Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías

Licenciatura en Ingeniería en Computación

Materia: Seminario de Solución de Problemas de Inteligencia Artificial I. Clave: I7039.

Profesor: Sencion Echauri Felipe

Estudiante: Silva Moya José Alejandro. Código: 213546894.

Actividad 7: Ant Colony Optimization

Instrucciones: Implementar y evaluar el rendimiento del algoritmo de optimización por colonia de hormigas para las siguientes funciones.

- Sphere
- Rosenbrock
- Rastrigin
- Quartic

Para cada función realizar 5 ejecuciones con 2, 4, 8 y 16 dimensiones, cada ejecución se detendrá a las 2000 generaciones.

Se deberá graficar el comportamiento del algoritmo; para ello se deberá promediar el valor del mejor fitness de las 5 ejecuciones en la generación 0, 100, 200, ... 2000. Se deberá generar una gráfica para cada dimensión y además una gráfica en la que se incluyan las ejecuciones para 2, 4, 8 y 16 dimensiones, es decir un total de 5 gráficas por función.

Desarrollo

En este caso, los agregados que se pusieron en el código principal de la colonia de hormigas fueron menores. Solamente se agregó una variable para obtener el mejor de cada 100 generaciones, y un arreglo de compendio de todos esos valores.

```
69          ###
70          graphData = 0   #El dato que obtenemos de cada 100 generaciones
71          graphArray = np.array([])    #Array donde almacenamos cada graphData
72          ###
```

Y posteriormente se acumulan todas para retornar el arreglo.

```
94              ###
95              if generacion%self._f == 0:
96                  graphData = self._best._L
97                  graphArray = np.append(graphArray, [graphData])
98              ###
99              generacion += 1
100             print('generación: ', generacion, 'Mejor histórico: ', self._best._x, self._best._L)
101         ###
102         return graphArray
103         ###
```

Con esto, haremos las 5 iteraciones necesarias para cada dimensión, y podremos promediar todo en un mismo txt en lugar de 5, como se hizo en la entrega pasada; es decir, mejoré esa parte.

Además de esto, se agregó una variable al constructor para poder manera más fácilmente cada cuándo tomar el mejor valor (en este caso es cada 100 generaciones, y viene representado con la variable self._f de la imagen anterior.

```
21  class ACO:
22      def __init__(self, N, n, B, a, Q, p, t0, problema, g, f):
```

```
62          #Criterio de cada cuando obtener el mejor historico
63          self._f = f
```

Esto termina las modificaciones del código principal.

```
10      ros = rosenbrock.Rosenbrock()
11      sph = sphere.Sphere()
12      qua = quartic.Quartic()
13      ras = rastrigin.Rastrigin()
14
15      aux = np.array([])
16      graph = np.array([])
17      ejecuciones = 5
18      draw = drawer.Drawer()
```

En el main generamos un objeto de cada problema a resolver (cuyas clases veremos más adelante), agregamos un arreglo auxiliar y uno llamado “graph”, que serán los utilizados para cargar todos los datos obtenidos de cada 100 generaciones, una suma y su respectivo promedio; para el promedio se usa la variable “ejecuciones”, y el objeto “draw” servirá para facilitar el graficar los resultados.

```
21      individuos = 32
22      dimensiones = 2
23      intervalos = 8
24      a = 1
25      Q = 20
26      evaporacion = 0.9
27      t0 = 0.00001
28      generaciones = 2000
29      bestFitnessPos = 100
```

Los valores de generación de los objetos no cambiaron, pero se agregó la variable “bestFitnessPos”, que es el valor de self._f.

```
32      graphName = "Quartic" + str(dimensiones) + "D"
33      aco = ACO.ACO(individuos,
34                    dimensiones,
35                    intervalos,
36                    a,
37                    Q,
38                    evaporacion,
39                    t0,
40                    qua,
41                    generaciones,
42                    bestFitnessPos)
43
44      for i in range(ejecuciones):
45          aux = aco.run()
46          if i == 0:
47              graph = aux
48          else:           #Sumamos el resto de las ejecuciones.
49              for a in range(len(aux)):
50                  graph[a] = graph[a] + aux[a]
51
52      for x in range(len(graph)):
53          graph[x] = graph[x]/ejecuciones    #Obtenemos el promedio de cada posicion
54
55      draw.drawIndividual(graph, graphName)
56      file = open(graphName + ".txt", "w")
57      for y in range(len(graph)):
58          file.write(str(graph[y]) + "\n")
59      file.close()
60      #draw.drawGroup("Quartic")
```

Para evitar tener que hacer tantas cosas a mano, traté de automatizar todo lo posible. Entonces lo que se hace aquí es dar una base de nombre para los archivos de cada ejecución, agregar el fitnesspos a la instancia del objeto, y comenzar con todo el algoritmo propiamente.

Para cada ejecución que pedimos (en este caso 5), obtenemos cada resultado y los vamos sumando en su respectiva posición en un arreglo, para al final promediar entre el mismo valor de ejecuciones, y obtener el vector final que necesitamos para graficar la ecuación. Posteriormente la mandamos graficar (en breve veremos ese código), y lo guardamos en un txt porque lo vamos a necesitar para la gráfica que tiene el promedio de los mejores para cada una de las 4 dimensiones.

Finalmente tenemos una línea que grafica el grupo de todos los resultados de las dimensiones en una sola imagen, pero debido a ciertos problemas, tuve que comentarlo y hacer por separado las gráficas individuales, y posteriormente la grupal, para cada uno de los 4 casos de ecuaciones.

```python
import matplotlib.pyplot as plt
import numpy as np

class Drawer:

    def __init__(self):
        pass

    def drawIndividual(self, array, title):
        plt.plot(array, 'ro', array, 'b')
        plt.axis([0,20,0,1000]) #xmin, xmax, ymin, ymax
        plt.ylabel('Evaluaciones')
        plt.xlabel('Generaciones')
        plt.title(title)
        plt.savefig(title + '.png', dpi=None, facecolor='w', edgecolor='w',
                orientation='portrait', papertype=None, format='png',
                transparent=False, bbox_inches=None, pad_inches=0.1,
                frameon=None, metadata=None)
        plt.show()
```

Para graficar cada resultado individual, lo que hacemos es pasarle el arreglo de datos a graficar y su respectivo título; le damos colores y valores a los ejes, así como títulos, y antes de mostrar el resultado nos aseguramos de guardarlo, y automatizamos ese proceso.

```
21      def drawGroup(self, title):
22          a1 = a2 = a3 = a4 = np.array([])
23          dimensionsArray = np.array([2,4,8,16])
24
25          for i in range(4):
26              file = open(title + str(dimensionsArray[i]) + "D" + ".txt", "r")
27              data = file.read().splitlines()
28              file.close()
29              aux = np.array([])
30              for a in range(len(data)):
31                  num = float(data[a])
32                  aux = np.append(aux, [num])
33              if i == 0:
34                  a1 = aux
35              if i == 1:
36                  a2 = aux
37              if i == 2:
38                  a3 = aux
39              if i == 3:
40                  a4 = aux
41
42          plt.plot(a1, 'ro', a1, 'b',
43                   a2, 'ro', a2, 'b',
44                   a3, 'ro', a3, 'b',
45                   a4, 'ro', a4, 'b')
46          plt.axis([0,20,0,1000])
47          plt.ylabel('Evaluaciones')
48          plt.xlabel('Generaciones')
49          plt.title(title)
50          plt.savefig(title + '.png', dpi=None, facecolor='w', edgecolor='w',
51                  orientation='portrait', papertype=None, format='png',
52                  transparent=False, bbox_inches=None, pad_inches=0.1,
53                  metadata=None)
54          plt.show()
```

Para poder graficar el conjunto de dimensiones de una sola ecuación, el proceso es bastante similar; lo que cambia es que primero tenemos que leer los archivos de texto que contienen los valores necesarios, agregarlos a arreglos, y finalmente graficar. Con esto, el código estaría terminado. Ahora solo resta mostrar las implementaciones de las ecuaciones que faltaban.

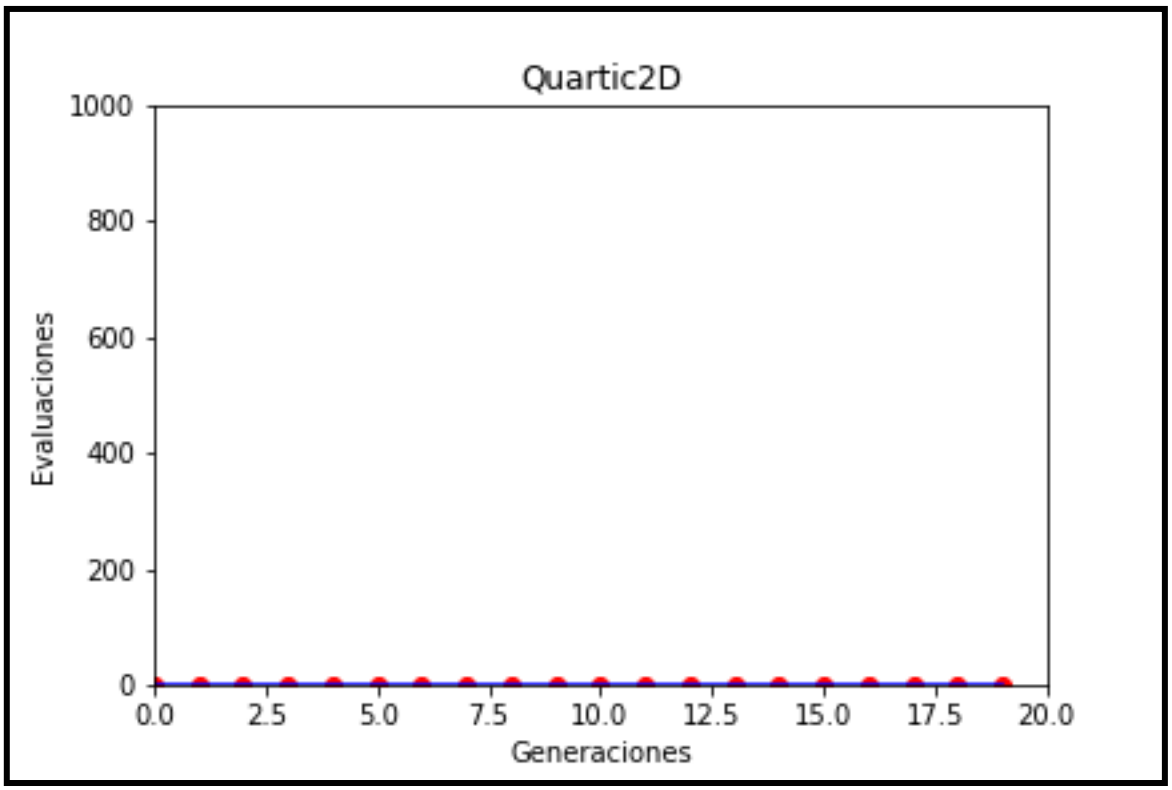
```python
class Quartic:
    MIN_VALUE = -1.28
    MAX_VALUE = 1.28

    def __init__(self):
        pass

    def fitness(self, vector):
        z = 0

        for dimension in range(len(vector)):
            z += dimension*(vector[dimension]**4)

        return z
```

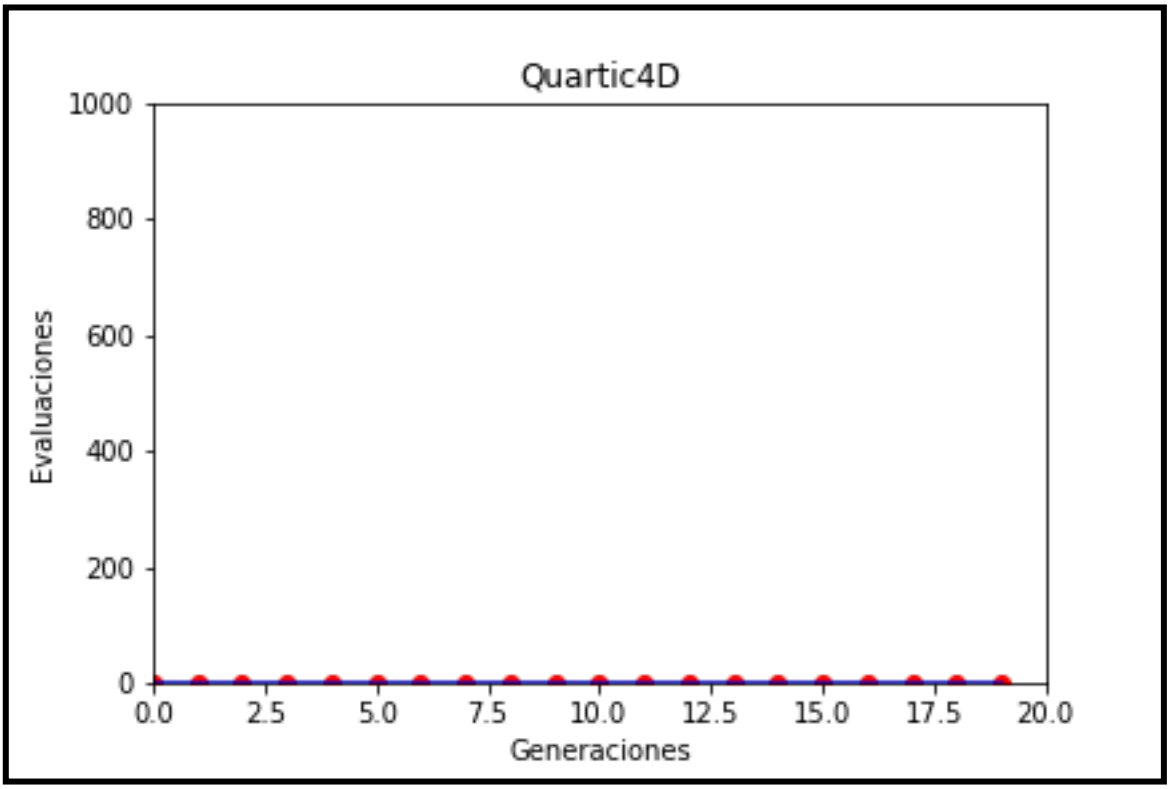
```python
import math

class Rastrigin:

    MIN_VALUE = -5.12
    MAX_VALUE = 5.12

    def __init__(self):
        pass

    def fitness(self, vector):
        z = 0

        for dimension in range(len(vector)):
            z += vector[dimension]**2 - (10*math.cos(2*math.pi*vector[dimension]))
        z += 10*(len(vector))

        return z
```

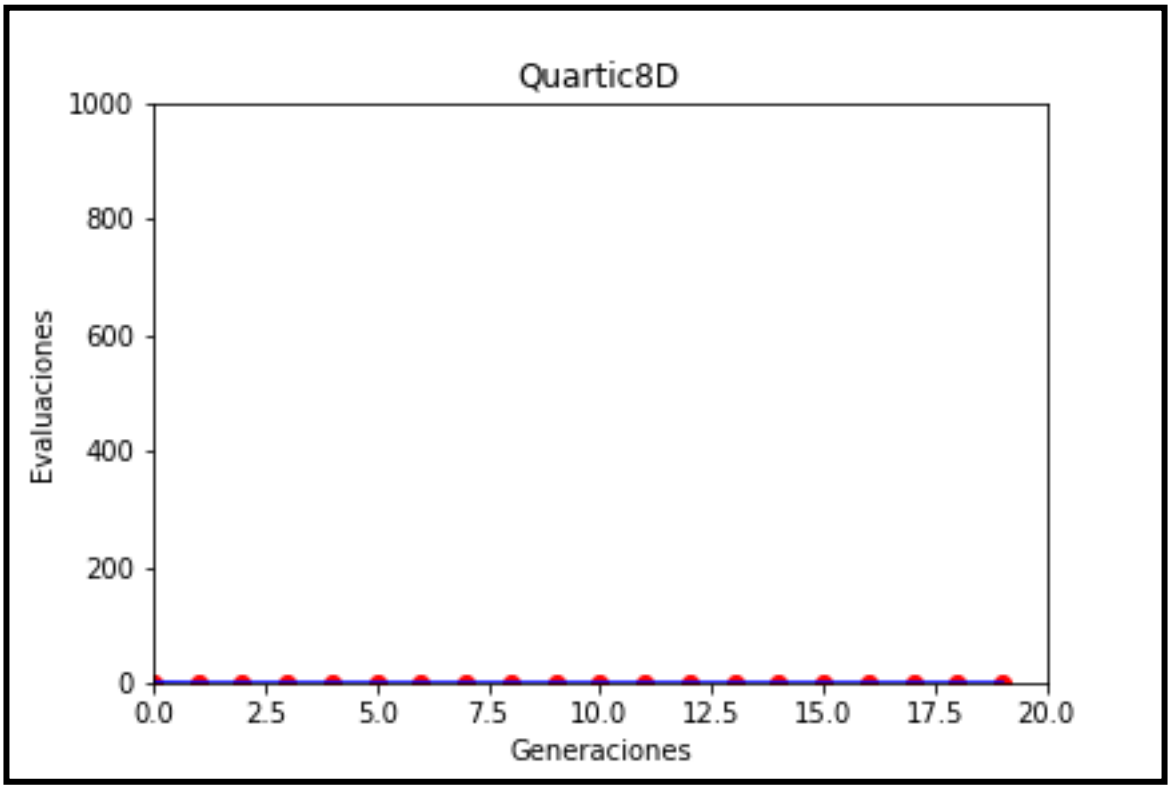
```python
class Sphere:
    MIN_VALUE = -5.12
    MAX_VALUE = 5.12

    def __init__(self):
        pass    #Nothing happens, but we need the constructor.

    def fitness(self, vector):
        z = 0

        for dimension in range(len(vector)):
            z += (vector[dimension]**2)

        return z
```

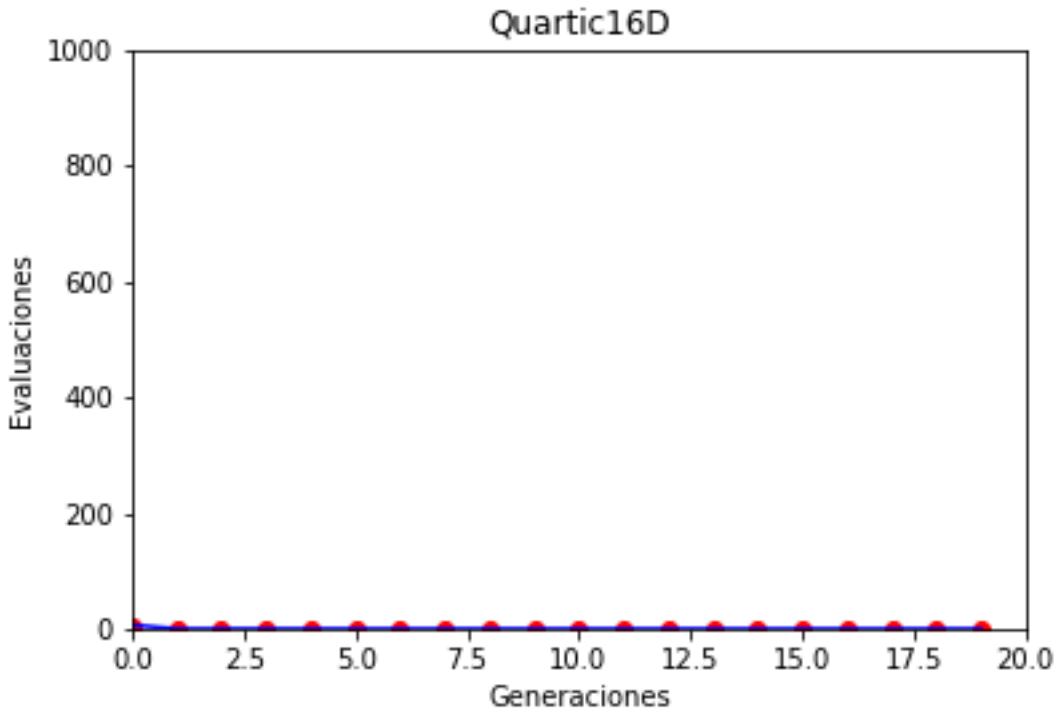
Resultados obtenidos

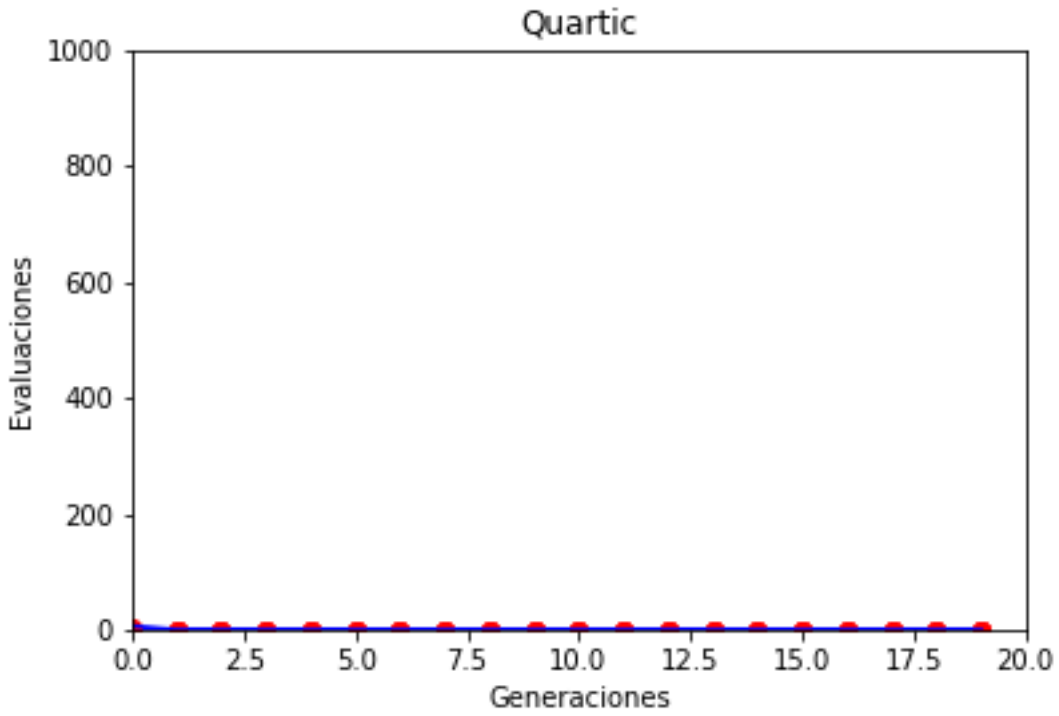
Quartic



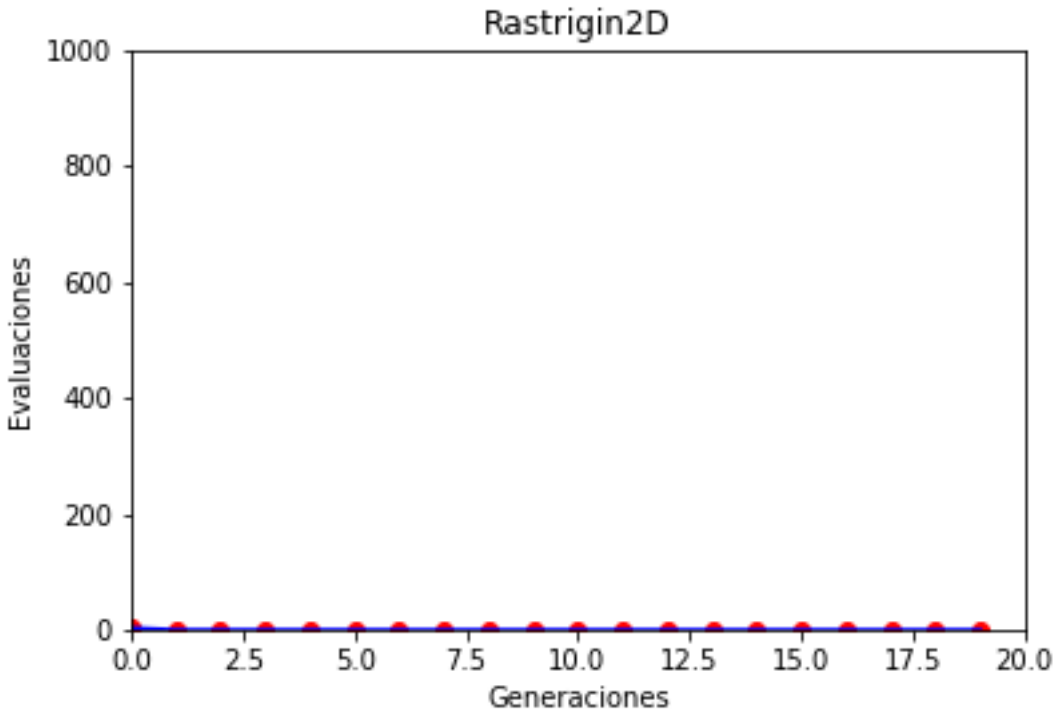

Quartic4D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

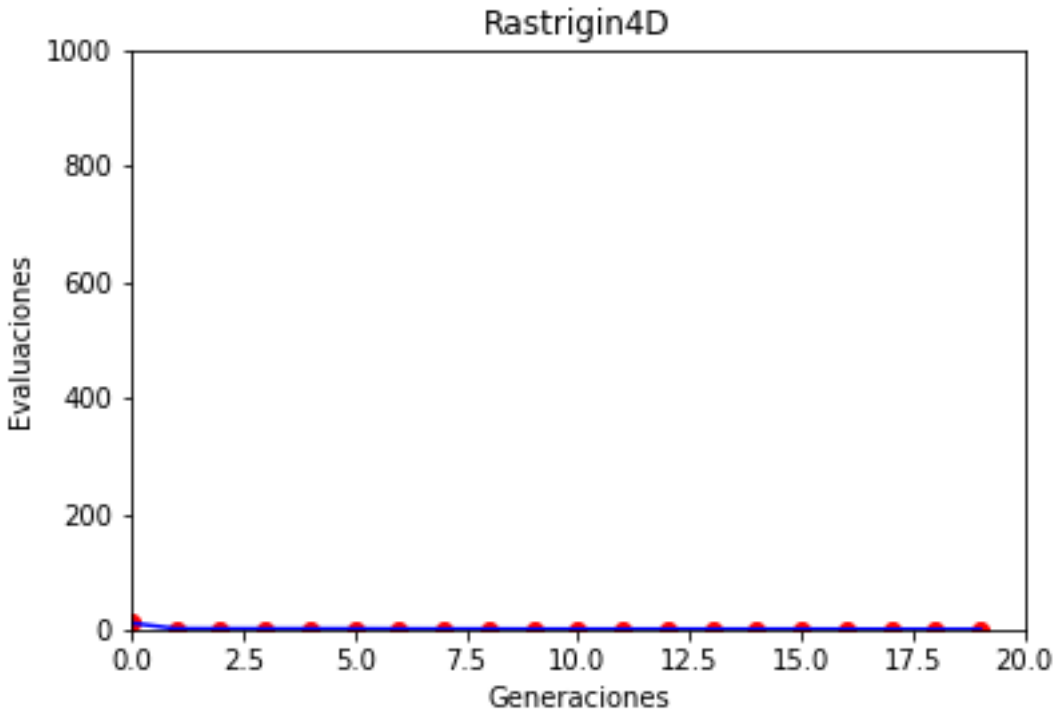

Quartic8D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

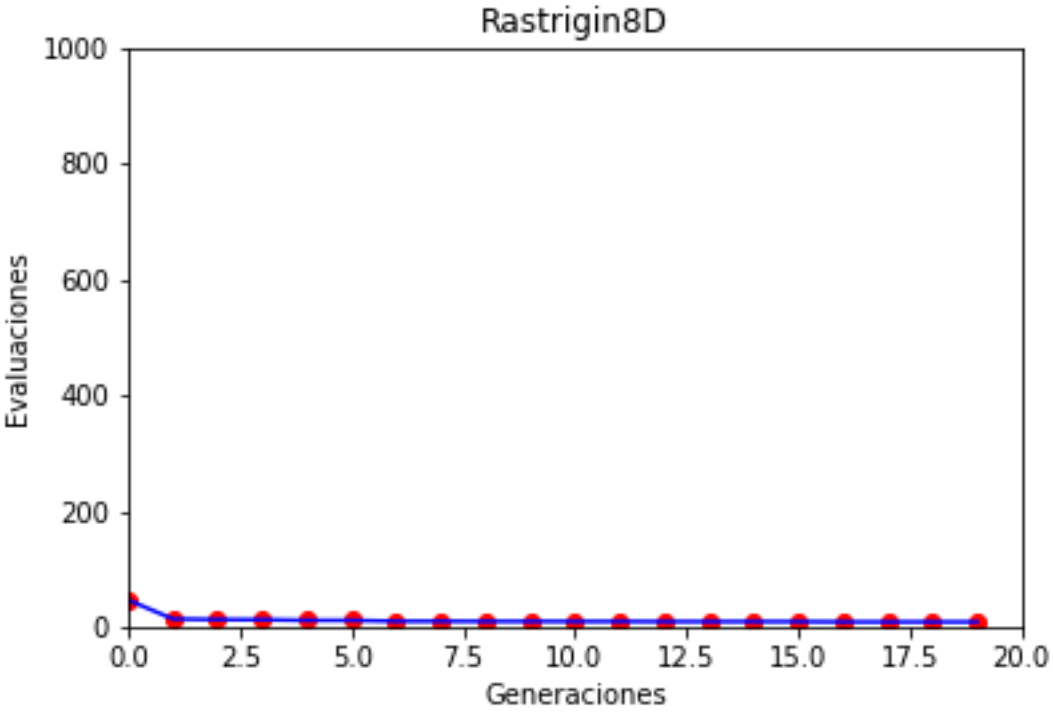

Quartic16D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

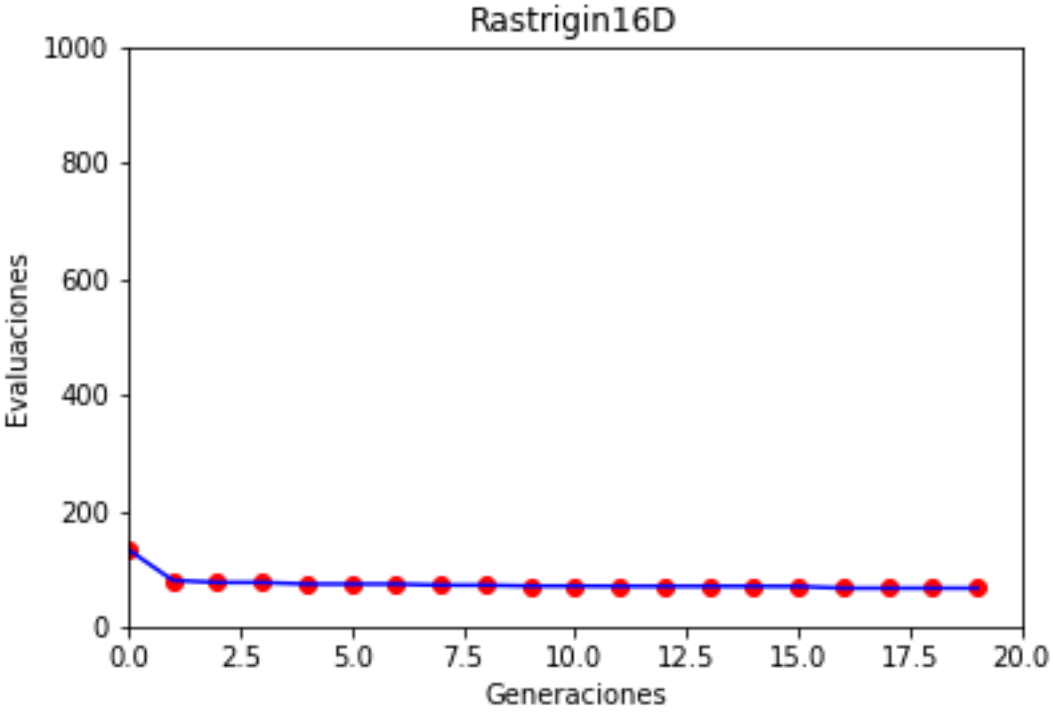

Quartic
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

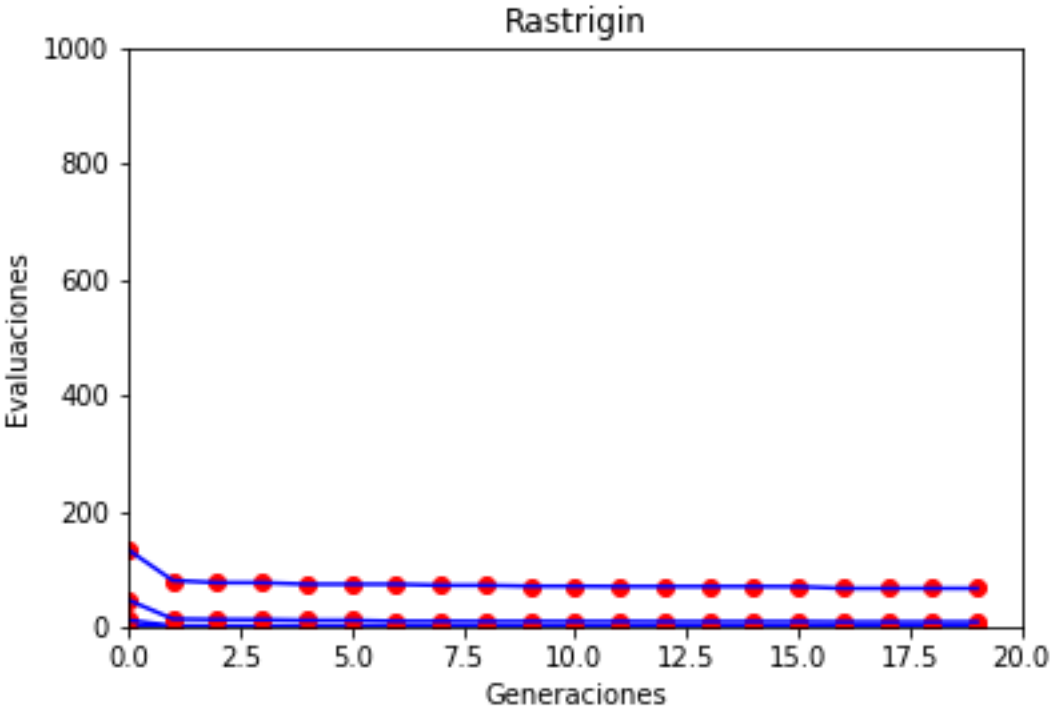
Rastrigin


Rastrigin2D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones


Rastrigin4D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

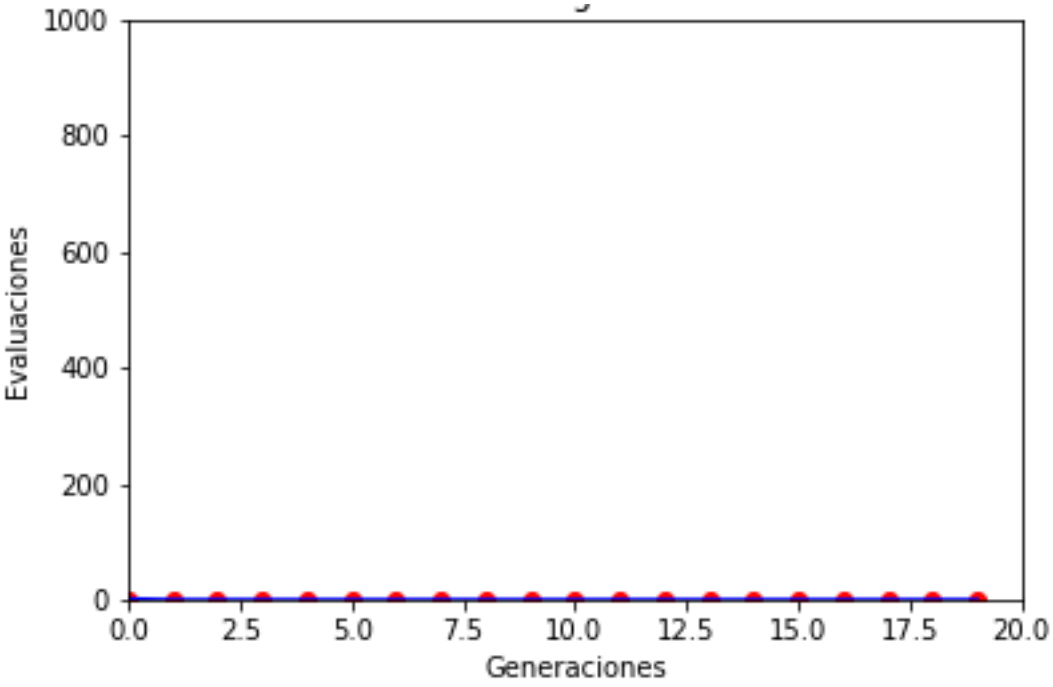

Rastrigin8D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones


Rastrigin16D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

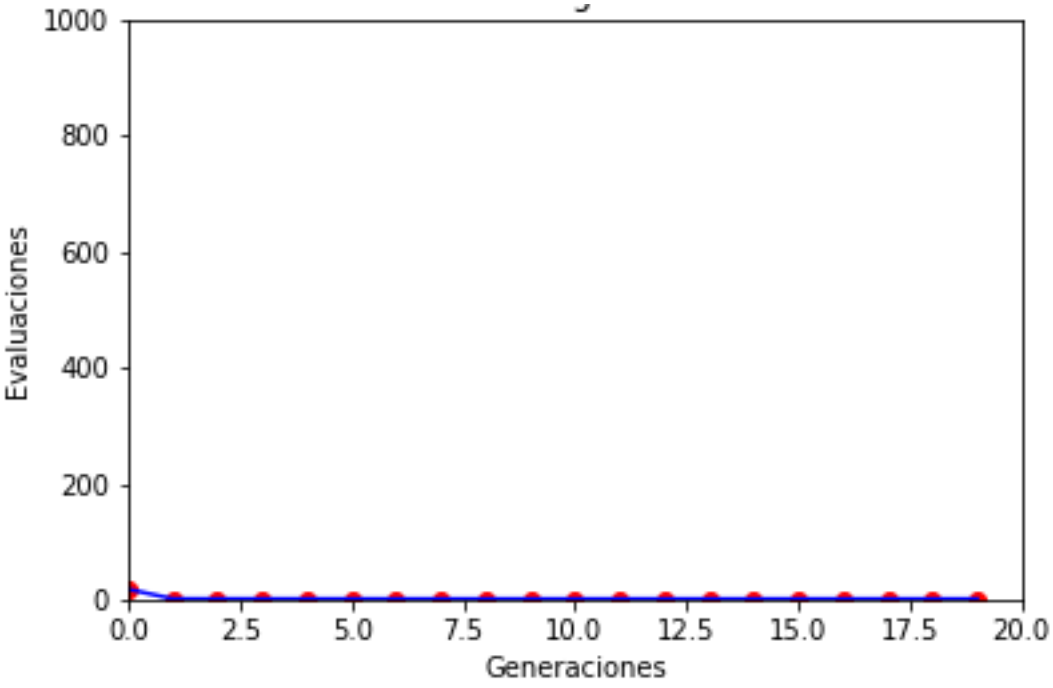

Rastrigin
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

Rosenbrock

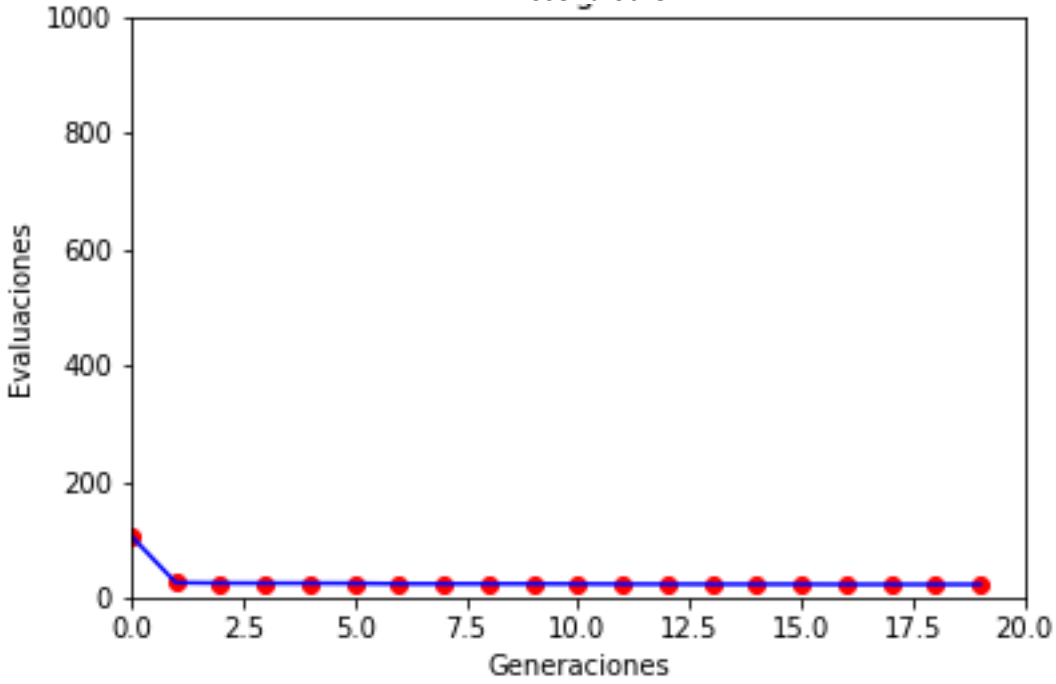
Rosenbrock2D



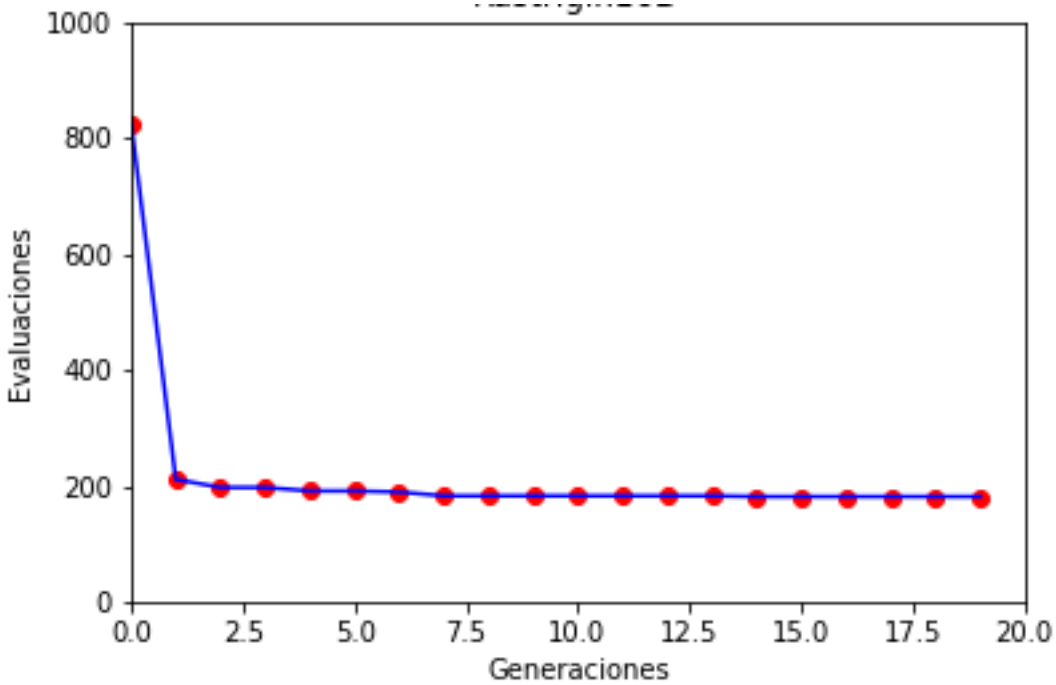
Rosenbrock4D

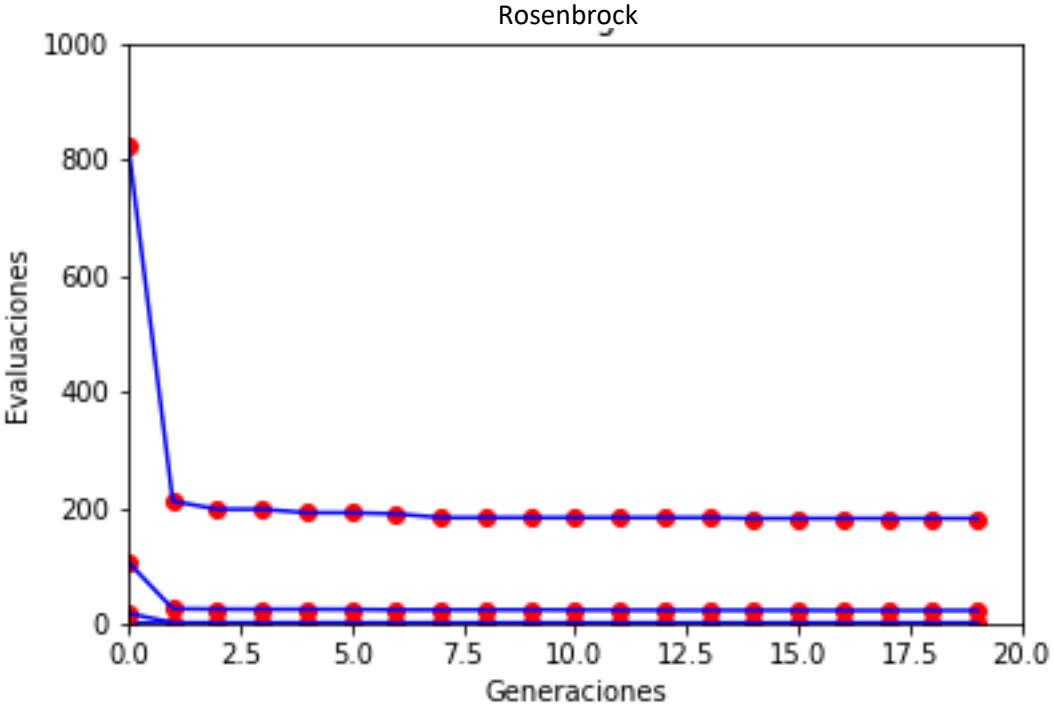


Rosenbrock8D

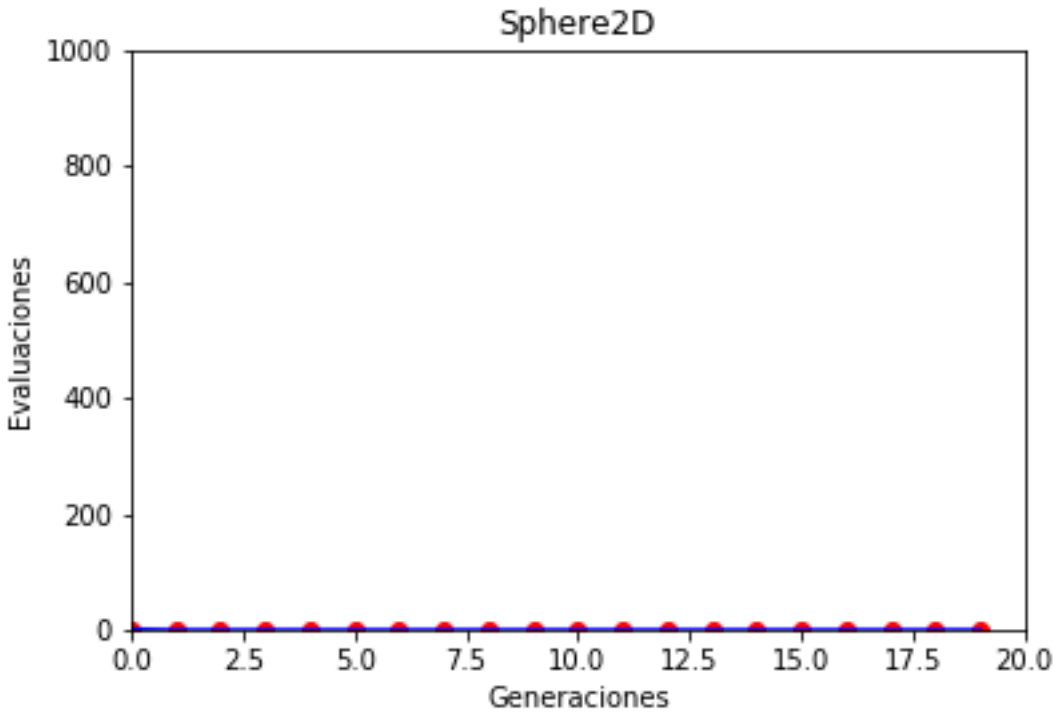


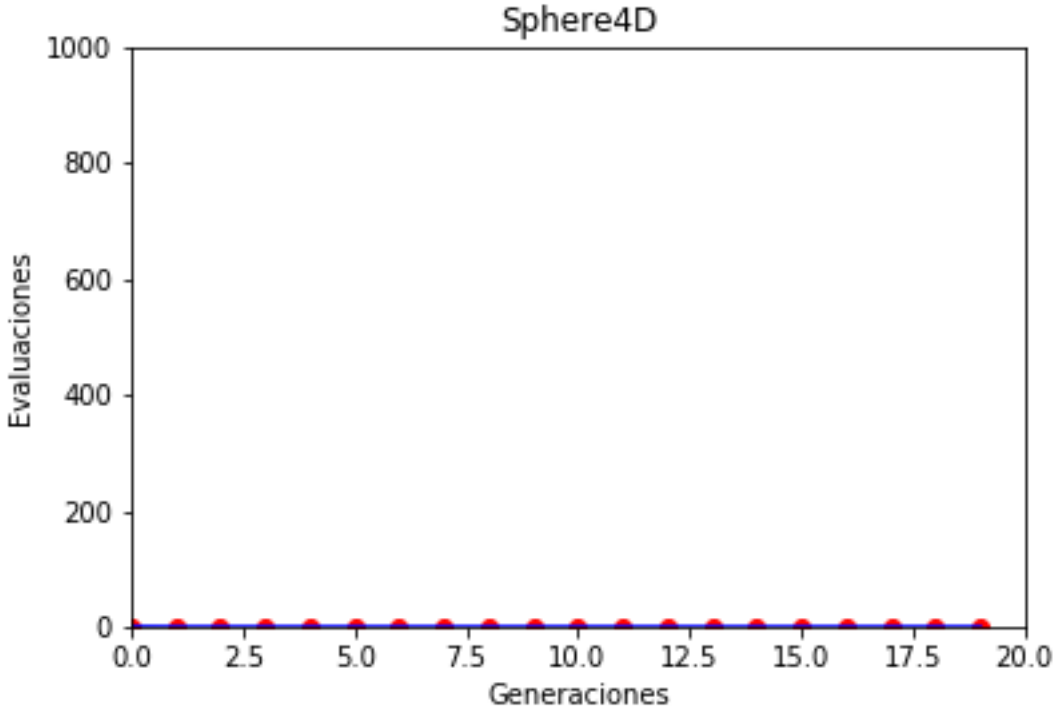
Rosenbrock16D

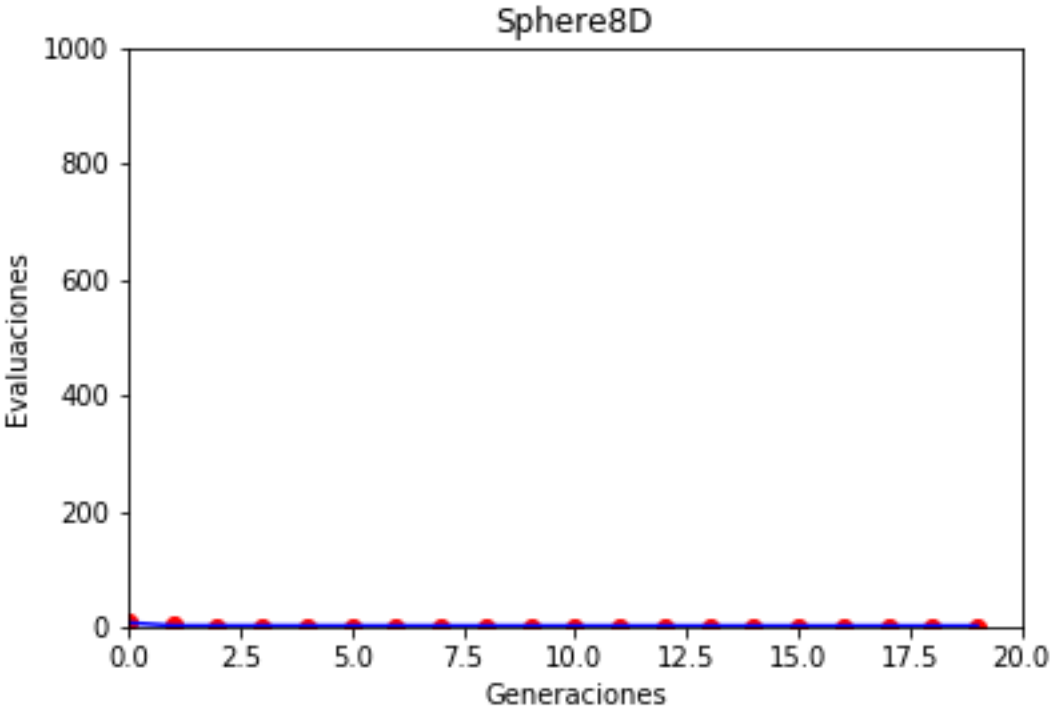


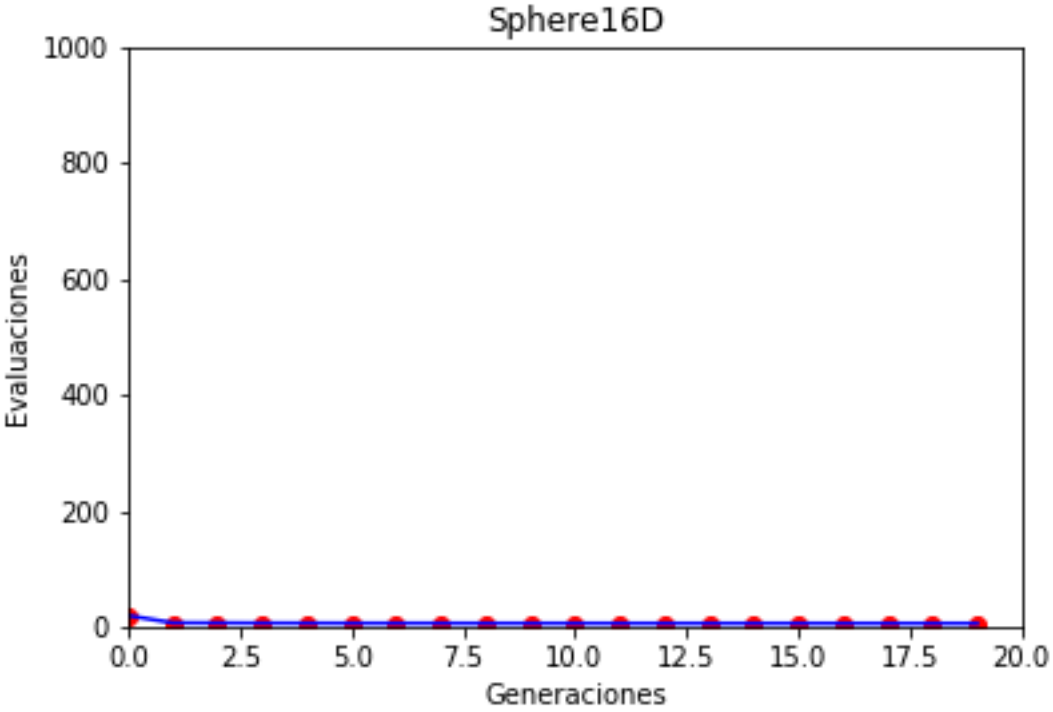

Rosenbrock
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

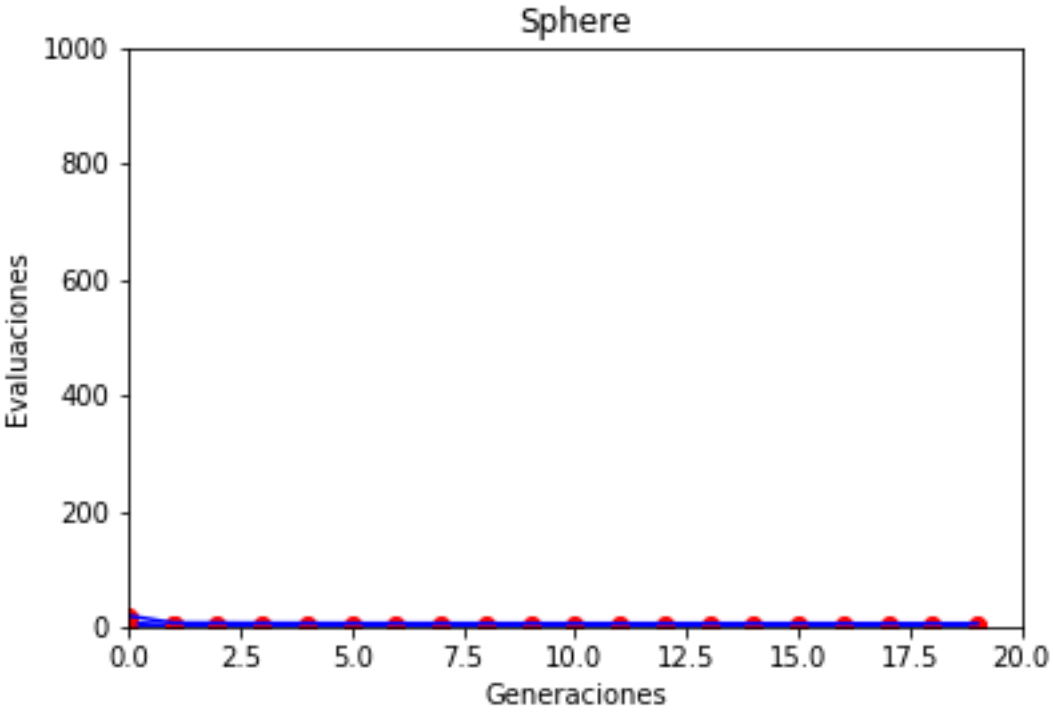
Sphere


Sphere2D
Evaluaciones
Generaciones
0
200
400
600
800
1000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Sphere4D
Evaluaciones
Generaciones
0
200
400
600
800
1000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Sphere8D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones


Sphere16D
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones


Sphere
1000
800
600
400
200
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

Conclusiones

Si bien el algoritmo funciona y cumple con su propósito, resulta importante considerar que existe bastante espacio para mejoras, debido a que la complejidad aumenta bastante por los ciclos anidados, sin mencionar la cantidad de iteraciones, generaciones e individuos.

Resulta bastante importante entender cómo funciona el código en su totalidad, para poder aplicarlo correctamente, mejorarlo, e incluso idear nuevas soluciones en otros algoritmos. A fin de cuentas, por más tardado que sea, funciona bastante bien y nos entrega mejores resultados que los algoritmos previos, así que en general es bastante bueno, y aquí se nota más el acercamiento a la emulación de la naturaleza.