



UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM
GEOFÍSICA

Tópicos em Métodos Numéricos
Método de Resíduos Ponderados

MARCELO LUCAS ALMEIDA

`marcelolucasif@gmail.com`

Belém - Pará

20 de Janeiro de 2025

Sumário

1	Introdução	2
2	Método de Resíduos Ponderados	4
2.1	Método da Colocação - Problema Unidimensional	4
2.2	Método de Mínimos Quadrados (MMQ)	13
2.3	Método de Galerkin	19

1 Introdução

O estudo das equações diferenciais é uma etapa fundamental na formação de cientistas e engenheiros. Por meio da formulação de princípios e leis físicas, diversos fenômenos naturais podem ser descritos matematicamente, e essa descrição muitas vezes conduz à formulação de equações diferenciais, sejam elas ordinárias ou parciais. Compreender esses tipos de equações e saber como resolvê-las é essencial para interpretar e modelar o mundo ao nosso redor.

Neste contexto, esta apostila tem como objetivo introduzir o estudante a algumas técnicas numéricas para a resolução de equações diferenciais com problema de valor de contorno, com foco nos métodos de resíduos ponderados, que constituem uma família de técnicas aproximadas para esse tipo de problema.

Apresentaremos os principais métodos desse grupo, como o Método de Colocação, o Método dos Mínimos Quadrados e o Método de Galerkin. Daremos ênfase especial a este último, por ser um ponto de partida importante para a compreensão de métodos mais avançados, como o Método dos Elementos Finitos (MEF).

Serão apresentadas as formulações matemáticas de cada método, sem, no entanto, exigir rigor formal nos conceitos teóricos envolvidos, de modo a facilitar a compreensão por parte do aluno.

Após a formulação matemática de cada técnica, incluiremos exemplos práticos com suporte computacional em linguagem Python, que tem se mostrado uma excelente escolha para aplicações em computação numérica e científica.

Como o foco desta apostila está nas técnicas numéricas, e não na linguagem de programação em si, será assumido que o leitor possui conhecimentos básicos de programação em Python. Para os que ainda não têm familiaridade com a linguagem, recomendamos as seguintes obras introdutórias: [7] e [10].

Para aqueles que desejam aprofundar seus conhecimentos em computação numérica com as bibliotecas NumPy, SciPy, Matplotlib e Pandas, recomendamos o livro de [5], que oferece uma excelente abordagem prática.

A ideia principal usada no método de resíduos ponderados é substituir a nossa função incógnita $u(x)$ da nossa equação diferencial por um conjunto de funções linearmente independentes que aproximam a solução da nossa equação diferencial e minimizar o *resíduo* em um sentido ponderado.

Para entender com mais clareza o parágrafo anterior, partimos de um problema piloto simples, porém, didático do ponto de vista da matemática.

Seja uma equação diferencial ordinária de 2 ordem linear, de coeficientes não constante e não homogênea sobre algum domínio Ω da reta real \mathbb{R} :

$$a(x)\frac{d^2u}{dx^2} + b(x)\frac{du}{dx} + c(x)u(x) = h(x) \quad \text{com } u(a) = u_0, u(b) = u_1 \quad (1.1)$$

Com $a \leq x \leq b$. Para obtermos a expressão que caracteriza os métodos de ponderação de resíduos, trocamos $u(x)$ por um conjunto de funções linearmente independentes e que satisfazem as condições de contorno do problema [11].

$$u(x) \approx u_n(x) \quad (1.2)$$

Onde $u_n(x)$ é a nossa solução aproximada do problema. Substituindo (1.2) em (1.1) não teremos mais uma igual, mas sim, um *resíduo* dado por :

$$R(x) = a(x)\frac{d^2u_n}{dx^2} + b(x)\frac{du_n}{dx} + c(x)u_n(x) - h(x) \neq 0 \quad (1.3)$$

A solução numérica do problema é obtida minimizando esse resíduo em um sentido ponderado :

$$\int_a^b R(x)w(x)dx = 0 \quad (1.4)$$

Onde $w(x)$ é a nossa *função peso*.

A forma como definimos $u_n(x)$ e a função $w(x)$ definem uma técnica específica dos métodos de resíduos ponderados. Adiante veremos as principais técnicas usadas.

Observação

A ideia mostrada até o momento são facilmente estendidas para resolver de forma numérica as equações diferenciais em várias variáveis, ou seja, as Equações Diferenciais Parciais [11], [4].

2 Método de Resíduos Ponderados

2.1 Método da Colocação - Problema Unidimensional

O método da Colocação é uma técnica pertencente aos métodos de ponderação de resíduos e é usada para se obter a solução aproximada de uma equação diferencial com condições de contorno, sendo ordinária ou parcial. Focaremos em desenvolver esse método aproximado para o caso mais simplório, ou seja, aplicando à equações diferenciais ordinárias.

Consideremos o caso onde temos as condições de *contorno do tipo Dirichlet Homogênea*, ou seja :

$$u(a) = u(b) = 0 \quad (2.1)$$

Partindo de uma equação diferencial de 2 ordem não homogênea :

$$a(x) \frac{d^2 u}{dx^2} + b(x) \frac{du}{dx} + c(x)u(x) = h(x) \quad (2.2)$$

O resíduo será dado por :

$$R(x) = a(x) \frac{d^2 u_n}{dx^2} + b(x) \frac{du_n}{dx} + c(x)u_n(x) - h(x) \quad (2.3)$$

Minimizando esse resíduo em um sentido ponderado, teremos :

$$\int_a^b R(x)w(x)dx = 0 \quad (2.4)$$

Agora especificamos a função de peso $w(x)$, que no método da colocação é a *função generalizada de Dirac* [1].

$$\int_a^b R(x)\delta(x - x_i)dx = 0 \quad (2.5)$$

A equação (2.5) acima só será igual à zero, se o resíduo $R(x)$ for calculado nos pontos x_i .

$$R(x_i) = 0 \quad (2.6)$$

Os pontos x_i são conhecidos como *pontos de colocação* e podem ser gerados de várias maneiras ao longo do intervalo $[a, b]$: *pontos uniformemente espaçados*, *Polinômios ortogonais* e etc.

Como o resíduo é determinado pela equação (2.3), as funções de aproximação¹ $u_n(x)$ são um conjunto linearmente independentes que satisfazem as condições de contorno, ou seja :

¹Também chamadas de *funções de base globais*.

$$u_n(x) = \sum_{j=1}^n c_j N_j(x) \quad (2.7)$$

Com

$$N_j(a) = N_j(b) = 0 \quad \text{para } j = 1, 2, \dots, n. \quad (2.8)$$

O sistema linear que é formado através da equação (2.6) depende da quantidade de pontos de colocação x_i . No caso de usarmos 2 pontos (x_1, x_2) , temos :

$$\begin{bmatrix} \mathcal{L}(N_1(x_1)) & \mathcal{L}(N_2(x_1)) \\ \mathcal{L}(N_1(x_2)) & \mathcal{L}(N_2(x_2)) \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} h(x_1) \\ h(x_2) \end{bmatrix} \quad (2.9)$$

$$\mathbf{K}\mathbf{c} = \mathbf{b} \quad (2.10)$$

Onde temos que o símbolo \mathcal{L} representa um operador diferencial de 2 ordem, que nesse caso em específico vale :

$$\mathcal{L}(N_1(x_1)) = a(x_1) \frac{d^2 N_1}{dx^2}(x_1) + b(x_1) \frac{dN_1}{dx}(x_1) + c(x_1) N_1(x_1)$$

$$\mathcal{L}(N_2(x_1)) = a(x_1) \frac{d^2 N_2}{dx^2}(x_1) + b(x_1) \frac{dN_2}{dx}(x_1) + c(x_1) N_2(x_1)$$

$$\mathcal{L}(N_1(x_2)) = a(x_2) \frac{d^2 N_1}{dx^2}(x_2) + b(x_2) \frac{dN_1}{dx}(x_2) + c(x_2) N_1(x_2)$$

$$\mathcal{L}(N_2(x_2)) = a(x_2) \frac{d^2 N_2}{dx^2}(x_2) + b(x_2) \frac{dN_2}{dx}(x_2) + c(x_2) N_2(x_2)$$

E assim por diante. Agora veremos como podemos aplicar o método a um problema de valor de contorno simples. Para isso, sempre iremos considerar no momento apenas condições de contorno homogêneas².

Seja a equação diferencial de 2 ordem :

$$\frac{d^2 u}{dx^2} + u = x \quad \text{com} \quad u(0) = 0, u(1) = 0$$

Mediantes as técnicas aprendidas nos curso introdutórios de equações diferenciais³, obtemos a solução analítica da EDO acima, dada por :

²Quando estudarmos o método de Galerkin, veremos como podemos usar esse método para resolver condições de contorno não homogêneas.

³Para maiores detalhes sobre a teoria das equações diferenciais ordinária e parciais, consulte as seguinte referências : [12], [2] e [6]

$$u(x) = x - \frac{\sin(x)}{\sin(1)}$$

Esse resultado pode ser facilmente obtido em python usando o módulo *SymPy*⁴, que serve como uma *SAC* (Sistema de Computação Algébrica), que manipula símbolos de forma exata, similarmente como se estivessemos resolvendo o problema à mão pela matemática.

O seguinte código obtém a solução analítica da EDO do exemplo acima

Código Python

```

1  """
2  =====
3  descrição : Solução da Equação Diferencial usando o SymPy
4  data : 05/01/25
5  programador : Marcelo L. Almeida
6  =====
7  """
8  # modulos usados :
9  import numpy as np
10 import scipy as sp
11 from sympy import (symbols, diff, dsolve, Eq, init_printing,
12                    lambdify, sin, cos, integrate, Function)
13 import matplotlib.pyplot as plt
14 # deixando a saída simbolica em latex :
15 init_printing(use_latex= True)
16
17 ## ===== codigo para resolução da edo ===== ##
18 # criando os símbolos
19 x = symbols("x", real = True)
20 u = Function("u")(x)
21
22 # criando a equação diferencial
23 eq_edo = Eq(diff(u,x,2) + u, x)
24
25 # criando as condições de contorno como dicionarios
26 pvc = {u.subs(x,0) : 0,
27        u.subs(x,1) : 0}
28
29 # resolvendo a equação diferencial :
30 sol_analitic = dsolve(eq_edo,u, ics=pvc)
31 print(f"Solução Analítica: \n")
32

```

⁴Para aprender sobre esse magnífico pacote, consulte o seguinte endereço de sua documentação : [SymPy](#)

33 `print(sol_analitic)`

Ao executar o código acima, a saída que represente a nossa solução analítica é :

Saída do Código

$$u(x) = x - \frac{\sin(x)}{\sin(1)}$$

Onde podemos observar que a solução obtida em Python bate com a nossa solução em (2.1).

Agora que estamos de posse da solução analítica, nos resta aplicar o método da colocação para aproximar a solução exata.

Antes disso, precisamos definir duas coisas importantes : *quais funções de base global usar ? e quais pontos de colocação x_i usar no intervalo de $[0, 1]$?*. Para responder a primeira pergunta, devemos considerar o seguinte Teorema

Teorema das Funções de Bases Globais

Para que as nossas funções de base aproximem de forma satisfatória a nossa solução analítica, elas devem satisfazer os seguintes itens :

1. **Propriedade da Completude** : As funções de aproximação devem representar de forma exata a solução analítica quando $n \rightarrow \infty$. isso significa satisfazer :

$$\lim_{n \rightarrow \infty} \left\| u(x) - \sum_{j=1}^n c_j \phi_j(x) \right\| = 0$$

2. **Linearidade** : As funções de base devem formar um conjunto de funções linearmente independentes dentro do intervalo de definição da nossa EDO/EDP.
3. **Satisfazer as Condições de Contorno** : O conjunto de funções L.I devem satisfazer todas as condições de contorno do problema.
4. **Diferenciabilidade** : Se a nossa EDO/EDP contiver derivadas de ordem k , então as nossas funções de base devem ser no mínimo de classe \mathcal{C}^k (contínuas e diferenciáveis até a ordem k).

Mediante a esses fatos, podemos escolher o nosso conjunto por :

$$\phi_j(x) = x^j(1-x)$$

que para o nosso problema diferencial, satisfaz todas as condições de contorno :

$$\phi_j(0) = \phi_j(1) = 0 \quad \text{para } j = 1, 2, \dots, n.$$

Agora por último, precisamos decidir como gerar os pontos de colocação, que determinará o tamanho do sistema linear que será resolvido para obter a solução numérica do problema. Uma das forma mais eficientes de gerar os pontos de colocação para o método é usando os pontos de *chebyshev*.

Esse pontos são obtidos calculando as raízes dos polinômios de *chebyshev* $T_n(x)$, que são usualmentes usados em técnicas de quaadradura numérica ([9]) , [3] e [8].

Esse pontos são obtidos para um intervalo de $[-1, 1]$, mas usando a fórmula a seguir podemos gerar para um intervalo $[a, b]$.

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos \left(\frac{(2i-1)\pi}{2N} \right) \quad (2.11)$$

Agora que temos todas as ferramentas necessárias para aplicar o método aproximado, um exemplo de código a seguir é usado para computar a solução aproximada para o problema diferencial acima. Ao final do código, é plotado na mesma figura a solução analítica e numérica pelo método da colocação.

No código a seguir usaremos o Sympy para realizar os cálculos matemáticos exigidos no método de aproximação. Como se trata de um exemplo relativamente simples, podemos usar o paradigma simbólico para computar as contas de forma exata, entretanto, para problemas mais desafiadores, a computação simbólica pode ser ineficiente e extremamente lenta, portanto , analisar o problema em questão é fundamental.

Polinômios de Chebyshev

Os polinômios de *Chebyshev* formam um conjunto ortogonal de funções com aplicações em teoria da aproximação e análise numérica. Existem dois tipos de polinômios de Chebyshev : **1º Tipo** ($T_n(x)$) e **2º Tipo** ($U_n(x)$).

Polinômios de Chebyshev de Primeiro Tipo:

Definidos por:

$$\begin{aligned} T_0(x) &= 1, & T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad (n \geq 1) \end{aligned}$$

Propriedades: Ortogonais em $[-1, 1]$ com peso $w(x) = \frac{1}{\sqrt{1-x^2}}$:

$$\int_{-1}^1 \frac{T_m(x)T_n(x)}{\sqrt{1-x^2}} dx = \begin{cases} 0 & n \neq m \\ \pi & n = m = 0 \\ \frac{\pi}{2} & n = m \neq 0 \end{cases}$$

Polinômios de Chebyshev de Segundo Tipo:

Definidos por:

$$\begin{aligned} U_0(x) &= 1, & U_1(x) &= 2x \\ U_{n+1}(x) &= 2xU_n(x) - U_{n-1}(x) \quad (n \geq 1) \end{aligned}$$

Propriedades: Ortogonais em $[-1, 1]$ com peso $w(x) = \sqrt{1-x^2}$:

$$\int_{-1}^1 T_m(x)T_n(x)\sqrt{1-x^2}dx = \begin{cases} 0 & n \neq m \\ \frac{\pi}{2} & n = m \neq 0 \end{cases}$$

Código Python

```
1  """
2  =====
3  descrição : Solução da Equação Diferencial usando o
4  método da colocação
5  data : 05/01/25
6  programador : Marcelo L. Almeida
7  =====
8  """
9  import numpy as np
10 import scipy as sp
11 from sympy import (symbols, diff, dsolve, Eq, init_printing,
    lambdify,
```

```

12         sin,cos, integrate, Function, Matrix,
           zeros)
13 import matplotlib.pyplot as plt
14 from scipy.special import roots_chebyc
15 # deixando a saída simbolica em latex :
16 init_printing(use_latex= True)

1 """
2 =====
3 descrição : Solução da Equação Diferencial usando o
4 método da colocação.
5 data : 05/01/25
6 programador : Marcelo L. Almeida
7 =====
8 """
9 # criando os símbolos
10 x = symbols("x", real = True)
11 u = Function("u")(x)
12
13 # criando a equação diferencial
14 eq_edo = Eq(diff(u,x,2) + u, x)
15
16 # criando as condições de contorno como dicionarios
17 pvc = {u.subs(x,0) : 0,
18        u.subs(x,1) : 0}
19
20 # resolvendo a equação diferencial :
21 sol_analitic = dsolve(eq_edo,u, ics=pvc)
22
23 # criando o vetor de pontos no eixo x :
24 a , b = 0 , 1
25 vetx = np.linspace(a,b,100)
26
27 # avaliando a nossa solução analitica nesse vetor
28 sol_analitic = lambdify(x,sol_analitic.rhs,"numpy")
29 u_analtic = sol_analitic(vetx)
30
31 ## ===== computando o metodo da colocação
32      ##
33
34 N = 4 # quantidade de pontos de colocação
35 i = np.arange(1,N+1)
36 vet_xi = (a + b)/2 + ((b - a)/2)*np.cos(((2*i -1)/(2*N))*np.
37 pi)
38
39 # criando as funções de base : phi_j
40 lista_phi_j = Matrix([x**(i+1)*(1 - x) for i in range(N)])
41
42 K = zeros(N,N)

```

```

41 F = zeros(N,1)
42
43 # montando a matriz global do sistema :
44
45 for i in range(N) :
46     for j in range(N) :
47
48         expr = diff(lista_phi_j[j],x,2).subs(x,vet_xi[i
49             ]) + lista_phi_j[j].subs(x,vet_xi[i])
50
51         K[i,j] = expr
52
53
54 for i in range(N) :
55
56     F[i] = vet_xi[i]
57
58 # resolvendo o sistema linear usando a fatoração LU
59 cj = K.LUsolve(F)
60
61 ## ===== montando a solução geral =====
62 ##
63
64 u_n = sum(cj[i]*lista_phi_j[i] for i in range(N))
65
66 u_n = lambdify(x,u_n,"numpy")
67
68 ## ===== criando o plote ===== ##
69 fig, ax = plt.subplots(figsize = (8,6))
70
71 ax.plot(vetx, sol_analitic(vetx),"-b", label = "Solução_Analí
72     tica", zorder = 1)
73 ax.scatter(vetx[:,2], u_n(vetx[:,2]), facecolors = "none",
74     edgecolors = "r",zorder = 2,
75     label = "Solução_NUMérica_(Colocação)")
76 ax.set_xlabel(r"$x$")
77 ax.set_ylabel(r"$u(x)$")
78 ax.set_title(r"Plote_da_Solução_analítica_e_Numérica_para_a_
79     EDO:\dfrac{d^2u}{dx^2}+uu=x$")
80 ax.grid(True)
81 ax.legend(frameon = False , loc = 0)
82 ax.minorticks_on()
83
84 ## ===== salvando os dados em arquivos ===== ##
85 M_analitic = np.vstack((vetx,u_analtic)).T
86 np.savetxt("dados_analitic_coloc.txt", M_analitic,fmt="%.5f",
87     header="Solução_analítica_Método_da_colocação")

```

```

84 M_numeric = np.vstack((vetx,u_n(vetx))).T
85 np.savetxt("dados_numeric_coloc.txt", M_numeric,fmt="%.5f",
    header="Solução numérica - Método da colocação")

```

Após executar o código acima, temos o seguinte gráfico na página a seguir, que mostra a solução analítica e numérica usando o método da colocação. Na figura (1), temos em linha contínua e azul a solução analítica da equação diferencial e em círculos furados e vermelhos a solução numérica via colocação (apenas com alguns valores).

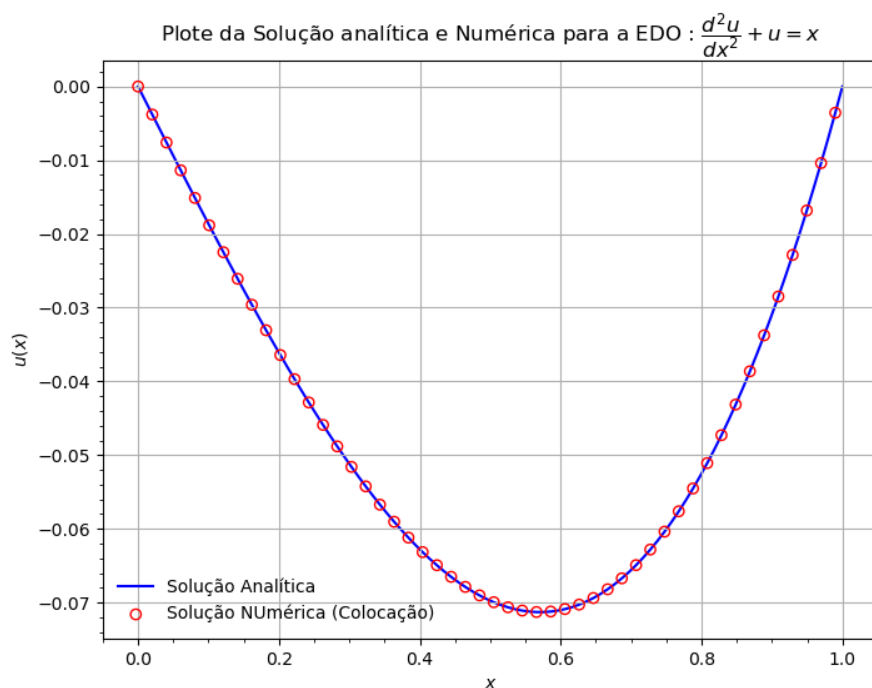


Figura 1: Solução Numérica e Analítica

Tão importante como a implementação do método numérico, é a implementação da análise do erro entre a solução exata e aproximada. Para isso, usaremos duas métricas para quantificar essa discrepância : *Erro Absoluto*.

O código a seguir cria a função para cálculo do erro e plota seus gráficos para comparação.

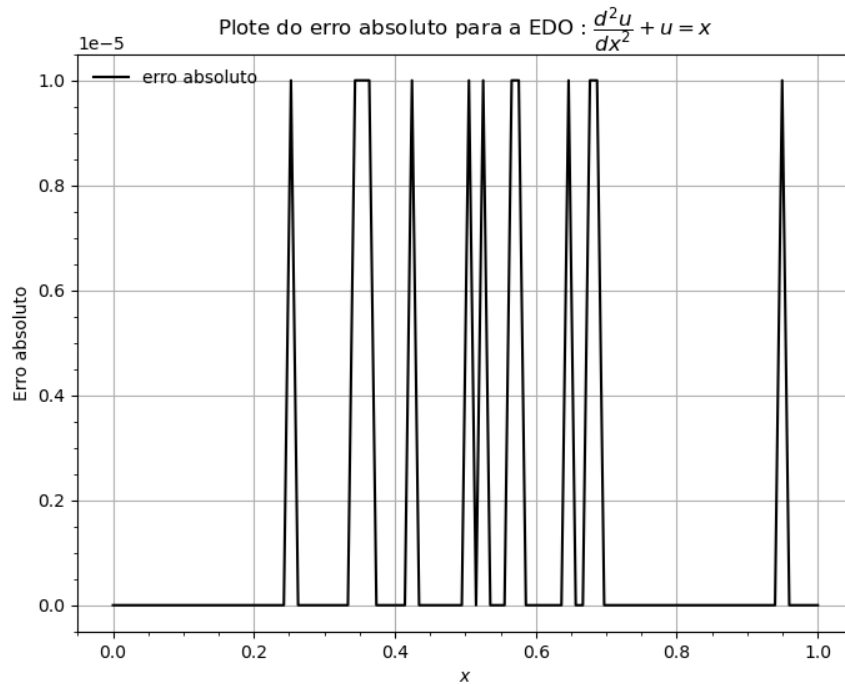


Figura 2: Erro Absoluto

2.2 Método de Mínimos Quadrados (MMQ)

Outro Método bastante importante para resolver equações diferenciais Ordinária e Parciais é o **Método de Mínimos Quadrados**. Diferente do método da colocação visto anteriormente, o método MMQ minimiza a integral usando o *quadrado do resíduo* sobre o intervalo do domínio do problema diferencial. Para entendermos como funciona esse método bem importante, consideramos a seguinte equação diferencial ordinária de 2 ordem genérica :

$$a(x)\frac{d^2u}{dx^2} + b(x)\frac{du}{dx} + c(x)u(x) = h(x) \quad \text{com} \quad u(a) = u_0, u(b) = u_1 \quad (2.12)$$

Aqui supomos que aproximamos a solução $u(x)$ por um conjunto de funções de base linearmente independentes, que satisfazem as condições de contorno do problema, como fizemos no método da colocação. Assim teremos

um resíduo $R(x)$. No método de minimizamos o quadrado do resíduo, que é equivalente a usar a função peso $w(x) = R(x)$, ou seja :

$$I = \int_a^b R^2(x) dx = 0 \quad (2.13)$$

Que é equivalente a minimizar a norma L^2 do resíduo :

$$\min_{c_j} \int_a^b \left\| R(x) \right\|^2 dx \quad (2.14)$$

Isso é matematicamente equivalente À resolver o seguinte problema diferencial :

$$\frac{\partial}{\partial c_j} \left(\int_a^b |R(x)|^2 dx \right) = 0 \quad (2.15)$$

Substituindo a solução aproximada $u_n(x) = \sum_{j=1}^n c_j \phi_j(x)$ no resíduo da equação (2.15) e usando a condição de minimização , teremos :

$$\int_a^b L(u_n(x) - h(x)) L(\phi_k(x)) dx = 0 \quad \text{para } k = 1, 2, \dots, n. \quad (2.16)$$

Isso gera um sistema linear da seguinte forma :

$$\sum_{j=1}^n c_j \int_a^b L(\phi_j(x)) L(\phi_k(x)) dx = \int_a^b L(\phi_k(x)) h(x) dx \quad (2.17)$$

A equação (2.17) acima pode ser escrita mna forma matricial :

$$\mathbf{A} \mathbf{c} = \mathbf{b} \quad (2.18)$$

onde :

- \mathbf{A} é a matriz do sistema, que possui a característica de ser simétrica e positiva definida.
- \mathbf{c} é o vetor de incógnitas, que são os coeficientes c_j .
- \mathbf{b} é o vetor de termos independentes, que são os produtos internos entre as funções de base e a função $h(x)$.

Mediante a essas características, podemos resolver o sistema linear usando a fatoração de Cholesky, que é uma técnica eficiente para resolver sistemas lineares com matrizes simétricas e positivas definidas. Essa fatoração decompõe a matriz \mathbf{A} em um produto de uma matriz triangular inferior por sua transposta e resolve o sistema em duas etapas:

- Primeiro, resolvemos o sistema triangular inferior $\mathbf{L}\mathbf{y} = \mathbf{b}$.
- Em seguida, resolvemos o sistema triangular superior $\mathbf{L}^T\mathbf{c} = \mathbf{y}$.

Essa abordagem é mais eficiente do que usar métodos diretos, como a eliminação de Gauss, especialmente para matrizes grandes e esparsas. A seguir, apresentamos um exemplo de aplicação do método de mínimos quadrados para resolver uma equação diferencial de 2ª ordem. O código a seguir implementa o método de mínimos quadrados para resolver a equação diferencial dada por:

$$\frac{d^2u}{dx^2} + u = x \quad \text{com} \quad u(0) = 0, u(1) = 0$$

A seguir será mostrado o código em Python que implementa o método de Mínimos Quadrados para obter a solução aproximada do PVC. Como os códigos estão ficando cada vez maiores, vou dividir em partes o código completo, ficando apenas uma parte de forma completa na pasta do método colocado no repositório do Github (repositório).

Módulos Python

```

1  """
2  =====
3  descrição : Solução da Equação Diferencial usando o
4  método de mínimos quadrados
5  data : 10/01/25
6  programador : Marcelo L. Almeida
7  =====
8  """
9  import numpy as np
10 import matplotlib.pyplot as plt
11 import sympy as smp
12 from sympy import (symbols, diff, integrate, lambdify,
13                    symbols,
14                    Eq, solve, dsolve, Function, Matrix, sin,
15                    cos, exp,
16                    init_printing)
17 import numpy.linalg as la
18 init_printing(use_latex=True)

```

Funções criadas Python

```

1
2  # criando a função para o operador L de segunda ordem
3
4  def L(phi) :
5      """

```



```

6     função que calcular o operador L sobre
7     cada função de aproximação phi.
8     """
9     a_x = 1 # termo da segunda derivada
10    b_x = 0  # termo da primeira derivada
11    c_x = 1  # termo da função
12
13    L_un = a_x*diff(phi,x,2) + b_x *diff(phi,x,1) + c_x * phi
14    return L_un
15
16
17    # criando a função para verificar se
18    # uma matriz é simetrica
19    def is_symmetric(A):
20        """
21        Verifica se a matriz A é simétrica.
22        """
23        return np.allclose(A, A.T)
24
25    # criando uma função para
26    # verificar se é positiva definida
27
28    def is_positive_definite(A):
29        """
30        verific se a matriz A é definida possitiva
31        """
32
33        autovalores = la.eigvals(A)
34
35        # teste de verificação
36
37        if (autovalores > 0).all() :
38
39            return True
40        else :
41
42            return False

```

Código Principal Python

```

1    ## ===== função principal ===== ##
2
3    if __name__ == "__main__" :
4
5        x = symbols('x', real = True)
6
7        ## solução analitica :
8        u = Function("u")(x)
9
10       # criando a equação diferencial

```

```

11 eq_edo = Eq(diff(u,x,2) + u, x)
12
13 # criando as condições de contorno como dicionarios
14 pvc = {u.subs(x,0) : 0,
15        u.subs(x,1) : 0}
16
17 # resolvendo a equação diferencial :
18 sol_analitic = dsolve(eq_edo,u, ics=pvc)
19
20 # criando a nossa funções de base
21 n = 3 # quantidade de termos usados
22 a , b = 0, 1 # intervalo de integração
23 lista_phi_j = [x**(i)*(1 - x) for i in range(1,n+1)] #
24               lista de funções phi
25 h_x = x # função fonte
26 # criando o sistema de equações :
27 A = np.empty((n,n), dtype= np.float64)
28 b = np.empty((n,1), dtype=np.float64)
29
30 # criando a matriz
31 for i in range(n) :
32
33     for j in range(n) :
34
35         phi_i = L(lista_phi_j[i]) # operador L{phi_i(x)}
36         phi_j = L(lista_phi_j[j]) # operador L{phi_j(x)}
37         integrando = integrate(phi_i*phi_j, (x, 0, 1)).
38             evalf()
39         A[i,j] = np.float64(integrando)
40
41 # criando o vetor b :
42 for i in range(n) :
43
44     phi_i = L(lista_phi_j[i])
45     term = integrate(phi_i*h_x,(x,0,1))
46     b[i] = np.float64(term)
47
48 # verificando se a matriz é simétrica e positiva definida
49 if is_symmetric(A) and is_positive_definite(A):
50     print("A matriz A é simétrica e positiva definida.")
51
52 # resolve usando a decomposição de cholesky
53 L = la.cholesky(A)
54 y = la.solve(L,b)
55 ci = la.solve(L.T,y)
56
57

```

```

58
59     else:
60         print("A matriz A não é simétrica ou não é positiva definida.")
61         # resolve usando a decomposição LU
62         P, L, U = la.lu(A)
63         y = la.solve(L, b)
64         ci = la.solve(U, y)
65
66     # criando a função de aproximação
67     phi = sum(ci[i] * lista_phi_j[i] for i in range(n))
68
69
70     # plotando a função de aproximação
71     x_vals = np.linspace(0, 1, 100)
72     sol_analitic = lambdify(x, sol_analitic.rhs, "numpy")
73     u_analtic = sol_analitic(x_vals)
74     phi_func = lambdify(x, phi[0], 'numpy')
75     y_vals = phi_func(x_vals)
76     plt.plot(x_vals, u_analtic, "-b", label = "Solução Exata",
77             zorder = 1)
78     plt.scatter(x_vals[:,2], y_vals[:,2], facecolors = "none",
79               , edgecolors = "r", label='Aproximação-MMQ',
80               s=20, zorder = 2)
81     plt.title('Função de Aproximação')
82     plt.xlabel('x')
83     plt.ylabel('phi(x)')
84     plt.legend(frameon = False , loc = 0)
85     plt.grid()
86     plt.show()

```

Após executar o código, temos a figura da solução via MMQ :

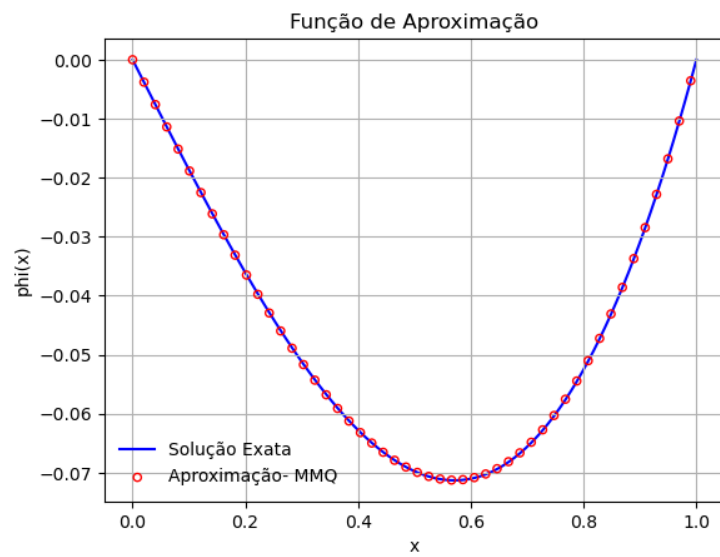


Figura 3: Solução Numérica e Analítica

2.3 Método de Galerkin

Referências

- [1] BASSALO, J. M. F., AND CATTANI, M. S. D. Elementos de física matemática. *São Paulo: Editora Livraria da Física 2* (2011).
- [2] BOYCE, W. E., AND DiPRIMA, R. C. *Equações Diferenciais Elementares e Problemas de Valores de Contorno*, vol. 10. LTC Rio de Janeiro, 2010.
- [3] CHAPRA, S. C., AND CANALE, R. P. *Numerical Methods for Engineers*, vol. 1221. McGraw-Hill, 2011.
- [4] FARLOW, S. J. *Partial Differential Equations for Scientists and Engineers*. Courier Corporation, 1993.
- [5] JOHANSSON, R., AND JOHN, S. *Numerical Python*, vol. 1. Springer, 2019.
- [6] KREYSZIG, E. *Matemática Superior para Engenharia*. Livros Técnicos e Científicos, 2009.
- [7] MATTHES, E. *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. No Starch Press, 2023.
- [8] NAKAMURA, J. *Applied Numerical Methods with Software*. Prentice Hall PTR, 1990.
- [9] PRESS, W. H. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.
- [10] SWEIGART, A. *Automate the Boring Stuff with Python*. No Starch Press, 2025.
- [11] ZIENKIEWICZ, O. C., AND MORGAN, K. *Finite Elements and Approximation*. Courier Corporation, 2006.
- [12] ZILL, D. G. *Equações Diferenciais com Aplicações em Modelagem*. Cengage Learning, 2016.