Universidade Federal do Ceará
Centro de tecnologia
Departamento de Engenharia de Teleinformática
Engenharia de computação

Disciplina: Métodos Númericos
Relatório de raízes de equações algébricas não-Lineares
Professor: Cesar Lincoln
Equipe:
Luiky Magno - 397877
Luan Brasil - 397743

Turma: T01
Fortaleza-Ceará
26/09/2018

Introdução

Muitas vezes, é útil encontrar valores para variáveis independentes tais que uma função assuma certo valor. Mais comum ainda é achar tal valor de variável que seja raiz da função. Para tal fim, neste relatório, analisaremos e faremos uso dos métodos númericos Bisseção, Posição Falsa, Ponto Fixo (MPF) e suas variações (Newton-Raphson e Secante).

Todos os métodos acima citados buscam encontrar raízes de equações através de sequências de passos bem definidas e necessitam de certas condições para convergir. Analisaremos cada um deles nas seguintes páginas e verificaremos suas eficiências, assim como os compararemos entre si nos baseando em critérios como tempo de execução e número de iterações.

1 Métodos

Bisseção

Esse método consiste em dividir pela metade um intervalo dado até um certa precisão. Para que convirja, é preciso que a função seja contínua no intervalo dado e que haja somente uma raíz nesse mesmo intervalo. Para garantir essa segunda condição, devemos ter $f(a) * f(b) < 0$, pelo teorema dos sinais, e $f'(x) > 0$ ou $f'(x) < 0$ $\forall x \in [a, b]$.

A estratégia para esse método é calcular o valor de $x_k = \frac{b_k - a_k}{2}$ e fazer a verificação: Se $f(x_k) * f(b_k) > 0$, $b_{k+1} = x_k$ e $a_{k+1} = a_k$. Se não, $a_{k+1} = x_k$ e $b_{k+1} = b_k$. Isso é feito enquanto o tamanho do intervalo for maior que ϵ. Além disso, uma característica desse método é a possibilidade de calcular facilmente o número de iterações (k) que levará para convegir

$$k > \frac{\log_c(b-a) - \log_c(\epsilon)}{\log_c(2)}$$

onde ϵ é a precisão pedida, c é a base do logarítimo e k é o menor inteiro maior que o resultado do lado direito da desigualdade

Implementação

```python
from timeit import default_timer as timer
def bissecao(f, a, b, e, maxIter):
        Fa = f(a)
        if Fa == 0.0:
                return a
        Fb = f(b)
        if Fb == 0.0:
                return b
        if Fa*Fb > 0.0:
                return "Erro: no signal changing"
        print("counter\t\t a\t\t Fa\t\t b\t\t Fb\t\t X\t\t F\t\t rangeSize\t\t")
        counter = 0
        start = timer()
        while counter < maxIter:
                rangeSize = float(b - a)
                averageX = float((b+a)/2)
                F = f(averageX)
                Fa = f(a)
                Fb = f(b)
                print("\t%d\t%e\t%e\t%e\t%e\t%e\t%e\t%e" %
                (counter, a, Fa, b, Fb, averageX, F, rangeSize))
                if  F == 0 or rangeSize < e:
                        end = timer()
                        if counter != 0:
                                return [averageX, (end-start)/counter]
                        return [averageX, 0]
                if F*Fa > 0.0:
                        a = averageX
                        Fa = f(a)
                else:
                        b = averageX
                        Fb = f(b)
                counter += 1
        end = timer()
        if counter != 0:
                return [averageX, (end-start)/counter]
        return [averageX, 0]
```

Posição falsa

A ideia desse método é aproximar a função por uma reta secante que passa pelos pontos (a,f(a)) e (b,f(b)). Assim como o anterior, suas restrições para convergir são que a função deve ser contínua no intervalo e deve haver somente uma raíz no mesmo e, para isso, deve-se também respeitar o teorema dos sinais e a derivada deve ser monótona.

O melhor critério de parada para a posição falsa é que o módulo da função seja menor que uma certa precisão ϵ. Isso porque o intervalo não diminui deterministicamente como o da bisseção. Além disso, a aproximação da raiz é dada pela equação desenvolvida por semelhança de triângulos:

$$x_k = \frac{f(b_k) * a_k - f(a_k) * b_k}{f(b_k) - f(a_k)}$$

Analogamente ao método anterior, o algoritmo tem como base a seguinte estrutura: Se $f(x_k)*f(b_k) > 0$, $b_{k+1} = x_k$ e $a_{k+1} = a_k$. Se não, $a_{k+1} = x_k$ e $b_{k+1} = b_k$.

Implementação

```python
from timeit import default_timer as timer

def falsePosition(f, a, b, e, maxIter):
        Fa = f(a)
        if Fa == 0.0:
                return a
        Fb = f(b)
        if Fb == 0.0:
                return b
        if Fa*Fb > 0.0:
                return "Erro: no signal changing"
        print("counter\t\t a\t\t Fa\t\t b\t\t Fb\t\t X\t\t F\t\t rangeSize\t\t")
        counter = 0
        start = timer()
        while counter < maxIter:
                rangeSize = b - a
                x = (a*Fb - b*Fa)/(Fb-Fa)
                F = f(x)
                print("\t%d\t%e\t%e\t%e\t%e\t%e\t%e\t%e" %
                (counter, a, Fa, b, Fb, x, F, rangeSize))
                if rangeSize <= e or abs(F) <= e:
                        end = timer()
                        if counter != 0:
                                return [x, (end-start)/counter]
                        return [x, 0]
                if Fa*F > 0.0:
                        a = x
                else:
                        b = x
                Fa = f(a)
                Fb = f(b)
                counter += 1
        end = timer()
        if counter != 0:
                return [x, (end-start)/counter]
        return [x, 0]
```

Posição fixa

Esse método consiste em encontrar uma função de iteração para a utilizarmos na aproximação da raiz . Quando a função da qual queremos extrair a raiz é igualada a 0, podemos isolar a variável independente para assim definirmos tal função de iteração.

$$x = \varphi(x)$$

Onde $\varphi(x) : f(\xi) = 0 \Leftrightarrow \varphi(\xi) = \xi$

Podemos notar que a raiz será o ponto que $\phi(x)$ cruzar a função identidade e, desse modo, teremos $\varphi(\xi) = \xi$, onde ξ é raiz exata. Portanto, definiremos as iterações de modo que $x_{k+1} = \varphi(x_k)$. Para que esse método convirja, entretanto, são necessárias as seguintes condições, onde I é um intervalo que contenha ξ.

- $\varphi(x)$ e $\varphi'(x)$ devem ser contínuas em I
- $|\varphi'(x)| < M \leq 1$
- Deve-se ter $x_0 \in I$

Implementação

```
from timeit import default_timer as timer

def mpf(f, phi, x, e, iterMax):
        if f(x) == 0.0:
                return x
        print("counter\t\t f(x)\t\t x\t\t")
        counter = 0
        start = timer()
        while counter < iterMax:
                F = f(x)
                print("\t%d\t%e\t%e" % (counter, F, x))
                if abs(F) < e:
                        end = timer()
                        if counter != 0:
                                return [averageX, (end-start)/counter]
                        return [averageX, 0]
                x = phi(x)
                counter += 1
        end = timer()
        if counter != 0:
                return [averageX, (end-start)/counter]
        return [averageX, 0]
```

Newton-Raphson

Partindo do método anterior, temos que para valores muito grandes de k, o erro da iteração seguinte é dado por

$$e_{k+1} = \varphi'(\xi) * e_k$$

O método de Newton-Rapson nada mais é que uma otimização do método da posição fixa. Essa otimização ocorre porque usamos $\varphi(x)$ tal que $\varphi'(\xi) = 0$ para ser a função iteradora.

A partir desse pressuposto e de simples manipulações algébricas, concluímos que o $\varphi(x)$ ótimo é dado por

$$\varphi(\xi) = 1 - \frac{f(x)}{f'(x)}$$

Para que esse método convirja é necessário que, além das condições do método do ponto fixo, $f'(x) \neq 0$. Além disso, notamos uma desvantagem considerável nesse método que é a necessidade do calculo da derivado da função que queremos a raiz. Existem, contudo, meios de driblar esse empecilho. No caso de funções polinomiais a derivada é facilmente calculada usando método de Horner. Já para as demais funções, podemos realizar aproximações do valor da derivada no ponto.

Implementação

```
from timeit import default_timer as timer

def newton(f, f_linha, x, e, iterMax):
        if f(x) == 0.0:
                return x

        print("counter\t\t f(x)\t\t x\t\t")
        counter = 0
        start = timer()
        while counter < iterMax:
                F = f(x)
                print("\t%d\t%e\t%e" % (counter, F, x))
                if abs(F) < e:
                        end = timer()
                        if counter != 0:
                                return [x, (end-start)/counter]
                        return [x, 0]
                x = x - (f(x)/f_linha(x))
                counter += 1
        end = timer()
        if counter != 0:
                return [x, (end-start)/counter]
        return [x, 0]
```

Secante

Como mencionado anteriormente, um meio de evitar o cálculo da derivada da função é calcular sua aproximação. Assim, como temos que a derivada é o valor da inclinação da reta tangente à função no ponto, podemos aproximá-la pela inclinação da reta secante.

A partir desse conceito e de manipulação algébrica, chegamos à seguinte expressão para a nova função de iteração:

$$\varphi(x) = \frac{x_{k-1}f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})}$$

Para que $\varphi(x)$ convirja, basta que as mesmas condições do método de Newton-Rapson sejam satisfeitas.

Implementação

```
from timeit import default_timer as timer

def secante(f, x0, x1, e, iterMax):
        if f(x0) == 0:
                return x0
        if f(x1) == 0:
                return x1
        print("counter\t\t f(x)\t\t x\t\t")
        counter = 0
        start = timer()
        while counter < iterMax:
                F = f(x1)
                print("\t%d\t%e\t%e" % (counter, f(x1), x1))
                if abs(F) < e:
                end = timer()
                if counter != 0:
                return [x1, (end-start)/counter]
                return [x1, 0]
                x2 = (x0*f(x1) - x1*f(x0))/(f(x1) - f(x0))
                x0 = x1
                x1 = x2
                counter += 1
        end = timer()
        if counter != 0:
                return [x1, (end-start)/counter]
        return [x1, 0]
```

2 Testes

Para todos os métodos, testaremos as duas funções abaixo em cada um, para que possamos comparar os resultados. Além disso, a iteração 0 será desconsiderada, uma vez que esta resulta dos cálculos iniciais. Ademais, utilizaremos uma precisão $\epsilon = 10^{-4}$. Ademais, o retorno de cada função é uma lista onde o elemento 0 é a aproximação da raíz e o elemento 1 o tempo médio de iteração.

As funções que testaremos são as seguintes:

$$f_1(x) = x - 10x + 5 \tag{1}$$

$$f_2(x) = 4\sin(x) - e^x \tag{2}$$

Bisseção

Utilizaremos o intervalo de $[0, 1]$

$f_1(x)$

k	a	f(a)	b	f(b)	x	f(x)	rangeSize
0	$0.00000e+0$	$5.00000e+0$	$1.00000e+0$	$-4.00000e+0$	$5.00000e-1$	$2.62500e+0$	$1.00000e+0$
1	5.00000e-1	2.62500e+0	1.00000e+0	-4.00000e+0	7.50000e-1	-2.03125e-1	5.00000e-1
2	5.000000e-1	2.625000e+0	7.500000e-1	-2.031250e-1	6.250000e-1	1.337891e+0	2.500000e-1
3	6.250000e-1	1.337891e+0	7.500000e-1	-2.031250e-1	6.875000e-1	5.983887e-1	1.250000e-1
4	6.875000e-1	5.983887e-1	7.500000e-1	-2.031250e-1	7.187500e-1	2.052917e-1	6.250000e-2
5	7.187500e-1	2.052917e-1	7.500000e-1	-2.031250e-1	7.343750e-1	2.986908e-3	3.125000e-2
6	7.343750e-1	2.986908e-3	7.500000e-1	-2.031250e-1	7.421875e-1	-9.959459e-2	1.562500e-2
7	7.343750e-1	2.986908e-3	7.421875e-1	-9.959459e-2	7.382812e-1	-4.818505e-2	7.812500e-3
8	7.343750e-1	2.986908e-3	7.382812e-1	-4.818505e-2	7.363281e-1	-2.256935e-2	3.906250e-3
9	7.343750e-1	2.986908e-3	7.363281e-1	-2.256935e-2	7.353516e-1	-9.783789e-3	1.953125e-3
10	7.343750e-1	2.986908e-3	7.353516e-1	-9.783789e-3	7.348633e-1	-3.396582e-3	9.765625e-4
11	7.343750e-1	2.986908e-3	7.348633e-1	-3.396582e-3	7.346191e-1	-2.043722e-4	4.882812e-4
12	7.343750e-1	2.986908e-3	7.346191e-1	-2.043722e-4	7.344971e-1	1.391384e-3	2.441406e-4
13	7.344971e-1	1.391384e-3	7.346191e-1	-2.043722e-4	7.345581e-1	5.93535e-4	1.220703e-4
14	7.345581e-1	5.935350e-4	7.346191e-1	-2.043722e-4	7.345886e-1	1.945886e-4	6.103516e-5

Resultados: $\xi \approx 0.734588623046875$ e Tempo médio de iteração$\approx 9.972128568084113e-05s$

$f_2(x)$

k	a	f(a)	b	f(b)	x	f(x)	rangeSize
0	-1.000000e+0	-3.733763e+0	1.000000e+0	6.476021e-1	0.000000e+0	-1.000000e+0	2.000000e+0
1	0.000000e+0	-1.000000e+0	1.000000e+0	6.476021e-1	5.000000e-1	2.689809e-1	1.000000e+0
2	0.000000e+0	-1.000000e+0	5.000000e-1	2.689809e-1	2.500000e-1	-2.944096e-1	5.000000e-1
3	2.500000e-1	-2.944096e-1	5.000000e-1	2.689809e-1	3.750000e-1	1.009870e-2	2.500000e-1
4	2.500000e-1	-2.944096e-1	3.750000e-1	1.009870e-2	3.125000e-1	-1.370839e-1	1.250000e-1
5	3.125000e-1	-1.370839e-1	3.750000e-1	1.009870e-2	3.437500e-1	-6.214576e-2	6.250000e-2
6	3.437500e-1	-6.214576e-2	3.750000e-1	1.009870e-2	3.593750e-1	-2.567695e-2	3.125000e-2
7	3.593750e-1	-2.567695e-2	3.750000e-1	1.009870e-2	3.671875e-1	-7.701243e-3	1.562500e-2
8	3.671875e-1	-7.701243e-3	3.750000e-1	1.009870e-2	3.710938e-1	1.220853e-3	7.812500e-3
9	3.671875e-1	-7.701243e-3	3.710938e-1	1.220853e-3	3.691406e-1	-3.234683e-3	3.906250e-3
10	3.691406e-1	-3.234683e-3	3.710938e-1	1.220853e-3	3.701172e-1	-1.005535e-3	1.953125e-3
11	3.701172e-1	-1.005535e-3	3.710938e-1	1.220853e-3	3.706055e-1	1.080047e-4	9.765625e-4
12	3.701172e-1	-1.005535e-3	3.706055e-1	1.080047e-4	3.703613e-1	-4.486786e-4	4.882812e-4
13	3.703613e-1	-4.486786e-4	3.706055e-1	1.080047e-4	3.704834e-1	-1.703154e-4	2.441406e-4
14	3.704834e-1	-1.703154e-4	3.706055e-1	1.080047e-4	3.705444e-1	-3.114992e-5	1.220703e-4
15	3.705444e-1	-3.114992e-5	3.706055e-1	1.080047e-4	3.705750e-1	3.842875e-5	6.103516e-5

Resultados: $\xi \approx 0.370574951171875$ e Tempo médio de iteração$\approx 0.00014379160008199204s$

Posição falsa

Utilizaremos o intervalo de $[0, 1]$

$f_1(x)$

k	a	f(a)	b	f(b)	x	f(x)	rangeSize
0	0.000000e+0	5.000000e+0	1.000000e+0	-4.000000e+0	5.555556e-1	2.085048e+0	1.000000e+0
1	5.555556e-1	2.085048e+0	1.000000e+0	-4.000000e+0	7.078449e-1	3.442176e-1	4.444444e-1
2	7.078449e-1	3.442176e-1	1.000000e+0	-4.000000e+0	7.309941e-1	4.708530e-2	2.921551e-1
3	7.309941e-1	4.708530e-2	1.000000e+0	-4.000000e+0	7.341238e-1	6.269904e-3	2.690059e-1
4	7.341238e-1	6.269904e-3	1.000000e+0	-4.000000e+0	7.345399e-1	8.319008e-4	2.658762e-1
5	7.345399e-1	8.319008e-4	1.000000e+0	-4.000000e+0	7.345951e-1	1.103251e-4	2.654601e-1
6	7.345951e-1	1.103251e-4	1.000000e+0	-4.000000e+0	7.346024e-1	1.463018e-5	2.654049e-1

Resultados: $\xi \approx 0.7346023886866092$ e Tempo médio de iteração$\approx 9.860633326752577e-05s$

$f_2(x)$

k	a	f(a)	b	f(b)	x	f(x)	rangeSize
0	0.000000e+0	-1.000000e+0	1.000000e+0	6.476021e-1	6.069427e-1	4.466222e-1	1.000000e+0
1	0.000000e+0	-1.000000e+0	6.069427e-1	4.466222e-1	4.195585e-1	1.081394e-1	6.069427e-1
2	0.000000e+0	-1.000000e+0	4.195585e-1	1.081394e-1	3.786153e-1	1.827556e-2	4.195585e-1
3	0.000000e+0	-1.000000e+0	3.786153e-1	1.827556e-2	3.718201e-1	2.874919e-3	3.786153e-1
4	0.000000e+0	-1.000000e+0	3.718201e-1	2.874919e-3	3.707542e-1	4.469907e-4	3.718201e-1
5	0.000000e+0	-1.000000e+0	3.707542e-1	4.469907e-4	3.705885e-1	6.937076e-5	3.707542e-1

Resultados: $\xi \approx 0.37058852288011002$ e Tempo médio de iteração$\approx 0.00013610980022349395s$

Ponto fixo

Para cada $f_i(x)$ escolhemos $\varphi_i(x)$ para que o resultado convergisse

- $\varphi_1(x) = \sqrt{\frac{x+5}{10}}$ com $x_0 = 0$
- $\varphi_1(x) = x - s\sin(x) + 0.5e^x$ com $x_0 = 1$

$f_1(x)$

k	f(x)	x
0	5.000000e+00	0.000000e+00
1	3.535534e-01	7.071068e-01
2	3.815530e-02	7.316798e-01
3	4.195072e-03	7.342826e-01
4	4.621451e-04	7.345682e-01
5	5.092265e-05	7.345996e-01

Resultados: $\xi \approx 0.734609752736254$ e Tempo médio de iteração$\approx 8.747100000618956e-05s$

$f_2(x)$

k	f(x)	x
0	6.476021e-01	1.000000e+00
1	5.369424e-01	6.761989e-01
2	8.269966e-02	4.077277e-01
3	-9.555905e-03	3.663779e-01
4	1.362361e-03	3.711559e-01
5	-1.901947e-04	3.704747e-01
6	2.663321e-05	3.705698e-01

Resultados: $\xi \approx 0.37056977749916131$ e Tempo médio de iteração$\approx 9.527166669916672e-05s$

Newton-Raphson

Usaremos $x_0 = 1$ para $f_1(x)$, $x_0 = 0$ para $f_2(x)$ neste método para cumprir as condições de convergência.

$f_1(x)$

k	f(x)	x
0	-4.000000e+00	1.000000e+00
1	-4.005699e-01	7.647059e-01
2	-6.770476e-03	7.351212e-01
3	-2.088175e-06	7.346037e-01

Resultados: $\xi \approx 0.7346036675194284$ e Tempo médio de iteração$\approx 0.00011001833263435401s$

$f_2(x)$

k	f(x)	x
0	-1.000000e+00	0.000000e+00
1	-8.683364e-02	3.333333e-01
2	-1.835271e-03	3.697535e-01
3	-9.357810e-07	3.705577e-01

Resultados: $\xi \approx 0.37055768553223589$ e Tempo médio de iteração$\approx 0.00016104466703836806s$

Secante

Para $f_1(x)$ e $f_2(x)$ utilizamos $x_0 = 0$ e $x_1 = 1$

$f_1(x)$

k	f(x)	x
0	-4.000000e+00	1.000000e+00
1	2.085048e+00	5.555556e-01
2	3.442176e-01	7.078449e-01
3	-4.393307e-02	7.379574e-01
4	7.117108e-04	7.345491e-01
5	1.420127e-06	7.346034e-01

Resultados: $\xi \approx 0.7346033991599847$ e Tempo médio de iteração$\approx 0.00014364559974637813s$

$f_2(x)$

k	f(x)	x
0	6.476021e-01	1.000000e+00
1	4.466222e-01	6.069427e-01
2	-1.819540e+00	-2.665185e-01
3	1.402583e-01	4.347982e-01
4	3.174169e-02	3.846066e-01
5	-1.443434e-03	3.699253e-01
6	1.309363e-05	3.705638e-01

Resultados: $\xi \approx 0.37056383891649708$ e Tempo médio de iteração$\approx 0.00020729566661625843s$

2.1 Conclusão

Ao realizar os testes nos cinco métodos abordados, vimos que todos conseguiram obter uma boa aproximação das raízes exatas ou esperadas. Os métodos da bisseção e posição falsa têm o semelhante princípio de reduzir o intervalo no qual a raiz está. Dentre esses, o da posição falsa se mostrou mais eficiente no número de iterações, mas possuiu um resultado muito semelhante quanto ao tempo médio de cada iteração. Quanto aos demais, que se mostraram mais eficientes em relação ao número de iterações que os dois últimos, porém não em relação ao tempo (com exceção do método do ponto fixo que se manteve aproximadamente igual), vemos que o método de Newton-Raphson converge em menos iterações, o que equilibra a necessidade do cálculo da derivada. Por fim quanto ao tempo de execução, o método que registrou a maior média por iteração foi o da secante, devido à sua relativa complexidade ao realizar suas operações