optim_tarea08

May 2, 2022

1 Curso de Optimización (DEMAT)

1.1 Tarea 8

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

1.1.1 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.

1.2 Ejercicio 1 (3 puntos)

Sea $x = (x_1, x_2, ..., x_n)$ 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

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

$$x_0 = (-1.2, 1, -1.2, 1, ..., -1.2, 1)$$

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.

1.2.1 Solución:

[1]:
```
import numpy as np

# Implementación de la función cuadrática y su gradiente

def cuad(x):
    n = x.shape[0]

    b = np.ones((n,))
    A = np.diag(b * n) + np.ones((n,n))

    return 0.5 * x.T @ A @ x - b.T @ x

def g_cuad(x):
    n = x.shape[0]

    b = np.ones((n,))
    A = np.diag(b * n) + np.ones((n,n))

    return A @ x - b
```

[2]:
```
# Implementación de la función tridiagonal generalizada y su gradiente
```

[3]:
```
#  Implementación de la función generalizada de Rosenbrock y su gradiente

def rosenbrock(x):
    x1 = x[:-1]
    x2 = x[1:]

    x3 = 100 * (x2 - x1**2)**2 + (1-x1)**2
    return np.sum(x3)

def g_rosenbrock(x):
    n = x.shape[0]

    x1 = x[:-1]
    x2 = x[1:]
```

```
    d1 = -400 * (x2 - x1**2) * x1 - 2 * (1 - x1)
    d2 = 200 * (x2 - x1**2)

    d = np.zeros((n,))
    d[0] = d1[0]
    d[1:-1] = d1[1:] + d2[:-1]
    d[-1] = d2[-1]

    return d
```

1.3 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 $k = 0, 1, ..., N$:

- 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

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

- Calcular

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

3. 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.

1.3.1 Solución:

[4]:
```
# En esta celda puede poner el código de las funciones
# o poner la instrucción para importarlas de un archivo .py

def backtracking(f, xk, pk, p, beta):
    alpha = 1
    fk = f(xk)
    while True:
        if f(xk + alpha * pk) <= fk - beta*alpha*(pk.T @ pk):
            return alpha
        alpha = p * alpha

def Fletcher_Reeves(fun, grad, x0, maxN, tol):
    xk = x0
    res = 0

    gnext = grad(xk)
    pk = -gnext

    for k in range(maxN):
        gk = gnext
        if np.linalg.norm(gk) < tol:
            res = 1
            break

        alpha = backtracking(fun, xk, pk, 0.8, 0.0001)
        xk = xk + alpha * pk

        gnext = grad(xk)
        betak = (gnext.T @ gnext) / (gk.T @ gk)
        pk = -gnext + betak * pk

        if np.abs(gnext.T @ gk) > 0.2 * np.linalg.norm(gnext)**2:
            # Reinicio
            pk = - gnext

    return xk, gk, k, res
```

[5]:
```
# Pruebas realizadas

EPS_M = np.finfo(float).eps
MAX_N = 50000
TOL = EPS_M ** (1/3)

def short(x):
    if x.shape[0] <= 8:
        print('(', *x, ')')
        return
    y = np.ones((8,))
    y[:4] = x[:4]
    y[4:] = x[-4:]
    print('(', *x[:4], '...', *x[-4:], ')' )

def test_Fletcher_Reeves(n, fun, grad, maxN, tol, mess):
    x0 = np.ones((n,))
    x0[::2] = -1.2

    print(f"Test Fletcher-Reeves | {mess}")
    print(f'n = {n}')
    print(f'f(x0) = {fun(x0)}')

    xk, gk, k, res = Fletcher_Reeves(fun, grad, x0, maxN, tol)

    print('xk = ', end='')
    short(xk)
    print(f'||gk|| = {np.linalg.norm(gk)}')
    print(f'f(xk) = {fun(xk)}')
    print(f'k = {k}')
    print(f'res = {res}')
    print()

nn = (2, 10, 20)
for n in nn:
    test_Fletcher_Reeves(n, cuad, g_cuad, MAX_N, TOL, 'cuadrática')
print()
for n in nn:
    test_Fletcher_Reeves(n, rosenbrock, g_rosenbrock, MAX_N, TOL, 'Rosenbrock')
```

```
Test Fletcher-Reeves | cuadrática
n = 2
f(x0) = 2.66
xk = ( 0.2499991865323705 0.2499991865323705 )
||gk|| = 4.601667816800304e-06
f(xk) = -0.2499999999973531
k = 32
```

```
res = 1

Test Fletcher-Reeves | cuadrática
n = 10
f(x0) = 62.49999999999999
xk = ( 0.049999926177129915 0.049999926177129915 0.049999926177129915
0.049999926177129915 … 0.04999992617712991 0.04999992617712991
0.049999926177129894 0.04999992617712989 )
||gk|| = 4.668968258306211e-06
f(xk) = -0.24999999999945502
k = 50
res = 1

Test Fletcher-Reeves | cuadrática
n = 20
f(x0) = 248.0
xk = ( 0.025000026834587897 0.025000026834587897 0.025000026834587897
0.025000026834587897 … 0.02500002683458789 0.02500002683458789
0.02500002683458789 0.02500002683458789 )
||gk|| = 4.800317013934596e-06
f(xk) = -0.24999999999971195
k = 63
res = 1


Test Fletcher-Reeves | Rosenbrock
n = 2
f(x0) = 24.199999999999996
xk = ( 1.0000029421689873 1.0000058839580541 )
||gk|| = 6.040269183391008e-06
f(xk) = 8.656373449210375e-12
k = 16816
res = 1

Test Fletcher-Reeves | Rosenbrock
n = 10
f(x0) = 2057.0
xk = ( 0.9999999929684152 0.9999999845984702 0.9999999716214408
0.9999999396661543 … 0.9999995254722672 0.9999990451347599 0.9999980879936743
0.9999961651179716 )
||gk|| = 6.051541949616569e-06
f(xk) = 4.8903966157907195e-12
k = 21547
res = 1

Test Fletcher-Reeves | Rosenbrock
n = 20
f(x0) = 4598.0
```

```
xk = ( 1.000000000093608 0.9999999997352695 1.0000000003631793 0.999999999414696
… 0.9999994981489969 0.9999989924753271 0.9999979808082122 0.9999959510718759
)
||gk|| = 5.9474116387793085e-06
f(xk) = 5.4501739383569125e-12
k = 23323
res = 1
```

1.4 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?

1.4.1 Solución:

[6]:
```python
# En esta celda puede poner el código de las funciones
# o poner la instrucción para importarlas de un archivo .py

def backtracking(f, xk, pk, p, beta):
    alpha = 1
    fk = f(xk)
    while True:
        if f(xk + alpha * pk) <= fk - beta*alpha*(pk.T @ pk):
            return alpha
        alpha = p * alpha

def Hestenes_Stiefel(fun, grad, x0, maxN, tol):
    xk = x0
    res = 0

    gnext = grad(xk)
    pk = -gnext

    for k in range(maxN):
        gk = gnext
```

```
        if np.linalg.norm(gk) < tol:
            res = 1
            break

        alpha = backtracking(fun, xk, pk, 0.9, 0.0001)
        xk = xk + alpha * pk

        gnext = grad(xk)

        yk = gnext - gk
        betak = (gnext.T @ yk) / (pk.T @ yk)
        pk = -gnext + betak * pk

        if np.abs(gnext.T @ gk) > 0.2 * np.linalg.norm(gnext)**2:
            # Reinicio
            pk = - gnext

    return xk, gk, k, res
```

[7]:
```
# Pruebas realizadas

EPS_M = np.finfo(float).eps
MAX_N = 50000
TOL = EPS_M ** (1/3)

def short(x):
    if x.shape[0] <= 8:
        print('(', *x, ')')
        return
    y = np.ones((8,))
    y[:4] = x[:4]
    y[4:] = x[-4:]
    print('(', *x[:4], '...', *x[-4:], ')' )

def test_Hestenes_Stiefel(n, fun, grad, maxN, tol, mess):
    x0 = np.ones((n,))
    x0[::2] = -1.2

    print(f"Test Hestenes_Stiefel | {mess}")
    print(f'n = {n}')
    print(f'f(x0) = {fun(x0)}')

    xk, gk, k, res = Hestenes_Stiefel(fun, grad, x0, maxN, tol)

    print('xk = ', end='')
    short(xk)
    print(f'||gk|| = {np.linalg.norm(gk)}')
```

```
    print(f'f(xk) = {fun(xk)}')
    print(f'k = {k}')
    print(f'res = {res}')
    print()

nn = (2, 10, 20)
for n in nn:
    test_Hestenes_Stiefel(n, cuad, g_cuad, MAX_N, TOL, 'cuadrática')
print()
for n in nn:
    test_Hestenes_Stiefel(n, rosenbrock, g_rosenbrock, MAX_N, TOL, 'Rosenbrock')
```

```
Test Hestenes_Stiefel | cuadrática
n = 2
f(x0) = 2.66
xk = ( 0.2500010031173648 0.2500010031173648 )
||gk|| = 5.674488727717848e-06
f(xk) = -0.249999999995975
k = 151
res = 1

Test Hestenes_Stiefel | cuadrática
n = 10
f(x0) = 62.49999999999999
xk = ( 0.0500000943650406 0.0500000943650406 0.0500000943650406
0.0500000943650406 … 0.0500000943650406 0.0500000943650406 0.0500000943650406
0.0500000943650406 )
||gk|| = 5.9681691953423504e-06
f(xk) = -0.2499999999991095
k = 513
res = 1

Test Hestenes_Stiefel | cuadrática
n = 20
f(x0) = 248.0
xk = ( 0.02499996868255284 0.02499996868255284 0.02499996868255284
0.02499996868255284 … 0.024999968682552845 0.024999968682552845
0.024999968682552845 0.024999968682552845 )
||gk|| = 5.602235257728152e-06
f(xk) = -0.2499999999996076
k = 138
res = 1


Test Hestenes_Stiefel | Rosenbrock
n = 2
f(x0) = 24.199999999999996
```

```
xk = ( 1.000000040638996 1.0000000933769284 )
||gk|| = 5.338234889058483e-06
f(xk) = 1.628995072026969e-14
k = 22365
res = 1

Test Hestenes_Stiefel | Rosenbrock
n = 10
f(x0) = 2057.0
xk = ( 0.9999999953186732 0.9999999919770695 0.9999999813335353
0.9999999661309428 … 0.9999997148413826 0.9999994318134096 0.999998858202447
0.99999771205561 )
||gk|| = 5.9764020896617025e-06
f(xk) = 1.747528377143113e-12
k = 22673
res = 1

Test Hestenes_Stiefel | Rosenbrock
n = 20
f(x0) = 4598.0
xk = ( 1.0000000001018456 0.9999999997263639 1.0000000003955472
0.9999999994057973 … 0.9999996729243544 0.9999993428259519 0.9999986833108813
0.9999973595619527 )
||gk|| = 5.975891725190085e-06
f(xk) = 2.3235621297272833e-12
k = 24297
res = 1
```

1.4.2 Conclusión

Notemos que con ambos métodos se obtuvo un resultado convergente en todas las pruebas, más aún, llegaban al mismo mínimo. La única parte donde se desempeñaron distinto fue en el número de iteraciones. Notemos que el método *Fletcher-Reeves* siempre tuvo menos iteraciones que *Hestenes-Stiefel*. Por lo tanto en estas funciones resulto mejor usar el método *Fletcher-Reeves*.

[ ]: