Proyecto: Modelos de Optimización, Optimización no lineal.

Integrantes:

Karen Dianelis Cantero Lopez C - 311

Chavely González Acosta C - 312

Luis Alejandro Rodríguez Otero C - 311

Algoritmos:

Se nos dió la tarea de resolver 3 problemas clásicos de optimización no lineal utilizando softwares de programación y analizando los resultados. Decidimos utilizar dos algoritmos implementados en Python para llevar a cabo dicha tarea, estos algoritmos son: El método ***BFGS*** y el método ***differential_evolution***, ambos implementados en la biblioteca ***Scipy.Optimize***. Los Problemas objeto de estudio fueron extraídos de "A Literature Survey of Benchmark Functions For Global Optimization Problems", bibliografía que se puede consultar en este repositorio.

Trid Function (Problem #150 & #151):

El primer problema se trataba de la función *Trid 6 Function*, una función de 6 dimensiones y con un único mínimo local, que a su vez es global. Para optimizar esta función utilizamos el método *minimize* de la biblioteca *Scipy.Optimize*. Esta es una función que se utiliza para minimizar una función de una o más variables y utiliza varios métodos de optimización que pueden ser especificados al llamar a dicha función. En concreto el que utilizamos nosotros fue el método *BFGS*, es un algoritmo iterativo especializado en resolver problemas de optimización no lineal sin restricciones. Escogimos este método ya que es capaz de resolver problemas con gran número de parámetros y de gran complejidad, y se caracteriza por tener convergencia superlineal, es decir, que converge a la solución en un número aceptable de iteraciones.

Para hallar el mínimo global de *Trid 6 Function* utilizamos el siguiente código de Python:

```
from scipy.optimize import minimize as min
import numpy as np

def Problem150():
    #Función Objetivo
    def func_obj(x):
        Sum1 = sum((x-1)**2)
```

```
        Sum2 = sum(x[:-1] *  x[1:])

        return Sum1 - Sum2

    #Solución inicial
    x0 = np.zeros(6)

    #Límites de las variables
    bounds = [(-36,36)] * 6

    #Minimizar
    Solution = min(func_obj, x0 = x0, bounds = bounds)

    #Mostrar los resultados
    print(f'Valor minimo de la funcion: {Solution.fun}')
    print(f'Numero de iteraciones: {Solution.nit}')

#Invocar el método de optimización
Problem150()
```

El cual tras ejecutarlo obtuvimos:

Valor mínimo de la función: **-49.99999999977126**

Número de iteraciones: **7**

Tiempo de cpu empleado: **aproximadamente 1.2 segundos**

Lo cual constituye una solución con menos del 0.00001 de error con respecto a la solución real en un número muy bajo de iteraciones.

El segundo problema se trata de la función *Trid 10 Function*, cuya única diferencia es que esta es de dimensión 10, a diferencia de la anterior que es de dimensión 6. Utilizamos una vez más el algoritmo *BFGS* por lo que se nota una gran similitud con el código utilizado:

```
from scipy.optimize import minimize as min
import numpy as np

def Problem151():
    #Función Objetivo
    def func_obj(x):
        Sum1 = sum((x-1)**2)
        Sum2 = sum(x[:-1] *  x[1:])

        return Sum1 - Sum2

    #Solución inicial
    x0 = np.zeros(10)
```

```
    #Límites de las variables
    bounds = [(-100,100)] * 10

    #Minimizar
    Solution = min(func_obj, x0 = x0, bounds = bounds)

    #Mostrar los resultados
    print(f'Valor minimo de la funcion: {Solution.fun}')
    print(f'Numero de iteraciones: {Solution.nit}')

#Invocar el método de optimización
Problem150()
```

Al ejecutarlo obtuvimos:

Valor mínimo de la función: **-209.99999996834958**

Número de iteraciones: **10**

Tiempo de cpu empleado: **aproximadamente 1.2 segundos**

Lo cual constituye una solución muy cercana a la real y tambien en un número bajo de iteraciones.

Aumentando la dimensión

Si bien *Trid 6 Function* y *Trid 10 Function* se diferencian en las dimensiones, tambien lo hacen en su valor mínimo, la función de más dimensión tiene un valor mínimo bastante menor que la otra, **¿Qué sucedería si aumentamos aún más la dimensión?**

Utilizando el mismo algoritmo probamos la *Trid Function* para dimensión 50, 100, 500 y 1000, esto fue lo que obtuvimos:

D = 50:

Valor mínimo de la función: **-22036.628544175997**

Número de iteraciones: **49**

Tiempo de cpu empleado: **aproximadamente 1.4 segundos**

D = 100:

Valor mínimo de la función: **-151276.26162570715**

Número de iteraciones: **99**

Tiempo de cpu empleado: **aproximadamente 2.3 segundos**

D = 500:

Valor mínimo de la función: **-1098666.3563336134**

Número de iteraciones: **9**

Tiempo de cpu empleado: **aproximadamente 5.5 segundos**

D = 1000:

Valor mínimo de la función: **-2604246.5334501266**

Número de iteraciones: **6**

Tiempo de cpu empleado: **aproximadamente 13 segundos**

Vemos como a medida que aumenta la dimensión disminuye el valor del mínimo de la función. Sin embargo algo curioso es que a partir de la dimensión 500 el número de iteraciones del algoritmo vuelve a ser el mismo que cuando teníamos 10 dimensiones, es decir, converge con bastante rapidez. No obstante el tiempo de cpu crece a medida que aumentan las dimensiones. Esto significa que, como explicamos anteriormente, el algoritmo *BFGS* es muy útil para resolver problemas con grandes dimensiones, la razón por la que aumenta el tiempo de cpu es porque evaluar una función de más de 100 variables es altamente costoso. Incluso hicimos una pequeña prueba con dimensión 2000, el algoritmo tardó más de 40 segundos en hallar la solución pero lo hizo en menos de 10 iteraciones.

Trefethen Function (Problem #152):

Nuestro 3er problema es la *Trefethen Function*, otro problema de optimización lineal de dimensión 2, pero no por eso más sencillo que los anteriores, de hecho fue todo lo contrario. La función es el resultado de componer varias funciones seno, por lo que tiene una cantidad bastante grande de mínimos locales **(Ver Trefethen Grafic)**. Esto constituye la debilidad más grande del algoritmo *BFGS*, tiene gran probabilidad de estancarse en mínimos locales. Al principio intentamos usar este algoritmo para la *Trefethen Function* pero vimos que dependía mucho del valor inicial que escogíamos, daba un valor distinto para todos los valores iniciales y solo coincidía con el valor real si el valor inicial era muy cercano al mínimo. Por esto nos vimos obligados a recurrir a un algoritmo diferente, *differential_evolution*.

El algoritmo *differential_evolution* es un algoritmo de optimización global que, a diferencia de otros métodos de optimización, es un método estocástico que no utiliza métodos de gradiente para encontrar el mínimo. En su lugar, el algoritmo muta cada solución candidata mezclándola con otras soluciones para crear distintas soluciones candidatas de prueba. Es muy útil para problemas de muchos mínimos locales ya que puede buscar en grandes áreas del espacio de candidatos.

El principal problema de este algoritmo es que es muy costoso, no requiere ni siquiera un punto inicial por lo que desde el principio está buscando soluciones

en casi toda la función, lo que requiere una cantidad grande de evaluaciones de la función objetivo y de iteraciones. Para utilizar este algoritmo hicimos lo siguiente:

```
from scipy.optimize import differential_evolution
from math import sin, e

def Problem152():
    #Función Objetivo
    def func_obj(x):
        return e ** sin(50 * x[0]) + sin(60 * e ** x[1]) + sin(70 * sin(x[0])) + sin(sin(80

    #Límites de las variables
    bounds = [(-10, 10), (-10, 10)]

    #Minimizar
    Solution = differential_evolution(func_obj, bounds)
    x1 = Solution.x[0]
    x2 = Solution.x[1]

    #Mostrar los resultados
    print(f'Minimo Global en:{x1}, {x2}')
    print(f'Valor minimo de la funcion: {func_obj([x1,x2])}')
    print(f'Numero de iteraciones: {Solution.nit}')


#Invocar el método de optimización
Problem152()
```

Al ser un método basado en metaheurísticas no siempre devuelve exactamente la misma solución para el mismo problema, pero si muy cercanas a la solución real. Ejecutamos el algoritmo unas 10 veces, en todas obtuvimos valores de menos de 0.3 de error con respecto al valor real y en 3 ocaciones obtuvimos soluciones que coincidían totalmente con la solución real, eso si, el algoritmo osciló entre las 40 y 80 iteraciones para hallar las soluciones, y empleó un tiempo de cpu de aproximadamente 2 segundos en cada ejecución.

Todos los resultados pueden ser comprobados ejecutando los archivos py que se encuentran en este repositorio

Bibliografía:

A Literature Survey of Benchmark Functions For Global Optimization Problems.

https://docs.scipy.org: Optimization and root finding (scypi.optimize).

https://machinelearningmastery.com: Differential Evolution Global Optimization With Python.