Revista Eletrônica Paulista de Matemática

ISSN 2316-9664
Volume 11, dez. 2017

Métodos numéricos aplicados às equações de Euler: comparação entre *MatLab*, *Octave* e *Fortran*

Numerical methods applied to Euler's equations: comparison between *MatLab*, *Octave* and *Fortran*

Raphael de Oliveira Garcia
Departamento de Ciências Atuariais, Universidade Federal de São Paulo - Osasco - SP
gr.gubim@gmail.com

Graciele Paraguaia Silveira
Departamento de Física, Química e Matemática, Universidade Federal de São Carlos - Sorocaba - SP
gracimat@gmail.com

Resumo

Cada vez mais recursos computacionais são exigidos em modelagem matemática, fazendo com que linguagens de programação interpretadas ganhem mais espaço na computação científica e na matemática aplicada. Este estudo teve como propósito avaliar estruturas das linguagens interpretadas *MatLab* e *Octave*, importantes na escrita de códigos para se obter soluções aproximadas de EDP's. Equações de Euler, que descrevem o escoamento de fluidos ideais foram usadas e suas soluções foram encontradas via métodos numéricos de volumes finitos: Lax-Friedrich, Nessyahu-Tadmor, Lax-Wendroff e Godunov. Em todas as situações, comparou-se o *MatLab* e o *Octave* com a linguagem compilada *Fortran 90*. Os resultados mostraram que o tempo de execução e o custo computacional não dependem apenas do método escolhido para resolver um determinado problema matemático, mas também, da linguagem de programação escolhida para implementar os métodos, atrelada à técnica de programação empregada.
Palavras-chave: Linguagens de programação. Modelagem matemática. Equações de Euler. Métodos numéricos.

Abstract

Increasingly computational resources are required in mathematical modeling, then interpreted programming languages gain more space in scientific computing and applied mathematics.The purpose of this study was to evaluate the structures of the languages interpreted *MatLab* and *Octave*, important in the writing of codes to obtain approximate solutions of EDP's. Euler's equations that describing the flow of ideal fluids were used and their solutions were found via finite volume numerical methods: Lax-Friedrich, Nessyahu-Tadmor, Lax-Wendroff and Godunov. In all situations, we compared *MatLab* and *Octave* with the compiled language *Fortran 90*. The results showed that execution time and computational cost depend not only on the method chosen to solve a given mathematical problem, but also on the programming language chosen to implement the methods, coupled with the programming technique used.
Keywords: Programming languages. Mathematical modeling. Euler equations. Numerical methods.

1 Introdução

Cada vez mais o uso de métodos numéricos, para a obtenção de soluções aproximadas de sistemas de equações diferenciais, vem sendo intensificado devido aos avanços obtidos em análise numérica, computação científica, recursos computacionais, algoritmos e linguagens. Isso possibilita que novos sistemas, fenômenos, propriedades e características sejam descobertos e detalhados por meio de aplicações de métodos numéricos, em diversas áreas do conhecimento, tais como Biomatemática [1], Física-Matemática [2], Mecânica dos fluidos [3], entre outros.

A grande evolução da tecnologia de computadores fez com que nas últimas décadas, as linguagens de programação interpretadas ganhassem cada vez mais espaço na computação científica voltada para a matemática aplicada. Nesta linha, dois dos principais softwares utilizados são: o *MatLab* e o *Octave* [4].

O *MatLab* é uma linguagem comercial que disponibiliza pacotes computacionais voltados para diversas áreas do conhecimento [5]. Com isso, tornou-se uma das linguagens mais utilizadas tanto para fins de pesquisas quanto para fins educacionais. Além disso, *o MatLab* é utilizado na indústria e em empresas.

Em contrapartida, na linha de softwares livres GNU, o *Octave* é uma linguagem que possui características similares ao *MatLab* tanto na questão de escrita quanto em relação a pacotes computacionais voltados para áreas específicas [6].

Desse modo, pesquisadores vêm explorando as potencialidades das duas linguagens na computação científica, como por exemplo, em cálculo numérico [4], operações elementares com funções e matrizes [7], métodos computacionais em álgebra linear [8], métodos numéricos em equações diferenciais [9], entre outros.

Por outro lado, pesquisas são desenvolvidas com o intuito de comparar o desempenho dos pacotes computacionais fornecidos por tais linguagens. Comparações entre os pacotes do *MatLab* e os pacotes de softwares livres *Octave*, *FreeMat* e *Scilab* foram realizadas com a proposta de encontrar uma alternativa de software livre para o *MatLab*, em alguns ambientes educacionais e de pesquisas [10]. Existem ainda comparações voltadas para a ordem de precisão que os pacotes conseguem atingir [8], dependendo do sistema operacional utilizado, e comparações entre as escritas das linguagens.

No âmbito de desenvolvimento de pesquisa, cada vez mais recursos computacionais são exigidos em modelagem matemática. A complexidade da modelagem aumenta consideravelmente a medida que se acrescentam variáveis, equações, refinamento de malha, acoplamento de métodos, acoplamento de sistemas, entre outros. Assim, preocupações associadas ao custo computacional tais como, tempo de execução ou tempo de evolução temporal, consumo de memória e a forma da escrita do código passam a ser primordiais para a viabilidade de execução do código desenvolvido para a modelagem matemática.

Neste ambiente de complexidade, linguagens intepretadas também são utilizadas, como em fenômenos biológicos [1, 11]. Entretanto, em algumas situações, o uso de uma linguagem compilada é exigida para a viabilidade da execução do código, como em astrofísica relativística [2]. Neste contexto, surge a necessidade de comparações entre as linguagens interpretadas e compiladas.

Este estudo teve como propósito avaliar algumas estruturas básicas das linguagens interpretadas *MatLab* e *Octave*, interessantes na escrita de códigos voltados para a obtenção de soluções aproximadas de equações diferenciais parciais (EDP's). Um sistema de EDP's escolhido é for-

mado pelas equações de Euler, que descrevem o escoamento de fluidos ideais. Para obter as soluções aproximadas, foram adotados métodos numéricos de volumes finitos clássicos, a saber, Lax-Friedrich, Nessyahu-Tadmor, Lax-Wendroff e Godunov. Em todas as situações, comparou-se o *MatLab* e o *Octave* com a linguagem compilada *Fortran 90*.

A escolha do sistema mencionado se deve ao caráter não linear de suas equações. Uma implementação computacional não criteriosa pode resultar em oscilações espúrias e/ou dissipação excessiva, que prejudicam o resultado final das simulações. A complexidade computacional de tal sistema, em uma e em duas dimensões, fornece condições suficientes para explorar o desempenho de estruturas básicas das linguagens em códigos aplicados em modelagem matemática.

2 Recursos computacionais

Este trabalho foi totalmente executado em um computador comum com as seguintes especificações e configurações.

Informações sobre hardwares:

Memória RAM: 6.0 Gbytes, Processador Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz e bogomips: 5786.73 (informação obtida pelo programa *cpuinfo*).

Informações sobre o sistema operacional:

UBUNTU 13.04 GNU/LINUX, kernel 3.8.0-30, 64 bits.

Informações sobre softwares:

Para gerar os executáveis em *Fortran 90* foi utilizado o compilador *gfortran* versão 4.7.3, com sua configuração padrão. O *MatLab* utilizado foi R2012a (7.14.0.739) 64-bit e todos os programas foram executados no terminal, para isto, carregou-se o MatLab no terminal via comando:

matlab -nosplash -nojvm.

A versão do *Octave* utilizada foi a 3.6.4 (2013) 64-bit e executada via terminal.

O *MatLab* possui internamente um compilador chamado *Just-In-Time* (*JIT*), que atua para diminuir o tempo de execução. Como padrão, o *JIT* vem ativado e seu uso é opcional, podendo ser desativado usando-se o comando *feature accel off* e ativado pelo comando *feature accel on*. Na Seção 3 mostrou-se a importância do *JIT* para o *MatLab*. Mais detalhes sobre esse complilador encontram-se no guia online do [5], na parte de *Perform Acceleration*.

No guia do *Octave* [6], o usuário é alertado de que um compilador semelhante ao *Just-In-Time* do *MatLab* está em desenvolvimento e que ainda não faz parte da opção padrão. Por essa razão, os testes realizados com o *Octave* não possuem tal recurso, já os testes com o *MatLab* foram executados em duas situações: com o *JIT* e sem o *JIT*, verificando-se que de fato o *JIT* atuou como um acelerador de performance como apresentado no guia do *MatLab*.

3 Comparações entre as linguagens

Antes de realizar as comparações entre as linguagens utilizadas na obtenção de soluções aproximadas, através de métodos numéricos para EDP's aplicadas em dinâmica de fluidos, nesta seção foram avaliadas estruturas fundamentais como laços de repetições, tomadas de decisões e funções, em cada linguagem. Com o intuito de se obter o tempo de *CPU* gasto por cada linguagem, fez-se uso de suas funções internas: $CPU_TIME()$ para o *Fortran* e $cputime$ para o *MatLab* e *Octave*.

A primeira estrutura testada foi o laço de repetição. Uma recomendação para se ter um melhor desempenho em tempo de execução em linguagens interpretadas é evitar, sempre que possível, laços de repetições e utilizar uma versão vetorizada [5]. Entretanto, em diversos problemas de evolução temporal de EDP's, discretizados por meio de esquemas numéricos explícitos, torna-se inevitável o uso do laço de repetição e, dependendo do problema, há a necessidade de grande quantidade de iterações desses laços (como por exemplo, problemas de mecânica de fluidos, previsão numérica oceânica, astrofísica, entre outros). Desse modo, testou-se um laço de repetição variando a quantidade de iterações à serem realizadas.

Teste 1: Laços de repetições

Neste primeiro teste, computou-se quanto tempo demora para os programas feitos em *Fortran*, *MatLab* e *Octave* executarem um laço de repetição vazio (sem código dentro dele). Os trechos de códigos em cada linguagem estão expostos a seguir:

Códigos:

Fortran	**MatLab/Octave**
⋮	⋮
CALL CPU_TIME(tempo1)	tempo = cputime;
DO i=1,Nt,1	for i=1:Nt
END DO	end
CALL CPU_TIME(tempo2)	CPUt = cputime - tempo;
CPUt = tempo2 - tempo1	⋮
⋮	

Na Tabela 1 estão os tempos gastos para realizar um laço de repetição a partir do número 1 até o número final Nt. Para cada Nt foram realizadas dez simulações para se obter uma média de tempo gasto por cada linguagem. Como o *MatLab* tem a opção de executar com ou sem o seu compilador interno *Just-In-Time* (*JIT*), na Tabela 1 constam as duas execuções, para verificar sua funcionalidade e importância.

Neste teste, o *Octave* demorou mais para executar tais tarefas. Tem-se a mesma ordem de magnitude no tempo de execução para o *Fortran* e o *MatLab* com *JIT*, enquanto que o *Octave* está em uma ordem de magnitude maior. Dividindo os valores do *MatLab* pelos valores obtidos pelo *Fortran*, segue que o *MatLab* foi $1,38$ vezes mais lento (isto é, 38%) que o *Fortran*. Comparando o *Octave* com o *MatLab*, tem-se que o primeiro demora em média $19,58$ vezes para realizar laços

Tabela 1: Tempo gasto em média para realizar laços de repetições.

Número Final	Fortran 90	MatLab		Octave
		com JIT	sem JIT	
2^{22}	0,0072	0,0100	0,1920	0,2144
2^{23}	0,0140	0,0210	0,3800	0,4264
2^{24}	0,0292	0,0420	0,7580	0,8532
2^{25}	0,0588	0,0840	1,5180	1,7104
2^{26}	0,1160	0,1509	3,0390	3,4148
2^{27}	0,2340	0,3350	6,0820	6,8324
2^{28}	0,4656	0,6650	24,3100	13,8728
2^{29}	0,9320	1,3320	48,6480	27,8492
2^{30}	1,8628	2,6850	97,2790	55,7052

de repetições. Entretanto, ao forçar que o *MatLab* não execute o seu compilador interno, o mesmo inicia com um comportamento semelhante ao *Octave*. O tempo de execução fica ligeiramente melhor do que o *Octave* nas execuções iniciais, mas nas execuções mais longas o *Octave* fica quase duas vezes mais rápido que o *MatLab*.

Ao remover a opção do uso do *JIT* no *MatLab*, o tempo de execução aumenta drasticamente. De início, os tempos de execução do *MatLab* e do *Octave* são semelhantes, mas à medida que o tamanho do laço de repetição aumenta, o *Octave* reduz sua diferença para o *MatLab* a ponto de, no final, o *MatLab* ficar quase duas vezes mais lento que o *Octave*. Isso revela a fundamental importância do compilador interno *Just-in-Time* para o *MatLab*.

Teste 2: Laços de repetições com tomadas de decisões

Em vários códigos, o método numérico escolhido exige tomadas de decisões dentro de um laço de repetição. Se o método numérico utilizado requer algum critério de parada ou algum refinamento e/ou adaptação de malha, a combinação entre laços de repetições e tomadas de decisões são inevitáveis. Sendo assim, neste segundo teste foi acrescentada uma tomada de decisão dentro do laço de repetição do *Teste 1* para a realização de comparações entre os tempos de execução. A seguir encontram-se os trechos relevantes de cada código:

Códigos:

Fortran

```
⋮
CALL CPU_TIME(tempo1)
DO i=1,Nt,1
      IF(i<j) THEN
      END IF
END DO
CALL CPU_TIME(tempo2)
CPUt = tempo2 - tempo1
⋮
```

MatLab/Octave

```
⋮
tempo = cputime;
for i=1:Nt
     if(i<j)
     end
end
CPUt = cputime - tempo;
⋮
```

A Tabela 2 expõe o grande acréscimo no tempo de execução do *Octave* ao adicionar uma tomada de decisão. Enquanto que o *MatLab* teve um pequeno acréscimo, o *Fortran* manteve praticamente o mesmo gasto. Efetuando a razão entre os resultados do *MatLab* e do *Fortran*, em média, o Fortran efetuou $2,2106$ vezes mais rápido o laço de repetição com tomadas de decisões do que o *MatLab*, entretanto, o tempo de CPU do *MatLab* manteve a mesma ordem de grandeza.

Neste teste o *Octave* ficou $247,71$ vezes mais lento que o *MatLab*, deixando o tempo de CPU em duas ordens de grandezas maior que a ordem de grandeza do *MatLab*. Assim, o tempo de CPU sofreu uma grande elevação em comparação aos tempos das outras linguagens.

Tabela 2: Tempo gasto em média para realizar laços de repetições com tomadas de decisões.

Número Final	Fortran 90	MatLab		Octave
		com JIT	sem JIT	
2^{22}	0,0073	0,0160	2,0100	4,0028
2^{23}	0,0158	0,0330	3,9986	8,0060
2^{24}	0,0293	0,0670	8,0243	16,2508
2^{25}	0,0595	0,1300	16,0886	32,6436
2^{26}	0,1188	0,2630	32,0100	65,2800
2^{27}	0,2361	0,5240	63,8800	130,7148
2^{28}	0,4712	1,0510	127,4814	261,1212
2^{29}	0,9391	2,0990	254,5043	522,4264
2^{30}	1,8755	4,2100	514,0014	1.043,9744

Eliminando a ação do compilador interno do *MatLab*, a execução do código em *MatLab* passa a ser, em média, duas vezes mais rápido do que o *Octave*. Isso ressalta a importância do *JIT* para o *MatLab*.

Teste 3: Construção de matriz

Uma ferramenta imprescindível na computação científica voltada para métodos numéricos são os *array's* para armazenar valores numéricos.

Matrizes vinculadas a sistemas lineares ou sistemas não lineares são comuns em vários problemas aplicados. Em alguns casos da área de Física Estatística, Sistemas Biológicos, Ciências dos Materiais, entre outros, a matriz do sistema linear ou não linear chega na ordem de bilhões. Isso por si só, faz com que a construção dessas matrizes seja um trabalho delicado e complexo.

Nesse contexto, o terceiro teste tem como objetivo verificar o tempo de execução de uma matriz. Assim, um algoritmo para construir e armazenar uma matriz A, de ordem $n \times n$, foi definido, onde A é da forma:

$$A = [a_{ij}]_{n\times n}, \ com \ a_{ij} = i + j \ \ e \ \ i, j = 1, \ldots, n \, .$$

A seguir, são apresentados trechos dos códigos em *Fortran* e em *MatLab/Octave*. Neste exemplo, utilizou-se dois laços de repetições para construir a matriz e a definição da matriz fica diretamente dentro desses laços. Além disso, o código para o *MatLab/Octave* possui duas versões e a única diferença entre elas foi que, no código 2, a matriz A é criada com seus elementos iguais a zero, antes da execução dos laços de repetições da matriz. Assim, no código 1, não há uma matriz A pré-definida. A Tabela 3 mostra os resultados obtidos na execução de cada código.

Códigos:

Fortran

```
⋮
CALL CPU_TIME(tempo1)
DO i=1,m,1
    DO j=1,n,1
        A(i,j)=i+j
    END DO
END DO
CALL CPU_TIME(tempo2)
CPUt = tempo2 - tempo1
⋮
```

MatLab/Octave(1)
Código 1

```
⋮
tempo = cputime;
for i=1:m
    for j=1:n
        A(i,j)=i+j;
    end
end
CPUt = cputime - tempo;
⋮
```

MatLab/Octave(2)
Código 2

```
⋮
tempo = cputime;
A=zeros(m,n);
for i=1:m
    for j=1:n
        A(i,j)=i+j;
    end
end
CPUt = cputime - tempo;
⋮
```

Os códigos foram executados dez vezes por meio de laços de repetições encaixados. Os tempos gastos em média para construir a matriz A, para alguns n, estão apresentados na Tabela 3.

Tabela 3: Tempo gasto em média para construir uma matriz.

Ordem	Fortran 90	MatLab(1)	Octave(1)	Octave(2)	MatLab(2)	
					com JIT	sem JIT
$2^6 \times 2^6$	$< p$	$< p$	0,0132	0,024008	$< p$	0,0110
$2^7 \times 2^7$	$< p$	$< p$	0,0476	0,068813	$< p$	0,0250
$2^8 \times 2^8$	$< p$	0,0030	0,1872	0,24430	$< p$	0,086999
$2^9 \times 2^9$	0,0080	0,0160	0,7536	0,94830	0,0080	0,307999
$2^{10} \times 2^{10}$	0,0164	0,0960	3,6118	3,7574	0,0200	1,20199
$2^{11} \times 2^{11}$	0,0640	0,6900	15,7210	15,306	0,0690	4,82899
$2^{12} \times 2^{12}$	0,2400	5,7660	99,0700	61,4021	0,3010	18,8800
$2^{13} \times 2^{13}$	1,0480	48,3821	691,5600	257,1927	1,1260	78,14699
$2^{14} \times 2^{14}$	4,3480	426,2614	5.068,0100	1.419,2614	4,5440	329,86
$2^{15} \times 2^{15}$	$39,50^{(*)}$	-	-	-	-	

Note que na Tabela 3 foram utilizados os símbolos $< p$ e $-$. O primeiro símbolo significa que o tempo de execução foi aproximadamente zero, isto é, o tempo de CPU ficou abaixo da precisão fixada nas rotinas: $CPU_TIME()$ do *Fortran* e $cputime$ do *MatLab/Octave*. Já o segundo símbolo, representa que a execução do código foi interrompida devido a uma longa demora no tempo de CPU.

Ainda na Tabela 3, no último exemplo, o resultado obtido pelo *Fortran* está com asterisco, pois neste caso, o computador precisou acionar a memória *swap*. As dez execuções foram realizadas através de um laço de repetição externo aos laços de repetições utilizados na construção da matriz A.

Neste exemplo, ao dividir os resultados do *MatLab(1)* pelo *Fortran* e os resultados do *Octave(1)* pelo *MatLab(1)*, tem-se um comportamento diferente dos testes anteriores. A medida que aumenta a ordem da matriz A, o *MatLab(1)* demora cada vez mais para executar a mesma tarefa que o *Fortran*, veja a Tabela 4. No caso do *Octave(1)* com o *MatLab(1)*, vemos que a

medida que aumenta a ordem da matriz A, o tempo gasto pelo *Octave(1)* aproxima-se cada vez mais do tempo gasto pelo *MatLab(1)*, conforme mostra a Tabela 4.

Tabela 4: Razões entre *Fortran* e *MatLab*, e entre *Octave* e *MatLab*. Dados da Tabela 3.

Ordem	MatLab(1)/Fortran	Octave(1)/MatLab(1)	MatLab(2)/Fortran
$2^8 \times 2^8$	-	62,400	-
$2^9 \times 2^9$	2,000	47,100	1,000
$2^{10} \times 2^{10}$	4,00	22,784	1,220
$2^{11} \times 2^{11}$	10,781	17,182	1,078
$2^{12} \times 2^{12}$	24,025	17,182	1,254
$2^{13} \times 2^{13}$	46,166	14,294	1,074
$2^{14} \times 2^{14}$	98,036	11,889	1,045

Ao considerar o código 2, cuja matriz A está definida com zeros antes dos laços de repetições, o tempo de execução muda significativamente. O *MatLab(2)* com o *JIT* passou a ter um desempenho similar ao do *Fortran* até o penúltimo exemplo. O *Octave(2)* também adquiriu importantes melhoras. Isso mostra que não apenas o *JIT* é fundamental para o desempenho do *MatLab*, mas também a forma como o código é escrito (isso também vale para o *Octave*). Assim pode-se concluir que, mesmo que o *MatLab/Octave* não dependa da declaração de variáveis, como as linguagens compiladas, defini-las antes de laços de repetições foi crucial para o tempo de execução do código. Esse procedimento faz um papel similar ao de declaração de variáveis, isto é, define um lugar na memória para a matriz, antes de efetuar as operações matemáticas.

Este exemplo também diferencia-se dos testes anteriores porque a matriz A precisa ser armazenada na memória do computador. Como o *MatLab* e o *Octave* são linguagens interpretadas, cada linha do código é analisada antes de ser executada. Assim, ao passar pela primeira vez na linha que define a matriz, tais linguagens gastam tempo de CPU para reservar um espaço na memória para o *array*, onde a matriz A será armazenada. Logo, a primeira execução da construção da matriz A tende a ser mais lenta do que as demais, pois nas outras construções da matriz A, durante a mesma execução do programa, já haverá espaço reservado na memória por se tratar apenas de uma recolocação de números nas posições da memória reservada. De fato, isto é observado a partir dos tempos gastos para construir a matriz A para uma ordem fixa, conforme a Tabela 5.

Os dados da Tabela 5 revelam que o *MatLab* e o *Octave* demoram muito mais para executar a primeira construção da matriz A do que as demais execuções. Excluindo a primeira execução, calculou-se a média entre a segunda e a décima execuções. Esse resultado representa o tempo que as linguagens *MatLab* e *Octave* levaram para construir a matriz A, sem os tempos extras de procedimentos internos, voltados para reservar um lugar na memória para a matriz A. Pode-se determinar a diferença relativa percentual, isto é, a porcentagem de tempo que as linguagens demoraram para armazenar a matriz A na memória, segundo a seguinte expressão:

$$D(\%) = \frac{(\text{tempo primeira execução}) - \text{Média}(2-10)}{\text{tempo primeira execução}} \times 100\% \ .$$

Assim para o exemplo da Tabela 5, obtemos que o *MatLab* destinou em média

$$D(\%) = \frac{0,4670 - 0,0544}{0,4670} \times 100\% = 88,35\%$$

Tabela 5: Detalhes de um dos casos: matriz de ordem $2^{10} \times 2^{10}$.

Execuções	Fortran 90	MatLab(1)	Octave(1)	Octave(2)	MatLab(2)	
					com JIT	sem JIT
1	0,0200	0,4670	4,4200	3,8877	0,0400	1,3300
2	0,0200	0,0500	2,9480	3,7892	0,0300	1,2100
3	0,0160	0,0600	3,2120	3,7852	0,0300	1,2100
4	0,0160	0,0500	3,6600	3,7852	0,0300	1,2100
5	0,0120	0,0500	3,6600	3,7997	0,0200	1,2000
6	0,0160	0,0600	3,6640	3,7891	0,0200	1,2000
7	0,0240	0,0600	3,6600	3,8341	0,0200	1,2100
8	0,0160	0,0500	3,6520	3,7879	0,0200	1,2000
9	0,0160	0,0500	3,6560	3,7883	0,0200	1,2000
10	0,0120	0,0600	3,6560	3,7845	0,0100	1,2000
Média Total	0,0164	0,0960	3,6188	3,8031	0,0240	1,2170
Média (2-10)	0,0160	0,0544	3,5298	3,7937	0,0220	1,2044

do tempo gasto na primeira execução para armazenar/alocar um espaço na memória para a matriz A. Já o *Octave* destinou

$$D(\%) = \frac{4,42 - 3,5298}{4,42} \times 100\% = 20,14\% \, .$$

No *Fortran*, ao realizar tal procedimento encontra-se

$$D(\%) = \frac{0,02 - 0,016}{0,02} \times 100\% = 20\% \, .$$

Em relação apenas à primeira execução, tem-se o *Octave* mais próximo dos resultados do *MatLab*, enquanto o *MatLab* fica muito mais distante dos resultados do *Fortran*. Na Tabela 6 considera-se somente o tempo de CPU gasto na primeira simulação e na Tabela 7 foram expostos os resultados das razões entre os tempos de execução dos métodos, usando os dados da Tabela 6.

Tabela 6: Tempo gasto em uma primeira execução para construir uma matriz.

Ordem	Fortran 90	MatLab	Octave
$2^6 \times 2^6$	-	-	0,0200
$2^7 \times 2^7$	-	-	0,0560
$2^8 \times 2^8$	-	0,0200	0,2080
$2^9 \times 2^9$	0,0082	0,0400	0,8240
$2^{10} \times 2^{10}$	0,0200	0,4670	4,0040
$2^{11} \times 2^{11}$	0,0653	5,7200	24,9240
$2^{12} \times 2^{12}$	0,2453	51,7600	161,8560
$2^{13} \times 2^{13}$	1,0700	430,0900	1.095,0600
$2^{14} \times 2^{14}$	4,4610	3.491,8400	7.906,3320

Os dados da Tabela 7 demonstram que em uma execução com a matriz na ordem de $2^9 \times 2^9$, o *MatLab* e o *Fortran* estão na mesma ordem de grandeza. Por outro lado, na ordem de

$2^{14} \times 2^{14}$, o *MatLab* é quase mil vezes mais lento que o *Fortran*, enquanto o *Octave* passa a ser aproximadamente duas vezes mais lento que o *MatLab*.

Tabela 7: Razões entre Fortran e MatLab, e entre Octave e MatLab. Dados da Tabela 6

Ordem	MatLab/Fortran	Octave/MatLab
$2^8 \times 2^8$	-	62,400
$2^9 \times 2^9$	5,000	10,400
$2^{10} \times 2^{10}$	23,350	8,5739
$2^{11} \times 2^{11}$	89,375	4,3573
$2^{12} \times 2^{12}$	215,670	3,1270
$2^{13} \times 2^{13}$	410,390	2,5461
$2^{14} \times 2^{14}$	803,090	2,2642

Teste 4: Acesso a uma função externa (construção de matriz)

Esse teste realiza a mesma construção e armazenamento da matriz definida no Teste 3, com o diferencial de que agora a matriz a ser construída está definida em uma função externa (sub-rotina) ao programa principal. Assim, os códigos escritos no *MatLab/Octave* têm que localizar o arquivo onde encontra-se a função externa e armazená-lo na memória. Consequentemente, o tempo de execução para construir a matriz será aumentado.

Neste caso, também foram testados dois códigos para o *MatLab/Octave*, diferenciando-se apenas na definição da matriz A. O *MatLab(2)/Octave(2)* refere-se ao código cuja matriz A está definida antes dos laços de repetições (semelhante ao Teste 3).

Códigos Principais:

Fortran

```
⋮
CALL CPU_TIME(tempo1)
DO i=1,m,1
    DO j=1,n,1
        A(i,j)=construction(i,j)
    END DO
END DO
CALL CPU_TIME(tempo2)
CPUt = tempo2 - tempo1
⋮
```

MatLab/Octave

```
⋮
tempo = cputime;
for i=1:m
    for j=1:n
        [A(i,j)]=construction(i,j);
    end
end
CPUt = cputime - tempo;
⋮
```

Sub-rotina:

Fortran

```
REAL FUNCTION construction(i,j)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: i,j
    construction = i + j
END FUNCTION construction
```

MatLab/Octave

```
function [aij]=construction(i,j)
    aij = i + j;
end
```

Na Tabela 8 encontram-se os tempos médios de execução de cada linguagem, para construir e armazenar uma determinada matriz definida por uma função externa. O intuito foi incluir o tempo que cada código demora para acessar uma rotina externa ao programa principal, em vários cenários. Para cada exemplo foram executadas dez simulações através de um laço de repetição. Assim, tem-se uma diferença entre a primeira execução e as demais, conforme as Tabelas 11 e 12.

Tabela 8: Tempo gasto em média para construir uma matriz através de uma função externa.

Ordem	Fortran 90	MatLab(1)	Octave(1)	Octave(2)	MatLab(2)
$2^4 \times 2^4$	<p	<p	0,0048	0,010443	<p
$2^5 \times 2^5$	<p	<p	0,0168	0,026993	0,0010
$2^6 \times 2^6$	<p	0,0020	0,0616	0,081597	0,0020
$2^7 \times 2^7$	0,0004	0,0030	0,2412	0,295882	0,0080
$2^8 \times 2^8$	0,0016	0,0130	0,9752	1,14167	0,0210
$2^9 \times 2^9$	0,0044	0,0510	4,4300	4,56201	0,0570
$2^{10} \times 2^{10}$	0,0180	0,2550	18,7480	18,17388	0,19900
$2^{11} \times 2^{11}$	0,0656	1,5440	77,1360	77,12934	0,7750
$2^{12} \times 2^{12}$	0,2588	9,5540	318,2978	315,9814	3,0530
$2^{13} \times 2^{13}$	1,0512	61,7710	1.341,2000	1.230,58962	12,3100
$2^{14} \times 2^{14}$	4,7610	-	-		48,7950

Ainda na Tabela 8, verifica-se que o código referente ao *MatLab(2)* de início é mais lento que o código do *MatLab(1)* e a medida que aumenta a ordem da matriz, essa situação se inverte. Assim, definir antes a matriz é mais eficiente apenas para matrizes de ordens elevadas. O mesmo fato ocorreu ao comparar o *Octave(1)* com o *Octave(2)*.

Efetuando a razão entre os valores da Tabela 9, a diferença entre o *MatLab(1)* e o *Fortran* aumenta, ao passo que a diferença entre o *MatLab(1)* e o *Octave(1)* diminui, com a ordem da matriz aumentada, quando comparados os tempos de execução em média. O mesmo fenômeno é observado quando analisados os tempos de execução da primeira simulação, ver Tabela 10.

Tabela 9: Razões entre *Fortran* e *MatLab*, e entre *Octave* e *MatLab*.

Ordem	MatLab(1)/Fortran	Octave(1)/MatLab(1)
$2^6 \times 2^6$	-	30,800
$2^7 \times 2^7$	7,5000	80,400
$2^8 \times 2^8$	8,1250	75,015
$2^9 \times 2^9$	11,591	86,863
$2^{10} \times 2^{10}$	18,214	73,522
$2^{11} \times 2^{11}$	23,537	49,959
$2^{12} \times 2^{12}$	36,917	33,316
$2^{13} \times 2^{13}$	58,761	21,712

Os tempos gastos em uma primeira execução, para cada exemplo da Tabela 8, estão expostos na Tabela 11. Observa-se em um dos exemplos que se a ordem da matriz é $2^{11} \times 2^{11}$ (Tabela 12), então o *MatLab* destinou em média

Tabela 10: Razões entre *Fortran* e *MatLab*, e entre *Octave* e *MatLab*. Dados da Tabela 11.

Ordem	MatLab(1)/Fortran	Octave(1)/MatLab(1)
$2^6 \times 2^6$	-	7,2000
$2^7 \times 2^7$	10	3,2000
$2^8 \times 2^8$	10	50,800
$2^9 \times 2^9$	7,5	21,052
$2^{12} \times 2^{10}$	26,625	52,786
$2^{11} \times 2^{11}$	23,333	11,527
$2^{12} \times 2^{12}$	93,625	5,1840
$2^{13} \times 2^{13}$	435,51	2,7895

$$D(\%) = \frac{7,49 - 0,0639}{7,49} \times 100\% = 88,21\%$$

do tempo gasto na primeira execução para armazenar/alocar um espaço na memória para a matriz A e para a sub-rotina, mantendo uma porcentagem semelhante ao Teste 3.

Já o *Octave* destinou

$$D(\%) = \frac{86,34 - 76,11}{86,34} \times 100\% = 11,85\% \,,$$

o que mostra uma diminuição na porcentagem destinada para a primeira execução, ao comparar com o Teste 3 onde obteve-se $20,14\%$.

No *Fortran* encontrou-se

$$D(\%) = \frac{0,08 - 0,064}{0,08} \times 100\% = 20\% \,,$$

mantendo o mesmo resultado do Teste 3.

Tabela 11: Tempo gasto em uma primeira execução, para construir uma matriz através de uma função externa.

Ordem	Fortran 90	MatLab(1)	Octave(1)
$2^4 \times 2^4$	<p	<p	0,0120
$2^5 \times 2^5$	<p	<p	0,0320
$2^6 \times 2^6$	<p	0,0010	0,0720
$2^7 \times 2^7$	0,0008	0,0080	0,2560
$2^8 \times 2^8$	0,0040	0,0300	1,0160
$2^9 \times 2^9$	0,0080	0,2130	4,4840
$2^{10} \times 2^{10}$	0,0360	0,8400	44,34
$2^{11} \times 2^{11}$	0,0800	7,4900	86,34
$2^{12} \times 2^{12}$	0,2800	61,4000	318,2978
$2^{13} \times 2^{13}$	1,1040	480,8000	1.341,200
$2^{14} \times 2^{14}$	4,5240	-	-

Tabela 12: Detalhes de um dos casos: matriz de ordem $2^{11} \times 2^{11}$.

Execuções	Fortran 90	MatLab	Octave
1	0,0800	7,4900	86,3400
2	0,0634	0,7699	76,2040
3	0,0640	0,7699	76,2640
4	0,0640	0,7600	76,2799
5	0,0640	0,9200	76,1040
6	0,0640	0,9399	76,1080
7	0,0640	0,9500	76,0519
8	0,0640	0,9500	75,9480
9	0,0640	0,9499	76,1479
10	0,0640	0,9400	75,9120
Média Total	0,0656	1,5440	771360
Média (2-10)	0,0639	0,8833	76,1133

A partir das simulações realizadas, pode-se afirmar que os testes evidenciam a diferença entre as linguagens compiladas e interpretadas. Nas operações básicas de laço de repetição, tomada de decisão, alocação de *array* e acesso a função externa, o *Fortran* leva grande vantagem em relação ao *MatLab* e ao *Octave*, principalmente quando necessita-se executar diversas vezes tais operações. Em geral, também observou-se grandes diferenças entre o *MatLab* e o *Octave*. Em alguns cenários, o *MatLab* assemelha-se ao *Fortran*, como no caso do Teste 3, já em outras situações, como no Teste 4, tem-se o *Matlab* mais lento que o *Fortran*, um pouco mais de dez vezes.

Em códigos longos, criar subrotinas externas ao programa principal, muitas vezes faz com que o código fique mais organizado, legível e útil para outros futuros códigos, tornando tal prática indispensável. Sendo assim, uma alternativa para diminuir essa diferença é fazer com que o *MatLab* reconheça as funções ou subrotinas externas, antes de executar o programa principal. Tal conduta é semelhante ao que uma linguaguem compilada faz, uma vez que nessas linguagens, as subrotinas precisam ser declaradas e compiladas antes da criação do arquivo executável.

O próximo passo foi fazer com que tais diferenças fossem combinadas em códigos de métodos numéricos, para equações diferenciais parciais que descrevem o escoamento de um fluido ideal, através das equações de Euler.

4 Simulações em hidrodinâmica

Nesta seção serão apresentadas comparações entre as linguagens *Fortran*, *MatLab* e *Octave* ao implementar métodos numéricos aplicados às equações de Euler unidimensionais e bidimensionais. Tais equações são de fundamental importância para as ciências e engenharias. Sua implementação representa bem os desafios de se obter soluções aproximadas para sistemas de EDP's não lineares, tornando-se um bom teste para as linguagens consideradas.

Desse modo, os códigos desenvolvidos para obter soluções aproximadas para as equações de Euler estão inseridos em um cenário que combina as estruturas testadas na Seção 3. Na Subseção 4.1 as comparações são realizadas em um problema unidimensional das equações de Euler e na

Subseção 4.2, em um problema bidimensional.

4.1 Equações de Euler unidimensionais

O escoamento de um fluido ideal é descrito pelas equações de Euler [3]

$$\frac{\partial}{\partial t}\rho + \frac{\partial}{\partial x}(\rho v) = 0 \qquad \text{(equação da continuidade)} \qquad (1)$$

$$\frac{\partial}{\partial t}\rho v + \frac{\partial}{\partial x}\left(\rho v^2 + p\right) = 0 \qquad \text{(equação do movimento)} \qquad (2)$$

$$\frac{\partial}{\partial t}E + \frac{\partial}{\partial x}(v(E+p)) = 0 \qquad \text{(equação da energia)} \qquad (3)$$

em que

$\rho = \rho(x,t)$	(densidade do fluido em escoamento)
$v = v(x,t)$	(velocidade do fluido)
$p = p(x,t)$	(distribuição de pressão no fluido)
$E = E(x,t)$	(energia total do fluido)
$\varepsilon = \varepsilon(x,t)$	(energia interna do fluido)
γ	(calor específico do fluido)

com

$$E = \frac{1}{2}\left(\rho v^2 + \varepsilon\right) \quad \text{e} \quad \varepsilon = \frac{p}{(\gamma-1)\rho}\,.$$

Tal sistema de EDP's é hiperbólico, o que torna possível o surgimento de descontinuidades na solução, mesmo a partir de dados iniciais suaves [12]. Isso faz com que este sistema seja um bom exemplo teste para métodos numéricos e para novos algoritmos desenvolvidos.

Com o intuito de obter soluções aproximadas das equações de Euler, implementou-se os métodos de volumes finitos Lax-Friedrich (L-F), Lax-Wendroff (L-W), Godunov com solucionador de problemas de Riemann de Hartern-Lax-Leer (Godunov-HLL) [3] e Nessyahu-Tadmor (N-T) [13]. Os métodos L-F, L-W e N-T são esquemas centrais que independem de solucionadores de problemas de Riemann fundamentais, pois nessa formulação as descontinuidades ficam centradas em cada célula média (ou volume finito), ao contrário dos métodos de Godunov que dependem de problemas de Riemann para determinar os autovalores associados às velocidades de propagação das descontinuidades, definidas em cada extremidade do volume finito.

Todos os esquemas foram escritos em *Fortran 90* e em *MatLab/Octave*. Os programas foram executados com as condições iniciais conhecidas como *problema teste de Sod*, confome os seguintes valores:

Condições Iniciais $(\gamma = 1,4)$

$$\rho(x,0) = \begin{cases} 1,0 & \text{se } x < 0 \\ 0,125 & \text{se } x > 0 \end{cases}, \quad v(x,0) = \begin{cases} 0,0 & \text{se } x < 0 \\ 0,0 & \text{se } x > 0 \end{cases}, \quad p(x,0) = \begin{cases} 1,0 & \text{se } x < 0 \\ 0,1 & \text{se } x > 0 \end{cases}.$$

Utilizando o domínio $x \in [-2,2]$ e as quantidades de subintervalos de cada simulação conforme mostra a Tabela 13, para atingir o tempo final de $0,6$, comparações foram feitas entre os métodos numéricos escritos em cada linguagem, para algumas distintas malhas, obtendo-se

os resultados expostos na tabela mencionada. Os gráficos da Figura 1 apresentam as soluções aproximadas de cada método, com a malha de 64 subintervalos. Já os gráficos da Figura 2, as soluções para a malha de 256 subintervalos (independente da linguagem escolhida).

Tabela 13: Tempo de execução para obter soluções aproximadas.

Schemes	subintervals	iterations	Fortran 90	MatLab	Octave
Lax-Friedrich	64	96	6e-4	0,050	0,560
	128	192	0,002	0,181	2,188
	256	384	0,010	0,816	9,835
	512	768	0,024	2,357	41,672
	1.024	1.536	0,070	6,845	167,680
Nessyahu-Tadmor	64	96	0,005	0,462	1,984
	128	192	0,0979	0,924	8,904
	256	384	0,036	2,991	37,176
	512	768	0,102	10,049	148,88
	1.024	1.536	0,241	35,151	593,783
Lax-Wendroff	64	96	0,002	0,192	0,912
	128	192	0,004	0,541	3,881
	256	384	0,010	1,474	17,040
	512	768	0,032	4,629	70,440
	1.024	1.536	0,089	12,647	280,464
Godunov-HLL	64	96	0,004	0,460	1,396
	128	192	0,017	1,963	5,704
	256	384	0,041	4,841	25,844
	512	768	0,143	15,988	103,400
	1.024	1.536	0,363	51,711	405,32

As Figuras 1 e 2 mostram o caráter dissipativo do método Lax-Friedrich, o caráter dispersivo (oscilatório) do Lax-Wendroff e dos métodos Nessyahu-Tadmor e Godunov com pouca dissipação numérica, entretanto, ambos os métodos possuem custos computacionais maiores que os métodos de Lax-Friedrich e Lax-Wendroff.

Um resultado observado na Tabela 13 é que o tempo de execução e o custo computacional não dependem apenas do método numérico escolhido, para resolver um determinado problema matemático (neste caso sistema de EDP's), mas também da linguagem de programação adotada para implementar os métodos. Por exemplo, nota-se que para uma malha de 1.024 subintervalos, com a linguagem *Octave*, o método Nessyahu-Tadmor foi mais lento que o método Godunov-HLL, enquanto que com o *MatLab* houve uma inversão, isto é, o Godunov-HLL foi mais lento que o método Nessyahu-Tadmor.

Pode-se observar ainda grandes diferenças entre os tempos de execuções de cada linguagem. Assim, a escolha da linguagem depende do problema a ser resolvido, combinado com o método numérico a ser empregado para resolvê-lo.

Conforme as exigências computacionais aumentam, cada vez mais a diferença entre as linguagens ficam evidentes e perceptíveis a ponto de tornar inviável a execução de um programa no computador disponível para tal tarefa. Isso é mais visível nas equações de Euler Bidimensionais.

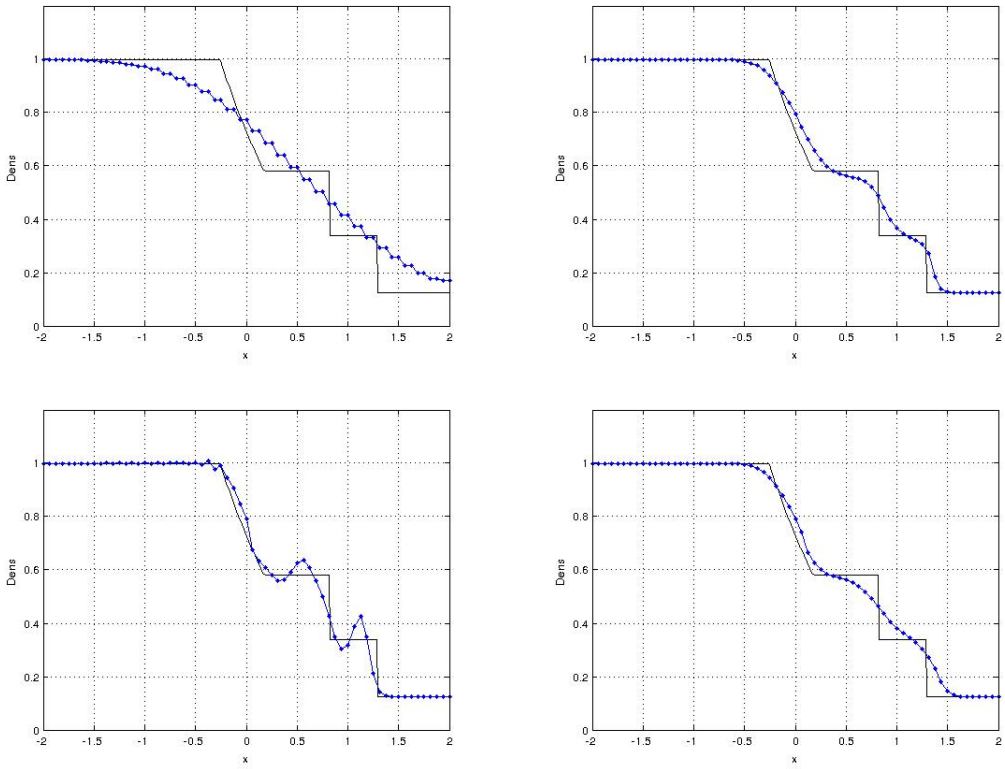

Figura 1: **Evolução da densidade com 64 subintervalos:** na primeira linha à esquerda tem-se a solução aproximada pelo método Lax-Friedrich e à direita pelo método Nessyahu-Tadmor; na segunda linha à esquerda pelo método Lax-Wendroff e à direita pelo método Godunov.

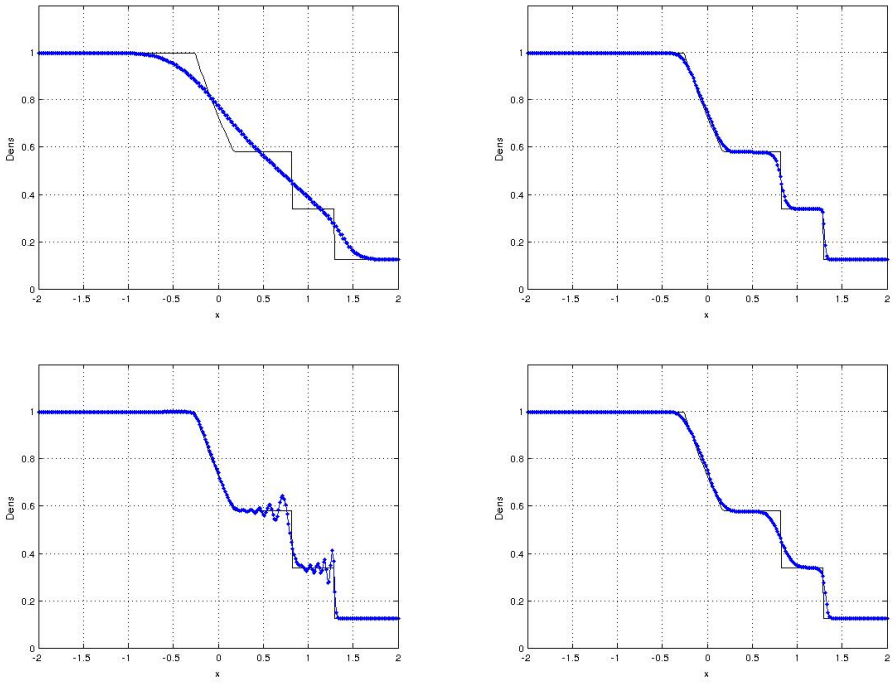

Figura 2: **Evolução da densidade com 256 subintervalos:** na primeira linha à esquerda tem-se a solução aproximada pelo método Lax-Friedrich e à direita pelo método Nessyahu-Tadmor; na segunda linha à esquerda pelo método Lax-Wendroff e à direita pelo método Godunov.

4.2 Equações de Euler Bidimensionais

As equações Bidimensionais de Euler são dadas por

$$\begin{aligned}
&\frac{\partial}{\partial t}\rho + \frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\rho v) = 0 && \text{(equação da continuidade)}\\
&\frac{\partial}{\partial t}\rho u + \frac{\partial}{\partial x}\left(\rho u^2 + p\right) + \frac{\partial}{\partial y}(\rho uv) = 0 && \text{(equação do movimento)}\\
&\frac{\partial}{\partial t}\rho v + \frac{\partial}{\partial x}(\rho uv) + \frac{\partial}{\partial y}\left(\rho v^2 + p\right) = 0 && \text{(equação do movimento)}\\
&\frac{\partial}{\partial t}E + \frac{\partial}{\partial x}(u(E+p)) + \frac{\partial}{\partial y}(v(E+p)) = 0 && \text{(equação da energia)}
\end{aligned} \tag{4}$$

em que

$\rho = \rho(x,y,t)$	(densidade do fluido em escoamento)
$\mathbf{V}(x,y,t) = (u,v)$	(velocidade do fluido)
$p = p(x,y,t)$	(distribuição de pressão no fluido)
$E = E(x,y,t)$	(energia total do fluido)
$\varepsilon = \varepsilon(x,y,t)$	(energia interna do fluido)
γ	(calor específico do fluido)

com

$$E = \frac{1}{2}\left[\rho\left(u^2+v^2\right)\right]+\varepsilon \quad \text{e} \quad \varepsilon = \frac{p}{(\gamma-1)\rho}\,.$$

Utilizando os seguintes valores

- Domínio: $[0,1]\times[0,1]$;
- $\Delta t = 0,2\times min\{\Delta x, \Delta y\}$ para Lax-Friedrich;
- $\Delta t = 0,08\times min\{\Delta x, \Delta y\}$ para Nessyahu-Tadmor;
- Tempo Final: 0.2;

e as condições iniciais mostradas na Figura 3, foram obtidos os valores apresentados na Tabela 14 e as soluções aproximadas expostas nas Figuras 6 e 7.

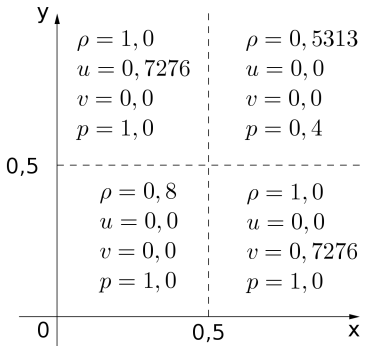

Figura 3: Condições iniciais para densidade, componentes da velocidade e pressão.

Tabela 14: Tempo de execução para obter-se soluções aproximadas

Schemes	subintervals	iterations	Fortran 90	MatLab	Octave
Lax-Friedrich	32×32	32	<p	0,100	17,008
	64×64	64	0,003	0,640	147,411
	128×128	128	0,040	3,51	1.195,392
	256×256	256	0,355	71,42	9.670,444
	512×512	512	9,206	1.150,400	89.921,345
Nessyahu-Tadmor	32×32	80	0,006	1,728	174,416
	64×64	160	0,09	10,816	1.468,05
	128×128	320	0,744	106,560	11.808,671
	256×256	640	5,360	1.078,081	93.801,600
	512×512	1280	34,411	13.772,800	-

Na referida tabela nota-se a discrepância entre as linguagens com relação ao tempo de execução. Por exemplo, no método Nessyahu-Tadmor com a malha 512×512, o *Octave* demorou aproximadamente $2,4$ dias para obter os gráficos das Figuras 6 e 7, enquanto que o *MatLab* demorou $5,4$ horas. Esse longo tempo de execução do *Octave* pode torná-lo inviável para simulações de médio e grande porte, ou ainda, em problemas operacionais como por exemplo, em previsões numéricas (fluidos atmosféricos) do tempo em que não se pode esperar nem mesmo um dia para a obtenção de resultados numéricos.

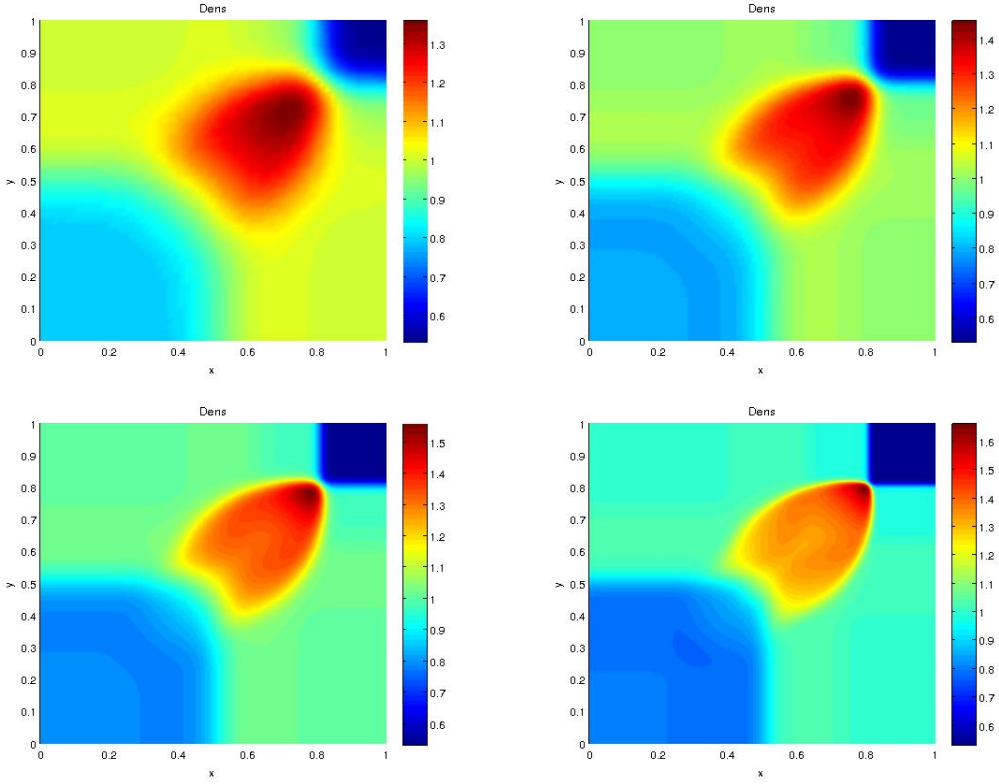

Figura 4: Evolução da densidade pelo Método L-F: na primeira linha à esquerda tem-se a solução obtida com a malha 64×64 e à direita com a malha 128×128; na segunda linha malhas 256×256 à esquerda e 512×512 à direita.

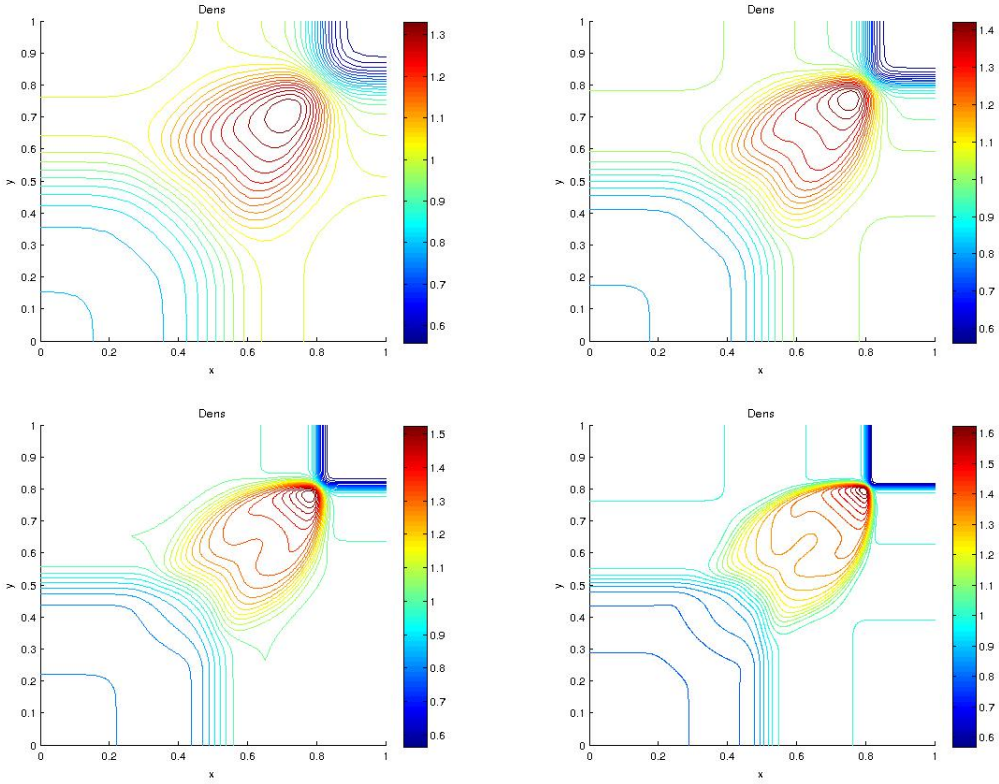

Figura 5: Evolução da densidade pelo Método L-F: na primeira linha à esquerda tem-se a solução obtida com a malha 64×64 e à direita com a malha 128×128; na segunda linha malhas 256×256 à esquerda e 512×512 à direita.

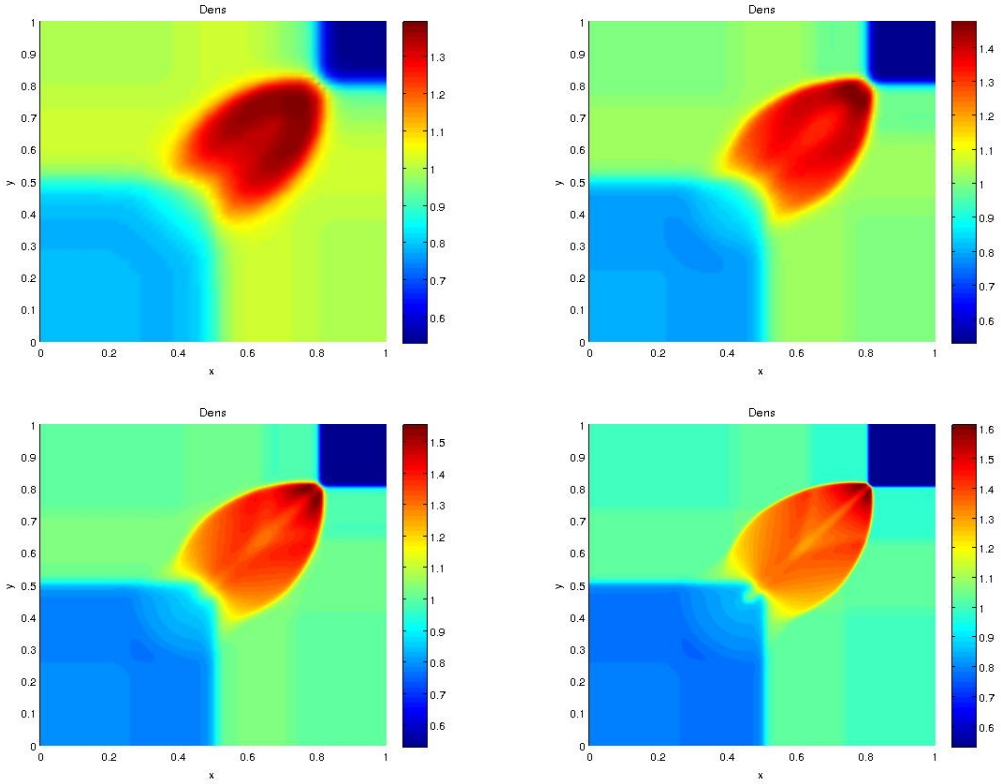

Figura 6: Evolução da densidade pelo Método N-T: na primeira linha à esquerda tem-se a solução obtida com a malha 64×64 e à direita com a malha 128×128; na segunda linha malhas 256×256 à esquerda e 512×512 à direita.

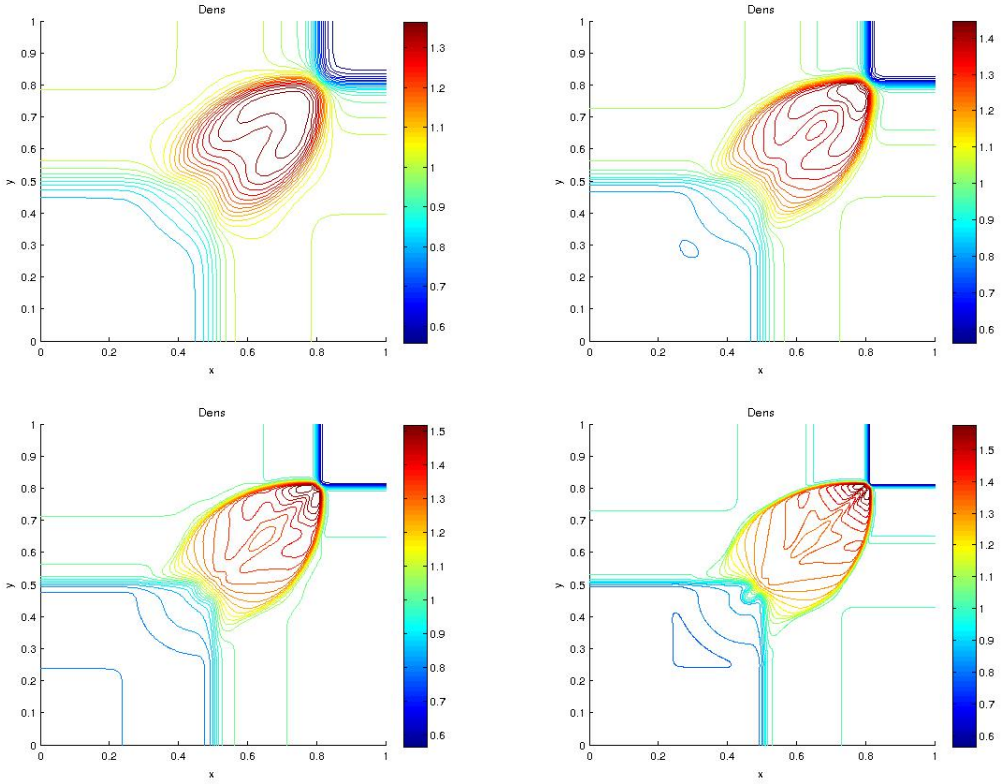

Figura 7: Evolução da densidade pelo Método N-T: na primeira linha à esquerda tem-se a solução obtida com a malha 64×64 e à direita com a malha 128×128; na segunda linha malhas 256×256 à esquerda e 512×512 à direita.

5 Conclusões

Neste trabalho foram analisadas as diferenças entre o *Fortran 90*, o *MatLab* e o *Octave*. De início investigou-se as diferenças entre o tempo de execução em testes relacionados a alguns comandos básicos essenciais para a implementação computacional de métodos numéricos, a saber, comandos de laço de repetição, tomada de decisão, acesso a funções e armazenamentos de *array's*.

Os testes 1, 2 e 3 foram executados pelo *MatLab* tanto com a função *JIT* ativada quanto desativada, evidenciando a importância do papel do seu compilador interno, que é acelerar a execução do código a fim de diminuir o tempo. Assim, uma opção para o *Octave* também diminuir o tempo de execução de suas funções básicas é desenvolver um compilador interno.

Entretanto, verificou-se que apenas o *JIT* ativo não é suficiente para que o *MatLab* tenha um desempenho semelhante ao do *Fortran*. No Teste 3 foi possível observar que mesmo o *MatLab* sendo uma linguagem interpretada, que não depende de declaração de variáveis para executar o código, o procedimento de defini-las antes de utilizá-las em alguma operação matemática, teve um forte impacto na redução no tempo de execução. Com esses cuidados, o *MatLab* passou a ter um desempenho semelhante ao do *Fortran*. Já o *Octave*, embora tenha eduzido o tempo de execução, obteve poucas melhorias.

No Teste 4, os códigos construíram a mesma matriz do Teste 3. Entretanto, a definição dos elementos da matriz foi escrita em uma subrotina externa ao programa principal. Dessa forma, o Teste 4 adicionou um tempo necessário para acionar a função externa, para definir cada elemento da matriz. Nessa situação, o *MatLab* passou a ser em torno de 11 vezes mais lento que o *Fortran* e o *Octave* em torno de 100 vezes mais lento que o *MatLab*. Ainda neste exemplo, ficou evidenciada a importância de se definir previamente a matriz a ser construída. Uma alternativa para diminuir essa diferença é fazer com que o *MatLab* reconheça as funções ou subrotinas externas, antes de executar o programa principal, isto é, utilizar-se de procedimentos semelhantes aos recursos de linguagens compiladas.

Um resultado observado na Tabela 13 foi que o tempo de execução e o custo computacional não dependem apenas do método numérico escolhido para resolver um determinado problema matemático (neste caso sistema de equações diferenciais parciais), mas também, da linguagem de programação escolhida para implementar os métodos, atrelada à técnica de programação empregada. Por exemplo, nota-se que para a malha de 1.024 subintervalos, com a linguagem *Octave*, o método Nessyahu-Tadmor foi mais lento que o método Godunov-HLL enquanto que, com o *MatLab* o Godunov-HLL foi mais lento que o método Nessyahu-Tadmor.

De modo geral, houve grandes diferenças entre os tempos de execuções de cada linguagem. Assim, a escolha da linguagem depende do problema a ser resolvido, combinado com o método numérico a ser empregado para resolver tal problema, sem se descuidar da técnica de programação aplicada na escrita do código.

Referências

[1] SILVEIRA, G. P.; BARROS, L. C. Numerical methods integrated with fuzzy logic and stochastic method for solving PDEs: an application to dengue. **Fuzzy Sets and Systems**, v. 225, p. 39-57, 2013.

[2] GARCIA R. O.; OLIVEIRA, S. R. Substructures in simulations of relativistic jet formation. **Braz. J. Phys.**, v. 47, n. 2, p. 197-214, 2017.

[3] TORO, E. **Riemann solvers and numerical methods for fluid dynamics**. Berlin: Springer-Verlag, 2017.

[4] QUARTERONI, A.; SALERI F. **Cálculo científico com MATLAB e Octave.** Mailand: Springer-Verlag, 2017.

[5] MATHWORKS. **MATLAB.** [S.l.], c1994-2017. Disponível em: <http://www.mathworks.com/products/matlab>. Acesso em: 18 jul. 2017.

[6] OCTAVE. [S.l.: s.n.], c1998-2017. Disponível em: <http://www.octave.org>. Acesso em: 18 jul. 2017.

[7] BREWSTER, M. W. Study of free alternative numerical computational packages. **SIAM,** Philadelphia, v. 5, 2012.

[8] ALMEIDA, E. S.; MEDEIROS, A. C.; FRERY, A. C. How good are MatLab, Octave and Scilab for computational modelling? **Comput. Appl. Math.**, v. 31, n. 3, p. 523-538, 2012.

[9] WOUWER, A. V.; SAUCEZ, P.; VILAS, C. **Simulation of ODE/PDE models with MATLAB, OCTAVE and SCILAB.** Cham: Springer, 2014.

[10] SHARMA, N.; GOBBERT, M. K. A comparative evaluation of MATLAB, Octave and SciLab for research and teaching. **HPCF Publications**. 2010. Disponível em: <https://userpages.umbc.edu/ gobbert/papers/SharmaGobbertTR2010.pdf>. Acesso em: 18 jul. 2017.

[11] SILVEIRA, G. P.; BARROS, L. C. Analysis of the dengue risk by means of a Takagi-Sugeno-style model. **Fuzzy Sets and Systems**, v. 277, p. 122-137, 2015.

[12] LEVEQUE, R. J. **Finite volume methods for hyperbolic problems.** New York: Cambridge University Press, 2002.

[13] NESSYAHU, H.; TADMOR, E. Non-oscillatory central differencing for hyperbolic conservation laws. **J. Comput. Phys.**, v. 87, n. 2, p. 408-463, 1990.

Artigo recebido em ago. 2017 e aceito em nov. 2017.