**PARTICLE SWARM OPTIMIZATION**

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.

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 ---

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()

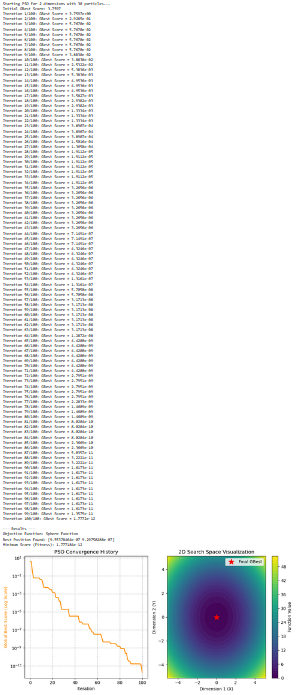
plt.show()

# Execute the example

if \_\_name\_\_ == '\_\_main\_\_':

run\_pso\_example()

Output:



**ANT COLONY OPTIMIZATION**

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:

