Método Quasi-Newton

Gustavo Oliveira
Paula Villela
Wagner Morais

Introdução

- Surge para suprir a deficiência do método de Newton para problemas mais complexos onde não conseguimos calcular a derivada ou quando se torna um processo muito "caro".
- O método Quasi-Newton é basicamente o método de Newton, onde o cálculo da derivada é trocada por uma regra recursiva que permite a construção gradativa de uma matriz Hk que corresponde a uma estimativa da inversa da Hessiana da função objetivo.
- Métodos DFP (Davidon-Fletcher-Powell) e o método BFGS (Broyden- Fletcher-Goldfarb- Shanno).
- Família de Broyden.

Introdução

BFGS

A correção proposta pelo método BFGS é dada por:

$$C_k^{BFGS} = \left(1 - \frac{r_k' H_k r_k}{r_k' v_k}\right) \frac{v_k v_k'}{v_k' r_k} - \frac{v_k r_k' H_k + H_k r_k v_k'}{r_k' v_k}$$

DFP

A correção proposta pelo método DFP é dada por:

$$C_k^{DFP} = \frac{v_k v_k'}{v_k' r_k} - \frac{H_k r_k r_k' H_k}{r_k' H_k r_k}$$

Introdução

FAMÍLIA BROYDEN

- A correção genérica utilizada pelos métodos conhecidos como família de Broyden é dada por:

$$C_k = (1-\alpha)C_k^{DFP} + \alpha C_k^{BFGS}$$

- Em todos os casos da família de Broyden, incluindo os casos extremos BFGS e DFP, a fórmula de atualização para a estimativa da inversa da Hessiana fica:

$$H_{k+1} = H_k + C_k(\alpha)$$

- Para $\alpha = 0$, obtém-se o método DFP, e para $\alpha = 1$ o método BFGS.

Objetivo

Dado um problema de minimização linear irrestrito, utilizando o método de Quasi-Newton (Família de Broyden) encontrar o ponto ótimo x*.

Estrutura básica do Algoritmo

Algoritmo de Quasi-Newton

$k \leftarrow 0$

$H_k \leftarrow I$

$g_k \leftarrow \text{gradiente}(f(\cdot), x_k)$

enquanto (não critério de parada)

$$
\begin{aligned}
d_k &\leftarrow -H_k g_k \\
\alpha_k &\leftarrow \arg\ \min_\alpha f(x_k + \alpha d_k) \\
x_{k+1} &\leftarrow x_k + \alpha_k d_k \\
g_{k+1} &\leftarrow \text{gradiente}(f(\cdot), x_{k+1}) \\
v_k &\leftarrow x_k - x_{k+1} \\
r_k &\leftarrow g_k - g_{k+1} \\
C_k^{DFP} &\leftarrow \frac{v_k v_k'}{v_k' r_k} - \frac{H_k r_k r_k' H_k}{r_k' H_k r_k} \\
C_k^{BFGS} &\leftarrow \left(1 + \frac{r_k' H_k r_k}{r_k' v_k}\right) \frac{v_k v_k'}{v_k' r_k} - \frac{v_k r_k' H_k + H_k r_k v_k'}{r_k' v_k} \\
C_k &\leftarrow (1-\alpha) C_k^{DFP} + \alpha C_k^{BFGS} \\
H_{k+1} &\leftarrow H_k + C_k(\alpha) \\
k &\leftarrow k + 1
\end{aligned}
$$

fim-enquanto

Materiais e Metodologia

- O algoritmo foi desenvolvido em Python, utilizando o Jupyter Notebook e o Google Colab;
- Para a construção do algoritmo foi utilizado as seguintes bibliotecas:
  - Sympy (utiliza variáveis simbólicas e funções para manipular elas);
  - Numpy (calcular multiplicação matricial) ;
  - Scipy (encontrar o arg min da função);
  - Matplotlib (plot dos gráficos);

Materiais e Metodologia

FUNÇÃO gradiente:

Função para calcular o gradiente de uma função de variáveis simbólicas.

```
def gradiente(fx, symx):
    g = []
    for x in symx:
        g.append(diff(fx, x))
    return(g)
```

FUNÇÃO subsGrad:

Função para substituir o ponto x0 = (x1, x2, ...) nas coordenadas do gradiente

```
def subsGrad(gx, symx, x0):
    subsFx = gx[:]
    for i in range(len(subsFx)):
        for j in range(len(symx)):
            subsFx[i] = subsFx[i].subs(symx[j], x0[j])

    return subsFx
```

Materiais e Metodologia

FUNÇÃO <u>subsFx:</u>

Função para encontrar o valor da mesma dado um ponto x.

```python
def subsFx(fx, symx, x):
    subsFx = fx
    for i in range(len(symx)):
        subsFx = subsFx.subs(symx[i], x[i])

    return subsFx
```

Materiais e Metodologia

FUNÇÃO <u>calculaCDFP:</u>

Função que retorna a correção proposta pelo método CDFP.
((vk*vk')/(vk'*rk))-((hk*rk*rk'*hk)/(rk'*hk*rk))

```
def calculaCDFP(vk_temp, rk_temp, hk):

    #transformando em vetores numpy
    vk = np.array(vk_temp)
    rk = np.array(rk_temp)
    #numerador 1ª parte
    parte1 = np.outer(vk, vk.T)
    #denominador 1ª parte
    parte2 = np.dot(vk.T, rk)
    #divisão 1ª parte
    parte3 = parte1 / parte2
```

```
    #numerador 2ª parte
    parte4 = np.dot(hk, rk)
    parte4 = np.reshape(parte4, (len(rk), 1))
    parte5 = np.dot(rk.T, hk)
    parte6 = np.outer(parte4, parte5)
    #denominador 2ª parte
    parte7 = np.dot(rk.T, hk)
    parte8 = np.dot(parte7, rk)
    #divisão 2ª parte
    parte9 = parte6 / parte8
    #subtração dos dois membros
    CDFP = parte3 - parte9

    return CDFP
```

Materiais e Metodologia

FUNÇÃO calculaBFGS:

Função que retorna a correção proposta pelo método BFGS.
(1+((rk'*hk*rk)/(rk'*vk)))*((vk*vk')/(vk'*rk))-((vk*rk'*hk + hk*rk*vk')/(rk'*vk))

```
def calculaCBFGS(vk_temp, rk_temp, hk):

    #transformando em vetores numpy
    vk = np.array(vk_temp)
    rk = np.array(rk_temp)
    #numerador primeira parte
    parte1 = np.dot(rk.T, hk)
    parte2 = np.dot(parte1, rk)
    #denomiador primeira parte
    parte3 = np.dot(rk.T, vk)
    #divisao primeira parte
    parte4 = parte2 / parte3
```

```
    #soma primeira parte
    parte5 = 1 + parte4
    #numerador segunda parte
    parte6 = np.outer(vk, vk.T)
    #denominador segunda parte
    parte7 = np.dot(vk.T, rk)
    #divisao segunda parte
    parte8 = parte6 / parte7
    #primeira * segunda parte
    parte9 = parte5 * parte8
```

```
    #numerador terceira parte
    parte10 = np.outer(vk, rk.T)
    parte11 = np.dot(parte10, hk)
    parte12 = np.dot(hk, rk)
    parte12 = np.reshape(parte12, (len(rk), 1))
    parte13 = parte12 * vk.T
    parte14 = parte11 + parte13
    #denominador terceira parte
    parte15 = np.dot(rk.T, vk)
    #divisao terceira parte
    parte16 = parte14 / parte15
    BFGS = parte9 - parte16
    return BFGS
```

Materiais e Metodologia

FUNÇÃO critParada:

Função responsável para verificar se a busca pelo ponto ótimo está se estabilizando em torno de um ponto

```python
def critParada(gk, funcoes):
    if len(funcoes) < 6:
        elevQuadrado = list(map(lambda x: x ** 2, gk))
        soma = sum(elevQuadrado)
        raiz = sqrt(soma)

        return raiz > 0.001
    else:
        try:
            Deltaf = max(funcoes) - min(funcoes)
            fcincomais = max(funcoes[-6:])
            fcincomenos = min(funcoes[-6:])
            deltinhaf = fcincomais - fcincomenos
```

```python
    if deltinhaf < (0.0001 * Deltaf):
        return False
    else:
        return True

except:
    elevQuadrado = list(map(lambda x: x ** 2, gk))
    soma = sum(elevQuadrado)
    raiz = sqrt(soma)

    return raiz > 0.001
```

Materiais e Metodologia

Funções de plotagem de gráficos:

```
def graph3D(fx, x, symx):
    plt.rcParams['figure.figsize'] = 8, 6
    plotting.plot3d(fx, title = f"Gráfico da Função: {fx}")

def graphConvertion(fx, x, symx):
    iteracoes = []
    valores = []

    for i in range(0, len(x)):
        iteracoes.append(i)
        valores.append(fx.subs([(symx[0], x[i][0]), (symx[1], x[i][1])]))

    plt.figure(figsize=(10, 8))
    plt.xlabel("Iteracoes", fontsize = 16)
    plt.ylabel("F(x)", fontsize = 16)
    plt.title("Gráfico de convergência", fontsize = 16)
    plt.plot(iteracoes, valores)
    plt.yscale("log")
    plt.xticks(iteracoes)
    plt.show()
```

Materiais e Metodologia

Funções de plotagem de gráficos:

```python
def graphNivel(fx, x, symx):
    fig, ax = plt.subplots(figsize=(10, 10))
    #curvas de nível
    xvec = np.linspace(-5, int(max(x[:][0])) + 5, 500)
    yvec = np.linspace(-5, int(max(x[:][1])) + 5, 500)
    xgraf, ygraf = np.meshgrid(xvec, yvec)

    funclam = lambdify(symx, fx)
    funcao = funclam(xgraf, ygraf)

    contornp = ax.contour(xgraf, ygraf, funcao, 100)
    plt.grid()


    plt.scatter(x[-1][0], x[-1][1], s=100, c="red")
    plt.scatter(x[0][0], x[0][1], s=100, c="green")
```

```python
for i in range(len(x)-1):
    plt.plot([x[i][0], x[i+1][0]], [x[i][1],
    x[i+1][1]], linewidth = 3, color = "green")
    plt.scatter(x[i][0], x[i][1], s=50, c="green")


plt.annotate(
        f"x0: {tuple(x[0])}",
        xy=(x[0][0], x[0][1]), xytext=(-20, 20),
        textcoords='offset points', ha='right', va='bottom',
        bbox=dict(boxstyle='round,pad=0.5', fc='yellow'),
        arrowprops=dict(arrowstyle = '->',
        connectionstyle='arc3,rad=0'), fontsize=13)

plt.annotate(
        f"x*: ({x[-1][0]:.2f}, {x[-1][1]:.2f})",
        xy=(x[-1][0], x[-1][1]), xytext=(-20, 20),
        textcoords='offset points', ha='right', va='bottom',
        bbox=dict(boxstyle='round,pad=0.5', fc='yellow'),
        arrowprops=dict(arrowstyle = '->',
         connectionstyle='arc3,rad=0'), fontsize=13)

plt.title("Gráfico de busca - Quasi Newton")
plt.show()
```

```python
if __name__ == '__main__':
    n = int(input("Digite quantas variáveis sua função possui: "))
    symx = symbols(f"x(1:{n+1})")

    f = str(input("Digite a sua função: "))
    fx = sympify(f, convert_xor=True)

    x0 = list(map(float,input("Digite o seu ponto x0 (ex: 1 2): ")
    .strip().split()))[:n]

    x = [x0, ]

    QuasiNewton(fx, x, symx, n)


    resp = str(input("\n\nGostaria de visualizar os gráficos?(sim/nao) "))


    if resp in ["sim", "Sim", "SIM", "s", "S"]:
        graph3D(fx, x, symx)

        graphNivel(fx, x, symx)

        graphConvertion(fx, x, symx)
```

Função Principal

Algoritmo do método de Quasi-Newton

```
def QuasiNewton(fx, x, symx, n):

    fg = gradiente(fx, symx)
    hk = np.eye(n)
    gk = subsGrad(fg,symx, x[0])
    k = 0
    funcoes = []
    a = symbols("a")
    while critParada(gk, funcoes):
        X = x[k] - a * np.dot(hk, gk)
        Fdealfa = fx.subs([(symx[0], X[0]), (symx[1], X[1])])
        DifFdealfa = diff(Fdealfa, a)
        try:
            solveAlfa = nroots(DifFdealfa, maxsteps = 10000)
        except:
            print("\n\nNão foi possível encontrar o ponto ótimo desta função")
            exit(1)
        alfa = solveAlfa[0]
        try:
            for i in range(len(solveAlfa)):
                if(Fdealfa.subs(a, solveAlfa[i]) < Fdealfa.subs(a, alfa)):
                    alfa = solveAlfa[i]
        except:
            alfa = solveAlfa[0]

        for i in range(len(X)):
            X[i] = X[i].subs(a, alfa)
        G = subsGrad(fg, symx, X)
        vk = list(map(lambda i, j: i - j, x[k], X))
        rk = list(map(lambda i, j: i - j, gk, G))
        CDFP = calculaCDFP(vk, rk, hk)
        CBFGS = calculaCBFGS(vk, rk, hk)
        beta = np.random.rand()
        ck = (1-beta) * CDFP + beta * CBFGS
        H = hk + ck
        k += 1
        x.append(X)
        gk = np.copy(G)
        funcoes.append(subsFx(fx, symx, X))
        hk = np.copy(H)
    print(f"\n\nQuantidade de execuções: {len(x)}")
    print(f"Ponto ótimo encontrado: {tuple(x[-1])}")
    print(f"A função no ponto: {fx.subs([(symx[0], x[-1][0]), (symx[1], x[-1][1])])}")
```

Análise de Resultados

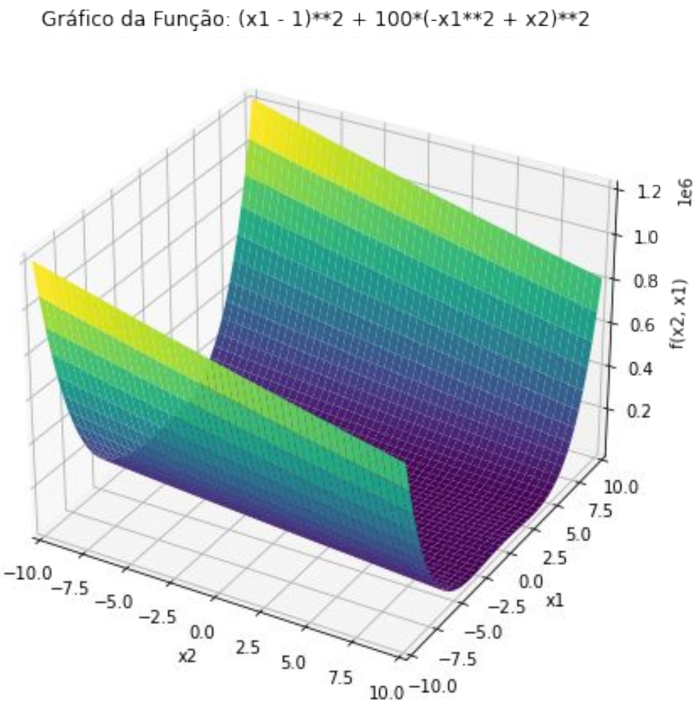
Exemplo 1:

$f(x) = 100(x_2 - x_1^2)^2 + (x_1 - 1)^2$

$X_0 = [15, 25]$

```
Digite quantas variáveis sua função possui: 2
Digite a sua função: 100*(x2-x1^2)^2+(x1-1)^2
Digite o seu ponto x0 (ex: 1 2): 15 25


Quantidade de execuções: 26
Ponto ótimo encontrado: (0.999999999999997, 0.999999999999993)
A função no ponto: 2.07569025686279E-29
```



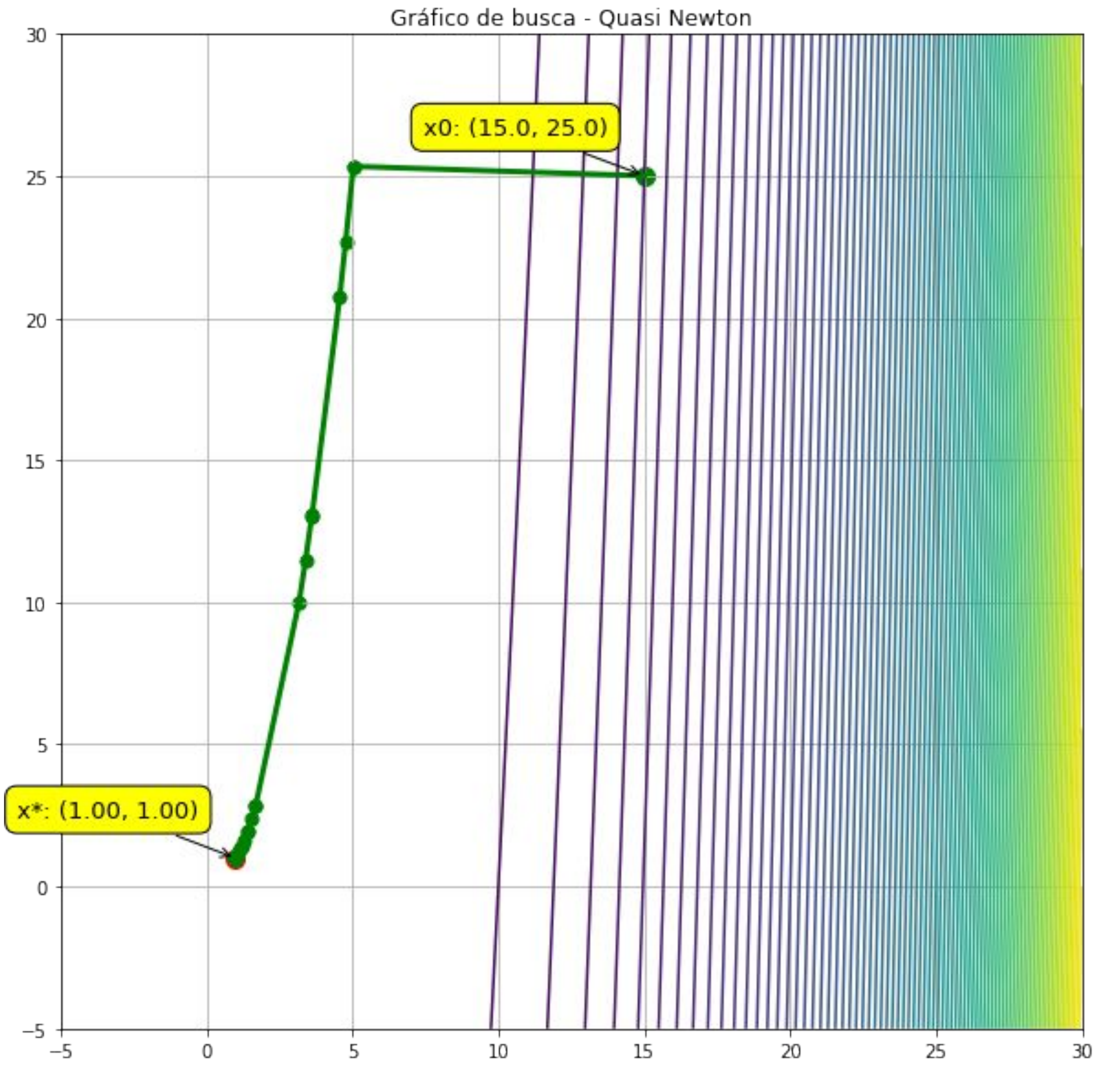


Gráfico de busca - Quasi Newton
x0: (15.0, 25.0)
x*: (1.00, 1.00)
30
25
20
15
10
5
0
−5
−5
0
5
10
15
20
25
30

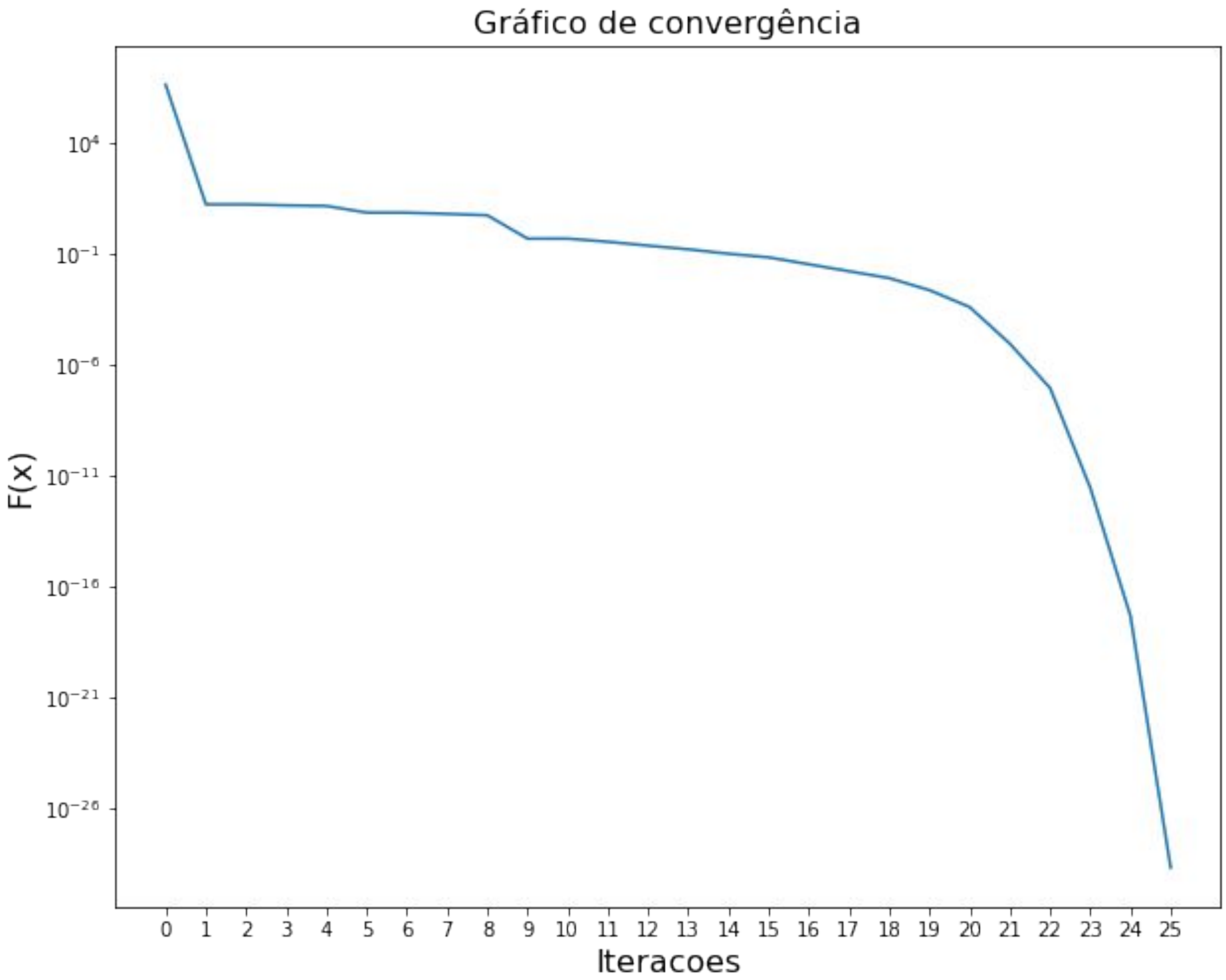


Gráfico de convergência
F(x)
10^{4}
10^{-1}
10^{-6}
10^{-11}
10^{-16}
10^{-21}
10^{-26}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Iteracoes

Análise de Resultados

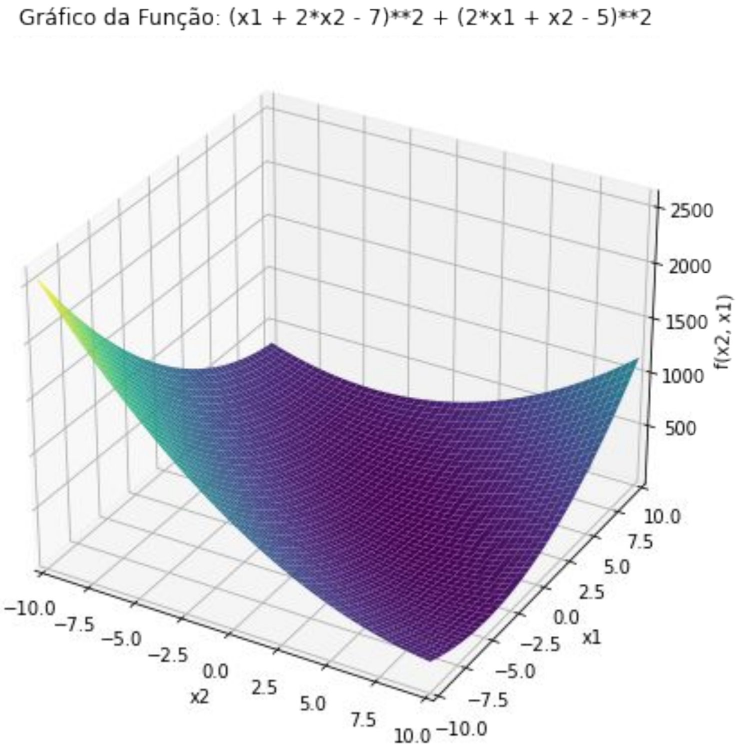
Exemplo 2:

$f(x) = (x_1+2x_2-7)^2 + (2x_1+x_2-5)^2$

$X_0 = [-10, 10]$

```
Digite quantas variáveis sua função possui: 2
Digite a sua função: (x1+2*x2-7)^2 + (2*x1+x2 -5)^2
Digite o seu ponto x0 (ex: 1 2): -10 10
```

```
Quantidade de execuções: 3
Ponto ótimo encontrado: (1.0000000000000, 3.0000000000000)
A função no ponto: 5.67979851759128E-29
```



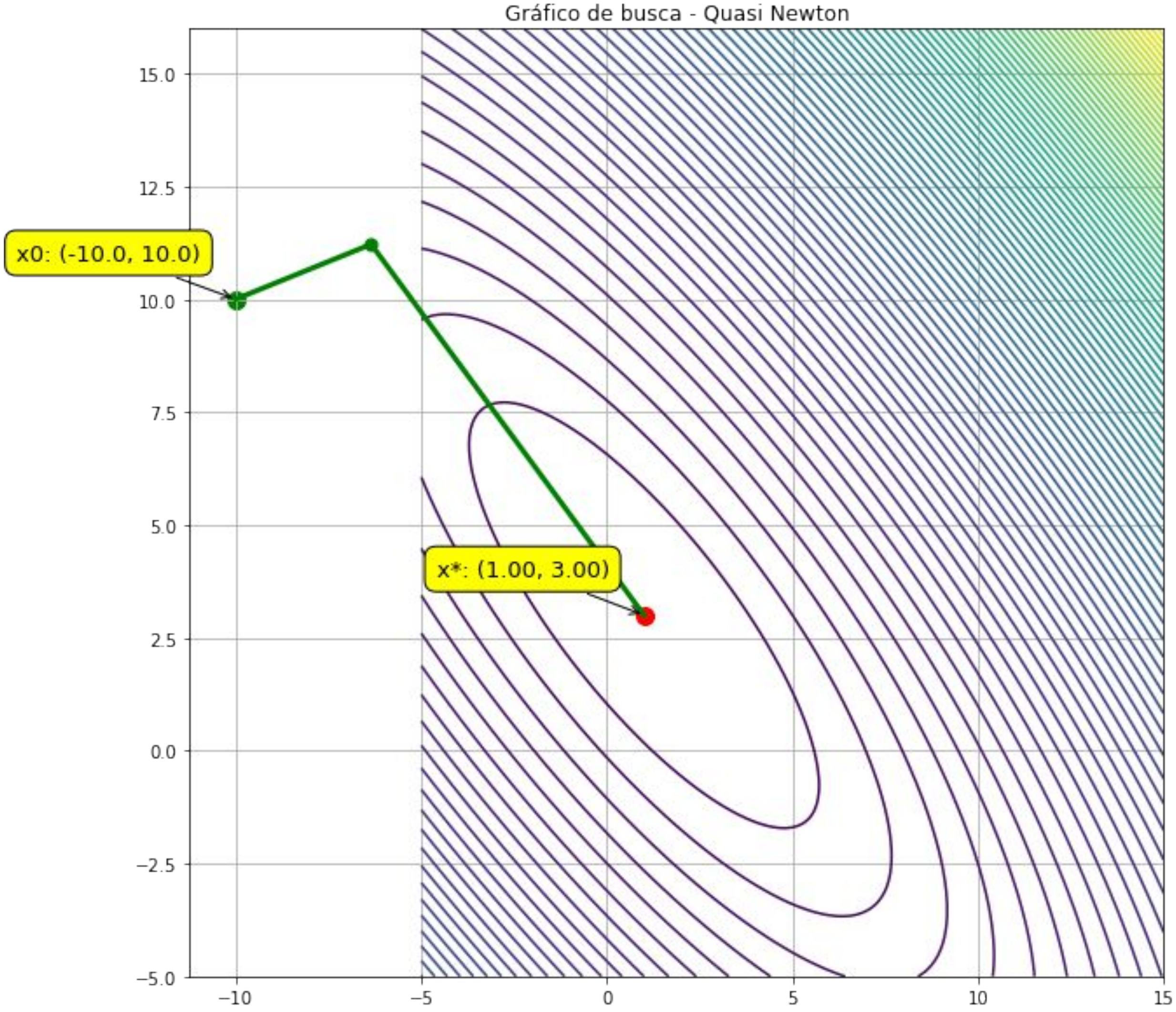


Gráfico de busca - Quasi Newton
x0: (-10.0, 10.0)
x*: (1.00, 3.00)
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−10
−5
0
5
10
15

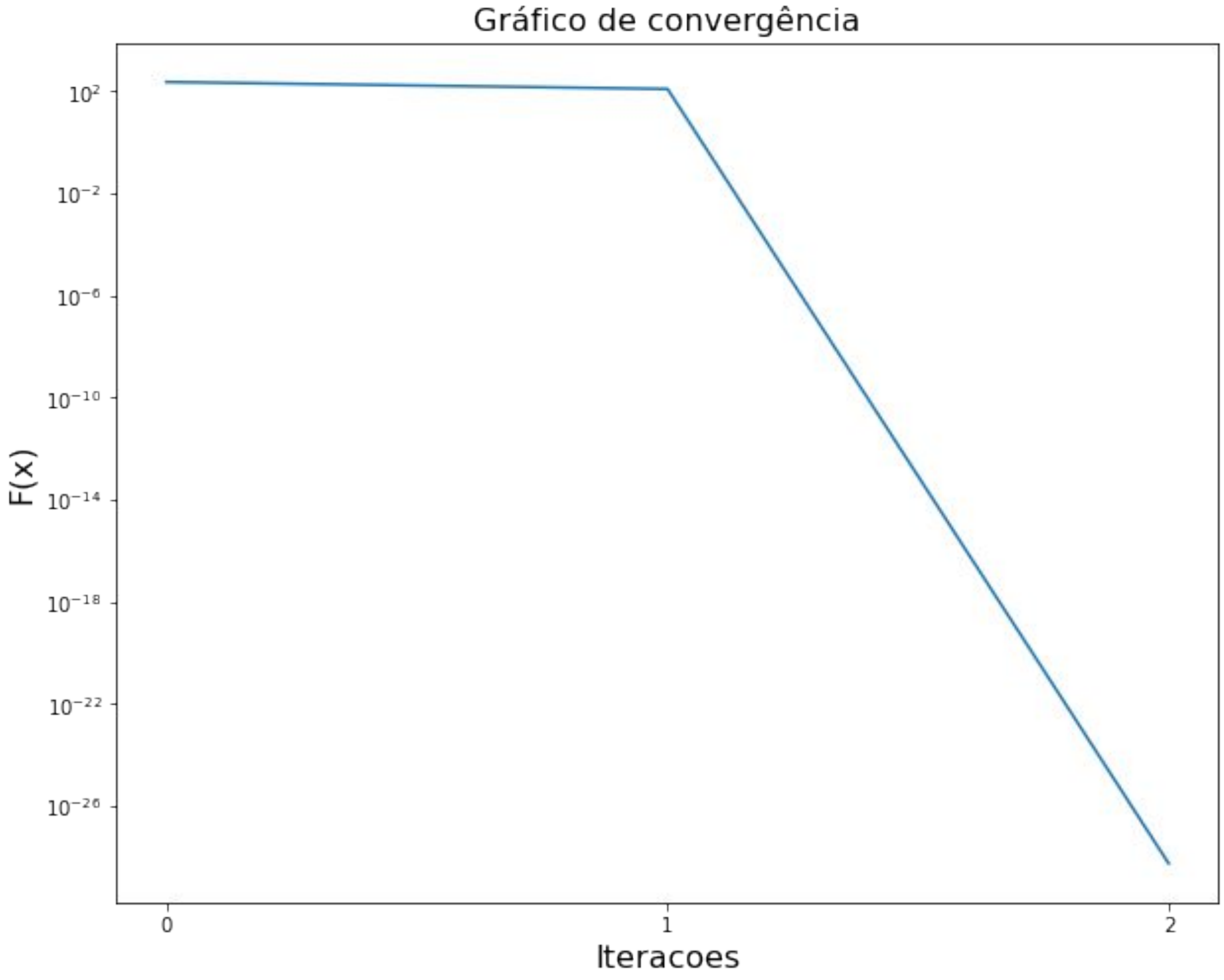
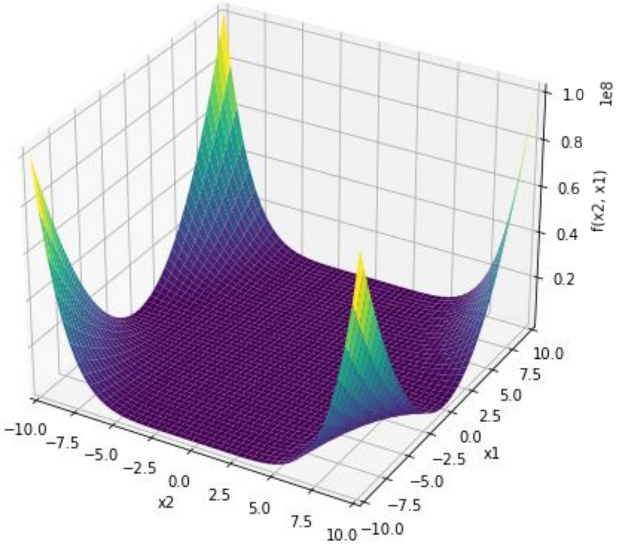


Gráfico de convergência
F(x)
10^{2}
10^{-2}
10^{-6}
10^{-10}
10^{-14}
10^{-18}
10^{-22}
10^{-26}
0
1
2
Iteracoes

Análise de Resultados



Exemplo 3:

$f(x) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2$

$X_0 = [-4.5, 4.5]$

```
Digite quantas variáveis sua função possui: 2
Digite a sua função: (1.5-x1+x1*x2)^2+(2.25-x1+x1*x2^2)^2+(2.625-x1+x1*x2^3)^2
Digite o seu ponto x0 (ex: 1 2): -4.5 4.5


Quantidade de execuções: 146
Ponto ótimo encontrado: (3.0000000000000, 0.50000000000000)
A função no ponto: 1.38666955995881E-31
```

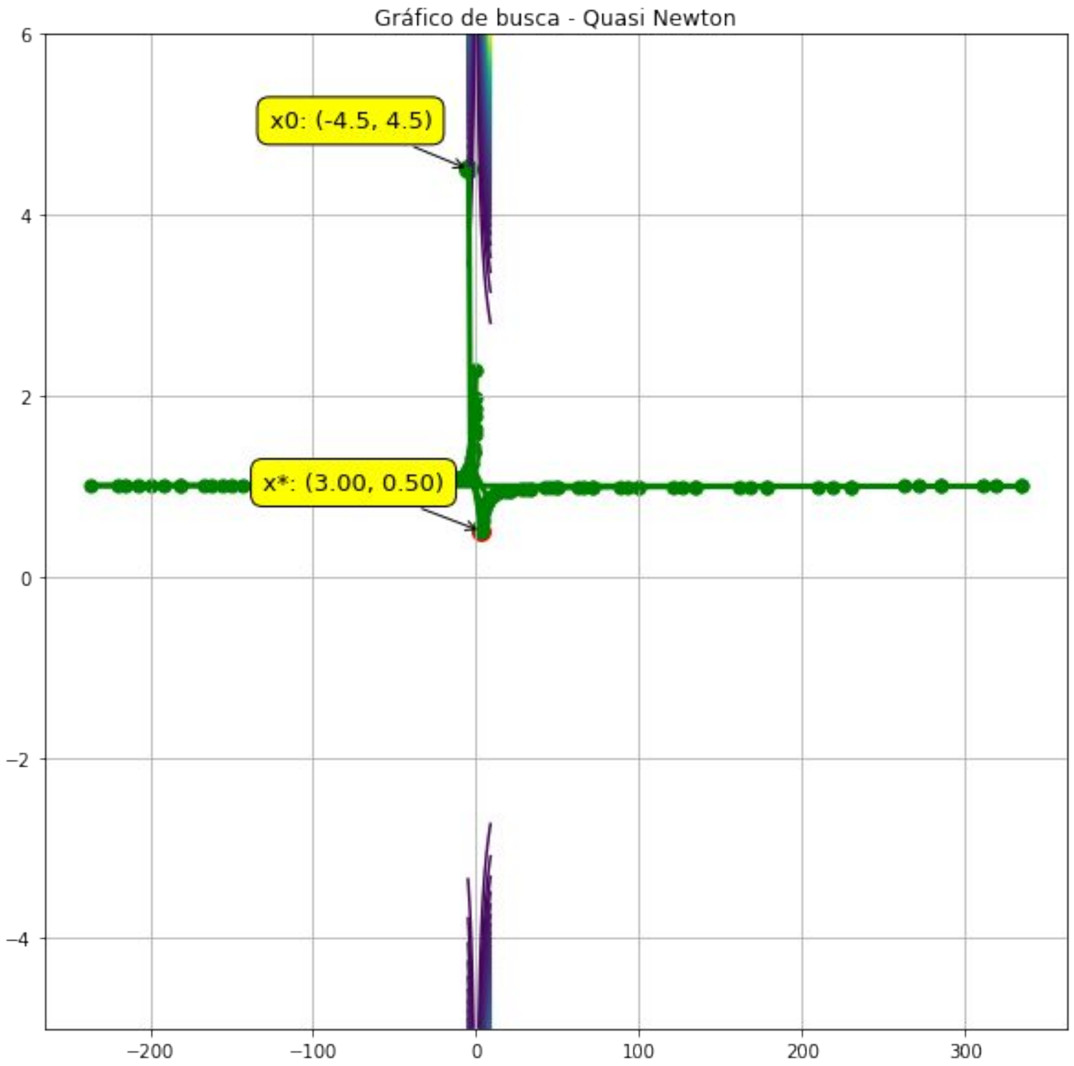


Gráfico de busca - Quasi Newton
x0: (-4.5, 4.5)
x*: (3.00, 0.50)
6
4
2
0
−2
−4
−200
−100
0
100
200
300

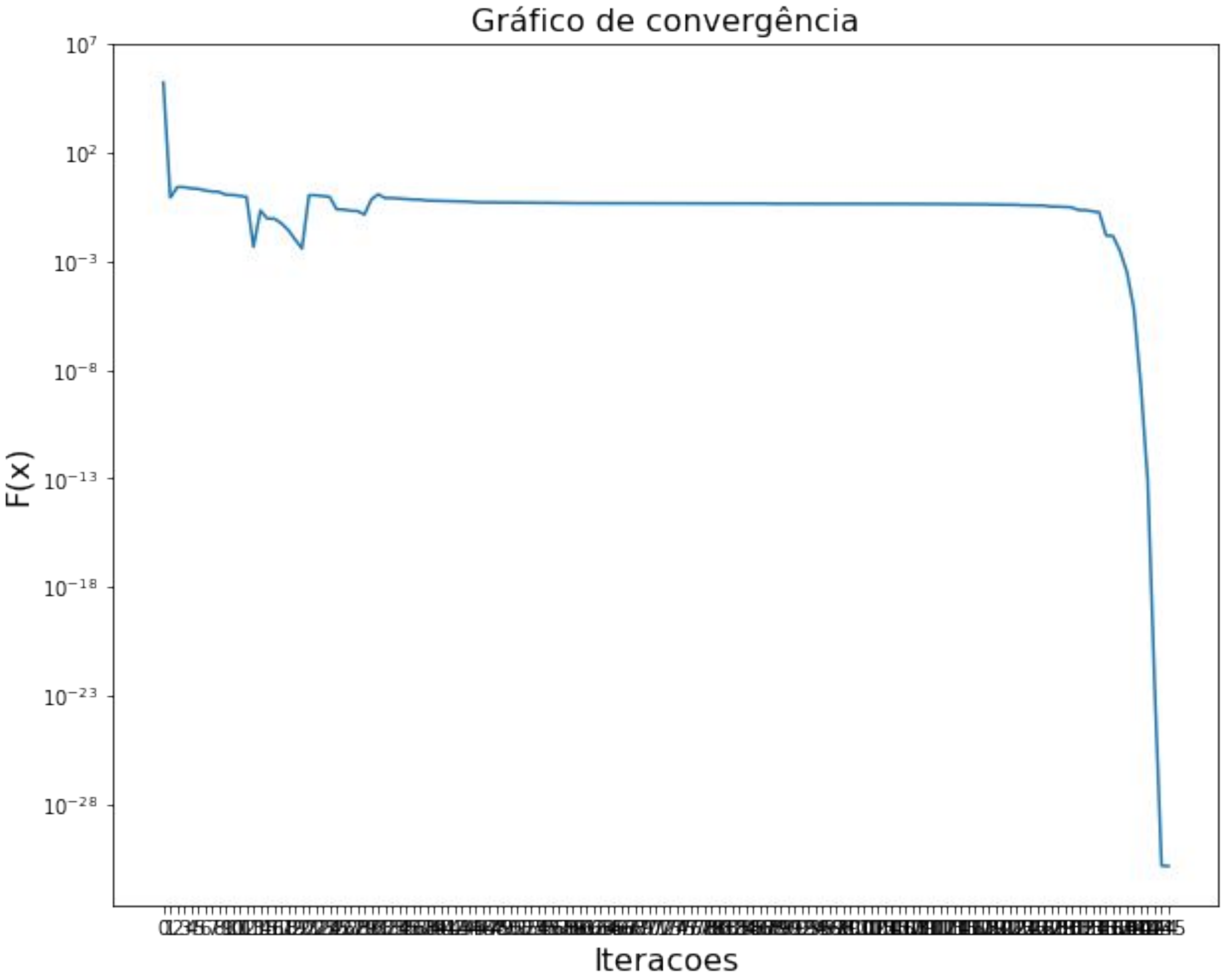


Gráfico de convergência
F(x)
Iteracoes
10^{7}
10^{2}
10^{-3}
10^{-8}
10^{-13}
10^{-18}
10^{-23}
10^{-28}

Conclusão

- Método eficiente, confiável e aplicável
- Recomendado para aplicações gerais de projetos de engenharia

Referências

- TAKAHASHI, R. H. C. Otimização Escalar e Vetorial (notas de aula), 2007. http://150.164.25.15/~taka/
- LUENBERGER, D. Linear and Nonlinear Programming. Addison Wesley, 1984.
- FRIEDLANDER, A. Elementos de Programação Não-Linear (notas de aula), http://www.ime.unicamp.br/~friedlan/livro.htm
- SymPy Documentation Release 1.9, https://docs.sympy.org/latest/index.html