```
'''
    Arquivo: classesRT.py
    Reune as classes utilizadas no tracamento de raios, definindo os elementos
    que compoem o meio onde os raios serao tracados, definindo tambem os
    proprios raios.
'''

from eqDiferencialOrdinaria import eqDiferencialOrdinaria as EDO
import numpy as np

class ray(EDO):
    '''
        Herda: EDO
        Define, a partir da teoria matematica, o que e um raio, armazenando
        em arrays os seus pontos e direcoes ao longo do tempo.
    '''
    def __init__(self, dimension, XP, time = 0.):
        '''
            Definicao de construtor
            Recebe:         dimension - numero de EDOs que definem o raio (4)
                            XP - array de posicoes/direcoes ao longo do tempo
                            time - array que guarda o tempo de transito do raio
        '''
        super(ray, self).__init__(dimension)
        self.XP             = XP
        self.time           = np.array([0.])

    def evaluate(self, Y, v):
        '''
            Definicao de funcao
            Avalia as quatro equacoes que definem o raio usando os pontos e
            direcoes passadas em Y e o modelo de velocidade definido em v.
            Recebe:         Y - ultimos valores calculados pelo RK4 para a
                                posicao e a direcao do raio em questao
                            v - objeto velocidade que, quando chamado (utilizan-
                                do a funcao __call__) retorna a velocidade no
                                ponto (x, y) passado por parametro
            Retorna:        retorno - array contendo os valores calculados pelas
                                      equacoes do raio
        '''
        retorno = np.zeros(4) # Array que servira de retorno
        Vxy     = v(Y[0], Y[1], "0")
        iVxy    = 1. / Vxy
        dVx     = v(Y[0], Y[1], "1x")
        dVy     = v(Y[0], Y[1], "1y")

        retorno[0] = Vxy * Vxy * Y[2] #  dx/dt
        retorno[1] = Vxy * Vxy * Y[3] #  dy/dt
        retorno[2] = iVxy * dVx       # dPx/dt
        retorno[3] = iVxy * dVy       # dPy/dt

        return retorno

class source(object):
    '''
        Herda: object
        Define a fonte de raios, que os cria de acordo com os parametros que re-
        cebeu.
    '''
    def __init__(self, posY, angMin, angMax, nRays, initialVelocity):
        '''
            Definicao de um construtor
            Recebe:         posY - Posicao de onde partirao os raios
                            angMin - Inicio do intervalo que sera iluminado
                                     pelos raios
                            angMax - Fim desse intervalo
                            nRays - Numero de raios a serem tracados
                            initialVelocity - velocidade no ponto de partida dos
```

```
                                        raios
        '''
        self.posY            = posY
        self.angMin          = angMin
        self.angMax          = angMax
        self.nRays           = nRays
        self.initialVelocity = initialVelocity
        self.genRays()

    def genRays(self):
        '''
            Procedimento que cria os raios de acordo com os parametros da fonte.
        '''
        # Determinando 'passo angular'
        h = (self.angMax - self.angMin) / (self.nRays - 1)

        # Criando lista vazia
        self.rays = []

        for i in range(0, self.nRays):
            # Determinando angulo do raio
            ang = self.angMin + i * h
            # Criando vetor posicao-direcao do raio
            Y = np.zeros((4, 2))
            Y[:, 0] = np.nan    # Para permitir o anexamento no metodo 'go'
            Y[0, 1] = Y[1, 1] = 0. # Posicao inicial
            Y[2, 1] = np.sin(ang) / self.initialVelocity # Direcao inicial
            Y[3, 1] = np.cos(ang) / self.initialVelocity

            # Criando raio auxiliar
            Aux = ray(4, Y)

            # Colocando raio na lista
            self.rays.append(Aux)

class velocity(object):
    '''
        Herda: object
        Define a velocidade como uma funcao quadratica com coeficientes a, b e c
        bem como as derivadas dessa funcao
    '''
    def __init__(self, type, a = 0., b = 0., c = 1.):
        '''
            Definicao de um construtor
            Recebe:         type - tipo de velocidade (neste trabalho foi defi-
                                   nido apenas o modelo quadratico de veloci-
                                   dade, entretanto, essa variavel permite que
                                   mais modelos sejam implementados)
                            a, b, c - coeficientes da funcao quadratica que
                                      define a velocidade
        '''
        self.type = type
        self.a    = a
        self.b    = b
        self.c    = c

    def getGradientVelocity(self, x, y, derv = "0"):
        '''
            Funcao que retorna a velocidade (ou uma de suas derivadas) em um da-
            do ponto (x, y)
            Recebe          x - coordenada x do ponto em que se deseja calcular
                                a velocidade
                            y - coordenada y do mesmo ponto
                            derv - a derivada desejada
            Retorna         a velocidade no ponto (x, y) ou a derivada desejada
                            da funcao
        '''
```

```
        return {
            "0"  : self.a * x + self.b * y + self.c,
            "1x" : self.a,
            "1y" : self.b
        }.get(derv, "0")

    def __call__(self, x, y, derv = "0"):
        '''
            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 {
            '0' : self.getGradientVelocity(x, y, derv)
        }.get(self.type, '0')

class interface(object):
    '''
        Herda: object
        Responsavel por permitir a interpretacao de uma interface como uma reta,
        na sua forma parametrica, ou seja, com um vetor diretor e um ponto por
        onde ela passa. Alem disso, sao definidos os seus pontos extremos (pon-
        tos laterais).
    '''

    def __init__(self, diretor, lateralPoints):
        '''
            Definicao de construtor
            Recebe:         diretor - vetor diretor da interface
                            lateralPoints - pontos extremos da interface
        '''
        self.vDiretor = diretor
        self.vNormal  = np.array([-diretor[1], diretor[0]])
        self.a        = diretor[1] / diretor[0]
        self.b        = lateralPoints[0]
        self.lP       = lateralPoints

    def __call__(self, x):
        '''
            Funcao que possibilita que, ao se chamar um objeto da classe inter-
            face, passando-se um valor na coordenada x para ele, se obtenha o
            valor y correspondente, como em uma reta.
            Recebe:         x - coordenada nas abscissas
            Retorna:        y - valor correspondente a x nas ordenadas
        '''
        return self.a * x + self.b

class layer(object):
    '''
        Herda: object
        Define como e interpretada uma camada. A camada e interpretada como
        possuindo uma interface superior e uma interface inferior. Entre ambas
        as interfaces, ha uma parcela do meio que possui uma determinada velo-
        cidade.
    '''
    def __init__(self, mediumsDimension, lateralPoints, velocity):
        '''
            Definicao de construtor
            Recebe:         mediumsDimension - dimensao do meio
                            lateralPoints - pontos extremos da interface supe-
                                            rior
                            velocity - modelo de velocidade para a camada
        '''
        self.mediumsDimension = mediumsDimension
        self.lateralPoints    = lateralPoints
        self.velocity         = velocity
        self.makeDirector()
```

```
        self.supInt = interface(self.director, lateralPoints)

    def makeDirector(self):
        '''
            Metodo responsavel por criar um vetor diretor para a camada (e,
            consequentemente, para sua interface superior)
        '''
        # Criando vetor paralelo a interface
        self.director = np.array([self.mediumsDimension[0],
                        self.lateralPoints[1] - self.lateralPoints[0]])
        # Calculando a norma do vetor
        directorNorm = np.linalg.norm(self.director)
        # Normalizando o vetor
        self.director /= directorNorm

    def setInfInt(self, Int):
        '''
            Metodo responsavel por setar a interface inferior da camada
        '''
        self.infInt = Int

class medium(object):
    '''
        Herda: object
        Define o meio por meio dos seus principais elementos
    '''
    def __init__(self, dimension, s0, layers):
        '''
            Definicao de construtor
            Recebe:         dimension - dimensoes do meio
                            s0 - fonte que dispara raios no meio
                            layers - camadas (e, consequentemente, interfaces do
                                     meio)
        '''
        self.dimension  = dimension
        self.s0         = s0
        self.layers     = layers
```

```
#!-*- coding: utf8 -*-

from abc import ABCMeta, abstractmethod

class eqDiferencialOrdinaria(object):
    '''
        Herda: object
        Define uma equacao diferencial ordinaria ou um sistema de equacoes desse
        tipo (dependendo do valor da variavel dimension)
    '''
    def __init__(self, dimension, k = 0, a = 0, r = 0, b = 0):
        '''
            Definicao de construtor
            Recebe:         dimension - Numero de EDOs envolvidos no sistema
                            k, a, r, b - constantes para o caso do sistema de
                                         EDOs ser do tipo Lotka-Volterra, ou
                                         algo parecido
        '''
        self.dimension = dimension
        self.k         = k
        self.a         = a
        self.r         = r
        self.b         = b

    @abstractmethod
    def evaluate(Y, v):
        '''
            Funcao que, usando os valores das EDOs contidas em Y, calcula as
            proximos valores das EDOs utilizando as formulas das mesmas. v e
            utilizada na realizacao dos calculos.
            Retorna:        Y - Valores das EDOs
                            v - objeto velocity usado nos calculos
            Retorna:        array com os proximos valores das EDOs
        '''
        pass
```

```
class criticalAngle(Exception):
    '''
        Herda: Exception
        Define a excecao de angulo critico
    '''
    def __init__(self, ray, layer, actualAngle, criticAngle):
        '''
            Definicao de construtor
            Recebe:         ray - numero do raio com que ocorreu a excecao
                            layer - numero da camada com a qual ocorreu a
                                    excecao
                            actualAngle - angulo do raio ray
                            criticAngle - angulo critico da camada layer
        '''
        self.ray         = ray
        self.layer       = layer
        self.actualAngle = actualAngle
        self.criticAngle = criticAngle
        super(criticalAngle, self).__init__(self.getMsg())

    def getMsg(self):
        '''
            Funcao que retorna a mensagem exibida pela excecao
        '''
        return "O atual angulo do raio ", self.ray, " (", self.actualAngle, ") e \
        maior que o angulo critico da camada ", self.layer, " (" \
        , self.criticAngle, ")"

class singularMatrix(Exception):
    '''
        Classe que define a excecao de existencia de matriz singular para um
        determinado raio numa determinada camada
    '''
    def __init__(self, A):
        print "A seguinte matriz e singular: "
        print A
```

```
from utilRT     import projVetorial
from rungeKutta import rungeKutta4Ordem as RK4
import numpy                            as np

def refract(i, v1, v2, r, s):
    '''
        Procedimento que realiza a refracao do raio
        i   - objeto interface
        v1  - objeto velocidade da camada 1
        v2  - objeto velocidade da camada 2
        r   - objeto ray
        s   - sentido do raio (descendo ou subindo)
    '''
    # Vetor direcao atual do raio
    XY = np.array([r.XP[0, -1], r.XP[1, -1]])
    P  = np.array(r.XP[2, -1], r.XP[3, -1])

    # Preparando as velocidades a serem utilizadas
    v1_ponto = v1(XY[0], XY[1], "0")
    v2_ponto = v2(XY[0], XY[1], "0")
    sinTheta1 = np.linalg.norm(projVetorial(P, i.vDiretor))

    # Aplicando a lei de fato
    sinTheta2 = v2_ponto / v1_ponto * sinTheta1

    # Obtendo o cosTheta2 e as novas componentes do raio
    cosTheta2 = np.sqrt(1 - sinTheta2 * sinTheta2)

    # Definindo o novo vetor para o raio
    if s > 0:   # Descendo
        P = cosTheta2 * i.vNormal + sinTheta2 * i.vDiretor
    else:   # Subindo
        P = cosTheta2 * -i.vNormal + sinTheta2 * i.vDiretor

    # Colocando nova direcao para o raio
    r.XP[2, -1], r.XP[3, -1] = P

def reflect(i, r):
    '''
        Procedimento que realiza a reflexao do raio
        i - objeto interface
        r - objeto ray
    '''
    P        = np.array([r.XP[2, -1], r.XP[3, -1]])
    S        = projVetorial(P, i.vNormal)
    r.XP[2, -1], r.XP[3, -1] = P - 2 * S

def go(v, i, r, s, dimX, tMax):
    '''
        Procedimento que calcula o prosseguimento do raio
        v    - objeto velocity
        i    - objeto interface
        r    - objeto ray
        s    - sentido do raio (descendo ou subindo)
        dimX - dimensao em X do meio
        tMax - tempo maximo que um raio pode permanecer em uma camada
    '''
    # Usando o rungeKutta
    # O tempo maximo deve ser o tempo atual + tMax
    tMax     = r.time[-1] + tMax
    h        = .01      # Passo do RK4
    To       = r.time[-1]   # Tempo inicial
    Yo       = r.XP[:, -1]  # Valores iniciais das EDOs
    paramRK  = [v, i, dimX, s]  # Parametros do RK4
    XP, time = RK4(r, tMax, h, To, Yo, paramRK)

    # Anexando os arrays
    r.XP   = np.append(r.XP  , XP  , axis=1)
```

```
    r.time = np.append(r.time, time        )
```

```
'''
    Arquivo: RTscript.py
    Programa central do tracamento de raios. Recebe os parametros do tracamento
    pela entrada do usuario e aplica as acoes necessarias para que o tracamento
    ocorra, utilizando os demais arquivos como acessorios
'''

from classesRT import ray, source, layer, medium
from methodsRT import refract, reflect, go
from exceptionsRT import criticalAngle, singularMatrix
from utilRT import (buildMedium, degreesToRadians, radiansToDegrees, plot,
userInput)
import numpy as np

(dimension, posY, nLayers, nRays, angMin, angMax, lateralPoints,
velocidades, tMax, refletora) = userInput()
angMin = degreesToRadians(angMin)
angMax = degreesToRadians(angMax)

# Construindo um meio
medium = buildMedium(dimension, posY, nLayers, nRays, angMin, angMax,
            lateralPoints, velocidades)

# Tracando raios
u = 0   # Para indexacao do array M
for i in range(0, 1):
    for j in range(0, nRays):
        k = 0
        # Descendo
        while True:
            # Tracando o raio
            go(medium.layers[k].velocity, medium.layers[k + 1].supInt,
                medium.s0.rays[j], 1, dimension[0], tMax)
            # Caso SIM, partir para a reflexao
            if k + 1 == refletora:
                reflect(medium.layers[k + 1].supInt, medium.s0.rays[j])
                break
            else:
                # Executando a lei de Snell para refracao
                refract(medium.layers[k + 1].supInt, medium.layers[k].velocity,
                    medium.layers[k + 1].velocity, medium.s0.rays[j], 1)
            k += 1
        # Subindo
        while True:
            # Tracando o raio
            go(medium.layers[k].velocity, medium.layers[k].supInt,
                medium.s0.rays[j], -1, dimension[0], tMax)
            # Caso SIM, chegamos ao topo
            if k == 0:
                break
            # Executando a lei de Snell para refracao
            else:
                refract(medium.layers[k].supInt, medium.layers[k].velocity,
                    medium.layers[k - 1].velocity, medium.s0.rays[j], -1)
            k -= 1

plot(medium.s0.rays, dimension[0], lateralPoints)
```

```
from exceptionsRT import singularMatrix as sm
import numpy as np

# ---------------------- Funcoes auxiliares ao Runge-Kutta ------------------- #
def didItTouchTheInterface(x, y, _i, s):
    '''
        Funcao que determina se o raio ainda esta na camada atual
        Recebe:         (x, y) - ponto atual do raio
                        _i     - proxima interface
                        s      - sentido que o raio esta seguindo (para cima ou
                                 para baixo)
        Retorna:        Caso o raio esteja descendo pelo meio, se a ultima coor-
                        denada y calculada para ele foi maior que o y da inter-
                        face calculado para o x do raio, entao ele ultrapassou a
                        interface.
                        Caso ele esteja subindo pelo meio e seu y for menor que
                        o y calculado para o x do raio na interface, entao o
                        raio ultrapassou a interface
    '''
    if s == 1:
        if y > _i(x):
            return 1
        else:
            return 0
    else:
        if y < _i(x):
            return 1
        else:
            return 0
# ---------------------------------------------------------------------------- #

def rungeKutta4Ordem(EDO, tMax, h, To, Yo, paramRK = 0):
    '''
        Funcao que implementa o metodo de Runge-Kutta de quarta ordem (RK4)
        Recebe:         EDO     - edos a serem resolvidas numericamente pelo RK4
                        tMax    - final do intervalo I de tempo para o qual a
                                  edo sera resolvida (I = [To, tMax])
                        h       - Passo dado no tempo
                        To      - Inicio do intervalo I
                        Yo      - array com os valores iniciais das equacoes a
                                  serem resolvidas pelo RK4
                        paramRK - parametros para o RK4
       Retorna:         T       - array com todos os valores calculados para o
                                  tempo durante a execucao do RK4
                        Y       - Valores das equacoes calculados para cada pas-
                                  so dado pelo RK4
    '''
    # Variavel para indexacao dos vetores
    i = 0

    # Determinando o passo
    N = int(tMax / h)

    # Recebe o numero de equacoes a serem resolvidas pelo metodo
    numEq = EDO.dimension

    # Criando vetores de tempo e imagem
    T = np.zeros(        N + 1)
    Y = np.zeros((numEq, N + 1))

    # Criacao de vetores de constantes
    K1 = np.zeros((numEq, 1))
    K2 = np.zeros((numEq, 1))
    K3 = np.zeros((numEq, 1))
    K4 = np.zeros((numEq, 1))

    # Preenchimento inicial dos vetores
    T[0]          = To
```

```
    Y[0:numEq, 0] = Yo

    # Descrevendo quais sao os parametros
    v    = paramRK[0]
    _i   = paramRK[1]
    dimX = paramRK[2]
    s    = paramRK[3]

    for i in range(0, N):
        # Constantes do metodo de Runge-Kutta
        K1 = EDO.evaluate(Y[0:numEq, i]                , v)
        K2 = EDO.evaluate(Y[0:numEq, i] + 0.5 * h * K1, v)
        K3 = EDO.evaluate(Y[0:numEq, i] + 0.5 * h * K2, v)
        K4 = EDO.evaluate(Y[0:numEq, i] +       h * K3, v)

        # Prepara o proximo Y
        Y[0:numEq, i + 1] = (Y[0:numEq, i] + (h / 6.0) * (K1 + 2.0 * (K2 + K3) \
                            + K4))

        # Testando se o raio saiu do dominio
        if Y[0, i + 1] > dimX or Y[0, i + 1] < 0. or Y[1, i + 1] < 0.:
            return Y[:,:i + 2], T[:i + 1]

        # Testando se o raio passou para outra camada
        if (didItTouchTheInterface(Y[0, i + 1], Y[1, i + 1], _i, s)):
            # Imaginamos uma reta ligando os dois ultimos pontos tracados
            diretor = np.array([Y[2, i], Y[3, i]])
            # Criando a matriz com as componentes dos vetores diretores da reta
            # e da interface (matriz A)
            A = np.array([[diretor[0], -_i.vDiretor[0]], [diretor[1], -_i.vDiretor[1]]])

            # Para o caso da matriz A ser singular
            det = np.linalg.det(A)
            if det > -.0005 and det < .0005:
                raise sm()

            # Criando a matriz B
            B = np.array([-Y[0, i], _i.b - Y[1, i]])

            # Resolvendo o sistema linear
            X = np.linalg.solve(A, B)

            # Recolhendo o parametro s_0 para determinar o ponto de intersecao
            # entre as retas
            s_0 = X[1]

            # Determinando o ponto de intersecao
            Y[0, i + 1] =      + _i.vDiretor[0] * s_0
            Y[1, i + 1] = _i.b + _i.vDiretor[1] * s_0
            return Y[:,:i + 2], T[:i + 1]

        # Prepara o proximo tempo
        T[i + 1] = T[i] + h
    return Y, T
```

```
from classesRT import ray, source, velocity, layer, medium
import matplotlib.pyplot as plt
import numpy as np

def userInput():
    '''
        Funcao que recebe as entradas do usuario
        Retorna:        dimension - array com as dimensoes do meio
                        posY - posicao em y da fonte
                        nLayers - numero de camadas do meio
                        nRays - numero de raios a serem tracados
                        angMin - angulo minimo da emissao dos raios
                        angMax - angulo maximo da emissao dos raios
                        lateralPoints - coordenadas em y dos pontos laterais de
                                        cada interface
                        velocidades - array dos coeficientes das funcoes veloci-
                                      dade de cada camada
                        tMax - tempo maximo que os raios podem permanecer em
                               uma camada
                        refletora - camada onde os raios devem refletir
    '''
    print "Seja bem-vindo ao tracador de raios!"
    print ""
    print "OBS: muitas entradas aqui serao em ponto flutuante"
    print "por isso, nao se esqueca de usar '.' ao inves de ','"
    print ""
    dimension = np.zeros(2)
    print "Digite as dimensoes desejadas para o meio (separadas por espaco): "
    dimension = input()
    print "Digite a posicao vertical da fonte: "
    posY = input()
    print "Digite o numero de camadas: "
    nLayers = input()
    print "Digite o numero de raios: "
    nRays = input()
    print "Digite os angulos minimo e maximo de emissao dos raios"
    print "OBS: separado por espaco"
    angMin, angMax = input()
    lateralPoints = np.zeros(nLayers, dtype=(float, 2))
    print "A seguir, entre com os pontos laterais de cada interface"
    print "LEMBRE-SE: voce definiu ", nLayers, " camadas, entao haverao "
    print nLayers - 1, " interfaces, sem contar o topo do meio!"
    print "OBS: separe os pontos com espaco para cada interface"
    for i in range(1, nLayers):
        print "Interface: ", i
        lateralPoints[i, :] = input()
    velocidades = np.zeros(nLayers, dtype=(float, 3))
    print "A seguir, entre com os coeficientes das funcoes velocidade de cada ",
    print "camada"
    print "LEMBRE-SE: voce definiu ", nLayers, " camadas, entao haverao "
    print nLayers, " funcoes velocidade!"
    print "OBS: separe os coeficientes com espaco para cada camada"
    for i in range(0, nLayers):
        print "Camada: ", i
        velocidades[i, :] = input()
    print "Digite o tempo maximo que o raio pode ficar dentro de cada camada: "
    tMax = input()
    print "Digite a camada refletora: "
    refletora = input()

    return (dimension, posY, nLayers, nRays, angMin, angMax, lateralPoints,
        velocidades, tMax, refletora)

def buildMedium(dimension, posY, nLayers, nRays, angMin, angMax, lateralPoints,
                velocidades):
    '''
        Funcao que constroi o meio em que os raios propagarao
        Recebe:         dimension - dimensoes do meio
```

```
                        posY - posicao y da fonte
                        nLayers - numero de camadas do meio
                        nRays - numero de raios que propagarao no meio
                        angMin - inicio do intervalo angular de disparo dos
                                 raios
                        angMax - fim desse intervalo
                        lateralPoints - lista de coordenadas em y dos pontos
                                        laterais das interfaces
                        velocidades - array contendo os coeficientes dos modelos
                                      de velocidades das interfaces
        Retorna:        m - objeto medium
    '''
    # Criando camadas
    layers = []
    for i in range(0, nLayers):
        vel = velocity("0", velocidades[i][0], velocidades[i][1],
                velocidades[i][2])
        aux = layer(dimension, [lateralPoints[i][0], lateralPoints[i][1]], vel)
        layers.append(aux)

    # Setando as interfaces inferiores nas camadas
    for i in range(nLayers - 2, 0, -1):
        layers[i].setInfInt(layers[i + 1].supInt)

    # Construindo a fonte e seus raios
    s0 = source(posY, angMin, angMax, nRays, layers[0].velocity(0., 0., "0"))
    # Criando o meio
    m = medium(dimension, s0, layers)

    return m

def deprecated_somaEspessuras(l, k):
    '''
        Soma as espessuras das camadas passadas por parametro ate a camada k
    '''
    summ = 0.
    for i in range(0, k):
        summ += l[i].espessura
    return summ

def degreesToRadians(angle):
    '''
        Funcao que retorna um angulo passado em graus, por parametro, convertido
        para radianos
    '''
    return angle * np.pi / 180

def radiansToDegrees(angle):
    '''
        Funcao que retorna um angulo passado em radianos, por parametro, conver-
        tido para graus
    '''
    return angle * 180 / np.pi

def plot(rays, dimX, pontosLaterais):
    '''
        Metodo que realiza e salva a plotagem dos raios propagando pelo meio.
        Recebe:         rays - lista de raios, que possuem os pontos a serem
                               plotados
                        dimX - dimensao do meio em x
                        pontosLaterais - pontos usados para auxiliar a exibicao
                                         das interfaces
    '''
    # Criando figura
    fig = plt.figure()

    # Adicionando eixos
    fig.add_axes()
```

```
    # Criando subplot no eixo
    ax = fig.add_subplot(111)

    # Plotando os raios
    for i in range(0, len(rays)):
        ax.plot(rays[i].XP[0, :], rays[i].XP[1, :])

    # Colocando titulo e legendas no grafico
    ax.set(ylabel='Profundidade', xlabel='Alcance')

    # Plotando as Camadas
    for i in range(0, pontosLaterais.size / 2):
        ax.plot((0., dimX), (pontosLaterais[i][0], pontosLaterais[i][1]), '-k')

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

    # Salvando a imagem
    # Definindo caminho da plotagem
    caminho = 'TR.png'
    plt.savefig(caminho)

def projVetorial(v, u):
    '''
        Funcao que retorna a projecao vetorial do vetor v sobre o vetor u
        Recebe:         v - array que representa o vetor a ser projetado
                        u - array que representa o vetor que recebera a projecao
        Retorna:        a projecao vetorial de v sobre u
    '''
    num = v.dot(u)
    den = u.dot(u)
    return num / den * u
```