```
import numpy as np

'''
Arquivo: openBoundaries.py

Esse programa le o arquivo data.npy que estive na mesma pasta que ele e resolve
a equacao da onda em um meio bidimensional nao-homogeneo
(com condicoes de contorno "absorventes" - o dominio e expandido alem do campo
de visao) para os parametros passados pelo arquivo data.npy pelo metodo de
diferencas finitas

Apos resolver a equacao, o arquivo salva os seguintes arquivos binarios:
    U.npy - Malha que representa a propagacao da onda 2D ao longo do tempo
    X.npy - Eixo x usado para os calculos que geraram a malha de U.npy
    Y.npy - Eixo y usado para os calculos que geraram a malha de U.npy
    V.npy - Velocidades usadas durante o calculo da malha de U.npy
'''

class interface(object):
    '''
        Herda: object
        Define uma interface na forma de uma reta
    '''

    def __init__(self, a, b):
        '''
            Definicao de construtor
            Recebe:     a - coeficiente angular
                        b - termo independente
        '''
        self.a = a
        self.b = b

    def __call__(self, x):
        '''
            Definicao de funcao
            Recebe :    x - valor nas abscissas
            Retorna:    y - valor nas ordenadas para o valor x
        '''
        return self.a * x + self.b

class velocity(object):
    '''
        Herda: object
        Define uma velocidade como uma funcao quadratica
    '''
    def __init__(self, a = 0., b = 0., c = 1.1):
        '''
            Definicao de construtor
            Recebe:     a - Termo 'a' da funcao quadratica
                        b - Termo 'b' da funcao quadratica
                        c - Termo 'c' da funcao quadratica
        '''
        self.a    = a
        self.b    = b
        self.c    = c

    def getGradientVelocity(self, x, y):
        '''
            Definicao de funcao
            Recebe:     x - valor nas abscissas
                        y - valor nas ordenadas
            Retorna:    y - valor da velocidade em (x, y)
        '''
        return self.a * x + self.b * y + self.c

    def __call__(self, x, y):
        '''
```

```
            Uma funcao call para uma classe permite que um objeto desta
            seja chamado como uma funcao. No caso de um objeto velocity ser
            chamado, ele retornara a velocidade v_type(x, y), sendo type o tipo
            de velocidade a ser retornada e derv a derivada da velocidade
        '''
        return self.getGradientVelocity(x, y)

# Criando a classe que define a onda 2D
class wave2D(object):
    '''
        Herda: object
        Define uma onda (por meio de algumas propriedades desta) em um ambiente
        bidimensional heterogeneo
    '''

    def __init__(self, Lx, Ly, tMax, Mx, Ny, w, A, Xp, Yp, Tp):
        '''
            Define um de ondas bidimensionais
            Recebe:     Lx - Comprimento do dominio em relacao ao eixo x
                        Ly - Comprimento do dominio em relacao ao eixo y
                        tMax - Tempo maximo para a propagacao da onda
                        Mx - Numero de pontos no eixo x
                        Ny - Numero de pontos no eixo y
                        w  - Frequencia dominante da onda
                        A  - Amplitude da onda
                        Xp - Posicao em x do pico do pulso da fonte
                        Yp - Posicao em y do pico do pulso da fonte
                        Tp - Tempo do pico do pulso da fonte
        '''

        self.Lx   = 3 * Lx
        self.Ly   = 3 * Ly
        self.tMax = tMax
        self.Mx   = 3 * Mx
        self.Ny   = 3 * Ny
        self.w    = w
        self.A    = A
        self.Xp   = Xp
        self.Yp   = Yp
        self.Tp   = Tp
        self.dx   = float(self.Lx) / float (Mx - 1) # Intervalo em x
        self.dy   = self.dx # Intervalo em y
        self.dt   = self.dy / 2.0 # Intervalo no tempo
        # Numero de pontos no tempo
        self.Ot   = int(np.ceil(self.tMax / self.dt)) + 1
        self.R    = np.power(np.pi, 2) * np.power(self.w, 2)

    def evaluateFXYT(self, X, Y, T):
        '''
            Funcao define a fonte da equacao da onda para os valores de
            X, Y e T. Atualmente, a fonte esta definida como uma wavelet Ricker
            Recebe:     X - array de valores no eixo das abscissas
                        Y - array de valores no eixo das ordenadas
                        T - array de valores no eixo temporal
            Retorna:    a funcao da wavelet Ricker
        '''
        termoT = self.R * np.power(T-self.Tp, 2)

        D = np.power(X-self.Xp,2) + np.power(Y-self.Yp,2)

        termoD = self.R * D

        return self.A * np.exp(-termoT) * ((1 - 2 * termoD) * np.exp(-termoD))

    def getVelocityMatrix(self, interfaces, velocidades, X, Y):
        # TODO: Documentar
        # Criando matriz de velocidades
        velocities = np.zeros((int(self.Mx), int(self.Ny)))
```

```
        # Preenchendo a matriz
        k = 0
        # TODO: Comentar hard!!!
        for i in range(0, int(self.Mx - 1)):
            # Definindo posicao x
            x = X[i]
            for j in range(0, int(self.Ny - 1)):
                # Definindo posicao y
                y = Y[j]
                if x >= 0. and x <= self.Lx / 3:
                     if y >= 0. and y <= self.Ly / 3:
                         while y > interfaces[k](x) and k < len(interfaces) - 1:
                             k += 1
                         velocities[i, j] = velocidades[k](x, y)
                else:
                    while y > interfaces[k].b and k < len(interfaces) - 1:
                        k += 1
                    velocities[i, j] = velocidades[k](x, y)
                # Para garantir que nao haja divisao por zero
                if velocities[i, j] >= -.00005 and velocities[i, j] <= .00005:
                    media = np.mean(velocities[i, :j + 2])
                    velocities[i, j] = 1.
                k = 0

        return velocities

# Carregando os dados do arquivo
data = np.load('data.npy')

# Criando objeto do tipo wave2D
#                  Lx      Ly       tMax     Mx       Ny       w
Onda2D = wave2D(data[0], data[1], data[2], data[3], data[4], data[5],
#     A        Xp       Yp       Tp
    data[6], data[7], data[8], data[9])

# Criando as velocidades e interfaces das camadas
# TODO: Esses dados devem ser provinientes do introdutor.py
l0 = velocity(0., .1, 1.1)
A  = interface( 0.03, 1.5)
l1 = velocity(0., .2, 2.2)
B  = interface(-0.05, 3. )
l2 = velocity(0., .5, 2.2)
C  = interface(-0.01, 4. )
l3 = velocity(0., .3, 2.1)
D  = interface( 0.1 , 6.5)
l4 = velocity(0., .2, 2.6)
# l0 = velocity(3., 1.1, 1.1)
# A  = interface( 0.03, 1.5)
# l1 = velocity(2., 2.8, 1.2)
# B  = interface(-0.05, 3. )
# l2 = velocity(5., 4.5, 1.2)
# C  = interface(-0.01, 4. )
# l3 = velocity(4., 3., 1.1)
# D  = interface( 0.1 , 6.5)
# l4 = velocity(10., 13.5, 1.3)

# Criando listas de interfaces e velocidades
interfaces  = [ A,  B,  C,  D    ]
velocidades = [l0, l1, l2, l3, l4]

# Criando vetores X, Y, T
X = np.linspace(-Onda2D.Lx / 3, 2 * Onda2D.Lx / 3, Onda2D.Mx)
Y = np.linspace(-Onda2D.Ly / 3, 2 * Onda2D.Ly / 3, Onda2D.Ny)
T = np.linspace(0., Onda2D.tMax, Onda2D.Ot)

# Recebendo matriz de velocidades
velocidades = Onda2D.getVelocityMatrix(interfaces, velocidades, X, Y)
```

```
# Criando array de Mx * Ny * Ot pontos
U = np.zeros((int(Onda2D.Mx), int(Onda2D.Ny), int(Onda2D.Ot)))

# Aplicando condicoes iniciais U = 0. e dt(U) = 0.
U[:, :, 0:2] = 0.

# Aplicando condicoes de fronteira
U[                  :,                   0, :] = 0.
U[int(Onda2D.Mx - 1),                    :, :] = 0.
U[                  :, int(Onda2D.Ny) - 1, :] = 0.
U[                  0,                    :, :] = 0.

# Indices para os pontos interiores
i = np.arange(1, int(Onda2D.Mx - 1))
j = np.arange(1, int(Onda2D.Ny - 1))

[ii, jj] = np.meshgrid(i, j)

# Criando dx^2
dx2 = Onda2D.dx * Onda2D.dx

np.savetxt('vel', velocidades)

# Aplicando o MDF
d = 2 * velocidades[ii, jj]
c = 1 / (d * d)
for k in range(1, Onda2D.Ot - 1):
    U[ii, jj, k + 1] = c * (Onda2D.evaluateFXYT(X[ii], Y[jj], T[k]) - \
    4. * U[ii, jj, k] + U[ii - 1, jj, k] + U[ii + 1, jj, k] + \
    U[ii, jj - 1, k] + U[ii, jj + 1, k]) - U[ii, jj, k - 1] + 2. * U[ii, jj, k]

# Salvando arrays (um em cada arquivo, exceto o T, para evitar confusao)
np.save('data/X', X)
np.save('data/Y', Y)
np.save('data/U', U)
np.save('data/V', velocidades)
```

```
import numpy as np

'''
Arquivo: reflect.py

Esse programa le o arquivo data.npy que estive na mesma pasta que ele e resolve
a equacao da onda em um meio bidimensional nao-homogeneo
(com condicoes de contorno refletoras) para os parametros
passados pelo arquivo data.npy pelo metodo de diferencas finitas

Apos resolver a equacao, o arquivo salva os seguintes arquivos binarios:
    U.npy - Malha que representa a propagacao da onda 2D ao longo do tempo
    X.npy - Eixo x usado para os calculos que geraram a malha de U.npy
    Y.npy - Eixo y usado para os calculos que geraram a malha de U.npy
    V.npy - Velocidades usadas durante o calculo da malha de U.npy
'''

class interface(object):
    '''
        Herda: object
        Define uma interface na forma de uma reta
    '''

    def __init__(self, a, b):
        '''
            Definicao de construtor
            Recebe:     a - coeficiente angular
                        b - termo independente
        '''
        self.a = a
        self.b = b

    def __call__(self, x):
        '''
            Definicao de funcao
            Recebe :    x - valor nas abscissas
            Retorna:    y - valor nas ordenadas para o valor x
        '''
        return self.a * x + self.b

class velocity(object):
    '''
        Herda: object
        Define uma velocidade como uma funcao quadratica
    '''
    def __init__(self, a = 0., b = 0., c = 1.1):
        '''
            Definicao de construtor
            Recebe:     a - Termo 'a' da funcao quadratica
                        b - Termo 'b' da funcao quadratica
                        c - Termo 'c' da funcao quadratica
        '''
        self.a    = a
        self.b    = b
        self.c    = c

    def getGradientVelocity(self, x, y):
        '''
            Definicao de funcao
            Recebe:     x - valor nas abscissas
                        y - valor nas ordenadas
            Retorna:    y - valor da velocidade em (x, y)
        '''
        return self.a * x + self.b * y + self.c

    def __call__(self, x, y):
        '''
            Uma funcao call para uma classe permite que um objeto desta
```

```
            seja chamado como uma funcao. No caso de um objeto velocity ser
            chamado, ele retornara a velocidade v_type(x, y), sendo type o tipo
            de velocidade a ser retornada e derv a derivada da velocidade
        '''
        return self.getGradientVelocity(x, y)

# Criando a classe que define a onda 2D
class wave2D(object):
    '''
        Herda: object
        Define uma onda (por meio de algumas propriedades desta) em um ambiente
        bidimensional heterogeneo
    '''

    def __init__(self, Lx, Ly, tMax, Mx, Ny, w, A, Xp, Yp, Tp):
        '''
            Define um de ondas bidimensionais
            Recebe:     Lx - Comprimento do dominio em relacao ao eixo x
                        Ly - Comprimento do dominio em relacao ao eixo y
                        tMax - Tempo maximo para a propagacao da onda
                        Mx - Numero de pontos no eixo x
                        Ny - Numero de pontos no eixo y
                        w  - Frequencia dominante da onda
                        A  - Amplitude da onda
                        Xp - Posicao em x do pico do pulso da fonte
                        Yp - Posicao em y do pico do pulso da fonte
                        Tp - Tempo do pico do pulso da fonte
        '''
        self.Lx   = Lx
        self.Ly   = Ly
        self.tMax = tMax
        self.Mx   = Mx
        self.Ny   = Ny
        self.w    = w
        self.A    = A
        self.Xp   = Xp
        self.Yp   = Yp
        self.Tp   = Tp
        self.dx   = float(self.Lx) / float (Mx - 1) # Intervalo em x
        self.dy   = self.dx # Intervalo em y
        self.dt   = self.dy / 2.0 # Intervalo no tempo
        # Numero de pontos no tempo
        self.Ot   = int(np.ceil(self.tMax / self.dt)) + 1
        self.R    = np.power(np.pi, 2) * np.power(self.w, 2)

    def evaluateFXYT(self, X, Y, T):
        '''
            Funcao define a fonte da equacao da onda para os valores de
            X, Y e T. Atualmente, a fonte esta definida como uma wavelet Ricker
            Recebe:     X - array de valores no eixo das abscissas
                        Y - array de valores no eixo das ordenadas
                        T - array de valores no eixo temporal
            Retorna:    a funcao da wavelet Ricker
        '''
        termoT = self.R * np.power(T-self.Tp, 2)

        D = np.power(X-self.Xp,2) + np.power(Y-self.Yp,2)

        termoD = self.R * D

        return self.A * np.exp(-termoT) * ((1 - 2 * termoD) * np.exp(-termoD))

    def getVelocityMatrix(self, interfaces, velocidades):
        '''
            Funcao que retorna um array bidimensional de velocidades. Para cada
            ponto do meio em que a onda se propaga calcula-se uma velocidade,
            dependendo de qual camada o ponto se encontra.
            Recebe:     interfaces  - lista de interfaces (se encontram de uma
```

```
                                    camada para outra)
                        velocidades - lista de objetos velocidade, cujos itens
                                      sao as funcoes velocidade de cada camada
                                      do meio
        '''
        # Criando matriz de velocidades
        velocities = np.zeros((int(self.Mx), int(self.Ny)))

        # Preenchendo a matriz
        k = 0
        for i in range(0, int(self.Mx - 1)):
            x = i * self.dx     # Dando um passo nas abscissas
            for j in range(0, int(self.Ny - 1)):
                y = j * self.dy     # Dando um passo nas ordenadas
                while y > interfaces[k](x) and k < len(interfaces) - 1:
                    k += 1      # Determinando em que camada o ponto se encontra
                # Calculando a velocidade naquele ponto
                velocities[i, j] = velocidades[k](x, y)
                k = 0

        return velocities

# Carregando os dados do arquivo
data = np.load('data.npy')

# Criando objeto do tipo wave2D
#                  Lx      Ly       tMax     Mx       Ny       w
Onda2D = wave2D(data[0], data[1], data[2], data[3], data[4], data[5],
#     A        Xp       Yp       Tp
    data[6], data[7], data[8], data[9])

# Criando as velocidades e interfaces das camadas
# TODO: Esses dados devem ser provinientes do introdutor.py
l0 = velocity(0., .1, 1.1)
A  = interface( 0.03, 1.5)
l1 = velocity(0., .2, 2.2)
B  = interface(-0.05, 3. )
l2 = velocity(0., .5, 2.2)
C  = interface(-0.01, 4. )
l3 = velocity(0., .3, 2.1)
D  = interface( 0.1 , 6.5)
l4 = velocity(0., .2, 2.6)
# l0 = velocity(3., 1.1, 1.1)
# A  = interface( 0.03, 1.5)
# l1 = velocity(2., 2.8, 1.2)
# B  = interface(-0.05, 3. )
# l2 = velocity(5., 4.5, 1.2)
# C  = interface(-0.01, 4. )
# l3 = velocity(4., 3., 1.1)
# D  = interface( 0.1 , 6.5)
# l4 = velocity(10., 13.5, 1.3)

# Criando listas de interfaces e velocidades
interfaces  = [ A,  B,  C,  D    ]
velocidades = [l0, l1, l2, l3, l4]

# Recebendo matriz de velocidades
velocidades = Onda2D.getVelocityMatrix(interfaces, velocidades)

# Criando array de Mx * Ny * Ot pontos
U = np.zeros((int(Onda2D.Mx), int(Onda2D.Ny), int(Onda2D.Ot)))

# Criando vetores X, Y, T
X = np.linspace(0., Onda2D.Lx  , Onda2D.Mx)
Y = np.linspace(0., Onda2D.Ly  , Onda2D.Ny)
T = np.linspace(0., Onda2D.tMax, Onda2D.Ot)

# Aplicando condicoes iniciais U = 0. e dt(U) = 0.
```

```
U[:, :, 0:2] = 0.

# Aplicando condicoes de fronteira
U[                  :,                    0, :] = 0.
U[int(Onda2D.Mx - 1),                     :, :] = 0.
U[                  :, int(Onda2D.Ny) - 1, :] = 0.
U[                  0,                    :, :] = 0.

# Indices para os pontos interiores
i = np.arange(1, int(Onda2D.Mx - 1))
j = np.arange(1, int(Onda2D.Ny - 1))

[ii, jj] = np.meshgrid(i, j)

# Criando dx^2
dx2 = Onda2D.dx * Onda2D.dx

# Aplicando o MDF
d = 2 * velocidades[ii, jj]
c = 1 / (d * d)
for k in range(1, Onda2D.Ot - 1):
    U[ii, jj, k + 1] = c * (Onda2D.evaluateFXYT(X[ii], Y[jj], T[k]) - \
    4. * U[ii, jj, k] + U[ii - 1, jj, k] + U[ii + 1, jj, k] + \
    U[ii, jj - 1, k] + U[ii, jj + 1, k]) - U[ii, jj, k - 1] + 2. * U[ii, jj, k]

# Salvando arrays (um em cada arquivo, exceto o T, para evitar confusao)
np.save('data/X', X)
np.save('data/Y', Y)
np.save('data/U', U)
np.save('data/V', velocidades)
```

```
#!/usr/bin/python2.7
#!-*- coding: utf8 -*-

import numpy as np
import matplotlib.pyplot as plt

# Carregando arrays a partir de arquivos
X = np.load('data/X.npy')
Y = np.load('data/Y.npy')
U = np.load('data/U.npy')

print "Quantos snapshots voce deseja?"
N = input()

# Definindo passo
h = U.shape[2] / N

counter = 0

# Criando figura
fig = plt.figure()

# Adicionando eixos
fig.add_axes()

# Criando eixo para plotagem
ax = fig.add_subplot(111)

# Formando base para o plot (?)
[Y, X] = np.meshgrid(Y, X)

markers = np.array([(0., 0.), (1.5, 1.9), (3., 2.3), (4., 3.8), (6.5, 8.)], dtype=(float, 2
))

# Invertendo o eixo y
plt.gca().invert_yaxis()

# Cria as imagens de N em N quadros
for i in range(h, U.shape[2], h):

    # Buscando o maior valor de U para fixar o eixo em z
    M = max(abs(U.min()), abs(U.max()))

    # Criando plot
    ax.contourf(X, Y, U[:,:,i], 20, cmap=plt.cm.seismic, vmin=-M, vmax=M )

    # Plotando as Camadas
    for i in range(0, markers.size / 2):
        # TODO: Trocar o 15. por uma variavel passada por parametro
        ax.plot((0., 15.), (markers[i][0], markers[i][1]), '-k')

    # Configurando o titulo do grafico e suas legendas
    ax.set(title='Onda em 2D', ylabel='Y', xlabel='X')

    # Definindo titulo da plotagem
    titulo = "Teste 0%d - MDF - 1D" % counter

    # Definindo caminho da plotagem
    caminho = 'images/Teste0%d.png' % counter

    # Incrementando o contador
    counter += 1

    # Salvando a imagem
    plt.savefig(caminho)
```

```
#!/usr/bin/python2.7
#!-*- coding: utf8 -*-

import numpy as np

print "Seja bem-vindo ao Wave Plotter 2000!"

print "Insira o tamanho em x: "
Lx = input()

print "Insira o tamanho em y: "
Ly = input()

print "Insira o tempo maximo de animacao: "
tMax = input()

print "Insira o numero de pontos em x: "
Mx = input()

print "Insira o numero de pontos em y: "
Ny = input()

print "Insira o alpha: "
alpha = input()

print "Insira o omega: "
w = input()

print "Insira a Amplitude da onda: "
A = input()

print "Insira em a posicao em x do pico do pulso da onda: "
Xp = input()

print "Insira em a posicao em y do pico do pulso da onda: "
Yp = input()

print "Insira o tempo de pico do pulso da fonte: "
Tp = input()

# Criando um array com todos os valores recebidos
a = np.array([Lx, Ly, tMax, Mx, Ny, w, A, Xp, Yp, Tp])

# Salvando dados em um arquivo
np.save('data', a)
```

```
#!/usr/bin/python2.7
#!-*- coding: utf8 -*-

import numpy as np
import matplotlib.pyplot as plt

# Carregando arrays a partir de arquivos
X = np.load('data/X.npy')
Y = np.load('data/Y.npy')
V = np.load('data/V.npy')

# Criando figura
fig = plt.figure()

# Adicionando eixos
fig.add_axes()

# Criando eixo para plotagem
ax = fig.add_subplot(111)

# Formando base para o plot (?)
[Y, X] = np.meshgrid(Y,X)

markers = np.array([(0., 0.), (1.5, 1.9), (3., 2.3), (4., 3.8), (6.5, 8.)], dtype=(float, 2
))

# Invertendo o eixo y
plt.gca().invert_yaxis()

# Buscando o maior valor de U para fixar o eixo em z
M = max(abs(V.min()), abs(V.max()))

# Criando plot
plot = ax.contourf(X, Y, V, 20, cmap=plt.cm.seismic, vmin=-M, vmax=M)

# Desenhando a barra de cores
plt.colorbar(plot)

# Plotando as Camadas
for i in range(0, markers.size / 2):
    # TODO: Trocar o 15. por uma variavel passada por parametro
    ax.plot((0., 15.), (markers[i][0], markers[i][1]), '-k')

# Configurando o titulo do grafico e suas legendas
ax.set(title='Velocidades', ylabel='Y', xlabel='X')

# Definindo caminho da plotagem
caminho = 'images/Velocidades.png'

# Salvando a imagem
plt.savefig(caminho)
```