ALUNOS

Lucas Oliveira de Paula – 10773680

Victor Nascimento Pereira – 10773530

Título

Relatório sobre um problema inverso para obtenção de distribuição de Temperatura (parte 1)

Versão Original

Relatório apresentado ao Instituto de Matemática e Estatística da Universidade de São Paulo como parte complementar ao EP1 de Métodos Numéricos e Aplicações (MAP-3121)

Área de concentração Métodos Numéricos

São Paulo 2020

Sumário

1. Introdução

Há problemas diretos, onde se determina o efeito da causa, e problemas inversos, onde a partir do efeito observado tenta-se apurar a causa. Este EP resolverá problemas diretos relacionados a equação do calor.

Utilizando de métodos computacionais sendo aplicados através da linguagem Python, e depondo da biblioteca permitida, neste EP, tratou-se a evolução da distribuição de temperatura de uma barra aplicando equações e refinando acerca de novos dados sendo colhido e aplicados sobre suas fontes de calos e distribuição inicial.

2. Listagem de funções

As funções utilizadas no programa estão apresentadas e brevemente explicadas a seguir. Apenas seus propósitos, funcionalidades (de um modo geral) e particularidades/observações são ditas, desta forma, é necessário um conhecimento prévio de linguagem python para entendê-las.

A última função apresentada não consta no programa em si. Ela foi elaborada para realização dos testes pedidos no enunciado do exercício programa.

2.1. Função main

A função main é a porta de entrada do programa. A partir dela, o usuário pode digitar as entradas e escolher as opções para o funcionamento do programa. O seu código pode ser visto a seguir:

```
def main():
    b = 1
    while b == 1 or b == 2 or b == 3:
        print(" \n1 - Teste do metodo 11\n2 - Euler Implicito\
                \n3 - Crank Nicolson\
                \nDigite outro valor inteiro para sair\n")
        b = int(input("Qual teste deseja executar? - "))
        if b == 1 or b == 2 or b == 3:
            n = int(input("Escolha o valor de N: "))
            lmbda = float(input("Escolha o valor de lambda (Para \
                      delta ser igual a \
                  deltax, lambda = n; Para inserir M, lambda = 0): "))
            if lmbda == 0:
                m=int(input("Escolha o valor de M(multiplo de 10): "))
            else:
                m = n ** 2 / lmbda
            if int(m) == m:
                m = m-1
            a = int(input("\nSelecione a função f(t,x) desejada:\
```

```
                    \n1: f(t,x) = 10x²(x-1)-60xt+ 20t\
               \n2: f(t,x) = 10cos(10t)x²(1-x)²-(1 +sin(10t))(12x²-12x+ 2)\
\n3:f(t,x)=25t²e^(t-x) cos(5tx) - 10te^(t-x) sen(5tx) - 5xe^(t-x) sen(5tx)\
                    \n4: f(t,x) = 10000(1-2t²)N (p = 0.25 e gh cte):  "))

                if a != 1 and a != 2 and a != 3 and a != 4:
                    print ("Teste Invalido")
                    break

                x = fazx(n)
                t = fazt(m)

                if b == 1:
                    metodo11 (x,t,a)
                elif b == 2:
                    euler (x,t,a)
                elif b == 3:
                    crank (x,t,a)
            else:
                print("Teste invalido")
```

O usuário pode escolher o método a ser utilizado e após isto, escolhe os valores para de N e λ(lambda), ou se preferir, digitar o valor de M manualmente. Quando se digita o valor de λ e se calcula o valor de M, este pode não ser um número inteiro, por isso, antes de chamar a função que faz o vetor t (tempo), devemos notar se M (a partir de um λ digitado) é um número inteiro. Por isso, esta parte será explicada na próxima função (fazt). Após a escolha da função que será utilizada, o programa chamará as funções que criam os vetores de tempo e espaço, e depois, chamará a função método de resolução escolhida.

2.2. Funções fazt e fazx

```
def fazx(n):
   x=[]
   for i in range(n + 1):
       x.append(i / n)
   return x
def fazt(m):
   t=[]
   for i in range(0, int(m) + 2):
       t.append(i / (int(m)+1))
   return t
```

Como dito anteriormente, deve-se usar um M inteiro. Se caso não for, seja por exemplo M = 102.5, ao usar a função fazt, cria-se um vetor com int(m)+2 = 104 valores (como int(m) é o maior valor inteiro menor que m, devemos somar 2 a int(m) para que o comando for vá de 0 a int(m)+1 = 103, saindo do comando em int(m)+2). Isso tudo garante que a partir do λ digitado, o valor de M inteiro utilizado para o vetor de tempo seja maior que o calculado a partir de N e λ. O que também garante que o novo valor de λ, calculado com N e o novo M (inteiro dessa vez), seja menor que o λ inserido anteriormente (para minimizar os erros ao inserir um λ de próximo a 0.5). Mas e se caso M calculado a partir de N e λ digitados já for um valor inteiro (por exemplo, M = 200) ? Neste caso, é subtraída uma unidade de M, ainda na função main, para que o valor de M seja correto (int(M-1) + 2 = 201, ou seja, vetor tempo de 0 a 200).

A partir destas funções, são criados o vetor x, que vai de 0 a 1 (de 1/N em 1/N) com N+1 valores, e o vetor t, que vai de 0 a 1 (de 1/M em 1/M) com M+1 valores (M inteiro).

2.3. Função metodo11

Após a criação dos vetores, é chamada a função que resolve a malha. Se for escolhida a solução pelo método 11 (denominamos assim por estar desta forma no enunciado do exercício programa), a função respectiva é chamada. São passados os vetores x e t, e o valor de a, que dita a função f(t,x) escolhida. O método 11 é definido como:

$$u_i^{k+1} = u_i^k + \Delta t \left(\frac{u_{i-1}^k - 2u_i^k + u_{i+1}^k}{\Delta x^2} + f(x_i, t_k) \right) \quad (1)$$

A função cria uma matriz u de tamanho M+1 x N+1, com o tempo variando entre linhas e o espaço variando entre as coluna. Para k = 0 (t = 0), as funções têm uma função inicial, da mesma forma para i = 0 (x = 0) e i = N = len(x)-1 (x = 1), onde as funções são determinadas de uma forma. Está é a condição de fronteira. A partir dos valores da primeira linha e primeira e última coluna definidos, para cada função como dito anteriormente, é possível calcular os valores interiores a partir da equação (1). As funções f(t,x), condições de fronteira e funções iniciais (para t=0) serão apresentadas para cada função posteriormente. Nota se que onde há k+1 e k, na equação (1), no programa, foi alterado para k e k-1, respectivamente. Isto é apenas uma troca de notação para adequar-se a linguagem do programa. Este padrão segue nas funções euler e crank também.

```
def metodo11(x, t, a):
    u = []
    linha = []
    conta = 0
    uo = 0
    p = 0.25
    meiodeltax = x[1] / 2
    for k in range(len(t)):
        if k == 0:
            if a == 1 or a == 4:
                linha.append([0] * len(x))
                u = linha
                plt.plot(linha, 'k', color='orange')
            elif a == 2:
                for j in range(len(x)):
                    uo = x[j] * x[j] * (1 - x[j]) * (1 - x[j])
                    linha.append(uo)
                u.append(linha)
                plt.plot(linha, 'k', color='orange')
            elif a == 3:
                for j in range(len(x)):
                    uo = math.exp(-x[j])
                    linha.append(uo)
                u.append(linha)
                plt.plot(linha, 'k', color='orange')
            else:
                print("f(t,x) invalida")
                break
        else:
            linha = []
            if a == 1 or a == 2 or a == 4:
                linha.append(0)
            elif a == 3:
                g1 = math.exp(t[k])
                linha.append(g1)
            for i in range(1, len(x) - 1):
                if a == 1:
                    f = (10 * x[i] ** 2 * (x[i] - 1)
                         - 60 * x[i] * t[k - 1]
                         + 20 * t[k - 1])
                elif a == 2:
                    b = (10 * math.cos(10 * t[k-1])
```

```
                    * x[i] * x[i] * (1 - x[i])
                    * (1 - x[i]))
                c = (1 + math.sin(10 * t[k -1])) \
                    * (12 * x[i] * x[i] - 12 * x[i] + 2)
                f = b - c
            elif a == 3:
                b = 25*t[k-1]*t[k-1] * (math.exp(t[k-1]-x[i])) \
                    * (math.cos(5*t[k-1]*x[i]))
                c = 10*t[k-1] * (math.sin(5*t[k-1]*x[i]))
                d = 5*x[i] * (math.exp(t[k-1]*x[i])) *
                    (math.sin(5*t[k-1]*x[i]))
                f = b - c - d
            elif a == 4:
                if x[i] <= (p + meiodeltax) and x[i] >= (p- meiodeltax):
                    f = 10000 * (1 - 2 * t[k - 1] * t[k - 1]) * (1/x[1])
                elif x[i] < (p - meiodeltax) or x[i] > (p + meiodeltax):
                    f = 0
            else:
                print("f(t,x) invalida")
                break
            conta = u[k - 1][i] + (1 / (len(t) - 1)) * (
                    ((u[k - 1][i - 1] - 2 * u[k - 1][i] + u[k-1][i+1])
                    / ((1 / (len(x) - 1)) ** 2)) + f)
            linha.append(conta)
        if a == 1 or a == 2 or a == 4:
            linha.append(0)
        elif a == 3:
            g2 = math.exp(t[k] - 1) * math.cos(5 * t[k])
            linha.append(g2)
        u.append(linha)
calculatrunc(t, x, u, a,1)
```

2.4. Função euler

Para o segundo método de resolução, a matriz u possui funções para valor inicial e para fronteira (dadas por $g_1(t)$ e $g_2(t)$) (igualmente ao método anterior, pois é uma característica da função problema, e não do modo de resolução). Desta forma, a função euler calcula os pontos interiores da matriz, da mesma forma que a função método 11, porém da seguinte forma:

$$u_i^{k+1} = u_i^k + \lambda\left(u_{i-1}^{k+1} - 2u_i^{k+1} + u_{i+1}^{k+1}\right) + \Delta t f\left(x_i, t_{k+1}\right) \quad (2)$$

Está função nos dá um sistema linear com a seguinte matriz:

$$\begin{bmatrix} 1+2\lambda & -\lambda & 0 & \cdots & 0 \\ -\lambda & 1+2\lambda & -\lambda & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -\lambda & 1+2\lambda & -\lambda \\ 0 & \cdots & 0 & -\lambda & 1+2\lambda \end{bmatrix} \begin{bmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{N-2}^{k+1} \\ u_{N-1}^{k+1} \end{bmatrix} = \begin{bmatrix} u_1^k + \Delta t f_1^{k+1} + \lambda g_1(t^{k+1}) \\ u_2^k + \Delta t f_2^{k+1} \\ \vdots \\ u_{N-2}^k + \Delta t f_{N-2}^{k+1} \\ u_{N-1}^k + \Delta t f_{N-1}^{k+1} + \lambda g_2(t^{k+1}) \end{bmatrix} \tag{3}$$

Para isso, devemos desenvolver as matrizes de coeficientes (matriz A) e o vetor de termos independentes (vetor b). Como a matriz A possui apenas a diagonal principal e uma diagonal secundária (duas na verdade, mas são iguais), podemos representar a matriz A apenas por um vetor chamado “principal”, de tamanho N-1, e um vetor chamado “secundaria”, de tamanho N-2. Para resolução deste sistema linear, utiliza-se duas outras funções, a ldl e a resolvex, que serão explicadas posteriormente.

```
def euler (x, t, a):
    n = len(x) - 1
    m = len(t) - 1
    lmbda = n*n/m
    u = []
    linha = []
    principal = []
    secundaria = []
    b = []
    vetorx = []
    uo = 0
    p = 0.25
    meiodeltax = x[1] / 2

    for j in range (0,n-1):      # Matriz A
        principal.append( 1 + 2*lmbda )
        if j != n-2:
            secundaria.append( -lmbda )
    for k in range(0, m+1):
        b = []
        vetorx = []
        if k == 0:
            if a == 1 or a == 4:
                linha.append([0] * (n+1))
                u = linha
                plt.plot(linha, 'k', color='orange')
            elif a == 2:
                for j in range(n+1):
                    uo = x[j] * x[j] * (1 - x[j]) * (1 - x[j])
```

```
                linha.append(uo)
            u.append(linha)
            plt.plot(linha, 'k', color='orange')
        elif a == 3:
            for j in range(n+1):
                uo = math.exp(-x[j])
                linha.append(uo)
            u.append(linha)
            plt.plot(linha, 'k', color='orange')

    else:
        linha = []
        if a == 1 or a == 2 or a == 4:
            g1 = 0
            linha.append(g1)
        elif a == 3:
            g1 = math.exp(t[k])
            linha.append(g1)
        if a == 1 or a == 2 or a == 4:
            g2 = 0
        elif a == 3:
            g2 = math.exp(t[k] - 1) * math.cos(5 * t[k])

        for i in range (1, n):                          # Matriz B
            if a == 1:
                f = (10 * x[i] ** 2 * (x[i] - 1) - 60 * x[i] * t[k] +
                    20 * t[k])
            elif a == 2:
                c = (10 * math.cos(10 * t[k]) * x[i] * x[i] * (1 - x[i])
                    * (1 - x[i]))
                d = (1 + math.sin(10 * t[k])) * (12 * x[i] * x[i] - 12
                    * x[i] + 2)
                f = c - d
            elif a == 3:
                c = 25*t[k]*t[k] * (math.exp(t[k]-x[i])) *
                    (math.cos(5*t[k]*x[i]))
                d = 10*t[k] * (math.sin(5*t[k]*x[i]))
                e = 5*x[i] * (math.exp(t[k]*x[i])) *
                    (math.sin(5*t[k]*x[i]))
                f = c - d - e
            elif a == 4:
                if x[i] <= (p + meiodeltax) and x[i] >= (p- meiodeltax):
```

```
                    f = 10000 * (1 - 2 * t[k] * t[k]) * (1/x[1])
                elif x[i] < (p - meiodeltax) or x[i] > (p + meiodeltax):
                    f = 0
            if i == 1:
                valorb = u[k-1][i] + ( f / m ) + lmbda * g1
            elif i == (n-1):
                valorb = u[k-1][i] + ( f / m ) + lmbda * g2
            else:
                valorb = u[k-1][i] + ( f / m )
            b.append( valorb )
        l,d = ldl(principal, secundaria, n)
        vetorx = resolvex (l, d, n, b)
        for j in range(0, len(vetorx)):
            linha.append(vetorx[j])
        linha.append(g2)
        u.append(linha)
calculatrunc(t,x,u,a,2)
```

2.5. Função crank

Igualmente aos outros dois métodos de resolução, a matriz u neste método possui funções para valor inicial e para fronteira (dadas por $g_1(t)$ e $g_2(t)$). Desta forma, a função crank calcula os pontos interiores da matriz, da mesma forma que as funções anteriores, mas da forma:

$$u_i^{k+1}=u_i^k+\frac{\lambda}{2}\left(\left(u_{i-1}^{k+1}-2u_i^{k+1}+u_{i+1}^{k+1}\right)+\left(u_{i-1}^k-2u_i^k+u_{i+1}^k\right)\right)+\frac{\Delta t}{2}\left(f\left(x_i,t_{k+1}\right)+f\left(x_i,t_k\right)\right) \quad (4)$$

Está função nos dá um sistema linear com a seguinte matriz:

$$\begin{bmatrix} 1+\lambda & \frac{-\lambda}{2} & 0 & \dots & 0 \\ \frac{-\lambda}{2} & 1+\lambda & \frac{-\lambda}{2} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \frac{-\lambda}{2} & 1+\lambda & \frac{-\lambda}{2} \\ 0 & \dots & 0 & \frac{-\lambda}{2} & 1+\lambda \end{bmatrix} \begin{bmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{N-2}^{k+1} \\ u_{N-1}^{k+1} \end{bmatrix} = \begin{bmatrix} u_1^k+\left(u_0^k-2u_1^k+u_2^k\right)+\frac{\Delta t}{2}\left(f_1^k+f_1^{k+1}\right)+\frac{\lambda}{2}g_1^{k+1} \\ u_2^k+\left(u_1^k-2u_2^k+u_3^k\right)+\frac{\Delta t}{2}\left(f_2^k+f_2^{k+1}\right) \\ \vdots \\ u_{N-2}^k+\left(u_{N-3}^k-2u_{N-2}^k+u_{N-1}^k\right)+\frac{\Delta t}{2}\left(f_{N-2}^k+f_{N-2}^{k+1}\right) \\ u_{N-1}^k+\left(u_{N-2}^k-2u_{N-1}^k+u_N^k\right)+\frac{\Delta t}{2}\left(f_{N-1}^k+f_{N-1}^{k+1}\right)+\frac{\lambda}{2}g_2^{k+1} \end{bmatrix} \quad (5)$$

Para isso, devemos desenvolver as matrizes de coeficientes (matriz A) e o vetor de termos independentes (vetor b), da mesma forma que foi feito no método Euler Implícito. Para resolução deste sistema linear também utiliza-se duas outras funções, a ldl e a resolvex.

```
def crank (x, t, a):
    n = len(x) - 1
    m = len(t) - 1
    lmbda = n*n/m
    u = []
    linha = []
    principal = []
    secundaria = []
    b = []
    vetorx = []
    uo = 0
    p = 0.25
    meiodeltax = x[1] / 2
    for j in range (0,n-1):                          # Matriz A
        principal.append( 1 + lmbda )
        if j != n-2:
            secundaria.append( -lmbda/2 )
    for k in range(0, m+1):
        b = []
        vetorx = []
        if k == 0:
            if a == 1 or a == 4:
                linha.append([0] * (n+1))
                u = linha
                plt.plot(linha, 'k', color='orange')
            elif a == 2:
                for j in range(n+1):
                    uo = x[j] * x[j] * (1 - x[j]) * (1 - x[j])
                    linha.append(uo)
                u.append(linha)
                plt.plot(linha, 'k', color='orange')
            elif a == 3:
                for j in range(n+1):
                    uo = math.exp(-x[j])
                    linha.append(uo)
                u.append(linha)
                plt.plot(linha, 'k', color='orange')  "Tenta"""
        else:
            linha = []
            if a == 1 or a == 2 or a == 4:
                g1 = 0
                linha.append(g1)
```

```
        elif a == 3:
            g1 = math.exp(t[k])
            linha.append(g1)
        if a == 1 or a == 2 or a == 4:
            g2 = 0
        elif a == 3:
            g2 = math.exp(t[k] - 1) * math.cos(5 * t[k])
        for i in range (1, n):                          # Matriz B
            if a == 1:
                f1 = (10 * x[i] ** 2 * (x[i] - 1) - 60 * x[i] * t[k-1]
                      + 20 * t[k-1])
                f2 = (10 * x[i] ** 2 * (x[i] - 1) - 60 * x[i] * t[k]
                      + 20 * t[k])
                f = f1 + f2
            elif a == 2:
                c1 = (10 * math.cos(10 * t[k-1]) * x[i] * x[i]
                      * (1 - x[i]) * (1 - x[i]))
                d1 = (1 + math.sin(10 * t[k-1])) *
                      (12 * x[i] * x[i] - 12 * x[i] + 2)
                f1 = c1 - d1
                c2 = (10 * math.cos(10 * t[k]) * x[i] * x[i] *
                      (1 - x[i]) * (1 - x[i]))
                d2 = (1 + math.sin(10 * t[k])) *
                      (12 * x[i] * x[i] - 12 * x[i] + 2)
                f2 = c2 - d2
                f = f1 + f2
            elif a == 3:
                b1 = 25*t[k]*t[k] * (math.exp(t[k]-x[i])) *
                       (math.cos(5*t[k]*x[i]))
                c1 = 10*t[k] * (math.sin(5*t[k]*x[i]))
                d1 = 5*x[i] * (math.exp(t[k]*x[i])) *
                      (math.sin(5*t[k]*x[i]))
                f1 = b1 - c1 - d1
                b2 = 25*t[k-1]*t[k-1] * (math.exp(t[k-1]-x[i])) *
                      (math.cos(5*t[k-1]*x[i]))
                c2 = 10*t[k-1] * (math.sin(5*t[k-1]*x[i]))
                d2 = 5*x[i] * (math.exp(t[k-1]*x[i])) *
                      (math.sin(5*t[k-1]*x[i]))
                f2 = b2 - c2 - d2
                f = f1 + f2
            elif a == 4:
                if x[i] <= (p + meiodeltax) and x[i] >= (p -meiodeltax):
```

```
                    f1 = 10000 * (1 - 2 * t[k-1] * t[k-1]) * (1/x[1])
                    f2 = 10000 * (1 - 2 * t[k] * t[k]) * (1/x[1])
                    f = f1 + f2
                elif x[i] < (p - meiodeltax) or x[i] > (p + meiodeltax):
                    f = 0
            if i == 1:
                valorb = ((lmbda/2) *
                          (u[k-1][i-1]-2*u[k-1][i]+u[k-1][i+1])) +
                           (u[k-1][i]) + (f/(2*m)) + ((lmbda/2)*g1)
            elif i == (n-1):
                valorb=((lmbda/2)*(u[k-1][i-1]-2*u[k-1][i]+u[k-1][i+1]))
                        + (u[k-1][i]) + (f/(2*m)) + ((lmbda/2)*g2)
            else:
                valorb=((lmbda/2)*(u[k-1][i-1]-2*u[k-1][i]+u[k-1][i+1]))
                        + (u[k-1][i]) + (f/(2*m))
            b.append( valorb )
        l,d = ldl(principal, secundaria, n)
        vetorx = resolvex (l, d, n, b)
        for j in range(0, len(vetorx)):
            linha.append(vetorx[j])
        linha.append(g2)
        u.append(linha)
calculatrunc(t, x, u, a,3)
```

2.6. Função resolvex

Para os métodos Euler Implícito e Crank-Nicolson, é preciso resolver um sistema de equações lineares. As matrizes do sistema foram apresentadas nas funções referentes a cada método. Para a resolução desse sistema linear ser facilitada, a matriz dos coeficientes das variáveis, que é tridiagonal simétrica (matriz A , transformada nos vetores principal e secundaria), é transformada em um produto de 3 matrizes diagonais ($A = LDL^t$, L é bidiagonal triangular unitária inferior e D diagonal). A função ldl é capaz de fazer essa transformação, retornando os vetores l e d, que representam as matrizes L e D, por L ser composta por apenas duas diagonais iguais e D ser composta apenas por uma diagonal. Com os vetores l e d, a solução (x), é encontrada resolvendo primeiro Ly = b; com o vetor y em mãos, faz-se Dz = y; e por fim, $L^tx = z$. O vetor x é retornado para as funções euler e crank, para comporem o interior da matriz u a cada passo de k.

```
def resolvex (l, d, n, b):
    y = []
    z = []
    x = []
```

```
    a = 0
    for i in range(0,n-1):
        x.append(i)
    y.append(b[0])
    for i in range(1,n-1):
        a = b[i] - l[i-1]*y[i-1]
        y.append(a)
    for i in range (0,n-1):
        a = y[i]/d[i]
        z.append(a)
    x[n-2] = z[n-2]
    for i in range(1, n-1):
        x[n-2-i] = z[n-2-i] - l[n-2-i]*x[n-1-i]
    return x
```

2.7. Função ldl

A função ldl transforma os vetores que representam a matriz A (principal e secundaria) em outros dois vetores, l e d. Com o fim de facilitar a resolução do sistema linear calculado a cada passo de k.

A transformação de matrizes (consequentemente de vetores correspondentes) é proveniente da igualdade:

$$A = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & 0 \\ 0 & 0 & l_{N-1} & 1 \end{bmatrix} \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & d_{N-1} \end{bmatrix} \begin{bmatrix} 1 & l_2 & 0 & 0 \\ 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & l_{N-1} \\ 0 & \cdots & 0 & 1 \end{bmatrix} \quad (6)$$

```
def ldl(principal, secundaria, n):
    l = []
    d = []
    d.append( principal[0] )
    l.append( secundaria[0]/d[0] )
    for i in range (1,n-1):
        d.append( principal[i] - l[i-1]*l[i-1]*d[i-1] )
        if len(secundaria) > 1 and i <(n-2):
            l.append( secundaria[i]/d[i] )
    return (l,d)
```

2.8. Função calculatrunc

Esta função calcula o erro máximo de truncamento na matriz u, para N, M, função e método dados. A variável b (recebida de cada função método de

resolução) diz qual a equação do erro de truncamento para cada k e i da matriz u. Para o método 11, o erro de truncamento é definido por:

$$\tau_i^k(\Delta t, \Delta x) = \frac{u(t_{k+1}, x_i) - u(t_k, x_i)}{\Delta t} - \frac{u(t_k, x_{i-1}) - 2u(t_k, x_i) + u(t_k, x_{i+1})}{\Delta x^2} - f(x_i, t_k) \quad (7)$$

Da mesma forma, o erro de truncamento do método Euler implícito é dado por:

$$\tau_i^k(\Delta t, \Delta x) = \frac{u(t_{k+1}, x_i) - u(t_k, x_i)}{\Delta t} - \frac{u(t_{k+1}, x_{i-1}) - 2u(t_{k+1}, x_i) + u(t_{k+1}, x_{i+1})}{\Delta x^2} - f(x_i, t_{k+1}) \quad (8)$$

E por fim, o erro de truncamento do método Crank-Nicolson foi calculado por (com outra notação, para ser possível vizualiza-la):

$$\tau_1^{k+1}(\Delta t, \Delta x) = \frac{u_i^{k+1} - u_i^k}{\Delta t} - \frac{u_{i-1}^k - 2u_i^k + u_{i+1}^k + u_{i-1}^{k+1} - 2u_i^{k+1} + u_{i+1}^{k+1}}{\Delta x^2} - (f_i^{k+1} + f_i^k) \quad (9)$$

```
def calculatrunc(t, x, u, a, b):
    maior = 0
    p = 0.25
    meiodeltax = x[1] / 2

    for k in range(0, len(t) - 1):
        for i in range(1, len(x) - 1):
            e1 = (len(t) - 1) * (u[k + 1][i] - u[k][i])
            e2 =  (len(x) - 1) * (len(x) - 1) * (u[k][i - 1] - 2 * u[k][i]
                  + u[k][i + 1])
            e2k =  (len(x) - 1) * (len(x) - 1) * (u[k+1][i - 1] - 2
                   * u[k+1][i] + u[k+1][i + 1])

            if a == 1:
                f = 10* x[i]* x[i]* (x[i]-1) - 60* x[i]* t[k] + 20* t[k]
                fk =10* x[i]* x[i]* (x[i]-1) - 60* x[i]* t[k+1] + 20* t[k+1]
            elif a == 2:
                b = 10* math.cos(10* t[k])* x[i]* x[i]* (1-x[i])* (1-x[i])
                c = (1 + math.sin(10 * t[k])) * (12 * x[i] * x[i] - 12
                     * x[i] + 2)
                f = b - c
                bk = 10 * math.cos(10 * t[k+1]) * x[i] * x[i] * (1 - x[i])
                        * (1 - x[i])
                ck = (1 + math.sin(10 * t[k+1])) * (12 * x[i] * x[i] - 12
                      * x[i] + 2)
                fk = bk - ck
            elif a == 3:
```

```
                b = 25* t[k]* t[k]* (math.exp(t[k]-x[i])) *
                      (math.cos(5*t[k]*x[i]))
                c = 10*t[k] * (math.sin(5*t[k]*x[i]))
                d = 5*x[i] * (math.exp(t[k]*x[i])) * (math.sin(5*t[k]*x[i]))
                f = b - c - d
                bk = 25* t[k+1]* t[k+1]* (math.exp(t[k+1]-x[i])) *
                       (math.cos(5*t[k+1]*x[i]))
                ck = 10*t[k+1] * (math.sin(5*t[k]*x[i]))
                dk = 5* x[i] * (math.exp(t[k+1]*x[i])) *
                       (math.sin(5*t[k+1]*x[i]))
                fk = bk - ck - dk
            elif a == 4 and x[i] >= (p - meiodeltax) and x[i] <= (p +
                                                                  meiodeltax):
                f = 10000 * (1 - 2 * t[k] * t[k]) * (1/x[1])
                fk = 10000 * (1 - 2 * t[k+1] * t[k+1]) * (1/x[1])
            elif a == 4 and x[i] < (p - meiodeltax) or x[i] > (p +
                                                                  meiodeltax):
                f = 0
                fk = 0
            if b == 2:
                e2 = e2k
            elif b == 3:
                e2 = e2 +e2k
                f = f+fk
            e = e1 - e2 - f
            if abs(e) > maior:
                maior = e
                posk = k
                posi = i
    print ("\nDADOS:")
 print("\nMaior erro de truncamento = ",f'{abs(maior):.2e}'," ,em t= ",
f'{abs(t[posk]):.2e}'," e x= ", f'{abs(x[posi]):.2e}')


    erroreal(t,x,u,a,b)
```

2.9. Função erroreal

Esta função tem por objetivo calcular o maior erro real entre os diversos valores de t e x , ou seja, a diferença entre a solução analítica u(t,x) e a matriz aproximada, dada por u. A função recebe o valor de a, variável que identifica a solução analítica a ser utilizada. Há uma particularidade para a = 4: esta função não possui solução analitica, desta forma, um erro é calculado, para cada k e i da matriz, da forma:

$$e_i^{k+1} = e_i^k + \Delta t(\frac{e_{i-1}^k - 2e_i^k + e_{i+1}^k}{\Delta x^2} + \tau_i^k)) \; . \tag{10}$$

Note que é necessário o erro de truncamento para calcular este erro, logo, a variável b, recebida pela função erroreal, é utilizada para definir o tipo do erro de truncamento (Método 11, Euler Implícito ou Crank-Nicolson). Com o erro encontrado para um dado k e i, calcula-se o erro real, obtido numericamente, somando u_i^{k+1} com e_i^{k+1}.

Além de encontrar os maiores erros obtidos (um em t = 1s e outro em qualquer outro tempo), a função também cria uma matriz (usol), de tamanho 3 x N+1, que armazena os valores da solução u(t,x) (solução real numérica para a função a = 4) para t = 0, t = 0.5 e t = 1. Esta matriz é criada para qualquer método ou função utilizada, e com ela, é chamada a função que desenvolve o gráfico de comparação entre a solução real analítica e a solução numérica. Outra funcionalidade, é a função que traça o gráfico da solução numérica com curvas de intervalos de 0.1 (de 0 a 1). Após o usuário decidir o gráfico que deseja ver, as funções que desenham o gráfico são chamadas com dois parâmetros em especial: maiorureal+margem e menorureal+margem. Estes parâmetros são definidos para ajuste da escala dos gráficos. As curvas são traçadas em função de i (de 0 a N), lembrando que a barra tem comprimento unitário, ou seja, x é definido por x = i/N (x[i] = i/N, para o vetor x). Ou seja, u_i^{k+1} é correspondente a u(k/M, i/N) (solução real).

```
def erroreal(t, x, u, a, b):
    erro = []
    linhaerro = []
    maior = 0
    menor = 0
    maiort1 = 0
    p = 0.25
    meiodeltax = x[1] / 2
    maiorureal = 0
    menorureal = 0
    usollinha = []
    usol = []
    n = len(x)-1
    m = len(t)-1
    div = m / 2
    lmbda = n*n/m

    if a == 4:
        for k in range (0, len(t)-1):
            usollinha = []
            if k == 0:
```

```
            linhaerro.append([0] * len(x))
            erro = linhaerro
            usollinha.append ([0] * len(x))
            usol = usollinha
        elif k>0:
            linhaerro = []
            linhaerro.append(0)
            for i in range(1, len(x) - 1):
                e1 = (len(t) - 1) * (u[k + 1][i] - u[k][i])
                e2 = (len(x) - 1) * (len(x) - 1) * (u[k][i - 1] - 2 *
                      u[k][i] + u[k][i + 1])
                e2k = (len(x) - 1) * (len(x) - 1) * (u[k+1][i - 1] - 2
                       * u[k+1][i] + u[k+1][i + 1])
                if x[i] >= (p - meiodeltax) and x[i] <= (p +
                                                        meiodeltax):
                    f = 10000 * (1 - 2 * t[k] * t[k]) * (1/x[1])
                    fk = 10000 * (1 - 2 * t[k] * t[k+1]) * (1/x[1])
                elif x[i] < (p - meiodeltax) or x[i] > (p + meiodeltax):
                    f = 0
                    fk = 0
                if b == 1:
                    e2 = e2k
                if b == 2:
                    e2 = e2k + e2
                    f = f + fk
                errotrunc = e1 - e2 - f
                e = erro[k-1][i] + (1 / (len(t)-1))* ((erro[k-1][i-1]
                     - 2* erro[k-1][i]+ erro[k-1][i+1])* (len(x)-1)*
                       (len(x)-1) + errotrunc)
                linhaerro.append(e)
                ureal = e + u[k][i]

                if k/div == int(k/div) or k == len(t)-2:
                    if i == 1:
                        usollinha.append (0)
                    usollinha.append(ureal)
                    if i == len(x)-2:
                        usollinha.append(0)
                if ureal > maiorureal:
                    maiorureal = ureal
                if ureal < menorureal:
                    menorureal = ureal
```

```
                if e > maior:
                    maior = e
                if k == (len(t) - 1) and e > maiort1:
                    maiort1 = e
                linhaerro.append(0)
            if k/div == int(k/div) or k == len(t)-2:
                usol.append(usollinha)
            erro.append(linhaerro)
else:
    for k in range(0, len(t)):
        usollinha = []
        for i in range(0, len(x)):
            if a == 1:
                ureal = 10 * t[k] * x[i] * x[i] * (x[i] - 1)
            elif a == 2:
                ureal = ((1 + math.sin(10 * t[k]))* x[i]* x[i]*
                         (1 - x[i]) * (1 - x[i]))
            elif a == 3:
                ureal = (math.exp(t[k]- x[i]))* (math.cos(5* t[k]*x[i]))
            e = ureal - u[k][i]
            if ureal > maiorureal:
                maiorureal = ureal
            if ureal < menorureal:
                menorureal = ureal
            if e > maior:
                maior = e
            if e < menor:
                menor = e
            if k == len(t) - 1 and e > maiort1:
                maiort1 = e
            if k/div == int(k/div):
                usollinha.append(ureal)
        if k/div == int(k/div):
            usol.append(usollinha)
margem = (maiorureal - menorureal)/50
print("Maior erro absoluto =", max(f'{abs(maior):.2e}',
                                   f'{abs(menor):.2e}'))
print("Maior erro absoluto em (t=1) = ", f'{abs(maiort1):.2e}')

print ("\nQual gráfico deseja ver? -")
print ("1 - Gráfico da solução numérica comparada com ureal")
b = int(input("2 - Gráfico da solução numérica e sua evolução temporal\
```

```
                \nQual deseja plotar? - "))
  if b == 1:
      plotadorreal(t, u, usol, n, lmbda, maiorureal+margem,
                                                    menorureal-margem)
  if b == 2:
      plotadortemporal(t, u, n, lmbda, maiorureal+margem,
                                                  menorureal-margem )
```

2.10. Função plotadorreal

Com a matriz usol recebida, a função tem a funcionalidade de apenas traçar o gráfico com a função real analitíca(também numérica para a função 4) e a função numérica. Lembrando que como foi definido que usol teria 3 linhas, a função desenha a curva da solução para t=0, t=0.5 e t=1. Como a solução numérica é exata para t = 0, não é necessário traçar esta curva, assim, apenas as outras duas curvas são traçadas.

```
def plotadorreal(t, u, usol, n, lmbda, maior, menor):
  m = len(u)-1
  div= m / (len(usol)-1)
  for i in range (0, len(u), int(div)):
      a = int (i/div)
      if a == 0:
          plt.plot(usol[a], label='u real t=%.2f'% t[i] ,color='black')
          plt.legend()
      elif a == 1:
          plt.plot(u[i], label='u aproximado t=%.2f'% t[i],color='red')
          plt.legend()
          plt.plot(usol[a], label='u real t=%.2f'% t[i] ,color='green')
          plt.legend()
      elif a == 2:
          plt.plot(u[i], label='u aproximado t=%.2f'% t[i],color='blue')
          plt.legend()
          plt.plot(usol[a], label='u real t=%.2f'% t[i] ,color='pink')
          plt.legend()
  plt.axis([0, n, menor, maior])
  plt.title("Grafico para N= %d e lambda= %.2f " % (len(u[0])-1, lmbda))

  plt.grid(True)
  plt.xlabel("Barra")

  plt.ylabel("Temperatura")
  plt.show()
```

2.11. Função plotadortemporal

Por fim, há a função que desenha o gráfico que mostra a evolução temporal da solução numérica obtida. Como foi dito, esta função traça as curvas em intervalos de 0.1s em 0.1s, ou seja, em intervalos de M/10 em M/10 (de 0 a M). Desta forma, é preciso que o M utilizado seja um valor múltiplo de 10 para ser possível visualizar este gráfico.

```
def plotadortemporal (t, u, n, lmbda, maior, menor):
   m = len(u)-1
   div= m / 10
   print ('\n1 - De 0 a 0.5s (de 0.1s em 0.1s)')
   print ('2 - De 0.6 a 1s (de 0.1s em 0.1s)')
   y = int(input("Em que intervalo deseja ver o gráfico temporal?  "))

   if y == 1:
       for i in range (0, 6*int(div), int(div)) :
           if i == 0:
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='red')
           elif i == int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='orange')
           elif i == 2*int(div) or i == 7*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='olive')
           elif i == 3*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='yellow')
           elif i == 4*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='green')
           elif i == 5*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='cyan')
   elif y == 2:
       for i in range (int (len(u)/2), len(u), int(div)) :
           if i == 6*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='blue')
           elif i == 7*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='purple')
           elif i == 8*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='magenta')
           elif i == 9*int(div):
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='brown')
           elif i == m:
               plt.plot(u[i], label='u para t=%.2f'% t[i] ,color='black')
   plt.legend()
```

```
    plt.axis([0, n, menor, maior])
    plt.title("Grafico para N= %d e lambda= %.2f com intervalos de 0.1s" %
                                                    (len(u[0])-1, lmbda))

    plt.grid(True)
    plt.xlabel("Barra")

    plt.ylabel("Temperatura")
    plt.show()
```

2.12. Função main (teste.py)

O código seguinte não consta no programa desenvolvido para cálculo da solução numérica, mas utilizamos ele para realização dos testes para os valores de N e λ pedidos.

```
    def main():

        a=int(input('1-Calcula lamba\n2-Faz Matriz\n3-Integra\
            \n4-Tarefa 1_A Integrais\n5-Notacao cientifica\
            \nQual deseja testar? - '))

        elif a==4:

            b = 0

            l = 0

            v=[10, 20, 40, 80, 160, 320, 640]

            b = int(input("\n1 - Teste do metodo 11\
                    \n2 - Euler Implicito\
                    \n3 - Crank Nicolson\
                    \nQual teste deseja rodar? - "))

            c = int(input("\nSelecione a função f(t,x) desejada:\
                    \n1: f(t,x) = 10x²(x-1)-60xt+ 20t\
          \n2: f(t,x) = 10cos(10t)x²(1-x)²-(1 +sin(10t))(12x²-12x+ 2)\
```

```
\n3:f(t,x)=25t²e^(t-x)cos(5tx)-10te^(t-x)sen(5tx) - 5xe^(t-x) sen(5tx)\

            \n4: f(t,x) = 10000(1-2t²)N (p = 0.25 e gh cte):  "))

            for i in v:

                x = ep1.fazx(i)

    l=float(input("\nLambda (0.25 ou 0.5 ou 0.51) = "))

                integrais(x, i, l, b, c)
```

No programa de testes, inicialmente, é perguntado qual dos testes presentes no programa deseja ser feito, como visto em códigos supracitados, focando no teste de funções, há um vetor com os valores de N a serem usados, depois, pergunta-se qual método deverá ser testado, então qual função através deste método.

Faz-se x, vetor que representa a barra, sob N divisões equidistantes, utilizando-se o método de ep1, e recolhe-e o lambda a ser utilizado, então é chamada a função integrais, que calcula pelos métodos numéricos o que se pede, codigo de def integrais abaixo:

```
    def ef integrais(x, n, lmbda, b, a):

        tete=int(input("1- para delta t = delta x:

                                            \n2- para m=n²/lambda: "))

        if teste ==1:

            t=ep1.fazt(n)

        elif teste ==2:

            m = ((n**2) / lmbda) - 1    # Tirado de (27)

            t=ep1.fazt(m)

        else:

            print("opcao invalida")

        if b == 1:

            ep1.metodo11 (x,t,a)
```

```
        elif b == 2:

            ep1.euler (x,t,a)

        elif b == 3:

            ep1.crank (x,t,a)
```

Na função integrais, foi recebido o vetor x, o número n, o lambda do teste, os indicadores b e a. Primeiramente pese-de pasa saber que tipo de delta t será utilizado, caso seja igual, faz-se t com o valor de n, caso contrário calcula-se M com os dados N e lambda, após isso é feito o vetor t, correspondente ao tempo total sob M subdivisões de mesmo intervalo. Por último é chamado o método correspondente ao indicador b, passando os vetores x, t, e o indicador a, que representa a função a ser calculada numericamente pelo método em questão, das formas aludidas.

3. Funções, resultados e gráficos

3.1 Função (a) - $f(t,x) = 10\cos(10t)x^2 - (1+\text{sen}(10t))\,(12x^2-12x+2)$

A fonte de calor dada tem condição inicial e de fronteiras nulas. A solução real analítica também é dada. Nesta configuração, temos.

$$f(t, x) = 10\cos(10t)x^2\,(1-x)^2 - (1+\text{sen}(10t))(12x^2-12x+2) \quad (11)$$

$$u_0(x) = 0 \quad (12)$$

$$g_1(t) = 0 \quad (13)$$

$$g_2(t) = 0 \quad (14)$$

$$u(t,x) = (1+\text{sen}(10t))\,x^2\,(1-x)^2 \quad (15)$$

3.1.1. Exercício 1a - Método 11 para função (a)

Temos um número de passos da ordem de N^3/λ, ou seja N vezes M passos. Pois é criada uma matriz M por N.

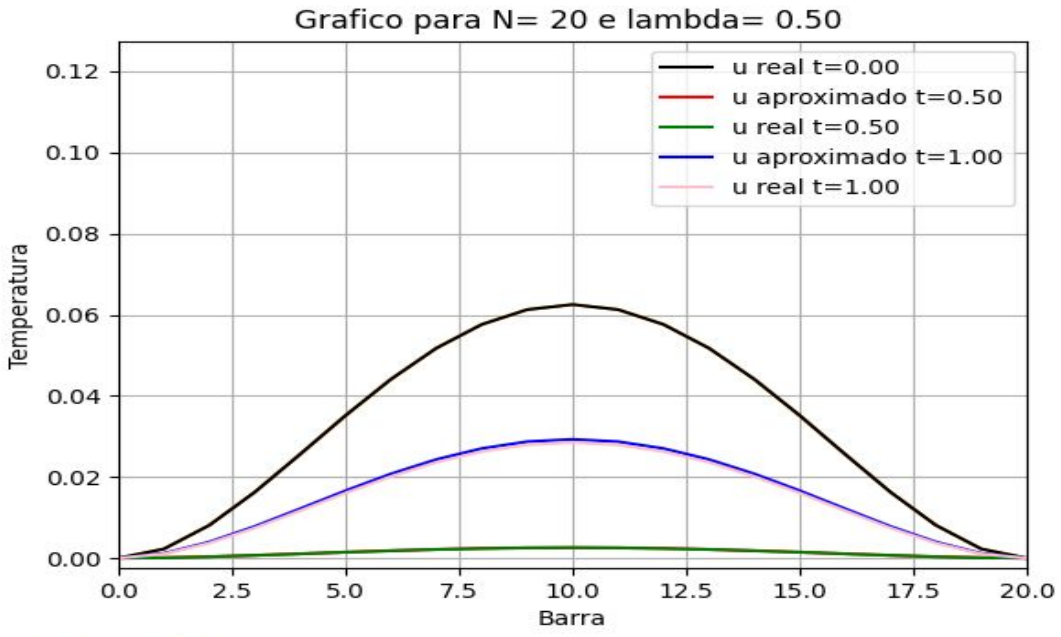

figura 01 - Gráfico da função (a) para N = 20 e λ = 0.5 com método 11

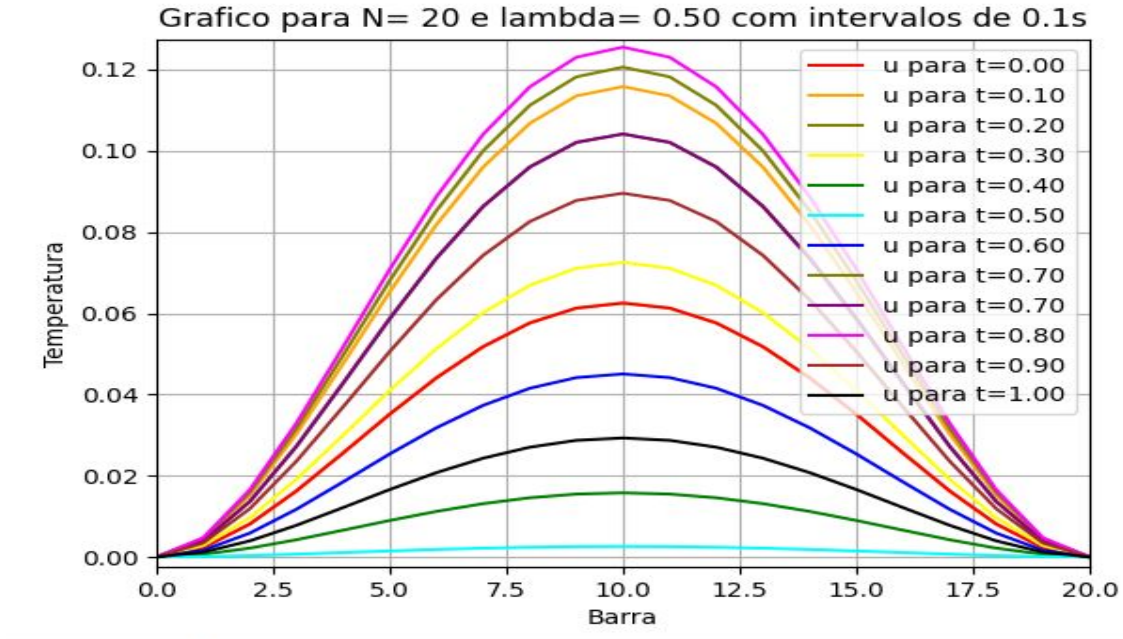

figura 02 - Gráfico temporal da função (a) para N = 20 e λ = 0.5 com método 11

Maior erro de truncamento = 1.13e-14 ,em t= 1.59e-01 e x= 5.00e-01

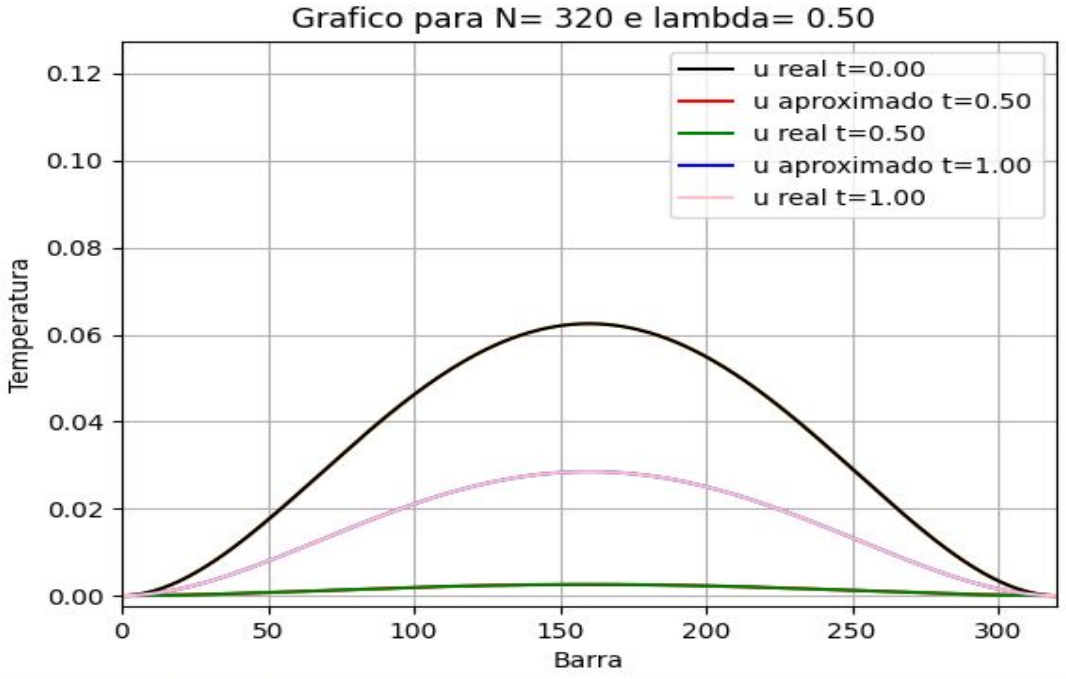

figura 03 - Gráfico da função (a) para N = 320 e λ = 0.5 com método 11

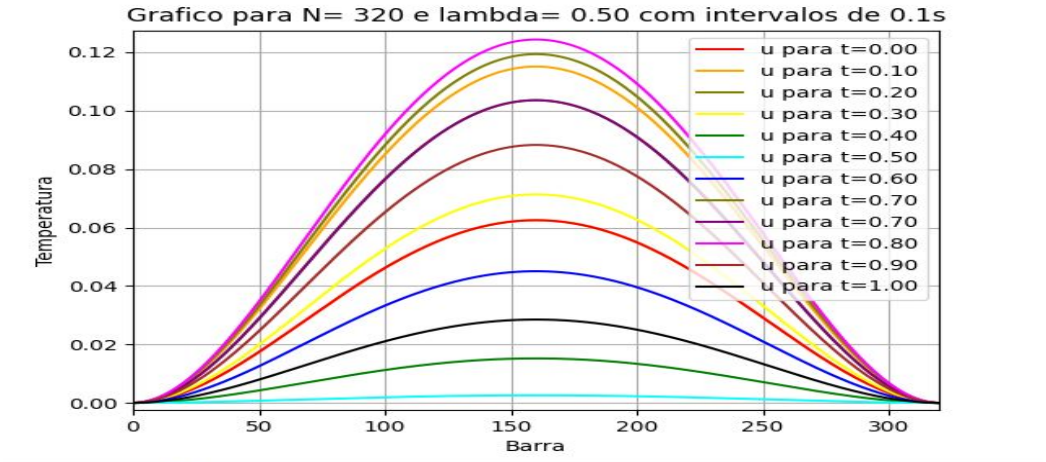

figura 04 - Gráfico temporal da função (a) para N = 320 e λ = 0.5 com método 11

Maior erro de truncamento = 1.71e-13 ,em t= 7.86e-01 e x= 5.00e-01

Como podemos notar, o erro de truncamento é extremamente baixo (da ordem de 10^{-13}), porém há um leve aumento de 10 vezes quando comparado ao calculado com N = 20 e N = 320. Isso se deve a erros de arredondamento nos cálculos, que por conta do N ser maior, o programa calcula muitos mais pontos. Os erros absolutos reduzem em 100 vezes para cada aumento de 10 vezes de N, pois este é de ordem 2 para o erro.

Para testarmos a condição de que convergência, calculamos utilizando λ = 0.51 (que pode ser visto no gráfico a seguir). É notório o erro em questão. Dessa forma, devemos utilizar o método 11 com a condição de λ <= 0.5.

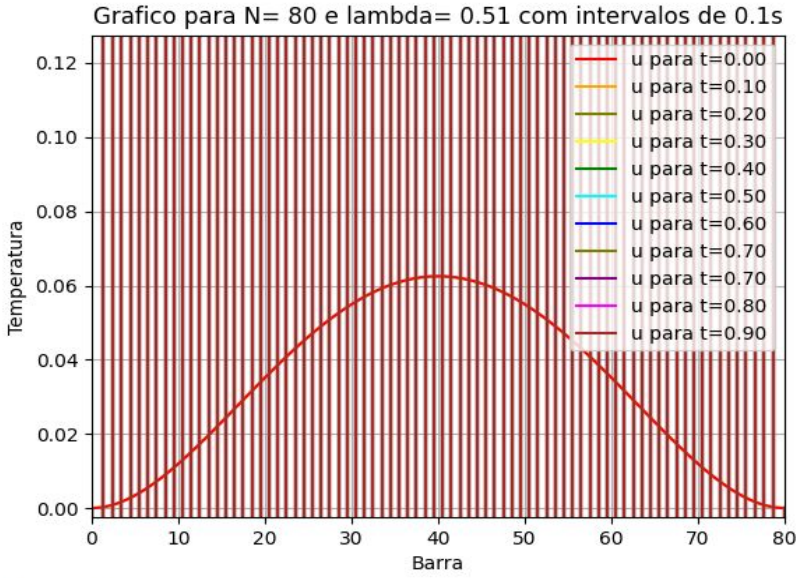

figura 05 - Gráfico temporal da função (a) para N = 80 e λ = 0.51 com método 11

3.1.2. Método Euler Implícito para função (a)

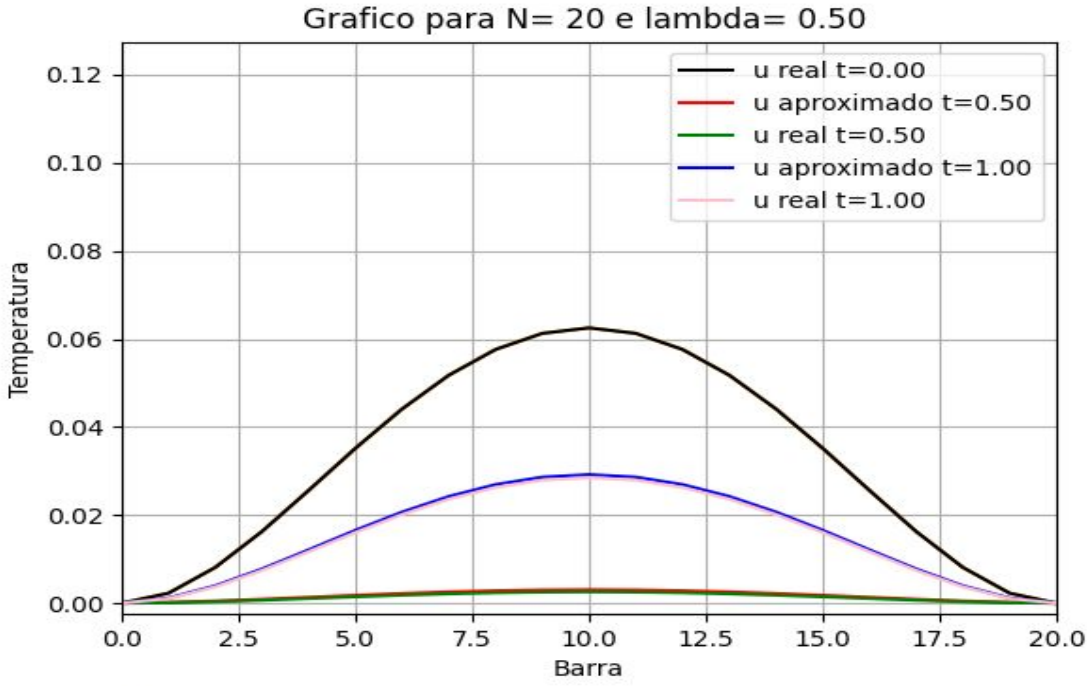

figura 06 - Gráfico da função (a) para N = 20 e λ = 0.5 com Euler Implícito

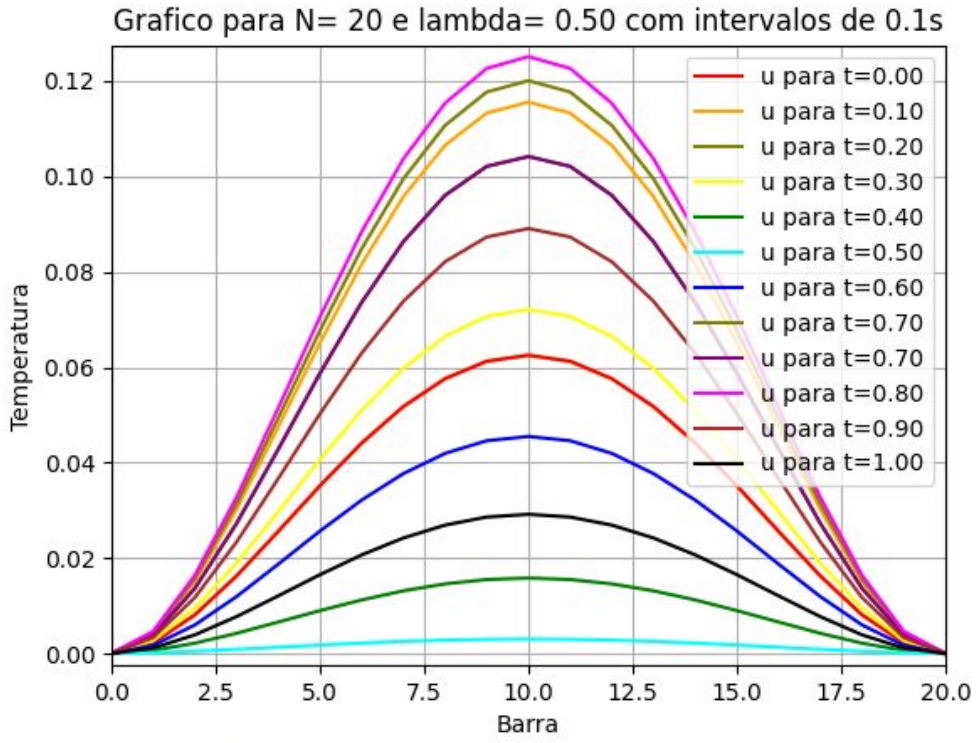

figura 07 - Gráfico temporal da função (a) para N = 20 e λ = 0.50 com Euler Implícito

Assim como para o método 11, o gráfico para λ = 0.51 foi traçado com o método Euler Implícito. Vemos que o gráfico não possui erros notórios, ou seja, λ <= 0.5 não é um fator limitante para este método.

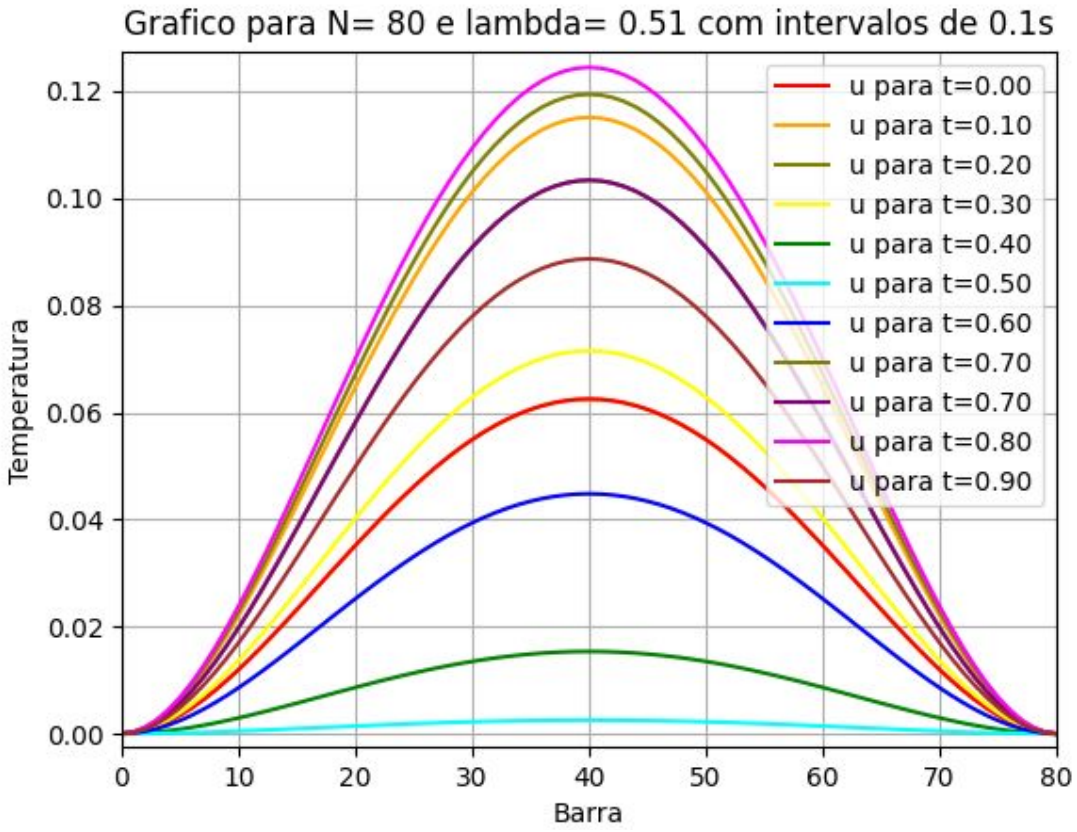

figura 08 - Gráfico temporal da função (a) para N = 80 e λ = 0.51 com Euler Implícito

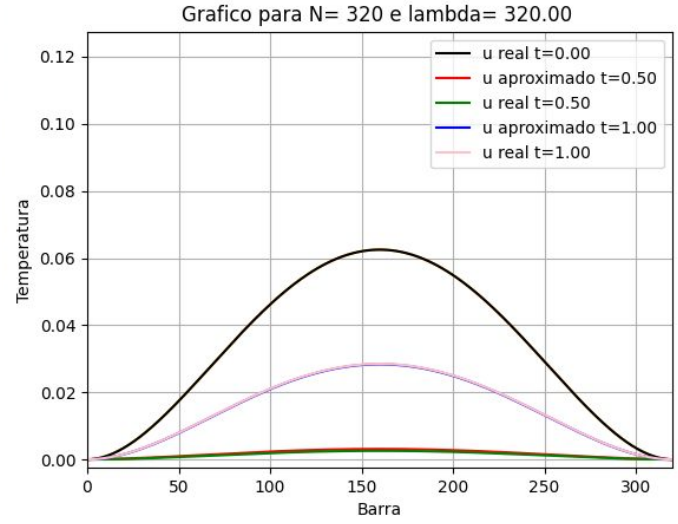

figura 09 - Gráfico temporal da função (a) para N = 320 e λ = 320 com Euler Implícito e delta t = delta x

Maior erro de truncamento = 1.01e-02 ,em t= 9.97e-01 e x= 5.00e-01

Maior erro absoluto = 6.65e-04

Maior erro absoluto em (t=1) = 1.25e-04

O que nos mostra um valor de erro baixo, ou seja, é eficiente utilizar o método Euler implícito para Δx = Δt.

3.1.3. Método Crank-Nicolson para função (a)

Os gráficos para o método Crank-Nicolson são apresentados a seguir. É possível notar a não limitância com λ <= 0.5 para o método Crank-Nicolson (assim como mostrado em Euler Implícito).

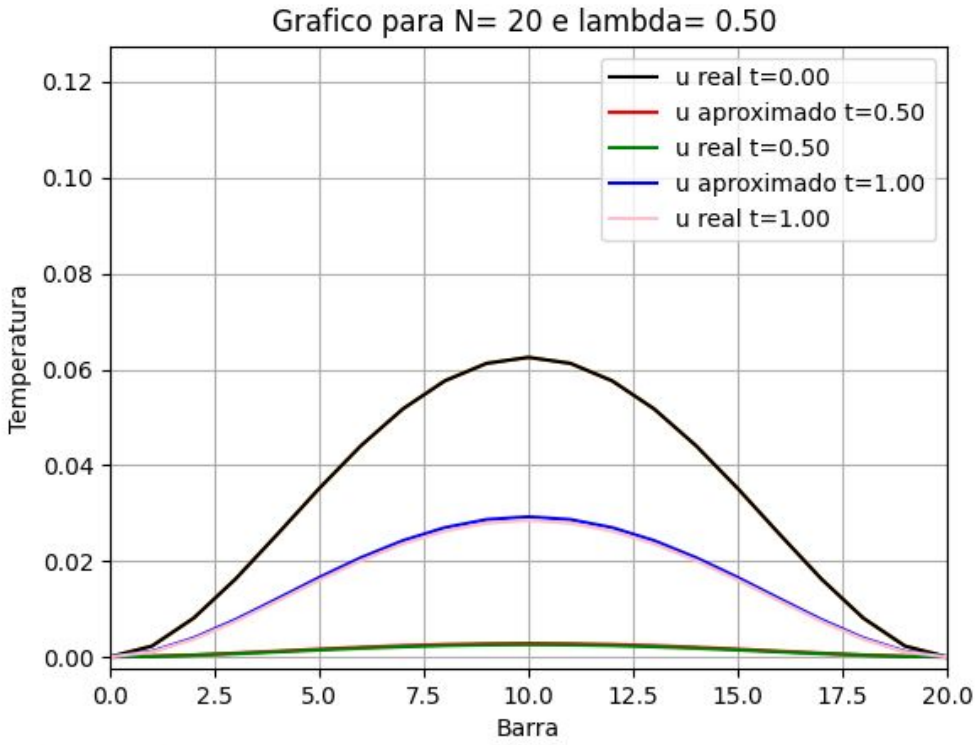

figura 10 - Gráfico da função (a) para N = 20 e λ = 0.5 com Crank-Nicolson

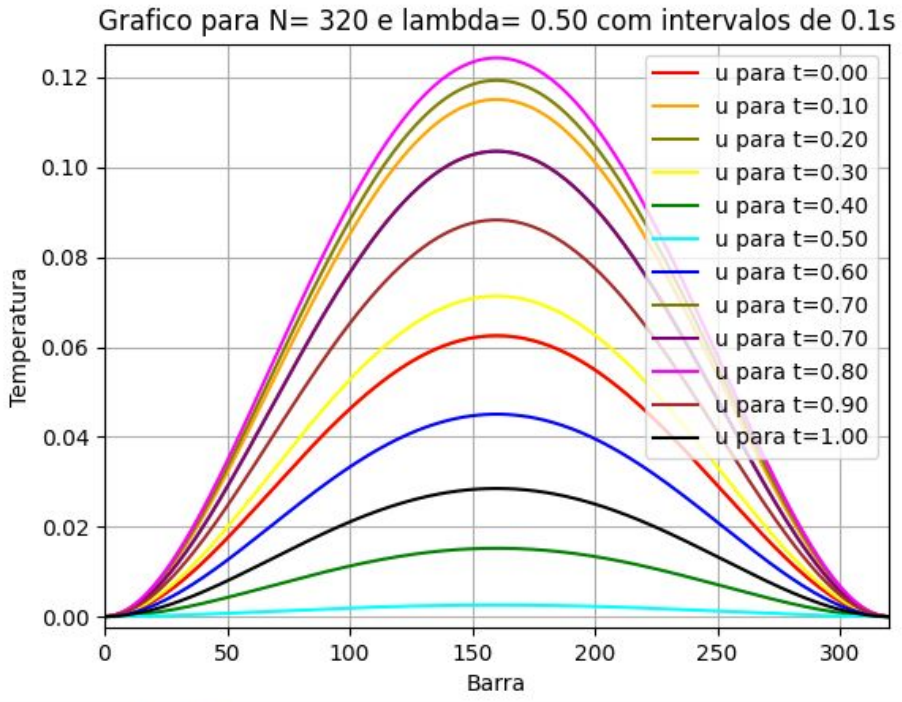

figura 10 - Gráfico temporal da função (a) para N = 320 e λ = 0.5 com Crank-Nicolson

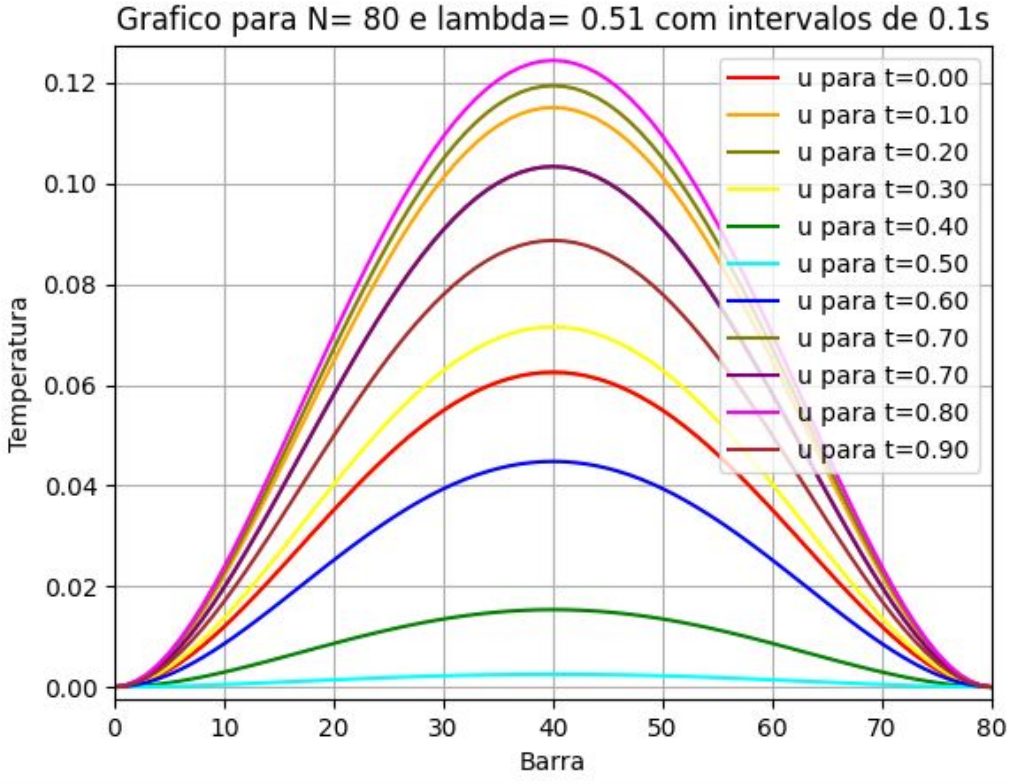

figura 11 - Gráfico temporal da função (a) para N = 80 e λ = 0.51com Crank-Nicolson

3.2. Função (b) - solução real $u(t, x) = e^{t-x}\cos(5tx)$

Para este problema, nos foi dado inicialmente a função real:

$$u(t, x) = e^{t-x}\cos(5tx) \quad (16)$$

A partir dela, podemos encontrar a condição inicial igualando t a 0 na solução, ou seja, temos Da mesma forma, as condições de fronteiras podem ser encontradas, igualando x a 0 e a 1. Para a fonte de calor f(t,x), devemos utilizar a equação diferencial da fonte de calor:

$$f(t,x)=\frac{\partial u(t,x)}{\partial t}-\frac{\partial^2 u(t,x)}{\partial x^2} \quad (17)$$

Assim, obtemos:

$$u_0(x) = e^{-x} \quad (18)$$

$$g_1(t) = e^{t} \quad (19)$$

$$g_2(t) = e^{t-1}\cos(5t) \quad (20)$$

$$f(t,x) = 25t^2 e^{t-x}\cos(5tx) - 10t\,\mathrm{sen}(5tx) - 5x\, e^{t-x}\,\mathrm{sen}(5tx) \quad (21)$$

3.2.1. Exercício 1b - Método 11 para função (b)

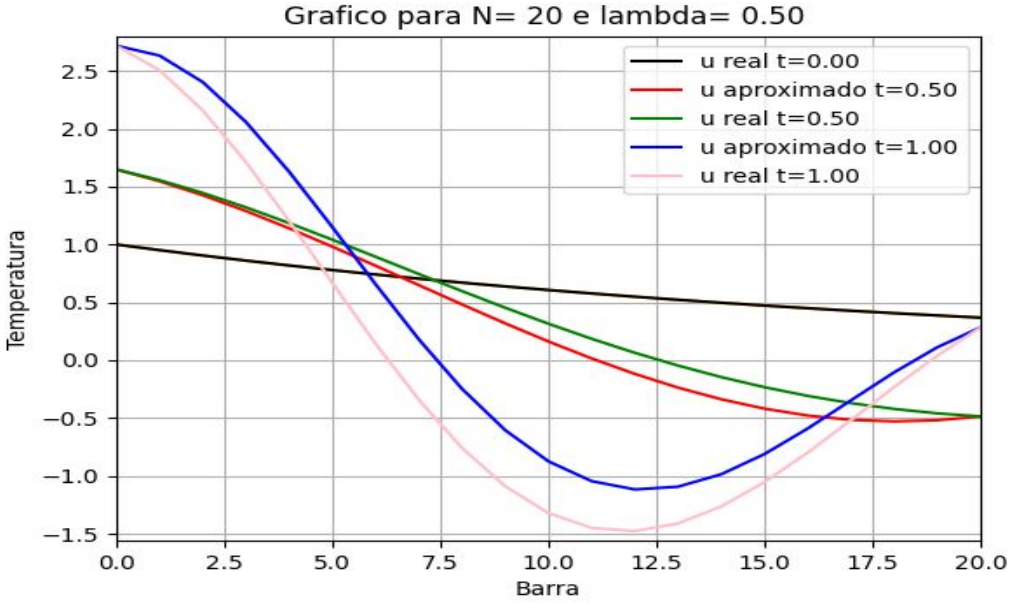

figura 12 - Gráfico da função (b) para N = 20 e λ = 0.5 com método 11

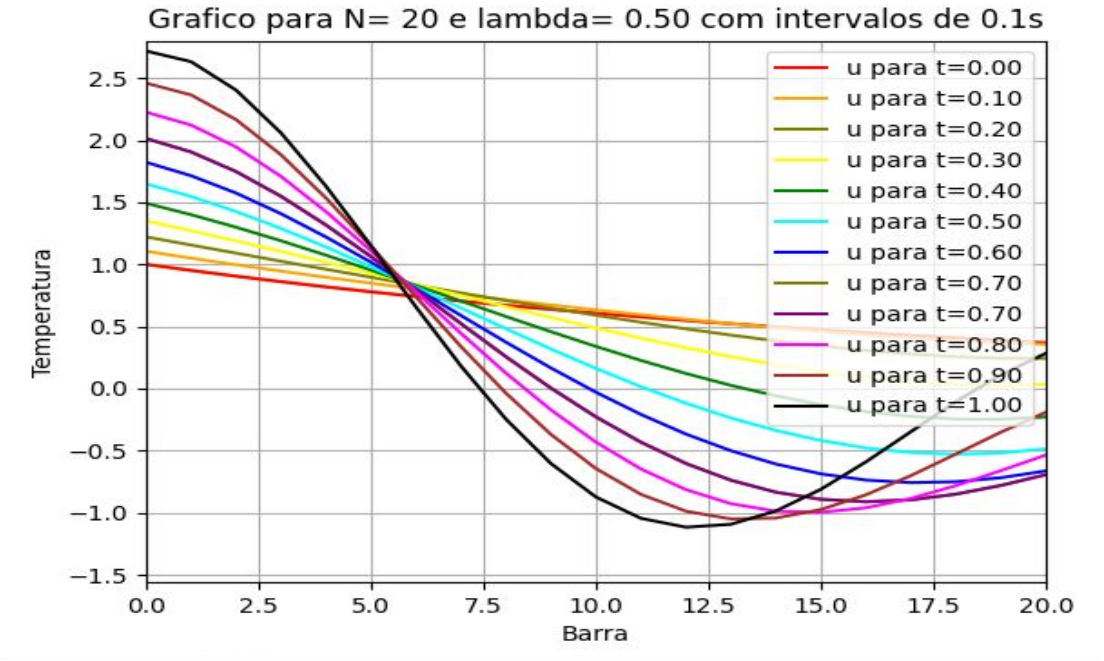

figura 13 - Gráfico temporal da função (b) para N = 20 e λ = 0.5 com método 11

Maior erro de truncamento = 1.85e-13 ,em t= 8.92e-01 e x= 5.00e-02

Maior erro absoluto = 5.22e-01

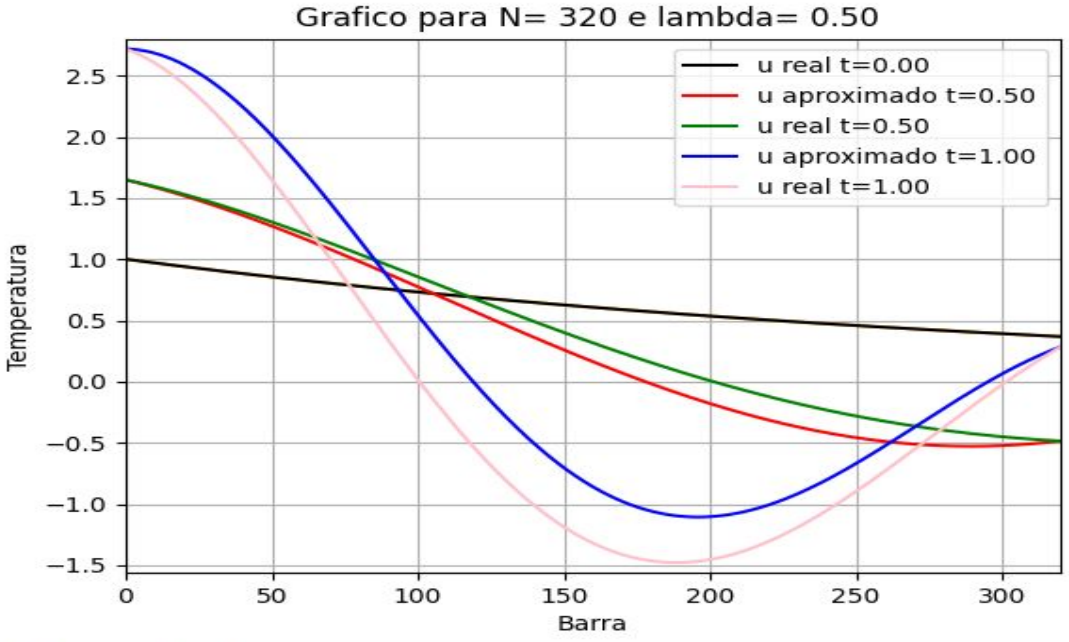

figura 14 - Gráfico da função (b) para N = 320 e λ = 0.5 com método 11

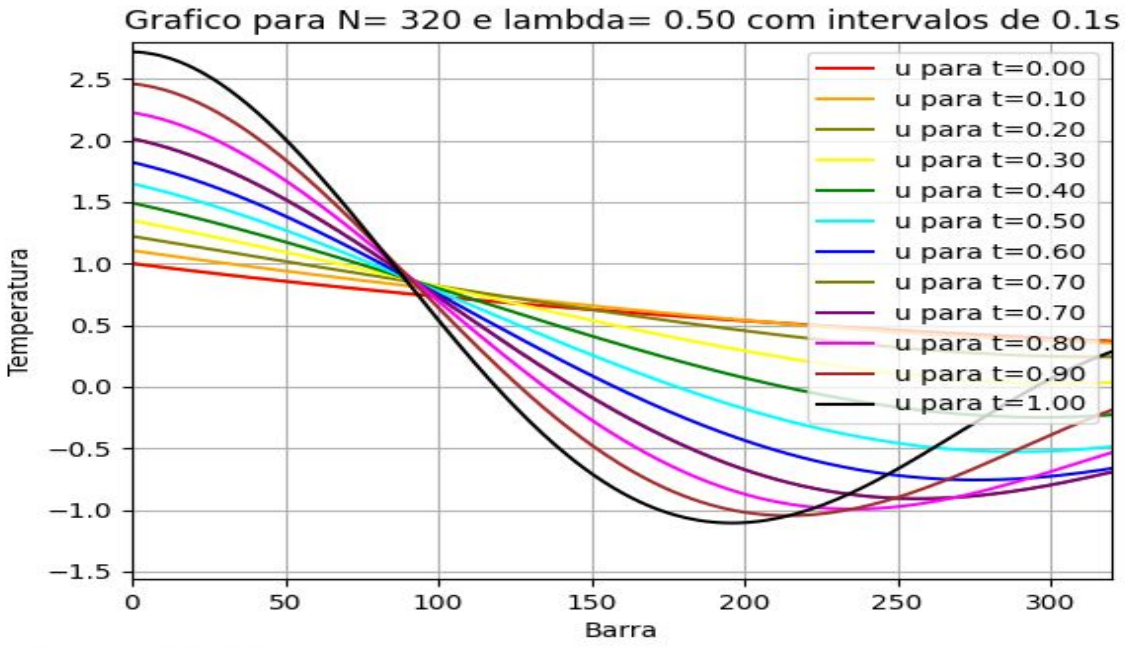

figura 15 - Gráfico temporal da função (b) para N = 320 e λ = 0.5 com método 11

Maior erro de truncamento = 4.55e-11 ,em t= 9.65e-01 e x= 6.25e-03

Maior erro absoluto = 5.33e-01

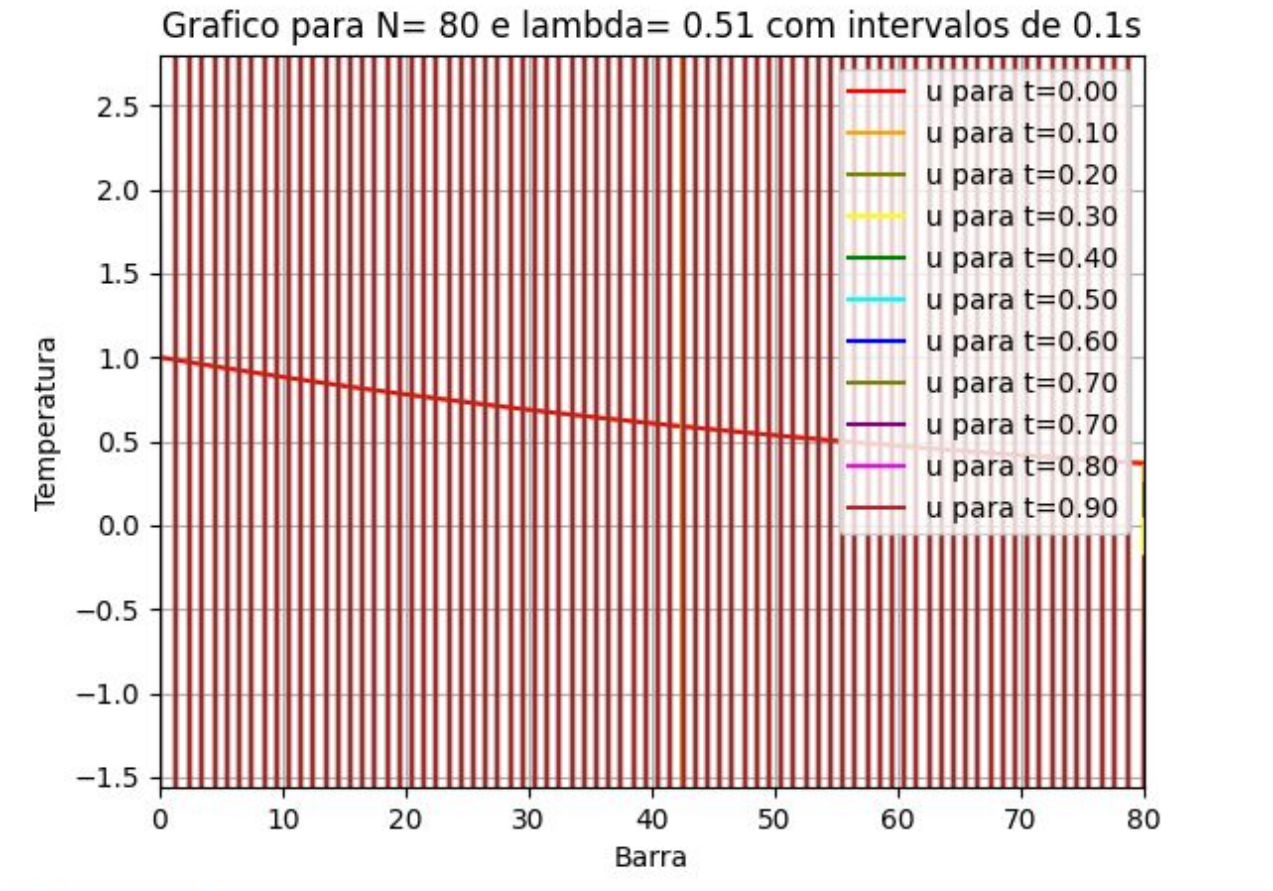

figura 16 - Gráfico temporal da função (b) para N = 80 e λ = 0.51 com método 11

3.2.2. Método Euler Implícito para função (b)

Podemos notar também o baixo erro para o método Euler Implícito. Serão apresentados gráficos com diferentes valores de N e λ. Lembrando que para outros valores de N e λ, podemos utilizar o código, mas para não enchermos o relatório com gráficos, optamos por mostrar alguns para vizualização.

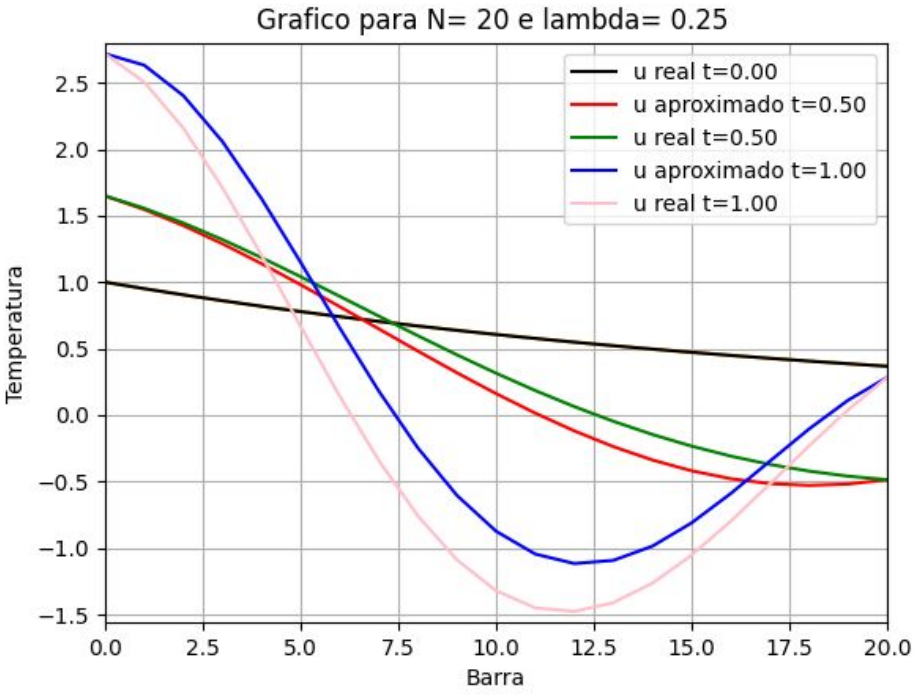

figura 17 - Gráfico da função (b) para N = 20 e λ = 0.25 com Euler Implícito

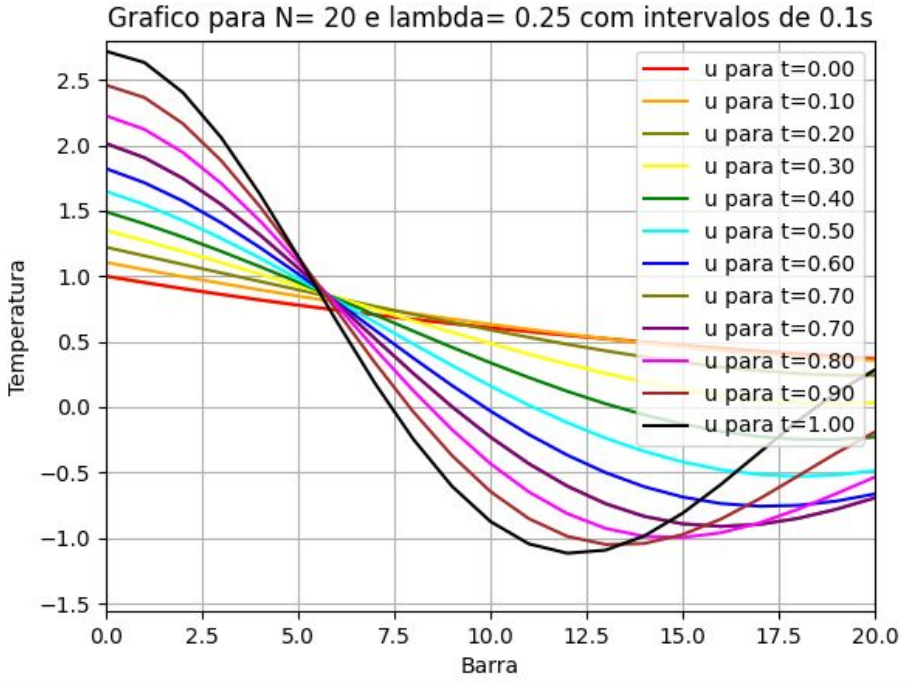

figura 18 - Gráfico temporal da função (b) para N = 20 e λ = 0.25 com Euler implícito

Maior erro de truncamento = 1.22e-02 ,em t= 7.72e-01 e x= 9.50e-01

Maior erro absoluto = 5.22e-01

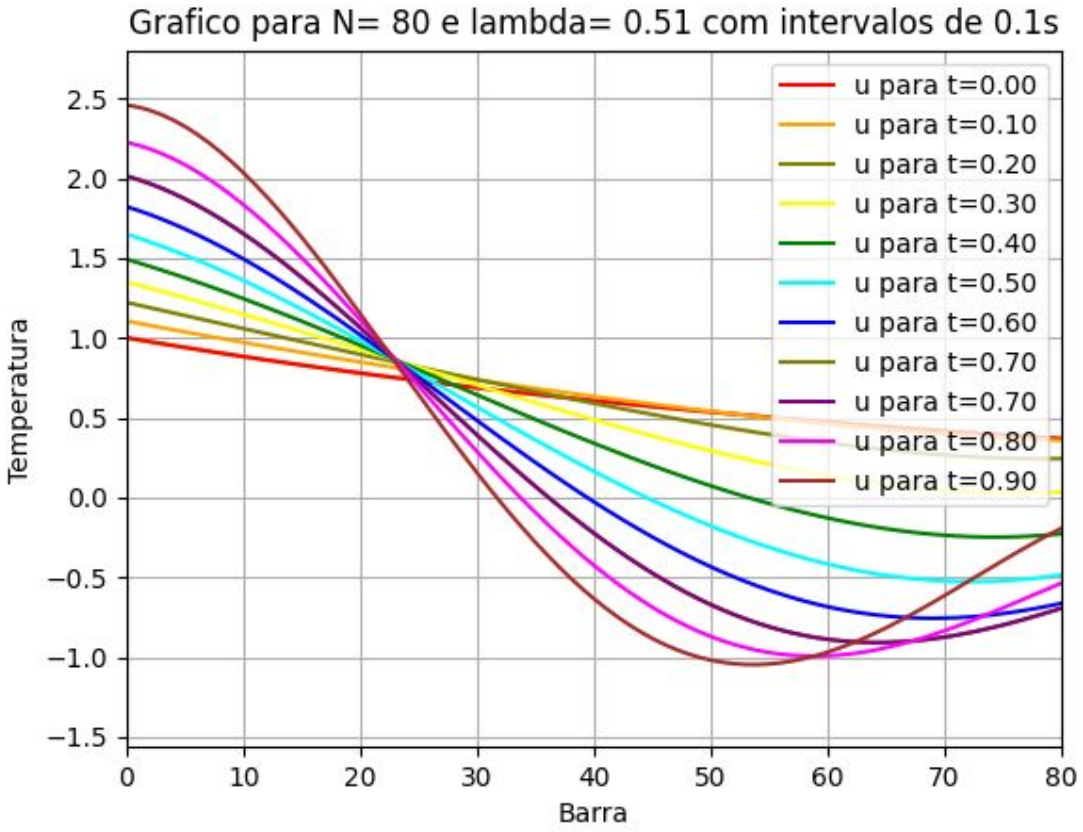

figura 19 - Gráfico temporal da função (b) para N = 80 e λ = 0.51 com Euler implícito

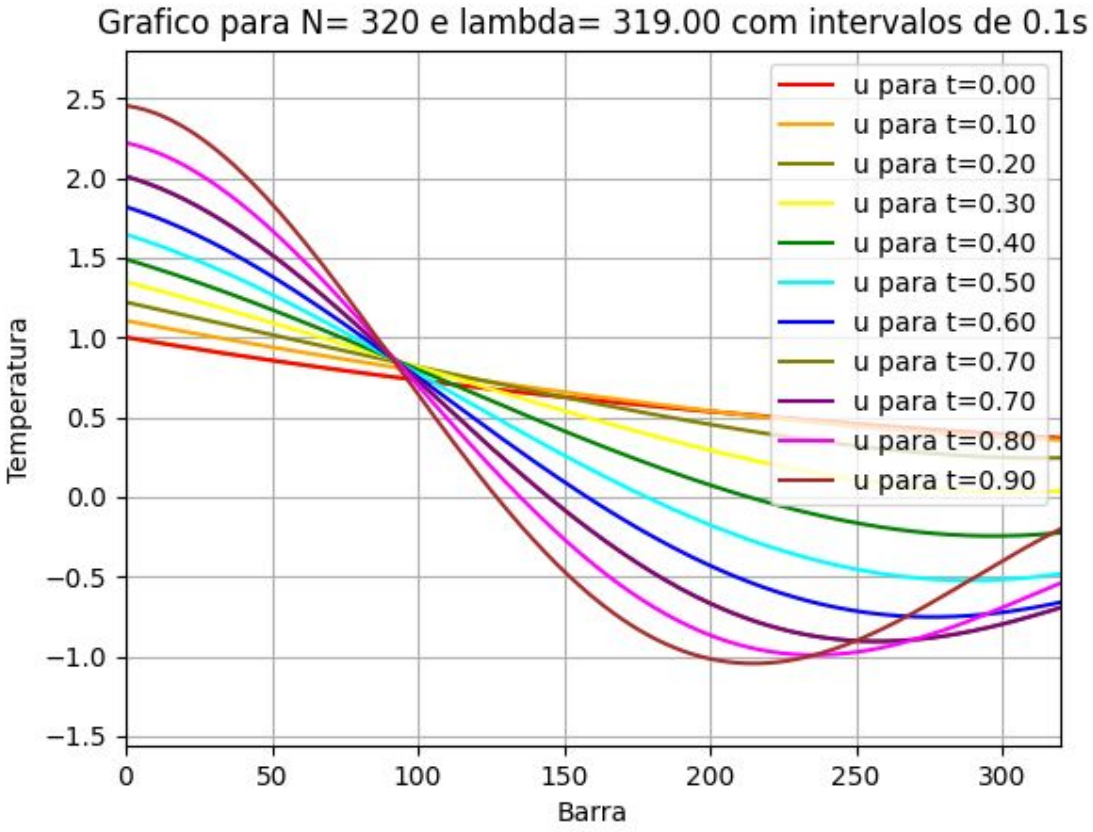

figura - Gráfico temporal da função (b) para N = 320 e λ = 319 com Euler Implícito com delta t = delta x

Maior erro de truncamento = 6.17e-02 ,em t= 7.48e-01 e x= 9.97e-01

Maior erro absoluto = 5.33e-01

Maior erro absoluto em (t=1) = 0.00e+00

3.2.3. Método Crank-Nicolson para função (b)

Lembrando que todos os gráficos estão traçados em função de i (vai de 0 a N), porém cada valor de i corresponde a um x = i/N.

Utilizando N = M (Δx = Δt), como visto nas últimas figuras, o valor não converge tão adequadamente ao real, desta forma, é preferível utilizar um valor bem maior para M.

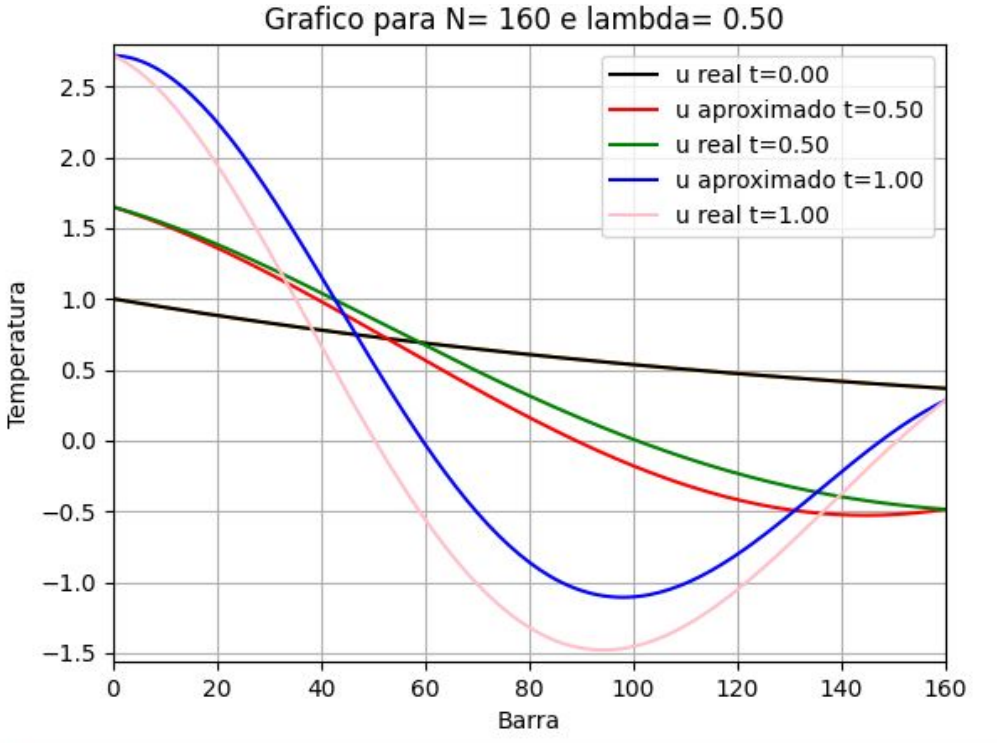

figura 20 - Gráfico da função (b) para N = 160 e λ = 0.50 com Crank-Nicolson

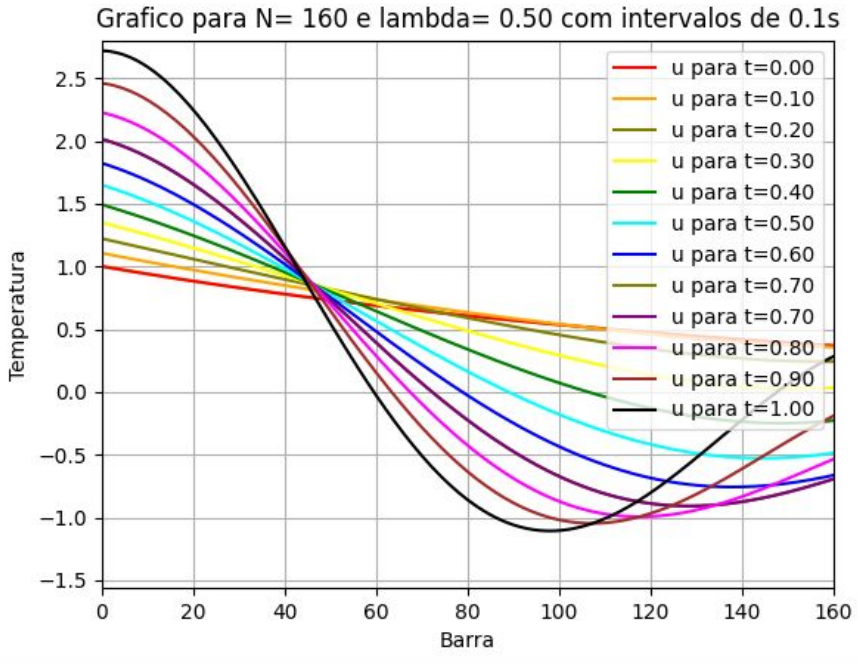

figura 21 - Gráfico temporal da função (b) para N = 160 e λ = 0.5 com Crank-Nicolson

Maior erro de truncamento = 1.93e-04 ,em t= 7.50e-01 e x= 9.94e-01

Maior erro absoluto = 5.32e-01

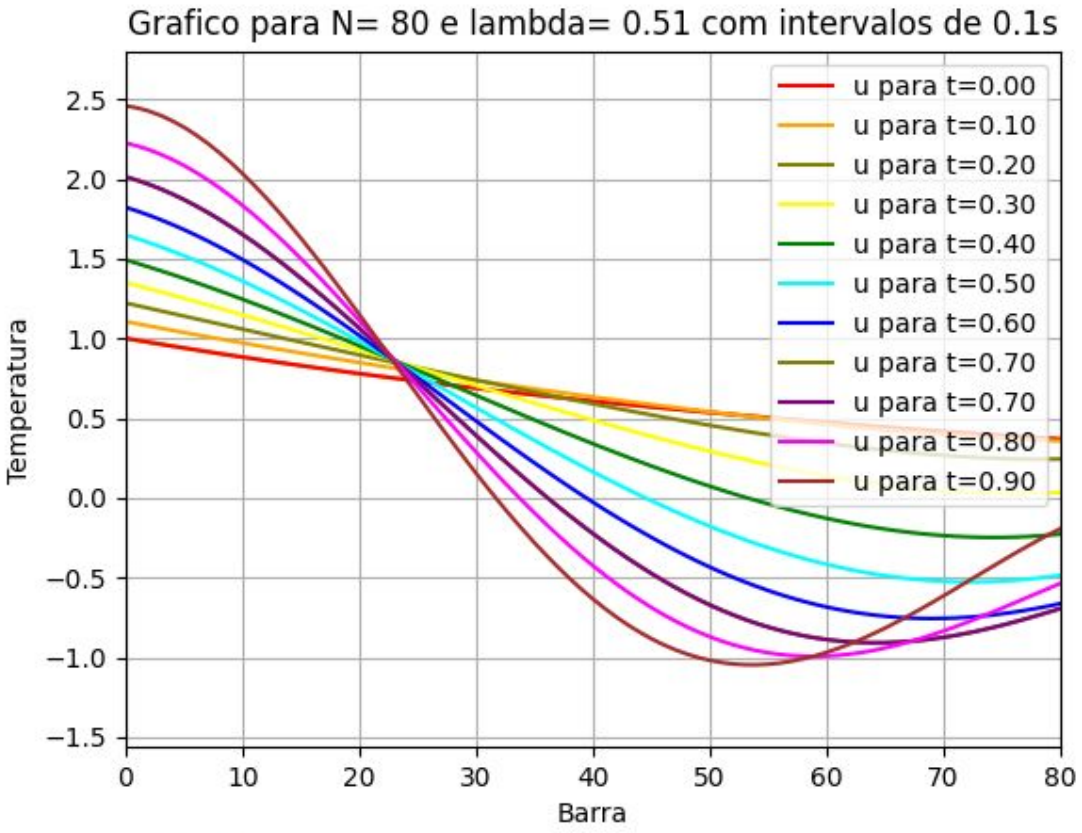

figura 22 - Gráfico temporal da função (b) para N = 80 e λ=0.51 com Crank-Nicolson

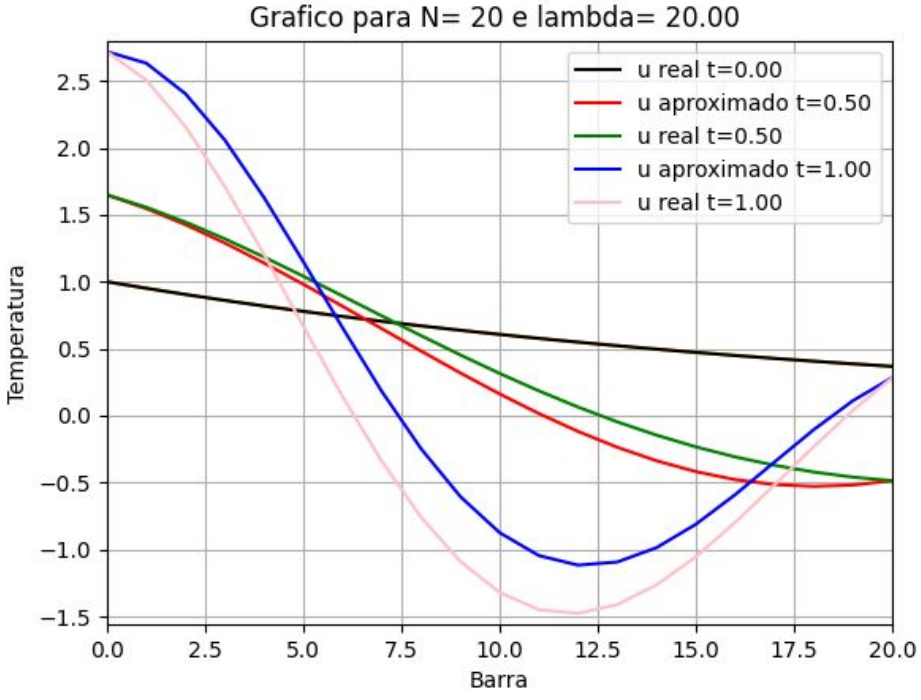

figura 23 - Gráfico temporal da função (b) para N = 20 e λ=20 com Crank-Nicolson

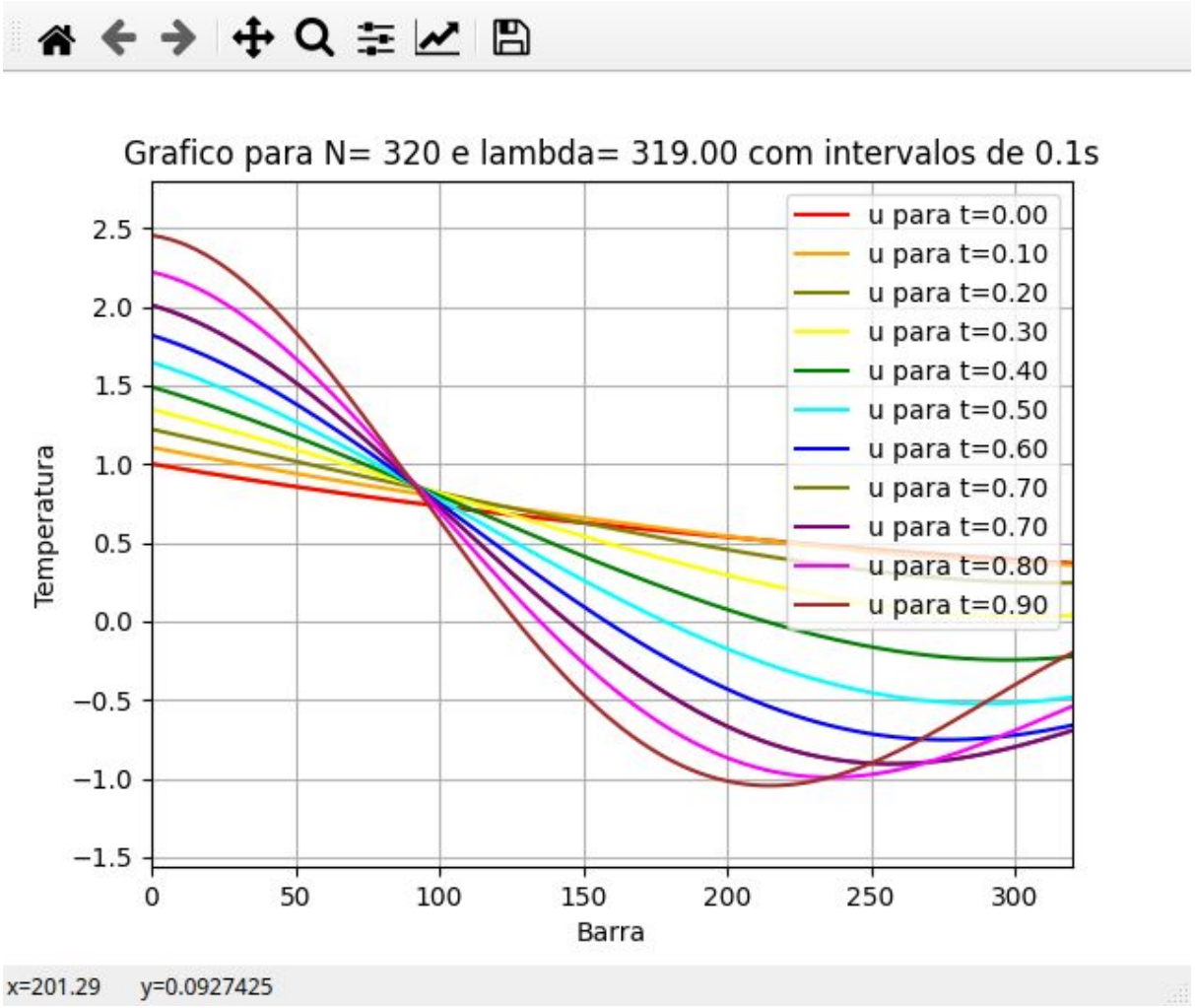

figura - Gráfico temporal da função (b) para N = 320 e λ = 319 com Crank-Nicolson com delta t = delta x

Maior erro de truncamento = 3.08e-02 ,em t= 7.45e-01 e x= 9.97e-01

Maior erro absoluto = 5.33e-01

Maior erro absoluto em (t=1) = 0.00e+00

3.3. Função (c) - f(t,x)=10000(1−2t²)g(x) (para p−h/2 ≤ x ≤ p+h/2)

Neste caso, temos uma fonte pontual, encontrada no ponto p (x= 0.25). Com condição inicial e de fronteiras nulas. A fonte pontual é definida como:

$$f(t,x) = 10000\,(1-2t^2)\,g_h(x) \tag{22}$$

Sendo a função $g_h(x)$ definida como:

$g_h(x) = 1/h$ para p−h/2 ≤ x ≤ p+h/2

$g_h(x) = 0$ para outros valores de x (23)

Dado que h = Δx = 1/N, temos:

$f(t, x) = 10000(1-t^2)$ N (24)

$u_0(x) = 0$ (25)

$g_1(t) = 0$ (26)

$g_2(t) = 0$ (27)

3.2.1. Exercício 1c - Método 11 para função (c)

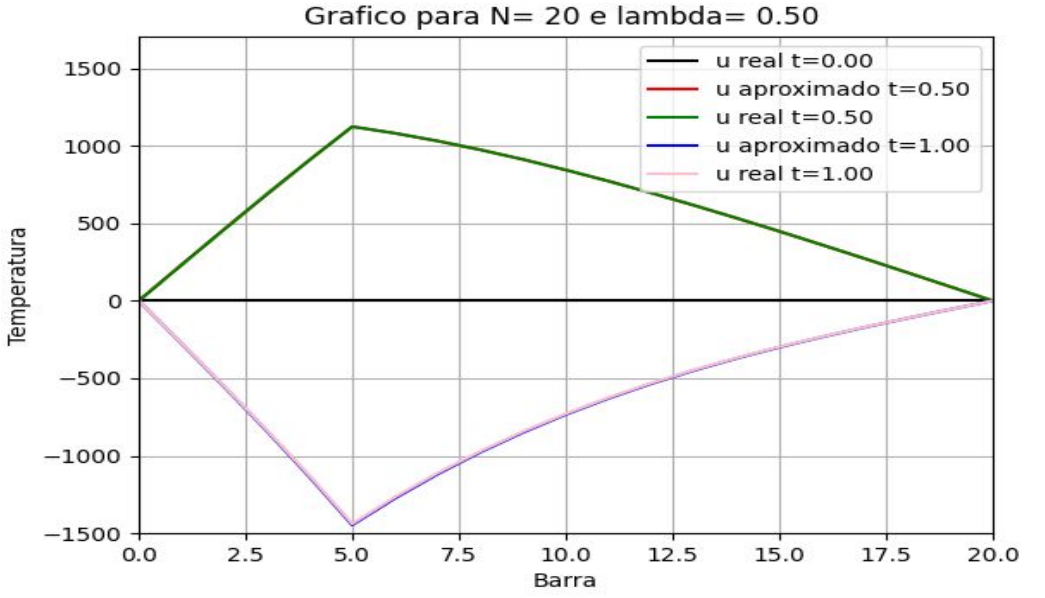

figura 24 - Gráfico da função (c) para N = 20 e λ = 0.50 com método 11

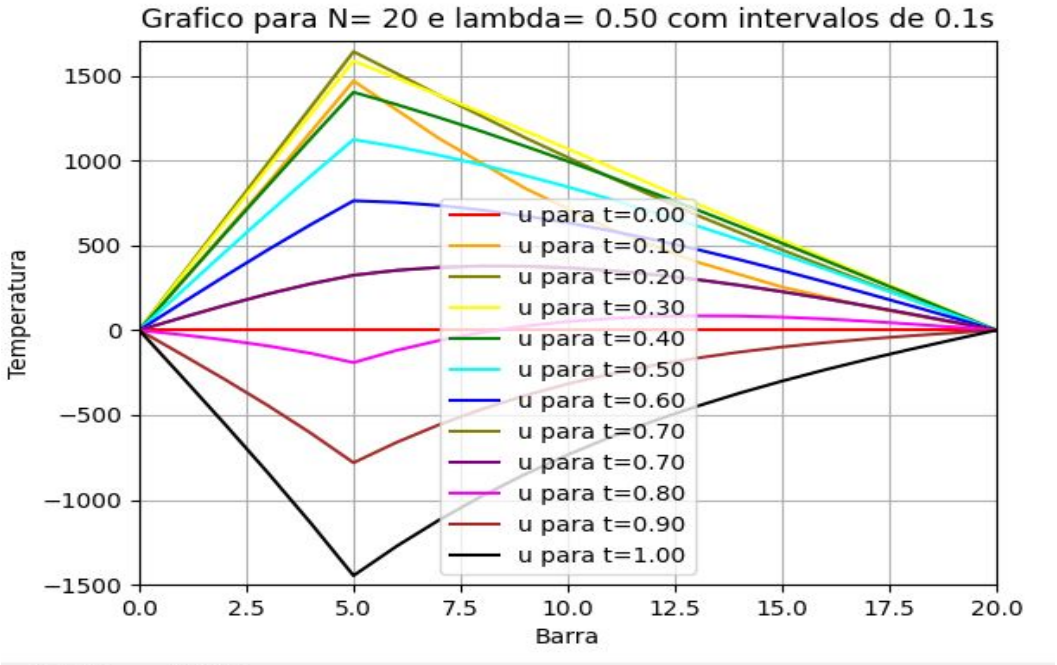

figura 24 - Gráfico temporal da função (c) para N = 20 e λ=0.5 com método 11

Maior erro de truncamento = 1.46e-10 ,em t= 4.37e-02 e x= 2.50e-01

Maior erro absoluto = 1.25e+02

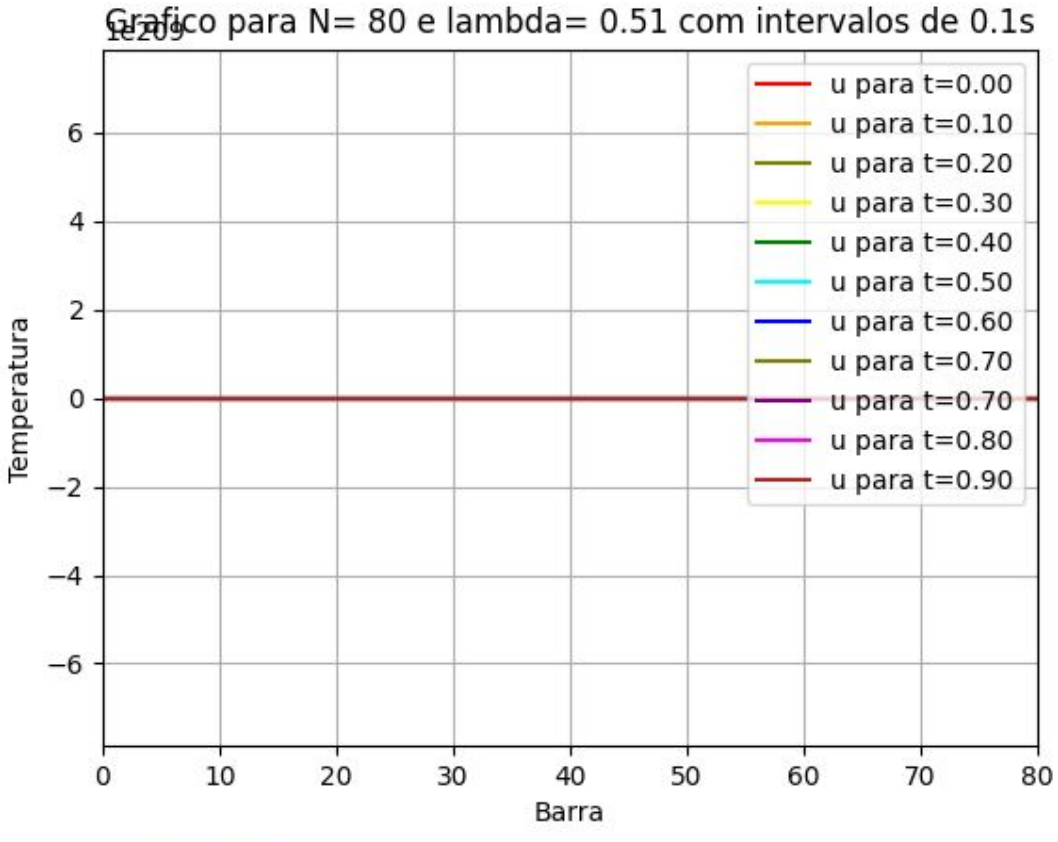

figura 25 - Gráfico temporal da função (c) para N = 80 e λ=0.51 com método 11

3.3.2. Método Euler Implícito para função (c)

Apesar de os gráficos temporais estarem sendo representados com as 11 (0 a 1s de 0.1s em 0.1s) curvas, o programa separa as curvas em dois períodos (de 0 a 0.5 e de 0.6 a 1). Optamos por traçar todas as curvas para melhor visualização no relatório e não enchermos de gráficos redundantes.

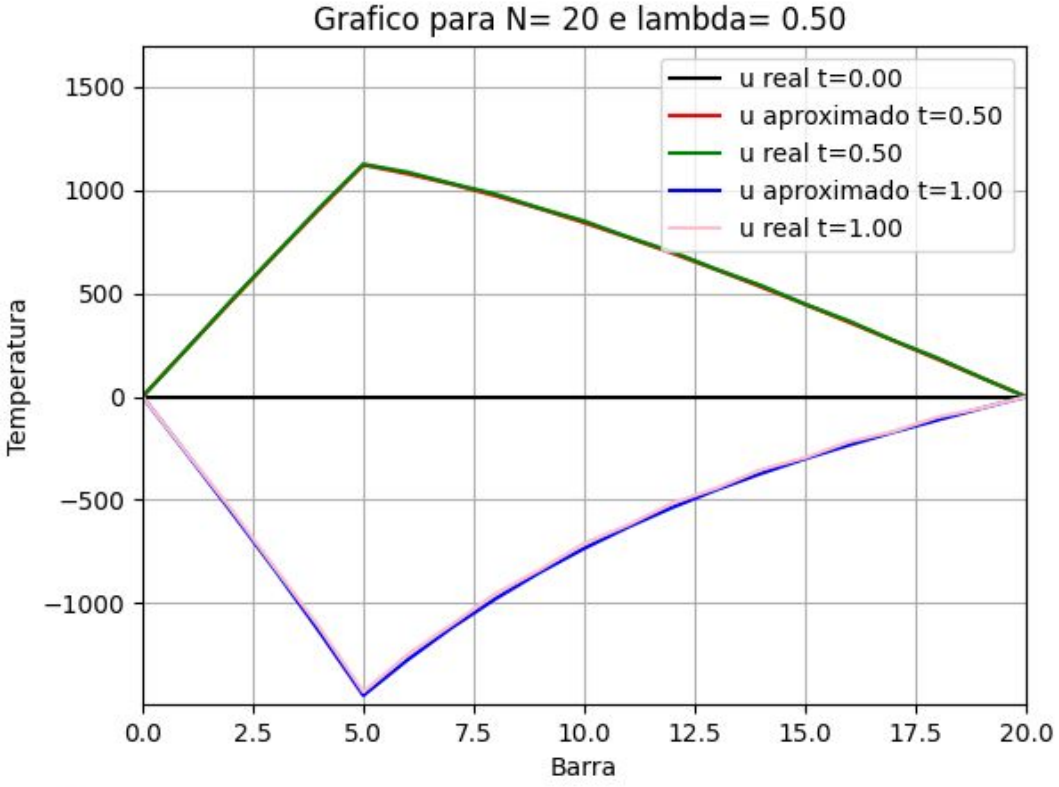

figura 26 - Gráfico da função (c) para N = 20 e λ = 0.50 com Euler implícito

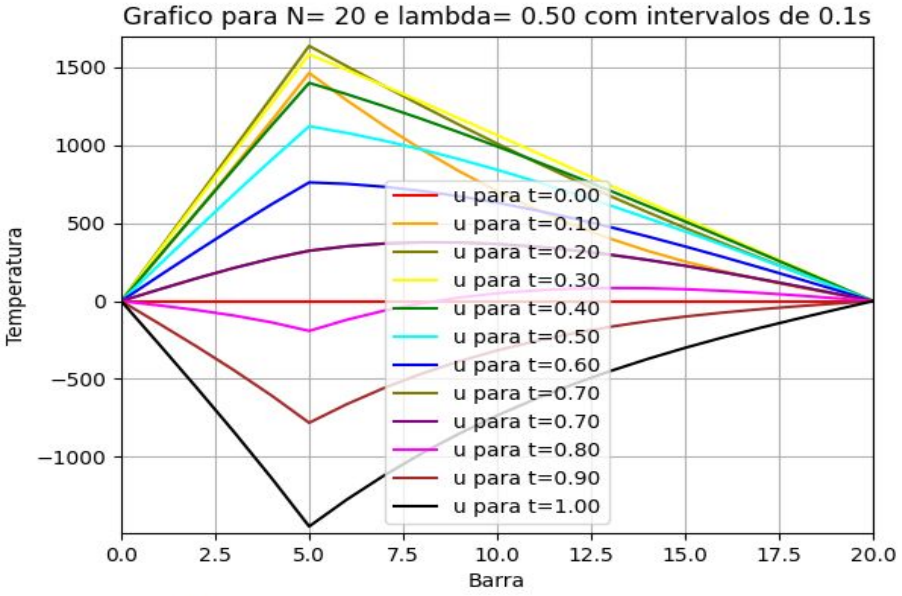

figura 27 - Gráfico temporal da função (c) para N = 20 e λ=0.5 com Euler implícito

Maior erro de truncamento = 1.36e-10 ,em t= 9.99e-01 e x= 3.50e-01

Maior erro absoluto = 1.84e+01

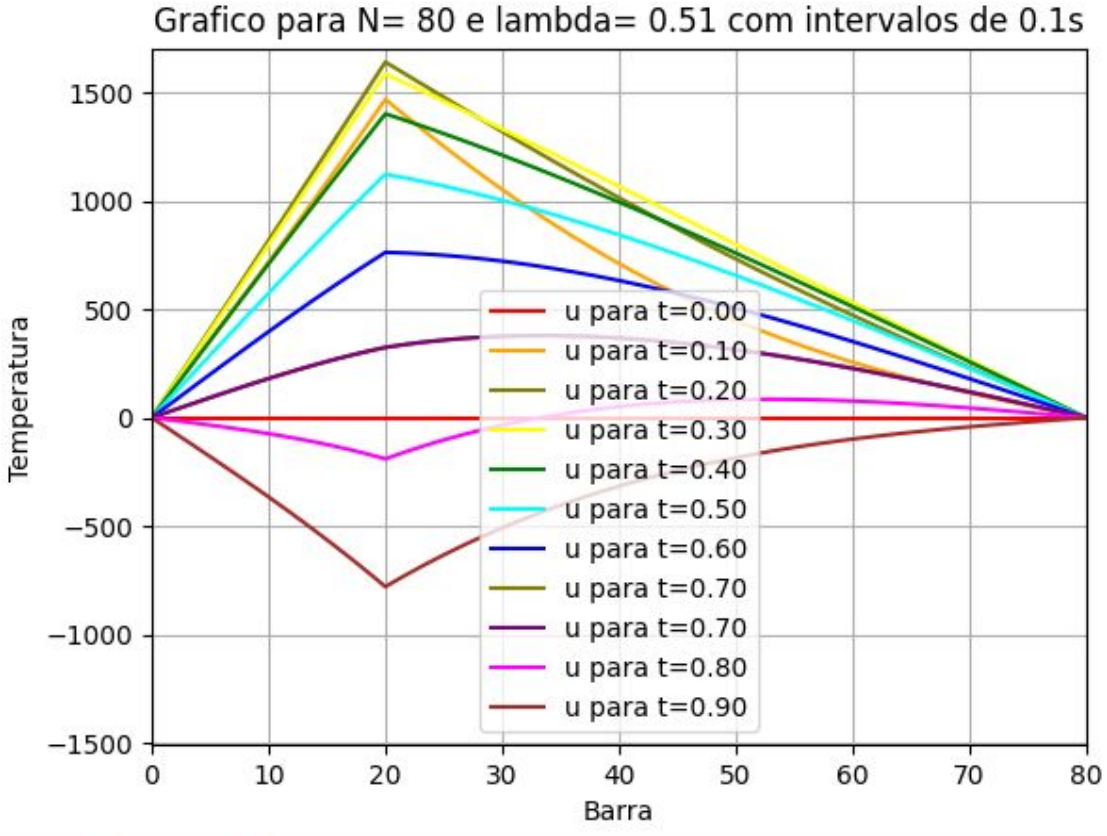

figura 28 - Gráfico temporal da função (c) para N = 80 e λ=0.51 com Euler implícito

3.3.3. Método Crank-Nicolson para função (c)

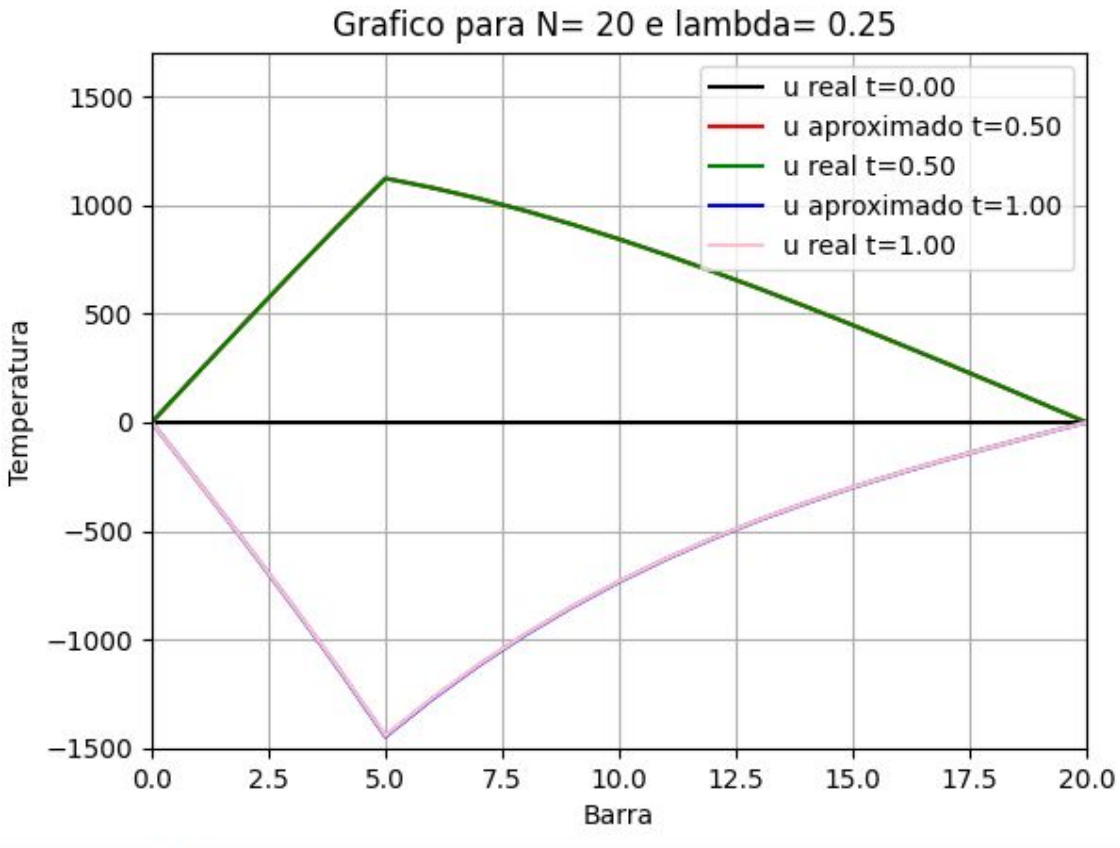

figura 29 - Gráfico da função (c) para N = 20 e λ = 0.25 com Crank-Nicolson

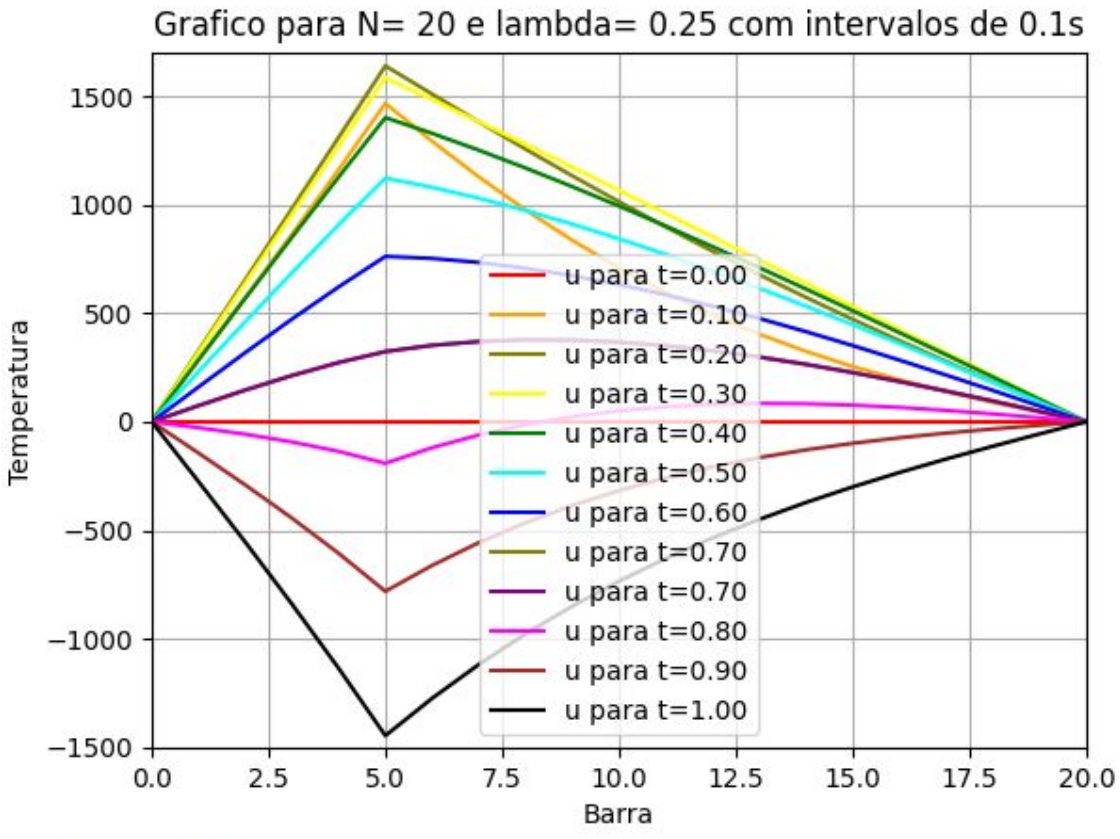

figura 30 - Gráfico temporal da função (c) para N = 20 e λ=0.25 com Crank-Nicolson

Maior erro de truncamento = 7.03e+03 ,em t= 9.99e-01 e x= 2.50e-01

Maior erro absoluto = 3.21e+00

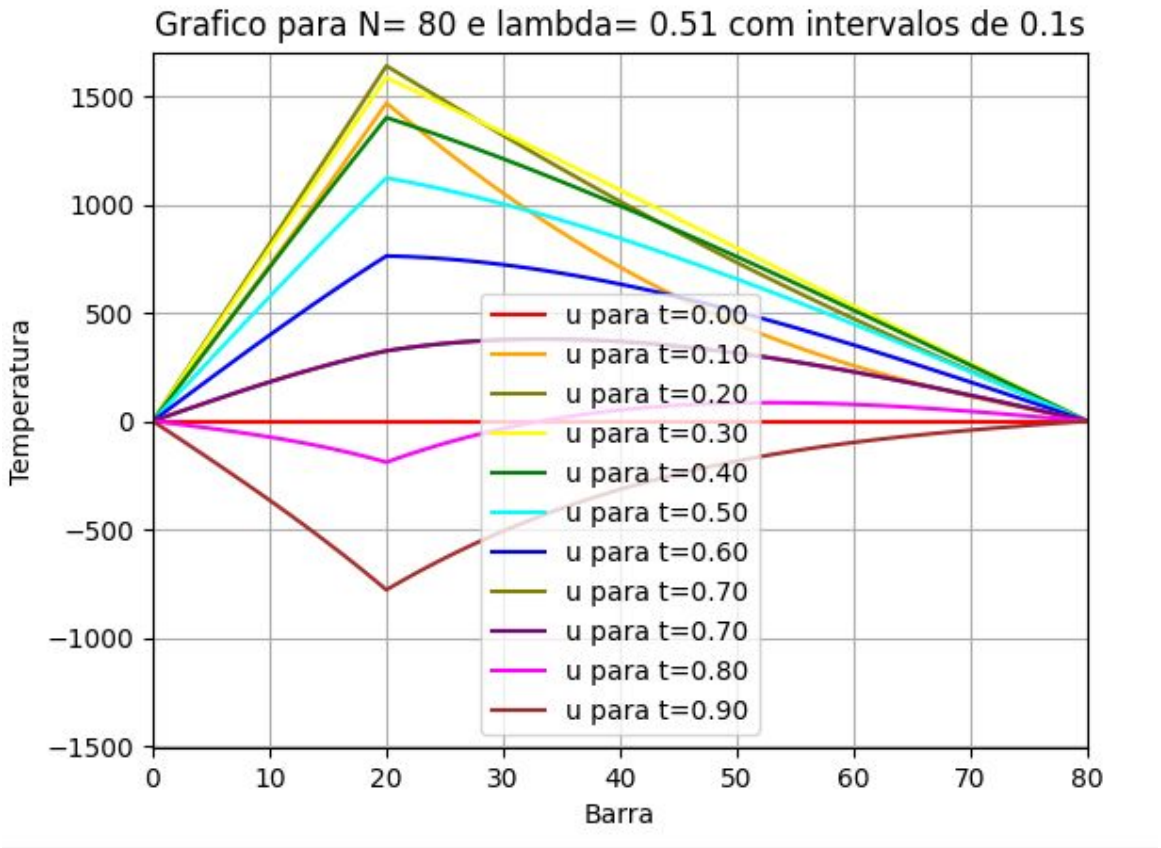

figura 31 - Gráfico temporal da função (c) para N = 80 e λ=0.51 com Crank-Nicolson