Tarefa 03 - MO431

Patrick de Carvalho Tavares Rezende Ferreira - 175480

In [16]:

```
import time

import pybobyqa
from scipy.optimize import minimize, line_search
from numpy import array
```

Abaixo definimos a função objetivo a ser minimizada (função de himmelblau) e seu gradiente. A função recebe o valr do ponto (x, y) a ser avaliado e retorna um float com o valor real da função naquele ponto, enquanto que o gradiente também recebe um ponto onde será avaliado e retorna um vetor que é o próprio gradiente da função naquele ponto.

In [17]:

```
def himmelblau(x):
    # Garantindo que trabalhamos com array numpy, e nao uma lista
    x = array(x)

    return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2


def grad_himmelblau(x, *args):
    # Para x_0
    dx_0 = 2 * (2 * x[0]*(x[0] ** 2 + x[1] - 11) + x[0] + x[1] ** 2 - 7)

    # Para x_1
    dx_1 = 2 * (x[0] ** 2 + 2 * x[1] * (x[0] + x[1] ** 2 - 7) + x[1] - 11)

    return array([dx_0, dx_1])
```

Abaixo executamos as 5 técnicas de minimização solicitadas pelo roteiro e, em seguida, ocorre a discussão acerca dos resultados.

In [18]:

```
# =====GRADIENTE-CONJUGADO=====================================================
x = array([4, 4])
t0 = time.time()
val = minimize(himmelblau, x, method="CG", jac=None)
tf = time.time()
print("\nGRADIENTE-CONJUGADO")
print("\niterações: ", val.nit)
print("chamadas do gradiente: ", val.nfev)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n")

# ======BFGS-SEM-GRADIENTE-PASSADO==========================================
x = array([4, 4])
t0 = time.time()
val = minimize(himmelblau, x, method="L-BFGS-B", jac=None)
tf = time.time()
print("\nL-BFGS-B-sem-grad")
print("\niterações: ", val.nit)
print("chamadas do gradiente: ", val.nfev)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n")

# ======BFGS-COM-GRADIENTE-PASSADO==========================================
x = array([4, 4])
t0 = time.time()
val = minimize(himmelblau, x, method="L-BFGS-B", jac=grad_himmelblau)
tf = time.time()
print("\nL-BFGS-B-com-grad")
print("\niterações: ", val.nit)
print("chamadas do gradiente: ", val.nfev)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n")

# =========NELDER-MEAN=========================================================
x = array([4, 4])
t0 = time.time()
val = minimize(himmelblau, x, method="Nelder-Mead", options={'initial_simplex':a
rray([[-4, -4], [-4, 1], [4, -1]])})
tf = time.time()
print("\nNelder-mead")
print("\niterações: ", val.nit)
print("avaliações dos vértices do triângulo: ", val.nfev)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n")

# =======LINE-SEARCH==========================================================
x = array([4, 4])
x_new = x.copy()
pk = array([-1, -1]) # -grad_himmelblau(x_new)

t0 = time.time()
```

```
while(1):
    alpha, fc, gc, new_fval, old_fval, new_slope = line_search(himmelblau, grad_
himmelblau, x_new, pk=pk)

    # Se ainda nao convergiu, continue iterando
    if alpha is not None:
        x_new = x_new + alpha * pk
    else:
        # Quando convergir, sai do loop
        break
tf = time.time()
print("\nLine-Search começando na direção [-1,-1]")
print("\niterações: ", fc)
print("chamadas do gradiente: ", gc)
print("x: ", x_new)
print("himmelblau(x): ", old_fval)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n")


x = array([4, 4])
x_new = x.copy()
pk = -grad_himmelblau(x_new)

t0 = time.time()
while(1):
    alpha, fc, gc, new_fval, old_fval, new_slope = line_search(himmelblau, grad_
himmelblau, x_new, pk=pk)

    # Se ainda nao convergiu, continue iterando
    if alpha is not None:
        x_new = x_new + alpha * pk
    else:
        # Quando convergir, sai do loop
        break
tf = time.time()
print("\nLine-Search começando na direção oposta ao gradiente")
print("\niterações: ", fc)
print("chamadas do gradiente: ", gc)
print("x: ", x_new)
print("himmelblau(x): ", old_fval)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n")

# ==========BOBYQA=========================================================
x = array([4, 4])
print("\nBOBYQA")
# Estabelecendo os limites (lower <= x <= upper)
lower = array([-10.0, -10.0])
upper = array([10.0, 10.0])
# Executa a minimizacao
t0 = time.time()
val = pybobyqa.solve(himmelblau, x, bounds=(lower, upper))
tf = time.time()

# Imprime resultados
print(val)
print("Tempo demandado pela otimização [s]: ", tf - t0)
```

```
GRADIENTE-CONJUGADO

iterações:  8
chamadas do gradiente:  68
x:  [2.99999999 2.        ]
himmelblau(x):  1.9486292751344026e-15
Tempo demandado pela otimização [s]:  0.0020203590393066406



L-BFGS-B-sem-grad

iterações:  9
chamadas do gradiente:  30
x:  [2.99999985 2.00000019]
himmelblau(x):  8.502778926721376e-13
Tempo demandado pela otimização [s]:  0.0015976428985595703



L-BFGS-B-com-grad

iterações:  9
chamadas do gradiente:  10
x:  [2.99999986 2.00000019]
himmelblau(x):  8.287611020495648e-13
Tempo demandado pela otimização [s]:  0.0007684230804443359



Nelder-mead

iterações:  40
avaliações dos vértices do triângulo:  77
x:  [ 3.58441449 -1.84811588]
himmelblau(x):  1.0686566996168641e-08
Tempo demandado pela otimização [s]:  0.0035278797149658203



Line-Search começando na direção [-1,-1]

iterações:  13
chamadas do gradiente:  0
x:  [2.5 2.5]
himmelblau(x):  8.125
Tempo demandado pela otimização [s]:  0.0013186931610107422



Line-Search começando na direção oposta ao gradiente

iterações:  13
chamadas do gradiente:  0
x:  [-2.36228496 -4.45809648]
himmelblau(x):  208.07835720768088
Tempo demandado pela otimização [s]:  0.001665353775024414



BOBYQA
```

```
****** Py-BOBYQA Results ******
Solution xmin = [3. 2.]
Objective value f(xmin) = 1.287703555e-21
Needed 58 objective evaluations (at 58 points)
Approximate gradient = [6.40527549e-09 5.63832659e-08]
Approximate Hessian = [[73.79866528 20.17722326]
 [20.17722326 34.16328251]]
Exit flag = 0
Success: rho has reached rhoend
******************************

Tempo demandado pela otimização [s]:  0.10146498680114746
```

Discussão dos Resultados

As saídas das 5 técnicas de otimização (Conjugado do gradiente, busca em linha, Nelder Mead, BFGS, BOBYQA) para a função de Himmelblau são apresentadas acima.

A menor quantidade de steps (iterações) necessários à otimização foi para a técnica de gradiente conjugado, como era de se esperar, já que é uma técnica de passo grande e, se for aplicada adequadamente, converge de maneira rápida. Embora tenha feito menos iterações que o BFGS, seu tempo ainda foi pior, devido às diversas chamadas do gradiente, sendo de 0.005189180374145508s contra aproximadamente 0.00168s para ambos os BFGS.

O segundo método que precisou de menos iterações foi o BFGS, que é um método quadrático e realizou menos consultas ao gradiente que o próprio gradiente conjugado, sendo de 30 para o gradiente nao passado (cáculo implícito na função) e 10 para o gradiente passado analiticamente. Além disso, o valor de mínimo local encontrado foi menor que o do gradiente conjugado, o que indica que a aproximação por uma função quadrática naquele local parece ter sido uma adequada abordagem.

O line-search fez 13 iterações, mas nenhuma consulta ao gradiente, tendosido mais rápido que o gradiente conjugado para encontrar o mínimo local, com 0.002579212188720703s. Entretando, a qualidade do mínimo encontrada por este método é inferior, sendo de valor 8.125 quando iniciamos a busca na direção (-1, -1) e de 208.07835720768088 se começarmos na direção em que o gradiente aponta na posição inicial (não convergindo o algoritmo).

Enquanto que BFGS e Graidiente conjugado convergiram para o mínimo em [x=2.99999985, y=2.00000019], o método de Nelder-Mead encontrou um mínimo em [x=3.58441449, y=-1.84811588] com valor de 1.0686566996168641e-08. Com muitas iterações, cálculos dos vértices do triângulo e um tempo de execução de 0.007497310638427734s, fica claro que este método é recomendável principalmente para casos em que o cálculo do gradiente seja altamente custoso por algum motivo.

Com 58 avaliações da função objetivo e um tempo de execução de 0.10592770576477051s (o mais lento deste experimento), o método NEWOA ou BOBYQA foi o que encontrou o melhor mínimo local, sendo de 1.287703555e-21, no ponto [x=3. y=2.] e sem a necessidade de cálculo do gradiente. Isto indica que este método é uma boa alternativa a funções com um gradiente custoso computacionalmente e que produz bons resultados, embora deva-se levar em consideração que seu tempo de execução é alto.