CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
DE MINAS GERAIS

ENGENHARIA DE COMPUTAÇÃO

Modelagem Matemática da Propagação de Ondas em Meios Não Homogêneos

Orientando:
Marcelo Lopes de Macedo
FERREIRA CÂNDIDO

Orientador:
Prof. Dr. Luis Alberto
D'AFONSECA

BELO HORIZONTE
8 de agosto de 2018

Sumário

Capítulo 1

Introdução

O ser humano interage e necessita interagir com ondas a todo momento. Essas pertubações mecânicas ou eletromagnéticas fazem parte do nosso cotidiano desde o nosso acordar, pela luz (um tipo de onda eletromagnética) que é emitida/refletida até nós e que nos permite observar o mundo ao nosso redor pelos nossos olhos ou pelo som (composto por ondas mecânicas) emitido por carros, fábricas, choro de crianças ou um belo canto de pássaros que são captados por nossos ouvidos. Interagimos com as ondas até indiretamente, como o leitor agora interage, por exemplo, ou estando sentado em uma cadeira ou andando sobre um calçado ou se deslocando em algum meio de transporte, que provavelmente tem algum mineral adquirido através de estudos sismológicos, que utilizam massivamente os conceitos de ondulatória.

Na área da Geologia (e também na Medicina, Física, construção civíl, por exemplo), aplicam-se, pelo menos, dois métodos para o estudo de fenômenos ondulatórios e de estruturas geológicas não homogêneas: o **método de diferenças finitas** e o **traçamento de raios**. Utiliza-se esses métodos (principalmente o segundo) na busca por minérios e pela história da formações de estruturas geológicas.

Neste trabalho, objetivamos comparar esses dois métodos numéricos computacionais na simulação da propagação de ondas através de meios não-homogêneos. Tal comparação se dará nos aspectos da dificuldade de implementação dos métodos, seus custos, sua intuitividade e qual seria a melhor área para a aplicação de tais métodos.

Para alcançar tal objetivo, o autor deste relatório e orientando dessa iniciação científica passa por uma série de estudos prévios (Capítulos 2, 3, 4, 5 e 6) que dão base ao estudo realizado nesse trabalho, para enfim abordar os métodos a serem comparados (Capítulos 7, 8 e 9).

Os principais códigos desenvolvidos durante o projeto, bem como esse próprio relatório, podem ser encontrados no repositório do autor (`https://github.com/MarceloFCandido/PIC`). Os códigos envolvidos diretamente na comparação dos métodos também se encontram no apêndice desse relatório.

Capítulo 2

Uma Breve Introdução à Computação Científica

A computação científica visa a resolução de problemas físicos, matemáticos, químicos, biológicos ou de outra área da ciência utilizando-se o Cálculo Numérico executado por computadores. Essas resoluções são de cunho numérico (ou seja, não são analíticas) e obtidas através da elaboração e execução de algoritmos que modelam o problema a ser estudado. Esses algoritmos são executados por computadores e se baseiam em operações aritméticas e lógicas, que são as únicas que um computador pode realizar [13].

2.1 Soluções de Problemas

Segundo Campos [13], a resolução de problemas se dá por meio de quatro etapas:

1. **definição do problema**: determina-se qual o problema a ser resolvido;
2. **modelagem matemática**: obtém-se o modelo matemático do problema real por meio de uma formulação matemática;
3. **solução numérica**: determina-se qual o método numérico a ser utilizado para a resolução do problema modelado matemático. Implementa-se o método na forma de uma algoritmo, que, após ser codificado em um programa, deve ser executado por um computador;
4. **análise dos resultados**: avalia-se se a solução obtida é satisfatória para o problema. Caso contrário, modela-se novamente o problema através de uma nova formulação matemática a fim de se obter uma nova solução numérica.

2.2 Aritmética Computacional

Tudo o que um computador consegue entender são *strings* (cadeias de caracteres) de zeros e uns. Cada um dos dos componentes dessas *strings* é conhecido por *bit*, que também pode ser definido como um dígito **binário**. Toda a matemática que um computador consegue realizar é baseada nessa representação. Por conta disso, um computador consegue representar a forma exata de alguns números racionais. Entretanto, os demais números são representados por aproximações. Como sabemos, a maior parte dos números desse tipo tem casas decimais e, para serem representados computacionalmente, precisam seguir um padrão específico [5].

Em 1985, o Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE, em inglês) adotou o padrão técnico IEEE-754 para ponto flutuante, que é a forma pela qual um computador consegue representar números com casas decimais. Muitos fabricantes de microprocessadores adotaram esse padrão, que define um ponto flutuante como um "real longo" de 64 *bits* na forma

$$(-1)^s \cdot 2^{c-1023}(1-f)$$

na qual s é um indicador de sinal, c é um expoente chamado de "característica" e f é uma fração binária que recebe o nome de "mantissa". Na string de 64 *bits*, s recebe um *bit*, c recebe 11 e f recebe 54 *bits* [5].

O uso desse formato para a representação de números leva a erros de arredondamento em cálculos computacionais caso um dos números envolvidos nos cálculos não seja uma potência de dois. Isso ocorre por conta da representação numérica da máquina ser finita, o que leva a aproximações numéricas. Esse erro deve ser controlado para que não interfiram significativamente nos resultados dos cálculos [5, 13].

2.3 Tipos de Erros Envolvidos na Solução de Problemas

Também segundo Campos [13], existem pelo menos três tipos de erros que podem ser encontrados na solução de problemas (excetuando-se os erros grosseiros referidos pelo autor). São eles:

1. **erro de arredondamento e/ou truncamento**: a transformação de um número em seu correspondente em ponto flutuante pode se dar por

   - **arredondamento**: seja uma máquina com suporte a números de k algarismos. Se ela receber um número i de $k+n$ algarismos e o $(k+1)$-ésimo algarismo for maior ou igual a 5, soma-se 1 ao k-ésimo algarismo e corta-se (trunca-se) os n últimos algarismos de i. Caso o $(k+1)$-ésimo algarismo for menor que 5, realiza-se apenas o truncamento;
   - **truncamento**: ainda considerando uma máquina como a do caso anterior, se ela receber um número i de $k+n$ algarismos, ela truncará os últimos n algarismos de i [5];

2. **erro absoluto e erro relativo**: o erro absoluto é medido por

   $$e_{\text{abs}} = v_{\text{real}} - v_{\text{aprox.}} \tag{2.1}$$

   onde e_{abs} é o erro absoluto, v_{real} é o valor real e $v_{\text{aprox.}}$ é o valor aproximado. Já o erro relativo é dado por

   $$e_{\text{rel}} = \frac{v_{\text{real}} - v_{\text{aprox.}}}{v_{\text{real}}} \tag{2.2}$$

   onde e_{rel} é o erro relativo, v_{real} é o valor real e $v_{\text{aprox.}}$ é o valor aproximado [13].

3. **erro na modelagem**: em alguns problemas, é necessária a criação de uma expressão matemática na etapa de modelagem. Essa expressão é criada a partir de dados experimentais, que podem não estar próximos o bastante do problema real e, dessa forma, interferirem significativamente nos resultados. Com isso, torna-se necessária uma nova modelagem [13];

Capítulo 3

Equações Diferenciais Ordinárias

3.1 Definição e Características de Equações Diferenciais

Primeiramente, antes de definir uma equação diferencial ordinária, é necessário se definir e caracterizar uma equação diferencial. As equações diferenciais são, basicamente, aquelas cujas incógnitas são funções (variáveis dependentes) de uma variável independente, podendo envolver também algumas derivadas dessas funções.

3.1.1 A Ordem de uma Equação Diferencial

No mundo das equações diferenciais, a **ordem** da equação é determinada pela derivada de maior ordem existente na equação. Ou seja, se em uma equação, na qual a incógnita é y, a derivada de maior ordem da função-incógnita é $y^{(n)}$, então a equação diferencial é de ordem n. Para maior elucidação, temos por exemplo a equação

$$y' + 3y = 0$$

que é dita de primeira ordem, pois a derivada de ordem mais alta, que está na equação, da função y é y', de ordem 1. Já no caso da equação

$$y'' + 5y = y''' + 3t$$

a ordem é 3, pois a derivada de y com ordem mais alta na equação é y'''.

3.1.2 Linearidade

Uma equação diferencial é dita **linear** quando todos os coeficientes que multiplicam as derivadas de y são funções da(s) variável(eis) independentes $a_i(t)$, com $i = 0, 1, ..., n$, não sendo essas funções da própria y ou uma de suas derivadas. Por exemplo: $... + xy^{(i)} + \cdots$, sendo t a variável de y.

A equação diferencial ordinária linear geral é

$$a_{n+1}(t)y^{(n)}(t) + a_n(t)y^{(n-1)}(t) + ... + a_3(t)y''(t) + a_2(t)y'(t) + a_1(t)y(t) + a_0 = f(t) \qquad (3.1)$$

Qualquer equação que saia desse padrão é dita **não-linear**.

3.1.3 Solução de uma Equação Diferencial Ordinária

Uma função $y(t)$ é dita solução de uma equação diferencial se ela e suas respectivas derivadas estão definidas em determinado intervalo e podem ser substituídas na equação sem ocorrer desigualdades [20]. Por exemplo, $y(t) = e^t$ é solução de

$$y' - y = 0 \tag{3.2}$$

pois $y'(t) = e^t$ e a subtração entre $y(t)$ e $y'(t)$ leva a um resultado nulo. Em contrapartida, $y(t) = \cos t$ não pode ser solução de (3.2) pois a função subtraída de sua derivada $y'(t) = -\operatorname{sen} t$ não leva a um resultado nulo.

3.2 A Definição de uma Equação Diferencial Ordinária

Uma Equação Diferencial Ordinária (EDO) é uma equação como

$$a_{n+1}y^{(n)}(t) + a_n y^{(n-1)}(t) + \cdots + a_3 y''(t) + a_2 y'(t) + a_1 y(t) + a_0 = f(t) \tag{3.3}$$

sendo a função $y(t)$ e suas derivadas $y'(t)$, $y''(t)$, ..., $y^{(n-1)}(t)$ e $y^{(n)}(t)$ as incógnitas da equação. Ou seja, a solução dessas equações são funções com **uma** variável independente (por isso é dita ordinária, se houvessem mais variáveis independentes a equação diferencial seria dita parcial, tipo que será tratado mais a frente) [4, 20].

3.3 Sistemas de Equações Diferenciais Lineares

Assim como existem sistemas de equações de n-ésimo grau, as equações diferenciais também podem ser agrupadas em sistemas como o abaixo

$$\begin{cases} y_1'(t) = a_1 y_1(t) + b_1 y_2(t) \\ y_2'(t) = a_2 y_2(t) + b_2 y_1(t) \end{cases} \tag{3.4}$$

Mais a frente serão mostradas aplicações para esse tipo de sistema.

3.4 Problema de Valor Inicial (PVI)

Um problema de valor inicial (PVI) é dado por

$$y' = f(t) \tag{3.5}$$

$$y(t_0) = y_0 \tag{3.6}$$

sendo (3.5) uma equação diferencial ordinária e (3.6) sua condição inicial, ou seja, como o sistema que (3.5) descreve se encontrava no instante t_0, em geral consideramos que t se refere a tempo [20].

Um PVI também pode ser descrito por um sistema de equações diferenciais ordinárias. Nesse caso, devem haver n condições iniciais para cada uma das n equações que formarem o sistema.

3.5 Problema de Valores de Contorno para Fronteiras com Dois Pontos (PVC)

Considere uma equação diferencial de segunda ordem como

$$y'' + p(t)y' + q(t)y = g(t) \tag{3.7}$$

com as seguintes condições iniciais

$$y(\alpha) = y_0 \tag{3.8}$$
$$y(\beta) = y_1 \tag{3.9}$$

Esse tipo de problema é chamado **problema de valores de contorno com dois pontos** [4]. Esse tipo de problema costuma ser usado para descrição de situações espaciais em equilíbrio, ou seja, que não variam no tempo.

3.6 Aplicações das Equações Diferenciais Ordinárias

Por se tratarem de equações envolvendo taxas de variação, as EDO's são muito úteis para descrever situações físicas, químicas e biológicas. Alguns exemplos de situações nas quais esse tipo de equação é utilizado são

- o modelo de crescimento populacional de Malthus;
- o carregamento e o descarregamento de um capacitor;
- a função logística criada por Verhulst;
- as equações presa-predador de Lotka-Volterra;
- os sistemas massa-mola;
- descrições de reações químicas.

Algumas dessas aplicações serão mais detalhadas no Capítulo 4, que também exporá exemplos de como podem ser resolvidas numericamente e os resultados desses exemplos.

Capítulo 4

Aproximação de Problemas de Valores Iniciais (PVI's): Implementação e Exemplos

As equações diferenciais ordinárias podem ser solucionadas analiticamente usando-se ou a integração ou o método dos fatores integrantes ou séries de potências ou a transformada de Laplace, dependendo de sua forma. Esses métodos de resolução de EDO's não serão apresentados aqui, por fugirem ao escopo desse trabalho, mas o leitor pode se inteirar sobre eles, podendo encontrar informações bem fundamentadas e diversos exercícios em [4, 20].

Esse trabalho visa o uso de métodos numéricos para a aproximação de PVI's (e também, rapidamente, PVC's). Dentre os métodos para resolução de PVI's, destacam-se o método de Euler e os métodos de Runge-Kutta (no caso desse trabalho, o de 4ª ordem). Para a resolução de PVC's pode-se usar o método de Diferenças Finitas, por exemplo, a ser visto mais a frente.

4.1 O Método de Euler

Esse é um método pouco usado na prática para a aproximação de PVI's [5] devido a sua alta divergência do resultado real após algumas iterações. Entretanto, por poder ser facilmente explicado e por suas variações serem utilizadas para a finalidade deste trabalho, será definido aqui esse método.

Seja um PVI como o da Seção 3.4 e seja $y(t)$ a solução desse PVI. Podemos representar qualquer função por uma série de Taylor como

$$T_n(x) = \sum_{j=0}^{n} \frac{f^{(j)}(c)}{j!}(x-c)^j \tag{4.1}$$

Seja então

$$y(t) = y(t_0) + hy'(t_0) + \frac{h^2}{2}y''(t_0) + \frac{h^3}{6}y'''(t_0) + \cdots \tag{4.2}$$

a expansão da solução $y(t)$ em série de Taylor em torno do valor inicial t_0 e sendo h o passo usado para a geração dos valores de t.

Se truncarmos (4.2) logo depois do termo da primeira derivada, tendo $t_1 = t_0 + h$ e y_1 como a aproximação de $y(t_1)$, além de se saber que $y' = f(t,y)$, temos que

$$y_1 = y_0 + f(t_0, y_0)$$

Logo, as aproximações y_i de $y(t_i)$ podem ser obtidas semelhantemente à forma da equação anterior [13]

$$y_{i+1} = y_i + hf(t_i, y_i) \tag{4.3}$$

Um código em Python para o método de Euler, baseado no algoritmo de Frederico [13], seria

```
# a  - margem do intervalo a esquerda
# b  - margem do intervalo a direita
# m  - numero de subintervalos
# y0 - valor de f(a)

def Euler(a, b, m, y0):
    h       = (b - a) / m # Calculando o tamanho do passo de acordo com o numero de pontos
    x       = a
    y       = y0
    Fxy     = f(x, y) # Avaliando a funcao no extremo esquerdo (sendo a < b)

    vetX[0] = x # Colocando os primeiros valores de x e y nos vetores
    vetY[0] = y

    for i in range(0, m):
        x   = a + i * h # Calculando x(i)
        y   = y + h * Fxy
        Fxy = f(x, y)

        print "i: " + i + "x: " + x + "y: " + y + "Fxy: " + Fxy

        vetX[i + 1] = x
        vetY[i + 1] = y

    return vetX, vetY
```

Listing 4.1: Método de Euler

4.2 O Método de Runge-Kutta

Segundo Burden [5], os métodos de Runge-Kutta apresentam a vantagem, em relação aos Taylor, de não necessitarem do cálculo das derivadas de uma função $f(x,y)$, além de terem o erro de truncamento local de alta ordem .

O método de Runge-Kutta de quarta ordem (RK4) consiste em um sistema não-linear com 11 equações e 13 incógnitas. Essas equações podem ser obtidas de forma semelhante a obtenção do método de Euler, através da expansão e truncamento de um polinômio de Taylor para uma função $f(x,y)$ [13]. O método é definido por

$$y' = f(y(t),t) \tag{4.4}$$

$$K_1 = f(y^*(t_0),t_0) \tag{4.5}$$

$$K_2 = f(y^*(t_0) + \frac{h}{2}K_1, t_0 + \frac{h}{2}) \tag{4.6}$$

$$K_3 = f(y^*(t_0) + \frac{h}{2}K_2, t_0 + \frac{h}{2}) \tag{4.7}$$

$$K_4 = f(y^*(t_0) + hK_3, t_0 + h) \tag{4.8}$$

$$y^*(t_0 + h) = y^*(t_0) + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \tag{4.9}$$

com $y(t_0) = y_0$.

Um código em Python para esse método pode ser visto abaixo:

```
def rungeKutta4Ordem(EDO, tMax, N, To, Yo, paramRK = 0):
    '''
        Definicao do metodo de Runge-Kutta de 4 ordem
        Recebe:         EDO     - objeto da classe EqDiferencialOrdinaria
                        tMax    - tempo maximo que a resolucao pode alcancar
                        N       - numero maximo de passos que a resolucao pode
                                  dar
                        To      - instante inicial para a resolucao
                        Yo      - valores das EDO's a serem resolvidas no
                                  instante inicial
                        paramRK - array com parametros para o metodo
        Retorna:
                        T - array com os instantes de tempo pelos quais o metodo
                            passa
                        Y - array com os valores das EDO's para cada instante de
                            tempo
    '''
    # Variavel para indexacao dos vetores
    i = 0

    # Recebe o numero de equacoes a serem resolvidas pelo metodo
    numEq = EDO.getDimension()

    # 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

    # Determinando o passo
    h = tMax / N

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

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

        # Prepara o proximo tempo
        T[i + 1] = T[i] + h

    return (T, Y)
```

Listing 4.2: Método de Runge-Kutta de quarta ordem (RK4)

O código acima é utilizado nesse trabalho como uma ferramenta para a resolução de EDO's que serão apresentadas no Capítulo 8, além é claro, de usá-lo na demonstração de exemplos da resolução numérica de EDO's pelo mesmo.

Basta importá-lo no código em que se deseja usá-lo e realizar a chamada da função *rungeKutta4Ordem*, passando-se os devidos parâmetros a ele. Dentre esses parâmetros, se encontra um por nome *EDO*, que é um objeto da classe *EqDiferencialOrdinaria* (cuja definição é apresentada na próxima seção), que representa a EDO a ser resolvida pelo método.

4.3 Exemplos de Resoluções de PVI's Usando o RK4

O código a seguir abstrai uma ou mais EDO('s) computacionalmente. Assim como o código apresentado para o método de Runge-Kutta de 4ª ordem, apresentado na seção anterior, esse código pode ser chamado por um programa.

```
#!-*- 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
```

Listing 4.3: Classe abstrata que define uma equação diferencial ordinária de forma básica

A função abstrata (que tem apenas a sua assinatura definida, não sendo implementada em primeiro momento) *evaluate* tem seu código escrito pelo programa que a chama, visto que cada EDO apresenta um comportamento próprio.

4.3.1 Modelo de Crescimento Populacional de Malthus

Usado na área da ecologia e nos estudos sobre dinâmica populacional, o modelo descrito por Malthus

$$\begin{cases} \dfrac{dy}{dt} = ky \\ y(t_0) = y_0 \end{cases} \tag{4.10}$$

no qual k representa a taxa de crescimento populacional e y_0 é a população existente no tempo t_0, diz que a taxa de crescimento populacional $y'(\dfrac{dy}{dt})$ é proporcional à população existente y naquele instante t. Sua solução analítica é $y(t) = y_0 e^{kt}$ [20, 26].

Como exemplo de resolução numérica, por meio do método de Runge-Kutta, para esse modelo apresentamos o seguinte código. Seja a equação diferencial

$$\frac{dy}{dt} = e^t \tag{4.11}$$

(cuja solução analítica é $y(t) = e^t$).

O código a seguir realiza um resolução numérica para esse problema de Malthus utilizando o método de Runge-Kutta de 4ª ordem. A classe *EqMaltus* extende a classe *EqDiferencialOrdinaria*, definindo a função *evaluate* para retornar e^t. São pedidos 50 passos e parte-se do instante $t = 0,0$.

```
# Definindo a classe Maltus
class EqMaltus(edo):
    def __init__(self, dimension):
        # Passando argumento para o construtor da superclasse
        edo.__init__(self, dimension)

    # Implementando o metodo evaluate (especifico para esse caso)
    def evaluate(self, sliceT, sliceY):
        e = 2.71828182846
        return e ** sliceT

def main():
    print "Esse programa busca plotar o grafico de e^x por meio do metodo de Runge-Kutta da 4a ordem"

    # Numero de passos
    N = 50

    # Intervalo maximo para T
    tMax = 10.0

    # Valores iniciais
    To = 0.0
    Yo = 1.0

    # Instanciando EDO
    EDO = EqMaltus(1)

    # chama a funcao para execucao do Runge-Kutta
    T, Y = RK4(EDO, tMax, N, To, Yo)

    #chama a funcao para plotagem do grafico
    plotar(T, Y)

main()
exit()
```

Listing 4.4: Resolução por RK4 da equação 4.11

O gráfico gerado com o auxílio desse código é

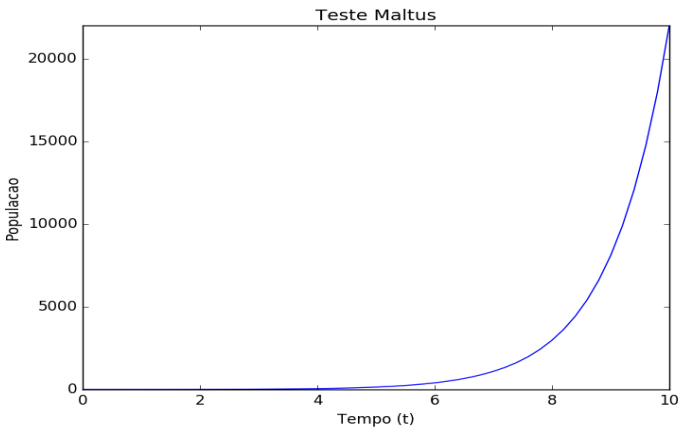


Fig. 4.1: Resolução numérica do problema 4.11

Como podemos ver, a resolução numérica condiz com a solução analítica exponencial.

4.3.2 Equações de Lotka-Volterra

Definição

As equações de Lotka-Volterra foram propostas pelo matemático Vito Volterra e pelo biofísico Alfred J. Lotka no ano de 1925. Elas são um par de equações diferenciais não-lineares e de primeira ordem.

As equações de Lotka-Volterra são mais utilizadas em quadros biológicos envolvendo presas e predadores (como lobos e ovelhas ou golfinhos e peixes), sendo a população das presas considerada a única fonte de alimento para a população de predadores e que não há competição entre os predadores.

Como descrevem um quadro simples, essas equações, em sua definição, não proporcionam uma análise muito próxima da realidade, mas o entendimento sobre elas possibilita o entendimento de situações mais complexas.

Principais Situações Descritas pelas Equações

Por suas relações diretas com o contexto de presa-predador, as equações de Lotka-Volterra podem descrever as seguintes situações

- Extinção de predadores;
- Extinção de presas;
- Existência de predadores e presas ao mesmo tempo;
- Geral.

Cada uma dessas situações para as equações será demonstrada a seguir, considerando x como o número de indivíduos para a população de presas, y para o número de indivíduos da população de predadores e t para o tempo.

Caso 01: Extinção de Predadores

Caso $y = 0$ (população de predadores extinta), a população de presas cresce proporcionalmente à população atual, o que é representado por k

$$y = 0$$

$$\frac{dx}{dt} = kx\,, \qquad k > 0$$

Caso 02: Extinção de Presas

Caso $x = 0$, ou seja, a população de presas seja extinta, a população de predadores deve se extinguir também, visto que as presas citadas no problema são sua única fonte de alimento. Esse decrescimento na população de predadores se dá proporcionalmente à população atual, sendo tal proporção representada por r

$$x = 0$$

$$\frac{dx}{dt} = -ry, \qquad r > 0$$

Caso 03: Existência de Predadores e Presas ao Mesmo Tempo

Diz-se que as situações em que presa e predador se encontram levam à morte de uma presa e as ocorrências de tais situações são proporcionais ao produtos das populações de presas e predadores. Essa situação é modelada utilizando-se duas constantes de proporcionalidade: a para a taxa de predação e b para a conversão de presa para predador. Logo:

- População de presas sofre queda: $-axy$
- População de predadores aumenta: bxy

Caso geral

Considerando todos os três casos acima em conjunto, tem-se:

$$\frac{dx}{dt} = x(k - ay)$$

$$\frac{dy}{dt} = y(rx - b)$$

onde k, a, b e r relacionam as duas populações.

A Relação Entre as Equações de Caso Geral e sua Solução Geral

Relacionando ambas as equações de caso geral, encontra-se

$$\frac{dy}{dx} = \frac{y(rx - b)}{x(k - ay)}$$

cuja solução geral, a partir de integração, é

$$kln(y) - a(y) + C = -bln(x) + b(x) + D$$

sendo C e D constantes de integração [28].

Implementação

O código a seguir propõe uma resolução para um problema de equações de Lotka-Volterra do livro Cálculo, vol. 2, de James Stewart [23]. Nesse exemplo, temos que as equações regem as populações de presas e predadores tendo-se

$$\begin{aligned} k &= 2,0 \\ a &= 0,01 \\ r &= 0,5 \\ b &= 0,0001 \end{aligned}$$

sendo sua condição inicial $t = 0,0$ com 2000 presas e 35 predadores.

Espera-se que os resultados exibam oscilações entre as populações de presas e predadores. A medida que a população predatória cresce, a de presas deve cair, levando também a predatória a queda, o que fará a de presas subir. Isso deve se dar em ciclos.

```
class EqPresaPredador(edo):
    # Inicializacao de variaveis sem valor numerico
    k = a = r = b = None

    def setK(self, k):
        self.k = k

    def getK(self):
        return self.k

    def setA(self, a):
        self.a = a

    def getA(self):
        return self.a

    def setR(self, r):
        self.r = r

    def getR(self):
        return self.r

    def setB(self, b):
        self.b = b

    def getB(self):
        return self.b

    def __init__(self, dimension, k, a, r, b):
        edo.__init__(self, dimension)

        self.setK(k)
        self.setA(a)
        self.setR(r)
        self.setB(b)

    def evaluate(self, T, Y):
        # Criacao de vetor para retorno
        dY = np.zeros(2)

        # Equacao para presas
        dY[0] = self.k * Y[0] - self.a * Y[0] * Y[1]

        # Equacao para predadores
        dY[1] = -1 * self.r * Y[1] + self.b * Y[0] * Y[1]

        return dY

def main():
    # Numero de passos
    N = 100000

    # Intervalo maximo para T
    tMax = 100.0

    # Parametros
    k = 2.0
    a = 0.01
    r = 0.5
    b = 0.0001

    # Valores iniciais
    To = 0.0    # Instante inicial
    Co = 2000   # Populacao inicial de presas
    Lo = 35     # Populacao inicial de predadores
    Yo = np.array([Co, Lo])

    # Instanciando sistema de EDO's Lotka-Voltera
    EDO = EqPresaPredador(2, k, a, r, b)

    # chama a funcao para execucao do Runge-Kutta
    T, Y = RK4(EDO, tMax, N, To, Yo)

```

```
74      # Plotando as equacoes
75      plotar(T, Y)
76
77  def intro():
78      # Apresenta o algoritmo
79      print "Esse algoritmo retorna a solucao para o seguinte problema de Lotka-Volterra:\n"
80      print "Populacoes de pulgoes e joaninhas descritas pelas equacoes de Lotka-Volterra com\n"
81      print "k = 2.0, a = 0.01, r = 0.5, b = 0.0001\n"
82      print "Instante t = 0 com 2000 coelhos e 35 lobos."
83      print "Fonte: Calculo. v.2. Stewart, James."
84
85      # Faz as inicializacoes do programa pela funcao main
86      main()
87
88  intro()
89  exit()
```

Listing 4.5: Problema envolvendo as equações de Lotka-Volterra [23]

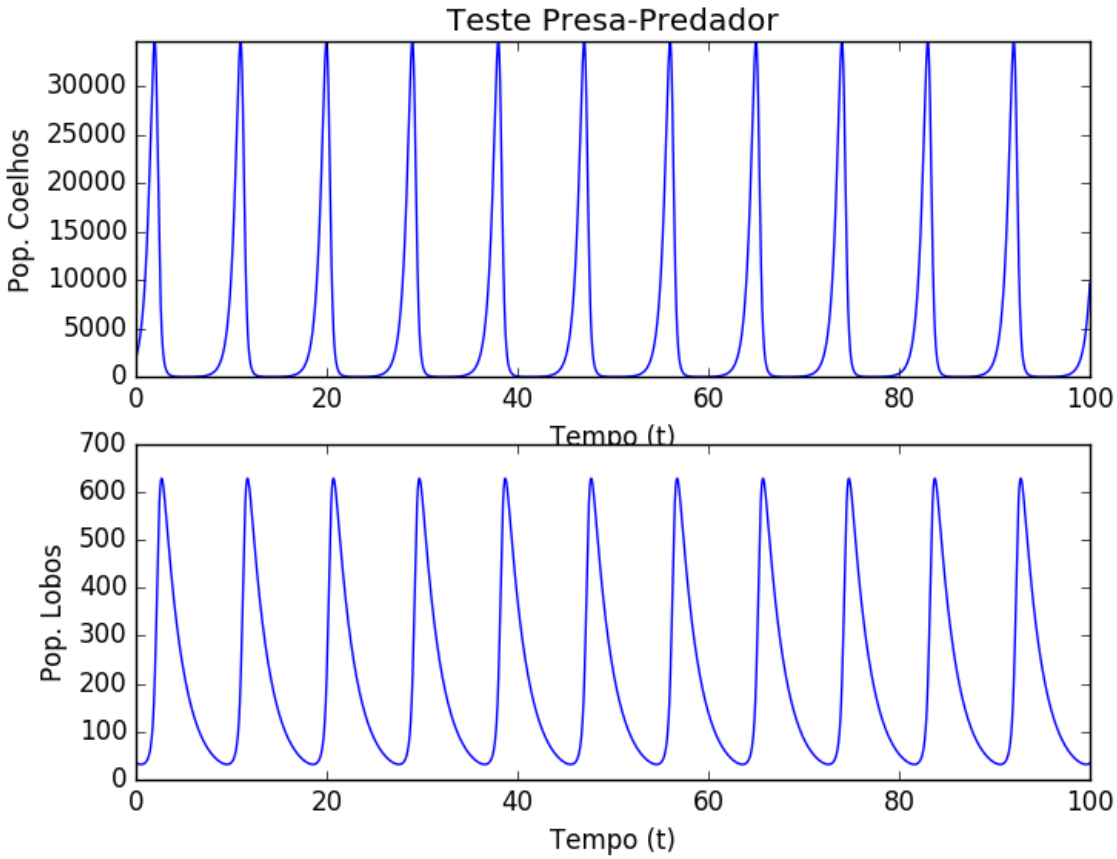


Fig. 4.2: Resultado do código 4.5

A Figura 4.2 ilustra, quantitativamente, as populações de joaninhas (predadores) e pulgões (presas) ao longo do tempo t. Podemos perceber que logo após a população de presas atingir seus picos, a população de predadores cresce e aplaca as presas, cuja população quase se anula. Com essa rápida queda, a população de predadores tende a decrescer também. Esse processo se repete ao longo do tempo.

Nesse tipo de problema, as populações de presas e predadores podem sofrer influência de fatores externos também, como o clima. Sendo ω a frequência da interferência do clima sobre a população de presas, α a amplitude dessa interferência e β a média dessa população ao longo

do tempo, podemos modelar a influência K do clima sobre a população de presas ao longo do tempo pela equação

$$K(t) = k(1 + \alpha \operatorname{sen} \omega t + \beta) \tag{4.12}$$

cujo código em Python se dá por

```
# Definicao de nao-homogeneidade
Kt = self.k * (1 + self.C * np.sin(self.freq * sliceT) + self.media)
```

Listing 4.6: Definição da influência do clima sobre a população de presas ao longo do tempo

dentro do método *evaluate* do Código 4.5

Para esse caso de influência pelo clima obtemos um resultado como da Figura 4.3. Nele, percebemos que as altas da população de presas se dão próximas à média da variação climática. Crescendo a população de presas, começa a crescer a população predatória, que causa um decrescimento drástico na primeira. Isso leva a população de predadores a decrescer também.

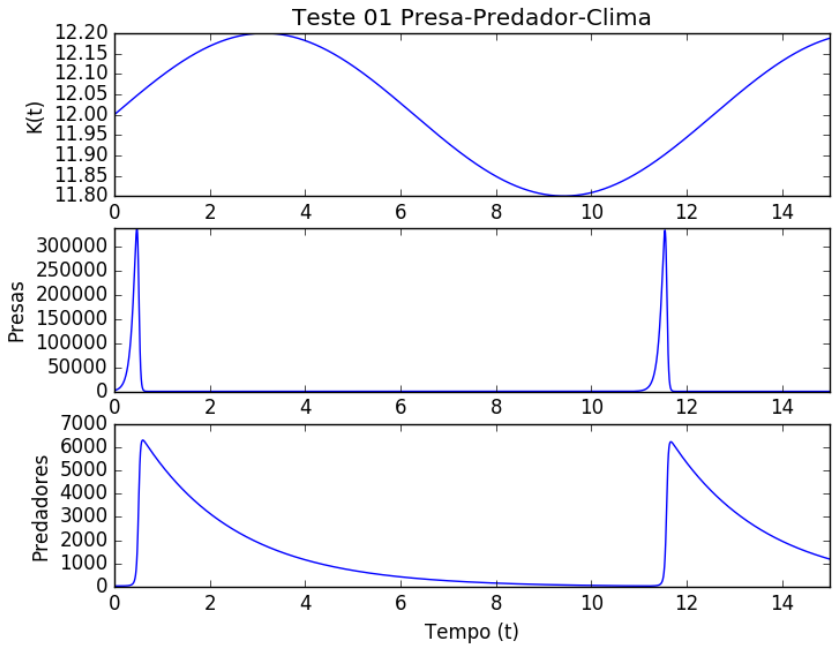


Fig. 4.3: Relação de presas e predatores agora influenciada pelo clima

4.3.3 Sistema Massa-Mola

Seja um corpo de massa m conectado a uma mola de constante elástica k como na Figura 4.4. Esse conjunto é nomeado **Sistema Massa-Mola**. Os experimentos envolvendo este sistema, em geral, consistem em tirar o conjunto de seu repouso e iniciar sua oscilação, medindo-se então as posições e velocidades do corpo durante as oscilações.

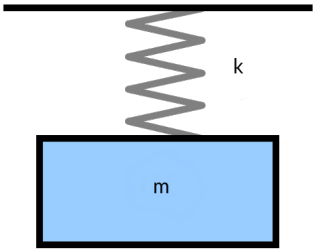


Fig. 4.4: Sistema massa-mola

O sistema vertical possui três forças agindo sobre ele: a força peso, que podemos nomear por P, apontando para baixo, a força elástica da mola, que podemos nomear por F_e e a força de resistência, que nomeamos por F_r. F_e última aponta para a direção contrária ao movimento da massa. Essas forças podem ser descritas pelas seguintes equações

$$P = mg \tag{4.13}$$
$$F_e = -ky(t) \tag{4.14}$$
$$F_r = \gamma y'(t) \tag{4.15}$$

sendo g a aceleração da gravidade, $y(t)$ o deslocamento feito na mola, γ a constante de amortecimento (resistência do ar) e $y'(t)$ a velocidade das oscilações na mola.

Pode haver ainda uma quarta força atuando sobre o sistema: a força externa representada por F_{ext}. Essa pode representar, por exemplo, um pistão conectado à mola, que fica gerando outras oscilações no sistema (como veremos na próxima subsubseção) ou, simplesmente, a força gravitacional, como escolhemos para este caso.

Se adotarmos o sistema cartesiano como referencial (termos positivos apontam para cima ou direita, termos negativos apontam para baixo ou esquerda) temos que, pela segunda Lei de Newton

$$F = ma \tag{4.16}$$

escrevemos a aceleração na forma de derivada

$$F = my''(t) \tag{4.17}$$

definimos a força resultante ($F = my''(t)$) como o somatório das forças atuantes sobre o sistema, considerando o sentido das mesmas

$$my''(t) = F_e + F_r - P \tag{4.18}$$

substituímos cada força atuante por suas fórmulas

$$my''(t) = -ky(t) + \gamma y'(t) - mg \tag{4.19}$$

isolamos a força externa no lado direito da equação

$$my''(t) - \gamma y'(t) + ky(t) = -mg \tag{4.20}$$

essa última equação, satisfeita por $y(t)$ descreve o movimento da massa durante as oscilações do sistema. Nesse tipo de equação, que descreve um comportamento oscilatório, o termo direito representa a força externa F_{ext} que age sobre o sistema. Nesse caso, podemos perceber que F_{ext} é a força gravitacional e que seu sinal negativo indica que ela está apontando para baixo.

O coeficiente γ que multiplica $y'(t)$ representa o amortecimento do sistema massa-mola. Empiricamente, o amortecimento pode se dar por um líquido colocado abaixo da massa ou pelo atrito da massa com o ar, por exemplo. O sistema também pode ser considerado sem amortecimento ($\gamma = 0$), o que significa que ele oscilará permanentemente. Mais detalhes sobre o amortecimento desses sistemas podem ser encontrados em [4, 20].

A solução analítica geral desse sistema é

$$u(t) = c_1 \cos \omega t + c_2 \operatorname{sen} \omega t \tag{4.21}$$

na qual, c_1 e c_2 são constantes e ω_0 é a frequência natural dada por $\sqrt{\frac{k}{m}}$ do sistema [20].

O exemplo a seguir resolve um sistema massa-mola amortecido descrito por

$$s'' + 0,125s' + s = 0$$

com condições iniciais $t = 0.0$, $V_0 = 0.0$ e $P_0 = 2.0$, sendo que V_0 e P_0 representam, respectivamente, a velocidade e a posição inicial da massa no sistema.

```
class EqMassaMola(edo):
    m = gamma = k = None

    def setM(self, m):
        self.m = m

    def getM(self):
        return self.m

    def setGamma(self, gamma):
        self.gamma = gamma

    def getGamma(self):
        return self.gamma

    def setK(self, k):
        self.k = k

    def getK(self):
        return self.k

    def __init__(self, dimension, m, gamma, k):
        # Inserindo dimensao do sistema
        edo.__init__(self, dimension)

        # Inserindo parametros
        self.setM(m)
        self.setGamma(gamma)
        self.setK(k)

    def evaluate(self, sliceT, sliceY):
        # Criacao de vetor para retorno
        valores = np.zeros(2)

        # Equacao para velocidade
        valores[0] = sliceY[1]

        # Equacao para posicao
        valores[1] = ((-self.gamma * sliceY[1] - self.k * sliceY[0]) / self.m)

        return valores

def main():
```

```
    # Numero de passos
    N = 10000

    # Define o intervalo maximo em t
    tMax = 50.0

    # Coeficientes da EDO
    m = 1.0        # Massa
    gamma = 0.125 # Coeficiente de amortecimento
    k = 1.0        # Constante elastica (da mola)

    # Valores iniciais
    To = 0.0  # Instante inicial
    Vo = 0.0  # Velocidade inicial
    Po = 2.0  # Posicao inicial
    Yo = np.array([Vo, Po])

    EDO = EqMassaMola(2, m, gamma, k)

    # chama a funcao para execucao do Runge-Kutta
    T, Y = RK4(EDO, tMax, N, To, Yo)

    # Plota os vetores
    plotar(T, Y)

def intro():
    # Apresenta o algoritmo
    print "Resolve a seguinte EDO"
    print "s'' + 0.125s' + s = 0"

    # Faz as inicializacoes do programa pela funcao main
    main()

intro()
exit()
```

Listing 4.7: Código da resolução numérica de um problema envolvendo um sistema massa-mola amortecido

cujo resultado é exibido na figura 4.5

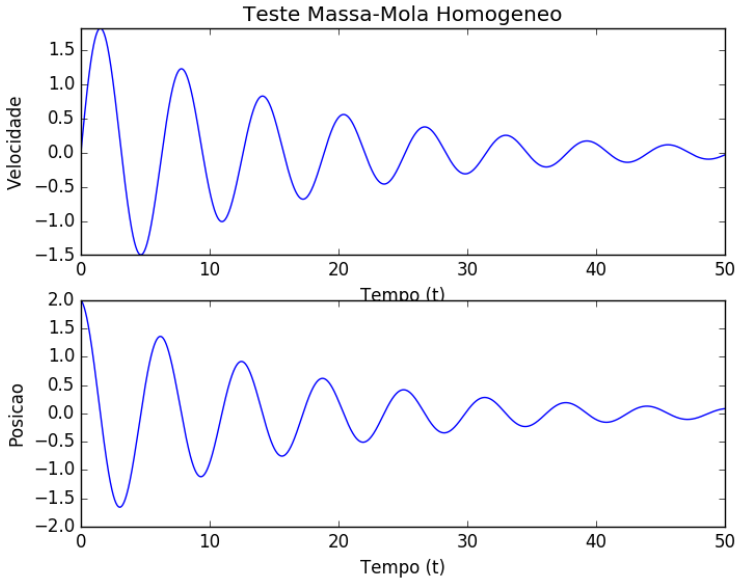


Fig. 4.5: Gráficos da posição e da velocidade da massa do sistema ao longo do tempo

No primeiro gráfico da Figura 4.5 vemos o gráfico da velocidade da massa em cada instante de tempo. Podemos perceber o efeito do amortecimento sobre o sistema pela diminuição da

amplitude do movimento e da velocidade dele ao longo do tempo.

Sistema Massa-Mola Forçado

Suponha que a força externa F_{ext}, que em (4.19) era $-mg$ dependesse do tempo t e fosse periódica, como um pistão que forçasse a mola de um lado para o outro. Teríamos então um **Sistema Massa-Mola Forçado**, como na Figura 4.6, no qual $F_{ext} = F_0 \cos \omega t$, sendo F_0 é amplitude da força periódica e ω é a frequência dela. Nesse sistema ocorre um movimento chamado **batimento**, que é uma oscilação com uma frequência cuja amplitude também oscila [20].

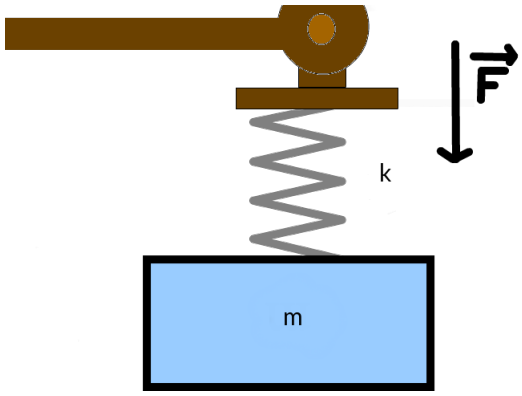


Fig. 4.6: Imagem meramente ilustrativa de um sistema massa-mola forçado

A solução analítica geral desse caso se dá por

$$u(t) = c_1 \cos \omega_0 t + c_2 \operatorname{sen} \omega_0 t + u_p(t) \tag{4.22}$$

$$\text{com } u_p(t) = t^s [A \cos \omega t + B \operatorname{sen} \omega t] \tag{4.23}$$

sendo c_1 e c_2 constantes da solução geral e ω_0 a frequência natural do sistema. O termo $u_p(t)$ é a solução particular (também chamada solução transiente [4]) do problema na qual s é "o menor inteiro não negativo que garanta que nenhuma parcela de $u_p(t)$ seja solução da equação homogênea correspondente [...]" [20] e A e B são coeficientes da solução que podem ser determinados pelo método de coeficientes a determinar, como pode ser visto em [20].

O código a seguir resolve, matematica e numericamente, o mesmo sistema massa-mola apresentado no exemplo anterior, porém agora com uma força externa agindo sobre o mesmo. A força externa F_{ext} é descrita por

$$F_{ext} = \cos \omega_0 t$$

```
class EqMassaMola(edo):
    m = gamma = k = freq = None

    def setM(self, m):
        self.m = m

    def getM(self):
        return self.m

    def setGamma(self, gamma):
        self.gamma = gamma

    def getGamma(self):
        return self.gamma

    def setK(self, k):
        self.k = k
```

```
18
19      def getK(self):
20          return self.k
21
22      def setFreq(self, freq):
23          self.freq = freq
24
25      def getFreq(self):
26          return self.freq
27
28      def __init__(self, dimension, m, gamma, k, freq):
29          # Inserindo dimensao do sistema
30          edo.__init__(self, dimension)
31
32          # Inserindo parametros
33          self.setM(m)
34          self.setGamma(gamma)
35          self.setK(k)
36          self.setFreq(freq)
37
38      def evaluate(self, sliceT, sliceY):
39          # Criacao de vetor para retorno
40          valores = np.zeros(2)
41
42          # Equacao para velocidade
43          valores[0] = sliceY[1]
44
45          # Equacao para posicao
46          valores[1] = ((np.cos(self.freq * sliceT) - self.gamma * sliceY[1] - self.k * sliceY
        [0]) / self.m)
47
48          return valores
49
50  def main():
51      # Numero de passos
52      N = 10000
53
54      # Define o intervalo maximo em t
55      tMax = 100.0
56
57      # Coeficientes da EDO
58      m = 1.0          # Massa
59      gamma = 0.125    # Coeficiente de amortecimento
60      k = 1.0          # Constante elastica
61      freq = 2.0       # Frequencia da aplicacao da forca
62
63      # Valores iniciais
64      To = 0.0  # Instante inicial
65      Vo = 0.0  # Velocidade inicial
66      Po = 2.0  # Posicao inicial
67
68      EDO = EqMassaMola(2, m, gamma, k, freq)
69
70      Yo = np.array([Vo, Po])
71
72      # chama a funcao para execucao do Runge-Kutta
73      T, Y = RK4(EDO, tMax, N, To, Yo)
74
75      # Plota os vetores
76      plotar(T, Y)
77
78  def intro():
79      # Apresenta o algoritmo
80      print "Resolve a seguinte EDO"
81      print "s'' + 0.125s' + s = 0"
82
83      # Faz as inicializacoes do programa pela funcao main
84      main()
85
86  intro()
```

```
87 exit()
```

Listing 4.8: Código da resolução numérica de um problema envolvendo um sistema massa-mola forçado

cujo resultado é exibido na Figura 4.7

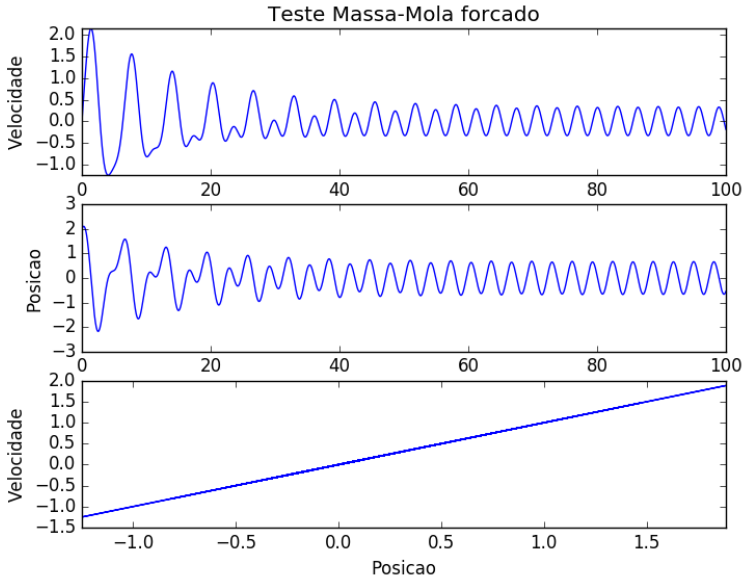


Fig. 4.7: Gráficos da posição e da velocidade da massa do sistema ao longo do tempo, seguido de um gráfico da velocidade pela posição

O primeiro gráfico descreve a velocidade v ao longo do tempo t e o segundo descreve a posição y da massa ao longo de t também. Podemos perceber a ação da solução transiente até cerca de 50 unidades de tempo em ambos os gráficos, sendo que permanece depois apenas a solução geral (também chamada solução estado estacionário - [4]).

O terceiro gráfico descreve a velocidade $v(t)$ em função da posição $y(t)$ da massa. O sinal da velocidade indica o sentido dela.

Capítulo 5

Ondulatória

5.1 Definição

Ondas são pertubações que se propagam através do espaço e/ou meio material, transportando energia, por meio de oscilações periódicas e com velocidade constante [18]. Elas estão continuamente presentes no cotidiano humano, como nas ondas sonoras, micro-ondas, as ondas marítimas e a própria luz.

As ondas possuem propriedades físicas, características e podem ser classificadas quanto a sua natureza, direção de oscilação e de propagação.

5.2 Classificações Quanto à Natureza

Com relação à natureza das ondas, elas podem ser classificadas em três tipos, principalmente [14]

1. **Mecânicas:** são as ondas que se movimentam apenas em meio material e obedecem às Leis de Newton. Alguns exemplos dessas ondas são as sonoras, marítimas e sísmicas.

2. **Eletromagnéticas:** ondas criadas pela interação entre um campo elétrico e um campo magnético. Ao contrário das ondas mecânicas, essas não necessitam de meio material para a sua propagação, podendo se movimentar até mesmo no vácuo (no qual todas as ondas eletromagnéticas alcançam a velocidade da luz). Alguns exemplos dessas ondas são a própria luz, as ondas de rádio e as ondas emitidas por aparelhos micro-ondas.

3. **Materiais:** estão relacionadas aos elementos mais básicos da matéria, como prótons e elétrons. Por causa disso, são mais estudadas em laboratórios.

5.3 Classificações Quanto à Direção de Oscilação

Com relação à direção que as ondas oscilam (principalmente as mecânicas), elas podem ser classificadas em dois tipos,

1. **Transversais:** são as ondas nas quais os elementos do meio no qual ela propaga se movimentam perpendicularmente a sua direção de propagação. Por exemplo, se um indivíduo amarrar a ponta de um barbante longo em um objeto fixo e sacudir a outra ponta para cima e para baixo rapidamente, uma onda será transmitida ao longo do barbante em formato de

pulso. As partículas da corda subirão e descerão, movimentando-se perpendicularmente ao movimento da onda, que segue a estrutura do barbante.

2. **Longitudinais:** são as ondas nas quais os elementos do meio em que ela se propaga se movimentam no mesmo sentido que a onda se propaga. Um exemplo desse caso são as ondas sonoras. Elas são transmitidas por pulsos longitudinais no ar. Por exemplo, em um tubo com ar, se for colocado um êmbolo em uma de suas extremidades e este for movimentado bruscamente para frente e, em seguida, para trás, será gerada uma onda sonora e na estrutura do meio em que ela se propaga (o próprio ar) seria possível perceber que o movimento das moléculas se dá no sentido do pulso da onda sonora [14].

5.4 Classificações Quanto à Direção de Propagação

Com relação à direção que as ondas se propagam (o comportamento das ondas em diferentes dimensões), elas podem ser classificadas em três tipos [31]

1. **Unidimensionais:** são as que se propagam apenas em uma direção, como pulsos em cordas.

2. **Bidimensionais:** são as que se propagam em duas direções, em um plano, como as vibrações em uma folha de papel chacoalhada ou as vibrações em um espelho d'água após alguém atiram uma pedra sobre ele.

3. **Tridimensionais:** são as que se propagam no espaço, em três direções, como as ondas sonoras no ar.

5.5 Características Gerais das Ondas

Em decorrência do fato de que as ondas oscilam periodicamente, algumas características gerais do fenômeno podem ser estudadas sobre este fato [14]

- **Comprimento:** intervalo no espaço, paralelo à direção da propagação da onda, em que se inicia e termina uma onda (oscilação completa), se iniciando outra logo em seguida;

- **Amplitude:** deslocamento máximo e mínimo efetuado por um elemento do meio em que a onda se propaga quando a onda passa por ele. Por exemplo, a altura máxima e mínima alcançada por uma partícula de uma corda na qual foi efetuado um pulso quando o pulso passa pela partícula;

- **Período:** espaço de tempo necessário para a realização de uma oscilação completa;

- **Frequência:** número de oscilações completas por unidade de tempo (segundos no Sistema Internacional de Unidades);

- **Fase:** corresponde ao intervalo no espaço em que um elemento do meio em que a onda se propaga alcança suas amplitudes máxima e mínima, como uma partícula de uma corda alcançando a crista e o vale de um pulso emitido na mesma corda (ou seja, realizando uma oscilação completa).

5.6 Propriedades Físicas das Ondas

Por ser um fenômeno físico, é de se esperar que a ondulatória obedeça as leis da Física, o que lhe atribui algumas propriedades físicas

- **Superposição:** No estudo de ondas, há também a questão da superposicão das ondas lineares, sendo que quando ondas se superpõem elas se somam algebricamente, gerando uma onda resultante ou uma onda total, sem se afetar mutuamente. Essa propriedade é consequência direta do princípio da superposição, que afirma que a ocorrência simultânea de efeitos individuais leva à formação de um efeito total formado pela soma dos individuais. Isso ocorre por exemplo, em lagos nos quais as ondas provocadas por embarcações se superpõem continuamente. Isso ocorre também em uma rua com tráfego intenso, na qual o barulho dos automóveis se superpõem uns aos outros, chegando ao mesmo tempo aos ouvidos das pessoas que ali estão [14];

- **Interferência:** ocorre como consequência da superposição de ondas e depende da fase relativa das duas ondas. Ou seja, se elas estão alinhadas (em fase) e com amplitude igual, seu deslocamento conjunto ocorre em dobro. Se elas não estão alinhadas (fora de fase) e, novamente, com amplitude igual, seu deslocamento é nulo, pois elas se cancelam. Em consequência desse fenômeno, se duas ondas de mesma amplitude e comprimento se movimentarem em um meio, uma contra a outra, a onda resultante será chamada estacionária, pois não se move para nenhum lado e possui pontos que são fixos [14];

- **Reflexão:** quando uma onda está se propagando por um meio e se depara com outro meio de características estruturais diferentes, ela sofre interferência em si mesma e é refletida. Por exemplo, se um barbante está preso a um ponto imóvel e um pulso é emitido nele, quando o mesmo chegar ao ponto fixo, a terceira Lei de Newton agirá, criando um pulso de mesma amplitude, comprimento mas sentido e posição contrários. Já se o ponto em que o barbante estiver preso for móvel, a reflexão ocorrerá da mesma forma, mas a posição do pulso refletido será a mesma do pulso original [14];

- **Refração:** quando a onda entra em contato com outro meio material, sua direção e velocidade variam, mas sua frequência se mantem [22];

- **Difração:** quando ocorre o espalhamento de ondas após uma onda entrar em contato com uma fenda com tamanho equivalente ao seu comprimento [22];

- **Dispersão:** quando uma onda se separa em outras de diferentes frequências [22];

- **Vibração:** ondas produzidas através de vibrações de objetos, como cordas de violão ou tubos de flauta [22];

- **Polarização:** como já vimos anteriormente, em uma onda transversal as oscilações são em alguma direção perpendicular à direção de propagação da onda. Dessa forma, uma onda pode oscilar para baixo, para a direita, para a esquerda, etc.... A fixação da direção de oscilação de uma onda, fazendo com que ele oscile apenas em uma direção só, é chamada polarização [8].

- **Ressonância:** suponha, por exemplo, fossem criados vários pulsos indo para a direita em um barbante e este estiver preso em sua extremidade direita. Quando o primeiro pulso alcançasse a extremidade e fosse refletido, começaria a voltar pelo barbante e, pelo

princípio da superposição, geraria interferência no contato com os outros pontos. Esse pulso em questão (e os demais por consequência) chegaria à extremidade esquerda e seria refletido e, em seguida, novas interferências seriam geradas. Em certas frequências, a interferência entre os pulsos gera ondas estacionárias (que permanecem no mesmo lugar, sem locomoção) por ressonância [14].

5.7 As Ondas Transportam Energia, Não Matéria

É necessário saber e entender que uma onda não transporta matéria, somente energia. Quando uma ginasta olímpica cria um pulso em uma fita, por exemplo, fornece-se energia para o movimento da fita, que é transportada ao longo dela. Essa energia se apresenta em forma de energia cinética e energia potencial elástica.

É possível se observar a energia cinética através do movimento oscilatório de cada parte da fita, quando cada partícula vai até a crista e desce até o vale. Já a energia potencial elástica se deve ao fato de que cada parte da fita é esticada e comprimida (como uma mola) enquanto a onda passa por ela. A energia é transportada ao longo da fita pelo trabalho das forças de tensão que ligam as partes da fita.

5.8 A Equação Geral de Deslocamento da Onda

Sabe-se que uma onda cujos pontos não se locomovem espacialmente é chamada onda estacionária [9]. Já uma onda que se locomove enquanto se propaga é chamada senoidal. Isso pode ser justificado pela equação geral de deslocamento de uma onda

$$y(x,t) = y_m \operatorname{sen}(kx - \omega t)$$

na qual, t representa o tempo, y_m representa o deslocamento paralelo ao eixo y dos elementos do meio onde a onda se propaga (amplitude), x é a posição do referido elemento, ω representa a frequência angular e o elemento k representa o número de onda.

Nessa equação, percebe-se a existência de uma função seno: $\operatorname{sen}(kx - \omega t)$, que é responsável por expressar matematicamente as oscilações das ondas em sua locomoção e propagação, já que é uma função periódica, assim como a função cosseno. Fato observável na representação gráfica das funções seno e cosseno a seguir, juntamente com o código utilizado para gerá-las

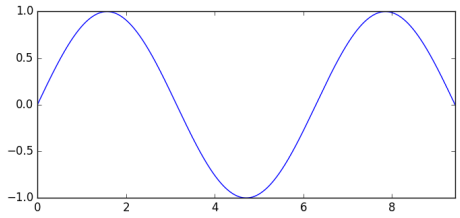


Fig. 5.1: Gráfico da função $y(x) = \operatorname{sen}(x)$

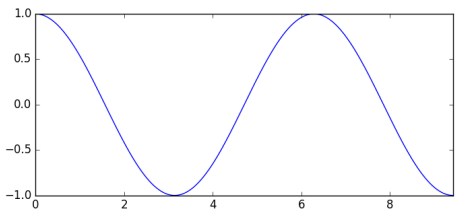


Fig. 5.2: Gráfico da função $y(x) = \cos(x)$

Dessa forma, pode-se entender o motivo do nome senoidal.

```
#Codigo para plotagem do seno

import numpy as np
import matplotlib.pyplot as plt

pi = 3.14159265359
xvals = np.arange(0, 3*pi, 0.01)
yvals = np.sin(xvals)
plt.plot(xvals, yvals)
plt.axis([0, 3*pi, -1, 1])
plt.show()

```

```
#Codigo para plotagem do cosseno

import numpy as np
import matplotlib.pyplot as plt

pi = 3.14159265359
xvals = np.arange(0, 3*pi, 0.01)
yvals = np.cos(xvals)
plt.plot(xvals, yvals)
plt.axis([0, 3*pi, -1, 1])
plt.show()

```

5.9 A Equação Geral da Onda

O comportamento das ondas pode ser estudado através da equação geral da onda

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{v}\frac{\partial^2 u}{\partial x^2} \tag{5.1}$$

na qual, v representa a velocidade de propagação da onda e t representa o tempo. Essa equação foi obtida por meio da aplicação da segunda lei de Newton ao movimento dos elementos do meio em que a onda se propaga, como os elementos de uma fita, por exemplo. Essa equação é uma equação diferencial geral parcial (a ser estudada mais a fundo no capítulo 6), o que indica que existem taxas que se relacionam no fenômeno.

5.10 Algumas Aplicações do Estudo de Ondas

Como dito anteriormente, o estudo do fenômeno físico das ondas possibilita um melhor bem-estar para a humanidade. Ele pode ser utilizado, por exemplo, nas áreas:

- **Acústica:** o estudo apropriado das ondas sonoras facilita o alcance do som de uma orquestra até a sua plateia, por exemplo, contribuindo para as Artes e Cultura;
- **Sísmica:** o estudo sobre as ondas sísmicas podem facilitar a previsão de terremotos e a evolução da tecnologia de prédios resistentes a tremores de alta intensidade;
- **Estruturas:** pontes usadas em grandes estradas, por exemplo, costumam estar em contato constante com ondas marítimas ou fluviais, além do próprio vento. Se os cálculos dos engenheiros responsáveis pela construção das mesmas não assegurar a estabilidade da ponte, a mesma pode ruir pelo fenômeno da ressonância ou pelo impacto constante das ondas.

Capítulo 6

Equações Diferenciais Parciais e a Equação da Onda

6.1 Definição de uma Equação Diferencial Parcial e Alguns Exemplos

Uma equação diferencial parcial (EDP) é baseada numa função como

$$f(x_1, x_2, ..., y, \frac{dy}{dx_1}, ..., \frac{dy}{dx_n}, \frac{d^2y}{dx_1 dx_1}, ..., \frac{d^2y}{dx_1 dx_n}, ...) = c \tag{6.1}$$

na qual c é uma constante, ou seja, em uma função de y e suas derivadas parciais. Por meio dessa definição podemos perceber, assim como já tinha sido dito na seção 3.2, que uma EDP possui mais de uma variável independente. Uma EDP baseada numa função como 6.1, mas com $c = 0$, é dita equação diferencial parcial linear [7]. Por exemplo, uma EDP linear de segunda ordem seria na forma [25]

$$Ay_{x_1x_1} + By_{x_1x_2} + Cy_{x_2x_2} + Dy_{x_1} + Ey_{x_2} + F = 0 \tag{6.2}$$

Alguns exemplos de EDP's são:

- $u_t = a^2 u_{xx}$, a equação unidimensional do calor (sendo $u_t = a^2(u_{xx} + u_{yy})$ a sua versão bidimensional) que determina a distribuição do calor através da(s) dimensão(ões) de um corpo ao longo do tempo;
- $u_{xx} + u_{yy} = 0$, a equação bidimensional de Laplace (sendo $u_{xx} + u_{yy} + u_{zz} + = 0$ sua versão tridimensional), que serve de modelo para as funções potenciais gravitacionais e elétricas, por exemplo [27];
- $u_{tt} = a^2 u_{xx}$, a equação unidimensional da onda (sendo sua versão bidimensional $u_{tt} = a^2(u_{xx} + u_{yy})$ e a tridimensional, $u_{tt} = a^2(u_{xx} + u_{yy} + u_{zz})$), que descreve o comportamento das ondas das mais variadas naturezas (e que é peça-chave desse projeto) [21].

6.1.1 Tipos de Equações Diferencias Parciais

Equações Diferenciais Parciais Elípticas

Considerando uma equação como (6.2), ela é dita elíptica quando $B^2 - AC < 0$ [7]. Podemos perceber que equações como a de Laplace, citada na seção anterior, que são elípticas, seguem o

formado da equação da cônica elipse

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = c^2$$

ou da quádrica elipsóide,

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = d^2$$

dependendo do número de dimensões consideradas.

Equações Diferenciais Parciais Parabólicas

Ainda considerando uma equação como (6.2), caso $B^2 - AC = 0$, essa equação é do tipo parabólica [7]. Podemos ver que equações como a do calor, apresentada na seção 6.1, tem o mesmo formato que a cônica parábola [30]

$$y^2 = 4px$$

ou da quádrica parabolóide,

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = \frac{z}{c}$$

dependendo do número de dimensões consideradas [29].

Equações Diferenciais Parciais Hiperbólicas

Mais uma vez considerando a equação (6.2), se $B^2 - AC > 0$ então a equação é do tipo hiperbólica, como a equação da onda citada em (6.1). Assim como as equações anteriores, as EDP's anteriores são similares à equação da cônica hipérbole

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = c^2 \tag{6.3}$$

ou da quádrica hiperboloide,

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = d^2$$

6.2 Resolução de Equações Diferenciais Parciais

As EDP's podem ser resolvidas ou analítica ou numericamente. Dentre os métodos de resolução analítica encontramos:

- separação de variáveis;
- método das características;
- mudança de variáveis;

entre outras opções, que podem ser encontradas em [7].

Contudo, esses métodos nem sempre são suficientes para resolver os problemas que envolvem as EDP's, sendo necessário então recorrer ao métodos numéricos, entre eles os mais comuns

- método das diferenças finitas;
- método dos elementos finitos;
- método dos volumes finitos [7];

dos quais usaremos apenas o primeiro mais a frente. Também utilizaremos, por trabalharmos com a equação da onda neste projeto, o método de traçamento de raios.

6.3 A Equação da Onda

6.3.1 Em uma Dimensão Espacial

A equação da onda é uma equação diferencial parcial hiperbólica que segue o seguinte formato para casos unidimensionais sem fonte de emissão para a onda:

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{v}\frac{\partial^2 u}{\partial x^2} \tag{6.4}$$

e, para casos onde há fonte de emissão da onda:

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{v}\frac{\partial^2 u}{\partial x^2} + f(x,t) \tag{6.5}$$

na qual v representa a velocidade de propagação da onda e $f(x,t)$ é uma função que representa a fonte de emissão da onda. Essas equações (ou alguma generalização delas) definem o comportamento de uma onda, seja ela de natureza acústica, aquática, eletromagnética ou sísmica, por exemplo [4].

Durante a modelagem de um problema físico envolvendo a propagação de ondas é necessário, além, claro, da equação da onda, a definição de condições de fronteira (também conhecidas como condições de contorno) e uma condição inicial. Supondo uma corda de comprimento L, fixada nos pontos $x = 0$ e $x = L$, tem-se como condições de fronteira

$$u(0,t) = u_a,\ u(L,t) = u_b$$

e como condições iniciais

$$u(x,0) = f(x),\ 0 \leq x \leq L \text{ para a posição inicial}$$

$$\text{e } u_t(x,0) = g(x),\ 0 \leq x \leq L \text{ para a velocidade inicial}$$

Se $u_a = u_b = 0$, por exemplo, temos que a onda refletirá nas bordas do domínio considerado.

A solução $u(x,t)$ da EDP descreve o deslocamento vertical da corda no ponto x, em um instante t de tempo [4].

6.3.2 Em duas Dimensões Espaciais

Imagine uma membrana elástica, como uma membrana de tambor ou até mesmo o tímpano que se encontra no nosso ouvido. Nessa membrana, cada ponto (x,y) está ligado ao seus vizinhos de modo que, se a membrana é excitada ela vibra ou de acordo com a equação

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{v}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2}\right) \tag{6.6}$$

ou, se a excitação for realizada por uma fonte, pela equação

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{v}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2}\right) + f(x,y,t) \tag{6.7}$$

Se tal membrana, ou o meio pelo qual a onda se propaga, for homogêneo, v é constante. Do contrário, v é uma função em x e y.

A ligação de cada ponto (x,y) aos seus vizinhos, de modo que qualquer excitação na membrana gere oscilações por ela, pode ser imaginada como um sistema massa-mola infinito (ou quase infinito). Para isso, basta que assumamos cada ponto como uma massa infinitesimal m e cada ligação como uma pequena mola de constante elástica k. As pequenas molas são os elementos que levam à vibração.

Assim como o caso unidimensional, o modelo da propagação de ondas em meios bidimensionais necessita de uma condição inicial (quando a baqueta toca a membrana de um tambor de uma bateria) e de condições de contorno (a membrana tem seu contorno ligado a um suporte). Podemos definir as condições de contorno como [11]

$$u(0,y,t) = u(a,y,t) = 0, \qquad 0 \leq x \leq a,\ t \geq 0, \tag{6.8}$$

$$u(x,0,t) = u(x,b,t) = 0, \qquad 0 \leq y \leq b,\ t \geq 0. \tag{6.9}$$

e as condições iniciais como

$$u(x,y,0) = f(x,y), \tag{6.10}$$

$$u_t(x,y,0) = g(x,y) \tag{6.11}$$

Capítulo 7

O Método de Diferenças Finitas

O método de diferenças finitas (MDF) vem com o objetivo principal de aproximar derivadas de funções, servindo também na resolução de EDO's e EDP's. Essa aproximação se dá discretizando-se o domínio no qual a função será aplicada e combinando valores de uma função (que estejam próximos uns dos outros) através de certas fórmulas, usando determinados pesos em conjunto aos valores da função [12].

7.1 Diferenças Finitas

Uma diferença finita é definida por um quociente de diferenças como

$$\frac{f(x+\Delta x)-f(x)}{\Delta x} \tag{7.1}$$

Pode-se perceber a similaridade da expressão acima com

$$\frac{f(x_0+h)-f(x_0)}{h} \tag{7.2}$$

que, como sabemos, tende para $f'(x_0)$ quando $h \to 0$. Então, se fizermos $x = x_0$, $\Delta x = h$ e $\Delta x \to 0$, temos que a equação (7.1) é uma aproximação de $f'(x_0)$ e, sendo assim, podemos chamá-la **diferença finita progressiva** de ordem um (sendo ordem referente à acurácia da fórmula em relação à função analítica) [15]. Por meio disso podemos perceber o motivo pelo qual se utiliza as diferenças finitas para a aproximação de derivadas.

7.2 Fórmulas de Diferenças Finitas

Como vimos na seção anterior, podemos obter uma fórmula para diferenças finitas progressivas de ordem um. Além desse tipo, existem dois outros tipos de fórmulas de diferenças finitas [6]

- regressivas, *e. g.*

$$f'(x_0) \cong \frac{f(x_0)-f(x_0-\Delta x)}{\Delta x}, \text{ também de ordem um;} \tag{7.3}$$

- centrais, *e. g.*

$$f'(x_0) \cong \frac{f(x_0+\Delta x)-f(x_0-\Delta x)}{\Delta x}, \text{ ainda de ordem um} \tag{7.4}$$

Cada uma delas pode ser a melhor para uma determinada situação.

Para a afirmação a seguir, assuma que $f(x_i + j\Delta x) = u_{i+j}$, sendo i a posição do *array* u em que o MDF está centrado, j um valor inteiro para referenciar um vizinho do centro e que u_w representa $f(x_w)$. Dependendo das derivadas requeridas para o método de diferenças finitas, o número de chamadas à função bem como os coeficientes que as acompanham, além dos denominadores das expressões, serão diferentes. Se representarmos graficamente essas fórmulas, perceberemos que elas possuem formatos específicos que damos o nome de *stencil* (ou estêncil, em português). Por exemplo, as fórmulas de diferenças finitas de ordem dois para a segunda derivada $f''(x)$ em uma dimensão (usadas na aplicação do MDF sobre a equação da onda) são:

- regressivas
$$\frac{u_i - 2u_{i-1} + u_{i-2}}{(\Delta x)^2}; \tag{7.5}$$
- progressivas
$$\frac{u_{i+2} - 2u_{i+1} + u_i}{(\Delta x)^2}; \tag{7.6}$$
- centrais
$$\frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} \text{ [6].} \tag{7.7}$$

7.3 As Diferenças Finitas e as Dimensões

Se um dado problema envolve uma função u, com n variáveis (ou seja, n dimensões), e suas respectivas derivadas, é necessário que o método de diferenças finitas seja aplicado de acordo com as variáveis da função u. Ou seja, se para u com uma variável tínhamos a indexação do *array* (o que representa os valores da função) como sendo u_{i+j}, como explicado na seção anterior, agora teremos $u_{i_1+j_1,i_2+j_2,\ldots,i_n+j_n}$ como representação para a indexação do *array* n-dimensional [10].

Nessa representação, $i_1, i_2, \ldots, i_n$ representam as posições do *array* u e $j_1, j_2, \ldots, j_n$ representam os deslocamentos dessas posições até seus vizinhos em uma determinada dimensão. A dimensão, por sua vez, é dada pela ordem dos i's na indexação. Para tentar deixar mais claro, podemos dar como exemplo a função u com [10]

- duas dimensões
$$u(x_i, t_j) \equiv u_{i,j}; \tag{7.8}$$
- três dimensões
$$u(x_i, y_j, t_k) \equiv u_{i,j,k} \tag{7.9}$$

7.4 Como o Método é Aplicado?

A maneira como o método é aplicado pode variar um pouco dependendo da finalidade para o qual ele é utilizado. Contudo, ele segue etapas gerais que enunciaremos e explicaremos aqui.

7.4.1 Discretização

Como sabemos, é impossível que nossos computadores consigam representar um domínio contínuo, visto que, por exemplo, entre dois números naturais existem infinitos números reais. Dessa forma, torna-se necessária a discretização do domínio em que o problema que desejamos resolver está inserido.

Imagine o domínio desse problema (considerando aqui o caso unidimensional para a propagação de ondas) como uma cama retangular de comprimento a e largura b. Essa cama representa o espaço-tempo unidimensional para o problema. Sendo assim, temos que a variável que percorre o comprimento da cama seja x e a que percorre a largura seja t. Então temos que a cama, contínua, é dado por

$$0 \leq x \leq a$$
$$0 \leq t \leq b$$

Agora imagine que precisemos que essa cama elástica seja representada numerica ou computacionalmente. Para tal, dividimos seu comprimento e sua largura em espaços de mesmo tamanho, que podem ser iguais para o comprimento e a largura da cama ou diferentes para cada um desses. Os tamanhos desses espaços também podem ser variáveis (dependendo do problema) para cada dimensão dessa cama. Teríamos então um domínio discreto como o da seguinte imagem

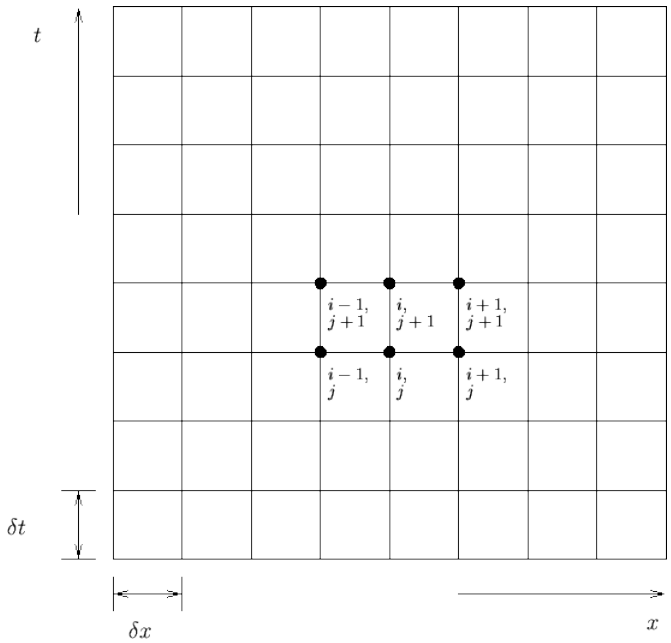


Fig. 7.1: Domínio discretizado [17]

na qual δx e δt são os tamanhos dos espaços criados nas dimensões da cama elástica, que é o nosso domínio.

Como podemos ver na Figura 7.1, a discretização do domínio leva à criação de pontos ao longo dele. Esses pontos são indexados na imagem por i (referente à variável x) e j (referente à variável t). Dessa forma temos que o domínio, agora discreto, do nosso problema é dado por

$$0 \leq x_i \leq a$$
$$0 \leq t_j \leq b$$

.

7.4.2 Cálculo

Tendo sido feita a discretização do domínio do problema, devemos partir para a parte dos cálculos, a ser realizada por meio das fórmulas de diferenças finitas apropriadas. Por exemplo, caso o problema fosse avaliar a função $f''(x)$ no domínio que definimos, deveríamos aplicar uma das fórmulas (7.5), (7.6) ou (7.7) (dependendo de onde o método esteja focado no domínio do problema) para todos os pontos (x_i, t_j) discretizados. Assim, obteríamos uma malha com todos os valores da derivada para todos os pontos.

Capítulo 8

O Método de Traçamento de Raios

8.1 O Funcionamento e a Finalidade

O traçamento de raios é uma técnica que, basicamente, consiste em modelar a propagação de ondas através de um *meio* elástico utilizando ferramentas chamadas *raios*.

No caso do traçamento tratado nesse trabalho, consideraremos um modelo de propagação de ondas sísmicas em uma geologia acamadada. Esses raios são emitidos por uma fonte na superfície do meio e traçados, de acordo com uma lei matemática (a ser mostrada na Seção 8.3), até que alguma camada do meio seja encontrada. Nesse momento, aplica-se a lei de Snell sobre o raio de acordo com a propriedade da camada (se ela é refletora ou transmissora) e inicia-se novamente o traçamento, como se a última camada encontrada fosse a nova fonte. Isso ocorre até que a superfície seja alcançada. Na superfície podem existir receptores que podem ser atingidos pelos raios traçados [10].

Podemos ter alguma ideia do procedimento descrito acima com a ajuda da Figura 8.1. A fonte emite os raios que atravessam as camadas transmissoras até alcançar a camada refletora previamente definida. A partir daí, os raios sobem até os receptores.

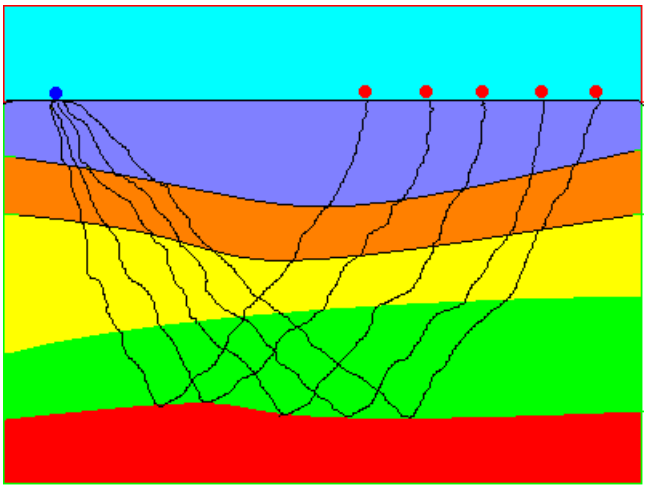


Fig. 8.1: Imagem meramente ilustrativa sobre o traçamento de raios. O ponto azul representa uma fonte e os pontos vermelhos, receptores. As camadas da subsuperfície são ilustradas pelas cores lilás, laranja, amarelo, verde e vermelho, respectivamente, sendo essa última a camada refletora dos raios e as demais, transmissoras. As linhas pretas que ligam a fonte aos receptores são os raios.

Existem vários outros tipos de traçamento de raios, podendo-se ter todos os raios emitidos pela fonte refletidos em um único ponto de alguma camada do meio ou todos os raios emitidos refletindo para um único receptor. No geral, todos esses tipos de traçamento de raios visam possibilitar a análise da estrutura de um determinado meio usando-se as propriedades da propagação de ondas para isso [10].

8.2 O Meio

O meio a ser vasculhado pelo traçamento de raios pode ser formado ou não por camadas. Para esse trabalho, assumimos um meio formado por camadas, sendo essas dispostas uma por cima da outra e com a condição de que uma camada pode ter contato apenas com, no máximo, duas outras. Além disso, consideramos a atmosfera encontrada acima da primeira camada como uma dessas também. Esse meio é uma adaptação do que seria um meio **acamadado** no ramo da sismologia [10].

Cada uma das camadas do meio possui como propriedades principais sua espessura, os formatos e posições de suas interfaces (que separam uma camada de outra) e a sua velocidade sísmica, que é "a taxa pela qual uma onda sísmica viaja através de um meio, que é, a distância dividida pelo tempo de trânsito" [2], sendo o tempo de trânsito a "duração da passagem de um sinal a partir de uma fonte através da Terra e voltando para um receptor" [3].

Na sismologia, a determinação dessas propriedades das camadas é importante para que se possa cogitar sobre os materiais que formam cada camada e, dependendo da aplicação do método, como esses materiais poderiam ser alcançados fisicamente.

8.3 O Raio

Modelar campos de ondas computacionalmente tem um alto custo. Para contornar esse problema, utilizamos a teoria dos raios ao nosso favor. Um raio é como uma parte infinitesimal de uma frente de onda ao longo do tempo, sendo também perpendicular a esta. Um conjunto de raios que simulam partes da mesma frente de onda tomam o formato dela no decorrer do tempo, o que seria possível observar se a propagação da frente de onda fosse simulada por completo. Utilizando apenas pequenas partes (raios) de um domínio muito maior (frente de onda), o traçamento de raios consegue superar a simulação completa no que tange ao custo computacional.

Como dito, o raio simula uma pequena parte de uma frente de onda **ao longo do tempo**. Ora, mas o que significa a expressão explicitada? Quer dizer que cada ponto do corpo do raio é a posição da parte infinitesimal da frente de onda representada no tempo. Para tentar entender essa questão mais facilmente, veja a Figura 8.2

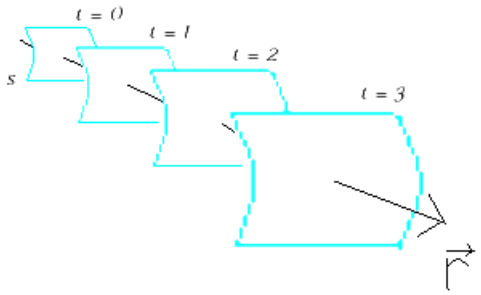


Fig. 8.2: Propagação de uma parte infinitesimal s de uma frente de onda, ao longo do tempo t, na direção de um vetor $\vec{r}$. A trajetória do vetor $\vec{r}$ ao longo do tempo forma o raio.

Imagine que o vetor $\vec{r}$ é sempre perpendicular à frente de onda e a segue. Se traçássemos os pontos de sua trajetória ao longo do tempo, teríamos então um raio e, por ele, a forma como a frente de onda propagou no decorrer do tempo.

8.3.1 Uma Brevíssima Visão Sobre a Teoria dos Raios

A teoria dos raios possui se baseia numa teoria matemática muito complexa e densa, em um nível que foge ao escopo desse trabalho. Por isso, explicaremos do que se trata essa teoria da forma mais básica possível.

Assumamos ondas de alta frequência. Sabemos que a equação da onda é algo como

$$\nabla^2 u(\vec{x},t) - \frac{1}{\alpha^2}\frac{\partial^2 u(\vec{x},t)}{\partial t^2} = 0 \tag{8.1}$$

onde ∇^2 é o operador laplaciano, que representa

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{8.2}$$

para o caso de duas dimensões espaciais e $\vec{x}$ é o vetor posição em x e y. Então, se assumirmos uma solução harmônica como

$$u(\vec{x},t) = A(\vec{x})e^{-i\omega(T(\vec{x})+t)} \tag{8.3}$$

na qual ω é a frequência da onda, $A(\vec{x})$ é a amplitude de uma determinada posição (x,y) da onda e $T(\vec{x}) = c$ é uma frente de onda (o ∇T define o *raypath*, ou seja, uma linha, sempre perpendicular à frente de onda no caso dos meios isotrópicos, que define as direções que a propagação da onda assumiu [1]), e, usando as derivadas da solução, realizarmos as devidas substituições e manipulações na equação da onda 8.1 teremos que a equação resultante pode ser separada em suas partes real e imaginária [19].

A parte real,

$$\nabla^2 A - \omega^2 A|\nabla T|^2 = \frac{-A\omega^2}{\alpha^2} \tag{8.4}$$

que, ao ser dividida por $A\omega^2$ e levando-se em conta a hipótese de ondas de alta frequência ($\omega \to \infty$), leva à **equação iconal**

$$|\nabla T| = \frac{1}{\alpha^2} \tag{8.5}$$

que "descreve a propagação cinemática de ondas de alta frequência" [19] (tradução nossa). O lado direito dessa equação, $\frac{1}{\alpha^2}$, é chamado **vagarosidade** [19]. Já a parte imaginária,

$$2\omega\nabla A \cdot \nabla T + \omega A\nabla^2 T = 0 \tag{8.6}$$

ao ser dividida por ω, leva a **equação de transporte**,

$$2\nabla A \cdot \nabla T + A\nabla^2 T = 0 \tag{8.7}$$

que "pode ser usada para computar a amplitude das ondas que estão propagando" [19].

Da equação iconal (8.5), utilizando o processo aplicado em [16], obtemos o sistema de equações

$$\frac{d\vec{x}}{dT} = v(x)^2\vec{p} \tag{8.8}$$

$$\frac{d\vec{p}}{dT} = \frac{1}{v(x)}\nabla v(\vec{x}) \tag{8.9}$$

nas quais, $\vec{p}$ é chamado **vetor vagarosidade**. A solução desse sistema é algo no formato $(\vec{x}, \vec{p})$, sendo $\vec{x}$, quando o problema é resolvido computacionalmente, um *array* de alguns dos pontos pelos quais o raio passou. Já $\vec{p}$ é um *array* de algumas direções assumidas pelo raio ao longo do traçamento [16]. Digo algumas porque a solução analítica possui infinitos pontos e infinitas direções durante o traçamento, mas isso não ocorre computacionalmente.

Capítulo 9

Equação da Onda em Meios Não-Homogêneos

9.1 Resolução da Equação da Onda por Diferenças Finitas

Como já dito, o objetivo desse trabalho é a comparação de métodos que consigam resolver a equação da onda com fonte, simulando a propagação de oscilações através de meios acamadados de composição não-homogênea. Portanto, seguindo o escopo desse objetivo, apresentaremos nesse relatório apenas as simulações para o caso em duas dimensões das oscilações, como se dá no estudo da sísmica.

9.1.1 Reflexão na Borda do Domínio de Propagação da Onda

Suponhamos um meio acamadado cujas interfaces, na forma de retas que podem ser representadas por funções, se distribuem como na figura abaixo:

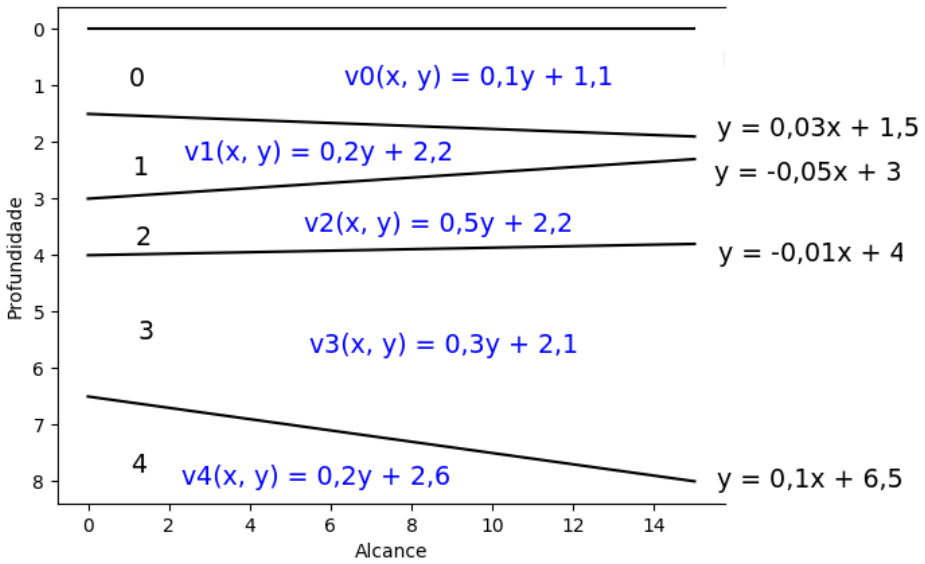


Fig. 9.1: Meio acamadado com interfaces retas (em preto) representadas por funções e camadas representadas por números de 0 a 4. As funções de velocidade de cada camada também são apresentadas na imagem

Além da distribuição das interfaces, a imagem também apresenta como se dá a propriedade de não-homogeneidade do meio através das funções de velocidades para cada camada. Em um ambiente real, a não-homogeneidade se dá através da formação das camadas por diversos tipos de materiais que estão misturados e cuja distribuição depende da posição em que se encontram e da pressão que sofrem.

As bordas do meio definido refletem as ondas. Para tal, podemos pensar o meio por onde a onda propagará como uma cama elástica, que é presa em suas bordas. Para imitar isso, usamos as condições de contorno como em (6.8) que pode ser implementado em código: imagine um *array* tridimensional (visto que o problema se dá em x, y e t) no qual, para cada posição no eixo que representa t exista um *array* xy que representa o nosso meio. Então aplicamos as condições de contorno zerando as "bordas" de cada um desses arrays.

Para que simulemos o espalhamento das oscilações precisamos de uma fonte as inicie. Para isso, podemos utilizar a *wavelet* de Ricker (também conhecida como *mexican hat wavelet*) como fonte. Essa *wavelet* é dada pela função

$$f(t) = (1 - 2\pi^2 f_M^2 t^2)^{-\pi^2 f_M^2 t^2} \tag{9.1}$$

na qual, t representa o tempo e f_M representa a frequência de pico [24]. Podemos visualisar um exemplo da *wavelet* na Figura 9.2

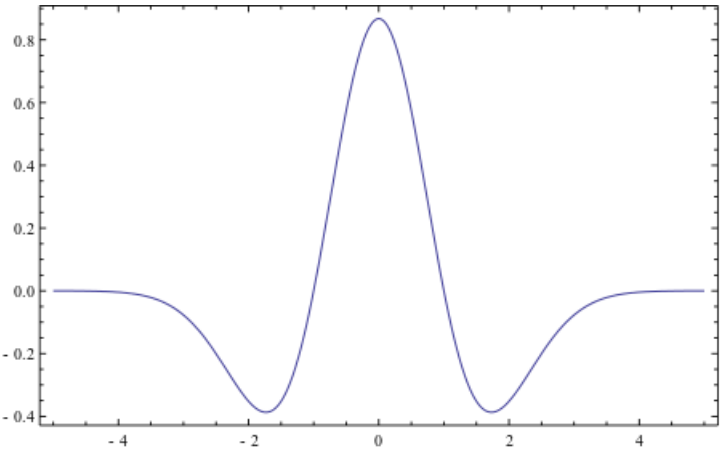


Fig. 9.2: Exemplo da *wavelet* de Ricker, por JonMcLoone [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Suponhamos também que esse meio está englobado por um sistema de coordenadas cartesianas cuja origem se encontra no extremo esquerdo superior da imagem (9.1) e cujo eixo das ordenadas aponta para baixo. Assume-se o eixo assim porque a simulação da propagação de ondas se dará por todo o meio, que é subterrâneo, e para baixo.

Prática Numérica

Utilizando o arquivo *userInput.py*, recebemos os parâmetros para a onda e o meio que o usuário deseja que a simulação siga, gerando um arquivo *data.npy*. Após isso executamos o arquivo *reflect.py*, que lerá o arquivo *data.py*, criará o meio e a onda como desejado e realizará a simulação usando o método de diferenças finitas. Por fim o programa salva um arquivo *U.npy*, que contém a simulação da propagação de ondas. Esse último arquivo pode ser usado pelo programa *snapshoter.py* para gerar *snapshots* ("fotografias") das superfícies de contorno da onda.

Os *snapshots* da Figura 9.4 foram criados para uma simulação da propagação, com no máximo 90 s, de uma onda com frequência dominante de um hertz, amplitude de um quilômetro, pico da fonte na origem do meio que tem 15 km de extensão e 9,5 km de profundidade. As velocidades do meio se distribuem de acordo com o que podemos ver na imagem 9.3.

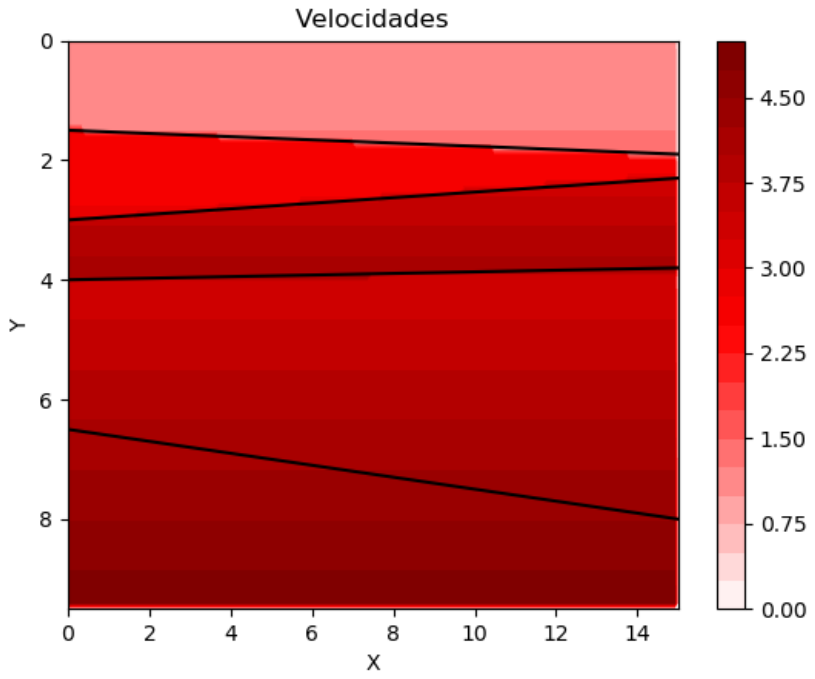
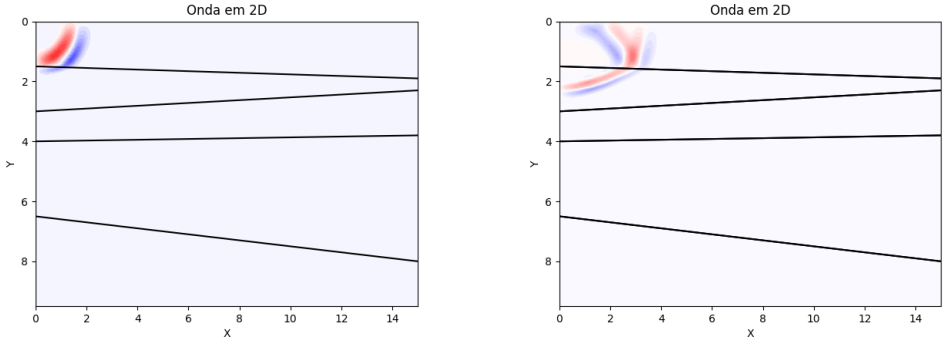
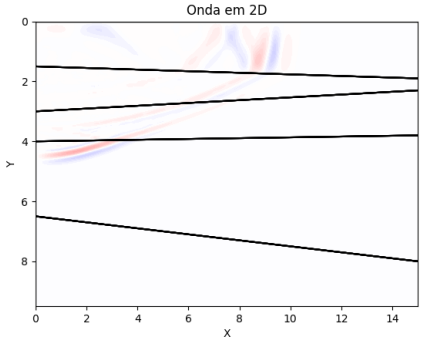


Fig. 9.3: Distribuição das velocidades no meio usado como exemplo

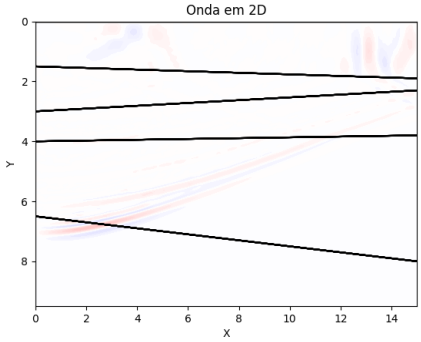


(a) 1

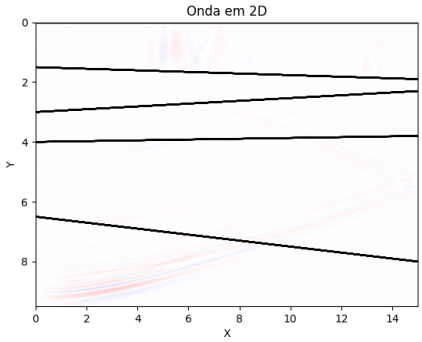
(b) 2



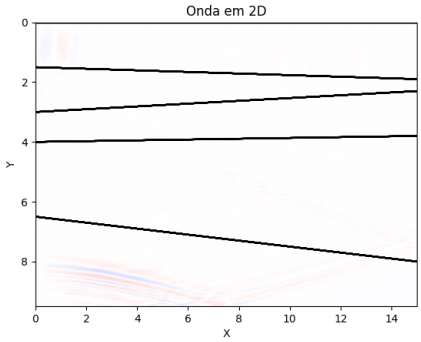
(c) 3



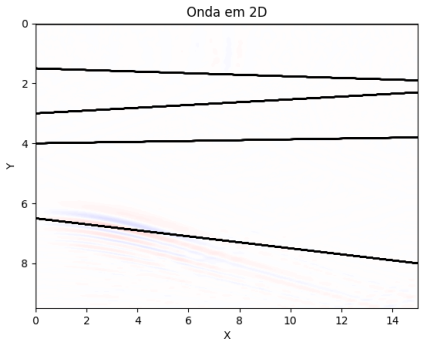
(d) 4



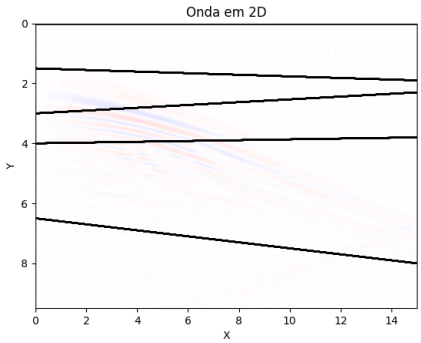
(e) 5



(f) 6



(e) 7



(f) 8

Fig. 9.4: *Snapshots* da simulação numérica da propagação de uma onda

Na Figura 9.4 podemos perceber que, como vimos no Capítulo 5, a cada vez que a onda

encontra uma outra função de velocidades, ocorre uma refração e uma reflexão da onda. Por uma falha não ocorrem refrações e reflexões significantes nas camadas 3 e 4 pois, como se pode ver na Figura 9.3, o gradiente de velocidades dessas camadas parece ser o mesmo.

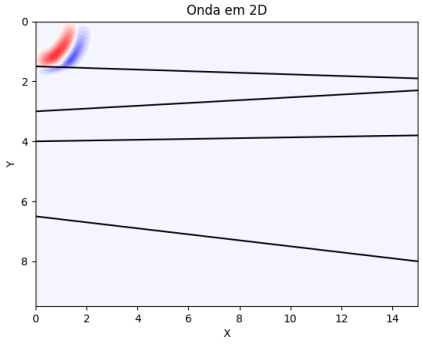
9.1.2 Domínio com Bordas "Absorventes"

Existem várias sugestões de implementações do método de diferenças finitas para a equação da onda com condições de contorno "absorventes" (ou "abertas").

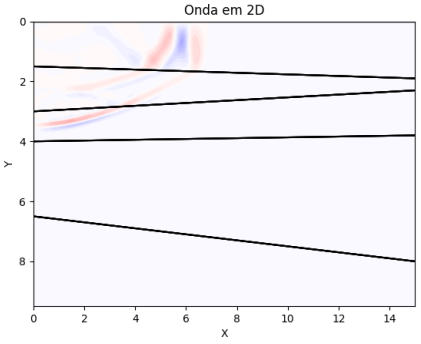
Expandiremos o domínio de simulação de propagação das ondas, mas apresentaremos nas imagem apenas a parte que nos interessa. Dessa forma, durante a sua propagação, a onda não refletirá nas bordas da imagem como na seção anterior, mas dará a impressão de sair pelas bordas dela.

Prática Numérica

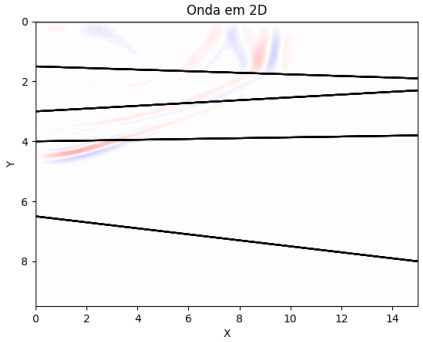
Utilizando a mesma distribuição de velocidades visto na Figura 9.3, obtemos a simulação demonstrada pelos *snapshots* da Figura 9.5. Pode-se perceber que, ao contrário do caso das bordas refletoras, no caso atual, as frentes de onda passam além das extremidades da parte do meio que foi exibida. É como se o primeiro caso fosse um corpo caindo numa bacia de água, com o tempo, a superfície da possui um comportamento ondulatório complexo, com várias ondas se sobrepondo. Já o segundo o caso é como se o corpo caísse sobre a superfície do mar, onde não haveria bordas para as reflexões. Isso desconsiderando as reflexões que surgem com a passagem da frente de onda de uma camada para outra.



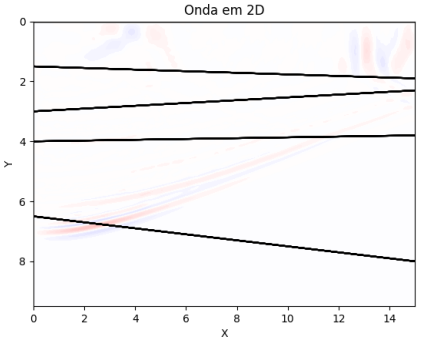
(a) 1



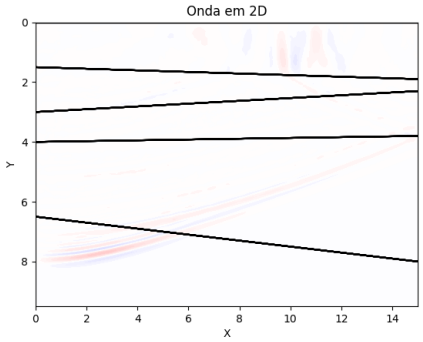
(b) 2



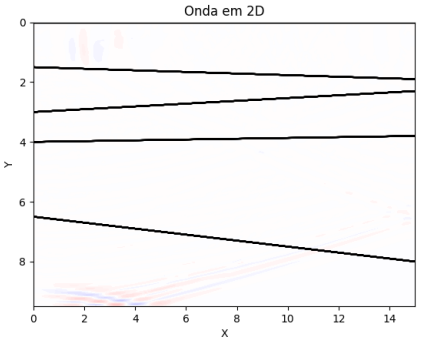
(c) 3



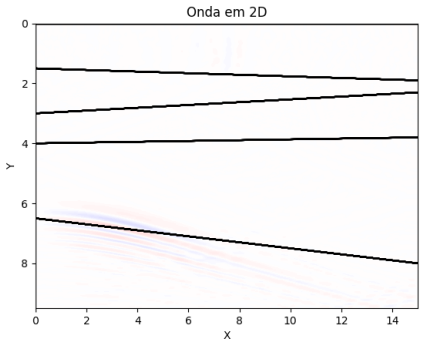
(d) 4



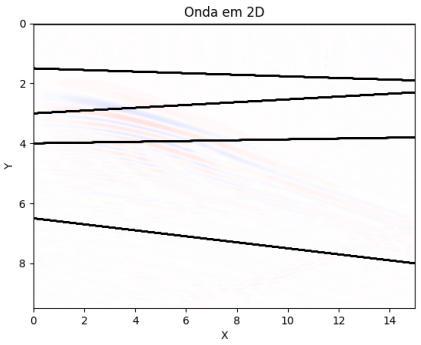
(e) 5



(f) 6



(e) 7



(f) 8

Fig. 9.5: *Snapshots* da simulação numérica da propagação de uma onda.

9.2 Resolução da Equação da Onda pelo Traçamento de Raios

9.2.1 Implementação

Foi implementado um traçamento de raios simples, no qual há apenas uma reflexão nas interfaces que são definidas como refletoras. Neste traçamento, não são representadas as reflexões que ocorrem com as refrações.

O raio é implementado como um *array* de pontos que indicam sua posição (x, y) no tempo t e outro *array* de componentes que constituem o vetor direção do raio ao longo do tempo. Já as interfaces são implementadas como retas em sua forma paramétrica: com um vetor diretor e um ponto. Além disso, a interface também possui um vetor normal.

O Traçamento

As posições e direções do raio ao longo do tempo são calculadas com o auxílio do método de Runge-Kutta de quarta ordem. Isso é feito porque o raio é definido matematicamente como um sistema de equações diferenciais para a posição e a direção ao longo do tempo.

A interseção do raio com com a próxima interface se dá da seguinte forma: quando a função que implementa o método de Runge-Kutta percebe que o raio ultrapassou a interface, ela cria uma reta ligando os dois últimos pontos traçados para o raio e calcula a interseção dessa reta com a interface. O ponto de interseção calculado torna-se o último ponto do traçamento do raio para aquela camada.

Lei de Snell

Temos que a lei de Snell se dá por

$$\frac{\operatorname{sen}\theta_2}{\operatorname{sen}\theta_1} = \frac{v_2}{v_1} = \frac{n_1}{n_2}$$

na qual, θ_1 e θ_2 são os ângulos que, respectivamente, os raios incidente e refratado fazem com a normal, v_1 e v_2 são as velocidades dos meios acima e abaixo da reta preta (nas Figuras 9.6 e 9.7), respectivamente, assim como n_1 e n_2 são os coeficientes de refração de cada um, também respectivamente. A implementação da reflexão e da refração se baseiam nessa lei.

A Reflexão

Estando o raio sobre a interface que ele deve refletir, calculamos a projeção vetorial, $\vec{s}$, do último vetor direção, $\vec{p}$, do raio sobre a normal da interface. Após isso, somamos $\vec{p}$ com menos duas vezes $\vec{s}$ e obtemos o vetor direção refletido do raio. Também podemos visualizar isso pela seguinte imagem.

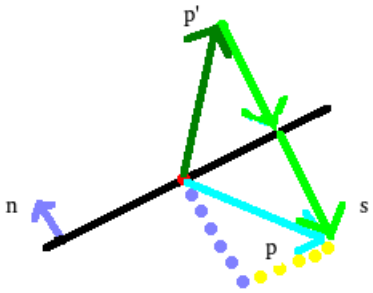


Fig. 9.6: Esquema ilustrativo da reflexão do raio $\vec{p}$, em azul ciano, sobre uma interface, em preto. O vetor $\vec{s}$, em verde, é a projeção vetorial de $\vec{p}$ sobre o vetor normal $\vec{n}$ da interface, em lilás

A Refração

Estando o raio sobre uma interface não refletora, projetamos o vetor $\vec{p}$ sobre o vetor diretor $\vec{d}$ da interface. Dessa forma, sendo $\vec{p}$ um vetor unitário, obtemos o seno do ângulo da direção do raio naquele momento pela norma do vetor projetado. Com esse seno, aplicamos a lei de Snell e obtemos o seno do ângulo refratado. Então, pela seguinte igualdade trigonométrica

$$\cos\theta = \sqrt{1 - \text{sen}^2\,\theta}$$

obtemos o cosseno do ângulo refratado. Assim, podemos multiplicar vetor $\vec{n}$ da interface pelo cosseno encontrado e somar esse vetor a $\vec{d}$ multiplicado pelo seno, obtendo então o novo vetor $\vec{p}$ do raio.

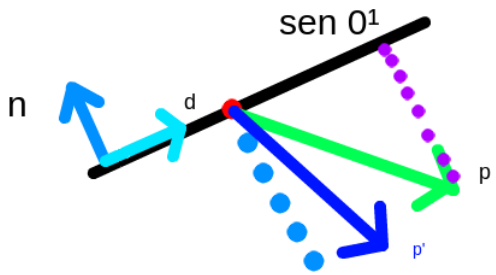


Fig. 9.7: Esquema ilustrativo da reflexão do raio $\vec{p}$, em verde, sobre uma interface, em preto. A norma da projeção de $\vec{p}$ sobre $\vec{d}$, em azul claro, é o seno do ângulo de $\vec{p}$ com a normal $\vec{n}$, em azul escuro. Após a aplicação da lei de Snell sobre esse seno, obtém-se o seno do ângulo refratado, que é usado para obter o cosseno do mesmo ângulo. Obtém-se $\vec{p'}$ multiplicando o cosseno e o seno encontrados por $\vec{n}$ e $\vec{d}$, respectivamente

9.2.2 Traçando Raios

Utilizando a mesma distribuição de velocidades usada nos exemplos de implementação do método de diferenças finitas, podemos ver abaixo a simulação da propagação de ondas pelo mesmo meio das outras simulações, mas agora pelo método de traçamento de raios. Os ângulos mínimo e máximo de emisão dos raios foram, respectivamente, 10^o e 18^o. A cada encontro do raio com uma interface podemos ver uma refração (calculada usando a função velocidade das duas camadas naquele ponto), exceto na última interface.

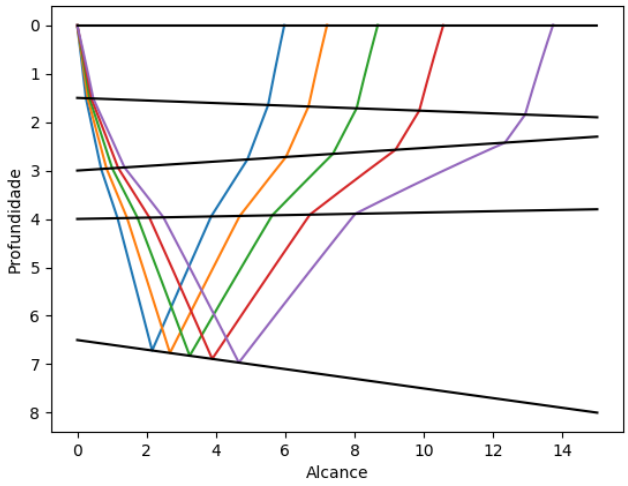


Fig. 9.8: Simulação da propagação de ondas pelo método do traçamento de raios

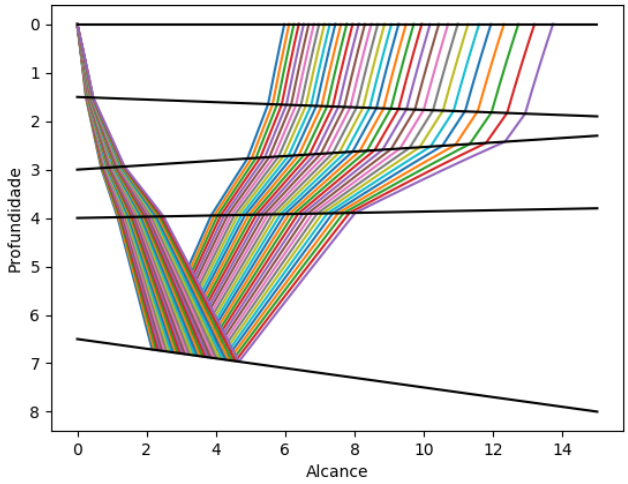


Fig. 9.9: A mesma simulação que a Figura **??** exibe, mas agora com 35 raios

Podemos ver pelas simulações seguintes que é também possível decidir sobre qual interface os raios devem refletir. Isso é útil para se observar como se dá a reflexão sobre cada interface

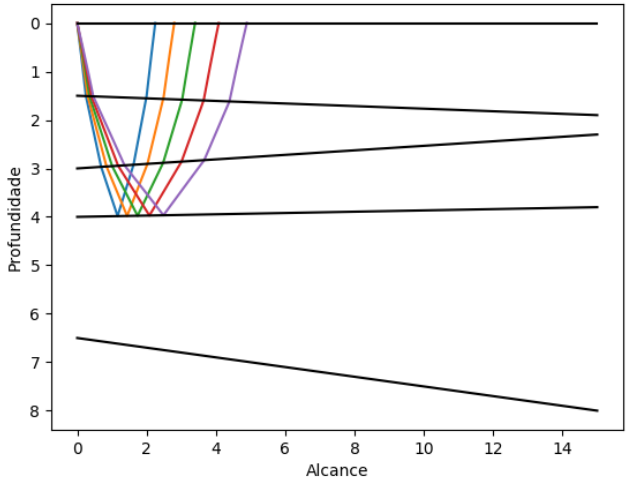


Fig. 9.10: Traçamento de raios com reflexão na terceira interface

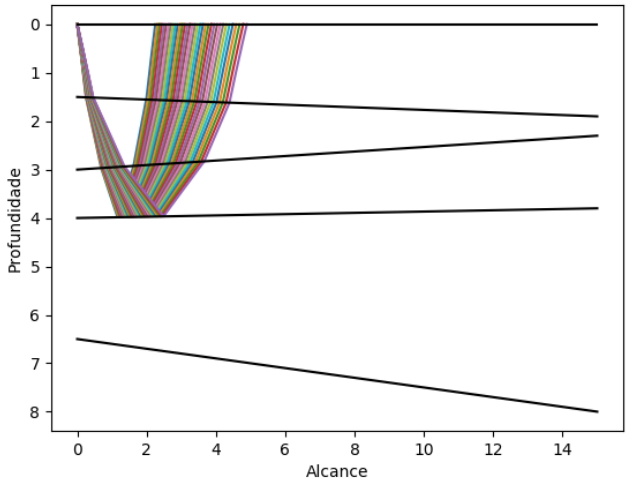


Fig. 9.11: O mesmo traçamento da Figura **??**, mas agora com mais raios

9.3 Comparação Entre os Métodos

Tanto o método de diferenças finitas quanto o traçamento de raios possuem suas vantagens e desvantagens. Durante a execução deste trabalho pude notar as seguintes características sobre cada uma dessas abordagens para a simulação de propagação de ondas.

9.3.1 Método das Diferenças Finitas

Possui uma forma fácil de se implementar, necessitando-se apenas conhecer os conceitos de derivadas parciais e diferenças finitas, que são relativamente fáceis. Para para a resolução

numérica de um problema de equações diferenciais parciais, como a da onda, bastam poucas linhas de código. O uso da biblioteca *Numpy* possibilita isso, além de acelerar os tratamentos numéricos no processo.

Além disso, o a visualização da transformação dos dados, através do tempo, possibilitada método e pelas bibliotecas gráficas permite entender, intuitivamente, que a equação da onda é um modelo que imita bem o fenômeno físico. Essa visualização, em pequena escala, poderia ser utilizada na ministração de aulas de Cálculo Numérico, Métodos Numéricos Computacionais e/ou Física.

Contudo, o processo de simulação por esse método tem um preço alto para grandes domínios (bem como da visualização gráfica de seus dados, que pode custar mais ainda). É comum que, para evitar instabilidades numéricas tenha-se que aumentar o número de pontos no *array*, para que os passos dados pelo método não sejam muito grandes. Isso afeta a quantidade de memória e de processamento (e, consequentemente, de tempo e de recursos financeiros) necessária para a execução da simulação a tal ponto que, para uma simulação na área de Sismologia, por exemplo, que tem um grande volume de dados, a utilização do método se torna inviável.

9.3.2 Traçamento de Raios

A teoria do traçamento de raios é, matematicamente, complicada e nada intuitiva nos primeiros contatos. A implementação em código do traçamento também segue isso, sendo repleta de condicionais e cuidados que devem ser levados em conta. É necessária uma boa compreensão da teoria matemática antes da implementação do método, bem como a paciência com os erros que aparecerão e tempo para a implementação dos detalhes do traçamento.

Contudo, imitar uma parte infinitesimal da frente de onda ao invés de simulá-la completamente é muito mais eficiente e menos custoso. Ou seja, o uso do traçamento de raios gasta menos memória (basta armazenar, além do meio e suas informações, as informações dos raios) e processamento (não é necessária computação sobre um grande *array* tridimensional, mas apenas os *arrays* de posição e direção dos raios) que o método de diferenças finitas para fazer, fundamentalmente, a mesma coisa. É por isso que o traçamento é uma das técnicas mais utilizadas nos ramos como a já citada Sismologia.

Por fim, até mesmo a exibição gráfica dos dados do traçamento, pelo menos em pequena escala como feito nesse projeto, é mais rápida do que no caso da exibição dos dados conseguidos pelo método anterior, visto que exibe-se “linhas” e não malhas inteiras.

Capítulo 10

Conclusão

Para alcançar o objetivo proposto de se comparar os métodos de diferenças finitas e traçamento de raios para a simulação da propagação de ondas em meios não-homogêneos foi necessário o estudo não só dos métodos em si, mas dos assuntos que dão base a eles:

- **Computação numérica**: era necessária a implementação computacional dos métodos numéricos, sendo preciso o entendimento sobre a aritmética computacional e os erros envolvidos em modelagens matemáticas implementadas em computadores;
- **Equações diferenciais parciais**: a teoria de raios assume o raio como uma EDO, sendo necessário o entendimento sobre esse tipo de equação;
- **Resolução numérica de problemas de valor inicial**: visto que o raio é, matematicamente, uma EDO e o traçamento de raios é implementado computacionalmente, é necessário se conhecer como se dá a resolução numérica de EDO's, usando-se, por exemplo, o método de Runge-Kutta;
- **Ondulatória**: para se entender a propagação de ondas é preciso se entender o comportamento delas;
- **Equações Diferenciais Parciais (e a equação da onda)**: a equação da onda é uma EDP e precisamos entendê-la tanto para aplicar o método de diferenças finitas sobre ela, quanto para entender a teoria dos raios.

A partir daí vimos as ideias dos métodos envolvidos na comparação:

- **Método da diferenças finitas**: modela-se o domínio a ser estudado como uma malha subdividida em retângulos ou paralelepípedos e, para cada ponto dessa malha, aplica-se alguma fórmula (específica para o problema) de diferenças finitas, que são modelagens computacionais para as derivadas;
- **Traçamento de raios**: ao invés de se modelar toda a frente de onda, separa-se uma fração infinitesimal dela e determina-se a posição e a direção dessa fração dentro de um período de tempo previamente determinado.

Comparando-se as características observadas para cada método, vimos que o traçamento de raios é mais rápido e menos custoso que o método de diferenças finitas, entretanto, a teoria dos raios não é tão intuitiva quando a definição de diferenças finitas.

Esse trabalho me possibilitou enxergar aplicações das disciplinas de Cálculo e Física, bem como da programação, no mundo real. Os estudos realizados durante o projeto permitiram a

aquisição de novos conhecimentos sobre ondas (sua presença e aplicações em lugares inusitados, como o estudo do interior da Terra), equações diferenciais e na forma como elas podem ser modeladas para resolver problemas nas áreas das Ciências Exatas e Engenharias. Além disso, pude perceber a necessidade da otimização de modelagens numéricas computacionais, para que sejam resolvidas mais rapidamente e com custo menor, dada a sua importância.

Por fim, o trabalho pode ser melhorado com, por exemplo, uma análise mais técnica da complexidade e do tempo de execução dos programas envolvidos. Além disso, podem ser estudados alguns instrumentos da Sismologia, como os traços sísmicos ou variações do traçamento de raios, como pode ser visto em [16]. Também pode ser feita a otimização dos programas principais envolvidos, como por programação paralela e com técnicas de computação de alto desempenho.

Appendices

```
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)
```

```
'''
    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
```

Referências Bibliográficas

[1] Ray path - schlumberger oilfield glossary, jan 2018.

[2] Seismic velocity - schlumberger oilfield glossary, jan 2018.

[3] Traveltime - schlumberger oilfield glossary, jan 2018.

[4] W. E. Boyce and R. C. DiPrima. *Equações Diferenciais Elementares e Problemas de Valores de Contorno*. LTC, 2011.

[5] R. L. Burden and J. D. Faires. *Análise Numérica*. CENGAGE Learning, 2008.

[6] S.-H. Chen. Finite difference methods.

[7] W. contributors. Partial differential equation — wikipedia, the free encyclopedia, 2017. [Online; accessed 29-December-2017].

[8] W. contributors. Polarization (waves) — wikipedia, the free encyclopedia, 2018. [Online; accessed 31-March-2018].

[9] W. contributors. Standing wave — wikipedia, the free encyclopedia, 2018. [Online; accessed 31-March-2018].

[10] L. A. D'Afonseca. Notas de aula. Notas de aula obtidas durante as reuniões do projeto., 2017.

[11] R. C. Daileda. The two dimensional wave equation, mar 2012.

[12] B. Fornberg. Finite difference method, 2011.

[13] f. Frederico Ferreira Campos. *Algoritmos Numéricos*. LTC, 2007.

[14] D. Halliday, R. Resnick, and J. Walker. *Fundamentos de Física, volume 2: Gravitação, Ondas e Termodinâmica*. Rio de Janeiro, 8 edition, 2009.

[15] D. A. R. Justo, E. Sauter, F. S. de Azevedo, L. F. Guidi, M. C. dos Santos, and P. H. de Almeida Konzen. *Cálculo Numérico - Um Livro Colaborativo*. UFGRS, 2017.

[16] E. X. Miqueles. Modelamento sísmico em meios analíticos. Master's thesis, Universidade Federal do Paraná, 2006.

[17] J. M. Moehlis. Solution of the diffusion equation by finite differences. `https://me.ucsb.edu/~moehlis/APC591/tutorials/tutorial5/node3.html`, oct 2001.

[18] H. M. Nussenzveig. *Curso de Física Básica*, volume 2. São Paulo, 2002.

[19] N. Rawlinson. Lecture 6: Ray theory. http://rses.anu.edu.au/~nick/teachdoc/lecture6.pdf, apr 2008.

[20] R. J. Santos. *Introdução às Equações Diferenciais Ordinárias*. Imprensa Universitária da UFMG, 2013.

[21] U. Sodré. *Equações Diferenciais Parciais*. Ulysses Sodré, 2003.

[22] S. Stein and M. Wysession. *An Introduction to Seismology, Earthquakes, and Earth Structure*. Blackwell Publishing Ltd, 1 edition, 2003.

[23] J. Stewart. *Cálculo, volume 2*. CENGAGE Learning, São Paulo, 7 edition, 2013.

[24] SubSurfWiki. Ricker wavelet. http://www.subsurfwiki.org/wiki/Ricker_wavelet, feb 2018.

[25] E. W. Weisstein. Partial differential equation, dec 2017.

[26] Wikipedia. Malthusian growth model — wikipedia, the free encyclopedia, 2017. [Online; accessed 11-December-2017].

[27] Wikipédia. Equação de laplace — wikipédia, a enciclopédia livre, 2015. [Online; accessed 24-setembro-2015].

[28] Wikipédia. Equação de lotka-volterra — wikipédia, a enciclopédia livre, 2017. [Online; acessado 12-novembro-2017].

[29] Wikipédia. Paraboloide — wikipédia, a enciclopédia livre, 2017. [Online; accessed 30-maio-2017].

[30] Wikipédia. Parábola — wikipédia, a enciclopédia livre, 2017. [Online; accessed 24-junho-2017].

[31] Wikipédia. Onda — wikipédia, a enciclopédia livre, 2018. [Online; accessed 29-janeiro-2018].