MO431 Tarefa 3 Versão 1

Autor: Pedro Henrique Vaz Valois

RA: 265676

Abril 2021

In [ ]:
```
import time
import numpy as np
from scipy.optimize import minimize, line_search, minimize_scalar
import matplotlib.pyplot as plt
```

Minimização da função de Himmelblau 2D

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

$$\nabla f(x,y) = (x^2 + y - 11)\begin{pmatrix} 4x \\ 1 \end{pmatrix} + (x + y^2 - 7)\begin{pmatrix} 2 \\ 4y \end{pmatrix}$$

In [ ]:
```
class Function:

  def __init__(self):
    self.calls = 0
    self.grad_calls = 0
    self.t = None

  def __call__(self, w):
    self.calls += 1

    x, y = w
    a1 = (x * x + y - 11)
    a2 = (x + y * y - 7)
    return a1 * a1 + a2 * a2

  def grad(self, w):
    self.grad_calls += 1

    x, y = w

    b1 = x * x + y - 11
    b2 = x + y * y - 7

    df_dx = b1 * 4 * x + b2 * 2
    df_dy = b1 + b2 * 4 * y

    return np.array([df_dx, df_dy])

  def start(self):
    self.t = time.time()

  def stop(self):
    self.t = time.time() - self.t

  def print_results(self, x_min = None):
    if hasattr(self, 'result'):
      x_min = self.result.x
    f_min = self(x_min)
    print(f"""Resultados:
    ponto de mínimo = {x_min}
    função no mínimo = {f_min}
    chamadas à função = {self.calls}
    chamadas ao gradiente = {self.grad_calls}
    tempo de execução = {self.t * 1000:.2f}ms
    """)
```

In [ ]:
```
# hiperparâmetros
w0 = np.array([4, 4]) # ponto inicial
```

1 Conjugado gradiente

In [ ]:
```
f = Function()

f.start()
f.result = minimize(f, x0=w0, method='CG', jac=f.grad)
f.stop()

f.print_results()
```

```
Resultados:
    ponto de mínimo = [3.00000001 2.00000006]
    função no mínimo = 6.510705790815223e-14
    chamadas à função = 28
    chamadas ao gradiente = 27
    tempo de execução = 3.03ms
```

O método do Conjugado Gradiente convergiu para o muito próximo do mínimo [3, 2], com poucas chamadas às funções e em tempo bem pequeno.

2 Descida do gradiente com busca em linha

In [ ]:
```
f = Function()

f.start()

w_old = [0, 0]
w = w0
while abs(f(w) - f(w_old)) > 1e-5:
  alpha, *_ = line_search(f, f.grad, w, -f.grad(w))
  w_old = w
  w = w - alpha * f.grad(w)

f.stop()

f.print_results(w)
```

```
Resultados:
    ponto de mínimo = [-3.77926013 -3.28318417]
    função no mínimo = 1.435926405677847e-07
    chamadas à função = 66
    chamadas ao gradiente = 28
    tempo de execução = 2.75ms
```

A função `line_search` realiza apenas um passo da minimização. Assim, rodamos um loop para encontrar o mínimo.

Ref: https://en.wikipedia.org/wiki/Wolfe_conditions

Assim, o método de Descida do Gradiente com busca em linha convergiu para o muito próximo do mínimo [-3.78, -3.28], com uma quantidade superior de chamadas, mas em tempo inferior se comparado ao CG.

3 Nelder-Mead

In [ ]:
```
# triângulo inicial
t = [(-4, -4), (-4, 1), (4, 1)]

f = Function()

f.start()
f.result = minimize(f, x0=w0, method='Nelder-Mead', jac=f.grad, options={"initial_sim
f.stop()

f.print_results()
```

```
Resultados:
    ponto de mínimo = [-3.77932958 -3.28316504]
    função no mínimo = 5.251460418262908e-08
    chamadas à função = 77
    chamadas ao gradiente = 0
    tempo de execução = 4.58ms
```

```
/usr/local/lib/python3.7/dist-packages/scipy/optimize/_minimize.py:506: RuntimeWarnin
g: Method Nelder-Mead does not use gradient information (jac).
  RuntimeWarning)
```

O método Nelder-Mead também convergiu ao ponto [-3.78, -3.28], mas sem usar o gradiente, como informado pela mensagem de aviso do scipy. De modo geral, a execução foi rápida apesar de mais lenta do que as anteriores.

4 BFGS

4.1 BFGS com gradiente

In [ ]:
```
f = Function()

f.start()
f.result = minimize(f, x0=w0, method='BFGS', jac=f.grad)
f.stop()

f.print_results()
```

```
Resultados:
    ponto de mínimo = [3.00000001 1.9999998 ]
    função no mínimo = 6.746925461708176e-13
    chamadas à função = 19
    chamadas ao gradiente = 18
    tempo de execução = 3.85ms
```

O método BFGS com gradiente foi capaz de convergir ao ponto [3, 2] rapidamente e com o menor número de chamadas visto entre os métodos testados.

4.2 BFGS sem gradiente

In [ ]:
```
f = Function()

f.start()
f.result = minimize(f, x0=w0, method='BFGS', jac=None)
f.stop()

f.print_results()
```

```
Resultados:
    ponto de mínimo = [3.00000001 2.        ]
    função no mínimo = 5.246091205787869e-15
    chamadas à função = 65
    chamadas ao gradiente = 0
    tempo de execução = 3.81ms
```

O método BFGS sem gradiente foi capaz de convergir ao mesmo ponto [3, 2] do teste BFGS com gradiente, mas com número de chamadas e tempo superiores.

5 NEWOA ou BOBYQA

In [ ]:
```
!pip install Py-BOBYQA
```

```
Requirement already satisfied: Py-BOBYQA in /usr/local/lib/python3.7/dist-packages (1.
3)
Requirement already satisfied: scipy>=0.17 in /usr/local/lib/python3.7/dist-packages
(from Py-BOBYQA) (1.4.1)
Requirement already satisfied: pandas>=0.17 in /usr/local/lib/python3.7/dist-packages
(from Py-BOBYQA) (1.1.5)
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-packages
(from Py-BOBYQA) (1.19.5)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-packages
(from pandas>=0.17->Py-BOBYQA) (2018.9)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist
-packages (from pandas>=0.17->Py-BOBYQA) (2.8.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (fro
m python-dateutil>=2.7.3->pandas>=0.17->Py-BOBYQA) (1.15.0)
```

In [ ]:
```
import pybobyqa
```

In [ ]:
```
f = Function()

f.start()
f.result = pybobyqa.solve(f, w0)
f.stop()

f.print_results()
```

```
Resultados:
    ponto de mínimo = [3. 2.]
    função no mínimo = 1.287703554675167e-21
    chamadas à função = 59
    chamadas ao gradiente = 0
    tempo de execução = 111.66ms
```

O método BOBYQA foi o que obteve a melhor convergência (vide o valor da função no ponto), mas para isso teve tempo bastante superior a todos os outros métodos testados.