Métodos Numéricos para Ciências Computacionais e Engenharia

Prof. Gustavo Oliveira

Dec 24, 2021

CONTENTS

Conteúdo para formação complementar empregável em cursos de graduação introdutórios sobre métodos numéricos para aprendizagem ativa baseada em problemas.

Este conteúdo é ensinado na disciplina *Cálculo Numérico* (GDCOC0072) ministrada pelo Prof. Gustavo Oliveira (UFPB/CI/DCC). Todo material é desenvolvido no âmbito do Projeto Numbiosis com suporte do Programa Institucional de Monitoria.

Aplicações específicas em Javascript que simulam diversos métodos do curso foram desenvolvidas pelo egresso Vinícius Veríssimo e estão disponíveis aqui.

Part I

Programa geral do curso

Notebooks com notas de aula, exemplos resolvidos, algoritmos e exercícios de programação.

CHAPTER

ONE

I. INTRODUÇÃO

- *Aula 00 - Modelagem*
- *Aula 01 - Ponto flutuante*
- *Aula 02 - Erros numéricos*

CHAPTER

TWO

II. DETERMINAÇÃO DE RAÍZES

- *Aula 03 - Análise gráfica*
- *Aula 04 - Bisseção*
- *Aula 05 - Ponto fixo*
- *Aula 06 - Newton*
- *Aula 07 - Secante*
- *Aula 08 - Müller*

CHAPTER

THREE

III. SOLUÇÃO DE SISTEMAS DE EQUAÇÕES

- *Aula 09 - Eliminação Gaussiana*
- *Aula 10 - Fatoração LU*
- *Aula 11 - Cholesky*
- *Aula 12 - Jacobi*
- *Aula 13 - Newton não-linear*

CHAPTER

FOUR

IV. INTERPOLAÇÃO E AJUSTE DE CURVAS

- *Aula 14 - Interpolação de Lagrange*
- *Aula 15 - Interpolação de Newton*
- *Aula 16 - Mínimos quadrados*
- *Aula 17 - Ajuste não linear*

CHAPTER

FIVE

V. INTEGRAÇÃO E DIFERENCIAÇÃO NUMÉRICA

- *Aula 18 - Integração por Newton-Cotes*
- *Aula 19 - Quadratura Gaussiana*
- *Aula 20 - Diferenciação numérica*

CHAPTER

SIX

VI. MÉTODOS NUMÉRICOS PARA EDOS

- *Aula 21 - Solução numérica de EDOs*
- *Aula 22 - Método de Euler*
- *Aula 23 - Métodos de Taylor e Runge-Kutta de 2a. ordem*

Part II

Code sessions

Notebooks com um compêndio de funções de utilidade predefinidas em módulos Python para resolução direta de problemas aplicados às ciências matemáticas, computacionais e engenharias.

- *Code session 1 - bisect*
- *Code session 2 - newton*
- *Code session 3 - roots*
- *Code session 4 - fsolve*
- *Code session 5 - solve*
- *Code session 6 - interp*
- *Code session 7 - fit*
- *Code session 8 - integrate*
- Code session 9 - solve_ivp

Part III

Listas de exercícios

Notebooks contendo solucionário matemático e computacional de exercícios gerais de menor complexidade.

- *Lista 1*
- *Lista 2*
- *Lista 3*
- *Lista 4*
- *Lista 5*
- *Lista 6*

Part IV

Conteúdo Extra

Notebooks com conteúdos complementares não contemplados no curso regular.

- *Números em ponto flutuante e seus problemas*
- *Malhas numéricas*
- Campos de direção para EDOs
- *Melhoramentos do método de Euler*
- *Estabilidade do método de Euler*
- *Método de Euler implícito*
- *Métodos de múltiplos passos:* Adams-Bashfort
- *EDOs de ordem superior*
- Sistemas de EDOs
- *Transformada de Fourier*
- *Otimização de código*

Part V

Como contribuir?

O projeto Numbiosis não recebe financiamento direto para bolsas. Todo o conteúdo é desenvolvido pelo Prof. Gustavo Oliveira e alunos (monitores e/ou tutores bolsistas ou voluntários, bem como aqueles que se matriculam no curso e contribuem com melhorias). O material é revisado constantemente, mas possui suporte limitado.

Você é estudante da UFPB e gostaria de contribuir com o projeto? Entre em contato com o Prof. Gustavo.

CHAPTER

SEVEN

ALGUNS TEMAS ABERTOS NO ÂMBITO DO PROJETO NUMBIOSIS

- Implementação de gráficos interativos para visualização 3D de processos iterativos.
- Desenvolvimento de APIs para integração de códigos da base Numbiosis em Github/Gitlab.
- Configuração de web server JupyterHub para hospedagem de materiais para mini-cursos remotos.
- Desenvolvimento de códigos demonstrativos em Python para aplicações em Engenharias.
- Geração de material didático portável (projeto de ensino).
- Integração de ferramentas de *autograding*.
- Programação orientada a objetos para criação de *smart courses* (módulos para geração de questões customizadas, avaliações e compilações em Latex).

Part VI

Iniciação científica

Consulte projetos nos horizontes estratégicos do TRIL Lab no CI/UFPB. Algumas temas de interesse são:

- Computação científica para aplicações em engenharia
- Ciência de dados para o setor energético
- Dinâmica dos fluidos computacional

Part VII

Cadernos

CHAPTER

EIGHT

MODELAGEM MATEMÁTICA EM CIÊNCIAS COMPUTACIONAIS E ENGENHARIAS

Ao deparar-se com um problema real, profissionais envolvidos nos diversos setores da economia buscam as melhores estratégias existentes em seu campo de atuação para resolvê-lo. Supondo que um problema específico possua apenas uma solução e que pudéssemos assemelhá-lo à uma questão específica, a solução desse problema equivaleria à resposta para a pergunta e a forma de respondê-la à estratégia seguida para buscar a solução, a qual também poderíamos chamar de *método*.

Alguns exemplos de perguntas que eventualmente surgiriam em domínios específicos do conhecimento são:

- que carga máxima seria admissível para um elevador de edifício-garagem de 15 andares, de maneira que seus cabos, mecanismos e estrutura de elevação não sejam danificados por excesso de tensão? (Engenharia Civil)
- quantos mililitros de um fármaco anti-alérgico poderiam ser injetados por minuto na corrente sanguínea de um indivíduo de modo que não haja efeitos colaterais de overdose? (Biologia)
- que magnitude mínima de pressão seria necessária para mover um fluido refrigerante por um nanotubo de um componente eletrônico aquecido de maneira a resfriá-lo em 2% de sua temperatura operacional? (Engenharia Mecânica)

Evidentemente, perguntas como essas podem ser extremamente difíceis de responder, sendo necessário o envolvimento de uma equipe multidisciplinar, com experiência em variados assuntos. A primeira das perguntas pode envolver profissionais especializados em ciência dos materiais, eletrônica e pneumática; a segunda, matemáticos aplicados, cardiologistas e químicos; a terceira, físicos, analistas de energia e estatísticos.

Resolver um problema científico no mundo atual requer não apenas conhecimento teórico e prático, mas também outras habilidades como a capacidade de pensar computacionalmente e resolver problemas utilizando o enorme potencial da computação de alto desempenho. Na verdade, a maioria dos problemas atuais não pode ser resolvida sem a intervenção de métodos numéricos. Há, certamente, um amplo leque de opções metodológicas para resolvê-los. Entretanto, em linhas gerais, são quatro as fases de resolução de um problema aplicado, como mostra a Fig. **??**.

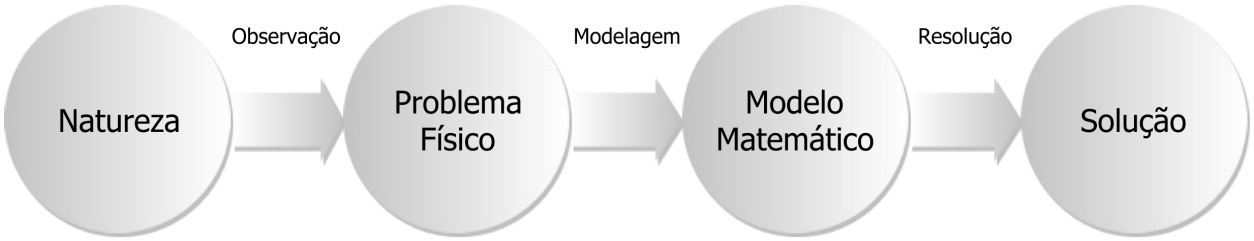


Fig. 1: As quatro fases fundamentais para a resolução de um problema aplicado nas ciências computacionais e engenharias.

8.1 Modelos matemáticos

Um modelo matemático pode ser definido, de forma geral, como uma formulação ou equação que expressa as características essenciais de um sistema ou processo físico em termos matemáticos. A fórmula pode variar de uma simples relação algébrica a um conjunto grande e complicado de equações diferenciais.

Por exemplo, com base em suas observações, Newton formulou sua segunda lei do movimento. Se escrevermos a taxa de variação temporal da velocidade pela derivada $\frac{dv}{dt}$ (em m/s), um modelo matemático que obtemos para a segunda lei de Newton é

$$\frac{dv}{dt} = \frac{F}{m},$$

onde F é a força resultante (em N) agindo sobre o corpo e m a massa (em kg).

Este modelo matemático, assim como vários outros, possui as seguintes características:

- descrevem um processo ou sistema natural em termos matemáticos;
- representam uma idealização (simplificação) da realidade. Isto é, o modelo ignora alguns "detalhes" do processo natural e se concentra em suas manifestações essenciais;
- produzem resultados que podem ser reproduzidos e usados para propósitos de previsão. Por exemplo, se a força sobre um corpo e a sua massa forem conhecidas, o modelo pode ser usado para estimar a aceleração $a = \frac{dv}{dt}$ do corpo.

Consideremos um paraquedista em queda livre. Durante seu movimento, duas forças principais atuam sobre ele. A força gravitacional F_G, com sentido para baixo, e a força da resistência do ar (arrasto) F_D, em sentido oposto. Se o sentido positivo for conferido à força gravitacional, podemos modelar a força resultante como

$$F = F_G - F_D = mg - cv,$$

onde g é a constante gravitacional e c o *coeficiente de arrasto*, medido em kg/s. Vale ressaltar que ao assumirmos $F_D = cv$, estamos dizendo que a força de arrasto é linearmente proporcional à velocidade. Entretanto, na realidade, esta relação é não-linear.

Dessa maneira, podemos chegar a um modelo mais completo substituindo a força resultante assim obtendo:

$$\frac{dv}{dt} = \frac{mg - cv}{m} = g - \frac{c}{m}v.$$

Esta *equação diferencial ordinária* (EDO) possui uma solução geral que pode ser encontrada por técnicas analíticas. Uma solução particular para esta EDO é obtida ao impormos uma *condição inicial*. Visto que o paraquedista está em repouso antes da queda, temos que $v = 0$ quando $t = 0$. Usando esta informação, concluímos que o perfil de velocidade é dado por

$$v(t) = \frac{gm}{c}(1 - e^{-(c/m)t}).$$

Como veremos adiante em um estado de caso real apresentado por Yan Ferreira, esta função cresce exponencialmente até atingir uma estabilização na *velocidade terminal*.

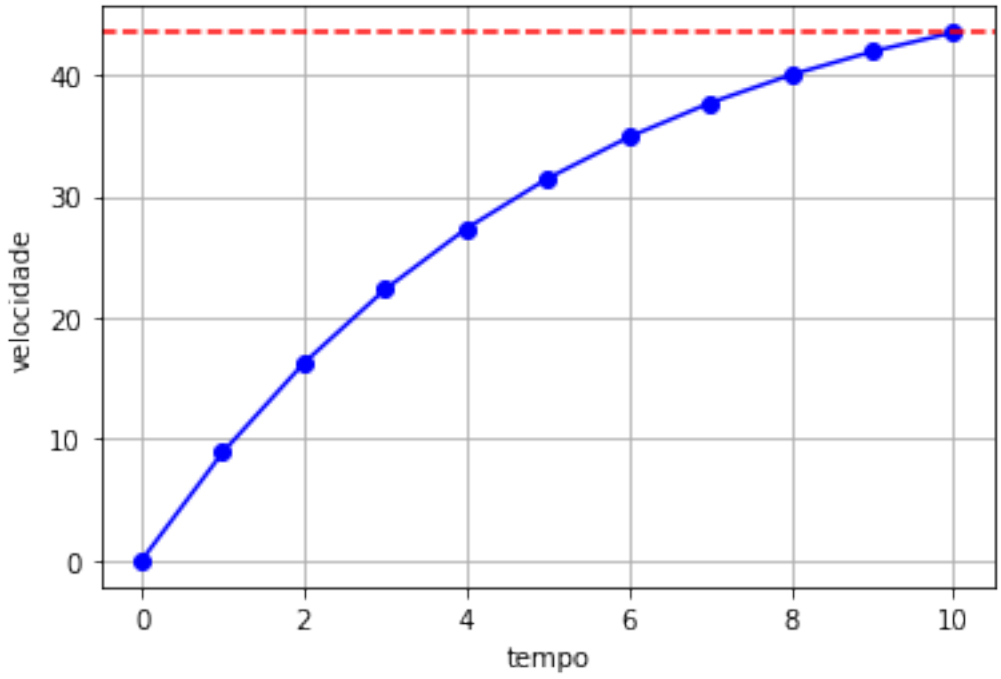
8.2 O salto de paraquedas de Yan e Celso

Como exemplo, veremos a análise do salto de paraquedas de Yan com seu irmão Celso. Vamos calcular a aceleração que seria atingida por Yan desde o salto até o momento da abertura de seu paraquedas. Na época do salto, Yan estava com 65 kg e o ar apresentava um coeficiente de arrasto estimado em 12,5 kg/s.

Utilizando a fórmula dada, podemos calcular a velocidade atingida por Yan em relação ao tempo. A seguir, criamos uma função para calcular $v(t)$ nos 10 primeiros segundos do salto, que foi o tempo que Yan permaneceu em queda até a abertura do paraquedas.

```
# velocidade no salto de Yan
import numpy as np
import matplotlib.pyplot as plt


arr_yan = 11*[0]
contador = 0
while (contador <= 10):
    a = ((9.8 * 65)/12.5)*(1 - np.exp(-(12.5/65)*contador))
    arr_yan[contador] = a
    print("{0:.4f} m/s".format(a))
    contador += 1
else:
    print('==> Abertura do paraquedas.')

plt.plot(arr_yan,'o-b')
plt.xlabel('tempo')
plt.ylabel('velocidade')
plt.grid()
plt.axhline(a,color='r',ls='--');
```

```
0.0000 m/s
8.9153 m/s
16.2709 m/s
22.3397 m/s
27.3467 m/s
31.4778 m/s
34.8861 m/s
37.6982 m/s
40.0183 m/s
41.9325 m/s
43.5119 m/s
==> Abertura do paraquedas.
```



Porém, Celso, irmão de Yan, também saltou com ele, em separado. Celso, tem mais 20kg a mais do que Yan. Então, vamos ver como a massa influenciou a velocidade no salto de Celso e comparas as curvas.

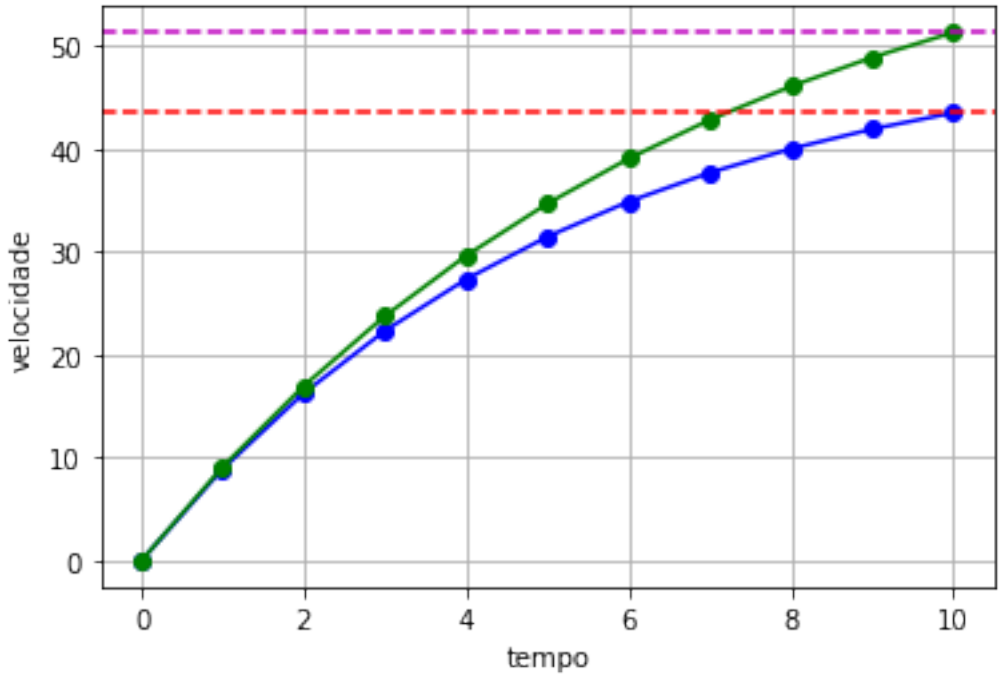
```
# velocidade no salto de Yan


from math import exp
import matplotlib.pyplot as plt

arr_celso = arr_yan[:]
contador = 0
while (contador <= 10):
    b = ((9.8 * 85)/12.5)*(1- exp(-(12.5/85)*contador))
    arr_celso[contador] = b
    print("{0:.4f} m/s".format(b))
    contador += 1
else:
    print('==> Abertura do paraquedas')

plt.plot(arr_yan,'o-b')
plt.plot(arr_celso,'o-g')
plt.xlabel('tempo')
plt.ylabel('velocidade')
plt.grid()
plt.axhline(a,color='r',ls='--')
plt.axhline(b,color='m',ls='--');
```

```
0.0000 m/s
9.1135 m/s
16.9806 m/s
23.7719 m/s
29.6344 m/s
34.6952 m/s
```

(continues on next page)

(continued from previous page)

```
39.0638 m/s
42.8351 m/s
46.0905 m/s
48.9008 m/s
51.3268 m/s
==> Abertura do paraquedas
```



Podemos observar que quanto maior a massa, maior é a velocidade atingida. Por esta razão, Celso chegou ao solo antes de Yan.

Este breve exemplo nos mostra a implementação de um modelo matemático em Python, onde utilizamos partes de programação *estruturada* e *modular*. Estruturada, no sentido das instruções e modular no sentido de que aproveitamos os pacotes (ou **módulos**) *math* e *numpy* para invocarmos a função exponencial `exp` e os as funções para plotagens gráficas.

8.2.1 Deslocamento até a abertura do paraquedas

Sabemos da Física e do Cálculo que o deslocamento é a integral da velocidade com relação ao tempo. Portanto, se D_Y e D_C foram os deslocamentos de Yan e Celso em seus saltos, podemo usar a integral:

$$D = \int_0^{10} v(t)\, dt,$$

em cada caso para computar esses deslocamentos. No Python, podemos fazer isso com o código abaixo (que você entenderá mais tarde como fazer).

```
import sympy as sy

t,g,m,c = sy.symbols('t g m c')
v = g*m/c*(1 - sy.exp(-c/m*t))
s1 = sy.integrate(v,(t,0,10)).subs({'m':65.0,'g':9.8,'c':12.5})
```

(continues on next page)

(continued from previous page)

```
s2 = sy.integrate(v,(t,0,10)).subs({'m':85.0,'g':9.8,'c':12.5})
print("Yan voou incríveis DY = {0:.2f} metros em 10 segundos!".format(s1))
print("Celso voou incríveis DC = {0:.2f} metros em 10 segundos!".format(s2))
```

```
Yan voou incríveis DY = 283.34 metros em 10 segundos!
Celso voou incríveis DC = 317.38 metros em 10 segundos!
```

8.3 Programação estruturada e modular

8.3.1 Prgramação estruturada

A idéia central por trás da programação estruturada é que qualquer algoritmo numérico pode ser composto de três estruturas de controle fundamentais: *sequencia*, *seleção* e *repetição*.

Nos primórdios da computação, os programadores usualmente não prestavam muita atenção ao fato de o programa ser claro e fácil de entender. Hoje, é reconhecido que existem muitos benefícios em escrever um código bem organizado e bem estruturado. Além do benefício óbvio de tornar o software mais fácil de ser compartilhado, isso também ajuda a garantir um desenvolvimento de programa mais eficiente.

Portanto, algoritmos bem estruturados são, invariavelmente, fáceis de corrigir e testar, resultando em programas que têm um tempo de desenvolvimento e atualização menor. Embora a programação estruturada seja flexível o suficiente para permitir criatividade e expressões pessoais, suas regras impõem restrições suficientes para garantir um código final de mais qualidade, mais limpo e mais elegante, quando comparada à versão não estruturada.

8.3.2 Programação modular

Na programação modular, a idéia é que cada módulo desenvolva uma tarefa específica e tenha um único ponto de entrada e um único ponto de saída, de modo que o desenvolvedor possa reutilizá-lo invariavelmente em várias aplicações.

Dividir tarefas ou objetivos em partes mais simples é uma maneira de torná-los mais fáceis de tratar. Pensando dessa maneira, os programadores começaram a dividir grandes problemas em subproblemas menores, ou **módulos**, que podem ser desenvolvidos de forma separada e até mesmo por pessoas diferentes, sem que isso interfira no resultado final.

Hoje em dia, todas as linguagens de programação modernas, tais como C++, Java, Javascript e a própria Python utilizam módulos (também conhecidos como *pacotes* ou *bibliotecas*). Algumas características diferenciais da programação modular são a manutençao facilitada e a reusabilidade do código em programas posteriores.

Abaixo, mostramos um exemplo avançado de como criar um módulo em Python para lidar com pontos na Geometria Plana. Nosso módulo poderia ser salvo em um arquivo chamado `ponto.py`, por exemplo e utilizado em programas próprios que viermos a desenvolver. Neste exemplo, a *classe* `Ponto` possui funções para realizar as seguintes operações:

- criar um novo ponto;
- calcular a distância Euclidiana entre dois pontos;
- calcular a área de um triângulo pela fórmula de Heron e
- imprimir o valor da área de um triângulo.

```
"""
Módulo: ponto.py
Exemplo de programação modular.
Classe para trabalhar com pontos do espaço 2D.
"""
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import matplotlib.pyplot as plt

class Ponto:

    # inicialização de um ponto arbitrário com coordenadas (xp,yp)
    def __init__(self, xp, yp):
        self.x = xp
        self.y = yp

    # fórmula da distância entre dois pontos
    def dist(P1,P2):
        return ( (P2.x - P1.x)**2 + (P2.y - P1.y)**2 )**0.5

    # área de um triângulo ABC pela fórmula de Heron
    def area_heron(P1,P2,P3):

        a = Ponto.dist(P1,P2)  # comprimento |AB|
        b = Ponto.dist(P2,P3)  # comprimento |BC|
        c = Ponto.dist(P3,P1)  # comprimento |CA|

        p = 0.5*(a + b + c)  # semiperímetro

        A = ( p*(p - a)*(p - b)*(p - c) )**0.5  # área

        return A

    def imprime_area_triangulo(P1,P2,P3):

        txt = 'Area do triângulo P1 = ({0}.{1}); P2 = ({2}.{3}); P3 = ({4}.{5}) :: A␣
↪= {6}'
        area = Ponto.area_heron(P1,P2,P3)

        print(txt.format(P1.x,P1.y,P2.x,P2.y,P3.x,P3.y,area))
```

Exemplo: usando a classe `ponto.py` para calcular a área de um triângulo retângulo

```
# Cálculo da área para o triângulo
# P1 = (0,0);  P2 = (1,0);  P3 = (0,1)

P1 = Ponto(0.0,0.0)
P2 = Ponto(1.0,0.0)
P3 = Ponto(0.0,1.0)
Ponto.imprime_area_triangulo(P1,P2,P3)
```

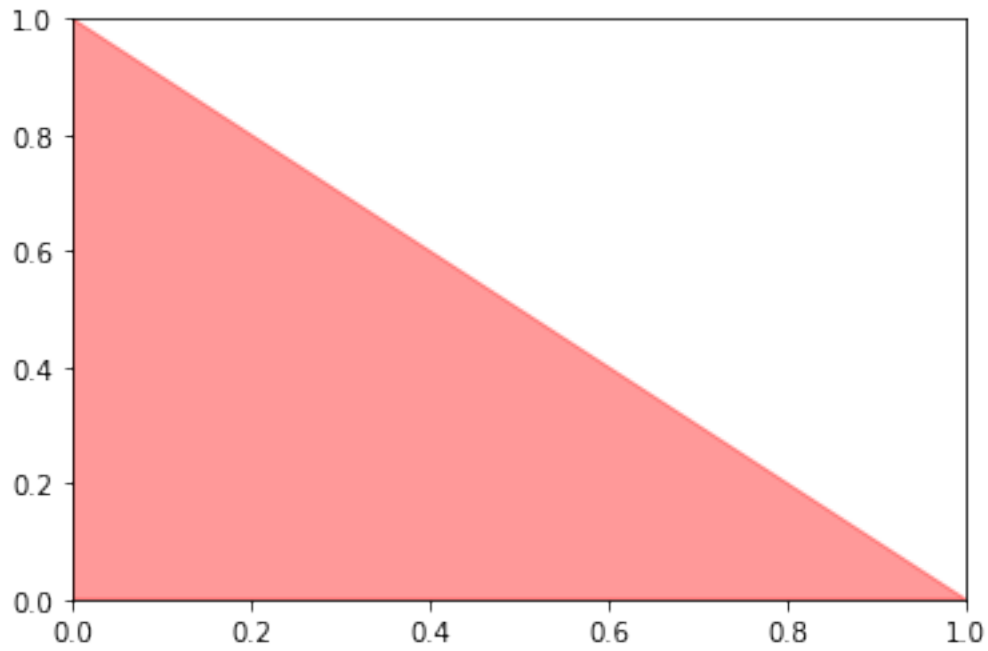
```
Area do triângulo P1 = (0.0.0.0); P2 = (1.0.0.0); P3 = (0.0.1.0) :: A = 0.
↪49999999999999983
```

```
# plotagem do triângulo
P = np.array([[P1.x,P1.y],[P2.x,P2.y],[P3.x,P3.y]])
plt.figure()
pol = plt.Polygon(P,color='red',alpha=0.4)
```

(continues on next page)

(continued from previous page)

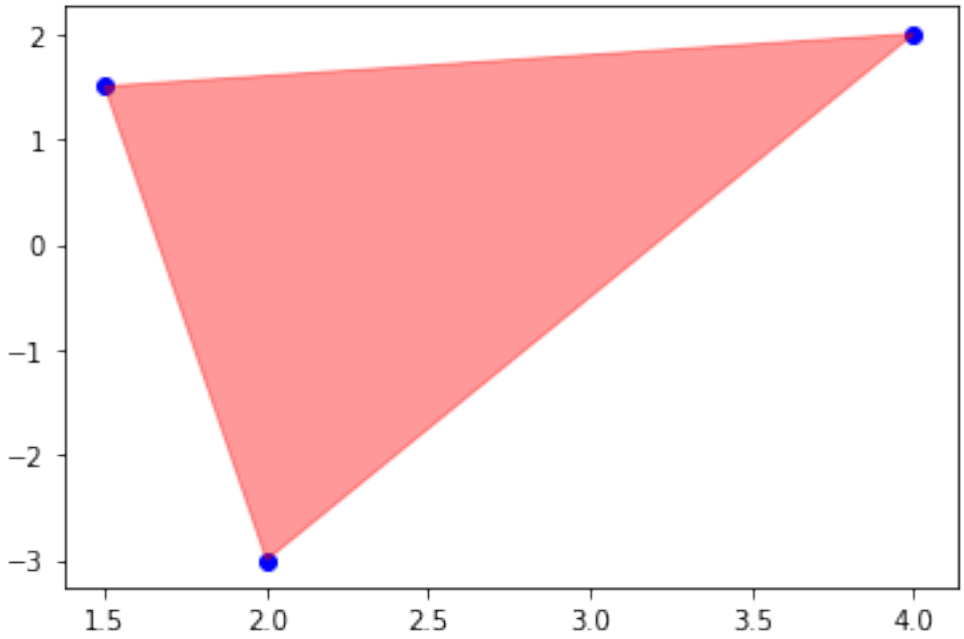
```
plt.gca().add_patch(pol);
```



Exemplo: usando a classe `ponto.py` para calcular a área de um triângulo qualquer

```
# Cálculo da área para o triângulo
# P1 = (4,2);  P2 = (3,2);  P3 = (2,-3)

P4 = Ponto(4.0,2.0)
P5 = Ponto(1.5,1.5)
P6 = Ponto(2.0,-3.0)
Ponto.imprime_area_triangulo(P4,P5,P6)

plt.figure()
plt.scatter(P4.x,P4.y,color='blue')
plt.scatter(P5.x,P5.y,color='blue')
plt.scatter(P6.x,P6.y,color='blue')

Px = [P4.x,P5.x,P6.x]
Py = [P4.y,P5.y,P6.y]
plt.fill(Px,Py,color='red',alpha=0.4);
```

```
Area do triângulo P1 = (4.0.2.0); P2 = (1.5.1.5); P3 = (2.0.-3.0) :: A = 5.75
```



8.4 O quarto paradigma da ciência

Nota: o texto desta seção pode ser encontrado em sua forma expandida neste link.

Durante uma palestra proferida na Califórnia em 2009 para o *Computer Science and Telecommunications Board – National Research Council (NRC- CSTB)*, o renomado cientista da computação Jerry Nicholas "Jim" Gray (1944 – 2012), pontuou o surgimento do quarto paradigma da ciência. Ao utilizar o termo "eScience", Gray queria dizer que a exploração científica seria grandemente influenciada pelo uso intensivo dos dados nos anos vindouros.

A evolução dos supercomputadores nesses últimos vinte anos elevou a capacidade de compreensão da natureza e permitiu que a ciência subisse mais um degrau na escada do conhecimento. A fim de entender o que é o quarto paradigma da ciência, vejamos quais são os três primeiros retrocedendo no tempo.

Há alguns milhares de anos, a ciência era essencialmente **empírica**. Tentava-se compreender os fenômenos naturais pela observação. Há algumas centenas de anos, equações, modelos e generalizações possibilitaram que a ciência se tornasse **teórica**, ou seja, passou-se a descrever com clareza e tecnicidade como um certo fenômeno funcionava. De algumas décadas para cá, as simulações de alta complexidade revelaram o terceiro pilar da ciência, tornando-a **computacional**. Atualmente, o amálgama entre teoria, experimentação e computação gerou uma vertente de **exploração dos dados**. Em outras palavras, poderíamos dizer que a ciência agora possui uma quarta faceta. Ela se tornou "datificada".

Os dados são exploráveis porque um dia puderam ser capturados, mensurados, processados e simulados. Diante disso, a ciência atual é:

1. Experimental;
2. Teórica;
3. Computacional e
4. Datificada,

sendo o último paradigma a consumação dos três anteriores.

8.4.1 Ciência e Engenharia Computacional

A **Ciência e Engenharia Computacional**, internacionalmente conhecida pelo acrônimo CSE (*Computational Science and Engineering*) é uma área interdisciplinar que compreende a ciência da computação, matemática aplicada, biologia e outras áreas do núcleo STEM (*Science, Technology, Engineering, and Mathematics*) voltada à resolução de problemas práticos das engenharias. A característica peculiar da CSE é o uso de métodos numéricos e sua integração com modelos matemáticos para subsidiar simulações computacionais e a resolução de equações diferenciais ordinárias ou parciais.

Equações diferenciais descrevem uma ampla variedade de fenômenos físicos, desde a absorção de um fármaco pelo organismo humano em escala nanométrica, até as ondas de choque macroscópicas causadas pela explosão de uma dinamite. Estes são apenas dois exemplos de situações que interessariam a indústria de biotecnologia e de construção ocorrendo em escalas extremamente distintas. Portanto, a CSE é afeita a problemas do mundo real caracterizados por multi-escalas pelos quais se compreende a dinâmica intrínseca de um fenômeno complexo modelável analiticamente.

8.4.2 Por que a CSE importa em um mundo de dados?

Embora em muitas áreas já se saiba com riqueza de detalhes como mecanismos intrínsecos de sistemas dinâmicos funcionam, em outras, este não é o caso. A indisponibilidade de dados em certos domínios do conhecimento decorre, principalmente, da dificuldade de capturá-los e coletá-los, seja pela inexistência de infraestrutura tecnológica, seja pelo alto investimento necessário para obtê-los. Um exemplo é a exploração da subsuperfície terreste. Dominar o conhecimento acerca da mecânica das rochas, a salinidade de aquíferos, o potencial geotérmico, ou a constituição química do gás natural nas partes mais baixas do planeta equivale a adentrar dezenas, centenas ou milhares de quilômetros na litosfera. Essa não é uma tarefa fácil. Pelo contrário, há tantas variáveis, riscos e custos envolvidos que sua execução pode ser inviabilizada.

A CSE entra em cena para preencher lacunas que os "dados", por si só, não conseguem fechar. Uma vez que a exploração de dados só é exequível quando os próprios dados estão disponíveis, seria impossível explorar dados acerca de algo desconhecido. Claramente, seria paradoxal. Para dar outra ilustração, tomemos o exemplo das imagens digitais. A tecnologia atual provê condições suficientes para tirarmos fotos de milhares de objetos, seres e indivíduos (p.ex. uma formiga, um prédio, um planeta) sem qualquer dificuldade.

Processar imagens, hoje em dia, é um dos grandes carros-chefe para que modelos de inteligência artificial sejam implementados com precisão. Mas a manipulação de imagens só é possível por causa do progresso na física, óptica álgebra linear – afinal, imagens são matrizes – e ciência dos materiais – para construir carcaças de celulares, câmeras DSLR e satélites.

Aqui, cabe uma pergunta: será que no processos de fabricação desses componentes materiais, ninguém apelou para simulações computacionais? É bem improvável, porque nenhum gestor aprovaria o envio de um satélite responsável por fotografar o oceano Atlântico para o espaço sem ter uma margem aceitável de confiabilidade de que ele orbitaria corretamente e cumpriria seu propósito com segurança. E como se adquire essa confiabilidade? Vai-se para um laboratório, cria-se um protótipo e faz-se 1, 2, 10, 1.000, 100.000 simulações, até que se reduzam ao máximo as incertezas.

Estando em algum lugar entre o terceiro e o quarto paradigma, a CSE não apenas intermedia a análise de incertezas, como também simula processos, assim gerando economia de recursos. A CSE usa o primeiro, o segundo e o terceiro na transição para o quarto paradigma e, naturalmente, agrega valor a processos de um mundo que vive em transição tecnológica.

8.4.3 Engenharia centrada em dados: um novo ramo na árvore do futuro?

Mark Girolami, um professor de Engenharia Civil em Cambridge, liderou um grande programa de ciência de dados e inteligência artificial no Alan Turing Institute entre os anos de 2017 e 2020. Em sua percepção, a ciência de dados já havia se impregnado nas engenharias de tal forma que a inauguração de um novo termo para descrever essa interseção crescente seria justificável. Ele o chamou de engenharia centrada em dados (data-centric engineering, ou DCE).

Segundo Girolami, a DCE é explicada por um desenvolvimento substancial que impacta as engenharias, profissões associadas, suas práticas e também a política. Ao relembrar que a engenharia erigiu-se sobre dados desde seus primórdios, citou em seu artigo uma clássica fala de Lord Kelvin de 1889:

> "Quando você pode medir o que está falando e expressar isso em números, você sabe alguma coisa sobre aquilo; quando não o pode expressar em números, seu conhecimento é escasso e insatisfatório; pode ser o princípio do conhecimento, mas você, em seus pensamentos, empurrou parcamente a fronteira da ciência." [tradução livre]

Em suma, "medir é saber". Girolami então pontuou que dados derivados da observação e medição experimentais foram os responsáveis por conduzir o desenvolvimento da filosofia natural e impulsionar o estado-da-técnica da engenharia por todo o século XIX. Em suas palavras,

> "os dados sempre estiveram no coração da ciência e da prática na engenharia".

Com a irrupção da DCE, uma pergunta que se faz é: seria a DCE um novo ramo na árvore do futuro da ciência de dados? Enquanto uma resposta objetiva é aguardada, poderíamos intuir, com base na opinião de vários experts que se reuniram no DCEng Summit, realizado no último setembro, em Londres, que as engenharias não serão mais as mesmas daqui para a frente. É consensual que o big data abriu enormes oportunidades para praticamente todas as áreas da engenharia – Aeronáutica, Civil, Mecânica, Offshore, entre outras – haja vista o nível de detalhamento provido por muitos bancos de dados quanto no que tange à compreensão de variados fenômenos que foram observados e medidos experimentalmente ao longo de décadas. Entretanto, uma gama de desafios acompanha essa evolução furtivamente. Ética e privacidade na gestão de dados, lentidão da difusão tecnológica em países de baixa renda e carência de profissionais qualificados e currículos contemporâneos são alguns deles.

Ley et al., emitindo pontos de vista sobre como a DCE se projeta em suas áreas de atuação – estatística, engenharia e desenvolvimento de software –, concluíram que:

1. o pensamento centrado em dados tornou-se necessário em vários domínios do conhecimento e a riqueza por eles disponibilizada acelerará a pesquisa no âmbito da engenharia de maneira imensurável;
2. dados solitários não valem muita coisa e podem transmitir mensagens equivocadas se não forem analisados com cuidado e geridos de forma segura;
3. pelo fato de a educação baseada em dados ser uma habilidade indispensável para a formação de futuros engenheiros, as universidades, empresas e gestores devem se mobilizar para assegurar um currículo interdisciplinar que forme profissionais com "mente aberta" e explore habilidades flexíveis capazes de lidar com dados.

Na Engenharia Mecânica, em particular, o conceito de **ciência de dados mecanicista** (*mechanistic data science*, MDS) e sua incorporação na educação de engenheiros, bem como de estudantes de nível médio nos Estados Unidos foi recentemente debatida no 16th USCCM e na conferência MMLDT-CSET 2021. A proposta da MDS é explanada, por exemplo, nesta apresentação do Prof. Wing Kam Liu da Northwestern University.

8.4.4 Engenharia computacional no enfrentamento da Covid-19

A pandemia da Covid-19 desencadeou enormes desafios para a comunidade global. Concomitantemente, pesquisadores de diversas áreas mobilizaram-se para apresentar estratégias de enfrentamento à doença e propostas para mitigação dos riscos de contaminação do vírus SARS-CoV-2. A CSE não ficou de fora. A dinâmica dos fluidos computacional (computational fluid dynamics, CFD) contribuiu magnificamente para a elaboração de protocolos, equipamentos de proteção individual e coletiva, bem como para controle de engenharia em ambientes hospitalares. A seguir, parafraseamos três objetivos reportados pela Siemens Digital Industries Software em estudos de caso reais guiados por CFD:

entender a dinâmica espaço-temporal de partículas virais exaladas por seres humanos na forma de gotículas e aerossóis; aperfeiçoar o projeto de sistemas de ventilação interior, bem como de exaustão e filtração visando conforto e segurança hospitalar; projetar equipamentos respiratórios, dispositivos para esterilização, purificação e produção de vacinas.

8.5 Considerações finais

A datificação está se consolidando como o quarto paradigma da ciência e a engenharia será cada vez mais influenciada por uma cultura baseada em dados. No TRIL Lab, defendemos uma formação estratégica e trabalhamos para que nossos alunos e colaboradores se adaptem a um cenário que demanda cada vez mais habilidades flexíveis e interdisciplinaridade. Em um mundo centrado em dados, a engenharia computacional terá um leque incomensurável de oportunidades, compreendendo, modelando e resolvendo problemas de engenharia do mundo real.

O conteúdo deste livro tem o objetivo de proporcionar a nossos estudantes uma formação moderna e ampla em métodos numéricos com aplicações às ciências computacionais e engenharias. Entretanto, qualquer perfil profissional que se beneficie da computação científica terá nele um estímulo adicional para enveredar-se pelo frutífero e versátil universo das técnicas fundamentais da engenharia computacional.

CHAPTER

NINE

CONVERSÃO NUMÉRICA E PONTO FLUTUANTE

```
%matplotlib inline
```

9.1 Sistema binário

Simples exercícios de conversão numérica para introduzi-lo à computação numérica com Python.

9.1.1 Exercícios de conversão numérica

```
# (100)_2 -> base 10
c = int('100',base=2)
print(c)

# representação
print(1*2**2 + 0*2**1 + 0*2**0)

# (4)_10 -> base 2
# obs: note que '0b' indica que o número é binário
c = bin(4)
print(c)
```

```
4
4
0b100
```

```
# (222)_8
c = int('222',base=8)
print(c)

# representação
print(2*8**2 + 2*8**1 + 2*8**0)

# (146)_10 -> base 8

c = oct(146)
# obs: note que '0o' indica que o número é octal
print(c)
```

```
146
146
0o222
```

```
# (2AE4)_16
c = int('2ae4',base=16)
print(c)

# representação
# obs: A = 10; E = 14
print(2*16**3 + 10*16**2 + 14*16**1 + 4*166**0)

# (146)_10 -> base 8

c = oct(146)
# obs: note que '0o' indica que o número é octal
print(c)
```

```
10980
10980
0o222
```

```
"Brincando com Python e divisões sucessivas"

print('Esquema de divisões sucessivas:\n')

print(str(4) + ' | 2')
print( str( len(str(4))*' ') + '  ---')
print( str( 4 % 2) + '    ' + str(4 // 2) + ' | 2' )
print( str( 5*len(str(4))*' ') + '  ---')
print( str( 4*len(str(4))*' ') + str(4 % 2 % 2) + '    ' + str(4 // 2 // 2))
```

```
Esquema de divisões sucessivas:

4 | 2
   ---
0    2 | 2
       ---
     0   1
```

Exercício: estude a codificação do esquema acima. O que os operadores `//` e `%` estão fazendo?

9.2 Máquina binária

O código abaixo é um protótipo para implementação de uma máquina binária. Uma versão muito mais robusta e melhor implementada pode ser vista aqui: https://vnicius.github.io/numbiosis/conversor/index.html.

```
"""
Converte inteiro para binário
por divisões sucessivas.
! Confronte com a função residente 'bin()'
"""
```

(continues on next page)

(continued from previous page)

```
def int2bin(N):

    b = [] # lista auxiliar

    # divisões sucessivas
    while N >= 2:
        b.append(N % 2)
        N = N//2

    b.append(N)
    b.reverse()
    b = [str(i) for i in b] # converte para string
    s = ''
    s = s.join(b)

    return s # retorna string


"""
Converte parte fracionária para binário
por multiplicações sucessivas.
"""
def frac2bin(Q):

    count = 0 # contador (limite manual posto em 10!)
    b = []  # lista auxiliar

    # multiplicações sucessivas
    Q *= 2
    while Q > 0 and count <= 10:
        if Q > 1:
            Q = Q-1
            b.append(1)
        else:
            b.append(0)
        Q *= 2
        count += 1

    b = [str(i) for i in b] # converte para string
    s = ''
    s = s.join(b)

    return s # retorna string


def convert(app,btn):
    print(btn)



# Função principal
def main():

    # Pré-criação da interface com usuário

    # todo: tratamento de exceção no tipo de entrada
```

(continues on next page)

(continued from previous page)

```
    #       contagem de casas decimais no caso de dízimas
          print('*** MÁQUINA BINÁRIA ***')
    #      N = input('Selecione a parte inteira:\n')
    #      Q = input('Selecione a parte fracionária:\n')
    #      print('Seu número é: ' + int2bin( int(N) ) + '.' + frac2bin( float(Q) )  +
↪'.')
    #      print('*** ***')


if __name__ == "__main__":
    main()
```

```
*** MÁQUINA BINÁRIA ***
```

9.3 Sistema de ponto flutuante

9.3.1 A reta "perfurada"

Como temos estudado, a matemática computacional opera no domínio $\mathbb{F}$, de pontos flutuantes, ao invés de trabalhar com números reais (conjunto $\mathbb{R}$). Vejamos um exemplo:

Exemplo: Considere o sistema de ponto flutuante $\mathbb{F}(2, 3, -1, 2)$. Determinemos todos os seus números representáveis:

Como a base é 2, os dígitos possíveis são 0 e 1 com mantissas:

- 0.100
- 0.101
- 0.110
- 0.111

Para cada expoente no conjunto $e = \{-1, 0, 1, 2\}$, obteremos 16 números positivos, a saber:

- $(0.100 \times 2^{-1})_2 = (0.01)_2 = 0.2^0 + 0.2^{-1} + 1.2^{-2} = 1/4$
- $(0.100 \times 2^{0})_2 = (0.1)_2 = 0.2^0 + 1.2^{-1} = 1/2$
- $(0.100 \times 2^{1})_2 = (1.0)_2 = 1.2^0 + 0.2^{-1} = 1$
- $(0.100 \times 2^{2})_2 = (10.0)_2 = 1.2^1 + 0.2^1 + 0.2^{-1} = 2$
- $(0.101 \times 2^{-1})_2 = (0.0101)_2 = 0.2^0 + 0.2^{-1} + 1.2^{-2} + 0.2^{-3} + 1.2^{-4} = 5/16$
- $(0.101 \times 2^{0})_2 = (0.101)_2 = 0.2^0 + 1.2^{-1} + 0.2^{-2} + 1.2^{-3} = 5/8$
- $(0.101 \times 2^{1})_2 = (1.01)_2 = 1.2^0 + 0.2^{-1} + 1.2^{-2} = 1$
- $(0.101 \times 2^{2})_2 = (10.1)_2 = 1.2^1 + 0.2^1 + 0.2^{-1} = 2$

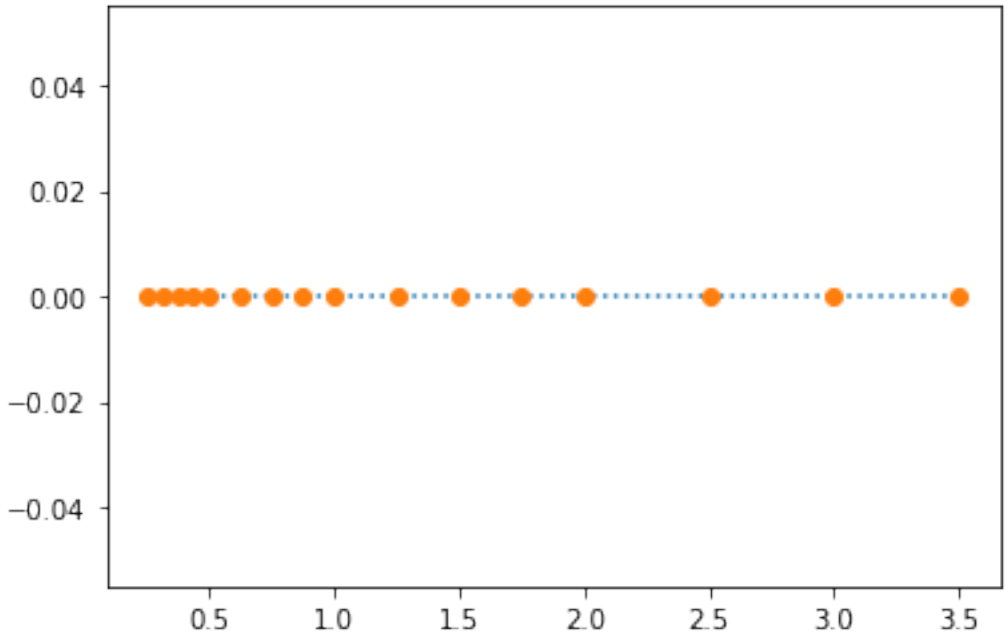
$(\ldots)$

Fazendo as contas para os números restantes, obtemos a seguinte tabela:

	m	0.100	0.101	0.110	0.111
e					
-1		1/4	5/16	3/8	7/16
0		1/2	5/8	3/4	7/8
1		1	5/4	3/2	7/4
2		2	5/2	3	7/2

Na reta real, esses valores ficariam dispostos da seguinte forma:

```
from matplotlib.pyplot import plot
x = [1/4,1/2,1,2,5/16,5/8,5/4,5/2,3/8,3/4,3/2,3,7/16,7/8,7/4,7/2]
x = sorted(x)

plot(x,16*[0],':')
plot(x,16*[0],'o');
```

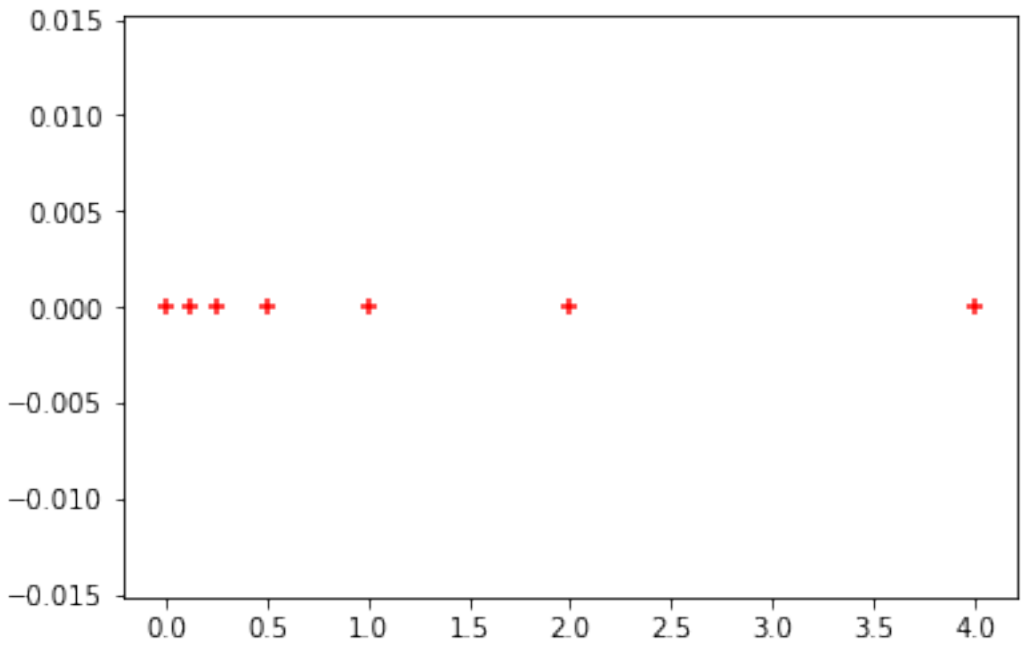


Isto é, $\mathbb{F}$ é uma reta "perfurada", para a qual apenas 16 números positivos, 16 simétricos destes e mais o 0 são representáveis. Logo, o conjunto contém apenas 33 elementos.

9.4 Simulador de $\mathbb{F}$

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

def simulacao_F(b,t,L,U):
    x = []
    epsm = b**(1-t) # epsilon de máquina
    M = np.arange(1.,b-epsm,epsm)
    print(M)

    E = 1
    for e in range(0,U+1):
        x = np.concatenate([x,M*E])
        E *= b
    E = b**(-1)

    y = []
    for e in range(-1,L-1,-1):
        y = np.concatenate([y,M*E])
        E /= b
    yy = np.asarray(y)
    xx = np.asarray(x)
    x = np.concatenate([yy,np.array([0.]),xx])
    return x

Y = simulacao_F(2,2,-3,2)
X = np.zeros(Y.shape)

plt.scatter(Y,X,c='r',marker='+');
```

```
[1.]
```



9.5 Limites de máquina para ponto flutuante

```
import numpy as np

# limites de máquina para ponto flutuante
#help(np.finfo)

# epsilon de máquina para tipo float (64 bits)
print('Epsilon de máquina do numpy - 64 bits')
print(np.finfo(float).eps)

# função para calculo do epsilon: erro relativo
def eps_mach(func=float):
    eps = func(1)
    while func(1) + func(eps) != func(1):
        epsf = eps
        eps = func(eps) / func(2)
    return epsf

# número máximo representável
print('número máximo representável')
print(np.finfo(float).max)

# número mínimo representável
print('número mínimo representável')
print(np.finfo(float).min)

# número de bits no expoente
print('número de bits no expoente')
print(np.finfo(float).nexp)

# número de bits na mantissa
```

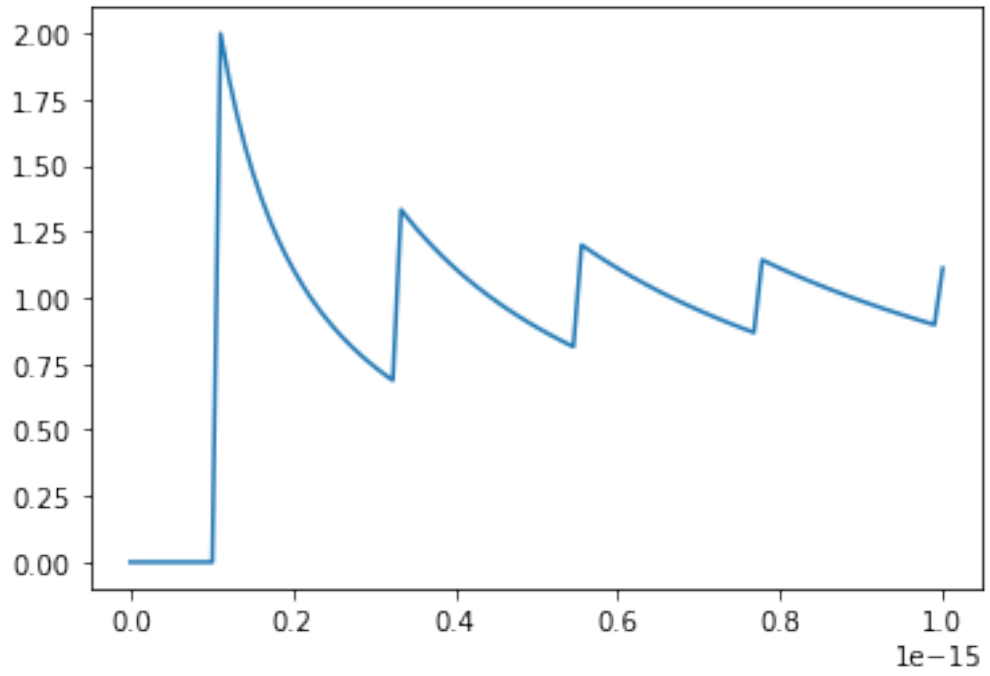
(continues on next page)

(continued from previous page)

```
print('número de bits na mantissa')
print(np.finfo(float).nmant)
```

```
Epsilon de máquina do numpy - 64 bits
2.220446049250313e-16
número máximo representável
1.7976931348623157e+308
número mínimo representável
-1.7976931348623157e+308
número de bits no expoente
11
número de bits na mantissa
52
```

```
from matplotlib.pyplot import plot

x = np.linspace(1e-15,1e-20,num=100)
f = ((1+x)-1)/x
plot(x,f);
```



CHAPTER

TEN

ERROS NUMÉRICOS E SEUS EFEITOS

```
%matplotlib inline
```

10.1 Motivação

Exemplo: Avaliar o polinômio $P(x) = x^3 - 6x^2 + 4x - 0.1$ no ponto $x = 5.24$ e comparar com o resultado exato.

Vamos fazer o seguinte:

1. Com uma calculadora, computar o valor de $P(5.24)$ e assuma que este é seu valor exato.
2. Calcular $P(5.24)$ usando arredondamento com dois dígitos de precisão.

Passo 1

Faça as suas contas! Suponhamos que seja -0.007776.

Passo 2

Vamos "imitar" as contas feitas na mão…

```
# parcelas

p1 = 5.24**3
print('p1: {0:.20g}'.format(p1)) # 20 dígitos significativos
print('p1 (com arredondamento): {0:.2f}'.format(p1))

print('\n')

p2 = - 6*5.24**2
print('p2: {0:.20g}'.format(p2))
print('p2 (com arredondamento): {0:.2f}'.format(p2))

print('\n')

p3 = 4*5.24
print('p3: {0:.20g}'.format(p3))
print('p3 (com arredondamento): {0:.2f}'.format(p3))

print('\n')

p4 = - 0.1
print('p4: {0:.20g}'.format(p4))
print('p4 (com arredondamento): {0:.2f}'.format(p4))
```

(continues on next page)

(continued from previous page)

```
print('\n')

Px = p1 + p2 + p3 + p4
print('Px: {0:.20g}'.format(Px))
print('Px: (com arredondamento): {0:.2f}'.format(Px))
```

```
p1: 143.87782400000000393
p1 (com arredondamento): 143.88


p2: -164.74560000000002447
p2 (com arredondamento): -164.75


p3: 20.960000000000000853
p3 (com arredondamento): 20.96


p4: -0.10000000000000000555
p4 (com arredondamento): -0.10


Px: -0.0077760000000196838332
Px: (com arredondamento): -0.01
```

Conclusão: o cálculo com dois dígitos afeta o resultado drasticamente!

Agora, vamos comparar o resultado de se avaliar $P(5.24)$ com as duas formas do polinômio e 16 dígitos de precisão:

```
# ponto de análise
x = 5.24

# P1(x)
P1x = x**3 - 6*x**2 + 4*x - 0.1
print('{0:.16f}'.format(P1x))

# P2(x)
P2x = x*(x*(x - 6) + 4) - 0.1 # forma estruturada (forma de Hörner)
print('{0:.16f}'.format(P2x))
```

```
-0.0077760000000197
-0.0077759999999939
```

O que temos acima? Dois valores levemente distintos. Se computarmos os erros absoluto e relativo entre esses valores e nosso valor supostamente assumido como exato, teríamos:

Erros absolutos

```
x_exato = -0.007776
EA1 = abs(P1x - x_exato)
print(EA1)

EA2 = abs(P2x - x_exato)
print(EA2)
```

```
1.968390728190883e-14
6.1287780406260595e-15
```

Claro que $EA_1 > EA_2$. Entretanto, podemos verificar pelo seguinte teste lógico:

```
# teste é verdadeiro
EA1 > EA2
```

```
True
```

Erros relativos

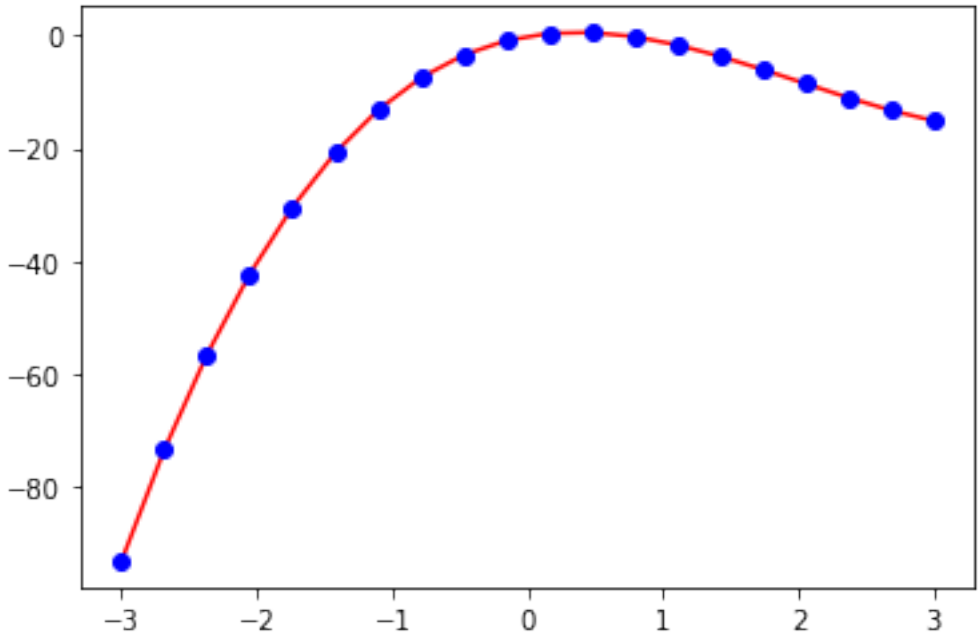
Os erros relativos também podem ser computados como:

```
ER1 = EA1/abs(x_exato)
print(ER1)

ER2 = EA2/abs(x_exato)
print(ER2)
```

```
2.5313666772002096e-12
7.881659002862731e-13
```

Gráfico de $P(x)$

```
import numpy as np
import matplotlib.pyplot as plt

# eixo x com 20 pontos
x = np.linspace(-3,3,num=20,endpoint=True)

# plotagem de P1(x) e P2(x)
P1x = lambda x: x**3 - 6*x**2 + 4*x - 0.1
P2x = lambda x: x*(x*(x - 6) + 4) - 0.1
plt.plot(x,P1x(x),'r',x,P2x(x),'bo');
```



10.1.1 Função de Airy

A função de Airy é solução da equação de Schrödinger da mecânica quântica. Muda o comportamento de oscilatório para exponencial.

Abaixo, vamos criar uma função aproximada (perturbada) para a função de Airy (assumindo-a como uma aproximação daquela que é exata) e outra para calcular diretamente o erro relativo para valores dessas funções.

```
from scipy import special
import matplotlib.pyplot as plt

# eixo x
x = np.linspace(-10, -2, 100)

# funções de Airy e derivadas (solução exata)
ai, aip, bi, bip = special.airy(x)

# função de Airy (fazendo papel de solução aproximada)
ai2 = 1.1*ai + 0.05*np.cos(x)
```

Podemos usar o conceito de *função anônima* para calcular diretamente o **erro relativo percentual** para cada ponto x:

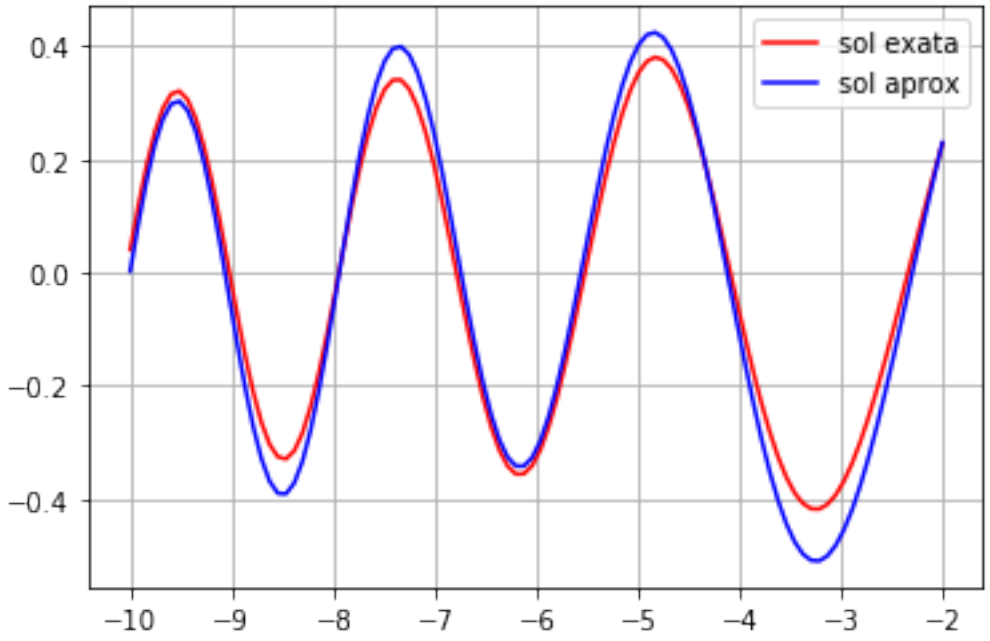
$$ER_p(x) = \frac{\mid f_{aprox}(x) - f_{ex}(x) \mid}{\mid f_{ex}(x) \mid},$$

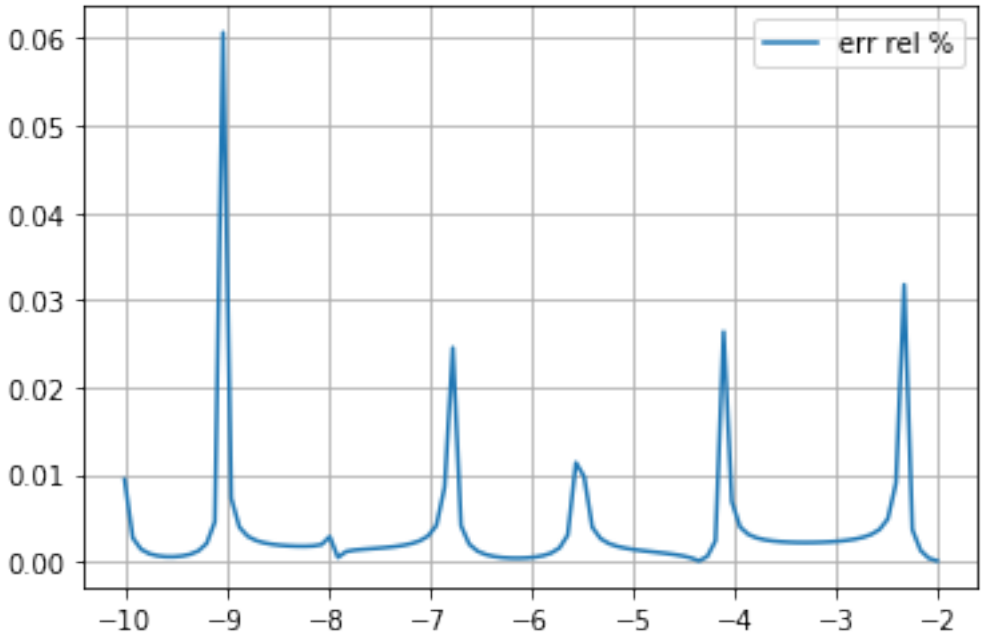
onde $f_{aprox}(x)$ é o valor da função aproximada (de Airy) e onde $f_{ex}(x)$ é o valor da função exata (de Airy).

```
# define função anônima para erro relativo
r = lambda fex,faprox: (np.abs(fex-faprox)/np.abs(fex))/100

# calcula erro relativo para função de Airy e sua aproximação
rel = r(ai,ai2)
```

A seguir, mostramos a plotagem das funções exatas e aproximadas, bem como do erro relativo pontual.

```
# plotagens
plt.plot(x, ai, 'r', label='sol exata')
plt.plot(x, ai2, 'b', label='sol aprox')
plt.grid()
plt.legend(loc='upper right')
plt.show()

plt.plot(x,rel,'-', label='err rel %')
plt.grid()
plt.legend(loc='upper right');
```





10.2 Erro de cancelamento

Ocorre quando números de grandezas próximas são subtraídos. No exemplo, a seguir, induzimos uma divisão por zero usando o valor do épsilon de máquina ϵ_m ao fazer

$$\frac{1}{(1+0.25\epsilon_m)-1}$$

Isto ocorre porque o denominador sofre um *cancelamento subtrativo*, quando, para a matemática precisa, deveria valer $0.25\epsilon_m$.

10.3 Propagação de erros

Vamos comparar duas situações. Calcular

$$e^{-v} = \sum_{i=0}^{\infty}(-1)^i \frac{v^i}{i!}$$

e comparar com a identidade $$e^{-v} = \frac{1}{e^v}.$$

```
# somatória (primeiros 20 termos)
v = 5.25
s = 0
for i in range(20):
    print('{0:5g}'.format(s))
    s += ((-1)**i*v**i)/np.math.factorial(i)

print('\ncaso 1: {0:5g}'.format(s))

print('caso 2: {0:5g}'.format(1/np.exp(v)))
```

```
    0
    1
-4.25
9.53125
-14.5859
17.0679
-16.1686
12.9133
-8.89814
5.41562
-2.93407
1.44952
-0.642652
0.272671
-0.0969786
0.0416401
-0.00687642
0.00904307
0.00412676
0.00556069

caso 1: 0.00516447
caso 2: 0.00524752
```

CHAPTER

ELEVEN

ANÁLISE GRÁFICA, LOCALIZAÇÃO E REFINAMENTO DE RAÍZES

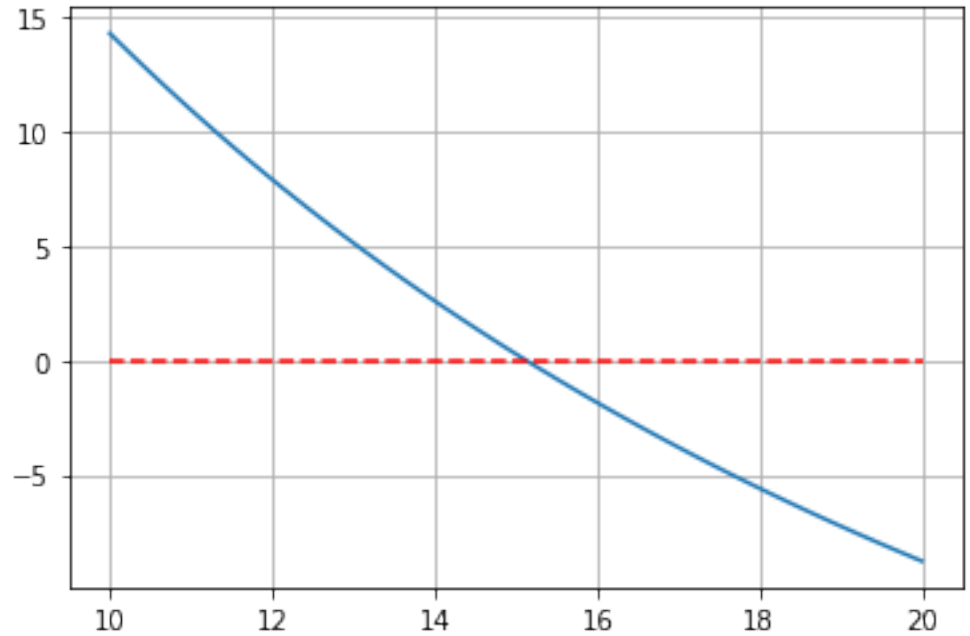
11.1 Estudo de caso: salto do paraquedista

```
%matplotlib inline
```

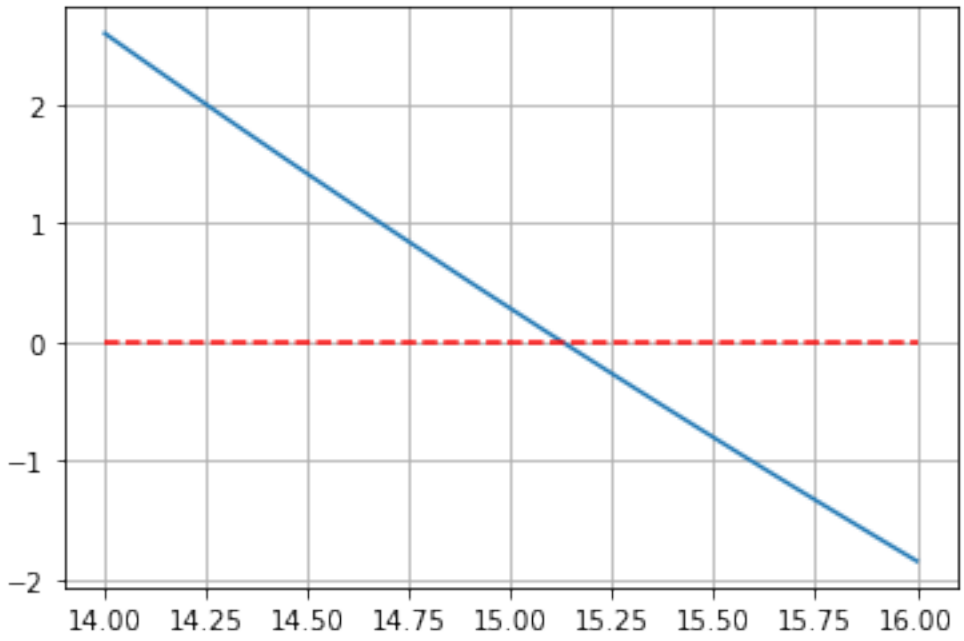
```
import numpy as np
import matplotlib.pyplot as plt

# parametros
t = 12.0
v = 42.0
m = 70.0
g = 9.81
```

```
# localizacao
a,b = 1,20
c = np.linspace(a,b,20)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

plt.plot(c,f,'k--o');
plt.xlabel('c')
plt.ylabel('f(c)')
plt.title('Variação do coef. de arrasto')
plt.grid(True)
```

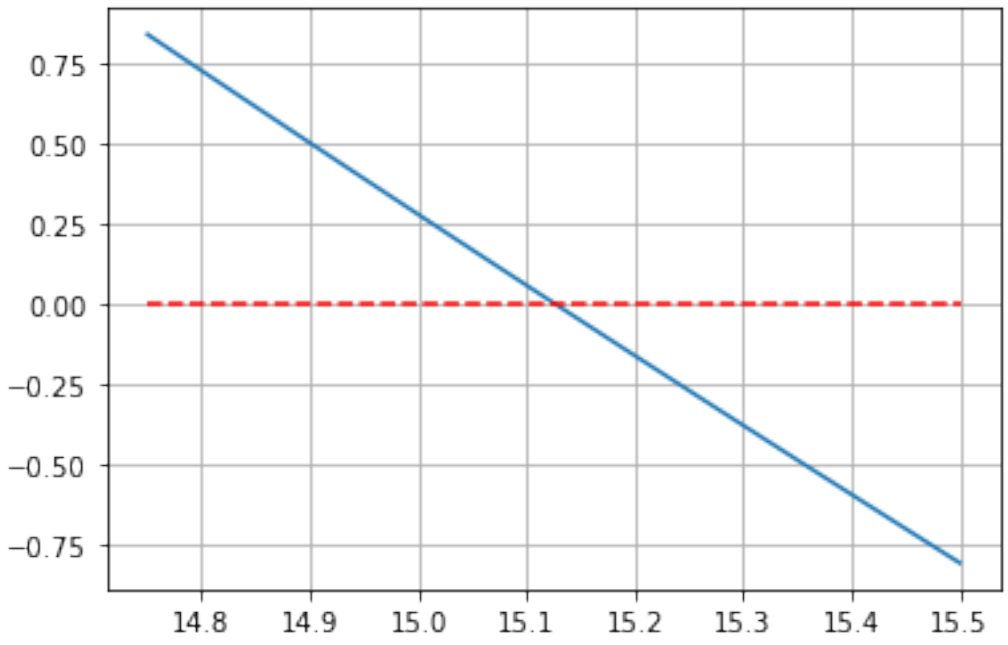

```
# refinamento
a,b = 10,20
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

plt.plot(c,f)
plt.plot(c,0*c,'r--')
plt.grid()
```



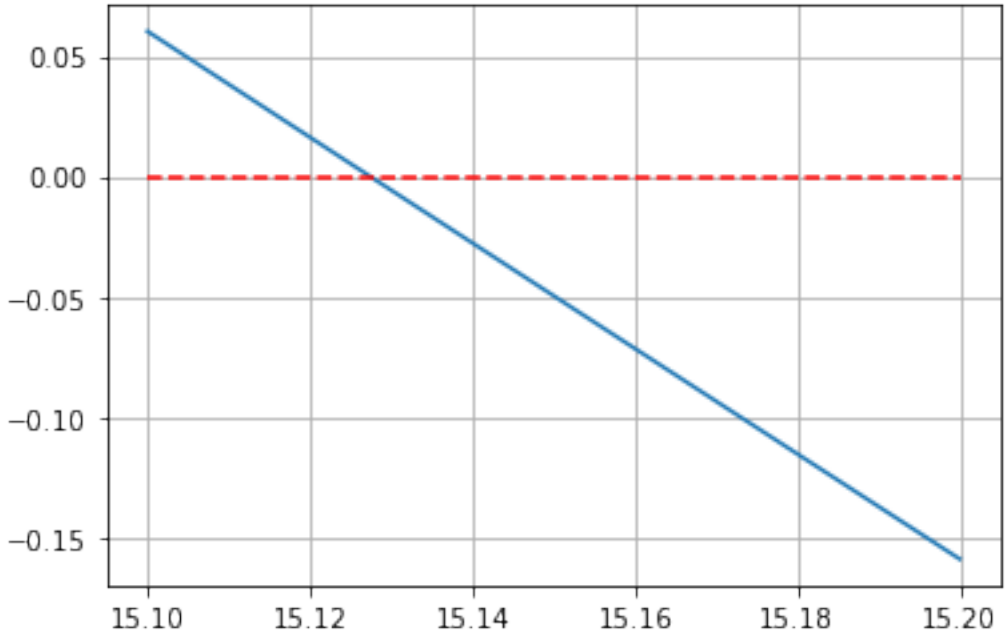
```
# refinamento
a,b = 14,16
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

plt.plot(c,f)
plt.plot(c,0*c,'r--')
plt.grid()
```




```
# refinamento
a,b = 14.75,15.5
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

plt.plot(c,f)
plt.plot(c,0*c,'r--')
plt.grid()
```




```
# refinamento
a,b = 15.1,15.2
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

plt.plot(c,f)
plt.plot(c,0*c,'r--')
plt.grid()
```

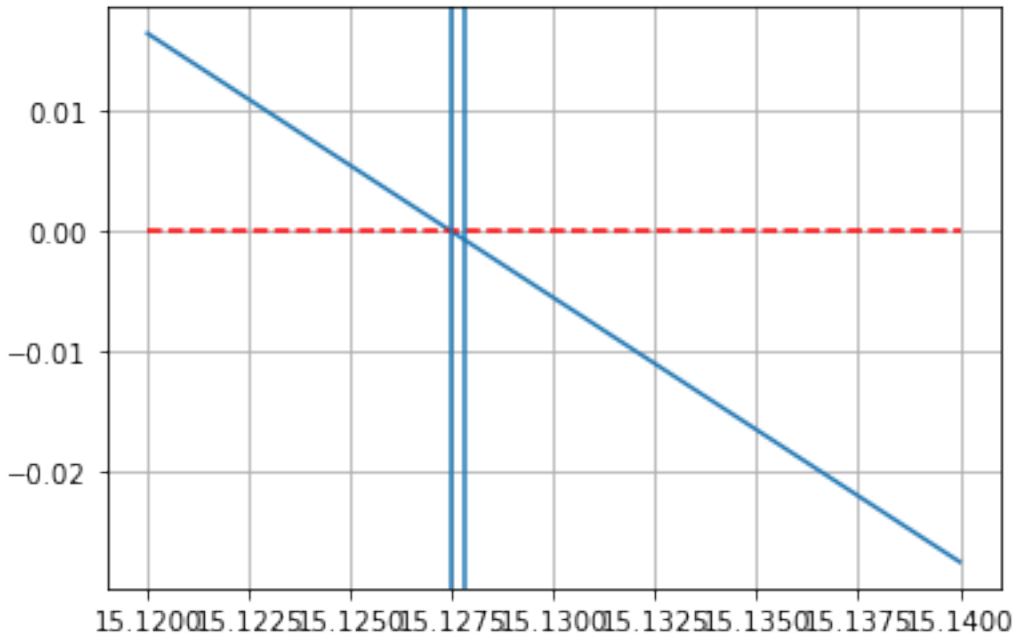



```
# refinamento
a,b = 15.12,15.14
```

(continues on next page)

(continued from previous page)

```
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

plt.plot(c,f)
plt.plot(c,0*c,'r--')
plt.axvline(x=15.1278)
plt.axvline(x=15.1275)
plt.grid()
```



11.2 Métodos iterativos

A primeira estrutura de controle fundamental para computação numérica é o laço `for`. Vamos ver como usá-lo para computar somatórias e produtórios.

Por exemplo, vamos computar:

$s = \sum_{i=1}^{n} i = i + i + ... + i = s_1 + s_2 + ... + s_n$, para um valor n finito.

Podemos fazer isto da seguinte maneira:

```
n = 10

s = 0.0
for i in range(n):
    s += i
```

Detalhando... Considere o par (i, s_i). Há soma abaixo?

```
s = 0.
for i in range(n):
    print(i+1,s)
```

```
1 0.0
2 0.0
3 0.0
4 0.0
5 0.0
6 0.0
7 0.0
8 0.0
9 0.0
10 0.0
```

```
# Aqui há um incremento constante de 1
s = 0.
for i in range(n):
    s = s + 1
    print(i+1,s)
```

```
1 1.0
2 2.0
3 3.0
4 4.0
5 5.0
6 6.0
7 7.0
8 8.0
9 9.0
10 10.0
```

O exemplo anterior também poderia ser reproduzido como:

```
s = 0.
for i in range(n):
    s += 1 # produz o mesmo que s <- s + 1
    print(i+1,s)
```

```
1 1.0
2 2.0
3 3.0
4 4.0
5 5.0
6 6.0
7 7.0
8 8.0
9 9.0
10 10.0
```

No próximo exemplo, o incremento não é mais constante:

```
s = 0.
for i in range(n):
    s += i
    print(i+1,s)
```

```
1 0.0
2 1.0
3 3.0
```

(continues on next page)

(continued from previous page)

```
4 6.0
5 10.0
6 15.0
7 21.0
8 28.0
9 36.0
10 45.0
```

11.3 Determinação de raízes por força bruta

No computador, sabemos que uma função matemática $f(x)$ pode ser representada de duas formas principais:

- através de uma função programada (em Python, por exemplo) que retorna o valor da função para um dado argumento
- uma coleção de pontos $(x, f(x))$ na forma de uma tabela.

A segunda forma é bem mais útil para análise gráfica. Esta forma é também adequada para resolver problemas de determinação de raízes e de otimização com simplicidade. No primeiro caso, basta pesquisar todos os pontos e procurar onde a função cruza o eixo x, como fizemos anteriormente. No segundo caso, buscamos um ponto de mínimo ou máximo local, ou global.

Abordagens que seguem esse caminho podem chegar a examinar uma grande quantidade de pontos. Por essa razão, são chamados de métodos de *força bruta*, isto é, não seguem uma técnica elaborada.

11.3.1 Algoritmo numérico

Em geral, queremos resolver o problema $f(x) = 0$ especialmente quando f é não-linear. Para isso, desejamos encontrar os x onde f cruza o eixo. Um algoritmo em força bruta deverá percorrer todos os pontos sobre a curva e verificar se um ponto está abaixo do eixo e seu sucessor imediato está acima, ou vice-versa. Se isto ocorrer, então deve haver uma raiz neste intervalo.

Algoritmo. Dado um conjunto de $n+1$ pontos (x_i, y_i), $y_i = f(x_i)$, $i = 0, \ldots, n$, onde $x_0 < \ldots < x_n$. Verificamos se $y_i < 0$ e se $y_{i+1} > 0$. Uma expressão compacta para esta checagem é o teste $y_i\, y_{i+1} < 0$. Se o produto for negativo, então a raiz de f está no intervalo $[x_i, x_{i+1}]$. Assumindo uma variação linear entre os pontos, temos a aproximação

$$f(x) \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i.$$

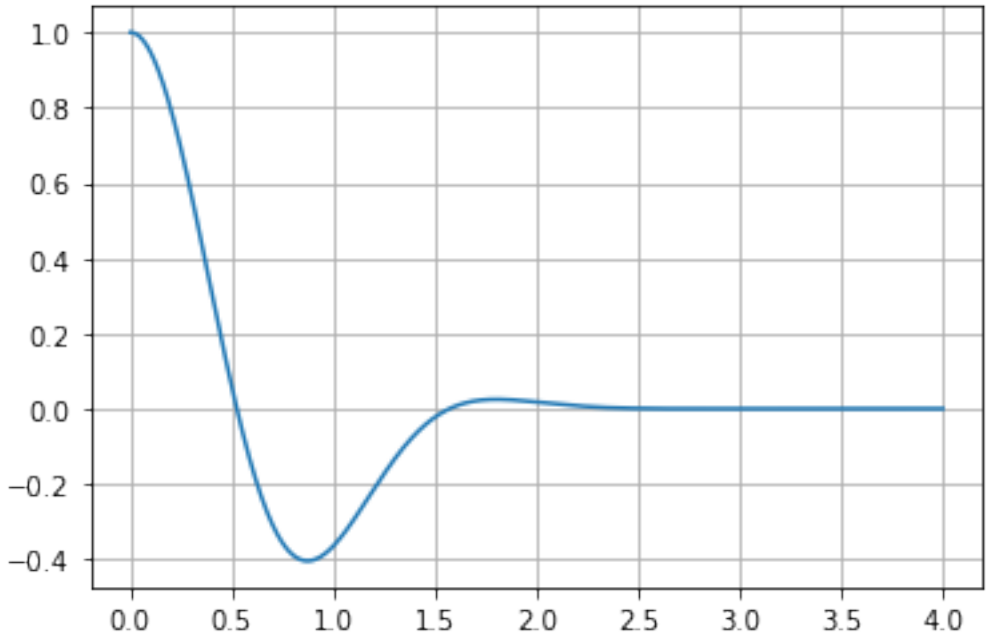
Logo, $f(x) = 0$ implica que a raiz é

$$x = x_i - \frac{x_{i+1} - x_i}{y_{i+1} - y_i} y_i.$$

Exemplo. Encontre a raiz da função $f(x) = \exp(-x^2)\cos(3x)$ usando o algoritmo de força bruta.

Vamos plotar esta função apenas para visualizar seu comportamento.

```
from numpy import exp, cos

f = lambda x: exp(-x**2)*cos(3*x)
x = np.linspace(0,4,1000)
plt.plot(x,f(x)); plt.grid()
```



Nesta plotagem, vemos claramente que a função possui duas raizes: uma próxima de $x = 0.5$ e outra em $x = \pi/6$.

Implementemos o algoritmo.

```
def forca_bruta(f,a,b,n):
    from numpy import linspace
    x = linspace(a,b,n)
    y = f(x)
    raizes = []
    for i in range(n-1):
        if y[i]*y[i+1] < 0:
            raiz = x[i] - (x[i+1] - x[i])/(y[i+1] - y[i])*y[i]
            raizes.append(raiz)
    if len(raizes) == 0:
        print('Nenhuma raiz foi encontrada')
    return raizes
```

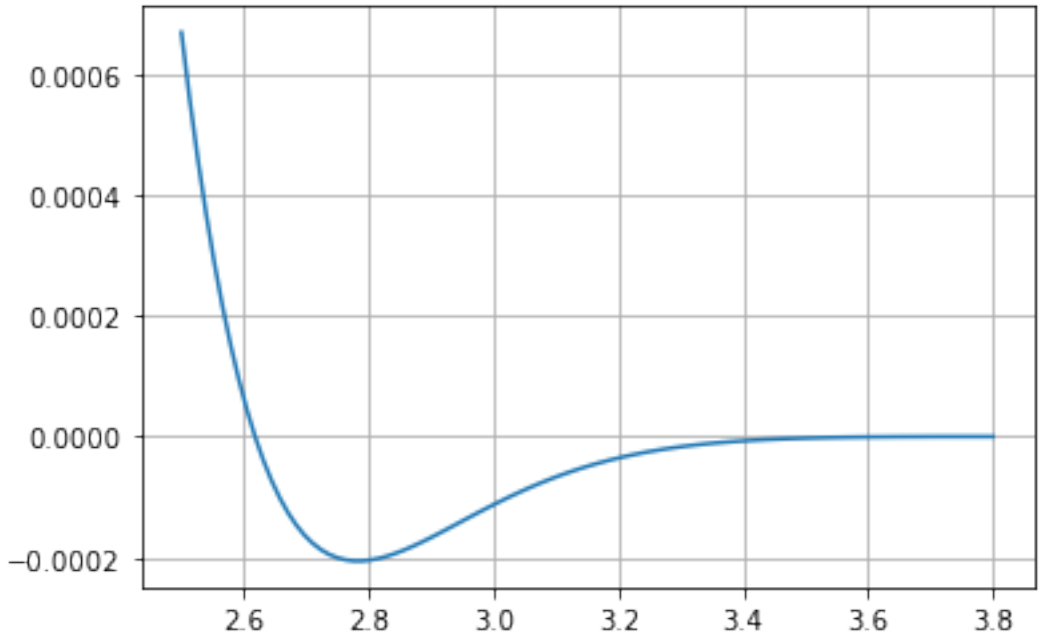
Agora aplicamos o algoritmo na mesma função.

```
a,b,n = 0,4,1000
raizes = forca_bruta(f,a,b,n)
print(raizes)
```

```
[0.5236017411236913, 1.5708070694852787, 2.6180048381439596, 3.665219264613299]
```

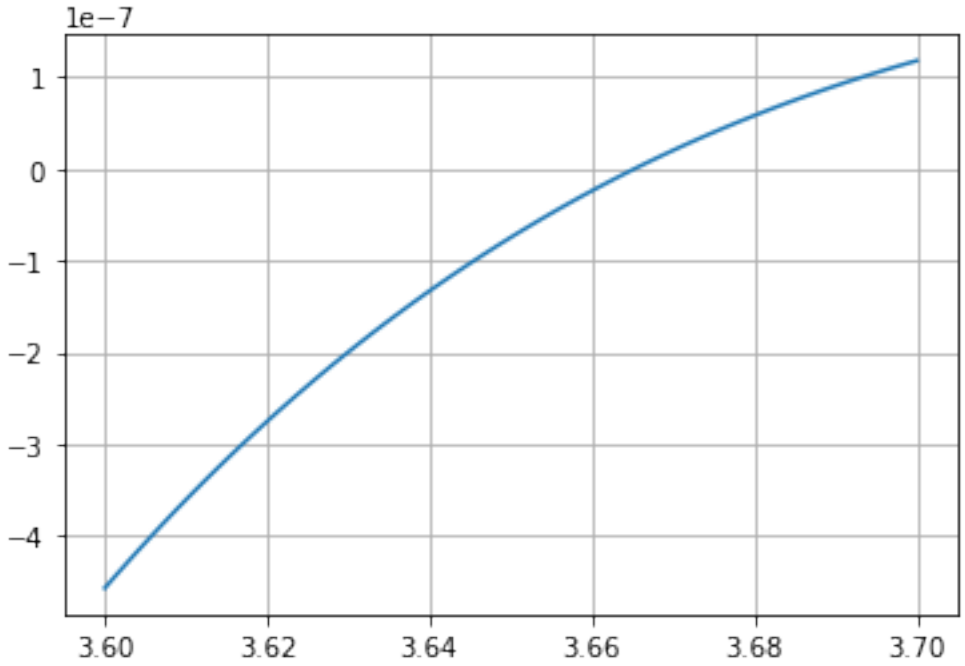
Temos, na verdade, 4 raízes! Plotemos o gráfico ampliado no intervalo [2.5,3.8].

```
x2 = np.linspace(2.5,3.8,100)
plt.plot(x2,f(x2)); plt.grid()
```



Conseguimos enxergar mais uma raiz. Agora, plotemos um pouco mais ampliado entre [3.6,3.7].

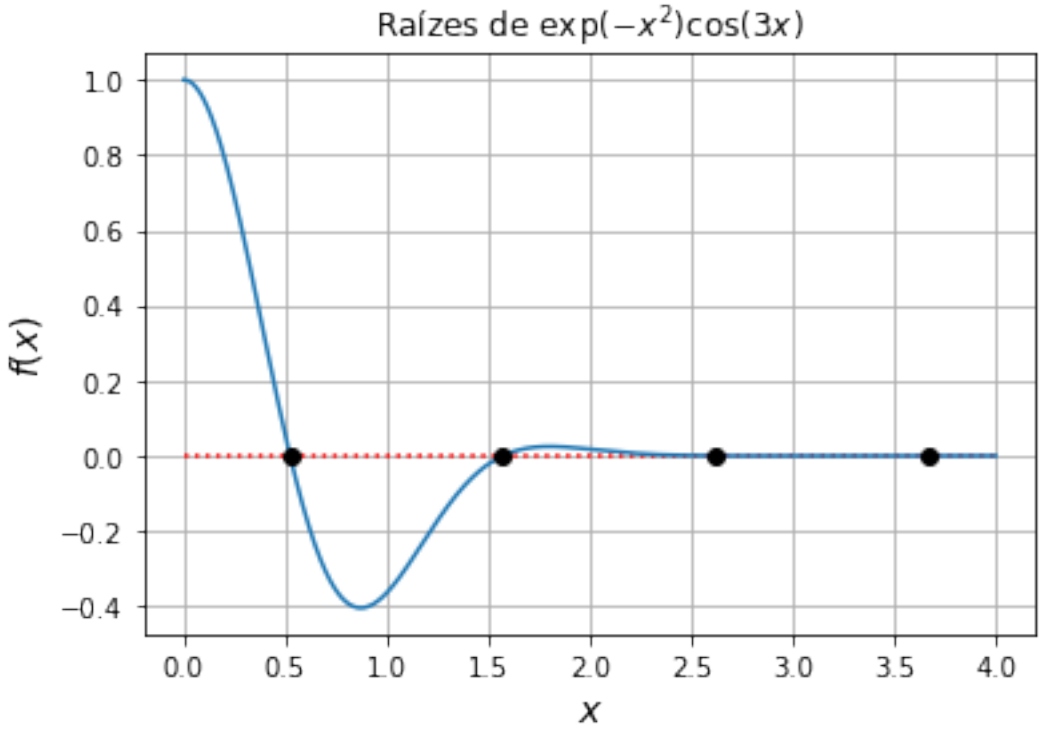
```
x3 = np.linspace(3.6,3.7,100)
plt.plot(x3,f(x3)); plt.grid()
```



Dessa forma, podemos identificar que, de fato existe uma quarta raiz.

Este exemplo mostrou uma aplicação do método de força bruta para determinação de raízes. Para finalizar, podemos embelezar o gráfico.

```
r = np.array(raizes) # vetoriza a lista
plt.plot(x,0*f(x),'r:',x,f(x),'-',r,np.zeros(4),'ok',)
plt.xlabel('$x$',fontsize=14)
plt.ylabel('$f(x)$',fontsize=14)
plt.grid()
plt.title('Raízes de $\exp(-x^2)\cos(3x)$');
```



CHAPTER

TWELVE

IMPLEMENTAÇÕES DO MÉTODO DA BISSEÇÃO

```
%matplotlib inline
```

```
"""MB Metodo da bissecao para funcoes unidimensionais
entrada:
    f: uma string dependendo de x, i.e., a funcao
           (e.g., 'x^2 + 1', 'x^2*cos(x)', etc.)
    a: limite inferior do dominio
    b: limite superior do dominio
    tol: tolerancia
    N: numero maximo de iteracoes do metodo

saida:
    xm: raiz da funcao
"""

import inspect, re

def bissecao(f,a,b,tol,N,var):

    # TODO identificar a variável usada na função
    #      Aqui, tentei assumir que apenas uma era usada (e.g. 'x'),
    #      mas foi complicado generalizar quando há objeto numpy
    #var = re.search('[a-zA-Z]+',f)
    #var = var.group()

    # cria função anônima
    f = eval('lambda ' + var + ' :' + f)

    # Se função não for de uma variável, lança erro.
    # Mais aplicável se o caso geral fosse implementado.
    if len(inspect.getfullargspec(f).args) - 1 > 0:
        raise ValueError('O código é válido apenas para uma variável.')

    # calcula valor da função nos extremos
    fa = f(a)
    fb = f(b)

    # verifica sinal da função para o intervalo passado
    if fa*fb >= 0:
        raise ValueError('A função deve ter sinais opostos em a e b!')

    # flag usada para prevenir a obtenção da raiz
    # antes de o intervalo ter sido
```

(continues on next page)

(continued from previous page)

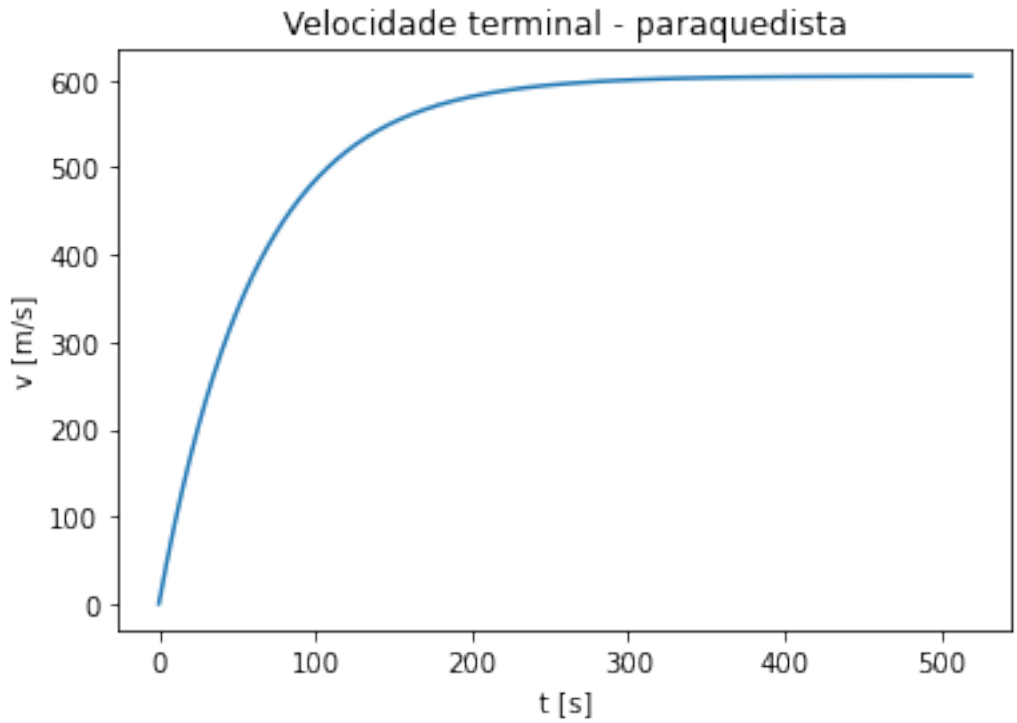
```
    # suficientemente reduzido
    done = 0;

    # loop principal

    # bisecta o intervalo
    xm = (a+b)/2

    i = 1 # contador

    while abs(a-b) > tol and ( not done or N != 0 ):
    # avalia a função no ponto médio
        fxm = f(xm)
        print("(i = {0:d}) f(xm)={1:f} | f(a)={2:f} | f(b)={3:f}".format(i,fxm,fa,fb))

        if fa*fxm < 0:        # Raiz esta à esquerda de xm
            b = xm
            fb = fxm
            xm = (a+b)/2
        elif fxm*fb < 0:      # Raiz esta à direita de xm
            a = xm
            fa = fxm
            xm = (a+b)/2
        else:                 # Achamos a raiz
            done = 1

        N -= 1                # Atualiza passo
        i += 1                # Atualiza contador

    print("Solução encontrada: {0}".format(xm))

    return xm
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import bisect, newton

# Dados de entrada

t = np.arange(0,520,1)    # tempo [s]
c = 1.46    # coeficiente de arrasto [kg/s]
m = 90      # massa [kg]
g = 9.81    # constante de gravidade [m/s2]

# Dados de saída

## velocidade terminal [m/s]
v_ms1 = (g*m/c)*(1 - np.exp((-c/m)*t))

# velocidade terminal [km/h]
v_kh1 = (1/3.6)*v_ms1;

# gráfico tempo x velocidade
plt.figure
plt.plot(t,v_ms1)
plt.xlabel('t [s]')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('v [m/s]')
plt.title('Velocidade terminal - paraquedista');
```




```
import sympy as sp

time = 12      # tempo [s]
mass = 70      # massa [kg]
vel = 42      # velocidade [m/s]
grav = 9.81    # constante de gravidade [m/s2]

# defino variáveis simbólicas
g,m,t,v,c = sp.symbols('g,m,t,v,c')

# expressão geral
f_g = (g*m/c)*(1 - sp.exp((-c/m)*t)) - v

# expressão particular com valores substituídos
# convertida para string
f_s = str(f_g.subs({'g':grav,'m':mass,'v':vel,'t':time}))

# TODO
# para esta função, teremos que substituir 'exp' por 'np.exp')
print('f(c) = ' + f_s + '\n')
f_s = '-42 + 686.7*(1 - np.exp(-6*c/35))/c'

# resolve bisseção
xm = bissecao(f_s,12,16,1e-5,100,'c')
```

```
f(c) = -42 + 686.7*(1 - exp(-6*c/35))/c

```

(continues on next page)

(continued from previous page)

```
(i = 1) f(xm)=2.600284 | f(a)=7.910578 | f(b)=-1.844622
(i = 2) f(xm)=0.281205 | f(a)=2.600284 | f(b)=-1.844622
(i = 3) f(xm)=-0.804573 | f(a)=0.281205 | f(b)=-1.844622
(i = 4) f(xm)=-0.267556 | f(a)=0.281205 | f(b)=-0.804573
(i = 5) f(xm)=0.005337 | f(a)=0.281205 | f(b)=-0.267556
(i = 6) f(xm)=-0.131479 | f(a)=0.005337 | f(b)=-0.267556
(i = 7) f(xm)=-0.063164 | f(a)=0.005337 | f(b)=-0.131479
(i = 8) f(xm)=-0.028937 | f(a)=0.005337 | f(b)=-0.063164
(i = 9) f(xm)=-0.011806 | f(a)=0.005337 | f(b)=-0.028937
(i = 10) f(xm)=-0.003236 | f(a)=0.005337 | f(b)=-0.011806
(i = 11) f(xm)=0.001050 | f(a)=0.005337 | f(b)=-0.003236
(i = 12) f(xm)=-0.001093 | f(a)=0.001050 | f(b)=-0.003236
(i = 13) f(xm)=-0.000022 | f(a)=0.001050 | f(b)=-0.001093
(i = 14) f(xm)=0.000514 | f(a)=0.001050 | f(b)=-0.000022
(i = 15) f(xm)=0.000246 | f(a)=0.000514 | f(b)=-0.000022
(i = 16) f(xm)=0.000112 | f(a)=0.000246 | f(b)=-0.000022
(i = 17) f(xm)=0.000045 | f(a)=0.000112 | f(b)=-0.000022
(i = 18) f(xm)=0.000012 | f(a)=0.000045 | f(b)=-0.000022
(i = 19) f(xm)=-0.000005 | f(a)=0.000012 | f(b)=-0.000022
Solução encontrada: 15.127429962158203
```

12.1 Tarefas

- Melhore o código Python tratando os TODOs:

Tente generalizar o código da bisseção para que identifique automaticamente a variável de entrada utilizada pelo usuário (use expressões regulares e remova o argumento `var` da definição da função).

Note que o trecho simbólico abaixo foi necessário para substituir a função da chamada `exp`, não interpretada por `eval` por uma nova string que usasse `np.exp`.

```
# TODO
# para esta função, teremos que substituir 'exp' por 'np.exp')
print('f(c) = ' + f_s + '\n')
f_s = '-42 + 686.7*(1 - np.exp(-6*c/35))/c'
```

Tente fazer as correções necessárias no código. **Sugestão:** verifique a função `sympy.core.evalf` do módulo `sympy`)

- Adicione mecanismos de plotagem no código Python
- Crie um código em Javascript para adicionarmos na página do projeto Numbiosis com o máximo possível de GUI (labels + input data).
- Teste a implementação com um problema realista.

Problema sugerido: Uma reação química reversível

$$2A + B \iff C$$

pode ser caracterizada pela relação de equilíbrio

$$K = \frac{c_c}{c_a^2 c_b},$$

onde a nomenclatura c_i representa a concentração do constituinte i. Suponha que definamos uma variável x como o número de moles de C que são produzidos. A conservação da massa pode ser usada para reformular a relação de equilíbrio

como

$$K = \frac{(c_{c,0} + x)}{(c_{a,0} - 2x)^2(c_{b,0} - x),}$$

onde o subscrito 0 designa a concentração inicial de cada constituinte. Se $K = 0,016$, $c_{a,0} = 42$, $c_{b,0} = 28$ e $c_{c,0} = 4$, determine o valor de x.

(a) Obtenha a solução graficamente.

(b) Com base em (a), resolva a raiz com suposições iniciais de $x_l = 0$ e $x_u = 20$, com critério de erro de $\epsilon_s = 0,5\%$. (Vide clipping *Definições de erro* para entender ϵ_s.)

© Use o método da bisseção.

12.2 Tarefa: Falsa Posição

Programe uma nova função para executar o método da falsa posição ou estenda o código anterior para uma nova função que contemple os dois casos (sugestão: use `switch... case...`).

CHAPTER

THIRTEEN

MÉTODO DA ITERAÇÃO LINEAR (PONTO FIXO)

Este notebook explora aspectos do método da *iteração linear*, ou também chamado de método do *ponto fixo*.

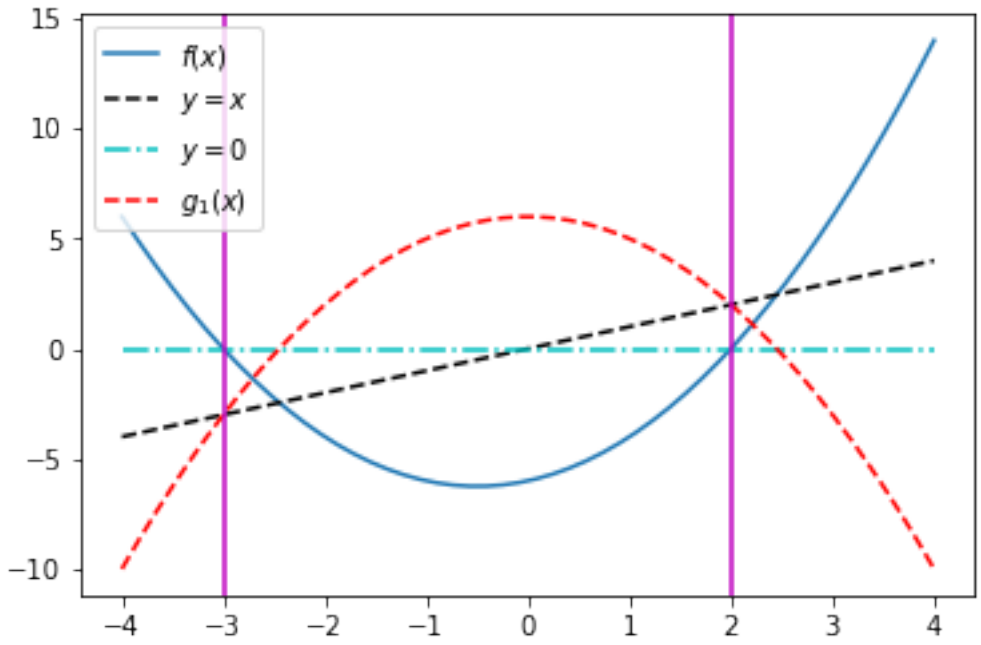
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

13.1 Exemplo

Estudamos a função $f(x) = x^2 + x - 6$.

```
x = np.linspace(-4,4,50)
f = lambda x: x**2 + x - 6

xr = np.roots([1,1,-6])
print('Raízes: x1 = {:f}, x2 = {:f}'.format(xr[0], xr[1]))

# função de iteração
g1 = lambda x: 6 - x**2

plt.plot(x,f(x),label='$f(x)$');
plt.plot(x,x,'k--',label='$y=x$');
plt.plot(x,0*x,'c-.',label='$y=0$');
plt.plot(x,g1(x),'r--',label='$g_1(x)$');

plt.axvline(-3,-5,10,color='m');
plt.axvline(2,-5,10,color='m');
plt.legend(loc='best');
```

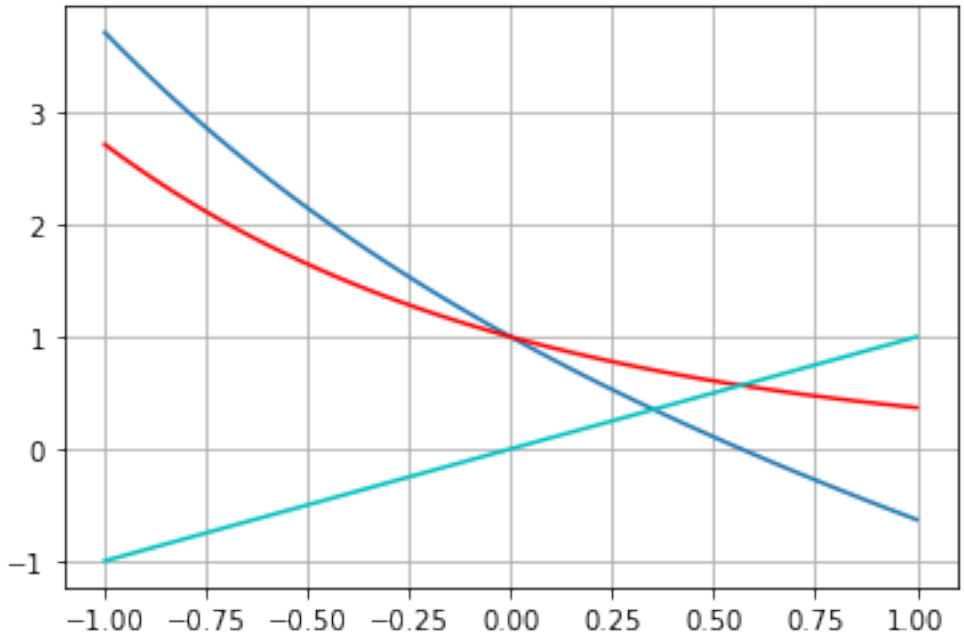
```
Raízes: x1 = -3.000000, x2 = 2.000000
```



13.2 Exemplo

Estudamos a função $f(x) = \exp(x) - x$

```
x2 = np.linspace(-1,1,50)

f2 = lambda x: np.exp(-x) - x
g2 = lambda x:  np.exp(-x)

plt.plot(x2,f2(x2),x2,g2(x2),'r',x2,x2,'c')
plt.grid(True)
```



13.3 Implementação do método do ponto fixo

```
def ponto_fixo(x0,f,g,tol,N,vis):
    """
    Resolve problema de determinacao de raizes pelo
    metodo do ponto fixo (iteracao linear).

    entrada:

       x0  - aproximacao inicial         (float)
        f  - funcao a ser resolvida      (str)
        g  - funcao de iteracao          (str)
      tol  - tolerancia                  (float)
        N  - numero maximo de iteracoes  (int)
      vis  - flag para plotagem          (bool)

    saida:

       x   - raiz aproximada para f      (float)
    """
    from numpy import linspace
    from matplotlib.pyplot import plot,legend

    # funcoes
    f = eval('lambda x:' + f)
    g = eval('lambda x:' + g)

    # inicializacao
    it = 0 # contador
    x, xn = x0, x0 + 1 # iteradas atual, anterior

    e = abs(x-xn)/abs(x) # erro
```

(continues on next page)

(continued from previous page)

```
    # tabela
    print('i\t x\t\t f(x)\t\t ER')
    print('{0:d}\t {1:f}\t {2:f}\t {3:e}'.format(it,x,f(x),e))

    # laco
    while e >= tol and it <= N:
        it += 1
        xn = x
        x = g(xn)
        e = abs(x-xn)/abs(x)
        print('{0:d}\t {1:f}\t {2:f}\t {3:e}'.format(it,x,f(x),e))

        if it > N:
            print('Solução nao alcancada com N iteracoes.')
            break

    if vis == True:
        dx = 2*x
        dom = linspace(x - dx,x + dx,30)
        plot(dom,f(dom),label='$f(x)$')
        plot(dom,dom*0,label='$y=0$')
        plot(dom,g(dom),label='$g(x)$')
        plot(dom,dom,label='$y=x$')
        legend()

    return x
```

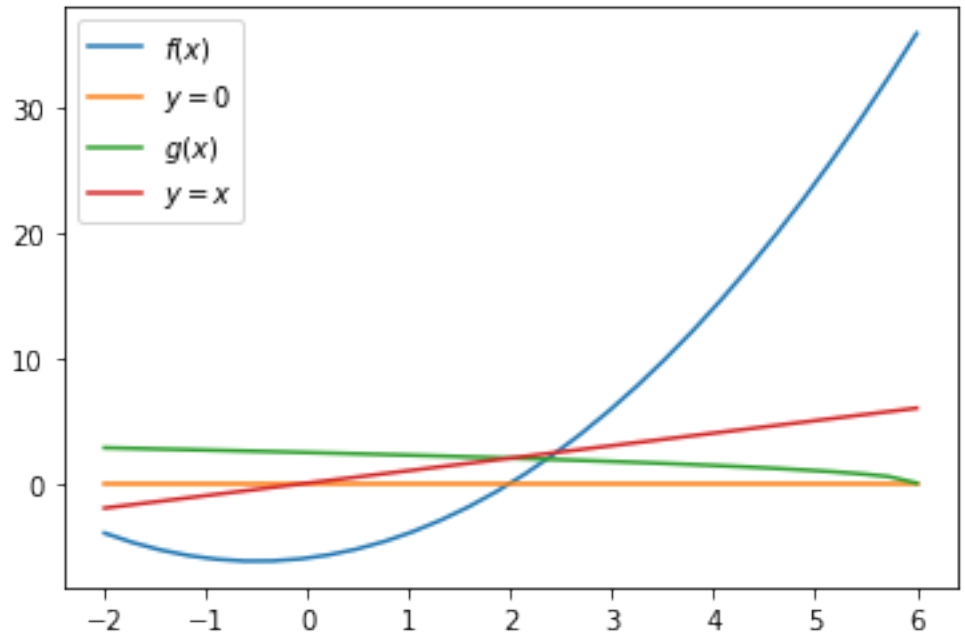
13.4 Estudo de caso: $f(x) = x^2 + x - 6$

Função de iteração: $g(x) = \sqrt{6 - x}$

```
f = 'x**2 + x - 6'
g = '(6 - x)**(1/2)'

x0 = 0.1
tol = 1e-5
N = 100

ponto_fixo(x0,f,g,tol,N,True)
```

```
i          x                 f(x)                ER
0          0.100000          -5.890000           1.000000e+01
1          2.428992          2.328992           9.588307e-01
2          1.889711          -0.539280           2.853771e-01
3          2.027385          0.137674           6.790695e-02
4          1.993142          -0.034243           1.718024e-02
5          2.001714          0.008572           4.282174e-03
6          1.999572          -0.002142           1.071346e-03
7          2.000107          0.000536           2.677864e-04
8          1.999973          -0.000134           6.694973e-05
9          2.000007          0.000033           1.673724e-05
10          1.999998          -0.000008           4.184321e-06
```

```
1.9999983262723453
```

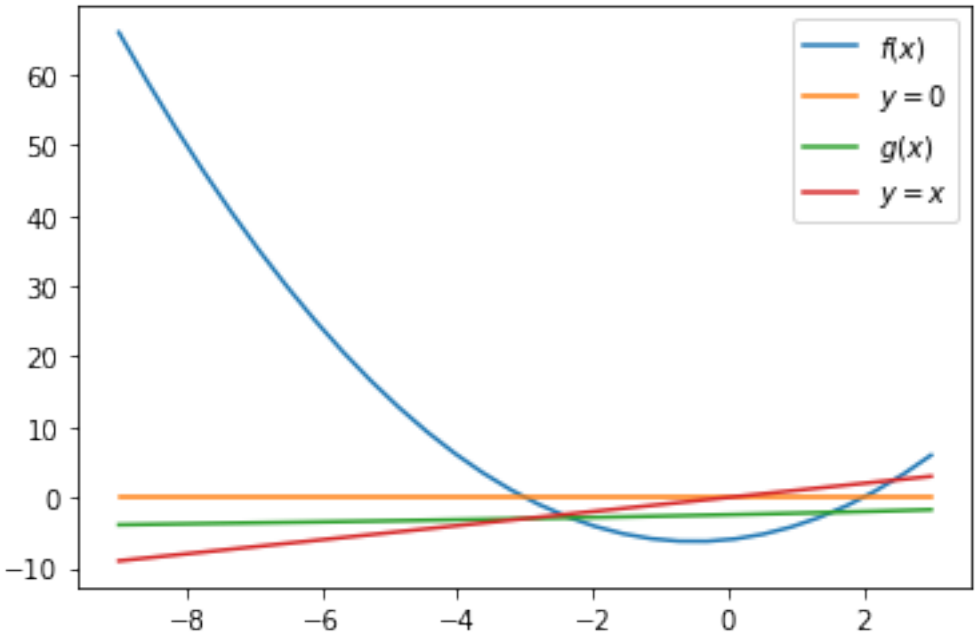


Função de iteração: $g(x) = -\sqrt{6 - x}$

```
f = 'x**2 + x - 6'
g = '-(6 - x)**(1/2)'

x0 = 0.1
tol = 1e-5
N = 100

ponto_fixo(x0,f,g,tol,N,True)
```

```
i          x                  f(x)                  ER
0          0.100000          -5.890000          1.000000e+01
1          -2.428992          -2.528992          1.041169e+00
2          -2.903273          -0.474281          1.633608e-01
3          -2.983835          -0.080563          2.699970e-02
4          -2.997305          -0.013469          4.493853e-03
5          -2.999551          -0.002246          7.488072e-04
6          -2.999925          -0.000374          1.247965e-04
7          -2.999988          -0.000062          2.079929e-05
8          -2.999998          -0.000010          3.466545e-06
```

```
-2.9999979200736955
```


f(x)
y = 0
g(x)
y = x
60
50
40
30
20
10
0
−10
−8
−6
−4
−2
0
2

CHAPTER

FOURTEEN

IMPLEMENTAÇÃO DO MÉTODO DE NEWTON

```
%matplotlib inline
```

- Analisar a dependência da estimativa inicial.
- Executar o código duas vezes: para $x_0 = 0.0$ e $x_0 = 1.0$ em $f(x) = -0.9x^2 + 1.7x + 2.5$.

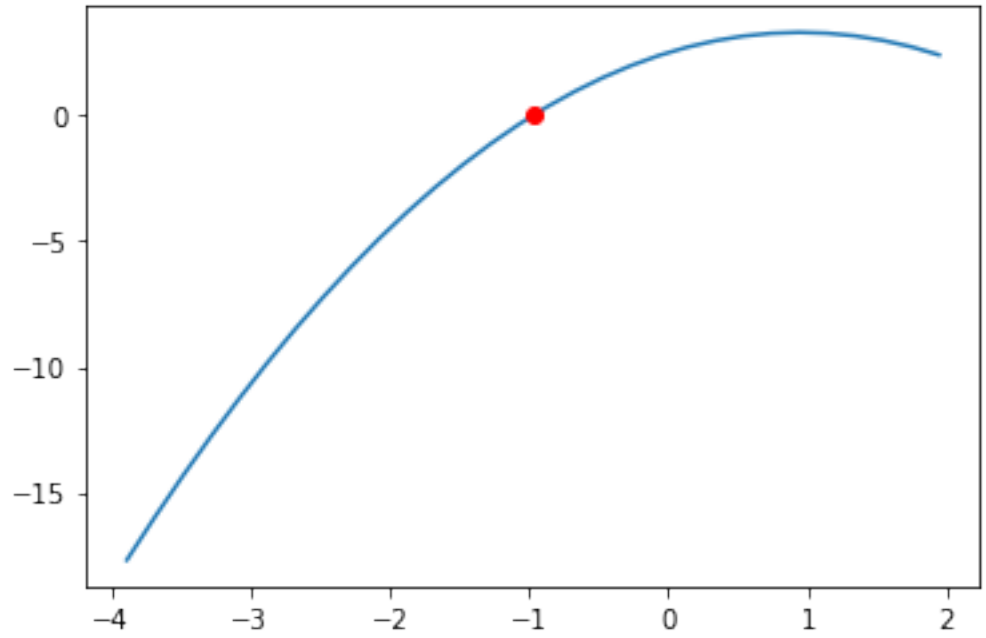
```
# Método de Newton

from numpy import linspace
from matplotlib.pyplot import plot

def newton(x0,f,df,tol,nmax,var,plotar):

    f = eval('lambda x:' + f)
    df = eval('lambda x:' + df)

    it = 0 # contador de iteracoes

    # imprime estimativa inicial
    print('Estimativa inicial: x0 = {0}\n'.format(x0))

    # Loop
    for i in range(0,nmax):

        x = x0 - f(x0)/df(x0) # funcao de iteracao

        e = abs(x-x0)/abs(x) # erro

        # tabela
        print('{0:d}  {1:f}  {2:f}  {3:f}  {4:e}'.format(i,x,f(x),df(x),e))

        if e < tol:
            break
        x0 = x

    if i == nmax:
        print('Solução não obtida em {0:d} iterações'.format(nmax))
    else:
        print('Solução obtida: x = {0:.10f}'.format(x))

    # plotagem
    if plotar:
        delta = 3*x
        dom = linspace(x-delta,x+delta,30)
```

(continues on next page)

(continued from previous page)

```
        plot(dom,f(dom),x,f(x),'ro')

    return x


# parametros
x0 = 0. # estimativa inicial
tol = 1e-3 # tolerancia
nmax = 100 # numero maximo de iteracoes
f = '-0.9*x**2 + 1.7*x + 2.5'    # funcao
df = '-1.8*x + 1.7'    # derivada dafuncao
var = 'x'
plotar = True

# chamada da função
xm = newton(x0,f,df,tol,nmax,var,plotar)
```

```
Estimativa inicial: x0 = 0.0

0  -1.470588  -1.946367  4.347059  1.000000e+00
1  -1.022845  -0.180427  3.541121  4.377432e-01
2  -0.971893  -0.002336  3.449407  5.242539e-02
3  -0.971216  -0.000000  3.448188  6.974331e-04
Solução obtida: x = -0.9712156364
```



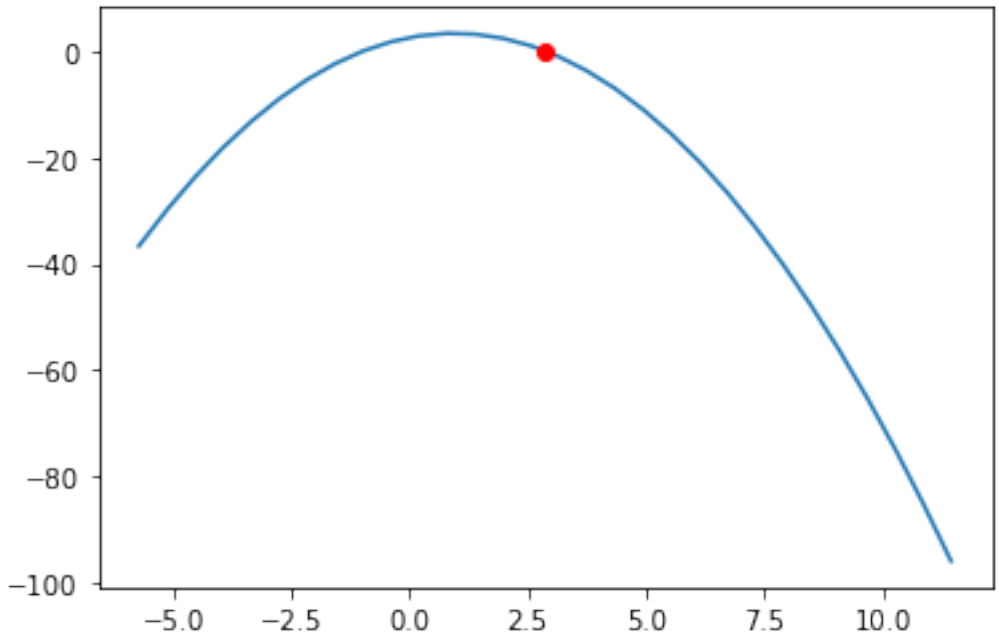

```
# chamada da função
xm = newton(1.0,f,df,tol,nmax,var,plotar)
```

```
Estimativa inicial: x0 = 1.0

0  34.000000  -980.100000  -59.500000  9.705882e-01
1  17.527731  -244.202079  -29.849916  9.397833e-01
```

(continues on next page)

(continued from previous page)

```
2  9.346734  -60.235844  -15.124121  8.752787e-01
3  5.363967  -14.276187  -7.955141  7.425039e-01
4  3.569381  -2.898486  -4.724886  5.027724e-01
5  2.955930  -0.338690  -3.620674  2.075323e-01
6  2.862387  -0.007875  -3.452297  3.268017e-02
7  2.860106  -0.000005  -3.448190  7.975862e-04
Solução obtida: x = 2.8601057637
```



14.1 Desafio

1. Generalize o código acima para que a expressão da derivada seja calculada diretamenta e não manualmente. (dica: use computação simbólica)
2. Resolva o problema aplicado abaixo com este método ou desenvolva o seu para resolver e compare com a função residente do `scipy`.

14.1.1 Problema aplicado

Um jogador de futebol americano está prestes a fazer um lançamento para outro jogador de seu time. O lançador tem uma altura de 1,82 m e o outro jogador está afastado de 18,2 m. A expressão que descreve o movimento da bola é a familiar equação da física que descreve o movimento de um projétil:

$$y = x\tan(\theta) - \frac{1}{2}\frac{x^2 g}{v_0^2}\frac{1}{\cos^2(\theta)} + h,$$

onde x e y são as distâncias horizontal e verical, respectivamente, $g = 9,8\,m/s^2$ é a aceleração da gravidade, v_0 é a velocidade inicial da bola quando deixa a mão do lançador e θ é o Ângulo que a bola faz com o eixo horizontal nesse mesmo instante. Para $v_0 = 15,2\,m/s$, $x = 18,2\,m$, $h = 1,82\,m$ e $y = 2,1\,m$, determine o ângulo θ no qual o jogador deve lançar a bola.

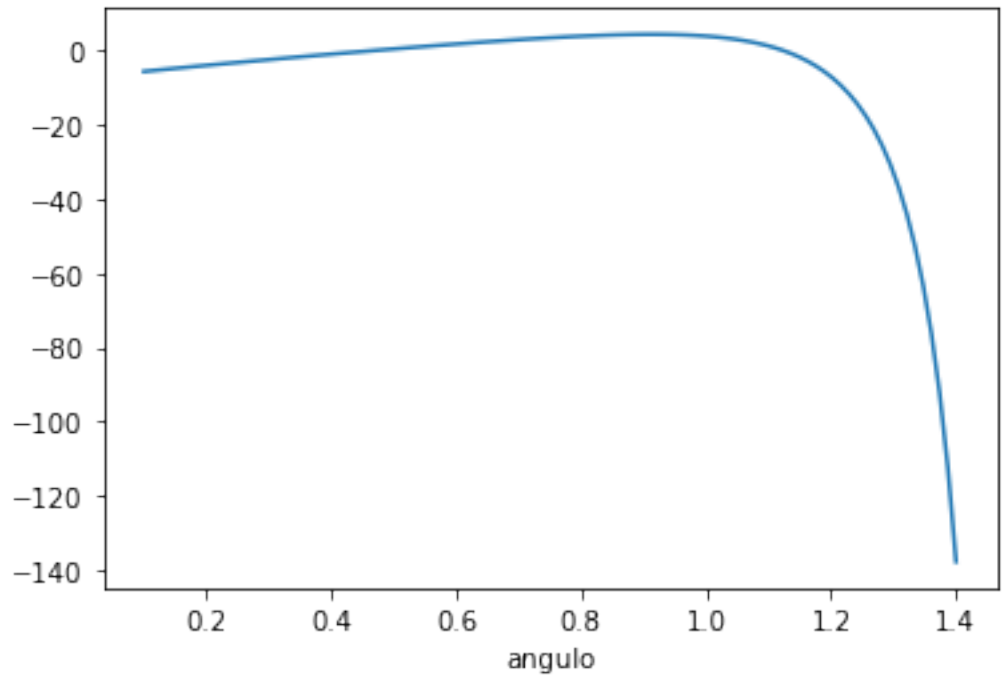
14.1.2 Solução por função residente

- Importar módulos
- Definir função $f(\theta)$

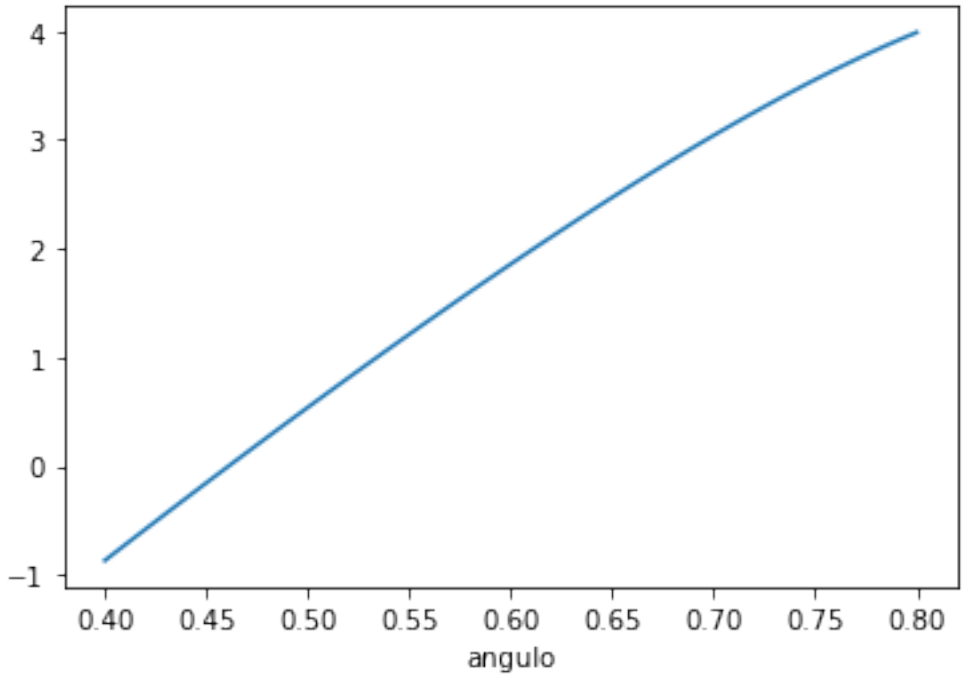
```
from scipy.optimize import newton
import numpy as np
import matplotlib.pyplot as plt

v0 = 15.2
x = 18.2
h = 1.82
y = 2.1
g = 9.8

# f(theta) = 0
f = lambda theta: x*np.tan(theta) - 0.5*(x**2*g/v0**2)*(1/(np.cos(theta)**2)) + h - y
```

14.2 Localização

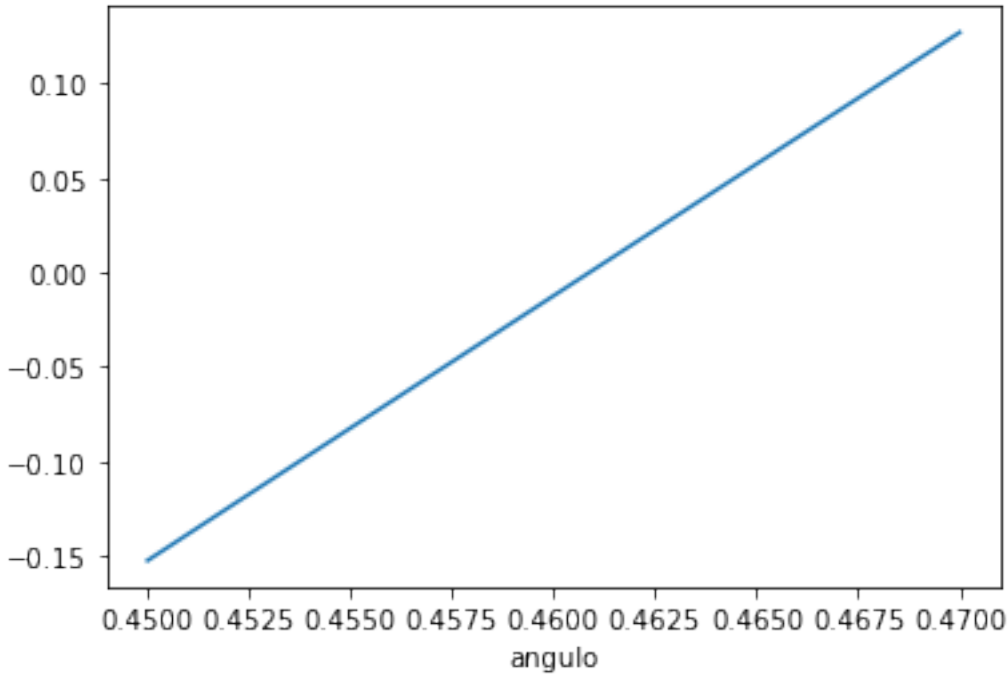
```
th = np.linspace(0.1,1.4,100,True)
plt.plot(th,f(th))
plt.xlabel('angulo');
```




```
th = np.linspace(0.4,0.8,100,True)
plt.plot(th,f(th))
plt.xlabel('angulo');
```




```
th = np.linspace(0.45,0.47,100,True)
plt.plot(th,f(th))
plt.xlabel('angulo');
```



14.3 Refinamento

Para a função residente, nem é preciso fazer um processo de localização prolongado de modo a entrar com uma estimativa inicial muito próxima. Plotar a função até um intervalo razoável já é suficiente para ter uma noção sobre onde a raiz está.

Quanto à escolha da estimativa inicial, ainda que seja "mal feita", o método poderá encontrar a raiz de modo rápido, pois sua programação é robusta.

Vejamos então, qual é a raiz com uma estimativa inicial de 0.47.

```
ang = newton(f,0.47)
ang
```

```
0.4608834641642987
```

```
np.rad2deg(ang)
```

```
26.406677343983237
```

CHAPTER

FIFTEEN

IMPLEMENTAÇÃO DO MÉTODO DA SECANTE

```
%matplotlib inline
```

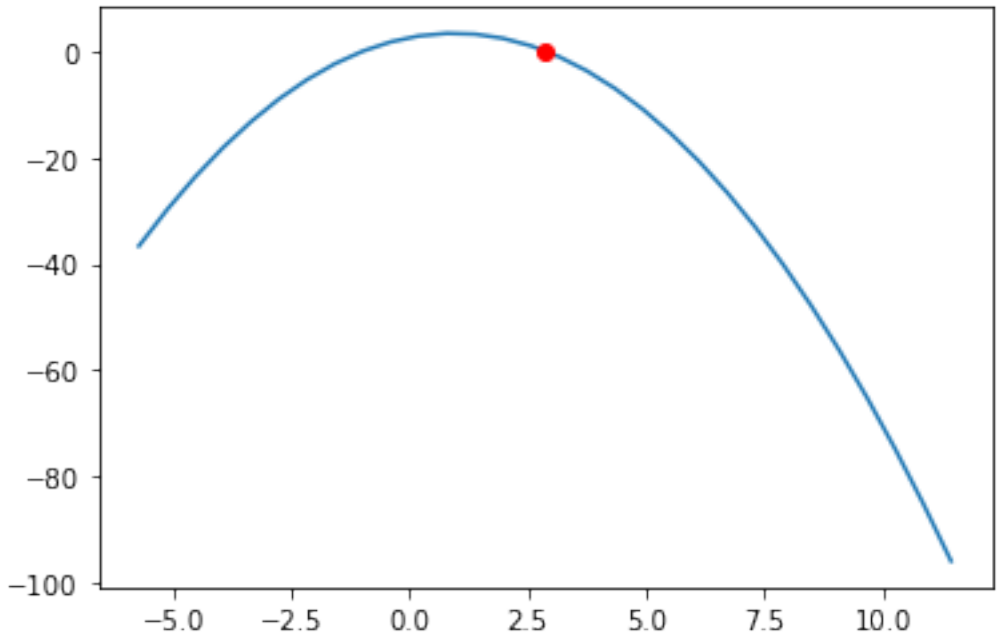
```
# Método da Secante

from numpy import linspace
from matplotlib.pyplot import plot

def secante(xa,xb,f,tol,nmax,var,plotar):

    f = eval('lambda x:' + f)

    # imprime estimativas iniciais
    print('Estimativas iniciais: xa = {0}; xb = {1} \n'.format(xa,xb))

    # Loop
    for i in range(0,nmax):

        x = (xa*f(xb) - xb*f(xa))/(f(xb) - f(xa))

        e = abs(x-xb)/abs(x) # erro

        # tabela
        print('{0:d}  {1:f}  {2:f}  {3:e}'.format(i,x,f(x),e))

        if e < tol:
            break
        xa = xb
        xb = x

    if i == nmax:
        print('Solução não obtida em {0:d} iterações'.format(nmax))
    else:
        print('Solução obtida: x = {0:.10f}'.format(x))

    # plotagem
    if plotar:
        delta = 3*x
        dom = linspace(x-delta,x+delta,30)
        plot(dom,f(dom),x,f(x),'ro')

    return x
```

(continues on next page)

(continued from previous page)

```
# parametros
xa = 1.0 # estimativa inicial 1
xb = 2.0 # estimativa inicial 2
tol = 1e-3 # tolerancia
nmax = 100 # numero maximo de iteracoes
f = '-0.9*x**2 + 1.7*x + 2.5'    # funcao
var = 'x'
plotar = True

# chamada da função
xm = secante(xa,xb,f,tol,nmax,var,plotar)
```

```
Estimativas iniciais: xa = 1.0; xb = 2.0

0  4.300000  -6.831000  5.348837e-01
1  2.579345  0.897168  6.670898e-01
2  2.779097  0.273423  7.187654e-02
3  2.866660  -0.022642  3.054518e-02
4  2.859963  0.000487  2.341478e-03
5  2.860104  0.000001  4.933559e-05
Solução obtida: x = 2.8601041641
```



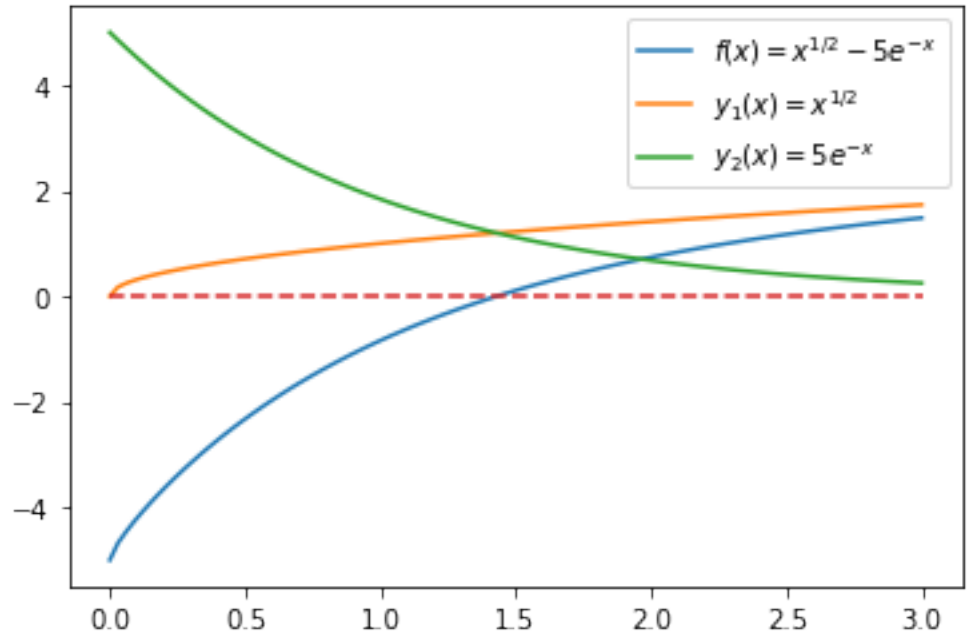
15.1 Problema

Determinar a raiz positiva da equação: $f(x) = \sqrt{x} - 5e^{-x}$, pelo método das secantes com erro inferior a 10^{-2}.

15.1.1 Resolução

Para obtermos os valores iniciais x_0 e x_1 necessários para iniciar o processo iterativo, dividimos a equação original $f(x) = 0$ em outras duas y_1 e y_2, com $y_1 = \sqrt{x}$ e $y_2(x) = e^{-x}$, que colocadas no mesmo gráfico, produzem uma interseção próximo a $x = 1.5$. Assim, podemos escolher duas estimativas iniciais próximas deste valor. Podemos escolher $x_0 = 1.4$ e $x_1 = 1.5$.

```
from numpy import sqrt, exp
from matplotlib.pyplot import plot,legend

fx = lambda x: sqrt(x) - 5*exp(-x)

x = linspace(0,3,100)
plot(x,fx(x),label='$f(x) = x^{1/2} - 5e^{-x}$');
plot(x,sqrt(x),label='$y_1(x) = x^{1/2}$');
plot(x,5*exp(-x),label='$y_2(x) = 5e^{-x}$');
plot(x,fx(x)*0,'--');
legend();
```



Vejamos o valor de $f(x = 1.5)$.

```
fx(1.5)
```

```
0.10909407064943988
```

Vamos montar uma função anônima para computar o valor da interseção da secante com o eixo x, a saber:

```
xm = lambda a,b: ( a*fx(b) - b*fx(a) ) / (fx(b) - fx(a) )
```

Vamos usar os nosso valores estimados:

```
x0 = 1.4
x1 = 1.5
x2 = round(xm(x0,x1),3)
print(x2)
```

```
1.431
```

Agora, usamos este novo valor e o anterior.

```
x3 = round(xm(x1,x2),3)
print(x3)
```

```
1.43
```

Calculemos o erro relativo entre as estimativas x_1 e x_2:

```
err = lambda a,b: abs(a - b)/abs(a)
e1 = err(x2,x1)
print(round(e1,3))
print("{0:e}".format(e1))
```

```
0.048
4.821803e-02
```

Agora, calculemos o erro relativo entre as estimativas x_2 e x_3:

```
e2 = err(x3,x2)
print(round(e2,3))
print("{0:e}".format(e2))
```

```
0.001
6.993007e-04
```

O erro está diminuindo. Além disso, o valor da raiz está se estabilizando em torno de 1.430. Isto significa que as estimativas iniciais foram muito boas. Com efeito, o uso das interseções proporcionou uma boa escolha.

CHAPTER

SIXTEEN

RAÍZES DE POLINÔMIOS COM O MÉTODO DE MÜLLER

```
%matplotlib inline
```

16.1 Computação com polinômios

Como exemplo, vamos implementar a forma aninhada de um polinômio de grau 3 (também conhecida como **forma de Hörner**)

```
import sympy as sy
import numpy as np
import matplotlib.pyplot as plt

sy.init_printing()

# escreve a forma aninhada de um polinomio de grau n

# grau do polinomio
n = 3

# variavel independente
x = sy.Symbol('x');

# coeficientes do polinomio
a = [ sy.Symbol('a'+ str(i)) \
     for i in range(0,n+1) ]

# forma aninhada simbolica
p, dp = 0, 0
for j in range(n,-1,-1):
    dp = dp*x + p
    p  = a[j] + p*x

# determinacao de derivada de modo simbolico
dp2 = sy.diff(p,x)
```

Imprimindo o polinômio simbólico

```
p
```

$$a_0 + x\left(a_1 + x\left(a_2 + a_3 x\right)\right)$$

Imprimindo a derivada simbólica do polinômio implementada pelo usuário

```
dp
```

$$a_1 + x\left(a_2 + a_3 x\right) + x\left(a_2 + 2a_3 x\right)$$

Imprimindo a derivada simbólica do polinômio pela função residente `diff`

```
dp2
```

$$a_1 + x\left(a_2 + a_3 x\right) + x\left(a_2 + 2a_3 x\right)$$

Verificando igualdade

```
dp == dp2
```

```
True
```

16.2 Calculando raízes de polinômios

```
# define valores dos coeficientes aj para o polinômio
# na ordem a0 + a1x + a2x**2 + ...
v = [-1, 2.2,3.5,4]

# escreve o polinômio
pn = p.subs(dict(zip(a,v)))
print(pn)
```

```
x*(x*(4*x + 3.5) + 2.2) - 1
```

Calcula todas as raízes do polinômio `pn`

```
rc = sy.roots(pn,x,multiple=True)
rc
```

$$[0.28424794239786,\ -0.57962397119893 - 0.737258363375504i,\ -0.57962397119893 + 0.737258363375504i]$$

Calcula apenas as raízes reais de `pn`

```
rr = sy.roots(pn,x,multiple=True,filter='R')
rr
```

$$[0.28424794239786]$$

16.3 Avaliando polinômios

Podemos avaliar polinômios usando a função `polyval` do *Numpy*. Entretanto, como ela recebe coeficientes do maior para o menor grau, para mantermos a consistência com nosso polinômio anterior, devemos converter a lista `v` para um objeto `array` e fazer uma inversão (`flip`).

```
vi = np.flip(np.asarray(v),axis=0)
vi
```

```
array([ 4. ,  3.5,  2.2, -1. ])
```

Agora, vamos avaliar o polinômio em $x = \pi$

```
xi = np.pi
np.polyval(vi,xi)
```

$$164.48022596290957$$

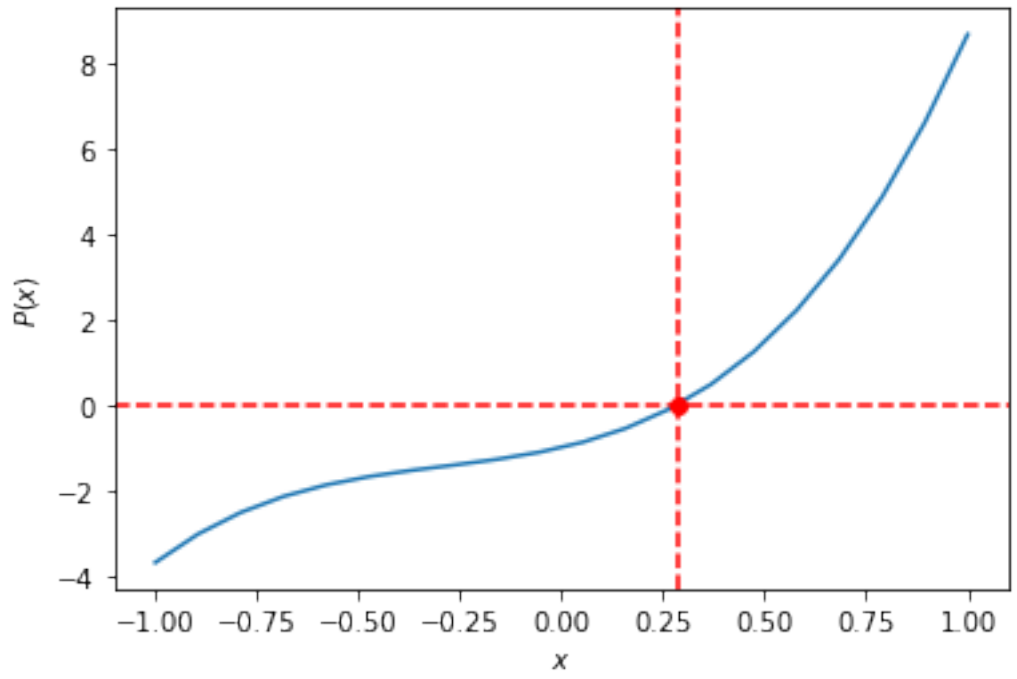
Note que se avaliássemos o polinômio em um ponto arbitrário, a forma impressa é idêntica àquela que obtivemos anteriormente.

```
np.polyval(vi,x)
```

$$x\left(x\left(4.0x + 3.5\right) + 2.2\right) - 1.0$$

Agora vamos plotar o polinômio. Antes, vamos converter nosso polinômio para uma função a fim de a avaliarmos em um intervalo. Vamos escolher o intervalo $-1 \leq x \leq 1$

```
# converte para função
f = sy.lambdify(x,pn)

# intervalo
xf = np.linspace(-1,1,num=20,endpoint=True)

# plotagem do polinômio com destaque para a raiz real
plt.axhline(y=f(rr[0]),c='r',ls='--')
plt.axvline(x=rr[0],c='r',ls='--')
plt.plot(xf,f(xf))
plt.plot(rr[0],f(rr[0]),'ro')
plt.xlabel('$x$')
plt.ylabel('$P(x)$');
```



16.4 Implementação: Método de Müller

```
# \TODO caso complexo (verificar aritmética)
def metodo_muller(f,x0,dx,EPS,N):
    """
        Busca aproximação para raiz da função f
        pelo método de Muller.

        ENTRADA:
             f: função; ex. f = lambda x: x^3 + 2*x
            x0: estimativa inicial
             h: incremento (produz valores vizinhos)
           EPS: erro
             N: iterações
        SAÍDA:
             x: aproximação de raiz de f
    """

    if N < 3:
        raise("N deve ser maior do que 3")

    # escolhendo os dois pontos adicionais
    # na vizinhança de x0 para ter as 3
    # estimativas iniciais
    x1 = x0 - dx
    x2 = x0 + dx

    h0 = x1 - x0
    h1 = x2 - x1
    d0 = (f(x1) - f(x0))/h0
    d1 = (f(x2) - f(x1))/h1
    d = (d1 - d0)/(h1 + h0)
    i = 3
    while i <= N:

        b = d1 + h1*d

        # discriminante
        D = (b**2 - 4*f(x2)*d)**0.5

        # Verificando o denominador:
        # Esta condição irá definir o maior denominador
        # haja vista que b + sgn(b)D.
        # (critério de sgn(b))
        if abs(b - D) < abs(b + D):
            E = b + D
        else:
            E = b - D

        h = -2*f(x2)/E
        x = x2 + h
        if abs(h) < EPS:
            return x

        # atualização
        x0 = x1
        x1 = x2
```

(continues on next page)

(continued from previous page)

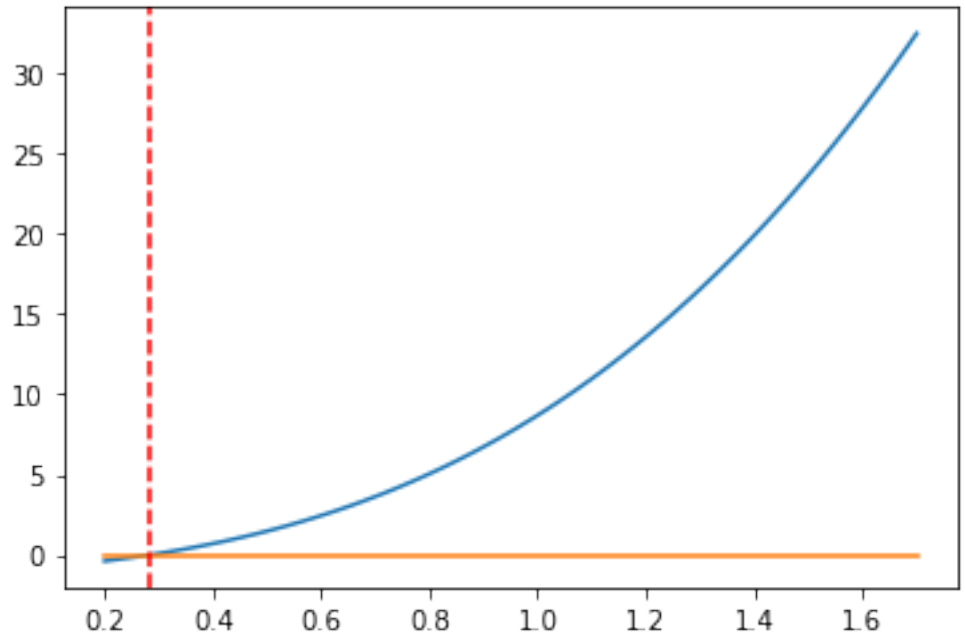
```
        x2 = x
        h0 = x1 - x0
        h1 = x2 - x1
        d0 = (f(x1) - f(x0))/h0
        d1 = (f(x2) - f(x1))/h1
        d = (d1 - d0)/(h1 + h0)

        i += 1
```

16.5 Exemplos

Exemplo. Determinando raízes para o polinômio $P(x) = 4x^3 + 3.5x^2 + 2.2x - 1$ com estimativas iniciais $x_0 = 0.5$ $x_1 = 1.0$ e $x_2 = 1.5$, $\epsilon = 10^{-5}$ e $N = 100$. Notemos que o segundo argumento da função desempenha o papel de x_1 e o terceiro argumento opera como um "raio" de comprimento $dx = 0.5$ que fará com que $x_0 = x_1 - dx$ e $x_2 = x_1 + dx$. Isto decorre de como o a função foi programada. Veja o código anterior.

```
f = lambda x: 4*x**3 + 3.5*x**2 + 2.2*x - 1

x0 = metodo_muller(f,1.0,0.5,1e-5,100)
```

```
X = np.linspace(0.2,1.7,100)
plt.plot(X,f(X))
plt.plot(X,0*f(X))
plt.axvline(x=x0.real,c='r',ls='--');
```



Exemplo. Determinando raízes para o polinômio $P(x) = x^4 - 3x^3 + x^2 + x + 1$ com estimativas iniciais $x_0 = -0.5$ $x_1 = 0.0$ e $x_2 = 0.5$, $\epsilon = 10^{-5}$ e $N = 100$.

```
f2 = lambda x: x**4 - 3*x**3 + x**2 + x + 1

metodo_muller(f2,-0.5,0.5,1e-5,100)
```

```
(-0.3390928377617365-0.4466300999972928j)
```

Com essas estimativas a raiz é um número complexo. Escolhamos agora estimativas diferentes:

- Caso 1: $x_0 = 0.5$ $x_1 = 1.0$ e $x_2 = 1.5$
- Caso 2: $x_0 = 1.5$ $x_1 = 2.0$ e $x_2 = 2.5$

```
# caso 1
c1 = metodo_muller(f2,1.5,0.5,1e-5,100)
print(c1)
```

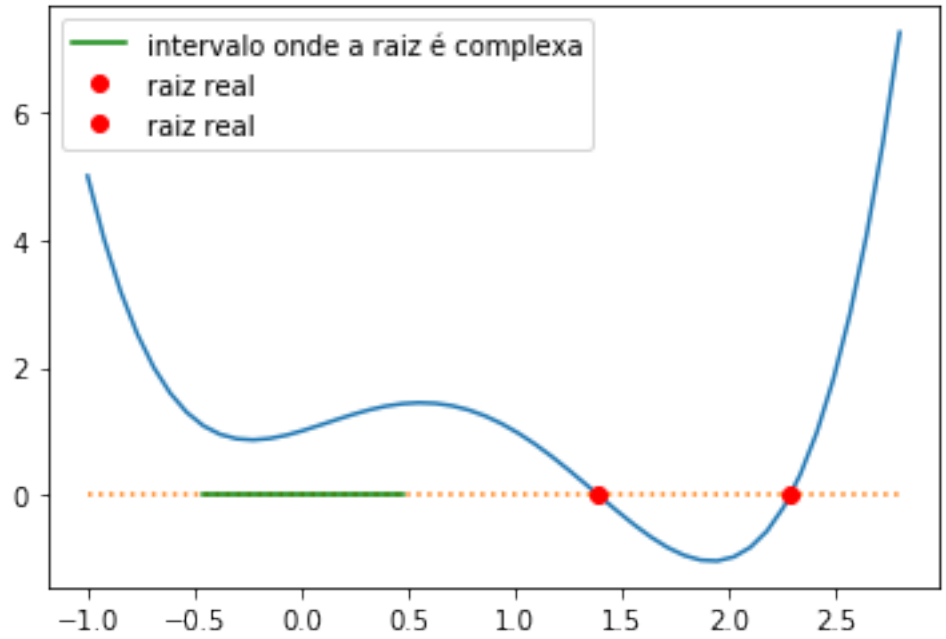
```
1.3893906833348133
```

```
# caso 2
c2 = metodo_muller(f2,2.0,0.5,1e-5,100)
print(c2)
```

```
2.2887949921884836
```

Por que há resultados diferentes? Vamos verificar o gráfico deste polinômio no domínio $[-1, 2.8]$.

```
from matplotlib.pyplot import plot,legend
from numpy import linspace, where, logical_and

x = linspace(-1,2.8,50)
plot(x,f2(x))
plot(x,0*f2(x),':')
xi = where( logical_and(x >= -0.5,x <= 0.5) )
xi = x[xi]
plot(xi,0*xi,'-g',label='intervalo onde a raiz é complexa')
plot(c1,0,'or',c2,0,'or',label='raiz real')
legend();
```



Na primeira escolha de estimativas iniciais, obtivemos uma raiz complexa porque no intervalo $[-0.5, 0.5]$, o polinômio não intersecta o eixo x. Nos outros dois casos, temos as duas raízes reais do polinômio.

CHAPTER

SEVENTEEN

ÁLGEBRA LINEAR COM PYTHON: ELIMINAÇÃO GAUSSIANA E CONDICIONAMENTO

```
%matplotlib inline
```

17.1 Solução de sistemas lineares

Métodos adequados para a resolução de sistemas lineares e realizar operações no escopo da Álgebra Linear são encontrados no submódulo `linalg` do `scipy`. Importamos essas funcionalidades com:

```
from scipy import linalg
```

Vamos calcular a solução do sistema linear $\mathbf{Ax} = \mathbf{b}$ com

$$\mathbf{A} = \begin{bmatrix} 4 & -2 & -3 & 6 \\ -6 & 7 & 6.5 & -6 \\ 1 & 7.5 & 6.25 & 5.5 \\ -12 & 22 & 15.5 & -1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 12 \\ -6.5 \\ 16 \\ 17 \end{bmatrix}$$

Vamos importar os módulos e escrever a matriz **A**.

```
import numpy as np
from scipy import linalg

A = np.array([[4,-2,-3,6],[-6,7,6.5,-6],[1,7.5,6.25,5.5],[-12,22,15.5,-1]])
print(A)
```

```
[[  4.    -2.    -3.     6.  ]
 [ -6.     7.     6.5   -6.  ]
 [  1.     7.5    6.25   5.5 ]
 [-12.    22.    15.5   -1.  ]]
```

Agora, vamos escrever o vetor **b**.

```
b = np.array([12,-6.5,16,17])
print(b)
```

```
[12.  -6.5 16.  17. ]
```

Podemos checar as dimensões com

```
# dimensões de A
A.shape
```

```
(4, 4)
```

```
# dimensão de b
b.shape
```

```
(4,)
```

A solução do sistema pode ser obtida através do método `linalg.solve`.

```
x = linalg.solve(A,b)
print(x)
```

```
[ 2.   4.  -3.   0.5]
```

17.2 Inversão de matrizes

Matematicamente, a solução do sistema anterior é dada por $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Podemos até invocar a matriz inversa aqui com `linalg.inv(A).dot(b)` e a solução é a mesma que no caso anterior.

```
x2 = linalg.inv(A).dot(b)
print(x2)
```

```
[ 2.   4.  -3.   0.5]
```

Por outro lado, este método é ineficiente. Computacionalmente, a inversão de matrizes **não** é aconselhável.

17.3 Verificação da solução

Podemos usar o fato de que $\mathbf{A}\mathbf{A}^{-1}\mathbf{b} - \mathbf{b} = \mathbf{0}$.

```
x3 = A.dot(linalg.inv(A).dot(b)) - b
print(x3)
```

```
[ 5.32907052e-15 -3.19744231e-14 -4.44089210e-14 -1.20792265e-13]
```

Note que o vetor é próximo do vetor nulo, mas não identicamente nulo.

Podemos também computar a **norma do resíduo (erro)**: $||\mathbf{r}|| = ||\mathbf{b} - \mathbf{A}\mathbf{x}|| = \langle \mathbf{b} - \mathbf{A}\mathbf{x}, \mathbf{b} - \mathbf{A}\mathbf{x} \rangle^{1/2}$

```
r = b - A.dot(x)
np.sqrt(r.dot(r))
```

```
1.464821375527116e-14
```

Como a norma do resíduo é próxima de zero, a solução do sistema linear é assumida como correta.

CHAPTER

EIGHTEEN

ELIMINAÇÃO GAUSSIANA

A Eliminação Gaussiana (EG) é um algoritmo utilizado para resolver sistemas de equações lineares ao reduzir a matriz plena associada do sistema a uma matriz triangular. Este processo também é chamado de *escalonamento*. Abaixo, usaremos uma matriz genérica 3x3 para exemplificação.

18.1 Passos

- Escrever o sistema linear na forma de *matriz estendida* usando os coeficientes das variáveis como elementos da matriz e o vetor independente como sendo a última coluna;

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \quad (L_1) \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \quad (L_2) \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \quad (L_3) \end{array}$$

$$\downarrow$$

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

- Realizar operações elementares de combinação linear e permutação entre linhas;
  - Multiplicação por escalar:$$ \begin{array}{c} L_2 \leftarrow L_2 .w\ \Rightarrow\ \end{array} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ w.a_{21} & w.a_{22} \end{bmatrix} \end{array} $$

```
- Combinação linear:
```

$$L_2 \leftarrow L_2 - L_1.w \ \Rightarrow\ \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} \\ a_{21} - a_{11}.w & a_{22} - a_{12}.w \end{bmatrix}$$

```
- Permutação:
```

$$L_2 \leftarrow L_1 \, e \, L_1 \leftarrow L_2 \ \Rightarrow\ \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \rightarrow \begin{bmatrix} a_{21} & a_{22} \\ a_{11} & a_{12} \end{bmatrix}$$

- Anular todos os elementos na porção triangular inferior da matriz original, isto é, todas as entradas exatamente abaixo das entradas dispostas na diagonal principal;

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} a_{11}^{(k)} & a_{12}^{(k)} & a_{13}^{(k)} & b_1^{(k)} \\ 0 & a_{22}^{(k)} & a_{23}^{(k)} & b_2^{(k)} \\ 0 & 0 & a_{33}^{(k)} & b_3^{(k)} \end{array}\right]$$

- A partir da forma triangular, realizar a substituição regressiva.

$$\left[\begin{array}{ccc|c} a_{11}^{(k)} & a_{12}^{(k)} & a_{13}^{(k)} & b_1^{(k)} \\ 0 & a_{22}^{(k)} & a_{23}^{(k)} & b_2^{(k)} \\ 0 & 0 & a_{33}^{(k)} & b_3^{(k)} \end{array}\right] \rightarrow \begin{cases} x_3 = \frac{b_3^{(k)}}{a_{33}^{(k)}},\ a_{33}^{(k)} \neq 0 \\ x_2 = \frac{b_2^{(k)} - a_{23}^{(k)}.x_3}{a_{22}^{(k)}},\ a_{22}^{(k)} \neq 0 \\ x_1 = \frac{b_1^{(k)} - a_{12}^{(k)}.x_2 - a_{13}^{(k)}.x_3}{a_{11}^{(k)}},\ a_{11}^{(k)} \neq 0 \end{cases}$$

Vejamos um exemplo numérico de como funciona a Eliminação Gaussiana.

```
# matriz
M = np.array([[1.0,1.5,-2.0],[2.0,1.0,-1.0],[3.0,-1.0,2.0]])
print(M)
```

```
[[ 1.   1.5 -2. ]
 [ 2.   1.  -1. ]
 [ 3.  -1.   2. ]]
```

```
# zeramento da segunda linha
m1 = M[1,0]/M[0,0]
M[1,:] += - m1*M[0,:]
print(M)
```

```
[[ 1.   1.5 -2. ]
 [ 0.  -2.   3. ]
 [ 3.  -1.   2. ]]
```

```
# zeramento da terceira linha
m2 = M[2,0]/M[0,0]
M[2,:] += - m2*M[0,:]
print(M)
```

```
[[ 1.   1.5 -2. ]
 [ 0.  -2.   3. ]
 [ 0.  -5.5  8. ]]
```

```
# eliminação
M = np.array([[1.0,1.5,-2.0],[2.0,1.0,-1.0],[3.0,-1.0,2.0]])
b = np.array([-2,3,1])

m,n = M.shape
for i in range(m):
    for j in range(i+1,n):
        pivo = M[j,i]/M[i,i]
        for k in range(m):
            M[j,k] += -pivo*M[i,k]

print(M)
```

```
[[ 1.    1.5  -2.  ]
 [ 0.   -2.    3.  ]
 [ 0.    0.   -0.25]]
```

```
# função simples para eliminação
def eliminacao(M):
    m,n = M.shape
    for i in range(m):
        for j in range(i+1,n):
            pivo = M[j,i]/M[i,i]
            for k in range(m):
                M[j,k] += -pivo*M[i,k]
    return M
```

```
# matriz randômica 5x5
M2 = np.random.rand(5,5)
print(eliminacao(M2))
```

```
[[ 4.42389663e-01  6.77549133e-01  6.62310314e-01  8.92758362e-01
   8.67074978e-01]
 [ 0.00000000e+00 -8.94767973e-01 -1.17650434e+00 -1.23381565e+00
  -1.54014118e+00]
 [-1.11022302e-16  0.00000000e+00  1.14215572e-01 -6.53327797e-01
  -5.87564401e-02]
 [-2.58946634e-16  0.00000000e+00  0.00000000e+00 -1.60708761e+00
  -2.20279294e-01]
 [ 3.32024317e-17  0.00000000e+00  0.00000000e+00  0.00000000e+00
  -9.80998229e-01]]
```

18.2 Condicionamento

Vamos ver como pequenas "perturbações" em matrizes podem provocar mudanças drásticas nas soluções de sistemas. Isto ocorre quando temos um problema *mal condicionado*.

```
A1 = np.array([[1,2],[1.1,2]])
b1 = np.array([10,10.4])
print('matriz')
print(A1)
print('vetor')
print(b1)
```

```
matriz
[[1.  2. ]
 [1.1 2. ]]
vetor
[10.  10.4]
```

```
# solução do sistema A1x1 = b1
x1 = linalg.solve(A1,b1)
print(x1)
```

```
[4. 3.]
```

```
d = 0.045
A2 = np.array([[1,2],[1.1-d,2]])
b2 = np.array([10,10.4])
```

(continues on next page)

(continued from previous page)

```
print('matriz')
print(A2)
print('vetor')
print(b2)
```

```
matriz
[[1.    2.   ]
 [1.055 2.   ]]
vetor
[10.  10.4]
```

```
# solução do sistema perturbado A2x1 = b2
x2 = linalg.solve(A2,b2)
print(x2)
```

```
[7.27272727 1.36363636]
```

A solução muda drasticamente aqui! Isto se deve à quase dependência linear em que a matriz se encontra. Ou seja, $\det(\mathbf{A}_2) \approx 0$.

```
print(linalg.det(A1),linalg.det(A2))
```

```
-0.20000000000000018 -0.11000000000000032
```

```
linalg.norm(A)*linalg.norm(linalg.inv(A))
```

```
169.28388045827452
```

18.3 Método de Gauss-Jordan

O método de Gauss-Jordan é uma variação da eliminação de Gauss, em que não apenas as entradas da porção inferior da matriz plena do sistema são anuladas, mas também as entradas da porção superior, resultando em uma matriz diagonal.

Além disso, todas as linhas são normalizadas através da sua divisão pelo respectivo elemento pivô. Por exemplo, a_{11} é o elemento pivô da primeira equação, a_{22} da segunda, e assim por diante). A partir daí, a obtenção dos valores das variáveis é imediata.

O método é melhor ilustrado no seguinte exemplo.

```
# Matriz aumentada
AB = np.array([[3. , -0.1, -0.2, 7.85], [0.1, 7., -0.3, -19.3], [0.3, -0.2, 10., 71.
 ↪4]])
print(AB)
```

```
[[  3.    -0.1   -0.2    7.85]
 [  0.1    7.    -0.3  -19.3 ]
 [  0.3   -0.2   10.    71.4 ]]
```

```
# Normalização da 1a. linha
AB[0,:] /= AB[0,0] # L1 <- L1/a11
print(AB)
```

```
[[ 1.00000000e+00 -3.33333333e-02 -6.66666667e-02  2.61666667e+00]
 [ 1.00000000e-01  7.00000000e+00 -3.00000000e-01 -1.93000000e+01]
 [ 3.00000000e-01 -2.00000000e-01  1.00000000e+01  7.14000000e+01]]
```

```
# Eliminação de x1 da 2a. e 3a. linhas
m1 = AB[1,0]
AB[1,:] -= m1*AB[0,:] # L2 <- L2 - m1*L1
m2 = AB[2,0]
AB[2,:] -= m2*AB[0,:] # L3 <- L3 - m2*L1
print(AB)
```

```
[[ 1.00000000e+00 -3.33333333e-02 -6.66666667e-02  2.61666667e+00]
 [ 0.00000000e+00  7.00333333e+00 -2.93333333e-01 -1.95616667e+01]
 [ 0.00000000e+00 -1.90000000e-01  1.00200000e+01  7.06150000e+01]]
```

```
# Normalização da 2a. linha
AB[1,:] /= AB[1,1] # L2 <- L2/a22
print(AB)
```

```
[[ 1.00000000e+00 -3.33333333e-02 -6.66666667e-02  2.61666667e+00]
 [ 0.00000000e+00  1.00000000e+00 -4.18848168e-02 -2.79319372e+00]
 [ 0.00000000e+00 -1.90000000e-01  1.00200000e+01  7.06150000e+01]]
```

```
# Eliminação de x2 da 1a. e 3a. linhas
m3 = AB[0,1]
AB[0,:] -= m3*AB[1,:] # L1 <- L1 - m3*L2
m4 = AB[2,1]
AB[2,:] -= m4*AB[1,:] # L3 <- L3 - m4*L2
print(AB)
```

```
[[ 1.00000000e+00  0.00000000e+00 -6.80628272e-02  2.52356021e+00]
 [ 0.00000000e+00  1.00000000e+00 -4.18848168e-02 -2.79319372e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00120419e+01  7.00842932e+01]]
```

```
# Normalização da 3a. linha
AB[2,:] /= AB[2,2] # L3 <- L3/a33
print(AB)
```

```
[[ 1.          0.         -0.06806283  2.52356021]
 [ 0.          1.         -0.04188482 -2.79319372]
 [ 0.          0.          1.          7.        ]]
```

```
# Eliminação de x3 da 1a. e 2a. linhas
m5 = AB[0,2]
AB[0,:] -= m5*AB[2,:] # L1 <- L1 - m5*L3
m6 = AB[1,2]
AB[1,:] -= m6*AB[2,:] # L2 <- L2 - m5*L3
print(AB)
```

```
[[ 1.   0.   0.   3. ]
 [ 0.   1.   0.  -2.5]
 [ 0.   0.   1.   7. ]]
```

Do último resultado, vemos que a matriz identidade é obtida, apontando para o vetor solução $[3 \quad -2.5 \quad 7]^T$.

18.4 Notas

Para aqueles acostumados com a notação para matrizes do Matlab, o método `np.mat` pode ajudar. No exemplo a seguir, as linhas da matriz são passadas como uma expressão do tipo `str` e separadas com `;`.

```
# cria matriz
np.array(np.mat('1 2; 3 4'))
```

```
array([[1, 2],
       [3, 4]])
```

```
np.array(np.mat('1 2 3'))
```

```
array([[1, 2, 3]])
```

18.5 Tarefa

Implemente o algoritmo pleno para a eliminação gaussiana. Verifique exceções (erros que devem ser observados para evitar falhas no algoritmo, tal como pivôs nulos e matrizes singulares, por exemplo) e use o seu método para resolver os sistemas lineares da forma $\mathbf{Ax} = \mathbf{b}$ da lista de exercícios.

CHAPTER

NINETEEN

FATORAÇÃO LU COM PYTHON

19.1 Decomposição LU

Suponha o sistema de equações

$$AX = B$$

A motivação para a decomposição LU baseia-se na observação de que sistemas triangulares são mais fáceis de resolver. Semelhantemente à Eliminação Gaussiana, a decomposição LU (que, na verdade, é uma segunda abordagem da própria Eliminação Gaussiana), explora a ideia de "fatoração" de matrizes, em que a matriz original do sistema é fatorada ("quebrada") como

$$A = LU,$$

onde L é uma matriz triangular inferior e U é triangular superior. Nosso objetivo é determinar L e U, de maneira que o vetor X seja obtido através da resolução de um par de sistemas cujas matrizes são triangulares.

19.1.1 Exemplo

Consideraremos que as matrizes triangulares inferiores L sempre terão a sua diagonal principal formada por entradas iguais a 1. Este tipo de fatoração é conhecido como *Fatoração de Doolittle*.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix} = LU$$

onde

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \quad \text{e} \quad \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

Multiplicando LU e definindo a resposta igual a A temos:

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix}$$

Agora, através de substituição de recorrência, facilmente encontramos L e U.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 4 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

19.2 Usando a decomposição LU para resolver sistemas de equações

Uma vez decomposta a matriz A no produto LU, podemos obter a solução X de forma direta. O procedimento, equivalente à substituição de recorrência mencionada anteriormente, pode ser resumido como segue: dada A, encontre L e U tal que $A = LU$, ou seja, $(LU)X = B$. Em outras palavras:

- Defina $Y = UX$.
- Resolva o sistema triangular $LY = B$ para Y.
- Finalmente, resolva o sistema triangular $UX = Y$ para X.

O benefício desta abordagem é a resolução de somente sistemas triangulares. Por outro lado, o custo empregado é termos de resolver dois sistemas.

19.2.1 Exemplo

Encontre a solução $X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ do sistema

$$\begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 13 \\ 4 \end{bmatrix}.$$

- As matrizes L e U já foram obtidas anteriormente.

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 2 & 4 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

- A próxima etapa é resolver $LY = B$, para o vetor $Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$.

$$LY = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 13 \\ 4 \end{bmatrix} = B$$

Este sistema pode ser resolvido por substituição direta, obtendo $Y = \begin{bmatrix} 3 \\ 4 \\ -6 \end{bmatrix}$.

- Agora que encontramos Y, concluímos o procedimento resolvendo $UX = Y$ para X. Ou seja, resolvemos:

$$UX = \begin{bmatrix} 1 & 2 & 4 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ -6 \end{bmatrix}$$

Realizando a substituição regressiva (baixo para cima; da direita para a esquerda), obtemos a solução do problema.

$$X = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix}$$

Abaixo, temos uma implementação de uma fatoração LU sem pivoteamento.

```
import numpy as np

def lu_nopivot(A):
    '''
    Realiza fatoração LU para a matriz A

    entrada:
        A  -  matriz:  array (nxn)

    saida:
        L,U  - matriz triangular inferior, superior : array (nxn)
    '''

    n = np.shape(A)[0] # dimensao
    L = np.eye(n) # auxiliar

    for k in np.arange(n):
        L[k+1:n,k] = A[k+1:n,k]/A[k,k]
        for l in np.arange(k+1,n):
            A[l,:] = A[l,:] - np.dot(L[l,k],A[k,:])

    U = A
    return (L,U)
```

Exemplo: Fatoração **LU** da matriz

$$\mathbf{A} = \begin{bmatrix} 4 & -2 & -3 & 6 \\ 1 & 4 & 2 & 3 \\ 2 & -3 & 3 & -2 \\ 1 & 5 & 3 & 4 \end{bmatrix}$$

```
A = np.array([[ 4., -2., -3.,  6.],[ 1.,  4.,  2.,  3.],[ 2.,  -3.,  3., -2.],[ 1., 
 ↪5.,  3.,  4.]])

L,U = lu_nopivot(A)
```

```
L
```

```
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.25      ,  1.        ,  0.        ,  0.        ],
       [ 0.5       , -0.44444444,  1.        ,  0.        ],
       [ 0.25      ,  1.22222222,  0.06796117,  1.        ]])
```

```
U
```

```
array([[ 4.        , -2.        , -3.        ,  6.        ],
       [ 0.        ,  4.5       ,  2.75      ,  1.5       ],
       [ 0.        ,  0.        ,  5.72222222, -4.33333333],
       [ 0.        ,  0.        ,  0.        ,  0.96116505]])
```

CHAPTER

TWENTY

FATORAÇÃO DE CHOLESKY

```
%matplotlib inline
```

20.1 Matrizes positivas definidas

Definição (baseada em autovalores) uma matriz **A** é *positiva definida* se todos os seus autovalores são positivos ($\lambda > 0$).

Entretanto, não é conveniente computar todos os autovalores de uma matriz para saber se ela é ou não positiva definida. Há meios mais rápidos de fazer este teste como o da "energia".

Definição (baseada em energia): uma matriz **A** é *positiva definida* se $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ para todo vetor não-nulo **x**.

Para a ordem $n = 2$, temos os eguinte resultado:

$$\mathbf{x}^T\mathbf{A}\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = a_{11}x_1^2 + 2a_{12}x_1x_2 + a_{22}x_2^2 > 0$$

Em muitas aplicações, este número é a "energia" no sistema.

O código abaixo mostra que esta multiplicação produz uma forma quadrática para a ordem n.

```
import sympy as sp
sp.init_printing(use_unicode=True)

x1,x2,x3 = sp.symbols('x1,x2,x3')

# ordem do sistema
n = 3

x = sp.zeros(1,n)
A = sp.zeros(n,n)
aux = 0*A;
for i in range(n):
    x[i] = sp.symbols('x' + str(i+1))
    for j in range(n):
        if i == j: # diagonal
            A[i,i] = sp.symbols('a' + str(i+1) + str(i+1))
        elif j > i: # triang. superior
            aux[i,j] = sp.symbols('a' + str(i+1) + str(j+1))

# matriz
A = A + aux.T + aux # compõe D + L + U
A = sp.Matrix(A)
```

(continues on next page)

(continued from previous page)

```
# vetor
x = sp.Matrix(x)
x = x.T

# matriz positiva definida
c = (x.T)*A*x

# expressãoo quadrática
sp.expand(c[0])
```

$$a_{11}x_1^2 + 2a_{12}x_1x_2 + 2a_{13}x_1x_3 + a_{22}x_2^2 + 2a_{23}x_2x_3 + a_{33}x_3^2$$

Ainda falando sobre o caso $n = 2$, observamos que os autovalores da matriz **A** são positivos se, e somente se

$$a_{11} > 0 \qquad \text{e} \qquad a_{11}a_{22} - a_{12}^2 > 0.$$

Na verdade, esta regra vale para todos os **pivôs**. Estes dois últimos valores são os pivôs de uma matriz simétrica 2x2 (verifique a eliminação de Gauss quando aplicada à segunda equação).

A teoria da Álgebra Linear permite-nos elencar as seguintes declarações, **todas equivalentes**, acerca da determinação de uma matriz positiva definida **A**:

1. Todos os seus n pivôs são positivos.
2. Todos os determinantes menores superiores esquerdos (ou principais) são positivos (veja Critério de Sylvester).
3. Todos os seus n autovalores são positivos.
4. $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ para todo vetor não-nulo **x**. (Definição baseada na "energia").
5. $\mathbf{A} = \mathbf{G}\mathbf{G}^T$ para uma matriz **G** com colunas independentes.

20.1.1 Interpretação geométrica

Matrizes positivas definidas realizam transformações "limitadas" no sentido de "semiplano" do vetor transformado. Por exemplo, se tomarmos um vetor $\mathbf{x} \in \mathbb{R}^2$ não-nulo e usarmos o fato de que para uma matriz positiva definida, a inequação $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ deve valer, ao chamarmos $\mathbf{y} = \mathbf{A}\mathbf{x}$, a expressão anterior é o produto interno entre **x** e **y**, a saber $\mathbf{x}^T\mathbf{y}$. Se o produto interno é positivo, já sabemos que os vetores não são ortogonais. Agora, para verificar que eles realmente pertencem a um mesmo semiplano, usaremos a seguinte expressão para o ângulo entre dois vetores:

$$\cos(\theta) = \frac{\mathbf{x}^T\mathbf{y}}{||\mathbf{x}||\mathbf{y}||}$$

Uma vez que a norma (comprimento) de um vetor é sempre um número real positivo, o produto $||\mathbf{x}||\mathbf{y}||$ no denominador acima é um número real positivo. Se $\mathbf{x}^T\mathbf{y} > 0$, então $\cos(\theta) > 0$, e o efeito geométrico da transformação é

$$|\theta| < \frac{\pi}{2},$$

ou seja, o ângulo entre **x** e **y** é sempre menor do que 90 graus (ou $\pi/2$ radianos).

20.2 Método da Fatoração de Cholesky

Trata-se de um algoritmo para resolução de sistemas lineares $\mathbf{Ax} = \mathbf{b}$ através da decomposição da matriz $\mathbf{A}$ em dois fatores simétricos, $\mathbf{G}$ e $\mathbf{G}^T$. O método é aplicável apenas a sistemas cuja matriz associada é simétrica e positiva definida.

20.2.1 Passos

- Primeiramente, é necessário checar se a matriz associada ao sistema cumpre os requisitos da fatoração de Cholesky:
  - **Simetria**: a matriz é simétrica quando sua transposta é igual a ela própria: $A = A^T$
  - **Definição positiva**: averiguamos se o Critério de Sylvester é satisfeito. Ou seja, verificamos se todos os determinantes menores principais da matriz, constituídos pelas k primeiras linhas e k primeiras colunas dela, são maiores do que zero:

$$\det(\mathbf{A}_k) > 0, \text{ onde } k = 1, 2, \dots, n \text{ para matrizes } A_{nxn}$$

Concluídas as verificações anteriores, decompomos a matriz $\mathbf{A}_{nxn}$ em uma triangular inferior $\mathbf{A}$ e sua transposta $\mathbf{A}^T$, a qual é triangular superior. O processo é descrito abaixo:

- Para uma matriz A_{4x4} obtém-se o fator de Cholesky da seguinte forma:

$$\mathbf{A} = \mathbf{G}\mathbf{G}^T \Rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} g_{11} & & & \\ g_{21} & g_{22} & & \\ g_{31} & g_{32} & g_{33} & \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ & g_{22} & g_{32} & g_{42} \\ & & g_{33} & g_{43} \\ & & & g_{44} \end{bmatrix}$$

- Este sistema pode ser resolvido comparando cada coluna de $\mathbf{A}$ com a multiplicação de $\mathbf{G}$ por cada coluna de $\mathbf{G}^T$ de maneira que $\mathbf{x}_{kxn} = \mathbf{G}\mathbf{G}^T_{kxn}$, $k = 1, 2, \dots, n$. Através dessa sequência são obtidos sistemas simples, em que cada coluna k terá os valores de g_{kn}e, após a atualização do iterador $k \to k+1$, não será necessário resolver novamente as $k-1$ linhas do próximo sistema gerado.

De posse dos valores que geram as matrizes triangulares, é possível seguir para a última etapa, a qual consiste em obter o vetor $\mathbf{x}$ fazendo

$$\mathbf{Ax} = \mathbf{b} \implies (\mathbf{G}\mathbf{G}^T)\mathbf{x} = \mathbf{b} \implies \mathbf{G}(\mathbf{G}^T\mathbf{x}) = \mathbf{b} \implies \mathbf{Gy} = B.$$

Aqui, usamos o mesmo processo utilizado na Fatoração LU. Primeiramente, obtemos o vetor $\mathbf{y}$ após a resolução do sistema $\mathbf{Gy} = \mathbf{b}$. Enfim, obtemos $\mathbf{x}$ através da relação $\mathbf{G}^T\mathbf{x} = \mathbf{y}$. Fazendo com que, assim, seja obtido o vetor $\mathbf{x}$ que resolve o sistema linear proposto no inicio.

20.3 Algoritmo para a fatoração de Cholesky

O código abaixo é uma implementação de um algoritmo para a fatoração de Cholesky por computação simbólica.

```
# implementação de algoritmo simbólico
# para a decomposição de Cholesky

B = A[:,:] # faz cópia da matriz A

for k in range(0,n):
    for i in range(0,k):
        s = 0.
        for j in range(0,i):
            s += B[i,j]*B[k,j]
```

(continues on next page)

(continued from previous page)

```
        B[k,i] = (B[k,i] - s)/B[i,i]
    s = 0.
    for j in range(0,k):
        s += s + B[k,j]*B[k,j]
    B[k,k] = sp.sqrt(B[k,k] - s)

# saída
B
```

$$\begin{bmatrix} \sqrt{a_{11}} & a_{12} & a_{13} \\ \frac{a_{12}}{\sqrt{a_{11}}} & \sqrt{a_{22} - \frac{a_{12}^2}{a_{11}}} & a_{23} \\ \frac{a_{13}}{\sqrt{a_{11}}} & \frac{a_{23} - \frac{a_{12}a_{13}}{a_{11}}}{\sqrt{a_{22} - \frac{a_{12}^2}{a_{11}}}} & \sqrt{a_{33} - \frac{\left(a_{23} - \frac{a_{12}a_{13}}{a_{11}}\right)^2}{a_{22} - \frac{a_{12}^2}{a_{11}}} - \frac{2a_{13}^2}{a_{11}}} \end{bmatrix}$$

20.3.1 Tarefa

Converta o código simbólico acima para uma versão numérica (ou implemente a sua própria versão) e aplique-o na matriz abaixo para encontrar o fator de Cholesky:

$$\mathbf{A} = \begin{bmatrix} 6 & 15 & 55 \\ 15 & 55 & 225 \\ 55 & 225 & 979 \end{bmatrix}$$

20.4 Cálculo do fator de Cholesky com Python

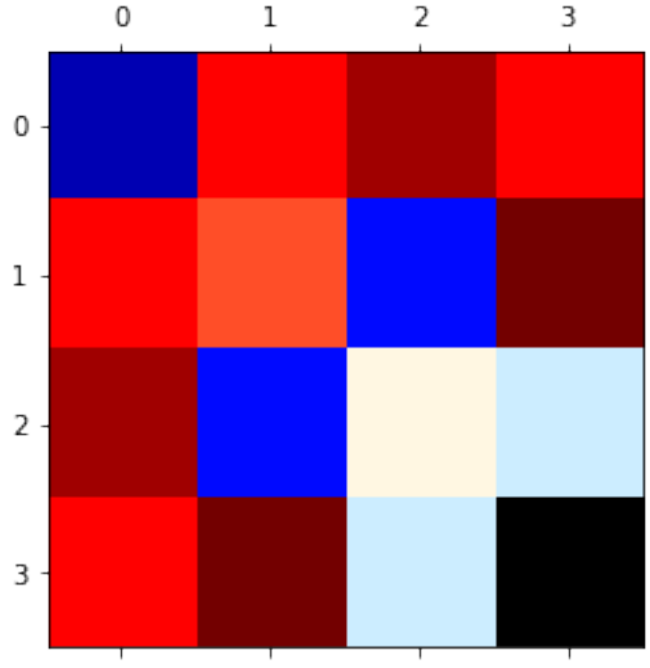
```
import matplotlib.pyplot as plt
from scipy import array, linalg, dot, zeros

# matriz
A = array([[16, -4, 12, -4],
           [-4, 2, -1, 1],
           [12, -1, 14, -2],
           [-4, 1, -2, 83]])

# fator de Cholesky do Scipy
L = linalg.cholesky(A, lower=True, overwrite_a=False, check_finite=True)

# fator de Cholesky implementado
n = A.shape[0]
G = zeros(A.shape, dtype=float)

print('Matriz A = \n', A)
print('Matriz L = \n', L)
print('Matriz L^T = \n', L.T)

# prova real por produto interno
A2 = dot(L, L.T)
print('Matriz LL^T = \n', A2)
```

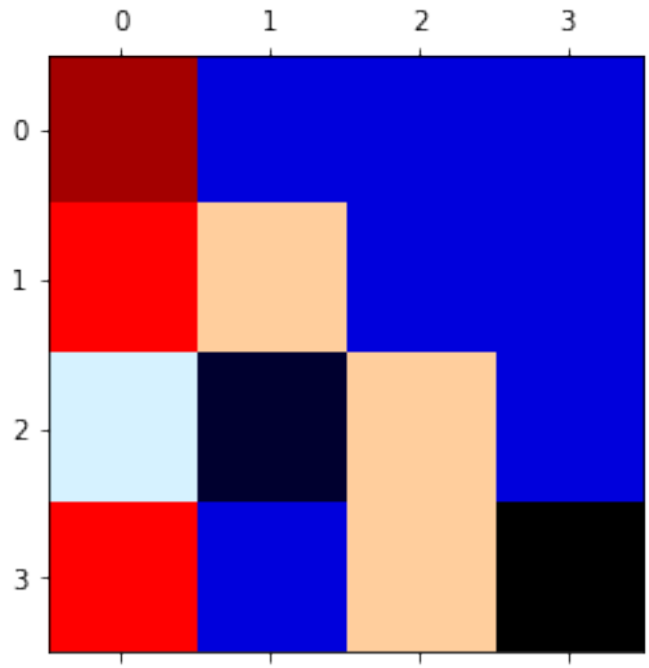
(continues on next page)

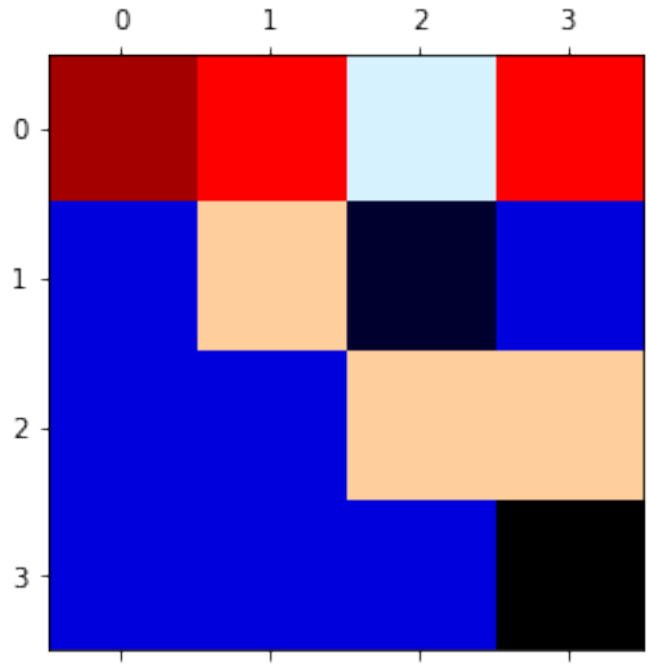
(continued from previous page)

```
# prova real usando norma de Frobenius da diferenca de matrizes
print('Norma || A - LL^T || = ', linalg.norm(A-A2))

plt.matshow(A,cmap=plt.cm.flag);
plt.matshow(L,cmap=plt.cm.flag);
plt.matshow(L.T,cmap=plt.cm.flag);
```

```
Matriz A =
 [[16 -4 12 -4]
 [-4  2 -1  1]
 [12 -1 14 -2]
 [-4  1 -2 83]]
Matriz L =
 [[ 4.  0.  0.  0.]
 [-1.  1.  0.  0.]
 [ 3.  2.  1.  0.]
 [-1.  0.  1.  9.]]
Matriz L^T =
 [[ 4. -1.  3. -1.]
 [ 0.  1.  2.  0.]
 [ 0.  0.  1.  1.]
 [ 0.  0.  0.  9.]]
Matriz LL^T =
 [[16. -4. 12. -4.]
 [-4.  2. -1.  1.]
 [12. -1. 14. -2.]
 [-4.  1. -2. 83.]]
Norma || A - LL^T || =  0.0
```




0
1
2
3
0
1
2
3


0
1
2
3
0
1
2
3

CHAPTER

TWENTYONE

MÉTODO DE JACOBI-RICHARDSON

O método de Jacobi-Richardson (MJR) é um método iterativo que busca uma solução aproximada para sistemas lineares. Um método iterativo como tal é também conhecido como *aproximações sucessivas*, em que uma sequência convergente de vetores é desejada. O MJR é de especial interesse em sistemas cujas matrizes são *diagonalmente dominantes*.

21.1 Instalações necessárias

Este notebook usa o módulo `plotly` para plotagem. Para instalá-lo via `conda` e habilitá-lo para uso como extensão do Jupyter Lab, execute os comandos abaixo em uma célula:

```
import sys
!conda install --yes --prefix {sys.prefix} nodejs plotly
!jupyter labextension install jupyterlab-plotly@4.12.0
```

21.2 Descrição do método

Suponha um sistema linear nas incógnitas $x_1, x_2, ..., x_n$ da seguinte forma:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

Suponha também que todos os termos a_{ii} sejam diferentes de zero $(i = 1, ..., n)$. Se não for o caso, isso pode, geralmente, ser resolvido permutando equações.

Inicialmente, o método sugere que as variáveis sejam isoladas em cada equação. Assim, escrevemos

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}[b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n] \\ x_2 &= \frac{1}{a_{22}}[b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n] \\ &\vdots \\ x_n &= \frac{1}{a_{nn}}[b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n\,n-1}x_{n-1}] \end{aligned}$$

A partir dessas equações "isoladas" e de um vetor de estimativa inicial ("chute inicial") $[x_1^{(0)} \;\; x_2^{(0)} \;\; \dots \;\; x_n^{(0)}]^T$, criamos o processo iterativo inserindo, à direita, um contador.

No primeiro passo, obtemos $[x_1^{(1)} \;\; x_2^{(1)} \;\; \dots \;\; x_n^{(1)}]^T$. Em seguida, estimamos $[x_1^{(2)} \;\; x_2^{(2)} \;\; \dots \;\; x_n^{(2)}]^T$. Repetindo o processo, a iterada k é construída como segue:

$$
\begin{aligned}
x_1^{(k+1)} &= \frac{1}{a_{11}}[b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)}] \\
x_2^{(k+1)} &= \frac{1}{a_{22}}[b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)}] \\
&\vdots \\
x_n^{(k+1)} &= \frac{1}{a_{nn}}[b_n - a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \cdots - a_{n\,n-1}x_{n-1}^{(k)}]
\end{aligned}
$$

Logo, para todo $i = 1, 2, \dots, n$, esperamos que a sequência $\{x_i^{(k+1)}\}_k$ convirja para o vetor solução $\mathbf{x}$ do sistema original, caracterizado por $\mathbf{Ax} = \mathbf{b}$.

Entretanto, a convergência é garantida apenas se houver dominância dos valores da diagonal principal sobre seus pares nas mesmas linhas. Podemos verificar a convergência do processo iterativo acima por meio do *critério das linhas*, embora exista uma maneira mais versátil de fazer esta verificação usando normas de matrizes. Por enquanto, entendamos o critério das linhas.

21.3 Critério das linhas

O critério das linhas estabelece que

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| < |a_{ii}|, \forall i = 1, 2, \dots, n.$$

Em palavras: *"o valor absoluto do termo diagonal na linha i é maior do que a soma dos valores absolutos de todos os outros termos na mesma linha"*. Se satisfeita, esta equação implica que a matriz $\mathbf{A}$ é (estritamente) diagonalmente dominante.

É importante observar que o critério das linhas pode deixar de ser satisfeito se houver troca na ordem das equações, e vice-versa. Todavia, uma troca cuidadosa pode fazer com que o sistema passe a satisfazer o critério. Por teorema, admite-se que se a matriz do sistema linear satisfaz o critério das linhas, então o algoritmo de Jacobi-Richardson converge para uma solução aproximada.

21.4 Vantagens e desvantagens do método

O método tem a vantagem de ser de fácil implementação no computador do que o método de escalonamento e está menos propenso à propagação de erros de arredondamento. Por outro lado, sua desvantagem é não funcionar para todos os casos.

21.5 Implementação do método de Jacobi

```
"""
JACOBI: metodo de Jacobi para solução do sistema Ax=b.

x = jacobi(A,b,x0,N)

entrada:
   A: matriz n x n (tipo: lista de listas)
   b:  vetor n x 1 (tipo: lista)
  x0:  vetor n x 1, ponto de partida (tipo: lista)
   N:  numero de iterações do método
saidas:
x:  matriz n x N,
    contendo a sequencia de aproximações da solução
sp:  norma de B. O metodo converge se sp < 1.
"""
import numpy as np
from matplotlib.pyplot import plot

"""
JACOBI: metodo de Jacobi para solução do sistema Ax=b.

x = jacobi(A,b,x0,N)

entrada:
   A: matriz n x n (tipo: lista de listas)
   b:  vetor n x 1 (tipo: lista)
  x0:  vetor n x 1, ponto de partida (tipo: lista)
   N:  numero de iterações do método
saidas:
x:  matriz n x N,
    contendo a sequencia de aproximações da solução
v:  norma de G. O metodo converge se v < 1.
"""
def jacobi(A,b,x0,N):

    f = lambda obj: isinstance(obj,list)
    if not all([f(A),f(b),f(x0)]):
        raise TypeError('A, b e x0 devem ser do tipo list.')
    else:
        A = np.asarray(A)
        b = np.asarray(b)
        x0 = np.asarray(x0)

    n = np.size(b)
    x = np.zeros((n,N),dtype=float)
    x[:,0] = x0

    P = np.diag(np.diag(A))
    Q = P-A # elementos de fora da diagonal
    G = np.linalg.solve(P,Q) # matriz da função de iteração
    c = np.linalg.solve(P,b) # vetor da função de iteração
    v = np.linalg.norm(G)

    #processo iterativo
    X = x0[:]
```

(continues on next page)

(continued from previous page)

```
    j = 1
    for j in range(1,N):
        X = G.dot(X) + c
        x[:,j] = X

    return x,v
```

```
# Exemplo 15.2
A = [[10,2,1],[1,5,1],[2,3,10]]
b = [7,-8,6]
x = [1,1,1]
N = 10

sol,v = jacobi(A,b,x,N)
print(sol[:,-1])
print(v)
```

```
[ 0.99977882 -2.00026225  0.9996773 ]
0.5099019513592785
```

21.5.1 Tarefa para turma de computação:

(Este código depende da biblioteca `plotly`, mas não *está otimizado*.)

Desenvolver código para plotagem 3D da convergência para o método de Jacobi.

```
# plotagem 3D da convergência
import plotly.graph_objs as go
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=False)

xp = sol[0,:]
yp = sol[1,:]
zp = sol[2,:]

points = go.Scatter3d(x = xp, y = yp, z = zp, mode = 'markers', marker = dict(size =␣
 ↪0.1,color = "rgb(227,26,28)"))

vecs = []
for i in range(len(xp)):
    v = go.Scatter3d( x = [0,xp[i]], y = [0,yp[i]], z = [0,zp[i]], marker = dict(size␣
 ↪= 0.1, color = "rgb(0,255,0)"),
                      line = dict( color = "rgb(0,255,0)", width = 4) )
    vecs.append(v)


cx = np.sum(xp)/np.size(xp)
cy = np.sum(yp)/np.size(yp)
cz = np.sum(zp)/np.size(zp)

vector = go.Scatter3d( x = [0,cx], y = [0,cy], z = [0,cz], marker = dict(size = 1,␣
 ↪color = "rgb(100,100,100)"),
                      line = dict( color = "rgb(100,100,100)", width = 4) )
```

(continues on next page)

(continued from previous page)

```
data = [points,vector] + vecs
layout = go.Layout(margin = dict( l = 0,r = 0, b = 0, t = 0))
fig = go.Figure(data=data,layout=layout)


# vector

fig.add_cone(x=[cx],y=[cy],z=[cz],u=[cx],v=[cy],w=[cz],sizeref=0.1,anchor='tip',
 ↪colorscale='gray')

for i in range(len(xp)):
    fig.add_cone(x=[xp[i]],y=[yp[i]],z=[zp[i]],u=[xp[i]],v=[yp[i]],w=[zp[i]],
 ↪sizeref=0.1,anchor='tip',colorscale='jet')


iplot(fig, show_link=True,filename='jacobi-3d-vectors')
```

CHAPTER

TWENTYTWO

MÉTODO DE NEWTON PARA SISTEMAS NÃO-LINEARES

Este método determina, a cada iteração, a solução aproximada do sistema não-linear através de uma linearização das funções-alvo com a matriz Jacobiana associada ao sistema.

22.1 Passos

Para o método de Newton não-linear, basicamente criamos uma espécie de "caminho" onde somamos um vetor de deslocamento **s** às aproximações sucessivas que dá a direção para onde os vetores devem prosseguir a fim de atingir convergência.

Obs.: este processo iterativo usa critérios de parada naturais em algoritmos iterativos.

Para encontrarmos o vetor solução, devemos resolver a equação matricial linearizada

$$\mathbf{J}(\mathbf{x}^{(i)})\mathbf{s}^{(i)} = -\mathbf{F}(\mathbf{x}^{(i)})$$

Em seguida, atualizamos o novo vetor da sequencia como:

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \mathbf{s}^{(i)}.$$

Acima, $\mathbf{J}(\mathbf{x}^{(i)})$ é a matriz Jacobiana formada a partir das derivadas parciais das funções componentes do vetor $\mathbf{F}$.

No caso de um sistema em que $\mathbf{F} = [f_1(x_1, x_2) \;\; f_2(x_1, x_2)]^T$, teríamos o sistema abaixo:

$$\begin{bmatrix} \frac{\partial f_1(x_1,x_2)}{\partial x_1} & \frac{\partial f_1(x_1,x_2)}{\partial x_2} \\ \frac{\partial f_2(x_1,x_2)}{\partial x_1} & \frac{\partial f_2(x_1,x_2)}{\partial x_2} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \end{bmatrix} = \begin{bmatrix} -f_1(x_1, x_2) \\ -f_2(x_1, x_2) \end{bmatrix}$$

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import sympy as sy
from scipy.optimize import root
```

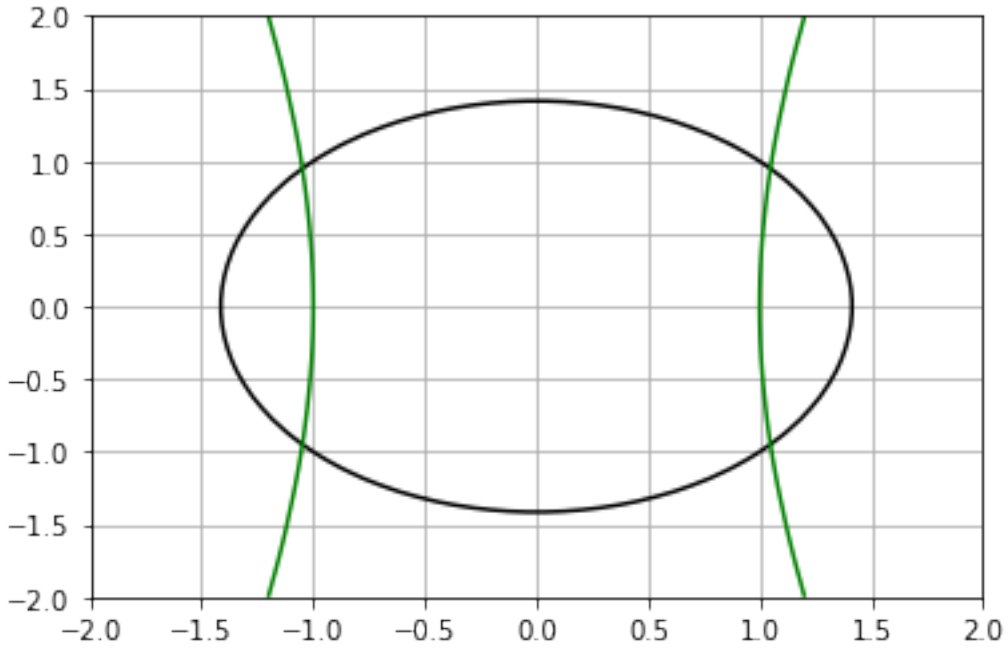
No exemplo a seguir, mostramos como podemos resolver um sistema de equações não-lineares usando o `scipy`.

Procuramos as soluções para o sistema não-linear

$$\begin{cases} f_1(x, y) & : x^2 + y^2 = 2 \\ f_2(x, y) & : x^2 - \frac{y^2}{9} = 1 \end{cases}$$

Vamos plotar o gráficos das funções:

```
x = np.linspace(-2,2,50,endpoint=True)
y = x[:]
X,Y = np.meshgrid(x,y)

# define funções para plotagem
F1 = X**2 + Y**2 - 2
F2 = X**2 - Y**2/9 - 1

# curvas de nível
C = plt.contour(X,Y,F1,levels=[0],colors='k')
C = plt.contour(X,Y,F2,levels=[0],colors='g')
plt.grid(True)
```



Pela figura, vemos que existem 4 pontos de interseção entre as curvas e, portanto, 4 soluções, as quais formam o conjunto

$$S = \{(x_1^*, y_1^*), (x_2^*, y_2^*), (x_3^*, y_3^*), (x_4^*, y_4^*)\}$$

Agora, vamos usar a função `root` do `scipy` para computar essas soluções com base em estimativas iniciais.

```
# define função para o vetor F(x)
def F(x):
    return [ x[0]**2 + x[1]**2 - 2,
             x[0]**2 - x[1]**2/9 - 1 ]


x,y = sy.symbols('x,y')

# usa computação simbólica para determinar a matriz Jacobiana
f1 = x**2 + y**2 - 2
f2 = x**2 - y**2/9 - 1

# gradientes

f1x,f1y = sy.diff(f1,x),sy.diff(f1,y)
```

(continues on next page)

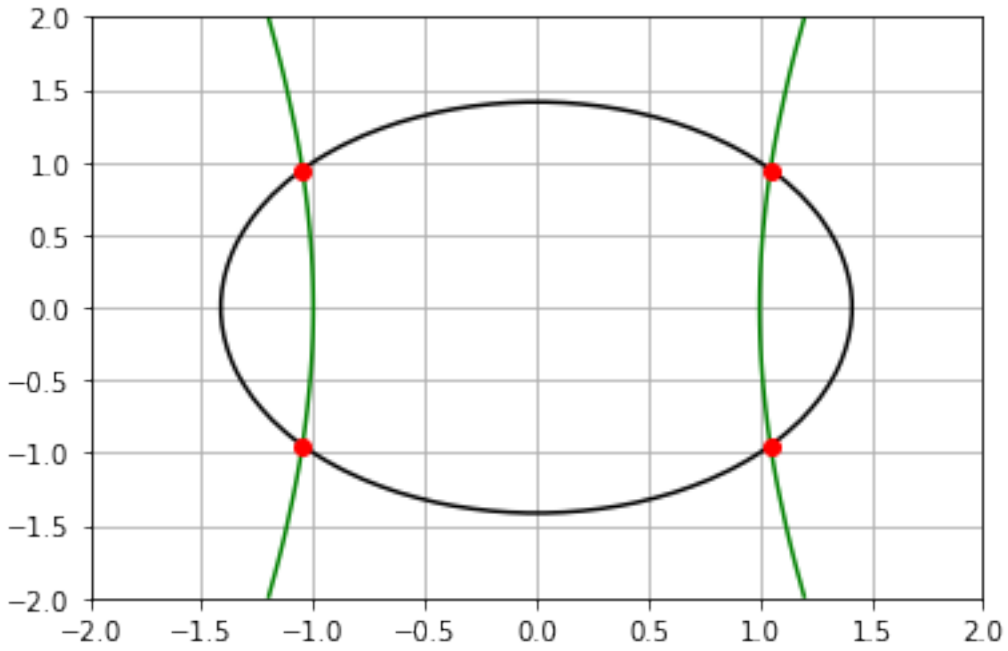
(continued from previous page)

```
f2x,f2y = sy.diff(f2,x),sy.diff(f2,y)

# imprime derivadas parciais
print(f1x)
print(f1y)
print(f2x)
print(f2y)

# monta matriz Jacobiana
def jacobian(x):
    return np.array([[2*x[0], 2*x[1]], [2*x[0],-2*x[1]/9]])

# resolve o sistema não-linear por algoritmo de Levenberg-Marqardt modificado

inicial = [[2,2],[-2,2],[-2,-2],[2,-2]]

S = []
i = 1
for vetor in inicial:
    aux = root(F,vetor,jac=jacobian, method='lm')
    S.append(aux.x)
    s = 'Solução x({0})* encontrada: {1}'
    print(s.format(i,aux.x))
    i +=1
```

```
2*x
2*y
2*x
-2*y/9
Solução x(1)* encontrada: [1.04880885 0.9486833 ]
Solução x(2)* encontrada: [-1.04880885  0.9486833 ]
Solução x(3)* encontrada: [-1.04880885 -0.9486833 ]
Solução x(4)* encontrada: [ 1.04880885 -0.9486833 ]
```

Em seguida, vamos plotar as soluções e as curvas

```
# curvas de nível
C = plt.contour(X,Y,F1,levels=[0],colors='k')
C = plt.contour(X,Y,F2,levels=[0],colors='g')
plt.grid(True)

# imprime interseções
for i in range(len(S)):
        plt.plot(S[i][0],S[i][1],'or')
```



22.1.1 Exercício:

Resolva os sistemas não-lineares da Lista de Exercícios 4 usando a mesma abordagem acima.

CHAPTER

TWENTYTHREE

NOTA: RAÍZES DE SISTEMAS NÃO-LINEARES

- Uma equação linear tem a forma:

$$f(x) = a_1x_1 + a_2x_2 + ... + a_nx_n$$

- Uma equação não-linear possui "produtos de incógnitas", e.g.

$$f_2(x) = a_1x_1x_2 + a_2x_2^2 + a_nx_nx_1$$

- Um sistema de equações não-lineares é composto de várias equações não-lineares

$$f_1(x_1, x_2, ..., x_n) = 0$$

$$f_2(x_1, x_2, ..., x_n) = 0$$

$$\vdots$$

$$f_n(x_1, x_2, ..., x_n) = 0$$

- A solução do sistema é o vetor $(x_1^*, x_2^*, ..., x_n^*)$ que satisfaz as n equações simultaneamente.

23.1 Iteração de Ponto Fixo para sistemas não-lineares

- Aplicar o algoritmo iterativo em cada componente:

$$x_1^{i+1} = \tilde{f}_1(x_1^i, x_2^i, ..., x_n^i)$$

$$x_2^{i+1} = \tilde{f}_2(x_1^i, x_2^i, ..., x_n^i)$$

$$\vdots$$

$$x_3^{i+1} = \tilde{f}_3(x_1^i, x_2^i, ..., x_n^i)$$

As formas funcionais mudam porque devemos isolar a variável x_i.

Exemplo: encontrar a raiz do sistema abaixo:

$$f_1(x, y) = x^2 + xy - 10 = 0$$

$$f_2(x, y) = y + 3xy^2 - 57 = 0$$

Solução:

Reescrevamos as equações na forma

$$x = \tilde{f}_1(x, y) = \sqrt{10 - xy}$$

$$y = \tilde{f}_2(x, y) = \sqrt{\frac{57 - y}{3x}}$$

de onde temos a iteração de ponto fixo dada por

$$x^{i+1} = \sqrt{10 - x^i y^i}$$

$$y^{i+1} = \sqrt{\frac{57 - y^i}{3x^i}}, \quad i = 0, 1, 2, \ldots,$$

Usando $(x^0, y^0) = (1.5, 3.5)$ como "chute" inicial, computamos

$$x^1 = \sqrt{10 - x^0 y^0} = \sqrt{10 - 1.5(3.5)} = 2.17945$$

$$y^1 = \sqrt{\frac{57 - y^0}{3x^1}} = \sqrt{\frac{57 - 3.5}{3(2.17945)}} = 2.86051 (ovalorde\ \mathrm{x}^1 \text{ pode ser usado diretamente em vez de } x^0.)$$

$$y^2 = \sqrt{\frac{57 - y^1}{3x^2}} = \sqrt{\frac{57 - 2.86051}{3(1.94053)}} = 3.04955$$

...

O processo iterativo converge para a solução $(x^*, y^*) = (2, 3)$.

Notas:

- A convergência por iteração de PF depende de como as equações $\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_n$ são formuladas, bem como de um bom "chute" inicial.
- A iteração de PF é bastante restritiva nas soluções de sistemas não-lineares.

23.2 Newton-Raphson para sistema não-linear

- Depende de série de Taylor em $n-$dimensões.
- Para 2 dimensões, por exemplo, o método de Newton-Raphson pode ser escrito como:

$$u_{i+1} = u_i + (x_{i+1} - x_i)\frac{\partial u_i}{\partial x} + (y_{i+1} - y_i)\frac{\partial u_i}{\partial y}$$

$$v_{i+1} = v_i + (x_{i+1} - x_i)\frac{\partial v_i}{\partial x} + (y_{i+1} - y_i)\frac{\partial v_i}{\partial y}$$

- A estimativa da raiz corresponde aos valores de x e y para os quais $u_{i+1} = 0$ e $v_{i+1} = 0$. Então,

$$\frac{\partial u_i}{\partial x}x_{i+1} + \frac{\partial u_i}{\partial y}y_{i+1} = -u_i + \frac{\partial u_i}{\partial x}x_i + \frac{\partial u_i}{\partial y}y_i$$

$$\frac{\partial v_i}{\partial x}x_{i+1} + \frac{\partial v_i}{\partial y}y_{i+1} = -v_i + \frac{\partial v_i}{\partial x}x_i + \frac{\partial v_i}{\partial y}y_i,$$

que é um sistema nas incógnitas x_{i+1} e y_{i+1}.

- Manipulações algébricas permitem solucionar este sistema (e.g. regra de Cramer):

$$x_{i+1} = x_i - J^{-1}\left(u_i\frac{\partial v_i}{\partial y} - v_i\frac{\partial u_i}{\partial y}\right)$$

$$y_{i+1} = y_i - J^{-1}\left(v_i\frac{\partial u_i}{\partial x} - u_i\frac{\partial v_i}{\partial x}\right),$$

onde

$$J = \frac{\partial u_i}{\partial x}\frac{\partial v_i}{\partial y} - \frac{\partial u_i}{\partial y}\frac{\partial v_i}{\partial x}$$

é o determinante da matriz *Jacobiana* do sistema.

Exemplo: resolver o mesmo sistema do PF

Solução:

- Inicialmente, calculemos as derivadas parciais no ponto inicial:

$$\frac{\partial u_0}{\partial x} = 2x_0 + y_0 = 2(1.5) + 3.5 = 6.5, \quad \frac{\partial u_0}{\partial y} = x_0 = 1.5$$

$$\frac{\partial v_0}{\partial x} = 3y_0^2 = 3(3.5)^2 = 36.75, \quad \frac{\partial v_0}{\partial y} = 1 + 6x_0y_0 = 1 + 6(1.5)(3.5) = 32.5$$

- O determinante é $J = 6.5(32.5) - 1.5(36.75) = 156.125$.
- Calculamos os valores da função no ponto inicial

$$u(x_0, y_0) = u_0 = -2.5, \quad v(x_0, y_0) = v_0 = 1.625$$

- Calculamos os valores no próximo passo, i.e., (x^1, y^1).

$$x^1 = 1.5 - \frac{\cdots}{156.125} = 2.03603$$

$$y^1 = 3.5 - \frac{\cdots}{156.125} = 2.84388$$

- O processo iterativo converge para a solução $(x^*, y^*) = (2, 3)$.

Ref.: Chapra & Canale, sec. 6.6

```
%matplotlib inline
```

CHAPTER

TWENTYFOUR

POLINÔMIO INTERPOLADOR DE LAGRANGE

O polinômio interpolador de Lagrange é representado por:

$$P_n(x) = \sum_{i=0}^{n} L_i(x) f(x_i)$$

onde

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

Por exemplo, a versão linear ($n = 1$) seria

$$P_1(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1)$$

e a versão de segundo grau ($n = 1$) seria

$$P_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f(x_0) + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f(x_1) + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f(x_2)$$

CHAPTER

TWENTYFIVE

FUNÇÕES DE BASE DE LAGRANGE

Código gerador de funções de base de Lagrange de grau n por computação simbólica.

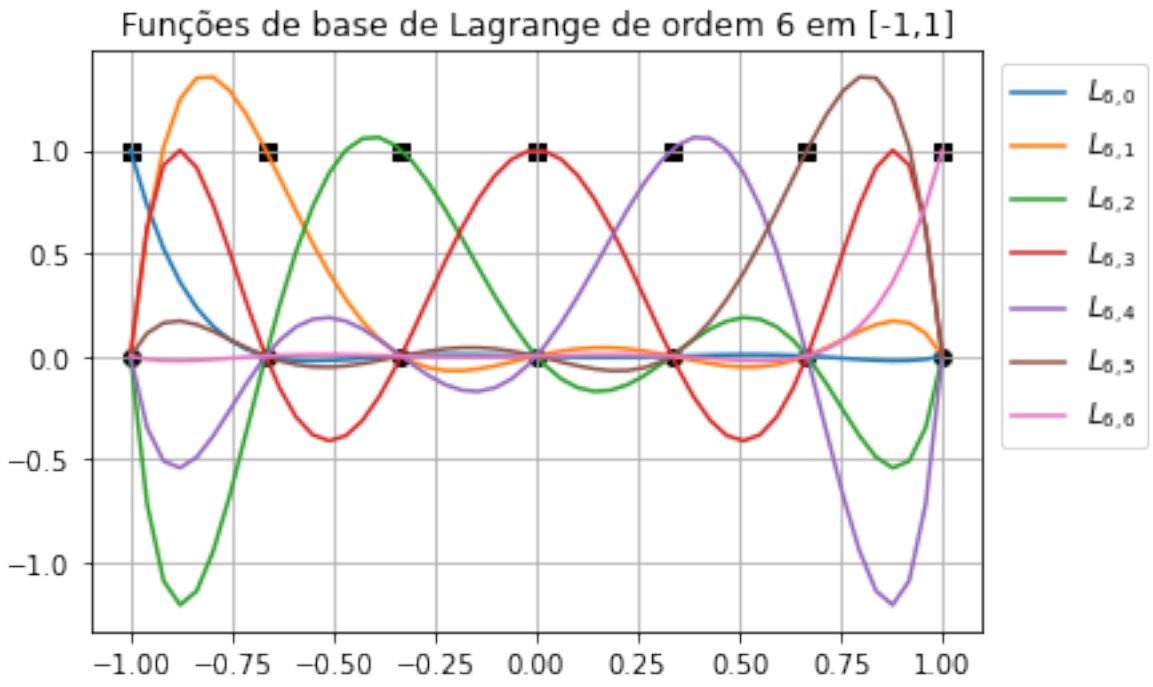
```
from sympy import Symbol

def symbolic_vector(n,var):
    """Cria uma lista com n variáveis simbólicas.

        entrada:
            n: numero de pontos
            var: uma string (ex. 'x')
        saida:
            V: ['var0','var1',...,'varn-1']
    """
    if not isinstance(var,str):
        raise TypeError(f'{var} must be a string.')

    V = [Symbol(var + str(i)) for i in range(0,n)]

    return V

def L_nj(X,j):
    """ Calcula a função de base de Lagrange L_{n,j}(x).

        entrada:
            X: um vetor contendo variáveis simbólicas
    """
    # pega a variavel base passada e converte para simbólica
    x = str(X[1])
    x = Symbol(x[0:-1])
    L = 1.0
    for i in range(0,len(X)):
        if i != j:
            L *= (x - X[i])/(X[j] - X[i])

    return L
```

```
import numpy as np
import matplotlib.pyplot as plt

# número de nós de interpolação: interpolação de (n-1)-ésimo grau
n = 7

# domínio de interpolação
x0,x1 = -1,1
```

(continues on next page)

(continued from previous page)

```
# constroi vetor simbolico em x
X = symbolic_vector(n,'x')

# constroi pontos numericos
xp = np.linspace(x0,x1,num=n,endpoint=True)

# cria malha numérica
xv = np.linspace(x0,x1)

# matriz das funções
Y = np.zeros((n,len(xv)))
for i in range(0,n):
    Y[i,] = np.zeros(np.shape(xv))

# montagem de dict para substituição: [xk,x0,x1,x2,...]
k = [str(i) for i in X]
k.insert(0,'x')

# preenche matriz
for i in range(0,Y.shape[0]):
    for j in range(0,np.size(xv)):
        v = list(np.concatenate([np.asarray([xv[j]]),xp]))
        d = dict(zip(k,v))
        Y[i,j] = L_nj(X,i).subs(d)

# plotagem das funções

leg = []
for i in range(0,Y.shape[0]):
    plt.plot(xv,Y[i,])
    s = '$L_{' + str(n-1) + ',' + str(i) + '}$'
    leg.append(s)

plt.grid()

# nós
plt.scatter(xp,np.zeros(xp.shape),c= 'k')
plt.scatter(xp,np.ones(xp.shape),c='k',marker='s')


plt.legend(leg,loc='best',bbox_to_anchor=(0.7, 0.5, 0.5, 0.5))
plt.title('Funções de base de Lagrange de ordem ' + str(n-1) + ' em ['+str(x0)+',
 ↪'+str(x1)+']');
```


Funções de base de Lagrange de ordem 6 em [-1,1]
1.0
0.5
0.0
−0.5
−1.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
$L_{6,0}$
$L_{6,1}$
$L_{6,2}$
$L_{6,3}$
$L_{6,4}$
$L_{6,5}$
$L_{6,6}$

CHAPTER

TWENTYSIX

POLINÔMIO INTERPOLADOR DE NEWTON (DIFERENÇAS DIVIDIDAS)

26.1 Interpolação Linear

Excluindo-se o caso de função constante, a forma mais simples de interpolação é ligar dois pontos dados com uma reta. Usando semelhança de triângulos entre nós e valores de função, obtemos

$$\frac{f_1(x) - f(x_0)}{x - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

a qual pode ser reorganizada para fornecer

$$f_1(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0) \qquad (1)$$

A notação $f_1(x)$ indica que esse é um polinômio interpolador de primeiro grau. Observe que, além de representar a inclinação da reta ligando os pontos, o termo $[f(x_1) - f(x_0)]/(x_1 - x_0)$ é uma aproximação por diferenças divididas da primeira derivada.

26.2 Interpolação Quadrática

Com três pontos, a interpolação quadrática é obtida a partir de

$$f_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) \qquad (2)$$

Um procedimento simples pode ser usado para determinar os valores dos coeficientes.

Para b_0, a Equação (2) com $x = x_0$ pode ser usada para calcular

$$b_0 = f(x_0) \qquad (3)$$

A Equação (3) pode ser substituída na Equação (2), a qual pode ser calculada em $x = x_1$ para

$$b_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \qquad (4)$$

Finalmente, as Equações (3) e (4) podem ser substituídas na Equação (2), a qual pode ser calculada em $x = x_2$ e resolvida (depois de algumas manipulações algébricas) por

$$b_2 = \frac{\dfrac{f(x_2) - f(x_1)}{x_2 - x_1} - \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0} \qquad (5)$$

26.3 Forma Geral dos Polinômios Interpoladores de Newton

A análise anterior pode ser generalizada para ajustar um polinômio de grau n a $n+1$ pontos dados. O polinômio de grau n é

$$f_n(x) = b_0 + b_1(x - x_0) + \cdots + b_n(x - x_0)(x - x_1)\ldots(x - x_{n-1}) \qquad (6)$$

Como foi feito anteriormente com as interpolações linear e quadrática, os pontos dados podem ser usados para calcular os coeficientes b_0, b_1, \dots , b_n. Para um polinômio de grau n, $n + 1$ pontos dados são necessários: $(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))$. Usamos esses pontos dados e as seguintes equações para calcular os coeficientes

$$\begin{aligned} b_0 &= f(x_0) && (7)\\ b_1 &= f[x_1, x_0] && (8)\\ b_2 &= f[x_2, x_1, x_0] && (9)\\ &\vdots\\ b_n &= f[x_n, x_{n-1}, \ldots, x_1, x_0] && (10) \end{aligned}$$

onde a função com colchetes corresponde a diferenças divididas. Por exemplo, a primeira diferença dividida finita é representada em geral por

$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j}$$

A segunda diferença dividida finita, que representa a diferença das duas primeiras diferenças divididas, é expressa em geral por

$$f[x_i, x_j, x_k] = \frac{f[x_i, x_j] - f[x_j, x_k]}{x_i - x_k}$$

Analogamente, a n-ésima diferença dividida é

$$f[x_n, x_{n-1}, \ldots, x_1, x_0] = \frac{f[x_n, x_{n-1}, \ldots, x_1] - f[x_{n-1}, x_{n-2}, \ldots, x_0]}{x_n - x_0}$$

26.4 Funções de Base de Newton

Código gerador de funções de base de Newton de grau n por computação simbólica.

```
from sympy import Symbol

def symbolic_vector(n,var):
    """Cria uma lista com n variáveis simbólicas.

        entrada:
            n: numero de pontos
            var: uma string (ex. 'x')
        saida:
            V: ['var0','var1',...,'varn-1']
    """
    if not isinstance(var,str):
        raise TypeError("{0} must be a string.".format(var))
```

(continues on next page)

(continued from previous page)

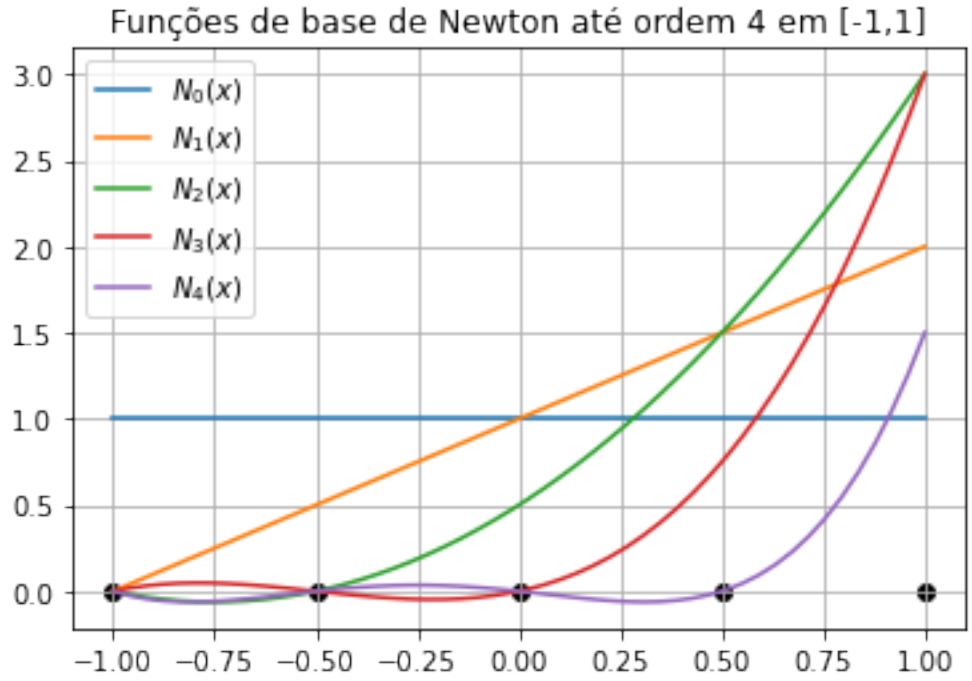
```
    V = [Symbol(var  + str(i)) for i in range(0,n)]

    return V

def N_nj(X,j):
    """ Calcula a função de base de Newton N_{n,j}(x).

        entrada:
            X: um vetor contendo variáveis simbólicas
    """
    # pega a variavel base passada e converte para simbólica
    x = X[1]
    x = str(x)
    x = Symbol(x[0:-1])
    N = x/x;
    if j > 0:
        for k in range(0,j):
            N *= (x - X[k])

    return N
```

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# número de nós de interpolação: interpolação de (n-1)-ésimo grau
n = 5

# domínio de interpolação
x0,x1 = -1,1

# constroi vetor simbolico em x
X = symbolic_vector(n,'x')

# constroi pontos numericos
xp = np.linspace(x0,x1,num=n,endpoint=True)

# cria malha numérica
xv = np.linspace(x0,x1)

# matriz das funções
Y = np.zeros((n,len(xv)))
for i in range(0,n):
    Y[i,] = np.zeros(np.shape(xv))

# montagem de dict para substituição: [xk,x0,x1,x2,...]
k = [str(i) for i in X]
k.insert(0,'x')

# preenche matriz
for i in range(0,Y.shape[0]):
    for j in range(0,np.size(xv)):
        v = list(np.concatenate([np.asarray([xv[j]]),xp]))
        d = dict(zip(k,v))
        Y[i,j] = N_nj(X,i).subs(d)
```

(continues on next page)

(continued from previous page)

```
# plotagem das funções

# nós
plt.scatter(xp,np.zeros(xp.shape),c= 'k')

leg = []
for i in range(0,Y.shape[0]):
    plt.plot(xv,Y[i,])
    s = '$N_{' + str(i) + '}(x)$'
    leg.append(s)

plt.grid()
plt.legend(leg,loc='best')
plt.title('Funções de base de Newton até ordem ' + str(n-1) + ' em ['+str(x0)+',
 ↪'+str(x1)+']');
```



26.4.1 Exemplo:

Encontre o polinômio interpolador de Newton de ordem 1 $P_1(x)$ para a tabela abaixo

x	y
-1	4
0	1

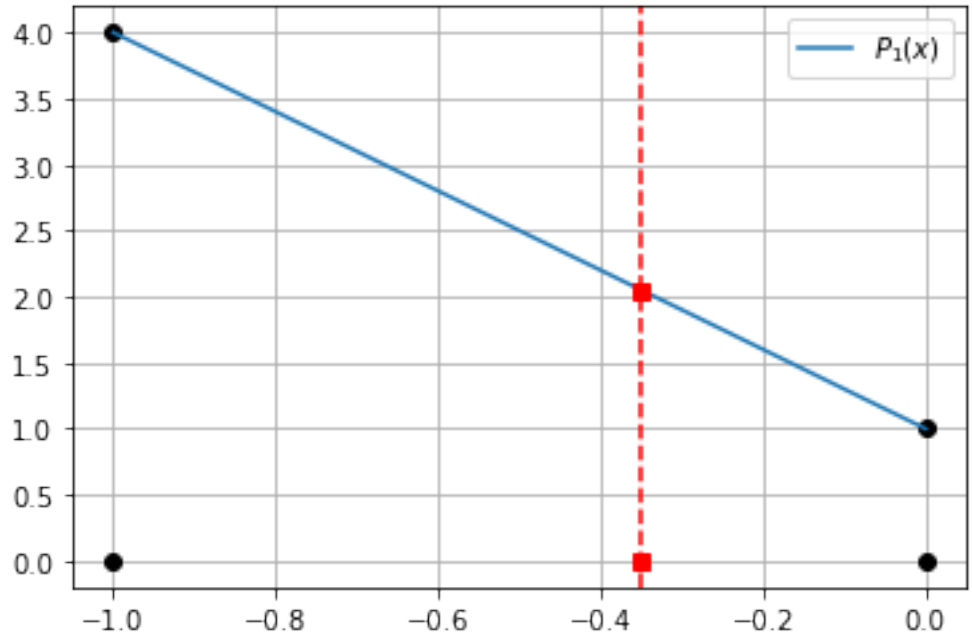
Compute o valor de $P_1(0.35)$.

```
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# interpolação linear

# coeficiente a0 = y0
# coeficiente a1 = (y1-y0)/(x1-x0)

# pontos
x0,y0 = -1,4
x1,y1 =  0,1

# ordem 0
a0 = y0

# interpolador de Newton
a1 = (y1-y0)/(x1-x0)
P1 = lambda x: a0 + a1*(x-x0)

# ponto interpolado
xp = -0.35
yp = P1(xp)
yp

# plotagem

# nós
plt.plot([x0,x1],[0,0],'ok')

# valores nodais
plt.plot([x0,x1],[y0,y1],'ok')

# interpolador
x = np.linspace(x0,x1,30,endpoint=True)
plt.plot(x,P1(x),label='$P_1(x)$')

# ponto interpolado
plt.plot(xp,0,'sr')
plt.plot(xp,yp,'sr')
plt.axvline(xp,0,yp,c='r',ls='dashed')

plt.grid()
plt.legend(loc='best');
```




```
P1(xp)
```

```
2.05
```

26.4.2 Exemplo:

Encontre o polinômio interpolador de Newton de ordem 2 $P_2(x)$ para a tabela abaixo

x	y
-1	4
0	1
2	-1

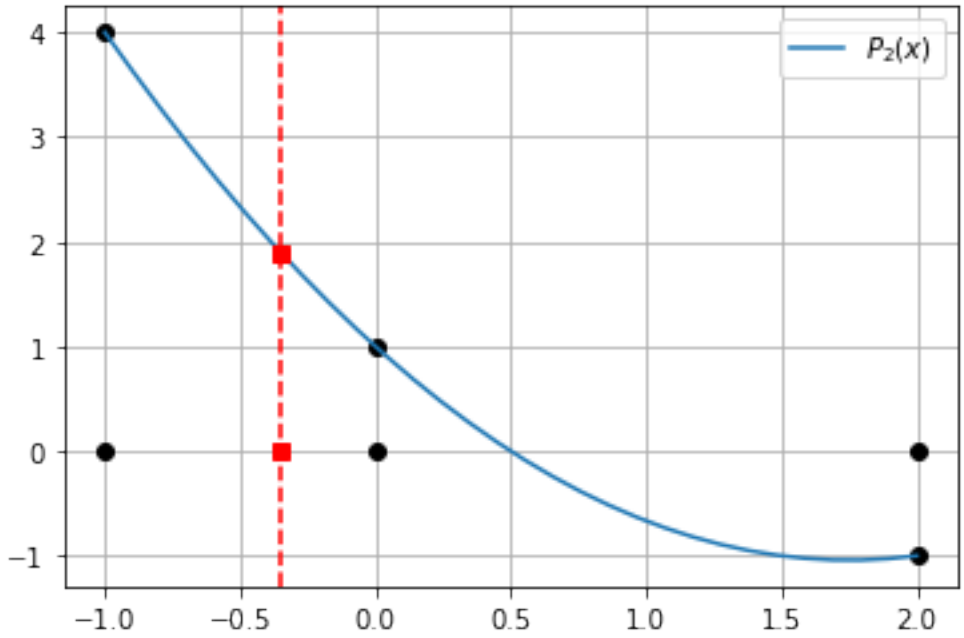
Compute o valor de $P_2(0.35)$.

```
# interpolação quadrática

# Usando tabela DD:
# https://vnicius.github.io/numbiosis/interpolador-newton/index.html


# par adicional
x2,y2 = 2.,-1.

# coeficiente a2 = f[x0,x1,x2] = ( f[x1,x2] - f[x0,x1] ) / (x2 - x0)
# a2 = ( (y2-y1)/(x2-x1) - (y1-y0)/(x1-x0) )/(x2-x0)*(xx-x0)*(xx-x1)

# interpolador de Newton
P2 = lambda xx: P1(xx) + ( ( (y2-y1)/(x2-x1) - (y1-y0)/(x1-x0) )/(x2-x0) )*(xx-
↪x0)*(xx-x1)
```

(continues on next page)

(continued from previous page)

```
# ponto interpolado
yp = P2(xp)
yp

# plotagem

# nós
plt.plot([x0,x1,x2],[0,0,0],'ok')

# valores nodais
plt.plot([x0,x1,x2],[y0,y1,y2],'ok')

# interpolador
x = np.linspace(x0,x2,30,endpoint=True)
plt.plot(x,P2(x),label='$P_2(x)$')

# ponto interpolado
plt.plot(xp,0,'sr')
plt.plot(xp,yp,'sr')
plt.axvline(xp,0,yp,c='r',ls='dashed')

plt.grid()
plt.legend(loc='best');
```




```
P2(xp)
```

```
1.8983333333333332
```

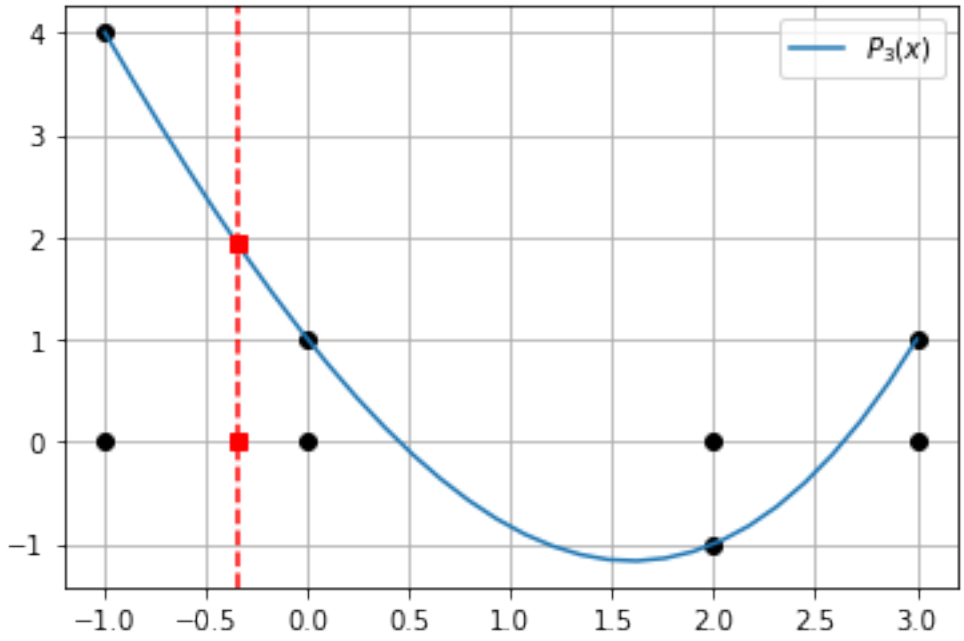
26.4.3 Exemplo:

Encontre o polinômio interpolador de Newton de ordem 3 $P_3(x)$ para a tabela abaixo

x	y
-1	4
0	1
2	-1
3	1

Compute o valor de $P_3(0.35)$.

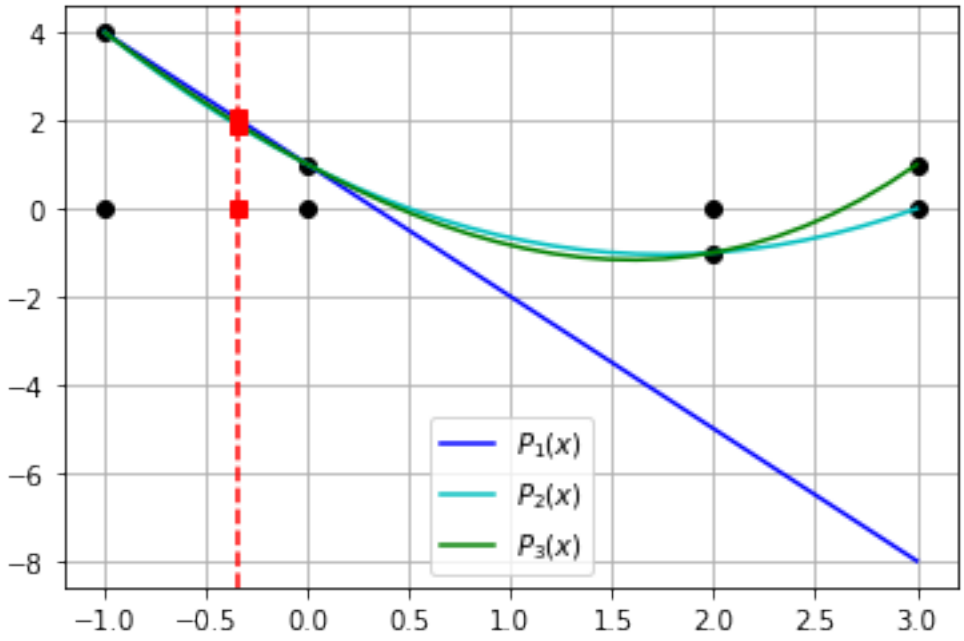
```
# interpolação quadrática

# Usando tabela DD:
# https://vnicius.github.io/numbiosis/interpolador-newton/index.html


# par adicional
x3,y3 = 3.,1.

# coeficiente a3 = f[x0,x1,x2,x3]

# interpolador de Newton
P3 = lambda xxx: P2(xxx) + 1/12*(xxx-x0)*(xxx-x1)*(xxx-x2)


# ponto interpolado
yp = P3(xp)
yp

# plotagem

# nós
plt.plot([x0,x1,x2,x3],[0,0,0,0],'ok')

# valores nodais
plt.plot([x0,x1,x2,x3],[y0,y1,y2,y3],'ok')

# interpolador
x = np.linspace(x0,x3,30,endpoint=True)
plt.plot(x,P3(x),label='$P_3(x)$')

# ponto interpolado
plt.plot(xp,0,'sr')
plt.plot(xp,yp,'sr')
plt.axvline(xp,0,yp,c='r',ls='dashed')

plt.grid()
plt.legend(loc='best');
```




```
P3(-0.35)
```

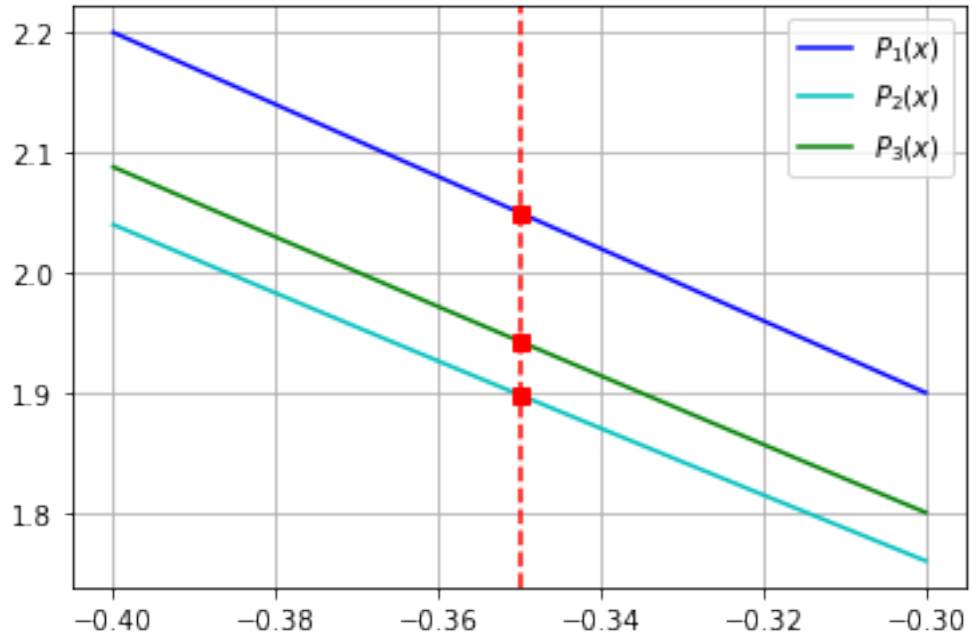
```
1.9428854166666665
```

```
# ponto interpolado
YP = [P1(xp),P2(xp),P3(xp)]

# plotagem
# nós
plt.plot([x0,x1,x2,x3],[0,0,0,0],'ok')

# valores nodais
plt.plot([x0,x1,x2,x3],[y0,y1,y2,y3],'ok')

# interpoladores
x = np.linspace(x0,x3,30,endpoint=True)
plt.plot(x,P1(x),'b',label='$P_1(x)$')
plt.plot(x,P2(x),'c',label='$P_2(x)$')
plt.plot(x,P3(x),'g',label='$P_3(x)$')

# ponto interpolado
plt.plot(xp,0,'sr')
plt.plot([xp,xp,xp],YP,'sr')
plt.axvline(xp,0,max(YP),c='r',ls='dashed')

plt.grid()
plt.legend(loc='best');
```



26.5 Comparação (zoom)

```
# interpoladores
x = np.linspace(-0.4,-0.3,30,endpoint=True)
plt.plot(x,P1(x),'b',label='$P_1(x)$')
plt.plot(x,P2(x),'c',label='$P_2(x)$')
plt.plot(x,P3(x),'g',label='$P_3(x)$')

# ponto interpolado
plt.plot([xp,xp,xp],YP,'sr')
plt.axvline(xp,0,max(YP),c='r',ls='dashed')

plt.grid()
plt.legend(loc='best');
```


2.2
2.1
2.0
1.9
1.8
−0.40
−0.38
−0.36
−0.34
−0.32
−0.30
$P_1(x)$
$P_2(x)$
$P_3(x)$

CHAPTER

TWENTYSEVEN

AJUSTE DE CURVAS POR MÍNIMOS QUADRADOS

27.1 Regressão Linear

O exemplo mais simples de aproximação por mínimos quadrados é ajustar uma reta a um conjunto de pares de observações $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. A expressão matemática do ajuste por uma reta é

$$y = a_0 + a_1 x + e \qquad (1)$$

onde a_0 e a_1 são coeficientes representando a intersecção com o eixo y e a inclinação, respectivamente, e e é o erro ou resíduo entre o modelo e a observação, o qual pode ser representado, depois de se reorganizar a equação acima, por

$$e = y - a_0 - a_1 x$$

Portanto, o erro ou resíduo é a discrepância entre o valor verdadeiro de y e o valor aproximado, $a_0 + a_1 x$, previsto pela equação linear.

27.1.1 Critério para um Melhor Ajuste

Uma estratégia para ajustar uma melhor reta pelos dados é minimizar a soma dos quadrados dos resíduos entre o y medido e o y calculado com o modelo linear

$$S_r = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_{i,medido} - y_{i,modelo})^2 = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i)^2 \qquad (2)$$

Esse critério tem diversas vantagens, incluindo o fato de que ele fornece uma única reta para um dado conjunto de dados.

27.1.2 Ajuste por Mínimos Quadrados por uma Reta

Para determinar os valores de a_0 e a_1, a Equação (2) é derivada com relação a cada coeficiente

$$\frac{\partial S_r}{\partial a_0} = -2 \sum (y_i - a_0 - a_1 x_i)$$
$$\frac{\partial S_r}{\partial a_1} = -2 \sum [(y_i - a_0 - a_1 x_i) x_i]$$

Igualando essas derivadas a zero será obtido um S_r mínimo

$$0 = \sum y_i - \sum a_0 - \sum a_1 x_i$$
$$0 = \sum y_i x_i - \sum a_0 x_i - \sum a_1 x_i^2$$

Agora, pode-se expressar essas equações como um conjunto de duas equações lineares simultâneas em duas variáveis (a_0 e a_1)

$$na_0 + \left(\sum x_i\right) a_1 = \sum y_i$$
$$\left(\sum x_i\right) a_0 + \left(\sum x_i^2\right) a_1 = \sum x_i y_i$$

Essas são as chamadas equações normais. Elas podem ser resolvidas simultaneamente para obter-se

$$a_1 = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2} \qquad (3)$$
$$a_0 = \overline{y} - a_1 \overline{x} \qquad (4)$$

onde $\overline{y}$ e $\overline{x}$ são as médias de y e x, respectivamente.

27.1.3 Quantificação do Erro da Regressão Linear

Um "desvio padrão" para a reta de regressão pode ser determinado por

$$s_{y/x} = \sqrt{\frac{S_r}{n-2}}$$

onde $s_{y/x}$ é chamado erro padrão da estimativa. O subscrito "y/x" indica que o erro é para um valor previsto de y correspondente a um valor particular de x. Além disso, observe que agora estamos dividindo por $n-2$ porque duas estimativas provenientes dos dados —a_0 e a_1 — foram usadas para calcular S_r; portanto, perdemos dois graus de liberdade.

Exatamente como no caso do desvio padrão, o erro padrão da estimativa quantifica a dispersão dos dados. Entretanto, erro padrão da estimativa quantifica a dispersão em torno da reta de regressão, em contraste com o desvio padrão original que quantificava a dispersão em torno da média.

A fim de quantificar o "quão bom" é um ajuste de reta, dois conceitos utilizados são os coeficiente de determinação (r^2) e o coeficiente de correlação ($r = \sqrt{r^2}$).

$$r^2 = \frac{S_t - S_r}{S_t}$$

$$S_t = \sum_{i=1}^{n} (y_i - \overline{y})^2$$

Para um ajuste perfeito, $S_r = 0$ e $r = r_2 = 1$, significando que a reta explica 100% da variação dos dados. Para $r = r_2 = 0$, $S_r = St$ e o ajuste não representa nenhuma melhora. Uma formulação alternativa para r que é mais conveniente para implementação computacional é

$$r = \frac{n\sum x_i y_i - (\sum x_i)(\sum y_i)}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}} \qquad (5)$$

27.2 Regressão Polinomial

O procedimento dos mínimos quadrados pode ser prontamente estendido para ajustar dados por um polinômio de grau mais alto. Por exemplo, suponha que se queira ajustar um polinômio de segundo grau ou quadrático

$$y = a_0 + a_1 x + a_2 x_2 + e \qquad (6)$$

Nesse caso, a soma dos quadrados dos resíduos é

$$S_r = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i - a_2 x_i^2)^2 \qquad (7)$$

Seguindo um procedimento análogo ao anterior, toma-se a derivada da Equação (7) com relação a cada um dos coeficientes desconhecidos do polinômio, como em

$$\frac{\partial S_r}{\partial a_0} = -2\sum(y_i - a_0 - a_1 x_i - a_2 x_i^2)$$
$$\frac{\partial S_r}{\partial a_1} = -2\sum[(y_i - a_0 - a_1 x_i - a_2 x_i^2)x_i]$$
$$\frac{\partial S_r}{\partial a_2} = -2\sum[(y_i - a_0 - a_1 x_i - a_2 x_i^2)x_i^2]$$

Essas equações podem ser igualadas a zero e reorganizadas para determinar o seguinte conjunto de equações normais

$$na_0 + \left(\sum x_i\right) a_1 + \left(\sum x_i^2\right) a_2 = \sum y_i$$
$$\left(\sum x_i\right) a_0 + \left(\sum x_i^2\right) a_1 + \left(\sum x_i^3\right) a_2 = \sum x_i y_i$$
$$\left(\sum x_i^2\right) a_0 + \left(\sum x_i^3\right) a_1 + \left(\sum x_i^4\right) a_2 = \sum x_i^2 y_i$$

Nesse caso, vê-se que o problema de determinar o polinômio de segundo grau por mínimos quadrados é equivalente a resolver um sistema de três equações lineares simultâneas. Na forma matricial, temos

$$\begin{bmatrix} n & \sum x_i & \sum x_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \end{bmatrix} \qquad (8)$$

O caso bidimensional pode ser facilmente estendido para um polinômio de grau m.

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m + e$$

27.3 Nota

O presente texto (conteúdo teórico) é um resumo baseado no livro Métodos Numéricos para Engenharia (Chapra e Canale).

27.4 Motivação: Evolução da População Paraibana

Este exemplo lê um arquivo CSV, plota o gráfico de dispersão, compara modelos e formata os dados para visualização.

```
import pandas as pd
from bokeh.io import output_notebook
from bokeh.plotting import figure, show
from bokeh.models import PrintfTickFormatter, Range1d

from scipy import stats
from scipy import interpolate

# tabela de dados
df = pd.read_csv("file-pop-pb.csv")
x = df['ano']
y = df['pop. urbana']

# grafico de dispersao
f = figure(plot_width=400, plot_height=400)
```

(continues on next page)

(continued from previous page)

```
f.scatter(x,y,fill_color="blue",radius=1,alpha=1)

# formatacao
f.background_fill_color = "blue"
f.background_fill_alpha = 0.02
f.x_range=Range1d(1960, 2020)
e = 5.e5
f.y_range=Range1d(y.min() - e,y.max() + e)
f.xaxis.axis_label = "Ano"
f.yaxis.axis_label = "Pop. urbana - PB"
f.yaxis[0].formatter = PrintfTickFormatter(format="%1.2e")

# interpolacao

f.line(x,y,color='green')

# estimando o valor da populacao em 1975 e 1982 por interpolacao
p = interpolate.interp1d(x, y)
xx = [1975,1982]
yy = p(xx)
print("População obtida por interpolação linear em: 1975 = {:d}; 1982 = {:d}; ".
 ↪format(int(yy[0]),int(yy[1])))
f.square(xx,yy,color='black',line_width=4)

# ajuste por minimos quadrados

slope, intercept, r_value, p_value, std_err = stats.linregress(x,y)
px = intercept + slope*x
f.line(x,px,color='red')

output_notebook()
show(f)
```

```
População obtida por interpolação linear em: 1975 = 1225580; 1982 = 1552017;
```

```
df.set_index('ano')
```

```
      pop. urbana
ano
1970      1002156
1980      1449004
1991      2015576
2000      2447212
2010      2902044
```

27.5 Exemplo

Vamos resolver um problema de regressão linear por meio das equações normais passo a passo. Em primeiro lugar, importemos os módulos de computação numérica e de plotagem.

```
# importação de módulos
import numpy as np
import matplotlib.pyplot as plt
import sympy as sy
%matplotlib inline
```

Consideremos a simples tabela abaixo de um experimento fictício que busca correlacionar densidade e diâmetro médio de grãos em alimentos.

densidade	diâmetro médio de grão
1.0	0.5
2.1	2.5
3.3	2.0
4.5	4.2

Vamos escrever os dados como *arrays*.

```
# tabela de dados
x = np.array([1,2,3,4]) # densidade
y = np.array([0.5,2.5,2.0,4.0]) # diâmetro
```

Agora, calculamos os coeficientes linear α_0 e angular α_1 pelas fórmulas das equações normais vistas em aula.

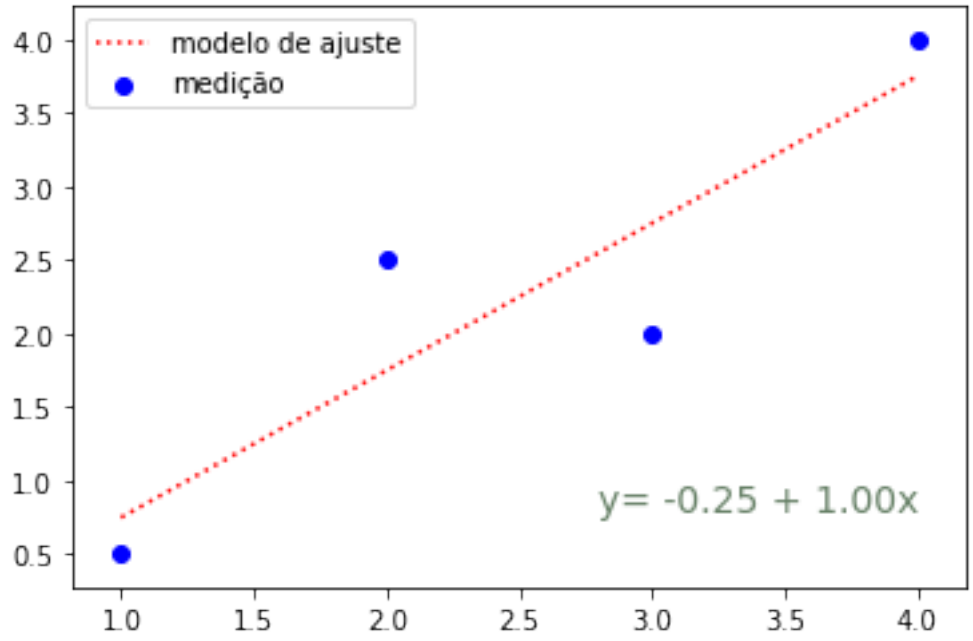
```
m = np.size(x)
alpha1 = (m*np.dot(x,y) - np.sum(x)*np.sum(y))/(m*np.dot(x,x)-np.sum(x)**2)
alpha0 = np.mean(y) - alpha1*np.mean(x)
```

Podemos agora escrever a equação da reta de regressão usando o *array* `x` como abscissa. Este será o nosso *modelo de ajuste*.

```
y2 = alpha0 + alpha1*x
```

Enfim, plotamos o gráfico de dispersão dos valores *medidos* juntamente com o *modelo de ajuste* da seguinte forma:

```
mod = plt.plot(x,y2,'r:'); # modelo
med = plt.scatter(x,y,c='b'); # medição
plt.legend({'modelo de ajuste':mod, 'medição':med}); # legenda

# esta linha adiciona a equação de ajuste ao gráfico na posição (x,y) = (2.8,0.8)
# com fonte tamanho 14 e cor RGB = [0.4,0.5,0.4].
plt.annotate('y= {0:.2f} + {1:.2f}x'.format(alpha0,alpha1),(2.8,0.8),fontsize=14,c=[0.
 ↪4,0.5,0.4]);
```



Na prática, podemos calcular regressão linear usando o módulo `scipy.stats`. Vide *Code session 7*.

CHAPTER

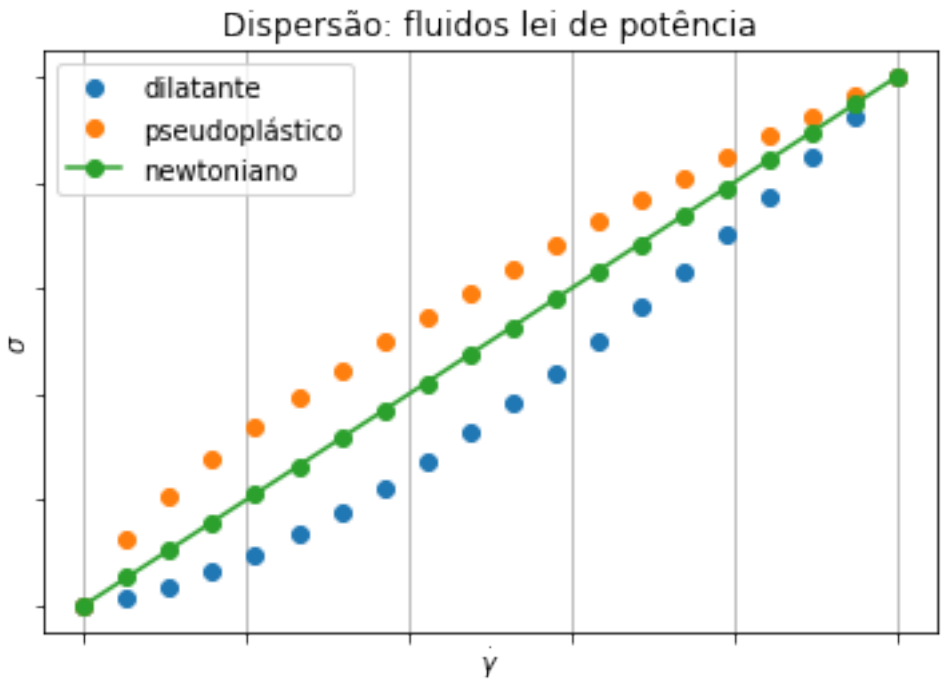
TWENTYEIGHT

AJUSTE DE CURVAS: CASO NÃO-LINEAR

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

28.1 Motivação: comportamento de fluidos lei de potência

```
gamma = np.linspace(0,1,20,True)
n1 = 1.5 # dilatante
n2 = 0.7 # pseudoplastico
k = 0.1
tau1 = k*gamma**n1
tau2 = k*gamma**n2
tau3 = k*gamma

plt.plot(gamma,tau1,'o',label='dilatante')
plt.plot(gamma,tau2,'o',label='pseudoplástico')
plt.plot(gamma,tau3,'-o',label='newtoniano')
plt.legend()
plt.xlabel('$\dot{\gamma}$')
plt.ylabel('$\sigma}$')
plt.grid(axis='x')
plt.title('Dispersão: fluidos lei de potência')
plt.tick_params(axis='both',which='both',labelbottom=False,labelleft=False)
```



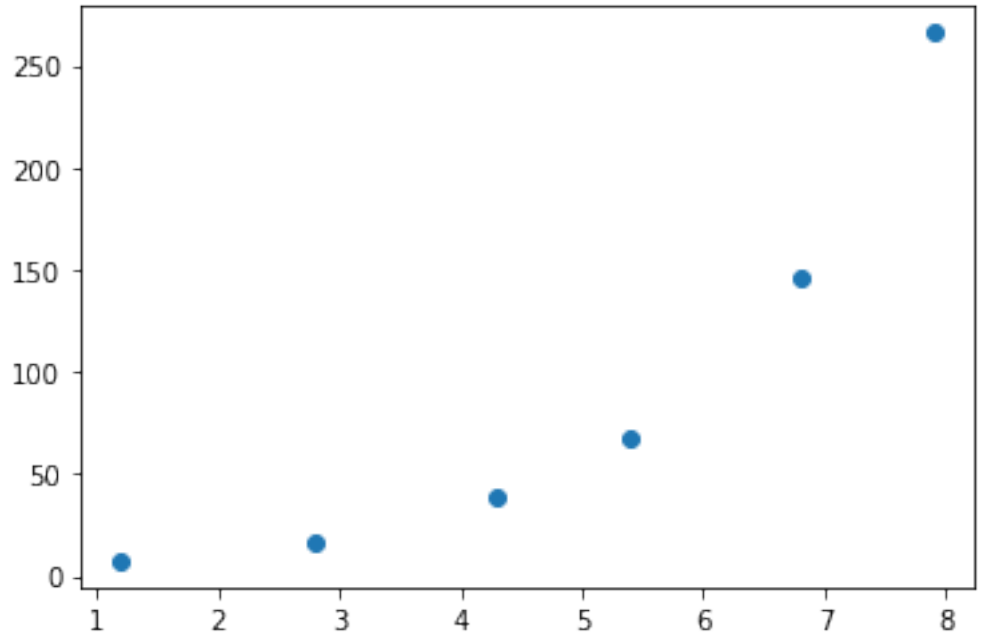
28.2 Exemplo

Determine os parâmetros a e b de modo que $y = f(x) = ae^{bx}$ ajuste os seguintes dados no sentido de mínimos quadrados:

x	y
1.2	7.5
2.8	16.1
4.3	38.9
5.4	67.0
6.8	146.6
7.9	266.2

Plotando o gráfico de dispersão:

```
x = np.array([1.2,2.8,4.3,5.4,6.8,7.9])
y = np.array([7.5,16.1,38.9,67.0,146.6,266.2])
plt.plot(x,y,'o');
```



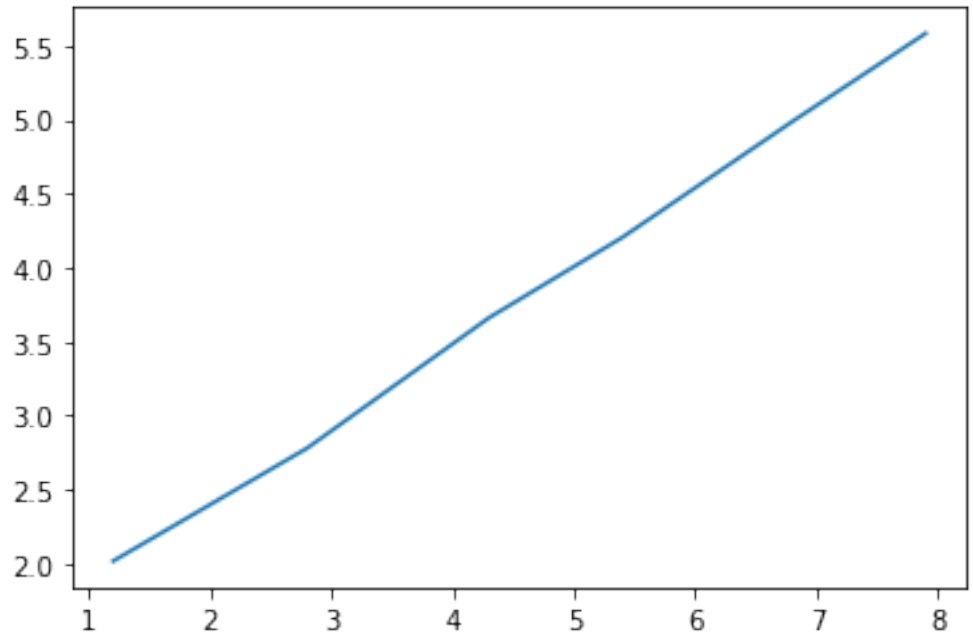
Teste de alinhamento: façamos a linearização

$$Y = Z + bx,$$

onde $Y = \log(y)$ e $Z = \log(a)$.

Plotemos a dispersão (x, Y).

```
Y = np.log(y)
plt.plot(x,Y);
```



O teste do alinhamento nos diz que a função de ajuste é adequada para a regressão linear.

Regressão linear: façamos a regressão linear para buscar os parâmetros b e Z do modelo linearizado.

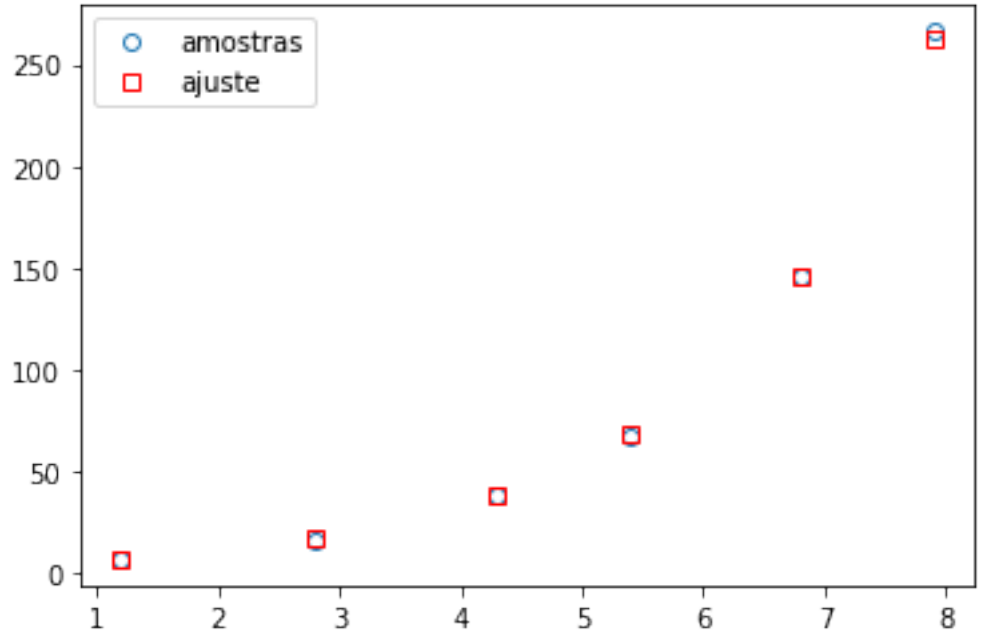
```
from scipy.stats import linregress

b,Z,R,p,e = linregress(x,Y)
print(f'Inclinação = {b:.3f}; offset = {Z:.3f}; R2 = {R*R:.3f}.')
```

```
Inclinação = 0.537; offset = 1.332; R2 = 0.999.
```

De fato, o coeficiente $R^2 \approx 1.0$ mostra correlação quase máxima no modelo linearizado.

Comparação entre dados experimentais e ajustados: plotaremos agora a dispersão e o modelo ajustado.

```
a = np.exp(Z) # recupera o valor de a
modelo = lambda x: a*np.exp(b*x)

plt.plot(x,y,'o',mfc='None')
plt.plot(x,modelo(x),'rs',mfc='None');
plt.legend(('amostras','ajuste'));
```



Estimando valores não tabelados: visto que o modelo exponencial se acomoda bem aos dados experimentais, agora podemos estimar valores que são desconhecidos, tais como $x = 3.2$ ou $x = 7.5$.

```
xp = np.array([3.2,7.5]) # valores procurados

for p in xp:
    print(f'Em x = {p:g}, a estimativa é f(x) = {modelo(p):g}')
```

```
Em x = 3.2, a estimativa é f(x) = 21.097
Em x = 7.5, a estimativa é f(x) = 211.969
```

28.2.1 Exercício

A seguinte tabela mostra a variação de condutividade térmica relativa k de sódio com a temperatura T em graus Celsius. Busque um modelo não-linear que ajusta os dados no sentido de mínimos quadrados.

k	T
1.00	79
0.932	190
0.839	357
0.759	524
0.693	690

CHAPTER

TWENTYNINE

INTEGRAÇÃO NUMÉRICA: REGRAS DE NEWTON-COTES

As fórmulas de Newton-Cotes são os esquemas mais comuns de integração numérica. Elas são baseadas na estratégia de substituir uma função complicada ou dados tabulados por uma função aproximadora simples que seja fácil de integrar:

$$I = \int_a^b f(x)dx \cong \int_a^b f_n(x)dx \qquad (1)$$

em que $f_n(x)$ é um polinômio da forma

$$f_n(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$$

em que n é o grau do polinômio.

29.1 A Regra do Trapézio

A **regra do trapézio** corresponde ao caso no qual o polinômio na Equação (1) é de primeiro grau:

$$I = \int_a^b f(x)dx \cong \int_a^b f_1(x)dx$$

onde

$$f_1(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a)$$

O resultado da integração é

$$I = (b - a)\frac{f(a) + f(b)}{2} \qquad (2)$$

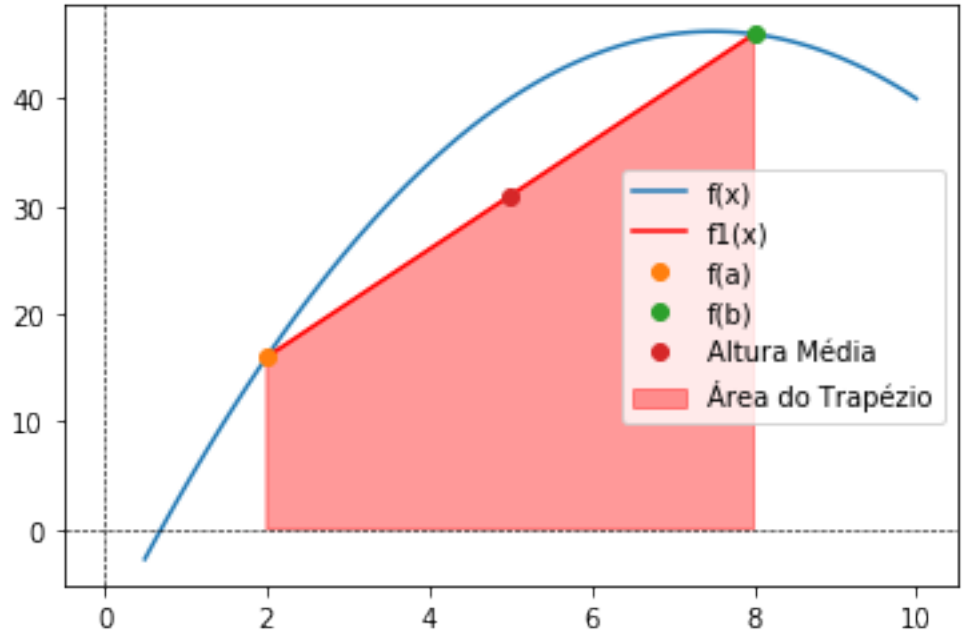
Geometricamente, a regra dos trapézios é equivalente a aproximar a integral pela área do trapézio sob a reta ligando $f(a)$ e $f(b)$ (conforme figura abaixo). Portanto, a estimativa da integral pode ser representada por

$$I \cong \text{largura} \times \text{altura média}$$
$$I \cong (b - a) \times \text{altura média}$$

em que, para a regra dos trapézios, a altura média é a média dos valores da função nas extremidades, ou $[f(a) + f(b)]/2$, conforme Equação (2).

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sy
import scipy as sp
from scipy import interpolate
from scipy import integrate
%matplotlib inline
```

```
# Representação geométrica da regra do trapézio

def f(x):
    return -x**2 + 15*x - 10

x = np.linspace(0.5,10,100)
y = f(x)

plt.plot(x,y, label = 'f(x)')
plt.plot([2,8], [f(2),f(8)], 'r', label = 'f1(x)')
plt.plot(2,f(2), 'o', label = 'f(a)')
plt.plot(8,f(8), 'o', label = 'f(b)')
plt.plot(np.mean([2,8]), np.mean([f(2),f(8)]), 'o', label = 'Altura Média')
plt.fill([2, 2, 8, 8],[0, f(2), f(8), 0],color='r',alpha=0.4, label = 'Área do␣
 ↪Trapézio')
plt.legend()
plt.axvline(x=0,color='k',linewidth=0.6,linestyle='--')
plt.axhline(y=0,color='k',linewidth=0.6,linestyle='--');
```



29.2 Aplicação Múltipla da Regra do Trapézio

Uma maneira de melhorar a acurácia da regra do trapézio é dividir o intervalo de integração de a a b em diversos segmentos e aplicar o método a cada segmento (conforme figura abaixo). As áreas correspondentes aos segmentos individuais podem então ser somadas parafornecer a integral para o intervalo inteiro. As equações resultantes são chamadas fórmulas de integração por aplicações múltiplas ou compostas.

Existem $n + 1$ pontos base igualmente espaçados $(x_0, x_1, x_2, \ldots, x_n)$. Consequentemente, existem n segmentos de largura igual:

$$h = \frac{b - a}{n} \qquad (3)$$

Se a e b forem designados por x_0 e x_n, respectivamente, a integral total pode ser representada como

$$I = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx$$

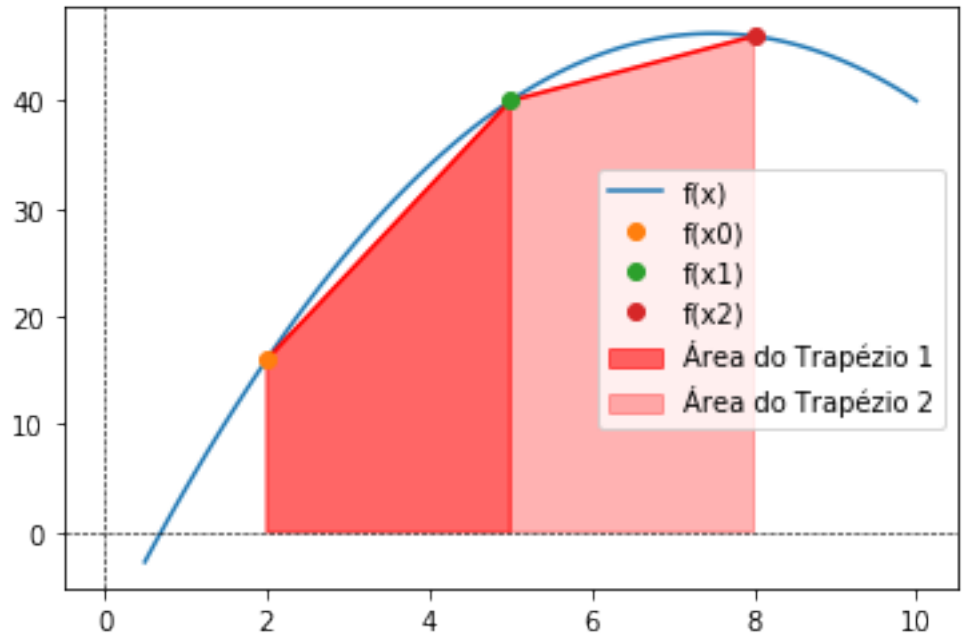
Substituindo cada integral pela regra do trapézio, obtém-se

$$I = h\frac{f(x_0) + f(x_1)}{2} + h\frac{f(x_1) + f(x_2)}{2} + \cdots + h\frac{f(x_{n-1}) + f(x_n)}{2}$$

ou, agrupando termos,

$$I = \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right] \qquad (4)$$

```
# Representação geométrica da aplicação múltipla da regra do trapézio
# É possível observar, visualmente, que houve uma redução do erro do resultado em comparação com o caso anterior

xd = np.array([2,5,8])
yd = f(xd)

plt.plot(x,y, label = 'f(x)')
plt.plot(xd,yd, 'r')
plt.plot(2,f(2), 'o', label = 'f(x0)')
plt.plot(5,f(5), 'o', label = 'f(x1)')
plt.plot(8,f(8), 'o', label = 'f(x2)')
plt.fill([2, 2, 5, 5],[0, f(2), f(5), 0], color='r', alpha=0.6, label = 'Área do Trapézio 1')
plt.fill([5, 5, 8, 8],[0, f(5), f(8), 0], color='r', alpha=0.3, label = 'Área do Trapézio 2')
plt.legend()
plt.axvline(x=0,color='k',linewidth=0.6,linestyle='--')
plt.axhline(y=0,color='k',linewidth=0.6,linestyle='--');
```



29.3 A Regra 1/3 de Simpson

A **regra 1/3 de Simpson** é obtida quando um polinômio interpolador de segundo grau é substituído na Equação (1):

$$I = \int_a^b f(x)dx \cong \int_a^b f_2(x)dx$$

Se a e b forem designados por x_0 e x_2 e se $f_2(x)$ for representado por um polinômio de Lagrange de segundo grau, a integral se torna

$$I = \int_{x_0}^{x_2} \left[\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_1) + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_2) \right] dx$$

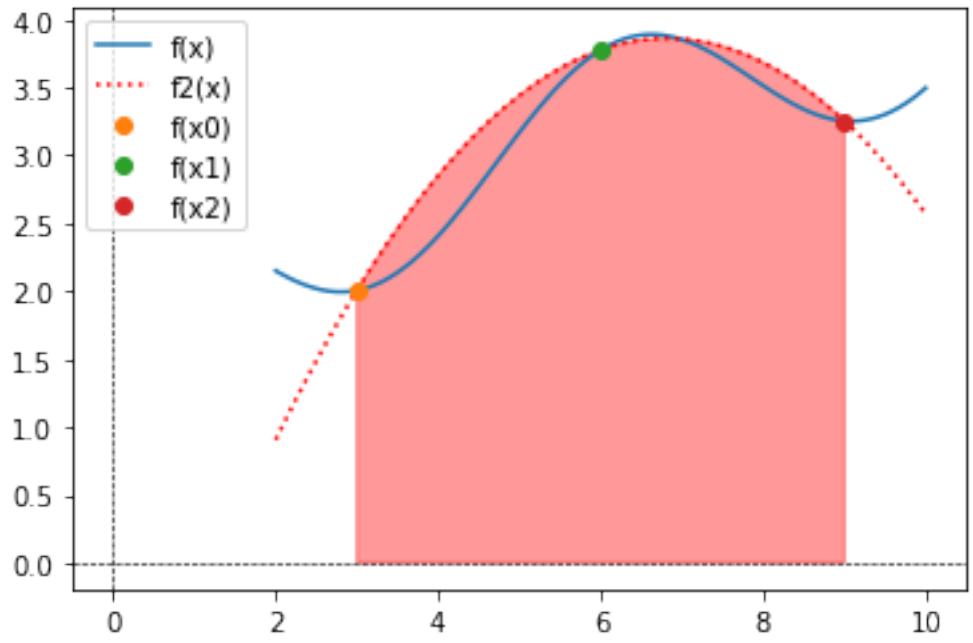
Depois da integração e de manipulações algébricas, obtém-se a seguinte fórmula:

$$I \cong \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] \qquad (5)$$

em que, para esse caso

$$h = \frac{b-a}{2} \qquad (6)$$

```
# Descrição gráfica da regra 1/3 de Simpson: ela consiste em tomar a área sob uma
 ↪parábola ligando três pontos

def f(x):
    return 0.6*np.sin(x+np.pi/2) + 0.2*x + 2

def f2(x):
    return -0.127382111059529*x**2 + 1.73647155618593*x - 2.0569711669823 # Obtido
 ↪pelo MMQ

x = np.linspace(2,10,100)
y = f(x)

xc = np.linspace(2,10,100)
yc = f2(xc)

xf = np.linspace(3,9,100)
yf = f2(xf)

xf = np.concatenate((np.array([3]), xf, np.array([9])))
yf = np.concatenate((np.array([0]), yf, np.array([0])))

plt.plot(x,y, label = 'f(x)')
plt.plot(xc,yc, ':r', label = 'f2(x)')
plt.plot(3,f(3), 'o', label = 'f(x0)')
plt.plot(6,f(6), 'o', label = 'f(x1)')
plt.plot(9,f(9), 'o', label = 'f(x2)')
plt.fill(xf,yf, color='r', alpha=0.4)
plt.legend()
plt.axvline(x=0,color='k',linewidth=0.6,linestyle='--')
plt.axhline(y=0,color='k',linewidth=0.6,linestyle='--');
```



29.4 Aplicações Múltiplas da Regra 1/3 de Simpson

Do mesmo modo como no caso da regra do trapézio, a regra de Simpson pode ser melhorada dividindo-se o intervalo de integração em diversos segmentos de mesmo comprimento

$$h = \frac{b-a}{2}$$

A integral total pode ser representada como

$$I = \int_{x_0}^{x_2} f(x)dx + \int_{x_2}^{x_4} f(x)dx + \cdots + \int_{x_{n-2}}^{x_n} f(x)dx$$

Substituindo cada integral individual pela regra 1/3 de Simpson, obtemos

$$I \cong 2h\frac{f(x_0) + 4f(x_1) + f(x_2)}{6} + 2h\frac{f(x_2) + 4f(x_3) + f(x_4)}{6} + \cdots + 2h\frac{f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)}{6}$$

ou

$$I \cong (b-a)\frac{f(x_0) + 4\sum_{i=1,3,5}^{n-1} f(x_i) + 2\sum_{i=2,4,6}^{n-2} f(x_j) + f(x_n)}{3n} \qquad (7)$$

29.5 Regra 3/8 de Simpson

De uma maneira parecida com a dedução da regra do trapézio e da regra 1/3 de Simpson, um polinômio de Lagrange de ordem três pode ser ajustado a quatro pontos e integrado

$$I = \int_a^b f(x)dx \cong \int_a^b f_3(x)dx$$

para fornecer

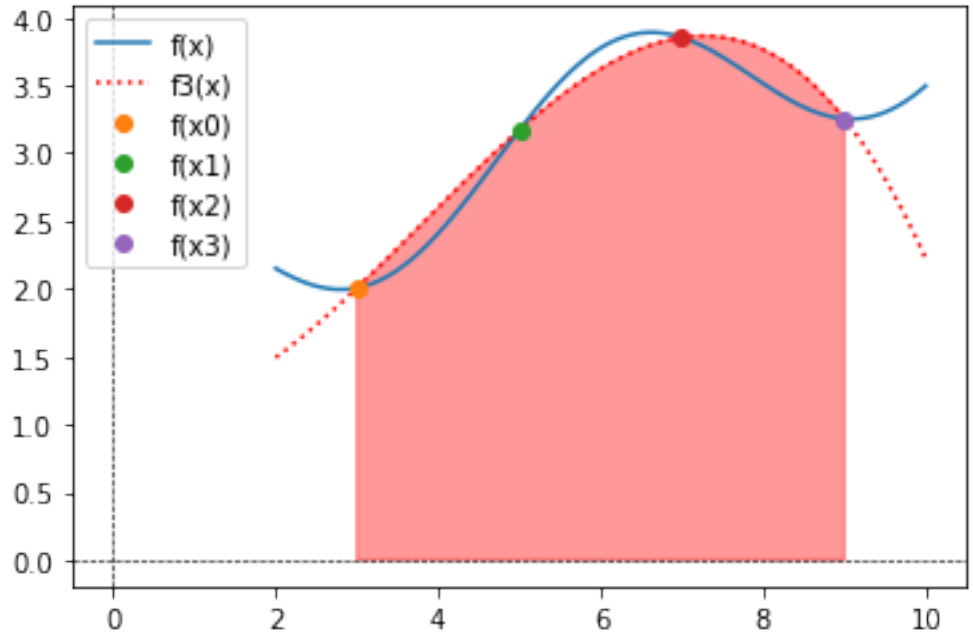
$$I \cong \frac{3h}{8}[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)] \qquad (8)$$

em que

$$h = \frac{b-a}{3} \qquad (9)$$

Essa equação é chamada regra 3/8 de Simpson porque h é multiplicada por 3/8.

```
# Descrição gráfica da regra 3/8 de Simpson: ela consiste em tomar a área sob uma␣
↪equação cúbica ligando quatro pontos

def f3(x):
    return -0.0166482246490701*x**3 + 0.189467273747325*x**2 - 0.117878777555138*x +␣
↪1.10393743650415 # Obtido pelo MMQ

xc = np.linspace(2,10,100)
yc = f3(xc)

xf = np.linspace(3,9,100)
yf = f3(xf)

xf = np.concatenate((np.array([3]), xf, np.array([9])))
yf = np.concatenate((np.array([0]), yf, np.array([0])))

plt.plot(x,y, label = 'f(x)')
plt.plot(xc,yc, ':r', label = 'f3(x)')
plt.plot(3, f(3), 'o', label = 'f(x0)')
plt.plot(5, f(5), 'o', label = 'f(x1)')
plt.plot(7, f(7), 'o', label = 'f(x2)')
plt.plot(9, f(9), 'o', label = 'f(x3)')
plt.fill(xf,yf, color='r', alpha=0.4)
plt.legend()
plt.axvline(x=0,color='k',linewidth=0.6,linestyle='--')
plt.axhline(y=0,color='k',linewidth=0.6,linestyle='--');
```



29.6 Integração com Segmentos Desiguais

Até esse ponto, todas as fórmulas para integração numérica foram baseadas em dados igualmente espaçados. Na prática, existem muitas situações nas quais essa hipótese não é válida e precisamos lidar com segmentos de tamanhos distintos. Por exemplo, dados obtidos experimentalmente, muitas vezes, são desse tipo. Para tais casos, um método é aplicar a regra do trapézio para cada segmento e somar os resultados:

$$I = h_1 \frac{f(x_0) + f(x_1)}{2} + h_2 \frac{f(x_1) + f(x_2)}{2} + \cdots + h_n \frac{f(x_{n-1}) + f(x_n)}{2}$$

em que h_i é a largura do segmento i.

29.7 Implementações de Newton-Cotes: Regra do Trapézio e 1/3 Simpson Generalizadas

```
''' Newton-Cotes: Regra do Trapezio
    assume Y igualmente espaçado e
    com pelo menos 2 pontos
'''
def integral_trapezio(h,Y):
    val = 0.0
    for i in range(1,Y.size-1):
        val += 2*Y[i]
    val = 0.5*h*( val + Y[0] + Y[-1] )
    return val

''' Newton-Cotes: Regra 1/3 de Simpson
    assume Y igualmente espaçado e
    com pelo menos 3 pontos
'''
def integral_onethird_simpson(h,Y):
    val = 0.0
    for i in range(1,Y.size-1,2):
        val += 4*Y[i]

    for i in range(2,Y.size-2,2):
        val += 2*Y[i]

    val = h/3.0*( val + Y[0] + Y[-1] )
    return val

''' HELPER'''
def print_metodo(flag):
    print('*** Método de integração: ' + str.upper(flag) + ' ***')

# limites de integração
a = 0
b = 93

# integrando
f = lambda v: 97000*v/(5*v**2 + 570000)

# pontos de integração (ímpares para testar com 1/3 simpson)
npi = [11,101,1001,10001,100001]
```

(continues on next page)

(continued from previous page)

```
# integral exata
x = sy.Symbol('x')
vex = sy.integrate(f(x),(x,a,b))
print('Ingegral exata (simbólica): ' + str(vex))
vex = float(vex)
print('Integral exata (numérica): ' + str(vex))

metodos = ['trapezio','13simpson']
no = False

# integração numérica
for metodo in metodos:
    I = []
    EREL = []
    for n in npi:
        v = np.linspace(a,b,num=n,endpoint=True)
        fv = f(v)
        h = (b-a)/n

        if metodo == 'trapezio':
            if no == False:
                print_metodo(metodo)
                no = True
            val = integral_trapezio(h,fv)
            erel = abs((vex - val)/vex)*100
            I.append(val)
            EREL.append(erel)
            print("no. de pontos de integração = {0:d} \t I = {1:.10f} \t EREL = {2:.
↪10f}%".format(n,val,erel) )

        elif metodo == '13simpson':
            if no == False:
                print_metodo(metodo)
                no = True
            val = integral_onethird_simpson(h,fv)
            erel = abs((vex - val)/vex)*100
            I.append(val)
            EREL.append(erel)
            print("no. de pontos de integração = {0:d} \t I = {1:.10f} \t EREL = {2:.
↪10f}%".format(n,val,erel) )

    no = False
```

```
Ingegral exata (simbólica): -9700*log(114000) + 9700*log(122649)
Integral exata (numérica): 709.3432392521672
*** Método de integração: TRAPEZIO ***
no. de pontos de integração = 11          I = 644.6326388025          EREL = 9.
↪1226076276%
no. de pontos de integração = 101          I = 702.3175905243          EREL = 0.
↪9904441657%
no. de pontos de integração = 1001          I = 708.6345799441          EREL = 0.
↪0999035825%
no. de pontos de integração = 10001          I = 709.2723117737          EREL = 0.
↪0099990350%
no. de pontos de integração = 100001          I = 709.3361458882          EREL = 0.
↪0009999903%
```

(continues on next page)

(continued from previous page)

```
*** Método de integração: 13SIMPSON ***
no. de pontos de integração = 11          I = 644.8576901994          EREL = 9.
↪0908809000%
no. de pontos de integração = 101          I = 702.3200388852          EREL = 0.
↪9900990068%
no. de pontos de integração = 1001          I = 708.6346046475          EREL = 0.
↪0999000999%
no. de pontos de integração = 10001          I = 709.2723120210          EREL = 0.
↪0099990001%
no. de pontos de integração = 100001          I = 709.3361458907          EREL = 0.
↪0009999900%
```

29.7.1 Tarefa

Implemente uma função para realizar a integração numérica pela regra 3/8 de Simpson, use-a para calcular o valor de

$$\int_{a=0}^{b=93} \frac{97000v}{(5v^2+570000)}\,dx$$

e compare o resultado com os obtidos pelas anteriores.

29.8 Integração simbólica

Vamos ver alguns exemplos de integração simbólica. Para termos uma impressão mais elegante de expressões, antes usamos a seguinte instrução:

```
sy.init_printing()
```

29.8.1 Regra quadratura de Simpson

Vamos usar símbolos para chegar à expressão da regra de quadratura de Simpson.

```
a,b,X = sy.symbols("a, b, x")
f = sy.Function("f")
```

Definimos tuplas para os pontos de amostra e pesos.

```
x = a, (a + b)/2, b # ponto médio
w = [sy.symbols("w_%d" % i) for i in range(len(x))] # pesos
```

```
q = sum([w[i] * f(x[i]) for i in range(len(x))])
q
```

$$w_0 f(a) + w_1 f\left(\frac{a}{2} + \frac{b}{2}\right) + w_2 f(b)$$

Para calcular valores aproximados dos pesos w_i, escolhemos a base polinomial

$$\{\phi_n(x) = x^n\}_{n=0}^{2}$$

para a interpolação de $f(x)$ e um objeto simbólico para representar cada uma dessas funções.

```
phi = [sy.Lambda(X, X**n) for n in range(len(x))]
phi
```

$$\left[(x \mapsto 1)\,,\ (x \mapsto x)\,,\ \left(x \mapsto x^2\right)\right]$$

Agora temos que descobrir os valores dos pesos. A integral $\int_a^b \phi_n(x)\,dx$ pode ser calculada analiticamente para cada função de base. Isto nos ajuda a resolver o seguinte sistema:

$$\sum_{i=0}^{2} w_i \phi_n(x_i) = \int_a^b \phi_n(x)\,dx$$

O sistema pode ser construído no `sympy` da seguinte forma:

```
eqs = [q.subs(f, phi[n]) - sy.integrate(phi[n](X), (X, a, b)) for n in␣
 ↪range(len(phi))]
eqs
```

$$\left[a - b + w_0 + w_1 + w_2,\ \frac{a^2}{2} + aw_0 - \frac{b^2}{2} + bw_2 + w_1\left(\frac{a}{2} + \frac{b}{2}\right),\ \frac{a^3}{3} + a^2 w_0 - \frac{b^3}{3} + b^2 w_2 + w_1\left(\frac{a}{2} + \frac{b}{2}\right)^2\right]$$

Em seguida, resolvemos o sistema para obter as expressões analíticas para os pesos:

```
w_sol = sy.solve(eqs,w)
w_sol
```

$$\left\{w_0 : -\frac{a}{6} + \frac{b}{6},\ w_1 : -\frac{2a}{3} + \frac{2b}{3},\ w_2 : -\frac{a}{6} + \frac{b}{6}\right\}$$

Substituindo a solução na expressão simbólica para a regra de quadratura:

```
q.subs(w_sol).simplify()
```

$$-\frac{(a-b)\left(f(a) + f(b) + 4f\left(\frac{a}{2} + \frac{b}{2}\right)\right)}{6}$$

Podemos verificar no material que esta é, de fato, a expressão para a regra de quadratura de Simpson.

29.9 Integração múltipla

A integração em 2 ou mais variáveis pode ser feita usando as funções `dblquad`, `tplquad` e `nquad`, onde o número de funções e de limites de integração deve se adequar ao tipo de integral.

Abaixo, temos alguns exemplos:

29.9.1 Integração dupla

Neste exemplo, integramos $\$\int_0^1 \int_0^1 e^{-x^2-y^2}\, dxdy\$$

```
integrate.dblquad(lambda x, y: np.exp(-x**2-y**2), 0, 1, lambda x: 0, lambda x: 1)
```

$$(0.5577462853510337,\ 8.291374381535408e-15)$$

29.9.2 Integração tripla

Neste exemplo, integramos $\$\int_0^1 \int_0^1 \int_0^1 e^{-x^2-y^2-z^2}\, dxdydz\$$

```
def f(x, y, z):
    return np.exp(-x**2-y**2-z**2)

a, b = 0, 1
g, h = lambda x: 0, lambda x: 1
q, r = lambda x, y: 0, lambda x, y: 1
integrate.tplquad(f, 0, 1, g, h, q, r)
```

$$(0.4165383858866382,\ 8.291335287314424e-15)$$

CHAPTER

THIRTY

INTEGRAÇÃO NUMÉRICA: QUADRATURA GAUSSIANA

Uma característica das fórmulas de Newton-Cotes é estimar a integral com base em valores da função tomados em nós igualmente espaçados. Consequentemente, a posição desses nós é predeterminada, isto é, fixada.

Agora, suponhamos que a restrição de ter nós fixados seja removida e que pudéssemos escolher arbitrariamente as posições por eles ocupadas. Com uma escolha adequada, será que conseguiríamos reduzir o erro de integração?

De fato, o método da Quadratura Gaussiana permite-nos estabelecer um conjunto mínimo de *nós* e *pesos* que realizam uma espécie de "balanceamento" do erro (subestimação vs. superestimação) e determinam aproximações numéricas de integrais com alta ordem.

As figuras abaixo mostram, por exemplo, como a integral de uma função $f(x)$ é melhor calculada através da colocação de pontos em posições "especiais". A regra do trapézio é posta em contraste com uma outra regra, que aprenderemos, cujo par de pontos, embora defina também um trapézio, ajuda a estimar melhor a integral de $f(x)$ do que o observado no primeiro caso.

Em particular, daremos enfoque à chamada *Quadratura de Gauss-Legendre*.

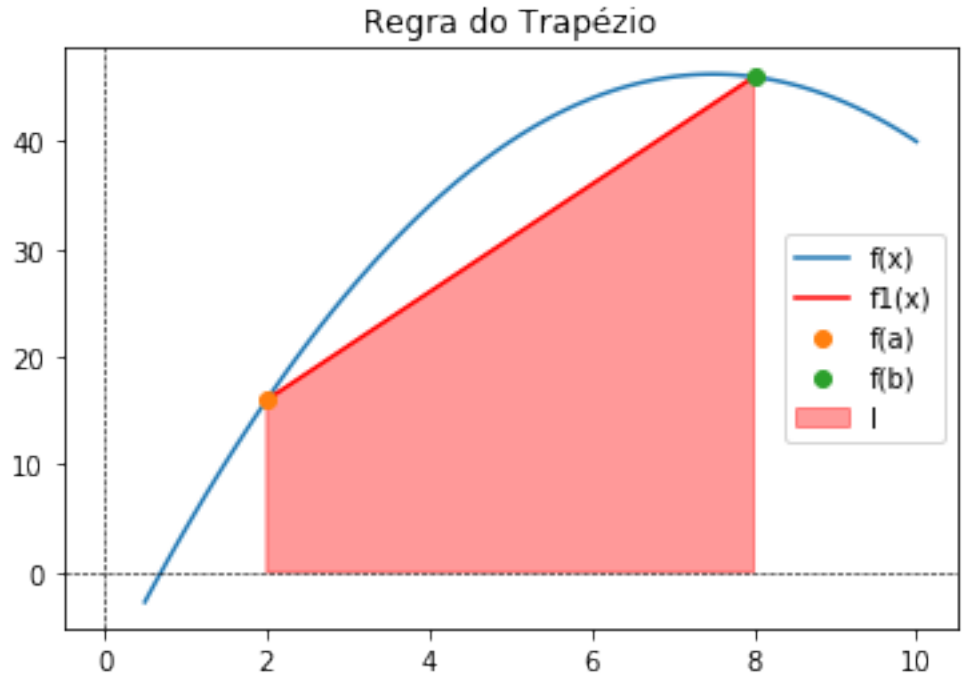
```
import matplotlib.pyplot as plt
import numpy.polynomial.legendre as leg
import numpy as np
%matplotlib inline
```

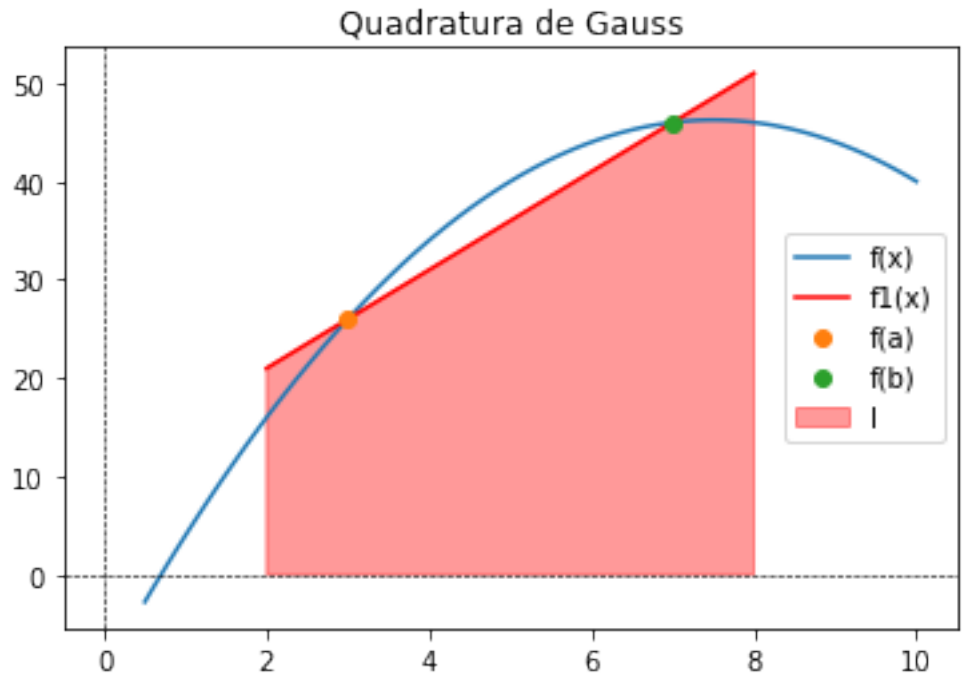
```
def f(x):
    return -x**2 + 15*x - 10

x = np.linspace(0.5,10,100)
y = f(x)

# Representação geométrica da regra do trapézio

plt.figure()
plt.plot(x,y, label = 'f(x)')
plt.plot([2,8], [f(2),f(8)], 'r', label = 'f1(x)')
plt.plot(2,f(2), 'o', label = 'f(a)')
plt.plot(8,f(8), 'o', label = 'f(b)')
plt.fill([2, 2, 8, 8],[0, f(2), f(8), 0],color='r',alpha=0.4, label = 'I')
plt.title('Regra do Trapézio')
plt.legend()
plt.axvline(x=0,color='k',linewidth=0.6,linestyle='--')
plt.axhline(y=0,color='k',linewidth=0.6,linestyle='--')

# Representação geométrica da quadratura de Gauss

def f1(x):
```

(continues on next page)

(continued from previous page)

```
    return 5.0*x + 11.0

plt.figure()
plt.plot(x,y, label = 'f(x)')
plt.plot([2,8], [f1(2),f1(8)], 'r', label = 'f1(x)')
plt.plot(3,f(3), 'o', label = 'f(a)')
plt.plot(7,f(7), 'o', label = 'f(b)')
plt.fill([2, 2, 8, 8],[0, f1(2), f1(8), 0],color='r',alpha=0.4, label = 'I')
plt.title('Quadratura de Gauss')
plt.legend()
plt.axvline(x=0,color='k',linewidth=0.6,linestyle='--')
plt.axhline(y=0,color='k',linewidth=0.6,linestyle='--');
```





30.1 Dedução da Fórmula de Gauss-Legendre para Dois Pontos

O objetivo da quadratura de Gauss-Legendre é determinar os coeficientes de uma equação da forma

$$I \cong c_0 f(x_0) + c_1 f(x_1) \qquad (1)$$

em que os c's são coeficientes desconhecidos. Entretanto, em contraste com a regra do trapézio que usa extremidades fixas a e b, os argumentos da função x_0 e x_1 não são fixados nas extremidades, mas são incógnitas. Assim, agora temos um total de quatro incógnitas que devem ser calculadas e necessitamos de quatro condições para determiná-las exatamente.

Podemos obter duas dessas condições, supondo que a Equação (1) calcula a integral de uma função constante e de uma função linear exatamente. Então, para chegar a duas outras condições, simplesmente estendemos esse raciocínio supondo que ele também calculará a integral de uma função parabólica ($y = x^2$) e de uma cúbica ($y = x^3$) exatamente. Fazendo isso, determinamos todas as quatro incógnitas e, de quebra, deduzimos uma fórmula de integração linear de dois pontos que é exata para funções cúbicas. As quatro equações a serem resolvidas são:

$$c_0 f(x_0) + c_1 f(x_1) = \int_{-1}^{1} 1dx = 2 \qquad (2)$$

$$c_0 f(x_0) + c_1 f(x_1) = \int_{-1}^{1} xdx = 0 \qquad (3)$$

$$c_0 f(x_0) + c_1 f(x_1) = \int_{-1}^{1} x^2 dx = \frac{2}{3} \qquad (4)$$

$$c_0 f(x_0) + c_1 f(x_1) = \int_{-1}^{1} x^3 dx = 0 \qquad (5)$$

As Equações (2) até (5) podem ser resolvidas simultaneamente por

$$c_0 = c_1 = 1$$
$$x_0 = -\frac{1}{\sqrt{3}}$$
$$x_1 = \frac{1}{\sqrt{3}}$$

o que pode ser substituído na Equação (1) para fornecer a fórmula de Gauss-Legendre de dois pontos

$$I \cong f\left(\frac{-1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \qquad (6)$$

Logo, chegamos ao interessante resultado que uma simples adição de valores da função em $x = 1/\sqrt{3}$ e $-1/\sqrt{3}$ fornece uma estimativa da integral que tem acurácia de terceira ordem.

Observe que os extremos de integração nas Equações (2) até (5) são de -1 a 1. Isso foi feito para simplificar a matemática e tornar a formulação tão geral quanto possível. Uma simples mudança de variável pode ser usada para transformar outros extremos de integração para essa fórmula, o que se faz supondo que uma nova variável x_d está relacionada à variável original x de uma forma linear como em

$$x = a_0 + a_1 x_d \qquad (7)$$

Se o extremo inferior, $x = a$, corresponder a $x_d = -1$, esses valores podem ser substituídos na Equação (7) para fornecer

$$a = a_0 + a_1(-1) \qquad (8)$$

Analogamente, o extremo superior, $x = b$, corresponde a $x_d = 1$, e fornece

$$b = a_0 + a_1(1) \qquad (9)$$

As Equações (8) e (9) podem ser resolvidas simultaneamente por

$$a_0 = \frac{b + a}{2}$$

e

$$a_1 = \frac{b - a}{2}$$

o que pode ser substituído na Equação (7) para fornecer

$$x = \frac{(b + a) + (b - a)x_d}{2} \qquad (10)$$

Essa equação pode ser derivada para dar

$$dx = \frac{b - a}{2} dx_d \qquad (11)$$

Os valores de x e dx, nas Equações (10) e (11), respectivamente, podem ser substituídos na equação a ser integrada. Essas substituições transformam efetivamente o intervalo de integração sem mudar o valor da integral.

30.2 Fórmulas com Mais Pontos

Além da fórmula de dois pontos descrita acima, versões com mais pontos podem ser desenvolvidas na forma geral

$$I \cong c_0 f(x_0) + c_1 f(x_1) + \cdots + c_{n-1} f(x_{n-1}) \qquad (12)$$

em que n é o número de pontos. Os valores para os c's e os x's para até (e incluindo) a fórmula de quatro pontos estão resumidos na tabela abaixo.

Pontos	Fatores de Peso	Argumentos da Função
3	c_0 = 0.5555556	x_0 = - 0.774596669
c_1 = 0.8888889	x_1 = 0.0	
c_2 = 0.5555556	x_2 = 0.774596669	
4	c_0 = 0.3478548	x_0 = - 0.861136312
c_1 = 0.6521452	x_1 = - 0.339981044	
c_2 = 0.6521452	x_2 = 0.339981044	
c_3 = 0.3478548	x_3 = 0.861136312	

30.3 Implementação

Vamos implementar abaixo o código para gerar a tabela de pontos (nós) e pesos para integração numérica consoante as fórmulas de quadratura de **Gauss-Legendre**.

```
# número de pontos de quadratura
n = 8

# pontos e pesos
(pontos,pesos) = leg.leggauss(n)
```

Isto é, para a regra de 2 pontos, os nós de Gauss-Legendre são

```
print(pontos)
```

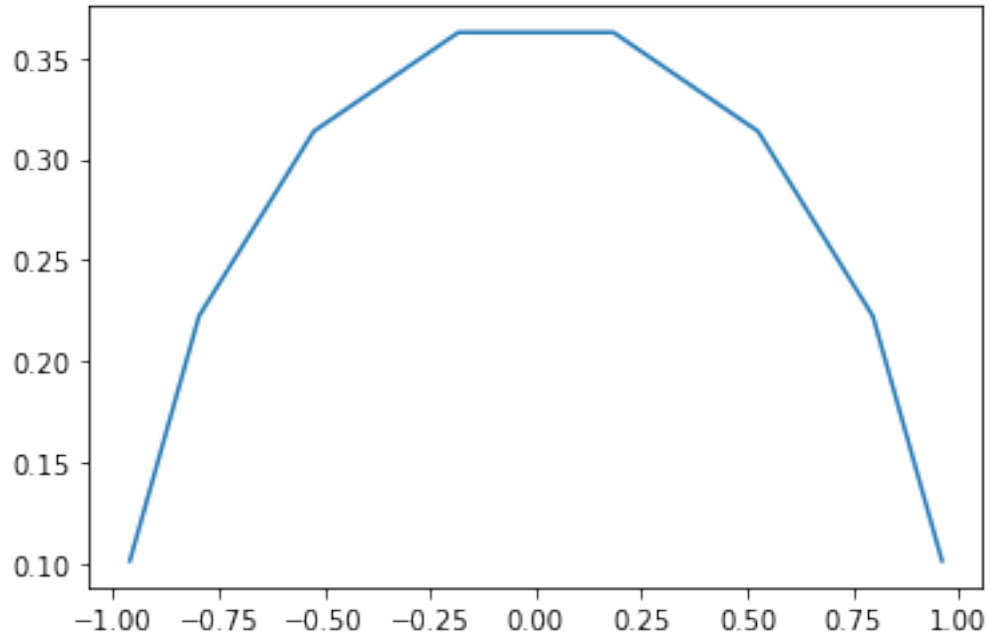
```
[-0.96028986 -0.79666648 -0.52553241 -0.18343464  0.18343464  0.52553241
  0.79666648  0.96028986]
```

e os pesos correspondentes são:

```
print(pesos)
```

```
[0.10122854 0.22238103 0.31370665 0.36268378 0.36268378 0.31370665
 0.22238103 0.10122854]
```

```
plt.plot(pontos,pesos)
plt.autoscale;
```



30.3.1 Tabela de pesos/pontos - Quadratura de Gauss-Legendre

Para gerarmos uma tabela de pontos e pesos, basta fazer:

```
# número máximo de pontos
N = 16

for i in range(1,N+1):
    (pontos,pesos) = leg.leggauss(i)
    print('REGRA DE {0} PONTO(S):\n-> Pontos:'.format(i))
    print(pontos)
    print('-> Pesos:')
    print(pesos)
    print('\n')
```

```
REGRA DE 1 PONTO(S):
-> Pontos:
[0.]
-> Pesos:
[2.]


REGRA DE 2 PONTO(S):
-> Pontos:
[-0.57735027  0.57735027]
-> Pesos:
[1. 1.]


REGRA DE 3 PONTO(S):
-> Pontos:
[-0.77459667  0.          0.77459667]
-> Pesos:
```

(continues on next page)

(continued from previous page)

```
[0.55555556 0.88888889 0.55555556]


REGRA DE 4 PONTO(S):
-> Pontos:
[-0.86113631 -0.33998104  0.33998104  0.86113631]
-> Pesos:
[0.34785485 0.65214515 0.65214515 0.34785485]


REGRA DE 5 PONTO(S):
-> Pontos:
[-0.90617985 -0.53846931  0.          0.53846931  0.90617985]
-> Pesos:
[0.23692689 0.47862867 0.56888889 0.47862867 0.23692689]


REGRA DE 6 PONTO(S):
-> Pontos:
[-0.93246951 -0.66120939 -0.23861919  0.23861919  0.66120939  0.93246951]
-> Pesos:
[0.17132449 0.36076157 0.46791393 0.46791393 0.36076157 0.17132449]


REGRA DE 7 PONTO(S):
-> Pontos:
[-0.94910791 -0.74153119 -0.40584515  0.          0.40584515  0.74153119
  0.94910791]
-> Pesos:
[0.12948497 0.27970539 0.38183005 0.41795918 0.38183005 0.27970539
 0.12948497]


REGRA DE 8 PONTO(S):
-> Pontos:
[-0.96028986 -0.79666648 -0.52553241 -0.18343464  0.18343464  0.52553241
  0.79666648  0.96028986]
-> Pesos:
[0.10122854 0.22238103 0.31370665 0.36268378 0.36268378 0.31370665
 0.22238103 0.10122854]


REGRA DE 9 PONTO(S):
-> Pontos:
[-0.96816024 -0.83603111 -0.61337143 -0.32425342  0.          0.32425342
  0.61337143  0.83603111  0.96816024]
-> Pesos:
[0.08127439 0.18064816 0.2606107  0.31234708 0.33023936 0.31234708
 0.2606107  0.18064816 0.08127439]


REGRA DE 10 PONTO(S):
-> Pontos:
[-0.97390653 -0.86506337 -0.67940957 -0.43339539 -0.14887434  0.14887434
  0.43339539  0.67940957  0.86506337  0.97390653]
-> Pesos:
[0.06667134 0.14945135 0.21908636 0.26926672 0.29552422 0.29552422
```

(continues on next page)

(continued from previous page)

```
 0.26926672 0.21908636 0.14945135 0.06667134]


REGRA DE 11 PONTO(S):
-> Pontos:
[-0.97822866 -0.8870626  -0.73015201 -0.51909613 -0.26954316  0.
  0.26954316  0.51909613  0.73015201  0.8870626   0.97822866]
-> Pesos:
[0.05566857 0.12558037 0.18629021 0.23319376 0.26280454 0.27292509
 0.26280454 0.23319376 0.18629021 0.12558037 0.05566857]


REGRA DE 12 PONTO(S):
-> Pontos:
[-0.98156063 -0.90411726 -0.76990267 -0.58731795 -0.3678315  -0.12523341
  0.12523341  0.3678315   0.58731795  0.76990267  0.90411726  0.98156063]
-> Pesos:
[0.04717534 0.10693933 0.16007833 0.20316743 0.23349254 0.24914705
 0.24914705 0.23349254 0.20316743 0.16007833 0.10693933 0.04717534]


REGRA DE 13 PONTO(S):
-> Pontos:
[-0.98418305 -0.9175984  -0.80157809 -0.64234934 -0.44849275 -0.23045832
  0.          0.23045832  0.44849275  0.64234934  0.80157809  0.9175984
  0.98418305]
-> Pesos:
[0.040484   0.0921215  0.13887351 0.17814598 0.20781605 0.22628318
 0.23255155 0.22628318 0.20781605 0.17814598 0.13887351 0.0921215
 0.040484  ]


REGRA DE 14 PONTO(S):
-> Pontos:
[-0.98628381 -0.92843488 -0.82720132 -0.6872929  -0.51524864 -0.31911237
 -0.10805495  0.10805495  0.31911237  0.51524864  0.6872929   0.82720132
  0.92843488  0.98628381]
-> Pesos:
[0.03511946 0.08015809 0.12151857 0.15720317 0.1855384  0.20519846
 0.21526385 0.21526385 0.20519846 0.1855384  0.15720317 0.12151857
 0.08015809 0.03511946]


REGRA DE 15 PONTO(S):
-> Pontos:
[-0.98799252 -0.93727339 -0.84820658 -0.72441773 -0.57097217 -0.39415135
 -0.20119409  0.          0.20119409  0.39415135  0.57097217  0.72441773
  0.84820658  0.93727339  0.98799252]
-> Pesos:
[0.03075324 0.07036605 0.10715922 0.13957068 0.16626921 0.186161
 0.19843149 0.20257824 0.19843149 0.186161   0.16626921 0.13957068
 0.10715922 0.07036605 0.03075324]


REGRA DE 16 PONTO(S):
-> Pontos:
[-0.98940093 -0.94457502 -0.8656312  -0.75540441 -0.61787624 -0.45801678
```

(continues on next page)

(continued from previous page)

```
 -0.28160355 -0.09501251  0.09501251  0.28160355  0.45801678  0.61787624
  0.75540441  0.8656312   0.94457502  0.98940093]
-> Pesos:
[0.02715246 0.06225352 0.09515851 0.12462897 0.14959599 0.16915652
 0.18260342 0.18945061 0.18945061 0.18260342 0.16915652 0.14959599
 0.12462897 0.09515851 0.06225352 0.02715246]
```

A partir daí, podemos organizar uma tabela para a regra de até 8 pontos/pesos como segue:

```
# número máximo de pontos
N = 8

header='| Regra | nó(s) | peso(s) |\n|---|---|---|'
print(header)
for i in range(1,N+1):
    (pontos,pesos) = leg.leggauss(i)
    p = ', '.join([str(p) for p in pontos])
    w = ', '.join([str(p) for p in pesos])
    row = '|' + str(i) + '|' + p + '|' + w + '|'
    print(row)
```

```
| Regra | nó(s) | peso(s) |
|---|---|---|
|1|0.0|2.0|
|2|-0.5773502691896257, 0.5773502691896257|1.0, 1.0|
|3|-0.7745966692414834, 0.0, 0.7745966692414834|0.5555555555555557, 0.
 ↪8888888888888888, 0.5555555555555557|
|4|-0.8611363115940526, -0.33998104358485626, 0.33998104358485626, 0.
 ↪8611363115940526|0.3478548451374537, 0.6521451548625462, 0.6521451548625462, 0.
 ↪3478548451374537|
|5|-0.906179845938664, -0.5384693101056831, 0.0, 0.5384693101056831, 0.
 ↪906179845938664|0.23692688505618942, 0.4786286704993662, 0.568888888888889, 0.
 ↪4786286704993662, 0.23692688505618942|
|6|-0.932469514203152, -0.6612093864662645, -0.23861918608319693, 0.23861918608319693,
 ↪ 0.6612093864662645, 0.932469514203152|0.17132449237916975, 0.36076157304813894, 0.
 ↪46791393457269137, 0.46791393457269137, 0.36076157304813894, 0.17132449237916975|
|7|-0.9491079123427585, -0.7415311855993945, -0.4058451513773972, 0.0, 0.
 ↪4058451513773972, 0.7415311855993945, 0.9491079123427585|0.12948496616887065, 0.
 ↪2797053914892766, 0.3818300505051183, 0.41795918367346896, 0.3818300505051183, 0.
 ↪2797053914892766, 0.12948496616887065|
|8|-0.9602898564975362, -0.7966664774136267, -0.525532409916329, -0.18343464249564978,
 ↪ 0.18343464249564978, 0.525532409916329, 0.7966664774136267, 0.9602898564975362|0.
 ↪10122853629037669, 0.22238103445337434, 0.31370664587788705, 0.36268378337836177, 0.
 ↪36268378337836177, 0.31370664587788705, 0.22238103445337434, 0.10122853629037669|
```

Tabela de quadratura de Gauss-Legendre

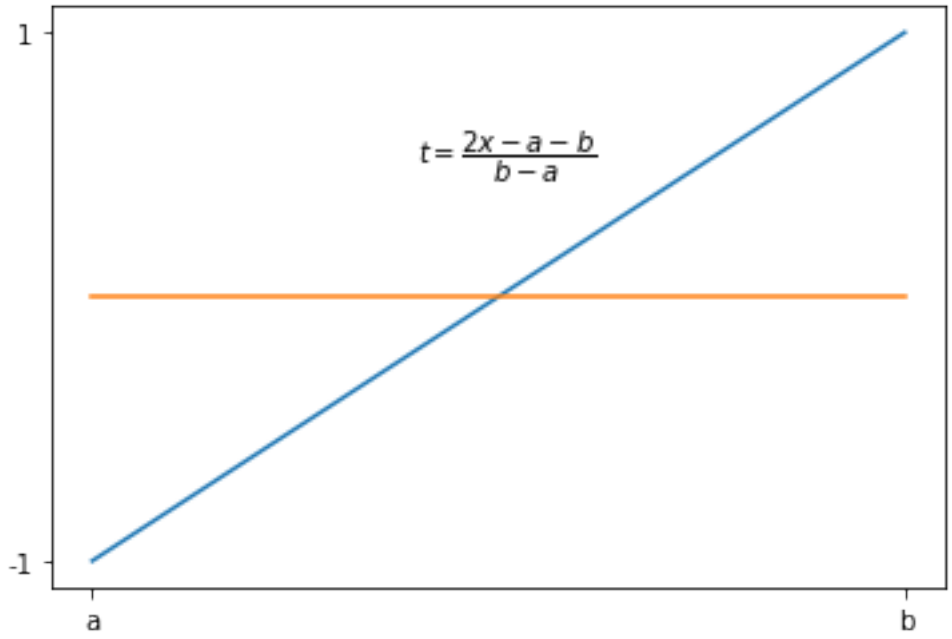
Regra	nó(s)	peso(s)
1	0.0	2.0
2	-0.57735026919, 0.57735026919	1.0, 1.0
3	-0.774596669241, 0.0, 0.774596669241	0.555555555556, 0.888888888889, 0.555555555556
4	-0.861136311594, -0.339981043585, 0.339981043585, 0.861136311594	0.347854845137, 0.652145154863, 0.652145154863, 0.347854845137
5	-0.906179845939, -0.538469310106, 0.0, 0.538469310106, 0.906179845939	0.236926885056, 0.478628670499, 0.568888888889, 0.478628670499, 0.236926885056
6	-0.932469514203, -0.661209386466, -0.238619186083, 0.238619186083, 0.661209386466, 0.932469514203	0.171324492379, 0.360761573048, 0.467913934573, 0.467913934573, 0.360761573048, 0.171324492379
7	-0.949107912343, -0.741531185599, -0.405845151377, 0.0, 0.405845151377, 0.741531185599, 0.949107912343	0.129484966169, 0.279705391489, 0.381830050505, 0.417959183673, 0.381830050505, 0.279705391489, 0.129484966169
8	-0.960289856498, -0.796666477414, -0.525532409916, -0.183434642496, 0.183434642496, 0.525532409916, 0.796666477414, 0.960289856498	0.10122853629, 0.222381034453, 0.313706645878, 0.362683783378, 0.362683783378, 0.313706645878, 0.222381034453, 0.10122853629

30.4 Transformação de variáveis

Uma integral $\int_a^b f(x)\,dx$ sobre $[a,b]$ arbitrário ser transformada em uma integral em $[-1,1]$ utilizando a mudança de variáveis:

$$t = \frac{2x-a-b}{b-a} \Rightarrow x = \frac{1}{2}[(b-a)t + a + b]$$

```
x = np.linspace(2,4)
y = x - 3
plt.plot(x,y);
plt.plot(x,x*0);
plt.xticks([2,4],['a','b']);
plt.yticks([-1,1],['-1','1']);
plt.annotate('$t = \dfrac{2x - a - b}{b - a}$',(2.8,0.5));
```



30.4.1 Tarefa

Defina uma função como a seguinte que retorne `output`, tal que `type(output)` seja `str`.

```
def print_gauss_legendre_table(N):
    header='| Regra | nó(s) | peso(s) |\n|---|---|---|'
    print(header)
    for i in range(1,N+1):
        (pontos,pesos) = leg.leggauss(i)
        p = ', '.join([str(p) for p in pontos])
        w = ', '.join([str(p) for p in pesos])
        row = '|' + str(i) + '|' + p + '|' + w + '|'
        print(row)
```

Então, reimprima a tabela para 8 pesos/pontos anterior com o código.

```
output = print_gauss_legendre_table(8)
```

Em seguida, use o código abaixo para converter a saída da célula de código do Jupyter diretamente para Markdown.

```
from IPython.display import display, Markdown
display(Markdown(output))
```

Por último, incorpore esta funcionalidade em `print_gauss_legendre_table(N)`, para `N` dado.

Regra	nó(s)	peso(s)
1	0.0	2.0
2	-0.57735026919, 0.57735026919	1.0, 1.0
3	-0.774596669241, 0.0, 0.774596669241	0.555555555556, 0.888888888889, 0.555555555556
4	-0.861136311594, -0.339981043585, 0.339981043585, 0.861136311594	0.347854845137, 0.652145154863, 0.652145154863, 0.347854845137
5	-0.906179845939, -0.538469310106, 0.0, 0.538469310106, 0.906179845939	0.236926885056, 0.478628670499, 0.568888888889, 0.478628670499, 0.236926885056
6	-0.932469514203, -0.661209386466, -0.238619186083, 0.238619186083, 0.661209386466, 0.932469514203	0.171324492379, 0.360761573048, 0.467913934573, 0.467913934573, 0.360761573048, 0.171324492379
7	-0.949107912343, -0.741531185599, -0.405845151377, 0.0, 0.405845151377, 0.741531185599, 0.949107912343	0.129484966169, 0.279705391489, 0.381830050505, 0.417959183673, 0.381830050505, 0.279705391489, 0.129484966169
8	-0.960289856498, -0.796666477414, -0.525532409916, -0.183434642496, 0.183434642496, 0.525532409916, 0.796666477414, 0.960289856498	0.10122853629, 0.222381034453, 0.313706645878, 0.362683783378, 0.362683783378, 0.313706645878, 0.222381034453, 0.10122853629
9	-0.968160239508, -0.836031107327, -0.613371432701, -0.324253423404, 0.0, 0.324253423404, 0.613371432701, 0.836031107327, 0.968160239508	0.0812743883616, 0.180648160695, 0.260610696403, 0.31234707704, 0.330239355001, 0.31234707704, 0.260610696403, 0.180648160695, 0.0812743883616
10	-0.973906528517, -0.865063366689, -0.679409568299, -0.433395394129, -0.148874338982, 0.148874338982, 0.433395394129, 0.679409568299, 0.865063366689, 0.973906528517	0.0666713443087, 0.149451349151, 0.219086362516, 0.26926671931, 0.295524224715, 0.295524224715, 0.26926671931, 0.219086362516, 0.149451349151, 0.0666713443087
11	-0.978228658146, -0.887062599768, -0.730152005574, -0.519096129207, -0.269543155952, 0.0, 0.269543155952, 0.519096129207, 0.730152005574, 0.887062599768, 0.978228658146	0.0556685671162, 0.125580369465, 0.186290210928, 0.233193764592, 0.26280454451, 0.272925086778, 0.26280454451, 0.233193764592, 0.186290210928, 0.125580369465, 0.0556685671162
12	-0.981560634247, -0.90411725637, -0.769902674194, -0.587317954287, -0.367831498998, -0.125233408511, 0.125233408511, 0.367831498998, 0.587317954287, 0.769902674194, 0.90411725637, 0.981560634247	0.0471753363865, 0.106939325995, 0.160078328543, 0.203167426723, 0.233492536538, 0.249147045813, 0.249147045813, 0.233492536538, 0.203167426723, 0.160078328543, 0.106939325995, 0.0471753363865

CHAPTER

THIRTYONE

DIFERENCIAÇÃO NUMÉRICA

31.1 Fórmula de Derivação de Alta Acurácia

A expansão em série de Taylor progressiva pode ser escrita como

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2}h^2 + ...$$

o que pode ser resolvido por

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f''(x_i)}{2}h + O(h^2) \qquad (1)$$

Truncamos esse resultado excluindo os termos da segunda derivada e de derivadas de ordem superior e ficamos com o resultado final

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h) \qquad (2)$$

Em contraste a essa abordagem, agora manteremos o termo da segunda derivada substituindo a seguinte aproximação da segunda derivada

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + O(h) \qquad (3)$$

na Equação (1) para obter

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{2h^2} + O(h^2)$$

ou, agrupando os termos,

$$f'(x_i) = \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{2h} + O(h^2) \qquad (4)$$

Observe que a inclusão do termo da segunda derivada melhorou a precisão para $O(h^2)$.

31.2 Derivada de Dados Não Uniformemente Espaçados

Uma forma de tratar dados não uniformemente espaçados é ajustar um polinômio interpolador de Lagrange aos dados. Para uma derivada de primeira ordem mais acurada, por exemplo, podemos deduzir um polinômio de segundo grau para cada conjunto de três pontos adjacentes analiticamente através das expressões das funções de base de Lagrange. Em seguida, derivamos as funções de base de modo a obter

$$f'(x) = f(x_{i-1})\frac{2x - x_i - x_{i+1}}{(x_{i-1} - x_i)(x_{i-1} - x_{i+1})} + f(x_i)\frac{2x - x_{i-1} - x_{i+1}}{(x_i - x_{i-1})(x_i - x_{i+1})} + f(x_{i+1})\frac{2x - x_{i-1} - x_i}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)}, \qquad (5)$$

em que x é o valor no qual se quer estimar a derivada.

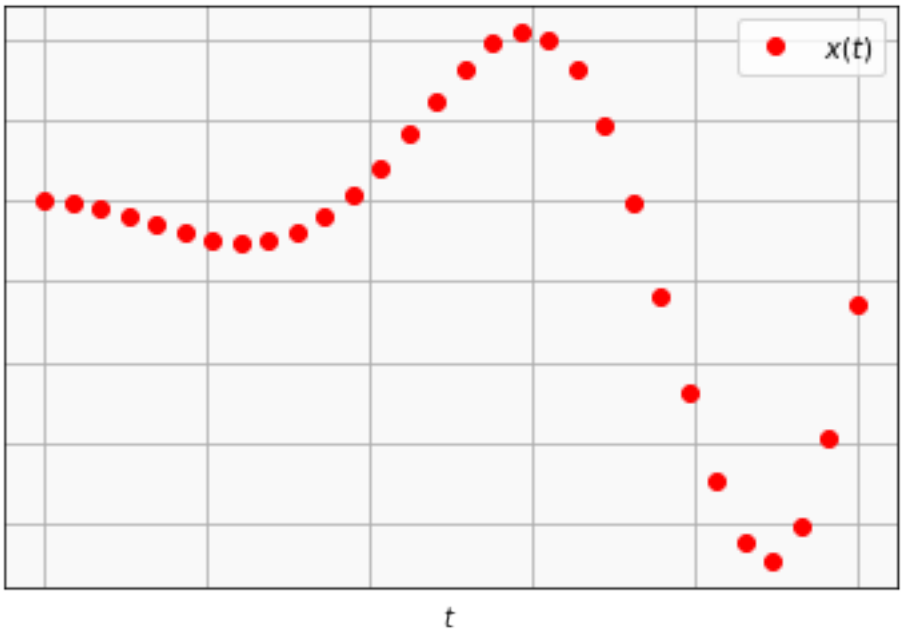
```
import numpy as np
import sympy as sy
import matplotlib.pyplot as plt
%matplotlib inline
```

31.3 Motivação

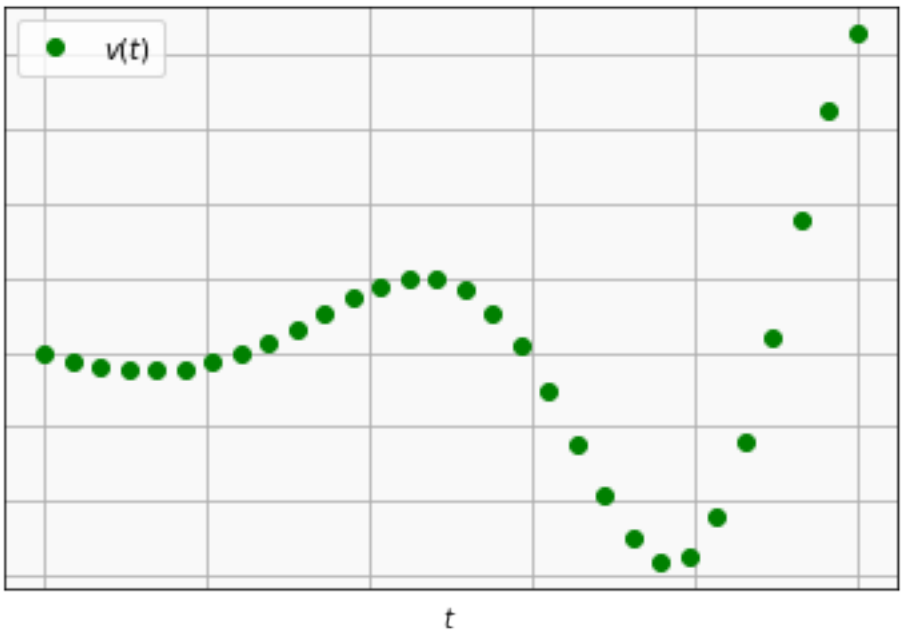
Gerando curvas de deslocamento, velocidade e aceleração para uma partícula.

```
# eixo temporal
x = np.linspace(0,1,30,endpoint=True)

# lei de movimento
y = 0.93*np.cos(2.5*(x-1))*np.sin(6*x*x)

# velocidade e aceleração (funções analíticas)
# derivação simbólica
t = sy.Symbol('t')
st = 0.93*sy.cos(2.5*(t-1))*sy.sin(6*t*t)
dt = sy.diff(st,t) # v(t)
dt2 = sy.diff(dt,t) # a(t)

# converte para numérico
dy = np.asarray([dt.subs(t,xn) for xn in x])
dy2 = np.asarray([dt2.subs(t,xn) for xn in x])

# função para plotagem
def plotting(x,y,c,lab):
    plt.plot(x,y,c,label=lab)
    plt.grid()
    plt.xlabel('$t$')
    plt.tick_params(
        axis='both',
        which='both',
        bottom=False,
        left=False,
        labelbottom=False,
        labelleft=False)
    plt.legend()
    ax = plt.gca()
    ax.set_facecolor('#F9F9F9')
```

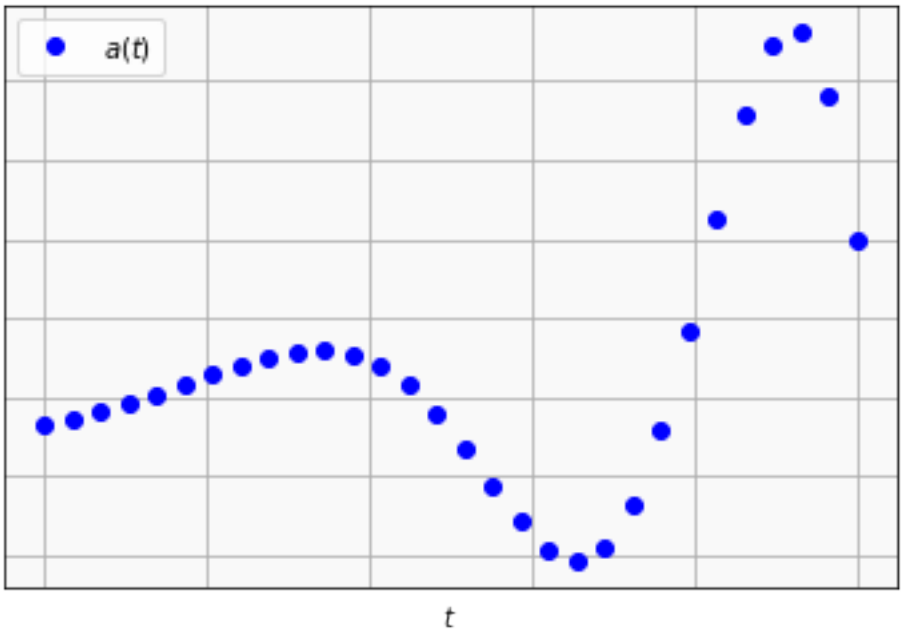
```
# curva: deslocamento
plotting(x,y,'or','$x(t)$')
```




```
# curva: velocidade
plotting(x,dy,'og','$v(t)$')
```



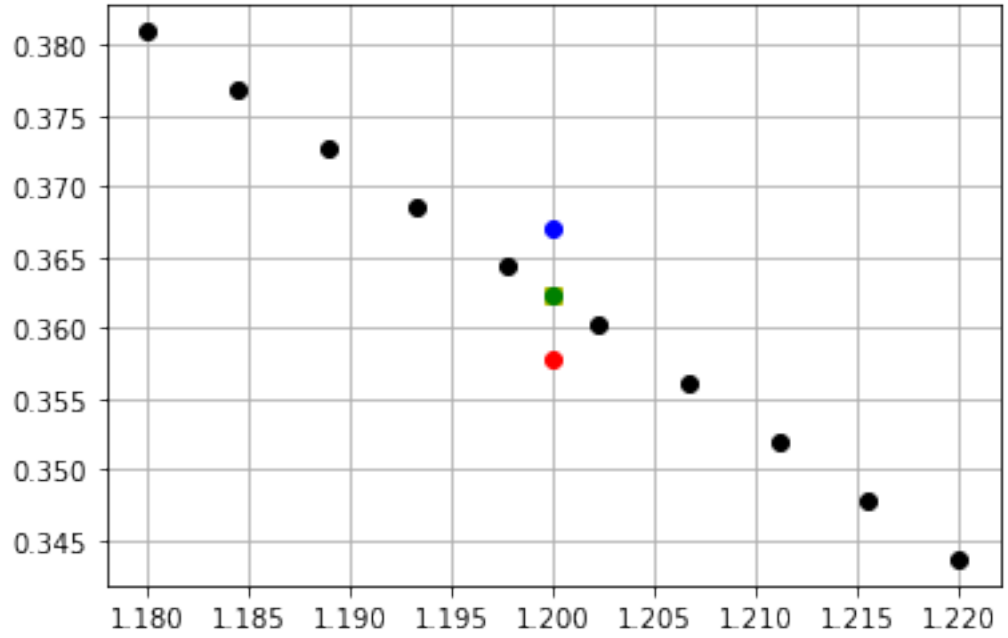

```
# curva: aceleração
plotting(x,dy2,'ob','$a(t)$')
```



31.4 Derivada numérica progressiva e regressiva (primeira ordem)

```
# Exemplo: f(x) = sen(x);
# Derivada analítica (verdadeira): f'(x=x0) = cos(x0)
x0 = 1.2
h = .01
n = 10
x = np.linspace(x0-2*h,x0+2*h,n)

# expressões analíticas de f(x) e f'(x)
f = np.sin(x)
df = np.cos(x)

# valor "exato" de f'(x) no ponto x=x0
dfe = np.cos(x0)

dfp = (np.sin(x0+h) - np.sin(x0))/h       # DF progressiva
dfr = (np.sin(x0) - np.sin(x0-h))/h       # DF regressiva
dfc = (np.sin(x0+h) - np.sin(x0-h))/(2*h) # DF centrada

plt.plot(x,df,'ok') # derivada exata
plt.plot(x0,dfe,'sy') # derivada exata no ponto x0
plt.plot(x0,dfp,'or') # derivada numérica PROGRESSIVA no ponto x0
plt.plot(x0,dfr,'ob') # derivada numérica REGRESSIVA no ponto x0
plt.plot(x0,dfc,'og') # derivada numérica CENTRADA no ponto x0
plt.grid(True)

print('h={0:.3f}'.format(h))
print('f\'e({0:g}) = {1:.8f}'.format(x0,dfe))
print('f\'p({0:g}) = {1:.8f}'.format(x0,dfp))
print('f\'r({0:g}) = {1:.8f}'.format(x0,dfr))
print('f\'c({0:g}) = {1:.8f}'.format(x0,dfc))
```

```
h=0.010
f'e(1.2) = 0.36235775
f'p(1.2) = 0.35769156
f'r(1.2) = 0.36701187
f'c(1.2) = 0.36235172
```



31.4.1 Comparando resultados

Exemplo: considere a função $f(x) = \frac{2^x}{x}$. Calcule a segunda derivada em $x = 2$ numericamente com a fórmula de diferença centrada a 3 pontos usando:

- Os pontos $x = 1.8$, $x = 2$ e $x = 2.2$.
- Os pontos $x = 1.9$, $x = 2$ e $x = 2.1$.

Compare os resultados com a derivada analítica.

Solução

Vamos primeiro computar a seguda derivada analítica de $f(x)$ por computação simbólica.

```
from sympy.abc import x
sy.init_printing()

# função
f = 2**x/x

# derivada segunda
d2fdx2 = sy.diff(f,x,2)
d2fdx2
```

$$\frac{2^x \left(\log\left(2\right)^2 - \frac{2\log\left(2\right)}{x} + \frac{2}{x^2}\right)}{x}$$

```
# derivada em x = 2
d2fdx2_p2 = d2fdx2.subs(x,2)
```

```
# valor numérico
d2fdx2_p2_num = float(d2fdx2_p2)
d2fdx2_p2_num
```

0.5746116667165122

Agora, vamos montar uma função para a fórmula da segunda derivada a 3 pontos (supondo que os pontos dados sejam igualmente espaçados:

```
# função
def der_num_2(x1,x2,x3):
    d2fdx2_num = lambda x1,x2,x3: (f.subs(x,x1) - 2*f.subs(x,x2) + f.subs(x,x3))/(x2 -
↪ x1)**2
    return float(d2fdx2_num(x1,x2,x3))
```

Aplicamos a nossa aproximação numérica ao primeiro conjunto de pontos para estimar a derivada numérica no ponto $x = 2$ por diferença centrada.

```
df2_1 = der_num_2(1.8,2.0,2.2)
df2_1
```

0.5774817738923302

Em seguida, aplicamos a nossa aproximação numérica ao segundo conjunto de pontos:

```
df2_2 = der_num_2(1.9,2.0,2.1)
df2_2
```

0.5753244156644123

Esperamos que o segundo conjunto de pontos nos dê uma estimativa mais próxima para a derivada analítica já que os pontos estão mais próximos um do outro. Para verificar isto, vamos medir o erro relativo percentual entre as derivadas numéricas e a derivada analítica.

```
# erro relativo percentual
erp = lambda v,va: abs(v-va)/abs(v)*100

# erro para o grupo 1
print('erro relativo percentual 1: ' + str(erp(d2fdx2_p2_num,df2_1)) + '%' )

# erro para o grupo 2
print('erro relativo percentual 2: ' + str(erp(d2fdx2_p2_num,df2_2)) + '%' )
```

```
erro relativo percentual 1: 0.49948640831095764%
erro relativo percentual 2: 0.12404011077131777%
```

CHAPTER

THIRTYTWO

INTRODUÇÃO À SOLUÇÃO NUMÉRICA DE EDOS

Equações diferenciais ordinárias (EDOs) surgem em diversos problemas aplicados. Alguns exemplos:

- *Química*: decréscimo radioativo de carbono 14;
- *Engenharia*: queda da pressão atmosférica;
- *Economia*: precificação de ativos financeiros.

Nem sempre é possível obter soluções analíticas (forma fechada) para EDOs. Então, precisamos obter soluções aproximadas por meio de métodos numéricos.

No passado, muito esforço era empregado para se desenvolver métodos computacionais ótimos, mas a insuficiência de poder computacional era um entrave. Hoje em dia, com a evolução tecnológica, a capacidade computacional de alto desempenho permite que soluções numéricas sejam obtidas com menor esforço de processamento e margem de erro satisfatória. A seguir, faremos uma breve introdução teórica sobre modelos clássicos descritos por EDOs e a resolubilidade das equações.

32.1 Modelos clássicos

32.1.1 EDOs de primeira ordem

- **Crescimento e decrescimento**: modelo (de Malthus) utilizado em crescimento populacional, mor- talidade de espécies biológicas.

$$y'(t) = ky \Rightarrow y(t) = ce^{kt}, c, k \in \mathbb{R}$$

Interpretação: taxa de mudança da quantidade y é proporcional à própria quantidade ao longo do tempo. Se $k > 0$, temos uma lei de crescimento; se $k < 0$, temos uma lei de descrescimento (ou queda).

- **Lei do resfriamento de Newton:** modelo utilizado para determinar a troca de calor entre um corpo material e um meio externo.

$$T'(t) = k(T - T_\infty) \Rightarrow T(t) = T_0 + (T_\infty - T_0)e^{kt},$$

onde T é a *temperatura do corpo*, $k > 0 \in \mathbb{R}$ a *condutividade térmica* (dependente do material do corpo e nem sempre constante), T_∞ a *temperatura do ambiente* e T_0 a *temperatura inicial*.

Interpretação: taxa de mudança da temperatura é proporcional a diferença entre a temperatura do objeto e do ambiente com o qual troca calor.

32.1.2 EDOs de segunda ordem

- **Variação da quantidade de movimento em um sistema (2a. lei de Newton)**: modelo utilizado para descrever a perda de equilíbrio de sistemas mecânicos (cordas vibrantes, molas amortecidas, escoamentos de fluidos viscosos).

$$my''(t) = -by'(t) - ky + f(t),$$

onde m é a *massa*, y é um *deslocamento*, t o *tempo*, b > 0 uma *constante de amortecimento* (absorvedor de choque), $k > 0$ um *parâmetro da mola/empuxo* e $f(t)$ uma *força externa*.

Interpretação: taxa de mudança da quantidade de movimento do corpo é igual às forças aplicadas sobre o mesmo. A solução geral desta equação não homogênea será omitida aqui (cf. Weiglholfer and Lindsay, p.32).

- **2a. lei de Kirchhoff**: modelo usado em circuitos elétricos e eletromagnetismo.

$$LQ''(t) + RQ'(t) + \frac{1}{C}Q(t) = U(t),$$

onde $Q(t)$ é a *carga elétrica*, L é a *indutância*, R a *resistência*, C a *capacitância* e $U(t)$ a *força eletromotora* (tensão elétrica).

Interpretação: a força eletromotora (bateria, por exemplo) em qualquer circuito fechado equilibra todas as diferenças de potencial (d.d.p.) naquele circuito. Em outras palavras: em um circuito fechado, a soma de todas as d.d.p. é nula.

32.2 Teoria geral de resolubilidade de EDOs

32.2.1 Problema de Valor Inicial (PVI)

Um *problema de valor inicial* (PVI) é formado por uma EDO e uma *condição inicial*.

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases},$$

Acima, t é a *variável independente*, $y(t)$ é a *variável dependente* (solução da EDO) e t_0 é a *condição inicial*.

Exemplo: A EDO $y'(t) = -[y(t)]^2 + y(t)$ possui a chamada solução trivial $y(t) \equiv 0$ e a solução geral:

$$y(t) = \frac{1}{1 + ce^{-t}}$$

Observemos que $y(t)$ é indefinida quando $1 + ce^{-t} = 0$, ou $t = \ln(-c)$. Se $y(0) = y_0 \neq 0$ for uma condição inicial geral, $c = \frac{1}{y_0} - 1$ e teremos os seguintes resultados adicionais:

condição inicial	valores de c	existência de solução
$y_0 > 0$	$c > -1$	$0 \leq t < \infty$
$y_0 < 0$	$c < -1$	$0 < t < \ln(1 - y_0^{-1})$

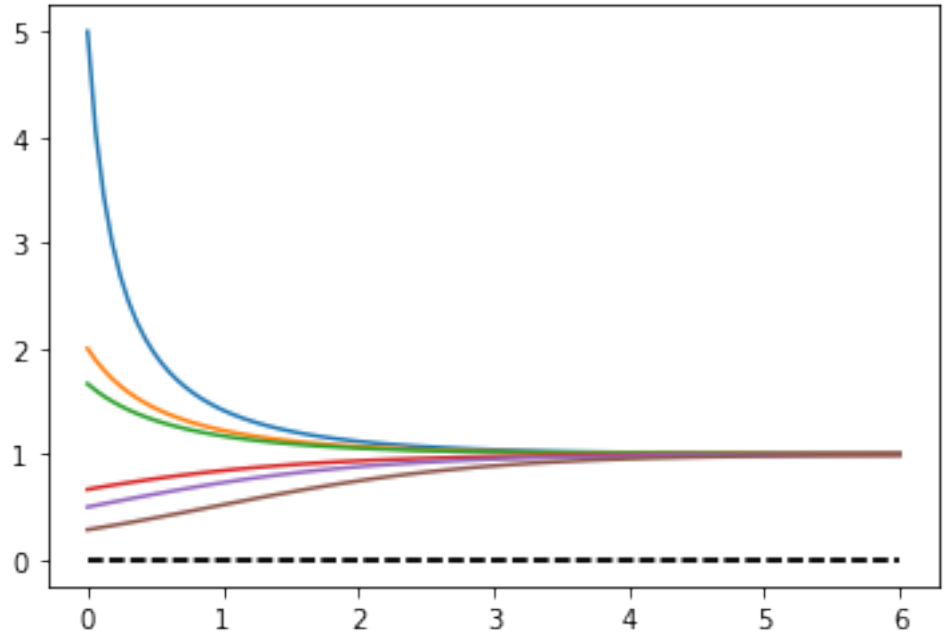
Abaixo vemos os gráficos de $Y(t)$ quando $c = \{-0.8, -0.5, -0.4, 0, 0.5, 1, 0, 2.5\}$ e para a solução trivial.

```
import numpy as np
import matplotlib.pyplot as plt

# variavel independente
t = np.linspace(0,6,100)

# condicao inicial
```

(continues on next page)

(continued from previous page)

```
C = [-0.8,-0.5,-0.4,0,0.5,1,0,2.5]

for c in C:

    # solucoes não triviais
    if c != 0:
        plt.plot(t,1./(1 + c*np.exp(-t)))

    # solucao trivial
    else:
        plt.plot(t,0*t,'k--')
```




```
import numpy as np
import matplotlib.pyplot as plt
```

CHAPTER

THIRTYTHREE

MÉTODO DE EULER

As considerações anteriores sobre malhas numéricas são fundamentais para sugerir a notação que utilizaremos para expressar os métodos numéricos que estudaremos. O primeiro deles, chamado *método de Euler*, é considerado o método numérico mais simples para resolver PVIs. Embora não seja muito eficiente, é o ponto de partida para a compreensão de uma enorme família de métodos.

Ao longo do texto, denotaremos por $y(t_n) = y_h(t_n) = y_n, \;\; n = 0, 1, 2, \ldots, N$ a solução aproximada de um PVI.

33.1 Definição do método de Euler

A derivação do método de Euler inicia-se com a seguinte aproximação para a derivada:

$$y'(t) \approx \frac{y(t+h) - y(t)}{h},$$

conhecida como *aproximação por diferença finita avançada (ou progressiva)*. Se aplicarmos esta definição ao nosso PVI padrão, em $t = t_n$ teremos $y'(t_n) = f(t_n, y(t_n))$, donde segue que

$$\frac{y(t_{n+1}) - y(t_n)}{h} \approx f(t_n, y(t_n))$$

$$y(t_{n+1}) \approx y(t_n) + h\, f(t_n, y(t_n)).$$

O método de Euler toma esta aproximação como exata, de modo que o esquema numérico resultante é

$$y_{n+1} = y_n + h\, f(t_n, y_n), \;\; 0 \leq n \leq N - 1.$$

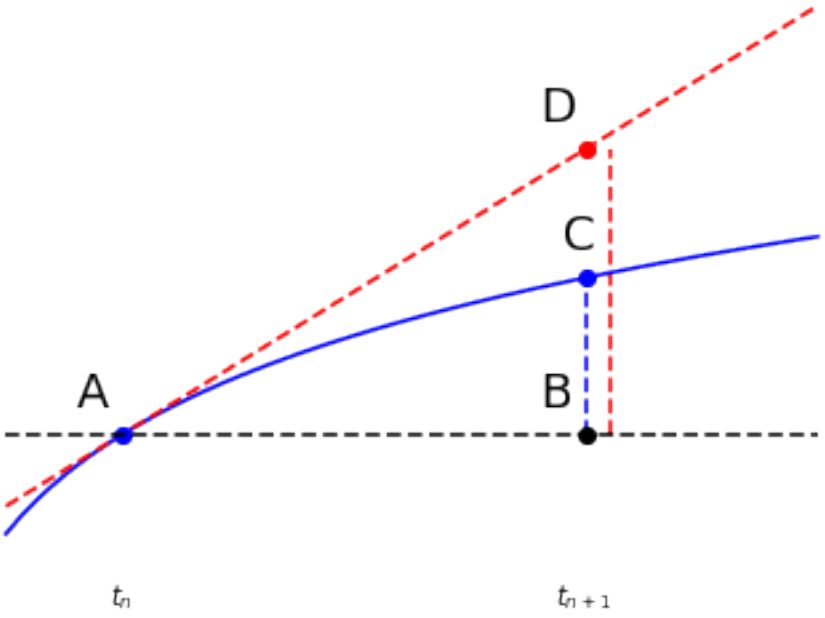
A estimativa inicial é $y_0 = Y_0$ ou alguma aproximação de Y_0. Quando Y_0 é obtido empiricamente, seu valor é conhecido apenas aproximadamente. A fórmula anterior permite o cálculo sequencial das iteradas do método de Euler $y_1, y_2, \ldots, y_n$, aproximações para os valores exatos de y nesses instantes.

33.2 Interpretação geométrica

A figura a seguir ajuda-nos a interpretar o método de Euler geometricamente. A aproximação numérica da curva exata (azul) é feita por meio de retas tangentes. O valor $y(t_{n+1})$ é erroneamente computado (comprimento BD) e excede o valor exato (comprimento BC) por uma quantidade (comprimento CD). Isto é, a equação da reta tangente a $(t_n, y(t_n))$ é $r(t) = y(t_n) + f(t_n, y(t_n))(t - t_n)$. Na verdade, $r(t_{n+1})$ coincide com o ponto D.

```
x = np.linspace(0.5,4,50)

plt.plot(x,np.log(x),'b')
x0 = 1.0
x1 = 3.0
plt.plot(x,np.log(x0) + x/x0 -1,'r--')
plt.plot(x0,np.log(x0),'ob')
plt.plot(x1,np.log(x1),'ob')
plt.plot(x1,0,'ok')
plt.plot(x,0*x,'k--')
plt.plot(x1,np.log(x0) + x1/x0 -1,'or')
plt.vlines(x1+0.1,ymin=0,ymax=np.log(x0) + x1/x0 -1,colors='r',linestyles='dashed')
plt.vlines(x1,ymin=0,ymax=np.log(x1),colors='b',linestyles='dashed')


xt = [x0,x1]
plt.box(False)
locs, labels = plt.xticks()
plt.xticks(xt, ('$t_n$','$t_{n+1}$'))
plt.tick_params(axis='both',width=0.0,labelleft=False)

fs = 18
plt.annotate('A',xy=(x0-0.2,np.log(x0)+0.2),fontsize=fs)
plt.annotate('B',xy=(x1-0.2,0+0.2),fontsize=fs)
plt.annotate('C',xy=(x1-0.1,np.log(x1)+0.2),fontsize=fs)
plt.annotate('D',xy=(x1-0.2,np.log(x0) + x1/x0 - 1 +0.2),fontsize=fs)
```

```
Text(2.8, 2.2, 'D')
```



Exemplo: A solução exata do PVI

$$\begin{cases} y'(t) = -y(t) \\ y(0) = 1 \end{cases}$$

é $y(t) = e^{-t}$. O método de Euler é dado por

$$y_{n+1} = y_n - hy_n = (1-h)y_n, \;\; n \geq 0,$$

com $Y_0 = 1$ e $t_n = nh$.

Para $h = 0.1$, temos, por exemplo

$$y_1 = (1-h)y_0 = 0.9(1) = 0.9$$

$$y_2 = (1-h)y_1 = 0.9(0.9) = 0.81$$

$$y_2 = (1-h)y_1 = 0.9(0.9) = 0.81,$$

cujos erros são

$$y_{h,1} - y_1 = e^{-0.1} - y_1 = 0.004837$$

$$y_{h,2} - y_2 = e^{-0.2} - y_2 = 0.008731$$

33.3 Implementação computacional

O seguinte código implementa o método de Euler explícito.

```
from numpy import *

def euler_expl(t0,tf,y0,h,fun):
    """
    Resolve o PVI y' = f(t,y), t0 <= t <= tf, y(t0) = y0
    com passo h usando o metodo de Euler explicito.

    Entrada:
        t0  - tempo inicial
        tf  - tempo final
        y0  - condicao inicial
        h   - passo
        fun - funcao f(t,y) (anonima)

    Saida:
        t   - nos da malha numerica
        y   - solucao aproximada
    """

    n = round((tf - t0)/h) + 1
    t = linspace(t0,t0+(n-1)*h,n)
    y = linspace(t0,t0+(n-1)*h,n)
    y = zeros((n,))

    y[0] = y0

    for i in range(1,n):
        y[i] = y[i-1] + h*f(t[i-1],y[i-1])

    return (t,y)
```

Exemplo: Resolva

$$\begin{cases} y'(t) = \frac{y(t)+t^2-2}{t+1} \\ y(0) = 2 \\ 0 \leq t \leq 6 \\ h = 0.1 \end{cases}$$

Defina $y_h(t)$ como a solução numérica, calcule o erro relativo e plote o gráfico de $y_h(t)$ juntamente com o da solução exata $y(t) = t^2 + 2t + 2 - 2(t+1)\ln(t+1)$

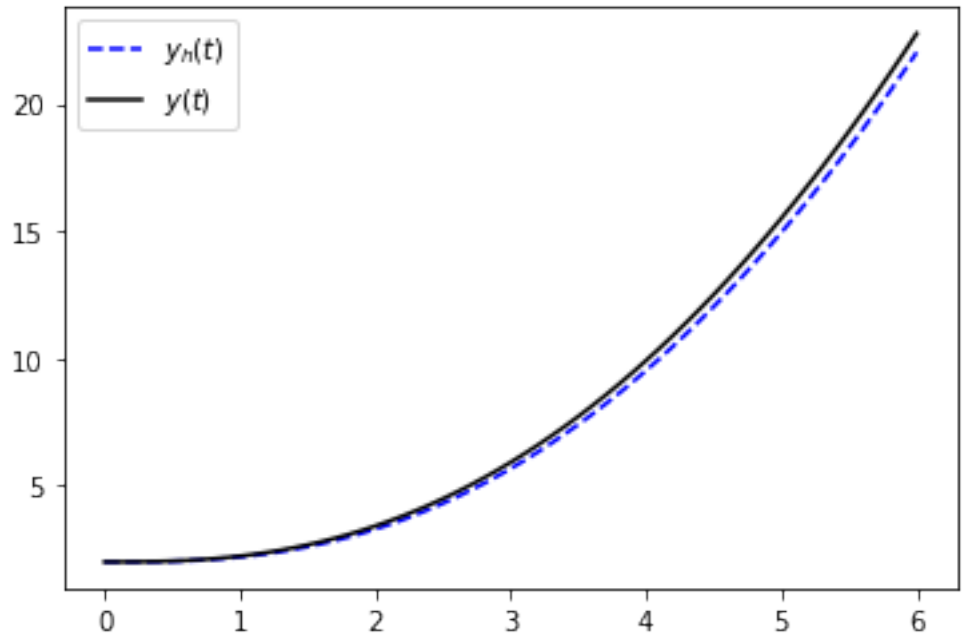
Solução: O processo iterativo de Euler será dado por

$$y_{n+1} = y_n + \frac{h(y_n + t_n^2 - 2)}{t_n + 1}, \;\; n \geq 0, \;\; y_0 = 2, t_n = nh.$$

Entretanto, vamos usar o nosso programa `euler_expl`.

```
# define funcao
f = lambda t,y: (y + t**2 - 2)/(t+1)

# invoca metodo
t0 = 0.0
tf = 6.0
y0 = 2.0
h = 0.1
t,y = euler_expl(t0,tf,y0,h,f)

# plota funcoes
yex = t**2 + 2*t + 2 - 2*(t+1)*log(t+1)
plt.plot(t,y,'b--',label='$y_h(t)$')
plt.plot(t,yex,'k',label='$y(t)$')
plt.legend()
```

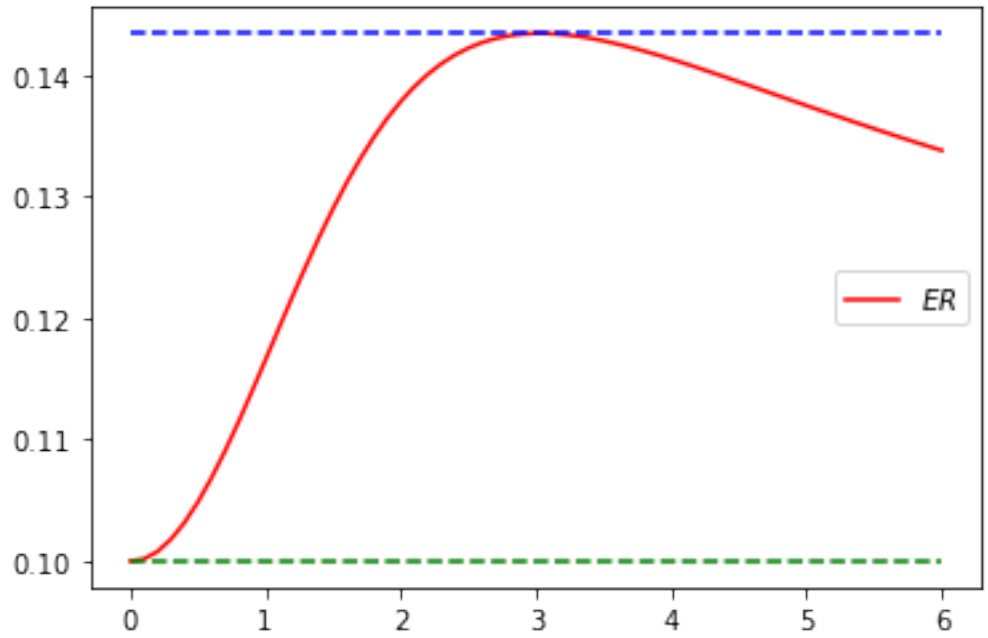
```
<matplotlib.legend.Legend at 0x7fd023af8b20>
```



Agora, vamos computar a curva do erro relativo e plotá-la.

```
def erro_relativo(y,yh):
    return abs(y-yh)/abs(yh)

e = erro_relativo(yex,y)+0.1

plt.plot(t,e,'r',label='$ER$')
plt.legend()
emax = np.max(e)
emin = np.min(e)

plt.plot(t,np.ones(t.shape)*emax,'b--')
plt.plot(t,np.ones(t.shape)*emin,'g--')
```

```
[<matplotlib.lines.Line2D at 0x7fd023bee880>]
```



Exercício: Resolva o PVI do exemplo anterior usando por computador para $h = 0.2, 0.1, 0.05$. Produza um `script` para imprimir, para cada h, os dados da saída na forma da tabela a seguir:

t	$y_h(t)$	EA	ER
⋮	⋮	⋮	⋮

33.4 Análise de erro

A análise de erro para o método de Euler tem o propósito de entender como o método funciona para que se possa estimar o erro ao usá-lo e, possivelmente, acelerar sua convergência. Procedimentos similares são aplicáveis a métodos numéricos mais eficientes.

Para a análise, assumiremos que o PVI padrão possui solução única $y(t)$ em $[t_0, b]$ e que esta solução tem uma segunda derivada $y''(t)$ limitada neste intervalo.

Consideremos a série de Taylor para aproximar $y(t_{n+1})$:

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(\xi_n), \;\; t_n \leq \xi_n \leq t_{n+1}$$

Uma vez que $y(t)$ satisfaz a EDO, temos que

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + \frac{h^2}{2}y''(\xi_n), \;\; t_n \leq \xi_n \leq t_{n+1}$$

O termo $T_{n+1} = \frac{h^2}{2}y''(\xi_n)$, *erro de truncamento* para o método de Euler, é o erro na aproximação

$$y(t_{n+1}) \approx y(t_n) + hy'(t_n).$$

Assim, tendo em vista que

$$y_{n+1} \approx y_n + hf(t_n, y_n),$$

subtraímos as equações para obter

$$y(t_{n+1}) - y_{n+1} \approx y(t_n) - y_n + h[f(t_n, y(t_n)) - f(t_n, y_n)] + \frac{h^2}{2}y''(\xi_n),$$

mostrando que o error em y_{n+1} consiste de duas partes:

- **erro propagado:** $T_{n+1} = y(t_n) - y_n + h[f(t_n, y(t_n)) - f(t_n, y_n)]$.
- **erro de truncamento:** $T_{n+1} = \frac{h^2}{2}y''(\xi_n)$.

O erro propagado pode ser simplificado pelo teorema do valor médio:

$$f(t_n, y(t_n)) - f(t_n, y_n) = \frac{\partial f(t_n, \zeta_n)}{\partial y}[y(t_n) - y_n], \;\; y(t_n) \leq \zeta_n < y_n$$

Se definirmos o erro $e_k = y(t_k) - y_k, k \geq 0$, podemos reescrever a última equação como

$$e_{n+1} = \left[1 + h\frac{\partial f(t_n, \zeta_n)}{\partial y}\right] e_n + T_{n+1},$$

assim obtendo uma equação geral para análise de erro do método de Euler.

33.4.1 Teoremas para limite de erro

Teorema: Suponha que $f(t, y)$ seja definida em um conjunto convexo $D \subset \mathbb{R}^2$. Se existe uma constante $L > 0$ com

$$\left|\frac{\partial f}{\partial y}(t, y)\right| \leq L, \;\; \forall (t, y) \in D$$

Teorema (limite de erro): Suponha que f seja contínua e satisfaça a condição de Lipschitz com constante L em $D = \{(t, y);\ a \leq t \leq b\}$ e que exista uma constante M com $|y''(t)| \leq M, \;\; \forall t \in [a, b]$. Seja $y(t)$ única solução do PVI

$$y' = f(t, y), \quad y(a) = y_0, \quad t_0 \leq t \leq b$$

e sejam $w_0, w_1, \ldots, w_N$ aproximações geradas pelo método de Euler para algum inteiro positivo N. Então, para cada $i = 0, 1, 2, \ldots, N$,

$$|y(t_i) - w_i| \leq \frac{hM}{2L}[e^{L(t_i - a)} - 1].$$

O último teorema fornece um limitante de erro para o método de Euler. Ele evidencia a dependência linear do tamanho de passo h. Logo, à medida que $h- > 0$, uma maior precisão nas aproximações deve ser obtida.

Exemplo: Consideremos o PVI

$$\begin{cases} y'(t) = y - t^2 + 1 \\ y(0) = 0.5 \\ 0 \leq t \leq 2 \\ h = 0.2 \end{cases}$$

Uma vez que $f(t, y) = y - t^2 + 1$, temos $\dfrac{\partial f}{\partial y}(t, y) = 1, \forall y$. Assim, $L = 1$. A solução exata deste problema é $y(t) = (t + 1)^2 - 0.5e^t$, de modo que $y''(t) = 2 - 0.5e^t$ e $|y''(t)| \leq 0.5e^2 - 2 = M, \ \ \forall t \in [0, 2]$. Pelo teorema do limite de erro, temos que

$$|y_i - w_i| \leq 0.1(0.5e^2 - 2)(e^{t_i} - 1)$$

```
def f(t,y):
    return y - t**2 + 1

# solucao numerica
t,y = euler_expl(0,2,0.5,0.2,f)

# solucao exata
yex = (t+1)**2 - 0.5*np.exp(t)

# erro
erro = np.abs(y - yex)

# limite de erro
lim_erro = 0.1*(0.5*np.exp(1)**2 - 2)*(np.exp(t) - 1.)

# tabela

print("Imprimindo comparação...\n\n ti |    ei    |   Ei\n")
for i in range(len(y)):
    print("{0:.1f} | {1:0.5f} | {2:0.5f}\n".format(t[i],erro[i],lim_erro[i]))
```

```
Imprimindo comparação...

 ti |    ei    |   Ei

0.0 | 0.00000 | 0.00000

0.2 | 0.02930 | 0.03752

0.4 | 0.06209 | 0.08334

0.6 | 0.09854 | 0.13931

0.8 | 0.13875 | 0.20767

1.0 | 0.18268 | 0.29117

1.2 | 0.23013 | 0.39315
```

(continues on next page)

(continued from previous page)

```
1.4 | 0.28063 | 0.51771

1.6 | 0.33336 | 0.66985

1.8 | 0.38702 | 0.85568

2.0 | 0.43969 | 1.08264
```

33.5 Problemas

1. Resolva os seguintes problemas usando o método de Euler com passos $h = 0.2, 0.1, 0.05$. Compute o erro relativo usando a solução exata $y(t)$. Para valores selecionados de t, observe a razão com que o erro diminui à medida que h é reduzido pela metade. a. $y'(t) = [\cos(y(t))]^2, 0 \leq t \leq 10, \quad y(0) = 0; \quad y(t) = \tan^{-1}(t)$

b. $y'(t) = \frac{1}{1+t^2} - 2[y(t)]^2, 0 \leq t \leq 10, \quad y(0) = 0; \quad y(t) = \frac{t}{1+t^2}$

c. $y'(t) = \frac{1}{4}y(t)\left[1 - \frac{1}{20}y(t)\right], 0 \leq t \leq 20, \quad y(0) = 1; \quad y(t) = \frac{20}{1+19e^{-t/4}}$

d. $y'(t) = -[y(t)]^2, y(t) = \frac{1}{t}, 1 \leq t \leq 10, \quad y(1) = 1 \quad y(t) = \frac{1}{t}$

1. Considere o problema linear

$$y'(t) = \lambda y(t) + (1 - \lambda)\cos(t) - (1 + \lambda)\text{sen}(t), \quad y(0) = 1.$$

A solução exata é $y(t) = \text{sen}(t) + \cos(t)$. Resolva este problema usando o método de Euler com vários valores de λ e h, para $0 \leq t \leq 10$. Comente sobre os resultados.

a. $\lambda = -1; \;\; h = 0.5, 0.25, 0.125.$

b. $\lambda = 1; \;\; h = 0.5, 0.25, 0.125.$

c. $\lambda = -5; h = 0.5, 0.25, 0.125, 0.0625.$

d. $\lambda = 5; h = 0.125, 0.0625.$

1. Faça uma análise do erro obtido pelo método de Euler ao ser resolver o caso a. do problema 2, $h = 0.25$.

CHAPTER

THIRTYFOUR

MÉTODOS DE TAYLOR DE ORDEM SUPERIOR

Métodos que usam o desenvolvimento em série de Taylor de $y(t)$ teoricamente fornecem solução para qualquer ED. Sob o ponto de vista computacional, Métodos de Taylor de ordem mais elevada são inaceitáveis, pois, exceto uma classe restrita de funções, o cálculo das derivadas totais é complicado.

Estes métodos são obtidos retendo-se termos de ordem superior na série de Taylor. Por sua vez, o Método de Euler é um caso particular, como veremos a seguir, já que tem os termos de ordem ≥ 2 truncados na série.

Suponhamos que $y(t)$ a solução para o PVI $\begin{cases} y' = f(t,y) \\ a \leq t \leq b \\ y(a) = \alpha \end{cases}$ $declasse$\mathcal{C}^{n+1}.$Hajavistaqueasriede Taylorde$y(t)$emrelac

como

$$y(t_{i+1}) = y(t_i) + hy'(t) + \frac{h^2}{2}y''(t_i) + \cdots + \frac{h^n}{n!}y^{(n)}(t_i) + \frac{h^{n+1}}{(n+1)!}y^{(n+1)}(\xi_i),$$

para $\xi_i \in (t_i, t_{i+1})$.

A diferenciação sucessiva de $y(t)$ fornece

$$\begin{aligned} y'(t) &= f(t, y(t)) \\ y''(t) &= f'(t, y(t)) \\ &\vdots \\ y^k(t) &= f^{(k-1)}(t, y(t)) \end{aligned}$$

Substituindo-as na série de Taylor, temos:

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}f'(t_i, y(t_i)) + \cdots + \frac{h^n}{n!}f^{(n-1)}(t_i, y(t_i)) + \frac{h^{n+1}}{(n+1)!}f^{(n)}(\xi_i, y(\xi_i))$$

Excluindo o termo de resto envolvendo ξ, obtemos o **Método de Taylor de ordem** n através do seguinte processo iterativo

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + T^{(n)}(t_i, w_i), \quad i = 0, 1, 2, \ldots, N \end{aligned}$$

em que

$$T^{(n)}(t_i, w_i) = f(t_i, w_i) + \frac{h}{2}f'(t_i, y(t_i)) + \cdots + \frac{h^{n-1}}{n!}f^{(n-1)}(t_i, w_i).$$

Logo, vemos que o método de Euler é o Método de Taylor de ordem 1. As derivadas sucessivas podem ser calculadas pela Regra da Cadeia. Por exemplo,

$$y' = f(t,y) \Rightarrow y'' = f_t(t,y) + f_y(t,y)y' = f_t(t,y) + f_y(t,y)f(t,y).$$

Todavia, os métodos da família Runge-Kutta são alternativas numéricas melhores para métodos de Taylor pois simulam o efeito das derivadas a partir de cálculos médios que não necessitam de derivadas analíticas. Estudaremos métodos de Runge-Kutta em breve.

Exemplo: Aplique o Método de Taylor de ordem 2 ao PVI

$$\begin{cases} y' = y - t^2 + 1 \\ 0 \leq t \leq 2 \\ y(0) = 0.5. \end{cases}$$

Para o método de ordem 2, precisamos da primeira derivada de $f(t, y(t)) = y(t) - t^2 + 1$ em relação a t. Então,

$$f'(t, y(t)) = \frac{d}{dt}(y - t^2 + 1) = y' - 2t = (y - t^2 + 1) - 2t,$$

de modo que

$$T^{(2)}(t_i, w_i) = f(t_i, w_i) + \frac{h}{2} f'(t_i, w_i) = w_i - t_i^2 + 1 + \frac{h}{2}(w_i - t_i^2 + 1 - 2t_i) = \\ = \left(1 + \frac{h}{2}\right)(w_i - t_i^2 + 1) - ht_i$$

Como $N = 10$, temos $h = 0.2$ e $t_i = 0.2i, \forall i = 1, 2, ..., 10$. Assim, o método de segunda ordem torna-se

$$w_0 = 0.5$$
$$w_{i+1} = w_i + h\left[\left(1 + \frac{h}{2}\right)(w_i - t_i^2 + 1) - ht_i\right] =$$
$$= w_i + 0.2\left[\left(1 + \frac{0.2}{2}\right)(w_i - 0.04i^2 + 1) - 0.04i\right]$$
$$= 1.22w_i - 0.0088i^2 - 0.008i + 0.22.$$

Os dois primeiros passos dão a aproximação:

$$y(0.2) \approx w_1 = 1.22(0.5) - 0.0088(0)^2 - 0.008(0) + 0.22 = 0.83$$

$$y(0.4) \approx w_2 = 1.22(0.83) - 0.0088(0.2)^2 - 0.008(0.2) + 0.22 = 1.2158$$

$$(...)$$

Os demais passos seguem da mesma forma

34.1 Métodos de Runge-Kutta

O objetivo principal dos métodos de Runge-Kutta (RK) é imitar o comportamento de $f(t, y)$ avaliando-a em vários pontos "abstratos" dentro de um mesmo passo numérico.

Esquemas do tipo RK são usados para reter precisão e substituir aproximações de baixa ou alta ordem via séries de Taylor. São populares na resolução de PVIs e mais simples de programar do que os métodos de Taylor.

34.1.1 Forma geral

A forma geral de um método RK é dada por:

$$\begin{cases} w_{n+1} & = \\ w_n + hF(t_n, w_n; h), \ \ n \geq 0 & \\ w_0 & = \\ y_0 \end{cases}$$

O termo $F(t_n, w_n; h)$ representa uma *inclinação média*, de maneira que, informalmente, métodos RK seja refraseados como:

valor futuro = valor atual + passo x inclinação média.

Se fizéssemos uma analogia com o movimento uniforme da física, w seria uma posição do espaço, h o tempo e F a velocidade, resultando em

posição final = posição inicial + tempo x velocidade

ou, equivalentemente, $s_f = s_0 + vt$.

34.1.2 Métodos de Runge-Kutta de 2a. ordem

Para métodos RK2, a inclinação média é dada pela expressão

$$F(t, w; h) = b_1 f(t, w) + b_2 f(t + \alpha h, w + \beta h f(t, w),$$

em que $\alpha, \beta, b_1, b_2 \in \mathbb{R}$ são constantes a serem determinadas de modo que atinjamos um erro de truncamento

$$T_{n+1}(w) = w_{n+1} - [w_n + hF(t_n, w_n; h)] \equiv \mathcal{O}(h^3),$$

i.e., seja de terceira ordem.

Abaixo, vamos resolver o PVI:

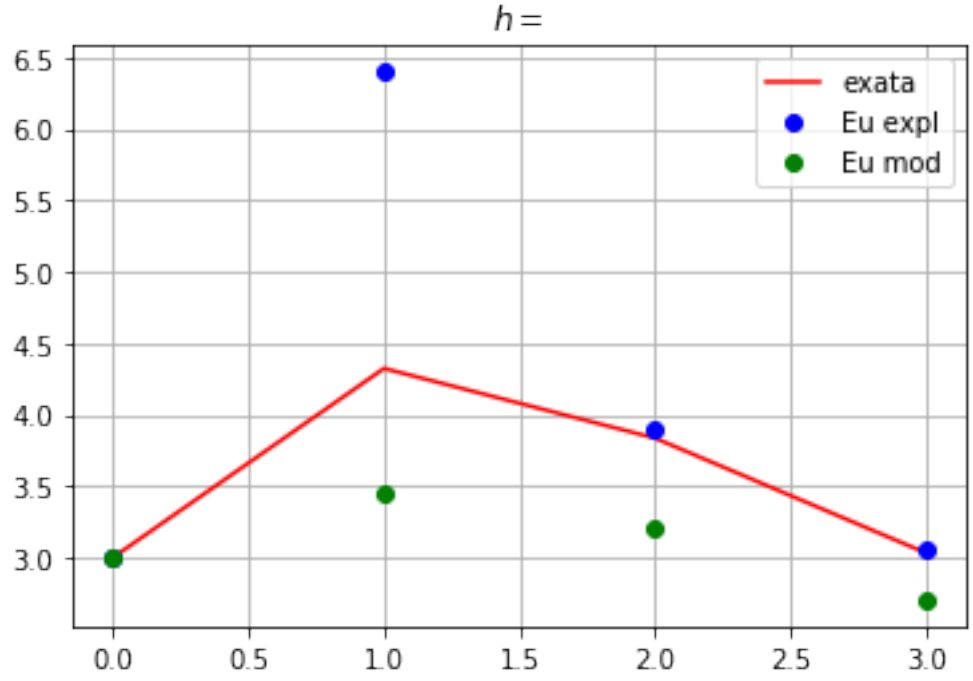
\begin{cases} y' = -1.2y + 7e^(-0.3x) \ y(0) = 3 \ 0 < x \leq 2 \ h = 1,0.5,0.25,0.1 \end{cases}

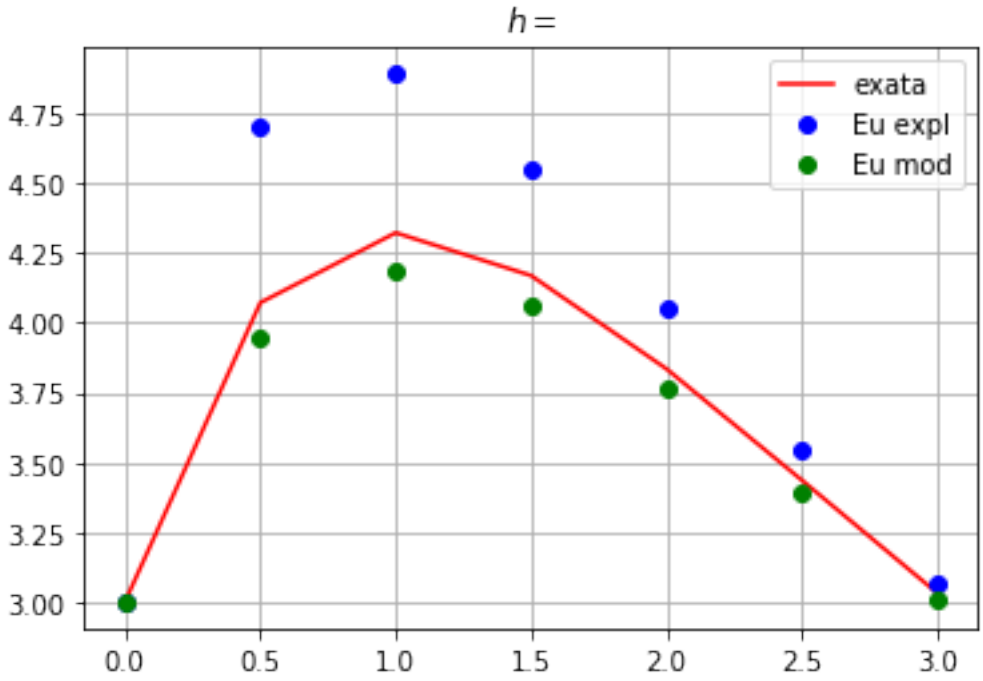
```
# definição
e = np.exp(1)

# dados de entrada
a = 0
b = 3.0
ns = [4,7,13,31]
w0 = 3
ode = '-1.2*y + 7*e**(-0.3*x)'

# soluções numéricas

for n in ns:
    # MEE
    x,we = ode_euler_expl(ode,a,b,n,w0)

    # MEM
    x,wm = ode_euler_mod(ode,a,b,n,w0)

    # conversao de dados
```

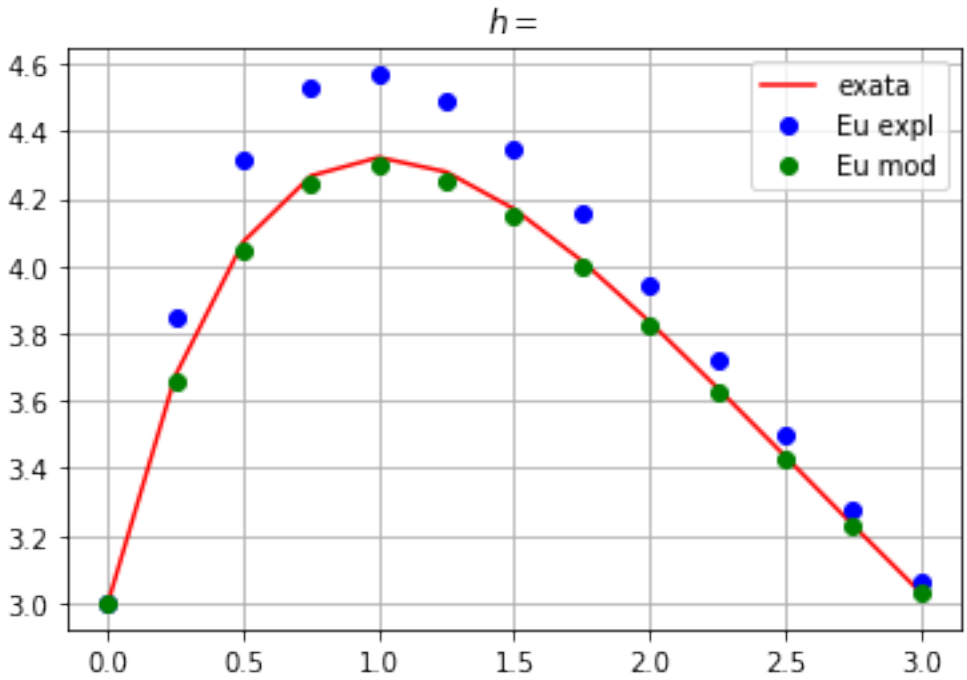
(continues on next page)

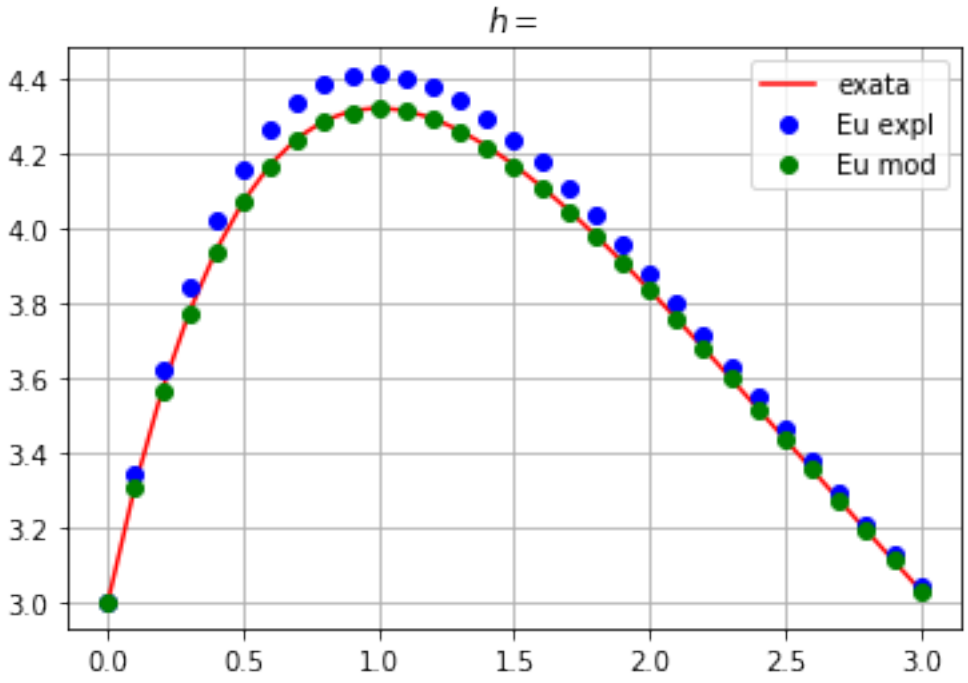
(continued from previous page)

```
    x = np.asarray(x)
    we = np.asarray(we)
    wm = np.asarray(wm)

    # solução exata
    y = 70/9*e**(-0.3*x) - 43/9*e**(-1.2*x)

    # curvas
    plt.figure()
    plt.plot(x,y,'r',label='exata')
    plt.plot(x,we,'bo',label='Eu expl')
    plt.plot(x,wm,'go',label='Eu mod')
    plt.legend()
    tit = '$h = ' + str((b-a)/(n-1)) + '$'
    plt.title('$h=$')
    plt.grid()
```




h =
exata
Eu expl
Eu mod
4.75
4.50
4.25
4.00
3.75
3.50
3.25
3.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0


h =
exata
Eu expl
Eu mod
4.6
4.4
4.2
4.0
3.8
3.6
3.4
3.2
3.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0


h =
exata
Eu expl
Eu mod
4.4
4.2
4.0
3.8
3.6
3.4
3.2
3.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Part VIII

Code sessions

CHAPTER

THIRTYFIVE

CODE SESSION 1

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

35.1 Determinação de raízes

`bisect`

A função `bisect` localiza a raiz de uma função dentro de um intervalo dado usando o método da bisseção. Os argumentos de entrada obrigatórios desta função são:

1. a função-alvo `f` (contínua)
2. o limite esquerdo `a`
3. o limite direito `b`

Parâmetros opcionais relevantes são:

- `xtol`: tolerância (padrão: 2e-12)
- `maxiter`: número máximo de iterações (padrão: 100)
- `disp`: mostra erro se algoritmo não convergir (padrão: True)

O argumento de saída é:

- `x0`: a estimativa para a raiz de `f`

Como importá-la?

```
from scipy.optimize import bisect
```


```
from scipy.optimize import bisect
```

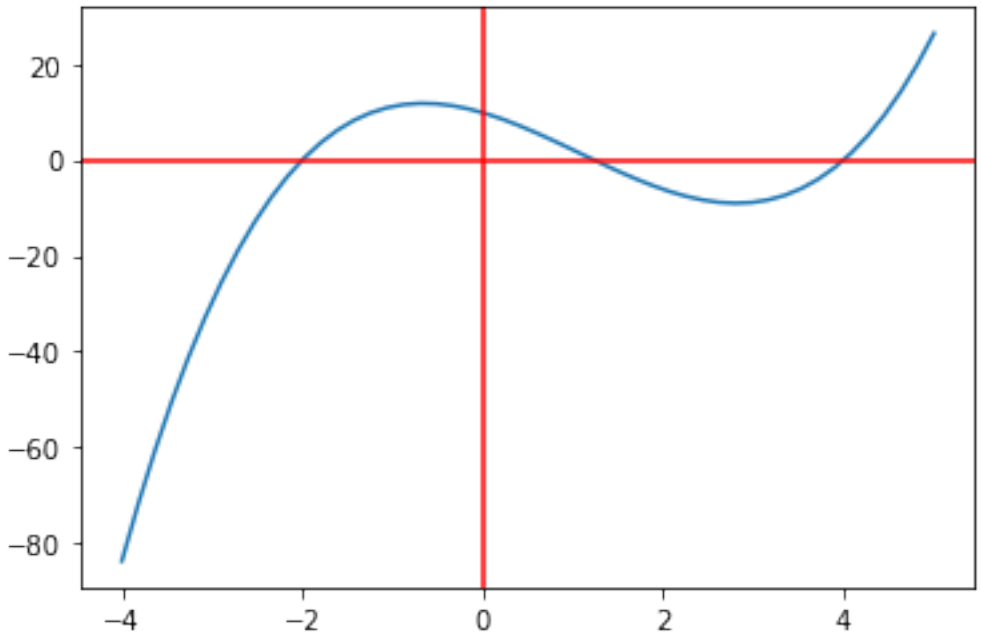
35.1.1 Problema 1

Encontre a menor raiz positiva (real) de $x^3 - 3.23x^2 - 5.54x + 9.84 = 0$ pelo método da bisseção.

Resolução

```
# função
def f(x):
    return x**3 - 3.23*x**2 - 5.54*x + 9.84
```

```
# analise gráfica
x = np.linspace(-4,5)
plt.plot(x,f(x));
plt.axhline(y=0,color='r');
plt.axvline(x=0,color='r');
```



Pelo gráfico, vemos que a menor raiz positiva está localizada no intervalo $(0, 2]$. Vamos determiná-la utilizando este intervalo de confinamento.

```
# resolução com bisect

x = bisect(f,0,2) # raiz

print('Raiz: x = %f' % x)
```

```
Raiz: x = 1.230000
```

35.1.2 Problema 2

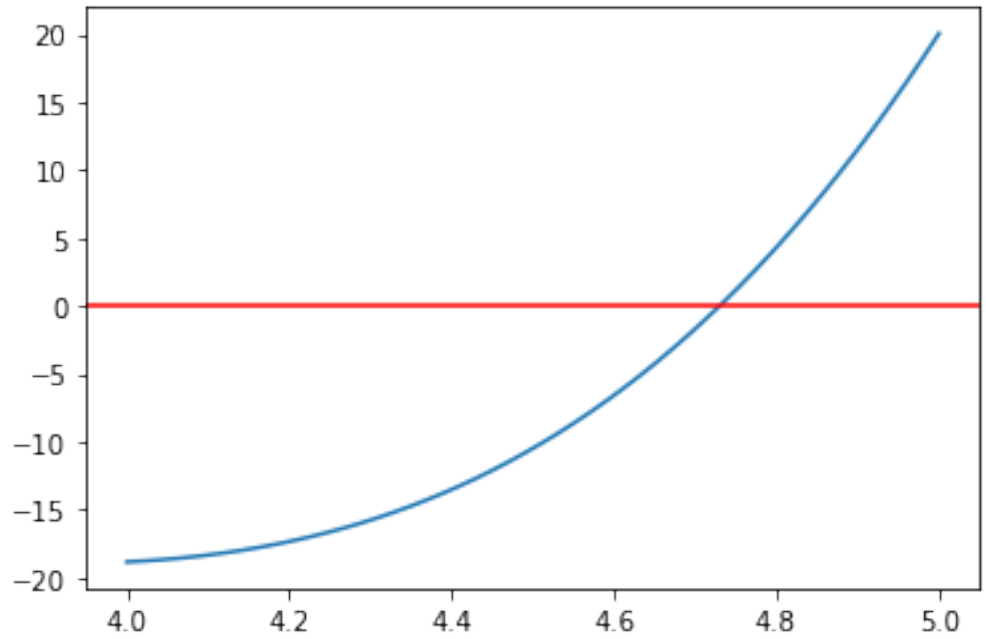
Determine a menor raiz não nula positiva de $\cosh(x)\cos(x) - 1 = 0$ dentro do intervalo $(4, 5)$.

Resolução

Sigamos o procedimento aprendido com `bisect`.

```
# função
def f(x):
    return np.cosh(x)*np.cos(x) - 1
```

```
# analise gráfica
x = np.linspace(4,5)
plt.plot(x,f(x));
plt.axhline(y=0,color='r');
```




```
# resolução com bisect

x = bisect(f,4,5) # raiz

print('Raiz: x = %f' % x)
```

```
Raiz: x = 4.730041
```

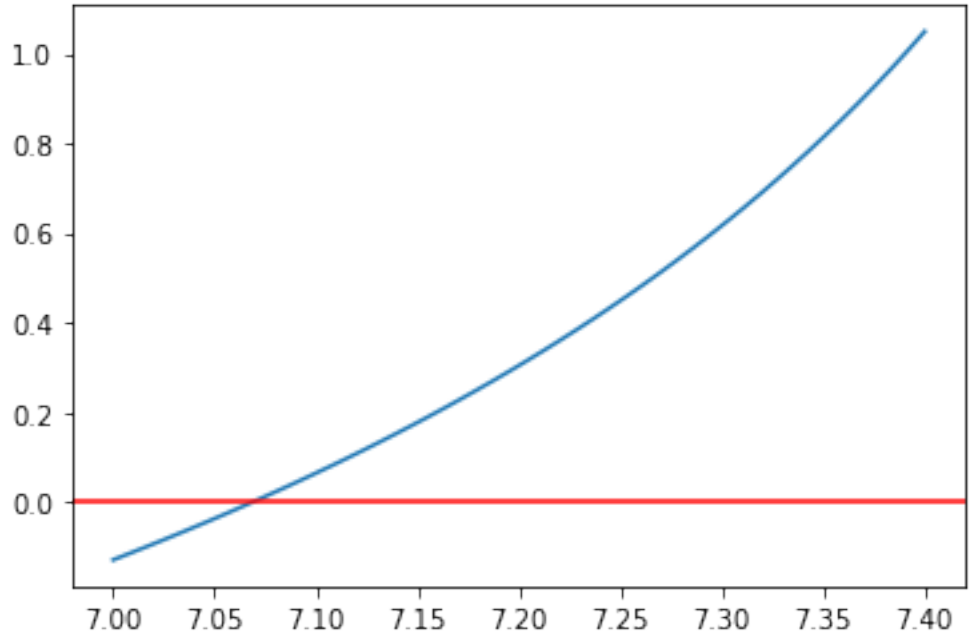
35.1.3 Problema 3

Uma raiz da equação $\tan(x) - \tanh(x) = 0$ encontra-se em $(7.0, 7.4)$. Determine esta raiz com três casas decimais de precisão pelo método da bisseção.

Resolução

```
# função
def f(x):
    return np.tan(x) - np.tanh(x)
```

```
# analise gráfica
x = np.linspace(7,7.4)
plt.plot(x,f(x));
plt.axhline(y=0,color='r');
```



Para obter as 3 casas decimas, vamos imprimir o valor final com 3 casas decimais.

```
x = bisect(f,7,7.4) # raiz

print('Raiz: x = %.3f' % x)
```

```
Raiz: x = 7.069
```

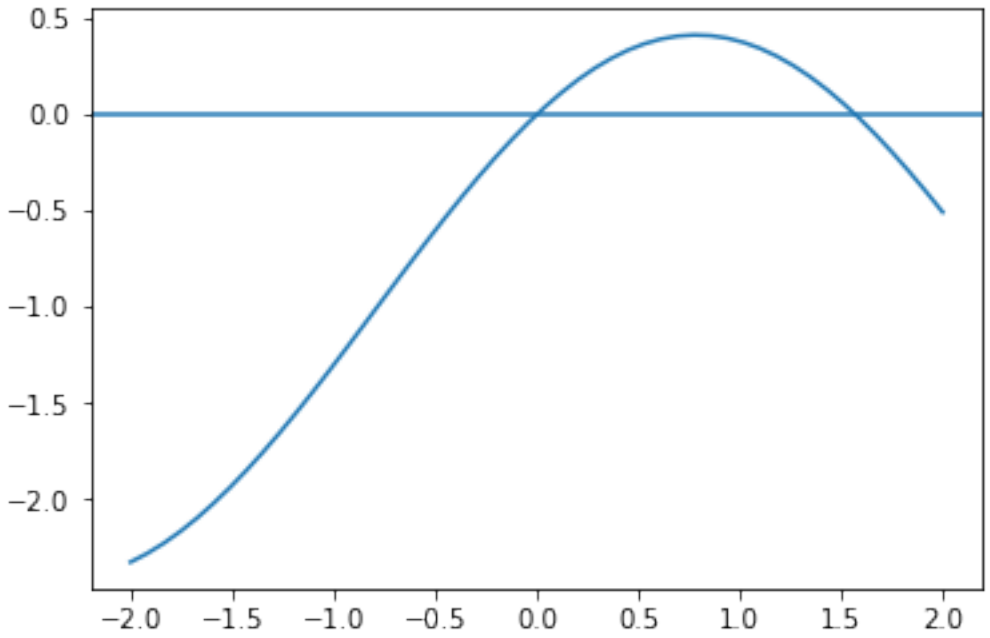
35.1.4 Problema 4

Determine as raízes de $\text{sen}(x) + 3\cos(x) - 1 = 0$ no intervalo $(-2, 2)$.

Resolução

```
# função
def f(x):
    return np.sin(x) + np.cos(x) - 1
```

```
# analise gráfica
x = np.linspace(-2,2)
plt.plot(x,f(x));
plt.axhline(y=0);
```



A análise gráfica mostra duas raízes. Vamos encontrar uma de cada vez.

```
# resolução com bisect

x1 = bisect(f,-2,1) # raiz 1
x2 = bisect(f,1,2) # raiz 2

print('Raízes: x1 = %f; x2 = %f' % (x1,x2))
```

```
Raízes: x1 = -0.000000; x2 = 1.570796
```

35.1.5 Problema 5

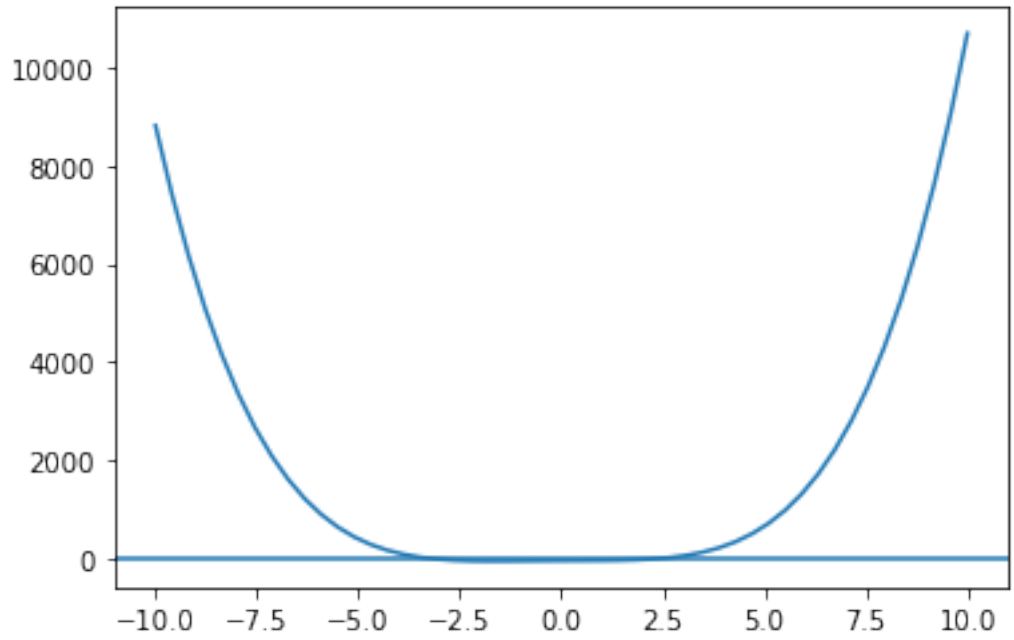
Determine todas as raízes reais de $P(x) = x^4 + 0.9x^3 - 2.3x^2 + 3.6x - 25.2$

Resolução

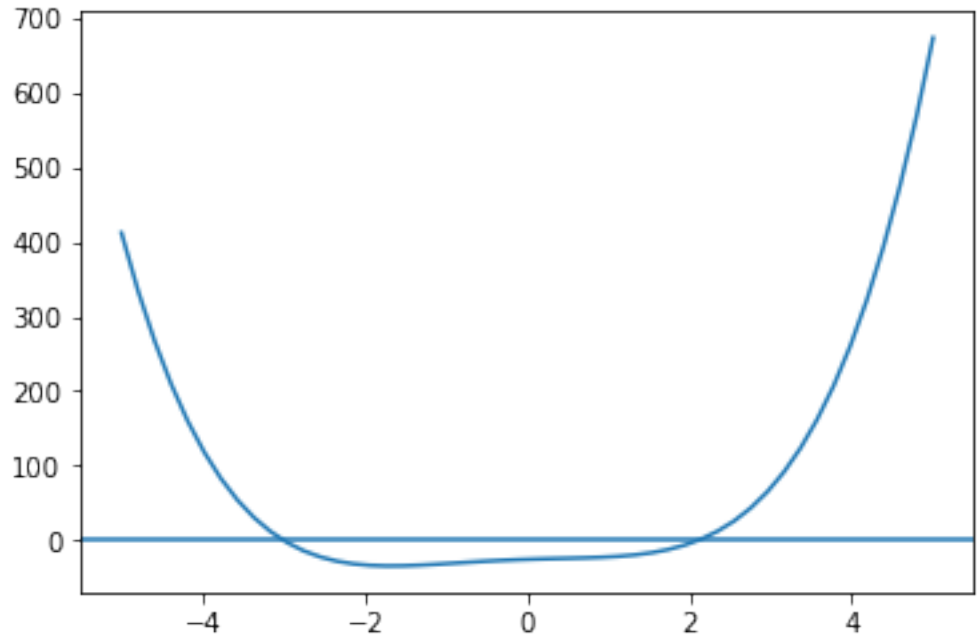
```
# função
def P(x):
    return x**4 + 0.9*x**3 - 2.3*x**2 + 3.6*x - 25.2
```

```
# define função para analise gráfica
def analise_grafica(xrange,f):
    plt.plot(xrange,f(xrange));
    plt.axhline(y=0);
```

```
# analise gráfica 1
xrange = np.linspace(-10,10)
analise_grafica(xrange,P)
```




```
# refinamento
xrange = np.linspace(-5,5)
analise_grafica(xrange,P)
```




```
# resolução com bisect

x1 = bisect(P,-4,-2) # raiz 1
x2 = bisect(P,1,3) # raiz 2

print('Raízes: x1 = %f; x2 = %f' % (x1,x2))
```

```
Raízes: x1 = -3.000000; x2 = 2.100000
```

35.1.6 Problema 6

Um jogador de futebol americano está prestes a fazer um lançamento para outro jogador de seu time. O lançador tem uma altura de 1,82 m e o outro jogador está afastado de 18,2 m. A expressão que descreve o movimento da bola é a familiar equação da física que descreve o movimento de um projétil:

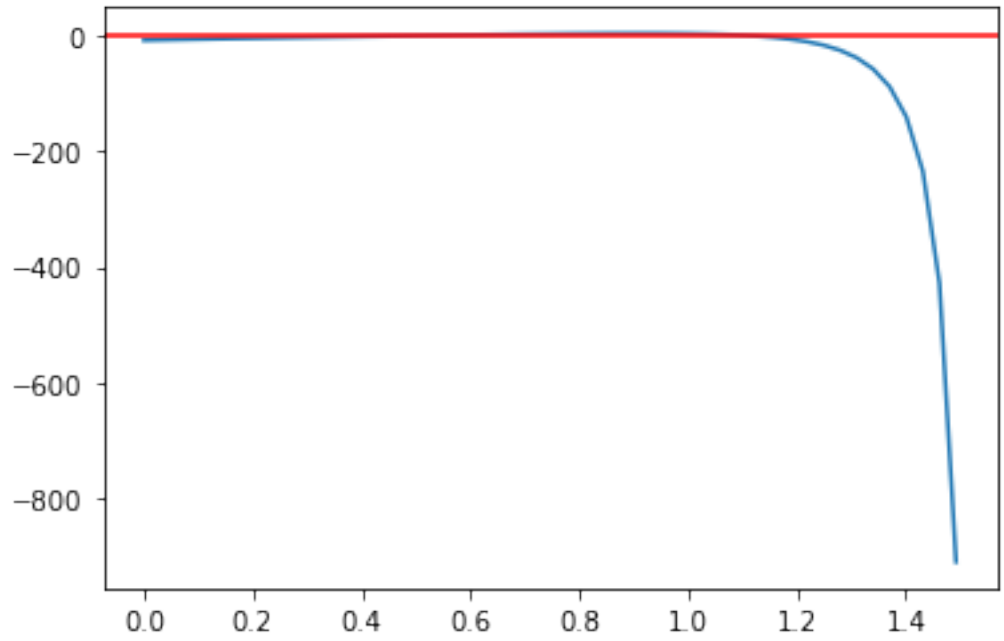
$$y = x\tan(\theta) - \frac{1}{2}\frac{x^2 g}{v_0^2}\frac{1}{\cos^2(\theta)} + h,$$

onde x e y são as distâncias horizontal e verical, respectivamente, $g = 9,8\,m/s^2$ é a aceleração da gravidade, v_0 é a velocidade inicial da bola quando deixa a mão do lançador e θ é o Ângulo que a bola faz com o eixo horizontal nesse mesmo instante. Para $v_0 = 15,2\,m/s$, $x = 18,2\,m$, $h = 1,82\,m$ e $y = 2,1\,m$, determine o ângulo θ no qual o jogador deve lançar a bola.

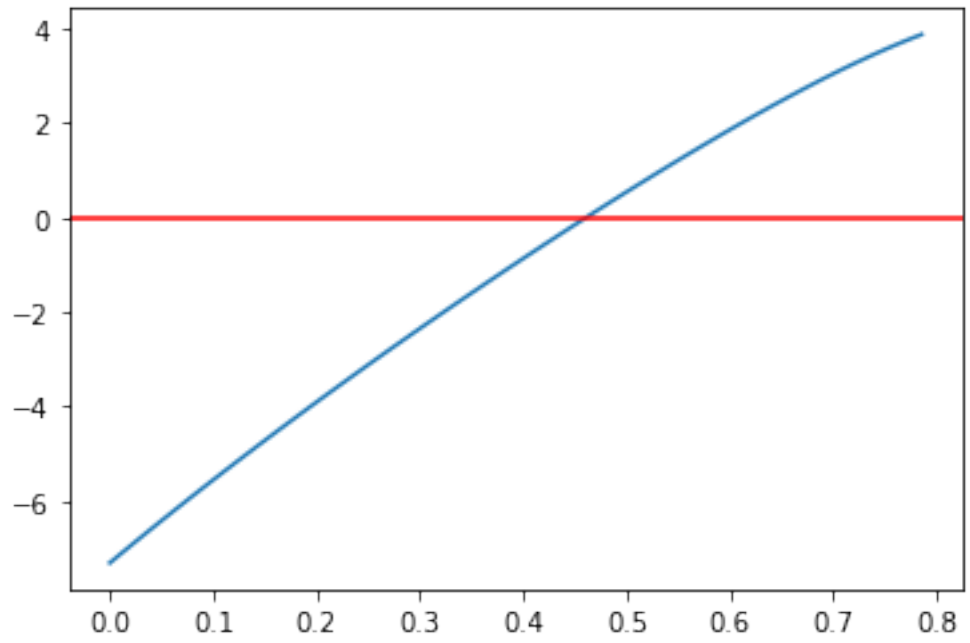
Resolução

```
# parâmetros do problema
v0 = 15.2
x = 18.2
h = 1.82
y = 2.1
g = 9.8

# função f(theta) = 0
f = lambda theta: x*np.tan(theta) - 0.5*(x**2*g/v0**2)*(1/(np.cos(theta)**2)) + h - y
```

```
# análise gráfica
th = np.linspace(0,0.95*np.pi/2,50)
plt.plot(th,f(th));
plt.axhline(y=0,color='r');
```




```
# análise gráfica - 2
th = np.linspace(0,np.pi/4,50)
plt.plot(th,f(th));
plt.axhline(y=0,color='r');
```




```
# resolução por bisseção
xr = bisect(f,0.1,0.6)
print('Ângulo de lançamento: %.2f graus' % np.rad2deg(xr))
```

```
Ângulo de lançamento: 26.41 graus
```

35.1.7 Problema 7

A equação de Bernoulli para o escoamento de um fluido em um canal aberto com um pequeno ressalto é

$$\frac{Q^2}{2gb^2h_0^2} + h_0 = \frac{Q^2}{2gb^2h^2} + h + H,$$

onde $Q = 1.2\,m^3/s$ é a vazão volumétrica, $g = 9.81\,m/s^2$ é a aceleração gravitacional, $b = 1.8\,m$ a largura do canal, $h_0 = 0.6\,m$ o nível da água à montante, $H = 0.075\,m$ a altura do ressalto e h o nível da água acima do ressalto. Determine h.

Resolução

Para este problema, definiremos duas funções, uma auxiliar, que chamaremos `a`, e a função `f(h)` que reescreve a equação de Bernoulli acima em função de h.

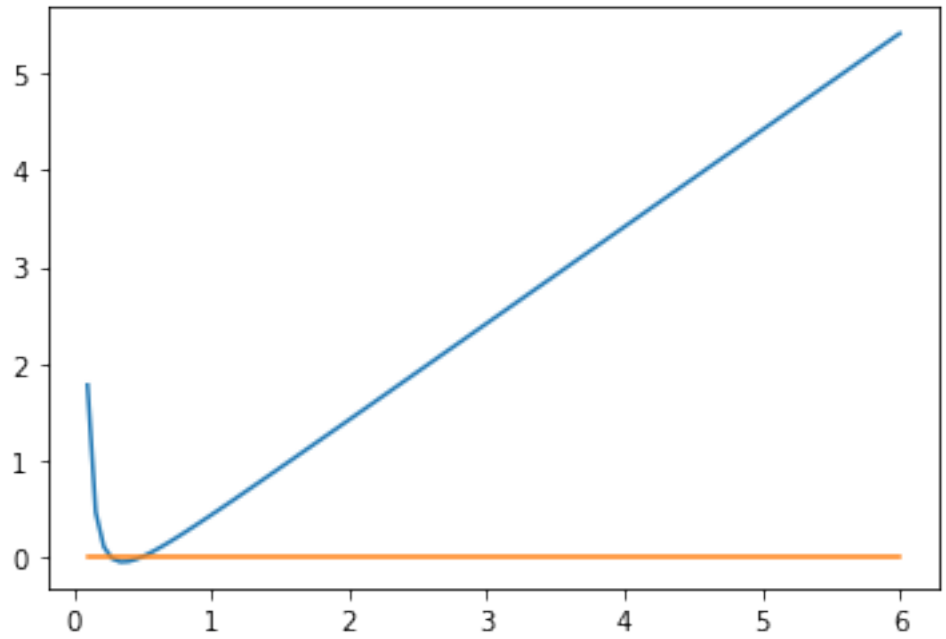
```
# função para cálculo de parâmetros
def a(Q,g,b,H,h0):
    return Q,g,b,H,h0

# função do nível de água
def f(h):
    frate,grav,width,bHeight,ups = a(Q,g,b,H,h0)
    c = lambda arg: frate**2/(2*grav*width**2*arg**2)
    return c(h) - c(h0) + h - h0 + H
```

Note que a função `a` é apenas uma conveniência para o cálculo do termo comum envolvendo a vazão e para construírmos uma generalização para os dados de entrada. Em seguida, definiremos os parâmetros de entrada do problema.

```
# parâmetros de entrada
Q = 1.2 # m3/s
g = 9.81 # m/s2
b = 1.8 # m
h0 = 0.6 # m
H = 0.075 # m
```

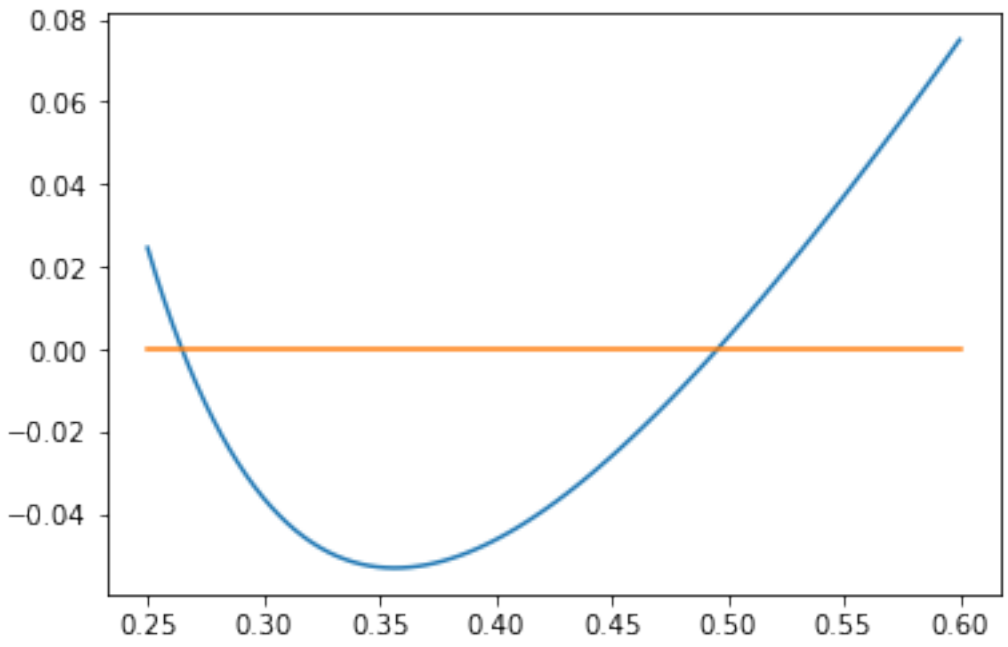
A partir daí, podemos realizar a análise gráfica para verificar o comportamento de `f(h)`.

```
# análise gráfica
h = np.linspace(0.1,6,num=100)
plt.plot(h,f(h),h,f(h)*0);
```



Ampliemos a localização.

```
# análise gráfica
h = np.linspace(0.25,0.6,num=100)
plt.plot(h,f(h),h,f(h)*0);
```



Verificamos que `f(h)` admite duas soluções. Vamos determinar cada uma delas.

```
# solução
h1 = bisect(f,0.25,0.32)
print('Raiz: h1 = %f' % h1)

h2 = bisect(f,0.4,0.55)
print('Raiz: h2 = %f' % h2)
```

```
Raiz: h1 = 0.264755
Raiz: h2 = 0.495755
```

Nota: as duas soluções viáveis dizem respeito ao regime de escoamento no canal aberto. Enquanto h_1 é um limite para escoamento supercrítico (rápido), h_2 é um limite para escoamento subcrítico (lento).

35.1.8 Problema 8

A velocidade v de um foguete Saturn V em voo vertical próximo à superfície da Terra pode ser aproximada por

$$v = u\ln\left(\frac{M_0}{M_0 - \dot{m}t}\right) - gt,$$

onde $u = 2510\,m/s$ é a velocidade de escape relativa ao foguete, $M_0 = 2.8 \times 10^6\,kg$ é a massa do foguete no momento do lançamento, $\dot{m} = 13.3 \times 10^3\,kg/s$ é a taxa de consumo de combustível, $g = 9.81\,m/s^2$ a aceleração gravitacional e t o tempo medido a partir do lançamento.

Determine o instante de tempo t^* quando o foguete atinge a velocidade do som ($335\,m/s$).

Resolução

Seguiremos a mesma ideia utilizada no Problema 7. Primeiramente, construímos uma função auxiliar para calcular parâmetros e, em seguida, definimos uma função `f(t)`.

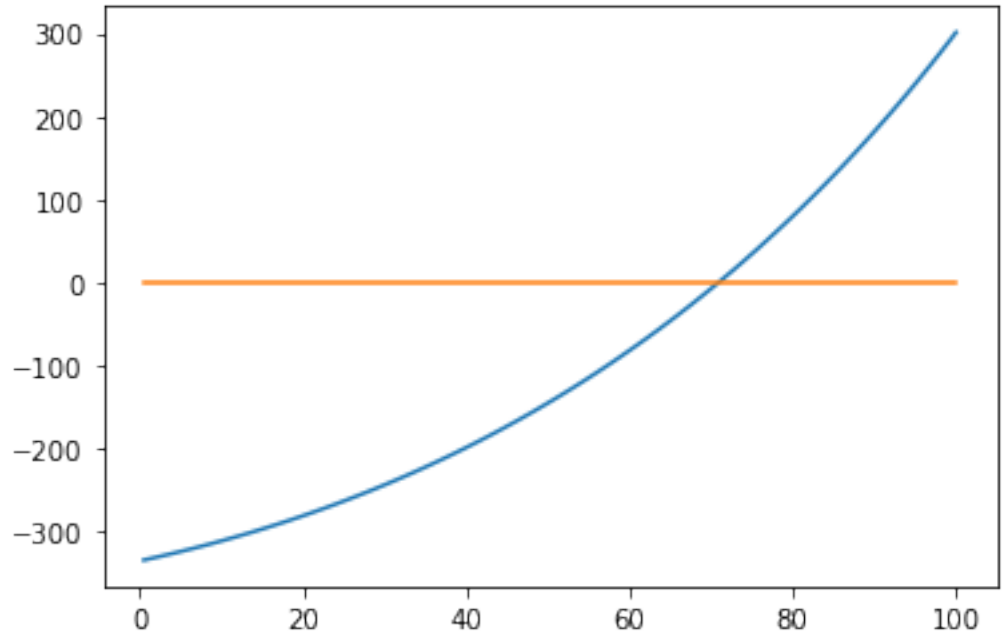
```
# função para cálculo de parâmetros
def a(u,M0,m,g,v):
    return u,M0,m,g,v

# função do tempo
def f(t):
    escape,mass,fuel,grav,vel = a(u,M0,m,g,v)
    return escape*np.log(mass/(mass - fuel*t)) - g*t - vel
```

Definimos os parâmetros do problema.

```
# parâmetros de entrada
u = 2510.0 # m/s
M0 = 2.8e6 # kg
m = 13.3e3 # kg/s
g = 9.81 # m/s2
v = 335.0 # m/s
```

Utilizaremos a análise gráfica para determinar o intervalo de refinamento da raiz.

```
# análise gráfica
t = np.linspace(0.5,100,num=100)
plt.plot(t,f(t),t,f(t)*0);
```



Podemos verificar que a raiz está entre 60 e 80 segundos. Utilizaremos estes limitantes.

```
# solução
tr = bisect(f,60,80)
print('Raiz: tr = %.2f s = %.2f min' % (tr,tr/60) )
```

```
Raiz: tr = 70.88 s = 1.18 min
```

O foguete rompe a barreira do som em 1 minuto e 18 segundos!

CHAPTER

THIRTYSIX

CODE SESSION 2

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import sympy as sy
```

36.1 Determinação de raízes

36.2 `newton`

A função `newton` localiza a raiz de uma função dentro de um intervalo dado usando o método de Newton. Os argumentos de entrada obrigatórios desta função são:

1. a função-alvo `f` (contínua)
2. a estimativa inicial `x0`

Parâmetros opcionais relevantes são:

- `fprime`: a derivada da função, quando disponível. Caso ela não seja especificada, o *método da secante* é usado.
- `fprime2`: a segunda derivada da função, quando disponível. Se ela for especificada, o *método de Halley* é usado.
- `tol`: tolerância (padrão: 1.48e-08)
- `maxiter`: número máximo de iterações (padrão: 50)
- `disp`: mostra erro se algoritmo não convergir (padrão: True)

O argumento de saída é:

- `x`: a estimativa para a raiz de `f`

Como importá-la?

```
from scipy.optimize import newton
```

```
from scipy.optimize import newton
```

36.2.1 Problema 1

Encontre a menor raiz positiva (real) de $x^3 - 3.23x^2 - 5.54x + 9.84 = 0$ pelo método de Newton.

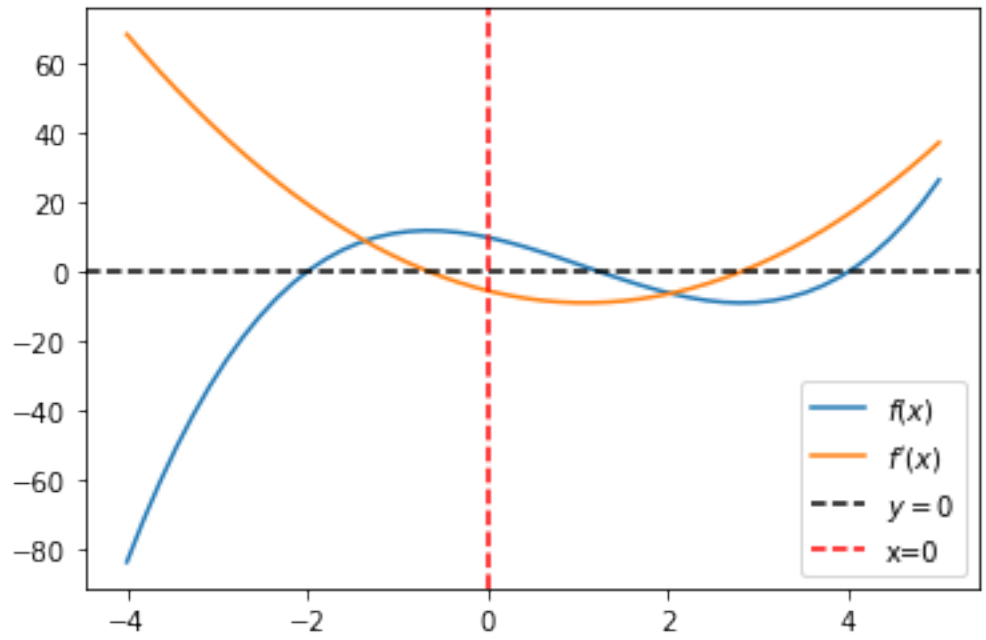
Resolução

Definimos a função e sua primeira derivada.

```
# função
def f(x):
    return x**3 - 3.23*x**2 - 5.54*x + 9.84

# 1a. derivada
def df(x):
    return 3*x**2 - 2*3.23*x - 5.54
```

Realizamos a análise gráfica.

```
# analise gráfica
x = np.linspace(-4,5)
plt.plot(x,f(x));
plt.plot(x,df(x));
plt.axhline(y=0,color='k',ls='--');
plt.axvline(x=0,color='r',ls='--');
plt.legend(['$f(x)$','$f\'(x)$','$y=0$','x=0']);
```



Vamos realizar um estudo de diferentes estimativas iniciais e ver o que acontece.

Estimativa inicial: $x_0 = -1$

```
# resolução com newton
x0 = -1.
x = newton(f,x0,df) # raiz
print('Raiz: x = %f' % x)
```

```
Raiz: x = -2.000000
```

Estimativa inicial: $x_0 = 0$

```
# resolução com newton
x0 = 0.
x = newton(f,x0,df) # raiz
print('Raiz: x = %f' % x)
```

```
Raiz: x = 1.230000
```

Estimativa inicial: $x_0 = 3$

```
# resolução com newton
x0 = 3.
x = newton(f,x0,df) # raiz
print('Raiz: x = %f' % x)
```

```
Raiz: x = 4.000000
```

36.2.2 Problema 2

Determine a menor raiz não nula positiva de $\cosh(x)\cos(x) - 1 = 0$ dentro do intervalo $(4, 5)$.

Resolução:

Primeiramente, vamos escrever a função `f(x)`.

```
# função
f = lambda x: np.cosh(x)*np.cos(x) - 1
```

Para computar a primeira derivada, vamos utilizar computação simbólica. Veja no início deste notebook que inserimos a instrução

```
import sympy as sy
```

a qual nos permitirá utilizar objetos do módulo `sympy`.

Em primeiro lugar, devemos estabelecer uma variável simbólica `xs`.

```
xs = sy.Symbol('x')
```

Em seguida, devemos utilizar as funções `cosh` e `cos` **simbólicas** para derivar `f`. Elas serão **chamadas de dentro do módulo sympy**.

Escrevemos a expressão simbólica para a derivada.

```
d = sy.diff(sy.cosh(xs)*sy.cos(xs) - 1)
d
```

$$-\sin(x)\cosh(x) + \cos(x)\sinh(x)$$

Note que `d` é um objeto do módulo `sympy`

```
type(d)
```

```
sympy.core.add.Add
```

Podemos agora aproveitar a expressão de `d` para criar nossa derivada. Se imprimirmos `d`, teremos:

```
print(d)
```

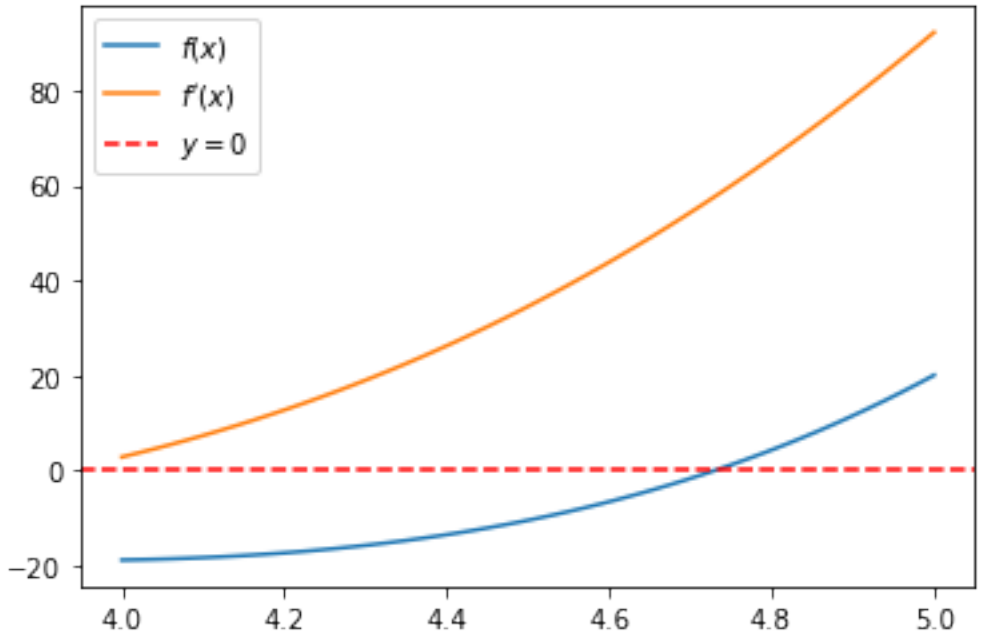
```
-sin(x)*cosh(x) + cos(x)*sinh(x)
```

Porém, precisamos indicar que as funções serão objetos numpy. Logo, adicionamos `np`, de modo que:

```
df = lambda x: - np.sin(x)*np.cosh(x) + np.cos(x)*np.sinh(x)
```

Agora, realizamos a análise gráfica.

```
# analise gráfica
x = np.linspace(4,5)
plt.plot(x,f(x));
plt.plot(x,df(x));
plt.axhline(y=0,color='r',ls='--');
plt.legend(['$f(x)$','$f\'(x)$','$y=0$']);
```



Agora, vamos resolver optando pela estimativa inicial $x_0 = 4.9$.

```
# resolução com newton
x0 = 4.9
x = newton(f,x0,df) # raiz
print('Raiz: x = %f' % x)
```

```
Raiz: x = 4.730041
```

36.3 *Homework*

1. Reproduza os Problemas de 3 a 8 da *Code Session 1* resolvendo com o método `newton`.
2. Para os casos possíveis, determine a derivada. Caso contrário, utilize como método da Secante.
3. Pesquise sobre o método de Halley e aplique-o aos problemas usando também a função `newton`, mas avaliando-a também com a segunda derivada.

CHAPTER

THIRTYSEVEN

CODE SESSION 3

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

37.1 Determinação de raízes de polinômios

37.1.1 `roots`

A função `roots` computa as raízes de uma função dentro de um intervalo dado usando o método de Hörner. O único argumento de entrada desta função é

1. o *array* `p` com os coeficientes dos termos do polinômio.

$$P(x) = p_n x^n + p_{n-1} x^{n-1} + ... + p_1 x + p_0$$

O argumento de saída é:

- `x`: *array* com as raízes de $P(x)$.

Como importá-la?

```
from numpy import roots
```

Porém, como já fizemos uma importação do `numpy` acima, basta utilizarmos

```
np.roots(p)
```

37.1.2 Problema 1

Determine as raízes de $P(x) = 3x^3 + 7x^2 - 36x + 20$.

Resolução

Para tornar claro, em primeiro lugar, vamos inserir os coeficientes de $P(x)$ em um *array* chamado `p`.

```
p = np.array([3,7,-36,20])
```

Em seguida, fazemos:

```
x = np.roots(p)
```

Podemos imprimir as raízes da seguinte forma:

```
for i, v in enumerate(x):
    print(f'Raiz {i}: {v}')
```

```
Raiz 0: -5.0
Raiz 1: 1.9999999999999987
Raiz 2: 0.6666666666666669
```

37.1.3 `polyval`

Podemos usar a função `polyval` do `numpy` para avaliar $P(x)$ em $x = x_0$. Verifiquemos, analiticamente, se as raízes anteriores satisfazem realmente o polinômio dado.

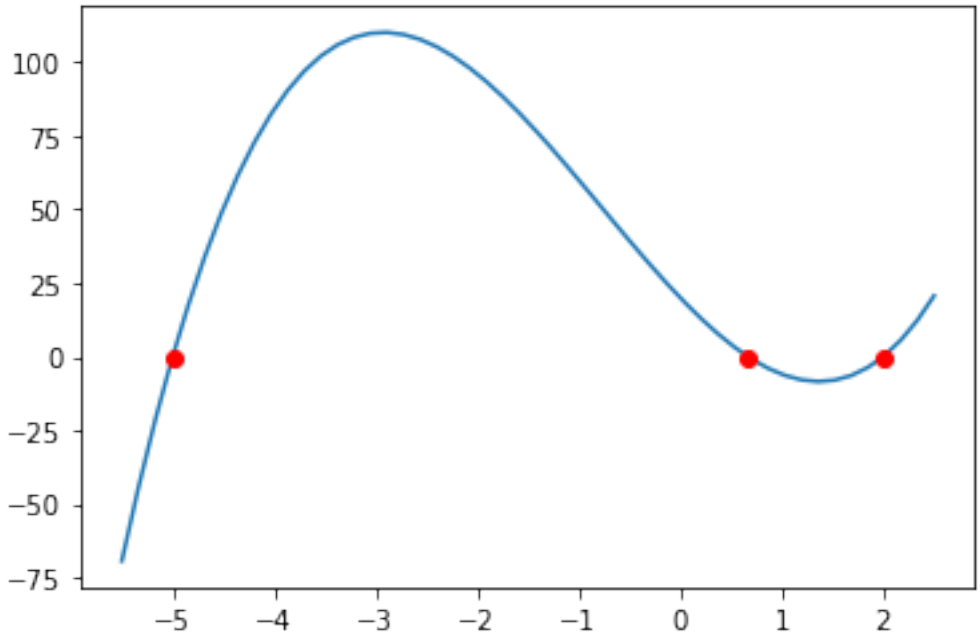
```
for i in x:
    v = np.polyval(p,i)
    print(f'P(x) = {v}')
```

```
P(x) = 0.0
P(x) = -3.552713678800501e-14
P(x) = -7.105427357601002e-15
```

Note que as duas últimas raízes são "muito próximas" de zero, mas não exatamente zero.

Podemos também fazer uma verificação geométrica plotando o polinômio e suas raízes.

```
xx = np.linspace(np.min(x)-0.5,np.max(x)+0.5)
plt.plot(xx,np.polyval(p,xx));
for i in x:
    plt.plot(i,np.polyval(p,i),'or')
```



37.1.4 Problema 2

Determine as raízes de $P(x) = x^4 - 3x^2 + 3x$.

Resolução

Resolvendo diretamente com `roots` e usando `polyval` para verificação, temos:

```
# coeficientes e raízes
p = np.array([1,0,-3,3,0])
x = np.roots(p)
```

```
# imprimindo as raizes
for i, v in enumerate(x):
    print(f'Raiz {i}: {v}')
```

```
Raiz 0: (-2.1038034027355357+0j)
Raiz 1: (1.051901701367768+0.5652358516771712j)
Raiz 2: (1.051901701367768-0.5652358516771712j)
Raiz 3: 0j
```

Note que, neste caso, as raízes são complexas.

37.1.5 Problema 3

Determine as raízes de $P(x) = x^5 - 30x^4 + 361x^3 - 2178x^2 + 6588x - 7992$.

Resolução

```
# coeficientes e raízes
p = np.array([1,-30,361,-2178,6588,-7992])
x = np.roots(p)
```

```
# imprimindo as raizes
for i, v in enumerate(x):
    print(f'Raiz {i}: {v}')
```

```
Raiz 0: (6.000000000009944+0.9999999999996999j)
Raiz 1: (6.000000000009944-0.9999999999996999j)
Raiz 2: (6.00026575921113+0j)
Raiz 3: (5.999867120384507+0.00023015565268626 68j)
Raiz 4: (5.999867120384507-0.00023015565268626 68j)
```

CHAPTER

THIRTYEIGHT

CODE SESSION 4

38.1 `fsolve`

A função `fsolve` do submódulo `scipy.optimize` pode ser usada como método geral para busca de raízes de equações não-lineares escalares ou vetoriais.

Para usar `fsolve` em uma equação escalar, precisamos de, no mínimo:

- uma função que possui pelo menos um argumento
- estimativa inicial para a raiz

Para equações vetoriais (sistemas), precisamos de mais argumentos. Vejamos o exemplo do paraquedista:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve

t = 12.0
v = 42.0
m = 70.0
g = 9.81

def param(t,v,m,g):
    return [t,v,m,g]

def fun(c):
    p = param(t,v,m,g)
    return p[3]*p[2]/c*(1 - np.exp(-c/p[2]*p[0])) - p[1]

# estimativa inicial
c0 = -1000.0

# raiz
c_raiz = fsolve(fun,c0)

# impressao (estilo Python 2)
print('Minha raiz é %.6f' % c_raiz)

# impressao (estilo Python 3)
print("Minha raiz é {0:.6f}".format(c_raiz[0]))
```

```
Minha raiz é 15.127432
Minha raiz é 15.127432
```

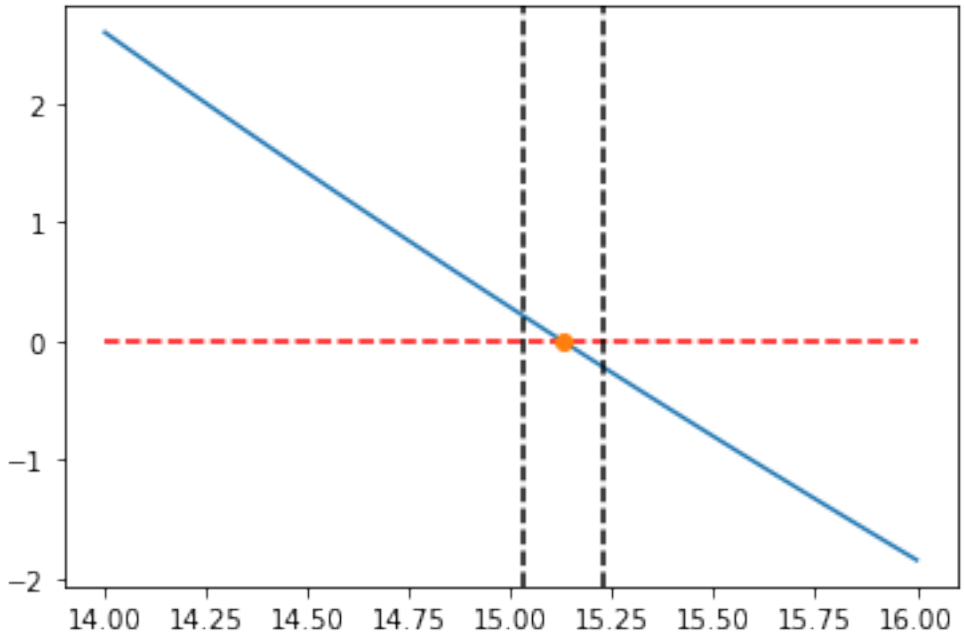
38.1.1 Como incorporar tudo em uma só função

```
def minha_fun(t,v,m,g,c0):

    p = [t,v,m,g]
    f = lambda c: p[3]*p[2]/c*(1 - np.exp(-c/p[2]*p[0])) - p[1]
    c_raiz = fsolve(f,c0)
    print("---> Minha raiz é {0:.6f}".format(c_raiz[0]))
    return f,c_raiz
```

```
fc,c_raiz = minha_fun(t,v,m,g,c0)
```

```
---> Minha raiz é 15.127432
```

```
a,b = 14,16
c = np.linspace(a,b,100)

delta = 0.1
plt.plot(c,fc(c),c,0*c,'r--',c_raiz,fc(c_raiz),'o');
plt.axvline(c_raiz - delta,c='k',ls='--');
plt.axvline(c_raiz + delta,c='k',ls='--');
```



38.1.2 Problema 1

Resolva o sistema não-linear abaixo:

$$\begin{cases} x^2 + y^2 = 2 \\ x^2 - y^2 = 1 \end{cases}$$

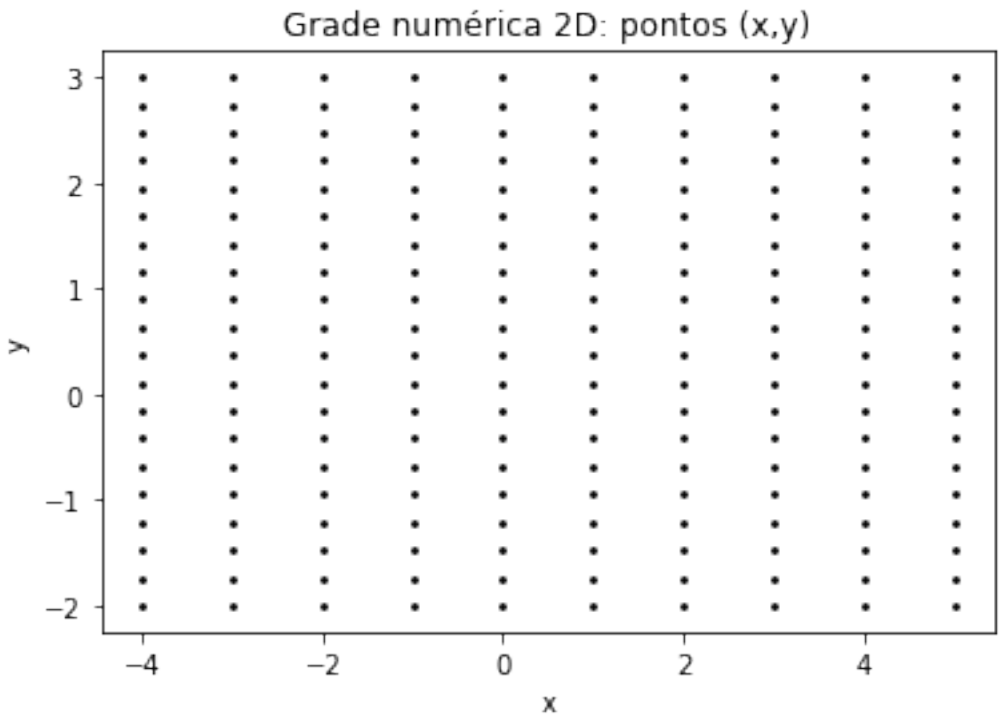
Resolução

Primeiramente, vamos plotar as curvas de nível 0 das funções que compõem o sistema. Faremos isto usando uma *grade numérica* e a função `contour`.

Criando uma grade numérica bidimensional uniforme

Uma *grade numérica* é um conjunto de pontos separados por uma distância uniforme (ou variável). Neste caso em particular, criaremos uma *grade numérica* bidimensional. Podemos fazer isto da seguinte forma:

```
import numpy as np
import matplotlib.pyplot as plt

# limites do domínio:
# região do plano [a,b] x [c,d]
a, b = -4.0, 5.0
c, d = -2.0, 3.0

# no. de pontos em cada direção
nx, ny = 10, 20

# distribuição dos pontos
x = np.linspace(a,b,nx)
y = np.linspace(c,d,ny)

# grade numérica 2D
[X,Y] = np.meshgrid(x,y)

# plotando pontos da grade numérica
plt.scatter(X,Y,s=3,c='k');
plt.title('Grade numérica 2D: pontos (x,y)')
plt.xlabel('x'); plt.ylabel('y');
```

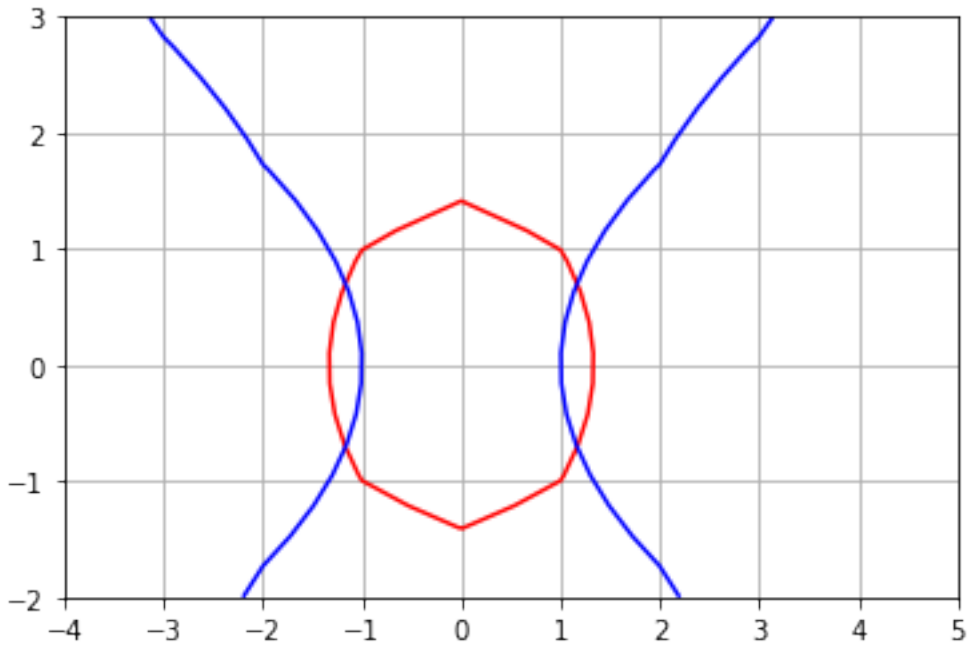


Plotando curvas de nível

Plotaremos as curvas de nível 0 das funções não-lineares para realizar a análise gráfica e localizar as raízes para então escolhermos um vetor de estimativa inicial.

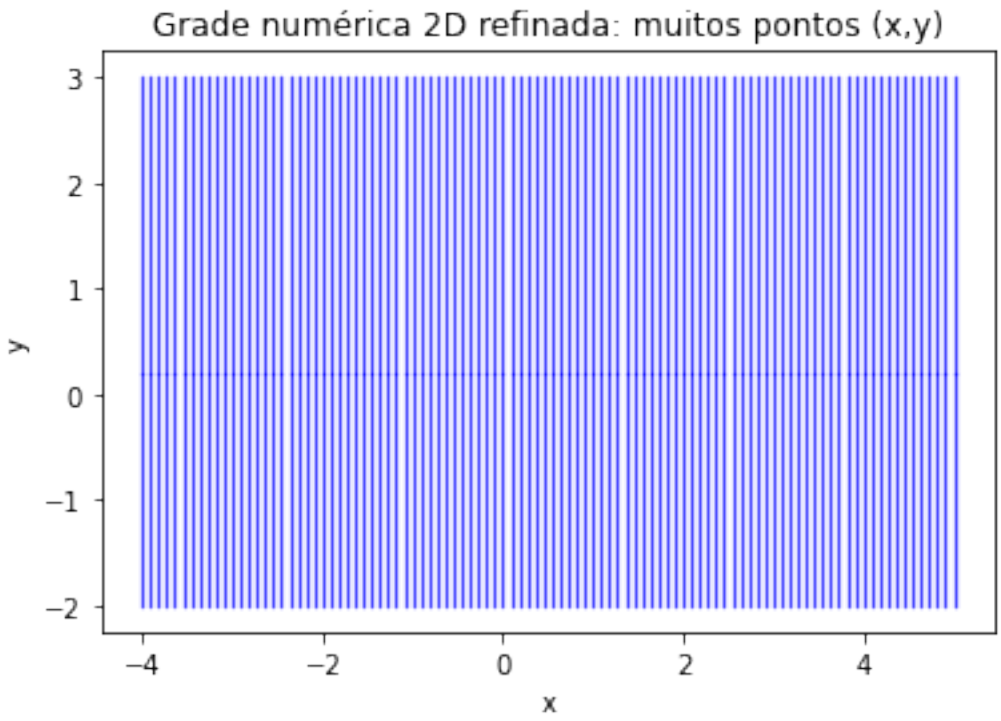
Para plotar curvas de nível das funções sobre a grade numérica anterior, fazemos o seguinte:

```
# funções definidas sobre a grade 2D
F = X**2 + Y**2 - 2
G = X**2 - Y**2 - 1

# contorno de nível 0
plt.contour(X,Y,F,colors='red',levels=0);
plt.contour(X,Y,G,colors='blue',levels=0);
plt.grid()
```



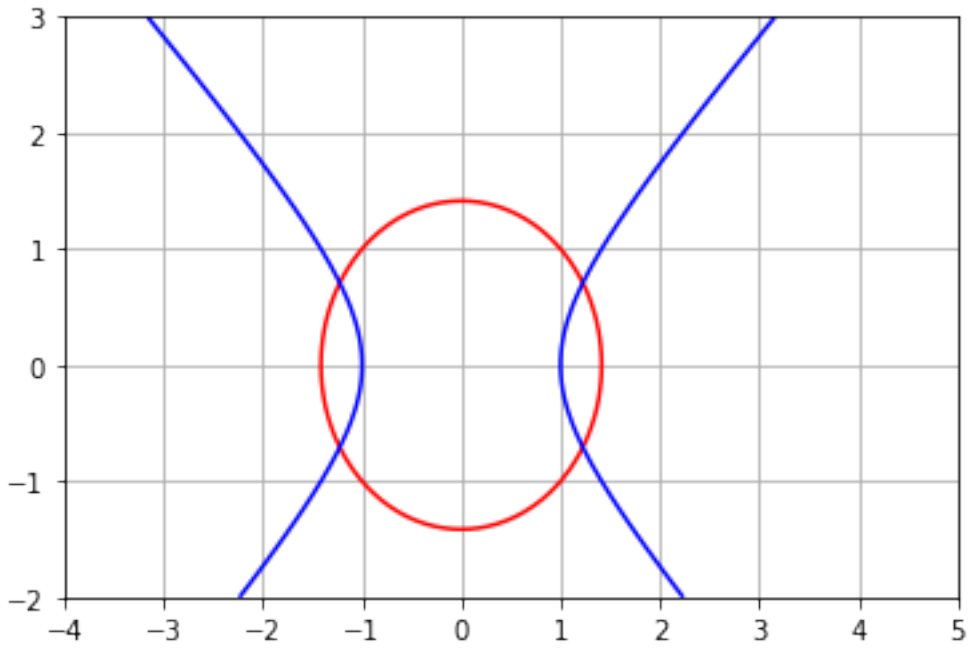
Por que a figura está meio "tosca"? Porque temos poucos pontos na grade. Vamos aumentar o número de pontos. Este processo é conhecido como *refinamento de malha*.

```
# refinando a malha numérica
nx2, ny2 = 100, 200

# redistribuição dos pontos
x2 = np.linspace(a,b,nx2)
y2 = np.linspace(c,d,ny2)

# grade numérica 2D refinada
[X2,Y2] = np.meshgrid(x2,y2)

# plotando pontos da grade numérica
plt.scatter(X2,Y2,s=0.1,c='b');
plt.title('Grade numérica 2D refinada: muitos pontos (x,y)')
plt.xlabel('x'); plt.ylabel('y');
```



Vamos plotar novamente as curvas de nível das funções sobre a grade numérica refinada.

```
# funções definidas sobre a grade 2D refinada
F2 = X2**2 + Y2**2 - 2
G2 = X2**2 - Y2**2 - 1

# contorno de nível 0 na malha refinada
plt.contour(X2,Y2,F2,colors='red',levels=0);
plt.contour(X2,Y2,G2,colors='blue',levels=0);
plt.grid()
```

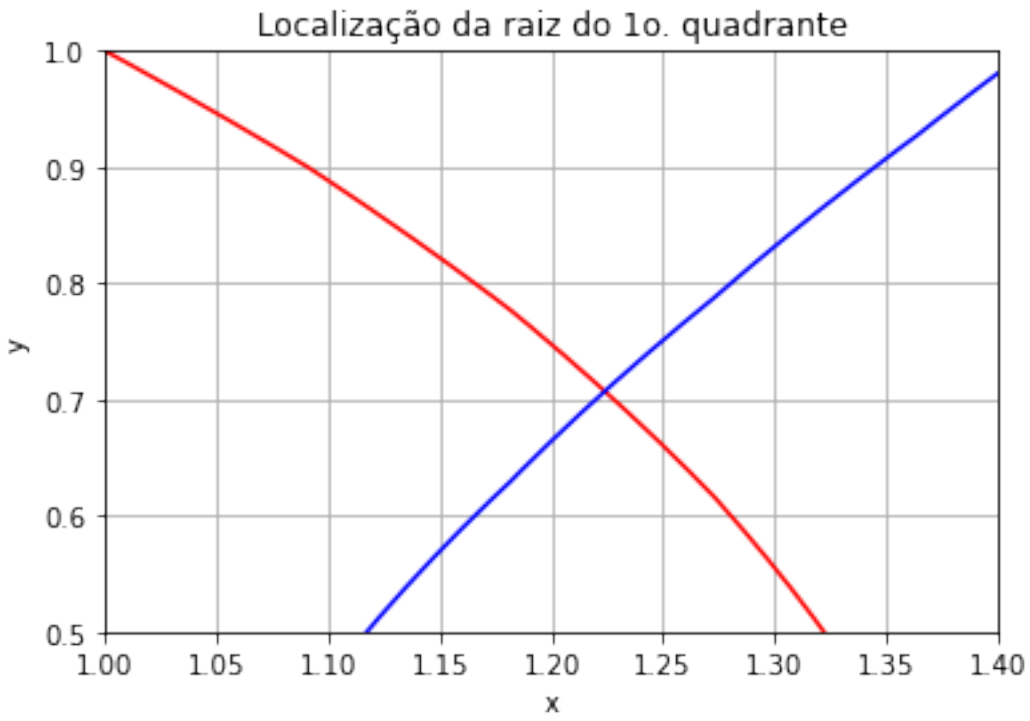


Estimativa inicial

A partir do gráfico anterior, vemos que há 4 raízes possíveis para o sistema não-linear. Vamos escolher uma delas para aproximar. Por conveniência, escolhamos a que se encontra no primeiro quadrante.

Vamos fazer uma plotagem localizada no primeiro quadrante:

```
# contorno de nível 0 na malha refinada
plt.contour(X2,Y2,F2,colors='red',levels=0);
plt.contour(X2,Y2,G2,colors='blue',levels=0);
plt.xlim([1.0,1.4])
plt.ylim([0.5,1.0])
plt.xlabel('x'); plt.ylabel('y'); plt.grid()
plt.title('Localização da raiz do 1o. quadrante');
```



Observando o gráfico, faremos a escolha do ponto $(x_0, y_0) = (1.2, 0.7)$ como estimativa inicial.

Resolução do sistema não-linear

Para resolver o sistema não-linear, primeiro definimos uma função que retornará uma tupla contendo cada função do sistema em cada uma de suas coordenadas.

```
# função que returna uma lista com as funções do sistema
def F(vars):
    x,y = vars # cria x,y como variáveis locais
    f = x**2 + y**2 - 2 # f(x,y) = 0
    g = x**2 - y**2 - 1 # g(x,y) = 0
    return [f,g]
```

Em seguida, usamos a função `fsolve` passando o vetor inicial escolhido, isto é, $(x_0, y_0) = (1.2, 0.7)$, para determinar a solução aproximada (x_1, y_1).

```
from scipy.optimize import fsolve

xr, yr = fsolve(F,(1.2,0.7))
print(f'A solução aproximada é o vetor (xr,yr) = ({xr:.3f},{yr:.3f})')
```

```
A solução aproximada é o vetor (xr,yr) = (1.225,0.707)
```

Verificação e plotagem

Podemos verificar que x_r e y_r satisfazem às equações dentro de uma certa precisão:

```
xr**2 + yr**2 - 2
```

```
3.956968086527013e-11
```

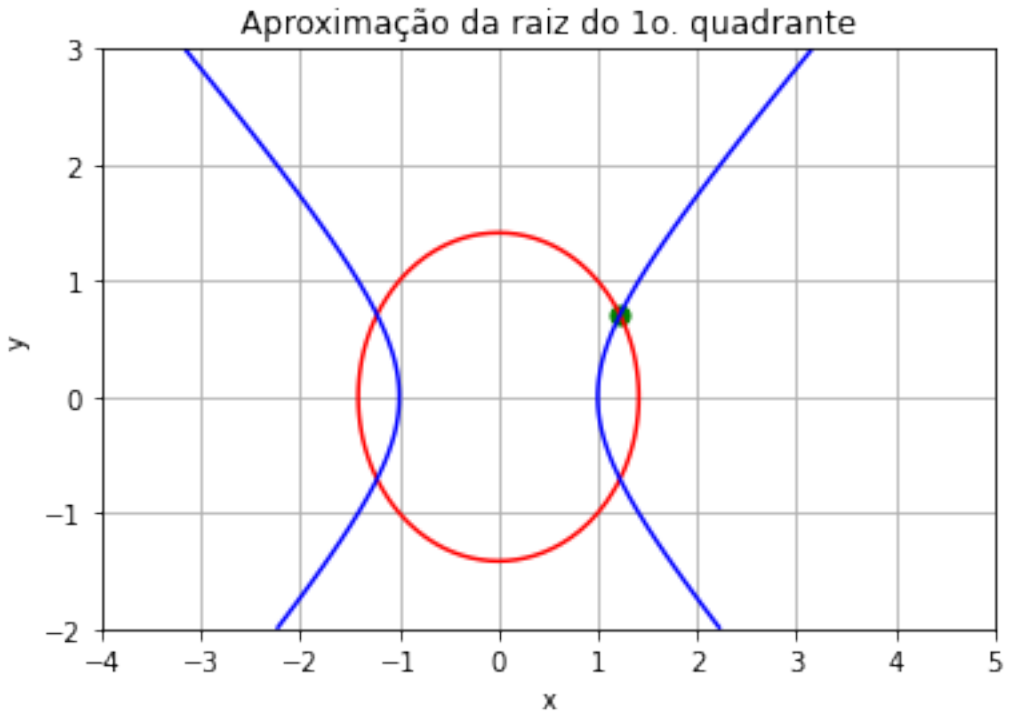
```
xr**2 - yr**2 - 1
```

```
-6.642031369352708e-11
```

Isto mostra que os valores estão muito próximos de zero.

Finalmente, podemos plotar as curvas de nível destacando a solução obtida.

```
# contorno de nível 0 na malha refinada
plt.contour(X2,Y2,F2,colors='red',levels=0);
plt.contour(X2,Y2,G2,colors='blue',levels=0);
plt.xlabel('x'); plt.ylabel('y'); plt.grid()
plt.title('Aproximação da raiz do 1o. quadrante');
plt.scatter(xr,yr,c='green',s=50);
```



CHAPTER

THIRTYNINE

CODE SESSION 5

```
import numpy as np
```

39.1 linalg.solve

A função `solve` é o método mais simples disponibilizado pelos módulos `numpy` e `scipy` para resolver sistemas matriciais lineares. Como a função pertence ao escopo da Álgebra Linear, ela está localizada em submódulo chamado `linalg`. `solve` calculará a solução exata do sistema como um método direto se a matriz do sistema for determinada (quadrada e sem colunas linearmente dependentes). Se a matriz for singular, o método retorna um erro. Se for de posto deficiente, o método resolve o sistema linear usando um algoritmo de mínimos quadrados.

Os argumentos de entrada obrigatórios desta função são:

1. a matriz `A` dos coeficientes
2. o vetor independente `b`

O argumento de saída é:

- `x`: o vetor-solução do sistema.

Como importá-la?

```
from numpy.linalg import solve
```


```
from numpy.linalg import solve
```

39.1.1 Problema 1

Uma rede elétrica contém 3 *loops*. Ao aplicar a lei de Kirchoff a cada *loop*, o cientista Hely Johnson obteve as seguintes equações para as correntes i_1, i_2 e i_3 em cada *loop:*

$$
\begin{aligned}
& & to \\
(50+R)i_1 - Ri_2 - 30i_3 &= & \\
& & 0 \\
-Ri_1 + (65+R)i_2 - 15i_3 &= & \\
& & 0 \\
-30i_1 - 15i_2 + 45i_3 &= & \\
& & 120 (1)
\end{aligned}
$$

$$
\begin{aligned}
& = \\
0 - Ri_1 + (65+R)i_2 - 15i_3 \\
0 - 30i_1 - 15i_2 + 45i_3 \\
120
\end{aligned}
$$

Ajude Hely Johnson em seus experimentos e calcule as correntes para os valores de resistência $R = \{5\Omega, 10\Omega, 20\Omega\}$.

39.1.2 Resolução

Podemos começar definindo uma lista para armazenar os valores das resistências de teste:

```
R = [5.,10.,15.]
```

Em seguida, escreveremos a matriz dos coeficientes e o vetor independente (lado direito). Note que precisamos de uma "lista de listas", ou melhor, um *"array de arrays"*, onde cada *array* está associado à uma linha da matriz.

Todavia, vamos definir uma função para montar o sistema em função do valor de R e resolvê-lo.

```
# montagem do sistema
def resolve_sistema(R):
    A = np.array([ [50+R,-R,-30],[-R,65+R,-15],[-30,-15,45] ])
    b = np.array([0,0,120])
    x = solve(A,b)
    return x
```

Além disso, usaremos um laço `for` para calcularmos todas as respostas necessárias de uma só vez e organizaremos os resultados em dicionário (objeto `dict`):

```
# salva soluções agrupadas em um dicionário
sols = {}
for r in R:
    sols[r] = resolve_sistema(r)
    print('Para o valor de resistência R = {0:g} ohms: i1 = {1:g} A, i2 = {2:g} A, i3
 ↪= {3:g} A'.format(r, sols[r][0], sols[r][1], sols[r][2]))
```

```
Para o valor de resistência R = 5 ohms: i1 = 2.82927 A, i2 = 1.26829 A, i3 = 4.97561 A
Para o valor de resistência R = 10 ohms: i1 = 2.66667 A, i2 = 1.33333 A, i3 = 4.88889
 ↪A
Para o valor de resistência R = 15 ohms: i1 = 2.54545 A, i2 = 1.38182 A, i3 = 4.82424
 ↪A
```

39.1.3 Tarefa

Retorne ao notebook da *Aula 09* e use o conteúdo da aula para fazer uma verificação das soluções encontradas para este problema em cada caso. Use a função `linalg.cond` para calcular o *número de condicionamento* da matriz do sistema em cada caso. (**Sugestão:** vide `numpy.allclose`)

39.2 `linalg.cholesky`

Assim como `solve`, a função `cholesky` está disponível tanto via `numpy` como `scipy` para determinar a decomposição de Cholesky de uma matriz simétrica e positiva definida.

Na prática, não é recomendável verificar se uma matriz é positiva definida através dos critérios teóricos. A função `cholesky` não realiza essa checagem. Portanto, é importante que, pelo menos, se saiba que a matriz é simétrica. Para testar se ela atende a propriedade de definição positiva, a abordagem mais direta é usar `cholesky` e verificar se ela retorna uma decomposição de Cholesky. Se não for o caso, um erro será lançado.

O argumento de entrada desta função é:

- `A`: matriz dos coeficientes

O argumento de saída é:

- `L`: o fator de Cholesky

Como importá-la?

```
from numpy.linalg import cholesky
```

```
from numpy.linalg import cholesky
```

39.2.1 Problema 2

Calcule o fator de Cholesky para a matriz A do Problema 1 para $R = 5$.

```
R = 5.
B = np.array([ [50+R,-R,-30],[-R,65+R,-15],[-30,-15,45] ])

L = cholesky(B)
L
```

```
array([[ 7.41619849,  0.        ,  0.        ],
       [-0.67419986,  8.33939174,  0.        ],
       [-4.04519917, -2.12572731,  4.91097211]])
```

Podemos verificar a decomposição multiplicando a matriz triangular (fator de Cholesky) pela sua transposta.

```
np.matmul(L,L.T)
```

```
array([[ 55.,  -5., -30.],
       [ -5.,  70., -15.],
       [-30., -15.,  45.]])
```

```
B
```

```
array([[ 55.,  -5., -30.],
       [ -5.,  70., -15.],
       [-30., -15.,  45.]])
```

Entretanto, não é verdade que

```
# por que a igualdade falha
# para algumas entradas?
B == np.matmul(L,L.T)
```

```
array([[ True, False,  True],
       [False,  True, False],
       [ True, False,  True]])
```

39.3 Problema 3

Verificar se uma matriz simétrica é positiva definida.

```
# cria matriz simétrica
C = np.tril(B) - 60
C = np.tril(C) + np.tril(C,-1).T
C
```

```
array([[ -5., -65., -90.],
       [-65.,  10., -75.],
       [-90., -75., -15.]])
```

```
# erro!
# matriz não é PD
#cholesky(C)
```

```
# cria outra matriz simétrica
D = np.tril(B) + 1
D = np.tril(D) + np.tril(D,-1).T
D
```

```
array([[ 56.,  -4., -29.],
       [ -4.,  71., -14.],
       [-29., -14.,  46.]])
```

```
# matriz é PD
cholesky(D)
```

```
array([[ 7.48331477,  0.        ,  0.        ],
       [-0.53452248,  8.40917866,  0.        ],
       [-3.87528801, -1.91117697,  5.22776678]])
```

39.4 Problema 4

Resolva o Problema 1 para $R = 10$ usando a fatoração de Cholesky.

```
R = 10.
D = np.array([ [50+R,-R,-30],[-R,65+R,-15],[-30,-15,45] ])
b = np.array([0,0,120])

# fator
L = cholesky(D)

# passo 1
# Ly = b
y = solve(L,b)

# passo 2
# L^T x = y
x = solve(L.T,y)

# solução
x
```

```
array([2.66666667, 1.33333333, 4.88888889])
```

Compare a solução via Cholesky com a do Problema 1:

```
x, sols[10]
```

```
(array([2.66666667, 1.33333333, 4.88888889]),
 array([2.66666667, 1.33333333, 4.88888889]))
```

Mais uma vez, note que:

```
# A comparação falha.
# Por quê?
x == sols[10]
```

```
array([False,  True,  True])
```

CHAPTER

FORTY

CODE SESSION 6

```
%matplotlib inline
```

40.1 interp1d

A classe `interp1d` do submódulo `scipy.interpolate` pode ser usada como uma estrutura genérica para o cálculo de interpolação unidimensional do tipo $y_k = f(x_k)$.

Para usar `interp1d`, precisamos de, no mínimo, uma tabela de dados fornecida por dois parâmetros:

1. `x`: array de valores independentes
2. `y`: array de valores dependentes

Um dos argumentos opcionais relevantes de `interp1d` é:

- `kind`: tipo de dado `str` ou `int` que especifica o tipo de interpolação.

O valor padrão de `kind` é `'linear'`, o qual equivale à configuração de uma interpolação linear. Outras opções relevantes, bem como o que elas realizam estão dispostas na tabela a seguir:

opção	interpolação
`'nearest'`	vizinho mais próximo
`'zero'`	interpolação por spline de ordem 0
`'slinear'`	interpolação por spline de ordem 1
`'quadratic'`	interpolação por spline de ordem 2
`'cubic'`	interpolação por spline de ordem 3

Se um valor inteiro for passado para `'kind'`, ele indicará a ordem da spline interpolatória. Por exemplo, `'kind' = 4` indica uma interpolação por spline de ordem 4.

Em Python, a classe `interp1d` é chamada da seguinte forma:

```
from scipy.interpolate import interp1d
```

Podemos, agora, resolver alguns problemas de interpolação unidimensional por meio desta classe.

Em primeiro lugar, vamos importar alguns módulos necessários para nossos cálculos.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
```

40.1.1 Problema 1

Valores de entalpia por unidade de massa, h, de um plasma de Argônio em equilíbrio *versus* temperatura estão tabelados no arquivo `file-cs6-entalpia.csv`. Usando esses dados:

- Escreva um programa para interpolar valores de h para temperaturas no intervalo $5000 - 30000\,^{\circ}K$, com incrementos de $500\,^{\circ}K$.
- Plote o gráfico de dispersão marcando com asteriscos os valores de entalpia tabelados. Em seguida, plote gráficos de linha para as seguintes interpolações: `'nearest'`, `'zero'`, `'slinear'` e `'quadratic'`.
- Compare os valores interpolados de h para cada um dos métodos de interpolação `'zero'`, `'slinear'` e `'quadratic'` do item anterior para $T = 15150\,^{\circ}K$.

Observação: note que a temperatura da tabela deve ser multiplicada por 1000.

Resolução

Em primeiro lugar, vamos ler a tabela de dados, atribuir os valores tabelados em arrays e corrigir os valores de temperatura pelo fator 1000.

```
# atribuindo colunas da matriz de dados em h e T
h, T = np.loadtxt('file-cs6-entalpia.csv', delimiter=',', skiprows=1, unpack=True)

# temperatura em milhares de Kelvin
T = 1e3*T
```

Criamos um array para o intervalo de temperaturas desejado para interpolação usando `arange`. Notemos que esta função exige um deslocamento do valor do incremento no último elemento do array, isto é, 30000 + 500 = 30500.

```
# array de temperaturas com incremento de 500 K
t = np.arange(5000.0,30500.0,500)
```

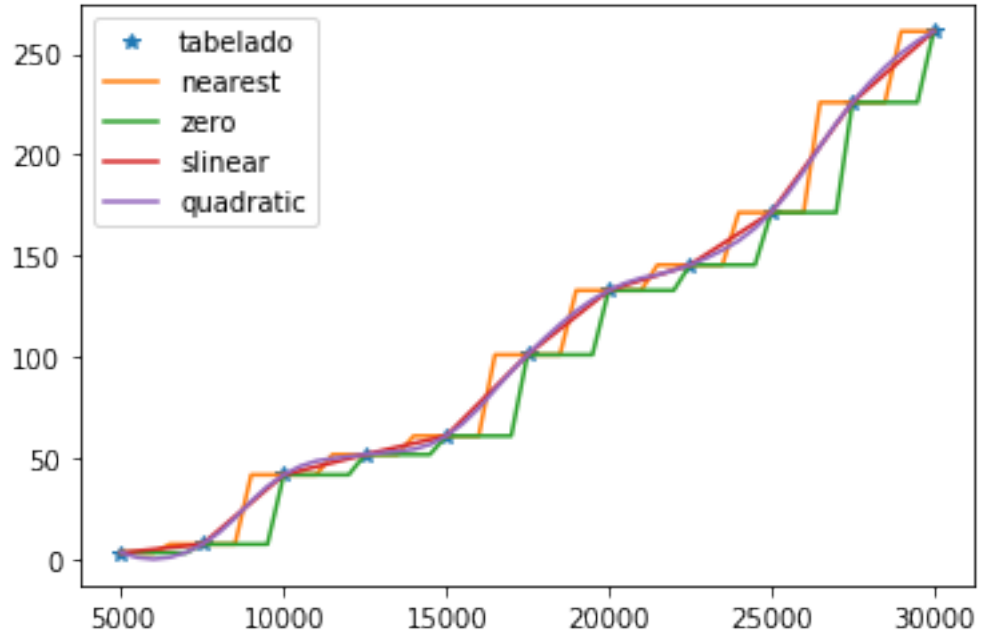
Em seguida, usamos os valores tabelados para posterior aplicação de `interp1d` sobre `t` como uma função e imprimimos os valores interpolados de entalpia:

```
# montagem da interpolação
f = interp1d(T,h)

# valores interpolados
hi = f(t)

hi
```

```
array([  3.3 ,   4.14,   4.98,   5.82,   6.66,   7.5 ,  14.36,  21.22,
        28.08,  34.94,  41.8 ,  43.8 ,  45.8 ,  47.8 ,  49.8 ,  51.8 ,
        53.64,  55.48,  57.32,  59.16,  61.  ,  69.02,  77.04,  85.06,
        93.08, 101.1 , 107.46, 113.82, 120.18, 126.54, 132.9 , 135.42,
       137.94, 140.46, 142.98, 145.5 , 150.68, 155.86, 161.04, 166.22,
       171.4 , 182.28, 193.16, 204.04, 214.92, 225.8 , 232.82, 239.84,
       246.86, 253.88, 260.9 ])
```

Vamos determinar os valores interpolados para cada método de interpolação e plotá-los juntamente com o gráfico de dispersão dos valores tabelados.

```
# métodos de interpolação
m = ['nearest', 'zero', 'slinear', 'quadratic']

# objetos de interpolação para cada método
F = [interp1d(T,h,kind=k) for k in m]

# valores interpolados
him = [f(t) for f in F]

# plotagem dos valores tabelados
plt.plot(T,h,'*',label='tabelado');

# plotagem dos métodos
for i in range(4):
    plt.plot(t,him[i],label=m[i])

# legenda
plt.legend();
```



Até aqui, já cumprimos os dois primeiros requisitos do problema. Para o terceiro, usaremos as informações pré-computadas na lista `F` para estimar os valores de entalpia quando $T = 15150\,^{\circ}K$. Teremos os seguintes três valores:

```
# calcula h(15150) para os métodos 'zero', 'slinear' e 'quadratic'
h_15150 = [f(15150) for f in F[1:]]

h_15150
```

```
[array(61.), array(63.406), array(62.60353522)]
```

Isto é, os valores de entalpia em $T = 15150\,^{\circ}K$ podem ser organizdos na tabela a seguir:

método	valor
`'zero'`	61.0 MJ/kg
`'slinear'`	63.406 MJ/kg
`'quadratic'`	62.604 MJ/kg

Levando em conta que quanto mais alta é a ordem de interpolação, melhor é a interpolação, podemos inferir que desses três valores, $62.604\,MJ/kg$ é o mais confiável para usar.

40.1.2 Problema 2

O arquivo `file-cs6-salinidade.csv` tabela valores de salinidade da água (em ppt) em função da profundidade oceânica (em metros). Use interpolação por spline cúbica para gerar uma tabela de salinidades para profundidades de 0 a 3000 m com espaçamento de 10 m e estime os valores nas profundidades de 250 m, 750 m e 1800 m.

40.1.3 Problema 3

A tabela a seguir apresenta a potência de um motor a Diesel (em hp) em diferentes rotações (em rpm). Gere uma tabela de valores interpolados com espaçamento de 10 rpm e destaque as potências em 2300 rpm e 3650 rpm.

velocidade (rpm)	potência (hp)
1200	65
1500	130
2000	185
2500	225
3000	255
3250	266
3500	275
3750	272
4000	260
4400	230

CHAPTER

FORTYONE

CODE SESSION 7

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

41.1 Regressão Linear

41.1.1 `linregress`

A regressão linear é o modelo mais básico para realizar ajuste de dados e frequentemente aplicado em estudos estatísticos. Em Python, a regressão linear pode ser realizada com a função `linregress`. Esta função calcula a regressão linear por mínimos quadrados (a rigor, o termo deveria ser traduzido como *quadrados mínimos*) para dois conjuntos de medição.

Os argumentos de entrada obrigatórios desta função são:

1. o primeiro conjunto de dados `x` (lista ou objeto tipo *array*)
2. o segundo conjunto de dados `y` (lista ou objeto tipo *array*)

Os argumentos de saída são:

- `slope`: coeficiente angular da reta obtida pela regressão linear
- `intercept`: coeficiente linear da reta obtida pela regressão linear
- `rvalue`: valor do coeficiente de correlação
- `pvalue`: valor- p de teste de hipótese
- `stderror`: medida de erro

Não utilizaremos os 2 últimos aqui.

Nota: para obter o valor do *coeficiente de determinação* R^2, o valor de `rvalue` deve ser elevado ao quadrado, i.e. `R2 = rvalue**2`.

Como importá-la?

```
from scipy.stats import linregress
```


```
from scipy.stats import linregress
```

41.1.2 Problema 1

A tabela a seguir lista a massa M e o consumo médio C de automóveis fabricados pela Ford e Honda em 2008. Faça um ajuste linear $C = b + aM$ aos dados e calcule o desvio padrão.

modelo	massa (kg)	C (km/litro)
Focus	1198	11.90
Crown Victoria	1715	6.80
Expedition	2530	5.53
Explorer	2014	6.38
F-150	2136	5.53
Fusion	1492	8.50
Taurus	1652	7.65
Fit	1168	13.60
Accord	1492	9.78
CR-V	1602	8.93
Civic	1192	11.90
Ridgeline	2045	6.38

Nota: esta tabela está disponível em formato .csv no arquivo `file-cs7-autos.csv`.

41.1.3 Resolução

Vamos ler o arquivo de dados e convertê-lo em uma matriz.

```
# fname: nome do arquivo
# delimiter: separador dos dados
# skiprows: ignora linhas do arquivo (aqui, estamos removendo a primeira)
# usecols: colunas a serem lidas (aqui, estamos lendo a 2a. e 3a. colunas)
dados = np.loadtxt(fname='file-cs7-autos.csv',delimiter=',',skiprows=1,usecols=(1,2))
print(dados)
```

```
[[1198.     11.9 ]
 [1715.      6.8 ]
 [2530.      5.53]
 [2014.      6.38]
 [2136.      5.53]
 [1492.      8.5 ]
 [1652.      7.65]
 [1168.     13.6 ]
 [1492.      9.78]
 [1602.      8.93]
 [1192.     11.9 ]
 [2045.      6.38]]
```

Armazenamos os *arrays* devidamente:

```
M = dados[:,0] # massa
C = dados[:,1] # consumo
```

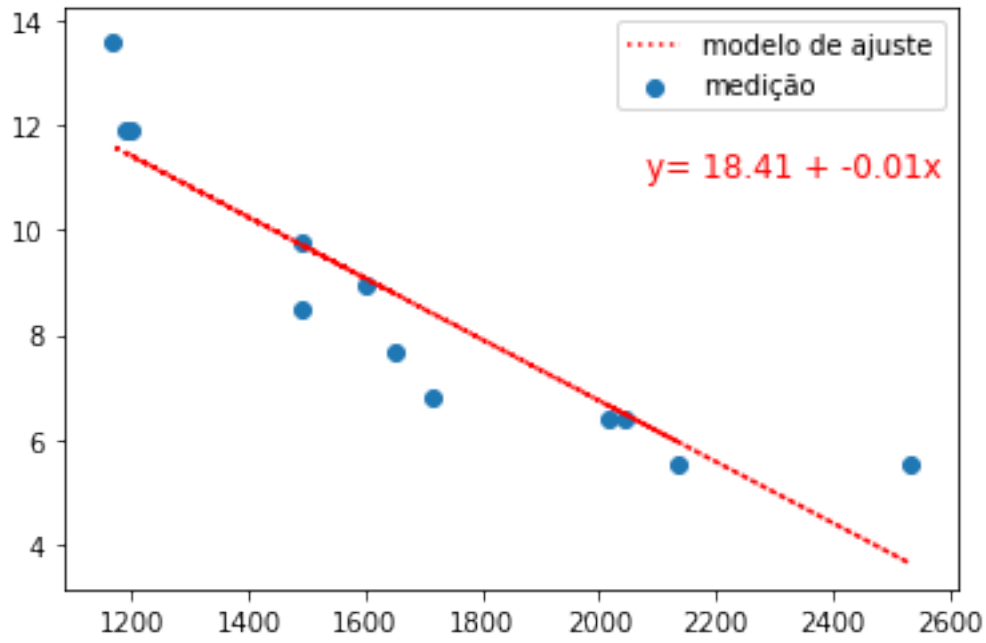
Fazemos a regressão linear:

```
a,b,R, p_value, std_err = linregress(M,C)
print(f'Regressão linear executada com a = {a:.3f}, b = {b:.3f} e R2 = {R*R:.2f}')
```

```
Regressão linear executada com a = -0.006, b = 18.410 e R2 = 0.83
```

Enfim, podemos visualizar o resultado:

```
C2 = b + a*M # ajuste
mod = plt.plot(M,C2,'r:'); # modelo
med = plt.scatter(M,C); # medição
plt.legend({'modelo de ajuste':mod, 'medição':med}); # legenda

plt.annotate('y= {0:.2f} + {1:.2f}x'.format(b,a),(2080,11),fontsize=12,c='r');
```



41.2 Medindo o desvio padrão do ajuste por mínimos quadrados

Para calcular o desvio padrão do ajuste, precisamos reconhecer o número de amostras n, o número de parâmetros do modelo de ajuste m e calcular a soma S dos quadrados. A fórmula utilizada é a seguinte:

$$\sigma = \sqrt{\frac{S}{n-m}},$$

onde $S = \sum_{k=0}^{n} [y_i - \phi(x_i)]^2$.

Notemos que se $n = m$ (caso da interpolação), $\sigma = \infty$, i.e. seria indefinido, já que o denominador anular-se-ia.

O modelo de ajuste $\phi(x)$ é considerado polinomial. Então, no caso da regressão linear, temos apenas 2 parâmetros: o coeficiente linear e o angular.

Sabemos que $m = 2$. Agora, resta usar n e calcular S. Isto é tudo de que precisamos para calcular σ para o nosso problema.

```
n = M.size # número de amostras
m = 2 # número de parâmetros.
```

O cálculo de S pode ser feito da seguinte maneira:

```
S = np.sum( (C - C2)*(C - C2) ) # soma dos quadrados (resíduos)
```

Enfim, σ será dado por:

```
sigma = np.sqrt(S/(n-m)) # desvio padrão
print(f'σ = {sigma:.3f}')
```

```
σ = 1.164
```

41.3 Ajuste polinomial (linear)

O ajuste de formas lineares de ordem superior (polinomial) pode ser realizado por meio da função `polyfit`.

41.3.1 `polyfit`

Esta função ajusta um polinômio de grau g à tabela de dados.

Os argumentos de entrada obrigatórios desta função são:

1. o primeiro conjunto de dados `x` (lista ou objeto tipo *array*)
2. o segundo conjunto de dados `y` (lista ou objeto tipo *array*)
3. o grau do polinômio `g`

O principal argumento de saída é:

- `p`: lista dos g+1 coeficientes do modelo (ordenados do maior para o menor grau)

Como importá-la?

```
from numpy import polyfit
```

Como já importamos o `numpy`, basta chamar a função com:

```
np.polyfit(x,y,deg)
```

41.3.2 Problema 2

Refaça o Problema 1 ajustando os dados com polinômios de grau 2, 3, 4 e 5 e plote os gráficos dos modelos ajustados aos dados em apenas uma figura.

Resolução

Uma vez que já temos as variáveis armazenadas na memória, basta criarmos os ajustes.

```
p2 = np.polyfit(M,C,2)
p3 = np.polyfit(M,C,3)
p4 = np.polyfit(M,C,4)
p5 = np.polyfit(M,C,5)
```

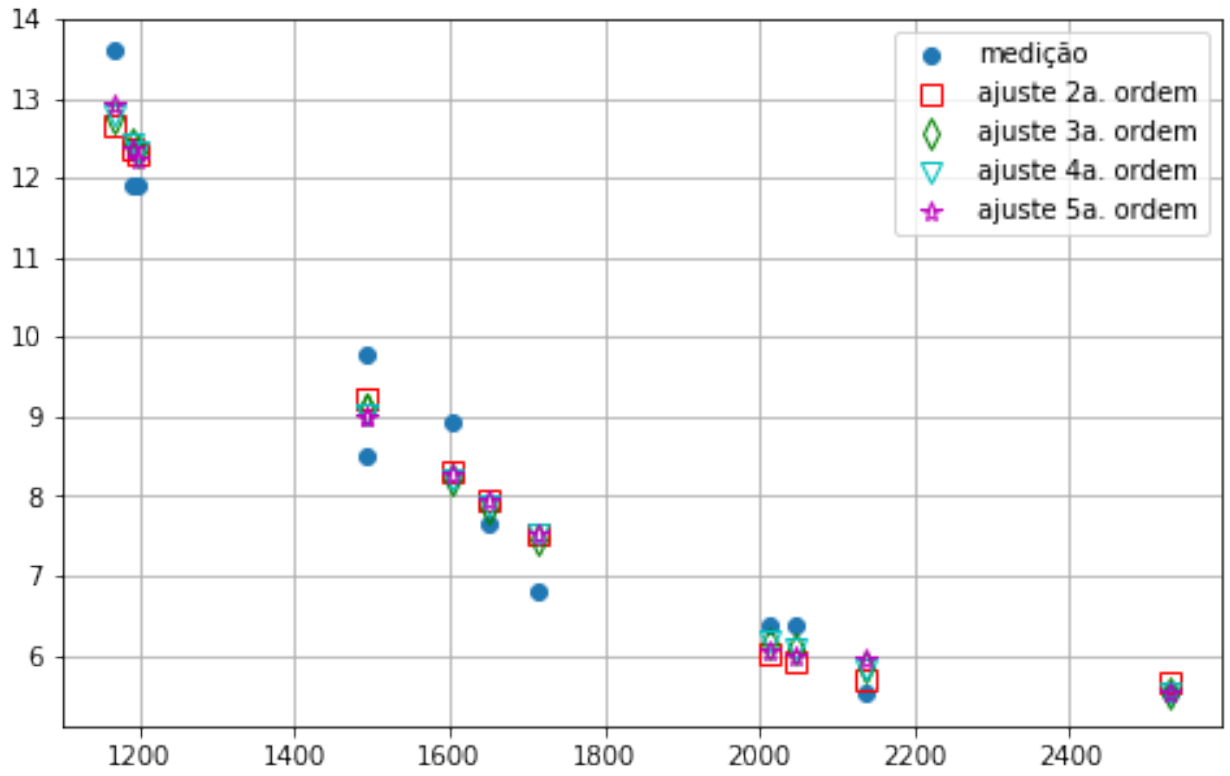
Para imprimir a lista dos coeficientes, basta fazer:

```
print(p2)
print(p3)
print(p4)
print(p5)
```

```
[ 5.26020025e-06 -2.45829326e-02  3.41991952e+01]
[-1.89135665e-09  1.56050998e-05 -4.27556580e-02  4.44279113e+01]
[ 3.37176332e-12 -2.66523211e-08  8.21041425e-05 -1.20066752e-01
  7.72222694e+01]
[-3.12853412e-14  2.86132736e-10 -1.03205187e-06  1.83990614e-03
 -1.63096318e+00  5.87812297e+02]
```

Para plotarmos as curvas, devemos nos atentar para o grau dos modelos. Podemos criá-las da seguinte forma:

```
C22 = p2[0]*M**2 + p2[1]*M    + p2[2] # modelo quadrático
C23 = p3[0]*M**3 + p3[1]*M**2 + p3[2]*M    + p3[3] # modelo cúbico
C24 = p4[0]*M**4 + p4[1]*M**3 + p4[2]*M**2 + p4[3]*M    + p4[4] # modelo de quarta␣
 ↪ordem
C25 = p5[0]*M**5 + p5[1]*M**4 + p5[2]*M**3 + p5[3]*M**2 + p5[4]*M + p5[5] # modelo de␣
 ↪quinta ordem

plt.figure(figsize=(8,5))
plt.grid(True)
med = plt.plot(M,C,'o', ms=6); # medição
mod2 = plt.plot(M,C22,'rs',ms=8, markerfacecolor='None'); # modelo 2
mod3 = plt.plot(M,C23,'gd',ms=8, markerfacecolor='None'); # modelo 3
mod4 = plt.plot(M,C24,'cv',ms=8, markerfacecolor='None'); # modelo 4
mod5 = plt.plot(M,C25,'m*',ms=8, markerfacecolor='None'); # modelo 5

plt.legend(('medição',
            'ajuste 2a. ordem',
            'ajuste 3a. ordem',
            'ajuste 4a. ordem',
            'ajuste 5a. ordem'));
```



Exercício complementar

Escreva uma função genérica que recebe a tabela de dados e o grau do modelo polinomial de ajuste e retorna os coeficientes do modelo, o desvio padrão do ajuste e os gráficos de dispersão conjuntamente com os dos modelos de ajuste.

41.3.3 Problema 3

A intensidade de radiação de uma substância radioativa foi medida em intervalos semestrais. A tabela de valores está disponível no arquivo `file-cs7-radiacao.csv`, onte t é o tempo e γ é a intensidade relativa de radiação. Sabendo que a radioatividade decai exponencialmente com o tempo, $\gamma(t) = ate^{-bt}$, estime a meia-vida radioativa (tempo no qual γ atinge metade de seu valor) da substância.

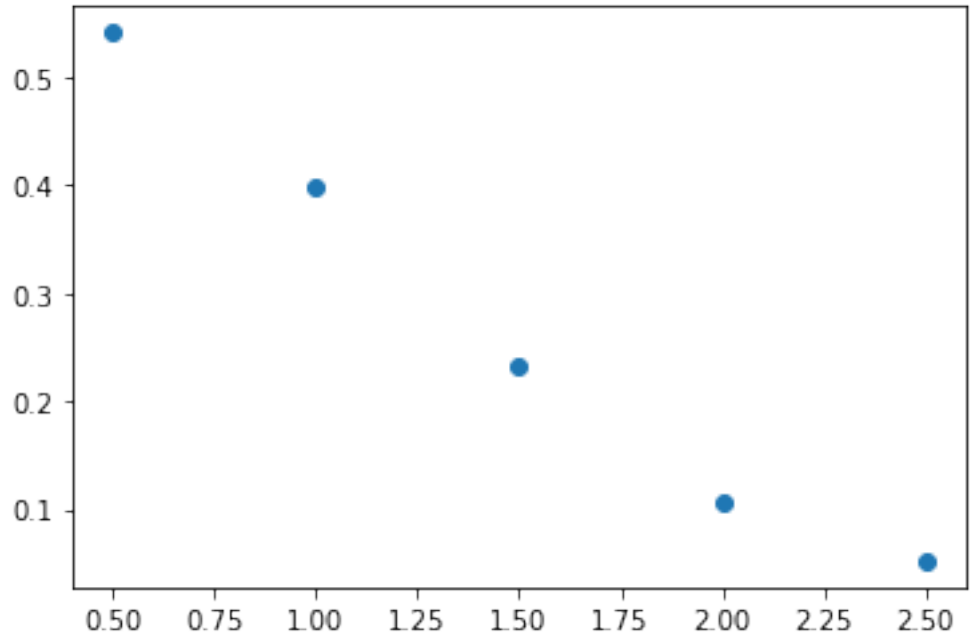
Resolução

Primeiramente, vamos ler o arquivo.

```
dados = np.loadtxt(fname='file-cs7-radiacao.csv',delimiter=',',skiprows=1)
```

Agora, vamos coletar os dados e plotá-los apenas para verificar o comportamento da dispersão.

```
t = dados[:,0]
g = dados[:,1]
plt.plot(t,g,'o');
```



Teste de alinhamento

Para ajustarmos um modelo não-linear à exponencial, antes precisamos convertê-la a uma forma linear, ou seja, linearizá-la. Para isso, aplicamos log (nome da função *logaritmo natural* em Python) em ambos os lados da função. Ao deslocar o termo t fora da exponencial para o lado esquerdo da equação, teremos:

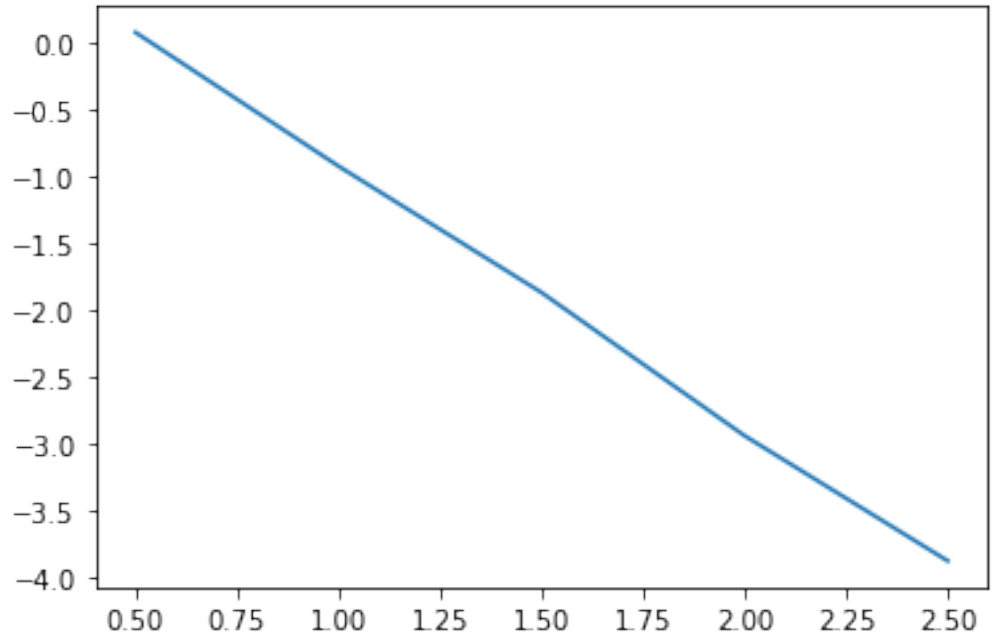
$$\log\left(\frac{\gamma}{t}\right) = \log(a) - bt$$

Definindo $z = \log(\frac{\gamma}{t})$, $x = \log(a)$, podemos agora fazer uma regressão linear nas variáveis t e z para o modelo

$$z = x - bt$$

Agora, plotando a dispersão no plano (t, z), verificamos se a curva é aproximadamente uma reta.

```
z = np.log(g/t)
plt.plot(t,z);
```



Como se vê, o teste de alinhamento mostra que a função exponencial é um modelo não-linear satisfatório para modelar o comportamento físico em questão.

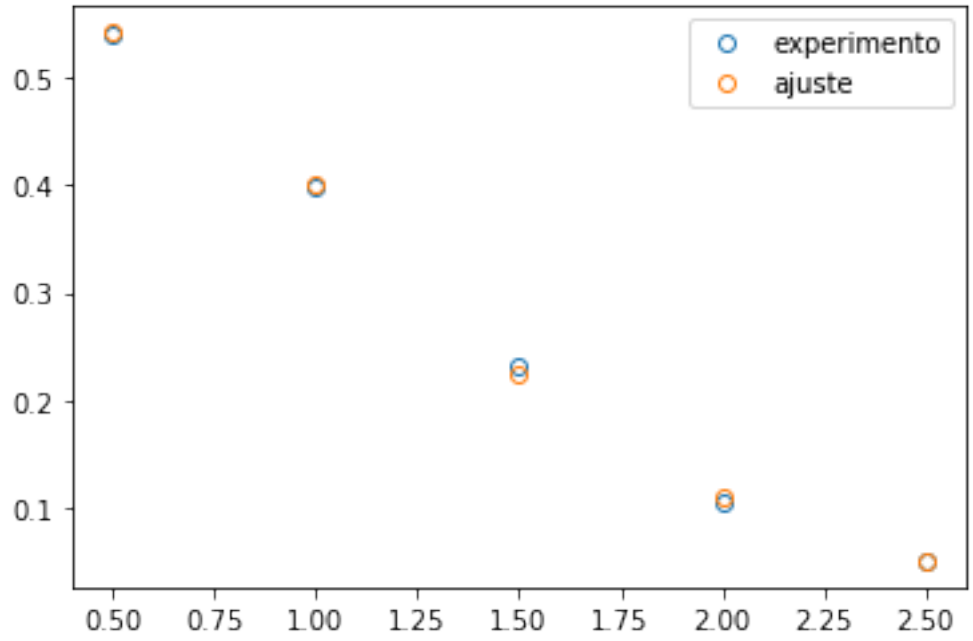
Computando a regressão linear, temos:

```
b,x,R, p_value, std_err = linregress(t,z)
print(f'Regressão linear executada com inclinação = {b:.3f}, interceptação = {x:.3f}␣
↪e R2 = {R*R:.2f}')
```

```
Regressão linear executada com inclinação = -1.984, interceptação = 1.072 e R2 = 1.00
```

Vemos que, de fato, as variáveis têm uma altíssima correlação, visto que $R^2 \approx 1$. Agora, para plotar o modelo de ajuste, recuperamos o valor de a operando inversamente e o usamos na curva do modelo para comparar com os dados experimentais.

```
a = np.exp(x); # recuperando parâmetro de ajuste
mod = lambda t: a*t*np.exp(b*t)

plt.plot(t,g,'o',mfc="None");
plt.plot(t,mod(t),'o',mfc="None");
plt.legend(('experimento','ajuste'));
```



O modelo está bem ajustado. Para estimar a meia-vida da substância, devemos encontrar o instante de tempo t_m tal que $\gamma(t_m) = 0.5\gamma_0$. Então, notemos que:

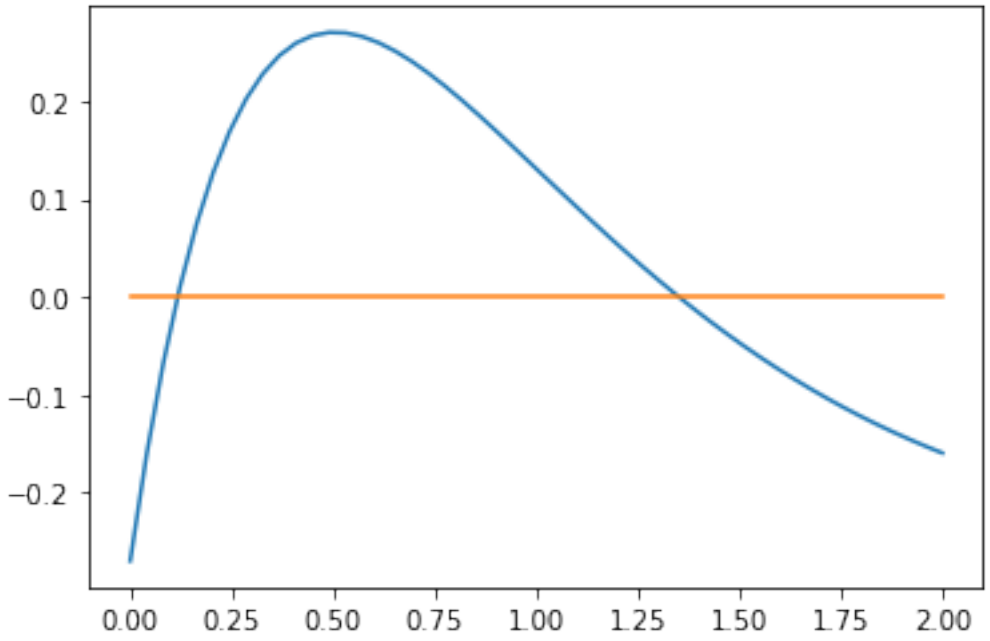
$$\gamma(t_m) = at_m e^{-bt_m} \Rightarrow 0.5\gamma_0 = at_m e^{-bt_m}$$

Todavia, não conseguimos uma relação explícita para t_m, fato que nos leva a resolver um segundo problema de determinação de raízes do tipo $f(t_m) = 0$ com

$$f(t_m) = at_m e^{-bt_m} - 0.5\gamma_0$$

Vamos resolver este problema usando a função `fsolve` do módulo `scipy.optimize`, mas antes precisamos passar a ela uma estimativa inicial. Rapidamente, façamos uma análise gráfica da curva $f(t_m)$ para $t_m = [0, 2]$ (este intervalo é obtido após algumas plotagens prévias).

```
f = lambda tm: mod(tm) - 0.5*g[0]
ttm = np.linspace(0,2)
plt.plot(ttm,f(ttm),ttm,0*f(ttm));
```



Existem duas raízes no intervalo. Porém, observando os valores tabelados de t, é fácil ver que o valor para a condição inicial deve ser maior do que $t_0 = 0.5$ e, portanto, mais próximo da segunda raiz no gráfico. Então, escolhamos para `fsolve` o valor inicial de $t_m^0 = 1.25$.

```
from scipy.optimize import fsolve

tm = fsolve(f,1.25)
print(f'Meia-vida localizada em tm = {tm[0]:.3f}.')
```

```
Meia-vida localizada em tm = 1.351.
```

Uma última verificação mostra que este valor de t_m é condizente com os dados experimentais, pois o seguinte erro é pequeno.

```
mod(tm) - 0.5*g[0]
```

```
array([1.11022302e-16])
```

CHAPTER

FORTYTWO

CODE SESSION 8

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

A integração numérica em uma variável pode ser realizada em Python utilizando a função `quad` do módulo `scipy.integrate`.

42.1 `quad`

Esta função calcula a integral definida $\int_a^b f(x)\,dx$ numericamente através de regras de quadratura.

Os argumentos de entrada obrigatórios desta função são:

1. a função `f` a ser integrada
2. o limite inferior `a`
3. o limite superior `b`

Os principais argumentos de saída são:

- `y`: valor numérico da integral
- `abserr`: estimativa do erro absoluto

Como importá-la?

```
from scipy.integrate import quad
```

```
from scipy.integrate import quad
```

42.1.1 Problema 1

O período de um pêndulo simples de comprimento L é $\tau = 4\sqrt{\frac{L}{g}}h(\theta_0)$, onde g é a aceleração gravitacional, θ_0 representa a amplitude angular, e

$$h(\theta_0) = \int_0^{\pi/2} \frac{1}{\sqrt{1 - \text{sen}^2(\theta_0/2)\text{sen}^2\theta}}\,d\theta$$

Calcule $h(\tau)$, para $\tau = 15^\circ, 30^\circ, 45^\circ$ e compare estes valores com $h(0^\circ) = \pi/2$ (a aproximação usada para pequenas amplitudes).

Resolução

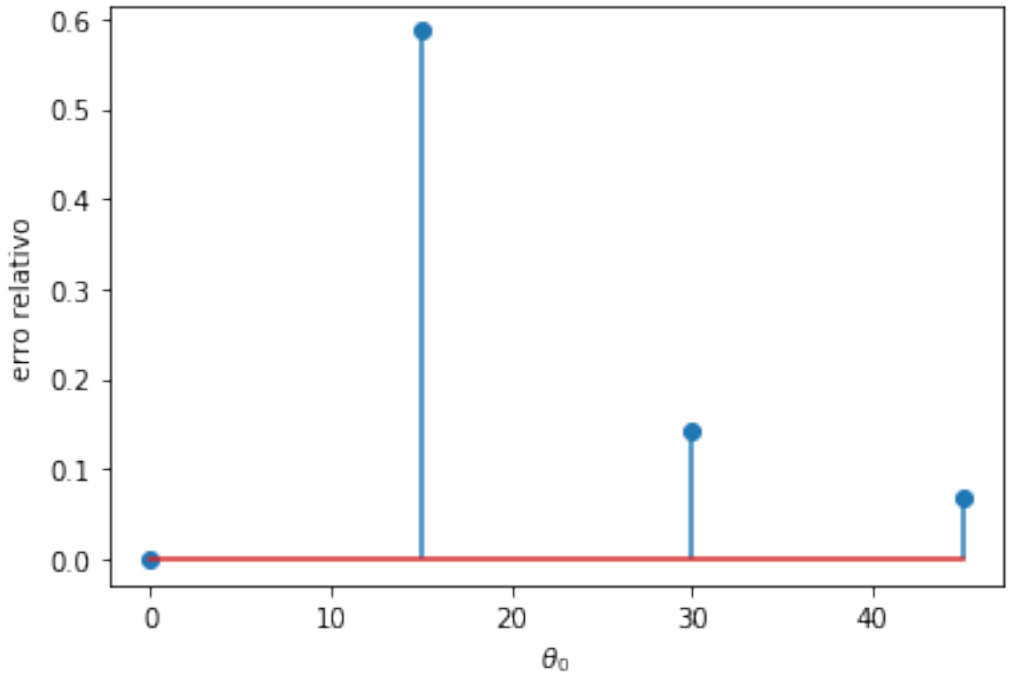
Em primeiro lugar, fazemos os cálculos diretos da integral para os distintos valores de τ.

```
# cálculo direto das integrais caso a caso

theta0 = np.array([0,15,30,45]) # ângulos

vals,errs = [],[] # integrais, erros

for t in theta0:
    f = lambda theta: 1/(np.sqrt(1. - np.sin(t/2)**2 * np.sin(theta)**2))
    v,e = quad(f,0,np.pi/2)
    print(f'Integral h({t}) = {v:g}')
    vals.append(v)
    errs.append(e)

# converte listas para arrays
vals = np.asarray(vals)
errs = np.asarray(errs)
```

```
Integral h(0) = 1.5708
Integral h(15) = 2.49203
Integral h(30) = 1.79372
Integral h(45) = 1.67896
```

Vemos que o valor das integrais é muito sensível. Para realizar uma comparação mais interessante, utilizaremos um cálculo relativo tomando o valor em $h(15)$ como referência.

```
plt.stem(theta0, (vals - vals[0])/vals[0], use_line_collection=True);
plt.xlabel('$\\theta_0$')
plt.ylabel('erro relativo');
```



Podemos verificar que a mudança de valor da integral entre 15° e 30° é da ordem de 60%, enquanto que nos demais casos, ela se limita a 20%.

42.1.2 Problema 2

Uma corrente elétrica alternada é descrita por

$$i(t) = i_0 \left(\text{sen}\left(\frac{\pi t}{t_0}\right) - \beta \, \text{sen}\left(\frac{2\pi t}{t_0}\right)\right),$$

onde $i_0 = 1\,A$, $t_0 = 0.05\,s$ e $\beta = 0.2$. Calcule a corrente RMS definida por

$$i_{rms} = \sqrt{\frac{1}{t_0}\int_0^{t_0} i^2(t)\,dt}$$

Resolução

Neste caso, basta passarmos os valores iniciais e finais para computar a integral.

```
i0, t0, beta = 1.0, 0.05, 0.2 # parâmetros iniciais

i2 = lambda t: ( i0 * ( np.sin( (np.pi*t) / t0 ) - beta * np.sin( (2*np.pi*t) / t0 )␣
 ↪) )**2 # função

i_rms = np.sqrt( 1.0/t0 * quad(i2, 0, t0)[0] )

print(f'Corrente RMS = {i_rms:g} A')
```

```
Corrente RMS = 0.72111 A
```

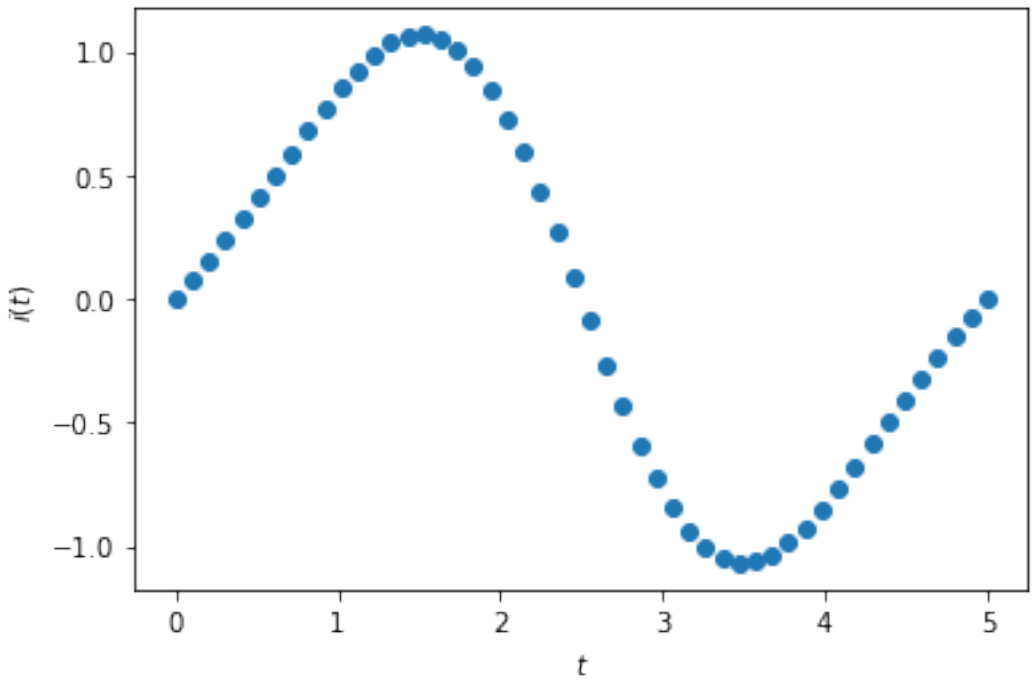
42.2 Regra do Trapézio Generalizada

A regra do trapézio generalizada (composta) pode ser calculada usando

```
scipy.integrate.cumtrapz
```

Vamos utilizar a função $i(t)$ do Problema 2 e estimar sua integral no intervalo $t = [0, 5]$ utilizando a regra do trapézio generalizada.

```
# Visualização
t = np.linspace(0,5)
i = lambda t: i0 * ( np.sin( (np.pi*t) / t0 ) - beta * np.sin( (2*np.pi*t) / t0 ) )
 ↪# função

plt.plot(t,i(t),'o');
plt.xlabel('$t$')
plt.ylabel('$i(t)$');
```




```
from scipy.integrate import cumtrapz

i = lambda t: i0 * ( np.sin( (np.pi*t) / t0 ) - beta * np.sin( (2*np.pi*t) / t0 ) ) #␣
↪função

T = cumtrapz(i(t), t)[-1] # pega último valor, já que é cumulativa

print(f'Integral por Trapézio = {T:g}')
```

```
Integral por Trapézio = -4.97605e-15
```

42.3 Quadratura Gaussiana (QG)

O cálculo de uma integral por quadratura Gaussiana pode ser calculado usando

```
scipy.integrate.quadrature
```

Vamos utilizar a função $i(t)$ do Problema 2 e estimar sua integral no intervalo $t = [0, 5]$ utilizando a QG para várias ordens (controladas pelo argumento `miniter`).

```
from scipy.integrate import quadrature

for ordem in range(1,11):
    I_QG,err_QG = quadrature(i,0,5,miniter=ordem)
    print(f'Integral por QG (ordem {ordem}) = {I_QG:g}')
```

```
Integral por QG (ordem 1) = 3.60822e-14
Integral por QG (ordem 2) = 9.71445e-15
Integral por QG (ordem 3) = 4.35416e-14
```

(continues on next page)

(continued from previous page)

```
Integral por QG (ordem 4) = 6.92502e-14
Integral por QG (ordem 5) = 6.58501e-14
Integral por QG (ordem 6) = -5.7801e-14
Integral por QG (ordem 7) = -1.39472e-14
Integral por QG (ordem 8) = -1.60288e-14
Integral por QG (ordem 9) = -5.89806e-16
Integral por QG (ordem 10) = -4.02976e-14
```

CHAPTER

FORTYTHREE

CODE SESSION 9

```
import numpy as np
import matplotlib.pyplot as plt
```

A integração numérica de uma EDO pode ser realizada em Python utilizando a função `solve_ivp` do módulo `scipy.integrate`.

43.1 `solve_ivp`

Esta função resolve um problema de valor inicial (PVI) para uma EDO ou um sistema de EDOs. Por padrão, o método de resolução é um algoritmo de Runge-Kutta com precisão de 4a. ordem.

Os argumentos de entrada obrigatórios desta função são:

1. a função `f(t,y)` a ser integrada
2. o domínio de integração, definido como uma tupla `(t0,tf)`
3. a condição inicial `y0`

O principal argumento de saída é um objeto `sol`, em que:

- `sol.t`: retorna os valores do domínio
- `sol.y`: retorna os valores da solução numérica

Como importá-la?

```
from scipy.integrate import solve_ivp
```


```
from scipy.integrate import solve_ivp
```

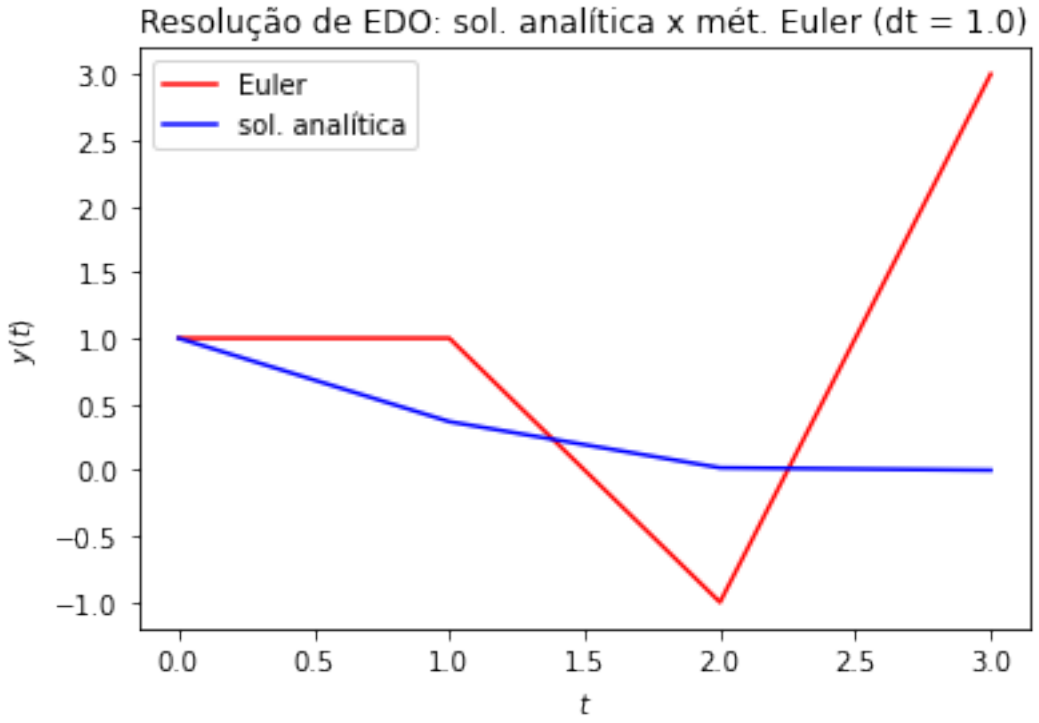

Exemplo: Resolver numericamente o PVI

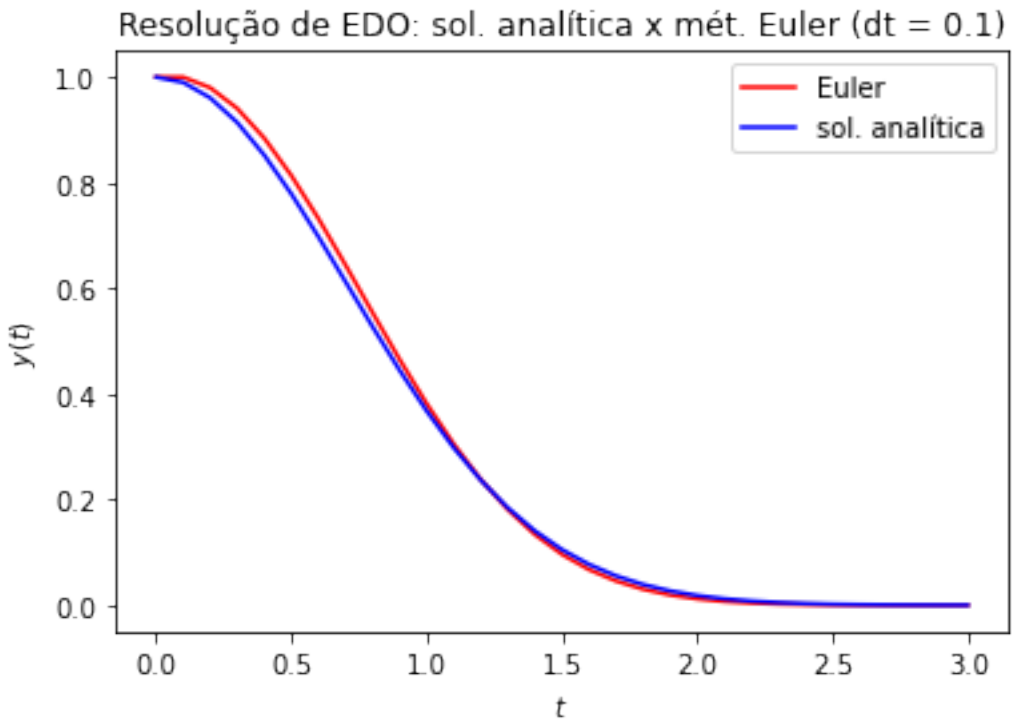
$$\begin{cases} y'(t) = -2y(t) \\ y(0) = 1 \\ 0 < t \leq 3 \end{cases}$$

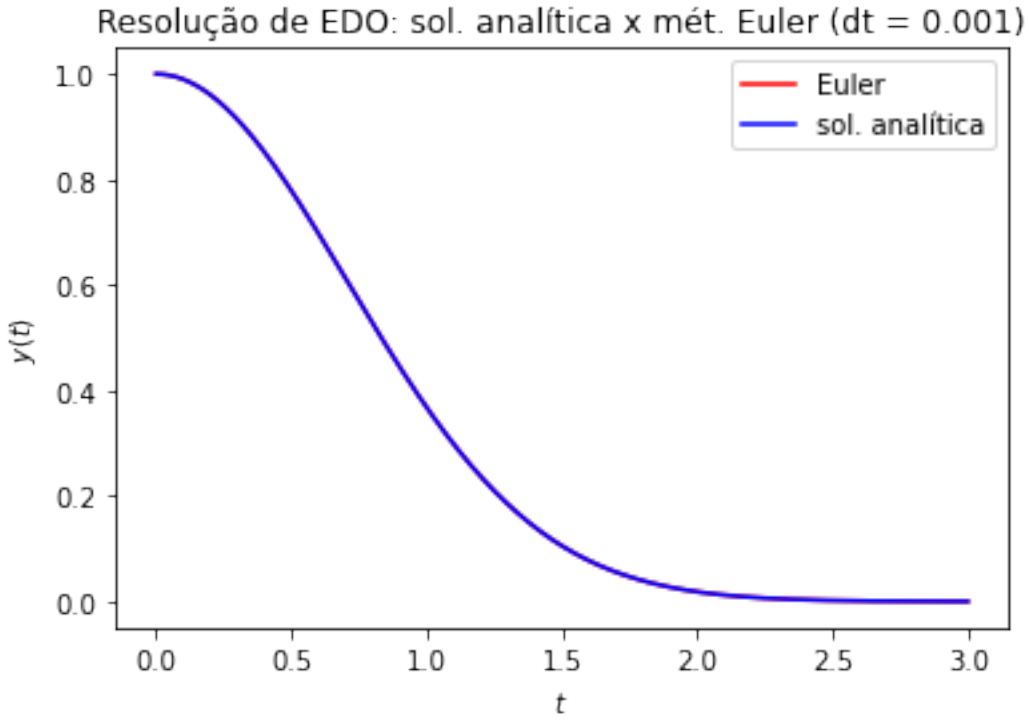
para $h = 1.0, 0.1, 0.001$.

Compare a solução numérica com a analítica: $y_{an}(t) = e^{-t^2}$.

```
f = lambda t,y: -2*y*t
yan = lambda t: np.exp(-t**2)

y0 = 1
a = 0.0
b = 3.0
T = [1.0,0.1,0.001]

for k in T:

    t = np.arange(a, b+k, k)
    y = solve_ivp(f, (a,b), [y0])

    y2 = t*0
    y2[0] = y0
    dt = (b-a)/ (len(t)-1)
    for i in range(0,len(t)-1):
        y2[i+1] = y2[i] + f(t[i],y2[i])*dt

    fig = plt.figure()
    ax = fig.add_subplot(111)
    #plt.plot(t,y,'k',label='odeint')
    plt.plot(t,y2,'r',label='Euler')
    plt.plot(t,yan(t),'b',label='sol. analítica')
    plt.legend()
    s = 'Resolução de EDO: sol. analítica x mét. Euler (dt = ' + str(k) + ')'
    plt.title(s)
    plt.xlabel('$t$')
    plt.ylabel('$y(t)$')
```







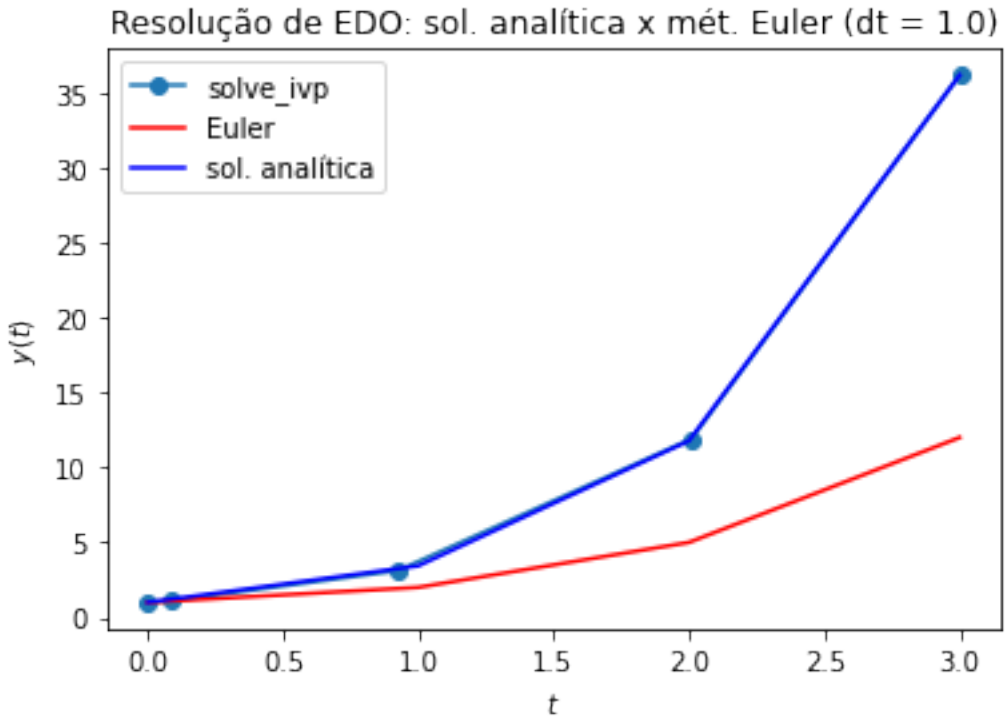
Exemplo: Resolver numericamente o PVI

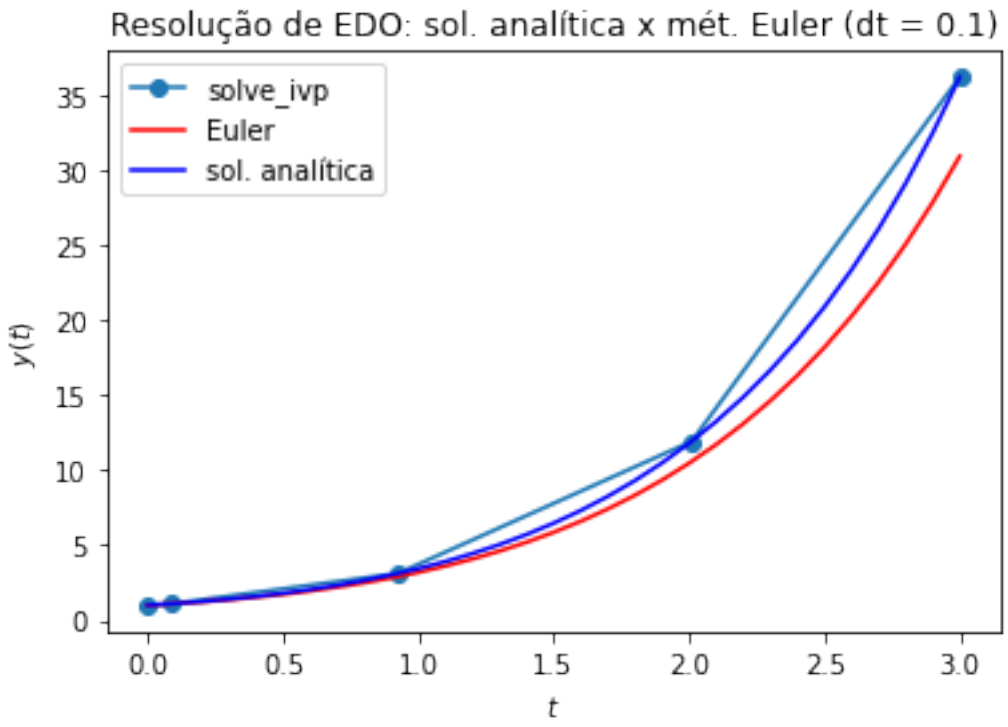
$$\begin{cases} y'(t) = y + t \\ y(0) = 1 \\ 0 < t \leq 5 \end{cases}$$

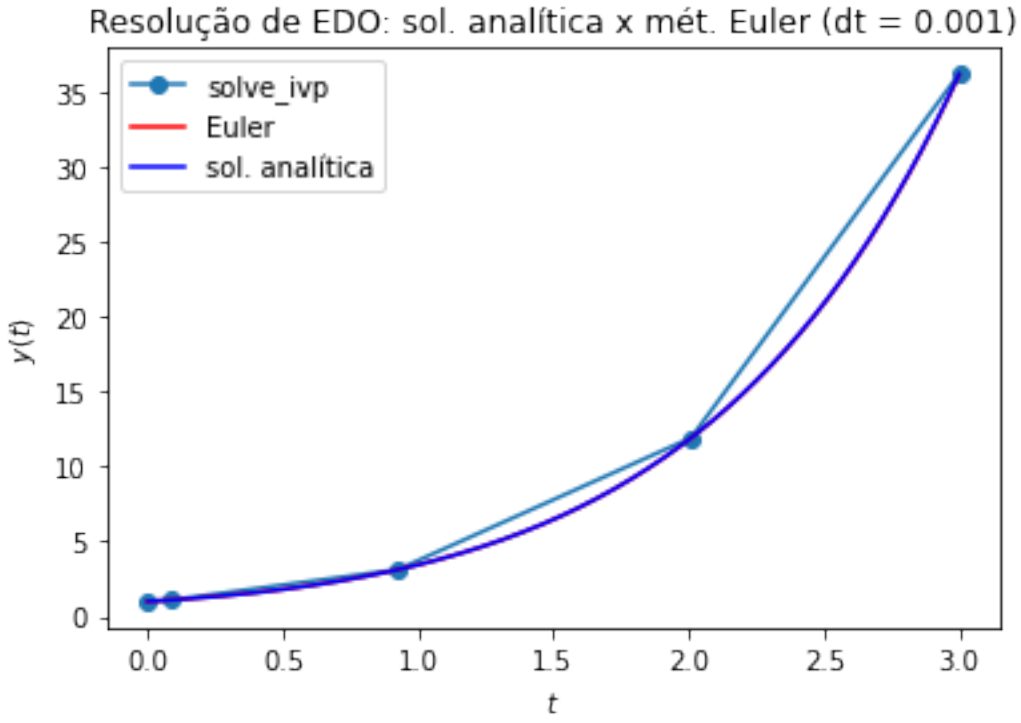
para $h = 1.0, 0.1, 0.001$.

Compare a solução numérica com a analítica: $y_{an}(t) = 2e^t - t - 1$.

```
f = lambda t,y: y + t
yan = lambda t: 2*np.exp(t) - t - 1

y0 = 1
a = 0.0
b = 3.0
T = [1.0,0.1,0.001]

for k in T:

    t = np.arange(a, b+k, k)
    sol = solve_ivp(f, (a,b), [y0])

    y2 = t*0
    y2[0] = y0
    dt = (b-a)/ (len(t)-1)
    for i in range(0,len(t)-1):
        y2[i+1] = y2[i] + f(t[i],y2[i])*dt

    fig = plt.figure()
    ax = fig.add_subplot(111)
    plt.plot(sol.t,sol.y[0],'o-',label='solve_ivp')
    plt.plot(t,y2,'r',label='Euler')
    plt.plot(t,yan(t),'b',label='sol. analítica')
    plt.legend()
    s = 'Resolução de EDO: sol. analítica x mét. Euler (dt = ' + str(k) + ')'
    plt.title(s)
    plt.xlabel('$t$')
    plt.ylabel('$y(t)$')
```








```
"""
%matplotlib inline
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from sympy import Symbol, integrate, sin, cos

# PVI
# y' = f(T,y)
# y(t0) = 1, T = (t0,tn]

# parametros
t0 = 0
tn = 10
nt = 35
y0_ex2 = 1

# var. independente
T = np.linspace(t0,tn,nt)

# EDO
f_ex2 = lambda t,y: y**2*t*(np.sin(t))

# solucao analitica simbolica
fun = 'y**2*t*(sin(t))'
tt = Symbol('t')
yan_ex2 = integrate(fun,(tt,t0,tn))
print(yan_ex2)

yan = yan_ex2.subs(t,T)

y_ex2 = solve_ivp(f_ex2, (t0,tn),[y0_ex2])

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(y_ex2.t,y_ex2.y[0],'k',label='solve_ivp')
plt.legend()
s = 'Resolução de EDO: sol. analítica x mét. Euler (dt = ' + str((tn-t0)/nt) + ')'
plt.title(s)
plt.xlabel('$t$');
plt.ylabel('$y(t)$')
""";
```

Part IX

Exercícios resolvidos

CHAPTER

FORTYFOUR

LISTA DE EXERCÍCIOS 1

Solucionário matemático e computacional de exercícios selecionados da Lista de Exercícios 1.

44.1 Exercícios computacionais

44.1.1 Q

Considere a função quadrática $f(x) = x^2 - 100.0001 + 0.01$:

a) Implemente a fórmula de Bhaskara para calcular suas raízes

b) Racionalize a fórmula de Bhaskara para a solução x_2 (multiplique por $\frac{-b+\sqrt{\Delta}}{-b+\sqrt{\Delta}}$) e refaça o cálculo para x_2.

c) Compare os resultados para x_2.

44.1.2 S

```
from math import sqrt

# a)

# fórmula de Bhaskara
a, b, c = 1, -100.0001, 0.01
Delta = b**2 - 4*a*c

# raízes
x1 = (- b + sqrt(Delta))/2*a
x2 = (- b - sqrt(Delta))/2*a

print('x1 = {0}'.format(x1))
print('x2 = {0}'.format(x2))

# b)

# Racionalizando, obtemos

x2b = 2*c/(-b + sqrt(Delta))
print('x2b = {0}'.format(x2b))


# c)
```

(continues on next page)

(continued from previous page)

```
"""
Discussão: o valor calculado em b) possui um erro de arredondamento.
           Como b < 0, o numerador envolve a subtração de
           dois números quase iguais (erro de cancelamento)
           e isto afeta o resultado.

           Com a racionalização da expressão, obtemos uma fórmula
           menos propensa a erros.
"""
b2 = sqrt(Delta)
print(b2)
print('Diferença no numerador: ' + str(-b - b2))
print('Erro absoluto na solução: ' + str(abs(x2 - x2b)))
```

```
x1 = 100.0
x2 = 0.00010000000000331966
x2b = 0.0001
99.9999
Diferença no numerador: 0.000200000000066393
Erro absoluto na solução: 3.3196508692975857e-15
```

CHAPTER

FORTYFIVE

LISTA DE EXERCÍCIOS 2

Solucionário matemático e computacional de exercícios selecionados da Lista de Exercícios 2.

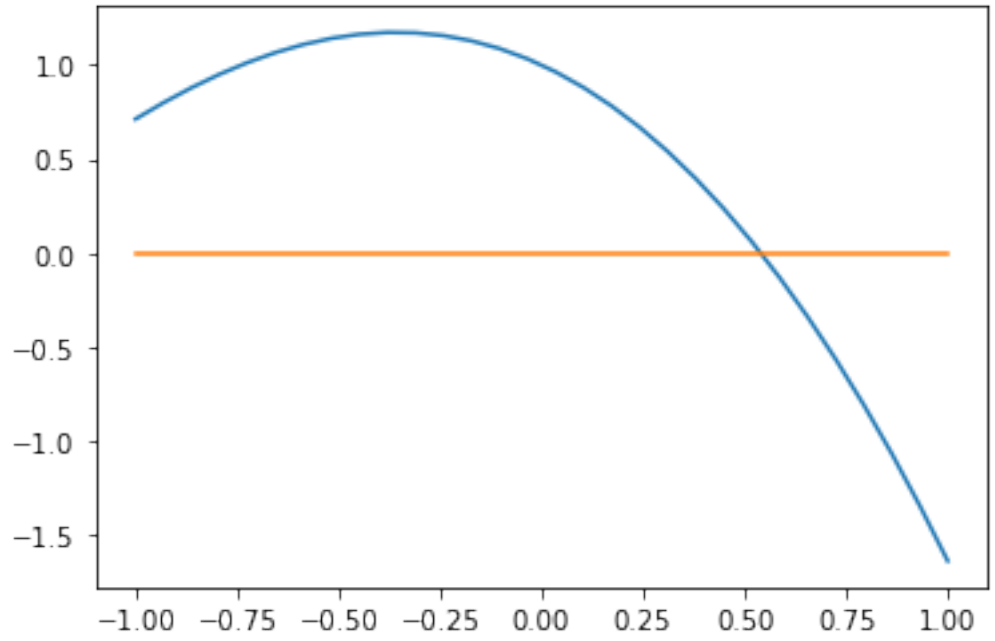
```
%matplotlib inline
```

```
import metodosRaizes as rz
import numpy as np
import sympy as sy
import matplotlib.pyplot as plt
```

45.1 Q12 - L2

Primeiramente, recorramos à análise gráfica da função.

```
from numpy import cos, exp

# funcao
p = lambda x: 2*cos(x) - exp(x)

# plotagem
x = np.linspace(-1,1,30)
plt.plot(x,p(x),x,0*p(x));
```



Vemos que a raiz está próxima de $x = 0.5$. Assim, qualquer valor próximo a este é razoável como aproximação inicial.

Agora, antes de aplicar o método de Newton, vamos calcular a derivada da função simbolicamente.

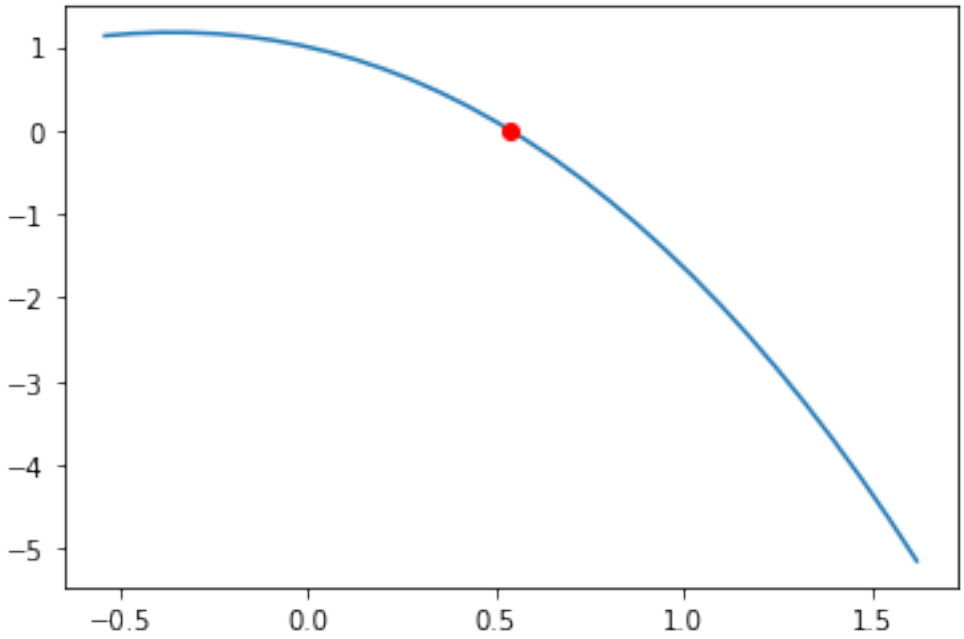
```
# derivada simbolica
xsym = sy.Symbol('x')
sy.diff(2*sy.cos(xsym) - sy.exp(xsym),xsym)
```

$$-e^x - 2\sin(x)$$

Resolvamos com a nossa função programada.

```
x0 = 0.2
tol = 1e-3
f = '2*cos(x) - exp(x)'
df = '-2*sin(x) - exp(x)'
xr = rz.newton(x0,f,df,tol,10,True)
```

```
Estimativa inicial: x0 = 0.2

i         x                 f(x)                 df(x)                 ER
0         0.200000         0.738730         -1.618741         -
1         0.656361         -0.343328         -3.148240         6.952896e-01
2         0.547307         -0.020734         -2.769371         1.992555e-01
3         0.539820         -0.000096         -2.743662         1.386900e-02
4         0.539785         -0.000000         -2.743542         6.499799e-05
Solução obtida: x = 0.5397851616
```



Verificação:

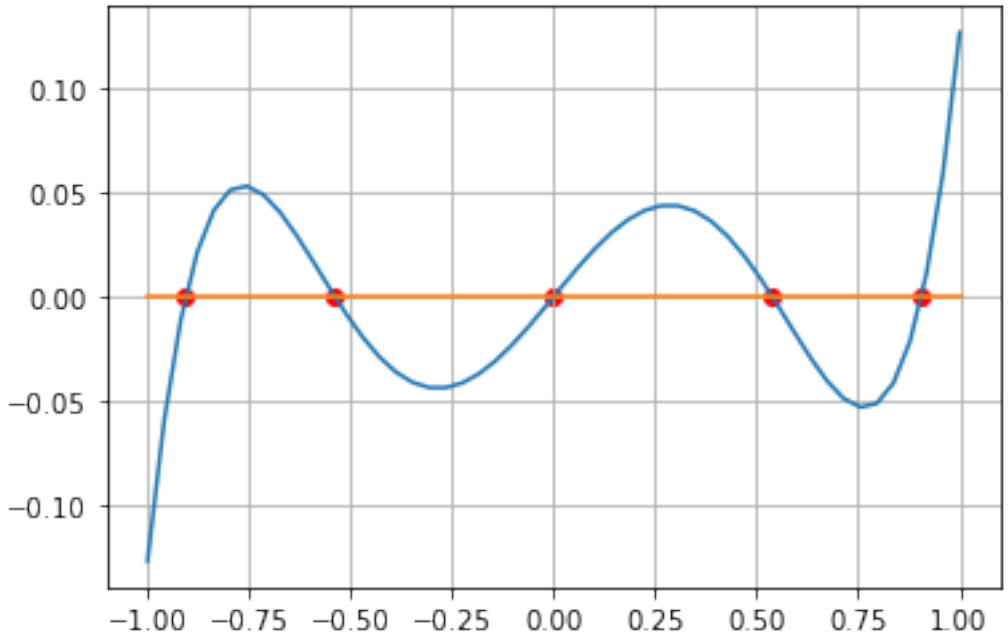
```
2*cos(xr) - exp(xr)
```

```
-2.1118811144305027e-09
```

45.2 Q16 - L2

```
p = lambda x: x**5 - 10/9*x**3 + 5/21*x
x = np.linspace(-1,1,50,endpoint=True)
plt.plot(x,p(x),x,0*p(x))

xr = np.roots([1,0,-10/9,0,5/21,0])
print(xr)

plt.scatter(xr,0*xr,c='r');
plt.grid(True);
```

```
[-0.90617985 -0.53846931  0.90617985  0.53846931  0.        ]
```



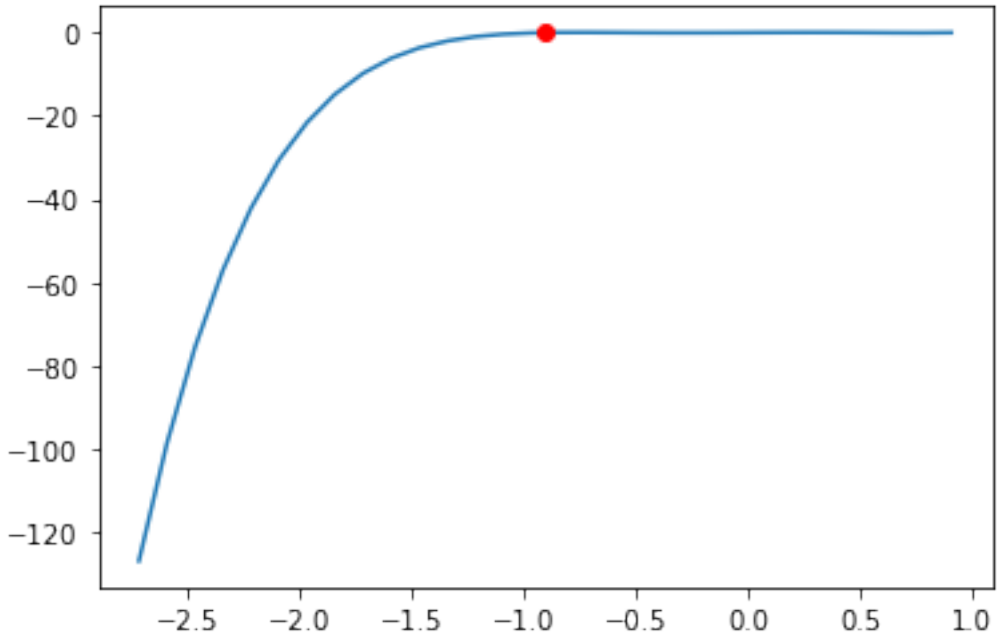

```
tol = 1e-3
nmax = 100
```

45.2.1 item b.1)

```
# Aproximacao de xi1 pelo MNR
x0 = -0.8
p = 'x**5 - 10/9*x**3 + 5/21*x'
dp = '5*x**4 -10/3*x**2 + 5/21'
plotar = True
rz.newton(x0,p,dp,tol,nmax,True)
```

```
Estimativa inicial: x0 = -0.8

i          x                  f(x)                  df(x)                  ER
0          -0.800000          0.050733          0.152762          -
1          -1.132103          -0.517006          4.179132          2.933506e-01
2          -1.008392          -0.143443          2.018543          1.226817e-01
3          -0.937329          -0.031683          1.169040          7.581417e-02
4          -0.910227          -0.003604          0.908555          2.977501e-02
5          -0.906261          -0.000070          0.873137          4.376698e-03
6          -0.906180          -0.000000          0.872424          8.905774e-05
Solução obtida: x = -0.9061798789
```

```
-0.9061798789481786
```




```
ff = lambda x: x**5 - 10/9*x**3 + 5/21*x
dff = lambda x: 5*x**4 - 30/9*x**2 + 5/21

xx = 0.857
ff(xx),dff(xx)
```

```
(-0.03303209894521281, 0.4870085727669044)
```

45.2.2 item b.2)

```
# Aproximacao de xi2 pelo MB

var = 'x'
a,b = -0.75, -0.25
rz.bissecao(p,a,b,tol,100,var)
```

```
i=1 a=-0.750000 b=-0.250000 xm=-0.500000 f(a)=0.052874 f(b)=-0.043139 f(xm)=-0.011409␣
 ↪b-a=0.500000
i=2 a=-0.750000 b=-0.500000 xm=-0.625000 f(a)=0.052874 f(b)=-0.011409 f(xm)=0.027090␣
 ↪b-a=0.250000
i=3 a=-0.625000 b=-0.500000 xm=-0.562500 f(a)=0.027090 f(b)=-0.011409 f(xm)=0.007512␣
 ↪b-a=0.125000
i=4 a=-0.562500 b=-0.500000 xm=-0.531250 f(a)=0.007512 f(b)=-0.011409 f(xm)=-0.002211␣
 ↪b-a=0.062500
i=5 a=-0.562500 b=-0.531250 xm=-0.546875 f(a)=0.007512 f(b)=-0.002211 f(xm)=0.002605␣
 ↪b-a=0.031250
i=6 a=-0.546875 b=-0.531250 xm=-0.539062 f(a)=0.002605 f(b)=-0.002211 f(xm)=0.000183␣
 ↪b-a=0.015625
Solução encontrada: -0.5390625
```

```
-0.5390625
```

45.2.3 item b.3)

```
# Aproximacao de xi3 pelo MFP
a,b = -0.35, 0.25
rz.falsa_posicao(p,a,b,tol,100,var)
```

```
i=1 a=-0.350000 b=0.250000 xm=-0.057823 f(a)=-0.040947 f(b)=0.043139 f(xm)=-0.013553␣
↪b-a=0.600000
i=2 a=-0.057823 b=0.250000 xm=0.015767 f(a)=-0.013553 f(b)=0.043139 f(xm)=0.003750 b-
↪a=0.307823
i=3 a=-0.057823 b=0.015767 xm=-0.000181 f(a)=-0.013553 f(b)=0.003750 f(xm)=-0.000043␣
↪b-a=0.073590
Solução encontrada: -0.00018060189341005293
```

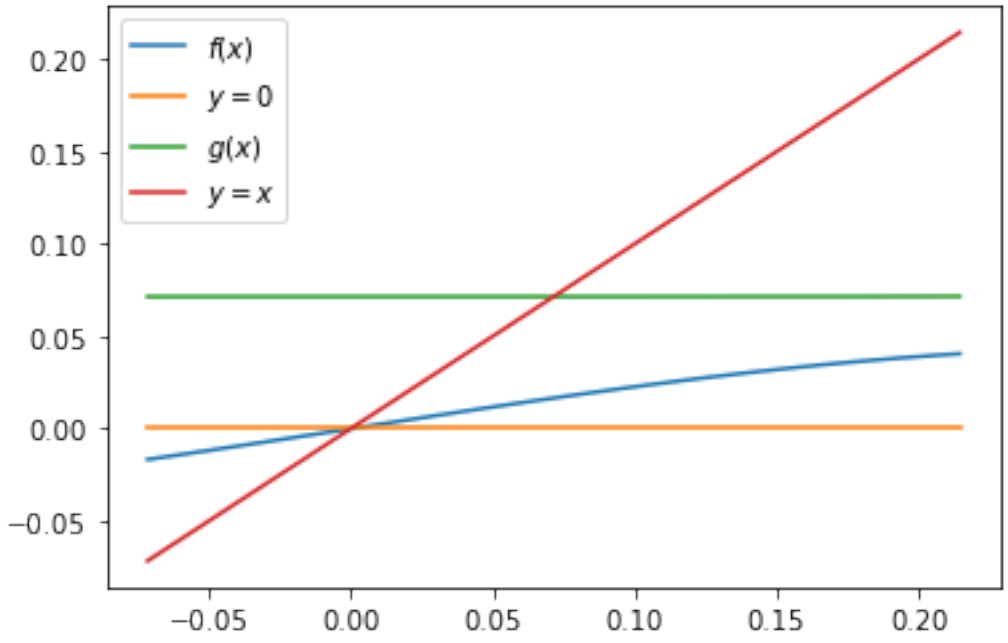
```
-0.00018060189341005293
```

45.2.4 item b.4)

```
# Aproximacao de xi4 pelo MPF

g = '( 0.9*(5/21 + x**5))**1./3.'
#g = eval('lambda x:' + g)

#plt.plot(x,g(x))
rz.ponto_fixo(0.4,p,g,tol,nmax,True)
```

```
i          x                 f(x)                ER
0          0.400000          0.034367          2.500000e+00
1          0.074501          0.017281          4.369086e+00
2          0.071429          0.016604          4.299795e-02
3          0.071429          0.016604          1.829703e-06
```

```
0.07142912925875246
```

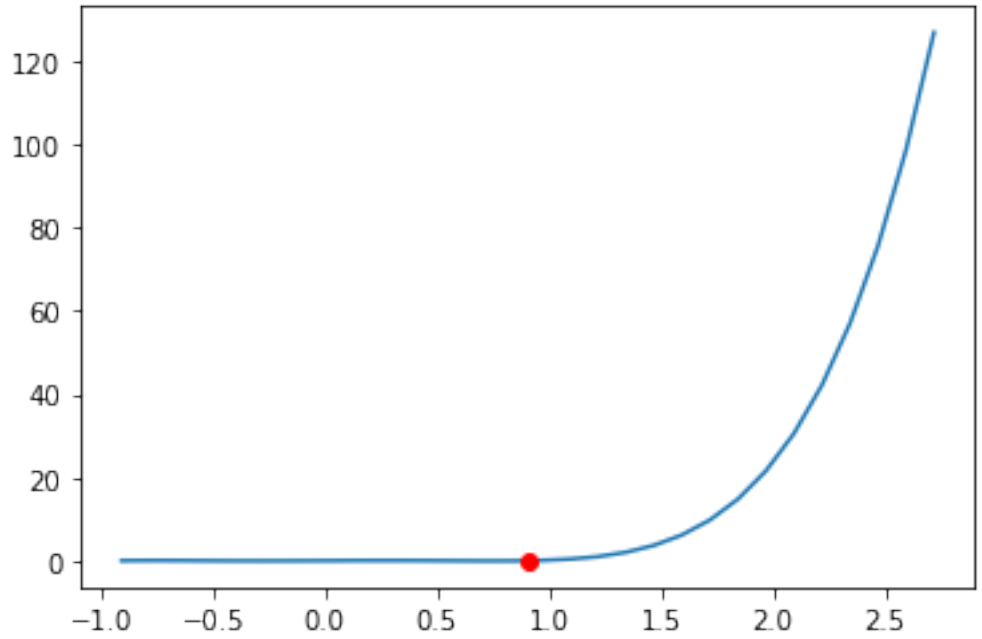


45.2.5 item b.5)

```
# Aproximacao de xi5 pelo MS

p = 'x**5 - 10/9*x**3 + 5/21*x'
x0,x1=0.8,1.
rz.secante(x0,x1,p,tol,nmax,True)
```

```
Estimativas iniciais: xa = 0.8; xb = 1.0

i          x                  f(x)                  ER
1          0.857094          -0.032986          1.667334e-01
2          0.886562          -0.015467          3.323813e-02
3          0.912577          0.005764          2.850796e-02
4          0.905514          -0.000579          7.799970e-03
5          0.906159          -0.000018          7.112008e-04
Solução obtida: x = 0.9061586909
```

```
0.9061586908755153
```



45.3 Exemplos

```
# exemplo bissecao
rz.bissecao('x**3-9*x+3',0,1,1e-3,100,var)
```

```
i=1 a=0.000000 b=1.000000 xm=0.500000 f(a)=3.000000 f(b)=-5.000000 f(xm)=-1.375000 b-
 ↪a=1.000000
i=2 a=0.000000 b=0.500000 xm=0.250000 f(a)=3.000000 f(b)=-1.375000 f(xm)=0.765625 b-
 ↪a=0.500000
i=3 a=0.250000 b=0.500000 xm=0.375000 f(a)=0.765625 f(b)=-1.375000 f(xm)=-0.322266 b-
 ↪a=0.250000
i=4 a=0.250000 b=0.375000 xm=0.312500 f(a)=0.765625 f(b)=-0.322266 f(xm)=0.218018 b-
 ↪a=0.125000
i=5 a=0.312500 b=0.375000 xm=0.343750 f(a)=0.218018 f(b)=-0.322266 f(xm)=-0.053131 b-
 ↪a=0.062500
i=6 a=0.312500 b=0.343750 xm=0.328125 f(a)=0.218018 f(b)=-0.053131 f(xm)=0.082203 b-
 ↪a=0.031250
i=7 a=0.328125 b=0.343750 xm=0.335938 f(a)=0.082203 f(b)=-0.053131 f(xm)=0.014474 b-
 ↪a=0.015625
i=8 a=0.335938 b=0.343750 xm=0.339844 f(a)=0.014474 f(b)=-0.053131 f(xm)=-0.019344 b-
 ↪a=0.007812
i=9 a=0.335938 b=0.339844 xm=0.337891 f(a)=0.014474 f(b)=-0.019344 f(xm)=-0.002439 b-
 ↪a=0.003906
i=10 a=0.335938 b=0.337891 xm=0.336914 f(a)=0.014474 f(b)=-0.002439 f(xm)=0.006017 b-
 ↪a=0.001953
i=11 a=0.336914 b=0.337891 xm=0.337402 f(a)=0.006017 f(b)=-0.002439 f(xm)=0.001789 b-
 ↪a=0.000977
Solução encontrada: 0.33740234375
```

```
0.33740234375
```

```
rz.falsa_posicao('x**3-9*x+3',0,1,1e-3,100,var)
```

```
i=1 a=0.000000 b=1.000000 xm=0.375000 f(a)=3.000000 f(b)=-5.000000 f(xm)=-0.322266 b-
 ↪a=1.000000
i=2 a=0.000000 b=0.375000 xm=0.338624 f(a)=3.000000 f(b)=-0.322266 f(xm)=-0.008790 b-
 ↪a=0.375000
i=3 a=0.000000 b=0.338624 xm=0.337635 f(a)=3.000000 f(b)=-0.008790 f(xm)=-0.000226 b-
 ↪a=0.338624
Solução encontrada: 0.33763504551140067
```

```
0.33763504551140067
```

45.4 Problemas aplicados

A temperatura T (em graus Kelvin) de um semicondutor pode ser calculada como uma função da resistência R (em ohms) pela equação de Steinhart-Hart:

$$T = \frac{1}{A + B\ln(R) + C\ln(R)^3},$$

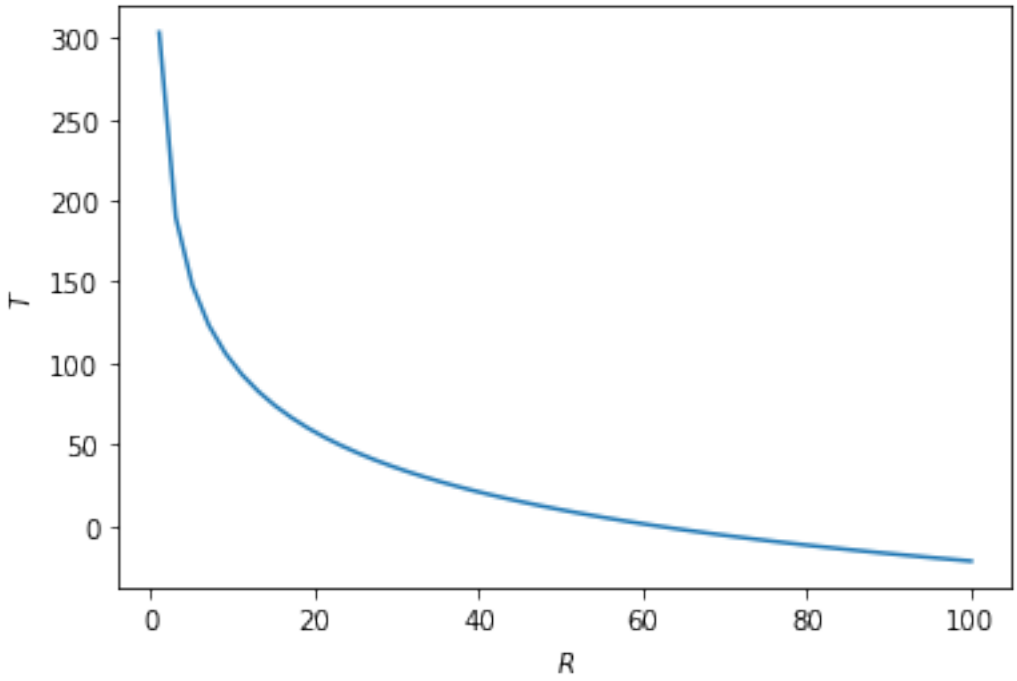
para constantes A, B e C dependentes do material. Supondo que $A = 1.4056 \times 10^{-3}$, $B = 240.7620 \times 10^{-6} K^{-1}$ e $C = 7.48 \times 10^{-7}$, determine a resistência do semicondutor para uma temperatura de 135 graus Celsius.

Sugestão: use o método da bisseção e tolerância de 10^{-6}.

45.4.1 Solução computacional

```
from scipy.optimize import bisect

# constantes
A = 1.4056e-3
B = 240.7620e-6
C = 7.48e-7
Tc = 135.0 # temperatura em Celsius
Tk = Tc + 273.15 # temperatura em Kelvin

# f(R) = 0
f = lambda R: 1/(A + B*np.log(R) + C*np.log(R)**3) - Tk

# localização
R = np.linspace(1,100,50,True)
plt.plot(R,f(R))
plt.xlabel('$R$')
plt.ylabel('$T$')

# refinamento a=1; b=100
x = bisect(f,1,100,xtol=1e-6)
print("Para {0:.2f} graus Celsius, a resistência é de {1:.2f} ohms".format(Tc,x))
```

```
Para 135.00 graus Celsius, a resistência é de 61.61 ohms
```



45.5 Problema aplicado (revisar!)

Quando se calcula o pagamento de uma hipoteca, a relação entre o montante do empréstimo E, o pagamento mensal P, a duração do empréstimo em anos a, e a taxa de juros anual t é dada pela equação:

$$P = \frac{Et}{12\left(1 - \frac{1}{(1+\frac{t}{12})^{12a}}\right)}.$$

Determine a taxa t de um empréstimo de R$ 170.000 00 por 20 anos se o pagamento mensal for de R$ 1.250,00.

Sugestão: usar método de Newton como função programada pelo estudante ou função residente.

45.5.1 Solução computacional:

Resolver o problema $f(t) = 0$ com método de Newton.

```
from scipy.optimize import newton
from sympy.utilities.lambdify import lambdastr

E = 170000 # montante do empréstimo
P = 1250 # pagamento mensal
a = 20 # duração do empréstimo em anos

# define função para computar
# t: taxa de juros (incógnita)
def taxa_anuidade(E,P,a):

    Esym,t,asym,Psym = sy.symbols('Esym t asym Psym')
```

(continues on next page)

(continued from previous page)

```
    fsym = Esym*t/( 12*(1 - 1/(1 + t/12)**(12*asym) ) ) - Psym
    dfsym = sy.diff(fsym,t)

    f = fsym.subs({'Esym':E,'asym':a,'Psym':P})
    df = dfsym.subs({'Esym':E,'asym':a,'Psym':P})

    ft = eval(lambdastr(t,f))
    dft = eval(lambdastr(t,df))

    # Aplicação do método de Newton

    # parâmetros
    t0 = 1.0 # estimativa inicial
    tole=1e-3 # tolerância de erro
    nmax=50 # número máximo de iterações

    # método de Newton
    t = newton(ft,t0,dft,tol=tole,maxiter=nmax)
    msg = "A taxa do empréstimo é de {0:.2f}% a.a.".format(t*100)

    return (t,msg)

# invoca função
taxa_anuidade(E,P,a)
# não sei se a equação do livro está correta: negativo até 12
# aa = np.arange(1,50)
# tt = []
# for i in aa:
#     t=taxa_anuidade(E,P,i)
#     print(t[1])
#     tt.append(t[0])
# tt = np.asarray(tt)
# plt.plot(aa,tt)
```

```
(0.0632479251521274, 'A taxa do empréstimo é de 6.32% a.a.')
```

CHAPTER

FORTYSIX

LISTA DE EXERCÍCIOS 3

Solucionário matemático e computacional de exercícios selecionados da Lista de Exercícios 3.

```
%matplotlib inline
```

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Funções Implementadas

def gauss_simples(AB):
    '''Realiza o cálculo de um sistema linear através do método de eliminação de␣
 ↪Gauss sem pivotamento.

    Sinopse:
        X = gauss_simples(AB)

    Entradas:
        AB - Matriz aumentada (np.array) do sistema linear

    Saídas:
        X - Vetor solução do sistema linear

        @ney
    '''

    # Eliminação progressiva de variáveis
    for i in range(len(AB)):
        for j in range(len(AB)):
            if (j > i):
                m = AB[j,i]/AB[i,i] # Fator multiplicador
                AB[j,:] -= m*AB[i,:] # Eliminação de variável

    # Substituição regressiva
    A = AB[:,0:-1]
    B = AB[:,-1]
    X = np.zeros((len(AB),1))

    for i in range(len(A)-1, -1, -1):
        for j in range(len(A)):
            X[i] -= A[i,j]*X[j]
        X[i] += B[i]
        X[i] /= A[i,i]
```

(continues on next page)

(continued from previous page)

```
    X = np.around(X, decimals=3)

    return X

def gauss_pivparc(AB):
    '''Realiza o cálculo de um sistema linear através do método de eliminação de␣
 ↪Gauss com pivotamento parcial.

    Sinopse:
         X = gauss_pivparc(AB)

    Entradas:
         AB - Matriz aumentada (np.array) do sistema linear

    Saídas:
         X - Vetor solução do sistema linear

         @ney
    '''

    # Eliminação progressiva de variáveis
    for i in range(len(AB)):
        if (i < len(AB)): # Pivotamento parcial
            AB_new = np.flip(AB[i:, i:], 0)
            AB[i:, i:] = AB_new
        for j in range(len(AB)):
            if (j > i):
                m = AB[j,i]/AB[i,i] # Fator multiplicador
                AB[j,:] -= m*AB[i,:] # Eliminação de variável

    # Substituição regressiva
    A = AB[:,0:-1]
    B = AB[:,-1]
    X = np.zeros((len(AB),1))

    for i in range(len(A)-1, -1, -1):
        for j in range(len(A)):
            X[i] -= A[i,j]*X[j]
        X[i] += B[i]
        X[i] /= A[i,i]

    X = np.around(X, decimals=3)

    return X

def gaussjordan(AB):
    '''Realiza o cálculo de um sistema linear através do método de eliminação de␣
 ↪Gauss-Jordan simples.

    Sinopse:
         X = gaussjordan(AB)

    Entradas:
         AB - Matriz aumentada (np.array) do sistema linear

    Saídas:
         X - Vetor solução do sistema linear
```

(continues on next page)

(continued from previous page)

```
        @ney
    '''

    for i in range(len(AB)):
        AB[i,:] /= AB[i,i]
        for j in range(len(AB)):
            if (j != i):
                AB[j,:] -= AB[j,i]*AB[i,:]

    AB = np.around(AB, decimals=3)

    return AB[:,-1]

def lu_solver(AB):
    '''Realiza o cálculo de um sistema linear através do método de decomposição LU.

    Sinopse:
        L, U, X = lu_solver(AB)

    Entradas:
        AB - Matriz aumentada (np.array) do sistema linear

    Saídas:
        L (lu[0]) - Matriz triangular inferior
        U (lu[1]) - Matriz triangular superior
        X (lu[2]) - Vetor solução do sistema linear

        @ney
    '''

    # Decomposição LU
    U = np.copy(AB)
    U = U[:,0:-1]
    L = np.zeros((len(AB),len(AB)))

    for i in range(len(AB)):
        for j in range(len(AB)):
            if (j > i):
                m = U[j,i]/U[i,i]
                U[j,:] -= m*U[i,:]
                L[j,i] = m
            elif (j == i):
                L[i,i] = 1

    # Substituição progressiva
    B = np.copy(AB)
    B = B[:,-1]
    D = np.zeros((len(AB),1))

    for i in range(len(L)):
        for j in range(len(L)):
            if (i > j):
                D[i] -= L[i,j]*D[j]
        D[i] += B[i]

    # Substituição regressiva
```

(continues on next page)

(continued from previous page)

```
    X = np.zeros((len(AB),1))

    for i in range(len(U)-1, -1, -1):
        for j in range(len(U)):
            X[i] -= U[i,j]*X[j]
        X[i] += D[i]
        X[i] /= U[i,i]

    L = np.around(L, decimals=3)
    U = np.around(U, decimals=3)
    X = np.around(X, decimals=3)

    return (L, U, X)

def cholesky(AB):
    '''Realiza o cálculo de um sistema linear através do método LU, por decomposição
 ↪de Cholesky.

    Sinopse:
        C = cholesky(AB)

    Entradas:
        AB - Matriz aumentada (np.array) do sistema linear

    Saídas:
        G (C[0]) - Matriz triangular inferior
        X (C[1]) - Vetor solução do sistema linear

        @ney
    '''

    # Decomposição de Cholesky
    A = np.copy(AB)
    A = A[:,0:-1]
    G = np.zeros((len(A),len(A)))

    for k in range(len(A)):
        for i in range(k):
            s1 = 0
            for j in range(i):
                s1 += G[i,j]*G[k,j]
            G[k,i] = (A[k,i] - s1)/G[i,i]
        s2 = 0
        for j in range(k):
            s2 += (G[k,j])**2
        G[k,k] = np.sqrt(A[k,k] - s2)

    # Substituição progressiva
    B = np.copy(AB)
    B = B[:,-1]
    D = np.zeros((len(AB),1))

    for i in range(len(G)):
        for j in range(len(G)):
            if (i > j):
                D[i] -= G[i,j]*D[j]
        D[i] += B[i]
```

(continues on next page)

(continued from previous page)

```
        D[i] /= G[i,i]

    # Substituição regressiva
    GT = G.transpose()
    X = np.zeros((len(AB),1))

    for i in range(len(GT)-1, -1, -1):
        for j in range(len(GT)):
            X[i] -= GT[i,j]*X[j]
        X[i] += D[i]
        X[i] /= GT[i,i]

    G = np.around(G, decimals=3)
    X = np.around(X, decimals=3)

    return G,X
```

46.1 Sistemas lineares

46.1.1 Questão 1

Escreva o seguinte conjunto de equações na forma matricial:

$$8 = 6x_3 + 2x_2 \tag{1}$$
$$2 - x_1 = x_3 \tag{2}$$
$$5x_2 + x_1 = 13 \tag{3}$$

Mulitiplique a matriz dos coeficientes por sua transposta, i.e. AA^T.

```
# Solução

A = np.array([[0, 2, 6], [1, 0, 1], [1, 5, 0]])
AT = A.transpose()
B = np.dot(A,AT)

print("A matriz dos coeficientes desse sistema é: \n\n", A)
print("\nA sua transposta é: \n\n", AT)
print("\nLogo, o produto entre A e sua transposta é: \n\n", B)
```

```
A matriz dos coeficientes desse sistema é:

 [[0 2 6]
 [1 0 1]
 [1 5 0]]

A sua transposta é:

 [[0 1 1]
 [2 0 5]
 [6 1 0]]

Logo, o produto entre A e sua transposta é:
```

(continues on next page)

(continued from previous page)

```
[[40  6 10]
 [ 6  2  1]
 [10  1 26]]
```

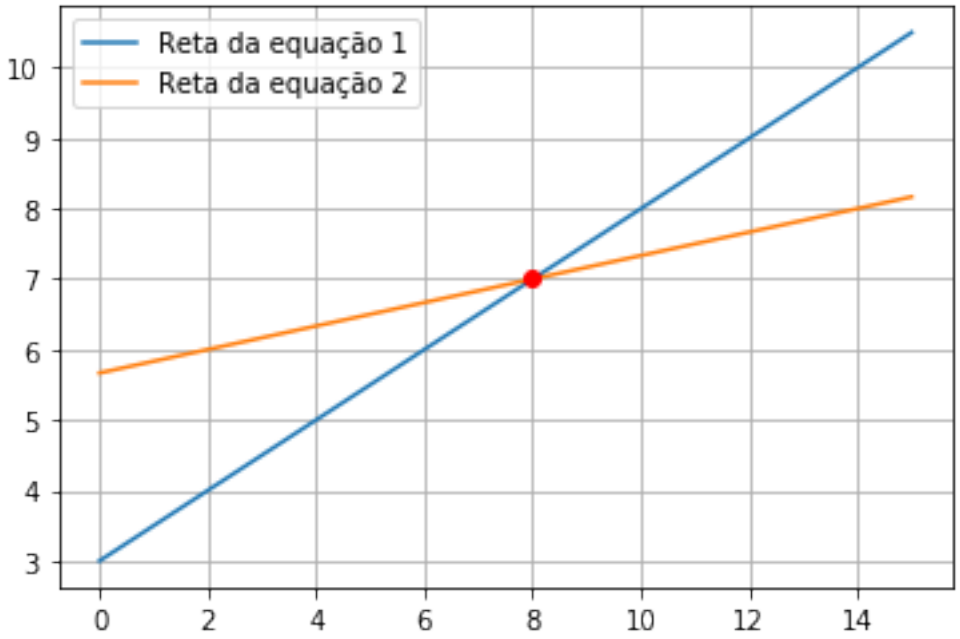
46.1.2 Questão 2

Use o metodo gráfico para resolver:

$$4x_1 - 8x_2 = -24 \tag{4}$$
$$-x_1 + 6x_2 = 34 \tag{5}$$

```
# Solução

def f1(x):
    return (4*x + 24)/8 # x2 isolado na primeira equação

def f2(x):
    return (x + 34)/6 # x2 isolado na segunda equação

x1 = np.linspace(0, 15, 100)

y1 = f1(x1) # Valores de x2 na primeira equação
y2 = f2(x1) # Valores de x2 na segunda equação

plt.plot(x1,y1, label="Reta da equação 1")
plt.plot(x1,y2, label="Reta da equação 2")
plt.grid()
plt.legend()
plt.plot(8,7,'or')
```

```
[<matplotlib.lines.Line2D at 0x7f84f958d8d0>]
```



46.1.3 Questão 3

Para o conjunto de equações:

$$2x_2 + 5x_3 = 9 \tag{6}$$
$$2x_1 + x_2 + x_3 = 9 \tag{7}$$
$$3x_1 + x_2 = 10 \tag{8}$$

(i) Calcule o determinante.

(ii) Use a regra de Cramer para determinar x_1, x_2, x_3.

```
# Solução

# (i)
A = np.array([[0, 2, 5], [2, 1, 1], [3, 1, 0]])
det = np.linalg.det(A)
print("(i) det(A) = ", det)

# (ii)
A = tuple(A)
B = np.array([[9], [9], [10]])
X = []

for coluna in range(len(A)):
    C = np.asarray(A)
    for linha in range(len(A)):
        C[linha][coluna] = B[linha]
    X.append(np.linalg.det(C)/np.linalg.det(np.asarray(A)))

X = [round(i,3) for i in X]

print("\n(ii) Os valores de [x1, x2, x3] são:", X)
```

```
(i) det(A) =  0.9999999999999991

(ii) Os valores de [x1, x2, x3] são: [6.0, -8.0, 5.0]
```

46.1.4 Questão 4

Dadas as equações:

$$10x_1 + 2x_2 - x_3 = 27 \tag{9}$$
$$-3x_1 - 6x_2 + 2x_3 = -61.5 \tag{10}$$
$$x_1 + 5x_2 + 5x_3 = -21.5 \tag{11}$$

Resolva por Eliminação de Gauss Simples. Mostre todos os passos de cálculo.

```
# Solução

AB = np.array([[10., 2., -1., 27], [-3., -6., 2., 61.5], [1., 5., 5., -21.5]])

# Eliminação progressiva de variáveis

f = 0;

for i in range(len(AB)):
    for j in range(len(AB)):
        if (j > i):
            m = AB[j,i]/AB[i,i] # Fator multiplicador
            f += 1
            print("\nO fator multiplicativo", f, "é:", m)
            AB[j,:] -= m*AB[i,:] # Eliminação de variável
            print("Após a eliminação de variável, temos:\n", AB)

# Substituição regressiva

A = AB[:,0:-1]
B = AB[:,-1]
X = np.zeros((3,1))

for i in range(len(A)-1, -1, -1):
    for j in range(len(A)):
        X[i] -= A[i,j]*X[j]
    X[i] += B[i]
    X[i] /= A[i,i]

X = np.around(X, decimals=3)

print("\nA solução do problema é: \n", X)
```

```
O fator multiplicativo 1 é: -0.3
Após a eliminação de variável, temos:
 [[ 10.    2.   -1.   27. ]
 [  0.   -5.4   1.7  69.6]
 [  1.    5.    5.  -21.5]]

O fator multiplicativo 2 é: 0.1
Após a eliminação de variável, temos:
```

(continues on next page)

(continued from previous page)

```
 [[ 10.    2.   -1.   27. ]
 [  0.   -5.4   1.7  69.6]
 [  0.    4.8   5.1 -24.2]]

O fator multiplicativo 3 é: -0.8888888888888888
Após a eliminação de variável, temos:
 [[10.          2.         -1.          27.        ]
 [ 0.         -5.4         1.7        69.6       ]
 [ 0.          0.          6.61111111 37.66666667]]

A solução do problema é:
 [[  5.489]
 [-11.095]
 [  5.697]]
```

46.1.5 Questão 5

Use Eliminação de Gauss com pivotamento parcial para resolver:

$$8x_1 + 2x_2 - 2x_3 = -2 \tag{12}$$
$$10x_1 + 2x_2 - 4x_3 = 4 \tag{13}$$
$$12x_1 + 2x_2 + 2x_3 = 6 \tag{14}$$

```
# Solução

AB = np.array([[8., 2., -2., -2], [10., 2., -4., 4], [12., 2., 2., 6]])
X = gauss_pivparc(AB)

print("A solução do problema é: \n", X)
```

```
A solução do problema é:
 [[  2.5]
 [-11.5]
 [ -0.5]]
```

46.1.6 Questão 6

Dadas as equações:

$$2x_1 - 6x_2 - x_3 = -38 \tag{15}$$
$$-3x_1 - x_2 + 7x_3 = 34 \tag{16}$$
$$-8x_1 + x_2 - 2x_3 = -20 \tag{17}$$

Resolva por Eliminação de Gauss com pivotamento parcial.

```
# Solução

AB = np.array([[2., -6., -1., -38], [-3., -1., 7., 34], [-8., 1., -2., -20]])
X = gauss_pivparc(AB)
print("A solução do problema é: \n", X)
```

```
A solução do problema é:
 [[1.63 ]
 [5.812]
 [6.386]]
```

46.1.7 Questão 7

Use Eliminação de Gauss-Jordan para resolver:

$$2x_1 + x_2 - x_3 = 1 \tag{18}$$
$$5x_1 + 2x_2 + 2x_3 = -4 \tag{19}$$
$$3x_1 + x_2 + x_3 = 5 \tag{20}$$

Não utilize pivotamento. Substitua seus resultados nas equações originais para verificá-los.

```
# Solução

AB = np.array([[2., 1., -1., 1], [5., 2., 2., -4], [3., 1., 1., 5]])
X = gaussjordan(AB)
print("A solução do problema é", X)
```

```
A solução do problema é [ 14. -32.  -5.]
```

46.1.8 Questão 8

Resolva:

$$x_1 + x_2 - x_3 = -3 \tag{21}$$
$$6x_1 + 2x_2 + 2x_3 = 2 \tag{22}$$
$$-3x_1 + 4x_2 + x_3 = 1 \tag{23}$$

(i) Por Eliminação de Gauss simples.

(ii) Por Eliminação de Gauss com pivotamento parcial.

(iii) Por Eliminação de Gauss-Jordan sem pivotamento parcial.

```
# Solução

# (i) Eliminação de Gauss simples
AB = np.array([[1., 1., -1., -3], [6., 2., 2., 2], [-3., 4., 1., 1]])
X = gauss_simples(AB)
print("(i) A solução do problema, pelo método da eliminação de Gauss simples, é: \n",␣
 ↪X)

# (ii) Eliminação de Gauss com pivotamento parcial
AB = np.array([[1., 1., -1., -3], [6., 2., 2., 2], [-3., 4., 1., 1]])
X = gauss_pivparc(AB)
print("\n(ii) A solução do problema, pelo método da eliminação de Gauss com␣
 ↪pivotamento parcial, é: \n", X)

# (iii) Eliminação de Gauss-Jordan sem pivotamento parcial
AB = np.array([[1., 1., -1., -3], [6., 2., 2., 2], [-3., 4., 1., 1]])
```

(continues on next page)

(continued from previous page)

```
X = gaussjordan(AB)
print("\n(ii) A solução do problema, pelo método da eliminação de Gauss-Jordan sem␣
↪pivotamento parcial, é: \n", AB[:,-1])
```

```
(i) A solução do problema, pelo método da eliminação de Gauss simples, é:
 [[-0.25]
 [-0.5 ]
 [ 2.25]]

(ii) A solução do problema, pelo método da eliminação de Gauss com pivotamento␣
↪parcial, é:
 [[-0.25]
 [-0.5 ]
 [ 2.25]]

(ii) A solução do problema, pelo método da eliminação de Gauss-Jordan sem pivotamento␣
↪parcial, é:
 [-0.25 -0.5   2.25]
```

46.2 Fatoração LU

46.2.1 Questão 9

Resolva o seguinte sistema de equações por decomposição LU sem pivotamento (usando a definição A = LU):

$$8x_1 + 4x_2 - x_3 = 11 \tag{24}$$
$$-2x_1 + 5x_2 + x_3 = 4 \tag{25}$$
$$2x_1 - x_2 + 6x_3 = 7 \tag{26}$$

Em seguida, determine a matriz inversa $A^{-1} = U^{-1}L^{-1}$. Verifique seus resultados comprovando que $AA^{-1} = I$.

```
#Solução
AB = np.array([[8., 4., -1., 11.], [-2., 5., 1., 4.], [2., -1., 6., 7.]])

L, U, X = lu_solver(AB)

A = np.copy(AB)
A = A[:,0:-1]

print("A solução do sistema é: \n", X)
print("\nA inversa de A é: \n", np.around(np.linalg.inv(A), decimals=3))
print("\nO produto entre as inversas de U e L é: \n", np.around(np.dot(np.linalg.
↪inv(U),np.linalg.inv(L)), decimals=3))
print("\nO produto entre A e sua inversa é: \n", np.around(np.dot(A,np.linalg.
↪inv(A))))
```

```
A solução do sistema é:
 [[1.]
 [1.]
 [1.]]

A inversa de A é:
```

(continues on next page)

(continued from previous page)

```
 [[ 0.099 -0.074  0.029]
 [ 0.045  0.16  -0.019]
 [-0.026  0.051  0.154]]

O produto entre as inversas de U e L é:
 [[ 0.099 -0.074  0.029]
 [ 0.045  0.16  -0.019]
 [-0.026  0.051  0.154]]

O produto entre A e sua inversa é:
 [[ 1.  0.  0.]
 [-0.  1.  0.]
 [-0.  0.  1.]]
```

46.2.2 Questão 10

Resolva o seguinte sistema de equações por decomposição LU com pivotamento parcial.

$$2x_1 - 6x_2 - x_3 = -38 \tag{27}$$
$$-3x_1 - x_2 + 7x_3 = -34 \tag{28}$$
$$-8x_1 + x_2 - 2x_3 = -20 \tag{29}$$

Em seguida, determine a matriz inversa $A^{-1} = U^{-1}L^{-1}$. Verifique seus resultados comprovando que $AA^{-1} = I$.

```
#Solução
AB = np.array([[2., -6., -1., -38.], [-3., -1., 7., -34.], [-8., 1., -2., -20.]])

L, U, X = lu_solver(AB)

A = np.copy(AB)
A = A[:,0:-1]

print("A matriz L é: \n", L)
print("\nA matriz U é: \n", U)
print("\nA solução do sistema é: \n", X)
print("\nA inversa de A é: \n", np.around(np.linalg.inv(A), decimals=3))
print("\nO produto entre as inversas de U e L é: \n", np.around(np.dot(np.linalg.
 ↪inv(U),np.linalg.inv(L)), decimals=3))
print("\nO produto entre A e sua inversa é: \n", np.around(np.dot(A,np.linalg.
 ↪inv(A))))
```

```
A matriz L é:
 [[ 1.   0.   0. ]
 [-1.5  1.   0. ]
 [-4.   2.3  1. ]]

A matriz U é:
 [[  2.    -6.    -1.  ]
 [  0.   -10.     5.5 ]
 [  0.     0.   -18.65]]

A solução do sistema é:
 [[ 4.]
 [ 8.]
```

(continues on next page)

(continued from previous page)

```
 [-2.]]

A inversa de A é:
 [[-0.013 -0.035 -0.115]
 [-0.166 -0.032 -0.029]
 [-0.029  0.123 -0.054]]

O produto entre as inversas de U e L é:
 [[-0.013 -0.035 -0.115]
 [-0.166 -0.032 -0.029]
 [-0.029  0.123 -0.054]]

O produto entre A e sua inversa é:
 [[ 1. -0. -0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

46.2.3 Questão 11

Resolva o seguinte sistema de equações por decomposição LU.

$$10x_1 + 2x_2 - x_3 = 27 \tag{30}$$
$$-3x_1 - 6x_2 + 2x_3 = -61.5 \tag{31}$$
$$x_1 + x_2 + 5x_3 = -21.5 \tag{32}$$

Em seguida, determine a matriz inversa $A^{-1} = U^{-1}L^{-1}$. Verifique seus resultados comprovando que $AA^{-1} = I$.

```
#Solução
AB = np.array([[10., 2., -1., 27.], [-3., -6., 2., -61.5], [1., 1., 5., -21.5]])

L, U, X = lu_solver(AB)

A = np.copy(AB)
A = A[:,0:-1]

print("A matriz L é: \n", L)
print("\nA matriz U é: \n", U)
print("\nA solução do sistema é: \n", X)
print("\nA inversa de A é: \n", np.around(np.linalg.inv(A), decimals=3))
print("\nO produto entre as inversas de U e L é: \n", np.around(np.dot(np.linalg.
 ↪inv(U),np.linalg.inv(L)), decimals=3))
print("\nO produto entre A e sua inversa é: \n", np.around(np.dot(A,np.linalg.
 ↪inv(A))))
```

```
A matriz L é:
 [[ 1.     0.     0.   ]
 [-0.3    1.     0.   ]
 [ 0.1   -0.148  1.   ]]

A matriz U é:
 [[10.     2.    -1.   ]
 [ 0.    -5.4    1.7  ]
 [ 0.     0.     5.352]]
```

(continues on next page)

(continued from previous page)

```
A solução do sistema é:
 [[ 0.5]
 [ 8. ]
 [-6. ]]

A inversa de A é:
 [[ 0.111  0.038  0.007]
 [-0.059 -0.176  0.059]
 [-0.01   0.028  0.187]]

O produto entre as inversas de U e L é:
 [[ 0.111  0.038  0.007]
 [-0.059 -0.176  0.059]
 [-0.01   0.028  0.187]]

O produto entre A e sua inversa é:
 [[ 1. -0.  0.]
 [ 0.  1.  0.]
 [-0. -0.  1.]]
```

46.3 Fatoração de Cholesky

46.3.1 Questão 12

Determine quais matrizes a seguir são (i) simétricas, (ii) singulares, (iii) diagonalmente dominantes, (iv) positivas definidas.

(a) \begin{pmatrix} 2 & 1 \ 1 & 3 \end{pmatrix}

(b) \begin{pmatrix} 2 & 1 & 0 \ 0 & 3 & 0 \ 1 & 0 & 4 \end{pmatrix}

© \begin{pmatrix} 4 & 2 & 6 \ 3 & 0 & 7 \ -2 & -1 & -3 \end{pmatrix}

(d) \begin{pmatrix} 4 & 0 & 0 & 0 \ 6 & 7 & 0 & 0 \ 9 & 11 & 1 & 0 \ 5 & 4 & 1 & 1 \end{pmatrix}

```
# Solução

A = np.array([[2., 1.], [1., 3.]])
B = np.array([[2., 1., 0.], [0., 3., 0.], [1., 0., 4.]])
C = np.array([[4., 2., 6.], [3., 0., 7.], [-2., -1., -3.]])
D = np.array([[4., 0., 0., 0.], [6., 7., 0., 0.], [9., 11., 1., 0.], [5., 4., 1., 1.
 ↪]])

menA = []
for i in range(1,len(A)+1):
    menA.append(np.linalg.det(A[0:i, 0:i]))
print("Os determinantes dos menores principais de (a) são:", np.around(menA,␣
 ↪decimals=3))

menB = []
for i in range(1,len(B)+1):
    menB.append(np.linalg.det(B[0:i, 0:i]))
print("\nOs determinantes dos menores principais de (b) são:", np.around(menB,␣
 ↪decimals=3))
```

(continues on next page)

(continued from previous page)

```
menC = []
for i in range(1,len(C)+1):
    menC.append(np.linalg.det(C[0:i, 0:i]))
print("\nOs determinantes dos menores principais de (c) são:", np.around(menC,␣
↪decimals=3))

menD = []
for i in range(1,len(D)+1):
    menD.append(np.linalg.det(D[0:i, 0:i]))
print("\nOs determinantes dos menores principais de (d) são:", np.around(menD,␣
↪decimals=3))
```

```
Os determinantes dos menores principais de (a) são: [2. 5.]

Os determinantes dos menores principais de (b) são: [ 2.  6. 24.]

Os determinantes dos menores principais de (c) são: [ 4. -6.  0.]

Os determinantes dos menores principais de (d) são: [ 4. 28. 28. 28.]
```

Solução:

(i) Apenas a matriz em (a) e simétrica.

(ii) Apenas a matriz em (d) é singular.

(iii) As matrizes em (a) e (b) são diagonalmente dominantes.

(iv) As matrizes em (a), (b) e (d) são positivas-definidas.

46.3.2 Questão 13

Determine a fatoração de Cholesky $A = GGT$ das matrizes a seguir.

(a) \begin{pmatrix} 2 & −1 & 0 \ −1 & 2 & −1 \ 0 & −1 & 2 \end{pmatrix}

(b) \begin{pmatrix} 4 & 1 & 1 & 1 \ 1 & 3 & −1 & 1 \ 1 & −1 & 2 & 0 \ 1 & 1 & 0 & 2 \end{pmatrix}

© \begin{pmatrix} 4 & 1 & −1 & 0 \ 1 & 3 & −1 & 0 \ −1 & −1 & 5 & 2 \ 0 & 0 & 2 & 4 \end{pmatrix}

```
# Solução

# (a)
print("(a)")
A = np.array([[2., -1., 0.], [-1., 2., -1.], [0., -1., 2.]])
G = np.zeros((len(A),len(A)))

for k in range(len(A)):
    for i in range(k):
        s1 = 0
        for j in range(i):
            s1 += G[i,j]*G[k,j]
        G[k,i] = (A[k,i] - s1)/G[i,i]
    s2 = 0
    for j in range(k):
```

(continues on next page)

(continued from previous page)

```
        s2 += (G[k,j])**2
    G[k,k] = np.sqrt(A[k,k] - s2)

G = np.around(G, decimals=3)

print("A matriz G é:\n", G)

# (b)
print("\n(b)")
B = np.array([[4., 1., 1., 1.], [1., 3., -1., 1.], [1., -1., 2., 0.], [1., 1., 0., 2.
 ↪]])
G = np.zeros((len(B),len(B)))

for k in range(len(B)):
    for i in range(k):
        s1 = 0
        for j in range(i):
            s1 += G[i,j]*G[k,j]
        G[k,i] = (B[k,i] - s1)/G[i,i]
    s2 = 0
    for j in range(k):
        s2 += (G[k,j])**2
    G[k,k] = np.sqrt(B[k,k] - s2)

G = np.around(G, decimals=3)

print("A matriz G é:\n", G)

# (c)
print("\n(c)")
C = np.array([[4., 1., -1., 0.], [1., 3., -1., 0.], [-1., -1., 5., 2.], [0., 0., 2.,␣
 ↪4.]])
G = np.zeros((len(C),len(C)))

for k in range(len(C)):
    for i in range(k):
        s1 = 0
        for j in range(i):
            s1 += G[i,j]*G[k,j]
        G[k,i] = (C[k,i] - s1)/G[i,i]
    s2 = 0
    for j in range(k):
        s2 += (G[k,j])**2
    G[k,k] = np.sqrt(C[k,k] - s2)

G = np.around(G, decimals=3)

print("A matriz G é:\n", G)
```

```
(a)
A matriz G é:
 [[ 1.414  0.     0.   ]
 [-0.707  1.225  0.   ]
 [ 0.    -0.816  1.155]]

(b)
```

(continues on next page)

(continued from previous page)

```
A matriz G é:
 [[ 2.     0.     0.     0.   ]
 [ 0.5    1.658  0.     0.   ]
 [ 0.5   -0.754  1.087  0.   ]
 [ 0.5    0.452  0.084  1.24 ]]

(c)
A matriz G é:
 [[ 2.     0.     0.     0.   ]
 [ 0.5    1.658  0.     0.   ]
 [-0.5   -0.452  2.132  0.   ]
 [ 0.     0.     0.938  1.766]]
```

46.3.3 Questão 14

Resolva os seguintes sistemas de equações por decomposição de Cholesky.

(a) \begin{cases} 8x_1 + 20x_2 + 15x_3 = 50 \ 20x_1 + 80x_2 + 50x_3 = 250 \ 15x_1 + 50x_2 + 60x_3 = 100 \end{cases}

(b) \begin{cases} 6x_1 + 15x_2 + 55x_3 = 152.6 \ 15x_1 + 55x_2 + 225x_3 = 585.6 \ 55x_1 + 225x_2 + 979x_3 = 2488.8 \end{cases}

© \begin{cases} 2x_1 − x_2 = 3 \ −x_1 + 2x_2 − x_3 = − 3 \ −x_2 + 2x_3 = 1 \end{cases}

(d) \begin{cases} 4x_1 + x_2 + x_3 + x_4 = 0.65 \ x_1 + 3x_2 − x_3 + x_4 = 0.05 \ x_1 − x_2 + 2x_3 = 0 \ x_1 + x_2 + 2x_4 = 0.5 \end{cases}

```python
# Solução

# (a)
print("(a)")
AB = np.array([[8., 20., 15., 50.], [20., 80., 50., 250.], [15., 50., 60., 100.]])
G, X = cholesky(AB)
print("A solução do sistema é: \n", G)

# (b)
print("\n(b)")
AB = np.array([[6., 15., 55., 152.6], [15., 55., 225., 585.6], [55., 225., 979., 2488.
 ↪8]])
G, X = cholesky(AB)
print("A solução do sistema é: \n", G)

# (c)
print("\n(c)")
AB = np.array([[2., -1., 0., 3.], [-1., 2., -1., -3.], [0., -1., 2., 1.]])
G, X = cholesky(AB)
print("A solução do sistema é: \n", G)

# (d)
print("\n(d)")
AB = np.array([[4., 1., 1., 1., 0.65], [1., 3., -1., 1., 0.05], [1., -1., 2., 0., 0.],
 ↪ [1., 1., 0., 2., 0.5]])
G, X = cholesky(AB)
print("A solução do sistema é: \n", G)
```

```
(a)
A solução do sistema é:
 [[2.828 0.    0.   ]
 [7.071 5.477 0.   ]
 [5.303 2.282 5.164]]

(b)
A solução do sistema é:
 [[ 2.449  0.     0.   ]
 [ 6.124  4.183  0.   ]
 [22.454 20.917  6.11 ]]

(c)
A solução do sistema é:
 [[ 1.414  0.     0.   ]
 [-0.707  1.225  0.   ]
 [ 0.    -0.816  1.155]]

(d)
A solução do sistema é:
 [[ 2.     0.     0.     0.   ]
 [ 0.5    1.658  0.     0.   ]
 [ 0.5   -0.754  1.087  0.   ]
 [ 0.5    0.452  0.084  1.24 ]]
```

CHAPTER

FORTYSEVEN

LISTA DE EXERCÍCIOS 4

Solucionário matemático e computacional de exercícios selecionados da Lista de Exercícios 4.

```
%matplotlib inline
```

```
import numpy as np
import sympy as sy
import matplotlib.pyplot as plt
```

```
# Funções Implementadas

# Solução de Sistemas

def gaussjacobi(AB,ER,X0):
    '''Realiza o cálculo de um sistema linear através do método de iterativo de Gauss-
 ↪Jacobi.

    Sinopse:
         X = gaussjacobi(AB,ER,X0)

    Entradas:
         AB - Matriz aumentada (np.array) do sistema linear
         ER - Erro relativo (forma decimal) entre a iteração i e a iteração i-1
         X0 - Vetor estimativa inicial da solução

    Saídas:
         X - Vetor solução do sistema linear

         @ney
    '''

    A = AB[:,0:-1]
    B = AB[:,-1]
    X = X0
    erro = 1

    while (erro > ER):
        Xp = np.copy(X)
        for i in range(len(A)):
            s = 0
            for j in range(len(A)):
                if (i != j):
                    s += A[i,j]*Xp[j]
            X[i] = (B[i] - s)/A[i,i]
```

(continues on next page)

(continued from previous page)

```
        EA = np.absolute(X - Xp)
        Emax = np.amax(EA)
        idx = np.where(EA == Emax)

        if (len(idx[0]) > 1):
            erro = np.absolute(EA[idx[0][0]]/X[idx[0][0]])
        else:
            erro = np.absolute(EA[idx[0]]/X[idx[0]])

    X = np.around(X, decimals=3)

    return X

def gaussseidel(AB,ER,X0):
    ''' Função que realiza o cálculo de um sistema linear através do método de
 ↪iterativo de Gauss-Seidel.

    Sinopse:
         X = gaussseidel(AB,ER,X0)

    Entradas:
         AB - Matriz aumentada (np.array) do sistema linear
         ER - Erro relativo (forma decimal) entre a iteração i e a iteração i-1
         X0 - Vetor estimativa inicial da solução

    Saídas:
         X - Vetor solução do sistema linear

         @ney
    '''

    A = AB[:,0:-1]
    B = AB[:,-1]
    X = X0
    erro = 1

    while (erro > ER):
        Xp = np.copy(X)
        for i in range(len(A)):
            s = 0
            for j in range(len(A)):
                if (i != j):
                    s += A[i,j]*X[j]
            X[i] = (B[i] - s)/A[i,i]
        EA = np.absolute(X - Xp)
        Emax = np.amax(EA)
        idx = np.where(EA == Emax)

        if (len(idx[0]) > 1):
            erro = np.absolute(EA[idx[0][0]]/X[idx[0][0]])
        else:
            erro = np.absolute(EA[idx[0]]/X[idx[0]])

    X = np.around(X, decimals=3)

    return X
```

(continues on next page)

(continued from previous page)

```
def jacobian(F,Xs):
    '''Função que realiza o cálculo da matriz jacobiana.

    Sinopse:
         J = jacobian(F,Xs)

    Entradas:
         F - Vetor contendo as equações (simbólicas) do sistema não-linear
         Xs - Vetor contendo as variáveis (simbólicas) do sistema não-linear

    Saídas:
         J - Matriz jacobiana (simbólica)

         @ney
    '''
    J = sy.zeros(len(Xs), len(Xs))

    for i in range(len(Xs)):
        for j in range(len(Xs)):
            J[i,j] = sy.diff(F[i],Xs[j])

    return J

def newtonnaolin(F,Xs,ER,X0):
    '''Função que realiza o cálculo de um sistema não-linear através do método de
 ↪iterativo de Newton.

    Sinopse:
         X = newtonnaolin(F,Xs,ER,X0)

    Entradas:
         F  - Vetor (sy.Matrix) contendo as equações (simbólicas) do sistema não-
 ↪linear
         Xs - Vetor (sy.Matrix) contendo as variáveis (simbólico) do sistema não
 ↪linear
         ER - Erro relativo (forma decimal) entre a iteração i e a iteração i-1
         X0 - Vetor estimativa inicial da solução

    Saídas:
         X - Vetor solução do sistema linear

         @ney
    '''

    J = jacobian(F,Xs)
    erro = 1
    X = np.copy(X0)

    while (erro > ER):
        Xp = np.copy(X)
        A = J[:,:]
        B = F[:,:]
        subs = []

        for i in range(len(Xs)):
            subs.append((Xs[i],X[i]))
```

(continues on next page)

(continued from previous page)

```
        A = np.asarray(A.subs(subs), dtype=float)
        B = np.asarray(B.subs(subs), dtype=float)
        B *= -1

        S = np.linalg.solve(A,B)
        S = np.transpose(S)

        X += S[0]

        EA = np.absolute(X - Xp)
        Emax = np.amax(EA)
        idx = np.where(EA == Emax)

        if (len(idx[0]) > 1):
            erro = np.absolute(EA[idx[0][0]]/X[idx[0][0]])
        else:
            erro = np.absolute(EA[idx[0]]/X[idx[0]])

    X = np.around(X, decimals=3)

    return X

# Interpolação

def int_newton(X, Y, x0):
    '''Função que realiza a interpolação de grau n de n+1 pontos dados, utilizando o
 ↪método das diferenças divididas
    de Newton.

    Sinopse:
         y0 = int_newton(X, Y, x0)

    Entradas:
         X - Array com as coordenadas em x para os pontos dados
         Y - Array com as coordenadas em y para os pontos dados
         x0 - Ponto ao qual se deseja a estimativa f(x0)

    Saídas:
         y0 - Estimativa desejada f(x0)

         @ney
    '''

    n = len(X)
    x = sy.symbols("x")

    dx = []
    f = []
    aux = np.diff(Y)

    for i in range(1,n):
        dx.append(X[i:] - X[:-i])
        f.append(aux/dx[i-1]) # Diferenças Divididas
        aux = np.diff(f[i-1])

    b = []
```

(continues on next page)

(continued from previous page)

```
    for i in f:
        b.append(i[0]) # Coeficientes do polinômio

    f = Y[0]
    aux = 1

    for i in range(n-1):
        aux *= x - X[i]
        f += b[i]*aux # Polinômio

    y0 = f.subs(x,x0) # Estimativa desejada

    return float(y0)

def int_lagrange(X, Y, x0):
    '''Função que realiza a interpolação de grau n de n+1 pontos dados, utilizando o
 ↪método de Lagrange.

    Sinopse:
         y0 = int_lagrange(X, Y, x0)

    Entradas:
         X - Vetor com as coordenadas em x para pontos dados
         Y - Vetor com as coordenadas em y para pontos dados
         x0 - Ponto ao qual se deseja a estimativa f(x0)

    Saídas:
         y0 - Estimativa desejada f(x0)

         @ney
    '''

    n = len(X)
    x = sy.symbols("x")
    f = 0

    for i in range(n):
        L = 1
        for j in range(n):
            if (i != j):
                L *= (x - X[j])/(X[i] - X[j])
        f += L*Y[i] # Polinômio

    y0 = f.subs(x,x0) # Estimativa desejada

    return float(y0)
```

47.1 Métodos iterativos para sistemas lineares

47.1.1 Questão 1

Verifique o condicionamento e o critério das linhas para os sistemas abaixo e em seguida aplique o método de Gauss-Jacobi para determinar uma solução aproximada, com erro absoluto inferior a 10^{-2}, tomando a aproximação inicial $x^{(0)} = 0$.

(a) \begin{cases} 2x_1 − x_2 = 1 \ x_1 + 2x_2 = 3 \end{cases}

(b) \begin{cases} x_1 − 0.25x_2 − 0.25x_3 = 0 \ −0.25x_1 + x_2 − 0.25x_4 = 0 \ −0.25x_1 + x_3 − 0.25x_4 = 0.25 \ −0.25x_2 + x_4 = 0.25 \end{cases}

Obs. Tome como número de condicionamento o valor $C = det()$, em que e obtida de A fazendo com que o maior elemento em valor absoluto em cada linha de seja igual a 1.

```
# Solução

# (a)
A = np.array([[2., -1.], [1., 2.]]) # Matriz de coeficientes constantes
B = np.array([[1.], [3.]]) # Vetor de constantes
X = np.array([[0.], [0.]]) # Estimativa inicial (X0)
EA = 10**(-2)
erro = 1

while (erro > EA):
    Xp = np.copy(X)
    for i in range(len(A)):
        s = 0
        for j in range(len(A)):
            if (i != j):
                s += A[i,j]*Xp[j]
        X[i] = (B[i] - s)/A[i,i]
    erro = np.amax(np.absolute(X - Xp))

X = np.around(X, decimals=3)

print("(a)\nO número C de condicionamento é:", np.around(np.linalg.det(A),␣
↪decimals=3))
print("\nA solução do problema é:\n", X)

# (b)
A = np.array([[1., -0.25, -0.25, 0.], [-0.25, 1., 0., -0.25], [-0.25, 0., 1., -0.25],␣
↪[0., -0.25, 0., 1.]]) # Coeficientes
B = np.array([[0.], [0.], [0.25], [0.25]]) # Vetor de constantes
X = np.array([[0.], [0.], [0.], [0.]]) # Estimativa inicial (X0)
EA = 10**(-2)
erro = 1

while (erro > EA):
    Xp = np.copy(X)
    for i in range(len(A)):
        s = 0
        for j in range(len(A)):
            if (i != j):
                s += A[i,j]*Xp[j]
        X[i] = (B[i] - s)/A[i,i]
    erro = np.amax(np.absolute(X - Xp))
```

(continues on next page)

(continued from previous page)

```
X = np.around(X, decimals=3)

print("\n(b)\nO número C de condicionamento é:", np.around(np.linalg.det(A),␣
↪decimals=3))
print("\nA solução do problema é:\n", X)
```

```
(a)
O número C de condicionamento é: 5.0

A solução do problema é:
 [[0.998]
 [1.002]]

(b)
O número C de condicionamento é: 0.812

A solução do problema é:
 [[0.107]
 [0.093]
 [0.343]
 [0.272]]
```

47.1.2 Questão 2

Considere o sistema

\begin{cases} 15c_1 − 2c_2 − c_3 = 3800 \ −3c_1 + 18c_2 − 6c_3 = 1200 \ −4c_1 − c_2 + 12c_3 = 2350 \end{cases}

(a) Verifique o condicionamento do sistema.

(b) Verifique o critério das linhas.

© Verifique o critério de Sassenfeld.

(d) Determine uma solução aproximada do sistema com erro relativo percentual abaixo de 5% usando o método iterativo de Gauss-Jacobi.

(e) Repita o item (d) usando o método de Gauss-Seidel.

(f) Compare o número de iterações nos itens (d) e (e).

```
# Solução

AB = np.array([[15., -2., -1., 3800.], [-3., 18., -6., 1200.], [-4., -1., 12., 2350.
↪]])
ER = 5/100
X0 = np.array([[0.], [0.], [0.]])

# (d) Gauss-Jacobi
A = AB[:,0:-1]
B = AB[:,-1]
X = np.copy(X0)
erro = 1
k1 = 0

while (erro > ER):
```

(continues on next page)

(continued from previous page)

```
    Xp = np.copy(X)
    for i in range(len(A)):
        s = 0
        for j in range(len(A)):
            if (i != j):
                s += A[i,j]*Xp[j]
        X[i] = (B[i] - s)/A[i,i]
    EA = np.absolute(X - Xp)
    Emax = np.amax(EA)
    idx = np.where(EA == Emax)
    erro = np.absolute(EA[idx[0]]/X[idx[0]])
    k1 += 1

X = np.around(X, decimals=3)

print("(d) A solução de X é:\n", X)

# (e) Gauss-Seidel
A = AB[:,0:-1]
B = AB[:,-1]
X = np.copy(X0)
erro = 1
k2 = 0

while (erro > ER):
    Xp = np.copy(X)
    for i in range(len(A)):
        s = 0
        for j in range(len(A)):
            if (i != j):
                s += A[i,j]*X[j]
        X[i] = (B[i] - s)/A[i,i]
    EA = np.absolute(X - Xp)
    Emax = np.amax(EA)
    idx = np.where(EA == Emax)
    erro = np.absolute(EA[idx[0]]/X[idx[0]])
    k2 += 1

X = np.around(X, decimals=3)

print("\n(e) A solução de X é:\n", X)

# (f)
print("\n(f) O número de iterações pelo método de Gauss-Jacobi foi:", k1)
print("O número de iterações pelo método de Gauss-Seidel foi:", k2)
```

```
(d) A solução de X é:
 [[301.184]
 [216.637]
 [311.689]]

(e) A solução de X é:
 [[302.072]
 [220.055]
 [314.862]]
```

(continues on next page)

(continued from previous page)

```
(f) O número de iterações pelo método de Gauss-Jacobi foi: 4
O número de iterações pelo método de Gauss-Seidel foi: 3
```

47.1.3 Questão 3

Use o método iterativo de Gauss-Seidel para determinar uma solução aproximada do sistema com erro relativo percentual abaixo de 5% dos seguintes sistemas. Antes verifique o critério de Sassenfeld e caso algum dos sitemas não o satisfaça, reorganize o sistema permutando linhas e/ou colunas para garantir a convergência.

(a) \begin{cases} 10x_1 + 2x_2 − x_3 = 27 \ −3x_1 − 6x_2 + 2x_3 = −61.5 \ x_1 + x_2 + 5x_3 = −21.5 \end{cases}

(b) \begin{cases} −3x_1 + x_2 + 12x_3 = 50 \ 6x_1 − x_2 − x_3 = 3 \ 6x_1 + 9x_2 + x_3 = 40 \end{cases}

© \begin{cases} 2x_1 − 6x_2 − x_3 = −38 \ −3x_1 − x_2 + 7x_3 = −34 \ −8x_1 + x_2 − 2x_3 = −20 \end{cases}

```
# Solução

# (a)
AB = np.array([[10., 2., -1., 27.], [-3., -6., 2., -61.5], [1., 1., 5., -21.5]])
ER = 5/100
X0 = np.array([0., 0., 0.])
X = gaussseidel(AB,ER,X0)
print("(a)\nA solução do sistema linear é: \n", X)

# (b)
AB = np.array([[6., -1., -1., 3.], [6., 9., 1., 40.], [-3., 1., 12., 50.]])
ER = 5/100
X0 = np.array([0., 0., 0.])
X = gaussseidel(AB,ER,X0)
print("\n(b)\nA solução do sistema linear é: \n", X)

# (c)
AB = np.array([[-8., 1., -2., -20.], [2., -6., -1., -38.], [-3., -1., 7., -34.]])
ER = 5/100
X0 = np.array([0., 0., 0.])
X = gaussseidel(AB,ER,X0)
print("\n(c)\nA solução do sistema linear é: \n", X)
```

```
(a)
A solução do sistema linear é:
 [ 0.5  8.  -6. ]

(b)
A solução do sistema linear é:
 [1.697 2.829 4.355]

(c)
A solução do sistema linear é:
 [ 4.005  7.992 -1.999]
```

47.1.4 Questão 4

Dos seguintes sistemas abaixo identifique qual(is) deles nao pode(m) ser resolvido(s) usando o método iterativo de Gauss-Seidel.

(a) \begin{cases} 9x + 3y + z = 13 \ −6x + 8z = 2 \ 2x + 5y − z = 6 \end{cases}

(b) \begin{cases} x + y + 6z = 8 \ x + 5y − z = 5 \ 4x + 2y − 2z = 4 \end{cases}

© \begin{cases} −3x + 4y + 5z = 6 \ −2x + 2y − 3z = −3 \ 2y − z = 1 \end{cases}

Solução

(a) Convergente

(b) Divergente

© Divergente

47.1.5 Questão 5

Uma companhia de eletrônica produz transistores, resistores e chips de computador. Cada transistor usa quatro unidades de cobre, uma unidade de zinco e duas unidades de vidro. Cada resisitor usa três, três e uma unidades de cada material, respectivamente, e cada chip de computador usa duas, uma e três unidades desses materiais, respectivamente. Colocando essas informações em uma tabela, tem-se:

Componente	Cobre	Zinco	Vidro
Transistor	4	1	2
Resistor	3	3	1
Chip de Computador	2	1	3

O fornecimento desses materiais varia de semana para semana. Assim, a companhia precisa determinar uma meta de produção diferente para cada semana. Por exemplo, em uma semana a quantidade total de materiais disponíveis era 960 unidades de cobre, 510 unidades de zinco e 610 unidades de vidro. Determine o sistema de equações que modela essa meta de produção e determine a sua solução, pelo método de Gauss-Seidel, para determinar o número de transistores, resistores e chips de computador fabricados nessa semana.

```
# Solução

AB = np.array([[4., 3., 2., 960.], [1., 3., 1., 510.], [2., 1., 3., 610.]])
ER = 1/100
X0 = np.array([[0.], [0.], [0.]])

X = gaussseidel(AB,ER,X0)
X = np.transpose(np.rint(X))
print("[T,R,C] =", X[0])
```

```
[T,R,C] = [120. 100.  90.]
```

47.2 Sistemas não-lineares

47.2.1 Questão 6

Usando o Método de Newton, determinar uma raiz para cada sistema não-linear abaixo com precisão $\epsilon = 10^{-3}$:

(i) $\$ \begin{cases} x^2 + y^2 = 2 \\ x^2 - y^2 = 1 \end{cases} \qquad x^{(0)} = \begin{bmatrix} 1.2 \\ 0.7 \end{bmatrix} \$$

(ii) $\$ \begin{cases} 3x^2y - y^3 = 4 \\ x^2 - xy^3 = 9 \end{cases} \qquad x^{(0)} = \begin{bmatrix} -1 \\ -2 \end{bmatrix} \$$

(i) $\$ \begin{cases} (x-1)^2 + y^2 = 4 \\ x^2 + (y-1)^2 = 4 \end{cases} \qquad x^{(0)} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \$$

```
# Solução

x, y = sy.symbols("x, y")
Xs = sy.Matrix([x, y])
ER = 1/100

# (i)
F = sy.Matrix([x**2 + y**2 - 2, x**2 - y**2 - 1])
X0 = np.array([1.2, 0.7])
X = newtonnaolin(F,Xs,ER,X0)
print("(i) A solução do problema é:", X)

# (ii)
F = sy.Matrix([3*(x**2)*y - y**3 - 4, x**2 - x*y**3 - 9])
X0 = np.array([-1., -2.])
X = newtonnaolin(F,Xs,ER,X0)
print("\n(ii) A solução do problema é:", X)

# (iii)
F = sy.Matrix([(x-1)**2 + y**2 - 4, x**2 + (y-1)**2 - 4])
X0 = np.array([2., 1.])
X = newtonnaolin(F,Xs,ER,X0)
print("\n(iii) A solução do problema é:", X)
```

```
(i) A solução do problema é: [1.225 0.707]

(ii) A solução do problema é: [3.002 0.148]

(iii) A solução do problema é: [1.823 1.823]
```

47.3 Interpolação Polinomial

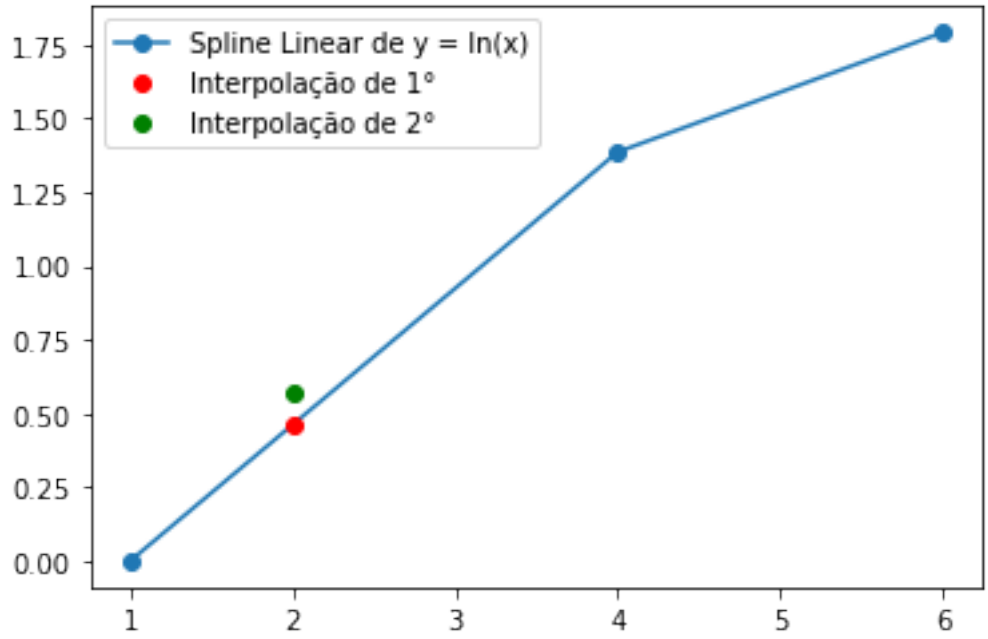
47.3.1 Questão 7

Use um polinômio interpolador de Lagrange de primeiro e segundo graus para calcular $ln(2)$, com base nos seguintes dados:

x_i	$f(x_i)$
1	0
4	1.386294
6	1.791759

```
# Solução

X = [1, 4, 6]
Y = [0, 1.386294, 1.791759]

# (i) Polinômio de Primeiro Grau
y1 = int_lagrange(X[0:2],Y[0:2],2)
print("(i)\nA estimativa de ln(2) para o polinômio de primeiro grau é:", y1)

# (i) Polinômio de Segundo Grau
y2 = int_lagrange(X,Y,2)
print("\n(ii)\nA estimativa de ln(2) para o polinômio de segundo grau é:", y2)

plt.plot(X,Y, 'o-')
plt.plot(2,y1, 'or', 2,y2, 'og')
plt.legend(['Spline Linear de y = ln(x)', 'Interpolação de 1°', 'Interpolação de 2°
↪']);
```

```
(i)
A estimativa de ln(2) para o polinômio de primeiro grau é: 0.46209799999999995

(ii)
A estimativa de ln(2) para o polinômio de segundo grau é: 0.5658441999999999
```



47.3.2 Questão 8

Faça uma estimativa de $log(10)$ usando interpolação linear

(i) entre $log(8) = 0.9030900$ e $log(12) = 1.0791812$

(ii) entre $log(9) = 0.9542425$ e $log(11) = 1.0413927$

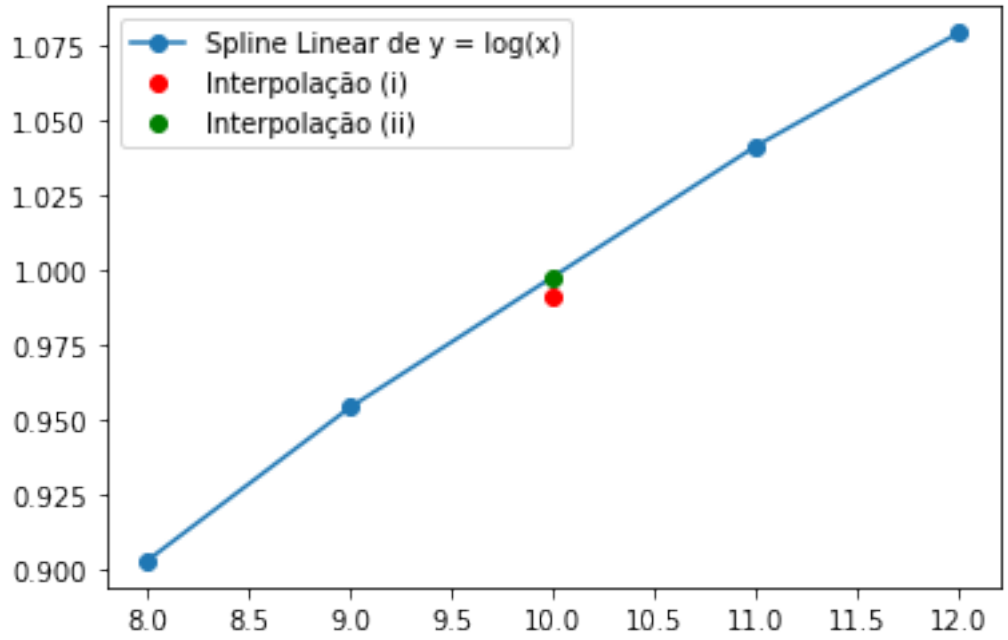
```
# Solução

# (i)
X = [8, 12]
Y = [0.9030900, 1.0791812]

y1 = int_lagrange(X,Y,10)
print("(i)\nA estimativa de ln(10) é:", y1)

# (i)
X = [9, 11]
Y = [0.9542425, 1.0413927]

y2 = int_lagrange(X,Y,10)
print("\n(ii)\nA estimativa de ln(10) é:", y2)

X = [8, 9, 11, 12]
Y = [0.9030900, 0.9542425, 1.0413927, 1.0791812]

plt.plot(X,Y, 'o-')
plt.plot(10,y1, 'or', 10,y2, 'og')
plt.legend(['Spline Linear de y = log(x)', 'Interpolação (i)', 'Interpolação (ii)']);
```

```
(i)
A estimativa de ln(10) é: 0.9911356000000002
```

(continues on next page)

(continued from previous page)

```
(ii)
A estimativa de ln(10) é: 0.9978176
```



47.3.3 Questão 9

Considere os dados e faça o que se pede:

x	y
1	3
2	6
3	19
5	99
7	291
8	444

(i) Calcule f(4) usando polinomios interpoladores de Lagrange de primeiro a terceiro graus.

(ii) Calcule f(4) usando polinomios interpoladores de Newton de primeiro a quarto graus.

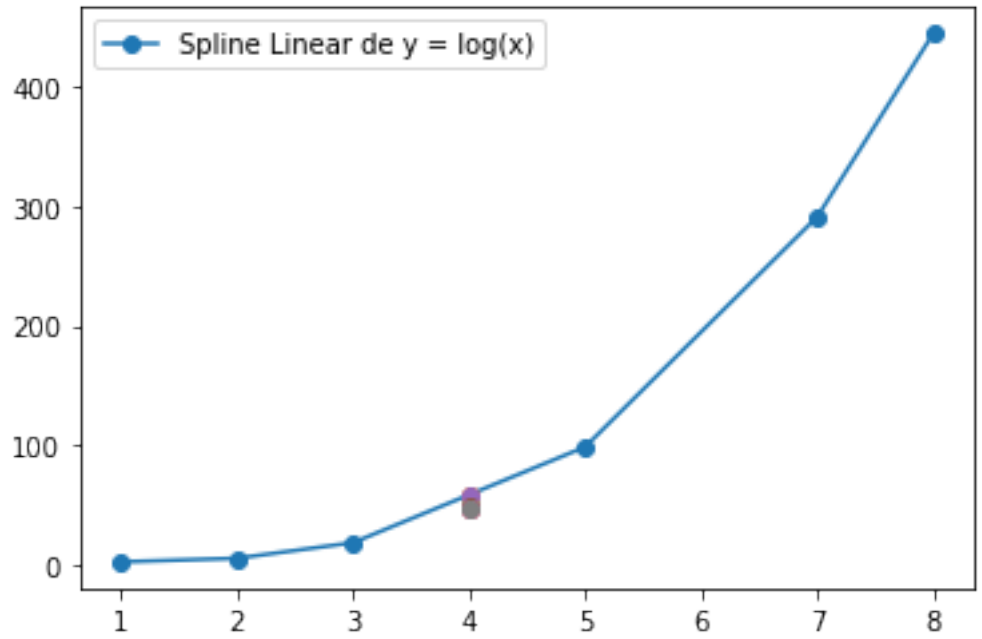
```
# Solução

X = np.array([1, 2, 3, 5, 7, 8])
Y = np.array([3, 6, 19, 99, 291, 444])

# (i)
y1 = int_lagrange(X[2:4],Y[2:4],4)
print("(i)\nA estimativa de f(4) de primeiro grau é:", y1)
y2 = int_lagrange(X[1:4],Y[1:4],4)
print("A estimativa de f(4) de segundo grau é:", y2)
```

(continues on next page)

(continued from previous page)

```
y3 = int_lagrange(X[:4],Y[:4],4)
print("A estimativa de f(4) de terceiro grau é:", y3)

# (i)
y4 = int_newton(X[2:4],Y[2:4],4)
print("\n(ii)\nA estimativa de f(4) de primeiro grau é:", y4)
y5 = int_newton(X[1:4],Y[1:4],4)
print("A estimativa de f(4) de segundo grau é:", y5)
y6 = int_newton(X[:4],Y[:4],4)
print("A estimativa de f(4) de terceiro grau é:", y6)
y7 = int_newton(X[:-1],Y[:-1],4)
print("A estimativa de f(4) de quarto grau é:", y7)

plt.plot(X,Y, 'o-')
plt.plot(4,y1, 'o', 4,y2, 'o', 4,y3, 'o', 4,y4, 'o', 4,y5, 'o', 4,y6, 'o', 4,y7, 'o')
plt.legend(['Spline Linear de y = log(x)']);
```

```
(i)
A estimativa de f(4) de primeiro grau é: 59.0
A estimativa de f(4) de segundo grau é: 50.0
A estimativa de f(4) de terceiro grau é: 48.0

(ii)
A estimativa de f(4) de primeiro grau é: 59.0
A estimativa de f(4) de segundo grau é: 50.0
A estimativa de f(4) de terceiro grau é: 48.0
A estimativa de f(4) de quarto grau é: 48.0
```



47.3.4 Questão 10

Conhecendo a seguinte tabela

x	f(x)
-1	15
0	8
3	-1

determine:

(i) O polinômio de interpolação para a função definida por este conjunto de pares de pontos. Ou seja, determine a solução do sistema

$$Xa = f$$

para o vetor de a dos coeficientes de $P_2(x)$.

(ii) O polinomio de interpolação na forma de Newton.

```
# Solução

X = np.array([-1, 0, 3])
Y = np.array([15, 8, -1])

n = len(X)
x = sy.symbols("x")

dx = []
f = []
aux = np.diff(Y)

for i in range(1,n):
    dx.append(X[i:] - X[:-i])
    f.append(aux/dx[i-1])
    aux = np.diff(f[i-1])

b = []

for i in f:
    b.append(i[0])

f = Y[0]
aux = 1

for i in range(n-1):
    aux *= x - X[i]
    f += b[i]*aux

print("O polinômio é: P(x) =", sy.simplify(f))
```

```
O polinômio é: P(x) = 1.0*x**2 - 6.0*x + 8.0
```

47.3.5 Questão 11

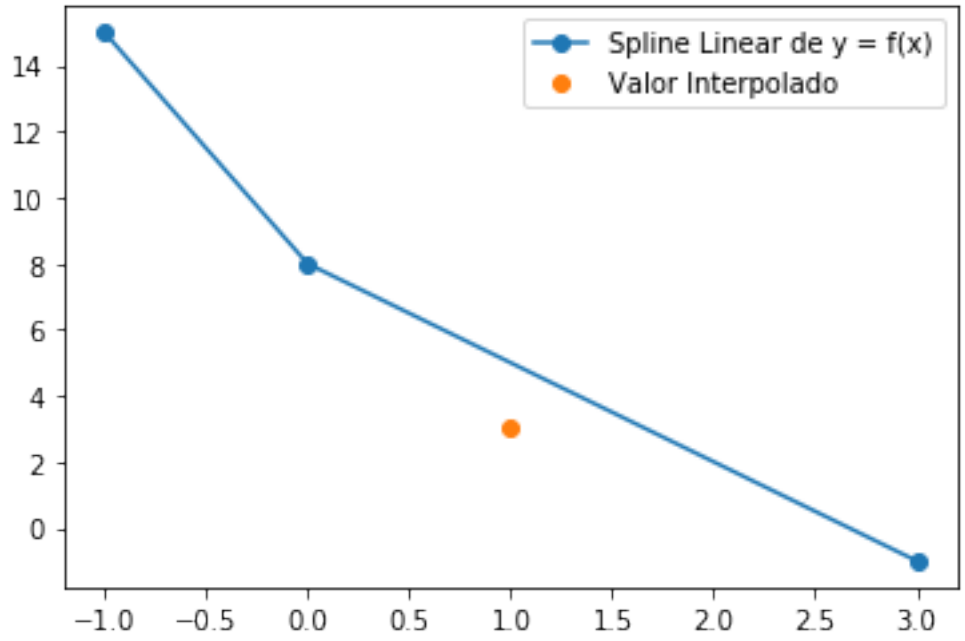
Usando os dados da tabela do exercício anterior:

(i) Determine $P_2(x)$ pela forma de Lagrange.

(ii) Calcule uma aproximação para $f(1)$ usando o item (i).

```
# Solução

X = [-1, 0, 3]
Y = [15, 8, -1]

# (i)
n = len(X)
x = sy.symbols("x")
f = 0

for i in range(n):
    L = 1
    for j in range(n):
        if (i != j):
            L *= (x - X[j])/(X[i] - X[j])
    f += L*Y[i]

print("(i)\nP(x) =", sy.simplify(f))

# (ii)
y = f.subs(x,1)

print("\n(ii)\nA aproximação para f(1) é:", float(y))

plt.plot(X,Y, 'o-')
plt.plot(1,y, 'o')
plt.legend(['Spline Linear de y = f(x)', 'Valor Interpolado']);
```

```
(i)
P(x) = x**2 - 6*x + 8

(ii)
A aproximação para f(1) é: 3.0
```



47.3.6 Questão 12

Dada a tabela:

x	xe^{3x}
0	1
0.1	1.3499
0.2	1.8221
0.3	2.4596
0.4	3.3201
0.5	4.4817

calcule $f(0.25)$, onde $f(x) = xe^{3x}$ usando polinômio de interpolação do 2° grau:

(i) Usando $x_0 = 0.2, x_1 = 0.3, x_2 = 0.4$;

(ii) Usando $x_0 = 0.1, x_1 = 0.2, x_2 = 0.3$;

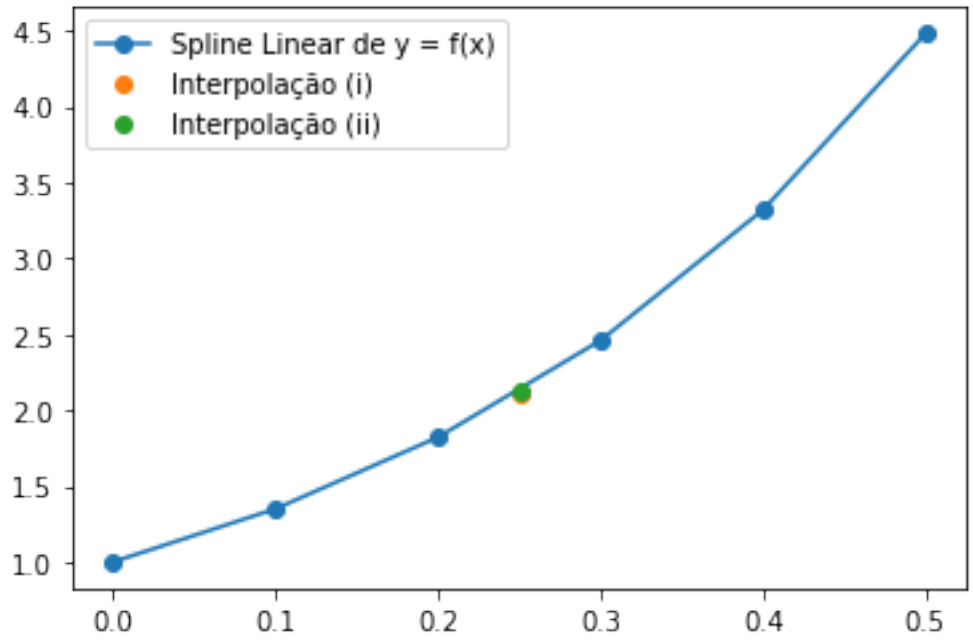
```
# Solução

# (i)
X = [0.2, 0.3, 0.4]
Y = [1.8221, 2.4596, 3.3201]

y1 = int_lagrange(X,Y,0.25)
print("(i)\nA estimativa de f(0.25) é:", y1)

# (i)
X = [0.1, 0.2, 0.3]
Y = [1.3499, 1.8221, 2.4596]

y2 = int_lagrange(X,Y,0.25)
```

(continues on next page)

(continued from previous page)

```
print("\n(ii)\nA estimativa de f(0.25) é:", y2)

X = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
Y = [1, 1.3499, 1.8221, 2.4596, 3.3201, 4.4817]

plt.plot(X,Y, 'o-')
plt.plot(0.25,y1, 'o', 0.25,y2, 'o')
plt.legend(['Spline Linear de y = f(x)', 'Interpolação (i)', 'Interpolação (ii)']);
```

```
(i)
A estimativa de f(0.25) é: 2.112975

(ii)
A estimativa de f(0.25) é: 2.1201875
```



47.3.7 Questão 13

Para a seguinte função tabelada

x	xe^{3x}
0	1
0.1	1.3499
0.2	1.8221
0.3	2.4596
0.4	3.3201
0.5	4.4817

construir a tabela de diferenças divididas.

```
# Solução

X = np.array([0, 0.1, 0.2, 0.3, 0.4, 0.5])
Y = np.array([1, 1.3499, 1.8221, 2.4596, 3.3201, 4.4817])

n = len(X)
x = sy.symbols("x")

dx = []
f = []
aux = np.diff(Y)

for i in range(1,n):
    dx.append(X[i:] - X[:-i])
    f.append(aux/dx[i-1])
    aux = np.diff(f[i-1])

for i in range(len(f)):
    print("\n As", i+1, "ªs diferenças divididas são:\n", f[i])
```

```
 As 1 ªs diferenças divididas são:
 [ 3.499  4.722  6.375  8.605 11.616]

 As 2 ªs diferenças divididas são:
 [ 6.115  8.265 11.15  15.055]

 As 3 ªs diferenças divididas são:
 [ 7.16666667  9.61666667 13.01666667]

 As 4 ªs diferenças divididas são:
 [6.125 8.5  ]

 As 5 ªs diferenças divididas são:
 [4.75]
```

CHAPTER

FORTYEIGHT

LISTA DE EXERCÍCIOS 5

Solucionário matemático e computacional de exercícios selecionados da Lista de Exercícios 5.

```
%matplotlib inline
```

```
# importação de módulos
import numpy as np
import matplotlib.pyplot as plt
```

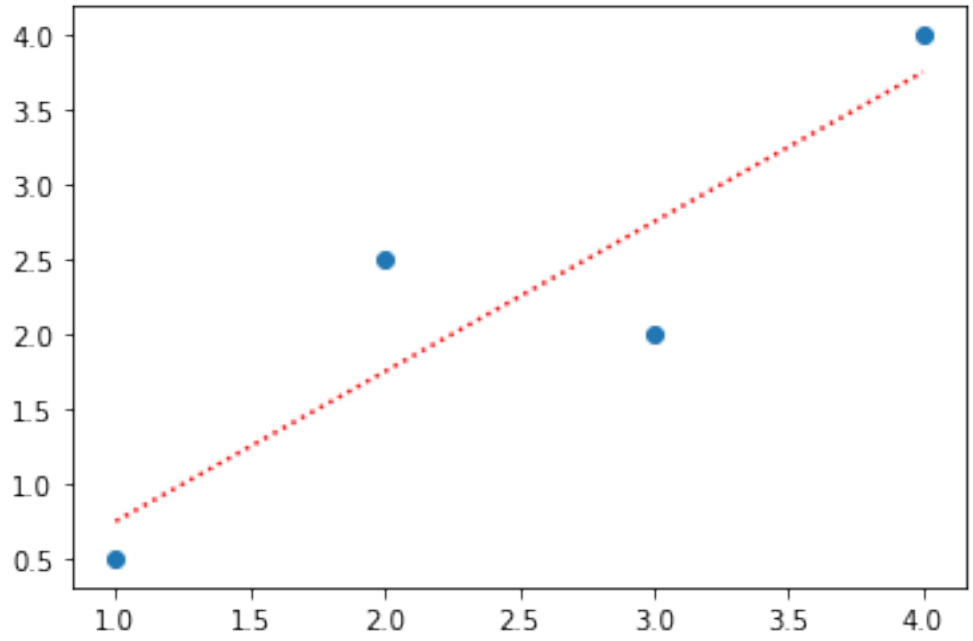
48.1 Ajuste de curvas

Obs.:: função `polyfit` retorna coeficientes da curva de ajuste na ordem:

```
P = p[0] + p[1]*x + p[2]*x**2 + p[3]*x**3 + ...
```

48.1.1 Exemplo: solução fechada 2D

```
# tabela de dados
x = np.array([1,2,3,4])
y = np.array([0.5,2.5,2.0,4.0])

# min quad
m = np.size(x)
alpha1 = (m*np.dot(x,y) - np.sum(x)*np.sum(y))/(m*np.dot(x,x)-np.sum(x)**2)
alpha0 = np.mean(y) - alpha1*np.mean(x)

# plot
plt.scatter(x,y)
y2 = alpha0 + alpha1*x
plt.plot(x,y2,'r:');
```



48.2 Resolução da Lista 5

48.2.1 Função-base para computar ajuste e plotar resultados

```
"""
Resolve problema de ajuste polinomial discreto e plota resultado

    entrada:
        x,y : tabela de valores (numpy arrays)
        g   : grau do polinômio de ajuste (int)
"""
def resolve_ajuste(x,y,g):

    c = np.polyfit(x, y, g) # ajuste
    p = np.poly1d(c)
    plt.plot(x,y,'d',x,p(x),'o-')
    plt.xlabel(str(x))
    for i in range(g+1):
        print('Coeficiente de ajuste p[' + str(i) + ']: ' + str(c[i]))

    return p
```

48.2.2 L5-Q2

Ajuste os dados abaixo pelo método dos mínimos quadrados:

\begin{array}{clccccccc} x & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline y & 0.5 & 0.6 & 0.9 & 0.8 & 1.2 & 1.5 & 1.7 & 2.0 \end{array}

a) por reta

b) por parábola do tipo $ax^2 + bx + c$;

48.2.3 Solução

Metodologia matemática

a)

$$g_1(x) = xg_2(x) = 1$$

\begin{array}{clccccccc} g_1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline g_2 & 1 & 1 & 1& 1& 1 & 1 & 1 & 1 \\ \hline f & 0.5 & 0.6 & 0.9 & 0.8 & 1.2 & 1.5 & 1.7 & 2.0 \end{array}

$$\begin{aligned} a_{11} &= < g_1, g_1 > && = 204 \\ a_{12} &= < g_1, g_2 > && = 36 \\ a_{21} &= a_{12} \\ a_{22} &= < g_2, g_2 > && = 8 \\ b_1 &= < g_1, f > && = 50.5 \\ b_2 &= < g_2, f > && = 9.2 \end{aligned}$$

$$\begin{bmatrix} 204 & 36 \\ 36 & 8 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \begin{bmatrix} 50.5 \\ 9.2 \end{bmatrix}$$

$$\begin{aligned} 204\alpha_1 + 36\alpha_2 &= 50.5 \\ 36\alpha_1 + 8\alpha_2 &= 9.2 \end{aligned}$$

$$\alpha_1 = 0.2166,\ \alpha_2 = 0.175$$

$$f(x) = 0.1749x + 0.2167$$

b)

$$g_1(x) = x^2 g_2(x) = xg_3(x) = 1$$

\begin{array}{clccccccc} g_1 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 \\ \hline g_2 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline g_3 & 1 & 1 & 1& 1& 1 & 1 & 1 & 1 \\ \hline f & 0.5 & 0.6 & 0.9 & 0.8 & 1.2 & 1.5 & 1.7 & 2.0 \end{array}

$$\begin{aligned} a_{11} &= < g_1, g_1 > &&= 8772 \\ a_{12} = a_{21} &= < g_1, g_2 > &&= 1296 \\ a_{13} = a_{31} &= < g_1, g_3 > &&= 204 \\ a_{22} &= < g_2, g_2 > &&= 204 \\ a_{23} = a_{32} &= < g_2, g_3 > &&= 36 \\ a_{33} &= < g_3, g_3 > &&= 8 \\ b_1 &= < g_1, f > &&= 319.1 \\ b_2 &= < g_2, f > &&= 50.5 \\ b_3 &= < g_3, f > &&= 9.2 \end{aligned}$$

$$\begin{bmatrix} 8872 & 1296 & 204 \\ 1296 & 204 & 36 \\ 204 & 36 & 8 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} 319.1 \\ 50.5 \\ 9.2 \end{bmatrix}$$

$$\begin{aligned} 8872\alpha_1 + 1296\alpha_2 + 204\alpha_3 &= 319.1 \\ 1296\alpha_1 + 204\alpha_2 + 36\alpha_3 &= 50.5 \\ 204\alpha_1 + 36\alpha_2 + 8\alpha_3 &= 9.2 \end{aligned}$$

$$\alpha_1 = 0.4071,\ \alpha_2 = 0.07738,\ \alpha_3 = 0.01547$$

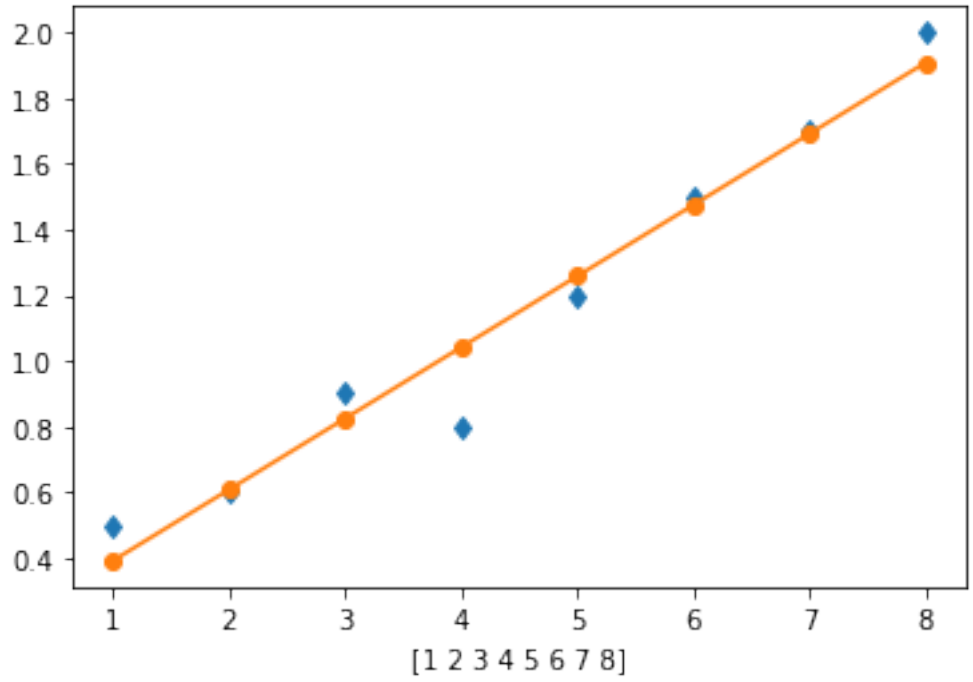
$$f(x) = 0.01547 + 0.07738x + 0.4071x^2$$

Metodologia computacional

a)

```
# tabela de dados
x = np.arange(1,9)
y = np.array([0.5,.6,.9,.8,1.2,1.5,1.7,2.])

# grau 1
g = 1
p = resolve_ajuste(x,y,g)
```

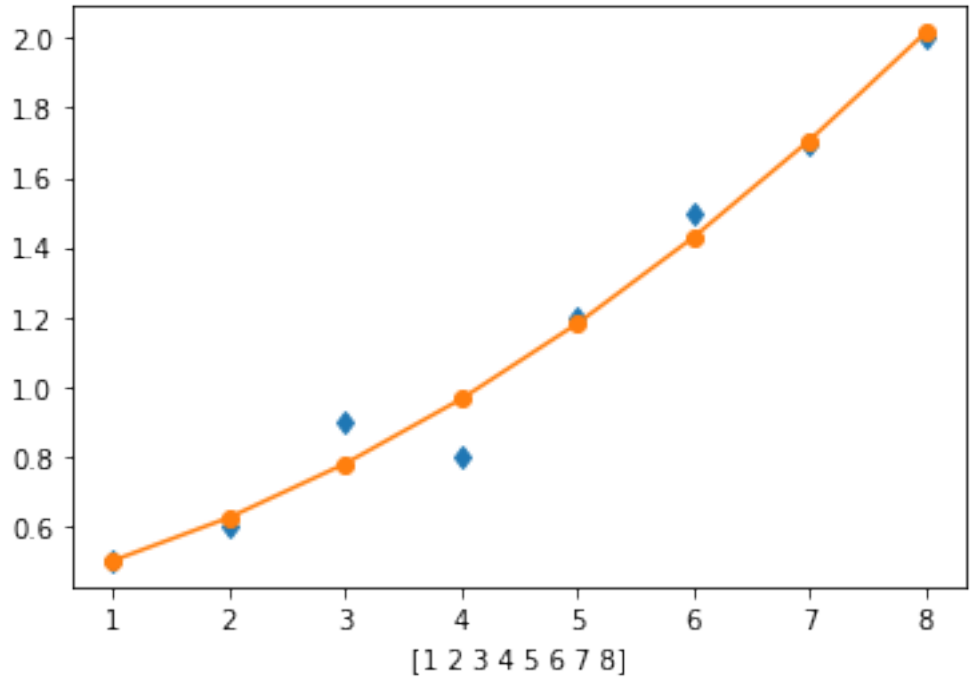
```
Coeficiente de ajuste p[0]: 0.2166666666666667
Coeficiente de ajuste p[1]: 0.17499999999999996
```



b)

```
# grau 2
g = 2
p = resolve_ajuste(x,y,g)
```

```
Coeficiente de ajuste p[0]: 0.015476190476190437
Coeficiente de ajuste p[1]: 0.07738095238095268
Coeficiente de ajuste p[2]: 0.407142857142857
```



48.2.4 L5-Q3

Dada a tabela abaixo, faça o gráfico de dispersão dos dados e ajuste uma curva da melhor maneira possível:

\begin{array}{clccccc} x & 0.5 & 0.75 & 1 & 1.5 & 2.0 & 2.5 & 3.0 \ \hline y & -2.8 & -0.6 & 1& 3.2& 4.8 & 6.0 & 7.0 \end{array}

48.2.5 Solução

Metodologia matemática

$$g_1(x) = 1 g_2(x) = x g_3(x) = x^2 g_4(x) = x^3$$

\begin{array}{clccccc} g_1 & 1 & 1 & 1& 1& 1 & 1 & 1 \ g_2 & 0.5 & 0.75 & 1 & 1.5 & 2.0 & 2.5 & 3.0 \ \hline g_3 & 0.25 & 0.5625& 1& 2.25& 4 & 6.25 & 9.0\ \hline g_4 & 0.1250 & 0.4219 & 1& 3.375& 8 &15. 625 & 27 \ f

& -2.8 & -0.6 & 1& 3.2& 4.8 & 6.0 & 7.0 \end{array}

$$
\begin{aligned}
a_{11} &= < g_1, g_1 > = 7 \\
a_{12} = a_{21} &= < g_1, g_2 > = 11.25 \\
a_{13} = a_{31} &= < g_1, g_3 > = 23.3125 \\
a_{14} = a_{41} &= < g_1, g_4 > = 55.5468 \\
a_{22} &= < g_2, g_2 > = 23.3125 \\
a_{23} = a_{32} &= < g_2, g_3 > = 55.5468 \\
a_{24} = a_{42} &= < g_2, g_4 > = 142.5039 \\
a_{33} &= < g_3, g_3 > = 142.5039 \\
a_{34} &= < g_3, g_4 > = 381.5185 \\
a_{44} &= < g_4, g_4 > = 1049.7248 \\
b_1 &= < g_1, f > = 18.6 \\
b_2 &= < g_2, f > = 49.55 \\
b_3 &= < g_3, f > = 126.86 \\
b_4 &= < g_4, f > = 332.34
\end{aligned}
$$

$$
\begin{bmatrix}
7 & 11.25 & 23.3125 & 55.5468 \\
11.25 & 23.3125 & 55.5468 & 142.5039 \\
23.3125 & 55.5468 & 142.5039 & 381.5185 \\
55.5468 & 142.5039 & 381.5185 & 1049.7248
\end{bmatrix}
\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix}
=
\begin{bmatrix} 18.6 \\ 49.55 \\ 126.8625 \\ 332.3468 \end{bmatrix}
$$

$$\alpha_1 = -8.1043,\ \alpha_2 = 12.7882\ \alpha_3 = -4.3250,\ \alpha_4 = 0.5813$$

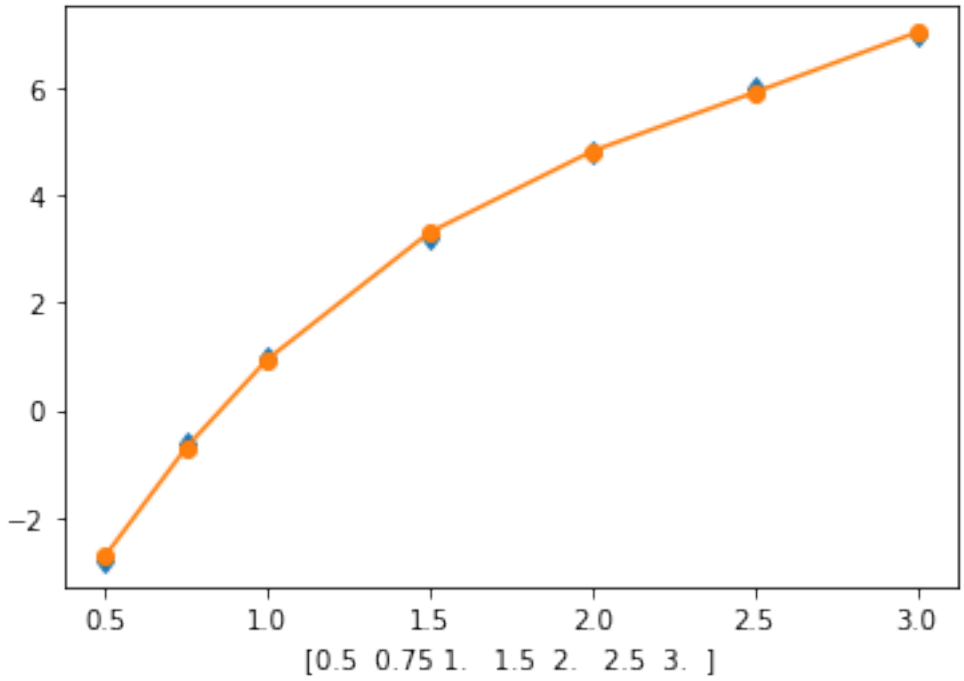
$$f(x) = 0.5813 - 4.3250x + 12.79x^2 - 8.1043x^3$$

Metodologia computacional

```
# tabela
x = np.array([0.5,0.75,1,1.5,2.0,2.5,3.0])
y = np.array([-2.8,-0.6,1,3.2,4.8,6.0,7.0])

# grau (teste para g = 2,3,4,5 e veja o erro)
g = 3
p = resolve_ajuste(x,y,g)

# erro
np.sum((p(x)-y)**2)
```

```
Coeficiente de ajuste p[0]: 0.588808229974697
Coeficiente de ajuste p[1]: -4.365386461549416
Coeficiente de ajuste p[2]: 12.85159200943416
Coeficiente de ajuste p[3]: -8.131256481519904
```

```
0.040251260748212274
```



48.2.6 L5-Q4

A tabela abaixo mostra as alturas e pesos de nove homens entre as idades de 25 a 29 anos extraída ao acaso entre funcionários de uma grande indústria:

\begin{array}{clccccccccc} \text{Altura/cm} & 183& 173 & 168 & 188 & 158 & 163 & 193 &163 &178 \ \hline \text{Peso/kg} & 79 & 69 & 70& 81 & 61 &63 &79 & 71 & 73 \end{array}

Metodologia matemática

$$g_1(x) = 1 g_2(x) = x$$

\begin{array}{clccccccccc} g_1 & 1 & 1 & 1& 1& 1 & 1 & 1&1\ \hline g_2 & 183& 173 & 168 & 188 & 158 & 163 & 193 &163 &178 \ \hline f & 79 & 69 & 70& 81 & 61 &63 &79 & 71 & 73 \end{array}

$$\begin{aligned} a_{11} &= < g_1, g_1 > &= 9 \\ a_{12} = \ a_{21} &= < g_1, g_2 > &= 1567 \\ a_{22} &= < g_2, g_2 > &= 274.021 \\ b_1 &= < g_1, f > &= 646 \\ b_2 &= < g_2, f > &= 113.103 \end{aligned}$$

$$\begin{bmatrix} 9 & 1567 \\ 1567 & 274.021 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \begin{bmatrix} 646 \\ 113.103 \end{bmatrix}$$

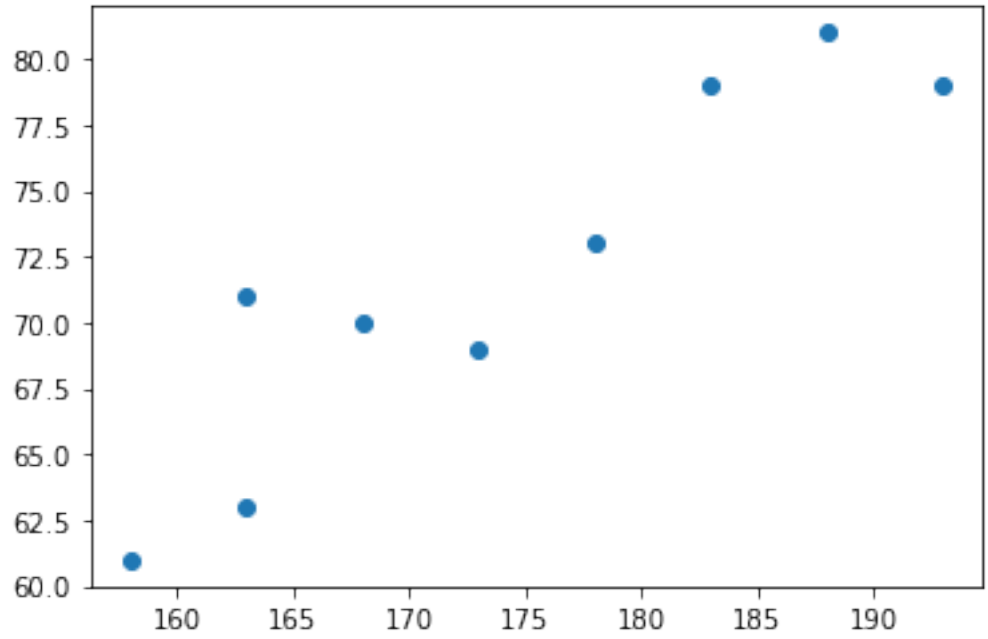
$$\alpha_1 = -20.0832, \ \alpha_2 = 0.5276$$

$$f(x) = 0.5276x - 20.0832$$

Metodologia computacional

```
altura = np.array([183,173,168,188,158,163,193,163,178])
peso = np.array([79,69,70,81,61,63,79,71,73])

# gráfico de dispersão
plt.scatter(altura,peso)
```

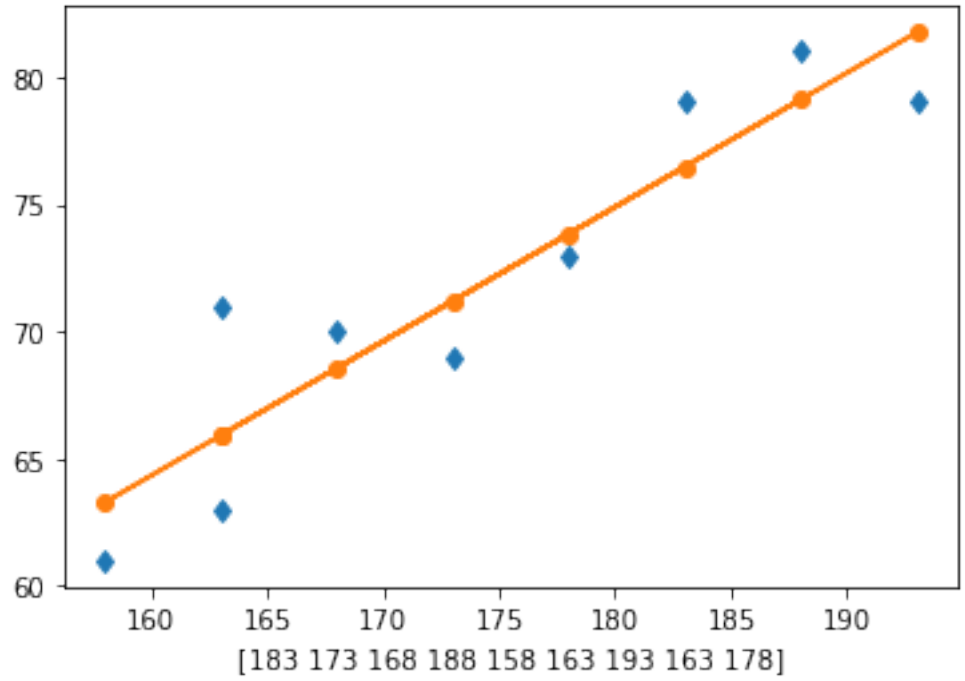
```
<matplotlib.collections.PathCollection at 0x7fc4720ed610>
```



48.2.7 solucao-L5-Q4-b

```
# ajuste da reta
p = resolve_ajuste(altura,peso,1)
```

```
Coeficiente de ajuste p[0]: 0.527570093457944
Coeficiente de ajuste p[1]: -20.078037383177612
```



48.2.8 solucao-L5-Q4-c

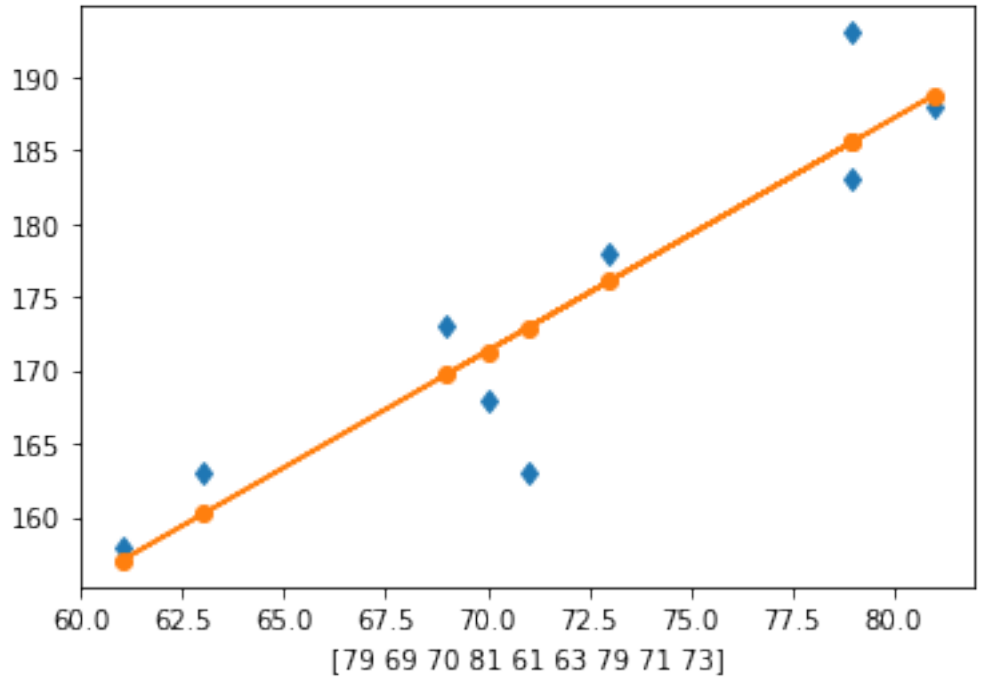
```
# estimativa de peso (kg)
alt = 175
p_c = p(alt)
print(p_c)

# estimativa de altura (cm)
alt = 80
a_c = (alt - p[0])/p[1]
print(a_c)
```

```
72.2467289719626
189.69619131975205
```

48.2.9 solucao-L5-Q4-d

```
p2 = resolve_ajuste(peso,altura,1)
```

```
Coeficiente de ajuste p[0]: 1.5856741573033717
Coeficiente de ajuste p[1]: 60.294943820224695
```



48.2.10 solucao-L5-Q4-e

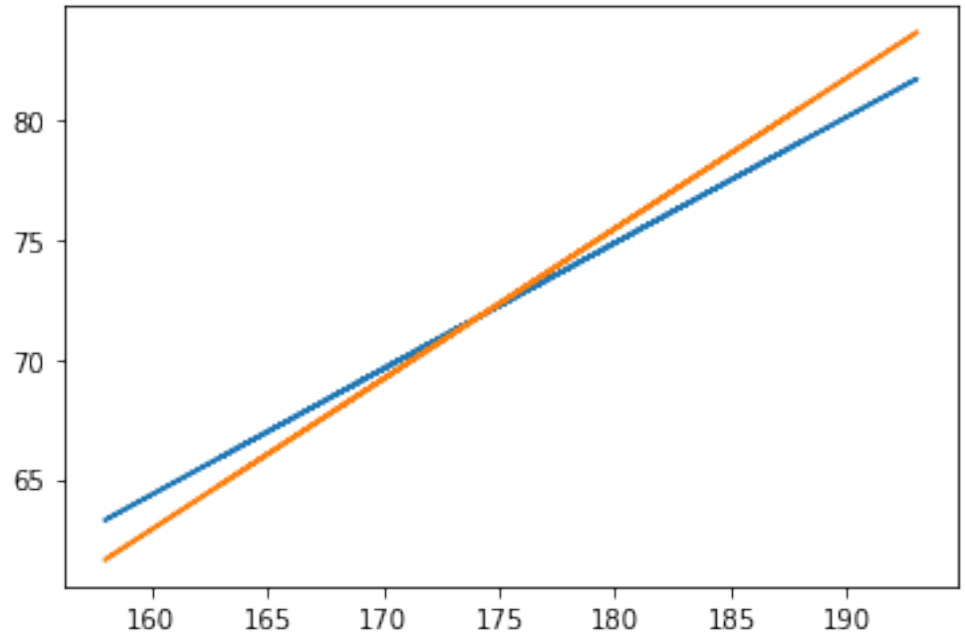
```
# estimativa de peso a partir de altura com novo ajuste
alt = 175
p_e = (alt - p2[0])/p2[1]
print(p_e)

# comparação com item (c): pesos
# pequena diferença entre os valores
dif = abs(p_c-p_e)
print(dif)
```

```
72.33835252435782
0.09162355239521958
```

48.2.11 solucao-L5-Q4-f

```
# comparação das retas de ajuste
# influência do resíduo => inclinações diferentes

plt.plot(altura,p(altura),altura,(altura - p2[0])/p2[1])
```

```
[<matplotlib.lines.Line2D at 0x7fc4724d15d0>,
 <matplotlib.lines.Line2D at 0x7fc4724d1810>]
```



48.3 solucao-L5-Q5-a

48.3.1 AJUSTE POR RETA

```
"""
Nota: pesquisa no IBGE em maio de 2018
      mostra que a população já rompeu 209 mi.
"""

# tabela de dados
ano = np.array([1900, 1920, 1940, 1950, 1960, 1970, 1980, 1991, 2000, 2010, 2015])
hab = np.array([17.4, 30.6, 41.2, 51.9, 70.2, 93.1, 119.0, 146.2, 175.8, 198.6, 207.
 ↪8])

# plotagem
fig,ax = plt.subplots(1,1)
p = resolve_ajuste(ano,hab,1)
ax.grid(axis='x')

# plotagem do comportamento preditivo
```

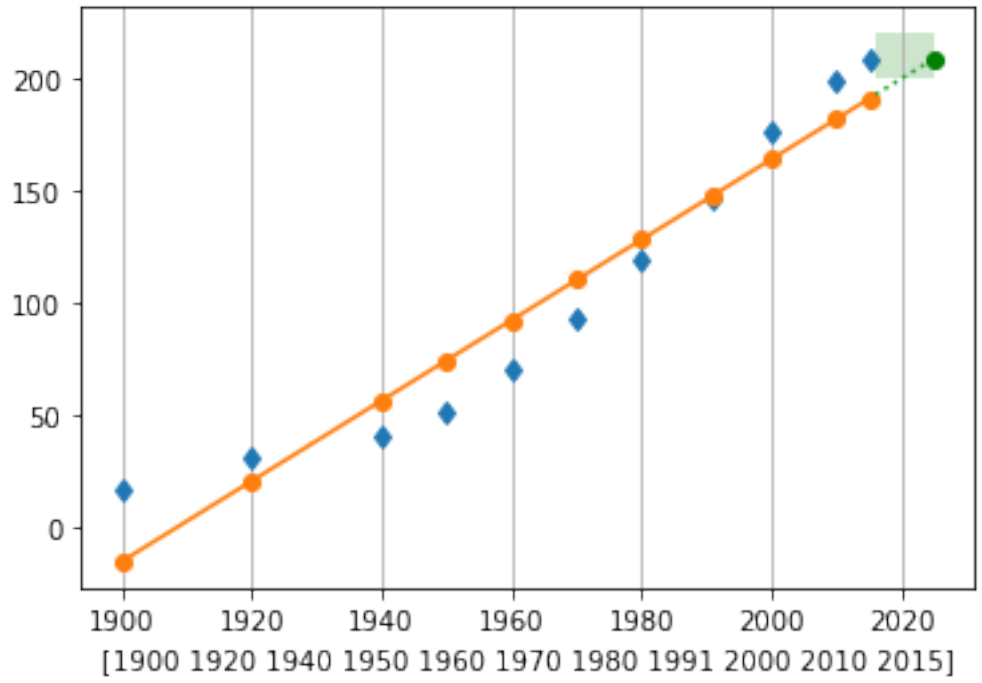
(continues on next page)

(continued from previous page)

```
# linha tracejada e area
ano_m = np.linspace(2016, 2025, num=10, endpoint=True)
v2 = np.ones(np.shape(ano_m))
ax.fill_between(ano_m,200*v2,220*v2,facecolor='g',alpha=0.2)
plt.plot(ano_m,p(ano_m),':')
plt.plot(2025,p(2025),'go')

# estimativa no ano 2025
p(2025)
```

```
Coeficiente de ajuste p[0]: 1.7924185769967504
Coeficiente de ajuste p[1]: -3420.8153029001523
```

```
208.83231551826748
```




```
# abrindo vetor de 1900 a 2024
anos = np.arange(ano[0],ano[-1]+10)
pops = p(anos) # população

# encontrando ano em que a populacao superou 100 mi
lim100 = np.nonzero(pops>100)
lim150 = np.nonzero(pops>150)
lim200 = np.nonzero(pops>200)
np.shape(lim100)
print('Marca de 100 milhões de pessoas: ' + str(anos[lim100[0][0]]))
print('Marca de 150 milhões de pessoas: ' + str(anos[lim150[0][0]]))
print('Marca de 200 milhões de pessoas: ' + str(anos[lim200[0][0]]))
```

```
Marca de 100 milhões de pessoas: 1965
Marca de 150 milhões de pessoas: 1993
```

(continues on next page)

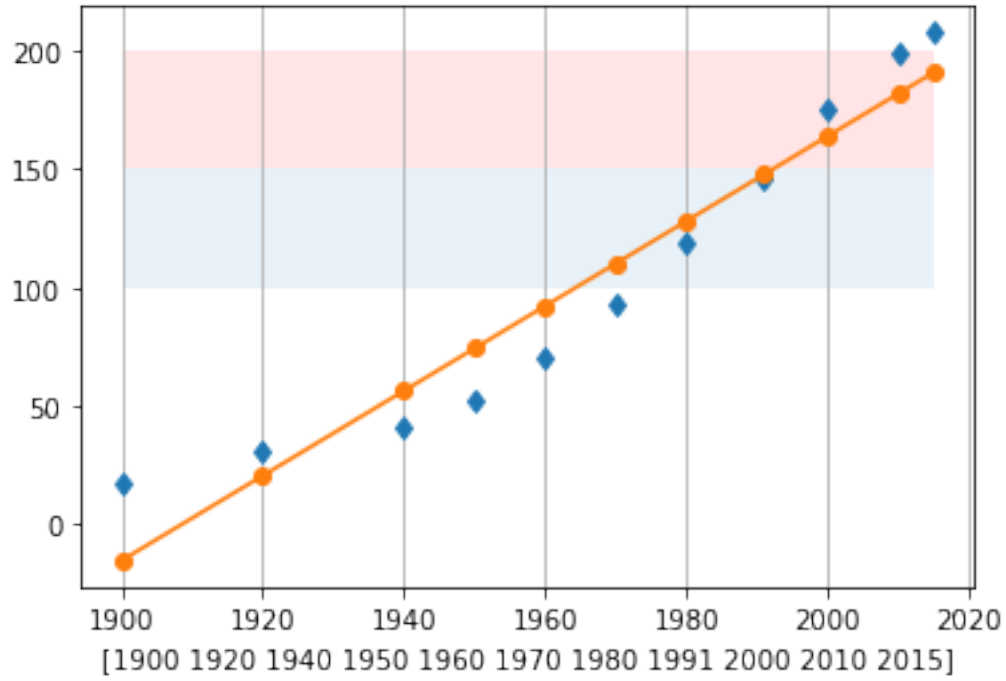
(continued from previous page)

```
Marca de 200 milhões de pessoas: 2021
```

```
# plotagem por faixas de valores
fig,ax = plt.subplots(1,1)
p = resolve_ajuste(ano,hab,1)
ax.grid(axis='x')
v1 = np.ones(np.shape(ano))
ax.fill_between(ano,100*v1,150*v1,alpha=0.1)
ax.fill_between(ano,150*v1,200*v1,facecolor='r',alpha=0.1)
```

```
Coeficiente de ajuste p[0]: 1.7924185769967504
Coeficiente de ajuste p[1]: -3420.8153029001523
```

```
<matplotlib.collections.PolyCollection at 0x7fc47246b410>
```



48.3.2 AJUSTE POR PARÁBOLA

```
# plotagem
fig,ax = plt.subplots(1,1)
p = resolve_ajuste(ano,hab,2)
ax.grid(axis='x')

# plotagem do comportamento preditivo

# linha tracejada e area
ano_m = np.linspace(2016, 2025, num=10, endpoint=True)
v2 = np.ones(np.shape(ano_m))
ax.fill_between(ano_m,200*v2,260*v2,facecolor='g',alpha=0.2)
plt.plot(ano_m,p(ano_m),':')
```

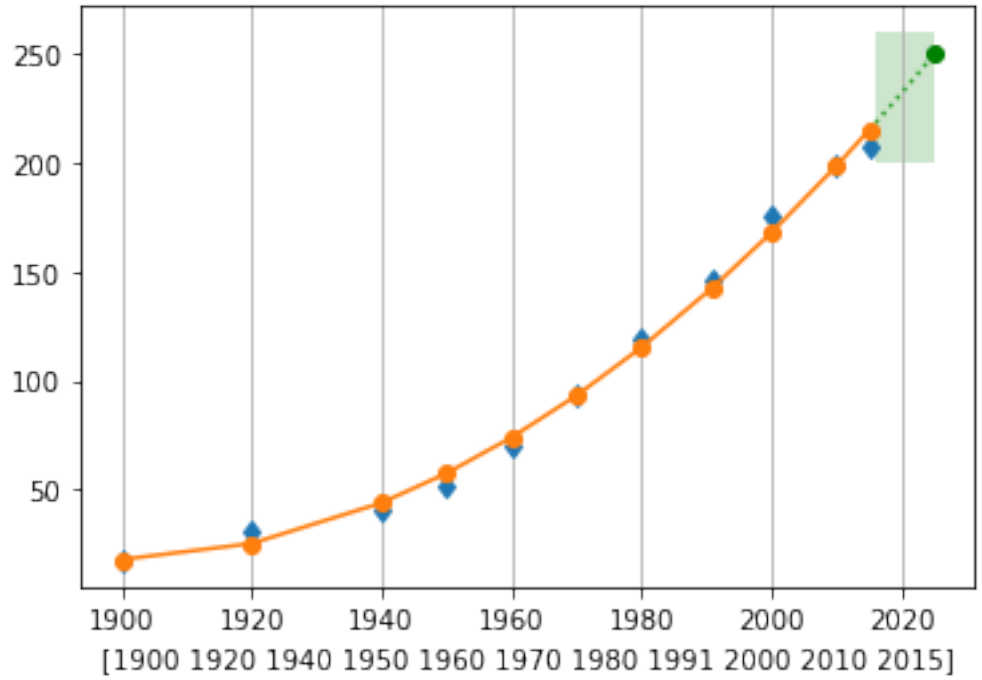
(continues on next page)

(continued from previous page)

```
plt.plot(2025,p(2025),'go')

# estimativa no ano 2025
p(2025)
```

```
Coeficiente de ajuste p[0]: 0.01418349430843006
Coeficiente de ajuste p[1]: -53.81354030390032
Coeficiente de ajuste p[2]: 51061.1779002923
```

```
249.95013340016885
```




```
# abrindo vetor de 1900 a 2024
anos = np.arange(ano[0],ano[-1]+10)
pops = p(anos) # população

# encontrando ano em que a populacao superou 100 mi
lim100 = np.nonzero(pops>100)
lim150 = np.nonzero(pops>150)
lim200 = np.nonzero(pops>200)
np.shape(lim100)
print('Marca de 100 milhões de pessoas: ' + str(anos[lim100[0][0]]))
print('Marca de 150 milhões de pessoas: ' + str(anos[lim150[0][0]]))
print('Marca de 200 milhões de pessoas: ' + str(anos[lim200[0][0]]))
```

```
Marca de 100 milhões de pessoas: 1974
Marca de 150 milhões de pessoas: 1994
Marca de 200 milhões de pessoas: 2011
```

```
# plotagem por faixas de valores
fig,ax = plt.subplots(1,1)
```

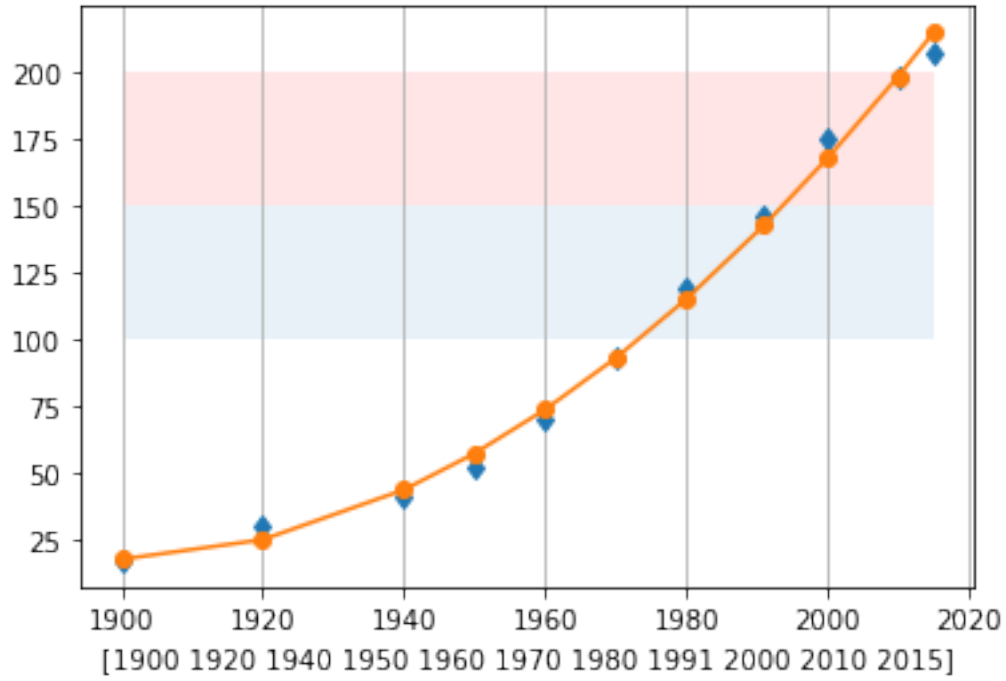
(continues on next page)

(continued from previous page)

```
p = resolve_ajuste(ano,hab,2)
ax.grid(axis='x')
v1 = np.ones(np.shape(ano))
ax.fill_between(ano,100*v1,150*v1,alpha=0.1)
ax.fill_between(ano,150*v1,200*v1,facecolor='r',alpha=0.1)
```

```
Coeficiente de ajuste p[0]: 0.01418349430843006
Coeficiente de ajuste p[1]: -53.81354030390032
Coeficiente de ajuste p[2]: 51061.1779002923
```

```
<matplotlib.collections.PolyCollection at 0x7fc471ff9310>
```



48.4 Integração Numérica

48.4.1 Função-base para integrais de Newton-Cotes

```
def integral_newton_cotes(x,y,metodo):

    # diff computa h = x[k+1] - x[k];
    # duplo diff retorna 0 se igualmente espaçado
    h = np.diff(x)
    hh = np.diff(h)

    # verifica se vetor é, de fato, de zeros, dentro de tolerância
    # se não, lança erro
    np.testing.assert_allclose(hh, 0*hh, atol=1e-08)

    # switch
```

(continues on next page)

(continued from previous page)

```
    if metodo is 'trapezio':

        # montando vetor de somas y[k] + y[k+1]
        cs = np.cumsum(y)
        u = np.concatenate((np.array([0]),cs[0:-2]))
        ss = cs[1:]-u # somas

        # integral (regra generalizada)
        integral = h[0]/2*np.sum(ss)

    elif metodo is 'simpson13':

        if np.size(x) % 2 is 0:
            raise ValueError('Regra de Simpson válida apenas para número ímpar de
↪pontos.')

        # constroi pesos

        # ignora primeiro e ultimo
        ie = np.array(range(0,np.size(x)))
        ie = ie[1:-1] % 2

        # pesos para intermediarios
        for v in range(np.size(ie)):
            if ie[v] == 1:
                ie[v] = 4
            elif ie[v] == 0:
                ie[v] = 1

        # concatena para recriar
        ie = np.concatenate(([1],ie,[1]))

        # integral (regra generalizada)
        integral = h[0]/3*(np.sum(y*ie))


    return integral
```

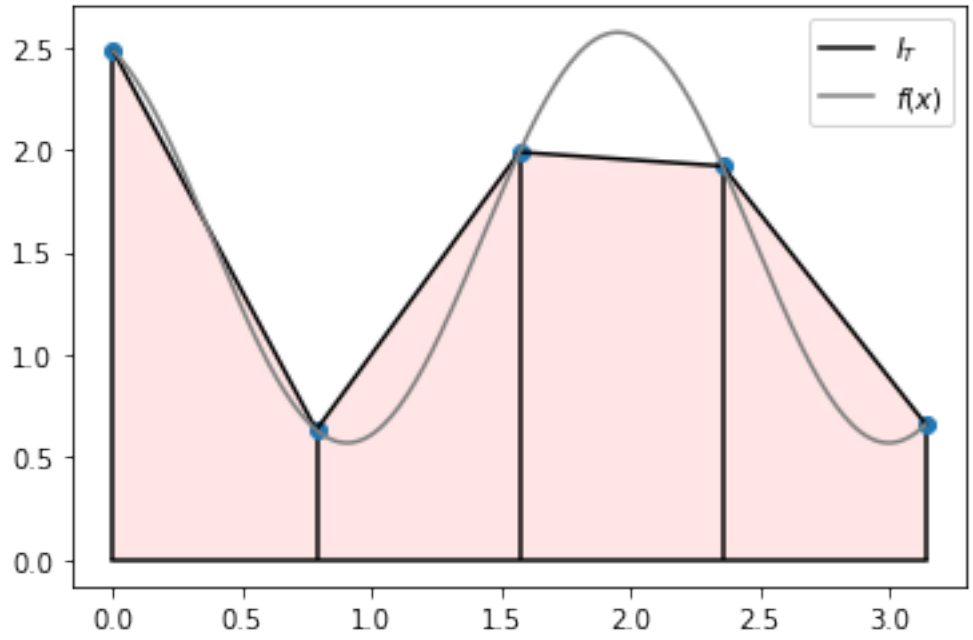
48.4.2 Exemplo

Integração numérica pela regra do trapézio para a função $\int_{a=0}^{b=\pi} [\text{sen}(3x+2) + 0.5\pi]\, dx \approx I_T$

```
# função
a = 0
b = np.pi
x = np.linspace(a,b,num=5,endpoint=True)
f = lambda x: np.sin(3*x + 2) + np.pi/2
y = f(x)

# integração por trapézio
integral_newton_cotes(x,y,'trapezio')
```

```
4.799420241706493
```

```
# plotagem dos trapézios
plt.stem(x,y,'-ok',basefmt='k-',use_line_collection=True)
plt.plot(x,y,'-k',label='$I_T$')
plt.fill_between(x,0*y,y,facecolor='r',alpha=0.1)

# "imitação" da função contínua
xx = np.linspace(a,b,num=200,endpoint=True)
plt.plot(xx,f(xx),color=[0.5,0.5,0.5],label='$f(x)$')
plt.legend();
```



48.4.3 Exemplo

Idem, para regra 1/3 de Simpson I_S

```
# integração por 1/3 Simpson

integral_newton_cotes(x,y,'simpson13')
```

```
4.016218444253924
```

48.5 Função residente em Python para integração

```
import scipy.integrate as sp
```

48.5.1 Exemplo

Mesma função $f(x)$, agora calculada com o Scipy.

```
# scipy::quad (erro absoluto)
integral_ = sp.quad(f,a,b)
integral_
```

```
(4.657370976179917, 1.5031433917569477e-13)
```

48.5.2 solucao-L5-Q9

```
# dados
x = np.array([1.00, 1.05, 1.10, 1.15, 1.20, 1.25, 1.30])
y = np.array([1.0000, 1.0247, 1.0488, 1.0723, 1.0954, 1.1180, 1.1401])

# integral implementada: trapézio
print(integral_newton_cotes(x,y,'trapezio'))

# integral implementada: 1/3 Simpson
print(integral_newton_cotes(x,y,'simpson13'))

# integral scipy: 1/3 Simpson

f = lambda x: np.sqrt(x)
sp.quad(f,x[0],x[-1])
```

```
0.32146250000000026
0.2857383333333336
```

```
(0.32148536841925296, 3.569204580991558e-15)
```

48.5.3 solucao-L5-Q10

```
# caso h = 0.1
x = np.array([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
y = np.array([1, 0.995, 0.980, 0.955, 0.921, 0.877, 0.825, 0.764, 0.696, 0.6216, 0.
↪5403])

print(integral_newton_cotes(x,y,'trapezio'))

# integral implementada: 1/3 Simpson
print(integral_newton_cotes(x,y,'simpson13'))

# residente
f = lambda x: np.cos(x)
print(sp.quad(f,x[0],x[-1]))


# caso h = 0.2
x2 = np.concatenate((x[0:-1:2],[1]))
y2 = np.concatenate((y[0:-1:2],[0.5403]))
```

(continues on next page)

(continued from previous page)

```
print(integral_newton_cotes(x2,y2,'trapezio')) # TODO <=== CHECAR ISTO AQUI!

# integral implementada: 1/3 Simpson
print(integral_newton_cotes(x2[0:-1],y2[0:-1],'simpson13'))

# residente
print(sp.quad(f,x2[0],x2[-1]))
```

```
0.8404750000000001
0.7270900000000001
(0.8414709848078965, 9.34220461887732e-15)
0.83843
0.6557999999999999
(0.8414709848078965, 9.34220461887732e-15)
```

48.5.4 solucao-L5-Q11

```
x = np.array([1.2, 1.3, 1.4, 1.5, 1.6])
y = np.array([0.93204, 0.96356, 0.98545, 0.99749, 0.99957])

print(integral_newton_cotes(x,y,'trapezio'))

# integral implementada: 1/3 Simpson
print(integral_newton_cotes(x,y,'simpson13'))

# residente
f = lambda x: np.sin(x)

print(sp.quad(f,x[0],x[-1]))
```

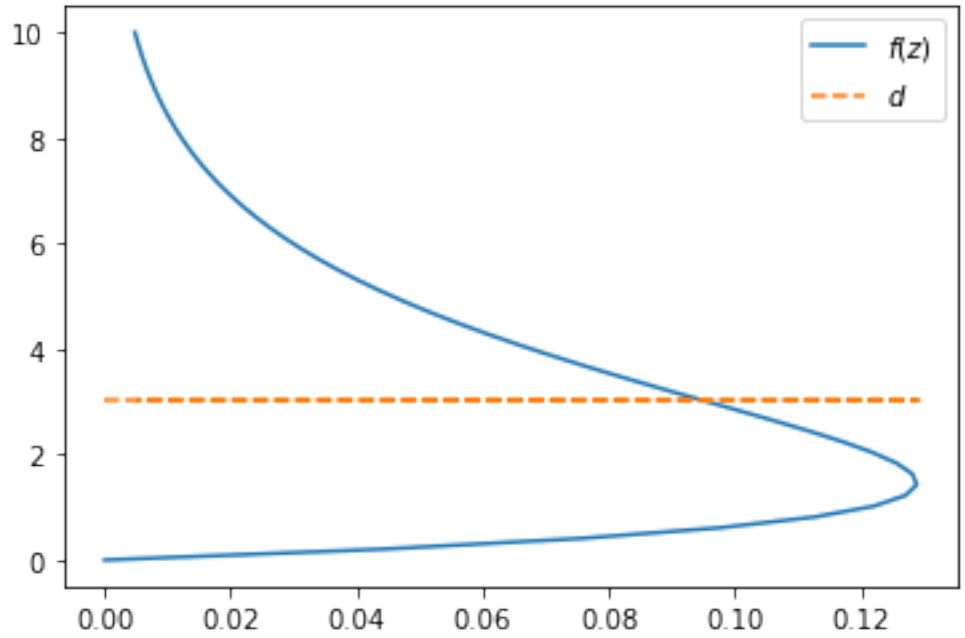
```
0.39123050000000037
0.35870866666666695
(0.3915572767779624, 4.34715904138419e-15)
```

48.5.5 solucao L5-Q17

```
z = np.linspace(0,10,num=50,endpoint=True)
f = lambda z: z/(4+z)*np.exp(-0.5*z)

d = integral_newton_cotes(z,z*f(z),'trapezio')/integral_newton_cotes(z,f(z),'trapezio
 ↪')
print(d)

# residente
f2 = lambda z: ( z/(4+z)*np.exp(-0.5*z) )*z
print(sp.quad(f2,0,10)[0]/sp.quad(f,0,10)[0])
```

```
3.0523917901871935
3.0476097824918496
```

```
# plotagem da força
plt.plot(f(z),z,label='$f(z)$')
plt.plot(f(z),np.ones(np.shape(f(z)))*d,'--',label='$d$')
plt.legend();
```



48.5.6 QUADRATURA GAUSSIANA

Algumas informações:

- A quadratura gaussiana pode ser chamada como

```
from scipy.integrate import quadrature
```

Depois de importar, veja

```
help(quadrature)
```

- A tabela de pesos de quadratura pode ser acessada no Numpy através do comando

```
np.polynomial.legendre.leggauss(ord)
```

Integra exatamente polinômios de grau até

```
2*ord - 1
```

48.5.7 solucao-L5-Q18a

```
from scipy.integrate import quadrature

# integração
f = lambda z: z**3 + z**2 + z + 1
quadrature(f, -1, 1,maxiter=3) # grau máximo = 3; erro = 0
```

```
(2.6666666666666665, 0.0)
```

```
# Verificando computação numérica x simbólica com Sympy
import sympy as sy

# cria símbolo para z
zsym = sy.Symbol('z')

# integração
f = zsym**3 + zsym**2 + zsym + 1
val = sy.integrate(f, (zsym,-1,1))
float(val)
```

```
2.6666666666666665
```

48.5.8 solucao-L5-Q18b

```
# integração
f = lambda x: x**2 -1
quadrature(f, -2, 0,maxiter=3)
```

```
(0.6666666666666667, 2.220446049250313e-16)
```

```
# integração simbólica
f = zsym**2 + -1
val = sy.integrate(f, (zsym,-2,0))
float(val)
```

```
0.6666666666666666
```

Exemplos de integrandos com singularidades

Esses casos não são bem manipulados pelo submódulo `integrate`.

48.5.9 solucao-L5-Q18c

```
# função com singularidade

# integração
f = lambda x: ((1-x**2)**(-1/2))*x**2*x**2
quadrature(f, -1, 1)
```

```
/Users/gustavo/anaconda3/lib/python3.7/site-packages/scipy/integrate/quadrature.
↪py:251: AccuracyWarning: maxiter (50) exceeded. Latest difference = 6.973792e-04
  AccuracyWarning)
```

```
(1.1436022273829545, 0.0006973792417572788)
```

48.5.10 solucao-L5-Q18d

```
# função com singularidade

# integração
f = lambda x: (x**3 +2*x**2)/(4*(4-x**2)**1/2)
quadrature(f, -2, 2,maxiter=3)
```

```
/Users/gustavo/anaconda3/lib/python3.7/site-packages/scipy/integrate/quadrature.
↪py:251: AccuracyWarning: maxiter (3) exceeded. Latest difference = 1.333333e+00
  AccuracyWarning)
```

```
(3.3333333333333357, 1.333333333333336)
```

CHAPTER

FORTYNINE

LISTA DE EXERCÍCIOS 6

Solucionário matemático e computacional de exercícios selecionados da Lista de Exercícios 6.

```
%matplotlib inline
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

49.1 Funções-base para resolucão

```
"""
Implementação do método de Euler
'expr' deve conter variáveis 't' e 'y'.
Para casos em que são 0, fazer '0*t'
ou '0*y'.
"""
def met_euler(expr,n,a,b,y0,mov):

    # expressão
    f = eval('lambda t,y:' + expr)

    # malha
    t = np.linspace(a,b,num=n,endpoint=True)

    # passo
    h = (b-a)/(n-1)
    print('Tamanho do passo: h = {0}'.format(h))

    # y(t)
    y = 0*t

    # resolve para frente
    if mov is 'front':
        # cond. inicial
        y[0] = y0

        # esquema
        for i in range(0,n-1):
            y[i+1] = y[i] + h*f(t[i],y[i])

        return t,y
```

(continues on next page)

(continued from previous page)

```
    # resolve para trás
    elif mov is 'back':

        # cond. inicial
        y[-1] = y0

        # esquema
        for i in range(n-1,0,-1):
            y[i-1] = y[i] - h*f(t[i],y[i])

        return t,y


def met_pt_medio(expr,n,a,b,y0):

    # expressão
    f = eval('lambda t,y:' + expr)

    # malha
    t = np.linspace(a,b,num=n,endpoint=True)

    # passo
    h = (b-a)/(n-1)
    print('Tamanho do passo: h = {0}'.format(h))

    # y(t)
    y = 0*t

    # cond. inicial
    y[0] = y0

    # esquema
    for i in range(0,n-1):
        tmed = t[i] + h/2
        ymed = y[i] + h/2*f(t[i],y[i])
        y[i+1] = y[i] + h*f(tmed,ymed)

    return t,y
```

49.2 Soluções da Lista 6

49.2.1 solucao-L6-Q1a

```
# PVI
# h = 0.1 => 5 pontos
expr = 'y**2 - t/2 + 3'


# resolve para a frente
y0 = 1
a = 1.2
```

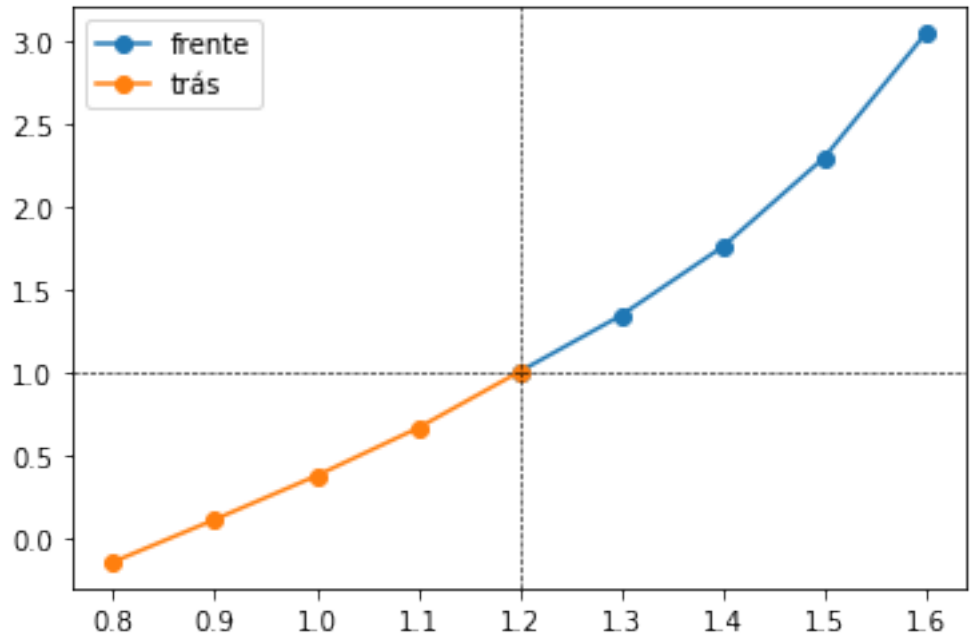
(continues on next page)

(continued from previous page)

```
b = 1.6
n = 5
mov = 'front'
x1,y1 = met_euler(expr,n,a,b,y0,mov)


# resolve para trás
y0 = 1
a = 0.8
b = 1.2
n = 5
mov = 'back'
x2,y2 = met_euler(expr,n,a,b,y0,mov)

# aproximação y(0.8)
print('Aproximação y(0.8) = {0}'.format(y2[0]))

# aproximação y(1.2)
print('Aproximação y(1.6) = {0}'.format(y1[-1]))

# plotagem
plt.plot(x1,y1,'o-',label='frente')
plt.plot(x2,y2,'o-',label='trás')
plt.axvline(x=1.2,color='k',linewidth=0.6,linestyle='--')
plt.axhline(y=1.0,color='k',linewidth=0.6,linestyle='--')
plt.legend()
```

```
Tamanho do passo: h = 0.10000000000000003
Tamanho do passo: h = 0.09999999999999998
Aproximação y(0.8) = -0.14851547391331865
Aproximação y(1.6) = 3.042921559591501
```

```
<matplotlib.legend.Legend at 0x1518bfbda0>
```



49.2.2 solucao-L6-Q1b

```
# resolve por ponto médio
y0 = 1
a = 1.2
b = 1.6
n = 5
xm,ym = met_pt_medio(expr,n,a,b,y0)

# resolve por função residente
f = eval('lambda t,y:' + expr)
xr = np.linspace(1.2,1.6,num=5,endpoint=True)
yr = odeint(f,1,xr)
yr
```

```
Tamanho do passo: h = 0.10000000000000003
```

```
array([[1.        ],
       [1.39644385],
       [1.7989217 ],
       [2.20909012],
       [2.62852487]])
```

49.2.3 Solução analítica para a EDO do PVI

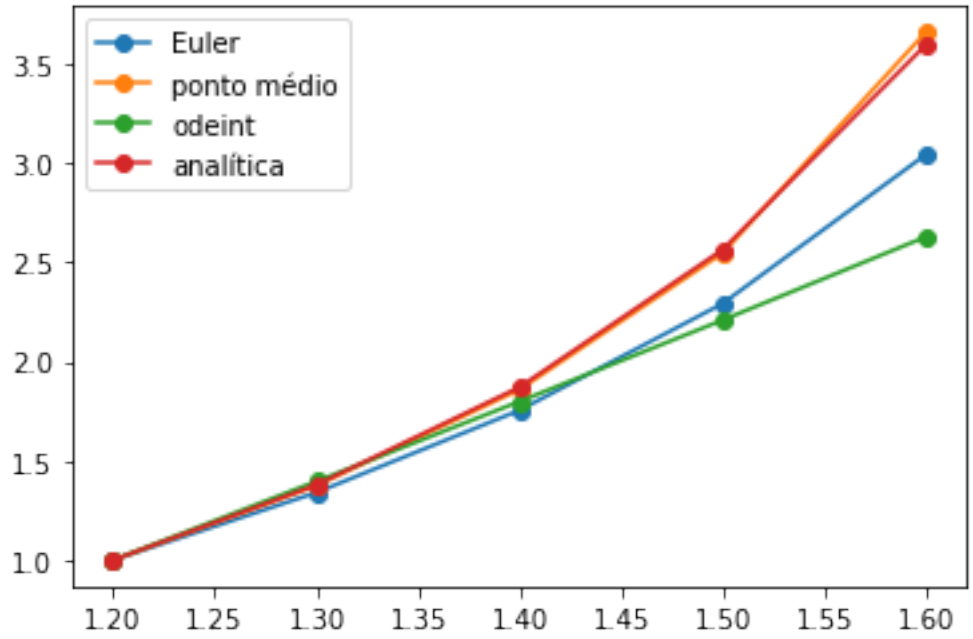
```
# Encontrando a solução por computação simbólica
import sympy as sp
sp.init_printing()

# variável simbólica
tsym = sp.symbols('t')

# função
f = sp.symbols('f', cls=sp.Function)

# EDO
edo = sp.Eq( f(tsym).diff(tsym), f(tsym)**2 - tsym/2 + 3)

# solução
# lembre que: t0 = 1.2; y0 = 1
sol = sp.dsolve(edo,f(tsym),ics={f(1.2):1})

# usa membro direito e remove big-oh
sol = sol.rhs.removeO()

# substitui expressão simbólica por numérica
yt = [sol.subs(tsym,i) for i in xr]
yt = np.asarray(yt)

print("Solução analítica:")
sol
```

```
Solução analítica:
```

$$3.4t + 13.4137\,(t-1.2)^5 + 8.33166666666667\,(t-1.2)^4 + 5.95333333333333\,(t-1.2)^3 + 3.15\,(t-1.2)^2 - 3.08$$

49.2.4 Comparação de soluções

```
# compara todos os resultados
plt.plot(x1,y1,'o-',label='Euler')
plt.plot(xm,ym,'o-',label='ponto médio')
plt.plot(xr,yr.T[0],'o-',label='odeint')
plt.plot(xr,yt,'o-',label='analítica')
plt.legend();
```




```
# variável simbólica
tsym = sp.symbols('t')

# função
f = sp.symbols('f', cls=sp.Function)

# EDO
edo = sp.Eq( f(tsym).diff(tsym) - 4*sp.exp(0.8*tsym) + 0.5*f(tsym),0)

# solução
sol = sp.dsolve(edo,f(tsym),ics={f(0):2})

# usa membro direito e remove big-oh
sol = sol.rhs.removeO()

# substitui expressão simbólica por numérica
yt = [sol.subs(tsym,i) for i in xr]
yt = np.asarray(yt)
```

(continues on next page)

(continued from previous page)

```
print("Solução analítica:")
sol
```

```
Solução analítica:
```

$$\left(3.07692307692308e^{1.3t} - 1.07692307692308\right) e^{-0.5t}$$

Part X

Extra

CHAPTER

FIFTY

NÚMEROS EM PONTO FLUTUANTE E SEUS PROBLEMAS

Este capítulo é um compêndio sobre a representação numérica em computadores, sistema de ponto flutuante, bem como notas sobre causas de erros. O texto é essencialmente uma reprodução adaptada do conteúdo encontrado no site floating-point-gui.de.

50.1 Perguntas e respostas

50.1.1 Por que a soma `0.1 + 0.2` não é exatamente 0.3, mas resulta em `0.30000000000000004`?

Porque, internamente, os computadores usam um formato de ponto flutuante binário limitado que não pode representar com precisão números como 0.1, 0.2 ou 0.3. Quando o código é compilado ou interpretado, `0.1` já é arredondado para o número mais próximo consoante este formato, o que resulta em um pequeno erro de arredondamento antes mesmo de o cálculo acontecer.

```
# ?!
0.1 + 0.2
```

```
0.30000000000000004
```

50.1.2 Este sistema parece um tanto estúpido. Por que os computadores o utilizam para fazer cálculos?

Não se trata de estupidez. O sistema é apenas diferente. Números decimais não podem representar com precisão qualquer número, a exemplo da fração `1/3`. Então, temos que usar algum tipo de arredondamento para algo como `0.33`. Afinal, não podemos esperar que a soma `0.33 + 0.33 + 0.33` seja exatamente igual a `1`, não é mesmo?

Os computadores usam números binários porque são mais rápidos ao lidar com eles e, para a maioria dos cálculos usuais, um pequeno erro na 17a. casa decimal não é relevante, pois, de qualquer maneira, os números com os quais trabalhamos não são "redondos" (ou exatamente "precisos").

50.1.3 O que pode ser feito para evitar este problema?

Isso depende do tipo de cálculo pretendido.

Se a precisão dos resultados for absoluta, especialmente quando se trabalha com cálculos monetários, é melhor usar um tipo de dado especial, a saber o `decimal`. Em Python, por exemplo, há um módulo de mesmo nome para esta finalidade específica. Por exemplo:

```
# 0.1 + 0.2
from decimal import Decimal

a,b = Decimal('0.1'), Decimal('0.2')
print(a+b)
```

```
0.3
```

Se apenas se deseja enxergar algumas casas decimais, pode-se formatar o resultado para um número fixo de casas decimais que será exibido de forma arredondada.

```
# imprime resultado como float
print(0.1 + 0.2)

# imprime resultado com 3 casas decimais
print(f'{0.1 + 0.2:.3f}')
```

```
0.30000000000000004
0.300
```

50.1.4 Por que outros cálculos como `0.1 + 0.4` funcionam corretamente?

Nesse caso, o resultado, `0.5`, **pode** ser representado exatamente como um número de ponto flutuante e é possível que haja erro de cancelamento nos números de entrada. Porém, pode não podemos confiar interamente nisto. Por exemplo, quando tais dois números são primeiro armazenados em representações de ponto flutuante de tamanhos diferentes, os erros de arredondamento podem não compensar um com o outro.

Em outros casos, como `0.1 + 0.3`, o resultado **não é realmente** `0.4`, mas próximo o suficiente para que `0.4` seja o menor número mais próximo do resultado do que qualquer outro número em ponto flutuante. Muitas linguagens de programação exibem esse número em vez de converter o resultado real de volta para a fração decimal mais próxima. Por exemplo:

```
print(0.1 + 0.4)
print(0.1 + 0.1 + 0.1 - 0.3) # 0 !
```

```
0.5
5.551115123125783e-17
```

50.2 Comparação de números em ponto flutuante

Devido a erros de arredondamento, a representação da maioria dos números em ponto flutuante torna-se imprecisa. Enquanto essa imprecisão permanecer pequena, ela poderá ser geralmente ignorada. No entanto, às vezes, os números que esperamos ser iguais (por exemplo, ao calcular o mesmo resultado por diferentes métodos corretos) diferem ligeiramente de tal forma que um mero teste de igualdade implica em falha. Por exemplo:

```
a = 0.15 + 0.15
b = 0.1 + 0.2

# Os resultados abaixo deveriam ser verdadeiros (True),
# mas não o são!
print(a == b)
print(a >= b)
```

```
False
False
```

50.2.1 Margens de erro: absoluto x relativo

Ao compararmos dois números reais, o melhor caminho a seguir não é verificar se os números são exatamente iguais, mas se a *diferença entre ambos é muito pequena*. A margem de erro com a qual a diferença é comparada costuma ser chamada de "epsilon". A forma mais simples seria por meio do erro absoluto. Por exemplo:

```
# comparação por erro absoluto
if abs(a - b) < 1e-5:
    print('OK!')
```

```
OK!
```

Isto é, a expressão acima é matematicamente equivalente a $|a - b| < \epsilon$ para $\epsilon = 10^{-5}$.

Entretanto, essa é uma maneira ruim de comparar números reais, porque um *epsilon* fixo escolhido que parece "pequeno" pode, na verdade, ser muito grande quando os números comparados também forem muito pequenos. A comparação retornaria *verdadeiro* para números bastante diferentes. E quando os números são muito grandes, o *epsilon* pode acabar sendo menor que o menor erro de arredondamento, de forma que a comparação sempre retorna *falso*.

Assim, é razoável verificar se o *erro relativo* é menor que o *epsilon*.

```
# comparação por erro relativo
if abs((a - b)/b) < 1e-5:
    print('OK!')
```

```
OK!
```

Mas, esta forma ainda não é inteiramente correta para alguns casos especiais, a saber:

- quando tanto `a` quanto `b` são iguais a zero, a fração resultante `0.0/0.0` é uma indefinição do tipo *not a number* (`NaN`), a qual gera uma exceção em algumas plataformas, ou retorna *falso* para todas as comparações;

```
a,b = 0,0
abs((a-b)/b) # exceção 'ZeroDivisionError'
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-175-fb6d2c8ce8ce> in <module>
      1 a,b = 0,0
----> 2 abs((a-b)/b) # exceção 'ZeroDivisionError'

ZeroDivisionError: division by zero
```

- quando apenas `b` é igual a zero, a divisão produz o *infinito* (∞), que pode gerar uma exceção ou ser maior do que epsilon mesmo quando `a` for menor;

```
a,b = 1e-1,0
abs((a-b)/b) # exceção 'ZeroDivisionError'
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-176-4bc2e683f9ff> in <module>
      1 a,b = 1e-1,0
----> 2 abs((a-b)/b) # exceção 'ZeroDivisionError'

ZeroDivisionError: float division by zero
```

- quando `a` e `b` são muito pequenos, mas estão em lados opostos a zero, a comparação retorna *falso*, ainda que ambos sejam os menores números diferentes de zero possíveis.

```
a,b = -0.009e-20,1.2e-19
abs((a-b)/b) < 1e-5
```

```
False
```

Além disso, o resultado pode não ser sempre comutativo. Isto é, `abs((a-b)/b)` pode ser diferente de `abs((b-a)/b)`.

É possível escrever uma função – veja abaixo – capaz de passar em vários testes para casos especiais, porém ela usa uma lógica com pouca obviedade. A margem de erro deve ser definida de maneira diferente dependendo dos valores de `a` ou `b`, porque a definição clássica de erro relativo torna-se insignificante nesses casos.

```
def compare_float(a:float,b:float,eps:float) -> bool: # ":" e "->" são apenas␣
↪anotações didáticas

    import sys
    MIN = sys.float_info.min # menor float : ~ 1.80e+308
    MAX = sys.float_info.max # maior float : ~ -2.23e+308

    diff = abs(a-b)
    a = abs(a)
    b = abs(b)

    if (a == b): # trata 'inf'
        return True
    elif (a == 0 or b == 0 or (a + b < MIN)): # a ou b = 0 ou ambos extremamente␣
↪próximos de 0
        return diff < (eps*MIN)
    else: # erro relativo
        return diff/(min(a + b, MAX)) < eps
```

Exemplos:

```
print(compare_float(1e-3,1e-3,1e-10))
print(compare_float(1e-3,1.1e-3,0.01))
print(compare_float(10.111,10.1111,1e-5))
print(compare_float(10.111,10.1111,1e-6))
```

```
True
False
True
False
```

50.2.2 Referências para estudo

Miscelânea

- IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic
- What Every Computer Scientist Should Know About Floating-Point Arithmetic, artigo publicado por David Goldberg na ACM Computing Surveys, Vol. 23 (1), 1991. DOI: 10.1145/103162.103163
- William Kahan's Homepage (arquiteto do padrão IEEE 754, e vários outros links)
- Decimal Arithmetic FAQ (Frequently Asked Questions)
- Python documentation - `decimal` module
- Is floating point math broken? @StackOverflow
- The Perils of Floating Point

Visualizadores interativos de números no padrão IEEE 754

- IEEE 754 Visualization
- [Float exposed]

Exemplos de comparação de números em ponto flutuante

- Random ASCII – tech blog of Bruce Dawson

Livros

- Modern Computer Architecture and Organization, por Jim Ledin
- Computer Organization and Architecture Designing for Performance, por William Stallings
- Numerical Computing with IEEE Floating Point Arithmetic, por Michael L. Overton

CHAPTER

FIFTYONE

PROCESSAMENTO DE SINAIS

Na ciência e engenharias, o processamento de sinais é bastante útil, principalmente em áreas como *teoria do controle*. Um sinal pode ser:

- *temporal*: uma quantidade que varia no tempo (ex.: sinal de áudio, música etc.)
- *espacial*: uma quantidade que varia no espaço (ex.: uma imagem 2D)

Sinais são frequentemente funções contínuas. Em aplicações computacionais, entretanto, eles se tornam discretos, visto que são amostrados em um conjunto limitado de pontos separados de distâncias uniformes.

Um algoritmo importantíssimo utilizado para processar sinais é a Transformada Rápida de Fourier, conhecida como *Fast Fourier Transform*, ou simplesmente *FFT*.

51.1 Análise espectral

A análise espectral, como o nome já diz, estuda espectros de frequencias e forma um arcabouço fundamental para aplicação da FFT e de transformadas de Fourier mais gerais. Transformadas são integrais matemáticas que nos permitem transformar um sinal do *domínio temporal* (onde ele é descrito como uma função do tempo) para o *domínio de frequencias* (onde ele é representado como uma função da frequencia).

A representação de um sinal no domínio de frequencias é útil para muitos objetivos, entre os quais podemos citar a extração de frequencias dominantes, a aplicação de filtros e a resolução equações diferenciais.

51.2 Transformadas de Fourier

A expressão matemática para a transformada de Fourier $F(v)$ de um sinal contínuo $f(t)$ é

$$F(v) = \int_{-\infty}^{+\infty} f(t) e^{-2\pi i v t} \, dt.$$

A transformada de Fourier inversa é dada por:

$$f(t) = \int_{-\infty}^{+\infty} F(v) e^{2\pi i v t} \, dv,$$

Acima,

- $F(v)$ é o espectro de amplitudes do sinal $f(t)$ (uma função complexa)
- v é a frequencia
- $F(t)$ é um sinal contínuo com duração infinita

- t é uma coordenada temporal.

Usualmente, aplicações computacionais baseiam-se em amostrar a função $f(t)$ em N pontos uniformemente espaçados $x_0, x_1, \ldots, x_N$ durante um intervalo finito de tempo $0 \leq t \leq T$, de modo que a transformada contínua anterior seja adaptada para a *Transformada de Fourier Discreta* (DFT), dada por:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i n k}{N}},$$

cuja DFT inversa é

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{-2\pi i n k}{N}}.$$

Acima,

- X_k é a DFT das amostras x_n;
- k é uma faixa de frequencia (*bin*).

O cálculo eficiente da DFT é feito pelo algoritmo conhecido como FFT.

51.3 Módulo `fftpack`

Podemos usar o submódulo `fftpack` do `scipy` para ter acesso a implementações da FFT. Veremos uma aplicação das funções `fft` e `ifft`, que são, respectivamente, implementações da FFT geral e de sua inversa.

51.3.1 Importação de módulos

```
from scipy import fftpack
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

Aqui, criamos um sinal simulado com componentes senoidais puras em 1 Hz e em 22 Hz em cima de um ruído de distribuição normal. A função abaixo gera amostras ruidosas deste sinal:

```
def amostra_sinal(t):
    return (2 * np.sin(2 * np.pi * t) +
            3 * np.sin(22 * 2 * np.pi * t) +
            2 * np.random.randn(*np.shape(t)))
```

Uma vez que a DFT toma amostras discretas como entrada e retorna um espectro de frequencias discretas como saída, para usá-la em processos que são originalmente contínuos, primeiro devemos reduzir os sinais para valores discretos usando amostragem (*sampling*).

Pelo teorema da amostragem, um sinal contínuo com largura de banda B (i.e. o sinal não contém frequencias maiores do que B) pode ser completamente reconstruído a partir de amostras discretas com frequencia de amostragem $f_s \geq 2B$. Este resultado nos diz sob quais circunstâncias podemos trabalhar com sinais discretos em vez de contínuos. Ele permite-nos determinar uma taxa de amostragem adequada.

Digamos que estejamos interessados em computar o espectro de frequencias deste sinal até frequencias de 30 Hz. Precisamos escolher a frequencia de amostragem $f_s = 60$ Hz e, se quisermos obter um espectro de frequencias com resolução de $\Delta f = 0.01$ Hz, precisamos coletar pelo menos $N = f_s/\Delta f = 6000$ amostras, correspondendo a um período de amostragem de $T = N/f_s = 100$ segundos.

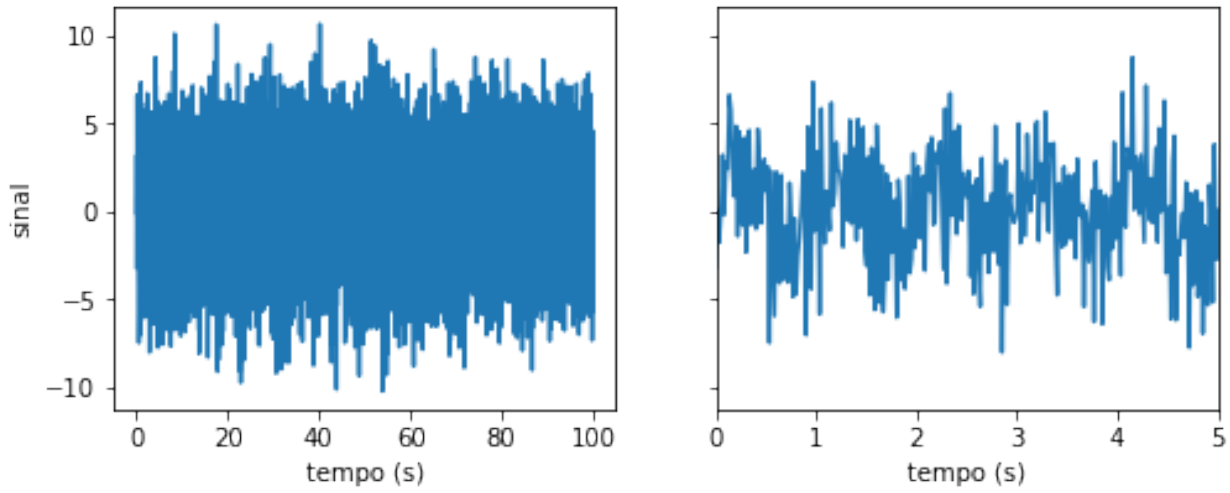
```
B = 30.0 # largura de banda [Hz]
f_s = 2*B # frequencia de amostragem [Hz]
delta_f = 0.01 # espaçamento
N = int(f_s / delta_f) # número de amostras
T = N / f_s # tempo total de amostragem
```

Em seguida, amostramos o sinal em N pontos.

```
t = np.linspace(0,T,N)
f_t = amostra_sinal(t)
```

O sinal é plotado como segue:

```
fig, ax = plt.subplots(1, 2, figsize=(8, 3), sharey=True)
ax[0].plot(t, f_t)
ax[0].set_xlabel("tempo (s)")
ax[0].set_ylabel("sinal")
ax[1].plot(t, f_t)
ax[1].set_xlim(0, 5)
ax[1].set_xlabel("tempo (s)");
```



O sinal é muitor ruidoso. À direita, vemos uma plotagem ampliada. Para revelar as componentes senoidais no sinal, podemos usar a FFT para calcular o espectro do sinal, isto é, sua representação no domínio das frequencias.

```
F = fftpack.fft(f_t)
```

`F` contém as componentes de frequencia do espectro nas frequencias que são determinadas pela taxa de amostragem e pelo número de amostras. Ao computar estas frequencias, é conveniente usar:

```
f = fftpack.fftfreq(N,1.0/f_s)
```

para retornar as frequencias correspondendo a cada *bin* de frequnecia.

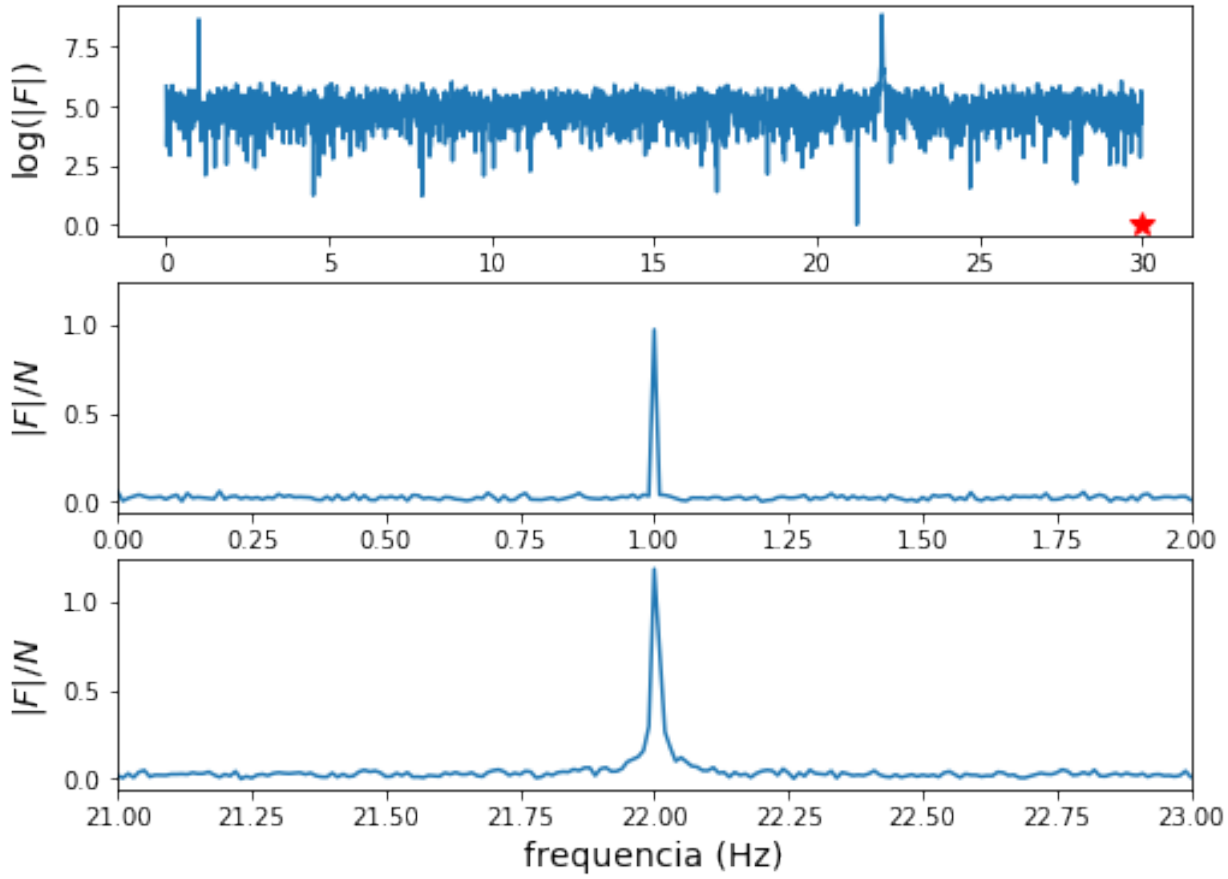
`f` possui frequencias positivas e negativas. Para selecionar as positivas, fazemos:

```
mask = np.where(f >= 0)
```

Em seguida, podemos plotar o espectro com as componentes positivas de frequencia (em escala logarítmica para contrastar sinal e ruído), bem como os dois picos em 1 Hz e 22 Hz que se sobressaem. Ambos correspondem às componentes senoidais existentes no sinal.

Embora o ruído esconda as componentes senoidais no sinal em seu domínio temporal, podemos vê-las claramente presentes no domínio das frequencias.

```
fig, axes = plt.subplots(3, 1, figsize=(8, 6))
axes[0].plot(f[mask], np.log(abs(F[mask])), label="real")
axes[0].plot(B, 0, 'r*', markersize=10)
axes[0].set_ylabel("$\log(|F|)$", fontsize=14)
axes[1].plot(f[mask], abs(F[mask])/N, label="real")
axes[1].set_xlim(0, 2)
axes[1].set_ylabel("$|F|/N$", fontsize=14)
axes[2].plot(f[mask], abs(F[mask])/N, label="real")
axes[2].set_xlim(21, 23)
axes[2].set_xlabel("frequencia (Hz)", fontsize=14)
axes[2].set_ylabel("$|F|/N$", fontsize=14);
```



51.3.2 Filros no domínio das frequencias

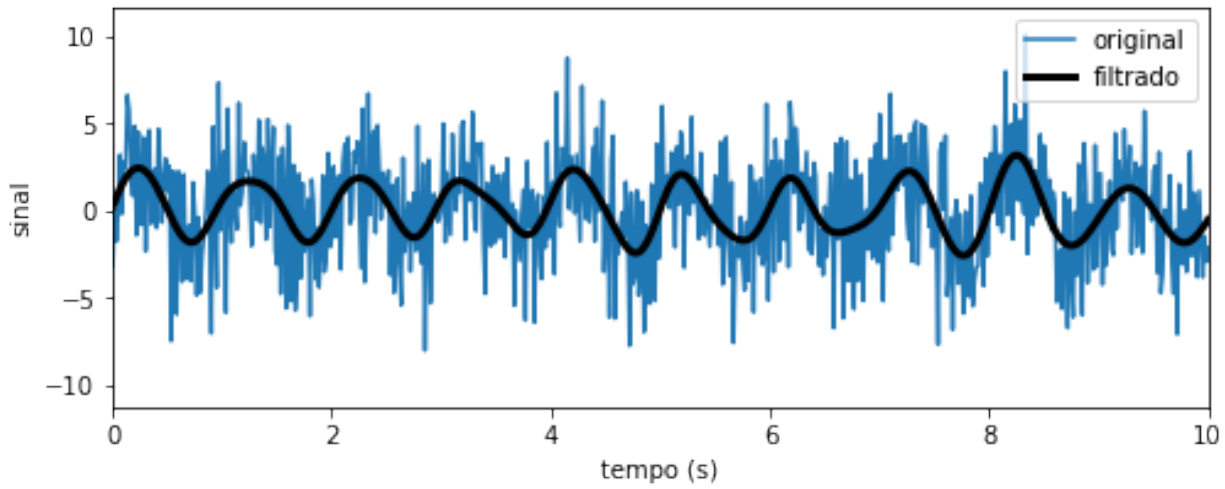
Podemos também computar o sinal no domínio temporal a partir da representação no domínio das frequencias usando a função FFT inversa.

Por exemplo, a aplicação de um filtro passa-baixa (*low pass*) de 2 Hz, i.e. que suprime componentes com frequencia maiores do que 2 Hz nos leva a:

```
F_filtered = F * (abs(f) < 2)
f_t_filtered = fftpack.ifft(F_filtered)
```

O cálculo da FFT inversa para o sinal filtrado resulta em um sinal no domínio temporal onde as oscilações de alta frequencia estão ausentes. Este exemplo resume a essência de muitos filtros aplicáveis ao domínio de frequencias.

```
fig, ax = plt.subplots(figsize=(8, 3))
ax.plot(t, f_t, label='original')
ax.plot(t, f_t_filtered.real, color="black", lw=3, label='filtrado')
ax.set_xlim(0, 10)
ax.set_xlabel("tempo (s)")
ax.set_ylabel("sinal")
ax.legend();
```



CHAPTER

FIFTYTWO

OTIMIZAÇÃO DE CÓDIGO

Python possui diversas bibliotecas para computação numérica que são eficientes e de alto desempenho, tais como *numpy* e *scipy*. A computação de alta performance é atingida não por Python puro, mas pelo aproveitamento de bibliotecas que são compiladas externamente.

Às vezes, temos necessidade de desenvolver código do zero usando puramente Python. Entretanto, essa escolha pode levar a códigos lentos. A solução é escrever rotinas em linguages externas, tais como C, C++ ou Fortran que façam os cálculos que consomem mais tempo e criar interfaces entre elas e o código Puython.

Existem métodos que permitem criar módulos de extensão para Python, tais como:

- módulo *ctypes*
- API Python para C
- CFFI (C foreign function interface)

Embora úteis, todas elas exigem conhecimento aprofundado de outras linguagens e são mais úteis para códigos-fonte já escritos nessas linguagens. Algumas alternativas de desenvolvimento que se aproximam de Python que valem a pena ser consideradas antes de partirmos para uma implementação direta em linguagem complicada existem. Duas delas são: *numba* e *cython*.

52.1 *numba*

[Numba] é um compilador *just-in-time* (JIT) para código Python usando *numpy* que produz código de máquina executável de forma mais eficiente do que o código Python original. Para conseguir isso, *numba* aproveita o conjunto de compiladores [LLVM], que se tornou muito popular nos últimos anos por seu design e interface modular e reutilizável.

52.2 *cython*

[Cython] é um superconjunto da linguagem Python que pode ser traduzido automaticamente para C ou C++ e compilado em um código de máquina executável de modo muito mais rápido do que o código Python. Cython é amplamente utilizado em projetos Python orientados computacionalmente para acelerar partes críticas de tempo de um código que é escrita em Python. Várias bibliotecas dependem muito do Cython. Isso inclui NumPy, SciPy, Pandas e scikit-learn, apenas para mencionar algumas.

Veremos como usar *numba* e *cython* para acelerar códigos originalmente escritos em Python. É recomendado que se faça um perfilamento de código com o módulo `cProfile` ou outras ferramentas para identificar gargalos de cálculo antes de tentarmos otimizar algo.

52.3 Exemplos com numba

```
import numba
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Não precisamos alterar o código-alvo a ser acelerado. Basta usar o JIT do Numba como "decorador" `@numba.jit` para uma função.

52.3.1 Soma de elementos em um array

Consideramos o seguinte código simples.

```
def py_sum(val):
    s = 0
    for v in val:
        s += v
    return s
```

```
val = np.random.rand(50000) # 50000 aleatórios
%timeit py_sum(val)
```

```
6.91 ms ± 1.19 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

A versão vetorizada da somatória é mais rápida.

```
%timeit np.sum(val)
```

```
22.6 µs ± 119 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Teste da função. Se assert não lança erro, temos a condição válida.

```
assert abs(py_sum(val) - np.sum(val)) < 1e-9
```

Tratando com numba.

```
@numba.jit
def jit_sum(val):
    s = 0
    for v in val:
        s += v
    return s
```

```
assert abs(jit_sum(val) - np.sum(val)) < 1e-9 # função numba produz mesmo valor
```

```
%timeit jit_sum(val)
```

```
46.5 µs ± 151 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Comparando isto com a implementação pura, temos uma grande diferença.

52.3.2 Soma cumulativa

```
def py_cumsum(val):
    o = np.zeros_like(val)
    s= 0
    for n in range(len(val)):
        s += val[n]
        o[n] = s
    return o
```

```
%timeit py_cumsum(val)
```

```
16.8 ms ± 3.18 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit np.cumsum(val)
```

```
149 µs ± 4.16 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Utilizando numba:

```
@numba.jit
def jit_cumsum(val):
    o = np.zeros_like(val)
    s= 0
    for n in range(len(val)):
        s += val[n]
        o[n] = s
    return o
```

```
%timeit jit_cumsum(val)
```

```
62.3 µs ± 1.31 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

52.3.3 Fractal de Julia

O fractal de Julia exige um número variável de iterações para cada elemento de uma matriz com coordenadas no plano complexo:

- Um ponto z no plano complexo pertence ao conjunto de Julia se a fórmula de iteração $z \leftarrow z^2 + c$ não diverge após um número grande de iterações.

```
def py_julia_fractal(z_re, z_im, j):
    for m in range(len(z_re)):
        for n in range(len(z_im)):
            z = z_re[m] + 1j * z_im[n]
            for t in range(256):
                z = z ** 2 - 0.05 + 0.68j
                if np.abs(z) > 2.0:
                    j[m, n] = t
                    break
```

Decorador:

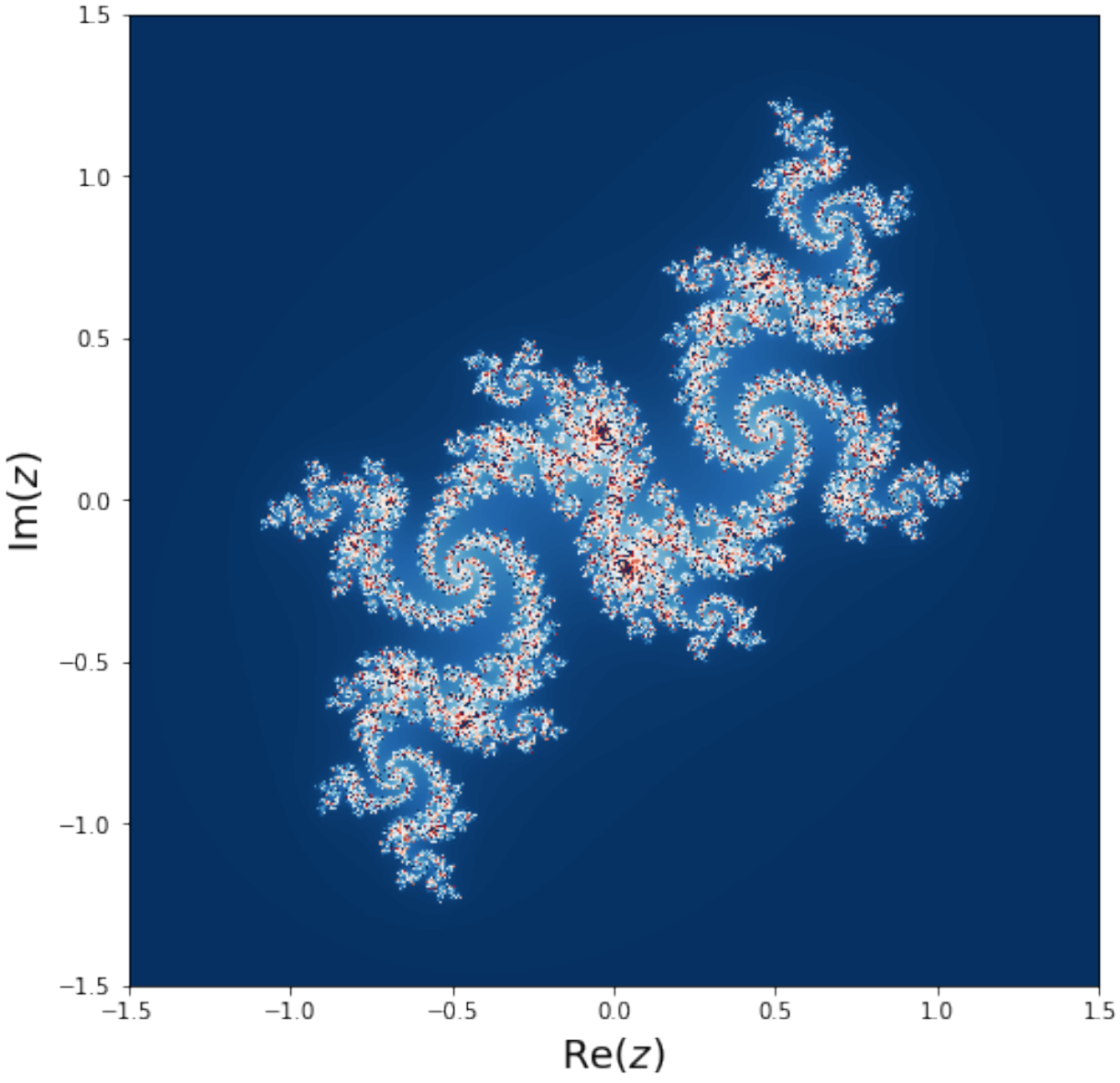
```
jit_julia_fractal = numba.jit(nopython=True)(py_julia_fractal) # nopython :
```

Chamada da função:

```
N = 1024
j = np.zeros((N, N), np.int64)
z_real = np.linspace(-1.5, 1.5, N)
z_imag = np.linspace(-1.5, 1.5, N)
jit_julia_fractal(z_real, z_imag, j)
```

Visualização do fractal gerado por Numba:

```
fig, ax = plt.subplots(figsize=(8, 8))
ax.imshow(j, cmap=plt.cm.RdBu_r, extent=[-1.5, 1.5, -1.5, 1.5])
ax.set_xlabel("$\mathrm{Re}(z)$", fontsize=18)
ax.set_ylabel("$\mathrm{Im}(z)$", fontsize=18);
```



Comparação do tempo em cada chamada:

```
%timeit py_julia_fractal(z_real, z_imag, j)
```

```
56.7 s ± 2.22 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit jit_julia_fractal(z_real, z_imag, j)
```

```
141 ms ± 6.84 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

52.3.4 `numba.vectorize`

Aqui, vamos construir a função de Heaviside:

```
def py_Heaviside(x):
    if x == 0.0:
        return 0.5
    if x < 0.0:
        return 0.0
    else:
        return 1.0
```

```
x = np.linspace(-2, 2, 50001)
%timeit [py_Heaviside(xx) for xx in x]
```

```
17.2 ms ± 443 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Para ter a função aplicada elemento a elemento em um array, fazemos:

```
np_vec_Heaviside = np.vectorize(py_Heaviside)
np_vec_Heaviside(x)
```

```
array([0., 0., 0., ..., 1., 1., 1.])
```

A função `vectorize` não resolve o problema com a performance.

```
%timeit np_vec_Heaviside(x)
```

```
8.25 ms ± 283 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Melhor performance é atingida com:

```
def np_Heaviside(x):
    return (x > 0.0) + (x == 0.0)/2.0
```

```
%timeit np_Heaviside(x)
```

```
194 µs ± 4.91 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Um desempenho ainda melhor pode ser alcançado usando Numba e o decorador `vectorize`, que obtém uma lista de assinaturas de função para a qual gerar o código compilado por JIT.

Aqui, escolhemos gerar funções vetorizadas para duas assinaturas - uma que recebe matrizes de números de ponto flutuante de 32 bits como entrada e saída, definidos como `numba.float32`, e um que recebe matrizes de números de ponto flutuante de 64 bits como entrada e saída, definido como `numba.float64`:

```
@numba.vectorize([numba.float32(numba.float32), numba.float64(numba.float64)])
def jit_Heaviside(x):
    if x == 0.0:
        return 0.5
    if x < 0.0:
        return 0.0
    else:
        return 1.0
```

```
%timeit jit_Heaviside(x)
```

```
31.4 µs ± 650 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

CHAPTER

FIFTYTHREE

MALHAS NUMÉRICAS

Ao transferir a informação de nossos modelos matemáticos do universo *contínuo* para o universo *discreto* do computador, precisamos criar uma *malha numérica* (a construção Cartesiana em gradeado recebe o nome de *grade numérica*), que pode ser estruturada por uma sucessão de pontos tal como uma imitação de uma reta "furada". Em uma dimensão, uma malha numérica *uniforme*, cujo espaçamento entre seus pontos é igual, pode ser definida como uma progressão aritmética:

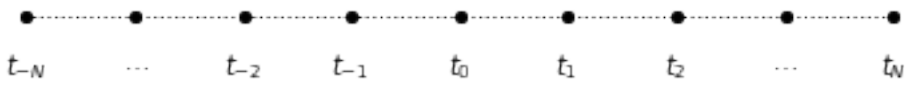
$$t_n = t_0 \pm nh, \;\; n = 1, 2, \ldots, N,$$

onde h é conhecido como *passo*.

53.1 Malhas uniformes

A figura abaixo mostra os pontos de uma malha numérica uniforme. A linha pontilhada foi desenhada meramente para representar a porção do contínuo que o computador **não** captura, isto é, a informação que é perdida.

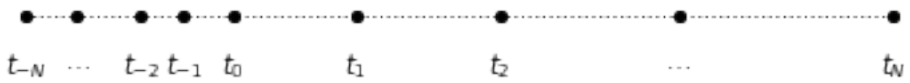
```
# desenha uma malha numerica uniforme

import numpy as np
import matplotlib.pyplot as plt

# pontos
x = np.arange(9)
y = 0*x

# configuracoes
plt.figure(figsize=(6.4,0.5))
plt.plot(x,y,'ok',markersize=4)
plt.plot(x,y,':k',linewidth=1)
plt.ylim((-0.02,0.1))
plt.box(False)
locs, labels = plt.xticks()
plt.xticks(x, ('$t_{-N}$','$\ldots$','$t_{-2}$','$t_{-1}$','$t_{0}$','$t_{1}$','$t_{2}
↪$','$\ldots$','$t_{N}$') )
plt.tick_params(axis='both',width=0.0,labelleft=False)
```



53.2 Malhas não uniformes

Uma malha numérica *não uniforme* é aquela para a qual o tamanho do passo não é constante, como vemos na Figura a seguir. Neste caso, podemos ter comprimentos arbitrários $h_0 \neq h_1 \neq ... \neq h_{N-1}$ todos distintos, tanto à esquerda quanto à direita de t_0.

```
# desenha uma malha numerica nao-uniforme

# pontos
x = np.array([-0.01,0.06,0.15,0.21,0.28,0.45,0.65,0.9,1.2])
y = 0*x

# configuracoes
plt.figure(figsize=(6.4,0.5))
plt.plot(x,y,'ok',markersize=4)
plt.plot(x,y,':k',linewidth=1)
plt.ylim((-0.02,0.1))
plt.box(False)
locs, labels = plt.xticks()
plt.xticks(x, ('$t_{-N}$','$\ldots$','$t_{-2}$','$t_{-1}$','$t_{0}$','$t_{1}$','$t_{2}
↪$','$\ldots$','$t_{N}$') )
plt.tick_params(axis='both',width=0.0,labelleft=False)
```



53.3 Refinamento de malha

Quando uma malha não uniforme possui uma ou mais regiões onde há uma acumulação de pontos, dizemos que ela está *refinada* nessas regiões. Na figura, os pontos t_{-2}, t_{-1} e t_0, por exemplo, acumulam-se próximos um do outro, ao passo que os pontos t_1, t_2 e t_N afastam-se cada vez mais um do outro e da região de refinamento. Então, poderíamos dizer que a malha foi refinada entre t_{-2} e t_0.

O refinamento faz mais sentido apenas quando queremos capturar informações localizadas com mais precisão. Esta situação ocorre, por exemplo, em problemas práticos que envolvem variações ou mudanças abruptas de propriedades, como é o caso da massa específica nas proximidades de uma interface de dois fluidos imiscíveis.

Nos esquemas acima, os nós com índice negativo são apenas ilustrativos e têm pouco efeito prático. Podemos trabalhar apenas com a indexação positiva sem problema algum.

53.4 Malha numérica bidimensional uniforme

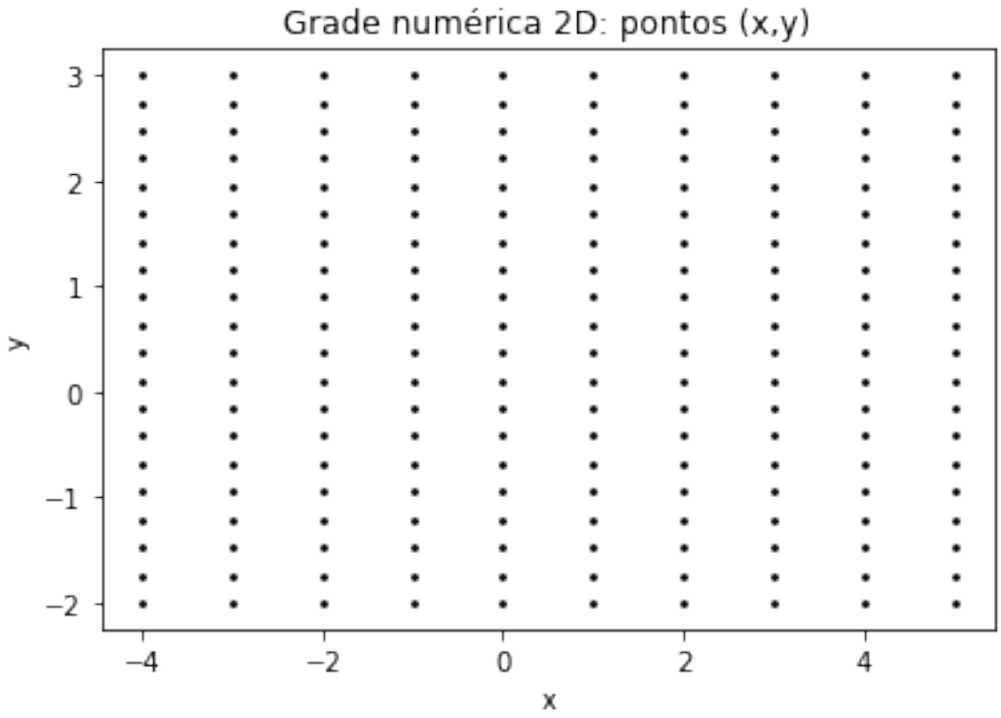
Em particular, uma *malha numérica* bidimensional dependerá de cópias de *arrays* unidimensionais que formam as coordenadas do plano Cartesiano em cada dimensão

```
import numpy as np
import matplotlib.pyplot as plt

# limites do domínio:
```

(continues on next page)

(continued from previous page)

```
# região do plano [a,b] x [c,d]
a, b = -4.0, 5.0
c, d = -2.0, 3.0

# no. de pontos em cada direção
nx, ny = 10, 20

# distribuição dos pontos
x = np.linspace(a,b,nx)
y = np.linspace(c,d,ny)

# grade numérica 2D
[X,Y] = np.meshgrid(x,y)

# plotando pontos da grade numérica
plt.scatter(X,Y,s=3,c='k');
plt.title('Grade numérica 2D: pontos (x,y)')
plt.xlabel('x'); plt.ylabel('y');
```



53.4.1 Plotando curvas de nível

Plotaremos as curvas de nível 0 de funções não-lineares é útil para realizarmos análise gráfica e escolher vetores de estimativa inicial.

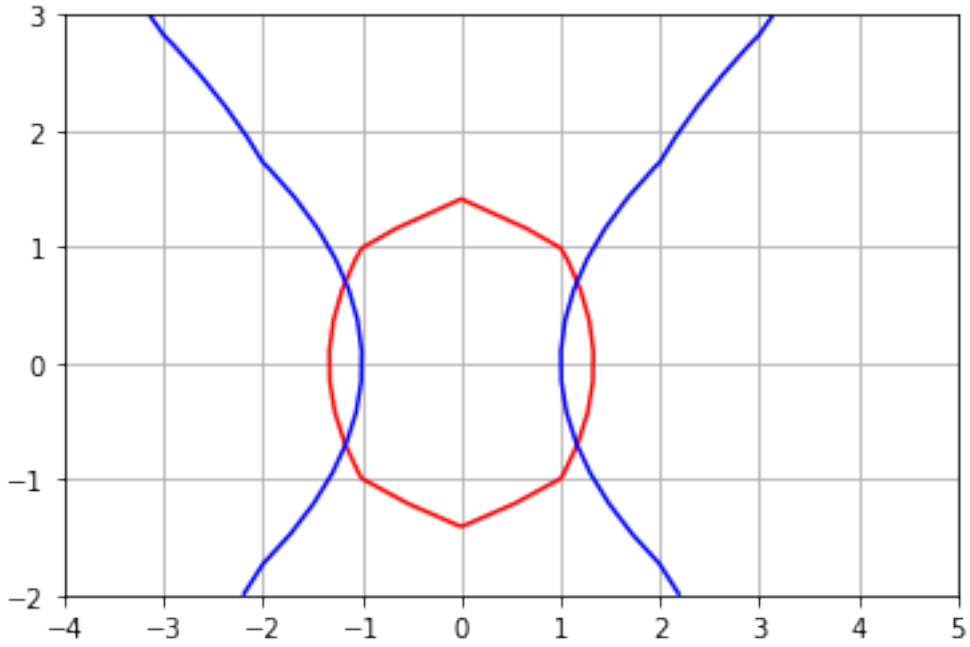
Para plotar curvas de nível das funções sobre a grade numérica anterior, fazemos o seguinte:

```
# funções definidas sobre a grade 2D
F = X**2 + Y**2 - 2
G = X**2 - Y**2 - 1
```

(continues on next page)

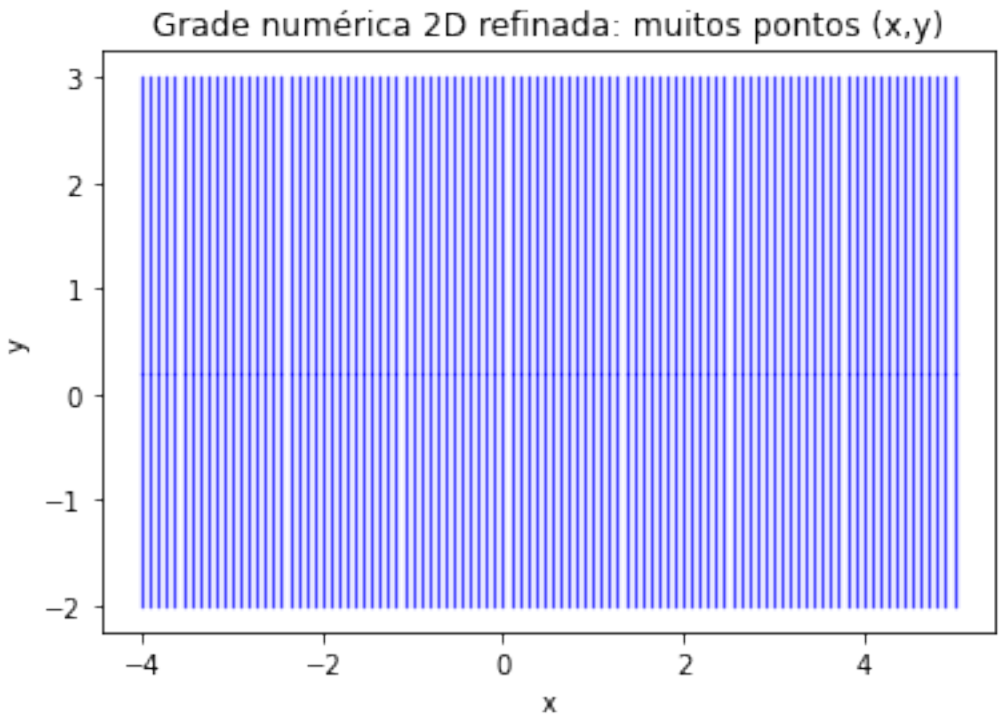
(continued from previous page)

```
# contorno de nível 0
plt.contour(X,Y,F,colors='red',levels=0);
plt.contour(X,Y,G,colors='blue',levels=0);
plt.grid()
```



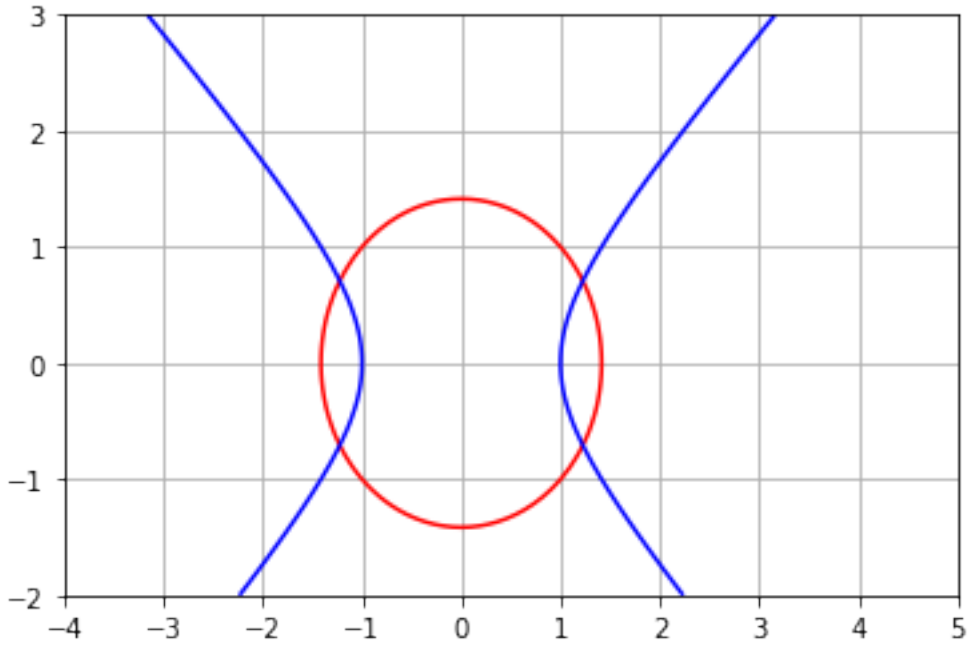
Por que a figura está meio "tosca"? Porque temos poucos pontos na grade. Vamos aumentar o número de pontos. Este processo é conhecido como *refinamento de malha*.

```
# refinando a malha numérica
nx2, ny2 = 100, 200

# redistribuição dos pontos
x2 = np.linspace(a,b,nx2)
y2 = np.linspace(c,d,ny2)

# grade numérica 2D refinada
[X2,Y2] = np.meshgrid(x2,y2)

# plotando pontos da grade numérica
plt.scatter(X2,Y2,s=0.1,c='b');
plt.title('Grade numérica 2D refinada: muitos pontos (x,y)')
plt.xlabel('x'); plt.ylabel('y');
```



Vamos plotar novamente as curvas de nível das funções sobre a grade numérica refinada.

```
# funções definidas sobre a grade 2D refinada
F2 = X2**2 + Y2**2 - 2
G2 = X2**2 - Y2**2 - 1

# contorno de nível 0 na malha refinada
plt.contour(X2,Y2,F2,colors='red',levels=0);
plt.contour(X2,Y2,G2,colors='blue',levels=0);
plt.grid()
```


3
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2
3
4
5

CHAPTER

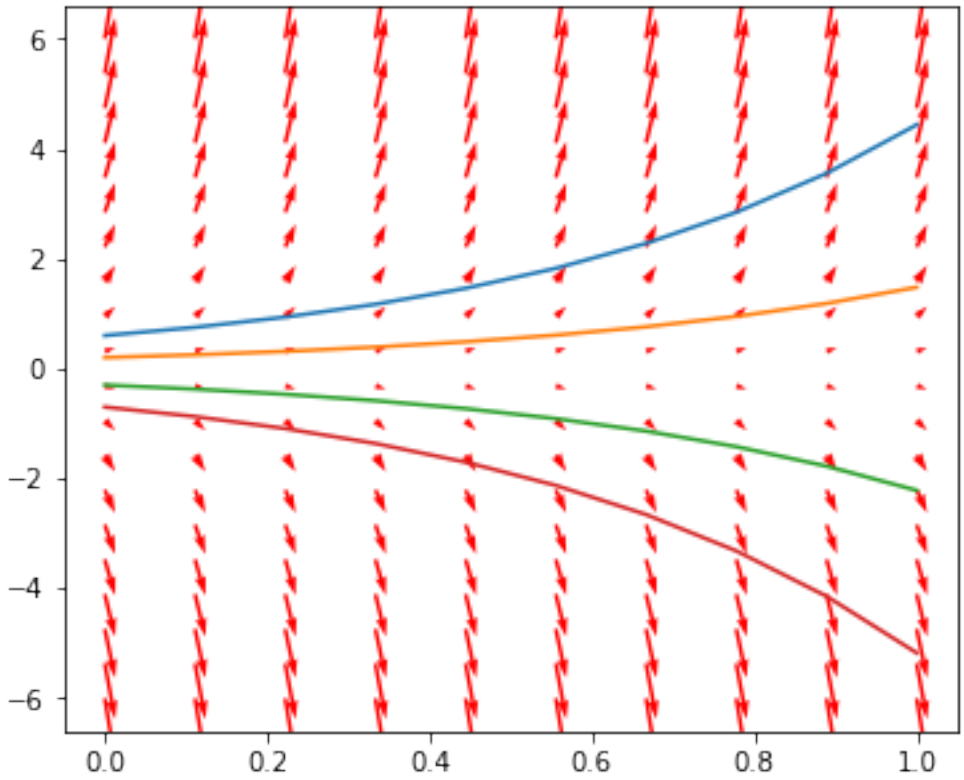
FIFTYFOUR

CAMPOS DE DIREÇÃO

Campos de direção são úteis para entender o comportamento das soluções de uma EDO. O gráfico de uma solução da equação $y' = f(t, y)$ é aquele que, para todo ponto (t, y) do plano, conhecemos a inclinação da curva $y(t)$, solução da EDO. Campos de direção podem ser plotados em Python através das funções `meshgrid`, do pacote `numpy`, e `quiver`, do pacote `matplotlib`.

Exemplo Consideremos a EDO $y' = y$. A inclinação é dada por $f(t, y) = y$ e é independente de t. Vamos gerar o diagrama do campo de direções para esta EDO pelo código a seguir.

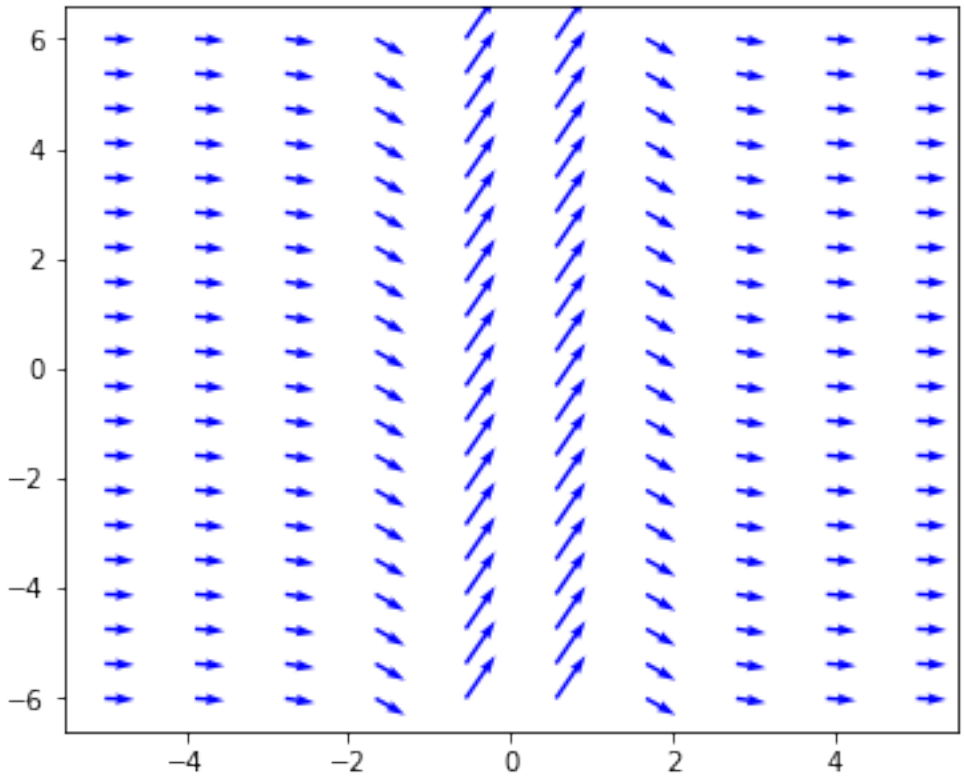
```
import numpy as np
import matplotlib.pyplot as plt

# parametros
t0, tb = 0, 1
y0, yb = -6, 6
nt, ny = 10, 20

# dominio (t,y)
t = np.linspace(t0,tb,nt)
y = np.linspace(y0,yb,ny)
[T,Y] = np.meshgrid(t,y)

# EDO
dt = np.ones(T.shape)
dy = Y

# campo
f = plt.figure(figsize=(6,5))

plt.quiver(T,Y,dt,dy,color='red')

# solucoes particulares
y_part = lambda c: c*np.exp(t)

for c in [0.6,0.2,-0.3,-0.7]:
    aux = y_part(c)
    plt.plot(t,y_part(aux))
```



A solução geral desta EDO é $y(t) = ce^t$. Na figura, destacamos quatro soluções particulares, para valores $c \in \{0.6, 0.2, -0.3, -0.7\}$.

Exemplo: Vamos gerar o diagrama do campo de direções para a EDO $y' = 1/(1-t^2)$.

```
import numpy as np
import matplotlib.pyplot as plt

# parametros
t0, tb = -5, 5
y0, yb = -6, 6
nt, ny = 10, 20

# dominio (t,y)
t = np.linspace(t0,tb,nt)
y = np.linspace(y0,yb,ny)
[T,Y] = np.meshgrid(t,y)

# EDO
dt = np.ones(T.shape)
dy = 1./(1-T**2)

# campo
f = plt.figure(figsize=(6,5))
plt.quiver(T,Y,dt,dy,color='b');
```


6
4
2
0
−2
−4
−6
−4
−2
0
2
4

CHAPTER

FIFTYFIVE

PROBLEMAS

1. Use o Python para plotar os campos de direção para a família de soluções de cada PVI do Problema 1 da Aula 1.
2. Em cada caso, plote as soluções particulares que você encontrou com a substituição de c.

CHAPTER

FIFTYSIX

VARIAÇÕES DO MÉTODO DE EULER

56.1 Método de Euler Melhorado

O *Método de Euler Melhorado* (MEM) é uma técnica numérica explícita de passo simples usada para resolver EDOs que modifica o método explícito de Euler. O MEM usa uma *inclinação ponderada* com derivadas computadas em t_i e t_{i+1}. No início do intervalo, a inclinação é

$$\left.\frac{dy}{dt}\right|_{t=t_i} = f(t_i, y_i),$$

a mesma assumida no Método de Euler Explícito. Todavia, existe uma diferença fundamental na estimativa da inclinação em t_{i+1}. Primeiramente, um valor aproximado para y_{i+1} é determinado como

$$\tilde{y}_{i+1} = y_i + hf(t_i, y_i),$$

que corresponde à estimativa calculada pelo Método de Euler (reta com inclinação constante $f(t_i, y_i)$). Em seguida, usamos este valor para determinar uma nova inclinação que será usada para a integração da EDO, a saber:

$$\left.\frac{dy}{dt}\right|_{t=t_{i+1}, y=\tilde{y}_{i+1}} = f(t_{i+1}, \tilde{y}_i),$$

O valor $\tilde{y}_{i+1}$ é usado como um *preditor*. Então, dispondo dessas duas inclinações, uma inclinação ponderada pela média resulta da seguinte equação:

$$\left.\overline{\frac{dy}{dt}}\right|_{t=t_i} = \frac{1}{2}\left(\left.\frac{dy}{dt}\right|_{t=t_i} + \left.\frac{dy}{dt}\right|_{t=t_{i+1}, y=\tilde{y}_{i+1}}\right)$$

Por sua vez, construímos a *equação de correção* (ou *corretor*) como:

$$y_{i+1} = y_i + \phi(t_i, y_i)h = y_i + \frac{f(t_i, y_i) + f(t_{i+1}, \tilde{y}_{i+1})}{2}h$$

Na realidade, o MEM é um algoritmo da família *preditor-corretor*. A rigor de notação, a aproximação para a solução do PVI é dada pelo processo iterativo a seguir:

$$\begin{aligned} w_0 &= \alpha \\ \tilde{w}_{i+1} &= w_i + hf(t_i, w_i) \\ w_{i+1} &= w_i + \frac{h}{2}[f(t_i, w_i) + f(t_{i+1}, \tilde{w}_{i+1})], \quad i = 0, 1, \ldots, N-1. \end{aligned}$$

56.1.1 Implementação computacional

O seguinte código implementa o MEM:

```
from numpy import *

def euler_melh(t0,tf,y0,h,fun):
    """
    Resolve o PVI y' = f(t,y), t0 <= t <= tf, y(t0) = y0
    com passo h usando o metodo de Euler melhorado.

    Entrada:
        t0  - tempo inicial
        tf  - tempo final
        y0  - condicao inicial
        h   - passo
        fun - funcao f(t,y) (anonima)

    Saida:
        t   - nos da malha numerica
        y   - solucao aproximada
    """

    n = round((tf - t0)/h + 1)
    t = linspace(t0,t0+(n-1)*h,n)
    y = linspace(t0,t0+(n-1)*h,n)
    y = zeros((n,))

    y[0] = y0

    for i in range(1,n):
        ytilde = y[i-1] + h*f(t[i-1],y[i-1])
        ymean = 0.5*( f(t[i-1],y[i-1]) + f(t[i],ytilde) )
        y[i] = y[i-1] + h*ymean

    return (t,y)

### COPIA

def euler_expl(t0,tf,y0,h,fun):
    """
    Resolve o PVI y' = f(t,y), t0 <= t <= tf, y(t0) = y0
    com passo h usando o metodo de Euler explicito.

    Entrada:
        t0  - tempo inicial
        tf  - tempo final
        y0  - condicao inicial
        h   - passo
        fun - funcao f(t,y) (anonima)

    Saida:
        t   - nos da malha numerica
        y   - solucao aproximada
    """

    n = round((tf - t0)/h + 1)
    t = linspace(t0,t0+(n-1)*h,n)
```

(continues on next page)

(continued from previous page)

```
    y = linspace(t0,t0+(n-1)*h,n)
    y = zeros((n,))

    y[0] = y0

    for i in range(1,n):
        y[i] = y[i-1] + h*f(t[i-1],y[i-1])

    return (t,y)
```

O exemplo a seguir é o mesmo que resolvemos com o Método de Euler Explícito.

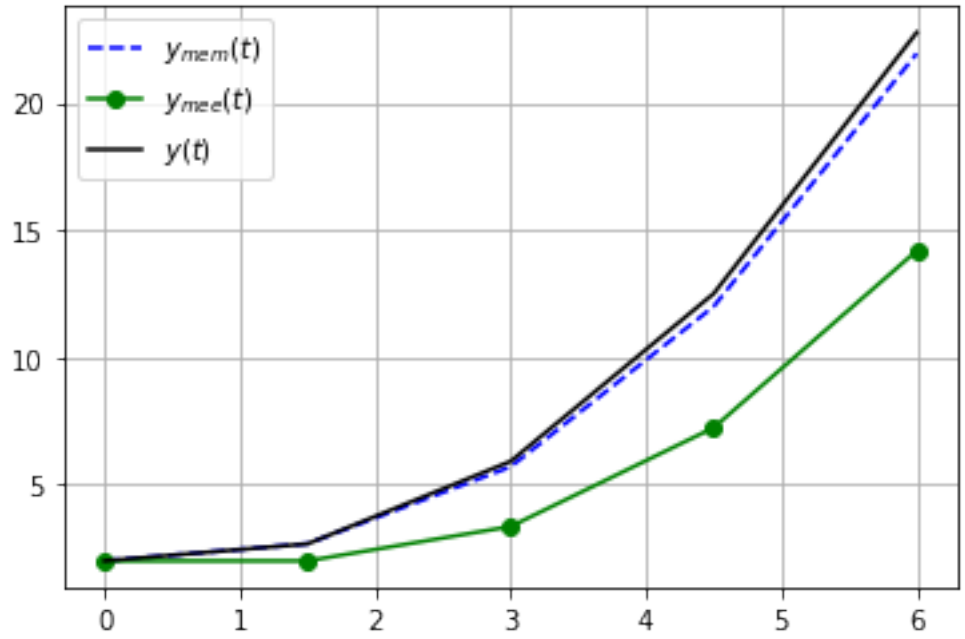
Exemplo: Resolva numericamente

$$\begin{cases} y'(t) = \frac{y(t)+t^2-2}{t+1} \\ y(0) = 2 \\ 0 \leq t \leq 6 \\ h = 0.5 \end{cases}$$

Defina $y_{mem}(t)$ como a solução numérica obtida pelo método de Euler melhorado e $y_{mee}(t)$ como a solução numérica obtida pelo método de Euler explícito. Plote o gráfico das funções aproximadas juntamente com o da solução exata $y(t) = t^2 + 2t + 2 - 2(t+1)\ln(t+1)$

```
import matplotlib.pyplot as plt

# define funcao
f = lambda t,y: (y + t**2 - 2)/(t+1)

# invoca metodo
t0 = 0.0
tf = 6.0
y0 = 2.0
h = 1.5
t,ymem = euler_melh(t0,tf,y0,h,f)
t,ymee = euler_expl(t0,tf,y0,h,f)

# plota funcoes
yex = t**2 + 2*t + 2 - 2*(t+1)*log(t+1)
plt.plot(t,ymem,'b--',label='$y_{mem}(t)$')
plt.plot(t,ymee,'go-',label='$y_{mee}(t)$')
plt.plot(t,yex,'k',label='$y(t)$')
plt.grid(True)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f83ac799f10>
```



56.2 Método do Ponto Médio (ou Ponto Central)

O *Método do Ponto Médio* (MPM) é outra variação do Método de Euler e sua dedução é feita de maneira semelhante à anterior. Neste caso, a inclinação utilizada para o avanço é calculada no ponto médio $t_{i+1/2}$ do passo discreto e dada pela expressão

$$\left.\frac{dy}{dt}\right|_{t=t_i+1/2} = f(t_{i+1/2}, y_{i+1/2})$$

O MPM pode ser formalmente escrito pelo processo iterativo:

$$\begin{aligned} w_0 &= \alpha \\ t_m &= t_i + \frac{h}{2} \\ w_m &= w_i + \frac{h}{2} f(t_i, w_i) \\ w_{i+1} &= w_i + h f(t_m, w_m), \quad i = 0, 1, \ldots, N-1 \end{aligned}$$

O MPM é assim chamado pela relação com a fórmula de quadratura de mesmo nome. Juntamente com o MEM, exige maior esforço computacional, mas são recompensados com redução de erro e melhor acurácia. Além disso, ambos são casos particulares da família de métodos de Runge-Kutta.

Exercício: usando os códigos anteriores como bases, implemente o método do Ponto Médio com o Python.

CHAPTER

FIFTYSEVEN

ESTABILIDADE NUMÉRICA PARA O MÉTODO DE EULER

A seguir, aplicaremos um resultado derivado do contexto da estabilidade teórica do PVI ao método de Euler e estudar alguns aspectos de sua estabilidade numérica.

Definamos uma solução numérica $\{z_n\}$ tal que

$$z_{n+1} = z_n + hf(t_n, z_n), \quad n = 0, 1, \dots, N(h) - 1, \qquad z_0 = y_0 + \epsilon.$$

De modo análogo ao que sabemos para o PVI, usamos uma condição inicial perturbada para verificar que $\{z_n\}$ comporta-se como uma segunda solução $\{y_n\}$ à medida que $h \to 0$.

Tomemos o erro $e_n = z_n - y_n, \quad n \geq 0$. Então, $e_0 = \epsilon$. Subtraindo $y_{n+1} = y_n + hf(t_n, y_n)$ da anterior, obtemos

$$e_{n+1} = e_n + h[f(t_n, z_n) - f(t_n, y_n).$$

Isto tem exatamente o mesmo formato que o erro total para o método de Euler considerando nulo o erro de truncamento. A partir do teorema do limite do erro, podemos concluir que (consultar demonstração):

$$\max_{0 \leq n \leq N(h)} |z_n - y_n| \leq e^{(b-t_0)K}|\epsilon| \leq \kappa|\epsilon|,$$

para uma constante $\kappa \geq 0$. Vale relembrar que K é a constante de Lipschitz. Este resultado diz-nos que, ressalvadas as condições teóricas do PVI, a estabilidade numérica do método é encontrada.

57.1 Notas sobre análise de estabilidade

Consideremos o PVI

$$\begin{cases} y'(t) = -100y(t) \\ y(0) = y_0 \end{cases}$$

cuja solução analítica é $y(t) = y_0 e^{-100t}$. Se $y_0 > 0$, a solução é positiva e decai rapidamente com $t \to \infty$. Se o método de Euler explícito for aplicado a este PVI, teremos

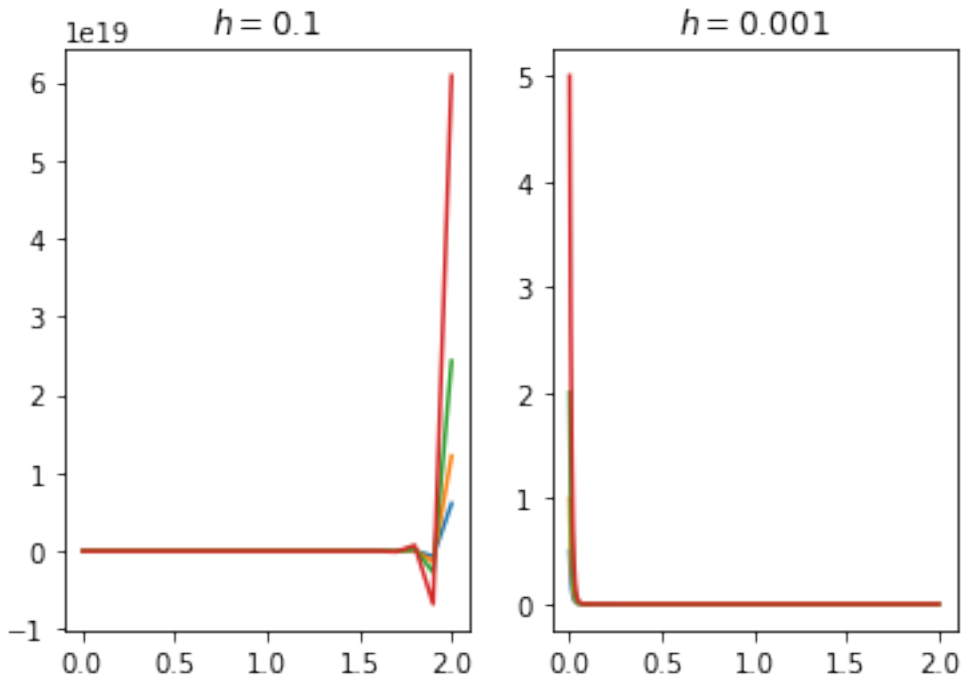
$$w_{n+1} = (1 - 100h)w_n.$$

Com um passo $h = 0.1$, a aproximação numérica $w_{n+1} = -9w_n$ cresce de acordo com $w_n = (-9)^n w_0$ e oscilla com uma amplitude crescente sem, de fato, aproximar a solução verdadeira. Ao se reduzir o passo para $h = 0.001$, a solução numérica transforma-se para $w_{n+1} = 0.9w_n$, de modo que $w_n = 0.9^n w_0$ significa um decaimento suave.

Reutilizemos nosso código do método de Euler para realizar este teste numérico.

```
from numpy import *

def euler_expl(t0,tf,y0,h,fun):
    """
    Resolve o PVI y' = f(t,y), t0 <= t <= tf, y(t0) = y0
    com passo h usando o metodo de Euler explicito.

    Entrada:
        t0  - tempo inicial
        tf  - tempo final
        y0  - condicao inicial
        h   - passo
        fun - funcao f(t,y) (anonima)

    Saida:
        t   - nos da malha numerica
        y   - solucao aproximada
    """

    n = round((tf - t0)/h + 1)
    t = linspace(t0,t0+(n-1)*h,n)
    y = linspace(t0,t0+(n-1)*h,n)
    y = zeros((n,))

    y[0] = y0

    for i in range(1,n):
        y[i] = y[i-1] + h*f(t[i-1],y[i-1])

    return (t,y)
```

```
import matplotlib.pyplot as plt
f = lambda t,y: -100*y

# h = 0.1
plt.subplot(121)
for y0 in [0.5,1,2,5]:
    (t,ynum) = euler_expl(0,2,y0,0.1,f)
    plt.plot(t,ynum)
    plt.title('$h=0.1$')

# h = 0.001
plt.subplot(122)
for y0 in [0.5,1,2,5]:
    (t,ynum) = euler_expl(0,2,y0,0.001,f)
    plt.plot(t,ynum)
    plt.title('$h=0.001$')
```



Exercício: Repita o mesmo experimento numérico anterior para o PVI

$$\begin{cases} y'(t) = \lambda(y - \text{sen}(t)) + \cos(t) \\ y(\pi/4) = 1/2 \end{cases}$$

para $\lambda = 2$ e $\lambda = -0.2$. Compare as soluções numéricas obtidas pelo método de Euler para $h = \pi/10$ e $h = \pi/20$ para cada valor de λ.

57.1.1 Região de estabilidade

Considere o PVI mais geral

$$\begin{cases} y'(t) = \lambda y(t) \\ y(0) = y_0 \end{cases},$$

em que $\lambda \in \mathbb{C}$. A EDO teste é estável, i.e. a taxa de crescimento da perturbação é limitada quando $\Re\{\lambda\} = \alpha < 0$. Neste, a solução numérica pelo método de Euler explícito corresponde ao processo

$$w_{n+1} = (1 + h\lambda)w_n$$

e a solução numérica será estável se, e somente se, o *fator de amplificação* for limitado por 1, ou seja,

$$|1 + h\lambda| \leq 1.$$

Assim, diz-se que o método de Euler será **estável** para esta EDO teste para valores de h que satisfizerem a condição acima (para λ real e negativo, $h \leq -2/\lambda$). O conjunto

$$\mathcal{S} = \{h\lambda \in \mathbb{C}; |1 + h\lambda| \leq 1\}$$

é chamado de *região de estabilidade* para o método de Euler e trata-se de um disco de raio unitário centrado em $(-1, 0)$ no plano de Argand-Gauss.

CHAPTER

FIFTYEIGHT

MÉTODO DE EULER IMPLÍCITO

A forma do *Método de Euler Implícito* (MEI) é similar àquela do MEE, exceto por uma característica distintiva. Em vez de a inclinação utilizada para avanço ser tomada em t_i, ela é tomada em t_{i+1}. Isto é, o processo numérico avança com a estimativa $f(t_{i+1}, y_{i+1})$ e não $f(t_i, y_i)$. O fato de usar esta inclinação ainda desconhecida é a razão de o método ser denominado "implícito". O esquema numérico resultante é o processo iterativo:

$$w_0 = \alpha$$
$$w_{i+1} = w_i + hf(t_{i+1}, w_{i+1}) \;\; i = 0, 1, \ldots, N-1.$$

Note, entretanto, que w_{i+1} aparece não apenas no lado esquerdo, mas também no lado direito da equaçõe. Esta incógnita nem sempre é obtenível de modo explícito, isto é, por isolamento. Quando este é o caso, f é linear ou uma função simples, mas, em geral, f é não-linear e dependente do valor futuro w_{i+1}.

CHAPTER

FIFTYNINE

CÓDIGO PREDITOR/CORRETOR

```
import matplotlib.pyplot as plt
```

```
# MEI: preditor/corretor

def euler_impl(t0,tf,y0,h,f,tol):

    n = round((tf - t0)/h) + 1
    t = linspace(t0,t0+(n-1)*h,n)
    y = zeros(n)
    y[0] = y0

    i = 1
    # iteracoes
    while i < n:

        # preditor (MEE)
        yf = y[i-1] + h*f(t[i-1],y[i-1])

        # iteracoes internas (maximo 10)
        count = 0
        diff = 1.0
        while diff > tol and count < 10:

            # corretor (MEI)
            yf2 = y[i-1] + h*f(t[i],yf)

            diff = abs(yf2-yf)
            yf = yf2
            count += 1

        if count >= 10:
            print('Nao convergindo apos 10 passos em t = {0:f}'.format(t[i]))

        y[i] = yf2
        i += 1

    return t,y
```

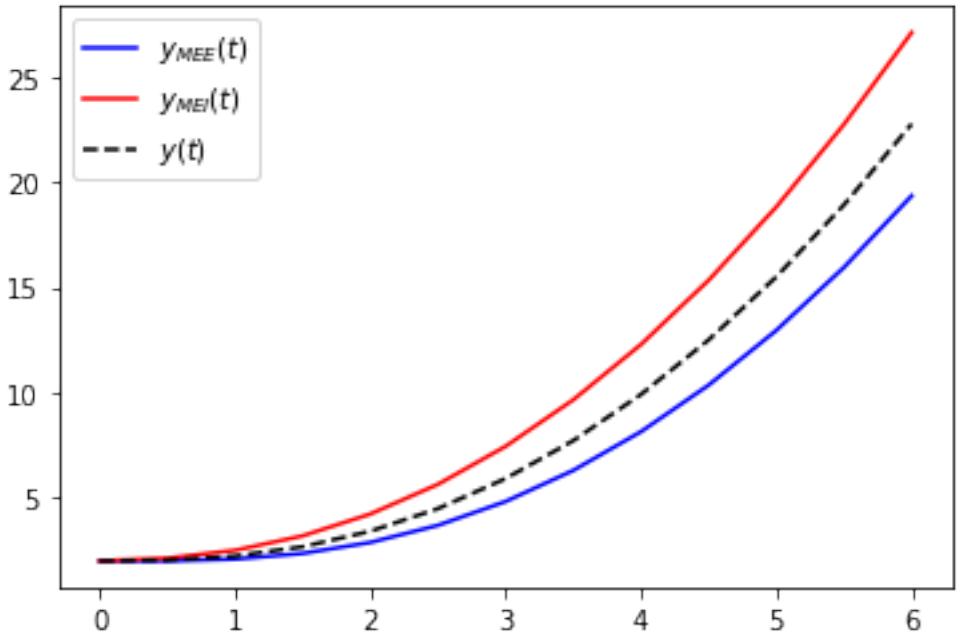
```
from numpy import linspace, zeros

def euler_expl(t0,tf,y0,h,fun):
    """
    Resolve o PVI y' = f(t,y), t0 <= t <= tf, y(t0) = y0
```

(continues on next page)

(continued from previous page)

```
    com passo h usando o metodo de Euler explicito.

    Entrada:
        t0  - tempo inicial
        tf  - tempo final
        y0  - condicao inicial
        h   - passo
        fun - funcao f(t,y) (anonima)

    Saida:
        t   - nos da malha numerica
        y   - solucao aproximada
    """

    n = round((tf - t0)/h) + 1
    t = linspace(t0,t0+(n-1)*h,n)
    y = linspace(t0,t0+(n-1)*h,n)
    y = zeros((n,))

    y[0] = y0

    for i in range(1,n):
        y[i] = y[i-1] + h*f(t[i-1],y[i-1])

    return (t,y)
```

```
from numpy import log

# define funcao
f = lambda t,y: (y + t**2 - 2)/(t+1)

# invoca metodo
t0 = 0.0
tf = 6.0
y0 = 2.0
h = 0.5
tol = 1e-3
t,y1 = euler_expl(t0,tf,y0,h,f)
t,y2 = euler_impl(t0,tf,y0,h,f,tol)


# plota funcoes
yex = t**2 + 2*t + 2 - 2*(t+1)*log(t+1)
plt.plot(t,y1,'b-',label='$y_{MEE}(t)$')
plt.plot(t,y2,'r-',label='$y_{MEI}(t)$')
plt.plot(t,yex,'k--',label='$y(t)$')
plt.legend();
```


$y_{MEE}(t)$
$y_{MEI}(t)$
$y(t)$
25
20
15
10
5
0
1
2
3
4
5
6

CHAPTER

SIXTY

MÉTODO TRAPEZOIDAL

```
def trapezoidal(t0,tf,y0,h,f,tol):

    n = round((tf - t0)/h) + 1
    t = linspace(t0,t0+(n-1)*h,n)
    y = zeros(n)
    y[0] = y0

    i = 1
    # iteracoes
    while i < n:

        # f(tn,yn)
        fyt = f(t[i-1],y[i-1])

        # Euler
        yt1 = y[i-1] + h*fyt

        # iteracoes internas (maximo 10)
        count = 0
        diff = 1.0
        while diff > tol and count < 10:

            # corretor (Trapezoidal)
            yt2 = y[i-1] + 0.5*h*( fyt + f(t[i],yt1) )
            diff = abs(yt2-yt1)
            yt1 = yt2
            count += 1

        if count >= 10:
            print('Nao convergindo apos 10 passos em t = {0:f}'.format(t[i]))

        y[i] = yt2
        i += 1

    return t,y
```

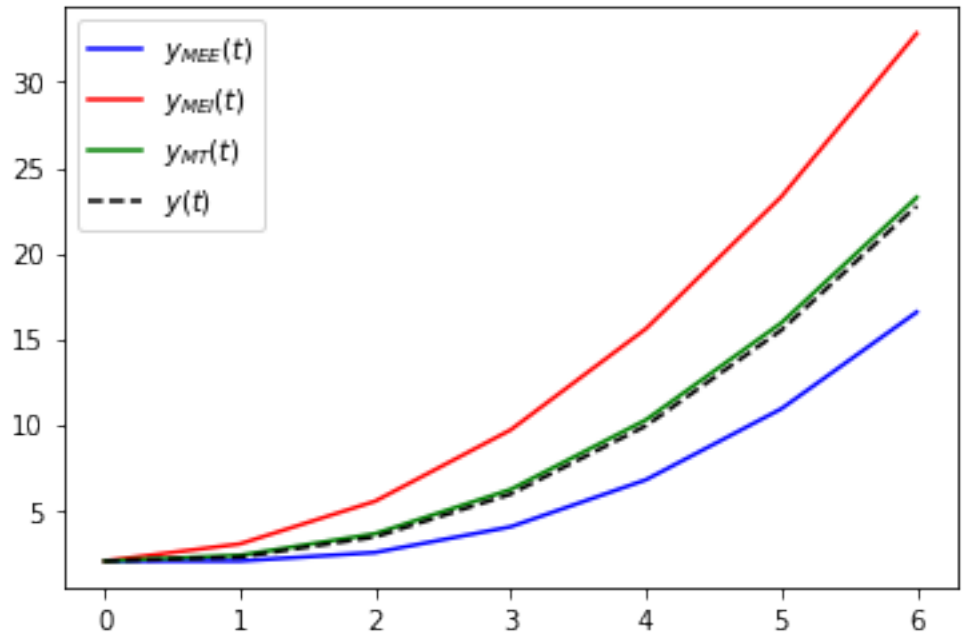
```
# define funcao
f = lambda t,y: (y + t**2 - 2)/(t+1)

# invoca metodo
t0 = 0.0
tf = 6.0
y0 = 2.0
```

(continues on next page)

(continued from previous page)

```
h = 1.0
tol = 1e-3
t,y1 = euler_expl(t0,tf,y0,h,f)
t,y2 = euler_impl(t0,tf,y0,h,f,tol)
t,y3 = trapezoidal(t0,tf,y0,h,f,tol)


# plota funcoes
yex = t**2 + 2*t + 2 - 2*(t+1)*log(t+1)
plt.plot(t,y1,'b-',label='$y_{MEE}(t)$')
plt.plot(t,y2,'r-',label='$y_{MEI}(t)$')
plt.plot(t,y3,'g-',label='$y_{MT}(t)$')
plt.plot(t,yex,'k--',label='$y(t)$')
plt.legend();
```

```
Nao convergindo apos 10 passos em t = 1.000000
```



CHAPTER

SIXTYONE

MÉTODOS DE ADAMS-BASHFORT

```
%matplotlib inline
from numpy import *
from matplotlib.pyplot import *
```

```
# Metodo de Adams-Bashfort de 2a. ordem

def adams_bashfort_2nd_order(t0,tf,y0,h,fun):
    '''
    Resolve o PVI y' = f(t,y), t0 <= t <= b, y(t0)=y0
    usando a formula de Adams-Bashfort de ordem 2
    com passo h. O metodo de Euler eh usado para
    obter y1. A funcao f(t,y) deve ser definida
    pelo usuario.

    Saida:

    A rotina AB2 retorna dois vetores, t e y,
    contendo, nesta orde, os pontos nodais e
    a solucao numerica.
    '''

    # malha numerica
    n = round((tf - t0)/h) + 1
    t = linspace(t0,t0+(n-1)*h,n)
    y = np.zeros(n)

    y[0] = y0 # condicao inicial

    f1 = fun(t[0],y[0]) # f(t_i,y_i)
    y[1] = y[0] + h*f1 # Euler

    for i in range(2,n):
        f2 = fun(t[i-1],y[i-1]) # f(t_i-1,y_i-1)
        y[i] = y[i-1] + h*(3*f2 - f1)/2 # esquema AB2
        f1 = f2 # atualiza

    return t,y


def tab_erro_rel(t,y_n,y_e):

    # erro relativo
    e_r = abs(y_n - y_e)/abs(y_e)
```

(continues on next page)

(continued from previous page)

```
    print('i \t t \t y_ex \t y_num \t e_r')
    for i in range(len(e_r)):

        if i % 10 == 0:
            print('{0:d} \t {1:f} \t {2:f} \t {3:f} \t {4:e} \n'.format(i,t[i],y_e[i],
 ↪y_n[i],e_r[i]))

def plot_fig(t,y_n,y_e,h):
    plot(t,y_e,'r-')
    plot(t,y_n,'bo')
    title('Adams-Bashfort, 2a. ordem: $h=' + str(h) + '$')
    legend(['$y_{exata}$','$y_{num}$'])
```

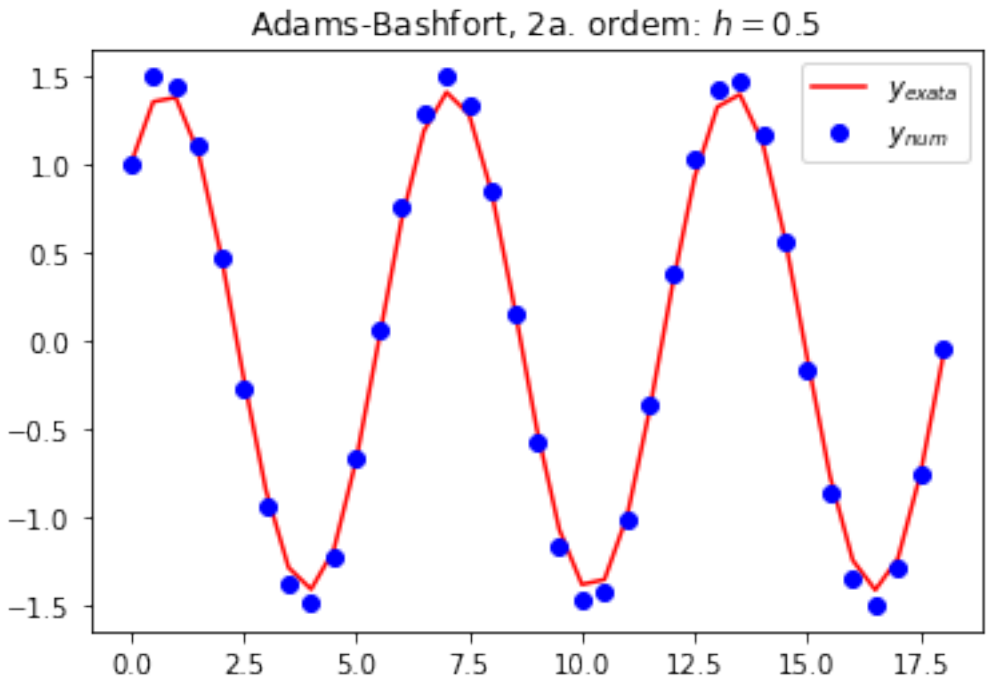
Exemplo: Use o esquema de Adams-Bashfort de 2a. ordem para resolver o PVI.

$$\begin{cases} y'(t) = -y(t) + 2\cos(t) \\ y(0) = 1 \\ 0 \leq t \leq 18 \end{cases}$$

Solução exata: $y(t) = \text{sen}(t) + \cos(t)$

```
# define funcao
f = lambda t,y: -y + 2*cos(t)

# parametros
t0 = 0.0
tf = 18.0
y0 = 1.0
h = 0.5

# solucao numerica
t,y_num = adams_bashfort_2nd_order(t0,tf,y0,h,f)

# solucao exata
y_ex = sin(t) + cos(t)

plot_fig(t,y_num,y_ex,h)

tab_erro_rel(t,y_num,y_ex)
```

```
i          t          y_ex          y_num          e_r
0          0.000000          1.000000          1.000000          0.000000e+00

10          5.000000          -0.675262          -0.670163          7.550607e-03

20          10.000000          -1.383093          -1.477388          6.817733e-02

30          15.000000          -0.109400          -0.167908          5.348066e-01
```



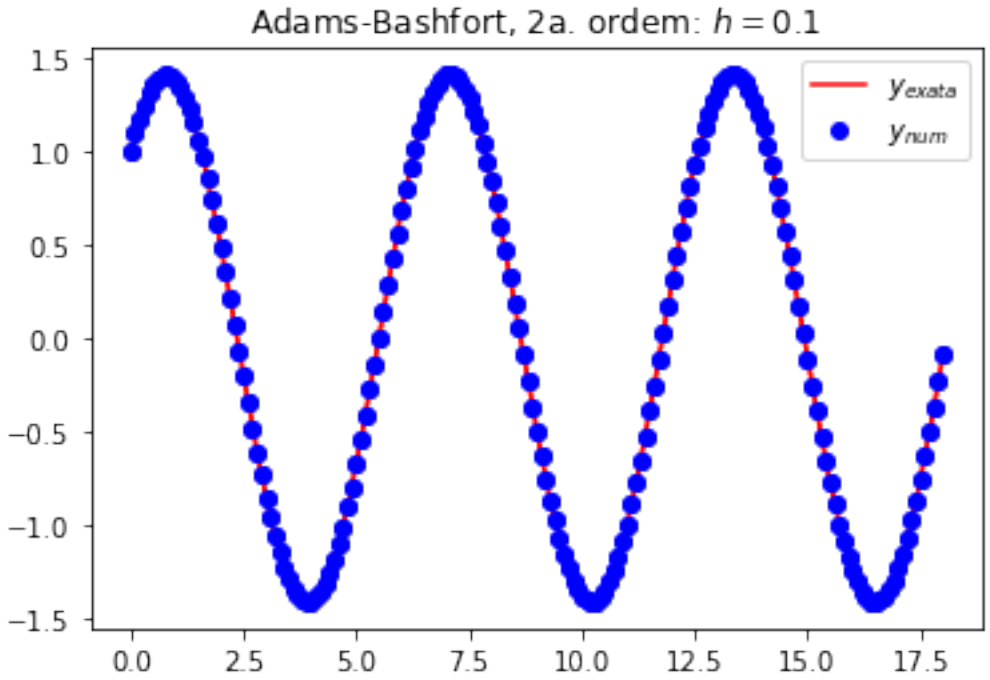

```
h = 0.1

# solucao numerica
t,y_num = adams_bashfort_2nd_order(t0,tf,y0,h,f)

# solucao exata
y_ex = sin(t) + cos(t)

plot(t,y_ex,'r-')
plot(t,y_num,'bo')
title('Adams-Bashfort, 2a. ordem: $h=0.5$')
legend(['$y_{exata}$','$y_{num}$'])

tab_erro_rel(t,y_num,y_ex)

plot_fig(t,y_num,y_ex,h)
```

```
i          t          y_ex          y_num          e_r
0          0.000000          1.000000          1.000000          0.000000e+00

10          1.000000          1.381773          1.384518          1.986208e-03

20          2.000000          0.493151          0.491752          2.835038e-03

30          3.000000          -0.848872          -0.852907          4.752486e-03

40          4.000000          -1.410446          -1.413326          2.041670e-03

50          5.000000          -0.675262          -0.674309          1.410789e-03

60          6.000000          0.680755          0.684675          5.758689e-03
```

(continues on next page)

(continued from previous page)

70	7.000000	1.410889	1.414177	2.330243e-03
80	8.000000	0.843858	0.843492	4.337392e-04
90	9.000000	-0.499012	-0.502694	7.379922e-03
100	10.000000	-1.383093	-1.386706	2.612468e-03
110	11.000000	-0.995565	-0.995786	2.227766e-04
120	12.000000	0.307281	0.310655	1.097903e-02
130	13.000000	1.327614	1.331481	2.913030e-03
140	14.000000	1.127345	1.128150	7.144782e-04
150	15.000000	-0.109400	-0.112397	2.739478e-02
160	16.000000	-1.245563	-1.249607	3.246745e-03
170	17.000000	-1.236561	-1.237934	1.110338e-03
180	18.000000	-0.090671	-0.088110	2.823799e-02

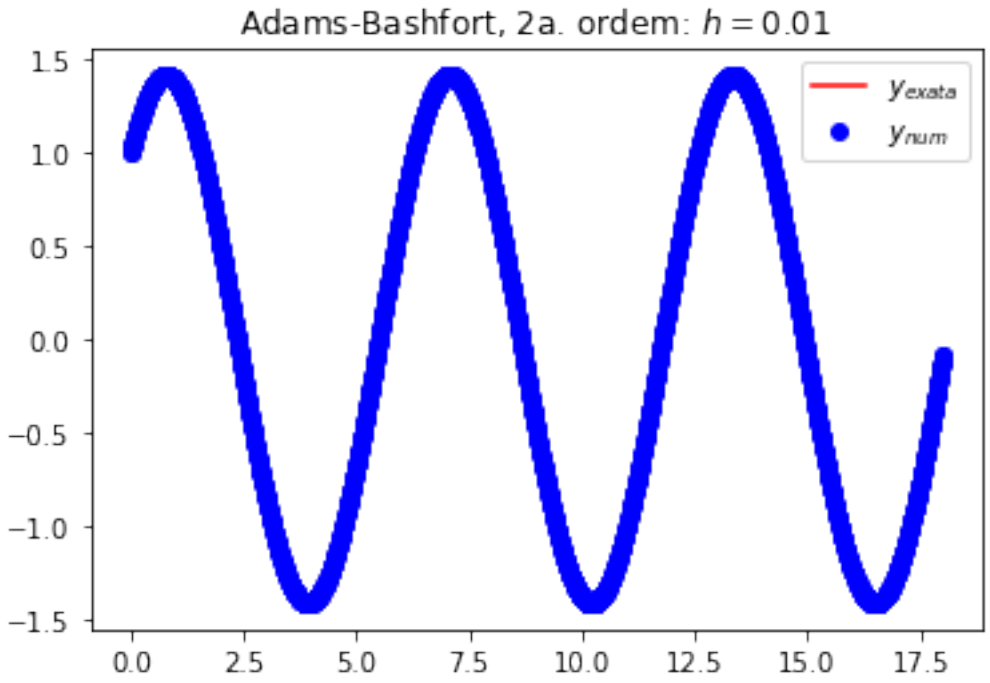



```
h = 0.01

# solucao numerica
t,y_num = adams_bashfort_2nd_order(t0,tf,y0,h,f)

# solucao exata
y_ex = sin(t) + cos(t)
```

(continues on next page)

(continued from previous page)

```
plot(t,y_ex,'r-')
plot(t,y_num,'bo')
title('Adams-Bashfort, 2a. ordem: $h=0.1$')
legend(['$y_{exata}$','$y_{num}$'])

#tab_erro_rel(t,y_num,y_ex)

plot_fig(t,y_num,y_ex,h)
```



CHAPTER

SIXTYTWO

EDOS DE ORDEM SUPERIOR (REDUÇÃO DE ORDEM)

EDOs de ordem superior podem ser reescritas como sistemas de EDOs de primeira ordem. A fim de verificarmos como isto pode ser feito, consideremos, inicialmente o caso geral de uma EDO de segunda ordem.

$$\begin{cases} y''(t) = f(t, y, y') \\ y(t_0) = y_0 \\ y'(t_0) = y'_0 \end{cases}$$

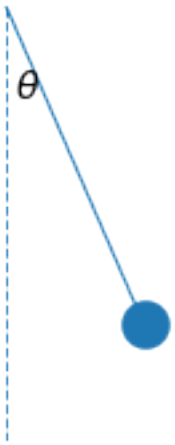
Se fizermos $y_1(t) = y(t)$ e $y_2(t) = y'(t)$, o PVI acima pode ser reformulado como um sistema de duas EDOs de primeira ordem como:

$$\begin{cases} y'_1(t) = y_2(t) \\ y'_2(t) = f(t, y_1, y_2) \\ y_1(t_0) = y_0 \\ y_2(t_0) = y'_0 \end{cases}$$

62.1 Movimento pendular simples

Como exemplo (simplificado) prático derivado de uma EDO de segunda ordem, temos o movimento de um pêndulo submetido apenas à força gravitacional como mostra a figura abaixo. O pêndulo de massa m fica pendurado por uma corda de comprimento l e se move em relação ao eixo $\theta = 0$. O movimento é completamente descrito com a especificação da posição inicial $\theta(0) = \theta_0$ e a velocidade angular inicial $\theta'(0) = \theta'_0$.

```
import numpy as np
import matplotlib.pyplot as plt

#plt.autoscale(enable=True, axis='x', tight=True)
plt.figure(figsize=(3,3))

theta = np.pi/10
x = np.linspace(0,1,10)
plt.axvline(x=0.5,ymin=0,ymax=1.0,linestyle='--',linewidth=1.0)
plt.plot([0.5,0.5 + np.sin(theta)],[1.0,1.0-np.cos(theta)],linewidth=1.0)
plt.scatter([0.5 + np.sin(theta)],[1.0-np.cos(theta)],s=300)
plt.box(False); locs, labels = plt.xticks(); plt.tick_params(axis='both',width=0.0,
 ↪labelleft=False,labelbottom=False)
x1,x2 = plt.xlim([0,1])
y1,y2 = plt.ylim([-0.3,1])

plt.annotate('$\\theta$',xy=(0.52,0.72),fontsize=14);
```



Uma vez que o comprimento do arco realizado pelo pêndulo é $s = l\theta$ e a força-peso tangente ao arco oposta ao movimento é determinada em função de θ por $-mg\text{sen}(\theta)$, a segunda lei de Newton diz-nos que

$$m\frac{d^2s}{dt^2} = -mg\text{sen}(\theta) \Rightarrow \frac{d^2\theta}{dt^2} + \frac{g}{l}\text{sen}(\theta) = 0$$

Tomando a EDO acima juntamente com as condições iniciais e valendo-se das substituições $y_1(t) = \theta(t)$ e $y_2(t) = \theta'(t)$, obtemos um sistema de EDOs de primeira ordem que nos conduz ao seguinte PVI: $$\begin{cases} y_1'(t) = y_2(t) \\ y_2'(t) = -\frac{g}{l}\text{sen}(\theta) \\ y_1(0) = \theta(0) = \theta_0 \\ y_2(0) = \theta'(0) = \theta_0' \end{cases}$$

62.2 Hipótese de ângulos pequenos

No movimento pendular, se θ for considerado muito pequeno, podemos dizer que $\theta \approx \text{sen}(\theta)$. Logo, a EDO de segunda ordem reduz-se a

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\theta = 0,$$

de maneira que, analogamente ao caso anterior, temos um novo sistema levemente modificado na segunda equação: $$\begin{cases} y_1'(t) = y_2(t) \\ y_2'(t) = -\frac{g}{l}y_1(t) \\ y_1(0) = \theta(0) = \theta_0 \\ y_2(0) = \theta'(0) = \theta_0' \end{cases}$$

Vale dizer que a solução analítica para este caso é dada por $\theta(t) = \theta_0 \cos\left(\sqrt{\frac{g}{l}}t\right)$.

62.3 Caso geral de ordem m

De modo similar, uma EDO de ordem m pode gerar um sistema de EDOs de ordem 1. Considerando o PVI generalizado
$\$\begin{cases} \dfrac{d^m y(t)}{dt^m} = f(t, y, y', \dots, y^{(m-1)}) \\ y(t_0) = y_0, \;\; y'(t_0) = y'_0, \dots y^{(m-1)}(t_0) = y_0^{(m-1)}, \end{cases}$ $ usamos as substituições

$$
\begin{aligned}
y_1(t) &= y(t) \quad (1)\\
y_2(t) &= y'(t) \quad (2)\\
&\quad (3)\\
y_m(t) &= y^{(m-1)}(t) \quad (4)\\
&\quad (5)
\end{aligned}
$$

para reformular o PVI original no sistema:

$$
\begin{cases}
y'_1(t) = y_2(t) \\
y'_2(t) = y_3(t) \\
\vdots \\
y'_{m-1}(t) = y_m(t) \\
y'_m(t) = f(t, y_1, \dots, y_m) \\
y_1(t_0) = y_0 \\
y_2(t_0) = y'_0 \\
\vdots \\
y_m(t_0) = y_0^{(m-1)}
\end{cases}
$$

Exemplo: O PVI

$$
\begin{cases}
y'''(t) + 3y''(t) + 3y'(t) + y(t) = -4\text{sen}(t) \\
y(0) = y'(0) = 1, y''(0) = -1
\end{cases}
$$

pode ser reformulado no sistema

$$
\begin{cases}
y'_1(t) = y_2(t) \\
y'_2(t) = y_3(t) \\
y'_3(t) = -y_1(t) - 3y_2(t) - 3y_3(t) - 4\text{sen}(t) \\
y_1(0) = 1 \\
y_2(0) = 1 \\
y_3(0) = -1.
\end{cases}
$$

Exercício resolvido: utilizando os parâmetros $g = 9.867\, m/s^2$, $\theta_0 = \pi/25$ e $\theta'_0 = 0.5\, m/s$, resolva numericamente o sistema de EDOs do movimento pendular sob a hipótese de ângulos pequenos.

Para resolver o exercício, utilizaremos o método de Euler que implementamos anteriormente. Basta definirmos uma nova função `fsys` e os parâmetros necessários para o funcionamento da função `euler_sys`.

```
from numpy import *

def euler_sys(t0,tf,y0,h,f):
    """
    Resolve o PVI de um sistema de EDOs escalares
    y' = f(t,y), t0 <= t <= b, y(t0)=y0 pelo metodo de Euler com tamanho de passo h.
```

(continues on next page)

(continued from previous page)

```
    O usuario deve fornecer um vetor f contendo as funcoes a serem avaliadas como
    membro direito.

    Entrada:
        t0  - tempo inicial
        tf  - tempo final
        y0  - condicao inicial
        h   - passo
        f   - vetor de funcoes f(t,y) (anonima)

    Saída:
        t   - vetor contendo os valores nodais t[i], i = 1,2,...,n
        Y   - matriz de dimensoes n x m, com m sendo o numero de EDOs
              (a i-esima linha y[i,:] traz as estimativas de todas
               as funcoes y_j no tempo t[i])
    """

    m = y0.size
    n = round((tf - t0)/h + 1)
    t = linspace(t0,t0+(n-1)*h,n)
    Y = zeros((n,m))

    Y[0,:] = y0


    for i in range(1,n):
        Y[i,:] = Y[i-1,:] + h*f(t[i-1],Y[i-1,:])

    return t,Y
```

```
# define funcao f(t,y)

def params(g,l):
    return g,l

def fsys(t,y):

    aux = params(g,l)
    # considera g e l como parâmetros
    F = array([y[1],-aux[0]/aux[1]*y[0]])

    return F
```

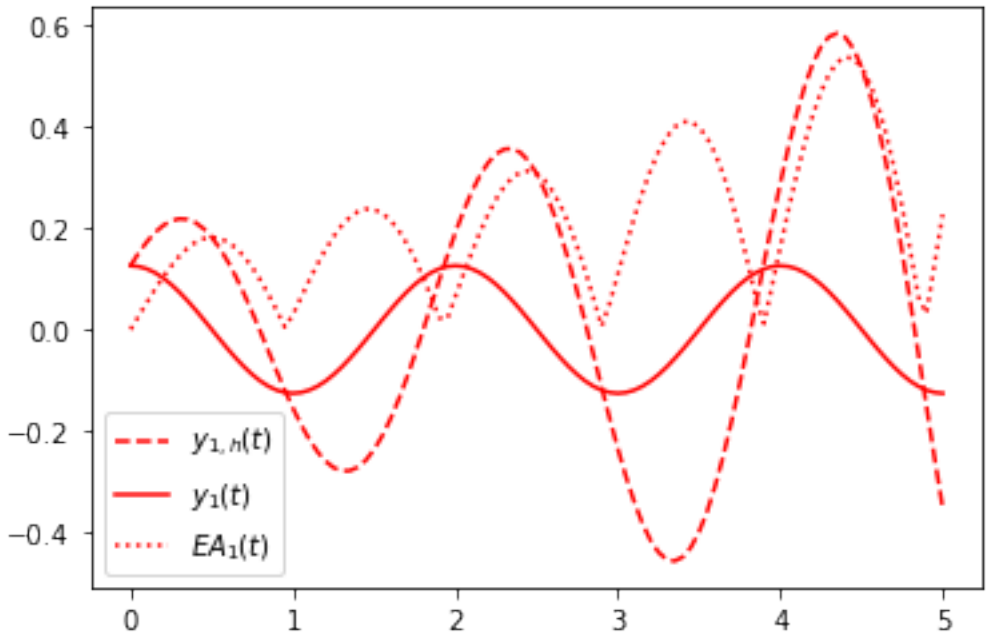
```
from matplotlib.pyplot import plot,legend


# parametros
t0,tf = 0,5
h = 0.05

theta0 = np.pi/25
dtheta0 = 0.5
y0 = array([theta0,dtheta0])
g, l = 9.867, 1.0

# solucao numerica do sistema
```

(continues on next page)

(continued from previous page)

```
t,Y = euler_sys(t0,tf,y0,h,fsys)
y1_num = Y[:,0]
y2_num = Y[:,1]

# solucao analitica
y1_an = theta0*np.cos(np.sqrt(g/l)*t)

# plotagem
plot(t,y1_num,'r--',label='$y_{1,h}(t)$')
#plot(t,y2_num,'b--',label='$y_{2,h}(t)$')

plot(t,y1_an,'r',label='$y_1(t)$')
#plot(t,y2_an,'b',label='$y_2(t)$')

# erro absoluto
plot(t,abs(y1_an - y1_num),'r:',label='$EA_1(t)$')
#plot(t,abs(y2_an - y2_num),'b:',label='$EA_2(t)$')

legend(loc=3,fontsize=10)
```

```
<matplotlib.legend.Legend at 0x7fb2d222e100>
```



Exercício: Converta as EDOs de ordem superior abaixo para sistemas de EDOs escalares.

1.

$$\begin{cases} y'''(t) + 4y''(t) + 5y'(t) + 2y(t) = 2t^2 + 10t + 8 \\ y(0) = 1, y'(0) = -1, y''(0) = 3 \end{cases}$$

Solução analítica: $y(t) = \exp(-t) + t^2$.

1.

$$\begin{cases} y''(t) + 4y'(t) + 13y(t) = 40\cos(t) \\ y(0) = 3, y'(0) = 4, y''(0) = 3 \end{cases}$$

Solução analítica: $y(t) = 3\cos(t) + \text{sen}(t) + \exp(-2t)\text{sen}(3t)$.

CHAPTER

SIXTYTHREE

PROBLEMAS

Problema: Converta o seguinte sistema de EDOs de 2a. ordem para um sistema maior de EDOs de primeira ordem. Este sistema surge do estudo da atração gravitacional de uma massa por outra:

$$\begin{cases} x''(t) = \dfrac{-cx(t)}{r(t)^3} \\ y''(t) = \dfrac{-cy(t)}{r(t)^3} \\ z''(t) = \dfrac{-cz(t)}{r(t)^3}, \end{cases}$$

onde c é uma constante positiva e $r(t) = [x(t)^2 + y(t)^2 + z(t)^2]^{1/2}$, com t denotando o tempo.

Problema: Resolva o problema do pêndulo numericamente pelo método de Euler. Considere $l = 1$ e $g = 32.2\, ps/s^2$. Como valores iniciais, escolha $0 < \theta(0) \leq \pi/2$ e $\theta'(0) = 1$. Experimente vários valores de h de modo a obter erros pequenos. Plote os gráficos $t\, vs.\, \theta(t)$, $t\, vs. \theta'(t)$ e $\theta(t)\, vs. \theta'(t)$. O movimento parece periódico no tempo? (Use a hipótese de pequenos ângulos.)

CHAPTER

SIXTYFOUR

SISTEMAS DE EDOS

Embora EDOs escalares sejam responsáveis por descrever uma vasta quantidade de fenômenos naturais, muitas aplicações são melhor descritas através de um sistema de EDOs escalares ou de ordem superior. O tratamento numérico de EDOs de alta ordem baseia-se em uma conversão a sistemas de primeira ordem. Vejamos como escrever a forma geral de um sistema de duas EDOS de primeira ordem:

$$\begin{cases} y_1'(t) & = \\ f_1(t, y_1(t), y_2(t)) & \\ y_2'(t) & = \\ f_2(t, y_1(t), y_2(t)) & \end{cases}$$

As funções $f_1(t, z_1, z_2)$ e $f_2(t, z_1, z_2)$ definem EDOs e as incógnitas são as funções $y_1(t)$ e $y_2(t)$. O problema de valor inicial então consiste de resolver o sistema anterior, sujeito às condições iniciais $y_1(t_0) = y_{1,0}$ e $y_2(t_0) = y_{2,0}$.

Exemplo: O PVI $\$ \begin{cases} y_1'(t) & = \\ y_1(t) - 2y_2(t) + 4\cos(t) - 2\text{sen}(t) & \\ y_2'(t) & = \\ 3y_1(t) - 4y_2(t) + 5\cos(t) - 5\text{sen}(t) & \\ y_1(0) & = \\ 1 & \\ y_2(0) & = \\ 2 & \end{cases} \$$

tem por solução as funções $y_1(t) = \cos(t) + \text{sen}(t)$ e $y_2(t) = 2\cos(t)$.

Exemplo (Modelo de Lotka-Volterra): O sistema de EDOs dado por $\$ \begin{cases} y_1'(t) & = \\ Ay_1(t)[1 - By_2(t)] & \\ y_2'(t) & = \\ Cy_2(t)[Dy_1(t) - 1] & \\ y_1(0) & = \\ y_{1,0} & \\ y_2(0) & = \\ y_{2,0} & \end{cases} \$$

com constantes A, B, C, $D > 0$ é conhecido como modelo *predador-presa*. A variável t é o tempo, $y_1(t)$ o número de presas no tempo t (e.g. coelhos) e $y_2(t)$ o número de predadores (e.g. raposas). Para apenas um tipo de predador e de presa, este modelo é uma aproximação razoável da realidade.

64.1 Sistema com m EDOs de primeira ordem

Um PVI com m EDOs escalares é dado por

$$\begin{cases} y_1'(t) & = \\ f_1(t, y_1(t), y_2(t), \dots, y_m(t)) & \\ y_2'(t) & = \\ f_2(t, y_1(t), y_2(t), \dots, y_m(t)) & \\ \vdots & \vdots \\ \vdots & \\ y_m'(t) & = \\ f_m(t, y_1(t), y_2(t), \dots, y_m(t)) & \\ y_1(t_0) & = \\ y_{1,0} & \\ y_2(t_0) & = \\ y_{2,0} & \\ \vdots & \vdots \\ \vdots & \\ y_m(t_0) & = \\ y_{m,0} & \\ t_0 \leq t \leq b, & \end{cases}$$

cuja solução procurada são as funções $y_1(t), y_2(t), \dots, y_m(t)$. Entretanto, a forma anterior não é computacionalmente adequada para se trabalhar. Assim, simplificamo-na para a forma vetorial

$$\begin{cases} \mathbf{y}'(t) & = \\ \mathbf{f}(t, \mathbf{y}(t)) & \\ \mathbf{y}(t_0) & = \\ \mathbf{y}_0, & \end{cases}$$

onde

$$\mathbf{y}(t) = \begin{bmatrix} y_1(t) \\ \vdots \\ y_m(t) \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} y_{1,0} \\ \vdots \\ y_{m,0} \end{bmatrix}, \quad \mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} f_1(t, y_1, y_2, \dots, y_m) \\ \vdots \\ f_m(t, y_1, y_2, \dots, y_m) \end{bmatrix},$$

com $\mathbf{y} = [y_1, y_2, \dots, y_m]^T$.

Exemplo: O primeiro PVI pode ser reescrito como

$$\begin{cases} \mathbf{y}'(t) & = \\ \mathbf{A}\mathbf{y}(t) + \mathbf{G}(t) & \\ \mathbf{y}(0) & = \\ \mathbf{y}_0, & \end{cases}$$

com

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 1 & -2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{G}(t) = \begin{bmatrix} 4\cos(t) - 2\text{sen}(t) \\ 5\cos(t) - 5\text{sen}(t) \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

64.2 Métodos numéricos para sistemas

Tanto o método de Euler quanto outros métodos numéricos podem ser usados de forma similar para sistemas de EDOs quando aplicados a cada EDO individual. Para isto, lança-se mão da forma matriz/vetor, em que a derivação é essencialmente a mesma feita para o caso individual.

Lembremos que a série de Taylor desenvolvida para o método de Euler é dada por:

$$y_j(t_{n+1}) = y_j(t_n) + hy_j'(t_n) + \frac{h^2}{2}y_j''(\xi_{n,j}), \;\; t_n \leq \xi_{n,j} \leq t_{n+1}, \quad j = 1, \ldots, m$$

para m EDOs. Desprezando-se os termos de erro, o método de Euler em forma vetorial é escrito como

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n), \quad \mathbf{y}_0 = \mathbf{y}(0)$$

64.2.1 Implementação computacional

O seguinte código é uma implementação do método de Euler para resolver sistemas de EDOs. O usuário necessita especificar uma função adicional para determinar as EDOs como componentes de um vetor. Isto é feito pela função

```
def fsys(t,y):
    (...)
    return F
```

```
from numpy import *

def euler_sys(t0,tf,y0,h,f):
    """
    Resolve o PVI de um sistema de EDOs escalares
    y' = f(t,y), t0 <= t <= b, y(t0)=y0 pelo metodo de Euler com tamanho de passo h.
    O usuario deve fornecer um vetor f contendo as funcoes a serem avaliadas como
    membro direito.

    Entrada:
        t0  - tempo inicial
        tf  - tempo final
        y0  - condicao inicial
        h   - passo
        f   - vetor de funcoes f(t,y) (anonima)

    Saída:
        t   - vetor contendo os valores nodais t[i], i = 1,2,...,n
        Y   - matriz de dimensoes n x m, com m sendo o numero de EDOs
              (a i-esima linha y[i,:] traz as estimativas de todas
               as funcoes y_j no tempo t[i])
    """

    m = y0.size
    n = round((tf - t0)/h + 1)
    t = linspace(t0,t0+(n-1)*h,n)
    Y = zeros((n,m))

    Y[0,:] = y0


    for i in range(1,n):
```

(continues on next page)

(continued from previous page)

```
        Y[i,:] = Y[i-1,:] + h*f(t[i-1],Y[i-1,:])

    return t,Y
```

Exercício resolvido: Resolva o PVI abaixo pelo método de Euler:

$$\begin{cases} \mathbf{y}'(t) & = \\ \mathbf{A}\mathbf{y}(t) + \mathbf{G}(t) & \\ \mathbf{y}(0) & = \\ \mathbf{y}_0, & \end{cases}$$

com

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 1 & -2 \\ 3 & -4 \end{bmatrix}, \quad \mathbf{G}(t) = \begin{bmatrix} 4\cos(t) - 2\text{sen}(t) \\ 5\cos(t) - 5\text{sen}(t) \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

Resolver numericamente pelo método de Euler para $0 < t \leq 5$ e $h = 0.5$.

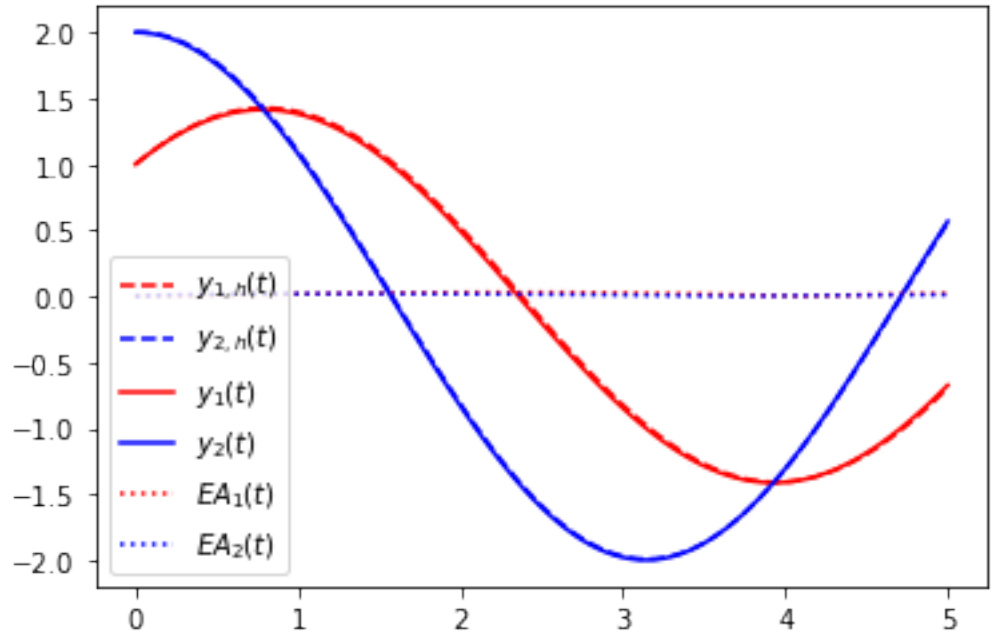
```
# define funcao vetorial f(t,y)
def fsys(t,y):
    """
    Função definida pelo usuario para montar vetor das m funcoes fm(t,(y1,...,ym))
    Em cada componente prescrevemos a funcao da EDO correspondente no sistema
    de m EDOs.

    Isto e,

    F = [f1(t,y1,...,ym),f2(t,y1,...,ym),...,fm(t,y1,...,ym)]^T

    Neste exemplo, o sistema possui 2 EDOs, com:

    f1(t,y1,y2) = y1 - 2y2 + 4cos(t) - 2sin(t)
    f2(t,y1,y2) = 3y1 - 4y2 + 5cos(t) - 5sin(t)
    """
    F = array([y[0]-2*y[1]+4*cos(t)-2*sin(t),3*y[0]-4*y[1]+5*cos(t)-5*sin(t)])

    return F
```

```
from matplotlib.pyplot import plot,legend


# parâmetros
t0,tf = 0,5
h = 0.05
y0 = array([1,2])

# solução numérica do sistema
t,Y = euler_sys(t0,tf,y0,h,fsys)
y1_num = Y[:,0]
y2_num = Y[:,1]

# solução analítica
y1_an = cos(t) + sin(t)
y2_an = 2*cos(t)

# plotagem
```

(continues on next page)

(continued from previous page)

```
plot(t,y1_num,'r--',label='$y_{1,h}(t)$')
plot(t,y2_num,'b--',label='$y_{2,h}(t)$')

plot(t,y1_an,'r',label='$y_1(t)$')
plot(t,y2_an,'b',label='$y_2(t)$')

# erro absoluto
plot(t,abs(y1_an - y1_num),'r:',label='$EA_1(t)$')
plot(t,abs(y2_an - y2_num),'b:',label='$EA_2(t)$')

legend(loc=3,fontsize=10);
```



Exercício: Considere o PVI

$$\begin{cases} \mathbf{y}'(t) & = \\ \mathbf{A}\mathbf{y}(t) + \mathbf{G}(t) & \\ \mathbf{y}(0) & = \\ \mathbf{y}_0, & \end{cases}$$

com

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix}, \quad \mathbf{G}(t) = \begin{bmatrix} -2\exp(-t) + 2 \\ -2\exp(-t) + 1 \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

para $t \in (0, t_f]$ e $0 < h < 1$ convenientemente escolhidos. Verifique que a sua solução analítica é dada por $\mathbf{y} = [\exp(-t), 1]^T$, resolva-o numericamente, plote os gráficos das curvas da solução numérica e do *erro relativo* para cada uma. Compare os gráficos com aqueles das curvas da solução analítica.

64.3 Problemas

Problema: considere o modelo de Lotka-Volterra com os parâmetros $A = 4$, $B = 1$, $C = 3$ e $D = 1$. Usando o método de Euler, resolva-o para $0 \leq t \leq 5$. Use passos de $h = 0.001$, 0.0005 e 0.00025 e a condição inicial values $y_1(0) = 3$, $y_2(0) = 5$. Plote as funções y_1 e y_2 em função de t e, depois, plote o gráfico de y_1 *versus* y_2. Comente os resultados.

Problema: Considere o esquema numérico

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n))]$$

Tente adaptá-lo para resolver o PVI do **Exercício resolvido**.

CHAPTER

SIXTYFIVE

MÉTODO DAS LINHAS

```
from numpy import *
import sympy as sy
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

O código abaixo usa o método de Euler explícito para resolver a EDP do calor unidimensional.

```
def met_linhas_euler(d0,d1,f,G,t0,T,h,m):
    """
    Usa o metodo das linhas para resolver
    ut = uxx + G(x,t), 0 < x < 1, 0 < t < T
    com C.C.
    u(0,t) = d0(t), u(1,t) = d1(t)
    e C.I.
    u(x,0) = f(x).

    Usa o metodo de Euler para resolver o
    sistema de EDOs. Para a discretizacao,
    usa passo espacial delta = 1/m e passo
    temporal h. Para estabilidade numérica,
    usa h = 1/(2*m**2) ou menor.
    """
    x = linspace(0,1,m+1)
    delta = 1/m
    delta_sqr = delta**2

    N = round((T - t0)/h) + 1
    t = linspace(t0,(N-1)*h,N)

    # inicializa u
    u = zeros((m+1,N))

    # construir vetores, se funcao constante
    if isinstance(d0,float):
        d0 = d0*ones(N)
        u[0,:] = d0
    else:
        u[0,:] = d0(t)

    if isinstance(d1,float):
        d1 = d1*ones(N)
        u[m,:] = d1
    else:
        u[m,:] = d1(t)
```

(continues on next page)

(continued from previous page)

```
    if isinstance(f,float):
        f = f*ones(N)
        u[:,0] = f
    else:
        u[:,0] = f(x)

    # Resolve para u usando Euler
    for n in range(1,N):
        g = G(x[1:m-1],t[n-1])
        u[1:m,n] = u[1:m,n-1] + (h/delta_sqr)*(u[0:(m-1),n])
        - 2*u[1:m-1,n-1] + u[2:m,n-1] + h*g

    return x,t,u
```

A solução exata para a EDP é $u(x,t) = \exp(-0.1t)\text{sen}(\pi x)$. A partir desta função obteremos todos os demais termos da EDP. Abaixo, encontraremos as derivadas parciais em relação ao tempo e ao espaço.

```
# u(x,t) = exp(-0.1*t)*sin(pi*x)

# variaveis simbólicas
xsym,tsym = sy.symbols('x,t')

# u(x,t)
u = sy.exp(-0.1*tsym)*sy.sin(sy.pi*xsym)

# dudt
dudt = sy.diff(u,tsym)

# d2udx2
d2udx2 = sy.diff(u,xsym,2)

# G(x,t)
Gxt = dudt - d2udx2

print(Gxt)
```

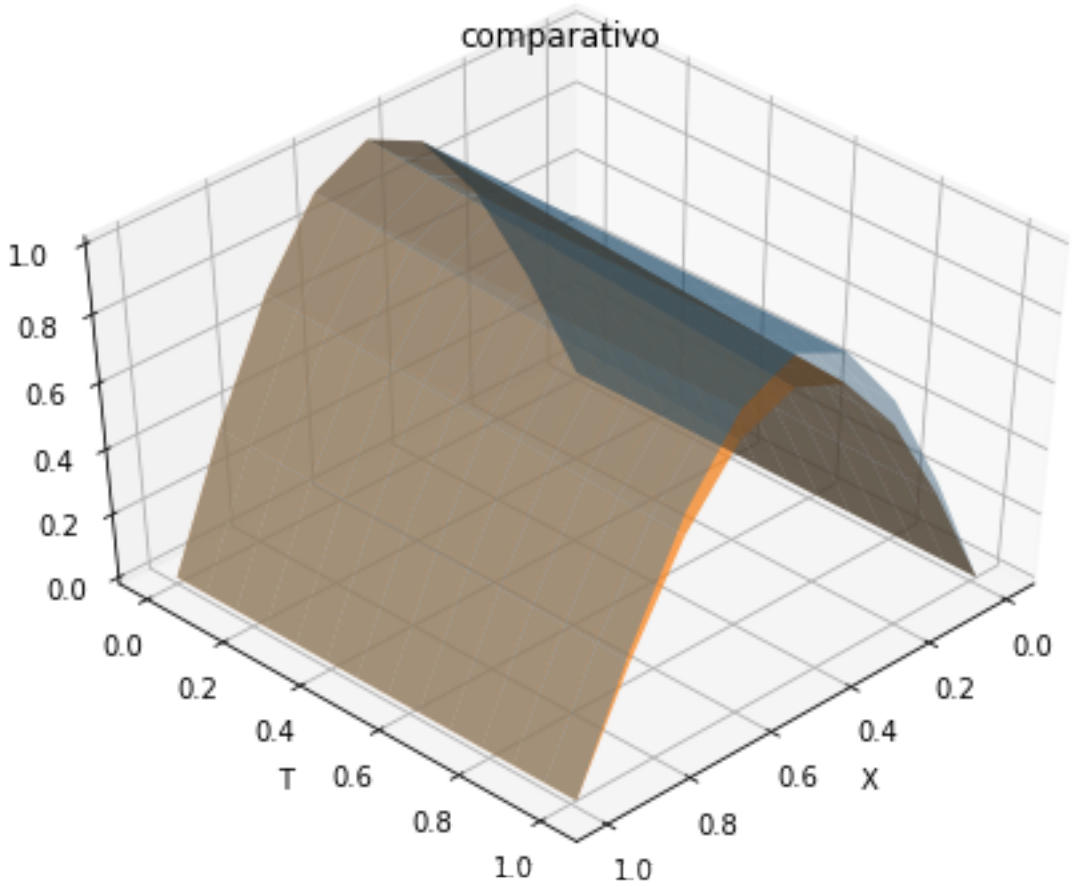
```
-0.1*exp(-0.1*t)*sin(pi*x) + pi**2*exp(-0.1*t)*sin(pi*x)
```

```
# d0 = u(0,t) = 0
# d1 = u(1,t) = 0
# f = sin(pi*x)
# -0.1*exp(-0.1*t)*sin(pi*x) + pi**2*exp(-0.1*t)*sin(pi*x)
d0 = 0.0
d1 = 0.0
f = lambda x: sin(pi*x)
G = lambda x,t: -0.1*exp(-0.1*t)*sin(pi*x) + pi**2*exp(-0.1*t)*sin(pi*x)
t0 = 0.0
T = 1.0
h = 0.078
m = 8

# sol. numerica
x,t,un = met_linhas_euler(d0,d1,f,G,t0,T,h,m)

TT,X = meshgrid(t,x)
```

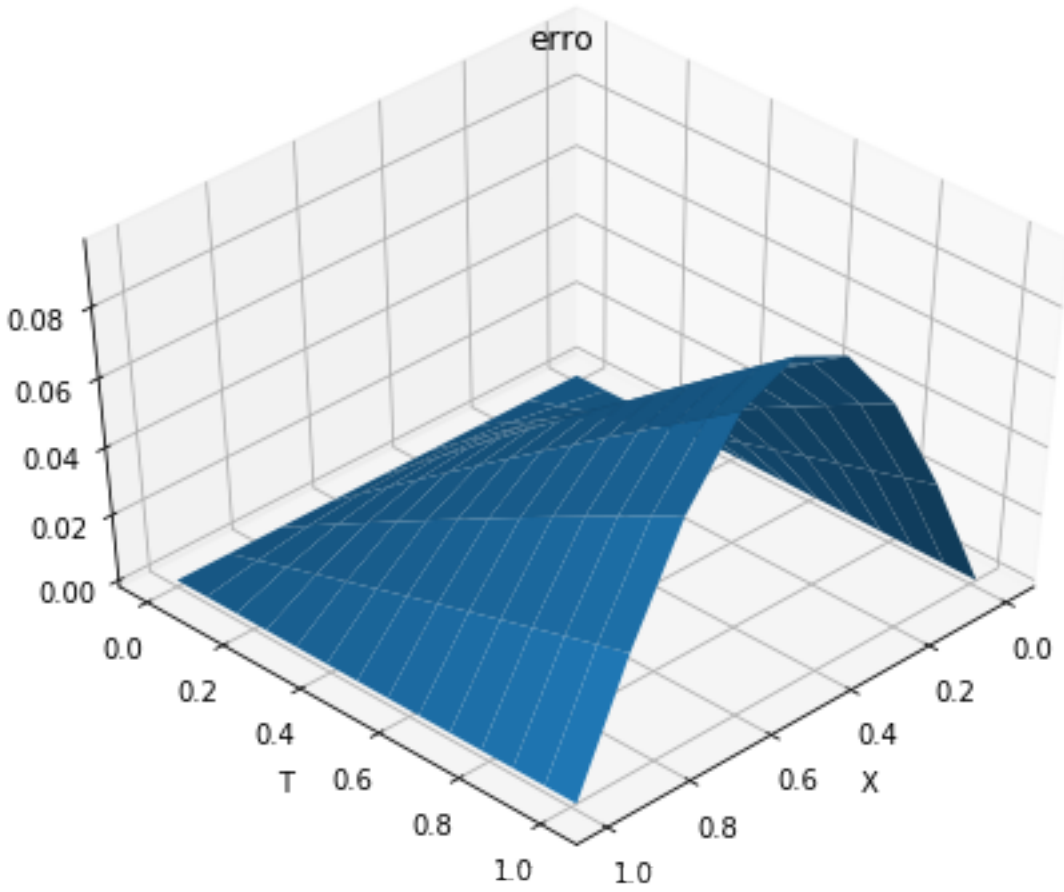
(continues on next page)

(continued from previous page)

```
# sol. exata
ue = exp(-0.1*TT)*sin(pi*X)

# plotagem
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, TT, un, alpha=0.4)
ax.plot_surface(X, TT, ue, alpha=0.7)

plt.xlabel('X')
plt.ylabel('T')
plt.title('comparativo')
ax.view_init(45, 45)
```




```
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1, projection='3d')
ax.plot_surface(X, TT, abs(ue-un), alpha=1.)
plt.xlabel('X')
plt.ylabel('T')
plt.title('erro')
ax.view_init(45, 45)
```


erro
0.08
0.06
0.04
0.02
0.00
0.0
0.2
0.4
T
0.6
0.8
1.0
0.0
0.2
0.4
0.6
X
0.8
1.0