Instituto Politécnico Nacional
Escuela Superior de Cómputo
Evolutionary Computing
3 Introduction to Genetic Algorithms
Martínez Coronel Brayan Yosafat
Rosas Trigueros Jorge Luis
24/09/2021
8/10/2021

## Theorical framework

In this practice, we try to find a function minimum using genetic algorithms, the functions we are talking about are Ackley and Rastrigin, both are used for optimizations test. But let us talk about them deeper.

Used widely in problem optimization, Ackley function is characterized by a nearly flat outer region, and a large hole at the center. The function poses a risk for optimization algorithms, particularly hill climbing algorithms, to be trapped in one of its many local minima [1].

$$f(x) = -Ae^{-B\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}} - e^{\frac{1}{d}\sum_{i=1}^{d}\cos(Cx_i)}A + e \qquad (1)$$

Proposed by Rastrigin in 1974 as a 2-D function for optimization problems and generalized later by Rudolph and then popularized by Hoffmeister & Bäck. This function is hard to find a minimum for two reasons, the domain, and the number of local minima [2].

$$f(x) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)] \qquad (2)$$

Genetic algorithms are inspired by natural selection, they belong to the evolutionary algorithms class. Used for optimization with good solutions and search problems, they relied on biological inspired operators such as mutation, crossover, and selection. Based on a population of individuals representing candidates for solutions, created by random methods, and then iterated over time, we finally come to a solution (or more) to resolve the problem. Best candidates are selected every iteration, but these rules are general and can be modified according to the problem to resolve [3].

In general terms, with a population, at the end of each iteration (called generation), individuals are evaluated with a function (called fitness function, because it checks how much an individual fits the solution) and then, using biological inspired operators, the next generation is created. Over the time, result come to improve and converge (if there is a solution) [4].

## Material and equipment

In this practice, DataSpell with Jupiter Notebooks, Conda and Python were used for development, all of them were run in a personal computer which has an Intel Core i7 with Windows 10.

## Practice development

### Ackley function

Now, let us define a framework to work with the algorithms using OOP, this attemps to reduce considerably the time spent at developing and just making changes between versions of the problem. At the beginning it will slow, but, over the time, development will increase at speed notoriously.

We separate four major sections about each problem, the fitness function also known as Ackley for this problem and Rastrigin for the second, the chromosome class, the selective pressure and, at the end, the iteration code. Now, this let us observe in detail what is happening behind the scenes, let's see what are we refering to.

```
import numpy as np

K_A = 20
K_B = 0.2
K_C = 2 * np.pi

def ackley(x: np.ndarray, y: np.ndarray) -> np.ndarray:
    return -K_A * np.exp(-K_B * np.sqrt(0.5 * (np.power(x, 2) +
np.power(y, 2)))) \
           -np.exp(0.5 * (np.cos(K_C * x) + np.cos(K_C * y))) \
           + K_A + np.exp(1)
```

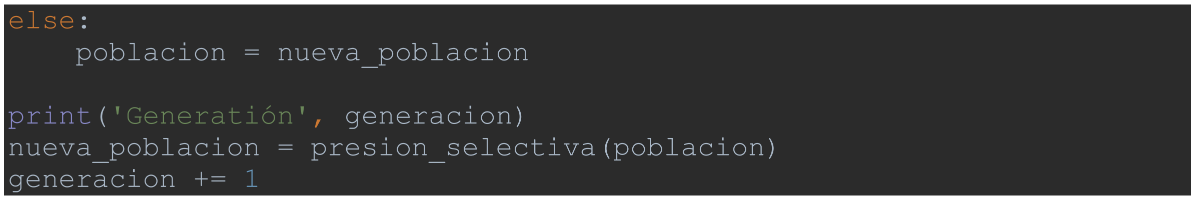
Code 1. Ackley function

```
BITS = 10      # Con signo, usados 9 para el número, no se usa
complemento a 2
POWERS = np.array([2 ** i for i in range(BITS-2, -1, -1)])

class Cromosoma:
    """
    Se conforma por una lista de bits, sirve como index lógico de
POWERS
    """
    def __init__(self, list = None):
        if list is None:
            self.list = np.array([np.random.choice([0, 1], p=[0.3,
0.7]) for i in range(0, BITS)])
        else:
            self.list = list

    def __str__(self):
        return str(self.list)


    def to_int(self):
        """
        Función para convertir los bits en suma de potencias de 2
        :return: entero de numpy
        """
        signo = 1 if self.list[0] == 0 else -1
        index = self.list[1:] == 1
        return signo * np.sum(POWERS[index])

    @staticmethod
    def crossover(c1, c2):
        mitad = int(BITS / 2)
        l1 = np.append(c1.list[0:mitad].copy(), c2.list[mitad:],
axis=None)
        l2 = np.append(c2.list[0:mitad].copy(), c1.list[mitad:],
axis=None)
        return [Cromosoma(l1), Cromosoma(l2)]

    @staticmethod
    def mutar(c):
        index = np.random.randint(low=0, high=len(c.list))
        c.list[index] = 1 - c.list[index]
```
Code 2. Chromosome class definition

```
K_POBLACION = 10
K_BASE = 2

def presion_selectiva(poblacion: list[Cromosoma]) -> list[Cromosoma]:
    # Evaluación y búsqueda del mejor
    x = np.array([c.to_int(0) for c in poblacion])
    y = np.array([c.to_int(1) for c in poblacion])
    evaluacion = ackley(x, y)

    best = evaluacion.argmin()
    print("Best so far:")
    print("Value: " + str(poblacion[best]))
    print(f"Integer value: {x[best]}, {y[best]}")
    print("Rastrigin function value: " + str(evaluacion[best]))

    # Cálculo de probabilidades
    indice_ordenado = evaluacion.argsort()
    ruleta = []
    potencia = K_POBLACION

    for i in indice_ordenado:
        probabilidad = K_BASE ** potencia
        ruleta.extend([i] * probabilidad)
        potencia -= 1

    # Nueva generación
    nueva = list[Cromosoma]()
    nueva.append(poblacion[indice_ordenado[0]])
    nueva.append(poblacion[indice_ordenado[1]])

    for i in range(2, int(K_POBLACION/2)):
        c1 = poblacion[np.random.choice(ruleta)]
        c2 = poblacion[np.random.choice(ruleta)]
        hijos = Cromosoma.crossover(c1, c2)

        hijos[0] = Cromosoma.mutar(hijos[0])
        hijos[1] = Cromosoma.mutar(hijos[1])

        nueva.extend(hijos)

    return nueva
```
Code 3. Selective pressure definition

```
poblacion = list[Cromosoma]()
nueva_poblacion = list[Cromosoma]()
generacion = 0
```

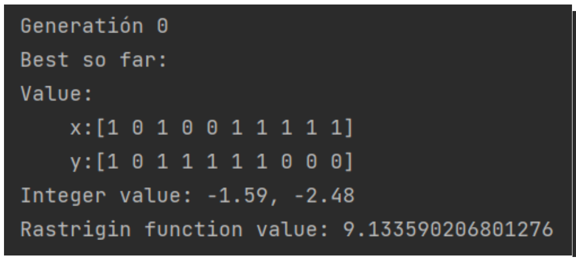
```
if len(nueva_poblacion) == 0:
    poblacion = [Cromosoma() for _ in range(0, K_POBLACION)]
```



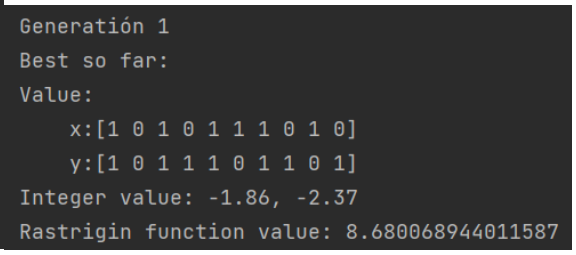
```
else:
    poblacion = nueva_poblacion

print('Generatión', generacion)
nueva_poblacion = presion_selectiva(poblacion)
generacion += 1
```

Code 4 and 5. Settings for runing and iteration

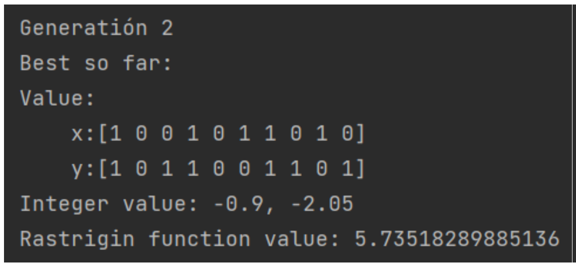
Talking about results, the Ackley function did not take many generations to converge, observe that detail because that changes a lot in Rastrigin function.



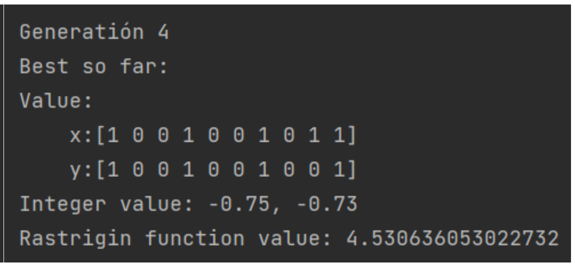
```
Generatión 0
Best so far:
Value:
    x:[1 0 1 0 0 1 1 1 1 1]
    y:[1 0 1 1 1 1 1 0 0 0]
Integer value: -1.59, -2.48
Rastrigin function value: 9.133590206801276
```



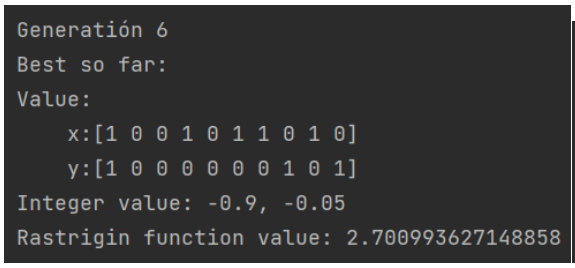
```
Generatión 1
Best so far:
Value:
    x:[1 0 1 0 1 1 1 0 1 0]
    y:[1 0 1 1 1 0 1 1 0 1]
Integer value: -1.86, -2.37
Rastrigin function value: 8.680068944011587
```



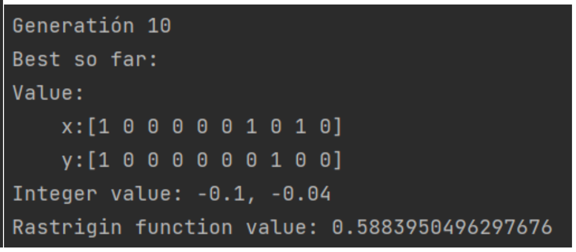
```
Generatión 2
Best so far:
Value:
    x:[1 0 0 1 0 1 1 0 1 0]
    y:[1 0 1 1 0 0 1 1 0 1]
Integer value: -0.9, -2.05
Rastrigin function value: 5.73518289885136
```



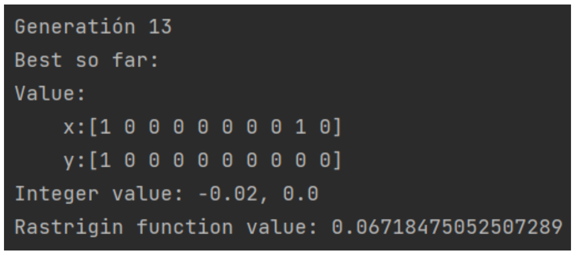
```
Generatión 4
Best so far:
Value:
    x:[1 0 0 1 0 0 1 0 1 1]
    y:[1 0 0 1 0 0 1 0 0 1]
Integer value: -0.75, -0.73
Rastrigin function value: 4.530636053022732
```



```
Generatión 6
Best so far:
Value:
    x:[1 0 0 1 0 1 1 0 1 0]
    y:[1 0 0 0 0 0 0 1 0 1]
Integer value: -0.9, -0.05
Rastrigin function value: 2.700993627148858
```



```
Generatión 10
Best so far:
Value:
    x:[1 0 0 0 0 0 1 0 1 0]
    y:[1 0 0 0 0 0 0 1 0 0]
Integer value: -0.1, -0.04
Rastrigin function value: 0.5883950496297676
```



```
Generatión 13
Best so far:
Value:
    x:[1 0 0 0 0 0 0 0 1 0]
    y:[1 0 0 0 0 0 0 0 0 0]
Integer value: -0.02, 0.0
Rastrigin function value: 0.06718475052507289
```

Fig. 1, 2, 3, 4, 5, 6 and 7. Results from Ackley optimization

## Rastrigin function

Following the framework of defining independent parts of the code, we show the function definition, the chromosome class, the selection pressure and the code for running it.

Function definition

```
import numpy as np

K_A = 10

def rastrigin(x: np.ndarray, y: np.ndarray, z: np.ndarray) -> 
np.ndarray:
    return K_A * 3 \
           + np.power(x, 2) - K_A * np.cos(2*np.pi * x) \
           + np.power(y, 2) - K_A * np.cos(2*np.pi * y) \
           + np.power(z, 2) - K_A * np.cos(2*np.pi * z)
```

Code 6. Rastrigin function definition

Chromosome class definition

```
BITS = 10
DIVISION = 100
POWERS = np.array([2 ** © for © in range(BITS-2, -1, -1)])

class Cromosoma:
    """
    Se conforma por dos listas de bits, sirven como index lógico de 
POWERS
    """
    def __init__(self, *args):
        if len(args) == 0:
            self.x = np.array([np.random.choice([0, 1], p=[0.3, 0.7]) 
for © in range(0, BITS)])
            self.y = np.array([np.random.choice([0, 1], p=[0.3, 0.7]) 
for © in range(0, BITS)])
            self.z = np.array([np.random.choice([0, 1], p=[0.3, 0.7]) 
for © in range(0, BITS)])
        else:
            self.x = args[0]
            self.y = args[1]
            self.z = args[2]

    def __str__(self):
        return f'\n\tx:{self.x} \n\ty:{self.y} \n\tz:{self.z}'


    def to_int(self, index):
        if index == 0:
            signo = 1 if self.x[0] == 0 else -1
            index = self.x[1:] == 1
            return signo * np.sum(POWERS[index]) / DIVISION
        elif index == 1:
            signo = 1 if self.y[0] == 0 else -1
            index = self.y[1:] == 1
            return signo * np.sum(POWERS[index]) / DIVISION
        else:
            signo = 1 if self.z[0] == 0 else -1
            index = self.z[1:] == 1
            return signo * np.sum(POWERS[index]) / DIVISION

    @staticmethod
    def crossover(c1, c2):
        mitad = int(BITS / 2)
        x1 = np.append(c1.x[0:mitad], c2.x[mitad:], axis=None)
        x2 = np.append(c2.x[0:mitad], c1.x[mitad:], axis=None)
        y1 = np.append(c1.y[0:mitad], c2.y[mitad:], axis=None)
        y2 = np.append(c2.y[0:mitad], c1.y[mitad:], axis=None)
        z1 = np.append(c1.y[0:mitad], c2.y[mitad:], axis=None)
        z2 = np.append(c2.y[0:mitad], c1.y[mitad:], axis=None)
        return [Cromosoma(x1, y1, z1), Cromosoma(x2, y2, z2)]

    @staticmethod
    def mutar©:
        index = np.random.randint(low=1, high=len(c.x))
        c.x[index] = 1 - c.x[index]
        index = np.random.randint(low=1, high=len(c.y))
        c.y[index] = 1 - c.y[index]
        index = np.random.randint(low=1, high=len(c.z))
        c.z[index] = 1 - c.z[index]
        return c
```

Code 7. Chromosome class definition

Selection pressure

```
K_POBLACION = 10
K_BASE = 2

def presion_selectiva(poblacion: list[Cromosoma]) -> list[Cromosoma]:
    # Evaluación y búsqueda del mejor
    x = np.array([c.to_int(0) for c in poblacion])
    y = np.array([c.to_int(1) for c in poblacion])
    z = np.array([c.to_int(2) for c in poblacion])
    evaluacion = rastrigin(x, y, z)

    best = evaluacion.argmin()
    print("Best so far:")
    print("Value: " + str(poblacion[best]))
    print(f"Integer value: {x[best]}, {y[best]}, {z[best]}")
    print("Rastrigin function value: " + str(evaluacion[best]))
    # Cálculo de probabilidades
    indice_ordenado = evaluacion.argsort()
    ruleta = []
    potencia = K_POBLACION

    for i in indice_ordenado:
        probabilidad = K_BASE ** potencia
        ruleta.extend([i] * probabilidad)
        potencia -= 1

    # Nueva generación
    nueva = list[Cromosoma]()
    nueva.append(poblacion[indice_ordenado[0]])
    nueva.append(poblacion[indice_ordenado[1]])

    for i in range(2, int(K_POBLACION/2)):
        c1 = poblacion[np.random.choice(ruleta)]
        c2 = poblacion[np.random.choice(ruleta)]
        hijos = Cromosoma.crossover(c1, c2)

        hijos[0] = Cromosoma.mutar(hijos[0])
        hijos[1] = Cromosoma.mutar(hijos[1])

        nueva.extend(hijos)

    return nueva
```

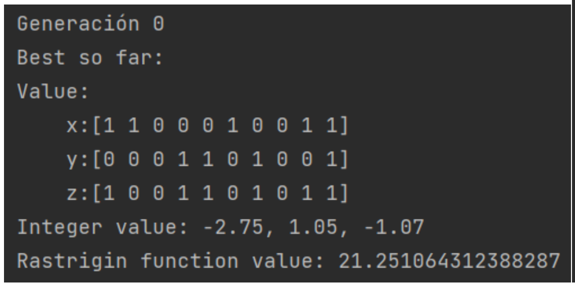
Code 8. Selection pressure definition

Settings for running

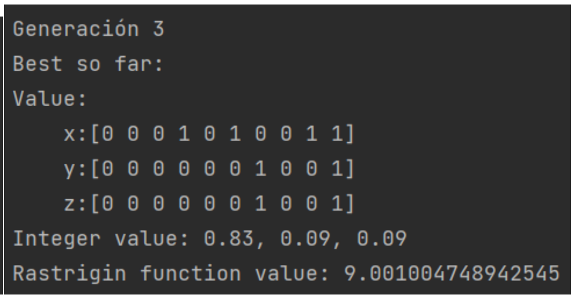
```
 eneració = list[Cromosoma]()
nueva_poblacion = list[Cromosoma]()
 eneración = 0
```

```
if len(nueva_poblacion) == 0:
     eneració = [Cromosoma() for _ in range(0, K_POBLACION)]
else:
     eneració = nueva_poblacion

print('Generación',  eneración)
nueva_poblacion =  enerac_selectiva( eneració)
 eneración += 1
```

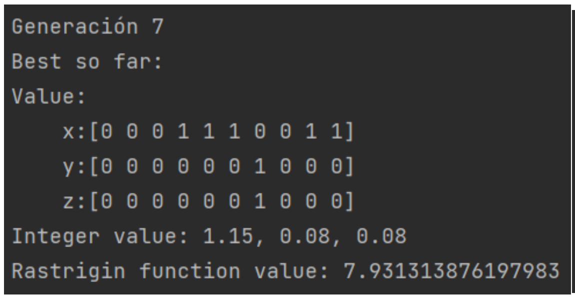
Code 9 and 10, Setting up and running codes



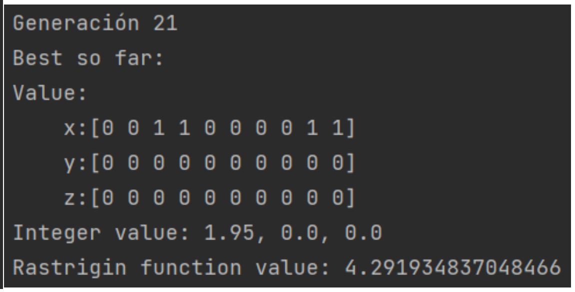
```
Generación 0
Best so far:
Value:
    x:[1 1 0 0 0 1 0 0 1 1]
    y:[0 0 0 1 1 0 1 0 0 1]
    z:[1 0 0 1 1 0 1 0 1 1]
Integer value: -2.75, 1.05, -1.07
Rastrigin function value: 21.25106431238287
```



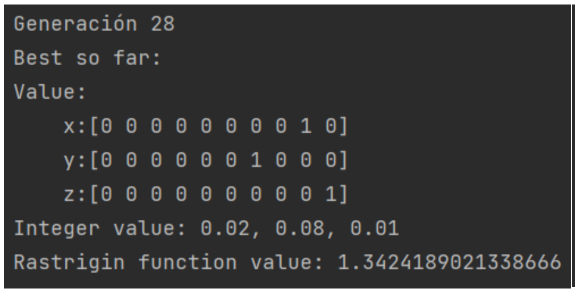
```
Generación 3
Best so far:
Value:
    x:[0 0 0 1 0 1 0 0 1 1]
    y:[0 0 0 0 0 0 1 0 0 1]
    z:[0 0 0 0 0 0 1 0 0 1]
Integer value: 0.83, 0.09, 0.09
Rastrigin function value: 9.001004748942545
```



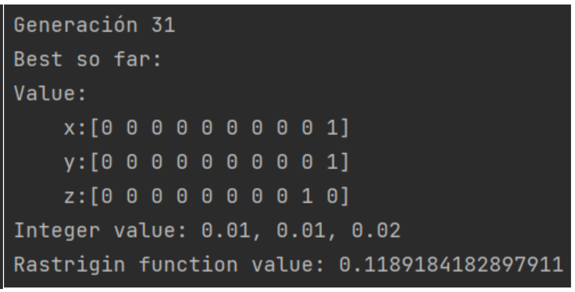
```
Generación 7
Best so far:
Value:
    x:[0 0 0 1 1 1 0 0 1 1]
    y:[0 0 0 0 0 0 1 0 0 0]
    z:[0 0 0 0 0 0 1 0 0 0]
Integer value: 1.15, 0.08, 0.08
Rastrigin function value: 7.931313876197983
```



```
Generación 21
Best so far:
Value:
    x:[0 0 1 1 0 0 0 0 1 1]
    y:[0 0 0 0 0 0 0 0 0 0]
    z:[0 0 0 0 0 0 0 0 0 0]
Integer value: 1.95, 0.0, 0.0
Rastrigin function value: 4.291934837048466
```



```
Generación 28
Best so far:
Value:
    x:[0 0 0 0 0 0 0 0 1 0]
    y:[0 0 0 0 0 0 1 0 0 0]
    z:[0 0 0 0 0 0 0 0 0 1]
Integer value: 0.02, 0.08, 0.01
Rastrigin function value: 1.3424189021338666
```



```
Generación 31
Best so far:
Value:
    x:[0 0 0 0 0 0 0 0 0 1]
    y:[0 0 0 0 0 0 0 0 0 1]
    z:[0 0 0 0 0 0 0 0 1 0]
Integer value: 0.01, 0.01, 0.02
Rastrigin function value: 0.1189184182897911
```

Fig. 8, 9, 10, 11, 12 and 13. Results from Rastrigin optimization

## Conclusions and recommendations

In theory, I saw these concepts easy, and, they are, but in practice is not just that, there are lots of obstacles. As I said in earlier practices, this kind of programming has not been my field, I really like it, but it seems hard, a lot. Rastrigin function feels harder than Ackley, I really enjoy seeing how I overcome a lot of things that I didn't even know were a problem. I feel pretty satisfied with this work, I means a lot, and I know I have to become better and better. To be honest, looking at these practices make me feel that I should look for what I have always seen as impossible.

Now, talking about the results, maybe with better operators at Rastrigin problem we can find it so much faster, but, Rastrigin function is considerably harder and that is actually something good, optimization test should be straight forward with it.

## References

[1] S. Surjanovic, D. Bingham. *ACKLEY FUNCTION*. 2013. Virtual Library of Simulation Experiments. Accessed on: Sep. 29, 2021. [Online] Available: https://www.sfu.ca/~ssurjano/ackley.html

[2] Wikipedia. *Rastrigin function*. Apr. 20, 2021. Wikipedia. Accessed on: Sep. 29, 2021. [Online] Available: https://en.wikipedia.org/wiki/Rastrigin_function

[3] Wikipedia. *Genetic algorithm*. Sep. 19, 2021. Wikipedia. Accessed on: Oct. 3, 2021. [Online] Available: https://en.wikipedia.org/wiki/Genetic_algorithm

[4] J. Trigueros, Class lecture, Topic: "Genetic Algorithms" Escuela Superior de Cómputo, Instituto Politécnico Nacional. Mexico City, Sep. 20, 2021.