Optimización I. Parcial II

Y. Sarahi García Gozález

Librerías

In [ ]:
```python
import numpy as np
import functions as fn #libreria con las funciones Himmelblau,Beale,Rosenbrock,Hartman
from math import exp
```

In [ ]:
```python
#imprimimos el epsilon de la máquina
epsilon = np.finfo(float).eps
print("Epsilon de la máquina:", epsilon)
```

```
Epsilon de la máquina: 2.220446049250313e-16
```

In [ ]:
```python
def imprime(f,n,x0,metodo,metodo_args):

    '''
    Esta función imprime:
    - la dimensión $n$,
    - f(x_0)
    - el  número $k$ de iteraciones realizadas
    - f(x_k)
    - las primeras y últimas 4 entradas del punto $\mathbf{x}_k$ que devuelve el algoritmo,
    - la norma del gradiente g_k,
    - promedio de iteraciones del algoritmo 1
    - la variable $bres$ para saber si el algoritmo puedo converger.

    '''

    xk,fk,gk,k,indicador= metodo(*metodo_args)



    print('Dimensión n = ', n )
    print('f(x_0) = ', f(x0) )
    print('Número de iteraciones = ', k )
    print('f(x_k) = ', fk )
    print('Primeras cuatro entradas de x_k= ', xk[:4])
    print('Últimas cuatro entradas de x_k= ', xk[-4:])
    print('Norma del gradiente ||gk||   = ', np.linalg.norm(gk))



    if(indicador):
        print("Sí se cumplio el criterio de convergencia")
```

In [ ]:
```python
#definimos la funcion que genera la matriz A de acuerdo a la instrucción anterior
def genera_A1(n):
    return np.ones((n,n))+n*np.eye(n)
#definimos la funcion que genera la matriz A de acuerdo a la instrucción anterior
def genera_A2(n):
    A = np.empty([n,n], dtype=float)
    for i in range(n):
        for j in range(n):
            u=0.25*((i-j)**2)
            A[i][j]=exp(-1*u)
    return A
```

Ejercicio 1

1. Programar la función $Q(x;\mu)$ y su gradiente

$$\nabla Q(\mathbf{x};\mu) = \nabla f(\mathbf{x}) + \mu c_1(\mathbf{x})\nabla c_1(\mathbf{x}).$$

In [ ]:
```python
def c1(x):
    return x[0]**2+x[1]**2-4

def Dc1(x):
    dc1=2*x[0]
    dc2=2*x[1]
    return np.array(dc1,dc2)
```

2. Programar el método de penalización cuadrática usando el método BFGS modificado:

a) Dar la función $f(\mathbf{x})$, $c_1(\mathbf{x})$, la función $Q(\mathbf{x};\mu)$, su gradiente $\nabla Q(\mathbf{x};\mu)$, un punto inicial $\mathbf{x}_0$, μ_0, una tolerancia $\tau > 0$, el número máximo de iteraciones N, y los parámetros que se necesiten para usar el método BFGS modificado.

b) Para $k = 0, 1, \ldots, N$ repetir los siguientes pasos:

b1) Definir $\tau_k = \left(1 + \frac{10N}{10k+1}\right)\tau$

b2) Calcular el punto $\mathbf{x}_{k+1}$ como el minimizador de $Q(\mathbf{x};\mu_k)$ con el método BFGS modificado usando como punto inicial a $\mathbf{x}_k$ y la tolerancia τ_k.

b3) Imprimir el punto $\mathbf{x}_{k+1}$, $f(\mathbf{x}_{k+1})$, $Q(\mathbf{x};\mu_k)$, el número de iteraciones realizó el algoritmo BFGS y el valor $c_1(\mathbf{x}_{k+1})$.

b4) Si $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| < \tau$, terminar devolviendo $\mathbf{x}_{k+1}$

b5) En caso contrario, hacer $\mu_{k+1} = 2\mu_k$ y volver al paso (b1)

In [ ]:
```python
def backtracking(alpha_ini,x_k,f,f_k,df,p_k,mu,rho=0.5,c=0.001,iter_max=500):

    '''
        Esta funcion parte de un tamaño de paso inicial alpha_ini y lo va recortando hasta que
        cumple la cond de descenso suficiente

        parametros:
            valores (float): alpha_ini, rho entre (0,1), f(x_k), Df(x_k) (gradiente en el punto x_k), c_1,
            direccion de descenso (np.rray): p_k

        returns:
            el tamaño de paso a_k
            numero de iteraciones realizadas i_k
    '''

    alpha=alpha_ini #fijamos alpha como el alpha inicial

    for i in range(iter_max):
        x_kp=x_k+alpha*p_k
        gp=c*np.dot(df(x_k,mu),p_k) #hacemos el producto gradiente por direccion de descenso p


        #si la condicion de descenso se cumple, terminamos
        if f(x_kp,mu)<=(f_k + alpha*gp):
            return alpha

        alpha=alpha*rho #si no se cumple la cond, hacemos alpha*rho

    return  alpha

H = np.identity(2)

def BFGS_modificado(f, Df, x0, tol,mu, H0=H, max_iter=5000):
    xk = x0
    gk = Df(xk,mu)
    Hk = H0
    n = len(xk)
    I = np.identity(n)
    for k in range(max_iter):

        if np.linalg.norm(gk) <= tol:
            return xk, f(xk,mu),k, True

        pk = np.dot(-Hk, gk)
        if np.dot(pk,gk) > 0:
            l1 = 10e-5 + (np.dot(pk,gk)/np.dot(gk,gk))
            Hk = Hk + l1*I
            pk = pk - l1*gk

        alpha = backtracking(1, xk, f,f(xk,mu), Df, pk,mu)
        xk_new = xk + alpha*pk
        gk_new = Df(xk_new,mu)
        y = gk_new - gk
        s = xk_new - xk

        if np.dot(y,y) < tol:
            return xk, f(xk,mu), k, True

        if np.dot(y,s) <= 0:
            l2 = 10e-5 - (np.dot(y,s)/np.dot(y,y))
            Hk = Hk + l2*I
        else:
            rho_k = 1/np.dot(y,s)
            Hk = (np.identity(n) - rho_k * (s@y.T)) @ Hk @ (np.identity(n) - rho_k * (y@s.T)) + rho_k * (s@s.T)

        xk = xk_new
        gk = gk_new


    return xk, f(xk,mu), k, False
```

In [ ]:
```python
def penalizacion(c,mu,x0,tol,Q,D_Q,max_iter=1000):
    x_k=x0

    for k in range(max_iter):
        tol=(1+((10*max_iter)/((10*k)+1)))*tol

        #############################
        x_old=x_k
        x_k,f_k,it,_=BFGS_modificado(Q, D_Q,  x_old, tol,mu)
        print(x_k,'\n')
        print(f_k,'\n')
        print(it,'\n')
        print(Q(x_k,mu),'\n')
        print(c(x_k),'\n')
        print("-----------")
        #############################

        if np.linalg.norm(x_k-x_old)<tol:
            return x_k

        mu=2*mu
```

3. Probar el algoritmo tomando como f a la función de Beale, $c_1(\mathbf{x}) = x_1^2 + x_2^2 - 4$, $\mu_0 = 0.5$, $N = 1000$ y $\tau = \epsilon_m^{1/3}$.

   Use los puntos iniciales $\mathbf{x}_0 = (0, 2)$ y $\mathbf{x}_0 = (0, -2)$.

In [ ]:
```python
mu=0.5
x0=np.array([0,2])
tol=(epsilon)**(1.0/3.0)
Q=lambda  x, mu=0.5: fn.Beale(x) + (mu/2) * (c1(x))**2
DQ=lambda x,mu: fn.D_Beale(x) + mu * c1(x) * Dc1(x)

penalizacion(c1,mu,x0,tol, Q, DQ,max_iter=1000)
```

```
[-1.49494416  1.44499334]

1.2641428363038072

53

1.2641428363038072

0.32286377519318155

-----------
[-1.49494416  1.44499334]

1.2902030906368054

0

1.2902030906368054

0.32286377519318155

-----------
Out[ ]:  array([-1.49494416,  1.44499334])
```

In [ ]:
```python
mu=0.5
x0=np.array([0,-2])
tol=(epsilon)**(1.0/3.0)

Q=lambda  x, mu: fn.Beale(x) + (mu/2) * (c1(x))**2
DQ=lambda x,mu: fn.D_Beale(x) + mu * c1(x) * Dc1(x)

penalizacion(c1,mu,x0,tol, Q, DQ,max_iter=1000)
```

```
[ 2.30427309 -0.18490285]

2.0595627997687145

12

2.0595627997687145

1.3438635182249037

-----------
[ 2.30427309 -0.18490285]

2.511055088672668

0

2.511055088672668

1.3438635182249037

-----------
Out[ ]:  array([ 2.30427309, -0.18490285])
```

4. Para verificar el resultado obtenido haga lo siguiente:

- Genere una partición $\theta_0 < \theta_1 < \ldots \theta_m$ del intervalo $[0, 2\pi]$ con $m = 1000$
- Evalue la función de Beale en los puntos $(2\cos\theta_i, 2\sin\theta_i)$ para $i = 0, 1, \ldots, m$.
  e imprima el punto en donde la función tuvo el menor valor y el valor de la función en ese punto.

In [ ]:
```python
lista = np.linspace(0, np.pi, 1000)
beale_list=[fn.Beale([2*np.cos(x),2*np.sin(x)]) for x in lista]

#minimo Beale
min_beale = min(beale_list)

#minimo puntos
min_index = np.argmin(beale_list)
min_point = lista[min_index]
min_sincos= [2*np.cos(min_point ),2*np.sin(min_point )]

print("El mínimo de la lista de valores de Beale es:", min_beale)
print("El punto que corresponde al mínimo de beale_list es:", min_sincos)
```

```
El mínimo de la lista de valores de Beale es: 0.5340819590427182
El punto que corresponde al mínimo de beale_list es: [1.993822324641864, 0.1570749431313392]
```

Ejercicio 2

Programar el método de Newton para resolver el sistema de ecuaciones no lineales (Algoritmo 1 de la Clase 24):

$$\begin{aligned}
2x_0 + x_1 &= 5 - 2x_2^2 \\
x_1^3 + 4x_2 &= 4 \\
x_0x_1 + x_2 &= \exp(x_2)
\end{aligned}$$

1. Programar la función $\mathbf{F}(\mathbf{x})$ correspondiente a este sistema de ecuaciones y su Jacobiana $\mathbf{J}(\mathbf{x})$

In [ ]:
```python
def F(x):
    f1=2*x[0]+x[1]+2*x[2]**2
    f2=x[1]**3+4*x[2]-4
    f3=x[0]*x[1]+x[2]-np.exp(x[2])
    return np.array([f1,f2,f3])

def D_F(x):
    df1_dx = [2, 1, 4*x[2]]
    df2_dx = [0, 3*(x[1]**2), 4]
    df3_dx = [x[1], x[0], 1 - np.exp(x[2])]
    return np.array([df1_dx, df2_dx, df3_dx])
```

2. Programe el algoritmo del método de Newton. Use como condición de paro que el ciclo termine cuando $\|\mathbf{F}(\mathbf{x}_k)\| < \tau$, para una tolerancia τ dada. Haga que el algoritmo devuelva el punto $\mathbf{x}_k$, el número de iteraciones k, el valor $\|\mathbf{F}(\mathbf{x}_k)\|$ y una variable indicadora $bres$ que es 1 si se cumplió el criterio de paro o 0 si terminó por iteraciones.

In [ ]:
```python
def Newton_nolineal(F,DF,x0,tol,N=100):
    x_k=x0
    for k in range(N):
        f_k=F(x_k)

        if np.linalg.norm(f_k)<tol:
            return x_k,k,np.linalg.norm(f_k),True

        df_k=DF(x_k)
        s=np.linalg.solve(df_k,-f_k)
        x_k=x_k+s

    return x_k,k,np.linalg.norm(f_k), False
```

3. Para probar el algoritmo y tratar de encontrar varias raíces, haga un ciclo para hacer 20 iteraciones y en cada iteración haga lo siguiente:

- Dé el punto inicial $\mathbf{x}_0$ como un punto aleatorio generado con `numpy.random.randn(3)`
- Ejecute el método de Newton usando $\mathbf{x}_0$, la tolerancia $\tau = \sqrt{\epsilon_m}$ y un máximo de iteraciones $N = 100$.
- Imprima el punto $\mathbf{x}_k$ que devuelve el algoritmo, la cantidad de iteraciones realizadas, el valor de $\|\mathbf{F}(\mathbf{x}_k)\|$ y la variable indicadora $bres$.

In [ ]:
```python
tol=np.sqrt(epsilon)
for k in range(20):
    x_0=np.random.randn(3)
    xk,k,normfk,bres=Newton_nolineal(F,D_F,x_0,tol,N=100)
    print(xk,"\n")
    print(k,"\n")
    print(normfk,"\n")
    print(bres,"\n")
    print("--------------------")
```

```
[-4.50564931 -1.74071109  2.31862133]

21

1.0048591735576161e-14

True

--------------------
[-4.50564931 -1.74071109  2.31862133]

12

1.3721740766844603e-11

True

--------------------
[-5.59315732 -1.83783156  2.55187639]

20

6.209631313307019e-14

True

--------------------
[-4.50564931 -1.74071109  2.31862133]

22

1.4537899548029866e-10

True

--------------------
[-5.59315732 -1.83783156  2.55187639]

19

3.94914928765923e-14

True

--------------------
[152.73615204  -7.06137103  89.02144944]

99

1.246863021817399e+39

False

--------------------
[-4.50564931 -1.74071109  2.31862133]

10

2.5121479338940403e-15

True

--------------------
[-4.50564931 -1.74071109  2.31862133]

28

6.990190716529265e-11

True

--------------------
[-4.5056493  -1.74071109  2.31862132]

29

3.767299991431769e-10

True

--------------------
[-4.50564927 -1.74071108  2.31862132]

9

8.084823804387054e-09

True

--------------------
[-5.59315732 -1.83783156  2.55187639]

20

1.008254649593363e-12

True

--------------------
[-5.59315732 -1.83783156  2.55187639]

34

5.8224452132708424e-12

True

--------------------
[-4.50564931 -1.74071109  2.31862133]

12

4.4762782523669927e-14

True

--------------------
[-4.50564931 -1.74071109  2.31862133]

10

3.66205343881779e-14

True

--------------------
[-4.50564927 -1.74071108  2.31862132]

7

7.355543913893329e-09

True

--------------------
[-4.50564931 -1.74071109  2.31862133]

26

1.30836851243432e-14

True

--------------------
[-5.59315733 -1.83783156  2.55187639]

32

1.719332327806561e-09

True

--------------------
[-5.59315732 -1.83783156  2.55187639]

33

3.3523685875004812e-12

True

--------------------
[-4.50564931 -1.74071109  2.31862133]

10

6.718388284492729e-12

True

--------------------
[-1093.48440786    -5.04653372    33.12115187]

99

658609474216390.9

False

--------------------
```

In [ ]: