Queen's University, Kingston

APSC 200/293: Engineering Design & Practice II

Mathematics & Engineering Section

Group Formation in Multi-Agent Systems

App Guide

Instructor:
Bahman Gharesifard
bahman.gharesifard@queensu.ca
Scott Kyle
scott.kyle@queensu.ca

Technical Teaching Assistants:
Bryony Schonewille
14bhs1@queensu.ca
Hugh Corley
hugh.corley@queensu.ca
Ian Hogeboom-Burr
15ijhb@queensu.ca

Teaching Assistants:
Eve Stiles
ecs3@queensu.ca
Alex Everitt
aje4@queensu.ca
Garrett Richardson
gsr@queensu.ca
Michaela Wiederick
mrw140@queensu.ca
Danielle Moffatt
dlm9@queensu.ca

Revision	Date	Prepared by	Revision History
001	October 15, 2017	Scott Kyle & Fernando Camacho Cadena	APSC 200 P2 17-18
002	October 16, 2018	Scott Kyle & Ian Hogeboom-Burr	APSC 200 P2 18-19

October 16, 2018

Contents

1 Introduction

Multi-agent systems are becoming prevalent in the modern world. From self-driving cars to search and rescue drones, independent communication between agents within systems plays a key role in many applications. In your projects, you will use this premise of agents communicating with each other in a decentralized manner to develop a real-world application whose mathematical dynamics can be simulated and evaluated using MATLAB. This document will introduce you to a few common algorithms that may be applicable to your project and will provide apps that will help you visualize the performance of these algorithms. These apps can be opened using MATLAB's App Editor. Every time you run an app to completion, it will produce an EXCEL file called "agentData.xlsx" containing positional data for each agent. This can then be used for further performance analysis within EXCEL or to test the algorithms on the course's robots. The algorithms will also create a .mat file called "agentData.mat" which will be found in your MATLAB folder. You must have MATLAB 2018 or later to run these apps. Instructions for downloading the newest version of MATLAB for free can be found by visiting this web page.

2 Formation Algorithm

The formation algorithm is designed to make a system of agents converge to the same position. This algorithm could be used in instances where agents must meet at a point or surround a target.

2.1 Formation Consensus Dynamics

The formation algorithm is essentially an averaging function that takes in agent position and returns agent velocity. The continuous-time dynamics of the system are described by:

$$\dot{\vec{q}} = -L\vec{q}$$

Where $\vec{q}$ is an $N \times 2$ position vector containing x and y position data for all agents in the system and N is the number of agents in the system. $\vec{q}_i \in \mathbb{R}^2$ is the ith row of $\vec{q}$ and represents the position of the ith agent. L is called the Laplacian matrix for the system. L is an $N \times N$ matrix with the property:

$$L\begin{bmatrix}\alpha\\ \alpha\\ \vdots\\ \alpha\end{bmatrix} = \begin{bmatrix}0\\ 0\\ \vdots\\ 0\end{bmatrix} \quad \forall \alpha \in \mathbb{R}$$

This implies that $\dot{\vec{q}} = [0, 0, \ldots, 0]^T$ if $\vec{q}_i = \vec{q}_j \ \ \forall i, j \in [1, N]$.

L is a function of the adjacency matrix A for the system, and is given by:

$$L = D - A$$

The adjacency matrix, A, is an $N \times N$ matrix where:

$$A(i,j) = \begin{cases} 1 & \text{if agent } i \text{ receives communication data fom agent } j \\ 0 & \text{if agent } i \text{ does not receive communication data from agent } j \end{cases}$$

D is called the degree matrix of the system. D is a diagonal $N \times N$ matrix given by:

$$D(i,i) = \sum_{j=1}^{N} A(i,j)$$

The system then updates position in discrete time as follows:

$$\vec{q}(t+1) = \vec{q}(t) - L\vec{q}(t)\Delta t$$

Where $\Delta t \in \mathbb{R}_{>0}$ is a time step determined within the algorithm to ensure convergence of the system.

NOTE: A and L will be symmetric matrices in the default algorithms, as the communication lines between agents go both ways. If your situation has unidirectional communication lines, these matrices will not be symmetric.

2.1.1 Introducing Offsets

Offsets are differences between an agent's final position and the consensus position of the system. Instead of agents moving to an average value in a decentralized way, setting an offset allows you to set where individual agents will move (with the network average as the origin). For instance, if the consensus position of the system is (1,2) and an agent has an x offset of 1 and a y offset of -2, the agent will converge to the point (2,0). The only change in the system dynamics when offsets are introduced is within the position update step:

$$\vec{q}(t+1) = \vec{q}(t) - L\left[\vec{q}(t) - G\right]\Delta t$$

Where G is the $N \times 2$ offset vector containing the x and y offsets for every agent.

2.1.2 Introducing Delays

Delays are time steps where agents are idle before responding to changes in the system and moving to a new position. If an agent has an infinite delay, it is referred to as a "stubborn" agent, and is described by $\dot{\vec{q}}_i(t) = 0, \forall\, t \in \mathbb{Z}_{\geq 0}$. Otherwise, agent i will update $\tau_i \in \mathbb{Z}_{\geq 0}$ time-steps late when responding to a movement signal. We now have:

$$\vec{q}(t+1) = \begin{bmatrix} \vec{q_1}(t-\tau_1) \\ \vec{q_2}(t-\tau_2) \\ \vdots \\ \vec{q_N}(t-\tau_N) \end{bmatrix} - L\left[\vec{q}(t) - G\right]\Delta t$$

2.2 Using the App

The formation app uses the consensus dynamics to update the positions of your system of agents and plots these updated positions at every time iteration. Figure 1 shows a screenshot of the app. **Plot** displays initial positions, **Play** commences the algorithm, and **Stop** halts the algorithm. If you want your agent data to be exported to EXCEL, wait until the plot has reached the final iteration (as set by the user in the app) and do not press **Stop** prior to this. You have control over many aspects of the dynamics:

(i) *Allowing Offset*: This check-box enables x and y offsets to each agent as desired. For your project, you may design a cost function which makes agents with random initial positions reach a designated formation using offsets in an optimized way.

(ii) *Picking a Graph Type*: You can pick from four different types of graphs: cyclic, path, switching and custom graphs. These graph types alter the adjacency matrix used to represent communication between agents. A cyclic graph is a directed graph that connects a number of agents in a closed chain (agent 2 connects to agent 1 and agent 3, etc.). A path graph is similar but doesn't make a cycle (i.e. the first agent and the last agent are not connected). The app also allows you to create a custom graph, where you may define your own adjacency matrix in the MATLAB Command Window (make sure that it is the correct size for the number of agents in your network). Similarly, you can have a switching topology where the algorithm alternates between a series of adjacency matrices by selecting the switching graph option. You can choose how many switching graphs you would like, and then denote them as:

```
A1 = [...;...;...], A2 = [...;...;...], ...
```

in the Command Window. If you no longer want MATLAB to prompt you to do this when you choose a custom or switching graph type, you can delete the code in the GraphTypeDropDown-Changed callback function.

(iii) *Introducing Delays*: You can simulate delays in communication by setting delays for individual agents within the app. This makes selected agents update τ iterations late, where τ denotes the delay for that agent. To create a stubborn agent, enter "Inf" in the delay edit field.

(iv) *Distance from Average Plot*: This allows you to track the movement of agents over time

in relation to the true consensus value of the network.

(v) *Arena Plot*: This plot shows the positions of agents within space, as well as the path each agent has followed. Black lines are drawn to represent communication channels between agents. Stubborn agents are cyan and filled in, while all other agents are coloured with blank faces. The path history of each agent matches the colour of the agent, as well as the agent's data in the second plot.

Note: Within the PlayButtonPushed callback function, there are a few sections designated for adding complexity that is unique to your application. You may add a cost function that optimizes different aspects of the consensus dynamics (e.g. movement, communication, aspects particular to your application). Your team can also add perturbations to the dynamics to simulate different environmental activity which could be present in your application. Another area of openness is restricting movement due to power ratings (i.e. your agents' locations are updated, but due to weight, friction and battery conservation, your agents can only accelerate enough to move a smaller distance). Use your creativity to design a realistic model for your application!

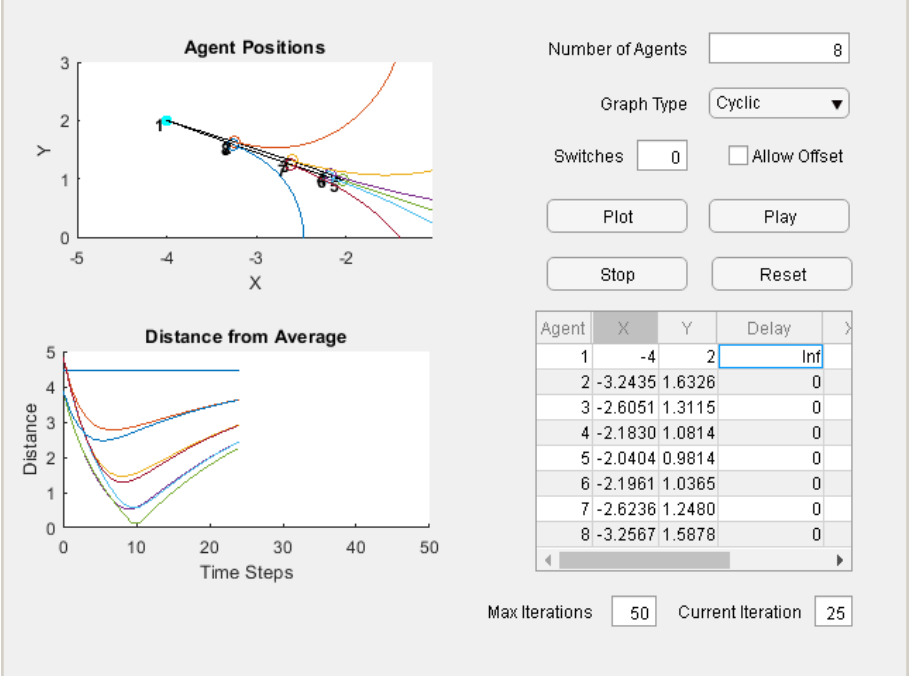

Figure 1: The formation app in use

2.3 Gearing Towards Your Application: Example

Consider a situation where you are doing research on coral bleaching in a shallow sea. You have 5 submersibles sampling coral around several kilometers of the surrounding area and now their sample containers are full. Instead of wasting time and resources by having each agent return all the way back to shore to drop off its samples, you have taken a boat out to the first agent's location and want the other agents to meet you there. The one hitch is that, due to the difficulties of underwater communication, each agent has only one input and one output channel for communication. Is it possible to have the agents meet with these restrictions?

There are many possible ways to tackle this problem. One way would be to make a linear communication chain, where the first submersible relays information to the second, who relays information

to the third, and so on. This gives an A matrix of:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Note that this matrix is asymmetric since the communication channels are unidirectional. You may also notice that agent 5 is not sending data to anyone and agent 1 is not receiving data. By isolating agent 1 so that it receives no data, it will not move, which is what we want for our system.

The default Δt in the formation algorithm is designed to optimize convergence, but in this situation we have physical limitations on how quickly agents can communicate and alter their speed. We will set $\Delta t = 10$ so that each iteration represents 10 seconds passing. We can model communication delays in the system that occur due to distance between agents by setting: $\tau_i = round\left(\frac{\|\vec{q}_i(t) - \vec{q}_j(t)\|}{1000}\right)$, so that there is an extra 10 second delay for information sent by agent j to agent i for every 1000 meters between them. We can also consider the remaining energy levels of each agent by decreasing energy based on the work done at each time step:

$$E_i(t+1) = E_i(t) - F_d(\vec{q}_i(t+1) - \vec{q}_i(t)) - B\Delta t$$

Where E_i is the remaining energy for the ith agent, F_d is the force of drag on the agent, and B is a constant energy depletion that occurs due to the robot being on. This model assumes that the robot's speed is constant for each time step and that the only force acting against the robot's motion is drag. Lastly, we should cap the velocity of each robot so that it can not travel faster than would be physically possible. This could be done by taking $\dot{\vec{q}}_i = \alpha \frac{\dot{\vec{q}}_i}{\|\dot{\vec{q}}_i\|}$ if $\|\dot{\vec{q}}_i\| > \alpha$, where α is the maximum speed of the submersible in meters per 10 seconds (since our iteration size is 10 seconds). We could then run the simulation and see how well the system converges. Based on the performance, we could then try altering our A matrix or playing with restrictions to map out the relationships between our constraints and our system performance. We could also consider creating a cost function for this simulation in an attempt to prevent agents from wasting their batteries. Note that all parameters and values used in this simulation are fabricated for the purposes of the example.

When creating your own application, you should consider how every aspect of the dynamics can be changed to better simulate your situation. Does your situation require energy considerations? What influences delays? What determines if agents can communicate? Should the agents be offset? All of these questions should be thought out and researched to improve the quality of your simulation.

3 Flocking Algorithm

The flocking algorithm is designed to find a consensus velocity that all agents within the system will travel at, causing the agents to "flock" together. This algorithm could be used to model self-driving vehicles, animal behaviour, or any situation where agents are required to stick together while traveling.

3.1 Flocking Consensus Dynamics

The dynamics for the flocking algorithm are very similar to the formation algorithm, and are given by the system of differential equations:

$$\dot{\vec{q}} = \vec{v}$$
$$\dot{\vec{v}} = -L\vec{v}$$

The Laplacian matrix is $L = D - A$, as with the formation dynamics, although the adjacency matrix is redefined. Instead of the adjacency matrix representing open or closed communication channels between agents, it is assumed that all agents within the system are in communication,

but the effect of that communication is dependent on the distance between agents. The matrix entries for A are given by:

$$A(i,j) = \frac{K}{(\sigma^2 + d^2)^\beta}$$

Where $K, \sigma, \beta \in \mathbb{R}_{>0}$ are user-defined parameters, and $d = \|\vec{q}_i - \vec{q}_j\|$ is the Euclidean distance between agent i and agent j. The diagonal degree matrix D is once again defined as:

$$D(i,i) = \sum_{j=1}^{N} A(i,j)$$

The algorithm works by separately updating the x and y velocity. The updating step for x velocity would be:

$$\vec{v}_x(t+1) = \vec{v}_x(t) - L\vec{v}_x(t)\Delta t$$

3.1.1 Introducing a Leader

A leader is a single agent that follows a parametrized path independent of the velocities of other agents within the system. The goal of leader-based flocking is to have the other agents follow the path of the leader.

3.1.2 Introducing a Trigger Sequence

A trigger sequence, T, is a $1 \times T_{max}$ array, where $T_{max} \in \mathbb{Z}_{>0}$ is the number of iterations the system will run for. T is defined by:

$$T(t) = \begin{cases} 1 & \text{iff } \vec{v}_x(t+1) = \vec{v}_x(t) - L\vec{v}_x(t)\Delta t \\ 0 & \text{iff } \vec{v}_x(t+1) = \vec{v}_x(t) \end{cases}$$

This sequence determines when the algorithm will allow agents to communicate and update. If your Δt is very small, it may not be cost-effective or technologically possible in your application for agents to communicate at every iteration. You can use the trigger sequence to represent real world constraints on communication. Your team may want to study the number of communications that your model requires for the agents to remain well-connected (i.e. no rogue agents). The default trigger sequence is $T(t) = 1, \forall t \in [0, T_{max}]$.

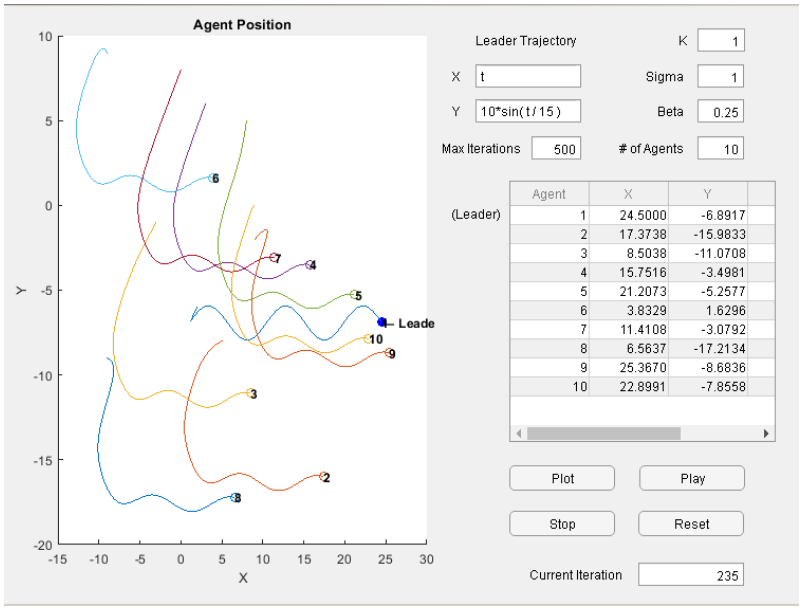

Figure 2: The flocking app in use

3.2 Using the App

The flocking algorithm is highly customizable, and your team can change the way agents interact using the flocking coefficients, as shown in the app window in Figure 2. To get an idea of the influence of flocking coefficients, you can analyze the dynamics equations of the system while also visualizing different parameter effects using the app. You can also introduce a leader into your flock by entering parametrized paths (in terms of t) in the trajectory entry fields. Agent 1 in the position table will become the leader of your flock, and will follow the parametrized path starting at the initial point it is given. You may want to create your own way of establishing adjacency matrix entries such that all agents in your network are highly influenced by the leader. If no parametrized path is entered, the flock will have no leader.

Pressing **Plot** will display the initial positions of all agents, while **Play** will commence the flocking algorithm. There will be a slight delay between pressing **Play** and when graphing starts. The plotting will stop when the algorithm reaches the final iteration, as set by the user in the app, or when the user presses the **Stop** button. If you wish to receive EXCEL data, you must wait for the app to reach the final iteration for the data to be exported.

The V_x and V_y columns are for inputting initial velocity values. Note that leaders will follow the initial velocity for one iteration before updating and following their parametric path.

3.3 Gearing Towards Your Application: Example

Consider the very simple situation where a rover is following another rover through the vast expanses of Mars. The lead rover is newly designed with a highly accurate GPS, and is heading on a direct course to the mission base. The follower rover is an older model with aged components. It can only communicate with the lead robot if they are within 10 meters of each other, can only process one communication every 60 seconds, and has no GPS. Without the lead robot, the follower robot is helpless to return to base, as it will gradually get knocked off course by the uneven Martian terrain. The mission base is 30 minutes of driving away, and we want to determine whether this mission is possible. To model this system we must consider a few things. First we redefine the A matrix:

$$A(i,j) = \begin{cases} \frac{(\sigma^2+d^2)^\beta}{K} & \text{if } \|q_i - q_j\| \leq 10 \\ 0 & \text{otherwise} \end{cases}$$

This makes it so that the agents can only communicate if they are within 10 meters of each other, and that the signals will be stronger as the follower agent gets further away (this is the opposite to the default dynamics, where influence is higher as agents are closer). This models an increasing "urgency" of the signals as the follower gets further off course.

We can define the leader's trajectory as $\vec{v}_L = (0.1t, 0.1t)$ so that it moves in a straight diagonal line at 0.141 meters per second. We define our trigger sequence as:

$$T(t) = \begin{cases} 1 & \text{if } t \equiv 0 \pmod{60} \\ 0 & \text{otherwise} \end{cases}$$

So that the velocity update step only occurs every 60 seconds.

Lastly, to model the rover going off path due to the uneven terrain, we can add noise values to the velocities of the follower rover: $\vec{v}_f(t) = \vec{v}_f(t) + Noise$, where $Noise$ is a 2×1 array with $Noise(k) = r,\ [r \in \mathbb{R} | -0.05 < r < 0.05]$.

We can now simulate the system using the flocking app, experimenting with the values of K, σ, and β to see whether the follower rover can make it back. Note that all parameters and values used in this simulation are fabricated for the purposes of the example.

When working on your own application, it's important to consider and research every aspect of the simulation and how they can be altered to better suit your project. Think about what determines if agents can communicate, how agents determine priority for the agents around them,

how often agents can update, and so on. Think mathematically and experiment with different dynamics.

4 Opinion Algorithm

4.1 Hegselmann-Krause Dynamics

The Hegselmann-Krause dynamics are used to simulate opinion changes within systems of agents, with agents moving based on the influence of agents around them. These systems can be used to model influence fields in social media, the spread of fake news, or other situations where agents' views or positions are altered by the agents around them.

4.1.1 One-Dimensional Dynamics

In the one-dimensional opinion algorithm, each agent has a communication radius $r_c \in \mathbb{R}_{>0}$. $\vec{q}$ is an $N \times 1$ position vector, where N is the number of agents in the system and $\vec{q}_i \in \mathbb{R}$ is the x-position of the ith agent. The adjacency matrix is an $N \times N$ matrix for the system, defined by:

$$A(i,j) = \begin{cases} 1 & \text{if } |\vec{q}_i - \vec{q}_j| \leq r_{c_i} \\ 0 & \text{if } |\vec{q}_i - \vec{q}_j| > r_{c_i} \end{cases}$$

Since in many cases $r_{c_i} \neq r_{c_j}$, the adjacency matrix for the system may be asymmetric.

We define the degree matrix as the $N \times N$ diagonal matrix with:

$$D(i,i) = \sum_{j=1}^{N} A(i,j)$$

Then we once again define our Laplacian matrix as $L = D - A$ and use the dynamics $\dot{\vec{q}} = -L\vec{q}$. We update agent position using:

$$\vec{q}(t+1) = \vec{q}(t) - L\vec{q}(t)\Delta t$$

Where $\Delta t \in \mathbb{R}_{>0}$ is a time-step determined within the system to achieve convergence.

Note: You may recognize these dynamics from the formation algorithm. That's because the opinion algorithm functions in a very similar way, with agents moving to what they perceive as the consensus position based on the other agents within their r_c.

4.1.2 Two-Dimensional Dynamics

The two-dimensional dynamics are very similar to the one-dimensional dynamics. Each agent has a radius of communication $r_c \in \mathbb{R}_{>0}$ and a position $\vec{q}_i \in \mathbb{R}^2$, with an overall $N \times 2$ position vector $\vec{q}$. A is defined as:

$$A(i,j) = \begin{cases} 1 & \text{if } \|\vec{q}_i - \vec{q}_j\| \leq r_{c_i} \\ 0 & \text{if } \|\vec{q}_i - \vec{q}_j\| > r_{c_i} \end{cases}$$

Where $\|.\|$ is the Euclidean norm. We define A, D, and L in the same way as in one dimension. We then update the position in the same way::

$$\vec{q}(t+1) = \vec{q}(t) - L\vec{q}(t)\Delta t$$

The two-dimensional dynamics simulate networks with more complicated opinion profiles, with agents being swayed in two different directions independently. For instance, a two-dimensional opinion could be a model of political inclinations, where agents fall on both an economic spectrum and a social spectrum, which could be treated independently of each other.

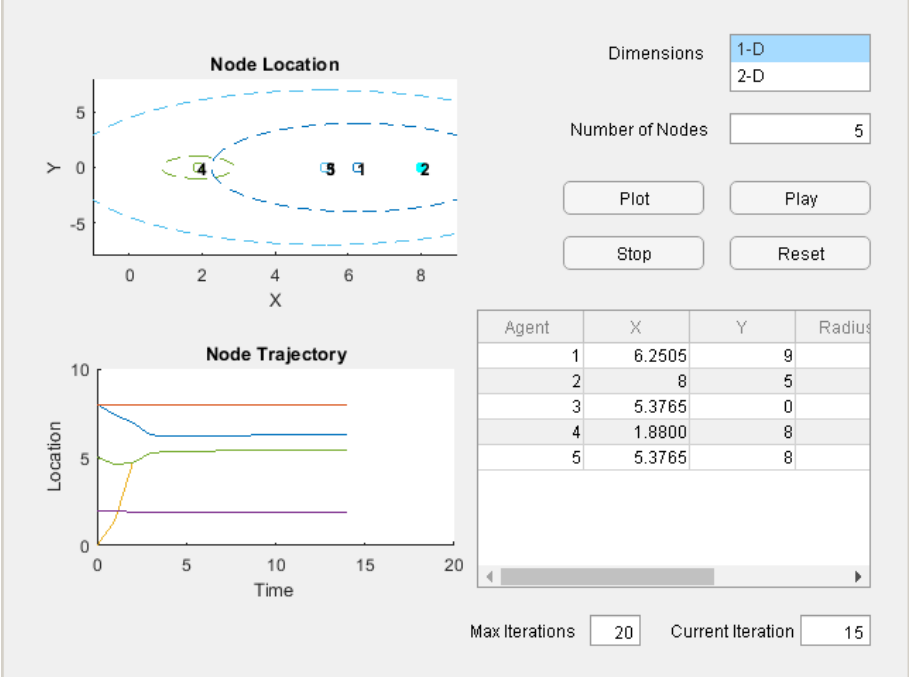

Figure 3: The opinion app in use

4.2 Using the App

The opinion dynamics app can be seen in Figure 3. With this app, you have control over a few options: you can specify the number of agents you want in your system, you can control the radius of communication of each agent, and you have the option to work in either one dimension or two dimensions. Stubborn agents, where $\vec{q}_i = 0, \forall t \in \mathbb{Z}_{>0}$, can be created by setting the radius of communication of an agent to zero so that it receives no information from others. Stubborn agents are displayed in cyan with filled-in centres, while other agents are coloured. The radius of communication of each agent is also displayed.

(i) *One Dimension*: In one dimension, pay attention to the agents' x positions (opinions), their radii, as well as their left and right noise values. A noise value is simply a reduction in the radius on that particular side, which limits the communication with other agents.

(ii) *Two Dimensions*: In two dimensions, pay attention to the x and y positions (opinions) and the agent radii. Noise values do not apply to two-dimensional dynamics.

The **Plot** button will display the initial positions of the agents in the Node Location Plot. When the **Play** button is pressed, the top plot shows the opinion profile of the agents at a particular time while the bottom plot shows the evolution of the opinions over all time. The **Stop** button will halt the dynamics. If you want EXCEL data to be exported, let the algorithm run fully until the final iteration, as set by the user. The second plot is omitted for the two-dimensional dynamics.

4.3 Gearing Towards Your Application: Example

Consider a situation where we're modeling the spread of fake news within online communities. One of the defining characteristics of this system could be trustworthiness, where agents "trust" agents that are closer to them moreso than agents that are far away. This represents how individuals are likely to be swayed by sources they're familiar with and that echo their own political views. We know that the standard Hegselmann-Krause dynamics make agents move to the consensus position of all agents within their radius r_c, equally weighting every opinion they can see. These dynamics

don't suit our system, as we want agents to favour opinions that are closer to their own. To change this we can redefine the A matrix as:

$$A(i,j) = \begin{cases} \frac{1}{(1+d)^2} & \text{if } d \leq r_{c_i} \\ 0 & \text{if } d > r_{c_i} \end{cases}$$

Where $d = |q_i - q_j|$ in the one-dimensional case, and $d = \|q_i - q_j\|$ in the two-dimensional case. r_{c_i} is a communication radius value that would be determined through research.

This new definition checks if agents can communicate, and then weights the influence of opinion based on the distance between the agents, with smaller distances resulting in greater influence. In this way, it achieves our desired goal. We could also model an "influence leader" k by setting: $A(i,k) = 10 \times A(i,k), \forall i \in [1,N]$, which increases the influence of the kth agent on all other agents by an order of magnitude. Lastly, we could cap velocity by normalizing $\dot{\vec{q}}_i$ if it exceeds some maximum speed, as seen in section 2.3. Note that all parameters and values used in this simulation are fabricated for the purposes of the example.

For your own application, research how agents within your system act. Once you have an understanding of the basic characteristics of the system, you can start altering the algorithm in many ways to simulate your project. Perhaps you want to introduce boundaries that agents can't cross or define a relationship between the number of agents in contact and the size of r_c. There are many possibilities, so get creative!

5 Lloyd's (Deployment) Algorithm

Lloyd's algorithm takes a system of agents and allocates them evenly around an area according to the distribution of some resource of interest. This could be applied to search and rescue missions, robots collecting soil sample, or other situations where agents must cover some limited area and so they spread out to service it.

5.1 Lloyd's Algorithm Dynamics

The deployment algorithm uses Lloyd's algorithm to find the optimal positions that agents should be in to increase coverage of some density D over an area A. In this case, $A \subset R^2$, and D is a function $D : A \subset R^2 \to R_{\geq 0}$. D may also be time/iteration varying, but this is described in more detail in section 5.4.2. It is important that the range of D be nonnegative, since information is nonnegative and (more importantly) the app uses functions based on a nonnegative density function. Let the number of agents be N, the set of all agents be V ($|V| = N$), each agent be denoted by $v_i \in V, i \in \{1, ..., N\}$, the position of each v_i be $p_i \in P$, and let the set of all agent positions be $P \subset A$. This algorithm takes into account a radius of observation r_o and a radius of communication r_c. If a point $x \in A$ is within r_c of the position of p_i (i.e. $x \in \bar{B}_{r_o}(p_i)$, where $\bar{B}_{r_o}(p_i) = \{x \in A : \|x - p_i\| \leq r_o\}$, and $\|.\|$ is the Euclidean Norm (this will be important later!), then v_i can see x and $D(x)$. Furthermore, if an agent v_j is at p_j, and p_j is within r_c of another agent's position p_i (i.e. $p_j \in \bar{B}_{r_c}(p_i)$), then v_i and v_j can communicate with one another. Note that communication goes both ways, i.e. if v_i can communicate with v_j, then v_j can communicate with v_i. In addition, it is assumed that if any two agents v_i and v_j are in the same connected graph, which is to say that there exists a path between any v_i and v_j, then they can communicate with one another. By doing this, we can partition V into sets of communicated agents $V_{com_k} \subseteq V, k \in \{1, ..., K\}$, where K is the number of communicated sets. We then assume that the communicated agents pass information about the density at points $x \in A$ to each other. We can then analyze certain subsets of A in the following way. Let $O_k \subseteq A$ be the observed region by all the agents in V_{com_k}, i.e.

$$O_k = \bigcup_i \bar{B}_{r_o}(p_i), i \in V_{com_k}$$

Then, we determine the points in O_k which are closest to each agent $v_i, i \in V_{com_k}$, which in turn partition O_k (these partitions are known as Voronoi regions). If we let $R_{k_i} \subseteq O_k$, $i \in V_{com_k}$ be

such partitions, then

$$R_{k_i} = \{\in O_k : \|k_x - p_{i_k}\| \leq \|k_x - p_{j_k}\| \forall j \in V_{com_k}\}.$$

Then, agent v_i is assigned the partition R_{k_i}, that is to say that agent v_i has the task of covering R_{k_i}. This is done by calculating the mass $M_{k_i} \in R$ of each region R_{k_i} as follows:

$$M_{k_i} = \int_{R_{k_i}} D(x,y)dA$$

Then, the centroid $C_{k_i} \in A$ of R_{k_i} is calculated as follows:

$$C_{k_i} = \frac{1}{M_{k_i}} \left(\int_{R_{k_i}} xD(x,y)dA, \int_{R_{k_i}} yD(x,y)dA \right)$$

Finally, the agent v_i will move towards their C_{k_i}. Note that these integrals are computed by discretizing the density functions and taking sums. This sacrifices a certain degree of accuracy but significantly decreases computing time. If the density is not time/iteration varying, then the agents will converge at C_{k_i}, which is the optimal positions for the agents to be in so as to cover the density over A. Note however that this heavily depends on r_o and r_c since not all of A may be visible, and if $r_c < r_o$, the agents may not know that they are covering the same area. The process of determining O_k and R_{k_i} is then repeated every time the agents move.

5.2 Using the App

The MATLAB code works off the overarching *PlotButtonPushed* Callback Function, which is called by the app when the plot button is pressed. Before going into a description of this function, the basic parameters, variables, and plots used in the function code are described below.

5.2.1 Parameters and Variables

These include inputs to functions used in *PlotButtonPushed* as well as outputs. For a description of each of these functions see the separate Lloyd's Algorithm Functions document.

(i) *sides*: Dimensions of A, which is restricted to a square region. Hence, A is an area with dimensions $sides \times sides$. The units of these dimensions is determined by the user.

(ii) *Partition Number*: The space A is discretized by dividing each unit of space into Partition Number $\in$ Z$>$0 of parts. In other words, for each unit of space, the algorithm considers Partition Number of them. For example, if A has dimensions 5×5 and $Partition\_Number = 10$, then A is represented in MATLAB by a 50×50 matrix.

(iii) *Density, D*: Once the space is discretized, a density matrix is created. The (y,x) positions in A are represented by the matrix as $(y,x) = \left(\frac{i}{Partition\_Number}, \frac{j}{Partition\_Number}\right)$, or equivalently, if you have (y,x) positions, the entry to the matrix is $(i,j) = (floor(y*Partition\_Number), floor(x*Partition\_Number))$.The reason why points are represented as (y,x) as opposed to (x,y) is so that x runs horizontally and y runs vertically.. Therefore, $Density(i,j)$ is a $(Partition\_Number)(sides) \times (Partition\_Number)(sides)$ matrix. If $T = (Partition\_Number)(sides)$, then the matrix looks as follows:

$$Density = \begin{bmatrix} D(x_1,y_1) & D(x_2,y_1) & \dots & D(x_T,y_1) \\ D(x_1,y_2 & D(x_2,y_2) & \dots & D(x_T,y_2) \\ \vdots & \vdots & \ddots & \vdots \\ D(x_1,y_T) & D(x_2,y_T) & \dots & D(x_T,y_T) \end{bmatrix}$$

If your application requires the density to be dynamic, i.e. it varies over time, then the matrix *Density* changes with every iteration according to some function. The way in which the density changes depends on your application. Details on how to change this are described in section 5.4.1 and 5.4.2. If the Density is not dynamic, then it is static.

(iv) *Agent Positions*: An $N \times 2$ matrix with the x-position of agent v_i in the first column of row i, and the y-positions in the second column, as below:

$$Agent\_Positions = \begin{bmatrix} x_{v_1} & y_{v_1} \\ x_{v_2} & y_{v_2} \\ \vdots & \vdots \\ x_{v_N} & y_{v_N} \end{bmatrix}$$

(v) *com_mat*: An N × N communication matrix where:

$$com\_mat(i,j) = \begin{cases} 1 & \text{agent } v_i \text{ communicates with } v_j \\ 0 & \text{agent } v_i \text{ does not communicate with } v_j \end{cases}$$

Note that since communication goes both ways we also have that com_mat is symmetric (i.e. $com\_mat(i,j) = com\_mat(j,i)$).

(vi) *coms*: A $1 \times K$ cell array which contains V_{com_i} in cell i (see 5.1). V_{com_i} is represented by a vector containing the indices of the connected agents, i.e. each $v_j \in V_{com_i}$, in other words, all the agents in each cell can communicate with one another. The matrix is shown below:

$$coms = \left(\begin{bmatrix} a \\ b \\ \vdots \\ c \end{bmatrix} \begin{bmatrix} d \\ e \\ \vdots \\ f \end{bmatrix} \cdots \begin{bmatrix} g \\ h \\ \vdots \\ i \end{bmatrix} \right)$$

where $a, b, \cdots \in V$.

(vii) *Agent Points*: A $1 \times N$ cell array that contains $r \times 2$ matrices. Matrix i contains the discretized points of R_{k_i}, hence r is the number of points in R_{k_i} (note that there is only a finite number of them since the space has been discretized). The first column of $Agent\_Points$ contains the x-coordinates, and the second column contains the y-coordinates. If the total number of points in each R_{k_i} is K_i, then the matrix looks as follows:

$$Agent\_Points = \left(\begin{bmatrix} [x_{1,v_1} & y_{1,v_1} \\ x_{2,v_1} & y_{2,v_1} \\ \vdots & \vdots \\ x_{K_1,v_1} & y_{K_1,v_1}] \end{bmatrix} \begin{bmatrix} [x_{1,v_2} & y_{1,v_2} \\ x_{2,v_2} & y_{2,v_2} \\ \vdots & \vdots \\ x_{K_2,v_2} & y_{K_2,v_2}] \end{bmatrix} \cdots \begin{bmatrix} [x_{1,v_N} & y_{1,v_N} \\ x_{2,v_N} & y_{2,v_N} \\ \vdots & \vdots \\ x_{K_N,v_N} & y_{K_N,v_N}] \end{bmatrix} \right)$$

(viii) *Mass*: A $1 \times N$ array containing the mass of R_{k_i} in column i. Here $Mass(i) = \sum_{(i,j)} Density(i,j)$ for (i,j) corresponding to $(x,y) \in R_{k_i}$, which are the points in the matrix $AgentPoints\{i,:\}$. The matrix looks like this:

$$Mass = \begin{bmatrix} M1 & M2 & \dots & M_N \end{bmatrix}$$

(ix) *Centroids*: An $N \times 2$ matrix containing the x-position of the centroid of R_{k_i} in the first column of row i, and the y-position in the second column. The centroids in MATLAB are calculated as:

$$Centroids(i,1) = \frac{1}{Mass(i)} \sum_{i,j} x \cdot (Density(i,j)) \quad \text{for} \quad (x,y) \in R_k$$

$$Centroids(i,2) = \frac{1}{Mass(i)} \sum_{i,j} y \cdot (Density(i,j)) \quad \text{for} \quad (x,y) \in R_k$$

corresponding to the appropriate (i, j) in Density. Note that these formulas are the discretized versions of those taught in APSC 172. The matrix then looks as follows:

$$Centroids = \begin{bmatrix} x_{c,1} & y_{c,1} \\ x_{c,2} & y_{c,2} \\ \vdots & \vdots \\ x_{c,N} & y_{c,N} \end{bmatrix}$$

(x) *covered mass*: The sum of the density values that are within r_o of all of the agents. This assumes that if for an agent i, and a point $x \in A$ we have that $x \in B_{r_o}(p_i)$, then x is "covered" by the agent.

(xi) $MOVEMENTSCALE$: This is used in the simulation if Velocity Type = 2 (i.e. if Proportional Velocity is selected). The user may have the agents move as follows. Given a position p_i of an agent i in $AgentPositions(i,:)$, and the centroid C_{k_i} of the agent's assigned region R_{k_i} in $Centroids(i,:)$, then after one iteration, agent i will have moved a distance of $MOVEMENTSCALE * \|C_{k_i} - p_i\|$. Note that MOVEMENTSCALE must be greater than zero in order for the agents to move, and less than 2 in order for the algorithm to converge. A MOVEMENTSCALE of 1.8 is generally accepted to lead to the fastest convergence, however this value may not be realistic given one's application.

(xii) *velocity*: Each iteration represents a single unit of time. The units of space and time are determined by the user. For example, if you decide that each unit of space in the algorithm represents 1 km, and each unit of time is 1 hour, then velocity will be in units of km/h. Hence, after one iteration agent i will have moved a distance of $1 \times velocity$ in the direction of the centroid of R_{k_i}.

(xiii) *max velocity*: Once the distance d to move after one iteration is determined, the agent's velocity is taken to be $v = d/\delta t = d/1$. Note that $\delta t = 1$ since each iteration is considered to be one unit of time. If $v > max\_velocity$, the velocity of the agent will be set to max velocity for that iteration. Note that if a constant velocity v is implemented and $v < max\_velocity$, then max velocity has no effect on the algorithm.

(xiv) *delay*: If your density is dynamic, i.e. it changes over time, then delay determines the number of iterations between each change. For example, if $delay = 5$, then Density will only change every 5th iteration. Note that delay must be a natural number.

(xv) *r_o, r_c, Number of Agents*: These are the radius of observation r_o, the radius of communication r_c, and number of agents N described in 5.1.

(xvi) *Percent Covered, Total Mass*: Percent Covered is an array that contains the percentage value of the covered mass (see (x)) relative to the Total Mass of the whole area A, for each iteration. Percent Covered is a vector that has the latest value appended after each iteration for the sake of plotting its corresponding graph in the app.

(xvii) *distance traveled*: An array that contains the sum of the distances traveled by all agents each iteration. Similar to Percent Covered, this vector grows every iteration for the sake of plotting its corresponding graph in the app.

(xviii)E: A $1 \times N$ array containing the Energy of agent i in column i, that looks like:

$$E = \begin{bmatrix} E_1 & E_2 & \dots & E_N \end{bmatrix}$$

You must determine the units of the energy that you wish to use and how you want to represent this in the array E. The energy will decrease after every iteration by some parameter (you may choose how to represent this). In the regular algorithm, we decrease the energy of each agent depending on the velocity they moved at (i.e. $E(i)_{j+1} = E(i)_j - \frac{1}{2} m_i v_i^2$ (where m_i is the mass of the agent, and v_i is its velocity)). Note that the only place where the energy of the agents is used, is in the function *distance_between* only if algorithm type = 2 (i.e. a custom algorithm is being used). You may however wish to use this as an output to see how the energy of the agents decreases. An example of how and why you may want to use this is given in section 5.4.4.

Note: Keep in mind that the units of time and space are relevant for *sides*, *Agent_Positions*, *Centroids*, *MOVEMENTSCALE*, *velocity*, *delay*, r_c and r_o, and must therefore be consistent throughout. The interpretation of your inputs to the app (see section 4.3) and its outputs will depend on which units you choose.

5.2.2 Plots

There is a total of five plots in the app, two of which appear on the same graph. Here the plots are described as they are now, but some should be changed to what is relevant to your application (how to do this is explained in section 5.4.6).

(1) **Arena, Plot 1**: x and y axes represent the position of the agent in A. The interpretation of each unit of space is determined by you (see (i)). This plot contains the position of each agent in blue and the centroids of their assigned regions in red. The radius of observation r_o is shown in magenta around every agent. The contours of the density are in colours ranging from blue to yellow, where yellow means that there is a higher density, and blue means there is a lower density. Note that the colours are relative, and blue simply means lower than another point (also, if your density is constant (eg. 1 everywhere), then no contours will show). Apologies for any flashbacks to bugs crawling along hills in 172 that this may cause. Finally, the blue straight lines that partition the plot are the regions R_{k_i}, i.e. the assigned regions of each agent.

(2) **Coverage, Plot 2 (Editable)**: The independent variable is the iteration number, and the dependent variable is the percentage of the total covered mass. That is, given a covered mass by the agents (this is described in (x)), the total mass of the region A is calculated according to the density over A, and the percentage of total covered mass is displayed. You should cater this to your project by considering what constitutes "coverage" in your particular application. Since the goal of Lloyd's Algorithm is to maximize coverage, this is likely the most important plot in the app.

(3) **Distance Travelled, Plot 2 (Editable)**: The independent variable is the iteration number, and the dependent variable is the total distance moved by all agents. This plot is therefore nondecreasing. You may wish to change this depending on your application, see section 5.4.6 to see how this can be done. Something to note is that the way in which your agents 'act' on the mass is dependent on your application, and hence you may want this plot to take that into account (see 5.4.6). This plot appears superimposed on the coverage plot, and corresponds to the y-axis on the right side of the graph.

(4) **Assigned Points, Plot 3**: The plot has identical units to Arena 1. This plot is there to help you understand which agents are assigned which regions. Although this is also represented in the Arena plot in the app, the partitions (lines in light blue) are considering which agents are closest to certain points using the Euclidean distance. In section 5.4.4 we describe how you may use other notions of distance that may alter the way in which regions are assigned to certain points. Hence, Assigned Points shows the agents as diamonds in a particular colour with their index next to them. The circular points of the same colour are the points assigned to that particular agent.

(5) **Communication, Plot 4**: The plot is identical in units to Arena 1, where the points are the positions of the agents. If a line connects any two agents, then they can communicate with one another. The connected agents, i.e. the agents in each V_{com_i}, have lines of the same colour connecting them. Note also that if there exists a path between any two agents, whether it be direct or through five other agents, then they can communicate with one another.

5.2.3 MATLAB Algorithm Description

We can now show how the main function works overall. Below we show a basic description of how the process described in 5.1 is implemented in MATLAB. Note that a variety of functions can be changed to be geared towards your application. A description of how this may be done, which functions may be changed, and some ideas of how they may be changed is presented in section 5.4. The functions that are used in MATLAB to perform the actions in 1 are described in the Lloyd's Algorithm Functions document, and referenced in square brackets beside each step in Algorithm 1.

Algorithm 1: `LloydsPlay`

Data: `Number_of_Agents`, `sides`, `Partition_Number`, `Agent_Positions`, `Density`, `r_o`, `velocity`, `MOVEMENTSCALE`, `max_velocity`, `r_c`, `delay`
Result: Move agents towards their corresponding `Centroids`

Create (`Partition_Number`)(`sides`) × (`Partition_Number`)(`sides`) `meshgrid`;
Fix `Agent_Positions` such that
`Agent_Positions(i,:)` $\neq$ `Agent_Positions(j,:)` $\forall i, j \in \{1, \ldots, N\}^1, i \neq j$ [(xiii)];
while *Stop and Reset not pressed* **do**
  Create communication matrices `coms` and `com_mat` [(ix)];
  Create communication graph in plot 4 [(viii)];
  Assign points in agent's corresponding area (R_{k_i} for agent i) to `Agent_Points` [(iii)];
  Calculate the masses of each agent's assigned area using `Agent_Points` to create `Mass`[(xviii)];
  Calculate `covered_mass` [(x)];
  Calculate `Percent_Covered` based on `covered_mass` and display in plot 2;
  Calculate `Centroids` [(v)];
  Move agents towards `Centroids` updating `Agent_Positions`[(xix)];
  Decrase energy contents of agents `E` [(xix)];
  Plot `Agent_Positions` in plot 1;
  Calculate `distance_travelled` and display in plot 3 [(xix)];
  if `Density` *is dynamic* **then**
    Update `Density`[(xv)];
    Recalculate `Total_Mass`;
  end
end

5.3 How to Run the App

The app has some basic parameters that you can change which are specific to your application. The running app is shown in Figure 4. On the right hand side you can input the parameters for your application. In order to open the app you must first open the MATLAB app designer by entering "appdesigner" in the command window . The app will open up and after pressing play you will be able to choose parameters described below. If it seems like the program is not running, it is best to press ctrl-c in the MATLAB Command Window, then run it again.

NOTE: To run the app, you **must** have the algorithm's accompanying functions downloaded and in your MATLAB file path. To do this, download the "Lloyd Functions" folder, and place the contents in the MATLAB folder you work from, along with the app file itself.

5.3.1 Inputs and Buttons

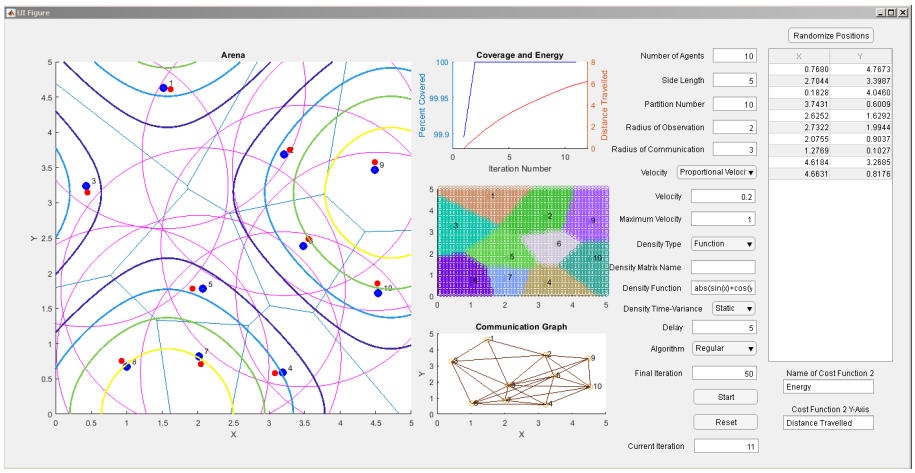

Figure 4: The Lloyd's app in use

(i) **Number of Agents**: (xv) A number $N \in \mathbb{Z}_{\geq 1}$ such that $N \geq 3$ (this is necessary since the Voronoi diagram is dual of the Delauny Triangulation (don't worry about this, it'll make sense in third year)).

(ii) **Side Length**: (i) A number $s \in \mathbb{R}_{>0}$ which represent sides. Keep units in mind, and

you have to input a decimal number here (opposed to a fraction) so that MATLAB can interpret it. Program runs slower as Side Length increases, see note in 5.3.1.

(iii) **Partition Number**: (ii) A number $p \in \mathbb{Z}_{\geq 1}$ representing Partition Number. Precision increases but program runs slower as Partition Number increases, see note in 5.3.1.

(iv) **Radius of Observation**: (xv) A number $r_o \in \mathbb{R}_{>0}$ corresponding to r_o. Keep units in mind.

(v) **Radius of Communication**: (xv) A number $r_c \in \mathbb{R}_{\geq 0}$ representing r_c. Keep units in mind.

(vi) **Type of Velocity/Velocity Edit Field**: Here you have a choice between making the velocity a constant for all the agents, or making it proportional to the distance to their corresponding centroid:

(a) Constant: (xii) A number $v \in \mathbb{R}_{>0}$. This choice will make the velocity of all agents the same for each iteration. Keep units in mind.

(b) Proportional Velocity: (xi) A number $m \in \mathbb{R}_{>0}$. Must have $0 < m < 2$ in order for the algorithm to be stable (i.e. to converge). This value corresponds to MOVEMENTSCALE.

(vii) **Maximum Velocity**: (xiii) A number $v_{max} \in \mathbb{R}_{>0}$ that bounds the velocity at which agents can move. The program considers velocity to be distance/iteration , so consider what unit of time you would like an iteration to represent (seconds, hours, days). Set this constraint according to your application - are you simulating helicopter flight? If so, what's the maximum speed of a helicopter?

(viii) **Density**: You may choose between having a symbolic function, a matrix as a representation of the density distribution over the arena, or your own representation of such. See section 5.4.1 for a thorough description of how to use this:

(a) Function: A function in terms of x and y. The function must not take negative values within the arena (consider using MATLAB's abs() function to ensure this). This function is discretized into the matrix Density (iii) by the MATLAB code.

(b) Matrix: Use the text edit field to enter the name of the matrix file within your MATLAB environment that you'd like to use. This must be a numeric matrix type and a .mat file. This matrix will be the Density matrix (iii) used. Must be a square matrix of dimensions $(sides * Partition\_Number) \times (sides * Partition\_Number)$.

(ix) **Types of Density**: Your Density may be static or dynamic.

(a) Static: If selected, the density distribution does not change with time.

(b) Dynamic: If selected, Density will change after a certain number of iterations. See section 5.4.2 for a full description of how you may use this.

(x) **Delay**: If your Density is time-varying and $delay = d$, then it will change every d^{th} iteration (see 5.2.1). You must have an input here even if your Density is not iteration-varying. Note that delay must be a natural number. If $delay = n$ then the density changes every n^{th} iteration. Consider what this would mean for your application.

(xi) **Type of Algorithm**: You may choose between the **Regular Algorithm** and a **Custom Algorithm**:

(a) Regular Algorithm: Runs the algorithm described in section 5.1.

(b) Custom Algorithm: Runs a custom algorithm with edited functions. The functions you can edit and how to edit them are described in section 5.4.

(xii) **Agent Starting Positions**: The starting values of Agent Positions (iv). Must be numbers $x, y \in \mathbb{R}_{\geq 0}$ such that $\frac{1}{Partition\_Number} \leq x, y \leq sides$. If you want to input your own positions, then type in a decimal into the table. If you want to have random positions generated for you, click Randomize,

(xiii) **Randomize**: Populates Agent Starting Positions with randomly generated x and y values.

(xiv) **Name of Cost Function 2**: Determines the title of the second cost function in plot 2 in the app.

(xv) **Cost Function 2 y-axis**: Determines y-axis label of the second cost function in plot 2 in the app.

(xvi) **Reset**: Can be pressed after the algorithm has stopped. Will clear the figures. After pressing Reset, press Start to execute the algorithm from the beginning, or you may enter new parameters and/or select other options in the app, and then press Start. Note that if you want to save the Excel file generated by the algorithm, you should rename the file before pressing Start again since the current Excel file will be overwritten.

(xvii) **Final Iteration**: The algorithm will automatically stop after this number of iterations.

(xviii) **Start**: Runs the algorithm with the selected options and parameters. All fields must be selected/filled before this button is pressed.

Note: Since the algorithm checks through every point in the arena A in order to assign points to particular agents (that is, after A is partitioned into discrete slots), the algorithm will take more time to calculate this the larger $sides$ and $Partition\_Number$ are. Note that if you are using a function to represent your Density, Matlab has to evaluate that function at $(sides * Partition\_Number)^2$ points. Therefore, the larger $sides * Partition\_Number$ is, the more time it will take to discretize the function into a matrix. This happens before any plotting begins, which means that if nothing is plotting, the algorithm is still discretizing the Density function. If one wants to run the algorithm in less than 3 minutes then it is recommended that $sides * Partition\_Number \leq 25$ (which will partition the plane into ≤ 252 slots). However, one could run the algorithm with a much larger value of $sides * Partition\_Number$ if one is prepared to let the program run for a long period of time (perhaps even overnight).

5.4 Gearing Towards Your Application

The algorithm will take a different form depending on each application. Here we describe the main functions and inputs that can be changed in order to change the algorithm's behaviour. If a function is to be changed, then the app lets you decide between Regular Algorithm and Custom Algorithm. In Matlab, this is represented by the variable algorithm type. If $algorithm\_type = 1$, then Regular Algorithm was selected. If $algorithm\_type = 2$, then Custom Algorithm was selected. In the sections below we have added the option of algorithm type. However, if you wish to alter other functions, add algorithm type as one of the inputs to the function. It is best to add inputs to functions or parameters in the overarching function PlayButtonPushed. In order to see which functions you would have to edit or call, see the Llody's Algorithm Functions document.

5.4.1 Density

5.4.1.1 Description: Depending on the application, it may be important to think of Density as a tool to make the algorithm behave in a particular way, besides it being only a description of the physical density or distribution of your system. The way to think of this is that you may have some starting distribution of information, then on top of that you would have a function, F that has as one of its inputs the value of the existing distribution, i.e. $F : A \subset \mathbb{R}^2 \times D \to \mathbb{R}_{\geq 0}$,

where D is the physical density or distribution function. F may have other inputs besides the physical density or distribution and position as well, and this is specific to your application. Your input to the algorithm would then be the image of F after it is applied to the physical density or distribution.

5.4.1.2 How to Change It: In the app, the Density will be an input of either a symbolic function or a (Partition Number $*$ sides $\times$ Partition Number $*$ sides matrix). If the input is a symbolic function, then it must be in terms of x and y (e.g. if $D(x, y) = x^2 + y^2$, then the input in the app should be $x^2 + y^2$). As mentioned before, make sure that this function is nonnegative. An easy fix is to simply take the absolute value of your function. After you input a function into the app, fun_to_array (xiv) will convert it into a matrix. If you are using a matrix, make sure that you save it in MATLAB as a .mat file. This is done by simply saving the matrix at the end of the script that creates it. Within your script make sure you name the final density variable 'D', otherwise the app will not be able to interpret the struct file created when opening the .mat file. Save your matrix using the line of code "`save('density.mat','D')`". Make sure it is in the same folder as the rest of the functions the app uses and that it has the appropriate dimensions. To select the Density in the app, use the text edit field to enter the name of the .mat file. Depending on your application, it may be easier to work with a matrix so that you can change values of specific entries and/or have a better understanding of how the density is laid out.

5.4.1.3 Example: Imagine you are modeling a system where the agents are drones that have the ability to dump water on certain areas during a forest fire. Assume also that these drones are very expensive and it is therefore unfeasible to deploy enough robots to pour water on every square meter of the forest. It would therefore be more useful to deploy the agents such that they concentrate on the parts of the fire closer to the perimeter of the forest, in order to decrease the chance of the fire spreading. Therefore, your Density should be changed to address this. This could be done by creating a function F that increases the density closer to the perimeter. Then, this new Density matrix, the image of F, will be your input to your algorithm and the agents will move towards the areas of higher density.

5.4.2 Dynamic Density (function: iteration_variation_fun)

5.4.2.1 Description: As mentioned above, Density should be used as a tool geared for your application. Hence there may be instances where Density may change over time. Here we talk about two occasions of this. First, Density may vary because of the particular system you have chosen, we'll call these external changes. Second, Density may vary because your agents cause it to change since they have 'acted' on the Density, we'll call these internal changes. The way to think of this is that having some starting Density, you have a function G that changes Density after every iteration, i.e. $G : A \subset \mathbb{R}^2 \times Im(F) \times \mathbb{Z}_{\geq 1} \to \mathbb{R}_{\geq 0}$, where F is from section 4.4.2.1 and the iteration number is $j \in \mathbb{Z}_{\geq 1}$. G may however have other inputs as well depending on your application.

5.4.2.2 How to Change It: Since the way your Density changes and its inputs are specific to your application, you have to change this in the MATLAB function iteration_variation_fun. Here we have entered an arbitrary function G that externally changes the density depending on the iteration, and also uses delay described in section 5.2.1. The x and y positions corresponding to the i and j entries to the matrix have been calculated, and your function G should be stated where it is indicated in the code. We have also entered an internally changing density option, where the density at all locations within the radius of observation of an agent is halved. Note that you may need other inputs depending on your application. Since this function is called by other functions in MATLAB, you should change inputs to all so that the code works well. See the Lloyd Algorithms Function documentation to see which functions are called by iteration_variation_fun, and which function iteration_variation_fun calls.

5.4.2.3 Internally Changing Density Example: Assume that your application involves imaging some region of the night sky, and that some regions are more important to you than others so that they may be analyzed (perhaps you want to observe a supernova and there is more chance of one happening in some specific region of space). Hence, Density should be a

matrix (or function) of how important each region of the sky is. Assume that you want to do this by sending a constellation of small telescopes to orbit around the Earth and focus on certain regions. Assume also that you are able to turn this into a 2 dimensional problem (perhaps you only observe at one specific point in time at night so you don't care about the time when the telescopes are rotating about the Earth; then, since the images are 2 dimensional, you may create a map that could be seen by the telescopes at specific times of the Earth's rotation about the Sun). Although this may be an interesting project to develop in 3 dimensions, let us assume that it is 2 dimensional. Assume now that it is not enough that the image one telescope takes of one region is of enough quality to be analyzed; rather three images of the same region should be taken for better results. Then, as an agent covers one region the Density at the points of the that region should decrease by a certain amount; for example, this could be done by decreasing the density as $Density(i, j)/Number\_of\_Agents$.

5.4.2.4 Externally Changing Density Example: Assume that your application consists of a group of fishing boats in the ocean. The iteration-varying density can be used to represent fish swimming in a certain direction. The code currently implemented represents fish swimming to the right, with no 'new' fish entering the arena. One could also add an internal density change to represent the fish being extracted from the ocean.

5.4.3 Communication

5.4.3.1 Description: The way in which agents communicate may not be as described in section 5.1, and is specific to your application. However, it is still an assumption that communication between agents goes both ways.

5.4.3.2 How to Change It: The output that you will be changing is the communication matrix com_mat described in section 5.2.1, where com_mat(i, j) =com_mat$(j, i) = 1$ if agents i and j can communicate with one another, otherwise com_mat$(i, j) = 0$. The section in the function titled Editable com_mat is where you create your specific communication matrix.

5.4.3.3 Example: Assume that your application involved creating a communication network between satellites on Earth and Satellites on Mars (perhaps data is being collected by some robot in Mars and this must be sent back to Earth through some satellite orbiting Mars). However, assume that the satellites have only a limited range of communication (as in section 5.1). However, since signals cannot travel through planets or the sun, at times when Mars and the Earth are not on the same side of the Sun, a communication network that goes around the Sun may be necessary. Hence, communication no longer depends solely on whether the satellites are within a radius of communication of one another. This may be addressed by having certain regions of your arena A not allow signals to pass through (these areas would be the planets and the Sun).

5.4.4 Distance (function: distance_between)

5.4.4.1 Description: As mentioned in section 5.1, the distance between agents is calculated using the Euclidean distance. However, there are other ways in which to calculate distance between points and the agents in order to make the algorithm more applicable to reality. In the Regular Algorithm, distance is $d = \sqrt{x^2 + y^2}$, and distance is only calculated between points and the positions of the agents (i.e. never between two points where no agent is positioned). This may be helpful to introduce parameters under which some points may seem closer or farther from some agents compared to others.

5.4.4.2 How to Change It: The output to this function is simply d where $d = \sqrt{x^2 + y^2}$. The function that calculates this is distance_between. In order to change the way in which distance is calculated, go to the function under the section Editable distance. Here it is important to see which functions used distance_between so that you may gear the changes towards your application.

5.4.4.3 Example Setup: Assume your application involves weakling terminator agents that do not have an infinite amount of energy. Then, in order to save energy, it is better to have the agents with lower energy be assigned areas that are not far away from them, compared to agents

with a larger amount of energy. One way to approach this is to make the distance between points and the agents be proportional to the energy each of the agents has. More specifically, if some agent v_i has energy $E_{i,1}$ at some iteration t_1 and power $E_{i,2}$ at some iteration t_2 such that $E_{i,1} > E_{i,2}$, then a point $x \in A$ should appear farther away from v_i at t_2 compared to t_1 (assuming the agent's position p_i has not changed) since at t_2 the agent has less energy to reach that point. This will hence change the way in which A is partitioned for each agent.

5.4.4.4 Example in MATLAB: The way this is implemented in Matlab is as follows: If you choose algorithm type = 2, then the distance between (as it is currently) will calculate the distance d between an agent's position p_i and a point $x \in A$, given an energy content E_i and iteration number j as follows:

$$d = \frac{\|p_i - x\|}{E_i(j)}$$

where $\|.\|$ is the Euclidean distance. We assume that $E_i(j)$ is a function of the iteration number since the energy, as described in section (xviii), decreases after the agents have moved. In the code in MATLAB (in the function PlayButtonPushed), we have initiated the energies of the agents as percentages of their total. Hence, agents except for 2 and 3 (which have lower energies) start with 100% (the entries to E are 100) of their total energy (choosing 2 and 3 is just for an example). This will cause the function to assign points to act in the way described in the example above 5.4.4.3. In order to decrease energy, arbitrary parameters regarding the units of space and time were chosen in the function move_agents (you would need to change these to gear them towards your application if you wish to use this), which decreases the energy. The energies are then decreased using $E(i)_{j+1} = E(i)_j - 100 * \frac{\frac{1}{2} m_i v_i^2}{E(i)_{total}}$, where $E(i)_{total}$ is the total energy of the agent (that is, the energies are represented as percentages of the total). The function in which the energy changes is also particular to your application, and should be changed accordingly. Depending on how you wish to decrease the energy, you would need to change different functions that are most suitable to your application. However, if you want to change the energy as described above or simply change parameters such as the units of time, space, or the agent's mass or total energy, then this can be done in the function move_agents. If you run the app with these parameters (i.e. setting algorithm type to 'custom'), you will see in the figure Assigned Points (the 3rd plot in the app), that agents 2 and 3 do not have many points (if any) assigned to them, even though points exist within their radii and are closer in terms of the Euclidean distance.

5.4.5 Velocity (function: velocity_fun)

5.4.5.1 Description: We have included two possible ways in which agents move towards their assigned centroids as described in section 5.2.1. However, the way in which they move depends on your application. The velocity may then depend on other parameters.

5.4.5.2 How to Change It: The output of velocity_fun is the distance in the x and y directions that the agents are to move after one iteration, i.e. Δx and Δy. You can change the way in which they move towards their centroids within the function under the section "Editable velocity".

5.4.5.3 Example: Assume that your application involves cleaning up ocean oil spills. Suppose that the robots that do this can only collect oil at certain rates, and they do this as they move through the area of interest. Although the agents should move to central positions (depending on your Density), it may be beneficial that the agents do not move immediately to these centres so that they can clean up oil as they move. Therefore, the velocity of the agents should depend on the Density. That is to say that if they are in an area of high density, then depending on the rate at which they can clean up oil, they should move slowly through this area (perhaps you could consider an 'area' to be some radius around the agent, perhaps the radius of observation itself or smaller). On the other hand, if the amount of oil is small or there is no oil in some area, they should move quickly through this. A way to do this would be to have Density as an input to velocity_fun and make the velocity of an agent proportional to the Density (perhaps with some maximum depending on the rate at which the agents can collect oil).

5.4.6 Cost Function 2

5.4.6.1 Description: The Regular Algorithm has, as described in section 5.2.2, the total distance traveled by all agents plotted on the second cost function. Depending on the application, you may wish to keep track of other parameters.

5.4.6.2 How to Change It: Throughout the functions used in MATLAB, distance traveled is used to output to the appropriate plot. Look through PlayButtonPushed to see how it is plotted. First, you need to output as a vector the data you want to plot; the i^{th} row in this vector will have the data you want to plot for iteration i. Note that this vector must have a new row for every iteration. Use horzcat to do this. If you want to plot to an axis within the app, you can do this using plot(app.UIAxes2, ...), where the first argument of the plot function is the axes within the app that you are targeting. You can also choose to create a new plot in a figure window in the same manner as you would in a MATLAB script. We recommend to look through the code of PlayButtonPushed so that you can see how plotting is done in more detail. Also note that iteration number is not a single value, but a vector of the same size of YourDataVector, where row i contains the value i. Also note that since there is no specific place of where to change this, there is no instance when algorithm_type will be needed. If you don't change the plots, then the total distance traveled each iteration will be displayed (you must input the names of the y-axis and title into the app regardless).

5.4.6.3 Example: You may want to plot the energy expended by each agent separately. Depending on how you model the power (this could depend on the distance they travel, or the velocity at which they travel every iteration, or some other parameter), you can create a function that calculates this; currently this is done in as described in section 5.4.4.4, but you may wish to change this. Then you would plot N lines to display how the power of each agent changes over every iteration. You would plot these lines by plotting one YourDataVector (as described in the above section) for each agent against iterations. Look to the Percent Covered section of the PlayButtonPushed function for an example of how to create such a plot.

5.5 Generic Example

Here we state some parameters that can be input to the app so that you can see how the algorithm works:

Number of Agents	10
Side Length	5
Partition Number	5
Radius of Observation	2
Radius of Communication	3
Velocity	Proportional Velocity
Maximum Velocity	0.2
Density	Function
Density Function	`abs(sin(x)+cos(y))`
Density Type	Static
Delay	5 (this will not be used)
Algorithm Type	Regular Algorithm
Agent Starting Positions	Randomize
Name of Cost Function 2	Distance
Cost Function 2 y-axis	Distance Moved

6 Working with and Creating Apps

6.1 Altering the Existing Apps

Once you've explored the capabilities of the apps and you are ready to begin re-purposing an app for your needs, you can begin editing apps in the app designer window by clicking on the Code View tab. To access your code in app designer view, enter "appdesigner" in the MATLAB Command Window. Once the app designer has launched, you can open your apps in it.

With all of the apps, the main body of the code can be found in the PlayButtonPushed callback function. This is where you'll want to start.

If you choose to work with the existing apps, then there are two ways you will need to alter the code. Some alterations will require directly changing aspects of the default code in PlayButtonPushed, while others will necessitate the creation of new functions which will be called within the main body of PlayButtonPushed. When writing these functions, you may choose to write them as scripts within MATLAB's traditional script editor, or you could create local functions within the app itself, which can be done by pressing the Private Function button under the Editor heading in Code View of the app. Both of these options will achieve the same results. Within the code of these apps, you will notice commented sections which indicate where to put and how to construct functions you may want to implement, such as cost functions or system perturbations.

If you are working with the Lloyd's app, then you will likely need to alter existing functions within the Lloyd's app package. More information on this process can be found in the Lloyd's Algorithm section and the Lloyd's Algorithm Supplemental Documentation.

To save time, you may want to input your desired position, delay, offset, radius, etc. values in the 'startupfcn' function within the app file, as well as in the 'ResetButtonPushed' callback function, so that the app always returns to your default values. You could also choose to change the parameters with which agent data is randomized. The default random agent data generation will place agents between 0 and 10 and will not create stubborn agents for the formation algorithm.

6.2 Creating Your Own Algorithm/App

For a full picture of how to use MATLAB's App Designer, please see the MATLAB App Designer web page. Here are a few important things to keep in mind:

- Look at the existing algorithms and think about how you can reshape and appropriate aspects from each to use in your dynamics.
- Make sure you consider the real-world constraints your project would face. Your simulation should consider energy use, sensor range, limits on time-steps, or other physical limitations that affect your system's dynamics.
- Your application will likely require the use of an adjacency matrix which records how agents communicate with each other. Consider how you define communication channels between agents, and how you may want to alter how interactions affect agents depending on distance or other metrics.
- If you are writing your own app from scratch, it may be beneficial to write your initial code as a MATLAB script prior to making an app. This way you can take advantage of your MATLAB Workspace which lets you keep track of values, array sizes, and variables more easily than in the App Designer, where you must print these values to the Command Window if you wish to view them. This makes debugging much simpler, and the code can easily be transfered into the App Designer when you are ready.
- Make sure your code exports data in the proper format if you intend to use it in conjunction with the Robot Execution App or the real course robots. The data should be formatted in columns as: Agent 1 x-position, Agent 1 y-position, Agent 2 x-position, etc. Name your agent data variable "H" and then use the command "`save('agentData.mat','H')`"

6.3 Common Problems

- Ensure you do not change the file name of the app in the downloading process. If the app is renamed in any way, it may fail to open.
- If you are using local functions within the app, make sure you use "app" as the first argument in the function.
- If the Lloyd's app is giving you troubles, a common issue is that the position values are greater than the side length (or negative). This should not occur as currently constructed, unless the user manually changes position values. Another issue could arise if a function that produces negative values is used for the density.

7 Robot Execution App

While your application may achieve success within the ideal conditions present in the simulations, real physical agents, be they robots, drones, submarines, or even opinions in many cases, have restrictions on the ways in which they move. Your simulation may use a cost function to consider depleting energy sources for your agents, but the motion of these agents can be restricted in other physical ways. For instance, the real robots used in the course have the nonholonomic constraint that they can not move perpendicular to the direction in which their wheels face, and so they will require time to turn to face the point they are traveling to. Similar constraints would apply if you were working with airplanes, who require time and space to change directions. The ideal agents within the apps do not have orientations, and don't exhibit these physical limitations. The simulated robots must also travel at a constant speed, unlike the proportional velocities exhibited within some simulations.

The Robot Execution App allows you to use data taken from any of the algorithms (or your own custom algorithm) to visualize how the two-wheeled course robots would behave when executing your algorithm. The execution of your algorithm by the robots will not perfectly match your previous simulations with the apps. You can play around with this app to explore how these physical constraints affect your system, see how the initial angles of the robots could be optimized for your application, and to compare this simulation's performance to the actual robots. Doing this will allow you to see how translating a basic algorithm to an ideal physical simulation (and then to real world execution if you use the code on the real robots) adds complexities and problems to a project. Understanding these issues could be very important in your final report, depending on your application.

The app window can be seen in Figure 5. The relevant fields you can change are:

1. Number of Bots: The app lets you choose the number of robots (agents) used in your simulation. You should choose the same amount that you ran the simulation with. If you choose fewer, the algorithm will only run with the first n agents, where n is the input value.

2. Icon: You can change the icon from a robot to a plane or submersible, but this effect is only aesthetic. The physical constants that are particular to the course robots still apply.

3. Desired Velocity: The desired velocity allows you to choose the speed at which the robots travel. The robots have a maximum linear speed of approximately 0.388 m/s, although many speeds over 0.35 m/s will result in saturating the robot controller, resulting in uncontrolled motion. You should keep this value around 0.3 m/s. The units on the plot grid are assumed to be meters by default.

4. Data File Name: This is where you enter the name of the simulation data file you'd like to use. Make sure it is in the current folder you are working with in MATLAB (you can do this by checking the main MATLAB window). The app is designed to read .mat files; make sure you enter the correct file extension along with the file name. The default algorithms will export .mat files named "agentData.mat".

5. End Time: The end time lets you choose how long (in real world seconds) you would let the algorithm would run (NOTE: the plotting runs slower than real world time). This value is independent of how long your original simulation ran.

6. Plot Path: This will commence the execution of the simulation

7. Ready/End: When you want to plot the path, make sure the switch is in the "ready" position. To stop plotting and/or reset the simulation, place the switch in the "end" position.

8. Reset: This will clear the figures and prepare the app to run a new simulation.

9. Algorithm: You can choose the algorithm you used; if you created your own algorithm, make sure the .mat data is formatted in columns as: Agent 1 X Positions, Agent 1 Y Positions, Agent 2 X Positions, Agent 2 Y Positions, etc. Also make sure the data variable you're saving to the .mat file is named "H". The algorithm type mostly affects the aesthetics of the plot, but it is also used to determine how to interpret the data.

10. Agents Drive Backwards: You can choose whether the agents will be allowed to drive backwards, based on whether this would be practical in your application.

11. Show Errors and Show Control Inputs: These two check boxes allow you to decide if you want to see the errors and control inputs for each robot over the course of your simulation.

12. Initial Angles: You can change the initial angle of each robot. The angle should be between -π and π and represents the direction the robot faces, measured from the positive x-axis. Pressing "Randomize Angles" will create random values.

The robots move point to point based on the input data. For efficiency, the app sorts through the data and discards points that are too close together. The error graph lets you see the distance between each robot and its desired point over time. When the error drops to zero, the robot has reached its final destination. The algorithm considers that a robot has reached a desired point if it is within 15 cm of that point (since it is not possible to steer a robot to an exact location, we need to place an acceptable zone around each point). The control inputs graph lets you see the controls for each wheel for each robot, where the inputs are in the range [-255, 255], and negatives indicate backwards spin. If you see control inputs excessively spiking or flatlining at +/- 255 this means the controller has been saturated, and your desired velocity should be reduced.

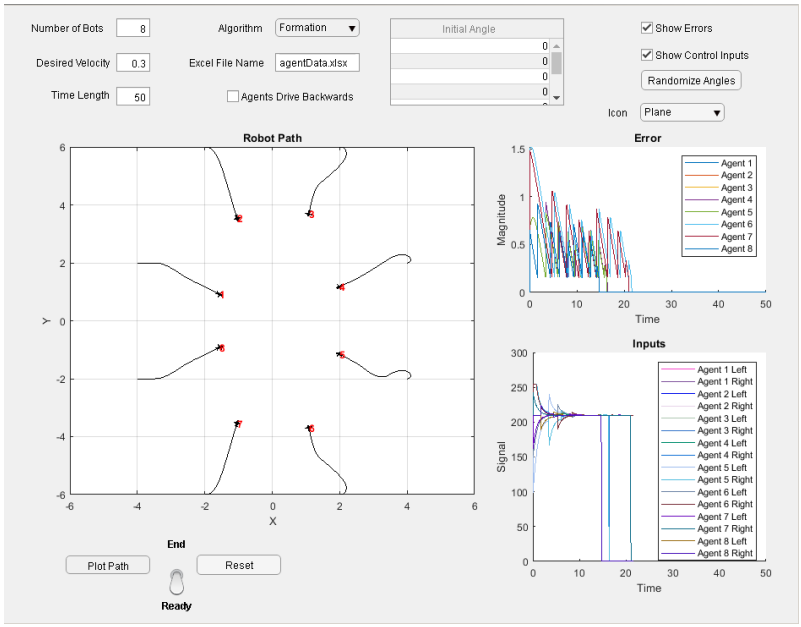

Figure 5: The robot execution app being used with a formation algorithm simulation