Notas de Aula – Métodos Numéricos

Aula 2 – Descrição de dados

Prof. Paulo Ribeiro

Sumário

Nesta aula você aprenderá...

1. a encontrar raízes de equações não lineares usando métodos intervalados (bisseção) e abertos (ponto-fixo, Newton-Raphson, secante);

2. a usar funções da biblioteca `scipy` para calcular essas raízes.

Nessa parte do curso, nosso objetivo é construir aproximações numéricas para a solução de *equações algébricas em uma única variável real.* Observamos que obter uma solução para uma dada equação é equivalente a encontrar um *zero de uma função real* apropriada, isto é, encontrar os valores de x quando satisfazem $f(x) = 0$.

Com isso, iniciamos discutindo condições de existência e unicidade de raízes de funções de uma variável real. Então, apresentamos o **método da bisseção** como uma primeira abordagem numérica para a solução de tais equações.

Em seguida, exploramos outra abordagem via **iteração do ponto fixo**. Essa abordagem é pressuposto para a apresentação da abordagem construída pelo grande Isaac Newton, referenciada como **método de Newton**, para o qual estudamos aplicações e critérios de convergência. Por fim, apresentamos o **método das secantes** como uma das possíveis variações do método de Newton.

Método da Bisseção

O método da bisseção é um método referido como intervalar, pois objetiva, basicamente, encontrar a raíz de uma equação definida por uma função univariada dentro de um intervalo pré-definido.

Nesse ponto, é interessante responder uma pergunta: *como garantir que a raiz da equação está de fato dentro do intervalo desejado?.* A resposta à essa pergunta é dada pelo Teorema de Bolzano, que nos fornece condições suficientes para a existência do zero de uma função, como uma aplicação direta do Teorema do Valor Intermediário (visto com certeza por vocês na disciplina de Cálculo I!)

Teorema 1. ***Teorema de Bolzano*** *- Se $f : [a, b] \to \mathbb{R}$, $y = f(x)$, é uma função contínua tal que $f(a) \cdot f(b) < 0$, condição equivalente a dizer que a função troca de sinal no intervalo, então existe $x^* \in (a, b)$ tal que $f(x^*) = 0$.*

O resultado é uma consequência imediata do teorema do valor intermediário que estabelece que dada uma função contínua $f : [a, b] \to \mathbb{R}$, $y = f(x)$, tal que $f(a) < f(b)$, então para qualquer $k \in (f(b), f(a))$ existe $x^* \in (a, b)$ tal que $f(x^*) = k$. Ou seja, nestas notações, se $f(a) \cdot f(b) < 0$, então $f(a) < 0 < f(b)$. Logo, tomando $k = 0$, temos que existe $x^* \in (a, b)$ tal que $f(x^*) = k = 0$, que é o teorema de Bolzano.

Em outras palavras, se $f(x)$ é uma função contínua em um dado intervalo no qual ela troca de sinal, então ela tem pelo menos um zero neste intervalo. como ilustra a imagem abaixo.

Exemplo 1.

Mostre que existe pelo menos uma solução da equação $e^x = x + 2$ no intervalo $(-2, 0)$.

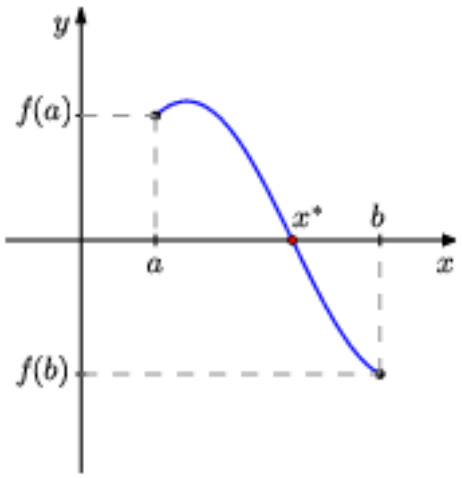


Figura 1

Primeiramente, observamos que resolver a equação $e^x = x+2$ é equivalente a resolver $f(x) = 0$ com $f(x) = e^x - x - 2$. Agora, como $f(-2) = e^{-2} > 0$ e $f(0) = -2 < 0$, temos do teorema de Bolzano que existe pelo menos um zero de $f(x)$ no intervalo $(-2, 0)$. E, portanto, existe pelo menos uma solução da equação dada no intervalo $(-2, 0)$.

O **método da bisseção** explora o fato de que uma função contínua $f : [a, b] \to \mathbb{R}$ com $f(a) \cdot f(b) < 0$ tem um zero no intervalo (a, b). Assim, a ideia para aproximar o zero de uma tal função $f(x)$ é tomar, como aproximação inicial, o ponto médio do intervalo $[a, b]$, isto é

$$x^{(0)} = \frac{(a+b)}{2}. \tag{1}$$

Pode ocorrer de $f(x^{(0)}) = 0$ e, neste caso, o zero de $f(x)$ é $x^* = x^{(0)}$. Caso contrário, se $f(a) \cdot f(x^{(0)}) < 0$, então $x^* \in (a, x^{(0)})$. Neste caso, tomamos como nova aproximação do zero de $f(x)$ o ponto médio do intervalo $[a, x^{(0)}]$, isto é, $x^{(1)} = (a + x^{(0)})/2$. No outro caso, temos $f(x^{(0)}) \cdot f(b) < 0$ e, então, tomamos $x^{(1)} = (x^{(0)} + b)/2$.

Novamente, fazemos o cálculo de $f(x^{(1)})$, e se obtivermos 0, encontramos a raiz, se não, novamente avaliamos os intervalos particionados por $x^{(1)}$, e assim por diante até.... O procedimento é ilustrado na figura abaixo.

De forma mais precisa, suponha que queiramos calcular uma aproximação com uma certa precisão TOL para um zero x^* de uma dada função contínua $f : [a, b] \to \mathbb{R}$ tal que $f(a) \cdot f(b) < 0$. Iniciamos, tomando $n = 0$ e:

$$a^{(n)} = a, \quad b^{(n)} = b \quad \text{e} \quad x^{(n)} = \frac{a^{(n)} + b^{(n)}}{2}. \tag{2}$$

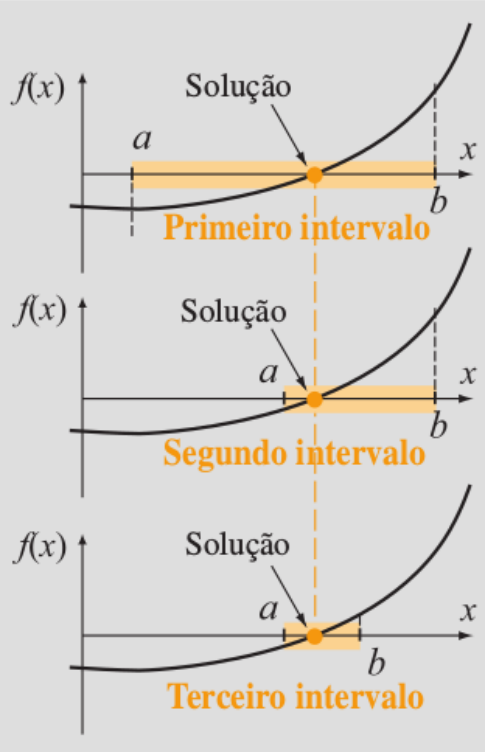


Figura 2

Precisamos estabelecer um **critério de parada** para o algoritmo iterativo.

Naturalmente, o primeiro critério de parada tem de ser $f(x^{(n)}) = 0$. Mas não encontrando exatamente essa condição, pode-se adotar

$$\frac{\left| b^{(n)} - a^{(n)} \right|}{2} < TOL, \tag{3}$$

em que TOL é a tolerância estabelecida na análise da raiz como aceitável.

Caso o critério de parada não seja satisfeito, preparamos a próxima iteração $n+1$ da seguinte forma: se $f(a^{(n)}) \cdot f(x^{(n)}) < 0$, então definimos $a^{(n+1)} = a^{(n)}$ e $b^{(n+1)} = x^{(n)}$; no outro caso, se $f(x^{(n)}) \cdot f(b^{(n)}) < 0$, então definimos $a^{(n+1)} = x^{(n)}$ e $b^{(n+1)} = b^{(n)}$. Trocando n por $n+1$, temos a nova aproximação do zero de $f(x)$ dada por:

$$x^{(n+1)} = \frac{a^{(n+1)} + b^{(n+1)}}{2}. \tag{4}$$

Voltamos a verificar o critério de parada acima e, caso não satisfeito, iteramos novamente. Iteramos até obtermos a aproximação desejada ou o número máximo de iterações ter sido atingido.

Exemplo 2.

Use o método da bisseção para calcular uma solução de $e^x = x + 2$ no intervalo

$[-2, 0]$ **com precisão** $TOL = 10^{-1}$**.**

Primeiramente, observamos que resolver a equação dada é equivalente a calcular o zero de $f(x) = e^x - x - 2$. Além disso, temos $f(-2) \cdot f(0) < 0$. Desta forma, podemos iniciar o método da bisseção tomando o intervalo inicial $[a^{(0)}, b^{(0)}] = [-2, 0]$ e:

$$x^{(0)} = \frac{a^{(0)} + b^{(0)}}{2} = -1. \tag{5}$$

Apresentamos as iterações na tabela abaixo. Observamos que a precisão $TOL = 10^{-1}$ foi obtida aproximando o zero de $f(x)$ por $x^{(4)} = -1,8125$.

n	$a^{(n)}$	$b^{(n)}$	$x^{(n)}$	$f\left(a^{(n)}\right) f\left(x^{(n)}\right)$	$\frac{\left\|b^{(n)} - a^{(n)}\right\|}{2}$
0	-2	0	-1	< 0	1
1	-2	-1	$-1,5$	< 0	0,5
2	-2	$-1,5$	$-1,75$	< 0	0,25
3	-2	$-1,75$	$-1,875$	> 0	0,125
4	$-1,875$	$-1,75$	$-1,8125$	< 0	0,0625

Figura 3

Método da bisseção usando a biblioteca `scipy`

A busca por raízes que solucionem equações não lineares é implementada no Python no módulo `optimize` da biblioteca `scipy`, cujas funções são mostradas na documentação.

O módulo `optimize` provê funções para minimização (ou maximização) de funções objetivo, possivelmente sujeitas à restrições. Ele inclui solucionadores de problemas não lineares (com suporte à algoritmos de otimização global e local), programação linear, mínimos quadrados não lineares e restritos, busca de raiz e ajuste de curva.

Para a solução de equações não lineares (busca de raiz), nosso objetivo aqui, o módulo apresenta diversas funções:

- **brentq:** encontra a raiz de uma equação usando o método de Brent;
- **brenth:** encontra a raiz de uma equação usando o método de Brent com extrapolação hiperbólica;
- **ridder:** encontra a raiz de uma equação usando o método de Ridder;
- **bisect:** encontra a raiz de uma equação usando o método da Bisseção (o estudado nessa sessão);

- **newton:** encontra a raiz de uma equação usando o método de Newton, da secante ou Halley, dependendo dos parâmetros (próximas sessões) ;
- **tom748:** encontra a raiz de uma equação usando o método do algoritmo TOMS 748;
- **fixed_point:** encontra o ponto fixo de uma função (próxima sessão).

Além dessas funções, que implementam alguns dos métodos que consideraremos nesse curso, a partir a versão 1.2.1 da biblioteca `scipy` trouxe, como uma das principais modificações para esse módulo, a implementação da função `root_scalar`, cujo objetivo é encontrar a raiz de uma equação não linear. A diferença dela para as demais funções listadas acima é que ela funciona como uma interface unificada para todos esses métodos (exceto `fixed_point`) em uma função só, isto é, podemos usar a função `root_scalar` para encontrar a raiz de uma função com qualquer dos métodos disponíveis, simplesmente, para tal, escolhando os parâmetros certos. Essa será nossa abordagem nesse documento.

É interessante verificarmos se a versão da biblioteca `scipy` instalada em nossa máquina de fato é a 1.2.1, uma vez que versões anteriores não possuem a função `root_scalar`. Para verificar a versão, basta ver a saída do comando

```
import scipy
scipy.__version__
```

Saída:

```
'1.2.1'
```

Caso a versão não seja igual ou superior à 1.2.1, é preciso que sua biblioteca seja atualizada. Se sua instalação foi feita via anaconda (o que recomendo fortemente), basta então executar o comando `conda update --all` (no linux, pode ser executado em um terminal, como super usuário; em qualquer SO, pode ser rodado diretamente de dentro de um notebook).

Uma vez garantida a versão correta, vamos importar as bibliotecas `numpy` e `scipy`,

```
import numpy as np
import scipy.optimize as opt
```

Para o método da bisseção, a função `root_scalar` possui três argumentos obrigatórios (não *default*, que precisam ser atribuídos pelo usuário): `f`, que define a função da equação que será avaliada, `method`, que define o tipo do método usado pela função, no nosso caso `'bisect'`, e `bracket`, que recebe uma lista com dois valores, referentes aos limites `a` e `b` do intervalo $[a, b]$ no qual a equação será avaliada.

Usaremos como exemplo, pra verificar a usabilidade da função `root_scalar`, o mesmo exemplo estudado no exemplo anterior, em que a equação baseada na função

$$e^x = x + 2 \tag{6}$$

teve suas raízes avaliadas no intervalo $[-2, 0]$.

Lembremos que, para avaliar a função, precisamos escrevê-la no formato $f(x) = 0$; dessa forma, teremos $f(x) = e^x - x - 2 = 0$, que é implementada a seguir.

```
def f(x): return np.exp(x)-x-2
```

Antes de encontrar as raízes, vamos verificar (embora já feito no exemplo acima), a existência de raízes para a citada equação no citado intervalo, usando o Teorema de Bolzano. Para isso, usaremos a função `sign` da biblioteca `numpy`, que retorna o sinal do valor real avaliado. De acordo com a documentação da função (ver link), existem três possíveis resultados: -1, se $x < 0$; 0, se $x = 0$ e 1, se $x > 0$. Assim

```
np.sign(f(-2)*f(0))
```

Saída:

```
-1.0
```

Dessa forma, como o resultado foi -1, significa que $f(-2) \cdot f(0) < 0$, indicando, de acordo com o Teorema de Bolzano, que existe ao menos uma raiz no intervalo assinalado.

Assim, para encontrar a raiz, basta, então, usar

```
raiz = opt.root_scalar(f,method='bisect',bracket=[-2,0])
raiz
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 42
    iterations: 40
          root: -1.8414056604360667
```

Notemos que a saída mostra um conjunto de informações, agrupado em um objeto chamado de `RootResults`, descrito pela sua documentação. Esse objeto possui como atributos mais importantes

- `root`: valor da raiz encontrada;
- `iterations`: número de iterações para encontrar a raiz;
- `function_calls`: número de vezes que a função foi chamada durante os cálculos;
- `converged`: indica se o método de fato convergiu e encontrou a raiz;

- `flag`: causa da parada do algoritmo.

Dessa forma, o que vemos na saída é que o método convergiu e encontrou a raiz, de valor -1.8414056604360667, após 40 iterações, durante as quais a função foi chamada 42 vezes.

Nós podemos acessar os valores de cada atributo desse individualmente, bastanto, para isso, colocar o nome do atributo após um ponto (.) após o nome da variável que recebeu os resultados da função.

Dessa forma, para saber somente o resultado da raiz, basta fazer

```
raiz.root
```

Saída:

```
−1.8414056604360667
```

e, para saber o número de iterações, basta fazer

```
raiz.iterations
```

Saída:

```
40
```

seguindo esse procedimento para qualquer um dos possíveis atributos do objeto `RootResults`.

Podemos ainda aplicar esse mesmo procedimento diretamente na função,

```
opt.root_scalar(f,method='bisect',bracket=[−2,0]).root
```

Saída:

```
−1.8414056604360667
```

obtendo o mesmo resultado.

Analisemos, agora, o valor encontrado pela função.

Pode ser visto que o resultado difere um pouco do que conseguimos no exemplo trabalhado acima, em que a raiz encontrada foi -1.8125.

Essa diferença tem haver com o critério de parada adotado, no caso, a precisão (ou tolerância), que no exemplo foi considerado como 10^{-1}.

Existem dois parâmetros na função `root_scalar` para a definição da tolerância: `xtol`, que define a tolerância absoluta e `rtol`, que define a tolerância relativa como critério de parada. Vamos testar os dois

```
opt.root_scalar(f,method='bisect',bracket=[−2,0], xtol=1e−1)
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 7
    iterations: 5
          root: −1.8125
```

```
opt.root_scalar(f,method='bisect',bracket=[−2,0], rtol=1e−1)
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 6
    iterations: 4
          root: −1.875
```

Notemos que as duas raízes são diferentes e, ainda mais, quando adotamos como critério `rtol`, a convergência acontece com uma iteração a menos do que quando usamos `xtol`. Porque isso acontece?

Se a escolha for `xtol`, a função testa, como critério de parada, se o erro absoluto é menor que o valor do parâmetro, isto é, se

$$\varepsilon_a = |x_{i+1} - x_i| = \frac{|b_i - a_i|}{2} \leq \text{xtol}, \tag{7}$$

ao passo que, se a escolha for `rtol`, a função testa se o erro relativo é menor que o valor do parâmetro, isto é, se

$$\varepsilon_r = \frac{|x_{i+1} - x_i|}{x_{i+1}} \leq \text{rtol}. \tag{8}$$

A escolha de que parâmetro escolher como critério de parada fica, literalmente, a gosto do "freguês".

Mas o que acontece se usarmos os dois, com mesmo valor, no mesmo cálculo?

```
opt.root_scalar(f,method='bisect',bracket=[−2,0], rtol=1e−1, xtol=1e−1)
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 5
    iterations: 3
          root: −1.75
```

Deixo a critério do aluno interpretar esse resultado.

Um outro parâmetro importante é o `maxiter`, que define o número máximo de

iterações como critério de parada do algoritmo. Ou seja, podemos estabelecer dois critérios objetivos de parada: número máximo de iterações e tolerância. A função irá parar as iterações assim que um dois dois critérios for atingido primeiro. Vejamos um exemplo

```
opt.root_scalar(f,method='bisect',bracket=[-2,0], xtol=1e-10, maxiter=100)
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 37
    iterations: 35
          root: -1.8414056604378857
```

Como vemos, com uma tolerância de 10^{-10}, e um número máximo de 100 iterações, nossa saída mostra uma convergência com apenas 35 iterações. Dessa forma, podemos afirmar que a parada se deu porque o critério da tolerância foi atingido antes do critério do número máximo de iterações.

Vamos agora repetir o teste, porém colocando um número máximo de 30 iterações.

```
opt.root_scalar(f,method='bisect',bracket=[-2,0], xtol=1e-10, maxiter=30)
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 32
    iterations: 30
          root: -1.8414056617766619
```

Exemplo 3.

Considere a equação $\sqrt{x} = \cos(x)$. Use o método da bisseção com intervalo inicial $[a, b] = [0, 1]$ para calcular e mostrar os resultados de todas as aproximações até $n = 6$ iterações da solução desta equação. Formate a saída de forma similar ao exemplo feito em sala.

```
def f(x): return np.cos(x)- np.sqrt(x)
```

Para fins de visualização, vamos colocar nossos resultados em um `dataframe` da biblioteca `pandas`.

```
vp = opt.root_scalar(f,method='bisect',bracket=[0,1], xtol=1e-40).root
```

```
r = []
```

```
e = []
for i in range(1,7):
    x = opt.root_scalar(f,method='bisect',bracket=[0,1], maxiter=i).root
    r.append(x)
    e.append(np.around(np.absolute(x − vp)*100/2, decimals=2))

import pandas as pd
pd.DataFrame({'Iterações': range(0,6), 'Raiz': r, 'Erro':e}).set_index('Iterações')
```

Saída:

Iterações	Raiz	Erro
0	0.500000	7.09
1	0.500000	7.09
2	0.625000	0.84
3	0.625000	0.84
4	0.625000	0.84
5	0.640625	0.05

Exemplo 4.

Trace o gráfico e isole as três primeiras raízes positivas da função:

$$f(x) = 5\sin(x^2) - \exp\left(\frac{x}{10}\right)$$

em intervalos de comprimento 0.1**. Então, use o método da bisseção para obter aproximações dos zeros desta função com precisão de** 10^{-5}**.**

Para traçar o gráfico, usaremos o módulo `matplotlib`, com o atributo `%matplotlib inline` para que o gráfico apareça *inline*.

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

Definindo a função, temos

```
def f(x): return 5*np.sin(x**2)−np.exp(x/10)
```

E, para traçar o gráfico, temos

```
x = np.linspace(0,2.7,200)
plt.plot(x,f(x))
```

```
plt.plot([0.46, 1.7, 2.56], [f(0.46), f(1.7), f(2.56)], 'o', color='r')
plt.grid(True)
```

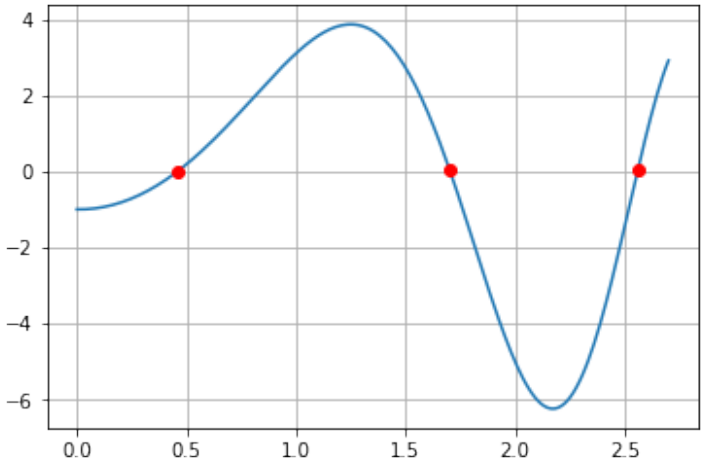


Figura 4

Após testar algumas resoluções para poder ver a curva, pode-se estimar que as três primeiras raízes positivas se encontram, é possível observar que temos uma raiz próxima a 0.5, uma próxima a 1.6 e outra próxima a 2.5.

Dessa forma, definirei como três intervalos para testar $[0.4, 0.5]$, $[1.7, 1.8]$ e $[2.5, 2.6]$.

Nessa caso, precisamos, antes, verificar se as raízes estão, de fato, nesses intervalos e podemos fazer isso aplicando o teste a partir do Teorema de Bolzano.

Dessa forma, temos:

```
np.sign(f(0.4)*f(0.5))
```

Saída:

```
−1.0
```

```
np.sign(f(1.7)*f(1.8))
```

Saída:

```
−1.0
```

```
np.sign(f(2.5)*f(2.6))
```

Saída:

```
−1.0
```

Pelo resultado, vemos que as três raízes se encontram, realmente, nos intervalos estipulados.

Dessa forma, podemos estimar a raíz pelo método da bisseção, com precisão de 10^{-5}, fazendo:

```
print('1ª raiz positiva: ',opt.root_scalar(f,method='bisect',
    bracket=[0.4,0.5],xtol=1e-5).root)
print('2ª raiz positiva: ',opt.root_scalar(f,method='bisect',
    bracket=[1.7,1.8],xtol=1e-5).root)
print('3ª raiz positiva: ',opt.root_scalar(f,method='bisect',
    bracket=[2.5,2.6],xtol=1e-5).root)
```

Saída:

```
1ª raiz positiva: 0.459307861328125
2ª raiz positiva: 1.703570556640625
3ª raiz positiva: 2.558209228515625
```

Exemplo 5.

O desenho abaixo mostra um circuito não linear envolvendo uma fonte de tensão constante, um diodo retificador e um resistor. Sabendo que a relação entre a corrente (I_d) e a tensão (v_d) no diodo é dada pela seguinte expressão:

$$I_d = I_R \left[\exp\left(\frac{v_d}{v_t}\right) - 1 \right], \tag{9}$$

em que I_R é a corrente de condução reversa e v_t, a tensão térmica dada por $v_t = \frac{kT}{q}$ com k, a constante de Boltzmann, T a temperatura de operação e q, a carga do elétron. Aqui $I_R = 1pA = 10^{-12}$ A, $T = 300$ K. Escreva o problema como uma equação na incógnita v_d e, usando o método da bisseção, resolva este problema com 3 algarismos significativos para os seguintes casos:

a) $v_s = 30$ V e $R = 1$ kΩ.

b) $v_s = 3$ V e $R = 1$ kΩ.

c) $v_s = 3$ V e $R = 10$ kΩ.

d) $v_s = 300$ mV e $R = 1$ kΩ.

e) $v_s = -300$ mV e $R = 1$ kΩ.

f) $v_s = -30$ V e $R = 1$ kΩ.

g) $v_s = -30$ V e $R = 10$ kΩ.

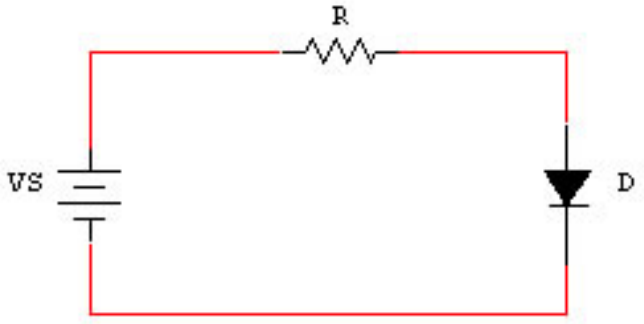


Figura 5

Olhando para o circuito, que faz uma divisão de tensão entre o resistor e o diodo, podemos verificar que

$$v_s = v_r + V_d \quad \Rightarrow \quad v_r = v_s - v_d \tag{10}$$

e (11)

$$I_d = I_r = \frac{v_r}{R} = \frac{v_s - v_d}{R}, \tag{12}$$

atentando que I_r é a corrente sobre o resistor R e I_R é a corrente reversa do diodo.

Dessa forma, podemos reescrever a equação dada no problema como:

$$I_d = I_R \left[\exp\left(\frac{v_d}{v_t}\right) - 1\right] \quad \Rightarrow \tag{13}$$

$$\frac{v_s - v_d}{R} = I_R \left[\exp\left(\frac{v_d}{v_t}\right) - 1\right] \quad \Rightarrow \tag{14}$$

$$\underbrace{I_R \left[\exp\left(\frac{v_d}{v_t}\right) - 1\right] - \left(\frac{v_s - v_d}{R}\right)}_{f(v_d)} = 0 \tag{15}$$

(16)

Substituindo os valores das constantes I_R e v_t, temos, então

$$f(v_d) = 10^{-12} \cdot \left[\exp\left(\frac{1,60217653 \cdot 10^{-19} \cdot v_d}{4,1419509 \cdot 10^{-25}}\right) - 1\right] - \left(\frac{v_s - v_d}{R}\right) = 0 \qquad (17)$$

Escrevendo a função no python, temos, para o primeiro caso:

```
def fa(vd, vs = 30, R = 1e+3): return 1e−12 * np.exp(((1.60217653e−19 *
    vd)/(4.1419509e−25))−1) − ((vs − vd)/(R))
```

Colocamos apenas um parâmetro *default*, para que possamos variá-los de acordo com os casos pedidos no problema. Pegaremos somente um dos casos, o primeiro, para verificar o funcionamento de nossa implementação.

Para encontrar o intervalo, podemos fazer uma inspeção visual no gráfico da função para o primeiro caso, para podermos verificar onde aproximadamente a raiz se localiza.

Dessa forma,

```
x = np.linspace(0,1,1000)
plt.plot(x,fa(x))
plt.grid(True)
```

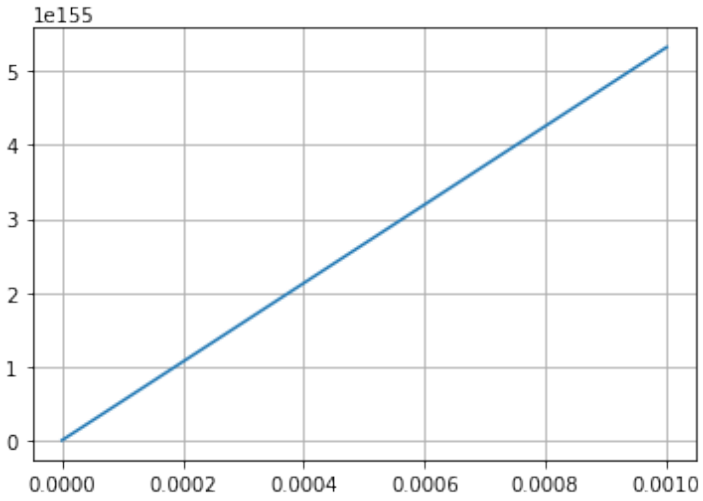


Figura 6

Assim, usando o método da bisseção, obtemos:

```
opt.root_scalar(fa,method='bisect', bracket=[0,0.1]).root
```

Saída:

```
6.495182024082167e−05
```

Repetindo o procedimento para o caso d), para fins de teste, temos:

```
def fd(vd, vs = 300e-3, R = 1e+3): return 1e-12 * np.exp(((1.60217653e-19 *
    vd)/(4.1419509e-25))-1) - ((vs - vd)/(R))


x = np.linspace(0,.1,1000)
plt.plot(x,fd(x))
plt.grid(True)
```

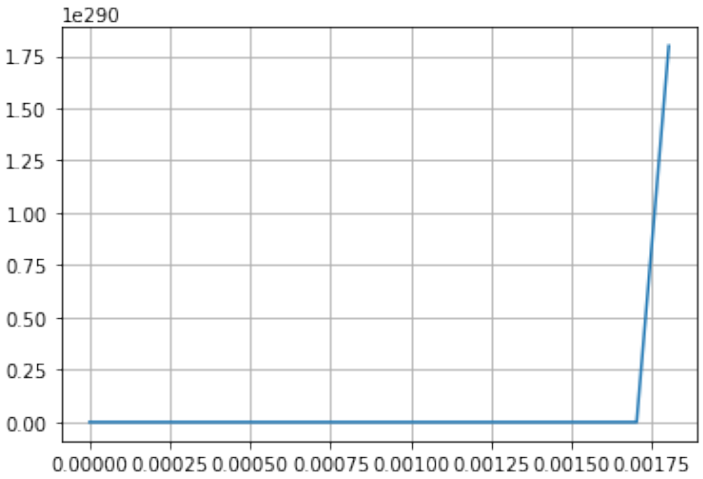


Figura 7

```
opt.root_scalar(fd,method='bisect', bracket=[0,0.1]).root
```

Saída:

```
5.304607184370981e-05
```

Ótimo, faz sentido!

Mas como podemos fazer para não ter de repetir função por função, o que é pouquíssimo produtivo?

Podemos, aqui, usar o parâmetro `args` da função `root_scalar`. Basicamente, esse argumento recebe uma tupla com os parâmetros extras da função, que será chamada no formato `(f, (x)+args)`.

Assim, se fizermos a função de forma genérica

```
def f(vd, vs, R): return 1e-12 * np.exp(((1.60217653e-19 * vd)/(4.1419509e-25))-1)
    - ((vs - vd)/(R))
```

e, então, usarmos o parâmetro `args`, temos

```
vs = 300e-3
```

```
R = 1e+3
opt.root_scalar(f, args=(vs,R), method='bisect', bracket=[0,0.1]).root
```

Saída:

```
5.304607184370981e-05
```

que é o mesmo resultado visto anteriormente.

Dessa forma, para resolver para todos os casos, podemos fazer

```
vs = np.array([30, 3, 3, 300e-3, -300e-3, -30, -30])
R = np.array([1e+3, 1e+3, 1e+4, 1e+3, 1e+3, 1e+3, 1e+4])
```

A escolha dos intervalos deve ser usada na base do teste, verificando as características, como feito nos casos específicos analisados anteriormente. Nos testes, verificamos que, para casos em que a tensão da fonte é positiva (diodo em condução), existe troca de sinal no intervalo $[0, 0.1]$, mas, para tensões negativas, só existirá mudança de sinal para valores menores que -30 e próximos de 0 (o que condiz com a teoria). Assim, escolhemos o intervalo $[-32, 0.1]$ para analisar os casos em que a tensão da fonte é negativa (diodo não conduz).

```
vd = []
for i in range(len(vs)):
    Vs = vs[i]
    r = R[i]
    if Vs>0:
        a,b = 0,.1
    else:
        a,b = -32,.1
    x = opt.root_scalar(f, args=(Vs,r), method='bisect', bracket=[a,b]).root
    vd.append(x)
vd
```

Saída:

```
[6.495182024082167e-05,
 5.899912648601458e-05,
 5.3046482207719243e-05,
 5.304607184370981e-05,
 -0.299999999999072,
 -30.000000000000098,
 -30.000000000000098]
```

Podemos, assim, plotar um gráfico de $v_d \times v_s$ e ver a relação entre as variáveis.

```
plt.plot(vs,vd)
plt.xlabel('$V_S (V)$')
plt.ylabel('$V_d (V)$')
```

Saída:

```
Text(0, 0.5, '$V_d (V)$')
```

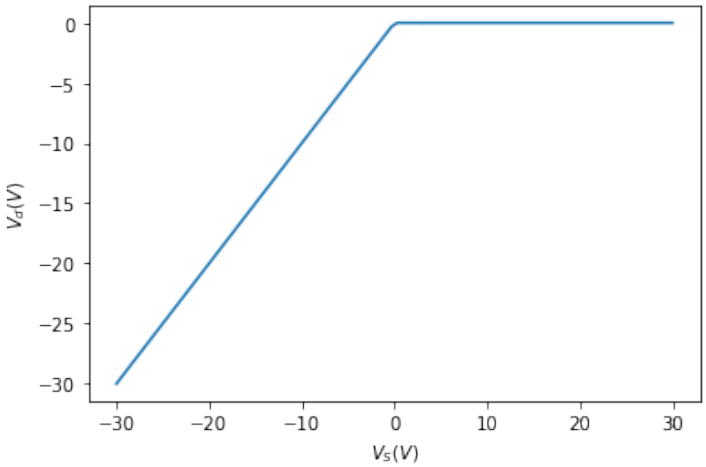


Figura 8

A título de conferência, podemos verificar a curva característica do diodo, plotando o gráfico $I_d \times V_d$, para verificar se o modelo condiz com a realidade.

Dessa forma, calculando I_d diretamente dos vetores de v_d, v_s e de R, temos

```
id = (vs - vd)*1000/R
id
```

Saída:

```
array([ 2.99999350e+01, 2.99994100e+00, 2.99994695e-01, 2.99946954e-01,
       -9.27979915e-13, 9.80548975e-13, 9.80548975e-14])
```

```
plt.plot(vd,id)
plt.xlabel('$V_d (V)$')
plt.ylabel('$I_d (mA)$')
```

Saída:

```
Text(0, 0.5, '$I_d (mA)$')
```

que condiz totalmente com a teoria!

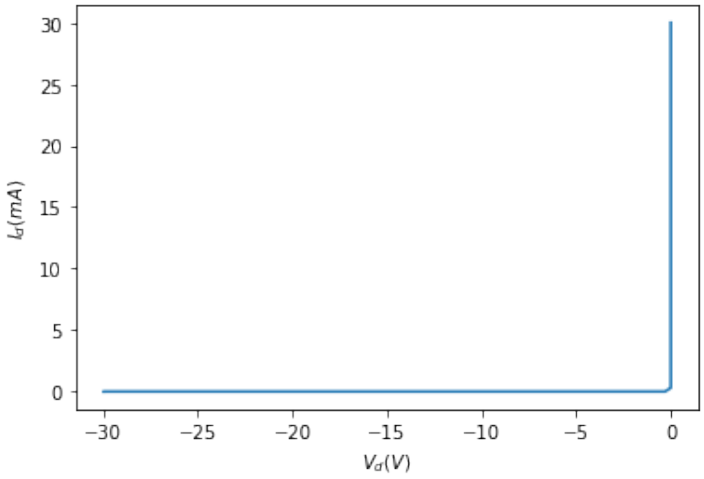


Figura 9

Método dos Pontos Fixos

No método da bisseção, no qual escolhíamos um dado intervalo e, a partir dele, íamos nos aproximando da raiz a cada iteração, até encontrar uma aproximação que fosse aceitável de acordo com os critérios de parada adotados.

Nessa sessão veremos um conjunto de métodos, ditos abertos, baseados em fórmulas que testa um ou dois valores específicos, que não necessariamente limitam a raiz. Esses valores são chamados de **pontos fixos**.

A grande vantagem desses métodos é a convergência mais rápida, se comparados com o método da bisseção. Porém, diferente desse, não existe garantia de convergência, podendo, em alguns casos, que os cálculos divirjam, sendo levados para valores longe da raiz da equação avaliada.

Mas o que é um ponto fixo?

Pontos fixos são pontos de uma função que não são alterados por uma aplicação, isto é, pontos em que $y = x$, ou , como $y = f(x)$, em que $f(x) = x$.

Visualmente, pode-se identificar esses pontos, para funções univariadas, traçando-se a reta em que $y = x$ e verificando os pontos em que essa reta corta a curva da função $f(x)$, como mostrado na figura a seguir:

Exemplo 1.

Seja a função $f : \mathbb{R} \to \mathbb{R}$, definida por $f(x) = x^3$. Encontre os três pontos fixos dessa função.

Por tentativa e erro é fácil verificar que $x = 1$, $x = 0$ e $x = -1$ são pontos fixos de

$f(x)$, pois satisfazem $f(x) = x$, uma vez que $f(1) = 1$, $f(0) = 0$ e $f(-1) = -1$. Mas existem mais pontos fixos reais para essa função?

Essa dúvida pode ser sanada pela inspeção do gráfico da função sobreposto pelo gráfico da reta $y = x$,

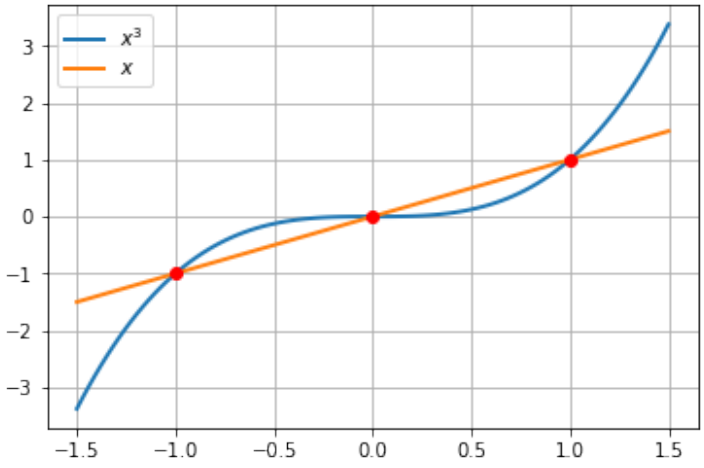


Figura 10

onde se vê claramente que não há mais nenhum ponto de encontro entre a reta $y = x$ e a função $f(x) = x^3$ além dos três pontos assinalados.

O código para geração desse gráfico está listado abaixo.

```
def f1(x): return x**3
def f2(x): return x
x = np.linspace(−1.5,1.5,100)
plt.plot(x,f1(x), linewidth=2, label='$x^3$')
plt.plot(x,f2(x), linewidth=2, label='$x$')

plt.plot([−1.0, 0, 1], [f1(−1.0), f1(0), f1(1.0)], color='r',marker='o')

plt.grid()
plt.legend()
```

Saída:

```
<matplotlib.legend.Legend at 0x7f367ace1978>
```

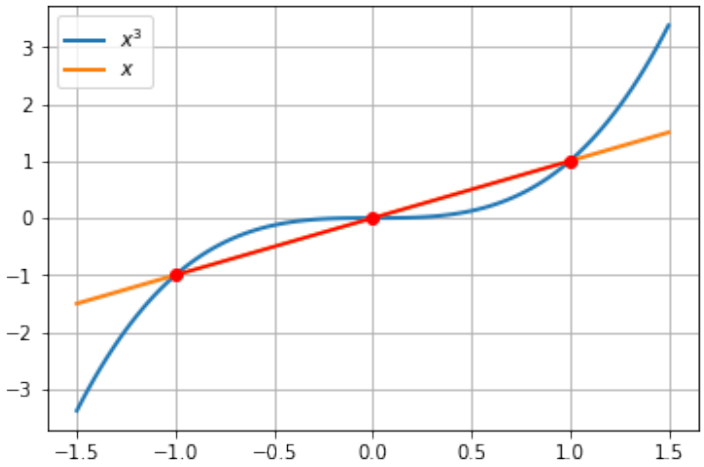


Figura 11

Método da Iteração de Ponto Fixo

Como dito anteriormente, nos métodos abertos, um valor inicial é avaliado para uma fórmula específica, de acordo com cada método, e, após uma certa quantidade de iterações, se encontra o valor da raiz.

No caso da Iteração de Ponto Fixo, a fórmula usada para resolver uma equação $f(x) = 0$ é determinada, basicamente, pelo isolamento do x da equação, de forma que

$$x = g(x) \tag{18}$$

seja a função a avaliar cada valor testado, em que x é o de **ponto fixo** da função $f(x)$.

Um dos resultados diretos do Teorema do Ponto Fixo de Banach (tipicamente visto em disciplinas de análise real e funcional, cuja complexidade e escopo fogem ao esperado para essa disciplina) é que, a partir de uma sequência de valores $x_0, x_1, x_2, \ldots, x_k$, é possível determinar o valor do ponto fixo p, de forma iterativa, considerando que

$$\lim_{k\to\infty} x_k = p, \tag{19}$$

isto é, se obtivermos iterativamente valores de x que satisfaçam a relação (1), esse valor tenderá a ser o ponto fixo da função à medida que o número de iterações tende ao infinito.

Assim, tomando o valor inicial x_0 como o valor a ser considerado na primeira iteração, podemos estimar cada valor do(s) ponto(s) fixo(s), de forma iterativa, da seguinte forma

$$x_{i+1} = g(x_i), \tag{20}$$

garantindo, pelo resultado descrito acima, que se o número de iterações for suficiente (atender ao critério de parada estabelecido), a k-ésima iteração terá o valor do ponto fixo desejado.

Se essa sequência de valores for convergente, então, a estimação do ponto fixo encontrada será também a raiz de $f(x) = 0$.

O critério de parada desse método pode se dar pela medida do erro relativo

$$\varepsilon_r = \left| \frac{x_{i+1} - x_i}{x_{i+1}} \right|, \tag{21}$$

ou pelo número de iterações.

Importante notar que, mesmo que um ou os dois critérios acima sejam satisfeitos, não há garantia de convergência.

Fica ainda uma pergunta: como determinar a função $g(x)$?

Basicamente, a ideia é conseguir escrever a relação $x = g(x)$ a partir da equação a ser avaliada $f(x) = 0$. Vejamos alguns exemplos.

Exemplo 2.

Considere a equação $x^2 - 2x + 3 = 0$. Qual a fórmula a ser usada pelo método da Iteração de Ponto fixo para a busca de raiz?

Basta isolar o x, obtendo

$$x^2 - 2x = -3 \quad \Rightarrow \quad x(x-2) = -3 \quad \Rightarrow \quad x = \frac{-3}{x-2}. \tag{22}$$

É interessante perceber que essa mesma relação poderia ser escrita de outra forma, como

$$x^2 - 2x + 3 = 0 \quad \Rightarrow \quad x^2 + 3 = 2x \quad \Rightarrow \quad x = \frac{x^2+3}{2}. \tag{23}$$

Temos, então, duas possíveis expressões. Mas, qual escolher?

Nesse caso precisamos atentar para limitações que cada arranjo da função possui. No equação (5), temos a limitação de, por exemplo, se $x_i = 2$, encontrarmos uma divergência, pois $x_{i+1} \to \infty$. Logo, é mais interessante usar o arranjo (6), que não tem essa inconsistência.

Exemplo 3.

Considere a equação $\sin(x) = 0$. Qual a fórmula a ser usada pelo método da Iteração de Ponto fixo para a busca de raiz?

Uma possível e simples solução é somar x à ambos os lados da equação. Dessa forma,

$$x = \sin(x) + x. \tag{24}$$

O exemplo a seguir ilustra a aplicação do método da Iteração do Ponto Fixo para encontrar a raiz de uma equação.

Exemplo 4.

(Chapra - Exep. 6.1) Cálcule a raiz da equação $e^{-x} - x = 0$ pelo método da iteração de ponto fixo, com valor inicial $x_0 = 0$.

Nesse exemplo, podemos fazer usar a fórmula $x = e^{-x}$ para avaliar as iterações, de forma que, para cada iteração i,

$$x_{i+1} = e^{-x_i}. \tag{25}$$

Dessa forma, temos, para cada iteração,

i	x_i	ε_a (%)
0	0	
1	1.000000	100.0
2	0.367879	171.8
3	0.692201	46.9
4	0.500473	38.3
5	0.606244	17.4
6	0.545396	11.2
7	0.579612	5.90
8	0.560115	3.48
9	0.571143	1.93
10	0.564879	1.11

Figura 12

No livro, o autor afirma que o valor verdadeiro da raiz é 0.56714329.

Método do Ponto Fixo no Python

Como dito no material sobre método da bisseção, as funções que implementam métodos de localização de raízes em equações estão localizadas no submódulo `optimize` e, olhando a documentação, é possível ver que existe uma função `fixed_point` que cálcula o ponto fixo de uma função. Vamos também que, diferente do método anterior, esse não consta como um dos métodos considerados pela função `root_scalar`.

Olhando a documentação, é possível ver que a função `fixed_point` possui dois parâmetros obrigatórios: `func`, a função a ser avaliada $g(x)$, e `x0`, o valor inicial para início das iterações. Vamor ver como ela trabalha para o exemplo anterior:

```
def g(x): return np.exp(-x)


opt.fixed_point(g, 0)
```

Saída:

```
array(0.56714329)
```

Vemos que esse valor bate certinho com o resultado apontado no exemplo do livro do Chapra. Um indicativo de que o método funciona.

Mas podemos (e devemos) explorar outros argumentos da função `fixed_point`.

Além desses argumentos, obrigatórios, existem também os argumentos `xtol`, que define a tolerância ou precisão esperada como critério de parada e `maxiter`, que define o número máximo de iterações, similares ao que vimos no método da bisseção. Dessa forma, se avaliarmos a função com tolerância de 10^{-1}, obteremos

```
opt.fixed_point(g, 0, xtol=1e-1)
```

Saída:

```
array(0.56735086)
```

Ao conferir esses resultados com o do exemplo acima do Chapra, algo estranho nos aparece: não existe esse valor em nenhuma iteração. Isso, obviamente, tem explicação.

Na documentação, se dermos uma analisada no parâmetro `method` da função `fixed_point`, veremos que existem dois possíveis valores para esse argumento: `'del2'`, o parâmetro *default*, que habilita a função a trabalhar com uma variante do método que leva à uma aceleração de convergência, chamado de método de Steffensen (esse método é descrito em alguns livros de Análise Numérica, como o famoso livro do Burden; nesse curso, não intencionamos discutí-lo.) e `'iteration'`, que faz com que a função simplesmente faça as iterações como fizemos no exemplo anterior.

De fato, se escolhermos essa última, teremos:

```
opt.fixed_point(g, 0, xtol=1e-1, method='iteration')
```

Saída:

```
0.5796123355033789
```

que equivale a 7ª iteração no exemplo trabalhado (ver tabela no exemplo 4), cujo erro relativo, de fato, é inferior à 10%. O parâmetro `maxiter` funciona de forma similar à função `root_scalar`, não havendo necessidade de maior discussão, salvo o alerta de que, caso o número de iterações escolhidos acabe antes do método convergir, haverá um erro e que, diferente da função `root_scalar`, não existe um parâmetro `disp` que nos permita "passar por cima do erro".

Por exemplo, se ajustarmos como critério para parada 7 iterações, com a tolerância de 10^{-1}, obteremos o resultado, justamente porque essa tolerância é alcançada na 7ª iteração (ver tabela do exemplo anterior).

```
opt.fixed_point(g, 0, xtol=1e-1, method="iteration", maxiter=7)
```

Saída:

```
0.5796123355033789
```

Mas, se ajustarmos para apenas 5 iterações, o critério da tolerância não será satisfeito e, portanto, não haverá convergência, gerando erro.

```
opt.fixed_point(g, 0, xtol=1e-1, method="iteration", maxiter=5)
```

Saída:

```
---------------------------------------------------------------
RuntimeError Traceback (most recent call last)
<ipython-input-43-32170d17ef0a> in <module>
----> 1 opt.fixed_point(g, 0, xtol=1e-1, method="iteration", maxiter=5)
/opt/anaconda3/lib/python3.6/site-packages/scipy/optimize/minpack.py in
fixed_point(func, x0, args, xtol, maxiter, method)
    893 use_accel = {'del2': True, 'iteration': False}[method]
    894 x0 = _asarray_validated(x0, as_inexact=True)
--> 895 return _fixed_point_helper(func, x0, args, xtol, maxiter, use_accel)
/opt/anaconda3/lib/python3.6/site-packages/scipy/optimize/minpack.py in
_fixed_point_helper(func, x0, args, xtol, maxiter, use_accel)
    847 p0 = p
    848 msg = "Failed to converge after %d iterations, value is %s" % (maxiter, p)
--> 849 raise RuntimeError(msg)
    850
    851
RuntimeError: Failed to converge after 5 iterations, value is 0.6062435350855974
```

Mesmo assim, apesar disso, na mensagem de erro é possível ver o valor obtido na última iteração.

Exemplo 5.

Encontre a raiz da equação $\cos(x) = 0$ usando o método da Iteração de Ponto Fixo. Considere uma tolerância de 10^{-2}, e escolha qual valor inicial usar para aplicação do método, justificando sua escolha.

Primeiro implementamos a equação no formato $g(x) = x$, de forma similar ao exemplo 3 acima. Assim, basta escolher $\cos(x) + x = x$, ou seja, fazer $g(x) = \cos(x) + x$. Como ponto de partida x_0, podemos escolher, pela própria definição da função cosseno, $x_0 = \pi/2$.

Dessa forma, aplicando à função `fixed_point` do `scipy`, obtemos:

```
def g(x): return np.cos(x) + x
opt.fixed_point(g, np.pi/2, xtol=1e−2)
```

Saída:

```
array(1.57079633)
```

Método de Newton-Raphson

O método de Newton-Raphson parte de definição de que a primeira derivada de uma função é o coeficiente angular da reta tangente ao valor dessa função em um ponto específico, como ilustrado abaixo.

Ele usa essa definição da reta tangente para gerar uma sequência $x_0, x_1, x_2, \ldots, x_n, \ldots$ de aproximações para a raiz, usando a definição da primeira derivada, temos:

$$f'(x) = \frac{f(x_i) - 0}{x_i - x_{i+1}}. \tag{26}$$

A partir do conceito do ponto fixo, definido anteriormente, nós podemos estimar a raiz, de forma iterativa, se usarmos a fórmula, obtida do rearranjo da equação anterior,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \text{se } f'(x_i) > 0. \tag{27}$$

A equação 27 é chamada de *fórmula de Newton-Raphson*, e, atendendo ao critério de parada, similar ao do método do Ponto Iterativo, podemos estimar de forma relativamente

rápida a raiz da função.

Exemplo 1.

(Chapra, exemplo 6.3) Cálcule a raiz da equação $e^{-x} - x = 0$ pelo método de Newton-Raphson, com valor inicial $x_0 = 0$.

A primeira derivada da função $f(x)$ é

$$f'(x) = -e^{-x} - 1. \tag{28}$$

Dessa forma, temos que a fórmula de Newton-Raphson para essa função é dada por

$$x_{i+1} = x_i - \frac{e^{-x_i} - x_i}{-e^{-x_i} - 1}. \tag{29}$$

Começando com o valor inicial $x_0 = 0$, obtemos, então:

i	x_i	ε_t (%)
0	0	100
1	0.500000000	11.8
2	0.566311003	0.147
3	0.567143165	0.0000220
4	0.567143290	$< 10^{-8}$

Figura 13

Método de Newton-Raphson usando Python

A função `root_scalar` também implementa o método de Newton-Raphson, bastando para isso usar, além do parâmetro `f`, referente à função analisada, os parâmetros `fprime`, referente a primeira derivada da função atribuída à `f`, `x0`, referente ao valor inicial x_0 e `method`, cujo valor atribuído é `'newton'`.

Vejamos um exemplo.

Exemplo 2.

Calcular a raiz do exemplo anterior usando a função *root_ scalar* com o método de Newton-Raphson.

Antes de tudo, precisamos definir a função $f(x)$ e sua primeira derivada, $f'(x)$.

```
def f1(x): return np.exp(−x) − x #função
def f2(x): return −np.exp(−x) − 1 #1ª derivada
```

Em seguida, ajustar os parâmetros citados para os mesmos valores do exemplo e calcular a raiz.

```
opt.root_scalar(f1, fprime=f2, x0=0, method='newton')
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 10
    iterations: 5
          root: 0.567143290409784
```

Método da Secante

O método da Newton-Raphson, embora geralmente apresente uma rápida convergência se comparado com os métodos intervalados, apresenta algumas limitações. Um problema potencial é o fato de que nem sempre é uma tarefa simples encontrar a derivada de uma função.

Nesses casos, a derivada pode ser aproximada por uma diferença dividida regressiva, do tipo

$$f'(x) \approx \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}. \tag{30}$$

Substituindo essa aproximação na equação (27), obtemos

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}, \tag{31}$$

conhecida como a fórmula do *Método da Secante*.

É necessário atentar que, diferente dos dois métodos anteriores, é necessário partir de duas suposições iniciais, x_{-1} e x_0. Porém, como não existe a necessidade de garantir a mudança de sinal de $f(x)$ entre as estimativas, esse método não pode ser classificado como um método intervalar.

O critério de parada é o mesmo dos outros dois métodos e, assim como ambos, não há garantias de convergência.

Exemplo 1.

Encontre as raízes de $f(x) = \cos(x) - x$.

Da inspeção do gráfico das funções $y = \cos(x)$ e $y = x$, sabemos que esta equação possui uma raiz em torno de $x = 0,8$. Iniciamos o método com $x_{-1} = 0,7$ e $x_0 = 0,8$.

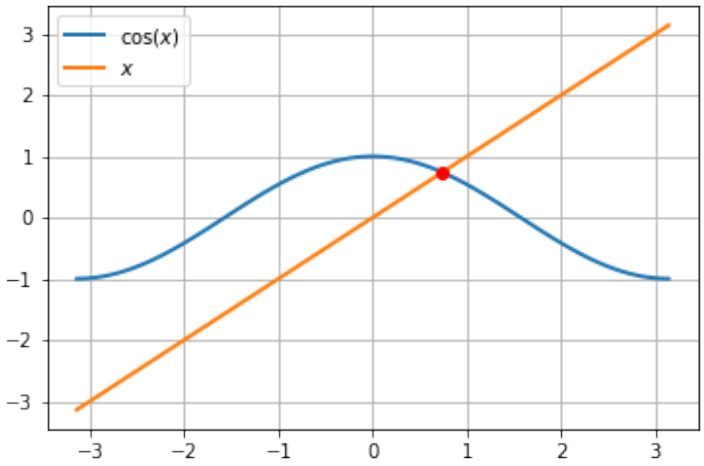


Figura 14

$x_{(i-1)}$	$x_{(i)}$	$\frac{f(x_i)(x_{i-1}-x_i)}{f(x_{i-1})-f(x_i)}$	$x^{(n+1)}$
$0,7$	$0,8$	$\frac{f(0,8)-f(0,7)}{0,8-0,7} = -1,6813548$	$0,8 - \frac{f(0,8)}{-1,6813548} = 0,7385654$
$0,8$	$0,7385654$	$-1,6955107$	$0,7390784$
$0,7385654$	$0,7390784$	$-1,6734174$	$0,7390851$
$0,7390784$	$0,7390851$	$-1,6736095$	$0,7390851$

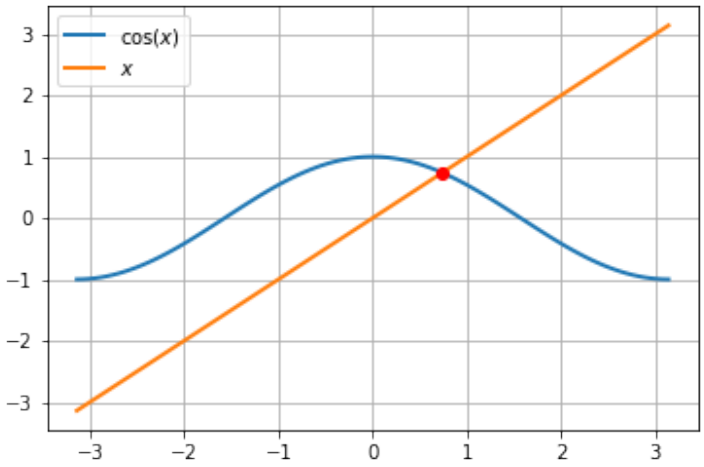
Figura 15

O gráfico acima foi gerado pelo código a seguir.

```
def f1(x): return np.cos(x)
def f2(x): return x
x = np.linspace(−3.14,3.14,100)
plt.plot(x,f1(x), linewidth=2, label='$\cos(x)$')
plt.plot(x,f2(x), linewidth=2, label='$x$')
plt.plot(0.7389, f1(0.7389), color='r',marker='o')
plt.grid()
plt.legend()
```

Saída:

```
<matplotlib.legend.Legend at 0x7f367aa2da20>
```



Método da Secante usando Python

A função `root_scalar` também implementa o método da secante, bastando para isso usar, além do parâmetro `f`, referente à função analisada, os parâmetros `x0`, referente ao primeiro valor inicial x_{-1}, `x1`, referente ao primeiro valor inicial x_0, e `method`, cujo valor atribuído é `'secant'`..

Vejamos um exemplo.

Exemplo 2.

Calcular a raiz do exemplo anterior usando a função *root_ scalar* com o método da secante.

Antes de tudo, precisamos definir a função $f(x)$

```
def f(x): return np.cos(x) − x
```

Em seguida, ajustar os parâmetros citados para os mesmos valores do exemplo e calcular a raiz.

```
opt.root_scalar(f, x0=0.7, x1 = 0.8, method='secant')
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 5
    iterations: 4
          root: 0.7390851332151595
```

Exemplo 3.

Determine a raiz de $f(x) = x^{10} - 1$, com valor inicial de 0.5, usando os três métodos abertos estudados aqui e compare a velocidade de convergência deles.

Vamos primeiro definir a função

```
def f(x): return x**(10) - 1
```

Para o método da Iteração em Ponto Fixo, precisamos encontrar a função $g(x) = x$. Existem algumas formas de fazer isso, como, gerando, por exemplo, $x^{-9} = x$. Porém, essa escolha pode gerar problemas por ter a limitação de não ser definida para valores de $x = 0$. Podemos optar, então, por fazer $x^{10} + x - 1 = x$, ou seja, fazer $g(x) = x^{10} + x - 1$.

Dessa forma, a função será definida, para o método da Iteração em Ponto Fixo, como

```
def g(x): return x**(10) + x - 1
```

Já para o método de Newton-Raphson, precisamos definir a primeira derivada da função $f(x)$, $f'(x) = 10x^9$. Assim, temos

```
def f1(x): return 10*x**9
```

O resultado para o método do ponto fixo é:

```
opt.fixed_point(g, 0.5)
```

Saída:

```
/opt/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: RuntimeWarning:
overflow encountered in double_scalars
"""Entry point for launching an IPython kernel.

array(52788.4988177)
```

Para o método de Newton-Raphson, obtemos

```
opt.root_scalar(f, fprime=f1, x0=0.5, method='newton')
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 86
```

```
    iterations: 43
         root: 1.0
```

E para a secante, escolheremos o valor 0.5 para `x0` e 0.8 para `x1`

```
opt.root_scalar(f, x0=0.5, x1 = 0.8, method='secant')
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 7
    iterations: 6
          root: 0.5000349220665965
```

Como é possível ver, somente o método de Newton-Raphson convergiu corretamente. Só por curiosidade, se usássemos o método da bisseção para calcular essa raiz, obteríamos, considerando um intervalo entre 0.5 e 1.5,

```
opt.root_scalar(f,method='bisect',bracket=[0.5,1.5])
```

Saída:

```
     converged: True
          flag: 'converged'
 function_calls: 3
    iterations: 1
          root: 1.0
```

obtemos a convergência com uma única iteração!
