Escola Politécnica Da Universidade de São Paulo

MAP3121 - Métodos Numéricos e Aplicações

EP3

Aluno:
Felipe Cardenas Lima Namour
NUSP: 11807111

Aluno:
Brenda Moreira Santos
NUSP: 11374818

1 Introdução

O desenvolvimento desse trabalho tem como objetivo explorar e analisar a modelagem da difusão térmica de um processador ou chip de computador. Visando isso, serão estudados os parâmetros fornecidos no enunciado da ativada juntamento com a influência deles nos resultados encontrados.

No código utilizamos as biblotecas numpy, math e re do Python. Para a implementação do método de elementos finitos com o intuito de resolver a equação:

$$L(u(x)) := (-k(x)u'(x))' + q(x)u(x) = f(x)x \in (0, L), u(0) = a, u(L) = b \quad (1)$$

2 O código e suas funções

O código desenvolvido para esse projeto conta com o uso de funções previamente criadas para os projetos anteriores do curso, sendo elas as funções responsáveis pela realização das integrais necessárias, como podem ser vistas abaixo:

```
def integralSimples_fphi(a, b, funcaoselect, x, i, h):
    w = np.sqrt(3)/3
    t = (w/2)*(b-a)
    t1 = (a+b)/2
    resultado = fvezesphi(t + t1, x, i, funcaoselect, h) +
                                        fvezesphi(-t + t1, x, i,
                                        funcaoselect, h)
    return resultado*((b-a)/2)

def integralSimples_phiphi(a, b, x, j, i, h, k, q):
    w = np.sqrt(3)/3
    t = (w/2)*(b-a)
    t1 = (a+b)/2
    resultado = phiphi(t + t1, k, q, x, i, j, h) + phiphi(-t + t1,
                                        k, q, x, i, j, h) #(xvar, k,
                                        q, x, i, j, h)
    return resultado*((b-a)/2)
```

Com isso, temos outras funções intermediarias que são usadas diversas vezes para obter os resultados finais desejados.

funções para determinar ϕ:

```
        def phi(x, x0, x1, xi, h):
            if  (x0 <= x <= xi):
                return ((x-x0)/h)
```

```
            elif (xi <= x <= x1):
                return ((x1-x)/h)
            else:
                return 0

        def phi_l(x, x0, x1, xi, h):
            if  (x0 <= x <= xi):
                return (1/h)
            elif (xi <= x <= x1):
                return (-1/h)
            else:
                return 0
```

Manipulações algébricas de ϕ e f:

```
def fvezesphi(xvar, x, i, funcaoselect, h): #multiplica f e phi
    return (funcaoescolhida(xvar, funcaoselect, 0) * phi(xvar, x[i-
                                        1], x[i+1], x[i], h))

def phiphi(xvar, k, q, x, i, j, h): #soma dos produtos vetoriais de
                                      phi_1 com phi_1, multiplicado
                                     pelo k(xvar) e de phi com phi
    phiphi = ( k(xvar)*phi_l(xvar, x[i-1], x[i+1], x[i], h)*phi_l(
                                        xvar, x[j-1], x[j+1], x[j], h
                                        ) + q(xvar)*phi(xvar, x[i-1],
                                         x[i+1], x[i], h)*phi(xvar, x
                                        [j-1], x[j+1], x[j], h) )
    return phiphi

def produtointerno_phiphi(q, k, x, j, i, h):
    return integralSimples_phiphi(x[i-1], x[i], x, j, i, h, k, q) +
                                         integralSimples_phiphi(x[i],
                                         x[i+1], x, j, i, h, k, q)#(L
                                        , x, j, i, h, k, q):

def produtointerno_f_phi(x, i, funcaoselect, h):
    return ((integralSimples_fphi(x[i-1], x[i], funcaoselect, x, i,
                                         h) + integralSimples_fphi(x[
                                        i], x[i+1], funcaoselect, x,
                                        i, h))) #integralSimples(a, b
                                        , funcaoselect, x, i):
```

Montagem das matrizes do sistema para achar a solução, onde a matriz A é a matriz dos produtos vetoriais entre ϕ, e a matriz B é a que contém os produtos vetoriais entre ϕ e $f(x)$:

```
    def montarMatrizA_k1_q0(n, h, x, k, q): #MATRIZ QUANDO K(X) = 1
                                         e Q(X) = 0
```

```
    A = np.zeros((n,n))
    Am = np.zeros(n)
    As = np.zeros(n)
    Ai = np.zeros(n)
    for i in range(0, n):
       for j in range(0, n):
            #diagonal principal
            if i == j: #2/h
                valor = produtointerno_phiphi(q, k, x, j+1, i+1, h)
                A[i][j] = valor
                Am[j] = valor
            #diagonal superior
            elif (j - i) == 1: #-1/h
                valor = produtointerno_phiphi(q, k, x, j+1, i+1, h)
                A[i][j] = valor
                As[j] = valor
            #diagonal inferior
            elif (j - i) == -1: #-1/h
                valor = produtointerno_phiphi(q, k, x, j+1, i+1, h)
                A[i][j] = valor
                Ai[j] = valor
    return A, As, Am, Ai


def montarMatrizB(n, x, funcaoselect, h): #VETOR SOLUCAO DA MATRIZ
                                          A
    B = np.zeros(n)
    for i in range(0,n):
        B[i] = produtointerno_f_phi(x, i+1, funcaoselect, h)
    return B
```

Outro ponto muito importante para o desenvolvimento do programa é encontrar o erro dos cálculos feitos, para que possam ser analizados. Com esse intuito, foi implementada a função a seguir que encontra o $\bar{u}$ e depois o erro por meio do módulo da diferença entre esse valor obtido e o $\bar{U}$, que é encontrado calculando na função escolhida com o n fornecido. Após o erro ser calculado para todos os valores de no interval0 [0, it], a função retornará o erro máximo entre eles.

```
    def calcularErro(n, funcaoselect, alphas, h, x, it):

    def ubarra(xvar, alphas, x, h, n):
        ubarra = 0.0
        for j in range(1,n+1):
            ubarra += alphas[j-1]*phi(xvar,x[j-1],x[j+1],x[j],h)
        return ubarra

    erromax = 0.0
    xvar = np.linspace(0,1,it)
```

```
    for i in range(0,it):
        U = funcaoescolhida(xvar[i], funcaoselect, 1)
        Ubarra = ubarra(xvar[i], alphas, x, h, n)
        erro = abs(Ubarra - U)
        if erro >= erromax:
            erromax = erro

    return erromax
```

Por fim, temos uma função responsável por retornar a função de cada caso de teste:

```
    def funcaoescolhida(x, n, b):
    if   n == 1: ## VALIDACAO
        if b == 1:
            return (x**2)*((x-1)**2) #u(x)
        else:
            return (12*x*(1 - x)) - 2 #f(x)

    elif n == 2: ##COMPLEMENTO
        if b == 1:
            return ((x-1)*(math.e**(-x) - 1)) #u(x)
        else:
            return (math.e**x + 1) #f(x)
    elif n == 3: ##EQUILIBRIO FORCANTES DE CALOR 4.3 - COM
                                        CONSTANTES
        if b == 1:
            return  0#u(x)
        else:
            Qmais =  37.5  * 1000000
            Qmenos = 11.25 * 1000000
            return  Qmais - Qmenos#f(x)
    elif n == 4: ##EQUILIBRIO FORCANTES DE CALOR 4.3 - NAO
                                        CONSTANTE
        if b == 1:
            return  0#u(x)
        else:
            Qmais = 37.5 * 1000000
            Qmenos = 11.25 * 1000000
            L = 0.02
            sigma = 1
            theta = 0.01
            return  (Qmais)*math.e**( (-(x-(L/2))**2) / (sigma**2)
                                                ) - (Qmenos)*( math.e
                                                **((-x**2)/(theta**2)
                                                ) + math.e**((-(x-L)
                                                **2)/(theta**2)) )#f(
                                                x)
```

2.1 Códigos para os equilíbrios

Após isso, temos o desenvolvimentos das diferentes funções principais dependendo de quais informações são pedidas e para qual cenário. A princípio serão determinadas as temperaturas em equilibrio com forçantes de calor com aquecimento e resfriamento constantes. Nesse caso, os αs resultantes da decomposição LU, implementada no EP 1, serão Com esse fim, começamos determinando os αs pelo método da decomposição LU, implementado no EP1, em seguida é determinado o ū por meio de

$$\bar{u} = \sum_{i=1}^{n+1} \alpha(i-1)\phi(x_var, x(i-1), x(i+1), x(i), h) \tag{2}$$

sendo ϕ, a primeira função descrita nessa secção do relatório. Após a obtenção desses valores, cada componente do vetor resultante, de tamanho n+3 será dado por

$$\bar{u} = \sum_{i=0}^{n+3} va + ((vb - va)x(i)) - 273,5 \tag{3}$$

Desta forma, são encontrados os valores das temperaturas em C$^{\underline{o}}$ para cada x em mm.

```
def main_equilibriocomforcantesdecalor_constante(n, plotar): #4
                                                .3
L = 0.02
va = 20 + 273.5
vb = 20 + 273.5
def k(x):
    return 3.6
def q(x):
    return 0
#CALOR TOTAL CONSTANTE:
funcaoselect = 3
h = L/(n+1)
x = np.zeros(n+2)
for i in range(0, n+2, 1):
    x[i] = (i)*h
A, a, b, c = montarMatrizA_k1_q0(n, h, x, k, q)
B = montarMatrizB(n, x, funcaoselect, h)
l, u = ep1.decompLU(a ,b ,c, n)
alphas = ep1.solucaoLU(l, u, c, B, n)

def ubarra(xvar, alphas, x, h, n):
    ubarra = 0.0
    for j in range(1,n+1):
        ubarra += alphas[j-1]*phi(xvar,x[j-1],x[j+1],x[j],h)
    return ubarra
```

```
    resultado = np.zeros(n+2)
    for i in range(0,n+2):
        resultado[i] = ubarra(x[i], alphas, x, h, n) + va + ((vb-va
                                                ) * x[i]) - 273.5

    #RESULTADOS:
    for i in range(0,n+2):
        print("A temperatura em x = " + str(x[i]) + "mm vale: " +
                                                str(resultado[i]) + "[U+FFFD]C
                                                ")
    if plotar == 1:
        plt.plot(x*1000, resultado)
        plt.title('Temperatura com geracao de calor constante')
        plt.ylabel('Temperatura ([U+FFFD])
        plt.xlabel('x(mm)')
        plt.show()
```

Outra análise feita é a da obteção dos valores para um modelo realista no equilibrio com forçantes de calor com aquecimento e resfriamento, que se difere da anterior pela função utilizada: nesse caso temos

$$u(x) = 0 \tag{4}$$

$$f(x) = (Q_+ e^{\frac{-(x-\frac{L}{2})^2)}{\sigma^2}}) - (Q_-(e^{\frac{-x^2}{\theta^2}} + e^{\frac{-(x-L)^2)}{\theta^2}}) \tag{5}$$

ao invés de

$$u(x) = 0 \tag{6}$$

$$f(x) = Q_+ - Q_- \tag{7}$$

como é no primeiro caso de equilíbrio apresentado.

```
    def main_equilibriocomforcantesdecalor(n, plotar): #4.3
    L = 0.02
    va = 20 + 273.5
    vb = 20 + 273.5
    def k(x):
        return 3.6
    def q(x):
        return 0
    #CALOR TOTAL CONSTANTE:
    funcaoselect = 4
```

```
    h = L/(n+1)
    x = np.zeros(n+2)
    for i in range(0, n+2, 1):
        x[i] = (i)*h
    A, a, b, c = montarMatrizA_k1_q0(n, h, x, k, q)
    B = montarMatrizB(n, x, funcaoselect, h)
    l, u = ep1.decompLU(a ,b ,c, n)
    alphas = ep1.solucaoLU(l, u, c, B, n)

    def ubarra(xvar, alphas, x, h, n):
        ubarra = 0.0
        for j in range(1,n+1):
            ubarra += alphas[j-1]*phi(xvar,x[j-1],x[j+1],x[j],h)
        return ubarra

    resultado = np.zeros(n+2)
    for i in range(0,n+2):
        resultado[i] = ubarra(x[i], alphas, x, h, n) + va + ((vb-va
                                            ) * x[i]) - 273.5 #ubarra
                                            (x[i], alphasGER, x, h, n
                                            ) + va + (vb-va) * x[i] -
                                             ubarra(x[i], alphasDIS,
                                            x, h, n) - 273.5

    # RESULTADOS:
    for i in range(0,n+2):
        print("A temperaatura em x = " + str(x[i]) + "mm vale: " +
                                                str(resultado[i]) + "
                                            [U+FFFD]C
    if plotar == 1:
        plt.plot(x*1000, resultado)
        plt.title('Temperatura')
        plt.ylabel('Temperatura (C[U+FFFD])
        plt.xlabel('x(mm)')
        plt.show()
```

Por fim, vamos calcular no equilíbrio com forçantes de calor com dois materiais no chip. A diferença observada aqui é no função k(x) que será utilizada, visto que a função selecionada é a mesma que para quando trabalhamos com o modelo realista. Uma vez que nos outros casos k(x) = 3,6, nesse k(x) =

$$\begin{cases} k_s, (\frac{L}{2} - d) <= x <= (\frac{L}{2} + d)) \\ k_a \end{cases}$$

```
    def main_doismateriais(n, plotar, ks, ka, d):
    L =  0.02
    va = 20 + 273.5
    vb = 20 + 273.5
```

```
def k(x):
    if (L/2 - d) <= x <= (L/2 + d):
        return ks
    else:
        return ka
def q(x):
    return 0
funcaoselect = 4
h = L/(n+1)
x = np.zeros(n+2)
for i in range(0, n+2, 1):
    x[i] = (i)*h
A, a, b, c = montarMatrizA_k1_q0(n, h, x, k, q)
B = montarMatrizB(n, x, funcaoselect, h)
l, u = ep1.decompLU(a ,b ,c, n)
alphas = ep1.solucaoLU(l, u, c, B, n)

def ubarra(xvar, alphas, x, h, n):
    ubarra = 0.0
    for j in range(1,n+1):
        ubarra += alphas[j-1]*phi(xvar,x[j-1],x[j+1],x[j],h)
    return ubarra

resultado = np.zeros(n+2)
for i in range(0,n+2):
    resultado[i] = ubarra(x[i], alphas, x, h, n) + va + ((vb-va
                                        ) * x[i]) - 273.5

# RESULTADOS:
for i in range(0,n+2):
    print("A temperaatura em x = " + str(x[i]) + "mm vale: " +
                                            str(resultado[i]) + "
                                        [U+FFFD]C
if plotar == 1:
    plt.plot(x*1000, resultado)
    plt.title('Temperatura')
    plt.ylabel('Temperatura (C[U+FFFD])
    plt.xlabel('x(mm)')
    plt.show()
```

2.2 Implementação das validações

Para verificar o desempenho do código mostrado acima e a validade dos resultados obtidos foram feitas duas funções de validação. Primeiramente, temos a responsável pela validação da implementação do método dos elementos finitos por meio do cálculo da aproximação de acordo com o n fornecido. Calculando as matrizes A e B, para K =1 e q = 0, fazemos a decomposição LU e os αs desejados são o resultado dessa

decomposição. Assim, como é uma validação é encontrado o erro, com a função apresentada anteriormente e então temos a validação do método de acordo com o erro encontrado em relação aos valores obtidos pela decomposição descrita.

```
    def main_validacao(n, plotar): # 4.2
    L = 1
    def k(x):
        return 1
    def q(x):
        return 0
    funcaoselect = 1
    h = L/(n+1)
    x = np.zeros(n+2)
    for i in range(0, n+2, 1):
        x[i] = (i)*h
    print("VETOR X:")
    print(x)
    A, a, b, c = montarMatrizA_k1_q0(n, h, x, k, q)
    print("MATRIZ A (As(a), Am(b), Ai(c)):")
    #print(A)
    print(a)
    print(b)
    print(c)
    B = montarMatrizB(n, x, funcaoselect, h)
    print("MATRIZ B(d):")
    print(B)

    ##DECOMPOSICAO LU
    l, u = ep1.decompLU(a ,b ,c, n)
    alphas = ep1.solucaoLU(l, u, c, B, n)
    print("ALPHAS / Solucao do LU:")
    print(alphas)

    #Calcular erro
    erro = calcularErro(n, funcaoselect, alphas, h, x, 1000)
    print("Erro encontrado: " + str(erro))

    #Plotar erro
    if plotar == 1:
        nvetor = np.linspace(0,63,63, dtype=int)
        errovetor = np.zeros(63)
        for i in range(0,63):
            errovetor[i] = calcularErro(nvetor[i], funcaoselect,
                                        alphas, h, x, 1000)
        print(errovetor)
        plt.plot(nvetor, errovetor)
        plt.title('Erro - Validacao')
        plt.ylabel('Erro')
        plt.xlabel('n')
```

```
        plt.show()
```

A próxima validação feita é com complemento e ela se diferencia da apresentada acima pois o valor de n para determinar a função escolhida é igual a 2 ao invés de 1, isso implica que

$$u(x) = (x-1)\dot{(}e^{-x} - 1) \tag{8}$$

$$f(x) = e^x + 1 \tag{9}$$

ao invés de

$$u(x) = (x^2)\dot{(}(x-1)^2) \tag{10}$$

$$f(x) = (12\dot{x}\dot{(}1-x)) - 2 \tag{11}$$

como é no caso da função anterior onde é feita apenas a validação.

```
    def main_validacao_comp(n, plotar): # 4.2 complemento
    L = 1
    def k(x):
        return math.e**x
    def q(x):
        return 0
    funcaoselect = 2
    h = L/(n+1)
    x = np.zeros(n+2)
    for i in range(0, n+2, 1):
        x[i] = (i)*h
    print("VETOR X:")
    print(x)
    A, a, b, c = montarMatrizA_k1_q0(n, h, x, k, q)
    print("MATRIZ A (As(a), Am(b), Ai(c)):")
    #print(A)
    print(a)
    print(b)
    print(c)
    B = montarMatrizB(n, x, funcaoselect, h)
    print("MATRIZ B(d):")
    print(B)

    ##EXCMS[U+FFFD][U+FFFD] LU
    l, u = ep1.decompLU(a ,b ,c, n)
    alphas = ep1.solucaoLU(l, u, c, B, n)
    print("ALPHAS / Solucao do LU:")
    print(alphas)
```

```
    #Calcular erro
    erro = calcularErro(n, funcaoselect, alphas, h, x, 1000)
    print("Erro encontrado: " + str(erro))

    #Plotar erro
    if plotar == 1:
        nvetor = np.linspace(0,63,63, dtype=int)
        errovetor = np.zeros(63)
        for i in range(0,63):
            errovetor[i] = calcularErro(nvetor[i], funcaoselect,
                                        alphas, h, x, 1000)
        print(errovetor)
        plt.plot(nvetor, errovetor)
        plt.title('Erro - Validacao(Complemento)')
        plt.ylabel('Erro')
        plt.xlabel('n')
        plt.show()
```

2.3 Main final

Assim, com os calculos já prontos e a validação já implementada, temos a main final para que possar ser obtido o que for desejado do código:

```
     if __name__ == "__main__":
    print("EP3")
    print("Insira o valor de n:")
    n = int(input())
    print("Voce quer que os resultados sejam plotados? 1-Sim, 0-
                                        N[U+FFFD]
    plotar = int(input())
    print("O que vc deseja rodar?")
    print("1 - Validacao")
    print("2 - Validacao(Complemento)")
    print("3 - Equilibrio com forcantes de calor com aquecimento e
                                        resfriamento constantes")
    print("4 - Equilibrio com forcantes de calor com aquecimento e
                                        resfriamento com modelo
                                        realista")
    print("5 - Equilibrio com forcantes de calor com dois materiais
                                         no chip")
    select = int(input())
    if  select == 1:
        main_validacao(n, plotar)
    elif select == 2:
        main_validacao_comp(n, plotar)
    elif select == 3:
        main_equilibriocomforcantesdecalor_constante(n, plotar)
    elif select == 4:
```

```
        main_equilibriocomforcantesdecalor(n, plotar)
    elif select == 5:
        print("Insira o raio do chip em milimetros: (Obs: Largura
                                            total = 20mm)")
        d = float(input())/1000
        print("Insira a condutividade do chip(Silicio: 3.6): ")
        ks = float(input())
        print("Insira a condutividade do outro material: ")
        ka = float(input())
        main_doismateriais(n, plotar, ks, ka, d)
```

3 Validação

Com intuito de validar o código e garantir seu funcionamento para aplicar na simulação da temperatura em chips, serão feitos testes para verificar o erro produzido com diversos valores para n.

Nesse primeiro teste iremos encontrar a solução para a seguinte função:

$$f(x) = 12x(1-x) - 2 \tag{12}$$

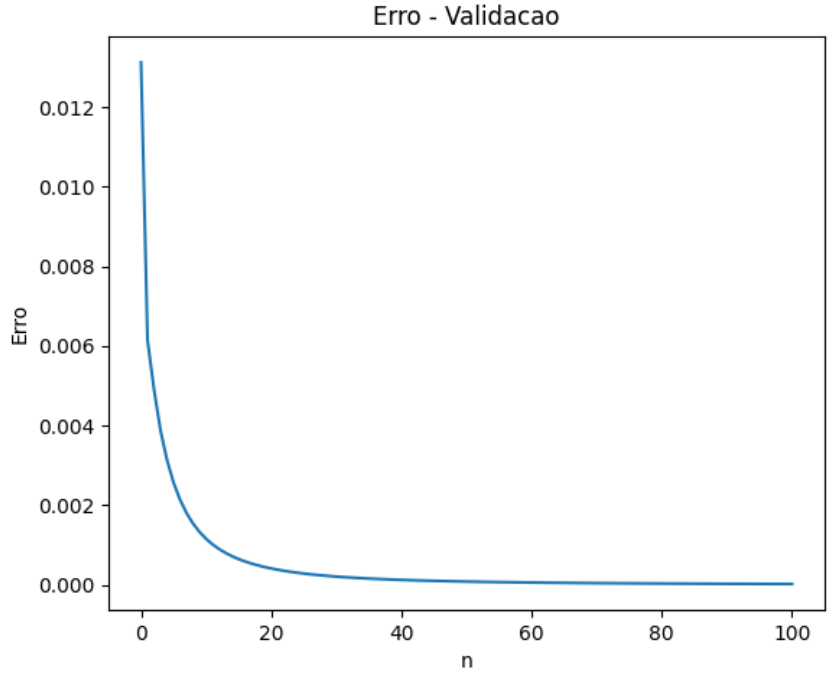
Considerando $k(x) = 1$, $q(x) = 0$, no intervalo $[0, 1]$, com condições de contorno homogêneas.

Para isto então, a solução obtida deve ser: $u(x) = x^2(1-x)^2$

Calculando pelo código e medindo o erro, obtivemos para cada valor de n seguinte o erro correspondente:

n	7	15	31	63
erro	0.002565931956901	0.000801145190387	0.000221620026570	0.0000581402865641

Além disso, também podemos visualizar o erro ocorrendo de forma mais contínua graficamente:



E assim concluímos que o código converge para a função $f(x)$.

4 Validação: Complemento

$$g(x) = e^x + 1 \tag{13}$$

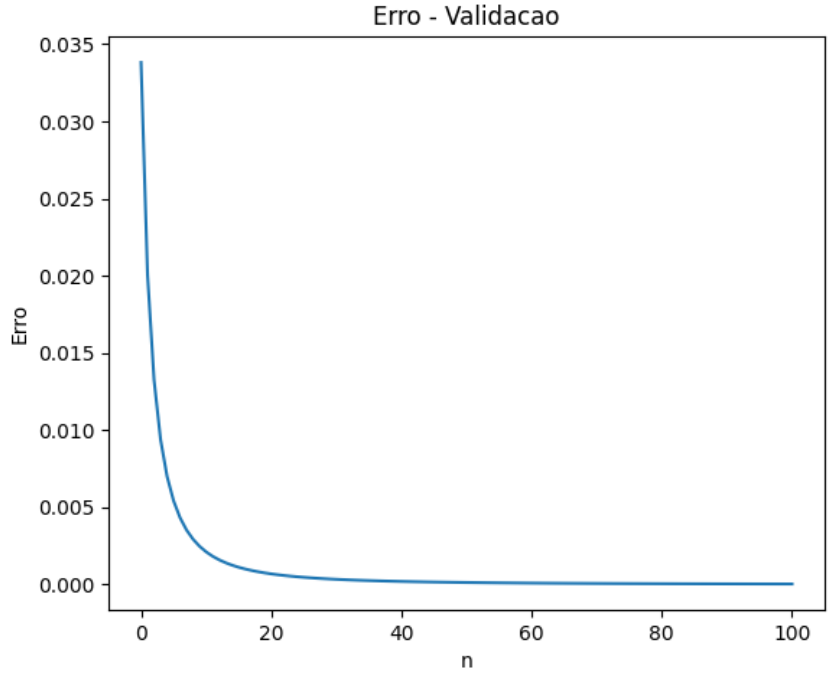
Considerando $k(x) = e^x$, $q(x) = 0$, no intervalo $[0, 1]$, com condições de contorno homogêneas.

Para isto então, a solução obtida deve ser: $u(x) = (x - 1)(e^{-x} - 1)$

Calculando pelo código e medindo o erro, obtivemos para cada valor de n seguinte o erro correspondente:

n	7	15	31	63
erro	0.00542051522377	0.00140903488700	0.00035889723118	0.00009060617292

Além disso, também podemos visualizar o erro ocorrendo de forma mais contínua graficamente:



E assim concluímos que o código converge para a função $g(x)$.

5 Equilíbrio com forçantes de calor

Com o código validado para o método dos elementos finitos, agora serão realizadas simulações de equilíbrio com forçantes de calor. Consideraremos o chip formado de silício, com produção de calor pelo chip e resfriamento externo.

Como o chip esquenta mais no centro que nas bordas, isso pode ser modelado pela seguinte gaussiana:

$$Q_+(x) = Q_+^0 e^{\frac{-(x-L/2)^2}{\sigma^2}} \tag{14}$$

Onde Q_+^0 é a constante que representa o máximo de calor gerado e σ representa a variação do calor ao longo do chip.

O resfriamento pode ser modelado de forma constante:

$$Q_-(x) = Q_+^0(constante) \tag{15}$$

Ou assumindo resfriamento maior nos extremos do chip:

$$Q_-(x) = Q_-^0(e^{-x^2/\theta^2} + e^{-(x-L)^2/\theta^2}) \tag{16}$$

Assim iremos simular a temperatura em regime utilizando os seguintes parâmetros:

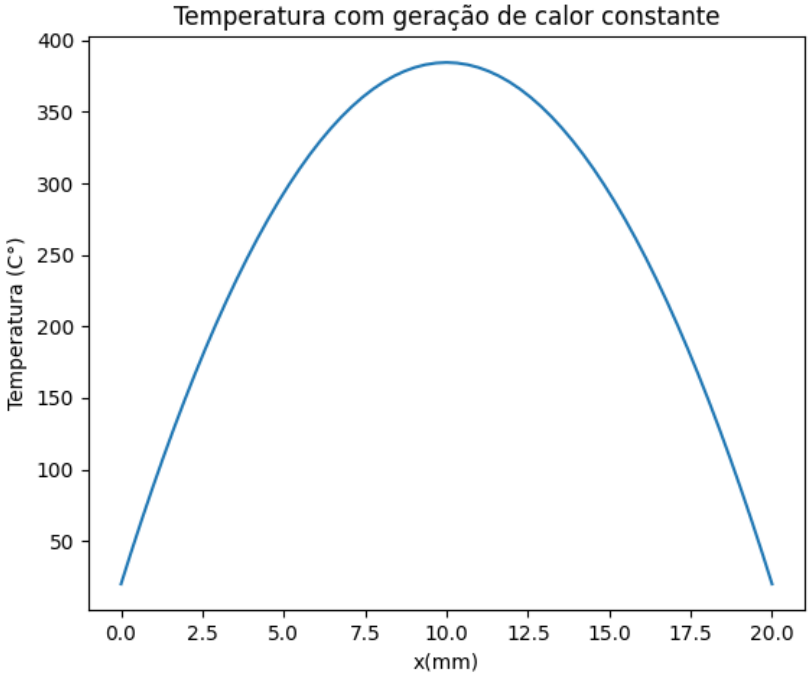
1. Material do chip: silício
2. Densidade: $\rho = 2300kg/m^3$
3. Calor específico: $C = 750J/Kg/K$
4. Condutividade térmica: $k(x) = k = 3,6\frac{W}{mK}$
5. Potência: $P = 30W$
6. Largura: $L = 20mm$
7. Altura: $h = 2mm$
8. Valores máximos: $Q^0_+ = P/V = 37,5 \cdot 10^6$ e $Q^0_- = 0.3Q^0_+ = 11.25 \cdot 10^6$

5.1 Aquecimento e resfriamento constantes

Com um primeiro teste mais simples, utilizando apenas os valores constantes do aquecimento e resfriamento, obtivemos para n=7:

- A temperatura em x = 0.0mm vale: 20.0 °C
- A temperatura em x = 0.0025mm vale: 179.5052083333332 °C
- A temperatura em x = 0.005mm vale: 293.4374999999998 °C
- A temperatura em x = 0.0075mm vale: 361.79687499999955 °C
- A temperatura em x = 0.01mm vale: 384.58333333333303 °C
- A temperatura em x = 0.0125mm vale: 361.7968749999998 °C
- A temperatura em x = 0.015mm vale: 293.43749999999955 °C
- A temperatura em x = 0.0175mm vale: 179.50520833333337 °C
- A temperatura em x = 0.02mm vale: 20.0 °C

Com n = 63 podemos visualizar graficamente a temperatura:



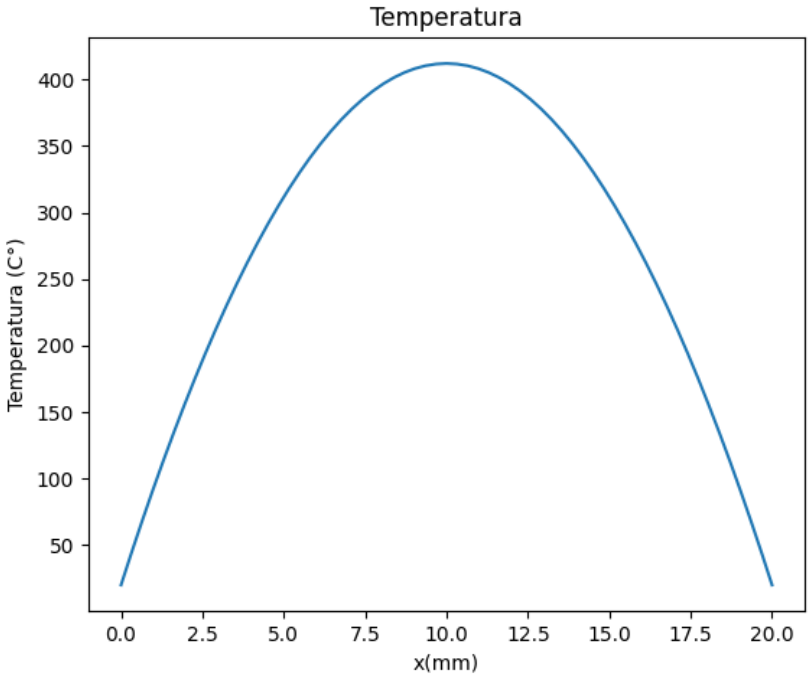
5.2 Aquecimento e resfriamento com modelo real

Agora, testando com um caso mais complexo, utilizando as equações 16 e 14 e utilizando os parâmetros de variação $\sigma = 1$ e $\theta = 0.05$

Vamos ter para n = 7 os valores:

- A temperaatura em x = 0.0mm vale: 20.0 °C
- A temperaatura em x = 0.0025mm vale: 188.8652354789856 °C
- A temperaatura em x = 0.005mm vale: 311.77537630914935 °C
- A temperaatura em x = 0.0075mm vale: 386.84819435650525 °C
- A temperaatura em x = 0.01mm vale: 412.1421795893707 °C
- A temperaatura em x = 0.0125mm vale: 386.8481943565055 °C
- A temperaatura em x = 0.015mm vale: 311.7753763091491 °C
- A temperaatura em x = 0.0175mm vale: 188.86523547898577 °C
- A temperaatura em x = 0.02mm vale: 20.0 °C

Com n = 63 podemos visualizar graficamente a temperatura:



5.3 Dois materiais diferentes no chip

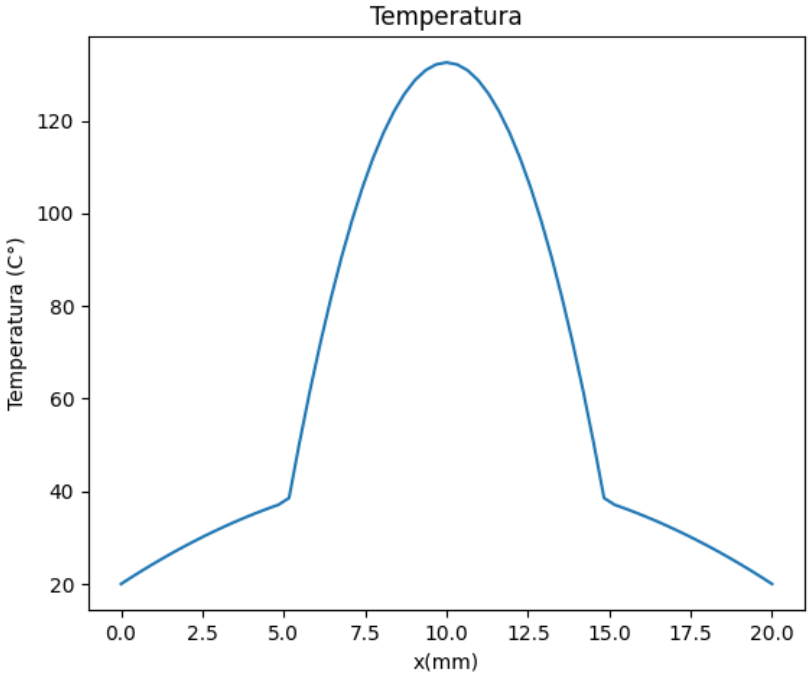
Agora vamos simular o caso onde o chip esta encoberto lateralmente pro outro material, por exemplo, um chip de silício encoberto de alumínio. Para isso utilizamos a função "k(x)"da seguinte forma:

$$k(x) = \begin{cases} k_s, & \text{se } x \in (L/2 - d, L/2 + d) \\ k_a, & \text{caso contrário.} \end{cases} \tag{17}$$

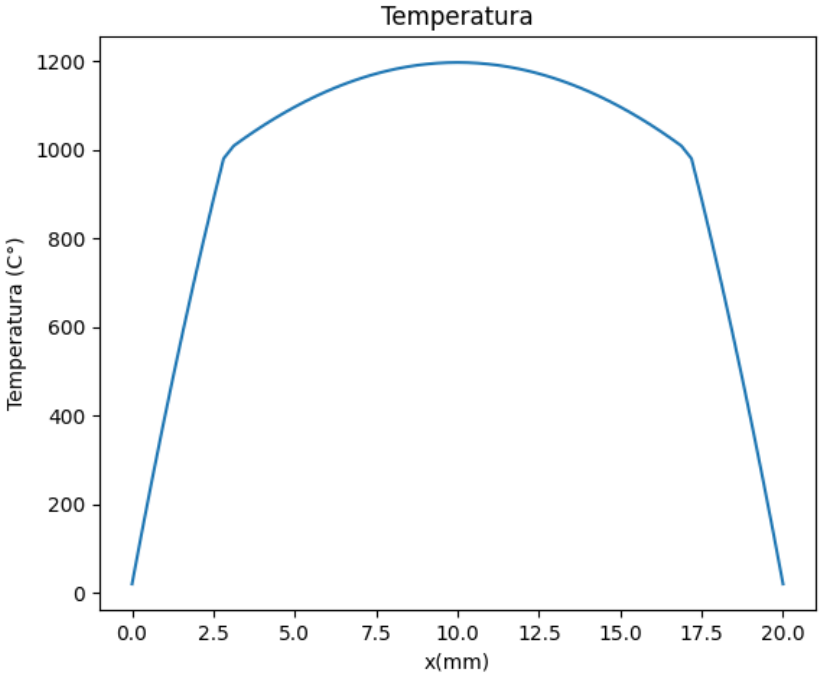
Onde k_s é a condutividade térmica do chip e o k_a do material em volta.

Como um primeiro teste, vamos supor o chip feito de silício de raio 5mm com um revestimento lateral de aluminío: $k_s = 3,6\frac{W}{mK}$, $k_a = 60\frac{W}{mK}$

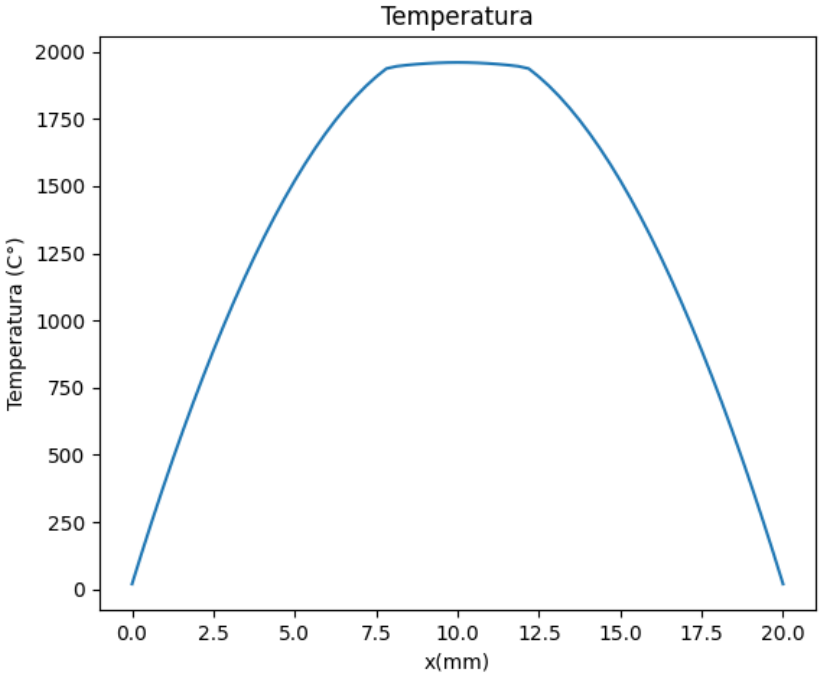
E considerando os mesmos parâmetros do primeiro teste com o modelo realista para a geração e resfriamento, o resultado obtido graficamente para n=61 está na imagem abaixo:



Além desse teste podemos realizar a modelagem de outros exemplos reais, tal como um chip de silício revestido com um polímetro isolante, cuja condutividade térmica vale $0.7\frac{W}{mK}$. Considerando o total sendo de 20mm, e a parte de silício com raio 7mm vamos ter a distribuição térmica da seguinte forma:



Agora realizando o mesmo teste porém com o silício com raio de 2mm, temos:



O que nos leva a conclusão que com um menor comprimento de silício e maior de isolante, temos uma maior temperatura no chip devido a menor dissipação do calor.