In [11]:
```python
# Importar la división futura (para garantizar que la división de números enteros sea de punto flotante)
from __future__ import division
# Importar el módulo 'random' para generar números aleatorios
import random
# Importar el módulo 'math' para operaciones matemáticas
import math

# Definir la nueva función a optimizar
def func_to_optimize(x):
    return (1.5 - x[0] + x[0] * x[1])**2 + (2.25 - x[0] + x[0] * x[1]**2)**2 + (2.625 - x[0] + x[0] * x[1]**3)**2

# Definición de la clase Particle (Partícula)
class Particle:
    # Constructor de la clase
    def __init__(self, x0):
        self.position_i = []          # posición de la partícula
        self.velocity_i = []          # velocidad de la partícula
        self.pos_best_i = []          # mejor posición individual
        self.err_best_i = -1          # mejor error individual
        self.err_i = -1               # error individual

        # Inicializar la posición y velocidad de la partícula
        for i in range(0, num_dimensions):
            self.velocity_i.append(random.uniform(-1, 1))
            self.position_i.append(x0[i])

    # Evaluar la aptitud actual
    def evaluate(self, costFunc):
        self.err_i = costFunc(self.position_i)

        # Comprobar si la posición actual es la mejor individual
        if self.err_i < self.err_best_i or self.err_best_i == -1:
            self.pos_best_i = self.position_i
            self.err_best_i = self.err_i

    # Actualizar la velocidad de la partícula
    def update_velocity(self, pos_best_g):
        w = 0.5       # peso de inercia constante (cuánto pesar la velocidad anterior)
        c1 = 1        # constante cognitiva
        c2 = 2        # constante social

        for i in range(0, num_dimensions):
            r1 = random.random()
            r2 = random.random()

            vel_cognitive = c1 * r1 * (self.pos_best_i[i] - self.position_i[i])
            vel_social = c2 * r2 * (pos_best_g[i] - self.position_i[i])
            self.velocity_i[i] = w * self.velocity_i[i] + vel_cognitive + vel_social

    # Actualizar la posición de la partícula en función de las nuevas actualizaciones de velocidad
    def update_position(self, bounds):
        for i in range(0, num_dimensions):
            self.position_i[i] = self.position_i[i] + self.velocity_i[i]

            # Ajustar la posición máxima si es necesario
            if self.position_i[i] > bounds[i][1]:
                self.position_i[i] = bounds[i][1]

            # Ajustar la posición mínima si es necesario
            if self.position_i[i] < bounds[i][0]:
                self.position_i[i] = bounds[i][0]

# Definición de la clase PSO (Optimización por Enjambre de Partículas)
class PSO:
    def __init__(self, costFunc, x0, bounds, num_particles, maxiter):
        global num_dimensions

        num_dimensions = len(x0)
        err_best_g = -1                   # mejor error para el grupo
        pos_best_g = []                   # mejor posición para el grupo

        # Establecer el enjambre (swarm)
        swarm = []
        for i in range(0, num_particles):
            swarm.append(Particle(x0))

        # Comenzar el bucle de optimización
        i = 0
        while i < maxiter:
            # Recorrer las partículas en el enjambre y evaluar la aptitud
            for j in range(0, num_particles):
                swarm[j].evaluate(costFunc)

                # Determinar si la partícula actual es la mejor (globalmente)
                if swarm[j].err_i < err_best_g or err_best_g == -1:
                    pos_best_g = list(swarm[j].position_i)
                    err_best_g = float(swarm[j].err_i)

            # Recorrer el enjambre y actualizar las velocidades y posiciones
            for j in range(0, num_particles):
                swarm[j].update_position(bounds)
                swarm[j].update_velocity(pos_best_g)
            i += 1

        # Imprimir los resultados finales
        print('FINAL:')
        print('Mejor Posición:', pos_best_g)
        print('Mejor Error:', err_best_g)

# Definir la ubicación inicial y los límites iniciales
initial = [random.uniform(-4.5, 4.5), random.uniform(-4.5, 4.5)]  # Ubicación inicial dentro del rango especificado
bounds = [(-4.5, 4.5), (-4.5, 4.5)]  # Límites de entrada dentro del rango especificado

# Utilizar la nueva función 'func_to_optimize' con la ubicación inicial y límites modificados
PSO(func_to_optimize, initial, bounds, num_particles=100, maxiter=100)
```

```
FINAL:
Mejor Posición: [3.0000000000019655, 0.5000000000002108]
Mejor Error: 2.382147939725129e-24
```

Out[11]: <__main__.PSO at 0x10b27b750>

In [ ]:

In [ ]: