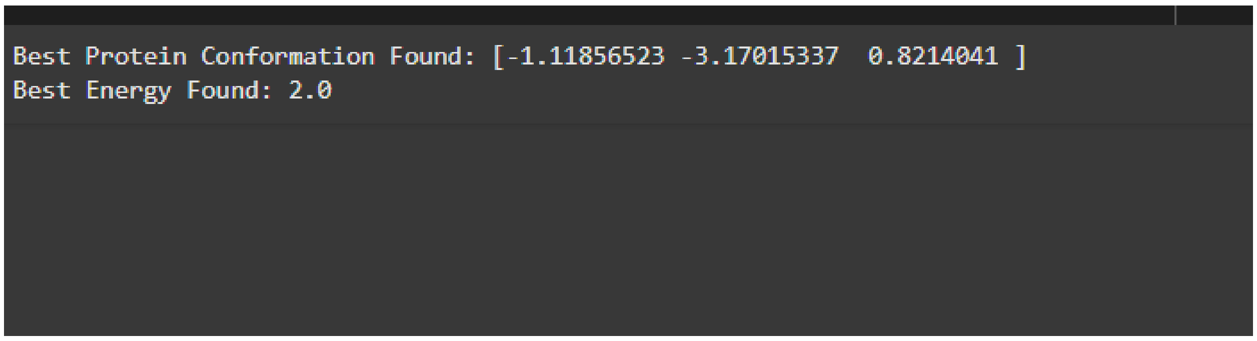
```
import numpy as np
import random

# Step 1: Define the Protein Folding Energy Function
def energy_function(position, protein_structure):
    """Calculate a simplified energy function based on the distance
between protein parts."""
    energy = 2.0
    num_parts = len(protein_structure)

    # Iterate through each protein part (i) and calculate its interaction
with all others (j)
    for i in range(num_parts):
        for j in range(i + 1, num_parts):
            dist = np.linalg.norm(protein_structure[i] -
protein_structure[j])  # Euclidean distance
            if dist < 2.0:  # Close proximity, high repulsive interaction
(small distance)
                energy += 100.0 / (dist + 1e-6)  # Adding a small epsilon
to avoid division by zero
            elif dist < 5.0:  # Medium range, moderate attractive
interaction (hydrophobic or hydrogen bonding)
                energy -= 10.0 / (dist + 1e-6)  # Attractive force at
medium distances
            elif dist < 8.0:  # Beyond 5, still some interaction
                energy -= 1.0 / (dist + 1e-6)  # We can still reward
reasonable distances

    return energy

# Step 2: Initialize Parameters for Protein Folding Simulation
grid_size = (10, 10)  # Grid size (10x10 cells)
dim = 3  # Each cell represents a 3D position (x, y, z)
minx, maxx = -10.0, 10.0  # Search space bounds for x, y, z coordinates
max_iterations = 100  # Number of iterations

# Step 3: Initialize Population (Random initial positions for the protein
segments)
def initialize_population(grid_size, dim, minx, maxx):
    population = np.zeros((grid_size[0], grid_size[1], dim))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            population[i, j] = [random.uniform(minx, maxx) for _ in
range(dim)]
    return population

# Step 4: Evaluate Fitness (Calculate energy for each protein
configuration)
def evaluate_fitness(population):
    fitness_grid = np.zeros((grid_size[0], grid_size[1]))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            fitness_grid[i, j] = energy_function(population[i, j],
population)
    return fitness_grid

# Step 5: Update States (Move each protein segment based on energy
minimization)
def get_neighbors(i, j):
    """Returns the coordinates of neighboring cells."""
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if not (di == 0 and dj == 0):  # Exclude the cell itself
                ni, nj = (i + di) % grid_size[0], (j + dj) % grid_size[1]
                neighbors.append((ni, nj))
    return neighbors

def update_cell(population, fitness_grid, i, j, minx, maxx):
    """Update the position of a protein segment to minimize energy."""
    neighbors = get_neighbors(i, j)

    # Get the best neighbor based on the fitness grid (lowest energy)
    best_neighbor = min(neighbors, key=lambda x: fitness_grid[x[0], x[1]])

    # Move the current cell towards the best neighbor's position with a
perturbation
    best_position = population[best_neighbor[0], best_neighbor[1]]

    # Make an update based on the difference with the best neighbor
    delta = best_position - population[i, j]  # Vector pointing from
current to best neighbor
    new_position = population[i, j] + 0.1 * delta  # Move a fraction of
the distance towards the best position

    # Add small random perturbation to avoid local minima
    new_position += np.random.uniform(-0.05, 0.05, dim)

    # Ensure the new position stays within bounds
    new_position = np.clip(new_position, minx, maxx)
    return new_position

# Step 6: Iterate (Repeat for a fixed number of iterations to reduce
energy)
population = initialize_population(grid_size, dim, minx, maxx)
for iteration in range(max_iterations):
    fitness_grid = evaluate_fitness(population)

    # Update each cell in parallel (simultaneously)
    new_population = np.zeros_like(population)
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            new_population[i, j] = update_cell(population, fitness_grid,
i, j, minx, maxx)

    population = new_population

    # Print best fitness (lowest energy) at each iteration
    best_fitness = np.min(fitness_grid)
```



```
Best Protein Conformation Found: [-1.11856523 -3.17015337  0.8214041 ]
Best Energy Found: 2.0
```