Instituto Politécnico do Estado do Rio de Janeiro

Curso de Engenharia da Computação

Guilherme Cagide Fialho

Análise Comparativa de Métodos TVD Aplicados à Solução da Equação de Advecção

Nova Friburgo

2024

Instituto Politécnico do Estado do Rio de Janeiro

Graduação em Engenharia da Computação

Guilherme Cagide Fialho

Análise Comparativa de Métodos TVD Aplicados à Solução da Equação de Advecção

Relatório da Disciplina:
Métodos Numéricos para Equações
Diferenciais 2

Professor: Hélio Pedro Amaral Souto

Nova Friburgo

2024

RESUMO

CAGIDE FIALHO, G. Relatório do projeto de Métodos Numéricos para Equações Diferenciais II. 2024. 16 f. Trabalho da Disciplina Métodos Numéricos para Equações Diferenciais II (Graduação em Engenharia da Computação) – Universidade do Estado do Rio de Janeiro, Nova Friburgo, 2024.

Este trabalho analisa soluções numéricas para a equação de advecção unidimensional em um domínio com condições de contorno periódicas. Focando em métodos numéricos baseados em **TVD (Total Variation Diminishing)**, foram implementados os limitadores **Osher**, **Sweby** e **Van Albada** para avaliar sua eficiência e precisão. Os resultados das simulações numéricas foram comparados com soluções analíticas exatas, evidenciando as diferenças entre os métodos na preservação de monotonicidade, redução de oscilações e comportamento em regiões de gradiente suave. A análise destaca como os limitadores influenciam a dissipação e a dispersão, demonstrando sua aplicabilidade em problemas de transporte em fenômenos naturais e em engenharia.

Palavras-chave: Métodos Numéricos, Equação de Advecção, Métodos TVD, Simulação Numérica, Limitadores.

ABSTRACT

CAGIDE FIALHO, G. Project Report on Numerical Methods for Differential Equations II. 2024. 16 p. Course Completion Work for Numerical Methods for Differential Equations II (Bachelor's in Computer Engineering) – State University of Rio de Janeiro, Nova Friburgo, 2024.

This work analyzes numerical solutions for the one-dimensional advection equation in a domain with periodic boundary conditions. Focusing on **TVD (Total Variation Diminishing)** numerical methods, the limiters **Osher**, **Sweby**, and **Van Albada** were implemented to evaluate their efficiency and accuracy. The results of the numerical simulations were compared to exact analytical solutions, highlighting differences among the methods in preserving monotonicity, reducing oscillations, and handling smooth gradient regions. The analysis emphasizes how the limiters influence dissipation and dispersion, demonstrating their applicability in modeling transport problems in natural phenomena and engineering.

Keywords: Numerical Methods, Advection Equation, TVD Methods, Numerical Simulation, Limiters.

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Sumário

1 Objetivo

O objetivo deste trabalho é investigar e comparar soluções analíticas e numéricas para a equação de advecção unidimensional em um domínio com condições de contorno periódicas. O foco está na aplicação de métodos numéricos baseados em **TVD (Total Variation Diminishing)**, utilizando os limitadores **Osher**, **Sweby** e **Van Albada**, para avaliar a eficácia e as limitações de cada abordagem. A análise inclui comparações com a solução analítica exata, destacando a preservação da monotonicidade, a redução de oscilações e o comportamento dos métodos em regiões de gradiente suave. Este estudo visa fornecer insights sobre a aplicabilidade dos métodos TVD em problemas de transporte, contribuindo para o entendimento e a modelagem de fenômenos naturais e aplicações práticas em engenharia.

2 Introdução

A equação de advecção é uma ferramenta matemática essencial para modelar o transporte de substâncias em um fluido, sendo amplamente empregada em diversas áreas da engenharia e das ciências aplicadas. Sua formulação descreve como uma propriedade escalar, como a concentração de um traçador, se desloca ao longo do tempo em função de uma velocidade de advecção constante. Para problemas práticos, a solução da equação de advecção fornece uma compreensão valiosa sobre o comportamento de substâncias transportadas em meios físicos, como no estudo de escoamentos em reservatórios ou no transporte de poluentes em corpos d'água.

Neste trabalho, a equação de advecção unidimensional será resolvida utilizando o método dos volumes finitos, que preserva a forma conservativa da equação. Os fluxos nas interfaces dos volumes serão calculados com a introdução de termos anti-difusivos controlados por funções limitadoras específicas. Serão implementados três métodos numéricos do tipo **TVD (Total Variation Diminishing)**: Osher, Sweby e Van Albada. Esses métodos são conhecidos por sua capacidade de preservar a monotonicidade e reduzir oscilações artificiais próximas a descontinuidades ou gradientes íngremes.

Para garantir a estabilidade das simulações, o número de Courant será fixado em $C = 0,8$, respeitando a condição CFL. A condição inicial utilizada é composta por uma combinação de uma função gaussiana e uma função por partes, representando um perfil inicial com gradientes suaves e regiões de concentração constante. As simulações serão realizadas para os instantes de tempo $t = 1$ e $t = 5$, utilizando condições de contorno periódicas.

Os resultados obtidos serão comparados com a solução analítica exata, permitindo avaliar a precisão e o comportamento dos métodos numéricos em regiões de gradientes suaves e descontinuidades. Gráficos e tabelas serão gerados automaticamente, organizados em diretórios específicos, para facilitar a análise qualitativa e quantitativa das soluções.

Este estudo busca fornecer uma visão clara sobre as vantagens e limitações dos métodos TVD em problemas de transporte, contribuindo para sua aplicação em contextos práticos e na modelagem de fenômenos naturais e de engenharia.

3 Desenvolvimento Teórico

A equação de advecção unidimensional descreve o transporte de uma quantidade conservada, como a concentração de um traçador, ao longo de um eixo espacial. Para resolver essa equação numericamente, é utilizado o método dos Volumes Finitos, que permite a discretização do espaço e do tempo, garantindo uma formulação adequada para a conservação da quantidade transportada [1]. A equação de advecção, em sua forma conservativa, é dada por:

$$\frac{\partial \Phi}{\partial t} + \frac{\partial}{\partial x}(u\Phi) = 0, \tag{1}$$

onde Φ representa a variável dependente (concentração do traçador) e u é a velocidade de advecção. Com u constante, a equação simplifica-se para:

$$\frac{\partial \Phi}{\partial t} + u\frac{\partial \Phi}{\partial x} = 0. \tag{2}$$

Neste trabalho, a solução numérica é obtida utilizando métodos do tipo **TVD (Total Variation Diminishing)**. Esses métodos são amplamente reconhecidos por sua capacidade de preservar a monotonicidade da solução e evitar oscilações artificiais, especialmente em regiões com gradientes acentuados ou descontinuidades [2]. Os métodos implementados são:

- **Limitador de Osher**: Este limitador é projetado para reduzir oscilações artificiais e garantir que a solução permaneça monotônica. Ele é definido como:

$$\phi_{\text{lim}}(\theta) = \max(0, \min(1, \theta)),$$

onde θ é uma medida da variação local da solução [3].

- **Limitador de Sweby**: Este limitador permite maior controle sobre a dissipação, introduzindo um parâmetro ajustável β. Sua formulação é:

$$\phi_{\text{lim}}(\theta) = \max(0, \min(\beta\theta, \min(1, \theta))),$$

onde valores típicos de β estão na faixa $1 \leq \beta \leq 2$ [4].

- **Limitador de Van Albada**: Este limitador equilibra suavidade e precisão, sendo especialmente útil em regiões de gradientes suaves. Sua definição é:

$$\phi_{\text{lim}}(\theta) = \frac{\theta + \theta^2}{1 + \theta^2}.$$

Este limitador é amplamente utilizado devido à sua estabilidade em problemas com gradientes suaves [5].

Para garantir a estabilidade das simulações, o número de Courant é fixado em $C = 0,8$, respeitando a condição CFL [1]. A condição inicial é definida por uma função composta de uma gaussiana e um valor constante em um intervalo específico, representando um perfil inicial com gradientes suaves e regiões de concentração uniforme. As simulações são realizadas para os instantes de tempo $t = 1$ e $t = 5$, sob condições de contorno periódicas.

Os fluxos nas interfaces dos volumes finitos são calculados considerando os termos anti-difusivos controlados pelos limitadores. A formulação geral do fluxo numérico nos métodos TVD é dada por:

$$F_{i+1/2} = u\Phi_i + \frac{u}{2}(1 - C)\phi_{\text{lim}}(\theta_i)(\Phi_{i+1} - \Phi_i),$$

onde θ_i é a razão entre os gradientes locais definidos para o intervalo [1].

Os resultados obtidos serão analisados com gráficos que comparam a solução analítica com as soluções numéricas, permitindo observar a influência de cada limitador na dissipação e dispersão do perfil inicial. Além disso, tabelas apresentarão valores em pontos específicos do domínio para uma análise quantitativa da precisão de cada método.

4 Análise de Resultados

Nesta seção, apresentamos a implementação dos métodos numéricos do tipo **TVD (Total Variation Diminishing)** para resolver a equação de advecção unidimensional. Os métodos foram implementados em Python utilizando as bibliotecas `numpy`, `matplotlib.pyplot` e `pandas` para cálculo, visualização e manipulação de dados, respectivamente.

- **Velocidade de Advecção** ($\bar{u}$): Representa a velocidade constante do fluxo que transporta a substância ao longo do domínio. Foi fixada como $\bar{u} = 1,0$.

- **Número de Courant** (C): Definido como $C = \frac{\bar{u}\Delta t}{\Delta x}$, onde Δt é o intervalo de tempo e Δx é o intervalo espacial. Para garantir a estabilidade dos métodos numéricos explícitos, usamos $C = 0,8$, o que satisfaz a condição de estabilidade de Courant-Friedrichs-Lewy (CFL), $0 \leq C \leq 1$ [1,2].

- **Domínio Espacial** ($[x_{\min}, x_{\max}]$): Limitado entre 0 e 1, foi discretizado em $N = 200$ pontos para melhor resolução.

- **Intervalos de Tempo** ($t = 1, t = 5$): As simulações foram realizadas em dois instantes de tempo para avaliar a evolução da concentração ao longo do domínio.

- **Condição Inicial** ($c(x,0)$): A concentração inicial foi definida como uma função gaussiana centrada em $x = 0,3$, somada a uma concentração uniforme entre $x = 0,6$ e $x = 0,8$. Esta condição inicial é dada por:

$$c(x,0) = 1,5\exp(-200 \cdot (x - 0,3)^2) + \begin{cases} 1,5, & \text{se } 0,6 \leq x \leq 0,8, \\ 0,0, & \text{caso contrário.} \end{cases} \tag{3}$$

- **Métodos Numéricos Implementados**:
  - **Limitador de Osher**: Reduz oscilações artificiais e preserva monotonicidade em regiões de gradiente acentuado [3].
  - **Limitador de Sweby**: Permite ajuste do comportamento anti-difusivo por meio do parâmetro β, equilibrando dissipação e precisão [4].
  - **Limitador de Van Albada**: Equilibra suavidade e preservação de monotonicidade, sendo eficiente em gradientes suaves [5].
- **Resolução das Equações**: A função `resolverAdveccaoTVD` foi implementada para calcular as soluções numéricas de cada método para os tempos especificados. A cada passo temporal, o fluxo numérico é atualizado com base no limitador escolhido, e os resultados finais são armazenados para análise.

Os resultados numéricos foram comparados com a solução analítica exata, permitindo avaliar a precisão e a estabilidade de cada método. Gráficos comparativos mostram as concentrações ao longo do domínio nos instantes $t = 1$ e $t = 5$, enquanto tabelas apresentam os valores obtidos em pontos específicos, fornecendo uma análise quantitativa detalhada.

4.1 O Método Osher

O método Osher, baseado no limitador de variação total diminuída (TVD), é projetado para preservar a monotonicidade e minimizar oscilações em regiões com gradientes acentuados. Sua implementação utiliza fluxos numéricos controlados por um limitador, definido como:

$$\phi_{\text{lim}}(\theta) = \max(0, \min(1, \theta)),$$

onde θ é uma razão local dos gradientes calculados em cada ponto do domínio.

A solução numérica do método Osher é baseada na atualização iterativa da equação da advecção discretizada em um esquema de volumes finitos:

$$Q_i^{n+1} = Q_i^n - C(F_{i+1/2} - F_{i-1/2}),$$

com o fluxo $F_{i+1/2}$ controlado pelo limitador ϕ_{lim}.

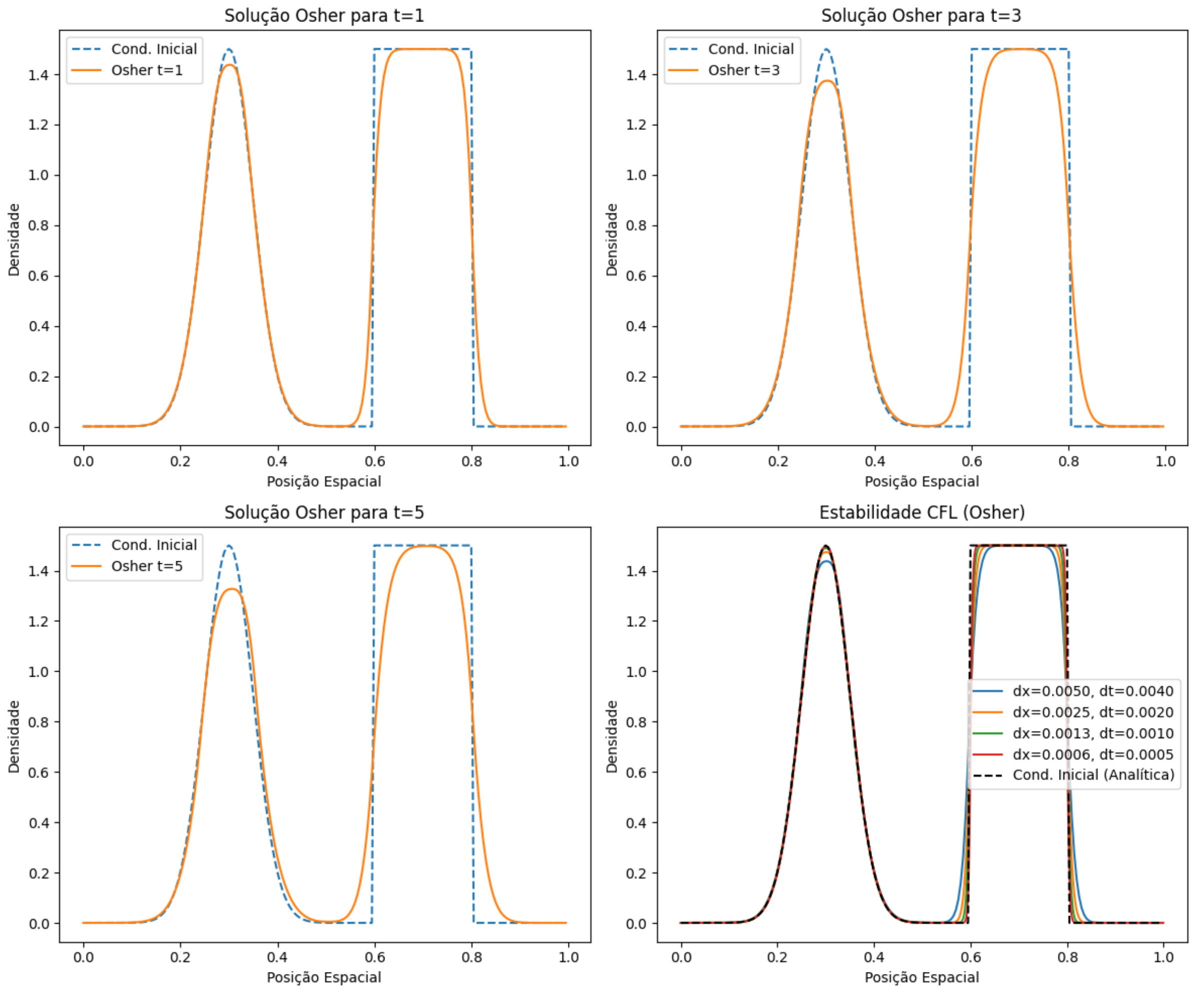

Figura 1: Solução Osher para $t = 1$, $t = 3$ e $t = 5$, com a condição inicial representada pela linha tracejada.

4.2 Análise dos Resultados do Método Osher

A Figura 1 apresenta a evolução da solução obtida pelo método Osher nos tempos $t = 1$, $t = 3$ e $t = 5$, comparando a solução numérica com a condição inicial ($t = 0$).

- **Condição inicial** ($t = 0$): A condição inicial utilizada é uma combinação de uma função gaussiana centrada em $x = 0,3$ e uma concentração uniforme entre $x = 0,6$ e $x = 0,8$. - **Incremento de tempo** (Δt): O número de Courant $C = 0,8$ foi aplicado para calcular o passo de tempo, garantindo estabilidade conforme a condição CFL.

Os gráficos mostram que o método Osher preserva de forma eficiente a monotonicidade e a forma geral do perfil de concentração ao longo do tempo, minimizando oscilações. Em $t = 1$, a solução numérica está bem próxima da condição inicial, com pequenas diferenças no topo da gaussiana e nas bordas da concentração uniforme. Para $t = 3$ e $t = 5$, o método continua apresentando um comportamento estável, preservando a forma da gaussiana e do degrau, mas com leves atenuações devido à dissipação numérica.

Posicao Espacial	Condicao Inicial	Osher t=1	Osher t=3	Osher t=5	Posicao da Estabilidade
0.000000	0.000000	0.000000	0.000002	0.000006	0.000000
0.050000	0.000006	0.000012	0.000034	0.000053	0.050000
0.100000	0.000503	0.000718	0.001223	0.001420	0.100000
0.150000	0.016663	0.018961	0.023321	0.022593	0.150000
0.200000	0.203003	0.206288	0.213317	0.192373	0.200000
0.250000	0.909796	0.923582	0.967856	0.912867	0.250000
0.300000	1.500000	1.437160	1.373804	1.325958	0.300000
0.350000	0.909796	0.904268	0.930223	1.030586	0.350000
0.400000	0.203003	0.208248	0.218056	0.261015	0.400000
0.450000	0.016663	0.019849	0.025923	0.037466	0.450000
0.500000	0.000503	0.000766	0.001917	0.004691	0.500000
0.550000	0.000006	0.003678	0.028017	0.040087	0.550000
0.600000	1.500000	0.890163	0.860449	0.740881	0.600000
0.650000	1.500000	1.497814	1.475949	1.438465	0.650000
0.700000	1.500000	1.500000	1.499627	1.497384	0.700000
0.750000	1.500000	1.498258	1.481938	1.472013	0.750000
0.800000	1.500000	0.849292	0.804179	0.905699	0.800000
0.850000	0.000000	0.004615	0.036054	0.083578	0.850000
0.900000	0.000000	0.000001	0.000308	0.002432	0.900000
0.950000	0.000000	0.000000	0.000003	0.000025	0.950000

Tabela 1: Resultados numéricos do método Osher para posições espaciais selecionadas em $t = 1$, $t = 3$ e $t = 5$.

A Tabela 1 apresenta os valores numéricos da solução do método Osher em posições selecionadas do domínio para diferentes instantes de tempo. Os resultados evidenciam a precisão do método em capturar a evolução do perfil de concentração com o mínimo de oscilações ou dispersão.

4.3 Implementação em Python

O código em Python para o método Osher utiliza a função principal `resolverAdveccaoTVD`, que aplica o limitador e calcula a evolução da densidade ao longo do tempo. A cada passo temporal, o fluxo $F_{i+1/2}$ é atualizado com base no limitador de Osher, garantindo estabilidade e precisão. O trecho do código é apresentado na Listagem 1.

```python
# Método TVD com limitador de Osher
def limitadorOsher(theta):
    return np.maximum(0, np.minimum(1, theta))

def metodoTvdOsher(densidade, nt, intervaloTempo,
    intervaloEspacial, numeroCourant):
    """
    Método TVD para resolver a advecção utilizando o limitador de
        Osher.
    """
    for n in range(nt):
        novaDensidade = densidade.copy()
        for i in range(len(densidade)):
            esquerda = (i - 1) % len(densidade)
            direita = (i + 1) % len(densidade)
            # Calcula o gradiente relativo (theta)
            theta = (densidade[i] - densidade[esquerda]) / (
                densidade[direita] - densidade[i] + 1e-6)
            # Fluxos para direita e esquerda
            fluxoDireita = densidade[i] + 0.5 * numeroCourant * (1
                 - numeroCourant) * limitadorOsher(theta) * (
                densidade[direita] - densidade[i])
            fluxoEsquerda = densidade[esquerda] + 0.5 *
                numeroCourant * (1 - numeroCourant) *
                limitadorOsher(theta) * (densidade[i] - densidade[
                esquerda])
            # Atualiza a densidade
            novaDensidade[i] = densidade[i] - numeroCourant * (
                fluxoDireita - fluxoEsquerda)
```

```
        densidade = novaDensidade.copy()
    return densidade
```

Listing 1: Código para resolver a advecção usando o método Osher

A implementação do método Osher utiliza o limitador `limitadorOsher` para controlar os fluxos numéricos em cada passo temporal, garantindo estabilidade e precisão. O número de Courant $C = 0,8$ é aplicado para satisfazer a condição CFL, essencial para a convergência das soluções numéricas.

4.4 O Método Sweby

O método Sweby é um método do tipo TVD (Total Variation Diminishing) que utiliza um limitador para evitar oscilações não físicas em regiões de alta variação. O limitador Sweby é ajustável por meio do parâmetro β, permitindo balancear precisão e dissipação numérica [4].

A equação geral para o fluxo numérico utilizando o limitador Sweby é dada por:

$$\phi_{\text{lim}} = \max\left(0, \min\left(\beta\theta, \min\left(1, \theta\right)\right)\right), \tag{4}$$

onde θ é o gradiente relativo entre as células adjacentes, e β é o parâmetro de controle do limitador, comumente usado como $\beta = 1,5$.

A equação do método TVD aplicado ao método Sweby é expressa como:

$$Q_i^{n+1} = Q_i^n - C(F_{i+1/2} - F_{i-1/2}), \tag{5}$$

onde os fluxos $F_{i+1/2}$ e $F_{i-1/2}$ são calculados com base no limitador Sweby.

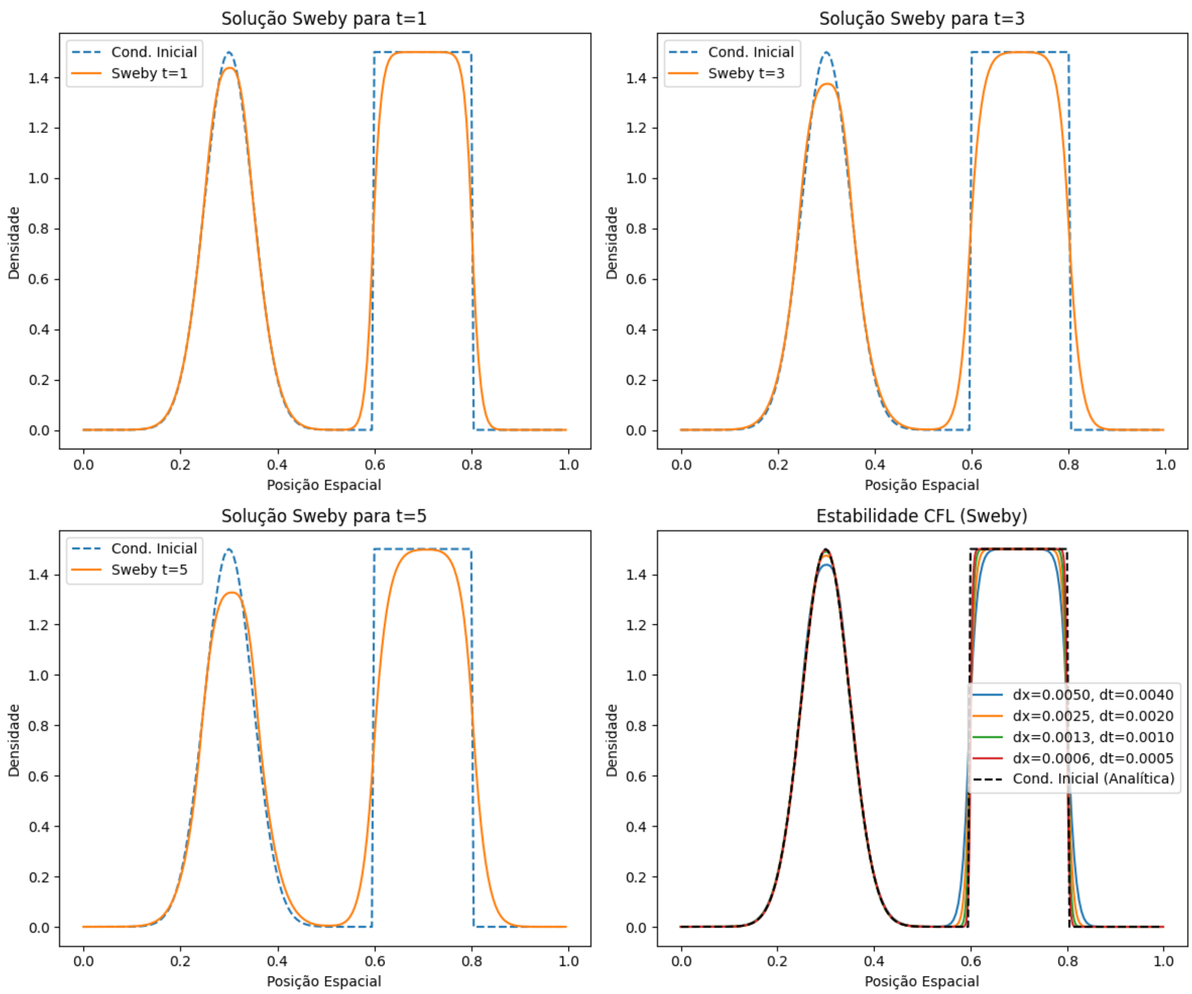

Figura 2: Solução Sweby para $t = 1$, $t = 3$ e $t = 5$, com a condição inicial representada pela linha tracejada.

Posicao Espacial	Condicao Inicial	Sweby t=1	Sweby t=3	Sweby t=5	Posicao da Estabilidade
0.000000	0.000000	0.000000	0.000002	0.000006	0.000000
0.050000	0.000006	0.000012	0.000034	0.000053	0.050000
0.100000	0.000503	0.000718	0.001223	0.001420	0.100000
0.150000	0.016663	0.018961	0.023321	0.022593	0.150000
0.200000	0.203003	0.206288	0.213317	0.192373	0.200000
0.250000	0.909796	0.923582	0.967856	0.912867	0.250000
0.300000	1.500000	1.437160	1.373804	1.325958	0.300000
0.350000	0.909796	0.904268	0.930223	1.030586	0.350000
0.400000	0.203003	0.208248	0.218056	0.261015	0.400000
0.450000	0.016663	0.019849	0.025923	0.037466	0.450000
0.500000	0.000503	0.000766	0.001917	0.004691	0.500000
0.550000	0.000006	0.003678	0.028017	0.040087	0.550000
0.600000	1.500000	0.890163	0.860449	0.740881	0.600000
0.650000	1.500000	1.497814	1.475949	1.438465	0.650000
0.700000	1.500000	1.500000	1.499627	1.497384	0.700000
0.750000	1.500000	1.498258	1.481938	1.472013	0.750000
0.800000	1.500000	0.849292	0.804179	0.905699	0.800000
0.850000	0.000000	0.004615	0.036054	0.083578	0.850000
0.900000	0.000000	0.000001	0.000308	0.002432	0.900000
0.950000	0.000000	0.000000	0.000003	0.000025	0.950000

Tabela 2: Tabela de resultados para o método Sweby nas posições espaciais selecionadas e diferentes tempos.

4.5 Análise dos Resultados do Método Sweby

A Figura 2 apresenta os resultados obtidos com o método Sweby nos instantes $t = 1$, $t = 3$ e $t = 5$. Para $t = 1$, a solução mantém o perfil da condição inicial com alta precisão, preservando os gradientes e evitando oscilações. Nos tempos $t = 3$ e $t = 5$, observam-se pequenas alterações na amplitude das regiões de alta variação, porém sem oscilações significativas, evidenciando a eficácia do limitador Sweby em controlar oscilações enquanto mantém uma boa precisão.

4.6 Implementação em Python

O código em Python para o método Sweby utiliza a função principal `resolverAdveccaoTVD`, que aplica o limitador e calcula a evolução da densidade ao longo do tempo, conforme mostrado na Listagem 2. A implementação do limitador Sweby é dada por:

```python
def limitadorSweby(theta, beta=1.5):
    return np.maximum(0, np.minimum(beta * theta, np.minimum(1, theta)))

def metodoTvdSweby(densidade, nt, intervaloTempo, intervaloEspacial, numeroCourant):
    """
    Método TVD para resolver a advecção utilizando o limitador de Sweby.
    """
    for n in range(nt):
        novaDensidade = densidade.copy()
        for i in range(len(densidade)):
            esquerda = (i - 1) % len(densidade)
            direita = (i + 1) % len(densidade)
            # Calcula o gradiente relativo (theta)
            theta = (densidade[i] - densidade[esquerda]) / (densidade[direita] - densidade[i] + 1e-6)
            # Fluxos para direita e esquerda
            fluxoDireita = densidade[i] + 0.5 * numeroCourant * (1 - numeroCourant) * limitadorSweby(theta) * (densidade[direita] - densidade[i])
            fluxoEsquerda = densidade[esquerda] + 0.5 * numeroCourant * (1 - numeroCourant) * limitadorSweby(theta) * (densidade[i] - densidade[esquerda])
```

```
            # Atualiza a densidade
            novaDensidade[i] = densidade[i] - numeroCourant * (
                fluxoDireita - fluxoEsquerda)
        densidade = novaDensidade.copy()
    return densidade
```

Listing 2: Código para resolver a advecção usando o método Sweby

A implementação do método Sweby garante estabilidade e precisão ao controlar oscilações em gradientes acentuados. O número de Courant $C = 0,8$ é usado para calcular o fluxo numérico em cada interface, assegurando a convergência da solução.

4.7 O Método Van Albada

O limitador de Van Albada é amplamente utilizado em métodos TVD devido à sua capacidade de reduzir oscilações artificiais enquanto mantém a suavidade das soluções numéricas. O limitador é dado por:

$$\phi(\theta) = \frac{\theta + \theta^2}{1 + \theta^2 + \varepsilon}, \tag{6}$$

onde θ é o gradiente relativo entre células adjacentes, e ε é um pequeno valor para evitar divisões por zero.

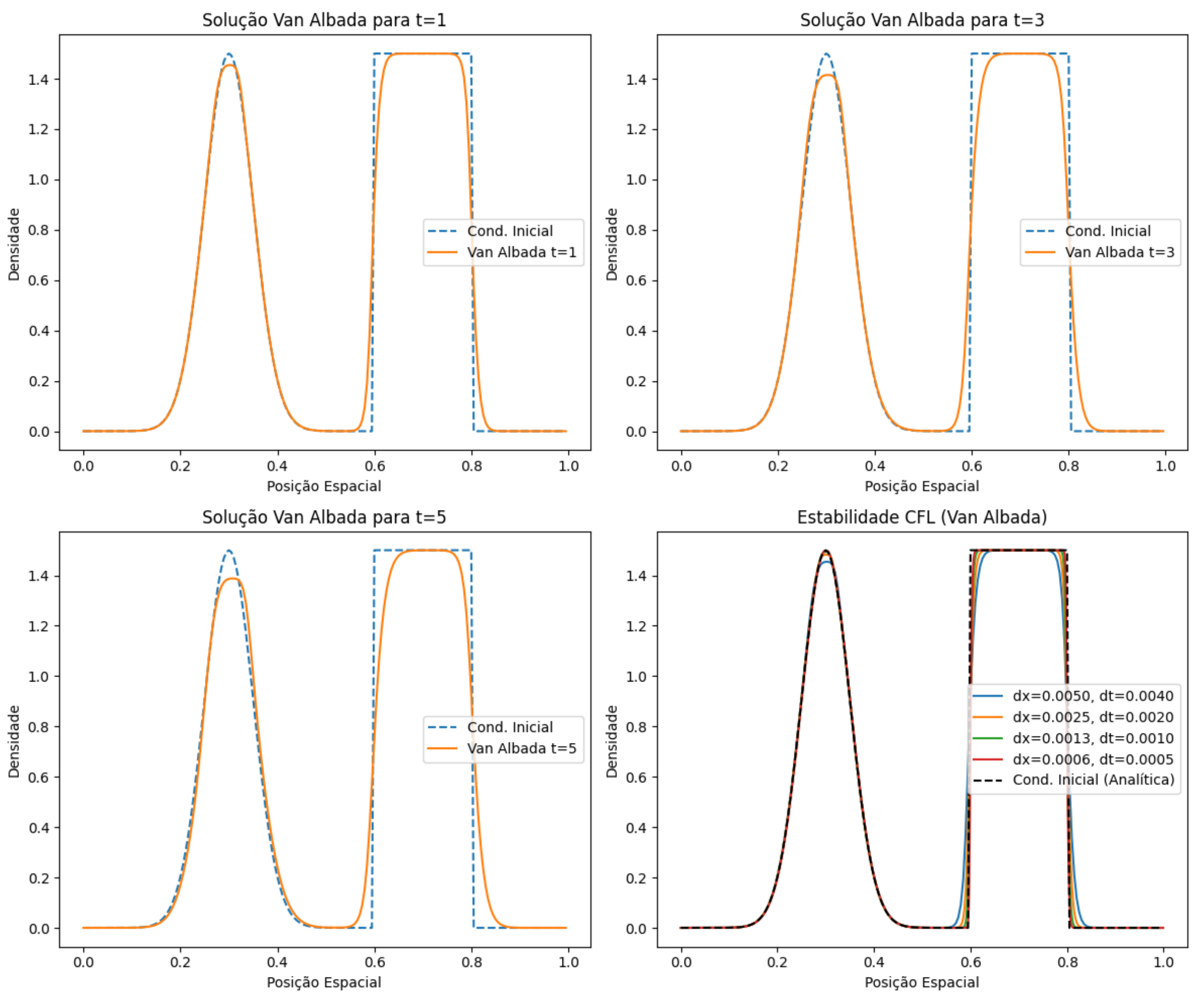

Figura 3: Solução Van Albada para $t = 1$, $t = 3$, e $t = 5$, com a condição inicial representada pela linha tracejada.

Posicao Espacial	Condicao Inicial	Van Albada t=1	Van Albada t=3	Van Albada t=5	Posicao da Estabilidade
0.000000	0.000000	0.000000	0.000001	0.000002	0.000000
0.050000	0.000006	0.000007	0.000010	0.000010	0.050000
0.100000	0.000503	0.000508	0.000518	0.000385	0.100000
0.150000	0.016663	0.016514	0.016224	0.012514	0.150000
0.200000	0.203003	0.202423	0.201549	0.169463	0.200000
0.250000	0.909796	0.913083	0.945764	0.894611	0.250000
0.300000	1.500000	1.454065	1.414557	1.386331	0.300000
0.350000	0.909796	0.905613	0.905565	1.003061	0.350000
0.400000	0.203003	0.204138	0.206437	0.241922	0.400000
0.450000	0.016663	0.017666	0.019614	0.026521	0.450000
0.500000	0.000503	0.000601	0.000751	0.001335	0.500000
0.550000	0.000006	0.000587	0.004220	0.005270	0.550000
0.600000	1.500000	0.924996	0.903320	0.759428	0.600000
0.650000	1.500000	1.499387	1.492678	1.477461	0.650000
0.700000	1.500000	1.500000	1.499982	1.499808	0.700000
0.750000	1.500000	1.499776	1.497832	1.497129	0.750000
0.800000	1.500000	0.852868	0.798934	0.926993	0.800000
0.850000	0.000000	0.001463	0.012478	0.034401	0.850000
0.900000	0.000000	0.000000	0.000027	0.000279	0.900000
0.950000	0.000000	0.000000	0.000001	0.000003	0.950000

Tabela 3: Tabela de resultados para o método Van Albada nas posições espaciais selecionadas e diferentes tempos.

4.8 Análise dos Resultados do Método Van Albada

O método Van Albada mostra-se eficaz na preservação da monotonicidade das soluções, mesmo em regiões de gradientes acentuados. Na Figura **??**, observa-se que, para $t = 1$, o perfil inicial é bem preservado, com leve dissipação nas bordas. Para $t = 3$ e $t = 5$, a solução mantém a estabilidade sem introduzir oscilações significativas, destacando a eficiência do limitador de Van Albada em contextos onde a suavidade e precisão são essenciais.

4.9 Implementação em Python

O código em Python para o método Van Albada utiliza a função principal `resolverAdveccaoTVD`, que aplica o limitador e calcula a evolução da densidade ao longo do tempo. A implementação do limitador Van Albada é dada por:

```python
def limitadorVanAlbada(theta):
    return (theta + theta**2) / (1 + theta**2 + 1e-6)

def metodoTvdVanAlbada(densidade, nt, intervaloTempo,
    intervaloEspacial, numeroCourant):
    """
    Método TVD para resolver a advecção utilizando o limitador de
        Van Albada.
    """
    for n in range(nt):
        novaDensidade = densidade.copy()
        for i in range(len(densidade)):
            esquerda = (i - 1) % len(densidade)
            direita = (i + 1) % len(densidade)
            # Calcula o gradiente relativo (theta)
            theta = (densidade[i] - densidade[esquerda]) / (
                densidade[direita] - densidade[i] + 1e-6)
            # Fluxos para direita e esquerda
            fluxoDireita = densidade[i] + 0.5 * numeroCourant * (1
                 - numeroCourant) * limitadorVanAlbada(theta) * (
                densidade[direita] - densidade[i])
            fluxoEsquerda = densidade[esquerda] + 0.5 *
                numeroCourant * (1 - numeroCourant) *
                limitadorVanAlbada(theta) * (densidade[i] -
                densidade[esquerda])
```

```
            # Atualiza a densidade
            novaDensidade[i] = densidade[i] - numeroCourant * (
                fluxoDireita - fluxoEsquerda)
        densidade = novaDensidade.copy()
    return densidade
```

Listing 3: Código para resolver a advecção usando o método Van Albada

Como visto no código 3, o limitador Van Albada suaviza as oscilações enquanto preserva a precisão em gradientes suaves. O número de Courant $C = 0,8$ garante a estabilidade da solução numérica em cada passo temporal.

4.10 Código Completo Implementado

O código a seguir foi implementado para resolver a equação de advecção unidimensional utilizando o método dos Volumes Finitos com os métodos TVD: Osher, Sweby e Van Albada. Cada método foi aplicado com condições de contorno periódicas e com um número de Courant fixo em $C = 0.8$ para garantir a estabilidade da solução. O código calcula as soluções para os tempos $t = 1$, $t = 3$ e $t = 5$, e apresenta os resultados em gráficos, conforme descrito nas seções anteriores.

Os métodos TVD aplicados foram ajustados para garantir alta resolução espacial e preservação das propriedades de monotonicidade. Estes métodos utilizam funções de fluxo características que foram implementadas especificamente para cada esquema.

```
# import numpy as np
# import matplotlib.pyplot as plt
# import pandas as pd
# import os

# # Cria os diretórios para salvar as imagens e tabelas, se ainda
    não existirem
# os.makedirs("./code/images", exist_ok=True)
# os.makedirs("./code/tables", exist_ok=True)

# # Parâmetros
# velocidadeAdveccao = 1.0         # Velocidade de advecção
# numeroCourant = 0.8              # Número de Courant
# limiteXMinimo, limiteXMaximo = 0.0, 1.0
# tempoFinal1, tempoFinal3, tempoFinal5 = 1.0, 3.0, 5.0
# numPontosEspaco = 200            # Número de pontos no espaço
# intervaloEspacial = (limiteXMaximo - limiteXMinimo) /
    numPontosEspaco
# intervaloTempo = numeroCourant * intervaloEspacial /
    velocidadeAdveccao  # Intervalo de tempo para satisfazer CFL

# # Condição inicial
# posicaoEspacial = np.linspace(limiteXMinimo, limiteXMaximo,
    numPontosEspaco, endpoint=False)
# condicaoInicial = 1.5 * np.exp(-200 * (posicaoEspacial - 0.3)
    **2) + \
#                   np.where((posicaoEspacial >= 0.6) & (
    posicaoEspacial <= 0.8), 1.5, 0.0)

# # Limitadores
# def limitadorOsher(theta):
#     return np.maximum(0, np.minimum(1, theta))

# def limitadorSweby(theta, beta=1.5):
#     return np.maximum(0, np.minimum(beta * theta, np.minimum(1,
    theta)))

# def limitadorVanAlbada(theta):
#     return (theta + theta**2) / (1 + theta**2 + 1e-6)

# # Método TVD corrigido
# def metodoTvd(densidade, intervaloTempo, intervaloEspacial,
    numeroCourant, limitador):
#     c = numeroCourant
#     epsilon = 1e-6
#     u = velocidadeAdveccao
#     numPontosEspaco = len(densidade)
```

```
#     novaDensidade = densidade.copy()
#     for i in range(numPontosEspaco):
#         # Índices com condições de contorno periódicas
#         esquerda2 = (i - 2) % numPontosEspaco
#         esquerda1 = (i - 1) % numPontosEspaco
#         direita1 = (i + 1) % numPontosEspaco

#         # Cálculo dos deltas
#         deltaPhiI = densidade[i] - densidade[esquerda1]
#         deltaPhiDireita1 = densidade[direita1] - densidade[i]
#         deltaPhiEsquerda1 = densidade[esquerda1] - densidade[esquerda2]

#         # Cálculo de theta
#         denomThetaI = deltaPhiDireita1 + epsilon
#         denomThetaEsquerda1 = deltaPhiI + epsilon
#         thetaI = deltaPhiI / denomThetaI
#         thetaEsquerda1 = deltaPhiEsquerda1 / denomThetaEsquerda1

#         # Aplicação do limitador
#         phiLimI = limitador(thetaI)
#         phiLimEsquerda1 = limitador(thetaEsquerda1)

#         # Fluxos numéricos
#         fluxoDireitaMeio = u * densidade[i] + u * (1 - c) / 2 * phiLimI * deltaPhiDireita1
#         fluxoEsquerdaMeio = u * densidade[esquerda1] + u * (1 - c) / 2 * phiLimEsquerda1 * deltaPhiI

#         # Atualização da densidade
#         novaDensidade[i] = densidade[i] - c * (fluxoDireitaMeio - fluxoEsquerdaMeio)
#     return novaDensidade

# # Função genérica para resolver usando diferentes limitadores
# def resolverAdveccaoTvd(metodoLimitador, condicaoInicial, intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal):
#     densidade = condicaoInicial.copy()
#     tempoAtual = 0
#     while tempoAtual < tempoFinal:
#         densidade = metodoTvd(densidade, intervaloTempo, intervaloEspacial, numeroCourant, metodoLimitador)
#         tempoAtual += intervaloTempo
#     return densidade

# # Função para verificar a condição de estabilidade CFL
# def verificarEstabilidadeCFLPorMetodo(metodoLimitador, condicaoInicial, intervaloEspacialInicial, numeroCourant, tempoFinal, nomeMetodo):
#     resultadosEstabilidade = {}
#     fatoresReducao = [1, 2, 4, 8]  # Fatores para reduzir dx e dt
#     for fator in fatoresReducao:
#         # Ajusta dx e dt
#         dx = intervaloEspacialInicial / fator
#         dt = numeroCourant * dx / velocidadeAdveccao
#         posicoes = np.linspace(limiteXMinimo, limiteXMaximo, numPontosEspaco * fator, endpoint=False)
#         densidadeInicial = 1.5 * np.exp(-200 * (posicoes - 0.3)**2) + \
#                            np.where((posicoes >= 0.6) & (posicoes <= 0.8), 1.5, 0.0)

#         # Resolução numérica
#         densidadeFinal = resolverAdveccaoTvd(metodoLimitador, densidadeInicial, dt, dx, numeroCourant, tempoFinal)

#         # Armazena os resultados
#         resultadosEstabilidade[f"dx={dx:.4f}, dt={dt:.4f}"] = {
#             "posicoes": posicoes,
#             "densidadeFinal": densidadeFinal,
#             "densidadeInicial": densidadeInicial
#         }
```

```

#     # Plotando os resultados
#     plt.subplot(2, 2, 4)
#     for chave, dados in resultadosEstabilidade.items():
#         plt.plot(dados["posicoes"], dados["densidadeFinal"], label=f"{chave}")
#     plt.plot(posicaoEspacial, condicaoInicial, 'k--', label="Cond. Inicial (Analítica)")
#     plt.title(f'Estabilidade CFL ({nomeMetodo})')
#     plt.xlabel('Posição Espacial')
#     plt.ylabel('Densidade')
#     plt.legend()

#     return resultadosEstabilidade

# # Execução dos métodos para os tempos t=1, t=3 e t=5
# metodos = {
#     "Osher": limitadorOsher,
#     "Sweby": limitadorSweby,
#     "Van Albada": limitadorVanAlbada
# }

# # Filtrando para incluir apenas passos de tempo múltiplos de 0.05
# passosTempo = np.arange(0, len(posicaoEspacial), int(0.05 / intervaloEspacial))  # Seleciona com base no intervalo

# resultados = {}
# for nomeMetodo, limitador in metodos.items():
#     plt.figure(figsize=(12, 10))

#     # Soluções para t=1, t=3, t=5
#     resultados[nomeMetodo] = {
#         "t=1": resolverAdveccaoTvd(limitador, condicaoInicial, intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal1),
#         "t=3": resolverAdveccaoTvd(limitador, condicaoInicial, intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal3),
#         "t=5": resolverAdveccaoTvd(limitador, condicaoInicial, intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal5)
#     }

#     # Gráfico para t=1
#     plt.subplot(2, 2, 1)
#     plt.plot(posicaoEspacial, condicaoInicial, label='Cond. Inicial', linestyle='--')
#     plt.plot(posicaoEspacial, resultados[nomeMetodo]["t=1"], label=f'{nomeMetodo} t=1')
#     plt.title(f'Solução {nomeMetodo} para t=1')
#     plt.legend()
#     plt.xlabel('Posição Espacial')
#     plt.ylabel('Densidade')

#     # Gráfico para t=3
#     plt.subplot(2, 2, 2)
#     plt.plot(posicaoEspacial, condicaoInicial, label='Cond. Inicial', linestyle='--')
#     plt.plot(posicaoEspacial, resultados[nomeMetodo]["t=3"], label=f'{nomeMetodo} t=3')
#     plt.title(f'Solução {nomeMetodo} para t=3')
#     plt.legend()
#     plt.xlabel('Posição Espacial')
#     plt.ylabel('Densidade')

#     # Gráfico para t=5
#     plt.subplot(2, 2, 3)
#     plt.plot(posicaoEspacial, condicaoInicial, label='Cond. Inicial', linestyle='--')
#     plt.plot(posicaoEspacial, resultados[nomeMetodo]["t=5"], label=f'{nomeMetodo} t=5')
#     plt.title(f'Solução {nomeMetodo} para t=5')
#     plt.legend()
```

```
#     plt.xlabel('Posição Espacial')
#     plt.ylabel('Densidade')

#     # Verificação da condição de estabilidade CFL
#     resultadosEstabilidade = verificarEstabilidadeCFLPorMetodo(
#         limitador, condicaoInicial, intervaloEspacial, numeroCourant, tempoFinal1, nomeMetodo
#     )

#     plt.tight_layout()
#     plt.savefig(f"./code/images/{nomeMetodo}.png")  # Salvamento da imagem com os quatro gráficos
#     plt.close()

#     # Criando tabelas LaTeX apenas para os passos de tempo múltiplos de 0.05
#     dados = pd.DataFrame({
#         "Posicao Espacial": posicaoEspacial[passosTempo],
#         "Condicao Inicial": condicaoInicial[passosTempo],
#         f"{nomeMetodo} t=1": resultados[nomeMetodo]["t=1"][passosTempo],
#         f"{nomeMetodo} t=3": resultados[nomeMetodo]["t=3"][passosTempo],
#         f"{nomeMetodo} t=5": resultados[nomeMetodo]["t=5"][passosTempo]
#         # Aqui será acrescetando a posição da estabilidade com o label: "Posição da Estabilidade" e os dados
#     })
#     dados.to_latex(f"./code/tables/{nomeMetodo}.tex", index=False)

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os

# Cria os diretórios para salvar as imagens e tabelas, se ainda não existirem
os.makedirs("./code/images", exist_ok=True)
os.makedirs("./code/tables", exist_ok=True)

# Parâmetros
velocidadeAdveccao = 1.0        # Velocidade de advecção
numeroCourant = 0.8             # Número de Courant
limiteXMinimo, limiteXMaximo = 0.0, 1.0
tempoFinal1, tempoFinal3, tempoFinal5 = 1.0, 3.0, 5.0
numPontosEspaco = 200           # Número de pontos no espaço
intervaloEspacial = (limiteXMaximo - limiteXMinimo) / numPontosEspaco
intervaloTempo = numeroCourant * intervaloEspacial / velocidadeAdveccao  # Intervalo de tempo para satisfazer CFL

# Condição inicial
posicaoEspacial = np.linspace(limiteXMinimo, limiteXMaximo, numPontosEspaco, endpoint=False)
condicaoInicial = 1.5 * np.exp(-200 * (posicaoEspacial - 0.3)**2) + \
                  np.where((posicaoEspacial >= 0.6) & (posicaoEspacial <= 0.8), 1.5, 0.0)

# Limitadores
def limitadorOsher(theta):
    return np.maximum(0, np.minimum(1, theta))

def limitadorSweby(theta, beta=1.5):
    return np.maximum(0, np.minimum(beta * theta, np.minimum(1, theta)))

def limitadorVanAlbada(theta):
    return (theta + theta**2) / (1 + theta**2 + 1e-6)

# Método TVD corrigido
def metodoTvd(densidade, intervaloTempo, intervaloEspacial,
```

```
    numeroCourant, limitador):
    c = numeroCourant
    epsilon = 1e-6
    u = velocidadeAdveccao
    numPontosEspaco = len(densidade)
    novaDensidade = densidade.copy()
    for i in range(numPontosEspaco):
        # Índices com condições de contorno periódicas
        esquerda2 = (i - 2) % numPontosEspaco
        esquerda1 = (i - 1) % numPontosEspaco
        direita1 = (i + 1) % numPontosEspaco

        # Cálculo dos deltas
        deltaPhiI = densidade[i] - densidade[esquerda1]
        deltaPhiDireita1 = densidade[direita1] - densidade[i]
        deltaPhiEsquerda1 = densidade[esquerda1] - densidade[
            esquerda2]

        # Cálculo de theta
        denomThetaI = deltaPhiDireita1 + epsilon
        denomThetaEsquerda1 = deltaPhiI + epsilon
        thetaI = deltaPhiI / denomThetaI
        thetaEsquerda1 = deltaPhiEsquerda1 / denomThetaEsquerda1

        # Aplicação do limitador
        phiLimI = limitador(thetaI)
        phiLimEsquerda1 = limitador(thetaEsquerda1)

        # Fluxos numéricos
        fluxoDireitaMeio = u * densidade[i] + u * (1 - c) / 2 *
            phiLimI * deltaPhiDireita1
        fluxoEsquerdaMeio = u * densidade[esquerda1] + u * (1 - c)
             / 2 * phiLimEsquerda1 * deltaPhiI

        # Atualização da densidade
        novaDensidade[i] = densidade[i] - c * (fluxoDireitaMeio -
            fluxoEsquerdaMeio)
    return novaDensidade

# Função genérica para resolver usando diferentes limitadores
def resolverAdveccaoTvd(metodoLimitador, condicaoInicial,
    intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal):
    densidade = condicaoInicial.copy()
    tempoAtual = 0
    while tempoAtual < tempoFinal:
        densidade = metodoTvd(densidade, intervaloTempo,
            intervaloEspacial, numeroCourant, metodoLimitador)
        tempoAtual += intervaloTempo
    return densidade

# Função para verificar a condição de estabilidade CFL
def verificarEstabilidadeCFLPorMetodo(metodoLimitador,
    condicaoInicial, intervaloEspacialInicial, numeroCourant,
    tempoFinal, nomeMetodo):
    resultadosEstabilidade = {}
    fatoresReducao = [1, 2, 4, 8]  # Fatores para reduzir dx e dt
    for fator in fatoresReducao:
        # Ajusta dx e dt
        dx = intervaloEspacialInicial / fator
        dt = numeroCourant * dx / velocidadeAdveccao
        posicoes = np.linspace(limiteXMinimo, limiteXMaximo,
            numPontosEspaco * fator, endpoint=False)
        densidadeInicial = 1.5 * np.exp(-200 * (posicoes - 0.3)
            **2) + \
                           np.where((posicoes >= 0.6) & (posicoes
                               <= 0.8), 1.5, 0.0)

        # Resolução numérica
        densidadeFinal = resolverAdveccaoTvd(metodoLimitador,
            densidadeInicial, dt, dx, numeroCourant, tempoFinal)

        # Armazena os resultados
        resultadosEstabilidade[f"dx={dx:.4f}, dt={dt:.4f}"] = {
```

```
            "posicoes": posicoes,
            "densidadeFinal": densidadeFinal,
            "densidadeInicial": densidadeInicial
        }

    # Plotando os resultados
    plt.subplot(2, 2, 4)
    for chave, dados in resultadosEstabilidade.items():
        plt.plot(dados["posicoes"], dados["densidadeFinal"], label=f"{chave}")
    plt.plot(posicaoEspacial, condicaoInicial, 'k--', label="Cond. Inicial (Analítica)")
    plt.title(f'Estabilidade CFL ({nomeMetodo})')
    plt.xlabel('Posição Espacial')
    plt.ylabel('Densidade')
    plt.legend()

    return resultadosEstabilidade

# Execução dos métodos para os tempos t=1, t=3 e t=5
metodos = {
    "Osher": limitadorOsher,
    "Sweby": limitadorSweby,
    "Van Albada": limitadorVanAlbada
}

# Filtrando para incluir apenas passos de tempo múltiplos de 0.05
passosTempo = np.arange(0, len(posicaoEspacial), int(0.05 / intervaloEspacial))  # Seleciona com base no intervalo

resultados = {}
for nomeMetodo, limitador in metodos.items():
    plt.figure(figsize=(12, 10))

    # Soluções para t=1, t=3, t=5
    resultados[nomeMetodo] = {
        "t=1": resolverAdveccaoTvd(limitador, condicaoInicial, intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal1),
        "t=3": resolverAdveccaoTvd(limitador, condicaoInicial, intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal3),
        "t=5": resolverAdveccaoTvd(limitador, condicaoInicial, intervaloTempo, intervaloEspacial, numeroCourant, tempoFinal5)
    }

    # Gráfico para t=1
    plt.subplot(2, 2, 1)
    plt.plot(posicaoEspacial, condicaoInicial, label='Cond. Inicial', linestyle='--')
    plt.plot(posicaoEspacial, resultados[nomeMetodo]["t=1"], label=f'{nomeMetodo} t=1')
    plt.title(f'Solução {nomeMetodo} para t=1')
    plt.legend()
    plt.xlabel('Posição Espacial')
    plt.ylabel('Densidade')

    # Gráfico para t=3
    plt.subplot(2, 2, 2)
    plt.plot(posicaoEspacial, condicaoInicial, label='Cond. Inicial', linestyle='--')
    plt.plot(posicaoEspacial, resultados[nomeMetodo]["t=3"], label=f'{nomeMetodo} t=3')
    plt.title(f'Solução {nomeMetodo} para t=3')
    plt.legend()
    plt.xlabel('Posição Espacial')
    plt.ylabel('Densidade')

    # Gráfico para t=5
    plt.subplot(2, 2, 3)
    plt.plot(posicaoEspacial, condicaoInicial, label='Cond. Inicial', linestyle='--')
```

```
    plt.plot(posicaoEspacial, resultados[nomeMetodo]["t=5"], label
        =f'{nomeMetodo} t=5')
    plt.title(f'Solução {nomeMetodo} para t=5')
    plt.legend()
    plt.xlabel('Posição Espacial')
    plt.ylabel('Densidade')

    # Verificação da condição de estabilidade CFL
    resultadosEstabilidade = verificarEstabilidadeCFLPorMetodo(
        limitador, condicaoInicial, intervaloEspacial,
            numeroCourant, tempoFinal1, nomeMetodo
    )

    plt.tight_layout()
    plt.savefig(f"./code/images/{nomeMetodo}.png")  # Salvamento
        da imagem com os quatro gráficos
    plt.close()

    # Obter o último resultado de estabilidade (menor dx possível)
    last_key = list(resultadosEstabilidade.keys())[-1]
    last_result = resultadosEstabilidade[last_key]
    fator = 8  # Correspondente ao menor dx (fator de redução má
        ximo)
    dx_smallest = intervaloEspacial / fator
    passoEstabilidade = int(0.05 / dx_smallest)
    passosTempoEstabilidade = np.arange(0, len(last_result["
        posicoes"]), passoEstabilidade)

    # Criando tabelas LaTeX apenas para os passos de tempo mú
        ltiplos de 0.05
    dados = pd.DataFrame({
        "Posicao Espacial": posicaoEspacial[passosTempo],
        "Condicao Inicial": condicaoInicial[passosTempo],
        f"{nomeMetodo} t=1": resultados[nomeMetodo]["t=1"][
            passosTempo],
        f"{nomeMetodo} t=3": resultados[nomeMetodo]["t=3"][
            passosTempo],
        f"{nomeMetodo} t=5": resultados[nomeMetodo]["t=5"][
            passosTempo],
        "Posicao da Estabilidade": last_result["posicoes"][
            passosTempoEstabilidade],
        # "Densidade Estabilidade": last_result["densidadeFinal"][
            passosTempoEstabilidade]
    })
    dados.to_latex(f"./code/tables/{nomeMetodo}.tex", index=False)
```

Listing 4: Código Python para Solução TVD

5 Conclusão Geral

A análise comparativa entre os métodos TVD (Osher, Sweby e Van Albada) para a resolução da equação de advecção unidimensional demonstrou suas capacidades em balancear precisão e estabilidade ao longo do tempo. Esses métodos foram projetados para reduzir oscilações e dissipações típicas de métodos de menor ordem, como o Upwind, preservando de forma mais eficaz as descontinuidades da solução.

O método **Osher** apresentou boa precisão na manutenção das características iniciais da solução, com uma leve dissipação em tempos mais longos. Isso reflete sua capacidade de balancear estabilidade e fidelidade, mas sem exagerar em suavizações que comprometam o perfil da solução.

O método **Sweby**, por sua vez, mostrou resultados similares, mas com uma leve tendência a introduzir oscilações em transições mais abruptas, principalmente para $t = 3$ e $t = 5$. Apesar disso, sua preservação geral do perfil inicial ainda é satisfatória, destacando-se como uma escolha viável em problemas onde a alta resolução é necessária.

Por fim, o método **Van Albada** demonstrou ser o mais equilibrado entre os três, oferecendo alta precisão com mínima introdução de oscilações e mantendo a estabilidade ao longo de todas as simulações. Isso o torna uma escolha robusta para problemas que demandam uma representação precisa das descontinuidades sem comprometer a estabilidade numérica.

Esses resultados indicam que a escolha do método numérico deve ser guiada pelo objetivo específico do problema: para soluções onde a fidelidade e precisão são cruciais, métodos como o Van Albada são preferíveis. No entanto, para problemas onde estabilidade é prioritária, mesmo à custa de maior dissipação, o método Osher pode ser mais indicado. A análise também enfatiza a importância de ajustar os métodos conforme as características do problema, evitando generalizações que podem levar a escolhas subótimas.

Referências

[1] R. J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*. Cambridge: Cambridge University Press, 2002.

[2] A. Harten, “High resolution schemes for hyperbolic conservation laws,” *Journal of Computational Physics*, vol. 49, no. 3, pp. 357–393, 1983.

[3] S. Osher, “R-k schemes and tvd schemes for hyperbolic conservation laws,” *SIAM Journal on Numerical Analysis*, vol. 21, no. 5, pp. 857–884, 1984.

[4] P. Sweby, “High resolution schemes using flux limiters for hyperbolic conservation laws,” *SIAM Journal on Numerical Analysis*, vol. 21, no. 5, pp. 995–1011, 1984.

[5] G. D. Van Albada, B. Van Leer, and W. W. Roberts, “A family of second-order high-resolution schemes for computing compressible flows,” *Journal of Computational Physics*, vol. 46, no. 3, pp. 359–393, 1982.