Joey Isola
CS 455.1001
Mar 7, 2024

Project 1: MSN Flocking Formation Control

Preliminaries

Parameters

Below are the parameter constants stored as global variables needed to run each program. The first block details how many nodes will be generated, the area they will be randomly placed in, and the interaction range.

Epsilon, bump_h, a, b, and c are all constants needed for equations.

Each program simulates 540 iterations and captures 6 scatter plots dispersed evenly throughout the simulation.

Amplitude, frequency, and phaseShift are constants for manipulating the gamma agent on case 2 and 3 to generate sine waves and circles.

c1 and c2 are constants for calculating u_i, used mainly to speed up the simulation so that the flocking nodes can keep up with the gamma agent when it moves faster. With the current parameters, each simulation takes roughly 2 minutes to complete.

```
Global Variables
"""
#Parameters
nodes = 100
x_length = 150
y_length = 150
desiredDistance = 15
scalingFactor = 1.2
interactionRange = (desiredDistance * scalingFactor)

epsilon = 0.1
bump_h = 0.1
delta_t = 0.009

iterations = 540
capture_iteration = iterations / 6

a = 3
b = 3
c = np.abs(a-b)/np.sqrt(4 * a * b)

amplitude = 100
frequency = 0.025
phaseShift = 0

c1_alpha = 100
c2_alpha = 2 * np.sqrt(c1_alpha)

c1_gamma = 70
c2_gamma = 2 * np.sqrt(c1_gamma)
```

Numpy Arrays

The arrays below keep track data necessary for kinematic equations and data logging after the simulation is complete.

```
#global numpy arrays -> collect data
x_coordinates = np.random.uniform(0, x_length, nodes)
y_coordinates = np.random.uniform(0, y_length, nodes)

adjacencyMatrix = np.zeros((nodes, nodes), dtype=int)

velocity_x = np.zeros((nodes), dtype=np.float64)
velocity_y = np.zeros((nodes), dtype=np.float64)

logPositions_x = np.zeros((nodes, iterations), dtype=np.float32)
logPositions_y = np.zeros((nodes, iterations), dtype=np.float32)

logVelocitity_x = np.zeros((nodes, iterations), dtype=np.float32)
logVelocitity_y = np.zeros((nodes, iterations), dtype=np.float32)

logVelocitityMagnitude = np.zeros((nodes, iterations), dtype=np.float32)

logConnectivity = np.zeros((iterations), dtype=np.float32)



gammaStart = np.array((50, 50), dtype=np.float32)
dynamicRendezvous = np.array((50 + amplitude, 50), dtype=np.float32)
dynamicRendezvousVelocity = np.array((0, 0), dtype=np.float32)

logGamma_x = np.zeros((iterations), dtype=np.float32)
logGamma_y = np.zeros((iterations), dtype=np.float32)
logGamma_velocityMagnitude = np.zeros((iterations), dtype=np.float32)

centerOfMass_x = np.zeros((iterations), dtype=np.float32)
centerOfMass_y = np.zeros((iterations), dtype=np.float32)
```

Equations

The equations are fundamental to achieving the flocking of algorithm 1, which is used and improved in algorithm 2. All algorithms below are implemented from [1] *Flocking for multi-agent dynamic systems: algorithms and theory.*

Euclidean Norm is a simple magnitude function, sigma norm takes a euclidean norm as input and outputs a norm that is differentiable at z = 0. Sigma_1_gradient is a special case of sigma gradient, taking 1 as a constant instead of epsilon.

The bump function is a piece-wise function that takes a sigma norm and constant: bump_h to produce smooth potential, necessary for the spatial adjacency matrix.

The phi_action function is used to produce a pairwise potential, which vanishes as z approaches the sigma norm of the interaction range. Uses phi() heper function as an uneven sigmoidal function.

```
"""
Math Functions as described in "Flocking for Multi-Agent Dynamic Systems:
Algorithms and Theory"
"""
#calculate magnitude of point1 and point2
def euclidean_norm(x1, x2, y1, y2):
    return np.sqrt(((x2 - x1) * (x2 - x1)) + ((y2 - y1) * (y2 - y1)))

#Sigma norm
def sigma_norm(z):
    return (np.sqrt(1 + (epsilon * (z ** 2))) - 1) / epsilon


def sigma_1_gradient(z):
    return z / (np.sqrt(1 + (z**2)))

#Bump function
#bump_h is h (defined in parameters)
def bump_function(z):
    if z < bump_h and z >= 0:
        return 1
    elif z >= bump_h and z <= 1:
        temp = (z - bump_h) / (1 - bump_h)
        return ((1/2) * (1 + np.cos(np.pi * (temp))))
    else:
        return 0

#uneven sigmoidal
def phi(z):
    return (1/2) * (((a + b) * sigma_1_gradient(z + c)) + (a - b))

#Pairwise Potential action function -> vanish for z >= sigma_norm of interaction range
def phi_action(z):
    norm_r = sigma_norm(interactionRange)
    norm_d = sigma_norm(desiredDistance)
    return ((bump_function(z / norm_r)) * (phi(z - norm_d)) )
```

Matrix Functions

A_i_j() function takes 2 neighboring nodes and produces a spatial adjacency matrix coefficeint, necessary for generating the velocity consensus term in producing an acceleration vector.
n_i_j() function takes 2 neighboring nodes and produces a gradient vector <x, y> of the neighboring nodes' sigma norm. This gradient vector is necessary for calculating the gradient based term.
getNeighbors() function takes advantage of numpy arrays to quickly find the neighbors of a node given that a complete adjacency matrix is constructed.

```python
#spatial adjacency matrix of i and j in terms of q (position)
#returns list
def a_i_j(i, j):
    norm = sigma_norm(euclidean_norm(x_coordinates[i], x_coordinates[j], y_coordinates[i], y_coordinates[j]))
    norm_r = sigma_norm(interactionRange)
    return bump_function(norm / norm_r)

#sigma norm gradient of position i an dj
def n_i_j(i, j):
    diff = [x_coordinates[j] - x_coordinates[i], y_coordinates[j] - y_coordinates[i]]
    # euclidean_norm = euclidean_norm(x_coordinates[i], x_coordinates[j], y_coordinates[i], y_coordinates[j])
    denom = np.sqrt(1 + (epsilon * (euclidean_norm(x_coordinates[i], x_coordinates[j], y_coordinates[i], y_coordinates[j]))**2))
    return (diff) / (denom)

def getNeighbors(node):
    return (np.nonzero(adjacencyMatrix[node])[0])
```

Alpha Agent

The function below brings all the previous equations together, producing a gradient based term and velocity consesus term, combining them as a single acceleration vector necessary for flocking. The alpha agent provides as algorithm 1's sole objective and algorithm 2's primary objective: forming an alpha lattice.

Constants c1_alpha and c2_alpha are used in case 3 and case 4 to speed up the simulation. Without these constants, the simulation took over 3000 iterations to produce quality data, taking close to 20 minutes to complete. These alpha coefficients must be greater than the gamma coefficients for algorithm 2 to work correctly, since the alpha agent is the primary objective and gamma agent is the secondary objective.

```python
def u_i_alpha(node):
    gradientSum = np.array([0, 0], dtype=np.float32)
    consensusSum = np.array([0, 0], dtype=np.float32)
    neighbors = getNeighbors(node)
    # print(neighbors)
    for neighbor in neighbors:
        v_1 = phi_action(sigma_norm(euclidean_norm(x_coordinates[node], x_coordinates[neighbor], y_coordinates[node], y_coordinates[neighbor])))
        gradientSum += n_i_j(node, neighbor) * v_1

        spat = a_i_j(node, neighbor)
        diff_x =np.array([velocity_x[neighbor] - velocity_x[node], velocity_y[neighbor] - velocity_y[node]], dtype=np.float32)
        consensusSum += (diff_x * spat)
    return (c1_alpha *gradientSum) + (c2_alpha*consensusSum)
```

$$u_i^{\alpha} = \underbrace{\sum_{j \in N_i} \phi_{\alpha}(\|q_j - q_i\|_{\sigma}) \mathbf{n}_{ij}}_{\text{gradient-based term}} + \underbrace{\sum_{j \in N_i} a_{ij}(q)(p_j - p_i)}_{\text{consensus term}} \quad (23)$$

Gamma Agent

Calculating the gamma agent for a given node as an acceleration vector requires taking the difference between the given node's and rendezvous point's position and velocity vectors. Constant coefficients c1_gamma and c2_gamma are for speeding up the simulation, where they must be greater than zero and less than the alpha agent's coefficients.

Calculating the gamma agent's velocity is different for each simulation. The picture below shows simulation 4, where the gamma agent moves in a continuous circle, where their instantaneous velocities are calculated by taking the derivative of both the x and y component of their position functions. As shown below, the derivative of a sine function gives a cosine function, and the

derivative of a cosine function gives a negative sine function. Simulations 2 and 3 have different functions for calculating velocity, and Simulation 1 does not have any gamma agent.
The function u_i_() adds the positive alpha agent acceleration vector to the negative gamma agent acceleration vector, providing a node's total acceleration vector.

```
def u_i_gamma(node, v_y, v_x):
    pDiff = np.array([0, 0], dtype=np.float32)
    vDiff = np.array([0, 0], dtype=np.float32)

    pDiff[0] = x_coordinates[node] - dynamicRendezvous[0]
    pDiff[1] = y_coordinates[node] - dynamicRendezvous[1]

    vDiff[0] = velocity_x[node] - v_x
    vDiff[1] = velocity_y[node] - v_y
    return -1 *(c1_gamma) * pDiff -  c2_gamma * vDiff

def u_i_(node, v_y, v_x):
    return u_i_alpha(node) + u_i_gamma(node, v_y, v_x)

#derivative of sin function
def gammaVelocity_y(chain, dt, multiplier, amp):
    return amp * chain * multiplier * np.cos(chain * dt + phaseShift)

def gammaVelocity_x(chain, dt, multiplier ,amp):
    return amp * chain * multiplier *(-1) * np.sin(chain * dt + phaseShift)
```

$$u_i = \sum_{j \in N_i} \phi_\alpha(\|q_j - q_i\|_\sigma)\mathbf{n}_{ij} + \sum_{j \in N_i} a_{ij}(q)(p_j - p_i) + f_i^\gamma(q_i, p_i, q_r, p_r)$$

Plotting Points
The function below takes a boolean variable and index integer, producing the scatter plots and adjacency matrix necessary for each simulation. To save CPU cycles, the function only adds points to the matplotlib plot when the boolean parameter is set to true, but always fills the adjacency matrix, as it is necessary for alpha agent calculations on each iteration.
Simulations 3 and 4 plot their dynamic gamma agents as well as their gamma agents previous trajectory, simulation 2 plots the coordinates of its static gamma agent, and simulation 1 does not plot any gamma agent.

```
#plots points based on x_coordinates and y_coordinates
#Also updates adjacency matrix
def plot_points(plot, iteration):
    #Add data from  coordinates
    if (plot == True):
            plt.plot(logGamma_x[0:iteration], logGamma_y[0:iteration])
            plt.plot(dynamicRendezvous[0], dynamicRendezvous[1], '^', color='purple')
            for x, y in zip(x_coordinates, y_coordinates):
                plt.plot(x, y, 'ro')

    # adjacencyMatrix = np.zeros((nodes, nodes), dtype=int)
    adjacencyMatrix.fill(0)
    for i in range(0, nodes):
        for j in range(i, nodes):
            mag = euclidean_norm(x_coordinates[i], x_coordinates[j], y_coordinates[i], y_coordinates[j])
            if mag <= interactionRange:
                adjacencyMatrix[i][j] = 1
                adjacencyMatrix[j][i] = 1

                #add line to plot
                if ((j != i) and (plot == True)):
                    plt.plot([x_coordinates[i], x_coordinates[j]], [y_coordinates[i], y_coordinates[j]], 'b-')
        adjacencyMatrix[i][i] = 0
```

Update Kinematics

The update_kinematics() function performs all the necessary math and physics calculations necessary to move each flocking node, move the dynamic gamma agent, and log data such as position, velocity, connectivity, and center of mass position where each is necessary.

The first screenshot below shows the first iteration case, where the iteration index equals zero. In this special case, the function takes data and plots all points without updating the position of any of the moving pieces. This case is for initializing and showing the positon of each starting point.

The second screenshot shows an else statement, which executes for every iteration besides the initial case. First, the dynamic gamma agent's velocity function is calculated as a function of time, then the gamma agent's position is updated. The gamma agent's position vector is calculated in the form of amplitude * sin(2πf * dt + phase) and cos(2πf * dt + phase) to achieve a circle in simulation 4. Simulation 3 sets the y component to a sign wave, leaving the x component a constant speed.

Next the functions iterates through every node, computing each node's respective acceleration vector and updating their current position and velocity vectors. Each node's corresponding position and velocity is logged, as well as center of mass and connectivity.

The third screenshot shows the end of the update_kinematics() function, logging the average x and y component of center of mass. Additionally, the function checks if the current iteration index is marked to show a scatter plot, plotting points with a True parameter if True, and False if not. This is so the global adjacency matrix is updated every iteration, while plotting points on matplotlib is reserved for certain conditions.

```python
def update_kinematics():
    time = 0
    for i in range(0, iterations):

        sumMass = np.array((0, 0) , dtype=np.float32)
        #Case: first iteration -> don't update anything
        if i == 0 :
            plot_points(True, i)
            plt.title(f'Node Position at {i * delta_t} seconds')
            plt.show()
            for j in range(nodes):

                logPositions_x[j, i] = x_coordinates[j]
                logPositions_y[j, i] = y_coordinates[j]

                logVelocitity_x[j, i] = 0
                logVelocitity_y[j, i] = 0
                logVelocitityMagnitude[j, i] = 0
                logConnectivity[i] = (1 / nodes) * np.linalg.matrix_rank(adjacencyMatrix)

                logGamma_x[i] = dynamicRendezvous[0]
                logGamma_y[i] = dynamicRendezvous[1]

                sumMass[0] += x_coordinates[j]
                sumMass[1] += y_coordinates[j]
```

```python
#case: each next iteration -> update velocity, position
else :
    time += delta_t
    gammaVelocity = gammaVelocity_y((2 * np.pi * frequency), (i/5) , (1/5) ,  amplitude)

    dynamicRendezvous[0] = gammaStart[0] + (i/5)
    dynamicRendezvous[1] = gammaStart[1] + amplitude * np.sin(2 * np.pi * frequency * (i/5)  + phaseShift)

    logGamma_x[i] = dynamicRendezvous[0]
    logGamma_y[i] = dynamicRendezvous[1]


    for j in range(nodes):

        u = u_i_(j, gammaVelocity, (1/5))
        tmpVelocity = [velocity_x[j], velocity_y[j]]
        tmpPosition = [x_coordinates[j], y_coordinates[j]]

        v = tmpVelocity + (u * delta_t)
        p = tmpPosition + (v * delta_t) + ((1/2) * (delta_t ** 2) * u)

        [x_coordinates[j], y_coordinates[j]] = p
        [velocity_x[j], velocity_y[j]] = v
        logVelocitityMagnitude[j, i] = euclidean_norm(0, v[0], 0, v[1])


        logPositions_x[j, i] = x_coordinates[j]
        logPositions_y[j, i] = y_coordinates[j]

        logConnectivity[i] = (1 / nodes) * np.linalg.matrix_rank(adjacencyMatrix)

        sumMass[0] += x_coordinates[j]
        sumMass[1] += y_coordinates[j]
```

```python
centerOfMass_x[i] = sumMass[0] / nodes
centerOfMass_y[i] = sumMass[1] / nodes
if (i != 0 and i % capture_iteration == 0):
    plot_points(True, i)
    plt.title(f'Node Position at {i * delta_t} seconds')
    plt.show()
elif (i != 0 and i % capture_iteration != 0):
    plot_points(False, i)
```

Data Collection Plots

The last four functions plot corresponding data that each datalog vector has collected over the course of the simulation. Trajectory is plotted using the record of each node's position over time, velocity is plotted showing the magnitude of each node's velocity over time. Magnitude is collected using a euclidean norm in updateKinematics(). Connectivity is calculated using the equation C = (1/(num_nodes))*rank(A), where A is the adjacency matrix. Connectivity is calculated in upate_kinematics(). Lastly, the center of mass over time of the flock is plotted compated to the gamma agent's position over time.

```
def plotTrajectory():
    for node in range(nodes):
        plt.plot(logPositions_x[node], logPositions_y[node])
    plt.title(f'Trajectory of Nodes over {iterations * delta_t} seconds')
    plt.show()

def plotVelocity():
    for node in range(nodes):
        plt.plot(logVelocitityMagnitude[node])
    plt.title(f'Velocity magnitude of Nodes over {iterations * delta_t} seconds')
    plt.show()
def plotConnectivity():
    plt.plot(logConnectivity)
    plt.title(f'Connectivity of Nodes over {iterations * delta_t} seconds')
    plt.show()
def plotCMass():
    plt.plot(centerOfMass_x, centerOfMass_y)
    plt.plot(logGamma_x, logGamma_y)
    plt.title(f'Center of Mass of Nodes over {iterations * delta_t} seconds')
    plt.show()
```

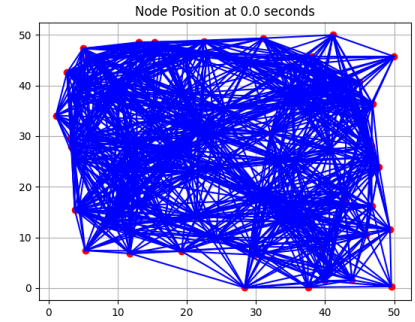
Main Function

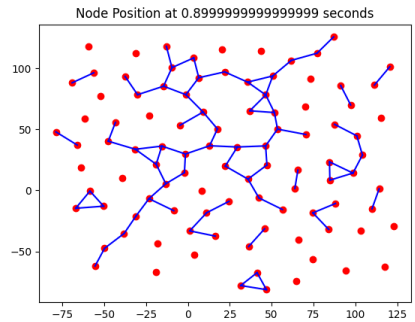
The main function is located at the bottom of each simulation's python file. First update_kinematics() is called, which runs the simulation, captures 6 different positions in time, and logs data. Each data logging function is called after update_kinematics is executed.

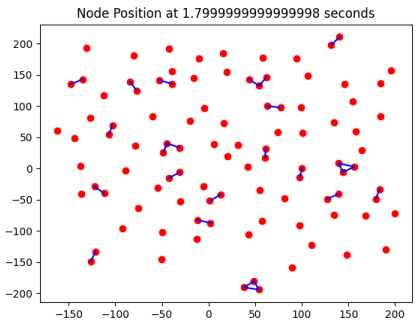
```
"""
Main
"""
plt.grid(True)
update_kinematics()
plotTrajectory()
plotVelocity()
plotConnectivity()
plotCMass()
```

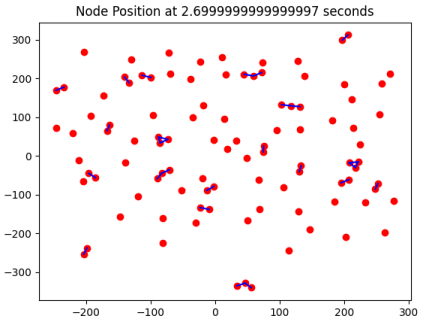
Data

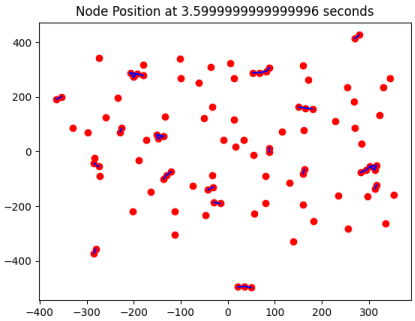
Case 1: Implement Algorithm 1 (MSN Fragmentation)

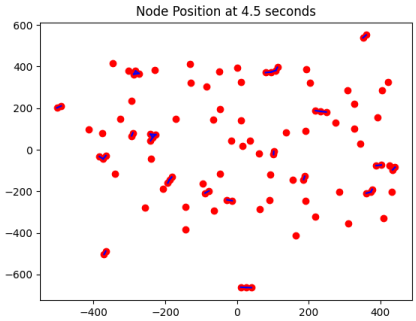


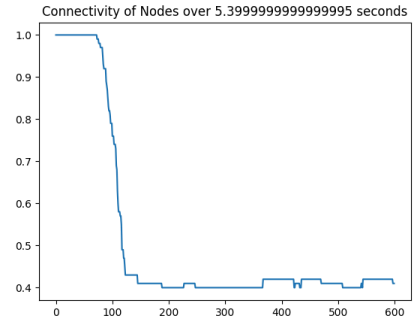


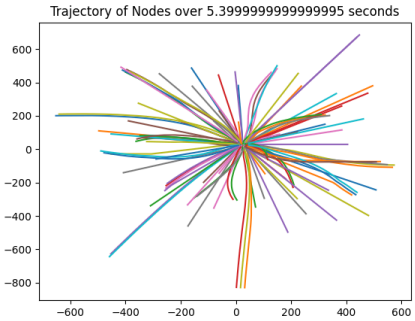


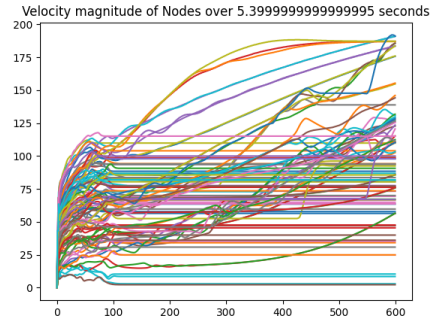






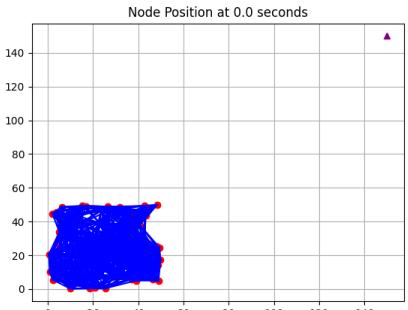


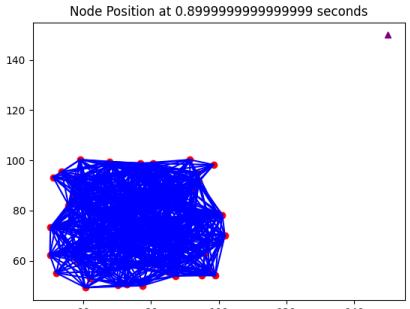


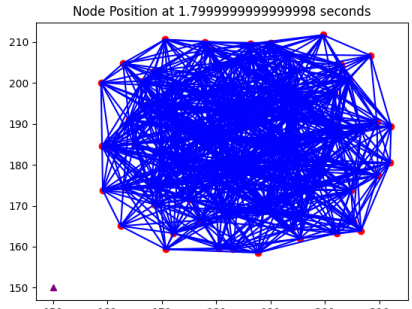


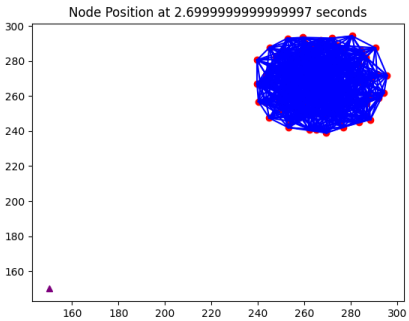
Data

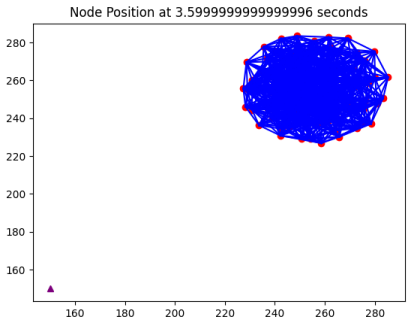
Case 2. Implement Algorithm 2 (MSN Quasi-Lattice Formation) with static target

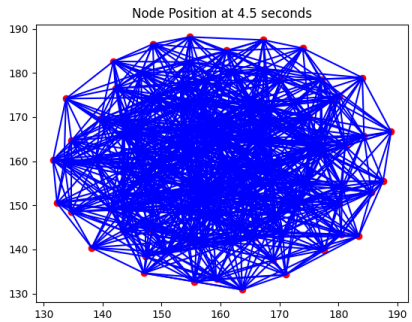


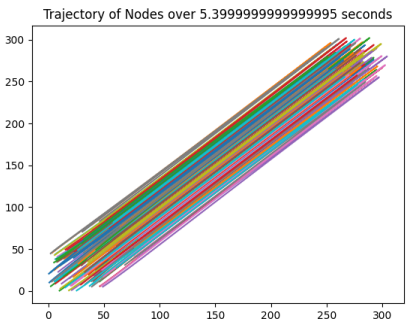


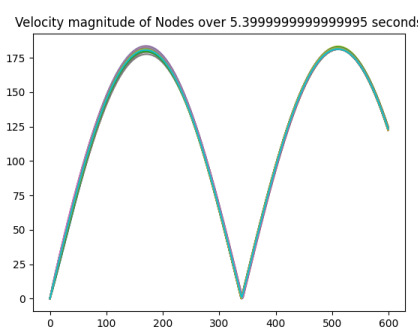


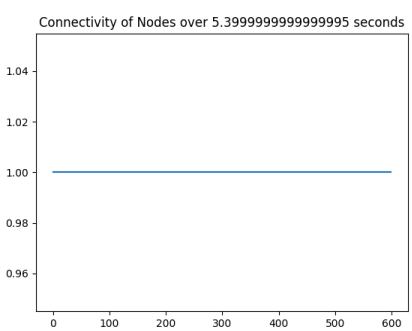






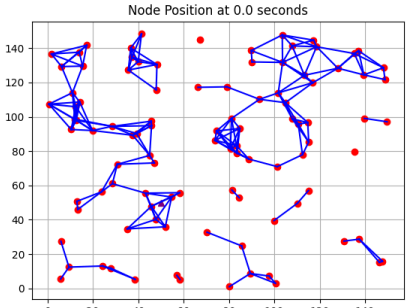


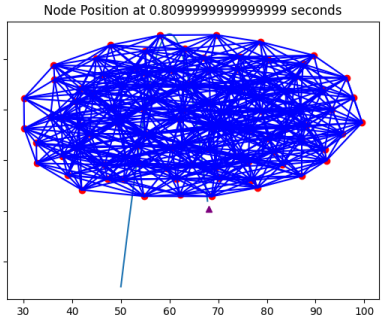


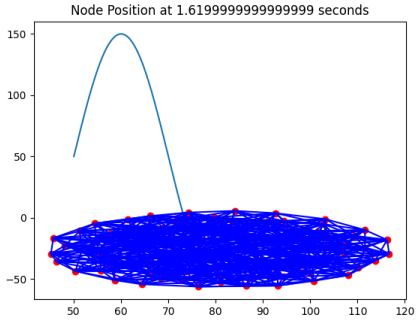


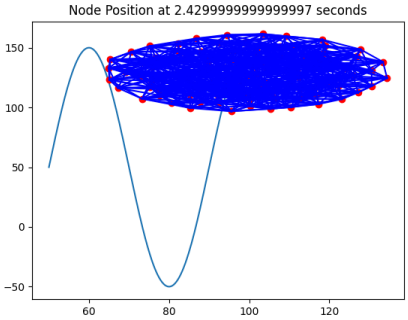
Data

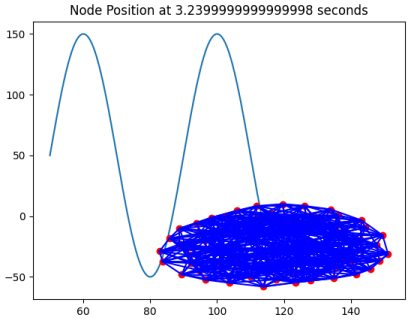
Case 3. Implement Algorithm 2 (MSN Quasi-Lattice Formation) with dynamic target

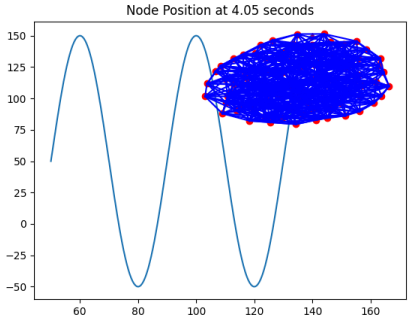


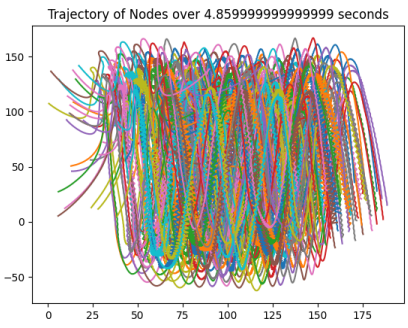


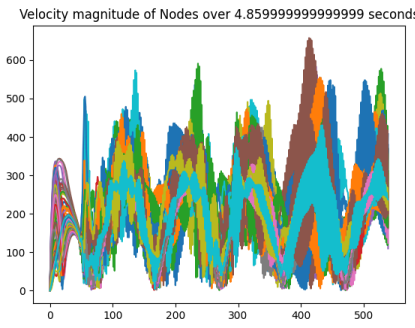


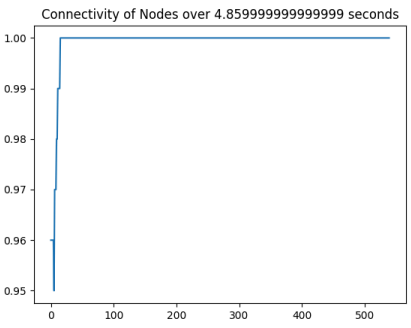


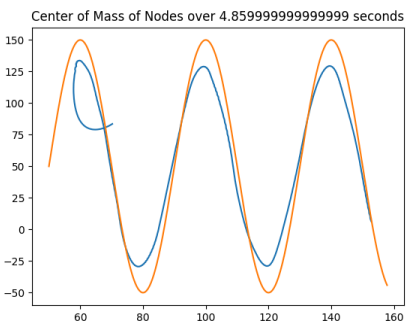






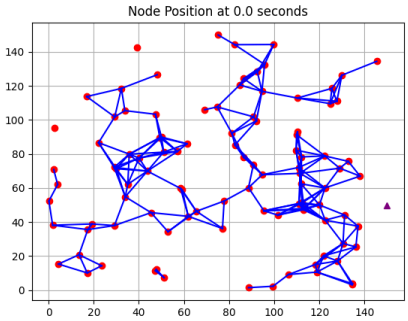


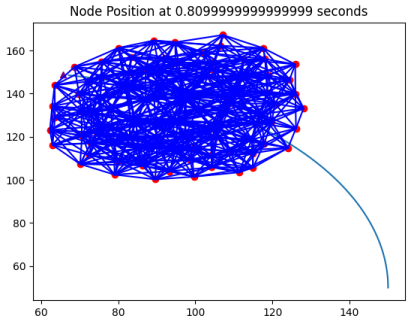


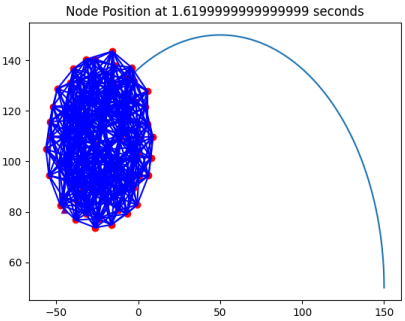


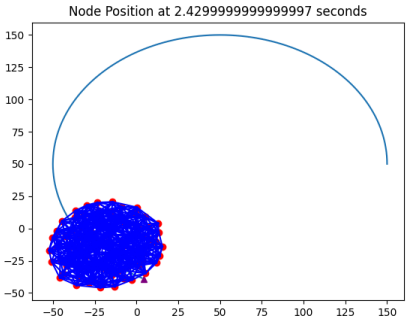
Data

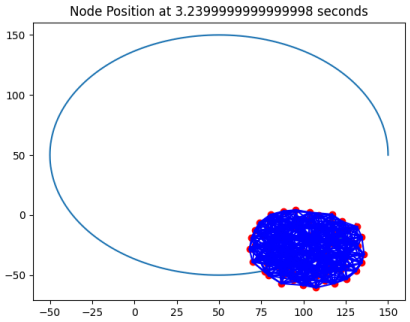
Case 4. Implement Algorithm 2 (MSN Quasi-Lattice Formation) with dynamic target

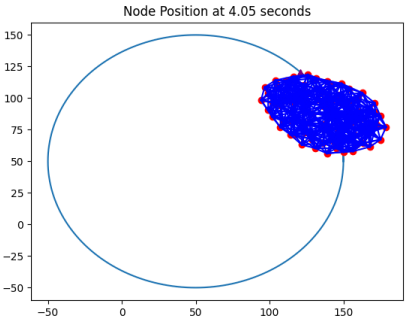


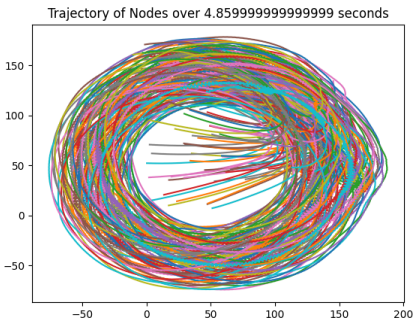


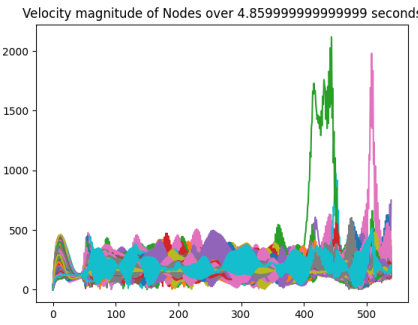


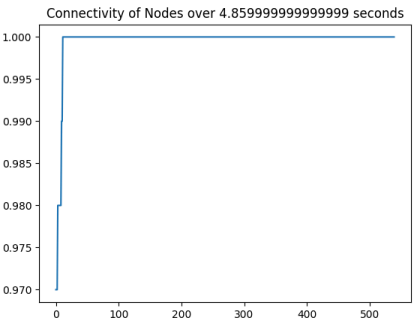


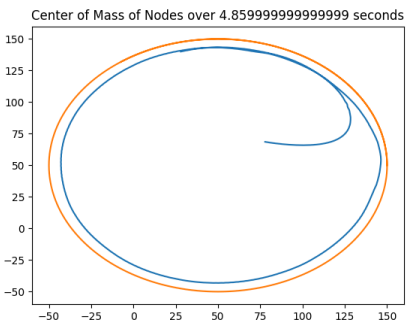












Conclusion

In Case 1, regular fragmentation is apparent as nodes continue to move further apart from each other. The alpha coefficients were set to 50 with no gamma coefficient to speed up the simulation, as using no alpha coefficients while keeping the other parameters constant made connectivity drop to 0.7 instead of 0.4. Using algorithm 1 was much faster than the other 3 cases, I suspect it is because the acceleration vector has to iterate through each node's neighbors, and in cases with regular fragmentation, nodes tend to have less neighbors. The trajectory graph gives the best view of fragmentation, as each node travels away from a common point. The velocity graph shows how fragmentation creates a velocity mismatch between nodes, as some nodes move at 200 units per iteration, where others are moving close to 0 units per iteration.

In Case 2, algorithm 2 becomes apparent as connectivity remains at 1.00 throughout the simulation. No coefficients were used in simulation 2, since adding high alpha and gamma coefficients seemed give a poor visualization of each node's movement. As seen in the trajectory graph, the flock oscillated between the point (0 , 0) and (300 , 300). This is to be expected as the gamma acceleration vector decreases in magnitude as nodes approach their target. Given that each node starts at a velocity magnitude of zero, and reaches peak speed once they approach their target, it would follow that it would take the same amount of time and distance to slow the node back to velocity magnitude of zero. The Velocity graph shows this point very well.

In Case 3, the nodes start in a random net of 150 square units, meaning 100 nodes with an interaction range of 18 units will start out with much less connectivity than the previous two cases of 50 square units. In examining the connectivity graph, it becomes apparent that the quasi-alpha lattice becomes fully connected by the 50th iteration, which would have been in between the first and second data catpures. The trajectory graph shows that the frequency of the gamma agent's sine wave was a bit too narrow, as the width of the flock is about the same size as the sine wave's period. This is fine for flocking, but it makes the trajectory graph a bit unreadable. In examining the center of mass and velocity magnitude graphps, it becomes apparent that although simulation 2 kept its connectivity at a perfect 1.00, the difference of each node's velocity in the flock spikes. The center of mass graph shows that at each valley and peak of the sine wave, the flock's sine wave has a smaller amplitude than the gamma agent. This is probably due to the fact that the gamma agent's y component of velocity reverses instantaneously.

In Case 4, the nodes start in the same spread that they do in Case 3, gradually moving in a counterclockwise direction. In examining the center of mass compared to the gamma agent's position plot, it is apparent that the flock's circle was of a smaller radius than the gamma agent's. This is most likely due to the fact that the entirety of the flock starts inside of the gamma agent's circle added to the fact that the gamma agent's velocity was just great enough to keep the flock following it without oscillating. In experimenting with different parameters in Case 4, the greater that the gamma agent's velocity is around the circle is, the smaller the radius of the

flock's center of mass plot will be. In terms of connectivity, the flock seemingly stays at 1.00 after around iteration 20; however, there are two outlier velocity spikes in the velocity graph. It seems that on two separate occasions, the lattice must have become too irregular, leaving a node to become seprated from the flock. This is the best explanation for the velocity spikes, as a node would only get to four times the rest of the flock's velocity if it were far away from the gamma agent. In both scenarios, the nodes quickly reattached themselves to the flock, showing the effectiveness of algorithm 2.

References

[1] R. Olfati-Saber, "Flocking for Multi-Agent Dynamic Systems: Algorithms and Theory," in IEEE Transactions on Automatic Control, vol. 51, no. 3, pp. 401-420, March 2006.