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 6: Algoritmo Genético Contínuo.

Instrucciones: Implementar y evaluar el rendimiento del algoritmo Genético Contínuo 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

```
27    def run(self):
28        self.crearIndividuos()
29        self._mejor_historico = self._individuos[0]
30        generacion = 0
31 #       file = open("C:/Users/qpqpq/Documents/7mo Semestre/Seminario IA 1/Actividad 6/datos_quartic/quartic16_5.txt", "w")
32        while generacion < self._generaciones:
33            self.evaluaIndividuos()
34            self.mejor()
35            hijos = np.array([])
36            while len(hijos) < len(self._individuos):
37                padre1 = self.ruleta()
38                padre2 = self.ruleta()
39                while padre1 == padre2:
40                    padre2 = self.ruleta()
41                h1, h2 = self.cruza(self._individuos[padre1], self._individuos[padre2])
42                hijos = np.append(hijos, [h1])
43                hijos = np.append(hijos, [h2])
44            self.mutacion(hijos)
45            self._individuos = np.copy(hijos)
46
47 #           print("Generación: ", generacion, 'Mejor Histórico: ', self._mejor_historico._cromosoma, -1 * (self._mejor_historico._fitness - self._problema.MAX_VALUE ** self._alelos))
48            print("Generación: ", generacion, 'Mejor Histórico: ', self._mejor_historico._cromosoma, -1 * (self._mejor_historico._fitness - ((self._problema.MAX_VALUE ** self._alelos) * 1000000)))
49 #           if generacion%100 == 0:
50 #               dato = -1 * (self._mejor_historico._fitness - ((self._problema.MAX_VALUE ** self._alelos) * 1000000))
51 #               file.write(str(dato) + "\n")
52            generacion += 1
53 #           if(generacion == 2000):
54 #               file.close()
```

Lo primero que se realizó fue un par de modificaciones en la función “run” del algoritmo genético continuo.

Comenzando por la línea 32, en la que se agregó la creación de archivos de texto plano para cada una de las 5 ejecuciones para cada una de las 4 gráficas de las instrucciones de la práctica.

Posteriormente, en la línea 47 se comentó la impresión original de las generaciones y los resultados de las ejecuciones, debido a que la evaluación de los individuos fue modificada, y se observará en una imagen posteriormente. Así pues, en la línea 48 se agrega una nueva evaluación, que corresponde sencillamente a aumentar la original por un valor mucho más alto, para asegurarnos de que jamás nos entregue un valor negativo que pueda hacer fallar la función “ruleta”, y al mismo tiempo para que sea suficientemente significativo como para poder usarlo como solución al problema actual, que es de minimización. Por esto se entiende que, como los algoritmos genéticos resuelven problemas de

maximización, siempre arrojan valores cada vez más altos, pero en este caso necesitamos que sean cada vez más bajos, debido a que estamos evaluando el punto más bajo en una gráfica dada.

A continuación, en las líneas 49 a 51, obtenemos el valor del mejor individuo históricamente cada 100 generaciones y lo escribimos en el archivo de texto.

Finalmente, en la línea 53 nos aseguramos de cerrar correctamente el archivo una vez que la ejecución del programa ha finalizado.

```
62      def evaluaIndividuos(self):
63          for i in self._individuos:
64              i._fitness = self._problema.fitness(i._cromosoma)
65              i._fitness *= -1
66              #i._fitness += self._problema.MAX_VALUE ** self._alelos
67              i._fitness += (self._problema.MAX_VALUE ** self._alelos) * 1000000
```

Como se mencionó con anterioridad, una parte de la evaluación de los individuos fue modificada, y se puede apreciar en la imagen arriba de este texto. De nuevo, podemos ver que lo único que se realizó fue una multiplicación del valor original por un número mucho más alto, para asegurarnos de tener ciertos valores más favorables que no causaran errores en el programa.

Además de esto, ninguna otra parte del código genético fue modificada.

```
16 def main():
17     #sp = sphere.Sphere()
18     #ro = rosenbrock.Rosenbrock()
19     #ra = rastrigin.Rastrigin()
20     #qu = quartic.Quartic()
21
22     #ag = AGC.AGC(32, 16, 2000, 0.02, sp)
23     #ag = AGC.AGC(32, 16, 2000, 0.02, ro)
24     #ag = AGC.AGC(32, 16, 2000, 0.02, ra)
25     #ag = AGC.AGC(32, 16, 2000, 0.02, qu)
26
27     #ag.run()
```

Por otro lado, la función principal del programa fue modificada como se puede observar en la presente imagen. Se creó un objeto de cada tipo de gráfica (que se verá a detalle más adelante), y se mandó ejecutar el programa 5 veces para cada una, para poder obtener los datos requeridos por las instrucciones de la práctica.

Las clases de cada tipo de objeto se pueden observar a detalle a continuación.

```python
class Sphere:
    MIN_VALUE = -5.12
    MAX_VALUE = 5.12
    def __init__(self):
        pass
    def fitness(self, cromosoma):
        z = 0
        for alelo in cromosoma:
            z += alelo**2
        return z
```

```python
class Rosenbrock:
    MIN_VALUE = -2.048
    MAX_VALUE = 2.048
    def __init__(self):
        pass
    def fitness(self, cromosoma):
        z = 0

        for i in range(len(cromosoma)-1):
            z += 100*((cromosoma[i+1] - cromosoma[i]**2)**2) + (cromosoma[i] - 1)**2

        return z
```

```python
import math

class Rastrigin:
    MIN_VALUE = -5.12
    MAX_VALUE = 5.12

    def __init__(self):
        pass

    def fitness(self, cromosoma):
        z = 0

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

        z += 10*(len(cromosoma))

        return z
```

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

    def __init__(self):
        pass

    def fitness(self, cromosoma):
        z = 0

        for i in range(len(cromosoma)):
            z += i * (cromosoma[i]**4)

        return z
```

Para todos los casos se puede observar y entender que la ecuación generadora de la gráfica ha sido “generalizada” en términos de las posiciones de los alelos del cromosoma de cada individuo, siendo sustituidos por las posiciones que la ecuación representa con “x”.

```python
    a1 = a2 = a3 = a4 = a5 = np.array([])
    b1 = b2 = b3 = b4 = np.array([])

    g = graphic.Graphic()

    a1 = g.leer("quartic2_1.txt")
    a2 = g.leer("quartic2_2.txt")
    a3 = g.leer("quartic2_3.txt")
    a4 = g.leer("quartic2_4.txt")
    a5 = g.leer("quartic2_5.txt")
    b1 = g.graficar(a1, a2, a3, a4, a5, 0.2, 'Quartic 2D')

    a1 = g.leer("quartic4_1.txt")
    a2 = g.leer("quartic4_2.txt")
    a3 = g.leer("quartic4_3.txt")
    a4 = g.leer("quartic4_4.txt")
    a5 = g.leer("quartic4_5.txt")
    b2 = g.graficar(a1, a2, a3, a4, a5, 0.2, 'Quartic 4D')

    a1 = g.leer("quartic8_1.txt")
    a2 = g.leer("quartic8_2.txt")
    a3 = g.leer("quartic8_3.txt")
    a4 = g.leer("quartic8_4.txt")
    a5 = g.leer("quartic8_5.txt")
    b3 = g.graficar(a1, a2, a3, a4, a5, 2.5, 'Quartic 8D')

    a1 = g.leer("quartic16_1.txt")
    a2 = g.leer("quartic16_2.txt")
    a3 = g.leer("quartic16_3.txt")
    a4 = g.leer("quartic16_4.txt")
    a5 = g.leer("quartic16_5.txt")
    b4 = g.graficar(a1, a2, a3, a4, a5, 19, 'Quartic 16D')

    g.mesh(b1, b2, b3, b4, 19, 'Quartic')
```

De regreso al main podemos observar el siguiente código.

Debido a que se necesitan 5 ejecuciones por dimensión de cada gráfica, generamos 5 arreglos (a1 – a5) en donde obtendremos los resultados (que se obtienen al ejecutar la función “leer”). Y posteriormente son graficados con la función “graficar”, que recibe como parámetros 5 arreglos (de los

cuáles se harán los promedios de cada mejor individuo por generaciones disponibles), el punto máximo en el eje Y para graficar dichos resultados, y el nombre que llevará la gráfica. Sobra decir que este procedimiento se tuvo que llevar a cabo múltiples veces para poder dar lugar a todos los resultados necesarios.

Finalmente, como se necesitan dichos resultados para poder generar una gráfica en conjunto con todo, el método retorna un arreglo ya promediado, y al final de todo ejecutamos la función “mesh”, en donde sencillamente graficamos en un solo diagrama los resultados obtenidos previamente, con una medida general alta en el eje Y, y un nombre general. Observemos entonces esa parte restante del algoritmo.

```
6   def __init(self, direccion):
7       self._direccion = direccion
8
9   def leer(self, direccion):  #Lee un txt con los datos de los mejores historicos, los transforma a floats (porque se leen como str)
10      array = np.array([])    #los agrega a un array, y lo retorna
11
12      archivo = open(direccion, "r")
13      datos = archivo.read().splitlines()
14      archivo.close()
15
16      for i in range(len(datos)):
17          num = float(datos[i])
18          array = np.append(array, [num])
19
20      return array
```

Como podemos observar, el método de lectura de los resultados obtenidos por cada iteración de las gráficas genera un arreglo, lee cada dato, lo convierte en un dato numérico, termina el proceso y retorna el arreglo calculado (esto para poder juntarlo con los otros parámetros para la función “mesh”).

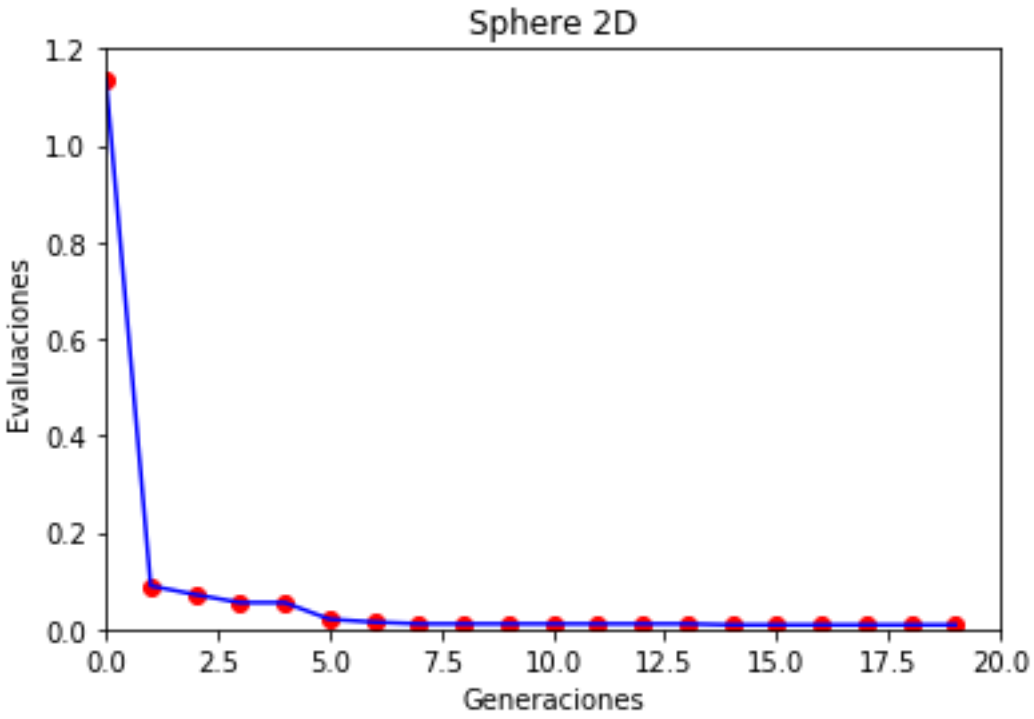
```
22  def graficar(self, array1, array2, array3, array4, array5, y_limit, title):    #y_limit son los puntos a graficar en el eje Y.
23      array_Y = np.array([])
24
25      for i in range(20):
26          Y = 0
27          Y += array1[i]
28          Y += array2[i]
29          Y += array3[i]
30          Y += array4[i]
31          Y += array5[i]
32          Y /= 5
33
34          array_Y = np.append(array_Y, [Y])
35
36      plt.plot(array_Y, 'ro', array_Y, 'b')
37      plt.axis([0,20,0,y_limit])
38      plt.ylabel('Evaluaciones')
39      plt.xlabel('Generaciones')
40      plt.title(title)
41      plt.show()
42      return array_Y
43
44  def mesh(self, g1, g2, g3, g4, y_limit, title):
45      plt.plot(g1, 'ro', g1, 'b', g2, 'ro', g2, 'b', g3, 'ro', g3, 'b', g4, 'ro', g4, 'b')
46      plt.axis([0,20,0,y_limit])
47      plt.ylabel('Evaluaciones')
48      plt.xlabel('Generaciones')
49      plt.title(title)
50      plt.show()
```

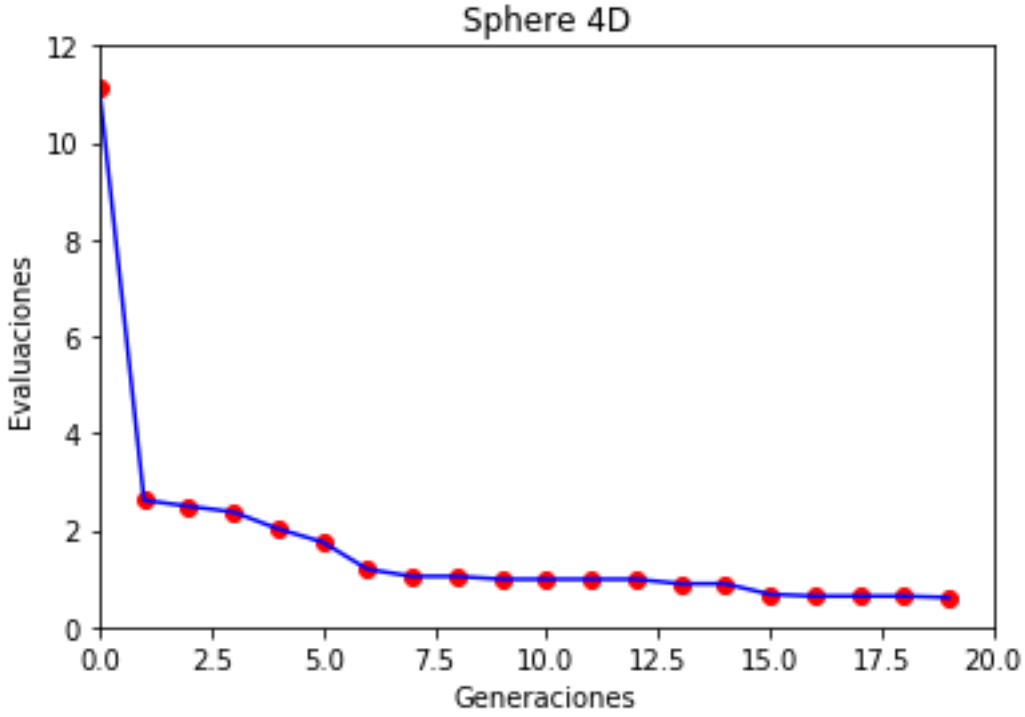
Ahora, para graficar tenemos primero una función que recibe los primeros 5 arreglos en crudo de las ejecuciones de una función, y los promedia, arrojando todo en un arreglo final que solamente contiene 20 datos. Posteriormente genera la función dos veces: una es en forma lineal azul, y otra en forma de puntos rojos; esto permite apreciar mejor tanto la gráfica genera como los puntos específicos de cada

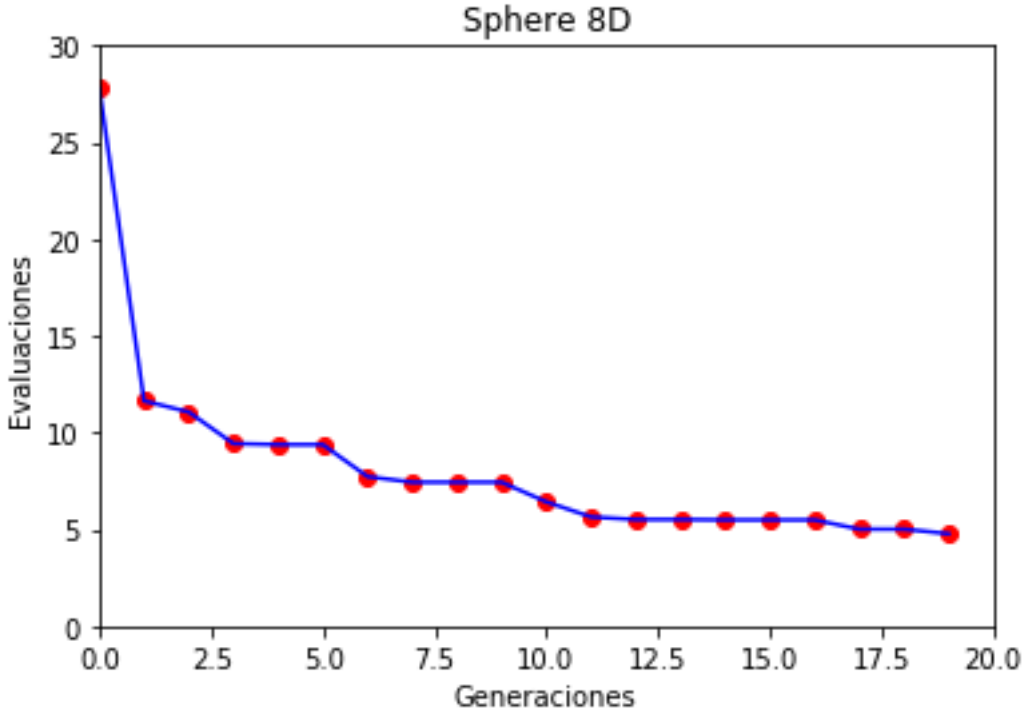
generación, permitiendo apreciar mejor el desempeño promedio del algoritmo. Finalmente retorna el arreglo para poder graficarlo nuevamente después, pero ahora en conjunto con todos los demás.

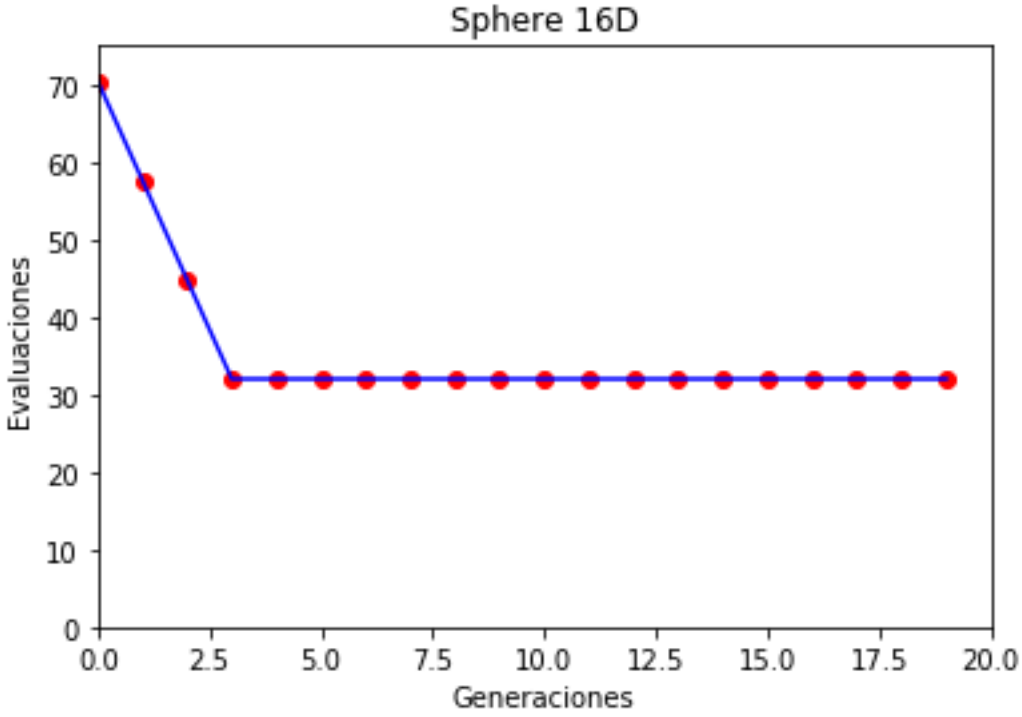
Así, obtenemos la función “mesh”, que hace prácticamente lo mismo, pero en conjunto con los 4 resultados promedio iniciales.

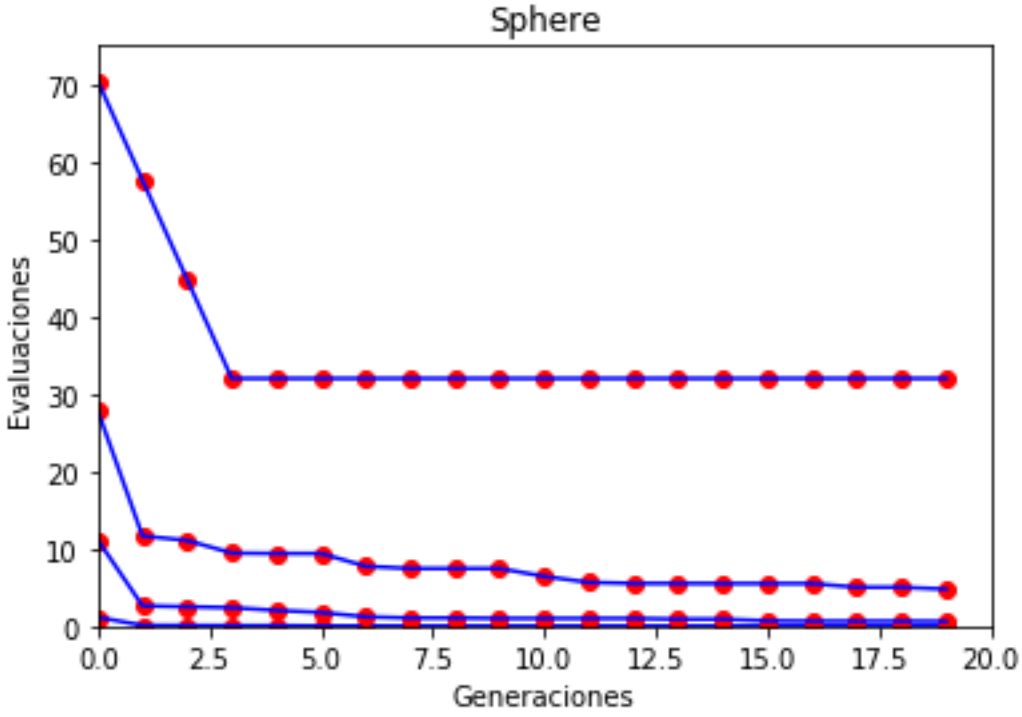
Resultados obtenidos

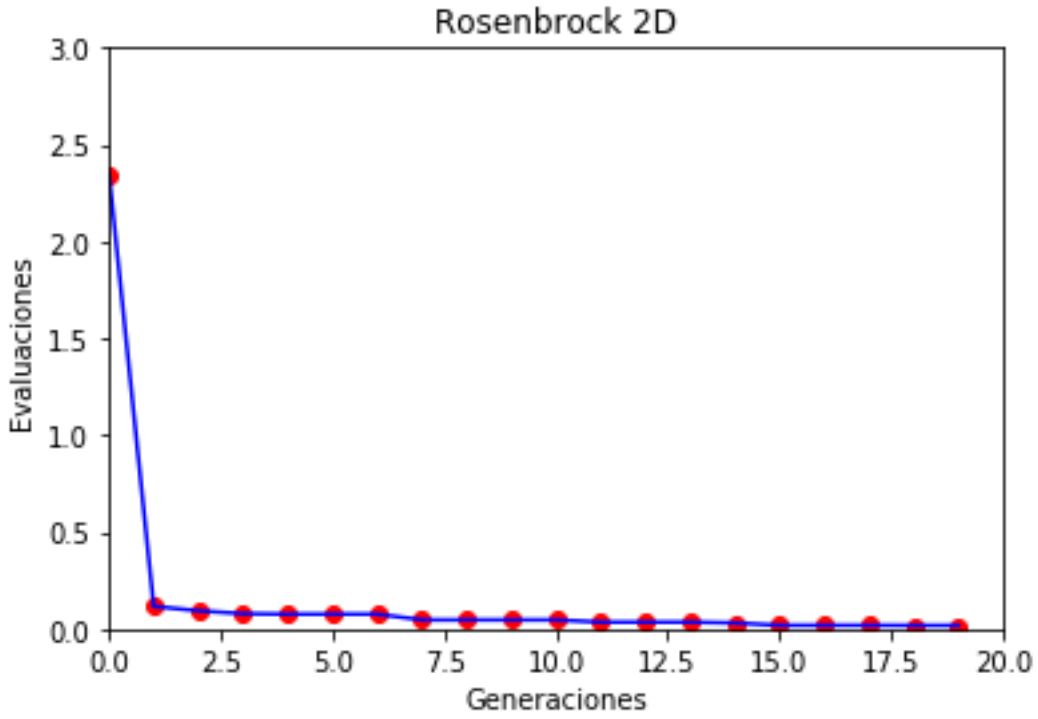


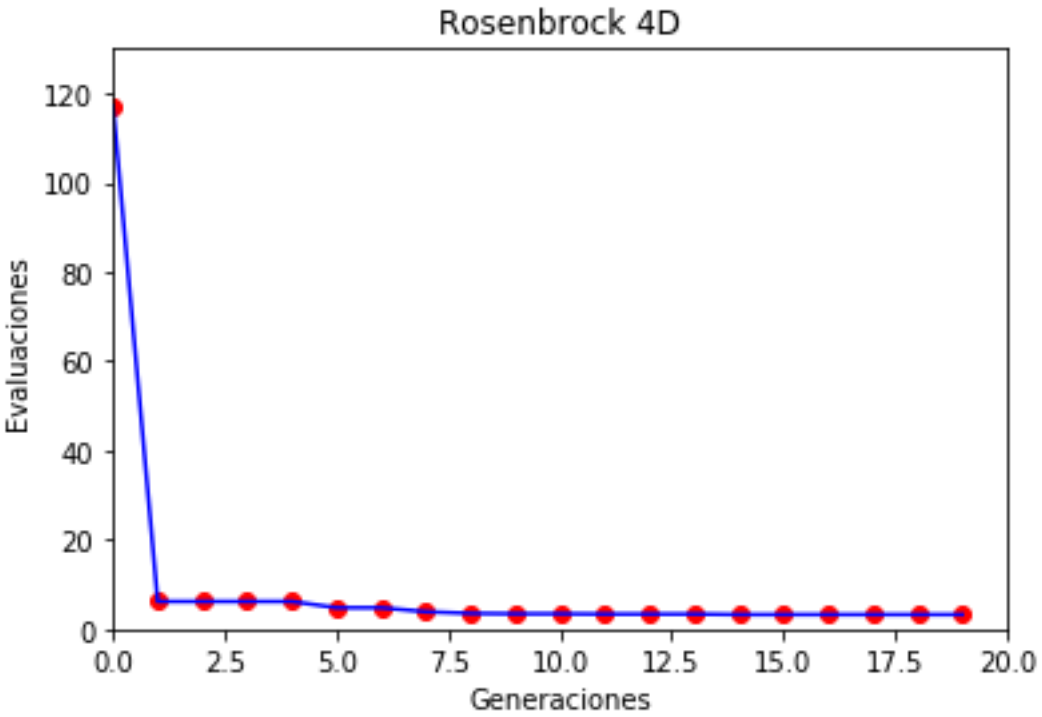

Sphere 4D
Evaluaciones
Generaciones
0
2
4
6
8
10
12
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

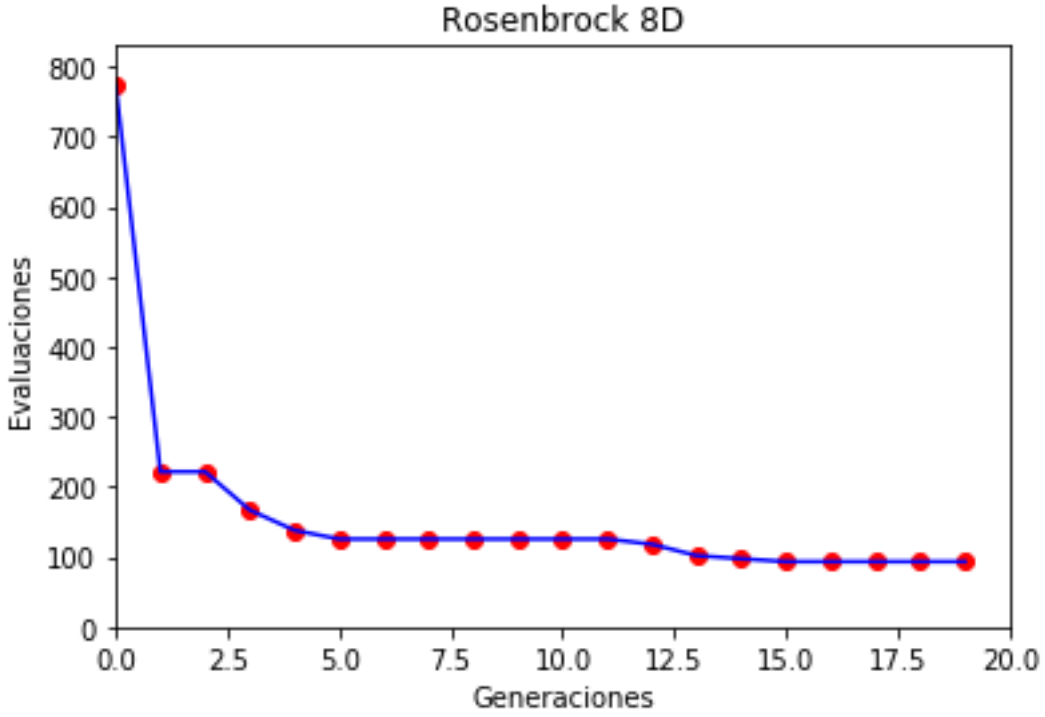

Sphere 8D
Evaluaciones
Generaciones
0
5
10
15
20
25
30
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

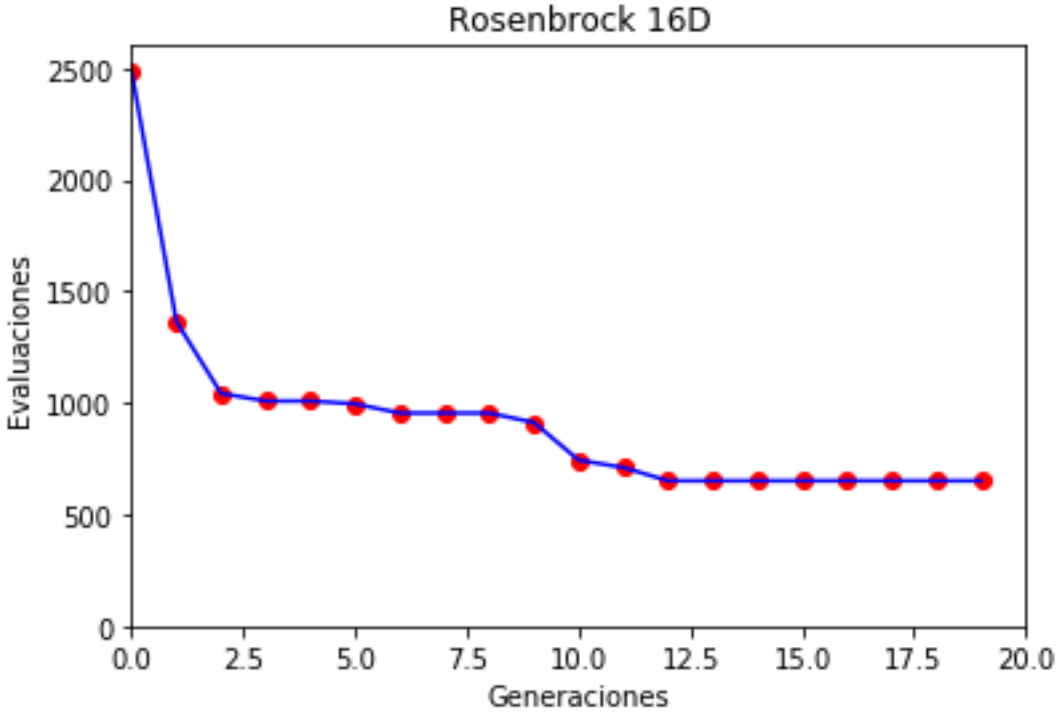

Sphere 16D
Evaluaciones
Generaciones
0
10
20
30
40
50
60
70
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

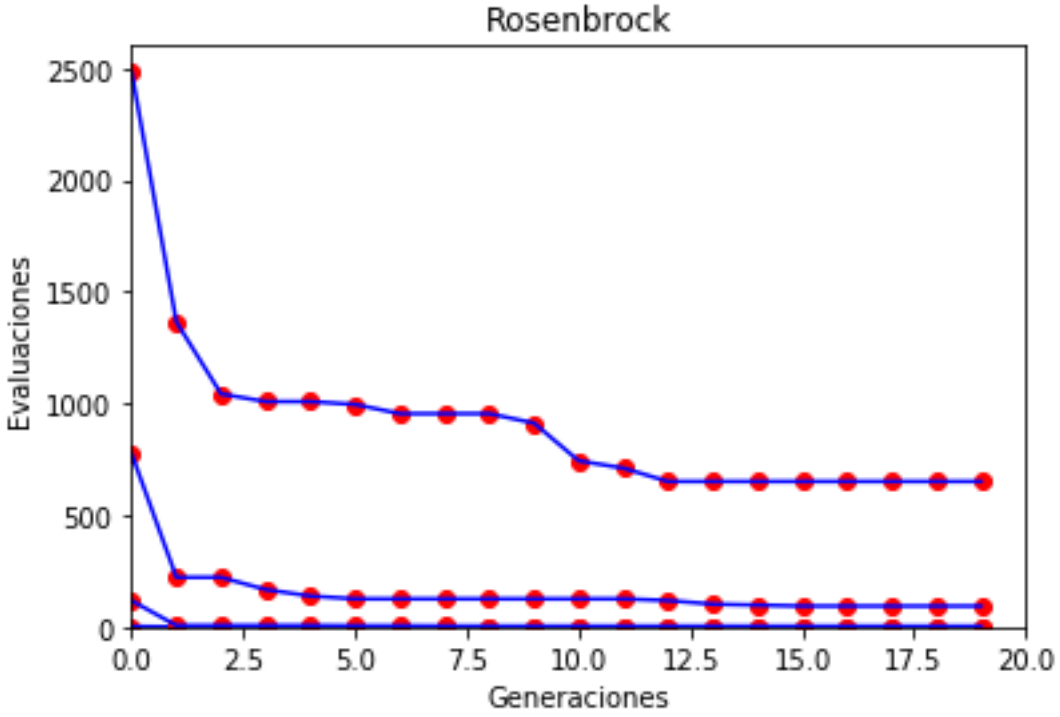

Sphere
Evaluaciones
Generaciones
0
10
20
30
40
50
60
70
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

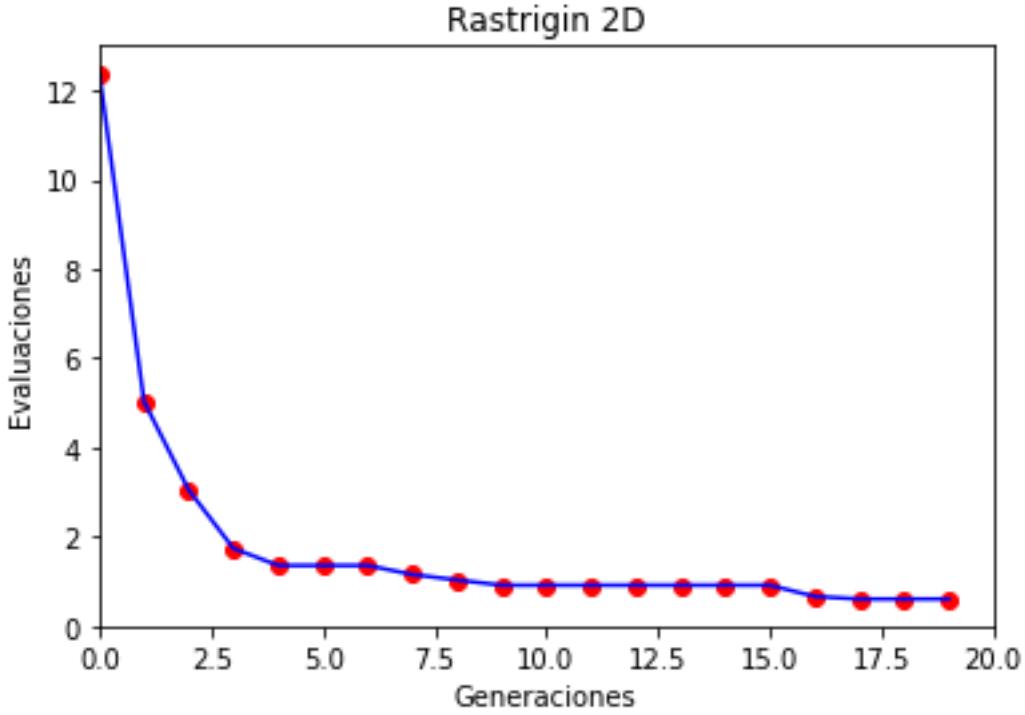

Rosenbrock 2D
Evaluaciones
Generaciones
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

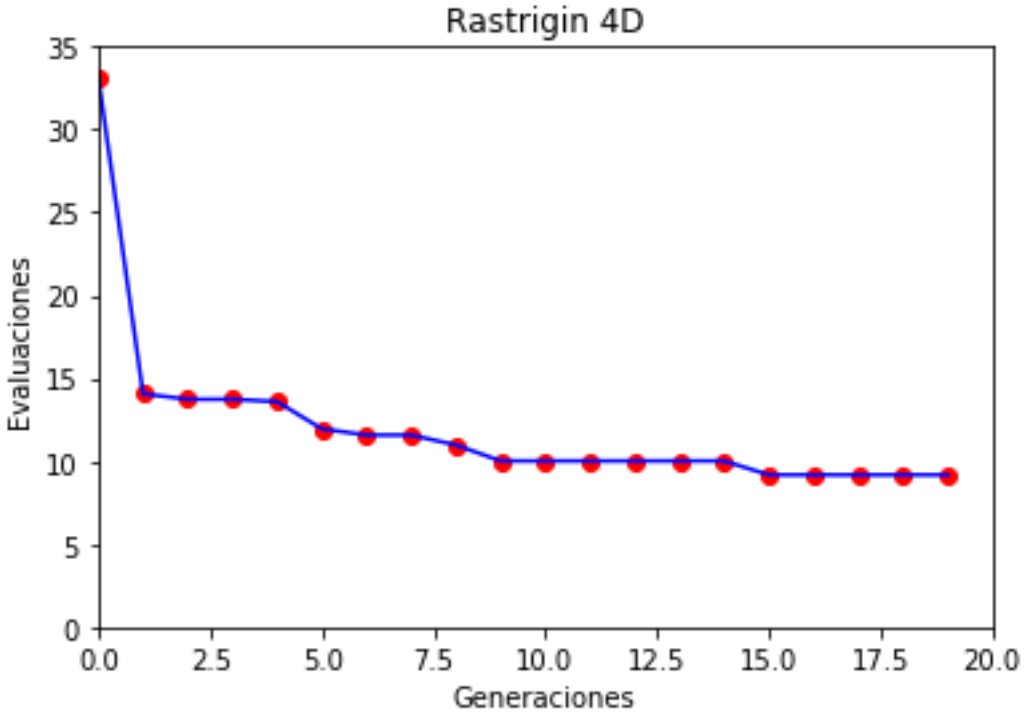

Rosenbrock 4D
Evaluaciones
Generaciones
120
100
80
60
40
20
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

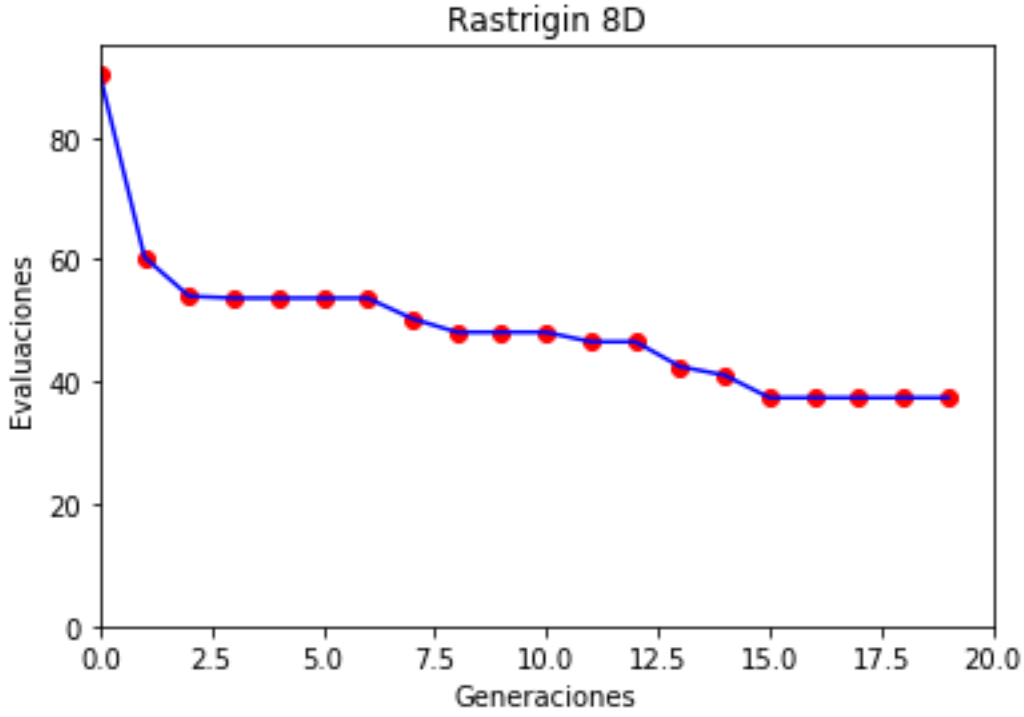

Rosenbrock 8D
Evaluaciones
Generaciones
0
100
200
300
400
500
600
700
800
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

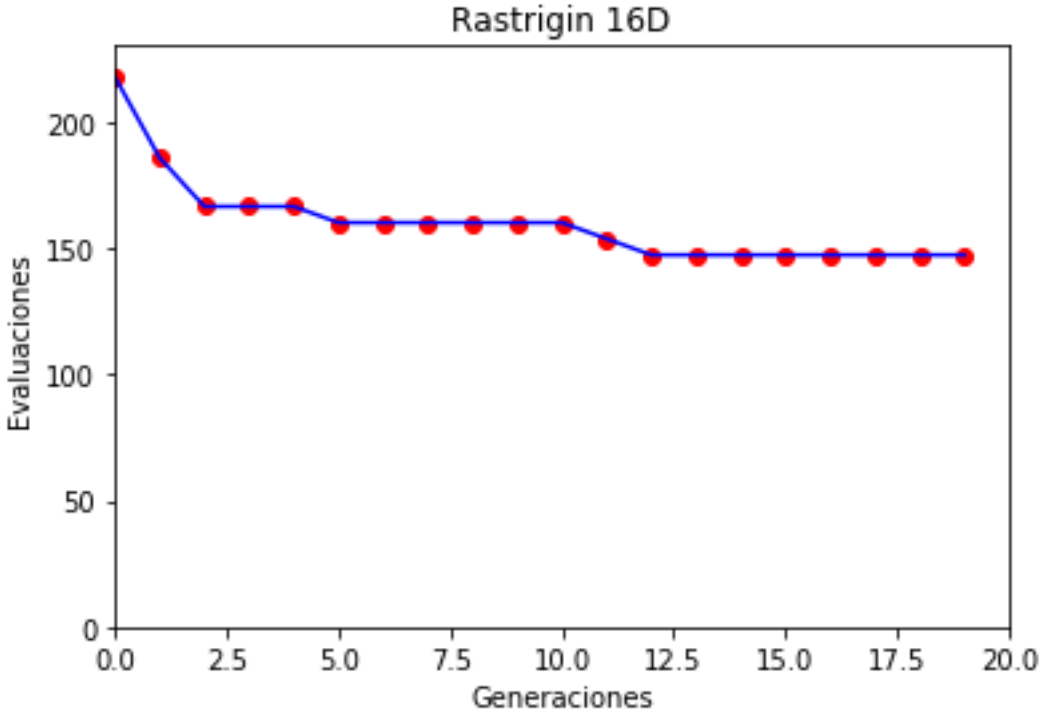

Rosenbrock 16D
Evaluaciones
Generaciones
0
500
1000
1500
2000
2500
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

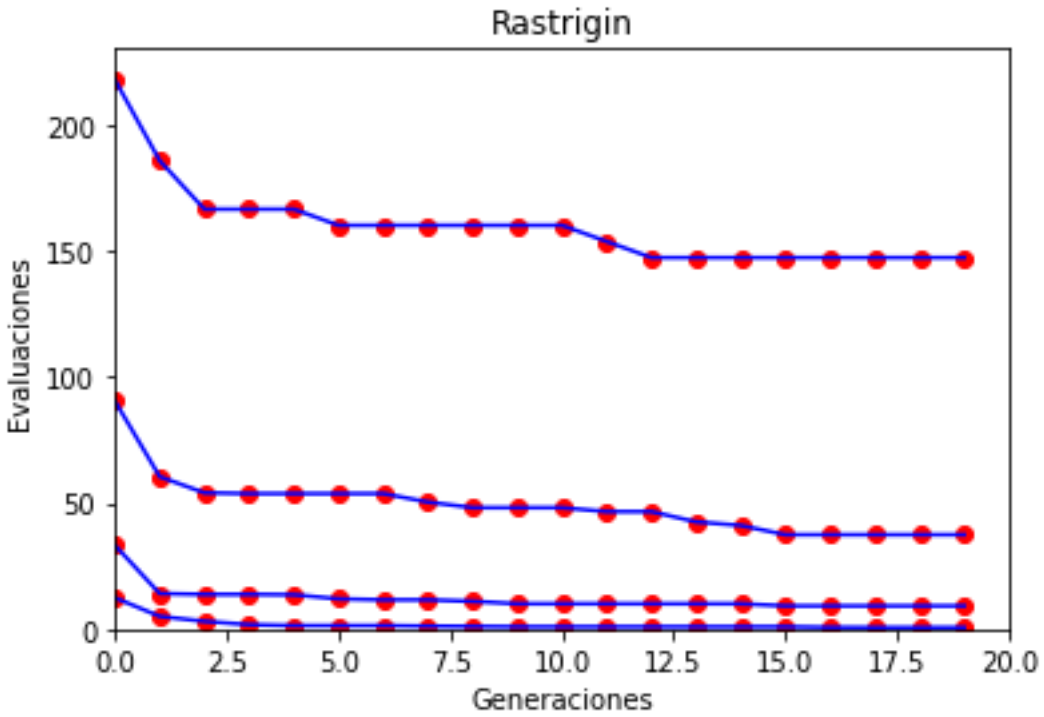

Rosenbrock
Evaluaciones
Generaciones
2500
2000
1500
1000
500
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

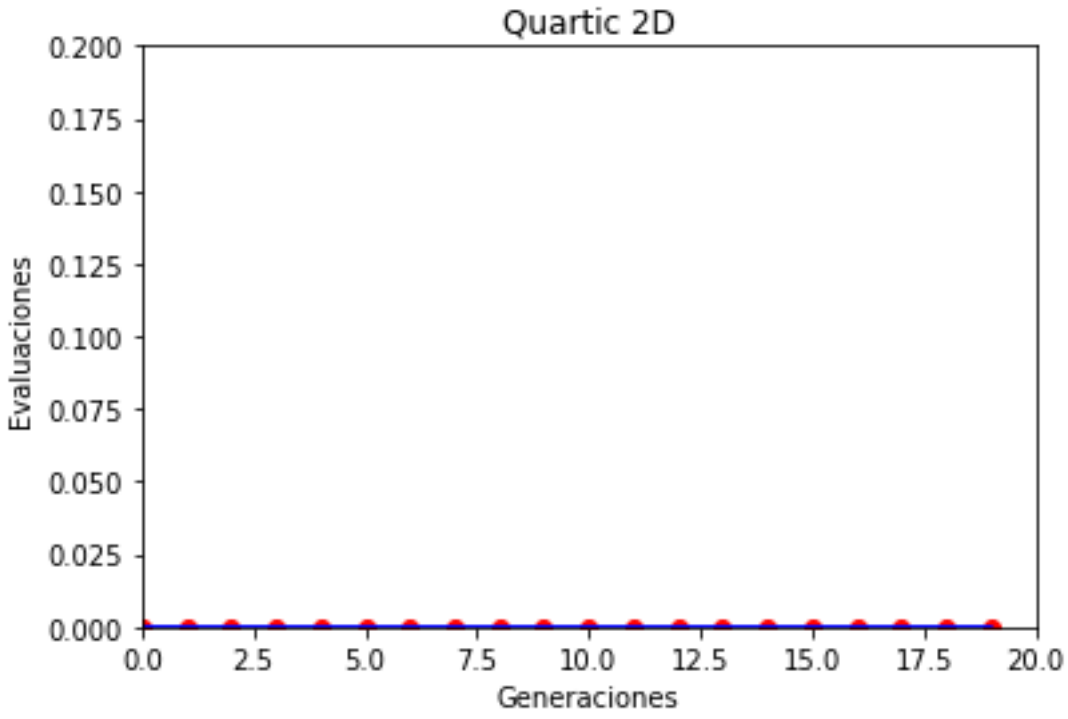

Rastrigin 2D
Evaluaciones
Generaciones
12
10
8
6
4
2
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

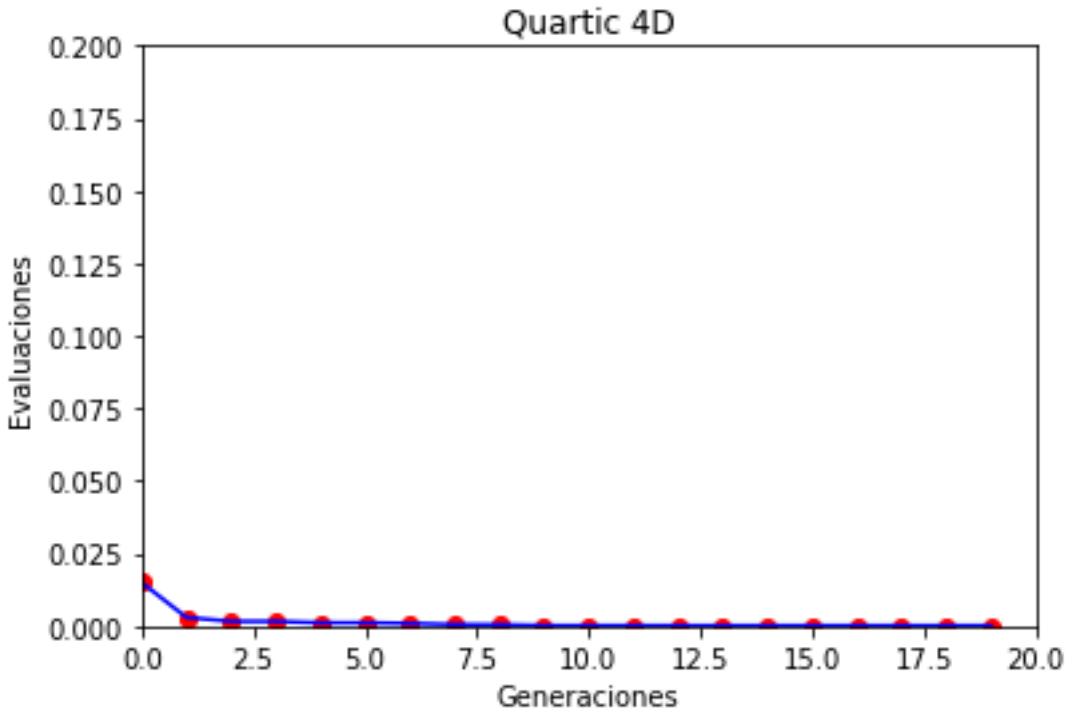

Rastrigin 4D
Evaluaciones
Generaciones
0
5
10
15
20
25
30
35
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

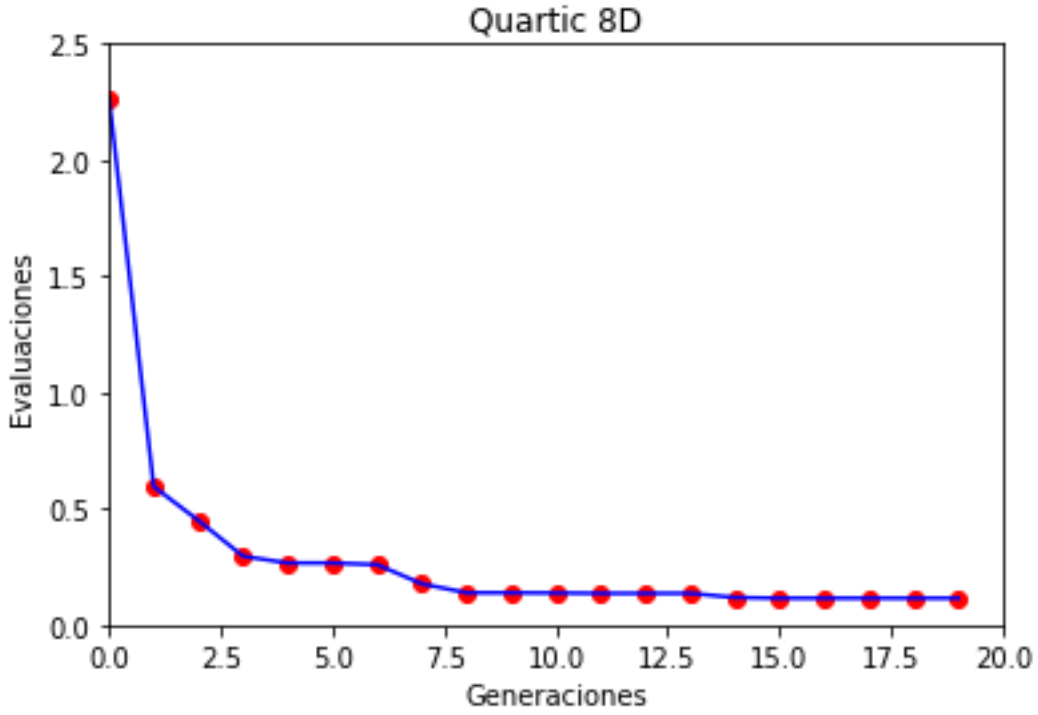

Rastrigin 8D
Evaluaciones
Generaciones
0
20
40
60
80
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

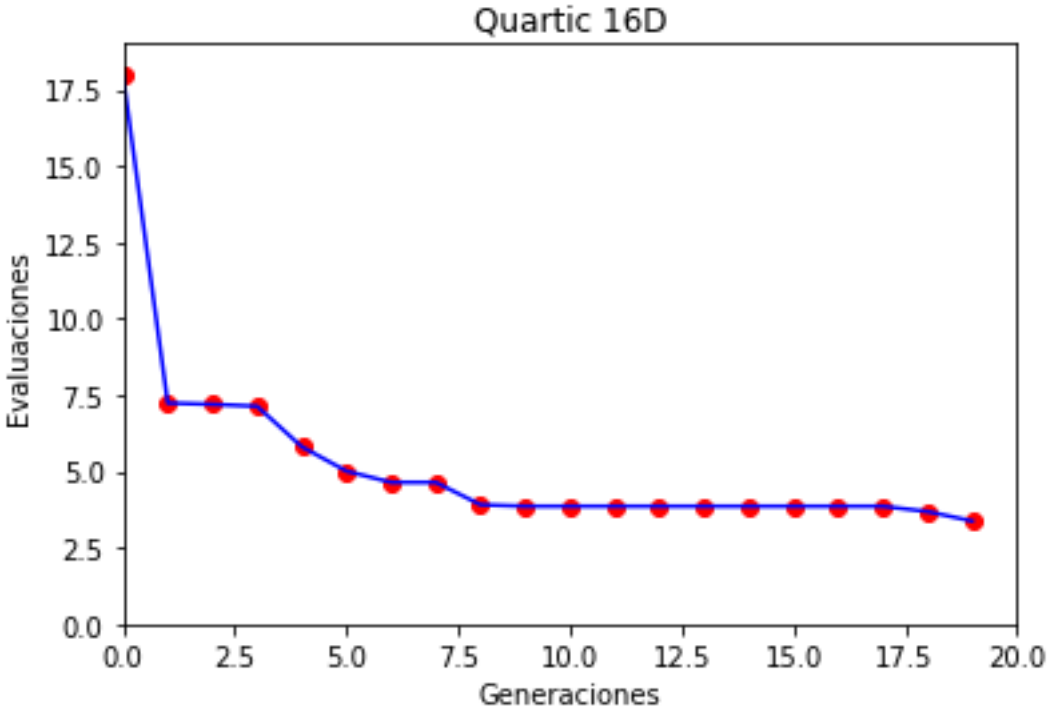

Rastrigin 16D
Evaluaciones
Generaciones
0
50
100
150
200
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

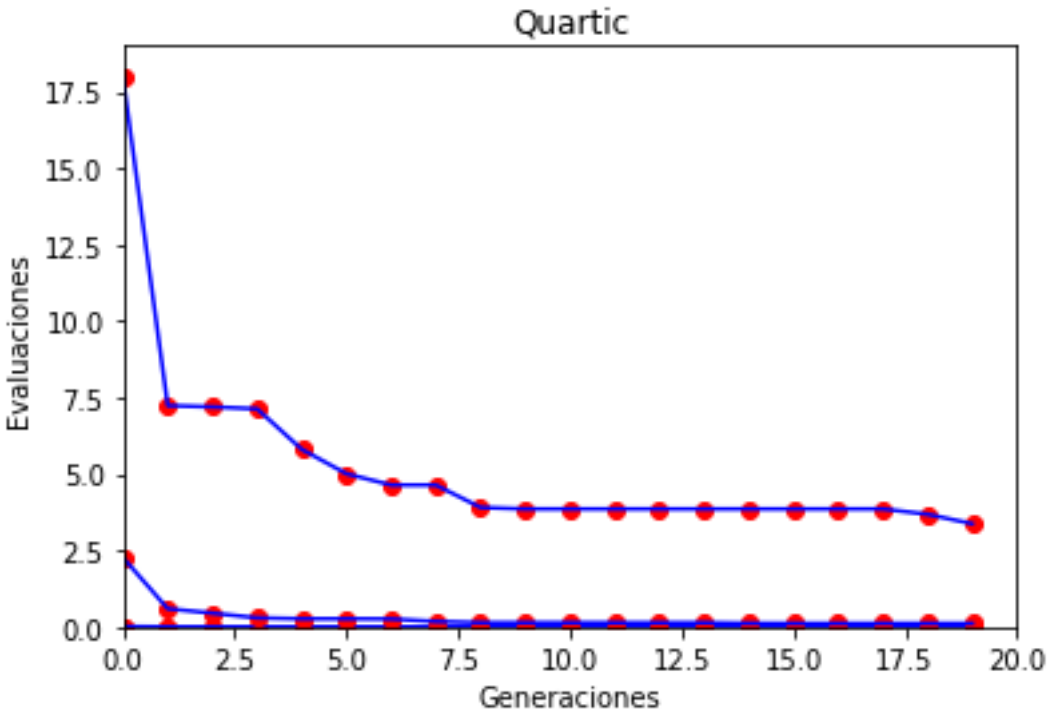

Rastrigin
Evaluaciones
Generaciones
0
50
100
150
200
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Quartic 2D
Evaluaciones
Generaciones
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Quartic 4D
Evaluaciones
Generaciones
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Quartic 8D
Evaluaciones
Generaciones
0.0
0.5
1.0
1.5
2.0
2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Quartic 16D
Evaluaciones
Generaciones
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0



Conclusiones

Resulta bastante interesante observar que en muchas de las ocasiones los resultados van mejorando conforme se realizan más ejecuciones; es decir, se puede ver que en la mayoría de las ecuaciones, en la primera ejecución el primer resultado es bastante alto, al igual que el último, y en las próximas ejecuciones del programa, esos mismos resultados son significativamente más bajos. Parecería como si el código siguiera mejorando aún al no estar activo.

Por otro lado, también es importante observar cómo es que, en muchas ocasiones, el algoritmo simplemente ya no llega a una solución óptima en concreto según se aumentan las dimensiones del problema. Esto es interesante porque resulta obvio, pero tampoco mejora aún si se aumentan las cantidades de individuos por generación, o si se juega un poco con el coeficiente de mutación. Entonces, parece que con esto sencillamente se puede observar que el algoritmo llega a su límite gradualmente, y para obtener mejores resultados, necesitaríamos sencillamente implementar mejores algoritmos.