UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Programa Institucional de Bolsas de Iniciação Científica (PIBIC)

# Relatório Parcial

# Estudo e implementação de redes neurais hierárquicas de aprendizado profundo para cálculo de interpolações em elementos finitos

Autor: Pedro Azevedo Coelho Carriello Corrêa
Orientador: Dr. Pablo Giovanni Silva Carvalho

São Carlos, SP
Março, 2024

# 1 Introdução e Motivação

A dinâmica dos fluidos é uma área na qual tendem a surgir sistemas de equações bastante complexas e, por isso, desde o início do uso de computadores para simulações, a Dinâmica dos Fluidos Computacional (CFD) se fez uma das suas grandes vertentes de estudo. Atualmente, com uma grande expansão do uso de Inteligência Artificial em diversos setores, naturalmente são aplicados diversos métodos de aprendizado de máquina na área de CFD. Nesse contexto, um dos métodos que se destacam são as Redes Neurais Artificiais, em especial as de *Multi-Layer Perceptron* (SHARMA et al., 2023).

## 1.1 Mudança de direcionamento da pesquisa

No início da pesquisa bibliográfica, o foco era o estudo da implementação de redes neurais para o aprimoramento de solução de sistemas por meio do Método de Diferenças Finitas. Porém, após um estudo inicial desse método (baseado em Langtangen e Linge (2017)) e do Método de Elementos Finitos (a partir de Becker e etc. (1981a)), da formação de um grupo de estudo com professores e alunos de pós-graduação focados na biblioteca `FEniCSx` (que se baseia em elementos finitos), uma decisão foi feita para uma mudança na metodologia do projeto. Por contar com o apoio dos integrantes do grupo de estudos de elementos finitos, e após a constatação de que a mudança de foco no estágio inicial do projeto seria possível, a alteração foi considerada uma decisão sensata pelo aluno e pelo orientador da pesquisa.

Tanto o de diferenças finitas, quanto o de elementos finitos, são métodos de solução numérica extensivamente aplicados e consolidados historicamente no contexto de equações de dinâmica dos fluidos (THOMÉE, 1984) e, no contexto de aprendizado de máquina, as redes neurais artificiais já se provaram como uma ferramenta capaz de aperfeiçoar ambos os métodos, como se pode verificar em Pantidis e Mobasher (2023), Meethal et al. (2023), Le-Duc, Nguyen-Xuan e Lee (2023) para elementos finitos e Tu e Nguyen (2022), Shi et al. (2020) para diferenças finitas. Assim, a mudança da metodologia da pesquisa ainda conserva o uso de métodos já consolidados para soluções numéricas.

# 2 Objetivos

Aqui, o objetivo é utilizar redes neurais do tipo *multi-layer perceptron* para aperfeiçoar métodos de soluções numéricas aplicados em equações de dinâmica dos fluidos e futuramente analisar, até certo escopo, o tempo de execução dos algoritmos, a precisão das soluções encontradas e outros parâmetros relevantes para a escolha do uso do método.

O projeto visa a formulação de um método que utiliza uma rede neural hierárquica para realimentar a localização dos nós de elementos finitos. Essa realimentação viria a

partir do treinamento da rede e, a partir dela, os elementos finitos ficariam mais refinados nas regiões de maior sensibilidade da solução, ou seja, os nós iriam se concentrar nas regiões que a solução mais flutua, de modo a diminuir seu erro.

# 3 Metodologia

Os principais modelos de redes neurais e métodos de aprendizado de máquina foram estudados pelo aluno por meio do curso das disciplinas SCC0230 - Inteligência Artificial e SCC0270 - Redes Neurais e Aprendizado Profundo, ministradas, respectivamente, pela Prof. Solange Oliveira Rezende e Prof. Moacir Antonelli Ponti. Já o método de elementos finitos foi estudado principalmente a partir da leitura de Becker e etc. (1981a).

Já a implementação do algoritmo está sendo em `Python`, principalmente através do uso da biblioteca `PyTorch`. Também estão sendo utilizadas outras bibliotecas auxiliares, como a `MatPlotLib` e `NumPy`.

## 3.1 Redes Neurais *Multi-Layer Perceptron*

Redes neurais de aprendizado profundo são redes que conseguem, em grande parte, superar limitações de modelos lineares *perceptron* incorporando mais camadas. A forma mais natural de fazer isso é alocando camadas conectadas em seguida, de forma que cada camada anterior alimenta a próxima, até gerar a saída do modelo. Essa arquitetura é chamada *multilayer perceptron* (MLP) (ZHANG et al., 2021a).

No geral, redes desse tipo são modelos de aprendizado supervisionado, ou seja, que são treinados a partir de um banco de dados de entradas e suas respectivas saídas esperadas. Porém, nesse projeto foi feita uma modificação em relação ao modelo *perceptron* original, na qual a função de perda (*Loss Function*) utilizada para treinar o modelo não é calculada a partir de um banco de dados, mas sim, recalculando a nova solução em elementos finitos considerando as novas posições dos nós dadas pela rede e retornando o seu erro na equação original. Tal abordagem é mais explicada na Seção 3.4.

## 3.2 Método de Elementos Finitos

Atualmente, a equação genérica governante no domínio $x_0 < x < x_L$ que o algoritmo em desenvolvimento resolveria numericamente tem a seguinte forma:

$$-k\frac{\mathrm{d}u(x)^2}{\mathrm{d}^2x} + c\frac{\mathrm{d}u(x)}{\mathrm{d}x} + bu(x) = f(x) \tag{1}$$

Onde $k$, $c$ e $b$ são coeficientes fixos e $f(x)$ uma função de $x$ dados pela equação que se queira resolver. Considerando condição de borda de Dirichlet onde $u(x_0) = u_0$ e

$u(x_L) = u_L$, e $v(x)$ uma função de teste definida em todo o intervalo, pode-se analisar a Equação 1 pelo ponto de vista do método de elementos finitos e reescrevê-la na forma variacional:

$$\int_{x_0}^{x_L} (ku'v' + cu'v + buv)\mathrm{d}x = \int_{x_0}^{x_L} (fv)\mathrm{d}x \tag{2}$$

Nesse contexto, $v$ e $u$ para satisfazerem corretamente o enunciado devem pertencer ao subespaço $H^1$, que são as funções cujas integrais da Equação 3 convergem.

$$\int_{x_0}^{x_L} \left[(v')^2 + v^2\right] \mathrm{d}x < +\infty \tag{3}$$

Dividi-se, arbitrariamente, o domínio do problema em $N$ nós, com elementos finitos com $h$ de comprimento. Após isso, são construídas uma função de forma $\phi_i$ para cada elemento e que geram uma base para o subespaço $H^h$ de $H^1$: $\{\phi_1, \phi_2, \ldots, \phi_N\}$.

Assim, procura-se uma função $u_h \in H_h$ que pode ser escrita como:

$$u_h(x) = \sum_{j=1}^{N} \alpha_j \phi_j(x) \tag{4}$$

Escrevendo $v_h$ da mesma forma e substituindo na Equação 2, temos:

$$\int_{x_0}^{x_L} (ku_h'v_h' + cu_h'v_h + bu_hv_h)\mathrm{d}x = \int_{x_0}^{x_l} (fv_h)\mathrm{d}x$$

Equivalentemente:

$$\sum_{j=1}^{N} K_{ij}\alpha_j = F_i, \quad i = 1, 2, \ldots N \tag{5}$$

Sendo $\mathbf{K}$ comumente chamada de matriz de rigidez e formada pelos elementos $K_{ij}$ e $\mathbf{F}$, chamado de vetor de carga, formado pelos $F_i$.

$$K_{ij} = \int_{x_0}^{x_L} (k\phi_i'\phi_j' + c\phi_i'\phi_j + b\phi_i\phi_j)\mathrm{d}x \tag{6}$$

$$F_i = \int_{x_0}^{x_L} (f\phi_i)\mathrm{d}x \tag{7}$$

Com $1 \leq i, j \leq N$.

Após resolver a Equação 5 para os coeficientes $\alpha_i$, utilizamos a Equação 4 para obter a aproximação de Galerkin para o problema. Todo esse processo está descrito em Becker e etc. (1981b).

Atualmente, o código do projeto utiliza esse procedimento para encontrar a primeira iteração da aproximação da solução e, após o treinamento da rede, a ideia era que os nós se deslocassem para posições que gerassem soluções melhores e as matrizes $K$ e $F$ fossem recalculadas considerando as novas funções de forma dos nós. Porém, estão ocorrendo problemas com a implementação do código com a biblioteca `PyTorch` e os nós ainda não se delocam.

Algo interessante é que há poucas restrições na escolha para a função utilizada como $\phi_i$, o que torna esse método bastante flexível. Até então, o modelo apenas trabalha com uma função de forma linear, mas há trabalho para futuramente adicionar também a opção de aproximações quadráticas.

A função de forma chapéu é definida de acordo com a Equação 8, considerando $h_i$ o tamanho de cada elemento finito, que agora pode ser variável. A partir dessa equação, é possível fazer uma análise da função nos nós e chegar na conclusão da Equação 9 que, inclusive, é uma restrição para a escolha de função de forma que essa equação se satisfaça.

$$\phi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{h_i}, & \text{para } x_{i-1} \leq x \leq x_i \\ \dfrac{x_{i+1} - x}{h_{i+1}}, & \text{para } x_i \leq x \leq x_{i+1} \\ 0, & \text{para } x \leq x_{i-1} \text{e } x \geq x_{i+1} \end{cases} \tag{8}$$

$$\phi_i(x_j) = \begin{cases} 1, & \text{se } i = j \\ 0, & \text{se } i \neq j \end{cases} \tag{9}$$

Assim, pode concluir, a partir da Equação 4, que $u_I \equiv u(x_i) = \alpha_i$ e, assim, pode-se escrever a Equação 4 como a 10.

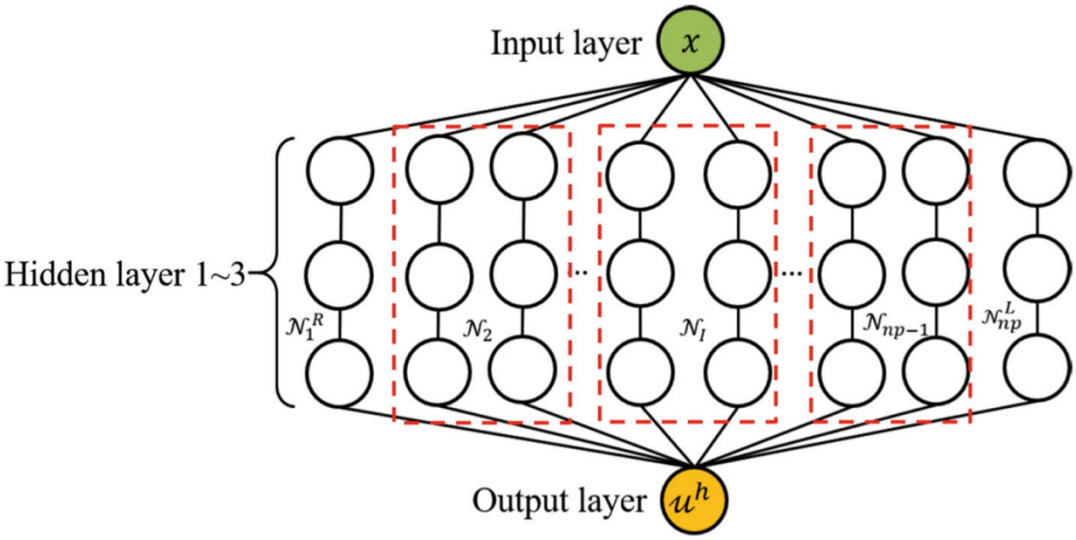
$$u_h = \sum_{i=1}^{N} u_I \phi_i(x) \tag{10}$$

## 3.3 Redes Neurais de Aprendizado Profundo Hierárquicas (HiDeNN)

Inspirado em Saha et al. (2021) e Zhang et al. (2021b), são implementadas redes neurais para cada nó e estas são colocadas em paralelo, assim como ilustrado na Figura 1. As redes de cada nó são as $\mathcal{N}_1^R, \mathcal{N}_2, \ldots, \mathcal{N}_I, \ldots, \mathcal{N}_{np-1}, \mathcal{N}_{np}^L$, sendo $np$ o número de nós que discretiza o domínio.

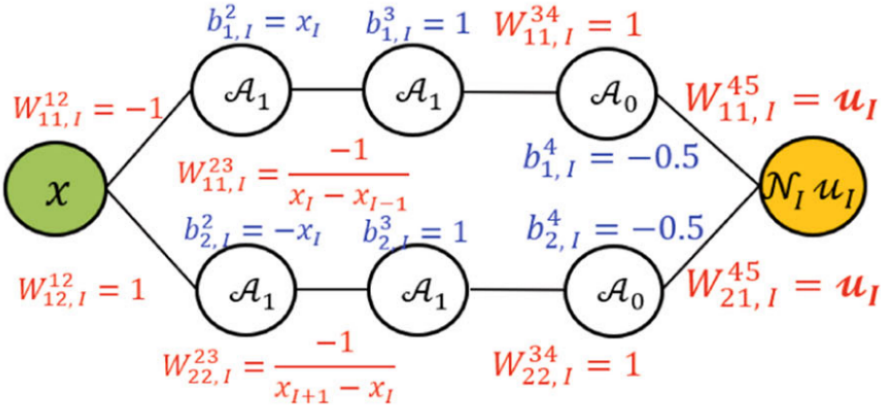
Os nós do interior possuem as camadas ilustradas na Figura 2. Vale ressaltar que os parâmetros de peso $W_{11,I}^{12}$, $W_{12,I}^{12}$, $W_{11,I}^{34}$ e $W_{22,I}^{34}$, e os parâmetros de viés $b_{1,I}^3$, $b_{2,I}^3$, $b_{1,I}^4$ e $b_{2,I}^4$ são fixos. Já os outros pesos e viéses variam de acordo com as coordenadas dos nós e o valor da solução calculada para o nó ($u_I$).

Figura 1 – Estrutura básica da rede neural de aprendizado profundo hierárquica do projeto




Fonte: Zhang et al. (2021b)

Figura 2 – Estrutura da rede neural, $\mathcal{N}_I$, de um nó no interior do domínio




Fonte: Zhang et al. (2021b)

Na Figura 2, $\mathcal{A}_0$ e $\mathcal{A}_1$ representam as funções de ativação de cada camada. A primeira, $\mathcal{A}_0(x) = x$, já $\mathcal{A}_1$ é uma função comumente chamada no campo de redes neurais de `ReLU`, sendo definida como o Listing 1.

Listing 1 – Definição da função `ReLU`

```
1    def ReLU(x):
2        return max(0, x)
```

As camadas ocultas da rede $\mathcal{N}_I$ podem ser interpretadas como geradoras da função de forma chapéu $\phi_i(x)$. Já a última camada realiza a multiplicação pelo termo $u_I$, o que, após a combinação de todas as redes hierárquicas, é análoga à interpolação das funções de forma, ou seja, a Equação 10.

## 3.4 Cálculo da perda para treinamento do modelo

Para treinar a rede neural hierárquica é necessário declarar uma função de perda que precisa, no decorrer do treinamento, ser minimizada. A rotina em desenvolvimento para a geração da rede e seu treinamento segue os seguinte passos:

1. É inserida uma equação no formato $-k \cdot u''(x) + c \cdot u'(x) + b \cdot u(x) = f(x)$, o domínio onde ela governa, e as condições de contorno de Dirichlet, i.e., os valores de $u(x)|_{x=x_0}$ e $u(x)|_{x=x_L}$.

2. O domínio é repartido em $N$ nós e $N-1$ elementos finitos de tamanhos iguais de $h$ de comprimento.

3. A partir de uma dada função de forma (seja chapéu ou quadrática) e a equação dada pelo problema, calcula-se as matrizes $\mathbf{K}$ e $\mathbf{F}$ baseadas no método de elementos finitos, exposto na Seção 3.2.

4. Para encontrar os coeficientes interpoladores das funções de forma, faz-se $\mathbf{K}\{u_I\} = \mathbf{F}$ e resolve para $\{u_I\}$, que é o vetor composto pelos coeficientes $u_I$.

5. Tendo a inicialização dos nós e coeficientes interpoladores feita, forma-se uma rede hierárquica, assim como exposta na Seção 3.3.

6. Para cálculo da perda:

   a) Utilizando a rede gerada, realiza-se um primeiro *forward* pela rede, ou seja, dado o domínio como entrada para a rede, retorna a sua saída, que é a primeira iteração de solução aproximada.

   b) A partir desse solução, são aproximados os valores de $u'$ e $u''$ e a função de perda é dada por $l(x, \theta) = \text{MSE}(-ku'' + cu' + bu, f)$, sendo $\theta$ o conjunto dos parâmetros atuais da rede e MSE a função de erro quadrático médio (*Mean Squared Error*). Ou seja, a perda é calculada pelo resíduo da equação original utilizando a solução aproximada e com base no erro quadrático médio.

7. A partir da função de perda, utiliza-se um método do estado da arte (como o `AdamW`) para calcular o *backpropagation* da rede e recalcular o valor dos nós dos elementos finitos.

8. Com esses novos nós, realiza novamente o passo 3 em diante até um certo número de iterações ou até uma perda satisfatória.

# 4 Atividades e Resultados

Como inicialmente a conceito do projeto era outro, a primeira atividade realizada foi o estudo das teorias que envolviam aquela ideia. Assim, foi estudado o método das diferenças finitas por meio da leitura Langtangen e Linge (2017), e, concomitantemente, o conceito de mínimos quadrados móveis e suas aplicações em diferenças finitas, como em Sousa et al. (2019). Durante o estudo dessas fontes, foram implementados códigos em *Python* para melhor entender como funciona esses métodos.

Após a decisão de mudança no projeto, os novos estudos foram acompanhados com frequentes reuniões com o orientador da pesquisa e com o grupo de estudos referentes à biblioteca `FEniCSx`. Durante os novos estudo também foram implementados códigos. Dentre esses novos códigos, vale mencionar:

- Realização de algumas lições de *jupyter notebooks* disponibilizados por Baratta (2023) para o exercício da biblioteca `FEniCSx`.
- Implementação de códigos para geração de aproximações por elementos finitos de equações de 1 dimensão.
- Primeira versão do código com aprimoramento pela rede hierárquica de aprendizado profundo.

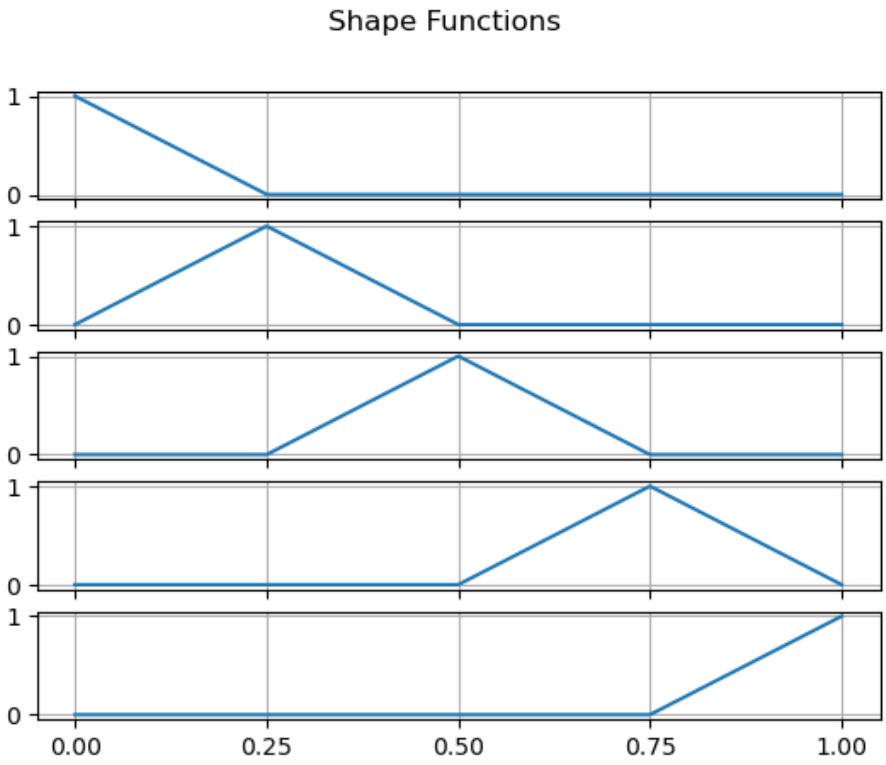
## 4.1 Códigos para implementações de elementos finitos para 1 dimensão

Foram elaborados códigos com um baixo uso de bibliotecas, para forçar a implementação do método de elementos finitos quase do zero. Nesses códigos, destaca-se um que recebe como entrada a equação do problema, o domínio, as condições de contorno, a quantidade de nós e o grau da função de forma e ele retorna a aproximação calculada. Tal código pode ser chamado para resolver o problema de Poisson com função de forma chapéu como exibido no Listing 2.

Listing 2 – Chamada para resolução do problema de Poisson

```
3     equation = [1, 0, 0, lambda x: 1] # [k, c, b, f(x)]
4     bound_cond = [0, 0] # condicoes de borda
5     dom = np.linspace(0,1,500) # dominio
6
7     u_aprox = FE1D_constructor(dom=dom,
8                                equation=equation,
9                                n=5,
10                               k=1,
11                               bound_cond=bound_cond)
```

Figura 3 – Funções de forma geradas pelo código do Listing 2




Fonte: Elaborada pelo autor

Com essa entrada, o algoritmo é capaz de retornar as matrizes de rigidez e de carga do modelo, além de imprimir o gráfico das funções de forma e da comparação com a solução exata (Figura 3 e 4).

Com esse mesmo código foram geradas soluções aproximadas para outros problemas utilizando outras funções de forma, como demonstrado nas Figuras 5 e 6.

## 4.2 Versão 1 da rede neural de aprendizado profundo hierárquica

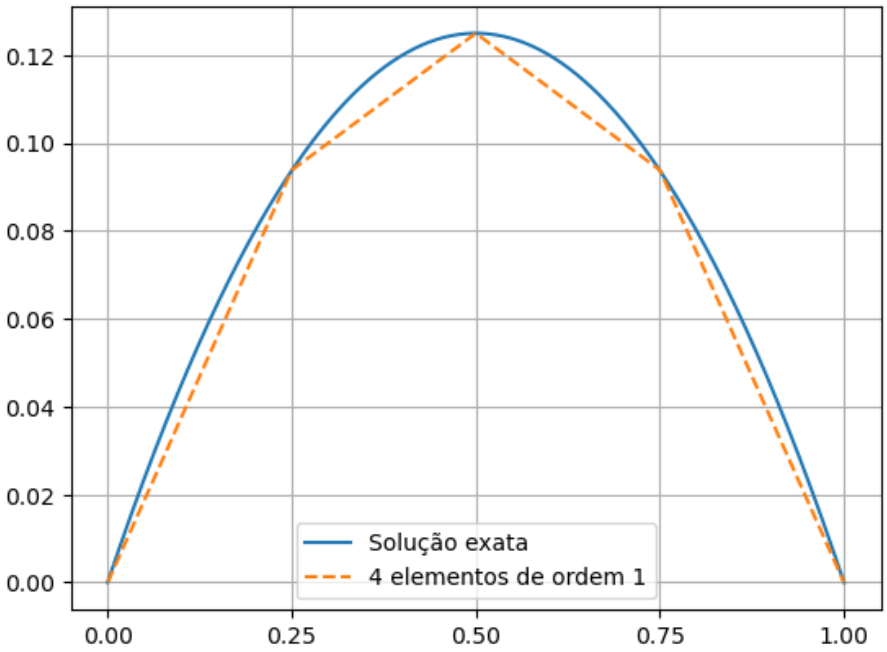
O algoritmo principal do projeto está sendo feito de forma orientada a objeto. Há uma classe principal chamada `HiDeNN_for_FEM`, responsável por, entre outras rotinas, chamar as redes hierárquicas e unir-las de forma paralela. Além disso, essa classe executa o algorimo de cálculo da função de perda e treinamento das redes exposto na Seção 3.4.

Quanto ao treinamento da rede, isso pode ser resumido pela função `train` exibida em parte no Listing 3.

Listing 3 – Parte da definição da função `train`

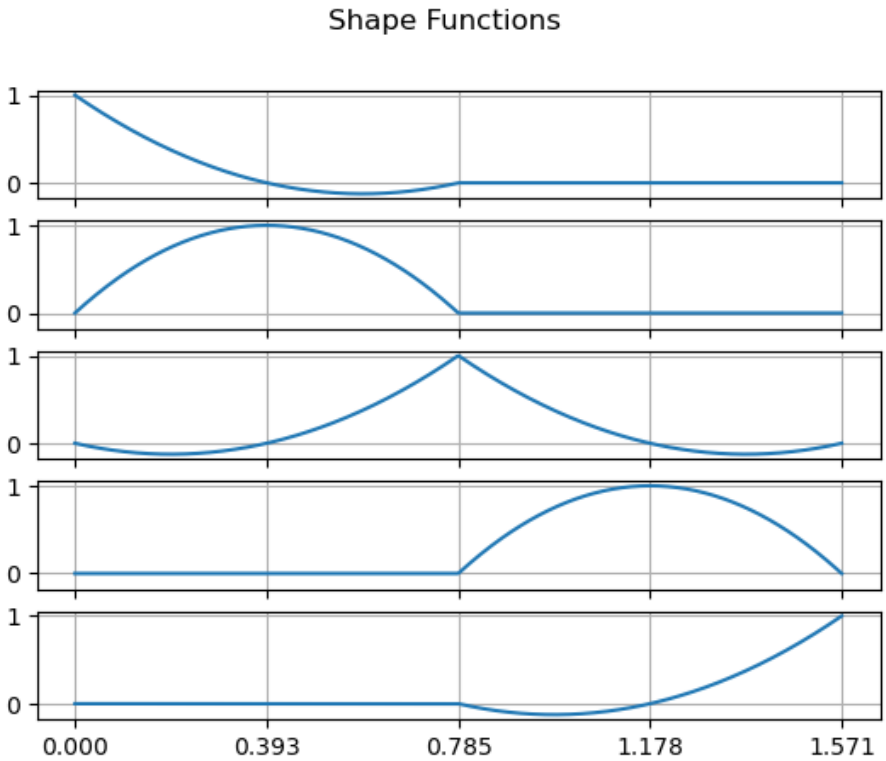
```
1     def train(self, epochs=1, lossfunc=nn.MSELoss(), lr=1e-3):  #
    utilizando um learning rate padrao de 0.001
2             for epoch in range(epochs):
3                 # -k * u''(x) + c * u'(x) + b * u(x) = f(x)
```

Figura 4 – Comparação geradas pelo código do Listing 2 com a solução exata



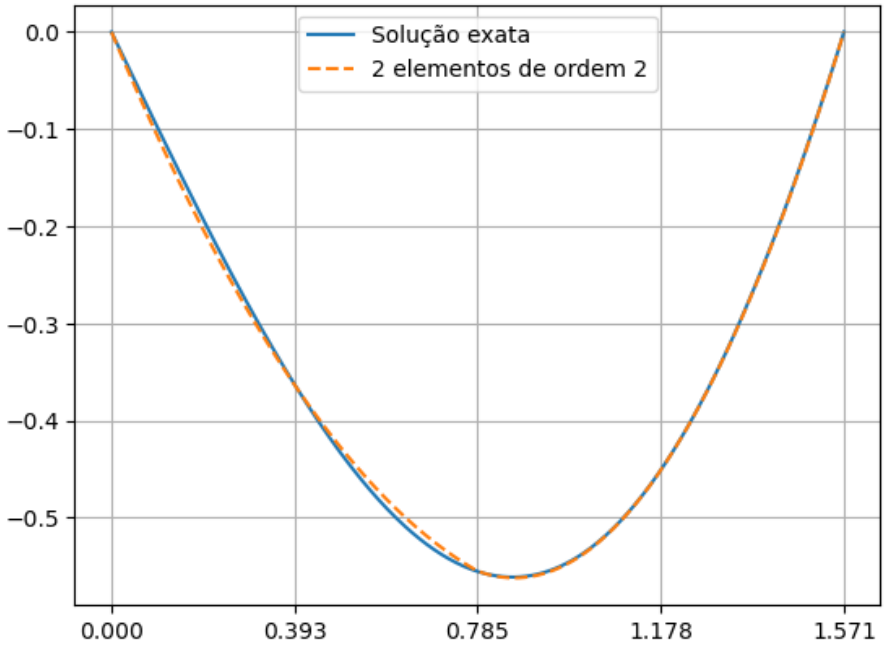

Fonte: Elaborada pelo autor

Figura 5 – Funções de forma de grau 2 geradas para o problema $u'' + u = 2\sin(x)$




Fonte: Elaborada pelo autor

Figura 6 – Comparação da solução aproximada com a exata para o problema $u'' + u = 2\sin(x)$




Fonte: Elaborada pelo autor

```
4
5                u = self.forward().reshape(-1) # saida do foward da rede
    eh interpretada como u
6                dudx = diff(u, self.dx) # derivada de u
7                ddudx2 = diff(dudx, self.dx) # derivada segunda de u
8
9                self.optimizer = torch.optim.SGD([self.nodes], lr=lr) #
   utilizando otimizador de descida do gradiente estocastico
10               self.optimizer.zero_grad() # zerando gradiente
11               loss = lossfunc(
12                                ( -self.k * ddudx2 + self.c * dudx[1:] +
    self.b * u[2:] ).requires_grad_(True),
13                                tensor(self.f_func(self.dom[0,2:]))
14                                ) # calculo da funcao de perda
15               loss.backward() # calculando backpropagation
16               self.optimizer.step() # alterando os parametros da rede
17
18               self.shape_func_arr = self.node_nets_func(torch.ones((
   self.num_nodes,1), dtype=torch.float64)) # gerando novas funcoes de
   forma com base nas novas posicoes dos nos
19
20               self.displacement_arr = solve(self.K,self.F) # alpha = K
   ^-1 x F # solucionando coeficientes interpoladores de cada funcao de
   forma
21               self.u_aprox_arr = self.displacement_arr @ self.
```

```
    shape_func_arr # recalculando solucao aproximada do problema
22
23              self.node_nets_arr = self.node_nets_func(self.
    displacement_arr) # gerando novas redes hierarquicas com base novos
    nos e novos coeficientes interpoladores
24
25          return loss # retorna valor da perda final
```

Os métodos `optim.SGD`, `backward`, `step`, `zero_grad`, `MSELoss`, dentre outros utilizados no código, fazem parte da biblioteca `PyTorch`, que é um módulo focado em redes neurais implementada em *Python*.

O projeto, ainda em desenvolvimento, passa por algumas complicações. Atualmente, a inicialização das redes para a formação da aproximação numérica para a solução utilizando funções chapéu funciona. Porém, durante o treinamento, a rede ainda não está sendo capaz de diminuir a sua perda pois, por algum motivo ainda não descoberto, os nós ainda não estão se deslocando em busca de um posicionamento ótimo.

Algumas tentativas já foram feitas para a correção do problema, vale citar:

- Alterações de várias das funções utilizadas de bibliotecas diferentes para outras funções análogas da biblioteca `PyTorch`, em busca de uniformizar o código e evitar incompatibilidades.
- Diminuição do número de objetos da classe `torch.tensor` a fim de não ocorrer o problema dos seus gradientes se perderem na execução do *backpropagation* (como em casos de *vanishing gradient*)
- Implementação das camadas da rede sem o uso das bibliotecas de redes neurais a fim de melhor entender o funcionamento do algoritmo.
- Reestruturações do código a fim de torná-lo mais simples e analisável.

Apesar das diferentes abordagens, até então o problema ainda não foi resolvido.

# 5 Próximas Etapas

A questão do treinamento da rede está gerando um travamento na pesquisa. Assim, sobre o futuro do projeto, o principal foco é a resolução desse problema. Além de mais testes no modelo a fim de identificar a causa do problema, uma ação que vai ser feita com o mesmo objetivo é procurar professores especialistas na área de redes neurais para marcar reuniões em busca de novos pontos de vista.

Uma outra ideia a ser implementada no futuro é a utilização da rede não só para recalcular a posição dos nós, mas também para encontrar os coeficientes interpoladores

ótimos. Nesse caso, o método de elementos finitos seria utilizado apenas para inicializar o modelo e gerar a primeira iteração da solução e, a partir desse ponto, a propria rede hierárquica chegaria nos melhores parâmetros. O que é um experimento interessante para fazer um comparativo dos métodos.

Assim que o problema do treinamento for corrigido, a próxima etapa será o teste do algoritmo em vários problemas a fim de analisar aspectos da precisão e eficiência da rede neural em comparativo com os elementos finitos.

Após um aprofundamento no estudo da biblioteca `FEniCSx` e atingir uma estabilidade do algoritmo da rede, pode ser possível a implementação do código em problemas de duas dimensões com o auxílio da biblioteca.

Finalmente, a última modificação que pode se tornar plauzível após a readaptação do código à nova biblioteca, seria a utilização de funções de forma quadráticas para a aproximação da solução.

# Referências


BARATTA, I. **FEniCSxCourse at ICMC23**. 2023. <https://github.com/IgorBaratta/FEniCSxCourse>. Accessed: 2024-3-11.

BECKER, E. B.; etc. **Finite Elements: v. 1: An Introduction**. Harlow, England: Longman Higher Education, 1981.

_____. _____. Harlow, England: Longman Higher Education, 1981. 40-59 p.

LANGTANGEN, H. P.; LINGE, S. **Finite difference computing with PDEs: A modern software approach**. 1. ed. Basel, Switzerland: Springer International Publishing, 2017.

LE-DUC, T.; NGUYEN-XUAN, H.; LEE, J. A finite-element-informed neural network for parametric simulation in structural mechanics. **Finite Elem. Anal. Des.**, v. 217, n. 103904, p. 103904, 2023.

MEETHAL, R. E.; KODAKKAL, A.; KHALIL, M.; GHANTASALA, A.; OBST, B.; BLETZINGER, K.-U.; WÜCHNER, R. Finite element method-enhanced neural network for forward and inverse problems. **Adv. Model. Simul. Eng. Sci.**, v. 10, n. 1, 2023.

PANTIDIS, P.; MOBASHER, M. E. Integrated finite element neural network (I-FENN) for non-local continuum damage mechanics. **Comput. Methods Appl. Mech. Eng.**, v. 404, n. 115766, p. 115766, 2023.

SAHA, S.; GAN, Z.; CHENG, L.; GAO, J.; KAFKA, O. L.; XIE, X.; LI, H.; TAJDARI, M.; KIM, H. A.; LIU, W. K. Hierarchical deep learning neural network (HiDeNN): An artificial intelligence (AI) framework for computational science and engineering. **Comput. Methods Appl. Mech. Eng.**, v. 373, n. 113452, p. 113452, 2021.

SHARMA, P.; CHUNG, W. T.; AKOUSH, B.; IHME, M. A review of physics-informed machine learning in fluid mechanics. **Energies**, v. 16, n. 5, p. 2343, 2023.

SHI, Z.; GULGEC, N. S.; BERAHAS, A. S.; PAKZAD, S. N.; TAKAC, M. Finite difference neural networks: Fast prediction of partial differential equations. In: **2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)**. [S.l.]: IEEE, 2020. p. 130–135.

SOUSA, F. S.; LAGES, C. F.; ANSONI, J. L.; CASTELO, A.; SIMAO, A. A finite difference method with meshless interpolation for incompressible flows in non-graded tree-based grids. **J. Comput. Phys.**, v. 396, p. 848–866, 2019.

THOMÉE, V. The finite difference versus the finite element method for the solution of boundary value problems. **Bull. Aust. Math. Soc.**, v. 29, n. 2, p. 267–288, 1984.

TU, S. N. T.; NGUYEN, T. FinNet: Solving time-independent differential equations with finite difference neural network. 2022.

ZHANG, A.; LIPTON, Z. C.; LI, M.; SMOLA, A. J. Dive into deep learning. p. 170–186, 2021.

ZHANG, L.; CHENG, L.; LI, H.; GAO, J.; YU, C.; DOMEL, R.; YANG, Y.; TANG, S.; LIU, W. K. Hierarchical deep-learning neural networks: finite elements and beyond. **Comput. Mech.**, v. 67, n. 1, p. 207–230, 2021.