Curso de Optimización (DEMAT)

Tarea 9

Leslie Janeth Quincosa Ramírez

Descripción:	Fechas
Fecha de publicación del documento:	Abril 28, 2022
Fecha límite de entrega de la tarea:	Mayo 8, 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 0 (0 puntos)

- Empezar a buscar un tema para el proyecto final del curso.
- En esta tarea no hay que poner la descripción del proyecto. Sólo buscar el tema y tenerlo listo para su entrega en la siguiente tarea.
- La fecha límite para mandar la descripción del proyecto se tiene que mandar es el domingo 17 de mayo.

Ejercicio 1. (2 puntos)

Programar las siguientes funciones y sus gradientes, de modo que dependan de la dimensión n de la variable $\mathbf{x}$:

- Función "Tridiagonal 1" generalizada

$$f(x) = \sum_{i=1}^{n-1} (x_i + x_{i+1} - 3)^2 + (x_i - x_{i+1} + 1)^4$$

- Función generalizada de Rosenbrock

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

In [324]:

```
# Implementación de la funciones y sus gradientes
import numpy as np
from numpy import linalg as LA
from numpy.linalg import eigvals
import sys

#intento de función tridiagonal
def Tridiagonal1(x):
    return np.sum( (x[:-1] + x[1:] - 3)**2 + (x[:-1] - x[1:] + 1)**4)

def gradTridiagonal1(x):
    n = x.size
    g = np.zeros(n)
    g[1:-1] = 2*(x[:-2] + x[1:-1] -3) - 4*(x[:-2] - x[1:-1] + 1)**3 + 2*(x[1:-1] + x[2:
] -3) + 4*(x[1:-1] - x[2:] + 1)**3
    g[0] = 2*(x[0] + x[1] - 3) + 4*(x[0] - x[1] + 1)**3
    g[n-1] = 2*(x[n-2] + x[n-1] - 3) - 4*(x[n-2] - x[n-1] + 1)**3
    return g

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

#Intento de gradiente de Rosenbrock
def gradRosenbrock(x):
    n = x.size
    g = np.zeros(n)
    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])
    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)
    return g

#Función Tridiagonal que usé
def Tridiag(x):
  f=0
  for i in range(len(x)-1):
    f += 100 * (x[i] + x[i+1] - 3)**2 + (x[i] - x[i+1] + 1)**4
    return f

#Gradiente de la tridiagonal que usé
def gradTridiag(x):
    n = len(x)
    g = np.zeros((n), dtype = np.float64)
    g[0] = 2*(x[0] + x[1] - 3) + 4*(x[0] - x[1] + 1)**3
    g[n-1] = 2*(x[n-2] + x[n-1] - 3) - 4*(x[n-2] - x[n-1] + 1)**3
    for i in range(1, n-1):
      g[i] = 2*(x[i-1] + x[i] -3) - 4*(x[i-1] - x[i] + 1)**3 + 2*(x[i] + x[i+1] -3) + 4
*(x[i] - x[i+1] + 1)**3
    return g

#Esta es la función de Rosenbrock generalizada que usé
def fR(x):
  f=0
  for i in range(len(x)-1):
    f += 100 * (x[i+1] - x[i]**2)**2 + (1-x[i])**2
    return f

#Usé este gradiente en el ejercicio
def gradR(x):
    n = len(x)
    g = np.zeros((n), dtype = np.float64)
    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)
    for i in range(1, n-1):
      g[i] = 200*(x[i] -x[i+1]**2) -400*x[i]*(x[i+1]-x[i]**2)- 2*(1-x[i])
    return g
```

En esta parte utilizé diferentes formas de crear las funciones y sus gradientes correspondientes ya que me salía error con los primeros. Los dejo de evidencia porque creo que estaban bien.

Ejercicio 1 (8 puntos)

Programar y probar el método BFGS modificado.

1. **Programar el algoritmo descrito en la diapositiva 16 de la clase 23. Agregue una variable res que indique si el algoritmo terminó porque se cumplió que la magnitud del gradiente es menor que la toleracia dada.**
2. **Probar el algoritmo con las funciones del Ejercicio 1 con la matriz H_0 como la matriz identidad y el punto inicial x_0 como:**

- **La función generalizada de Rosenbrock:**

$$x_0 = (-1.2, 1, \\ -1.2, 1, \ldots, \\ -1.2, 1) \in \mathbb{R}^n$$

- **La función Tridiagonal 1 generalizada:**

$$x_0 = (2, 2, \ldots, 2) \\ \in \mathbb{R}^n$$

Pruebe el algoritmo con la dimensión $n = 2, 10, 100$.

1. **Fije el número de iteraciones máximas a $N = 50000$, y la tolerancia $\tau = \epsilon_m^{1/3}$, donde ϵ_m es el épsilon máquina, para terminar las iteraciones si la magnitud del gradiente es menor que τ. En cada caso, imprima los siguiente datos:**

- n,
- $f(x_0)$,
- **Usando la variable res, imprima un mensaje que indique si el algoritmo convergió,**
- **el número k de iteraciones realizadas,**
- $f(x_k)$,
- **la norma del vector ∇f_k, y**
- **las primeras y últimas 4 entradas del punto x_k que devuelve el algoritmo.**

Solución:

In [325]:

```python
# En esta celda puede poner el código de las funciones
# o poner la instrucción para importarlas de un archivo .py
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 BFGS(f, g, H, x0, N, tol):
    res = 0
    xk = x0.copy()
    n = x0.size
    Hk = H.copy()
    I = np.identity(n)
    for k in range(N):
      gk = g(xk)
      fk = f(xk)
      norm = LA.norm(gk)
      if norm < tol:
        res = 1
        break
      pk = -Hk@gk
      if pk.T@gk > 0:
        lamb1 = 0.00001 + (pk.T@gk)/(gk.T@gk)
        Hk = Hk + lamb1*I
        pk = pk - lamb1*gk     ##Condiciones de Wolfe
      ak = Backtraking(f, fk, gk, xk, pk, 2, 0.5, 0.00001)
```

```
      xk1 = xk.copy()         #xk
      xk = xk + ak*pk #xk+1
      sk = ak*pk #aquí debería estar xk+1-xk, lo modifiqué porque ak era muuuy pequeña
      gk1 = gk.copy()
      gk = g(xk)
      yk = gk - gk1
      if yk.T@sk <= 0:
        lamb2 = 0.00001 - (yk.T@sk)/(sk.T@sk)
        Hk = Hk + lamb2*I
      else:
        rho_k = 1/(yk.T@sk)
        Hk = (I - rho_k*sk@yk.T)@Hk@(I - rho_k*yk@sk.T) + rho_k*sk@sk.T

    return ak, xk, fk, gk, k, res
```

In [326]:

```
# Pruebas realizadas

def vectorx0Ros(n):
  x = np.zeros(n)
  for k in range( int(n/2)):
    x[2*k] = -1.2
    x[2*k+1] = 1
  return x

def vectorx0tri(n):
  x = np.zeros(n)
  for k in range(n):
    x[k] = 2
  return x
```

In [327]:

```
#Función de prueba

def ProbarRos(f, g, n, N, tol):
    x0 = vectorx0Ros(n)
    H = np.identity(n)
    ak, xk, fk, gk, k, res = BFGS(f, g, H, x0, N, tol)
    if res == 1:
        print('¡Convergió! :)')
    else:
        print('¡No convergió! :(')
    print('Valor n:', n)
    print('f(x0):', f(x0))
    print('Valor k:', k)
    print('xk:', xk[:2], '...', xk[n-2:])
    print('f(xk):', fk)
    print('||fk||:', LA.norm(fk))
    print('res:', res)
    print('alpha_k:', ak)

def ProbarTri(f, g, n, N, tol):
    x0 = vectorx0tri(n)
    H = np.identity(n)
    ak, xk, fk, gk, k, res = BFGS(f, g, H, x0, N, tol)
    if res == 1:
        print('¡Convergió! :)')
    else:
        print('¡No convergió! :(')
    print('Valor n:', n)
    print('f(x0):', f(x0))
    print('Valor k:', k)
    print('xk:', xk[:2], '...', xk[n-2:])
    print('f(xk):', fk)
    print('||fk||:', LA.norm(fk))
    print('res:', res)
    print('alpha_k:', ak)
```

In [328]:

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

import warnings
warnings.simplefilter('error', RuntimeWarning)

print('Pruebas para la función de Rosenbrock')

ProbarRos(fR, gradR, 2, 10000, tol)
print(' ')
ProbarRos(fR, gradR, 10, 50000, tol)
print(' ')
ProbarRos(fR, gradR, 100, 50000, tol)
```

```
Pruebas para la función de Rosenbrock
¡Convergió! :)
Valor n: 2
f(x0): 24.199999999999996
Valor k: 6380
xk: [1.00000468 1.00000938] ... [1.00000468 1.00000938]
f(xk): 2.194654208075162e-11
||fk||: 2.194654208075162e-11
res: 1
alpha_k: 0.0001220703125

¡No convergió! :(
Valor n: 10
f(x0): 24.199999999999996
Valor k: 49999
xk: [-1.2  1. ] ... [-1.2  1. ]
f(xk): 24.199999999999996
||fk||: 24.199999999999996
res: 0
alpha_k: 1.3552527156068805e-20

¡No convergió! :(
Valor n: 100
f(x0): 24.199999999999996
Valor k: 49999
xk: [-1.2  1. ] ... [-1.2  1. ]
f(xk): 24.199999999999996
||fk||: 24.199999999999996
res: 0
alpha_k: 5.293955920339377e-23
```

In [329]:

```
print('Pruebas para la función de Tridiagonal')
ProbarTri(Tridiag, gradTridiag, 2, 50000, tol)
print(' ')
ProbarTri(Tridiag, gradTridiag, 10, 50000, tol)
print(' ')
ProbarTri(Tridiag, gradTridiag, 100, 50000, tol)
```

```
Pruebas para la función de Tridiagonal
¡Convergió! :)
Valor n: 2
f(x0): 101.0
Valor k: 2
xk: [1. 2.] ... [1. 2.]
f(xk): 0.0
||fk||: 0.0
res: 1
alpha_k: 0.125

¡No convergió! :(
Valor n: 10
f(x0): 101.0
Valor k: 49999
```

```
xk: [1.25 1.5 ] ... [1.5  2.25]
f(xk): 6.56640625
||fk||: 6.56640625
res: 0
alpha_k: 3.469446951953614e-18

¡No convergió! :(
Valor n: 100
f(x0): 101.0
Valor k: 49999
xk: [1.25 1.5 ] ... [1.5  2.25]
f(xk): 6.56640625
||fk||: 6.56640625
res: 0
alpha_k: 4.336808689942018e-19
```

El valor del α_k del backtraking es pequeño, por lo que afecta directamente al próximo valor de x_{k+1}, de modo que el método está convergiendo de forma lenta.

Para el vector de longitud 2 sí converge el método "rapidamente", pero no me converge para longitudes mayores. Intenté verificar la falla con diferentes métodos. Al principio si dejaba $s_k = x_{k+1} - x_k$ los valores de los gradientes se hacían muy grandes y me arrojaba resultados con NAN por lo que módifique el valor de $s_k = \alpha_k * p_k$, porque cuando hacía las divisiones me quedaba entre 0.

P.D. Se tardaba mucho en hacer los calculos para $n = 100$.