Coletânea de *Notebooks* para Métodos Numéricos

Projeto Numbiosis

Gustavo Charles P. de Oliveira
Departamento de Computação Científica
Universidade Federal da Paraíba

13 de Abril de 2020

Conteúdo

1 Modelagem Matemática em Ciências Computacionais e Engenharias

[1]:
```
%matplotlib inline
```

1.1 O que é um modelo matemático

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

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

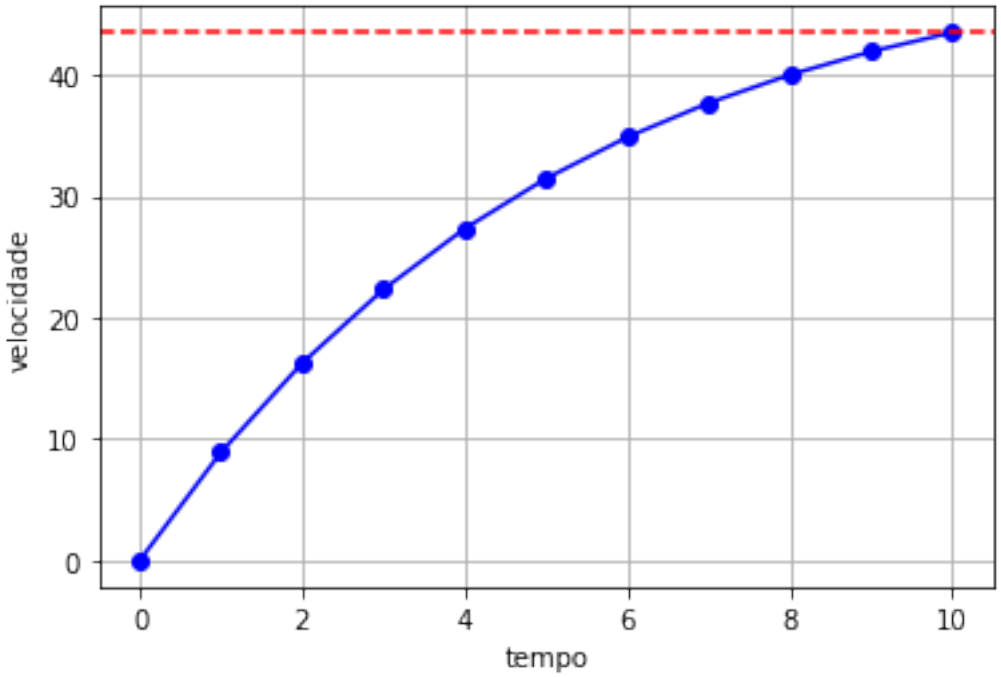
Como veremos adiante em um estado de caso real apresentado por Yan Ferreira, esta função cresce exponencialmente até atingir uma estabilização na *velocidade terminal*.

1.2 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.

[2]:
```
# 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.
```

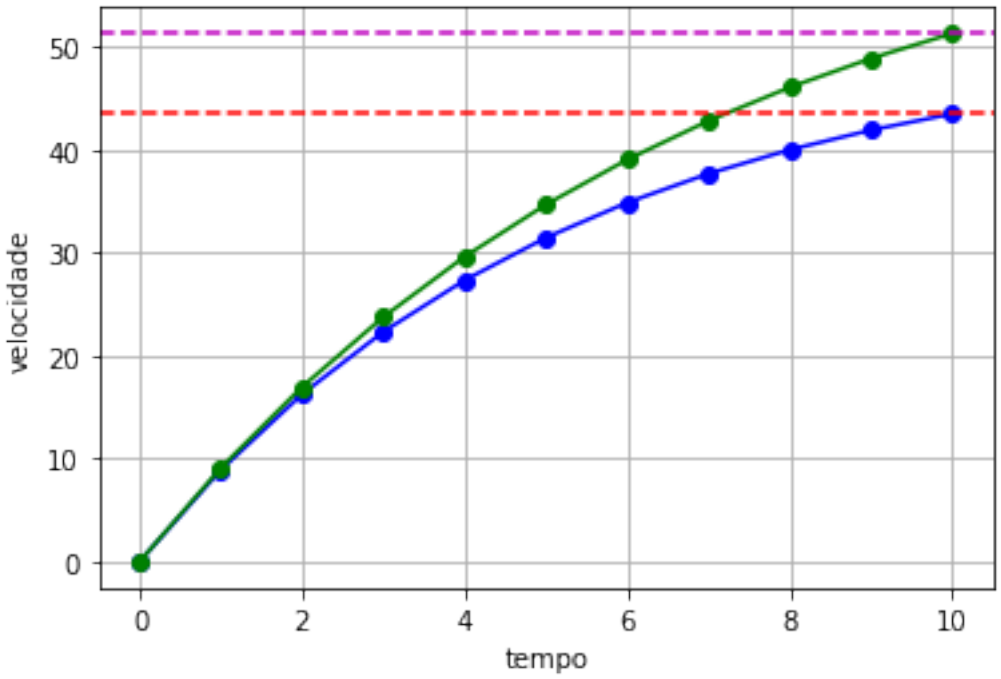


Porém, Celso, irmão de Yan, também saltou com ele, em separado. Celso, tem mais 20kg a mais do que Yan. Então, vamos ver como a massa influenciou a velocidade no salto de Celso e comparas as curvas.

[3]:
```
# 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
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.

1.2.1 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).

[4]:
```
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})
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!
```

1.3 Programação estruturada e modular

1.3.1 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.

1.3.2 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.

[5]:
```
"""
Módulo: ponto.py
Exemplo de programação modular.
Classe para trabalhar com pontos do espaço 2D.
"""

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))
```

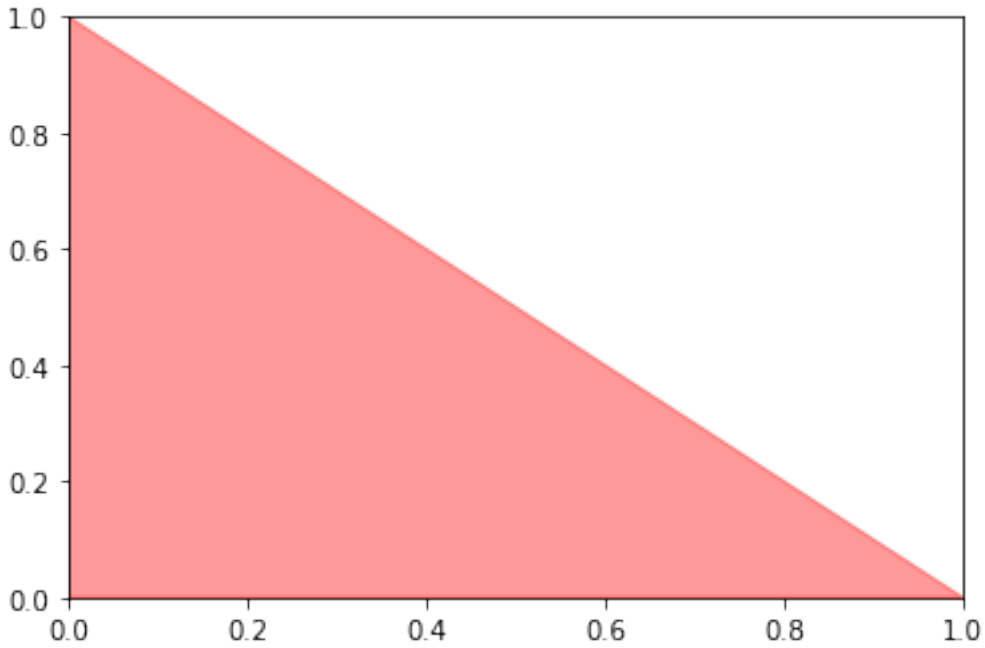
Exemplo: usando a classe `ponto.py` para calcular a área de um triângulo retângulo

[6]:
```
# 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
```

[7]:
```
# 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)
plt.gca().add_patch(pol);
```

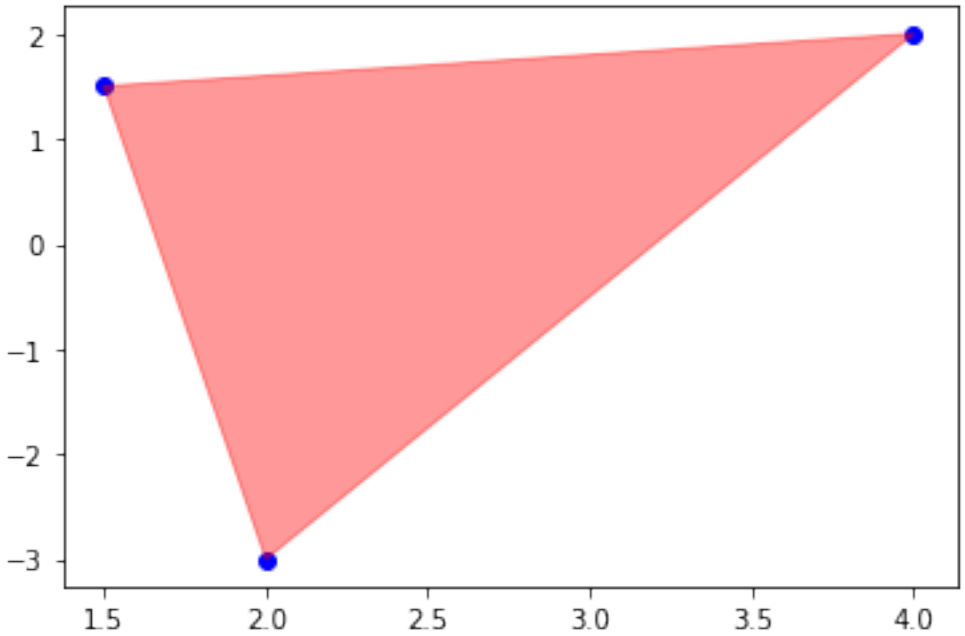


Exemplo: usando a classe `ponto.py` para calcular a área de um triângulo qualquer

[8]:
```
# 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
```



[9]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

2 Conversão numérica e ponto flutuante

[1]:
```
%matplotlib inline
```

2.1 Sistema binário

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

2.1.1 Exercícios de conversão numérica

[2]:
```
# (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
```

[3]:
```
# (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
```

[4]:
```
# (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
```

[5]:
```
"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?

2.2 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.

[6]:
```
"""
Converte inteiro para binário
por divisões sucessivas.
! Confronte com a função residente 'bin()'
"""


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
    #       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 ***
```

2.3 Sistema de ponto flutuante

2.3.1 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:

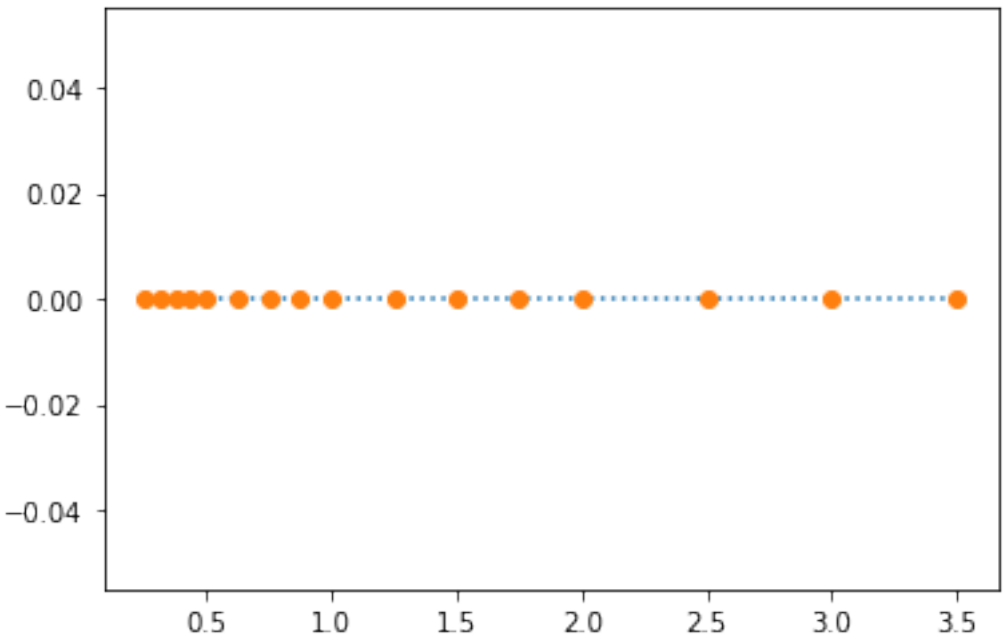
- $(0.100 \times 2^{-1})_2 = (0.01)_2 = 0.2^0 + 0.2^{-1} + 1.2^{-2} = 1/4$
- $(0.100 \times 2^{0})_2 = (0.1)_2 = 0.2^0 + 1.2^{-1} = 1/2$
- $(0.100 \times 2^{1})_2 = (1.0)_2 = 1.2^0 + 0.2^{-1} = 1$
- $(0.100 \times 2^{2})_2 = (10.0)_2 = 1.2^1 + 0.2^1 + 0.2^{-1} = 2$
- $(0.101 \times 2^{-1})_2 = (0.0101)_2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

(...)
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:

[7]:
```
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');
```



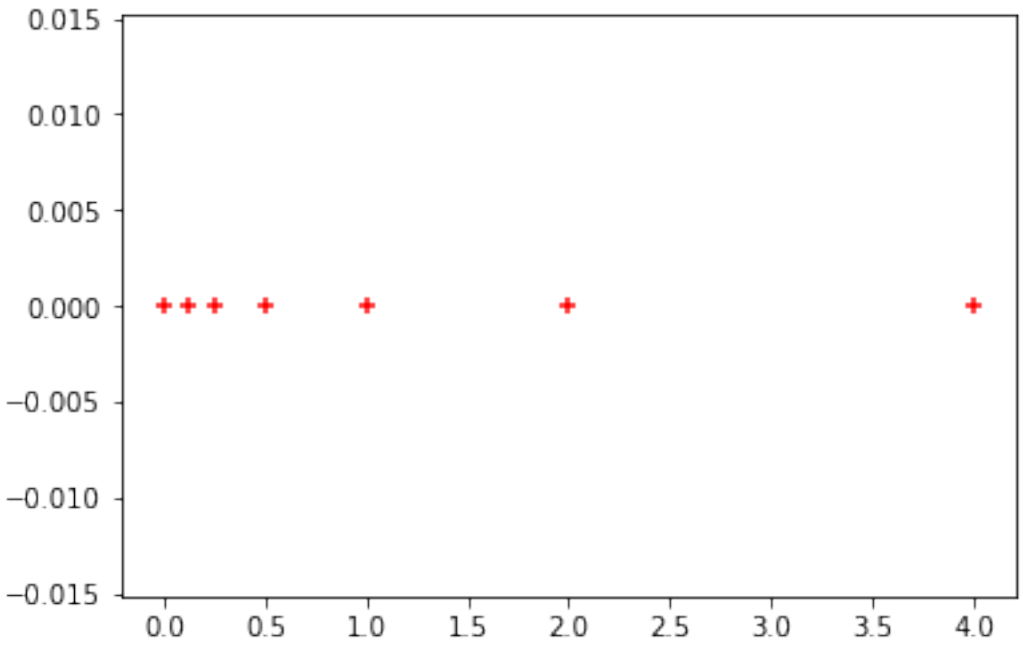
Isto é, $\mathbb{F}$ é uma reta "perfurada", para a qual apenas 16 números positivos, 16 simétricos destes e mais o 0 são representáveis. Logo, o conjunto contém apenas 33 elementos.

2.4 Simulador de $\mathbb{F}$

[8]:
```
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.]
```



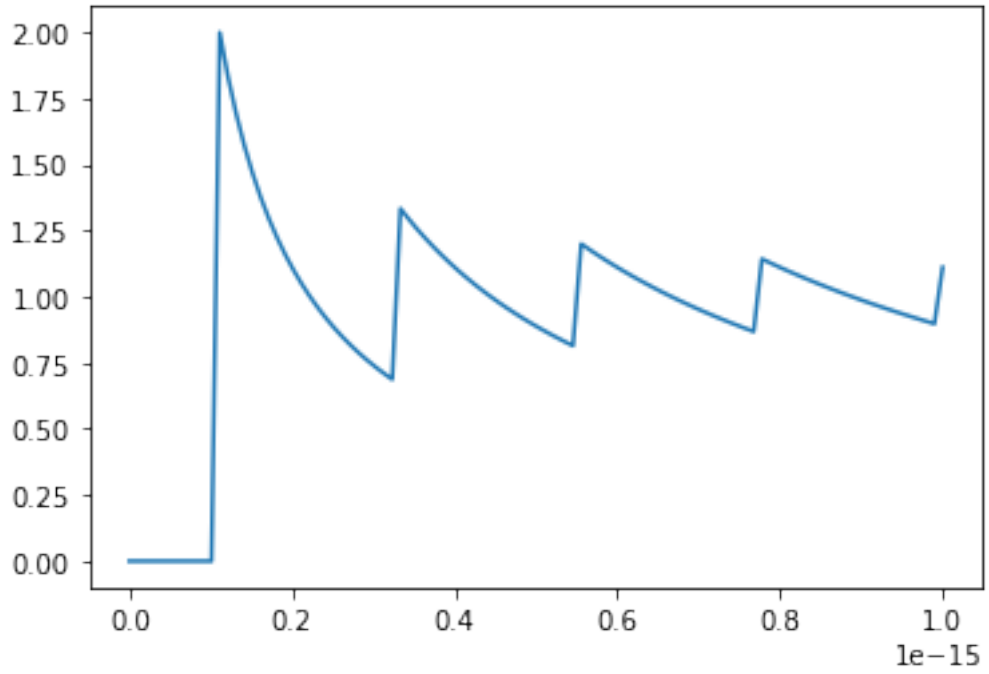
2.5 Limites de máquina para ponto flutuante

[9]:
```
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
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
```

[10]:
```
from matplotlib.pyplot import plot

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



[11]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

3 Erros numéricos e seus efeitos

[1]:
```
%matplotlib inline
```

3.1 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...

[2]:
```
# parcelas

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

```
print('p1 (com arredondamento): {0:.2f}'.format(p1)) # precisão 1

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))

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:

[3]:
```
# 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
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

[4]:
```
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 $EA2 > EA1$. Entretanto, podemos verificar pelo seguinte teste lógico:

[5]:
```
# teste é verdadeiro
EA1 > EA2
```

[5]:
```
True
```

Erros relativos

Os erros relativos também podem ser computados como:

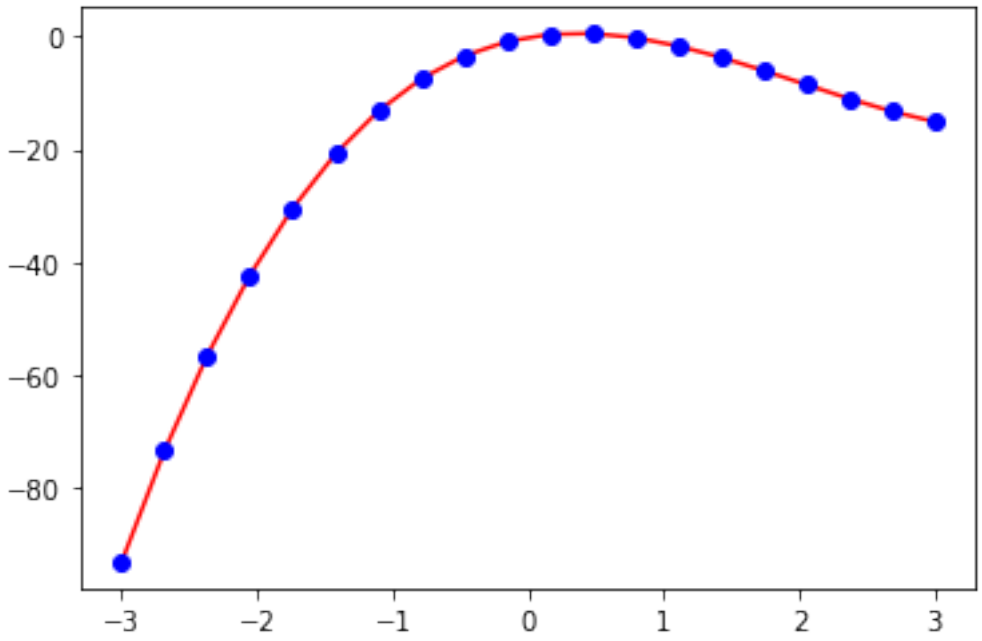
[6]:
```
ER1 = EA1/abs(x_exato)
print(ER1)

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

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

Gráfico de $P(x)$

[7]:
```
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');
```



3.1.1 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.

[8]:
```
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:

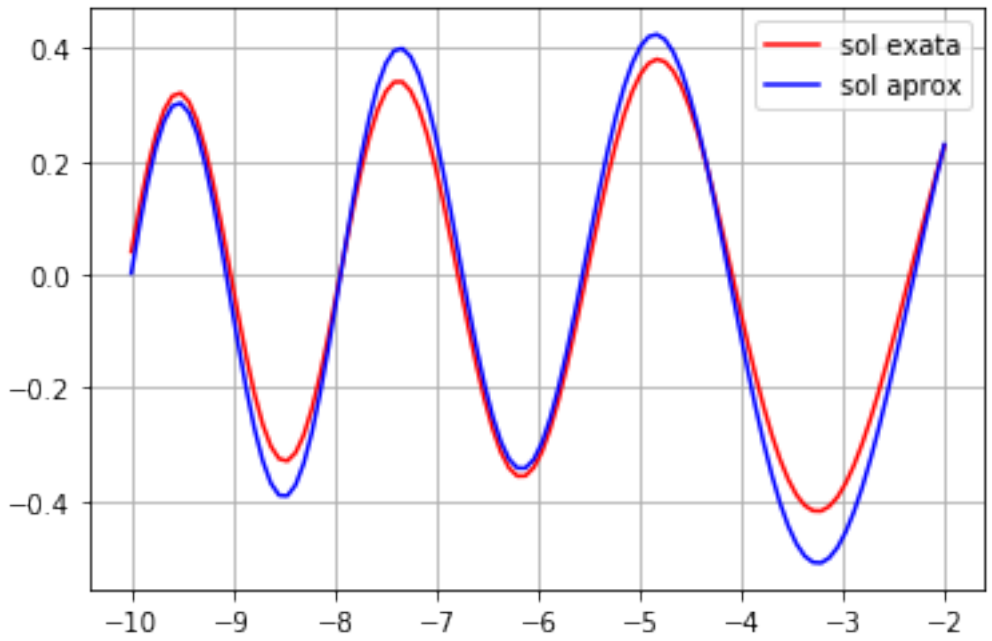
$$ER_p(x) = \frac{\mid f_{aprox}(x) - f_{ex}(x) \mid}{\mid f_{ex}(x) \mid},$$

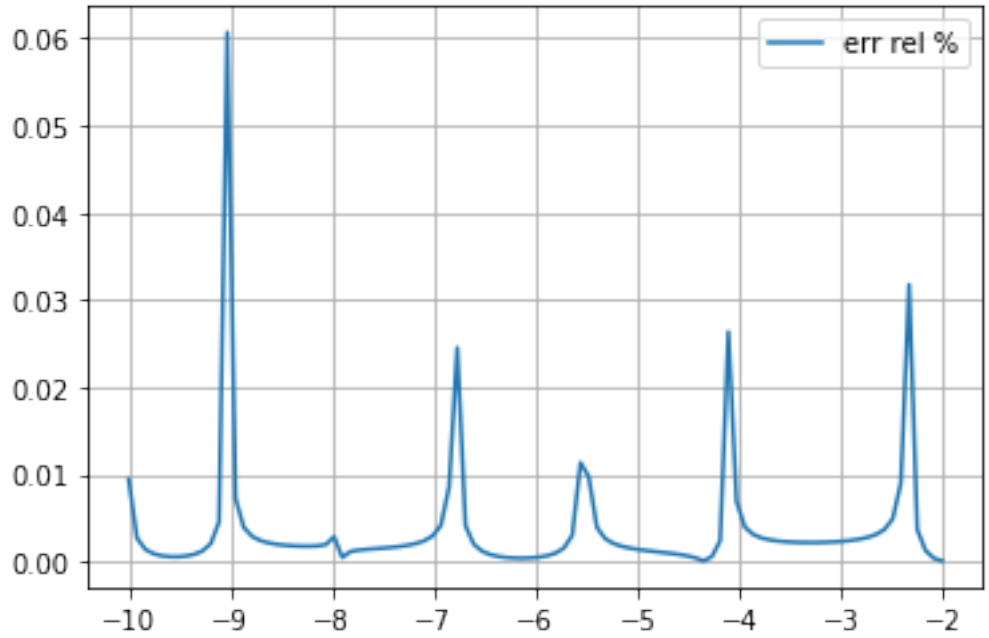
onde $f_{aprox}(x)$ é o valor da função aproximada (de Airy) e onde $f_{ex}(x)$ é o valor da função exata (de Airy).

[9]:
```
# 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.

[10]:
```
# 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');
```





3.2 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 $0.25\epsilon_m$.

3.3 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

$$e^{-v} = \frac{1}{e^v}.$$

[11]:
```
# 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
```

[12]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

[ ]:
```
```

4 Análise gráfica, localização e refinamento de raízes

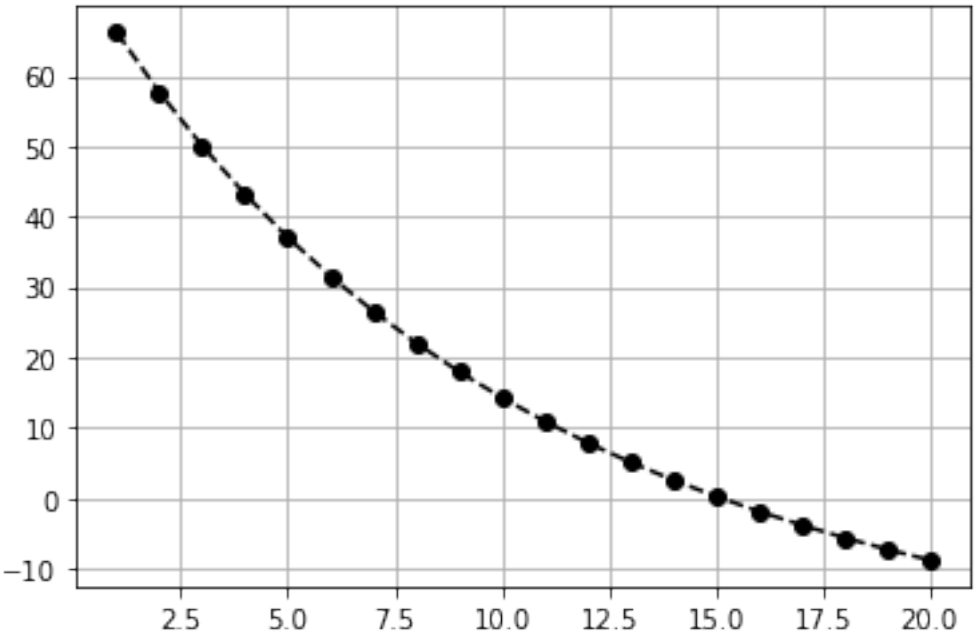
4.1 Estudo de caso: salto do paraquedista

[1]:
```
%matplotlib inline
```

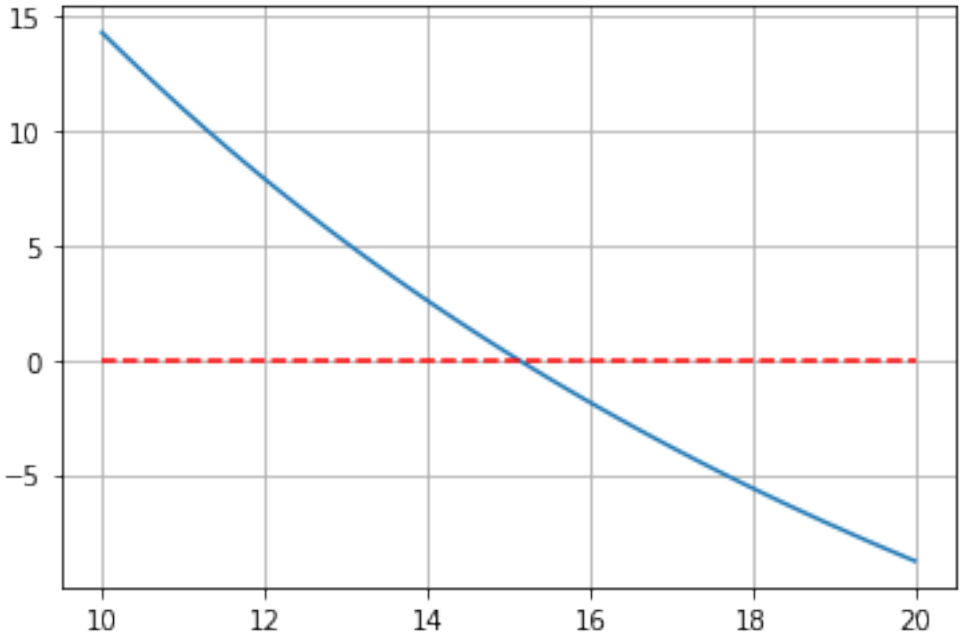
[2]:
```
import numpy as np
import matplotlib.pyplot as plt

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

[3]:
```
# localizacao
a,b = 1,20
c = np.linspace(a,b,20)
```

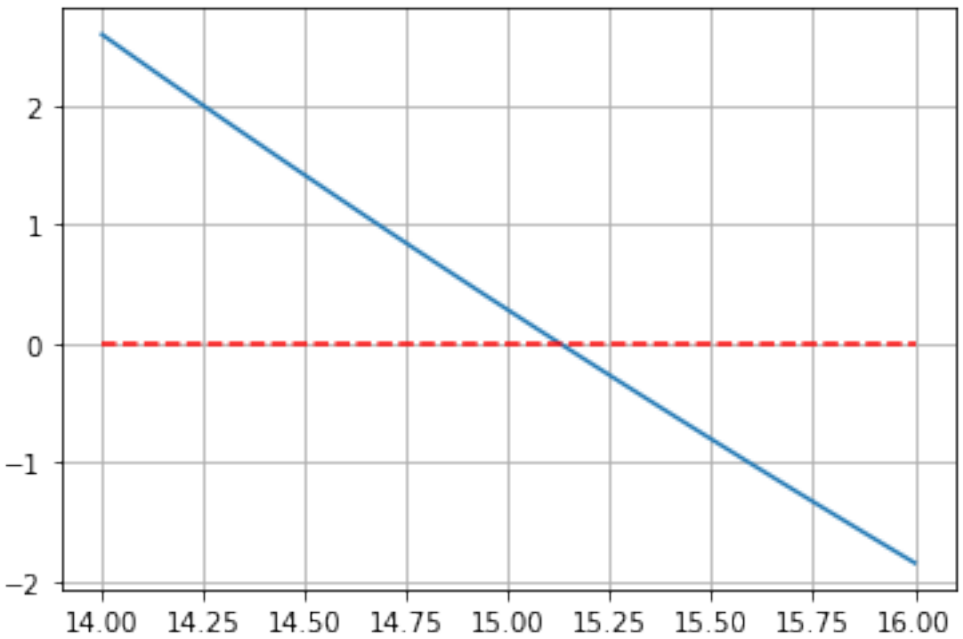
```
f = g*m/c*(1 - np.exp(-c/m*t)) - v

plt.plot(c,f,'k--o');
plt.grid(True)
```



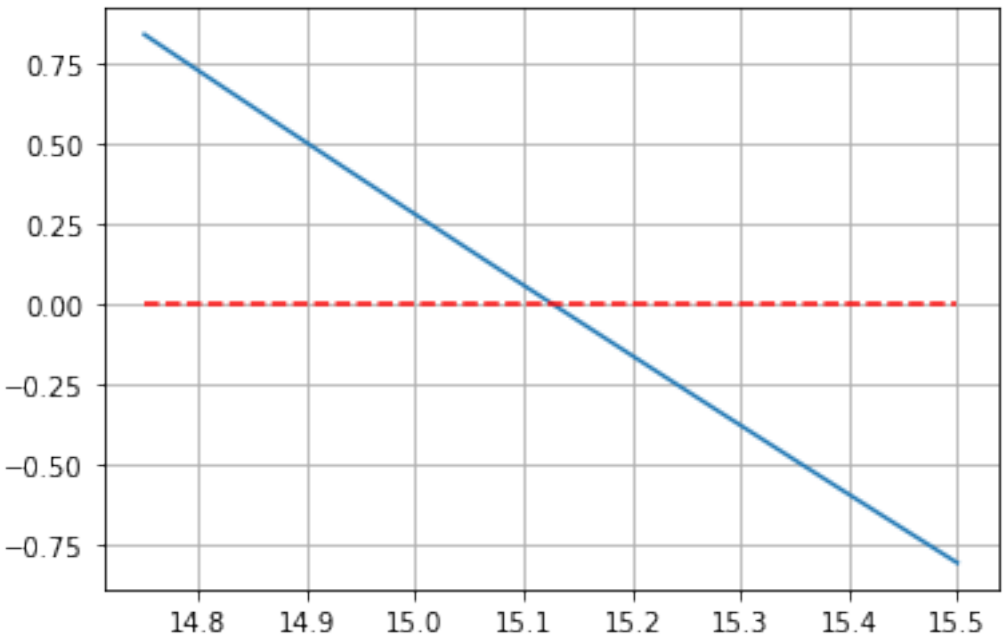
[4]:
```
# refinamento
a,b = 10,20
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

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



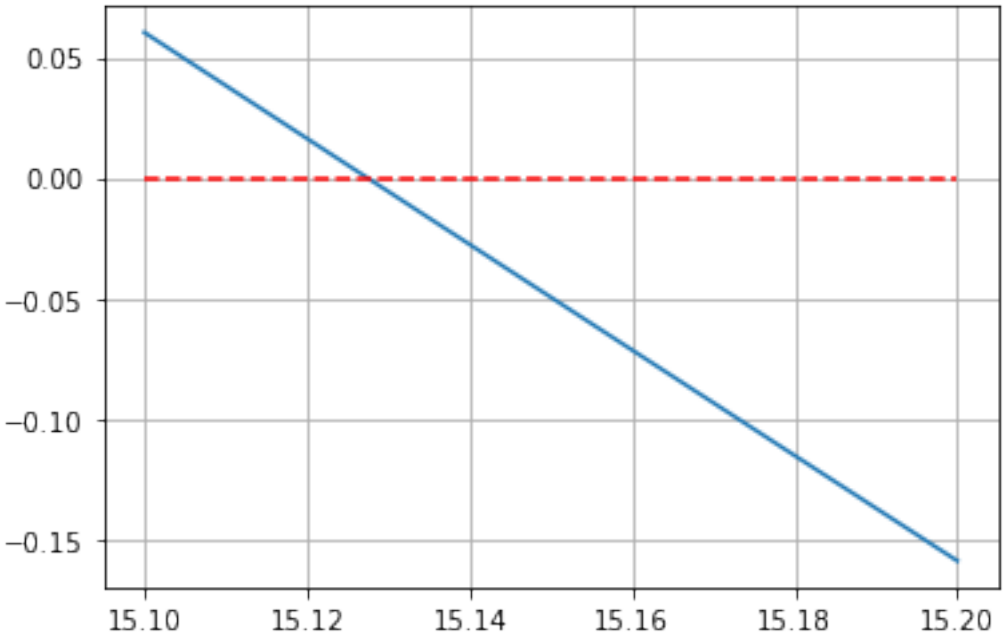
[5]:
```
# refinamento
a,b = 14,16
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

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



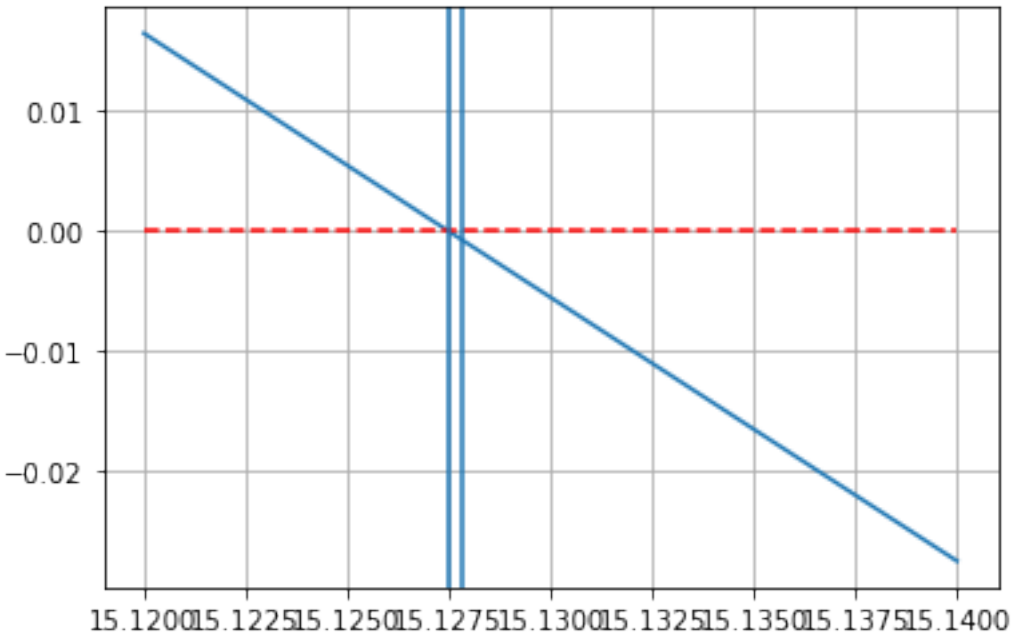
[6]:
```
# refinamento
a,b = 14.75,15.5
c = np.linspace(a,b,100)
f = g*m/c*(1 - np.exp(-c/m*t)) - v

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



[7]:
```
# 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()
```



[8]:
```
# refinamento
a,b = 15.12,15.14
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()
```



4.2 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:
$s = \sum_{i=1}^{n} i = i + i + \ldots + i = s_1 + s_2 + \ldots + s_n$, para um valor n finito.
Podemos fazer isto da seguinte maneira:

[9]:
```
n = 10

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

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

[10]:
```
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
```

[11]:
```
# 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
```

[12]:
```
s = 0.
for i in range(n):
    s += i # produz o mesmo que s <- s + i
    print(i+1,s)
```

```
1 0.0
2 1.0
3 3.0
4 6.0
5 10.0
6 15.0
7 21.0
8 28.0
9 36.0
10 45.0
```

[13]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

5 Implementações do método da bisseção

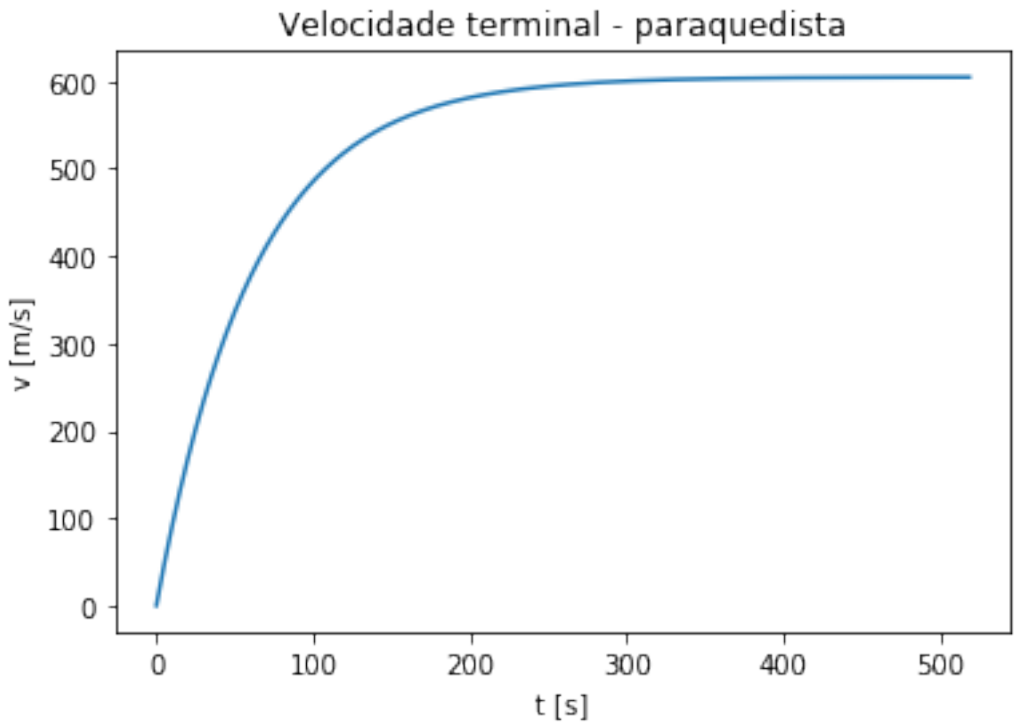
[1]:
```
%matplotlib inline
```

[2]:
```
"""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
    # 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
```

[3]:
```
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]')
plt.ylabel('v [m/s]')
plt.title('Velocidade terminal - paraquedista');
```



[4]:
```
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

(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
```

[5]:
```
# Exemplo 5.1
f = 'x**3 - 9*x + 3'
a = 0
b = 1
tol = 1e-3
N = 100
var = 'x'
bissecao(f,a,b,tol,N,var)

# relação erro desejado x número de iterações
k = np.log2((b-a)/tol)
print("O número de iterações é k = {0:g}  {1:g}.".format(k,np.round(k)))
```

```
(i = 1) f(xm)=-1.375000 | f(a)=3.000000 | f(b)=-5.000000
(i = 2) f(xm)=0.765625 | f(a)=3.000000 | f(b)=-1.375000
(i = 3) f(xm)=-0.322266 | f(a)=0.765625 | f(b)=-1.375000
(i = 4) f(xm)=0.218018 | f(a)=0.765625 | f(b)=-0.322266
(i = 5) f(xm)=-0.053131 | f(a)=0.218018 | f(b)=-0.322266
(i = 6) f(xm)=0.082203 | f(a)=0.218018 | f(b)=-0.053131
(i = 7) f(xm)=0.014474 | f(a)=0.082203 | f(b)=-0.053131
(i = 8) f(xm)=-0.019344 | f(a)=0.014474 | f(b)=-0.053131
(i = 9) f(xm)=-0.002439 | f(a)=0.014474 | f(b)=-0.019344
(i = 10) f(xm)=0.006017 | f(a)=0.014474 | f(b)=-0.002439
Solução encontrada: 0.33740234375
O número de iterações é k = 9.96578  10.
```

5.1 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.)

(c) Use o método da bisseção.

5.2 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...`).

[6]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

6 Método da Iteração Linear (Ponto Fixo)

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

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

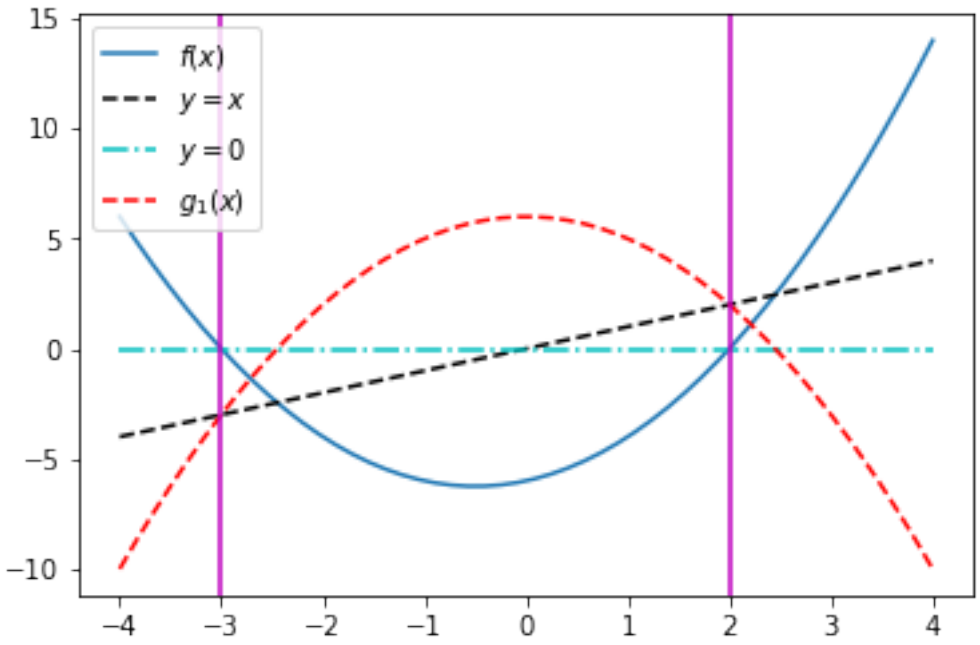
6.1 Exemplo

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

[2]:
```
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');
```

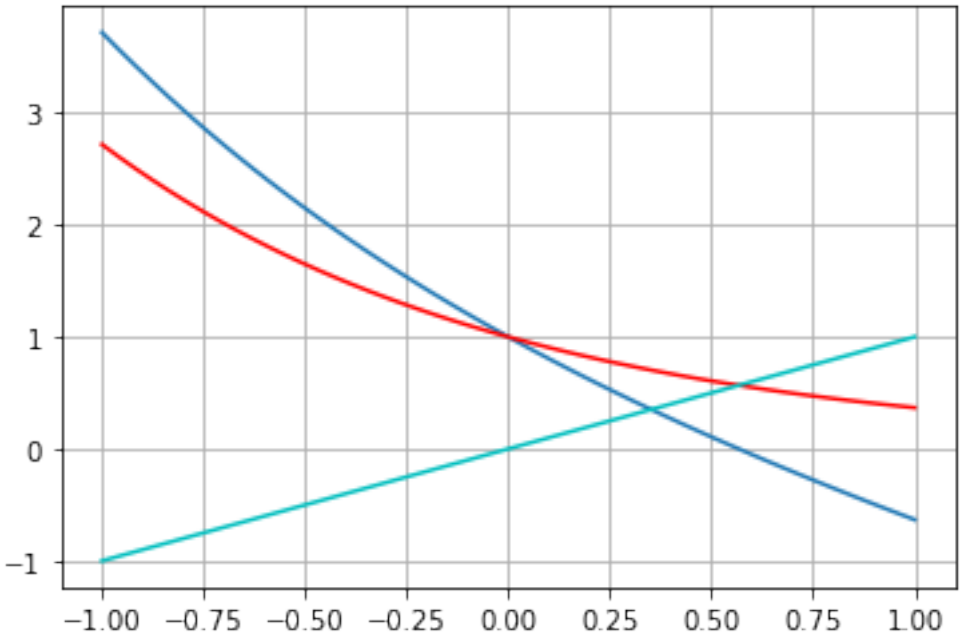
```
Raízes: x1 = -3.000000, x2 = 2.000000
```



6.2 Exemplo

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

[3]:
```
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)
```



6.3 Implementação do método do ponto fixo

[4]:
```
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

    # 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
```

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

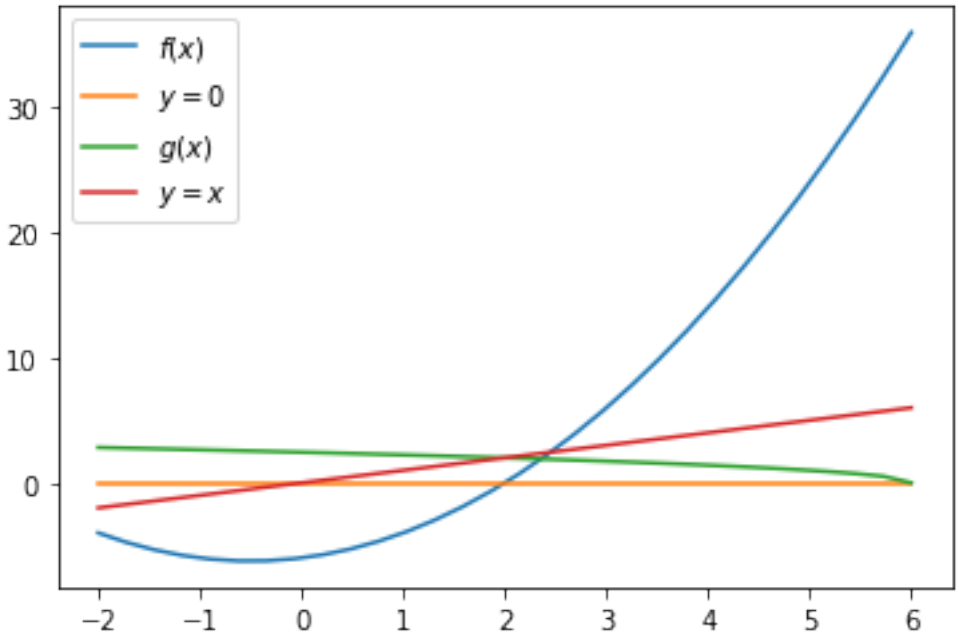
Função de iteração: $g(x) = \sqrt{6 - x}$

[5]:
```
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
```

[5]: `1.9999983262723453`

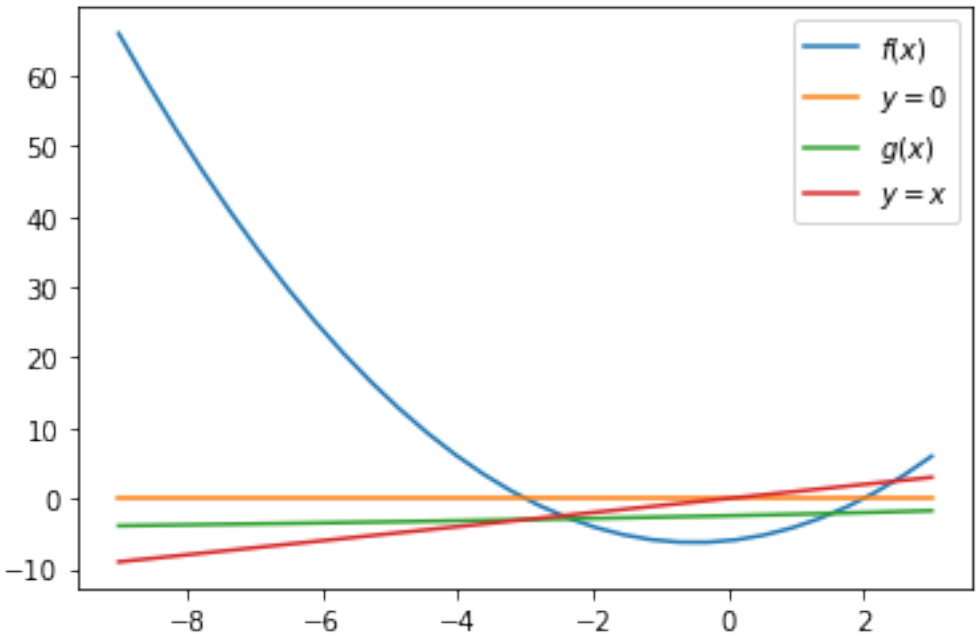


Função de iteração: $g(x) = -\sqrt{6-x}$

[6]:
```
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
```

[6]:
```
-2.9999979200736955
```



[7]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

7 Implementação do método de Newton

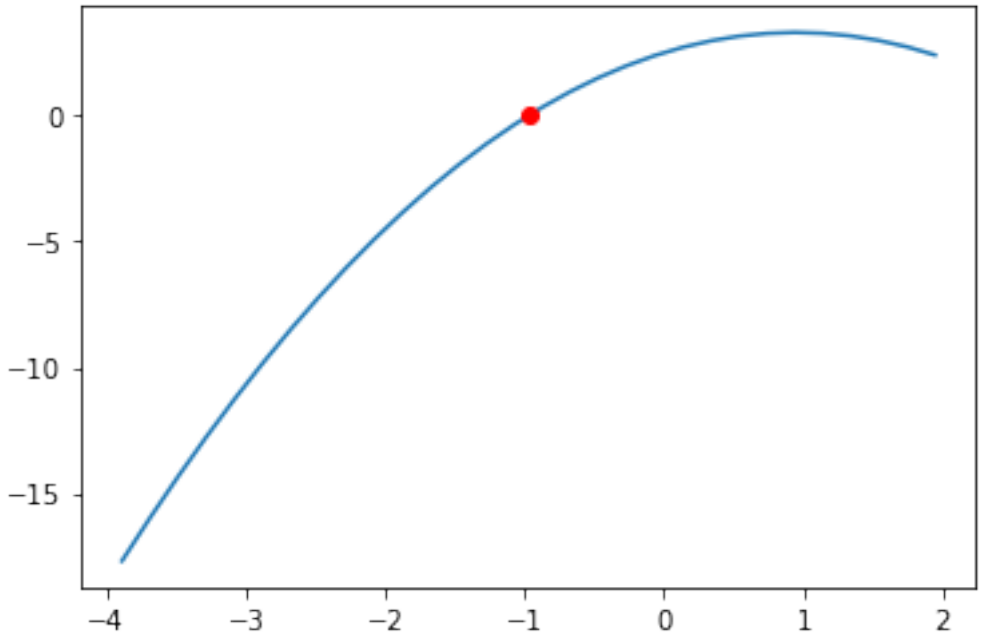
[1]:
```
%matplotlib inline
```

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

[2]:
```
# 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
```

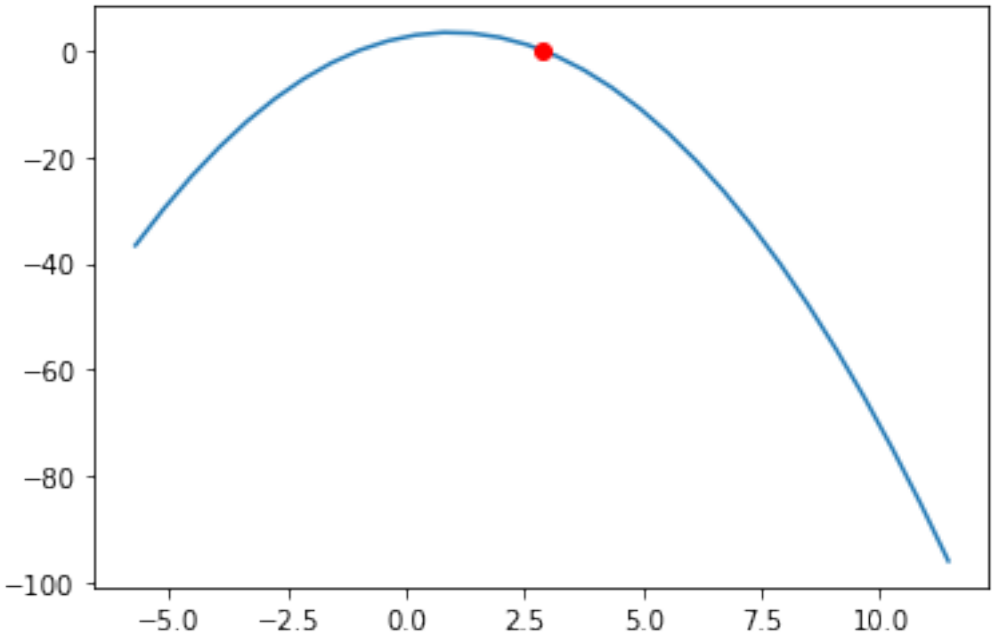
```
    # imprime estimativa inicial
    print('Estimativa inicial: x0 = {0}\n'.format(x0))

    # 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)
        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
```



[3]:
```
# 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
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
```



7.1 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`.

7.1.1 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.

7.1.2 Solução por função residente

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

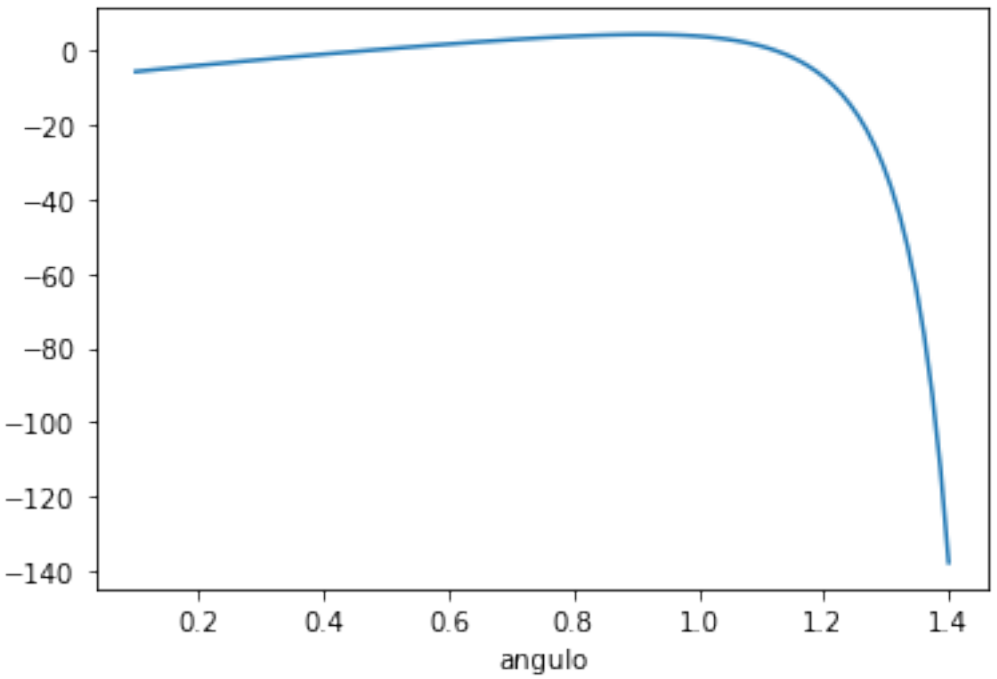
[4]:
```
from scipy.optimize import newton
import numpy as np
import matplotlib.pyplot as plt
```

```
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
```

7.2 Localização

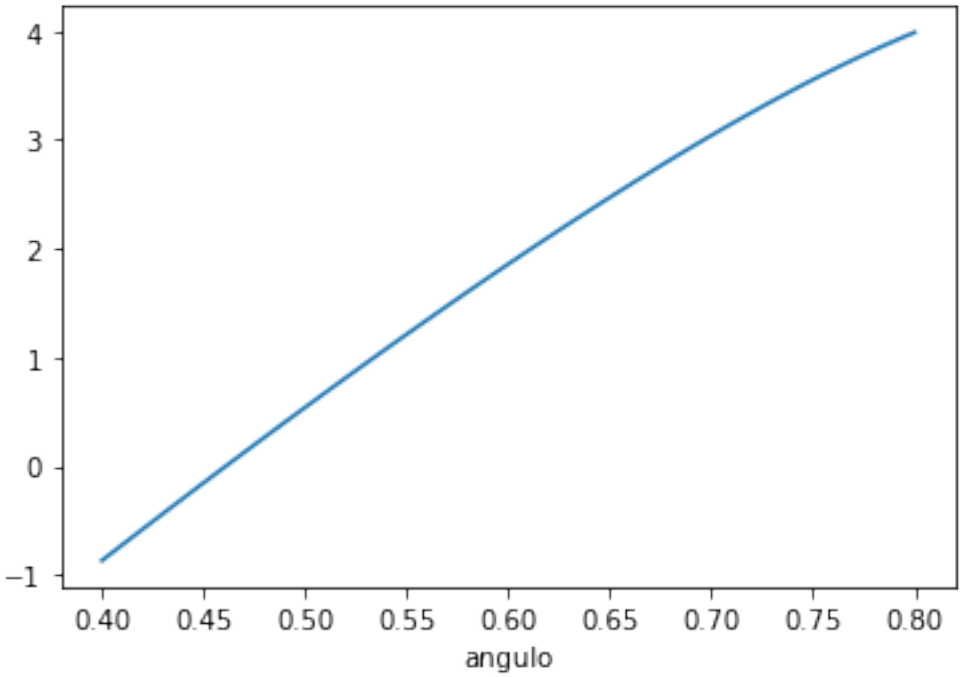
[5]:
```
th = np.linspace(0.1,1.4,100,True)
plt.plot(th,f(th))
plt.xlabel('angulo')
```

[5]:
```
Text(0.5, 0, 'angulo')
```



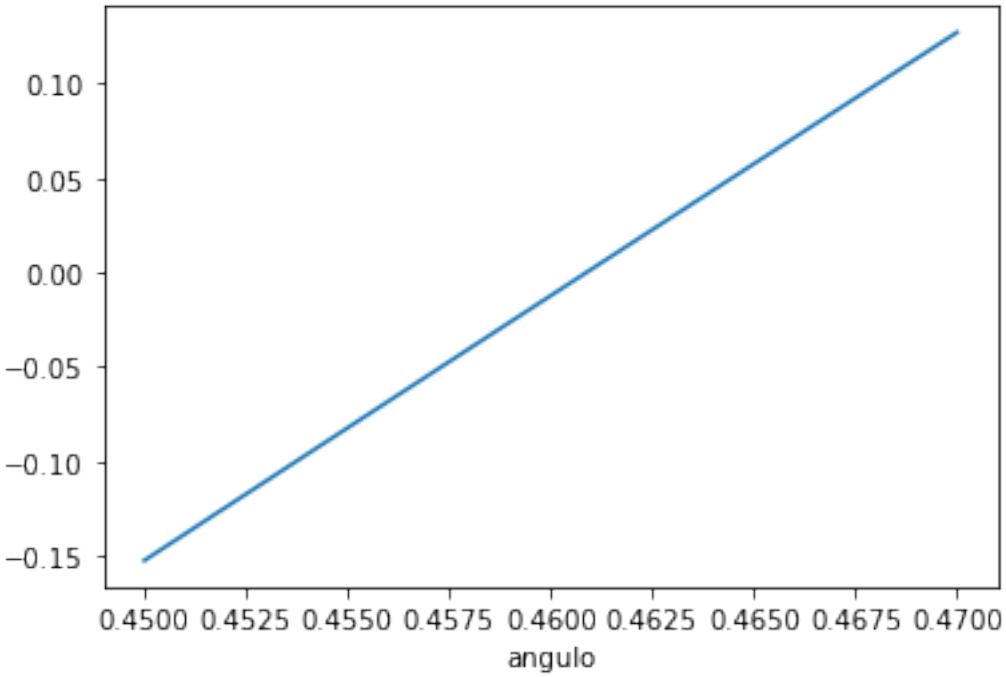
[6]:
```
th = np.linspace(0.4,0.8,100,True)
plt.plot(th,f(th))
plt.xlabel('angulo')
```

[6]:
```
Text(0.5, 0, 'angulo')
```



[7]:
```
th = np.linspace(0.45,0.47,100,True)
plt.plot(th,f(th))
plt.xlabel('angulo')
```

[7]:
```
Text(0.5, 0, 'angulo')
```



7.3 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, quale é a raiz com uma estimativa inicial de 0.47.

[8]:
```
ang = newton(f,0.47)
ang
```

[8]:
```
0.4608834641642987
```

[9]:
```
np.rad2deg(ang)
```

[9]:
```
26.406677343983237
```

[10]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

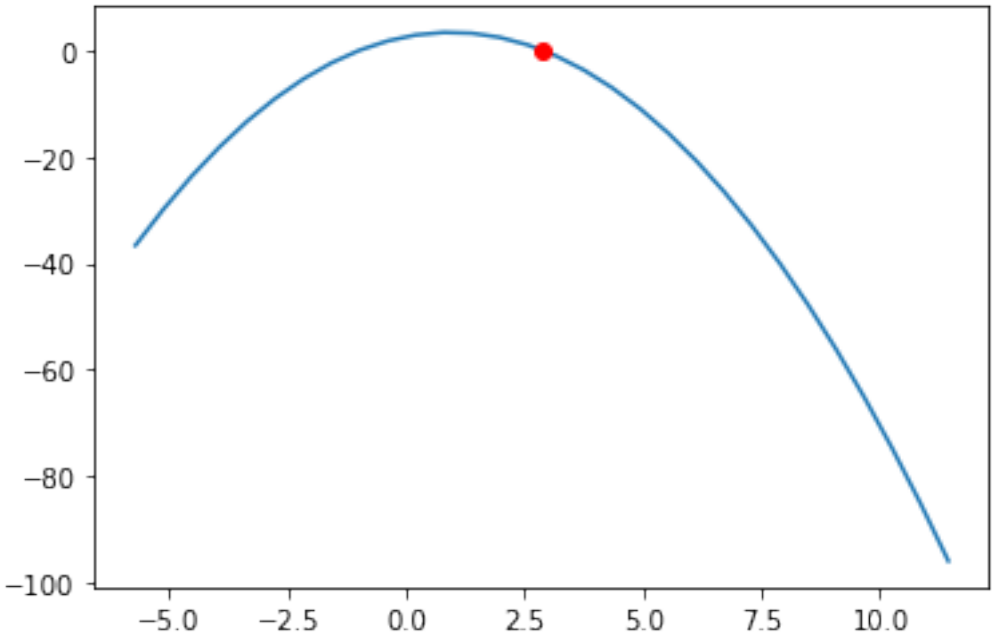
8 Implementação do método da secante

[1]:
```
%matplotlib inline
```

[2]:
```
# 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


# 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
```



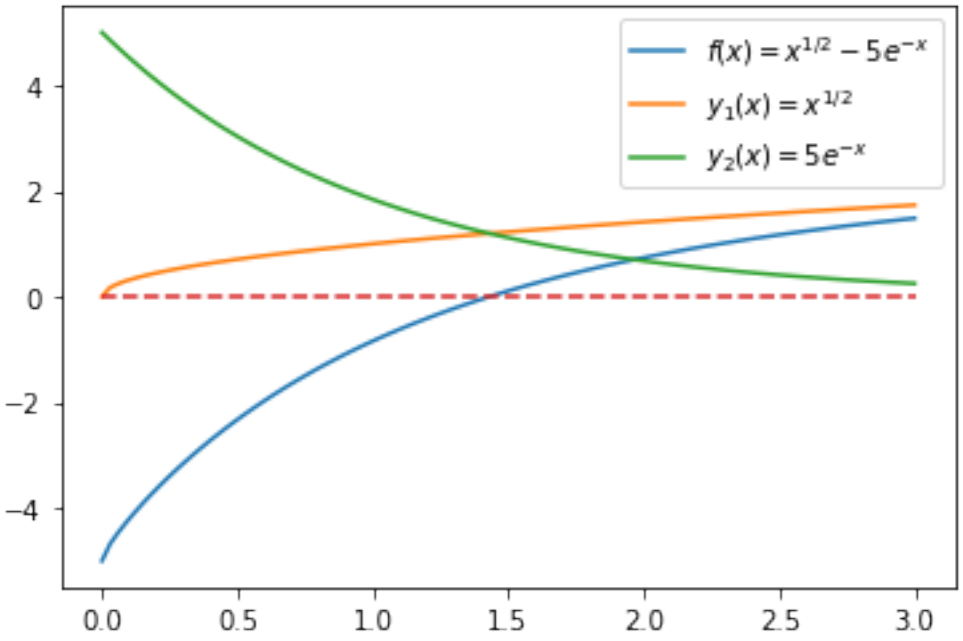
8.1 Problema

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

8.1.1 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 $x_1 = 1.5$.

[3]:
```
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 $f(x = 1.5)$.

[4]:
```
fx(1.5)
```

[4]:
```
0.10909407064943988
```

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

[5]:
```
xm = lambda a,b: ( a*fx(b) - b*fx(a) ) / (fx(b) - fx(a) )
```

Vamos usar os nosso valores estimados:

[6]:
```
x0 = 1.4
x1 = 1.5
x2 = round(xm(x0,x1),3)
print(x2)
```

```
1.431
```

Agora, usamos este novo valor e o anterior.

[7]:
```
x3 = round(xm(x1,x2),3)
print(x3)
```

```
1.43
```

Calculemos o erro relativo entre as estimativas x_1 e x_2:

[8]:
```
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:

[9]:
```
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.

[10]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

9 Raízes de polinômios com o método de Müller

[1]:
```
%matplotlib inline
```

9.1 Computação com polinômios

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

[2]:
```
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

[3]:
```
p
```

[3]: $a_0 + x\left(a_1 + x\left(a_2 + a_3 x\right)\right)$

Imprimindo a derivada simbólica do polinômio implementada pelo usuário

[4]:
```
dp
```

[4]: $a_1 + x\left(a_2 + a_3 x\right) + x\left(a_2 + 2a_3 x\right)$

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

[5]:
```
dp2
```

[5]: $a_1 + x\left(a_2 + a_3 x\right) + x\left(a_2 + 2a_3 x\right)$

Verificando igualdade

[6]:
```
dp == dp2
```

[6]: `True`

9.2 Calculando raízes de polinômios

[7]:
```
# 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`

[8]:
```
rc = sy.roots(pn,x,multiple=True)
rc
```

[8]: $[0.28424794239786,\ -0.57962397119893 - 0.737258363375504i,\ -0.57962397119893 + 0.737258363375504i]$

Calcula apenas as raízes reais de `pn`

[9]:
```
rr = sy.roots(pn,x,multiple=True,filter='R')
rr
```

[9]: $[0.28424794239786]$

9.3 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`).

[10]:
```
vi = np.flip(np.asarray(v),axis=0)
vi
```

[10]:
```
array([ 4. ,  3.5,  2.2, -1. ])
```

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

[11]:
```
xi = np.pi
np.polyval(vi,xi)
```

[11]: 164.48022596290957

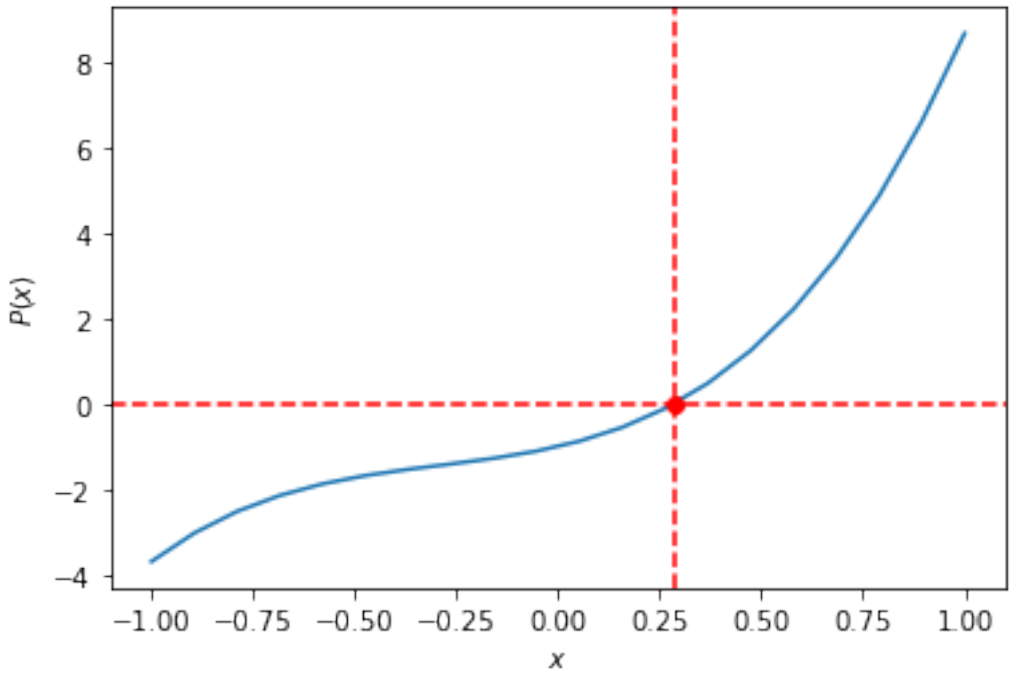
Note que se avaliássemos o polinômio em um ponto arbitrário, a forma impressa é idêntica àquela que obtivemos anteriormente.

[12]:
```
np.polyval(vi,x)
```

[12]: $x\left(x\left(4.0x + 3.5\right) + 2.2\right) - 1.0$

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 $-1 \leq x \leq 1$

[13]:
```
# 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('$P(x)$');
```



9.4 Implementação: Método de Müller

[14]:
```
# \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
```

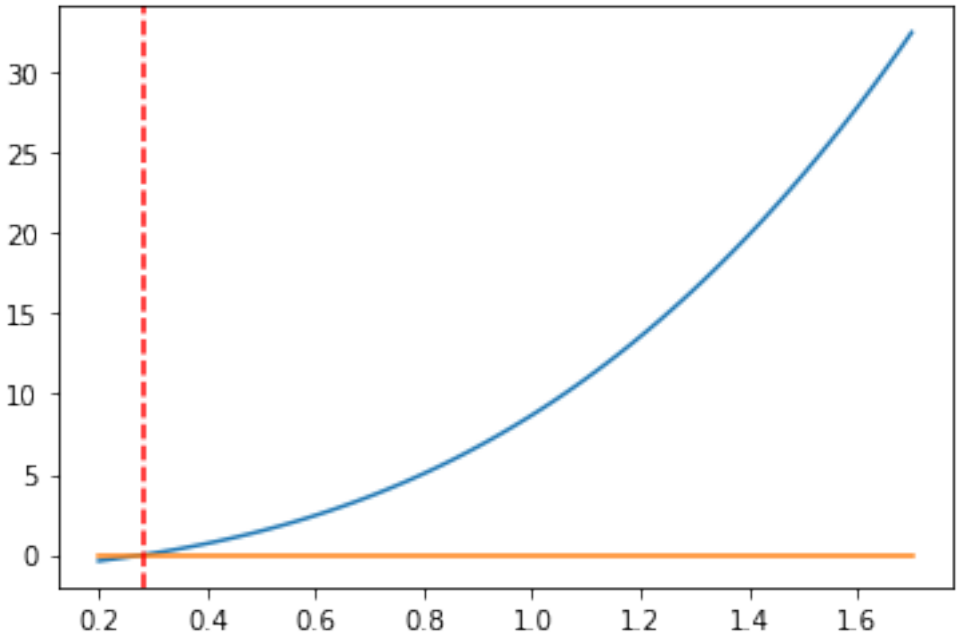
```
        h = -2*f(x2)/E
        x = x2 + h
        if abs(h) < EPS:
            return x

        # atualização
        x0 = x1
        x1 = x2
        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
```

9.5 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.

[15]:
```
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)
```

[16]:
```
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 $N = 100$.

[17]:
```
f2 = lambda x: x**4 - 3*x**3 + x**2 + x + 1

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

[17]:
```
(-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 $x_2 = 2.5$

[18]:
```
# caso 1
c1 = metodo_muller(f2,1.5,0.5,1e-5,100)
print(c1)
```

```
1.3893906833348133
```

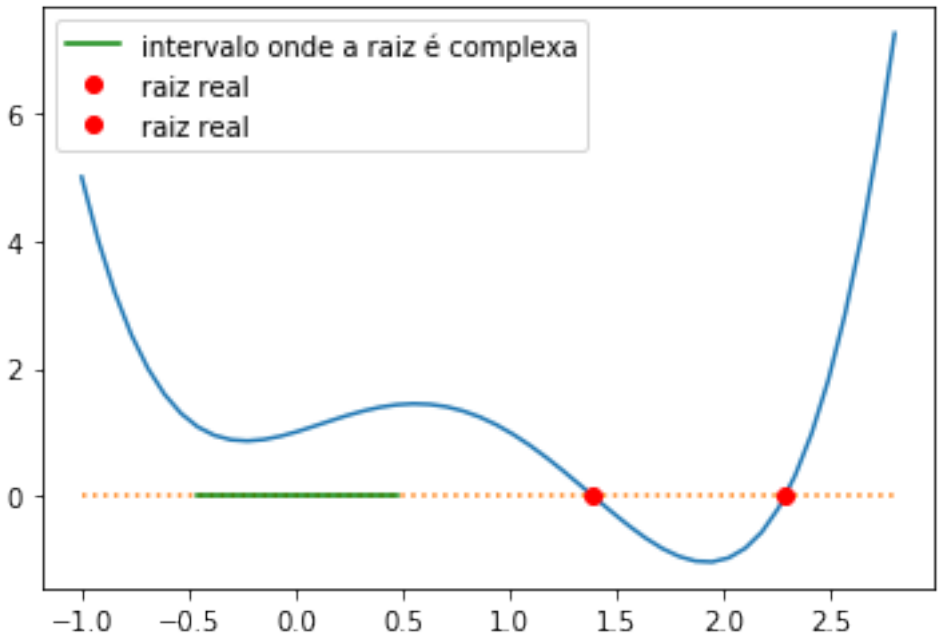
[19]:
```
# caso 2
c2 = metodo_muller(f2,2.0,0.5,1e-5,100)
print(c2)
```

```
2.2887949921884836
```

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

[20]:
```
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.

[21]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

10 Álgebra linear com Python: Eliminação Gaussiana e Condicionamento

[1]:
```
%matplotlib inline
```

10.1 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 $\mathbf{A}$.

[2]:
```
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 $\mathbf{b}$.

[3]:
```
b = np.array([12,-6.5,16,17])
print(b)
```

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

Podemos checar as dimensões com

[4]:
```
# dimensões de A
A.shape
```

[4]: `(4, 4)`

[5]:
```
# dimensão de b
b.shape
```

[5]: `(4,)`

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

[6]:
```
x = linalg.solve(A,b)
print(x)
```

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

10.2 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.

[7]:
```
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.

10.3 Verificação da solução

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

[8]:
```
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)**: $||\mathbf{r}|| = ||\mathbf{b} - \mathbf{A}\mathbf{x}|| = \langle \mathbf{b} - \mathbf{A}\mathbf{x}, \mathbf{b} - \mathbf{A}\mathbf{x} \rangle^{1/2}$

[9]:
```
r = b - A.dot(x)
np.sqrt(r.dot(r))
```

[9]:
```
1.4648213755527116e-14
```

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

11 Eliminação Gaussiana

Vamos ver como funciona a Eliminação Gaussiana.

[10]:
```
# 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. ]]
```

[11]:
```
# 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. ]]
```

[12]:
```
# 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. ]]
```

[13]:
```
# 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]]
```

[14]:
```
# 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
```

[15]:
```
# matriz randômica 5x5
M2 = np.random.rand(5,5)
print(eliminacao(M2))
```

```
[[ 8.40996592e-01  9.88112968e-02  1.02191471e-01  9.22021290e-01
   5.10618922e-02]
 [ 0.00000000e+00  2.14417054e-01  4.17568454e-01  1.85457900e-02
   6.84416673e-01]
 [ 0.00000000e+00  0.00000000e+00 -7.03848724e-01  1.03900307e-01
  -1.88516619e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 -6.17307230e-01
  -1.59535213e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.11022302e-16
   2.58216878e+00]]
```

11.1 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.

11.2 Condicionamento

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

[16]:
```
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]
```

[17]:
```
# solução do sistema A1x1 = b1
x1 = linalg.solve(A1,b1)
print(x1)
```

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

[18]:
```
d = 0.045
A2 = np.array([[1,2],[1.1-d,2]])
b2 = np.array([10,10.4])
print('matriz')
print(A2)
print('vetor')
print(b2)
```

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

[19]:
```
# 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, $\det(\mathbf{A}_2) \approx 0$.

[20]:
```
print(linalg.det(A1),linalg.det(A2))
```

```
-0.20000000000000018 -0.11000000000000032
```

[21]:
```
linalg.norm(A)*linalg.norm(linalg.inv(A))
```

[21]:
```
169.28388045827452
```

11.3 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 um expressão do tipo `str` e separadas com um ';'.

[22]:
```
# cria matriz
np.array(np.mat('1 2; 3 4'))
```

[22]:
```
array([[1, 2],
       [3, 4]])
```

[23]:
```
np.array(np.mat('1 2 3'))
```

[23]:
```
array([[1, 2, 3]])
```

[24]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

12 Fatoração LU com Python

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

[1]:
```
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

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

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

L,U = lu_nopivot(A)
```

[3]:
```
L
```

[3]:
```
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.25      ,  1.        ,  0.        ,  0.        ],
       [ 0.5       , -0.44444444,  1.        ,  0.        ],
       [ 0.25      ,  1.22222222,  0.06796117,  1.        ]])
```

[4]:
```
U
```

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

[5]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

13 Fatoração de Cholesky

[1]:
```
%matplotlib inline
```

13.1 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.

[2]:
```
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)

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

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

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

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

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 $\mathbf{x}$. (Definição baseada na "energia").

5. $\mathbf{A} = \mathbf{G}\mathbf{G}^T$ para uma matriz $\mathbf{G}$ com colunas independentes.

13.1.1 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 $\mathbf{x}$ e $\mathbf{y}$, 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 $\mathbf{x}$ e $\mathbf{y}$ é sempre menor do que 90 graus (ou $\pi/2$ radianos).

13.2 Algoritmo para a fatoração de Cholesky

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

[3]:
```
# 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]
        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
```

[3]:
$$\begin{bmatrix} \sqrt{a_{11}} & a_{12} & a_{13} \\ \frac{a_{12}}{\sqrt{a_{11}}} & \sqrt{a_{22} - \frac{a_{12}^2}{a_{11}}} & a_{23} \\ \frac{a_{13}}{\sqrt{a_{11}}} & \frac{a_{23} - \frac{a_{12}a_{13}}{a_{11}}}{\sqrt{a_{22} - \frac{a_{12}^2}{a_{11}}}} & \sqrt{a_{33} - \frac{\left(a_{23} - \frac{a_{12}a_{13}}{a_{11}}\right)^2}{a_{22} - \frac{a_{12}^2}{a_{11}}} - \frac{2a_{13}^2}{a_{11}}} \end{bmatrix}$$

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:

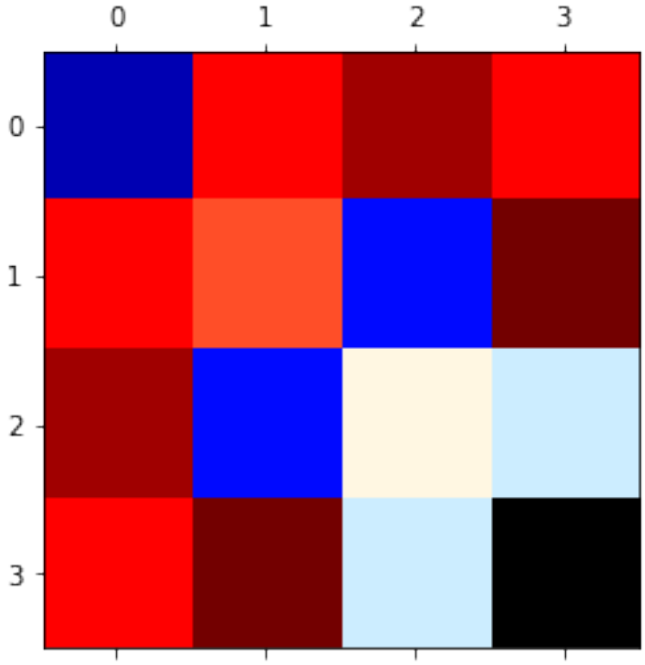
$$\mathbf{A} = \begin{bmatrix} 6 & 15 & 55 \\ 15 & 55 & 225 \\ 55 & 225 & 979 \end{bmatrix}$$

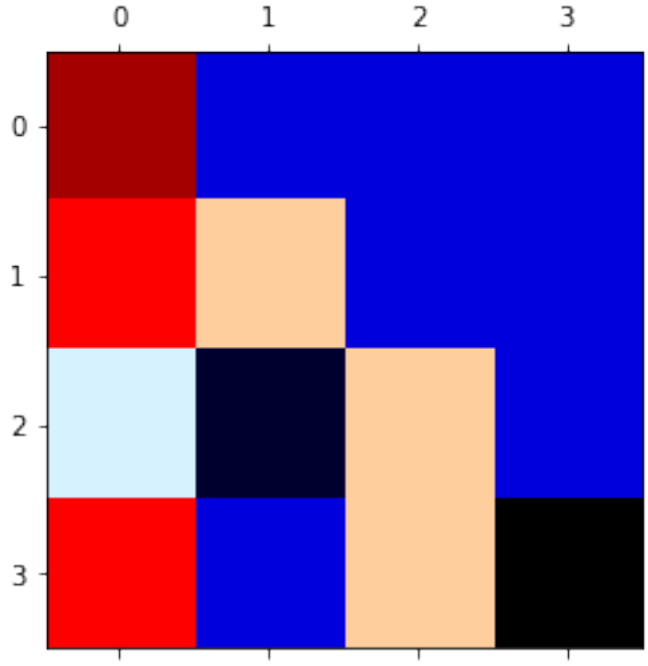
13.3 Cálculo do fator de Cholesky com Python

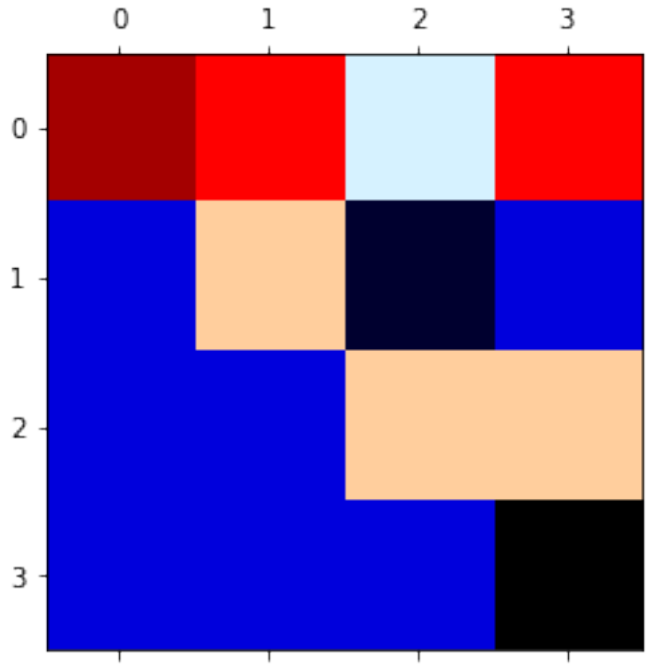
[4]:
```
import matplotlib.pyplot as plt
from scipy import array, linalg, dot, zeros

# 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)

# 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
```







[5]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

14 Implementação do método de Jacobi

[1]:
```
%matplotlib inline
```

[2]:
```
"""
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[:]
    j = 1
    for j in range(1,N):
        X = G.dot(X) + c
        x[:,j] = X

    return x,v
```

[3]:
```
# 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
```

14.0.1 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.

[4]:
```
# plotagem 3D da convergência

import sys
#sys.path.append('/anaconda/lib/python3.7/site-packages/')

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 = 10,color = "rgb(227,26,28)"))

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 =␣
↪10, color = "rgb(84,48,5)"),
                       line = dict( color = "rgb(84,48,5)", width = 10) )

data = [points,vector]
layout = go.Layout(margin = dict( l = 0,r = 0, b = 0, t = 0))
fig = go.Figure(data=data,layout=layout)
iplot(fig, show_link=True,filename='jacobi-3d-vectors')
```

[5]:
```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling();
```

15 Método de Newton para sistemas não-lineares

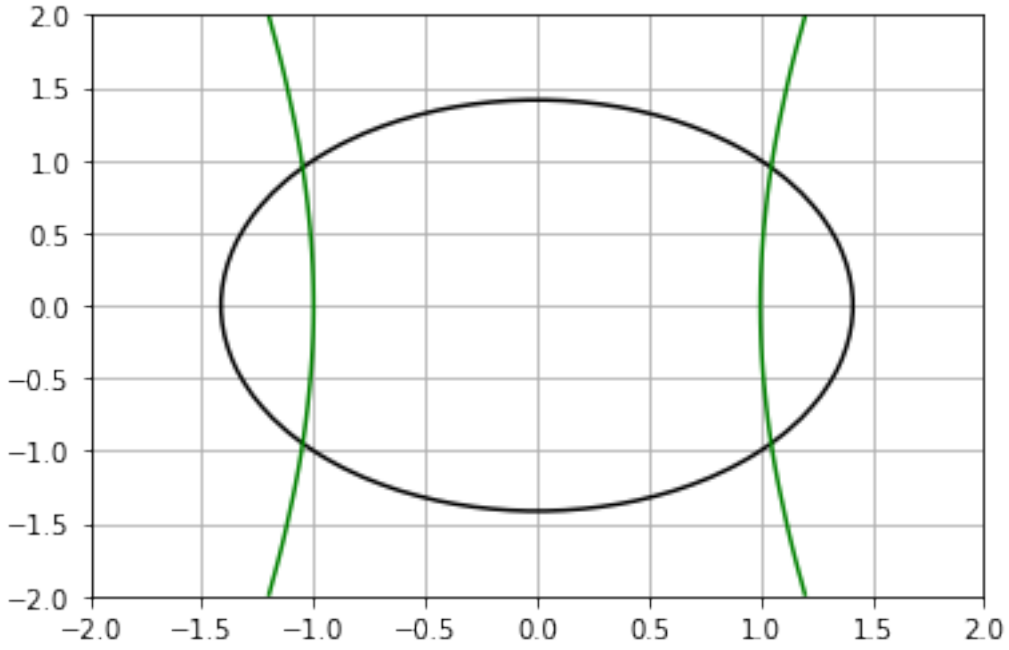
[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import sympy as sy
from scipy.optimize import root
```

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:

[2]:
```
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.

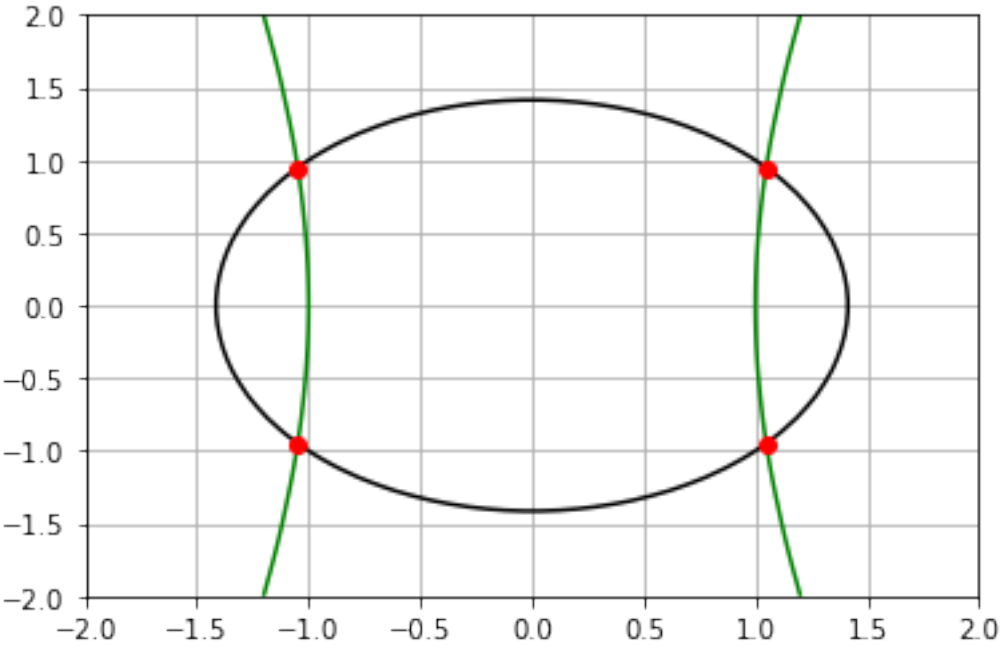
[3]:
```
# define função para o vetor F(x)
def F(x):
    return [ x[0]**2 + x[1]**2 - 2,
             x[0]**2 - x[1]**2/9 - 1 ]


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)
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

[4]:
```
# 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')
```



15.0.1 Exercício:

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

16 Nota: Raízes de sistemas não-lineares

- Uma equação linear tem a forma:

$$f(x) = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n$$

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

$$f_2(x) = a_1 x_1 x_2 + a_2 x_2^2 + a_n x_n x_1$$

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

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

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