Cap 3. Ejercicios

1. Maximizar la función $f(x)=x\ sen(10\ \pi$ x) + 1, con $x \in$[0,1].

Para maximizar la función que se nos da, usando algoritmo genético, vamos a considerar individuos representados por cadenas binarias que nos determinarán el valor de x que representan. Todo se realizará en el archivo “Ejercicio_1.py”.

Primero se determinó una población de 50 individuos, durante 100 generaciones, una longitud de cromosoma de 16, y una tasa de mutación de cromosomas del 0.01.

Tras esto se usa como función de aptitud el resultado de evaluar f(x), teniendo en cuenta que entre más alto el valor, mejor, pues estamos maximizando la funcion.

Ya con esto se creo el algoritmo genetico para el ejercicio, y se dejo correr hasta llegar al siguiente resultado:

```
PS C:\Trabajo\Cristian\Universidad\2025-1\Inteligencia-Artificial-Y-Mini-Robots\Cap3> python "Ejercicio_1.py"
Mejor x: 0.851194
f(x): 1.850595
```

Así podemos llegar a que el valor máximo de f(x), con x entre 0 y 1, es cuando x toma un valor cercano a 0.851194, dando f(x) = 1.850595 como el valor máximo que puede alcanzar.

2. Verdadera democracia. Suponga que usted es el jefe de gobierno y está interesado en que pasen los proyectos de su programa político. Sin embargo, en el congreso conformado por 5 partidos, no es fácil su tránsito, por lo que debe repartir el poder, conformado por ministerios y otras agencias del gobierno, con base en la representación de cada partido. Cada entidad estatal tiene un peso de poder, que es el que se debe distribuir. Suponga que hay 50 curules, distribuya aleatoriamente, con una distribución no informe entre los 5 partidos esas curules. Defina una lista de 50 entidades y asígneles aleatoriamente un peso político de 1 a 100 puntos. Cree una matriz de poder para repartir ese poder, usando AGs.

Para este ejercicio, vamos a tener 5 partidos políticos, A, B, C, D y E, entre los cuales se van a repartir 50 curules de forma aleatoria y no uniforme:

Partido	Número de curules
A	10
B	5
C	16
D	12
E	7

A partir de esto, se crearán aleatoriamente los pesos de las 50 entidades, identificadas con índices de 0 al 49, las cuales se van a repartir con respecto a las curules de los partidos, y se determinara que tan bien esta repartido el poder politico haciendo uso del error cuadratico medio como nuestra funcion de aptitud:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

El algoritmo genético lo desarrolle en python, usando una población de 8 individuos, cada uno conteniendo 5 listas con el ordenamiento de los individuos entre los distintos partidos políticos. Además, se indicó una probabilidad de mutación del 0.2, 100 generaciones antes de acabar, y una selección por torneo de 3 sobrevivientes. Todo el código en el archivo “Ejercicio_2.py”.

Al ejecutar una de las pruebas, obtenemos inicialmente los siguientes pesos para las 50 entidades que están siendo distribuidas:

Pesos de las entidades

36	45	59	65	62
57	90	44	18	99
73	22	22	35	100
36	83	65	80	42
95	11	93	29	94
2	92	57	38	19
33	76	65	57	99
82	14	60	98	1
85	57	88	2	15
4	8	16	85	92

Luego se empieza a evaluar y a pasar las generaciones de individuos, cada vez mejorando nuestra función objetivo:

Generación 1, mejor aptitud: 0.003462
Generación 2, mejor aptitud: 0.002735
Generación 3, mejor aptitud: 0.002722
Generación 4, mejor aptitud: 0.002264
Generación 5, mejor aptitud: 0.002264
Generación 6, mejor aptitud: 0.002264
Generación 7, mejor aptitud: 0.002264
Generación 8, mejor aptitud: 0.002264
Generación 9, mejor aptitud: 0.002264
Generación 10, mejor aptitud: 0.002264
Generación 11, mejor aptitud: 0.002264
Generación 12, mejor aptitud: 0.002264
Generación 13, mejor aptitud: 0.002170
Generación 14, mejor aptitud: 0.001589
Generación 15, mejor aptitud: 0.001472
Generación 16, mejor aptitud: 0.001285
Generación 17, mejor aptitud: 0.001108
Generación 18, mejor aptitud: 0.001108
Generación 19, mejor aptitud: 0.001101
Generación 20, mejor aptitud: 0.001098
Generación 21, mejor aptitud: 0.000986
Generación 22, mejor aptitud: 0.000463
Generación 23, mejor aptitud: 0.000463
Generación 24, mejor aptitud: 0.000463
Generación 25, mejor aptitud: 0.000460
Generación 26, mejor aptitud: 0.000239
Generación 27, mejor aptitud: 0.000213
…
Generación 95, mejor aptitud: 0.000213
Generación 96, mejor aptitud: 0.000176
Generación 97, mejor aptitud: 0.000157
Generación 98, mejor aptitud: 0.000157
Generación 99, mejor aptitud: 0.000120
Generación 100, mejor aptitud: 0.000118

Al final se logra encontrar un individuo que logra reordenar los puestos hasta llegar a una solución bastante cercana a cero, osea bastante buena al dejar baste equitativamente distribuido el poder entre los partidos políticos:

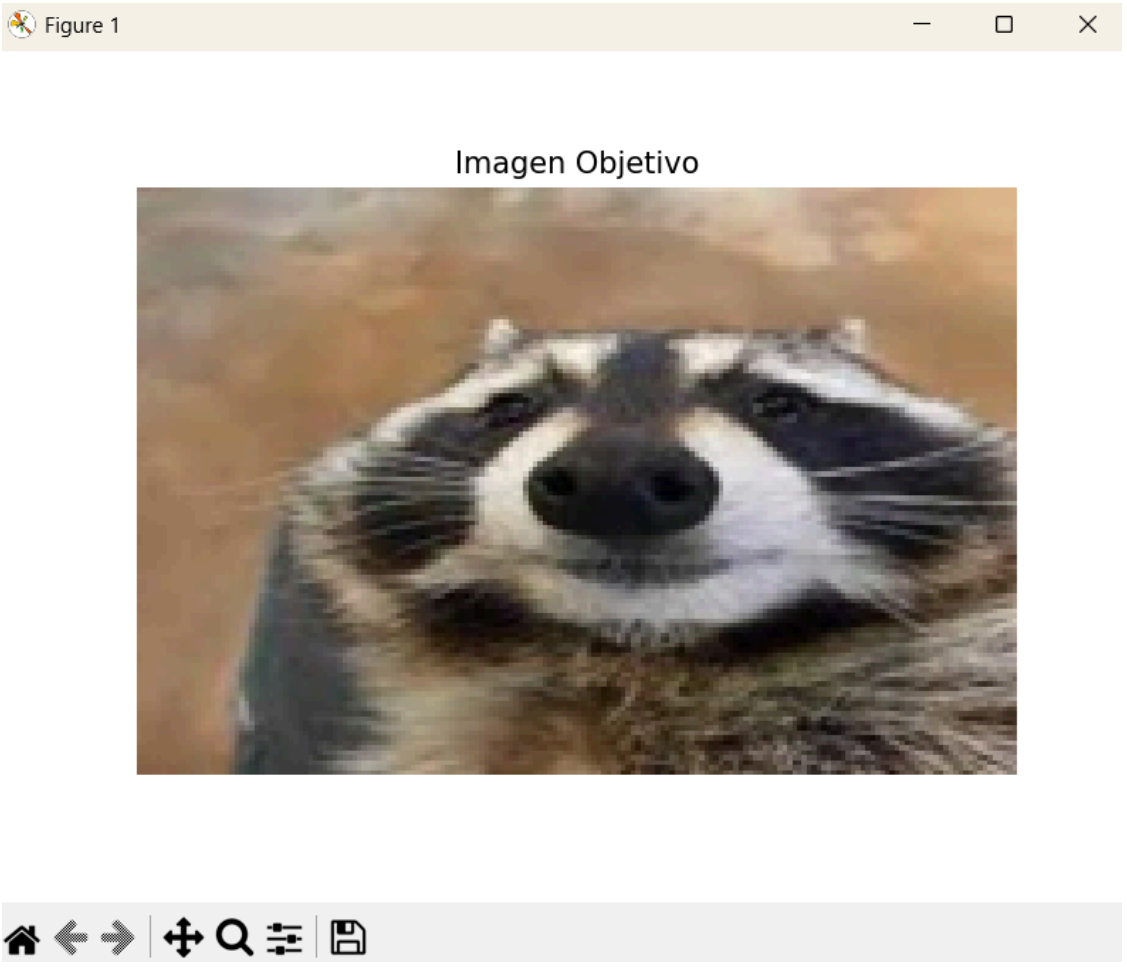
```
Mejor individuo:
A | 527 |  0.195185 | [13, 32, 33, 33, 19, 24, 24, 16] |
B | 482 |  0.178519 | [6, 14, 14, 14, 26] |
C | 530 |  0.196296 | [21, 29, 41, 42, 43, 44, 45, 46, 47, 48, 49, 28, 28, 5] |
D | 516 |  0.191111 | [7, 10, 15, 23, 39, 39, 31, 38, 9, 2] |
E | 530 |  0.196296 | [20, 18, 20, 20, 18, 40] |
```

Se evidencian problemas, pues duplica entidades en las curules, esto debido a que, en la creación de la nueva generación se combinan las listas de los individuos, llegando a generar duplicados en la misma, o en distintos partidos políticos (Se me dificulto la creación de una nueva generación, y no encontre mucha informacion sobre algoritmos genéticos para ordenamiento).

4. Genere aleatoriamente una población de 50 matrices de 120 por 180, con números de 0 a 255, preséntelas como una gráfica RGB. La función de aptitud es una imagen cualquiera. Evolucione la población inicial hasta llegar a la imagen.

El código para este ejercicio se encuentra en el archivo “Ejercicio_4.py”. Para resolverlo usamos de base la imagen de un mapache que encontramos por internet:

Al no tener demasiados colores, y usar contrastes entre negros y blancos, va a ser más facil identificar visualmente si el resultado se aproxima a la imagen original o no. Al cambiar el tamaño de la imagen, para que sea de 120x180, queda tal que así:



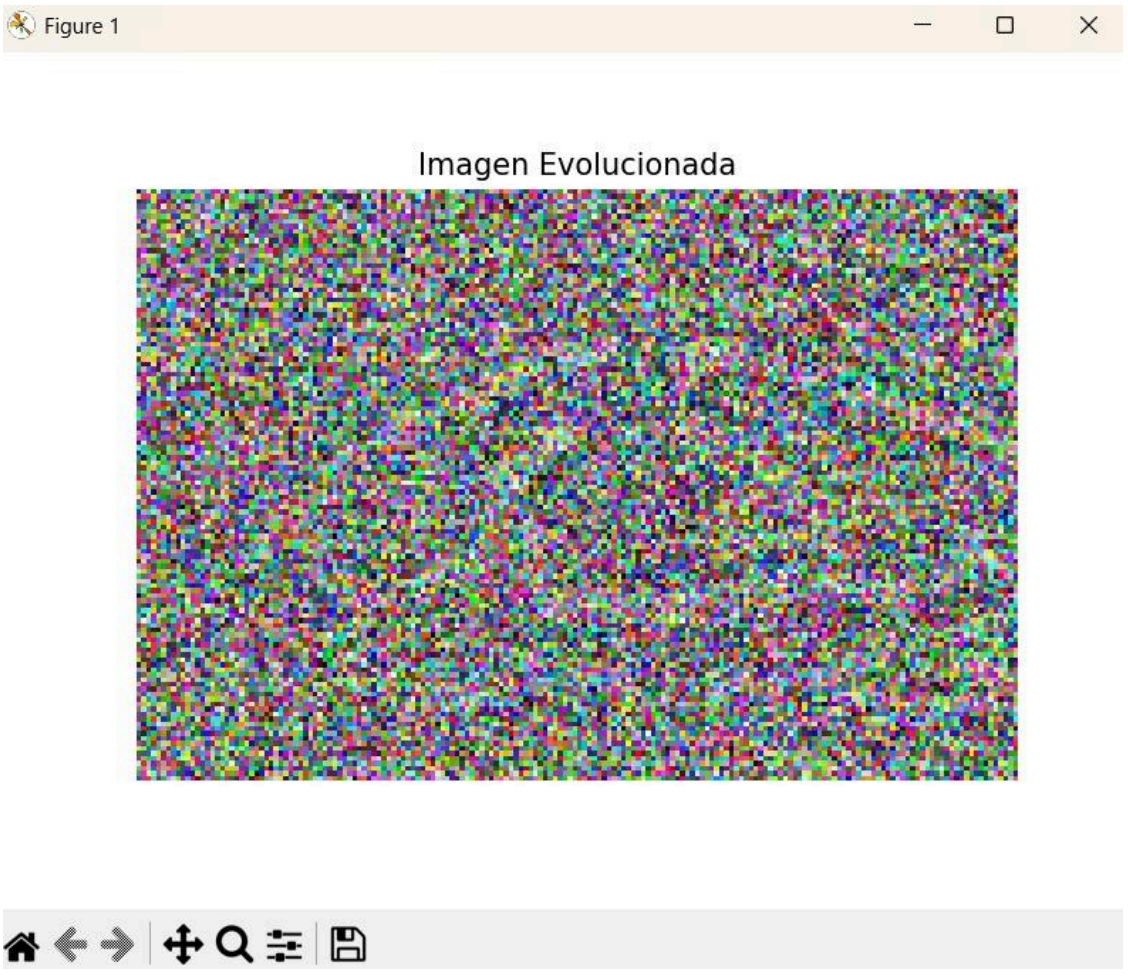
Ahora, para el algoritmo genético se va a usar una población de 50 matrices de 120x180 (así como lo indica el ejercicio), tomando como padres de la siguiente generación al mejor 30% de la población, osea 15 seleccionados por torneo. Tambien se usaran 500 generaciones, pero se detendra si tras 100 generaciones no hay mejoria.

Al ejecutarse el codigo se llega hasta las 766 generaciones (sinceramente, con mucha suerte de mutaciones favorables) y se logra una similitud del 14,25%. La ejecucion tardo cerca de 3 minutos nada más.

```
17  poblacion = 50
18  num_torneo = int(poblacion * 0.3)
19  generaciones = 1000
20
21  # Función de aptitud
22  def aptitud(img):
23      # Convertir ambas imágenes a escala de grises antes de calcular SSIM
24      imagen_gris = np.mean(img, axis=2)
25      objetivo_gris = np.mean(objetivo, axis=2)
26      return ssim(imagen_gris, objetivo_gris, data_range=255)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Generación 759, aptitud: 0.14250154550093397
Generación 760, aptitud: 0.14250154550093397
Generación 761, aptitud: 0.14250154550093397
Generación 762, aptitud: 0.14250154550093397
Generación 763, aptitud: 0.14250154550093397
Generación 764, aptitud: 0.14250154550093397
Generación 765, aptitud: 0.14250154550093397
Generación 766, aptitud: 0.14250154550093397
No hay mejoras significativas, terminando evolución.
```



Por curiosidad tambien realice una ejecucion mucho más exigente, pero que prometia una gran mejora en la calidad de la imagen resultante, pues implemente el codigo con una poblacion de 10.000, lo cual fue mucho más exigente computacionalmente, pero logro una similitud del 23,725% tras 626 generaciones. La ejecucion tardo cerca de 12 horas en completarse, pero dejo una imagen visualmente más semejante.

```python
poblacion = 10000
num_torneo = int(poblacion * 0.3)
generaciones = 1000

# Función de aptitud
def aptitud(img):
    # Convertir ambas imágenes a escala de grises antes de calcular SSIM
    imagen_gris = np.mean(img, axis=2)
    objetivo_gris = np.mean(objetivo, axis=2)
    return ssim(imagen_gris, objetivo_gris, data_range=255)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Generación 619, aptitud: 0.23724964697907022
Generación 620, aptitud: 0.23724964697907022
Generación 621, aptitud: 0.23724964697907022
Generación 622, aptitud: 0.23724964697907022
Generación 623, aptitud: 0.23724964697907022
Generación 624, aptitud: 0.23724964697907022
Generación 625, aptitud: 0.23724964697907022
Generación 626, aptitud: 0.23724964697907022
No hay mejoras significativas, terminando evolución.
```

Figure 1