Trabalho 1

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
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, e a realização de análise de convergência, para diferentes funções e pontos iniciais, dos seguintes métodos de otimização:

a. Univariante

b. Powell

c. Steepest Descent

d. Fletcher-Reeves

e. BFGS

f. Newton-Raphson

2 Implementações

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.

2.1 Pacotes utilizados

As seguintes bibliotecas são necessárias para execução do código final do arquivo principal :

```
import numpy as np
import matplotlib.pyplot as plt
from timeit import default_timer as timer
```

Os métodos de busca unidimensional, assim como os métodos de otimização foram implementados em arquivos distintos (osr_methods.py e line_search_methods.py). Com isso, no arquivo principal também é necessário realizar o import desses algoritmos.

```
import osr_methods as osr
import line_search_methods as lsm
```

Esses arquivos distintos necessitam apenas do pacote numpy. Com isso apenas o seguinte import é necessário nos dois arquivos citados acima.

```
import numpy as np
```

2.2 Busca Unidirecional

Os algoritmos 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
```

2.2.1 Passo Constante

Foi implementado um método que recebe como parâmetros de entrada um vetor direção ($\overrightarrow{dir}$), um ponto inicial ($\overrightarrow{P_1}$), a função que se deseja encontrar o mínimo ($f(\overrightarrow{P})$), 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 $f(\overrightarrow{P_1} - \epsilon\,\overrightarrow{dir})$ e $f(\overrightarrow{P_1} + \epsilon\,\overrightarrow{dir})$. Caso este último valor seja maior do que o primeiro, o sentido de busca considerado é o oposto do vetor direção ($-\overrightarrow{dir}$). Caso o primeiro seja o maior valor, o sentido do vetor direção é mantido na busca ($\overrightarrow{dir}$).

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 $f(\overrightarrow{P_1} + \alpha\,\overrightarrow{dir})$ 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 $f(\overrightarrow{P} - \epsilon\,\hat{dir})$ e $f(\overrightarrow{P})$, onde $\overrightarrow{P} = \overrightarrow{P_1} + \alpha\,\overrightarrow{dir}$. 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.

a. $f(\overrightarrow{P} - \epsilon\,\hat{dir})$ e $f(\overrightarrow{P})$, com $\overrightarrow{P} = \overrightarrow{P_1} + \alpha\,\overrightarrow{dir}$

b. $f(\overrightarrow{P_1} + \alpha\,\overrightarrow{dir}) < f(\overrightarrow{P_1} + (\alpha + \Delta\alpha)\,\overrightarrow{dir})$

```python
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
```

2.2.2 Seção Áurea

Implementado um método que recebe como parâmetros de entrada um intervalo de busca ($\overrightarrow{interv} = [\alpha^L, \alpha^U]$), um vetor direção de busca ($\overrightarrow{dir}$), um ponto inicial ($\overrightarrow{P_1}$), a função f ($f(\overrightarrow{P})$) 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 $f(\overrightarrow{P_1} + \alpha_E\,\overrightarrow{dir})$ e $f(\overrightarrow{P_1} + \alpha_D\,\overrightarrow{dir})$, 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 algortimo, quando a convergência foi obtida

Importante destacar que o algortimo 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
```

2.3 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
```

2.3.1 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
```

2.3.2 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 $\overrightarrow{P_n} - \overrightarrow{P_0}$, onde $\overrightarrow{P_n}$ é o ponto encontrado na otimização até o passo n do ciclo atual e $\overrightarrow{P_0}$ 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.

A implementação considerou como parâmetros de entrada o ponto mínimo encontrado no passo anterior ($\overrightarrow{P}$), o ponto inicial do ciclo atual ($\overrightarrow{P}_1$), 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 chamado 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
```

2.3.3 Steepest Descent

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

A implementação considerou como parâmetro de entrada apenas o vetor gradiente da função no ponto $\overrightarrow{P}$, 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
```

2.3.4 Fletcher-Reeves

O método Fletcher-Reeves utiliza $-\overrightarrow{\nabla} f$ no ponto $\overrightarrow{P_0}$ 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, $\beta^k = \frac{(|\overrightarrow{\nabla} f(\overrightarrow{P}^{k+1})|)^2}{(|\overrightarrow{\nabla} f(\overrightarrow{P}^{k})|)^2}$. 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, $\overrightarrow{d}^{k+1} = -\overrightarrow{\nabla} f(\overrightarrow{P}^{k+1}) + \beta^k \overrightarrow{d^k}$

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
```

2.3.5 BFGS

O método BFGS retorna $\overrightarrow{d}^k = -\bar{\bar{S}}^k \overrightarrow{\nabla} f(\overrightarrow{P}^k)$ como direção, sendo $\bar{\bar{S}}^0 = \bar{\bar{I}}$. É capaz de minimizar funções quadráticas em n + 1 passos, sendo n o número de dimensões.

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

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

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

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 $\bar{\bar{S}}$ calculada no último passo, o número do passo e o número de dimensões. O algortimo 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 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
```

2.3.6 Newton-Raphson

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

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 caclula a matriz Hessiana de f.

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

2.4 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 funcao  - apenas para automatizacao do plot de contorno
#1 = Questao 1 letra a
#2 = Questao 1 letra b
#3 = Questao 2 letra a
#4 = Questao 2 lebra b
func = ...

# nome da questao e letra - apenas para automatizcao 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 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()
```

Loop para chamada dos métodos e convergência da otimização.

```
start = timer()

while (norm_grad > tol_conv):
    if (passos == maxiter):
        print('Nao 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 dos mapas de contorno 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()
```

2.5 Exercício 2

Utilizando os métodos implementados na questão anterior, testar a sua implementação encontrando o ponto mínimo das seguintes funções:

a. Função 1:

$$f(x_1, x_2) = x_1^2 - 3x_1x_2 + 4x_2^2 + x_1 - x_2 \;\; com \; \overrightarrow{P_1} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \;\; e \;\; \overrightarrow{d} = \begin{bmatrix} -1 \\ -2 \end{bmatrix}$$

b. Função 2 - McCormick:

$$f(x_1, x_2) = \text{sen}\,(x_1 + x_2) + (x_1 - x_2)^2 - 1,5x_1 + 2,5x_2 \;\; com \; \overrightarrow{P_1} = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \;\; e \;\; \overrightarrow{d} = \begin{bmatrix} 1.453 \\ -4.547 \end{bmatrix}$$

c. Função 3 - Himmelblau:

$$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \;\; com \; \overrightarrow{P_1} = \begin{bmatrix} 0 \\ 5 \end{bmatrix} \;\; e \;\; \overrightarrow{d} = \begin{bmatrix} 3 \\ 1.5 \end{bmatrix}$$

- Para cada função acima, desenhar(na mesma figura): as curvas de nível e o segmento de reta conectando o ponto inicial ao ponto de mínimo.
- Adotar uma tolerância de 10^{-5} para verificação da convergência numérica.

2.5.1 Função 1

Importação das bibliotecas, da implementação dos métodos de busca unidimensional do exercício anterior e definição da função do exercício:

```
import numpy as np
import matplotlib.pyplot as plt
import linear_search_methods as lsm

def f(P):
  # P = [x1, x2]
  return P[0]**2 - 3*P[0]*P[1] + 4*(P[1]**2) + P[0] - P[1]
```

Cálculo do ponto mínimo usando os 3 métodos do exercício 1:

- Passo constante
- Bisseção
- Seção Áurea

```
#inputs de Ponto inicial e direcao de busca
P1 = np.array([1, 2])
dir = np.array([-1, -2])

#chama o metodo do passo constante
q2_a_pss_ct = lsm.passo_cte(dir.copy(), P1, f)

#funcao lsm.passo_cte entrega o intervalo de busca na segunda posicao do array de retorno
intervalo = q2_a_pss_ct[1]

#resgata o sentido unitario correto da busca unidimensional
sentido_busca = q2_a_pss_ct[2]

#chama o o metodo da bissecao
q2_a_bssc = lsm.bissecao(intervalo.copy(), sentido_busca.copy(), P1, f)

#chama o metodo da secao aurea
q2_a_sc_ar = lsm.secao_aurea(intervalo.copy(), sentido_busca.copy(), P1, f)

print(f'Passo constante : |\u03B1 min| = {intervalo[0]:.10f} e P min = ({q2_a_pss_ct[0][0]:.
                                        10f}, {q2_a_pss_ct[0][1]:.10f}) ')
print(f'Bissecao        : |\u03B1 min| = {q2_a_bssc[1]:.10f} e P min = ({q2_a_bssc[0][0]:.
                                        10f}, {q2_a_bssc[0][1]:.10f}) ')
print(f'Secao Aurea     : |\u03B1 min| = {q2_a_sc_ar[1]:.10f} e P min = ({q2_a_sc_ar[0][0]:.
                                        10f}, {q2_a_sc_ar[0][1]:.10f}) ')
```

Resultados obtidos (Saída do terminal):

Passo constante : $|\alpha\ min| = 2.1300000000$ e P min = (0.0474350416, 0.0948700832)
Bisseção : $|\alpha\ min| = 2.1344287109$ e P min = (0.0454544618, 0.0909089237)
Seção Áurea : $|\alpha\ min| = 2.1344280564$ e P min = (0.0454547546, 0.0909095092)

A solução para a função $f(x_1, x_2) = x_1^2 - 3x_1x_2 + 4x_2^2 + x_1 - x_2$ então fica :

- Passo constante: $|\alpha_{min}| = 2.1300000000$ e $\overrightarrow{P_{min}} = \begin{bmatrix} 0.0474350416 \\ 0.0948700832 \end{bmatrix}$
- Bisseção: $|\alpha_{min}| = 2.1344287109$ e $\overrightarrow{P_{min}} = \begin{bmatrix} 0.0454544618 \\ 0.0909089237 \end{bmatrix}$
- Seção Áurea: $|\alpha_{min}| = 2.1344280564$ e $\overrightarrow{P_{min}} = \begin{bmatrix} 0.0454547546 \\ 0.0909095092 \end{bmatrix}$

Como todos os métodos entregam respostas de P_{min} muito parecidas, com diferenças relativamente pequenas, para a etapa de desenhar as curvas de nível e o segmento de reta conectando P_1 ao P_{min} tomei a liberdade de plotar apenas o resultado da Seção Áurea.

```
#Escolhido o Pmin gerado pelo metodo da secao aurea para representar graficamente
Pmin = q2_a_sc_ar[0]

x1 = np.linspace(-7.5, 7.5, 100)
x2 = np.linspace(-7.5, 7.5, 100)
X1, X2 = np.meshgrid(x1, x2)
x3 = f([X1, X2])
niveis = plt.contour(X1, X2, x3, [0, 3, 10, 25, 40, 60, 100, 150], colors='black')
plt.clabel(niveis, inline=1, fontsize=10)
plt.annotate('', xy=Pmin, xytext=P1,
             arrowprops=dict(width=1, color='green', headwidth=10, headlength=10, shrink=
                                                                    0.05), fontsize='10')
plt.annotate(f'({P1[0]}, {P1[1]})', xy=P1, xytext=(5,-10), textcoords='offset points', color
                                        ='green')
plt.annotate(f'$P_{{min}}$=({round(Pmin[0], 7)}, {round(Pmin[1], 7)})', xy=Pmin, xytext=(0.
                                        03,0.95), textcoords='axes fraction', color='
                                        green')
plt.plot(P1[0], P1[1], marker="o", markersize=7, markeredgecolor="green", markerfacecolor="
                                        green")
plt.plot(Pmin[0], Pmin[1], marker="o", markersize=7, markeredgecolor="green",
                                        markerfacecolor="green")
plt.xlabel('$x_1$', fontsize='16')
plt.ylabel('$x_2$', fontsize='16')
```

```
plt.grid(linestyle='--')
plt.title("$f(x_1, x_2)$ - Metodo da Secao Aurea", fontsize='16')
plt.savefig("A_solution.pdf", format="pdf")
plt.show()
```

Figura 3: Função 1 - Seção Áurea

2.5.2 Função 2: McCormick

Importação das bibliotecas, da implementação dos métodos de busca unidimensional do exercício anterior e definição da função do exercício:

```
import numpy as np
import matplotlib.pyplot as plt
import linear_search_methods as lsm

def mcCormick(P):
  # P = [x1, x2]
  return np.sin(P[0] + P[1]) + (P[0] - P[1])**2 - 1.5*P[0] + 2.5*P[1]
```

Cálculo do ponto mínimo usando os 3 métodos do exercício 1:

- Passo constante
- Bisseção
- Seção Áurea

```
#inputs de Ponto inicial e direcao de busca
P1 = np.array([-2, 3])
dir= np.array([1.453, -4.547])

#chama o metodo do passo constante
q2_b_pss_ct = lsm.passo_cte(dir.copy(), P1, mcCormick)

#funcao lsm.passo_cte entrega o intervalo de busca na segunda posicao do array de retorno
intervalo = q2_b_pss_ct[1]

#resgata o sentido unitario correto da busca unidimensional
sentido_busca = q2_b_pss_ct[2]

#chama o o metodo da bissecao
q2_b_bssc = lsm.bissecao(intervalo.copy(), sentido_busca.copy(), P1, mcCormick)

#chama o metodo da secao aurea
q2_b_sc_ar = lsm.secao_aurea(intervalo.copy(), sentido_busca.copy(), P1, mcCormick)

print(f'Passo constante : |\u03B1| min = {intervalo[0]:.10f} e P min = ({q2_b_pss_ct[0][0]:.
                                        10f}, {q2_b_pss_ct[0][1]:.10f}) ')
print(f'Bissecao        : |\u03B1| min = {q2_b_bssc[1]:.10f} e P min = ({q2_b_bssc[0][0]:.
                                        10f}, {q2_b_bssc[0][1]:.10f}) ')
print(f'Secao Aurea     : |\u03B1| min = {q2_b_sc_ar[1]:.10f} e P min = ({q2_b_sc_ar[0][0]:.
                                        10f}, {q2_b_sc_ar[0][1]:.10f}) ')
```

Resultados obtidos (Saída do terminal):

Passo constante : $|\alpha\ min| = 4.7700000000$ e P min = (-0.5480690486, -1.5436545327)
Bisseção : $|\alpha\ min| = 4.7735791016$ e P min = (-0.5469796129, -1.5470637991)
Seção Áurea : $|\alpha\ min| = 4.7735766069$ e P min = (-0.5469803722, -1.5470614229)

A solução para a função $f(x_1, x_2) = \text{sen}\,(x_1 + x_2) + (x_1 - x_2)^2 - 1,5x_1 + 2,5x_2$ então fica :

- Passo constante: $|\alpha_{min}| = 4.7700000000$ e $\overrightarrow{P_{min}} = \begin{bmatrix} -0.5480690486 \\ -1.5436545327 \end{bmatrix}$
- Bisseção: $|\alpha_{min}| = 4.77357910169$ e $\overrightarrow{P_{min}} = \begin{bmatrix} -0.5469796129 \\ -1.5470637991 \end{bmatrix}$

- Seção Áurea: $|\alpha_{min}| = 4.7735766069$ e $\overrightarrow{P_{min}} = \begin{bmatrix} -0.5469803722 \\ -1.5470614229 \end{bmatrix}$

Como todos os métodos entregam respostas de P_{min} muito parecidas, com diferenças relativamente pequenas, para a etapa de desenhar as curvas de nível e o segmento de reta conectando P_1 ao P_{min} tomei a liberdade de plotar apenas o resultado do Passo Constante.

```
#Escolhido o Pmin gerado pelo metodo do passo constante para representar graficamente
Pmin = q2_b_pss_ct[0]

x1 = np.linspace(-7.5, 7.5, 100)
x2 = np.linspace(-7.5, 7.5, 100)
X1, X2 = np.meshgrid(x1, x2)
x3 = mcCormick([X1, X2])
niveis = plt.contour(X1, X2, x3, [0, 5, 15, 40, 60], colors='black')
plt.clabel(niveis, inline=1, fontsize=10)
plt.annotate('', xy=Pmin, xytext=P1,
                arrowprops=dict(width=1, color='green', headwidth=10, headlength=10, shrink=
                                                                    0.05), fontsize='10')
plt.annotate(f'({P1[0]}, {P1[1]})', xy=P1, xytext=(10,0), textcoords='offset points', color=
                                                    'green')
plt.annotate(f'$P_{{min}}$=({round(Pmin[0], 7)}, {round(Pmin[1], 7)})', xy=Pmin, xytext=(0.
                                                    03, 0.95), textcoords='axes fraction', color=
                                                    'green')
plt.plot(P1[0], P1[1], marker="o", markersize=7, markeredgecolor="green", markerfacecolor="
                                                    green")
plt.plot(Pmin[0], Pmin[1], marker="o", markersize=7, markeredgecolor="green",
                                                    markerfacecolor="green")
plt.xlabel('$x_1$', fontsize='16')
plt.ylabel('$x_2$', fontsize='16')
plt.grid(linestyle='--')
plt.title("$mcCormick(x_1, x_2)$ - Metodo do Passo Constante", fontsize='16')
plt.savefig("B_solution.pdf", format="pdf")
plt.show()
```

Figura 4: McCormick - Passo Constante

2.5.3 Função 3: Himmelblau

Importação das bibliotecas, da implementação dos métodos de busca unidimensional do exercício anterior e definição da função do exercício:

```
import numpy as np
import matplotlib.pyplot as plt
import linear_search_methods as lsm

def himmelblau(P):
  # P = [x1, x2]
  return (P[0]**2 + P[1] - 11)**2 + (P[0] + P[1]**2 - 7)**2
```

Cálculo do ponto mínimo usando os 3 métodos do exercício 1:

- Passo constante
- Bisseção
- Seção Áurea

```
#inputs de Ponto inicial e direcao de busca
P1 = np.array([0, 5])
dir = np.array([3, 1.5])

#chama o metodo do passo constante
q2_c_pss_ct = lsm.passo_cte(dir, P1, himmelblau)

#funcao lsm.passo_cte entrega o intervalo de busca na segunda posicao do array de retorno
intervalo = q2_c_pss_ct[1]

#resgata o sentido unitario correto da busca unidimensional
```

```
sentido_busca = q2_c_pss_ct[2]

#chama o o metodo da bissecao
q2_c_bssc = lsm.bissecao(intervalo.copy(), sentido_busca.copy(), P1, himmelblau)

#chama o metodo da secao aurea
q2_c_sc_ar = lsm.secao_aurea(intervalo.copy(), sentido_busca.copy(), P1, himmelblau)

print(f'Passo constante : |\u03B1 min| = {intervalo[0]:.10f} e P min = ({q2_c_pss_ct[0][0]:.
                                        10f}, {q2_c_pss_ct[0][1]:.10f}) ')
print(f'Bissecao        : |\u03B1 min| = {q2_c_bssc[1]:.10f} e P min = ({q2_c_bssc[0][0]:.
                                        10f}, {q2_c_bssc[0][1]:.10f}) ')
print(f'Secao Aurea     : |\u03B1 min| = {q2_c_sc_ar[1]:.10f} e P min = ({q2_c_sc_ar[0][0]:.
                                        10f}, {q2_c_sc_ar[0][1]:.10f}) ')
```

Resultados obtidos (Saída do terminal):

Passo constante : $|\alpha\ min| = 3.3900000000$ e P min = (-3.0321081775, 3.4839459113)
Bisseção : $|\alpha\ min| = 3.3921630859$ e P min = (-3.0340429004, 3.4829785498)
Seção Áurea : $|\alpha\ min| = 3.3921633455$ e P min = (-3.0340431325, 3.4829784337)

A solução para a função $f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$ então fica :

- Passo constante: $|\alpha_{min}| = 3.3900000000$ e $\overrightarrow{P_{min}} = \begin{bmatrix} -3.0321081775 \\ 3.4839459113 \end{bmatrix}$
- Bisseção: $|\alpha_{min}| = 3.3921630859$ e $\overrightarrow{P_{min}} = \begin{bmatrix} -3.0340429004 \\ 3.4829785498 \end{bmatrix}$
- Seção Áurea: $|\alpha_{min}| = 3.3921633455$ e $\overrightarrow{P_{min}} = \begin{bmatrix} -3.0340431325 \\ 3.4829784337 \end{bmatrix}$

Como todos os métodos entregam respostas de P_{min} muito parecidas, com diferenças relativamente pequenas, para a etapa de desenhar as curvas de nível e o segmento de reta conectando P_1 ao P_{min} tomei a liberdade de plotar apenas o resultado da Bisseção.

```
#Escolhido o Pmin gerado pelo metodo da bissecao para representar graficamente
Pmin = q2_c_bssc[0]

x1 = np.linspace(-7.5, 7.5, 1000)
x2 = np.linspace(-7.5, 7.5, 1000)
X1, X2 = np.meshgrid(x1, x2)
x3 = himmelblau([X1, X2])
niveis = plt.contour(X1, X2, x3, [10,50,100,200, 350, 1000], colors='black')
plt.clabel(niveis, inline=1, fontsize=10)
plt.annotate('', xy=Pmin, xytext=P1,
             arrowprops=dict(width=1, color='green', headwidth=10, headlength=10, shrink=
                                                                    0.05), fontsize='10')
plt.annotate(f'$P_1$=({P1[0]}, {P1[1]})', xy=P1, xytext=(0.03,0.95), textcoords='axes
                                        fraction', color='green')
plt.annotate(f'$P_{{min}}$=({round(Pmin[0], 7)}, {round(Pmin[1], 7)})', xy=Pmin, xytext=(0.
                                        55,0.95), textcoords='axes fraction', color='
                                        green')
plt.plot(P1[0], P1[1], marker="o", markersize=7, markeredgecolor="green", markerfacecolor="
                                        green")
plt.plot(Pmin[0], Pmin[1], marker="o", markersize=7, markeredgecolor="green",
                                        markerfacecolor="green")
plt.xlabel('$x_1$', fontsize='16')
plt.ylabel('$x_2$', fontsize='16')
plt.grid(linestyle='--')
plt.title("$himmelblau(x_1, x_2)$ - Metodo da Bissecao", fontsize='16')
plt.savefig("C_solution.pdf", format="pdf")
plt.show()
```

Figura 5: Himmelblau - Bisseção