Assignment No. 7

Title: Optimization of Warehouse Layout and Data Clustering Using Particle Swarm Optimization
Aim: Write a program to optimize the layout of a warehouse or Data Clustering problem using particle swarm optimization algorithm.
Objectives

- Implement the PSO algorithm for optimizing warehouse layout to minimize travel distance and maximize efficiency.
- Apply PSO for data clustering to group similar data points and compare its performance with standard clustering algorithms.
- Analyse the convergence and effectiveness of PSO in both scenarios.

Theory:
Introduction:
Particle Swarm Optimization (PSO) is a population-based metaheuristic algorithm inspired by the social behaviour of birds flocking or fish schooling. It is widely used for solving complex optimization problems, including warehouse layout design and data clustering. In warehouse layout optimization, PSO helps minimize travel distance and maximize space utilization by determining the optimal arrangement of storage locations and aisles. In data clustering, PSO is used to group similar data points by optimizing the placement of cluster centroids, often outperforming traditional methods like k-means in avoiding local minima.
Particle swarm optimization:
Below is a concise, practical step-by-step description of the canonical Particle Swarm Optimization (PSO) loop, plus a compact pseudocode you can use as a template.
Goal: minimize (or maximize) an objective function

$$f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n$$

Step-by-step PSO
Define the problem and the objective.

- Decide whether you minimise or maximise f(x).
- Specify the search space (bounds for each dimension).

Set PSO hyperparameters

- Swarm size (S) (number of particles).
- Inertia weight (w) (controls momentum).
- Cognitive coefficient (c_1) (particle's self-attraction).
- Social coefficient (c_2) (swarm attraction).
- Maximum velocity (v_{\max}) (optional clamping).
- Stopping criteria: max iterations, target fitness, or stagnation threshold.

Initialize swarm

- For each particle (i=1…S):
    - Initialise position x_i randomly within bounds.
    - Initialise velocity v_i (often zeros or small random values).
    - Set personal best p_i -> x_i.
    - Evaluate fitness f(p_i).

- Set global best g to the best p_i.

Main loop (repeat until stopping criteria)
For each particle (i):
Draw two random vectors r1,r2~U(0,1) (elementwise).
Update velocity:

$$\begin{aligned}\mathbf{v}_i \leftarrow\ & w\mathbf{v}_i \\ & + c_1\,\mathbf{r}_1 \odot (\mathbf{p}_i - \mathbf{x}_i) \\ & + c_2\,\mathbf{r}_2 \odot (\mathbf{g} - \mathbf{x}_i)\end{aligned}$$

Velocity clamping (optional): clip each component of

$$\mathbf{v}_i \in [-v_{\max},\ v_{\max}]$$

Update position:

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$$

- Handle boundaries: if x_i goes outside bounds, either clamp to boundary, reflect velocity, or wrap—choose based on problem semantics.
- Evaluate fitness: compute f(x_i).
- Update personal best: if f(x_i) better than f(p_i), set p_i <- x_i.
- Update global best: if p_i is better than current g, set g <- p_i.
- Optionally update (w) (e.g., linearly decrease) or adapt coefficients.
- Stop and return
- When stopping condition met, return g and f(g).
- Optionally perform a local search starting from g for refinement or restart PSO from different initial seeds.

Practical tips & common variants (short)

- Inertia scheduling: start with a larger (w) (favour exploration) and reduce it to favour exploitation.
- Constriction factor: an alternative to clamping; ensures theoretical convergence in some formulations.
- Neighbourhood (lBest) PSO: use a local best among neighbours instead of a single global best — helps avoid premature convergence.
- Binary or discrete PSO: different position/velocity interpretation for combinatorial problems.
- Perform multiple restarts or hybridize with local search if stagnation occurs.

Methodology

1. Warehouse Layout Optimization
    - Define the warehouse dimensions and storage requirements.
    - Represent each particle as a possible layout configuration (e.g., aisle positions, storage locations).
    - Use the travel distance or space utilization as the fitness function.
    - Update particle positions and velocities using standard PSO equations until convergence.
2. Data Clustering

- Initialize particles as cluster centroids.
- Assign data points to the nearest centroid and calculate the fitness (e.g., sum of squared errors).
- Update particle positions (centroids) using PSO equations.
- Repeat until convergence or maximum iterations.
-

Conclusion:

In conclusion, Particle Swarm Optimization (PSO) proved to be an efficient and flexible technique for solving both warehouse layout optimization and data clustering problems. It effectively minimized travel distance and improved layout efficiency, while also producing better clustering results by avoiding local minima compared to traditional methods like k-means. Overall, PSO demonstrated strong convergence, adaptability, and reliability for complex optimization tasks.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Generate synthetic data
data, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# PSO parameters
num_particles = 30
num_iterations = 100
num_clusters = 4
w = 0.5      # inertia weight
c1 = 1.5     # cognitive coefficient
c2 = 1.5     # social coefficient

# Initialize particles
class Particle:
    def __init__(self, data, num_clusters):
        self.data = data
        self.num_clusters = num_clusters
        self.position = data[np.random.choice(range(len(data)), num_clusters)]
        self.velocity = np.zeros_like(self.position)
        self.best_position = np.copy(self.position)
        self.best_score = self.evaluate()

    def evaluate(self):
        distances = np.linalg.norm(self.data[:, None] - self.position[None, :], axis=2)
        closest = np.argmin(distances, axis=1)
        score = sum(np.linalg.norm(self.data[i] - self.position[closest[i]])**2 for i in range(len(self.data)))
        return score

    def update(self, global_best):
        r1, r2 = np.random.rand(), np.random.rand()
        cognitive = c1 * r1 * (self.best_position - self.position)
        social = c2 * r2 * (global_best - self.position)
        self.velocity = w * self.velocity + cognitive + social
        self.position += self.velocity
        score = self.evaluate()
        if score < self.best_score:
            self.best_score = score
            self.best_position = np.copy(self.position)

# Initialize swarm
swarm = [Particle(data, num_clusters) for _ in range(num_particles)]
global_best = min(swarm, key=lambda p: p.best_score).best_position

# PSO loop
for _ in range(num_iterations):
    for particle in swarm:
        particle.update(global_best)
    global_best = min(swarm, key=lambda p: p.best_score).best_position

# Final clustering
distances = np.linalg.norm(data[:, None] - global_best[None, :], axis=2)
labels = np.argmin(distances, axis=1)
```

```
# Plot results
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap='viridis')
plt.scatter(global_best[:, 0], global_best[:, 1], c='red', marker='x')
plt.title("PSO-based Clustering")
plt.show()
```

Output: