ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Exercício Programa de Cálculo Numérico (MAP3121)

Turma 2	No. USP
Lucas Haug	10773565
Renzo Armando dos Santos Abensur	10772414

São Paulo

2020

Lucas Haug
Renzo Armando dos Santos Abensur

Exercício Programa de Cálculo Numérico (MAP3121)

Relatório apresentado como requisito para avaliação na disciplina MAP3121 - Métodos Numéricos e Aplicações, no curso de Engenharia Elétrica oferecido pela Escola Politécnica da Universidade de São Paulo.

Professor: Nelson Mugayar Kuhl

São Paulo
2020

SUMÁRIO

1. Introdução

Este relatório procura especificar a resolução encontrada para o exercício programa da disciplina MAP3121 - Métodos Numéricos e Aplicações, no primeiro semestre de 2020.

O exercício programa procura analisar a evolução da distribuição da temperatura em uma barra ao longo do tempo, por meio das seguintes equações:

$$u_t(t,x) = u_{xx}(t,x) + f(t,x) \text{ em } [0,T] \times [0,1], \quad (1)$$
$$u(0,x) = u_0(x) \text{ em } [0,1] \quad (2)$$
$$u(t,0) = g_1(t) \text{ em } [0,T] \quad (3)$$
$$u(t,1) = g_2(t) \text{ em } [0,T]. \quad (4)$$

Para a resolução do problema foram utilizados métodos de cálculo numérico, envolvendo um método explícito, além do método de Euler implícito e o método de Crank-Nicolson.

O exercício programa foi elaborado em python 3, sendo separado em arquivos diferentes para melhor organizar o código.

2. Desenvolvimento

2.1 Primeira tarefa

Para a primeira parte do exercício programa, se analisou a evolução aproximada da equação de calor por meio de fórmulas de diferenças finitas. Para tal se utilizou a equação:

$$u_i^{k+1} = u_i^k + \Delta t(\frac{u_{i-1}^k - 2u_i^k + u_{i+1}^k}{\Delta x^2} + f(x_i, t_k)) \ , \quad i = 1, \cdots, N-1, \text{ e } k = 0, \cdots, M-1$$

Para se fazer esses cálculos, no código, foi feita uma matriz de tamanho N+1 por M+1. Sendo N e M valores relacionados à discretização do espaço e do tempo e escolhidos em tempo de execução. Essa matriz é inicializada primeiro com as condições de contorno e as condições iniciais do problema. Depois de inicializar, os pontos internos são calculados com a equação acima ao se percorrer a matriz, sendo feito da seguinte forma:

```
# Inside points calculation
for k in range(0, M):
    for i in range(1, N):
        U[k + 1][i] = U[k][i] + Δt * (((U[k][i - 1] - 2 * U[k][i] + U[k][i + 1]) / (Δx**2))
                      + pb.heat_source(time_array[k], x_array[i], N, letter))
```

Onde U é a matriz que guarda os valores da solução aproximada em cada instante de tempo e cada posição e a função **“heat_source”** retorna o valor da fonte de calor em um determinado instante em uma determinada posição para o problema em questão. Além disso, o vetor **”time_array”** possui todos os valores de tempo utilizados dada a discretização escolhida, enquanto o vetor **“x_array”** possui todos os valores de posição.

Além disso, foi feito o cálculo do erro de truncamento e do erro de aproximação, quando possível. Para o cálculo do erro de truncamento, se teve como base as seguintes equações:

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

$$\tau(\Delta t, \Delta x) = \max_{k,i} |\tau_i^k(\Delta t, \Delta x)|$$

Dessa forma, então, iterativamente foi calculado de cada posição da em cada instante, procurando-se assim o erro máximo de truncamento da seguinte forma se baseando nas equações acima:

```
# Truncation error calculation
max_truncation_error = 0

for k in range(0, M):
  for i in range(1, N):
    first_term = (pb.u_solution(time_array[k + 1], x_array[i], letter)
                  - pb.u_solution(time_array[k], x_array[i], letter)) / Δt
    second_term = (pb.u_solution(time_array[k], x_array[i - 1], letter)
                   - 2 * pb.u_solution(time_array[k], x_array[i], letter)
                   + pb.u_solution(time_array[k], x_array[i + 1], letter)) / (Δx**2)

    current_truncation_error = abs(first_term - second_term
                                   - pb.heat_source(time_array[k], x_array[i], N, letter))

    if current_truncation_error > max_truncation_error:
      max_truncation_error = current_truncation_error
```

Onde a função **“u_solution”** retorna o valor da solução exata em um determinado instante e em uma determinada posição para o problema em questão.

Ṕor fim, para o cálculo do erro de aproximação da solução, se teve como base as seguintes equações:

$$e_i^k = u(t_k, x_i) - u_i^k$$

$$||e^k|| = \max_i |e_i^k|$$

Com as equações acima, considerando-se o instante de tempo t = 1 para o cálculo do erro máximo de aproximação, foi elaborada a seguinte lógica para se procurar qual o maior erro de aproximação:

```
# Approximation error calculation for T = 1
max_approx_error = 0

for i in range(0, N):
  current_approx_error = abs(pb.u_solution(time_array[M], x_array[i], letter) - U[M][i])

  if current_approx_error > max_approx_error:
     max_approx_error = current_approx_error
```

Para facilitar a intercompatibilidade de cada método com os diferentes problemas, foram feitas as seguintes funções (algumas já foram mencionadas acima), que têm retornos diferentes dependendo do problema a ser analisado, tendo como parâmetros de entrada em geral um determinado instante e uma determinada posição. Sendo elas:

- heat_source →retorna o valor da fonte de calor.

- u_solution →retorna o valor da solução exata.
- initial conditions →retorna o valor da condição inicial numa posição.
- boundary conditions →retorna o valor das condições de contorno em um instante.
- create u → retorna uma matriz com as condições de contorno e condições iniciais já preenchidas.

Item a)

Ao testarmos o programa com T=1,

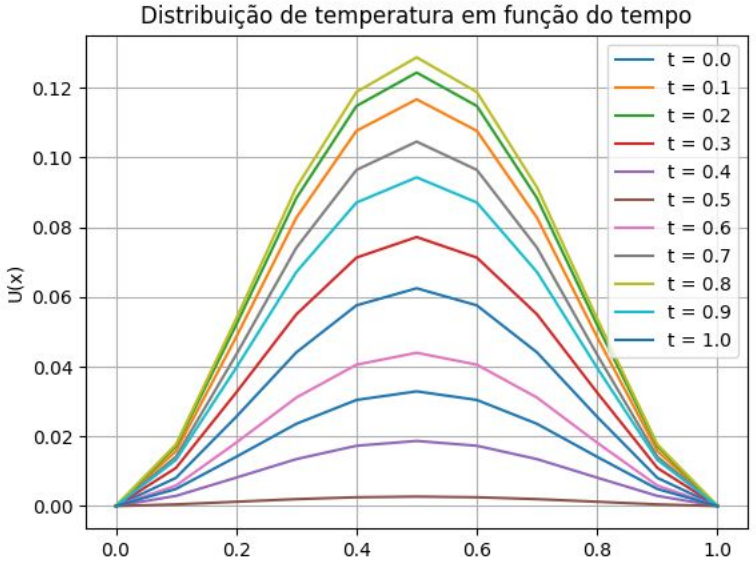
$f(t,\ x)\ =\ 10\ *\ cos(10t)x^{2}(1-x)^{2}-(1+sin(10t))(12x^{2}-12x+2)$,

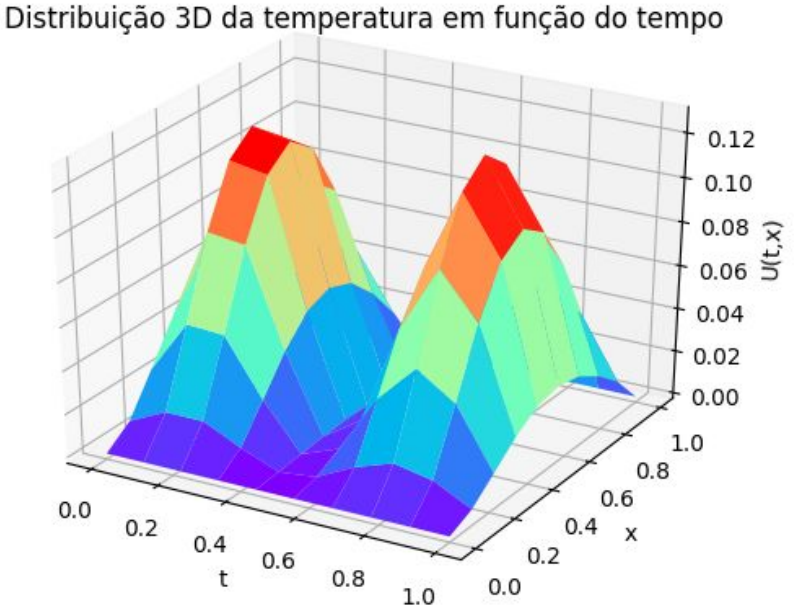
$u(t,\ x)\ =\ (1+sen(10t))x^{2}(1-x^{2})$, $u_{0}(x)\ =\ x^{2}(1-x)^{2}$.

Foram feitos testes para essa fonte de calor para λ = 0.25, λ = 0.5 e λ = 0.51. Além disso foram feitos testes para diferentes valores de N: 10, 20, 40, 80, 160, 320.

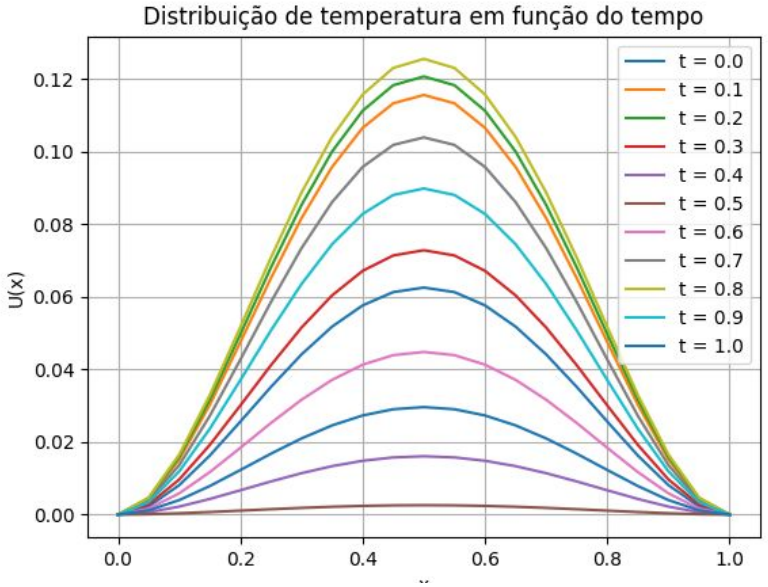
Dessa forma foram obtidas as seguintes soluções, podendo-se ver tanto um gráfico 2D com diferentes instantes de tempo, assim como um gráfico 3D mostrando a variação da temperatura no espaço e no tempo:

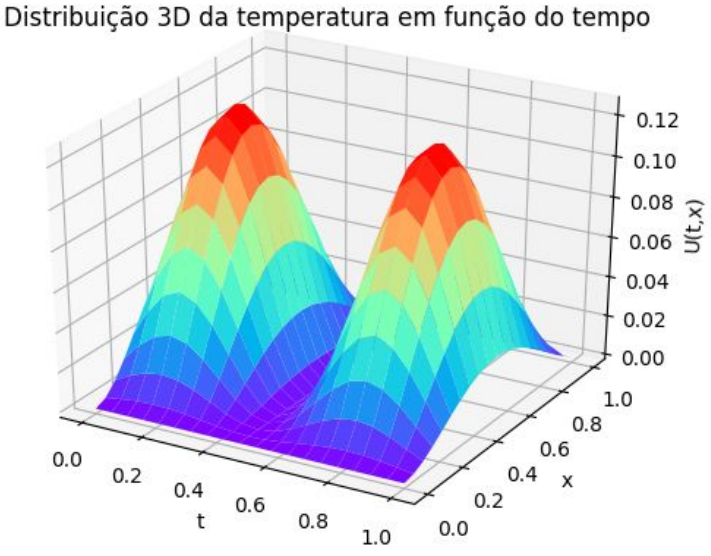
N = 10 e λ = 0.25


Distribuição de temperatura em função do tempo
U(x)
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

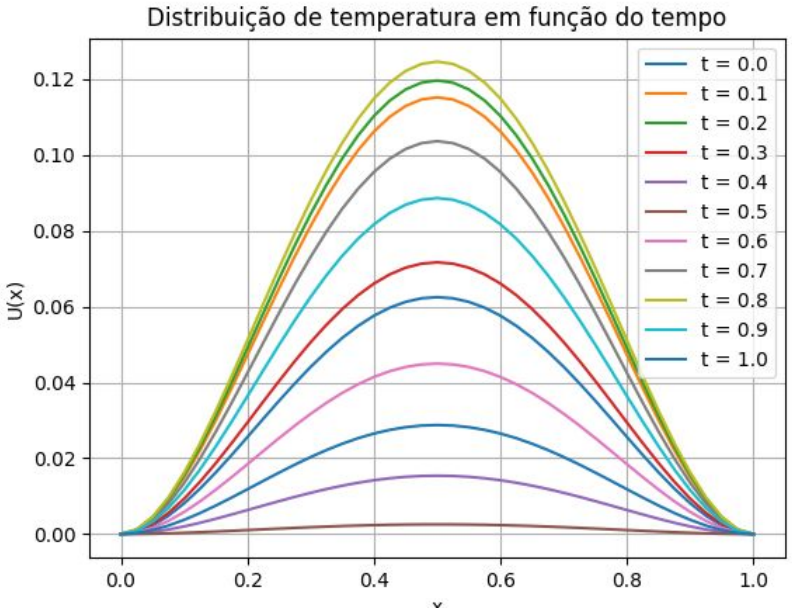

Distribuição 3D da temperatura em função do tempo
U(t,x)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
t
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0

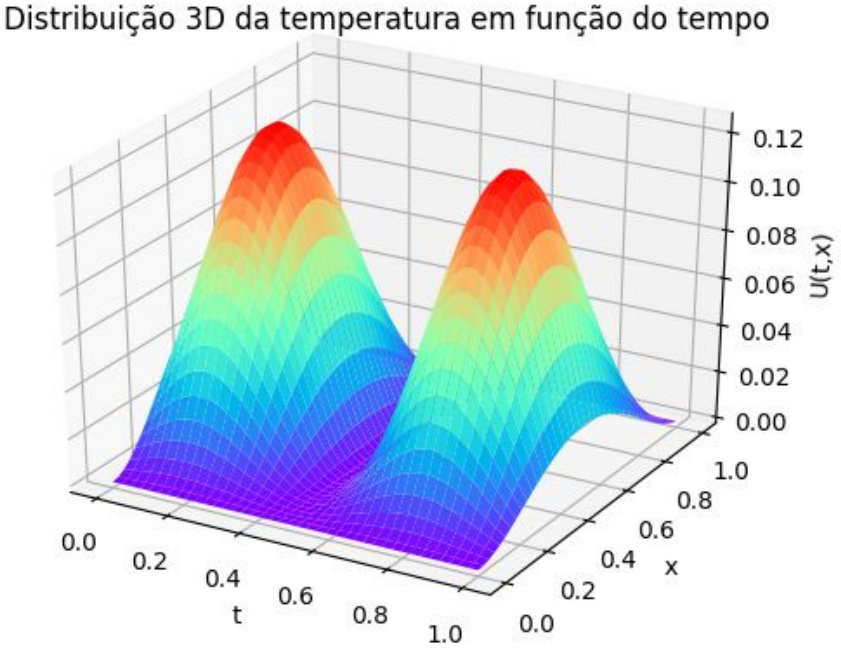
N = 20 e λ = 0.25


Distribuição de temperatura em função do tempo
U(x)
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0.0
0.2
0.4
0.6
0.8
1.0
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0

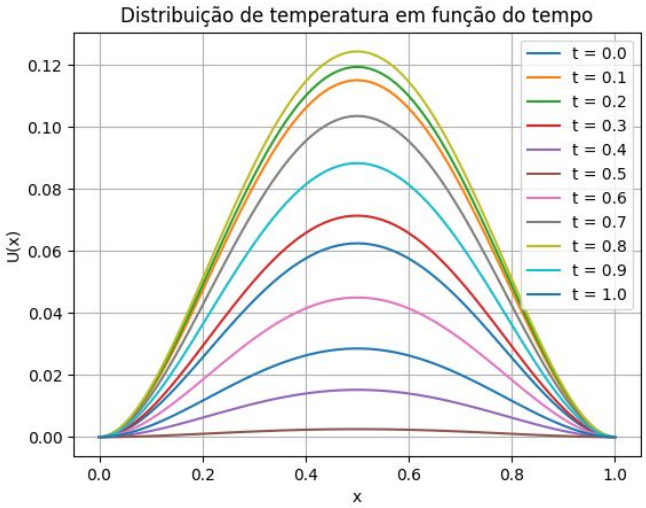

Distribuição 3D da temperatura em função do tempo
t
x
U(t,x)
0.0
0.2
0.4
0.6
0.8
1.0
0.12
0.10
0.08
0.06
0.04
0.02
0.00

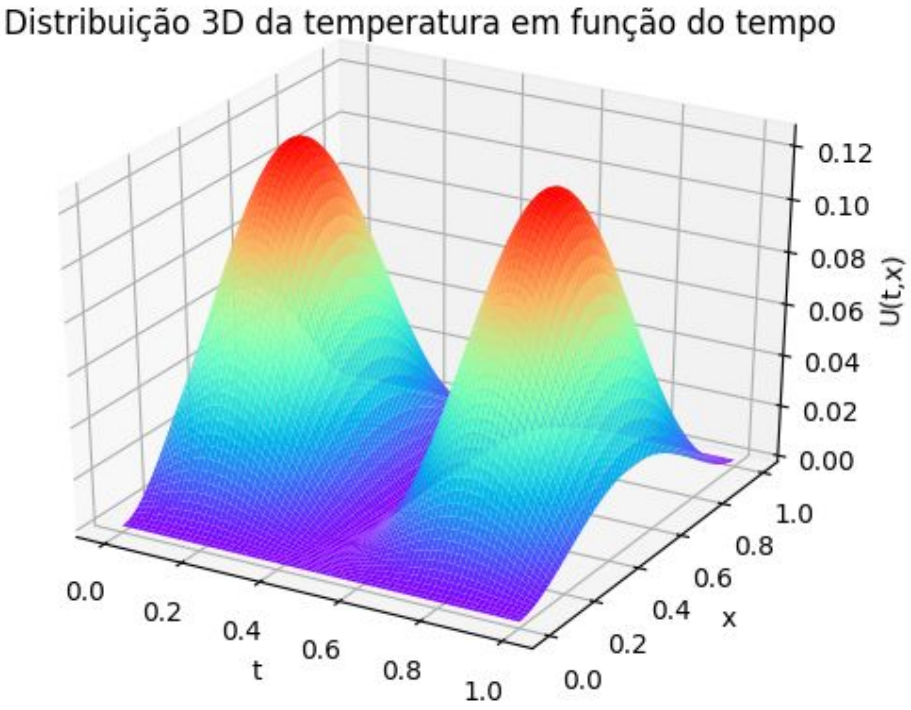
N = 40 e λ = 0.25


Distribuição de temperatura em função do tempo
U(x)
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

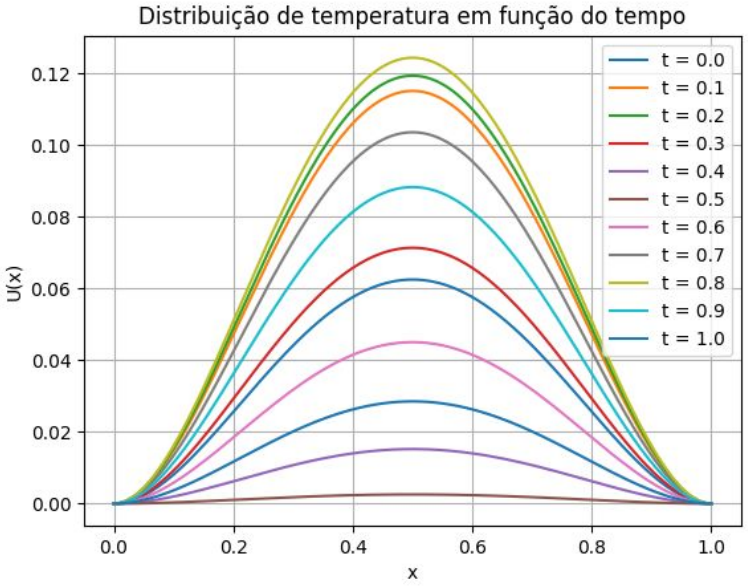

Distribuição 3D da temperatura em função do tempo
U(t,x)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
t
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0

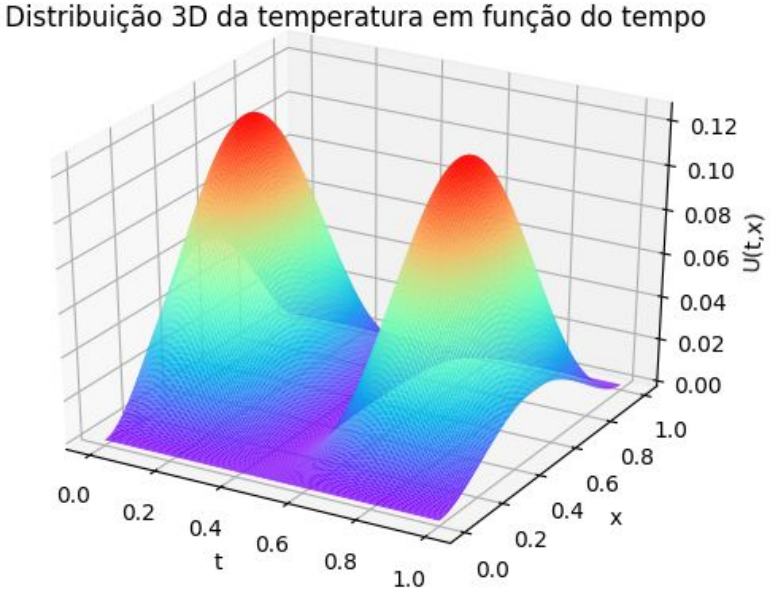
N = 80 e λ = 0.25


Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

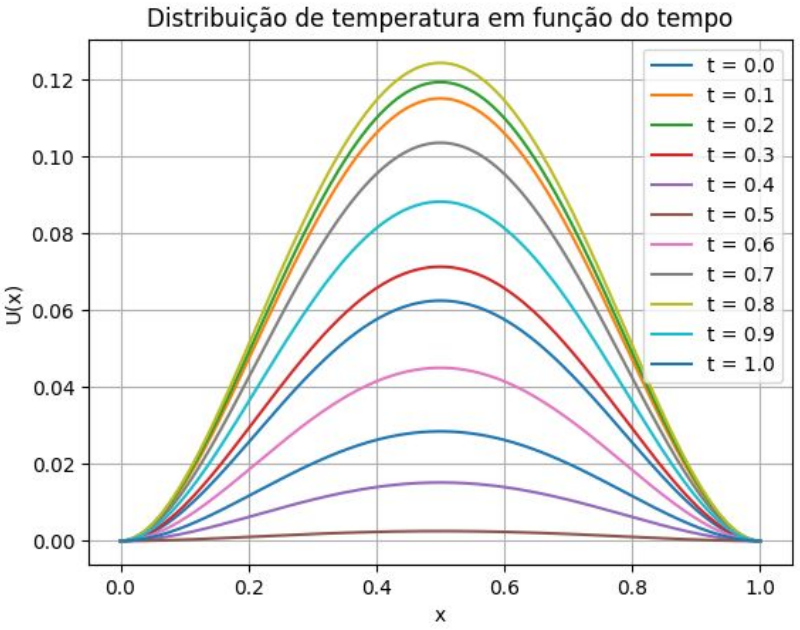

Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

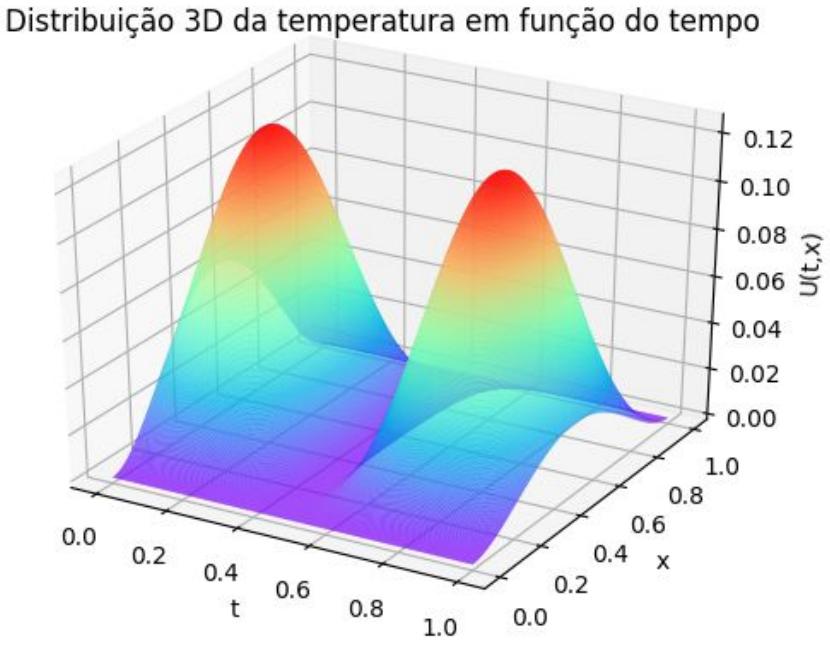
N = 160 e λ = 0.25


Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

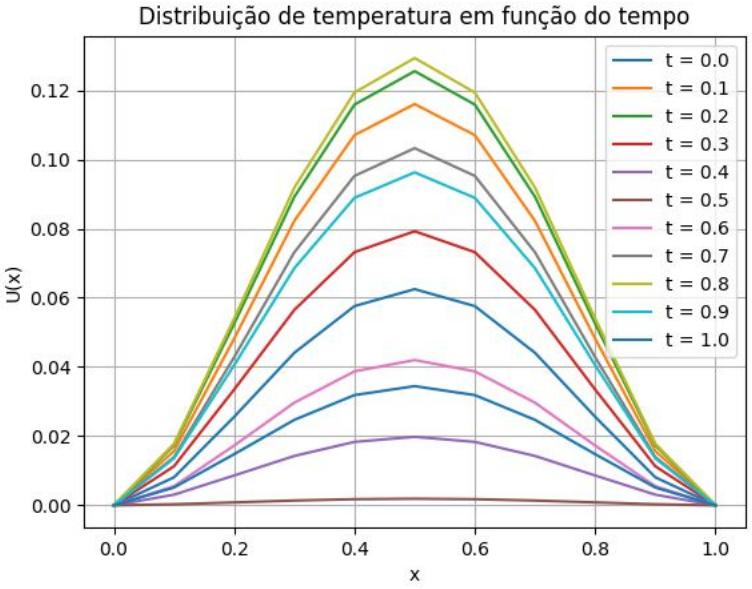

Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

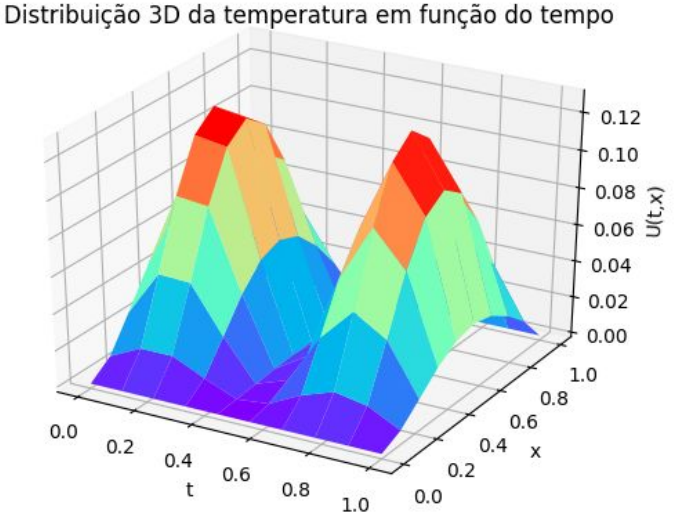
N = 320 e λ = 0.25


Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

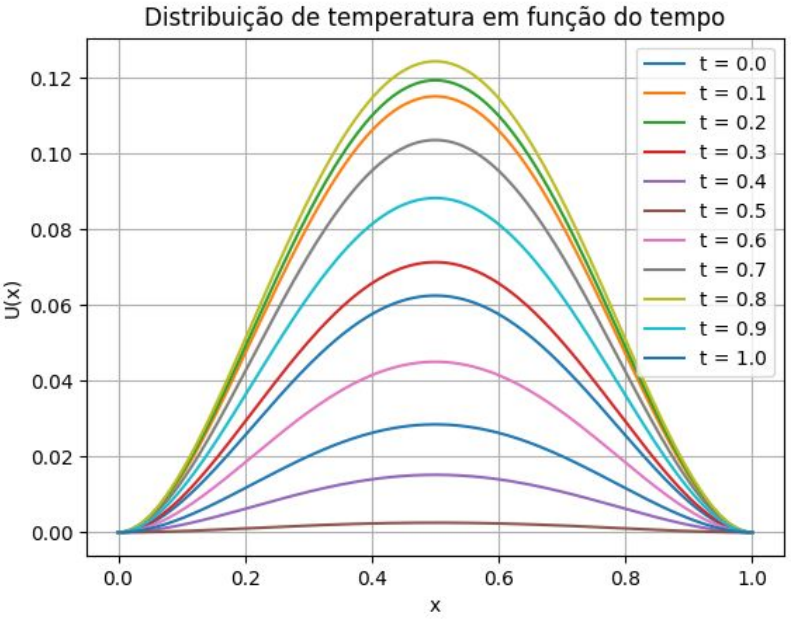

Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

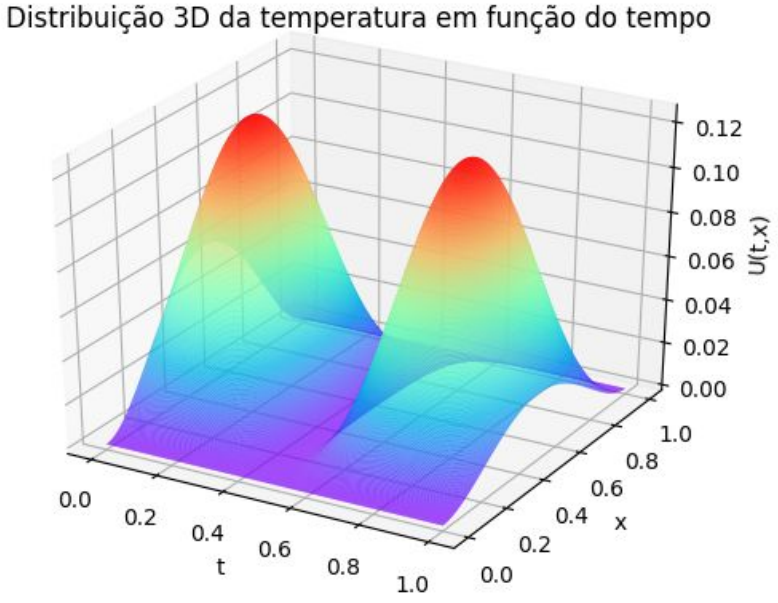
N = 10 e λ = 0.5


Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0
x

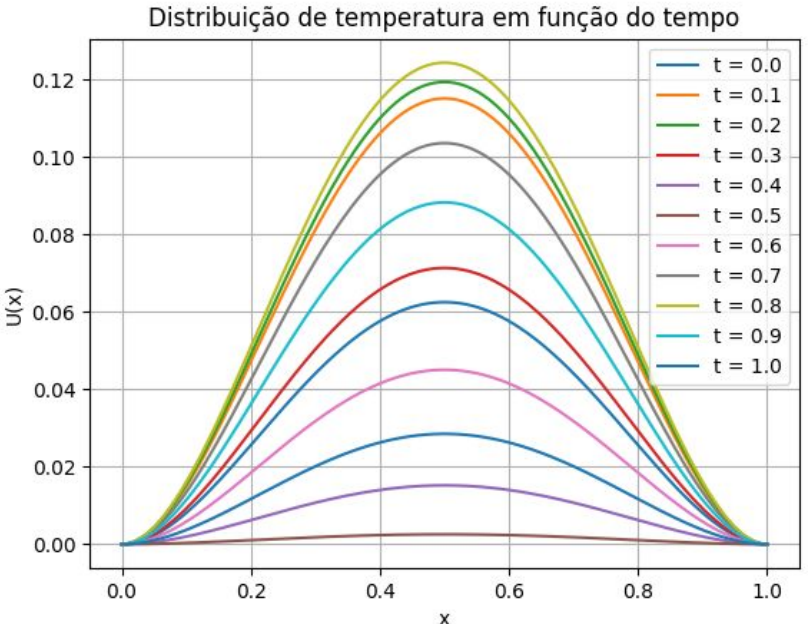

Distribuição 3D da temperatura em função do tempo
U(t,x)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
t
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0

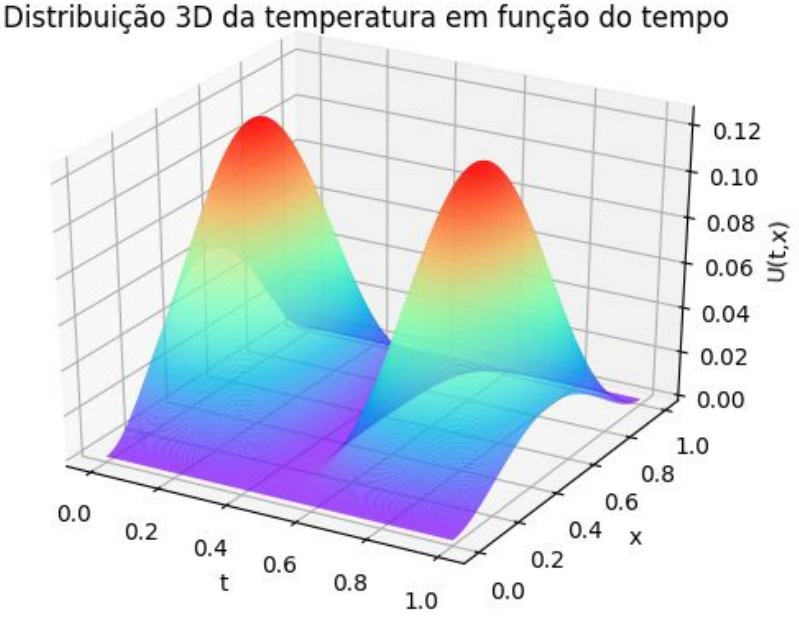
N = 320 e λ = 0.5


Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0


Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.0
0.2
0.4
0.6
0.8
1.0

Com a variação de N entre 10 a 320, podemos observar que tanto para o λ =0.25 quanto para o λ = 0.5 que os gráficos de distribuição de temperatura em função do tempo 2D e 3D tendem a tornar-se mais homogêneos com o aumento do N, ou seja, deixando-se o espaço e o tempo menos discretos, aproximando-se da solução exata, dada por $u(t,\ x)\ =\ (1+sen(10t))x^{2}(1-x^{2})$, a qual foi plotada e pode ser vista abaixo:





Cálculos dos erros

Para λ = 0.25 foram encontrados os seguintes valores de erro de truncamento e erro de aproximação, para valores de N diferentes:

N	Erro truncamento	Erro máximo de aproximação
10	0.04781213186166444	0.0030467842090773634
20	0.011953112410774569	0.0007578239365267428
40	0.00298828107030058	0.00018921504910418552
80	0.0007470703107266274	4.72887202614819e-05
160	0.00018676757901525676	1.182124019476205e-05
320	4.6691906011808726e-05	2.955251310773205e-06

Olhando na tabela, pode-se ver que a relação do erro de truncamento com o N. Ao se duplicar o valor do N, pode-se ver que o erro diminui quatro vezes, dessa forma com um valor de N muito elevado, o erro de truncamento tende a zero. Da mesma, o erro de aproximação da solução também diminui quatro vezes ao se dobrar o N.

Ao se dispor os dados da tabela acima em gráficos em escala logarítmica, pode-se ver o comportamento do erro descrito:

Gráfico para o erro de truncamento

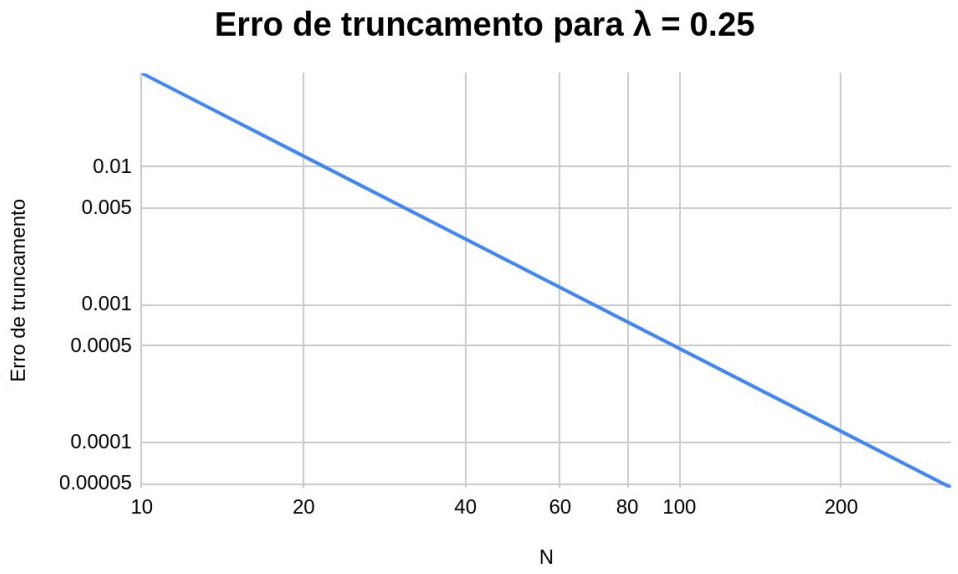
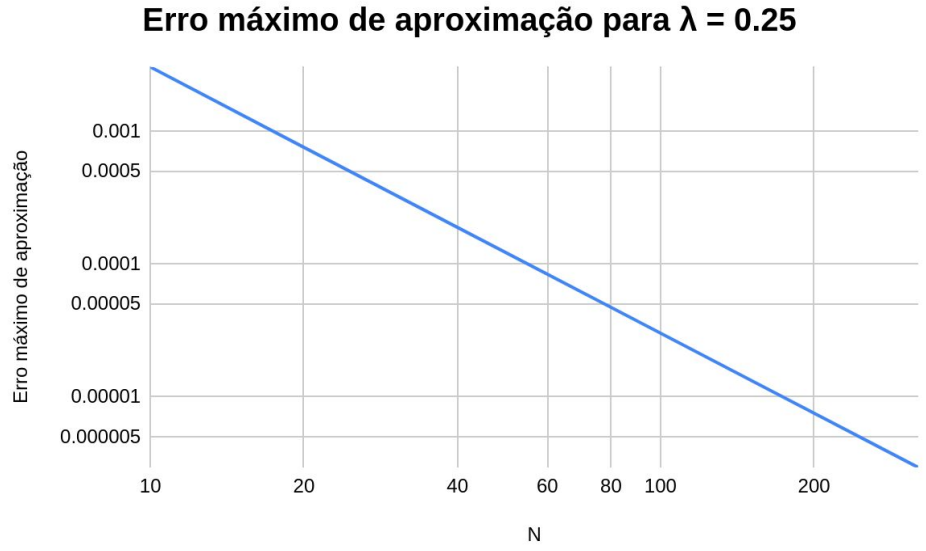


Gráfico para o erro de aproximação



Já com λ = 0.5, obteve-se os seguintes valores para os erros máximos de truncamento e de aproximação:

N	Erro truncamento	Erro máximo de aproximação
10	0.05562249945175868	0.0031637147966675702
20	0.01390619334550447	0.0007842085365461181
40	0.003476561759728547	0.0001956372990580199
80	0.0008691406157019799	4.8883476745417015e-05
160	0.0002172851568467138	1.2219254915624145e-05
320	5.4321293740722254e-05	3.054712856203484e-06

Pode-se observar que, assim como para λ = 0.25, o erro diminui quatro vezes ao se dobrar o valor do N, porém, pode ser ver que os valores do erros são ligeiramente maiores quando comparados com cada um dos casos em que λ = 0.25.

Para esses valores de erros, obtemos os seguintes gráficos em escala logarítmica:

Gráfico para o erro de truncamento

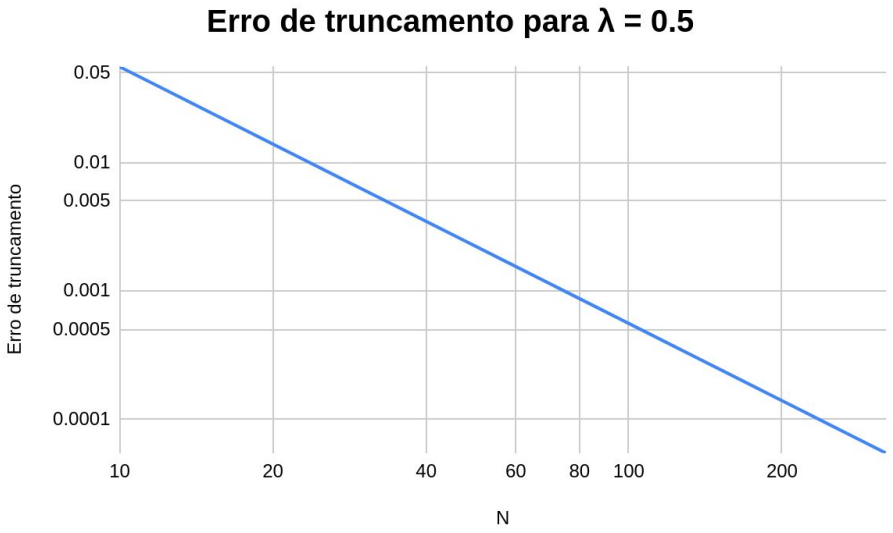
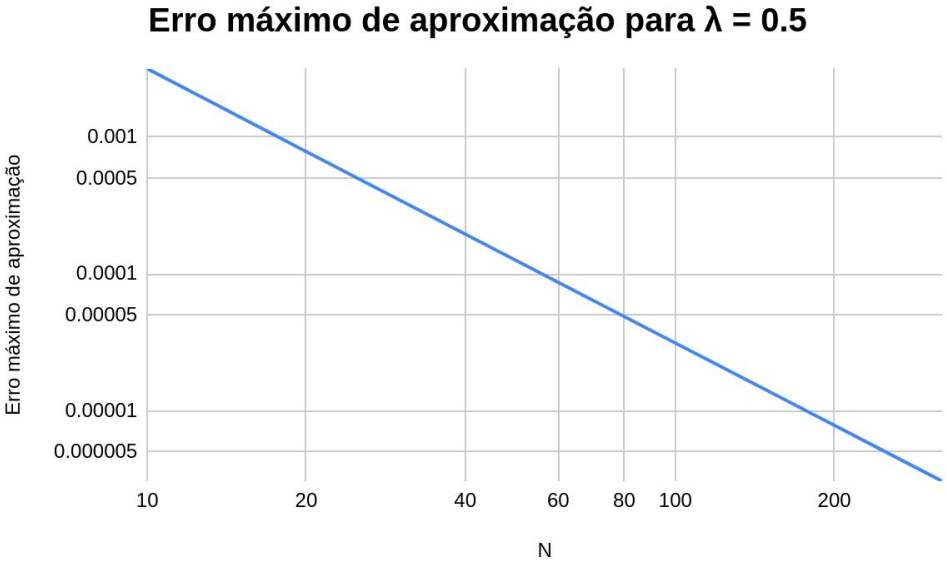


Gráfico para o erro de aproximação



Podemos observar pelos gráficos dos erros truncamentos e erros máximo de aproximação que esses diminuem linearmente em escala logarítmica com o aumento de N, porém quanto a variação do λ de 0 .25 para 0.5, não houve uma modificação significativa na forma como o erro decai, somente nos valores em si.

Estabilidade do método

Para testar a estabilidade do método, foi feito um teste com com $\lambda = 0.51$

Pode-se ver que o gráfico fica da seguinte forma com N = 20:

Gráfico para a solução em 2D

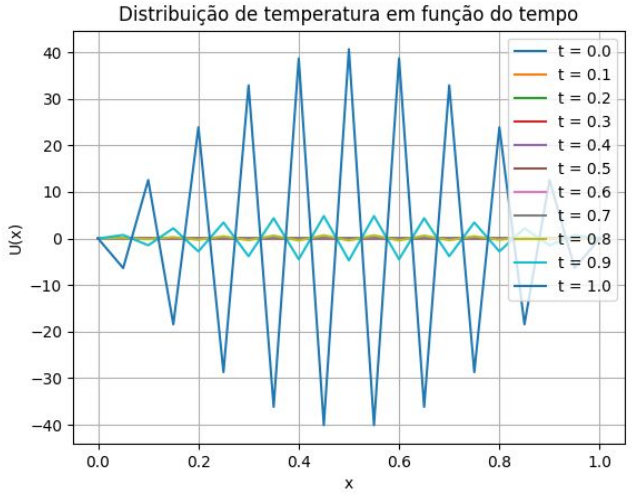
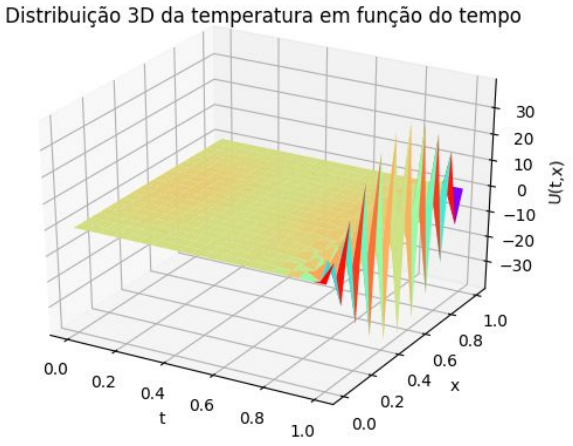


Gráfico para a solução em 3D



Além disso foram calculados alguns valores de erros para diferentes valores de N:

N	erro truncamento	erro máximo de aproximação
10	0.05593404086491027	0.003179075442371525
20	0.013984335016362204	40.60403679669783
40	0.00349609288689412	7.803984453547832e+39
80	0.0008740234279343007	2.5499357008114193e+198

Ao testarmos o código com λ = 0.51 podemos observar que os gráfico ficam instáveis, não correspondendo ao esperado, além de os valores dos erros de aproximações irem se amplificando enormemente com o aumento do N. Isso ocorre, uma vez que, o método utilizado para gerar a resolução do problema é o "condicionalmente convergente", em que a condição de $\lambda = \Delta t \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$ deve ser respeitada para conseguirmos chegar na solução do problema.

Fator de redução

Como foi visto anteriormente, ao para cada refinamento de malha, ou seja ao se dobrar o N, o erro cai pela quatro vezes, sendo esse o fator de redução do método.

Número de passos

Para esse método, o número de passos necessário para se resolver o problema é dado por $M * (N-1)$. Dessa forma pode-se ver o alto custo computacional do método, uma vez que M é dado por $M = \frac{N^2}{\lambda}$, então o número de passos pode ser calculado por $(N^3 - N^2) / \lambda$.

Dessa forma, o número de passos necessários ao se usar N = 640 é dado por seria de 1046937600 para λ = 0.25 , e se dobrarmos o N teríamos 4191027200 passos para o mesmo λ.

Item b)

Dada a solução exata igual a $u(t, x) = e^{(t-x)} * cos(5 * t * x)$, foi necessário encontrar os valores de $u_0(x)$, $g_1(t)$, $g_2(t)$ e $f(t, x)$.

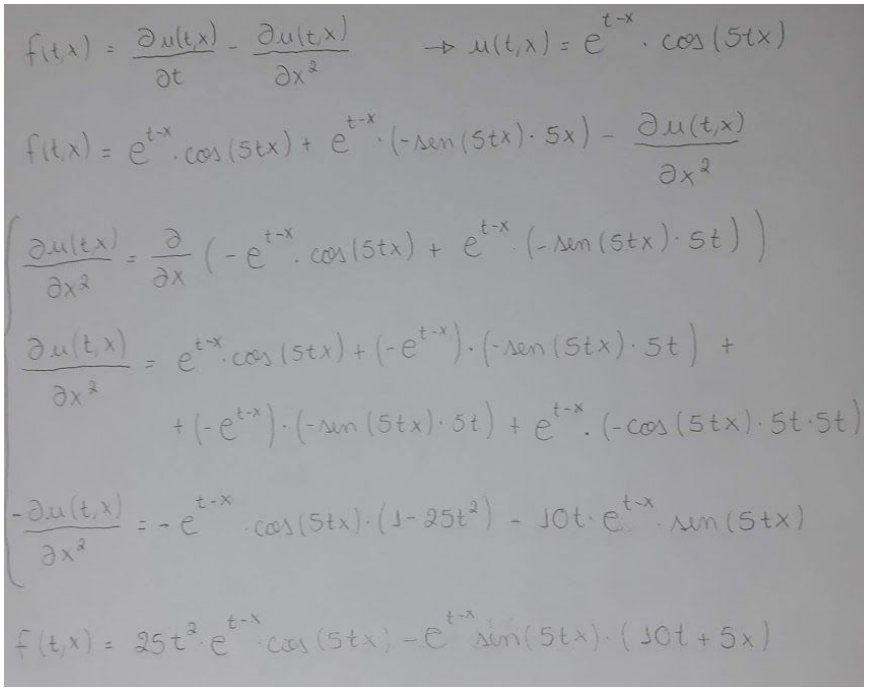
Para encontrar o valor de $u_0(x)$ foi necessário substituir na solução exata t = 0. Já para as condições de contorno $g_1(t)$ e $g_2(t)$, se substituiu na solução exata, os valores de x = 0 e x = 1, obtendo-se:

$$u_0(x) = e^{-x}$$
$$g_1(t) = e^{t}$$
$$g_1(t) = e^{(t-1)} * cos(5t)$$

Para o cálculo de $f(t, x)$ foi usada a seguinte equação:

$$u_t(t,x) = u_{xx}(t,x) + f(t,x)$$

Dessa forma, foram feitas as seguintes contas para se obter a função da fonte de calor:



Obtendo portanto:

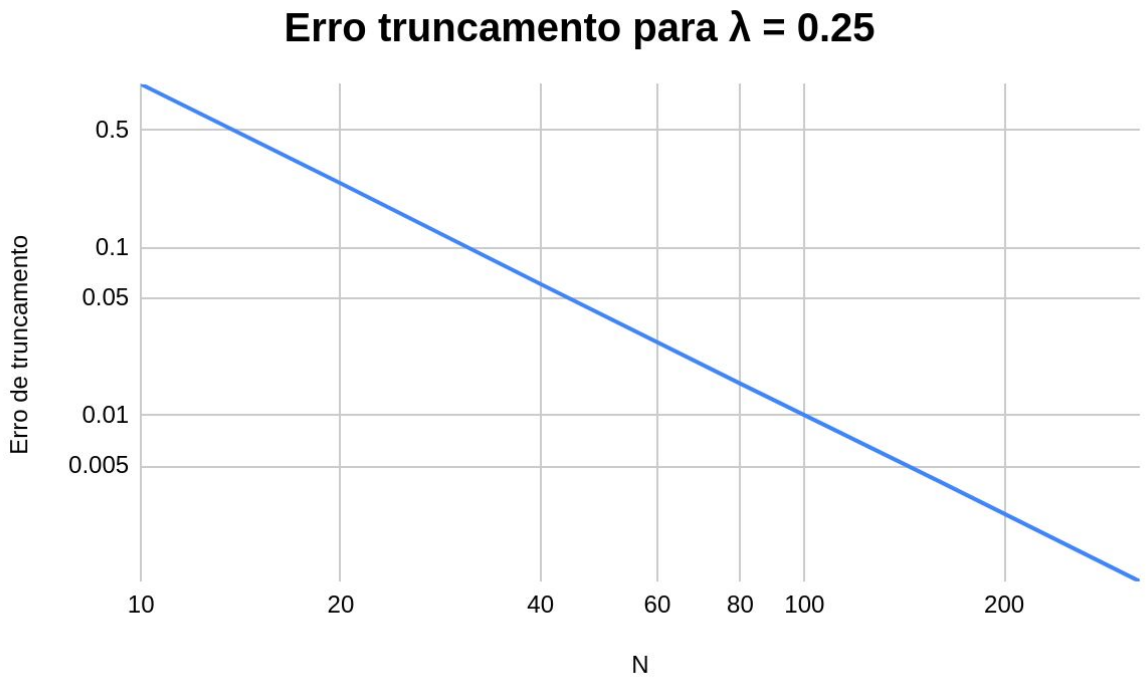
$$f(t,\ x)\ =\ 5 * e^{(t-x)} * ((5t^2 * cos(5 * t * x)) - (sin(5 * t * x)(2 * t + x)))$$

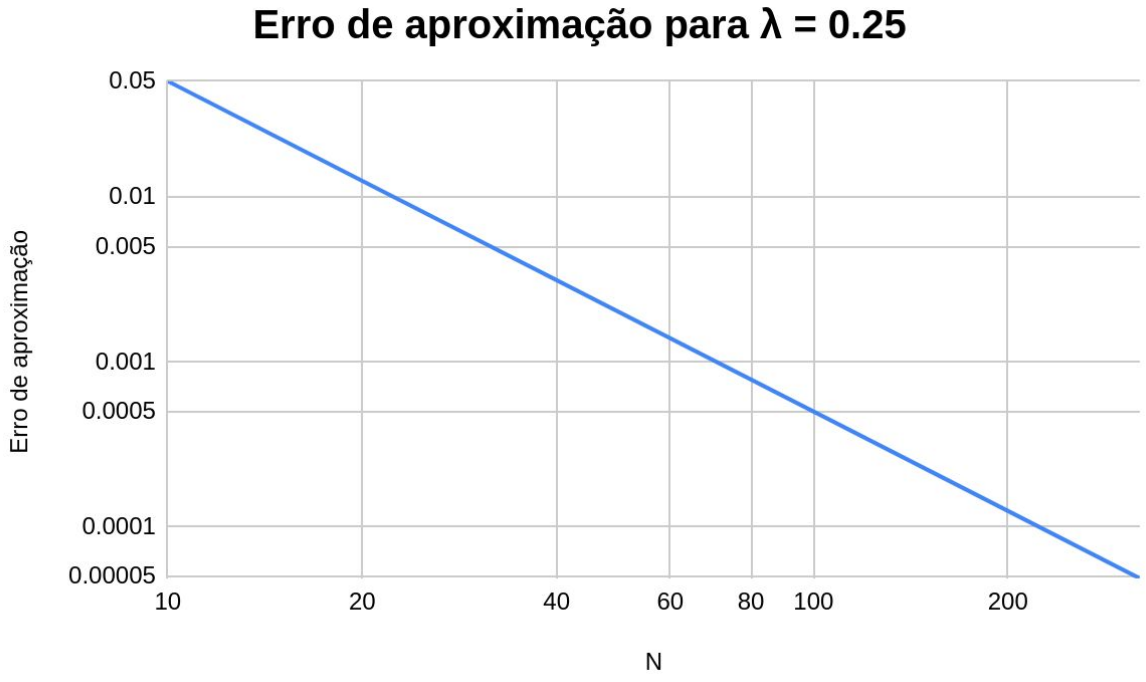
Repetindo os experimentos da parte a) temos:

Com λ = 0.25

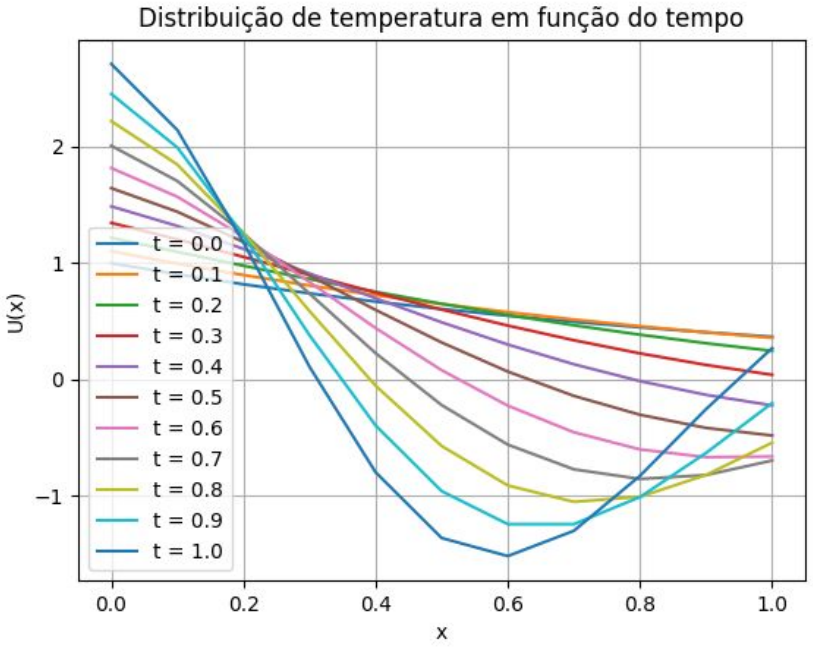
N	erro truncamento	erro máximo de aproximação
10	0.9366592480833589	0.05004831210205496
20	0.241052334904289	0.01249945308638023
40	0.0606369798395221	0.0031285343222757778
80	0.015501788268963423	0.0007818727804080883
160	0.004038316758141036	0.0001954955584333451
320	0.0010297042071982787	4.887286255383927e-05

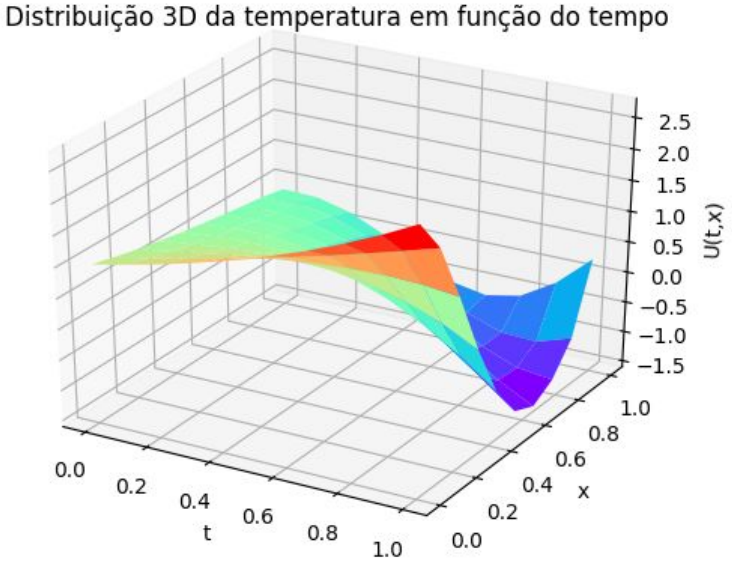
Obtendo-se o seguintes gráficos em escala logarítmica para os erros:



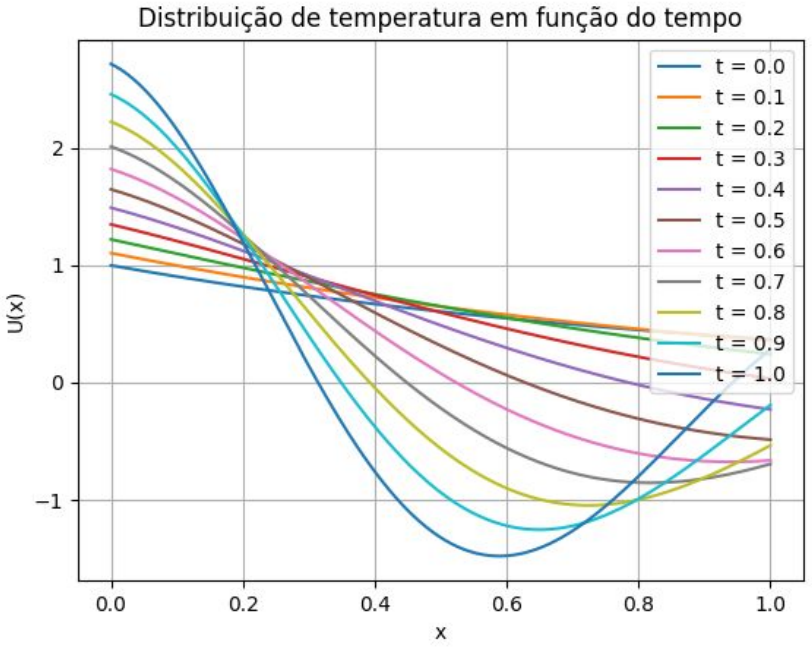


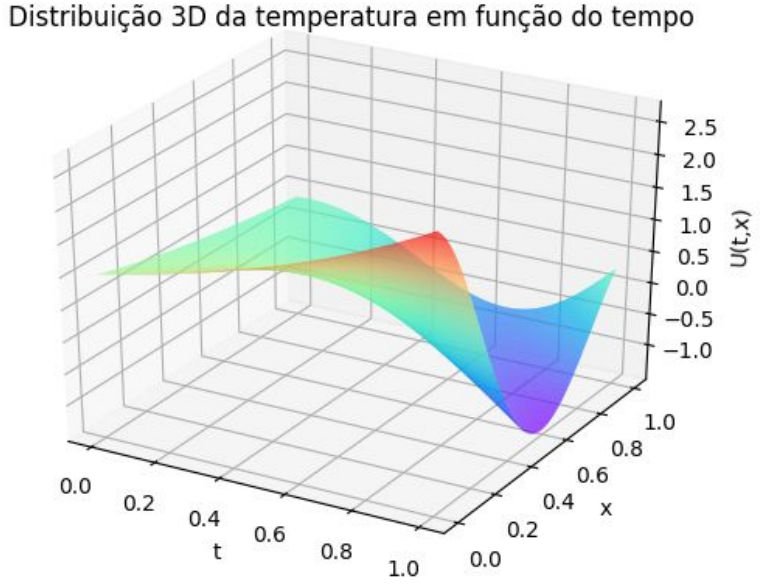
N = 10


Distribuição de temperatura em função do tempo
U(x)
x
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0


Distribuição 3D da temperatura em função do tempo
t
x
U(t,x)

N = 320

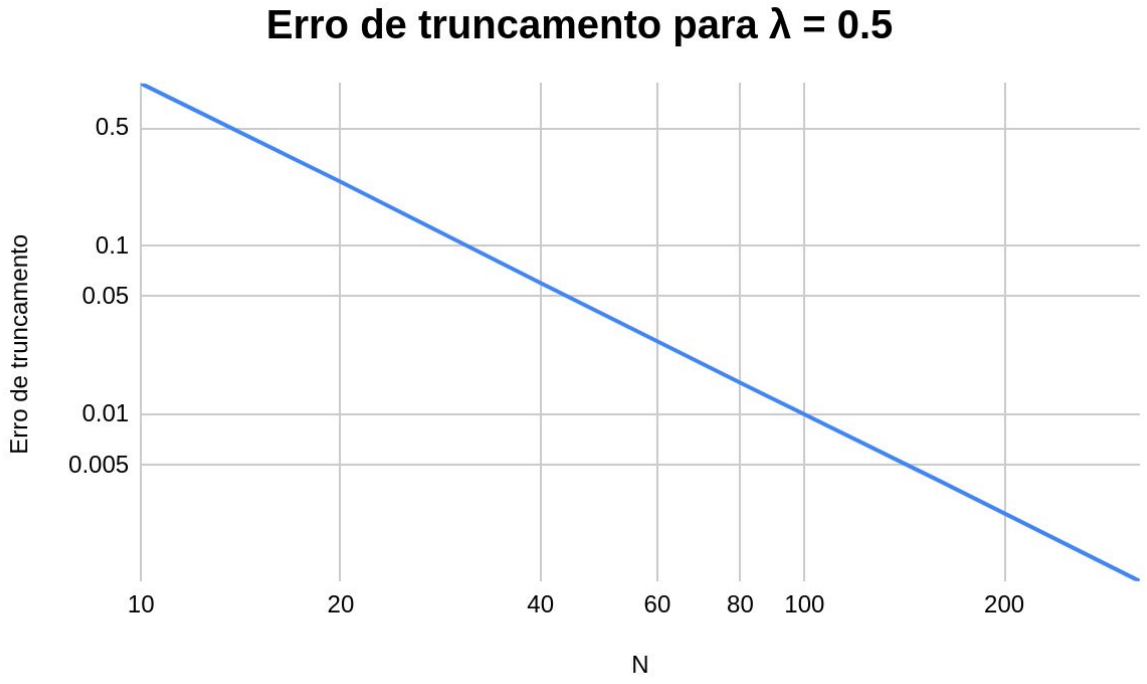

Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
x
2
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0

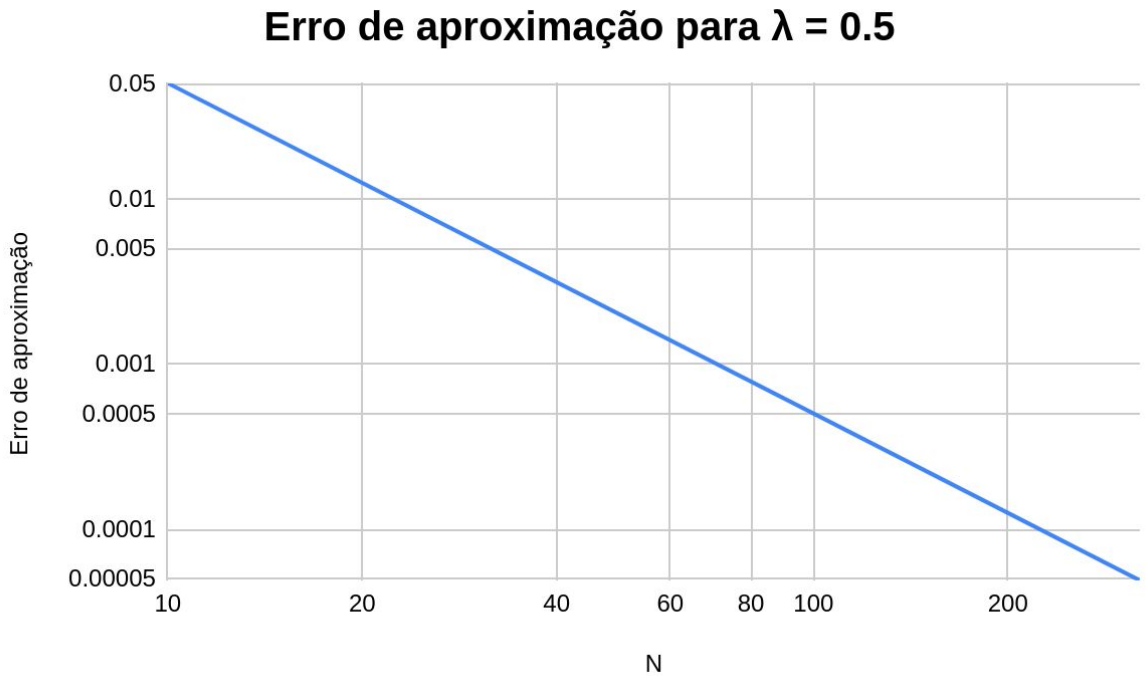

Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

Já para λ = 0.5

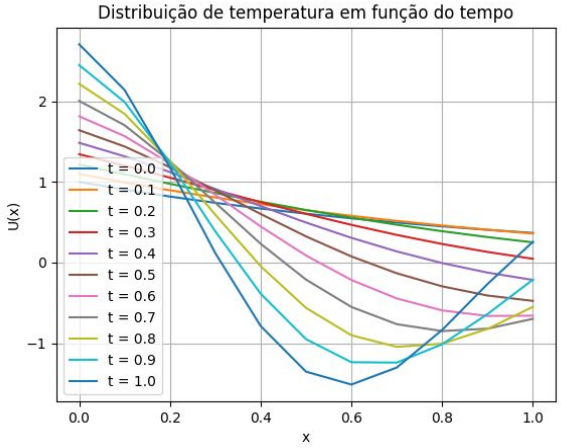
N	erro truncamento	erro máximo de aproximação
10	0.9187391190443677	0.050452152186500676
20	0.23977592223298672	0.012582235580246737
40	0.06034264115535137	0.003153736723830347
80	0.015446641641233327	0.0007882212592800197
160	0.004024948856212518	0.00019704199468506545
320	0.0010263850685703346	4.92612910780732e-05

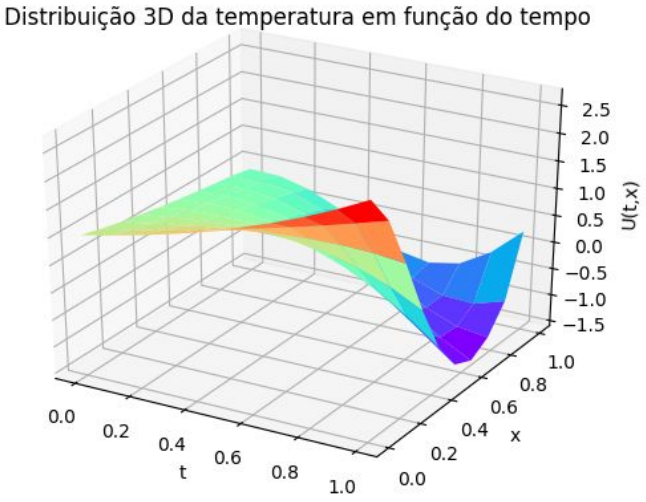
Obtendo-se os seguintes gráficos em escala logarítmica para os erros:



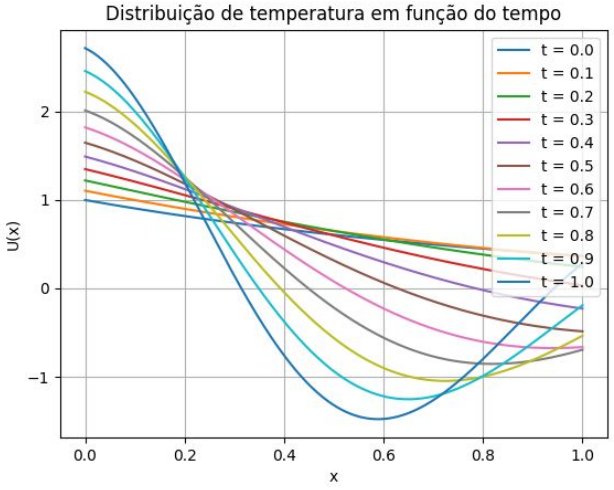

Erro de aproximação para λ = 0.5
Erro de aproximação
0.05
0.01
0.005
0.001
0.0005
0.0001
0.00005
10
20
40
60
80
100
200
N

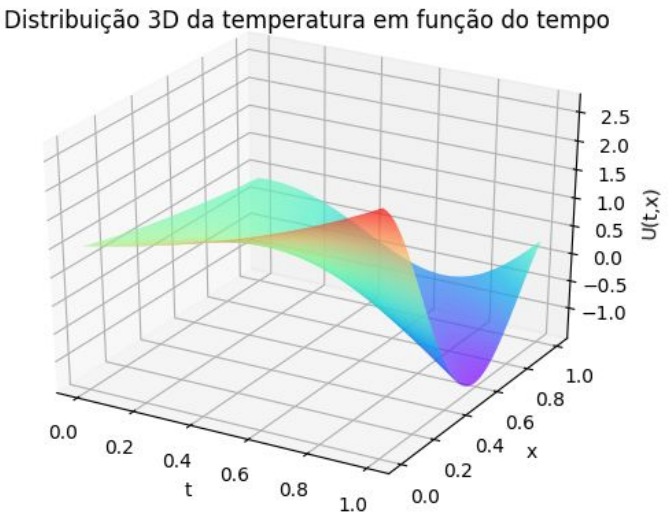
N = 10


Distribuição de temperatura em função do tempo
U(x)
x
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0


Distribuição 3D da temperatura em função do tempo
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
U(t,x)
0.0
0.2
0.4
0.6
0.8
1.0
t
0.0
0.2
0.4
0.6
0.8
1.0
x

N = 320


Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
2
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0
x



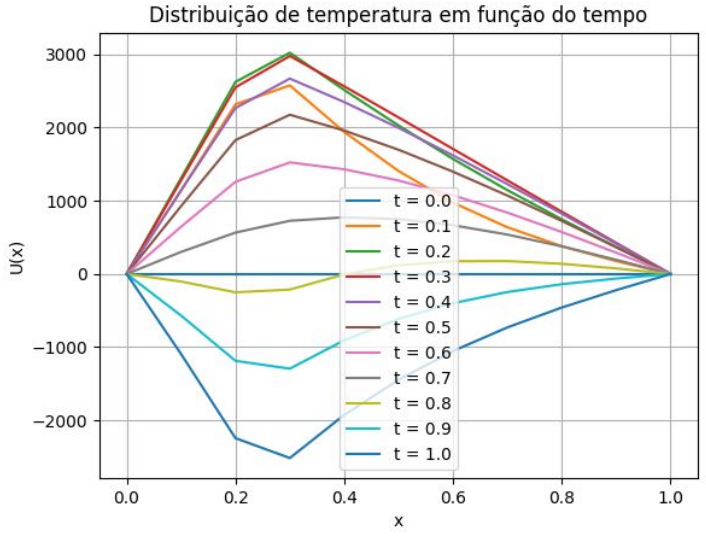
Item c)

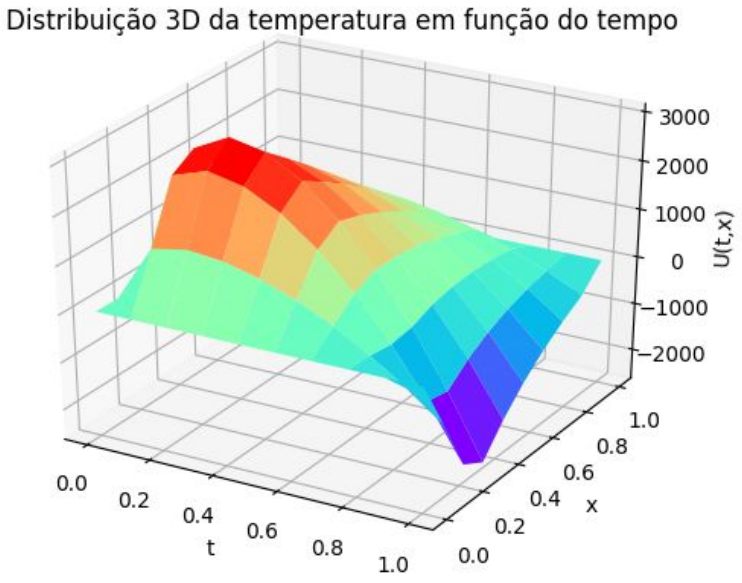
Para o item c foi necessário criar uma fonte pontual, a qual foi colocada no código da seguinte maneira:

```
r = 10000 * (1 - 2 * (t**2))

h = 1 / N
p = 0.25

if (p - h / 2) <= x <= (p + h / 2):
    gh = N
else:
    gh = 0

result = r * gh
```

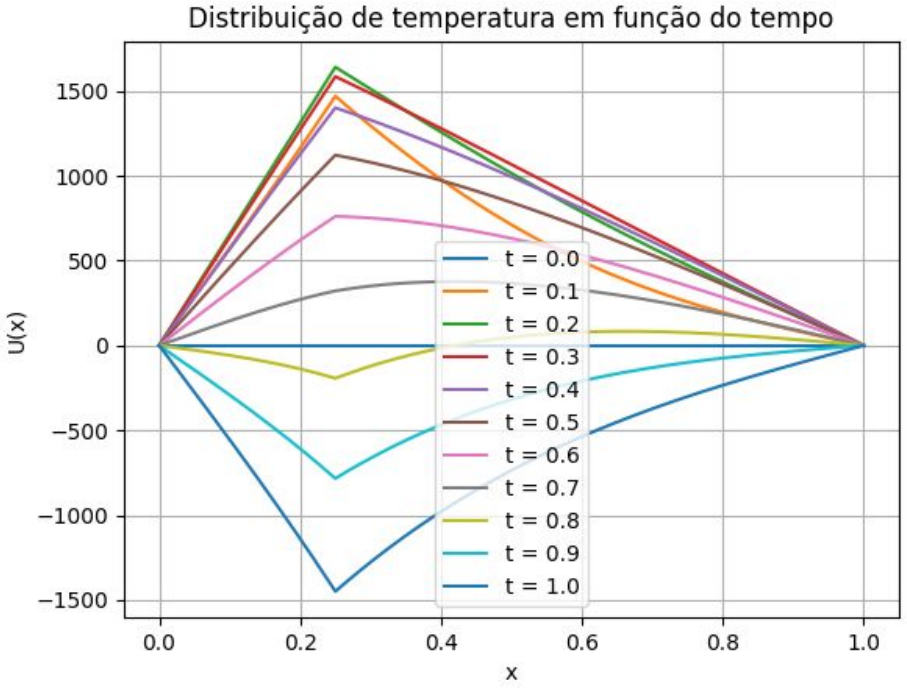
Então foram plotados os gráficos, para se obter a solução do problema numericamente, chegando nos seguintes gráficos:

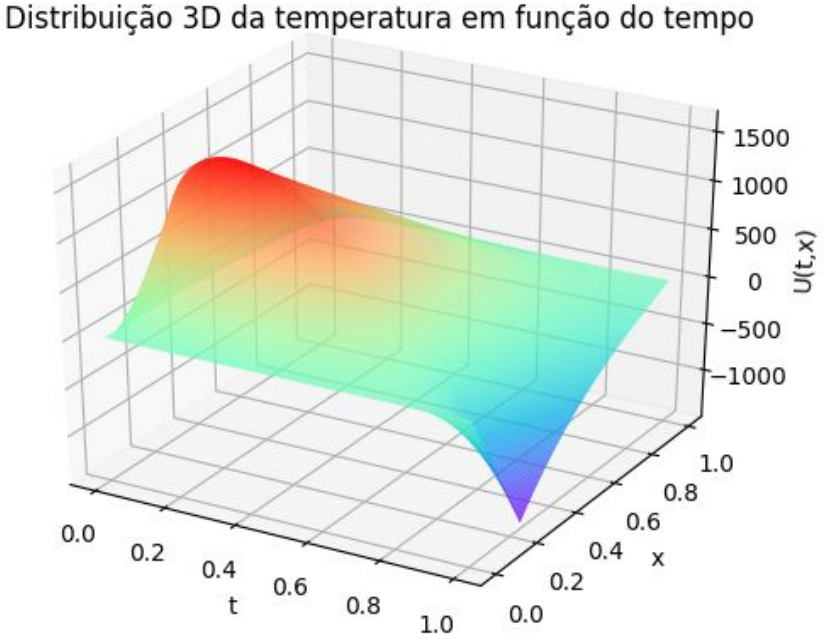
N = 10 e λ = 0.25





N = 320 λ = 0.25


Distribuição de temperatura em função do tempo
U(x)
x
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0


Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x

2.2 Segunda Tarefa

Na segundo parte da tarefa o foco foi a implementação de dois novos métodos para o mesmo problema, procurando assim encontrar as vantagens ou desvantagens que os métodos de Euler implícito e o de Crank-Nicolson possuem em comparação ao método de Condicionalmente convergente.

Item a)

Para implementação a implementação da função que decompõe a matriz tridiagonal A foi feita a seguinte função:

```
def matrix_decomposition(a_matrix_diag, a_matrix_subdiag):
  """
  Decompõe uma matrix A tridiagonal simétrica em três matrizes
  L, D e Lt, retornando apenas dois vetores que representam as
  matrizes L e D.
  """

  array_size = len(a_matrix_diag)

  l_matrix_array = np.zeros(array_size, dtype=float)
  d_matrix_array = np.zeros(array_size, dtype=float)

  l_matrix_array[0] = 0
  d_matrix_array[0] = a_matrix_diag[0]

  for i in range(1, array_size):
      l_matrix_array[i] = a_matrix_subdiag[i] / d_matrix_array[i - 1]

      d_matrix_array[i] = a_matrix_diag[i] - d_matrix_array[i - 1] * ((l_matrix_array[i])**2)

  return l_matrix_array, d_matrix_array
```

A função “matrix_decomposition” foi criada com o intuito de receber a matrix A tridiagonal simétrica e a decompor em três matrizes L, D e Lt uma vez que $A = LDL^t$. Uma vez decomposta a matriz, a função deve retornar dois vetores que representam essa matrizes (“L” e “D”), não é preciso retornar o vetor Lt, pois trata-se apenas da matriz transposta de L.

Então para se resolver um sistema $Ax = LDL^tx = b$, foi feita a seguinte função:

```
def solve_system(a_matrix_diag, a_matrix_subdiag, b_array):
    """
    Soluciona um sistema Ax = b, onde A é uma matrix tridiagonal
    simétrica.

    Para a resolução do sistema, é feita a decomposição de A para L*D*Lt.

    É feita a divisão do problema em três sistemas menores:
    L * y = b
    D * z = y
    Lt * x = z
    """

    array_size = len(a_matrix_diag)

    l_matrix_array, d_matrix_array = matrix_decomposition(a_matrix_diag, a_matrix_subdiag)

    # First system solution -> L * y = b
    y_array = np.zeros(array_size, dtype=float)

    y_array[0] = b_array[0]

    for i in range(1, array_size):
        y_array[i] = b_array[i] - l_matrix_array[i] * y_array[i - 1]

    # Second system solution -> D * z = y
    z_array = np.zeros(array_size, dtype=float)

    for i in range(0, array_size):
```

```
    z_array[i] = y_array[i] / d_matrix_array[i]

# Third system solution -> Lt * x = z
x_array = np.zeros(array_size, dtype=float)

x_array[-1] = z_array[-1]

for i in reversed(range(0, array_size - 1)):
    x_array[i] = z_array[i] - l_matrix_array[i + 1] * x_array[i + 1]

return x_array
```

Essa função “solve_system” recebe a matriz A tridiagonal simétrica e um vetor b para achar a solução da equação $Ax = b$ e através da decomposição da matriz A tridiagonal simétrica por meio da função “matrix_decomposition” implementada acima, resolver o problema, quebrando-o em três sistemas menores para facilitar a resolução:

$$L \times y = b$$
$$D \times z = y$$
$$Lt \times x = z$$

Para então retornar o valor do vetor x que representa a solução do problema .

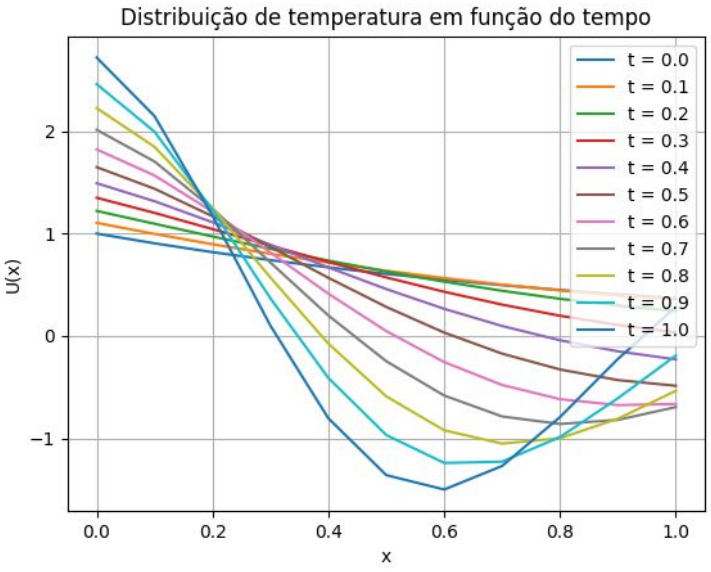
Item b)

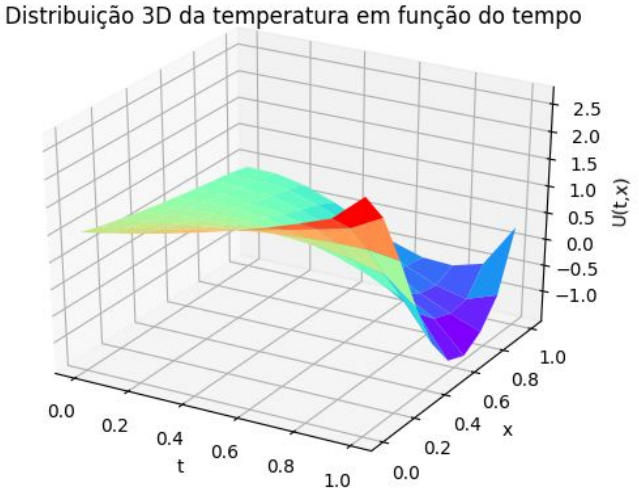
Repetindo os mesmos testes da primeira tarefa com o método de Euler implícito, e utilizando $\Delta t = \Delta x$.

Para demonstração do resultado foi utilizada a fonte do item b da primeira parte.

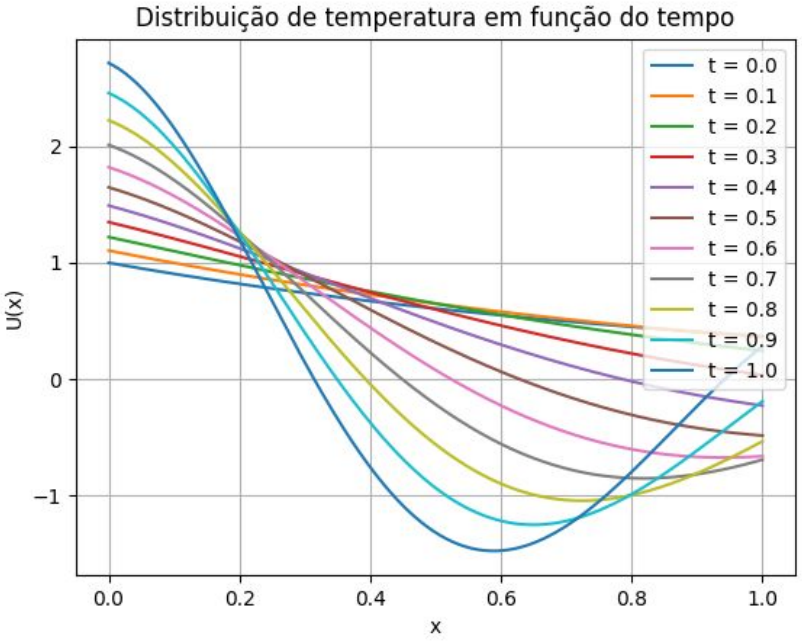
Obtemos assim, os seguintes resultados:

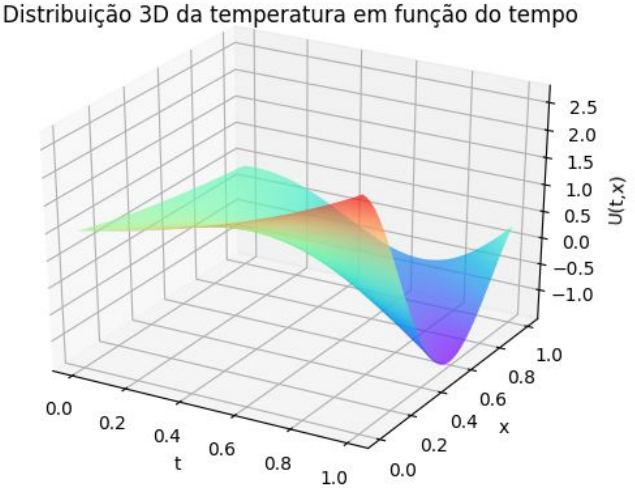
N = 10


Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
x
2
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0


Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

N = 320

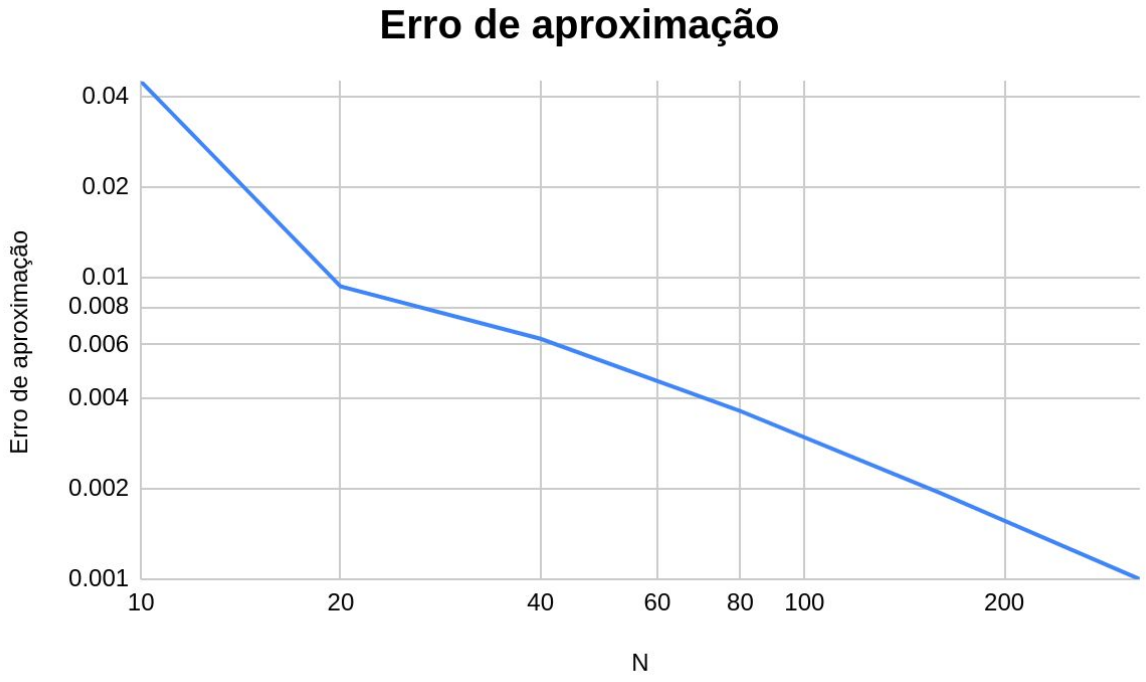

Distribuição de temperatura em função do tempo
t = 0.0
t = 0.1
t = 0.2
t = 0.3
t = 0.4
t = 0.5
t = 0.6
t = 0.7
t = 0.8
t = 0.9
t = 1.0
U(x)
x
2
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0

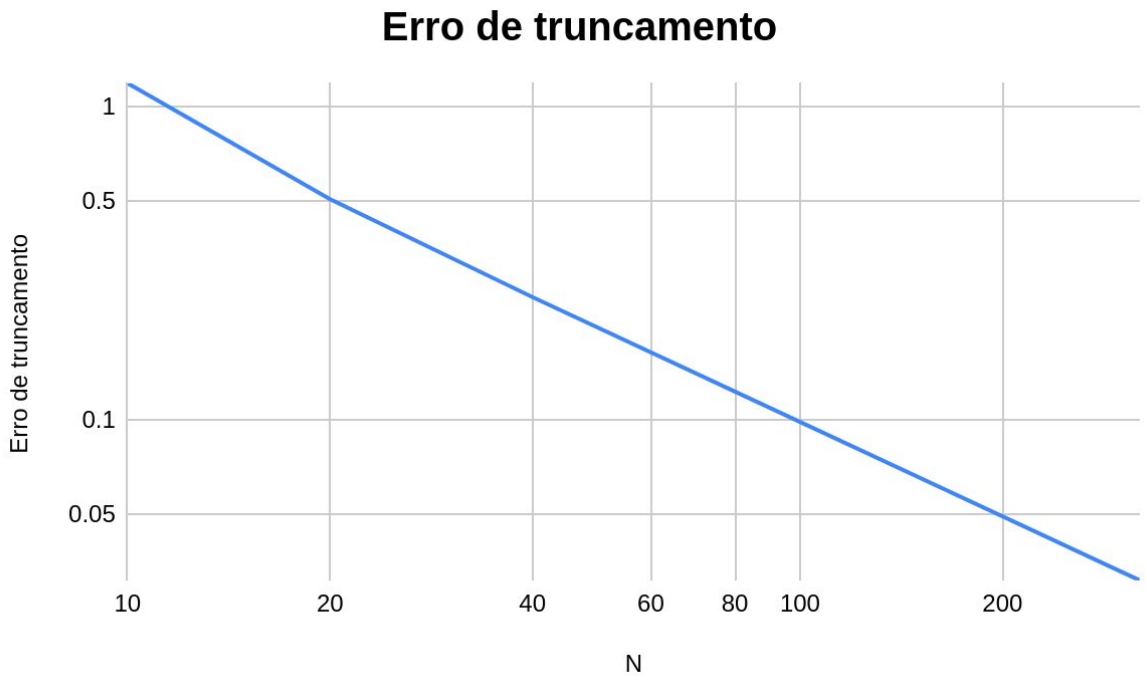

Distribuição 3D da temperatura em função do tempo
U(t,x)
t
x
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

Foram obtidos os seguintes valores de erros para diferentes valores de N para o método de Euler implícito:

N	Erro truncamento	Erro máximo de aproximação
10	1.1898730473488968	0.04496773139525201
20	0.5084911324580745	0.009371453935905216
40	0.24749051547502798	0.006275526446844593
80	0.1237251194276765	0.0036093258258742544
160	0.06185224218035401	0.0019292318120289753
320	0.030923826435108825	0.00099675666629917

Obtendo-se os seguintes gráficos em escala logarítmica:





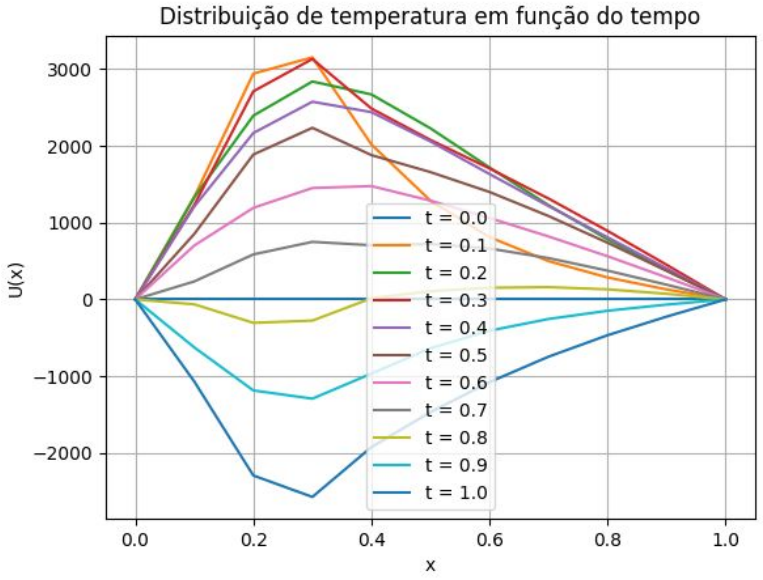
Podemos observar assim que ao se dobrar o valor de N, o erro cai pela metade, provando que a ordem de convergência do método é de 2.

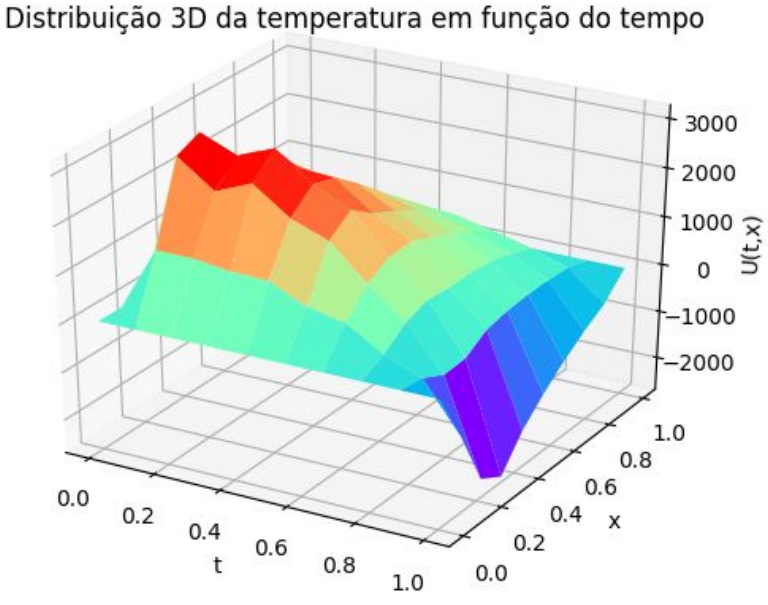
A partir da implementação do método de Euler implícito verificamos que a velocidade de execução do programa se torna muito mais rápida quando comparada ao Condicionamento Convergente. Porém o erro verificado para aproximação e truncamento é relativamente maior do que o método de Condicionamento Convergente, para um mesmo N = 320. No método de Condicionamento Convergente com λ = 0.25 temos um erro de aproximação igual a 2.955251310773205e-06 e um erro de Truncamento igual a 4.6691906011808726e-05 , já para o método de Euler com o mesmo N temos um erro de aproximação igual a 0.00099675666629917 e um erro de truncamento igual a 0.030923826435108825, ou seja um erro de truncamento cerca de 662,2952258 vezes maior e um erro de aproximação cerca de 337,283216038 vezes maior quando comparados.

Item c)

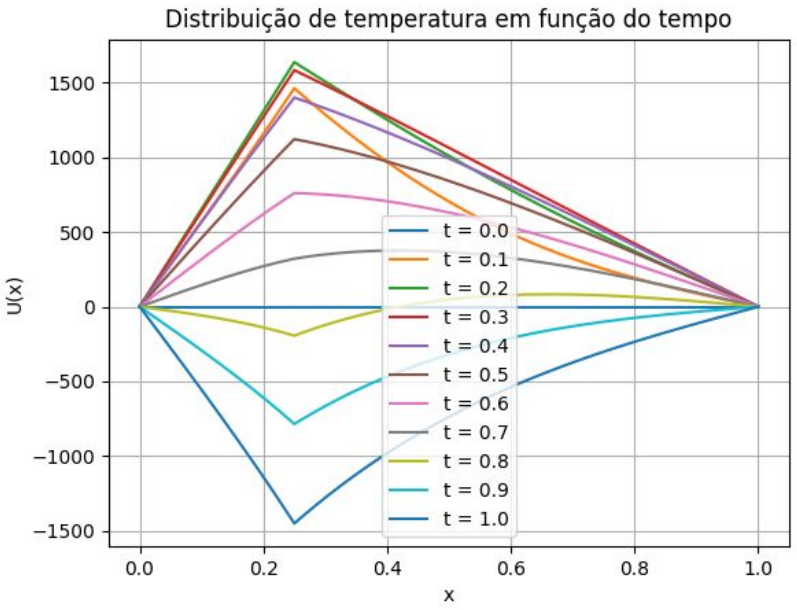
Repetindo os mesmos testes com o método de Crank-Nicolson e utilizando a fonte de calor do item b da primeira parte para demonstração, obtemos:

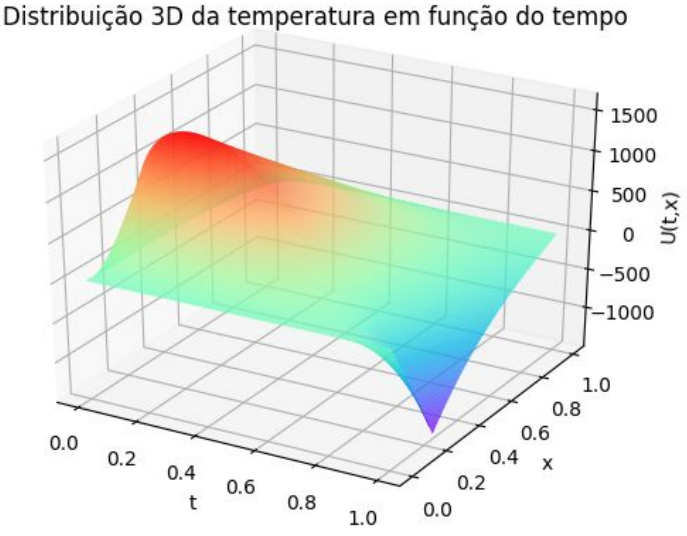
N = 10





N = 320

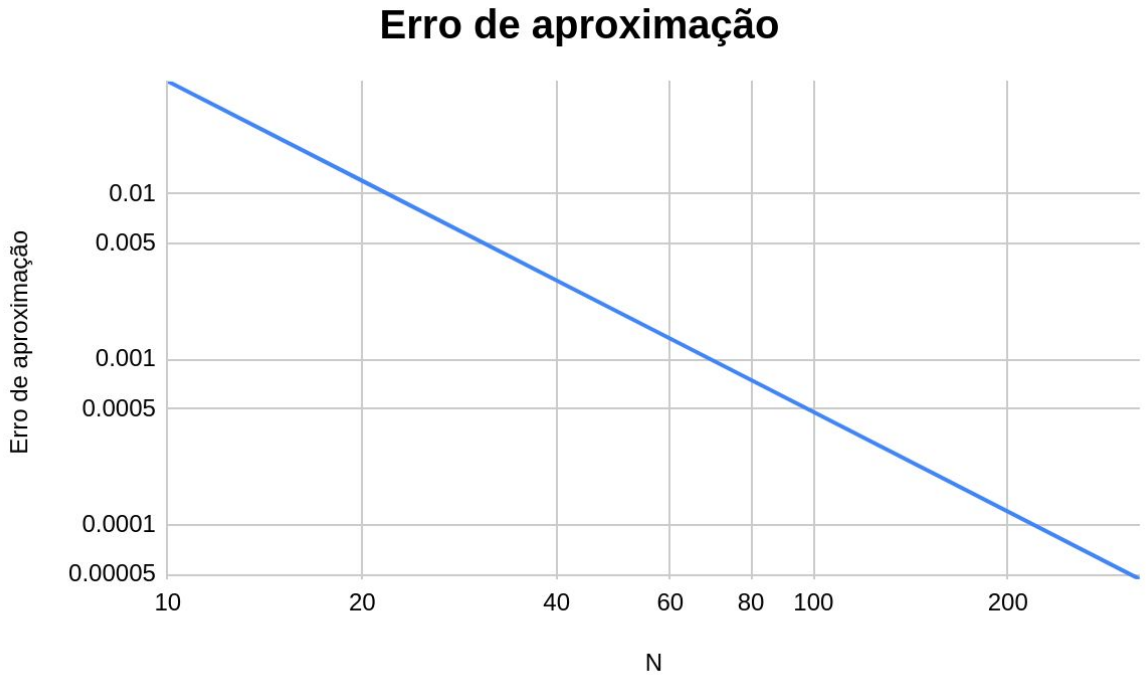




Além disso calculando o erro de aproximação para a solução utilizando o método de Crank-Nicolson, obtemos:

N	Erro de aproximação
10	0.0477794380800296
20	0.011990054057935406
40	0.0029926797918580217
80	0.0007487551993363706
160	0.00018717060447448475
320	4.679151404185511e-05

Com esses valores de erros obtemos o seguinte gráfico para o erro:



Conclusão

Após a implementação e observação do comportamento dos erros de truncamento, erros de aproximações e velocidade de execução dos métodos de Condicionamento Convergente, método de Euler e método de Crank-Nicolson, conseguimos chegar a conclusão que a partir de um mesmo N = 320, temos que, apesar do Método de Condicionamento Convergente possuir o menor erro de aproximação na ordem de 2.955251310773205e-06 (considerando o $\lambda = 0.25$), esse método foi o que mais demorou para rodar, já o método de Euler possui uma velocidade de execução muito maior, contudo o seu erro de aproximação foi muito maior comparativamente com o erro de aproximação do método Condicionamento Convergente sendo o erro na ordem de 0.00099675666629917.

Por fim, para o método de Crank-Nicolson temos um tempo de execução similar com o método de Euler, porém com um erro de aproximação menor parecido com o erro observado pelo método de Condicionamento Convergente dado por aproximadamente 6.06902862796202e-06. Sendo que o método de Euler possui um erro de aproximação 337,2832 maior que o método Condicionamento Convergente, o método de Crank um erro 2,053642141 maior que o Condicionamento Convergente e o método de Euler 164,2366 maior.

Portanto, para os métodos implícitos, por serem mais rápidos, pode se rodá-los para valores de N maiores, obtendo-se valores de erros menores.

Comparação do o erro de aproximação entre os métodos:

Euler > Crank-nicolson > Condicionamento Convergente

Velocidade de execução:

Crank-nicolson = Euler > Condicionamento Convergente