APSC 200 P2: Course Manual

Department of Mathematics and Engineering
Queen's University

June 5, 2019

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

You will be applying the engineering design process to a topic of your choice using a group formation algorithm of your choice. 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 develop your systems and their accompanying design parameters using the mathematics provided by your algorithm as part of the engineering design process. You will then test those systems in a simulation environment. 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 you design your simulation to be.

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!*

2 Formation Algorithm

The first algorithm that will be discussed in 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 agent position and returns the agent velocity. The continuous-time dynamics of the system can be described by the first order differential equation of:

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

Where $\vec{q}$ is a $N \times 2$ position vector with the x, y coordinate pairings for each agent in the system (for a total of N agents). The ith row of $\vec{q}$ represents the position of the ith agent in the system and is represent by $\vec{q_i}$. The Laplacian matrix, L, for the system is a $N \times N$ matrix with the property:

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

This implies that $\dot{\vec{q}} = [0, 0, ..., 0]^T$ if $\vec{q_i} = \vec{q_j}$ $\forall i, j \in [1, N]$. The Laplacian matrix, L is a function of the adjacency matrix, A, for the system and can be calculated using:

$$L = D - A$$

The adjacency matrix is an $N \times N$ matrix where

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

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

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

The system then updates position in discrete time as follows:

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

NOTE: A and L will most likely be symmetric as the communication lines between agents goes both ways. If your application has unidirectional communication lines, these matrices will not be symmetric.

2.1.1 Introducing Offsets

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

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

Where G is a $N \times 2$ offset vector 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 $\dot{\vec{q}} = 0, \forall t \in \mathbb{Z}_{\leq 0}$. Otherwise, agent i will update $\tau_i \in \mathbb{Z}_{\leq 0}$ times-steps late when responding to a movement signal. The position update equation is now:

$$\vec{q}(t + \Delta t) = \begin{bmatrix}\vec{q}_1(t - \tau_1)\\ \vec{q}_2(t - \tau_2)\\ \vdots\\ \vec{q}_N(t - \tau_N)\end{bmatrix} - L[\vec{q}(t) - G]\Delta t$$

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 apps functionality have been placed in external functions, which you will be required to create for the simulation to run. The functions that you will be responsible for are discussed in detail in Section 2.2.4. Upon simulation completion, the app will create an excel file containing the position of each agent for every iteration throughout the simulation. This data can be used to conduct further analysis to improve or validate your design.

2.2.1 Plots

There are two plots displayed in the formation simulation app. 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 plots iteration number, scaled by the **Time Step** factor whilst the dependent axis plots distance between agent and consensus position.

2.2.2 User Controls

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. The table in the app contains five columns; the current x and y *position, delay*, and x and y *offset* for each agent. You can manually input initial agent data for this table. Alternatively, use the Matrix Editor to create an $N \times 5$ matrix for initial agent data (See Section 2.2.3). While the simulation is running, the agent data table will be updated every iteration to reflect any changes in each agents' properties.

The **Number of Iterations** edit field allows the user to adjust the simulation duration. The **Current Iteration** edit field is non-editable and displays the current iteration while the simulation is running. The value of Δt in the position update equation in Section 2.1 can be modified via the **Time Step** edit field.

The **Start** button begins the simulation and the **Stop** button stops the simulation before completion. Under the **A Matrix** header, the name of the .mat file containing your system's adjacency matrix can be inserted. An adjacency matrix file name must be specified in order for the simulation to run. The initial conditions for the agents can be loaded into the simulation app by specifying a .mat file name under the **Agent Information** heading and pressing **Load**, avoiding the need to manually reenter data. Both the adjacency matrix and the agent information matrix can be created using the **MatrixEditorFormation** app.

2.2.3 Matrix Editor

This app can be used to create the .mat files that store the adjacency matrix or agent information.

To create the adjacency matrix file, edit the number of rows and columns to reflect the desired number of agents that are in your system. If your system will switch between adjacency matrices during the simulation, all the adjacency matrices required by your system can be stored under the same file name. This can be achieved by adjusting the depth to the number of adjacency matrices you wish to have. To switch between adjacency matrices in the app window, change the current depth value. Be sure to specify a file name and save your work before exiting the app. An existing adjacency matrix file can be loaded into the app to allow for adjustments to be made at a later time.

To create the agent information matrix, be sure to set the number of rows equal to the number of agents in your system, the number of columns to 5, and the depth to 1. The columns from left to right will store the following information; x position, y position, delay, x offset and y offset. This is how the information will appear when loaded into the simulation app. Be sure to save your work before exiting. Similar to the adjacency matrices, an existing agent information .mat file can be loaded into the app to be edited.

2.2.4 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.

getA.m – Selects and returns an adjacency matrix for a given iteration. This will be very simple for simple convergence behaviour. Switching capabilities based on iteration must be included, if necessary. Given inputs of the matrix from the .mat file and current simulation iteration t, return the 2D adjacency matrix for simulation iteration $t + 1$.

calcL.m – Calculates the Laplacian matrix, as given in Section 2.1. Any intermediate steps must also be performed. Given input of the adjacency matrix A, calculate and return the Laplacian matrix L.

moveAgents.m – Updates the position of each agent using the consensus dynamics formula in Section 2.1. Given inputs of current agent positions, time delays, and offsets, the Laplacian Matrix, current iteration, time step, calculate the updated agent positions. Later, you will introduce a cost function that will influence the movement behaviour of the agents.

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

Select and Research an 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 around several kilometers of the surrounding area and their sample containers are full. Instead of wasting time and resources by having each agent return back to shore to drop off its samples, a ship is to be sent out to the first agent's location. The other agents are meet the ship at the first agent's location. However, communication underwater is difficult and costly, restricting range of submersibles to choose from.

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 deployment algorithm could be used to model the design solution.

2.3.2 Week 2

Introduction to Deployment Algorithms
With a potential area of application selected, the four deployment 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 is the most suitable choice for the project.

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 needs to be conducted on several models of submersible robots to determine the efficacy and economic viability of each 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.

Two submersible robots have been selected for this project. The first is the Submersible Robot 1 (SR1). The second is the Submersible Robot 2 (SR2).

Number of Agents
The research group has specified the need for a cargo capacity of at least 10 units per mission. Examination of the technical documentation for the SR1 and SR2 reveals they have a cargo capacity of 1.25 and 2 units, respectively. This means that, based on the cargo capacity requirements, 8 SR1 submersibles or 5 SR2 submersibles would be required.

Adjacency Matrix
A system comprised of 8 SR1s has an 8-by-8 adjacency matrix. The SR1 technical documentation specifies that communication is unidirectional, i.e. agents can only communicate with one other agent in the system. Thus, a linear communication chain is required. Given that the first agent is at the ship's location, the other resulting adjacency matrix for the system will be of the following form:

$$A_{SR1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Note that this matrix is asymmetric since the communication channels are unidirectional. Notice that agent 8 is not sending data to any agent, and agent 1 is not receiving data from any agent. The isolation of agent 1 ensures that it will not move, as specified in the background information.

A system of 5 SR2s has a 5-by-5 adjacency matrix. Technical documentation for the SR2 specifies that communication is bidirectional. From this, the adjacency matrix of the system is:

$$A_{SR2} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Notice that bidirectional communication translates to a symmetric adjacency matrix. However, A_{SR2} allows for communication with agent 1, meaning agent 1 will move from its initial position. To change this, we will consider agent 1 to be a stubborn agent with an infinite time delay.

2.3.3 Week 3

MATLAB Coding
The *MatrixEditorFormation* app was used to enter the adjacency matrices developed in the previous week, saving them to separate *.mat* files. The *getA.m* function was written to return a static adjacency matrix every iteration, as our simulation does not require switching matrices. The *calcL.m* function was written based on Section 2.1.

Design Process
Continue evaluating the designs based on the metrics set out in the proposal report. At this point, it should be clear which design metrics are dependent on the simulation. In this project, only time to convergence is dependent on the simulation. All other parameters can be evaluated this week, independently of the

simulation.

Progress Report 1
Complete the first progress report as outlined in the report template.

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. The maximum cruising speeds for the SR1 and SR2 are 20km/h and 35km/h, respectively. *moveAgents.m* checks for this to ensure that movement does not exceed maximum speeds.

Time Step
The time step, Δt, is determined by the frequency at which the SR1 and SR2 can transmit/receive data. Based on technical documentation, these values are 2 seconds and 5 seconds, respectively.

Energy
Energy levels within each submersible must be considered in the model; 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.

The SR1 has a full charge battery capacity of 150kJ, whereas the SR2 has 200kJ. The energy model for this example will take into account energy depletion due to work by drag and change in kinetic energy.

Design Process
Decide on a decision making strategy. E.g. if using an evaluation matrix, be sure to justify the 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 modeled by decreasing their energy based on the work done at each time step

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

where E_i is the energy remaining for the ith agent, F_d is the force of drag on the agent, 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.

Design Process
The simulation should now be operational. The final metric(s) can be evaluated for each design using the simulation with parameters derived from prior research. For this example, time to convergence was evaluated using the simulation.

Progress Report 2
Complete the second progress report as outlined in the report template. This report will be similar in structure to the first progress report.

2.3.6 Week 6

Design Finalization
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:

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

Similar to the formation dynamics, the Laplacian matrix, is calculated using

$$L = D - A$$

Where 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 that agents are from each other. The resulting entries for the adjacency matrix are now:

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

Where K, σ and β are user defined parameters. The variable d denotes the Euclidean distance between agents i and j (i.e. $d = ||\vec{q}_i - \vec{q}_j||$). The degree matrix, D, is calculated in the same manner as in the formation algorithm using

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

The algorithm works by updating the velocity of each agent using:

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

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 goal 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 $1 \times T_{max}$, where $T_{max} \in \mathbb{Z}_{<0}$ is the number of iterations the simulation is to run for. The trigger sequence is defined by:

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

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 needs in order 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 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 3.2.4. 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 analyses to improve or validate your design.

3.2.1 Plots

The 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

The **Agent Configuration** 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** 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 **Time Step** edit field. To change the number of iterations that the simulation is to run for, adjust the value under **Simulation Duration**. The **Start** button begins the simulation and the **Stop** button stops the simulation from running.

The parameterized path that the leader is to follow is entered under the **Leader Trajectory** 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** section allows for a user specified .mat file to be loaded into the app that contains the initial x, y positions and velocity for each agent. This information is loaded into the table at the bottom right of the app window. Alternatively, this information can be entered manually within the app. Additional information can also be uploaded to the simulation in via the matrix. This additional information includes the leader trajectory equations and the K, Sigma and Beta values for the adjacency matrix. Note that during the simulation, the position and velocity of each agent is updated for each iteration of the simulation. These updates will not alter the original .mat file loaded into the simulation.

3.2.3 Matrix Editor

The **MatrixEditorFlocking** 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, 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.

3.2.4 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 – The first function you will need to create is the adjacency matrix for the current iteration of the simulation using the user defined parameters and the agents' current position. The parameters that are fed into this function are the user defined parameters of K, sigma, beta and the agents' current position. The function returns the resulting adjacency matrix. Recall that the formula to calculate each entry in the adjacency matrix can be found in Section 3.1.

calcL.m – This function calculates the Laplacian Matrix following the formula defined in Section 3.1. The parameters that are fed into this function is the adjacency matrix calculated in *calcA.m*. The function will return the calculated Laplacian matrix to the simulation app.

calcLeaderVelocity.m – This function takes in the derivative of the parametric equations for the leader agent's position. The function then evaluates the velocity function at each iteration of the simulation. The parameters fed into this function are the leader's x and y velocity function, the number of iterations the simulation is to run for and the time elapsed, Δt, between each iteration of the simulation. The function then returns the corresponding x and y velocities for each iteration of the simulation.

trigger.m – this function is used to create the desired trigger sequence for the simulation. The parameter fed into this function is the current iteration that the simulation is on. The function must return either a 1 if the trigger is activated and a 0 if it is not activated.

updateVelocity.m – this function uses the algorithm noted in Section 3.1 to update the velocity of each agent. It is important to note that calling of this function will be dependent on the trigger sequence defined in *trigger.m*. The parameters that are fed into this function are the Laplacian Matrix calculated in *calcL.m*, the user defined time step Δt and the agents' current velocity. The function then returns the updated velocity components for each agent. Cost and energy functions can later be introduced within this function to improve the accuracy of the simulation to real life.

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

Select and Research an Area of Application
The application area that has been selected involves some rovers following another rover through the vast expanses of Mars. A new rover is to be designed that has a highly accurate GPS that can return to the mission base solely with the use of the GPS system. The other rovers in the fleet are older models with aged components. These rovers have a radius of communication of 10 meters, can only process one communication every 60 seconds, and has no GPS system. Without the leader robot, the follower robots will be helpless in returning to the base as they will be gradually pushed off course by the uneven Martian terrain.

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 deployment algorithms could be implemented in the design solution.

3.3.2 Week 2

Introduction to Deployment Algorithms
With a potential area of application selected. The four possible deployment algorithms were examined in more detail. The applicability of each deployment 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.

Proposal Report
With the application area and deployment 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 robot that meets the desired performance abilities and satisfies the communication requirements of the existing fleet. 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:

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

This makes it so that the agents can only communicate if they are within 10 meters of each other, and that the signals will be stronger as the follower agent 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 and β.

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 Metrics and Triple Bottom Line
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. This value has a minimum of 60 seconds due to the older rover's design. 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.

Progress Report 1
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 $\vec{v}_L = (0.1t, 0.1t)$. This straight diagonal 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 to be:

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

This establishes that the velocity of each agent is to be updated after every minimum time period between communications. The third function to be written is the *updateVelocity.m* function that updates the velocity of each agent. To incorporate into the function the rover going off path due to uneven terrain, we can add noise values to the velocities of the follower rovers:

$$\vec{v}_f(t) = \vec{v}_f + Noise$$

where for each iteration *Noise* is a 2×1 array of randomly generated values ranging from - 0.05 to 0.05.

Design Metrics
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. Therefore introducing a cost 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.

Testing
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 to evaluate the effectiveness of the design. This could include 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 the remaining 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

Design Finalizing
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 $r_{c_i} \neq r_{c_j}$, the resulting A matrix will be asymmetric. The degree matrix is defined to be an $N \times N$ matrix with

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

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 update position formula will therefore be

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

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 r_c.

4.1.2 Two-Dimensional Dynamics

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

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

where $||.||$ is the Euclidean norm. The matrices of D and L are both defined in the same manner as in the one-dimensional case. The position of each agent is updated in the same way:

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

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

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 4.2.4. 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.

4.2.1 Plots

When the simulation is operating under the one-dimensional dynamics, two plots will be displayed in the application window. The plot in the upper left portion of the window is the **Node Location** plot. This 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.

For the two-dimensional dynamics, the **Node Trajectory** plot is removed.

4.2.2 User Controls

Under the **Simulation Controls** heading, the following commands are available. The **Time Step** entry window allows for the time step, Δt used in the position update equation, to be adjusted by the user. The number of iterations that the simulation runs for can be modified by altering the entry in the **Simulation Duration**. The **Start** button starts the simulation and the **Stop** button stops the simulations. The **Current Iteration** window displays the current iteration that the simulation is on.

The number of nodes in the simulation can be updated using the respectively named edit-field. 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.

4.2.4 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 – This function calculates the adjacency matrix given the information for all the nodes in the simulation using the equation given in Section 4.1.1 or Section 4.1.2 depending on the simulation dimension. The information passed into the function is a $N \times 4$ matrix for the one-dimensional simulation and $N \times 3$ matrix for the two-dimensional simulation. The function returns the resulting $N \times N$ adjacency matrix.

calcL.m – This function calculates the Laplacian matrix for the simulation given the current iteration's adjacency matrix calculated in *calcA.m*. The parameter fed into the function is the adjacency matrix. The

function returns the resulting Laplacian matrix. Recall from Section 4.1.1 for the equations required to determine the Laplacian matrix.

updateNodeData.m – This function uses the position update formula defined in Section 4.1.1 or Section 4.1.2 depending on the type of simulation being run. The parameters fed into the function is the node data for the current iteration. The function returns the updated node data to the app. 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.

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

Select and Research 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.

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 deployment algorithms could be implemented to generate the appropriate model for the system.

4.3.2 Week 2

Introduction to Deployment Algorithm
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 modeled, the most suitable algorithm to use is the Hegselmann-Krause opinion dynamics.

Proposal Report
With the area of application selected and deployment 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 ... .

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.

Progress Report 1
A one page summary 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, $\dot{\vec{q}}_i$, 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*.

Testing
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

Design Finalizing
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 sample, or other situations where agents must cover some limited area and so they spread out to service it.

5.1 Lloyd's Algorithm Dynamics

Lloyd's algorithm has many use cases and is very powerful, but is distinct from the previous algorithms' mathematics. 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, to maximize coverage of some density, D. In this case, $A \subset \mathbb{R}^2$, and D is a function $D : A \to \mathbb{R}_{\geq 0}$. D may also be time-variant, but this is described in more detail in Section 5.2.3. The range of D must be non-negative as density is non-negative by definition, and the simulation will behave unexpectedly otherwise.

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, then

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

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} as follows:

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

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

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

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}.

These are all aspects of real-world engineering design you must take into account when determining parameters for your simulations.

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.

5.2.1 Plots

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

In the **Arena** 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** 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** 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** section allows you to specify the name (including the *.mat* extension) of your density file. Density will be loaded from 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.

Simulation Controls contains the **Simulation Duration** field (i.e. the number of iterations for which your simulation runs) and the **Start** and **Stop** buttons to begin and end your simulation, respectively. The **Current Iteration** field displays the simulation's current iteration.

5.2.3 Matrix Editor

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

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 the function is such that

   $$D(x, y, t) \geq 0 \ \forall \ x, y \in [0, sides \times partitions], \forall \ t \in [0, duration]$$

   (b) If your density is comprised of discrete constant values or discrete functions of t, specify the **Rows** and **Columns** as $sides \times partitions$ and enter the values in the 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 MATLAB Functions

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

assignAgentPoints.m – Assigns points 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. Given inputs P, V_k, *sides*, *partitions*, and r_o, calculate $R_{k_i} \ \forall \ k \in \{1, ..., K\}, \ \forall \ i \in V$

calcCentroids.m – Calculates the centroid of each agent's observed region as discussed in Section 5.1.5. Given inputs R_{k_i}, M_{k_i}, D, P, and *partitions*, calculate $C_{k_i} \ \forall \ k \in \{1, ..., K\}, \ \forall \ i \in V$

calcCoverage.m – Calculates how much of the arena is being observed by considering a weighted average of the density matrix. Given inputs R_{k_i}, D, M, *partitions*, calculate covered mass over total mass, where M is total mass of the arena.

calcDensity.m – Calculates the density matrix for a given iteration. The input density D is either a continuous function of x, y, and t, or a matrix of discrete functions of t designed in the Matrix Editor app (See Section 5.2.3). Given inputs D, t, and optionally *sides* and *partitions*, calculate discretized density.

calcMass.m – Calculates the total mass of each agent's observed region. Given inputs R_{k_i}, D, and *partitions*, sum the density over each region.

communication.m – Determines which agents are in communication as seen in Section 5.1.3. Given inputs P and r_c, calculate $V_k \subseteq V$, $k \in \{1, ..., K\}$ and the symmetric adjacency matrix A_{com}.

energyFunction.m – Determines how agent energy changes. This can incorporate kinetic energy, friction, potential energy, etc. Inputs can vary based on your energy model, but the basic inputs are agent velocities, v, and change in agent positions, ΔP. Returns a vector indicating the change in agent energies, ΔE.

moveAgents.m – Moves each agent towards its centroid, as described in Section 5.1.6. Additionally, this function updates the distance travelled statistic and agent energy levels. Given inputs P, C_{k_i}, $sides$, E, $velocityType$, $maxVelocity$, and $scaleFactor$, determine velocity magnitude and direction. Calls **velocityFunction.m** to determine ΔP and **energyFunction.m** to determine ΔE. Returns updated P, E, and total distance travelled.

velocityFunction.m – Determines how much agents move over one iteration. Given inputs of velocity magnitude and direction, calculates the change in agent positions, ΔP.

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

Select and Research an 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.

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

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.

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

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.

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 *sortrows* library functions 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.

calcDensity.m
This function 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.

calcMass.m
The total mass of an agent's region is just the sum of the region's density. Hence, 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 said subscripts to linear indices.

calcCentroids.m
The implementation of this function 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.

moveAgents.m
You will need to calculate the magnitude and direction of velocity before calling *velcoityFunction*. Use the *vecnorm* library function and logical indexing to ensure magnitudes are within your expected range. 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.

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 assume every iteration to be 1 second, so position change is simply the product of speed and direction.

5.3.6 Week 6

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 kilometers of flight remain until fuel is depleted, and our input parameter will be distance travelled.

Design Finalization
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.)