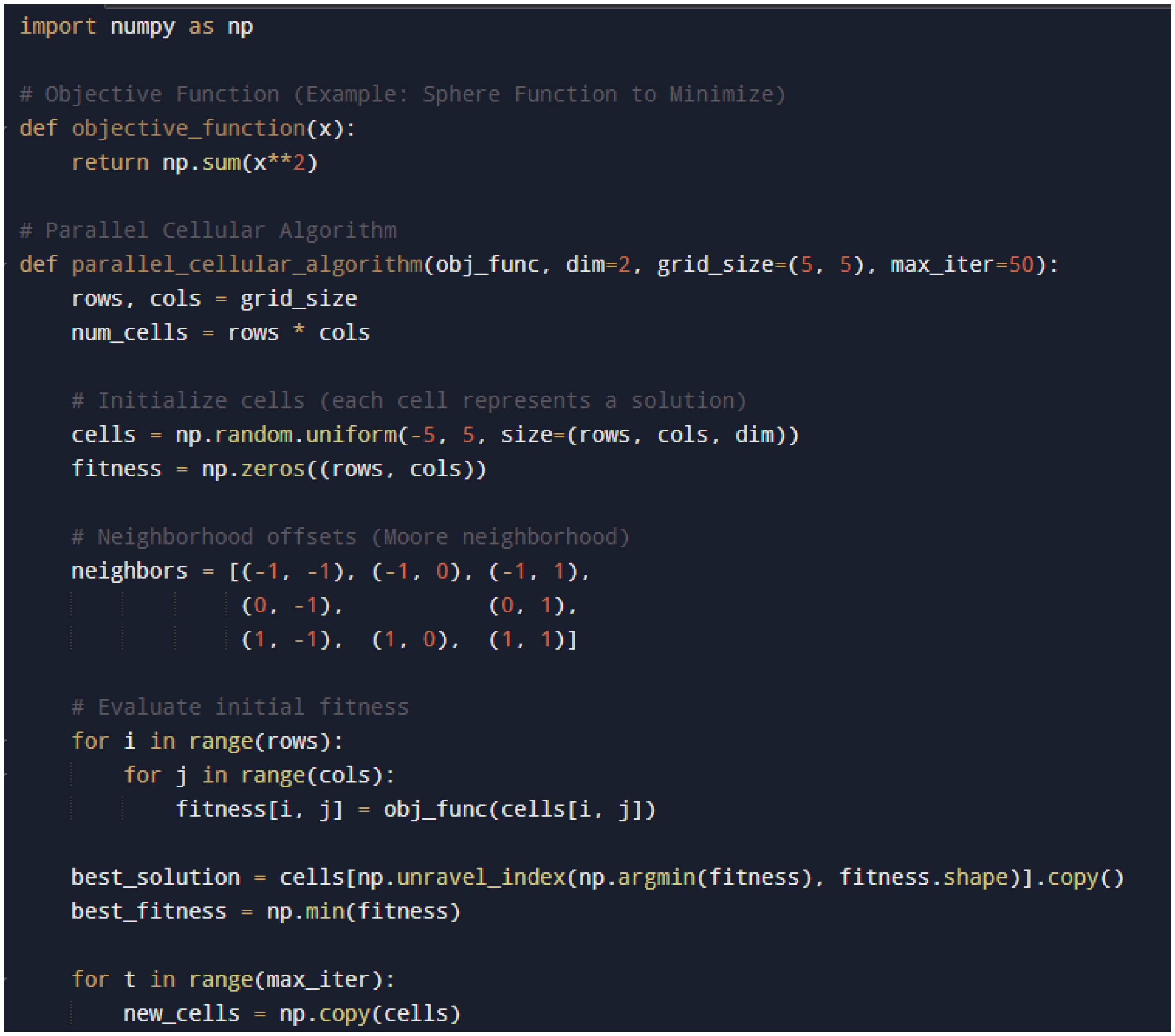
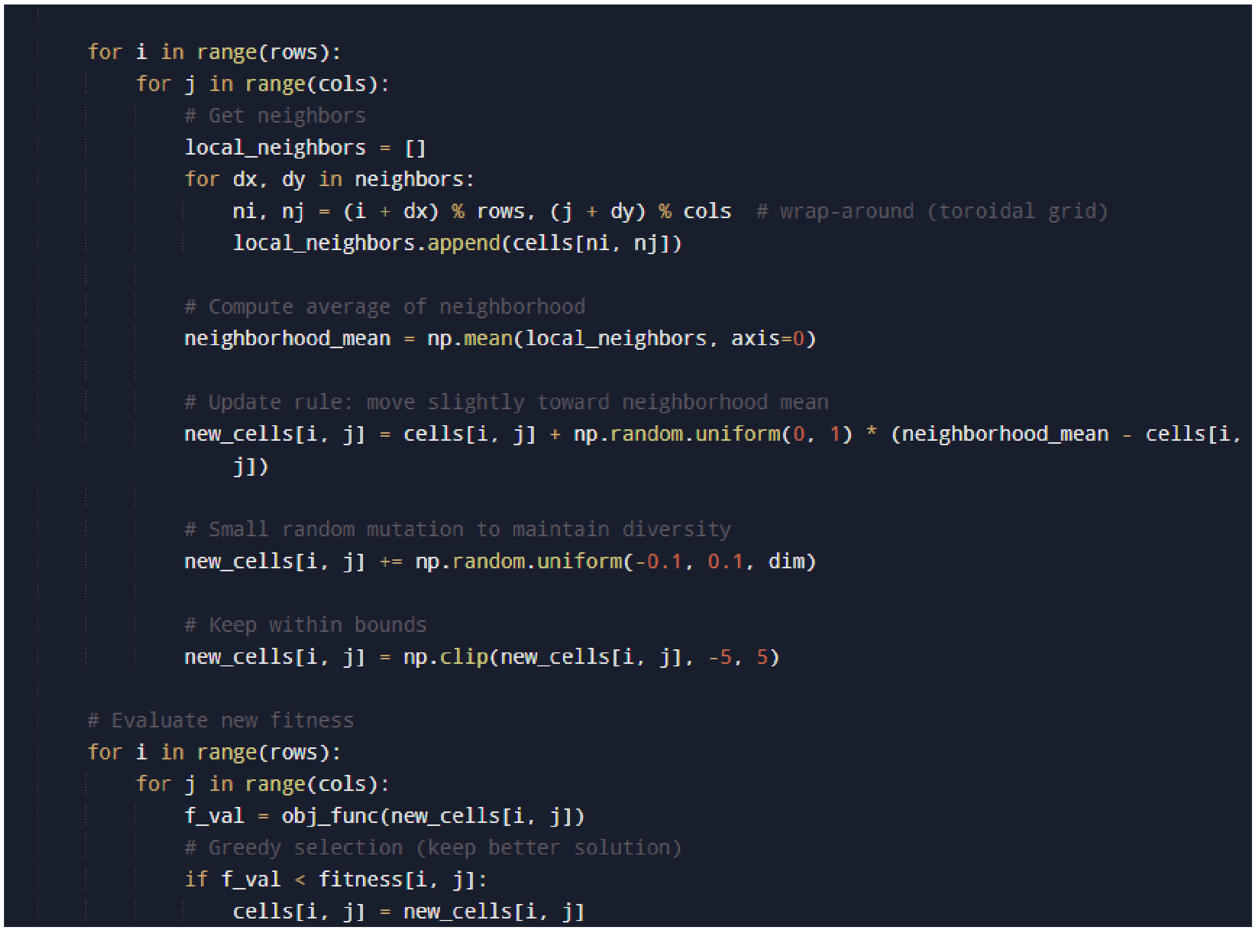
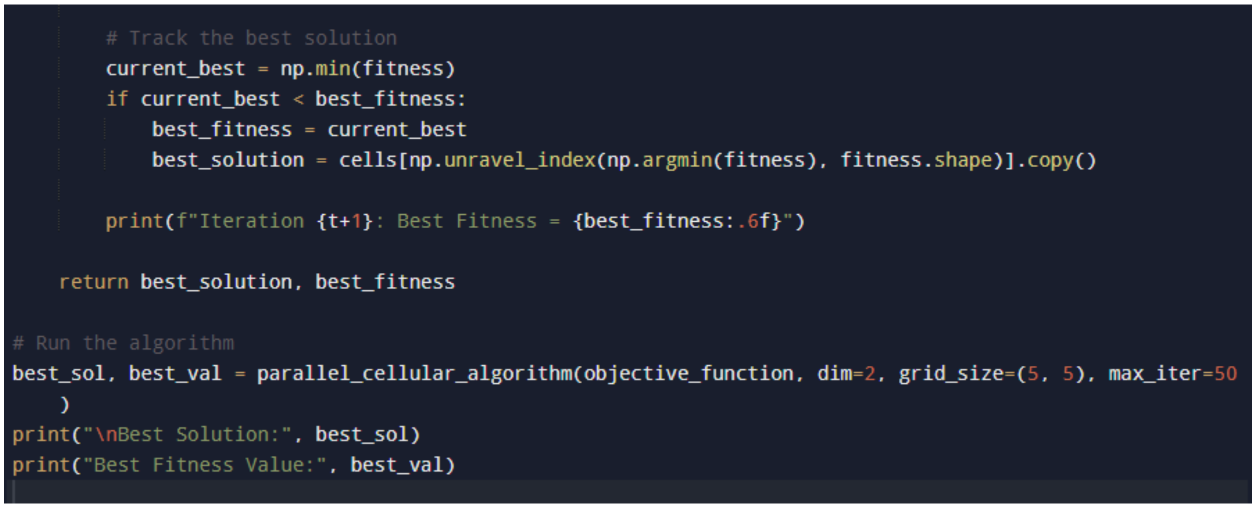
Parallel Cellular Algorithm (PCA)



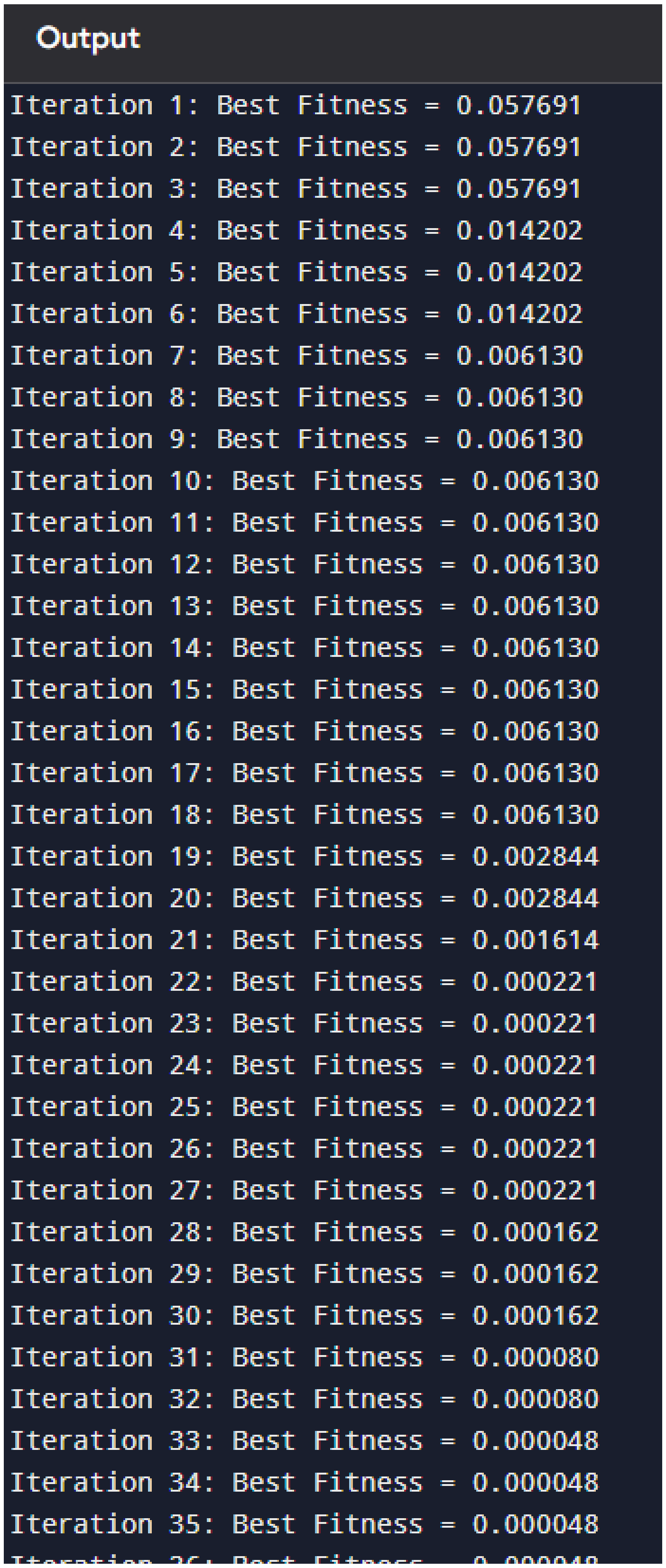
```python
import numpy as np

# Objective Function (Example: Sphere Function to Minimize)
def objective_function(x):
    return np.sum(x**2)

# Parallel Cellular Algorithm
def parallel_cellular_algorithm(obj_func, dim=2, grid_size=(5, 5), max_iter=50):
    rows, cols = grid_size
    num_cells = rows * cols

    # Initialize cells (each cell represents a solution)
    cells = np.random.uniform(-5, 5, size=(rows, cols, dim))
    fitness = np.zeros((rows, cols))

    # Neighborhood offsets (Moore neighborhood)
    neighbors = [(-1, -1), (-1, 0), (-1, 1),
                 (0, -1),           (0, 1),
                 (1, -1),  (1, 0),  (1, 1)]

    # Evaluate initial fitness
    for i in range(rows):
        for j in range(cols):
            fitness[i, j] = obj_func(cells[i, j])

    best_solution = cells[np.unravel_index(np.argmin(fitness), fitness.shape)].copy()
    best_fitness = np.min(fitness)

    for t in range(max_iter):
        new_cells = np.copy(cells)
```



```
        for i in range(rows):
            for j in range(cols):
                # Get neighbors
                local_neighbors = []
                for dx, dy in neighbors:
                    ni, nj = (i + dx) % rows, (j + dy) % cols  # wrap-around (toroidal grid)
                    local_neighbors.append(cells[ni, nj])

                # Compute average of neighborhood
                neighborhood_mean = np.mean(local_neighbors, axis=0)

                # Update rule: move slightly toward neighborhood mean
                new_cells[i, j] = cells[i, j] + np.random.uniform(0, 1) * (neighborhood_mean - cells[i,
                    j])

                # Small random mutation to maintain diversity
                new_cells[i, j] += np.random.uniform(-0.1, 0.1, dim)

                # Keep within bounds
                new_cells[i, j] = np.clip(new_cells[i, j], -5, 5)

        # Evaluate new fitness
        for i in range(rows):
            for j in range(cols):
                f_val = obj_func(new_cells[i, j])
                # Greedy selection (keep better solution)
                if f_val < fitness[i, j]:
                    cells[i, j] = new_cells[i, j]
```



```
        # Track the best solution
        current_best = np.min(fitness)
        if current_best < best_fitness:
            best_fitness = current_best
            best_solution = cells[np.unravel_index(np.argmin(fitness), fitness.shape)].copy()

        print(f"Iteration {t+1}: Best Fitness = {best_fitness:.6f}")

    return best_solution, best_fitness

# Run the algorithm
best_sol, best_val = parallel_cellular_algorithm(objective_function, dim=2, grid_size=(5, 5), max_iter=50
    )
print("\nBest Solution:", best_sol)
print("Best Fitness Value:", best_val)
```


Output

```
Iteration 1: Best Fitness = 0.057691
Iteration 2: Best Fitness = 0.057691
Iteration 3: Best Fitness = 0.057691
Iteration 4: Best Fitness = 0.014202
Iteration 5: Best Fitness = 0.014202
Iteration 6: Best Fitness = 0.014202
Iteration 7: Best Fitness = 0.006130
Iteration 8: Best Fitness = 0.006130
Iteration 9: Best Fitness = 0.006130
Iteration 10: Best Fitness = 0.006130
Iteration 11: Best Fitness = 0.006130
Iteration 12: Best Fitness = 0.006130
Iteration 13: Best Fitness = 0.006130
Iteration 14: Best Fitness = 0.006130
Iteration 15: Best Fitness = 0.006130
Iteration 16: Best Fitness = 0.006130
Iteration 17: Best Fitness = 0.006130
Iteration 18: Best Fitness = 0.006130
Iteration 19: Best Fitness = 0.002844
Iteration 20: Best Fitness = 0.002844
Iteration 21: Best Fitness = 0.001614
Iteration 22: Best Fitness = 0.000221
Iteration 23: Best Fitness = 0.000221
Iteration 24: Best Fitness = 0.000221
Iteration 25: Best Fitness = 0.000221
Iteration 26: Best Fitness = 0.000221
Iteration 27: Best Fitness = 0.000221
Iteration 28: Best Fitness = 0.000162
Iteration 29: Best Fitness = 0.000162
Iteration 30: Best Fitness = 0.000162
Iteration 31: Best Fitness = 0.000080
Iteration 32: Best Fitness = 0.000080
Iteration 33: Best Fitness = 0.000048
Iteration 34: Best Fitness = 0.000048
Iteration 35: Best Fitness = 0.000048
Iteration 36: Best Fitness = 0.000048
```