OPTIMIZACIÓN: TAREA 6

EZAU FARIDH TORRES TORRES.

Maestría en Ciencias con Orientación en Matemáticas Aplicadas.

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS.

NOTA: Se usa sólo la condición débil de Wolf ya que la fuerte no convergió en ningún caso para los parámetros dados.

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style = "dark")

def BACKTRAKING_WOLF(a_init: float, p: float, c1: float,
                     c2: float, xk: np.array, f, gradf,
                     dk: np.array, Nb: int):
    a = a_init
    for i in range(Nb): # STRONG / NORMAL
        #if (f(xk + a*dk) <= f(xk) + c1*a*gradf(xk).T @ dk) and (np.abs(grad
        if (f(xk + a*dk) <= f(xk) + c1*a*gradf(xk).T @ dk) and (gradf(xk + a
            return a, i
        a = p*a
    return a, Nb

def f_Himmelblau(x: np.array):
    return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2
def grad_Himmelblau(x: np.array):
    x1 = 4*x[0]*(x[0]**2 + x[1] - 11) + 2*(x[0] + x[1]**2 - 7)
    x2 = 2*(x[0]**2 + x[1] - 11) + 4*x[1]*(x[0] + x[1]**2 - 7)
    return np.array([x1,x2], dtype = float)

def f_Beale(x: np.array):
    return (1.5 - x[0] + x[0]*x[1])**2 + (2.25 - x[0] + x[0]*x[1]**2)**2 + (
def grad_Beale(x: np.array):
    x1 = 2*(x[1] - 1)*(1.5 - x[0] + x[0]*x[1]) + 2*(x[1]**2 - 1)*(2.25 - x[0
    x2 = 2*x[0]*(1.5 - x[0] + x[0]*x[1]) + 4*x[0]*x[1]*(2.25 - x[0] + x[0]*x
    return np.array([x1,x2], dtype = float)

def f_Rosenbrock(x: np.array):
    n = len(x)
    s = 0
    for i in range(n-1):
        s = s + 100*(x[i+1] - x[i]**2)**2 + (1 - x[i])**2
```

```
    return s
def grad_Rosenbrock(x: np.array):
    n = len(x)
    grad = np.zeros(n)
    grad[0] = -400*x[0]*(x[1] - x[0]**2) - 2*(1-x[0])
    grad[n-1] = 200*(x[n-1] - x[n-2]**2)
    for j in range(1,n-1):
        grad[j] = 200*(x[j]-x[j-1]**2) - 400*x[j]*(x[j+1] - x[j]**2) - 2*(1-
    return np.array(grad, dtype = float)
```

1.- Ejercicio 1:

1.1.

Programe el método de gradiente conjugado lineal, Algoritmo 1 de la Clase 18, para resolver el sistema de ecuaciones $\mathbf{Ax} = \mathbf{b}$, donde $\mathbf{A}$ es una matriz simétrica y definida positiva. Haga que la función devuelva el último punto $\mathbf{x}_k$, el último residual $\mathbf{r}_k$, el número de iteraciones k y una variable binaria $bres$ que indique si se cumplió el criterio de paro ($bres = True$) o si el algoritmo terminó por iteraciones ($bres = False$).

In [ ]:
```
def LINEAR_CONJUGATE_GRADIENT(xk: np.array, A: np.array,
                              b: np.array, maxiter: int, tol: float):
    """
    SOLVE THE SYSTEM OF EQUATIONS Ax = b, WHERE A IS A POSITIVE DEFINITE SYM

    Args:
    - xk:      initial guess.
    - A:       positive definite symmetric matrix.
    - b:       vector b.
    - maxiter: maximum number of iterations.
    - tol:     method tolerance.

    Outputs:
    - xk:  approach to the solution.
    - rk:  residual.
    - k:   number of iterations.
    - T/F: if the method converged.
    """
    rk = A @ xk - b
    pk = - rk
    for k in range(maxiter+1):
        if np.linalg.norm(rk) < tol:
            return xk, rk, k, True
        ak = (rk.T @ rk) / (pk.T @ A @ pk)
        xk = xk + ak * pk
```

```
        rk_new = rk + ak * A @ pk
        bk = (rk_new.T @ rk_new) / (rk.T @ rk)
        pk = - rk_new + bk * pk
        rk = rk_new
    return xk, rk, maxiter, False
```

1.2.

Pruebe el algoritmo para resolver el sistema de ecuaciones

$$\mathbf{A}_1\mathbf{x} = \mathbf{b}_1$$

donde

$$\mathbf{A}_1 = n\mathbf{I} + \mathbf{1} = \begin{bmatrix} n & 0 & \cdots & 0 \\ 0 & n & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & n \end{bmatrix} + \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix},$$

n es la dimensión de la variable independiente $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{I}$ es la matriz identidad y $\mathbf{1}$ es la matriz llena de 1's, ambas de tamaño n.

- Use $\mathbf{x}_0$ como el vector cero, el máximo número de iteraciones $N = n$ y una toleracia $\tau = \sqrt{n}\epsilon_m^{1/3}$, donde ϵ_m es el épsilon máquina.
- Pruebe el algoritmo resolviendo los dos sistemas de ecuaciones con $n = 10, 100, 1000$ y en cada caso imprima la siguiente información
- la dimensión n,
- el número k de iteraciones realizadas,
- las primeras y últimas 4 entradas del punto $\mathbf{x}_k$ que devuelve el algoritmo,
- la norma del residual $\mathbf{r}_k$,
- la variable $bres$ para saber si el algoritmo puedo converger.

In [ ]:
```
n = 10
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*(np.finfo(float).eps)**(1/3)
xk, rk, k, conv = LINEAR_CONJUGATE_GRADIENT(xk = x0, A = A1,
                            b = b1, maxiter = n, tol = tol)
print("DIMENSION:   ", n)
print("ITERACIONES: ", k)
```

```
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA rk:    ", np.linalg.norm(rk))
print("CONVERGENCIA:", conv)
```

```
DIMENSION:    10
ITERACIONES:  1
xk:           [0.05 0.05 0.05 0.05] ... [0.05 0.05 0.05 0.05]
NORMA rk:     6.280369834735101e-16
CONVERGENCIA: True
```

In [ ]:
```
n = 100
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*(np.finfo(float).eps)**(1/3)
xk, rk, k, conv = LINEAR_CONJUGATE_GRADIENT(xk = x0, A = A1,
                            b = b1, maxiter = n, tol = tol)
print("DIMENSION:   ", n)
print("ITERACIONES: ", k)
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA rk:    ", np.linalg.norm(rk))
print("CONVERGENCIA:", conv)
```

```
DIMENSION:    100
ITERACIONES:  1
xk:           [0.005 0.005 0.005 0.005] ... [0.005 0.005 0.005 0.005]
NORMA rk:     7.691850745534255e-16
CONVERGENCIA: True
```

In [ ]:
```
n = 1000
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*(np.finfo(float).eps)**(1/3)
xk, rk, k, conv = LINEAR_CONJUGATE_GRADIENT(xk = x0, A = A1,
                            b = b1, maxiter = n, tol = tol)
print("DIMENSION:   ", n)
print("ITERACIONES: ", k)
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA rk:    ", np.linalg.norm(rk))
print("CONVERGENCIA:", conv)
```

```
DIMENSION:    1000
ITERACIONES:  1
xk:           [0.0005 0.0005 0.0005 0.0005] ... [0.0005 0.0005 0.0005 0.000
5]
NORMA rk:     1.192021805172505e-13
CONVERGENCIA: True
```

1.3.

También aplique el algoritmo para resolver el sistema

$$\mathbf{A}_2\mathbf{x} = \mathbf{b}_2$$

donde $\mathbf{A}_2 = [a_{ij}]$ con

$$a_{ij} = exp\left(-0.25(i-j)^2\right), \qquad \mathbf{b}_2 = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

In [ ]:
```
n = 10
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*(np.finfo(float).eps)**(1/3)
xk, rk, k, conv = LINEAR_CONJUGATE_GRADIENT(xk = x0, A = A2,
                            b = b2, maxiter = n, tol = tol)
print("DIMENSION:   ", n)
print("ITERACIONES: ", k)
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA rk:    ", np.linalg.norm(rk))
print("CONVERGENCIA:", conv)
```

```
DIMENSION:    10
ITERACIONES:  5
xk:           [ 1.36909916 -1.16637682  1.60908281 -0.61339053] ... [-0.6133
9053  1.60908281 -1.16637682  1.36909916]
NORMA rk:     3.5332157369569496e-12
CONVERGENCIA: True
```

In [ ]:
```
n = 100
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*(np.finfo(float).eps)**(1/3)
xk, rk, k, conv = LINEAR_CONJUGATE_GRADIENT(xk = x0, A = A2,
                            b = b2, maxiter = n, tol = tol)
print("DIMENSION:   ", n)
print("ITERACIONES: ", k)
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA rk:    ", np.linalg.norm(rk))
print("CONVERGENCIA:", conv)
```

```
DIMENSION:    100
ITERACIONES:  100
xk:           [ 1.44610292 -1.41613796  2.11052474 -1.42522358] ... [-1.4249
2099  2.11046082 -1.41638734  1.44632425]
NORMA rk:     0.0002433600691892639
CONVERGENCIA: False
```

In [ ]:
```
n = 1000
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*(np.finfo(float).eps)**(1/3)
xk, rk, k, conv = LINEAR_CONJUGATE_GRADIENT(xk = x0, A = A2,
                            b = b2, maxiter = n, tol = tol)
print("DIMENSION:   ", n)
print("ITERACIONES: ", k)
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA rk:    ", np.linalg.norm(rk))
print("CONVERGENCIA:", conv)
```

```
DIMENSION:    1000
ITERACIONES:  262
xk:           [ 1.44628824 -1.41635954  2.1105181  -1.42507231] ... [-1.4250
7231  2.1105181  -1.41635954  1.44628824]
NORMA rk:     0.00018766135449068362
CONVERGENCIA: True
```

2.- Ejercicio 2:

Programar el método de gradiente conjugado no lineal descrito en el Algoritmo 3 de Clase 19 usando la fórmula de Fletcher-Reeves:

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

2.1.

Escriba la función que implemente el algoritmo.

- La función debe recibir como argumentos $\mathbf{x}_0$, la función f y su gradiente, el número máximo de iteraciones N, la tolerancia τ, y los parámetros para el algoritmo de backtracking: factor ρ, la constante c_1 para la condición de descenso suficiente, la constante c_2 para la condición de curvatura, y el máximo número de iteraciones N_b.
- Agregue al algoritmo un contador nr que se incremente cada vez que se aplique el reinicio, es decir, cuando se hace $\beta_{k+1} = 0$.
- Para calcular el tamaño de paso α_k use el algoritmo de backtracking usando las

condiciones de Wolfe con el valor inicial $\alpha_{ini} = 1$.

- Haga que la función devuelva el último punto $\mathbf{x}_k$, el último gradiente $\mathbf{g}_k$, el número de iteraciones k y una variable binaria $bres$ que indique si se cumplió el criterio de paro ($bres = True$) o si el algoritmo terminó por iteraciones ($bres = False$), y el contador $br·.

In [ ]:
```python
def NONLINEAR_CONJUGATE_GRADIENT_FR(xk: np.array, f, gradf, maxiter: int,
                                    tol: float, a_init: float, p: float,
                                    c1: float, c2: float, Nb: int):
    """
    THIS FINDS THE MINIMIZER OF f USING THE NON-LINEAR CONJUGATE GRADIENT ME

    Args:
    - xk:      initial guess.
    - f:       function to minimize.
    - gradf:   gradient of the function to minimize.
    - maxiter: maximum number of iterations.
    - tol:     method tolerance.
    - a_init:  initial value for the step size in backtracking.
    - p:       reduction factor for the step size in backtracking.
    - c1:      parameter for the sufficient descent condition.
    - c2:      parameter for the curvature condition.
    - Nb:      maximum number of iterations in backtracking.

    Outputs:
    - xk:  approach to the minimizer of f.
    - gk:  gradient of f at xk.
    - k:   number of iterations.
    - T/F: if the method converged.
    - nr:  number of restarts (bk = 0).
    """
    gk = gradf(xk)
    dk = -gk
    nr = 0
    for k in range(maxiter + 1):
        if np.linalg.norm(gk) < tol:
            return xk, gk, k, True, nr
        ak, k1 = BACKTRAKING_WOLF(a_init = a_init, p = p, c1 = c1, c2 = c2,
                                  xk = xk, f = f, gradf = gradf, dk = dk, Nb =
        xk = xk + ak*dk
        gk_n = gradf(xk)
        if np.abs(gk_n.T @ gk) < 0.2*np.linalg.norm(gk_n)**2:
            bk = (gk_n.T @ gk_n) / (gk.T @ gk)
        else:
            bk = 0
            nr += 1
        dk = -gk_n + bk*dk
        gk = gk_n
    return xk, gk, maxiter, False, nr
```

2.2.

Pruebe el algoritmo usando la siguientes funciones con los puntos iniciales dados. Fije $N = 5000$, $\tau = \sqrt{n}\epsilon_m^{1/3}$, donde n es la dimensión de la variable $\mathbf{x}$ y ϵ_m es el épsilon máquina. Para backtracking use $\rho = 0.5$, $c_1 = 0.001$, $c_2 = 0.01$, $N_b = 500$. Para cada función de prueba imprima:

- la dimensión n,
- $f(\mathbf{x}_0)$,
- el número k de iteraciones realizadas,
- $f(\mathbf{x}_k)$,
- las primeras y últimas 4 entradas del punto $\mathbf{x}_k$ que devuelve el algoritmo,
- la norma del vector gradiente $\mathbf{g}_k$,
- la variable $bres$ para saber si el algoritmo puedo converger.
- el número de reinicios nr.

In [ ]:
```
N = 5000
eps_m = np.finfo(float).eps
p = 0.5
c1 = 0.001
c2 = 0.01
Nb = 500
```

Función de cuadrática 1: Para $\mathbf{x} = (x_1, x_2, \ldots, x_n)$

- $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}_1 \mathbf{x} - \mathbf{b}_1^\top \mathbf{x}$, donde $\mathbf{A}_1$ y $\mathbf{b}_1$ están definidas como en el Ejercicio 1.
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{10}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{100}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{1000}$

In [ ]:
```
n = 10
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A1 @ x - b1.T @ x
gradf_cuad = lambda x: A1 @ x - b1
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_cuad,
                                  gradf = gradf_cuad, maxiter = N, tol = tol,
                                  a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
```

```
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    10
f(x0):        0.0
ITERACIONES:  9
f(xk):        -0.24999999999636202
xk:           [0.05000019 0.05000019 0.05000019 0.05000019] ... [0.05000019
0.05000019 0.05000019 0.05000019]
NORMA gk:     1.206313194339134e-05
CONVERGENCIA: True
REINICIOS:    9
```

In [ ]:
```
n = 100
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A1 @ x - b1.T @ x
gradf_cuad = lambda x: A1 @ x - b1
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    100
f(x0):        0.0
ITERACIONES:  21
f(xk):        -0.24999999999200012
xk:           [0.00500003 0.00500003 0.00500003 0.00500003] ... [0.00500003
0.00500003 0.00500003 0.00500003]
NORMA gk:     5.656829153792843e-05
CONVERGENCIA: True
REINICIOS:    21
```

In [ ]:
```
n = 1000
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A1 @ x - b1.T @ x
gradf_cuad = lambda x: A1 @ x - b1
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
```

```
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    1000
f(x0):        0.0
ITERACIONES:  251
f(xk):        -0.24999999999146394
xk:           [0.0005 0.0005 0.0005 0.0005] ... [0.0005 0.0005 0.0005 0.000
5]
NORMA gk:     0.00018476304776620826
CONVERGENCIA: True
REINICIOS:    251
```

Función de cuadrática 2: Para $\mathbf{x} = (x_1, x_2, \ldots, x_n)$

- $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}_2 \mathbf{x} - \mathbf{b}_2^\top \mathbf{x}$, donde $\mathbf{A}_2$ y $\mathbf{b}_2$ están definidas como en el Ejercicio 1.
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{10}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{100}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{1000}$

In [ ]:
```
n = 10
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A2 @ x - b2.T @ x
gradf_cuad = lambda x: A2 @ x - b2
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    10
f(x0):        0.0
ITERACIONES:  1571
f(xk):        -1.7934207913526614
xk:           [ 1.36889566 -1.16586292  1.60838905 -0.61279115] ... [-0.6127
9115  1.60838905 -1.16586292  1.36889566]
NORMA gk:     1.8950483039484323e-05
CONVERGENCIA: True
REINICIOS:    1230
```

In [ ]:
```
n = 100
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A2 @ x - b2.T @ x
gradf_cuad = lambda x: A2 @ x - b2
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    100
f(x0):        0.0
ITERACIONES:  5000
f(xk):        -14.494288136666577
xk:           [ 1.44208101 -1.40323557  2.08547581 -1.38702193] ... [-1.3870
2193  2.08547581 -1.40323557  1.44208101]
NORMA gk:     0.0006281049138677026
CONVERGENCIA: False
REINICIOS:    4014
```

In [ ]:
```
n = 1000
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A2 @ x - b2.T @ x
gradf_cuad = lambda x: A2 @ x - b2
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
```

```
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    1000
f(x0):        0.0
ITERACIONES:  5000
f(xk):        -141.43694656399958
xk:           [ 1.44220913 -1.40362866  2.08622634 -1.38814564] ... [-1.3881
4564  2.08622634 -1.40362866  1.44220913]
NORMA gk:     0.00029289489704420945
CONVERGENCIA: False
REINICIOS:    4029
```

Función de Beale : Para $\mathbf{x} = (x_1, x_2)$

$$f(\mathbf{x}) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1x_2^2)^2 + (2.625 - x_1 + x_1x_2^3)^2.$$

- $\mathbf{x}_0 = (2, 3)$

In [ ]:
```
x0 = np.array([2,3], dtype = float)
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_Beale,
                                gradf = grad_Beale, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Beale(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Beale(xk))
print("xk:          ", xk)
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    2
f(x0):        3347.203125
ITERACIONES:  78
f(xk):        3.377431057983643e-11
xk:           [2.99998551 0.49999649]
NORMA gk:     6.924867590358913e-06
CONVERGENCIA: True
REINICIOS:    65
```

Función de Himmelblau: Para $\mathbf{x} = (x_1, x_2)$

$$f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2.$$

- $\mathbf{x}_0 = (2, 4)$

In [ ]:
```
x0 = np.array([2,4], dtype = float)
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_Himmelb
                                gradf = grad_Himmelblau, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Himmelblau(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Himmelblau(xk))
print("xk:          ", xk)
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    2
f(x0):        130.0
ITERACIONES:  37
f(xk):        2.0568411381419677e-13
xk:           [ 3.58442828 -1.84812653]
NORMA gk:     6.585280676690073e-06
CONVERGENCIA: True
REINICIOS:    36
```

Función de Rosenbrock: Para $\mathbf{x} = (x_1, x_2, \ldots, x_n)$

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right] \quad n \geq 2.$$

- $\mathbf{x}_0 = (-1.2, 1.0) \in \mathbb{R}^2$
- $\mathbf{x}_0 = (-1.2, 1.0, \ldots, -1.2, 1.0) \in \mathbb{R}^{20}$
- $\mathbf{x}_0 = (-1.2, 1.0, \ldots, -1.2, 1.0) \in \mathbb{R}^{40}$

In [ ]:
```
x0 = np.array([-1.2, 1.0], dtype = float)
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_Rosenbr
                                gradf = grad_Rosenbrock, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb

print("DIMENSION:   ", n)
print("f(x0):       ", f_Rosenbrock(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Rosenbrock(xk))
print("xk:          ", xk)
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
```

```
print("REINICIOS:   ", nr)
```

```
DIMENSION:    2
f(x0):        24.199999999999996
ITERACIONES:  5000
f(xk):        7.904920227577363e-10
xk:           [1.00002811 1.0000562 ]
NORMA gk:     6.73075488715537e-05
CONVERGENCIA: False
REINICIOS:    4249
```

In [ ]:
```
x0 = np.array([-1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2,
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_Rosenbr
                                gradf = grad_Rosenbrock, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Rosenbrock(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Rosenbrock(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    20
f(x0):        4598.000000000001
ITERACIONES:  956
f(xk):        2.0658248626373865e-11
xk:           [1.         0.99999999 1.         0.99999999] ... [0.99999903
0.99999805 0.99999609 0.99999215]
NORMA gk:     2.5601483353585578e-05
CONVERGENCIA: True
REINICIOS:    788
```

In [ ]:
```
x0 = np.array([-1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2,
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_FR(xk = x0, f = f_Rosenbr
                                gradf = grad_Rosenbrock, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Rosenbrock(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Rosenbrock(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    40
f(x0):        9680.000000000002
ITERACIONES:  2681
f(xk):        1.6050071699166508e-10
xk:           [1. 1. 1. 1.] ... [0.99999727 0.99999454 0.99998906 0.9999780
7]
NORMA gk:     3.221647914928825e-05
CONVERGENCIA: True
REINICIOS:    2292
```

3.- Ejercicio 3 :

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}:

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

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

In [ ]:
```python
def NONLINEAR_CONJUGATE_GRADIENT_HS(xk: np.array, f, gradf, maxiter: int,
                                    tol: float, a_init: float, p: float,
                                    c1: float, c2: float, Nb: int):
    """
    THIS FINDS THE MINIMIZER OF f USING THE NON-LINEAR CONJUGATE GRADIENT ME

    Args:
    - xk:      initial guess.
    - f:       function to minimize.
    - gradf:   gradient of the function to minimize.
    - maxiter: maximum number of iterations.
    - tol:     method tolerance.
    - a_init:  initial value for the step size in backtracking.
    - p:       reduction factor for the step size in backtracking.
    - c1:      parameter for the sufficient descent condition.
    - c2:      parameter for the curvature condition.
    - Nb:      maximum number of iterations in backtracking.

    Outputs:
    - xk:  approach to the minimizer of f.
    - gk:  gradient of f at xk.
    - k:   number of iterations.
    - T/F: if the method converged.
```

```
    - nr:  number of restarts (bk = 0).
    """
    gk = gradf(xk)
    dk = -gk
    nr = 0
    for k in range(maxiter + 1):
        if np.linalg.norm(gk) < tol:
            return xk, gk, k, True, nr
        ak, k1 = BACKTRAKING_WOLF(a_init = a_init, p = p, c1 = c1, c2 = c2,
                                  xk = xk, f = f, gradf = gradf, dk = dk, Nb =
        xk = xk + ak*dk
        gk_n = gradf(xk)
        yk = gk_n - gk
        if np.abs(gk_n.T @ gk) < 0.2*np.linalg.norm(gk_n)**2:
            bk = (gk_n.T @ yk) / (dk.T @ yk)
        else:
            bk = 0
            nr += 1
        dk = -gk_n + bk*dk
        gk = gk_n
    return xk, gk, maxiter, False, nr
```

3.1.

Repita el Ejercicio 2 usando la fórmula de Hestenes-Stiefel.

In [ ]:
```
N = 5000
eps_m = np.finfo(float).eps
p = 0.5
c1 = 0.001
c2 = 0.01
Nb = 500
```

Función de cuadrática 1: Para $\mathbf{x} = (x_1, x_2, \ldots, x_n)$

- $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}_1 \mathbf{x} - \mathbf{b}_1^\top \mathbf{x}$, donde $\mathbf{A}_1$ y $\mathbf{b}_1$ están definidas como en el Ejercicio 1.
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{10}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{100}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{1000}$

In [ ]:
```
n = 10
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A1 @ x - b1.T @ x
gradf_cuad = lambda x: A1 @ x - b1
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_cuad,
                                  gradf = gradf_cuad, maxiter = N, tol = tol,
```

```
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    10
f(x0):        0.0
ITERACIONES:  9
f(xk):        -0.24999999999636202
xk:           [0.05000019 0.05000019 0.05000019 0.05000019] ... [0.05000019
0.05000019 0.05000019 0.05000019]
NORMA gk:     1.206313194339134e-05
CONVERGENCIA: True
REINICIOS:    9
```

In [ ]:
```
n = 100
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A1 @ x - b1.T @ x
gradf_cuad = lambda x: A1 @ x - b1
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    100
f(x0):        0.0
ITERACIONES:  21
f(xk):        -0.24999999999200012
xk:           [0.00500003 0.00500003 0.00500003 0.00500003] ... [0.00500003
0.00500003 0.00500003 0.00500003]
NORMA gk:     5.656829153792843e-05
CONVERGENCIA: True
REINICIOS:    21
```

In [ ]:
```
n = 1000
A1 = n*np.eye(n, dtype = float) + np.ones([n,n], dtype = float)
b1 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A1 @ x - b1.T @ x
gradf_cuad = lambda x: A1 @ x - b1
```

```
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    1000
f(x0):        0.0
ITERACIONES:  251
f(xk):        -0.24999999999146394
xk:           [0.0005 0.0005 0.0005 0.0005] ... [0.0005 0.0005 0.0005 0.000
5]
NORMA gk:     0.00018476304776620826
CONVERGENCIA: True
REINICIOS:    251
```

Función de cuadrática 2: Para $\mathbf{x} = (x_1, x_2, \ldots, x_n)$

- $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}_2 \mathbf{x} - \mathbf{b}_2^\top \mathbf{x}$, donde $\mathbf{A}_2$ y $\mathbf{b}_2$ están definidas como en el Ejercicio 1.
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{10}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{100}$
- $\mathbf{x}_0 = (0, \ldots, 0) \in \mathbb{R}^{1000}$

In [ ]:
```
n = 10
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A2 @ x - b2.T @ x
gradf_cuad = lambda x: A2 @ x - b2
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
```

```
print("REINICIOS:   ", nr)
```

```
DIMENSION:    10
f(x0):        0.0
ITERACIONES:  1571
f(xk):        -1.7934207913526614
xk:           [ 1.36889566 -1.16586292  1.60838905 -0.61279115] ... [-0.6127
9115  1.60838905 -1.16586292  1.36889566]
NORMA gk:     1.8950483039484323e-05
CONVERGENCIA: True
REINICIOS:    1230
```

In [ ]:
```
n = 100
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A2 @ x - b2.T @ x
gradf_cuad = lambda x: A2 @ x - b2
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    100
f(x0):        0.0
ITERACIONES:  5000
f(xk):        -14.494146346147428
xk:           [ 1.43574711 -1.38364255  2.04861357 -1.33199051] ... [-1.3319
9051  2.04861357 -1.38364255  1.43574711]
NORMA gk:     0.0012286586322716373
CONVERGENCIA: False
REINICIOS:    4035
```

In [ ]:
```
n = 1000
A2 = np.zeros((n,n), dtype = float)
for i in range(n):
    for j in range(n):
        A2[i,j] = np.exp(-0.25*(i-j)**2)
b2 = np.ones(n, dtype = float)
f_cuad = lambda x: 0.5 * x.T @ A2 @ x - b2.T @ x
gradf_cuad = lambda x: A2 @ x - b2
x0 = np.zeros(n, dtype = float)
tol = np.sqrt(n)*eps_m**(1/3)
```

```
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_cuad,
                                gradf = gradf_cuad, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_cuad(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_cuad(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    1000
f(x0):        0.0
ITERACIONES:  5000
f(xk):        -141.43680009612302
xk:           [ 1.43564684 -1.38334367  2.04804318 -1.33115958] ... [-1.3311
5958  2.04804318 -1.38334367  1.43564684]
NORMA gk:     0.0007796329487797677
CONVERGENCIA: False
REINICIOS:    4022
```

Función de Beale : Para $\mathbf{x} = (x_1, x_2)$

$$f(\mathbf{x}) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2.$$

- $\mathbf{x}_0 = (2, 3)$

In [ ]:
```
x0 = np.array([2,3], dtype = float)
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_Beale,
                                gradf = grad_Beale, maxiter = N, tol = tol,
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Beale(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Beale(xk))
print("xk:          ", xk)
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    2
f(x0):        3347.203125
ITERACIONES:  769
f(xk):        5.6113870648033834e-11
xk:           [3.00001871 0.50000457]
NORMA gk:     7.496323664379619e-06
CONVERGENCIA: True
REINICIOS:    585
```

Función de Himmelblau: Para $\mathbf{x} = (x_1, x_2)$

$$f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2.$$

- $\mathbf{x}_0 = (2, 4)$

In [ ]:
```
x0 = np.array([2,4], dtype = float)
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_Himmelb
                                gradf = grad_Himmelblau, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Himmelblau(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Himmelblau(xk))
print("xk:          ", xk)
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    2
f(x0):        130.0
ITERACIONES:  37
f(xk):        1.9833788488942771e-13
xk:           [ 3.58442828 -1.84812653]
NORMA gk:     6.466611484709132e-06
CONVERGENCIA: True
REINICIOS:    36
```

Función de Rosenbrock: Para $\mathbf{x} = (x_1, x_2, \ldots, x_n)$

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right] \quad n \geq 2.$$

- $\mathbf{x}_0 = (-1.2, 1.0) \in \mathbb{R}^2$
- $\mathbf{x}_0 = (-1.2, 1.0, \ldots, -1.2, 1.0) \in \mathbb{R}^{20}$
- $\mathbf{x}_0 = (-1.2, 1.0, \ldots, -1.2, 1.0) \in \mathbb{R}^{40}$

In [ ]:
```
x0 = np.array([-1.2, 1.0], dtype = float)
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_Rosenbr
                                gradf = grad_Rosenbrock, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Rosenbrock(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Rosenbrock(xk))
```

```
print("xk:          ", xk)
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    2
f(x0):        24.199999999999996
ITERACIONES:  1382
f(xk):        1.606990862536676e-11
xk:           [0.999996   0.99999198]
NORMA gk:     8.384887632319202e-06
CONVERGENCIA: True
REINICIOS:    1128
```

In [ ]:
```
x0 = np.array([-1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2,
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_Rosenbr
                                gradf = grad_Rosenbrock, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Rosenbrock(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Rosenbrock(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    20
f(x0):        4598.000000000001
ITERACIONES:  1618
f(xk):        2.104455538164449e-10
xk:           [1. 1. 1. 1.] ... [0.99999688 0.99999374 0.99998745 0.9999748
3]
NORMA gk:     2.615760044191093e-05
CONVERGENCIA: True
REINICIOS:    1250
```

In [ ]:
```
x0 = np.array([-1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2,
n = len(x0)
tol = np.sqrt(n)*eps_m**(1/3)
xk, gk, k, conv, nr = NONLINEAR_CONJUGATE_GRADIENT_HS(xk = x0, f = f_Rosenbr
                                gradf = grad_Rosenbrock, maxiter = N, tol =
                                a_init = 1, p = p, c1 = c1, c2 = c2, Nb = Nb
print("DIMENSION:   ", n)
print("f(x0):       ", f_Rosenbrock(x0))
print("ITERACIONES: ", k)
print("f(xk):       ", f_Rosenbrock(xk))
print("xk:          ", xk[:4], "...", xk[-4:])
print("NORMA gk:    ", np.linalg.norm(gk))
print("CONVERGENCIA:", conv)
print("REINICIOS:   ", nr)
```

```
DIMENSION:    40
f(x0):        9680.000000000002
ITERACIONES:  5000
f(xk):        0.11268245259510778
xk:           [1.00000798 0.99998119 1.00003116 0.99995921] ... [0.92147857
0.84769665 0.71728795 0.51265151]
NORMA gk:     1.2311603097582053
CONVERGENCIA: False
REINICIOS:    3977
```

3.2.

¿Hay alguna diferencia que indique que es mejor usar la fórmula de Hestenes-Stiefel respesto a Fletcher-Reeves?

> Comparando el rendimiento del algoritmo con ambas fórmulas usando exactamente los mismos parámetros, se puede notar que se tiene el mismo rendimiento en la función cuadrática 1, llegando a la misma solución por ambos métodos y coincidiendo en la convergencia. En la función cuadrática 2 se tuvo un desempeño similar, con algunas diferencias en el número de reinicios, ligeramente en favor de la fórmula de Fletcher-Reeves.
>
> En las función de Beale, la fórmula de Fletcher-Reeves tuvo mejor desempeño mientras que en Rosenbrock, Hestenes-Stiefel logró la convergencia del algoritmo en los casos donde Fletcher-Reeves no.
>
> En términos generales, Fletcher-Reeves tiene mejor desempeño ya que en prácticamente todos los casos tuvo una cantidad de iteraciones menor.

3.3.

La cantidad de reinicios puede indicar que tanto se comporta el algoritmo como el algoritmo de descenso máximo. Agregue un comentario sobre esto de acuerdo a los resultados obtenidos para cada fórmula.

> Comparando la cantidad de reincios de cada implementación, se obutvieron los siguientes resultados:

Fletcher-Reeves	Hestenes-Stiefel
9	9
21	21

251	251
1230	1230
4014	4035
4029	4022
65	585
36	36
4249	1128
788	1250
2292	3977

> Aquí podemos notar que, en términos generales, la fórmula de Fletcher-Reeves usa menor cantidad de reinicios, es decir se comporta más como el algoritmo de descenso máximo.