Projeto Computacional

*A*lgoritmos para minimização irrestrita

Gabriel Cunha Marchetti
g251055@dac.unicamp.br

Leonardo Rangel De Albuquerque
l236296@dac.unicamp.br

Matheus Queiroz Mota
m251495@dac.unicamp.br

Introdução

O projeto tem como finalidade implementar alguns dos métodos de otimização irrestrita apresentados na matéria, dentre eles temos que os métodos que aqui estão contemplados são: **Método do Gradiente**, **Método de Newton** e os métodos do tipo *Quasi-Newton* **Correção de Posto Um (CP1)** e **Davidon, Fletcher e Powell (DFP)**. Lembramos que estes métodos são utilizados para minimizar funções do tipo $f : \mathbb{R}^n \to \mathbb{R}$ e $f \in \mathbb{C}^2$. Primeiro vamos apresentar as funções que utilizaremos para testar a implementação dos métodos. Depois iremos apresentar a implementação de cada um dos métodos. Por fim concluiremos o trabalho apresentando os resultados observados.

A função quadrática.

Em primeiro lugar, vamos começar pela implementação da função mais simples dentre as quatro, a função quadrática. Ela pode ser facilmente escrita como:

$$f(x) = \sum_{i=1}^{n} i \cdot x_i^2 \tag{1}$$

```python
def quadratic_function(x0):
    func = 0
    for i in range(len(x0)):
        val = (i+1)*x0[i]**2
        func = func +val
        val = 0
    return func
```

Figura 1: Código usado para implementar a função quadrática dentro do python.

Veja que a variável de estoque `func` apenas armazenará o valor da função. `val` será uma variável auxiliar que armazenará o valor temporário de $i \cdot x_i^2$ e precisamos nos atentar que o índice da multiplicação na atribuição do valor de `val` precisa ser $(i+1)$, pois a indexação dentro do python começa em ZERO.

Precisamos agora gerar o gradiente dessa função. Para isso podemos simplesmente escrever a derivada parcial relativa a algum x_j.

$$\frac{\partial}{\partial x_j} f(x) = \frac{\partial}{\partial x_j} \left\{ \sum_{i=1}^{n} i \cdot x_i^2 \right\} = 2 \cdot j \cdot x_j \tag{2}$$

Portanto, dentro do gradiente teremos:

$$\nabla f(x) = \begin{bmatrix} 2 \cdot 1 \cdot x_1 \\ 2 \cdot 2 \cdot x_2 \\ 2 \cdot 3 \cdot x_3 \\ \vdots \\ 2 \cdot n \cdot x_n \end{bmatrix} \tag{3}$$

Desse modo, agora podemos criar uma nova função que será usada para implementar o gradiente da função quadrática:

```python
def quadratic(x0):

    x_grad = np.zeros(len(x0))
    for i in range(len(x0)):
        x_grad[i] = 2 * (i+1) * x0[i]
    return x_grad
```

Figura 2: Código usado para implementar o gradiente da função quadrática dentro do python.

Veja que a ideia aqui é algo bem simples, apenas inicializamos o vetor `x_grad` que armazenará as informações do nosso gradiente, assim como também temos que tomar cuidado com os índices.

Por fim, podemos simplesmente computar a hessiana da função quadrática. Para isso, vamos lembrar que:

$$\nabla^2 f(x) = \begin{bmatrix} \partial_{x_1} \nabla^T f(x) \\ \partial_{x_2} \nabla^T f(x) \\ \partial_{x_3} \nabla^T f(x) \\ \vdots \\ \partial_{x_n} \nabla^T f(x) \end{bmatrix} = \begin{bmatrix} \frac{\partial^2}{\partial x_1 \partial x_1} f(x) & \frac{\partial^2}{\partial x_2 \partial x_1} f(x) & \frac{\partial^2}{\partial x_3 \partial x_1} f(x) & \dots & \frac{\partial^2}{\partial x_n \partial x_1} f(x) \\ \frac{\partial^2}{\partial x_1 \partial x_2} f(x) & \frac{\partial^2}{\partial x_2 \partial x_2} f(x) & \frac{\partial^2}{\partial x_3 \partial x_2} f(x) & \dots & \frac{\partial^2}{\partial x_n \partial x_2} f(x) \\ \frac{\partial^2}{\partial x_1 \partial x_3} f(x) & \frac{\partial^2}{\partial x_2 \partial x_3} f(x) & \frac{\partial^2}{\partial x_3 \partial x_3} f(x) & \dots & \frac{\partial^2}{\partial x_n \partial x_3} f(x) \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{bmatrix} \tag{4}$$

Dessa forma, podemos olhar para a primeira linha!

$$\frac{\partial^2}{\partial x_1 \partial x_1} f(x) \overset{(2)}{=} \frac{\partial}{\partial x_1} \{ 2 \cdot 1 \cdot x_1 \} = 2 \cdot 1$$

Na segunda linha:

$$\frac{\partial^2}{\partial x_2 \partial x_1} f(x) \overset{(2)}{=} \frac{\partial}{\partial x_2} \{ 2 \cdot 1 \cdot x_1 \} = 0$$

Na terceira linha:

$$\frac{\partial^2}{\partial x_3 \partial x_1} f(x) \overset{(2)}{=} \frac{\partial}{\partial x_3} \{ 2 \cdot 1 \cdot x_1 \} = 0$$

Notemos que isso irá se repetir até a derivada parcial com relação a x_n. Desse modo, a primeira linha será:

$$\left[\nabla^2 f(x) \right]_{\text{primeira linha}} = \begin{bmatrix} 2 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Para as outras linhas o padrão irá se repetir, entretanto, temos que tomar cuidado com o índice i multiplicando, de modo que:

$$\frac{\partial^2}{\partial x_j \partial x_i} f(x) = \begin{cases} 0, \text{ se } i \neq j \\ 2 \cdot i, \text{ caso contrário.} \end{cases} \tag{5}$$

Portanto, usando a matriz resultante será:

$$\nabla^2 f(x)_{\text{Quadrática}} = \begin{bmatrix} 2 \cdot 1 & 0 & 0 & \dots & 0 \\ 0 & 2 \cdot 2 & 0 & \dots & 0 \\ 0 & 0 & 2 \cdot 3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 2 \cdot n \end{bmatrix} \tag{6}$$

Implementando a função 6 temos:

```
def Hessian_quadratic(x0):

    x_hessian =np.zeros(len(x0))
    for i in range(len(x0)):
        x_hessian[i] = 2 * (i+1)
    hessian = np.diag(x_hessian)
    return hessian
```

Figura 3: Código usado para implementar a hessiana da função quadrática dentro do python.

Veja que estamos primeiro gerando um vetor que será usado para criar uma matriz hessiana que é diagonal. Desse modo conseguimos finalizar a implementação das funções utilizadas no nosso programa para a função quadrática.

A função Rosenbrook.

Esta é a função que admite mais problemas relacionados com relações às suas derivadas. A função é definida como:

$$f(x) = \sum_{i=1}^{n/2} \left[10(x_{2i} - x_{2i-1}^2)^2 + (x_{2i-1} - 1)^2\right]$$

Veja que não podemos simplesmente derivar com relação à x_j visto temos tanto as variáveis "pares"e "ímpares". Desse modo, temos que:

$$\frac{\partial}{\partial x_{2j}} \left\{ \sum_{i=1}^{n/2} \left[10(x_{2i} - x_{2i-1}^2)^2 + (x_{2i-1} - 1)^2\right] \right\} = 20(x_{2j} - x_{2j-1}^2)$$

$$\frac{\partial}{\partial x_{2j-1}} \left\{ \sum_{i=1}^{n/2} \left[10(x_{2i} - x_{2i-1}^2)^2 + (x_{2i-1} - 1)^2\right] \right\} = -40x_{2j-1}(x_{2i} - x_{2i-1}^2) + 2(x_{2i-1} - 1)$$

Portanto, o gradiente ficará algo como:

$$\nabla f(x) = \begin{bmatrix} -40x_1(x_2 - x_1^2) + 2(x_1 - 1) \\ 20(x_2 - x_1^2) \\ -40x_3(x_4 - x_3^2) + 2(x_3 - 1) \\ 20(x_4 - x_3^2) \\ \vdots \\ -40x_{n-1}(x_n - x_{n-1}^2) + 2(x_{n-1} - 1) \\ 20(x_n - x_{n-1}^2) \end{bmatrix} \tag{7}$$

Veja que agora, nossa matriz ficará uma matriz separada por blocos 2x2, visto que cada componente depende do gradiente depende de duas variáveis específicas.

$$\nabla^2 f(x) = \begin{bmatrix} 120x_1^2 - 40x_2 + 2 & -40x_1 & 0 & 0 & \dots \\ -40x_1 & 20 & 0 & 0 & \dots \\ 0 & 0 & -120x_3^2 - 40x_4 + 2 & -40x_3 & \dots \\ 0 & 0 & -40x_3 & 20 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{8}$$

A implementação dessas funções pode ser feita do seguinte modo:

```python
def rosenbrok_function(x):
    func = 0
    if len(x) % 2 == 1:
        print("O vetor precisa ter tamanho par!")
    dim = int(len(x) / 2)
    for i in range(dim):
        par = 2*i
        impar = 2*i + 1
        func += 10 * ( x[impar] - x[par] ** 2 ) ** 2 + ( x[par] - 1 )**2
    return func
```

Figura 4: Código usado para implementar a função Rosenbrook.

```python
def rosenbrok(x):
    if len(x) % 2 == 1:
        print("O vetor precisa ter um n mero par de componentes.")
    x_grad = np.zeros(len(x))
    dim = int( len(x) / 2 )
    for i in range(dim):
        par = 2*i
        impar = 2*i + 1
        x_grad[par] = -40*(x[impar] - x[par]**2)*x[par] + 2 * ( x[par] - 1 )
        x_grad[impar] = 20*(x[impar] - x[par]**2)
    return x_grad
```

Figura 5: Código usado para implementar o gradiente da função Rosenbrook.

```python
def hessian_rosenbrok(x):
    if len(x) % 2 == 1:
        print("O vetor precisa ter n mero de componentes pares.")
    dim = len(x)
    rosen_hessian = np.zeros( (dim, dim) )
    dim = int( dim / 2 )
    for i in range(dim):
        par = 2*i
        impar = 2*i + 1
        rosen_hessian[par, par] = 120*x[par]**2 - 40*x[impar] + 2
        rosen_hessian[impar, impar] = 20
        rosen_hessian[impar, par] = rosen_hessian[par, impar] = -40*x[par]
    return rosen_hessian
```

Figura 6: Código usado para implementar a hessiana da função Rosenbrook.

A função Styblinsky-Tang.

Esta função é facilmente definida como:

$$f(x) = \sum_{i=1}^{n} \left[x_i^4 - 16x_i^2 + 5x_i \right]$$

Usando o mesmo procedimento, temos que:

$$\frac{\partial}{\partial x_j} f(x) = \frac{\partial}{\partial x_j} \left[\sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i \right] = 4x_j^3 - 32x_j + 5$$

Assim, usando o gradiente ficará:

$$\nabla f(x) = \begin{bmatrix} 4x_1^3 - 32x_1 + 5 \\ 4x_2^3 - 32x_2 + 5 \\ 4x_3^3 - 32x_3 + 5 \\ \vdots \\ 4x_n^3 - 32x_n + 5 \end{bmatrix} \quad \nabla^2 f(x) = \begin{bmatrix} 12x_1^2 - 32 & 0 & 0 & \dots & 0 \\ 0 & 12x_2^2 - 32 & 0 & \dots & 0 \\ 0 & 0 & 12x_3^2 - 32 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 12x_n^2 - 32 \end{bmatrix} \tag{9}$$

Desse modo, recebemos os seguintes códigos:

```
def tang_function(x0):
    func =0
    for i in range(len(x0)):
        val = x0[i]**4 -16*x0[i]**2 + 5*x0[i]
        func = func +val
        val =0
    return func
```

Figura 7: Código usado para implementar a função da Styblisnky.

O raciocínio é análogo ao desenvolvido na função quadrática.

```
def tang(x0):
    x_grad = np.zeros(len(x0))

    for i in range(len(x0)):
        x_grad[i] = (4*(x0[i])**3) -(32*(x0[i])) + 5
    return x_grad
```

Figura 8: Código usado para implementar o gradiente da função styblinsky-tang.

O código usado para implementar a função Styblisnky-Tang.

```
def hessian_tang(x0):
    x_hessian =np.zeros(len(x0))
    for i in range(len(x0)):
        x_hessian[i] = (12*x0[i]**3) -32
    hessian = np.diag(x_hessian)
    return hessian
```

Figura 9: Código usado para implementar a hessiana da função quadrática dentro do python.

A função Rastrigin.

Esta função é definida como:

$$f(x) = \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i)$$

Fazendo que:

$$\frac{\partial}{\partial x_j} f(x) = \frac{\partial}{\partial x_j}\left\{\sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i)\right\} = 2\cdot x_j + 10\sin(2\pi x_j)\cdot 2\pi = 2x_j + 20\pi\sin(2\pi x_j)$$

O gradiente dessa função fica:

$$\nabla f(x) = \begin{bmatrix} 2x_1 + 20\pi\sin(2\pi x_1) \\ 2x_2 + 20\pi\sin(2\pi x_2) \\ 2x_3 + 20\pi\sin(2\pi x_3) \\ \vdots \\ 2x_n + 20\pi\sin(2\pi x_n) \end{bmatrix}$$

Enquanto a matriz hessiana dessa função fica:

$$\nabla^2 f(x) = \begin{bmatrix} 2+40\pi^2\cos(2\pi x_1) & 0 & 0 & \dots & 0 \\ 0 & 2+40\pi^2\cos(2\pi x_2) & 0 & \dots & 0 \\ 0 & 0 & 2+40\pi^2\cos(2\pi x_3) & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 2+40\pi^2\cos(2\pi x_n) \end{bmatrix}$$

A implementação dessas funções fica:

```
def Rastrigin_function(x0):
    func =0
    for i in range(len(x0)):
        val = x0[i]**2 -10*(math.cos(2 * math.pi * x0[i]))
        func = func +val
        val =0
    return func
```

Figura 10: Código usado para implementar a função rastrigin dentro do python.

```
def Rastrigin(x0):
    x_grad = np.zeros(len(x0))
    x_hessian =np.zeros(len(x0))
    for i in range(len(x0)):
        x_grad[i] = 2*x0[i] + 20*(math.pi)*(math.sin(2*math.pi*x0[i]))
    return x_grad
```

Figura 11: Código usado para implementar a função gradiente dentro do python.

```
def hessian_Rastrigin(x0):
    x_hessian =np.zeros(len(x0))
    for i in range(len(x0)):
        x_hessian[i] = 2 + 40 * (math.pi)**2 * (math.cos(2 * math.pi * x0[i]))

    hessian = np.diag(x_hessian)
    return x_hessian
```

Figura 12: Código usado para implementar a função hessian dentro do python.

Dentro do nosso código fonte existem algumas rotinas de testes que nos fazem acreditar que as implementações estão corretas!

Os métodos de otimização.

O primeiro método abordado foi o método do gradiente devido a sua simplicidade, praticidade e usabilidade. Nele fizemos o seguinte código:

```
#gradiente
def gradiente(x0,grad,func, display):
    k=0
    while np.linalg.norm(grad(x0)) >= epslon and k < M:
        t = 1
        gradiente = grad(x0)
        d_k = -1*gradiente
        x1 = x0 + t*d_k
        armijo_aux = alpha*(np.dot(gradiente, d_k))
        while func(x1) > func(x0) + t*armijo_aux:
            t = t*sigma
            x1 = x0 + t*d_k
        x0= x1
        k = k + 1

        if(display):
            print(f"Valor da fun   o: {func(x0)}")
            print(f"Valor da norma do gradiente: {np.linalg.norm(grad(x0))}")

    return x1, func(x1)
```

Figura 13: Código do Gradiente Descendente.

Podemos perceber que a função é muito similar com a função fornecida na própria folha dos métodos. Desse modo não entraremos muito em detalhes, apenas que há rotinas de testes para as funções dentro dos códigos.

O segundo método abordado foi o método de Newton em que há que várias mudanças com relação ao método do gradiente. Segue o código do nosso método de Newton desenvolvido.

```
#newton
def newton(x0,grad,hessian,func):
    k=0
    n =len(x0)
    g = grad(x0)
    norm_g = np.linalg.norm(g)
    while norm_g>=epslon and k<M:
        mu=0
        h = hessian(x0)
        while True:
            try:
               d_k = np.linalg.solve((h + mu*np.eye(n)), -g)
               norm_dk=np.linalg.norm(d_k)
        #print(f'{k}: {d_k}')
            except np.linalg.LinAlgError:
                mu = max(2*mu,rho)
                continue
            if np.dot(g,d_k) > -gama*norm_g*norm_dk:
                mu = max(2*mu,rho)
                continue
            break
        if norm_dk < beta*norm_g:
            d_k = beta*(norm_g/norm_dk)*d_k
        t=1

        x1 = x0 + t*d_k
        while func(x1) > func(x0) + alpha*t*(np.dot(g,d_k)):
            t *= sigma
            x1 = x0 +t*d_k
        #print(f'Itera  o{k} f(xk) ={np.round(func(x1),4)} e xk = {np.round(x1,4)},
    grad(f(x1) = {grad(x1)}')

        x0=x1
        g = grad(x0)
        norm_dk = np.linalg.norm(d_k)
        norm_g=np.linalg.norm(g)
        k+=1
    print(f'A fun  o convergiu na Itera  o {k} f(x_k) ={np.round(func(x1),4)}, x_k =
    {np.round(x1,4)} e o gradiente de x_k  : {np.round(grad(x1),4)}')

    return x1, func(x1)
```

Figura 14: Código do Método de Newton.

Veja que esse código já é mais robusto, de modo que agora precisamos resolver sistemas lineares, caso eles tenham solução. Para isso usamos a biblioteca do python que lida com sistemas lineares, o `linalg`, que é invocado para resolve-los. Além disso usamos a função `try ... except` que tenta realizar algum comando e dada uma mensagem de erro tenta executar uma outra rotina de comandos para lidar com esse erro.

Por fim podemos apresentar os métodos *Quasi*-Newton. Primeiro iremos apresentar a implementação do método de correção de posto um que segue no bloco de código abaixo.

```python
#secantecp1
def secantecp1(x0,grad,func):
    k=0
    H0 = np.eye(len(x0))
    g = grad(x0)
    while np.linalg.norm(grad(x0))>epslon and k<M:

        d_k = - H0@g
        norm_dk = np.linalg.norm(d_k)
        norm_g=np.linalg.norm(g)
        if np.dot(g,d_k) > -gama*norm_g*norm_dk:
            d_k = -grad(x0)
            H0 = np.eye(len(x0))
           # print(f'foi pro grade em {k}')
        if np.linalg.norm(d_k) < beta*np.linalg.norm(grad(x0)):
            d_k = beta*(np.linalg.norm(grad(x0))/np.linalg.norm(d_k))*d_k
           # print(f'normalizou em {k}')
        t=1
        x1 = x0 + t*d_k
        while func(x1) > func(x0) + alpha*t*(np.dot(g,d_k)):
            t = t*sigma
            x1 = x0 +t*d_k
            sk = x1 -x0
            yk = grad(x1) - grad(x0)
            zk = H0@yk
            wk = sk-zk


        if np.dot(wk,yk)>0:

            H1 = H0 + np.outer(wk,wk)/np.dot(wk,yk)

        else:
            H1 = H0

        H0 = H1
        x0=x1
        g=grad(x0)

        k=k+1

    print(f'A fun  o convergiu na Itera  o {k} f(x_k) ={np.round(func(x1),4)}, x_k =
    {np.round(x1,4)} e o gradiente de x_k  : {np.round(grad(x1),4)}')



    return x1
```

Figura 15: Código do método da secante CP1.

Veja que esse código já é autossuficiente com relação ao processo, principalmente se seguirmos a folha de orientação do projeto, desse modo dispensamos comentários para o método. Por fim apresentamos a variação do método da secante DFP.

```python
#secanteDFP
def secantedfp(x0,grad,func):
    k=0
    H0 = np.eye(len(x0))
    g = grad(x0)
    while np.linalg.norm(grad(x0))>epslon and k<M:

        d_k = - H0@g
        norm_dk = np.linalg.norm(d_k)
        norm_g=np.linalg.norm(g)
        if np.dot(g,d_k) > -gama*norm_g*norm_dk:
            d_k = -grad(x0)
            H0 = np.eye(len(x0))
           # print(f'foi pro grade em {k}')
        if np.linalg.norm(d_k) < beta*np.linalg.norm(grad(x0)):
            d_k = beta*(np.linalg.norm(grad(x0))/np.linalg.norm(d_k))*d_k
           # print(f'normalizou em {k}')
        t=1
        x1 = x0 + t*d_k
        while func(x1) > func(x0) + alpha*t*(np.dot(g,d_k)):
```

```
            t = t*sigma
            x1 = x0 +t*d_k
            sk = x1 -x0
            yk = grad(x1) - grad(x0)
            zk = H0@yk
            wk = sk-zk

        if np.dot(sk,yk)>0:

            H1 = H0 + np.outer(sk,sk)/np.dot(sk,yk) - np.outer(zk,zk)/np.dot(zk,yk)


        else:
            H1 = H0

        H0 = H1
        x0=x1
        g=grad(x0)

        k=k+1

    print(f'A fun   o convergiu na Itera   o {k} f(x_k) ={np.round(func(x1),4)}, x_k =
    {np.round(x1,4)} e o gradiente de x_k   : {np.round(grad(x1),4)}')



    return x1
```

Figura 16: Código do método da secante DFP.

Igualmente autossuficiente e dispensando comentários aprofundados. Desse modo, podemos finalizar os comentários de implementação dos métodos e podemos analisar os efeitos práticos de cada método.

Comparando os métodos.

Uma maneira de avaliarmos os métodos é através de uma tabela comparando o tempo de execução dos métodos, assim como o número de iterações. Para essa comparação ser justa precisamos dar os mesmos pontos iniciais.

Primeiro devemos executar os métodos para a função quadrática, para isso iremos primeiro escolher algum ponto aleatório próximo ao ponto de convergência para testar o código na vizinhança de um candidato a ótimo local, para isso geramos um ponto aleatório na vizinhança do vetor ótimo da quadrática.

-	Gradiente	Newton	Secante CP1	Secante DFP
Tempo (ms)	6.73	0.47	8.23	9.36
Iterações	46	1	46	59

Primeiramente, como pode ser visto dentro do código, todos os métodos convergiram para o ótimo local. além disso, podemos ver que o método de Newton se destacou muito com relação aos outros, pois seu tempo estava uma ordem de grandeza menor, mesmo sendo um método que exige mais capacidade computacional no sentido de termos que calcular a Hessiana e resolver um sistema linear. Apesar disso, pelo trabalho que o método de Newton tem para escolher uma boa direção de descida ele consegue realizar o método em apenas uma iteração, o que é algo esperado. O método de Newton aproxima uma função por uma aproximação de Taylor de ordem 2 e calcula o minimizador dessa aproximação. Assim, como a função Quadrática aqui estudada é igual a sua aproximação por Taylor de ordem 2, o método atinge exatamente o mínimo global em apenas uma iteração.

Além disso, podemos observar que para esse caso a simplicidade e agilidade do método do gradiente acabou gerando mais velocidade e menos iterações que os métodos CP1 e DFP. A simplicidade do método do gradiente pode ser evidenciada se olharmos que o número de iterações é o mesmo do CP1 enquanto temos que o gradiente é 2ms mais rápido.

-	Gradiente	Newton	Secante CP1	Secante DFP
Tempo (ms)	72.44	1.63	51.62	11.49
Iterações	200(MAX)	11	200(MAX)	53

Com relação aos dados de otimização da função de Rosenbrook podemos ver agora que o método da secante DFP também começou a se destacar com relação ao método do gradiente e a secante CP1. Desse modo, podemos ver como a correção de posto maior e o desenvolvimento teórico são transmitidos no código. Além disso, podemos observar que o grau de dificuldade de otimização da função de Rosenbrook é muito maior se comparado com a função quadrática, uma vez que os métodos tanto do gradiente, como da secante CP1 não conseguiram convergir para o número de iterações máxima (200).

Além disso, foi observado justamente que apenas os métodos de Newton e Secante DFP conseguiram convergir para uma norma de gradiente satisfatória.

Para pontos mais dispersos da região do ponto ótimo, isto é, mais distantes do ponto ótimo não vemos grandes diferenças em questão de tempo e iterações.

-	Gradiente	Newton	Secante CP1	Secante DFP
Tempo (ms)	42.17	1.10	51.58	10.89
Iterações	200(MAX)	8	200(MAX)	54

Mostrando que a dificuldade, em questão de otimização, está justamente em superar esse grande "vale"da função de Rosenbrook. Vejamos que o método de Newton apresentou melhor performance se comparado com pontos na vizinhança o que reforça ainda mais o comportamento estacionário que essa função possui quando tenta alcançar o mínimo.

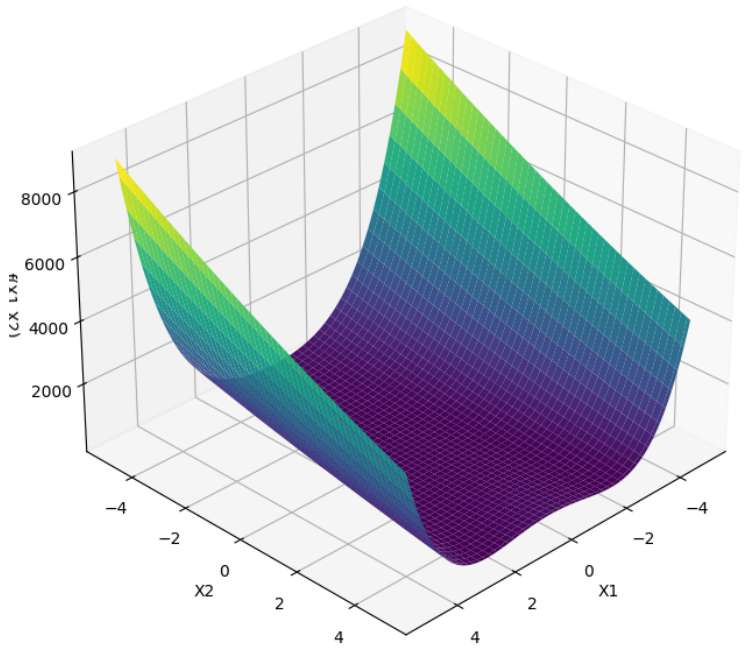

Figura 16: Função de Rosenbrook gerada pelo grupo.

Como podemos ver na Figura 16 a região "central"(ou crucial) da Rosenbrook apresenta pequenas normas do gradiente para esse caso de pontos em duas dimensões. Portanto, os nossos métodos terão mais dificuldade de encontrar o ótimo dessa função mesmo, principalmente se o caso se aproximar da Figura 16 pois a região de mínimo não é tão claro. Isso ocorre mesmo em dimensões maiores, uma vez que as funções apresentarão problemas similares.

Agora podemos olhar para a tabela de performance dos métodos para a função de Styblinsky-Tang:

-	Gradiente	Newton	Secante CP1	Secante DFP
Tempo (ms)	5.31	1.78	6.51	8.25
Iterações	11	8	11	21

Olhando para a tabela observamos que todas as funções tiveram facilidade de encontrar o mínimo da função de Styblinsky-Tang o que é curioso, pois ela apresenta comportamento similar ao "vale"da Rosenbrook como pode ser visualizado na Figura 17.

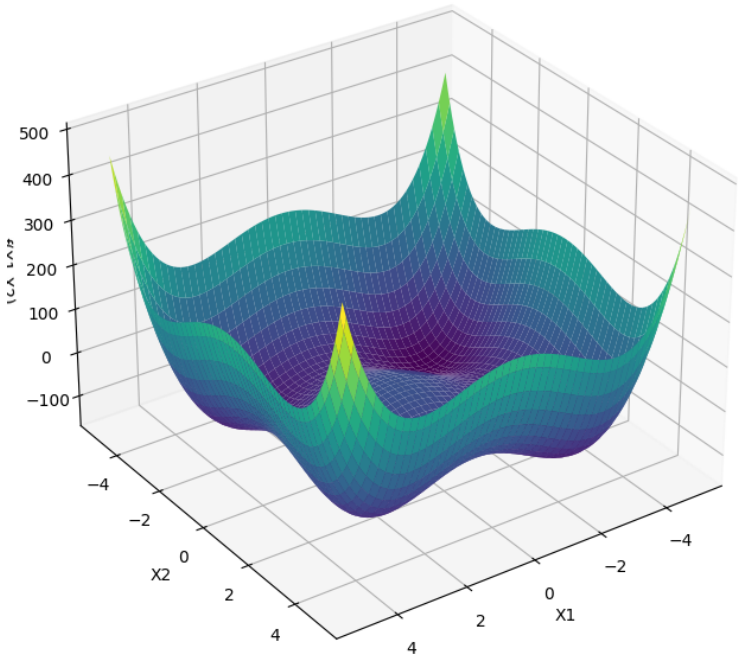

Figura 17: Visualização do comportamento da função de Styblinsky-Tang.

Vejamos que a função apresenta um grande "vale"central que apresenta algumas regiões de menor valor e um certo aumento no valor da função no centro desse vale. Acreditamos que por conta desse comportamento (pequena elevação dentro do vale) os métodos não tem dificuldade para encontrar o mínimo local dessa função. Novamente, esse conceito é exclusivo para superfícies tridimensionais, podemos estender esse conceito para funções de ordem maior.

Por fim, vamos ver os resultados apresentados pela função de Rastrigin.

-	Gradiente	Newton	Secante CP1	Secante DFP
Tempo (ms)	12.09	21.92	14.66	37.52
Iterações	28	200(MAX)	28	200(MAX)

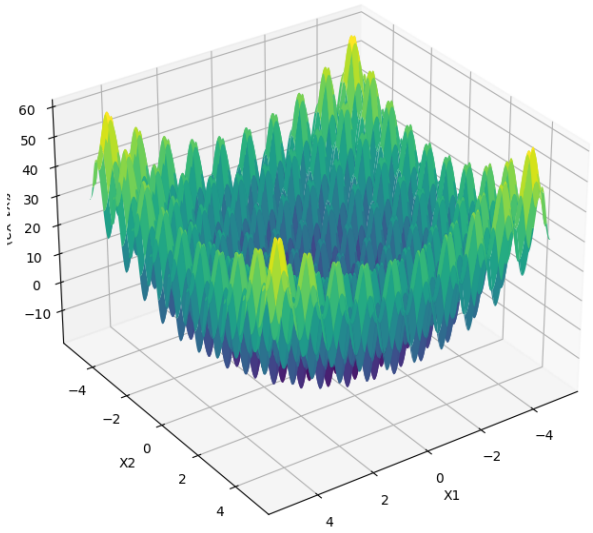

Figura 18: Visualização do comportamento da função de Rastrigin (retirada da internet).

Sabemos para esse caso que o valor da função no mínimo deve ser $f(x^*) = 10n$, como estamos testando para vetores com tamanho 10, então os valores deveriam se aproximar de 80. Vejamos que o método da secante DFP apresenta melhor valor de função conseguindo chegar em $f(x_k) = 88.0436$.

Como podemos observar na Figura 18 essa função apresenta um comportamento periódico e isso claramente atrapalha o método de newton para convergir, uma vez que esse foi o único teste que o número máximo de iterações foi alcançado.

Além disso, podemos observar que métodos mais simples como o gradiente ou a correção do posto um apresentam comportamento similar. Além disso, o método da secante DFP também apresentou certa queda na performance com relação ao aumento das oscilações do problema.

Agora vamos discutir com relação à variação dos pontos iniciais e da precisão para critério de parada, isto é, o ϵ. Primeiro vamos comparar para precisões diferentes e ver como fica o comportamento da função.Para isso vamos variar a dimensão do problema para que os métodos sejam mais influenciados pelos parâmetros que queremos verificar a sensibilidade.

ε	Método	Tempo (s)	Iterações	x_k	$f(x_k)$
10e-6	Gradiente	0.003	2	[0. 0.]	0.0
	Newton	0.0003	1	[0. 0.]	0.0
	Secante CP1	0.002	2	[0. 0.]	0.0
	Secante DFP	0.005	12	[0. 0.]	0.0
10e-4	Gradiente	0.0009	2	[0. 0.]	0.0
	Newton	0.001	1	[0. 0.]	0.0
	Secante CP1	0.0011	2	[0. 0.]	0.0
	Secante DFP	0.0039	11	[-0. 0.0003]	0.0
10e-3	Gradiente	0.001	2	[0. 0.]	0.0
	Newton	0.001	1	[0. 0.]	0.0
	Secante CP1	0.0076	2	[0. 0.]	0.0
	Secante DFP	0.0029	9	[0.0001 0.0014]	0.0

Tabela 1: Comparação dos métodos para $X_0 = [10, 10]$ e $n = 2$ com diferentes valores de ε para a função de Quadrática.

Podemos observar, na Tabela , que todos os métodos convergiram de modo satisfatório para os pontos estacionários observados. Além disso, vemos que os tempos foram em ordem de grandeza similar e a simplicidade do gradiente foram destacadas para ϵ's maiores. Além do mais, como já esperado, o método de Newton converge em uma iteração independentemente do ϵ em questão, visto que a convergência para esta função é exata.

Ainda mais podemos ressaltar que as convergências para os pontos estacionários no método DFP ficaram limitados por conta do ϵ visto que o critério de parada da comparação da norma do gradiente com o ϵ foi satisfeito com número similares de iterações quando reduzimos o parâmetro.

ε	Método	Tempo (s)	Iterações	x_k	$f(x_k)$
10e-6	Gradiente	0.049	22	[0. 0. -0. 0.]	0.0
	Newton	0.003	1	[0. 0. 0. 0.]	0.0
	Secante CP1	0.01	22	[0. 0. -0. 0.]	0.0
	Secante DFP	0.013	24	[0. 0. 0. 0.]	0.0
10e-4	Gradiente	0.006	15	[0.0003 0. 0.0001 0.]	0.0
	Newton	0.0005	1	[0. 0. 0. 0.]	0.0
	Secante CP1	0.005	15	[0.0003 0. 0.0001 0.]	0.0
	Secante DFP	0.011	19	[-0.0002 -0. 0. -0.]	0.0
10e-3	Gradiente	0.002	12	[0.0022 0. -0.0007 0.]	0.0
	Newton	0.0004	1	[0. 0. 0. 0.]	0.0
	Secante CP1	0.006	12	[0.0022 0. -0.0007 0.]	0.0
	Secante DFP	0.008	19	[-0.0002 -0. 0. -0.]	0.0

Tabela 2: Comparação dos métodos para $X_0 = [6, 6, 6, 6]$ e $n = 4$ com diferentes valores de ε para a Quadrática.

Vejamos agora pela Tabela que aumentando a dimensão do problema vemos que o método de Newton se destacou muito com relação aos outros métodos. Ainda, podemos observar como a diminuição dos parâmetros ϵ influenciaram para encontrar o ponto estacionário. Para mais, os tempos não variaram muito.

ε	Método	Tempo (s)	Iterações	x_k	$f(x_k)$
10e-6	Gradiente	0.083	200	[2.6342 6.9858]	2.6925
	Newton	0.004	24	[1. 1.]	0.0
	Secante CP1	0.126	200	[2.6342 6.9858]	2.6925
	Secante DFP	0.055	155	[1. 1.]	0.0
10e-4	Gradiente	0.045	200	[2.6342 6.9858]	2.6925
	Newton	0.004	23	[1.0001 1.0001]	0.0
	Secante CP1	0.068	200	[2.6342 6.9858]	2.6925
	Secante DFP	0.03	131	[0.999 0.9979]	0.0
10e-3	Gradiente	0.04	200	[2.6342 6.9858]	2.6925
	Newton	0.008	23	[1.0001 1.0001]	0.0
	Secante CP1	0.109	200	[2.6342 6.9858]	2.6925
	Secante DFP	0.026	118	[0.9917 0.9832]	0.0001

Tabela 3: Comparação dos métodos para $X_0 = [10, 10]$ e $n = 2$ com diferentes valores de ε para a função de Rosenbrook.

ε	Método	Tempo (s)	Iterações	x_k	$f(x_k)$
10e-6	Gradiente	0.1	200	[2.3462 5.5438 2.3462 5.5438]	3.6552
	Newton	0.004	19	[1. 1. 1. 1.]	0.0
	Secante CP1	0.134	200	[2.3462 5.5438 2.3462 5.5438]	3.6552
	Secante DFP	0.024	38	[1. 1. 1. 1.]	0.0
10e-4	Gradiente	0.07	200	[2.3462 5.5438 2.3462 5.5438]	3.6552
	Newton	0.0	19	[1.0001 1.0001 1.0001 1.0001]	0.0
	Secante CP1	0.128	200	[2.3462 5.5438 2.3462 5.5438]	3.6552
	Secante DFP	0.022	31	[0.9997 0.9994 0.9997 0.9994]	0.0
10e-3	Gradiente	0.073	200	[2.3462 5.5438 2.3462 5.5438]	3.6552
	Newton	0.002	18	[1.0001 1.0001 1.0001 1.0001]	0.0
	Secante CP1	0.093	200	[2.3462 5.5438 2.3462 5.5438]	3.6552
	Secante DFP	0.009	27	[0.9977 0.9954 0.9977 0.9954]	0.0

Tabela 4: Comparação dos métodos para $X_0 = [6, 6, 6, 6]$ e $n = 4$, com diferentes valores de ε para Rosenbrook.

Vejamos que para a função de Rosenbrook, dados da Tabela , todos os métodos, tirando o de Newton, apresentaram muita dificuldade para convergir. Mostrando o quão difícil é superar o vale da função Rosenbrook, mesmo para precisões menores de ϵ.

Vemos que para uma dimensão do problema maior, presente na Tabela o método da Secante DFP apresentou melhora, ressaltando que o problema de ϵ para é um defeito para esse método.

ε	**Método**	**Tempo (s)**	**Iterações**	x_k, $f(x_k)$
10e-6	Gradiente	0.006	14	$[-2.9035, -2.9035]$, -156.6647
	Newton	0.003	7	$[2.7468, 2.7468]$, -100.1178
	Secante CP1	0.010	14	$[-2.9035, -2.9035]$, -156.6647
	Secante DFP	0.006	8	$[-2.9035, -2.9035]$, -156.6647
10e-4	Gradiente	0.007	12	$[-2.9035, -2.9035]$, -156.6647
	Newton	0.003	6	$[2.7468, 2.7468]$, -100.1178
	Secante CP1	0.011	12	$[-2.9035, -2.9035]$, -156.6647
	Secante DFP	0.006	7	$[-2.9035, -2.9035]$, -156.6647
10e-3	Gradiente	0.025	11	$[-2.9036, -2.9036]$, -156.6647
	Newton	0.001	6	$[2.7468, 2.7468]$, -100.1178
	Secante CP1	0.016	11	$[-2.9036, -2.9036]$, -156.6647
	Secante DFP	0.005	7	$[-2.9035, -2.9035]$, -156.6647

Tabela 5: Comparação dos métodos para $X_0 = [10, 10]$ e $n = 2$ com diferentes valores de ε para a função de Styblinsky–Tang

Para a função de Styblinsky-Tang, Tabela 5, vemos que todas as funções conseguiram superar facilmente o vale que a função apresenta, exceto o método de Newton que teve sua convergência para um mínimo local. Ademais, a variação do intervalo de tolerância ϵ não implicou em uma diferença notável no tempo e na quantidade de iterações para a convergência de todos os métodos.

ε	Método	Tempo (s)	Iterações	x_k	$f(x_k)$
10e-6	Gradiente	0.137	75	[2.7468 2.7468 2.7468 2.7468]	-200.2356
	Newton	0.019	6	[-2.9035 -2.9035 -2.9035 -2.9035]	-313.3293
	Secante CP1	0.041	75	[2.7468 2.7468 2.7468 2.7468]	-200.2356
	Secante DFP	0.013	8	[-2.9035 -2.9035 -2.9035 -2.9035]	-313.3293
10e-4	Gradiente	0.015	50	[2.7468 2.7468 2.7468 2.7468]	-200.2356
	Newton	0.0	6	[-2.9035 -2.9035 -2.9035 -2.9035]	-313.3293
	Secante CP1	0.05	50	[2.7468 2.7468 2.7468 2.7468]	-200.2356
	Secante DFP	0.003	7	[-2.9035 -2.9035 -2.9035 -2.9035]	-313.3293
10e-3	Gradiente	0.015	38	[2.7467 2.7467 2.7467 2.7467]	-200.2356
	Newton	0.002	5	[-2.9036 -2.9036 -2.9036 -2.9036]	-313.3293
	Secante CP1	0.026	38	[2.7467 2.7467 2.7467 2.7467]	-200.2356
	Secante DFP	0.003	7	[-2.9035 -2.9035 -2.9035 -2.9035]	-313.3293

Tabela 6: Comparação dos métodos para $X_0 = [6, 6, 6, 6]$ e $n = 4$ com diferentes valores de ε com A função de Styblinsky–Tang.

Agora com o aumento da dimensão do problema, vemos que na Tabela o método de Newton e da Secante DFP conseguiram convergir para o mínimo global e com poucas iterações com uma facilidade similar ao caso $n = 2$. Por outro lado o método do gradiente e da secante CP1 apresentaram alta variação em relação ao aumento do tempo e da quantidade de iterações com a mudança no valor de ϵ. Podemos justificar que ambos os métodos, o do gradiente e da secante CP1, apresentaram um comportamento similar em muitos dos nossos teste devido ao fato de usarmos matriz de ajuste como a identidade e isto faz com que esse método se aproxime muito do gradiente.

ε	**Método**	**Tempo (s)**	**Iterações**	x_k, $f(x_k)$
10e-06	Gradiente	0.002	1	$[0.0, 0.0]$, -20.0
	Newton	0.043	118	$[9.9487, 9.9487]$, 178.9832
	Secante CP1	0.003	1	$[0.0, 0.0]$, -20.0
	Secante DFP	0.005	1	$[0.0, 0.0]$, -20.0
10e-4	Gradiente	0.003	1	$[0.0, 0.0]$, -20.0
	Newton	0.023	81	$[9.9487, 9.9487]$, 178.9832
	Secante CP1	0.001	1	$[0.0, 0.0]$, -20.0
	Secante DFP	0.001	1	$[0.0, 0.0]$, -20.0
10e-3	Gradiente	0.000	1	$[0.0, 0.0]$, -20.0
	Newton	0.015	63	$[9.9487, 9.9487]$, 178.9832
	Secante CP1	0.001	1	$[0.0, 0.0]$, -20.0
	Secante DFP	0.002	1	$[0.0, 0.0]$, -20.0

Tabela 7: Comparação dos métodos para $X_0 = [10, 10]$ e $n = 2$ com diferentes valores de ε para a função de Rastrigin

Já de primeira vista, é notável que o comportamento dos métodos na função de Rastrigin é atípico. Todos os métodos, exceto o de Newton, convergem em apenas uma iteração independentemente do ϵ em questão. No caso do método de Newton, ele apresenta uma dificuldade exorbitante em encontrar um ponto estacionário, sendo que ela vai aumentando notadamente conforme diminuímos a tolerância. Isso se deve ao fato da função de Rastrigin não possuir uma boa aproximação por Taylor de ordem 2 localmente.

ε	Método	Tempo (s)	Iterações	x_k	$f(x_k)$
10e-6	Gradiente	0.046	1	[0. 0. 0. 0.]	-40.0
	Newton	0.240	200	[5.9696 5.9696 5.9696 5.9696]	103.272
	Secante CP1	0.003	1	[0. 0. 0. 0.]	-40.0
	Secante DFP	0.004	1	[0. 0. 0. 0.]	-40.0
10e-4	Gradiente	0.004	1	[0. 0. 0. 0.]	-40.0
	Newton	0.071	187	[5.9696 5.9696 5.9696 5.9696]	103.272
	Secante CP1	0.0	1	[0. 0. 0. 0.]	-40.0
	Secante DFP	0.0	1	[0. 0. 0. 0.]	-40.0
10e-3	Gradiente	0.001	1	[0. 0. 0. 0.]	-40.0
	Newton	0.010	144	[5.9696 5.9696 5.9696 5.9696]	103.272
	Secante CP1	0.0	1	[0. 0. 0. 0.]	-40.0
	Secante DFP	0.0	1	[0. 0. 0. 0.]	-40.0

Tabela 8: Comparação dos métodos para $X_0 = [6, 6, 6, 6]$ e $n = 4$ com diferentes valores de ε para a função de Rastrigin.

Novamente observamos as mesmas propriedades qualitativas do que da tabela anterior, com a diferença de que se começa a observar um aumento no tempo computacional no método de Newton devido ao fato de ser calculada uma Hessiana 4 vezes maior que no caso anterior a cada iteração.

Conclusão.

Como podemos ver durante parte importante da otimização é sabermos como criar métodos, afinal toda teoria desenvolvida seria inútil caso não pudéssemos contar com a velocidade dos cálculos. Podemos perceber como cálculos gigantescos podem ser medidos em questão de milissegundos. Além disso, podemos perceber que não há um método simples e certeiro para resolver um problema de otimização, como pudemos observar o método do gradiente pela sua simplicidade e facilidade conseguiu entregar resultados satisfatórios. Enquanto o método de Newton, que possui cálculos mais elaborados, aparentaria ser aquele com melhor desempenho, todavia, percebe-se que para determinados tipos de funções, como a de Rastrigin, tal método não conseguiu convergir para um ponto estacionário, mesmo com mais de 200 iterações. Os métodos CP1 e DFP parecem ser um intermédio entre o método do gradiente e o método de Newton e isso acontece pois de fato são, desse modo misturar as duas ideias se mostra um método eclético e eficiente.