

**Instituto Politécnico Nacional
Escuela Superior de Cómputo**



Evolutionary Computing

Lab Session 3: Introduction to Genetic Algorithms

Alumno: David Arturo Oaxaca Pérez

Grupo: 3CV11

Profesor: Jorge Luis Rosas Trigueros

Fecha de realización: 04/10/2021

Fecha de entrega: 08/10/2021

Marco teórico

En esta práctica se usarán algoritmos genéticos para la optimización, estos algoritmos son una serie de pasos que describe un proceso a seguir para llegar a una solución óptima, están inspirados en la evolución natural y la evolución genética.

Estos algoritmos empiezan con un conjunto inicial de soluciones potenciales codificadas que suelen ser generadas aleatoriamente a las que usualmente se les llaman cromosomas, estos se depuran hasta escoger la mejor. Cada una de estas soluciones potenciales es evaluada por una función de aptitud para determinar si es más apta con respecto a las demás soluciones, esto hace de los algoritmos genéticos un método de búsqueda basado en probabilidades.

Los candidatos prometedores (Cromosomas en este caso) se conservan muchas veces mediante elitismo y se “reproducen” y se aplicando operadores genéticos como lo es el crossover, además, en el proceso se pueden agregar mutaciones mediante probabilidad. Esto prosigue hasta crear una nueva generación de soluciones que podremos evaluar nuevamente.

Simplificando esto, los pasos que se siguen en un algoritmo genético son:

- **Inicialización:** Se genera una población inicial.
- **Evaluación:** Aplicación de una función de evaluación para los individuos de la población.
- **Evolución:** Aplicación de operadores genéticos como “crossover” y mutaciones.
- **Termino:** Se detiene el algoritmo usando criterios y determinando la mejor solución de la población.

En la siguiente página se puede ver el ciclo de un algoritmo genético:

Selección (Se) -> Cruzamiento (Cr) -> Mutación (Mu) -> Evaluación ($f(x)$) -> Reemplazo (Re).

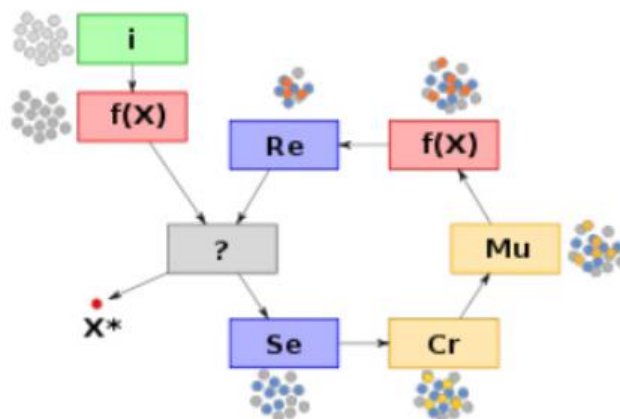


Figura 1. Ciclo del algoritmo genético

Algunos de los operadores genéticos ilustrados son los siguientes:

Crossover: En este operador genético ocurre un intercambio de información genética entre dos cromosomas.

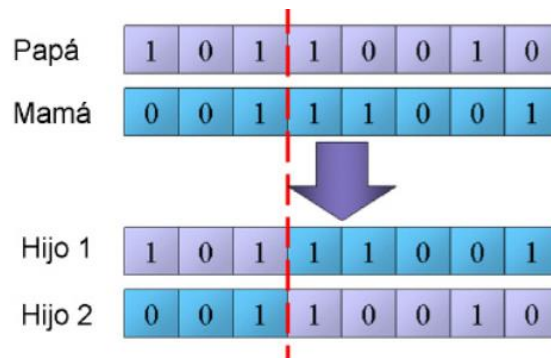


Figura 2. Ilustración del crossover.

Mutación: En este operador una parte del cromosoma se modifica al azar

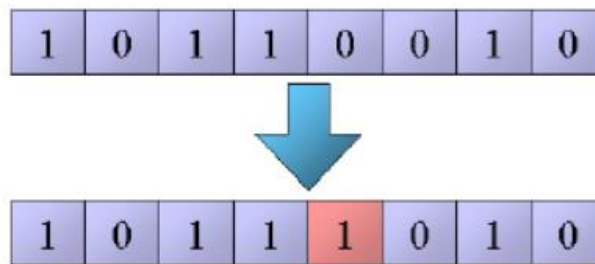


Figura 3. Ilustración de la mutación.

Funciones de para pruebas de optimización usadas para la práctica.

Función de Ackley:

Un dominio de dos dimensiones está definido por:

$$f(x, y) = -20 \exp \left[-0.2 \sqrt{0.5(x^2 + y^2)} \right] - \exp[0.5(\cos 2\pi x + \cos 2\pi y)] + e + 20$$

Esta función no convexa fue diseñada por David Ackley en 1987 para probar el rendimiento de algoritmos de optimización.

Los límites de evaluación del dominio son:

$$-5 \leq x, y \leq 5$$

El mínimo global optimo es:

$$f(0,0) = 0$$

La representación gráfica de la función es la siguiente:

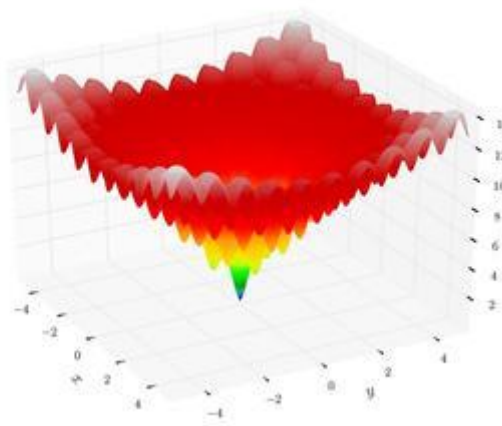


Figura 4. Grafica de la función de Ackley.

Función de Rastrigin:

En un dominio está definido por:

$$f(x, \dots, x_n) = 10n + \sum_{i=1}^n [x_i^2 - 10\cos(2\pi x_i)]$$

Donde n es el número de dimensiones.

Esta función no convexa fue diseñada por Rastrigin en 1974 para probar el rendimiento de algoritmos de optimización y fue propuesta originalmente para dos dimensiones, aunque posteriormente fue generalizada para n dimensiones.

Los límites de evaluación del dominio son:

$$x_i \in [-5.12, 5.12]$$

El mínimo global optimo es:

$$f(x^*) = 0 \text{ con } x^* = (0, \dots, 0)$$

La representación gráfica de la función es la siguiente:

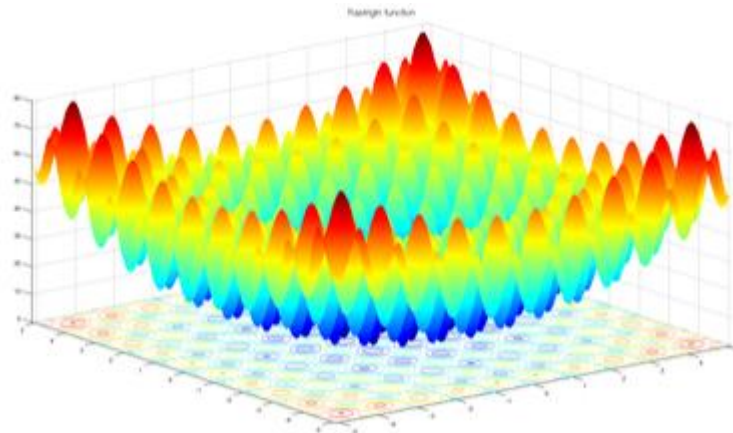


Figura 5. Grafica de la función de Rastrigin para 2 dimensiones.

Material y equipo

- El programa fue realizado en Google Colaboratory usando Python 3.9 como lenguaje.
- El sistema que se utilizó para acceder a Google Colaboratory fue una laptop de marca Lenovo, modelo Ideapad S540 con Ryzen 7 y 8 GB de RAM, con sistema operativo Windows 10.

Desarrollo de la practica

Para empezar a desarrollar esta práctica se utilizó el ejemplo proporcionado en clase como base.

Función de Ackley:

El primer cambio que se necesita, es principalmente la función, para esto cambiamos la función $f(x)$ por la `ackley_function(x, y)`, donde evaluaremos los valores que proporcione un cromosoma.

```
# Función a evaluar con el algoritmo
def ackley_function(x, y):
    #print('Ackley args: ', x, y)
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi * x) + cos(2 * pi * y))) + e + 20
```

Figura 6. Cambio para la función de Ackley.

Otro cambio importante fue el “diseño” del cromosoma, en este caso, tenemos que encontrar un punto en un plano de dos dimensiones y evaluar este cromosoma con la función de Ackley. Para esto el cromosoma será una lista con dos listas, es decir, estará compuesto por dos cadenas y cada una de ellas será de un largo de 8 bits, para tener más precisión en la evaluación de funciones, estableceremos los límites como $a = -5$, $b = 5$, al decodificar nuestros cromosomas tendremos un valor en esa distancia.

Se usarán 10 cromosomas para este problema y una probabilidad de mutación de 0.5.

```
'''
Chromosomes are 4 bits long but are composed of two strings to
represent a point in R^2
'''

L_chromosome=8
N_chains=2**L_chromosome
#Lower and upper limits of search space
a=-5
b=5
crossover_point=int(L_chromosome/2)

#Number of chromosomes
N_chromosomes=10
#probability of mutation
prob_m=0.5
```

Figura 7. Algunas de las variables como el largo de cada cadena del cromosoma, entre otros.

```
# Función para crear un cromosoma random

def random_chromosome():
    chromosome=[[ ],[ ]]
    for i in range(0,L_chromosome):
        if random.random()<0.5:
            chromosome[0].append(0)
        else:
            chromosome[0].append(1)

    for i in range(0,L_chromosome):
        if random.random()<0.5:
            chromosome[1].append(0)
        else:
            chromosome[1].append(1)

    return chromosome
```

Figura 8. Función donde se crean los cromosomas usando dos cadenas como si fuera una sola cadena más larga

Cabe mencionar que ambas cadenas de un solo cromosoma serán trabajadas como una cadena más larga, porque estando partidas será más fácil trabajarlas, en la función de crear cromosomas se puede ver cómo ambas cadenas son llenadas con valores aleatorios para darles más variedad.

Hay que resaltar que las funciones para evaluar cromosomas, compararlos y decodificarlos se modifican para trabajar con dos cadenas añadiendo valores adicionales para que cada uno trabaje con una cadena, pero se realice el mismo proceso.

La función para crear una rueda para la nueva generación permanece igual, ya que la lista de valores fitness no se modifican, se añaden valores de los cromosomas evaluados con la función de Ackley únicamente.

En la función de nextgeneration() usada para crear nuevas generaciones (que sean más aptas para resolver el problema) tiene también varias modificaciones, primero quitamos los elementos respectivos del botón, ya que ejecutaremos varias generaciones con un ciclo en vez de ir apretando un botón. El siguiente cambio importante es el crossover, para realizarlo hacemos que las variables o1 y o2 sean una lista que contiene dos listas, es decir, las dos cadenas de un cromosoma y trabajamos con cada una de estas cadenas para que ambas tengan un intercambio de información. Lo mismo se hace para la mutación.

```
#Two descendants are generated
o1=[F0[p1][0][0:crossover_point], F0[p1][1][0:crossover_point]]
o1[0].extend(F0[p2][0][crossover_point:L_chromosome])
o1[1].extend(F0[p2][1][crossover_point:L_chromosome])

o2=[F0[p2][0][0:crossover_point], F0[p2][1][0:crossover_point]]
o2[0].extend(F0[p1][0][crossover_point:L_chromosome])
o2[1].extend(F0[p1][1][crossover_point:L_chromosome])

#Each descendant is mutated with probability prob_m
if random.random() < prob_m:
    o1[0][int(round(random.random()*(L_chromosome-1)))]^=1
    o1[1][int(round(random.random()*(L_chromosome-1)))]^=1
if random.random() < prob_m:
    o2[0][int(round(random.random()*(L_chromosome-1)))]^=1
    o2[1][int(round(random.random()*(L_chromosome-1)))]^=1
#The descendants are added to F1
F1[2+2*i]=o1
F1[3+2*i]=o2
```

Figura 9. Modificaciones para el crossover y la mutación en la función nextgeneration().

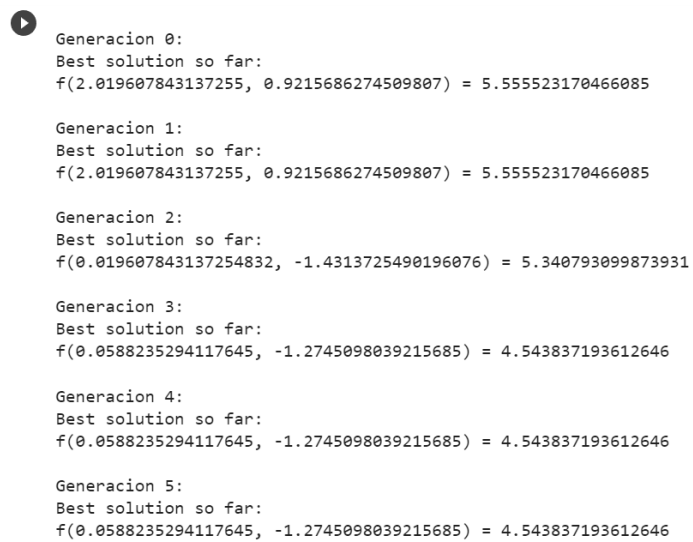
Finalmente, iteramos a través de 100 generaciones para obtener el resultado más óptimo para este problema.

```
F0.sort( key=cmp_to_key(compare_chromosomes))
evaluate_chromosomes()

for i in range(0, 100):
    print(f"\nGeneracion {i}:")
    nextgeneration()
```

Figura 10. Iteración para las generaciones de cromosomas.

Finalmente, podemos observar en los resultados y ver que el resultado va mejorando y se va acercando al mínimo de la función como se puede observar en las siguientes imágenes:



```

Generacion 0:
Best solution so far:
f(2.019607843137255, 0.9215686274509807) = 5.555523170466085

Generacion 1:
Best solution so far:
f(2.019607843137255, 0.9215686274509807) = 5.555523170466085

Generacion 2:
Best solution so far:
f(0.019607843137254832, -1.4313725490196076) = 5.340793099873931

Generacion 3:
Best solution so far:
f(0.0588235294117645, -1.2745098039215685) = 4.543837193612646

Generacion 4:
Best solution so far:
f(0.0588235294117645, -1.2745098039215685) = 4.543837193612646

Generacion 5:
Best solution so far:
f(0.0588235294117645, -1.2745098039215685) = 4.543837193612646
```

Figura 11. Las primeras iteraciones con un mínimo que no es el más óptimo y mejorara en las siguientes.


```

▶ Generacion 7:
Best solution so far:
f(0.0588235294117645, -1.2745098039215685) = 4.543837193612646

Generacion 8:
Best solution so far:
f(0.13725490196078471, -0.019607843137254832) = 0.8326667536233607

Generacion 9:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 10:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 11:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 12:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

```

Figura 12. El mejor resultado va mejorando conforme van avanzando las generaciones.

```

↳ Generacion 65:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 66:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 67:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 68:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 69:
Best solution so far:
f(0.0588235294117645, -0.019607843137254832) = 0.27479954703707676

Generacion 70:
Best solution so far:
f(0.019607843137254832, -0.019607843137254832) = 0.09880309325856729

```

Figura 13. El mejor resultado va mejorando conforme van avanzando las generaciones y se acerca aún más al mínimo deseado.

```

Generacion 94:
Best solution so far:
f(0.019607843137254832, -0.019607843137254832) = 0.09880309325856729

Generacion 95:
Best solution so far:
f(0.019607843137254832, -0.019607843137254832) = 0.09880309325856729

Generacion 96:
Best solution so far:
f(0.019607843137254832, -0.019607843137254832) = 0.09880309325856729

Generacion 97:
Best solution so far:
f(0.019607843137254832, -0.019607843137254832) = 0.09880309325856729

Generacion 98:
Best solution so far:
f(0.019607843137254832, -0.019607843137254832) = 0.09880309325856729

Generacion 99:
Best solution so far:
f(0.019607843137254832, -0.019607843137254832) = 0.09880309325856729

```

Figura 14. El mejor resultado final al llegar a la última generación planeada, se acerca mucho más al mínimo de la función que las primeras generaciones.

En este caso, no se podrá llegar al cero exacto ya que la decodificación del cromosoma para ser evaluado en la función contemplará un punto entre a y b que no será tan preciso, que se acerca bastante porque el algoritmo encuentra el mínimo en los puntos de dicho intervalo.

Función de Rastrigin:

La realización del algoritmo genético para esta función fue mucho más sencilla, pues todos los cambios importantes que se necesitan ya fueron pensados para la de Ackley, así que en este caso solo se trabajara cada cromosoma con 3 cadenas en vez de 2. Empezamos cambiando la función que teníamos previamente por la de Rastrigin la cual luce así:

```

def rastrigin_function(x, y, z):
    return 10*3 + (x**2 - 10*np.cos(2*np.pi*x)) + \
        (y**2 - 10*np.cos(2*np.pi*y)) + \
        (z**2 - 10*np.cos(2*np.pi*z))

```

Figura 15. Función de Rastrigin dadas tres dimensiones que se usara para evaluar los cromosomas

Si bien esta función se podría generalizar para n dimensiones, en este caso se usó así para poder realizar pruebas más seguras (en caso de que algo fallara) con las tres variables definidas.

Se modificaron todas las funciones que tenían que trabajar con los cromosomas para que operaran tres cadenas, por ejemplo, en la de `decode_chromosome` se devuelven tres valores, para poder evaluar a la función.

En la siguiente imagen podemos ver la creación de un cromosoma de manera aleatoria, con 3 cadenas, donde éstas representarían un punto en el plano R^3 . Además de ello, se usará una población de 10 cromosomas.

```
'''
Chromosomes are 4 bits long but are composed of three strings to
represent a point in R^2
'''

L_chromosome=8
N_chains=2**L_chromosome
#Lower and upper limits of search space
a=-5.12
b=5.12
crossover_point=int(L_chromosome/2)

#Number of chromosomes
N_chromosomes=10
#probability of mutation
prob_m=0.5
```

Figura 16. Características de los cromosomas y los operadores genéticos como la mutación

```
# Función para crear un cromosoma random

def random_chromosome():
    chromosome=[[],[],[ ]]
    for i in range(0,L_chromosome):
        if random.random()<0.5:
            chromosome[0].append(0)
        else:
            chromosome[0].append(1)

    for i in range(0,L_chromosome):
        if random.random()<0.5:
            chromosome[1].append(0)
        else:
            chromosome[1].append(1)

    for i in range(0,L_chromosome):
        if random.random()<0.5:
            chromosome[2].append(0)
        else:
            chromosome[2].append(1)

    return chromosome
```

Figura 17. Función para la creación de un cromosoma para la función de Rastrigin.

Al igual que en la función de Ackley, las tres cadenas se trabajarán como si fuera una sola, pero esta se divide en 3 para que se pueda operar de manera fácil y sin usar tantos índices.

El siguiente cambio se dio precisamente para estos operadores genéticos, el cambio fue parecido al de Ackley pero para trabajar con 3 cadenas en cada cromosoma, como se puede ver a continuación.

```
#Two descendants are generated
o1=[F0[p1][0][0:crossover_point], F0[p1][1][0:crossover_point], F0[p1][2][0:crossover_point]]
o1[0].extend(F0[p2][0][crossover_point:L_chromosome])
o1[1].extend(F0[p2][1][crossover_point:L_chromosome])
o1[2].extend(F0[p2][2][crossover_point:L_chromosome])

o2=[F0[p2][0][0:crossover_point], F0[p2][1][0:crossover_point], F0[p2][2][0:crossover_point]]
o2[0].extend(F0[p1][0][crossover_point:L_chromosome])
o2[1].extend(F0[p1][1][crossover_point:L_chromosome])
o2[2].extend(F0[p1][2][crossover_point:L_chromosome])

#Each descendant is mutated with probability prob_m
if random.random() < prob_m:
    o1[0][int(round(random.random()*(L_chromosome-1)))]^=1
    o1[1][int(round(random.random()*(L_chromosome-1)))]^=1
    o1[2][int(round(random.random()*(L_chromosome-1)))]^=1
if random.random() < prob_m:
    o2[0][int(round(random.random()*(L_chromosome-1)))]^=1
    o2[1][int(round(random.random()*(L_chromosome-1)))]^=1
    o2[2][int(round(random.random()*(L_chromosome-1)))]^=1
#The descendants are added to F1
F1[2+2*i]=o1
F1[3+2*i]=o2
```

Figura 18. Cambios para los operadores genéticos en la función nextgeneration() para la función de Rastrigin.

A continuación, el algoritmo se correrá para 500 generaciones mediante un ciclo como se puede ver a continuación, ya que es más complicado que los tres puntos a evaluar coincidan en comparación con la de Ackley.

```
F0.sort( key=cmp_to_key(compare_chromosomes))
evaluate_chromosomes()

for i in range(0, 500):
    print(f"\nGeneracion {i}:")
    nextgeneration()
```

Figura 19. Iteración para las generaciones de cromosomas para la función de Rastrigin.

Finalmente, podemos observar en los resultados y ver que el resultado va mejorando entre más generaciones van pasando:

▶ Generacion 1:
Best solution so far:
 $f(-0.060235294117647165, -0.2208627450980396, 3.1924705882352926) = 25.59494949328375$

Generacion 2:
Best solution so far:
 $f(-0.060235294117647165, -0.2208627450980396, 3.1924705882352926) = 25.59494949328375$

Generacion 3:
Best solution so far:
 $f(-0.060235294117647165, -0.2208627450980396, 3.1924705882352926) = 25.59494949328375$

Generacion 4:
Best solution so far:
 $f(-0.060235294117647165, -0.2208627450980396, 3.1924705882352926) = 25.59494949328375$

Generacion 5:
Best solution so far:
 $f(-0.060235294117647165, -0.2208627450980396, 3.1924705882352926) = 25.59494949328375$

Generacion 6:
Best solution so far:
 $f(-0.060235294117647165, -0.2208627450980396, 3.1924705882352926) = 25.59494949328375$

Figura 20. Primeras generaciones de soluciones que aún se alejan bastante de la solución deseada.

▶ Generacion 94:
Best solution so far:
☞ $f(-0.020078431372549055, -0.020078431372549055, -1.9877647058823529) = 4.140494000351977$

Generacion 95:
Best solution so far:
 $f(-0.020078431372549055, -0.020078431372549055, -1.9877647058823529) = 4.140494000351977$

Generacion 96:
Best solution so far:
 $f(-0.020078431372549055, -0.020078431372549055, -1.9877647058823529) = 4.140494000351977$

Generacion 97:
Best solution so far:
 $f(-0.020078431372549055, -0.020078431372549055, -1.9877647058823529) = 4.140494000351977$

Generacion 98:
Best solution so far:
 $f(-0.020078431372549055, -0.020078431372549055, -1.9877647058823529) = 4.140494000351977$

Generacion 99:
Best solution so far:
 $f(-0.020078431372549055, -0.020078431372549055, -1.9877647058823529) = 4.140494000351977$

Generacion 100:
Best solution so far:
 $f(-0.020078431372549055, -0.020078431372549055, -1.9877647058823529) = 4.140494000351977$

Figura 21. Mejores resultados 100 generaciones después.

```

Generacion 494:
Best solution so far:
f(-0.020078431372549055, -0.020078431372549055, -0.020078431372549055) = 0.2396249269381272

Generacion 495:
Best solution so far:
f(-0.020078431372549055, -0.020078431372549055, -0.020078431372549055) = 0.2396249269381272

Generacion 496:
Best solution so far:
f(-0.020078431372549055, -0.020078431372549055, -0.020078431372549055) = 0.2396249269381272

Generacion 497:
Best solution so far:
f(-0.020078431372549055, -0.020078431372549055, -0.020078431372549055) = 0.2396249269381272

Generacion 498:
Best solution so far:
f(-0.020078431372549055, -0.020078431372549055, -0.020078431372549055) = 0.2396249269381272

Generacion 499:
Best solution so far:
f(-0.020078431372549055, -0.020078431372549055, -0.020078431372549055) = 0.2396249269381272

```

Figura 22. La mejor solución después de 500 generaciones que se aproxima al 0

Como podemos ver, igual que en la de Ackley, después de varias generaciones se acerca más al mínimo de la función hasta donde el intervalo en el que es evaluado el cromosoma lo permite.

Conclusiones y recomendaciones

Esta práctica ayudo en muchos sentidos a entender mejor como se diseña un algoritmo genético y como se opera en sus diferentes etapas, los casos de la población se probaron con más cromosomas y demás, pero al final se usó la población establecida en el ejemplo para ver los cambios a través de las generaciones.

Una de las formas que se encontraron para ir entendiendo cuales iban a ser los cambios que debían realizar para llegar al mínimo de las funciones fue hacer bastantes pruebas, estuve cambiando las poblaciones y viendo cuales eran las formas en que se podía implementar el algoritmo para trabajar con más dimensiones, como el uso de una cadena más larga y operar con índices o hacer uso de más poblaciones para x y para y, en si fue una práctica en la que se hizo bastante experimentación para corregir errores que iba encontrando e ir viendo llegar a la mejor solución implementando un algoritmo genético. También ayudo bastante para el entendimiento del uso de operadores genéticos y no quedarnos estancados en un mínimo local.

Bibliografía

Garduño Juárez, R. (2018, 21 septiembre). *Algoritmos genéticos*. Conogasi.

<http://conogasi.org/articulos/algoritmos-geneticos/>

Carpitella, S. (s. f.). *Funciones test para optimización mono-objetivo*. Universidad

Politecnica de Valencia. Recuperado 4 de octubre de 2021, de

[https://m.riunet.upv.es/bitstream/handle/10251/105210/Izquierdo%3BCARPITELL](https://m.riunet.upv.es/bitstream/handle/10251/105210/Izquierdo%3BCARPITELLA%20-%20Funciones%20test%20para%20optimizaci%C3%B3n%20mono-objetivo.pdf?sequence=1&isAllowed=y)

[A%20-%20Funciones%20test%20para%20optimizaci%C3%B3n%20mono-
objetivo.pdf?sequence=1&isAllowed=y](https://m.riunet.upv.es/bitstream/handle/10251/105210/Izquierdo%3BCARPITELLA%20-%20Funciones%20test%20para%20optimizaci%C3%B3n%20mono-objetivo.pdf?sequence=1&isAllowed=y)