and similar parabolic PDEs (Imperial-College-London 2023). However our neural network managed to solve the problem yielding low mean squared error. This is great news because looking at the generalization properties of the Physics-Informed Neural Network, it can we utilized to solve more advanced ODEs and PDEs.

Due to the stability limit described in Equation 31, Δt is significantly smaller than Δx. This greatly limits our ability to decrease Δx, since this will quickly escalate to a problem too complex to solve with limited hardware resources. This is an inherent weakness of the explicit scheme. This also limits the accuracy of the method given that limited data is available. Whereas a PINN can train a descriptive model on smaller amounts of data.

6.2 Trial Solution

Through implementation (i.e Section 4.3) it was found that utilising a reasonable trial solution proved highly sufficient when modeling the PDE. As such being the impact on optimising the cost and loss magnitudes, which result in greater accuracy. However, it is important to understand that "reasonable" is an awfully loose term, where determining a suitable trial solution can be both intricate and complex. Methods and rules for which trial solutions are determined vary greatly with the problem, making it a non-stringent issue, while also having to adhere to potentially large network architectures. On many occasions the trial solution is determined simply from the boundary and initial conditions corresponding to the ODE/PDE, which is the case in this study (i.e Section 4.3), but in general this proves to be more deeply layered when considering sub-domains, collocation and optimising methods such as the Galerkin Method (Al-Nimr et al. 1994). In essence, addressing problems effectively requires more than assessing their geometries and intuitively selecting trial solutions. It involves conducting parameter searches, such as grid-searches, for different optimizers and regularization parameters. Additionally, considering transformative techniques like Laplace and Fourier transforms proves beneficial for simplifying complexities and converting constraints

into valuable information. This holds true, particularly for unsteady partial differential equations (PDEs).

6.3 Activation Functions

Figure 11 presents the outcomes of a grid-search, comparing epochs against activation functions, including Tanh, Sigmoid, ReLU, LeakyReLU, and ELU. The results highlight Tanh as the preferred activation function for this specific problem. Notably, ReLU and LeakyReLU exhibit poor performance, potentially due to their strict nature in handling negative values, leading to a phenomenon known as "dying ReLU." This issue, where the activation outputs are of the same value for any input, is particularly problematic for small datasets, stopping the network's ability to recover.

Given that Tanh maps zero inputs to values near zero, it provides a natural mechanism to prevent "neuron deaths" and facilitates recovery if the network is on the verge of shutting down. Hence making it the most suitable choice of the ones studied.

7 Conclusion

We explored methods for solving the one-dimensional heat equation with specified initial and boundary conditions as defined in Equation (20 $\rightarrow$ 23). Employing a finite difference approach, we approximated the time derivative using the Forward-Euler method explicit in time, while the second spatial derivative was approximated with the centered-difference method. This combined technique, termed the FTCS (Forward Time-Centered Space) method, utilized spatial step sizes of $\Delta x = 0.1$ and $\Delta x = 0.01$, with the time step size fixed at $\Delta t = (\Delta x)^2/2$. A finer spatial resolution ($\Delta x = 0.01$) significantly improved accuracy, albeit at the cost of increased computational steps.

Furthermore, we employed a neural network with 4 hidden layers, each containing 80 and activated by the Tanh activation function. This neural network yielded a continuous and differentiable function, effectively approximating the heat equation solution under the given conditions. Comparatively, it underperformed compared to the FTCS method, particularly when $\Delta x = 0.01$.

In conclusion, employing neural networks for solving differential equations shows promise, especially for limited amounts of data and showcasing a generalized model for solving PDEs. Scaling up problems to avoid round-off errors is advisable when operating at smaller scales. For complex equations like the previously mentioned Navier-Stokes equations, neural networks may require extensive training and tuning, rendering finite-difference methods, with their ease of implementation, more efficient for larger problems.

Bibliography

Glorot, Xavier and Yoshua Bengio (2010). 'Understanding the difficulty of training deep feedforward neural networks'. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hjorth-Jensen, Morten (2015). *Computational Physics*. https://www.asc.ohio-state.edu/physics/ntg/6810/readings/Hjorth-Jensen_lectures2015.pdf.

— (2023a). *Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week41.html (visited on 9th Nov. 2023).

— (2023b). *Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week43.html#the-trial-solution (visited on 11th Dec. 2023).

Imperial-College-London (2023). *Primer on Programming in Python and Mathematical/Computational Techniques for Scientists and Engineers*. URL: https://primer-computational-mathematics.github.io/book/c_mathematics/numerical_methods/7_FTCS.html%20(FTCS) (visited on 9th Dec. 2023).

Al-Nimr, M.A., M. Al-Jarrah and A.S. Al-Shyyab (1994). 'Trial solution methods to solve unsteady PDE'. In: *International Communications in Heat and Mass Transfer* 21.5, pp. 742–753. ISSN: 0735-1933. DOI: https://doi.org/10.1016/0735-1933(94)90075-2. URL: https://www.sciencedirect.com/science/article/pii/0735193394900752.

Raissi, Maziar, Perdikaris Paris and E Karniadakis George (2019). 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations'. In: *Journal of Computational Physics 378*.

Svoren, Mathias, Jonas Semprini Næss and Rebecca Nguyen (2023). 'Classification and Regression, From Linear and Logistic Regression to Neural Networks'. In: URL: https://github.uio.no/mathihs/Project-2-Classification-and-Regression-/blob/mathihs/Project_2-2.pdf.

Wright, Stephen and Jorge Nocedal (1999). *Numerical Optimization*. https://link.springer.com/book/10.1007/978-0-387-40065-5.

Appendix

A GitHub repository

https://github.uio.no/mathihs/Project_3

B Grid-search

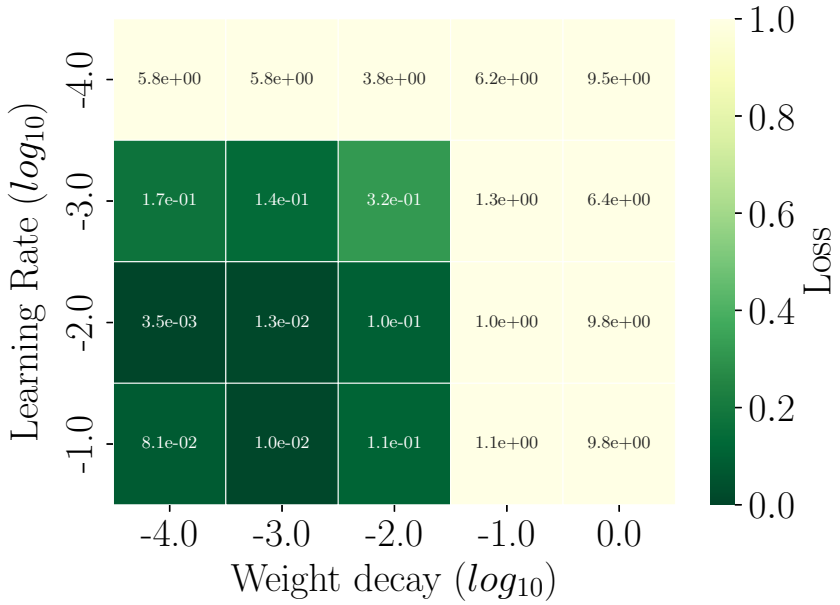


Figure 9: Grid-search of learning rate and weight decay.

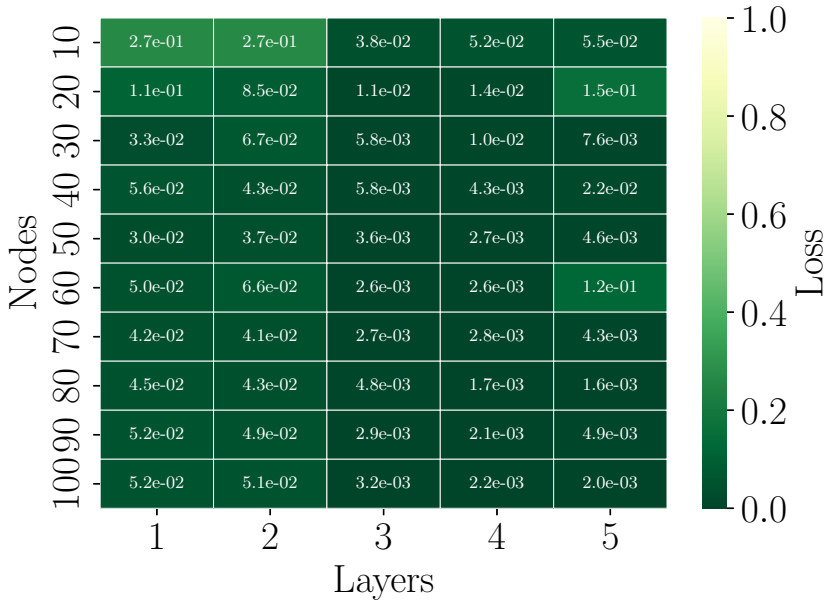


Figure 10: Grid-search of hidden layers and nodes.

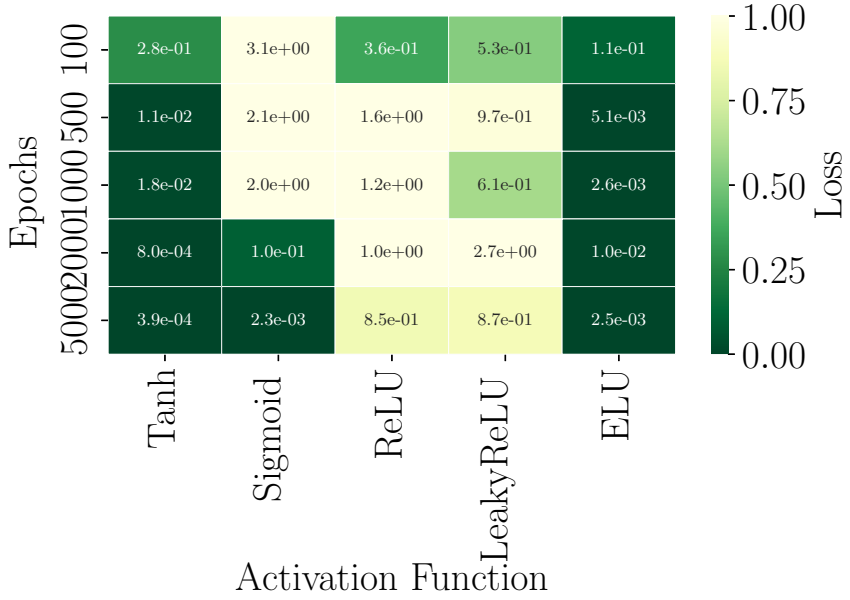


Figure 11: Grid-search of epochs and activation functions.