EP3 - Modelagem de um Sistema de Resfriamento de Chips

MAP3121 - Métodos Numéricos e Aplicações

Prof° Renato Vicente

Igor Costa D'Oliveira - 11391446

Paulo Gomes Ivonica - 11804532

13 de Julho de 2022

---

1. Introdução

O objetivo deste exercício-programa é modelar o comportamento da difusão térmica que ocorre em um processador ou chip de dimensões L × L de base e com valor h de altura. Para tal modela-se um resfriador ("cooler" ou placa fria) acoplado na parte superior do chip. Para simplificar, considerou-se o caso unidimensional analisando apenas a seção transversal do chip para cada x dentro de um intervalo [0, L]. Assumiremos que a espessura do chip (h) é fina de tal sorte que a variação de temperatura na vertical é desprezível.

Convencionou-se também que a troca de calor no topo do chip com o cooler é perfeita e logo não há perdas. A base é termicamente isolada, portanto não há trocas de calor entre o ambiente e a parte inferior do chip.

2. Equação de Calor

A distribuição de calor no interior do conjunto chip + bloco pode ser modelada pela equação do calor, fornecida no enunciado e obtida a partir da Lei de Fourier e da propriedade de conservação de energia em um sistema fechado. Sendo matematicamente escrita neste caso unidimensional como:

Figura 1 - Fórmula da Equação do Calor.

$$\rho C \frac{\partial T(t,x)}{\partial t} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial T(t,x)}{\partial x}\right) + Q(t,x)$$

Sendo T(x, t) a função que descreve a temperatura segundo a posição de um ponto e segundo o tempo de análise. A densidade do material é indicada por ρ, o calor específico do material é dado por C, já o K por sua vez é um parâmetro de condutividade térmica no material. E por fim Q(x,t) é a soma do calor gerado pelo chip e do calor absorvido pelo resfriador.

Considerando que tanto o processador quanto o resfriador atuam em regime constante, nota-se que a distribuição de temperatura no chip tende a um estado de equilíbrio. Fato este que permite concluir que a derivada da temperatura no tempo tende a zero, logo a nova equação de calor é dada por:

Figura 2 - Equação do Calor em Regime Estacionário.

$$-\frac{\partial}{\partial x}\left(k(x)\frac{\partial T(x)}{\partial x}\right) = Q(x),$$

Com esta equação é possível obter soluções de equilíbrio sabendo a quantidade de calor que é gerada e retirada do sistema, assim como as temperaturas nos extremos.

3. Método dos Elementos Finitos

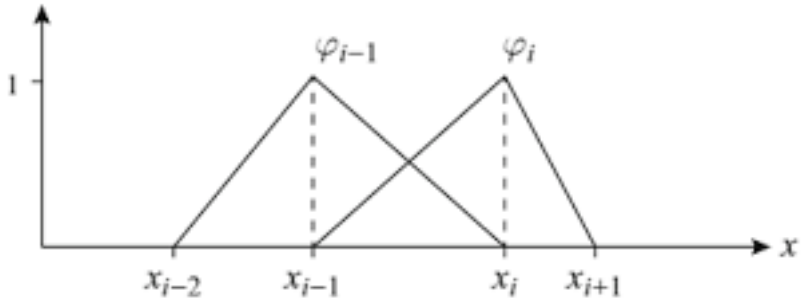
Para a solução numérica do problema, utilizou-se o método dos elementos finitos para aproximar o valor de uma função desejada.

O método consiste em criar um espaço vetorial dentro do espaço das funções, espaço esse formado por equações do tipo chapéu neste exercício. As funções do tipo chapéu são definidas a partir dos nós do intervalo [0, L], para determinar os **n** nós, dividiu-se o intervalo em trechos de comprimento $\frac{L}{n+1}$ o que resulta em **n** nós (obs: os extremos não são computados como nós), as funções chapéu precisam de intervalos chamados **suporte** para existirem, esse intervalo é formado para cada **i** no vetor dos nós e seus extremos são os nós $[x_{i-1}, x_{i+1}]$ e fora dele a função tem valor nulo.

Dados os suportes, se **x** está contido na primeira parte do **suporte**, ou seja: $[x_{i-1}, x_i]$ a função é uma reta crescente descrita por: $\phi_i(x) = \frac{x - x_{i-1}}{h}$ sendo h = $\frac{L}{n+1}$. Já se x está contido na segunda parte do

suporte, temos $[x_i,\ x_{i+1}]$, a função é uma reta decrescente descrita por: $\phi_i(x) = \frac{x_{i+1} - x}{h}$, graficamente as funções chapéu são do tipo:

Figura 3 - Exemplo de função chapéu.



Dado o conjunto das **n** funções chapéu, definidas para cada nó, agora foi necessário o uso destas para a projeção da solução U(x) que buscamos no espaço vetorial que criamos. Esta projeção é feita usando o produto interno dado por:

Figura 4 - Equação do produto interno usado.

$$\langle u, v \rangle_L = \int_0^1 \left[k(x)u'(x)v'(x) + q(x)u(x)v(x) \right] dx$$

sendo k(x) e q(x) parâmetros determinados pelo usuário do exercício-programa.

Neste exercício programa a integral acima vai de 0 até L e é resolvida numericamente usando a Quadratura de Gauss para dois pontos. Para isso, parametrizou-se os nós para que o método funcionasse para qualquer intervalo. O método no intervalo genérico [a; b] é dado por:

$$\int_a^b f(x)dx = A_1 f(x_1) + A_2 f(x_2)$$

Já para o intervalo [-1; 1] temos :

$$\int_{-1}^{1} f(x)dx = \frac{1}{2} f\left(\frac{1}{\sqrt{3}}\right) + \frac{1}{2} f\left(\frac{-1}{\sqrt{3}}\right)$$

Para adaptar a fórmula para qualquer intervalo, faz-se a seguinte parametrização.

$$x = \frac{(b-a)}{2}t + \frac{(b+a)}{2}$$

$$dx = \frac{(b-a)}{2}dt$$

e usamos $t_1 = \frac{1}{\sqrt{3}}$ e $t_2 = \frac{-1}{\sqrt{3}}$.

A projeção de U(x) neste espaço vetorial é a melhor aproximação possível para este problema dentro do espaço criado. Para obtê-la usamos as funções chapéu como base do espaço e resolvemos a seguinte equação matricial:

Figura 5 - Equação matricial da projeção.

$$\begin{bmatrix} \langle \phi_1, \phi_1 \rangle_L & \langle \phi_2, \phi_1 \rangle_L & \dots & \langle \phi_n, \phi_1 \rangle_L \\ \dots & \dots & \dots & \dots \\ \langle \phi_1, \phi_n \rangle_L & \langle \phi_2, \phi_n \rangle_L & \dots & \langle \phi_n, \phi_n \rangle_L \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ .. \\ \alpha_n \end{bmatrix} = \begin{bmatrix} \langle f, \phi_1 \rangle \\ .. \\ \langle f, \phi_n \rangle \end{bmatrix}$$

Dados os α_n achamos a solução aproximada na forma $U_n(x) = \sum_{i=1}^{n} \alpha_i \phi_i(x)$.

Note que os interiores dos suportes de ϕ_i e ϕ_j se interceptam apenas se $|i - j| \leq 1$. Desta constatação decorre que, $\langle \phi_i, \phi_j \rangle = 0$ se $|i - j| > 1$.
Isto faz com que a matriz do sistema linear acima seja tridiagonal.

Utilizando o algoritmo do primeiro exercício programa, resolve-se esse sistema linear associado à matriz de coeficientes e com o vetor dos coeficientes alfa acha-se a solução $U_n(x)$ aproximada.

4. Comentários sobre o código.

A primeira função é aquela que calcula os nós dentro de um intervalo [0, L]:

Figura 6 - Função que calcula os nós.

```
def nos(n, L):
    x = [0] * (n + 2)
    i = 0
    while i != (n + 2):
        x[i] = (L * i) / (n + 1)
        i = i + 1
    return x
```

O vetor **x** que recebe os valores dos nós tem **(n + 2)** posições porque além dos **n** valores dos nós ele recebe os valores dos extremos 0 e L. Note que determinou-se um sub-intervalo que espalha os nós equidistantemente uns dos outros por todo o intervalo inicial.

A segunda função implementada é a importada do primeiro exercício, que resolve um sistema linear cuja matriz de coeficientes seja tridiagonal:

Figura 7 - Função que implementa o algoritmo de decomposição LU.

```
def LUTriDig(a, b, c, d):
    n = len(a)
    u = [0] * n
    l = [0] * n
    x = [0] * n
    y = [0] * n

    u[0] = b[0]
    for i in list(range(1, n, 1)):
        l[i] = a[i] / u[i - 1]
        u[i] = b[i] - l[i] * c[i - 1]
    # solução de Ly = d
    y[0] = d[0]
    for i in list(range(1, n, 1)):
        y[i] = d[i] - l[i] * y[i - 1]

    # solução de Ux = y
    x[n - 1] = y[n - 1] / u[n - 1]

    i = n - 2
    while i >= 0:
        x[i] = (y[i] - c[i] * x[i + 1]) / u[i]
        i -= 1
    return x
```

Implementou-se também a função que determina as **n** funções chapéu, para tal, fez-se uso de uma comparação do argumento **x** da função com os nós do suporte, e dependendo da posição de **x** no suporte a reta descrita pela função será crescente e decrescente. Note que na mesma função já retornamos a equação da reta e sua derivada.

Nota-se também que a contagem das funções **phi** e suas derivadas **dphi** são feitas a partir do número 1 e para o **n**, totalizando exatamente n funções no vetor, e salienta-se que a indentação da linguagem python começa em 0 e é por isso que a posição no vetor é dada por **i -1 .**

Figura 8 - Implementação da função que calcula as funções chapéu.

```
def chapeu(n, L, x):  # função que cria as funções chapéu
    no = nos(n, L)
    phi = [0] * (n + 2)  # vetor que retorna a função chpéu pra cada i
    dphi = [0] * (n + 2)  # vetor que retorna a derivada da função chpéu pra cada i
    i = 1
    while i != (n + 1):  # implementação da lógica da função
        if no[i - 1] <= x <= no[i]:  # verifica se x está no primeiro sub-intervalo
            phi[i - 1] = (x - no[i - 1]) * (n + 1) / L  # descreve uma reta crescente
            dphi[i - 1] = (n + 1) / L  # a derivada de uma reta é uma constante
        elif no[i] <= x <= no[i + 1]:  # verifica se x está no segundo sub-intervalo
            phi[i - 1] = (no[i + 1] - x) * (n + 1) / L  # descreve uma reta decrescente
            dphi[i - 1] = -1 * (n + 1) / L
        else:
            phi[i] = 0  # fora do intervalo a função assume valor 0
            dphi[i] = 0
        i = i + 1
    return phi, dphi
```

Abaixo tem a função que estipula valores para o parâmetro **K(x)**, a condutividade térmica do material estudado, note que a função recebe o argumento **x** que é a posição do ponto estudado dentro do intervalo [0; L] e também recebe uma variável **ex**, esta última é a responsável por selecionar o teste a ser realizado.

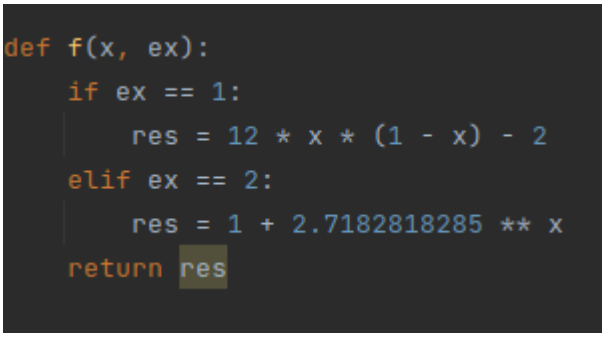
Figura 9 - Implementação da função que define K(x).

```
def k(x, ex):
    if ex == 1:
        f = 1
    elif ex == 2:
        f = 2.7182818285 ** x
    return f
```

Fez-se também uma função que generaliza a função f(x) cuja solução é a motivação da aproximação. Da mesma forma que a anterior, ela recebe o **x** e o **ex** em seu argumento, tal qual a sua derivada, ambas seguem logo abaixo:

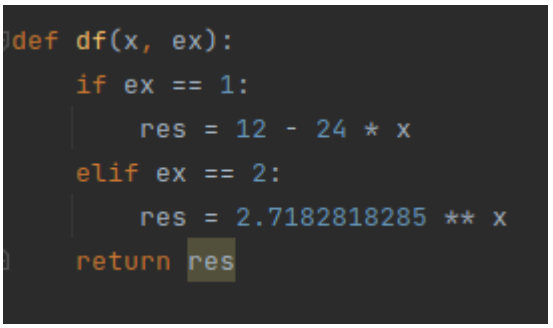
Nota-se no exemplo abaixo e no acima que o valor do número neperiano está em formato numérico 2,7182818285, isto se deu pelo entendimento da dupla que quanto menos bibliotecas usássemos menor seria a chance de erro de compilação em outra máquina, motivados por essa linha de raciocínio desde o primeiro exercício programa não utilizamos as bibliotecas **Numpy** e **Math** em nenhum trecho do código.

Figura 10 - Implementação da função f(x).



```
def f(x, ex):
    if ex == 1:
        res = 12 * x * (1 - x) - 2
    elif ex == 2:
        res = 1 + 2.7182818285 ** x
    return res
```

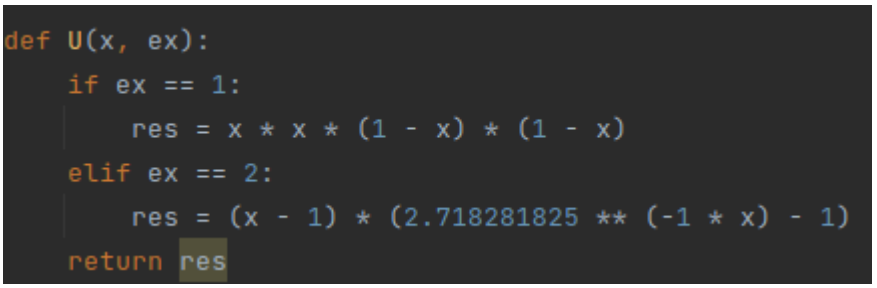
Figura 11 - Implementação da derivada da função f(x).



```
def df(x, ex):
    if ex == 1:
        res = 12 - 24 * x
    elif ex == 2:
        res = 2.7182818285 ** x
    return res
```

Semelhantemente às duas funções anteriores, fez-se uma função para definir a solução real do problema **U(x)** e esta varia tanto com o **x** quanto com o teste que estamos fazendo, logo ela também depende da variável **ex.**

Figura 12 - Implementação da função U(x).



```
def U(x, ex):
    if ex == 1:
        res = x * x * (1 - x) * (1 - x)
    elif ex == 2:
        res = (x - 1) * (2.718281825 ** (-1 * x) - 1)
    return res
```

As próximas quatro funções determinam os quatro vetores usados na decomposição LU, o vetor B define a diagonal principal, o vetor A define a

diagonal inferior, o vetor C define a diagonal superior e o vetor D é o vetor de respostas.

Para determinar os valores dos vetores foram criados subintervalos entre três nós consecutivos, e em cada um destes subintervalos foram calculados o peso e os dois pontos necessários para a aproximação usando o método de Gauss para dois pontos. Para cada iteração foram criadas as funções chapéu e já foram utilizadas para o cálculo dos produtos internos, e dessa forma calculando cada coeficiente usado no sistema linear.

Figura 13 - Inicialização dos vetores utilizados.

```
def vetorA(n, L, ex):  # função pra determinar o vetor das diagonais laterais da matriz de coeficintes
    no = nos(n, L)     # vetor de nós dentro do intervalo [0,L]
    vA = [0] * n       # declaração do vetor de resposta
    aux1 = [0] * n     # cada vetor auxiliar receberá metade da conta realizada
    aux2 = [0] * n
    j = 0
```

Figura 14 - Método iterativo para determinar cada termo do vetor A.

```
while j != (n - 1): # percorre-se o intervalo tomando três nós consecutivos
    c = no[j + 2]
    b = no[j + 1]
    a = no[j]

    A1 = (b - a) / 2  # determina o peso da aproximação de gauss para os dois subintervalos
    A2 = (c - b) / 2

    x1 = (b + a) / 2 - ((b - a) / (2 * 1.732051))  # determina o x1 e o x2 pra aproximação de gauss
    x2 = (b + a) / 2 + ((b - a) / (2 * 1.732051))
    x3 = (b + c) / 2 - ((c - b) / (2 * 1.732051))  # determina o x3 e o x4 pra aproximação de gauss
    x4 = (b + c) / 2 + ((c - b) / (2 * 1.732051))

    phi1, dphi1 = chapeu(n, L, x1)  # calcula a função chapéu pra x1
    phi2, dphi2 = chapeu(n, L, x2)  # calcula a função chapéu pra x2
    phi3, dphi3 = chapeu(n, L, x3)  # calcula a função chapéu pra x3
    phi4, dphi4 = chapeu(n, L, x4)  # calcula a função chapéu pra x4

    aux1[j] = A1 * ((k(x1, ex) * dphi1[j] * dphi1[j + 1] + q(x1) * phi1[j] * phi1[j + 1]) + (
            k(x2, ex) * dphi2[j + 1] * dphi2[j] + q(x2) * phi2[j + 1] * phi2[j]))
    aux2[j] = A2 * ((k(x3, ex) * dphi3[j] * dphi3[j + 1] + q(x3) * phi3[j] * phi3[j + 1]) + (
            k(x4, ex) * dphi4[j + 1] * dphi4[j] + q(x4) * phi4[j + 1] * phi4[j]))
    vA[j + 1] = aux1[j] + aux2[j]
    j = j + 1
return vA
```

Figura 15 - Inicialização dos vetores utilizados.

```
def vetorB(n, L, ex):  # função pra determinar o vetor da diagonal principal da matriz de coeficintes
    no = nos(n, L)     # vetor de nós dentro do intervalo [0,L]
    vB = [0] * n       # declaração do vetor de resposta
    aux1 = [0] * n     # cada vetor auxiliar receberá metade da conta realizada
    aux2 = [0] * n
    j = 0              # zera a variável que contará as iterações
```

Figura 16 - Método iterativo para determinar cada termo do vetor B.

```
while j != n:
    c = no[j + 2]
    b = no[j + 1]
    a = no[j]

    A1 = (b - a) / 2  # determina o peso da aproximação de gauss
    A2 = (c - b) / 2

    x1 = (b + a) / 2 - ((b - a) / (2 * 1.732051))  # determina o x1 e o x2 nós na aproximação de gauss
    x2 = (b + a) / 2 + ((b - a) / (2 * 1.732051))
    x3 = (b + c) / 2 - ((c - b) / (2 * 1.732051))
    x4 = (b + c) / 2 + ((c - b) / (2 * 1.732051))

    phi1, dphi1 = chapeu(n, L, x1)  # calcula a função chapéu pra x1
    phi2, dphi2 = chapeu(n, L, x2)  # calcula a função chapéu pra x2
    phi3, dphi3 = chapeu(n, L, x3)  # calcula a função chapéu pra x3
    phi4, dphi4 = chapeu(n, L, x4)  # calcula a função chapéu pra x4

    aux1[j] = A1 * ((k(x1, ex) * dphi1[j] * dphi1[j] + q(x1) * phi1[j] * phi1[j]) + (
            k(x2, ex) * dphi2[j] * dphi2[j] + q(x2) * phi2[j] * phi2[j]))
    aux2[j] = A2 * ((k(x3, ex) * dphi3[j] * dphi3[j] + q(x3) * phi3[j] * phi3[j]) + (
            k(x4, ex) * dphi4[j] * dphi4[j] + q(x4) * phi4[j] * phi4[j]))
    vB[j] = (aux1[j] + aux2[j])
    j = j + 1
return vB
```

Note que no vetor A e no vetor C logo abaixo o cálculo do produto interno é feito entre funções **phi** defasadas em uma posição já que **j = i + 1** o produto interno é feito entre duas funções **phi** consecutivas. Já no vetor B o produto interno é feito entre duas funções **phi** iguais.

Figura 17 - Inicialização dos vetores utilizados.

```
def vetorC(n, L, ex):  # função pra determinar o vetor das diagonais laterais da matriz de coeficintes
    no = nos(n, L)  # vetor de nós dentro do intervalo [0,L]
    vC = [0] * n    # declaração do vetor de resposta
    aux1 = [0] * n  # cada vetor auxiliar receberá metade da conta realizada
    aux2 = [0] * n
    j = 0  # zera a variável que contará as iterações
```

Figura 18 - Método iterativo para determinar cada termo do vetor C.

```
while j != n:
    c = no[j + 2]
    b = no[j + 1]
    a = no[j]

    A1 = (b - a) / 2  # determina o peso da aproximação de gauss
    A2 = (c - b) / 2

    x1 = (b + a) / 2 - ((b - a) / (2 * 1.732051))  # determina o x1 e o x2 nós na aproximação de gauss
    x2 = (b + a) / 2 + ((b - a) / (2 * 1.732051))
    x3 = (b + c) / 2 - ((c - b) / (2 * 1.732051))  # determina o x3 e o x4 nós na aproximação de gauss
    x4 = (b + c) / 2 + ((c - b) / (2 * 1.732051))

    phi1, dphi1 = chapeu(n, L, x1)  # calcula a função chapéu pra x1
    phi2, dphi2 = chapeu(n, L, x2)  # calcula a função chapéu pra x2
    phi3, dphi3 = chapeu(n, L, x3)  # calcula a função chapéu pra x3
    phi4, dphi4 = chapeu(n, L, x4)  # calcula a função chapéu pra x4

    aux1[j] = A1 * ((k(x1, ex) * dphi1[j] * dphi1[j + 1] + q(x1) * phi1[j] * phi1[j + 1]) + (
            k(x2, ex) * dphi2[j + 1] * dphi2[j] + q(x2) * phi2[j + 1] * phi2[j]))
    aux2[j] = A2 * ((k(x3, ex) * dphi3[j] * dphi3[j + 1] + q(x3) * phi3[j] * phi3[j + 1]) + (
            k(x4, ex) * dphi4[j + 1] * dphi4[j] + q(x4) * phi4[j + 1] * phi4[j]))
    vC[j] = aux1[j] + aux2[j]
    j = j + 1
return vC
```

O vetor D por sua vez já é diferente dos demais, o produto interno dele é feito entre a função phi e a função f(x) estudada, na forma:

Figura 19 - Implementação do método iterativo para determinar o vetor D.

```
def vetorD(n, L, ex):  # função pra determinar o vetor das respostas
    no = nos(n, L)
    vD = [0] * n
    j = 0
    while j != n:
        c = no[j + 2]
        b = no[j + 1]
        a = no[j]

        A1 = (b - a) / 2  # determina o peso da aproximação de gauss
        A2 = (c - b) / 2  # determina o peso da aproximação de gauss

        x1 = (b + a) / 2 - ((b - a) / (2 * 1.732051))  # determina o x1 e o x2 nós na aproximação de gauss
        x2 = (b + a) / 2 + ((b - a) / (2 * 1.732051))
        x3 = (b + c) / 2 - ((c - b) / (2 * 1.732051))  # determina o x1 e o x2 nós na aproximação de gauss
        x4 = (b + c) / 2 + ((c - b) / (2 * 1.732051))

        phi1, dphi1 = chapeu(n, L, x1)  # calcula a função chapéu pra x1
        phi2, dphi2 = chapeu(n, L, x2)  # calcula a função chapéu pra x2
        phi3, dphi3 = chapeu(n, L, x3)  # calcula a função chapéu pra x1
        phi4, dphi4 = chapeu(n, L, x4)  # calcula a função chapéu pra x2
        vD[j] = A1 * ((phi1[j] * f(x1, ex)) + (phi2[j] * f(x2, ex))) + A2 * (
                (phi3[j] * f(x3, ex)) + (phi4[j] * f(x4, ex)))
        j = j + 1
    return vD
```

Por último, a figura 20 ilustra a função que é chamada pela main para a resolução dos exercícios, logo esta função calcula a Un(x) a U(x) e o erro máximo entre as duas. A função main não será mostrada porque apenas possui a interface com o usuário.

Figura 20 - Função chamada na main para a resolução dos ex.

```
def resolucao(n, x, ex): # função pra determinar o erro e Un(x) e U(x) das respostas
    a_t = vetorA(n, 1, ex)
    b_t = vetorB(n, 1, ex)
    c_t = vetorC(n, 1, ex)
    d_t = vetorD(n, 1, ex)
    alfas = LUTriDig(a_t, b_t, c_t, d_t)
    print("Un(", x,") = ", Un(n, 1, alfas, x))
    print("U(", x,") =", U(x, ex))
    no = nos(n, 1)
    i = 0
    e_max = 0
    while i != n: # Determina o erro máximo
        e = U(no[i], ex) - Un(n, 1, alfas, no[i])
        if e < 0:
            e = e * -1
        if e > e_max:
            e_max = e
        i = i + 1
    print("Erro máximo = ", e_max)
```

5. Resolução dos Exercícios

5.1 Validação

O programa será testado utilizando um intervalo [0, 1], onde $k(x) = 1$, $q(x) = 0$, $f(x) = 12x(1 - x) - 2$, com condições de contorno homogêneas. Para este exercício é esperado uma solução exata igual a $u(x) = x^2(1 - x)^2$, e será calculada a diferença entre os resultados encontrados com a solução exata da seguinte maneira $e = ||u_n - u|| = max_{i=1,\dots,n}|u_n(x_i) - u(x_i)$.

Para encontrar os resultados deste teste basta escolher a primeira opção do programa e depois é possível escolher entre um valor desejável para N e um valor entre 0 e 1 para x ou a opção com os valores de N iguais aos do enunciado, onde n = 7, 15, 31 e 63. A figura abaixo ilustra as opções.

Figura 21 - Escolha do Teste.

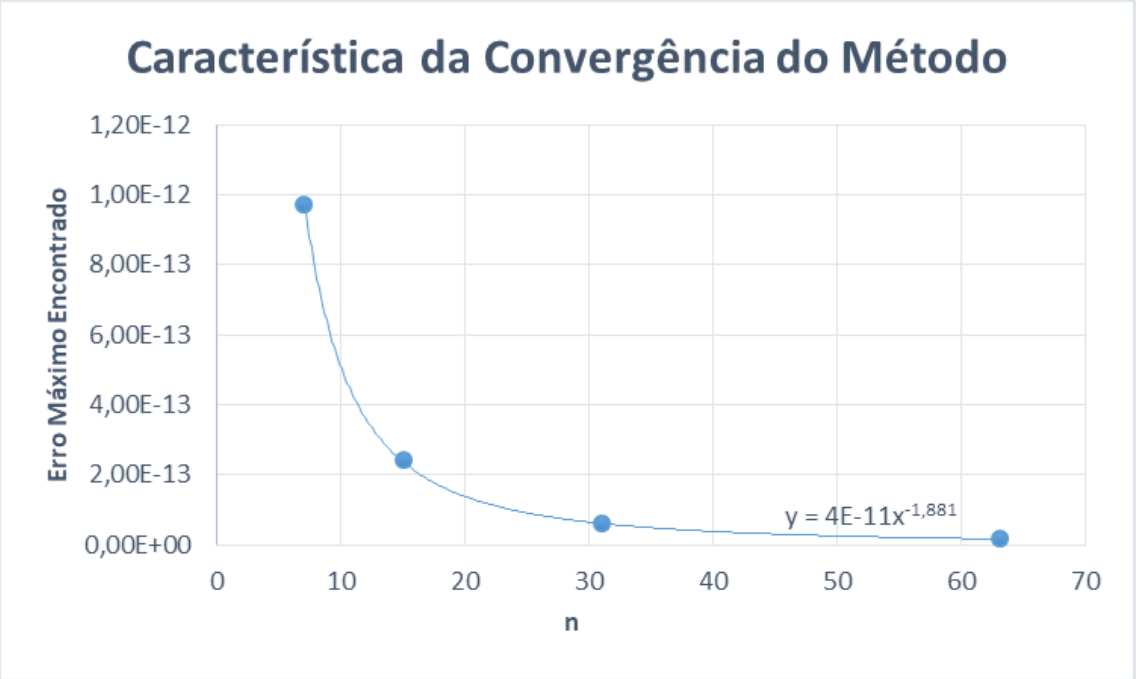
```
-------- Métodos Numéricos EP3 --------

1 - Validação: f(x) = 12x(x-1) - 2.
2 - Validação: f(x) = e^x - 1.
3 - Equilíbrio com forças de calor.
4 - Equilíbrio com variação de material.
5 - Sair.
Digite uma opção: 1

-------- Validação: f(x) = 12x(x-1) - 2 --------

1 - Escolher um 'N' e um 'x'.
2 - Usar parâmetros do enunciado.
3 - Voltar.

Digite uma opção:
```

Ao escolher a opção dois os resultados da figura 22 foram encontrados. É importante ressaltar que foi escolhido um x igual a 0,5 para testar os dois valores de $u_n(x)$ e $u(x)$. É possível observar que os erros máximos encontrados são muito pequenos na ordem de grandeza 10^{-13} para N pequenos. Logo, é possível afirmar que está funcionando como o esperado para funções polinomiais. Além disso, é possível observar na figura 23 que a convergência do método seguiu uma característica de segunda ordem, visto que quanto maior o valor do N menor foi o valor do erro máximo resultando na seguinte aproximação $erro = (4E - 11)N^{-1,881}$, logo é possível confirmar que o método tem convergência de segunda ordem como o esperado.

Figura 22 - Resultados encontrados no programa.

```
Paramêtros utilizados:

a = 0
b = L = 1
q(x) = 0
k(x) = 1
f(x) = 12 * x * (x - 1) - 2
u(x) = x^2 * (1 - x)^2

Resultados para n = 7:
Un( 0.5 ) =  0.0624999999990277
U( 0.5 ) = 0.0625
Erro máximo =  9.722986304971926e-13

Resultados para n = 15:
Un( 0.5 ) =  0.06249999999975694
U( 0.5 ) = 0.0625
Erro máximo =  2.430625145599663e-13

Resultados para n = 31:
Un( 0.5 ) =  0.06249999999993908
U( 0.5 ) = 0.0625
Erro máximo =  6.091654958240156e-14

Resultados para n = 63:
Un( 0.5 ) =  0.062499999999984325
U( 0.5 ) = 0.0625
Erro máximo =  1.567496132892643e-14
```

Figura 23 - Convergência do Método.



5.2 Complemento para Validação

Para verificar se o código está realmente funcionando será realizado um segundo teste adicional. Neste exercício será usado um intervalo entre [0, 1], onde $k(x) = e^x$, $q(x) = 0$, $f(x) = e^x + 1$, com condições de contorno homogêneas. Para este exercício é esperado uma solução exata igual a $u(x) = (x - 1).(e^{-x} - 1)$, e será calculada a diferença entre os resultados encontrados com a solução exata da seguinte maneira $e = ||u_n - u|| = max_{i=1,\dots,n}|u_n(x_i) - u(x_i)$.

Igual ao exercício do item 5.1, este exemplo possui dois modos de operação no nosso programa, o primeiro modo o usuário poderá escolher um N e um x o qual desejar, já o segundo modo apresentará os resultados para os valores de N iguais a 7, 15, 31 e 63 de acordo com o enunciado proposto. A figura 24 ilustra os resultados encontrados pelo programa com os seus respectivos erros máximos. É importante ressaltar que foi escolhido um x igual a 0,5 para testar dois valores de $u_n(x)$ e $u(x)$.

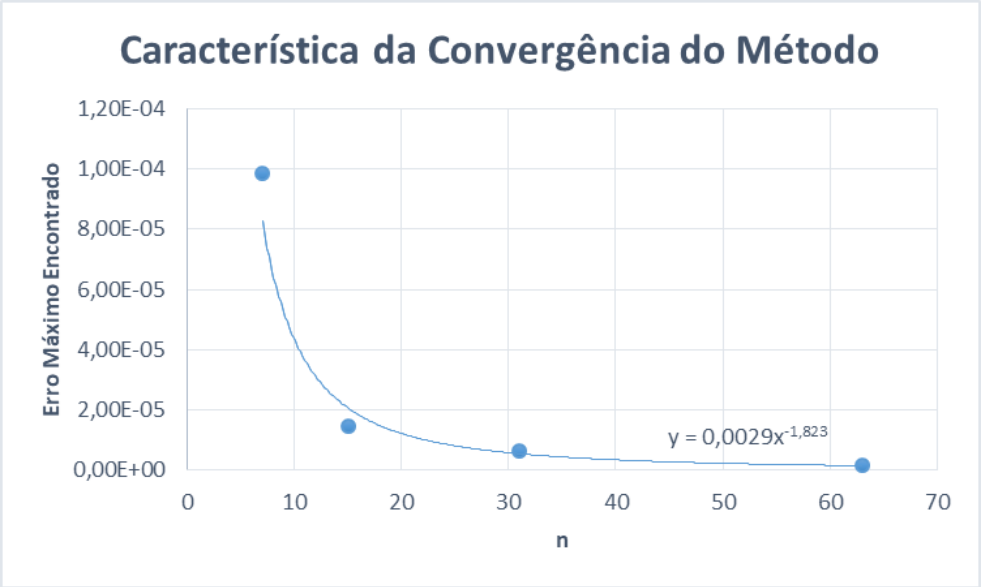
Figura 24 - Resultados encontrados no programa.

```
Paramêtros utilizados:

a = 0
b = L = 1
q(x) = 0
k(x) = e^x
f(x) = e^x + 1
u(x) = (x - 1) * (e^(-x)- 1 )

Resultados para n = 7:
Un( 0.5 ) =  0.19663806487101257
U( 0.5 ) = 0.1967346701436833
Erro máximo =  9.851549954836836e-05

Resultados para n = 15:
Un( 0.5 ) =  0.19671049869864038
U( 0.5 ) = 0.1967346701436833
Erro máximo =  2.4813096800913037e-05

Resultados para n = 31:
Un( 0.5 ) =  0.19672862602366983
U( 0.5 ) = 0.1967346701436833
Erro máximo =  6.210848629423227e-06

Resultados para n = 63:
Un( 0.5 ) =  0.19673315903499627
U( 0.5 ) = 0.1967346701436833
Erro máximo =  1.5536288059392334e-06
```

Neste teste os erros máximos encontrados foram maiores aos encontrados no item 5.1, porém este resultado já era esperado visto que a função a ser calculada era uma exponencial ao invés de um polinômio como no primeiro caso, logo o erro seria maior. Mesmo aumentando o erro ele ainda continuou muito pequeno na ordem de 10^{-6}. Além disso, é possível observar

na figura 25 que a convergência do método continuou sendo de segunda ordem, calculando as aproximações com n = 7, 15, 31 e 63 e avaliando em cada caso o valor do erro máximo.

Figura 25 - Convergência do Método.



Por último, utilizando a opção 1 do programa foi escolhido um valor 993 para N e para x o valor 0.5, com isso foi obtido os valores de Un, U e o erro máximo. Na figura 26 ilustra os resultados e é possível observar que o erro máximo foi muito menor do que o erro encontrado para N = 63 como o esperado. Se formos utilizar a aproximação encontrada na figura 25 $erro = 0,0029N^{-1,894}$ utilizando N = 993 temos o erro igual a 9,97E-9, que é muito próximo do erro máximo encontrado a partir do programa e com estes testes foi possível validar o código e a sua convergência .

Figura 26 - Teste com N = 993.

```
Escolha uma quantidade N de nós: 993
Escolha um valor para x entre 0 e 1 como argumento das funções: 0.5

Paramêtros utilizados:

a = 0
b = L = 1
q(x) = 0
k(x) = e^x
f(x) = e^x + 1
u(x) = (x - 1) * (e^(-x)- 1 )

Resultados:

Un( 0.5 ) =  0.19673466387906413
U( 0.5 ) = 0.1967346701436833
Erro máximo =  6.441292998982107e-09
```

5.3 Equilíbrio com forçantes de calor

Neste exercício vamos considerar que o chip seja formado apenas de silício com $k(x) = k = 3,6W/mK = 3600W/K$, e iremos considerar que há produção de calor pelo chip e que exista resfriamento. Além disso, será usado a temperatura externa como constante sendo de aproximadamente 20°C que é igual a 293,15 K. Também será assumido que o calor gerado pelo chip $Q_+ = 37,5\,MW/m^3$ e que o calor retirado pelo sistema seja de $Q_- = 7,5\,MW/m^3$. Logo, temos que $Q|(x) = Q_+ - Q_- = 30\,MW/m^3$. Para o comprimento total L foi escolhido 20 cm.

A figura 27 ilustra os resultados obtidos do EX3 com L igual a 20 cm e x igual a 10 cm. É possível observar que o programa está chegando nos valores esperados, porém o erro está muito maior do que os primeiros dois testes. Vale ressaltar que a temperatura está em K então para transformá-la em graus celsius basta subtrair por 273,15K. Logo, a temperatura encontrada para x igual a 10 cm foi de 20,4166°C.

Figura 27 - Teste com Ex3.

```
Paramêtros utilizados:

L = 0.02
x = 0.01

Resultados para n = 7:
Un( 0.01 ) =  293.56249229179565
U( 0.01 ) = 293.5666666666666
Erro máximo =  0.004174374870956399

Resultados para n = 15:
Un( 0.01 ) =  293.5624922917955
U( 0.01 ) = 293.5666666666666
Erro máximo =  0.004174374871126929

Resultados para n = 31:
Un( 0.01 ) =  293.5624922917955
U( 0.01 ) = 293.5666666666666
Erro máximo =  0.004174374871126929

Resultados para n = 63:
Un( 0.01 ) =  293.5624922917955
U( 0.01 ) = 293.5666666666666
Erro máximo =  0.004174374871126929
```

6. Conclusão

Neste programa foi utilizado o método dos elementos finitos para o cálculo de equações diferenciais. Além disso, foi utilizado o exercício programa 1 e 2 para a resolução de matrizes tridiagonais e a resolução de integrais de primeira e segunda ordem. Foi visto que o método apresenta uma eficiência elevada, tendo em vista aos erros máximos encontrados nos testes realizados. Também, foi possível concluir que algumas funções possuem maior complexidade, como funções exponenciais e senoidais, e com isso tiveram um erro maior em comparação a funções mais simples, como polinômios de graus menores. Entretanto, mesmo que o erro tenha aumentado, o seu valor ainda continuou pequeno e aceitável, visto que apresentou uma ordem de grandeza menor que 10^{-6}.

Vale ressaltar também que a convergência do método foi de segunda ordem, visto que quanto maior o N menor foi o erro com uma relação quadrática. Por fim, é possível concluir que o código está funcionando como o esperado e foi um sucesso, visto que está resolvendo as funções do problema rapidamente e com um grau de precisão elevado. Para N maiores que 1000 foi visto que o programa demorou unidades de segundos para o cálculo do problema, mas isso já era esperado.

7. Referências

[1] Enunciado Exercício-Proposto 3. Visto em: https://edisciplinas.usp.br/.
Acesso em 1 de Julho de 2022.

[2] Métodos Numéricos. Visto em: https://dlmf.nist.gov/
Acesso em 1 de Julho de 2022.

[3] Livro-texto. Visto em
https://www.pdfdrive.com/numerical-analysis-9th-edition-e157182502.html