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 8: Particle Swarm Optimization

Instrucciones: Implementar y evaluar el rendimiento del algoritmo de optimización por enjambre de partículas 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

```python
import random
import copy
import sys
import numpy as np

class Particle:
  def __init__(self, dimensions, ecuation, seed):
    #Inicializamos en cero los vectores de los individuos.
    self.rnd = random.Random(seed)
    self.position = [0.0 for i in range(dimensions)]   #Posicion de la particula, correspondiente a cada dimension.
    self.velocity = [0.0 for i in range(dimensions)]   #Velocidad de la particula, correspondiente a cada dimensaion.
    self.best_part_pos = [0.0 for i in range(dimensions)]  #Mejor posicion individual de la particula en cada iteracion.

    #Le damos un valor aleatorio a cada posicion de velocidad y localizacion de la particula.
    for i in range(dimensions):
      self.position[i] = ((ecuation.MAX_VALUE - ecuation.MIN_VALUE) * self.rnd.random() + ecuation.MIN_VALUE)
      self.velocity[i] = ((ecuation.MAX_VALUE - ecuation.MIN_VALUE) * self.rnd.random() + ecuation.MIN_VALUE)

    #self.fitness = fitness(self.position) #Fitness actual.
    self.fitness = ecuation.fitness(self.position)
    self.best_part_pos = copy.copy(self.position)
    self.best_part_err = self.fitness #Mejor fitness actual.

```

Lo primero que realizamos es el constructor de nuestras partículas, en donde especificamos que cada una de ellas tendrá un vector de posiciones correspondiente a cada dimensión para la cual se presente el problema, así como un vector de velocidad de la misma naturaleza, y por ende, un vector que tenga siempre la mejor posición obtenida por parte de la partícula.

Los vectores de posición y velocidad son posteriormente reinicializados con valores aleatorios considerablemente pequeños, o por lo menos dentro de valores aceptables dentro de nuestro rango de búsqueda de solución.

Finalmente, debemos obtener un fitness inicial de cada partícula para poder hacerlas funcionar, y de la misma manera, sobre escribir los valores de su mejor posición y fitness.

```python
def PSOrun(max_iterations, n, dimensions, ecuation, weight, C_L, S_L):
  rnd = random.Random(0)
  swarm = [Particle(dimensions, ecuation, i) for i in range(n)]   #Crea tantas particulas como le indiquemos.

  best_swarm_pos = [0.0 for i in range(dimensions)]
  best_swarm_fitness = sys.float_info.max #Mejor fitness del enjambre.
  for i in range(n): #Para cada particula.
    if swarm[i].fitness < best_swarm_fitness:   #Si el fitness de una particula es mejor que el global, actualizamos.
      best_swarm_fitness = swarm[i].fitness
      best_swarm_pos = copy.copy(swarm[i].position)

  iterations = 0
  w = weight    #Inercia de la velocidad.
  c1 = C_L #Proporcion de aprendizaje cognitivo.
  c2 = S_L #Proporcion de aprendizaje social.

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

```

Antes de comenzar con el ciclo iterativo del algoritmo principal aún necesitamos delimitar algunos valores. Primero, necesitamos obtener nuestro enjambre inicial, con tantas partículas como el usuario nos indique. Posteriormente, debemos delimitar valores del mejor fitness inicial (aunque en ese momento solo será un valor, mas no precisamente el mejor) para poder tener algo con qué trabajar en las iteraciones. Si el fitness y de una partícula generada es mejor que el actual global, actualizamos el fitness global y posición. Finalmente obtenemos por parámetro los valores de la inercia, y los aprendizajes cognitivos y sociales.

```python
  while iterations < max_iterations:
    if iterations % 100 == 0:
        graphData = best_swarm_fitness
        graphArray = np.append(graphArray, [graphData])

    print("Iteration: ", iterations, " ", best_swarm_pos, best_swarm_fitness)

    for i in range(n): #Procesamos cada particula
      for k in range(dimensions):
        r1 = rnd.random()
        r2 = rnd.random()

        #Actualizamos la velocidad de la particula.
        swarm[i].velocity[k] = ((w * swarm[i].velocity[k]) + (c1 * r1 * (swarm[i].best_part_pos[k] -
          swarm[i].position[k])) + (c2 * r2 * (best_swarm_pos[k] - swarm[i].position[k])))

        #Nos aseguramos de que las particulas no se salgan del dominio de busqueda de la solucion.
        if swarm[i].velocity[k] < ecuation.MIN_VALUE:
          swarm[i].velocity[k] = ecuation.MIN_VALUE
        elif swarm[i].velocity[k] > ecuation.MAX_VALUE:
          swarm[i].velocity[k] = ecuation.MAX_VALUE
        #Para cualquier limite (positivo o negativo) que rebasen, lo corregimos.
```

Para cada iteración (o generación):

- Cada 100 generaciones obtenemos el mejor fitness.
- Imprimimos la mejor posición obtenida y su fitness correspondiente.
- Para cada partícula:
    - Para cada dimensión:
        - Obtenemos un valor pseudorandom controlado con base en la inercia y velocidad actual, y un valor random con relación a cada tipo de aprendizaje y la posición actual de la partícula.

- Como es posible que la partícula se mueva a valores fuera de nuestro rango de búsqueda, necesitamos corregirla. Si la partícula se mueve a un valor mejor que el mínimo de nuestro rango de solución, automáticamente le damos el valor mínimo por defecto; y si se sale en un valor positivo, le damos el valor máximo predeterminado. Cuando ocurre esto, quiere decir que nuestra partícula ya encontró un punto mínimo en la ecuación, pero probablemente las iteraciones aún no terminan.

```
67        #Actualizamos la posición de acuerdo a la velicidad.
68        for k in range(dimensions):
69          swarm[i].position[k] += swarm[i].velocity[k]
70
71        #Calculamos el fitness de la nueva posicion.
72        #swarm[i].fitness = fitness(swarm[i].position)
73        swarm[i].fitness = ecuation.fitness(swarm[i].position)
74
75        #Verificamos si la nueva posicion es un nuevo mejor para la particula.
76        if swarm[i].fitness < swarm[i].best_part_err:
77          swarm[i].best_part_err = swarm[i].fitness
78          swarm[i].best_part_pos = copy.copy(swarm[i].position)
79
80        #Verificamos si la nueva posicion es un mejor global.
81        if swarm[i].fitness < best_swarm_fitness:
82          best_swarm_fitness = swarm[i].fitness
83          best_swarm_pos = copy.copy(swarm[i].position)
84
85      iterations += 1
86    return graphArray
```

- Posteriormente actualizamos la posición de la partícula una vez calculada su nueva velocidad. Esto debe ser efectivo para cada una de las dimensiones en que la partícula se está moviendo.
- Calculamos el nuevo fitness de la partícula que estamos trabajando.
- Si el nuevo fitness de esa partícula es mejor que su mejor actual, actualizamos su nuevo fitness y su nueva posición.
- Si el nuevo fitness de esa partícula es mejor que el mejor del enjambre, actualizamos el nuevo fitness y la nueva posición mejor del enjambre con los datos de la partícula.
- Aumentamos las iteraciones.

Retornamos el arreglo que tiene el mejor fitness del enjambre cada 100 iteraciones, para poder graficar los resultados.

```python
import PSO
import rastrigin
import quartic
import sphere
import rosenbrock
import drawer
import numpy as np

def main():
    ras = rastrigin.rastrigin()
    qua = quartic.quartic()
    sph = sphere.sphere()
    ros = rosenbrock.rosenbrock()

    aux = np.array([])
    graph = np.array([])
    ejecuciones = 5
    draw = drawer.Drawer()

    dimensions = 16
    num_particles = 50
    max_iterations = 2000    #Generaciones
    weight = 0.729    #Inercia de la velocidad.
    cognitive_learning = 1.49445 #Proporcion de aprendizaje cognitivo.
    social_learning = 1.49445 #Proporcion de aprendizaje social.

    graphName = "Sphere" + str(dimensions) + "D"
```

El main es bastante similar al de las entregas anteriores, dado que por ser un código bastante modular, por ende es bastante adaptable. Tenemos 4 objetos correspondientes a las ecuaciones que queremos resolver, arreglos en donde iremos obteniendo los valores a graficar, la cantidad de ejecuciones de cada ecuación para obtener resultados, y un objeto de nuestro graficador, que es exactamente el mismo código que el anterior.

Ahora tenemos número de partículas, inercia, aprendizaje cognitivo y aprendizaje social. El resto del código, así como el fitness de las ecuaciones son los mismos.

```
    for i in range(ejecuciones):
        aux = PSO.PSOrun(max_iterations, num_particles, dimensions, ras, weight,
                         cognitive_learning, social_learning)
        if i == 0:
            graph = aux
        else:           #Sumamos el resto de las ejecuciones.
            for a in range(len(aux)):
                graph[a] = graph[a] + aux[a]

    for x in range(len(graph)):
        graph[x] = graph[x]/ejecuciones    #Obtenemos el promedio de cada posicion

    draw.drawIndividual(graph, graphName)

    file = open(graphName + ".txt", "w")
    for y in range(len(graph)):
        file.write(str(graph[y]) + "\n")
    file.close()
     #draw.drawGroup("Sphere")

if __name__ == '__main__':
    main()
```

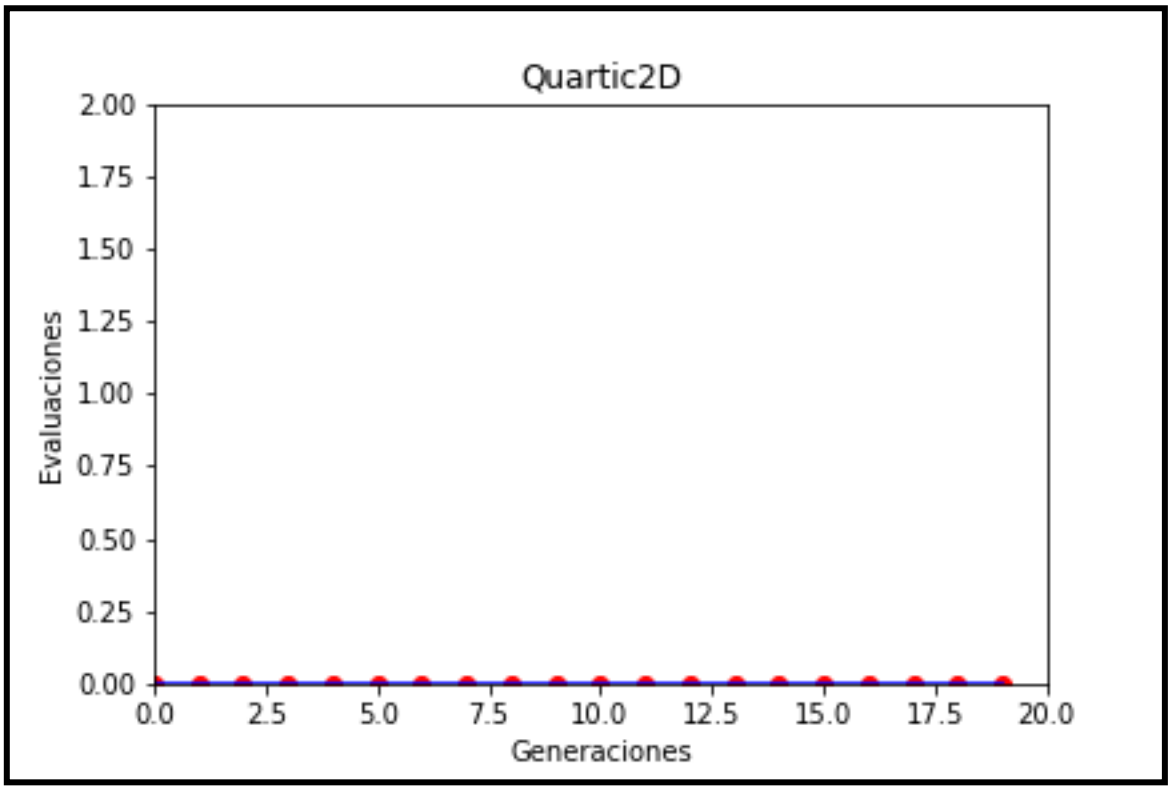
```
class sphere:
    MIN_VALUE = -5.12
    MAX_VALUE = 5.12

    def __init__(self):
        pass

    def fitness(self, vector):
        fit = 0.0
        for dimension in range(len(vector)):
            fit += vector[dimension]**2
        return fit
```

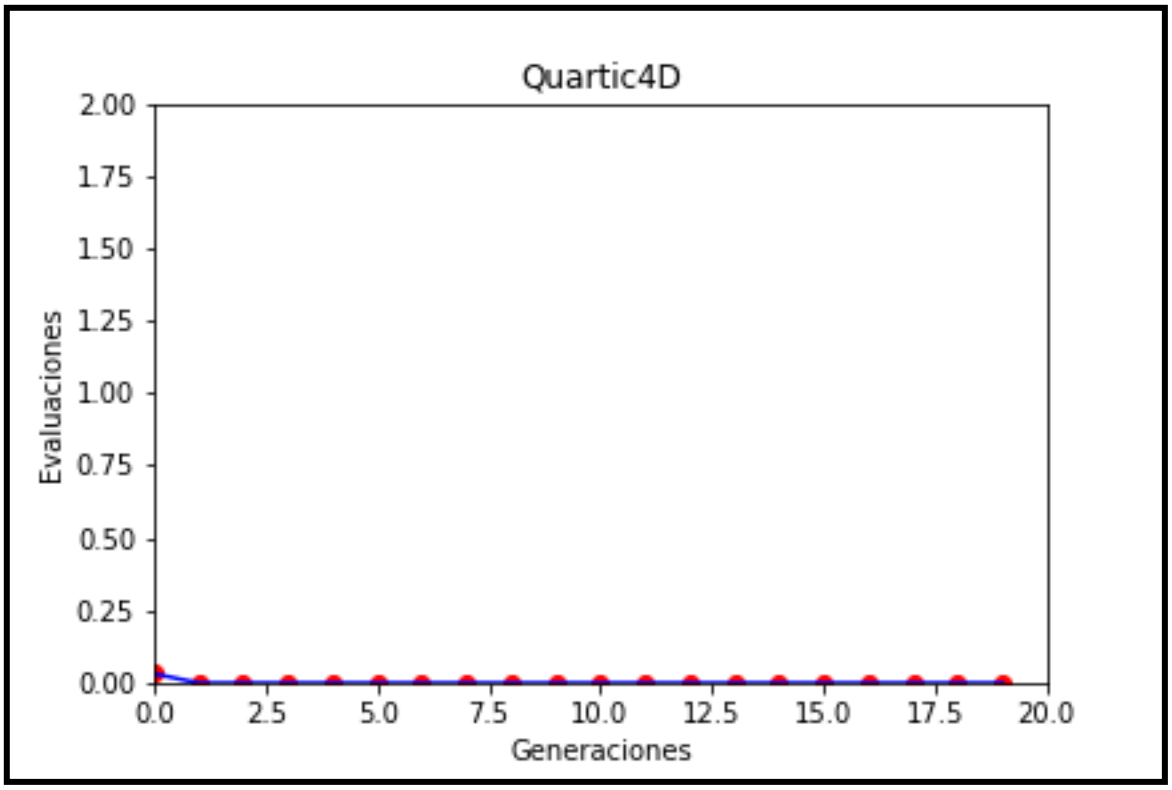
```python
import math

class rastrigin:
  MIN_VALUE = -5.12
  MAX_VALUE = 5.12

  def __init__(self):
      pass

  def fitness(self, vector):
      fit = 0.0
      for dimension in range(len(vector)):
          fit += vector[dimension]**2 - (10*math.cos(2*math.pi*vector[dimension]))
      fit += 10*len(vector)
      return fit
```

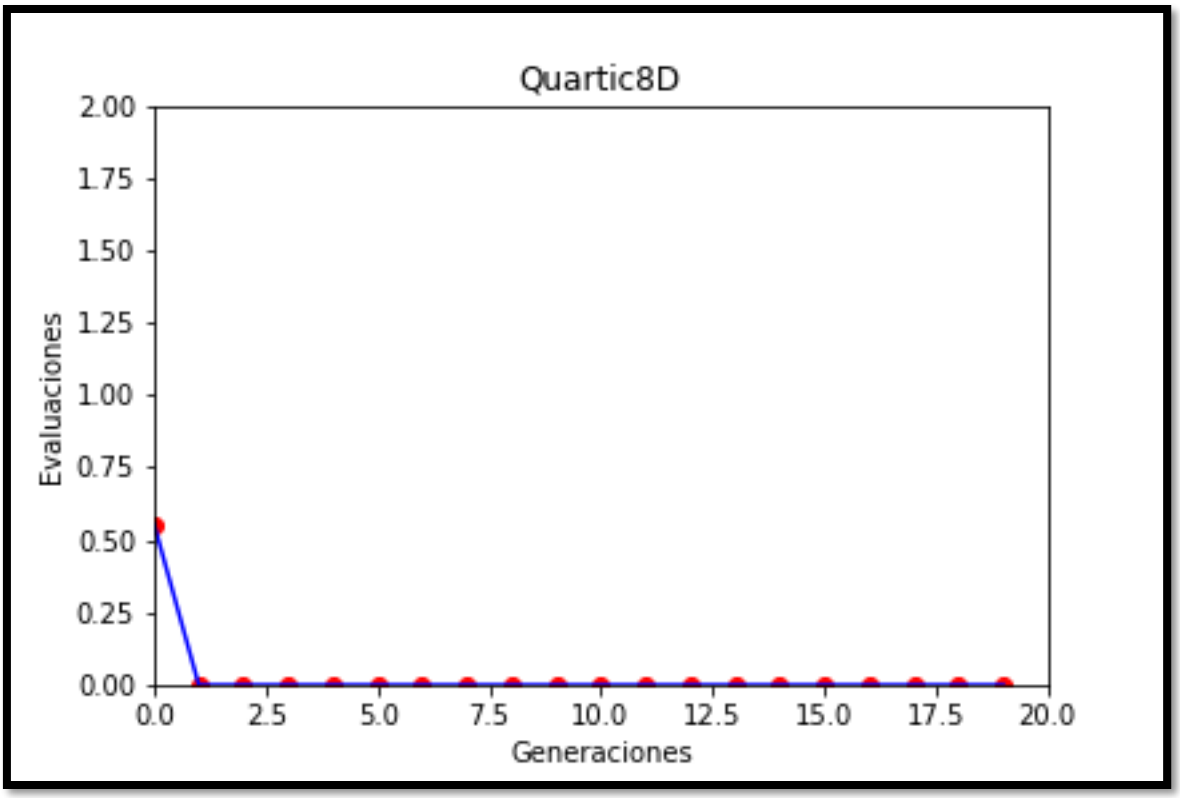
```python
class quartic:
    MIN_VALUE = -1.28
    MAX_VALUE = 1.28

    def __init__(self):
        pass

    def fitness(self, vector):
        fit = 0.0
        for dimension in range(len(vector)):
            fit += dimension*(vector[dimension]**4)
        return fit
```

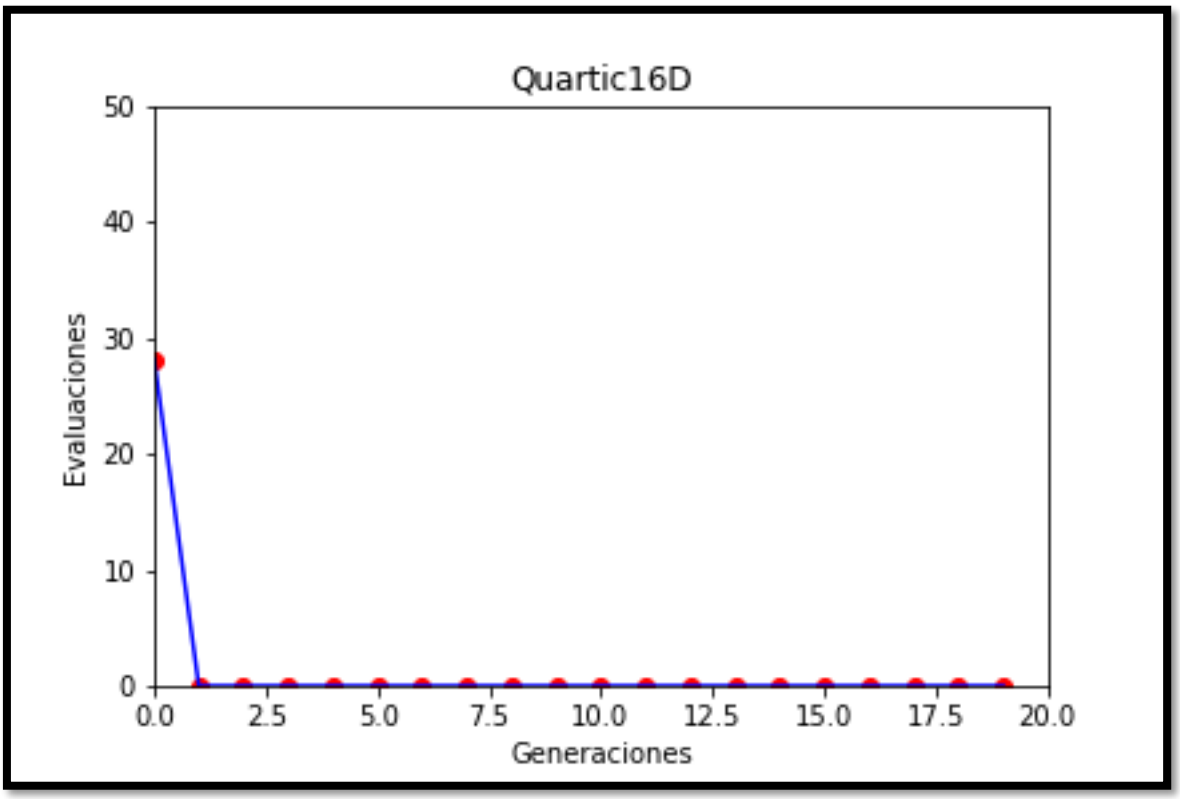
```python
class rosenbrock:
  MIN_VALUE = -2.048
  MAX_VALUE = 2.048

  def __init__(self):
      pass

  def fitness(self, vector):
      fit = 0.0
      for dimension in range(len(vector)-1):
          fit += 100 * (vector[dimension + 1] - vector[dimension]**2)**2 + (vector[dimension]-1)**2
      return fit
```

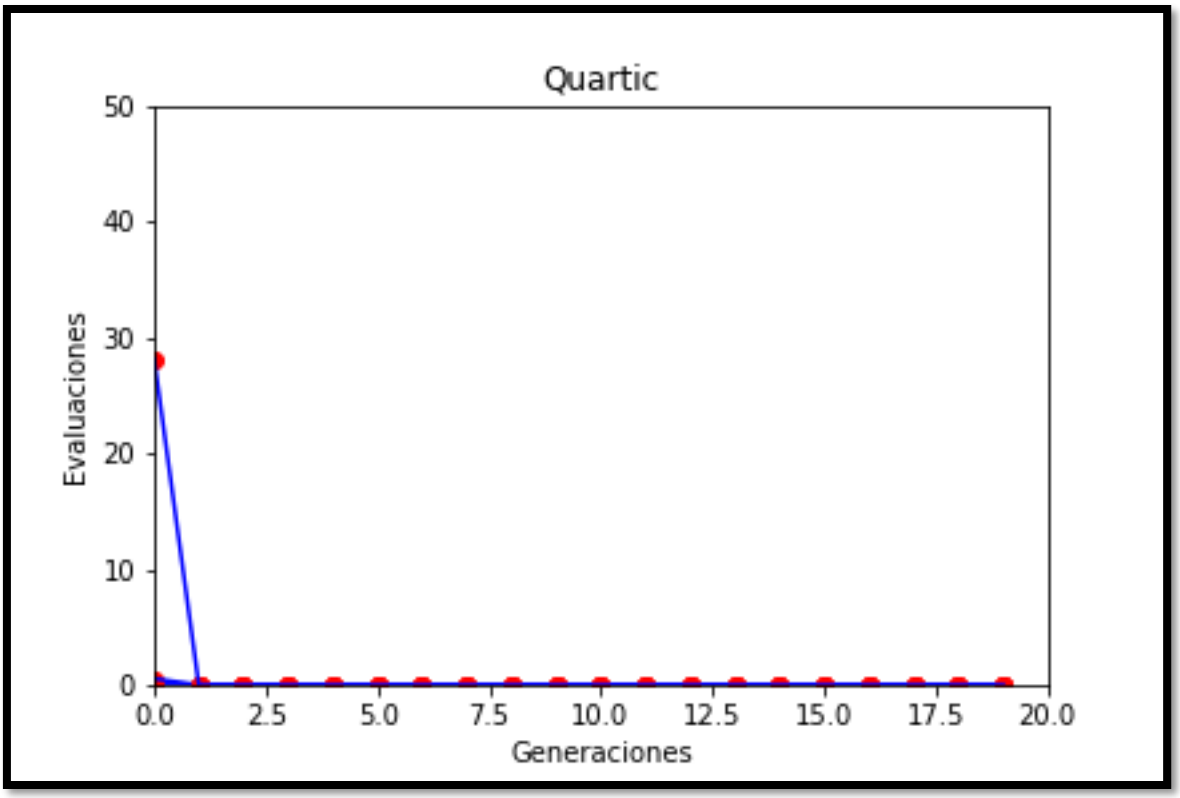
Resultados obtenidos

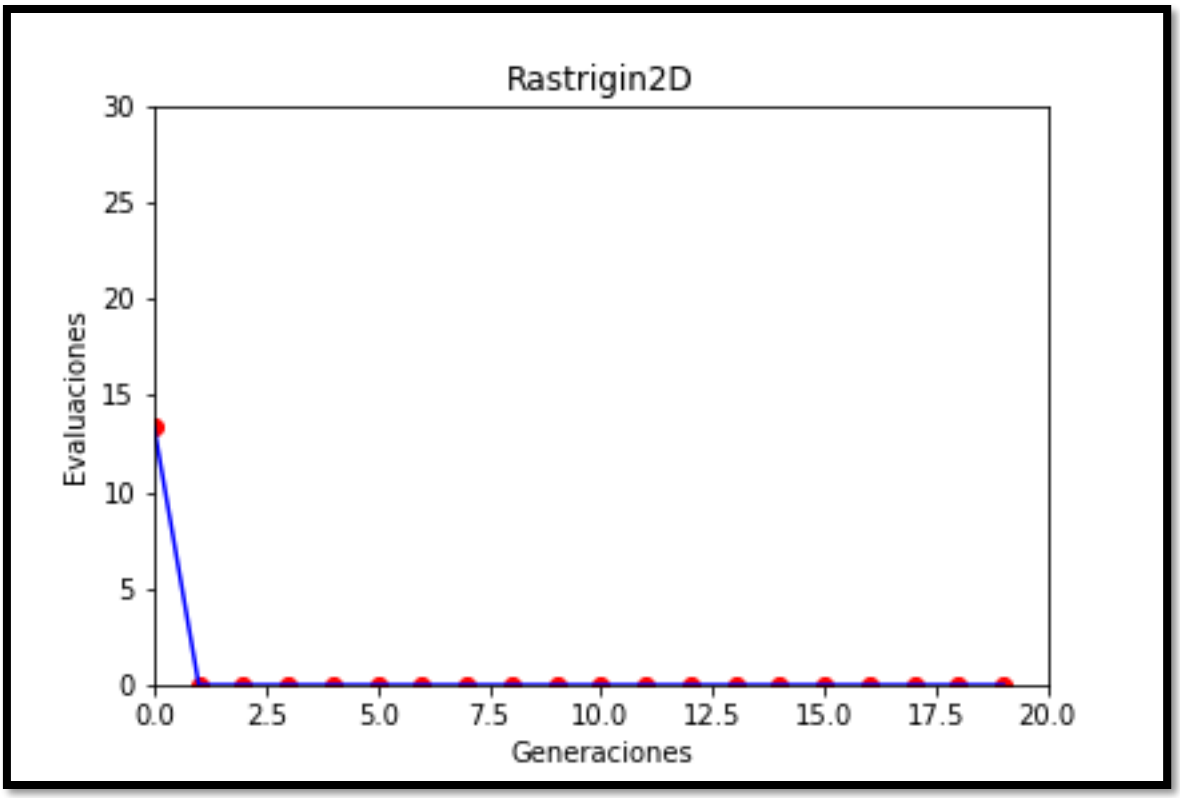


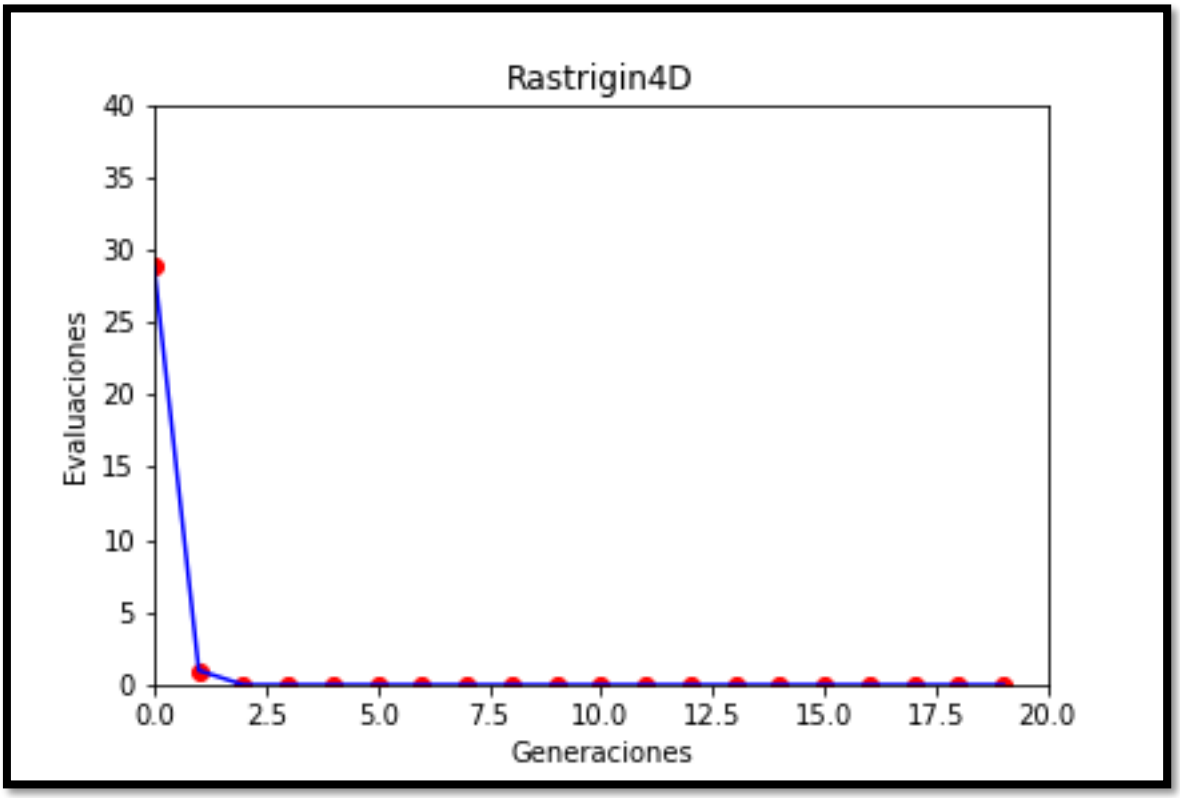


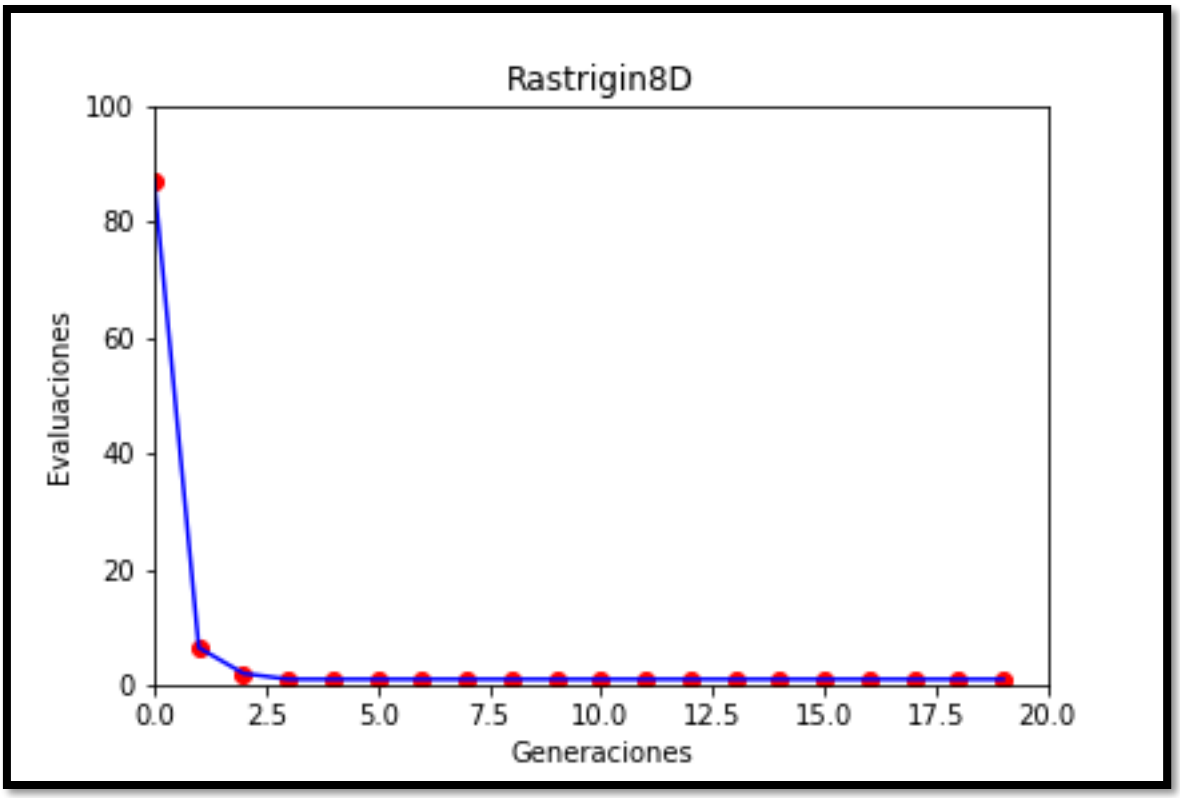

Quartic8D
Evaluaciones
Generaciones
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

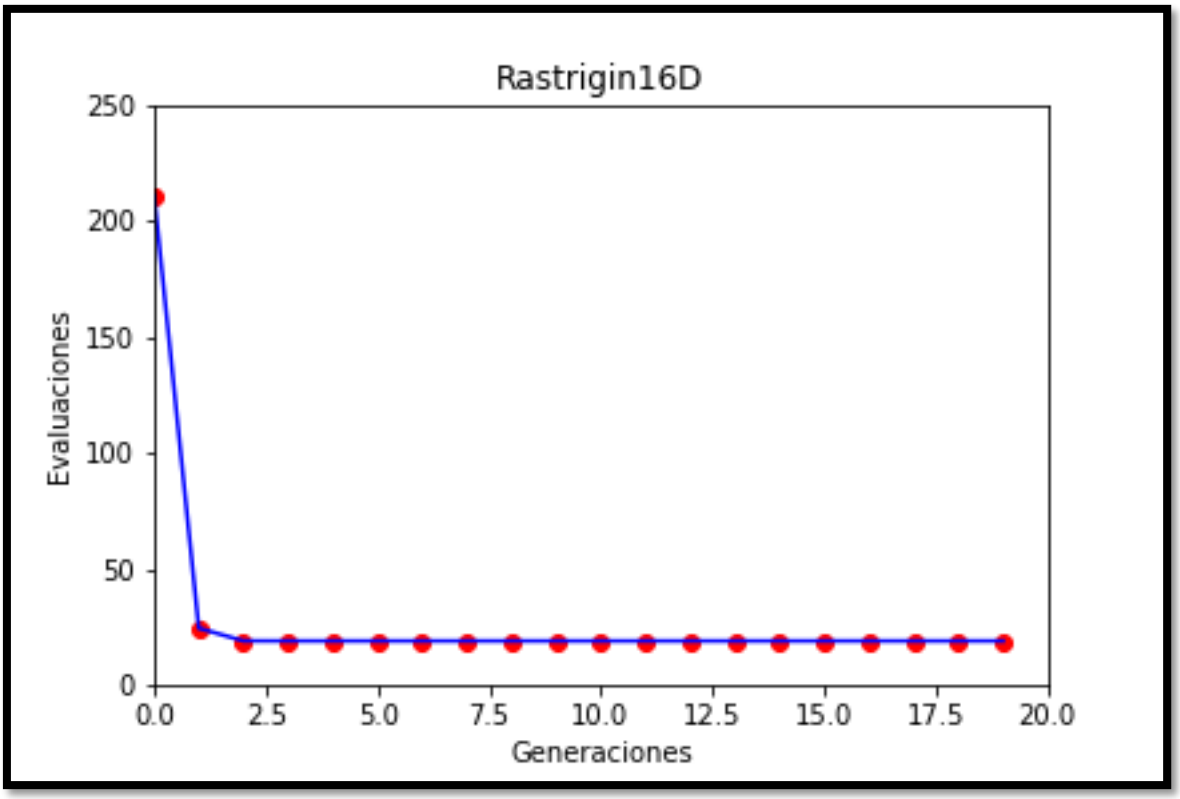

Quartic16D
Evaluaciones
Generaciones
50
40
30
20
10
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

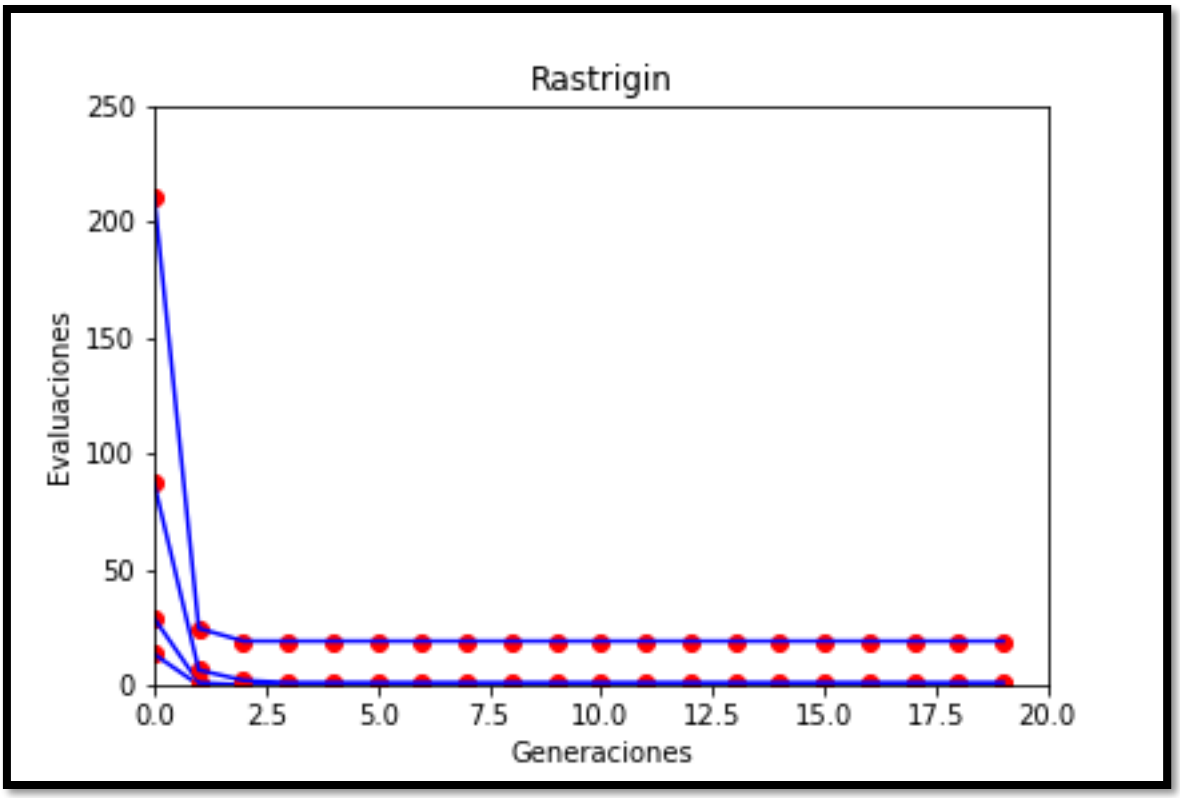

Quartic
50
40
30
20
10
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones

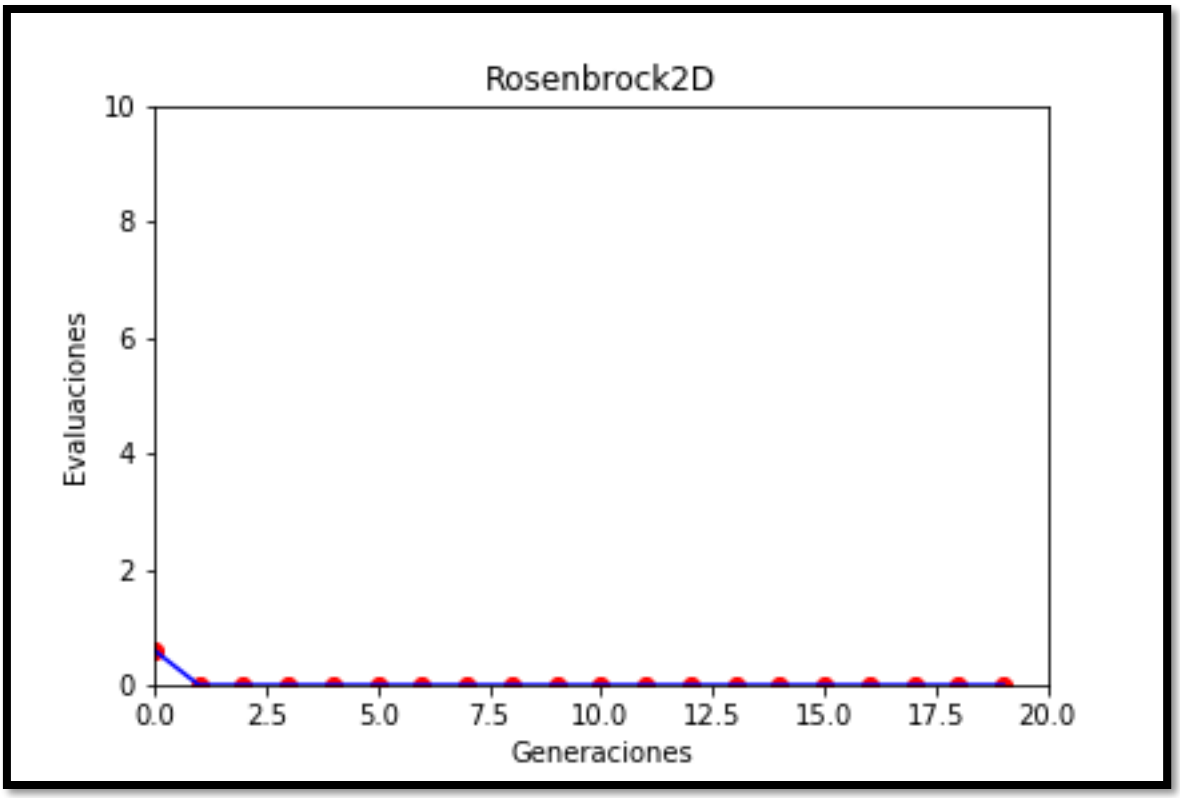

Rastrigin2D
Evaluaciones
Generaciones
0
5
10
15
20
25
30
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

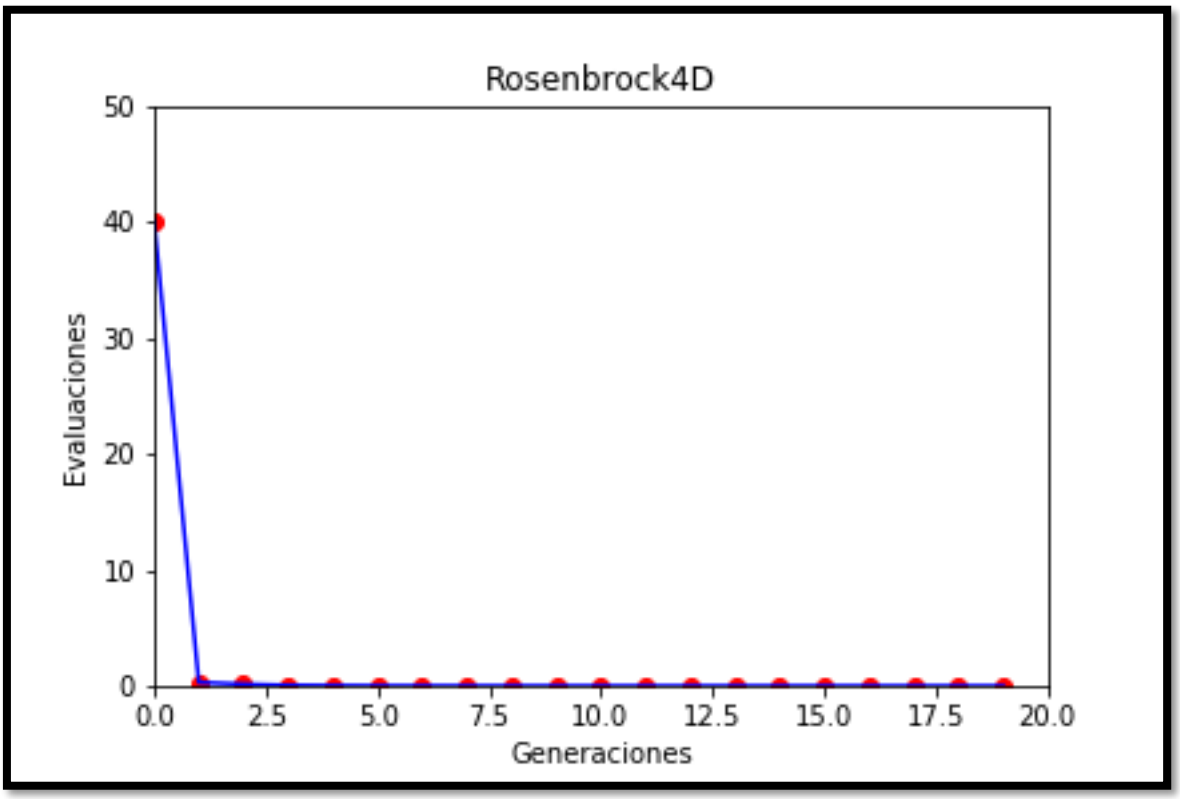

Rastrigin4D
Evaluaciones
Generaciones
0
5
10
15
20
25
30
35
40
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

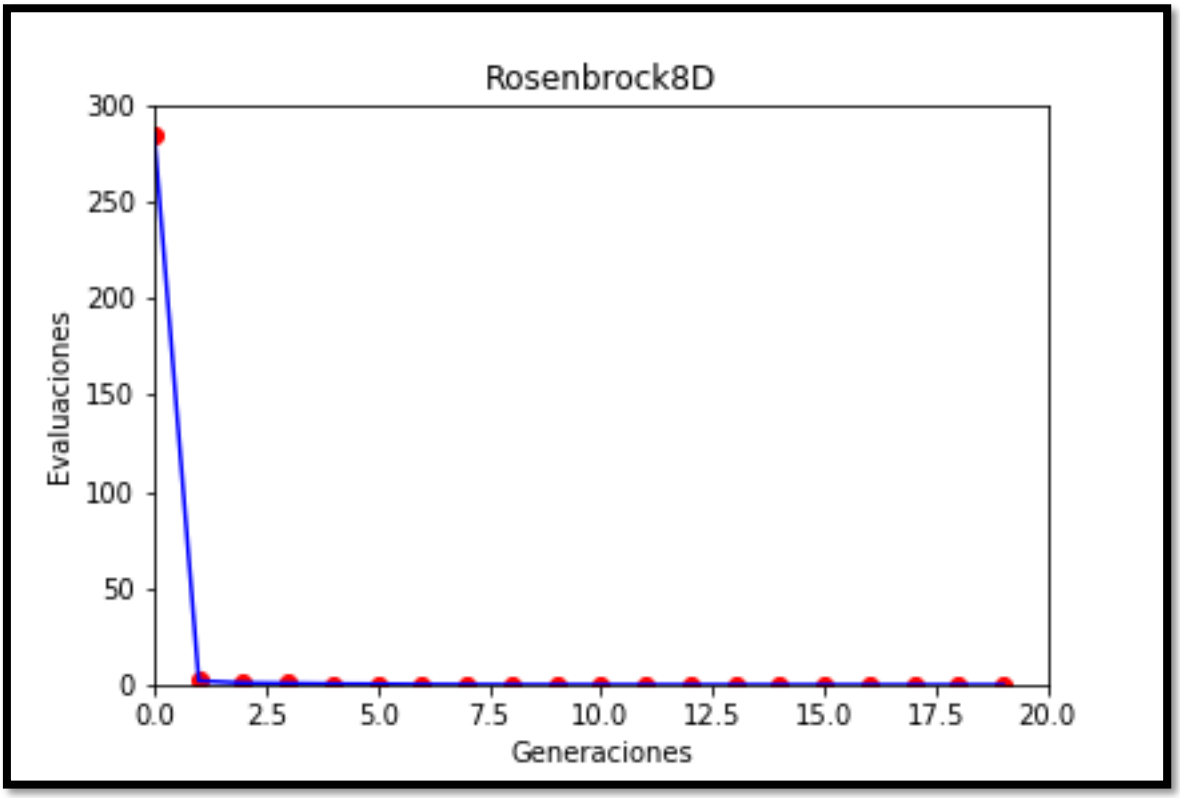

Rastrigin8D
Evaluaciones
Generaciones
0
20
40
60
80
100
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

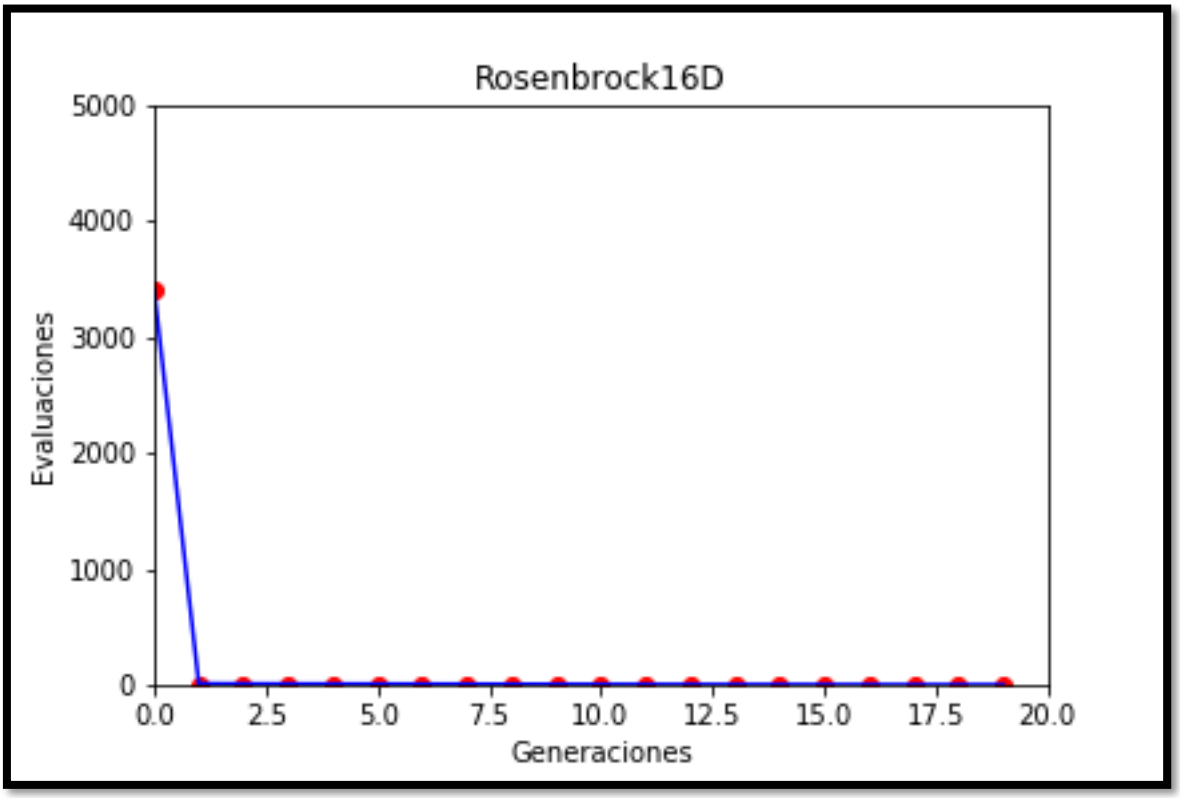

Rastrigin16D
Evaluaciones
Generaciones
0
50
100
150
200
250
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

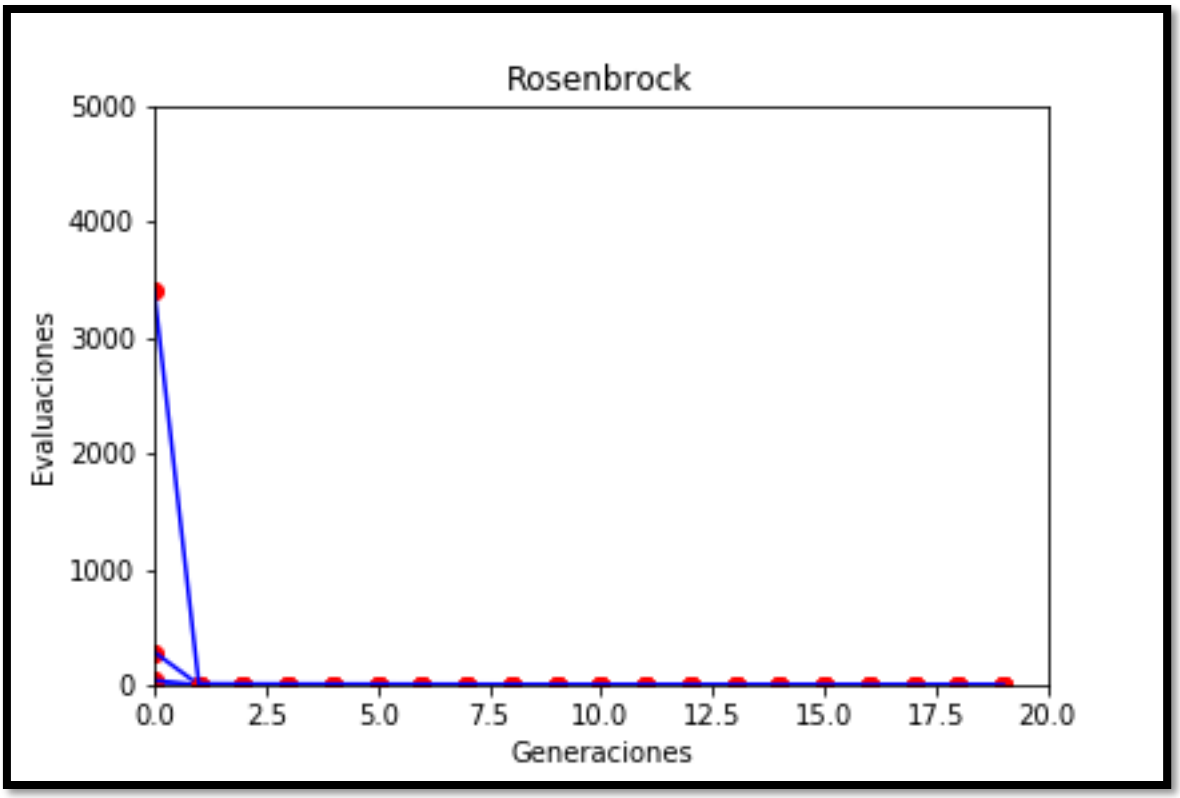

Rastrigin
Evaluaciones
Generaciones
250
200
150
100
50
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

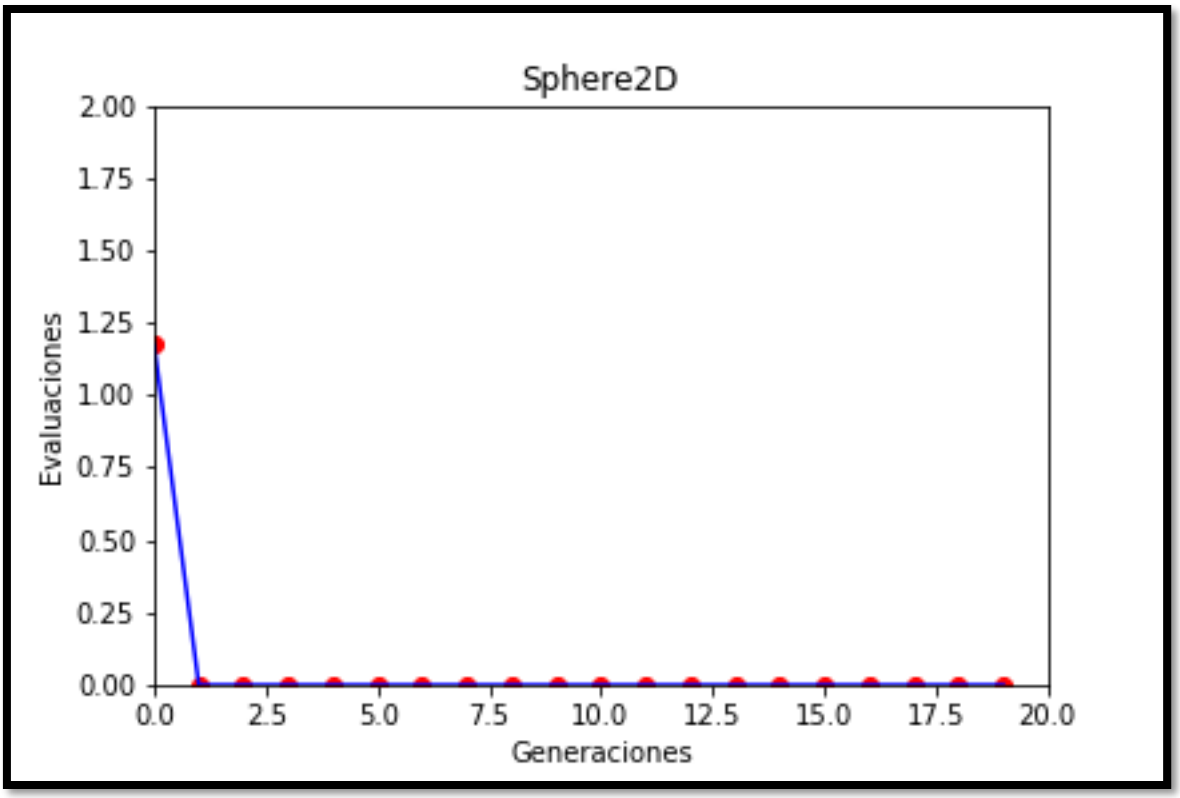

Rosenbrock2D
Evaluaciones
Generaciones
0
2
4
6
8
10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

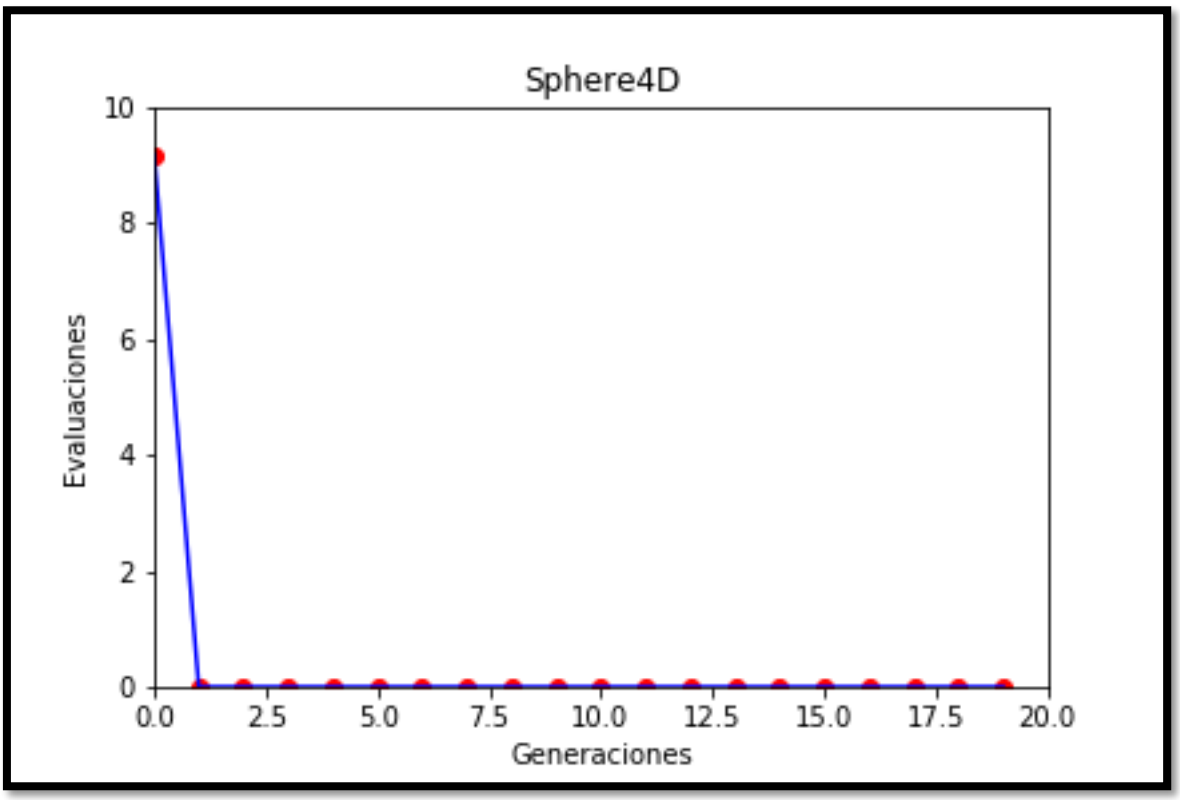

Rosenbrock4D
Evaluaciones
Generaciones
0
10
20
30
40
50
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

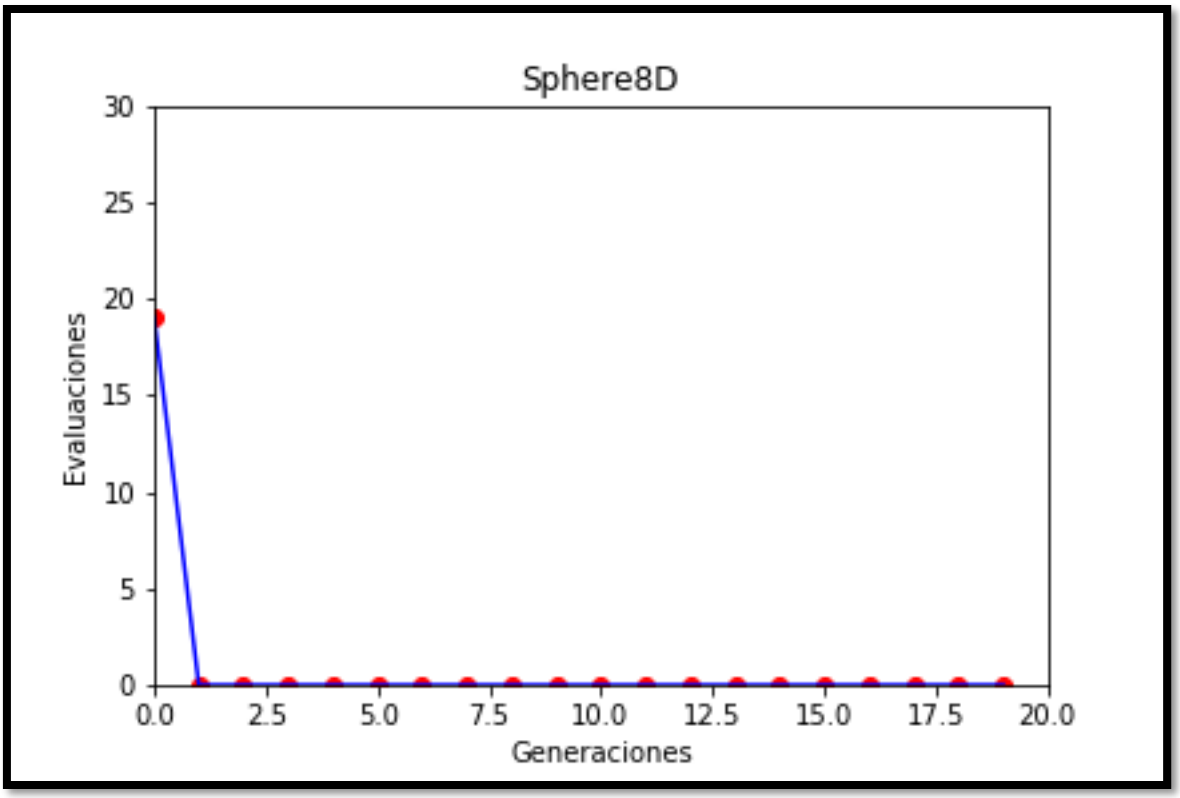

Rosenbrock8D
Evaluaciones
Generaciones
300
250
200
150
100
50
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

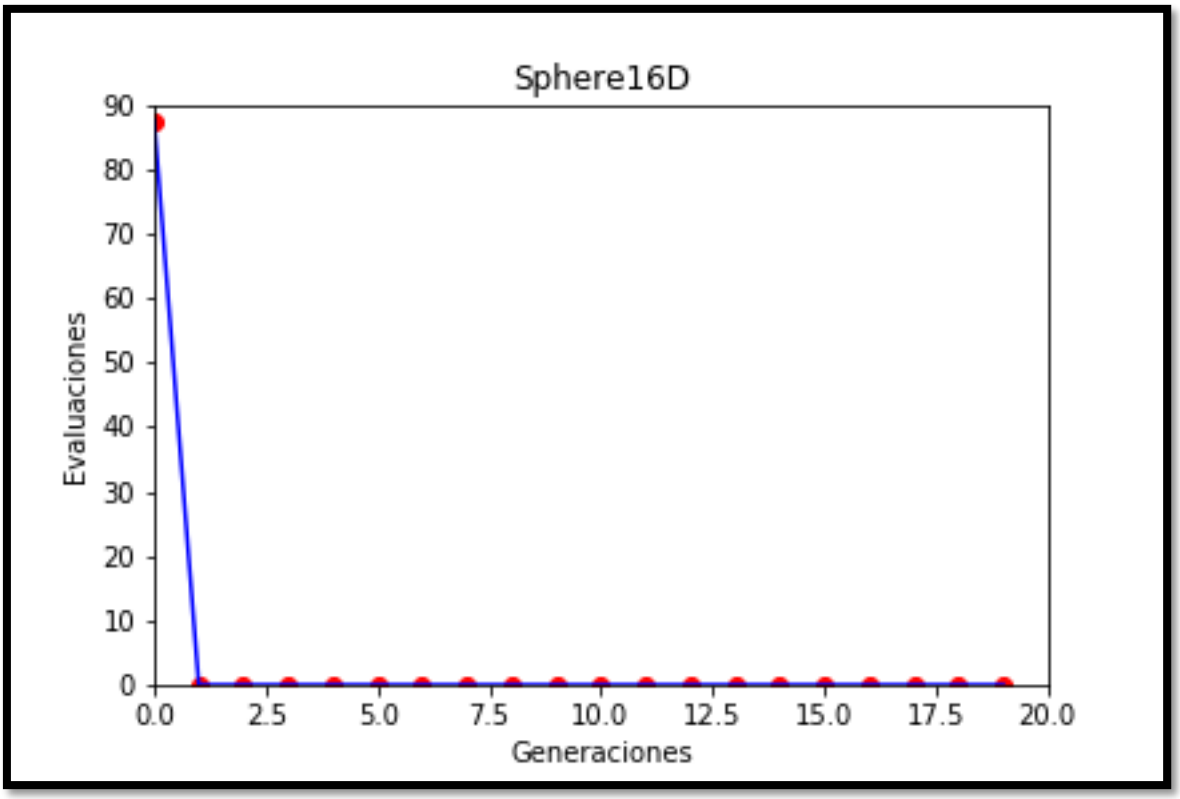

Rosenbrock16D
Evaluaciones
Generaciones
5000
4000
3000
2000
1000
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

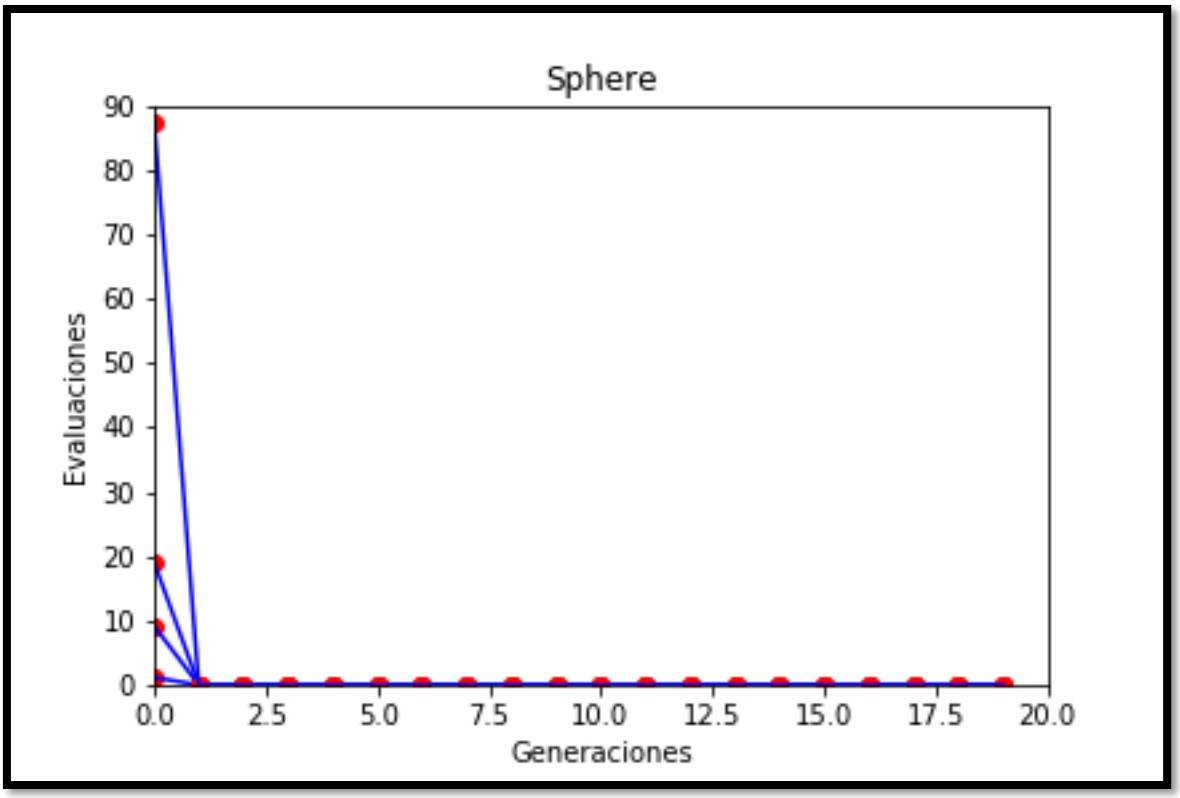

Rosenbrock
5000
4000
3000
2000
1000
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones


Sphere2D
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones


Sphere4D
10
8
6
4
2
0
Evaluaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Generaciones


Sphere8D
Evaluaciones
Generaciones
0
5
10
15
20
25
30
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Sphere16D
Evaluaciones
Generaciones
0
10
20
30
40
50
60
70
80
90
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Sphere
Evaluaciones
Generaciones
90
80
70
60
50
40
30
20
10
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Conclusiones

En la realización de esta actividad y la implementación de este código, podemos observar la gran diferencia de optimización y tiempos de ejecución comparados con la optimización por colonia de hormigas; si bien ambos algoritmos resultaros completos, el de la entrega anterior demostró altísimos tiempos de ejecución, tomando bastantes minutos para resolver prácticamente cualquiera de las ecuaciones con cualquier cantidad de dimensiones, mientras que el algoritmo presentado en el trabajo actual tomaba apenas un par de segundos para realizar las 10 mil ejecuciones en total para cada gráfica.

Con esto podemos ver más a detalla cómo es que los algoritmos de minimización se comportan y arrojan resultados para cada situación, y podemos ver también de mejor manera cómo es que el anidar ciclos en los que hay bastantes operaciones hace tan significativamente lento un proceso.

Gracias a la implementación actual, fue posible resolver los problemas de una forma mucho mejor y más rápida.