Métodos Numéricos

Trabajo práctico N°5

Oprimización

Grupo 11: CHEN, Carlos Angel, Legajo 60689
MOLDOVAN LOAYZA, Alexander Stephan, Legajo 60498
MOLINA, Facundo Nicolás, Legajo 60526

Profesor: Pablo Ignacio Fierens

Fecha de entrega: 19 de junio de 2022

Buenos Aires, Argentina

Secciones

Índice de Figuras

Índice de Códigos

1. Función minimi: Método de Nelder-Mead

El trabajo consistió en implementar el algoritmo de optimización de Nelder-Mead. La función programada, llamada **minimi**, recibe como parámetros la función a minimizar, su gradiente, el punto inicial de búsqueda, tolerancia (para determinar si finalizó la búsqueda) y número máximo de iteraciones. Se decidió que la función devolviera como respuesta el último punto óptimo conseguido, la función evaluada en dicho punto, y el número de iteraciones realizadas.

Debido a que el método de Nelder-Mead emplea un simplex en cada iteración en lugar de un solo punto, para conseguir el simplex inicial se obtuvieron puntos adicionales al alterar cada componente del punto original en un determinado porcentaje, elegido de forma aleatoria entre $-5\,\%$ y $+5\,\%$. Dicha tarea fue realizada mediante una función auxiliar, llamada `get_nearby_point`.

Otras funciones auxiliares encargadas de simplificar el código incluyen: `evaluate_and_order`, encargada de evaluar un arreglo de puntos en una función dada, y ordenar ambos arreglos, de puntos y evaluaciones, en función de los valores de este último; `replace_point`, encargada de introducir un punto nuevo al arreglo ordenado de puntos y evaluaciones, dado un índice; y `get_pos`, cuya función es determinar la posición a la que un punto dado debe ingresarse en el arreglo de evaluaciones.

Código 1: Definición de **minimi** y funciones auxiliares

```
import numpy as np
import random

# Función auxiliar get_nearby_point:
# Dado un punto n-dimensional, de forma aleatoria, consigue otro punto cercano
def get_nearby_point(xo):
    x = np.array(xo,dtype=np.longdouble)
    for i in range(len(x)):
        # Cada coordenada es alterada entre -5 % y +5 %
        x[i] = x[i] * (1 + (random.random()-0.5)/10)
    return x

# Función auxiliar evaluate_and_order
# Dado un arreglo de puntos y una función, evalúa la función en dichos
# puntos, y los ordena según dichas evaluaciones
def evaluate_and_order(X,func):
    f = np.array([func(xi) for xi in X])
    i = f.argsort()
    return X[i],f[i]

# Función auxiliar replace_point
# Dado un arreglo de puntos, una función, un arreglo de evaluaciones,
# un punto nuevo, y un índice de posición, modifica el arreglo de puntos
# y el de evaluaciones incluyendo el nuevo punto en la posición indicada
def replace_point(X,f,func,R,pos,p,S):
    I = np.array([i for i in range(pos)] + [p] + [i for i in range(pos,p)],dtype=np.int64)
    X[p] = R; f[p] = func(R)
    X, f = X[I], f[I]
    return X, f, S - X[p] + R

# Función auxiliar get_pos
```

```
32 # Dado un arreglo ordenado y un escalar, devuelve el índice donde debería
33 # colocarse dicho escalar en el arreglo.
34 def get_pos(fR,f):
35     for i,fi in enumerate(f):
36         if fR < fi:
37             break
38     return i
39 
40 #Función:    minimi
41 #           Encuentra el mínimo local de una función dada, mediante
42 #           el método de optimización de Nelder-Mead
43 #Recibe:     func: Función que se quiere optimizar
44 #           grad: Gradiente de la función anterior (no se emplea en este método)
45 #           xo:    Punto inicial para iterar
46 #           tol:   Tolerancia máxima entre la función evaluada en el punto óptimo y el peor
47 #           itmax:Numero máximo de iteraciones.
48 #Devuelve:   Xo: Punto óptimo final
49 #           fo: Función evaluada en dicho punto
50 #           k:   Número de iteraciones que llevó realizó
51 def minimi(func,grad,xo,tol,itmax):
52     random.seed() # Para conseguir n+1 puntos iniciales, se inicializa la librería random
53 
54     n = len(xo) # Dimensión del problema
55     o = 0       # Índice del punto óptimo
56     b = n - 1   # Índice del punto bueno
57     p = n       # Índice del punto peor
58 
59     # Se consiguen los n+1 puntos iniciales
60     X = np.array([get_nearby_point(xo) for i in range(n+1)])
61     # Se los ordena según el valor de f evaluada en dichos puntos
62     X,f = evaluate_and_order(X,func)
63     S = X[0:n].sum(axis=0) # Se calcula la suma de los primeros n
64 
65     for k in range(itmax): # En cada iteración...
66         # ... si la distancia entre el punto óptimo y el peor es menor a la tolerancia,
67         # y además la diferencia entre la función evaluada en dichos puntos es también menor
68         # a la tolerancia:
69         if np.linalg.norm(X[o]-X[p]) < tol and np.linalg.norm(f[o]-f[p]) < tol:
70             # Termina el algoritmo
71             break
72         M = S / n   # Punto medio para calcular reflexiones
73 
74         # REFLEXIÓN
75         R = 2*M - X[p]; fR = func(R)
76         if fR < f[b]: # Reflexión mejor que el bueno
77             if fR > f[o]: # Reflexión peor que el óptimo
78                 # Me quedo con la reflexión
79                 i = get_pos(fR,f) # Calculo su posición en el arreglo
80                 X,f,S = replace_point(X,f,func,R,i,p,S) # Lo coloco, eliminando el peor
81 
82             else: # Reflexión mejor que el óptimo
```

```
            # EXPANSION
            E = 3*M - 2*X[p]; fE = func(E)
            if fE < f[o]: # Expansión mejor que el óptimo
                # Me quedo con la expansión
                X,f,S = replace_point(X,f,func,E,o,p,S) # La coloco, eliminando el peor
            else: # Expansión peor que el óptimo
                # Me quedo con la reflexión
                X,f,S = replace_point(X,f,func,R,o,p,S) # La coloco, eliminando el peor
        else: # Reflexión peor que el bueno
            if fR < f[p]: # Reflexión mejor que el peor
                # Me quedo con la reflexión
                i = get_pos(fR,f) # Calculo su posición en el arreglo
                X,f,S = replace_point(X,f,func,R,i,p,S) # Lo coloco, eliminando el peor
            else: # Reflexión peor que el peor
                # CONTRACCIÓN
                C1, C2 = (R+M)/2, (X[p]+M)/2
                fC1,fC2 = func(C1),func(C2)
                # Calculo la mejor C
                if fC1<fC2:
                    C,fC = C1,fC1
                else:
                    C,fC = C2,fC2
                if fC < f[p]: # Contracción mejor que el peor
                    # Me quedo con la contracción
                    i = get_pos(fC,f) # Calculo su posición en el arreglo
                    X,f,S = replace_point(X,f,func,C,i,p,S) # Lo coloco, eliminando el peor
                else: # Contracción peor que el peor
                    # ENCOGIMIENTO
                    for i in range(1,n+1): # A todos los puntos que no sean el óptimo
                        X[i] = (X[i] + X[o])/2 # Se reemplazan por uno más cercano al óptimo
                    # Se los ordena según el valor de f evaluada en dichos puntos
                    X,f = evaluate_and_order(X,func)
                    S = X[0:n].sum(axis=0) # Se calcula la suma de los primeros n
    return X[o],f[o],k
```

2. Función temperatura()

Para la función `temperatura()` se escribieron 2 funciones auxiliares. Estas consisten en `read_temp_file()` el cual lee el archivo `temp.txt` y devuelve un arreglo ti que contiene todos los datos de la columna tiempo, otro arreglo yi, el cual contiene los valores de las temperaturas y por ultimo devuelve un entero N que contiene la cantidad de filas totales.

Una vez que lee los archivos, los paso por 2da función auxiliar `ejer2_func()` el cual toma los valores que devuelve `read_temp_file()` y los aplica en la siguiente ecuación:

$$\frac{1}{N}\sum_i \left| y_i - \left(a + b \cdot \cos\left(2\pi \frac{t_i}{T_1}\right) + c \cdot \cos\left(2\pi \frac{t_i}{T_2}\right)\right)\right|^2 \tag{1}$$

devolviendo la sumatoria de todos los términos.

Además para el correcto funcionamiento del código, se le agregaron los parámetros iniciales $a, b, c, T_1, T_2, tol, itmax$ y X_0.

Finalmente, se evalúa todo en la función `temperatura()`, este llama a la función `minimi()` con los parámetros necesario y devuelve un arreglo con los parámetros y luego los parámetros evaluados.

Código 2: Definición de `temperatura`

```
a = 1
b = 1
c = 1
T1 = 1
T2 = 1

ti = []
yi = []
N = 0

# Coleccion de puntos iniciales a b c T1 T2
X0 = np.array([10,10,10,11,11])

tol = 1e-12
itmax = 1000

parameter = np.array([a,b,c,T1,T2])

def read_temp_file():
    df = pd.read_csv('temp.txt',header=None,names=['ti','yi'],sep=' ')
    ti = np.array(df['ti'].tolist())
    yi = np.array(df['yi'].tolist())
    N = len(ti)
    return ti, yi, N

def eje2_func(parameter):
    global ti, yi, N
    a = parameter[0]
    b = parameter[1]
    c = parameter[2]
```

```
31      T1 = parameter[3]
32      T2 = parameter[4]
33      aux = abs(yi-(a + b*np.cos(2*np.pi*ti/T1) + c*np.cos(2*np.pi*ti/T2)))**2
34      return aux.sum()/N
35
36  def temperatura():
37      global ti, yi, N
38      ti, yi, N = read_temp_file()
39      param, param_eval, cant_ite = minimi(eje2_func, None, X0, tol, itmax)
40
41      return param, param_eval
```

3. Función `test()`

Se definió la función test() para probar el correcto funcionamiento de minimi() y temperatura().

Para la evaluación de minimi(), se estableció un banco de prueba con funciones tipicamente usadas para análisis de algoritmos de optimización. Se recuperaron del sitio web https://www.sfu.ca/~ssurjano/optimization.html las siguientes seis funciones representativas con mínimo global en $\mathbb{R}^2$, $\mathbb{R}^3$ y $\mathbb{R}^4$. Asimismo, para cada una se establecieron N+1 puntos iniciales para la aplicación del algoritmo.

Se comparó el resultado obtenido de minimi() con el de la implementación de Nelder-Mead de la función *optimization.minimize()* de la biblioteca Scipy. Para esto, se consideró una tolerancia de 10^{-9} para la aproximación al mínimo global mediante el algoritmo y un error de 10^{-6} al comparar la norma de los vectores resultantes.

Función esfera, en $\mathbb{R}^3$:

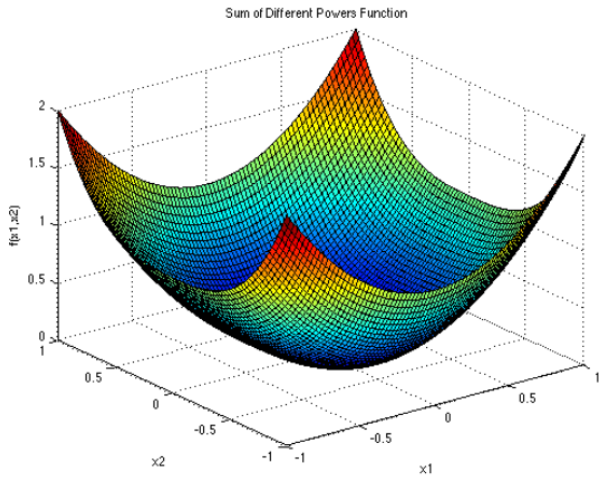
$$f_1(x_0, x_1, x_2) = x_0^2 + x_1^2 + x_2^2$$

$$x^* = (0, 0, 0)$$

Función suma de potencias, en $\mathbb{R}^4$:

$$f_2(x_0, x_1, x_2, x_3) = x_0^2 + |x_1|^3 + x_2^4 + |x_3|^5$$

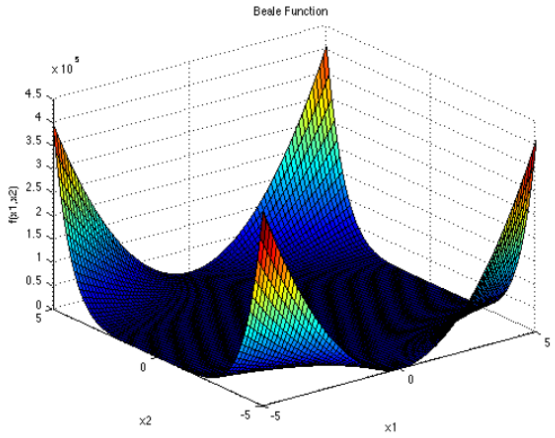
$$x^* = (0, 0, 0, 0)$$



Función de Beale, en $\mathbb{R}^2$:

$$f_3(x_0, x_1) = (1.5 - x_0 + x_0 x_1)^2 + (2.25 - x_0 + x_0 x_1^2)^2 + (2.625 - x_0 + x_0 x_1^3)^2$$

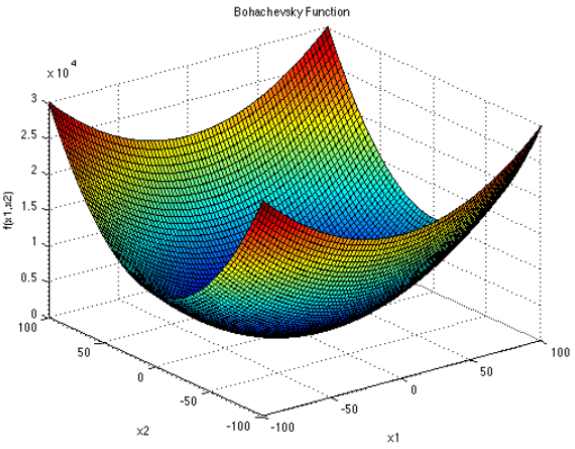
$$x^* = (3, 0.5)$$



Función de Bohachevsky, en $\mathbb{R}^2$:

$$f_4(x_0, x_1) = x_0^2 + 2x_1^2 - 0.3cos(3\pi x_0 + 4\pi x_1) + 0.3$$

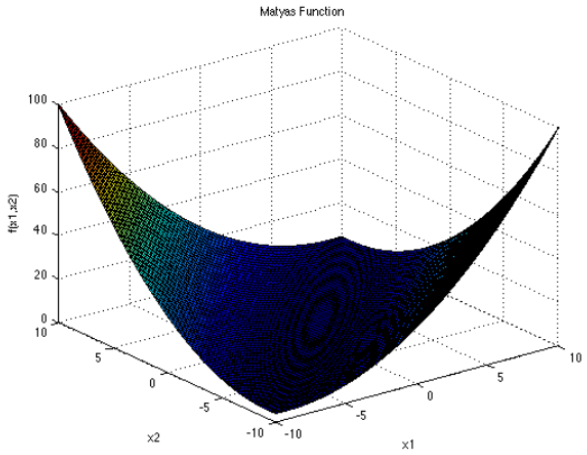
$$x^* = (0, 0)$$



Función de Matyas, en $\mathbb{R}^2$:

$$f_5(x_0, x_1) = 0.26(x_0^2 + x_1^2) - 0.48 \cdot x_0 x_1$$

$$x^* = (0, 0)$$



Función de Zakharov, en $\mathbb{R}^3$:

$$f_6(x_0, x_1, x_2) = x_0^2 + x_1^2 + x_2^2 + (0.5x_0 + x_1 + 1.5x_2)^2 + (0.5x_0 + x_1 + 1.5x_2)^4$$

$$x^* = (0, 0, 0)$$

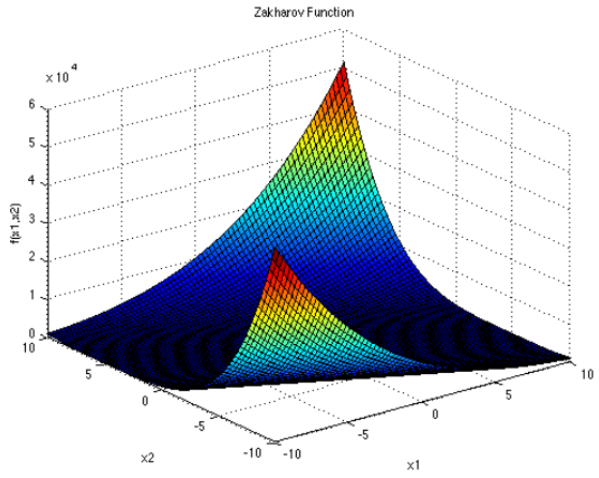


Figura 1: Función de Zakharov en $\mathbb{R}^2$

Código 3: Definición de test

```
1
2 def test():
3     testbench = np.array([
4         np.array(["Funcion Esfera R3", f1, [[10, 11, 12], [-11, -10, -15], [4, -7, 6], [-9, -7, 6]], [0,0,0]],
      ↪ dtype=object),
```

```
    np.array(["Funcion polinomio suma de diferente potencias R4", f2, [[-1, -1, -1, -1], [1, -1, -1, -1], [1, 1, -1, -1], [1, 1, 1, -1], [1, 1, 1, 1]], [0,0,0,0,0]], dtype=object),
    np.array(["Funcion Beale R2", f3, [[3, 3], [-1, -3], [4, -1.5]], [3, 0.5]], dtype=object),
    np.array(["Funcion de Bohachevsky R2", f4, [[3, 3], [-1, -2], [2, -1.5]], [0, 0]], dtype=object),
    np.array(["Funcion Matyas R2", f5, [[3, 3], [-1, -3], [4, -1.5]], [0,0]], dtype=object),
    np.array(["Funcion Zakharov R3", f6, [[-5, -5, -5], [5, 5, 5], [-5, 5, 5], [5, 3, 4]], [0,0,0]], dtype=object)], dtype=object)

tol = 1e-9  #tolerancia para algoritmos
eps = 1e-6  #tolerancia para comparacion de resultados
itmax = 5000

failed = 0
passed = 0

print("-------------------------------------")
print("TEST MINIMI()")
print("-------------------------------------")

for i in range(len(testbench)):
    print("-------------------------------------")
    print(testbench[i][0])
    print("-------------------------------------")
    x, fx, it = minimi(testbench[i][1], None, testbench[i][2][0], tol, itmax)    # minimi de la función i de testbench
    m_sp = minimize(testbench[i][1], testbench[i][2][0],method='Nelder-Mead',tol=tol,options={'maxiter':itmax})


    print("X0 function = ", testbench[i][3])
    print("X0 minimi() = ", x)
    print("X0 scipy.optimize.minimize() = ", m_sp['x'])
    print("Iterations minimi(): ", it)
    print("Iterations scipy.optimize.minimize(): ", m_sp.nit)
    if np.linalg.norm(m_sp.x - testbench[i][3]) > eps:
        print("FAILED")
        failed += 1
    else:
        print("PASSED")
        passed += 1

print("PASSED: ", passed)
print("FAILED: ", failed)

print("-------------------------------------")
print("TEST TEMPERATURA()")
print("-------------------------------------")

x, fx = temperatura()
m_sp = minimize(eje2_func, X0,method='Nelder-Mead',tol=tol,options={'maxiter':itmax})
print("X0 temperatura() = ", x)
```

```
52    print("X0 scipy.optimize.minimize() = ", m_sp['x'])
53    return
54
55 #Test: Funcion Esfera R3
56 def f1(x):
57    return x[0]**2 + x[1]**2 + x[2]**2
58
59 #Test: Funcion polinomio suma de diferente potencias R4
60 def f2(x):
61    return x[0]**2 + abs(x[1])**3 + x[2]**4 + abs(x[3])**5
62
63 #Test: Funcion Beale R2
64 def f3(x):
65    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
66
67 #Test: Funcion de Bohachevsky R2
68 def f4(x):
69    return x[0]**2 + 2*x[1]**2 - 0.3*np.cos(3*np.pi*x[0] + 4*np.pi*x[1]) + 0.3
70
71 #Test: Funcion Matyas R2
72 def f5(x):
73    return 0.26*(x[0]**2 + x[1]**2) - 0.48*x[0]*x[1]
74
75 #Test: Funcion Zakharov R3
76 def f6(x):
77    return x[0]**2 + x[1]**2 + x[2]**2 + (0.5*x[0] + x[1] + 1.5*x[2])**2 + (0.5*x[0] + x[1] + 1.5*x
       ↪ [2])**4
```