Lista 3

Axissimetria e Paredes Não-Alinhadas

Aeroacústica Computacional

Aluno:
José Pedro de Santana Neto - 201505394

Professor: **Andrey Ricardo da Silva, PhD.**

12 de novembro de 2015

1 Axissimetria

1.1 Procedimentos

Para a implementação da simulação em questão foi seguido os seguintes procedimentos:

1. Foram implementadas as condições anecóicas ao longo dos cantos do lattice, uma sobreposta a outra nos cantos;
2. As barreiras foram implementadas nos cantos e no duto de forma separada entre eles;
3. A condição de axissimetria foi implementada fazendo com que não haja reflxão não-física na parte aberta do lattice;
4. O *chirp* foi implementado utilizando a condição anecóica de massa variante e velocidade de partícula e mach 0.07 para os dois casos respectivamente.

1.2 Códigos

Segue os códigos desenvolvidos:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This is a simple implementation of the LBGK model.
%% By Andrey R. da Silva, August 2010
%%
%% The code does not take into account eny specific boundaru
    condiition.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all, clc
close all
```

```
tic
% Block 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Lattice size
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Nr = 251;                     % Number of lines   (cells in the y
    direction)
Mc = 502;                     % Number of columns (cells in the x
    direction)


% Block 2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Physical parameters (macro)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

c_p = 340;                    % Sound velocity on the fluid [m/s]
rho_p = 1.2;                  % physical density [kg/m^3]
rho_p = 1;                    % Fluid density  [kg/m^3]
Lx = .5;                      % Maximun dimenssion in the x direction
     [m]
Ly = 0.0833;                  % Maximun dimenssion on th y direction
     [m]
Dx = Lx/Mc                    % Lattice space (pitch)
Dt = (1/sqrt(3))*Dx/c_p       % lattice time step


% Block 3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Lattice parameters (micro - lattice unities)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
omega = 1.9;                                        % Relaxation
    frequency
tau = 1/omega;                                      % Relaxation time
rho_l = 1;                                          % avereged fluid
    density (latice density
cs = 1/sqrt(3);                                     % lattice speed of
```

```
    sound
cs2 = cs^2;                                   % Squared speed of
    sound cl^2
visc = cs2*(1/omega-0.5);                     % lattice viscosity
visc_phy = visc*(Dx^2)/Dt;                    % physical
    kinematic viscosity


% Block 4
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Lattice properties for the D2Q9 model
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


N_c=9 ;                                        % number of
    directions of the D2Q9 model
C_x=[1 0 -1  0 1 -1 -1  1 0];                  % velocity
    vectors in x
C_y=[0 1  0 -1 1  1 -1 -1 0];                  % velocity
    vectors in y
w0=16/36. ; w1=4/36. ; w2=1/36.;               % lattice weights
W = [w1 w1 w1 w1 w2 w2 w2 w2 w0];
f1=3.;
f2=4.5;
f3=1.5;                                        % coef. of the f
    equil.

% Array of distribution and relaxation functions
f=zeros(Nr,Mc,N_c);
feq=zeros(Nr,Mc,N_c);

% Filling the initial distribution function (at t=0) with initial
    values
f(:,:,:)=rho_l/9;
ux = zeros(Nr, Mc);
uy = zeros(Nr, Mc);
```

```

% Calculando a condicao anecoica
% 4.0.1 - Adding conditions anechoic
distance = 30;
% condicao anecoica para cima
growth_delta = 0.5;
[sigma_mat9_cima Ft_cima] = build_anechoic_condition(Mc, ...
Nr, distance, growth_delta);
% condicao anecoica para esquerda
growth_delta = -1;
[sigma_mat9_esquerda Ft_esquerda] = build_anechoic_condition(Mc, ...
Nr, distance, growth_delta);
% condicao anecoica para direita
growth_delta = 1;
[sigma_mat9_direito Ft_direito] = build_anechoic_condition(Mc, ...
Nr, distance, growth_delta);

% Vendo pontos de barreira da xirizuda
xl = [2 2 501 501];
yl = [1 250 250 1];
[vec1,vec2,vec3,vec4,vec5,vec6,vec7,vec8] = crossing3(Nr,Mc,xl,yl);
%xl = [250 250];
%yl = [1 50];
xl = [33 33 250];
yl = [1 21 21];
[vec1_duto,vec2_duto,vec3_duto,vec4_duto,vec5_duto,vec6_duto,
    vec7_duto,vec8_duto] = ...
crossing3(Nr,Mc,xl,yl);

%Block 5
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Begin the iteractive process
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Construindo chirp
total_time = 5*Mc*sqrt(3); % meia hora = 20*Mc*sqrt(3)
```

```
times = 0 : total_time - 1;
initial_frequency = 0;
frequency_max_lattice = 1.8/(2*pi*20*sqrt(3));
source_chirp = chirp(times, ...
initial_frequency, times(end), frequency_max_lattice);
% vetores de pressao e velocidade de particula
pressure(1:total_time) = 0;
particle_velocity(1:total_time) = 0;
for ta = 1 : total_time

    % Block 5.1
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% propagation (streaming)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    f(:,:,1) = [f(:,1:2,1) f(:,2:Mc-1,1)];
    f(:,:,2) = [f(1:2,:,2);f(2:Nr-1,:,2)];
    f(:,:,3) = [f(:,2:Mc-1,3) f(:,Mc-1:Mc,3)];
    f(:,:,4) = [f(2:Nr-1,:,4);f(Nr-1:Nr,:,4)];
    f(:,:,5) = [f(:,1:2,5) f(:,2:Mc-1,5)];
    f(:,:,5) = [f(1:2,:,5);f(2:Nr-1,:,5)];
    f(:,:,6) = [f(:,2:Mc-1,6) f(:,Mc-1:Mc,6)];
    f(:,:,6) = [f(1:2,:,6);f(2:Nr-1,:,6)];
    f(:,:,7) = [f(:,2:Mc-1,7) f(:,Mc-1:Mc,7)];
    f(:,:,7) = [f(2:Nr-1,:,7);f(Nr-1:Nr,:,7)];
    f(:,:,8) = [f(:,1:2,8) f(:,2:Mc-1,8)];
    f(:,:,8) = [f(2:Nr-1,:,8);f(Nr-1:Nr,:,8)];

    G=f;
    f(vec1)=G(vec3);
    f(vec3)=G(vec1);
    f(vec2)=G(vec4);
    f(vec4)=G(vec2);
    f(vec5)=G(vec7);
    f(vec7)=G(vec5);
    f(vec6)=G(vec8);
    f(vec8)=G(vec6);

    G=f;
```

```
    f(vec1_duto)=G(vec3_duto);
    f(vec3_duto)=G(vec1_duto);
    f(vec2_duto)=G(vec4_duto);
    f(vec4_duto)=G(vec2_duto);
    f(vec5_duto)=G(vec7_duto);
    f(vec7_duto)=G(vec5_duto);
    f(vec6_duto)=G(vec8_duto);
    f(vec8_duto)=G(vec6_duto);

    % Block 5.2
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % recalculating rho and u
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    rho=sum(f,3);

    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% Calculando uma fonte ABC dentro do duto
    % direita = 0.5
    density_source = rho_l + 0.0001*source_chirp(ta);
    Ux_t = (0.0001*source_chirp(ta))/sqrt(3) + 0.07*cs;
    Uy_t = 0;
    point_y = 1;
    distance_y = 18;
    point_x = 34;
    distance_x = 30;
    direction = 0.5;
    [sigma_source Ft_source] = build_source_anechoic(Nr, Mc, ...
    density_source, Ux_t, Uy_t, point_y, ...
    point_x, distance_x, distance_y, direction);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %if ta == 1
    %rho(15, 100) = rho_l + 0.0001 *sin(ta);
    %end

    rt0= w0*rho;
    rt1= w1*rho;
    rt2= w2*rho;
```

```
% Determining the velocities according to Eq.() (see slides)
ux = (C_x(1).*f(:,:,1)+C_x(2).*f(:,:,2)+C_x(3).*f(:,:,3)+C_x(4).*f(:,:,4)+C_x(5).*f(:,:,5)+C_x(6).*f(:,:,6)+C_x(7).*f(:,:,7)+C_x(8).*f(:,:,8))./rho ;
uy = (C_y(1).*f(:,:,1)+C_y(2).*f(:,:,2)+C_y(3).*f(:,:,3)+C_y(4).*f(:,:,4)+C_y(5).*f(:,:,5)+C_y(6).*f(:,:,6)+C_y(7).*f(:,:,7)+C_y(8).*f(:,:,8))./rho ;


% Block 5.3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Determining the relaxation functions for each direction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
uxsq=ux.^2;
uysq=uy.^2;
usq=uxsq+uysq;

feq(:,:,1)= rt1 .*(1 +f1*ux +f2.*uxsq -f3*usq);
feq(:,:,2)= rt1 .*(1 +f1*uy +f2*uysq -f3*usq);
feq(:,:,3)= rt1 .*(1 -f1*ux +f2*uxsq -f3*usq);
feq(:,:,4)= rt1 .*(1 -f1*uy +f2*uysq -f3*usq);
feq(:,:,5)= rt2 .*(1 +f1*(+ux+uy) +f2*(+ux+uy).^2 -f3.*usq);
feq(:,:,6)= rt2 .*(1 +f1*(-ux+uy) +f2*(-ux+uy).^2 -f3.*usq);
feq(:,:,7)= rt2 .*(1 +f1*(-ux-uy) +f2*(-ux-uy).^2 -f3.*usq);
feq(:,:,8)= rt2 .*(1 +f1*(+ux-uy) +f2*(+ux-uy).^2 -f3.*usq);
feq(:,:,9)= rt0 .*(1 - f3*usq);

% Block 5.4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Collision (relaxation) step
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Condicao Axissimetrica
    % termo de primeira ordem
        % y por x
h_1_leaf = zeros(Nr, Mc);
radius = 1:Nr;
radius = radius';
```

```
    for column = 1:Mc
        h_1_leaf(:, column) = uy(:, column)./radius;
    end
            % construindo a matriz com os pesos de cada direcao
    h_1 = zeros(Nr,Mc,N_c);
    for direction = 1:9
        h_1(:, :, direction) = - W(direction)*rho_l*h_1_leaf;
    end

    % termo de segunda ordem
    % parte 1
    part_1_second_term = zeros(Nr, Mc); % Nr => y; Mc => x;
    part_1_second_term = diff(rho, 1, 1)*cs2;
    part_1_second_term(Nr, :) = part_1_second_term(Nr - 1, :);

    % parte 2
    part_2_second_term = zeros(Nr, Mc); % Nr => y; Mc => x;
    u_r = uy;
    derivada_ux_x = diff(ux, 1, 2);
    derivada_ux_x(:, Mc) = derivada_ux_x(:, Mc - 1);
    derivada_ur_x = diff(u_r, 1, 2);
    derivada_ur_x(:, Mc) = derivada_ur_x(:, Mc - 1);
    derivada_ux_ur_x = derivada_ux_x.*u_r + derivada_ur_x.*ux;
    part_2_second_term = rho_l*derivada_ux_ur_x;

    % parte 3
    part_3_second_term = zeros(Nr, Mc); % Nr => y; Mc => x;
    derivada_ur_r = diff(u_r, 1, 1);
    derivada_ur_r(Nr, :) = derivada_ur_r(Nr - 1, :);
    derivada_ur_ur_r = 2*u_r.*derivada_ur_r;
    part_3_second_term = rho_l*derivada_ur_ur_r;

    % termo parcial para as 9 direcoes
    partial_part_second_term = zeros(Nr, Mc, 9);
    for direction = 1:9
        partial_part_second_term(:,:,direction) = part_1_second_term
   + ...
        part_2_second_term + part_3_second_term;
    end
```

```
% parte 4
part_4_second_term = zeros(Nr, Mc, 9); % Nr => y; Mc => x;
derivada_ux_r = diff(ux, 1, 1);
derivada_ux_r(Nr, :) = derivada_ux_r(Nr - 1, :);
for direction = 1:9
    part_4_second_term(:, :, direction) = ...
    rho_l*(derivada_ux_r - derivada_ur_x)*C_x(direction);
end

% parte total
total_part_second_term = part_4_second_term +
partial_part_second_term;

% calculando matriz de viscosidade
omega_matrix_second_term = zeros(Nr, Mc, 9);
viscosity_matrix(1:Nr, 1:Mc) = 3*visc;
radius = 1:Nr;
radius = radius';
for point_x = 1 : Mc
    viscosity_matrix(:, point_x) = viscosity_matrix(:, point_x)./
radius;
end
for direction = 1 : 9
    omega_matrix_second_term(:, :, direction) = ...
    W(direction)*viscosity_matrix;
end

% finalmente o termo de segunda ordem
h_2 = omega_matrix_second_term.*total_part_second_term;

% colidindo tudo
f = (1-omega)*f + omega*feq ...
- sigma_mat9_cima.*(feq- Ft_cima) ...
- sigma_mat9_esquerda.*(feq- Ft_esquerda) ...
- sigma_mat9_direito.*(feq- Ft_direito) ...
- sigma_source.*(feq- Ft_source) ...
+ h_1 + h_2;
```

```
    % Ploting the results in real time
    %surf(rho-1), view(2), shading flat, axis equal, caxis([-.00001 .00001])
    %grid off
    %if mod(ta, 100) == 0
        %vorticidade = curl(ux, uy);
        %velocity = sqrt(ux.^2 + uy.^2);
        %imagesc(flip(vorticidade))
        %imagesc(flip(rho-1), [-.000001 .000001]);
        %pause(.00001);
        %disp('Progresso: ');
        %disp((ta/total_time*100));
    %end
    pressure(ta) = (mean(rho(1:19, 250)-1))*cs2;
    particle_velocity(ta) = mean(ux(1:19, 250) - 0.07*cs);
    %disp('Progresso: ');
    %disp((ta/total_time*100));
end %  End main time Evolution Loop

fft_pressure = fft(pressure);
fft_particle_velocity = fft(particle_velocity);
impedance = fft_pressure./fft_particle_velocity;
number_helmholtz_max = (2*pi*20)/cs;
numbers_helmholtz = linspace(0, number_helmholtz_max, length(particle_velocity));

figure(2);
max_normalization = max([max(abs(real(impedance))) max(abs(real(impedance)))]);
plot(numbers_helmholtz, real(impedance)/max_normalization);
hold on;
plot(numbers_helmholtz, imag(impedance)/max_normalization, 'r');
ylabel('Impedancia','FontSize',20);
xlabel('Numero de Helmholtz','FontSize',20);
title('Impedancia do Sistema','FontSize',20);
legend('Real', 'Imaginaria');
axis([0 15 -1.3 1.3]);
```

```
toc
```

code/lista_3.m

```
%% build_anechoic_condition: function description
function [sigma_mat9 Ft] = build_anechoic_condition(
    number_lines_lattice, ...
number_columns_lattice, distance, growth_delta)

  lattice_sound_speed = 1/sqrt(3);
  lattice_sound_speed_pow_2 = lattice_sound_speed^2;
  w1=4/9;      % centro     %pesos de relaxacao devido ao D2Q9 (pg.20)
  w2=1/9;      % ortogonais
  w3=1/36;     % diagonais
  coef1=  1/(2*lattice_sound_speed_pow_2^2); %para uso na relaxacao
  coef2= -1/(2*lattice_sound_speed_pow_2);
 % funcoes distribuicao (Eq. 1.46)
  Ft = zeros(number_lines_lattice,number_columns_lattice,9);
 %funcoes target
  Ux_t=0;
  Uy_t=0;
  U_t=Ux_t^2+Uy_t^2;
  densi_t = 1;
  Ft(:,:,9)= w1*densi_t.*(1+coef2*U_t);
  Ft(:,:,1)= w2*densi_t.*(1 +Ux_t/lattice_sound_speed_pow_2 +coef1*(
    Ux_t.^2 )+coef2*U_t);
  Ft(:,:,2)= w2*densi_t.*(1 +Uy_t/lattice_sound_speed_pow_2 +coef1*(
    Uy_t.^2 ) +coef2*U_t);
  Ft(:,:,3)= w2*densi_t.*(1 -Ux_t/lattice_sound_speed_pow_2 +coef1*(
    Ux_t.^2 )+coef2*U_t);
  Ft(:,:,4)= w2*densi_t.*(1 -Uy_t/lattice_sound_speed_pow_2 +coef1*(
    Uy_t.^2) +coef2*U_t);
  Ft(:,:,5)= w3*densi_t.*(1 +(+Ux_t+Uy_t)/lattice_sound_speed_pow_2 +
    coef1*((+Ux_t+Uy_t).^2) +coef2*U_t);
  Ft(:,:,6)= w3*densi_t.*(1 +(-Ux_t+Uy_t)/lattice_sound_speed_pow_2 +
    coef1*((-Ux_t+Uy_t).^2) +coef2*U_t);
  Ft(:,:,7)= w3*densi_t.*(1 +(-Ux_t-Uy_t)/lattice_sound_speed_pow_2 +
    coef1*((-Ux_t-Uy_t).^2) +coef2*U_t);
```

```
Ft(:,:,8)= w3*densi_t.*(1 +(+Ux_t-Uy_t)/lattice_sound_speed_pow_2 + coef1*((+Ux_t-Uy_t).^2) +coef2*U_t);
%
D_t = distance;  % em numero de celulas
sigma_t = 0.3;
delta_t = 0:D_t - 1;
sigma = sigma_t*(delta_t/D_t).^2;
sigma = sigma';
sigma_mat = [];

% x e y nesse caso
sigma_mat9 = zeros(number_lines_lattice, number_columns_lattice, 9);
matrix_sigma_leaf = zeros(number_lines_lattice, number_columns_lattice);
% condicao anecoica para a direita
if growth_delta == 1
  % construindo a matriz de nove folhas sigma
  size_x = number_lines_lattice - D_t + 1 : number_lines_lattice;
  size_y = 1:number_columns_lattice;
  matrix_sigma_leaf(size_x, size_y) = 1;

  for number_column = 1: number_columns_lattice
    matrix_sigma_leaf(size_x, number_column) = ...
    matrix_sigma_leaf(size_x, number_column).*sigma;
  end
% condicao anecoica para a esquerda
elseif growth_delta == -1
  % construindo a matriz de nove folhas sigma
  size_x = 1 : D_t;
  size_y = 1 : number_columns_lattice;
  matrix_sigma_leaf(size_x, size_y) = 1;
  for number_column = 1: number_columns_lattice
    matrix_sigma_leaf(size_x, number_column) = ...
    matrix_sigma_leaf(size_x, number_column).*flip(sigma);
  end
% condicao anecoica para cima
elseif growth_delta == 0.5
  size_x = 1 : number_lines_lattice;
```

```
    size_y = number_columns_lattice - D_t + 1 :
   number_columns_lattice;
    matrix_sigma_leaf(size_x, size_y) = 1;
    for number_line =  1 : number_lines_lattice
      matrix_sigma_leaf(number_line, size_y) = ...
      matrix_sigma_leaf(number_line, size_y).*sigma';
    end
  % condicao anecoica para baixo
  elseif growth_delta == -0.5
    size_x = 1 : number_lines_lattice;
    size_y = 1 : D_t;
    matrix_sigma_leaf(size_x, size_y) = 1;
    for number_line =  1 : number_lines_lattice
      matrix_sigma_leaf(number_line, size_y) = ...
      matrix_sigma_leaf(number_line, size_y).*flip(sigma)';
    end
  end

  for number_leaf = 1:9
    sigma_mat9(:,:,number_leaf) = matrix_sigma_leaf;
  end

  % Agora eh y por x
  sigma_mat9_aux = zeros(number_columns_lattice, number_lines_lattice
   , 9);
  Ft_aux = zeros(number_columns_lattice, number_lines_lattice, 9);
  for direction = 1 : 9
    sigma_mat9_aux(:, :, direction) = sigma_mat9(:, :, direction)';
    Ft_aux(:, :, direction) = Ft(:, :, direction)';
  end
  sigma_mat9 = sigma_mat9_aux;
  Ft = Ft_aux;
```

code/build_anechoic_condition.m

```
%% build_source_anechoic: direction = 1 (virado para cima)
%                 direction = -1 (virado para baixo)
%                 direction = 0.5 (virado para direita)
%                 direction = -0.5 (virado para esquerda)
```

```
function [sigma_source Ft_source] = build_source_anechoic(Nr, Mc, ...
density, Ux_t, Uy_t, point_y, point_x, distance_x, distance_y, direction)

  lattice_sound_speed = 1/sqrt(3);
  lattice_sound_speed_pow_2 = lattice_sound_speed^2;
  cs2 = lattice_sound_speed_pow_2;
  cs = lattice_sound_speed;
  w1=4/9;     % centro    %pesos de relaxacao devido ao D2Q9 (pg.20)
  w2=1/9;     % ortogonais
  w3=1/36;    % diagonais
  coef1= 1/(2*cs2^2); %para uso na relaxacao
  coef2= -1/(2*cs2);
  % funcoes distribuicao (Eq. 1.46)
  Ft_source = zeros(Nr, Mc, 9);
  %funcoes target
  Ux_t = Ux_t;
  Uy_t = Uy_t;
  U_t=Ux_t^2+Uy_t^2;
  densi_t = density;
  Ft_source(:,:,9)= w1*densi_t.*(1+coef2*U_t);
  Ft_source(:,:,1)= w2*densi_t.*(1 +Ux_t/cs2 +coef1*(Ux_t.^2 )+coef2*U_t);
  Ft_source(:,:,2)= w2*densi_t.*(1 +Uy_t/cs2 +coef1*(Uy_t.^2 ) +coef2*U_t);
  Ft_source(:,:,3)= w2*densi_t.*(1 -Ux_t/cs2 +coef1*(Ux_t.^2 )+coef2*U_t);
  Ft_source(:,:,4)= w2*densi_t.*(1 -Uy_t/cs2 +coef1*(Uy_t.^2) +coef2*U_t);
  Ft_source(:,:,5)= w3*densi_t.*(1 +(+Ux_t+Uy_t)/cs2 +coef1*((+Ux_t+Uy_t).^2) +coef2*U_t);
  Ft_source(:,:,6)= w3*densi_t.*(1 +(-Ux_t+Uy_t)/cs2 +coef1*((-Ux_t+Uy_t).^2) +coef2*U_t);
  Ft_source(:,:,7)= w3*densi_t.*(1 +(-Ux_t-Uy_t)/cs2 +coef1*((-Ux_t-Uy_t).^2) +coef2*U_t);
  Ft_source(:,:,8)= w3*densi_t.*(1 +(+Ux_t-Uy_t)/cs2 +coef1*((+Ux_t-Uy_t).^2) +coef2*U_t);

  % posicionando a fonte
```

```
sigma_source = zeros(Nr, Mc, 9);
% assintota para a direita
if direction == 0.5
  sigma_t = 0.3;
  delta_t = 0:distance_x;
  sigma = sigma_t*(delta_t/distance_x).^2;
  sigma_source_leaf = zeros(Nr, Mc);
  size_x = point_x : point_x + distance_x;
  size_y = point_y : point_y + distance_y;
  sigma_source_leaf(size_y, size_x) = 1;
  for point_y = size_y(1) : size_y(end)
    sigma_source_leaf(point_y, size_x) = ...
    sigma_source_leaf(point_y, size_x).*flip(sigma);
  end
  for direction = 1:9
    sigma_source(:, :, direction) = sigma_source_leaf(:, :);
  end
end
```

code/build_source_anechoic.m

1.3 Resultados

Segue os gráficos obtidos:

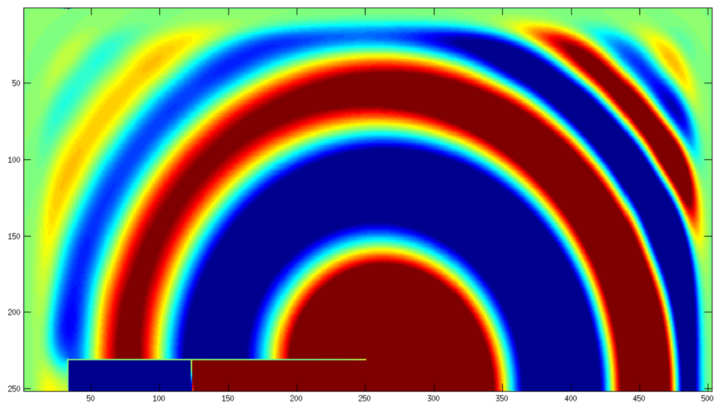

Figura 1: Gráfico de densidades.

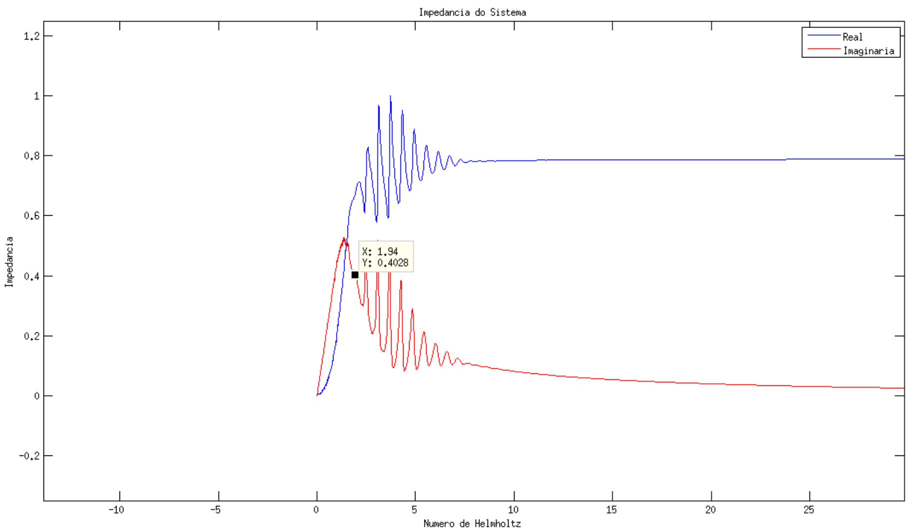

Figura 2: Gráfico de impedância sem escoamento.

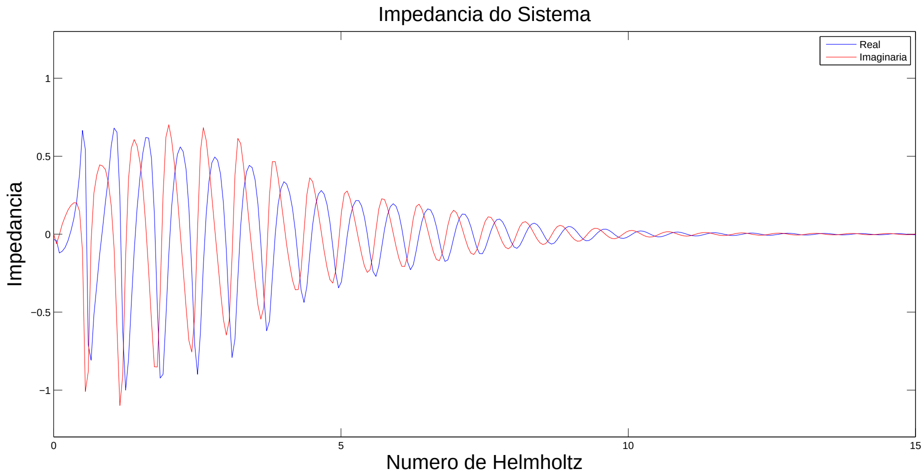

Figura 3: Gráfico de impedância com escoamento (Mach = 0.07).

2 Paredes Não-Alinhada

Para o desenvolvimento dessa questão foi criada uma nova função com essa assinatura:

```
function [q, vecloc] = funcao(i, vecx, vecy, Nr, Mc)
```

Ela recebe os parâmetros de número de colunas (Nr), número de colunas (Mc), pontos da parede não-alinhada ao longo da ordenada (vecy), pontos da parede não-alinhada ao longo da abscissa (vecx) e direção da célula lattice (i) que pode variar de 1 a 9. Nesse caso só foi possível implementar na direção 1, ou seja, todas as células que cruzam para a direita a barreira não-alinhada. De retorno a função devolve a matriz das distâncias das células que cruzam a barreira (q) e a matriz de células que cruzam a barreira.

O algorítmo pensado para resolver esse problema possui os seguintes passos:

1. Verificar se os pontos da barreira estão dentro do tamanho da matriz de lattice:

```
% verify if the points is inside of lattice
is_not_points_inside_lattice = ...
min(vecx) < 1 || max(vecx) > Mc ...
|| min(vecy) < 1 || max(vecy) > Nr;
if is_not_points_inside_lattice
    i = -1;
    message = 'The points x and y are outside of lattice.';
    disp(message);
end
```

2. Assumir uma rota de processamento para a direção lattice i inserida:

```
if i == 1
```

3. Restringir o domínio de lattice num quadrado de tamanho mínimo que caiba toda a barreira não-alinhada:

```
vecloc = zeros(Nr, Mc);
q = zeros(Nr, Mc);
% square that have the entire points
square_x = [floor(min(vecx)) ceil(max(vecx))];
square_y = [floor(min(vecy)) ceil(max(vecy))];
```

4. Ir iterando em todas as alturas dentro do quadrado mínimo calculado:

```
for point_y = square_y(1):square_y(2)
```

5. Calcular os dois pontos mais próximos para fazer a interpolação:

```
% looking for a x point
% (Verify whats p1 and p2 is nearest from height)
distances_y = abs(vecy - point_y);
slot_min = find(distances_y == min(distances_y));
p2_y = vecy(slot_min);
p2_x = vecx(slot_min);
distances_y(slot_min) = 10e10;
slot_min = find(distances_y == min(distances_y));
p1_y = vecy(slot_min);
p1_x = vecx(slot_min);

```

6. Realizar a interpolação linear:

```
% Agora tenho p1 e p2, tenho que agora fazer a
% interpolacao linear para achar o p3
% equacao da reta: y = ax + b
a = (p2_y - p1_y)/(p2_x - p1_x);
b = p1_y - a*p1_x;
p3_x = (point_y - b)/a;
```

7. Calculando a distância e adquirindo a célula que cruza a barreira a direita:

```
% e calcular a distancia em x do ponto que eu to para o x de
p3
distances_points = [square_x(1):square_x(2)] - p3_x;
distances_points = abs(distances_points);
point_x = find(distances_points == min(distances_points));
vecloc(point_y, point_x) = 1;
q(point_y, point_x) = min(distances_points);
```

Segue o código em sua totalidade:

```
function [q, vecloc] = funcao(i, vecx, vecy, Nr, Mc)

 % verify if the points is inside of lattice
 is_not_points_inside_lattice = ...
 min(vecx) < 1 || max(vecx) > Mc ...
 || min(vecy) < 1 || max(vecy) > Nr;
 if is_not_points_inside_lattice
   i = -1;
   message = 'The points x and y are outside of lattice.';
   disp(message);
 end

 % direction 1 of lattice cell
 if i == 1
```

```
    vecloc = zeros(Nr, Mc);
    q = zeros(Nr, Mc);
    % square that have the entire points
    square_x = [floor(min(vecx)) ceil(max(vecx))];
    square_y = [floor(min(vecy)) ceil(max(vecy))];
    %vecloc(square_x(1):square_x(2), ...
    %square_y(1):square_y(2)) = 1;

    % getting points on the left of curve
    distances_points = [];
    for point_y = square_y(1):square_y(2)
      % looking for a x point
      % (Verify whats p1 and p2 is nearest from height)
      distances_y = abs(vecy - point_y);
      slot_min = find(distances_y == min(distances_y));
      p2_y = vecy(slot_min);
      p2_x = vecx(slot_min);
      distances_y(slot_min) = 10e10;
      slot_min = find(distances_y == min(distances_y));
      p1_y = vecy(slot_min);
      p1_x = vecx(slot_min);

      % Agora tenho p1 e p2, tenho que agora fazer a
      % interpolacao linear para achar o p3
      % equacao da reta: y = ax + b
      a = (p2_y - p1_y)/(p2_x - p1_x);
      b = p1_y - a*p1_x;
      p3_x = (point_y - b)/a;

      % e calcular a distancia em x do ponto que eu to para o x de p3
      distances_points = [square_x(1):square_x(2)] - p3_x;
      distances_points = abs(distances_points);
      point_x = find(distances_points == min(distances_points));
      vecloc(point_y, point_x) = 1;
      q(point_y, point_x) = min(distances_points);
    end


  else
```

```
    q = 0; vecloc = 0;
  end
```

code/funcao.m

Também segue o script para o teste da função criada:

```
% script de teste da funcao 'funcao'
close all;
clear('all');

i = 1;
vecx = [1:40] + 0.3;
vecy = 20*sin([1:40] + 0.3) + 21;
%vecy = ([1:40] + 0.3);
Nr = 50;
Mc = 50;
[q, vecloc] = funcao(i, vecx, vecy, Nr, Mc);

figure;
h1 = axes;
imagesc(vecloc);
hold on;
plot(vecx, vecy, 'black');
grid on;
set(h1, 'Ydir', 'normal');
legend('Parede Nao-Alinhada');

figure;
h1 = axes;
imagesc(q);
hold on;
plot(vecx, vecy, 'black');
grid on;
set(h1, 'Ydir', 'normal');
legend('Parede Nao-Alinhada');
```

code/teste_funcao.m

Para o cálculo do quadrado mínimo que abrange a barreira não-alinhada foi adiquirido tais resultados:

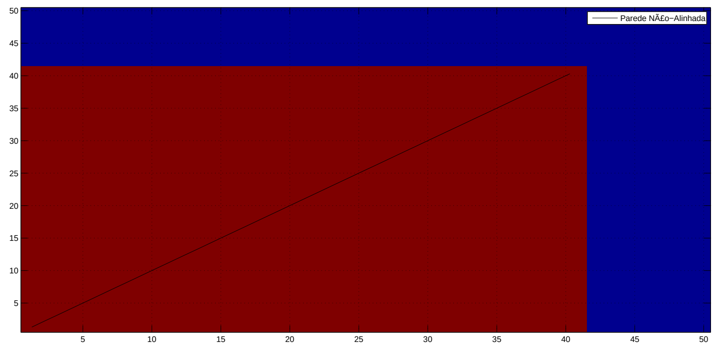

Figura 4: Barreira não-alinhada reta e deslocada em 0.3.

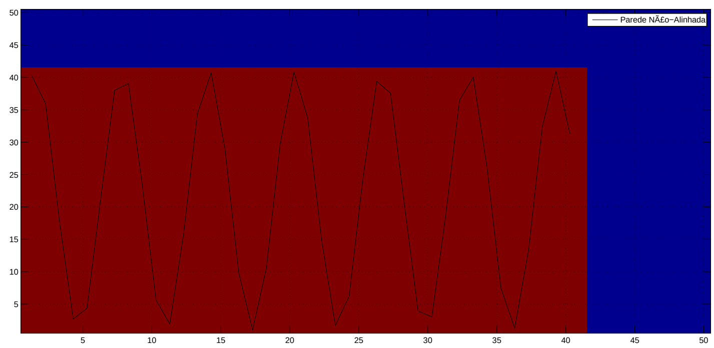

Figura 5: Barreira não-alinhada senoidal.

Para o cálculo das células que atravessam a barreira pela direita segue os resultados:

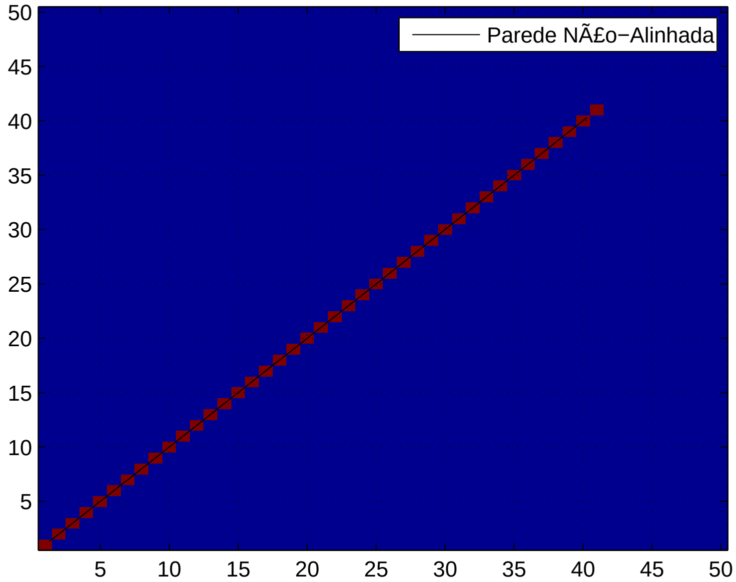

Figura 6: Células de Lattice que cruzam barreira não-alinhada reta e deslocada em 0.3.

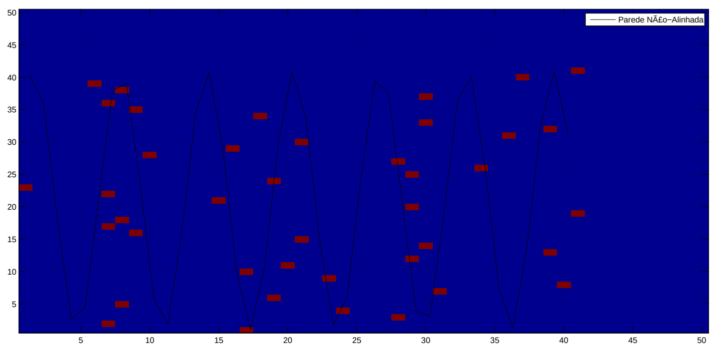

Figura 7: Células de Lattice que cruzam em barreira não-alinhada senoidal.

Referências