Curso de Optimización (DEMAT)

Tarea 8

Leslie Janeth Quincosa Ramírez

Descripción:	Fechas
Fecha de publicación del documento:	**Abril 11, 2022**
Fecha límite de entrega de la tarea:	**Mayo 1, 2022**

Indicaciones

- **Envie el notebook que contenga los códigos y las pruebas realizadas de cada ejercicio.**
- **Si se requiren algunos scripts adicionales para poder reproducir las pruebas, agreguelos en un ZIP junto con el notebook.**
- **Genere un PDF del notebook y envielo por separado.**

Ejercicio 1 (3 puntos)

Sea $x = (x_1, x_2, \ldots, x_n)$ **la variable independiente.**

Programar las siguientes funciones y sus gradientes:

- **Función cuadrática**

$$f(\mathbf{x}) = 0.5\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}.$$

Si $\mathbf{I}$
es la matriz identidad y $\mathbf{1}$
es la matriz llena de 1's, ambas de tamaño n
, entonces

$$\mathbf{A} = n\mathbf{I} + \mathbf{1} = \begin{bmatrix} n & 0 & \cdots & 0 \\ 0 & n & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & n \end{bmatrix} + \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

- **Función generalizada de Rosenbrock**

$$f(x) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

$$x_0 = (-1.2, 1, -1.2, 1, \ldots, -1.2, 1)$$

En la implementación de cada función y de su gradiente, se recibe como argumento la variable x
y definimos n
como la longitud del arreglo x
, y con esos datos aplicamos la definición correspondiente.

Estas funciones van a ser usadas para probar los algoritmos de optimización. El punto x_0
que aparece en la definición de cada función es el punto inicial que se sugiere para el algoritmo de optimización.

Solución:

In [1]:

```
# Implementación de la función cuadrática y su gradiente

import numpy as np
from numpy import linalg as LA
from numpy.linalg import eigvals
import sys

def function(x):
    n = x.size
    xT = x.T
    A = n*np.identity(n) + np.ones((n, n))
    b = np.ones(n)
    return 0.5*xT@A@x - b.T@x

def gradient(x):
    n = x.size
    xT = x.T
    A = n*np.identity(n) + np.ones((n, n))
    b = np.ones(n)
    return xT@A - b.T
```

In [2]:

```
from numpy.core.function_base import geomspace
#  Implementación de la función generalizada de Rosenbrock y su gradiente

def Rosenbrock(x):
    return np.sum( 100*(x[1:] - x[:-1]**2)**2 + (1-x[:-1])**2)

def gradRosenbrock(x):
    n = x.size
    g = np.zeros(n)
    g[0] = 200*(x[1]-x[0]**2)*(-2*x[0])-2*(1-x[0])
    g[n-1] = 200*(x[n-1]-x[n-2]**2)
    g[1:-1] = 200*(x[1:-1]-x[:-2]**2) -400*x[1:-1]*(x[2:]-x[1:-1]**2)-2*(1-x[1:-1])
    return g

def vectorx0(n):
    x = np.zeros(n)
    for k in range( int(n/2)):
      x[2*k] = -1.2
      x[2*k+1] = 1
    return x

#x[:-1] me da el vector sin el último
#x[1:] comienza en la segunda posición
```

Ejercicio 2 (3.5 puntos)

Programar el método de gradiente conjugado no lineal de Fletcher-Reeves:

La implementación recibe como argumentos a la función objetivo f
, su gradiente ∇f
, un punto inicial x_0
, el máximo número de iteraciones N
y una tolerancia $\tau > 0$
.

1. **Calcular** $\nabla f_0 = \nabla f(x_0)$ **, $p_0 = -\nabla f_0$ y hacer $res = 0$.**
2. **Para $k = 0, 1, \ldots, N$:**

- **Si $\|\nabla f_k\| < \tau$, hacer $res = 1$ y terminar el ciclo**
- **Usando backtracking calcular el tamaño de paso α_k**
- **Calcular $x_{k+1} = x_k + \alpha_k p_k$**
- **Calcular $\nabla f_{k+1} = \nabla f(x_{k+1})$**
- **Calcular**

$$\beta_{k+1} = \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$$

- **Calcular**

$$p_{k+1} = -\nabla f_{k+1} + \beta_{k+1} p_k$$

1. **Devolver $x_k, \nabla f_k, k, res$**

---

1. **Escriba la función que implemente el algoritmo anterior.**
2. **Pruebe el algoritmo usando para cada una de las funciones del Ejercicio 1, tomando el punto x_0 que se indica.**
3. **Fije $N = 50000$, $\tau = \epsilon_m^{1/3}$.**
4. **Para cada función del Ejercicio 1 cree el punto x_0 correspondiente usado $n = 2, 10, 20$ y ejecute el algoritmo. Imprima**

- **n,**
- **f(x0),**
- **las primeras y últimas 4 entradas del punto x_k que devuelve el algoritmo,**
- **f(xk),**
- **la norma del vector ∇f_k ,**
- **el número k de iteraciones realizadas,**
- **la variable res para saber si el algoritmo puedo converger.**

Solución:

In [18]:

```
# En esta celda puede poner el código de las funciones
# o poner la instrucción para importarlas de un archivo .py
#Implementación de la función de Fletcher-Reeves
def Backtraking(f, fk, gk, xk, pk, a0, rho, c):
    a = a0
    while f(xk + a*pk) > fk +c*a*gk.T@pk:
```

```
        a = rho*a
    return a

def FletcherReeves(f, g, x0, N, tol):
    res = 0
    xk = x0
    for k in range(N):
      gk = g(xk)
      norm = LA.norm(gk)
      if norm < tol:
          res = 1
          break
      else:
          pk = -gk
          fk = f(xk)
          gxk1 = g(xk)   #gradiente gk
          ak = Backtraking(f, fk, gk, xk, pk, 2, 0.6, 0.01)
          xk = xk + ak*pk
          gk = g(xk)        #gk+1
          bk = (gk.T @ gk)/(gxk1.T @ gxk1)
          pk = -gk + bk*pk
          if k+1 >= N:
              res = 0
              break
    return xk, fk, gk, k, res
```

In [19]:

```
# Pruebas realizadas

def Probar(f, g, n, N, tol):
    x0 = vectorx0(n)
    xk, fk, gk, k, res = FletcherReeves(f, g, x0, N, tol)
    if res == 1:
        print('¡Convergió! :)')
    else:
        print('¡No convergió! :(')
    print('Valor n:', n)
    print('f(x0):', f(x0))
    print('xk:', xk[:2], '...', xk[n-2:])
    print('f(xk):', fk)
    print('||fk||:', LA.norm(fk))
    print('Valor k:', k)
    print('res:', res)
```

In [20]:

```
# Pruebas
eps = sys.float_info.epsilon
tol = eps**(1/3)
ns = [2, 10, 20]

for n in ns:
  Probar(Rosenbrock, gradRosenbrock, n, 50000, tol)
  print("")
```

```
¡Convergió! :)
Valor n: 2
f(x0): 24.199999999999996
xk: [1.00000463 1.00000929] ... [1.00000463 1.00000929]
f(xk): 2.1578189665312123e-11
||fk||: 2.1578189665312123e-11
Valor k: 15665
res: 1

¡Convergió! :)
Valor n: 10
f(x0): 2057.0
xk: [0.99999999 0.99999997] ... [0.99999641 0.99999281]
f(xk): 1.7185202555415e-11
||fk||: 1.7185202555415e-11
Valor k: 19374
```

```
res: 1

¡Convergió! :)
Valor n: 20
f(x0): 4598.0
xk: [1. 1.] ... [0.99999649 0.99999296]
f(xk): 1.6479489759246556e-11
||fk||: 1.6479489759246556e-11
Valor k: 21238
res: 1
```

Ejercicio 3 (3.5 puntos)

Programar el método de gradiente conjugado no lineal de usando la fórmula de Hestenes-Stiefel:

En este caso el algoritmo es igual al del Ejercicio 2, con excepción del cálculo de β_{k+1} **. Primero se calcula el vector** $\mathbf{y}_k$ **y luego** β_{k+1} **:**

$$\mathbf{y}_k = \nabla f_{k+1} - \nabla f_k$$

$$\beta_{k+1} = \frac{\nabla f_{k+1}^\top \mathbf{y}_k}{\nabla p_k^\top \mathbf{y}_k}$$

1. **Repita el Ejercicio 2 usando la fórmula de Hestenes-Stiefel.**
2. **¿Cuál de los métodos es mejor para encontrar los óptimos de las funciones de prueba?**

Solución:

In [12]:

```
#Método HestenesStiefel

def HestenesStiefel(f, g, x0, N, tol):
    res = 0
    xk = x0
    for k in range(N):
      gk = g(xk)
      norm = LA.norm(gk)
      if norm < tol:
          res = 1
          break
      else:
          pk = -gk
          fk = f(xk)
          gxk1 = g(xk)    #gradiente gk
          ak = Backtraking(f, fk, gk, xk, pk, 2, 0.5, 0.01)
          xk = xk + ak*pk
          gk = g(xk)         #gk+1
          yk = gk - gxk1
          bk = (gk.T @ yk)/(pk.T @ yk)
          pk = -gk + bk*pk
          if k+1 >= N:
              res = 0
              break
    return xk, fk, gk, k, res
```

In [13]:

```
def Probar2(f, g, n, N, tol):
    x0 = vectorx0(n)
    xk, fk, gk, k, res = HestenesStiefel(f, g, x0, N, tol)
    if res == 1:
        print('¡Convergió! :)')
    else:
        print('¡No convergió! :(')
    print('Valor n:', n)
    print('f(x0):', f(x0))
    print('xk:', xk[:2], '...', xk[n-2:])
    print('f(xk):', fk)
    print('||fk||:', LA.norm(fk))
    print('Valor k:', k)
    print('res:', res)
```

In [14]:

```
eps = sys.float_info.epsilon
tol = eps**(1/3)

for n in ns:
  Probar2(Rosenbrock, gradRosenbrock, n, 50000, tol)
  print("")
```

```
¡Convergió! :)
Valor n: 2
f(x0): 24.199999999999996
xk: [0.99999528 0.99999055] ... [0.99999528 0.99999055]
f(xk): 2.2349270630763973e-11
||fk||: 2.2349270630763973e-11
Valor k: 11341
res: 1

¡Convergió! :)
Valor n: 10
f(x0): 2057.0
xk: [0.99999999 0.99999997] ... [0.99999605 0.99999209]
f(xk): 2.0821293276906092e-11
||fk||: 2.0821293276906092e-11
Valor k: 18605
res: 1

¡Convergió! :)
Valor n: 20
f(x0): 4598.0
xk: [1. 1.] ... [0.99999607 0.99999212]
f(xk): 2.062874650131741e-11
||fk||: 2.062874650131741e-11
Valor k: 20488
res: 1
```

¿Cuál de los métodos es mejor para encontrar los óptimos de las funciones de prueba?

Ambos métodos convergen y son muy parecidos, pero es más eficiente ya que el número de iteraciones k es menor.