Particle Swarm Code

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

# Set seeds for reproducibility
random.seed(42)
np.random.seed(42)

# --- 1. Objective Function ---

def sphere_function(position):
    """
    The classic Sphere function (f(x) = sum(x^2)), used for minimization.
    The global minimum is f(x)=0 at x=[0, 0, ..., 0].
    """
    # Ensures the input is treated as a NumPy array for vectorized operation
    return np.sum(position**2)

# --- 2. PSO Algorithm Implementation ---

def pso_optimizer(
    objective_func,
    num_particles=30,
    dimensions=2,
    search_range=(-10, 10),
    max_iterations=100,
    w=0.729,   # Inertia Weight (W)
    c1=1.4944, # Cognitive Constant (C1)
    c2=1.4944  # Social Constant (C2)
):
    """
    Particle Swarm Optimization (PSO) algorithm for continuous optimization.

    Args:
        objective_func (callable): The function to minimize.
        num_particles (int): Number of particles (S).
        dimensions (int): Dimensionality of the search space.
        search_range (tuple): (min, max) bounds for particle positions.
```

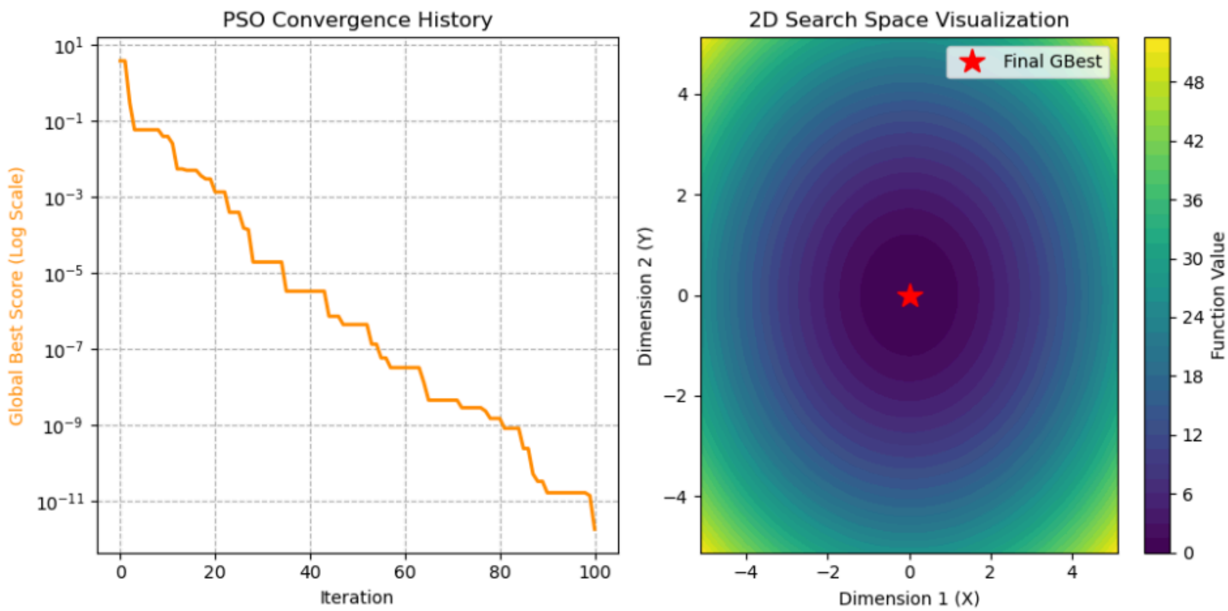
```python
    max_iterations (int): Maximum number of generations.
    w (float): Inertia weight.
    c1 (float): Cognitive constant.
    c2 (float): Social constant.
"""
min_bound, max_bound = search_range

# 1. Initialize particle positions and velocities (x_i and v_i)
# Positions: [num_particles, dimensions]
positions = np.random.uniform(min_bound, max_bound, (num_particles, dimensions))
# Velocities: [num_particles, dimensions]
velocities = np.random.uniform(-1, 1, (num_particles, dimensions))

# 2. Initialize Personal Best (PBest_i)
pbest_positions = positions.copy()
pbest_scores = np.array([objective_func(p) for p in positions])

# 3. Initialize Global Best (GBest)
gbest_index = np.argmin(pbest_scores)
gbest_position = pbest_positions[gbest_index].copy()
gbest_score = pbest_scores[gbest_index]

history = [(gbest_score, gbest_position)]

print(f"Starting PSO for {dimensions} dimensions with {num_particles} particles...")
print(f"Initial GBest Score: {gbest_score:.4f}")

for iteration in range(max_iterations):
    # --- Phase 1: Update PBest Positions ---
    for i in range(num_particles):
        current_score = objective_func(positions[i])

        # Check if current position is better than particle's personal best
        if current_score < pbest_scores[i]:
            pbest_scores[i] = current_score
            pbest_positions[i] = positions[i].copy()

    # --- Update GBest (Global Best) ---
    # Find the overall best position among all PBest's
    current_gbest_index = np.argmin(pbest_scores)
```

```
        current_gbest_score = pbest_scores[current_gbest_index]

        # Update GBest only if a better PBest was found
        if current_gbest_score < gbest_score:
            gbest_score = current_gbest_score
            gbest_position = pbest_positions[current_gbest_index].copy()

        # Record history for convergence plot
        history.append((gbest_score, gbest_position.copy()))

        # --- Phase 2: Update Velocity and Position ---

        # Generate two sets of random numbers R1 and R2
        r1 = np.random.rand(num_particles, dimensions) # Random_1
        r2 = np.random.rand(num_particles, dimensions) # Random_2

        # 1. Inertia component: W * v_i^t
        inertia_comp = w * velocities

        # 2. Cognitive component (PBest influence): C1 * r1 * (PBest_i - x_i^t)
        cognitive_comp = c1 * r1 * (pbest_positions - positions)

        # 3. Social component (GBest influence): C2 * r2 * (GBest - x_i^t)
        # NumPy handles broadcasting of the 1D gbest_position to the 2D positions matrix
        social_comp = c2 * r2 * (gbest_position - positions)

        # Update velocity: v_i^{t+1} = Inertia + Cognitive + Social
        velocities = inertia_comp + cognitive_comp + social_comp

        # Update position: x_i^{t+1} = x_i^t + v_i^{t+1}
        positions = positions + velocities

        # Apply position constraints (clipping to search space)
        positions = np.clip(positions, min_bound, max_bound)

        print(f"Iteration {iteration+1}/{max_iterations}: GBest Score = {gbest_score:.4e}")

    return gbest_position, gbest_score, history

# --- 3. Example Usage and Visualization ---
```

```python
def run_pso_example():
    # Set up PSO parameters
    search_range = (-5.12, 5.12) # Common range for Sphere function
    max_iter = 100

    # Run the PSO solver
    best_position, best_score, history = pso_optimizer(
        objective_func=sphere_function,
        num_particles=30,
        dimensions=2, # Using 2D for simple visualization
        search_range=search_range,
        max_iterations=max_iter
    )

    print("\n--- Results ---")
    print(f"Objective Function: Sphere Function")
    print(f"Best Position Found: {best_position}")
    print(f"Minimum Score (Fitness): {best_score:.6e}")

    # --- Visualization ---

    # Extract scores for convergence plot
    scores = [item[0] for item in history]

    plt.figure(figsize=(10, 5))

    # Plot 1: Convergence History
    plt.subplot(1, 2, 1)
    plt.plot(range(len(scores)), scores, color='darkorange', linewidth=2)
    plt.title('PSO Convergence History')
    plt.xlabel('Iteration')
    plt.ylabel('Global Best Score (Log Scale)', color='darkorange')
    plt.yscale('log') # Use log scale for better visualization of minimization
    plt.grid(True, which="both", ls="--")

    # Plot 2: Particle movement (only for 2D problems)
    if history and len(history[0][1]) == 2:
        plt.subplot(1, 2, 2)
```

```
        # Create a contour plot of the objective function
        x = np.linspace(search_range[0], search_range[1], 100)
        y = np.linspace(search_range[0], search_range[1], 100)
        X, Y = np.meshgrid(x, y)

        # Calculate Z values for the Sphere function across the grid
        Z = np.array([[sphere_function(np.array([X[i, j], Y[i, j]])) for j in range(100)] for i in
range(100)])

        plt.contourf(X, Y, Z, levels=50, cmap='viridis')
        plt.colorbar(label='Function Value')

        # Plot the final best position
        plt.plot(best_position[0], best_position[1], 'r*', markersize=15, label='Final GBest')
        plt.title('2D Search Space Visualization')
        plt.xlabel('Dimension 1 (X)')
        plt.ylabel('Dimension 2 (Y)')
        plt.legend()

    plt.tight_layout()
    plt.show()

# Execute the example
if __name__ == '__main__':
    run_pso_example()
```

Output:


PSO Convergence History
Global Best Score (Log Scale)
Iteration
2D Search Space Visualization
Final GBest
Dimension 2 (Y)
Dimension 1 (X)
Function Value

```
Starting PSO for 2 dimensions with 30 particles...
Initial GBest Score: 3.7597
Iteration 1/100: GBest Score = 3.7597e+00
Iteration 2/100: GBest Score = 2.9285e-01
Iteration 3/100: GBest Score = 5.7670e-02
Iteration 4/100: GBest Score = 5.7670e-02
Iteration 5/100: GBest Score = 5.7670e-02
Iteration 6/100: GBest Score = 5.7670e-02
Iteration 7/100: GBest Score = 5.7670e-02
Iteration 8/100: GBest Score = 5.7670e-02
Iteration 9/100: GBest Score = 3.8838e-02
Iteration 10/100: GBest Score = 3.8838e-02
Iteration 11/100: GBest Score = 2.5322e-02
Iteration 12/100: GBest Score = 5.3836e-03
Iteration 13/100: GBest Score = 5.3836e-03
Iteration 14/100: GBest Score = 4.9536e-03
Iteration 15/100: GBest Score = 4.9536e-03
Iteration 16/100: GBest Score = 4.9536e-03
Iteration 17/100: GBest Score = 3.5827e-03
Iteration 18/100: GBest Score = 2.9382e-03
Iteration 19/100: GBest Score = 2.9382e-03
Iteration 20/100: GBest Score = 1.3334e-03
Iteration 21/100: GBest Score = 1.3334e-03
Iteration 22/100: GBest Score = 1.3334e-03
Iteration 23/100: GBest Score = 3.8907e-04
Iteration 24/100: GBest Score = 3.8907e-04
Iteration 25/100: GBest Score = 3.8907e-04
Iteration 26/100: GBest Score = 1.5016e-04
Iteration 27/100: GBest Score = 1.3658e-04
Iteration 28/100: GBest Score = 1.9112e-05
Iteration 29/100: GBest Score = 1.9112e-05
Iteration 30/100: GBest Score = 1.9112e-05
Iteration 31/100: GBest Score = 1.9112e-05
Iteration 32/100: GBest Score = 1.9112e-05
Iteration 33/100: GBest Score = 1.9112e-05
Iteration 34/100: GBest Score = 1.9112e-05
Iteration 35/100: GBest Score = 3.2656e-06
Iteration 36/100: GBest Score = 3.2656e-06
Iteration 37/100: GBest Score = 3.2656e-06
Iteration 38/100: GBest Score = 3.2656e-06
Iteration 39/100: GBest Score = 3.2656e-06
Iteration 40/100: GBest Score = 3.2656e-06
Iteration 41/100: GBest Score = 3.2656e-06
Iteration 42/100: GBest Score = 3.2656e-06
Iteration 43/100: GBest Score = 3.2656e-06
Iteration 44/100: GBest Score = 7.1491e-07
```

Ant Colony Code

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

# --- 1. Utility Functions ---

def create_distance_matrix(cities):
   """Calculates the Euclidean distance matrix between all pairs of
cities."""
   num_cities = len(cities)
   dist_matrix = np.zeros((num_cities, num_cities))
   for i in range(num_cities):
       for j in range(i + 1, num_cities):
           # Calculate Euclidean distance
           distance = np.sqrt((cities[i][0] - cities[j][0])**2 +
(cities[i][1] - cities[j][1])**2)
           dist_matrix[i, j] = dist_matrix[j, i] = distance
   return dist_matrix

def calculate_tour_length(tour, dist_matrix):
   """Calculates the total length of a given tour (sequence of city
indices)."""
   length = 0
   num_cities = len(tour)
   for i in range(num_cities):
       # Add distance from current city to next city in the tour
       city_a = tour[i]
       city_b = tour[(i + 1) % num_cities] # Wrap around to the start city
       length += dist_matrix[city_a, city_b]
   return length

# --- 2. ACO Algorithm Implementation ---

def aco_tsp_solver(cities, num_ants=10, max_iterations=100, alpha=1.0,
beta=5.0, rho=0.5, initial_pheromone=1.0):
   """
   Ant Colony Optimization (ACO) algorithm for the Traveling Salesman
Problem (TSP).

   Args:
       cities (list of tuples): List of (x, y) coordinates for each city.
       num_ants (int): Number of artificial ants (M).
       max_iterations (int): Maximum number of generations to run.
```

```
        alpha (float): Influence of the pheromone trail (tau).
        beta (float): Influence of the heuristic information (eta,
1/distance).
        rho (float): Pheromone evaporation rate.
        initial_pheromone (float): Initial pheromone value (tau_0).
    """
    num_cities = len(cities)
    dist_matrix = create_distance_matrix(cities)

    # Heuristic matrix (eta_ij = 1 / distance_ij)
    # Avoid division by zero for d_ii by setting eta_ii to zero
    eta_matrix = 1.0 / (dist_matrix + np.finfo(float).eps)
    np.fill_diagonal(eta_matrix, 0)

    # Initialize pheromone matrix (tau_ij)
    pheromone_matrix = np.full((num_cities, num_cities), initial_pheromone)

    # Initialize best tour found so far
    best_tour = None
    best_length = float('inf')
    history = []

    print(f"Starting ACO for {num_cities} cities with {num_ants} ants...")

    for iteration in range(max_iterations):
        all_tours = []
        all_lengths = []

        # --- Phase 1: Tour Construction ---
        for ant in range(num_ants):
            start_city = random.randint(0, num_cities - 1)
            tour = [start_city]
            visited = {start_city}

            for _ in range(num_cities - 1):
                current_city = tour[-1]
                unvisited_cities = [c for c in range(num_cities) if c not
in visited]

                if not unvisited_cities:
                    break # Should not happen if TSP is solvable

                # Calculate probabilities P_ij
                probabilities = []
                denominator = 0.0
```

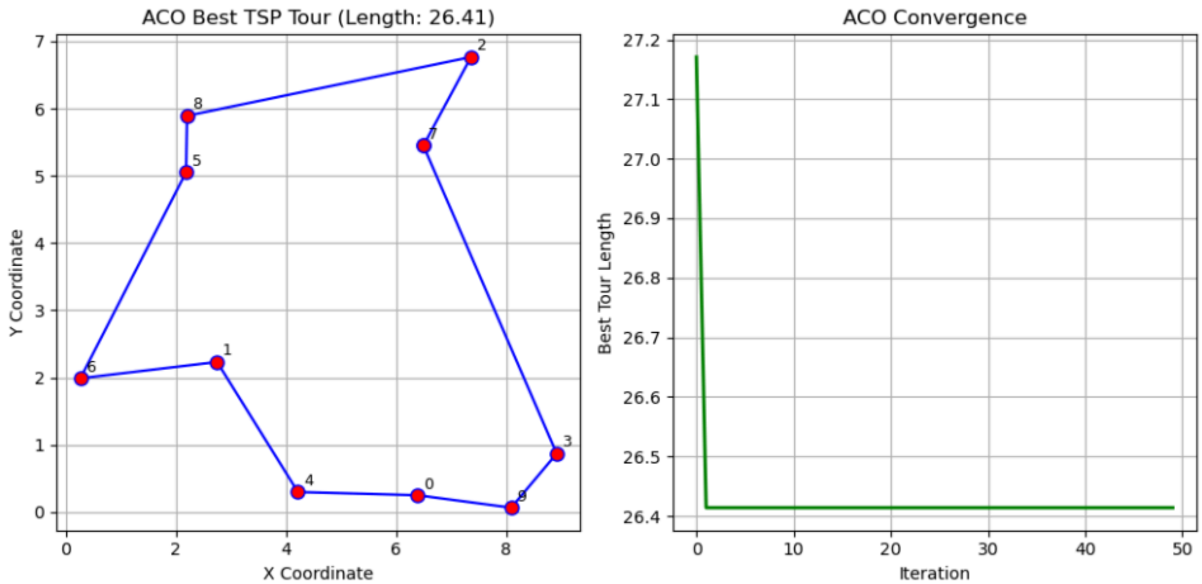
```
                for next_city in unvisited_cities:
                    tau = pheromone_matrix[current_city, next_city] ** alpha
                    eta = eta_matrix[current_city, next_city] ** beta
                    numerator = tau * eta
                    probabilities.append((next_city, numerator))
                    denominator += numerator

                if denominator == 0:
                    # Fallback to random choice if all probabilities are zero (rare)
                    next_city = random.choice(unvisited_cities)
                else:
                    # Select next city based on roulette wheel selection (weighted probability)
                    prob_values = [p[1] / denominator for p in probabilities]
                    next_city = random.choices(
                        [p[0] for p in probabilities],
                        weights=prob_values,
                        k=1
                    )[0]

                tour.append(next_city)
                visited.add(next_city)

            tour_length = calculate_tour_length(tour, dist_matrix)
            all_tours.append(tour)
            all_lengths.append(tour_length)

            # Update personal best (used for global best update below)
            if tour_length < best_length:
                best_length = tour_length
                best_tour = tour

        history.append(best_length)

        # --- Phase 2: Pheromone Update ---

        # 1. Evaporation: tau_ij = (1 - rho) * tau_ij
        pheromone_matrix = (1 - rho) * pheromone_matrix

        # 2. Deposition: The best ant deposits pheromone (Ant System variant)
        if best_tour is not None:
```

```
            # Pheromone deposit (Delta_tau = 1 / BestLength)
            delta_tau = 1.0 / best_length

            # Deposit pheromone along the best tour
            for i in range(num_cities):
                city_a = best_tour[i]
                city_b = best_tour[(i + 1) % num_cities]
                pheromone_matrix[city_a, city_b] += delta_tau
                pheromone_matrix[city_b, city_a] += delta_tau # TSP graph
is symmetric

        print(f"Iteration {iteration+1}/{max_iterations}: Best Length =
{best_length:.2f}")

    return best_tour, best_length, history

# --- 3. Example Usage and Visualization ---

def run_aco_example():
    # Define a simple set of 10 cities (x, y coordinates)
    random.seed(42) # for reproducibility
    np.random.seed(42)
    cities = [(random.uniform(0, 10), random.uniform(0, 10)) for _ in
range(10)]

    # Run the ACO solver
    best_tour, best_length, history = aco_tsp_solver(
        cities,
        num_ants=20,
        max_iterations=50,
        alpha=1.0,
        beta=5.0,
        rho=0.1
    )

    print("\n--- Results ---")
    print(f"Cities: {cities}")
    print(f"Best Tour (City Indices): {best_tour}")
    print(f"Best Tour Length: {best_length:.4f}")

    # --- Visualization ---
    if best_tour:
        # Prepare coordinates for plotting the best tour path
        x_coords = [cities[i][0] for i in best_tour]
        y_coords = [cities[i][1] for i in best_tour]
```

```
        # Close the loop for visualization
        x_coords.append(x_coords[0])
        y_coords.append(y_coords[0])

        plt.figure(figsize=(10, 5))

        # Plot 1: Tour Path
        plt.subplot(1, 2, 1)
        plt.plot(x_coords, y_coords, 'o-', color='blue',
markerfacecolor='red', markersize=8)
        # Label cities with their index
        for i, (x, y) in enumerate(cities):
            plt.text(x + 0.1, y + 0.1, str(i), fontsize=9)
        plt.title(f'ACO Best TSP Tour (Length: {best_length:.2f})')
        plt.xlabel('X Coordinate')
        plt.ylabel('Y Coordinate')
        plt.grid(True)

        # Plot 2: Convergence History
        plt.subplot(1, 2, 2)
        plt.plot(history, color='green', linewidth=2)
        plt.title('ACO Convergence')
        plt.xlabel('Iteration')
        plt.ylabel('Best Tour Length')
        plt.grid(True)
        plt.tight_layout()
        plt.show()

# Execute the example
if __name__ == '__main__':
    run_aco_example()
```

Output:

```
--- Results ---
Cities: [(6.394267984578837, 0.25010755222666936), (2.7502931836911926, 2.2321073814882277), (7.364712141640124, 6.766994874229113),
(8.921795677048454, 0.8693883262941615), (4.2192181968527045, 0.29797219438070344), (2.1863797480360336, 5.053552881033624), (0.2653
5969683863625, 1.988376506866485), (6.498844377795232, 5.449414806032166), (2.204406220406967, 5.892656838759088), (8.09430456677826
6, 0.06498759678061017)]
Best Tour (City Indices): [7, 2, 8, 5, 6, 1, 4, 0, 9, 3]
Best Tour Length: 26.4139
```



```
Starting ACO for 10 cities with 20 ants...
Iteration 1/50: Best Length = 27.17
Iteration 2/50: Best Length = 26.41
Iteration 3/50: Best Length = 26.41
Iteration 4/50: Best Length = 26.41
Iteration 5/50: Best Length = 26.41
Iteration 6/50: Best Length = 26.41
Iteration 7/50: Best Length = 26.41
Iteration 8/50: Best Length = 26.41
Iteration 9/50: Best Length = 26.41
Iteration 10/50: Best Length = 26.41
Iteration 11/50: Best Length = 26.41
Iteration 12/50: Best Length = 26.41
Iteration 13/50: Best Length = 26.41
Iteration 14/50: Best Length = 26.41
Iteration 15/50: Best Length = 26.41
Iteration 16/50: Best Length = 26.41
Iteration 17/50: Best Length = 26.41
Iteration 18/50: Best Length = 26.41
Iteration 19/50: Best Length = 26.41
Iteration 20/50: Best Length = 26.41
Iteration 21/50: Best Length = 26.41
Iteration 22/50: Best Length = 26.41
Iteration 23/50: Best Length = 26.41
Iteration 24/50: Best Length = 26.41
Iteration 25/50: Best Length = 26.41
Iteration 26/50: Best Length = 26.41
Iteration 27/50: Best Length = 26.41
Iteration 28/50: Best Length = 26.41
Iteration 29/50: Best Length = 26.41
Iteration 30/50: Best Length = 26.41
Iteration 31/50: Best Length = 26.41
Iteration 32/50: Best Length = 26.41
Iteration 33/50: Best Length = 26.41
Iteration 34/50: Best Length = 26.41
Iteration 35/50: Best Length = 26.41
Iteration 36/50: Best Length = 26.41
Iteration 37/50: Best Length = 26.41
Iteration 38/50: Best Length = 26.41
Iteration 39/50: Best Length = 26.41
Iteration 40/50: Best Length = 26.41
Iteration 41/50: Best Length = 26.41
Iteration 42/50: Best Length = 26.41
Iteration 43/50: Best Length = 26.41
Iteration 44/50: Best Length = 26.41
Iteration 45/50: Best Length = 26.41
Iteration 46/50: Best Length = 26.41
Iteration 47/50: Best Length = 26.41
```