import numpy as np

class Particle:

    def \_\_init\_\_(self, dim, bounds):

        self.position = np.random.uniform(bounds[0], bounds[1], dim)  # Random initial position

        self.velocity = np.random.uniform(-1, 1, dim)  # Random initial velocity

        self.best\_position = np.copy(self.position)  # Personal best position

        self.best\_value = np.inf  # Personal best value

class PSO:

    def \_\_init\_\_(self, func, dim, bounds, num\_particles=30, max\_iter=100, w=0.5, c1=1.5, c2=1.5):

        self.func = func  # Objective function to optimize

        self.dim = dim  # Dimensionality of the problem

        self.bounds = bounds  # Search space bounds (tuple: (min, max))

        self.num\_particles = num\_particles  # Number of particles in the swarm

        self.max\_iter = max\_iter  # Maximum number of iterations

        self.w = w  # Inertia weight

        self.c1 = c1  # Cognitive (personal) weight

        self.c2 = c2  # Social (global) weight

        # Initialize particles

        self.particles = [Particle(dim, bounds) for \_ in range(num\_particles)]

        self.global\_best\_position = np.random.uniform(bounds[0], bounds[1], dim)

        self.global\_best\_value = np.inf

    def update\_velocity(self, particle):

        r1 = np.random.random(self.dim)

        r2 = np.random.random(self.dim)

        cognitive\_term = self.c1 \* r1 \* (particle.best\_position - particle.position)

        social\_term = self.c2 \* r2 \* (self.global\_best\_position - particle.position)

        inertia\_term = self.w \* particle.velocity

        new\_velocity = inertia\_term + cognitive\_term + social\_term

        return new\_velocity

    def update\_position(self, particle):

        particle.position += particle.velocity

        # Clip the position to ensure it stays within bounds

        particle.position = np.clip(particle.position, self.bounds[0], self.bounds[1])

    def optimize(self):

        for i in range(self.max\_iter):

            for particle in self.particles:

                # Evaluate the objective function for the current position

                value = self.func(particle.position)

                # Update personal best

                if value < particle.best\_value:

                    particle.best\_value = value

                    particle.best\_position = np.copy(particle.position)

                # Update global best

                if value < self.global\_best\_value:

                    self.global\_best\_value = value

                    self.global\_best\_position = np.copy(particle.position)

            # Update velocities and positions of particles

            for particle in self.particles:

                particle.velocity = self.update\_velocity(particle)

                self.update\_position(particle)

            # Print the best value found so far

            print(f"Iteration {i+1}/{self.max\_iter} | Best Value: {self.global\_best\_value}")

        return self.global\_best\_position, self.global\_best\_value

# Example usage

def objective\_function(x):

    # Example: Sphere function (minimize)

    return np.sum(x\*\*2)

# Define bounds for the search space

bounds = (-5, 5)

dim = 10  # Dimensionality of the problem

# Initialize and run the PSO algorithm

pso = PSO(func=objective\_function, dim=dim, bounds=bounds, num\_particles=30, max\_iter=100)

best\_position, best\_value = pso.optimize()

print("Best Position:", best\_position)

print("Best Value:", best\_value)

OUTPUT:





