APSC 200 P2: Course Manual

Department of Mathematics and Engineering
Queen's University

August 21, 2021

Contents

1 Introduction

With continued technological advances, the use of multi-agents systems to solve complex problems is becoming increasingly feasible. From self-driving vehicles to search and rescue drones, the ability to have independent communication between agents is critical to the success of these systems. Four group formation algorithms will be presented in this course, which include the formation algorithm, the flocking algorithm, Hegselmann-Krause opinion dynamics, and Lloyd's algorithm. For your project, you will select an area of application that requires the agents to communicate with each other and/or converge in a decentralized manner using one of the four algorithms.

1.1 Project Overview

For the APSC 200 P2 project, you will be applying the engineering design process and mathematical concepts towards a topic of your choice involving group formation dynamics. As in APSC 100 and APSC 200 P1, **your primary task is to develop and showcase engineering design process skills.** This includes, but is not limited to, the development of a problem definition, background research, design criteria, proposed design alternatives, empirical data analysis, and justified decision making.

There is no physical prototyping in this course, nor are there labs to gather empirical data. Instead, you will be using the engineering design process to develop a model of your system using the mathematics provided. You will then test your system in a simulation environment to validate your design choices for the system. The 'shell' of the simulation environment (GUI, graphical plots, data output) are provided for you. **Your secondary task is to develop the simulation by implementing your algorithm in MATLAB.**

The simulation environment you develop will provide you with empirical data that can be used in decision making based on the design criteria you create. That means your design is only as accurate as your data, which is only as accurate as your simulation. At points during the project you may find that you need to make an assumption for your design. Any assumptions that you make must be well explained and justified.

You will begin developing the simulation in Week 3 of this project. It is important to verify that any code you write runs without errors. To do this, use the MATLAB Command Window and/or write a script to test your function(s) with dummy parameters. Then, verify the output of your function is what you would expect.

The mathematics in this course are well within the scope of APSC 171, 172, and 174. There are some elements of MTHE 237 and 280 depending on the algorithm you choose. This course will not teach you any new mathematical concepts, but will instead have you apply your existing mathematical knowledge. There are no submissions or marks for your technical work, besides anything included in your reports. As such, *don't let the coding become the main focus of your project!*

1.2 MATLAB Primer

Supplementary files in the form of a PDF and MATLAB Live Scripts (.mlx files) have been provided to assist you in the review and development of your MATLAB skills. The primer PDF contains a review of general programming concepts, syntax, MATLAB-specific concepts, and an overview of useful functions and features. In each section of the primer there are actual examples included to better illustrate the discussed concepts. Topics covered include arrays and matrices, calculating distances, the symbolic toolbox, plotting, and the MATLAB App Designer. These topics are supplemented through the use of MATLAB Live Scripts which allow you to interactively test your knowledge of the various concepts addressed in the primer PDF. For an optimal learning experience, it is recommended that you work through the Live Scripts while using the primer PDF as a reference such that you can familiarize yourself with the MATLAB environment and relevant concepts.

The table below provides a summary of the material(s) covered in each section of the primer.

Table 1: MATLAB Primer sections and the content covered in each section

Primer Section	**File Name**	**Topics Covered**
Introduction to MATLAB	N/A	• Command Window • Workspace • Editor (or Live Editor) • Current Folder • Search Documentation
Basic Concepts	basicConcepts.mlx	• Variables • Built-in Functions and Constants • Relational and Logical Operators • If Statements • Loops • Timing
Functions	functions.mlx	• Function Definition • Anonymous Functions
Arrays & Matrices	arrays.mlx	• Array Declaration • Array Indexing • Array Operations • Size and Dimension Manipulation • Array Logic • Cell Arrays
Distance & Norm Functions	distances.mlx	• Pairwise Distance • rangesearch function • vecnorm function
Symbolic Math	sym.mlx	• Symbolic Variables • Symbolic Functions
Plotting	plotting.mlx	• 2-D Line Plots • Plotting in 3-D
App Designer	appDesigner.mlx	• App Designer Basics

2 Formation Algorithm

The first algorithm that will be discussed is the formation algorithm. This algorithm is designed to have a system of agents converge to the same position. This is useful if the agents need to meet at a single point or surround a target.

2.1 Formation Consensus Dynamics

The formation algorithm is an averaging function that takes in agent positions and returns agent velocities. The continuous-time dynamics of the system can be described by the first order differential equation

$$\dot{\boldsymbol{q}} = -L \cdot \boldsymbol{q},$$

where $\boldsymbol{q}$ is an $n \times 2$ position vector containing the (x, y) coordinate pairings for each of the n agents in the system. In other words, the entry in the i-th row of the first column of $\boldsymbol{q}$ contains the x coordinate of i-th agent. Similarly, the entry in the i-th row of the second column of $\boldsymbol{q}$ contains the y coordinate of i-th agent. The Laplacian matrix, L, is a function of the adjacency matrix, A, and the degree matrix, D, and can be calculated by

$$L = D - A.$$

The adjacency matrix of the system is an $n \times n$ matrix given by

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

The degree matrix of the system is a diagonal $n \times n$ matrix given by

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

Hence, the Laplacian matrix for the system is an $n \times n$ matrix that has the property

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

In other words, the sum of each row is zero (and each column since the matrix is symmetric). This implies that $\dot{\boldsymbol{q}} = [0, 0, ..., 0]^T$ if $q_i = q_j$ for all $i, j \in [1, n]$. In order to model the formation algorithm, the system must be discretized from a continuous system. In the discrete time domain, the system updates the position of each agent using

$$\boldsymbol{q}(t + \Delta t) = \boldsymbol{q}(t) - L \cdot \boldsymbol{q}(t) \cdot \Delta t,$$

where Δt is a fixed time increment. In addition, the adjacency matrix for your system will most likely not be the same throughout the simulation. Entries in your adjacency matrix will be dependent on your system's current state and design constraints.

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 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

$$\boldsymbol{q}(t + \Delta t) = \boldsymbol{q}(t) - L \cdot [\boldsymbol{q} - \boldsymbol{g}] \cdot \Delta t,$$

where $\boldsymbol{g}$ is an offset vector of dimension $n \times 2$ containing the x and y offsets for all agents.

2.1.2 Introducing Delays

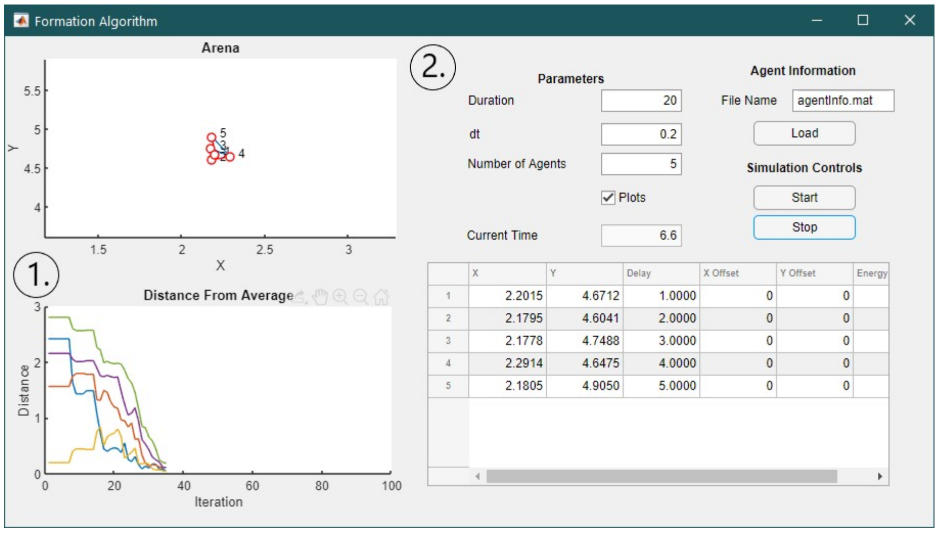
Delays are time steps where agents are idle before responding to changes in the system and moving to a new position. If an agent has an infinite delay, it is referred to as a “stubborn” agent and is described by $\dot{\boldsymbol{q}} = 0, \forall t \in \mathbb{R}_{\geq 0}$. Otherwise, agent i will update $\tau_i \in \mathbb{R}_{\geq 0}$ times-steps late when responding to a movement signal. With time delays, the position update equation is now

$$\boldsymbol{q}(t + \Delta t) = \begin{bmatrix} q_1(t - \tau_1) \\ q_2(t - \tau_2) \\ \vdots \\ q_N(t - \tau_N) \end{bmatrix} - L \cdot [\boldsymbol{q}(t) - \boldsymbol{g}] \cdot \Delta t.$$

2.2 Simulation App

The app provided uses the consensus dynamics to update each agent’s location for every iteration of the simulation. The critical components of the app’s functionality have been placed in external functions, which you will develop over the course of the project. The functions that you will be responsible for are discussed in detail in Section 2.2.5.

Upon simulation completion, the app will create an excel file named *agentData.xlsx* containing the positions, energy, and distance to mean for all agents, throughout the duration of the simulation. This data should be used to conduct further analysis to improve or validate your design.



Duration 20

dt 0.2

Number of Agents 5

[✓ Plots]

Current Time 6.6

File Name agentInfo.mat

	X	Y	Delay	X Offset	Y Offset	Energy
1	2.2015	4.6712	1.0000	0	0	
2	2.1795	4.6041	2.0000	0	0	
3	2.1778	4.7488	3.0000	0	0	
4	2.2914	4.6475	4.0000	0	0	
5	2.1805	4.9050	5.0000	0	0	

Figure 1: Screen capture of FormationApp.mlapp with default settings

2.2.1 Plots

Area 1 of Figure 1 displays the plots section of the Formation app. This section contains two separate plots, the **Arena** plot and the **Distance From Average** plot.

The plot in the top left portion of the app window is the **Arena** plot. The position of each agent is displayed as a dot with a numerical label on the plot. The lines connecting agents represent the communication graph as determined by the adjacency matrix. This plot is updated every iteration of the simulation to visualize the movement of the agents throughout the simulation.

The second plot in the app window is the **Distance From Average** plot. Upon initiation of the simulation, the consensus (average) position is calculated. Then, the distance between each agent and the consensus position is calculated and plotted for every iteration. The resultant plot effectively tracks agents as they converge to the consensus position. The independent axis displays simulated time whilst the dependent axis plots distance between agent and consensus position.

2.2.2 User Controls

Area 2 of Figure 1 displays the user controls section of the Formation app. This section is divided into three sub-sections: the **Parameters** sub-section, the **Agent Information** sub-section, and the **Simulation Controls** sub-section.

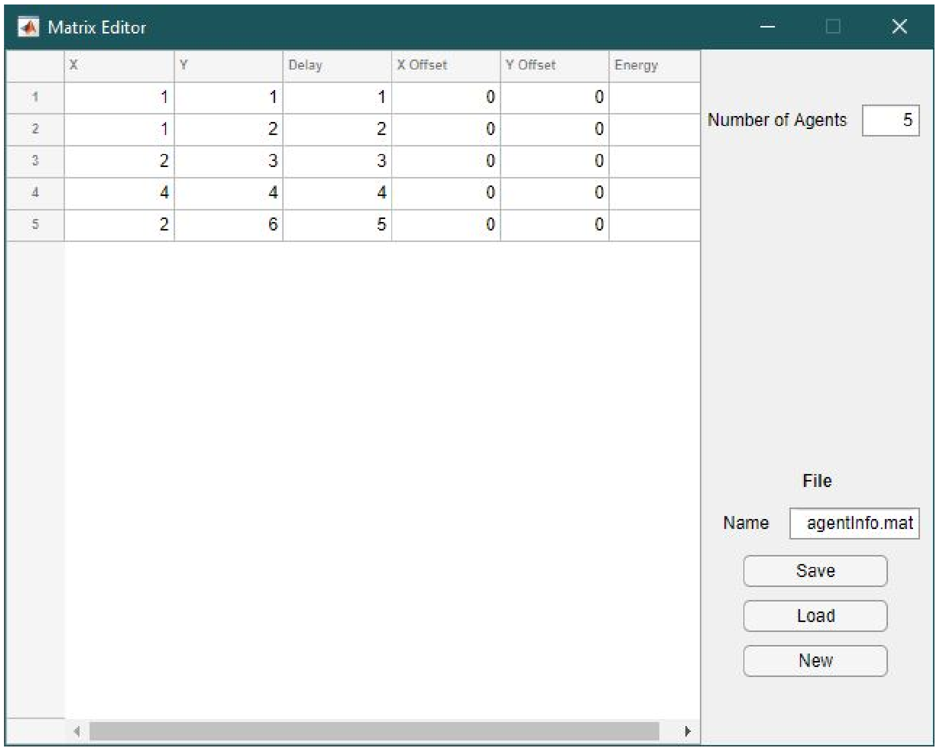
The **Parameters** sub-section is where you can alter the simulation parameters which affect the number of agents and the duration. The number of agents to be used in the simulation can be adjusted by the user by changing the value in the **Number of Agents** edit field. If plotting is enabled, the agent data table will be updated every iteration to reflect any changes in each agents' properties. The **Duration** edit field allows the user to adjust the simulation duration in arbitrary units of time. The **Current Time** edit field is non-editable and displays the current time while the simulation is running. The value of Δt in the position update equation in Section 2.1 can be modified via the **dt** edit field. The **Plots** checkbox, when enabled, will update the **Arena** plot, the **Distance From Average** plot, and the **Agent Information** table on every iteration. Disabling **Plots** will greatly increase the speed of your simulation. This is useful if you need to simulate for more than a few hundred iterations. The table in the app contains five columns: the current x and y positions; *delay*; *x offset*; *y offset*; and *energy*. This data can be edited directly or loaded from a file created in the Matrix Editor. See Section 2.2.3 for details.

The **Simulation Controls** sub-section contains the **Start** button which begins the simulation and the **Stop** button which stops the simulation before completion.

The **Agent Information** sub-section allows you to load in the initial conditions for the agents by specifying a .mat file name under the **Agent Information** heading and pressing **Load**, avoiding the need to manually re-enter data.

2.2.3 Matrix Editor

Using the **Matrix Editor Formation** app, you are able to create files for initial agent data. To do so, first enter the number of agents in the **Number of Agents** edit field. Next, edit the values in the table as you see fit. When complete, specify a filename including the *.mat* extension. Remember to enter this filename and press the **Load** button in the simulation app to load the data you have entered.


Matrix Editor

	X	Y	Delay	X Offset	Y Offset	Energy
1	1	1	1	0	0	
2	1	2	2	0	0	
3	2	3	3	0	0	
4	4	4	4	0	0	
5	2	6	5	0	0	

Number of Agents: 5

File

Name: agentInfo.mat

Figure 2: Screen capture of MatrixEditorFormation.mlapp with default settings

2.2.4 Parameters and Variables

i Number of Agents ($numAgents$): Total number of agents (N) used in simulation.

ii Agent Position ($agentPosition$): An $N \times 2$ matrix with the x-position of the i^{th}-agent in the first column of row i, and the y-positions in the second column, as below:

$$AgentPositions = \begin{bmatrix} q_x^1 & q_y^1 \\ q_x^2 & q_y^2 \\ \vdots & \vdots \\ q_x^N & q_y^N \end{bmatrix}$$

iii Adjacency Matrix (A): The $N \times N$ adjacency matrix:

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

iv Degree Matrix (D): The $N \times N$ degree matrix:

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

v Laplacian Matrix (L): The $N \times N$ Laplacian matrix:

$$L = D - A$$

vi Duration ($duration$): Total duration the simulation will run for.

vii Δt (dt): Amount of time simulated in each iteration.

viii Current Time (*time*): Current (simulated) time.

ix Positional Offset: The x and y positional offsets are saved within column 4 and 5 of the $N \times 6$ vector *agentData*. The positional offsets determine where agents will converge to with respect to the network average. Mathematically, this alters the position update equation as seen in Section 2.1.1 to be the equation

$$\boldsymbol{q}(t + \Delta t) = \boldsymbol{q}(t) - L \cdot [\boldsymbol{q} - \boldsymbol{g}] \cdot \Delta t,$$

where $\boldsymbol{g}$ is an offset vector of dimension $n \times 2$ containing the x and y offsets for all agents.

x Time Delay: The time delays are saved within column 3 of the $N \times 6$ vector *agentData*. The time delays determine the delay in an agent's reaction time when updating it's position. Mathematically, this alters the position update equation as seen in Section 2.1.2 to be the equation

$$\boldsymbol{q}(t + \Delta t) = \begin{bmatrix} q_1(t - \tau_1) \\ q_2(t - \tau_2) \\ \vdots \\ q_N(t - \tau_N) \end{bmatrix} - L \cdot [\boldsymbol{q}(t) - \boldsymbol{g}] \cdot \Delta t.$$

xi Energy: The energy metric is saved within column 6 of the $N \times 6$ vector *agentData*. You must determine the units of the energy that you wish to use and how you want to represent this in the energy array. The energy will decrease after every iteration by some parameter (you may choose how to represent this).

xii Agent Data (*agentData*): The agent data matrix is the $N \times 6$ matrix which contains the x positions of the agents in the 1^{st} column, the y positions of the agents in the 2^{nd} column, the delay values in the 3^{rd} column, the x offset values in the 4^{th} column, the y offset values in the 5^{th} column, and the energy values in the 6^{th} column.

2.2.5 MATLAB Functions

The following MATLAB functions that you will need to write in order for the simulation app to function are described below. The order that functions are listed is the order that you will be creating them during the project.

calcA.m – calcA.m is used to calculate the adjacency matrix based on the agent states and any relevant design constraints. The calcA function will take as input *agentPosition* and will output A. A's calculation will be determined by you in Week 2.

calcL.m – calcL.m is used to calculate the Laplacian matrix for the current iteration of the simulation using the adjacency matrix. The CalcL function takes as input A and outputs L. L is calculated as seen in Section 2.1 with the equation

$$L = D - A$$

moveAgents.m – moveAgents.m is used to update the position of each agent using the consensus dynamics formula in Section 2.1. The moveAgents function takes inputs *agentData*, L, *time*, and dt, and outputs *agentData*. The positional data in *agentData* is updated as seen in Section 2.1 with the equation

$$\boldsymbol{q}(t + \Delta t) = \begin{bmatrix} q_1(t - \tau_1) \\ q_2(t - \tau_2) \\ \vdots \\ q_N(t - \tau_N) \end{bmatrix} - L \cdot [\boldsymbol{q}(t) - \boldsymbol{g}] \cdot \Delta t.$$

Furthermore, in week 4 you will be tasked to model and introduce a cost function involving the energy parameter into the moveAgents function.

2.3 Example: Formation

This section provides a sample P2 project using the formation algorithm. This example is broken up into what is to be completed each week of the project to provide a reference to what will be expected in your projects. You can not use this application area for your project.

2.3.1 Week 1

Area of Application
The application area that has been selected involves using a number of submersible vehicles to conduct research on coral bleaching in a shallow sea. There are a number of submersibles sampling coral over an area of several squared kilometers. The submersibles take samples until their on-board storage containers are full. Instead of wasting time and resources by having each agent return to shore to deliver its samples, a ship is to be sent out to the first agent's location. The other agents are to meet the ship at the first agent's location. However, communication underwater is difficult and costly, which restricts the communication abilities between agents.

Algorithm Selection
With a potential area of application selected, each of the possible group dynamics algorithms are examined in more detail. Based on the application area's requirement that the agents converge to a single location, it was determined that the formation algorithm was the most suitable choice for the project.

Pitch Presentation
With the area of application selected, a brief pitch presentation of the application area is created. This presentation provides a high-level overview of the area of application and how a group-formation algorithm could be applied to model the design solution.

2.3.2 Week 2

Proposal Report
With the application area and formation algorithm selected, the proposal report can now be written. This report includes all the standard items listed on the rubric, such as an executive summary, introduction with background information, discussion, project plan, etc. The problem definition specifies that a performance analysis needs to be conducted on several models of submersible robots to determine the efficacy and economic viability of each system model. The main stakeholders are identified and include: the research group, the submersible vehicle company, state/federal governments, and more. Design metrics include time to convergence, travel range, redundancy in communication (i.e. fail-safe system), cargo capacity, initial cost(s), operational cost(s), maintenance cost(s), and robot lifespan.

Adjacency Matrix

Various communication methods between agents will be developed and then tested using the simulation. Each method will be defined using a number of design parameters. These parameters include the number of agents, the number of possible communication pairings for each agent, the distance between a pair of communicating agents and the radius of communication for each agent. These exact values are not defined at this point and will be determined through simulation testing and evaluation against a set of design metrics (these metrics will be defined in Week's 4 and 5).

The first potential communication method has the agents form a communication chain such that each agent can only communicate with the agent before itself. The exception to this chain will be with agent 1, which will not have a communication partner. This is done so that it does not move from its starting location, thus acting as the point of convergence for the other agents.

Another method is having each agent be in communication with the agent before and after itself. Like the first method, the first agent will not have any communication partners to ensure all agents converge to its starting position. For the last agent in this communication chain, it will only have one communication partner, the agent before itself.

For both methods, the strength of communication between agent i and its communication partner is determined using

$$A(i, i-1) = \begin{cases} \frac{(d_{(i,i-1)} - r_c)^2}{r_c^2}, & d \leq r_c, \\ 0, & \text{otherwise,} \end{cases}$$

for agent $i-1$, the agent before itself, and

$$A(i, i+1) = \begin{cases} \frac{(d_{(i,i+1)} - r_c)^2}{r_c^2}, & d \leq r_c, \\ 0, & \text{otherwise,} \end{cases}$$

for agent $i+1$, the agent after itself. The radius of communication for the agent is denoted by r_c. For a communication pairing with the agent before itself, the distance between the two agents is denoted by $d_{(i,i-1)}$. Similarly, for a communication pairing with the agent after itself, the distance between the two agents is denoted by $d_{(i,i+1)}$. This method could be extended to include more communication partners for each agent up to all the agents being in communication with each other.

2.3.3 Week 3

MATLAB Coding
The *calcA.m* function is used to calculate the adjacency matrix for the system given its current state and design constraints. The resulting adjacency matrix is then used in the *calcL.m* function to calculate the resulting Laplacian Matrix.

Design Process
Continue researching parameters that will be used to evaluate the final design. These metrics could include the maximum area of coverage by all the robots in the system, various performance specifications of the robots, the time to convergence and more. The parameters that will be evaluated using the simulation should also be noted. A method of scoring potential design solutions using the defined metrics should also be developed.

Begin research for the Triple Bottom Line (TBL) analysis of the project. This analysis focuses on the social, economic and environmental impacts that the final design solution will have. This research could include; the financial impact on the research group in purchasing the final design solution, the environmental impact of the submersible robots and many more. Determining how your design will influence these factors is important to keep in mind. Some areas may be important enough to include as a metric for design evaluation!

Reports
Complete the first progress report as outlined in the report template. This report is meant to highlight the items you have accomplished to date. The next steps for the project and areas of current or anticipated challenge should be noted with a plan on how these challenges will be overcome.

2.3.4 Week 4

MATLAB Coding
Translate the position update equation for the formation algorithm into code in *moveAgents.m*. Within this

function, we can model propagation delays in the communication system by setting

$$\tau_i = 10\left(\frac{||\vec{q}_i(t) - \vec{q}_j(t)||}{1000}\right),$$

so that there is an extra 10 second delay for information sent from agent j to agent i for every 1000 meters between them.

Design Process
Energy levels within each submersible must be considered in the model as agents need to reach the ship before running out of fuel. The metric for an energy resource could be actual fuel energy, time, distance, etc., so long as the depletion of energy can be reasonably modelled and incorporated into the *moveAgents.m* function. Conduct the necessary research to determine what the energy function will look like and the energy capacity available in each robot. The energy model for this example will take into account energy depletion due to work by drag and change in kinetic energy. Suppose after research that the SR1 has a full charge battery capacity of 150kJ, whereas the SR2 has 200kJ.

Decide on a decision making strategy for your final design. For example, if using an evaluation matrix, be sure to justify the design metrics and their weights through additional research.

2.3.5 Week 5

MATLAB Coding
Now comes time to integrate the energy function into *moveAgents.m*. The remaining energy levels for each agent can be modelled by decreasing their energy based on the work done at each time step

$$E_i(t+\Delta t) = E_i(t) - F_{d_i}(\vec{q}_i(t+\Delta t) - \vec{q}_i(t)) - P\Delta t,$$

where E_i is the energy remaining for the ith agent, F_{d_i} is the force of drag on agent i, and P is the constant power drawn per unit time while the robot is in use. 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. The drag for a body can be modelled using

$$F_{d_i} = C_d \frac{1}{2}\rho V_i^2 A_i,$$

where C_d denotes the drag coefficient for the submersible [1]. The submersible's velocity in m/s and cross-sectional area perpendicular to direction of the submersible's travel are denoted by V_i and A_i, respectively. The density of water is represented by ρ.

Design Process
The simulation should now be operational. Testing of the system with various sets of design parameters can be run with the simulation with the system's performance then being evaluated against the set design metrics. For example, testing for a radius of communication that provides a suitable level of performance to the system while remaining economically feasible could be now determined. Another example could be determining the optimum number of agents needed to provided the desired area of coverage.

Reports
Complete the second progress report as outlined in the report template. This report will be similar in structure to the first progress report.

Begin writing the final report detailing the Design Process, Design Solution and its evaluation against the Design Metrics, TBL Analysis and more. This report should include the revised sections from the proposal report based on the feedback given.

2.3.6 Week 6

Final Design
Review and finalize the evaluated design criteria for each design. Decide on a final proposed solution with

justification. Create figures and tables for use in the final report and presentation. Complete the final report and presentation. Pay significant attention to documenting the design process, with justification of each step along the way (evaluation matrices, design rubrics, etc.)

3 Flocking Algorithm

The second algorithm being discussed is the flocking algorithm. This 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

$$\dot{\boldsymbol{v}} = -L \cdot \boldsymbol{v}.$$

where the $n \times 2$ velocity vector, $\boldsymbol{v}$, is calculated by

$$\boldsymbol{v} = \dot{\boldsymbol{q}},$$

and the Laplacian matrix, L, is calculated by

$$L = D - A.$$

In the case of the flocking algorithm, the adjacency matrix, A, is calculated differently than in the formation algorithm. Unlike formation, where agents were either in open or closed communication with another agent, agents are now assumed to be in communication with each other, but the strength of the communication between agents is dependent on the distance between agents. In the flocking algorith, the entries for the adjacency matrix are calculated by

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

where K, σ, and β are user defined parameters. The parameter K denotes a proportional gain on the communication between the two agents. The parameter σ denotes a decrease in communication strength between agents. The parameter β denotes the rate at which the signal strength changes over distance. Furthermore, the variable d denotes the Euclidean distance between agents i and j and is calculated by

$$d = ||q_i - q_j||.$$

The degree matrix, D, is calculated in the same manner as in the formation algorithm by

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

In order to model the algorithm, the system needs to be translated into the discrete time domain from a continuous system. In the discrete time domain, the system updates the velocity of each agent using

$$\boldsymbol{v}(t + \Delta t) = \boldsymbol{v}(t) - L \cdot \boldsymbol{v}(t) \cdot \Delta t,$$

where Δt is a fixed time step.

3.1.1 Introducing a Leader

A leader is a single agent that is defined to follow a specified parameterized path independent of the velocities of the other agents in the system. The purpose of introducing a leader is to have the other agents follow the leader.

3.1.2 Introducing a Trigger Sequence

A trigger sequence, T, is a $1 \times T_{max}$ vector containing ones and zeros, where $T_{max} = \frac{duration}{\Delta t}$ is the number of iterations for which the simulation will run, and *duration* is the length of time for which the simulation will run. With a trigger sequence, the system updates the velocity of each agent using

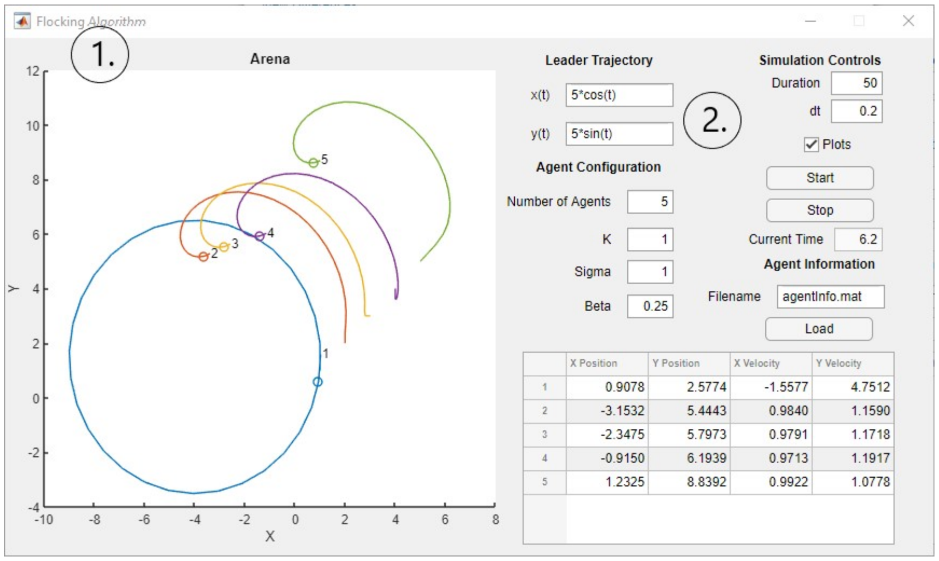
$$\boldsymbol{v}(t + \Delta t) = \begin{cases} \boldsymbol{v}(t) - L \cdot \boldsymbol{v}(t) \cdot \Delta t, & \text{if } T(t) = 1, \\ \boldsymbol{v}(t), & \text{if } T(t) = 0. \end{cases}$$

This sequence determines when the algorithm will allow the agents to communicate and update their velocities. If the time step that you set for your simulation is very small, it may not be possible or cost effective to have the agents update their velocities every iteration. You can use the trigger sequence to represent the real world limitations on the communication capabilities between agents. A possible area to explore is experimenting with the minimum number of communications your system requires to remain well connected (i.e. no rogue agents). The trigger sequence is one of the functions you will be designing for the Flocking Simulation's operation.

3.2 Simulation App

The app provided uses the consensus dynamics defined in Section 3.1 to update each agent's location for every iteration of the simulation. The critical components of the app's functionality have been placed in external functions, which you will develop over the course of the project. The functions that you will be responsible for are discussed in detail in Section 3.2.5.

Upon simulation completion, the app will create an excel file named *agentData.xlsx* containing the positions and velocities for all agents, throughout the duration of the simulation. This data should be used to conduct further analysis to improve or validate your design.



Leader Trajectory

x(t) 5*cos(t)

y(t) 5*sin(t)

Agent Configuration

Number of Agents 5

K 1

Sigma 1

Beta 0.25

Simulation Controls

Duration 50

dt 0.2

✓ Plots

Current Time 6.2

Agent Information

Filename agentInfo.mat

	X Position	Y Position	X Velocity	Y Velocity
1	0.9078	2.5774	-1.5577	4.7512
2	-3.1532	5.4443	0.9840	1.1590
3	-2.3475	5.7973	0.9791	1.1718
4	-0.9150	6.1939	0.9713	1.1917
5	1.2325	8.8392	0.9922	1.0778

Figure 3: Screen capture of FlockingApp.mlapp with default settings

3.2.1 Plots

Area 1 of Figure 3 displays the plots section of the Flocking app. This section contains a single plot, the **Arena** plot.

The **Arena** plot for the Flocking app displays the current position of each agent using a small circular marker with corresponding agent number. The coloured line trailing from each agents' current position marker indicates the path traced by that agent during the simulation thus far.

3.2.2 User Controls

Area 2 of Figure 3 displays the user controls section of the Flocking app. This section is divided into 4 sub-sections: the **Agent Configuration** sub-section, the **Simulation Controls** sub-section, the **Leader Trajectory** sub-section, and the **Agent Information** sub-section.

The **Agent Configuration** sub-section is where you can enter the design parameters for the **Number of Agents**, and **K**, **Sigma** and **Beta** that are used in the adjacency matrix calculation (see Section 3.1).

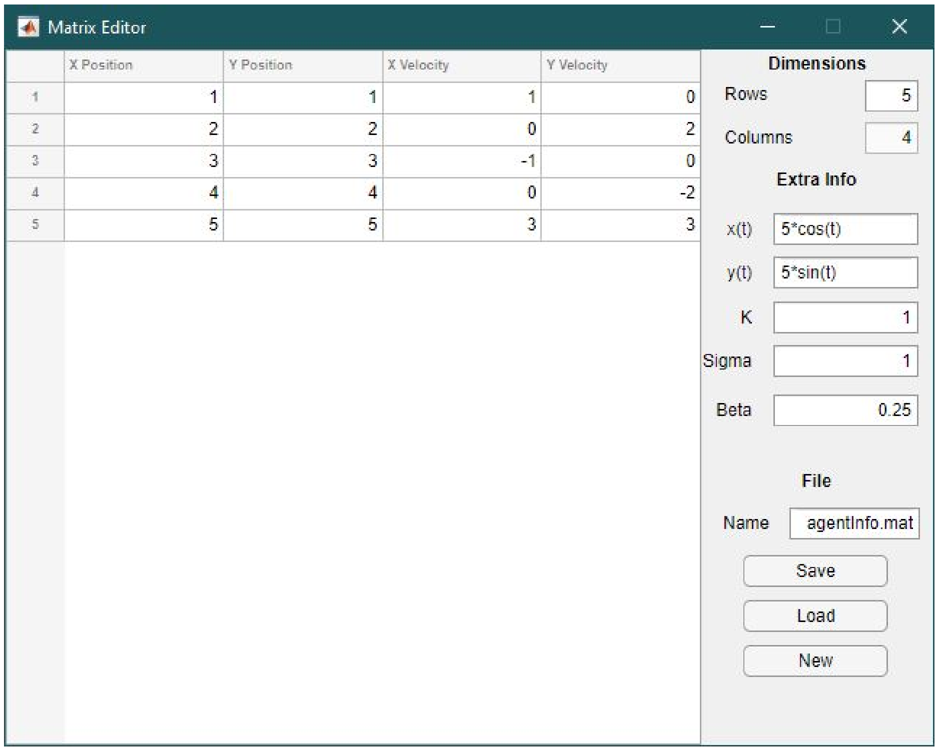
The **Simulation Controls** sub-section contains the following commands and input windows. You can adjust the value of Δt used in the position update equation for the flocking algorithm (Section 3.1) via the **dt** edit field. To change the length of time for which the simulation runs, adjust the value under **Duration**. The **Start** button begins the simulation and the **Stop** button stops the simulation from running. The **Plots** checkbox, when enabled, will update the **Arena** plot and the **Agent Information** table on every iteration. Disabling **Plots** will greatly increase the speed of your simulation. This is useful if you need to simulate for more than a few hundred iterations.

The parameterized path that the leader is to follow is entered under the **Leader Trajectory** sub-section. The parameterized equation for the x position of the leader is entered in the **x(t)** window and similarly **y(t)** for the parameterized y position equation of the leader. The parameterized equations must be written in terms of t.

The **Agent Information** sub-section allows parameters to be loaded from a *.mat* file created using the Matrix Editor. This file will contain: the initial x, y positions and velocities for each agent; the leader trajectories, $x(t)$ and $y(t)$; the proportional gain K; the communication noise *sigma*; the signal strength decay value *beta*. These parameters are loaded by pressing the **Load** button. Alternatively, this information can be entered directly into the app. Note that information entered directly into the app will not be saved to the *.mat* file. Instead, this is done using the Matrix Editor (See Section 3.2.3)

3.2.3 Matrix Editor

The **Matrix Editor Flocking** app can be used to create a matrix that stores the initial position and velocity information for each agent. The number of agents is reflected in the number of rows of the position velocity table. Additional information can also be stored in the .mat file by entering the x and y parametric equations for the leader trajectory and the K, *sigma*, and *beta* values in the appropriate text-boxes. The file name for the .mat file can also be altered to allow for multiple configurations to be created and stored.



	X Position	Y Position	X Velocity	Y Velocity
1	1	1	1	0
2	2	2	0	2
3	3	3	-1	0
4	4	4	0	-2
5	5	5	3	3

Dimensions

Rows: 5

Columns: 4

Extra Info

x(t): 5*cos(t)

y(t): 5*sin(t)

K: 1

Sigma: 1

Beta: 0.25

File

Name: agentInfo.mat

Figure 4: Screen capture of MatrixEditorFlocking.mlapp with default settings

3.2.4 Parameters and Variables

i Number of Agents ($numAgents$): Total number of agents (N) used in simulation.

ii Agent Position ($agentPosition$): An $N \times 2$ matrix with the x-position of the i^{th}-agent in the first column of row i, and the y-positions in the second column, as below:

$$AgentPositions = \begin{bmatrix} q_x^1 & q_y^1 \\ q_x^2 & q_y^2 \\ \vdots & \vdots \\ q_x^N & q_y^N \end{bmatrix}$$

iii K (k): The parameter K is a user defined constant scalar multiplier which denotes a proportional gain on the communication between agents.

iv σ ($sigma$): The parameter σ is a user defined offset value which denotes a decrease in communication strength between agents.

v β ($beta$): The parameter β is a user defined exponent value which denotes the rate at which the signal strength changes over distance.

vi Adjacency Matrix (A): The $N \times N$ adjacency matrix:

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

vii Degree Matrix (D): The $N \times N$ degree matrix:

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

viii Laplacian Matrix (L): The $N \times N$ Laplacian matrix:

$$L = D - A$$

ix Velocity Function ($velocityFunction$): The 1×2 symbolic expression in terms of symbolic variable t representing the parametric velocity functions $v_x^L(t)$ and $v_y^L(t)$ of the leader.

$$velocityFunction = \left[v_x^L(t), v_y^L(t)\right]$$

x Duration ($duration$): Total duration the simulation will run for.

xi Δt (dt): Amount of time simulated in each iteration.

xii Current Time ($time$): Current (simulated) time.

xiii Leader Velocity ($leaderVelocity$): $(duration/dt + 1) \times 2$ vector of the velocity function evaluated from $t = 0$ to $t = duration$ in discrete time steps of dt. Hence, the resultant vector can be shown as:

$$leaderVelocity = \begin{bmatrix} v_x^L(0), & v_x^L(0) \\ v_x^L(dt), & v_x^L(dt) \\ v_x^L(2 \cdot dt), & v_x^L(2 \cdot dt) \\ & \vdots \\ v_x^L(duration), & v_x^L(duration) \end{bmatrix}$$

where $v_x^L(t)$ is the x velocity of the leader agent at time t and $v_y^L(t)$ is the y velocity of leader agent at time t.

xiv Trigger Sequence ($trigger$): The trigger sequence is a $1 \times T_{max}$ vector containing ones and zeros which convey the logic for when to update agent velocities. If $trigger(t) = 1$ then the agents' velocities will be updated in following iteration. If $trigger(t) = 0$ then no velocity update will occur in the next iteration. For more information on trigger sequences, see Section Section 3.1.2.

xv Agent Velocity ($agentVelocity$): The $N \times 2$ vector of agent velocities where the first column represents the x velocities of agents 1 through N and the second column represents the y velocities of agents 1 through N as such:

$$agentVelocity = \begin{bmatrix} v_x^1(t), & v_y^1(t) \\ v_x^2(t), & v_y^2(t) \\ & \vdots \\ v_x^N(t), & v_y^N(t) \end{bmatrix}$$

where $v_x^i(t)$ is the x velocity of agent i at time t and $v_y^i(t)$ is the y velocity of agent i at time t.

3.2.5 MATLAB Functions

You will need to create the following MATLAB functions in order for the simulation to function. The order that functions are listed is the order that you will be creating them during the project.

calcA.m – calcA.m is used to calculate the adjacency matrix for the current iteration of the simulation using the user defined parameters and the agents' positions. The calcA function takes inputs K, $sigma$, $beta$, and $agentPosition$, and outputs A. A is calculated as seen in Section 3.1 with the equation

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

calcL.m – calcL.m is used to calculate the Laplacian matrix for the current iteration of the simulation using the adjacency matrix. The CalcL function takes as input A and outputs L. L is calculated as seen in Section 3.1 with the equation

$$L = D - A$$

calcLeaderVelocity.m – calcLeaderVelocity.m is used to calculate the velocity of the leader agent for all iterations of the simulation. Note that this function is called once at the beginning of the program, and the leaderVelocity returned should contain the leader's velocity at each discrete time step. The calcLeaderVelocity function takes inputs $velocityFunction$, $duration$, and dt, and outputs $leaderVelocity$. leaderVelocity is calculated by evaluating the pair of functions $\left[v_x^L(t), v_y^L(t)\right]$ in $velocityFunction$ $\forall t \in [0, T_{max}]$ where $T_{max} = duration/dt$.

trigger.m – trigger.m is used to calculate the trigger sequence discussed in Section 3.1.2. The trigger function takes as input $time$ and outputs $trigger$. $trigger$'s calculation will be determined by you over the course of the project.

updateVelocity.m – updateVelocity.m is used to update the agents' velocities using the dynamics governed by the flocking consensus dynamics. The updateVelocity function takes inputs L, dt, and $agentVelocity$, and outputs $agentVelocity$. $agentVelocity$ is calculated as seen in Section 3.1 with the equation

$$\boldsymbol{v}(t + \Delta t) = \boldsymbol{v}(t) - L \cdot \boldsymbol{v}(t) \cdot \Delta t$$

3.3 Example: Flocking

This section provides a sample P2 project using the flocking algorithm. This example is broken down into what is to be completed each week of the project to provide a reference as to what will be expected with your project. You can not use this application area for your project.

3.3.1 Week 1

Area of Application
The application area that has been selected involves some rovers following another rover through the vast expanses of Mars to conduct surveying. A new fleet of rovers is to be designed that will have 'leader' and 'follower' rovers. The 'leader' rovers with a highly accurate GPS system will be capable of effectively navigating and returning to base. The 'follower' rovers will not have GPS unit but can communicate with the leader robot. Without a leader robot, the 'follower' robots will be helpless in returning to base.

Algorithm Selection
With a potential area of application selected. The four possible group-dynamics algorithms were examined in more detail. The applicability of each algorithm was conducted to determine the most suitable algorithm for the project. Based on the application area's requirement that the agents follow a leader, the flocking algorithm was determined to be the most suitable option.

Pitch Presentation
With the area of application selected, a brief pitch presentation of the application area was created. This presentation provided a high-level overview of the area of application and how the group dynamics algorithms could be implemented in the design solution.

3.3.2 Week 2

Proposal Report
With the application area and flocking algorithm selected, the proposal report could then be written. This report includes the standard items listed in a design report; executive summary, introduction with background to the application area, discussion, project plan, etc. The report includes a problem definition

specifying a fleet of robots that meets the desired performance abilities and satisfies the communication requirements. The main stakeholders were identified and include; the astronauts on Mars, the rover development company and the federal government or space exploration corporation. Preliminary design metrics to evaluate the effectiveness of the final design solution are also included in the report. This will involve identifying parameters in the design that can be varied. Research will be conducted to determine the various parameters and the ranges these parameters can be based on the application area.

Adjacency Matrix
The flocking simulation will be used to help determine if the proposed design will meet performance requirements. To model this system, a few things must be considered. First we redefine the adjacency matrix, A, to be:

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

This makes it so that the agents can only communicate if they are within some set radius of communication, r_c, and that the signals will be stronger as the follower agent becomes farther 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 farther off course. Research will need to be conducted to determine the acceptable ranges for K, σ, β and r_c.

3.3.3 Week 3

MATLAB Coding
With the adjacency matrix equation determined, it can now be translated into equivalent code to complete the *calcA.m* function. In addition, the equations for determining the Laplacian matrix, L, can also be translated into code for the *calcL.m* function.

Design Process
Continue establishing design metrics to be used to evaluate aspects of the final design solution. An example of one design metric will be the maximum time between communications. The longer the time between communications can be, the more cost effective the system will be by reducing the cost of the communication equipment required.

At this point in the project, research for Triple Bottom Line (TBL) analysis for the project; social, economic and environmental considerations should begin. This research could include; budget limits of research groups for design solution, performance, cost and environmental impact of the new robot, cost of communication equipment to be used on robots and many more. Determining how these factors will influence the design choices you make is important to keep in mind.

Reports
Completed the first progress report. This one-page report highlights what has been accomplished to date, what challenges the team were facing and the strategy that will be used to overcome these challenges. An outline of the team's next steps for the project should also be included.

3.3.4 Week 4

MATLAB Coding
Wrote code for the *calcLeaderVelocity.m* function that uses the derivative of the parametric equations describing the leader's trajectory and calculates the corresponding velocity for each iteration of the simulation. As an example, the leader's trajectory can be defined to be $v_L = (0.1t, 0.1t)$. This straight line trajectory has the leader moving at 0.141 metres per second.

Introduced a trigger sequence to the algorithm through the *trigger.m* function to determine when the velocities of each agent will be updated. As an example, the trigger sequence is defined

$$T(t) = \begin{cases} 1, & t \equiv 0 \pmod{t_{update}}, \\ 0, & \text{otherwise.} \end{cases}$$

This establishes that the velocity of each agent is to be updated after some specified time period, t_{update}, between communications. This value will be varied during testing to determine the minimum number of communications required for consistent performance. The third function to be written is the *updateVelocity.m* function that updates the velocity of each agent. We then account for behaviour in which a rover is navigating over uneven terrain by adding noise, $\boldsymbol{e}(t)$, to the velocities of the follower rovers

$$\boldsymbol{v}(t) = \boldsymbol{v} + \boldsymbol{e}(t),$$

where for each iteration $\boldsymbol{e}(t)$ is a 2×1 array of values ranging from - 0.05 to 0.05 following a Gaussian distribution.

Design Process
Developed methods to evaluate design alternatives. This could include creating an evaluation matrix. If using an evaluation matrix, be sure to justify the categories made and the weights assigned to each category (may require additional research). Energy levels within the individual rovers will be an important consideration, as they need to have returned to base before they run out of energy. This will require further research as to what factors lead to the rover's energy being depleted. Once theses parameters are determined, an energy function can be incorporated into the *updateVelocity.m* function to improve the robustness of the design. Another factor to consider is that some rover movements such as accelerating/decelerating or turning could be more energy intensive. Introducing an energy function to account for this would improve simulation accuracy. These energy and cost functions will be added to *updateVelocity.m* in the following week.

At the end of this week, the flocking simulation app is functioning and preliminary testing can begin.

3.3.5 Week 5

MATLAB Coding
Based on research results, an appropriate energy and cost function can be incorporated into the *updateVelocity.m* function.

Design Process
With the simulation now complete including the energy and cost functions, testing of the system under various parameter settings can be conducted. The ranges that various parameters can vary within have been well researched in the weeks prior. Using the design metrics established in previous weeks, the most suitable design solution can be selected. Be sure to use *quantitative* design metrics when evaluating the effectiveness of a design. The evaluation of potential designs can then be compared against each other through the use of evaluation matrices, design rubrics, etc.

Reports
The second progress report was submitted. This one page report is similar in content and format to the progress report submitted in Week 3. Highlight any remaining challenges and/or tasks for the project.

Work on the final report also began detailing the Design Process, Design Solution and its justification, TBL analysis and more. This report should include the revised sections from the proposal report based on the feedback returned.

3.3.6 Week 6

Final Design
Complete any remaining tests and finalize the design specifications for the final design solution. Generate supporting materials (plots, tables, etc.) that can be used in the final report and presentation. Complete the final report and presentation for the project.

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 of $r_{c_i} \in \mathbb{R}_{>0}$. The position of the agents is stored in $\vec{q}$, a $n \times 1$ matrix, where n is the number of agents. The ith agent's position is denoted as q_i. The adjacency matrix is an $n \times n$ matrix with each entry determined using

$$A(i,j) = \begin{cases} 1, \text{ if } |q_i - q_j| \leq r_{c_i}, \\ 0, \text{ if } |q_i - q_j| > r_{c_i}. \end{cases}$$

In situations when $r_{c_i} \neq r_{c_j}$, A will be asymmetric. The degree matrix, D, is an $n \times n$ matrix, defined as

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

The Laplacian Matrix is then calculated using $L = D - A$ similar to Formation and Flocking algorithms. Again, the dynamics of $\dot{\vec{q}} = -L\vec{q}$ are used. The discrete version of the position update formula will therefore be

$$\boldsymbol{q}(t + \Delta t) = \boldsymbol{q}(t) - L \cdot \boldsymbol{q}(t) \cdot \Delta t,$$

where $\Delta t \in \mathbb{R}_{>0}$ is the user defined time-step between iterations of the simulation. Note: You may recognize these dynamics from the formation algorithm in Section 2.1. 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 radius of communication.

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 $r_{c_i} \in \mathbb{R}_{>0}$ and a position $q_i \in \mathbb{R}^2$ with an overall $n \times 2$ position vector, $\boldsymbol{q}$. The adjacency matrix, A, is an $n \times n$ matrix defined as:

$$A(i,j) = \begin{cases} 1, \text{ if } ||q_i - q_j|| \leq r_{c_i}, \\ 0, \text{ if } ||q_i - q_j|| > r_{c_i}, \end{cases}$$

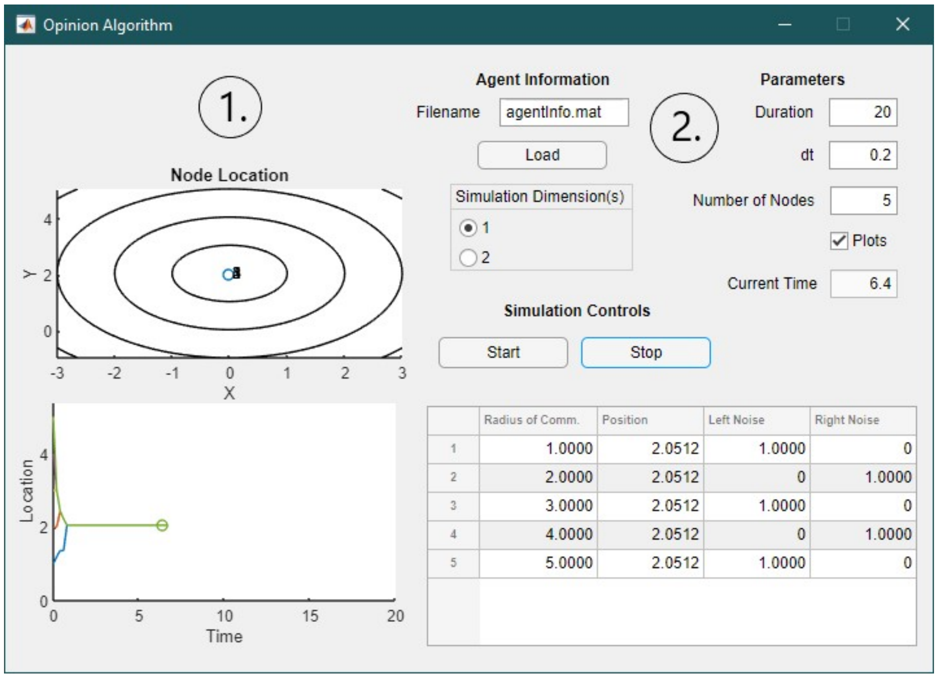
where $||.||$ is the Euclidean norm. The matrices of D and L are both defined in the same manner as in the one-dimensional case. Following a discrete-time system, the position of each agent is updated using

$$\boldsymbol{q}(t + \Delta t) = \boldsymbol{q}(t) - L \cdot \boldsymbol{q}(t) \cdot \Delta t.$$

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.

4.2 Simulation App

The app provided uses the consensus dynamics defined in Section 4.1 to update each agent's location for every iteration of the simulation. The critical components of the apps functionality have been inserted into external functions, which you will be required to create in order to get the app functioning. The functions that you will be responsible for are discussed in detail in Section 4.2.5. Upon simulation completion, the app will create an excel file containing the position of each agent for every iteration of the simulation. This data can then be used to conduct further analysis on the results to improve or validate your design.



Agent Information

Filename: agentInfo.mat

Simulation Dimension(s): [✓ 1] 2

Parameters

Duration: 20

dt: 0.2

Number of Nodes: 5

[✓] Plots

Current Time: 6.4

Simulation Controls

	Radius of Comm.	Position	Left Noise	Right Noise
1	1.0000	2.0512	1.0000	0
2	2.0000	2.0512	0	1.0000
3	3.0000	2.0512	1.0000	0
4	4.0000	2.0512	0	1.0000
5	5.0000	2.0512	1.0000	0

Figure 5: Screen capture of OpinionApp.mlapp with default settings

4.2.1 Plots

Area 1 of Figure 5 displays the plots section of the the Opinion app. This section contains two separate plots: the **Node Location** plot and the **Node Trajectory** plot.

Note: With one-dimensional dynamics, both plots are displayed; however, for the two-dimensional dynamics, the **Node Trajectory** plot is removed.

The **Node Location** plot displays the current location of each agent in the form of a circular marker. In addition, for each agent, a thin black circle with a radius matching the radius of communication of the agent is displayed to visualise the communication range on the agent.
The **Node Trajectory** plot displays each agent's position as the simulation progresses.

4.2.2 User Controls

Area 2 of Figure 5 displays the user controls section of the Opinion app. This section is divided into 3 sub-sections: the **Agent Information** sub-section, the **Parameters** sub-section, and the **Simulation Controls** sub-section.

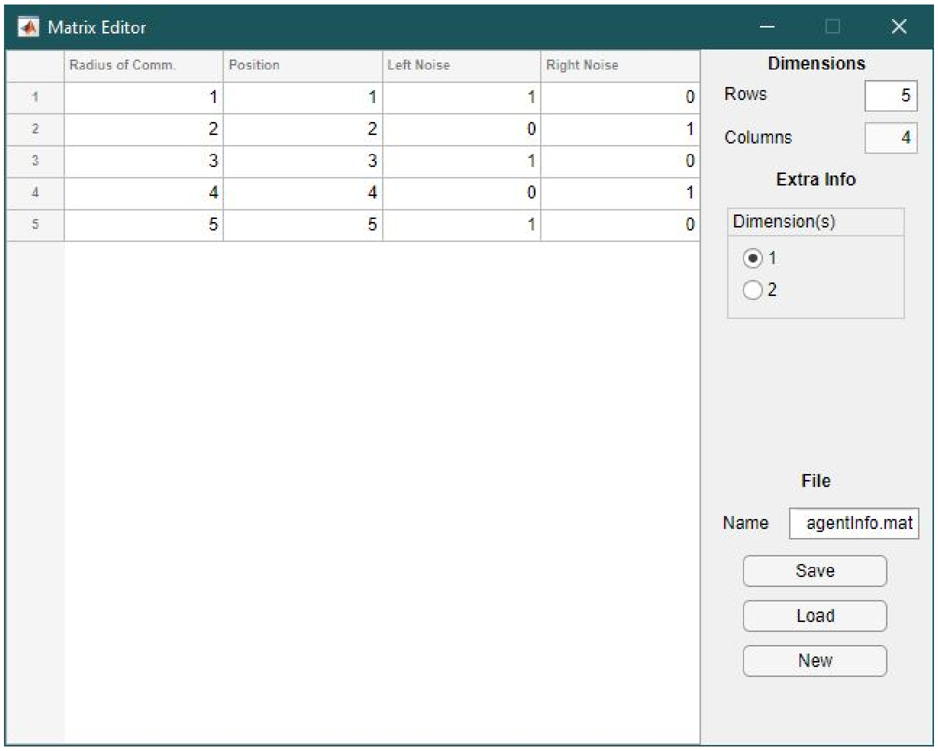
In the **Simulation Controls** sub-section, the **Start** button starts the simulation and the **Stop** button stops the simulation.

Simulation parameters can be modified in the **Parameters** sub-section. The number of nodes in the simulation can be updated using **Number of Nodes** field. The **dt** field is used to adjust the simulation time step, Δt. The length of time that the simulation runs for can be modified in the **Duration** field. The **Current Time** field displays the current simulation time. The **Plots** checkbox, when enabled, will update the **Node Location** plot, the **Node Trajectory** plot (if applicable), and the **Node Information** table on every iteration. Disabling **Plots** will greatly increase the speed of your simulation. This is useful if you need to simulate for more than a few hundred iterations.

When running a one-dimensional simulation, the data entry table will contain four columns with the headings of *Radius of Communication Position*, *Left Noise*, and *Right Noise*. Switching to the two-dimensional simulation will have the data entry table reconfigured with the headings of *Radius of Communication*, *X Position*, and *Y Position*. This table can be filled in manually each time the app is opened, or loaded into the app using a .mat file under the **Agent Information** section. Using the .mat loading feature will save you time by not having to write in the agent data each time the app is opened.

4.2.3 Matrix Editor

The **Matrix Editor Opinion** app allows for the user to save the initial conditions for the nodes in the simulation for either simulation dimension setting. For either dimension setting, the columns in the table are similar to those that would be shown in the Opinion App. The number of rows in the table is reflective of the number of nodes you want to have in your simulation. These initial settings can be saved under a user specified .mat file name and also reloaded into the app to be edited at a later point.



	Radius of Comm.	Position	Left Noise	Right Noise
1	1	1	1	0
2	2	2	0	1
3	3	3	1	0
4	4	4	0	1
5	5	5	1	0

Dimensions
Rows 5
Columns 4

Extra Info
Dimension(s) [✓ 1] 2

File
Name agentInfo.mat

Figure 6: Screen capture of MatrixEditorOpinion.mlapp with default settings

4.2.4 Parameters and Variables

i Number of Nodes ($numNodes$): Total number of nodes (N) used in simulation.

ii Node Data (*nodeData*): Matrix whose rows store the radius of communication, coordinates, and left and right noise for each agent.

iii Radius of Communication (*rComm*): The radius of communication r_c^i for node i. This value can be constant for all nodes or take different values for each.

iv Node Position (*nodePosition*): An $N \times dim$ matrix where row i describes the positional information of the i^{th}-node. For example, in the 1-D and 2-D cases respectively we have:

$$nodePosition\_1D = \begin{bmatrix} q^1 \\ q^2 \\ \vdots \\ q^N \end{bmatrix}, \qquad nodePosition\_2D = \begin{bmatrix} q_x^1 & q_y^1 \\ q_x^2 & q_y^2 \\ \vdots & \vdots \\ q_x^N & q_y^N \end{bmatrix}$$

v Left Noise (*leftNoise*): Random noise taking values between 0 and 1 (inclusive) applied to the positional information of the agent. This parameter is only present in the 1-D simulation case.

vi Right Noise (*rightNoise*): Random noise taking values between 0 and 1 (inclusive) applied to the positional information of the agent. This parameter is only present in the 1-D simulation case.

vii Adjacency Matrix (A): The $N \times N$ adjacency matrix. A must not necessarily be symmetric.

viii Degree Matrix (D): The $N \times N$ degree matrix:

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

ix Laplacian Matrix (L): The $N \times N$ Laplacian matrix:

$$L = D - A$$

x Duration (*duration*): Total duration the simulation will run for.

xi Δt (*dt*): Amount of time simulated in each iteration.

4.2.5 MATLAB Functions

You will need to create the following MATLAB functions in order for the simulation to function. The order that functions are listed is the order that you will be creating them during the project.

calcA.m – calcA.m is used to calculate the adjacency matrix. The calcA function takes as input *nodeData* and outputs A. If the simulation is one-dimensional, A is calculated as seen in Section 4.1.1 with the equation

$$A(i,j) = \begin{cases} 1, & \text{if } |q_i - q_j| \le r_{c_i}, \\ 0, & \text{if } |q_i - q_j| > r_{c_i}. \end{cases}$$

If the simulation is two-dimensional, A is calculated as seen in Section 4.1.2 with the equation

$$A(i,j) = \begin{cases} 1, & \text{if } ||q_i - q_j|| \le r_{c_i}, \\ 0, & \text{if } ||q_i - q_j|| > r_{c_i}, \end{cases}$$

calcL.m – calcL.m is used to calculate the Laplacian matrix. The calcL function takes as input A and outputs L. L is calculated as seen in Section 4.1.1 with the equation

$$L = D - A$$

updateNodeData.m – updateNodeData.m is used to update the *nodeData* matrix. This function must update the position of each agent, but it can also update the other node parameters depending on the cost function(s) or condition(s) introduced in the function. The updateNodeData function takes inputs *nodeData*, *L*, *dt*, and *time*, and outputs *nodeData*. If the simulation is one- or two-dimensional, positional data in *nodeData* is updated as seen in Section 4.1.1 or Section 4.1.2 with the equation

$$\boldsymbol{q}(t + \Delta t) = \boldsymbol{q}(t) - L \cdot \boldsymbol{q}(t) \cdot \Delta t,$$

4.3 Example: Opinion

This section provides a sample P2 project using the Hegselmann-Krause opinion dynamics algorithm. This example is broken down into what is to be completed each week of the project to provide a reference as to what will be expected with your project. You can not use this application area for your project.

4.3.1 Week 1

Area of Application
The application area that has been selected is the spread of fake news within online communities and its influence on political opinions. One of the defining characteristics of the system is trustworthiness, where agents "trust" the agents closer to them more than those farther away. This represents how individuals are more likely to be swayed by agents that are closer to them, reflecting how individuals are more likely to be swayed by sources they are familiar with and echo their own political views. A method to model this situation amongst a populous of potential voters is desired by a government agency.

Algorithm Selection
With the area of application decided, the most applicable deployment algorithm from the list of four options must be selected. Based on the system that is to be modelled, the most suitable algorithm to use is the Hegselmann-Krause opinion dynamics.

Pitch Presentation
With the area of application selected, a brief pitch presentation of the application area was created. This presentation provided a high-level overview of the area of application and how the opinion algorithm could be implemented to generate the appropriate model for the system.

4.3.2 Week 2

Proposal Report
With the area of application selected and opinion algorithm selected, the proposal report could then be written. This report provided a background description of the application area, a problem definition and how the deployment algorithm will be applied. The main stakeholders for the project were identified and included; the citizens of a country, social media and news platforms, a government agency and more. Some preliminary design metrics were also included in the report, which included the design parameters that can be varied for the project. Additional research was conducted to determine these various parameters and their applicable range of values.

Adjacency Matrix
By the standard Hegselmann-Krause dynamics, agents move to the consensus position of all agents within their radius of communication, r_c, equally weighing every opinion they can see. For this application, these dynamics do not suit our system since agents are to favour opinions that are closer to their own. To change this, the adjacency matrix can be redefined 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 $d = ||\vec{q_i} - \vec{q_j}||$ in the two dimensional case. The communication distance r_{c_i} is a value that would be determined through research. Whether the system is to be 1-dimensional or 2-dimensional will be dependent on the research conducted to determine if there is only one set of influence parameters or if there are two sets of independent influence parameters. An "influence leader" k could also be introduced by setting:

$$A(i,k) = 10 \times A(i,k), \forall i \in [1,n],$$

which will increase the influence that the kth agent will have on all other agents by an order of magnitude.

4.3.3 Week 3

MATLAB Coding
Using the adjacency matrix determined from Week 2, the code to generate the corresponding adjacency matrix in the simulation can be written in the *calcA.m* function. In addition, the *calcL.m* function can be written by translating the Laplacian matrix equation into MATLAB code.

Design Process
Continued establishing design metrics to be used to evaluate aspects of the final design solution. An example of one design metric is how quickly focused clusters begin to form.

Researched whether the simulation should be 1 dimensional or 2 dimensional. This should be finalized by the beginning of Week 4. Began researching the various components of the Triple Bottom Line (TBL) analysis for the project; social, economic and environmental considerations. This research could include; budget limits of research groups for design solution, performance and more. Determining how these factors will influence the design choices you make is important to keep in mind.

Reports
Submitted a one page Progress Report that highlighted the team's current position with the project, any challenges faced by the team and strategies to overcome these challenges and complete upcoming tasks.

4.3.4 Week 4

MATLAB Coding
With the adjacency and Laplacian matrices calculated, and the dimension of the simulation finalized, the *updateNodeData.m* function that updates the position and other corresponding data parameters (if applicable) for each node in the simulation can be written. The velocity of the nodes can be capped by normalizing the velocity, $\dot{q}_i$, if it exceeds some maximum speed, as can be seen in Section 2.3.

Design Process
Established methods to evaluate design alternatives. This could include creating an evaluation matrix. If using an evaluation matrix, be sure to justify the categories made and the weights assigned to each category (may require additional research). Certain movements by the nodes may be costly to the node, and thus can be accounted for by introducing a cost function to *updateNodeData.m.* This cost function will require some research and can be fully introduced in Week 5.

By the end of Week 4, the simulation should be operational, with most bugs in the code having been resolved.

4.3.5 Week 5

MATLAB Coding
Based on the results from research, an appropriate cost function was introduced to *updateNodeData.m.*

Design Process
Using the completed simulation, the design parameters of the system can be varied over their appropriate

ranges and evaluated against the established design metrics to determine the most optimum design solution.

Reports
Similar to the first progress report, the second progress report is a one-page summary that highlights the items the team has completed, the challenges the team faces and how they will overcome these challenges.

Began work on the final report, which includes the Design Process, Design Solution and its justification, TBL analysis and more. This report should include the updated sections from the proposal report based on the feedback returned.

4.3.6 Week 6

Final Design
Complete any remaining testing and design finalization. Generate supporting materials for the final report and final presentation. Completed the final report and presentation.

5 Lloyd's Algorithm

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

5.1 Lloyd's Algorithm Dynamics

Lloyd's algorithm is distinct from the previous algorithms' mathematics. This algorithm is better suited to model distributed convergence rather than convergence to a location or velocity. The following sections explain each component of Lloyd's algorithm in detail.

5.1.1 Density

Lloyd's algorithm allocates agents to optimal positions over a fixed area, $A \subset \mathbb{R}^2$, to maximize coverage of some density, $D : A \to \mathbb{R}_{\geq 0}$. D may be either constant or time-variant. The range of D must be non-negative as density is non-negative by definition, and the simulation will behave unexpectedly otherwise.

Example - Constant Density: Search and Rescue

Consider a scenario where robotic agents are tasked with exploring an area in search of multiple target objects. In this situation, the density of the area which we are trying to optimize our coverage of could be represented as a constant probability distribution map in the form of a matrix where cell (i, j) represents the probability of the presence of a target object at the cell's corresponding location in A.

Example - Time Variant Density: Wildfire Suppression

Consider a scenario where robotic agents are tasked with suppressing a growing wildfire through the use of a retardant dispersal system. In this situation, the density of the area which we are trying to optimize our coverage of could be represented as a time-variant heat map in the form of a function, $D(x, y, t)$, which takes in x and y coordinates as well as the time t and returns the heat intensity of location (x, y) at time t

For more detail on density and how to implement it in MATLAB, read Section 5.2.3.

5.1.2 Agents

For n agents, we denote the set of all agents as V ($|V| = n$). Let each agent be denoted $v_i \in V$, the position of each v_i be $p_i \in P$ for each $i \in \{1, ..., n\}$, where $P \subset A$ is the set of all agent positions.

5.1.3 Communication

Lloyd's algorithm takes into account a radius of communication, r_c. If agent v_j is at p_j, and p_j is within r_c of agent v_i'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. More formally, if v_i and v_j are in the same communication graph (i.e. there exists a path between v_i and v_j), they can communicate. By doing this, we partition V into sets of agents that can communicate with each other. Denote these sets $V_k \subseteq V, k \in \{1, ..., K\}$, where K is the number of disjoint communication graphs.

5.1.4 Observation

Lloyd's algorithm also takes into account a radius of observation r_o. Agent v_i at position p_i observes all the points within r_o of p_i. Formally, agent v_i observes the region $\bar{B}_{r_o}(p_i)$. Moreover, we assume agents in the same communication graph share information about their observed regions and the density within them. Let $O_k \subseteq A$ be the observed region by all agents in V_k, then

$$O_k = \bigcup_i \bar{B}_{r_o}(p_i), \; i \in V_k.$$

Then, we determine the points in O_k which are closest to each agent v_i, $i \in V_k$. This partitions O_k into Voronoi regions. Let $R_{k_i} \subseteq O_k$, $i \in V_k$ be such partitions. We then have

$$R_{k_i} = \{x \in O_k : \|x - p_{i_k}\| \leq \|x - p_{j_k}\| \; \forall j \in V_k\}.$$

Agent v_i is then assigned the partition R_{k_i}. The above process is repeated for each $k \in \{1, ..., K\}$, and thus each agent v_i is assigned a region R_{k_i}. Note that in some cases, $\bigcap_{k=1}^{K} O_k \neq \emptyset$, and $\bigcup_{k=1}^{K} O_k \neq A$ (i.e there may be some overlap in coverage and not complete coverage of the target area, respectively).

5.1.5 Convergence

Each agent then converges to the centroid of their assigned region. This is done by calculating the mass $M_{k_i} \in \mathbb{R}$ of each region R_{k_i} using

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

and the centroid $C_{k_i} \in A$ of R_{k_i} is given by

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

5.1.6 Movement

Finally, the agent v_i will move towards their C_{k_i}. The centroid of each region is effectively a weighted average position, so the net effect is that agents will disperse over A to maximize the mass of their observed regions.

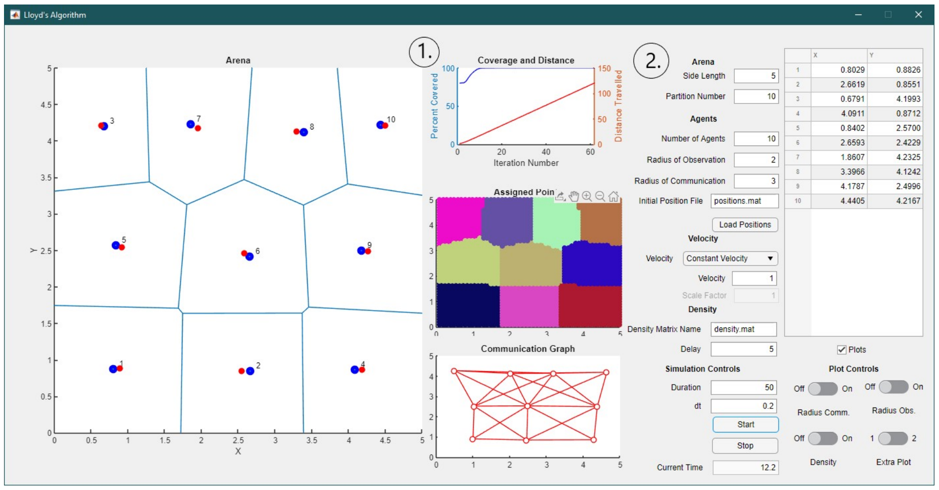
5.1.7 Limitations

In practice, few resources can be described with a continuous density function. Even if this were possible, it is not feasible to compute integrals to a high precision in real time. Instead, the integrals above are computed by discretizing the density functions and taking sums. This improves computation time at the cost of accuracy.

If density is time-invariant, agents will converge at C_{k_i}. This is the optimal positions for the agents to maximize the mass of their observed regions. Note that this heavily depends on r_o, r_c, and D. If $r_c < r_o$, the agents may not know that they are covering the same area. If r_o is sufficiently small and D has local maximums, then an agent may converge to a less-than-optimal C_{k_i}.

5.2 Simulation App

Upon first glance, the simulation app for Lloyd's algorithm can appear very intimidating. Don't be alarmed. This is your guide to understanding all components.



Arena
Side Length: 5
Partition Number: 10

Agents
Number of Agents: 10
Radius of Observation: 2
Radius of Communication: 3
Initial Position File: positions.mat

Velocity
Velocity: Constant Velocity
Velocity: 1
Scale Factor: 1

Density
Density Matrix Name: density.mat
Delay: 5

Simulation Controls
Duration: 50
dt: 0.2
Current Time: 12.2

[✓] Plots

	X	Y
1	0.8029	0.8826
2	2.6619	0.8551
3	0.6791	4.1993
4	4.0911	0.8712
5	0.8402	2.5700
6	2.6593	2.4229
7	1.8607	4.2325
8	3.3966	4.1242
9	4.1787	2.4996
10	4.4405	4.2167

Figure 7: Screen capture of LloydApp.mlapp with default settings

5.2.1 Plots

Area 1 of Figure 7 displays the plots section of the Lloyd app. This section contains four separate plots: the **Arena** plot, the **Coverage and Distance** plot, the **Assigned Points** plot, and the **Communication Graph** plot.

The **Arena Plot** displays agent positions as blue dots, region centroids as red dots, and a Voronoi diagram to illustrate each agent's assigned region. In the **Plot Controls** section, a contour diagram can be toggled to display density information, and observation radii can be toggled to display observation circles around each agent.

The **Coverage and Distance** plot displays coverage of the arena's density as a percentage of total unique observed mass over total arena mass on the left axis, in blue. On the right, the cumulative distance travelled by all agents is shown in red. The independent axis for both plots is time, measured in iterations.

The middle plot (by default, **Assigned Points** plot) can be configured to display two different plots. Plot 1 displays all discrete points assigned to each agent in a different colour. Plot 2 displays each agent's energy on a bar graph. This is useful if your agents' energy supply (battery, fuel level, etc.) drains as they move. The plot can be switched during the simulation in the **Plot Controls** section.

The **Communication Graph** plot displays one or more un-directed graphs to show which agents are in communication. In the **Plot Controls** section, communication radii can be toggled to display communication circles around each agent.

The **Agent Position Table** displays the current position of each agent throughout the simulation. Before the simulation starts, you can manually edit positions using this table. You can also load predefined positions for arbitrary agents into the table (see Section 5.2.2).

5.2.2 User Controls

Area 2 of Figure 7 displays the user controls section of the Lloyd app. This section is divided into 5 sub-sections: the **Arena** sub-section, the **Agents** sub-section, the **Velocity** sub-section, the **Density** sub-section, and the **Simulation Controls** sub-section.

In the **Arena** sub-section, you can specify physical dimensions of your simulation arena in the **Side Length** field. Additionally, each physical unit can be subdivided into multiple partitions for higher simulation resolution in the **Partitions** field. Note that the simulation arena must be a square. For example, if you want to simulate a 10m-by-10m arena with accuracy to 0.5m, you would specify a **Side Length** of 10 and **Partitions** of 2.

In the **Agents** sub-section, you can specify the **Number of Agents** for the simulation, the **Radius of Observation** for all agents, and the **Radius of Communication** for all agents. Note that changing the **Number of Agents** field will change the number of rows in the **Agent Position Table**. If you want to load initial positions from a *.mat* file, specify the file name (including the file extension) and press the **Load Positions** button. This will overwrite the **Agent Position Table** data and the **Number of Agents** field.

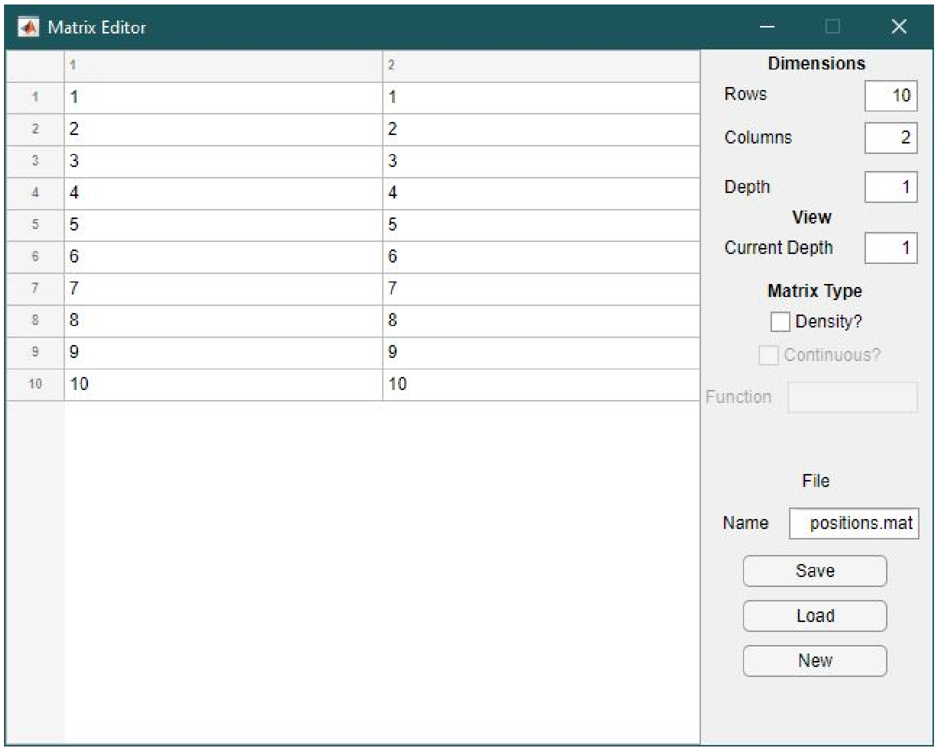
The **Velocity** sub-section allows you to change how agents move. If **Velocity Type** is Constant, agents will move the specified distance in the **Velocity** field every iteration. If **Velocity Type** is Proportional, agents will move at a velocity proportional to the distance to their centroid times the **Scale Factor**. Agents will not exceed **Maximum Velocity** in Proportional Velocity mode.

The **Density** sub-section allows you to specify the name (including the *.mat* extension) of your density file. Density will be loaded from the file when the simulation begins. You must create a density file using the **Matrix Editor**. The **Delay** field specifies the how often D will be updated. Delay is only useful if your density is time-variant.

The **Simulation Controls** sub-section contains: the **Duration** field to select the length of time for which your simulation will run; the **dt** field to select the simulated time step Δt; the **Start** and **Stop** buttons to begin and end your simulation, respectively. The **Current Time** field displays the simulation's current time. The **Plots** checkbox, when enabled, will update selected plots and tables. Disabling **Plots** will greatly increase the speed of your simulation. This is useful if you need to simulate for more than a few hundred iterations.

5.2.3 Matrix Editor

The **Matrix Editor Lloyd** app serves two purposes: to create your density matrix/function, and to create your initial agent position matrix.



	1	2
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	10	10

Dimensions
Rows 10
Columns 2
Depth 1
View
Current Depth 1
Matrix Type
☐ Density?
☐ Continuous?
Function
File
Name positions.mat

Figure 8: Screen capture of MatrixEditorLloyd.mlapp with default position settings

Density:

1. Ensure you have determined a density that models your topic, i.e. on paper. You must thoroughly consider this step, otherwise your simulation results will not accurately model your topic
2. Check the **Density?** box
3. Depending on your density, do one of the following:
   (a) If your density is a continuous function $D(x, y, t)$, check the **Continuous?** box and enter the expression in the **Function** field. Ensure that $D(x, y, t) \geq 0$ for all $(x, y) \in A$ and $t \in [0, duration]$
   (b) If your density is comprised of discrete constant values or discrete functions of t, the density matrix will be a square matrix with side lengths of $sides \times partitions$ where $sides$ is the side length of the arena and $partitions$ is the partition number of the arena. i.e., if the arena dimension is 2km $\times$ 2km and the partition number is 4, then both **rows** and **columns** should have a value of 8 such that the density matrix is an 8×8 matrix.
   (c) If you need more complex density behaviour, ask a TA for help
4. When you're done, specify a file name (including the *.mat* extension) and press the **Save** button

Agent Positions:

1. Specify the number of agents you would like in the **Rows** field
2. Set **Columns** to 2
3. Enter the initial positions for each agent in the grid. Column 1 is X, column 2 is Y
4. When you're done, specify a file name (including the *.mat* extension) and press the **Save** button

If you're finished editing one file, and would like create another, press the **New** button. If you need to edit a file, specify the proper file name and press the **Load** button. Be sure to save your changes.

5.2.4 Parameters and Variables

i Side Length (*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 (*partitions*): The space A is discretized by dividing each unit of space by $PartitionNumber$ where $PartitionNumber \in \mathbb{Z}_{>0}$. For example, if A has dimensions 5×5 and $PartitionNumber = 10$, then A is represented in MATLAB by a 50×50 matrix.

iii Density (*density*): Once the space is discretized, a density matrix is created with dimensions $sides \cdot PartitionNumber \times sides \cdot PartitionNumber$. A cell $(x, y) \in A$ can be mapped to its associated cell $(i, j) \in density$ with

$$(i, j) = (floor(x \cdot PartitionNumber), floor(y \cdot PartitionNumber)),$$

similarly, a cell $(i, j) \in D$ can be mapped to its associated cell $(x, y) \in A$ with

$$(x, y) = (\frac{i}{PartitionNumber}, \frac{j}{PartitionNumber}).$$

iv Agent Positions (*agentPositions*): 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:

$$AgentPositions = \begin{bmatrix} p_{1_x} & p_{1_y} \\ p_{2_x} & p_{2_y} \\ \vdots & \vdots \\ p_{N_x} & p_{N_y} \end{bmatrix}$$

v Adjacency Matrix (*adjMatrix*): The $N \times N$ symmetric adjacency matrix:

$$adjMatrix(i, j) = \begin{cases} 1, & \text{if agent } v_i \text{ communicates with agent } v_j \\ 0, & \text{otherwise.} \end{cases}$$

vi Communication Cells (*commCells*): The $1 \times K$ Cell array, where cell k represents the communication group V_k (see Section 5.1.3). Hence, the matrix looks as follows:

$$agentPoints = (\begin{bmatrix} v_1^1 \\ v_2^1 \\ \vdots \\ v_n^1 \end{bmatrix}, \begin{bmatrix} v_1^2 \\ v_2^2 \\ \vdots \\ v_m^2 \end{bmatrix}, \ldots, \begin{bmatrix} v_1^K \\ v_2^K \\ \vdots \\ v_p^K \end{bmatrix})$$

where $\bigcup_{i=1}^{K} V_i = V$

vii Agent Points (*agentPoints*): 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 *agentPoints* 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:

$$agentPoints = (\begin{bmatrix} x_{1,v_1} & y_{1,v_1} \\ x_{2,v_1} & y_{2,v_1} \\ \vdots & \vdots \\ x_{K_1,v_1} & y_{K_1,v_1} \end{bmatrix}, \begin{bmatrix} x_{1,v_2} & y_{1,v_2} \\ x_{2,v_2} & y_{2,v_2} \\ \vdots & \vdots \\ x_{K_2,v_2} & y_{K_2,v_2} \end{bmatrix}, \ldots, \begin{bmatrix} x_{1,v_N} & y_{1,v_N} \\ x_{2,v_N} & y_{2,v_N} \\ \vdots & \vdots \\ x_{K_N,v_N} & y_{K_N,v_N} \end{bmatrix})$$

viii Mass (*mass*): A $1 \times N$ array containing the mass of R_{k_i} in column i where the mass is calculated with

$$mass(i) = \sum_{(i,j)} density(x_i, y_i) \quad \forall \quad (x_i, y_i) \in R_{k_i}$$

ix Centroids (*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 are calculated with

$$centroids(i,1) = \frac{1}{mass(i)} \cdot \sum_{(x_i,y_i)} x_i \cdot density(x_i, y_i) \quad \forall \quad (x_i, y_i) \in R_{k_i}$$

$$centroids(i,2) = \frac{1}{mass(i)} \cdot \sum_{(x_i,y_i)} y_i \cdot density(x_i, y_i) \quad \forall \quad (x_i, y_i) \in R_{k_i}$$

x Percent of Covered Mass (*coverage*): Ratio of the mass contained by points that are covered by an agent (*coveredMass*) divided by the mass contained by the arena (*totalMass*). A point (x, y) is considered covered by agent i if $(x, y) \in B_{r_o}$.

xi Velocity Scale Factor (*scaleFactor*): This is used in the simulation if Velocity Type is set to Proportional Velocity. The user may have the agents move as follows: Given an agent in position p_i and the centroid C_{k_i} of the agent's assigned region R_{k_i}, then in one iteration, agent i will move a distance of $scaleFactor \cdot ||C_{k_i} - p_i||$. Note that *scaleFactor* must be greater than zero in order for the agents to move, and less than 2 in order for the algorithm to converge. A *scaleFactor* 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 (*velocity*): Each iteration represents a time step of Δt. 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 $\Delta t \cdot velocity$ in the direction of the centroid of R_{k_i}.

xiii Max Velocity (*maxVelocity*): Once the distance d to move after one iteration is determined, the agent's velocity is taken to be $v = \dfrac{d}{\Delta t}$. If $v > maxVelocity$, the velocity of the agent will be set to max velocity for that iteration. Note that if a constant velocity v is implemented and $v < maxVelocity$, then max velocity has no effect on the algorithm.

xiv Delay (*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 5^{th} iteration. Note that delay must be a natural number.

xv Radius of Observation (*rObs*): The radius of observation r_o is defined such that an agent v_i at position p_i observes all the points within r_o of p_i.

xvi Radius of Communication (*rComm*): The radius of communication r_c is defined such agent v_i at position p_i communicates with all agents within r_c of p_i.

xvii Number of Agents (*numAgents*): Total number of agents (N) used in simulation.

xviii Distance Travelled (*distanceTravelled*): An array that contains the sum of the distances traveled by all agents each iteration.

xix Energy (*energy*): A $1 \times N$ array containing the energy of agent i in column i. You must determine the units of the energy that you wish to use and how you want to represent this in the energy array. 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 move at with

$$energy(i)_{j+1} = energy(i)_j - \frac{1}{2}m_i v_i^2,$$

where m_i is the mass of the agent, v_i is its velocity, and j is the time step.

5.2.5 MATLAB Functions

The following functions are external MATLAB functions. Most will need to be programmed by you, but some will be given.

communication.m – communication.m is used to calculate which agents are in communication as seen in Section 5.1.3. The communication function takes inputs *agentPositions* and *rComm*, and outputs *commCells* and $adjMatrix$. $adjMatrix$ is calculated by looping through all agents v_i and v_j and setting $adjMatrix(i, j) = 1$ if they are within each others radius of communication. *comCells* is calculated using MATLAB's graph function to create a graph of the adjacency matrix as well as MATLAB's conncomp function to return the connected components of the graph where *commCells*(i) contains the agent IDs for all agents that belong to component i.

assignAgentPoints.m – assignAgentPoints.m is used to assign points in the arena to agents based on the algorithm discussed in Section 5.1.4. Agents in the same communication cell should not be assigned the same points. However, agents who cannot communicate may cover the same points. The assignAgentPoints function takes inputs *agentPositions*, *commCells*, *sides*, *partitions*, and *rObs*, and outputs *agentPoints*. *agentPoints* is calculated by looping through each communication cell and using MATLAB's rangesearch function to determine which points in the arena are within each agent's radius of observation.

calcMass.m – calcMass.m is used to calculate the total mass of each agent's observed region. The calcMass function takes inputs *agentPoints*, *density*, and *partitions*, and outputs *mass*. *mass* is calculated by summing the density over each agent's observed region.

calcCoverage.m – calcCoverage.m is used to calculate how much of the total mass in the arena is being observed by the agents. The calcCoverage function takes inputs *agentPoints*, *partitions*, *density*, and *totalMass*, and outputs *coverage*. $CalcMass$ is calculated by summing the density over each agent's observed region and dividing by the total mass in the arena.

calcCentroids.m – calcCentroids.m is used to calculate the centroid of each agent's observed region as discussed in Section 5.1.5. The calcCentroids functions takes inputs *agentPoints*, *mass*, *density*, *agentPositions*, and *partitions*, and outputs *centroids*. *centroids* is calculated by looping through each agent and taking the weighted average of the points in their observed region based on the points' masses.

moveAgents.m – moveAgents.m is used to calculate the movement of each agent towards its centroid, as described in Section 5.1.6. Additionally, this function updates the distance travelled statistic and agent energy levels. The moveAgents function takes inputs *agentPositions*, *centroids*, *sides*, *dt*, *energy*, *velocityType*, $maxVelocity$, and $scaleFactor$, and outputs the updated *agentPositions*, $distanceTravelled$, and *energy*. *agentPositions* is updated by calling the velocityFunction function to determine the positional change of each agent and adding the change to *agentPositions*. $distanceTravelled$ is updated by calculating and summing the distance travelled by each agent. *energy* is updated by calling the energyFunction function to determine the energy change of each agent and adding the change to *energy*.

velocityFunction.m – velocityFunction.m is used to calculate the positional change of each agent in one iteration. The velocityFunction function takes inputs *direction*, *velocity*, and *dt*, and outputs *deltaPosition*. *deltaPosition* is calculated by multiplying (element-wise) the *direction* and *velocity* vectors to obtain the positional change of each agent in one time unit, and then multiplying the resultant vector by *dt* to obtain the positional change of each agent in one iteration.

energyFunction.m – energyFunction.m is used to calculate the energy change of each agent in one iteration. This can incorporate kinetic energy, friction, potential energy, etc. The energyFunction function may take various inputs based on your energy model, but the basic inputs are *velocity*, *deltaPosition*, and *dt*. The energyFunction function returns a *deltaEnergy*. *deltaEnergy*'s calculation will be determined by you over the course of the project.

calcDensity.m – calcDensity.m is used to calculate the density matrix for a given iteration. The calcDensity function takes inputs D, *iteration*, *sides*, *partitions*, and outputs *density*. If D is either a continuous function of x, y, and t, or a matrix of discrete functions of t, *density* is calculated by creating a $sides \cdot partitions \times sides \cdot partitions$ matrix and calculating the density for each cell in the matrix. If D is a constant matrix, *density* is set equal to D. See Section 5.2.3 for further information regarding the initiation of D.

5.3 Example: Lloyd

This section provides a breakdown of what a P2 project using Lloyd's Algorithm could look like. You can not choose the exact same topic as the example given below.

5.3.1 Week 1

Area of Application
The topic for this sample project is search and rescue missions in the remote areas of British Columbia. In the mountainous backcountry regions of BC, a variety of Search and Rescue (SAR) techniques are employed to maximize the likelihood of recovering a missing person in the wilderness. Currently, helicopters are used extensively for better access to remote locations and for air surveillance. Recent advances in autonomous drone technology has allowed for the possibility of remote drone surveillance in SAR operations.

Algorithm Selection
After examining the four algorithms, it is easy to see that SAR operations fall under Lloyd's algorithm: rescuers spread out to maximize coverage of some area, and they need to remain in communication with one another for safety. The key parameters of Lloyd's algorithm are a density map, radii of communication, radii of observation, and movement behaviour.

Pitch Presentation
The area of application has been selected, and some background research has been conducted. A brief pitch presentation should now be created. This presentation provides a high-level overview of the area of application and how a deployment algorithm could apply to the topic.

5.3.2 Week 2

Proposal Report
With the application area and deployment algorithm selected, the proposal report can now be written. This report includes all the standard items listed on the rubric, such as an executive summary, introduction with background information, discussion, project plan, etc. The problem definition specifies that a performance analysis of current and upcoming air-based SAR techniques must be conducted to determine the efficacy and economic viability of new technology in the field. Stakeholders are identified to be the general public in BC, the BC Search and Rescue Association, The Government of British Columbia, and more. Design metrics include average coverage, maximum coverage, time to average coverage (TTAC), initial cost per drone, operational cost, drone maintenance cost, and drone lifespan.

Three designs are to be analyzed in this project. The first is a traditional helicopter with a human spotter (CH-149 Cormorant). The second is a fixed wing drone with thermal imaging (X8 Long Range Surveillance Drone). The last is a quadcopter drone with thermal imaging (RMUS Search and Rescue Drone).

Dimensions
The arena dimension was chosen to be $100km \times 100km$, with simulation accurate to within $500m$. Hence, we have the dimension parameters of $sides = 100$ and $partitions = 2$

Radii of Communication
The helicopter is equipped with a powerful VHF communication system, allowing it to communicate up to 48km at SAR height. The fixed wing drone has a communication range of 40km. The quadcopter drone has a communication range of 8km. All of the above information was found using each vehicle's technical documentation.

Radii of Observation
In general, this parameter will be tricky to estimate if it needs to be derived from your agent specifications or other hardware, unless it is listed by the manufacturer. For the sake of simplicity, we will assume the helicopter has a radius of observation of 2km. We will also assume both drones have a radii of observation of 1km.

5.3.3 Week 3

MATLAB Coding
Develop *communication.m*. The adjacency matrix can be created easily enough, as it is symmetric and only based on the *rComm* and *agentPositions* parameters. To create the communication cells, the *graph* and *conncomp* library functions in MATLAB might be useful.

To develop *assignAgentPoints.m*, it will be necessary to create a list of all points in the arena. This can be done using the *meshgrid* library function. Instead of using a loop, use the *rangesearch* library function to determine which points are within an agent's r_{Obs}. Remember that agents in the same communication cell should not be assigned any duplicate points. In this case, use the *sort* or the *sortrows* library function to sort points by distance, and *unique* to remove duplicates. Lastly, this process will execute much faster if all variables used are pre-allocated, i.e. avoid the use of appending new elements to an array/matrix in a loop.

5.3.4 Week 4

Density
SAR operations usually take careful planning to map out regions where it is more likely to find missing person(s). This could be the basis for a density map: a probability map over a $100km \times 100km$ area of interest. The missing person(s) could have been at an initial location within some radius, so the probability would be higher in this region. As time passes, it becomes more likely that individuals wander further, so this radius of uncertainty could grow larger, expanding the region of significant probability.

MATLAB Coding
calcDensity.m will look different depending on whether you used a single continuous function, or a matrix of discrete function. If density is a continuous function, D must be dicretized into dimensions that match your arena. In this case, the *meshgrid* library function will be useful. In either case, the symbolic function D must have t substituted using the *subs* library function. See the MATLAB Primer section on symbolic equations or consult Google for additional help. Finally, ensure the substituted density matrix is converted to numbers using the *double* library function.

Develop *calcMass.m*. The total mass of an agent's region is just the sum of the region's density. The *sum* library function will be useful. The *agentPoints* parameter specifies a list of subscript indexes for the density matrix. It may be useful to use the *sub2ind* library function to convert these subscripts to linear indices.

The implementation of *calcCentroids.m* follows almost directly from the procedure discussed in Section 5.1.5. The only difference is the calculation should be discretized.

5.3.5 Week 5

Movement Behaviour

Agent movement behaviour can vary depending on both physical capabilities and the application of the algorithm. The simulation app is designed for two default modes for movement behaviour: Constant and Proportional. Constant Velocity forces all agents to maintain the same velocity, moving the same distance every iteration. Proportional velocity determines a velocity proportional to the distance between an agent and its centroid, not exceeding a threshold maximum.

Both of the above behaviours would need to be programmed in your simulation. Take time to decide what movement behaviour(s) would be most realistic for your project, and program only what you need.

For this example project, we will consider the helicopter to be travelling at a constant velocity of 200km/h. The fixed wing drone will be travelling at a proportional velocity with a maximum speed of 100km/h. The quadcopter drone will be travelling at a proportional velocity with a maximum speed of 80km/h. The helicopter travels at a constant velocity because it has more momentum, and thus is more costly to rapidly accelerate. In contrast, both drones are lightweight and would be able to follow a proportional velocity path in the real world.

MATLAB Coding

To develop *moveAgents.m*, you will need to calculate the magnitude and direction of velocity before calling *velocityFunction*. Use the *vecnorm* library function and logical indexing to ensure magnitudes are within your expected range; agents need to remain within the boundaries of the arena. Call *velocityFunction*, supplying the direction and magnitude as parameters, then update agent positions. You may also wish to include a cost function here. If you are modelling energy, call *energyFunction* with any parameters you need.

Next, develop *velocityFunction.m*. This function only needs to apply basic kinematics to determine change in position given velocity magnitude and direction. If your simulation considers one iteration to be more than one second, this function should account for that. In this example, we use a simulated time step Δt of 1 second, so position change is simply the product of speed and direction.

5.3.6 Week 6

MATLAB Coding

The last function to develop is *energyFunction.m*. To calculate change in energy, we must first establish what metric constitutes energy, and which parameters to include in calculating energy change. Energy could be actual energy, distance, or time, so long as it accurately represents resource constraints on your system.

In this example, all agents start with a 'battery' that represents initial fuel levels. All the design specifications for the helicopter, fixed-wing, and quadcopter drones are listed online. They each have a range of 1000km, 40km, and 8km, respectively. The 'battery' in this case will be represented as how many kilometres of flight remain until fuel is depleted, and our input parameter will be distance travelled.

Final Design

Finalize the optimal design specifications. Create materials that can be included in the final report and presentations. Completed the Final Report and Final Presentation putting significant attention to the design process and design justification process (evaluation matrices, design rubrics, etc.)

References

[1] National Aeronautics and Space Administration, “The Drag Equation,” *National Aeronautics and Space Administration* 2015. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/drageq.html [Accessed June 26, 2019]