Definire Functie Reala:

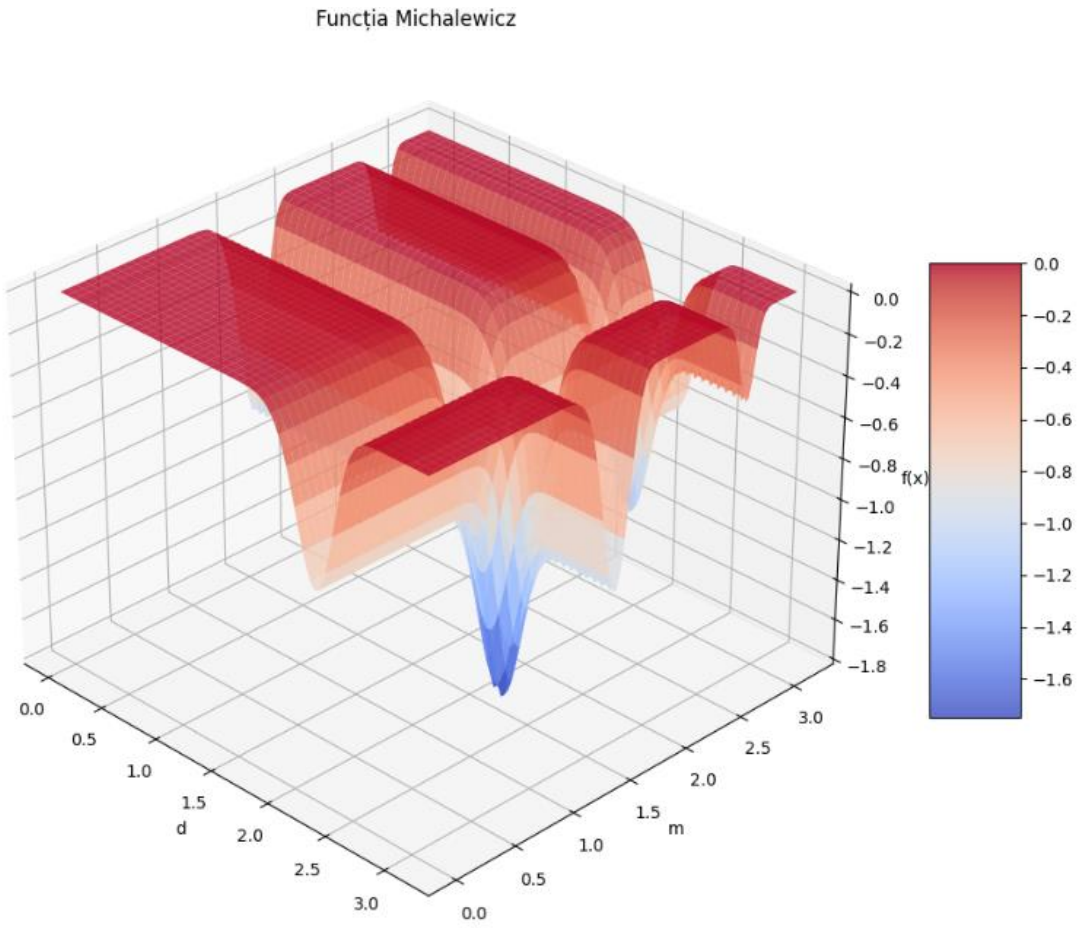
```python
#@title Functii Codificare Reala
def michalewicz(x, m=10):
    """
    Funcția Michalewicz
    INPUT
    -----
    x: vectorul de soluție
    m: parametru de dificultate (mai mare = mai multe local optima)
    OUTPUT
    -----
    valoarea funcției (de minimizat)
    """
    d = len(x)
    sum_term = 0.0
    for i in range(d):
        sum_term += np.sin(x[i]) * np.sin(((i + 1) * x[i]**2) / np.pi)**(2 * m)
    return -sum_term
```

Plot Functie Reala in 3D:

```
# Funcție pentru vizualizare 3D

def plot_michalewicz_3d():

   x = np.linspace(0, np.pi, 100)

   y = np.linspace(0, np.pi, 100)

   X, Y = np.meshgrid(x, y)

   Z = np.zeros_like(X)


   # Calcularea valorilor funcției

   for i in range(X.shape[0]):

      for j in range(X.shape[1]):

         Z[i, j] = michalewicz([X[i, j], Y[i, j]])


   # Plot 3D

   fig = plt.figure(figsize=(10, 8))

   ax = fig.add_subplot(111, projection='3d')
```

```
    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.8)

    fig.colorbar(surf, shrink=0.5, aspect=5)
    ax.set_xlabel('d')
    ax.set_ylabel('m')
    ax.set_zlabel('f(x)')
    ax.set_title('Funcția Michalewicz')

    # Ajustare unghi de vizualizare pentru a fi similar cu imaginea
    ax.view_init(elev=30, azim=-45)

    plt.tight_layout()
    plt.show()

    min_val = find_minimum(2)
    print(f"Valoarea minimă d=2, m=10: {min_val}")

    min_val_10 = find_minimum(10)
    print(f"Valoarea minimă d=10, m=10: {min_val_10}")

# Funcție pentru găsirea valorii minime aproximative
def find_minimum(dimension, num_trials=100000):
    best_val = 0
    for _ in range(num_trials):
        x = np.random.uniform(0, np.pi, size=dimension)
        val = michalewicz(x)
```

```
    if val < best_val:
        best_val = val
  return best_val

# Rulare vizualizare 3D
plot_michalewicz_3d()
```



Algoritm Evolutiv:

def init_population(pop_size, n_vars, lower_bound=0, upper_bound=3.14):

- Initializeaz o populatie de dimensiune pop_size cu valori din [0, pi]

def crossover(parents, crossover_rate=0.8):

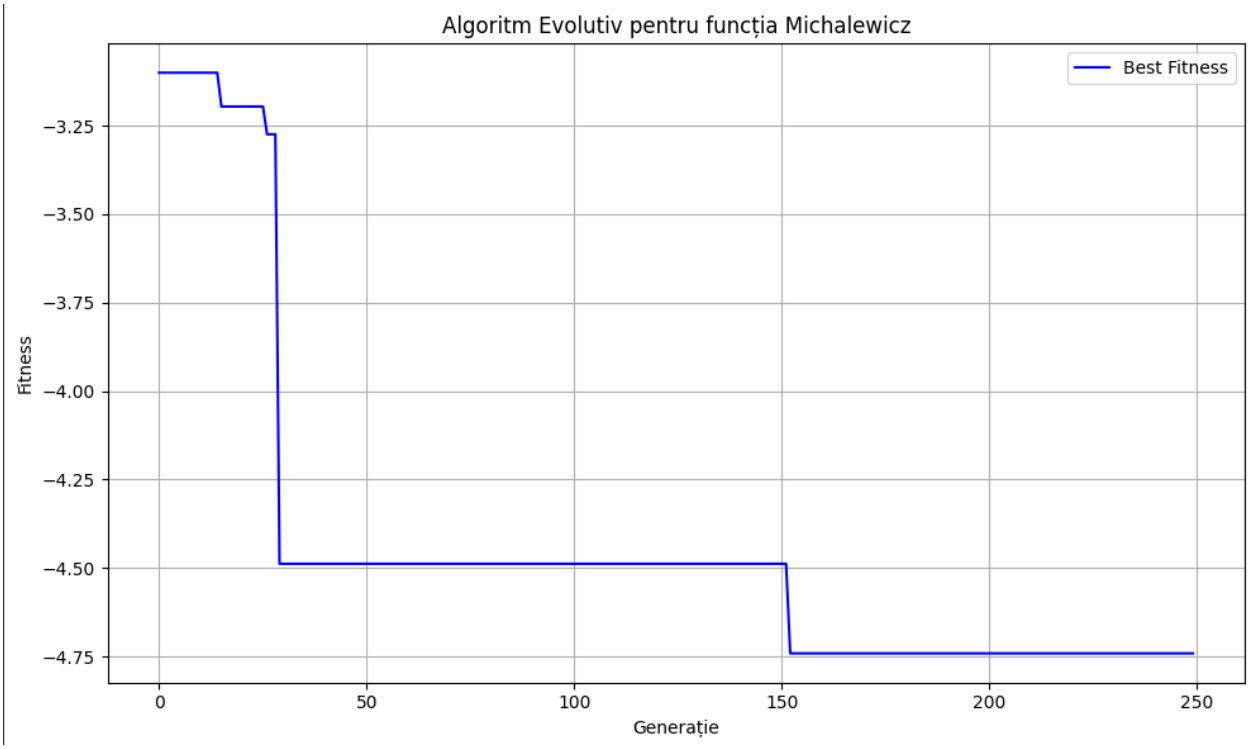
- Realizeaza incrucisarea pentru crearea noi descendenti noi

def mutation(population, mutation_rate=0.2):

- Efectueaza mutatia asupra descendentilor actuali cu un mutation_rate

def evolutionary_algorithm_michalewicz(pop_size, n_vars, generations, crossover_rate=0.8, mutation_rate=0.1, m=10):

- Ruleaza algoritmul evolutiv pentru minimizarea functiei reale Michalewicz primita ca tema prin initializarea unei populatii aleatoare pentru care se realizeaza incrucisare si mutatie, salvandu-se valoarea obtinuta pentru actuala populatie, urmand sa fie comparata cu valoarea obtinuta de catre generatia urmatoare.



def evolutionary_michalewicz_n_times(pop_size, n_vars, generations, crossover_rate, mutation_rate, m):

- Ruleaza algoritmul evolutiv definit inainte de n-ori pentru a forma un tabel din care sa observam evolutia algoritmul in timp

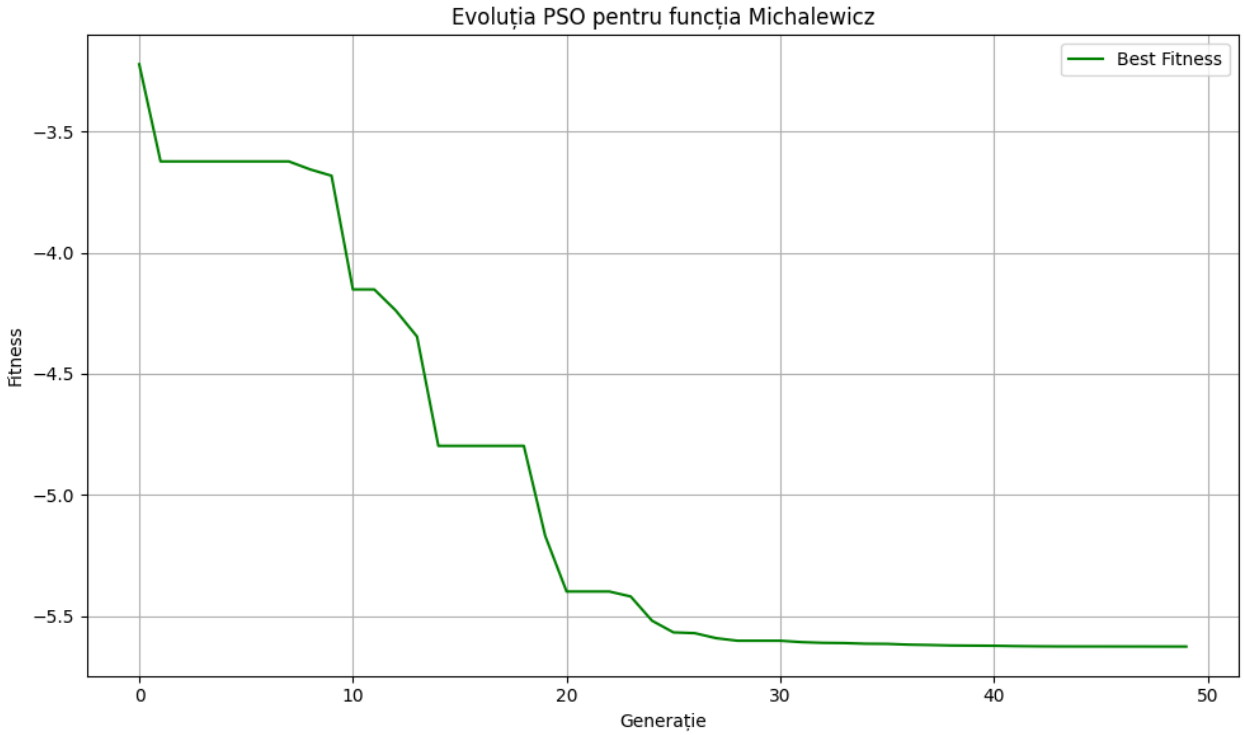
Particle swarm optimization

def init_particle(n_vars, left=0, right=1):

- Inițializează o particulă cu valori reale într-un interval specificat

def pso_michalewicz(n_vars, n, t, w, c1, c2):

- Algoritmul PSO pentru optimizarea funcției Michalewicz. Initializeaza mai multe particule impreuna cu viteza lor si cauta in mod iterativ un global_best prin aplicarea algoritmul PSO.



def pso_michalewicz_n_times(n_vars, n, t, w, c1, c2):

- Ruleaza algoritmul de n ori si returneaza cele mai bune solutii

Analiză Rezultate:

Algoritm Evolutiv:

n_vars	pop_size	generations	crossover_rate	mutation_rate	m	Best Solution	Avg Solution	Avg Time
10	100	250	0.2	0.1	10	-2.73874255959596	-3.39899973929153	0.07483012676239013
10	100	250	0.2	0.5	10	-2.7364894148930152	-3.2451106962545326	0.09120471477508545
10	100	250	0.5	0.1	10	-2.7272092275593156	-3.276505874483729	0.09157083034515381
10	100	250	0.5	0.5	10	-2.435380330728748	-3.252382011766455	0.09013099670410156
10	100	250	0.8	0.1	10	-2.773358436393313	-3.5436416022770474	0.07828643321990966
10	100	250	0.8	0.5	10	-2.5992316453140494	-3.1916512546202807	0.09372439384460449

PSO:

n_vars	t	n	w	c1	c2	Best Solution	Avg Solution	Avg Time
10	50	50	0.5	0.5	1.5	-2.2920557882670716	-3.0396025828226954	0.0038855540962219238
10	50	50	0.5	0.5	0.75	-2.0083398992548123	-3.103409911958668	0.0037275362014770 51
10	50	50	0.5	1.5	1.5	-2.1420583690651402	-3.063370288443216	0.0038487434387207 03
10	50	50	0.5	1.5	0.75	-2.2976162378875453	-2.976290253535632	0.00398439884185791
10	50	50	1	0.5	1.5	-2.288638563434163	-2.947174928019438	0.0036960697174072267
10	50	50	1	0.5	0.75	-2.2587167888033126	-3.052447450895688	0.0039602470397949 22
10	50	50	1	1.5	1.5	-2.223883884568799	-3.0425966589862337	0.0037477636337280274
10	50	50	1	1.5	0.75	-2.1888509457325536	-3.0810373258280865	0.004018139839172363

Prin compararea rezultatelor obținute în coloana **Best Solution** se poate observa faptul că algoritmul PSO a performat mai slab decat Algoritmul Evolutiv în a converge către o valoare apropiată de –4.5, diferența fiind totuși de ordinul zecimalelor. Din graficul algoritmul PSO se observă că pentru după un anumit număr de generații graficul descrește și acest fapt ar putea indica necesitatea algoritmul pentru un număr mai mare de generații decat 50 cate au fost alese, pentru a se realiza o mai buna comparatie cu Algoritmul Evolutiv din a carui grafic putem observa ca a ajuns la valoarea de –4.5 in mai putin de 50 de generatii.