EXAMEN_OP_EFFT

May 25, 2024

OPTIMIZACIÓN: EXÁMEN PARCIAL II EZAU FARIDH TORRES TORRES.

Maestría en Ciencias con Orientación en Matemáticas Aplicadas.

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS.

[ ]:
```
# FUNCIONES Y PAQUETES A UTLLIZAR
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
eps = np.finfo(float).eps

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 + (2.
 ↪625 - x[0] + x[0]*x[1]**3)**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] +␣
 ↪x[0]*x[1]**2) + 2*(x[1]**3 - 1)*(2.625 - x[0] + x[0]*x[1]**3)
    x2 = 2*x[0]*(1.5 - x[0] + x[0]*x[1]) + 4*x[0]*x[1]*(2.25 - x[0] +␣
 ↪x[0]*x[1]**2) + 6*x[0]*(x[1]**2)*(2.625 - x[0] + x[0]*x[1]**3)
    return np.array([x1,x2], dtype = float)
def Hess_Beale(x: np.array):
    x11 = 2*(x[1]**3 - 1)**2 + 2*(x[1]**2 - 1)**2 + 2*(x[1] - 1)**2
    x12 = 4*x[0]*x[1]*(x[1]**2 - 1) + 4*x[1]*(x[0]*x[1]**2 - x[0]+2.25) +␣
 ↪6*x[0]*x[1]**2*(x[1]**3 - 1) + 6*x[1]**2*(x[0]*x[1]**3 - x[0]+2.625) +␣
 ↪2*x[0]*(x[1]-1) + 2*(x[0]*x[1] - x[0]+1.5)
    x22 = 18*x[0]**2*x[1]**4 + 8*x[0]**2*x[1]**2 + 2*x[0]**2 +␣
 ↪12*x[0]*x[1]*(x[0]*x[1]**3 - x[0] + 2.625) + 4*x[0]*(x[0]*x[1]**2 - x[0]+2.
 ↪25)
    return np.array([[x11, x12], [x12, x22]], dtype = float)

def BACKTRAKING(alpha_i: float, p: float, c: float,
                xk: np.array, f, fxk: np.array,
                gradfxk: np.array, pk: np.array, Nb: int):
    alpha = alpha_i
    for i in range(Nb):
```

```
        if f(xk + alpha*pk) <= fxk + c*alpha*(gradfxk.T)@pk:
            return alpha, i
        alpha = p*alpha
    return alpha, Nb
def BFGS_MOD(f, gradf, xk: np.array, tol: float, Hk: np.array,
             N: int, alpha_i, p: float, c: float, Nb: int):
    """
    BFGS METHOD WITH MODIFICATION FOR THE HESSIAN MATRIX.

    Args:
    - f:       function to minimize.
    - gradf:   gradient of the function.
    - xk:      initial point.
    - tol:     tolerance.
    - Hk:      initial Hessian matrix.
    - N:       maximum number of iterations.
    - alpha_i: initial step size.
    - p:       reduction factor for the step size.
    - c:       constant for the Armijo condition.
    - Nb:      maximum number of iterations for the backtracking line search.

    Returns:
    - xk:  optimal point.
    - gk:  gradient at the optimal point.
    - k:   number of iterations.
    - T/F: if the method converged.
    """
    n = len(xk)
    for k in range(N-1):
        gk = gradf(xk)
        if np.linalg.norm(gk) < tol:
            return xk, k#, gk, True
        pk = - Hk @ gk
        if pk.T @ gk > 0:
            lb1 = 10**(-5) + (pk.T @ gk)/(gk.T @ gk)
            Hk = Hk + lb1*np.eye(n)
            pk = pk - lb1*gk
        ak = BACKTRAKING(alpha_i = alpha_i, p = p, c = c, xk = xk, f = f,
                         fxk = f(xk), gradfxk = gk, pk = pk, Nb = Nb)[0]
        xk_n = xk + ak * pk
        gk_n = gradf(xk_n)
        sk = xk_n - xk
        yk = gk_n - gk
        if yk.T @ sk <= 0:
            lb2 = 10**(-5) - (yk.T @ sk)/(yk.T @ yk)
            Hk = Hk + lb2*np.eye(n)
        else:
```

```
            rhok = 1/(yk.T @ sk)
            Hk = (np.eye(n) - rhok*np.outer(sk,yk)) @ Hk @ (np.eye(n) - rhok*np.
↪outer(yk,sk)) + rhok*np.outer(sk,sk)
        xk = xk_n
    return xk, N#, gk, False
```

---

1 1.- EJERCICIO 1:

Considere el problema

$$\min\ f(\mathbf{x}) \quad \text{sujeto a} \quad c_1(\mathbf{x}) = 0.$$

Encontrar la solución usando un penalización cuadrática (clase 29). Para esto contruimos la función

$$Q(\mathbf{x};\mu) = f(\mathbf{x}) + \frac{\mu}{2}(c_1(\mathbf{x}))^2$$

1.1 1.1.

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

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

[ ]:
```
def Q(f, c, x: np.array, mu: float):
    return f(x) + 0.5*mu*(c(x))**2
def gradQ(c, gradf, gradc, x: np.array, mu: float):
    return gradf(x) + mu*c(x)*gradc(x)
```

1.2 1.2.

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 $k = 0, 1, ..., N$ 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)

[ ]:
```
def QUADRATIC_P_BFGS(f, c, Q, gradf, gradQ, gradc, xk: np.array, mu: float, tol:
 ↪ float, N: int,
                     Hk: np.array, NBFGS: int, alpha_i: float, p: float, cb:␣
 ↪float, Nb: int):
    bres = 0
    for k in range(N):
        tk = (1 + (10*N)/(10*k+1))*tol
        xk_n, k1 = BFGS_MOD(f = lambda xk: Q(f = f, c = c, x = xk, mu = mu),
                            gradf = lambda xk: gradQ(c = c, gradf = gradf,␣
 ↪gradc = gradc, x = xk, mu = mu),
                            xk = xk, tol = tk, Hk = Hk, N = NBFGS, alpha_i =␣
 ↪alpha_i, p = p, c = cb, Nb = Nb)
        print(xk_n)
        print(f(xk_n))
        print(Q(f, c, xk_n, mu))
        print(k1)
        print(c(xk_n))
        print("")
        if np.linalg.norm(xk_n - xk) < tol:
            bres = 1
            return xk_n, k, bres
        mu = 2*mu
        xk = xk_n
```

1.3 1.3

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 $\mathbf{x}_0 = (0, -2)$.

[ ]:
```
mu = 0.5
N = 1000
tol = eps**(1/3)

alpha_i = 1 # Para backtracking
p = 0.5
cb = 0.001
Nb = 500

def c(x: np.array):
    return x[0]**2 + x[1]**2 - 4
def gradc(x: np.array):
    return np.array([2*x[0], 2*x[1]], dtype = float)
```

[ ]:
```

```

```
x0 = np.array([0,2], dtype = float)
xk, k, bres = QUADRATIC_P_BFGS(f = f_Beale, c = c, Q = Q, gradf = grad_Beale,␣
 ↪gradQ = gradQ,
                      gradc = gradc, xk = x0, mu = mu, tol = tol, N = N,
                      Hk = np.eye(2), NBFGS = N, alpha_i = alpha_i, p = p, cb =␣
 ↪cb, Nb = Nb)
```

```
[-1.49992474  1.44408303]
1.235999730204742
1.2640811163758396
10
0.33515003309620894

[-1.43800072  1.45765794]
1.2616241655828457
1.2801740012431
3
0.19261274963124464

[-1.3982482   1.46664967]
1.2791085487582916
1.2903783448337678
4
0.10615929575631267

[-1.37437289  1.47221241]
1.290005100272587
1.2963467848482912
4
0.05631023253239231

[-1.36103089  1.47537623]
1.2962305091173503
1.2996270923622686
4
0.029140106575466618

[-1.35391069  1.47708085]
1.2995943100525256
1.3013565895501313
4
0.014841999097180647

[-1.35022187  1.47796941]
1.3013479131945769
1.3022461539587962
4
0.00749266626533629
```

```
[-1.34834211  1.47842435]
1.3022438907037768
1.3026974983938482
3
0.0037650020338286794

[-1.34739582  1.47865192]
1.3026969727838003
1.3029248591556857
3
0.0018869882248466752

[-1.34692015  1.47876666]
1.3029247323559408
1.303038968100577
3
0.0009447045860850878

[-1.34668164  1.47882427]
1.3030389369037287
1.3030961305012871
3
0.0004726653049065277

[-1.34656221  1.47885314]
1.3030961227598534
1.303124738784129
3
0.00023641208178304396

[-1.34650246  1.47886759]
1.303124736855651
1.303139049708821
3
0.00011822603636257867

[-1.34647257  1.47887482]
1.3031390492275392
1.303146206868571
3
5.911803371283497e-05

[-1.34645762  1.47887844]
1.3031462067483532
1.303149785872995
3
2.9560272782447328e-05
```

```
[-1.34645015  1.47888025]
1.3031497858429537
1.3031515754813698
3
1.4780450626084018e-05

[-1.3464489   1.47887888]
1.3031515757666081
1.3031524705826978
1
7.390212435787191e-06
```

[ ]:
```
print(xk)
print(f_Beale(xk))
print(Q(f_Beale, c, xk, mu))
print(c(xk))
```

```
[-1.3464489   1.47887888]
1.3031515757666081
1.3031515757802619
7.390212435787191e-06
```

[ ]:
```
x0 = np.array([0,-2], dtype = float)
xk, k, bres = QUADRATIC_P_BFGS(f = f_Beale, c = c, Q = Q, gradf = grad_Beale,␣
 ↪gradQ = gradQ,
                      gradc = gradc, xk = x0, mu = mu, tol = tol, N = N,
                      Hk = np.eye(2), NBFGS = N, alpha_i = 1, p = 0.5, cb = 0.
 ↪5, Nb = 100)
```

```
[2.1237417  0.20575822]
0.3517591580703046
0.4281050577659986
15
0.552615235749772

[2.06982498 0.18694839]
0.42062516333280164
0.4715455909258934
4
0.3191251403230142

[2.03599599 0.17437182]
0.4686366271771879
0.4995019120592591
4
0.17568518686010837
```

```
[2.0161552  0.16704788]
0.49855600780480214
0.5157747769127545
4
0.09278676928299756

[2.00530023 0.16302848]
0.5154939246285484
0.5246360687839042
4
0.04780728018658831

[1.99959065 0.16094768]
0.5245622177419469
0.5292732914220117
5
0.024266936560021435

[1.99666594 0.15986175]
0.5292538638754547
0.5316472879384405
4
0.012230658360718571

[1.99518277 0.15932756]
0.5316424228155776
0.5328486410832313
5
0.00613957008789523

[1.99443674 0.15905387]
0.5328474018767679
0.5334529736016932
5
0.003076045871237554

[1.99406234 0.15891818]
0.5334526554993994
0.5337560623030653
5
0.0015395991860351899

[1.99387487 0.15884963]
0.5337559771313731
0.5339078383431276
5
0.0007701998821190159
```

```
[1.99378108 0.15881497]
0.533907816556799
0.5339837844174343
5
0.0003851944026118659

[1.9937342  0.15879728]
0.5339837795630822
0.534021771985665
6
0.0001926187300824722

[1.99371073 0.1587887 ]
0.5340217706030913
0.5340407694058396
5
9.631593406300709e-05

[1.99369899 0.1587844 ]
0.5340407691014838
0.5340502690249435
6
4.815929039025946e-05

[1.9936931  0.15878255]
0.5340502677797874
0.534055019062587
5
2.4082993504315198e-05

[1.99369018 0.15878136]
0.5340550185288732
0.5340573941373471
5
1.2041413305574622e-05
```

[ ]:
```
print(xk)
print(f_Beale(xk))
print(Q(f_Beale, c, xk, mu))
print(c(xk))
```

```
[1.99369018 0.15878136]
0.5340550185288732
0.534055018565122
1.2041413305574622e-05
```

1.4 1.4

Para verificar el resultado obtenido haga lo siguiente:

- Genere una partición $\theta_0 < \theta_1 < ...\theta_m$ 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 $i = 0, 1, ..., m$.
  e imprima el punto en donde la función tuvo el menor valor y el valor de la función en ese punto.

[ ]:
```
m = 1000
theta = np.linspace(0, 2 * np.pi, m + 1)
puntos = np.array([(2 * np.cos(t), 2 * np.sin(t)) for t in theta])
valores = np.array([f_Beale(punto) for punto in puntos])

indice_min = np.argmin(valores)
min_point = puntos[indice_min]
min_value = valores[indice_min]

print("PUNTO:", min_point)
print("VALOR:", min_value)
```

```
PUNTO: [1.99333186 0.16318122]
VALOR: 0.5342083939466316
```

NOTAMOS QUE SE OBTIENE EL PUNTO $(1.993331860.16318122)$ **CUANDO** $x_0 = (0, -2)$

---

2 2.- EJERCICIO 2:

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

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

2.1 2.1.

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

[ ]:
```
def F(x: np.array):
    f1 = 2*x[0] + x[1] + 2*x[2]**2 - 5
    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], dtype = float)
def jacF(x: np.array):
```

```
    Jac = np.zeros((len(x), len(x)), dtype = float)
    Jac[0,:] = np.array([2, 1, 4*x[2]], dtype = float)
    Jac[1,:] = np.array([0, 3*x[1]**2, 4], dtype = float)
    Jac[2,:] = np.array([x[1], x[0], 1 - np.exp(x[2])], dtype = float)
    return Jac
```

2.2 2.2.

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.

[ ]:
```
def NEWTON_METHOD(F, J, xk, tol, N):
    for k in range(N):
        Fk = F(xk)
        if np.linalg.norm(Fk) < tol:
            return xk, k, np.linalg.norm(Fk), 1
        Jk = J(xk)
        sk = np.linalg.solve(Jk, -Fk)
        xk = xk + sk
    return xk, N, np.linalg.norm(Fk), 0
```

2.3 2.3.

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*.

[ ]:
```
tol = np.sqrt(eps)
for i in range(20):
    x0 = np.random.randn(3)
    xk, N, NORMA, bres = NEWTON_METHOD(F = F, J = jacF, xk = x0, tol = tol, N =␣
 ↪100)
    print("xk          :", xk)
    print("ITERACIONES :", N)
    print("NORMA       :", NORMA)
    print("CONVERGENCIA:", bres)
    print("")
```

```
xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 6
NORMA       : 3.672556682548643e-15
CONVERGENCIA: 1
```

```
xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 17
NORMA       : 3.568292035180542e-15
CONVERGENCIA: 1

xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 7
NORMA       : 1.7798229048217483e-15
CONVERGENCIA: 1

xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 7
NORMA       : 8.759651536900359e-12
CONVERGENCIA: 1

xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 6
NORMA       : 5.30017380160744e-09
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 6
NORMA       : 7.585886082863531e-11
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 5
NORMA       : 3.2023728339893768e-15
CONVERGENCIA: 1

xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 8
NORMA       : 1.3800434484707094e-08
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 7
NORMA       : 9.713379248062749e-14
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 38
NORMA       : 4.303006528074667e-11
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 7
```

```
NORMA       : 1.9860273225978185e-15
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 11
NORMA       : 1.8518307057095934e-13
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 19
NORMA       : 1.17266462809333e-13
CONVERGENCIA: 1

xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 44
NORMA       : 1.837162083289999e-11
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 6
NORMA       : 2.715404889981033e-13
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 6
NORMA       : 5.304396248435318e-09
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 4
NORMA       : 8.344206004185846e-12
CONVERGENCIA: 1

xk=0        : [ 0.66819062  1.97278644 -0.91946515]
ITERACIONES : 4
NORMA       : 1.3478179954901265e-09
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 5
NORMA       : 2.5255021677253768e-11
CONVERGENCIA: 1

xk=0        : [1.42246939 0.97538853 0.76800804]
ITERACIONES : 6
NORMA       : 9.930136612989092e-16
CONVERGENCIA: 1
```

Aquí podemos verificar que en los 20 casos hay convergencia y además, las normas de $||F(x_k)||$ son muy cercanas a cero, además de un número pequeño de iteraciones.