8-8.py

```python
import numpy as np
import csv, copy, time
import matplotlib.pyplot as plt
from scipy.spatial import Voronoi, voronoi_plot_2d
from collections import deque

def InitializeParticles(l, n):
    positions = l*np.random.rand(n, 2)
    theta = 2*np.pi*np.random.rand(n,1)
    return positions, theta

def SaveData(data, filename):
    with open(filename, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerows(data)

def LoadData(filename):
    data = []
    with open(filename, 'r') as file:
        for line in file:
            line = line.split(',')
            line = [float(l) for l in line]
            data.append(line)
    return np.array(data)

def FindNeighbors(positions, rf, size):
    distances = [[] for _ in range(len(positions))]
    neighbors = [[] for _ in range(len(positions))]

    # In-line functions for different distance functions
    EuclidianDistance = lambda p1, p2: np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
    WrappedDistance = lambda p1, p2, l: np.sqrt((l - abs(p1[0] - p2[0]))**2 + (l - abs(p1[1] - p2[1]))**2)

    # Calculate distances to other particles (Euclidian and wrapped)
    for i, x in enumerate(positions):
        for j, y in enumerate(positions):
            dist = [EuclidianDistance(x, y), WrappedDistance(x, y, size)]
            distances[i].append(min(dist))

    # Determine if other particles are within radius
    for i in range(len(distances)):
        for j in range(len(distances[0])):
            if distances[i][j] < rf:
                neighbors[i].append(j)

    return neighbors

def OrientationUpdate(angles, neigborhood, eta, dt, history, h):
    if len(history) < h or h == 0:
```

```python
        for i, neighbors in enumerate(neigborhood):
            thetas = [angles[a] for _, a in enumerate(neighbors)]
            avgSin = np.mean([np.sin(thetaK) for thetaK in thetas])
            avgCos = np.mean([np.cos(thetaK) for thetaK in thetas])
            avgTheta = np.arctan(avgSin/avgCos)
            w = np.random.uniform(-1/2, 1/2)
            angles[i] = avgTheta + eta*w*dt
    elif h < 0:
        pass
        for i, neighbors in enumerate(neigborhood):
            # Single out particle history
            thetas = [[history[i][k] for i in range(len(history))] for _, k in enumerate(neighbors)]
            x = np.linspace(0, abs(h), num=abs(h))
            print(len(x), len(thetas[0]))
            # Determine trajectory
            coefficients = [np.polyfit(x, tH, 1) for tH in thetas]
            x1 = h+1
            # Extrapolate
            thetas1 = [np.polyval(c, x1) for c in coefficients]
            # Average trajectories
            avgSin = np.mean([np.sin(thetaK) for thetaK in thetas1])
            avgCos = np.mean([np.cos(thetaK) for thetaK in thetas1])
            avgTheta = np.arctan(avgSin/avgCos)
            # Randomize W
            w = np.random.uniform(-1/2, 1/2)
            angles[i] = avgTheta + eta*w*dt
    else:
        for i, neighbors in enumerate(neigborhood):
            thetas = [history[0][a] for _, a in enumerate(neighbors)]
            avgSin = np.mean([np.sin(thetaK) for thetaK in thetas])
            avgCos = np.mean([np.cos(thetaK) for thetaK in thetas])
            avgTheta = np.arctan(avgSin/avgCos)
            w = np.random.uniform(-1/2, 1/2)
            angles[i] = avgTheta + eta*w*dt
    return angles

def UpdatePositions(positions, v, angle, size):
    for i in range(len(positions)):
        positions[i][0] += v*np.cos(angle[i])
        positions[i][1] += v*np.sin(angle[i])

        # Ensure particle stays within grid (If it moves outside it placed on the other side (Wraparound))
        if positions[i, 0] > size:
            positions[i, 0] -= size
        elif positions[i, 0] < 0:
            positions[i, 0] += size

        if positions[i, 1] > size:
            positions[i, 1] -= size
        elif positions[i, 1] < 0:
            positions[i, 1] += size
    return positions
```

```python

def UpdateParticles(positions, angles, rf, l, eta, dt, v, thetaHistory, h):
    neighbors = FindNeighbors(positions, rf, l)
    angles = OrientationUpdate(angles, neighbors, eta, dt, thetaHistory, h)
    positions = UpdatePositions(positions, v, angles, l)
    return positions, angles

def FindCoefficients(positions, angles, rf):
    '''
    Function to calculate alignment and clustering coefficients

    Velocities calculated according to eqn. 8.3
    Alignment coefficient calculated with eqn. 8.5
    '''
    n = len(positions)
    vor = Voronoi(positions)

    # Alignment coefficient
    alignC = np.sum(np.cos(angles))/n

    # Clustering coefficient
    clusterCount = 0
    for _, regionIdx in enumerate(vor.point_region):
        region = vor.regions[regionIdx]
        if not -1 in region and len(region) > 2:
            area = 0.5 * np.abs(np.dot(vor.vertices[region, 0], np.roll(vor.vertices[region, 1], 1)) - np.dot(vor.vertices[region, 1], np.roll(vor.vertices[region, 0], 1)))
            if area < np.pi*rf**2:
                clusterCount += 1
    clusterC = clusterCount/n
    return alignC, clusterC

def PlotFunction(initialPositions, finalPositions, coefficients):

    fig, axs = plt.subplots(1, 3)

    # Plot the initial conditions
    ax = axs[0]
    data = initialPositions
    # Scatter plot of positions
    positions = [[data[j][i] for j in range(len(data))] for i in range(len(data[0]))]
    ax.scatter(positions[0], positions[1], s=10, color='b')
    # Add voroni tesselation
    vor = Voronoi(data)
    voronoi_plot_2d(vor, line_colors='k', line_width=1, show_vertices=False, line_alpha=0.5, point_size=0, ax=ax)
    ax.set_title('Initial positions')
    ax.set_xlabel('X: L')
    ax.set_ylabel('Y: L')

    # Plot the final conditions
    ax = axs[1]
    data = finalPositions
```

```
    # Scatter plot of positions
    positions = [[data[j][i] for j in range(len(data))] for i in range(len(data[0]))]
    ax.scatter(positions[0], positions[1], s=10, color='b')
    # Add voroni tesselation
    vor = Voronoi(data)
    voronoi_plot_2d(vor, line_colors='k', line_width=1, show_vertices=False, line_alpha=0.5, point_size=0, ax=ax)
    ax.set_title('Final positions')
    ax.set_xlabel('X: L')
    ax.set_ylabel('Y: L')

    # Plot the coefficients
    ax = axs[2]
    alignmentData = coefficients[0]
    clusteringData = coefficients[1]
    x = np.linspace(0, len(alignmentData), num=len(alignmentData))
    ax = axs[2]

    # Plot alignment data
    ax.plot(x, alignmentData, label='Alignment', color='r')
    ax.set_title('Alignment and Clustering Coefficients')
    ax.set_xlabel('Time Step')
    ax.set_ylabel('Coefficient Value')

    # Plot clustering data
    ax.plot(x, clusteringData, label='Clustering', color='g')

    # Add legend
    ax.legend()

    # Show the plot
    plt.show()

def VicsekModel(gen, v, size, h, mode='Load'):
    alignmentData = []
    clusteringData = []
    startTime = time.time()
    periodTime = time.time()
    if h != 0:
        thetaHistory = deque(maxlen=abs(h))
    else:
        thetaHistory = []

    if mode == 'Load':
        particles = [LoadData('Positions8.csv'), LoadData('Angles8.csv')]
    else:
        particles = InitializeParticles(l, n)
        SaveData(particles[0], 'Positions8.csv')
        SaveData(particles[1], 'Angles8.csv')

    for i in range(gen):
        particles = UpdateParticles(particles[0], particles[1], rf, size, eta, dt, v, thetaHistory, h)
```

```
        if h > 0:
            thetaHistory.append(copy.deepcopy(particles[1]))
        alignC, clusterC = FindCoefficients(particles[0], particles[1], rf)
        alignmentData.append(alignC)
        clusteringData.append(clusterC)

        if i % 100 == 0:
            print(f'\nGeneration: {i} \n  > Periodtime: {time.time()-periodTime:3.0f} seconds')
            periodTime = time.time()

    print(f'\n Simulation time: {(time.time()-startTime)%60:3.0f} minutes')

    PlotFunction(LoadData('Positions8.csv'), particles[0], [alignmentData, clusteringData])


# Variables:
l = 10**2
n = 100
v = 1
dt = 1
eta = 0.1
rf = 1
gen = 10**4
h = 0

VicsekModel(gen, v, l, h, 'Create')
```