```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Mar  1 21:50:07 2020

@author: Mark Millonas

    Object oriented python code implementing artificial ant swarms.
    Rewrite with new implementation of the animation - March 1, 2020

    (C) 2020 Mark Millonas, SF Bay Area, CA, USA.
    email: millonasm@gmail.com


    Notes
    -----
    SwarmSim-1.0 uses a retangular lattice with 8 nearest neighbor edges and
    toroidal boundary conditions.

    References
    ----------
    Consult one or more of the following reference for details.

    - M. M. Millonas, A Connectionist Type Model of Self-Organized Foraging and
    Emergent Behavior in Ant Swarms, Journal of Theoretical Biology 159, 529
    (1992).

    - M. M. Millonas, Swarms, Phase Transitions, and Collective Intelligence,
    Proceedings of ALIFE III (C. G. Langton, ed.) Santa Fe Institute:
    Addison-Wesley (1993).

    - E. M. Rauch, M. M. Millonas and D. R. Chialvo, Pattern Formation and
    Functionality in Swarm Models, Physics Letters A, 207, 185 (1995).

    - D. R. Chialvo and M. M. Millonas, How Swarms Build Cognitive Maps , In:
    The NATO ASI series. Series F, Computer and system sciences 144, 439
    (1995).

"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time
from tqdm import tqdm


class Swarm():
    r"""Implement ant/pheromone (particle/field) dynamics on a lattice.

    Class Parameters
    ----------
    num_ants, num_steps, size_lat_x, size_lat_y : int
    beta, delta, eta, kappa : float
    edge_vectors: list of int

    Class Attributes
    ----------
    edge_vectors

    """
```

```
edge_vectors = [
        [-1,1], [0,1], [1,1], [1,0], [1,-1], [0,-1], [-1,-1], [-1,0]
        ]

"""
 Technical note: Retangular LATTICE STRUCTURE

Edge_vectors describe the connectivity on the lattice of choice
explanation of RECTANGULAR lattice edge vectors: (displayed geometrically
below for clarity) where the * symbol represents the starting vertex
        [-1, 1] [0, 1] [1, 1]
        [-1, 0]    *   [1, 0]
        [-1,-1] [0,-1] [1,-1]
the directions in the 'edge_vectors' array are listed in clockwise order
starting in the upper left (northwest) corner. so for example
edge_vectors[2] = [1,1], etc.

"""

def __init__(self, **kwargs):
    """Parse out object parameters """

    #initialize swarm paramaters
    self.num_ants = kwargs['num_ants']
    self.num_steps = kwargs['num_steps']

    # initialize lattice parameters
    self.size_lat_x = kwargs['size_lat_x'] # nodes in x direction
    self.size_lat_y = kwargs['size_lat_y'] # nodes in y direction
    self.num_edge = 8 # retangular lattice

    # initialize pheromone parameters
    self.beta = kwargs['beta'] # osmotropotaxic sensitivity
    self.delta = kwargs['delta'] # recriprocal sensory capacity
    self.eta = kwargs['eta'] # layed down per ant per timestep
    self.kappa = kwargs['kappa'] # decay percentage per timestep

    # initialize pheromone array sigma to zero everywhere
    self.sigma = np.zeros((self.size_lat_x, self.size_lat_y))

def randomize_ants(self):
    """Create ants and randomize starting locations on the lattice. """
    self.ants = []
    for i in range(0, self.num_ants):
        """ create initial ant swarm data structure """
        x = np.random.randint(self.size_lat_x) # ant x coordinant
        y = np.random.randint(self.size_lat_y) # ant y coordinant
        n = np.random.randint(self.num_edge) # ant direction index
        self.ants.append([x, y, n])

def swarm_sim(self):
    """Run animation and output data array.

    Returns
    -------
    sim_data : ndarray of float
    sim_vectors : ndarray of float

    """
    # initialize swarm data array
```

```
        sim_data = np.zeros(( self.num_steps, self.num_ants, 2), dtype=float)
        # initialize ant vector data array
        sim_vectors = np.zeros(( self.num_steps, self.num_ants, 2), dtype=float)
        for step in tqdm(range(self.num_steps)):
            """ simulate swarm dynamics for num_step iterations """
            self._update_swarm() # move ants
            self._update_pheromone() # update pheromone
            # write ant locations to sim_data
            gants = np.array(self.ants)
            sim_data[step, :, 0] = gants[:,0]
            sim_data[step, :, 1] = gants[:,1]
            # write ant orientations to sim_vectors
            for ant_index in range(self.num_ants):
                edge = np.array( self.edge_vectors[gants[ant_index,2]] )
                norm = np.sqrt( np.dot(edge,edge) )
                sim_vectors[step, ant_index, :] = edge/norm
        return sim_data, sim_vectors

    def _update_swarm(self):
        """Move each ant once according to behavioral rules. """
        for i in range(0, self.num_ants):
            """ make local choice and move each ant  """
            # adjust directional bias to the orientation of the ant
            orientation_bias = self._directional_bias(self.ants[i][2])
            # get pheromone bias vector from pheromone field
            pheromone_bias = self._local_pheromone_weights(i)
            # combine biases
            bias = np.multiply(orientation_bias, pheromone_bias)
            # chose the next direction (angle) to move ...
            angle_new = self._weighted_choice(bias)
            # and update the direction of the ant ...
            self.ants[i][2] = angle_new
            # and get the correponding edge vector
            change = self.edge_vectors[angle_new]
            # update the lattice location of the ant
            x_new = self.ants[i][0] + change[0]
            y_new = self.ants[i][1] + change[1]
            # apply toroidal boundary conditions
            self.ants[i][0], self.ants[i][1] = self._apply_bcs(x_new, y_new)

    def _update_pheromone(self):
        """Implement pheromone drop by ants, and decay over time. """
        # ants lay down pheromone
        for i in range(self.num_ants):
            self.sigma[ self.ants[i][0] ][ self.ants[i][1] ] += self.eta
        # attenuate pheromone
        self.sigma = np.multiply(1 - self.kappa, self.sigma)

    def _pheromone_weight(self, sigma):
        """From pheromone concentration return weight factor.

        Parameters
        ----------
            sigma : float

        Returns
        -------
            weight_factor : float

        Notes
        -----
```

```
        Describes the osmotropotaxic weighted stochastic response of an ant
        to the local pheromone field.

        """
        weight_factor = np.power(
            1.0 + sigma/(1 + self.delta * sigma), self.beta)
        return weight_factor

    def _weighted_choice(self, weights):
        """From array of relative weights returns random choice index

        Parameters:
            weights : list of float

        Returns
        -------
            choice_index : int

        """
        # initialize probability density function (pdf)
        pdf = np.zeros(len(weights))
        # integrate ...
        i, norm = 0, 0
        for weight in weights:
            norm += weight
            pdf[i] = norm
            i += 1
        # and normalize pdf
        pdf = np.divide(pdf, norm)
        # roll uniform random [0, 1]
        random_number = np.random.rand()
        # select choice index
        choice_index = 0
        while random_number >= pdf[choice_index]:
            choice_index += 1
        return choice_index

    def _directional_bias(self, i):
        """From direction index return array of directional bias weights.

        Parameters
        ----------
            i : int

        Returns
        -------
            new_bias_array : list of float

        """
        # left/right-symetric relative turn weights - increments of 45 degrees
        w0 = 1.0  # go straight
        w45 = 0.5 # turn left/right 45 degrees
        w90 = 0.1 # turn left/right 90 degrees
        w135 = 1.0/20.0 # turn left/right 135 degrees
        w180 = 1.0/40.0 # turn 180 degrees
        # bias array for default "north" i = 1 direction
        bias_array = np.array([w45, w0, w45, w90, w135, w180, w135, w90])
        # return direction bias array for the direction specificed by i
        new_bias_array = np.roll(bias_array, i - 1)
        return new_bias_array
```

```
    def _apply_bcs(self, i, j):
        """Apply toriodal boundary conditions on lattice index i and j.

        Parameters
        ----------
            i,j : int

        Returns
        -------
            i,j : int

        """
        # apply x toroidal boundary condition
        if i == -1:
            i = self.size_lat_x - 1
        elif i == self.size_lat_x:
            i = 0
        # apply y toroidal boundary condition
        if j == -1:
            j = self.size_lat_y - 1
        elif j == self.size_lat_y:
            j = 0
        return i, j

    def _local_pheromone_weights(self, index):
        """Return pheromone weight vector for indexed lattice neighbors.

        Parameters
        ----------
            index: int

        Returns
        -------
            weight_vector : list of int

        """
        sigma_local = np.zeros(8)
        # get lattice coordinates of ant labled by 'index'
        lattice_location  =  [ self.ants[index][0], self.ants[index][1] ]
        for i in range(0, self.num_edge):
            # get lattice index of neighbor
            local_x = lattice_location[0] + self.edge_vectors[i][0]
            local_y = lattice_location[1] + self.edge_vectors[i][1]
            local_x, local_y = self._apply_bcs(local_x, local_y)
            # get the pheromone on neighbor
            sigma_local[i] = self.sigma[local_x][local_y]
        weight_vector = self._pheromone_weight(sigma_local)
        return weight_vector


def main():
    """Simulate the swarm. """

    # Use line below to repeat a run - Otherwise comment it out
    np.random.seed(123456789)

    settings = {
        # swarm parameters
        'num_steps'   : 100,  # number of iterations until termination
        'num_ants'    : 100,  # number of ants on lattice
        # lattice parameters
```

```
    'size_lat_x' : 100, # number of nodes in the x direction
    'size_lat_y' : 100, # number of nodes in the y direction
    # pheromone parameters
    'beta'       : 3.5,   # osmotropotaxic sensitivity
    'delta'      : 0.2,   # recriprocal sensory capacity
    'eta'        : 0.2,   # amount layed down per ant per time step
    'kappa'      : 0.015, # decay percentagefraction per time step
    # graphical display settings
    'size_ant'   : 20,    # radius in pixels of ant/circles
    'ant_body'   : 0.5,   # head distance from body in lattice spacing
    'heads'      : True,   # set 'True' for plotting heads
    'color_ant'  : "red", # color of ants/circles
    'color_head' : "yellow", # color of head/circle
    'title'      : "An Ant Swarm",   # title on window
    'plot_style' : "dark_background", # background color
    # movie file settings
    'save_movie' : False, # 'True to write movie file of the simulation
    'movie_name' : "short_test.mp4", # name of movie file (must have .mp4)
    }


""" Perform swarm simulation """
s = Swarm(**settings)    # set up swarm (particle/field/lattice) object
s.randomize_ants()       # randomizestarting locations and directions
sim_data, sim_vectors = s.swarm_sim() # simulate swarm

"""
    The numpy.ndarray of int 'sim_data' contains the record of the full
    simulation. Output 'sim_data' is three dimensional with shape
    (num_steps, num_ants, 3) where the first component holds the time
    step, the second the ant index (labeling the individual ant in the
    swarm), and the third the ant state (x, y int lattice coordinates and
    direction index nt). Output 'sim_vectors' contains unit vectors for
    each and that indicates their orientation at each time step.

    Everything below is for the graphical display and/or movie file output
    of the animation so use this code, or what you prefer below.

"""
# if desired set potitions of the ant 'heads' for graphics
delta = settings['ant_body'] * sim_vectors
sim_data_displaced = sim_data + delta
# set up figure for scatter plot
plt.style.use(settings['plot_style'])
fig = plt.figure(figsize=(7, 7))
ax = fig.add_axes([0, 0, 1, 1], frameon=False)

# updater function for FuncAnimation
def update(i, sim_data, sim_data_displaced, settings):
    ax.clear()
    ax.set_xlim(0, settings['size_lat_x'] ), ax.set_xticks([])
    ax.set_ylim(0, settings['size_lat_y'] ), ax.set_yticks([])
    #plot heads first
    if settings['heads'] == True:
        ax.scatter(sim_data_displaced[i, :, 0],
                   sim_data_displaced[i, :, 1],
                   s = settings['size_ant']-12,
                   color = settings['color_head']
                   )
    # plot bodies after (over) heads
    ax.scatter(sim_data[i, :, 0], sim_data[i, :, 1],
```

```
                    s = settings['size_ant'],
                    color = settings['color_ant']
                    )
        return ax,

    # animate swarm simulation
    anim = FuncAnimation(fig, update,
                         fargs = (sim_data, sim_data_displaced, settings),
                         interval=2,
                         # init_func=init,
                         # blit=True,
                         save_count=settings['num_steps']
                         )
    plt.show()

    if settings['save_movie'] == True:
        anim.save(settings['movie_name'],
                  fps=30,
                  extra_args=['-vcodec', 'libx264']
                  )


    return anim # need this to keep the reference to object 'anim' around


if __name__== "__main__":
    outer_anim = main() # execute main
    """ uncomment the line below to write animation to movie file """
    # outer_anim.save('short_test.mp4',
    #                 fps=30,
    #                 extra_args=['-vcodec', 'libx264']
    #                 )
```