* “Any general tips?”
  + Validate your behaviors with small swarms before validating on large swarms
  + Don’t worry about making the behaviors perfect – parameter tuning is a long process that won’t really aid understanding of the material
  + Every time there’s some variant, just make a whole new folder. It will get messy but it should make it easier to return to previous implementations
* “What are these dots? What’s object-oriented programming?”
  + The way I think about it is that each “Object” is like you’re copy-pasting a folder in your computer’s file browser. This folder is going to have stuff inside of it, including files (in this case, variables) and it might also have other folders (for example, the logic object inside the robot object). When you want to access something, you essentially drill down in this structure to get to what you want. For example, if, in the run file, we wanted to get the out\_drive property of the logic within the first element of the robot array, we could write “robot\_array(1).logic.out\_drive”. However, if we are operating within the logic object, we would need to specify that we want it to look at a property of it’s self. For this reason, in the robot logic object we would call it as self.out\_drive

Section 1

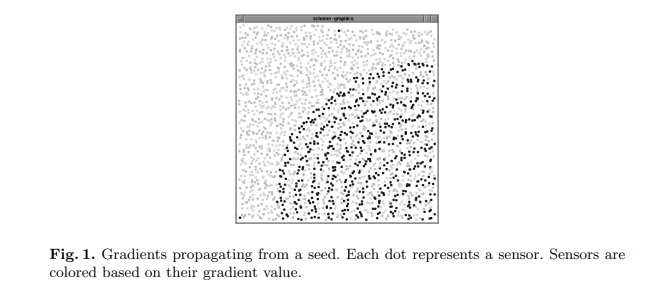
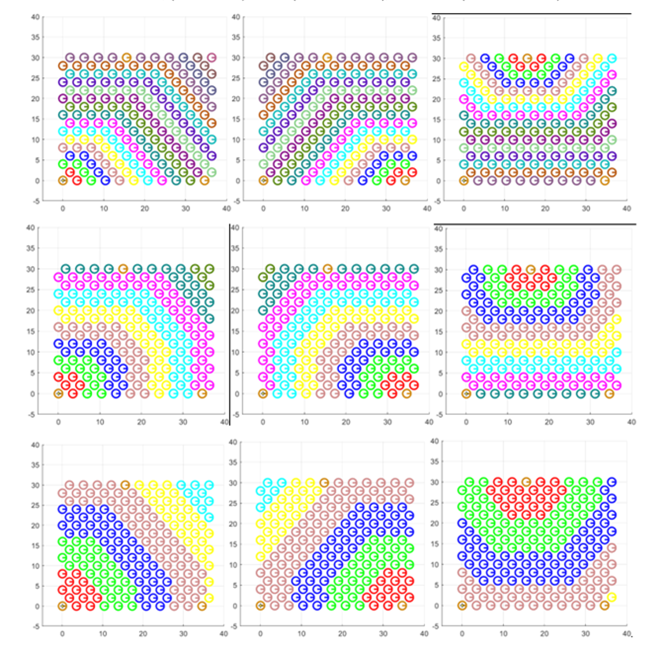
1.1:

1.2:

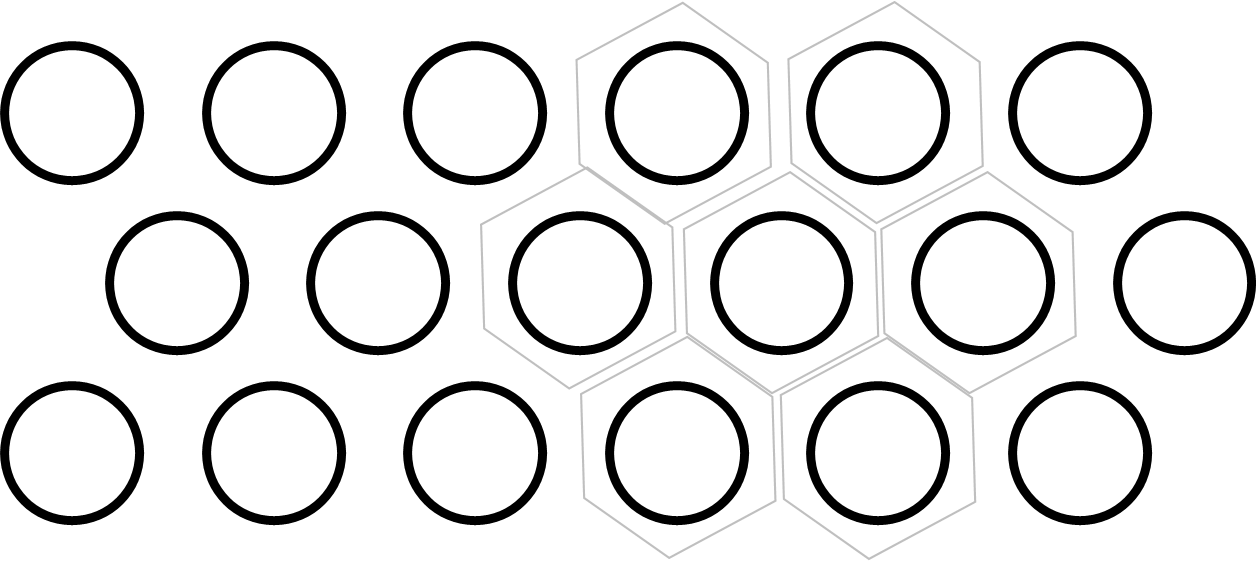
* “ What do you mean by parameter configurations?”
  + Any “magic value” you put into the behavior. Spin speed, orbit distance, forward motion speed are all good options. Just show different pictures of the paths traveled and qualitatively discuss
* “Should my orbit be smooth?”
  + It is unlikely that your orbit will be perfect, nor do I expect it to be. The goal here is to demonstrate that you are able to make the robots do a simple behavior based on inputs and outputs
* “How do I do this?”
  + A good, simple implementation would be to have the planet robot logic command a constant forward speed (self.out\_drive = 0.5, for example). And turn left or right based on whether the distance between the two robots is greater than or less than the desired orbit radius (robot logic can look at self.comm\_in(1).distance for this information)
  + You will have two roles for the robots: “sun” and “planet” assign a number that corresponds to these roles and base the robot behaviors on that (e.g. sun = 1, planet = 2). Refer to the example code for details on how to implement this.
* “Where should the robots start?”
  + Wherever is most convenient for you to place the robots
* “Where in the code do I place the robots?”
  + Robot starting locations should be defined in the function placeRobots.m

Section 2:

2.1:

* “How do I place the robots?”
  + Robot starting locations should be defined in the function placeRobots.m. You can either do this directly by defining 100 or so robots, or do this in a loop (recommended)
* “What sort of work should I be showing?”
  + The main two questions that I’ll be asking when I’m grading this is 1) did the student show evidence that they implemented the algorithm and that the algorithm is generally working? And 2) Does the student talk about how different communication ranges change how the hop count algorithm plays out? The ideal I think would be showing the hop values for each robot, but another way of doing it would be to select one seed and show the “rings” that form as a result of the hop values. A great example of this can be seen in the original paper  
      
    And here is an example from a student in the class:  
    

2.2:

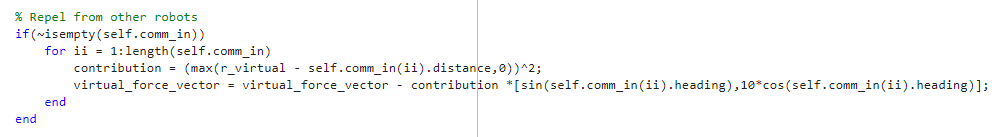
* “What is a hexagonal grid?”
  + Essentially, the robots should be placed as if they were hexagons:  
    
* “How do I place the robots?”
  + Robot starting locations should be defined in the function placeRobots.m. You can either do this directly by defining 100 or so robots, or do this in a loop (recommended)

2.3:

* “How do I do this?”
  + See the end of lecture 6 for hints.

2.4:

Section 3:

* “What are ‘shells’?”
  + Different people call them different things (layers, shells, rings, etc), but this refers to the tendency of the robots to form concentric rings about the center, segregated by color. So when I’m asking for different number of shells, you can just change the number of roles that are present
* “What does this portion of the code you provided mean?”  
  
  + Let’s go line by line:

% Repel from other robots

if(~isempty(self.comm\_in))

This is basically saying: Only do this next part if we actually have some comm\_in to operate on. Comm\_in is an array of messages and so if no messages were received (based on the communication radius passed to the communication handler within run.m), that self.comm\_in array will be empty. We need this line since if there are no messages and you try to operate on a message Matlab will become unhappy and throw an error, so we need this to essentially protect against that scenario.

for ii = 1:length(self.comm\_in)

length(self.comm\_in) is going to return how many messages are within that self.comm\_in array. Therefore, if there are three messages what this will resolve to is “for ii = 1:3” In other words, we would run this loop three times, once at ii = 1, then at ii = 2, then at ii = 3. Anytime you see a “x:y” sort of line in Matlab, this means “generate the numbers between x and y.” So 1:5 would return [1,2,3,4,5] and 2:4 would return [2,3,4]. You can optionally also specify how big of a jump you want, so 5:5:25 would return [5,10,15,20,25]. The best way to understand this is to just type it into the command window

contribution = (max(r\_virtual - self.comm\_in(ii).distance,0))^2;

This is a bit of a tricky line because there’s a fair amount happening. The overall question this line is answering is “If someone is within my virtual radius, how hard should I try to move away (without caring about the direction; only the magnitude”. So the first thing we want to do is to figure out how far inside our virtual radius the other robot is. We know the distance to the other robot because of a property of the communication, namely ‘distance’ So first thing we need to do is to figure out what the distance is to the other robot. We look at the ii member of the self.comm\_in array and get its distance property using self.comm\_in(ii).distance. We then subtract this from our virtual radius r\_virtual. If the other robot \***is**\* inside of our virtual radius, this number will be positive. If the other robot is outside the virtual radius, this number will be negative. Since we don’t care about a robot outside the radius, we use the max(xx,0) function to say “If the answer is negative, just return 0”. Finally, we take this whole thing and square it. This is a parameter I arbitrarily chose because it results in a very strong force when the other robot is close and a weaker force when it’s just barely inside the r\_virtual. This is a parameter that you could tune.

virtual\_force\_vector = virtual\_force\_vector - contribution \*[sin(self.comm\_in(ii).heading),10\*cos(self.comm\_in(ii).heading)];

So now that we know the magnitude of the force we want to add, we need to figure out the direction. There is a property of the messages called ‘heading’ which is the angle from the robot’s forward orientation that the communication is coming. In other words, if a robot is facing north, a robot directly west of it would have a heading of pi/2. Therefore, we can use this heading information to make an assessment of whether we should be turning, moving forward, moving backward, etc. If you look further down, you can see that we use the x component of the virtual force vector to define our spin and the y component to define forward/backward motion. So here, we see that the forward/backward motion is based on the cosine of the heading and the spin is based on the sine. That 10 in the equation is there as a tunable parameter weighting the strength of the contribution. The reason we subtract is because we want to move \***away**\* from the heading.

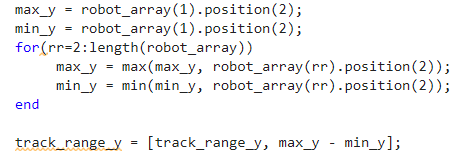
end

end

3.1:

* “But you implemented the algorithm and gave it to us. What’s to discuss?”
  + Explain to me in text how the algorithm is working. Show me that you understand the code that was provided

3.2:

* “How / where should I implement a cost function?”
  + The easiest place to implement a cost function would be in the main loop. Some examples might be: average distance from the light source for robots in shells 1/2/3. I’m not going to be picky with this: as long as you show some qualitative evidence of the robots doing what you want
  + A example of a simple one that tracks the spread in the y direction:
    - Right under robot\_radius = 1, put “track\_range\_y = [];” This is essentially creating a tracking variable that we’ll be adding to
    - After checking the exit condition:  
        
      Here we are essentially looping through all the robot positions and finding the maximum and minimum. We are then growing the tracking array using the last line, where we’re essentially saying “Update the tracking array to be itself with another number add another element to the array at the end.
    - At the end of the document,  
      

3.3:

* How do I make a video?
  + One option would be to use the functions I provided in assignment 1, but other options could be anything from using OBS to just taking a video on your phone

Section 4:

4.1:

4.2:

* “What sort of starting configurations?”
  + See the end of Lecture 6 for hints

4.3:

* “What would be a good cost function?
  + Average number of robots within communication range, Deviation from average alignment, etc might be good options. I’m not going to be picky with this. I’m basically going to look for some sort of quantitative assessment that you can show

4.4:

* How do I make a video?
  + One option would be to use the functions I provided in assignment 1, but other options could be anything from using OBS to just taking a video on your phone