UNIVERSIDADE FEDERAL DO PARANÁ

LARRY STEFFEN BERTONCELLO

# REDES NEURAIS GUIADAS PELA FÍSICA: IMPLEMENTAÇÕES MANUAIS E AUTOMATIZADAS NA SOLUÇÃO DAS EQUAÇÕES DE BURGERS, KORTEWEG-DE VRIES E KORTEWEG-DE VRIES MODIFICADA

CURITIBA

2024

LARRY STEFFEN BERTONCELLO

# REDES NEURAIS GUIADAS PELA FÍSICA: IMPLEMENTAÇÕES MANUAIS E AUTOMATIZADAS NA SOLUÇÃO DAS EQUAÇÕES DE BURGERS, KORTEWEG-DE VRIES E KORTEWEG-DE VRIES MODIFICADA

Trabalho apresentado à banca examinadora da Universidade Federal do Paraná, como requisito parcial à obtenção do Título de Bacharel em Matemática Industrial.

Orientador: Prof. Dr. Roberto Ribeiro Santos Junior
Coorientador: Prof. Dr. Thiago Oliveira Quinelato

# AGRADECIMENTOS

Primeiramente, agradeço aos meus orientadores, os professores Roberto Ribeiro Santos Júnior e Thiago Oliveira Quinelato, pela coragem de embarcarem nesta jornada comigo, explorando um tema desafiador e pouco familiar, com determinação e vontade de aprenderem junto comigo. Sua orientação e apoio foram fundamentais para a realização deste trabalho.

Agradeço também aos meus pais, Pedro e Sonia, que sempre estiveram ao meu lado me incentivando e oferecendo forças nos momentos mais difíceis. Um agradecimento especial à minha mãe, Sonia, cujo amor e apoio incondicionais foram minha maior inspiração.

À minha namorada e melhor amiga, Paula, que esteve ao meu lado em todos os momentos, oferecendo força, companhia e compreensão, mesmo nas fases mais complexas deste trabalho. Sem você, minha sanidade teria sido colocada à prova. Sua presença foi essencial para que eu pudesse superar os desafios e seguir em frente.

Aos membros do LABFLUID – Laboratório de Dinâmica dos Fluidos Computacional, por todo o apoio durante os estudos, pelas ideias compartilhadas e pelas discussões enriquecedoras que contribuíram significativamente para a construção deste trabalho.

A todos vocês, minha mais sincera gratidão.

"*A matemática é o alfabeto com o qual Deus escreveu o universo*".
(Galileu Galilei)

# RESUMO

Este trabalho investiga a aplicação de Redes Neurais Guiadas pela Física (PINNs) como abordagem moderna e eficiente para a solução de Equações Diferenciais Parciais (EDPs), com foco nas equações de Burgers, Korteweg-de Vries (KdV) e Korteweg-de Vries modificada (mKdV). Ao enfrentar o desafio de desenvolver métodos computacionais precisos e adaptáveis para EDPs com características complexas, este estudo apresenta duas abordagens complementares: uma implementação manual e outra automatizada utilizando a biblioteca `DeepXDE`. A implementação manual permite um maior controle sobre cada etapa do processo, sendo ideal para experimentos que demandam ajustes detalhados ou a exploração de configurações específicas. Por outro lado, a abordagem automatizada é de implementação mais simples, mas limita o controle do usuário sobre alguns processos e o deixa sujeito a possíveis alterações na biblioteca utilizada. Além disso, diferentemente das metodologias de PINNs comumente encontradas na literatura para problemas de valor inicial, introduzimos um parâmetro que permite a dependência da solução numérica em relação à condição inicial do problema. Isso representa um avanço em comparação com os métodos clássicos e aplicações anteriores de PINNs, que exigem simulações independentes para cada escolha de parâmetro no problema de valor inicial. Os resultados mostram que ambas as abordagens são eficazes e atendem a diferentes demandas no contexto de pesquisa e aplicação. Assim, este trabalho contribui para a disseminação do uso das PINNs, oferecendo uma introdução acessível em língua portuguesa e promovendo o acesso de novos pesquisadores à área.

**Palavras-chave:** Redes Neurais, Redes Neurais Guiadas pela Física (PINNs), Equações Diferenciais Parciais (EDPs)

# ABSTRACT

This work investigates the application of Physics-Informed Neural Networks (PINNs) as a modern and efficient approach for solving Partial Differential Equations (PDEs), focusing on the Burgers' equation, the Korteweg-de Vries (KdV) equation, and the modified Korteweg-de Vries (mKdV) equation. To address the challenge of developing computational methods that are both accurate and adaptable to PDEs with complex characteristics, this study presents two complementary approaches: a manual implementation and an automated implementation using the `DeepXDE` library. The manual implementation provides greater control over each stage of the process, making it ideal for experiments requiring fine-tuning or exploration of specific configurations. In contrast, the automated approach simplifies the development process but limits user control over some operations and exposes the user to potential updates in the library. Additionally, unlike conventional PINN methodologies commonly found in the literature for initial value problems, we introduce a parameter that allows the numerical solution to depend on the initial condition of the problem. This represents an advancement compared to classical methods and prior PINN applications, which require independent simulations for each parameter choice in initial value problems. The results demonstrate that both approaches are effective and address different demands in research and application contexts. Thus, this work contributes to the dissemination of PINNs, offering an accessible introduction in Portuguese and promoting entry for new researchers into this growing field.

**Keywords:** Neural Networks, Physics-Informed Neural Networks (PINNs), Partial Differential Equations (PDEs)

# Lista de Figuras

# Conteúdo

# INTRODUÇÃO

As equações diferenciais parciais (EDPs) desempenham um papel fundamental na modelagem de fenômenos naturais e de processos em diversas áreas do conhecimento. Elas aparecem em problemas que envolvem a propagação de ondas, como acústica e eletromagnetismo [8, 17], dinâmica de fluidos e transferência de calor [16, 43], difusão de substâncias e processos biológicos [11, 32]. Além disso, as EDPs têm aplicações significativas em contextos mais modernos, como a modelagem de sistemas financeiros, onde descrevem o comportamento de derivativos e taxas de variação no tempo [32, 44], na mecânica quântica para representar o comportamento de partículas subatômicas por meio da equação de Schrödinger [11, 38], e na climatologia, para modelar a dinâmica atmosférica e oceânica [15, 42].

Apesar de sua ampla aplicabilidade, muitas EDPs apresentam desafios intrínsecos, seja pela dificuldade de obtenção de soluções analíticas, seja pela impossibilidade de resolvê-las de forma exata em cenários mais complexos. Tradicionalmente, métodos numéricos como diferenças finitas, volumes finitos e métodos espectrais têm sido amplamente utilizados para obter soluções aproximadas [6, 29]. Contudo, essas abordagens enfrentam limitações relevantes, como alta demanda computacional [41], dificuldade em lidar com geometrias irregulares [34], condições iniciais e de contorno não triviais [29], além de apresentarem dificuldades em problemas de alta dimensionalidade devido ao fenômeno conhecido como "maldição da dimensionalidade" [2].

Nos últimos anos, o surgimento de métodos baseados em Redes Neurais Guiadas pela Física (PINNs, do inglês *Physics-Informed Neural Networks*), propostos por Raissi, Perdikaris e Karniadakis [35, 37], transformou a abordagem para a solução de EDPs. As PINNs integram as leis físicas diretamente no treinamento de redes neurais, evitando a necessidade de malhas discretas tradicionais e permitindo flexibilidade na resolução de problemas em espaços e dimensões complexos. Desde a proposta seminal, vários avanços foram alcançados, incluindo aplicações em problemas multiescala [30], dinâmica de fluidos [18], e otimização em engenharia [31].

Enquanto a maioria dos textos sobre PINNs está em inglês, este trabalho foi desenvolvido para oferecer um ponto de partida acessível, focando em uma explicação clara e prática para aqueles que desejam criar sua primeira rede neural voltada à solução de EDPs. Além disso, discutimos tanto a implementação manual, utilizando bibliotecas como TensorFlow, quanto a implementação automatizada, utilizando a biblioteca DeepXDE.

## Objetivo e Resultados

O objetivo principal deste trabalho é fornecer um guia introdutório para a implementação de PINNs para resolver EDPs. Como casos de referência, consideramos a equação de Burgers viscosa e a equação de Korteweg–de Vries (KdV), ambas amplamente reconhecidas por sua relevância em contextos físicos e matemáticos. Além disso, este trabalho tem como objetivo investigar a eficiência das PINNs na solução numérica de problemas de valor inicial a um parâmetro. Os resultados obtidos demonstram a eficiência das PINNs, destacando que soluções aproximadas com erro relativo da ordem de $10^{-4}$ podem ser alcançadas mediante a escolha adequada da configuração da rede neural, o que reforça o potencial desta abordagem como uma ferramenta robusta e precisa para problemas complexos envolvendo EDPs.

## Relevância do Tema de Estudo

A relevância deste trabalho reside em sua capacidade de preencher uma lacuna na literatura em português, oferecendo uma base sólida para estudantes de graduação e pós-graduação e profissionais que desejam adentrar o campo das PINNs. Além disso, a integração de implementações manuais e automatizadas serve como uma ponte para transitar entre os conceitos fundamentais e as ferramentas modernas disponíveis.

## Revisão Bibliográfica e Linha do Tempo

A evolução das Redes Neurais Guiadas pela Física (PINNs) representa uma mudança paradigmática na solução de equações diferenciais parciais (EDPs) ao unir técnicas de aprendizado de máquina com leis da física. O conceito foi introduzido por Raissi et al. [35], que mostraram como as redes neurais poderiam ser treinadas para resolver EDPs diretamente ao incluir as equações na função de perda. Essa abordagem foi expandida em 2019 por Raissi, Perdikaris e Karniadakis [37], explorando problemas de dinâmica de fluidos e transferência de calor, evidenciando as vantagens das PINNs em lidar com dados ruidosos e problemas inversos.

De acordo com Karniadakis et al. [22], as PINNs se destacam pela capacidade de integrar dados observacionais, leis físicas e incertezas em um único modelo. Além de resolver problemas bem-postos de forma eficiente, as PINNs mostram um desempenho robusto em problemas mal-postos e inversos, nos quais métodos tradicionais enfrentam grandes dificuldades. Isso é possível ao combinar diferentes tipos de viés: observacional, indutivo e de aprendizado, tornando a solução fisicamente consistente e mais interpretável.

Ferramentas como `DeepXDE` [30] facilitaram a implementação das PINNs, automatizando a construção e o treinamento dos modelos. Essa automação reduziu a barreira de entrada para novos pesquisadores e ampliou a aplicabilidade das PINNs em problemas multidimensionais e paramétricos, como aqueles encontrados na física computacional, ciências dos materiais e geofísica. Conforme destacado no trabalho de Karniadakis et al. [22], o uso de frameworks abertos e otimizados, aliados a técnicas como decomposição de domínio e regularização adaptativa, permitiu que as PINNs escalassem para problemas de maior complexidade e dimensões elevadas.

Em síntese, a trajetória das PINNs demonstra seu potencial para revolucionar o campo de soluções numéricas de EDPs, com aplicações que vão desde problemas diretos e inversos até a descoberta de física desconhecida. Essa evolução contínua é impulsionada por avanços teóricos, desenvolvimento de bibliotecas eficientes e integração com técnicas modernas de aprendizado profundo.

## Metodologia e Arquitetura do Trabalho

Este trabalho é estruturado para conduzir o leitor progressivamente. No Capítulo 1 discutimos a implementação manual de PINNs, abordando a equação de Burgers viscosa como estudo de caso. No Capítulo 2, exploramos a biblioteca DeepXDE, demonstrando como ela facilita a implementação de PINNs para equações mais complexas, como a equação de KdV. Por fim, no Capítulo 3 apresentamos uma análise comparativa entre os métodos, destacando vantagens, limitações e recomendações para futuras implementações. O trabalho é concluído com uma análise detalhada dos resultados obtidos, comparando as abordagens manual e automatizada. Demonstramos que ambas podem atingir alta precisão, mas que o uso de ferramentas como DeepXDE oferece vantagens significativas em termos de eficiência e simplicidade, especialmente para problemas que exigem cálculo de derivadas de ordem superior.

# Capítulo 1

# Implementação "Manual" de PINNs

O estudo apresentado neste capítulo foi desenvolvido com base nos trabalhos de Maziar Raissi et al. [35] e Jan Blandeschmit et al. [3], que exploram abordagens modernas para a solução de Equações Diferenciais Parciais (EDPs) por meio de Redes Neurais Guiadas pela Física (PINNs).

## 1.1 Equação de Burgers Viscosa

A Equação Diferencial Parcial (EDP) abordada neste estudo busca determinar $u = u(x,t)$, que satisfaz as seguintes condições:

$$\begin{aligned} u_t + uu_x - \nu u_{xx} = 0, \quad & (x,t) \in (-1,1) \times (0,1), \\ u(x,0) = u_0(x), \quad & x \in (-1,1), \\ u(-1,t) = u(1,t) = 0, \quad & t \in (0,1), \end{aligned} \tag{1.1}$$

onde $\nu$ representa a viscosidade do fluido. Essa equação, conhecida como **Equação de Burgers Viscosa**, foi inicialmente proposta como um modelo matemático para descrever fenômenos relacionados à turbulência no fluxo de fluidos [5, 23].

Além disso, devido à sua estrutura não linear e dissipativa, a Equação de Burgers tem sido amplamente utilizada como um caso de teste para técnicas numéricas e métodos computacionais em dinâmicas dos fluidos. LeVeque, por exemplo, realizou extensivos estudos utilizando a equação de Burgers para testar e comparar diferentes métodos numéricos, incluindo esquemas de diferenças finitas e volumes finitos, avaliando sua eficiência e precisão em contextos que envolvem choques e fenômenos dissipativos [28].

Adicionalmente, Raissi et.al. utilizaram a Equação de Burgers como um exemplo clássico para validar o desempenho de Redes Neurais Guiadas pela Física (PINNs), demonstrando a eficácia dessas redes na resolução de problemas de EDPs não lineares e dissipativos [35].

### 1.1.1 Modelagem Computacional

O tratamento numérico da Equação de Burgers Viscosa será realizado através de uma formulação computacional que transforma o problema de valor inicial e de fronteira em um problema de otimização. Essa abordagem permite que a solução aproximada seja obtida minimizando uma função de perda, que incorpora informações do operador diferencial, condições iniciais e de contorno.

Mais especificamente, buscamos determinar uma função aproximada $u_N(x,t)$ que resolve simultaneamente os seguintes problemas de otimização:

**Problema 1.** Operador Diferencial:

$$\min |\mathcal{D}(x,t)| \quad \text{sujeito a} \quad (x,t) \in (-1,1) \times (0,1),$$

onde

$$\mathcal{D} := u_{N_t} + u_N u_{N_x} - \nu u_{N_{xx}}.$$

**Problema 2.** Condição Inicial:

$$\min |u_N(x,0) - u_0(x)| \quad \text{sujeito a} \quad x \in (-1,1).$$

**Problema 3.** Condição de Contorno:

$$\min |u_N(-1,t)| + |u_N(1,t)| \quad \text{sujeito a} \quad t \in (0,1).$$

A função $u_N(x,t)$ será aproximada via Redes Neurais por meio da minimização do Erro Médio Quadrático (EMQ) associado a uma função de perda conforme definido em estudos prévios [3, 12, 35].

**Observação 1.1.** *Uma pergunta natural que surge é: "Por que é de se esperar que exista uma Rede Neural $u_N$ que aproxima a solução da EDP?"*

*A resposta desse problema é dado pelo Teorema da Aproximação Universal, que em uma de suas versões nos diz que:*

**Teorema 1.2** (Teorema da Aproximação Universal [14] p.48)**.** *Seja $\sigma : \mathbb{R} \to \mathbb{R}$ uma função contínua discriminatória qualquer. Então, dada qualquer $f : \Omega \subset \mathbb{R}^n \to \mathbb{R}$, onde $\Omega$ é compacto, contínua, e $\varepsilon > 0$, existem vetores $\mathbf{w}_1, \ldots, \mathbf{w}_N, \mathbf{w}_s, \mathbf{b}$ e uma função $F : \Omega \to \mathbb{R}$ tais que*

$$|F(x) - f(x)| < \varepsilon, \quad \forall x \in \Omega,$$

*onde* $F(x) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j^T x + \mathbf{b}_j)$

*Na notação do Teorema, $F$ representa uma rede neural de uma única camada, que aproxima a função $f$. Além disso:*

- $\sigma : \mathbb{R} \to \mathbb{R}$ *é uma função não linear (chamada de função de ativação)*
- $\mathbf{b} = (b_1, \ldots, b_N)$ *é o vetor de viesis/bias*

- $\mathbf{w}_j = (w_{1,j}, \ldots, w_{n,j})$ *é o vetor de pesos do j-ésimo neurônio*
- $\mathbf{w}_s = (\alpha_1, \ldots, \alpha_N)$ *é o vetor de pesos do neurônio de saída*

*Para o caso de redes neurais multicamadas fazemos uma série composições de funções afim com funções de ativação, a saber,*

$$F(\mathbf{x}) = W^L \sigma^L (W^{L-1} \sigma^{L-1} (\cdots \sigma^1 (W^0 \mathbf{x} + \mathbf{b}^0) \cdots) + \mathbf{b}^{L-1}) + \mathbf{b}^L,$$

*onde* $W^L$ *e* $\mathbf{b}^L$ *são matrizes de pesos e vetores de vieses.*

*Voltando para a pergunta "Por que é de se esperar que exista uma Rede Neural* $u_N$ *que aproxima a solução da EDP?", a resposta é a seguinte: a solução da EDP é pelo menos contínua, daí segue do Teorema 1.2 que existe uma função que a aproxima. Um leitor interessado em mais detalhes sobre PINNs pode consultar a referência [4].*

Neste contexto, os Problemas 1, 2 e 3 podem ser reformulados da seguinte maneira:

**Problema 1.** Operador diferencial:

$$\text{EMQ}_{\mathcal{D}} = \frac{1}{N_{\mathcal{D}}} \sum_{i=1}^{N_{\mathcal{D}}} |\mathcal{D}(x_i, t_i)|^2,$$

onde $(x_i, t_i), i = 1, \ldots, N_{\mathcal{D}}$, são $N_{\mathcal{D}}$ pontos de colocação escolhidos aleatoriamente em $(-1, 1) \times (0, 1)$.

**Problema 2.** Condição inicial:

$$\text{EMQ}_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} |u_N(x_i, 0) - u_0(x_i)|^2,$$

onde $x_i, i = 1, \ldots, N_0$, representam $N_0$ pontos de colocação no intervalo $(-1, 1)$.

**Problema 3.** Condição de contorno:

$$\text{EMQ}_b = \frac{1}{N_b} \sum_{i=1}^{N_b} |u_N(x_i, t_i)|^2,$$

onde $(x_i, t_i), i = 1, \ldots, N_b$, correspondem a $N_b$ pontos de colocação ao longo do conjunto cartesiano $\{-1, 1\} \times (0, 1)$.

A função de perda geral é então definida como:

$$\text{EMQ} = \text{EMQ}_{\mathcal{D}} + \text{EMQ}_0 + \text{EMQ}_b \tag{1.2}$$

A minimização desta função de perda é fundamental para que a rede neural seja capaz de aproximar corretamente a solução da equação diferencial. A seguir, será apresentada a implementação numérica desta abordagem.

### 1.1.2 Implementação Numérica

A implementação, baseada em [3, 35], faz uso das bibliotecas **NumPy** [13] e **TensorFlow** [1] no ambiente Python. Para garantir clareza e fluidez, a implementação será apresentada em passos organizados.[1]

**Passo 1. Importação das bibliotecas necessárias:**

As bibliotecas **NumPy** e **TensorFlow** constituem a base do funcionamento da implementação, sendo importadas conforme ilustrado no Código 1.1. Além disso, outras bibliotecas auxiliares também são importadas para atender às demandas específicas de cada etapa da implementação.

Código 1.1: Importação das bibliotecas necessárias

```
1  import numpy as np  # Biblioteca NumPy clássica do Python
2  from numpy import linalg as LA  # Funções de Álgebra Linear no NumPy
3  import matplotlib.pyplot as plt  # Biblioteca para visualização e criação de gráficos
4  import plotly.graph_objects as go  # Biblioteca para visualização de gráficos 3D da EDP
5  from mpl_toolkits.mplot3d import Axes3D  # Biblioteca para visualização de gráficos 3D da EDP
6  import warnings
7  warnings.filterwarnings('ignore')  # Ignorar avisos para diminuir a poluição visual
8  import tensorflow as tf  # Biblioteca TensorFlow para criação da rede neural
9  from time import time  # Função para cálculo do tempo computacional, equivalente ao tic toc
10 import scipy.io  # Para importar e exportar arquivos .mat
11
12 # Define a semente aleatória para reproduzir os resultados
13 import os
14 import random
15 seed_value = 0
16 os.environ['PYTHONHASHSEED'] = str(seed_value)
17 random.seed(seed_value)
18 np.random.seed(seed_value)  # Caso o NumPy não esteja fixado, a função de Bernoulli retornará diferentes
   ↪ valores
19 tf.random.set_seed(seed_value)  # Fixa os dados amostrados
20 print(tf.__version__)  # Impressão da versão do TensorFlow utilizado (código criado na versão 2.X)
```

No Código 1.1, entre as linhas 12 e 19, encontram-se as funções responsáveis pela fixação da semente aleatória.

**Passo 2. Definição dos parâmetros iniciais:**

Para o problema definido em (1.1), utilizamos $u_0(x) = -\operatorname{sen}(\pi x)$ e $\nu = \dfrac{0.01}{\pi}$, conforme sugerido por Raissi et.al. [35]. Com base nesses valores, o Código 1.2 foi desenvolvido para definir os parâmetros iniciais do problema.

Código 1.2: Definição dos parâmetros iniciais

[1] A implementação computacional apresentada foi realizada utilizando o ambiente Google Colab.

```python
# Definir o tipo de dado do TensorFlow, garantindo que todos os cálculos sejam realizados usando números de ponto flutuante de 32 bits.
DTYPE = 'float32'  # float64 também é uma opção
tf.keras.backend.set_floatx(DTYPE)

# Definição da viscosidade
pi = tf.constant(np.pi, dtype=DTYPE)
viscosity = 0.01 / pi

# Definição da condição inicial (x precisa ser um tensor)
def u_0(x):
    return -tf.sin(pi * x)

def u_b(x, t):
    n = x.shape[0]  # Retorna o número de linhas do vetor/tensor x
    return tf.zeros((n, 1), dtype=DTYPE)  # Retorna um tensor de zeros do tamanho (n, 1)

# Definição da função f(x,t) (operador diferencial)
def f(u, u_x, u_t, u_xx):
    return u_t + u * u_x - viscosity * u_xx
```

**Passo 3. Geração de Pontos de Colocação**

Com os parâmetros definidos, o próximo passo consiste em gerar um conjunto de pontos de colocação, os quais serão utilizados pela rede para o treinamento.

Neste trabalho, esses pontos são gerados por meio de amostragem aleatória a partir de uma distribuição uniforme. Essa estratégia demonstrou-se satisfatória em todos os experimentos realizados. No entanto, destacamos que outras abordagens podem ser adotadas. Por exemplo, Raissi et al. [35] aplicaram a técnica de amostragem de hipercubo latino para preenchimento de espaço, conforme descrito por Stein [40].

Código 1.3: Geração de Pontos de Colocação

```python
# Definição do tamanho da grade dos pontos de colocação
N_0 = 50  # quantidade de pontos na condição inicial
N_b = 50  # quantidade de pontos na condição de contorno
N_f = 10000  # quantidade de pontos para o operador diferencial

# Limites do domínio / contorno
xmin = -1.0  # espaço
xmax = 1.0  # espaço
tmin = 0.0  # tempo
tmax = 1.0  # tempo
```

```
11
12 # limite inferior / Lower bounds
13 lb = tf.constant([xmin, tmin], dtype=DTYPE)
14 # limite superior / Upper bounds
15 ub = tf.constant([xmax, tmax], dtype=DTYPE)
16
17 # pontos de amostra uniformemente distribuídos
18 x_0 = tf.random.uniform((N_0,1), lb[0], ub[0], dtype=DTYPE)  # vetor de tamanho (N_0, 1), valores
   ↪  distribuidos entre xmin e xmax
19 t_0 = tf.ones((N_0,1), dtype=DTYPE)*lb[1]  # vetor de tamanho (N_0, 1), com valor de tmin
20 XT_0 = tf.concat([x_0, t_0], axis=1)  # concatena x_0 e t_0, matriz de tamanho (N_0, 2), variáveis
   ↪  espaciais e temporais
21
22 # avalia x_0 as condições iniciais: u_0(x) = -tf.sin(pi * x).
23 u_0 = u_0(x_0)
24
25 # condições de contorno: dados uniformemente distribuidos
26 x_b = lb[0] + (ub[0] - lb[0]) * tf.keras.backend.random_bernoulli((N_b,1), 0.5, dtype=DTYPE)
27 '''
28 x_b é um vetor de tamanho (N_b, 1), com valores uniformemente distribuídos sendo lb[0] ou ub[0].
29 A distribuição de Bernoulli retorna 0 ou 1, a escolha do parâmetro 0.5 significa que
30 temos 50/50 para 0 e 1.
31 '''
32 t_b = tf.random.uniform((N_b,1), lb[1], ub[1], dtype=DTYPE)  # vetor de tamanho (N_b, 1), valores
   ↪  distribuidos entre tmin e tmax
33
34 XT_b = tf.concat([x_b, t_b], axis=1)  # concatena x_b e t_b, tamanho (N_b, 2), variáveis espaciais e
   ↪  temporais
35
36 # avalia (x_b, t_b) na condição de contorno: u_b
37 u_b = u_b(x_b, t_b)
38
39 # gera pontos de colocação uniformemente distribuídos:
40 x_f = tf.random.uniform((N_f,1), lb[0], ub[0], dtype=DTYPE)  # vetor de tamanho (N_f, 1), com valores
   ↪  uniformemente distribuídos entre xmin e xmax.
41 t_f = tf.random.uniform((N_f,1), lb[1], ub[1], dtype=DTYPE)  # vetor de tamanho (N_f, 1), com valores
   ↪  uniformemente distribuídos entre tmin e tmax.
42 XT_f = tf.concat([x_f, t_f], axis=1)  # concatena x_f e t_f, criando uma grade aleatória no plano
   ↪  espaço-tempo
43
44 # Coletando os dados inicial e de fronteira em uma lista
45 XT_data = [XT_0, XT_b]
46 u_data = [u_0, u_b]
```

**Passo 4. Visualização dos Pontos de Colocação:**

No Código 1.4, ilustramos os pontos de colocação (representados por círculos vermelhos) juntamente com as posições em que as condições iniciais ($X$ preto) e de contorno ($X$ azul) serão aplicadas. A visualização resultante é apresentada na Figura 1.1.

Código 1.4: Visualização dos Pontos de Colocação

```
fig = plt.figure(figsize=(9,6))
plt.scatter(x_0, t_0, c='black', marker='X', vmin=-1, vmax=1)
plt.scatter(x_b, t_b, c='blue', marker='X', vmin=-1, vmax=1)
plt.scatter(x_f, t_f, c='r', marker='.', alpha=0.1)
plt.xlabel('$x$')
plt.ylabel('$t$')
#plt.title('Posição dos pontos de colocação (vermelho), dos dados iniciais (preto) e de fronteira (azul)');
#plt.savefig('Xdata_Burgers.pdf', bbox_inches='tight', dpi=300)
```

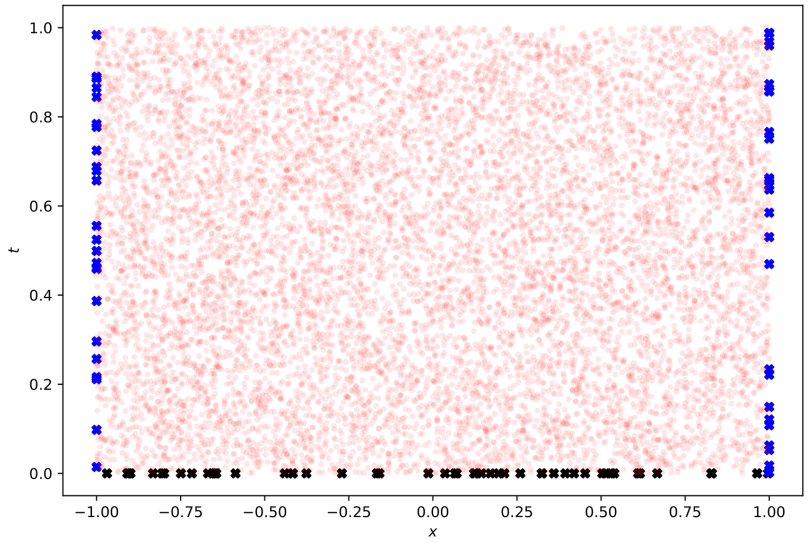



Figura 1.1: Posição dos pontos de colocação amostrados gerado pelos códigos 1.3 e 1.4

**Passo 5. Criação e Configuração da Rede Neural:**

A criação e configuração da Rede Neural são realizadas no Código 1.5.

Código 1.5: Criação e Configuração da Rede Neural

```
def init_model(num_hidden_layers=9, num_neurons_per_layer=40):  # número de camadas ocultas = 9 / número de neuronios por camada = 40
    # Inicializa uma rede neural feedforward
    model = tf.keras.Sequential()

    # A entrada é bidimensional (uma dimensão espacial + tempo)
```

```
6     model.add(tf.keras.Input(2))
7
8     # Introduz uma camada de dimensionamento para mapear a entrada para [lb, ub] = [[-1,0], [1,1]] --
      ↪  ajuste de escala para [-1,1] x [-1,1]
9     scaling_layer = tf.keras.layers.Lambda(
10                lambda x: 2.0*(x - lb)/(ub - lb) - 1.0)
11    model.add(scaling_layer)
12
13    # Anexa camadas ocultas
14    for _ in range(num_hidden_layers):  # _ é usado para representar uma variável que não é usada dentro
      ↪  do loop, serve apenas para iterar
15        model.add(tf.keras.layers.Dense(num_neurons_per_layer,
16            activation=tf.keras.activations.get('tanh'),
17            kernel_initializer='glorot_normal'))
18
19    # A saída é unidimensional
20    model.add(tf.keras.layers.Dense(1))
21
22    return model
```

Primeiramente, é introduzida uma camada de dimensionamento, cujo objetivo é ajustar os valores de entrada para uma faixa adequada ao funcionamento da rede. Redes neurais geralmente apresentam melhor desempenho quando os valores de entrada estão próximos de 0, pois isso auxilia no controle da propagação de gradientes durante o treinamento [3, 35]. Além disso, algoritmos de otimização convergem frequentemente de forma mais eficaz quando os dados estão nessa faixa.

No presente trabalho, os dados são informados ao longo do retângulo $[-1, 1] \times [0, 1]$, transformado no quadrado $[-1, 1] \times [-1, 1]$ utilizando a seguinte fórmula:

$$\text{scaled value} = 2 \cdot \frac{(x, t) - lb}{ub - lb} - 1$$

Também adotamos a função de ativação tangente hiperbólica (tanh), uma escolha popular em redes neurais, conforme indicado na linha 16 do Código 1.5. De acordo com Karlik et.al. [21], a tangente hiperbólica apresenta alta precisão e é amplamente aplicada em redes multicamadas, oferecendo bons resultados na maioria dos casos. Ressaltamos que o objetivo deste trabalho não é identificar a melhor função de ativação, mas demonstrar que, com essa escolha, o problema pode ser resolvido de forma eficiente.

Adicionalmente, utilizamos o inicializador de peso *Glorot Normal*, descrito por Glorot et.al. [9], devido à sua ampla aceitação na literatura. Esse inicializador evita problemas de gradientes desaparecendo ou explodindo em redes profundas, promovendo um treinamento mais estável e eficiente. Ele equilibra a variância dos sinais ao longo das camadas, facilitando a propagação adequada de ativações e gradientes,

o que melhora a convergência. Projetado especialmente para funções de ativação como tanh, o *Glorot Normal* otimiza o desempenho inicial das redes. Contudo, reiteramos que o objetivo principal não foi identificar o melhor inicializador, mas evidenciar que, com um inicializador bem estabelecido, o método é funcional.

**Passo 6. Definição do Resíduo da EDP:**

Com a Rede Neural criada, o Código 1.6 define a função responsável por avaliar o resíduo $\mathcal{D} := u_t + uu_x - \nu u_{xx}$ do Problema 1, uma equação diferencial parcial (EDP) não linear. Essa avaliação é realizada nos pontos $(x_i, t_i) \in (-1, 1) \times (0, 1)$, com $i = 1, \ldots, N_{\mathcal{D}}$, que foram previamente escalados para o quadrado $[-1, 1] \times [-1, 1]$.

Código 1.6: Configuração das rotinas para derivação automática (*automatic differentiation*)

```
def get_r(model, XT_f):  # recebe um modelo e uma base de dados
    # Utiliza o tf.GradientTape para calcular derivadas no TensorFlow
    with tf.GradientTape(persistent=True) as tape:  # persistent=True permite fazer as segundas derivadas
    ↪  (reutiliza o mesmo GradientTape)
        # Divide t e x para calcular derivadas parciais
        x, t = XT_f[:, 0:1], XT_f[:, 1:2]  # x é a primeira coluna de XT_f e t é a segunda coluna

        # Variáveis x e t são "marcadas" durante tape (para que o GradientTape acompanhe e calcule as
        ↪  derivadas em relação a essas variáveis)
        # para calcular as derivadas u_t e u_x
        tape.watch(x)
        tape.watch(t)

        # Determina a u
        u = model(tf.stack([x[:, 0], t[:, 0]], axis=1))  # utilizando o modelo e a u para calcular as
        ↪  derivadas

        # Calcula a derivada u_x dentro do tf.GradientTape pois precisamos de segundas derivadas
        u_x = tape.gradient(u, x)

    u_t = tape.gradient(u, t)
    u_xx = tape.gradient(u_x, x)

    # Libera o tape persistente para economizar recursos
    del tape

    # Retorna o resíduo calculado usando a função f fornecida
    return f(u, u_x, u_t, u_xx)
```

Para calcular as derivadas parciais, utilizamos a técnica de diferenciação automática fornecida pelo TensorFlow. No caso da equação de Burgers, isso implica o cálculo de $\partial_t u_N$, $\partial_x u_N$ e $\partial_{xx} u_N$. Aqui, $u_N$

representa a função aproximada pela Rede Neural, parametrizada pelos pesos e vieses. Esses parâmetros são ajustados durante o treinamento para minimizar o resíduo da EDP e satisfazer as condições iniciais e de contorno. Esse processo é realizado por meio da ferramenta `GradientTape`, que rastreia as variáveis monitoradas ("watched variables"), em nosso caso $t$ e $x$, permitindo a computação automática das derivadas necessárias [1].

**Passo 7. Cálculo da Função de Perda:**

A função apresentada no Código 1.7 é responsável pelo cálculo da função de perda do modelo, conforme definido em (1.2). Essa função, que será minimizada durante o treinamento, é dada por:

$$\mathrm{EMQ} = \mathrm{EMQ}_{\mathcal{D}} + \mathrm{EMQ}_0 + \mathrm{EMQ}_b,$$

onde os termos estão definidos nos Problemas 1, 2 e 3. Em particular, o operador do Problema 1 foi implementado no Código 1.6.

Código 1.7: Cálculo da Função de Perda

```python
def compute_loss(model, XT_f, XT_data, u_data):
    # Calcula EMQ_f
    f = get_r(model, XT_f)
    EMQ_f = tf.reduce_mean(tf.square(f))

    # Inicializa a perda
    loss = EMQ_f

    # Adiciona EMQ_0 e EMQ_b à perda
    for i in range(len(XT_data)):
        u_pred = model(XT_data[i])
        loss += tf.reduce_mean(tf.square(u_data[i] - u_pred))

    return loss
```

**Passo 8. Cálculo do Gradiente da Função de Perda:**

Em seguida, o gradiente da função de perda é calculado no Código 1.8, para posterior utilização no processo de otimização pelo método do gradiente descendente.

Código 1.8: Cálculo do Gradiente da Função de Perda

```python
def get_grad(model, XT_f, XT_data, u_data):
    with tf.GradientTape(persistent=True) as tape:
        # Esta tape é para derivadas em relação às variáveis treináveis do modelo
        tape.watch(model.trainable_variables)
        loss = compute_loss(model, XT_f, XT_data, u_data)
```

```
6
7     g = tape.gradient(loss, model.trainable_variables)
8     del tape
9
10    return loss, g
```

**Passo 9. Escolha da Taxa de Aprendizado:**

Com as funções definidas, seguimos a abordagem proposta por Raissi et.al. [35] e Blechschmidt et.al. [3], adotando uma taxa de aprendizado que decai por partes. Essa estratégia utiliza uma função degrau aplicada ao algoritmo do tipo gradiente descendente. Nos primeiros 1000 passos, utilizamos uma taxa de aprendizado de 0,01; de 1000 a 3000 passos, a taxa foi reduzida para 0,001; e, a partir de 3000 passos, aplicamos uma taxa de 0,0005. Essa configuração foi implementada conforme ilustrado no Código 1.9.

Código 1.9: Escolha da Taxa de Aprendizado

```
1 lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay(
2     [1000, 3000],
3     [1e-2, 1e-3, 5e-4])
```

**Passo 10. Configuração Inicial da Rede Neural:**

Também realizamos a configuração inicial da rede, incluindo a definição do número de camadas, neurônios por camada e a escolha do otimizador. Essa etapa foi implementada conforme descrito no Código 1.10.

Código 1.10: Parâmetros da rede e otimizador

```
1 # Inicialização do modelo
2 model = init_model(num_hidden_layers=9, num_neurons_per_layer=40)
3
4 # Escolha do otimizador
5 optim = tf.keras.optimizers.Adam(learning_rate=lr)
6
7 # Exibe um resumo da rede com informações sobre os parâmetros. None indica que o modelo pode aceitar
  ↪  lotes de exemplos de qualquer tamanho durante o treinamento e inferência
8 model.summary()
```

Mais detalhes sobre a escolha das camadas e neurônios definidos no Código 1.10 serão discutidos na Seção 1.2.

Além da definição de camadas e neurônios, é fundamental considerar outras escolhas importantes, como o inicializador de pesos (neste caso, *Glorot Normal*) e a função de ativação (tanh), já descritos

anteriormente. Outro elemento essencial é a escolha do otimizador da Rede Neural, o qual é responsável pelo ajuste dos pesos e a estabilidade do treinamento.

Optamos pelo Adam (*Adaptive Moment Estimation*) [24], que é amplamente utilizado devido à sua eficiência em problemas de aprendizado profundo. O Adam combina os benefícios de dois métodos consolidados: o *Momentum*, que acelera o treinamento ao suavizar as atualizações dos parâmetros, e o RMSProp, que ajusta dinamicamente o tamanho do passo com base na magnitude dos gradientes. O Adam oferece convergência rápida e estável, sendo uma escolha confiável para otimização em redes neurais.

**Passo 11. Configuração do Treinamento da Rede Neural:**

A configuração do processo de treinamento da rede foi implementada no Código 1.11.

Código 1.11: Configuração do Treinamento da Rede Neural

```
# Definição da função de um passo de treinamento como uma função do tf para aumentar a velocidade de treino
@tf.function  # Adicionado no TensorFlow 2.X, melhora significativamente o desempenho e tempo computacional
def train_step():
    loss, grad_theta = get_grad(model, XT_f, XT_data, u_data)

    # Realiza um passo do gradiente descendente
    optim.apply_gradients(zip(grad_theta, model.trainable_variables))

    return loss

# Número de épocas
N = 1000
hist = []   # Lista para armazenar os valores de perda durante o treinamento

# Tempo inicial
t0 = time()

for i in range(N+1):
    loss = train_step()
    hist.append(loss.numpy())

    # Exibe a perda a cada 100 iterações
    if i % 100 == 0:
        print('It {:05d}: loss = {:10.8e}'.format(i, loss))

# Exibe o tempo computacional
print('\nTempo computacional: {} segundos'.format(time() - t0))
```

A função `train_step`, definida no Código 1.11, realiza um passo do gradiente descendente utilizando todas as implementações descritas nos Códigos 1.1 a 1.10. Esse processo consiste em calcular a função de perda, determinar os gradientes correspondentes por meio de diferenciação automática e atualizar os pesos da rede neural utilizando o otimizador definido (neste caso, Adam).

A variável `N` representa o número de iterações ou épocas do treinamento. Cada época corresponde a uma passada completa pelo conjunto de dados de treino, durante a qual a rede neural ajusta seus parâmetros com base nos gradientes calculados. O valor de `N` determina, portanto, quantas vezes o modelo realizará esse processo de aprendizado iterativo. No treinamento de redes neurais, um número adequado de épocas é crucial para garantir a convergência da função de perda.

No caso desta implementação, foram utilizadas $N = 1000$ iterações, com o valor da função de perda sendo exibido a cada 100 passos para monitorar o progresso do treinamento. O histórico das perdas é armazenado na lista `hist`, permitindo a visualização do comportamento da função de perda ao longo das iterações.

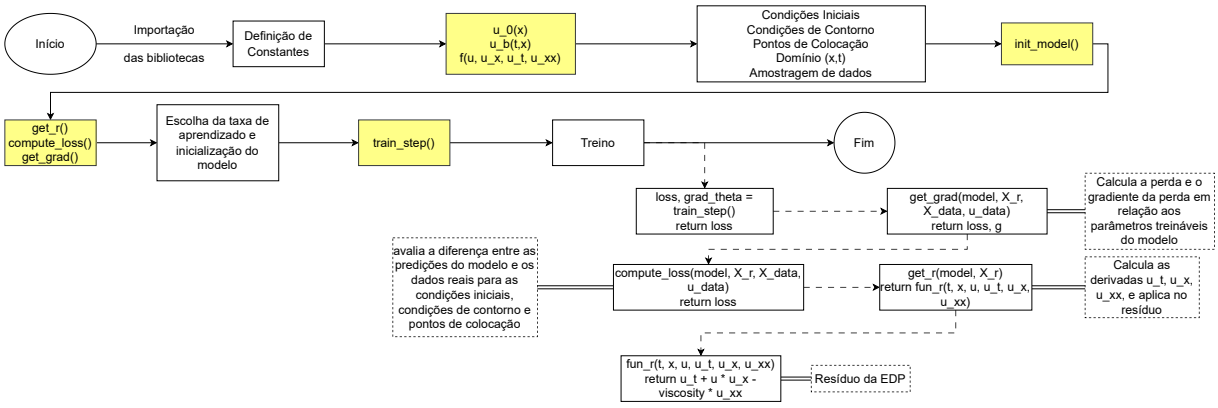



Figura 1.2: Fluxo do código

## 1.2 Escolha de Parâmetros da Rede

A seção anterior apresentou toda a implementação numérica para a criação de uma PINN. No entanto, alguns parâmetros são essenciais em etapas específicas dessa implementação. Por exemplo, no Código 1.3, é necessário selecionar os pontos de colocação para o treinamento da rede. Da mesma forma, no Código 1.5, escolhemos a função de ativação e o inicializador de pesos. Por fim, no Código 1.10, definimos a quantidade de camadas, neurônios por camada e o otimizador.

Neste trabalho, focamos em selecionar a melhor configuração para o número de camadas, quantidade de neurônios e quantidade de pontos de colocação. Para as demais escolhas, como função de ativação, inicializador de pesos e otimizador, optamos por abordagens bem estabelecidas na literatura.

### 1.2.1 Escolha da Quantidade de Camadas e Neurônios

Para determinar a melhor configuração da rede para a equação de Burgers Viscosa, realizamos testes iniciais utilizando $N_0 = 50$, $N_b = 50$ e $N_{\mathcal{D}} = 10000$, onde $N_{\mathcal{D}}$ representa pontos uniformemente distribuídos no intervalo definido por `t_min`, `t_max`, `x_min` e `x_max`. Para garantir a reprodutibilidade dos experimentos, fixamos uma semente aleatória para a geração desses pontos.

A taxa de aprendizado foi configurada para decair por partes, utilizando a função degrau do algoritmo do tipo gradiente descendente. Nos primeiros 1000 passos, foi adotada uma taxa de aprendizado de 0,01; de 1000 a 3000 passos, a taxa foi reduzida para 0,001; e, a partir de 3000 passos, utilizamos uma taxa de 0,0005. Essa configuração foi mantida para diversas combinações de (neurônios x camadas) a fim de identificar a melhor configuração para o problema em questão.

Os parâmetros utilizados ao longo dos treinamentos estão descritos na Tabela 1.1.

Tabela 1.1: Parâmetros utilizados

| **Parâmetro** | **Valor** |
|---|---|
| Otimizador | Adam |
| Função de ativação | tanh |
| Inicializador de pesos | Glorot Normal |
| Passos de treinamento | 5 000 |
| Pontos de colocação no interior ($N_{\mathcal{D}}$) | 10 000 |
| Pontos de colocação na borda ($N_b$) | 50 |
| Pontos de colocação na condição inicial ($N_0$) | 50 |

O critério adotado para a escolha da quantidade de camadas e neurônios nesta etapa foi baseado no valor da função de perda. Os resultados dessa análise estão representados no gráfico de calor da Figura 1.3.

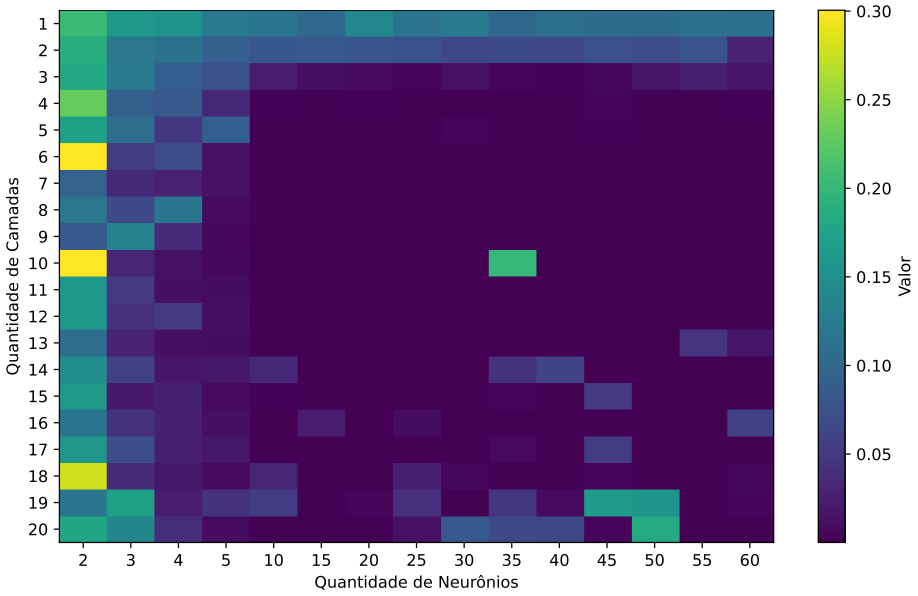



Figura 1.3: Perda mínima encontrada no treinamento de (neurônio x camada)

Encontramos uma região "ótima" entre as duplas (10, 4) e (60, 12), com vários valores da função de perda (1.2) alcançando a escala de $10^{-03}$.

No processo de verificação inicial da rede, alguns pontos chamam a atenção e são destacados nas Figuras 1.4 e 1.5:

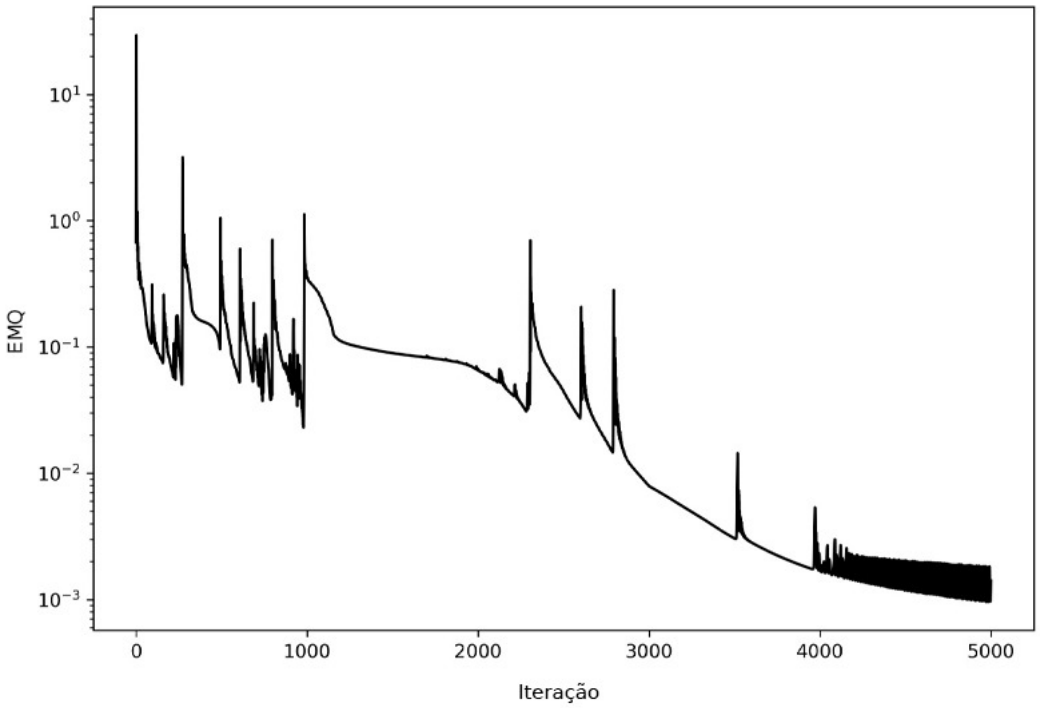



Figura 1.4: Treinamento de 9 Camadas e 50 Neurônios - Problema de Oscilação no treinamento

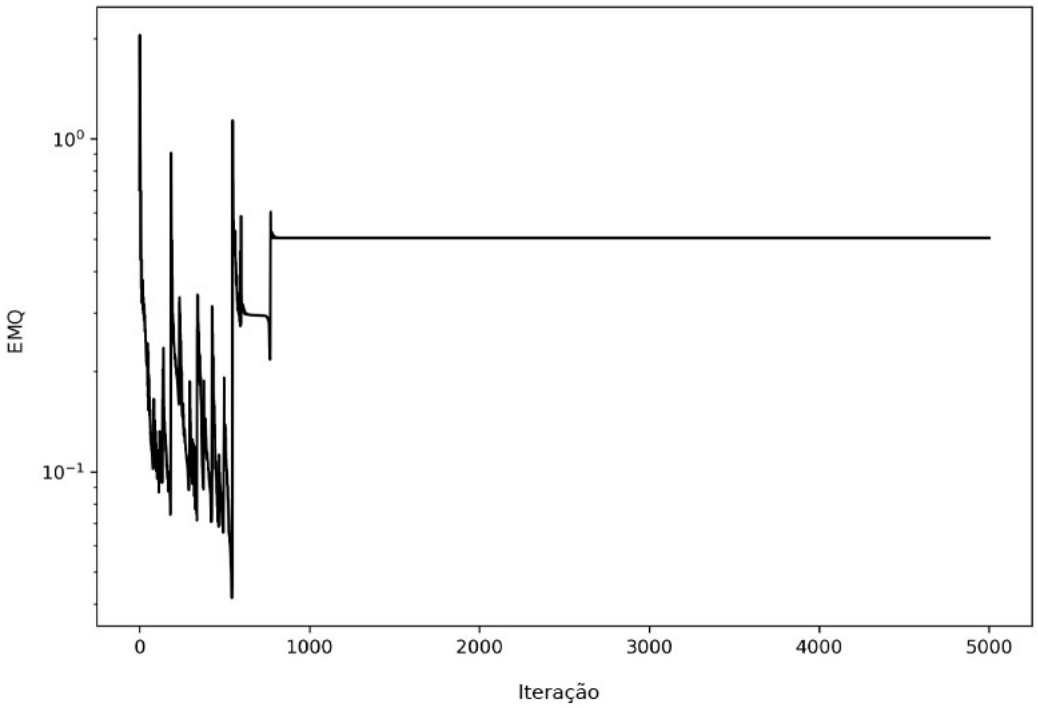



Figura 1.5: Treinamento de 19 Camadas e 25 Neurônios - Problema de não convergência da rede

Os problemas de oscilação, ilustrados na Figura 1.4, e de não convergência, mostrados na Figura 1.5, podem ser mitigados ajustando a taxa de aprendizado. No entanto, dependendo da configuração de camadas e neurônios, essa solução pode não ser viável, considerando o tempo e o esforço computacional necessários. Nesse contexto, explorar outras configurações que ofereçam melhores resultados com menor custo computacional pode ser uma abordagem mais eficiente.

É importante destacar que, em todas as simulações apresentadas até o momento, os parâmetros do otimizador (Adam) e do inicializador de pesos (*Glorot Normal*) permaneceram fixos. Na Figura 1.6, apresentamos o menor valor da função de perda obtido ao longo de 21 treinamentos, desta vez variando os parâmetros do inicializador de pesos e do otimizador. Contudo, os pontos de colocação foram mantidos constantes em todos os treinamentos. Esse estudo permite avaliar o impacto dos inicializadores na qualidade da rede treinada.

Abaixo, apresentamos a comparação das redes que alcançaram os melhores resultados, considerando a região "ótima" identificada na Figura 1.3:

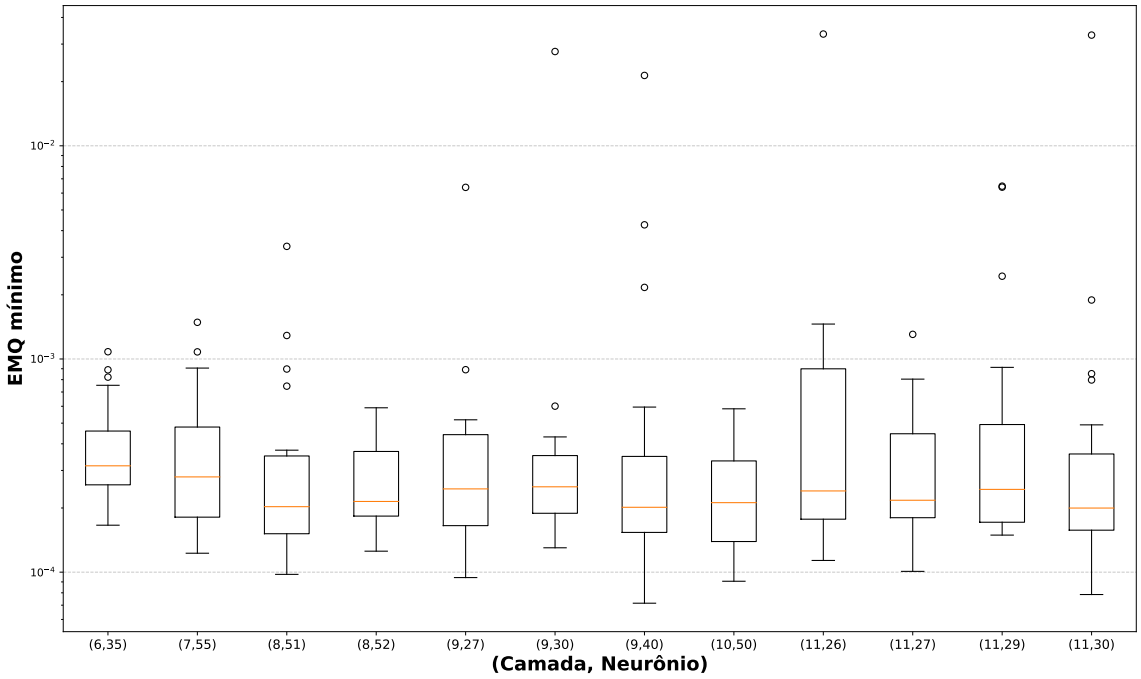



Figura 1.6: Comparação do menor EMQ encontrado em 21 treinamentos com otimizador (Adam) e Inicializador de Pesos (Glorot Normal) não fixos

A configuração escolhida para encontrar a solução foi a combinação (Camada, Neurônio) = (9, 40). Embora o gráfico boxplot da Figura 1.6 apresente alguns *outliers*, seus quantis, valores mínimos e máximos demonstraram resultados satisfatórios em comparação com as demais configurações testadas. Assim, sabemos que, com essa configuração, a rede estará na faixa de EMQ entre $10^{-03}$ e $10^{-05}$.

Entretanto, ao ajustar a taxa de aprendizado, as condições iniciais, a amostragem dos dados e ao fixar os parâmetros do otimizador e do inicializador de pesos, é possível obter resultados ainda melhores e mais

consistentes.

### 1.2.2 Quantidade de Pontos de Colocação

Para a escolha da quantidade de pontos de colocação, utilizamos como parâmetro o erro relativo em norma 2 de Frobenius para matrizes, baseado em uma solução de referência disponibilizada por Raissi et al. [35].

A configuração adotada foi de 9 camadas e 40 neurônios, com uma taxa de aprendizado que decai por partes. Desta vez, acrescentamos "degraus" com suas respectivas taxas de aprendizado, como definido na Tabela 1.2:

Tabela 1.2: Taxas de Aprendizado durante o Treinamento

| **Iteração** | **Taxa de Aprendizado** |
|---|---|
| 0 - 100 | $9 \cdot 10^{-3}$ |
| 101 - 200 | $7 \cdot 10^{-3}$ |
| 201 - 500 | $5 \cdot 10^{-3}$ |
| 501 - 1000 | $3 \cdot 10^{-3}$ |
| 1001 - 2000 | $1 \cdot 10^{-3}$ |
| 2001 - 4000 | $5 \cdot 10^{-4}$ |
| 4001 - 5000 | $1 \cdot 10^{-4}$ |
| 5001 - 7000 | $5 \cdot 10^{-5}$ |
| 7001 - 10000 | $1 \cdot 10^{-5}$ |
| 10001 - 15000 | $5 \cdot 10^{-6}$ |
| 15001 - 25000 | $1 \cdot 10^{-6}$ |
| 25001 - 35000 | $5 \cdot 10^{-7}$ |
| 35001 - 60000 | $1 \cdot 10^{-7}$ |

A configuração de taxa de aprendizado descrita demonstrou ser a mais eficiente para os experimentos realizados. Essa abordagem proporcionou uma redução linear e estável do erro relativo, permitindo à rede neural manter um bom desempenho em termos de EMQ, como apresentado na Figura 1.7. Além disso, o número de iterações utilizado, 60000, foi suficiente para garantir a convergência do modelo, mostrando-se adequado para a configuração de 9 camadas e 40 neurônios, bem como para a escolha dos pontos de amostra.

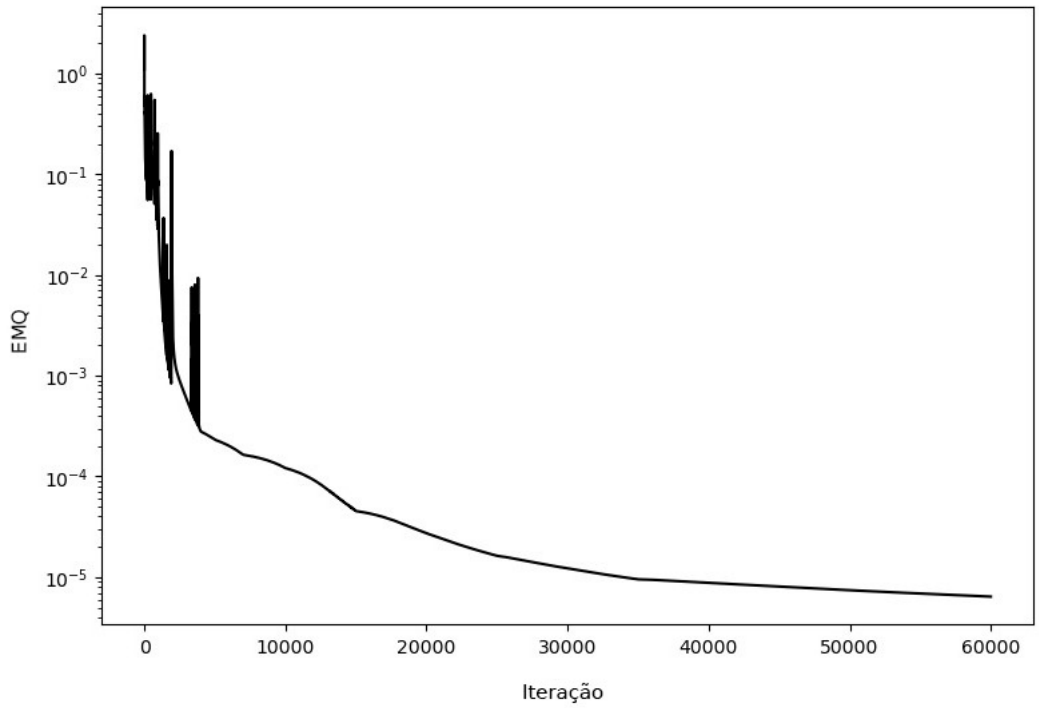



Figura 1.7: Treinamento de 9 Camadas e 40 Neurônios - Taxa de aprendizado conforme Tabela 1.2

A tabela 1.3 apresenta a avaliação de erros de modelos configurados com diferentes valores para os parâmetros $N_{0,b}$ (números de pontos de fronteira e condições iniciais) e $N_{\mathcal{D}}$ (número de pontos no domínio).

Tabela 1.3: Tabela de Perda e Erros do Modelo

| $N_{\mathcal{D}}$ / $N_{0,b}$ | 2000 | | 4000 | | 6000 | | 8000 | | 10000 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | EMQ | Erro Relativo | EMQ | Erro Relativo | EMQ | Erro Relativo | EMQ | Erro Relativo | EMQ | Erro Relativo |
| 20 | $2.26 \cdot 10^{-05}$ | $7.90 \cdot 10^{-02}$ | $1.01 \cdot 10^{-05}$ | $2.57 \cdot 10^{-03}$ | $5.58 \cdot 10^{-06}$ | $1.14 \cdot 10^{-03}$ | $3.98 \cdot 10^{-06}$ | $1.58 \cdot 10^{-03}$ | $9.08 \cdot 10^{-06}$ | $2.01 \cdot 10^{-03}$ |
| 60 | $5.05 \cdot 10^{-04}$ | $9.06 \cdot 10^{-02}$ | $4.58 \cdot 10^{-06}$ | $1.25 \cdot 10^{-03}$ | $7.37 \cdot 10^{-06}$ | $1.42 \cdot 10^{-03}$ | $8.18 \cdot 10^{-06}$ | $1.46 \cdot 10^{-03}$ | $8.00 \cdot 10^{-06}$ | $1.38 \cdot 10^{-03}$ |
| 100 | $4.41 \cdot 10^{-06}$ | $8.87 \cdot 10^{-03}$ | $8.71 \cdot 10^{-06}$ | $1.26 \cdot 10^{-03}$ | $6.65 \cdot 10^{-06}$ | $1.20 \cdot 10^{-03}$ | $5.58 \cdot 10^{-06}$ | $8.86 \cdot 10^{-04}$ | $\mathbf{4.69 \cdot 10^{-06}}$ | $\mathbf{7.13 \cdot 10^{-04}}$ |

Os valores de EMQ e erro relativo em cada configuração mostram que o modelo atinge resultados melhores (menores erros) à medida que aumenta a quantidade de pontos de treinamento.

Para $N_{0,b} = 20$, observa-se uma queda significativa no EMQ e erro relativo conforme $N_{\mathcal{D}}$ aumenta, com melhor resultado para $N_{\mathcal{D}} = 6000$, onde o EMQ é $5.58 \cdot 10^{-6}$ e o erro relativo é $1.14 \cdot 10^{-3}$. No entanto, conforme $N_{\mathcal{D}}$ aumenta além desse ponto, o erro relativo volta a subir ligeiramente.

Para $N_{0,b} = 60$, alcança o melhor erro relativo em $N_{\mathcal{D}} = 4000$ com um EMQ de $4.58 \cdot 10^{-6}$ e erro relativo de $1.25 \cdot 10^{-3}$.

Finalmente, para $N_{0,b} = 100$, a configuração com $N_{\mathcal{D}} = 10000$ apresentou EMQ de $4.69 \cdot 10^{-6}$ e erro relativo de $7.13 \cdot 10^{-4}$. Inicialmente, a análise dos resultados na tabela sugere que o aumento no número de pontos de colocação está diretamente associado a uma melhor aproximação da solução. No entanto, essa

relação nem sempre se mantém verdadeira, como será evidenciado posteriormente na análise da equação de Korteweg-de Vries (KdV), onde o aumento excessivo de pontos pode não resultar em melhorias significativas e, em alguns casos, até comprometer a eficiência do modelo.

## 1.3 Discussão dos Resultados

Nas seções anteriores, exploramos as escolhas de configuração de parâmetros que influenciam a acurácia do modelo desenvolvido para resolver a equação de Burgers Viscosa (1.1). O modelo final adotou uma arquitetura de rede neural com 9 camadas e 40 neurônios por camada, enquanto a taxa de aprendizado foi configurada para decair em vários pontos predefinidos ao longo do treinamento. Esse ajuste, realizado em degraus, permitiu alcançar um melhor controle sobre a convergência e resultou em um erro médio quadrático (EMQ) e erro relativo com melhor precisão, conforme evidenciado na Tabela 1.3.

Para representar as condições iniciais e de contorno, foi escolhido um valor de $N_{0,b} = 100$, enquanto para os pontos internos do domínio foi utilizado $N_{\mathcal{D}} = 10000$. Essa combinação apresentou o melhor erro relativo registrado na tabela, garantindo a precisão e robustez do modelo. Nesta análise, o foco esteve na escolha da arquitetura da rede e na configuração da taxa de aprendizado; aspectos como o otimizador, a função de ativação e o inicializador de pesos não foram o foco da otimização.

Na Figura 1.8, comparamos a solução obtida pelo nosso modelo, representada em preto e parametrizada com as configurações de parâmetros de melhor desempenho, com a solução de referência apresentada por Raissi et al. [35], representada em azul. Essa comparação visual evidencia que o modelo proposto é capaz de reproduzir a solução de referência com elevada precisão, demonstrando a eficácia das escolhas de parâmetros realizadas no treinamento.

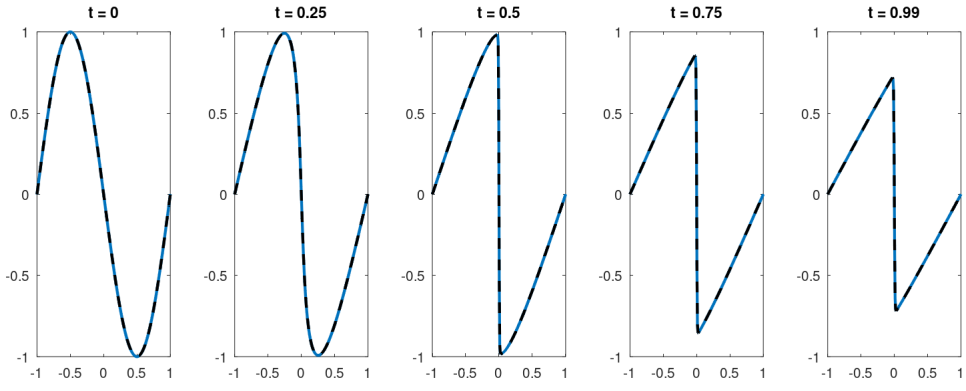



Figura 1.8: Apresentação da solução em diferentes tempos

# Capítulo 2

# Implementação "Automática" de PINNs: DeepXDE

Este capítulo tem por objetivo apresentar a biblioteca DeepXDE, desenvolvida por Lu Lu et.al. para o ambiente Python, e demonstrar como ela facilita a implementação de Redes Neurais Guiadas pela Física (PINNs). A DeepXDE oferece um conjunto completo de ferramentas que automatizam grande parte do processo de configuração e treinamento de PINNs, eliminando a necessidade de implementações manuais, como as apresentadas anteriormente nos Códigos 1.1 a 1.10.

A DeepXDE permite que o usuário configure rapidamente modelos PINN utilizando módulos prontos para definição de domínio, condições iniciais e de fronteira, além das condições de treinamento. Isso reduz significativamente o tempo necessário para simulações e experimentos, resultando em um processo mais eficiente e otimizado. A automação proporcionada pela biblioteca melhora a precisão das aproximações e a qualidade dos resultados obtidos [30].

Este capítulo explora as funcionalidades principais da DeepXDE, exemplificando seu uso na resolução da equação de Burgers viscosa. Essa abordagem permitirá uma comparação direta com a implementação manual apresentada no capítulo anterior. Através desse exemplo, serão demonstradas as principais etapas de configuração e parametrização fornecidas pela biblioteca, ilustrando como ela facilita a implementação e a otimização de modelos PINN para esse tipo de equação diferencial parcial. Assim como na abordagem manual, apresentaremos os passos em uma estrutura organizada para garantir a clareza e fluidez da explicação.

**Passo 1. Importação das Bibliotecas Necessárias:**

No Código 2.1, realizamos a importação das bibliotecas necessárias para a implementação utilizando a DeepXDE. Nesta etapa, também deixamos espaço para fixar a semente aleatória, garantindo a reprodutibilidade dos experimentos.[1]

[1] A implementação computacional apresentada foi realizada utilizando o ambiente Google Colab

Código 2.1: Importação das Bibliotecas Necessárias

```
#pip install deepxde -q   # -q para ele instalar sem as mensagens, retirar o -q para verificar o que instalou
import deepxde as dde

import os
os.environ['DDE_BACKEND'] = 'tensorflow'  # existem outras opções, como tensorflow.compat.v1, pytorch, jax, paddle.

import deepxde as dde
import numpy as np
import tensorflow as tf

# deixar a semente aleatória fixa
seed_value = 0
np.random.seed(seed_value)
tf.random.set_seed(seed_value)
dde.config.set_random_seed(seed_value)
```

No entanto, em alguns casos, a fixação da semente aleatória não foi suficiente para garantir a consistência dos resultados. Para resolver isso, foi necessário fixar o inicializador de pesos manualmente. Esse procedimento será detalhado posteriormente no Código 2.5.

**Passo 2. Definição da Geometria Computacional e do Domínio de Tempo:**

No Código 2.2, definimos uma geometria computacional e um domínio de tempo. Utilizamos as classes internas `Interval` e `TimeDomain` da biblioteca DeepXDE, combinando ambos os domínios por meio da classe `GeometryXTime`, conforme exemplificado.

Código 2.2: Geometria do Problema

```
# Definição da geometria espacial
geom = dde.geometry.Interval(-1, 1)  # Define um intervalo para x, coordenada espacial, no intervalo (-1, 1)
# Definição do domínio temporal
timedomain = dde.geometry.TimeDomain(0, 1)  # Define um domínio de tempo para t, coordenada temporal, no intervalo (0, 1)

# Combinando a geometria espacial e o domínio temporal
geomtime = dde.geometry.GeometryXTime(geom, timedomain)  # Combina a geometria espacial e o domínio temporal em uma geometria espacial-temporal
```

**Passo 3. Definição do Resíduo da Equação de Burgers Viscosa:**

No Código 2.3, definimos o resíduo da equação de Burgers Viscosa, que será utilizado como parte do processo de treinamento do modelo.

Código 2.3: Resíduo da Equação de Burgers Viscosa

```
def pde(x, y):
    # x = (x, t) representa as coordenadas espaciais (x) e temporais (t)
    # y = u(x, t) representa a solução da equação de Burgers viscosa no ponto (x, t)

    # Calculando a primeira derivada em relação à coordenada espacial (x)
    dy_x = dde.grad.jacobian(y, x, i=0, j=0)

    # Calculando a primeira derivada em relação à coordenada temporal (t)
    dy_t = dde.grad.jacobian(y, x, i=0, j=1)

    # Calculando a segunda derivada em relação à coordenada espacial (x)
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)

    # Retorna a equação diferencial parcial de Burgers viscosa
    return dy_t + y * dy_x - 0.01 / np.pi * dy_xx
```

**Passo 4. Definição das Condições de Contorno e Inicial:**

No Código 2.4, definimos a condição de contorno e a condição inicial. Para as condições de contorno, utilizamos `on_boundary`, que considera todo o limite do domínio computacional como a condição de limite. Incluímos também o espaço `geomtime` (a geometria do tempo criada) e `on_boundary` como as condições de contorno (`BCs`) na função `DirichletBC` fornecida pela DeepXDE.

Além disso, definimos `IC`, que representa a condição inicial para a equação de Burgers Viscosa. A condição inicial foi especificada utilizando o domínio computacional, a função inicial e o parâmetro `on_initial`.

Código 2.4: Condições de Contorno e Inicial

```
bc = dde.icbc.DirichletBC(geomtime, lambda x: 0, lambda _, on_boundary: on_boundary)
ic = dde.icbc.IC(geomtime, lambda x: -np.sin(np.pi * x[:, 0:1]), lambda _, on_initial: on_initial)
```

**Passo 5. Definição do Problema `TimePDE`:**

No Código 2.5, definimos o problema `TimePDE`, que combina o resíduo da equação de Burgers Viscosa, as condições de contorno (`BC`) e as condições iniciais (`IC`), para configurar o modelo a ser resolvido pela DeepXDE.

Código 2.5: Definição do Problema

```
data = dde.data.TimePDE(geomtime, pde, [bc, ic], num_domain=3000, num_boundary=150, num_initial=150)
```

**Passo 6. Escolha da Arquitetura da Rede Neural, Função de Ativação e Inicializador de Pesos:**

No Código 2.6, definimos a arquitetura da rede neural, incluindo a quantidade de camadas e neurônios por camada. Também escolhemos a função de ativação a ser utilizada e o inicializador de pesos, fundamentais para garantir um treinamento estável e eficiente.

Adicionalmente, incluímos um laço `for` necessário para a fixação dos inicializadores de pesos. Esse procedimento se torna relevante em situações onde a configuração padrão não assegura consistência nos resultados.

Código 2.6: Escolha da Arquitetura da Rede Neural, Função de Ativação e Inicializador de Peso

```
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh", "Glorot normal")
'''
    [2]: camada de entrada, 2 entradas, espaço e tempo
    [20]*3: 3 camadas com 20 neurônios
    tanh: função de ativação
    glorot normal: inicializador de pesos
'''

'''
código caso necessário a fixação dos inicializadores de pesos
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh", "zeros")
for d in net.denses:
    d.kernel_initializer = tf.keras.initializers.GlorotNormal()
'''
```

**Passo 7. Construção do Modelo, Escolha do Otimizador e Treinamento:**

No Código 2.7, realizamos a construção do modelo, definimos o otimizador, configuramos a taxa de aprendizado e executamos o treinamento do modelo utilizando as ferramentas da DeepXDE.

Código 2.7: Construção do Modelo, Escolha do Otimizador e Treinamento

```
model = dde.Model(data, net) # criação do modelo com data os dados, e net a Rede Neural
model.compile("adam", lr=1e-3) # compilação do modelo utilizando o otimizador ADAM, com taxa de
↪  aprendizagem lr = 1e-3, como sugerido pela documentação do DeepXDE
'''
    losshistory: lista que armazena o histórico de valores da perda.
    train_state: informações do treinamento, como iterações, tempo
'''
losshistory, train_state = model.train(iterations=10000)
```

A utilização da biblioteca DeepXDE apresenta diversas vantagens que facilitam e aprimoram o desenvolvimento de Redes Neurais Guiadas pela Física (PINNs). Primeiramente, a biblioteca implementa uma ampla variedade de algoritmos específicos para PINNs, permitindo uma compactação significativa de código, o que melhora a clareza e reduz o tempo necessário para o desenvolvimento. Além disso, a DeepXDE suporta domínios complexos sem necessidade de modificações manuais, integra de forma nativa as condições de contorno e iniciais, e oferece diferenciação automática. A biblioteca também possibilita a configuração de diferentes arquiteturas de redes neurais, métodos de amostragem e otimizadores, tornando-se uma ferramenta altamente configurável e eficiente para ajustar parâmetros [30].

Utilizando a DeepXDE, foi possível obter um erro relativo de $2.77 \cdot 10^{-04}$ com a melhor configuração de rede: 9 camadas, 40 neurônios por camada, taxa de aprendizado fixa de $10^{-3}$ e os mesmos otimizadores e inicializadores descritos na Tabela 1.1. Esse valor de erro está na mesma ordem de grandeza dos resultados obtidos com o código manual apresentado anteriormente, conforme ilustrado na Tabela 1.3.

Na Figura 2.1, a solução obtida pelo modelo de Rede Neural utilizando a biblioteca DeepXDE é representada em preto, enquanto a solução de referência, apresentada por Raissi et al. e Lu Lu et al. [30, 35], é mostrada em azul. Essa comparação visual evidencia que o uso da DeepXDE permite alcançar resultados precisos e comparáveis aos métodos tradicionais, oferecendo ainda o benefício adicional de um desenvolvimento mais rápido e simplificado.

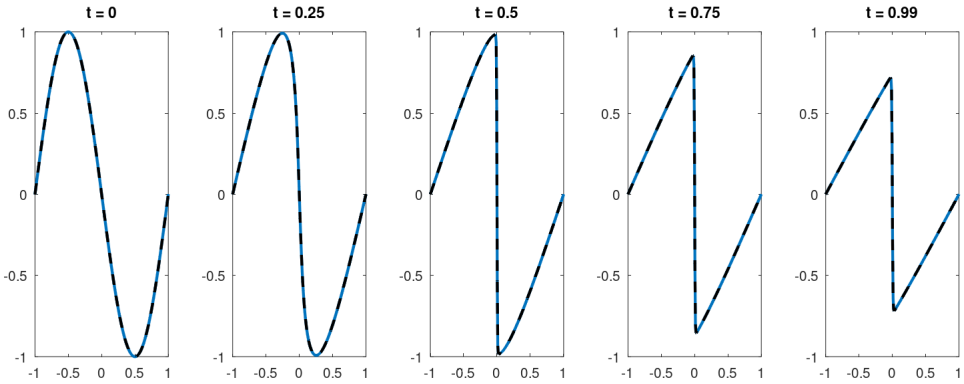



Figura 2.1: Apresentação da solução em diferentes tempos

## 2.1 Equação de KdV via DeepXDE

Observamos que a utilização da biblioteca DeepXDE proporciona resultados comparáveis, e em alguns casos até melhores, do que os obtidos com a implementação manual utilizando os Códigos 1.1 a 1.10.

Uma das principais vantagens da DeepXDE é a implementação de diferenciação automática, como no código 2.3, que facilita significativamente o uso da biblioteca em equações diferenciais parciais (EDPs) mais complexas, como a Equação de Korteweg–de Vries (KdV). Essa equação, conhecida por incluir derivadas de ordem mais alta, apresenta um desafio adicional no cálculo dessas derivadas. Com os códigos manuais, esse

processo torna-se mais demorado e menos preciso, enquanto a DeepXDE automatiza esse cálculo de forma eficiente.

### 2.1.1 Equação de Korteweg–De Vries (KdV)

A Equação Diferencial Parcial (EDP) abordada neste estudo busca determinar $u = u(x,t)$, que satisfaz as seguintes condições:

$$u_t - \frac{3}{2}uu_x - \frac{1}{6}u_{xxx} = 0, \qquad (x,t) \in \mathbb{R} \times (0,\infty). \tag{2.1}$$

Essa equação, conhecida como Equação de Korteweg–De Vries (KdV), foi formulada inicialmente por Diederik Korteweg e Gustav de Vries em 1895 [27]. Originalmente, ela surgiu como um modelo matemático para descrever a propagação de ondas aquáticas em águas rasas. Uma das características mais notáveis dessa equação é a existência das chamadas *ondas solitárias* — elevações isoladas que se propagam sem mudar de forma ou perder energia. Entretanto, a KdV ganhou notoriedade principalmente pela descoberta dos **sólitons**, uma classe especial de ondas solitárias que apresenta um comportamento extraordinário.

O termo *sóliton* foi introduzido em 1965 por Zabusky e Kruskal, que observaram uma propriedade singular das soluções da equação de KdV. Como descrevem Gondar e Cipolatti [10]:

> [...] A palavra 'sóliton' surgiu em 1965 a partir dos trabalhos de Zabuski e Kruskal. Eles observaram uma propriedade notável: o fato de que ondas solitárias da KdV, [...] ao se encontrarem, uma passa pela outra sem mudar de forma [...]. Essa é uma propriedade importante, porque mostra que a energia pode se propagar em pacotes localizados sem se dispersar. Como a KdV é uma equação não linear, estas soluções são excepcionais e, por isso, decidiram chamá-las de sólitons. O sufixo 'on' (que em grego significa partícula) ilustra, neste caso, o comportamento tipo partícula destas ondas. Esse fenômeno não é uma exclusividade da KdV. [...].

Esse trabalho pioneiro de Zabusky e Kruskal marcou um ponto de virada na física matemática, pois demonstrou que a equação de KdV transcende seu contexto original de ondas aquáticas. Sua descoberta impulsionou uma série de pesquisas subsequentes que mostraram que a equação KdV pode modelar fenômenos em diversas áreas da ciência, como óptica não linear, teoria de plasmas, supercondutividade, física de partículas e biologia [19]. Essa universalidade reforça o caráter fundamental da equação de Korteweg–De Vries na descrição de sistemas não lineares, especialmente aqueles que apresentam propagação de pacotes de energia localizados sem dissipação.

#### 2.1.1.1 Solução Exata e Formulação do Problema

A onda solitária solução exata para a equação de KdV [25], é dada por:

$$u(x,t) = A \cdot \operatorname{sech}^2\left(k(x - ct)\right), \tag{2.2}$$

onde $c = -\frac{A}{2}$ representa a velocidade da onda, $A$ é a amplitude, e $k = \sqrt{\frac{3A}{4}}$ está relacionado à largura da onda.

Aplicando essa solução como condição inicial em (2.1), obtemos o seguinte problema:

$$\begin{aligned} u_t - \frac{3}{2}uu_x - \frac{1}{6}u_{xxx} = 0, \qquad & (x,t) \in \mathbb{R} \times (0,\infty), \\ u(x,0) = A\text{sech}^2(kx). & \end{aligned}$$

Vamos utilizar esse problema de valor inicial para verificar a acurácia das redes neurais.

#### 2.1.1.2 Importância da KdV e Métodos Computacionais

A relevância da Equação de KdV na modelagem de fenômenos naturais e sua complexidade matemática tornam-na um caso de teste ideal para métodos computacionais modernos, como as Redes Neurais Guiadas pela Física (PINNs). Raissi [36], por exemplo, demonstra a aplicabilidade das PINNs na resolução da KdV, evidenciando a eficiência dessas redes na captura das dinâmicas complexas associadas a ondas solitárias.

Adicionalmente, o termo de terceira derivada, $u_{xxx}$, introduz dificuldades significativas para métodos numéricos tradicionais, devido à necessidade de alta precisão no cálculo de derivadas superiores. Nesse contexto, bibliotecas como a DeepXDE [30], que incorporam diferenciação automática, têm se mostrado ferramentas robustas e eficientes, simplificando implementações e fornecendo maior precisão em problemas envolvendo derivadas de alta ordem.

#### 2.1.1.3 Domínio Limitado e Condições de Contorno

Na prática computacional, o domínio contínuo $(x,t) \in \mathbb{R} \times (0,\infty)$ é restringido a um domínio finito para viabilizar a resolução numérica. Neste trabalho, adotamos as seguintes condições:

$$\begin{aligned} u_t - \frac{3}{2}uu_x - \frac{1}{6}u_{xxx} = 0, \qquad & (x,t) \in (-50,50) \times (0,60), \\ u(x,0) = A\text{sech}^2(kx), \qquad & x \in (-50,50), \\ u(-50,t) = u(50,t) = 0, \qquad & t \in (0,60), \end{aligned}$$

**Observação 2.1** (Condição de Fronteira)**.** *Vale ressaltar que, teoricamente, a solução exata para a KdV satisfaz:*

$$u(x,t) \to 0, \quad \textit{quando} \quad |x| \to +\infty.$$

*No entanto, para a resolução computacional do problema, é comum adotar condições de contorno periódicas, especialmente ao utilizar métodos numéricos baseados na Transformada de Fourier, como os métodos espectrais [39]. Neste trabalho, optamos por condições de contorno nulas, que também se adequam à solução exata no domínio finito considerado.*

Resolvemos, então, os seguintes problemas:

**Problema 1. Operador Diferencial:**

$$\min |\mathcal{D}(x,t)| \quad \text{sujeito a} \quad (x,t) \in (-50,50) \times (0,60),$$

onde

$$\mathcal{D} := u_{Nt} - \frac{3}{2} u_N u_{Nx} - \frac{1}{6} u_{Nxxx}.$$

**Problema 2. Condição Inicial:**

$$\min |u_N(x,0) - u_0(x)| \quad \text{sujeito a} \quad x \in (-50,50).$$

**Problema 3. Condição de Contorno:**

$$\min |u_N(-50,t)| + |u_N(50,t)| \quad \text{sujeito a} \quad t \in (0,60).$$

Os quais nos fornecem a seguinte função de perda:

$$\text{EMQ} = \text{EMQ}_{\mathcal{D}} + \text{EMQ}_0 + \text{EMQ}_b,$$

onde:

**Problema 1. Operador Diferencial:**

$$\text{EMQ}_{\mathcal{D}} = \frac{1}{N_{\mathcal{D}}} \sum_{i=1}^{N_{\mathcal{D}}} |\mathcal{D}(x_i,t_i)|^2,$$

onde $(x_i,t_i), i = 1,\ldots,N_{\mathcal{D}}$, são $N_{\mathcal{D}}$ pontos de colocação escolhidos aleatoriamente em $(-50,50)\times(0,60)$;

**Problema 2. Condição Inicial:**

$$\text{EMQ}_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} |u_N(x_i,0) - u_0(x_i)|^2,$$

onde $x_i, i = 1,\ldots,N_0$, representam $N_0$ pontos de colocação no intervalo $(-50,50)$;

**Problema 3. Condição de Contorno:**

$$\text{EMQ}_b = \frac{1}{N_b} \sum_{i=1}^{N_b} |u_N(x_i,t_i)|^2,$$

onde $(x_i,t_i), i = 1,\ldots,N_b$, correspondem a $N_b$ pontos de colocação ao longo do conjunto cartesiano $\{-50,50\} \times (0,60)$.

Os testes foram realizados com os parâmetros da Tabela 2.1, mantendo a semente aleatória fixa. O critério aqui considerado para escolha de camada, neurônio e pontos, foi o erro relativo, obtido ao comparar a solução da rede neural com a solução exata. Os resultados estão apresentados na Tabela 2.2.

Tabela 2.1: Parâmetros utilizados

| **Parâmetro** | **Valor** |
|---|---|
| Otimizador | Adam |
| Função de ativação | tanh |
| Inicializador de pesos | Glorot Normal |
| Passos de treinamento | 30 000 |
| Pontos de colocação no interior ($N_{\mathcal{D}}$) | 30 000 |
| Pontos de colocação na borda ($N_b$) | 300 |
| Pontos de colocação na condição inicial ($N_0$) | 300 |
| Taxa de aprendizado | $10^{-3}$ |
| Domínio Espacial | $[-50, 50]$ |
| Tempo final | 60 |

Tabela 2.2: Erro relativo para cada Camada e Neurônio

| C \ N | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | $9,96 \cdot 10^{-01}$ | $1,03 \cdot 10^{+00}$ | $3,36 \cdot 10^{-02}$ | $2,16 \cdot 10^{+00}$ | $1,23 \cdot 10^{-02}$ | $3,09 \cdot 10^{-02}$ | $2,85 \cdot 10^{-02}$ | $1,63 \cdot 10^{-02}$ | $3,84 \cdot 10^{-02}$ | $1,63 \cdot 10^{+00}$ |
| **2** | $1,00 \cdot 10^{+00}$ | $3,07 \cdot 10^{-02}$ | $2,41 \cdot 10^{-01}$ | $1,66 \cdot 10^{-02}$ | $5,23 \cdot 10^{-02}$ | $3,46 \cdot 10^{-01}$ | $1,04 \cdot 10^{-01}$ | $4,45 \cdot 10^{-01}$ | $1,93 \cdot 10^{-02}$ | $\mathbf{7,02 \cdot 10^{-04}}$ |
| **3** | $1,00 \cdot 10^{+00}$ | $1,19 \cdot 10^{-02}$ | $7,25 \cdot 10^{-03}$ | $4,02 \cdot 10^{-02}$ | $7,27 \cdot 10^{-01}$ | $3,52 \cdot 10^{-01}$ | $2,03 \cdot 10^{-03}$ | $3,81 \cdot 10^{-02}$ | $2,97 \cdot 10^{-01}$ | N/A |
| **4** | $1,00 \cdot 10^{+00}$ | $3,26 \cdot 10^{-03}$ | $7,81 \cdot 10^{-02}$ | $1,39 \cdot 10^{-01}$ | $9,91 \cdot 10^{-01}$ | $7,86 \cdot 10^{-01}$ | $1,04 \cdot 10^{+00}$ | $1,39 \cdot 10^{-01}$ | $2,05 \cdot 10^{-01}$ | $1,72 \cdot 10^{-01}$ |
| **5** | $1,00 \cdot 10^{+00}$ | $4,68 \cdot 10^{-03}$ | $4,03 \cdot 10^{-01}$ | $8,90 \cdot 10^{-03}$ | $3,85 \cdot 10^{-01}$ | $3,39 \cdot 10^{-03}$ | $4,64 \cdot 10^{-01}$ | $9,76 \cdot 10^{-02}$ | $4,47 \cdot 10^{-02}$ | $1,69 \cdot 10^{-03}$ |
| **6** | $1,00 \cdot 10^{+00}$ | $2,10 \cdot 10^{-01}$ | $1,75 \cdot 10^{-02}$ | $4,57 \cdot 10^{-01}$ | $5,83 \cdot 10^{-01}$ | $1,83 \cdot 10^{-03}$ | $5,07 \cdot 10^{-01}$ | N/A | N/A | N/A |
| **7** | $1,00 \cdot 10^{+00}$ | $2,30 \cdot 10^{-01}$ | $3,86 \cdot 10^{-01}$ | $1,02 \cdot 10^{-01}$ | $9,03 \cdot 10^{-01}$ | $\mathbf{9,63 \cdot 10^{-04}}$ | N/A | N/A | N/A | N/A |

Os valores indicados como N/A na Tabela 2.2 correspondem às configurações de (Camada, Neurônio) que não foram testadas.

Observe que foram identificadas duas configurações mais convenientes com os menores erros relativos: a dupla (Camada, Neurônio) = $(2, 10)$ e a dupla (Camada, Neurônio) = $(7, 6)$. Optamos por utilizar a configuração (Camada, Neurônio) = $(7, 6)$ devido ao seu menor custo computacional, resultando em um menor tempo de processamento.

Com a configuração fixada em (Camada, Neurônio) = $(7, 6)$, realizamos uma nova análise, amostrando diferentes quantidades de pontos para avaliar o impacto na precisão e no desempenho.

Tabela 2.3: Escolhendo diferentes pontos amostrados

| $N_b = N_0$ \ $N_{\mathcal{D}}$ | **15000** | **30000** | **60000** |
|---|---|---|---|
| **100** | $8,37 \cdot 10^{-01}$ | $4,91 \cdot 10^{-01}$ | $6,02 \cdot 10^{-01}$ |
| **200** | $2,59 \cdot 10^{-03}$ | $8,79 \cdot 10^{-03}$ | $6,32 \cdot 10^{-03}$ |
| **300** | $1,39 \cdot 10^{-03}$ | $\mathbf{9,63 \cdot 10^{-04}}$ | $9,08 \cdot 10^{-01}$ |
| **400** | $1,06 \cdot 10^{-03}$ | $3,87 \cdot 10^{-01}$ | $6,57 \cdot 10^{-01}$ |
| **500** | $2,27 \cdot 10^{-03}$ | $7,96 \cdot 10^{-01}$ | $1,28 \cdot 10^{-02}$ |

A Tabela 2.3 apresenta os erros relativos obtidos ao variar a quantidade de pontos amostrados para as condições iniciais ($N_0$) e de contorno ($N_b$), bem como os pontos internos do domínio ($N_{\mathcal{D}}$). Esses resultados refletem a precisão da solução obtida pela Rede Neural Guiada pela Física (PINN) ao resolver a equação de Korteweg–de Vries (KdV).

Os valores indicam que o erro relativo diminui consideravelmente ao aumentar os pontos de colocação em certos intervalos. A configuração com $N_0 = N_b = 300$ e $N_{\mathcal{D}} = 30000$ apresentou o menor erro relativo ($9,63 \cdot 10^{-4}$), destacando-se como a mais eficiente. Isso evidencia que, para essa quantidade de pontos, a rede consegue capturar de forma mais precisa as dinâmicas da solução exata da KdV.

No entanto, também é possível observar que o aumento descontrolado no número de pontos, como em $N_{\mathcal{D}} = 60000$, nem sempre resulta em maior precisão, podendo levar a maiores erros relativos devido ao aumento da complexidade do modelo e possíveis problemas de sobreajuste ou estabilidade numérica.

Dessa forma, a análise da tabela permite identificar uma configuração de pontos que balanceia precisão e custo computacional, sendo $N_0 = N_b = 300$ e $N_{\mathcal{D}} = 30000$ a escolha mais adequada para o problema em questão.

## 2.2 Discussão de resultados

Na Figura 2.2, comparamos a solução obtida pelo nosso Modelo de Rede Neural, parametrizado com a configuração que apresentou o melhor resultado de erro na Tabela 2.3, com a solução exata descrita em (2.2). A solução do modelo é representada em preto, enquanto a solução exata aparece em azul. Essa comparação evidencia a capacidade da rede neural de capturar com alta precisão as características da solução exata, mesmo em tempos diferentes.

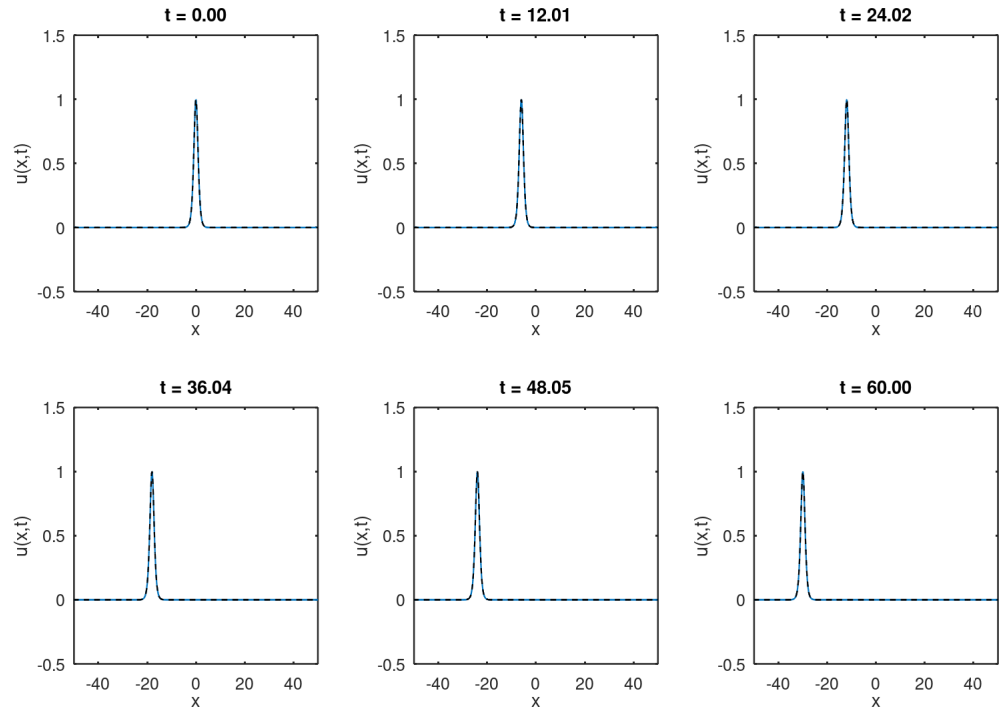



Figura 2.2: Apresentação da solução de KdV, em diferentes tempos

A análise dos resultados foi conduzida considerando a configuração $N_0 = N_b = 300$ e $N_{\mathcal{D}} = 30000$, que apresentou o menor erro relativo $(9,63 \cdot 10^{-4})$ na Tabela 2.3. Essa escolha equilibra precisão e custo computacional, sendo uma configuração eficiente para o problema. É importante destacar que aumentar o número de pontos de colocação nem sempre resultou em maior precisão, como observado para $N_{\mathcal{D}} = 60000$, onde o erro aumentou consideravelmente.

Para a rede neural, utilizamos uma arquitetura com 7 camadas e 6 neurônios por camada. Essa configuração foi escolhida após testes que indicaram sua eficiência em termos de precisão e menor tempo de treinamento, comparada a outras combinações testadas, como (Camada, Neurônio) $= (2, 10)$. Essa escolha evidencia a importância de balancear a profundidade da rede e a capacidade de representação dos neurônios, especialmente em problemas que envolvem derivadas de alta ordem, como a equação de KdV.

A taxa de aprendizado também desempenhou um papel crucial na convergência do modelo. Optamos por utilizar uma taxa de aprendizado fixa de $10^{-3}$, seguindo o que é amplamente adotado na literatura sobre a biblioteca DeepXDE como uma configuração padrão. Essa escolha permitiu uma convergência estável e resultados precisos, garantindo que o modelo capturasse as dinâmicas da solução exata sem oscilações ou instabilidades.

Por fim, a biblioteca DeepXDE demonstrou ser uma ferramenta eficiente para implementação. Sua integração de diferenciação automática facilitou o cálculo das derivadas de ordem superior, como $u_{xxx}$, reduzindo

significativamente o esforço computacional e aumentando a precisão do modelo. A flexibilidade para configurar arquiteturas de rede, condições de contorno e métodos de amostragem também foi essencial para alcançar os resultados apresentados.

Os resultados obtidos reafirmam o potencial das Redes Neurais Guiadas pela Física (PINNs) para resolver equações diferenciais parciais de alta complexidade, como a equação de KdV, com precisão e eficiência computacional.

## 2.3 Korteweg-de Vries (KdV) e Korteweg-de Vries Modificada (mKdV) com Amplitude Variável

Este capítulo foi elaborado em conjunto com o Professor Thiago de Oliveira Quinelato e seu orientando Guilherme Furquim, cujas contribuições foram essenciais para o desenvolvimento deste estudo. Ressaltamos a colaboração direta na formulação, implementação e análise das soluções apresentadas.

Uma variante amplamente conhecida da equação de Korteweg-de Vries (KdV) é a equação modificada de Korteweg-de Vries (mKdV). Ambas as equações, intimamente ligadas pela transformação de Miura, são amplamente empregadas na modelagem de fenômenos não lineares em diversos campos. Enquanto a equação KdV é frequentemente associada à propagação de ondas em águas rasas, a mKdV encontra aplicações em contextos como redes anarmônicas [45], ondas de Alfvén [20], engarrafamentos de tráfego [26, 33] e jatos oceânicos finos [7].

O problema que será abordado é descrito por:

$$u_t + u^p u_x + u_{xxx} = 0, \quad x \in \mathbb{R},\, t \in (0, \infty), \tag{2.3a}$$

$$u(x, 0) = u_0(x, a), \qquad x \in \mathbb{R}, \tag{2.3b}$$

onde $p$ é igual a 1 (para a equação KdV) ou 2 (para a equação mKdV), e $a$ é um parâmetro relacionado à amplitude da condição inicial $u_0(x, a)$.

A novidade neste estudo está em buscar uma solução numérica $u(x, t, a)$ que funcione para todos os valores de $a \in A \subset \mathbb{R}$ simultaneamente. Diferentemente de abordagens anteriores, como na Seção 2.1, que requerem simulações individuais para cada valor do parâmetro $a$, aqui desenvolvemos um modelo que resolve o problema de forma generalizada.

O problema descrito pela Equação (2.3) possui solução exata na forma de um soliton:

$$u(x,t) = \begin{cases} a\operatorname{sech}^2\left(k(x-ct)\right),\ k = \sqrt{\dfrac{a}{12}},\ c = \dfrac{a}{3}, & \text{se } p = 1, \\ a\operatorname{sech}\left(k(x-ct)\right),\ k = \dfrac{a}{\sqrt{6}},\ c = \dfrac{a^2}{6}, & \text{se } p = 2. \end{cases} \tag{2.4}$$

Como a solução $u(x, t)$ depende da amplitude $a$, indicamos $u(x, t, a)$ para destacar essa dependência. Para o tratamento numérico, o intervalo infinito em $x$ é truncado para $[\xi_1, \xi_2]$, com condições de contorno baseadas na solução exata. O parâmetro $a$ varia em um intervalo $A = [\alpha_1, \alpha_2]$.

A formulação numérica transforma o problema em um problema de otimização, buscando uma função $u_N(x,t,a)$ que minimiza simultaneamente os seguintes erros:

**Problema 1. Operador Diferencial:**

$$\min |\mathcal{D}(x,t,a)| \quad \text{sujeito a } (x,t,a) \in (\xi_1,\xi_2)\times(0,T)\times(\alpha_1,\alpha_2),$$

onde

$$\mathcal{D}(x,t,a) := (u_N)_t + u_N^p(u_N)_x + (u_N)_{xxx}.$$

**Problema 2. Condição Inicial (no tempo $t=0$):**

$$\min |u_N(x,0,a) - u_0(x,a)| \quad \text{sujeito a } t=0.$$

**Problema 3. Condição de Contorno:**

$$\min |u_N(x,t,a) - u(x,t,a)| \quad \text{sujeito a } x \in \{\xi_1,\xi_2\}.$$

A função $u(x,t,a)$ é aproximada por uma rede neural treinada para minimizar a função de custo total [3, 12, 35]:

$$\text{EMQ} = \text{EMQ}_{\mathcal{D}} + \text{EMQ}_0 + \text{EMQ}_b,$$

onde:

- $$\text{EMQ}_{\mathcal{D}} = \frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}} |\mathcal{D}(x_i,t_i,a_i)|^2,$$

  onde $(x_i,t_i,a_i)$, $i=1,\ldots,N_{\mathcal{D}}$, representa $N_{\mathcal{D}}$ pontos de colocação escolhidos aleatoriamente em $(\xi_1,\xi_2)\times(0,T)\times(\alpha_1,\alpha_2)$.

- $$\text{EMQ}_0 = \frac{1}{N_0}\sum_{i=1}^{N_0} |u_N(x_i,0,a_i) - u_0(x_i,a_i)|^2,$$

  onde $(x_i,a_i)$, $i=1,\ldots,N_0$, representa $N_0$ pontos de colocação em $(\xi_1,\xi_2)\times\{0\}\times(\alpha_1,\alpha_2)$.

- $$\text{EMQ}_b = \frac{1}{N_b}\sum_{i=1}^{N_b} |u_N(x_i,t_i,a_i) - u(x_i,t_i,a_i)|^2,$$

  onde $(x_i,t_i,a_i)$, $i=1,\ldots,N_b$, representa $N_b$ pontos de colocação no conjunto cartesiano $\{\xi_1,\xi_2\}\times(0,T)\times(\alpha_1,\alpha_2)$.

Os parâmetros de configuração da rede neural estão representados na Tabela 2.4. Foram treinadas duas redes, uma para cada equação do Modelo (2.4).

Tabela 2.4: Parâmetros utilizados nos exemplos

| Parâmetro | Valor |
|---|---|
| Camadas | 5 |
| Neurônios | 60 |
| Otimizador | Adam |
| Taxa de aprendizado | $[10^{-2}, 10^{-5}]$ |
| Função de ativação | tanh |
| Inicializador de pesos | Glorot Normal |
| Épocas | 100 000 |
| Pontos de colocação no interior ($N_{\mathcal{D}}$) | 10 000 |
| Pontos de colocação na borda ($N_b$) | 4 000 |
| Pontos de colocação na condição inicial ($N_0$) | 4 000 |
| Épocas entre a reamostragem de pontos de colocação interiores | 1 000 |
| Domínio espacial $[\xi_1, \xi_2]$ | $[-10, 10]$ |
| Domínio da amplitude do soliton $[\alpha_1, \alpha_2]$ | $[1, 2]$ |
| Tempo final ($T$) | 15 |

Inicialmente, utilizou-se uma taxa de aprendizado fixa de $1 \cdot 10^{-3}$, mas observou-se que essa estratégia gerava oscilações e convergência lenta. Para corrigir esse problema, adotou-se uma estratégia de ajuste progressivo da taxa de aprendizado, conforme detalhado na Tabela 2.5. Essa abordagem resultou em maior estabilidade durante o treinamento e maior precisão nos resultados.

Tabela 2.5: Taxas de aprendizado utilizadas nos exemplos.

| **Épocas** | **Taxa de Aprendizado** |
|---|---|
| $0 - 3000$ | $1 \cdot 10^{-2}$ |
| $3001 - 6000$ | $5 \cdot 10^{-3}$ |
| $6001 - 10000$ | $1 \cdot 10^{-3}$ |
| $10001 - 15000$ | $7 \cdot 10^{-4}$ |
| $15001 - 20000$ | $5 \cdot 10^{-4}$ |
| $20001 - 25000$ | $3 \cdot 10^{-4}$ |
| $25001 - 40000$ | $1 \cdot 10^{-4}$ |
| $40001 - 55000$ | $7 \cdot 10^{-5}$ |
| $55001 - 70000$ | $5 \cdot 10^{-5}$ |
| $70001 - 85000$ | $3 \cdot 10^{-5}$ |
| $85001 - 100000$ | $1 \cdot 10^{-5}$ |

A seleção desses pontos de ajuste foi baseada na análise do comportamento durante o treinamento. Observamos que a utilização de uma taxa de aprendizado fixa inicial resultou em padrões de erro instáveis, indicando a necessidade de reduções em momentos-chave do treinamento.

Com a implementação desses ajustes progressivos, obtivemos um controle mais refinado do processo de aprendizagem, permitindo ajustes mais precisos dos parâmetros e uma redução significativa do erro residual. Essa abordagem resultou em soluções mais precisas e consistentes para as equações de Korteweg-de Vries (KdV) e de Korteweg-de Vries modificada (mKdV), conforme ilustrado nas Figuras 2.3 e 2.4.

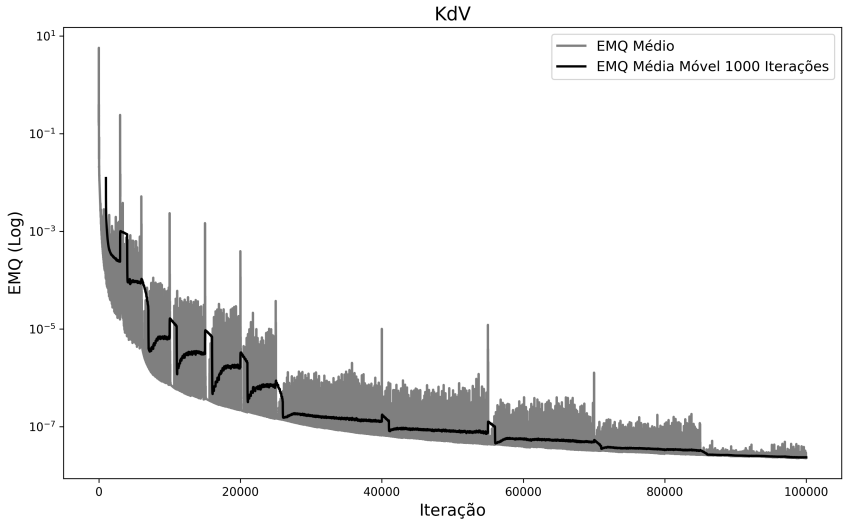



Figura 2.3: EMQ ao longo das iterações para a equação KdV.

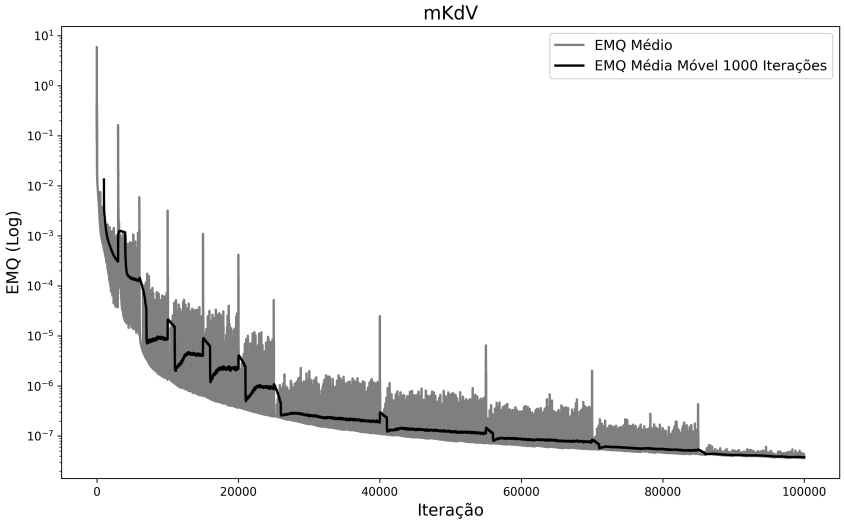



Figura 2.4: EMQ ao longo das iterações para a equação mKdV.

Para avaliar a eficiência e precisão do modelo de rede neural apresentado, realizamos uma comparação entre as soluções exatas e aquelas computadas pela Rede Neural. Duas medições de erro foram realizadas para cada caso exato.

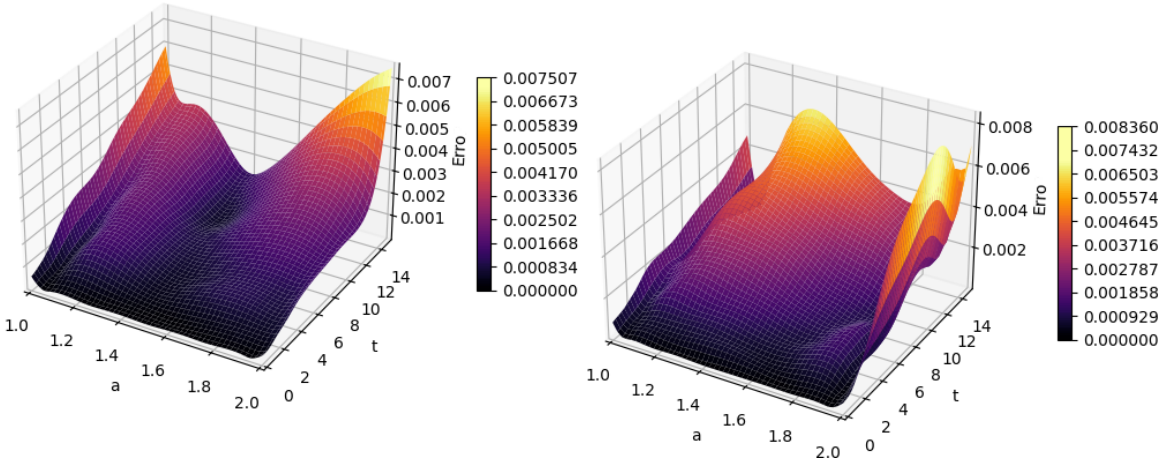
A medição do erro avalia a precisão das soluções computadas pela rede neural para uma amplitude de soliton escolhida $a \in [\alpha_1, \alpha_2]$ no instante de tempo $t \in (0, T]$. A partir de um conjunto de 200 pontos igualmente espaçados $\{x_i\}$ no eixo $x$, definimos um vetor $U(t, a)$ tal que $U(t, a)[i] = u(x_i, t, a)$, onde $u$ é uma solução de referência dada na Eq. (2.4), e um vetor $U_N(t, a)$ tal que $U_N(t, a)[i]$ é a aproximação para $U(t, a)[i]$ calculada pela rede neural. O erro de aproximação relativo $e(t, a)$ é medido usando a norma $L^2$:

$$e(t, a) = \frac{\|U(t, a) - U_N(t, a)\|}{\|U(t, a)\|}.$$

Para ambas as equações, avaliamos o erro $e(t, a)$ em cada ponto de uma grade retangular de 50 pontos igualmente espaçados no domínio da amplitude $[1, 2]$ e 50 pontos igualmente espaçados no domínio do tempo $[0, 15]$, incluindo os pontos extremos.

| | **Erro Máximo** | **Erro médio** |
|---|---|---|
| **KdV** | $7{,}506\,74 \cdot 10^{-3}$ | $1{,}565\,90 \cdot 10^{-3}$ |
| **mKdV** | $8{,}332\,73 \cdot 10^{-3}$ | $2{,}474\,66 \cdot 10^{-3}$ |

Tabela 2.6: Erros Máximos e Médios avaliados na grade.




(a) Superfície de Erro: KdV

(b) Superfície de Erro: mKdV

Figura 2.5: Comparação das superfícies de erro para KdV e mKdV.

### 2.3.1 Discussão de Resultados

Os resultados para o problema envolvendo as equações de Korteweg-de Vries (KdV) e Korteweg-de Vries modificada (mKdV) foram avaliados com base nos erros de aproximação relativos calculados ao longo de uma grade retangular no domínio das amplitudes e tempos. Os valores de erro máximo e erro médio obtidos estão apresentados na Tabela 2.6, onde observamos um erro relativo máximo na ordem de $10^{-3}$.

A análise dos erros revela que a rede neural apresentou um desempenho satisfatório em ambos os casos. Para a equação KdV, o erro máximo registrado foi de $7{,}506\,74 \cdot 10^{-3}$, enquanto o erro médio foi de $1{,}565\,90 \cdot 10^{-3}$. Já para a equação mKdV, os valores foram ligeiramente maiores, com um erro máximo de $8{,}332\,73 \cdot 10^{-3}$ e erro médio de $2{,}474\,66 \cdot 10^{-3}$. Esses resultados demonstram que o modelo conseguiu capturar com boa acurácia a dinâmica das soluções de referência, mesmo considerando a amplitude variável no intervalo $[1, 2]$.

Contudo, cabe destacar um problema específico encontrado durante a reprodução dos resultados para este estudo. Acreditamos que o ambiente de desenvolvimento utilizado, o Google Colab, passou por alterações que impactaram a execução do modelo. Essas alterações podem estar relacionadas a mudanças na GPU, nas bibliotecas instaladas ou no comportamento padrão do ambiente, resultando em inconsistências na reprodução dos resultados previamente obtidos. Apesar de todos os esforços para reproduzir os experimentos, incluindo a fixação de sementes aleatórias, a reconfiguração do ambiente e a instalação de versões específicas das bibliotecas, as métricas originais apresentaram variações significativas, inviabilizando a obtenção dos mesmos valores de erro observados inicialmente.

Essa limitação reforça a importância de realizar implementações em ambientes controlados, o que pode minimizar esses impactos e garantir maior reprodutibilidade dos experimentos.

Por fim, embora as inconsistências observadas representem uma limitação técnica, os valores apresentados na Tabela 2.6 ainda demonstram a capacidade das Redes Neurais Guiadas pela Física (PINNs) em resolver

problemas envolvendo as equações KdV e mKdV com boa precisão em contextos de amplitude variável.

# Capítulo 3

# Comparação entre as implementações: Vantagens e Desvantagens

Neste capítulo, realizamos uma análise comparativa entre duas abordagens distintas utilizadas ao longo deste trabalho para a solução de equações diferenciais parciais: a implementação manual e a implementação automatizada utilizando a biblioteca `DeepXDE`. Cada abordagem possui características únicas que impactam a eficiência, flexibilidade e acessibilidade das soluções computacionais.

A implementação manual destacou-se por oferecer um controle completo sobre o processo de solução, desde a formulação do modelo matemático até a escolha de algoritmos de otimização. Essa abordagem permitiu explorar detalhadamente como os diferentes componentes interagem, contribuindo para um aprendizado mais profundo dos fundamentos teóricos. No entanto, tal flexibilidade vem acompanhada de desafios, como a maior complexidade de desenvolvimento e o tempo necessário para implementar e testar cada etapa. Além disso, a suscetibilidade a erros foi um fator significativo, especialmente na codificação de derivadas e na escolha de pontos de colocação. A implementação manual, embora poderosa, mostrou-se menos adaptável a novos problemas, exigindo reconfiguração substancial ao aplicar os métodos a outras equações diferenciais parciais.

Por outro lado, a biblioteca `DeepXDE` proporcionou uma abordagem automatizada que simplifica o desenvolvimento de Redes Neurais Guiadas pela Física (PINNs). A definição direta de modelos diferenciais, condições de contorno e funções de perda em uma interface padronizada reduziu significativamente o tempo necessário para configurar os experimentos. A automação de cálculos complexos, como derivadas de alta ordem e implementação de pontos de colocação, não apenas minimizou os erros, mas também permitiu uma adaptação mais ágil a diferentes problemas matemáticos. Contudo, essa abordagem apresenta limitações em termos de flexibilidade, dificultando personalizações avançadas que podem ser necessárias em certos contextos. Além disso, a abstração de etapas críticas pode afastar o pesquisador de uma compreensão mais profunda dos fundamentos matemáticos e tornar a implementação dependente de ferramentas externas, sujeitas a mudanças ou desatualizações.

Um exemplo específico de desafios na utilização de ferramentas automatizadas foi observado durante a resolução das equações KdV e mKdV. Embora os experimentos iniciais tenham demonstrado resultados satisfatórios, acreditamos que o ambiente de desenvolvimento utilizado, o Google Colab, passou por alterações que impactaram a execução do modelo. Essas mudanças, possivelmente relacionadas a atualizações de GPU ou bibliotecas, resultaram em inconsistências na reprodução dos resultados previamente obtidos. Apesar de esforços para fixar sementes aleatórias, reinstalar bibliotecas específicas e reconfigurar o ambiente, as métricas apresentaram variações significativas, inviabilizando a reprodução exata dos experimentos originais. Esse problema ressalta a importância de realizar implementações em ambientes controlados, especialmente ao trabalhar com ferramentas que abstraem etapas críticas do desenvolvimento.

A escolha entre uma abordagem manual ou automatizada depende fortemente dos objetivos do projeto. Para estudos exploratórios ou que exigem alto nível de personalização, a implementação manual é mais indicada, pois permite um controle detalhado e maior liberdade para adaptações. Por outro lado, para projetos aplicados ou que demandam resultados rápidos e de fácil adaptação com diferentes equações, ferramentas como `DeepXDE` oferecem maior eficiência e simplificação.

Em suma, ambas as abordagens são valiosas e complementares. A implementação manual traz consigo a oportunidade de aprofundar o entendimento teórico, enquanto ferramentas automatizadas democratizam o acesso e a aplicação de metodologias avançadas em equações diferenciais parciais. A decisão por qual abordagem seguir deve ser guiada pelas necessidades específicas do problema, considerando as vantagens e limitações de cada método.

# Considerações finais

Neste trabalho, investigamos a aplicação de Redes Neurais Guiadas pela Física (PINNs) como uma abordagem moderna e eficiente para a solução de equações diferenciais parciais (EDPs), com foco em três classes importantes de problemas: a equação de Burgers, a equação de Korteweg-de Vries (KdV) e sua versão modificada, a mKdV. Apresentamos duas implementações distintas: uma manual, construída passo a passo utilizando bibliotecas como TensorFlow e NumPy, e outra automatizada por meio da biblioteca DeepXDE. Ambas as abordagens foram comparadas em termos de desempenho, facilidade de implementação e precisão dos resultados, fornecendo *insights* valiosos sobre as vantagens e limitações de cada uma.

Na implementação manual, realizamos o desenvolvimento completo do modelo, desde a formulação matemática das EDPs até a configuração detalhada da rede neural e da função de perda. Essa abordagem permitiu um controle minucioso sobre cada etapa do treinamento, facilitando ajustes específicos e experimentação. Por outro lado, a implementação automatizada, utilizando o DeepXDE, mostrou-se mais eficiente e acessível, reduzindo significativamente o tempo de desenvolvimento e permitindo a resolução de problemas mais complexos, como o caso paramétrico da mKdV, no qual a amplitude do soliton foi generalizada para um intervalo de valores.

Os resultados obtidos confirmaram a eficácia das PINNs em capturar soluções de EDPs com alta precisão, apresentando erros relativos da ordem de $10^{-4}$ para as equações de burgers e KdV e de $10^{-3}$ para a equação de KdV e mKdV. Além disso, exploramos o impacto de parâmetros críticos como o número de camadas, neurônios, pontos de colocação e taxa de aprendizado, demonstrando a importância de estratégias de ajuste progressivo para otimização do processo de treinamento.

Contudo, enfrentamos desafios específicos, como a reprodutibilidade dos experimentos no ambiente Google Colab, que pode ter passado por atualizações afetando a execução do código e o desempenho das GPUs. Essa limitação ressalta a necessidade de ambientes controlados para experimentos mais robustos e reprodutíveis.

De forma mais ampla, este trabalho contribui significativamente para a disseminação das PINNs, oferecendo uma introdução acessível em língua portuguesa para novos pesquisadores interessados na área. Além disso, estabelecemos uma base sólida para estudos futuros que envolvam problemas mais complexos, como EDPs multidimensionais, condições de contorno irregulares e sistemas acoplados.

Como próximos passos, sugerimos:

- Colisão de ondas solitárias na equação de KdV: Explorar a interação entre múltiplos solitons, um

fenômeno característico da KdV que exige precisão para capturar as propriedades de conservação e a dinâmica não linear.

- Problemas com condições de fronteira livre e móvel: Investigar EDPs onde as fronteiras do domínio são uma incógnita do problema e dependentes do tempo, como ocorre na dinâmica de ondas aquáticas.
- EDPs em dimensões maiores: Ampliar o uso das PINNs para problemas em domínios tridimensionais e em aplicações multidimensionais, como dinâmica de fluidos 3D e propagação de ondas em meios complexos.

Esses problemas representam desafios adicionais tanto em termos computacionais quanto na complexidade da formulação, oferecendo um campo fértil para a evolução das PINNs e a integração de técnicas avançadas, como PINNs adaptativas e métodos híbridos que combinam aprendizado de máquina com métodos numéricos tradicionais.

Por fim, as Redes Neurais Guiadas pela Física representam um avanço promissor no campo de simulação numérica e aprendizado profundo, oferecendo soluções precisas e adaptáveis para problemas científicos e de engenharia que antes eram limitados pelos métodos tradicionais. Com o contínuo avanço tecnológico e o aprimoramento das ferramentas computacionais, espera-se que as PINNs desempenhem um papel cada vez mais central na resolução de problemas que envolvem equações diferenciais, modelagem física e descoberta de novos fenômenos.

# Referências


[1] ABADI, Martín et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. **arXiv preprint arXiv:1603.04467**, 2016.

[2] BELLMAN, Richard. **Adaptive Control Processes: a Guided Tour: R-350**. Rand Corporation, 1961.

[3] BLECHSCHMIDT, Jan; ERNST, Oliver G. Three ways to solve partial differential equations with neural networks—A review. **GAMM-Mitteilungen**, v. 44, n. 2, p. e202100006, 2021.

[4] BRAMBURGER, Jason J. **Data-Driven Methods for Dynamic Systems**. Society for Industrial and Applied Mathematics: Philadelphia, PA. 2024. doi: 10.1137/1.9781611978162

[5] BURGERS, Johannes Martinus. A mathematical model illustrating the theory of turbulence. **Advances in applied mechanics**, v. 1, p. 171-199, 1948.

[6] CANUTO, Claudio et al. **Spectral methods**. Berlin: springer, 2006.

[7] CUSHMAN-ROISIN, Benoit; PRATT, Larry; RALPH, Elise. A general theory for equivalent barotropic thin jets. **Journal of physical oceanography**, v. 23, p. 91-91, 1993.

[8] FARLOW, Stanley J. **Partial differential equations for scientists and engineers**. Courier Corporation, 1993.

[9] GLOROT, Xavier; BENGIO, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In: **Proceedings of the thirteenth international conference on artificial intelligence and statistics**. JMLR Workshop and Conference Proceedings, 2010. p. 249-256.

[10] GONDAR, J. Lopez; CIPOLATTI, Rolci. Iniciação à física matemática. **Modelagem de Processos e Métodos de Solução. Rio de Janeiro, IMPA**, 2016.

[11] GRIFFITHS, David J.; SCHROETER, Darrell F. **Introduction to quantum mechanics**. Cambridge university press, 2019.

[12] GUO, Yanan et al. Solving partial differential equations using deep learning and physical constraints. **Applied Sciences**, v. 10, n. 17, p. 5917, 2020.

[13] HARRIS, Charles R. et al. Array programming with NumPy. **Nature**, v. 585, n. 7825, p. 357-362, 2020.

[14] HASSOUN, M. H. **Fundamentals of Artificial Neural Networks**. The MIT Press, 1995.

[15] HOLTON, James R.; HAKIM, Gregory J. **An introduction to dynamic meteorology**. Academic press, 2013.

[16] INCROPERA, Frank P. et al. **Fundamentals of heat and mass transfer**. New York: Wiley, 1996.

[17] JACKSON, John David. **Classical electrodynamics.** John Wiley & Sons, 2021.

[18] JIN, Xiaowei et al. NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations. **Journal of Computational Physics**, v. 426, p. 109951, 2021.

[19] JOSEPH, Antony. **Investigating seafloors and oceans: From mud volcanoes to giant squid**. Elsevier, 2016.

[20] KAKUTANI, Tsunehiko; ONO, Hiroaki. Weak non-linear hydromagnetic waves in a cold collision-free plasma. **Journal of the physical society of Japan**, v. 26, n. 5, p. 1305-1318, 1969.

[21] KARLIK, Bekir; OLGAC, A. Vehbi. Performance analysis of various activation functions in generalized MLP architectures of neural networks. **International Journal of Artificial Intelligence and Expert Systems**, v. 1, n. 4, p. 111-122, 2011.

[22] KARNIADAKIS, George Em et al. Physics-informed machine learning. **Nature Reviews Physics**, v. 3, n. 6, p. 422-440, 2021.

[23] KEVORKIAN, Jirair. **Partial Differential Equations: Analytical Solution Techniques**. Springer, 1990.

[24] KINGMA, Diederik P. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014.

[25] KNOBEL, Roger. **An introduction to the mathematical theory of waves**. American Mathematical Soc., 2000.

[26] KOMATSU, Teruhisa S.; SASA, Shin-ichi. Kink soliton characterizing traffic congestion. **Physical Review E**, v. 52, n. 5, p. 5574, 1995.

[27] KORTEWEG, Diederik Johannes; DE VRIES, Gustav. XLI. On the change of form of long waves advancing in a rectangular canal, and on a new type of long stationary waves. **The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science**, v. 39, n. 240, p. 422-443, 1895.

[28] LEVEQUE, Randall J.; LEVEQUE, Randall J. **Numerical methods for conservation laws**. Basel: Birkhäuser, 1992.

[29] LEVEQUE, Randall J. **Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems**. Society for Industrial and Applied Mathematics, 2007.

[30] LU, Lu; MENG, Xuhui; MAO, Zhiping; KARNIADAKIS, George Em. DeepXDE: A deep learning library for solving differential equations. **SIAM review**, v. 63, n. 1, p. 208-228, 2021.

[31] MENG, Xuhui et al. PPINN: Parareal physics-informed neural network for time-dependent PDEs. **Computer Methods in Applied Mechanics and Engineering**, v. 370, p. 113250, 2020.

[32] MURRAY, James D. **Mathematical biology: I. An introduction**. Springer Science & Business Media, 2007.

[33] NAGATANI, Takashi. TDGL and MKdV equations for jamming transition in the lattice models of traffic. **Physica A: Statistical Mechanics and Its Applications**, v. 264, n. 3-4, p. 581-592, 1999.

[34] QUARTERONI, Alfio; SACCO, Riccardo; SALERI, Fausto. **Numerical mathematics**. Springer Science & Business Media, 2010.

[35] RAISSI, Maziar; PERDIKARIS, Paris; KARNIADAKIS, George Em. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. **arXiv preprint arXiv:1711.10561**, 2017.

[36] RAISSI, Maziar. Deep hidden physics models: Deep learning of nonlinear partial differential equations. **Journal of Machine Learning Research**, v. 19, n. 25, p. 1-24, 2018.

[37] RAISSI, Maziar; PERDIKARIS, Paris; KARNIADAKIS, George Em. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. **Journal of Computational physics**, v. 378, p. 686-707, 2019.

[38] SAKURAI, Jun John; NAPOLITANO, Jim. **Modern quantum mechanics**. Cambridge University Press, 2020.

[39] SHEN, Jie; TANG, Tao; WANG, Li-Lian. **Spectral methods: algorithms, analysis and applications**. Springer Science & Business Media, 2011.

[40] STEIN, Michael. Large sample properties of simulations using Latin hypercube sampling. **Technometrics**, v. 29, n. 2, p. 143-151, 1987.

[41] TREFETHEN, Lloyd N. **Spectral methods in MATLAB**. Society for industrial and applied mathematics, 2000.

[42] VALLIS, Geoffrey K. **Atmospheric and oceanic fluid dynamics**. Cambridge University Press, 2017.

[43] WHITE, Frank M.; MAJDALANI, Joseph. **Viscous fluid flow**. New York: McGraw-Hill, 2006.

[44] WILMOTT, Paul; HOWISON, Sam; DEWYNNE, Jeff. **The mathematics of financial derivatives: a student introduction**. Cambridge university press, 1995.

[45] ZABUSKY, Norman J. A synergetic approach to problems of nonlinear dispersive wave propagation and interaction. In: **Nonlinear partial differential equations**. Academic Press, 1967. p. 223-258.