

CC-299 - Métodos Numéricos de Alta Ordem

Projetos 1, 2 e 3

Gustavo Pires

Instituto Tecnológico de Aeronáutica

2º Semestre, 2017

Conteúdo

1 INTRODUÇÃO	2
1.1 Objetivos	2
1.2 O Problema do Tubo de Choque	2
1.3 Equações Governantes	3
2 SOLUÇÃO EXATA	4
3 ESQUEMAS NUMÉRICOS	7
3.1 Método de Marcha no Tempo	7
3.1.1 Esquema de Euler Explícito	7
3.1.2 Condição de CFL	7
3.2 Dissipação Artificial	8
3.2.1 Dissipação Artificial Linear	8
3.2.2 Dissipação Artificial Não-linear	11
3.3 Forma Conservativa dos Esquemas Numéricos	11
3.4 Efeito do Refinamento da Malha	13
3.5 Projeto 1	16
3.5.1 Esquema Centrado	16
3.5.2 Esquema de Lax-Wendroff	34
3.5.3 Esquema Explícito de MacCormack	40
3.6 Projeto 2	46
3.6.1 Esquema FVS de Steger & Warming	46
3.6.2 Esquema FVS de van Leer (non-MUSCL)	55
3.6.3 Esquema FVS de Liou (AUSM ⁺)	63
3.6.4 Método de Roe	68
3.7 Projeto 3	74
3.7.1 Esquema TVD de Harten	74
4 CONCLUSÕES	93
5 ANEXOS	95

1 INTRODUÇÃO

Nesta seção serão apresentados os objetivos do presente trabalho, bem como a metodologia seguida. Os objetivos do trabalho são apresentados, seguidos de uma breve descrição do problema proposto. O relatório é composto pelos resultados dos projetos 1, 2 e 3, e portanto, cada seção será destinada a um destes projetos, detalhando a metodologia seguida e a forma como foi implementado cada um dos métodos numéricos. Por fim, uma análise comparativa entre os métodos será apresentada.

1.1 Objetivos

O presente trabalho tem como objetivo obter solução numérica para o problema de Riemann, observado no experimento de tubo de choque, para diferentes esquemas numéricos, levando em consideração diferentes razões de pressão como condições iniciais. Uma análise comparativa será apresentada, a fim de se obter um estudo sobre a qualidade e robustez de cada método numérico implementado, bem como a eficiência e custo computacional.

1.2 O Problema do Tubo de Choque

O tubo de choque é um problema bem conhecido, onde um tubo contendo um dado fluido é exposto inicialmente a duas regiões com uma dada razão de pressão, separadas apenas por uma membrana (diafragma) como mostra a Fig. 1. Ao romper-se a membrana, uma onda de choque propaga-se pelo fluido, causando mudança nos estados do fluido ao longo do tubo. A simplicidade deste experimento facilita a validação de soluções numéricas para este problema. Devido a este fato, diversos métodos numéricos são aplicados e validados utilizando este problema como base.

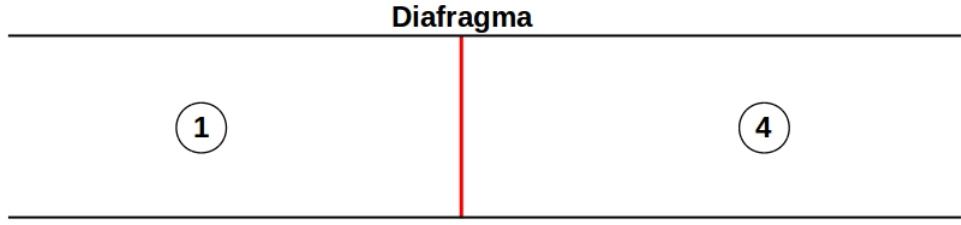


Figura 1: Condição inicial do tubo de choque

1.3 Equações Governantes

O problema apresentado pode ser modelado através das equações de Euler, descritas para o caso unidimensional, em coordenadas cartesianas, como

$$\frac{\partial}{\partial t} \underbrace{\begin{pmatrix} \rho \\ \rho u \\ e \end{pmatrix}}_Q + \frac{\partial}{\partial x} \underbrace{\begin{pmatrix} \rho u \\ \rho u^2 + p \\ (e + p)u \end{pmatrix}}_E = 0 \quad (1.1)$$

onde Q é o vetor de variáveis conservadas (vetor de solução) e E o vetor de fluxo.

Sabemos também que

$$\begin{aligned} p &= (\gamma - 1) \left(e - \frac{1}{2} \rho u^2 \right), \\ a &= \sqrt{\frac{\gamma p}{\rho}}, \\ H &= \frac{(e + p)}{\rho}, \end{aligned}$$

onde p é a pressão, a é a velocidade do som local e H a entropia total.

2 SOLUÇÃO EXATA

Para obtenção da solução exata do problema do tubo de choque, primeiramente dividimos o tubo em 6 regiões distintas à partir do momento do rompimento do diafragma ($t > t_0$). A Fig. 2 deixa evidente a existência destas 6 regiões. Foram seguidas as orientações dadas em [4] e [2].

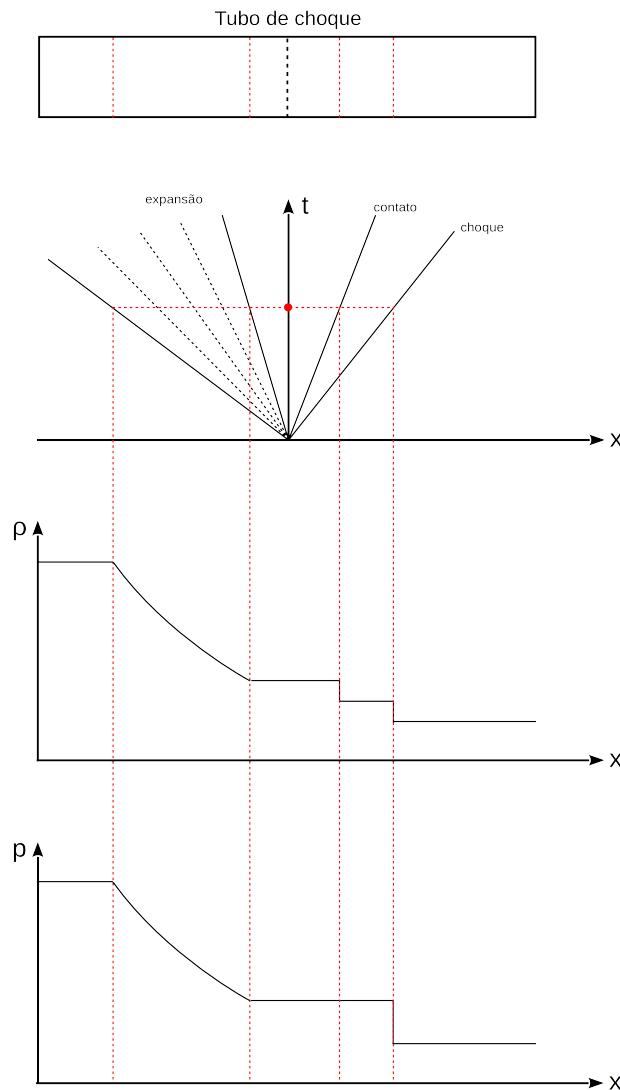


Figura 2: Solução exata para o tubo de choque em $t > t_0$

As regiões 1 e 6 são as regiões de alta e baixa pressão, respectivamente, onde as perturbações provenientes das ondas de expansão e choque ainda não

atingiram os fluidos após o rompimento do diafragma. A região 2 é a região de expansão. Nesta região, as propriedades do fluido são função apenas das propriedades da região 1, de alta pressão. Nas regiões 3 e 4 temos a pressão e velocidade constantes ao longo do tubo. Entretanto, alterações na densidade do fluido são observadas. A região 5 é a região onde existe um salto em todas as propriedades, com relação à região 6, devido à presença da onda de choque.

É possível estabelecer relações entre as propriedades do fluido ao longo de todas as regiões do tubo de choque. As propriedades das regiões 1 e 6 são dadas pelas condições iniciais do problema. As propriedades na região 2 podem ser escritas em função das propriedades na região 1

$$\begin{aligned}\frac{p_2}{p_1} &= \left[\frac{2}{\gamma+1} + \frac{\gamma-1}{a_1(\gamma+1)} \cdot \left(u_1 - \frac{x-x_0}{t} \right) \right]^{\frac{2\gamma}{\gamma-1}} \\ \frac{\rho_2}{\rho_1} &= \left[\frac{2}{\gamma+1} + \frac{\gamma-1}{a_1(\gamma+1)} \cdot \left(u_1 - \frac{x-x_0}{t} \right) \right]^{\frac{2\gamma}{\gamma-1}} \\ u_2 &= \frac{2}{\gamma+1} \cdot \left[a_1 + \frac{1}{2}(\gamma-1)a_1 + \frac{x-x_0}{t} \right],\end{aligned}\quad (2.1)$$

onde x_0 é a posição inicial do choque, e t é o tempo decorrente após o rompimento do diafragma. As propriedades nas regiões 3 e 4 apresentam pressão e velocidade constantes. Desta forma, é possível obter as propriedades nestas regiões através de um método iterativo, utilizando-se das Eqs. 2.2 - 2.4.

$$(p_3 - p_6) \sqrt{\frac{A}{p_3 + B}} + \frac{2a_1}{\gamma-1} \left[\left(\frac{p_3}{p_1} \right)^{\frac{\gamma-1}{2\gamma}} - 1 \right] + u_6 - u_1 = 0, \quad (2.2)$$

onde $A = \frac{2}{\rho_6(\gamma+1)}$ e $B = p_6 \frac{(\gamma-1)}{(\gamma+1)}$. Desta forma, a densidade e velocidade podem ser escritas como

$$\frac{\rho_3}{\rho_1} = \left(\frac{p_3}{p_1} \right)^{\frac{1}{\gamma}} \quad (2.3)$$

$$u_3 - u_1 = \frac{2a_1}{\gamma-1} \left[\left(\frac{p_3}{p_1} \right)^{\frac{\gamma-1}{2\gamma}} - 1 \right]. \quad (2.4)$$

Sendo a pressão e a velocidade constantes entre as regiões 3 e 4, podemos escrever

$$p_4 = p_3 \quad (2.5)$$

$$\frac{\rho_4}{\rho_6} = \left[\frac{p_6(\gamma-1) + p_3(\gamma+1)}{p_3(\gamma-1) + p_6(\gamma-1)} \right] \quad (2.6)$$

$$u_4 = u_3 . \quad (2.7)$$

As posições de cada região onde as descontinuidades são observadas, podem ser facilmente obtidas, através das seguintes definições

Região 1: $x - x_0 < t . (u_1 - a_1)$

Região 2: $t . (u_1 - a_1) \leq x - x_0 < t . (u_3 - a_3)$

Região 3: $t . (u_3 - a_3) \leq x - x_0 < t . u_3$

Região 4: $t . u_3 \leq x - x_0 < t . u_{choque}$

Região 6: $x - x_0 \geq t . u_{choque} ,$

onde u_{choque} é a velocidade da onda de choque, e pode ser escrita como

$$u_{choque} = u_6 + a_6 \sqrt{\frac{p_4(\gamma + 1)}{2\gamma p_6} + \frac{(\gamma - 1)}{2\gamma}} .$$

3 ESQUEMAS NUMÉRICOS

Esta seção apresentará os esquemas numéricos implementados para solução do problema do tubo de choque. Alguns estudos foram realizado para auxiliar a implementação dos métodos numéricos, como por exemplo o estudo de refinamento de malha, influência do coeficiente de dissipação artificial linear, entre outros.

3.1 Método de Marcha no Tempo

3.1.1 Esquema de Euler Explícito

Para a discretização temporal das equações, foi utilizado o método de Euler explícito. Neste caso, a discretização do termo transiente é dada por

$$\frac{\partial Q}{\partial t} = \frac{Q_j^{n+1} - Q_j^n}{\Delta t} + \Delta t \cdot \mathcal{O}(\Delta t) . \quad (3.1)$$

Este método nos fornece uma discretização de primeira ordem de precisão, $\mathcal{O}(\Delta t)$, conforme observado pela Eq. 3.1. Desta forma, é importante ficar atento ao passo no tempo utilizado, para que não seja comprometida a resolução da solução transiente. Este passo no tempo é obtido pela condição de CFL, conforme descrito a seguir.

3.1.2 Condição de CFL

Quando falamos em esquemas de marcha no tempo explícitos, devemos estabelecer um dado passo no tempo suficientemente grande para manter boa resolução da solução transiente, ao mesmo tempo em que se deseja manter a estabilidade do método. Este passo no tempo é obtido através da condição conhecida como condição de CFL (Courant-Friedrichs-Lowy). O número de CFL é um adimensional que indica com qual velocidade a informação se propaga ao longo da malha computacional. O número de CFL, para o caso unidimensional, é definido como

$$CFL = a^* \frac{\Delta t}{\Delta x} ,$$

onde a^* é uma velocidade característica, Δt é o passo no tempo e Δx é o espaçamento de malha. Para que seja respeitada a condição de CFL, é recomendado utilizar como velocidade característica o maior valor dos autovalores associados, λ_i , da solução. Para o caso das equações de Euler unidimensionais, temos três autovalores associados

$$\lambda_1 = u - a$$

$$\lambda_2 = u$$

$$\lambda_3 = u + a .$$

Com o objetivo de se obter a maior velocidade característica, é selecionada a maior velocidade característica, neste caso $\lambda_3 = u + a$, utilizando-se $u = u_{max}$ e $a = a_{max}$, onde u_{max} é adotada como a velocidade do choque, definida pela condição de entropia pela Eq. 3.2, e a_{max} a velocidade do som local, obtida com as propriedades do lado de maior pressão, definida pela Eq. 3.3.

$$u_{choque} = \frac{[E]}{[Q]} \quad (3.2)$$

$$a = \sqrt{\gamma \frac{p_4}{\rho_4}} \quad (3.3)$$

3.2 Dissipação Artificial

3.2.1 Dissipação Artificial Linear

Com o objetivo de se reduzir oscilações da solução nas regiões de descontínuidade (choque), modelos de dissipação artificial são introduzidos nas equações. Seguindo a proposta feita por Jameson *et al.* [3], ao adicionarmos o termo de dissipação artificial, estaremos resolvendo a seguinte equação

$$\frac{d}{dt}(Q) + S(Q) - D(Q) = 0 \quad (3.4)$$

onde $S()$ é o operador de discretização espacial implementado de acordo com cada método numérico implementado, e $D()$ é o termo de dissipação artificial adicionado à equação.

Dois modelos de dissipação artificial linear foram implementados. Um modelo de dissipação linear de diferenças de segunda ordem $D2(Q)$, e um modelo de diferenças de quarta ordem $D4(Q)$.

$$\begin{aligned} D2(Q)_j &= \frac{\mu}{8} \frac{\Delta x}{\Delta t} (Q_{j+1} - 2Q_j + Q_{j-1}) \\ D4(Q)_j &= -\frac{\mu}{8} \frac{\Delta x}{\Delta t} (Q_{j+2} - 4Q_{j+1} + 6Q_j - 4Q_{j-1} + Q_{j-2}) \end{aligned} \quad (3.5)$$

onde μ é o coeficiente de dissipação artificial, Q é o vetor de variáveis conservadas (vetor solução), Δx e Δt são o espaçamento de malha e o passo no tempo, respectivamente. O coeficiente de dissipação dos modelos lineares foi definido de tal forma que, com o menor valor possível do coeficiente, uma solução numérica fosse obtida (não houvesse divergência na solução) e não fossem observadas grandes oscilações na região de descontinuidade, mesmo para altas razões de pressão ($p_4/p_1 = 50$).

Uma razão de pressão de $p_4/p_1 = 40$ foi utilizada para o estudo do coeficiente de dissipação, a fim de se manter a análise dos resultados para $p_4/p_1 = 50$ na seção de discussão de resultados. Entretanto, esta razão mais alta foi levada em conta para a determinação do coeficiente. Foram testados os valores de $\mu = 1.0$, $\mu = 0.5$, $\mu = 0.4$ e $\mu = 0.2$, conforme ilustrado nas figuras. Por fim, bons resultados foram observados para $\mu = 0.4$, sendo este o valor assumido para os demais casos dos projetos.

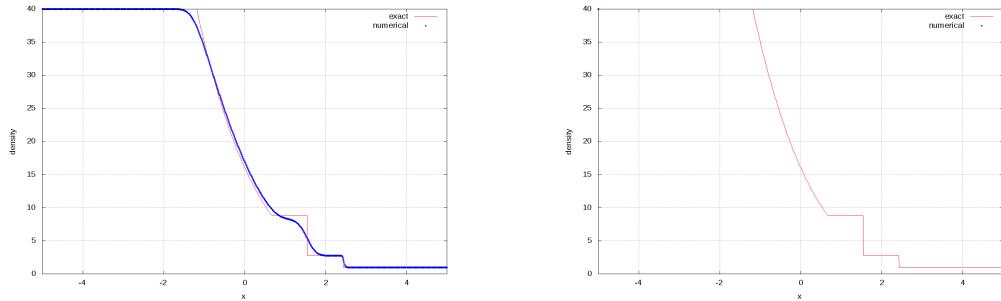


Figura 3: Resultado de densidade para $D2(Q)$ e $D4(Q)$, com $\mu = 1.0$

No primeiro caso, o valor do coeficiente é assumido $\mu = 1.0$. Uma grande dissipação é observada no modelo de diferenças segundas, ao passo que no modelo

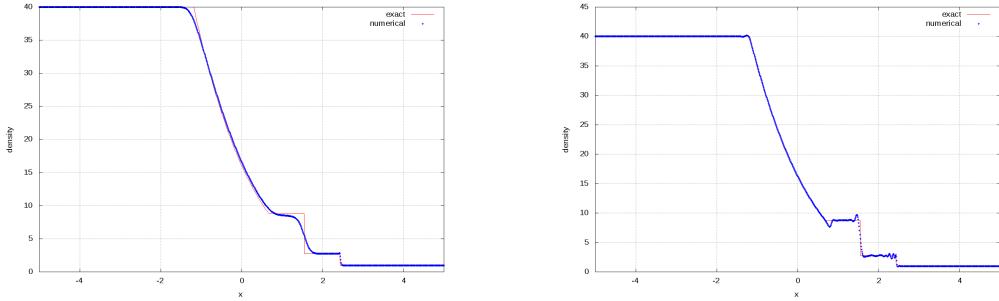


Figura 4: Resultado de densidade para $D2(Q)$ e $D4(Q)$, com $\mu = 0.5$

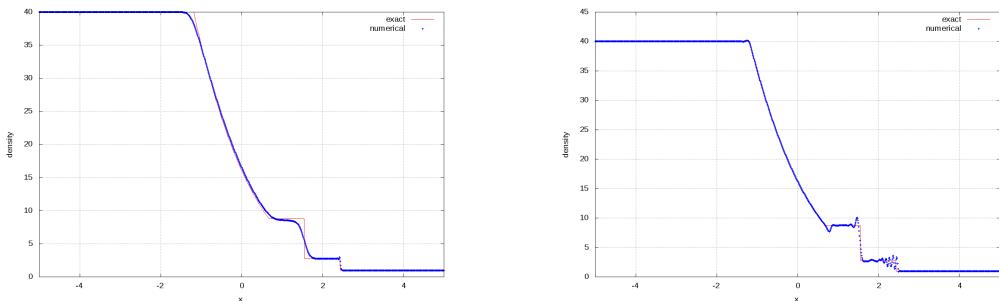


Figura 5: Resultado de densidade para $D2(Q)$ e $D4(Q)$, com $\mu = 0.4$

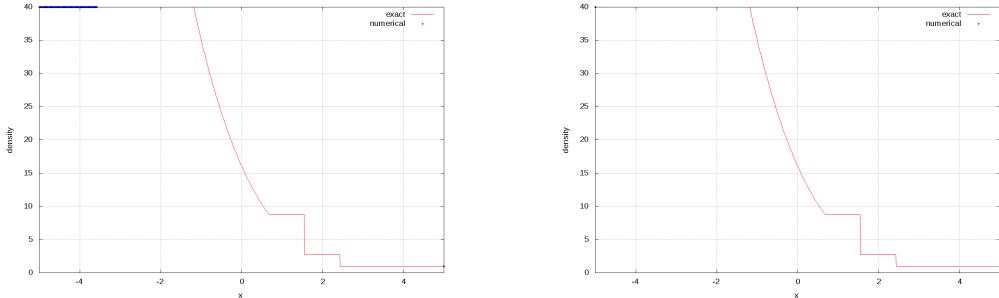


Figura 6: Resultado de densidade para $D2(Q)$ e $D4(Q)$, com $\mu = 0.2$

de diferenças quartas a solução acaba divergindo, conforme Fig. 3. A discussão sobre o por que disto ter acontecido será feita na seção de discussão de resultados.

Para o caso onde o coeficiente de dissipação é reduzido a $\mu = 0.2$, a solução numérica diverge tanto para o modelo de diferenças segundas, quanto para o caso de diferenças quartas, como mostra a Fig. 6.

Para os casos onde $\mu = 0.5$ e $\mu = 0.4$, bons resultados foram observados em ambos os modelos de dissipação linear, como mostrado nas Figs. 4 e 5,

respectivamente. Apesar de apresentar um pouco mais de oscilações no modelo de diferenças quartas, o valor de $\mu = 0.4$ foi selecionado.

3.2.2 Dissipação Artificial Não-linear

Para o modelo de dissipação não-linear, foi utilizado também como referência o trabalho proposto por Jameson [3]. Com a idéia de combinar os benefícios dos modelos de dissipação de diferenças segundas e diferenças quartas em um único modelo, Jameson define o termo de dissipação como

$$D(Q) = d_{j+\frac{1}{2}}(Q_j) - d_{j-\frac{1}{2}}(Q_j) \quad (3.6)$$

onde os subscritos $+1/2$ e $-1/2$ indicam o cálculo do termo de dissipação nas interfaces entre dois pontos de solução da malha. O termo $d_{j+\frac{1}{2}}$ é definido como

$$d_{j+\frac{1}{2}}(Q) = \frac{\Delta x}{\Delta t} \left[\epsilon_{j+\frac{1}{2}}^{(2)}(Q_{j+1} - Q_j) + \epsilon_{j+\frac{1}{2}}^{(4)}(Q_{j+2} - 3Q_{j+1} + Q_j - Q_{j-1}) \right], \quad (3.7)$$

onde

$$\begin{aligned} \epsilon_{j+\frac{1}{2}}^{(2)} &= \kappa^{(2)} \max(\nu_{j+1}, \nu_j) \\ \epsilon_{j+\frac{1}{2}}^{(4)} &= \max[0, (\kappa^{(4)} - \epsilon_{j+\frac{1}{2}}^{(2)})] . \end{aligned}$$

Neste contexto, $\kappa^{(2)}$ e $\kappa^{(4)}$ são constantes ajustáveis. No artigo original, Jameson define os valores de $\kappa^{(2)} = 1/4$ e $\kappa^{(4)} = 1/256$, entretanto estes valores não se adequaram para o problema proposto e estas constantes foram ajustadas para obter melhores resultados. Os valores utilizados ao longo deste trabalho foram de $\kappa^{(2)} = 1/2$ e $\kappa^{(4)} = 1/20$. Estes valores apresentaram bons resultados em regiões de descontinuidades, além de diminuirem o fenômeno de Gibbs na região do choque, conforme observado na Fig. 7.

3.3 Forma Conservativa dos Esquemas Numéricos

Ao trabalharmos com problemas que apresentam descontinuidade da solução, é importante escrever as equações na forma conservativa, de maneira que sejam respeitadas as leis de conservação. Desta forma, ao adotarmos a estratégia de diferenças finitas, estamos calculando a solução em pontos de solução. Para forçar

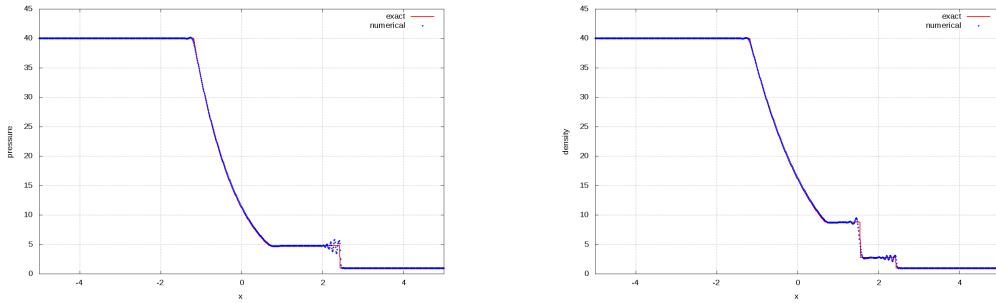


Figura 7: Resultados de pressão e densidade para modelo de dissipação não-linear

o esquema numérico a se manter na forma conservativa, faz-se necessário definir pontos de interface entre dois pontos de solução onde existem descontinuidades, conforme ilustrado na Fig. 8. Nestes pontos, calcula-se um fluxo numérico, tal que o método mantém-se na forma conservativa.

- Pontos de solução ▲ Pontos de fluxo

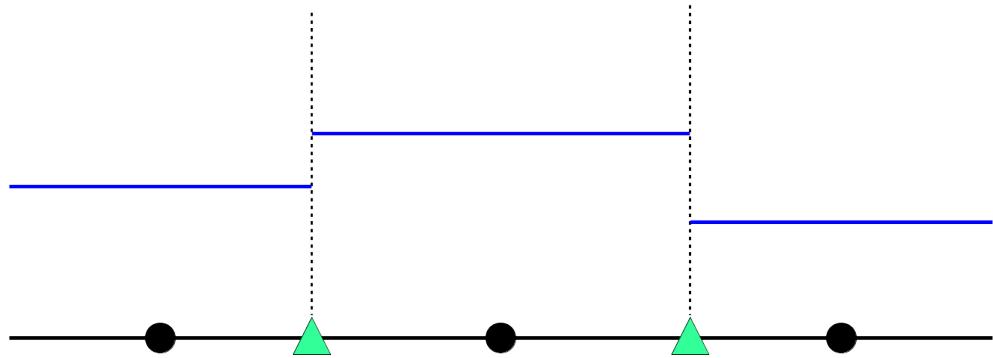


Figura 8: Ilustração de descontinuidade entre pontos de solução da malha

De uma forma geral, os esquemas numéricos podem ser escritos como

$$Q_j^{n+1} = Q_j^n - \frac{\Delta t}{\Delta x} (F_{j+\frac{1}{2}}^n - F_{j-\frac{1}{2}}^n) , \quad (3.8)$$

onde $F_{j+\frac{1}{2}}^n$ é o fluxo numérico entre os pontos j e $j+1$, e $F_{j-\frac{1}{2}}^n$ é o fluxo numérico entre os pontos j e $j-1$, ambos definidos de acordo com cada método numérico implementado.

3.4 Efeito do Refinamento da Malha

A fim de se obter um número *ideal* - entenda-se aqui por ideal como sendo um número de pontos de solução na malha que proporcione resolução numérica adequada, sem comprometer de forma exagerada o custo computacional para esta solução - de pontos na malha. O estudo de refinamento de malha foi feito utilizando-se o primeiro esquema numérico proposto (esquema centrado + termo de dissipação artificial) para solução do problema.

A proposta dos projetos 1, 2 e 3 é resolver as equações de Euler em uma única dimensão e, portanto, a malha computacional foi assumida na direção x . Os números de pontos de solução analisados para o estudo de refinamento de malha foram respectivamente 100, 200, 300, 500 e 1,000.

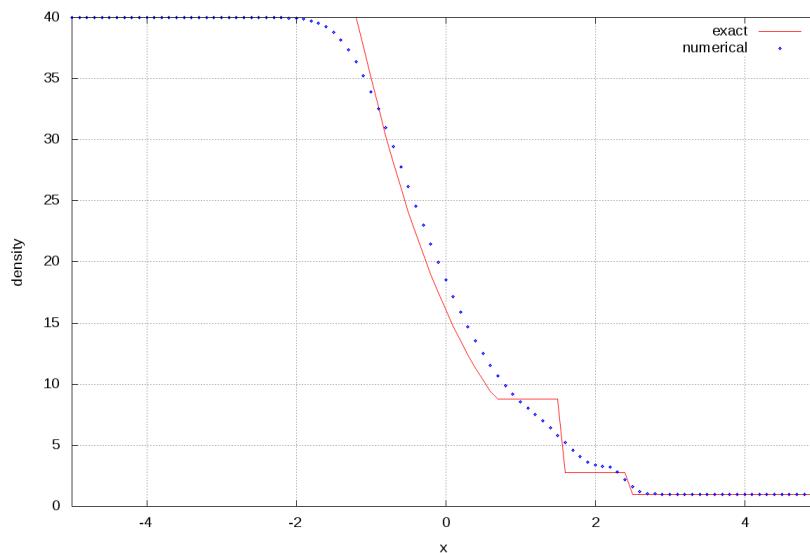


Figura 9: Resultados de densidade para malha com 100 pontos

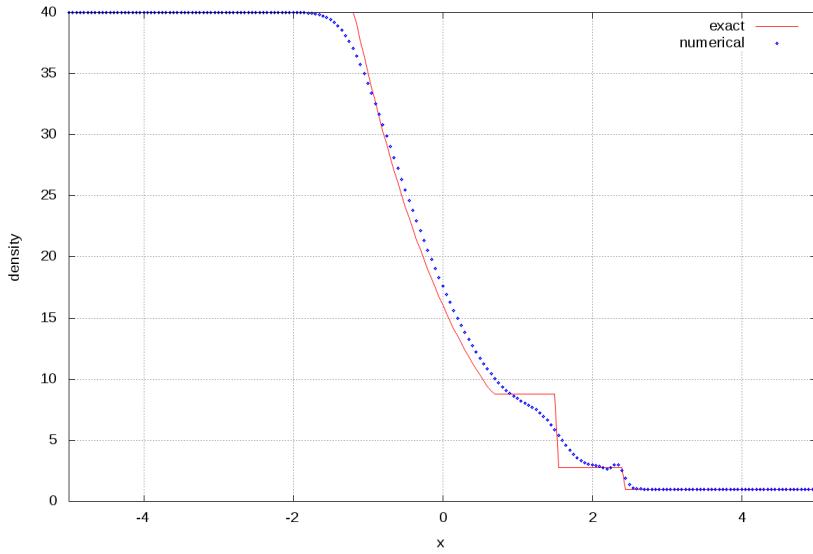


Figura 10: Resultados de densidade para malha com 200 pontos

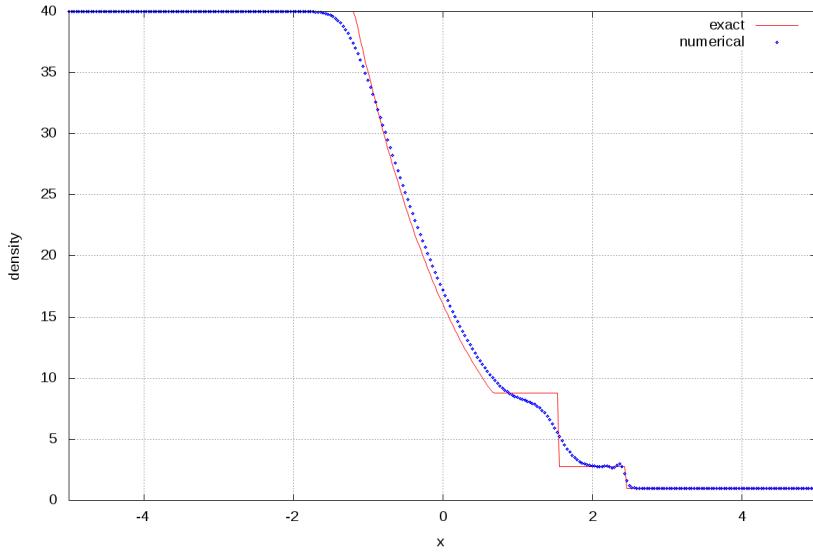


Figura 11: Resultados de densidade para malha com 300 pontos

É possível observar que, para uma malha mais grosseira, ou seja, para um menor número de pontos na malha computacional, o resultado fica grosso e, portanto, a solução numérica não captura bem os fenômenos presentes, por exemplo, nas regiões de descontinuidades. As Figuras (9 - 13) mostram os resultados de densidade, para uma razão de pressão inicial de 40, variando a quantidade de pontos da malha e utilizando-se o mesmo método de solução (centrado + termo

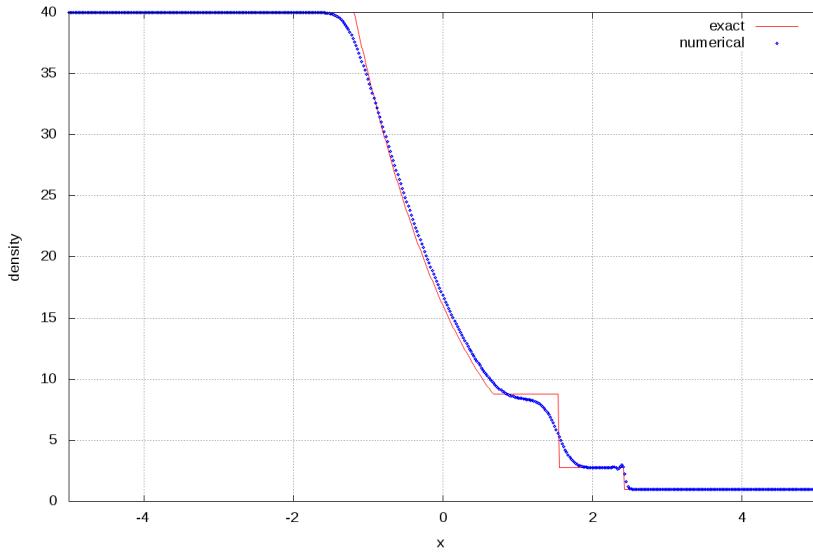


Figura 12: Resultados de densidade para malha com 500 pontos

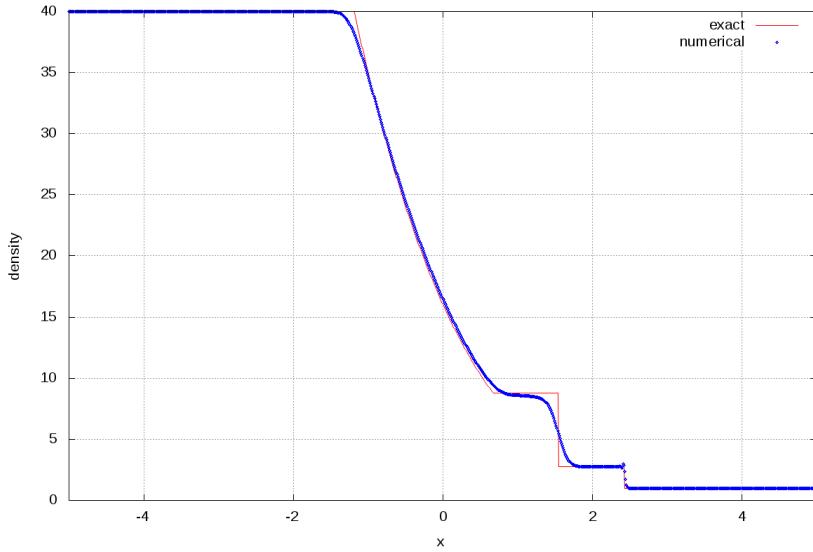


Figura 13: Resultados de densidade para malha com 1,000 pontos

de dissipação artificial de segunda ordem). As malhas computacionais com 500 e 1000 pontos de solução apresentaram bons resultados. Os resultados também foram gerados para uma malha de 10,000 pontos. Entretanto, este foi considerado um número exagerado de pontos para o problema proposto e, portanto, a malha com 1,000 pontos foi utilizada.

3.5 Projeto 1

A proposta do projeto 1 é resolver o problema do tubo de choque utilizando os seguintes métodos numéricos:

- Esquema Centrado + termo de dissipação artificial;
- Esquema de Lax-Wendroff;
- Esquema Explícito de MacCormack.

Cada um destes esquemas será descrito a seguir.

3.5.1 Esquema Centrado

Conforme mencionado anteriormente, o esquema centrado é o mais simples de todos os esquemas numéricos implementados para este problema. O fluxo numérico deste esquema é dado por

$$F_{j+\frac{1}{2}}^n = \frac{1}{2} (E_{j+1}^n + E_j^n) ,$$

onde E_{j+1}^n e E_j^n são os valores do vetor de fluxo calculados nos pontos $j+1$ e j da malha, respectivamente.

Este esquema, entretanto, por se tratar de um esquema centrado aplicado às equações de Euler que, expandindo em série de Taylor, anula as derivadas de ordem par (2, 4, 6, ...), fazendo com que a solução seja dispersiva, no caso do surgimento de oscilações. Desta forma, faz-se necessária a adição de termos de dissipação artificial. Conforme anteriormente descrito, três modelos distintos de dissipação artificial foram implementados: **(i)** dissipação linear de derivadas de 2^a ordem, **(ii)** dissipação linear de derivadas de 4^a ordem e **(iii)** dissipação não-linear. Desta forma, a equação discretizada torna-se

$$Q_j^{n+1} = Q_j^n - \frac{\Delta t}{\Delta x} (F_{j+\frac{1}{2}}^n - F_{j-\frac{1}{2}}^n + D(Q_j)) ,$$

onde $D(Q)$ é o termo de dissipação artificial selecionado entre as três opções apresentadas. É importante observar que todo fluxo numérico a seguir, pode

ser escrito como sendo o fluxo numérico de um esquema centrado simples com adição de um termo de dissipação artificial proveniente de cada método.

A. Esquema Centrado com Dissipação Artificial Linear de 2^a Ordem

Para o caso do esquema centrado com termo de dissipação artificial linear de derivadas de segunda ordem, temos $D(Q) = D2(Q)$. Os resultados são apresentados a seguir.

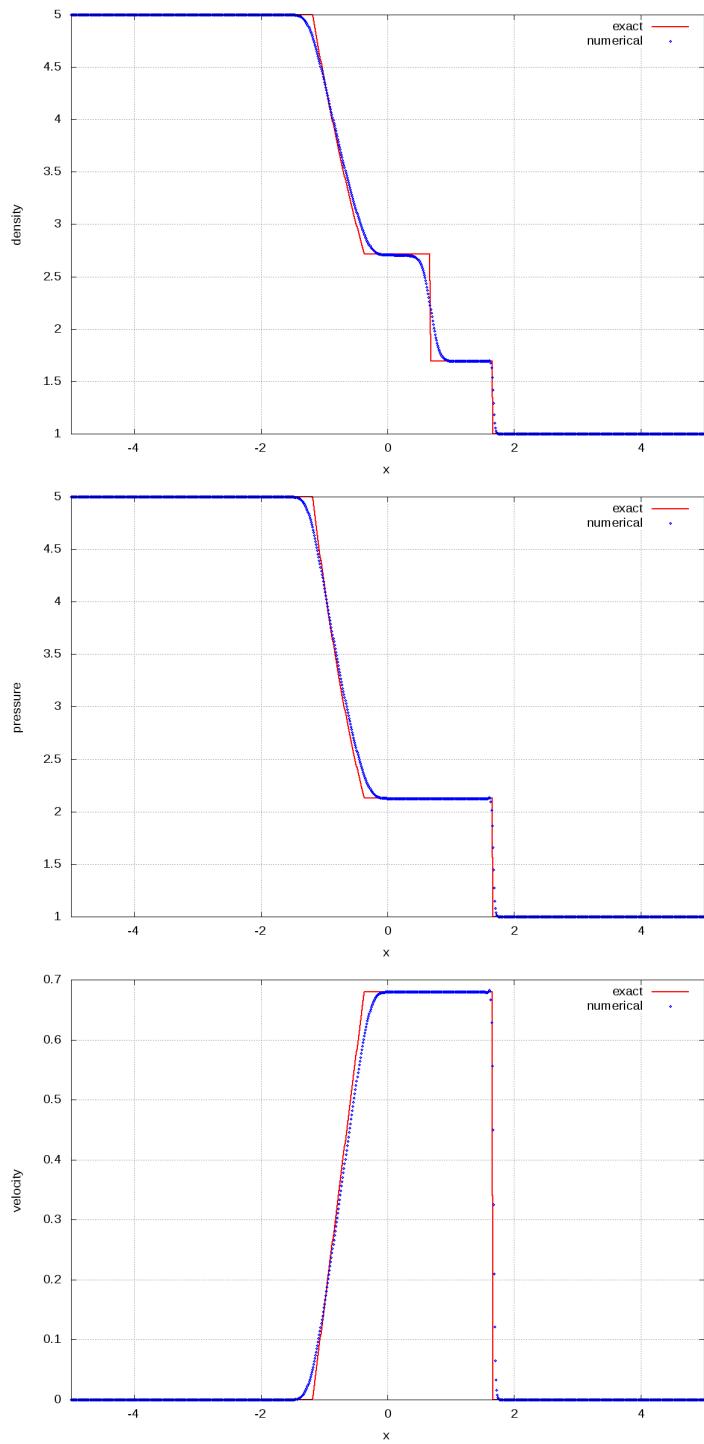


Figura 14: Resultados do esquema centrado com dissipação artificial D2(Q),

$$\frac{p_4}{p_1} = 5$$

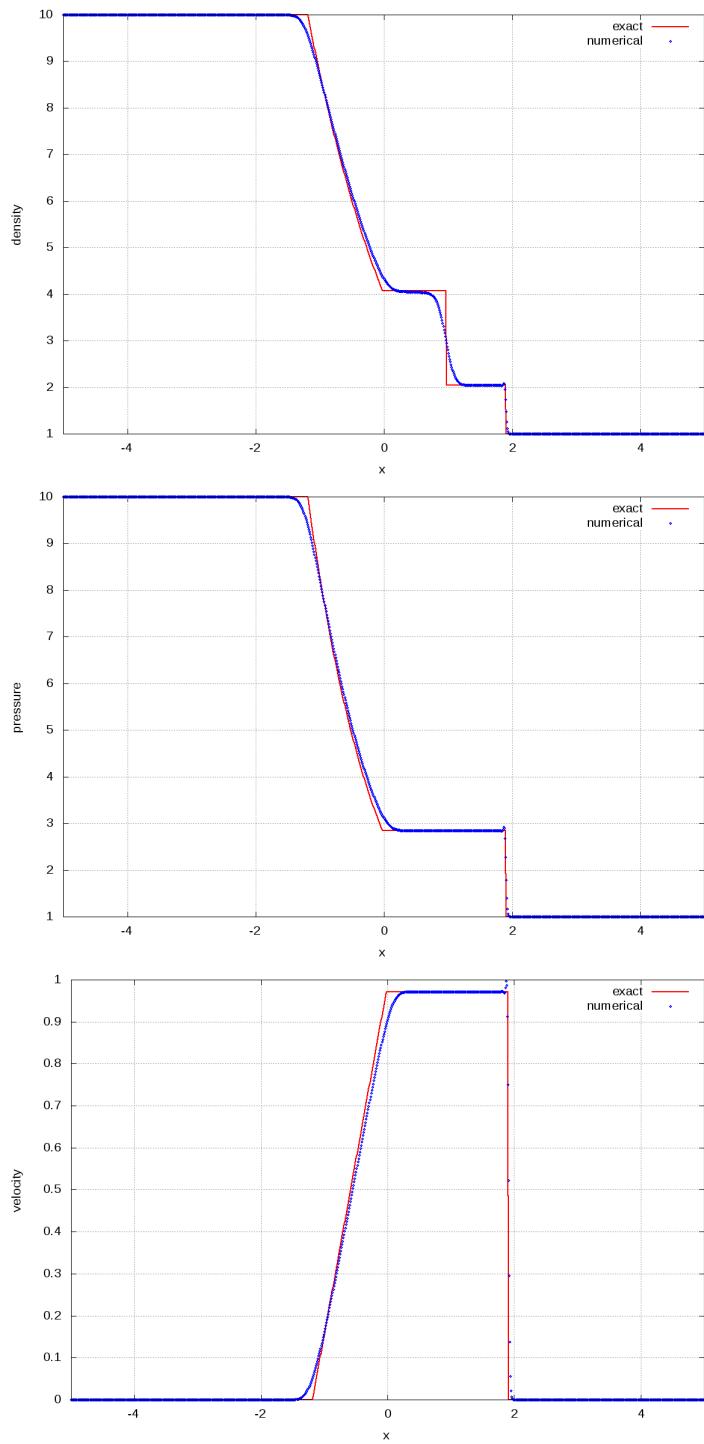


Figura 15: Resultados do esquema centrado com dissipação artificial D2(Q),

$$\frac{p_4}{p_1} = 10$$

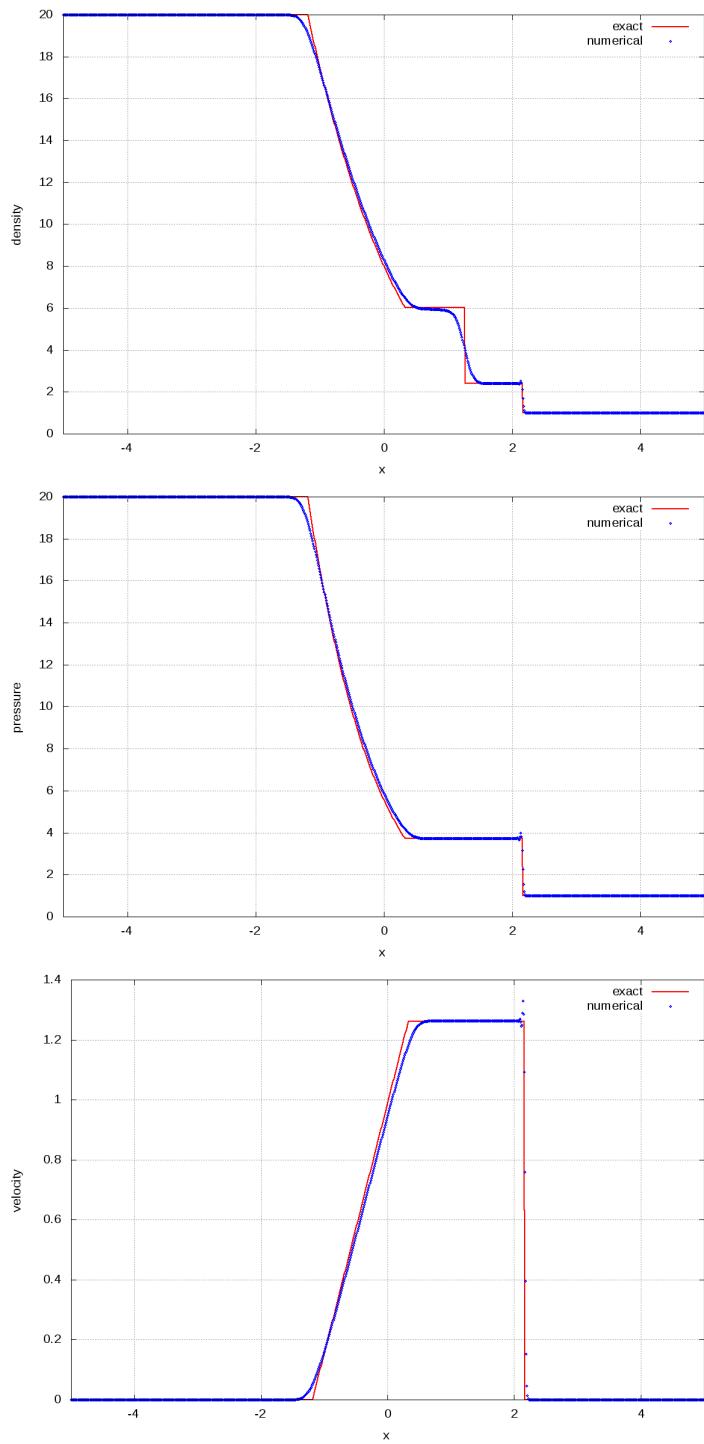


Figura 16: Resultados do esquema centrado com dissipação artificial D2(Q),

$$\frac{p_4}{p_1} = 20$$

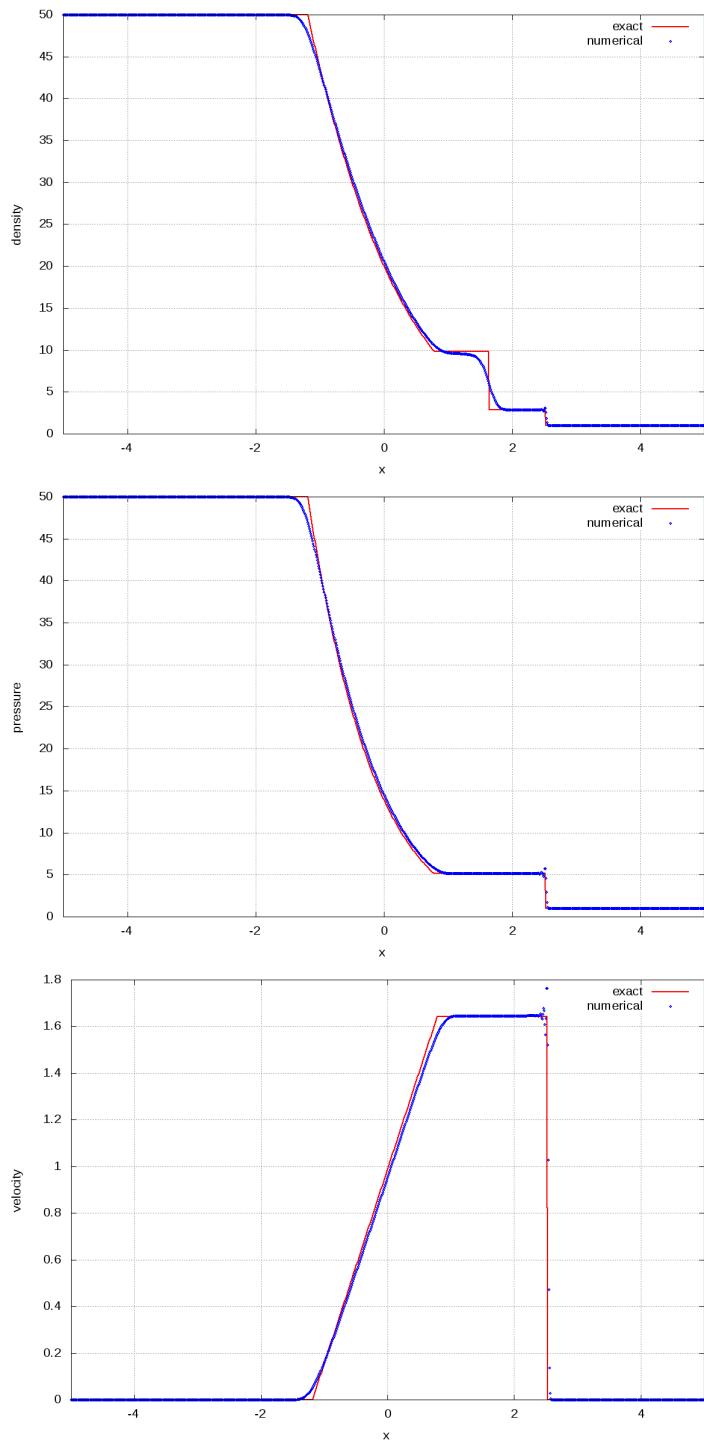


Figura 17: Resultados do esquema centrado com dissipação artificial D2(Q),

$$\frac{p_4}{p_1} = 50$$

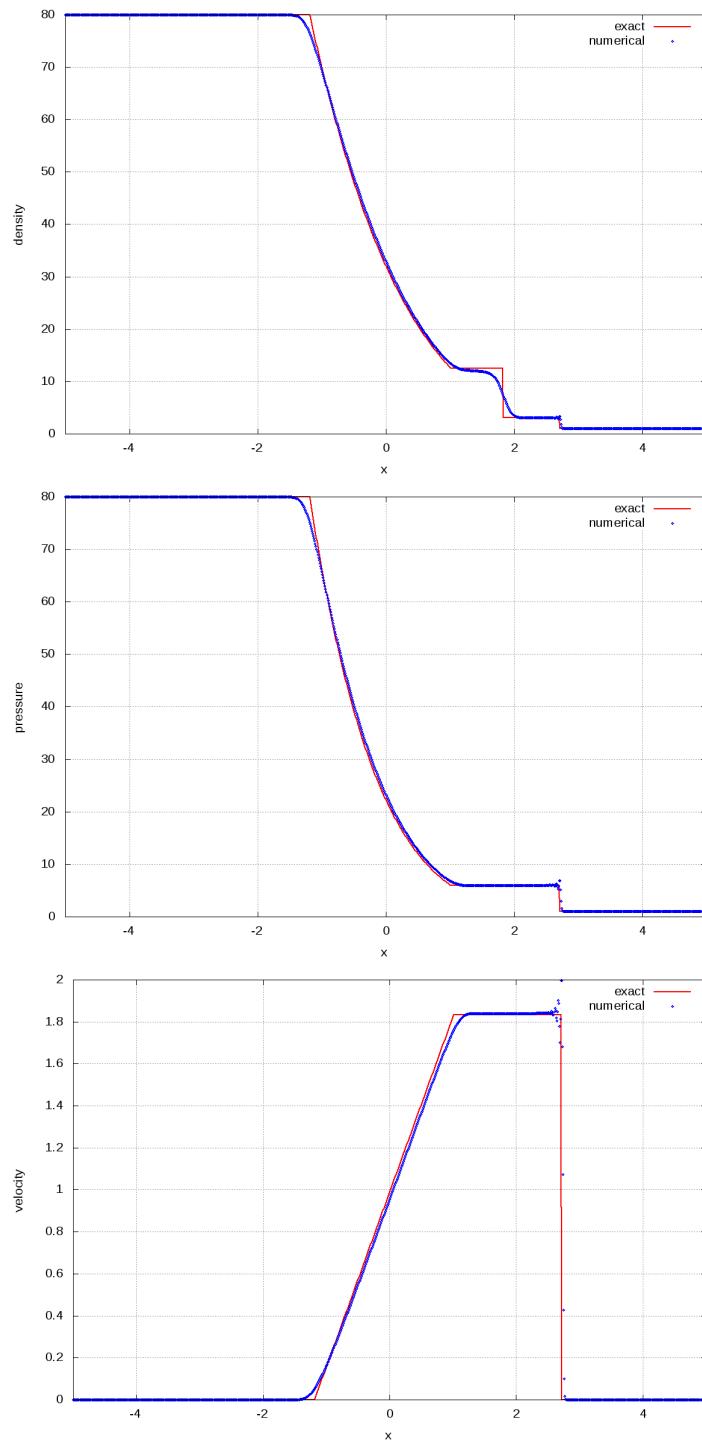


Figura 18: Resultados do esquema centrado com dissipação artificial D2(Q),

$$\frac{p_4}{p_1} = 80$$

B. Esquema Centrado com Dissipação Artificial Linear de 4^a Ordem

Para o caso do esquema centrado com termo de dissipação artificial linear de derivadas de segunda ordem, temos $D(Q) = D4(Q)$. Os resultados são apresentados a seguir.

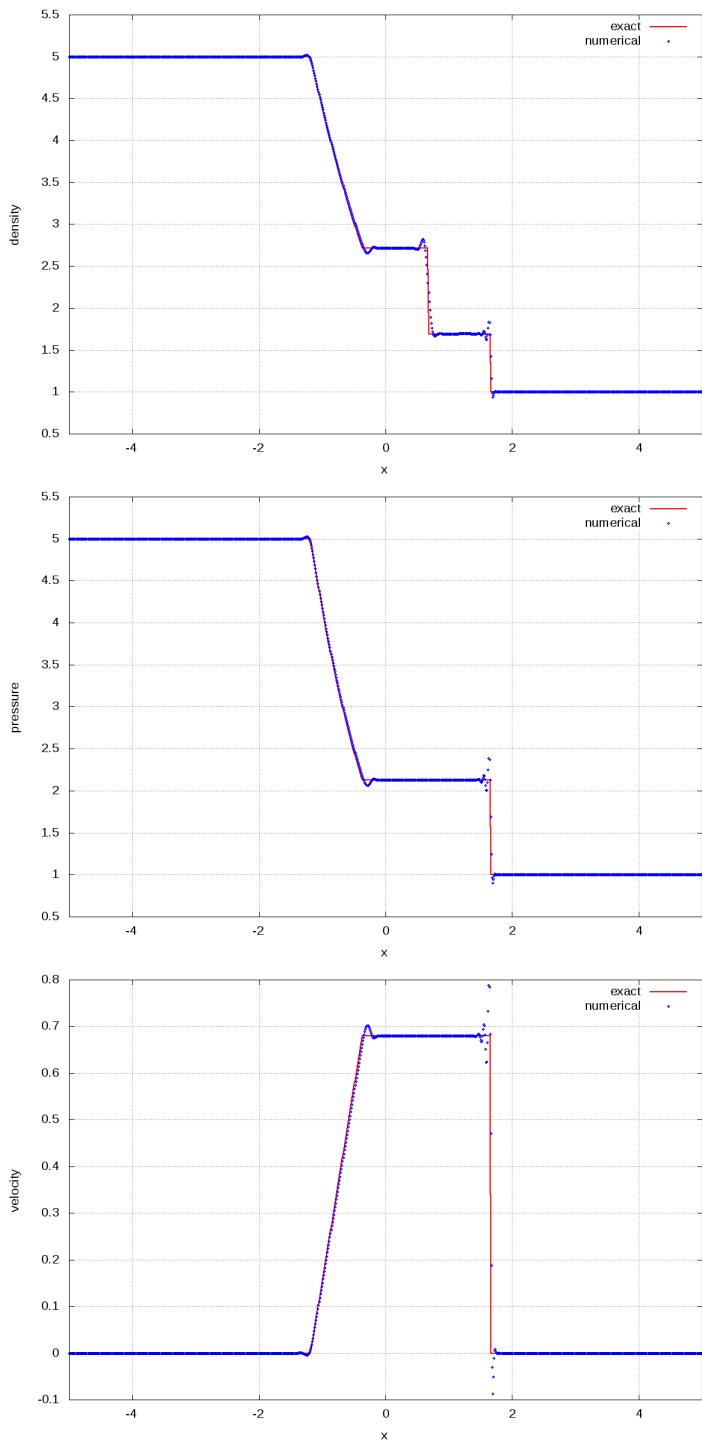


Figura 19: Resultados do esquema centrado com dissipação artificial D4(Q),

$$\frac{p_4}{p_1} = 5$$

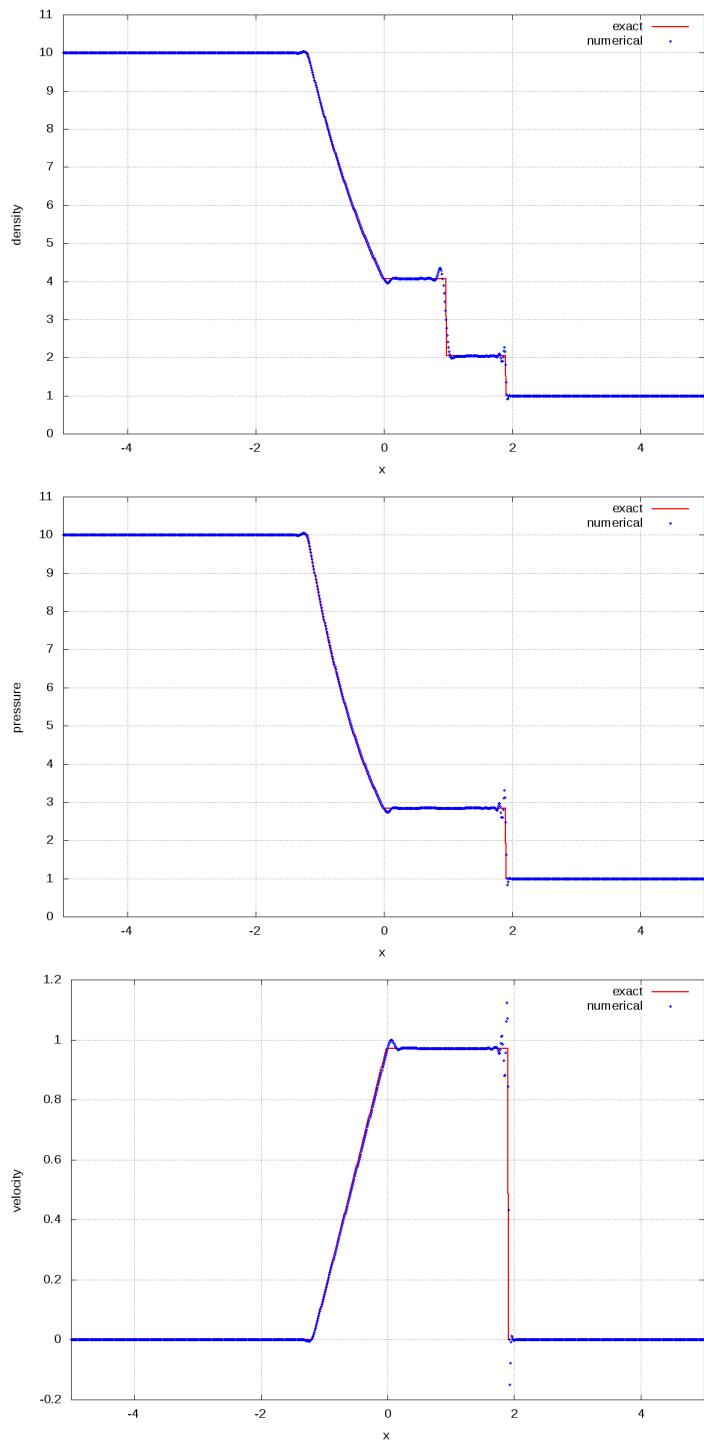


Figura 20: Resultados do esquema centrado com dissipação artificial D4(Q),

$$\frac{p_4}{p_1} = 10$$

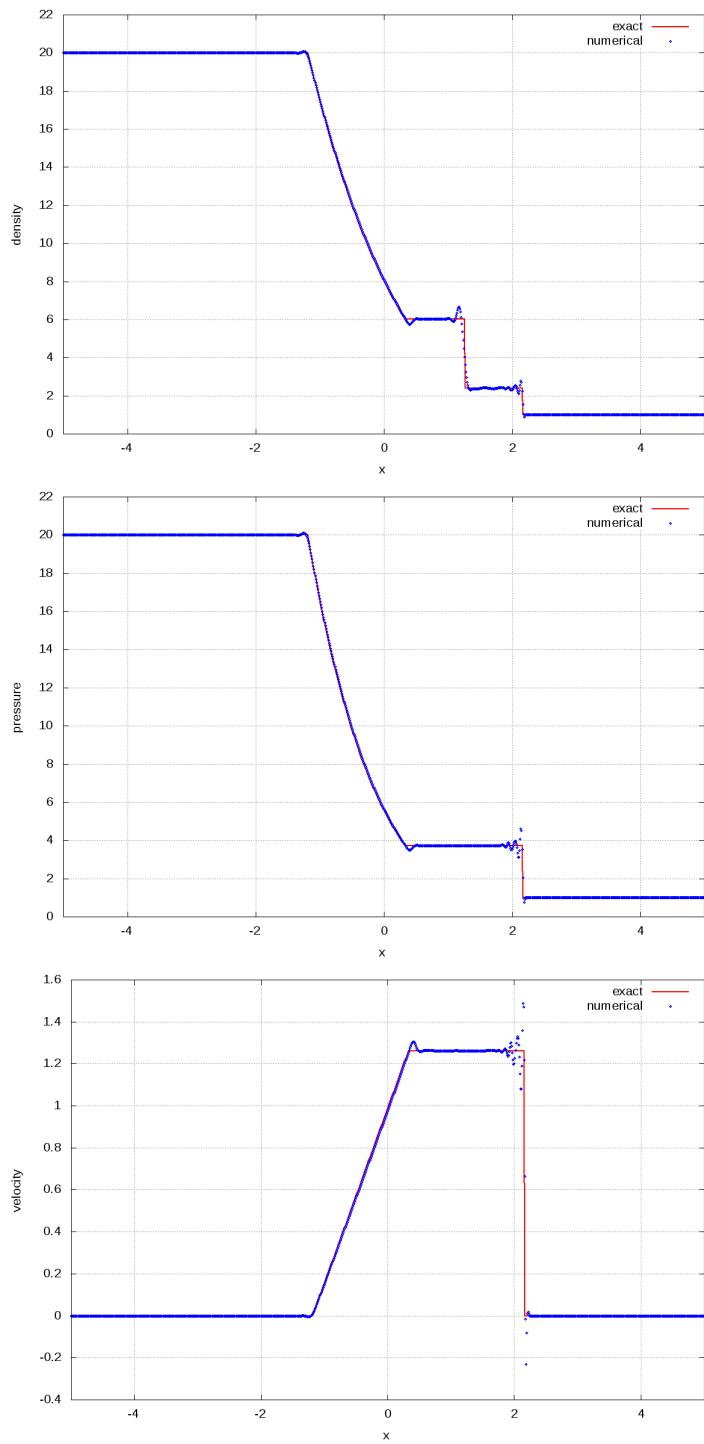


Figura 21: Resultados do esquema centrado com dissipação artificial D4(Q),

$$\frac{p_4}{p_1} = 20$$

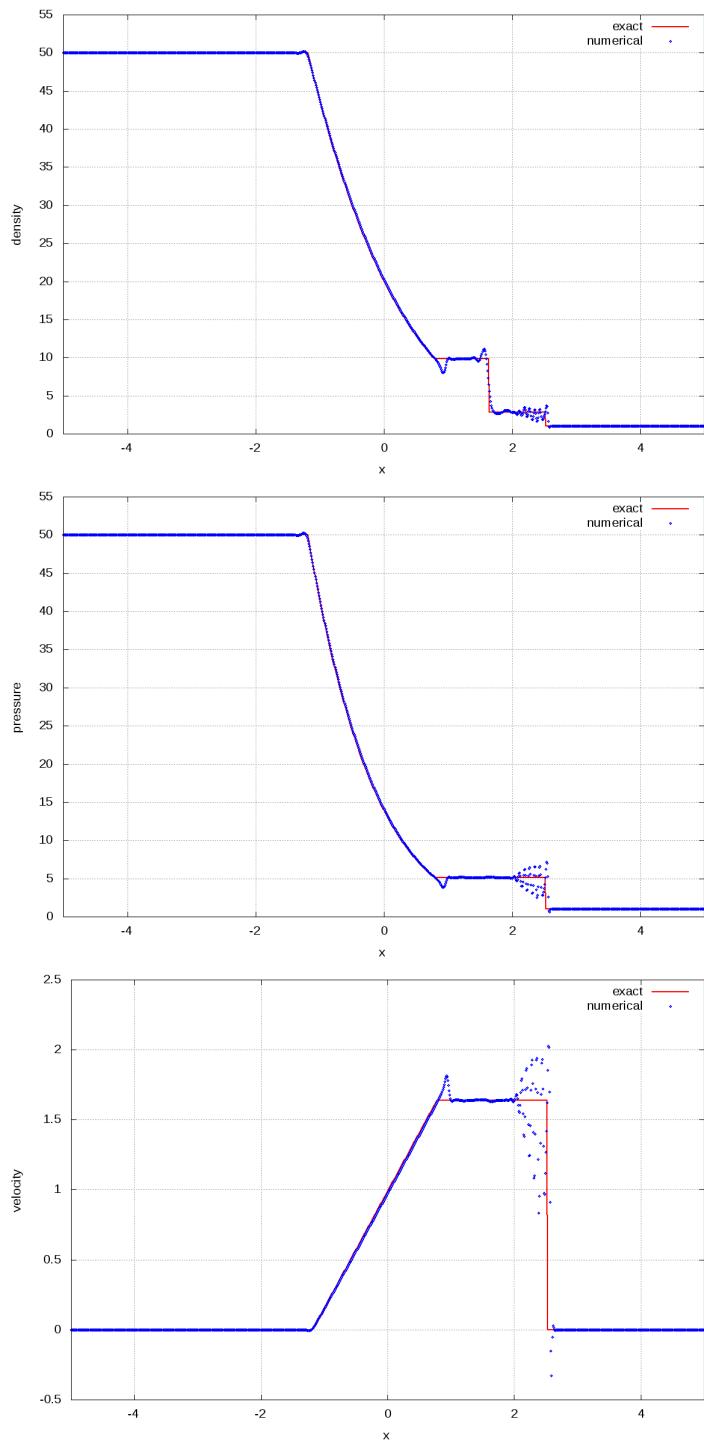


Figura 22: Resultados do esquema centrado com dissipação artificial D4(Q),

$$\frac{p_4}{p_1} = 50$$

C. Esquema Centrado com Dissipação Artificial não-Linear

Para o caso do esquema centrado com termo de dissipação artificial não-linear, temos $D(Q) = d_{j+\frac{1}{2}} - d_{j-\frac{1}{2}}$, onde $d_{j+\frac{1}{2}}$ e $d_{j-\frac{1}{2}}$ são definidos pela Eq. 3.7. Os resultados são apresentados a seguir.

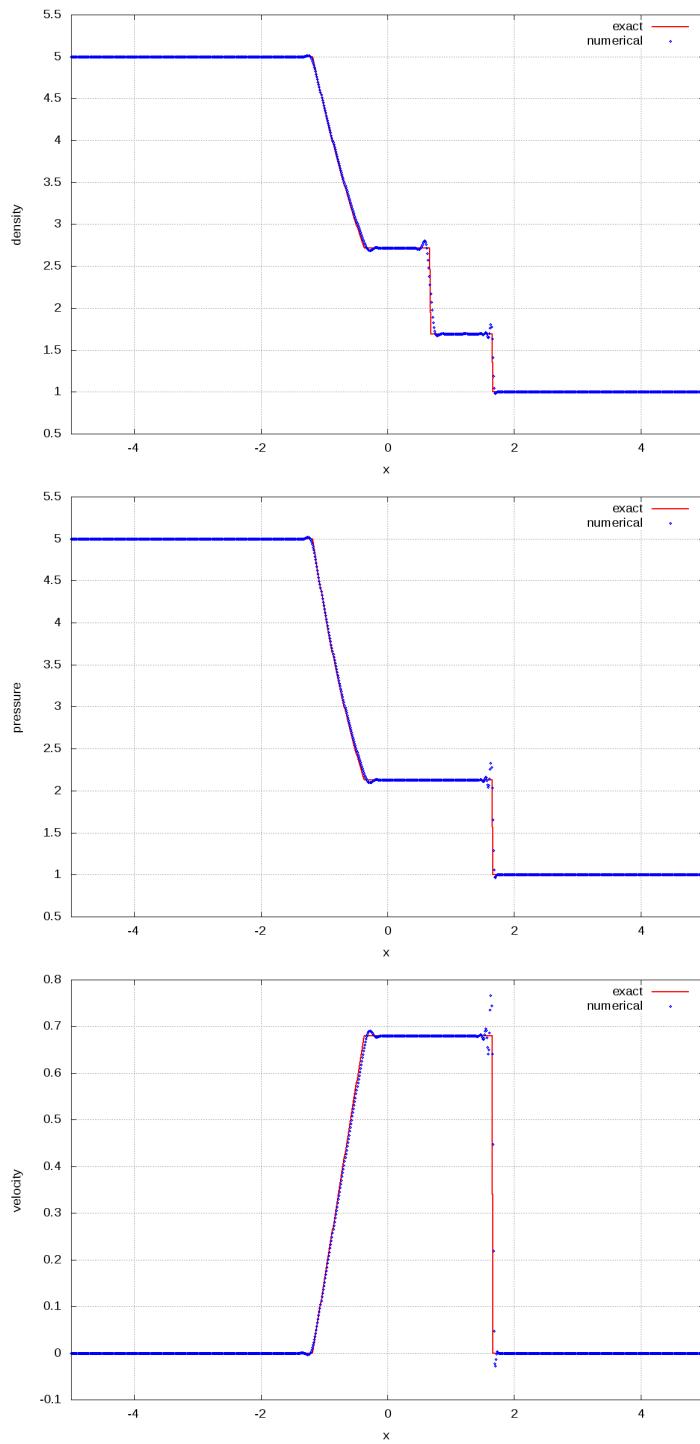


Figura 23: Resultados do esquema centrado com dissipação artificial não linear,

$$\frac{p_4}{p_1} = 5$$

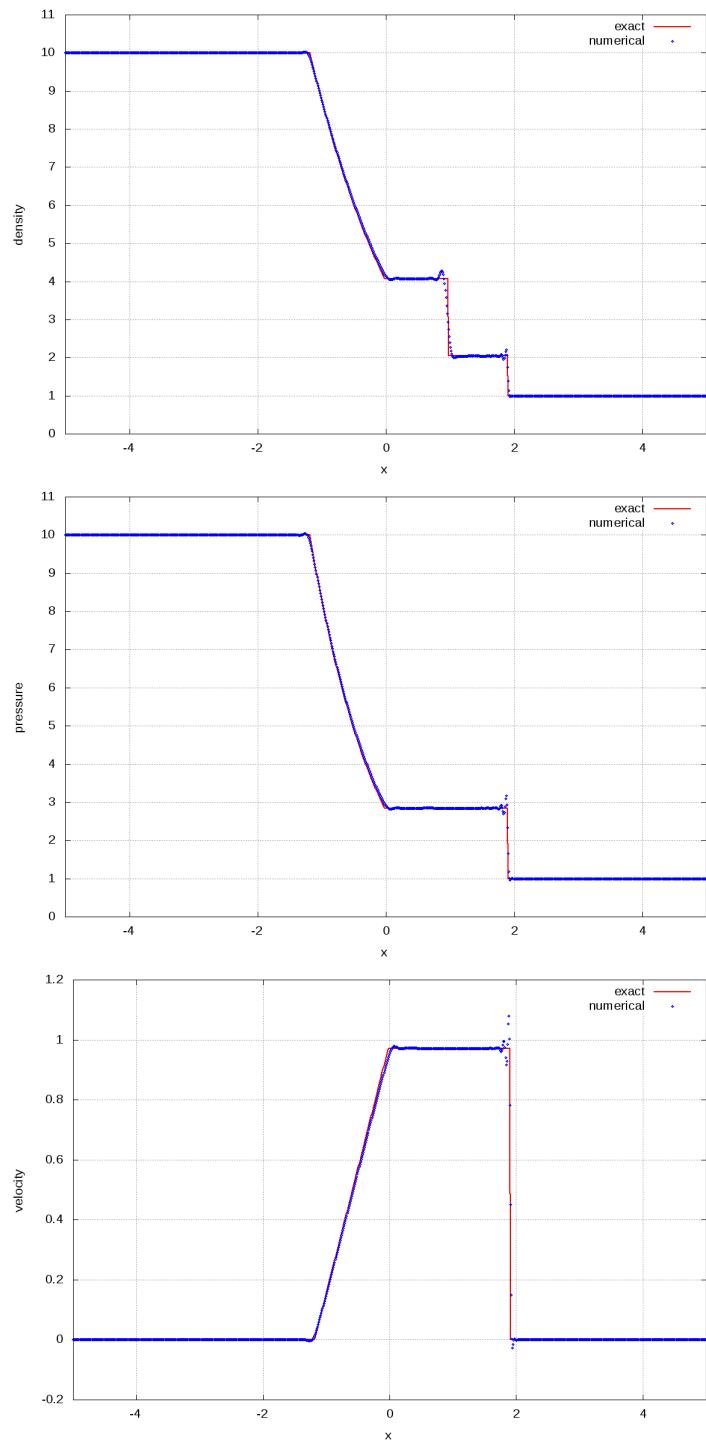


Figura 24: Resultados do esquema centrado com dissipação artificial não linear,

$$\frac{p_4}{p_1} = 10$$

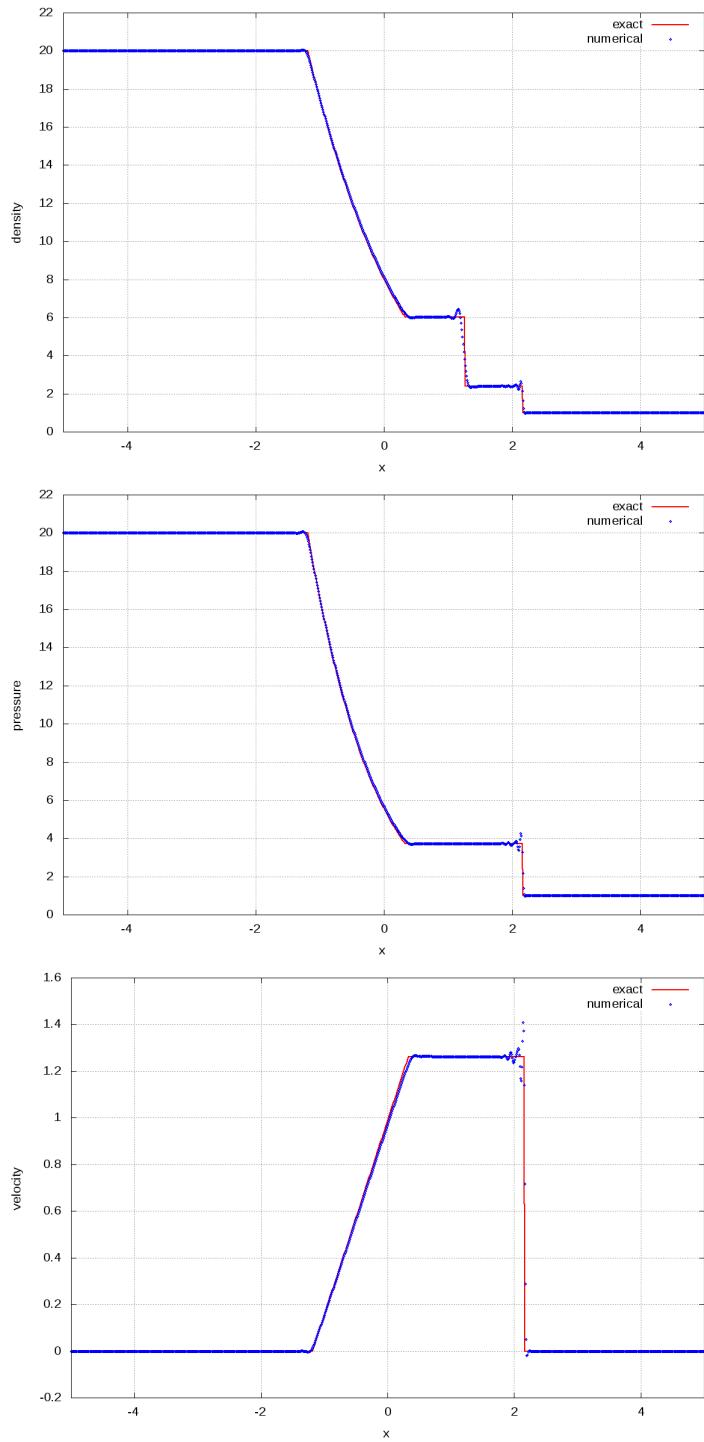


Figura 25: Resultados do esquema centrado com dissipação artificial não linear,

$$\frac{p_4}{p_1} = 20$$

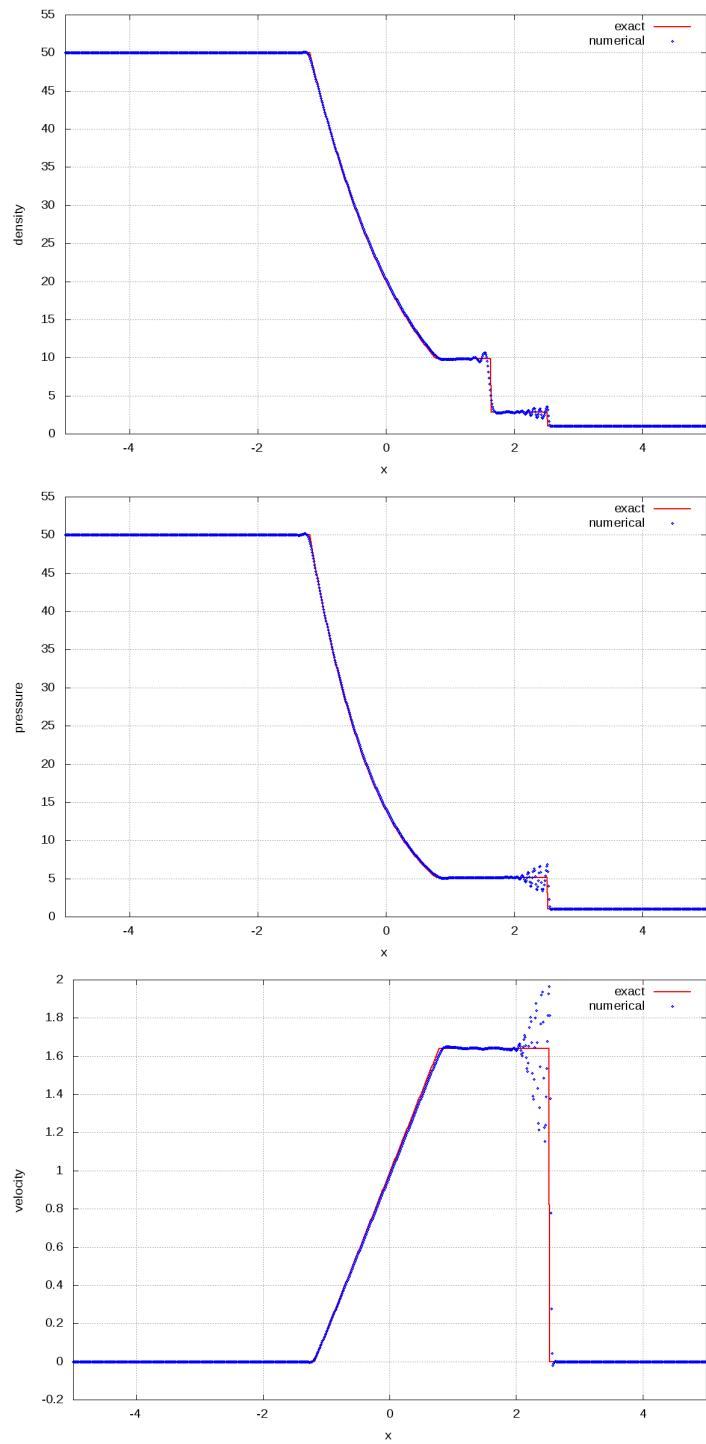


Figura 26: Resultados do esquema centrado com dissipação artificial não linear,

$$\frac{p_4}{p_1} = 50$$

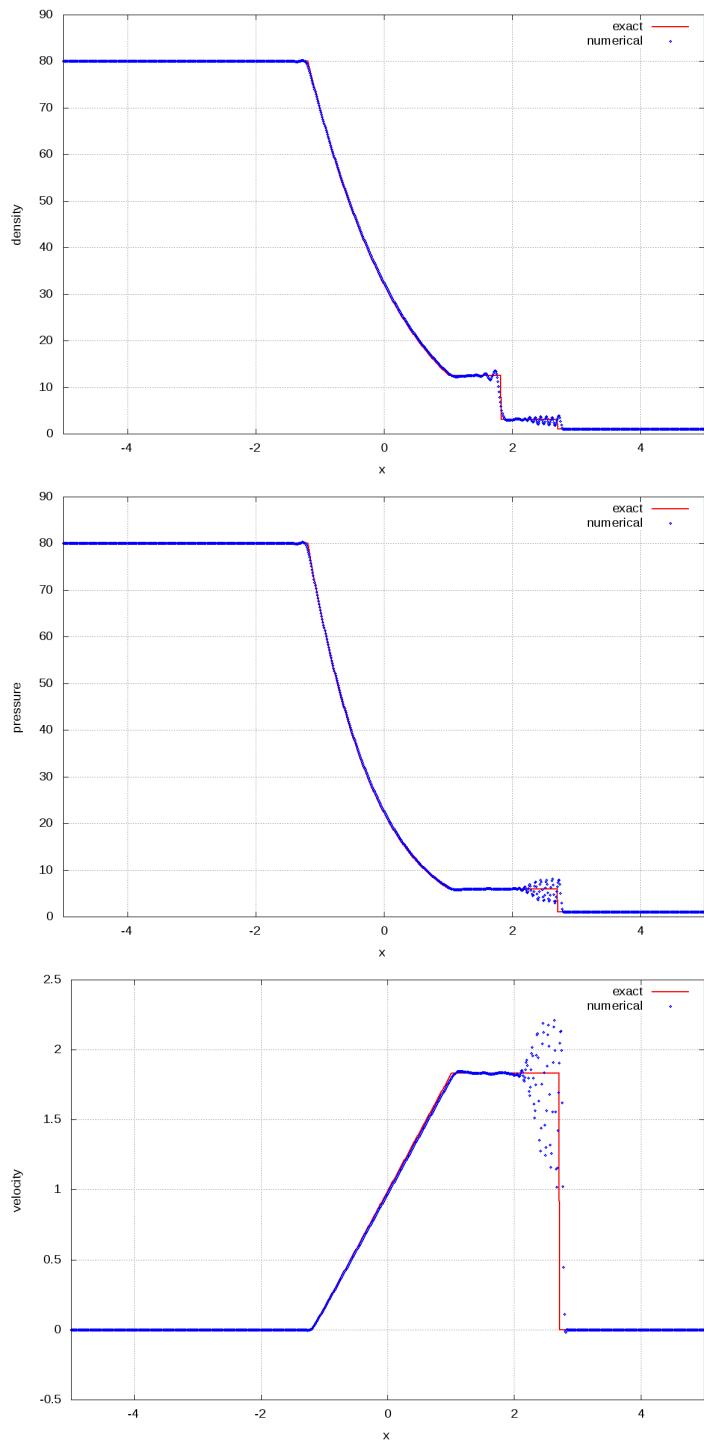


Figura 27: Resultados do esquema centrado com dissipação artificial não linear,

$$\frac{p_4}{p_1} = 80$$

3.5.2 Esquema de Lax-Wendroff

A forma do fluxo numérico para o método de Lax-Wendroff é

$$F_{j+\frac{1}{2}}^n = \underbrace{\frac{1}{2} (E_{j+1}^n + E_j^n)}_{\text{Fluxo numérico para esquema centrado}} - \underbrace{\frac{\Delta t}{2\Delta x} [A_{j+\frac{1}{2}} (E_{j+1}^n - E_j^n)] + D(Q_j)}_{\text{Termo de dissipação artificial}},$$

onde E_{j+1}^n e E_j^n são os valores do vetor de fluxo calculados nos pontos $j+1$ e j da malha, respectivamente, e $A_{j+\frac{1}{2}}$ é o jacobiano calculado no ponto de interface.

O jacobiano é definido como

$$A = \frac{\partial E}{\partial Q},$$

onde para o cálculo do mesmo nos pontos de interface, podemos assumir o valor médio entre os dois pontos de solução, obtendo

$$A_{j+\frac{1}{2}} = A\left(\frac{1}{2}(Q_{j+1} + Q_j)\right).$$

Os resultados são apresentados a seguir.

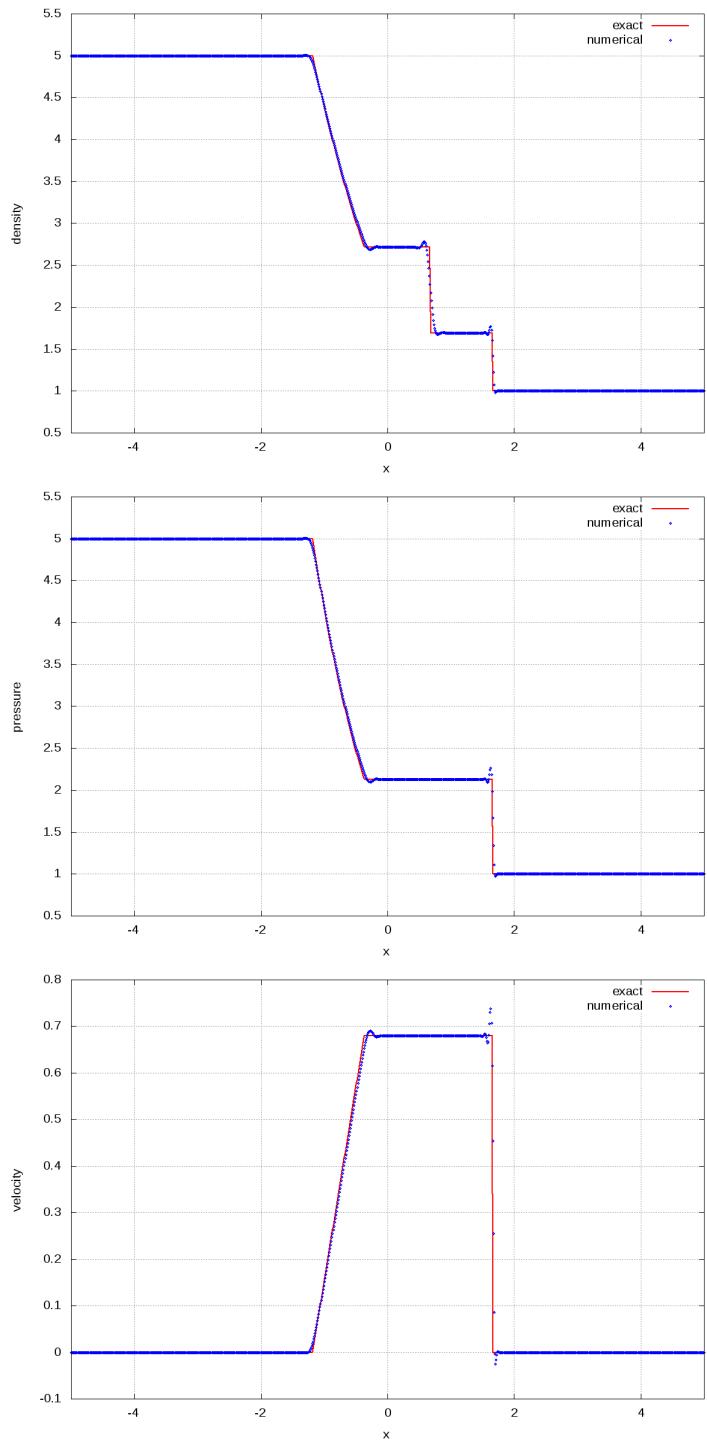


Figura 28: Resultados do esquema Lax-Wendroff, $\frac{p_4}{p_1} = 5$

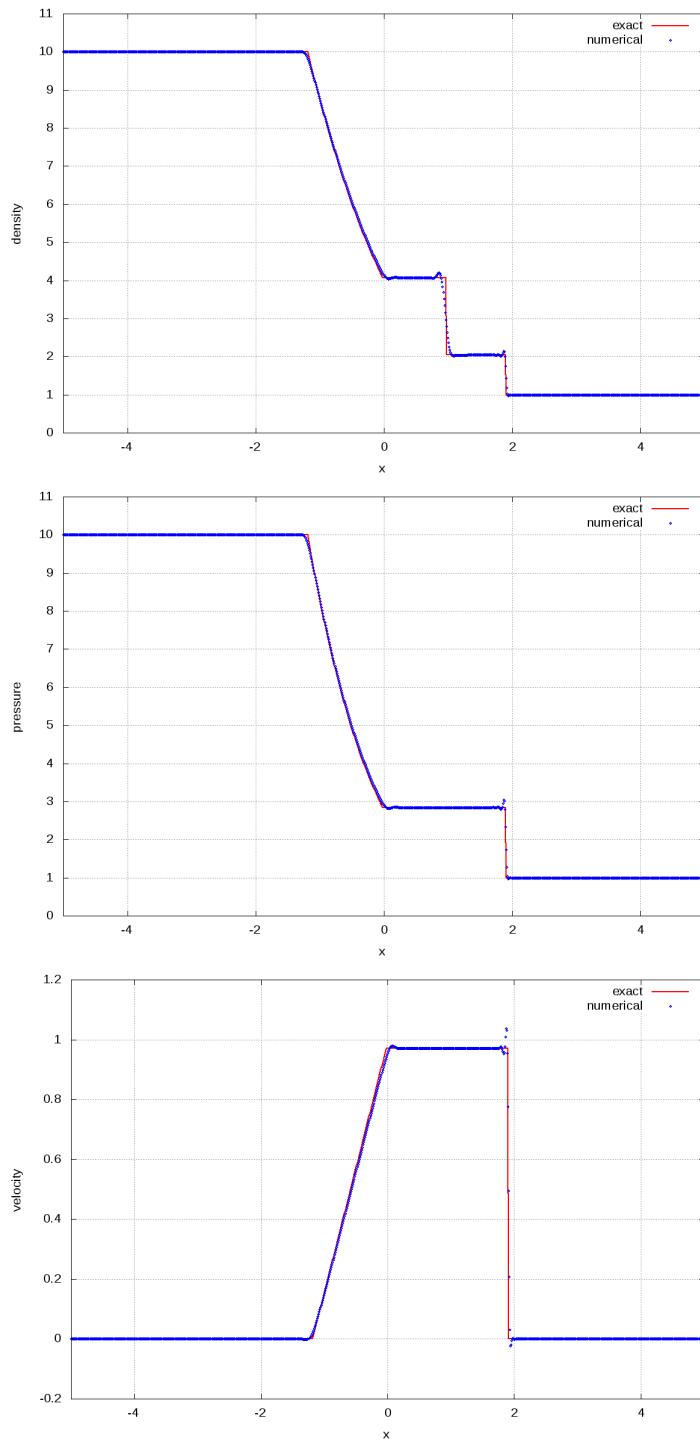


Figura 29: Resultados do esquema Lax-Wendroff, $\frac{p_4}{p_1} = 10$

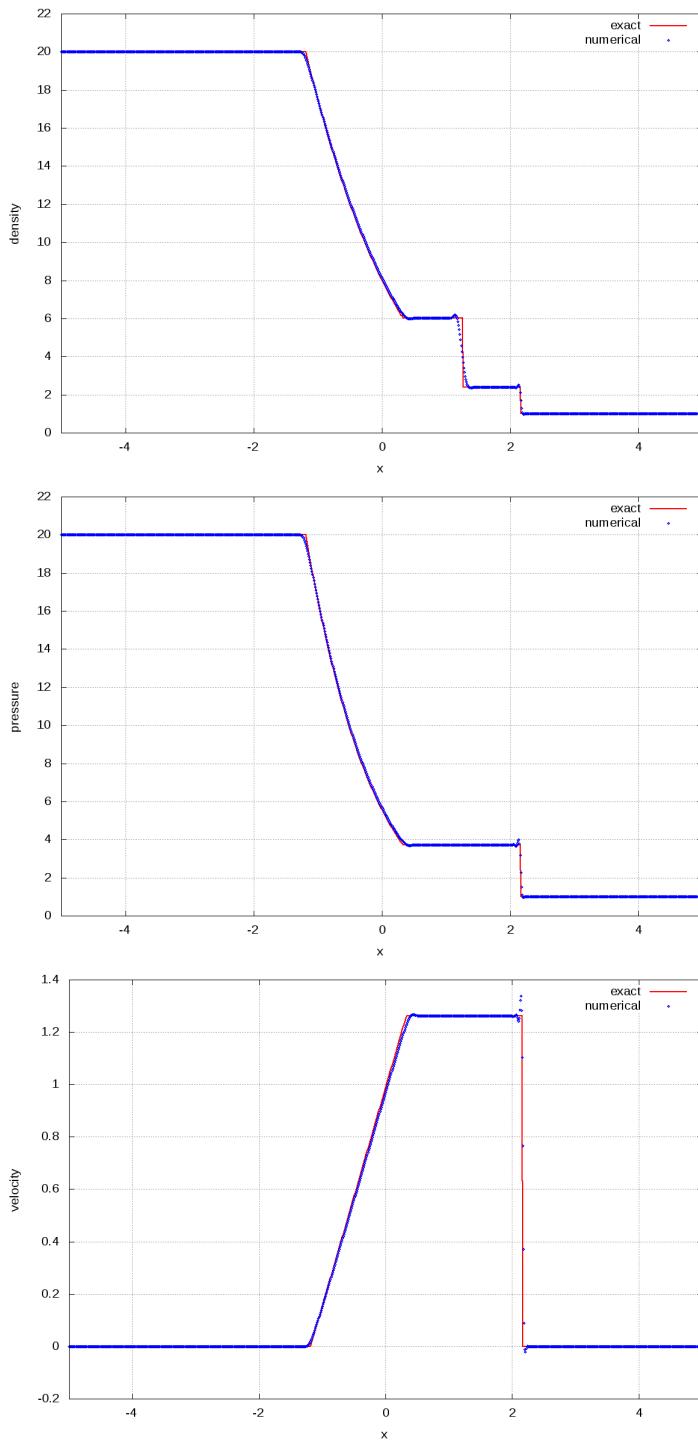


Figura 30: Resultados do esquema Lax-Wendroff, $\frac{p_4}{p_1} = 20$

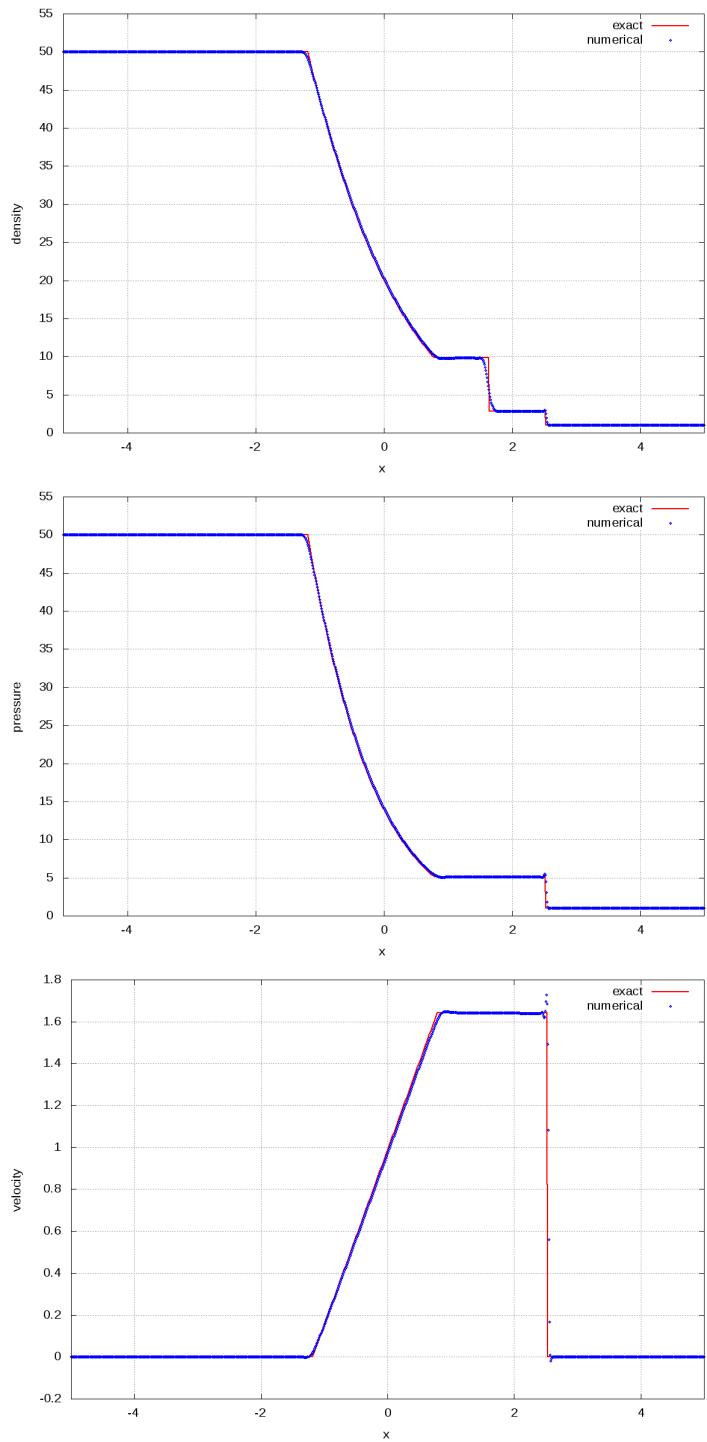


Figura 31: Resultados do esquema Lax-Wendroff, $\frac{p_4}{p_1} = 50$

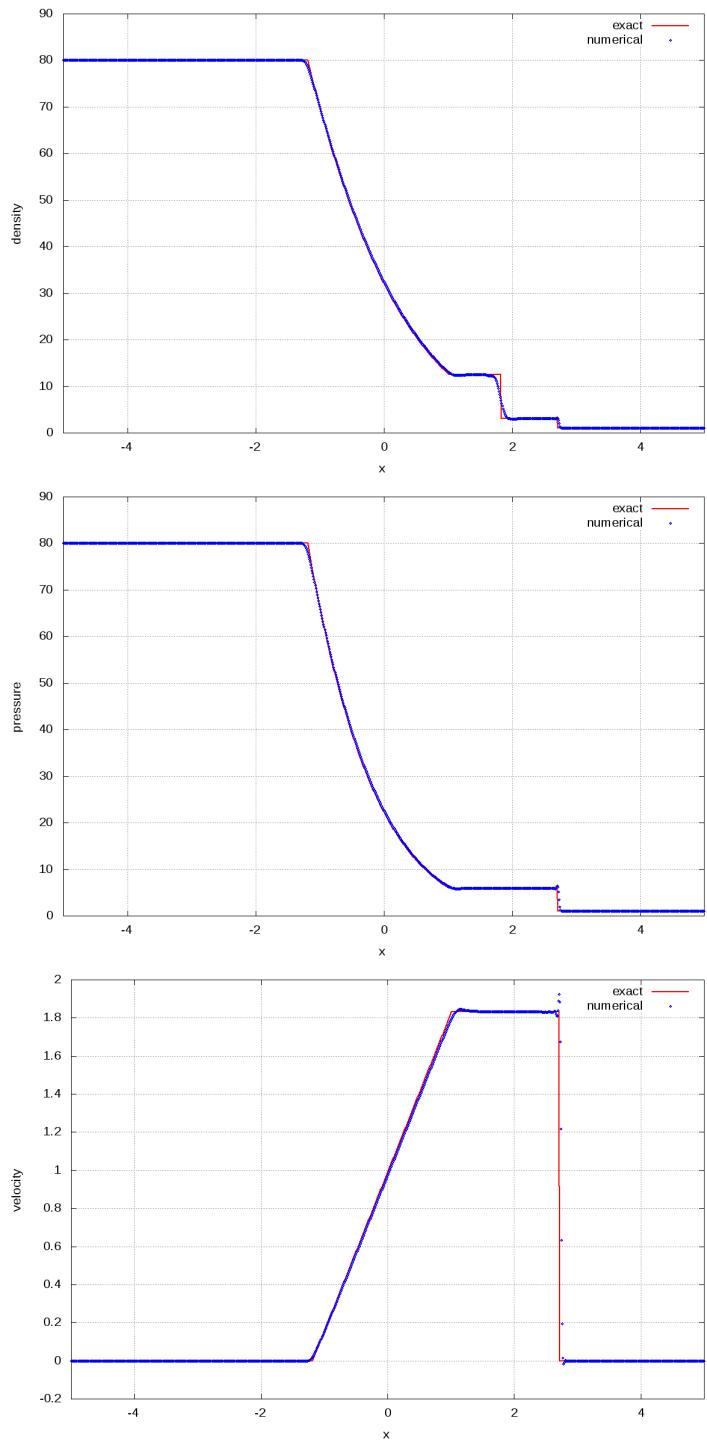


Figura 32: Resultados do esquema Lax-Wendroff, $\frac{p_4}{p_1} = 80$

3.5.3 Esquema Explícito de MacCormack

O esquema de MacCormack é derivado do esquema centrado de segunda ordem de Lax-Wendroff, aplicando-se passos de preditor-corretor. Desta forma, o esquema numérico do método de MacCormack é dado por:

Passo preditor:

$$\bar{Q}_j^{n+1} = Q_j^n - \frac{\Delta t}{\Delta x} (E_{j+1} - E_j) ,$$

Passo corretor:

$$Q_j^{n+1} = \frac{1}{2} (Q_j^n + \bar{Q}_j^{n+1} - \frac{\Delta t}{\Delta x} (E_j - E_{j-1})) + D(Q_j) .$$

Os resultados são apresentados a seguir.

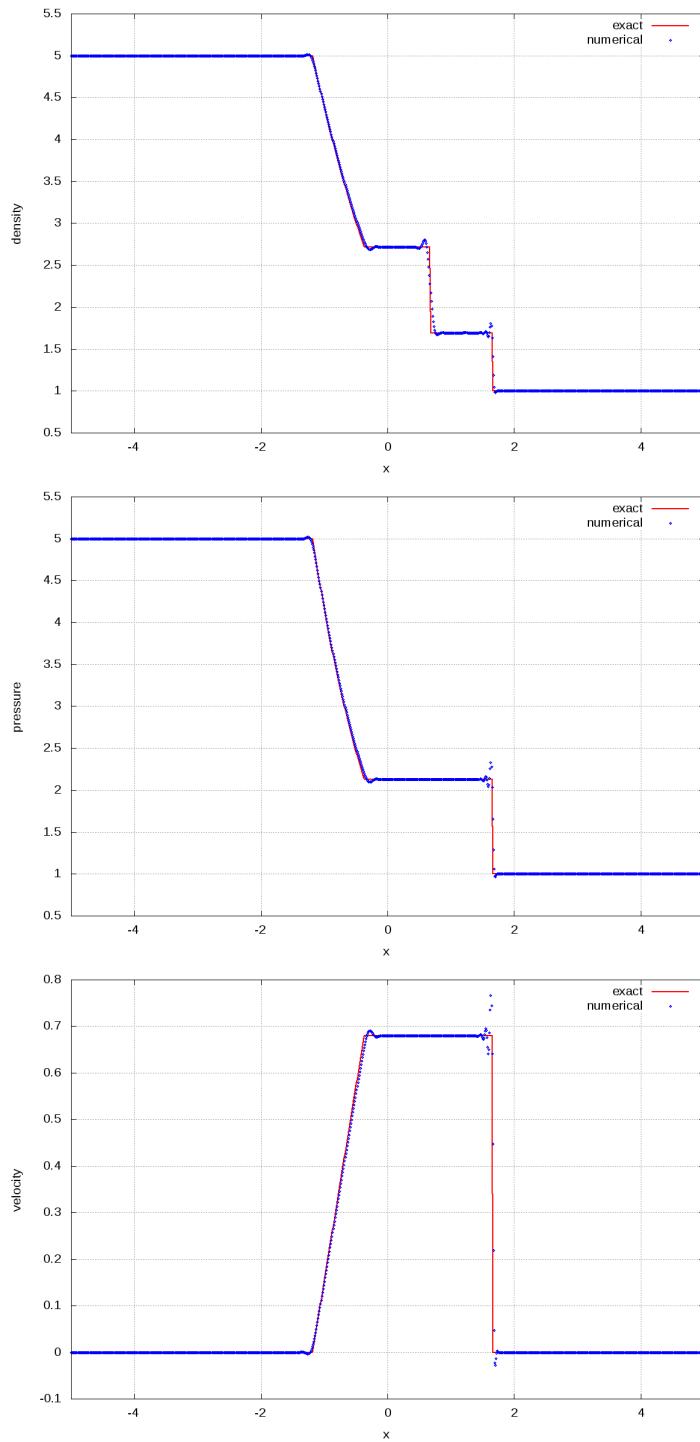


Figura 33: Resultados do esquema de MacCormack, $\frac{p_4}{p_1} = 5$

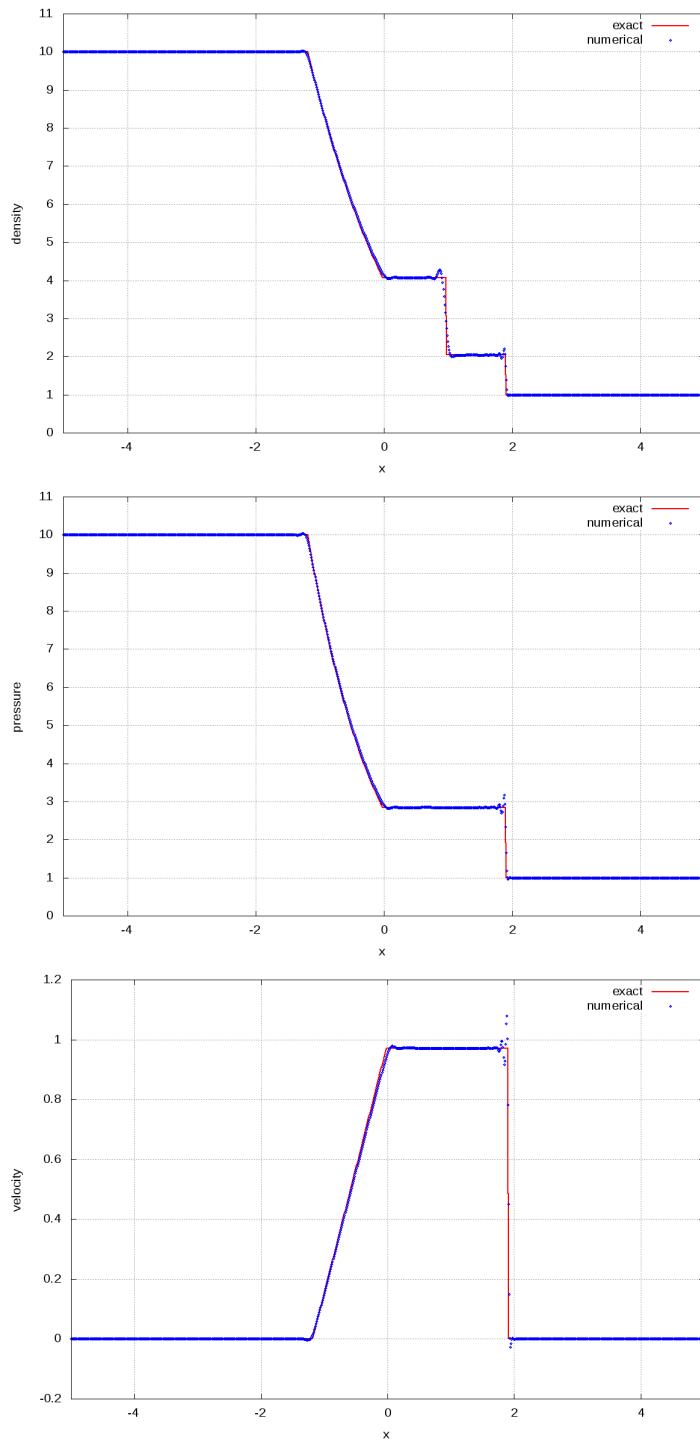


Figura 34: Resultados do esquema de MacCormack, $\frac{p_4}{p_1} = 10$

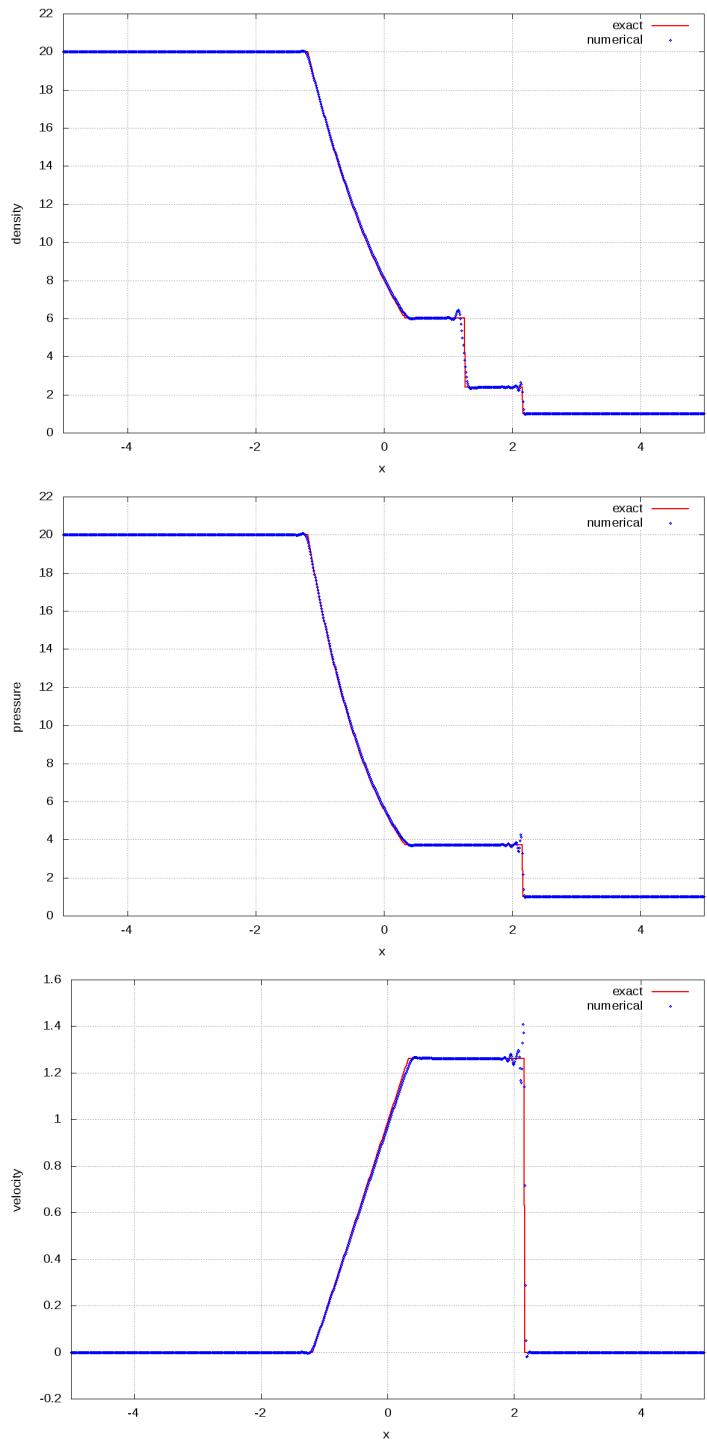


Figura 35: Resultados do esquema de MacCormack, $\frac{p_4}{p_1} = 20$

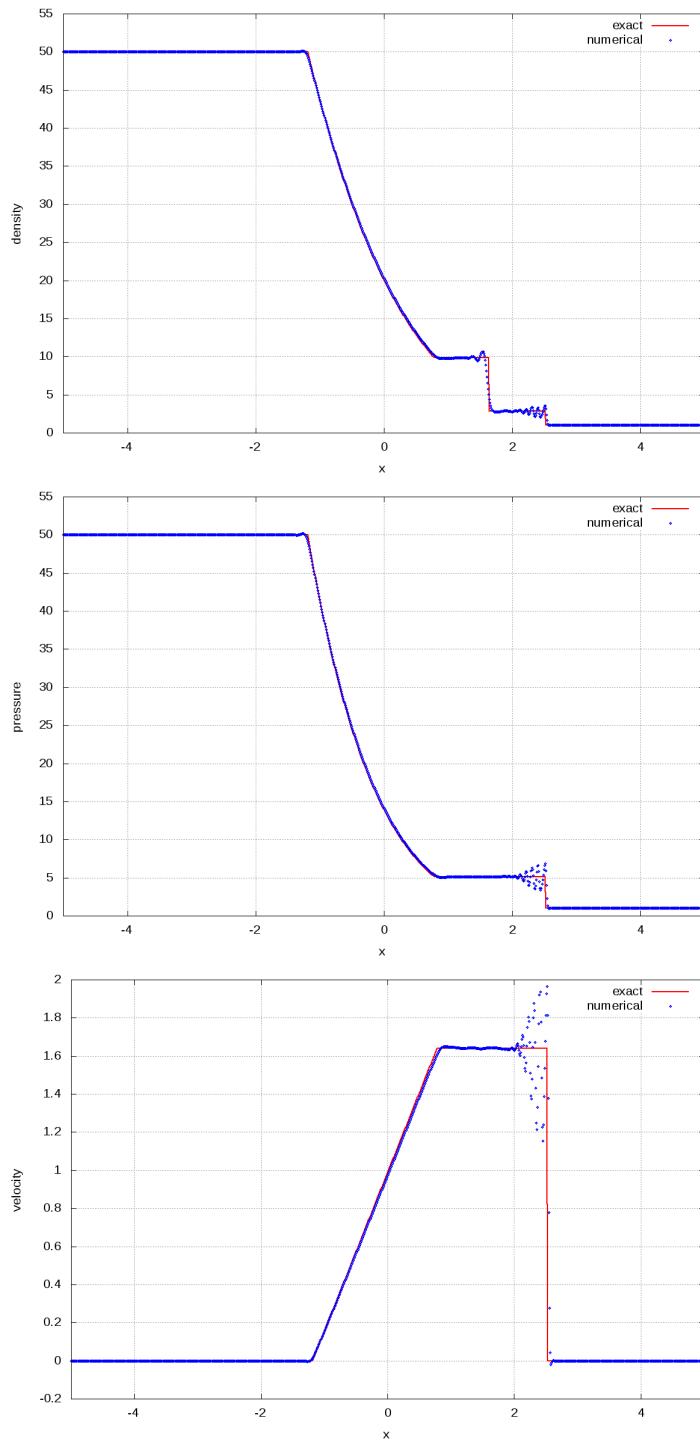


Figura 36: Resultados do esquema de MacCormack, $\frac{p_4}{p_1} = 50$

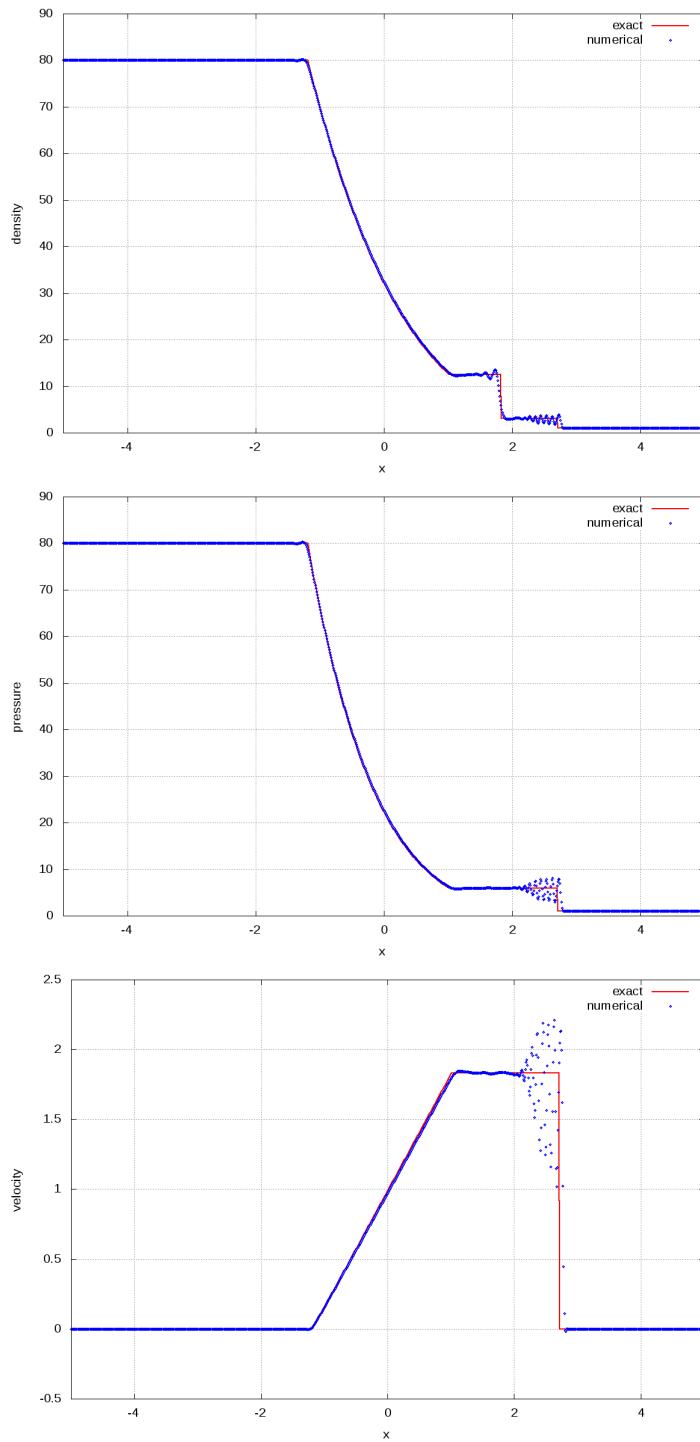


Figura 37: Resultados do esquema de MacCormack, $\frac{p_4}{p_1} = 80$

3.6 Projeto 2

Com o objetivo de se aplicar um esquema de discretização mais "inteligente", foram desenvolvidos os métodos chamados de Métodos de Separação de Vetores de Fluxo (do inglês *Flux Vector Splitting* - FVS). Estes métodos tem como principal característica o fato de literalmente separarem os vetores de fluxo, de acordo com a direção de propagação das características das equações sendo estudadas. Desta forma, podemos escrever o vetor de fluxo como sendo

$$E = E^+ + E^- ,$$

onde E^+ é o vetor de fluxo associado às características positivas, ≥ 0 , do problema, e E^- é o vetor de fluxo associado às características negativas, < 0 , do problema. Isto permite fazer uma discretização *one-sided* em ambos os vetores de fluxo, sendo aplicada uma discretização *backward* para E^+ , e uma discretização *forward* para E^- . Desta forma, por não estar mais anulando nenhuma das derivadas de ordem par provenientes da expansão em séries de Taylor das equações, não precisamos nos preocupar, a princípio, com termos de dissipação artificial, uma vez que numericamente já é introduzida dissipação pelo próprio processo de discretização.

A proposta do projeto 2 é resolver o problema do tubo de choque utilizando os seguintes métodos numéricos:

- Esquema de Separação de Vetores de Fluxo (FVS) de Steger & Warming;
- Esquema FVS non-MUSCL de van Leer;
- Esquema FVS de Liou - AUSM⁺;
- Método de Roe.

Cada um destes esquemas será descrito a seguir.

3.6.1 Esquema FVS de Steger & Warming

Dos esquemas do tipo FVS, o de Steger & Warming destaca-se por sua simplicidade. Para efetuar o processo de separação de vetores de fluxo, a proposta

feita por Steger & Warming é apenas separar os autovalores associados à solução da seguinte forma:

$$\lambda_i^\pm = \frac{1}{2} (\lambda_i \pm |\lambda_i|) .$$

Desta forma, os vetores de fluxo podem ser genericamente escritos como

$$E^\pm = \frac{\rho}{2\gamma} \begin{bmatrix} 2(\gamma - 1)\lambda_1^\pm + \lambda_2^\pm + \lambda_3^\pm \\ 2(\gamma - 1)\lambda_1^\pm u + \lambda_2^\pm(u + a) + \lambda_3^\pm(u - a) \\ (\gamma - 1)\lambda_1^\pm u^2 + \frac{\lambda_2^\pm}{2}(u - a)^2 + \frac{\lambda_3^\pm}{2}(u + a)^2 + \frac{(3-\gamma)(\lambda_2^\pm + \lambda_3^\pm)a^2}{2(\gamma-1)} \end{bmatrix} ,$$

e assim o método é escrito da seguinte forma

$$Q_j^{n+1} = Q_j^n - \frac{\Delta t}{\Delta x} \nabla E_j^+ - \frac{\Delta t}{\Delta x} \Delta E_j^- ,$$

onde $\nabla E_j^+ = E_j^+ - E_{j-1}^+$, e $\Delta E_j^- = E_{j+1}^- - E_j^-$. Os resultados são apresentados a seguir.

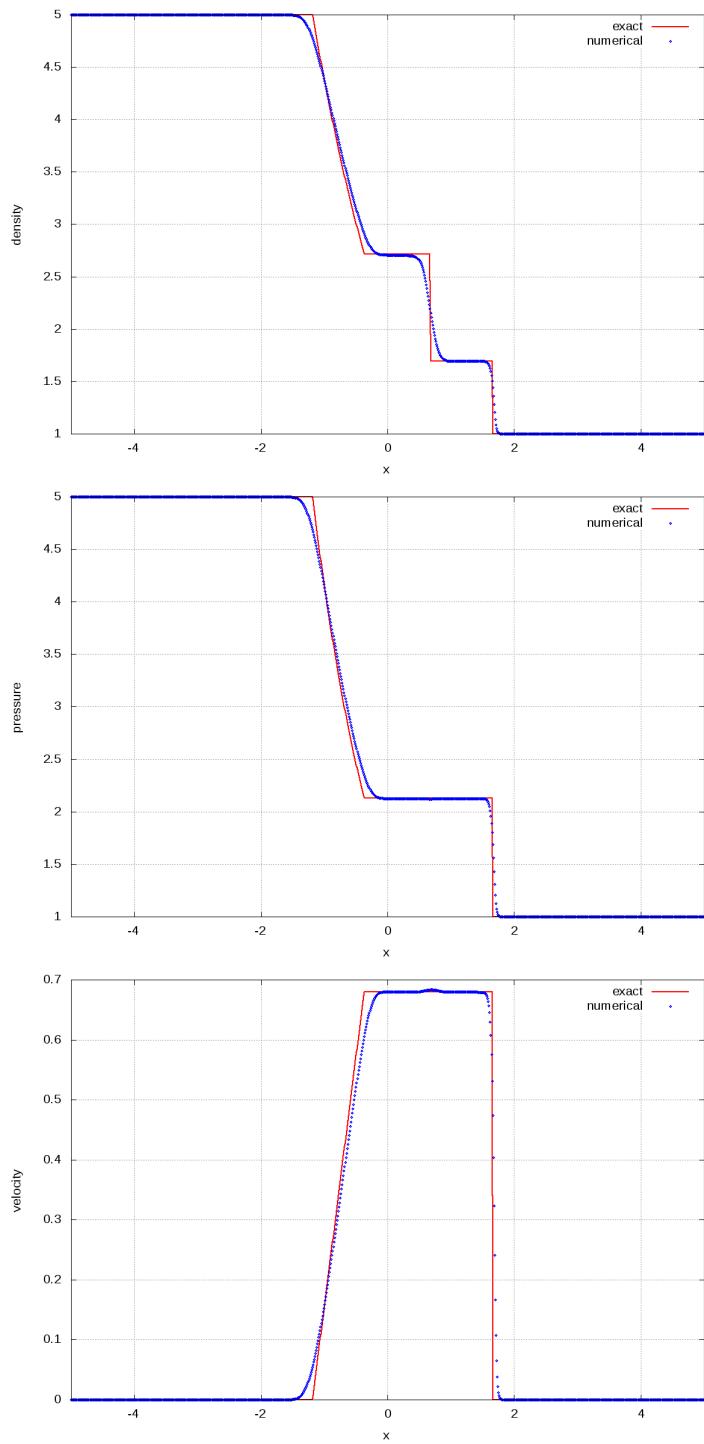


Figura 38: Resultados do esquema de Steger & Warming de primeira ordem,

$$\frac{p_4}{p_1} = 5$$

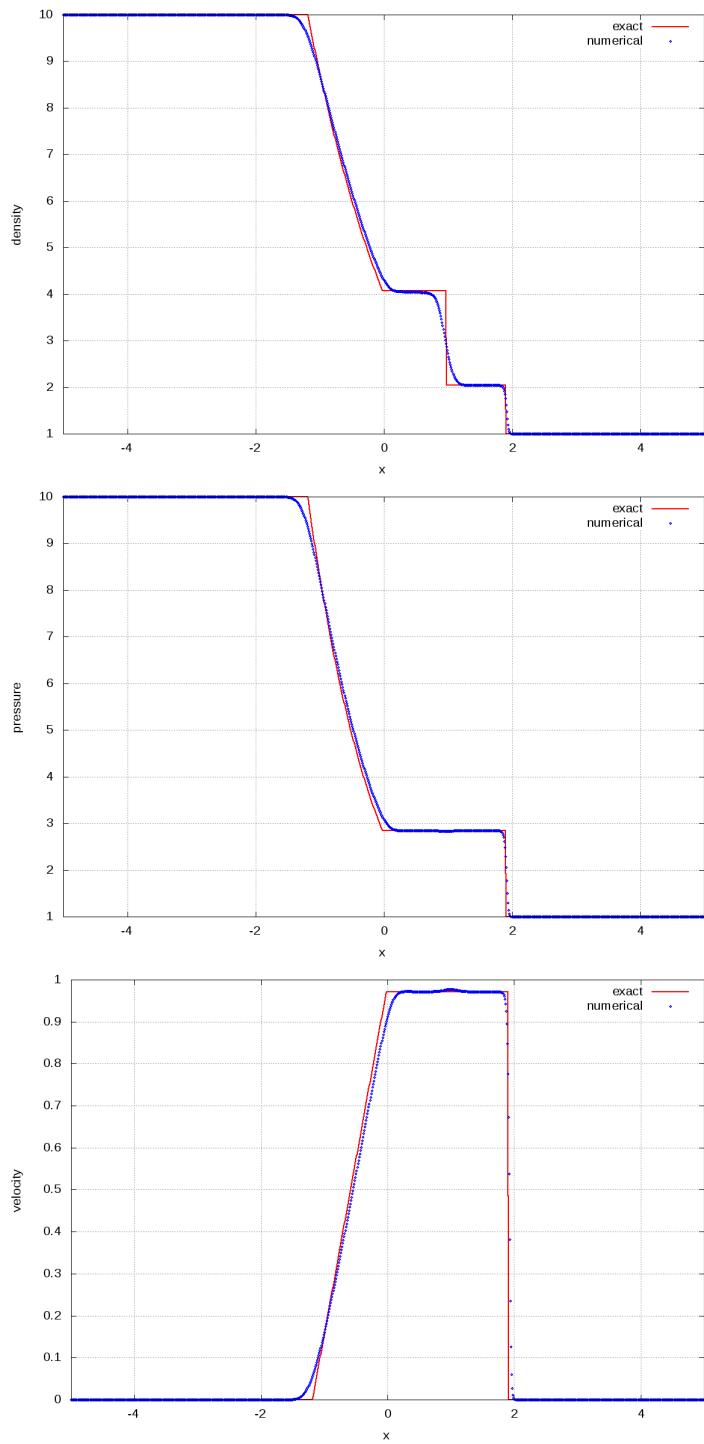


Figura 39: Resultados do esquema de Steger & Warming de primeira ordem,

$$\frac{p_4}{p_1} = 10$$

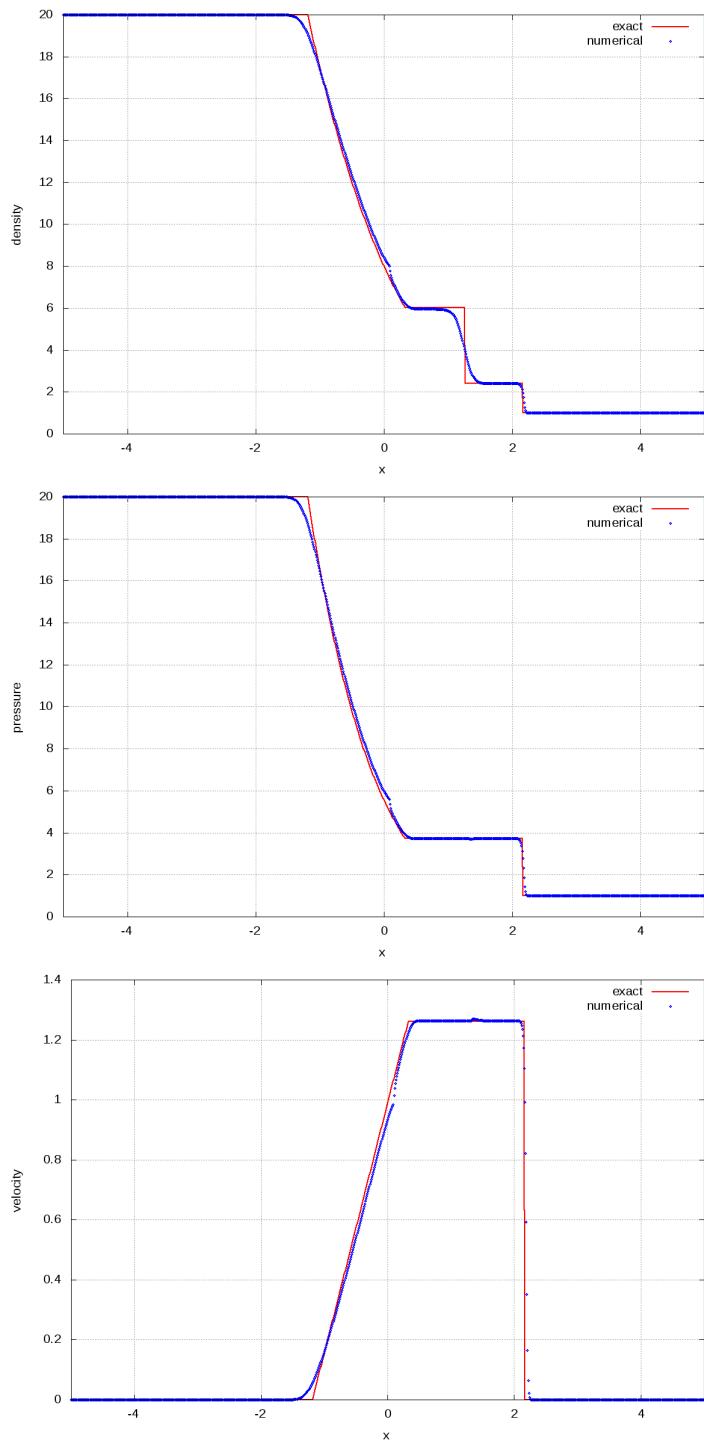


Figura 40: Resultados do esquema de Steger & Warming de primeira ordem,

$$\frac{p_4}{p_1} = 20$$

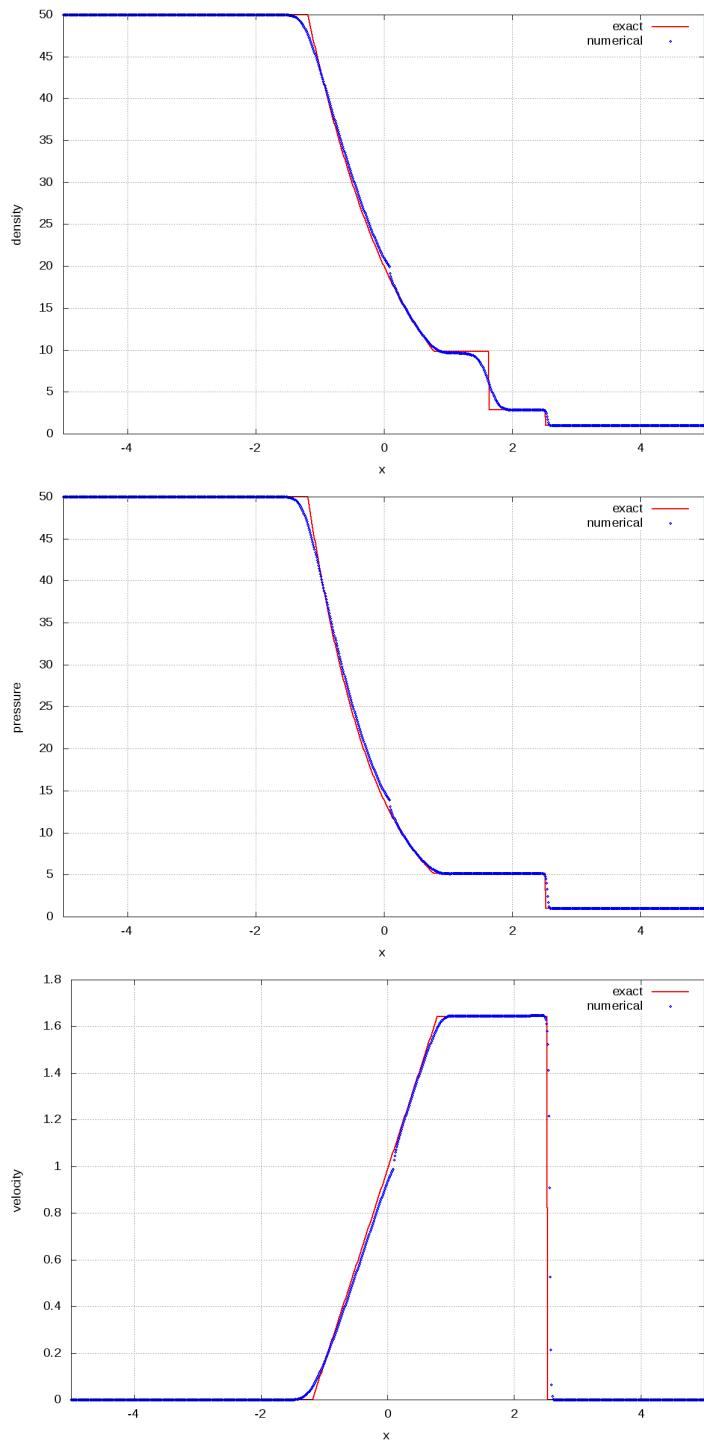


Figura 41: Resultados do esquema de Steger & Warming de primeira ordem,

$$\frac{p_4}{p_1} = 50$$

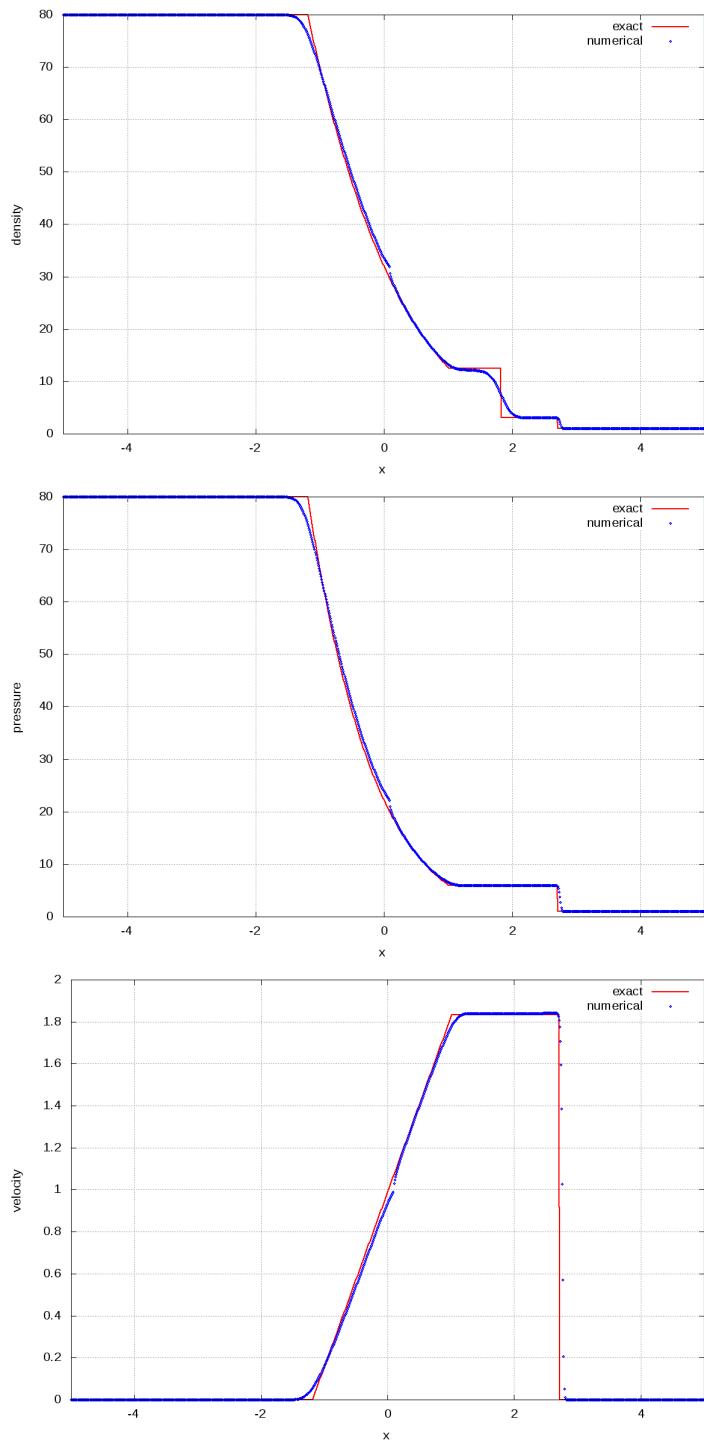


Figura 42: Resultados do esquema de Steger & Warming de primeira ordem,

$$\frac{p_4}{p_1} = 80$$

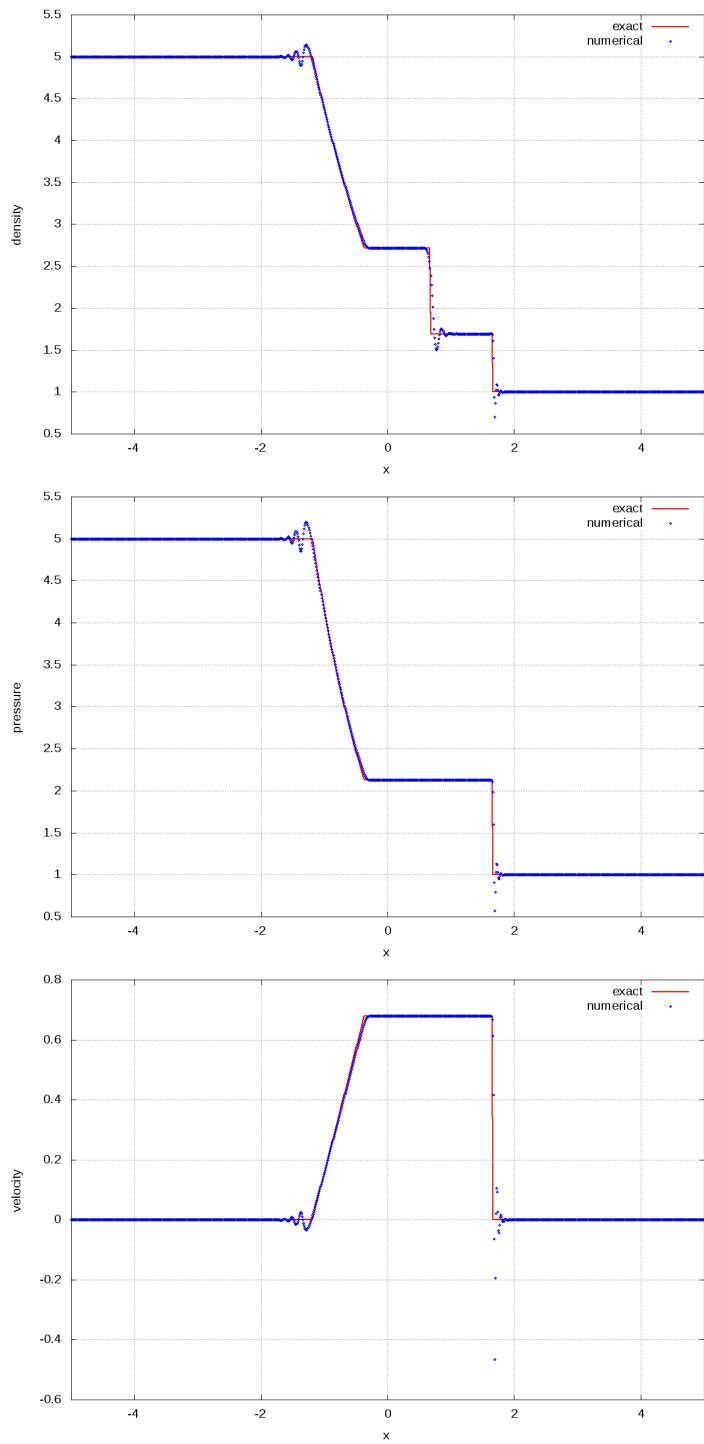


Figura 43: Resultados do esquema de Steger & Warming de segunda ordem,

$$\frac{p_4}{p_1} = 5$$

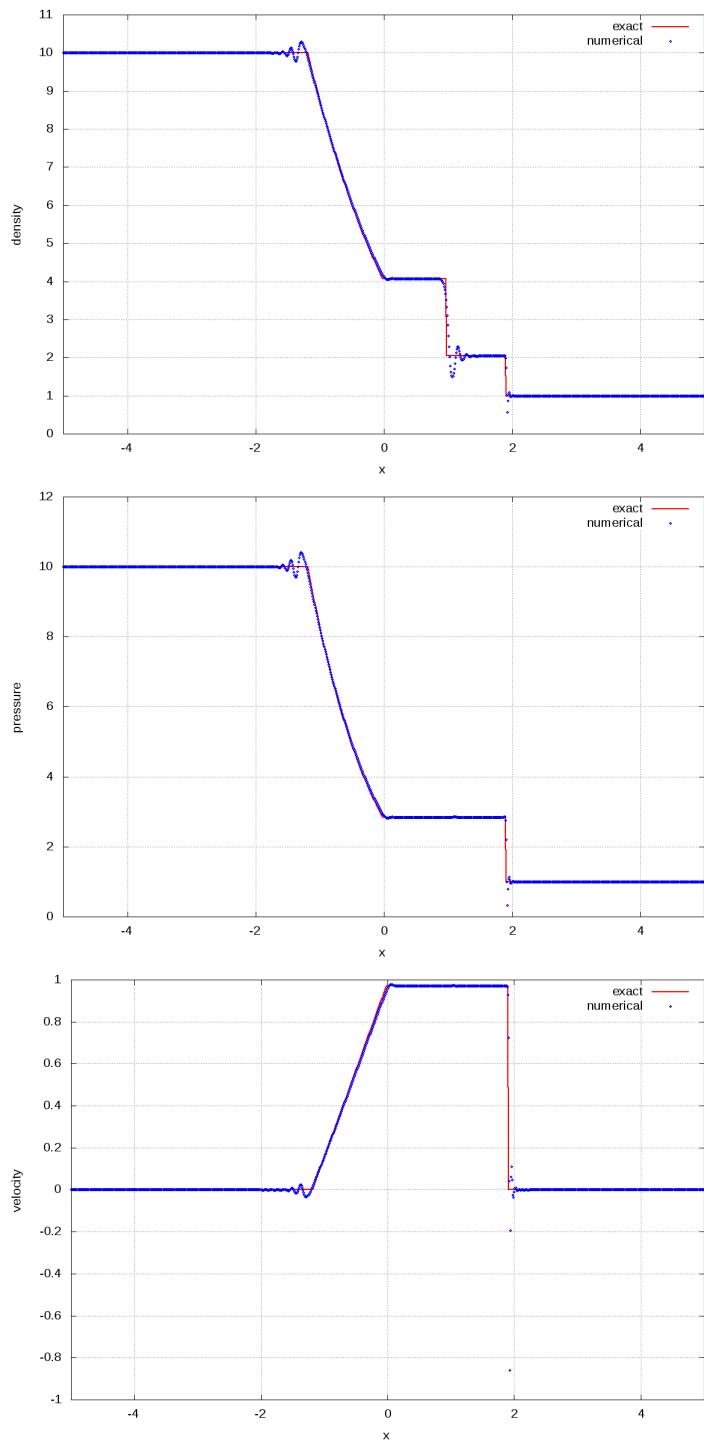


Figura 44: Resultados do esquema de Steger & Warming de segunda ordem,

$$\frac{p_4}{p_1} = 10$$

3.6.2 Esquema FVS de van Leer (non-MUSCL)

Com o surgimento da idéia de separação dos vetores de fluxo, van Leer observa que é importante se escolher a melhor maneira para efetuar esta separação. Isto se deve ao fato que, dependendo da forma como se é feita a separação dos vetores de fluxo, suas derivadas tornam-se não diferenciáveis em regiões onde os autovalores da solução trocam de sinal. Desta forma, van Leer propôs uma forma de se separar os vetores de fluxo, tal que as derivadas fossem continuamente diferenciáveis. Assim, os vetores de fluxo de van Leer podem ser genericamente escritos como

$$E^\pm = f_{mass}^\pm \begin{bmatrix} 1 \\ \frac{1}{\gamma} \cdot [(\gamma - 1)u_j \pm 2a_j] \\ \frac{1}{2} \frac{[(\gamma - 1)u_j \pm 2a_j]^2}{2(\gamma^2 - 1)} \end{bmatrix},$$

onde $f_{mass}^\pm = \pm \frac{\rho_j a_j}{4} (M_j \pm 1)^2$, sendo M_j o número de *Mach* no ponto j da malha. Entretanto, isto acaba implicando em uma condição de estabilidade um pouco mais restrita, sendo

$$CFL = \frac{\Delta t}{\Delta x} (|u| + a) \leq \frac{2\gamma + |M|(3 - \gamma)}{\gamma + 3}.$$

Os resultados são apresentados a seguir.

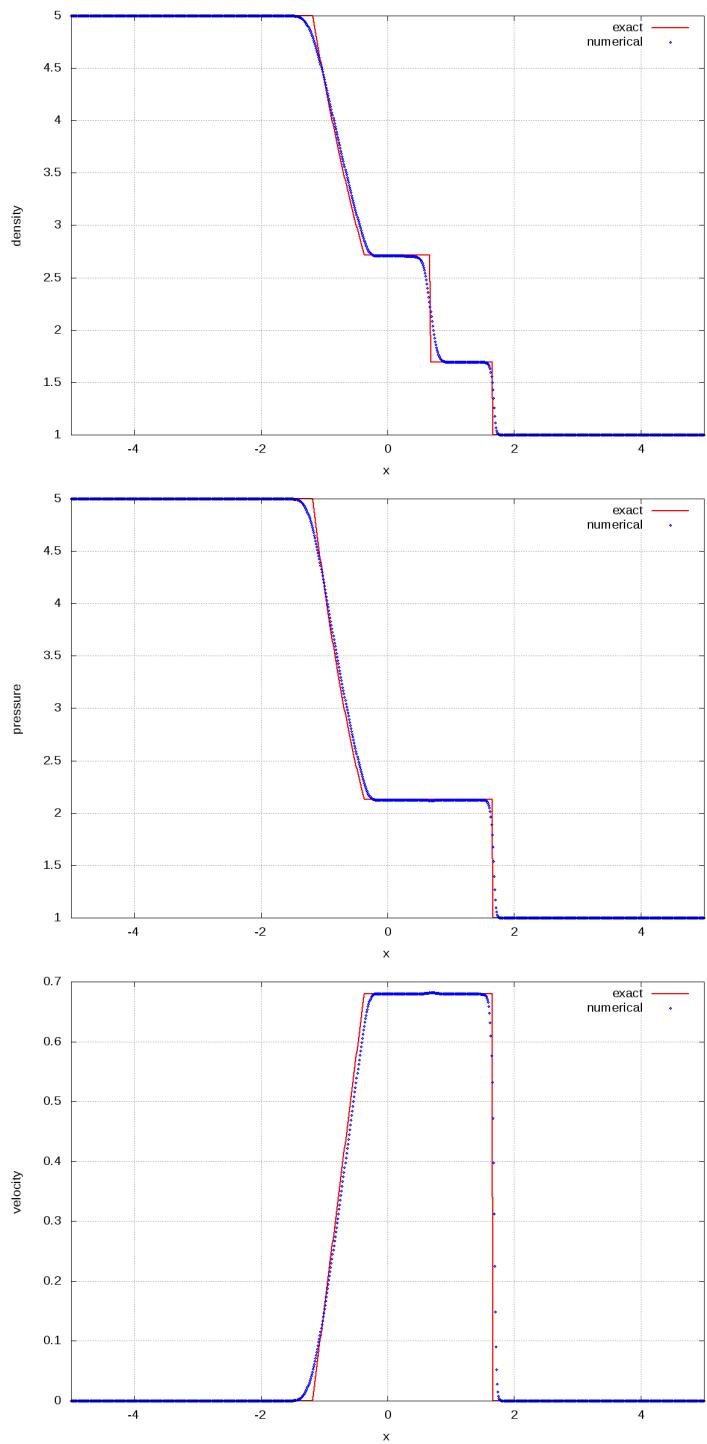


Figura 45: Resultados do esquema de van Leer de primeira ordem, $\frac{p_4}{p_1} = 5$

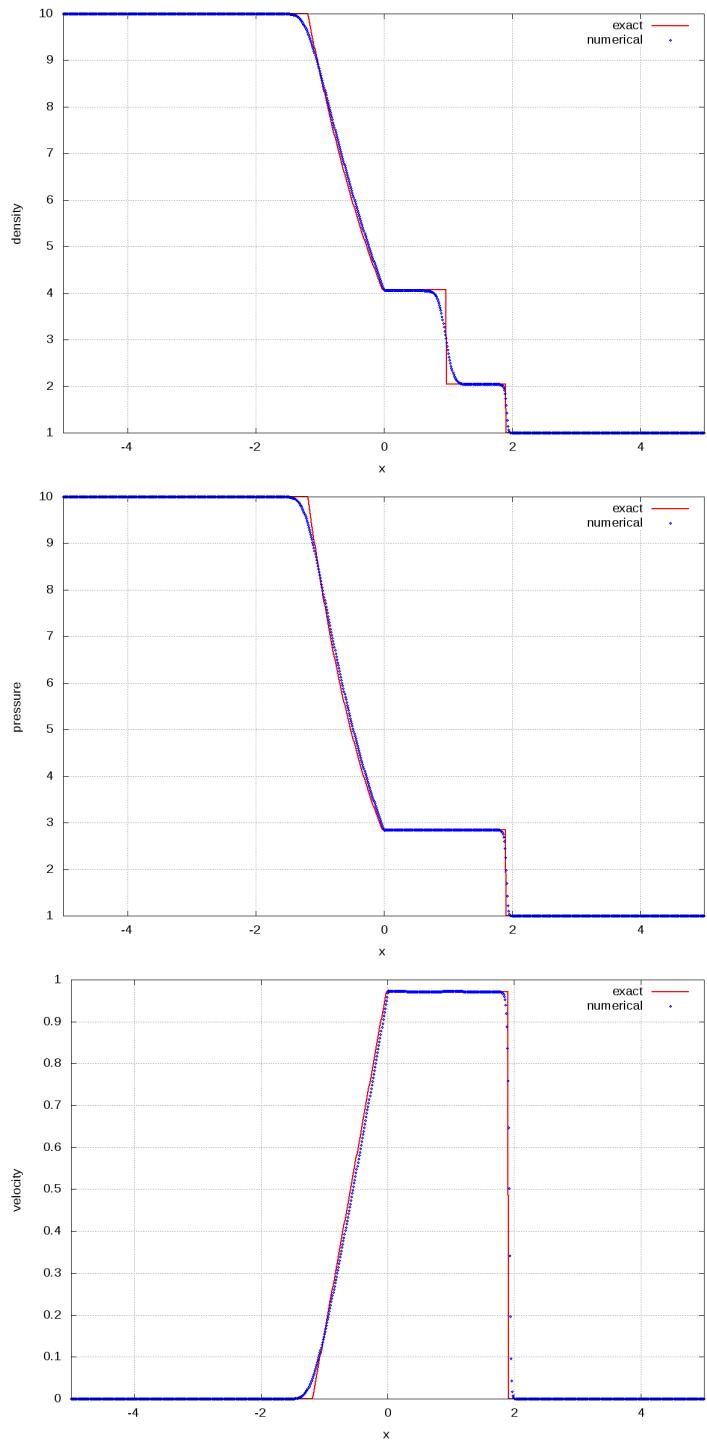


Figura 46: Resultados do esquema de van Leer de primeira ordem, $\frac{p_4}{p_1} = 10$

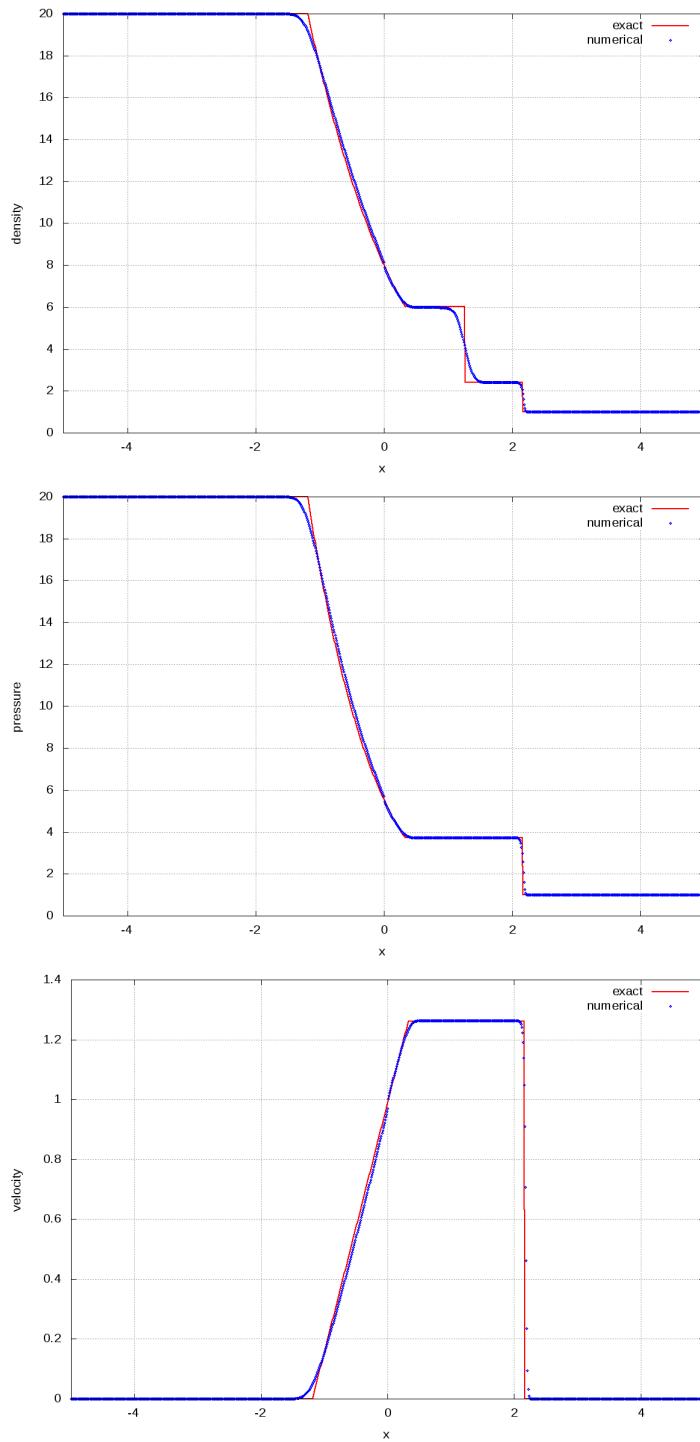


Figura 47: Resultados do esquema de van Leer de primeira ordem, $\frac{p_4}{p_1} = 20$

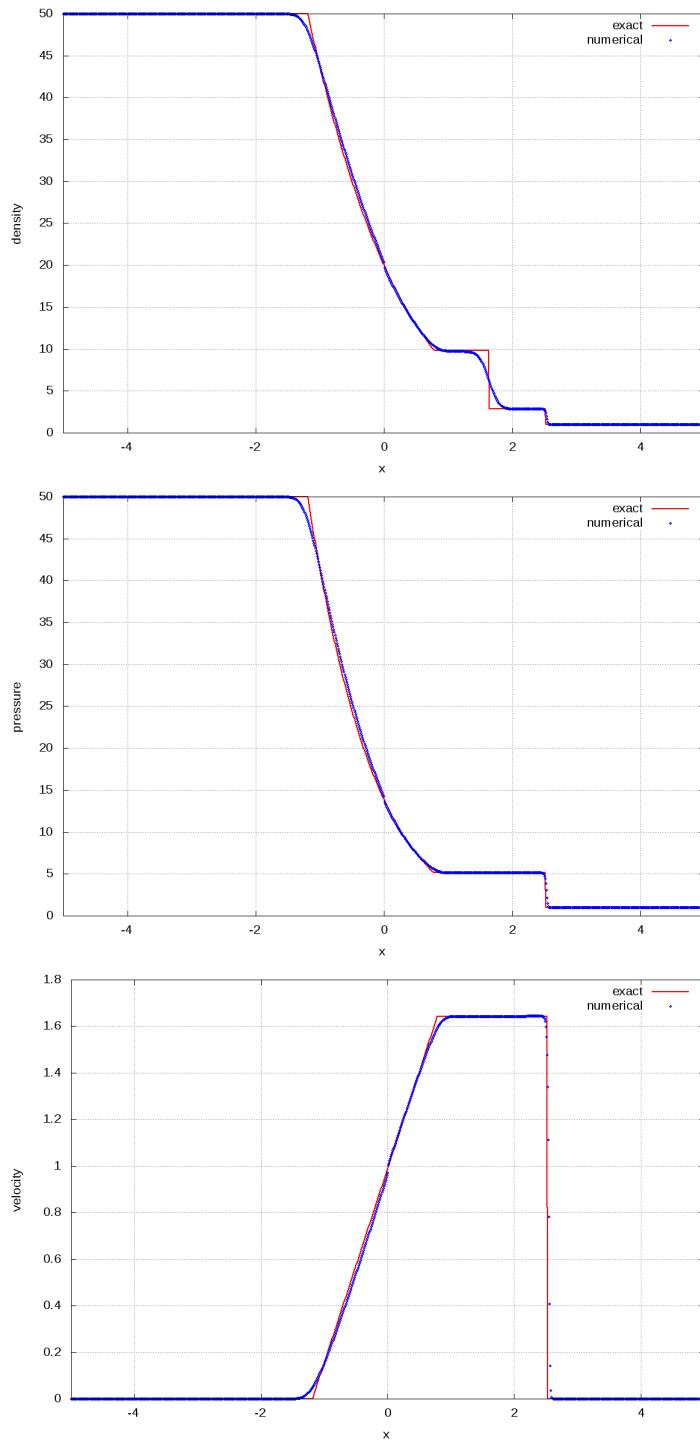


Figura 48: Resultados do esquema de van Leer de primeira ordem, $\frac{p_4}{p_1} = 50$

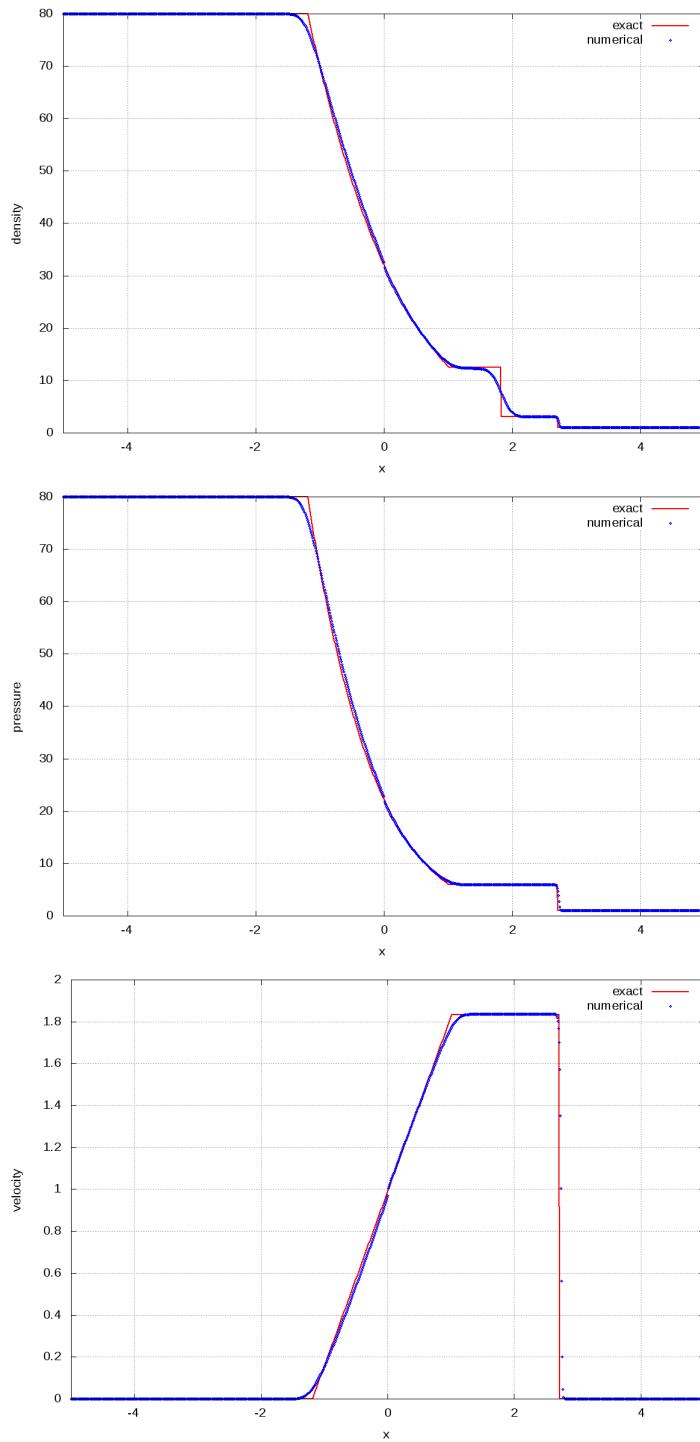


Figura 49: Resultados do esquema de van Leer de primeira ordem, $\frac{p_4}{p_1} = 80$

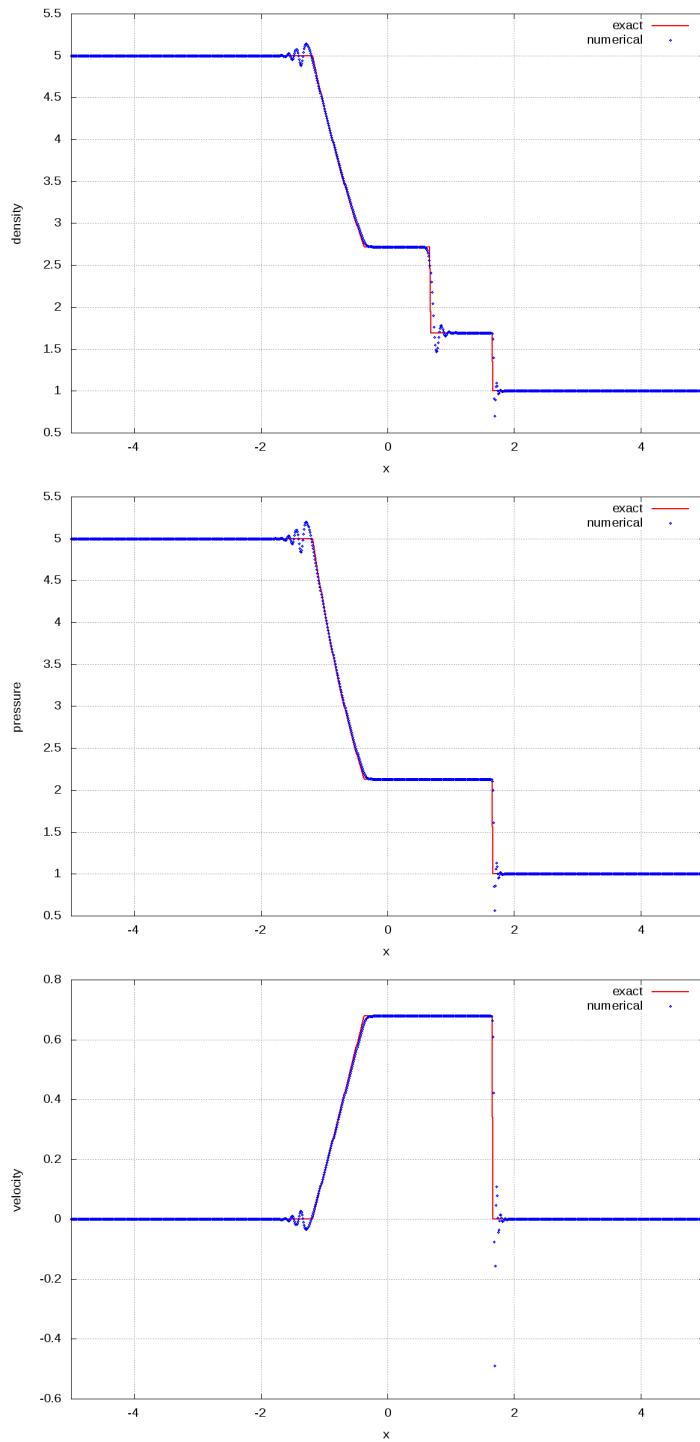


Figura 50: Resultados do esquema de van Leer de segunda ordem, $\frac{p_4}{p_1} = 5$

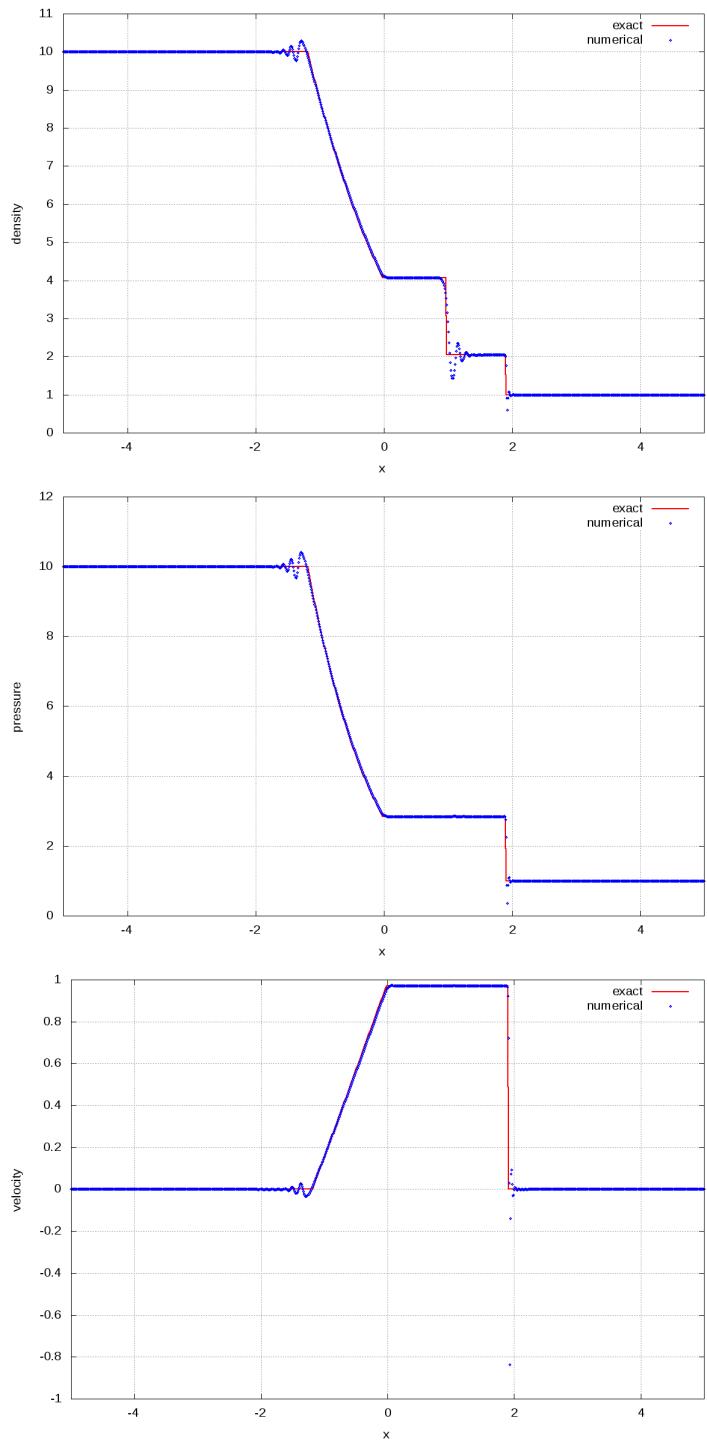


Figura 51: Resultados do esquema de van Leer de segunda ordem, $\frac{p_4}{p_1} = 10$

3.6.3 Esquema FVS de Liou (AUSM⁺)

A proposta de separação de vetores de fluxo de Liou, sugere que o fluxo numérico seja dividido em fluxo convectivo e mais um termo de pressão. Desta forma, o fluxo numérico pode ser escrito da seguinte forma

$$F_{j+\frac{1}{2}}^n = \frac{1}{2} M_{j+\frac{1}{2}} a_{j+\frac{1}{2}} (\Phi_{left} + \Phi_{right}) - \frac{1}{2} |M_{j+\frac{1}{2}}| a_{j+\frac{1}{2}} (\Phi_{right} - \Phi_{left}) + P_{j+\frac{1}{2}},$$

onde Φ é o vetor de variáveis conservadas, Q , reescrito da seguinte forma

$$\Phi = \begin{bmatrix} \rho \\ \rho u \\ \rho H \end{bmatrix},$$

e $P_{j+\frac{1}{2}}$ é definido como

$$P_{j+\frac{1}{2}} = \begin{bmatrix} 0 \\ p_{j+\frac{1}{2}} \\ 0 \end{bmatrix}.$$

Neste contexto, entenda-se os subscritos *left* e *right* como sendo a propriedade em j e $j+1$. A pressão $p_{j+\frac{1}{2}}$ é definida como

$$p_{j+\frac{1}{2}} = p_{left}^+ \cdot p_{left}^- + p_{right}^- \cdot p_{right}^+,$$

definindo assim

$$p_j^\pm = \begin{cases} \frac{1}{2}(1 \pm sign(M_j)), & \text{se } |M_j| \geq 1 \\ p_\alpha^\pm, & \text{se } |M_j| < 1 \end{cases},$$

onde

$$p_\alpha^\pm = \frac{1}{4}(M_j \pm 1)^2(2 \mp M_j) \pm \alpha M_j (M_j^2 - 1)^2.$$

A definição do número de Mach na interface também é dada por

$$M_{j+\frac{1}{2}} = M_{left}^+ + M_{right}^-,$$

$$M_{left} = \frac{u_j}{a_{j+\frac{1}{2}}} \quad , \quad M_{right} = \frac{u_{j+1}}{a_{j+\frac{1}{2}}}.$$

Para o cálculo da velocidade do som na interface, Liou faz algumas sugestões, sendo que a que apresentou melhores resultados foi

$$\begin{aligned} a_{j+\frac{1}{2}} &= \min(\tilde{a}_j, \tilde{a}_{j+1}) , \\ \tilde{a}_j &= a_j^* \cdot \min\left(1, \frac{a_j^*}{|u_j|}\right) , \\ a_j^* &= \sqrt{\frac{2(\gamma - 1)}{(\gamma + 1)} \cdot H_j} , \\ M^\pm &= \begin{cases} \frac{1}{2}(M \pm |M|), & \text{se } |M| \geq 1 \\ M_\beta^\pm(M), & \text{se } |M| < 1 \end{cases} \end{aligned}$$

onde M_β^\pm é definido como

$$M_\beta^\pm = \pm \frac{1}{4}(M \pm 1)^2 \pm \beta(M^2 - 1)^2 .$$

Nas definições de p_α e M_β , os valores definidos por Liou para as constantes $\alpha = 3/16$ e $\beta = 1/8$ são os mesmos utilizados neste projeto. Os resultados são apresentados a seguir.

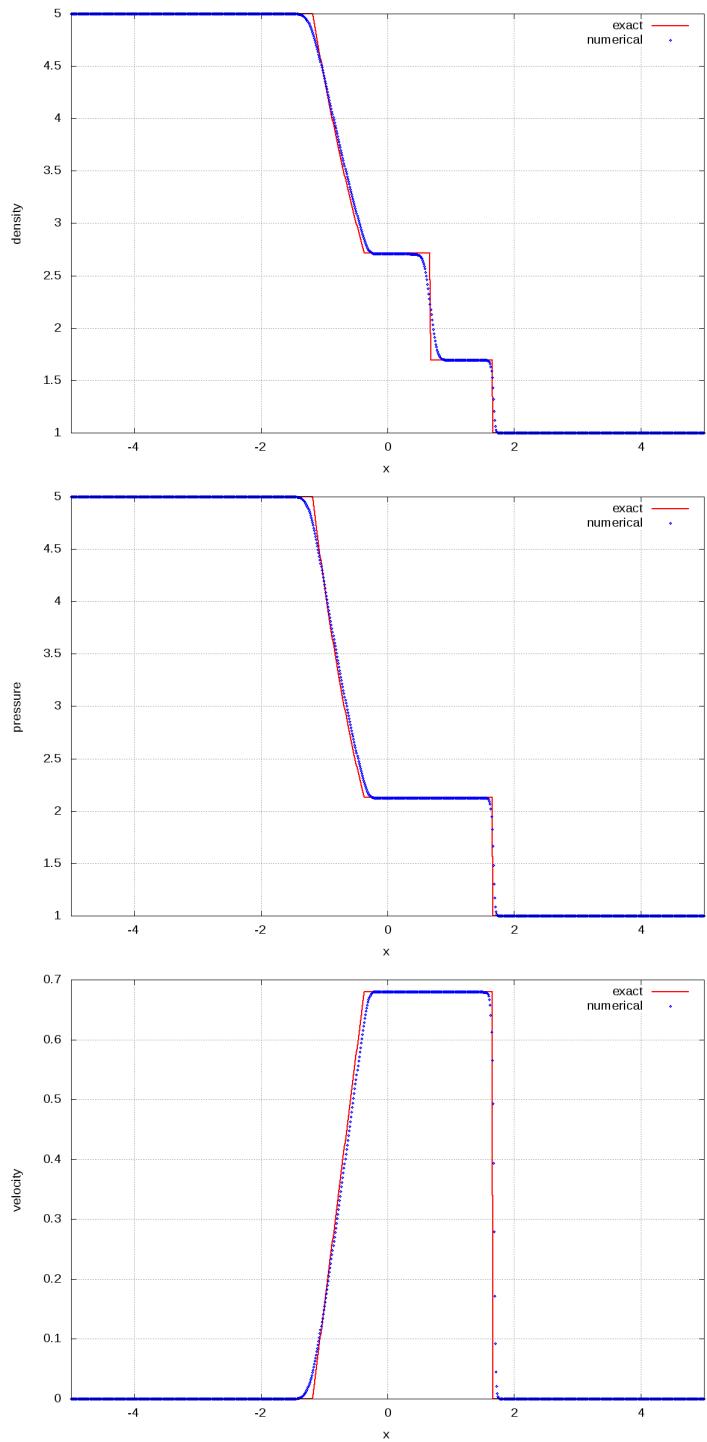


Figura 52: Resultados do esquema AUSM⁺, $\frac{p_4}{p_1} = 5$

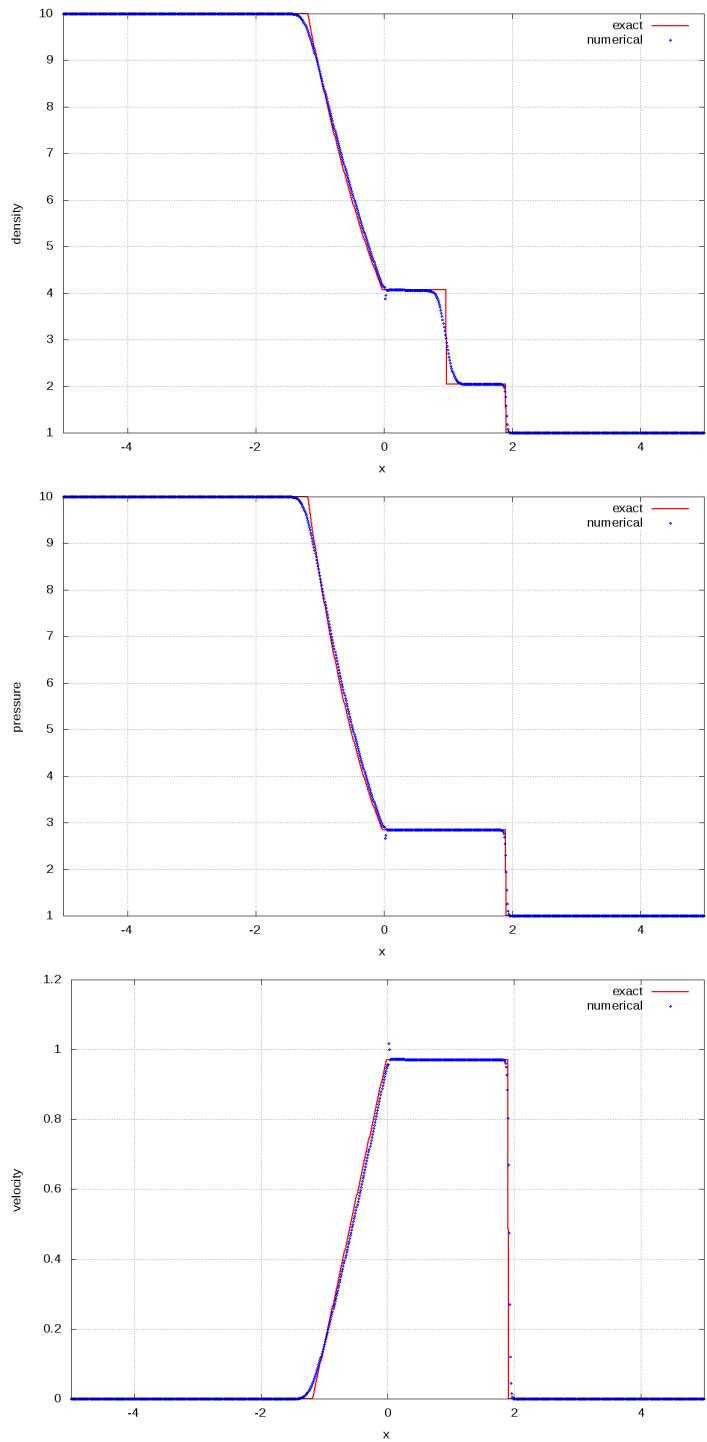


Figura 53: Resultados do esquema AUSM⁺, $\frac{p_4}{p_1} = 10$

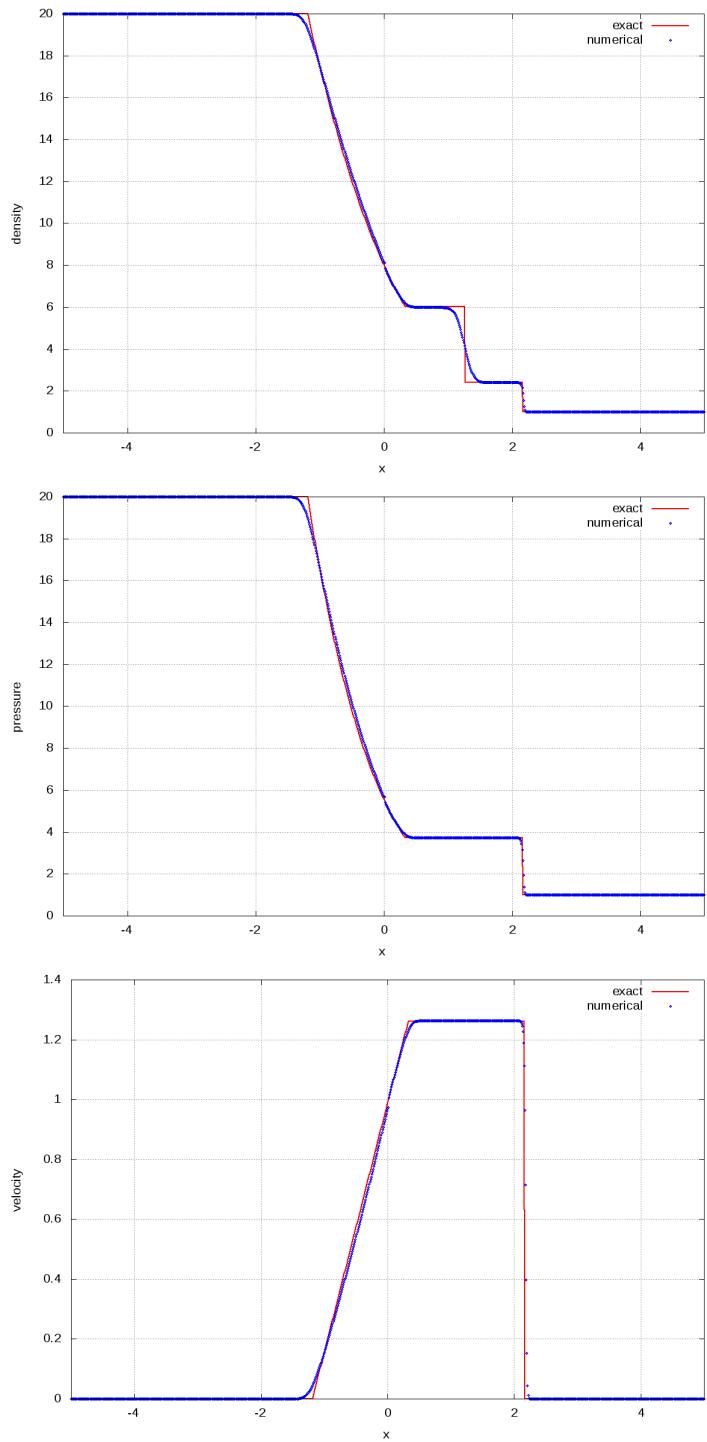


Figura 54: Resultados do esquema AUSM⁺, $\frac{p_4}{p_1} = 20$

3.6.4 Método de Roe

O método de Roe resolve de forma exata um problema de Riemann aproximado, daí a origem do nome do método *Approximate Riemann Solver*. O método consiste em se utilizar um vetor de parâmetros para descrever os vetores de fluxo. Desta forma, temos o seguinte fluxo numérico, para o caso das equações de Euler unidimensionais

$$\begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix}_{j+\frac{1}{2}} = \frac{1}{2} \left[E_j + E_{j+1} - \sum_{k=1}^3 |\lambda_k| \alpha_k \cdot R_k \right] ,$$

onde λ_i são os autovalores associados, α_k são as propriedades características e R_k são os autovetores vindos pela direita. Os autovetores R_k são definidos como

$$R_1 = \begin{bmatrix} 1 \\ \tilde{u} - \tilde{a} \\ \tilde{H} - \tilde{u} \cdot \tilde{a} \end{bmatrix} , \quad R_2 = \begin{bmatrix} 1 \\ \tilde{u} \\ \frac{1}{2}\tilde{u}^2 \end{bmatrix} , \quad R_3 = \begin{bmatrix} 1 \\ \tilde{u} + \tilde{a} \\ \tilde{H} + \tilde{u} \cdot \tilde{a} \end{bmatrix} ,$$

possuindo os seguintes autovalores associados

$$\lambda_1 = u - a , \quad \lambda_2 = u , \quad \lambda_3 = u + a .$$

As variáveis \tilde{u} , \tilde{H} e \tilde{a} são as variáveis com médias, definidas por Roe como

$$\begin{aligned} \tilde{u}_j &= \frac{\sqrt{\rho_j} \cdot u_j + \sqrt{\rho_{j+1}} \cdot u_{j+1}}{\sqrt{\rho_j} + \sqrt{\rho_{j+1}}} , \\ \tilde{H}_j &= \frac{\sqrt{\rho_j} \cdot H_j + \sqrt{\rho_{j+1}} \cdot H_{j+1}}{\sqrt{\rho_j} + \sqrt{\rho_{j+1}}} , \\ \tilde{a}_j &= \sqrt{(\gamma - 1) \left(\tilde{H}_j - \frac{1}{2} \tilde{u}_j^2 \right)} . \end{aligned}$$

As propriedades características podem ser escritas como

$$\alpha_1 = \frac{1}{2\tilde{a}} [\delta p - (\rho \cdot \tilde{a}) \delta u] , \quad \alpha_2 = -\frac{[\delta p - \tilde{a}^2 \delta \rho]}{\tilde{a}^2} , \quad \alpha_3 = \frac{1}{2\tilde{a}} [\delta p + (\rho \cdot \tilde{a}) \delta u] ,$$

onde o operador $\delta(\cdot) = (\cdot)_{j+1} - (\cdot)_j$. Os resultados são apresentados a seguir.

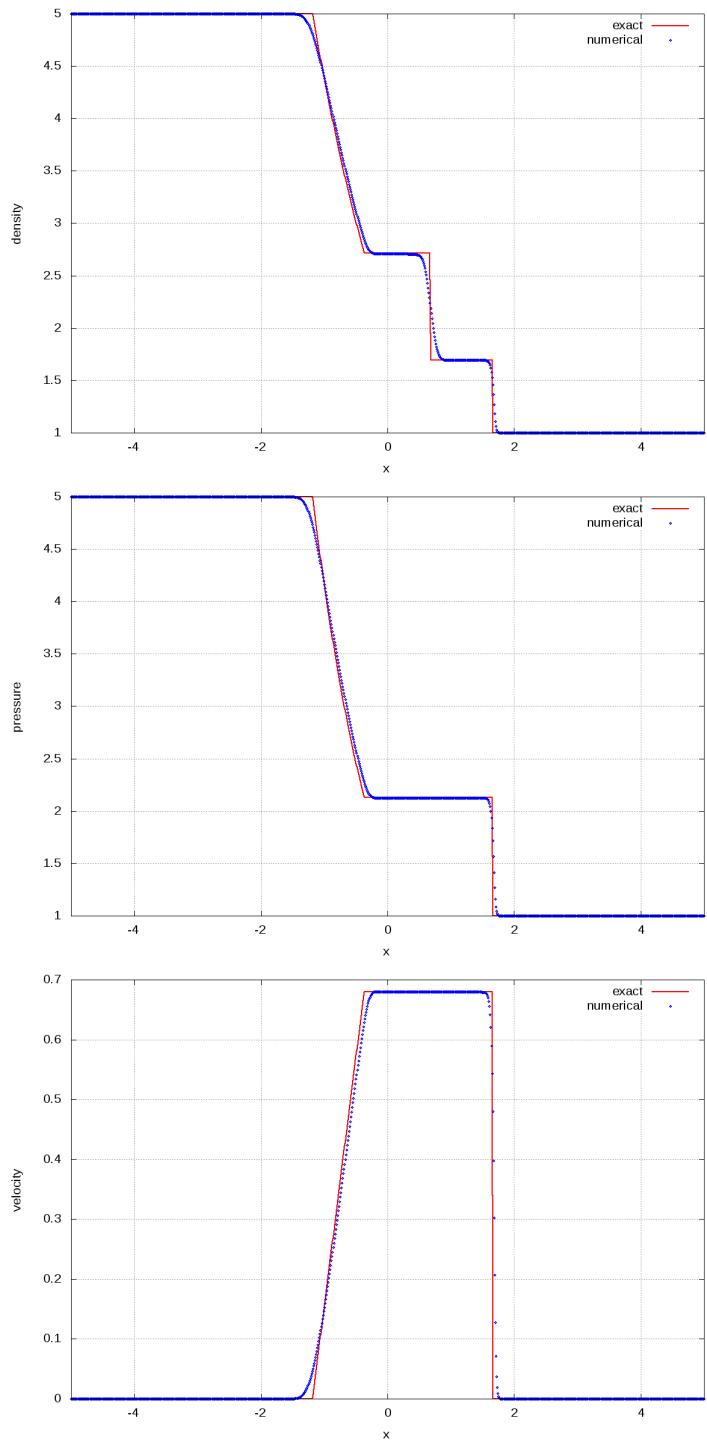


Figura 55: Resultados do esquema de Roe, $\frac{p_4}{p_1} = 5$

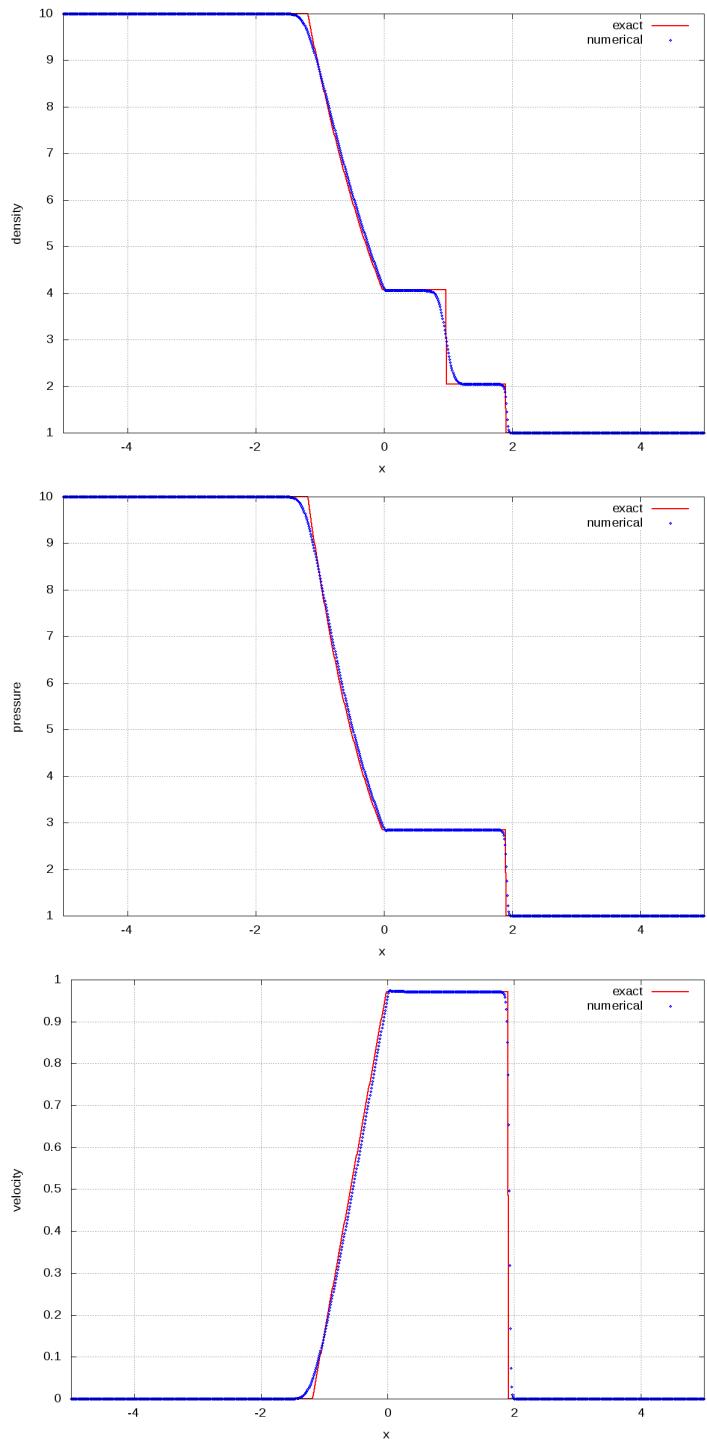


Figura 56: Resultados do esquema de Roe, $\frac{p_4}{p_1} = 10$

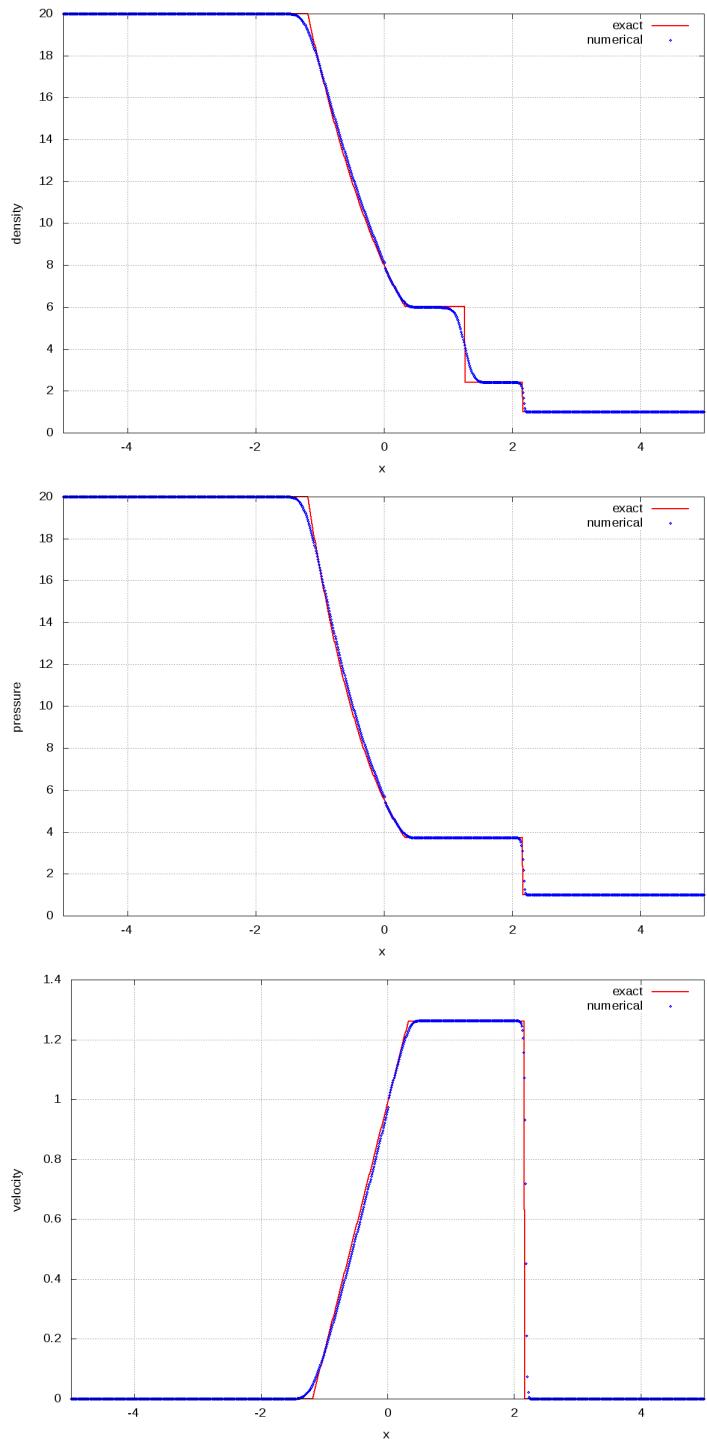


Figura 57: Resultados do esquema de Roe, $\frac{p_4}{p_1} = 20$

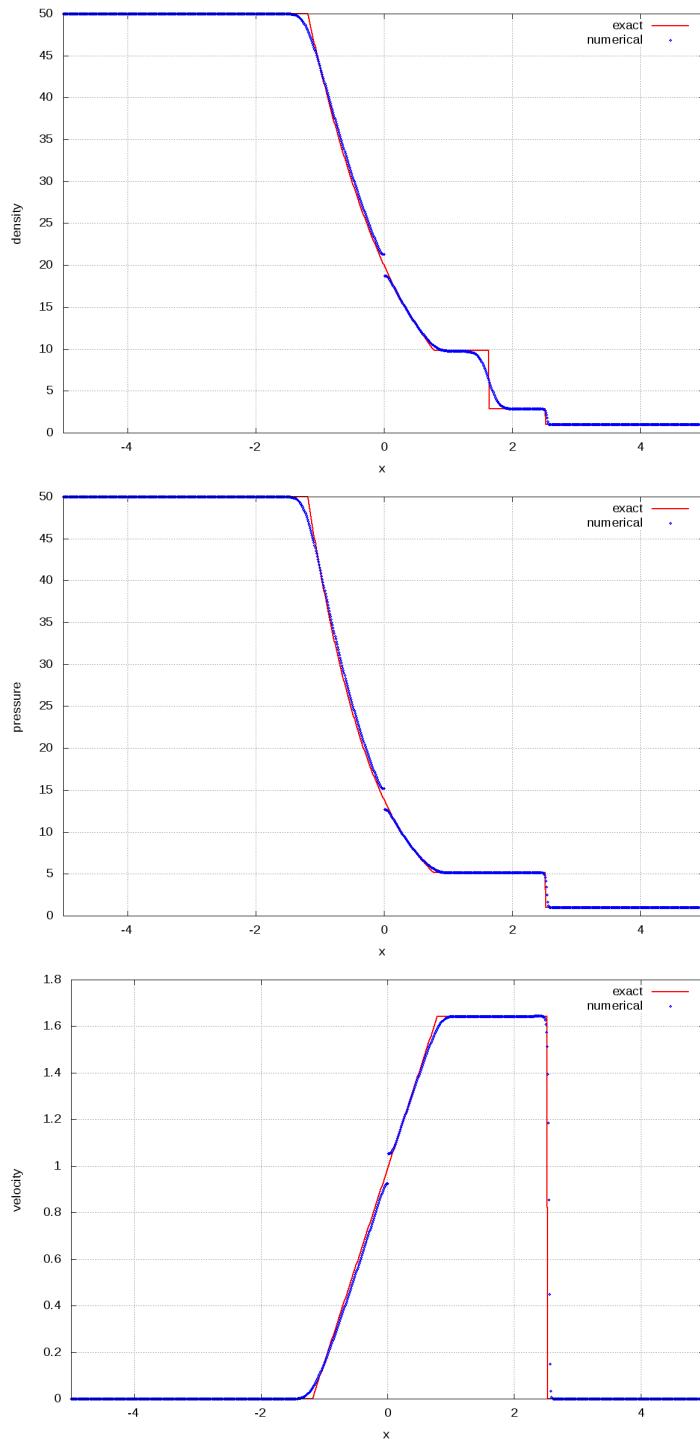


Figura 58: Resultados do esquema de Roe, $\frac{p_4}{p_1} = 50$

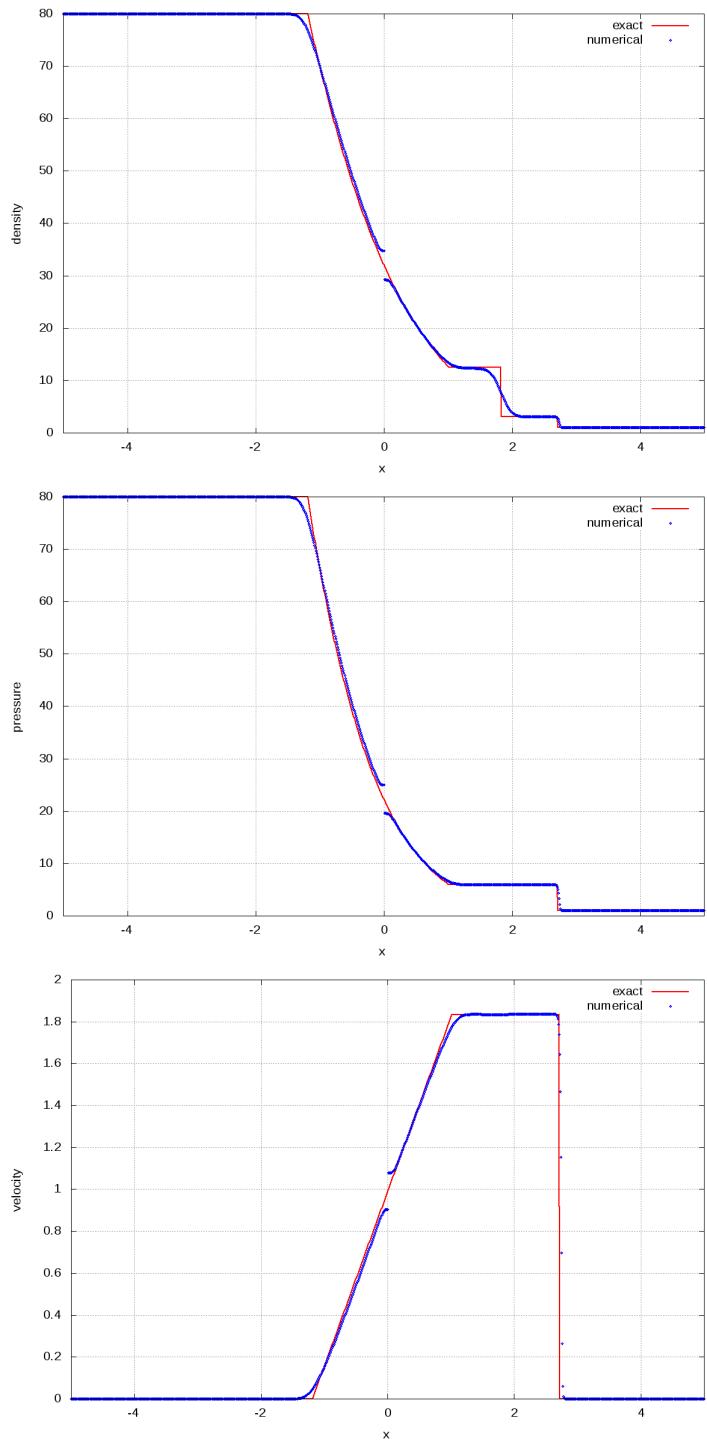


Figura 59: Resultados do esquema de Roe, $\frac{p_4}{p_1} = 80$

3.7 Projeto 3

A proposta do projeto 3 é resolver o problema do tubo de choque utilizando os seguintes métodos numéricos:

- Esquema TVD de Harten de primeira ordem;
- Esquema TVD de Harten de segunda ordem.

Cada um destes esquemas será descrito a seguir.

3.7.1 Esquema TVD de Harten

Harten [1] apresenta em seu trabalho uma nova classe de métodos, chamados inicialmente de TVNI (do inglês *total variation nonincreasing*) e posteriormente denominado TVD (do inglês *Total Variation Diminishing*), cuja principal característica é a de preservar a monotonicidade de uma solução fraca. Como é mencionado no artigo original de 1983, as seguintes propriedades de monotonicidade da solução são verificadas:

- (i) Nenhum máximo local da função pode ser criado.
- (ii) O valor de um mínimo local é não-decrescente; o valor de um máximo local é não-crescente.

Destas propriedades, diz-se que a solução é TVNI se

$$TV(u(t_2)) \leq TV(u(t_1)) \quad , \text{ para todo } t_2 \geq t_1 .$$

Harten diz que um esquema numérico que seja monotônico certamente garante monotonicidade da solução. Entretanto, um esquema que não seja monotônico, porém seja TVNI, garante monotonicidade da solução. O fluxo numérico generalizado para o método de Harten de segunda ordem pode ser escrito como

$$\begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix}_{j+\frac{1}{2}} = \frac{1}{2} \left\{ E_j + E_{j+1} + \frac{\Delta t}{\Delta x} \sum_{k=1}^3 R_{j+\frac{1}{2}}^k \left[g_j^k + g_{j+1}^k - \psi^k \left(\nu_{j+\frac{1}{2}}^k + \gamma_{j+\frac{1}{2}}^k \right) \alpha_{j+\frac{1}{2}}^k \right] \right\} , \quad (3.9)$$

onde temos

$$\nu_{j+\frac{1}{2}}^k = \frac{\Delta t}{\Delta x} a_{j+\frac{1}{2}}^k \begin{cases} a_{j+\frac{1}{2}}^1 = \hat{u}_{j+\frac{1}{2}} - \hat{c}_{j+\frac{1}{2}} \\ a_{j+\frac{1}{2}}^2 = \hat{u}_{j+\frac{1}{2}} \\ a_{j+\frac{1}{2}}^3 = \hat{u}_{j+\frac{1}{2}} + \hat{c}_{j+\frac{1}{2}} \end{cases}$$

$$g_j^k = s_{j+\frac{1}{2}}^k \max \left[0, \min \left(|\tilde{g}_{j+\frac{1}{2}}^k|, s_{j+\frac{1}{2}}^k \tilde{g}_{j+\frac{1}{2}}^k \right) \right]$$

$$s_{j+\frac{1}{2}}^k = \text{sign} \left(\tilde{g}_{j+\frac{1}{2}}^k \right)$$

$$\tilde{g}_{j+\frac{1}{2}}^k = \frac{1}{2} \left[\psi^k \left(\nu_{j+\frac{1}{2}}^k \right) - \left(\nu_{j+\frac{1}{2}}^k \right)^2 \right] \alpha_{j+\frac{1}{2}}^k$$

$$\gamma_{j+\frac{1}{2}}^k = \begin{cases} \frac{g_{j+1}^k - g_j^k}{\alpha_{j+\frac{1}{2}}^k} , \text{ se } \alpha_{j+\frac{1}{2}}^k \neq 0 \\ 0 , \text{ se } \alpha_{j+\frac{1}{2}}^k = 0 \end{cases} .$$

A função $\psi^k(\cdot)$, para que seja satisfeita a condição de entropia, é definida como

$$\psi^k(z) = \begin{cases} \frac{1}{2} \left(\frac{z^2}{\epsilon^k} + \epsilon^k \right) , \text{ se } |z| < \epsilon^k \\ |z| , \text{ se } |z| \geq \epsilon^k \end{cases} ,$$

sendo

$$\epsilon^k = \begin{cases} 0.2 , \text{ para campos genuinamente não lineares} \\ 0 , \text{ para campos linearmente degenerados} \end{cases} .$$

Poucas alterações a este fluxo numérico generalizado são feitas para tornar o método de primeira ordem. Além disso, o uso de limitadores também se faz necessário para o caso de segunda ordem. Estas questões serão melhores explicadas nas subseções a seguir.

A. Esquema de Primeira Ordem

Para o esquema de primeira ordem, como mencionado anteriormente, são feitas algumas poucas alterações na Eq. 3.10. Basicamente, para que tenhamos um método de primeira ordem à partir da equação generalizada do fluxo numérico, basta fazer

$$g_j^k = 0 ,$$

$$g_{j+1}^k = 0 ,$$

$$\gamma_{j+\frac{1}{2}}^k = 0 ,$$

resumindo o método a algo muito similar ao método de Roe, sendo o fluxo numérico dado por

$$\begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix}_{j+\frac{1}{2}} = \frac{1}{2} \left\{ E_j + E_{j+1} - \frac{\Delta t}{\Delta x} \sum_{k=1}^3 R_{j+\frac{1}{2}}^k \psi^k \left(\nu_{j+\frac{1}{2}}^k \right) \alpha_{j+\frac{1}{2}}^k \right\}. \quad (3.10)$$

Os resultados são apresentados a seguir.

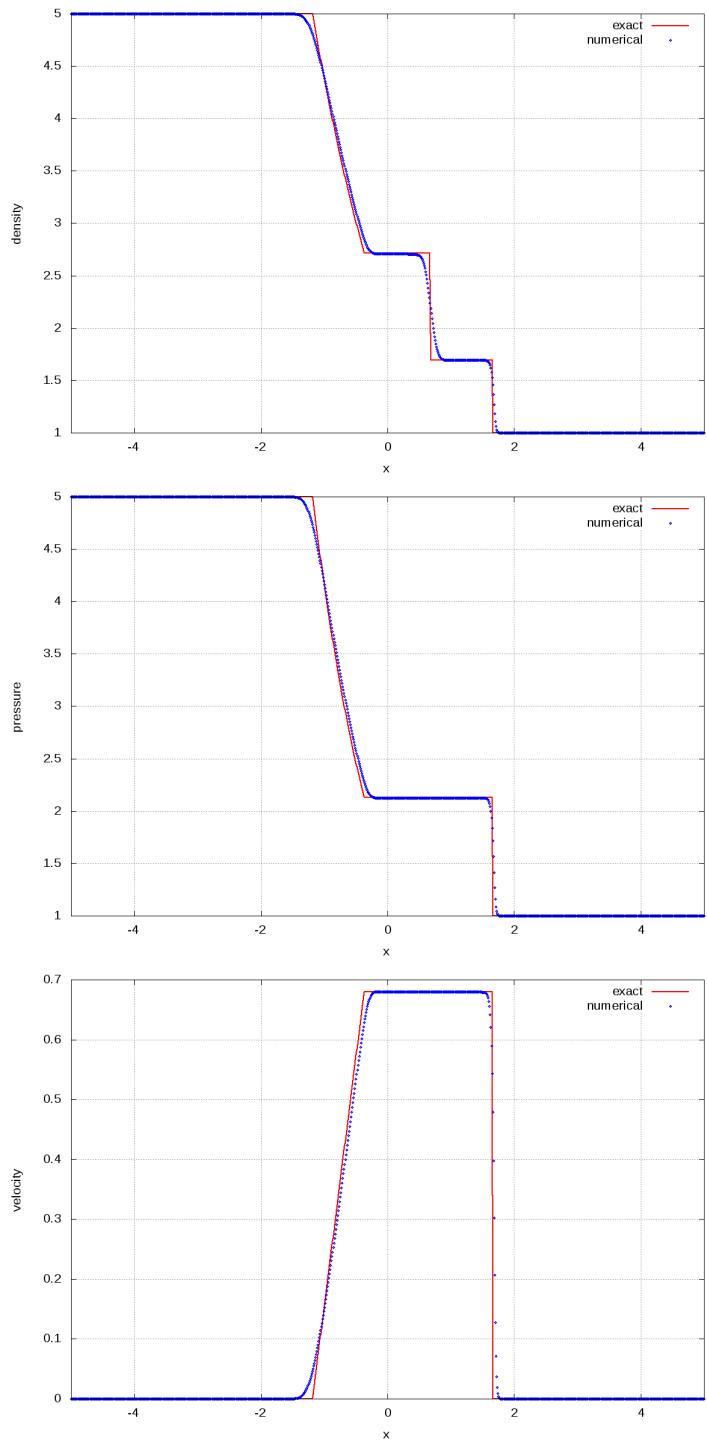


Figura 60: Resultados do esquema de Harten de primeira ordem, $\frac{p_4}{p_1} = 5$

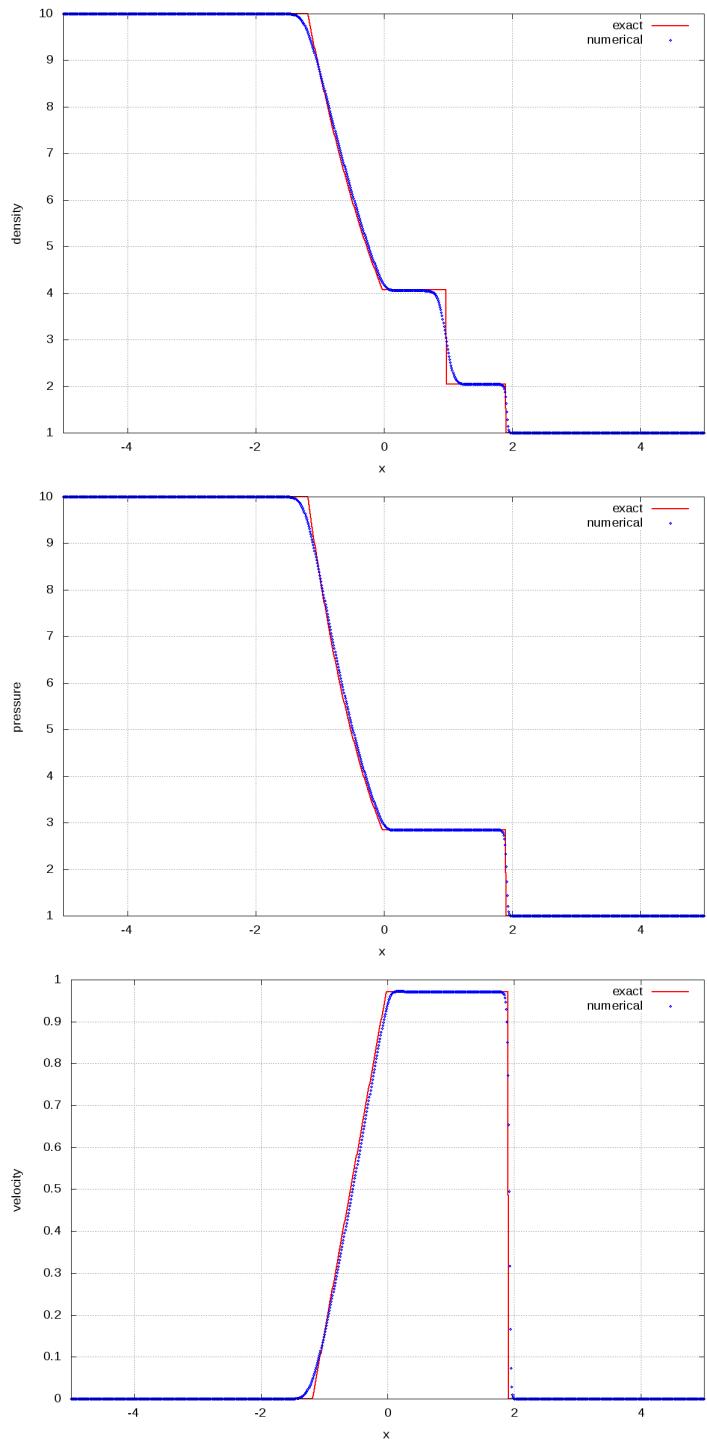


Figura 61: Resultados do esquema de Harten de primeira ordem, $\frac{p_4}{p_1} = 10$

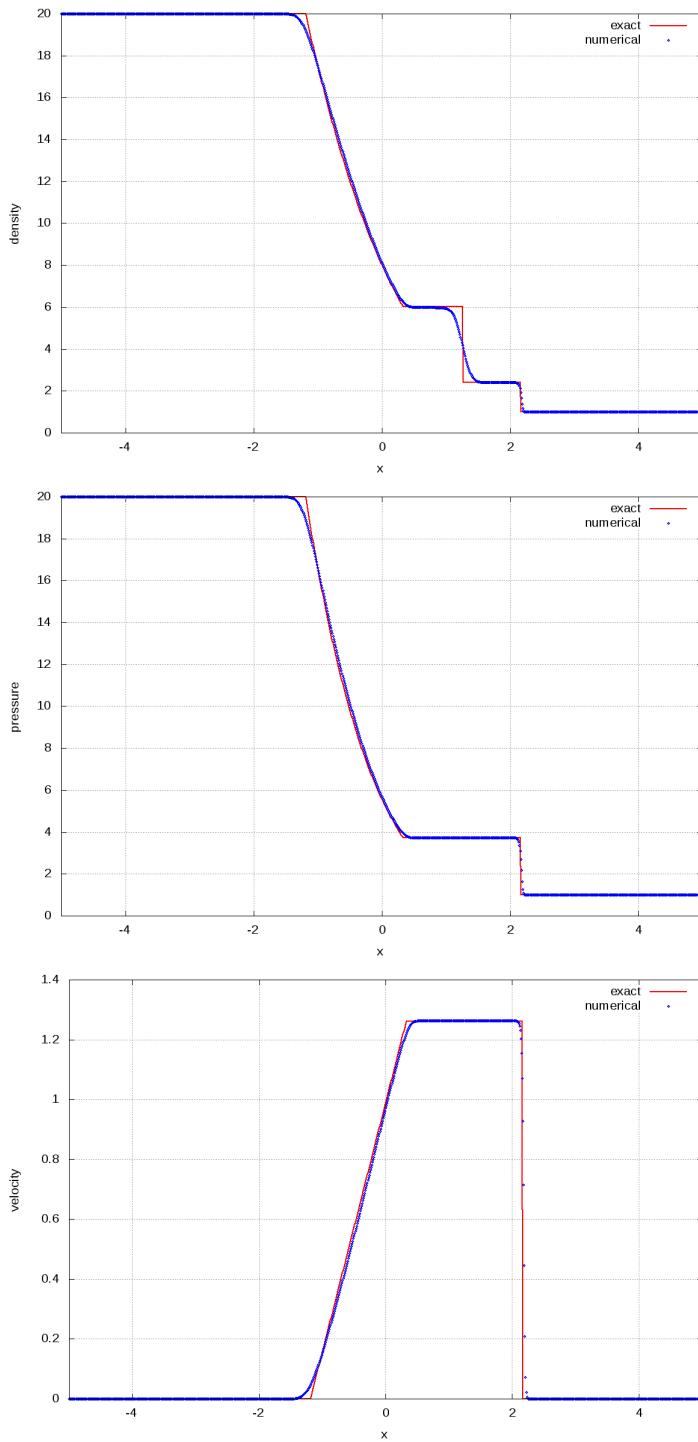


Figura 62: Resultados do esquema de Harten de primeira ordem, $\frac{p_4}{p_1} = 20$

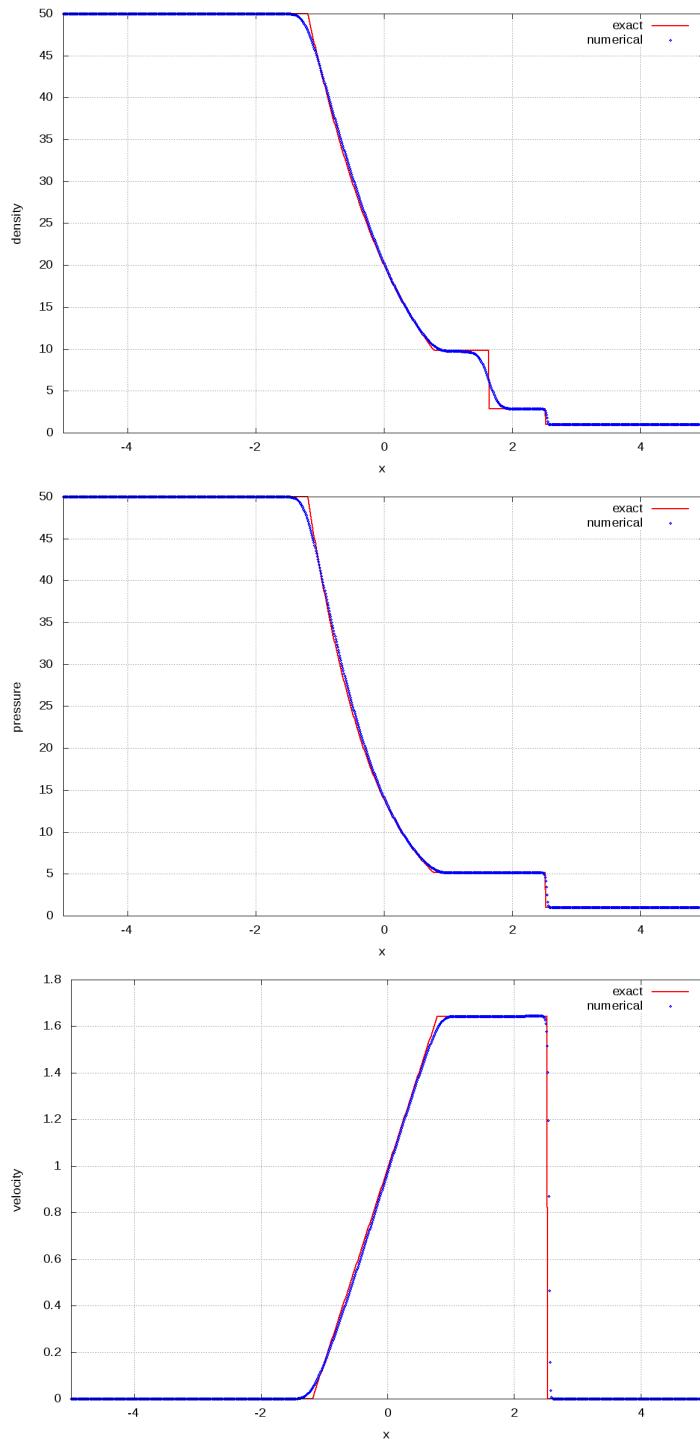


Figura 63: Resultados do esquema de Harten de primeira ordem, $\frac{p_4}{p_1} = 50$

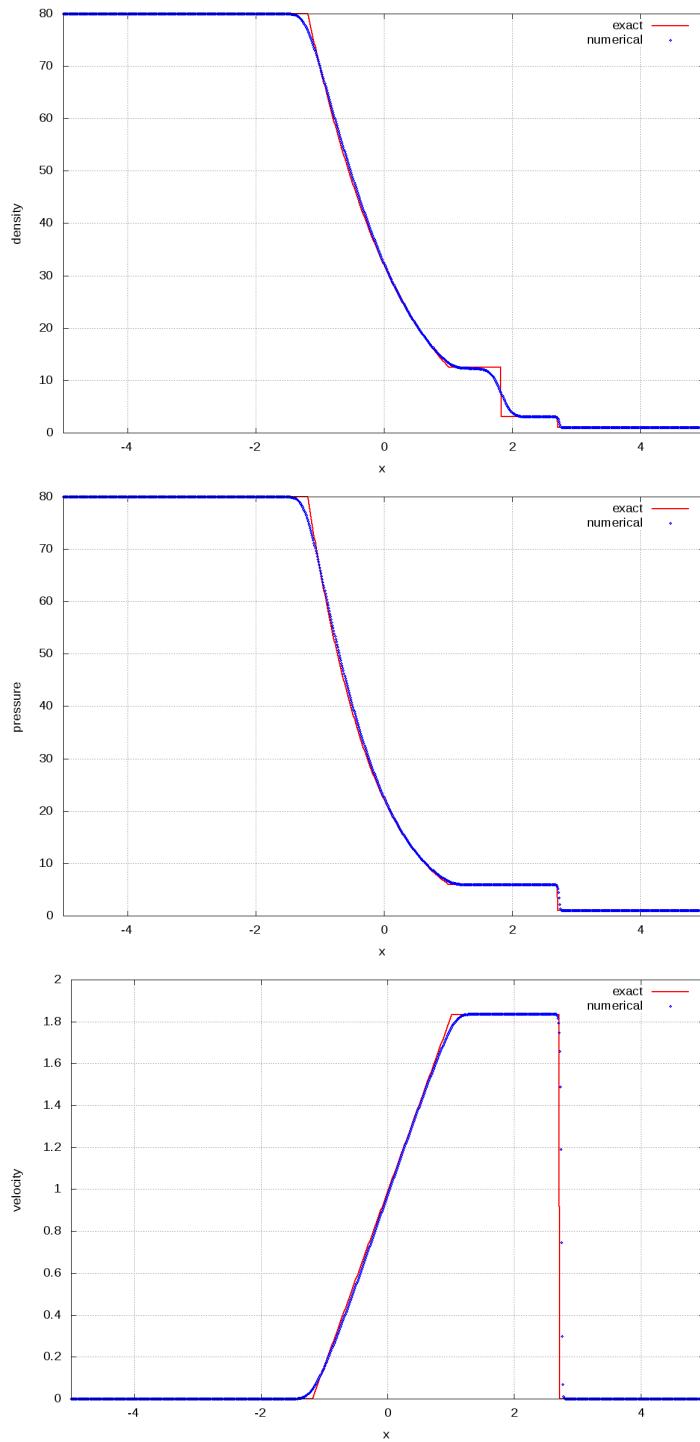


Figura 64: Resultados do esquema de Harten de primeira ordem, $\frac{p_4}{p_1} = 80$

B. Esquema de Segunda Ordem

Para o esquema de segunda ordem, a Eq. 3.10 define o fluxo numérico do método. Conforme anteriormente mencionado, para métodos de ordem maior que 1, o uso de limitadores se faz necessários. Isto se deve ao fato de que, para se manter a monotonicidade da solução e manter boa resolução na captura de fenômenos como ondas de choque, o método deve se manter de primeira ordem em regiões de grandes oscilações (variações de máximos e mínimos). Os resultados são apresentados a seguir.

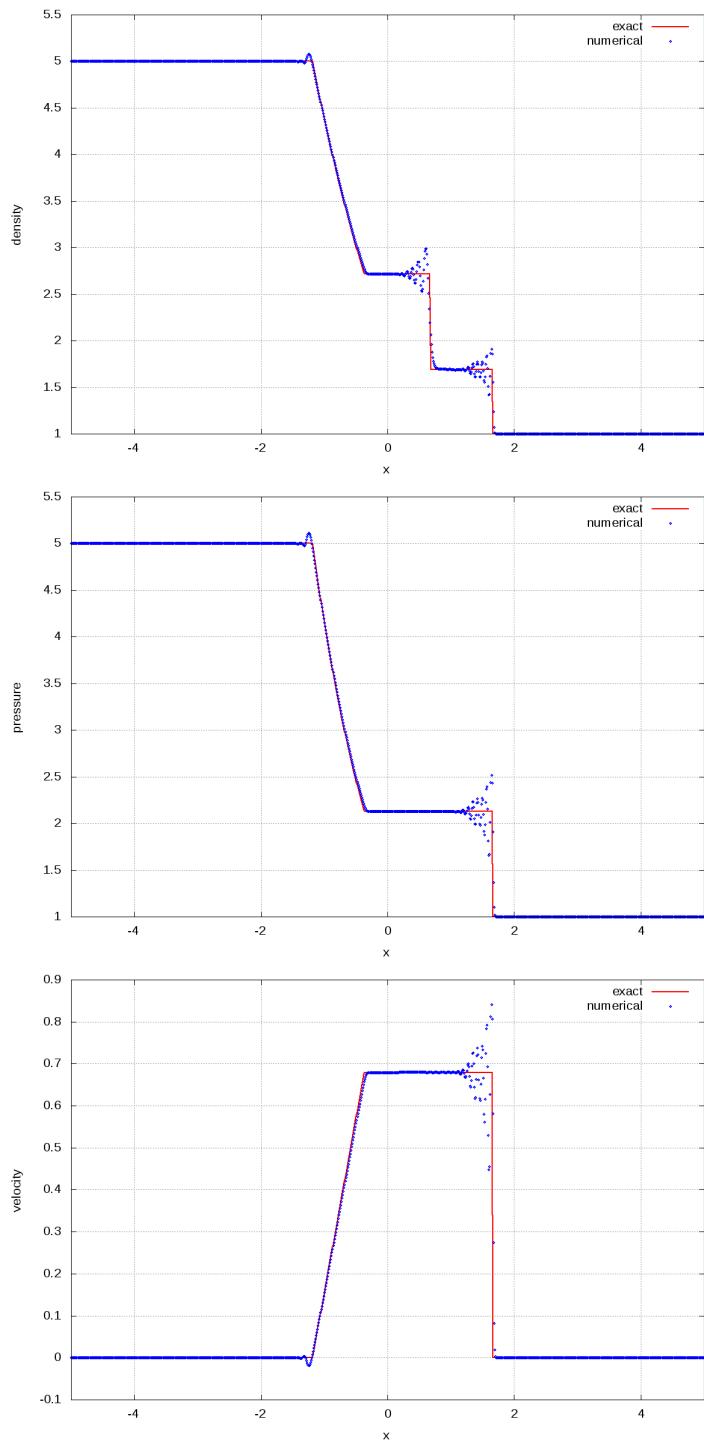


Figura 65: Resultados do esquema de Harten de segunda ordem e sem limitador,

$$\frac{p_4}{p_1} = 5$$

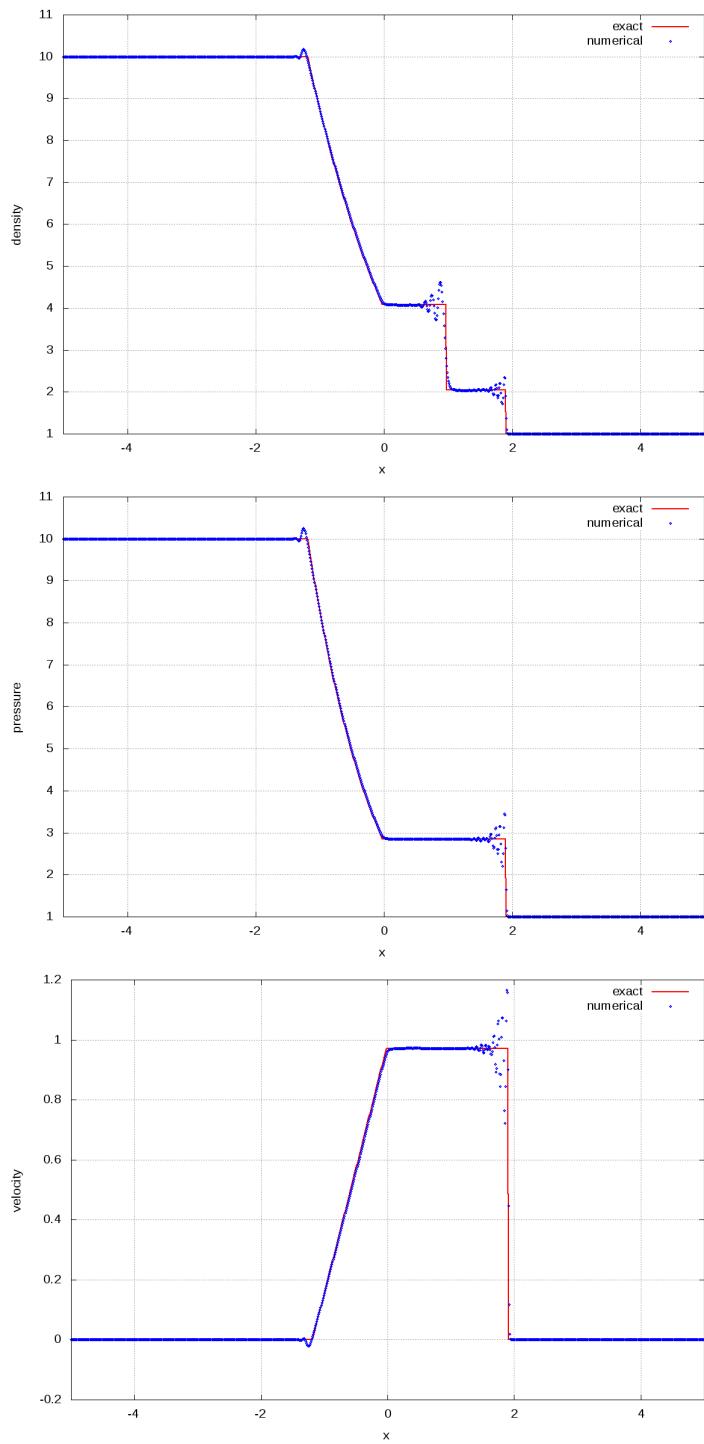


Figura 66: Resultados do esquema de Harten de segunda ordem e sem limitador,

$$\frac{p_4}{p_1} = 10$$

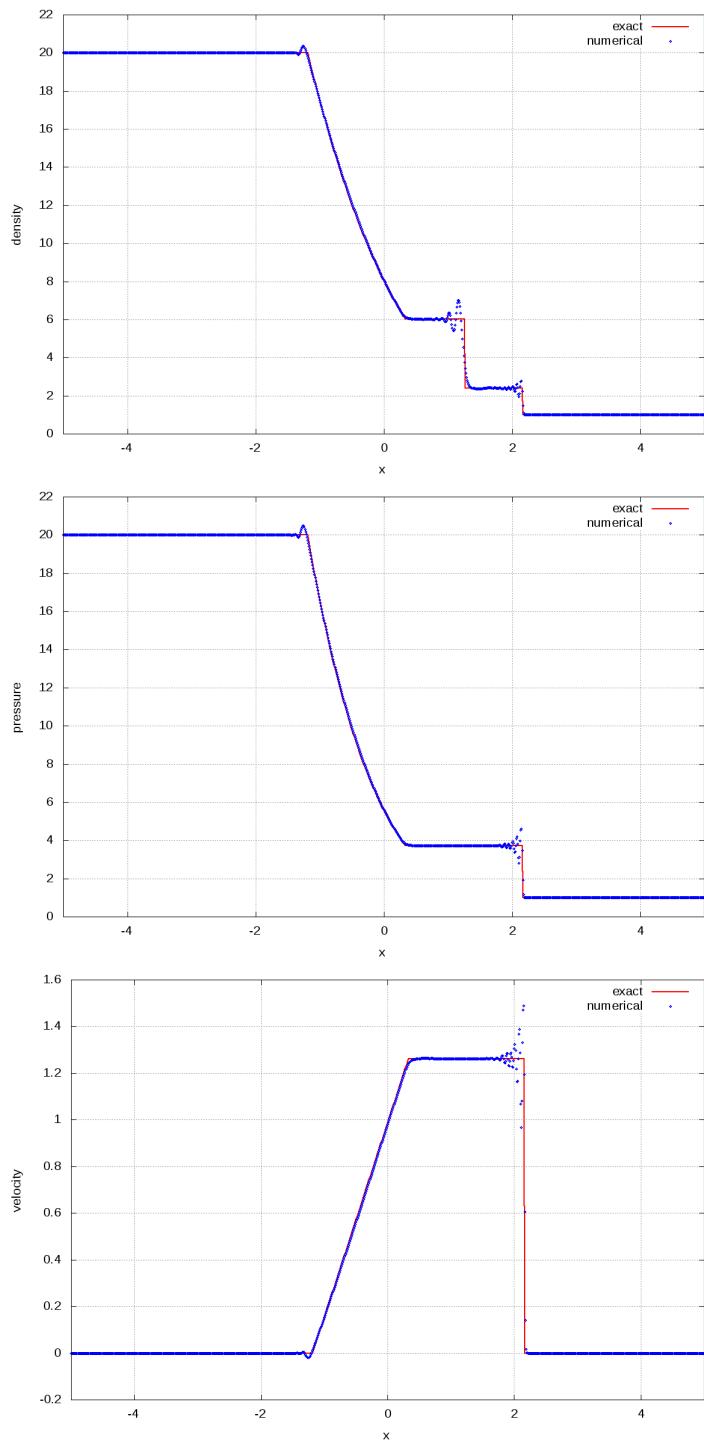


Figura 67: Resultados do esquema de Harten de segunda ordem e sem limitador,

$$\frac{p_4}{p_1} = 20$$

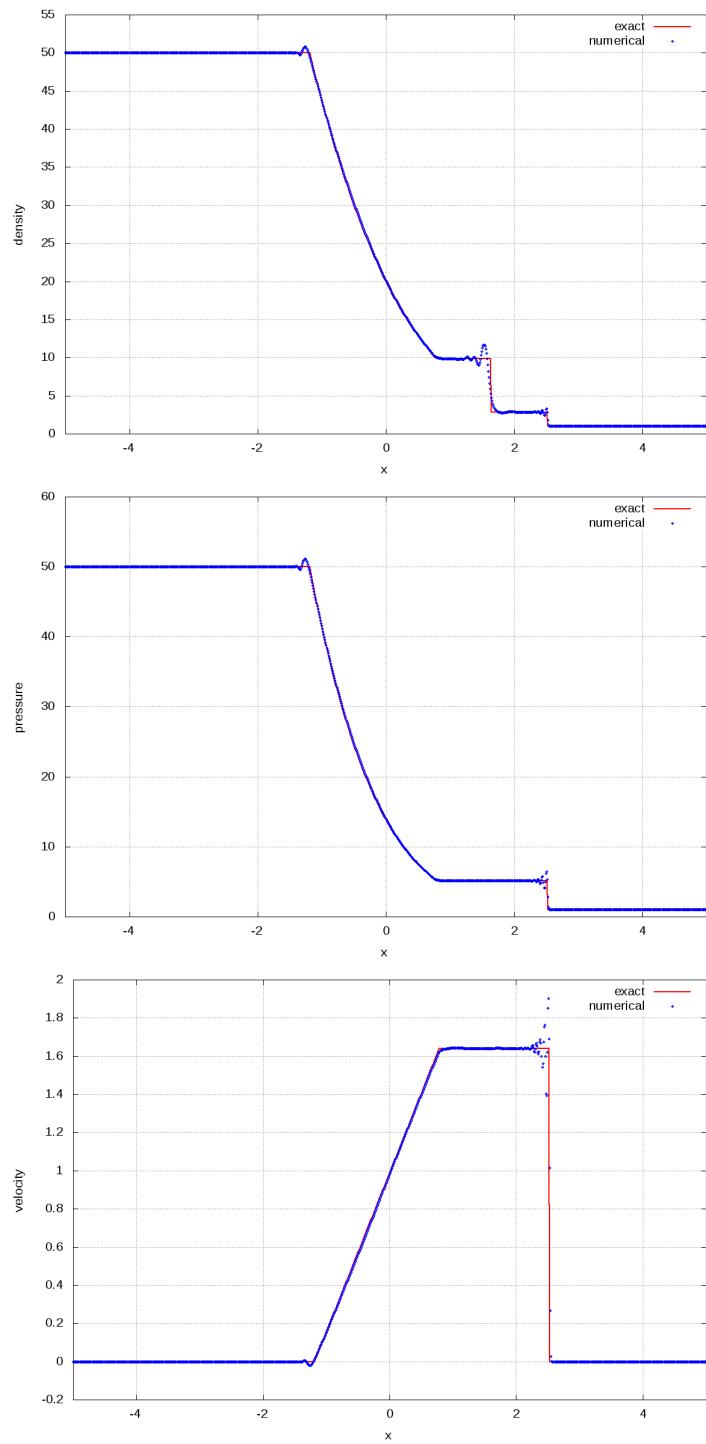


Figura 68: Resultados do esquema de Harten de segunda ordem e sem limitador,

$$\frac{p_4}{p_1} = 50$$

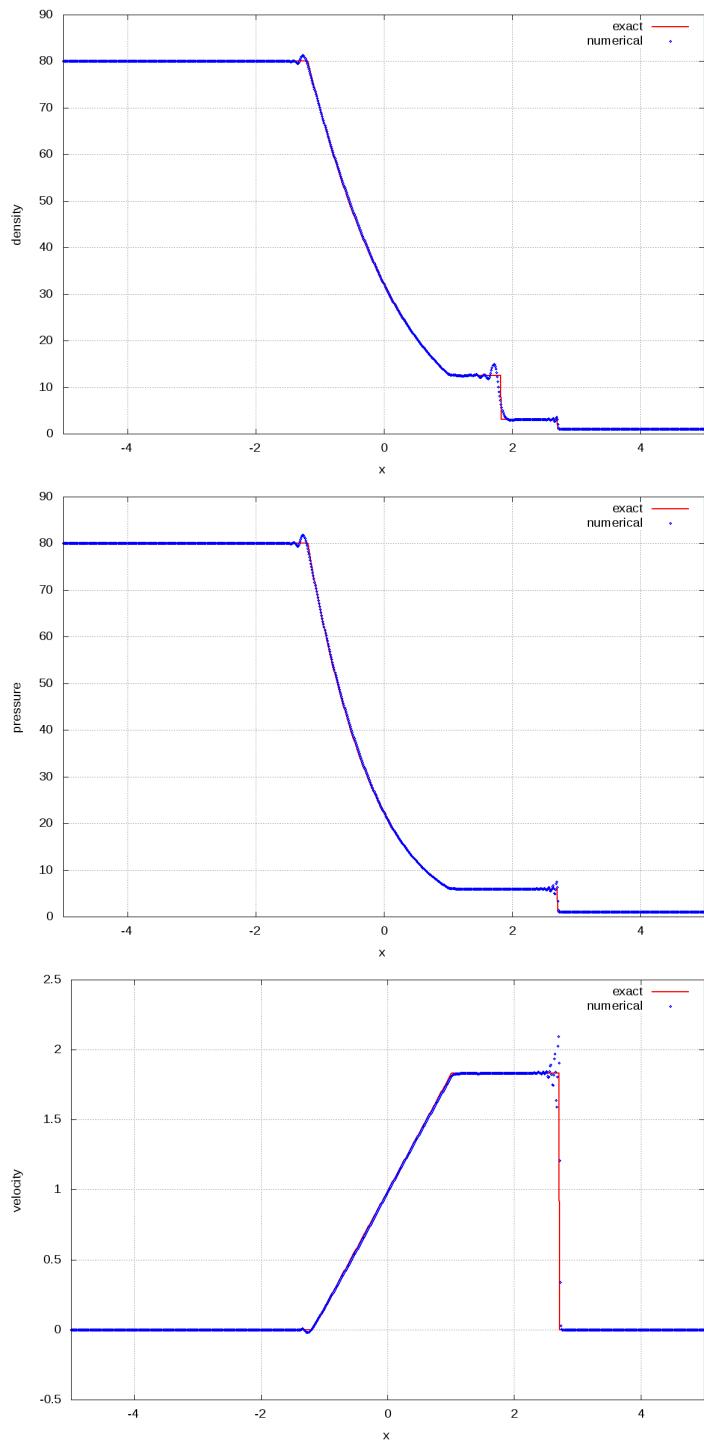


Figura 69: Resultados do esquema de Harten de segunda ordem e sem limitador,

$$\frac{p_4}{p_1} = 80$$

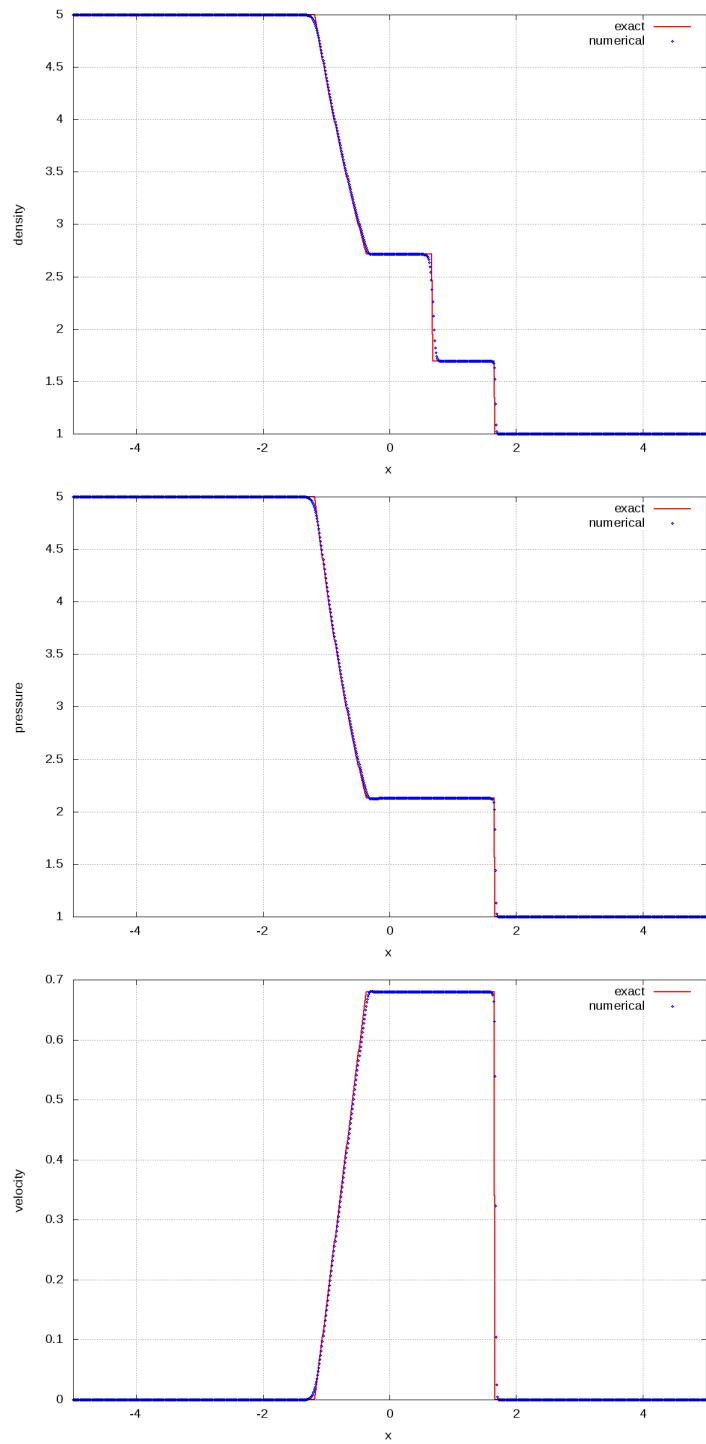


Figura 70: Resultados do esquema de Harten de segunda ordem e com limitador,

$$\frac{p_4}{p_1} = 5$$

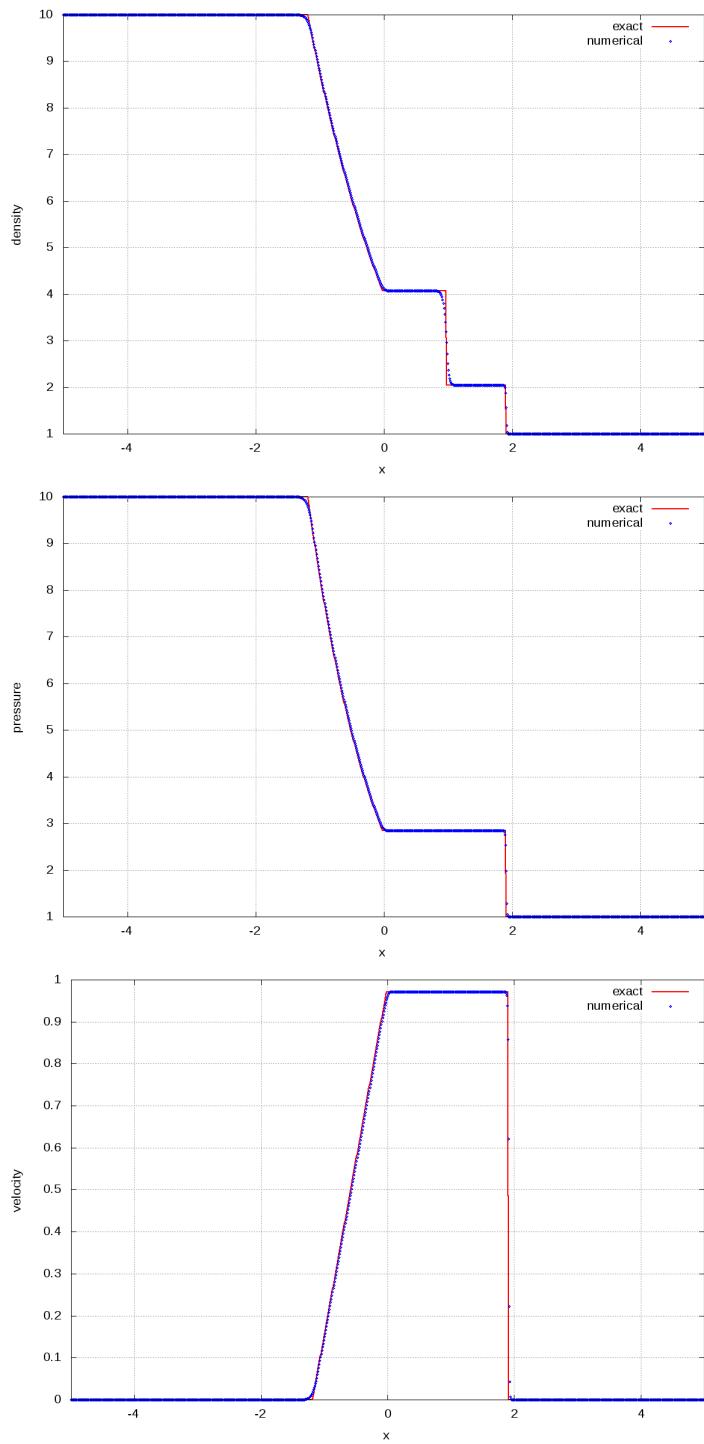


Figura 71: Resultados do esquema de Harten de segunda ordem e com limitador,

$$\frac{p_4}{p_1} = 10$$

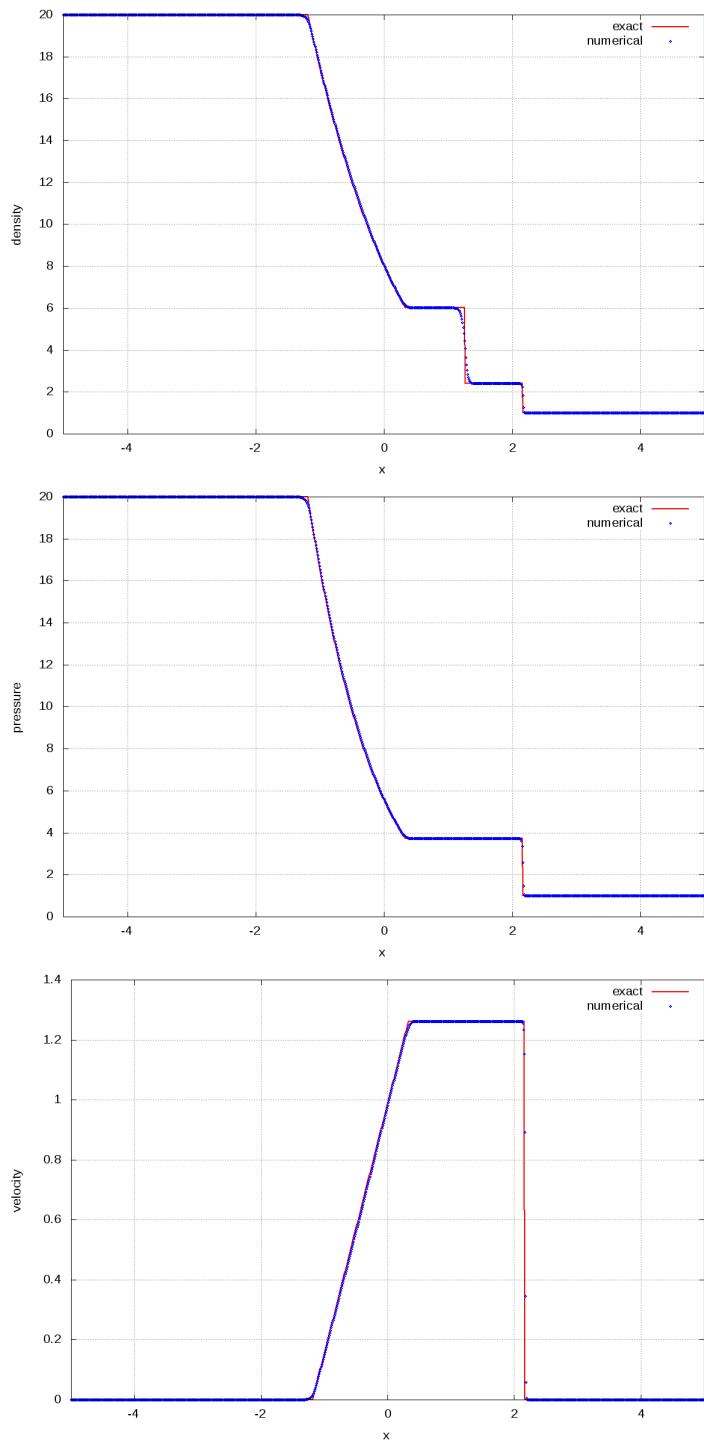


Figura 72: Resultados do esquema de Harten de segunda ordem e com limitador,

$$\frac{p_4}{p_1} = 20$$

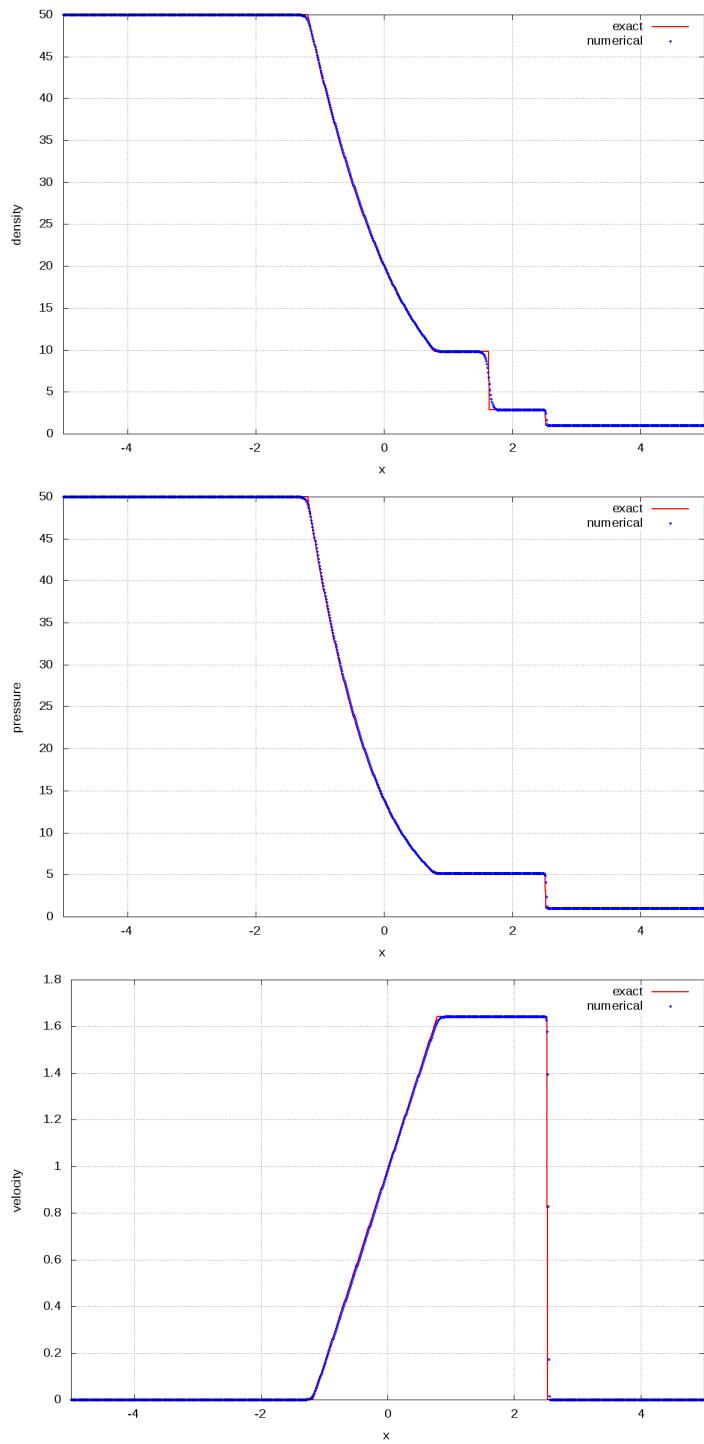


Figura 73: Resultados do esquema de Harten de segunda ordem e com limitador,

$$\frac{p_4}{p_1} = 50$$

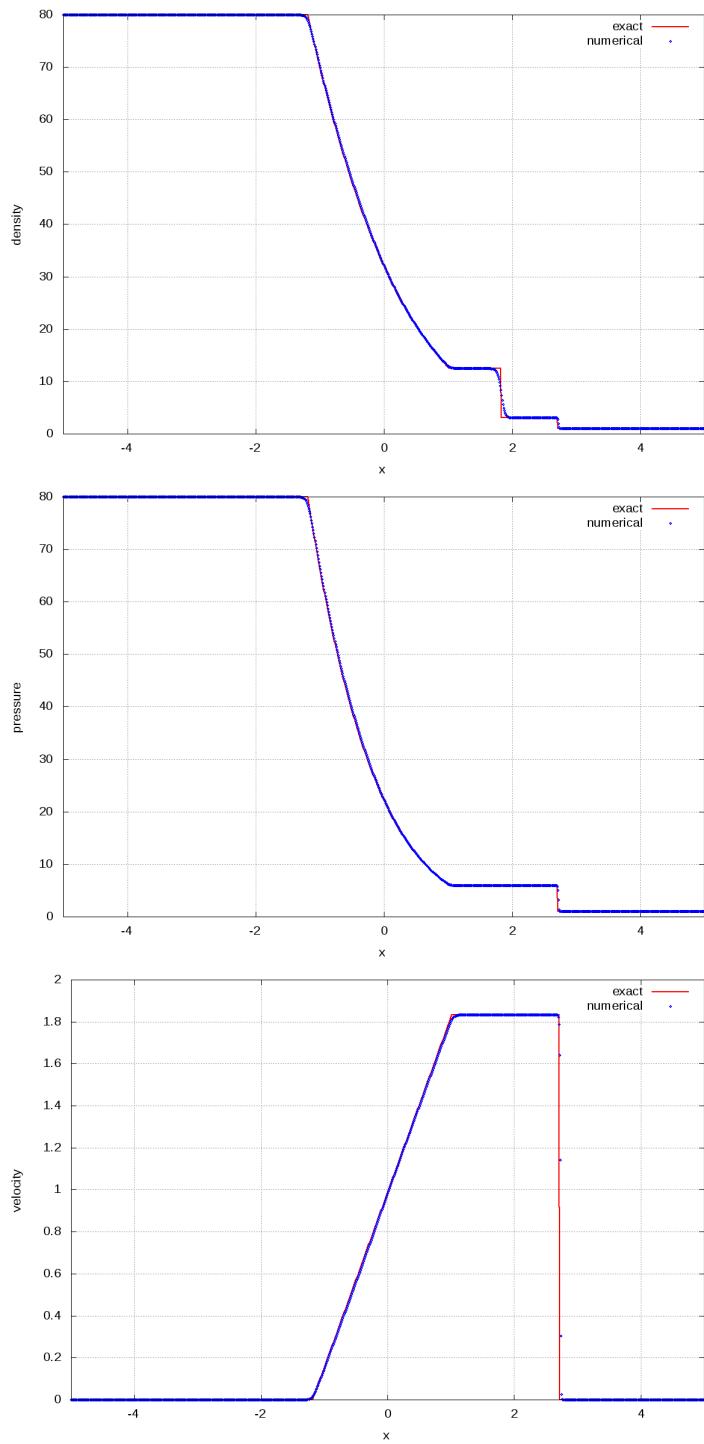


Figura 74: Resultados do esquema de Harten de segunda ordem e com limitador,

$$\frac{p_4}{p_1} = 80$$

4 CONCLUSÕES

Vimos ao longo do projeto 1 que existem algumas vantagens de alguns métodos em relação a outros. Com relação aos termos de dissipação artificial linear de segunda e quarta diferença, podemos afirmar que o modelo de segunda diferença apresenta maior dissipação, ao passo que o modelo de quarta diferença é um pouco mais suave. Isto pode ser observado pelo fato do aparecimento de oscilações nas regiões próximas às descontinuidades presentes na solução, no caso do modelo de diferença quarta. Já para o caso do modelo de dissipação não-linear, temos uma combinação de ambos os modelos lineares, apresentando bons resultados em regiões longe do choque e pequenas oscilações próximas ao choque. Neste caso, as oscilações se agravam apenas para os casos de altas razões de pressão (acima de 50). É importante ressaltar que, ao se selecionar um modelo e a quantidade de dissipação artificial que se deseja utilizar, deve-se levar em conta os tipos de fenômenos que se deseja observar ao longo da solução, de forma que a resolução da solução não seja degradada.

No projeto 2 tivemos resultados para esquemas do tipo FVS. Estes esquemas apresentaram a vantagem de permitir uma discretização *one-sided*, visto que faz-se a separação dos vetores de fluxo de acordo com as direções de propagação das características do problema. Isto se torna interessante do ponto de vista do termo de dissipação artificial, onde esse tipo de dissipação acaba por surgir da própria estratégia de discretização. Foi possível observar também que utilizando discretizações com ordem de precisão maior que 1, os métodos se tornam mais sensíveis a oscilações e, portanto, os resultados podem divergir mesmo para razões de pressão não tão altas (abaixo de 20).

Ao longo da implementação dos métodos para o projeto 3 foi possível observar os efeitos de se utilizar um limitador para um esquema TVD de ordem maior que 1. O esquema de Harten de primeira ordem já apresenta boas características para captura de choque, expansão e contato, mesmo para altas razões de pressão (acima de 50). Com o intuito de aumentar a ordem de precisão do método, faz-se necessário a definição do termo limitador. Observa-se nos resultados para o

método de segunda ordem sem uso do limitador que oscilações espúrias surgem nas regiões de descontinuidades, mesmo para razões de pressão mais baixas. Desta forma, é fundamental a utilização de limitador para métodos de ordens mais altas.

Não foi implementado, para nenhum dos métodos, um esquema implícito de marcha no tempo. Entretanto, estudos do efeito do refinamento da malha, efeito do valor da constante do termo de dissipação linear e simulações para razões de pressão acima de 50 foram realizados e os resultados foram apresentados ao longo do trabalho. Do ponto de vista de dificuldade de implementação de cada método, por se tratar de um problema unidimensional e regido pelas equações de Euler, todos foram relativamente simples de implementar. Entretanto, caso fosse necessária a expansão para o caso bi-dimensional, os métodos de Liou (AUSM⁺), Roe e Harten possuem mais alto grau de complexidade, visto que existe maior número de variáveis a serem calculadas e armazenadas antes de avançar a solução no tempo.

5 ANEXOS

Os códigos aqui apresentados foram criados e armazenados na plataforma GitHub, para controle de versão. O link para acesso é https://github.com/gupalmeida/cc299_projects/.

Input file:

```
1 #ifndef INPUT_H
2 #define INPUT_H
3
4 /* Output file name */
5 #define filename "output.dat"
6
7 /* Inputs */
8 #define p4 80.0
9 #define p1 1.0
10 #define tol 1.0e-5
11
12 /* Spatial discretization parameters */
13 #define l 5.0
14 #define imax 1001
15 // #define imax 501
16
17 /* Time discretization parameters */
18 #define cfl 0.1
19 #define tmax 1.0
20 #define itmax 200000
21 #define printAt 200
22
23 /* Initial states */
24 #define U0 0.0
25 #define T0 288.15
26
27 /* Fluid constants */
28 #define R 287.0
29 #define cp 1004.5
30 #define cv 717.5
31 #define gamma 1.4
```

```

32
33 /* Method selection */
34 #define method 7
35 #define order 2
36 #define mu 0.4
37 #define dissipModel 2
38 #define soundSpeedType 1
39 #define interfaceSoundSpeed 1
40 #define limiter 1
41
42 #endif
43
44 /* The order parameter sets the spatial discretization
45   order for the vector flux splitting schemes
46   which uses one-sided discretization for each
47   of the flux vectors (backward for E+ and forward
48   for E-).
49   The available options are:
50     1 - first order
51     2 - second order */
52
53 /* mu is a constant value used
54   in the linear dissipation model.
55   The model is selected by the dissipModel
56   parameter in which
57     0 - linear 2nd order difference model
58     1 - linear 4th order difference model
59     2 - non-linear jameson model */
60
61 /* soundSpeedType defines the way the code will
62   calculate the sound speed at each solution
63   point j. The available options are:
64     1 - a = aCritical * min( 1.0 , aCritical/|u| )
65     2 - a = sqrt( gamma * (p/rho) ) */
66
67 /* interfaceSoundSpeed defines the way the code will
68   calculate the sound speed at the interface between
69   two solution points. The available options are:

```

```

70      1 - a = min ( a_left , a_right )
71      2 - a = 0.5 * ( a_left + a_right )
72      3 - a = sqrt( a_left * a_right )
73      4 - Roe averaged sound speed */
74
75 /* The solution method is selected by changing
76   the method value.
77   Available solution methods
78   0 - centered scheme
79   1 - Lax-Wendroff method
80   2 - McCormack method
81   3 - Steger and Warming FVS scheme
82   4 - van Leer FVS non-MUSCL method
83   5 - Liou FVS scheme (AUSM+)
84   6 - Roe approximate Riemann solver
85   7 - Harten TVD method */

```

Main file:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 #include "input.h"
6 #include "mesh.h"
7 #include "aux.h"
8 #include "solvers.h"
9
10 void writeSolution( results * , grid * ,char fileName [] );
11
12 int main(){
13
14     /* parameter definition */
15     grid mesh;
16     results solution;
17     results exact;
18
19     /* generating mesh */
20     meshAlloc(&mesh);

```

```

21     createMesh(&mesh) ;
22
23     /* initializing variables */
24     allocSolution(&solution) ;
25     initSolution(&solution) ;
26     // writeSolution(&solution ,&mesh) ;
27
28     /* computing the exact solution */
29     exactSolution(&exact ,&mesh) ;
30     writeSolution(&exact ,&mesh , "exact.dat") ;
31
32     /* solving Euler equations */
33     switch (method){
34         case 0:
35             centeredScheme(&solution ,&mesh) ;
36             break;
37         case 1:
38             laxWendroff(&solution ,&mesh) ;
39             break;
40         case 2:
41             macCormack(&solution ,&mesh) ;
42             break;
43         case 3:
44             stegerWarming(&solution ,&mesh) ;
45             break;
46         case 4:
47             vanLeerNonMUSCL(&solution ,&mesh) ;
48             break;
49         case 5:
50             liouAUSMplus(&solution ,&mesh) ;
51             break;
52         default :
53             centeredScheme(&solution ,&mesh) ;
54             break;
55     }
56
57     /* writting output file */
58     writeSolution(&solution ,&mesh , filename) ;

```

```

59     freeMesh(&mesh) ;
60     freeSolution(&solution) ;
61     //freeSolution(&exact) ;
62 }
63
64 void writeSolution( results * solution , grid * mesh , char fileName [] )
{
65     /* Auxiliary function to write the
66      * output file with solution */
67     FILE *fp ;
68
69     fp = fopen (fileName , "w") ;
70     if (fp != NULL){
71         for (int i = 0; i < imax; i++){
72             fprintf(fp , "%f\t%f\t%f\t%f\n" ,mesh->x[ i ] ,solution->press
73             [ i ] ,solution->rho[ i ] ,solution->vel[ i ]) ;
74         }
75     }
76     else{
77         printf("Could not handle the specified file\n\n");
78     }
79 }

```

Mesh.h file:

```

1 #ifndef MESH_H
2 #define MESH_H
3
4 typedef struct grid{
5     /* x is a unidimensional array
6      * used to allocate x-coordinates
7      * of the computational mesh */
8
9     double *x;
10 }grid;
11
12 void meshAlloc(grid *);
13 void createMesh(grid *);

```

```

14 void freeMesh(grid *);
15
16 #endif
```

Mesh.c file:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5 #include "mesh.h"
6 #include "input.h"
7
8 void meshAlloc(grid * mesh){
9     mesh->x = malloc(imax * sizeof(double));
10 }
11
12 void createMesh(grid * mesh){
13     mesh->x[imax-1] = 1;
14     double dx = (double)((1/(imax-1))*2.0);
15     for (int i = (imax-2); i >= 0; --i){
16         mesh->x[i] = mesh->x[i+1] - dx;
17     };
18 }
19
20 void freeMesh(grid * mesh){
21     free(mesh->x);
22 }
```

Solvers.h file:

```

1 #ifndef SOLVERS_H
2 #define SOLVERS_H
3
4 void centeredScheme(results *, grid *);
5 void laxWendroff(results *, grid *);
6 void macCormack(results *, grid *);
7 void stegerWarming(results *, grid *);
8 void vanLeerNonMUSCL(results *, grid *);
9 void liouAUSMplus(results *, grid *);
```

```

10 void roeMethod( results *, grid *);
11 void hartenTVD( results *, grid *);
12 void exactSolution( results *, grid *);
13
14 #endif

```

Solvers.c file:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <time.h>
5
6 #include "input.h"
7 #include "mesh.h"
8 #include "aux.h"
9 #include "solvers.h"
10
11 /* beam and warming explicit centered scheme */
12 void centeredScheme(results * solution, grid * mesh){
13     /* BEAM and WARMING method using explicit
14      * Euler as time marching method and a
15      * simple central difference scheme for
16      * spatial discretization .
17      * =====
18      * Qnp1 = Qnp - 0.5*(dt/dx)*(Ejp1 - Ejm1)
19      * ===== */
20
21     double *Q1np1 = malloc(imax * sizeof(double));
22     double *Q2np1 = malloc(imax * sizeof(double));
23     double *Q3np1 = malloc(imax * sizeof(double));
24     int it = 0;
25     double dx = mesh->x[1] - mesh->x[0];
26     double a = sqrt(gamma);
27     double dt = (dx*cfl)/a;
28     double t = 0.0;
29     double lambda = dt/dx;
30
31     printf("\n\n=====\\n");

```

```

32     printf("          Centered Scheme\n");
33     printf("=====\\n\\n");
34
35     while (t <= (double) tmax && it < (int) itmax){
36         /* calculates the flux vector */
37         calcFluxes(solution);
38
39         /* calculates dissipation for the current solution Q */
40         calcDissipation(solution, lambda);
41
42         /* marching the solution */
43         for (int j = 2; j < imax-2; j++){
44             Q1np1[j] = solution->Q1[j] - 0.5*lambda*(solution->E1[j+1] - solution->E1[j-1]) + lambda*(solution->dissip1[j]);
45             Q2np1[j] = solution->Q2[j] - 0.5*lambda*(solution->E2[j+1] - solution->E2[j-1]) + lambda*(solution->dissip2[j]);
46             Q3np1[j] = solution->Q3[j] - 0.5*lambda*(solution->E3[j+1] - solution->E3[j-1]) + lambda*(solution->dissip3[j]);
47         }
48         for (int j = 2; j < imax-2; j++){
49             solution->Q1[j] = Q1np1[j];
50             solution->Q2[j] = Q2np1[j];
51             solution->Q3[j] = Q3np1[j];
52         }
53
54         calcPrimitives(solution);
55         t = t + dt;
56         it++;
57
58         if ( (it%printAt == 0) || (fmod(tmax,t) >= 1.0) ){
59             printf("Iteration: %d, simulation time: %lf\\n",it,t);
60         }
61     }
62
63     /* deallocating variables */
64     free(Q1np1);
65     free(Q2np1);
66     free(Q3np1);

```

```

67
68     printf("\n\n");
69
70 }
71
72 /* lax-wendroff method */
73 void laxWendroff(results * solution, grid * mesh){
74     /* LAX-WENDROFF method using explicit
75      * Euler as time marching method.
76      * =====
77      * Qjnp1 = Qjn - 0.5*lambda*(Ejp1 - Ejm1)
78      *      + 0.5*lambda2*(Ajph*(Ejp1 - Ej) )
79      *      - 0.5*lambda2*(Ajmh*(Ej - Ejm1) )
80      * ===== */
81
82     double *Q1np1 = malloc(imax * sizeof(double));
83     double *Q2np1 = malloc(imax * sizeof(double));
84     double *Q3np1 = malloc(imax * sizeof(double));
85     int it = 0;
86     double dx = mesh->x[1] - mesh->x[0];
87     double q1, q2, q3;
88     double temp1, temp2, temp3;
89     double a = sqrt(gamma);
90     double dt = (dx*cfl)/a;
91     double t = 0.0;
92     double lambda = dt/dx;
93
94     printf("\n\n=====\\n");
95     printf("      Lax-Wendroff Scheme\\n");
96     printf("=====\\n\\n");
97
98     while (t <= (double) tmax && it < (int) itmax){
99         /* calculates the flux vector */
100        calcFluxes(solution);
101
102        /* calculates dissipation for the current solution Q */
103        calcDissipation(solution, lambda);
104

```

```

105      /* marching the solution */
106      for (int j = 2; j < imax-2; j++){
107          /* calculating the variables in point j+1/2 */
108          q1 = 0.5*(solution->Q1[j] + solution->Q1[j+1]);
109          q2 = 0.5*(solution->Q2[j] + solution->Q2[j+1]);
110          q3 = 0.5*(solution->Q3[j] + solution->Q3[j+1]);
111
112          calcJacobian(q1,q2,q3,solution);
113
114          temp1 = solution->Q1[j]
115              - 0.5*lambda*(solution->E1[j+1] - solution->E1[j-1])
116              + 0.5*lambda*lambda*(solution->A[0][0]*(solution->E1
117 [j+1]-solution->E1[j]) + solution->A[0][1]*(solution->E2[j+1]-
118 solution->E2[j]) + solution->A[0][2]*(solution->E3[j+1]-solution
119 ->E3[j]));
120
121          temp2 = solution->Q2[j]
122              - 0.5*lambda*(solution->E2[j+1] - solution->E2[j-1])
123              + 0.5*lambda*lambda*(solution->A[1][0]*(solution->E1
124 [j+1]-solution->E1[j]) + solution->A[1][1]*(solution->E2[j+1]-
125 solution->E2[j]) + solution->A[1][2]*(solution->E3[j+1]-solution
126 ->E3[j]));
127
128          temp3 = solution->Q3[j]
129              - 0.5*lambda*(solution->E3[j+1] - solution->E3[j-1])
130              + 0.5*lambda*lambda*(solution->A[2][0]*(solution->E1
131 [j+1]-solution->E1[j]) + solution->A[2][1]*(solution->E2[j+1]-
132 solution->E2[j]) + solution->A[2][2]*(solution->E3[j+1]-solution
133 ->E3[j]));
134
135          /* calculating the variables in point j-1/2 */
136          q1 = 0.5*(solution->Q1[j] + solution->Q1[j-1]);
137          q2 = 0.5*(solution->Q2[j] + solution->Q2[j-1]);
138          q3 = 0.5*(solution->Q3[j] + solution->Q3[j-1]);
139
140          calcJacobian(q1,q2,q3,solution);
141
142          Q1np1[j] = temp1

```

```

134             - 0.5*lambda*lambda*(solution->A[0][0]*(solution->E1
135             [j]-solution->E1[j-1]) + solution->A[0][1]*(solution->E2[j]-
136             solution->E2[j-1]) + solution->A[0][2]*(solution->E3[j]-solution
137             ->E3[j-1]))
138             + lambda*(solution->dissip1[j]);
139
140
141             Q2np1[j] = temp2
142             - 0.5*lambda*lambda*(solution->A[1][0]*(solution->E1
143             [j]-solution->E1[j-1]) + solution->A[1][1]*(solution->E2[j]-
144             solution->E2[j-1]) + solution->A[1][2]*(solution->E3[j]-solution
145             ->E3[j-1]))
146             + lambda*(solution->dissip2[j]);
147
148
149             Q3np1[j] = temp3
150             - 0.5*lambda*lambda*(solution->A[2][0]*(solution->E1
151             [j]-solution->E1[j-1]) + solution->A[2][1]*(solution->E2[j]-
152             solution->E2[j-1]) + solution->A[2][2]*(solution->E3[j]-solution
153             ->E3[j-1]))
154             + lambda*(solution->dissip3[j]);
155
156
157             }
158
159
160             calcPrimitives(solution);
161             t = t + dt;
162             it++;
163
164
165             if ( (it%printAt == 0) || (fmod(tmax,t) >= 1.0) ){
166                 printf("Iteration: %d, simulation time: %lf\n",it,t);
167             }
168
169
170             /* deallocating variables */
171             free(Q1np1);
172             free(Q2np1);

```

```

163     free (Q3np1) ;
164
165     printf ("\n\n") ;
166
167 }
168
169 /* classical predictor-corrector macCormack scheme */
170 void macCormack(results * solution , grid * mesh){
171     /* MACCORMACK method using explicit
172      * Euler as time marching method .
173      * =====
174      * predictor step
175      * bQjnp1 = Qjn - lambda*(Ejp1 - Ej)
176      *
177      * corrector step
178      * Qjnp1 = 0.5*(Qjn + bQjnp1 - lambda*(Ej - Ejm1)
179      * ===== */
180
181     double *Q1np1 = malloc (imax * sizeof(double)) ;
182     double *Q2np1 = malloc (imax * sizeof(double)) ;
183     double *Q3np1 = malloc (imax * sizeof(double)) ;
184     double bQ1np1 , bQ2np1 , bQ3np1 ;
185     int it = 0 ;
186     double dx = mesh->x [1] - mesh->x [0] ;
187     double a = sqrt (gamma) ;
188     double dt = (dx*cfl)/a ;
189     double t = 0.0 ;
190     double lambda = dt/dx ;
191
192     printf ("\n\n=====\\n") ;
193     printf ("          MacCormack Scheme\\n") ;
194     printf ("=====\\n\\n") ;
195
196     while ( t <= (double) tmax && it < (int) itmax ){
197         /* calculates the flux vector */
198         calcFluxes (solution) ;
199
200         /* calculates dissipation for the current solution Q */

```

```

201     calcDissipation(solution, lambda);

202

203     /* marching the solution */
204     for (int j = 2; j < imax-2; j++){
205         /* predictor step */
206         bQ1np1 = solution->Q1[j] - lambda*(solution->E1[j+1] -
207             solution->E1[j]);
208         bQ2np1 = solution->Q2[j] - lambda*(solution->E2[j+1] -
209             solution->E2[j]);
210         bQ3np1 = solution->Q3[j] - lambda*(solution->E3[j+1] -
211             solution->E3[j]);
212
213         /* corrector step */
214         Q1np1[j] = 0.5*(solution->Q1[j] + bQ1np1 - lambda*(
215             solution->E1[j] - solution->E1[j-1])) + lambda*(solution->
216             dissip1[j]);
217         Q2np1[j] = 0.5*(solution->Q2[j] + bQ2np1 - lambda*(
218             solution->E2[j] - solution->E2[j-1])) + lambda*(solution->
219             dissip2[j]);
220         Q3np1[j] = 0.5*(solution->Q3[j] + bQ3np1 - lambda*(
221             solution->E3[j] - solution->E3[j-1])) + lambda*(solution->
222             dissip3[j]);
223     }
224
225     for (int j = 2; j < imax-2; j++){
226         solution->Q1[j] = Q1np1[j];
227         solution->Q2[j] = Q2np1[j];
228         solution->Q3[j] = Q3np1[j];
229     }
230
231     calcPrimitives(solution);
232     t = t + dt;
233     it++;
234
235     if ( (it%printAt == 0) || (fmod(tmax,t) >= 1.0) ){
236         printf("Iteration: %d, simulation time: %lf\n",it,t);
237     }
238 }
```

```

230     /* deallocating variables */
231     free(Q1np1);
232     free(Q2np1);
233     free(Q3np1);
234
235     printf("\n\n");
236
237 }
238
239 /* steger and warming flux vector splitting scheme */
240 void stegerWarming(results * solution, grid * mesh){
241     /* STEGER AND WARMING flux vector splitting scheme
242      * using explicit Euler as time marching method.
243      * =====
244      * Qjnp1 = Qjn -
245      * ===== */
246
247     double *Q1np1 = malloc(imax * sizeof(double));
248     double *Q2np1 = malloc(imax * sizeof(double));
249     double *Q3np1 = malloc(imax * sizeof(double));
250     int it = 0;
251     double dx = mesh->x[1] - mesh->x[0];
252     double a = sqrt(gamma);
253     double dt = (dx*cfl)/a;
254     double t = 0.0;
255     double lambda = dt/dx;
256
257     printf("\n\n=====\\n");
258     printf(" Steger and Warming FVS Scheme\\n");
259     printf("=====\\n\\n");
260
261     while (t <= (double) tmax && it < (int) itmax){
262         /* calculates the flux vector */
263         calcStegerFluxes(solution);
264
265         /* calculates dissipation for the current solution Q */
266         //calcDissipation(solution, lambda);
267

```

```

268     /* marching the solution */
269     for (int j = 2; j < imax-2; j++){
270         switch (order){
271             case 1:
272                 Q1np1[j] = solution->Q1[j]
273                     - lambda*(solution->E1p[j] - solution->E1p[j
274                         -1])
275                     - lambda*(solution->Elm[j+1] - solution->Elm
276                         [j]);
277                 Q2np1[j] = solution->Q2[j]
278                     - lambda*(solution->E2p[j] - solution->E2p[j
279                         -1])
280                     - lambda*(solution->E2m[j+1] - solution->E2m
281                         [j]);
282                 Q3np1[j] = solution->Q3[j]
283                     - lambda*(solution->E3p[j] - solution->E3p[j
284                         -1])
285                     - lambda*(solution->E3m[j+1] - solution->E3m
286                         [j]);
287                     break;
288             case 2:
289                 Q1np1[j] = solution->Q1[j]
290                     - (lambda/2.0)*(3.0*solution->E1p[j] - 4.0*
291                     solution->E1p[j-1] + solution->E1p[j-2])
292                     - (lambda/2.0)*(-3.0*solution->Elm[j] + 4.0*
293                     solution->Elm[j+1] - solution->Elm[j+2]);
294                 Q2np1[j] = solution->Q2[j]
295                     - (lambda/2.0)*(3.0*solution->E2p[j] - 4.0*
296                     solution->E2p[j-1] + solution->E2p[j-2])
297                     - (lambda/2.0)*(-3.0*solution->E2m[j] + 4.0*
298                     solution->E2m[j+1] - solution->E2m[j+2]);
299                 Q3np1[j] = solution->Q3[j]
300                     - (lambda/2.0)*(3.0*solution->E3p[j] - 4.0*
301                     solution->E3p[j-1] + solution->E3p[j-2])
302                     - (lambda/2.0)*(-3.0*solution->E3m[j] + 4.0*
303                     solution->E3m[j+1] - solution->E3m[j+2]);
304                     break;
305             default:

```

```

294         Q1np1[j] = solution->Q1[j]
295             - lambda*(solution->E1p[j] - solution->E1p[j
296             -1])
297             - lambda*(solution->E1m[j+1] - solution->E1m
298             [j]);
299
300         Q2np1[j] = solution->Q2[j]
301             - lambda*(solution->E2p[j] - solution->E2p[j
302             -1])
303             - lambda*(solution->E2m[j+1] - solution->E2m
304             [j]);
305
306         Q3np1[j] = solution->Q3[j]
307             - lambda*(solution->E3p[j] - solution->E3p[j
308             -1])
309             - lambda*(solution->E3m[j+1] - solution->E3m
310             [j]);
311
312         break;
313     }
314
315     for (int j = 2; j < imax-2; j++){
316         solution->Q1[j] = Q1np1[j];
317         solution->Q2[j] = Q2np1[j];
318         solution->Q3[j] = Q3np1[j];
319     }
320
321     calcPrimitives(solution);
322     t = t + dt;
323     it++;
324
325     if ( (it%printAt == 0) || (fmod(tmax,t) >= 1.0) ){
326         printf("Iteration: %d, simulation time: %lf\n",it,t);
327     }
328
329     /* deallocating variables */
330     free(Q1np1);
331     free(Q2np1);
332     free(Q3np1);

```

```

326
327     printf("\n\n");
328
329 }
330
331 /* van Leer non-MUSCL flux vector splitting scheme */
332 void vanLeerNonMUSCL(results * solution, grid * mesh){
333     /* STEGER AND WARMING flux vector splitting scheme
334     * using explicit Euler as time marching method.
335     * =====
336     * Qjnp1 = Qjn -
337     * ===== */
338
339     double *Q1np1 = malloc(imax * sizeof(double));
340     double *Q2np1 = malloc(imax * sizeof(double));
341     double *Q3np1 = malloc(imax * sizeof(double));
342     int it = 0;
343     double dx = mesh->x[1] - mesh->x[0];
344     double a = sqrt(gamma);
345     double dt = (dx*cfl)/a;
346     double t = 0.0;
347     double lambda = dt/dx;
348
349     printf("\n=====\n");
350     printf(" van Leer non-MUSCL FVS Scheme\n");
351     printf("=====\n\n");
352
353     while (t <= (double) tmax && it < (int) itmax){
354         /* calculates the flux vector */
355         calcVanLeerFluxes(solution);
356
357         /* calculates dissipation for the current solution Q */
358         //calcDissipation(solution, lambda);
359
360         /* marching the solution */
361         for (int j = 2; j < imax-2; j++){
362             switch (order){
363                 case 1:

```

```

364         Q1np1[j] = solution->Q1[j]
365             - lambda*(solution->E1p[j] - solution->E1p[j
366             -1])
367             - lambda*(solution->Elm[j+1] - solution->Elm
368             [j]);
369         Q2np1[j] = solution->Q2[j]
370             - lambda*(solution->E2p[j] - solution->E2p[j
371             -1])
372             - lambda*(solution->E2m[j+1] - solution->E2m
373             [j]);
374         Q3np1[j] = solution->Q3[j]
375             - lambda*(solution->E3p[j] - solution->E3p[j
376             -1])
377             - lambda*(solution->E3m[j+1] - solution->E3m
378             [j]);
379             break;
380         case 2:
381             Q1np1[j] = solution->Q1[j]
382                 - (lambda/2.0)*(3.0*solution->E1p[j] - 4.0*
383                     solution->E1p[j-1] + solution->E1p[j-2])
384                     - (lambda/2.0)*(-3.0*solution->Elm[j] + 4.0*
385                         solution->Elm[j+1] - solution->Elm[j+2]);
386             Q2np1[j] = solution->Q2[j]
387                 - (lambda/2.0)*(3.0*solution->E2p[j] - 4.0*
388                     solution->E2p[j-1] + solution->E2p[j-2])
389                     - (lambda/2.0)*(-3.0*solution->E2m[j] + 4.0*
390                         solution->E2m[j+1] - solution->E2m[j+2]);
391             Q3np1[j] = solution->Q3[j]
392                 - (lambda/2.0)*(3.0*solution->E3p[j] - 4.0*
393                     solution->E3p[j-1] + solution->E3p[j-2])
394                     - (lambda/2.0)*(-3.0*solution->E3m[j] + 4.0*
395                         solution->E3m[j+1] - solution->E3m[j+2]);
396             break;
397         default:
398             Q1np1[j] = solution->Q1[j]
399                 - lambda*(solution->E1p[j] - solution->E1p[j
400                 -1])
401                 - lambda*(solution->Elm[j+1] - solution->Elm

```

```

[j]) ;

389         Q2np1[j] = solution->Q2[j]
390             - lambda*(solution->E2p[j] - solution->E2p[j
391             -1])
392                 - lambda*(solution->E2m[j+1] - solution->E2m
393 [j]) ;
394         Q3np1[j] = solution->Q3[j]
395             - lambda*(solution->E3p[j] - solution->E3p[j
396             -1])
397                 - lambda*(solution->E3m[j+1] - solution->E3m
398 [j]) ;
399         break;
400     }
401 }
402
403 for (int j = 2; j < imax-2; j++){
404     solution->Q1[j] = Q1np1[j];
405     solution->Q2[j] = Q2np1[j];
406     solution->Q3[j] = Q3np1[j];
407 }
408
409 calcPrimitives(solution);
410 t = t + dt;
411 it++;
412
413 if ( (it%printAt == 0) || (fmod(tmax,t) >= 1.0) ){
414     printf("Iteration: %d, simulation time: %lf\n",it ,t );
415 }
416
417 /* deallocating variables */
418 free(Q1np1);
419 free(Q2np1);
420 free(Q3np1);
421 }
```

```

422
423 /* liou AUSM+ flux vector splitting scheme */
424 void liouAUSMplus(results * solution, grid * mesh){
425     /* STEGER AND WARMING flux vector splitting scheme
426     * using explicit Euler as time marching method.
427     * =====
428     * Qjnp1 = Qjn -
429     * ===== */
430
431     double *Q1np1 = malloc(imax * sizeof(double));
432     double *Q2np1 = malloc(imax * sizeof(double));
433     double *Q3np1 = malloc(imax * sizeof(double));
434     int it = 0;
435     double dx = mesh->x[1] - mesh->x[0];
436     double a = sqrt(gamma);
437     double dt = (dx*cfl)/a;
438     double t = 0.0;
439     double lambda = dt/dx;
440
441     printf("\n\n=====\\n");
442     printf("      Liou AUSM+ FVS Scheme\\n");
443     printf("=====\\n\\n");
444
445     while (t <= (double) tmax && it < (int) itmax){
446         /* calculates the flux vector */
447         calcFluxes(solution);
448         calcLiouFluxes(solution);
449
450         /* calculates dissipation for the current solution Q */
451         //calcDissipation(solution, lambda);
452
453         /* marching the solution */
454         for (int j = 2; j < imax-2; j++){
455             Q1np1[j] = solution->Q1[j]
456                 - lambda*(solution->E1p[j] - solution->E1p[j-1]);
457             Q2np1[j] = solution->Q2[j]
458                 - lambda*(solution->E2p[j] - solution->E2p[j-1]);
459             Q3np1[j] = solution->Q3[j]

```

```

460             - lambda*(solution->E3p[j] - solution->E3p[j-1]);
461         }
462
463         for (int j = 2; j < imax-2; j++){
464             solution->Q1[j] = Q1np1[j];
465             solution->Q2[j] = Q2np1[j];
466             solution->Q3[j] = Q3np1[j];
467         }
468
469         calcPrimitives(solution);
470         t = t + dt;
471         it++;
472
473         if ((it%printAt == 0) || (fmod(tmax,t) >= 1.0)){
474             printf("Iteration: %d, simulation time: %lf\n",it,t);
475         }
476     }
477
478     /* deallocating variables */
479     free(Q1np1);
480     free(Q2np1);
481     free(Q3np1);
482
483     printf("\n\n");
484
485 }
486
487 /* Roe approximate Riemann solver method */
488 void roeMethod(results * solution, grid * mesh){
489     /* ROE approximate riemann solver method
490      * using explicit Euler as time marching method.
491      * =====
492      * Qjnp1 = Qjn -
493      * ===== */
494
495     double *Q1np1 = malloc(imax * sizeof(double));
496     double *Q2np1 = malloc(imax * sizeof(double));
497     double *Q3np1 = malloc(imax * sizeof(double));

```

```

498     int it = 0;
499     double dx = mesh->x[1] - mesh->x[0];
500     double a = sqrt(gamma);
501     double dt = (dx*cfl)/a;
502     double t = 0.0;
503     double lambda = dt/dx;
504
505     printf("\n\n");
506     printf("      Roe's approximate Riemann solver      \n");
507     printf("=\n");
508
509     while (t <= (double) tmax && it < (int) itmax){
510         /* calculates the flux vector */
511         calcFluxes(solution);
512         calcRoeFluxes(solution);
513
514         /* calculates dissipation for the current solution Q */
515         //calcDissipation(solution, lambda);
516
517         /* marching the solution */
518         for (int j = 2; j < imax-2; j++){
519             Q1np1[j] = solution->Q1[j]
520                 - lambda*(solution->E1p[j] - solution->E1p[j-1]);
521             Q2np1[j] = solution->Q2[j]
522                 - lambda*(solution->E2p[j] - solution->E2p[j-1]);
523             Q3np1[j] = solution->Q3[j]
524                 - lambda*(solution->E3p[j] - solution->E3p[j-1]);
525         }
526
527         for (int j = 2; j < imax-2; j++){
528             solution->Q1[j] = Q1np1[j];
529             solution->Q2[j] = Q2np1[j];
530             solution->Q3[j] = Q3np1[j];
531         }
532
533         calcPrimitives(solution);
534         t = t + dt;
535         it++;

```

```

536
537     if ( (it%printAt == 0) || (fmod(tmax,t) >= 1.0)) {
538         printf("Iteration: %d, simulation time: %lf\n", it, t);
539     }
540 }
541
542 /* deallocating variables */
543 free(Q1np1);
544 free(Q2np1);
545 free(Q3np1);
546
547 printf("\n\n");
548
549 }
550
551 /* harten TVD scheme */
552 void hartenTVD(results * solution, grid * mesh){
553     /* HARTEN TVD scheme
554     * using explicit Euler as time marching method.
555     * =====
556     * Qjnp1 = Qjn -
557     * ===== */
558
559     double *Q1np1 = malloc(imax * sizeof(double));
560     double *Q2np1 = malloc(imax * sizeof(double));
561     double *Q3np1 = malloc(imax * sizeof(double));
562     int it = 0;
563     double dx = mesh->x[1] - mesh->x[0];
564     double a = sqrt(gamma);
565     double dt = (dx*cfl)/a;
566     double t = 0.0;
567     double lambda = dt/dx;
568
569     printf("\n=====\\n");
570     printf("      Harten's TVD scheme      \\n");
571     printf("=====\\n");
572
573     while (t <= (double) tmax && it < (int) itmax) {

```

```

574     /* calculates the flux vector */
575     calcFluxes(solution);
576     calcHartenFluxes(solution, lambda);
577
578     /* calculates dissipation for the current solution Q */
579     //calcDissipation(solution, lambda);
580
581     /* marching the solution */
582     for (int j = 2; j < imax-2; j++){
583         Q1np1[j] = solution->Q1[j]
584             - lambda*(solution->E1p[j] - solution->E1p[j-1]);
585         Q2np1[j] = solution->Q2[j]
586             - lambda*(solution->E2p[j] - solution->E2p[j-1]);
587         Q3np1[j] = solution->Q3[j]
588             - lambda*(solution->E3p[j] - solution->E3p[j-1]);
589     }
590
591     for (int j = 2; j < imax-2; j++){
592         solution->Q1[j] = Q1np1[j];
593         solution->Q2[j] = Q2np1[j];
594         solution->Q3[j] = Q3np1[j];
595     }
596
597     calcPrimitives(solution);
598     t = t + dt;
599     it++;
600
601     if ( (it%printAt == 0) || (fmod(tmax,t) >= 1.0) ){
602         printf("Iteration: %d, simulation time: %lf\n",it,t);
603     }
604 }
605
606     /* deallocating variables */
607     free(Q1np1);
608     free(Q2np1);
609     free(Q3np1);
610
611     printf("\n\n");

```

```

612
613 }
614
615
616 /* exact solution */
617 void exactSolution ( results * solution , grid * mesh){
618
619     double pr1 ,pr2 ,pr3 ,pr4 ,pr6 ;
620     double rhor1 ,rhor2 ,rhor3 ,rhor4 ,rhor6 ;
621     double ur1 ,ur2 ,ur3 ,ur4 ,ur6 ;
622     double A6, B6;
623     double g = gamma;
624     double x, x0, t ;
625     double a1, a3, a6 ;
626     double u_tail , u_head , u_contact , u_shock ;
627     double f , fl , pi , pOld , pNew ;
628
629     solution->rho = malloc (imax * sizeof(double));
630     solution->press = malloc (imax * sizeof(double));
631     solution->vel = malloc (imax * sizeof(double));
632
633     x0 = 0.0;
634     t = tmax;
635
636     pr1 = p4;
637     rhor1 = p4 ;
638     ur1 = 0.0;
639     a1 = sqrt ( g * pr1/rhor1 ) ;
640
641     rhor6 = p1 ;
642     pr6 = p1 ;
643     ur6 = 0.0;
644     a6 = sqrt ( g * pr6/rhor6 ) ;
645
646     A6 = 2.0 / ( rhor6 * (g + 1.0) );
647     B6 = pr6 * (g-1.0) / (g+1.0);
648
649     pi = (pr1+pr6)/10.0;

```

```

650     pNew = pi;
651     pOld = 0.0;
652
653     f = ( pi - pr6 ) * sqrt( A6 / (pi + B6)) + 2.0 * a1 / (g-1.0) * (
654         pow( (pi/pr1),((g-1.0)/(2.0*g)) ) - 1.0 );
655
656     fl = sqrt( A6 / (pi + B6) ) - 0.5 * A6 * (pi - pr6) * sqrt( (pi +
657         B6)/A6 ) / ((pi + B6)*(pi + B6)) + a1 / (g*pr1) * (pow ((pi/pr1)
658         ,(-g-1.0)/(2.0*g) ) );
659
660     // iterative method
661
662     while (fabs(pNew - pOld) > tol){
663         pOld = pi;
664         pNew = pi - f / fl;
665         pi = pNew;
666
667         f = ( pi - pr6 ) * sqrt( A6 / (pi + B6)) + 2.0 * a1 / (g-1.0) * (
668             pow( (pi/pr1),((g-1.0)/(2.0*g)) ) - 1.0 ) + ur6 - ur1;
669
670         fl = sqrt( A6 / (pi + B6) ) - 0.5 * A6 * (pi - pr6) * sqrt( (pi +
671             B6)/A6 ) / ((pi + B6)*(pi + B6)) + a1 / (g*pr1) * (pow ((pi/
672             pr1),(-g-1.0)/(2.0*g) ) );
673     }
674
675     pr3 = pNew;
676
677     rhor3 = rhor1 * pow( (pr3/pr1), 1.0/g );
678
679     ur3 = ur1 - ( 2.0 * a1 / (g - 1.0) ) * (pow( (pr3/pr1), ((g-1.0)
680         /(2.0*g)) ) -1.0 );
681
682     a3 = sqrt( g * pr3 / rhor3 );
683
684     pr4 = pr3;
685
686     rhor4 = rhor6 * ((pr6 * (g-1.0) + pr4 * (g+1.0)) / (pr4 * (g-1.0)
687         + pr6 * (g+1.0)) );

```

```

680
681     ur4 = ur6 + (pr4 - pr6) * sqrt( A6 / (pr4 + B6)) ;
682
683     u_head = ur1 - a1;
684
685     u_tail = ur3 - a3;
686
687     u_contact = ur3;
688
689     u_shock = ur6 + a6 * sqrt( (g+1.0)*pr4 / (2.0 * g * pr6) + (g-1.0)
700                                     /(2.0*g) );
701
702
703
704
705
706
707
708
709
710
711
712

```

```

713     solution->vel[ i ] = ur2;
714     //solution->s[ i ] = solution->press[ i ] / ( pow( solution->rho[ i
715     ], GAMMA) );
716
717     // Region 3
718     if ( ( t * u_tail ) < ( x - x0 ) && ( x - x0 ) <= t * u_contact )
719     {
720         solution->press[ i ] = pr3;
721         solution->rho[ i ] = rhor3;
722         solution->vel[ i ] = ur3;
723         //solution->s[ i ] = solution->press[ i ] / ( pow( solution->rho[ i
724         ], GAMMA) );
725     }
726
727     // Region 4
728     if ( t * u_contact < ( x - x0 ) && ( x - x0 ) <= t * u_shock ){
729         solution->press[ i ] = pr4;
730         solution->rho[ i ] = rhor4;
731         solution->vel[ i ] = ur4;
732         //solution->s[ i ] = sol->press[ i ] / ( pow( solution->rho[ i ],
733         GAMMA) );
734     }
735
736     // Region 5 - Shock!
737
738     // Region 6
739     if ( t * u_shock < x - x0 ){
740         solution->press[ i ] = pr6;
741         solution->rho[ i ] = rhor6;
742         solution->vel[ i ] = ur6;
743         //solution->s[ i ] = solution->press[ i ] / ( pow( solution->rho[ i
744         ], GAMMA) );
745     }

```

```

746     // free( solution->press );
747     // free( solution->vel );
748     // free( solution->rho );
749
750     return;
751 }
```

Aux.h file:

```

1 #ifndef AUX_H
2 #define AUX_H
3
4 typedef struct results{
5     double *press;
6     double *vel;
7     double *rho;
8     double *energy;
9     double *iEnergy;
10    double *Q1;
11    double *Q2;
12    double *Q3;
13    double *E1;
14    double *E2;
15    double *E3;
16    double *E1p;
17    double *E2p;
18    double *E3p;
19    double *Elm;
20    double *E2m;
21    double *E3m;
22    double *dissip1;
23    double *dissip2;
24    double *dissip3;
25    double **A;
26 } results;
27
28 /* AUXILIARY FUNCTIONS AND DEFINITIONS */
29
30 void allocSolution( results * );
```

```

31 void freeSolution(results *);
32 void initSolution(results *);
33 /*
34 void jacobian(results *);
35 */
36 void calcPrimitives(results *);
37 double max(double a, double b);
38 double min(double a, double b);
39 double ksiHarten(double a, double eps);
40 void calcFluxes(results *);
41 void calcStegerFluxes(results *);
42 void calcVanLeerFluxes(results *);
43 void calcLiouFluxes(results *);
44 void calcRoeFluxes(results *);
45 void calcHartenFluxes(results *, double lambda);
46 void calcDissipation(results *, double lambda);
47 void calcJacobian(double q1, double q2, double q3, results *);
48
49 #endif

```

Aux.c file:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4
5 #include "input.h"
6 #include "aux.h"
7 #include "mesh.h"
8
9 void allocSolution(results * solution){
10     solution->press = malloc(imax * sizeof(double));
11     solution->vel = malloc(imax * sizeof(double));
12     solution->rho = malloc(imax * sizeof(double));
13     solution->energy = malloc(imax * sizeof(double));
14     solution->iEnergy = malloc(imax * sizeof(double));
15     solution->Q1 = malloc(imax * sizeof(double));
16     solution->Q2 = malloc(imax * sizeof(double));
17     solution->Q3 = malloc(imax * sizeof(double));

```

```

18     solution->E1 = malloc(imax * sizeof(double));
19     solution->E2 = malloc(imax * sizeof(double));
20     solution->E3 = malloc(imax * sizeof(double));
21     solution->E1p = malloc(imax * sizeof(double));
22     solution->E2p = malloc(imax * sizeof(double));
23     solution->E3p = malloc(imax * sizeof(double));
24     solution->E1m = malloc(imax * sizeof(double));
25     solution->E2m = malloc(imax * sizeof(double));
26     solution->E3m = malloc(imax * sizeof(double));
27     solution->dissip1 = malloc(imax * sizeof(double));
28     solution->dissip2 = malloc(imax * sizeof(double));
29     solution->dissip3 = malloc(imax * sizeof(double));
30     solution->A = malloc(3 * sizeof(double *));
31
32     for (int j = 0; j < 3; j++){
33         solution->A[j] = malloc(3 * sizeof(double));
34     }
35 }
36
37 void freeSolution(results * solution){
38     free(solution->press);
39     free(solution->vel);
40     free(solution->rho);
41     free(solution->energy);
42     free(solution->iEnergy);
43     free(solution->Q1);
44     free(solution->Q2);
45     free(solution->Q3);
46     free(solution->E1);
47     free(solution->E2);
48     free(solution->E3);
49     free(solution->E1p);
50     free(solution->E2p);
51     free(solution->E3p);
52     free(solution->E1m);
53     free(solution->E2m);
54     free(solution->E3m);
55     free(solution->dissip1);

```

```

56     free ( solution->dissip2 );
57     free ( solution->dissip3 );
58
59     for ( int j = 0; j < 3; j++ ){
60         free ( solution->A[j] );
61     }
62     free ( solution->A );
63 }
64
65 void initSolution( results * solution ){
66     int mid = (imax - 1)/2;
67
68     for ( int i = 0; i < imax; i++ ){
69         if ( i <= mid ){
70             solution->press [ i ] = p4;
71             solution->rho [ i ] = p4;
72             solution->energy [ i ] = p4/(gamma - 1.0);
73             solution->iEnergy [ i ] = (p4/(gamma - 1.0))/solution->rho [
74                 i ];
75         }
76         else {
77             solution->press [ i ] = p1;
78             solution->rho [ i ] = p1;
79             solution->energy [ i ] = p1/(gamma - 1.0);
80             solution->iEnergy [ i ] = (p1/(gamma - 1.0))/solution->rho [
81                 i ];
82         }
83     }
84
85     for ( int i = 0; i < imax; i++ ){
86         solution->vel [ i ] = U0;
87     }
88
89     for ( int i = 0; i < imax; i++ ){
90         solution->Q1[ i ] = solution->rho [ i ];
91         solution->Q2[ i ] = solution->rho [ i ] * solution->vel [ i ];
92         solution->Q3[ i ] = solution->energy [ i ];
93     }

```

```

92
93     for (int i = 0; i < imax; i++){
94         solution->E1[i] = solution->Q2[i];
95         solution->E2[i] = (solution->Q2[i] * solution->Q2[i]) /
96             solution->Q1[i] + solution->press[i];
97         solution->E3[i] = (solution->Q3[i] + solution->press[i]) *
98             solution->vel[i];
99     }
100
101    for (int i = 0; i < imax; i++){
102        solution->dissip1[i] = 0.0;
103        solution->dissip2[i] = 0.0;
104        solution->dissip3[i] = 0.0;
105        solution->E1p[i] = 0.0;
106        solution->E2p[i] = 0.0;
107        solution->E3p[i] = 0.0;
108        solution->E1m[i] = 0.0;
109        solution->E2m[i] = 0.0;
110        solution->E3m[i] = 0.0;
111    }
112
113    /* initializing the jacobian matrix */
114    solution->A[0][0] = 0.0;
115    solution->A[0][1] = 1.0;
116    solution->A[0][2] = 0.0;
117    solution->A[1][0] = 0.0;
118    solution->A[1][1] = 0.0;
119    solution->A[1][2] = 0.0;
120    solution->A[2][0] = 0.0;
121    solution->A[2][1] = 0.0;
122    solution->A[2][2] = 0.0;
123 }
124
125 void calcPrimitives(results * solution){
126     double temp=0.0;
127     for (int j = 0; j < imax; j++){

```

```

128     solution->rho[j] = solution->Q1[j];
129     solution->vel[j] = solution->Q2[j] / solution->Q1[j];
130     temp = solution->Q3[j] - 0.5*(solution->Q2[j] * solution->Q2
131 [j])/solution->Q1[j];
132     solution->press[j] = temp*(gamma-1.0);
133     solution->iEnergy[j] = temp/solution->Q1[j];
134 }
135
136 }
137
138 void calcFluxes(results * solution){
139     for (int j = 0; j < imax; j++){
140         solution->E1[j] = solution->Q2[j];
141         solution->E2[j] = (solution->Q2[j] * solution->Q2[j])/
142 solution->Q1[j] + solution->press[j];
143         solution->E3[j] = (solution->Q3[j] + solution->press[j]) *
144 solution->vel[j];
145     }
146 }
147
148 void calcStegerFluxes(results * solution){
149     double a, u, p, rho;
150     double lbd1, lbd2, lbd3;
151     double lbd1p, lbd2p, lbd3p;
152     double lbd1m, lbd2m, lbd3m;
153     double gm = gamma;
154
155     for (int j = 0; j < imax; j++){
156         rho = solution->rho[j];
157         u = solution->vel[j];
158         p = solution->press[j];
159
160         a = sqrt(gamma*(p/rho));
161
162         lbd1 = u;

```

```

163     lbd2 = u + a;
164     lbd3 = u - a;
165
166     /* computing E+ fluxes */
167     lbd1p = 0.5*(lbd1 + fabs(lbd1));
168     lbd2p = 0.5*(lbd2 + fabs(lbd2));
169     lbd3p = 0.5*(lbd3 + fabs(lbd3));
170
171     solution->E1p[j] = (rho/(2.0*gm))*(2.0*(gm-1.0)*lbd1p
172                                         + lbd2p + lbd3p);
173     solution->E2p[j] = (rho/(2.0*gm))*(2.0*(gm-1.0)*lbd1p*u
174                                         + lbd2p*(u+a) + lbd3p*(u-a));
175     solution->E3p[j] = (rho/(2.0*gm))*((gm-1.0)*lbd1p*u*u
176                                         + 0.5*lbd2p*(u+a)*(u+a)
177                                         + 0.5*lbd3p*(u-a)*(u-a)
178                                         + ((3.0-gm)*(lbd2p + lbd3p)*a*a)/(2.0*(gm-1.0)));
179
180     /* computing E- fluxes */
181     lbd1m = 0.5*(lbd1 - fabs(lbd1));
182     lbd2m = 0.5*(lbd2 - fabs(lbd2));
183     lbd3m = 0.5*(lbd3 - fabs(lbd3));
184
185     solution->E1m[j] = (rho/(2.0*gm))*(2.0*(gm-1.0)*lbd1m
186                                         + lbd2m + lbd3m);
187     solution->E2m[j] = (rho/(2.0*gm))*(2.0*(gm-1.0)*lbd1m*u
188                                         + lbd2m*(u+a) + lbd3m*(u-a));
189     solution->E3m[j] = (rho/(2.0*gm))*((gm-1.0)*lbd1m*u*u
190                                         + 0.5*lbd2m*(u+a)*(u+a)
191                                         + 0.5*lbd3m*(u-a)*(u-a)
192                                         + ((3.0-gm)*(lbd2m + lbd3m)*a*a)/(2.0*(gm-1.0)));
193 }
194 //printf("11: %f, 12: %f, 13: %f\n",lbd1,lbd2,lbd3);
195
196 return;
197 }
198
199 void calcVanLeerFluxes( results * solution){
200     double a, u, p, rho, M;

```

```

201     double fmassp , fmassm;
202     double gm = gamma;
203
204     for (int j = 0; j < imax; j++){
205         rho = solution->rho[j];
206         u = solution->vel[j];
207         p = solution->press[j];
208
209         a = sqrt(gamma*(p/rho));
210
211         M = u/a;
212         fmassp = (rho*a/4.0)*(M + 1.0)*(M + 1.0);
213         fmassm = -(rho*a/4.0)*(M - 1.0)*(M - 1.0);
214
215         if (M <= -1.0){
216             solution->E1p[j] = 0.0;
217             solution->E2p[j] = 0.0;
218             solution->E3p[j] = 0.0;
219
220             solution->E1m[j] = solution->Q2[j];
221             solution->E2m[j] = (solution->Q2[j] * solution->Q2[j]) /
222             solution->Q1[j] + solution->press[j];
223             solution->E3m[j] = (solution->Q3[j] + solution->press[j]
224             ])*solution->vel[j];
225         }
226         else if (M >= 1.0) {
227             solution->E1m[j] = 0.0;
228             solution->E2m[j] = 0.0;
229             solution->E3m[j] = 0.0;
230
231             solution->E1p[j] = solution->Q2[j];
232             solution->E2p[j] = (solution->Q2[j] * solution->Q2[j]) /
233             solution->Q1[j] + solution->press[j];
234             solution->E3p[j] = (solution->Q3[j] + solution->press[j]
235             ])*solution->vel[j];
236         }
237         else {
238             solution->E1p[j] = fmassp;

```

```

235         solution->E2p[j] = fmassp*(1.0/gm)*((gm-1.0)*u + 2.0*a);
236         solution->E3p[j] = fmassp*(
237             ((gm-1.0)*u + 2.0*a)*((gm-1.0)*u + 2.0*a)
238             /(2.0*(gm*gm-1.0)));
239
240         solution->E1m[j] = fmassm;
241         solution->E2m[j] = fmassm*(1.0/gm)*((gm-1.0)*u - 2.0*a);
242         solution->E3m[j] = fmassm*(
243             ((gm-1.0)*u - 2.0*a)*((gm-1.0)*u - 2.0*a)
244             /(2.0*(gm*gm-1.0)));
245     }
246 }
247 //printf("11: %f, 12: %f, 13: %f\n",lbd1,lbd2,lbd3);
248
249 return;
250 }
251
252 void calcLiouFluxes( results * solution ){
253
254     double phiL[3], phiR[3];
255     double hL, hR;
256     double rhoL, rhoR;
257
258     double aL, aR;
259     double aCriticalL, aCriticalR;
260     double ajph;
261
262     double uL, uR;
263     double pjph;
264     double pL, pLplus, pR, pRminus;
265     double pAlphaPlus, pAlphaMinus;
266
267     double ML, MR;
268     double MLplus, MRminus;
269     double MbetaPlus, MbetaMinus;
270     double mach;
271 //double machPlus, machMinus;
272

```

```

273     double alpha = 3.0/16.0;
274     double beta = 1.0/8.0;
275
276     for (int j = 1; j < imax-2; j++){
277         uL = solution->vel[j];
278         uR = solution->vel[j+1];
279         pL = solution->press[j];
280         pR = solution->press[j+1];
281         rhoL = solution->Q1[j];
282         rhoR = solution->Q1[j+1];
283         hL = (solution->Q3[j]+solution->press[j])/solution->Q1[j];
284         hR = (solution->Q3[j+1]+solution->press[j+1])/solution->Q1[j
285             +1];
286
287         /* computing phiL and phiR */
288         phiL[0] = solution->Q1[j];
289         phiL[1] = solution->Q2[j];
290         phiL[2] = solution->Q3[j] + pL;
291
292         phiR[0] = solution->Q1[j+1];
293         phiR[1] = solution->Q2[j+1];
294         phiR[2] = solution->Q3[j+1] + pR;
295
296         /* computing the sound speed at interface j+1/2 */
297         aCriticalL = sqrt( (2.0*(gamma-1.0)/(gamma+1.0))*hL );
298         aCriticalR = sqrt( (2.0*(gamma-1.0)/(gamma+1.0))*hR );
299
300         switch (soundSpeedType){
301             case 1:
302                 aL = aCriticalL * min( 1.0, (aCriticalL/fabs(uL)) );
303                 aR = aCriticalR * min( 1.0, (aCriticalR/fabs(uR)) );
304                 break;
305             case 2:
306                 aL = sqrt( gamma* (pL/rhoL) );
307                 aR = sqrt( gamma* (pR/rhoR) );
308                 break;
309             default :
310                 aL = aCriticalL * min( 1.0, (aCriticalL/fabs(uL)) );

```

```

310             aR = aCriticalR * min( 1.0 , (aCriticalR/fabs(uR)) ) ;
311             break ;
312         }
313
314         switch (interfaceSoundSpeed){
315             case 1:
316                 ajph = min( aL,aR ) ;
317                 break ;
318             case 2:
319                 ajph = 0.5*( aL + aR ) ;
320                 break ;
321             case 3:
322                 ajph = sqrt( aL * aR ) ;
323                 break ;
324             case 4:
325                 //reserved for roe averages
326                 break ;
327             default:
328                 ajph = min( aL,aR ) ;
329                 break ;
330         }
331
332         /* computing (M,p)j+1/2 */
333         ML = fabs( uL ) / ajph ;
334         MR = fabs( uR ) / ajph ;
335
336         MbetaPlus = 0.25*(ML + 1.0)*(ML + 1.0)
337             + beta*(ML*ML - 1.0)*(ML*ML - 1.0) ;
338         MbetaMinus = -0.25*(MR - 1.0)*(MR - 1.0)
339             - beta*(MR*MR - 1.0)*(MR*MR - 1.0) ;
340
341         pAlphaPlus = 0.25*(ML + 1.0)*(ML + 1.0)*(2.0 - ML)
342             + alpha*ML*(ML*ML - 1.0)*(ML*ML - 1.0) ;
343         pAlphaMinus = 0.25*(MR - 1.0)*(MR - 1.0)*(2.0 + MR)
344             - alpha*MR*(MR*MR - 1.0)*(MR*MR - 1.0) ;
345
346         if ( fabs(ML) >= 1.0 ){
347             MLplus = 0.5*( ML + fabs(ML) ) ;

```

```

348     pLplus = 0.5*( 1.0 + (ML / fabs(ML)) ) ;
349 }
350 else {
351     MLplus = MbtaPlus;
352     pLplus = pAlphaPlus;
353 }
354
355 if ( fabs(MR) >= 1.0 ){
356     MRminus = 0.5*( MR - fabs(MR) );
357     pRminus = 0.5*( 1.0 - (MR / fabs(MR)) );
358 }
359 else {
360     MRminus = MbtaMinus;
361     pRminus = pAlphaMinus;
362 }
363
364 mach = MLplus + MRminus;
365 //machPlus = 0.5*( mach + fabs(mach) );
366 //machMinus = 0.5*( mach - fabs(mach) );
367 pjph = pLplus*pL + pRminus*pR;
368
369 /* computing the fluxes at j+1/2 */
370 solution->E1p[j] = 0.5 * ajph * mach * (phiL[0] + phiR[0])
371     - 0.5 * ajph * fabs( mach )*( phiR[0] - phiL[0] );
372 solution->E2p[j] = 0.5 * ajph * mach * (phiL[1] + phiR[1])
373     - 0.5 * ajph * fabs( mach )*( phiR[1] - phiL[1] ) + pjph;
374 solution->E3p[j] = 0.5 * ajph * mach * (phiL[2] + phiR[2])
375     - 0.5 * ajph * fabs( mach )*( phiR[2] - phiL[2] );
376
377 }
378
379 return;
380 }
381
382 void calcRoeFluxes( results * solution ){
383
384     double uL, uR;
385     double pL, pR;

```

```

386     double rhoL, rhoR;
387     double hL, hR;
388     double uRoe, rhoRoe, hRoe, aRoe;
389
390     double lbd1, lbd2, lbd3;
391     double r1[3], r2[3], r3[3];
392     double deltaQ[3];
393     double dp, du, drho;
394     double alpha1, alpha2, alpha3;
395
396     for (int j = 1; j < imax-2; j++){
397         uL = solution->vel[j];
398         uR = solution->vel[j+1];
399         pL = solution->press[j];
400         pR = solution->press[j+1];
401         rhoL = solution->rho[j];
402         rhoR = solution->rho[j+1];
403         hL = (solution->Q3[j] + solution->press[j])/solution->Q1[j];
404         hR = (solution->Q3[j+1] + solution->press[j+1])/solution->Q1
405             [j+1];
406         du = uR - uL;
407         dp = pR - pL;
408         drho = rhoR - rhoL;
409
410         /* computing roe averaged variables */
411         rhoRoe = sqrt( rhoL * rhoR );
412         uRoe = (sqrt( rhoL ) * uL + sqrt( rhoR ) * uR)
413             / ( sqrt( rhoL ) + sqrt( rhoR ) );
414         hRoe = (sqrt( rhoL ) * hL + sqrt( rhoR ) * hR)
415             / ( sqrt( rhoL ) + sqrt( rhoR ) );
416         aRoe = sqrt( (gamma-1.0)*(hRoe - 0.5 * uRoe * uRoe) );
417
418         /* computing the lambda eigenvalues */
419         lbd1 = uRoe - aRoe;
420         lbd2 = uRoe;
421         lbd3 = uRoe + aRoe;
422
423         /* computing the rk eigenvectors */

```

```

423     r1[0] = 1.0;
424     r1[1] = lbd1;
425     r1[2] = hRoe - uRoe*aRoe;
426     r2[0] = 1.0;
427     r2[1] = lbd2;
428     r2[2] = 0.5*uRoe*uRoe;
429     r3[0] = 1.0;
430     r3[1] = lbd3;
431     r3[2] = hRoe + uRoe*aRoe;
432
433     /* computing characteristic properties */
434     //alpha1 = du - (1.0/(rhoRoe * aRoe))*dp;
435     //alpha2 = drho - (1.0/(aRoe * aRoe))*dp;
436     //alpha3 = du + (1.0/(rhoRoe * aRoe))*dp;
437     alpha1 = 0.5 * (dp - rhoRoe * aRoe * du) / (aRoe*aRoe);
438     alpha2 = - (dp - drho * aRoe * aRoe) / (aRoe*aRoe);
439     alpha3 = 0.5 * (dp + rhoRoe * aRoe * du) / (aRoe*aRoe);
440
441     /* computing step in Q */
442     deltaQ[0] = fabs( lbd1 ) * alpha1 * r1[0]
443                 + fabs( lbd2 ) * alpha2 * r2[0]
444                 + fabs( lbd3 ) * alpha3 * r3[0];
445     deltaQ[1] = fabs( lbd1 ) * alpha1 * r1[1]
446                 + fabs( lbd2 ) * alpha2 * r2[1]
447                 + fabs( lbd3 ) * alpha3 * r3[1];
448     deltaQ[2] = fabs( lbd1 ) * alpha1 * r1[2]
449                 + fabs( lbd2 ) * alpha2 * r2[2]
450                 + fabs( lbd3 ) * alpha3 * r3[2];
451
452     /* computing the fluxes */
453     solution->E1p[j] = 0.5 * ( solution->E1[j] + solution->E1[j
454     +1] - deltaQ[0]);
455     solution->E2p[j] = 0.5 * ( solution->E2[j] + solution->E2[j
456     +1] - deltaQ[1]);
457     solution->E3p[j] = 0.5 * ( solution->E3[j] + solution->E3[j
458     +1] - deltaQ[2]);
459
460 }

```

```

458
459 }
460
461 void calcHartenFluxes( results * solution , double lambda ){
462
463     double uL, uR;
464     //double pL, pR;
465     double rhoL, rhoR;
466     double eL, eR;
467     double hL, hR;
468     double uRoe, hRoe, aRoe;
469
470     double lbd1, lbd2, lbd3;
471     double r1[3], r2[3], r3[3];
472     double deltaQ[3];
473     //double dp, du;
474     double drho, de, dqm;
475     double alpha1, alpha2, alpha3;
476     double C1, C2;
477     double g[3][imax], gBar[3][imax];
478     double nu[3], ksi[3], s[3], eps[3], gm[3];
479
480     eps[0] = 0.05;
481     eps[1] = 0.0;
482     eps[2] = 0.0;
483
484     for (int j = 1; j < imax-2; j++){
485         uL = solution->vel[j];
486         uR = solution->vel[j+1];
487         //pL = solution->press[j];
488         //pR = solution->press[j+1];
489         rhoL = solution->rho[j];
490         rhoR = solution->rho[j+1];
491         eL = solution->Q3[j];
492         eR = solution->Q3[j+1];
493         hL = (solution->Q3[j] + solution->press[j])/solution->Q1[j];
494         hR = (solution->Q3[j+1] + solution->press[j+1])/solution->Q1
[j+1];

```

```

495      //du = uR - uL;
496      //dp = pR - pL;
497      drho = rhoR - rhoL;
498      de = eR - eL;
499      dqm = solution->Q2[j+1] - solution->Q2[j];
500
501      /* computing roe averaged variables */
502      uRoe = (sqrt( rhoL ) * uL + sqrt( rhoR ) * uR)
503          / ( sqrt( rhoL ) + sqrt( rhoR ) );
504      hRoe = (sqrt( rhoL ) * hL + sqrt( rhoR ) * hR)
505          / ( sqrt( rhoL ) + sqrt( rhoR ) );
506      aRoe = sqrt( (gamma-1.0)*(hRoe - 0.5 * uRoe * uRoe) );
507
508      /* computing the lambda eigenvalues */
509      lbd1 = uRoe - aRoe;
510      lbd2 = uRoe;
511      lbd3 = uRoe + aRoe;
512
513      /* computing characteristic properties */
514      C1 = ((gamma-1.0)/(aRoe*aRoe))*(de
515          + 0.5 * uRoe * uRoe * drho
516          - uRoe * dqm);
517      C2 = (1.0/aRoe) * (dqm - uRoe * drho);
518
519      alpha1 = 0.5 * ( C1 - C2 );
520      alpha2 = drho - C1;
521      alpha3 = 0.5 * ( C1 + C2 );
522
523      /* computing step in Q */
524
525      nu[0] = lambda * lbd1;
526      nu[1] = lambda * lbd2;
527      nu[2] = lambda * lbd3;
528
529      switch (order){
530          case 1:
531              gBar[0][j] = 0.0;
532              gBar[1][j] = 0.0;

```

```

533         gBar [ 2 ][ j ] = 0.0;
534         break;
535     case 2:
536         ksi [ 0 ] = ksiHarten ( nu [ 0 ] , eps [ 0 ] ) ;
537         ksi [ 1 ] = ksiHarten ( nu [ 1 ] , eps [ 1 ] ) ;
538         ksi [ 2 ] = ksiHarten ( nu [ 2 ] , eps [ 2 ] ) ;
539
540         gBar [ 0 ][ j ] = 0.5 * ( ksi [ 0 ] - nu [ 0 ] * nu [ 0 ] ) * alpha1
541         ;
542         gBar [ 1 ][ j ] = 0.5 * ( ksi [ 1 ] - nu [ 1 ] * nu [ 1 ] ) * alpha2
543         ;
544         gBar [ 2 ][ j ] = 0.5 * ( ksi [ 2 ] - nu [ 2 ] * nu [ 2 ] ) * alpha3
545         ;
546         break;
547     default:
548         gBar [ 0 ][ j ] = 0.0;
549         gBar [ 1 ][ j ] = 0.0;
550         gBar [ 2 ][ j ] = 0.0;
551         break;
552     }
553
554     for ( int j = 1; j < imax-2; j++ ){
555         switch ( order ){
556             case 1:
557                 g [ 0 ][ j ] = 0.0;
558                 g [ 1 ][ j ] = 0.0;
559                 g [ 2 ][ j ] = 0.0;
560                 break;
561             case 2:
562                 switch ( limiter ){
563                     case 0:
564                         s [ 0 ] = copysign( 1.0 , gBar [ 0 ][ j ] );
565                         s [ 1 ] = copysign( 1.0 , gBar [ 1 ][ j ] );
566                         s [ 2 ] = copysign( 1.0 , gBar [ 2 ][ j ] );
567
568                         g [ 0 ][ j ] = gBar [ 0 ][ j ];
569                         g [ 1 ][ j ] = gBar [ 1 ][ j ];

```

```

568                     g [ 2 ] [ j ] = gBar [ 2 ] [ j ];
569                     break;
570             case 1:
571                 s [ 0 ] = copysign( 1.0 , gBar [ 0 ] [ j ] );
572                 s [ 1 ] = copysign( 1.0 , gBar [ 1 ] [ j ] );
573                 s [ 2 ] = copysign( 1.0 , gBar [ 2 ] [ j ] );
574
575                 g [ 0 ] [ j ] = s [ 0 ] * max( 0.0 , min( fabs ( gBar [ 0 ] [ j ] )
576 , s [ 0 ] * gBar [ 0 ] [ j - 1 ] ) );
577                 g [ 1 ] [ j ] = s [ 1 ] * max( 0.0 , min( fabs ( gBar [ 1 ] [ j ] )
578 , s [ 1 ] * gBar [ 1 ] [ j - 1 ] ) );
579                 g [ 2 ] [ j ] = s [ 2 ] * max( 0.0 , min( fabs ( gBar [ 2 ] [ j ] )
580 , s [ 2 ] * gBar [ 2 ] [ j - 1 ] ) );
581
582                     break;
583             default:
584                 s [ 0 ] = copysign( 1.0 , gBar [ 0 ] [ j ] );
585                 s [ 1 ] = copysign( 1.0 , gBar [ 1 ] [ j ] );
586                 s [ 2 ] = copysign( 1.0 , gBar [ 2 ] [ j ] );
587
588                 g [ 0 ] [ j ] = s [ 0 ] * max( 0.0 , min( fabs ( gBar [ 0 ] [ j ] )
589 , s [ 0 ] * gBar [ 0 ] [ j - 1 ] ) );
590                 g [ 1 ] [ j ] = s [ 1 ] * max( 0.0 , min( fabs ( gBar [ 1 ] [ j ] )
591 , s [ 1 ] * gBar [ 1 ] [ j - 1 ] ) );
592                 g [ 2 ] [ j ] = s [ 2 ] * max( 0.0 , min( fabs ( gBar [ 2 ] [ j ] )
593 , s [ 2 ] * gBar [ 2 ] [ j - 1 ] ) );
594
595                     break;
596             }
597         }
598     for ( int j = 1; j < imax - 2; j ++){
599         uL = solution -> vel [ j ];

```

```

600     uR = solution->vel[j+1];
601     //pL = solution->press[j];
602     //pR = solution->press[j+1];
603     rhoL = solution->rho[j];
604     rhoR = solution->rho[j+1];
605     eL = solution->Q3[j];
606     eR = solution->Q3[j+1];
607     hL = (solution->Q3[j] + solution->press[j])/solution->Q1[j];
608     hR = (solution->Q3[j+1] + solution->press[j+1])/solution->Q1
609     [j+1];
610     //du = uR - uL;
611     //dp = pR - pL;
612     drho = rhoR - rhoL;
613     de = eR - eL;
614     dqm = solution->Q2[j+1] - solution->Q2[j];
615
616     /* computing roe averaged variables */
617     //rhoRoe = sqrt( rhoL * rhoR );
618     uRoe = (sqrt( rhoL ) * uL + sqrt( rhoR ) * uR)
619     / ( sqrt( rhoL ) + sqrt( rhoR ) );
620     hRoe = (sqrt( rhoL ) * hL + sqrt( rhoR ) * hR)
621     / ( sqrt( rhoL ) + sqrt( rhoR ) );
622     aRoe = sqrt( (gamma-1.0)*(hRoe - 0.5 * uRoe * uRoe) );
623
624     /* computing the lambda eigenvalues */
625     lbd1 = uRoe - aRoe;
626     lbd2 = uRoe;
627     lbd3 = uRoe + aRoe;
628
629     /* computing the rk eigenvectors */
630     r1[0] = 1.0;
631     r1[1] = lbd1;
632     r1[2] = hRoe - uRoe*aRoe;
633     r2[0] = 1.0;
634     r2[1] = lbd2;
635     r2[2] = 0.5*uRoe*uRoe;
636     r3[0] = 1.0;
637     r3[1] = lbd3;

```

```

637     r3[2] = hRoe + uRoe*aRoe;
638
639     /* computing characteristic properties */
640     C1 = ((gamma-1.0)/(aRoe*aRoe))*(de
641             + 0.5 * uRoe * uRoe * drho
642             - uRoe * dqm);
643     C2 = (1.0/aRoe) * (dqm - uRoe * drho);
644
645     alpha1 = 0.5 * ( C1 - C2 );
646     alpha2 = drho - C1;
647     alpha3 = 0.5 * ( C1 + C2 );
648
649     /* computing gamma k */
650     if (fabs( alpha1 ) > eps[0]){
651         gm[0] = (g[0][j+1] - g[0][j]) / alpha1;
652     }
653     else {
654         gm[0] = 0.0;
655     }
656     if (fabs( alpha2 ) > eps[1]){
657         gm[1] = (g[1][j+1] - g[1][j]) / alpha2;
658     }
659     else {
660         gm[1] = 0.0;
661     }
662     if (fabs( alpha3 ) > eps[2]){
663         gm[2] = (g[2][j+1] - g[2][j]) / alpha3;
664     }
665     else {
666         gm[2] = 0.0;
667     }
668
669     /* computing step in Q */
670     nu[0] = lambda * lbd1;
671     nu[1] = lambda * lbd2;
672     nu[2] = lambda * lbd3;
673
674     ksi[0] = ksiHarten( (nu[0] + gm[0]) , eps[0] );

```

```

675     ksi[1] = ksiHarten( (nu[1] + gm[1]) , eps[1] );
676     ksi[2] = ksiHarten( (nu[2] + gm[2]) , eps[2] );
677
678     deltaQ[0] = r1[0] * (g[0][j] + g[0][j+1] - ksi[0] * alpha1)
679                 + r2[0] * (g[1][j] + g[1][j+1] - ksi[1] * alpha2)
680                 + r3[0] * (g[2][j] + g[2][j+1] - ksi[2] * alpha3);
681     deltaQ[1] = r1[1] * (g[0][j] + g[0][j+1] - ksi[0] * alpha1)
682                 + r2[1] * (g[1][j] + g[1][j+1] - ksi[1] * alpha2)
683                 + r3[1] * (g[2][j] + g[2][j+1] - ksi[2] * alpha3);
684     deltaQ[2] = r1[2] * (g[0][j] + g[0][j+1] - ksi[0] * alpha1)
685                 + r2[2] * (g[1][j] + g[1][j+1] - ksi[1] * alpha2)
686                 + r3[2] * (g[2][j] + g[2][j+1] - ksi[2] * alpha3);
687
688     /* computing the fluxes */
689     solution->E1p[j] = 0.5 * ( solution->E1[j] + solution->E1[j
690 +1] + (1.0/lambda)*deltaQ[0]);
691     solution->E2p[j] = 0.5 * ( solution->E2[j] + solution->E2[j
692 +1] + (1.0/lambda)*deltaQ[1]);
693     solution->E3p[j] = 0.5 * ( solution->E3[j] + solution->E3[j
694 +1] + (1.0/lambda)*deltaQ[2]);
695 }
696
697 void calcJacobian(double q1, double q2, double q3, results *
698 solution){
699     double gm = gamma;
700
701     solution->A[0][0] = 0.0;
702     solution->A[0][1] = 1.0;
703     solution->A[0][2] = 0.0;
704     solution->A[1][0] = (gm - 3.0)*(q2 * q2)/(2.0 * q1 * q1);
705     solution->A[1][1] = (3.0 - gm)*(q2 / q1);
706     solution->A[1][2] = (gm - 1.0);
707     solution->A[2][0] = (gm - 1.0)*(q2*q2*q2)/(q1*q1*q1) - gm*(q3*q2
708 )/(q1*q1);
709     solution->A[2][1] = gm*(q3/q1) - 1.5*(gm - 1.0)*(q2*q2)/(q1*q1);

```

```

708     solution->A[2][2] = gm*(q2/q1);
709 }
710
711 void calcDissipation(results * solution, double lambda){
712     /* case 0 - 2nd order linear artificial dissipation
713      case 1 - 4th order linear artificial dissipation
714      case 2 - Jameson non-linear artificial dissipation
715      default - 2nd order linear artificial dissipation */
716
717     double nu[imax];
718     double k2, k4;
719     double dph, dmh;
720     k2 = 1.0/2.0;
721     k4 = 1.0/20.0;
722
723     for (int j = 0; j < imax; j++){
724         nu[j] = 0.0;
725     }
726
727     switch (dissipModel){
728         case 0:
729             for (int j = 2; j < imax-2; j++){
730                 solution->dissip1[j] = (mu/8.0)*(1.0/lambda)*(
731                     solution->Q1[j+1] - 2.0*solution->Q1[j] + solution->Q1[j-1]);
732                 solution->dissip2[j] = (mu/8.0)*(1.0/lambda)*(
733                     solution->Q2[j+1] - 2.0*solution->Q2[j] + solution->Q2[j-1]);
734                 solution->dissip3[j] = (mu/8.0)*(1.0/lambda)*(
735                     solution->Q3[j+1] - 2.0*solution->Q3[j] + solution->Q3[j-1]);
736             }
737             break;
738         case 1:
739             for (int j = 2; j < imax-2; j++){
740                 solution->dissip1[j] = -(mu/8.0)*(1.0/lambda)*(
741                     solution->Q1[j+2] - 4.0*solution->Q1[j+1] + 6.0*solution->Q1[j]
742                     - 4.0*solution->Q1[j-1] + solution->Q1[j-2]);
743                 solution->dissip2[j] = -(mu/8.0)*(1.0/lambda)*(

```

```

solution->Q2[j+2] - 4.0*solution->Q2[j+1] + 6.0*solution->Q2[j]
- 4.0*solution->Q2[j-1] + solution->Q2[j-2]);
741           solution->dissip3[j] = -(mu/8.0)*(1.0/lambda)*(
solution->Q3[j+2] - 4.0*solution->Q3[j+1] + 6.0*solution->Q3[j]
- 4.0*solution->Q3[j-1] + solution->Q3[j-2]);
742       }
743
744   break;
745
746 case 2:
747
748 for (int j = 2; j < imax-2; j++){
749     nu[j] = fabs(solution->press[j+1] - 2.0*solution->
press[j] + solution->press[j-1]) / (fabs(solution->press[j+1]) +
2.0*fabs(solution->press[j]) + fabs(solution->press[j-1]));
750   }
751
752 for (int j = 2; j < imax-2; j++){
753     double eps2ph = k2*max(nu[j+1],nu[j]);
754     double eps4ph = max(0.0, (k4 - eps2ph));
755     double eps2mh = k2*max(nu[j],nu[j-1]);
756     double eps4mh = max(0.0, (k4 - eps2mh));
757     dph = eps2ph*(solution->Q1[j+1] - solution->Q1[j]) -
eps4ph*(solution->Q1[j+2] - 3.0*solution->Q1[j+1] + 3.0*
solution->Q1[j] - solution->Q1[j-1]);
758     dmh = eps2mh*(solution->Q1[j] - solution->Q1[j-1]) -
eps4mh*(solution->Q1[j+1] - 3.0*solution->Q1[j] + 3.0*solution-
->Q1[j-1] - solution->Q1[j-2]);
759     solution->dissip1[j] = (1.0/lambda)*(dph - dmh);
760   }
761
762 for (int j = 2; j < imax-2; j++){
763     double eps2ph = k2*max(nu[j+1],nu[j]);
764     double eps4ph = max(0.0, (k4 - eps2ph));
765     double eps2mh = k2*max(nu[j],nu[j-1]);
766     double eps4mh = max(0.0, (k4 - eps2mh));
767     dph = eps2ph*(solution->Q2[j+1] - solution->Q2[j]) -
eps4ph*(solution->Q2[j+2] - 3.0*solution->Q2[j+1] + 3.0*

```

```

    solution->Q2[j] - solution->Q2[j-1]);
768         dmh = eps2mh*(solution->Q2[j] - solution->Q2[j-1]) -
769     eps4mh*(solution->Q2[j+1] - 3.0*solution->Q2[j] + 3.0*solution
->Q2[j-1] - solution->Q2[j-2]);
770         solution->dissip2[j] = (1.0/lambda)*(dph - dmh);
771     }
772
773     for (int j = 2; j < imax-2; j++){
774         double eps2ph = k2*max(nu[j+1],nu[j]);
775         double eps4ph = max(0.0, (k4 - eps2ph));
776         double eps2mh = k2*max(nu[j],nu[j-1]);
777         double eps4mh = max(0.0, (k4 - eps2mh));
778         dph = eps2ph*(solution->Q3[j+1] - solution->Q3[j]) -
779     eps4ph*(solution->Q3[j+2] - 3.0*solution->Q3[j+1] + 3.0*
solution->Q3[j] - solution->Q3[j-1]);
780         dmh = eps2mh*(solution->Q3[j] - solution->Q3[j-1]) -
781     eps4mh*(solution->Q3[j+1] - 3.0*solution->Q3[j] + 3.0*solution
->Q3[j-1] - solution->Q3[j-2]);
782         solution->dissip3[j] = (1.0/lambda)*(dph - dmh);
783     }
784
785     break;
786
787     default:
788         for (int j = 2; j < imax-2; j++){
789             solution->dissip1[j] = (mu/8.0)*(1.0/lambda)*(
solution->Q1[j+1] - 2.0*solution->Q1[j] + solution->Q1[j-1]);
790             solution->dissip2[j] = (mu/8.0)*(1.0/lambda)*(
solution->Q2[j+1] - 2.0*solution->Q2[j] + solution->Q2[j-1]);
791             solution->dissip3[j] = (mu/8.0)*(1.0/lambda)*(
solution->Q3[j+1] - 2.0*solution->Q3[j] + solution->Q3[j-1]);
792         }
793         break;
794
795     //free(nu);
796     //free(eps2);
797     //free(eps4);

```

```

796 }
797
798 double max( double a, double b){
799     if (a > b){
800         return a;
801     }
802     else{
803         return b;
804     }
805 }
806
807 double min( double a, double b){
808     if (a < b){
809         return a;
810     }
811     else{
812         return b;
813     }
814 }
815
816 double ksiHarten( double a, double eps ){
817     double ksi;
818
819     if ( fabs( a ) < eps ){
820         ksi = 0.5 * ( ((a*a)/eps) + eps );
821     }
822     else {
823         ksi = fabs( a );
824     }
825
826     return ksi;
827 }
```

Referências

- [1] Harten A. High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, Vol. 49:p. 357–393, 1983.
- [2] S.K. Godunov. A finite difference method for the numerical computation of discontinuous solutions of the equations of fluid dynamics. *Math. Sb.*, Vol. 47:p. 271, 1959.
- [3] Antony Jameson, Wolfgang Schmidt, Eli Turkel, et al. Numerical solution of the euler equations by finite volume methods using runge kutta time stepping schemes. *AIAA paper*, Vol. 1259, 1981.
- [4] John D. Anderson Jr. *Modern Compressible Flow: With Historical Perspective*. McGraw-Hill Education, 1984.