

UNIVERSIDADE DE SÃO PAULO - USP
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA - IME
DEPARTAMENTO DE MATEMÁTICA APLICADA - MAP
Bacharelado em Matemática Aplicada e Computacional - BMAC
Métodos Numéricos em Equações Diferenciais II - MAP2320

Projeto Final

Jonatan de Sena Silva
Mateus Jingi Chou
Nathanaell Vinícius de Camargos Welter

nUSP: 6820402
nUSP: 11221352
nUSP: 10809021

Professor: Dr. Luis Carlos de Castro Santos

São Paulo
2022

Sumário

1	Introdução	2
2	Questão 1	3
2.1	Formulação Matemática	3
2.2	Método das Diferenças Finitas - MDF	3
2.2.1	<i>Forward in Time, Backward in Space - FTBS</i>	3
2.2.2	<i>Leapfrog</i>	4
3	Resultados e Discussões da Questão 01	5
3.1	Resultados para <i>FTBS</i> com $\lambda = 1$	5
3.2	Resultados para <i>FTBS</i> com $\lambda < 1$	9
3.3	Resultados para <i>Leapfrog</i> com $\lambda = 1$	13
3.4	Resultados para <i>Leapfrog</i> com $\lambda < 1$	17
3.5	Discussões da Questão 01	21
4	Questão 2	22
4.1	Formulação Matemática	22
4.2	Método das Diferenças Finitas - MDF	22
5	Resultados e Discussões da Questão 02	25
5.1	Resultados para <i>Crank-Nicolson</i>	25
5.2	Discussões da Questão 02	31
6	Conclusão	33
	Referências	34

1 Introdução

A resolução de equações diferenciais tem um papel fundamental em diversas áreas do conhecimento, dentre as quais podemos citar, por exemplo, a física, a química, as engenharia, a astronomia, a biologia e até mesmo as ciências sociais.

Por outro lado, ainda que a resolução de equações diferenciais tenha um papel fundamental em diversas áreas do conhecimento, com frequência, não é possível encontrar uma solução analítica explícita para uma EDO (Equação Diferencial Ordinária) ou uma EDP (Equação Diferencial Parcial) e, portanto, uma nova abordagem de resolução é necessária. Graças ao avanço da matemática e da tecnologia, esses problemas, que antes pareciam in-tratáveis, hoje vêm sendo resolvidos através de computadores, com os chamados Métodos Numéricos.

Um desses métodos é o chamado Método das Diferenças Finitas (MDF) que, segundo [4], leva à transformação das equações diferenciais em um conjunto de equações algébricas que podem ser, então, resolvidas.

Segundo o autor [4], esse método consiste em dividir o domínio de solução em um conjunto de pequenos elementos discretos, sendo para cada elemento atribuído uma equação algébrica que aproxima a derivada por diferenças finitas obtidas, segundo [3], via série de Taylor da função derivada.

Entretanto, é fundamental conseguir definir quando um sistema de equações algébricas lineares possui alguma solução (ou seja, quando é consistente) e se essa solução é única ou envolve um certo número de parâmetros arbitrários.

Desse forma, nesse trabalho, avaliaremos a resolução de duas equações diferenciais parciais (EDP). A primeira do tipo hiperbólica que será resolvida através de dois esquemas numéricos diferentes, o primeiro chamado de *Forward in Time, Back in Space* e o segundo de *Leapfrog*; e a segunda, do tipo parabólica, será resolvida via Método de *Crank-Nicolson*.

Inicialmente são apresentados os modelos matemáticos de ambas as EDP (Seções 2.1 e 4.1). Na sequência, a formulação matemática utilizada na discretização de cada uma das EDPs mencionadas (Seções 2.2 e 4.2). Logo em seguida esses problema são resolvidos através de um programa desenvolvido em *Python 3* e, finalmente, nas Seções 3 e 5 os resultados são apresentados e discutidos. Os algoritmos relativos aos métodos aqui estudados são apresentados nos Apêndices - A, B e C.

2 Questão 1

Nesta seção apresentamos e discutimos a equação da onda de primeira ordem cuja formulação matemática encontra-se na Seção 2.1.

Logo em seguida, na Seção 2.2, são apresentados dois esquemas numéricos para resolução das equações via MDF.

2.1 Formulação Matemática

Esta seção trata da formulação matemática da Equação Hiperbólica de uma onda unidimensional.

Para esta são consideradas condições de contorno periódicas e domínio contido no intervalo $-1 \leq x \leq 1$.

Em se tratando das condições iniciais, o problema proposto pede para que seja considerado uma equação do tipo senoidal e de período igual a $T = \frac{2\pi}{x}$.

A equação que rege o comportamento dessa onda é apresentada logo abaixo na Expressão 1.

$$\begin{aligned} u_t + au_x &= 0 & a > 0 & -1 \leq x \leq 1 \\ \text{C. C. Periódicas} \\ u_0 &= u_0(0, x) = \sin(2\pi x) \end{aligned} \tag{1}$$

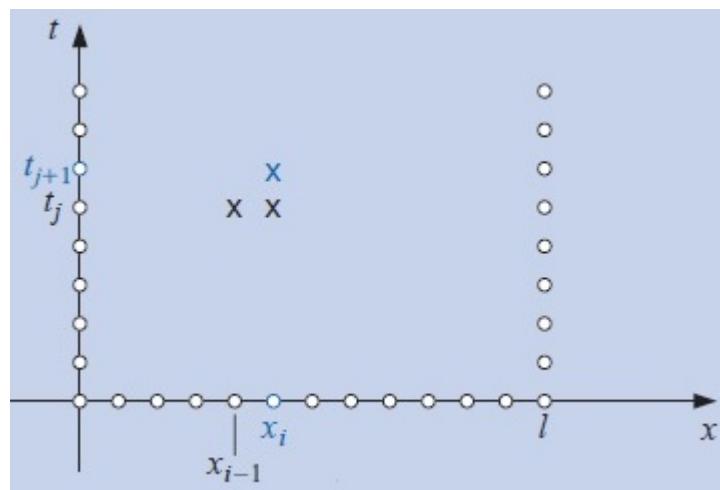
2.2 Método das Diferenças Finitas - MDF

Nesta seção são apresentados os dois esquemas de cálculo utilizados na resolução da Equação da Onda (Sistema de Equações 1). O primeiro chamado de *Forward in Time, Backward in Space* (Seção 2.2.1) e o segundo chamado de *Leapfrog* (Seção 2.2.2).

2.2.1 *Forward in Time, Backward in Space - FTBS*

O primeiro esquema de cálculo é chamado de *Forward in Time, Backward in Space - FTBS* e seu estêncil é apresentado logo abaixo, na Figura 1.

Figura 1: Estêncil de *Forward in Time, Backward in Space - FTBS*



Fonte: Adaptado de [1]

Para esse esquema de pontos (estêncil), a equação utilizada na discretização é a Equação 4:

$$\frac{v_m^{n+1} - v_m^n}{k} + a \frac{v_m^n - v_{m-1}^n}{h} = 0 \quad (2)$$

$$v_m^{n+1} = -ak \frac{v_m^n - v_{m-1}^n}{h} + v_m^n \quad (3)$$

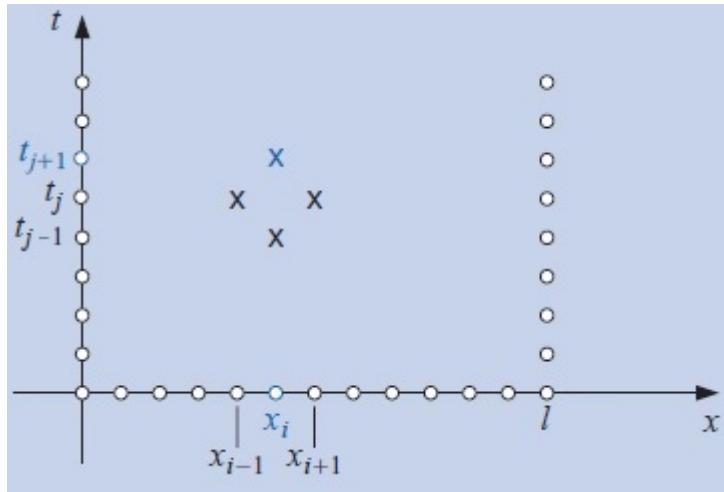
$$v_m^{n+1} = -\lambda(v_m^n - v_{m-1}^n) + v_m^n \quad (4)$$

em que $\lambda = \frac{ak}{h}$.

2.2.2 Leapfrog

O segundo esquema de cálculo é chamado de *Leapfrog* e seu estêncil é apresentado logo abaixo, na Figura 2.

Figura 2: Estêncil de *Leapfrog*



Fonte: Adaptado de [1]

Para esse esquema de pontos (estêncil), a equação utilizada na discretização é a Equação 7:

$$\frac{v_m^{n+1} - v_m^{n-1}}{2k} + a \frac{v_{m+1}^n - v_{m-1}^n}{2h} = 0 \quad (5)$$

$$v_m^{n+1} = -2ka \frac{v_{m+1}^n - v_{m-1}^n}{2h} + v_m^{n-1} \quad (6)$$

$$v_m^{n+1} = -\lambda(v_{m+1}^n - v_{m-1}^n) + v_m^{n-1} \quad (7)$$

onde $\lambda = \frac{ak}{h}$.

3 Resultados e Discussões da Questão 01

Nesta seção apresentamos as saídas da implementação computacional dos dois métodos elencados acima.

Em cada configuração de *CFL* (condição de convergência de *Courant–Friedrichs–Lewy*, aqui denotado por λ) selecionada, mostramos figuras que descrevem a evolução temporal da onda, a sobreposição das ondas ao longo do tempo e o perfil das ondas para diferentes refinamentos das partições do domínio. Juntamente com essas figuras, apresentamos uma tabela para visualização dos termos de erro encontrados, bem como da ordem alcançada pelo método. Algumas figuras animadas serão anexadas à pasta do trabalho por não serem compatíveis com o formato “.pdf”.

3.1 Resultados para *FTBS* com $\lambda = 1$

A seguir, na Tabela 1, apresentamos alguns dos principais resultados, obtidos para o método *Forward in Time, Backward in Space (FTBS)* para $\lambda = 1$.

Tabela 1: *Forward in Time, Backward in Space (FTBS), $\lambda = 1$*

m	n	h	k	λ	Erro	Tempo
32	32	0.0625	0.0625	1.0	1.7208e-15	1.9991e-03
64	64	0.03125	0.03125	1.0	1.8874e-15	2.9993e-03
128	128	0.015625	0.015625	1.0	1.9984e-15	3.9978e-03
256	256	0.0078125	0.0078125	1.0	2.2621e-15	7.9956e-03
512	512	0.00390625	0.00390625	1.0	2.2621e-15	1.9990e-02
1024	1024	0.001953125	0.001953125	1.0	2.2621e-15	4.6973e-02

Fonte: Os Autores, 2022

As Figuras 3 - 5 representam graficamente alguns resultados notáveis obtidos para esse método considerando $\lambda = 1$.

Figura 3: *FTBS* - Ondas Notáveis, $\lambda = 1$

Forward in Time, Backward in Space (FTBS) - Ondas Notáveis

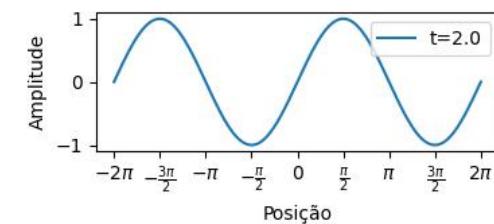
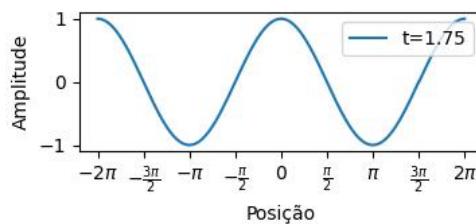
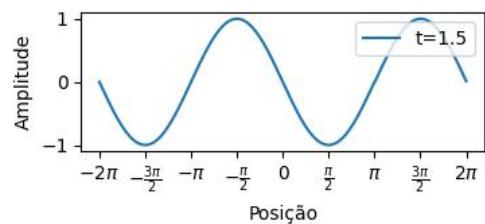
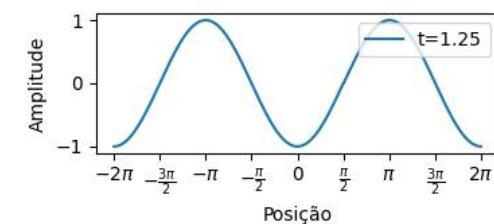
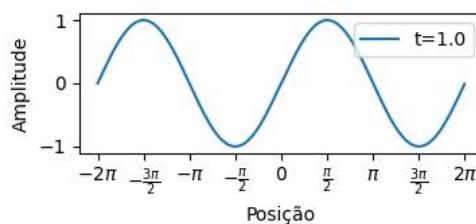
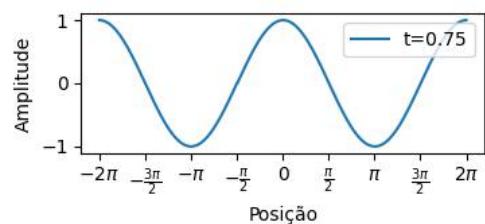
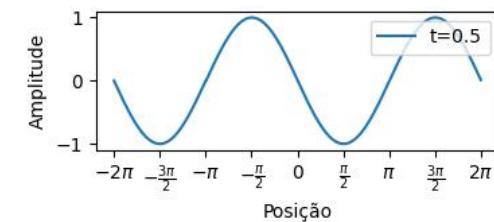
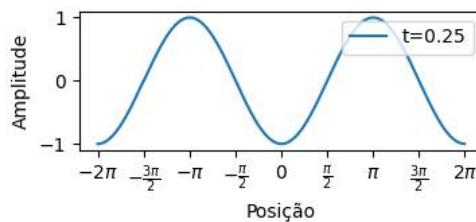
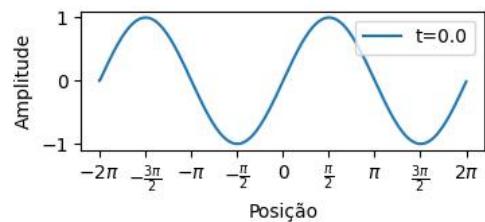
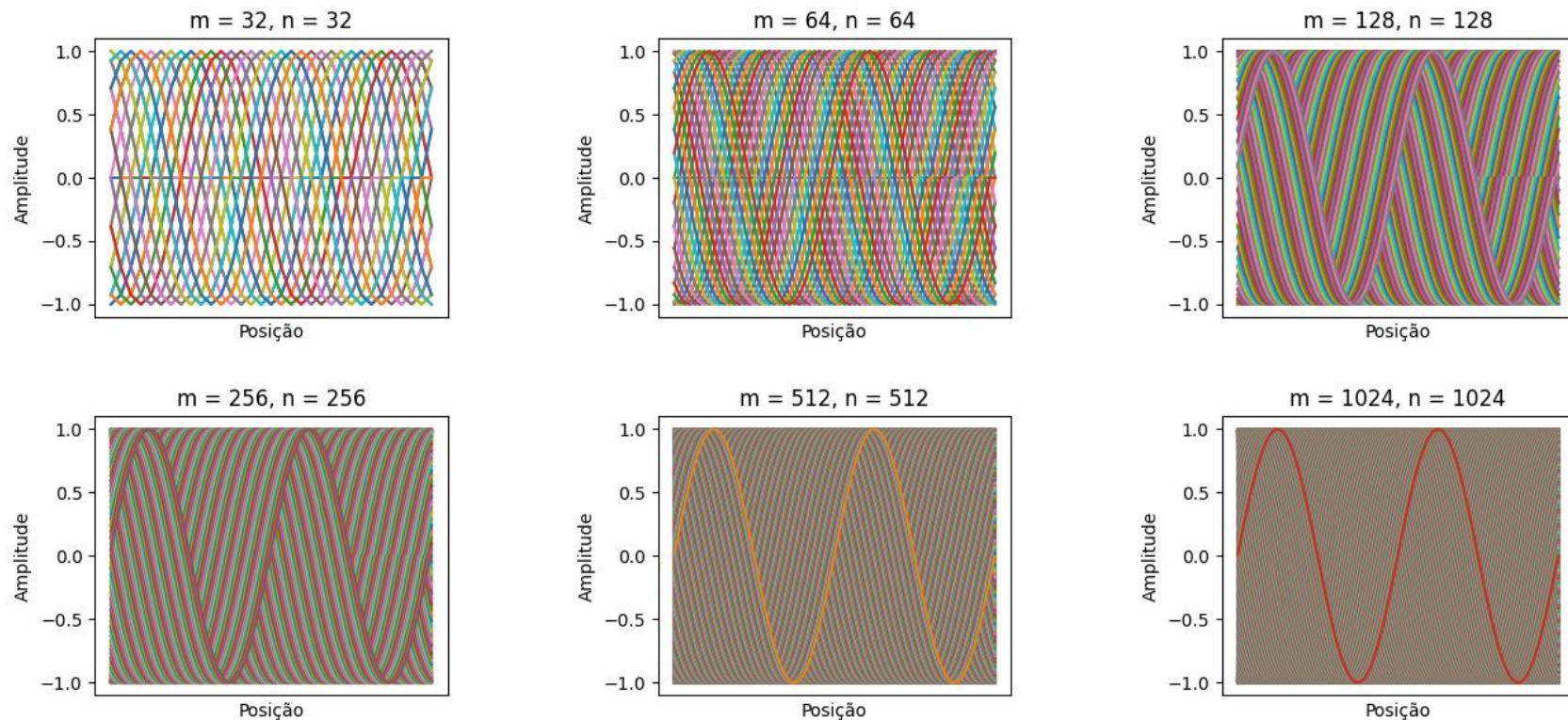


Figura 4: *FTBS* - Ondas Sobrepostas, $\lambda = 1$

Forward in Time, Backward in Space (FTBS) - Sobrepostas

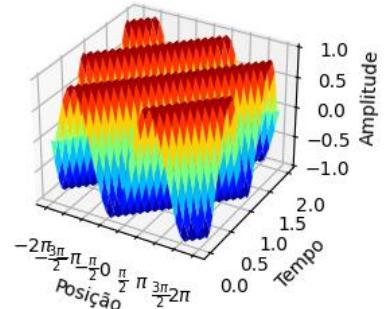


Fonte: Os Autores, 2022

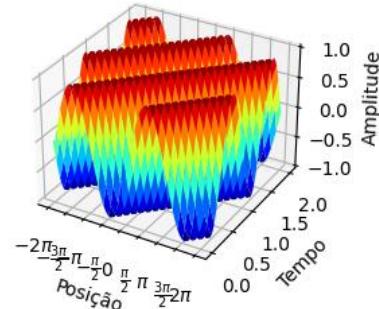
Figura 5: FTBS - Ondas 3D, $\lambda = 1$

Forward in Time, Backward in Space (FTBS) - Ondas 3D

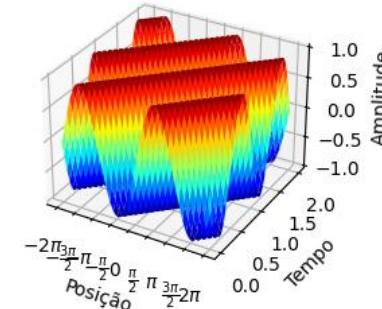
$m = 32, n = 32$



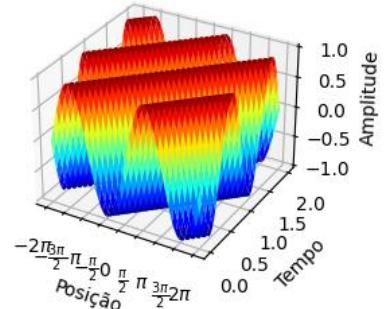
$m = 64, n = 64$



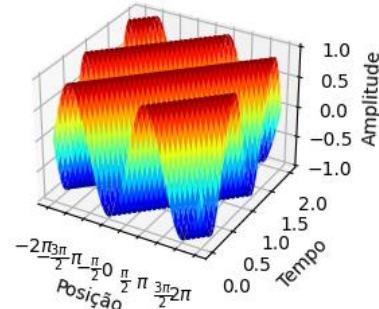
$m = 128, n = 128$



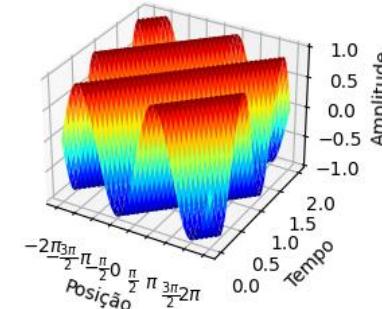
$m = 256, n = 256$



$m = 512, n = 512$



$m = 1024, n = 1024$



∞

Fonte: Os Autores, 2022

3.2 Resultados para *FTBS* com $\lambda < 1$

A seguir, na Tabela 2, apresentamos alguns dos principais resultados, obtidos para o método *Forward in Time, Backward in Space (FTBS)* para diferentes valores de $\lambda < 1$.

Tabela 2: *Forward in Time, Backward in Space (FTBS), $\lambda < 1$*

m	n	h	k	λ	Erro	Tempo	Ordem
32	40	0.0625	0.05	0.8	3.9727e-01	9.8825e-04	0.0000e+00
64	80	0.03125	0.025	0.8	2.2159e-01	2.0161e-03	8.4219e-01
128	160	0.015625	0.0125	0.8	1.1780e-01	3.9823e-03	9.1161e-01
256	320	0.0078125	0.00625	0.8	6.0662e-02	9.9945e-03	9.5745e-01
512	640	0.00390625	0.003125	0.8	3.0777e-02	1.9004e-02	9.7895e-01
1024	1280	0.001953125	0.0015625	0.8	1.5501e-02	4.5974e-02	9.8947e-01
32	64	0.0625	0.03125	0.5	7.2254e-01	1.9984e-03	0.0000e+00
64	128	0.03125	0.015625	0.5	4.6738e-01	2.9986e-03	6.2847e-01
128	256	0.015625	0.0078125	0.5	2.6950e-01	6.9950e-03	7.9433e-01
256	512	0.0078125	0.00390625	0.5	1.4497e-01	1.3992e-02	8.9450e-01
512	1024	0.00390625	0.001953125	0.5	7.5212e-02	3.3983e-02	9.4675e-01
1024	2048	0.001953125	0.0009765625	0.5	3.8313e-02	7.2997e-02	9.7313e-01
32	80	0.0625	0.025	0.4	7.8777e-01	5.9998e-03	0.0000e+00
64	160	0.03125	0.0125	0.4	5.3116e-01	5.0111e-03	5.6862e-01
128	320	0.015625	0.00625	0.4	3.1417e-01	7.9772e-03	7.5760e-01
256	640	0.0078125	0.003125	0.4	1.7140e-01	2.0988e-02	8.7420e-01
512	1280	0.00390625	0.0015625	0.4	8.9576e-02	4.5993e-02	9.3617e-01
1024	2560	0.001953125	0.00078125	0.4	4.5801e-02	1.1092e-01	9.6772e-01
32	160	0.0625	0.0125	0.2	8.7993e-01	7.0152e-03	0.0000e+00
64	320	0.03125	0.00625	0.2	6.3750e-01	8.9779e-03	4.6496e-01
128	640	0.015625	0.003125	0.2	3.9572e-01	1.8987e-02	6.8795e-01
256	1280	0.0078125	0.0015625	0.2	2.2189e-01	3.4980e-02	8.3465e-01
512	2560	0.00390625	0.00078125	0.2	1.1765e-01	7.5974e-02	9.1528e-01
1024	5120	0.001953125	0.000390625	0.2	6.0608e-02	1.8388e-01	9.5697e-01

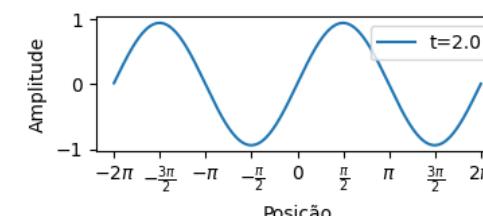
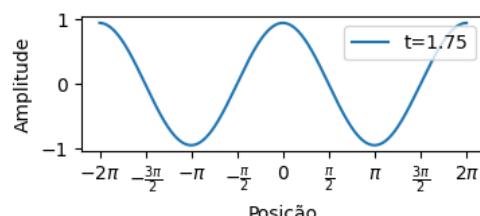
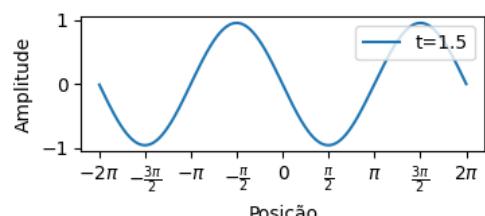
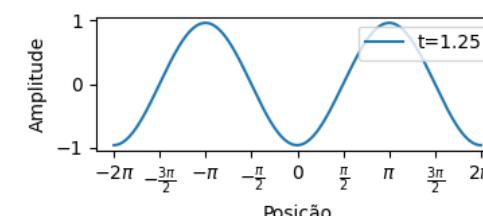
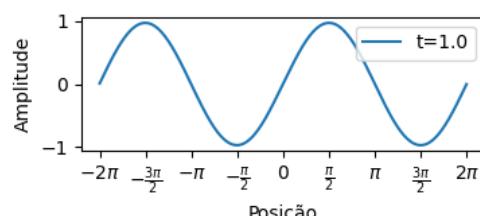
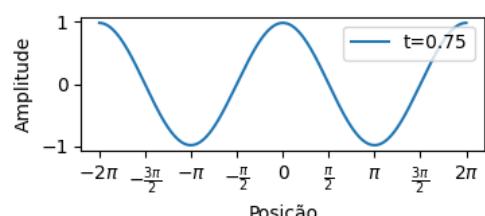
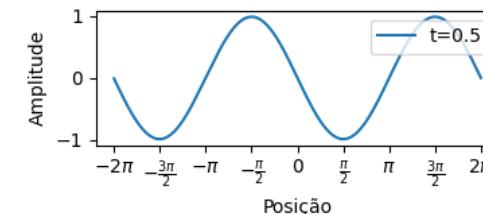
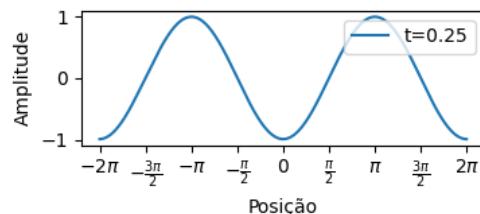
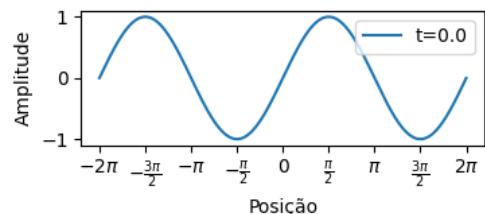
Fonte: Os Autores, 2022

As Figuras 6 - 8 representam graficamente alguns resultados notáveis obtidos para esse método considerando $\lambda = 0.2$.

Vale ressaltar que por se tratarem de resultados, visualmente, bastante similares, demais gráficos foram suprimidos. No entanto, os mesmos podem ser obtidos mediante execução do algoritmo apresentado no Apêndice A ou no programa anexo.

Figura 6: *FTBS* - Ondas Notáveis, $\lambda = 0.2$

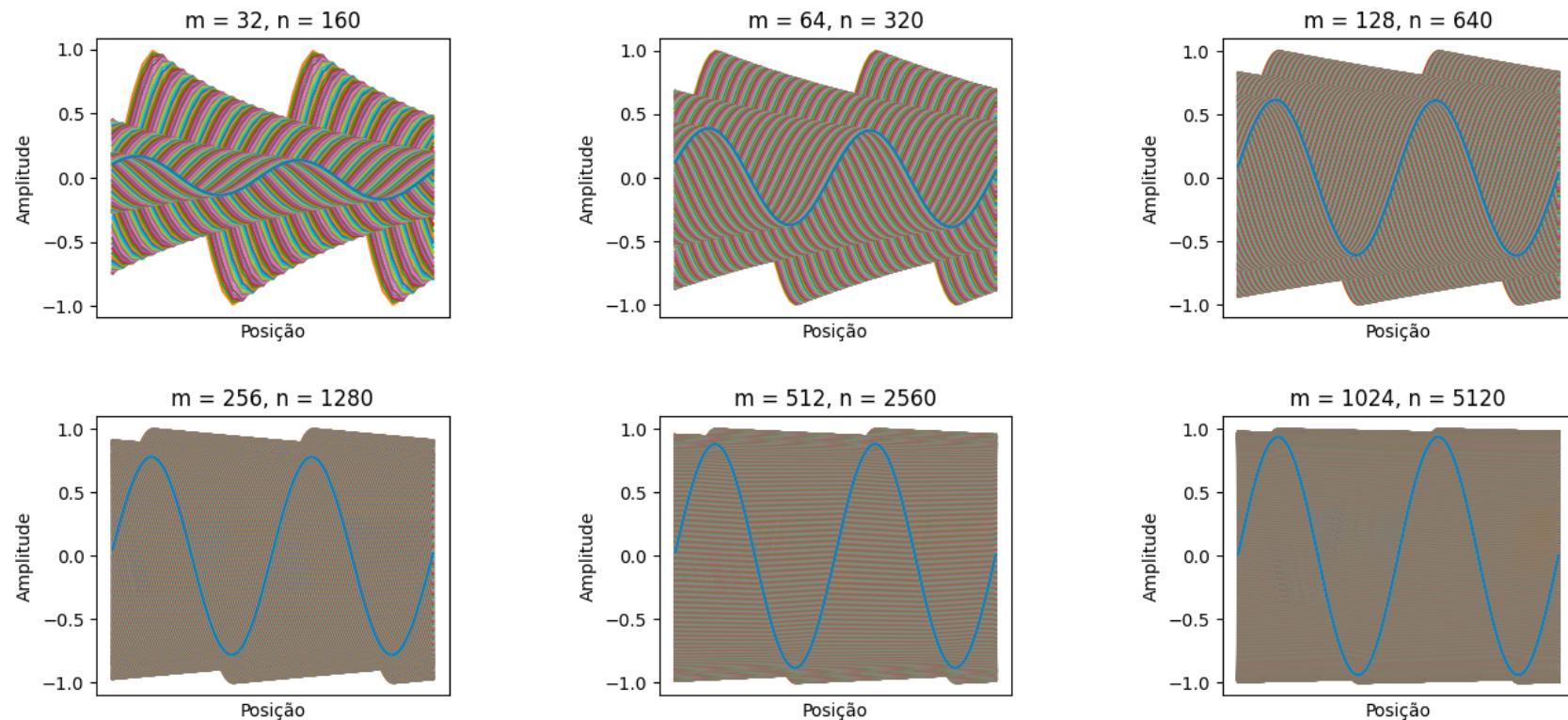
Forward in Time, Backward in Space (FTBS) - Ondas Notáveis



Fonte: Os Autores, 2022

Figura 7: *FTBS* - Ondas Sobrepostas, $\lambda = 0.2$

Forward in Time, Backward in Space (FTBS) - Sobrepostas

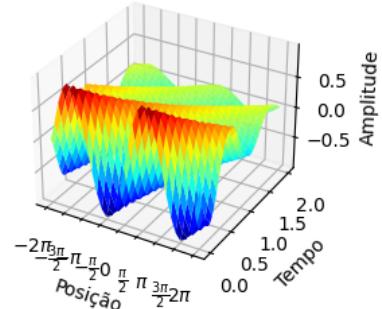


Fonte: Os Autores, 2022

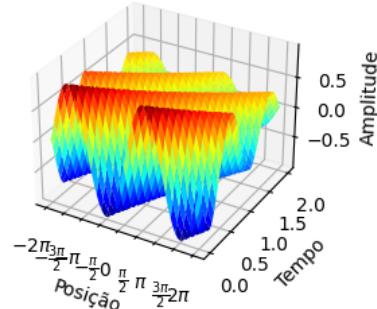
Figura 8: *FTBS* - Ondas 3D, $\lambda = 0.2$

Forward in Time, Backward in Space (FTBS) - Ondas 3D

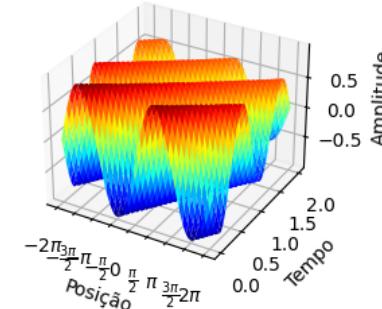
$m = 32, n = 160$



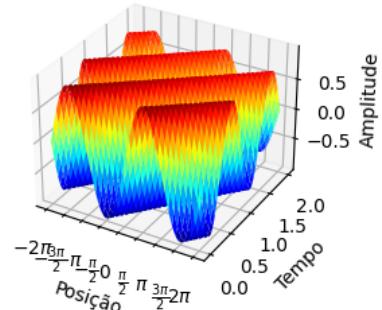
$m = 64, n = 320$



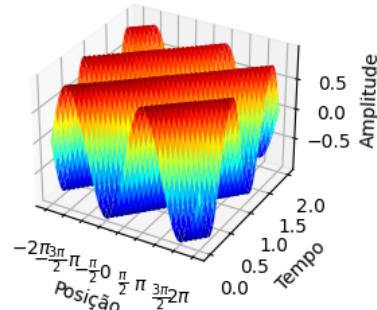
$m = 128, n = 640$



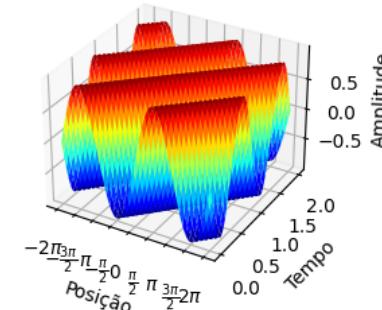
$m = 256, n = 1280$



$m = 512, n = 2560$



$m = 1024, n = 5120$



3.3 Resultados para *Leapfrog* com $\lambda = 1$

A seguir, na Tabela 3, apresentamos alguns dos principais resultados, obtidos para o método *Leapfrog* para $\lambda = 1$.

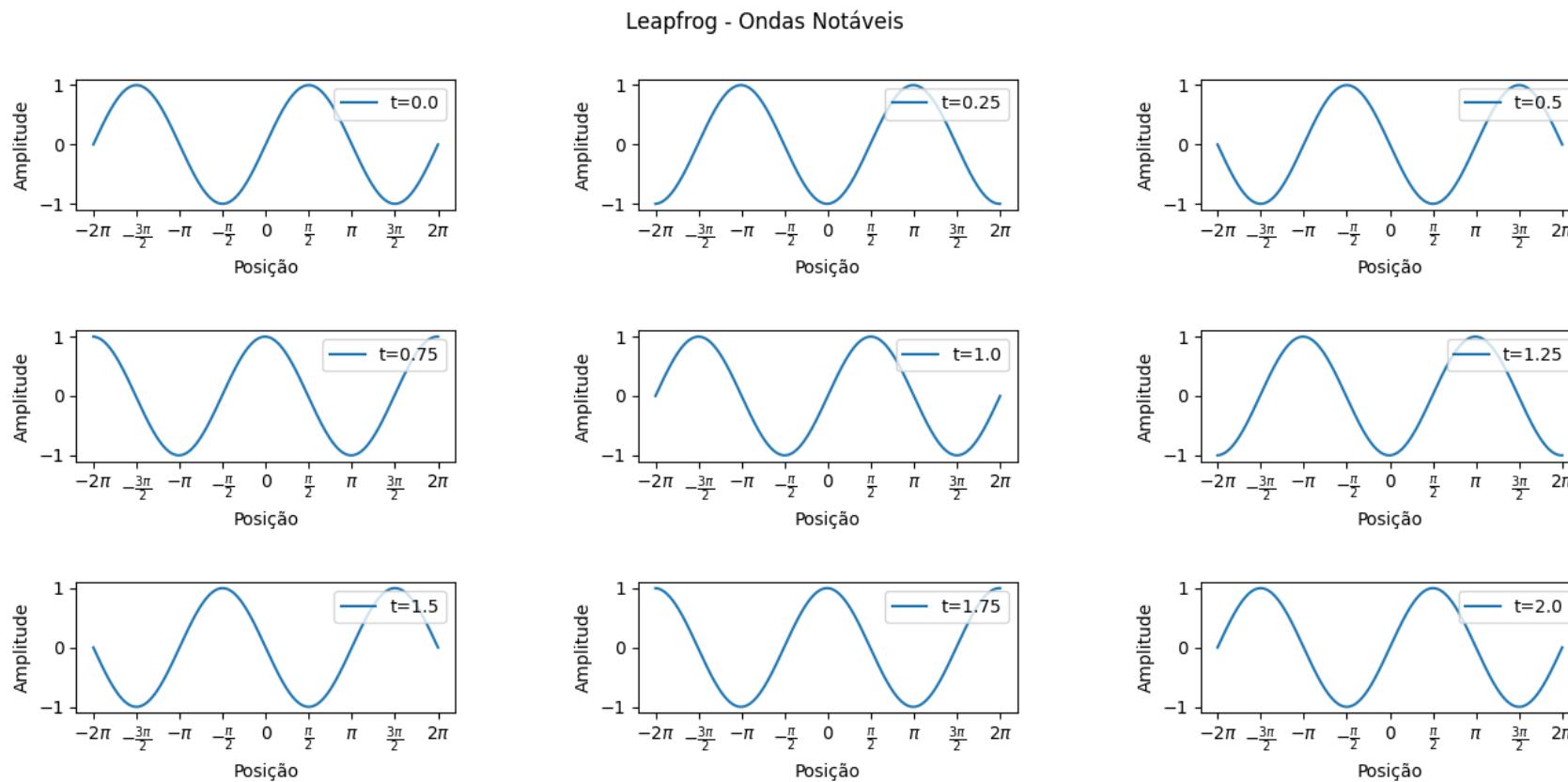
Tabela 3: *Leapfrog*, $\lambda = 1$

m	n	h	k	λ	Erro	Tempo
32	32	0.0625	0.0625	1.0	3.5527e-15	9.9969e-04
64	64	0.03125	0.03125	1.0	5.8842e-15	1.0006e-03
128	128	0.015625	0.015625	1.0	9.8810e-15	9.9707e-04
256	256	0.0078125	0.0078125	1.0	1.1581e-14	3.9964e-03
512	512	0.00390625	0.00390625	1.0	2.1892e-14	8.9946e-03
1024	1024	0.001953125	0.001953125	1.0	5.0404e-14	2.6985e-02

Fonte: Os Autores, 2022

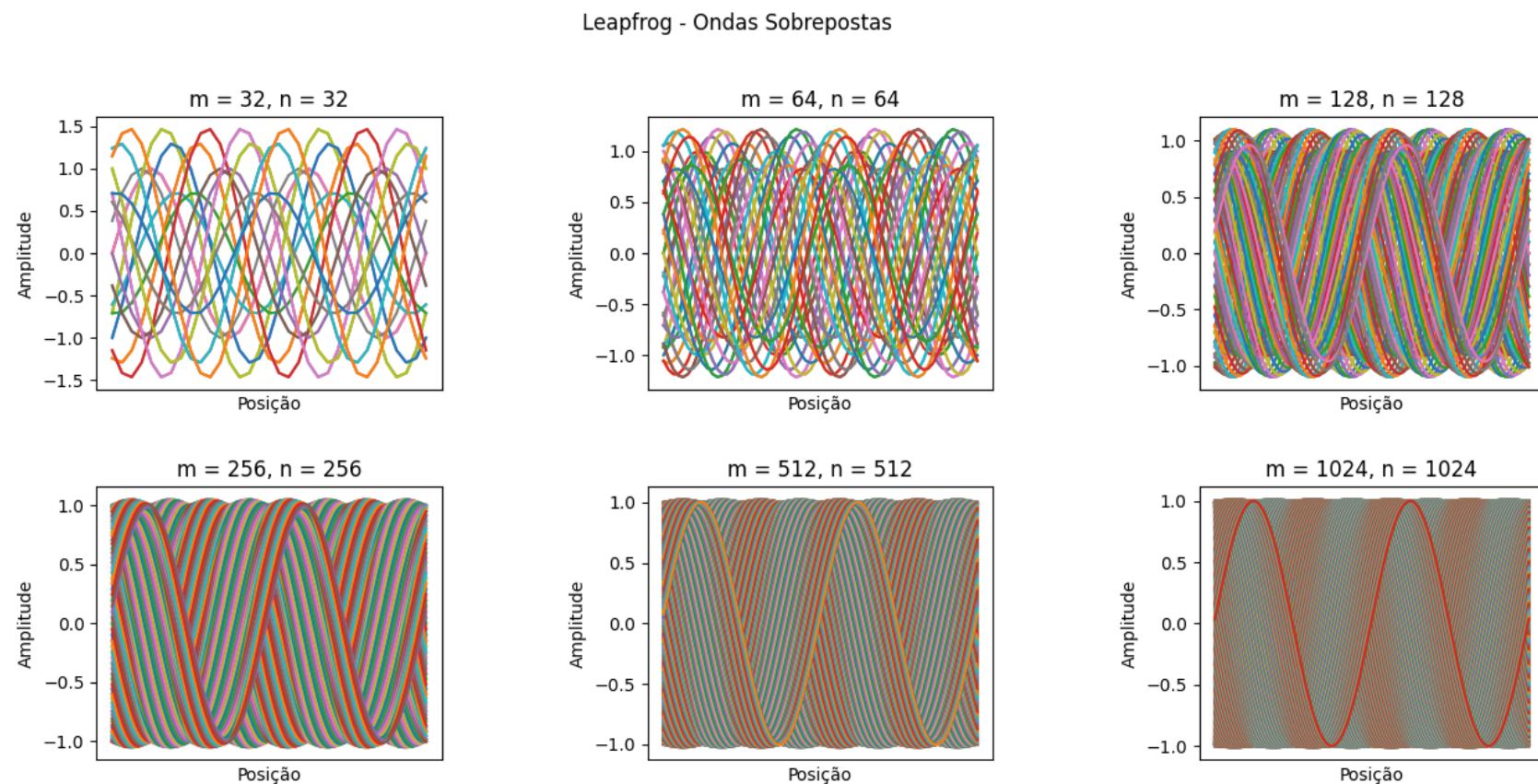
As Figuras 9 - 11 representam graficamente alguns resultados notáveis obtidos para esse método considerando $\lambda = 1$.

Figura 9: *Leapfrog - Ondas Notáveis*, $\lambda = 1$



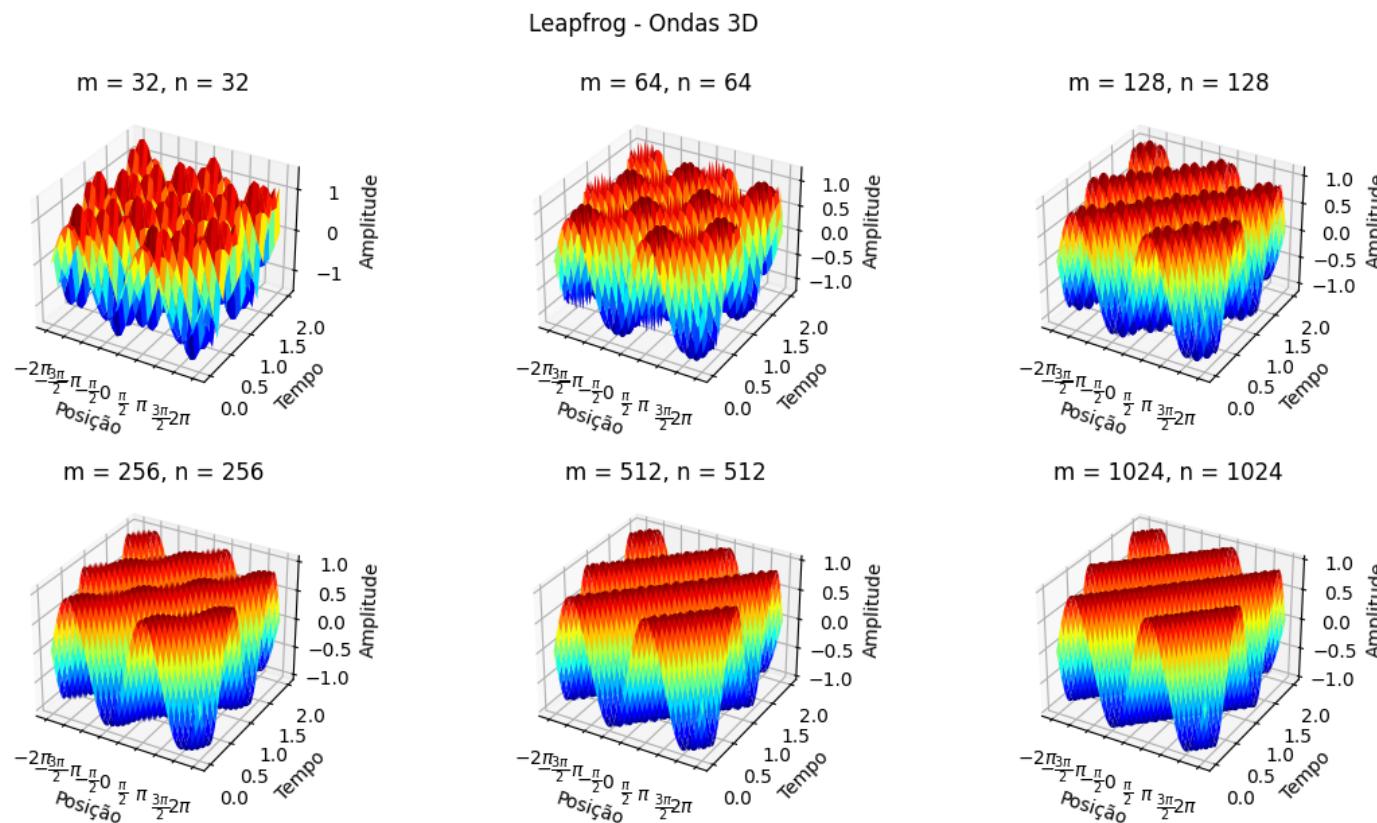
Fonte: Os Autores, 2022

Figura 10: *Leapfrog* - Ondas Sobrepostas, $\lambda = 1$



Fonte: Os Autores, 2022

Figura 11: *Leapfrog - Ondas 3D, $\lambda = 1$*



Fonte: Os Autores, 2022

3.4 Resultados para *Leapfrog* com $\lambda < 1$

A seguir, na Tabela 4, apresentamos alguns dos principais resultados, obtidos para o método *Leapfrog* para diferentes valores de $\lambda < 1$.

Tabela 4: *Leapfrog*, $\lambda < 1$

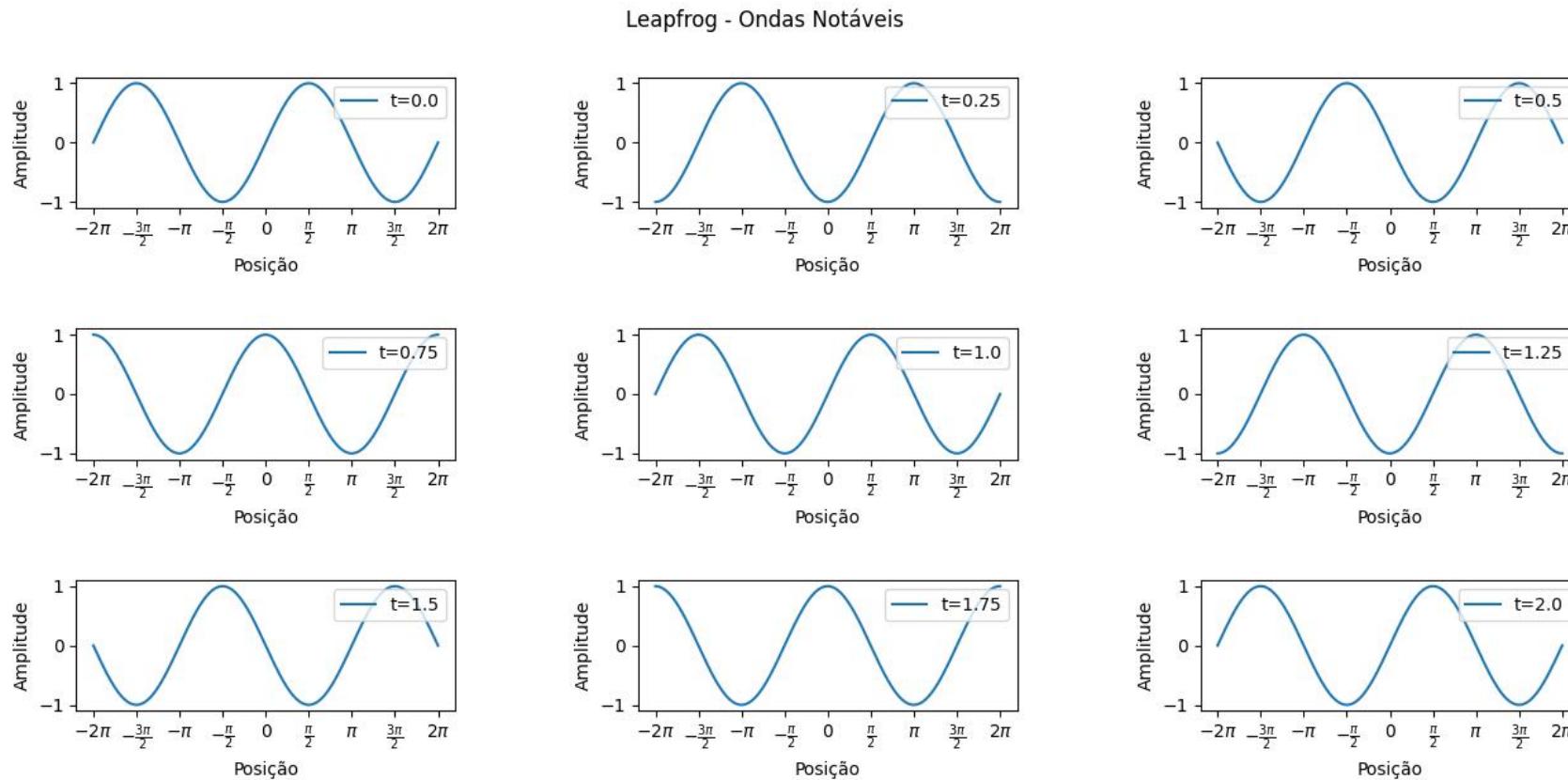
m	n	h	k	λ	Erro	Tempo	Ordem
32	40	0.0625	0.05	0.8	1.4517e-01	9.9921e-04	0.0000e+00
64	80	0.03125	0.025	0.8	3.0814e-02	0.0000e+00	2.2361e+00
128	160	0.015625	0.0125	0.8	7.3725e-03	1.9984e-03	2.0634e+00
256	320	0.0078125	0.00625	0.8	1.8232e-03	5.0173e-03	2.0157e+00
512	640	0.00390625	0.003125	0.8	4.5459e-04	1.1992e-02	2.0038e+00
1024	1280	0.001953125	0.0015625	0.8	1.1357e-04	2.9967e-02	2.0010e+00
32	64	0.0625	0.03125	0.5	2.7135e-01	0.0000e+00	0.0000e+00
64	128	0.03125	0.015625	0.5	6.2184e-02	1.9786e-03	2.1256e+00
128	256	0.015625	0.0078125	0.5	1.5232e-02	2.0139e-03	2.0294e+00
256	512	0.0078125	0.00390625	0.5	3.7904e-03	7.9796e-03	2.0067e+00
512	1024	0.00390625	0.001953125	0.5	9.4657e-04	1.6988e-02	2.0016e+00
1024	2048	0.001953125	0.0009765625	0.5	2.3658e-04	4.6975e-02	2.0004e+00
32	80	0.0625	0.025	0.4	2.9453e-01	9.9969e-04	0.0000e+00
64	160	0.03125	0.0125	0.4	6.9052e-02	1.9982e-03	2.0927e+00
128	320	0.015625	0.00625	0.4	1.7022e-02	3.9973e-03	2.0203e+00
256	640	0.0078125	0.003125	0.4	4.2429e-03	9.0134e-03	2.0043e+00
512	1280	0.00390625	0.0015625	0.4	1.0600e-03	2.1972e-02	2.0010e+00
1024	2560	0.001953125	0.00078125	0.4	2.6496e-04	5.9981e-02	2.0002e+00
32	160	0.0625	0.0125	0.2	3.1662e-01	1.9932e-03	0.0000e+00
64	320	0.03125	0.00625	0.2	7.7622e-02	2.9986e-03	2.0282e+00
128	640	0.015625	0.003125	0.2	1.9372e-02	1.0976e-02	2.0025e+00
256	1280	0.0078125	0.0015625	0.2	4.8443e-03	1.8989e-02	1.9996e+00
512	2560	0.00390625	0.00078125	0.2	1.2112e-03	4.6992e-02	1.9999e+00
1024	5120	0.001953125	0.000390625	0.2	3.0279e-04	1.2391e-01	2.0000e+00

Fonte: Os Autores, 2022

As Figuras 12 - 14 representam graficamente alguns resultados notáveis obtidos para esse método considerando $\lambda = 0.2$.

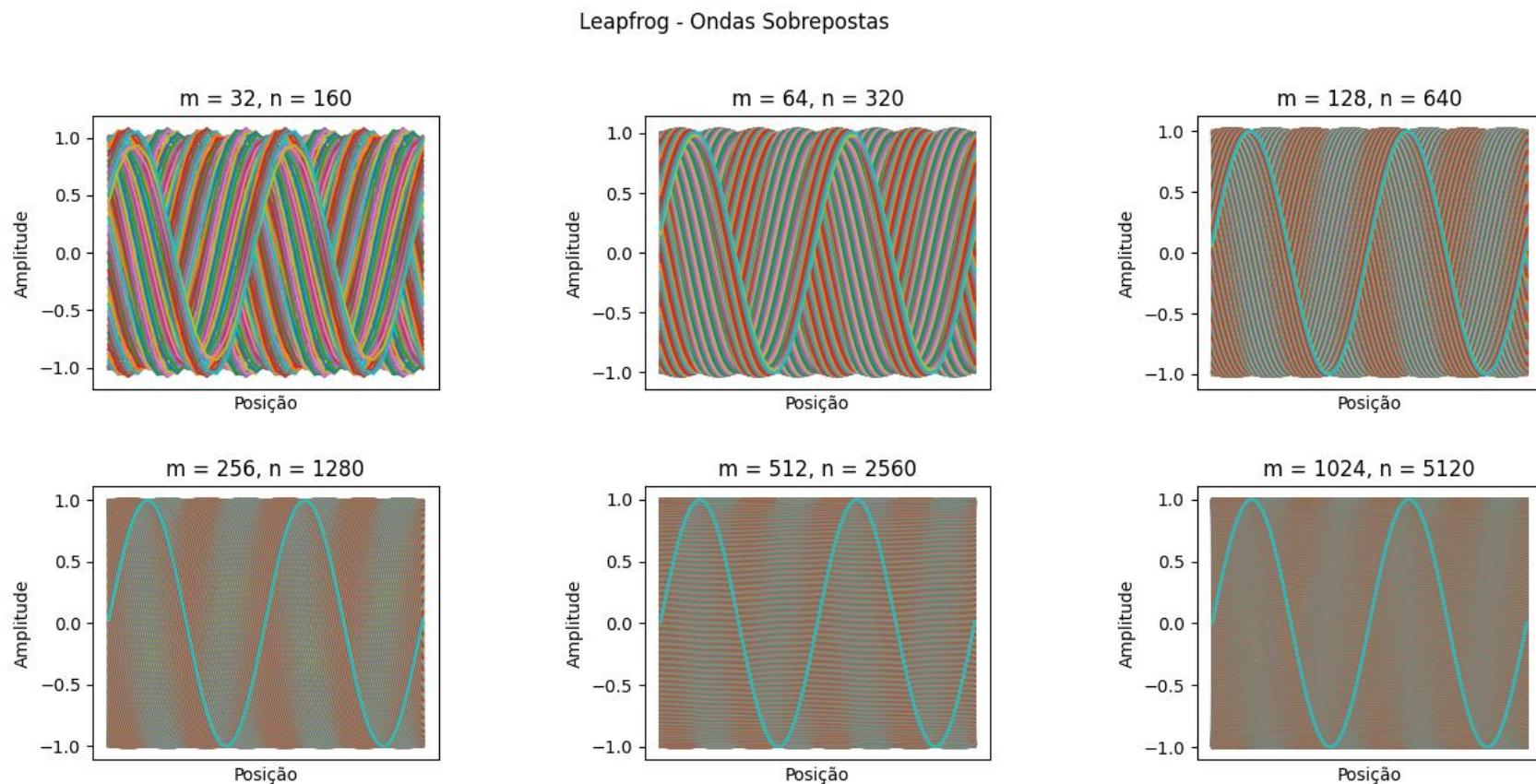
Vale ressaltar que por se tratarem de resultados, visualmente, bastante similares, demais gráficos foram suprimidos. No entanto, os mesmos podem ser obtidos mediante execução do algoritmo apresentado no Apêndice B ou no programa anexo.

Figura 12: *Leapfrog - Ondas Notáveis, $\lambda = 0.2$*



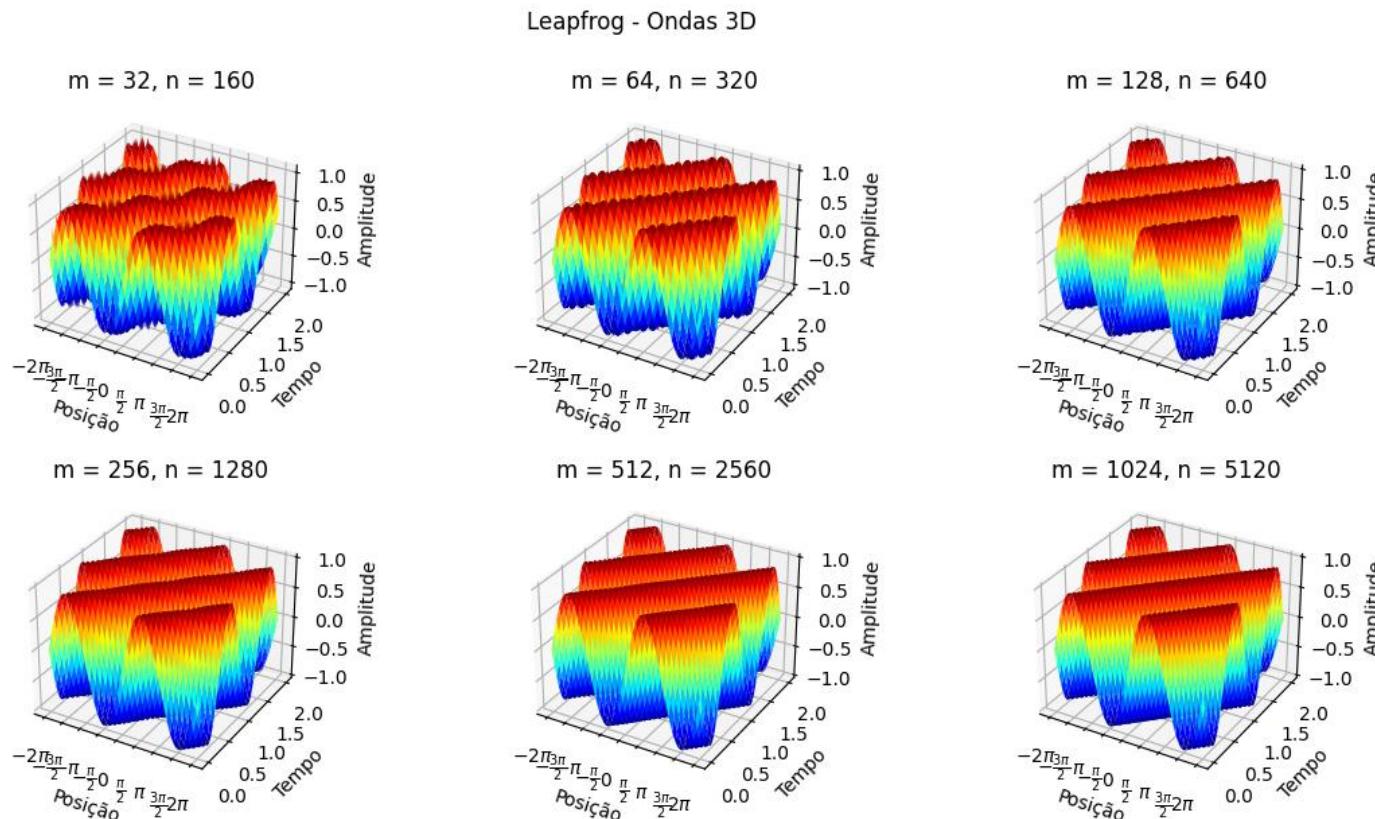
Fonte: Os Autores, 2022

Figura 13: *Leapfrog - Ondas Sobrepostas, $\lambda = 0.2$*



Fonte: Os Autores, 2022

Figura 14: *Leapfrog - Ondas 3D*, $\lambda = 0.2$



3.5 Discussões da Questão 01

Na Tabela 1 é apresentado o resultado da solução exata ($\lambda = 1$). Com essa condição de *CFL*, o erro tende a zero (10^{-15}). Esse erro é zero mesmo para uma divisão espacial de somente 32 partições. E mesmo que o tempo de processamento cresça mais de 23 vezes quando passamos de 32 divisões para 1024, ele é bem pequeno em termos absolutos ($4.6973 \cdot 10^{-2}$ segundos).

Na figura 3 apresentamos algumas ondas notáveis e mostramos seus deslocamento. A cada $\frac{25}{100}$ segundo, elas se deslocam $\frac{\pi}{2}$ da esquerda para a direita e completam um período a cada 2 segundos. A amplitude varia de -1 a 1 . A Figura 4 representa a sobreposição de algumas ondas com diferentes refinamentos da partição do domínio. A densidade de curvas apresentada em cada uma das sub-figuras segue a dessa partição, por exemplo, para $m = n = 32$ há 32 perfis de onda e para $m = n = 1024$ há 1024 perfis. A Figura 5 mostra o deslocamento da onda ao longo do tempo para diferentes níveis de refinamento da malha. As mesmas conclusões anteriores são vistas nesta figura, com perfil *3D* (posição \times tempo \times amplitude).

A Tabela 2 resume o comportamento da onda para diferentes valores de λ ($0.8, 0.5, 0.4, 0.2$). Para um mesmo nível de discretização espacial, quanto mais próximo o λ estiver de um, menores são os erros e os tempos de processamento relativos. Mesmo com a malha mais refinada para o caso $\lambda = 0.8$, i.e., 1024 divisões espaciais, o programa se mostrou mais rápido do que quando comparado com diversas execuções com menos divisões para os outros valores de λ . Ainda, quanto mais próximo o λ estiver da unidade, mais rápido ocorreu a velocidade convergência para a ordem certa (a ordem experimental de convergência para o método *FTBS* é 1 na dimensão espacial). Como esperado, apesar do aumento do tempo computacional, os erros diminuem conforme aumentamos o número de partição do domínio.

Nenhuma diferença significativa foi observada para as ondas notáveis no caso de $\lambda = 1$ e $\lambda = 0.2$ com a mesma partição (Figura 6). Entretanto, para $\lambda = 0.2$ e com menor nível de refinamento quando comparado ao caso exato, a Figura 7 mostra que a amplitude diminui a medida que a onda se desloca da esquerda para a direita (limitada pelo intervalo $[-1, 1]$), isso ocorre devido ao aumento do erro residual dos passos anteriores que se acumulam com o passar do tempo. Essa oscilação inicial tende a diminuir à medida em que aumentamos o número de divisões da espaciais. Esse mesmo efeito e seu contorno pode ser observado na Figura 8.

Esse resultados gerais podem ser estendidos à análise do método *Leapfrog*. Isto é, os resultados computacionais (menor tempo, menor erro, velocidade de convergência) são melhores quanto mais próximos os valores de λ estão de 1. A ordem de convergência experimental na dimensão espacial deste método é, no entanto, $O(h^2)$, isto é, ordem 2. Quando o comparamos com o método *FTBS*, percebemos pela Tabela 4 que ele foi marginalmente mais rápido.

4 Questão 2

Nesta seção apresentamos e discutimos a equação do calor em duas dimensões cuja formulação matemática encontra-se na Seção 4.1.

Logo em seguida, na Seção 4.2, apresentamos um esquema numérico denominado de Método de *Crank-Nicolson* para resolução dessa equação via MDF.

4.1 Formulação Matemática

Essa seção trata da formulação matemática bidimensional da Equação Parabólica do calor que, de forma particular, para este trabalho, é representada pela Equação 8.

$$\begin{aligned} u_t &= u_{xx} + u_{yy} + g(t, x, y) \quad 0 < \alpha < 1 \quad -0.5 \leq x, y \leq 0.5 \\ u(0, x, y) &= xy + \sin(\pi x) \cos(\pi y) \\ g(t, x, y) &= 2(\pi^2) \sin(\pi x) \cos(\pi y) - \alpha xy(e^{-\alpha t}) \end{aligned} \quad (8)$$

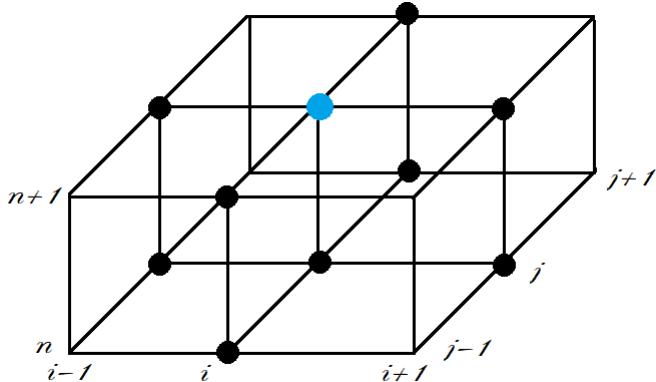
Nesta equação são consideradas condições de contorno e iniciais dadas por uma expressão manufaturada (Equação 9) e domínio contido no intervalo $-0.5 \leq x, y \leq 0.5$. A constante de condutividade térmica é considerada unitária.

$$u(t, x, y) = xy e^{-\alpha t} + \sin(\pi x) \cos(\pi y) \quad (9)$$

4.2 Método das Diferenças Finitas - MDF

O estêncil para o Método de *Crank-Nicolson* é apresentado logo abaixo, na Figura 15.

Figura 15: Estêncil de *Crank-Nicolson*



Fonte: Os Autores, 2022

Para a EDP apresentada na Equação 10:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + g(t, x, y) \quad (10)$$

o método de Crank-Nicolson pode ser desenvolvido da seguinte forma (Equação 11):

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} = \frac{a}{2} \left[\left(\frac{u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1}}{(\Delta x)^2} + \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{(\Delta x)^2} \right) \right. \\ \left. + \left(\frac{u_{i,j+1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j-1}^{n+1}}{(\Delta y)^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{(\Delta y)^2} \right) \right] + \frac{g(u_{i,j}^{n+1})}{2} + \frac{g(u_{i,j}^n)}{2} \quad (11)$$

Para o caso particular, onde $\Delta x = \Delta y$ podemos reescrever esta equação conforme o apresentado na Equação 12:

$$u_{i,j}^{n+1} - u_{i,j}^n = \frac{a\Delta t}{2\Delta x^2} \left[(u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) \right. \\ \left. + (u_{i,j+1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j-1}^{n+1} + u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n) \right] + \frac{\Delta t}{2} (g(u_{i,j}^{n+1}) + g(u_{i,j}^n)) \quad (12)$$

Definindo $\lambda = \frac{a\Delta t}{\Delta x^2}$ temos (Equação 13):

$$u_{i,j}^{n+1} - \frac{\lambda}{2} (u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j-1}^{n+1}) = u_{i,j}^n + \frac{\lambda}{2} [u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n \\ + u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n] + \frac{\Delta t}{2} (g(u_{i,j}^{n+1}) + g(u_{i,j}^n)) \quad (13)$$

Reorganizando a Equação 13 de tal forma que os termos u^{n+1} estejam do lado esquerdo da igualdade e os termos u^n estejam do lado direito da igualdade, podemos fazer (Equação 14):

$$u_{i,j}^{n+1} - \frac{\lambda}{2} (u_{i+1,j}^{n+1} - 4u_{i,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n+1} + u_{i,j-1}^{n+1}) = u_{i,j}^n + \frac{\lambda}{2} [u_{i+1,j}^n - 4u_{i,j}^n + u_{i-1,j}^n \\ + u_{i,j+1}^n + u_{i,j-1}^n] + \frac{\Delta t}{2} (g(u_{i,j}^{n+1}) + g(u_{i,j}^n)) \quad (14)$$

Agrupando os termos entre parênteses em comum temos (Equação 15):

$$u_{i,j}^{n+1} - \frac{\lambda}{2} (-4u_{i,j}^{n+1} + u_{i+1,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n+1} + u_{i,j-1}^{n+1}) = u_{i,j}^n + \frac{\lambda}{2} [-4u_{i,j}^n u_{i+1,j}^n + u_{i-1,j}^n \\ + u_{i,j+1}^n + u_{i,j-1}^n] + \frac{\Delta t}{2} (g(u_{i,j}^{n+1}) + g(u_{i,j}^n)) \quad (15)$$

Finalmente reagrupando todos os termos em comum fica (Equação 16):

$$(1 + 2\lambda)u_{i,j}^{n+1} - \frac{\lambda}{2} (u_{i+1,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n+1} + u_{i,j-1}^{n+1}) = (1 - 2\lambda)u_{i,j}^n + \frac{\lambda}{2} [u_{i+1,j}^n + u_{i-1,j}^n \\ + u_{i,j+1}^n + u_{i,j-1}^n] + \frac{\Delta t}{2} (g(u_{i,j}^{n+1}) + g(u_{i,j}^n)) \quad (16)$$

Esta última Equação é o chamado Método de *Crank-Nicolson*, discretizado para um domínio bidimensional.

Logo a seguir, apresentamos um esquema matricial (Equação 17) do Método de Crank-Nicolson para uma malha de 3×3 pontos:

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & -\frac{\lambda}{2} & -\frac{\lambda}{2} & 1+2\lambda & -\frac{\lambda}{2} & -\frac{\lambda}{2} \\ & & & & 1 & \\ & & & & & 1 \\ & & & & & & 1 \end{bmatrix} \begin{bmatrix} U_{i,j}^{n+1} \\ U_{i+1,j}^{n+1} \\ U_{i+2,j}^{n+1} \\ U_{i,j+1}^{n+1} \\ U_{i+1,j+1}^{n+1} \\ U_{i+2,j+1}^{n+1} \\ U_{i,j+2}^{n+1} \\ U_{i+1,j+2}^{n+1} \\ U_{i+2,j+2}^{n+1} \end{bmatrix} = \\ \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & \frac{\lambda}{2} & \frac{1}{2} & 1-2\lambda & \frac{\lambda}{2} & \frac{\lambda}{2} \\ & & & & 1 & \\ & & & & & 1 \\ & & & & & & 1 \end{bmatrix} \begin{bmatrix} U_{i,j}^n \\ U_{i+1,j}^n \\ U_{i+2,j}^n \\ U_{i,j+1}^n \\ U_{i+1,j+1}^n \\ U_{i+2,j+1}^n \\ U_{i,j+2}^n \\ U_{i+1,j+2}^n \\ U_{i+2,j+2}^n \end{bmatrix} + \frac{\Delta t}{2} \begin{bmatrix} g(U_{i,j}^{n+1}) + g(U_{i,j}^n) \\ g(U_{i+1,j}^{n+1}) + g(U_{i+1,j}^n) \\ g(U_{i+2,j}^{n+1}) + g(U_{i+2,j}^n) \\ g(U_{i,j+1}^{n+1}) + g(U_{i,j+1}^n) \\ g(U_{i+1,j+1}^{n+1}) + g(U_{i+1,j+1}^n) \\ g(U_{i+2,j+1}^{n+1}) + g(U_{i+2,j+1}^n) \\ g(U_{i,j+2}^{n+1}) + g(U_{i,j+2}^n) \\ g(U_{i+1,j+2}^{n+1}) + g(U_{i+1,j+2}^n) \\ g(U_{i+2,j+2}^{n+1}) + g(U_{i+2,j+2}^n) \end{bmatrix} \quad (17)$$

Observe que para um domínio bidimensional esta matriz é do tipo pentadiagonal. Este fenômeno é explicado pelo estêncil que, para cada ponto de interesse exige a avaliação dos valores de seus 4 vizinhos adjacentes (para um mesmo instante temporal), dois quais, para uma mesma coordenada j temos duas diagonais adjacentes a principal, deslocadas de uma unidade e, para uma mesma coordenada i , duas outras diagonais deslocadas de uma quantidade de pontos compatível com a dimensão da malha (neste caso, 3 pontos a partir da diagonal principal).

Note, no entanto, que para o contorno, o valor da diagonal principal é 1 e todo os outros pontos da mesma linha são zero. Isto é explicado pelo fato do problema modelado possuir em todas as duas extremidades condições de contorno de *Dirichlet* e, portanto, valores conhecidos que serão utilizados para a determinação dos valores dos pontos de sua vizinhança.

5 Resultados e Discussões da Questão 02

Nesta seção apresentamos os resultados obtidos na implementação da solução numérica. Apresentamos duas tabelas que nos permitem comparar o tempo computacional utilizando matrizes cheias e esparsas e verificar a ordem de convergência do método, fornecido pela teoria como sendo, na dimensão temporal, igual a 2. Também são apresentadas algumas figuras que representam uma comparação entre a solução analítica e a numérica para o método supracitado.

5.1 Resultados para *Crank-Nicolson*

Na Tabela 5 são apresentados alguns dos principais resultados obtidos na determinação da convergência temporal do Método de *Crank-Nicolson*. Vale ressaltar ainda, que nesta tabela os valores são separados por agrupamentos de diferentes valores de $\Delta x = \Delta y$ e os resultados são determinados considerando um método de manipulação algébrica de matriz cheia.

Tabela 5: Convergência temporal para diferentes $\Delta x = \Delta y$ - Matriz Cheia

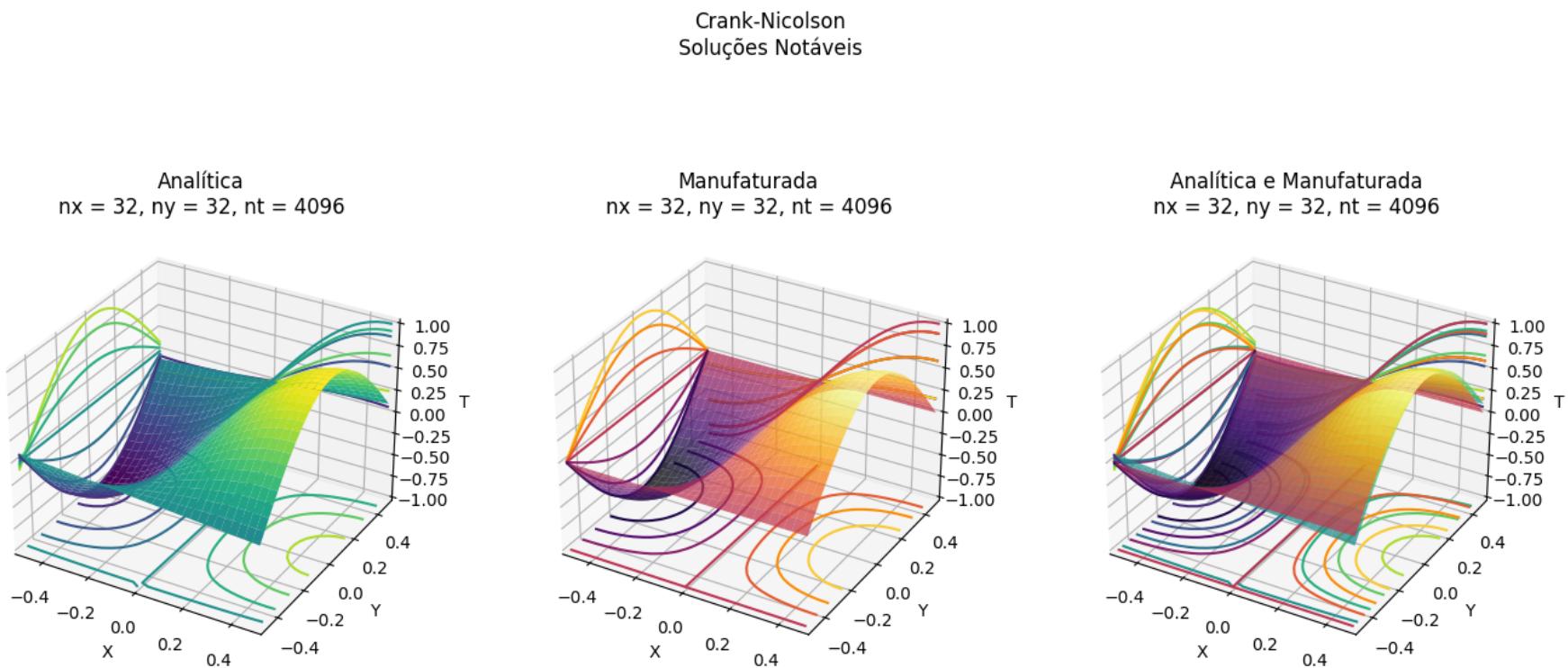
nx	ny	nt	dx	dy	dt	λ	Erro	Tempo	Ordem
8	8	128	0.125	0.125	0.0625	4.0	1.754e-02	4.615e-01	0.0000e+00
8	8	256	0.125	0.125	0.03125	2.0	1.754e-02	5.906e-01	0.0000e+00
8	8	512	0.125	0.125	0.015625	1.0	1.754e-02	2.020e+00	1.9991e+00
8	8	1024	0.125	0.125	0.0078125	0.5	1.754e-02	3.501e+00	1.9997e+00
8	8	2048	0.125	0.125	0.00390625	0.25	1.754e-02	4.987e+00	1.9999e+00
8	8	4096	0.125	0.125	0.001953125	0.125	1.754e-02	9.903e+00	1.9997e+00
8	8	8192	0.125	0.125	0.0009765625	0.0625	1.754e-02	1.971e+01	2.0016e+00
16	16	128	0.0625	0.0625	0.0625	16.0	8.792e-03	1.887e+00	0.0000e+00
16	16	256	0.0625	0.0625	0.03125	8.0	8.792e-03	4.113e+00	0.0000e+00
16	16	512	0.0625	0.0625	0.015625	4.0	8.792e-03	7.368e+00	2.0145e+00
16	16	1024	0.0625	0.0625	0.0078125	2.0	8.792e-03	1.540e+01	1.9998e+00
16	16	2048	0.0625	0.0625	0.00390625	1.0	8.792e-03	3.882e+01	1.9999e+00
16	16	4096	0.0625	0.0625	0.001953125	0.5	8.792e-03	7.082e+01	2.0000e+00
16	16	8192	0.0625	0.0625	0.0009765625	0.25	8.792e-03	1.308e+02	1.9998e+00
32	32	128	0.03125	0.03125	0.0625	64.0	4.476e-03	2.085e+01	0.0000e+00
32	32	256	0.03125	0.03125	0.03125	32.0	4.396e-03	3.027e+01	0.0000e+00
32	32	512	0.03125	0.03125	0.015625	16.0	4.396e-03	5.820e+01	1.0484e+01
32	32	1024	0.03125	0.03125	0.0078125	8.0	4.396e-03	1.148e+02	2.0012e+00
32	32	2048	0.03125	0.03125	0.00390625	4.0	4.396e-03	2.327e+02	1.9999e+00
32	32	4096	0.03125	0.03125	0.001953125	2.0	4.396e-03	4.559e+02	2.0000e+00
32	32	8192	0.03125	0.03125	0.0009765625	1.0	4.396e-03	9.901e+02	2.0001e+00

Fonte: Os Autores, 2022

As Figuras 16 e 17 representam graficamente alguns resultados notáveis obtidos para esse método considerando manipulações algébricas de matriz cheia.

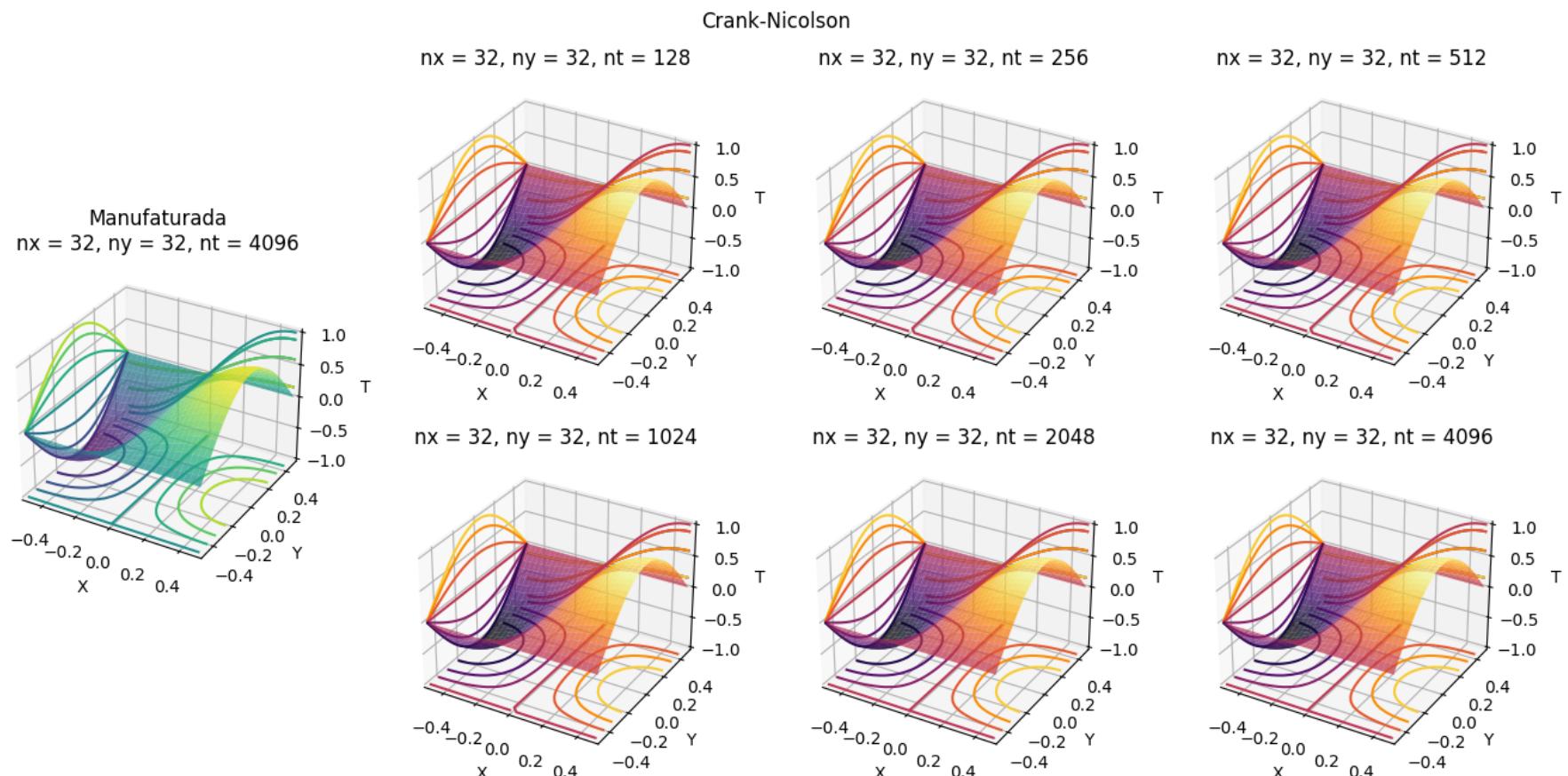
Enquanto a primeira figura apresenta os resultados analítico e numérico para um mesmo nível de refinamento isoladas e sobrepostas, a segunda apresenta uma evolução do nível de refinamento da dimensão temporal frente a solução manufaturada.

Figura 16: Crank-Nicolson - Matriz Cheia



Fonte: Os Autores, 2022

Figura 17: Crank-Nicolson - Matriz Cheia



Fonte: Os Autores, 2022

Na Tabela 6 são apresentados alguns dos principais resultados obtidos na determinação da convergência temporal do Método de *Crank-Nicolson*. Vale ressaltar ainda, que nesta tabela os valores são separados por agrupamentos de diferentes valores de $\Delta x = \Delta y$ e os resultados são determinados considerando um método que se beneficia da esparsidade da matriz.

Tabela 6: Convergência temporal para diferentes $\Delta x = \Delta y$ - Matriz Esparsa

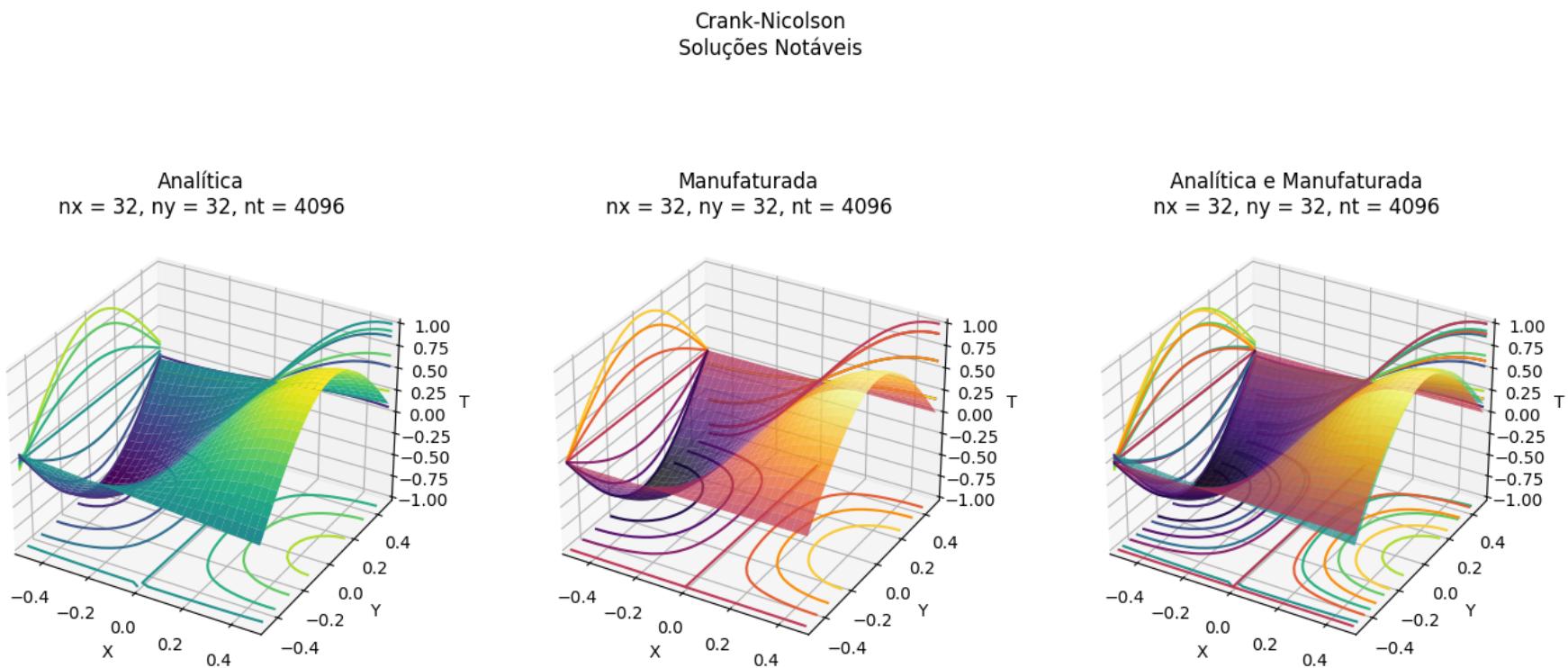
nx	ny	nt	dx	dy	dt	λ	Erro	Tempo	Ordem
8	8	128	0.125	0.125	0.0625	4.0	1.754e-02	2.988e-01	0.0000e+00
8	8	256	0.125	0.125	0.03125	2.0	1.754e-02	6.156e-01	0.0000e+00
8	8	512	0.125	0.125	0.015625	1.0	1.754e-02	1.236e+00	1.9991e+00
8	8	1024	0.125	0.125	0.0078125	0.5	1.754e-02	2.454e+00	1.9997e+00
8	8	2048	0.125	0.125	0.00390625	0.25	1.754e-02	4.928e+00	1.9999e+00
8	8	4096	0.125	0.125	0.001953125	0.125	1.754e-02	9.726e+00	1.9998e+00
8	8	8192	0.125	0.125	0.0009765625	0.0625	1.754e-02	2.050e+01	1.9995e+00
16	16	128	0.0625	0.0625	0.0625	16.0	8.792e-03	1.074e+00	0.0000e+00
16	16	256	0.0625	0.0625	0.03125	8.0	8.792e-03	3.269e+00	0.0000e+00
16	16	512	0.0625	0.0625	0.015625	4.0	8.792e-03	5.310e+00	2.0145e+00
16	16	1024	0.0625	0.0625	0.0078125	2.0	8.792e-03	9.786e+00	1.9998e+00
16	16	2048	0.0625	0.0625	0.00390625	1.0	8.792e-03	1.735e+01	1.9999e+00
16	16	4096	0.0625	0.0625	0.001953125	0.5	8.792e-03	3.579e+01	2.0000e+00
16	16	8192	0.0625	0.0625	0.0009765625	0.25	8.792e-03	7.067e+01	2.0002e+00
32	32	128	0.03125	0.03125	0.0625	64.0	4.476e-03	5.367e+00	0.0000e+00
32	32	256	0.03125	0.03125	0.03125	32.0	4.396e-03	1.326e+01	0.0000e+00
32	32	512	0.03125	0.03125	0.015625	16.0	4.396e-03	2.067e+01	1.0484e+01
32	32	1024	0.03125	0.03125	0.0078125	8.0	4.396e-03	4.022e+01	2.0012e+00
32	32	2048	0.03125	0.03125	0.00390625	4.0	4.396e-03	8.872e+01	1.9999e+00
32	32	4096	0.03125	0.03125	0.001953125	2.0	4.396e-03	3.017e+02	2.0000e+00
32	32	8192	0.03125	0.03125	0.0009765625	1.0	4.396e-03	7.122e+02	1.9999e+00

Fonte: Os Autores, 2022

As Figuras 18 e 19 representam graficamente alguns resultados notáveis obtidos para esse método considerando manipulações algébricas que se beneficiam da esparsidade da matriz.

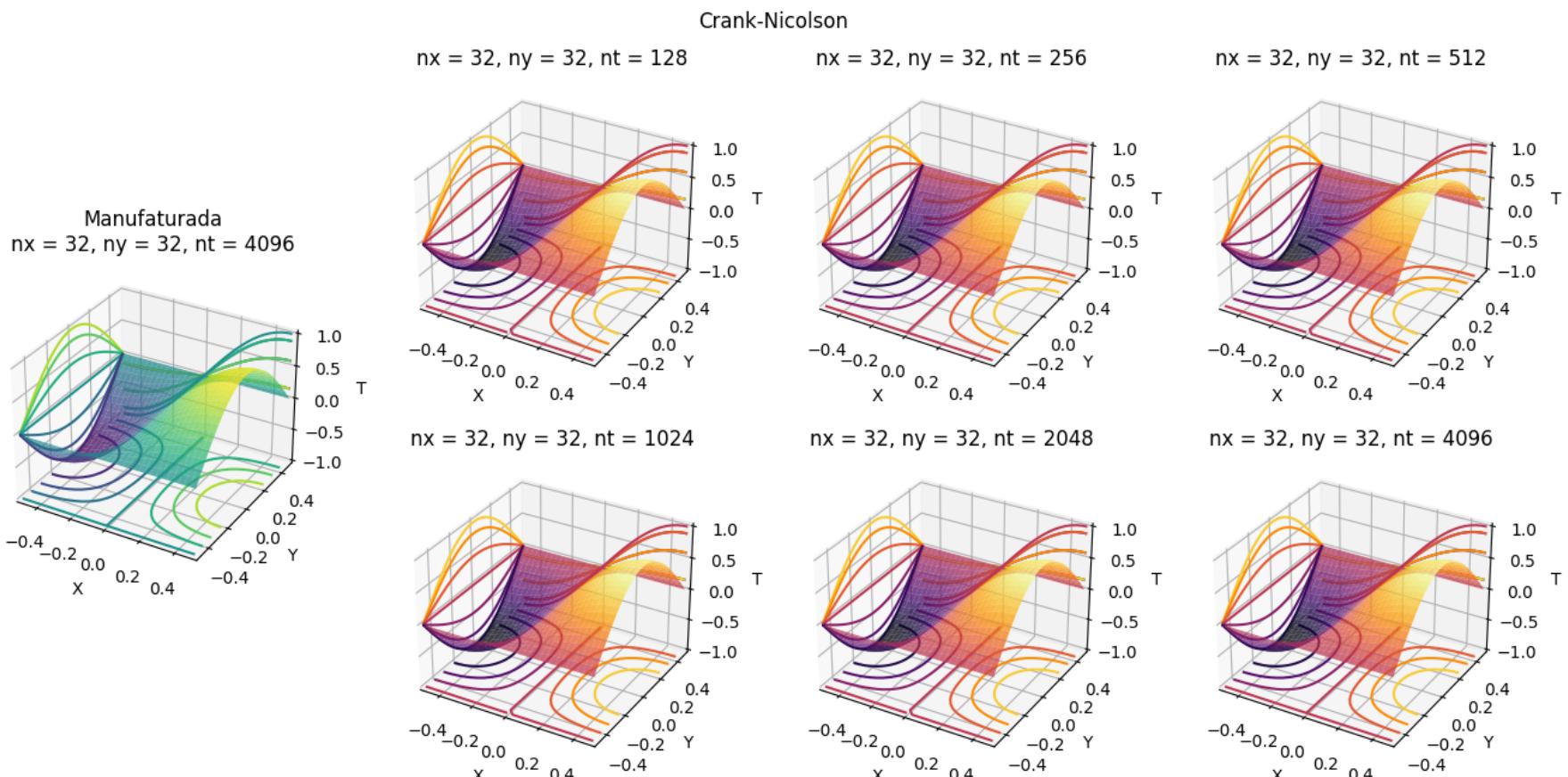
Enquanto a primeira figura apresenta os resultados analítico e numérico para um mesmo nível de refinamento isoladas e sobrepostas, a segunda apresenta uma evolução do nível de refinamento da dimensão temporal frente a solução manufaturada.

Figura 18: Crank-Nicolson - Matriz Esparsa



Fonte: Os Autores, 2022

Figura 19: Crank-Nicolson - Matriz Esparsa



Fonte: Os Autores, 2022

5.2 Discussões da Questão 02

As três perguntas do enunciado do exercício nortearão nossa discussão apresentada abaixo. Elas dizem respeito ao benefício relativo da adoção de matrizes esparsas (questão a), sobre a estabilidade do método (questão b) e sobre a ordem do erro (questão c).

Sobre este último questionamento (questão c), olhando para as Tabelas 5, 6, percebemos a correta convergência na dimensão temporal do método para a ordem 2 conforme refinamos a malha na dimensão temporal para um mesmo nível de discretização espacial (8, 16 e 32 partições). Esse resultado é alcançado tanto utilizando matrizes cheias quanto esparsas. E a principal diferença entre essas duas abordagens (questão a) está refletida no tempo computacional. O tempo total despendido para rodar o conjunto de rotinas apresentado nas tabelas foi praticamente 1,6 vezes maior ao utilizarmos as matrizes cheias (36.88, contra 22.75 minutos ao utilizarmos matrizes esparsas). De fato, o tempo com matrizes esparsas foi inferior em qualquer nível de partição utilizada.

Adotamos como critério para demarcação do regime permanente a solução proposta, de forma que nosso período final é $t_T = \lceil \frac{3}{\alpha} \ln 10 \rceil = 8$. Essa implementação é confirmada ao analisarmos a variação da curva após esse período, que se mostra praticamente invariável ao longo do tempo. Criamos uma animação que mostra a oscilação da superfície nesses segundos inciais e a anexamos aos arquivos entregues.

Em termos experimentais, não foi observado nenhum limite de estabilidade do Método de *Crank-Nicolson*. Este resultado é evidenciado pelas Tabelas 5 e 6 que deixam claro que mesmo para valores de λ variando de 0.0625 até 64 a convergência temporal para ordem 2 é alcançada.

Assim, resta-nos conversar sobre a estabilidade teórica do Método de *Crank-Nicolson* (questão b). Precisamos avaliar se o método é incondicionalmente estável ou se há alguma condição de estabilidade associado a ele.

Para responder a essa questão, é necessário que performemos uma análise de estabilidade de Von Neumann. Para o caso correspondente a versão unidimensional, o erro pode ser definido como sendo igual a (Equação 18):

$$\frac{\varepsilon_j^{n+1} - \varepsilon_j^n}{\Delta t} = \frac{\alpha}{2} \left[\frac{\varepsilon_{j+1}^n - \varepsilon_j^n + \varepsilon_{j-1}^n}{\Delta x^2} \right] + \frac{\alpha}{2} \left[\frac{\varepsilon_{j+1}^{n+1} - \varepsilon_j^{n+1} + \varepsilon_{j-1}^{n+1}}{\Delta x^2} \right] \quad (18)$$

$$\varepsilon_j^n = \sum_{m=0}^{M-1} \exp(\beta_m n \Delta t) \exp(i j \theta_m) = \sum_{m=0}^{M-1} \xi_m^n \exp(i j \theta_m) \quad (19)$$

Substituindo a representação de Fourier para o erro (Equação 19) na Equação 18, obtemos:

$$\begin{aligned} & \sum_{m=0}^{M-1} \xi_m^{n+1} \exp(i j \theta_m) - \sum_{m=0}^{M-1} \xi_m^n \exp(i j \theta_m) \\ &= \frac{\alpha \Delta t}{2(\Delta x)^2} \left[\sum_{m=0}^{M-1} \xi_m^n \exp(i[j+1]\theta_m) - 2 \sum_{m=0}^{M-1} \xi_m^n \exp(i j \theta_m) + \sum_{m=0}^{M-1} \xi_m^n \exp(i[j-1]\theta_m) \right] \\ &+ \frac{\alpha \Delta t}{2(\Delta x)^2} \left[\sum_{m=0}^{M-1} \xi_m^{n+1} \exp(i[j+1]\theta_m) - 2 \sum_{m=0}^{M-1} \xi_m^{n+1} \exp(i j \theta_m) + \sum_{m=0}^{M-1} \xi_m^{n+1} \exp(i[j-1]\theta_m) \right] \end{aligned} \quad (20)$$

Dividindo por $\xi_m^n \exp(ij\theta_m)$, obtemos:

$$\begin{aligned}
\xi_m - 1 &= \frac{\alpha\Delta t}{2(\Delta x)^2} [\exp(i\theta_m) - 2 + \exp(-i\theta_m)] \\
&\quad + \frac{\alpha\Delta t}{2(\Delta x)^2} \xi_m [\exp(i\theta_m) - 2 + \exp(-i\theta_m)] \\
&= \frac{\alpha\Delta t}{(\Delta x)^2} [\cos(\theta_m) - 1] + \frac{\alpha\Delta t}{(\Delta x)^2} \xi_m [\cos(\theta_m) - 1] = -\frac{2\alpha\Delta t}{(\Delta x)^2} [\xi_m + 1] \sin^2\left(\frac{\theta_m}{2}\right)
\end{aligned} \tag{21}$$

Rearranjando a Equação 21, temos:

$$\xi_m \left[1 + \frac{2\alpha\Delta t}{(\Delta x)^2} \sin^2\left(\frac{\theta_m}{2}\right) \right] = 1 - \frac{2\alpha\Delta t}{(\Delta x)^2} \sin^2\left(\frac{\theta_m}{2}\right) \tag{22}$$

Denotando a quantidade $[\frac{2\alpha\Delta t}{(\Delta x)^2}] \sin^2\left(\frac{\theta_m}{2}\right)$ por ζ_m , então a Equação 22 pode ser reescrita como (Equação 23):

$$\xi_m = \frac{1 - \zeta_m}{1 + \zeta_m} \tag{23}$$

Logo, a estabilidade depende do critério $-1 \leq \zeta_m \leq 1$ ser satisfeito. Uma vez que a quantidade $[\frac{2\alpha\Delta t}{(\Delta x)^2}] \sin^2\left(\frac{\theta_m}{2}\right)$, segue que o limite superior da desigualdade é sempre verdadeiro. O limite inferior do lado direito da Equação 23 ocorrerá para grandes valores positivos positivos de ζ_m e tenderá a -1 no limite $\zeta_m \rightarrow \infty$. Para qualquer valor finito de ζ_m , o lado direito é maior do que -1 . Em resumo, ambos os limites do critério de estabilidade são satisfeitos independentemente do valor de ζ_m .

Assim, o método de *Crank-Nicolson* é incondicionalmente estável para a equação de difusão transiente. Isso o torna uma escolha atraente para o cálculo de problemas instáveis, pois a precisão pode ser aprimorada sem perda de estabilidade com quase o mesmo custo computacional por etapa de tempo. Outra maneira de interpretar o aumento da ordem de precisão do método é que passos de tempo maiores podem ser usados para obter precisão comparável com os métodos de Euler para frente ou para trás. Esta é uma vantagem significativa para problemas em que um grande número de passos de tempo deve ser executado para alcançar o instante de tempo desejado. O mesmo resultado é estendível ao método de *Crank-Nicolson* bidimensional [2].

6 Conclusão

O objetivo desse trabalho foi estudar a aplicação do Método das Diferenças Finitas na resolução de duas EDPs. A primeira Hiperbólica, utilizando-se os esquemas numéricos *Forward in Time, Backward in Space*, e *Leapfrog* e a segunda Parabólica, via Método de *Crank-Nicolson*.

O estudo foi desenvolvido a nível computacional e implementado na linguagem computacional denominada *Python 3* e sua implementação obteve as corretas ordens dos erros para cada um dos métodos. A saber, 1 para o *FTBS* e 2 para o *leapfrog* e para o *Crank-Nicolson*.

Para ambos os métodos de Diferenças Finitas, a teoria nos mostra que eles são capazes de se aproximar satisfatoriamente dos resultados analíticos com erro relativamente baixo, embora o tempo de processamento cresça significativamente para o método de *Crank-Nicolson* a medida em que aumentamos o número de pontos para integração temporal e mesmo nível de refinamento espacial da malha.

Em termos de estabilidade, foi observado que para $\lambda = 1$ os dois métodos implementados para a EDP Hiperbólica possuem solução exata. No entanto, estes métodos possuem estabilidade condicionada a um valor de *CFL* de no máximo 1. Por outro lado, o método de *Crank-Nicolson* é incondicionalmente estável e mesmo para valores relativamente elevados ou reduzidos de λ a convergência experimental na dimensão temporal foi observada e validada como sendo igual a 2.

Embora este estudo tenha acontecido de forma preliminar, levando-se em conta até dois métodos diferentes, vale ressaltar que ainda há inúmeros outros métodos de solução a serem testados e, portanto, uma avaliação mais profunda ainda deve ser conduzida a fim de se determinar qual o melhor método frente a relação custo-benefício para a resolução de equações do tipo elípticas.

Sugere-se que, em trabalhos futuros, sejam desenvolvidos outros métodos de solução como, por exemplo, o Método dos Elementos Finitos ou Volumes Finitos.

Referências

- [1] Richard L. Burden; J. Douglas Faires. *Numerical Analysis*. BROOKS/COLE, 2011.
- [2] Sandip Mazumder. *Numerical Methods for Partial Differential Equations: Finite Difference and Finite Volume Methods*. Academic Press, 2016.
- [3] Wikipedia. Método das diferenças finitas.
- [4] Éliton Fontana. *Introdução ao Método de Diferenças Finitas com Aplicações em Engenharia Química*. Universidade Federal do Paraná - UFPR, 2019.

Apêndice A - Algoritmo FTBS

```
1 # Projeto Final
2 # EDP Hiperbolica - Forward in Time, Backward in Space (FTBS)
3
4 ######
5
6 # Importa Bibliotecas
7 import numpy as np
8 import matplotlib.pyplot as plt
9 from matplotlib.animation import FuncAnimation, PillowWriter
10 from prettytable import PrettyTable
11 import time
12
13 #####
14
15 """
16             COMO USAR
17 -----
18 Lambda = 1.0                      const = 1.00
19 Lambda = 0.8                      const = 1.25
20 Lambda = 0.5                      const = 2.00
21 Lambda = 0.4                      const = 2.50
22 Lambda = 0.2                      const = 5.00
23
24 """
25
26 # Parametro do Lambda
27 const = 1.00
28
29 #####
30
31 # Parametros da funcao
32 a = 1                                # Constante
33
34 x_0 = -1                             # Extremidade inferior do dominio
35 x_L = 1                               # Extremidade superior do dominio
36
37 t_0 = 0                               # Tempo inicial
38 t_T = (x_L-x_0)/a                   # Tempo final
39
40 m = [2**i for i in range(5,11)] # Quantidade de espacamentos na ←
        direcao x
41 n = [int(const*i) for i in m]    # Quantidade de espacamentos na ←
        direcao t
42
43 #####
44
45 # Velocidade inicial da onda
46 v = lambda x: np.sin(2*np.pi*x)
```

```

47
48 ######
49
50 # Figura 1 - Ondas Notaveis
51 def fig1(result):
52     fig = plt.figure()
53     fig.suptitle(f"Forward in Time, Backward in Space (FTBS) - ←
54                 Ondas Notaveis")
55     for i in range(9):
56         ax = fig.add_subplot(3,3,i+1)
57         t = (t_T-t_0)/8 * i
58         ax.plot(result[-1][0][int(result[-1][3]/8 * i)],label=f"←
59                  t={t:.3f}")
60         ax.legend(loc="upper right")
61         ax.set(xlabel="Posi o", ylabel='Amplitude')
62         pos_x= [result[-1][2]/8*i for i in range(9)]
63         eixo_x = [r"$-2\pi$",r"$-\frac{3\pi}{2}$",r"$-\pi$",r"$\leftarrow
64                  -\frac{\pi}{2}$","0",r"$\frac{\pi}{2}$",r"$\pi$",r"$\leftarrow
65                  \frac{3\pi}{2}$",r"$2\pi$"]
66     plt.xticks(pos_x, eixo_x)
67     plt.tight_layout()
68     plt.show()
69
70 # Figura 2 - Ondas Sobrepostas
71 def fig2(result):
72     fig= plt.figure()
73     fig.suptitle(f"Forward in Time, Backward in Space (FTBS) - ←
74                 Sobrepostas")
75     for i in range(len(result)):
76         if i <=3:
77             ax = fig.add_subplot(2,3,i+1)
78         if i > 3:
79             ax = fig.add_subplot(2,3,i+1)
80         for k in range(result[i][3]):
81             ax.plot(result[i][0][k])
82         plt.xticks([],[])
83         ax.set(title=f"m = {result[i][2]}, n = {result[i][3]}",
84                xlabel="Posi o", ylabel="Amplitude")
85     plt.tight_layout()
86     plt.show()
87
88 # Figura 3 - Ondas 3D
89 def fig3(result):
90     fig = plt.figure()
91     fig.suptitle(f"Forward in Time, Backward in Space (FTBS) - ←
92                 Ondas 3D")
93     for i in range(len(result)):
94         x = np.linspace(x_0,x_L,result[i][2]+1)
95         y = [j*result[i][5] for j in range(result[i][3]+1)]
96         Y, X = np.meshgrid(x,y)
97         W = np.array(result[i][0])

```

```

91     ax = fig.add_subplot(2,3,i+1,projection='3d')
92     ax.plot_surface(Y,X,W,cmap="jet")
93     pos_x= [-1+j*1/4 for j in range(9)]
94     eixo_x = [r"$-2\pi$",r"$-\frac{3\pi}{2}$",r"$-\pi$",r"$-\frac{\pi}{2}$",
95                 ,r"$0$",r"$\frac{\pi}{2}$",r"$\pi$",r"$\frac{3\pi}{2}$",r"$2\pi$"]
96     plt.xticks(pos_x, eixo_x)
97     ax.set(title=f"m = {result[i][2]}, n = {result[i][3]}",
98            xlabel="Posicao", ylabel="Tempo", zlabel="Amplitude")
99     plt.subplots_adjust(hspace=0.3)
100    plt.show()
101 #####
102
103 # Animacao
104 def ani(result):
105     val = result[-1][0]
106     fig = plt.figure()
107     ax = plt.axes(title="Animacao\nForward in Time, Bakward in Space (FTBS)",
108                   xlim=(-1,1), ylim=(-1,1),
109                   xlabel="Posicao", ylabel="Amplitude")
110     line, = ax.plot([], [])
111
112 # Funcao Inicializadora
113 def init():
114     line.set_data([],[])
115     return line,
116
117 # Fun o Animar
118 def animate(i):
119     x = np.linspace(-1,1,len(val))
120     y = val[i]
121     line.set_data(x,y)
122     return line,
123
124 # Animacao
125 anim = FuncAnimation(fig,animate,np.arange(len(val)),  

126                      init_func=init,interval=5,blit=True)
127
128 plt.show()
129
130 # Salvar
131 # writer = PillowWriter(fps=5)
132 # anim.save(f"ani1.gif", writer=writer)
133 #####
134
135 # Tabela - Resultados Notaveis
136 def tab1(result):

```

```

137     print(f"Forward in Time, Backward in Space (FTBS)")
138     tabela = PrettyTable()
139     tabela.field_names = ["m ", "n ", "h ", "k ", "Lambda ", "←
140         Erro", "Tempo", "Ordem"]
141     for i in range(len(result)):
142
142         # Ordem de convergencia
143         if i == 0:
144             ordem = 0
145         else:
146             ordem = np.log2(np.abs(result[i-1][7]/result[i][7]))
147
148         tabela.add_row([result[i][2], result[i][3], result[i][4], ←
149                         result[i][5], result[i][6], \
150                         f"{result[i][7]:.4e}", f"{result[i][8]:.4←
151                         e}", f"{ordem:.4e}"])
152
153 #####
154 # Forward in Time, Backward in Space (FTBS)
155 def ftbs(x_0,x_L,t_0,t_T,m,n,a,v):
156
157     # Tempo
158     ti = time.time()
159
160     # Passo 1
161     h = (x_L - x_0)/m    # Numero de espaceamento dos pontos na ←
162         direcao x
162     k = (t_T - t_0)/n    # Numero de espaceamento dos pontos na ←
163         direcao t
163     Lambda = a*k/h
164
165     # Passo 2
166     x = np.linspace(x_0,x_L,m+1)      # Discretizacao do espace
167
168
169     # Passo 3
170     sol = []                          # Vetor que armazena←
171         s solucoes
171     u0 = v(x)                         # Onda inicial
172     u = u0.copy()                     # Variavel ←
172         temporaria
173     for _ in range(n+1):               # Loop temporal
174         u = u -Lambda*(u - np.roll(u,1)) # MDF - FTBS
175         sol.append(u)                  # Armazena a solucao←
175         atual
176
177     # Solucao exata
178     ue = v(x+k*n)
179

```

```

180     # Erro
181     erro = max(abs(u-ue))    # Norma Infinito
182
183     # Tempo
184     tempo = time.time() - ti    # Tempo de processamento
185
186     # Resultados
187     resultado = (sol,x,m,n,h,k,Lambda,erro,tempo)    # Vetor ←
188             resultados
189
190     return (resultado)
191 #####
192
193 # Executa o programa
194 def Q1p1():
195
196     # Diferentes valores de "h"
197     result = []
198     for i in range(len(m)):
199         result.append(ftbs(x_0,x_L,t_0,t_T,m[i],n[i],a,v))
200
201     # Plota os Graficos
202     fig1(result)
203     fig2(result)
204     fig3(result)
205
206     # Animacao
207     if const == 1:
208         ani(result)
209
210     # Plota a Tabela
211     tab1(result)
212
213 # Chamada da funcao
214 Q1p1()
215 #####
216 #####
217
218 print("Fim")

```

Apendice B - Algoritmo Leapfrog

```
1 # Projeto Final
2 # EDP Hiperbolica - Leapfrog
3
4 ##########
5
6 # Importa Bibliotecas
7 import numpy as np
8 import matplotlib.pyplot as plt
9 from matplotlib.animation import FuncAnimation, PillowWriter
10 from prettytable import PrettyTable
11 import time
12
13 #####
14
15 """
16             COMO USAR
17 -----
18 Lambda = 1.0                      const = 1.00
19 Lambda = 0.8                      const = 1.25
20 Lambda = 0.5                      const = 2.00
21 Lambda = 0.4                      const = 2.50
22 Lambda = 0.2                      const = 5.00
23
24 """
25
26 # Parametro do Lambda
27 const = 1.00
28
29 #####
30
31 # Parametros da funcao
32 a = 1                                # Constante
33
34 x_0 = -1                             # Extremidade inferior do dominio
35 x_L = 1                               # Extremidade superior do dominio
36
37 t_0 = 0                               # Tempo inicial
38 t_T = (x_L-x_0)/a                    # Tempo final
39
40 m = [2**i for i in range(5,11)] # Quantidade de espacamentos na ←
        direcao x
41 n = [int(const*i) for i in m]    # Quantidade de espacamentos na ←
        direcao t
42
43 #####
44
45 # Velocidade inicial da onda
46 omega = 2*np.pi
```

```

47 v = lambda x: np.sin(omega*x)
48
49 ##########
50
51 # Figura 1 - Ondas Separadas
52 def fig1(result):
53     fig = plt.figure()
54     fig.suptitle(f"Leapfrog - Ondas Notaveis")
55     for i in range(9):
56         ax = fig.add_subplot(3,3,i+1)
57         t = (t_T-t_0)/8 * i
58         ax.plot(result[-1][0][int(result[-1][3]/8 * i)],label=f"←
59             t={t:.3f}")
60         ax.legend(loc="upper right")
61         ax.set(xlabel="Posicao", ylabel='Amplitude')
62         pos_x= [result[-1][2]/8*i for i in range(9)]
63         eixo_x = [r"$-2\pi$",r"$-\frac{3\pi}{2}$",r"$-\pi$",r"$\leftarrow
64             \frac{\pi}{2}$", "0",r"$\frac{\pi}{2}$",r"$\pi$",r"$\leftarrow
65             \frac{3\pi}{2}$",r"$2\pi$"]
66         plt.xticks(pos_x, eixo_x)
67     plt.tight_layout()
68     plt.show()
69
70 # Figura 2 - Ondas Sobrepostas
71 def fig2(result):
72     fig= plt.figure()
73     fig.suptitle(f"Leapfrog - Ondas Sobrepostas")
74     for i in range(len(result)):
75         if i <=3:
76             ax = fig.add_subplot(2,3,i+1)
77         if i > 3:
78             ax = fig.add_subplot(2,3,i+1)
79         for k in range(result[i][3]):
80             ax.plot(result[i][0][k])
81         plt.xticks([],[])
82         ax.set(title=f"m = {result[i][2]}, n = {result[i][3]}",
83               xlabel="Posicao", ylabel="Amplitude")
84     plt.tight_layout()
85     plt.show()
86
87 # Figura 3 - Ondas 3D
88 def fig3(result):
89     fig = plt.figure()
90     fig.suptitle(f"Leapfrog - Ondas 3D")
91     for i in range(len(result)):
92         x = np.linspace(x_0,x_L,result[i][2]+1)
93         y = [j*result[i][5] for j in range(result[i][3]+1)]
94         X,Y = np.meshgrid(x,y)
95         W = np.array(result[i][0])
96         ax = fig.add_subplot(2,3,i+1,projection='3d')
97         ax.plot_surface(X,Y,W,cmap="jet")

```

```

94     pos_x= [-1+j*1/4 for j in range(9)]
95     eixo_x = [r"$-2\pi$",r"${-\frac{3\pi}{2}}$,"$-\pi$",
96                 r"${-\frac{\pi}{2}}$,"0",r"${\frac{\pi}{2}}$",
97                 r"${\frac{3\pi}{2}}$,"$2\pi$"]
98
99     plt.xticks(pos_x, eixo_x)
100    ax.set(title=f"m = {result[i][2]}, n = {result[i][3]}",
101           xlabel="Posicao", ylabel="Tempo", zlabel="Amplitude")
102
103    plt.subplots_adjust(hspace=0.3)
104    plt.show()
105
106 #####
107
108 # Animacao
109 def ani(result):
110     val = result[-1][0] # primeiro nivel ja e config.
111     inicial de anim
112     fig = plt.figure()
113     ax = plt.axes(title="Anima o\nLeapfrog",
114                   xlim=(-1,1), ylim=(-1,1),
115                   xlabel="Posicao", ylabel="Amplitude")
116     line, = ax.plot([], [])
117
118     # Funcao Inicializadora
119     def init():
120         line.set_data([],[])
121         return line,
122
123     # Animar
124     def animate(i):
125         x = np.linspace(-1,1,len(val))
126         y = val[i]
127         line.set_data(x,y)
128         return line,
129
130     # Animacao
131     anim = FuncAnimation(fig,animate,np.arange(len(val)),
132                         init_func=init,interval=5,blit=True)
133
134 #####
135
136 # Tabela - Resultados Notaveis
137 def tab1(result):
138     print(f"Leapfrog")
139     tabela = PrettyTable()

```

```

140     tabela.field_names = ["m ", "n ", "h ", "k ", "Lambda ", "←
141         Erro", "Tempo", "Ordem"]
142     for i in range(len(result)):
143
143         # Ordem de convergencia
144         if i == 0:
145             ordem = 0
146         else:
147             ordem = np.log2(np.abs(result[i-1][7]/result[i][7]))
148
149         tabela.add_row([result[i][2], result[i][3], result[i][4], ←
150                         result[i][5], result[i][6], \
150                         f"[result[i][7]:.4e]", f"[result[i][8]:.4←
150                           e]", f"{ordem:.4e}"])
151     print(tabela)
152
153 ##########
154
155 # Leapfrog
156 def lf(x_0,x_L,t_0,t_T,m,n,a,v):
157
158     # Tempo
159     ti = time.time()
160
161     # Passo 1
162     h = (x_L - x_0)/m    # Numero de espacamento dos pontos na ←
162         direcao x
163     k = (t_T - t_0)/n    # Numero de espacamento dos pontos na ←
163         direcao t
164     Lambda = a*k/h
165
166     # Passo 2
167     x = np.linspace(x_0,x_L,m+1)      # Particao do espaco
168
169     # Passo 3
170     u1 = v(x)
171     u2 = v(x+k)
172     sol = []
173     sol.append(u1)                  # Armazena primeiro nivel
174     sol.append(u2)                  # Armazena segundo nivel
175     for _ in range(2,n+1):
176         u0 = np.copy(u1)
177         u1 = np.copy(u2)
178         u2[1:-1] = u0[1:-1] - Lambda*(u1[2::] - u1[:-2])
179         u2[0]   = u0[0]   - Lambda*(u1[1] - u1[-2])
180         u2[-1] = u2[0]
181         u = u2.copy()
182         sol.append(u)
183
184     # Solucao exata
185     ue = v(x+k*n)

```

```

186
187     # Erro
188     erro = max(abs(sol[-1]-ue))
189
190     # Tempo
191     tempo = time.time() - ti
192
193     # Resultados
194     resultado = (sol,x,m,n,h,k,Lambda,erro,tempo)      # Vetor ←
195             resultados
196
197     return (resultado)
198 #####
199
200 # executa o programa
201 def Q1p2():
202
203     # diferentes valores de "h"
204     result = []
205     for i in range(len(m)):
206         result.append(lf(x_0,x_L,t_0,t_T,m[i],n[i],a,v))
207
208     # Plota os Graficos
209     fig1(result)
210     fig2(result)
211     fig3(result)
212
213     # Animacao
214     if const == 1:
215         ani(result)
216
217     # Plota a Tabela
218     tab1(result)
219
220 # Chamada da funcao
221 Q1p2()
222 #####
223 #####
224
225 print("Fim")

```

Apêndice C - Algoritmo *Crank-Nicolson*

```
1 # Projeto Final
2 # EDP Parabolica - Crank-Nicholson
3
4 ######
5
6 # Importa Bibliotecas
7 from tkinter import ttk
8 import numpy as np
9 from numpy.linalg import norm
10 # np.set_printoptions(precision=4)
11 np.set_printoptions(linewidth=150)
12 import matplotlib.pyplot as plt
13 from matplotlib.animation import FuncAnimation, PillowWriter
14 from prettytable import PrettyTable
15 import scipy.linalg
16 import scipy.sparse as ss
17 import scipy.sparse.linalg
18 import time
19
20 #####
21
22 # Constantes
23 a = 1 # Constante de difusividade ←
24 alpha = 0.9 # Parametro da solucao ←
25         manufaturada
26
27 # Dominio
28 x_0 = -0.5 # Extremidade inferior de x
29 x_L = 0.5 # Extremidade superior de x
30 y_0 = -0.5 # Extremidade inferior de y
31 y_L = 0.5 # Extremidade superior de y
32 t_0 = 0 # Tempo inicial
33 t_T = np.ceil((3/alpha)*np.log(10)) # Tempo final
34
35 # Numero de divisoes
36 nx = 2**5 # Tamanho do passo em x
37 ny = 2**5 # Tamanho do passo em y
38 nt = [2**i for i in range(7,13)] # Tamanho do passo em t
39 #####
40
41 # Solucao manufaturada
42 utxy = lambda t,x,y,alpha=alpha: x*y*np.e**(-alpha*t) + np.sin(np.pi*x)*np.cos(np.pi*y)
43
44 # Termo forcante
```

```

45 g = lambda t,x,y,alpha=alpha: 2*(np.pi**2)*np.sin(np.pi*x)*np.←
    cos(np.pi*y) - alpha*x*y*(np.e**(-alpha*t))
46
47 ######
48
49 # Crank-Nicholson Bidimensional
50 def cn(x_0,x_L,y_0,y_L,t_0,t_T,a,alpha,nx,ny,nt, tipo):
51
52     # Inicio do tempo
53     ti = time.time()
54
55     # Tamanho dos espacamentos
56     dx = (x_L - x_0)/nx                      # Tamanho em x
57     dy = (y_L - y_0)/ny                      # Tamanho em y
58     dt = (t_T - t_0)/nt                      # Tamanho no tempo
59
60     # Lambda
61     Lambda = (a*dt/(dx**2))
62
63     # Particoes
64     x = np.linspace(x_0,x_L,nx+1)           # Particao em x
65     y = np.linspace(y_0,y_L,ny+1)           # Particao em y
66     t = np.linspace(t_0,t_T,nt+1)           # Particao no tempo
67
68     # Meshgrid
69     X, Y = np.meshgrid(x,y)
70
71     # Condicoes iniciais
72     W0 = utxy(t[0],X,Y)
73
74     # Constroi as matrizes A e B da equacao: Aw^{n+1} = Bw^n
75     n = (nx+1)*(ny+1)
76     def Matriz(Lambda, n, ny, const):
77
78         D_d = (1 + const*2*Lambda) * np.ones(n)          # ←
79             Diagonal principal
80         D_d [::ny+1] = 1                                # OESTE
81         D_d [ny::ny+1] = 1                             # LESTE
82         D_d [:ny+1] = 1                                # SUL
83         D_d [-(ny+1)::] = 1                            # NORTE
84
85         D_b = (-const*Lambda/2) * np.ones(n-1)          # ←
86             Diagonal inferior
87         D_b [ny::ny+1] = 0                            # OESTE
88         D_b [ny-1::ny+1] = 0                         # LESTE
89         D_b [:ny+1] = 0                                # SUL
90         D_b [-(ny+1)::] = 0                          # NORTE
91
92         D_b_ny = (-const*Lambda/2) * np.ones(n-(ny+1)) # ←
93             Diagonal inferior defasada de ny
94         D_b_ny [ny::ny+1] = 0                         # OESTE

```

```

92         D_b_ny [::ny+1] = 0                                # LESTE
93         D_b_ny [-(ny+1)::] = 0                            # NORTE
94
95         D_c = (-const*Lambda/2) * np.ones(n-1)           # ←
96             Diagonal superior
97         D_c [::ny+1] = 0                                # OESTE
98         D_c [ny::ny+1] = 0                            # LESTE
99         D_c [:ny+1] = 0                                # SUL
100        D_c [-(ny)::] = 0                            # NORTE
101
102        D_c_ny = (-const*Lambda/2) * np.ones(n-(ny+1)) # ←
103            Diagonal superior defasada de ny
104        D_c_ny [::ny+1] = 0                            # OESTE
105        D_c_ny [ny::ny+1] = 0                            # LESTE
106        D_c_ny [:ny+1] = 0                            # SUL
107
108    # Diagonais das matrizes A e B da equacao: Aw^{n+1} = Bw^{n}
109    elem_A = Matriz(Lambda, n, ny, const=1)
110    elem_B = Matriz(Lambda, n, ny, const=-1)
111
112    # Matrizes A e B da equacao: Aw^{n+1} = Bw^{n}
113    if tipo.upper() == "CHEIA":
114        A_cheia = np.diag(elem_A[2], -(ny+1)) + np.diag(elem_A[1], -1) + np.diag(elem_A[0]) + np.diag(elem_A[3], 1) + np.diag(elem_A[4], (ny+1))
115        B_cheia = np.diag(elem_B[2], -(ny+1)) + np.diag(elem_B[1], -1) + np.diag(elem_B[0]) + np.diag(elem_B[3], 1) + np.diag(elem_B[4], (ny+1))
116    if tipo.upper() == "ESPARSA":
117        A_esparsa = ss.diags([elem_A[0], elem_A[3], elem_A[1], elem_A[4], elem_A[2]], [0, 1, -1, ny+1, -(nx+1)], format='csc')
118        B_esparsa = ss.diags([elem_B[0], elem_B[3], elem_B[1], elem_B[4], elem_B[2]], [0, 1, -1, ny+1, -(nx+1)], format='csc')
119
120    # Transforma uma matriz em um vetor C-B
121    def ordenar_CB(matriz):
122        vet = []
123        for i in range(len(matriz)):
124            for j in range(len(matriz[i])):
125                vet.append(matriz[i][j])
126        return vet
127
128    # Loop temporal
129    w_sol = []
130    w = ordenar_CB(W0)
131    w_sol.append(w)
132    xx = ordenar_CB(X)

```

```

133     yy = ordenar_CB(Y)
134     for i in range(1,nt+1):
135         vetG = []
136         for j in range(n):
137             dtgg = dt*0.5*(g(t[i-1],xx[j],yy[j]) + g(t[i],xx[j],yy[j]))
138             vetG.append(dtgg)
139
140         # Zera o termo forcante no contorno
141         for k in range(ny+1):                      # SUL
142             vetG[k] = 0
143         for k in range(-(ny+1),0):                  # NORTE
144             vetG[k] = 0
145         for k in range((ny+1),len(vetG),ny+1):      # OESTE
146             vetG[k] = 0
147         for k in range(ny,len(vetG),ny+1):          # LESTE
148             vetG[k] = 0
149
150     # Resolve matriz cheia ou esparsa
151     if tipo.upper() == "CHEIA":
152         w_novo = scipy.linalg.solve(A_cheia, B_cheia @ w_sol[-1] + vetG)
153     if tipo.upper() == "ESPARSA":
154         w_novo = scipy.sparse.linalg.spsolve(A_esparsa, B_esparsa @ w_sol[-1] + vetG)
155
156     # Dirichlet no contorno
157     for k in range(ny+1):                      # SUL
158         w_novo[k] = utxy(t[i],xx[k],yy[k])
159     for k in range(-(ny+1),0):                  # NORTE
160         w_novo[k] = utxy(t[i],xx[k],yy[k])
161     for k in range((ny+1),len(w_novo),ny+1):    # OESTE
162         w_novo[k] = utxy(t[i],xx[k],yy[k])
163     for k in range(ny,len(w_novo),ny+1):          # LESTE
164         w_novo[k] = utxy(t[i],xx[k],yy[k])
165
166     w = w_novo
167     w_sol.append(w)
168
169     # Solucao final
170     w_sol_p = []
171     for i in range(len(w_sol)):                  # Numerica
172         val = np.reshape(w_sol[i],(nx+1,ny+1))   # vetor >> matriz
173         w_sol_p.append(val)
174     WFINAL = utxy(t[-1],X,Y)                    # Analitica
175
176     # Erro
177     erro = np.linalg.norm(w_sol_p[-1]-WFINAL)
178
179     # Tempo

```

```

180     tempo = time.time() - ti
181
182     return [w_sol_p,X,Y,nx,ny,nt,dx,dy,dt,Lambda,erro,tempo]
183
184 ##########
185
186 # Figura com "t" mais refinado
187 def fig1(result):
188     X, Y = result[-1][1], result[-1][2]
189
190     WFINAL = utxy(1,X,Y)          # Analitica
191     wfinal = result[-1][0][-1]    # Numerica
192
193     # Plota as solucoes
194     fig = plt.figure()
195     fig.suptitle(f"Crank-Nicolson\nSolucoes Notaveis")
196     for i in range(3):
197         if i == 0:
198             ax = fig.add_subplot(1,3,i+1, projection='3d',alpha=←
199                         =0.75)
200             ax.plot_surface(X,Y,WFINAL,cmap="viridis",label="←
201                             Analitico")
202             cset = ax.contour(X,Y,WFINAL,zdir='x',offset=x_0, ←
203                               cmap="viridis")
204             cset = ax.contour(X,Y,WFINAL,zdir='y',offset=y_L, ←
205                               cmap="viridis")
206             cset = ax.contour(X,Y,WFINAL,zdir='z',offset=-1, ←
207                               cmap="viridis")
208             ax.set(title=f"Analitica\nnx = {result[-1][3]}, ny =←
209                     {result[-1][4]}, nt = {result[-1][5]}",\←
210                     xlabel='X', ylabel='Y', zlabel='T',\←
211                     xlim3d=(x_0,x_L), ylim3d=(y_0,y_L), zlim3d=(-1,1))
212         if i == 1:
213             ax = fig.add_subplot(1,3,i+1, projection='3d')
214             ax.plot_surface(X,Y,wfinal,cmap="inferno", label="←
215                             Crank-Nicholson",alpha=0.75)
216             cset = ax.contour(X,Y,wfinal,zdir='x',offset=x_0, ←
217                               cmap="inferno")
218             cset = ax.contour(X,Y,wfinal,zdir='y',offset=y_L, ←
219                               cmap="inferno")
220             cset = ax.contour(X,Y,wfinal,zdir='z',offset=-1, ←
221                               cmap="inferno")
222             ax.set(title=f"Manufaturada\nnx = {result[-1][3]}, ←
223                     ny = {result[-1][4]}, nt = {result[-1][5]}",\←
224                     xlabel='X', ylabel='Y', zlabel='T',\←
225                     xlim3d=(x_0,x_L), ylim3d=(y_0,y_L), zlim3d=(-1,1))
226         if i == 2:
227             ax = fig.add_subplot(1,3,i+1,projection="3d")
228             ax.plot_surface(X,Y,WFINAL,cmap="viridis",label="←
229                             Analitico",alpha=0.75)

```

```

218         cset = ax.contour(X,Y,WFINAL,zdir='x',offset=x_0, ←
219                         cmap="viridis")
220         cset = ax.contour(X,Y,WFINAL,zdir='y',offset=y_L, ←
221                         cmap="viridis")
222         cset = ax.contour(X,Y,WFINAL,zdir='z',offset=-1, ←
223                         cmap="viridis")
224         ax.plot_surface(X,Y,wfinal,cmap="inferno", label="←
225                         Crank-Nicholson",alpha=0.75)
226         cset = ax.contour(X,Y,wfinal,zdir='x',offset=x_0, ←
227                         cmap="inferno")
228         cset = ax.contour(X,Y,wfinal,zdir='y',offset=y_L, ←
229                         cmap="inferno")
230         cset = ax.contour(X,Y,wfinal,zdir='z',offset=-1, ←
231                         cmap="inferno")
232         ax.set(title=f"Analitica e Manufaturada\nnx = {←
233             result[-1][3]}, ny = {result[-1][4]}, nt = {←
234             result[-1][5]}",\
235             xlabel='X', ylabel='Y', zlabel='T', \
236             xlim3d=(x_0,x_L), ylim3d=(y_0,y_L), zlim3d=(-1,1))
237
238     plt.tight_layout()
239     plt.subplots_adjust(wspace=0.25, hspace=0.225, right=0.95)
240     plt.show()
241
242 # Figura comparando variacao dos "t"
243 def fig2(result):
244     fig = plt.figure()
245     fig.suptitle(f"Crank-Nicolson")
246
247     # Solucao Manufaturada
248     X1, Y1 = result[-1][1], result[-1][2]
249     ax1 = fig.add_subplot(1,4,1, projection='3d')
250     ax1.plot_surface(X1, Y1, result[-1][0][-1], cmap="viridis", ←
251                     alpha=0.75)
252     cset = ax1.contour(X1, Y1, result[-1][0][-1], zdir='x', ←
253                         offset=x_0, cmap="viridis")
254     cset = ax1.contour(X1, Y1, result[-1][0][-1], zdir='y', ←
255                         offset=y_L, cmap="viridis")
256     cset = ax1.contour(X1, Y1, result[-1][0][-1], zdir='z', ←
257                         offset=-1, cmap="viridis")
258     ax1.set(title=f"Manufaturada\nnx = {result[-1][3]}, ny = {←
259             result[-1][4]}, nt = {result[-1][5]}",\
260             xlabel='X', ylabel='Y', zlabel='T', \
261             xlim3d=(x_0,x_L), ylim3d=(y_0,y_L), zlim3d=(-1,1))
262
263     # Solucoes Numericas
264     for i in range(len(result)):
265         X, Y = result[i][1], result[i][2]
266         if i in (0,1,2):
267             ax = fig.add_subplot(2,4,i+2, projection='3d')
268         if i in (3,4,5):

```

```

255         ax = fig.add_subplot(2,4,i+3, projection='3d')
256         ax.plot_surface(X, Y, result[i][0][-1], cmap="inferno", ←
257             alpha=0.75)
257         cset = ax.contour(X, Y, result[i][0][-1], zdir='x', ←
258             offset=x_0, cmap="inferno")
258         cset = ax.contour(X, Y, result[i][0][-1], zdir='y', ←
259             offset=y_L, cmap="inferno")
259         cset = ax.contour(X, Y, result[i][0][-1], zdir='z', ←
260             offset=-1, cmap="inferno")
260         ax.set(title=f"nx = {result[i][3]}, ny = {result[i][4]}, ←
261             nt = {result[i][5]}", \
261             xlabel="X", ylabel="Y", zlabel="T", \
262             xlim3d=(x_0,x_L), ylim3d=(y_0,y_L), zlim3d=(-1,1) ←
263             )
263
264     plt.tight_layout()
265     plt.subplots_adjust(wspace=0.25, hspace=0.225, right=0.95)
266     plt.show()
267
268 ##########
269
270 # Animacao
271 def ani(x_0,x_L,y_0,y_L,nx,ny,alpha):
272
273     # Funcao manufaturada: u(t,x,y)
274     def func(x, y, t):
275         return x*y*np.e**(-alpha*t) + np.sin(np.pi*x)*np.cos(np.←
276             pi*y)
276
277     # Regime permanente
278     t_rp = 8
279
280     # Dados para aplicar na animacao
281     x = np.linspace(x_0,x_L,nx)
282     y = np.linspace(y_0,y_L,ny)
283     tt = []
284     for i in range(t_rp):
285         tt.append(i*t_rp/101)
286     for i in range(t_rp,t_0,-1):
287         tt.append(i*t_rp/101)
288
289     # Grid
290     xx, yy = np.meshgrid(x,y)
291
292     # Define a superficie
293     fig = plt.figure()
294     ax = fig.add_subplot(1,1,1,projection="3d")
295
296     # Funcao periodica
297     def update(i):
298         ww = func(xx,yy,tt[i])

```

```

299     ax.clear()
300     ax.plot_surface(xx,yy,ww,cmap="viridis")
301     cset = ax.contour(xx,yy,ww,zdir="x",offset=x_0,cmap="←
302                         viridis")
303     cset = ax.contour(xx,yy,ww,zdir="y",offset=y_L,cmap="←
304                         viridis")
305     cset = ax.contour(xx,yy,ww,zdir="z",offset=-1,cmap="←
306                         viridis")
307     ax.set(title=f"Solucao Manufaturada\n0<t<8 (regime ←
308             permanente)", \
309             xlabel="X", ylabel="Y", zlabel="T", \
310             xlim3d=(x_0,x_L), ylim3d=(y_0,y_L), zlim3d=(-1,1)←
311             )
312
313     # Plotagem para chamar a funcao update() periodicamente
314     anim = FuncAnimation(fig, update,np.arange(len(tt)))
315
316     plt.show()
317
318     #####
319     # Tabela - Resultados Notaveis
320     def tab1(result):
321         print(f"Crank-Nicolson")
322         tabela = PrettyTable()
323         tabela.field_names = ["nx","ny","nt","dx","dy","dt", "Lambda←
324                               ", "Erro", "Tempo", "Ordem"]
325         for i in range(len(result)):
326
327             # Ordem de convergencia
328             if i == 0 or i == 1:
329                 ordem = 0
330             else:
331                 ordem = np.log2((result[i-2][10]-result[i-1][10])/(
332                                 result[i-1][10]-result[i][10]))
333
334             tabela.add_row([result[i][3],result[i][4],result[i][5],←
335                           result[i][6],result[i][7],\
336                           result[i][8],result[i][9],f"{result[i][10]:.4e}",f"{result[i][11]:.4e}",f"←
337                           {ordem:.4e}"])
338
339     print(tabela)
340
341     #####
342     # Funcao resultados
343     def Q2(x_0,x_L,y_0,y_L,t_0,t_T,a,alpha,nx,ny,nt, tipo):

```

```

340     resultados = []
341     for i in range(len(nt)):
342         resultados.append(cn(x_0,x_L,y_0,y_L,t_0,t_T,a,alpha,nx,←
343                               ny,nt[i],tipo))
344
345     # Figuras
346     fig1(resultados)      # Figura com "t" mais refinado
347     fig2(resultados)      # Figura comparando variacao dos "t"
348
349     # Animacao
350     ani(x_0,x_L,y_0,y_L,nx,ny,alpha)
351
352     # Tabela
353     tab1(resultados)
354 #####
355
356     # Chama a funcao
357     Q2(x_0,x_L,y_0,y_L,t_0,t_T,a,alpha,nx,ny,nt, tipo="esparsa")
358
359 #####
360
361 print("Fim")

```
