# Trabalho 2

**Opção 01**

MEC 2403 - Otimização, Algoritmos e Aplicações na Engenharia Mecânica

**Gustavo Henrique Gomes dos Santos**
gustavohgs@gmail.com

Professor: Ivan Menezes


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

# Trabalho 2

MEC 2403 - Otimização, Algoritmos e Aplicações na Engenharia Mecânica

Gustavo Henrique Gomes dos Santos

junho de 2023

## 1 Introdução

### 1.1 Objetivos

Esse trabalho tem como objetivo a implementação e teste dos seguintes métodos indiretos de otimização com restrição :

1. Penalidade
2. Barreira

Para a etapa de sequência de otimização sem restrição, da qual esses métodos fazem uso, foram utilizados os métodos implementados no trabalho 1. São eles:

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

Estes métodos, por sua vez, fazem uso dos seguintes algoritmos de busca unidimensional também implementados em trabalhos anteriores :

1. Passo constante
2. Seção áurea

## 2 Implementação

Foi utilizada a linguagem de programação Python para elaboração deste trabalho. A implementação consiste de um arquivo com o código principal, um segundo arquivo com os algoritmos dos métodos de otimização com restrição, um terceiro arquivo que encapsula os métodos e a convergência da otimização sem restrição e um quarto arquivo com os algoritmos da busca unidimensional.

### 2.1 Código Principal

O código principal trata das definições do ponto inicial, função a ser minimizada e as restrições de igualdade e desigualdade. A escolha de quais métodos OCR e OSR serão utilizados e a definição dos parâmetros numéricos também são feitos no código principal. Além disso, o controle de passos da otimização OC e a verificação de convergência.

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

```
import numpy as np
import matplotlib.pyplot as plt
import osr_methods as osr
import line_search_methods as lsm
import ocr_methods as ocr
```

Definição do ponto inicial, que irá variar em cada teste realizado com diferentes funções e método OCR.

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

Escolha dos métodos de OCR e OSR.

```
# Metodos OCR
# 1 - Penalidade
# 2 - Barreira
metodo_ocr = 1

if (metodo_ocr == 1):
    n_met_ocr = "Penalidade"
elif (metodo_ocr == 2):
    n_met_ocr = "Barreira"

# Metodos OSR
# 1 - Univariante
# 2 - Powell
# 3 - Stepest Descent
# 4 - Newton-Raphson
# 5 - Fletcher-Reeves
# 6 - BFGS
metodo_osr = 4

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

Controle numérico

```
# numero maximo de iteracoes na OSR
maxiter = 1000

# tolerancia para convergencia do gradiente na OSR
tol_conv = 1E-6

# tolerancia para a busca unidirecional na OSR
tol_search = 1E-7

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

#epsilon da maquina
eps = 1E-10

#parametros ocr
if metodo_ocr == 1:
    #penalidade
    r = 1
    beta = 10
elif metodo_ocr == 2:
    #barreira
    r = 10
    beta = 0.1

#tolerancia OCR
```

```
tol = 1E-6

ctrl_num_osr = [maxiter, tol_conv, tol_search, line_step, eps]
```

Definição de $f(\overrightarrow{x})$, $\overrightarrow{\nabla} f(\overrightarrow{x})$ e $\mathbf{H}(\overrightarrow{x})$. As reticências no código abaixo serão substituiídas pelos valores adequados a cada função.

```
def f(x):
  return ...

def grad_f(x):
  return ...

def hess_f(x):
  hess = np.zeros((2,2), dtype=float)
  hess[0,:] = ...
  hess[1,:] = ...
  return hess
```

Definição das restrições $c_l(\overrightarrow{x})$, $\overrightarrow{\nabla} c_l(\overrightarrow{x})$, $h_k(\overrightarrow{x})$, $\overrightarrow{\nabla} h_k(\overrightarrow{x})$, $\mathbf{W}_{c_l}(\overrightarrow{x})$ e $\mathbf{W}_{h_k}(\overrightarrow{x})$, sendo W a hessiana da restrição. As reticências no código abaixo serão substituídas pelos valores adequados a cada função. Podem ser definidas quantas restrições forem desejadas, apesar do exemplo abaixo estar com apenas uma de desigualdade e uma de igualdade.

```
def h1(x):
  return ...

def grad_h1(x):
  return ...

def hess_h1(x):
  hess = np.zeros((2,2), dtype=float)
  hess[0,:] = ...
  hess[1,:] = ...
  return hess

def c1(x):
  return ...

def grad_c1(x):
  return ...

def hess_c1(x):
  hess = np.zeros((2,2), dtype=float)
  hess[0,:] = ...
  hess[1,:] = ...
  return hess
```

Agrupamento das restrições em listas(ou arrays) para servirem de input para o restante do programa. E montagem de um array auxiliar para montagem da função $\phi$ no caso do método OCR de penalidade.

```
h_list = [h1]
grad_h_list = [grad_h1]
hess_h_list = [hess_h1]

c_list = [c1]
grad_c_list = [grad_c1]
hess_c_list = [hess_c1]

#para o metodo de penalidade
#controle de quais cls irao montar a phi
c_mont = []
if metodo_ocr == 1:
    for c in c_list:
        if c(x) > 0:
            c_mont.append(1)
        else:
            c_mont.append(0)
```

```
params = [f, grad_f, hess_f, h_list, grad_h_list, hess_h_list, c_list, grad_c_list,
                                                    hess_c_list, c_mont]
```

Verificação de convergência.

1. Penalidade : $\frac{1}{2} r_p^k p(\overrightarrow{x}^{k+1}) < tol$, sendo $p(\overrightarrow{x}) = \sum_{k=1}^{m} h_k^2(\overrightarrow{x}) + \sum_{l=1}^{p} \{max[0, c_l(\overrightarrow{x})]\}^2$
2. Barreira : $r_b^k b(\overrightarrow{x}^{k+1}) < tol$, sendo $b = \sum_{l=1}^{p} -\frac{1}{c_l(\overrightarrow{x})}$

A cada iteração do loop no código abaixo, o valor de $r$ é atualizado pelo fator $\beta$, e os parâmetros necessários para montagem da função $\phi$ pelos métodos da penalidade e da barreira também são atualizados. Após cada atualização e verificação de convergência, a otimização sem restrição é chamada, e esse processo é repetido até que seja atingido o critério de convergência da otimização com restrição.

Para o método da barreira também é feita uma avaliação se o ponto final da iteração continua dentro das restrições. Caso negativo, o passo é refeito com um passo reduzido no método do passo constante da busca unidimensional.

```
if metodo_ocr == 1:
  parc = (1/2)*r*ocr.p_penal(x, params)
elif metodo_ocr == 2:
  parc = r*ocr.b_bar(x, params)

listP_OCR = []
listP_OCR.append(x)

listResultsOSR = []

passos_OCR = 0
redo = 0
print(n_met)
while(parc > tol):
  passos_OCR = passos_OCR + 1
  if passos_OCR > 1:
      r = beta*r
      if metodo_ocr == 1:
          params[-1] = []
          for c in c_list:
              if c(x) > 0:
                  params[-1].append(1)
              else:
                  params[-1].append(0)
  listP_OSR, passos_OSR, conv_OSR, flag_conv_OSR, tempoExec_OSR = osr.osr_ctrl(x, params, r,
                                                      ctrl_num_osr, metodo_ocr, metodo_osr)

  if metodo_ocr == 2:
      redo = 0
      for c in c_list:
          if c(listP_OSR[-1]) > 0:
              redo = 1
              break
  if (redo == 0):
      ctrl_num_osr[3] = line_step
      x = listP_OSR[-1]
      listP_OCR.append(x)
      listResultsOSR.append([listP_OSR, params, r, metodo_ocr, metodo_osr])
      if metodo_ocr == 1:
          parc = (1/2)*r*ocr.p_penal(x, params)
      elif metodo_ocr == 2:
          parc = r*ocr.b_bar(x, params)
      print(f'{passos_OCR}: x={x}, r={r:.4e}, passos={passos_OSR}, conv_OCR={parc:.4e},
                                                      conv_OSR={conv_OSR:.4e}, tempo={
                                                      tempoExec_OSR}')
  elif (redo == 1):
      print(f'Refazendo passo {passos_OCR} com delta alpha = {0.1*ctrl_num_osr[3]}')
      passos_OCR = passos_OCR - 1
      r = r/beta
      ctrl_num_osr[3] = 0.1*ctrl_num_osr[3]
```

## 2.2 Métodos OCR

Os algoritmos referentes ao uso dos métdos OCR da penalidade e da barreira foram implementados em um arquivo denominado ocr_methods.py.

O seguinte pacote é necessário nesse arquivo:

```
import numpy as np
```

### 2.2.1 Método de Penalidade

**Pseudo-Função Objetivo:**

$\phi(\overrightarrow{x}, r_p) = f(\overrightarrow{x}) + \frac{1}{2} r_p \sum_{k=1}^{m} h_k^2(\overrightarrow{x}) + \frac{1}{2} r_p \sum_{l=1}^{p} \{max[0, c_l(\overrightarrow{x})]\}^2$

**Cálculo do gradiente:**

Sendo $p(\overrightarrow{x}) = \sum_{k=1}^{m} h_k^2(\overrightarrow{x}) + \sum_{l=1}^{p} \{max[0, c_l(\overrightarrow{x})]\}^2$, temos então que $\overrightarrow{\nabla}\phi(\overrightarrow{x}, r_p) = \overrightarrow{\nabla} f(\overrightarrow{x}) + \frac{1}{2} r_p \overrightarrow{\nabla} p(\overrightarrow{x})$.

$\overrightarrow{\nabla} p(\overrightarrow{x}) = 2 \sum_{k=1}^{m} \{h_k(\overrightarrow{x}) \overrightarrow{\nabla} h_k(\overrightarrow{x})\} + 2 \sum_{l=1}^{p} \{c_l^f(\overrightarrow{x}) c_l(\overrightarrow{x}) \overrightarrow{\nabla} c_l(\overrightarrow{x})\}$, sendo $c_l^f(\overrightarrow{x}) = \begin{cases} 1, & \text{se } c_l(\overrightarrow{x}) > 0 \\ 0, & \text{se } c_l(\overrightarrow{x}) \leq 0 \end{cases}$

**Cálculo da Hessiana:**

$H_\phi(\overrightarrow{x}) = H_f(\overrightarrow{x}) + \frac{1}{2} r_p H_p(\overrightarrow{x})$

$H_{p_{ixj}} = \frac{\partial^2 p}{\partial x_i \partial x_j} = 2 \sum_{k=1}^{m} \{\frac{\partial h_k}{\partial x_j} \frac{\partial h_k}{\partial x_i}\} + 2 \sum_{k=1}^{m} \{h_k \frac{\partial^2 h_k}{\partial x_j \partial x_i}\} + 2 \sum_{l=1}^{p} \{c_l^f \frac{\partial c_l}{\partial x_j} \frac{\partial c_l}{\partial x_i}\} + 2 \sum_{l=1}^{p} \{c_l^f c_l \frac{\partial^2 c_l}{\partial x_j \partial x_i}\}$

Interessante notar que $\frac{\partial c_l}{\partial x_j}$, $\frac{\partial c_l}{\partial x_i}$, $\frac{\partial h_k}{\partial x_j}$ e $\frac{\partial h_k}{\partial x_i}$ são componentes conhecidos de $\overrightarrow{\nabla} h_k$ e $\overrightarrow{\nabla} c_l$. Além disso, $\frac{\partial^2 c_l}{\partial x_j \partial x_i}$ e $\frac{\partial^2 h_k}{\partial x_j \partial x_i}$ são termos das hessianas das restrições e que também são conhecidos. Dessa forma, temos todos os inputs necessários para cálculo da hessiana de $p(\overrightarrow{x})$ e por conseguinte a hessiana de $\phi(\overrightarrow{x}, r)$.

```
#Metodo da Penalidade
def p_penal(x, params):
    #leitura dos parametros
    h_list = params[3]
    c_list = params[6]
    c_mont = params[9]

    p = 0
    for h in h_list:
        p = p + (h(x))**2

    for i in np.arange(len(c_list)):
        p = p + c_mont[i]*c_list[i](x)**2

    return  p

def phi_penal(x, params, r):
    #leitura dos parametros
    f = params[0]
    h_list = params[3]
    c_list = params[6]
    c_mont = params[9]

    p = 0
    for h in h_list:
        p = p + (h(x))**2

    for i in np.arange(len(c_list)):
        p = p + c_mont[i]*c_list[i](x)**2

    return f(x) + (1/2)*r*p

def grad_phi_penal(x, params, r):
    #leitura dos parametros
    grad_f = params[1]
    h_list = params[3]
    grad_h_list = params[4]
    c_list = params[6]
    grad_c_list = params[7]
    c_mont = params[9]

    dimens = x.size
```

```
    grad_p = np.zeros(dimens, dtype=float)

    for i in np.arange(len(h_list)):
        grad_p = grad_p + 2*h_list[i](x)*grad_h_list[i](x)
    for j in np.arange(len(c_list)):
        grad_p = grad_p + 2*c_mont[j]*c_list[j](x)*grad_c_list[j](x)

    return grad_f(x) + (1/2)*r*grad_p

def hess_phi_penal(x, params, r):
    #leitura dos parametros
    hess_f = params[2]
    h_list = params[3]
    grad_h_list = params[4]
    hess_h_list = params[5]
    c_list = params[6]
    grad_c_list = params[7]
    hess_c_list = params[8]
    c_mont = params[9]

    dimens = x.size
    hessian_p = np.zeros((dimens, dimens), dtype=float)

    for i in np.arange(dimens):
        for j in np.arange(dimens):
            for k in np.arange(len(grad_h_list)):
                hessian_p[i,j] = hessian_p[i,j] + 2*grad_h_list[k](x)[i]*grad_h_list[k](x)[j
                                                                                           ]
            for l in np.arange(len(grad_cl_list)):
                hessian_p[i,j] = hessian_p[i,j] + 2*c_mont[l]*grad_c_list[l](x)[j]*
                                                                grad_c_list[l](x)[i]

    for k in np.arange(len(h_list)):
        hessian_p = hessian_p + 2*h_list[k](x)*hess_h_list[k](x)

    for k in np.arange(len(c_list)):
        hessian_p = hessian_p + 2*c_mont[k]*c_list[k](x)*hess_c_list[k](x)

    return hess_f(x) + (1/2)*r*hessian_p
```

### 2.2.2 Método de Barreira

**Pseudo-Função Objetivo:**

$\phi(\overrightarrow{x}, r_b) = f(\overrightarrow{x}) + r_b \sum_{l=1}^{m} -\frac{1}{c_l(\overrightarrow{x})}$

**Cálculo do gradiente:**

Sendo $b(\overrightarrow{x}) = \sum_{l=1}^{m} -\frac{1}{c_l(\overrightarrow{x})}$ , temos então que $\overrightarrow{\nabla}\phi(\overrightarrow{x}, r_b) = \overrightarrow{\nabla} f(\overrightarrow{x}) + r_b \overrightarrow{\nabla} b(\overrightarrow{x})$.

$\overrightarrow{\nabla} b = \sum_{l=1}^{m} \frac{\overrightarrow{\nabla} c_l}{c_l^2}$

**Cálculo da Hessiana:**

$H_\phi(\overrightarrow{x}) = H_f(\overrightarrow{x}) + r_b H_b(\overrightarrow{x})$

$H_{b_{ixj}} = -2\sum_{l=1}^{m}\{\frac{1}{c_l^3}\frac{\partial c_l}{\partial x_i}\frac{\partial c_l}{\partial x_j}\} + \sum_{l=1}^{m}\{\frac{1}{c_l^2}\frac{\partial^2 c_l}{\partial x_i \partial x_j}\}$

$\frac{\partial c_l}{\partial x_j}$ e $\frac{\partial c_l}{\partial x_i}$ são componentes conhecidos de $\overrightarrow{\nabla} c_l$. Além disso, $\frac{\partial^2 c_l}{\partial x_j \partial x_i}$ são termos da hessiana das restrições e que também são conhecidos. Dessa forma, temos todos os inputs necessários para cálculo da hessiana de $b(\overrightarrow{x})$ e por conseguinte a hessiana de $\phi(\overrightarrow{x}, r)$.

```
#### Metodo da Barreira
def phi_bar(x, params, r):
    #leitura dos parametros
    f = params[0]
    c_list = params[6]

    b = 0
    for c in c_list:
```

```
        b = b - 1/cl(x)

    return f(x) + r*b

def b_bar(x, params):
    #leitura dos parametros
    c_list = params[6]

    b = 0
    for c in c_list:
        b = b - 1/c(x)

    return b

def grad_phi_bar(x, params, r):
    #leitura dos parametros
    grad_f = params[1]
    c_list = params[6]
    grad_c_list = params[7]

    dimens = x.size
    grad_b = np.zeros(dimens, dtype=float)

    for i in np.arange(len(c_list)):
        grad_b = grad_b + (c_list[i](x))**(-2)*grad_c_list[i](x)

    return grad_f(x) + r*grad_b

def hess_phi_bar(x, params, r):
    #leitura dos parametros
    hess_f = params[2]
    c_list = params[6]
    grad_c_list = params[7]
    hess_c_list = params[8]

    dimens = x.size
    hessian_b = np.zeros((dimens, dimens), dtype=float)

    for i in np.arange(dimens):
        for j in np.arange(dimens):
            for k in np.arange(len(c_list)):
                hessian_b[i,j] = hessian_b[i,j] - 2*((c_list[k](x))**(-3))*grad_c_list[k](x)
                                                                    [i]*grad_c_list[k](x)[j]

    for k in np.arange(len(cl_list)):
        hessian_b = hessian_b + ((c_list[k](x))**(-2))*hess_c_list[k](x)

    return hess_f(x) + r*hessian_b
```

## 2.3 Métodos e Otimização OSR

Os algoritmos de otimização sem restrição, Univariante, Powell, Steepest Descent, Newton-Raphson, Flecther-Reeves e BFGS foram implementados em um arquivo denominado osr_methods.py. Uma função chamada osr_ctrl também está presente nesse arquivo e faz a interface entre o código principal e os métodos de OSR. O código principal chama essa função a cada iteração OCR, e ela por sua vez faz todo o tratamento da OSR, chamando as funções $\phi$ do módulo OCR apresentado na seção anterior E retornando os resultados para o código principal.

Esses códigos foram discutidos com mais detalhes no trabalho 1.

```
import numpy as np
import osr_methods as osr
import line_search_methods as lsm
import ocr_methods as ocr
from timeit import default_timer as timer

def univariante(passo, dimens):
    #indice do vetor = (resto da divisao do passo pela dimensao) - 1
    #primeira posicao do vetor no python tem indice 0
    indice = passo%dimens - 1

    if (indice == -1) :
        #indice = -1 indica que se trata da ultima posicao do array
        #no pyton esse indice eh o tamanho do vetor - 1
        indice = dimens - 1
```

```
    #define a direcao canonica a ser utilizada
    ek = np.zeros(dimens)
    ek[indice] = 1

    return ek

def powell(P, P0, direcoes, passos, ciclos, dimens):
    #indice do vetor = (resto da divisao do passo pela dimensao) - 1
    #primeira posicao do vetor no python tem indice 0
    indice = passos%(dimens + 1) - 1

    if (indice == -1):
        #indice = -1 indica que se trata da ultima posicao do array
        #no pyton esse indice eh o tamanho do vetor - 1
        #direcao n + 1 do ciclo = Patual - P0
        dir = P - P0
        direcoes[dimens - 1] = dir
    elif (indice == 0):
        #indice = 0 significa que vamos usar a primeira direcao do conjunto
        #representa o inicio de um novo ciclo
        ciclos = ciclos + 1

        if (ciclos%(dimens+2) == 0):
            #se ciclo for multipl de dimens + 2, conjunto de direcoes = canonicas
            direcoes = np.eye(dimens, dtype=float)
        P0 = P.copy()
        dir = direcoes[indice].copy()

    else:
        dir = direcoes[indice].copy()
        direcoes[indice-1] = dir

    return dir, direcoes, P0, ciclos

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

def steepestDescent(grad):
    return -grad

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

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 necessrio
                                                                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

def osr_ctrl(P0, params, r, ctrl_num, metodo_ocr, metodo_osr):
    #controle numerico
```

```
maxiter = ctrl_num[0]
tol_conv = ctrl_num[1]
tol_search = ctrl_num[2]
line_step = ctrl_num[3]
eps = ctrl_num[4]

metodo = metodo_osr

#inicializacoes auxiliares dos metodos de OSR
passos = 0
dimens = P0.size
Pmin = P0.copy()
listPmin = []
listPmin.append(Pmin)

if metodo_ocr == 1:
    grad = ocr.grad_phi_penal(Pmin, params, r)
elif metodo_ocr == 2:
    grad = ocr.grad_phi_bar(Pmin, params, r)

norm_grad = np.linalg.norm(grad)
flag_conv = True

if (metodo == 2):
    direcoes = np.eye(dimens, dtype=float)
    ciclos = 0
    P1 = P0.copy()
elif (metodo == 5):
    #o metodo recebe a direcao anterior
    #inicializo a direcao com um vetor de zeros mas que nunca e usado
    #uso apenas para enviar como parametro na primeira iteracao do metodo, o qual
                                                    atualiza o valor de dir para a
                                                    iteracao 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()

#calculo do Pmin
start = timer()
while (norm_grad > tol_conv):
    if (passos == maxiter):
        flag_conv = False
        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):
        if metodo_ocr == 1:
            hess = ocr.hess_phi_penal(Pmin, params, r)
        elif metodo_ocr == 2:
            hess = ocr.hess_phi_bar(Pmin, params,r)
        dir = osr.newtonRaphson(grad, hess)
    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)
    dir = dir/np.linalg.norm(dir)
    intervalo = lsm.passo_cte(dir, Pmin, params, r, metodo_ocr, eps, line_step)
    alpha = lsm.secao_aurea(intervalo, dir, Pmin, params, r, metodo_ocr, tol_search)
    Pmin = Pmin + alpha*dir
    listPmin.append(Pmin)


    if metodo_ocr == 1:
        grad = ocr.grad_phi_penal(Pmin, params, r)
    elif metodo_ocr == 2:
        grad = ocr.grad_phi_bar(Pmin, params, r)
```

```
        norm_grad = np.linalg.norm(grad)

    end = timer()
    tempoExec = end - start

    return listPmin, passos, norm_grad, flag_conv, tempoExec
```

## 2.4 Busca Unidirecional

Os algoritmos dos métodos do Passo Constante e da Seção Áurea foram implementados em um arquivo denominado line_search_methods.py. Códigos discutidos no trabalho 1. Pequena adaptação realizada para trabalhar com as funções $\phi$ dos métodos OCR.

```
import ocr_methods as ocr
import numpy as np

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

    #define o sentido correto de busca
    if metodo_ocr == 1:
        f1 = ocr.phi_penal(P0 - eps*(direcao/np.linalg.norm(direcao)), params, r)
        f2 = ocr.phi_penal(P0 + eps*(direcao/np.linalg.norm(direcao)), params, r)
    elif metodo_ocr == 2:
        f1 = ocr.phi_bar(P0 - eps*(direcao/np.linalg.norm(direcao)), params, r)
        f2 = ocr.phi_bar(P0 + eps*(direcao/np.linalg.norm(direcao)), params, r)

    if (f1 > f2):
        sentido_busca = direcao.copy()
        flag = 0
    else:
        sentido_busca = -direcao.copy()
        flag = 1

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

    if metodo_ocr == 1:
        f1 = ocr.phi_penal(P, params, r)
        f2 = ocr.phi_penal(P_next, params, r)
    elif metodo_ocr == 2:
        f1 = ocr.phi_bar(P, params, r)
        f2 = ocr.phi_bar(P_next, params, r)

    while (f1 > f2):
        alpha = alpha + step
        P = P0 + alpha*sentido_busca
        P_next = P0 + (alpha+step)*sentido_busca

        if metodo_ocr == 1:
            f1 = ocr.phi_penal(P, params, r)
            f2 = ocr.phi_penal(P_next, params, r)
            f_eps = ocr.phi_penal(P - eps*(sentido_busca/np.linalg.norm(sentido_busca)),
                                                        params, r)
        elif metodo_ocr == 2:
            f1 = ocr.phi_bar(P, params, r)
            f2 = ocr.phi_bar(P_next, params, r)
            f_eps = ocr.phi_bar(P - eps*(sentido_busca/np.linalg.norm(sentido_busca)),
                                                        params, r)

        if (f_eps < f1):
            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

def secao_aurea(intervalo, direcao, P0, params, r, metodo_ocr, 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
    if metodo_ocr == 1:
        f1 = ocr.phi_penal(P0 + alpha_e*sentido_busca, params, r)
        f2 = ocr.phi_penal(P0 + alpha_d*sentido_busca, params, r)
    elif metodo_ocr == 2:
        f1 = ocr.phi_bar(P0 + alpha_e*sentido_busca, params, r)
        f2 = ocr.phi_bar(P0 + alpha_d*sentido_busca, params, r)

    #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_d = alpha_lower + Ra*beta

            if metodo_ocr == 1:
                f2 = ocr.phi_penal(P0 + alpha_d*sentido_busca, params, r)
            elif metodo_ocr == 2:
                f2 = ocr.phi_bar(P0 + alpha_d*sentido_busca, params, r)

        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

            if metodo_ocr == 1:
                f1 = ocr.phi_penal(P0 + alpha_e*sentido_busca, params, r)
            elif metodo_ocr == 2:
                f1 = ocr.phi_bar(P0 + alpha_e*sentido_busca, params, r)

    # 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
```

# 3 Teste da Implementação

## 3.1 Problema 1

$$\begin{cases} \textbf{Min} & f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2 \\ \textbf{s.t.:} & x_1^2 - x_2 \leq 0 \end{cases}$$

Obs.: Adotar $r_p^0 = 1$, $\beta = 10$ e $x^0 = \{3, 2\}$ para o método de penalidade e $r_b^0 = 10$, $\beta = 0.1$ e $x^0 = \{0, 1\}$ para o método de barreira.

$\overrightarrow{\nabla} f(\overrightarrow{x}) = \{4(x_1 - 2)^3 + 2(x_1 - 2x_2), -4(x_1 - 2x_2)\}$

$H_{f_{1x1}} = 12(x_1 - 2)^2 + 2, \quad H_{f_{1x2}} = -4, \quad H_{f_{2x1}} = -4, \quad H_{f_{2x2}} = 8$

**Definição da função, seu gradiente e sua hessiana, no código principal:**

```
def f(x):
    return (x[0]-2)**4 + (x[0] - 2*x[1])**2

def grad_f(x):
    return np.array([4*(x[0]-2)**3 + 2*(x[0] - 2*x[1]), 2*(x[0] - 2*x[1])*(-2)])

def hess_f(x):
    hess = np.zeros((2,2), dtype=float)
    hess[0,:] = np.array([12*(x[0]-2)**2 + 2, -4.])
    hess[1,:] = np.array([-4., 8.])
    return hess
```

$c(\overrightarrow{x}) = x_1^2 - x_2$

$\overrightarrow{\nabla} c(\overrightarrow{x}) = \{2x_1, -1\}$

$H_{c_{1x1}} = 2, \quad H_{c_{1x2}} = 0, \quad H_{c_{2x1}} = 0, \quad H_{c_{2x2}} = 0$

**Definição da restrição, seu gradiente e sua hessiana, no código principal:**

```
def c1(x):
    return x[0]**2 - x[1]

def grad_c1(x):
    return np.array([2*x[0], -1.])

def hess_c1(x):
    hess = np.zeros((2,2), dtype=float)
    hess[0,:] = np.array([2., 0.])
    hess[1,:] = np.array([0., 0])
    return hess
```

### 3.1.1 Penalidade - Prob. 1

Definição do ponto inicial $x^0 = \{3, 2\}$ no código principal :

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

**Controle numérico e parâmetros:**

- Máximas iterações na OSR : 1000
- Tolerância OSR: $10^{-6}$
- Tolerância Seção Áurea: $10^{-7}$
- $\Delta\alpha$: $10^{-2}$
- $\epsilon$: $10^{-10}$
- Tolerância OCR: $10^{-6}$
- $r_p^0 = 1$
- $\beta = 10$

**Prob. 1 - Penalidade - Univariante**

| Iter | $P_{min}$ | r | # Passos | Conv_OCR | Conv_OSR | t(s) |
|---|---|---|---|---|---|---|
| 1 | [1.25174114 0.7304246 ] | 1e+00 | 27 | 3.5e-01 | 9.6e-07 | 0.028 |
| 2 | [1.02501305 0.81147611] | 1e+01 | 1000 | 2.9e-01 | 3.7e-06 | 0.230 |
| 3 | [0.95576688 0.88122316] | 1e+02 | 1000 | 5.2e-02 | 3.5e-06 | 0.189 |
| 4 | [0.94659013 0.89267782] | 1e+03 | 1000 | 5.6e-03 | 1.6e-03 | 0.214 |
| 5 | [0.94526659 0.89319248] | 1e+04 | 1000 | 5.7e-04 | 1.5e-02 | 0.190 |
| 6 | [0.94513054 0.89323808] | 1e+05 | 1000 | 5.7e-05 | 1.5e-02 | 0.195 |
| 7 | [0.94511456 0.89323814] | 1e+06 | 1000 | 5.7e-06 | 3.3e-02 | 0.180 |
| 8 | [0.94511298 0.8932382 ] | 1e+07 | 1000 | 6.2e-07 | 3.0e-01 | 0.228 |

**Tabela 1:** Resultados obtidos para o problema 1, método de penalidade, univariante para $x^0 = \{3, 2\}$

**Prob. 1 - Penalidade - Powell**

| Iter | $P_{min}$ | r | # Passos | Conv_OCR | Conv_OSR | t(s) |
|---|---|---|---|---|---|---|
| 1 | [1.25174105 0.73042442] | 1e+00 | 6 | 3.5e-01 | 2.2e-07 | 0.027 |
| 2 | [1.02501313 0.81147619] | 1e+01 | 7 | 2.9e-01 | 9.1e-07 | 0.005 |
| 3 | [0.95576689 0.88122318] | 1e+02 | 53 | 5.2e-02 | 5.1e-07 | 0.023 |
| 4 | [0.94663397 0.89276033] | 1e+03 | 38 | 5.6e-03 | 4.8e-07 | 0.013 |
| 5 | [0.94568845 0.89398972] | 1e+04 | 30 | 5.7e-04 | 2.8e-07 | 0.009 |
| 6 | [0.94559354 0.89411344] | 1e+05 | 78 | 5.7e-05 | 3.0e-08 | 0.019 |
| 7 | [0.94558401 0.89412573] | 1e+06 | 1000 | 5.8e-06 | 5.2e-02 | 0.264 |
| 8 | [0.94558315 0.89412714] | 1e+07 | 1000 | 5.9e-07 | 1.4e-01 | 0.217 |

**Tabela 2:** Resultados obtidos para o problema 1, método de penalidade, powell para $x^0 = \{3, 2\}$

**Prob. 1 - Penalidade - Steepest Descent**

| Iter | $P_{min}$ | r | # Passos | Conv_OCR | Conv_OSR | t(s) |
|---|---|---|---|---|---|---|
| 1 | [1.2517411 0.73042453] | 1e+00 | 25 | 3.5e-01 | 6.2e-07 | 0.025 |
| 2 | [1.02501316 0.81147628] | 1e+01 | 1000 | 2.9e-01 | 1.6e-06 | 0.254 |
| 3 | [0.9557669 0.88122324] | 1e+02 | 1000 | 5.2e-02 | 1.2e-05 | 0.248 |
| 4 | [0.94663395 0.89276024] | 1e+03 | 1000 | 5.6e-03 | 1.1e-04 | 0.210 |
| 5 | [0.94568838 0.89398961] | 1e+04 | 1000 | 5.7e-04 | 1.3e-04 | 0.250 |
| 6 | [0.94556416 0.89405783] | 1e+05 | 1000 | 5.7e-05 | 1.1e-02 | 0.203 |
| 7 | [0.94555192 0.89406498] | 1e+06 | 1000 | 6.0e-06 | 1.8e-01 | 0.239 |
| 8 | [0.94555068 0.89406568] | 1e+07 | 1000 | 8.9e-07 | 1.8e+00 | 0.212 |

**Tabela 3:** Resultados obtidos para o problema 1, método de penalidade, steepest descent para $x^0 = \{3, 2\}$

**Prob. 1 - Penalidade - Newton-Raphson**

| Iter | $P_{min}$ | r | # Passos | Conv_OCR | Conv_OSR | t(s) |
|---|---|---|---|---|---|---|
| 1 | [1.25174109 0.73042445] | 1e+00 | 3 | 3.5e-01 | 2.7e-07 | 0.035 |
| 2 | [1.02501318 0.81147627] | 1e+01 | 4 | 2.9e-01 | 8.9e-08 | 0.009 |
| 3 | [0.95576692 0.88122322] | 1e+02 | 4 | 5.2e-02 | 1.0e-07 | 0.008 |
| 4 | [0.94663397 0.89276032] | 1e+03 | 1000 | 5.6e-03 | 4.6e-06 | 0.668 |
| 5 | [0.94568841 0.89398971] | 1e+04 | 1000 | 5.7e-04 | 1.3e-03 | 0.422 |
| 6 | [0.94559354 0.89411344] | 1e+05 | 1000 | 5.7e-05 | 5.1e-04 | 0.502 |
| 7 | [0.94558405 0.89412582] | 1e+06 | 1000 | 5.7e-06 | 2.0e-04 | 0.526 |
| 8 | [0.9455831 0.89412707] | 1e+07 | 1000 | 5.7e-07 | 4.2e-03 | 0.401 |

**Tabela 4:** Resultados obtidos para o problema 1, método de penalidade, Newton-Raphson para $x^0 = \{3, 2\}$

**Prob. 1 - Penalidade - Fletcher-Reeves**

| Iter | $P_{min}$ | r | # Passos | Conv_OCR | Conv_OSR | t(s) |
|---|---|---|---|---|---|---|
| 1 | [1.24978283 0.72770155] | 1e+00 | 1000 | 3.5e-01 | 1.9e-02 | 0.338 |
| 2 | [1.0247375 0.81081343] | 1e+01 | 1000 | 2.9e-01 | 5.2e-03 | 0.305 |
| 3 | [0.95598843 0.88169493] | 1e+02 | 1000 | 5.2e-02 | 1.0e-02 | 0.253 |
| 4 | [0.94673281 0.89294715] | 1e+03 | 1000 | 5.6e-03 | 2.2e-03 | 0.292 |
| 5 | [0.94569677 0.89400545] | 1e+04 | 1000 | 5.7e-04 | 2.9e-04 | 0.259 |
| 6 | [0.94557152 0.89407169] | 1e+05 | 1000 | 5.7e-05 | 2.0e-02 | 0.250 |
| 7 | [0.94555891 0.89407834] | 1e+06 | 1000 | 5.5e-06 | 1.4e-01 | 0.235 |
| 8 | [0.94555764 0.89407898] | 1e+07 | 1000 | 3.7e-07 | 1.4e+00 | 0.244 |

**Tabela 5:** Resultados obtidos para o problema 1, método de penalidade, Flecther-Reeves para $x^0 = \{3, 2\}$

**Prob. 1 - Penalidade - BFGS**

| Iter | $P_{min}$ | r | # Passos | Conv_OCR | Conv_OSR | t(s) |
|---|---|---|---|---|---|---|
| 1 | [1.25174106 0.73042443] | 1e+00 | 6 | 3.5e-01 | 4.5e-08 | 0.021 |
| 2 | [1.02501318 0.81147628] | 1e+01 | 4 | 2.9e-01 | 5.3e-08 | 0.003 |
| 3 | [0.95576696 0.88122325] | 1e+02 | 1000 | 5.2e-02 | 8.7e-06 | 0.278 |
| 4 | [0.94663396 0.89276031] | 1e+03 | 4 | 5.6e-03 | 6.6e-07 | 0.001 |
| 5 | [0.94568843 0.8939897 ] | 1e+04 | 1000 | 5.7e-04 | 6.2e-06 | 0.244 |
| 6 | [0.94559353 0.89411343] | 1e+05 | 1000 | 5.7e-05 | 3.1e-03 | 0.259 |
| 7 | [0.94558406 0.89412585] | 1e+06 | 1000 | 5.7e-06 | 1.0e-03 | 0.276 |
| 8 | [0.94558308 0.89412702] | 1e+07 | 1000 | 5.7e-07 | 7.1e-03 | 0.271 |

**Tabela 6:** Resultados obtidos para o problema 1, método de penalidade, BFGS para $x^0 = \{3, 2\}$

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

Definição do ponto inicial no código principal:

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

**Principais resultados obtidos:**

A tabela resumo abaixo mostra que para essa função quadrática os métodos apresentaram resultados satisfatórios, com os métodos de Powell, Fletcher-Reeves, BFGS e Newton-Rapshon respeitando o número máximo de passos para convergência.

| Iter | $P_{min}$ | r | # Passos | conv_OCR | conv_OSR | Tempo |
|---|---|---|---|---|---|---|
| 1: | [1.25174114 0.7304246 ] | 1e+00 | 27 | 3.5e-01 | 9.6e-07 | 0.028 |
| 2: | [1.02501305 0.81147611] | 1e+01 | 1000 | 2.9e-01 | 3.7e-06 | 0.230 |
| 3: | [0.95576688 0.88122316] | 1e+02 | 1000 | 5.2e-02 | 3.5e-06 | 0.189 |
| 4: | [0.94659013 0.89267782] | 1e+03 | 1000 | 5.6e-03 | 1.6e-03 | 0.214 |
| 5: | [0.94526659 0.89319248] | 1e+04 | 1000 | 5.7e-04 | 1.5e-02 | 0.190 |
| 6: | [0.94513054 0.89323808] | 1e+05 | 1000 | 5.7e-05 | 1.5e-02 | 0.195 |
| 7: | [0.94511456 0.89323814] | 1e+06 | 1000 | 5.7e-06 | 3.3e-02 | 0.180 |
| 8: | [0.94511298 0.8932382 ] | 1e+07 | 1000 | 6.2e-07 | 3.0e-01 | 0.228 |

**Tabela 7:** Resumo dos resultados obtidos na questão 1a para $x^0 = \{2, 2\}^t$

A figura abaixo mostra, por método, o valor da função a cada iteração. A ideia é podermos comparar a rapidez com que cada método se aproxima do mínimo. Para essa função e ponto inicial, os métodos univariante e Powell foram os que mais demoraram para se aproximar, levando por volta de 6 passos, enquanto os demais precisaram de no máximo 3 passos.

As figuras abaixo representam as curvas de nível e o caminho de otimização percorrido por cada método. Como a função só possui um mínimo, todos os métodos convergiram o ponto correto (mínimo mais próximo).

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

Definição do ponto inicial no código principal:

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

**Principais resultados obtidos:**

Segue tabela resumo e figuras nos mesmos moldes do que foi apresentado para o outro ponto inicial. As conclusões são basicamente as mesmas.

| 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 8:** Resumo dos resultados obtidos na questão 1a para $x^0 = \{-1, -3\}^t$

## 3.2 Questão 1 (b)

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

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

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

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

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])
```

A tabela resumo abaixo mostra que para essa função, que não é quadrática, todos os métodos conseguiram convergir dentro do limite especificado para o número máximo de iterações.

Os métodos de Powell, Fletcher-Reeves e BFGS não convergiram dentro do número de passos esperado para funções quadráticas, o que era de certa forma esperado. Newton-Rapshson convergiu em 1 passo, mas foi o único que encontrou um ponto diferente dos demais. Ele convergiu para um ponto de sela imediatamente abaixo do ponto inicial, enquanto os demais métodos convergiram para o mínimo mais próximo, localizado à direita do ponto inicial. Esses detalhes são melhores vistos na figura com as curvas de nível localizada mais abaixo neste documento.

| **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 9:** Resumo dos resultados obtidos na questão 1b para $x^0 = \{10, 2\}^t$

A figura abaixo deixa claro como Newton-Raphson, apesar de ter convergido em 1 iteração, praticamente não minimizou a função como os demais métodos, por ter encontrado um ponto de sela. Flecther-Reeves foi o método que mais demorou para se aproximar do ponto mínimo, e o motivo ficará mais evidente mais abaixo quando eu apresentar o caminho percorrido por este método.

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.

Nenhum dos métodos levou ao ponto de mínimo à esquerda do ponto incicial.

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

Definição do ponto inicial no código principal:

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

Seguem resultados para o segundo ponto inicial proposto no enunciado. Pela posição deste ponto e da proximidade com o mínimo mais à esquerda da função, dessa vez, todos os métodos convergiram corretamente para o mínimo esperado. Em relação aos resultados não há novas observações relevantes a serem feitas, com exceção do fato de todos os métodos terem conseguido se aproximar muito rápido do mínimo.

| **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 | $\{7.00000007, -2.00000017\}^t$ |
| BFGS | 8 | 0.04286 | $\{7.00000021, -2.00000023\}^t$ |
| Newton-Raphson | 6 | 0.01332 | $\{7.00000001, -2.00000001\}^t$ |

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

### 3.2.3 Formulação

A seguinte formulação foi apresentada em sala de aula :

$A = (x_A, y_A) =$ Posição inicial do ponto A

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

$\Pi = U - V$ = Energia Potencial total

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

$V$ = trabalho das forças externas

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

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

$V = W v_A$

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

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
```

### 3.2.4 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 11:** 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 12:** Resumo dos resultados obtidos na questão 2a para $x^0 = \{0.01, -0.1\}^t$ e $\text{tol} = 10^{-4}$

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 13:** Resumo dos resultados obtidos na questão 2a para $x^0 = \{0.01, -0.1\}^t$ e 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 global, terceiro do segundo ciclo, gera uma direção com módulo muito pequeno e que que necessita de um $\alpha$(aproximadante 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 combimação dos parâmetros de entrada. Alterar o ponto inicial ajuda a resolver esse problema encontrado no método de Powell.

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

## 3.3 Questão 2 (b)

### 3.3.1 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...). A rigidez de cada mola ($k_i$, i =1,...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,...n-1) correspondem às cargas nodais equivalentes aos pesos das molas.

### 3.3.2 Formulação

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

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

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 $Li_m$ = comprimeto inicial da mola m, com m=1,2,...$n_{molas}$

Seja $Lf_m$ = comprimeto final da mola m, com m=1,2,...$n_{molas}$

Apenas para facilidade do equacionamento e analogia com a formulação para 2 molas, considerei que cada nó livre (total $n_{molas} - 1$ nós livres) terá deslocamentos positivos tanto em $x$ quanto em $y$. Ou seja, cada nó, após equilíbrio, estará à direita do seu ponto inicial e abaixo (considerando eixo $y$ positivo para baixo) do nó livre anterior.

Então:

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

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

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

$V_n = W_n v_n$ = Trabalho das forças externas

Finalmente :

$\Pi = \sum_{m=1}^{n_{molas}} U_m - \sum_{n=1}^{n_{nodes}} V_n$ = 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 conseguimos 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 de forma numérica 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 necessar 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 molas0.01, -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
```

### 3.3.3 Resultados

Para esse exercício, variei o número de molas no sistema de 2 a 20, e com isso precisei de diferentes pontos iniciais para as rodadas dos métodos. Por simplificação, usei os mesmos delocamentos iniciais para todos os nós livres. Valores utilizados : $u_i = -1$ e $v_i = 5$, $\forall i$ com $i = 1, 2, ...n_{nodes}$

Exemplo de definição do ponto inicial com a premissa acima para o caso 4 molas:

```
P0 = np.array([-1, 5, -1, 5, -1, 5])
```

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

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

O gráfico abaixo mostra o número de passos por método e por discretizaçao do número de molas. Nele é possível constatar que os métodos Univariante e Steepest Descent já não conseguem convergir para um sistema de 4 molas em diante. O método de Powell até converge para 4 molas, mas a partir de 6 molas já não consegue mais convergir. O método Fletcher-Reeves, na sensibilidade feita nesse trabalho comvergiu até 10 molas, e no caso com 20 molas não convergiu, apesar de ter chegado bem próximo dos resultados dos métodos que convergiram. Os métodos BFGS e Newton-Raphson convergiram para todos os casos, com um número relativamente baixo de passos.

As tabelas abaixo resumem o estudo de convergência do valor para o deslocamento final do Ponto $A$, representado pelo vetor $\{u_A, v_a\}^t$. Pode-se notar que os métodos que não convergem, apresentam resultados divergentes dos que convergiram, como esperado. Também é possível notar a diferença entre posição prevista para o ponto $A$ caso a discretização seja muito pequena, 2 molas por exemplo, e caso seja muito alta, 20 molas por exemplo, lembrando sempre de olhar o resultado do BFGS e Newton-Raphson para essa comparação, dado que foram os métodos que atingiram convergência para todas as discretizações.

Deslocamento do Ponto $A$ (nó central)

| **Método** | **# 2 molas** | **4 molas** | **6 molas** |
|---|---|---|---|
| Univariante | $\{-0.2051, 7.7890\}^t$ | $\{-0.0867, 7.1702\}^t$ | $\{-0.1927, 7.1189\}^t$ |
| Powell | $\{-0.2051, 7.7890\}^t$ | $\{-0.0863, 7.1700\}^t$ | $\{-0.3735, 6.8136\}^t$ |
| Steepest Descent | $\{-0.2051, 7.7890\}^t$ | $\{-0.0860, 7.1676\}^t$ | $\{-0.0725, 6.9551\}^t$ |
| Fletcher-Reeves | $\{-0.2051, 7.7890\}^t$ | $\{-0.0863, 7.1700\}^t$ | $\{-0.0692, 7.0765\}^t$ |
| BFGS | $\{-0.2051, 7.7890\}^t$ | $\{-0.0863, 7.1700\}^t$ | $\{-0.0692, 7.0765\}^t$ |
| Newton-Raphson | $\{-0.2051, 7.7890\}^t$ | $\{-0.0863, 7.1700\}^t$ | $\{-0.0692, 7.0765\}^t$ |

**Tabela 14:** Resumo dos resultados obtidos na questão 2b

Deslocamento do Ponto $A$ (nó central)

| **Método** | **8 molas** | **10 molas** | **20 molas** |
|---|---|---|---|
| Univariante | $\{-0.4678, 7.1400\}^t$ | $\{-0.6909, 7.2969\}^t$ | $\{-1.0657, 6.7105\}^t$ |
| Powell | $\{-0.5929, 7.5628\}^t$ | $\{-0.7793, 6.6867\}^t$ | $\{-1.0733, 6.7864\}^t$ |
| Steepest Descent | $\{-0.1084, 6.6119\}^t$ | $\{-0.1652, 6.2609\}^t$ | $\{-0.5561, 5.4032\}^t$ |
| Fletcher-Reeves | $\{-0.0635, 7.0450\}^t$ | $\{-0.0610, 7.0306\}^t$ | $\{-0.0575, 7.0123\}^t$ |
| BFGS | $\{-0.0635, 7.0450\}^t$ | $\{-0.0610, 7.0306\}^t$ | $\{-0.0575, 7.0116\}^t$ |
| Newton-Raphson | $\{-0.0635, 7.0450\}^t$ | $\{-0.0610, 7.0306\}^t$ | $\{-0.0575, 7.0116\}^t$ |

**Tabela 15:** Resumo dos resultados obtidos na questão 2b

A figura abaixo resume o tempo de execução de cada método para cada discretização do número de molas. Importante destacar o tempo maior para os métodos de Powell e Newton-Raphson. O tempo do método de Newton-Raphson acredito que possa ter influência do uso do pacote numdifftools para cálculo da Hessiana, e o tempo para cálculo de sua inversa. Porém, o método de Powell sofreu novamente, principalmente no caso 4 molas, com uma direção com módulo pequeno e consequente $\alpha$ grande do passo constante, **reforçando novamente a ideia de se normalizar as direções na busca unidirecional para evitar problemas desse tipo.**

A figura abaixo mostra o comportamento do valor da função a cada iteração dos métodos para o caso com 20 molas, evidenciando o quão rápido Flecther-Reeves, BFGS e Newton-Raphson se aproximam do mínimo.

Um detalhe que também aparece, mas está sutil e pode passar despercebido dada a escala em $y$ utilizada, é que o primeiro passo do Newton-Raphson está aumentando consideravelmente o valor da função. Após investigação do resultado, notei que esse primeiro passo está usando uma direção com módulo muito grande, da ordem de $5.10^9$ e retornando um $\alpha$ da seção áurea da ordem de $3.10^{-7}$. Como o passo usado é $10^{-2}$, é possível constatar que o $\Delta\alpha$ absoluto, ao invés de ficar próximo ao $10^{-2}$ desejado, atinge valores acima de 1, dadas as

ordens de grandeza dos parâmetros. Pode ser que o problema seja o cálculo da Hessiana nesse primeiro passo, apesar de que para os demais passos, o Newton-Raphson se comporta adequadamente. **Esse exemplo é mais um reforço para que seja usado direções normalizadas.**