Simulated Phagocytosis Using Swarm Robotics

Joe MacInnes
CS 200

April 30, 2018

Abstract

This paper simulates the process of phagocytosis by using a swarm of Kilobots. Alone, each Kilobot is unable to perform any meaningful function, but as a group they perform emergent behavior matching that of a phagocyte. The simulation captures the searching, movement, and engulfing behavior of a phagocyte, but fails to simulate the destruction of foreign bodies. The paper explores and applies themes found in Particle Swarm Optimization, Ant Colony Optimization, and the Wolfpack Algorithm. It is found that phagocytosis is similar to a general optimization problem. The algorithm implemented simulates phagocytosis, but could also be used as an optimization algorithm where the strength of a chemical gradient across a search space represents the objective function.

Contents

List of Figures

List of Algorithms

1 Introduction

Humans are built on a centralized intelligence paradigm; our brain is an individual authority that controls most things happening in our body, from the mundane bodily functions to critical thought. As a result, most of modern computing is built on the same paradigm. Consider, for example, that the ever-present Von Neumann architecture has a central processing unit. Or, in the case of networking, a fundamental structure is the client-server structure, in which a centralized server sends information to clients.

Decentralized intelligence, while less relatable for humans is another approach to intelligence in organisms and systems. This form of intelligence is best exemplified by social animals like ants, bees, and birds. Instead of individual actors making their own decisions or following some dedicated central authority, these groups act as a collective. For example in an ant colony, each ant follows a set of simple rules to determine their activity. They sense their surrounding environment and can tell what their neighbors are doing, and adjust their behavior accordingly. Alone, they are incapable of anything impressive, but, as a group or swarm, can perform complex, emergent behaviors like building a colony.

In 1986, Craig Reynolds created one of the first algorithms simulating natural swarm behavior when he wrote his Boids algorithm, which models the flocking of birds. Each agent in the algorithm represents a bird which moves by combining three simple rules: avoid neighbors, steer towards the average heading of all nearby birds, and steer towards the average position of all nearby birds. Following, his work, the field of swarm intelligence was initiated, in which the collective behavior of natural and artificial systems using decentralized intelligence are studied. For the purposes of this paper, however, swarm intelligence is considered in terms of artificial systems [7].

Since the 1980s, swarm intelligence algorithms have been used in a variety of applications. This paper explores a simulation of phagocytosis, an important biological process, by applying swarm intelligence strategies to a group of robotic agents. It first describes two famous swarm intelligence algorithm families as well as the more recent Wolfpack Algorithm before exploring the mechanics behind phagocytosis. Finally, it defines the approach and implementation of the simulation used to model it.

1.1 Motivations

Decentralized intelligence has many advantages compared to centralized intelligence models. Swarms tend to not have a central point of failure; destroying a swarm agent will not have a huge impact on the performance of the swarm.

Swarms are robust and scalable. Most, if not all, of the agents follow similar sets of simple

rules, so it is easy to add agents to a swarm and immediately have them start contributing to any emergent behavior [2].

Finally, because of an agent's simplicity, its performance cost, or manufacturing cost, in the case of physical swarm agents, can be much lower than centralized-intelligence style autonomous agents.

These attributes make swarm intelligence applicable to a process like phagocytosis. Phagocytosis is the ingestion (and in most cases, destruction) of bacteria or other foreign bodies at the hands of special cells known as phagocytes. Cells are constantly dying and being created, and certainly don't possess any real centralized intelligence or thought. Instead, groups of cells cooperate to perform more complex functions in higher order structures like organs. Thus, any endeavor to simulate such a biological process benefits from a reliance on swarm intelligence; the pursuit of which might yield improvement on current swarm intelligence methods or insight into medical mechanisms.

2 Swarm Intelligence Algorithms

Before discussing phagocytosis in depth, this paper delves into optimization and swarm intelligence algorithms used to perform optimization, since the topic has relevance to the way phagocytosis occurs.

2.1 The Optimization Problem

Optimization - the search for solutions to an optimization problem - is critical to any machine learning model. When analytical methods can't be used to find an optimal weight vector for a model (e.g. the weight vector of linear regression can be calculated), an optimization method is required to find a good weight vector. In any optimization problem, there is a search space defined by a set of variables, and an objective function that, given values for each of the unknowns, will compute a quantity that the optimization technique seeks to minimize or maximize.

For example, Figure 1 shows the surface for an example function, $\frac{\sin(5x)\cos(5y)}{x}$. The plot's surface represents the value of the function for any pair of unknowns, x and y. This particular objective function has an infinite number of global optima, since each bump shares the same maximum, and each dip shares the same minimum. With a slight amount of random jitter, this optimization problem could be much more complex since there would only be one global maxima and minima and many other local optima.

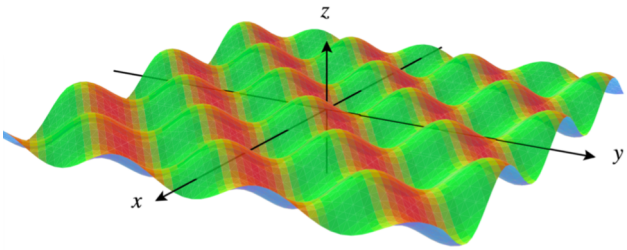

Figure 1: Surface plot of $\frac{\sin(5x)\cos(5y)}{x}$ [6]

One traditional optimization algorithm is gradient descent. In its simplest form, gradient descent first picks random values for each of the unknowns. Then, it calculates the partial derivative of the unknowns with respect to each unknown.

The resulting vector is the direction in which the gradient descent will step and then repeat the process. After this, it continues to take steps in the direction of the minimum objective value.

Gradient descent, however, is not without its problems. It can be slow to converge to an optimum, and can zig-zag towards the optimum depending on the shape of the objective function. In addition, its difficult to efficiently set the step size; too high, and the gradient descent will oscillate stepping over the optimum, too low, and it will never get there in time. Finally, it is hard to search the entire space effectively. If the objective function has many local optimum, then the gradient descent approach might easily get stuck in one near its initialization [3].

Swarm optimization algorithms can offer many advantages to traditional optimization techniques. Because of this, and the high demand for optimization algorithms, two algorithm families have come to dominate the swarm intelligence field: ant colony optimization (ACO) and particle swarm optimization (PSO).

2.2 Particle Swarm Optimization

PSO is an optimization technique used for searching a continuous search space. In every PSO algorithm, a swarm of particles blankets the search space, each of which is capable of calculating the value of the objective function at its position (i.e. its set of unknowns).The algorithm iterates through a series of discrete time steps, during which, the positions of the agents change. In each time step, a velocity is added to the current position of a particle, giving it a new position in the search space. This velocity is a combination of their previous velocity (initialized randomly at the start), a cognitive component, and a social component.

The cognitive component relates to a particle's memory; each particle keeps track of the location in which it has had the best objective function value. The cognitive component is proportional to the distance the particle currently is from that location. Thus, if the particle is far from its memorized best location, this has a large impact on the velocity that repositions the particle.

The social component is the optimum location across the particle's entire neighborhood (or the whole swarm if global PSO is used, see below). Similar to the cognitive component, the social component is proportional to the distance the particle currently is from that location. Thus, if the particle is far from the neighborhood's best location, this has a large impact on the velocity that repositions the particle. In order to make the algorithm stochastic, the social component and cognitive component are each multiplied by a random value between 0 and 1.

This is the basic structure of any PSO algorithm, but over time many variations have come to exist. One of the most important modifications is the way in which particles communicate. For example if each particle communicates with every other particle, in what is called a star social structure, then each particle always knows what the global best position is; its neighborhood is the whole swarm. This method converges very quickly, but can easily be trapped in local optima; each member of the swarm is drawn to the same location. If the global best happens to be in a local optima, that's where the swarm will gather. Thus, it's best for applications with a single optima.

Under another structure, the ring, each particle can only communicate with a finite number of nearest neighbors. Thus, multiple (overlapping) neighborhoods exist, each with their own local optima. This slows down the propagation of information among the swarm, effectively reversing the drawbacks and benefits of the star structure. A swarm member's social component will draw it towards the optima of its neighborhood instead of the optima of the entire swarm.

In addition to changing the communication structure there are a number of modifications that exist. Velocity clamping bounds the value that a particle velocity can be, which prevents the velocities from exploding, as they often do. The previous velocity of a particle can also be multiplied by an inertia weight to increase or decrease its impact on the velocity update - an inertia weight > 1 causes particles to accelerate toward convergence, while < 1 causes them to decelerate.

Overall, PSO offers much different optimization structure compared to gradient descent. While it is impossible to tell when PSO has found an optimum, it can cover a much greater search area. One of the greatest benefits compared to gradient descent is that PSO can be used in a search space where it is impossible to calculate partial derivatives, since the

velocity updates for each particle are based on distances alone [2].

2.3 Ant Colony Optimization

While PSO excels at continuous optimization problems, ACO is primarily used for discrete optimization problems. It also is more focused on probabilistic methods rather than velocities.

In an ant swarm, a common activity is foraging for food. Each ant moves toward the food by smelling the pheromones left by other ants on the surrounding ground. A simulation of this can be seen in Figure 2.

The figure depicts an ant colony foraging for food at three different time steps. The ants (black spheres) want to find the shortest of two paths from their home (blue square) to food (green square). In the initial time step, they randomly choose their path, with around half choosing the longer path.

Those ants on the shorter path return first, as seen in the second time step, and drop pheromones (green spheres) at that path's head. Once they've returned to their home, the ants probabilistically choose their path again, but this time with more weight on the shorter path. The third time step shows that, over time, the ants converge on the shortest path.

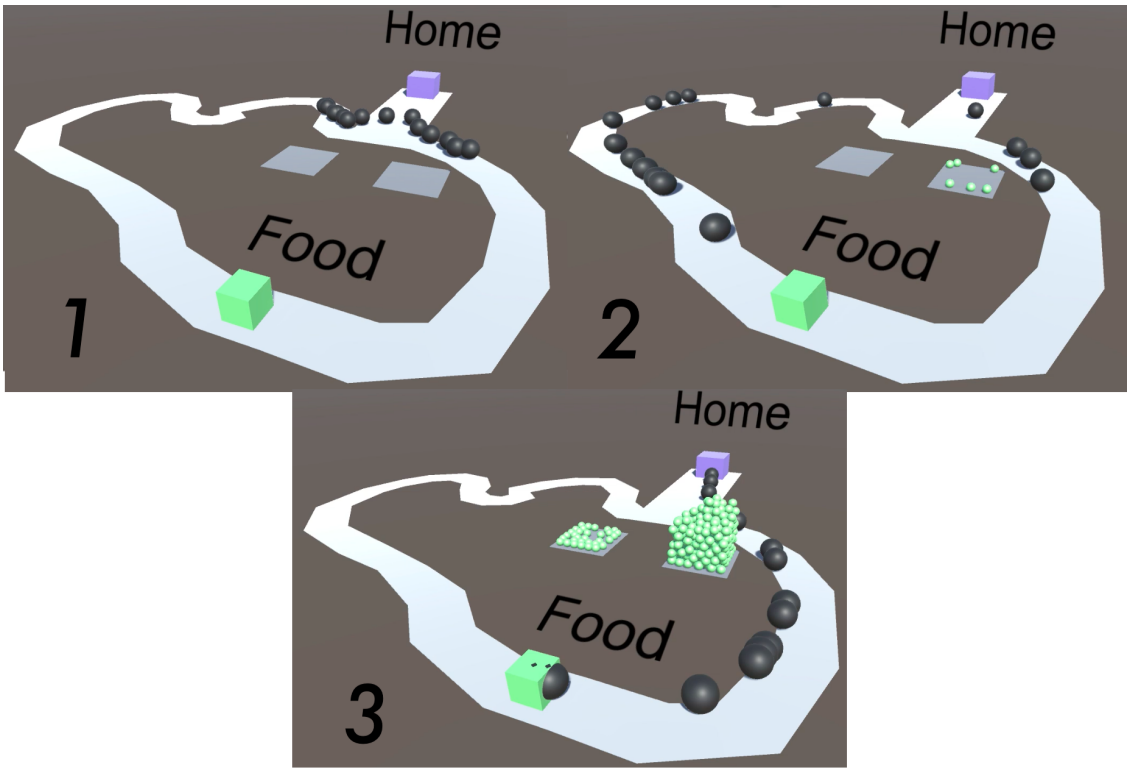

Figure 2: Ant Colony Optimization Simulation Using Unity

This behavior can be easily used to find shortest paths in graphs, and forms the basis for the simplest ACO algorithms. In these algorithms, agents in the swarm are analogous to ants. The swarm starts at a node in the graph and another node is designated as the food node. The edges from each node to another represent the paths the swarm agents can traverse. Each agent chooses a path based on the pheromone levels in the neighboring nodes.

Since the search space in phagocytosis is continuous, however, this is sufficient discussion on ACO [2].

2.4 Wolfpack Algorithm

A more recent addition in the world of swarm intelligence algorithms, the WPA borrows much more from PSO than ACO, and is used for continuous global optimization problems. The authors wrote the algorithm after they noticed the efficiency with which tundra wolves hunt. In it, each swarm agent represents a wolf, while the prey's scent is the objective function they seek to optimize (the prey's location is the highest scent concentration).

Unlike PSO or ACO, the WPA is not a homogeneous swarm - not all wolves are the same. The swarm agents can either be scouting wolves, ferocious wolves, besieging wolves, or the lead wolf. The swarm agent with the highest initial objective function becomes the lead wolf. This agent then calls all the other agents towards it, similar to a howling pack leader.

Other agents then enter the scouting behavior; they proceed to take steps in a random number of directions until they either find a higher objective function value (scent) than the lead wolf agent or hit a set number of max steps. If a scouting agent finds a higher objective function value, it becomes the new lead wolf. However, if it reaches the max number of scouting steps, it enters the ferocious state and moves towards the lead wolf, converging on the optimal value.

Agents conducting ferocious behavior keep moving towards the lead wolf until they are within a threshold distance (a parameter of the algorithm), at which point they move to the besieging behavior. In this final behavior, the agents continue taking steps towards the lead wolf while the steps increase their objective function value. If a step results in a lower objective function, they stop. When a besieging agent finds an objective function higher than the lead wolf agent, the prey is deemed 'found'. The agents with the lowest objective functions at this time are removed, and new agents are instantiated to take their place. In this way the algorithm simulates the winner-takes-all behavior of a wolf pack [11].

3 Phagocytosis

As stated earlier, phagocytosis is the biological process in which one cell ingests another body, be it a foreign particle, dead cell, or harmful bacteria. Once ingested, the phagocyte disposes of the body. Cells that engage in phagocytosis are known as phagocytes, and are a crucial line of defence in the body's immune system.

There are different kinds of phagocytes in the immune response and their exact role can vary. For example, macrophages (a white blood cell) digest bacteria and 'present' them; that is, once they digest the bacteria, they release its remains as antigens, which are important in building immunity. Neutrophils on the other hand (another kind of white blood cell) digest the bacteria, but don't present it.

Regardless of these variations in the immune response, a phagocyte's ultimate goal is to kill harmful bacteria. In order to find this bacteria, phagocytes rely on chemotaxis - movement of a cell in response to a chemical gradient. Phagocytes are eukaryotic cells, which are large enough that they can detect a chemical gradient across their body; receptor proteins cluster on one side of the cell. Often, the bacteria that a phagocyte is hunting releases a chemical creating a chemical gradient around them across which the phagocyte travels [10].

While the chemotaxis tell phagocytes *where* to go, it does not dictate their mechanism of motility - how they actually move. The method phagocytes use to move across chemical gradients is one employed by many other animal cells: cell crawling (see Figure 3). In this method, the cell traverses a substrate based on a series of adhesion and deadhesion. Because protein receptors cluster at an edge of the cell as it follows a chemotaxi, actin (a protein that helps maintain cellular shape) in the cytoskeleton of the cell polymerizes in this area, creating a protrusion. Adhesion molecules enter this protrusion and cause it to stick to the substrate. As this happens, the actin structure in the rear adhesion sites disassembles and the cell moves forward as the cytoskeleton contracts towards the adhesion sites [1].

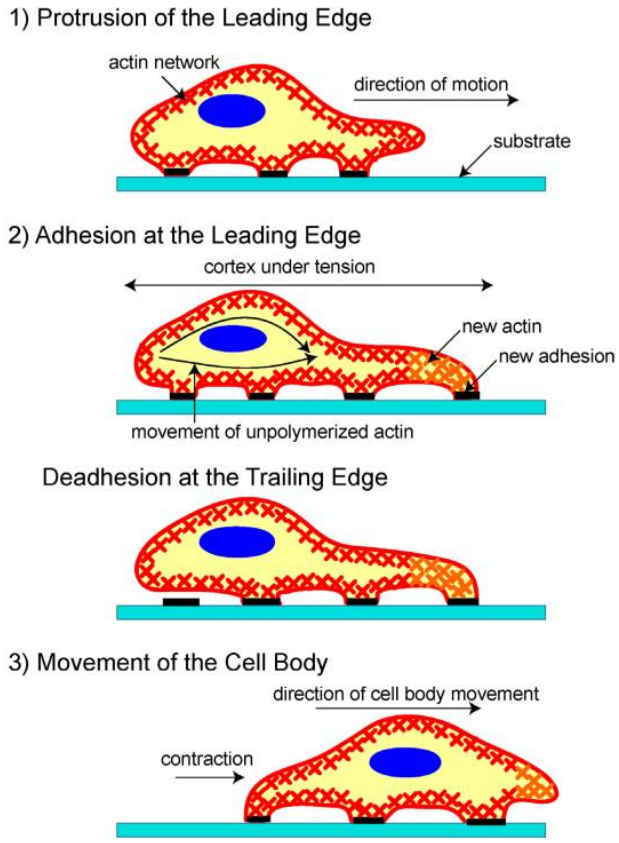

Figure 3: Process of Cell Crawling [1]

When the phagocyte comes into contact with the bacterium, it binds onto it using more protein receptors. It then proceeds to ingest the bacterium and kills it using oxygen full anti-microbial molecules created during the ingestion process or through a number of anaerobic methods [10].

4 Software

Following the description of phagocytosis as well as some algorithmic fundamentals that underly its simulation, this section explores the actual implementation of the simulation.

4.1 Swarm Robotics

The simulation uses a group of of robots operating with a rule-set that is swarm-like. Swarm robotics, the intersection of swarm intelligence and robotics, can be used to describe any robotic system in which there are multiple, simple robots whose coordination and interaction with the environment results in emergent behaviors.

4.2 Kilobots

The agents used in the simulation are Kilobots, a low-cost robot with a robust set of features developed by researchers at Harvard (see Figure 4 for an image). Kilobots are capable of stick-and-slip movement; they have three metal legs, which, when vibrated, can make the Kilobot rotate in a direction or travel forward. In addition, a Kilobot has the ability to detect ambient light and temperature. Each Kilobot also possesses an RGB LED. Finally, they can broadcast an infrared message roughly twice every second. Any Kilobot within in a 7cm radius can receive these messages. The message itself is 12 bytes long, 9 of which are dedicated to carrying data.

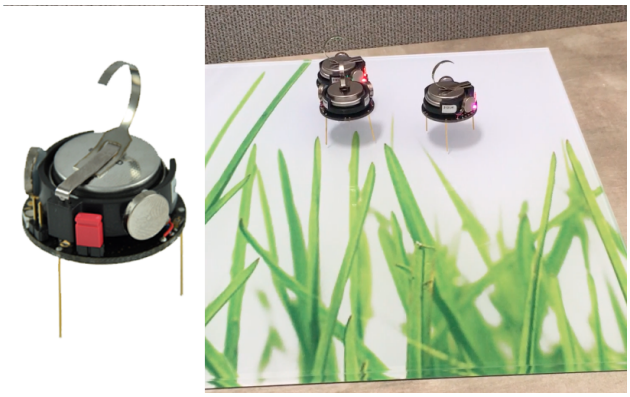

Figure 4: Example Kilobot [4] Alongside College of Wooster Kilobots

Operating with an AVR microcontroller (AVR is a family of microcontrollers developed by Atmel), Kilobots have 32kb of flash memory. Any new program is written to this storage via an overhead IR broadcasting board. The programs for a Kilobot are written in C and compiled using the avr-gcc compiler, which outputs files in HEX format - an ASCII representation of binary [4].

4.3 Description of Problem

Having outlined the capabilities and specifications of the Kilobots, it is important to define the problem in terms of these. Phagocytosis is a combination of global optimization and pattern formation. Phagocytes traverse a search space (e.g. the bloodstream) trying to maximize an objective function (e.g. the concentration of a chemical released by bad bacteria). Once they find a bacterium, they must engulf it, which is a similar process to pattern formation. Thus, the Kilobots are responsible for finding some sort of target, and cutting it off from the rest of the search space.

With this in mind, there are two general approaches one could take. In the first, the Kilobots traverse the search space in a dispersed manner, only coming together when they converge on a target. At this point, they are responsible for forming an appropriate pattern to surround the target's area and close it off from the rest of search space. This approach looks similar in many respects to PSO, but only loosely resembles phagocytosis, since the Kilobots aren't behaving like cells [10]. In addition, the search space actual phagocytes traverse is quite vast. An applied version of the simulation requires an infeasible swarm size in order to effectively maintain swarm communication between neighborhoods.

The second approach, which this project pursues, involves replicating individual phagocytes and their behavior. In this approach, groups of Kilobots make up distinct phagocytes. While traversing the search space, groups of Kilobots attempt to stick together and follow the method of cell crawling that phagocytes use. For the remainder of the paper, these groups are referred to as Kilocytes. As above, once a Kilocyte gets close enough to a target, its member Kilobots need to communicate and move to surround the target.

The stick-and-slip movement of each Kilobot requires that the search space for a Kilocyte is a flat, 2D space (a whiteboard is an appropriate surface). There are three methods that could be used simulate a chemical gradient on such a search space given each Kilobots sensory abilities: light, heat, or a secondary, static network of Kilobot's broadcasting gradient values. The last of these introduces many problems (e.g physical collision between broadcasting Kilobots and Kilocyte Kilobots), and temperature changes relatively slowly, leaving light as the most reasonable choice. Thus, each Kilocyte must move towards increased light in the search space and surround the sources of such light.

4.4 Implementation

Here, the implementation of the major roles of a Kilocyte - recognizing a gradient, moving across a gradient, and engulfing a target - are detailed.

4.4.1 Gradient

Each Kilocyte must be aware of the gradient of light along its body. This gradient is simulated by building upon existing gradient code for Kilobots [4]. In this pre-existing code, each Kilobot constantly broadcasts its own gradient value. A seed robot, who has a static gradient value of 0, is designated by manually calibrating its ID. All non-seed Kilobots grab gradient values from any incoming messages and set their gradient to the lowest received gradient incremented by 1. In this way Kilobots who receive a message from the seed Kilobot have a gradient value of 1, while those who can hear messages from these Kilobots, but not

the seed have a gradient value of 2 (and so on). The gradient number essentially tells each Kilobot how many hops it is from the seed.

If a Kilobot doesn't hear a repeat of the the lowest gradient value it has received in the last two seconds, it forgets this gradient and resets it based on current incoming messages. Thus, after pulling a Kilobot away from the seed by a few hops and leaving it for two seconds, it adjusts its gradient value accordingly. This behavior can be seen in Procedure 1.

Procedure 1 Basic Gradient Setting

```
1: while true do
2:     if ownID = seedRobotID then
3:         ownGradient ← 0
4:         continue
5:     minGradient ← ∞
6:     for all neighbor ∈ NeighborList do
7:         if neighbor.gradient < minGradient then
8:             minGradient ← neighbor.gradient
9:     ownGradient ← minGradient + 1
```

This model, however, is not enough for the Kilocyte. Here, to borrow from the terminology used in cell crawling, the seed robot represents the Kilocyte's adhesion site to the surface it traverses. Under the basic gradient setting, there is no way for this seed robot can change, since a Kilobot whose ID is chosen to be the seed ID, will always be the seed robot.

The Kilocyte, however, needs a way to dynamically change which of its Kilobots is the seed robot from which the the gradient is formed. When a Kilobot in the Kilocyte detects light higher than that of the current Kilobot acting as an adhesion site, it should become the new adhesion site.

In order to achieve this functionality, each Kilobot constantly collects light samples while broadcasting a message containing its gradient value, the highest light recording it knows about, its ID, and a time to live value. Under this setup, the maximum light recording in the entire Kilocyte will propagate throughout the Kilocyte and become known to each Kilobot. Then, a Kilocyte can make a decision about its gradient based on its own light recording compared to the highest known light recording. This behavior can be seen in Procedure 2.

The propogation of the highest known light recording amongst the Kilobots is shown in Figure 5. The numbers from top to bottom are the Kilobot's own recording, highest known recording, and time to live value.

Procedure 2 Adaptive Gradient Setting

```
1: while true do
2:     ProcessIncomingMessages()
3:     Broadcast(highestKnownRecording, ID, ownGradient, TTL)
4:     if ownRecording is highestKnownRecording then
5:         ownGradient ← 0
6:         continue
7:     minGradient ← ∞
8:     for all neighbor ∈ NeighborList do
9:         if neighbor.gradient < minGradient then
10:            minGradient ← neighbor.gradient
11:    ownGradient ← minGradient + 1
```

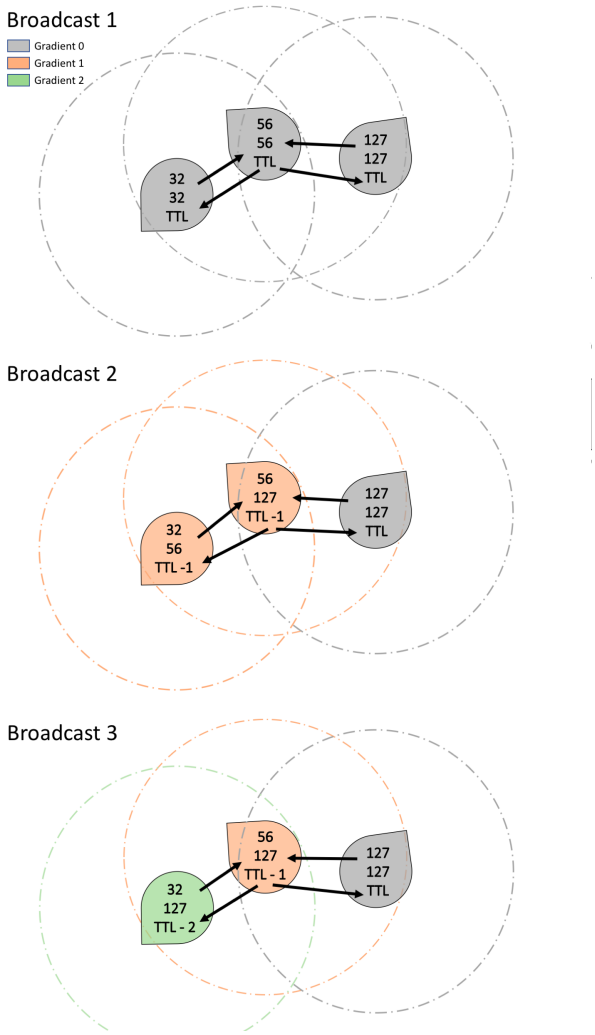

Figure 5: Light Reading Propogation in Kilobot Swarm.

4.4.2 Movement

While the Kilobot serving as the adhesion site determines the start of the light gradient across the Kilocyte, the other Kilobots must move towards it. This process is simulated by building upon pre-existing edge following code [5]. Since Kilobots have no sense of direction on their own, their movement is based on calculating their distance relative to other Kilobots and performing the appropriate behavior in relation to this - namely turning left or right, going forward, or stopping.

In the existing edge following code, one Kilobot moves along the edge of a group of static Kilobots who broadcast dummy messages so the mover can keep track of distances to its nearest neighbors. The moving Kilobot finds the nearest distance and turns right if it is below a certain threshold and left otherwise. Since it gains ground in the process of turning,

it oscillates between turning left and right as it moves back and forth over this threshold, thereby proceeding along the edge.

Once again, this model, is not enough for the Kilocyte, since *each* Kilobot must move towards the one serving as the adhesion point. In order to simulate the contraction of a phagocyte's cytoskeleton as it moves towards an adhesion point, the edge following code is changed to incorporate the gradient. Only, the Kilobots farthest away from the adhesion site Kilobot - in other words, those that form the 'back' of the Kilocyte - move at any given time. In addition, each Kilobot's orientation of left and right is randomized. In this way, not every Kilobot follows the edge of the Kilocyte in the same way. This prevents them from all swing around the adhesion Kilobot in the same direction.

At some point, a moving Kilobot following the edge moves beyond the adhesion site Kilobot, detects a higher light reading, and becomes the new adhesion site robot. In order for this new adhesion site to extend past the old, this new adhesion site Kilobot continues forward until the its nearest neighbor is far enough away. The movement behavior of the Kilobots can be seen in Procedure 3

Procedure 3 Kilobot Movement

```
1: leftMotor ← randomMotor
2: rightMotor ← otherMotor
3: while true do
4:     if ownGradient ≠ 0 then
5:         move ← true
6:         for all neighbor ∈ NeighborList do
7:             if (neighbor.isMoving) ∨ (neighbor.gradient > ownGradient) then
8:                 move ← false
9:         if move then
10:            if distanceNearestNeighbor < threshold then
11:                moveRight()
12:            else
13:                moveLeft()
14:        else
15:            stopMove()
16:    else
17:        if distanceNearestNeighbor < commRange / 1.25 then
18:            moveForward()
19:        else
20:            stopMove()
```

A final component to a Kilobot's movement behavior is momentum. When a Kilobot has no neighbors, it no longer attached to the Kilocyte. In order to reattach, it continues its current motion long enough to turn 180 degrees (with the default Kilobot turn speed, this is around 14 seconds). After this time, it moves forward until it reconnects (or doesn't as sometimes happens).

4.4.3 Engulfing

Kilobots lock into having a gradient value of 0 if they detect light above a certain threshold, even if this isn't the highest detected light of the swarm. This threshold is symbolic that it has reached the source of the light gradient and arrived at its target. In this manner, multiple Kilobots can be at gradient 0 once they reach the target. Other Kilobots will continue the movement behavior, following the edge until they too read light above the threshold. In this manner, Kilobots slowly travel around the edge of the light source, locking into place.

The value of the threshold determines how far away from the light source Kilobots begin encirclement.

4.4.4 Destruction

The Kilobots do not perform any destructive behavior once the light source is encircled. They have no analog to a real phagocyte's production and application of antimicrobial molecules to ingested bacteria.

The overall decision making of a Kilobot combines the gradient setting, movement, and engulfing behaviors described, is summarized in Figure 6.

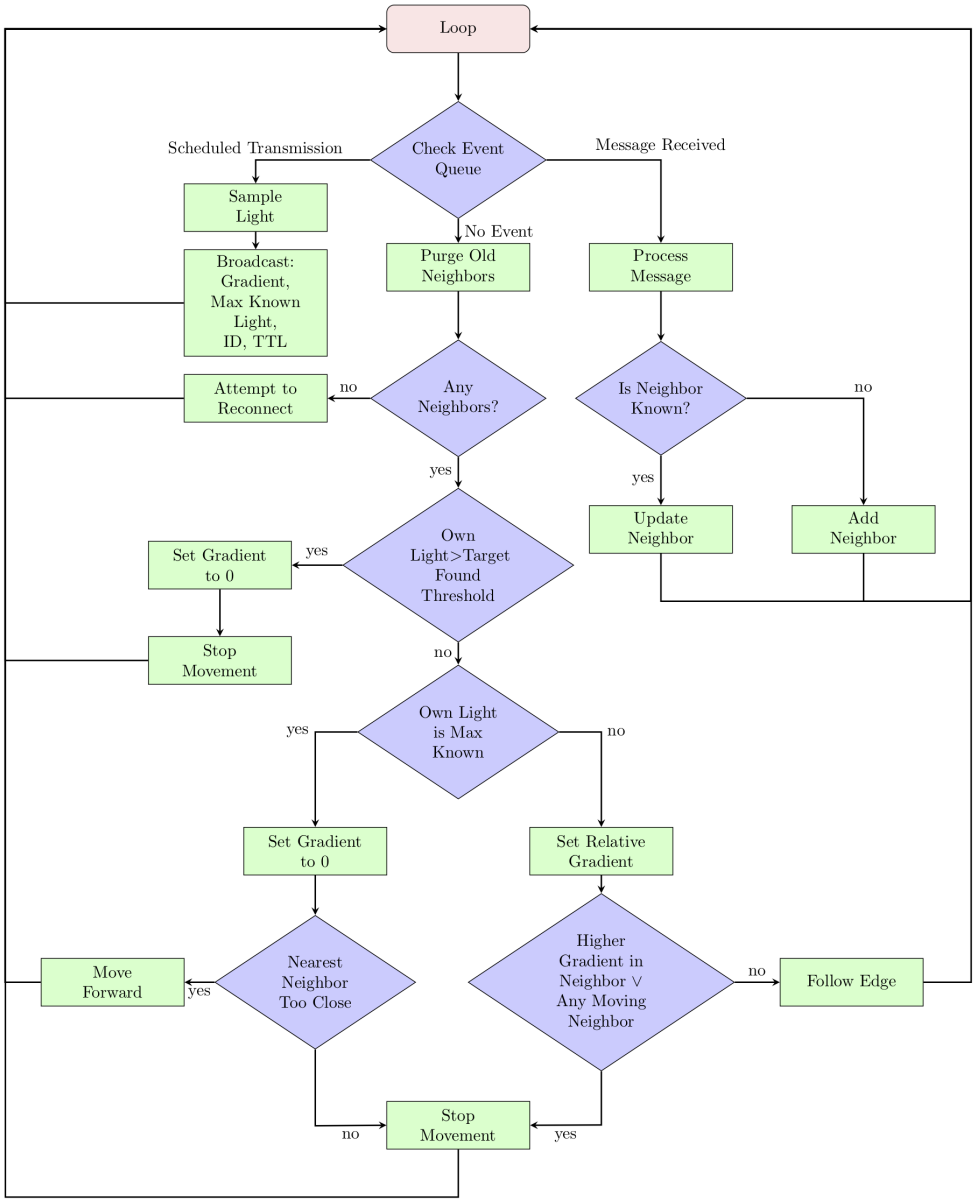

Figure 6: Flowchart a Kilobot's Behavior

4.5 Kilombo

It is important to note that during the implementation of the simulation, a change was made from using real Kilobots to using Kilombo, a Kilobots simulator written in C [5]. The change was made for two reasons.

The first of these was consistency and debugging purposes. Since Kilobots are physical robots, they require calibration, and the stick-and-slip movement style is not the most precise science. The Kilobots often need re-calibration, and only move consistently on the smoothest

of surfaces. In addition, the only way to get output from the Kilobots was to use their serial output ports and plugging them directly into a computer. Kilombo Kilobots, however, offer a much more forgiving testing environment. A Kilobot turning left will actually turn left instead of angrily vibrating backwards off of the search space, and the program can be tested as soon as it is compiled. Debugging information from a Kilobot also appears inside the Kilombo GUI while the simulation is running.

Secondly, Kilombo easily allows for the addition of Kilobots to a simulation. Only 10 physical Kilobots were available for the project.

Several additional libraries are required to run the simulator, but a port of a native Kilobot program will look very similar to the native program.

Figure 7 shows an example of what the simulator looks like while running. The simulator is essentially a birds-eye view of a surface containing Kilobots. Each circle represents a single Kilobot, while the connecting lines show a Kilobot's neighbors. The color of the Kilobot is equivalent to its LED. The text at the top is metrics concerning the simulator, while the text on the bottom is debugging information for a single Kilobot.

The Kilombo simulator is not capable of rendering a light gradient, but can simulate one. Thus, for the rest of the paper, this light gradient is drawn using photoediting tools to provide visualization.

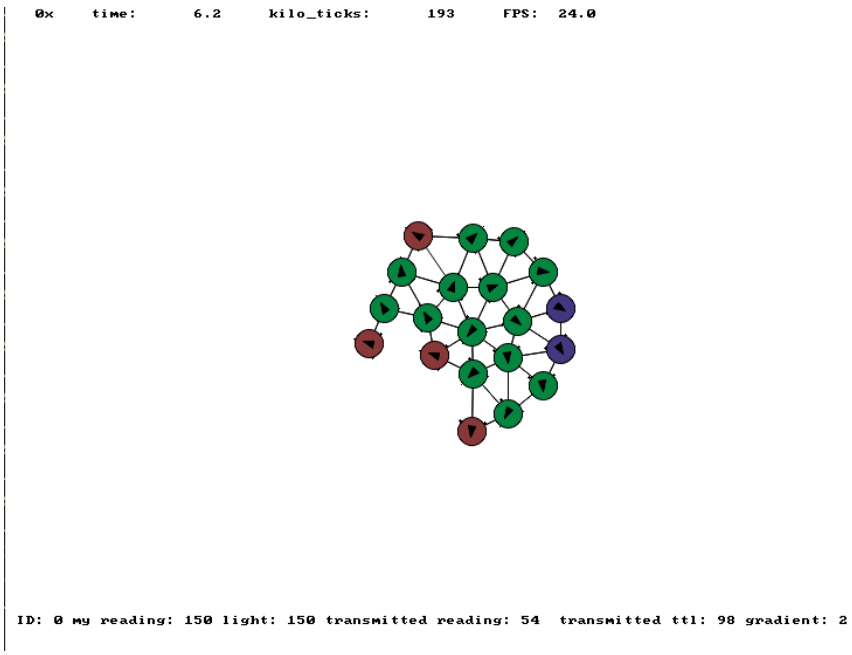

Figure 7: Screenshot of Kilombo Simulator.

4.6 Results

The implemented phagocytosis program is run in simulations using both a single light source and two light sources. The results of these simulations are compared to identical environments in which Kilobots independently move towards light. Before discussing these results, it is important to note that, in the simulations of phagocytosis, Kilobots light up the following colors to indicate status:

1. **Blue** The Kilobot has a gradient of 0. It could be an adhesion site, broadcasting its own light reading, or a Kilobot that has locked into place because its own light recording is above a set threshold.

2. **Green** The Kilobot has a gradient > 0. However, it detects either moving neighbors, or neighbors with a higher gradient, so it doesn't move.

3. **Red** The Kilobot has a gradient > 0. It detects no moving neighbors or neighbors with a higher gradient, so it follows the edge of the Kilocyte.

4. **Gray** The Kilobot detects no neighbors, and will attempt to turn 180 degrees before proceeding forward, in an effort to reconnect with the Kilocyte.

The first simulation is conducted using a single light source and a swarm of 18 Kilobots running the phagocytosis program. A visualization of this simulation is shown in Figure 8. As stated previously, the gradient seen and depiction of a bacteria are purely to aid the reader in observing the behavior, since the simulator cannot render the light gradient.

The visualization is split into four time steps, each taken at roughly quarterly intervals of the simulation. In the first time step, the maximum reading amongst all the Kilobots has propagated throughout the swarm, and several Kilobots are following the edge of the Kilocyte towards the adhesion site Kilobot.

The second and third time steps show the Kilocyte's progress across the search space, while the fourth depicts their convergence around the light source. In this simulation, it took 28 minutes for the Kilocyte to engulf the light source.

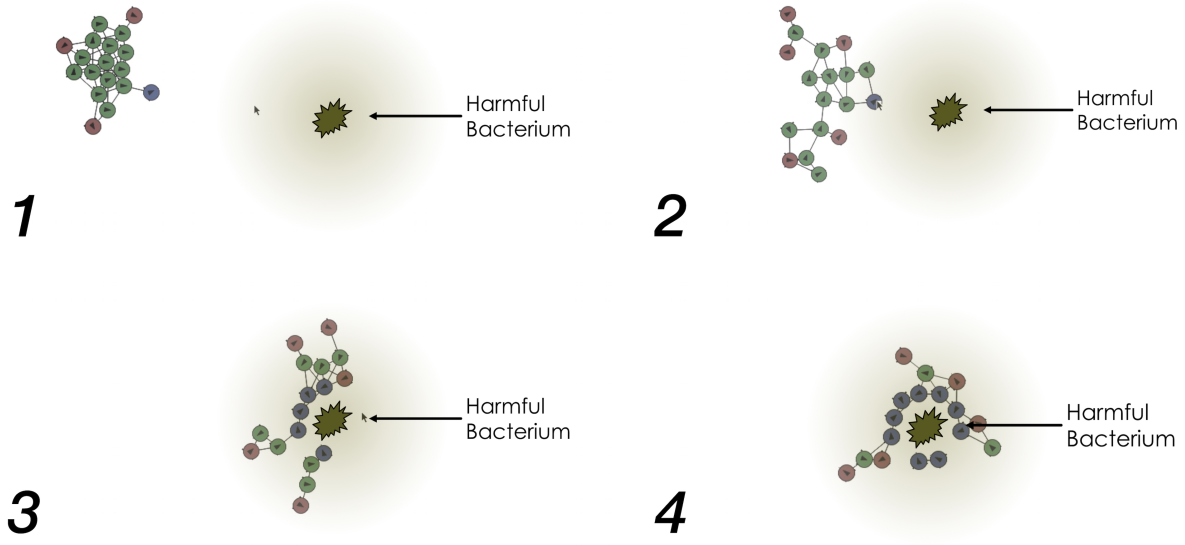

Figure 8: Phagocytosis Simulation With Single Target

The next simulation features a swarm of 18 Kilobots and single light source each in the same spot as the previous simulation. In this simulation, however, the Kilobots are moving towards the light independent of one another.

A visualization of this is shown in Figure 9. Again, the visualization is split into roughly quarterly intervals. Here, the Kilobots converged much more quickly on the light, only taking 5 minutes.

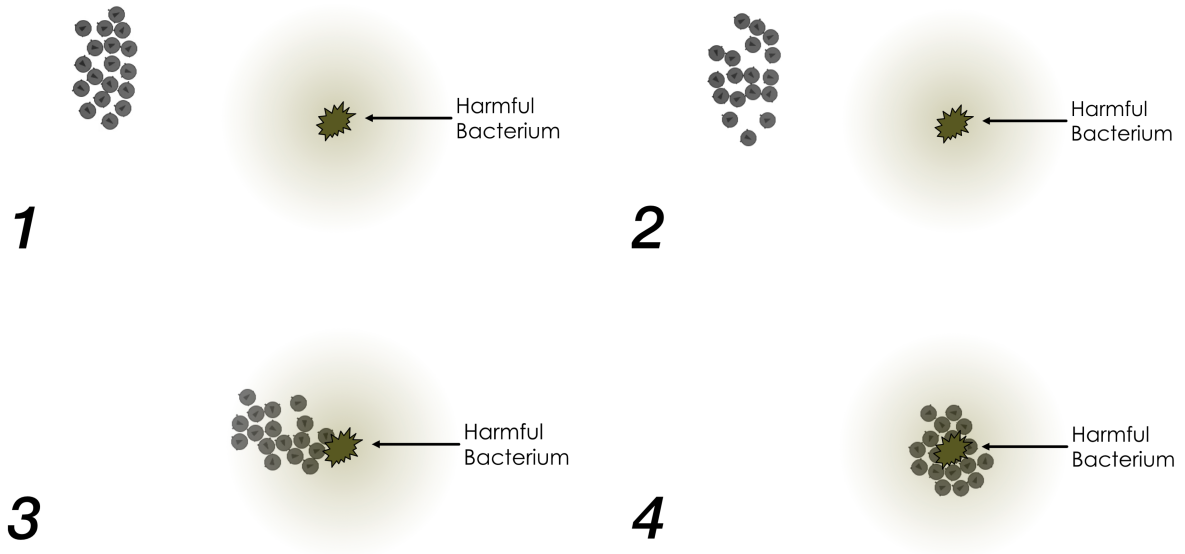

Figure 9: Kilobots Independently Move to Light With Single Light Source.

Despite the increase in speed, a critical flaw is revealed in this approach when more than one light source is considered. Figure 10 shows a visualization of a simulation in which 18 Kilobots independently move towards light in a search space featuring two lights. Again, the visualization is split into roughly quarterly intervals.

Here, the swarm splits, as Kilobots converge on both lights. One could imagine a search space with many more lights, where each Kilobot moves towards a unique light.

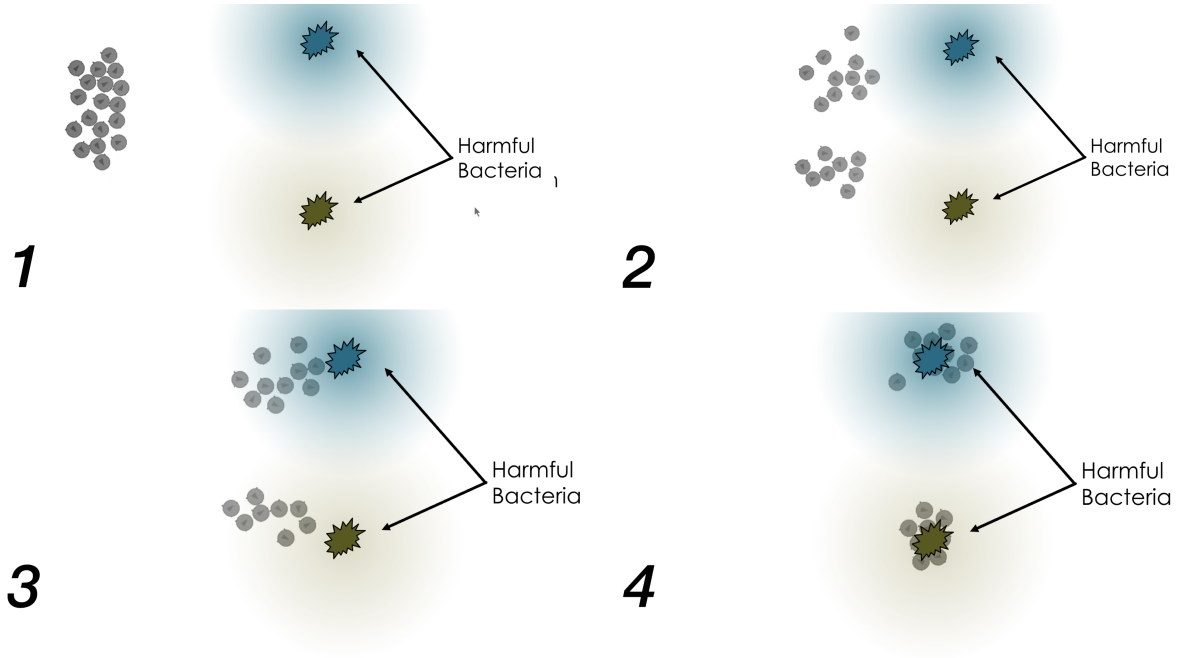

Figure 10: Kilobots Independently Move to Light With Two Light Sources.

In the same search space, however, a swarm of 18 Kilobots running the phagocytosis program is shown to converge on a single light source (see Figure 11). Because there is usually only one adhesion site Kilobot in a Kilocyte (there can be more than one if Kilobots share a light recording of the same value), the rest of the Kilobots are drawn to that Kilobot. Thus, the Kilobots in the Kilocyte are drawn to a single source.

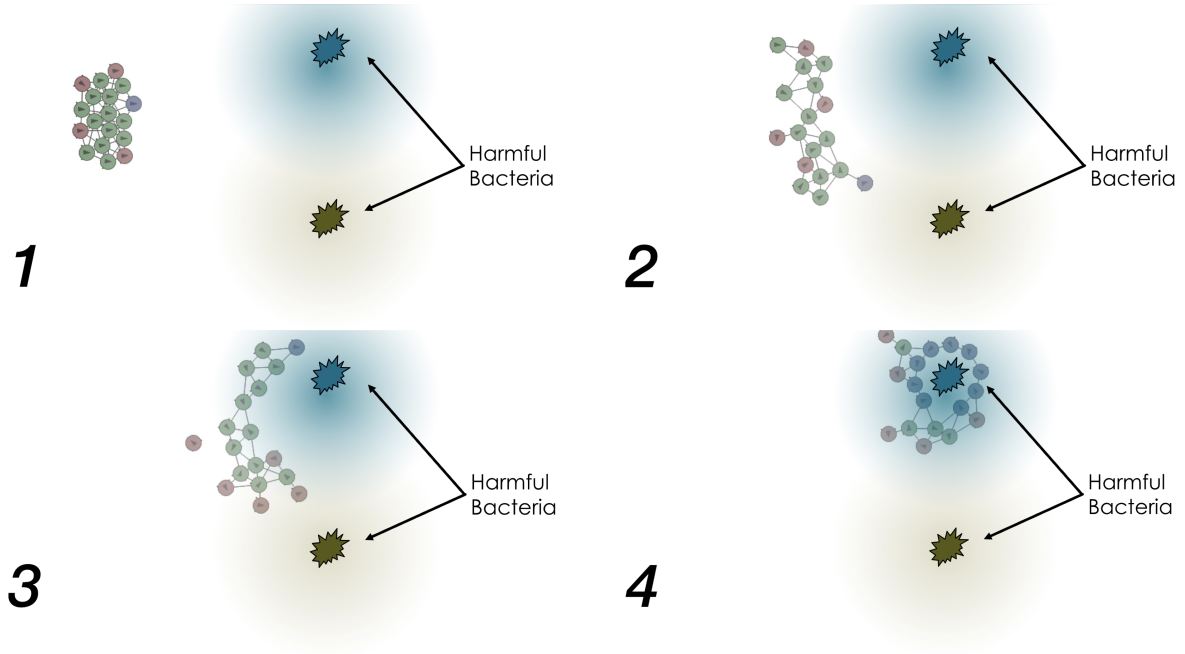

Figure 11: Phagocytosis Simulation With Two Targets.

Having compared those four simulations, Figure 12 shows the adaptability and scalability of a swarm of Kilobots running the phagocytosis program. It is divided into four frames. The first shows a healthy swarm. In the second, the swarm loses it's adhesion site Kilobot. Confusion ensues, but because of the time to live values being broadcasted in the swarm, a new adhesion site is chosen in frame 3. Frame 4 shows an Kilobot being added to the swarm and successfully integrating.

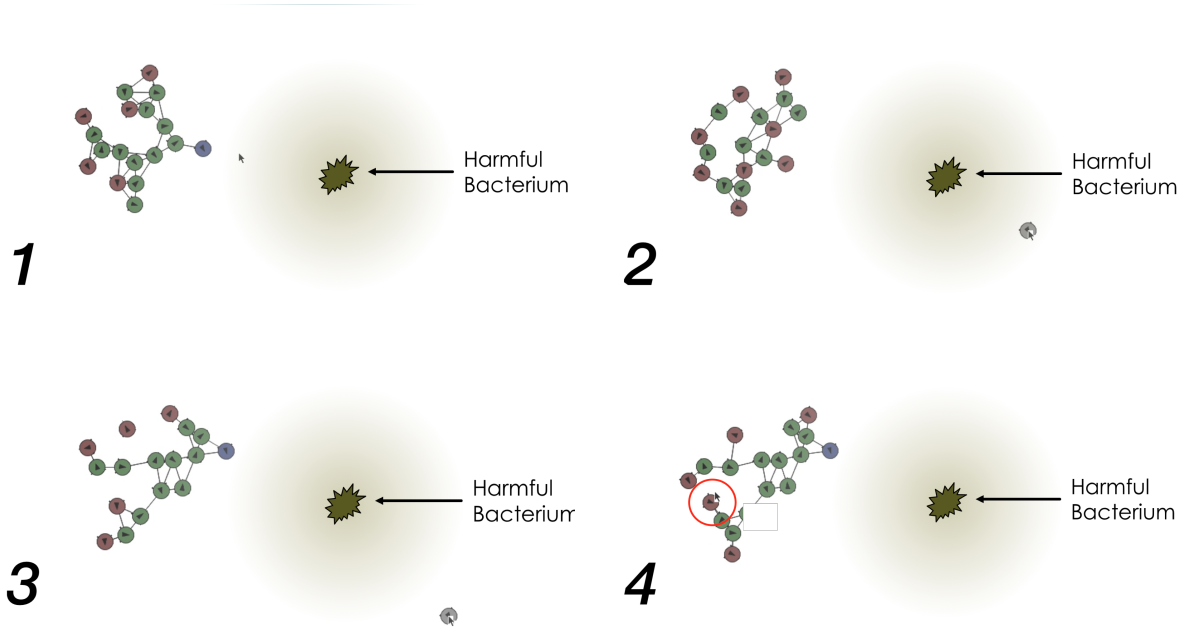

Figure 12: Scalability and Adaptability of Kilocyte.

5 Conclusion

As of right now, the implemented program successfully simulates a phagocytes behavior using a swarm of Kilobots up to the destruction of the ingested body. The Kilocyte's traversal of the search space follows patterns found in cell crawling, and it manages to completely engulf a light source. In future work, the destruction of the ingested light source should be an added priority.

The algorithm itself somewhat resembles the WPA. The Kilobots acting as adhesion sites are similar to the lead wolves in the WPA, calling the rest of the Kilobots to their location. The Kilobots being called, as they traverse the search space, also have the opportunity to become the new adhesion site, just like the scouting and ferocious wolves in the WPA.

It is interesting to see that the hunting patterns of wolf packs can be roughly translated down to the cellular level.

The algorithm might offer merit as a swarm optimization technique of its own. The Kilocyte swarm is, because of its traversal technique, resilient to being trapped in local optima. Unlike the Kilobots independently moving towards light, the Kilocyte can scout out in new directions as its component Kilobots move along the Kilocyte's edge.

Finally, with sustained interest in the field of nanotechnology, this algorithm could prove useful for augmented immune response. If Kilobot-like robots could be made small enough,

then perhaps some refined version of this algorithm might be used to introduce artificial phagocytes into the bloodstream.

References

[1] Revathi Ananthakrishnan and Allen Ehrlicher. The forces behind cell movement. *International journal of biological sciences*, 3(5):303, 2007.

[2] A.P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. Wiley, 2005.

[3] Nigel Goddard. Lecture notes on gradient descent. University of Edinburgh, November 2017.

[4] Self-Organizing Systems Research Group. Kilobots labs. Harvard University.

[5] Fredrik Jansson, Matthew Hartley, Martin Hinsch, Ivica Slavkov, Noemí Carranza, Tjelvar SG Olsson, Roland M Dries, Johanna H Grönqvist, Athanasius FM Marée, James Sharpe, et al. Kilombo: a kilobot simulator to enable effective research in swarm robotics. *arXiv preprint arXiv:1511.04285*, 2015.

[6] Ben Joffe. Function 3d examples, 2018.

[7] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[8] Erol Sahin, Thomas H. Labella, Vito Trianni, J.-L. Deneubourg, Philip Rasse, Dario Floreano, Luca Gambardella, Francesco Mondada, Stefano Nolfi, and Marco Dorigo. SWARM-BOT: Pattern formation in a swarm of self-assembling mobile robots. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, volume 4, pages 6–pp. IEEE, 2002.

[9] N E Shlyakhov, I V Vatamaniuk, and A L Ronzhin. Survey of Methods and Algorithms of Robot Swarm Aggregation. *Journal of Physics: Conference Series*, 803:012146, January 2017.

[10] Carel J. van Oss. [1] phagocytosis: An overview. In *Immunochemical Techniques Part J*, volume 132 of *Methods in Enzymology*, pages 3 – 15. Academic Press, 1986.

[11] Hu-Sheng Wu and Feng-Ming Zhang. Wolf Pack Algorithm for Unconstrained Global Optimization. *Mathematical Problems in Engineering*, 2014:1–17, 2014.

A Code Appendix

A.1 phagobot.h

```
1  /*
2   * Joe MacInnes
3   * April 29, 2018
4   *
5   * This file contains the neighbor data
6   * type definition as well as helper macro definitions
7   * for the Kilombo simulator
8   */
9
10 #ifndef GRN_H
11 #define GRN_H
12 #include <math.h>
13 #ifndef M_PI
14 #define M_PI 3.14159265358979323846264338327950288419716939937510582097494
15 #endif
16 #define GENES 6
17 #define DIFFGENES 6  // Maximal number of diffusing genes, limited by the mes
18 #define RB_SIZE 16  // Ring buffer size. Choose a power of two for faster cod
19                     // memory usage: 16*RB_SIZE
20                     // 8 works too, but complains in the simulator
21                     // when the bots are very dense
22 #define MAXN 20 // Max number of neighbors
23 #define TTL 100 // max times an original light reading can transmit
24 extern uint8_t NGenes;
25 enum BOTTYPE {LAST, FOLLOWER, LEADER};
26 enum BOTSTATES {WAIT, LISTEN, MOVE};
27 typedef struct {
28   uint8_t dist; // distance to neighbor
29   uint8_t n_bot_state;
30   uint8_t N_Neighbors; // number of neighbors
31   uint8_t grad; // gradient of neighbor
32   uint16_t ID; // kilo_uid of neighbor
33   uint16_t ttl; // time to live of light  reading sent by neighbor
34   uint16_t reading; // light reading sent by neighbor
35   uint32_t timestamp; // last time heard from neighbor
36 } Neighbor_t;
37 typedef struct {
38     message_t msg;
39     distance_measurement_t dist;
40 } received_message_t;
41
```

```
42 // Ring buffer operations. Taken from kilolib's ringbuffer.h
43 // but adapted for use with mydata->
44
45 // Ring buffer operations indexed with head, tail
46 // These waste one entry in the buffer, but are interrupt safe:
47 //   * head is changed only in popfront
48 //   * tail is changed only in pushback
49 //   * RB_popfront() is to be called AFTER the data in RB_front() has been us
50 //   * head and tail indices are uint8_t, which can be updated atomically
51 //     - still, the updates need to be atomic, especially in RB_popfront()
52
53 #define RB_init() {       \
54     mydata->RXHead = 0; \
55     mydata->RXTail = 0;\
56 }
57
58 #define RB_empty() (mydata->RXHead == mydata->RXTail)
59
60 #define RB_full()  ((mydata->RXHead+1)%RB_SIZE == mydata->RXTail)
61
62 #define RB_front() mydata->RXBuffer[mydata->RXHead]
63
64 #define RB_back() mydata->RXBuffer[mydata->RXTail]
65
66 #define RB_popfront() mydata->RXHead = (mydata->RXHead+1)%RB_SIZE;
67
68 #define RB_pushback() {\
69     mydata->RXTail = (mydata->RXTail+1)%RB_SIZE;\
70     if (RB_empty())\
71       { mydata->RXHead = (mydata->RXHead+1)%RB_SIZE;      \
72         printf("Full.\n"); }                              \
73   }
74
75
76 #endif
```

A.2 phagobot.c

```
1  /*
2   * Joe MacInnes
3   * April 29, 2018
4   *
5   * This program simulates the process of phagocytosis.
6   * It borrows some code from the follow_edge and gradient
7   * examples found at
8   * https://github.com/JIC-CSB/kilombo
9   *
10  * It is designed to run on the Kilombo simulator.
11  *
12  * To run the program requires installing the Kilombo
13  * simulator from the above link and compiling the code
14  * using the provided Makefile
15  *
16  * Once the executable is built
17  * './phagobot -b bot_pos.json'
18  * Will run the program using the simulator parameters
19  * and starting bot positions found in kilombo.json
20  * and bot_pos.json
21  */
22
23 #include <math.h>
24 #include <stdbool.h>
25 #include <kilombo.h>
26 #include "phagobot.h"
27
28 enum {STOP,LEFT,RIGHT,STRAIGHT};
29
30 typedef struct
31 {
32   Neighbor_t neighbors[MAXN]; // list of neighbor information
33   int N_Neighbors; // number of neighbors
34   uint8_t bot_type;
35   uint8_t bot_state;
36   uint8_t move_type;
37   uint8_t RXHead, RXTail;  // used for simulated message processing
38   uint16_t my_reading; // own light reading
39   uint32_t momentum_check; // time kilobot has been alone
40   int own_gradient; // hops from adhesion site kilobot
41   int min; // index of neighbor with highest light reading
42   bool lock; // flag for adhesion kilobot movement
43   bool lost; // flag for no neighbors
```

```
44    char message_lock; // prevents sending of message
45    message_t transmit_msg; // message data object
46    received_message_t RXBuffer[RB_SIZE]; // usde for simulated message process
47  } bot_info;
48
49  REGISTER_USERDATA(bot_info)
50
51  #ifdef SIMULATOR
52  #include <stdio.h>     // for printf
53  #else
54  #define DEBUG          // for printf to serial port
55  #include "debug.h"
56  #endif
57
58  // message rx callback function. Pushes message to ring buffer.
59  void rxbuffer_push(message_t *msg, distance_measurement_t *dist) {
60      received_message_t *rmsg = &RB_back();
61      rmsg->msg = *msg;
62      rmsg->dist = *dist;
63      RB_pushback();
64  }
65
66  message_t *message_tx()
67  {
68    if (mydata->message_lock)
69      return 0;
70    return &mydata->transmit_msg;
71  }
72
73  /* Accessor and mutator functions for bot state and move type
74   */
75  void set_bot_state(int state)
76  {
77    mydata->bot_state = state;
78  }
79
80  int get_bot_state(void)
81  {
82    return mydata->bot_state;
83  }
84
85  void set_move_type(int type)
86  {
87    mydata->move_type = type;
88  }
```

```
89
90  int get_move_type(void)
91  {
92    return mydata->move_type;
93  }
94
95  /* Process a received message at the front of the ring buffer.
96   * Go through the list of neighbors. If the message is from a bot
97   * already in the list, update the information, otherwise
98   * add a new entry in the list
99  */
100
101 void process_message()
102 {
103   mydata->lost = false; // Received a message from neighbors
104   uint8_t i;
105   uint16_t ID, recv_reading;
106   uint8_t *data = RB_front().msg.data;
107   // Extract ID from message
108   ID = data[0] | (data[1] << 8);
109   recv_reading = (data[4] << 8) | (data[5]);
110
111   uint8_t d = estimate_distance(&RB_front().dist);
112
113   // Search the neighbor list by ID
114   for (i = 0; i < mydata->N_Neighbors; i++)
115     if (mydata->neighbors[i].ID == ID)
116       {
117         // found it
118         break;
119       }
120
121   if (i == mydata->N_Neighbors){   // this neighbor is not in list
122     if (mydata->N_Neighbors < MAXN-1) // if we have too many neighbors,
123       mydata->N_Neighbors++;          // we overwrite the last entry
124                                       // sloppy but better than overflow
125   }
126   // i now points to where this message should be stored
127   mydata->neighbors[i].ID = ID;
128   mydata->neighbors[i].timestamp = kilo_ticks;
129   mydata->neighbors[i].dist = d;
130   mydata->neighbors[i].N_Neighbors = data[2];
131   mydata->neighbors[i].n_bot_state = data[3];
132   mydata->neighbors[i].reading = recv_reading;
133   mydata->neighbors[i].ttl = data[7] - 1;
```

```
134   mydata->neighbors[i].grad = data[6];
135 }
136
137
138 /* Finds the minimum light reading by comparing all
139  * neighbors' readings to own. Returns the index of neighbor
140  * with lowest light reading and highest time to live of
141  * reading or -1 if this kilobot has lowest light
142  */
143 int min_reading(){
144   int index = -1;
145   uint16_t min = mydata->my_reading;
146   uint16_t max_ttl = 0;
147   for (int i = 0; i < mydata->N_Neighbors; i++){
148     if (((mydata->neighbors[i].reading < min) && mydata->neighbors[i].ttl > 0
149     {
150       max_ttl = mydata->neighbors[i].ttl;
151       min = mydata->neighbors[i].reading;
152       index = i;
153     }
154   }
155   if (min == mydata->my_reading){
156     return -1;
157   }
158   return index;
159 }
160
161 /* Sets the gradient value of the kilobot by searching
162  * through all neighbors finding the lowest gradient
163  * and setting own_gradient to that value + 1.
164  * Forces gradient to be 0 if the kilobot's own light
165  * reading is below a hardcoded threshold (in this case 70)
166  */
167 void set_gradient(){
168   int min_grad = mydata->own_gradient;
169   if (mydata->my_reading < 70)
170     mydata->own_gradient = 0;
171   else if (mydata->min != -1){
172     for (int i = 0; i < mydata->N_Neighbors; i++){
173       if (mydata->neighbors[i].grad < min_grad)
174         min_grad = mydata->neighbors[i].grad;
175     }
176     mydata->own_gradient = min_grad + 1;
177   }
178   else
```

```
179     mydata->own_gradient = 0;
180 }
181
182 /* Go through the list of neighbors, remove entries older than a threshold,
183  * currently 2 seconds.
184  */
185 void purgeNeighbors(void)
186 {
187   int8_t i;
188   for (i = mydata->N_Neighbors-1; i >= 0; i--){
189     if (kilo_ticks - mydata->neighbors[i].timestamp  > 64){
190       //this one is too old.
191       mydata->neighbors[i] = mydata->neighbors[mydata->N_Neighbors-1];
192       //replace it by the last entry
193       mydata->N_Neighbors--;
194     }
195   }
196 }
197
198 void setup_message(void)
199 {
200   mydata->message_lock = 1;  //don't transmit while we are forming the messag
201   mydata->transmit_msg.type = NORMAL;
202   mydata->transmit_msg.data[0] = kilo_uid & 0xff;      // 0 low  ID
203   mydata->transmit_msg.data[1] = kilo_uid >> 8;        // 1 high ID
204   mydata->transmit_msg.data[2] = mydata->N_Neighbors; // 2 number of neighbor
205   mydata->transmit_msg.data[3] = get_bot_state();      // 3 bot state
206   mydata->my_reading = get_ambientlight();             // record current light
207   mydata->min = min_reading();
208   if (mydata->min != -1){ // broadcast own light reading with full TTL
209     mydata->transmit_msg.data[4] = mydata->neighbors[mydata->min].reading >>
210     // 4 low light reading
211     mydata->transmit_msg.data[5] = 0 |  mydata->neighbors[mydata->min].readin
212     // 5 high light reading
213     mydata->transmit_msg.data[7] = mydata->neighbors[mydata->min].ttl;
214     // 7 time to live
215     }
216   else{ // broadcast neighbor's light reading with their TTL
217     mydata->transmit_msg.data[4] = mydata->my_reading >> 8;
218     // 4 low light reading
219     mydata->transmit_msg.data[5] = 0 | mydata->my_reading;
220     // 5 high light reading
221     mydata->transmit_msg.data[7] = TTL;
222     // 7 time to live
223     }
```

```
224   mydata->transmit_msg.data[6] = mydata->own_gradient; // 6 gradient value
225   mydata->transmit_msg.crc = message_crc(&mydata->transmit_msg);
226   mydata->message_lock = 0;
227 }
228
229 void setup()
230 {
231   rand_seed(kilo_uid + 1); //seed the random number generator
232   mydata->message_lock = 0;
233   mydata->own_gradient = 0;
234   mydata->min = 0;
235   mydata->lost = false;
236   mydata->my_reading = get_ambientlight();
237   mydata->N_Neighbors = 0;
238   set_move_type(STOP);
239   set_bot_state(LISTEN);
240   setup_message();
241 }
242
243 void receive_inputs()
244 {
245   while (!RB_empty())
246   {
247     process_message();
248     RB_popfront();
249   }
250   purgeNeighbors();
251 }
252
253 /*
254  * Returns the distance of the nearest neighbor
255  */
256 uint8_t find_nearest_N_dist()
257 {
258   uint8_t i;
259   uint8_t dist = 90;
260
261   for(i = 0; i < mydata->N_Neighbors; i++)
262   {
263     if(mydata->neighbors[i].dist < dist)
264     {
265       dist = mydata->neighbors[i].dist;
266     }
267   }
268   return dist;
```

```
269 }
270
271 /*
272  * Returns the distance of the nearest neighbor
273  */
274 uint8_t find_farthest_N_dist()
275 {
276   uint8_t i;
277   uint8_t dist = 0;
278   for(i = 0; i < mydata->N_Neighbors; i++)
279   {
280     if(mydata->neighbors[i].dist > dist)
281     {
282       dist = mydata->neighbors[i].dist;
283     }
284   }
285   return dist;
286 }
287
288 /*
289  * Follows the edge of the 'cell' of kilobots.
290  * If too close to any neighbor, the kilobot turns one direction
291  * if far enough away, the kilobot turns the opposite direction
292  */
293 void follow_edge()
294 {
295   uint8_t desired_dist = 42;
296   set_bot_state(MOVE);
297   // currently using modulo 2 so that half the kilobots
298   // the edge in one direction and the other half in the
299   // other direction. This NEEDS to change, very sloppy code
300   if(kilo_uid % 2 == 1){
301   if(find_nearest_N_dist() > desired_dist)
302   {
303     if(get_move_type() == LEFT)
304       spinup_motors();
305     set_motors(0, kilo_turn_right);
306     set_move_type(RIGHT);
307   }
308   else
309   {
310     if(get_move_type() == RIGHT)
311       spinup_motors();
312     set_motors(kilo_turn_left, 0);
313     set_move_type(LEFT);
```

```
314       }
315     }
316     else{
317       if(find_nearest_N_dist() > desired_dist)
318       {
319         if(get_move_type() == RIGHT)
320           spinup_motors();
321         set_motors(kilo_turn_left, 0);
322         set_move_type(LEFT);
323       }
324       else
325       {
326         if(get_move_type() == LEFT)
327           spinup_motors();
328         set_motors(0, kilo_turn_right);
329         set_move_type(RIGHT);
330       }
331     }
332   }
333
334   void loop()
335   {
336     //receive messages
337     receive_inputs();
338     set_gradient();
339     // within stopping threshold
340     if (mydata->my_reading < 70){
341       spinup_motors();
342       set_motors(0, 0);
343       set_move_type(STOP);
344       set_bot_state(LISTEN);
345       set_color(RGB(0,0,1));
346       mydata->own_gradient = 0;
347     }
348     // kilobot is NOT an adhesion site
349     else if (mydata->own_gradient != 0){
350       bool highest = true;
351       bool only_mover = true;
352       mydata->lock = false;
353       // check to see if any neighbors are moving and if have
354       // highest gradient
355       for (int i = 0; i < mydata->N_Neighbors; i++){
356         if (mydata->neighbors[i].n_bot_state == MOVE){
357           only_mover = false;
358         }
```

```
359         if (mydata->neighbors[i].grad > mydata->own_gradient){
360           highest = false;
361         }
362       }
363       // Follow the edge if both no neighbors are moving and have
364       // highest gradient
365       if(highest && only_mover)
366       {
367         set_color(RGB(1,0,0));
368         follow_edge();
369       }
370       // Stay stationary
371       else{
372         set_color(RGB(0,1,0));
373         set_bot_state(LISTEN);
374         spinup_motors();
375         set_motors(0, 0);
376         set_move_type(STOP);
377       }
378     }
379     // If the kilobot is all alone
380     else if (mydata->N_Neighbors == 0){
381       // it appears that 14seconds is around how long it
382       // takes for the kilobot to turn 180 degrees
383       if (!mydata->lost){
384           mydata->momentum_check = kilo_ticks;
385       }
386       mydata->lost = true;
387       if (kilo_ticks - mydata->momentum_check > 515){
388         spinup_motors();
389         set_motors(kilo_turn_left, kilo_turn_right);
390         set_move_type(STRAIGHT);
391       }
392       else
393         set_color(RGB(1,1,1));
394     }
395     // Otherwise, the kilobot is an adhesion site
396     else {
397       // If far enough away from other kilobots stop moving
398       if (find_nearest_N_dist() > 55){
399         spinup_motors();
400         set_motors(0, 0);
401         set_move_type(STOP);
402         mydata->lock = true;
403       }
```

```
404       // Otherwise move forward
405       else if (!mydata->lock && mydata->N_Neighbors > 0){
406         spinup_motors();
407         set_motors(kilo_turn_left, kilo_turn_right);
408         set_move_type(STRAIGHT);
409       }
410       set_color(RGB(0,0,1));
411       set_bot_state(LISTEN);
412     }
413     setup_message(); // prepare the next message
414   }
415   /*
416    * Beyond this point is all helper code for the simulator
417    */
418   extern char* (*callback_botinfo) (void);
419   char *botinfo(void);
420   int16_t callback_lighting(double, double);
421   int main(void)
422   {
423     kilo_init();
424   #ifdef DEBUG
425     // setup debugging, i.e. printf to serial port, in real Kilobot
426     debug_init();
427   #endif
428     SET_CALLBACK(botinfo, botinfo);
429     SET_CALLBACK(reset, setup);
430     SET_CALLBACK(lighting, callback_lighting);
431     RB_init();                          // initialize ring buffer
432     kilo_message_rx = rxbuffer_push;
433     kilo_message_tx = message_tx;       // register our transmission function
434     kilo_start(setup, loop);
435     return 0;
436   }
437   #ifdef SIMULATOR
438   // provide a text string for the status bar, about this bot
439   static char botinfo_buffer[10000];
440   //print out information about the bot
441   char *botinfo(void)
442   {
443     int n;
444     char *p = botinfo_buffer;
445     n = sprintf (p, "ID: %d ", kilo_uid);
446     p += n;
447     n = sprintf (p, "my_reading: %i ", mydata->my_reading);
448     p += n;
```

```
449     n = sprintf (p, "light: %d ", get_ambientlight());
450     p += n;
451     n = sprintf (p, "transmitted reading: %i ", (mydata->transmit_msg.data[4] <
452     p += n;
453     n = sprintf (p, " transmitted ttl: %i ", (mydata->transmit_msg.data[7]));
454     p += n;
455     n = sprintf (p, "gradient: %i ", mydata->own_gradient);
456     p += n;
457
458     return botinfo_buffer;
459 }
460
461 /*
462  * Uses euclidean distance to setup a lighting callback function
463  * Lower values returned values correspond to higher light
464  * Light readings degrade linearly
465  */
466 int16_t callback_lighting(double x, double y){
467     double light_y = 0;
468     double light_x = -150;
469     double dist_x = pow(light_x + x, 2);
470     double dist_y = pow(light_y + y, 2);
471     double dist_c = sqrt(dist_x + dist_y);
472     return (int16_t)dist_c;
473 }
474 #endif
```