Understanding Particle Swarm Optimization (PSO): Optimizing complex functions can be a daunting task, but there's an algorithm that can make the process easier - Particle Swarm Optimization (PSO). Drawing inspiration from the collective intelligence of birds and fish, PSO is a powerful meta-heuristic algorithm that has become a cornerstone strategy for tackling optimization problems.

```
import numpy as np

class Particle:
    def __init__(self, dimension):
        # Initialize particle position and velocity randomly
        self.position = np.random.rand(dimension) * 10 - 5  # Random position in range [-5,
        self.velocity = np.random.rand(dimension) * 2 - 1  # Random velocity in range [-1, 1
        self.best_position = np.copy(self.position)  # Best position found by this particle
        self.best_value = float('inf')  # Best value (fitness) found by this particle

def sphere_function(x):
    """Sphere function to optimize: f(x) = sum(x_i^2)"""
    return np.sum(x**2)

def update_velocity(particle, global_best_position, inertia_weight, cognitive_coeff, social_
    """
    Update the particle's velocity based on:
    - Inertia from the previous velocity
    - Cognitive component (particle's best position)
    - Social component (global best position)
    """
    r1 = np.random.rand(len(particle.position))  # Random number for cognitive component
    r2 = np.random.rand(len(particle.position))  # Random number for social component
    cognitive_velocity = cognitive_coeff * r1 * (particle.best_position - particle.position)
    social_velocity = social_coeff * r2 * (global_best_position - particle.position)

    # Update the velocity
    particle.velocity = inertia_weight * particle.velocity + cognitive_velocity + social_vel

def update_position(particle):
    """Update the particle's position based on its velocity."""
    particle.position += particle.velocity
    # Limit the position to the range [-5, 5]
    particle.position = np.clip(particle.position, -5, 5)

def pso(num_particles, dimensions, max_iterations):
    """Main PSO algorithm implementation."""
    inertia_weight = 0.5  # Weight for inertia
    cognitive_coeff = 1.5  # Cognitive coefficient
    social_coeff = 1.5  # Social coefficient
```

```
    # Initialize particles
    particles = [Particle(dimensions) for _ in range(num_particles)]
    global_best_position = None  # Best position found by the swarm
    global_best_value = float('inf')  # Best value (fitness) found by the swarm

    # Main iteration loop
    for iteration in range(max_iterations):
        for particle in particles:
            # Evaluate fitness of the particle
            fitness_value = sphere_function(particle.position)
            # Update particle's best known position and value
            if fitness_value < particle.best_value:
                particle.best_value = fitness_value
                particle.best_position = np.copy(particle.position)

            # Update the global best position and value
            if fitness_value < global_best_value:
                global_best_value = fitness_value
                global_best_position = np.copy(particle.position)

        # Update velocity and position for each particle
        for particle in particles:
            update_velocity(particle, global_best_position, inertia_weight, cognitive_coeff,
            update_position(particle)

    return global_best_position, global_best_value


num_particles = int(input("Enter the number of particles: "))  # Number of particles in the
dimensions = int(input("Enter the number of dimensions: "))  # Number of dimensions for the
max_iterations = int(input("Enter the maximum number of iterations: "))  # Number of iterati

# Run the PSO algorithm
best_position, best_value = pso(num_particles, dimensions, max_iterations)

# Output the best position and corresponding value found
print("Best Position:", best_position)
print("Best Value (Objective Function Result):", best_value)
```

```
Enter the number of particles: 30
Enter the number of dimensions: 2
Enter the maximum number of iterations: 300
Best Position: [7.73671418e-35 3.94906792e-35]
Best Value (Objective Function Result): 7.545188377556087e-69
```

The **Ant Colony Optimization (ACO) algorithm** is a nature-inspired optimization technique based on the foraging behavior of ants. It was first introduced by Marco Dorigo in 1992 and is often used to solve combinatorial optimization problems, such as the **Traveling Salesman Problem (TSP)**, vehicle routing, network routing, and other similar problems where the goal is to find an optimal or near-optimal solution in a large search space. Basic Concept:

Ants in nature find the shortest path between their nest and a food source by leaving a chemical substance called pheromone on the ground as they travel. Other ants are attracted to areas with higher concentrations of pheromone, which reinforces successful paths. Over time, paths with stronger pheromone trails are more likely to be selected, leading to the optimal or near-optimal solution.

```
import numpy as np
import random
import matplotlib.pyplot as plt

# Define a class for the Ant Colony Optimization
class AntColony:
    def __init__(self, distance_matrix, num_ants, num_iterations, alpha=1, beta=2, rho=0.1,
        self.distance_matrix = distance_matrix
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha  # pheromone importance
        self.beta = beta    # heuristic importance (visibility)
        self.rho = rho      # pheromone evaporation rate
        self.Q = Q          # pheromone deposit constant
        self.num_cities = len(distance_matrix)

        # Initialize pheromone matrix
        self.pheromone = np.ones((self.num_cities, self.num_cities))  # uniform initial pher

        # Calculate visibility (1 / distance matrix)
        self.visibility = 1 / (distance_matrix + np.diag([np.inf] * self.num_cities))

    def run(self):
        best_route = None
        best_distance = float('inf')

        for iteration in range(self.num_iterations):
            all_routes = self.construct_all_routes()
            self.update_pheromone(all_routes)

            # Find the best route in this iteration
            for route, distance in all_routes:
                if distance < best_distance:
                    best_route = route
                    best_distance = distance

            print(f"Iteration {iteration+1}: Best Distance = {best_distance}")

        return best_route, best_distance
```

```
    def construct_all_routes(self):
        all_routes = []
        for _ in range(self.num_ants):
            route = self.construct_route()
            distance = self.calculate_route_distance(route)
            all_routes.append((route, distance))
        return all_routes

    def construct_route(self):
        route = []
        visited = [False] * self.num_cities
        current_city = random.randint(0, self.num_cities - 1)
        route.append(current_city)
        visited[current_city] = True

        while len(route) < self.num_cities:
            next_city = self.choose_next_city(route[-1], visited)
            route.append(next_city)
            visited[next_city] = True

        return route

    def choose_next_city(self, current_city, visited):
        probabilities = []
        for next_city in range(self.num_cities):
            if not visited[next_city]:
                pheromone = self.pheromone[current_city][next_city] ** self.alpha
                visibility = self.visibility[current_city][next_city] ** self.beta
                probabilities.append(pheromone * visibility)
            else:
                probabilities.append(0)

        # Normalize the probabilities
        total = sum(probabilities)
        probabilities = [prob / total for prob in probabilities]

        # Choose next city based on probability
        return random.choices(range(self.num_cities), probabilities)[0]

    def calculate_route_distance(self, route):
        distance = 0
        for i in range(len(route) - 1):
            distance += self.distance_matrix[route[i]][route[i + 1]]
        distance += self.distance_matrix[route[-1]][route[0]]  # Return to the starting city
        return distance

    def update_pheromone(self, all_routes):
        # Evaporate pheromone
        self.pheromone *= (1 - self.rho)

        # Deposit pheromone on the best routes
```

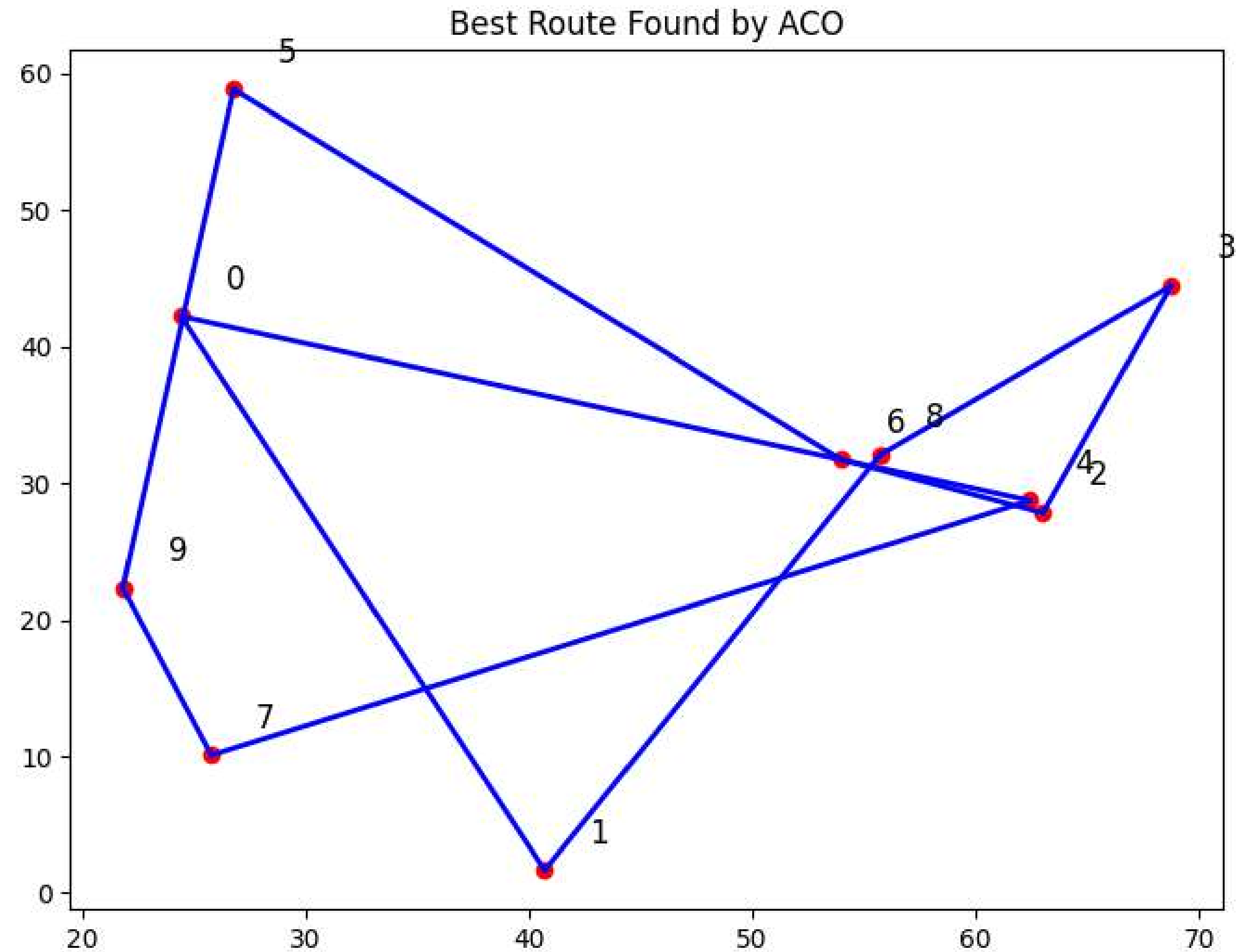
```
        for route, distance in all_routes:
            pheromone_deposit = self.Q / distance
            for i in range(len(route) - 1):
                self.pheromone[route[i]][route[i + 1]] += pheromone_deposit
            self.pheromone[route[-1]][route[0]] += pheromone_deposit  # Return to start

# Generate a random distance matrix for testing
def generate_random_distance_matrix(num_cities):
    matrix = np.random.randint(10, 100, size=(num_cities, num_cities))
    np.fill_diagonal(matrix, 0)  # Distance to itself is 0
    return matrix

# Main function to run the ACO
def main():
    # Generate a random distance matrix for 10 cities
    num_cities = 10
    distance_matrix = generate_random_distance_matrix(num_cities)

    # Display the distance matrix
    print("Distance Matrix:")
    print(distance_matrix)

    # Initialize the ACO algorithm
    num_ants = 50
    num_iterations = 100
    aco = AntColony(distance_matrix, num_ants, num_iterations, alpha=1, beta=2, rho=0.1, Q=1

    # Run the ACO algorithm
    best_route, best_distance = aco.run()

    print("\nBest Route:", best_route)
    print("Best Distance:", best_distance)

    # Visualize the best route
    plot_route(best_route, distance_matrix)

def plot_route(route, distance_matrix):
    # Plot the best route
    coordinates = np.random.rand(len(distance_matrix), 2) * 100  # Random 2D coordinates for
    x = coordinates[:, 0]
    y = coordinates[:, 1]

    plt.figure(figsize=(8, 6))
    plt.scatter(x, y, color='red')

    # Plot the route
    for i in range(len(route) - 1):
        plt.plot([x[route[i]], x[route[i + 1]]], [y[route[i]], y[route[i + 1]]], 'b-', lw=2)
    plt.plot([x[route[-1]], x[route[0]]], [y[route[-1]], y[route[0]]], 'b-', lw=2)  # Close

    for i, city in enumerate(route):
```

```
        plt.text(x[city] + 2, y[city] + 2, str(city), fontsize=12)

    plt.title("Best Route Found by ACO")
    plt.show()

if __name__ == "__main__":
    main()
```

```
Distance Matrix:
[[ 0 25 38 93 73 30 38 92 19 89]
 [58  0 66 50 50 16 77 22 16 63]
 [51 27  0 98 41 50 14 27 94 39]
 [94 76 11  0 76 35 92 23 40 31]
 [26 14 75 54  0 52 75 87 52 29]
 [84 45 84 85 13  0 74 35 33 16]
 [67 68 63 40 78 20  0 97 97 77]
 [86 54 51 74 24 24 87  0 73 22]
 [73 36 40 17 62 52 82 92  0 29]
 [41 79 92 68 18 58 48 17 35  0]]
Iteration 1: Best Distance = 193
Iteration 2: Best Distance = 193
Iteration 3: Best Distance = 186
Iteration 4: Best Distance = 186
Iteration 5: Best Distance = 186
Iteration 6: Best Distance = 186
Iteration 7: Best Distance = 186
Iteration 8: Best Distance = 186
Iteration 9: Best Distance = 186
Iteration 10: Best Distance = 186
Iteration 11: Best Distance = 186
Iteration 12: Best Distance = 186
Iteration 13: Best Distance = 186
Iteration 14: Best Distance = 186
Iteration 15: Best Distance = 186
Iteration 16: Best Distance = 186
Iteration 17: Best Distance = 186
Iteration 18: Best Distance = 186
Iteration 19: Best Distance = 186
Iteration 20: Best Distance = 186
Iteration 21: Best Distance = 186
Iteration 22: Best Distance = 186
Iteration 23: Best Distance = 186
Iteration 24: Best Distance = 186
Iteration 25: Best Distance = 186
Iteration 26: Best Distance = 186
Iteration 27: Best Distance = 186
Iteration 28: Best Distance = 186
Iteration 29: Best Distance = 186
Iteration 30: Best Distance = 186
Iteration 31: Best Distance = 186
Iteration 32: Best Distance = 186
Iteration 33: Best Distance = 186
Iteration 34: Best Distance = 186
Iteration 35: Best Distance = 186
Iteration 36: Best Distance = 186
Iteration 37: Best Distance = 186
Iteration 38: Best Distance = 186
Iteration 39: Best Distance = 186
Iteration 40: Best Distance = 186
Iteration 41: Best Distance = 186
Iteration 42: Best Distance = 186
Iteration 43: Best Distance = 186
Iteration 44: Best Distance = 186
Iteration 45: Best Distance = 186
```

```
Iteration 46: Best Distance = 186
Iteration 47: Best Distance = 186
Iteration 48: Best Distance = 186
Iteration 49: Best Distance = 186
Iteration 50: Best Distance = 186
Iteration 51: Best Distance = 186
Iteration 52: Best Distance = 186
Iteration 53: Best Distance = 186
Iteration 54: Best Distance = 186
Iteration 55: Best Distance = 186
Iteration 56: Best Distance = 186
Iteration 57: Best Distance = 186
Iteration 58: Best Distance = 186
Iteration 59: Best Distance = 186
Iteration 60: Best Distance = 186
Iteration 61: Best Distance = 186
Iteration 62: Best Distance = 186
Iteration 63: Best Distance = 186
Iteration 64: Best Distance = 186
Iteration 65: Best Distance = 186
Iteration 66: Best Distance = 186
Iteration 67: Best Distance = 186
Iteration 68: Best Distance = 186
Iteration 69: Best Distance = 186
Iteration 70: Best Distance = 186
Iteration 71: Best Distance = 186
Iteration 72: Best Distance = 186
Iteration 73: Best Distance = 186
Iteration 74: Best Distance = 186
Iteration 75: Best Distance = 186
Iteration 76: Best Distance = 186
Iteration 77: Best Distance = 186
Iteration 78: Best Distance = 186
Iteration 79: Best Distance = 186
Iteration 80: Best Distance = 186
Iteration 81: Best Distance = 186
Iteration 82: Best Distance = 186
Iteration 83: Best Distance = 186
Iteration 84: Best Distance = 186
Iteration 85: Best Distance = 186
Iteration 86: Best Distance = 186
Iteration 87: Best Distance = 186
Iteration 88: Best Distance = 186
Iteration 89: Best Distance = 186
Iteration 90: Best Distance = 186
Iteration 91: Best Distance = 186
Iteration 92: Best Distance = 186
Iteration 93: Best Distance = 186
Iteration 94: Best Distance = 186
Iteration 95: Best Distance = 186
Iteration 96: Best Distance = 186
Iteration 97: Best Distance = 186
Iteration 98: Best Distance = 186
Iteration 99: Best Distance = 186
Iteration 100: Best Distance = 186

Best Route: [7 4 0 1 8 3 2 6 5 9]
```

```
Best Route: [7, 4, 0, 1, 6, 3, 2, 8, 5, 9]
Best Distance: 186
```



The **Cuckoo Search (CS) algorithm**, inspired by the brood parasitism of cuckoos, is a powerful optimization method particularly suited for solving continuous optimization problems. It leverages the natural behavior of cuckoos laying eggs in the nests of other birds, leading to competition and adaptation that drives the search for optimal solutions.

```
import numpy as np
from scipy.special import gamma  # Import gamma function

# Objective Function (Rastrigin Function)
def rastrigin(x):
    A = 10
    return A * len(x) + np.sum(x**2 - A * np.cos(2 * np.pi * x))

# Lévy flight step size
def levy_flight(beta=1.5, dim=2):
    # Calculate the step size based on the Lévy distribution
    sigma = (gamma(1 + beta) * np.sin(np.pi * beta / 2) / gamma((1 + beta) / 2) * np.cos(np.
    u = np.random.normal(0, sigma, dim)
    v = np.random.normal(0, 1, dim)
    step = u / np.abs(v)**(1 / beta)
    return step

# Initialize the Cuckoo Search algorithm
def cuckoo_search(num_nests, num_iterations, dim, pa=0.25, alpha=0.01, beta=1.5):
    # Initialize the population (nests) with random solutions in the search space
    nests = np.random.uniform(-5.12, 5.12, (num_nests, dim))
    fitness = np.apply_along_axis(rastrigin, 1, nests)

    # Best solution
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    # Iterate for the number of iterations
    for iteration in range(num_iterations):
        # Generate new solutions using Lévy flights
        for i in range(num_nests):
            step = alpha * levy_flight(beta, dim)
            new_nest = nests[i] + step
            # Apply boundary check (constrain within search space)
            new_nest = np.clip(new_nest, -5.12, 5.12)

            # Evaluate fitness of the new nest
            new_fitness = rastrigin(new_nest)

            # If the new solution is better, replace the current nest
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness
```

```
        # Abandon the worst nests (with probability pa)
        worst_nests_idx = np.argsort(fitness)[:int(pa * num_nests)]
        for idx in worst_nests_idx:
            nests[idx] = np.random.uniform(-5.12, 5.12, dim)
            fitness[idx] = rastrigin(nests[idx])

        # Update the best solution
        current_best_nest = nests[np.argmin(fitness)]
        current_best_fitness = np.min(fitness)

        if current_best_fitness < best_fitness:
            best_nest = current_best_nest
            best_fitness = current_best_fitness

        # Print current iteration info
        print(f"Iteration {iteration + 1}/{num_iterations}, Best Fitness: {best_fitness}")

    return best_nest, best_fitness

# Parameters for Cuckoo Search
num_nests = 25          # Number of nests (solutions)
num_iterations = 100    # Number of iterations
dim = 10                # Number of dimensions (variables)
pa = 0.25               # Discovery probability (abandonment rate)
alpha = 0.01            # Step size scaling factor
beta = 1.5              # Lévy flight exponent

# Run Cuckoo Search
best_solution, best_fitness = cuckoo_search(num_nests, num_iterations, dim, pa, alpha, beta)

# Output the best solution
print(f"Best solution found: {best_solution}")
print(f"Best fitness (Rastrigin value): {best_fitness}")
```

```
Iteration 46/100, Best Fitness: 86.28732987013231
Iteration 47/100, Best Fitness: 86.28732987013231
Iteration 48/100, Best Fitness: 86.28732987013231
Iteration 49/100, Best Fitness: 86.28732987013231
Iteration 50/100, Best Fitness: 86.28732987013231
Iteration 51/100, Best Fitness: 86.28732987013231
Iteration 52/100, Best Fitness: 86.28732987013231
Iteration 53/100, Best Fitness: 86.28732987013231
```