Tarefa 03 - MO431

Patrick de Carvalho Tavares Rezende Ferreira - 175480

In [23]:

```
import time

import pybobyqa
from scipy.optimize import minimize, line_search
from numpy import array
from numpy.random import seed

seed(1234)
```

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 [24]:

```
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 [25]:

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

# =====GRADIENTE-CONJUGADO-com-passagem-de-gradiente=======================
x = array([4, 4])
t0 = time.time()
val = minimize(himmelblau, x, method="CG", jac=grad_himmelblau)
tf = time.time()
print("\nGRADIENTE-CONJUGADO com gradiente analítico")
print("\nquantidade de chamadas da função objetivo: ", val.nfev)
print("chamadas do gradiente: ", val.njev)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n-------------------------------------------------\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("\nquantidade de chamadas da função objetivo: ", val.nfev)
print("chamadas do gradiente: ", val.nit)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n-------------------------------------------------\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("\nquantidade de chamadas da função objetivo: ", val.nfev)
print("chamadas do gradiente: ", val.nit)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n-------------------------------------------------\n")

# ==========NELDER-MEAN=====================================================
x = array([4, 4])
t0 = time.time()
val = minimize(himmelblau, x, method="Nelder-Mead", options={'.nfevial_simplex':
array([[-4, -4], [-4, 1], [4, -1]])})
tf = time.time()
print("\nNelder-mead")
```

```
print("\nquantidade de chamadas da função objetivo: ", val.nfev)
print("avaliações dos vértices do triângulo: ", val.nit)
print("x: ", val.x)
print("himmelblau(x): ", val.fun)
print("Tempo demandado pela otimização [s]: ", tf - t0)
print("\n-------------------------------------------------\n")

# =======LINE-SEARCH============================================================
x = array([4, 4])
x_new = x.copy()
pk = -grad_himmelblau(x_new)
gc_counter = 0
fc_counter = 0

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

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

```
GRADIENTE-CONJUGADO sem passagem de gradiente

quantidade de chamadas da função objetivo:  68
chamadas do gradiente:  17
x:  [2.99999999 2.        ]
himmelblau(x):  1.9486292751344026e-15
Tempo demandado pela otimização [s]:  0.0019459724426269531

------------------------------------------------


GRADIENTE-CONJUGADO com gradiente analítico

quantidade de chamadas da função objetivo:  17
chamadas do gradiente:  17
x:  [3. 2.]
himmelblau(x):  8.490750396096654e-21
Tempo demandado pela otimização [s]:  0.0025606155395507812

------------------------------------------------


L-BFGS-B-sem-grad

quantidade de chamadas da função objetivo:  30
chamadas do gradiente:  9
x:  [2.99999985 2.00000019]
himmelblau(x):  8.502778926721376e-13
Tempo demandado pela otimização [s]:  0.0009605884552001953

------------------------------------------------


L-BFGS-B-com-grad

quantidade de chamadas da função objetivo:  10
chamadas do gradiente:  9
x:  [2.99999986 2.00000019]
himmelblau(x):  8.287611020495648e-13
Tempo demandado pela otimização [s]:  0.0010695457458496094

------------------------------------------------


Nelder-mead

quantidade de chamadas da função objetivo:  76
avaliações dos vértices do triângulo:  40
x:  [3.00002182 1.99995542]
himmelblau(x):  3.19442883153528e-08
Tempo demandado pela otimização [s]:  0.004319667816162109

------------------------------------------------


Line-Search começando na direção oposta ao gradiente

quantidade de chamadas da função objetivo:  208
chamadas do gradiente:  29
x:  [-3.77931025 -3.28318599]
himmelblau(x):  7.888609052210118e-31
```

```
Tempo demandado pela otimização [s]:  0.008239984512329102

------------------------------------------------


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]:  0.10332751274108887

------------------------------------------------


/home/patrick/anaconda3/lib/python3.7/site-packages/ipykernel_launch
er.py:56: OptimizeWarning: Unknown solver options: .nfevial_simplex
```

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 0.0025606155395507812 contra aproximadamente 0.0010s para ambos os BFGS.

O gradiente conjugado com passagem de gradiente analítico explícito não foi solicitado e foi calculado apenas a título de informação, tendo desempenhado menos consultas à função objetivo do que o método com passagem explícita, mas demorando aproximadamente o mesmo tempo. O mínimo encontrado, entretando, foi menor e calculado em outro ponto, como exibido acima. Neste relatório, quando cita-se gradiente conjugado, estaremos falando daquele sem passagem de gradiente explícita, conforme solictao no roteiro.

O segundo método que precisou de menos iterações foi o BFGS, que é um método quadrático e realizou menos consultas à função objetivo que o próprio gradiente conjugado, sendo de 30 para o gradiente nao passado (cáculo do gradiente 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. Tanto o método com passagem como o sem passagem de gradiente produziram resultados muito próximos de mínimo local e aproximadamente mesmo tempo computacional.

O line-search fez 208 chamadas à função objetivo e 29 ao gradiente, tendo sido mais lento que o gradiente conjugado para encontrar o mínimo local, com 0.008239984512329102. Entretando, a qualidade do mínimo encontrada por este método é superior, sendo de valor 7.888609052210118e-31 no ponto [x=-3.77931025, y=-3.28318599].

Enquanto que BFGS e Gradiente 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 0.10332751274108887s (o mais lento deste experimento), o método NEWOA ou BOBYQA foi o que encontrou o segundo 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.