

Trabalho 1

MEC 2403 - Otimização, Algoritmos e Aplicações na Engenharia
Mecânica

Gustavo Henrique Gomes dos Santos
gustavohgs@gmail.com

Professor: Ivan Menezes



PUC
RIO

Departamento de Engenharia Mecânica
PUC-RJ Pontifícia Universidade Católica do Rio de Janeiro
maio de 2023

Trabalho 1

MEC 2403 - Otimização, Algoritmos e Aplicações na Engenharia Mecânica

Gustavo Henrique Gomes dos Santos

maio de 2023

1 Introdução

1.1 Objetivos

Esse trabalho tem como objetivo a implementação, em Python, o teste dessa implementação, em diferentes funções e pontos iniciais, bem como a aplicação da implementação na minimização da Energia Potencial Total de um sistema de molas, dos seguintes métodos de otimização sem restrição :

- a. Univariante
- b. Powell
- c. Steepest Descent
- d. Fletcher-Reeves
- e. BFGS
- f. Newton-Raphson

2 Implementação

A estratégia adotada neste trabalho foi de implementar algoritmos que, dados inputs específicos de cada método, retornam a próxima direção de busca. Para a busca unidirecional na direção especificada por cada método, foram aproveitados e melhorados os códigos do passo constante e da seção áurea utilizados na resolução da Lista-1.

O código principal fica responsável pela chamada passo a passo dos métodos de OSR, recebendo uma direção de busca e repassando a mesma para a busca unidirecional. Com o valor de alpha obtido da seção áurea, um novo ponto é calculado e a convergência verificada.

2.1 Busca Unidirecional

Os algoritmos dos métodos do Passo Constante e da Seção Áurea foram implementados em um arquivo denominado `line_search_methods.py`. O seguinte pacote é necessário nesse arquivo:

```
import numpy as np
```

2.1.1 Passo Constante

Foi implementado um método que recebe como parâmetros de entrada um vetor direção (\vec{dir}), um ponto inicial (\vec{P}_1), a função que se deseja encontrar o mínimo ($f(\vec{P})$), um valor opcional de epsilon da máquina (ϵ default com valor 10^{-8}), um valor opcional de passo ($\Delta\alpha$ default com valor 0.01).

Para definir o sentido da busca, é feita uma comparação entre o valor de $f(\vec{P}_1 - \epsilon \vec{dir})$ e $f(\vec{P}_1 + \epsilon \vec{dir})$. Caso este último valor seja maior do que o primeiro, o sentido de busca considerado é o oposto do vetor direção ($-\vec{dir}$). Caso o primeiro seja o maior valor, o sentido do vetor direção é mantido na busca (\vec{dir}).

Com o passo default de 0.01 ou um qualquer outro passo desejado informado na passagem de parâmetro opcional, o método percorre o sentido de busca definido até encontrar um valor de $f(\vec{P}_1 + \alpha \vec{dir})$ que seja inferior

ao valor do próximo passo. Quando essa condição é alcançada, esse último valor de α é definido como mínimo (α_{min}).

Para garantir que a cada incremento de α não tenha sido pulado um mínimo da função, é feito uma comparação entre os valores de $f(\vec{P} - \epsilon \vec{dir})$ e $f(\vec{P})$, onde $\vec{P} = \vec{P}_1 + \alpha \vec{dir}$. Caso o último valor seja superior ao primeiro, o passo é desfeito e o α mínimo é considerado encontarado.

Caso uma das condições abaixo seja atingida, o algoritmo é interrompido e retorna o intervalo $[\alpha, \alpha + \Delta\alpha]$, fazendo os devidos ajustes para adequar os sinais de acordo com sentido de busca.

- $f(\vec{P} - \epsilon \vec{dir}) < f(\vec{P})$, com $\vec{P} = \vec{P}_1 + \alpha \vec{dir}$
- $f(\vec{P}_1 + \alpha \vec{dir}) < f(\vec{P}_1 + (\alpha + \Delta\alpha) \vec{dir})$

```
def passo_cte(direcao, P0, f, eps = 1E-8, step = 0.01):
    #line search pelo metodo do passo constante

    #define o sentido correto de busca
    if (f(P0 - eps*(direcao/np.linalg.norm(direcao))) > f(P0 + eps*(direcao/np.linalg.norm(direcao)))):
        sentido_busca = direcao.copy()
        flag = 0
    else:
        sentido_busca = -direcao.copy()
        flag = 1

    P = P0.copy()
    P_next = P + step*sentido_busca
    alpha = 0

    while (f(P) > f(P_next)):
        alpha = alpha + step
        P = P0 + alpha*sentido_busca
        P_next = P0 + (alpha+step)*sentido_busca
        if (f(P - eps*(sentido_busca/np.linalg.norm(sentido_busca))) < f(P)):
            alpha = alpha - step
            break

    intervalo = np.array([alpha, alpha + step])

    if(flag == 1):
        intervalo = -intervalo

    #retorna o intervalo de busca = [alpha min, alpha min + step]
    return intervalo
```

2.1.2 Seção Áurea

Implementando um método que recebe como parâmetros de entrada um intervalo de busca ($\overrightarrow{interv} = [\alpha^L, \alpha^U]$), um vetor direção de busca (\vec{dir}), um ponto inicial (\vec{P}_1), a função f ($f(\vec{P})$) e um parâmetro opcional para a tolerância de convergência com valor default de 10^{-5} .

Resumidamente, o método utiliza a razão áurea ($R_a = \frac{\sqrt{5}-1}{2}$) para comparar os valores de $f(\vec{P}_1 + \alpha_E \vec{dir})$ e $f(\vec{P}_1 + \alpha_D \vec{dir})$, onde $\alpha_E = \alpha^L + (1 - R_a)\beta$, $\alpha_D = \alpha^L + R_a\beta$ e $\beta = \alpha^U - \alpha^L$, para determinar em qual trecho, $[\alpha^L, \alpha_D]$ ou $[\alpha_E, \alpha^U]$, o ponto mínimo se encontra. Enquanto a convergência não é alcançada, os valores de $\alpha^L, \alpha^U, \alpha_E$ e α_D vão sendo recalculados e atualizados.

Quando o comprimento do trecho a ser avaliado é inferior à tolerância, o método finaliza e retorna o seguinte valor:

- α_{min} , tal que $\alpha_{min} = \frac{\alpha^L + \alpha^U}{2}$ e α^L, α^U são os extremos do intervalo do último passo do algoritmo, quando a convergência foi obtida

Importante destacar que o algoritmo identifica o sentido de busca e os sinal correto do valor de alpha mínimo através do valor do intervalo passado como parâmetro.

```
def secao_aurea(intervalo, direcao, P0, f, tol=0.00001):
    #line search pelo metodo da secao aurea

    #verifica o sentido da busca
    if(intervalo[1] < 0):
```

```

        intervalo = -intervalo
        sentido_busca = -direcao.copy()
        flag = 1
    else:
        sentido_busca = direcao.copy()
        flag = 0

    #atribui os limites superior e inferior da busca a variaveis internas do metodo
    alpha_upper = intervalo[1]
    alpha_lower = intervalo[0]
    beta = alpha_upper - alpha_lower

    #razao aurea
    Ra = (np.sqrt(5)-1)/2

    # define os pontos de analise de f com base na razao aurea
    alpha_e = alpha_lower + (1-Ra)*beta
    alpha_d = alpha_lower + Ra*beta

    #primeira iteracao avalia f nos 2 pontos selecionados pela razao aurea
    f1 = f(P0 + alpha_e*sentido_busca)
    f2 = f(P0 + alpha_d*sentido_busca)

    #loop enquanto a convergencia nao for obtida
    while (beta > tol):
        if (f1 > f2):
            #caso positivo, define novo intervalo variando de alpha_e ate alpha_upper
            # e aproveita os valores anteriores de alpha_d e f2 como novos alpha_e e f1
            alpha_lower = alpha_e
            f1 = f2
            alpha_e = alpha_d

            #calcula novo alpha_d e f2=f(alpha_d)
            beta = alpha_upper - alpha_lower
            #alpha_e = alpha_lower + (1-Ra)*beta
            alpha_d = alpha_lower + Ra*beta
            f2 = f(P0 + alpha_d*sentido_busca)
        else:
            #caso negativo, define novo intervalo variando de alpha_lower ate alpha_d
            # e aproveita os valores anteriores de alpha_e e f1 como novos alpha_d e f2
            alpha_upper = alpha_d
            f2 = f1
            alpha_d = alpha_e

            #calcula novo alpha_e e f1=f(alpha_e)
            beta = alpha_upper - alpha_lower
            alpha_e = alpha_lower + (1-Ra)*beta
            #alpha_d = alpha_lower + Ra*beta
            f1 = f(P0 + alpha_e*sentido_busca)

    # calcula Pmin e alpha min apos convergencia
    alpha_med = (alpha_lower + alpha_upper)/2
    alpha_min = alpha_med

    if (flag == 1):
        alpha_min = -alpha_min

    return alpha_min

```

2.2 Métodos OSR

Os algoritmos dos métodos Univariante, Powell, Steepest Descent, Fletcher-Reeves, BFGS e Newton-Raphson foram implementados em um arquivo denominado `osr_methods.py`. O seguinte pacote é necessário nesse arquivo:

```
import numpy as np
```

2.2.1 Univariante

O método univariante alterna entre as direções canônicas. A primeira vez que ele é chamado, retorna a primeira direção canônica. Na segunda vez, a segunda direção canônica, e assim por diante. Quando todas as direções canônicas são utilizadas, reinicia-se pela primeira direção.

A implementação considerou como parâmetros de entrada o número de dimensões e o passo em que a otimização se encontra. Ambos valores são facilmente calculados no código principal que irá chamar os métodos

de OSR. O número de dimensões é extraído do ponto inicial e o passo é um valor controlado durante o processo de convergência que irá ser implementado no código principal.

Toda vez que o método é chamado, inicializa-se um vetor com n zeros, sendo n o número de dimensões. O índice do vetor cujo valor será alterado para 1 é calculado através de manipulações em cima do resto da divisão do npumero do passo pelo número de dimensões.

```
def univariante(passo, dimens):
    indice = passo%dimens - 1
    if (indice == -1) :
        indice = dimens - 1
    ek = np.zeros(dimens)
    ek[indice] = 1

    return ek
```

2.2.2 Powell

O método de Powell consiste de ciclos de $n + 1$ passos, onde n é o número de dimensões. No primeiro ciclo e toda vez que o número do ciclo for múltiplo de $n + 2$, o conjunto de n direções é inicializado com as direções canônicas. A direção $n + 1$ de todo ciclo será o igual a $\vec{P}_n - \vec{P}_0$, onde \vec{P}_n é o ponto encontrado na otimização até o passo n do ciclo atual e \vec{P}_0 é o ponto inicial do primeiro passo do ciclo atual, que por sua vez é igual ao ponto final do ciclo anterior. Outra característica do método de Powell é que uma direção de um passo genérico k de um determinado ciclo será a direção do passo $k - 1$ do ciclo seguinte. Consegue

A implementação considerou como parâmetros de entrada o ponto mínimo encontrado no passo anterior (\vec{P}), o ponto inicial do ciclo atual (\vec{P}_1), um array com o conjunto de direções, um número representando o passo global da convergência no código principal, um número representado o ciclo do método e o número de dimensões.

Na primeira chamada do método, ou seja, para o primeiro passo do primeiro ciclo, o código principal envia o conjunto de direções canônicas e número de ciclo igual a zero. Com manipulações com o resto da divisão do número do passo pelo número de dimensões, o algoritmo consegue identificar qual direção utilizar do conjunto de direções enviado como parâmetro e atualizar o mesmo para o ciclo seguinte. O algoritmo também precisa atualizar o número do ciclo a cada $n + 1$ passos e voltar para as direções canônicas a cada $n + 2$ ciclos.

A cada achamada do método, são retornadas a direção para o passo atual, o conjunto de direções atualizado, o ponto inicial do ciclo atual e o número de ciclos atual. Esses valores precisam ser recebidos no código principal e utilizados para a chamada do método no passo seguinte.

```
def powell(P, P1, direcoes, passos, ciclos, dimens):
    indice = passos%(dimens + 1) - 1
    if (indice == -1):
        dir = P - P1
        direcoes[dimens - 1] = dir
    elif (indice == 0):
        ciclos = ciclos + 1
        if (ciclos%(dimens+2) == 0):
            direcoes = np.eye(dimens, dtype=float)
        P1 = P.copy()
        dir = direcoes[indice].copy()
    else:
        dir = direcoes[indice].copy()
        direcoes[indice-1] = dir

    return dir, direcoes, P1, ciclos
```

2.2.3 Steepest Descent

O método Steepest Descent recebe um ponto \vec{P} e a função a ser minimizada e retorna como direção o vetor com sentido oposto ao gradiente da função no ponto \vec{P} .

A implementação considerou como parâmetro de entrada apenas o vetor gradiente da função no ponto \vec{P} , uma vez que o código principal já faz o cálculo dessa variável para verificar convergência e a mesma já está disponível no momento de chamada do método.

```
def steepestDescent(grad):
    return -grad
```

2.2.4 Fletcher-Reeves

O método Fletcher-Reeves utiliza $-\vec{\nabla}f$ no ponto \vec{P}_0 como primeira direção. A partir da segunda direção ele utiliza uma correção β que é a razão ao quadrado entre o gradiente da função no ponto encontrado no passo anterior e o gradiente da função no ponto inicial do passo anterior. Ou seja, $\beta^k = \frac{(\|\vec{\nabla}f(\vec{P}^{k+1})\|)^2}{(\|\vec{\nabla}f(\vec{P}^k)\|)^2}$. A correção é então utilizada em conjunto com a direção do último passo e o gradiente da função no ponto obtido no último passo. Dessa forma, $\vec{d}^{k+1} = -\vec{\nabla}f(\vec{P}^{k+1}) + \beta^k \vec{d}^k$

A implementação considerou como parâmetros de entrada a direção utilizada no passo anterior, o gradiente da função no ponto obtido no passo anterior, o gradiente da função no ponto inicial do passo anterior e o número do passo. Para o primeiro passo do método, o código principal envia o gradiente da função no ponto inicial. O método retorna a direção a ser utilizada e o gradiente a ser usado no passo seguinte como gradiente da função f no ponto inicial do passo anterior. O código principal, recebe esses valores e utiliza nas chamadas subsequentes.

```
def fletcherReeves(dir_last, grad, grad_last, passo):
    if passo == 1:
        grad_last = grad.copy()
        return -grad, grad_last
    else:
        beta = (np.linalg.norm(grad)/np.linalg.norm(grad_last))**2
        grad_last = grad.copy()
        return -grad + beta*dir_last, grad_last
```

2.2.5 BFGS

O método BFGS retorna $\vec{d}^k = -\bar{S}^k \vec{\nabla}f(\vec{P}^k)$ como direção, sendo $\bar{S}^0 = \bar{I}$.

$$\begin{cases} \delta_x^k = \mathbf{x}^{k+1} - \mathbf{x}^k \\ \delta_g^k = \nabla f(\mathbf{x}^{k+1}) - \nabla f(\mathbf{x}^k) \end{cases}$$

Figura 1: Variáveis Auxiliares. Fonte: Aula-3_MetodosOSR.pdf

$$\mathbf{S}^{k+1} = \mathbf{S}^k + \frac{\left[(\delta_x^k)^t \ \delta_g^k + (\delta_g^k)^t \ \mathbf{S}^k \ \delta_g^k \right] \ \delta_x^k \ (\delta_x^k)^t - \frac{\mathbf{S}^k \ \delta_g^k \ (\delta_x^k)^t + \delta_x^k \ (\mathbf{S}^k \ \delta_g^k)^t}{(\delta_x^k)^t \ \delta_g^k}}{\left[(\delta_x^k)^t \ \delta_g^k \right]^2}$$

Figura 2: Cálculo de S^{k+1} . Fonte: Aula-3_MetodosOSR.pdf

A implementação considerou como parâmetros de entrada os pontos inicial e final do passo anterior, os gradientes da função nos pontos inicial e final do passo anterior, a matriz \bar{S} calculada no último passo, o número do passo e o número de dimensões. O algoritmo retorna a direção atual a ser utilizada na busca, bem como os demais parâmetros atualizados necessários para a chamada subsequente do método, que precisarão ser lidos no código principal.

```
def bfgs(P, P_last, grad, grad_last, S_last, passo, dimens):
    if (passo == 1):
        dir = -S_last.dot(grad)
    else:
        delta_x_k = P - P_last
        delta_g_k = grad - grad_last

        #para o numpy, vetor 1-D linha e vetor coluna sao a mesma coisa (nao e necessario
        #transpor)

        #matrizes
        A = np.outer(delta_x_k, np.transpose(delta_x_k))
        B = S_last.dot(np.outer(delta_g_k, np.transpose(delta_x_k)))
        C = np.outer(delta_x_k, np.transpose(S_last.dot(delta_g_k)))

        #Escalares
        d = np.transpose(delta_x_k).dot(delta_g_k)
        e = np.transpose(delta_g_k).dot(S_last.dot(delta_g_k))

        S = S_last + (d + e)*A/(d**2) - (B + C)/d
        dir = -S.dot(grad)
```

```

S_last = S.copy()
P_last = P
grad_last = grad
return dir, P_last, grad_last, S_last

```

2.2.6 Newton-Raphson

O método Newton-raphson retorna $\vec{d}^k = -\bar{H}(\vec{P}^k)^{-1} \vec{\nabla} f(\vec{P}^k)$.

A implementação considerou como parâmetros de entrada o ponto inicial ou ponto obtido no passo anterior, o gradiente da função nesse ponto e o método que calcula a matriz Hessiana de f.

```

def newtonRaphson(P, grad_P, hessian_f):
    return -np.linalg.inv(hessian_f(P)).dot(grad_P)

```

2.3 Código Principal

Os seguintes pacotes foram utilizados na implementação do código principal, já inclusos os arquivos .py com as implementações dos métodos OSR e busca unidimensional.

```

import numpy as np
import osr_methods as osr
import line_search_methods as lsm
import numdifftools as nd
import matplotlib.pyplot as plt
from timeit import default_timer as timer

```

Criada uma seção com as variáveis responsáveis pelo controle numérico da minimização das funções.

```

# numero maximo de iteracoes
maxiter = 200

# tolerancia para convergencia do gradiente
tol_conv = 1E-5

# tolerancia para a busca unidirecional
tol_search = 1E-5

# delta alpha do passo constante
line_step = 1E-2

#epsilon da maquina
eps = 1E-8

```

Seção para escolha do método a ser utilizado na minimização.

```

# 1 - Univariante
# 2 - Powell
# 3 - Steepest Descent
# 4 - Newton Raphson
# 5 - Fletcher Reeves
# 6 - BFGS

metodo = 1

if (metodo == 1):
    n_met = 'Univariante'
elif (metodo == 2):
    n_met = 'Powell'
elif (metodo == 3):
    n_met = 'Steepest Descent'
elif (metodo == 4):
    n_met = 'Newton Raphson'
elif (metodo == 5):
    n_met = 'Fletcher-Reeves'
elif (metodo == 6):
    n_met = 'BFGS'

```

Seção reservada para definição da função f, do gradiente de f, da Hessiana de f e ponto inicial. Nesse momento deixei em branco as definições, mas na seção de cada questão com sua função específica as definições serão apresentadas.

```

def f(Xn):
    return ...

def grad_f(Xn):
    return ...

def hessian_f(Xn):
    return ...

P0 = ...

# numero da função - apenas para automatização do plot de contorno
#1 = Questão 1 letra a
#2 = Questão 1 letra b
#3 = Questão 2 letra a
#4 = Questão 2 letra b
func = ...

# nome da questão e letra - apenas para automatização do plot
if (func == 1):
    n_func = 'Q1.a'
elif (func == 2):
    n_func = 'Q1.b'
elif (func == 3):
    n_func = 'Q2.a'
elif (func == 4):
    n_func = 'Q2.b'

```

Seção para inicialização de variáveis auxiliares para chamada inicial dos métodos de otimização OSR.

```

passos = 0
dimens = P0.size
Pmin = P0.copy()
listPmin = []
listPmin.append(Pmin)
grad = grad_f(Pmin)
norm_grad = np.linalg.norm(grad)

if (metodo == 2):
    direcoes = np.eye(dimens, dtype=float)
    ciclos = 0
    P1 = P0.copy()
elif (metodo == 5):
    #o método recebe a direção anterior
    #inicializo a direção com um vetor de zeros mas que nunca é usado
    #uso apenas para enviar como parâmetro na primeira iteração do método, o qual atualiza o
    #valor de dir para a iteração seguinte
    dir = np.zeros((1, dimens))
    grad_last = grad.copy()
elif (metodo == 6):
    S_last = np.eye(dimens)
    grad_last = grad.copy()
    P_last = P0.copy()

```

Loop para chamada dos métodos e convergência da otimização.

```

start = timer()

while (norm_grad > tol_conv):
    if (passos == maxiter):
        print('Não convergiu')
        break
    passos = passos + 1
    if (metodo == 1):
        dir = osr.univariante(passos, dimens)
    elif (metodo == 2):
        dir, direcoes, P1, ciclos = osr.powell(Pmin, P1, direcoes, passos, ciclos, dimens)
    elif (metodo == 3):
        dir = osr.steepestDescent(grad)
    elif (metodo == 4):
        dir = osr.newtonRaphson(Pmin, grad, hessian_f)
    elif (metodo == 5):
        dir, grad_last = osr.fletcherReeves(dir, grad, grad_last, passos)
    elif (metodo == 6):
        dir, P_last, grad_last, S_last = osr.bfgs(Pmin, P_last, grad, grad_last, S_last,
                                                    passos, dimens)

```

```

intervalo = lsm.passo_cte(dir, Pmin, f, eps, line_step)
alpha = lsm.secao_aurea(intervalo, dir, Pmin, f, tol_search)
Pmin = Pmin + alpha*dir
listPmin.append(Pmin)
grad = grad_f(Pmin)
norm_grad = np.linalg.norm(grad)

end = timer()

tempoExec = end - start

```

Seção para geração automática do gráfico de configuração dos nós da questão 2 letra b e das curvas de nível para as demais questões.

```

if (func < 4):
    if (func == 1):
        x1 = np.linspace(-6, 3, 100)
        x2 = np.linspace(-4, 2.5, 100)
        X1, X2 = np.meshgrid(x1, x2)
        X3 = f([X1, X2])
        niveis = plt.contour(X1, X2, X3, [0, 1, 3, 8, 15, 25, 40], colors='black')
    elif (func == 2):
        x1 = np.linspace(-5, 25, 100)
        x2 = np.linspace(-10, 10, 100)
        X1, X2 = np.meshgrid(x1, x2)
        X3 = f([X1, X2])
        niveis = plt.contour(X1, X2, X3, [50, 100, 200, 500, 1000, 2000, 5000], colors='black')
    elif (func == 3):
        x1 = np.linspace(-3, 3, 100)
        x2 = np.linspace(-6, 15, 100)
        X1, X2 = np.meshgrid(x1, x2)
        X3 = f([X1, X2])
        niveis = plt.contour(X1, X2, X3, [-2000, -1500, -1000, -500, 500, 2000], colors='black')

    plt.clabel(niveis, inline=1, fontsize=10)
    x = []
    y = []
    for P in listPmin:
        x.append(P[0])
        y.append(P[1])
    plt.plot(x, y, color='g', linewidth='3')
    plt.xlabel('$x_1$', fontsize='16')
    plt.ylabel('$x_2$', fontsize='16')
    plt.grid(linestyle='--')
    titulo = n_func + ' ' + n_met
    plt.title(titulo, fontsize='16')
    file_name = n_func + '_' + n_met + '_P0=' + np.array2string(P0, precision = 2, separator=',') + '.pdf'
    plt.savefig(file_name, format="pdf")
    plt.show()
elif (func == 4):
    x = []
    y = []
    n = int(dimens/2)
    m = n + 1
    Li = np.zeros(m, dtype=float) # comprimentos iniciais das molas
    Li = Li + 60/m
    print(Li)
    x.append(0)
    y.append(0)
    comp = 0
    for k in np.arange(n):
        comp = comp + Li[k]
        x.append(comp + Pmin[2*k])
        y.append(Pmin[2*k + 1])
    x.append(60)
    y.append(0)

    fig, ax = plt.subplots()
    ax.tick_params(top=True, labeltop=True, bottom=False, labelbottom=False)
    ax.xaxis.set_label_position('top')
    ax.spines['left'].set_position(('data', 0))
    ax.spines['top'].set_position(('data', 0))

    plt.plot(x, y, marker = 'o', color='g', linewidth='3')

```

```

plt.ylim([0, 8])
plt.xlim([0, 60])
plt.xlabel('x', fontsize='16')
plt.ylabel('y', fontsize='16')
plt.grid()
plt.gca().invert_yaxis()
plt.show()

```

3 Teste da Implementação

A questão 1 solicita a busca dos pontos de mínimo de duas funções distintas como forma de testar a implementação dos métodos de OSR e busca unidirecional apresentados na seção anterior. Uma das funções é quadrática e a outra não. Para funções quadráticas, é esperado que o método de Powell atinja convergência em até $(n + 1)^2$ passos, o método de Fletcher-Reeves em até $n + 1$ passos, o método BFGS em até $n + 1$ passos e o método Newton-Raphson em n passos, sendo n o número de dimensões (ou variáveis) da função.

O seguinte controle numérico foi utilizado para a convergência das funções da questão 1, independente do ponto inicial:

- Número máximo de passos (ou iterações): 200
- Tolerância para convergência do gradiente: 10^{-5}
- Tolerância para convergência da busca unidirecional: 10^{-6}
- $\Delta\alpha$ do passo constante: 10^{-2}

3.1 Questão 1 (a)

$$f(x_1, x_2) = x_1^2 - 3x_1x_2 + 4x_2^2 + x_1 - x_2$$

$$\vec{\nabla} f(x_1, x_2) = \begin{bmatrix} 2x_1 - 3x_2 + 1 \\ -3x_1 + 8x_2 - 1 \end{bmatrix}$$

$$H(x_1, x_2) = \begin{bmatrix} 2 & -3 \\ -3 & 8 \end{bmatrix}$$

Definição da função, gradiente e Hessiana no código principal :

```

def f(Xn):
    return Xn[0]**2 - 3*Xn[0]*Xn[1] + 4*(Xn[1]**2) + Xn[0] - Xn[1]

def grad_f(Xn):
    return np.array([2*Xn[0] - 3*Xn[1] + 1, -3*Xn[0] + 8*Xn[1] - 1])

def hessian_f(Xn):
    return np.array([[2, -3],
                   [-3, 8]]), dtype=float)

func = 1

```

3.1.1 Ponto inicial: $x^0 = \{2, 2\}^t$

Definição do ponto inicial no código principal:

```
P0 = np.array([2, 2])
```

Principais resultados obtidos:

Método	# Passos	Tempo(s)	P_{min}
Univariante	46	0.03397	$\{-0.71427545, -0.14285347\}^t$
Powell	6	0.09025	$\{-0.71428536, -0.14285781\}^t$
Steepest Descent	31	0.03088	$\{-0.71427858, -0.14285438\}^t$
Fletcher-Reeves	3	0.00991	$\{-0.71428613, -0.14285732\}^t$
BFGS	2	0.00495	$\{-0.71428573, -0.14285724\}^t$
Newton-Raphson	1	0.00241	$\{-0.71428661, -0.14285785\}^t$

Tabela 1: Resumo dos resultados obtidos na questão 1a para $x^0 = \{2, 2\}^t$

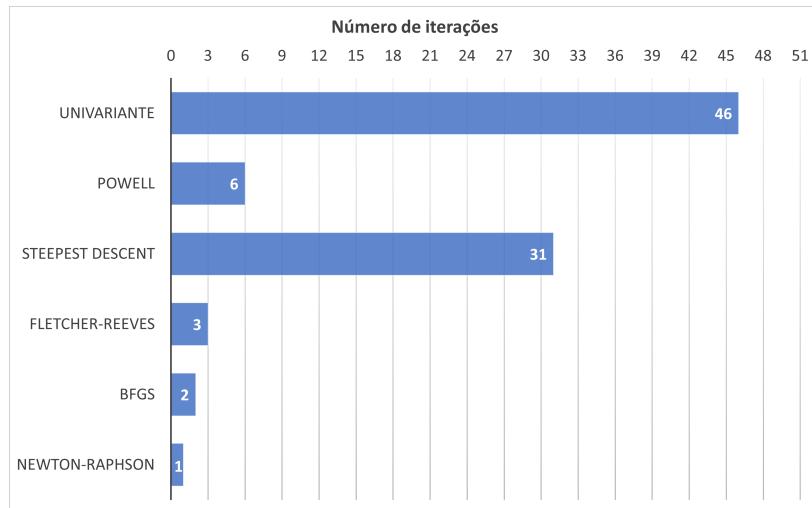


Figura 3: Número de passos por método OSR. Questão 1a e $x^0 = \{2, 2\}^t$

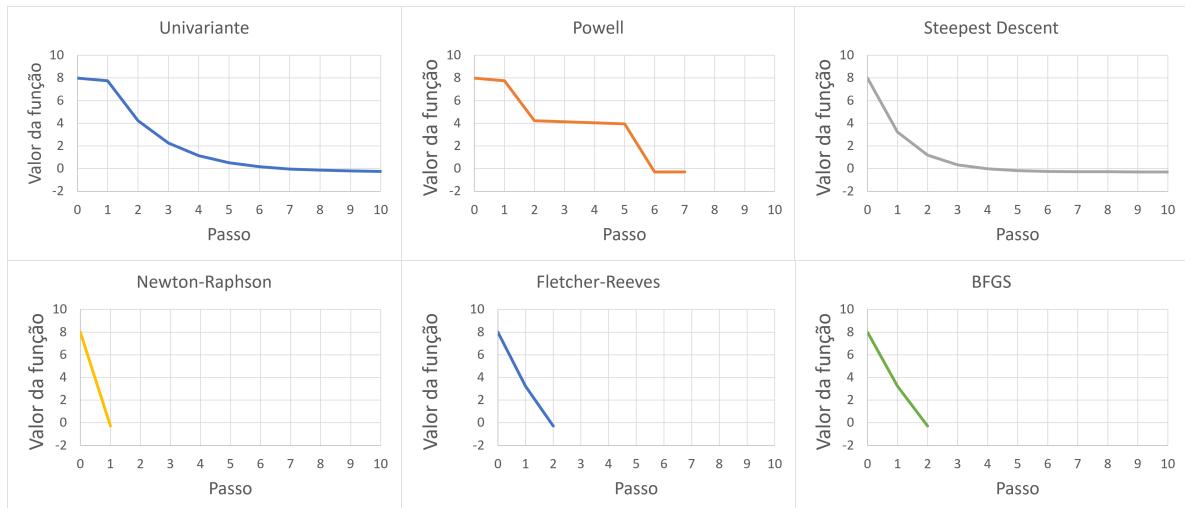


Figura 4: Gráficos de $f(x_1, x_2)$ versus passo da minimização, por método. Questão 1a e $x^0 = \{2, 2\}^t$

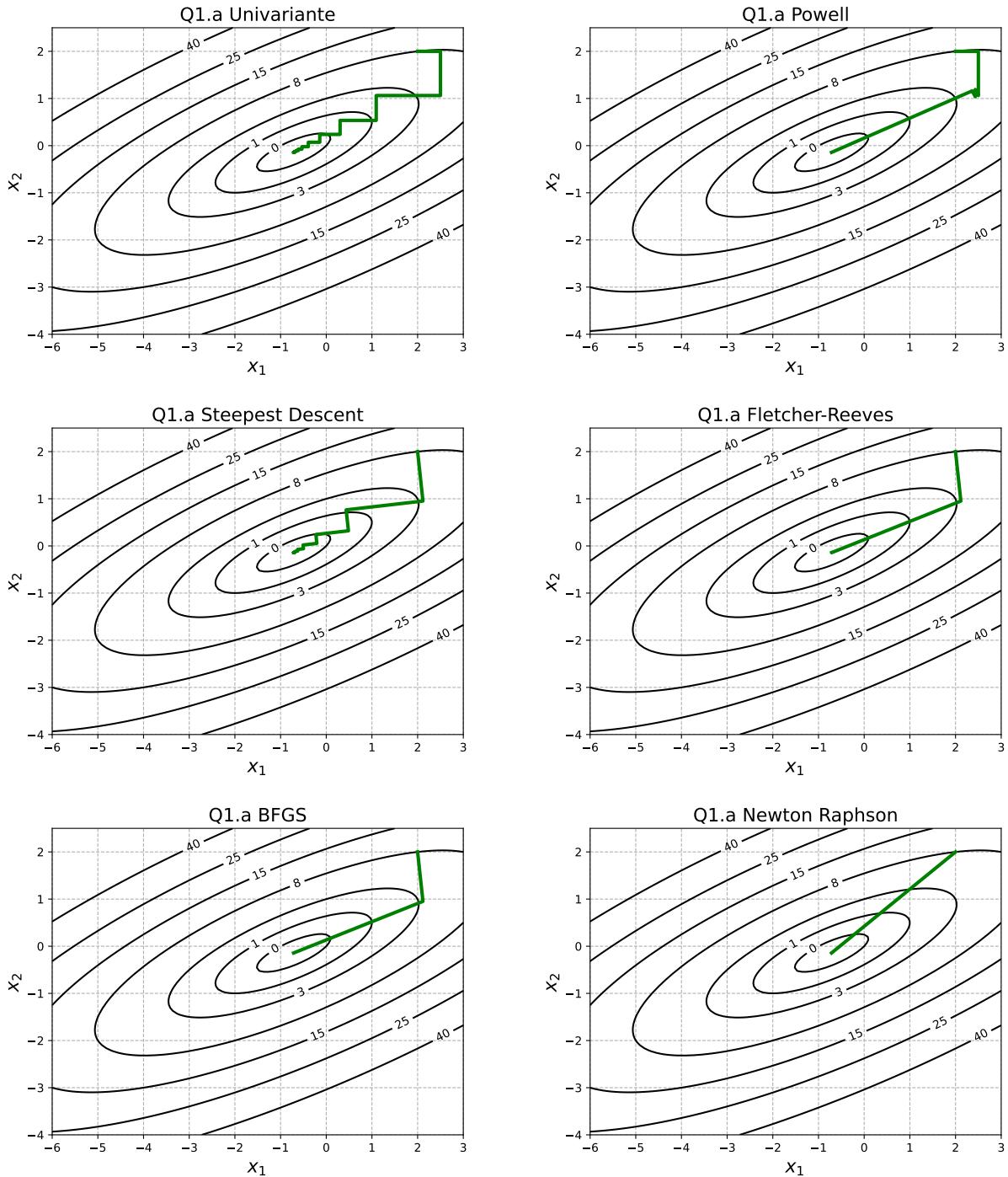


Figura 5: Curvas de nível e os pontos obtidos por método OSR. Questão 1a e $x^0 = \{2, 2\}^t$

3.1.2 Ponto inicial : $x^0 = \{-1, -3\}^t$

Definição do ponto inicial no código principal:

```
P0 = np.array([-1, -3])
```

Principais resultados obtidos:

Método	# Passos	Tempo(s)	P_{min}
Univariante	48	0.04435	$\{-0.7142935, -0.14286022\}^t$
Powell	6	0.02483	$\{-0.71428418, -0.14285757\}^t$
Steepest Descent	7	0.00690	$\{-0.71429411, -0.14286059\}^t$
Fletcher-Reeves	3	0.00467	$\{-0.71428581, -0.14285718\}^t$
BFGS	2	0.00332	$\{-0.71428583, -0.14285714\}^t$
Newton-Raphson	1	0.00219	$\{-0.71428562, -0.1428562\}^t$

Tabela 2: Resumo dos resultados obtidos na questão 1a para $x^0 = \{-1, -3\}^t$

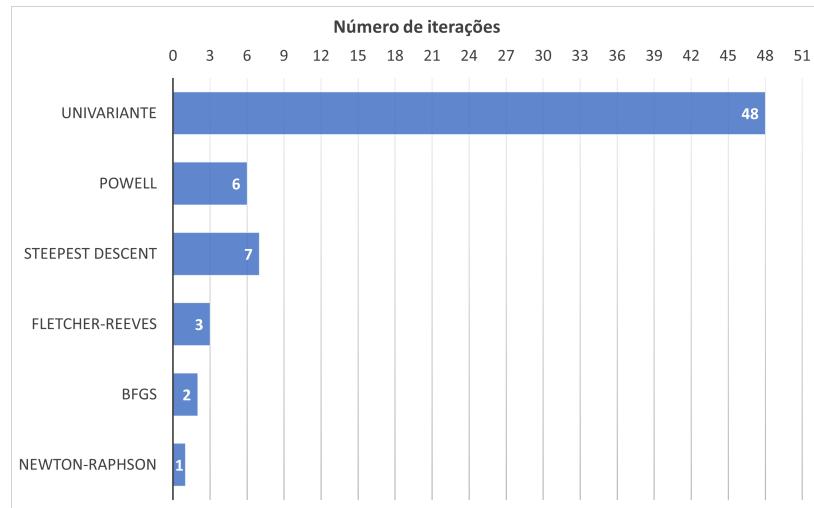


Figura 6: Número de passos por método OSR. Questão 1a e $x^0 = \{-1, -3\}^t$

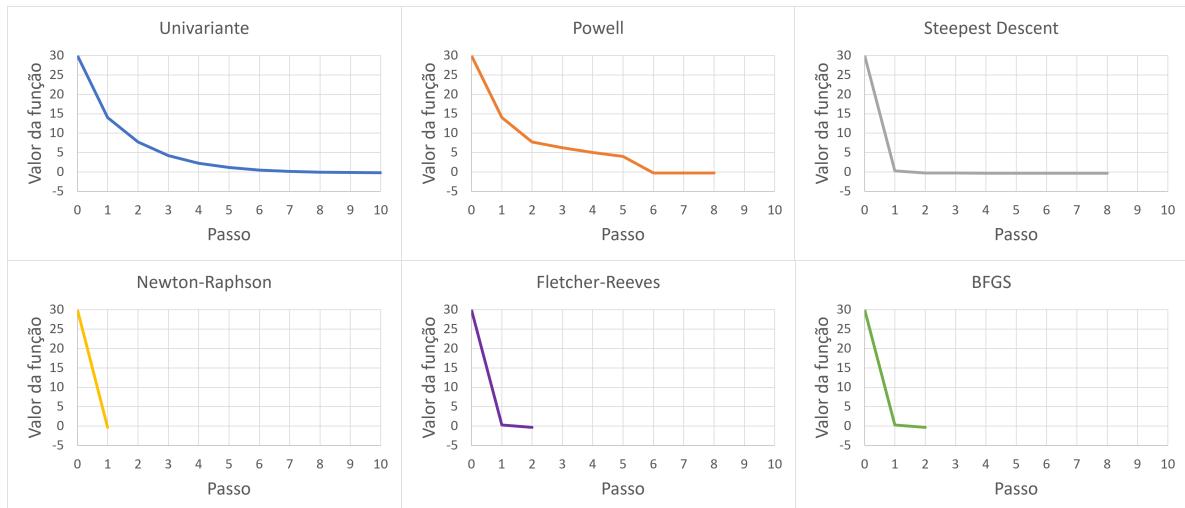


Figura 7: Gráficos de $f(x_1, x_2)$ versus passo da minimização, por método. Questão 1a e $x^0 = \{-1, -3\}^t$

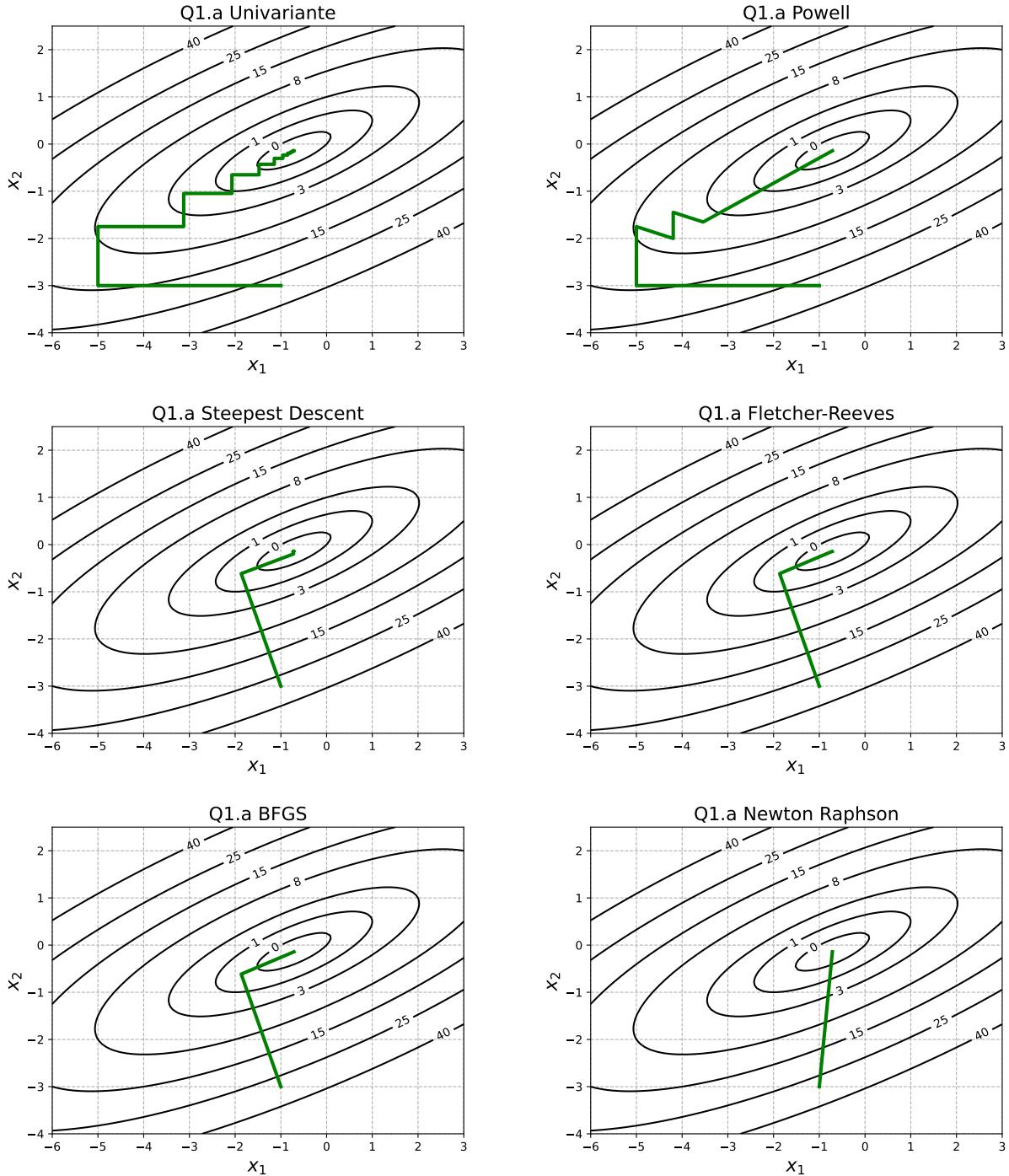


Figura 8: Curvas de nível e os pontos obtidos por método OSR. Questão 1a e $x^0 = \{-1, -3\}^t$

3.2 Questão 1 (b)

Considerar $a = 10$ e $b = 1$. Utilizei o site Wolfram Alpha para cálculo do gradiente e da Hessiana.

$$f(x_1, x_2) = (1 + a - bx_1 - bx_2)^2 + (b + x_1 + ax_2 - bx_1x_2)^2$$

$$\vec{\nabla} f(x_1, x_2) = \begin{bmatrix} 2(-a(bx_2^2 + b - x_2) + b^2x_1(x_2^2 + 1) - 2bx_1x_2 + x_1) \\ -2b(2ax_1x_2 + x_1^2 + 1) + 2a(ax_2 + x_1) + 2b^2(x_1^2 + 1)x_2 \end{bmatrix}$$

$$\begin{aligned} H_{1x1}(x_1, x_2) &= 2b^2 + 2(1 - bx_2)^2 \\ H_{1x2}(x_1, x_2) &= -2b(ax_2 + b(-x_1)x_2 + b + x_1) + 2(1 - bx_2)(a - bx_1) + 2b^2 \\ H_{2x1}(x_1, x_2) &= -2b(ax_2 + b(-x_1)x_2 + b + x_1) + 2(1 - bx_2)(a - bx_1) + 2b^2 \end{aligned}$$

$$H_{2 \times 2}(x_1, x_2) = 2(a - bx_1)^2 + 2b^2$$

Definição da função, gradiente e Hessiana no código principal :

```

def f(Xn):
    a = 10
    b = 1
    return (1 + a - b*Xn[0] - b*Xn[1])**2 + (b + Xn[0] + a*Xn[1] - b*Xn[0]*Xn[1])**2

def grad_f(Xn):
    a = 10
    b = 1
    return np.array([2*(-a*(b*(Xn[1]**2) + b - Xn[1]) + (b**2)*Xn[0]*(Xn[1]**2 + 1) - 2*b*Xn[0]*Xn[1] + Xn[0]),
                    -2*b*(2*a*Xn[0]*Xn[1] + Xn[0]**2 + 1) + 2*a*(a*Xn[1] + Xn[0]) + 2*(b**2)*(Xn[0]**2 + 1)*Xn[1]])

def hessian_f(Xn):
    a = 10
    b = 1
    hessian = np.zeros((2,2))
    hessian[0, 0] = 2*(b**2) + 2*((1 - b*Xn[1])**2)
    hessian[0, 1] = -2*b*(a*Xn[1] + b*(-Xn[0]*Xn[1]) + b + Xn[0]) + 2*(1-b*Xn[1])*(a - b*Xn[0]) + 2*(b**2)
    hessian[1, 0] = -2*b*(a*Xn[1] + b*(-Xn[0]*Xn[1]) + b + Xn[0]) + 2*(1-b*Xn[1])*(a-b*Xn[0]) + 2*(b**2)
    hessian[1, 1] = 2*((a-b*Xn[0])**2) + 2*(b**2)
    return hessian

func = 2

```

3.2.1 Ponto inicial : $x^0 = \{10, 2\}^t$

Definição do ponto inicial no código principal:

```
P0 = np.array([10, 2])
```

Método	# Passos	Tempo(s)	P_{min}
Univariante	64	0.03673	$\{13.00000142, 3.99999883\}^t$
Powell	15	0.02147	$\{13.00000057, 3.99999962\}^t$
Steepest Descent	55	0.00927	$\{13.00000099, 3.99999874\}^t$
Fletcher-Reeves	71	0.01208	$\{12.99999937, 4.00000089\}^t$
BFGS	9	0.01836	$\{13.00000042, 3.99999969\}^t$
Newton-Raphson	1	0.00272	$\{10, 0.99999967\}^t$

Tabela 3: Resumo dos resultados obtidos na questão 1b para $x^0 = \{10, 2\}^t$

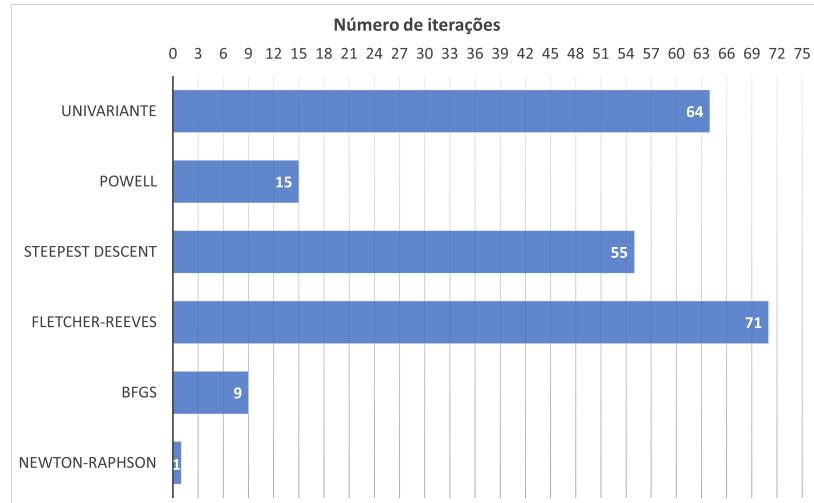


Figura 9: Número de passos por método OSR. Questão 1b e $x^0 = \{10, 2\}^t$

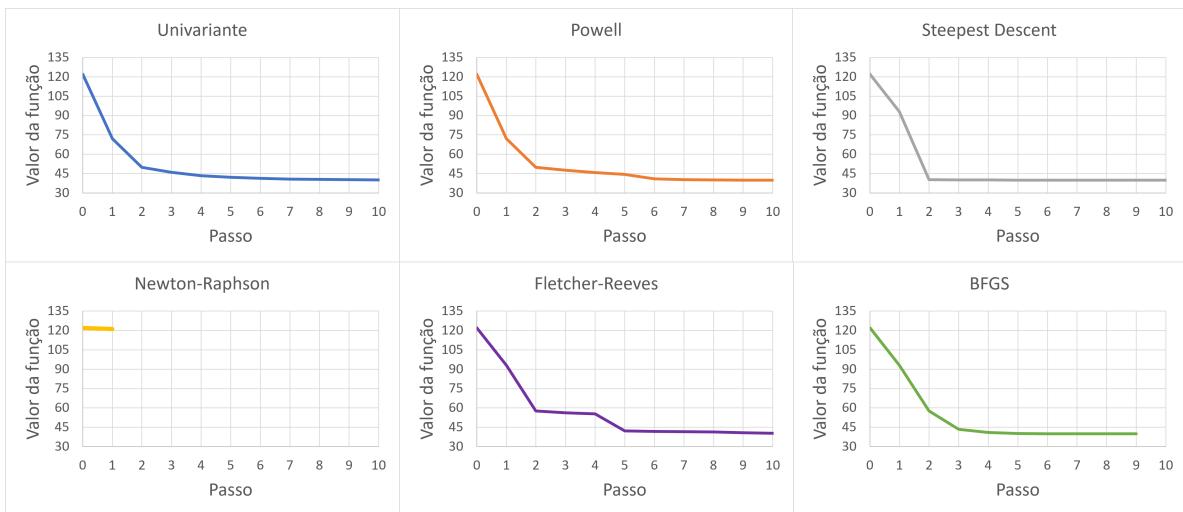


Figura 10: Gráficos de $f(x_1, x_2)$ versus passo da minimização, por método. Questão 1b e $x^0 = \{10, 2\}^t$

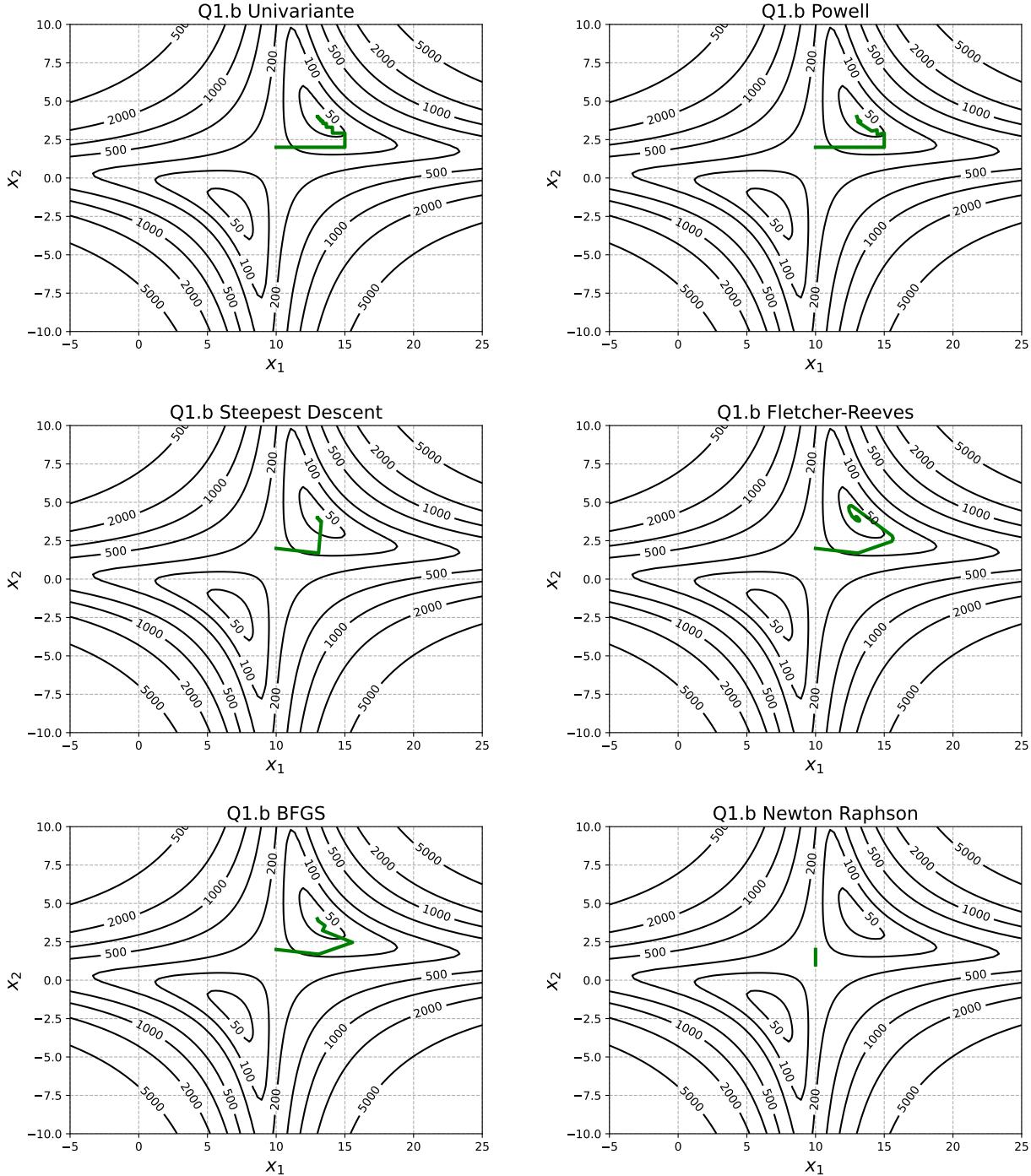


Figura 11: Curvas de nível e os pontos obtidos por método OSR. Questão 1b e $x^0 = \{10, 2\}^t$

Um resultado que chama a atenção é o método Fletcher-Reeves ter levado 71 passos para convergir. Alterar a tolerância de convergência global para 10^{-4} e a tolerância da seção áurea, na busca unidirecional, para 10^{-8} levou a uma redução modesta para 60 iterações. Analisando as curvas de nível e o "caminho" de otimização gerado pelo método, percebe-se que ocorre um movimento em espiral em torno do ponto mínimo. Ou seja, para essa função e ponto inicial, as direções geradas por Fletcher-Reeves levam a um elevado número de passos.

O método Newton-Raphson também se destaca por ter sido o único dentre os 6 métodos que encontrou um ponto de mínimo diferente. Enquanto os outros métodos levam a um mínimo à direita do ponto inicial, Newton-Raphson leva a um ponto de cela abaixo do ponto inicial. Nenhum dos métodos levou ao ponto de mínimo à esquerda do ponto incial.

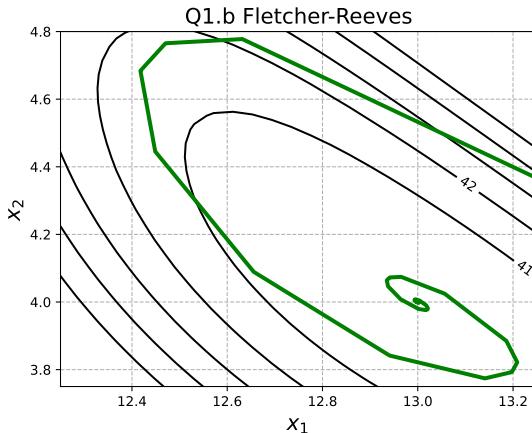


Figura 12: Movimento em espiral do método Fletcher-Reeves. Questão 1b e $x^0 = \{10, 2\}^t$

3.2.2 Ponto inicial : $x^0 = \{-2, -3\}^t$

Definição do ponto inicial no código principal:

```
P0 = np.array([-2, -3])
```

Método	# Passos	Tempo(s)	P_{min}
Univariante	61	0.04288	$\{7.00000124, -2.00000132\}^t$
Powell	15	0.13350	$\{7.0000002, -1.99999995\}^t$
Steepest Descent	45	0.01018	$\{7.000001, -2.0000009\}^t$
Fletcher-Reeves	21	0.00642	$\{7.0000007, -2.00000017\}^t$
BFGS	8	0.04286	$\{7.00000021, -2.00000023\}^t$
Newton-Raphson	6	0.01332	$\{7.00000001, -2.00000001\}^t$

Tabela 4: Resumo dos resultados obtidos na questão 1b para $x^0 = \{-2, -3\}^t$

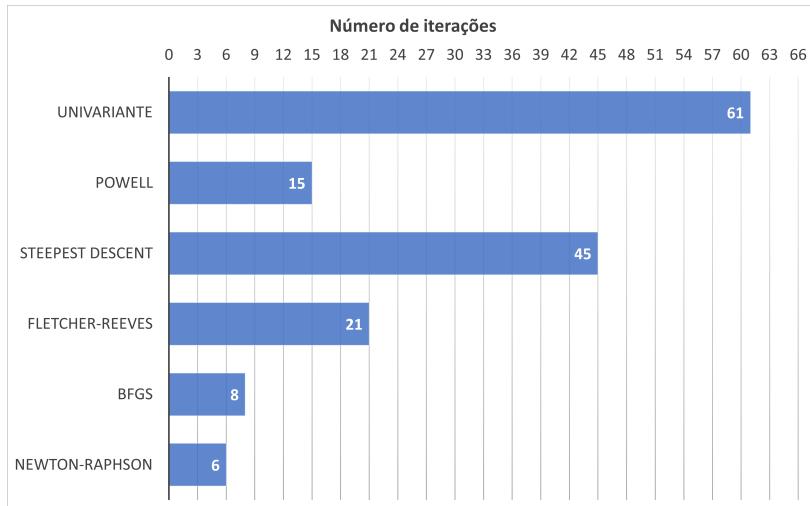


Figura 13: Número de passos por método OSR. Questão 1b e $x^0 = \{-2, -3\}^t$

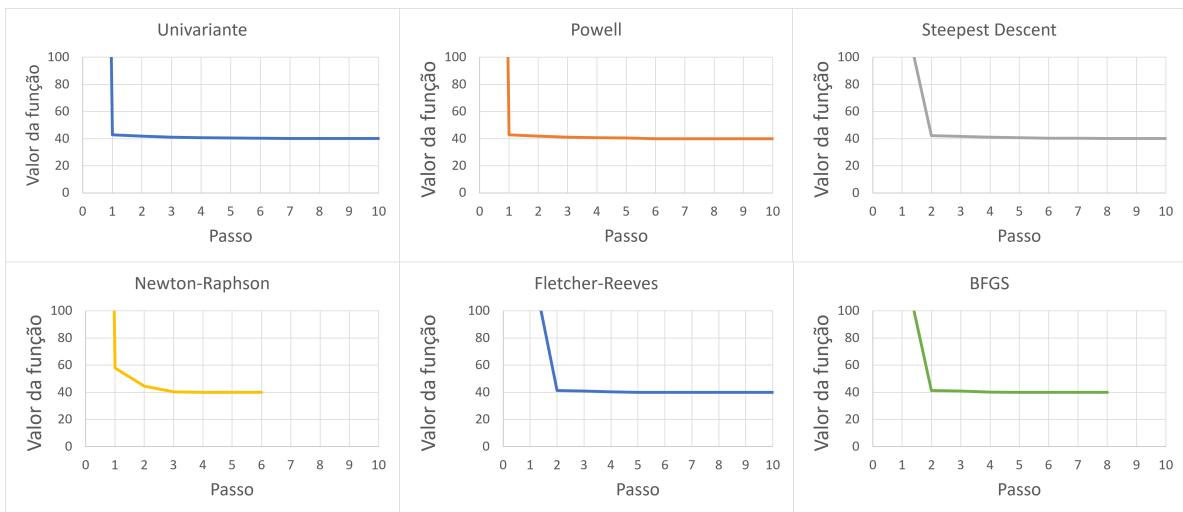


Figura 14: Gráficos de $f(x_1, x_2)$ versus passo da minimização, por método. Questão 1b e $x^0 = \{-2, -3\}^t$

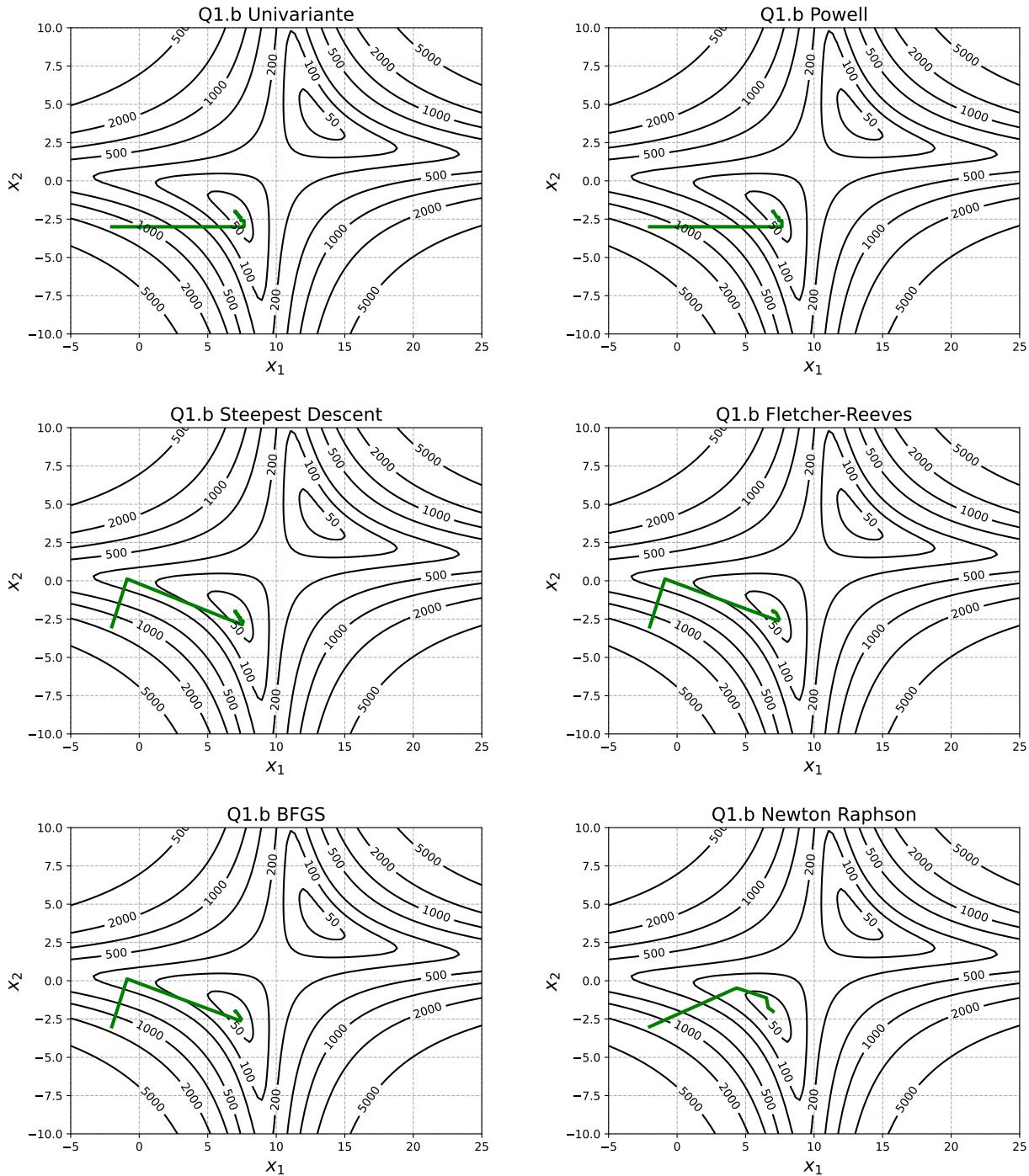


Figura 15: Curvas de nível e os pontos obtidos por método OSR. Questão 1b e $x^0 = \{-2, -3\}^t$

Todos os métodos convergiram para o ponto de mínimo mais próximo do ponto inicial.

4 Aplicação da Implementação

4.1 Questão 2 (a)

4.1.1 Enunciado

Determinar os deslocamentos (u_A, v_A) do ponto A , que minimizam a Energia Potencial Total (Π) do sistema de molas indicado na figura abaixo. Adotar o ponto inicial: $z^0 = \{0.01, -0.10\}^t$

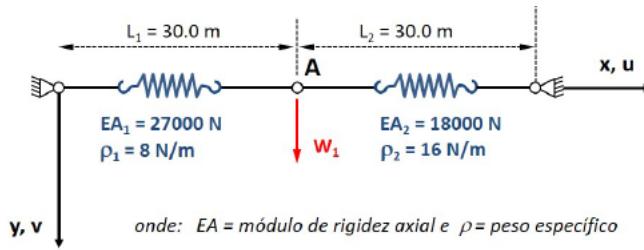


Figura 16: Sistema 2 molas. Fonte: Questão 2a do Trabalho 1.

4.1.2 Formulação

A seguinte formulação foi apresentada em sala de aula :

$$A = (x_A, y_A) = \text{Posição inicial do ponto A}$$

$$A' = (x_A + u_A, y_A + v_A) = \text{Posição final do ponto A}$$

$$\Pi = U - V = \text{Energia Potencial total}$$

$$U = U_{mola1} + U_{mola2} = \text{energia interna de deformação}$$

$$V = \text{trabalho das forças externas}$$

$$U_i = \frac{1}{2} K_i \Delta L_i^2$$

$$L'_1 = \sqrt{(L_1 + u_A)^2 + v_A^2}, \quad L'_2 = \sqrt{(L_2 - u_A)^2 + v_A^2} \quad \text{e} \quad W = \frac{1}{2}(\rho_1 L_1 + \rho_2 L_2)$$

$$V = Wv_A$$

$$\Pi = \frac{1}{2} \frac{EA_1}{L_1} (\sqrt{(L_1 + x_1)^2 + x_2^2} - L_1)^2 + \frac{1}{2} \frac{EA_2}{L_2} (\sqrt{(L_2 - x_1)^2 + x_2^2} - L_2)^2 - (\frac{\rho_1 L_1}{2} + \frac{\rho_2 L_2}{2}) x_2$$

Substituindo os valores do enunciado e usando o site Wolfram ALpha para cálculo do gradiente e Hessiana :

$$\Pi = 450(\sqrt{(30 + x_1)^2 + x_2^2} - 30)^2 + 300(\sqrt{(30 - x_1)^2 + x_2^2} - 30)^2 - 360x_2$$

$$\vec{\nabla} \Pi = \begin{bmatrix} \frac{900(x_1+30)(\sqrt{(x_1+30)^2+x_2^2}-30)}{\sqrt{(x_1+30)^2+x_2^2}} - \frac{600(30-x_1)(\sqrt{(x_1-30)^2+x_2^2}-30)}{\sqrt{(x_1-30)^2+x_2^2}} \\ 60(x_2(\frac{-450}{\sqrt{x_1^2+60x_1+x_2^2+900}} - \frac{300}{\sqrt{x_1^2-60x_1+x_2^2+900}} + 25) - 6) \end{bmatrix}$$

$$H_{1x1} = -\frac{600(30-x_1)^2(\sqrt{(30-x_1)^2+x_2^2}-30)}{((30-x_1)^2+x_2^2)^{3/2}} + \frac{600(30-x_1)^2}{(30-x_1)^2+x_2^2} + \frac{600(\sqrt{(30-x_1)^2+x_2^2}-30)}{\sqrt{(30-x_1)^2+x_2^2}} + \frac{900(\sqrt{(x_1+30)^2+x_2^2}-30)}{\sqrt{(x_1+30)^2+x_2^2}}$$

$$-\frac{900(x_1+30)^2(\sqrt{(x_1+30)^2+x_2^2}-30)}{((x_1+30)^2+x_2^2)^{3/2}} + \frac{900(x_1+30)^2}{(x_1+30)^2+x_2^2}$$

$$H_{1x2} = \frac{600(30-x_1)x_2(\sqrt{(30-x_1)^2+x_2^2}-30)}{((30-x_1)^2+x_2^2)^{3/2}} - \frac{900(x_1+30)x_2(\sqrt{(x_1+30)^2+x_2^2}-30)}{((x_1+30)^2+x_2^2)^{3/2}} - \frac{600(30-x_1)x_2}{(30-x_1)^2+x_2^2} + \frac{900(x_1+30)x_2}{(x_1+30)^2+x_2^2}$$

$$H_{2x1} = \frac{600(30-x_1)x_2(\sqrt{(30-x_1)^2+x_2^2}-30)}{((30-x_1)^2+x_2^2)^{3/2}} - \frac{900(x_1+30)x_2(\sqrt{(x_1+30)^2+x_2^2}-30)}{((x_1+30)^2+x_2^2)^{3/2}} - \frac{600(30-x_1)x_2}{(30-x_1)^2+x_2^2} + \frac{900(x_1+30)x_2}{(x_1+30)^2+x_2^2}$$

$$H_{2x2} = -\frac{600x_2^2(\sqrt{(30-x_1)^2+x_2^2}-30)}{((30-x_1)^2+x_2^2)^{3/2}} - \frac{900x_2^2(\sqrt{(x_1+30)^2+x_2^2}-30)}{((x_1+30)^2+x_2^2)^{3/2}} + \frac{600x_2^2}{(30-x_1)^2+x_2^2} + \frac{900x_2^2}{(x_1+30)^2+x_2^2} + \frac{600(\sqrt{(30-x_1)^2+x_2^2}-30)}{\sqrt{(30-x_1)^2+x_2^2}}$$

$$+\frac{900(\sqrt{(x_1+30)^2+x_2^2}-30)}{\sqrt{(x_1+30)^2+x_2^2}}$$

Definição da função, gradiente, Hessiana no código principal :

```

def f(Xn):
    return 450 * ((np.sqrt((30 + Xn[0])**2 + Xn[1]**2) - 30)**2) + 300 * ((np.sqrt((30 - Xn[0])**2 + Xn[1]**2) - 30)**2) - 360*Xn[1]

def grad_f(Xn):

```

```

    return np.array([(900*(Xn[0] + 30)*(np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - 30))/np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - (600*(30 - Xn[0]))*(np.sqrt((Xn[0] - 30)**2 + Xn[1]**2) - 30))/np.sqrt((Xn[0] - 30)**2 + Xn[1]**2),
    60*(Xn[1]*(-450/np.sqrt(Xn[0]**2 + 60*Xn[0] + Xn[1]**2 + 900) - 300/np.sqrt(Xn[0]**2 - 60*Xn[0] + Xn[1]**2 + 900) + 25) - 6)])
}

def hessian_f(Xn):
    hessian = np.zeros((2,2))
    hessian[0, 0] = -(600*(30 - Xn[0])**2*(np.sqrt((30 - Xn[0])**2 + Xn[1]**2) - 30))/((30 - Xn[0])**2 + Xn[1]**2)**(3/2) + \
    (600*(30 - Xn[0])**2)/((30 - Xn[0])**2 + Xn[1]**2) + \
    (600*(np.sqrt((30 - Xn[0])**2 + Xn[1]**2) - 30))/np.sqrt((30 - Xn[0])**2 + Xn[1]**2) + \
    (900*(np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - 30))/np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - \
    (900*(Xn[0] + 30)**2*(np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - 30))/((Xn[0] + 30)**2 + Xn[1]**2)**(3/2) + \
    (900*(Xn[0] + 30)**2)/((Xn[0] + 30)**2 + Xn[1]**2)

    hessian[0, 1] = (600*(30 - Xn[0])*Xn[1]*(np.sqrt((30 - Xn[0])**2 + Xn[1]**2) - 30))/((30 - Xn[0])**2 + Xn[1]**2)**(3/2) - \
    (900*(Xn[0] + 30)*Xn[1]*(np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - 30))/((Xn[0] + 30)**2 + Xn[1]**2)**(3/2) - \
    (600*(30 - Xn[0])*Xn[1])/((30 - Xn[0])**2 + Xn[1]**2) + \
    (900*(Xn[0] + 30)*Xn[1])/((Xn[0] + 30)**2 + Xn[1]**2)

    hessian[1, 0] = (600*(30 - Xn[0])*Xn[1]*(np.sqrt((30 - Xn[0])**2 + Xn[1]**2) - 30))/((30 - Xn[0])**2 + Xn[1]**2)**(3/2) - \
    (900*(Xn[0] + 30)*Xn[1]*(np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - 30))/((Xn[0] + 30)**2 + Xn[1]**2)**(3/2) - \
    (600*(30 - Xn[0])*Xn[1])/((30 - Xn[0])**2 + Xn[1]**2) + \
    (900*(Xn[0] + 30)*Xn[1])/((Xn[0] + 30)**2 + Xn[1]**2)

    hessian[1, 1] = -(600*Xn[1]**2*(np.sqrt((30 - Xn[0])**2 + Xn[1]**2) - 30))/((30 - Xn[0])**2 + Xn[1]**2)**(3/2) - \
    (900*Xn[1]**2*(np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - 30))/((Xn[0] + 30)**2 + Xn[1]**2)**(3/2) + \
    (600*Xn[1]**2)/((30 - Xn[0])**2 + Xn[1]**2) + \
    (900*Xn[1]**2)/((Xn[0] + 30)**2 + Xn[1]**2) + \
    (600*(np.sqrt((30 - Xn[0])**2 + Xn[1]**2) - 30))/np.sqrt((30 - Xn[0])**2 + Xn[1]**2) + \
    (900*(np.sqrt((Xn[0] + 30)**2 + Xn[1]**2) - 30))/np.sqrt((Xn[0] + 30)**2 + Xn[1]**2)

    return hessian

func = 3

```

4.1.3 Resultados

Definição do ponto inicial no código principal:

```
P0 = np.array([0.01, -0.1])
```

Utilizei o seguinte controle numérico para resolução da questão 2(a):

- Número máximo de passos (ou iterações): 200
- Tolerância para convergência do gradiente: Sensibilidade com 10^{-5} , 10^{-4} e 10^{-3}
- Tolerância para convergência da busca unidirecional: 10^{-6}
- $\Delta\alpha$ do passo constante: 10^{-2}

Principais resultados obtidos:

$\text{tol} = 10^{-5}$

Método	# Passos	Tempo(s)	P_{min}
Univariante	200	0.07850	$\{-0.20510911, 7.78899302\}^t$
Powell	200	91.21617	$\{-0.20510878, 7.78899254\}^t$
Steepest Descent	20	0.00304	$\{-0.20510889, 7.78899266\}^t$
Fletcher-Reeves	200	0.02659	$\{-0.20510878, 7.78899218\}^t$
BFGS	200	0.03755	$\{-0.20510893, 7.78899288\}^t$
Newton-Raphson	200	0.03705	$\{-0.2051089, 7.78899288\}^t$

Tabela 5: Resumo dos resultados obtidos na questão 2a para $x^0 = \{0.01, -0.1\}^t$ e $\text{tol} = 10^{-5}$

$\text{tol} = 10^{-4}$

Método	# Passos	Tempo(s)	P_{min}
Univariante	200	0.05557	$\{-0.20510911, 7.78899302\}^t$
Powell	9	92.83198	$\{-0.20510884, 7.78899251\}^t$
Steepest Descent	6	0.00090	$\{-0.20510891, 7.78899276\}^t$
Fletcher-Reeves	12	0.00123	$\{-0.20510891, 7.78899332\}^t$
BFGS	4	0.00324	$\{-0.2051089, 7.78899268\}^t$
Newton-Raphson	49	0.12737	$\{-0.20510894, 7.78899296\}^t$

Tabela 6: Resumo dos resultados obtidos na questão 2a para $x^0 = \{0.01, -0.1\}^t$ e $\text{tol} = 10^{-4}$

$\text{tol} = 10^{-3}$

Método	# Passos	Tempo(s)	P_{min}
Univariante	9	0.04030	$\{-0.20510877, 7.78899022\}^t$
Powell	8	91.89477	$\{-0.20510883, 7.78899446\}^t$
Steepest Descent	5	0.00073	$\{-0.20510863, 7.78899279\}^t$
Fletcher-Reeves	10	0.00255	$\{-0.2051088, 7.78899188\}^t$
BFGS	4	0.00500	$\{-0.20510894, 7.78899296\}^t$
Newton-Raphson	3	0.00425	$\{-0.20510893, 7.78899416\}^t$

Tabela 7: Resumo dos resultados obtidos na questão 2a para $x^0 = \{0.01, -0.1\}^t$ e $\text{tol} = 10^{-3}$

Pelas tabelas apresentadas acima, um aumento da tolerância da convergência do gradiente leva a uma redução drástica do número de iterações e, aparentemente, sem grandes perdas de precisão no valor de ponto mínimo obtido. Para a tolerância de 10^{-5} apenas o método Steepest Descent convergiu, levando um total de 20 passos, enquanto os demais métodos usaram o máximo especificado de 200 iterações. Para uma tolerância de 10^{-4} apenas o método univariante não convergiu em menos de 200 passos, enquanto que para uma tolerância de 10^{-3} todos os métodos convergiram em até 10 passos e com resultados satisfatórios.

Em termos de tempo de execução, o método de Powell é o que mais se destaca. Independente da tolerância, o mesmo leva por volta de 90 segundos para convergir. Após depuração do código e dos resultados ficou claro que o motivo disso ocorrer é que, para essa função e ponto inicial, o método de Powell, no sexto passo, segundo ciclo, gera uma direção com módulo muito pequeno e que que necessita de um α (aproximadamente 60000) muito grande para alcançar o mínimo. Isso faz com que o algoritmo leve o tempo apresentado. Essa talvez seja uma grande indicação de que trabalhar com direções normalizadas na busca unidirecional seja mais adequado, evitando esse tipo de problema que pode ocorrer com outros métodos também a depender da combinação dos parâmetros de entrada.

Para fins de apresentação dos demais resultados, irei utilizar como base o estudo feito com tolerância de 10^{-3} .

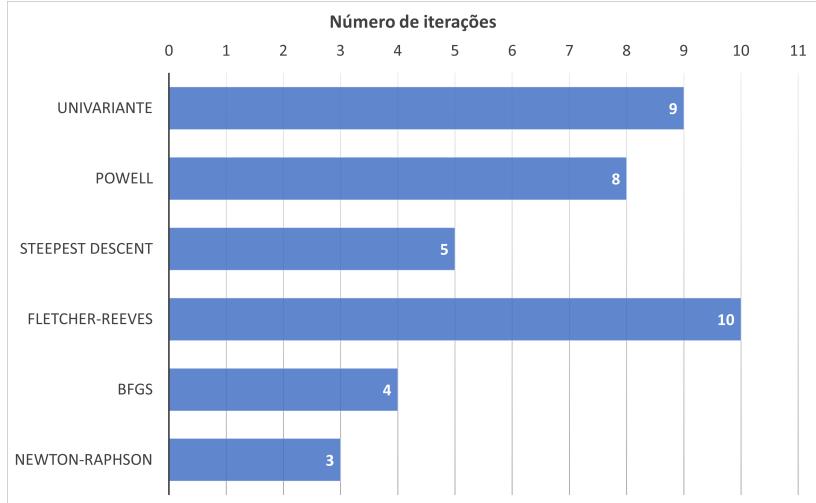


Figura 17: Número de passos por método OSR. Questão 2a e $x^0 = \{0.01, -0.1\}^t$

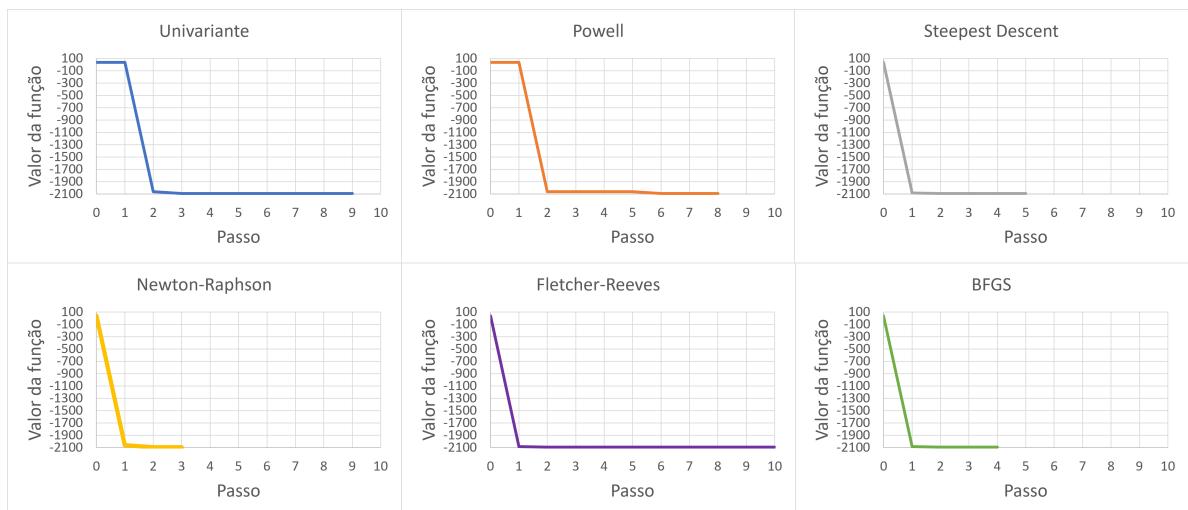


Figura 18: Gráficos de $f(x_1, x_2)$ versus passo da minimização, por método. Questão 2a com $x^0 = \{0.01, -0.1\}^t$ e $\text{tol} = 10^{-3}$

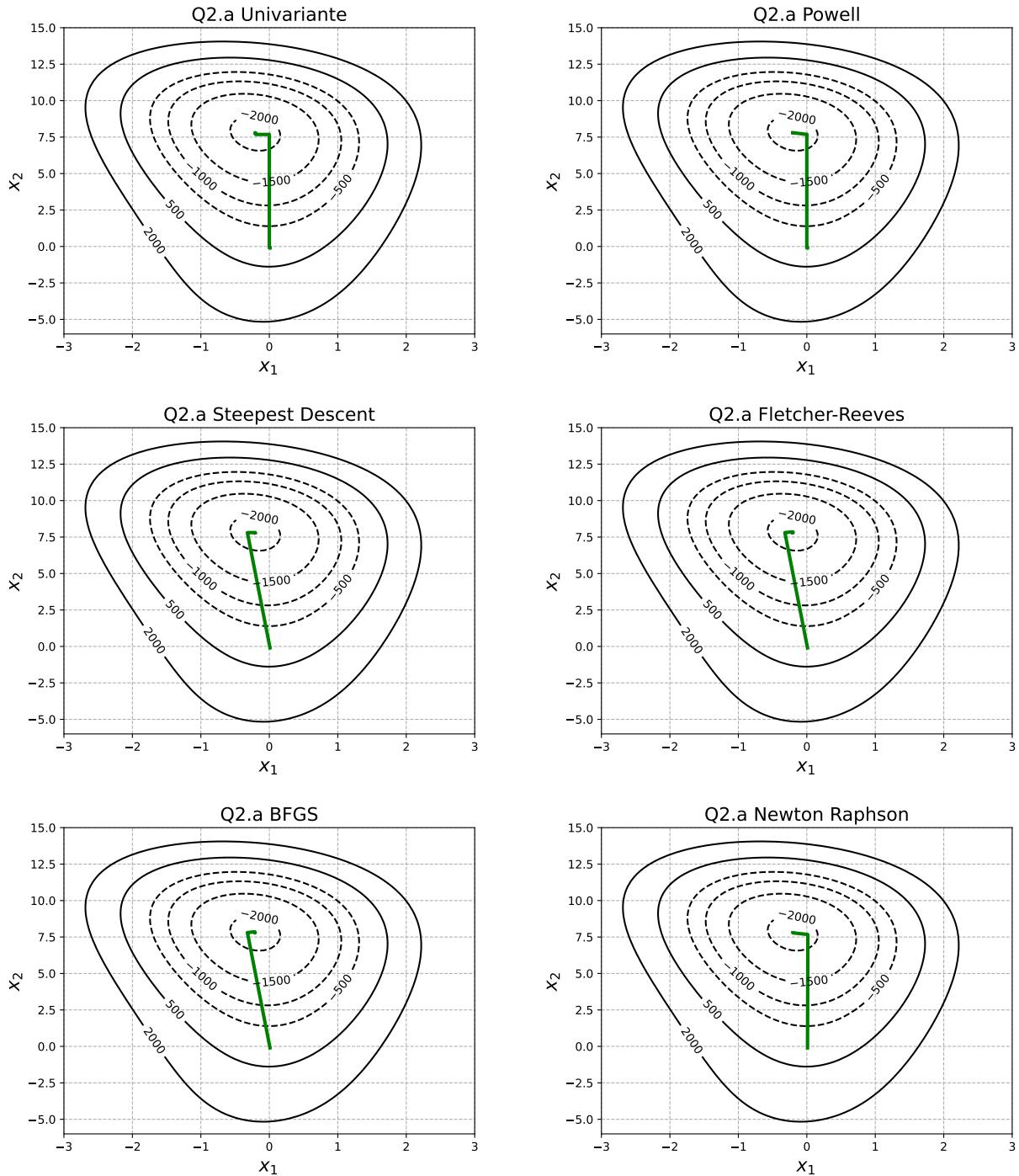


Figura 19: Curvas de nível e os pontos obtidos por método OSR. Questão 2a com $x^0 = \{0.01, -0.1\}^t$ e $\text{tol} = 10^{-3}$

Abaixo seguem as curvas de nível para um ponto inicial diferente do proposto no enunciado. $x^0 = \{-2, 10\}^t$

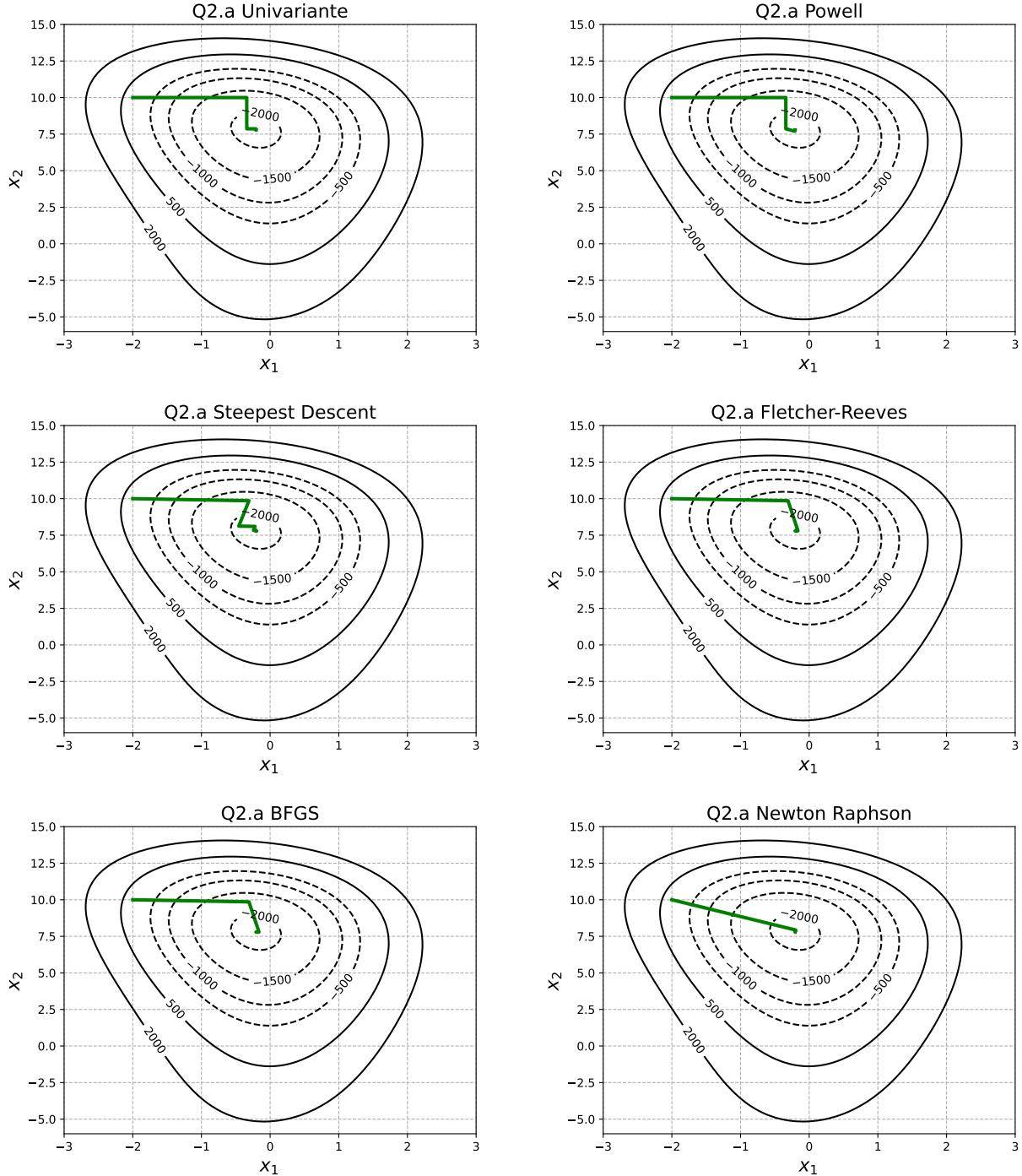


Figura 20: Curvas de nível e os pontos obtidos por método OSR. Questão 2a e $x^0 = \{-2, 10\}^t$ e $\text{tol} = 10^{-3}$

4.2 Questão 2 (b)

4.2.1 Enunciado

Desenvolver um estudo de convergência da solução do deslocamento do ponto A, do sistema de molas, para níveis crescentes de discretização do modelo (ou seja, considerando o número de molas $n = 2, 4, 6, \dots$). A rigidez de cada mola (k_i , $i = 1, \dots, n$) é obtida como a razão entre o módulo de rigidez axial do material e o seu comprimento. Os valores W_j (com $j = 1, \dots, n-1$) correspondem às cargas nodais equivalentes aos pesos das molas.

4.2.2 Formulação

Para a generalização do problema do sistema de molas, foi considerado que sempre existirá um número par de molas.

$$n_{\text{nodes}} = n_{\text{molas}} - 1$$

Como cada nó possui duas variáveis (Deslocamento horizontal (u_i) e Deslocamento vertical (v_i)), o número de dimensões será sempre $n_{dimens} = 2n_{nodes}$.

Usando a mesma ideia do equacionamento dos comprimentos finais de cada mola para o sistema com apenas 2 molas, e extrapolando para n_{dimens} , podemos escrever o seguinte :

Seja Li_m = comprimento inicial da mola m, com $m=1,2,\dots,n_{molas}$

Seja Lf_m = comprimento final da mola k, com $m=1,2,\dots,n_{molas}$

$$Lf_m = \sqrt{(Li_m + u_m - u_{m-1})^2 + (v_m - v_{m-1})^2}, \text{ com } u_0 = v_0 = u_{n_{molas}} = v_{n_{molas}} = 0$$

$$U_m = \frac{1}{2}K_m\Delta L_m^2, \text{ sendo } K_m = \frac{EA_m}{Li_m} \text{ e } \Delta L_m = Lf_m - Li_m = \text{Energia Interna de deformação}$$

$$W_n = \frac{1}{2}(\rho_n Li_n + \rho_{n+1} Li_{n+1}), \text{ com } n=1,2,\dots,n_{nodes}$$

$V_n = W_n v_n$ = Trabalho das forças externas

Finalmente :

$$\Pi = \sum_{m=1}^{n_{molas}} U_m - \sum_{n=1}^{n_{nodes}} V_n = \text{Energia Potencial Total}$$

Para o cálculo do $\vec{\nabla}\Pi$, o seguinte raciocínio foi adotado :

A variável u_n aparece apenas nos termos U_n e U_{n+1} , enquanto v_n aparece em U_n , U_{n+1} e V_n .

Dessa forma $\frac{\partial \Pi}{\partial u_n} = \frac{\partial U_n}{\partial u_n} + \frac{\partial U_{n+1}}{\partial u_n}$ e $\frac{\partial \Pi}{\partial v_n} = \frac{\partial U_n}{\partial v_n} + \frac{\partial U_{n+1}}{\partial v_n} - \frac{\partial V_n}{\partial v_n}$. Todos esses termos são possíveis de se calcular analiticamente usando as expressões apresentadas acima para Lf_m , U_m e V_n , e com isso temos uma expressão para cálculo do gradiente da função e que foi implementada no meu código.

O mesmo raciocínio poderia ser aplicado para cálculo da Hessiana da função, porém, como sua utilidade fica restrita ao método de Newton-Raphson, para fins desse trabalho usei um pacote pronto para cálculo diferencial numérico no Python (numdifftools) com resultados bem satisfatórios e coincidentes com todos os cálculos dos métodos analíticos de gradiente e Hessiana implementados para as funções da questão 1 e questão 2a.

Na implementação o parâmetro que controla o número de molas do sistema é o número de dimensões do ponto inicial. Ou seja, para um sistema 2 molas, 1 nó livre, é necessário fornecer x^0 com 2 dimensões ($x^0 = \{u_1, v_1\}^t$). Para um sistema 4 molas, 3 nós livres, é necessário informar $x^0 = \{u_1, v_1, u_2, v_2, u_3, v_3\}^t$, e assim por diante.

Definição da função, gradiente, Hessiana no código principal :

```
def f(Xn):
    dimens = Xn.size
    #numero de nos
    n = int(dimens/2)

    #numero de molas
    m = n + 1

    #Inicializacao dos vetores com as variaveis do problema
    Li = np.zeros(m, dtype=float) # comprimentos iniciais das molas
    EA = np.zeros(m, dtype=float)
    RHO = np.zeros(m, dtype=float)
    W = np.zeros(n, dtype=float) # peso em cada no

    #Atribuicao dos valores do problema
    #Cada mola mede inicialmente 60/n_molas
    #molas a esquerda possuem EA = 27000 e rho 8
    #molas a direita possuem EA = 18000 e rho 16
    Li = Li + 60/m

    EA[: int(m/2)] = 27000
    EA[int(m/2) : ] = 18000
    RHO[: int(m/2)] = 8
    RHO[int(m/2) : ] = 16

    #Calculo dos pesos atuando em cada no
    # W[j] = (1/2)*(RHO[j]*Li[j] + RHO[j+1]*Li[j+1])
    RHO_e = RHO[:m-1]
    RHO_d = RHO[1:m]
    Li_e = Li[:m-1]
    Li_d = Li[1:m]
    W = (1/2)*(RHO_e*Li_e + RHO_d*Li_d)

    Lf = np.zeros(m, dtype=float) # comprimentos finais das molas
```

```

U = np.zeros(m, dtype=float) # energia elastica das molas 0.01, -0.1
V = np.zeros(n, dtype=float) # trabalho em cada no (desloc vert)

#array com os deslocamentos horizontais do Xn
dx = Xn[0::2].copy()
#array com os deslocamentos verticais do Xn
dy = Xn[1::2].copy()

#Calculo dos comprimentos finais
# Lf[0] = np.sqrt( (Li[0] + dx[0]) **2 + dy[0]**2 )
# Lf[k] = np.sqrt(a**2 + b**2)
# Lf[m-1] = np.sqrt((Li[m-1] - dx[n-1]))**2 + dy[n-1]**2)
dx_d = np.zeros(m, dtype= float)
dx_d[1:] = dx.copy()
dx_e = np.zeros(m, dtype= float)
dx_e[:m-1] = dx.copy()
dy_d = np.zeros(m, dtype= float)
dy_d[1:] = dy.copy()
dy_e = np.zeros(m, dtype= float)
dy_e[:m-1] = dy.copy()

a = Li + dx_e - dx_d
b = dy_e - dy_d

Lf = np.sqrt(a**2 + b**2)

#calculo da energia elastica em cada mola
U = (1/2)*(EA/Li)*((Lf - Li)**2)

#calculo do trabalho em cada no
V = W*dy

#Calculo da Energia Total
E = np.sum(U) - np.sum(V)

return E

def grad_f(Xn):
    dimens = Xn.size
    #numero de nos
    n = int(dimens/2)

    #numero de molas
    m = n + 1

    #Inicializacao dos vetores com as variaveis do problema
    Li = np.zeros(m, dtype=float) # comprimentos iniciais das molas
    EA = np.zeros(m, dtype=float)
    RHO = np.zeros(m, dtype=float)
    W = np.zeros(n, dtype=float) # peso em cada no

    #Atribuicao dos valores do problema
    #Cada mola mede inicialmente 60 sobre numero de molas
    #molas a esquerda possuem EA = 27000 e rho 8
    #molas a direita possuem EA = 18000 e rho 16
    Li = Li + 60/m

    EA[ : int(m/2)] = 27000
    EA[int(m/2) : ] = 18000
    RHO[ : int(m/2)] = 8
    RHO[int(m/2) : ] = 16

    #Calculo dos pesos atuando em cada no
    # W[j] = (1/2)*(RHO[j]*Li[j] + RHO[j+1]*Li[j+1])
    RHO_e = RHO[:m-1]
    RHO_d = RHO[1:m]
    Li_e = Li[:m-1]
    Li_d = Li[1:m]
    W = (1/2)*(RHO_e*Li_e + RHO_d*Li_d)

    Lf = np.zeros(m, dtype=float) # comprimentos finais das molas

    #array com os deslocamentos horizontais do Xn
    dx = Xn[0::2].copy()
    #array com os deslocamentos verticais do Xn
    dy = Xn[1::2].copy()

```

```

#Calculo dos comprimentos finais
# Lf[0] = np.sqrt( (Li[0] + dx[0] )**2 + dy[0]**2 )
# Lf[k] = np.sqrt(a**2 + b**2)
# Lf[m-1] = np.sqrt((Li[m-1] - dx[n-1])**2 + dy[n-1]**2)
dx_d = np.zeros(m, dtype= float)
dx_d[1:] = dx.copy()

dx_e = np.zeros(m, dtype=float)
dx_e[:m-1] = dx.copy()
dy_d = np.zeros(m, dtype= float)
dy_d[1:] = dy.copy()
dy_e = np.zeros(m, dtype=float)
dy_e[:m-1] = dy.copy()

a = Li + dx_e - dx_d
b = dy_e - dy_d

Lf = np.sqrt(a**2 + b**2)

#calculo gradiente
gradx = np.zeros(n, dtype=float)
grady = np.zeros(n, dtype=float)
Li_e = np.zeros(n, dtype=float)
Li_d = np.zeros(n, dtype=float)
EA_e = np.zeros(n, dtype=float)
EA_d = np.zeros(n, dtype=float)
Lf_e = np.zeros(n, dtype=float)
Lf_d = np.zeros(n, dtype=float)

Li_e = Li[:m-1]
Li_d = Li[1:m]
EA_e = EA[:m-1]
EA_d = EA[1:m]
Lf_e = Lf[:m-1]
Lf_d = Lf[1:m]

dx_d = np.zeros(n, dtype=float)
dx_d[1:] = dx[:m-2]

dy_d = np.zeros(n, dtype=float)
dy_d[1:] = dy[:m-2]

dx_e = np.zeros(n, dtype=float)
dx_e[:m-2] = dx[1:]

dy_e = np.zeros(n, dtype=float)
dy_e[:m-2] = dy[1:]

deriv1 = np.zeros(n, dtype=float)
deriv2 = np.zeros(n, dtype=float)
deriv3 = np.zeros(n, dtype=float)
deriv4 = np.zeros(n, dtype=float)
deriv5 = np.zeros(n, dtype=float)

deriv1 = (1/2)*((Li_e + dx - dx_d)**2 + (dy - dy_d)**2)**(-1/2)*(2*Li_e + 2*dx - 2*dx_d)
deriv2 = (1/2)*((Li_d + dx_e - dx)**2 + (dy_e - dy)**2)**(-1/2)*(-2*Li_d - 2*dx_e + 2*dx)
deriv3 = (1/2)*((Li_e + dx - dx_d)**2 + (dy - dy_d)**2)**(-1/2)*(2*dy - 2*dy_d)
deriv4 = (1/2)*((Li_d + dx_e - dx)**2 + (dy_e - dy)**2)**(-1/2)*(-2*dy_e + 2*dy)
deriv5 = W

# dUk/dxk + dU(k+1)/dxk
gradx = (1/2)*(EA_e/Li_e)*2*(Lf_e - Li_e)*deriv1 + (1/2)*(EA_d/Li_d)*2*(Lf_d - Li_d)*
deriv2

#dUk/dyk + dU(k+1)/dyk - dVk/dyk
grady = (1/2)*(EA_e/Li_e)*2*(Lf_e - Li_e)*deriv3 + (1/2)*(EA_d/Li_d)*2*(Lf_d - Li_d)*
deriv4 - deriv5

grad = np.zeros(2*n, dtype=float)
grad[0::2] = gradx
grad[1::2] = grady

return grad

```

```

def hessian_f(Xn):
    return nd.Hessian(f)(Xn)

func = 4

```

4.2.3 Resultados

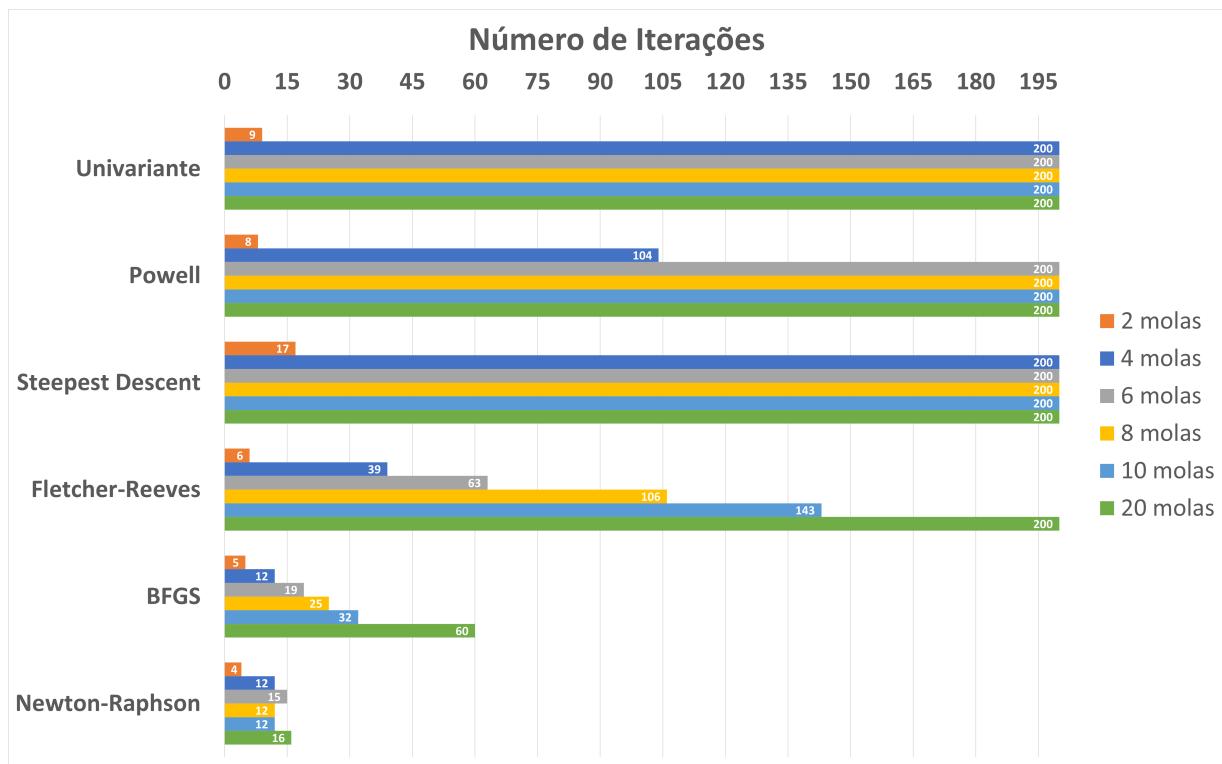


Figura 21: Número de passos, por método OSR, para diferentes números de molas. Questão 2b.

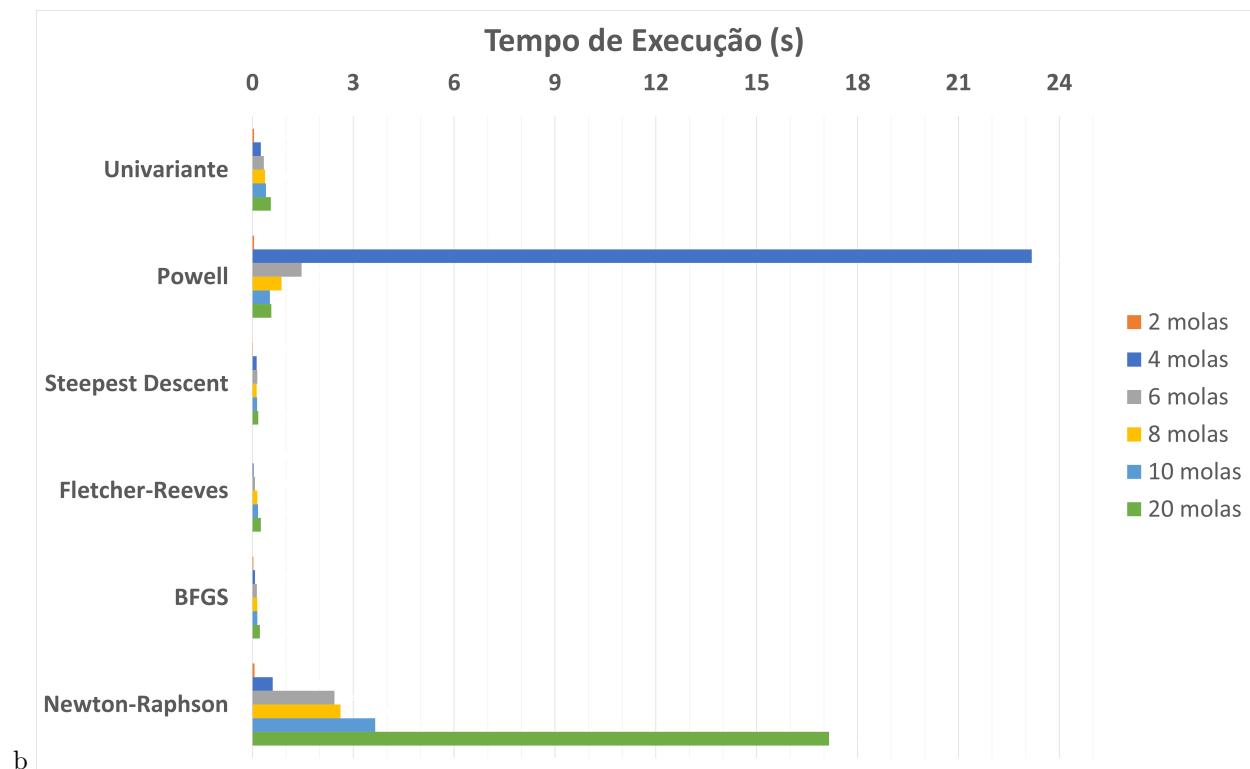


Figura 22: Tempo de Execução, por método OSR, para diferentes números de molas. Questão 2b.

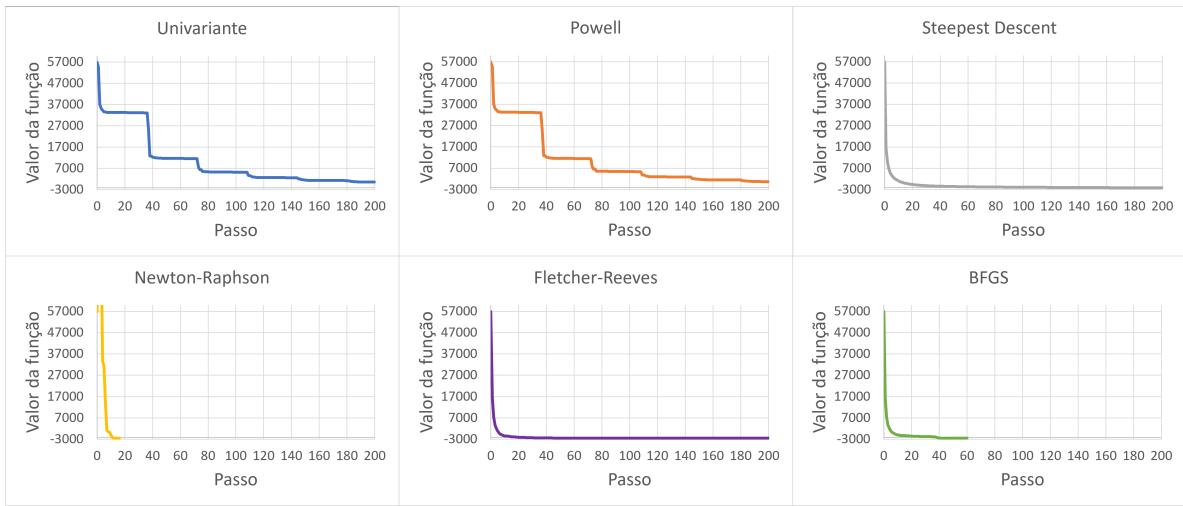


Figura 23: Gráficos de $f(x_1, x_2)$ versus passo da minimização, por método. Questão 2b com 20 molas

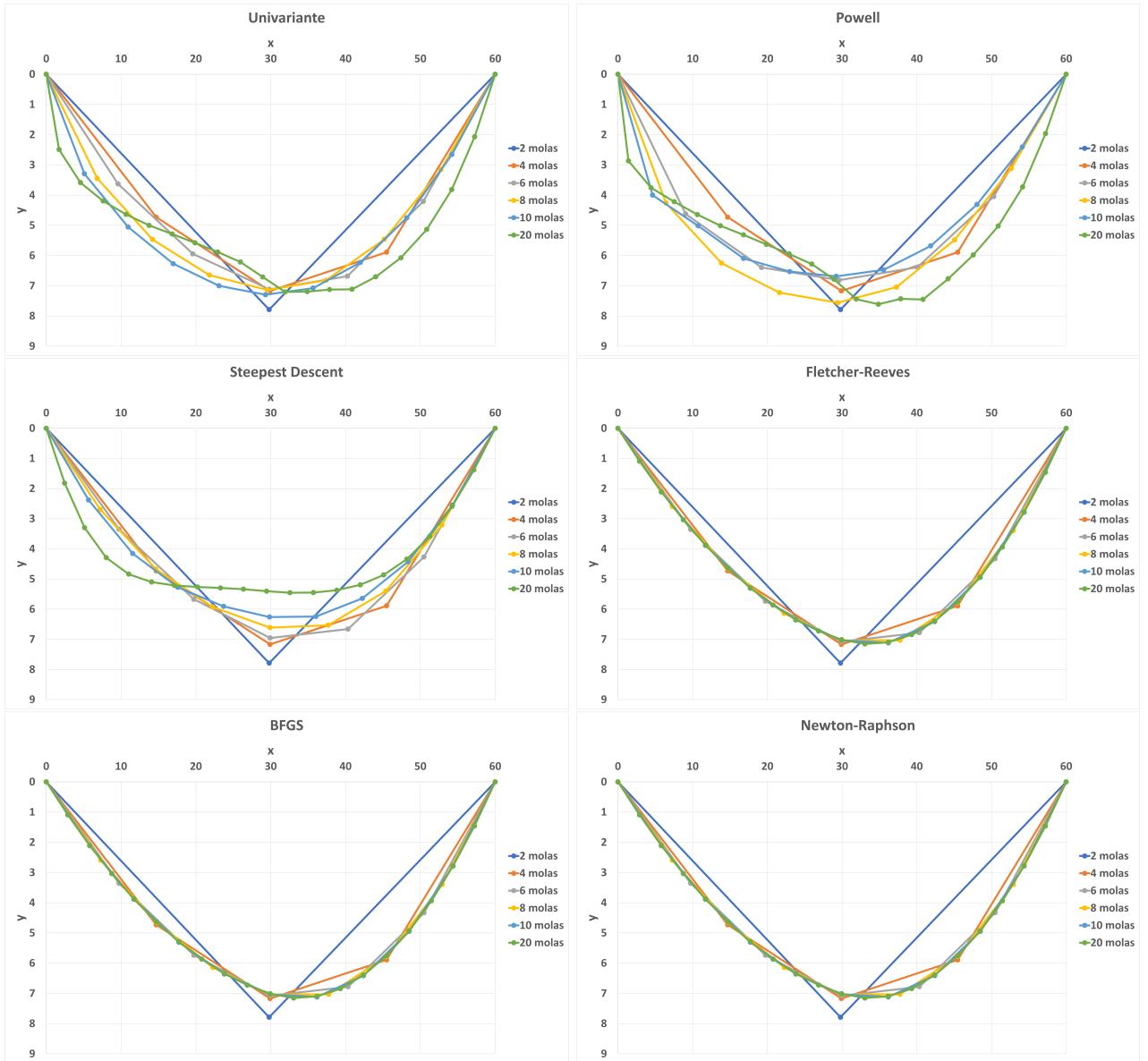


Figura 24: Posição dos nós, por método, para diferentes números de molas