THE PENNSYLVANIA STATE UNIVERSITY
DEPARTMENT OF ENGINEERING SCIENCE AND MECHANICS

A Computational Framework for Neural Network-Based Bioinspired Swarm Coordination

William Laplante
SPRING 2025

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Engineering Science
with honors in Engineering Science

Reviewed and approved by the following:

Dr. Christian Peco
Associate Professor of Engineering Science and Mechanics
Thesis Supervisor

Dr. Joseph Cusumano
Professor of Engineering Science and Mechanics
Academic Adviser

Dr. Lucas Passmore
Professor of Engineering Science and Mechanics
Honors Adviser

Dr. Vincent Meunier
P. B. Breneman Department Head Chair
Professor, Department of Engineering Science and Mechanics

The thesis of William S. Laplante was reviewed and approved by the following:

Christian Peco
Associate Professor of Engineering Science and Mechanics
Thesis Adviser

_____ _____
Signature Date

Joseph Cusumano
Professor of Engineering Science and Mechanics
Academic Adviser

_____ _____
Signature Date

Vincent Meunier
P. B. Breneman Department Head Chair
Professor, Department of Engineering Science and Mechanics

_____ _____
Signature Date

Abstract

Technical

Many swarm optimization algorithms are designed to imitate the behaviors of biological swarms, such as Ant Colony Optimization designed to imitate the foraging behaviors of ants and their pheromone trails. Though these behaviors can often be 'hard coded' such that they are easily programmable by humans, some of the behaviors can be harder to isolate and abstract, such as the path-finding and various memories exhibited by slime molds. Attempting to capture these behaviors using a neural network is one option, but testing and refining can become major investments in time if attempting to use real-world swarm robots. Two simulation frameworks were designed to allow more rapid testing, one based on Kilobots and their methods of sensing and actuating, and another designed to be fully modeler, such that any kind of swarm robot could be imitated. These frameworks underwent testing to ensure their ability to imitate well known swarm algorithm programs, both when hard coded and when run on a neural network. The frameworks performed adequately, producing the results expected in all benchmark tests, signaling their potential use in future swarm robotics research.

Non-Technical

Swarm robotics is the study of building and programming robots that are able to communicate and interact with each other in order to perform more complex tasks, even when there is no central robot or computer controlling them. The programs and algorithms used in swarm robotics are often inspired by biological swarms seen in nature, such as ants or flocks of birds. When attempting to design and study new algorithms, testing is often slowed down by the time and technical limitations imposed by physical robots. Two simulation frameworks were designed to allow faster testing of swarm robotic algorithms, one based on imitating a specific brand of robot, and the other built to be more generalized. These frameworks were tested using various benchmark simulations based on well known swarm algorithms to ensure their efficacy. In all tests, the simulations gave desirable results, confirming that these frameworks could be of use in future research.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Prof. Christian Peco first and foremost, for his guidance in writing and editing this thesis and for his flexibility in its scope. I would additionally like to thank three members of his research group in particular: Joe Sgarrella, who was my primary contact during the initial stages of research, Shishir Barai, who helped with ensuring the neural network code could run in the simulations developed, and Manik Kumar, who assisted with the development of the multiagent algorithm.

We acknowledge the support of the National Science Foundation grant 2339373. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Computations for this research were performed on the Pennsylvania State University's Institute for Computational and Data Sciences' Roar supercomputer.

Chapter 1

Introduction

1.1 Motivation

In nature, many organisms can communicate and cooperate among themselves, causing the group as a whole to begin displaying behaviors that would not be expected if observing the behavior of a single element of the group. These behaviors can be seen in the movements of bird flocks in flight [11], or the ability of ant colonies to find the quickest path to a food source [12]. The rules of these emergent behaviors are sometimes very difficult to formulate in a *ad-hoc manner*, but if understood, they can be key in advanced decentralized robotic swarms [8, 13]. Slime mold in particular has many behaviors of interest to swarm robotics, including pathway optimization [2], habituation [14], and periodic memory [15]. However, these dynamical responses are challenging to capture in mathematical models [16, 17], and call for a combination of computational mechanics, finite element modeling applied to biology [18, 19, 20, 21], multiagent simulation, super computing [22, 23, 24], and the use of tools such as machine learning to reproduce complex material behavior [25, 26, 27, 28]. In this work, we have focused on developing multiagent frameworks to aid in different parts of this multifaceted project.

1.2 Background

Physarum Polycephalum is a unicellular organism that is cultivated in many areas of research, including biology, computer science, engineering, and physics [29]. Part of this popularity is due to its ease of cultivation, requiring as little as agar and oats to grow a sample [29, 30, 31, 14]. Under normal conditions, it seeks out food by spreading pseudopods out in an area, using oscillations of fluid to cause outward pressure. When one of these pseudopods comes in contact with food, that pseudopod swells up, causing a similar decrease in size of other pseudopods [30]. This simple

behavior allows the shortest path between food sources to quickly become prioritized, allowing the slime mold to settle upon the optimal distance between any two points, despite not having a conventional brain.

Swarm robotics is a field focused on the coordination of multiple units as opposed to singular entities. More specifically, the focus is on the design of simpler robots that can communicate and collaborate with each other to perform tasks more complex than a single robot could perform alone. There are several benefits to swarm robotics compared to more traditional designs. Firstly, as a swarm, there is no single device or controller that can serve as a point of failure, any robot can be removed and replaced with any other. Additionally, the nature of the swarm means that it can be easily scaled up and down, with the effectiveness of a single robot being the same whether it is in a swarm of 20 or 2,000 [32].

1.3 Problem Statement

Swarm robots have many potential benefits over traditional singular robots in some scenarios, such as the ability to scale the swarm size up or down as needed, and the ability to replace or remove individual robots in the swarm without little to no change in the effectiveness of the swarm as a whole [32] . However, research is often limited by the challenges of testing swarms which require dozens or even hundreds of robots in order to produce meaningful results. A single Kilobot, a swarm robot designed for low cost, is still 14 dollars [33], meaning assembling large numbers of them can become prohibitively expensive. Additionally, continual maintenance and upkeep on a large number of robots can require a significant investment of time. Though software is available to simulate robotic swarms [34], lack of documentation can make it difficult to use, and if the software is not properly updated and maintained over time, it may become incompatible with more recent technology. The algorithm proposed here will allow rapid design and simulation of swarm robot algorithms, with precise control over the capabilities of the robot, the simulated environment, and the behavior of the robot. The simulated nature allows large swarms to be run with little to no cost. Additionally, the simulation design allows changes to the environment or the robot algorithms to be added or removed quickly and easily.

Chapter 2

Literature Review

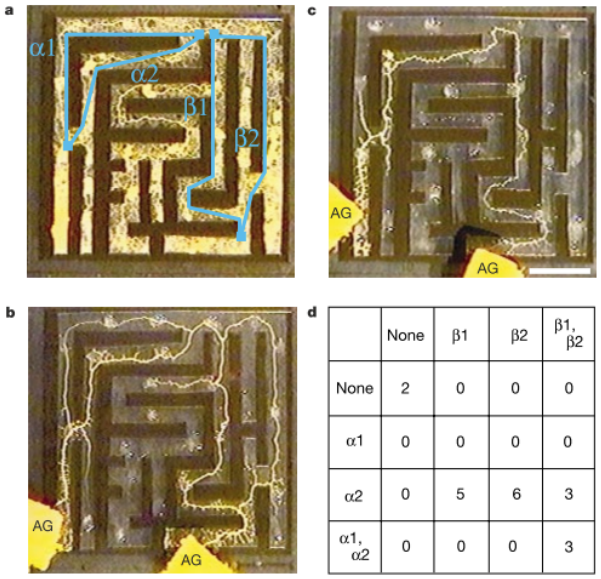
This literature review serves to provide an understanding of the current state of research into biologically inspired swarm control algorithms. To do so, it is split into four separate sections. The first gives an overview of *Physarum polycephalum (P. polycephalum)* and its behaviors that are of interest to researchers. Section 2.2 focuses on the field of swarm robotics, and what design challenges it currently faces, while Section 2.3 discusses some of the algorithms often used in swarm robotics, and the biological elements that inspire them. Finally, the last section will broadly look at the research that has gone into neural networks.

2.1 Slime Molds and Emergent Behavior

P. Polycephalum, despite not possessing neurons or any typical mind as we think of it, is capable of accomplishing tasks that are typically thought of as requiring more complex thought. Tasks like path-finding, optimizing the distance between points of interest, or remembering previous events such as where it was or what happened.

In one study, a sample of slime mold was placed inside a maze, while a food sample was placed at each exit from the maze. Within the maze itself were two separate forks, α and β, that allowed the slime to choose which direction to take. When allowed to grow over time, the slime mold first expanded over the entirety of the maze, before slowly pulling back all pseudopods which only led to dead ends. After about eight hours total, the slime mold simplified down so that it only had one or two pseudopods connecting it to the food sources, which followed the optimal or near optimal path. An example of this pattern of growth and shrinkage can be seen in Fig 2.1 [1].

Another study also investigated the path-finding abilities of Physarum Polycephalum, but in a more open-ended environment. A sample of slime mold was placed in an environment with 36 food sources, with the sources serving as representations of key geographical regions in Tokyo. Different sections of the environment were also subject to varying degrees of light, in order to



	None	β1	β2	β1, β2
None	2	0	0	0
α1	0	0	0	0
α2	0	5	6	3
α1, α2	0	0	0	3

Figure 2.1: Example of Physarum Polycephalum path-finding through a maze [1]. (a) the slime mold expands through the maze, while the two choices α and β are shown in blue. (b) all pathways that do not connect the two food sources begin to fall away. (c) the slime mold now contains only a single pathway connecting the two ends of the maze. (d) a representation of how many samples chose each path. None means the slime mold failed to grow.

represent features such as mountains and lakes that can obstruct construction. This light discouraged the slime mold from growing within or towards those areas. The growth and concentration of slime mold pseudopods can be seen in Fig. 2.2. After allowing the slime mold to optimize the connections between food sources, the resulting network was then compared to the real world Tokyo rail network [2]. Though there was some variation among samples, many of them closely resembled the actual rail network, showcasing a convergence towards a single optimal network.

With mathematical models in place, it is possible to program a device to behave like a slime mold, or as the case may be, several devices working together.

2.2 Swarm Robotics

Swarm robotics is the study of how individual entities are able to interact with one another in such a way that the collective exhibits unexpected or more complex behaviors. [32]. This emergent behavior can include coordinated motion, more efficient mapping of an area, optimized path-finding, and other behaviors, and has potential applications in fields such as search and rescue, shipping and warehouse management, and even in agriculture and food production [35] . However, the field is still relatively young, and many difficult questions remain unanswered regarding how to program the base behaviors of swarm robots. Two of these challenges often faced in the design of new swarm robotic algorithms are coordination and collective memory [36]. Coordination refers to the ability for individual agents of the swarm to communicate information to each other, whether directly or indirectly. Collective memory refers to the memory of the swarm as a whole, not just individual robots, to remember the stimuli it experienced and elements of its environment such as

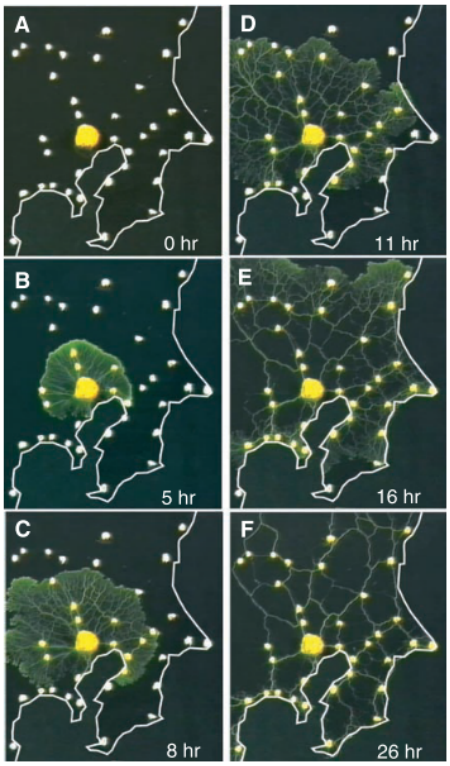


Figure 2.2: Growth of Physarum Polycephalum in a representation of Tokyo over 26 hours [2].

obstacles or food sources even if the robots do not all have those values saved in their memory.

Another challenge swarm robots face is the ability to determine their position in relation to the rest of the swarm, even when their sensors are unable to detect the position of every other robot. This can be a key challenge in ensuring that the swarm as a whole moves where to where it needs to be. One method of solving this issue involves using 'anchors', members of the swarm that have their coordinates defined at the start of the program, such as defining a single robot's position as the origin. From the anchor robot's position, other robots can then use their distance from the origin or other anchors to calculate their coordinates [37, 38]. A similar challenge arises from ensuring that all robots in a swarm can move collectively, instead of starting and stopping at different times, potentially causing them to lose cohesion. One possible solution to this is through the use of wave algorithms, where one robot initiates a task, transmitting an outward message to its neighbors instructing them to perform that task as well. These neighbors tell their neighbors until the entire swarm is performing the same task. Once the task is completed, a robot waits for all of its outward neighbors to complete their tasks before sending a message inwardly communicating its completion. Once all robots have transmitted their message of completion, the initiating robot transmits the command to start the next task, and the cycle continues [39].

Another key challenge in swarm robotics is the expense. Attempting to perform an experiment requiring dozens or hundreds of individual robots can quickly become costly, requiring a low-cost option in order for experimentation to remain viable at larger scales. Kilobots are designed with this in mind [33], with a single kilobot costing only $14 in 2014. Each Kilobot is approximately three centimeters in diameter, and is equipped with infrared sensors that allow the Kilobot to transmit and receive messages from other kilobots up to approximately six centimeters away.

Additionally, they are equipped with LEDs to help display information, while a pair of vibrating motors allowing a maximum speed of around one centimeter per second. Instead of wheels, three metal prongs are used to support the robot, with the vibration of the motors allowing the Kilobot to move and rotate.

Though the low cost makes Kilobots an enticing option, these robots are not without their drawbacks. Their form of locomotion causes a noticeable degree of variation in where a robot may move, and the very small range of communication between Kilobots can make larger or more complex forms of intercommunication difficult. Most importantly, many of the resources designed to aid in using Kilobots are outdated or neglected, causing difficulty in using them with more modern systems or software.

Coppelia, formerly known as V-REP, is a simulation software designed explicitly for use in robotics. Among the various tools available, a Kilobot simulation is included as part of the base software, allowing fast and efficient simulations of programs and scenarios that would take significant time and resources to prepare in the real world [34]. Other robotic simulation software exist as well, just as NetLogo, which is "a multi-agent programming language and modeling environment [40]."

2.3 Swarm Algorithms

Often when designing new swarm robotic algorithms, nature is used as a source of inspiration for how the swarm interacts with itself and the environment. These sources of inspiration can be as varied as fireflies, bats, bees[41], flocks of birds [11] , ant colonies [42], and slime molds [17]. Such algorithms use variations of how these animals or other lifeforms communicate with each other and navigate their environment. They serve as models for how machines could attempt to solve similar tasks.

Boids, a swarm robotics algorithm meant to imitate the movements of a flock of birds in flight, was first developed in 1987 [11]. In its simplest form, the algorithm uses a collection of individual agents each with their own velocity and a radius around them where they can sense other agents, as can be seen in Fig. 4.10. When an agent senses one or more other agents within its sensing range, it uses three rules to calculate a change in its own internal velocity. These rules are shown in Fig. 2.4. The first rule, avoidance, causes agents to move away from other agents, preventing them from clumping together. The second rule, cohesion, causes agents to change its velocity to move towards the average position of all other agents it detects, effectively causing it to be drawn to the center of whatever swarm it is in. The third rule, alignment, causes agents to change their velocity to match the average velocity of all neighbors, meaning that if given time, a swarm of agents will begin moving together in a single direction. These three rules are simple to implement and can be modified to work alongside other rules and limitations.

In one study by Martin et al [5], the boid algorithm was implemented with sliders allowing the user to change the range at which the various rules were implemented, with an example seen in Fig. 2.5. This could allow scenarios such as separation only happening to agents very close together, while alignment and cohesion can happen at ranges twice as far apart in comparison. The simple nature of the three rules also means that additional rules can be added to further refine the simulation. In Alaliyat et al [4], two possible rules are described. The first, obstacle avoidance, adds obstacles to the map that agents avoid in a manner similar to other agents. The second, the

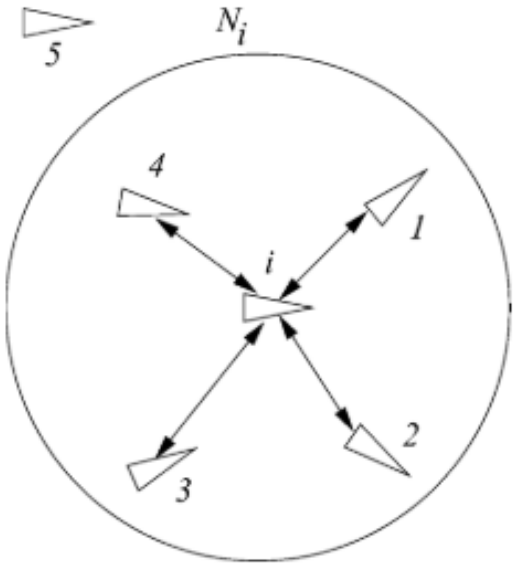


Figure 2.3: An example of the detection radius of a single boid. The central agent is able to detect agents 1 through 4, but not agent 5 [3].

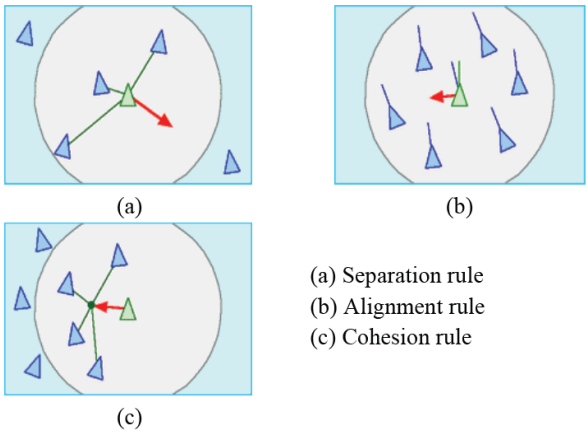


Figure 2.4: The three classic rules used by boids in navigation. Separation (a) causes an agent to move away from its neighboring agents Alignment (b) causes an agent to change its velocity to match its neighbors. Cohesion (c) causes an agent to move towards the average position of all neighbors [4].

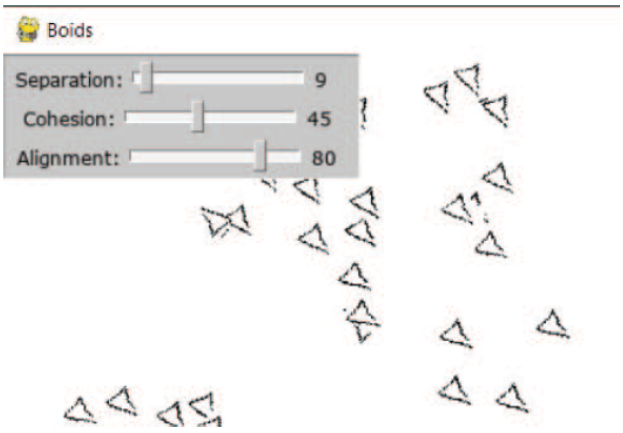


Figure 2.5: An example of a swarm simulation using the boid algorithm [5] Here, the three rules can have their radius of effect changed using a slider [5].

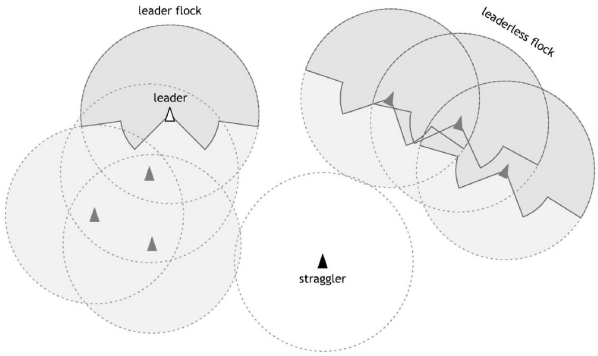


Figure 2.6: An example of a swarm simulation using a modified boid algorithm [6] Here, the agents have their vision limited to their front and sides, while also having the leader rule added alongside the three base rules.

leader rule, causes one agent in a swarm to be selected as the leader of the swarm, causing all other agents to follow them. The rules are not the only things that can be changed. In [6], instead of agents perceiving all other agents in a perfect circle around them, they have a more unique detection range, as can be seen in 2.6. In this case, agents are only able to perceive in a half circle directly in front of them and in small wedges on either side of them. Combining this with the leader rule, this means agents will only care about agents in front of them and to their side, and won't react to agents behind them.

Another well-known swarm robotics algorithm is the Ant Colony Optimization algorithm (ACO) meant to imitate the foraging behavior of a swarm of ants, including their ability to drop pheromones along the trail they take to help lead other ants to the food source they find[42]. In its most abstract state, this involves the movement of agents along the nodes and edges of graphs such as those seen in 2.7. Each node represents a point on the graph, and each edge represents a connection between two nodes, including the distance between them. A common challenge using graphs such as these is the Traveling Salesmen Problem (TSP), where the goal is to find the shortest path that visits

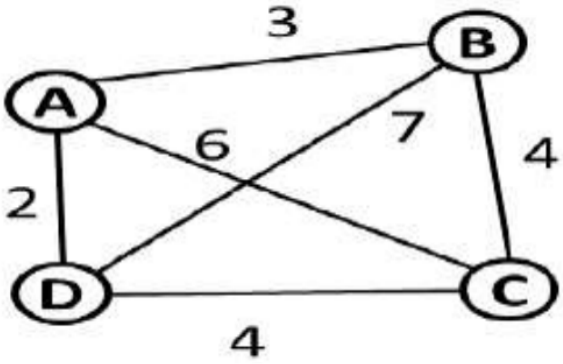


Figure 2.7: An example of a graph used in a Traveling Salesman Problem. A through D are nodes, while the length of each of the edges are labeled.

each node once and ends at the same node where it began [43]. When using an ACO algorithm to solve a TSP, a group of agents are placed onto the graph and allowed to travel. Each agent travels through the graph, randomly selecting which edge to travel through at each node, and placing pheromones on whichever edge they choose. Over time, the strength of these pheromones fades in proportion to the length of the edge, meaning that shorter paths will have stronger pheromones attached to them. As agents continue to travel, the random selection of edges becomes biased towards choosing edges with a stronger pheromone value attached to them, meaning that agents will prioritize taking shorter paths. This also means that they will drop their pheromones on that same edge, further increasing the likelihood that future agents will select the same edge [42].

This same behavior can be generalized to work in an open 2D environment, where agents are free to travel in any direction, not just along the edges between nodes. Many different variations of this algorithm have been implemented [7, 44, 45, 13, 8, 12]. In such cases, the algorithm follows a different method. The agents all begin at the same starting point, or ”nest”, and proceed to randomly walk around the environment until they detect a pheromone trail or a food source. Once the agent touches a food source, it begins traveling back to the nest while depositing pheromones before repeating the process. As can be seen in Fig. 2.8, this often means that a roughly straight line of pheromones is built between the nest and a food source, provided there are no obstacles in the way.

Methods of implementing the pheromone sensing in real world robots has led to various theorized designs, including alcohol and chemical sensors, heat trails, an overhead projector emitting light onto the environment floor, and even the use of phosphorescent paint which lights up in the presence of ultraviolet light [7]. This last option, chosen by Mayet et. al, required the use of robots with light sensors and ultraviolet LEDs in order to detect and produce trails. The concept was testing using a digital simulation tool, as can be seen in Fig. 2.8, and physical robots were designed as well. However, the robots often struggled with avoiding collisions with each other, and the trails produced often faded fast enough that a single robot might not be able to follow its own trail back to the nest, requiring the use of a secondary sensor that was always able to detect the angle the robot could take to travel back to the nest.

In another paper, Talamali et. al [8] attempted to implement a version of the Ant Colony Algorithm using Kilobots. Due to the limited abilities of the kilobots, an overhead camera and infrared

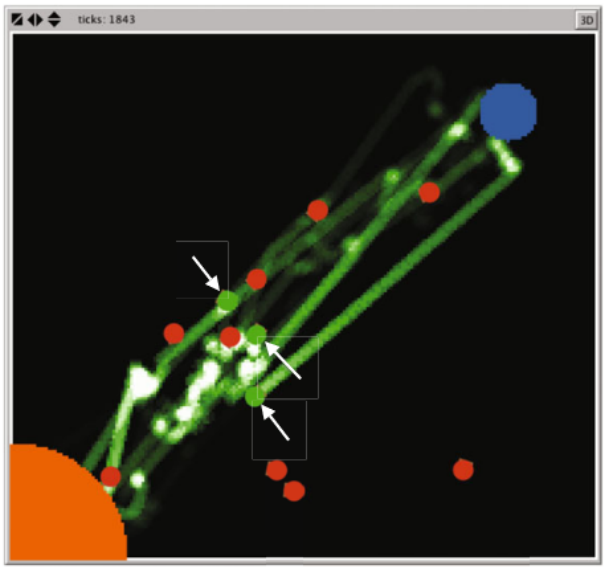


Figure 2.8: An ACO simulation using agents in an open 2D environment. The orange dot is the nest, the blue dot is the food source, and the gree trails represent pheromones. Agents are represented as dots, showing that they are either foraging (red) or returning to the nest (green) [7].

overhead control board were used. The camera was able to detect the positions of all Kilobots, feeding this information to a computer which in turn ran a simulated version of the environment including food sources and pheromone trails that was passed on to the overhead control board. This control board then projected infrared light down on to the environment that the Kilobots could then detect and respond to. In this way, the Kilobots are able to imitate behaviors such as pheromone dropping and sensing despite not having the built in capabilities. Due to the limited nature of a single Kilobots memory, many of its sensors had to be programmed to use as little memory as possible. For example, the kilobots pheromone sensing, as seen in Fig. 2.9, is limited to four 45 degree regions in front of it. In each of these regions, the only thing the kilobot can sense is whether or not pheromone is present there, with no detection of its strength. This means the entirety of a single Kilobot's pheromone sensing can be encoded in only four bits of information.

2.4 Neural Networks for Biomimicry

Neural networks are a type of model often used in machine learning. One example is provided in Fig. 2.10. At its most basic, a neural network is composed of various nodes, often represented as circles, which take input values and export output values. These inputs and outputs are often represented in the form of lines connecting different nodes, as seen in Fig. 2.11. These connections are often given weights as coefficients, which increase or decrease the value of outputs in relation to the inputs. After first being created, a neural network goes through a period of training, whereby data with known inputs and outputs is given to the neural network. As more data is fed, it adjusts the values of weights between different nodes so that its output more closely matches the output of the training data. Once enough data has been provided, the neural network is able to predict the output of novel data that is given to it, with the accuracy of its answers dependent on the complexity

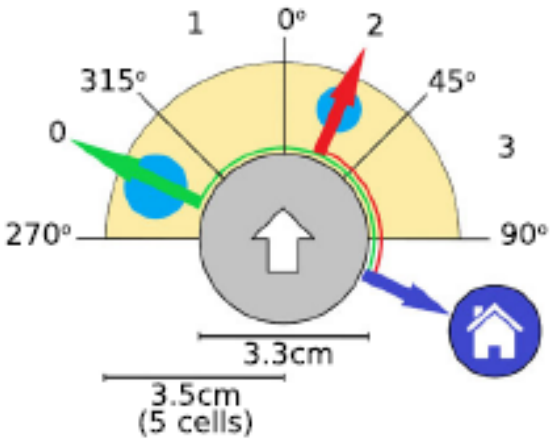


Figure 2.9: The pheromone sensing used for a kilobot based ACO [8]. The kilobot can detect whether or not there is pheromone in each of the four regions in front of it. If able to detect pheromone in multiple regions at once, it prioritizes the pheromone which angles it further away from the nest.

of the network and the amount of training data that it was given [9]. This design enables the system to learn the rules that connect input and output in an abstract way, as opposed to a human coding the connections into the program by hand. This allows the system to detect correlations that might otherwise go unnoticed and make use of them. The correlations can be as complex as the data fed in the system, from simple benchmarks to more realistic biological network and microstructural simulations [46, 47, 48]. Many of these simulations use phase-field methodologies [19, 49, 50] and large-scale supercomputer methodologies [51, 52, 53].

One of the most common styles of neural networks is feed forward neural networks, where there is a single layer of input nodes, a single layer of output nodes, and one or more hidden layers of nodes between them with all node outputs being pushed to the next layer only [54]. The number of hidden layers can range anywhere from one or two to hundreds of layers. This complexity allows the neural network to solve even abstract problems, such as disease detection or image recognition [9, 54, 55].

An alternative method of designing a neural network is as a graph neural network. In this case, each layer of the network is composed of the same nodes and connections, with each progressive layer allowing a node to pull information from further and further away from the original node [56]. This graph model can be extremely useful, since it allows the modeling of real-world connections between objects in the design of the network itself. The outputs in this case can be predictions about the graph on three different levels: node level, such as guessing the role of each individual node in a network, edge level, such as predicting what the connections are between different nodes, or graph level, which focuses on predicting properties of the graph as a whole [56, 57].

Neural networks are of interest to several researchers in the field of swarm robotics, as they allow the modeling and imitation of complex behaviors seen in nature even without full human understanding of their mechanisms. One example of this comes from Song et al [10]. In this study, another attempt at simulating foraging behaviors with ACO is performed, with a few key differences. First, each food source requires at least two foraging agents working together in order

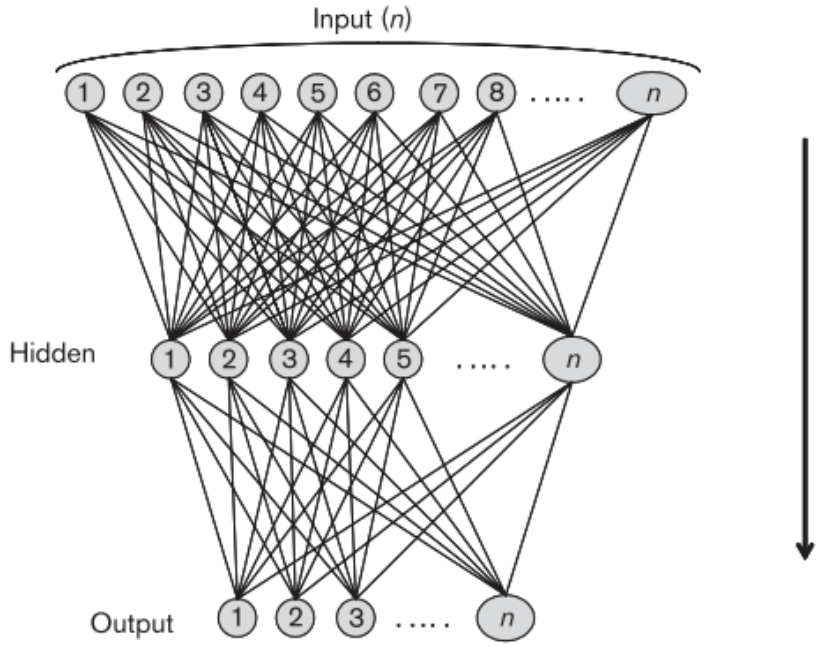


Figure 2.10: Layout of an example feed forward neural network [9].

to be carried back to the nest. Secondly, the behavior of pheromones is handled using a neural network. The neural network itself is only a single layer, consisting of a square grid of neurons. Each neuron corresponds to a two-dimensional point in the simulation, and is connected to the surrounding neurons as seen in Fig. 2.12

The foraging agents in this simulation are able to place repellent or attractive pheromones, which correspond to positive and negative values being applied to whichever neuron corresponds to the agents position. The repellent pheromone allows agents to avoid previously explored places and obstacles, while the attractive pheromone allows agents to lead the way to food sources they find. The neural network allows the modeling of dispersion and evaporation of the pheromones overtime. Dispersion is modeled by causing the value of a neuron to increase or decrease to match its neighbors, while evaporation is modeled by the pheromone value gradually returning to zero if not disturbed [10]. Over time, this causes a high pheromone value located at a single neuron to become a low pheromone value spread out over several nodes. This method of representing environmental details in simulation can be easily modified to fit similar scenarios, such as representing the extracellular slime left by P. Polycephalum.

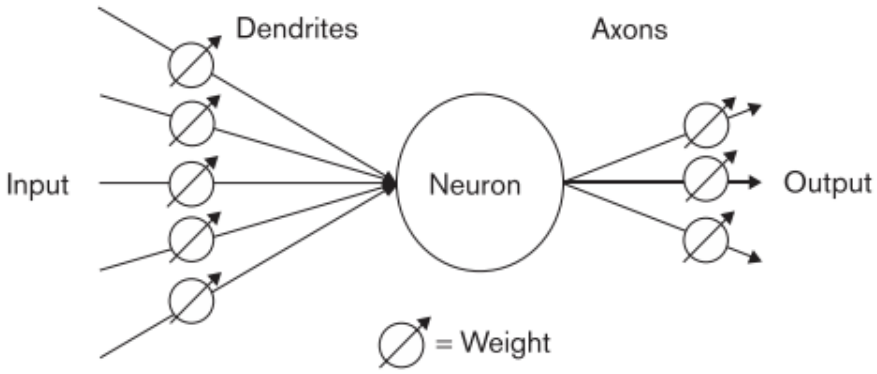


Figure 2.11: Parts of a single neural network element [9].

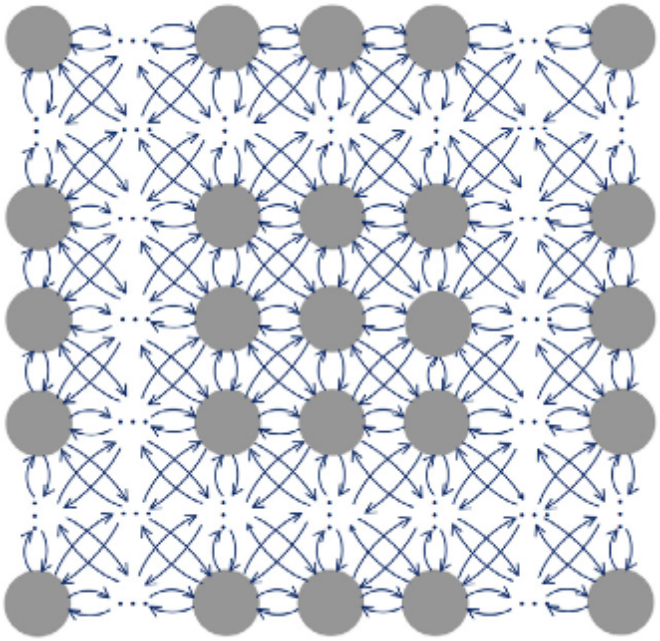


Figure 2.12: example of the neural network used by Song et al [10]

Chapter 3

Methodology

3.1 Simulation frameworks

For this paper, two primary computer frameworks were designed in order to imitate the behavior of swarm robots and their surrounding environment. The KBS framework, which was designed first, was intended to allow imitation of Kilobots only. Though this was sufficient for initial experiments, it soon became clear that Kilobots had fundamental design limitations which would cause difficulties if use of them in future tests continued. The multiagent framework was designed later with many of the same principles as the KBS framework, but with more generalized concepts which allows the imitation of swarm robots beyond just Kilobots.

3.1.1 KBS framework

In this section, we talk about the KBS framework developed, including it's general design and application. The KBS framework is built in Python, and utilizes two classes and one script to run the simulation environment. All together, this allows the framework to imitate the capabilities of real world Kilobots, including their communication with each other and their detection of the surrounding environment.

The first class, KBS, is used to imitate the Kilobot robots themselves. When a KBS object is initialized, its position, angle, maximum speed, maximum speed, size, detection radius, and status are all defined, along with any additional variables required for the simulation being performed. As seen in Fig. 3.1, the Kilobots follow a general pattern of behavior. They first compile a list of all other KBS objects and nodes within their radius of detection. Then, depending on what their current status is, they run through the corresponding function. This function can result in a change of position, angle speed, any other simulation-specific variables, or even change the status of the KBS object, resulting in a change of behavior.

The node class serves to approximate the real-world environment that Kilobots can detect. At its most basic, the node consists of only a set of X-Y coordinates. However, the user can call additional variables to the node class, or functions that update those variables at each time step. Such variables can include attractants or repellents, which KBS agents can be told to move towards or away from, or a pheromone variable which is increased by a KBS agent moving over it but naturally decays in strength over time. In the simulation, a collection of nodes are laid out in an even grid pattern to represent varying points of the environment.

The simulation begins by initializing the environment, placing nodes spaced ten units apart across the specified area, followed by the simulation script modifying the internal variables of any regions of nodes required, such as declaring a 20 by 20 unit square area to be filled with repellent. Following this, all KBS agents are declared, including all initialized variables. Once both the environment and the bots are initialized, the simulations continually loops through the behavior of all KBS agents until any specified end conditions are met. Additionally, the Pygame library is used to visualize the simulation environment [**?**].

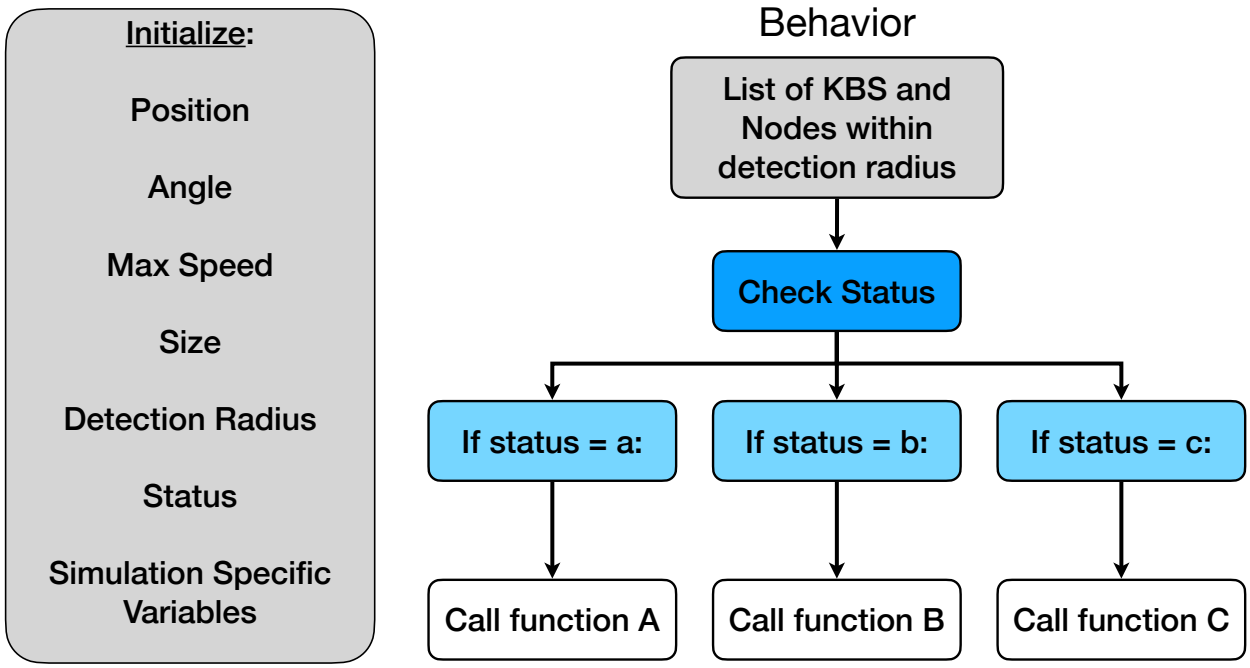


Figure 3.1: Flowchart representation of the script utilized for the physical Kilobots.

3.1.2 Multiagent framework

In this section, we explain the components of the Multiagent framework, also known as the Multi-Agent System (MAS). Unlike the KBS framework, which is designed only to imitate the behavior of Kilobots, the MAS allows any kind of swarm robot to be imitated. The MAS makes use of four base classes: Agent, Environment, Grid, and PhysicalTwinBase. These four classes do

very little on their own, but can be used to generate subclasses to represent more precise aspects of the simulation.

The Agent class is used to represent any entity that can sense its surrounding environment and perform actions in response to what it perceives. This connection between agent and environment is represented in Fig. 3.2

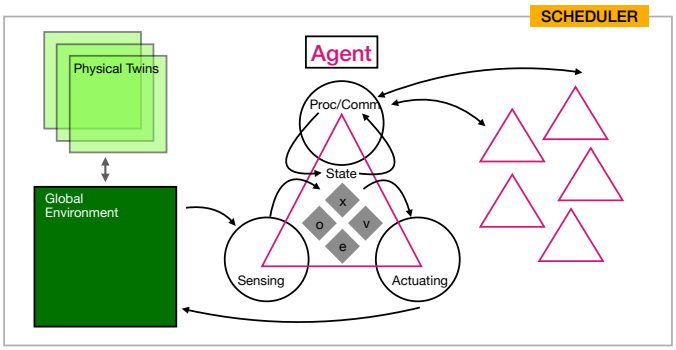


Figure 3.2: A representation of how the elements of the Multiagent Algorithm interact with each other

Each agent has four internal parameters and three functions that it uses. The parameters used are Attributes (X), Variables (V), Objectives (O), and Environment (E). Attributes represent long lasting values or values that the agent would not necessarily know, such as its size, range of vision, maximum speed, or it's global position in the simulation environment. Variables, while similar, are used to keep track of short term memory parameters, such as a list of surrounding neighbors, an internal timer, or the current status of the agent. Objectives is used to track, as the name suggests, the current objectives of the Agent, such as a target position it is aiming for. Environment is used to track the track the environment as the agent is able to perceive it, also known as its internal environment. This can be vary different from the global environment, such as having a reduced range or lower fidelity than the global environment. The coordinates recorded can also vary between the global and internal environment. While an agent may have been placed at the point [2,3] in the global environment, it might not know that and instead declare its starting point as [0,0] in its internal environment.

The functions that each agent has are Sensing, Processing, and Actuating. The sensing function allows the agents to gather information from the global environment and surrounding agents, storing it within its Environment parameter. The exact details of what environmental details are detected can depend on certain attributes. For example, the Agent may be designed so that it can only detect a certain variable in a cone in front of it, or simulates noise in a sensor by adding or subtracting a random amount from the values it measures. Processing allows the agent to update its internal parameters, including Variables and Objectives. Actuating is used to update external variables and includes updating Attributes such as position and angle.

The remaining three classes, PhysicalTwinBase, Grid, and Environment, are all closely entwined and build off of each other to simulate a real world environment. The PhysicalTwinBase attempts to imitate real-world processes or systems that a real-world swarm robot would be able to interact with. With this class, various subclasses can be generated, such as a field of force vectors, or scalar values to represent temperature across an environment. At the same time, the Grid class, similarly to the Node class in the KBS framework, is used to generate an array of points to represent regions of the simulation. This array of points does not need to be evenly spaced, and further points can be added to sections to allow higher fidelity where needed. These two classes are then passed onto Environment, which applies the continuous data of the physical twins to the discrete node points. These classes, including Agent, are all combined in the simulation script to create a model like what can be seen in Fig. 3.3.

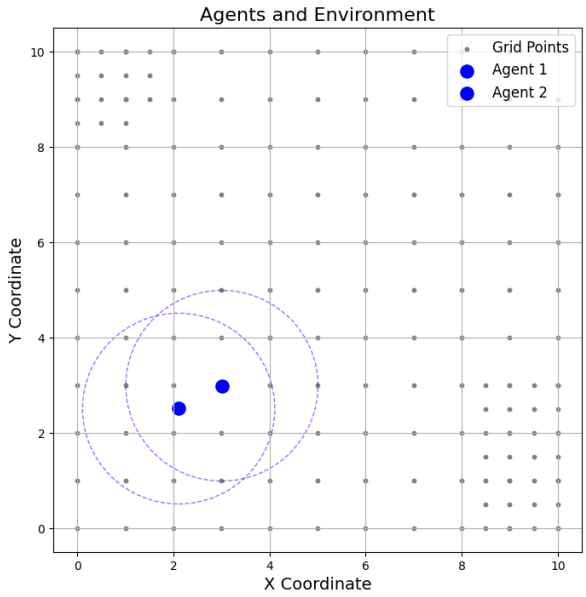


Figure 3.3: An example of the multiagent algorithm in action. The ability for the grid to be refined in regions is shown in the upper-left and bottom-right corners.

3.2 Neural Network Training

To generate the Neural Networks used in the 1D and 2D KBS experiments, a neural network encoder built and trained using Tensorflow was used. These neural networks use the positions of the neighboring agents as inputs, while the output values represent the target position the agent would then move toward. For the 1D case, the agent has two neighbors, each with only a single scalar value to represent position, while the 2D case had agents with either three or four neighbors, depending on its position in the matrix, with each neighbor having an X and Y value. In the 1D case, the neural network is relatively straightforward, as see in Fig. 3.4, while the 2D case requires a more complex arrangement of layers to account for the more complex data.

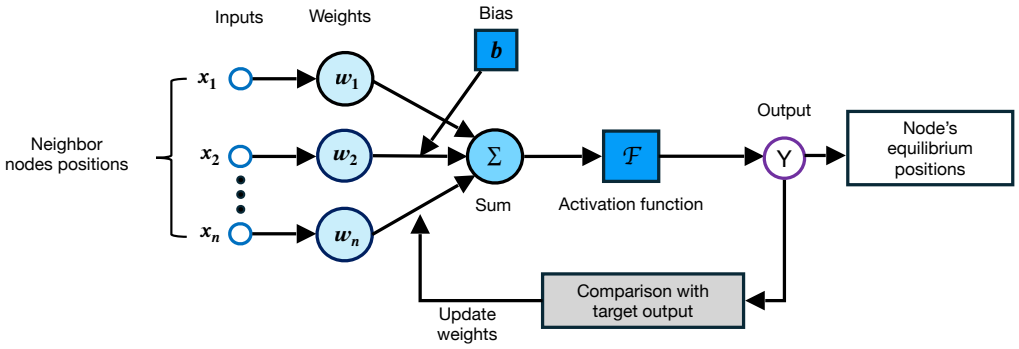


Figure 3.4: Schematic of the Neural Network model used in the 1D KBS simulation.

In order to train the neural network, it is given inputs matching those found in target data of the same scenario. The data used for this training was generated using a bio-lattice framework. The output value is then compared to the result recorded by the training data, with the weights and biases of the neural network then updated based on the errors recorded. This process is then repeated until the error between the neural network's output and the training data reaches a required minimum. This optimized neural network is then saved as a Keras file, to be used alongside the KBS agents. When used as part of the simulation, any KBS agent using the NN first collects all relevant information. In both the 1D and 2D information, this takes the form of the internally calculated coordinates of its neighboring KBS agents, as opposed to its global position defined at the beginning, which is used for drawing on the display window. This data is formatted to match the input data shape required by the NN and then fed into the NN for a single iteration run. The resulting output is then used to continue simulating the KBS agent's material-based behavior.

3.3 KBS Trilateration

For the KBS 2D Matrix Experiment, the individual agents made use of a Neural Network that assumed every agent knew their global coordinates. Real-world Kilobots are only able to receive information from their local environment. To address this, we create detectable reference points that KBS agents are able to measure their distance from, and enable the agents to determine their positions through trilateration.

The process of trilateration requires the target position (in this case, the agent) to be able to detect the distance to three different points with known coordinates, such as in 3.5. This results in the following system of equations.

$$r_1^2 = (x - x_1)^2 + (y - y_1)^2, \tag{3.1}$$

$$r_2^2 = (x - x_2)^2 + (y - y_2)^2, \tag{3.2}$$

$$r_3^2 = (x - x_3)^2 + (y - y_3)^2. \tag{3.3}$$

Where x_1, y_1, x_2, y_2, x_3, and y_3 are the coordinates of the three known points, r_1, r_2, and r_3 are the distances from the known points to the agent, and x and y represent the unknown coordinates of the agent.

These three equations can be simplified into a simpler system of equations.

$$Ax + By = C, \tag{3.4}$$

$$Dx + Ey = F, \tag{3.5}$$

Where:

$$A = (x_2 - x_1), \tag{3.6}$$
$$B = (y_2 - y_1), \tag{3.7}$$
$$C = \frac{1}{2}(r_1^2 - r_2^2 - x_1^2 + x_2^2 - y_1^2 + y_2^2), \tag{3.8}$$
$$D = (x_3 - x_2), \tag{3.9}$$
$$E = (y_3 - y_2), \tag{3.10}$$
$$F = \frac{1}{2}(r_2^2 - r_3^2 - x_2^2 + x_3^2 - y_2^2 + y_3^2). \tag{3.11}$$

Since the only unknowns in this system are x and y, they can by solved with the following solution:

$$x = \frac{BF - CE}{BD - AE}, \tag{3.12}$$
$$y = \frac{CD - AF}{BD - AE}. \tag{3.13}$$

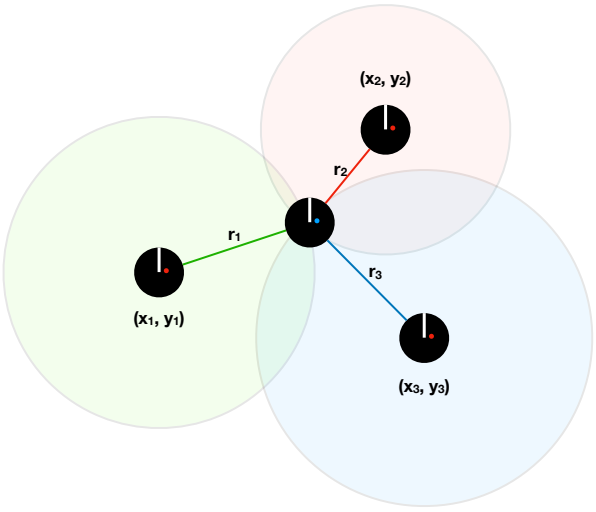


Figure 3.5: A Kilobot in position to perform trilateration. The values for x_1, y_1, x_2, y_2, x_3, and y_3 are known, and the values for r_1, r_2, and r_3 can be detected.

Chapter 4

Experimental Procedure and Results

In this section the various benchmark algorithms designed to evaluate the swarm robot simulation codes are explained, as well as the results of those benchmark algorithm tests.

4.1 KBS 1D Spring

The first benchmark designed was intended to imitate the behavior of a spring or similar elastic material as it is stretched in a one-dimensional line. A summary of the psuedocode is shown in Fig. 4.1. When the code is first initialized, three kinds of agent statuses are used: “base,” “end,” and “extension.” Base and end are placed on the far left and far right of the line, spaced a total of 190 units away from each other. At the same time, four additional agents labeled as extension are placed between the base and end. At the start of the simulation, the central agents are placed closer to the left end of the simulation, spaced 30 units apart from the base agent and each other, as can be seen in Fig. 4.2.

The base and end agents have their distance from the base agent and their ID number automatically declared before changing their statuses to ”base labeled” and ”end labeled.” Both of these agents remain static and take no other actions for the remainder of the simulation. For the next few time steps of the simulation, the extension agents determine which of them is directly to the right of the agent currently with the highest ID. The chosen agent then measures its distance to its neighbor with an ID and adds that amount to the neighbor’s position to find its own position. Lastly, the agent declares its ID to be one higher than its neighbor’s ID, and changes its status to ”labeled.” For example, the agent directly to the right of the base agent (which has a position and ID of zero) would measure a distance of 30 between itself and the base agent, and declare its position to be 30 and its ID to be 1 before declaring itself labeled.

Once a target is labeled, it begins the process of adjusting its position using information from the neural network. To do so, the agent collects the position of its left and right neighbors and

inputs these values into the neural network. In the case of the 1D simulation, the neural network was designed with two nodes on the input layer (representing the left and right agent positions), four nodes located within a single hidden layer, and a singular node in the output layer, meant to output the target position for the agent to move toward. In the training data, the target position was always the midpoint between the left and right agent, meaning the neural network was imitating a simple midpoint formula. The hyperparameters used for the training are summarized in Table 4.1

Table 4.1: Hyperparameters Tuning for the 1D Model

Model	Hidden Layers	Hidden Layer Size	Activation Function	Optimizer (learning rate)	Error Function	Percentage Error
1D	1	4	ReLU	Adam (lr=0.0005)	MSE	0.21%

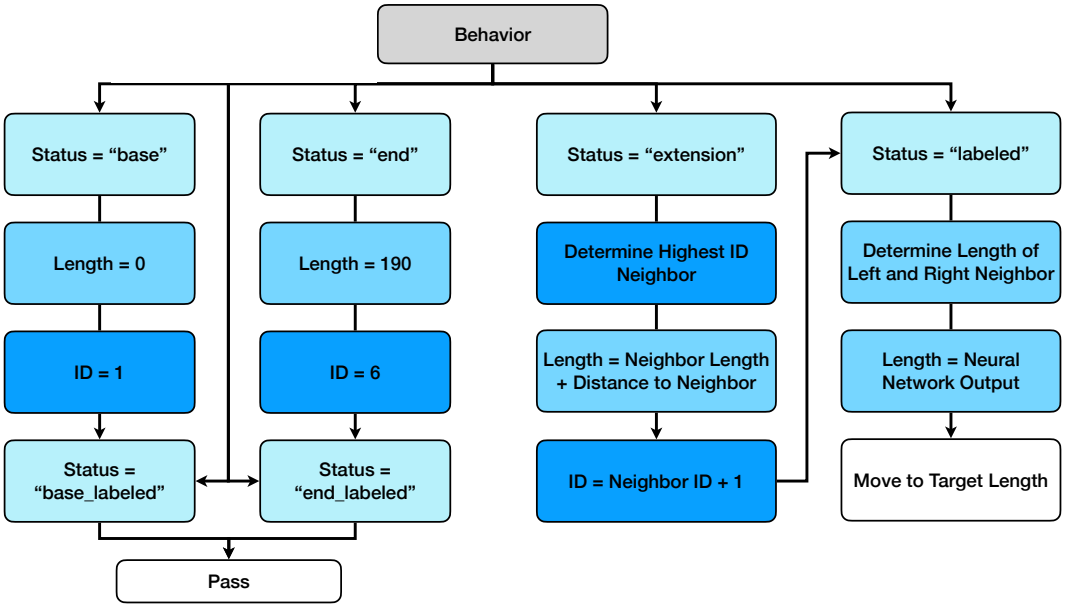


Figure 4.1: Flowchart representation of the behavior of KBS agents for the 1D simulation.

Once the neural network outputs a target position for the agent, it compares the target to its current length. If the target length is greater than its current length, it moves to the right, or to the left if it is lower.

In addition to the neural network simulation, a “hard coded” version was also designed where at each time step the central agents calculate the average position of their left and right neighbors and set that as their target position. The results of this simulation were visually compared to the neural network version to ensure the neural network was able to imitate this behavior.

When the simulation was run, the four central agents were able to determine their positions and ID numbers amongst themselves, and begin running the neural network. All agents began trending towards the right, with all agents reaching equilibrium when evenly spaced 38 units apart from each other. These results perfectly imitated the results of the hard coded simulation. These results

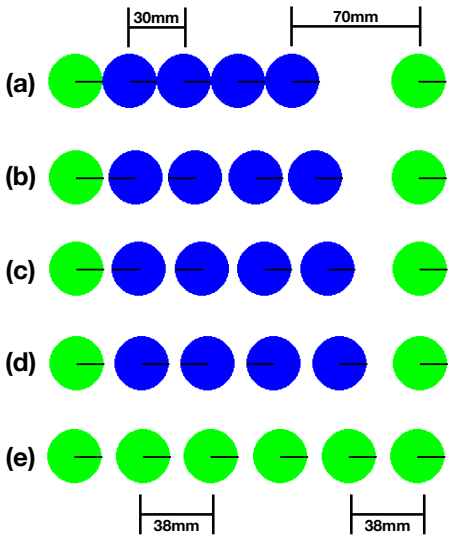


Figure 4.2: The results of the 1D spring simulation at various time steps. The central four agents begin spaced 30 units apart on the left-hand side (a). As the simulation continued, they moved to the left, maintaining equal distance between their left and right neighbors (b-d). By the end of the simulation, they have spread out to be equally spaced between the leftmost and rightmost agent (e).

show that the KBS is able to use neural networks to imitate rudimentary hard coded behavior. With the foundational goals of the framework now confirmed to be possible, the next step would be to test the framework's ability to use a more advanced neural network to imitate more complex scenarios.

4.2 KBS 2D Matrix

The 2D matrix simulation, similar to the 1D case, is meant to imitate the behavior of a material as it is stretched. In this case however, the material is two dimensional, meaning deformations perpendicular to the direction the material is stretched in must also be considered. Additionally, since agents must determine their position in 2D space, more complex methods are required, with six different agent statuses, as can be see in Fig. 4.3.

The initial agent placement, as can be seen in Fig. 4.4, places a four by three grid of agents in the center of the simulation, surrounding them with a five by four box of additional agents. This outer box of agents, all with the "reference" status, simply determine their subjective coordinates based on their objective position in the simulation. In this case, their coordinates are calculated to be 120 units less than their position in both the x and y direction, in order to ensure that the neural network receives the correct inputs.

Once the outer box of agents have determined their coordinates, the inner agents with the "seeker" status can begin to use trilateration to determine their own coordinates in relation. The corner inner agents, with three referenced agents around them, are able to determine their coordinates first, changing their status to "content." As more inner agents determine their coordinates,

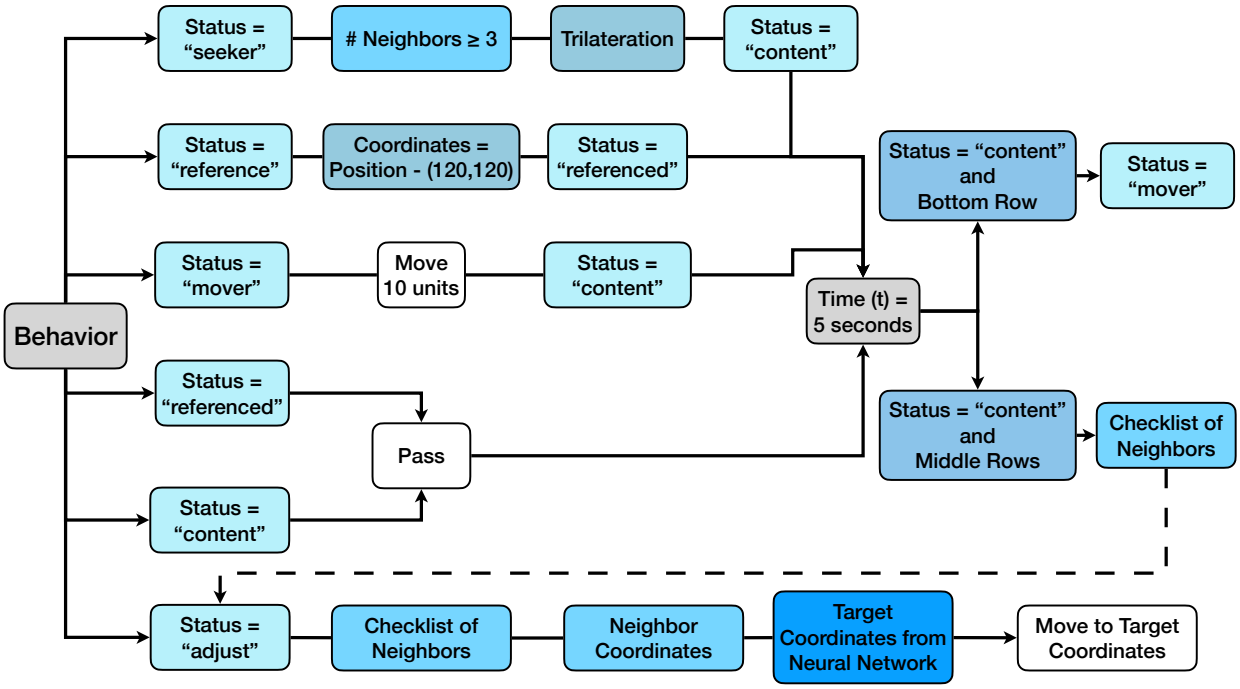


Figure 4.3: Flowchart representation of the behavior of KBS agents for the 2D simulation.

they can be used by other seeker agents for their own trilateration, until eventually all inner agents have determined their positions.

After a long enough period of time to ensure all agent coordinates are calculated, the bottom row of agents changes their status to "mover," and shifts ten units away from the rest of the 3x4 matrix. At the same time, the agents in the middle two rows change their status to "adjust," and begin using the neural network in order to determine their target coordinates.

When using the neural network, agents must first precisely determine which of their neighbors are above, below, to the left, and to the right of them, in order to ensure the X and Y coordinates of all neighbors are properly input into the neural network. If there is no neighbor, if the agent is on the left side of the box and has no left neighbor for example, a coordinate pair of [6000, 6000] is used to represent a "ghost node." Similar to the 1D simulation, the hyperparameters used as part of training can be found in 4.2

Table 4.2: Hyperparameters Tuning for the 2D Model

Model	Hidden Layers	Hidden Layer Size	Activation Function	Optimizer (learning rate)	Error Function	Percentage Error
2D	2	32	ReLU	Adam (lr=0.001)	MSE	0.13%

When the agent uses the neural network, it outputs a set of target coordinates for the agent to move towards. As all agents with the adjust status continue to move, they eventually achieve an equilibrium state where the target coordinates are so close to the agent's current coordinates that

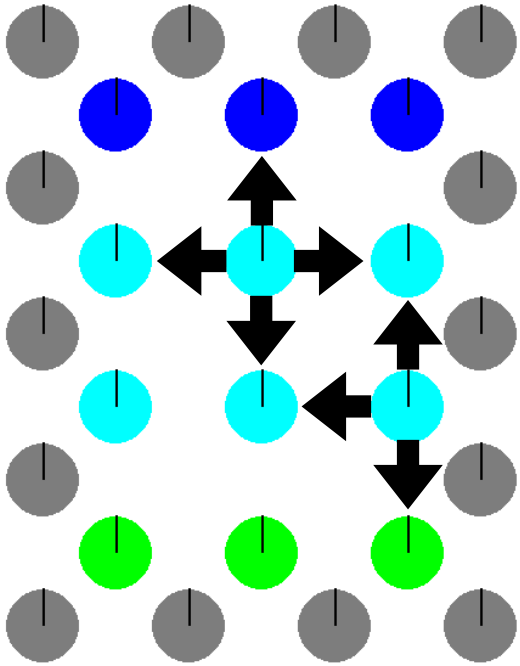


Figure 4.4: A diagram of the 2D matrix simulation near the beginning. The top row of blue agents all have the "mover" status. The middle two rows have the "adjust" status. The black arrows represent the neighboring agents used as part of the neural network: agents in the middle column have four neighbors, while agents on the sides have only three. The bottom row contains agents with the "content" status. The outer box of gray agents all have the "referenced" status.

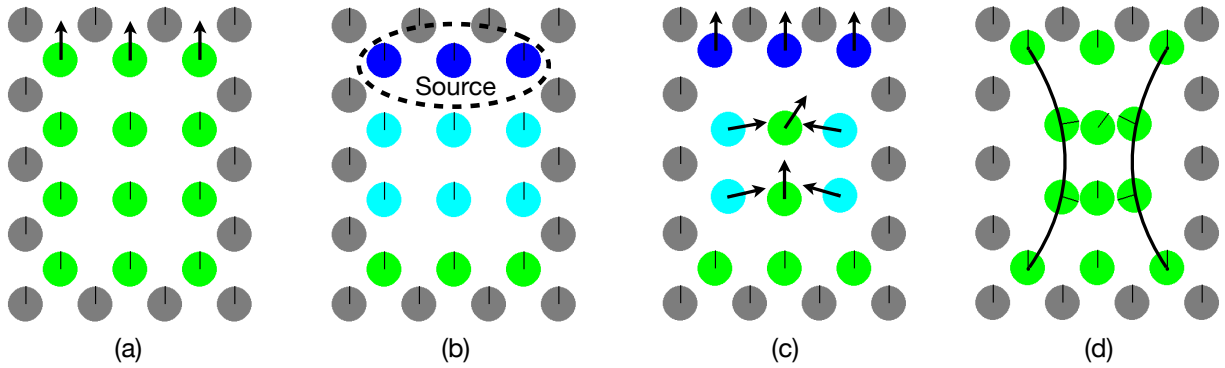


Figure 4.5: The 2D matrix simulation at various stages. The central row of agents begin by finding all of their coordinates, while the top row begins moving up (a). While the top row is still moving, the central rows of agents begin running the neural network (b-c). After finishing, the central rows have moved closer together horizontally, and spaced themselves apart vertically (d)

attempting to move towards them would instead cause them to overshoot.

When the simulation was run, the central two rows of agents performed as expected, with agents moving slightly towards the row of movers, and agents on the left and right side moving inwards in response to the deformation. As can be seen in Fig. 4.5, the movement of the agents was not perfectly in line with expectations, with one of the central agents moving slightly to the side, but the overall shape of the 2D matrix deformed into the desired shape. This shows that the KBS framework is able to successfully simulate the results of more complex neural networks.

4.3 KBS Ant Colony

Unlike the 1D Spring and 2D Matrix, the Ant Colony Optimization algorithm (ACO) is meant to imitate the behavior of a naturally occurring organism, in this case, the path-finding abilities of a swarm of ants.

Unlike the previous simulations which did not use the nodes at all, the nodes represent both food sources and pheromones in the environment. Food is represented by having a number of the nodes in the array marked as food, with each food source given a separate label, such as ”alpha” or ”beta” to distinguish them. Pheromones are stored as a numerical value, initially zero for all nodes in the graph. If the value for pheromones ever increases, it would begin to decay, thereby reducing its value by five percent at each time step before eventually reaching zero.

All agents in the simulation begin at the same origin point and with the same status: “seeking.” While seeking, agents begin by moving in one of 24 directions. Once this is finished, they pick a new direction to move in, repeating this process until (1) they reach the borders of the simulation, (2) they touch a food source, or (3) they have taken too many steps as determined by an internal counter. The direction they choose is not totally random. Instead, the agents weight each direction choice based on how close it is to the original direction they were moving towards, and if there

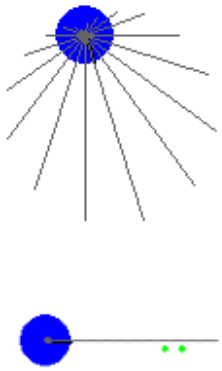


Figure 4.6: An instance of two agents used in the ACO simulation. The first agent, with no pheromones or food sources within its detection range, randomly chooses one of the directions designated by the gray lines. A larger line means a higher probability of that direction being chosen. The second agent, with a food source within its range of detection, is heavily weighted towards choosing the direction leading to the food source.

are any food sources or pheromones in that direction. The closer a direction is to the direction the agent moved the last time, the more likely it is to select it, with the opposite direction being impossible to select. This allows the agents to randomly walk while seeking, while also preventing them from getting stuck around the origin and not moving further away as they continue seeking. A visual example of these probabilities can be seen in Figure 4.6

If an agent touches the edge of the simulation or makes too many movements, the agent changes its status to "foodless." In this state, the agent begins backtracking along the same path it took until it eventually reaches its starting point, where it changes its status back to seeking and begins to randomly walk again. Additionally, it allows the maximum number of movements it can make before changing its status to foodless to increase, meaning that as the simulation continues, the agent will be able to explore further and further from the origin if it does not find a food source.

If an agent touches a food source, its status changes to "fooded." Similar to foodless, the agent begins backtracking to the origin point. While doing do, it also begins adding pheromones to the nodes around it. These pheromones encourage other agents which are seeking to follow the same path the original agent took, leading them to the same food source. When the agent begins seeking again and finds itself touching the same food source, there is a chance that instead of becoming fooded, the agent will instead continue seeking, allowing for the agent to create a trail containing multiple food sources simultaneously.

The simulation uses seven agents attempting to find a path between the origin and five food sources as seen in Figure 4.7. The first two food sources are very close to the origin, easily allowing the agents to create a path between the two. As the path becomes longer and the number of food sources along the path increases, the time it takes to build a path to the next food sources increases as well. Of particular note, the final two food sources are placed in such a way that agents in the simulation would often go to one or the other, but attempting to contact both was less likely. This

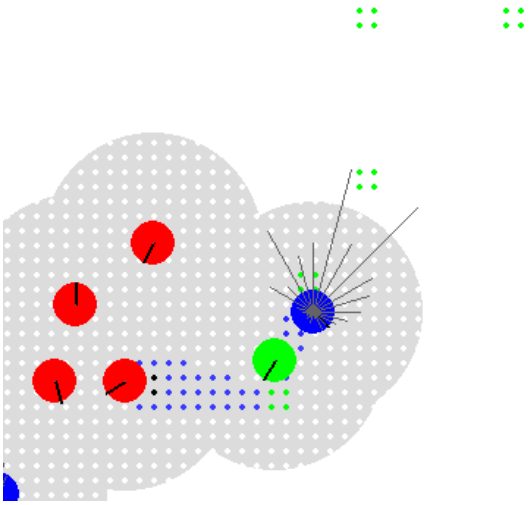


Figure 4.7: The early results of the ACO simulation. The red agents to the right are foodless, the green agent is fooded, and the two blue agents are still seeking. Green dots represent food sources, and blue dots represent pheromones.

leads to the final two food sources and the food source proceeding them to form first a forking pathway, as can be seen in Figure 4.8, and then a roughly triangular path between all three, instead of a single linear pathway, as can be seen in figure 4.9

The ACO simulation showed that the KBS framework was able to imitate more complicated swarm algorithms. However, the changes to the framework needed to do so begin to strain the credulity of the claim that it is meant to imitate real-world Kilobots. The agents used in this simulation are able to drop and sense pheromones in the environment, something real-world kilobots can only do with outside aid. More importantly, the agents do not directly interact with or collide with other agents, and can instantly change their angle after every step of their random walk—both of which are traits that do not align with physical reality, let alone physical Kilobots.

4.4 Multiagent Boids/Flocking

The final simulation uses a different swarm robot algorithm entirely, replacing the KBS framework with the multiagent framework instead. In this simulation, the agents attempt to imitate the behavior of birds in flight using a modified version of the boid algorithm.

In this simulation, the agents do not have a set goal or status, and all simply follow the same behavior pattern. When initialized, each agent is given their position, velocity, and detection radius. At each time step, an agent senses which agents are within its detection radius, similar to the KBS algorithm, and calculates its change in velocity from each of those agents based on one of three

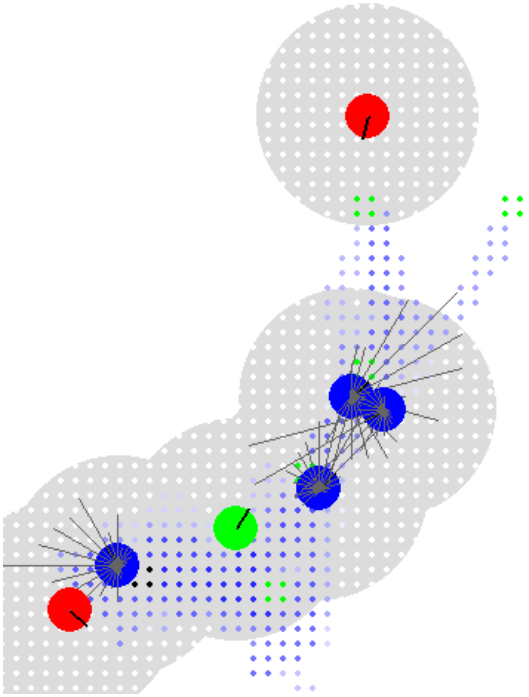


Figure 4.8: The results partway through the ACO simulation. All food sources have trails connecting them, but the final two food sources are separated by a forking pathway.

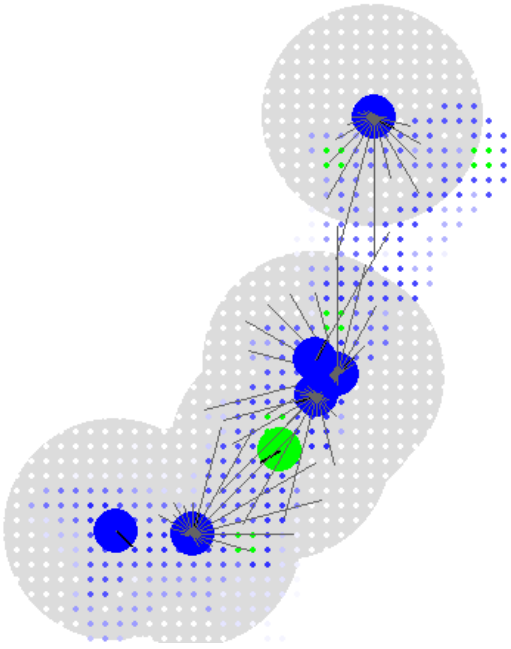


Figure 4.9: The results near the end of the ACO simulation. The final two food sources have formed a path between them, meaning agents have successfully made contact with all food sources in a single exploration.

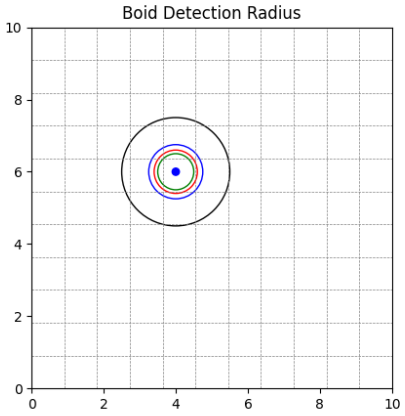


Figure 4.10: The ranges for each of the three rules controlling a single agent's movement in the boid algorithm. The black circle represents the maximum range of the agent's sensing. The blue circle represents the innermost range at which cohesion applies, the green circle represents the innermost range where alignment applies and the red circle represents the maximum range where separation applies.

behaviors. If another agent is within two fifths of its maximum vision radius, it applies a separation value proportional to the inverse of the distance squared, meaning that a neighboring agent that is too close to the agent will cause it to move away very quickly. Additionally, grid points around the outer edge of the simulation have been set to act as repellents, causing the same avoidance behavior if an agent gets too close to the edge. If a neighboring agent is in the outer half of the detection radius, it triggers cohesion, causing the agent to move towards it. Lastly, if a neighboring agent is in the outer two thirds of the detection radius, it triggers alignment, causing the agent to change its velocity to more closely match the neighboring agent's. All of these values are then averaged out by the number of neighboring agents, and multiplied by a weighting value: .9 for separation, .7 for cohesion, and .3 for alignment. Separation caused by the repelling grid points along the edge of the simulation is not affected by the number of neighbors and is given a weight of 1, ensuring it remains the largest change in velocity if triggered. The ranges for each of these rules can be seen in Figure 4.10

Like the ACO simulation, no neural network is used as part of this experiment. Instead, the results serve as a benchmark test that the multiagent framework is able to use a well-known swarm robotics algorithm such as boids and produce the expected results. As can be seen in Figures 4.11 and 4.12, the agents begin spaced apart on the environment, but eventually merge together, moving as a single flock.

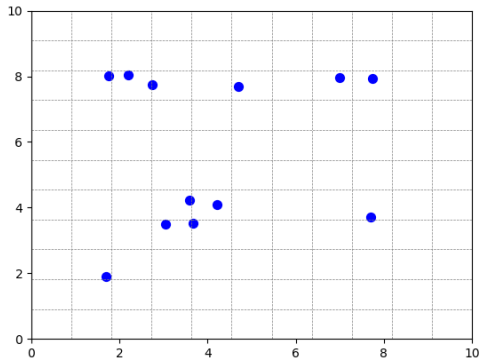


Figure 4.11: The multiagent boids algorithm near the beginning of the simulation. The boids are spread out across the environment.

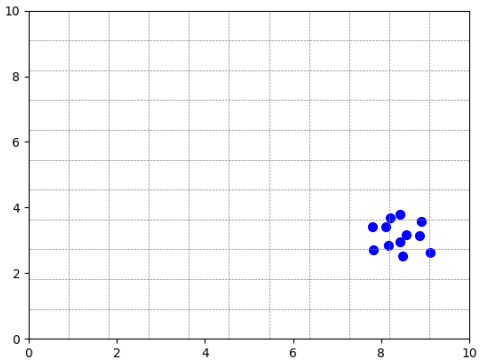


Figure 4.12: The multiagent boids algorithm near the end of the simulation. The boids have now all grouped together and are moving as a single unit.

Chapter 5

Summary and Conclusions

Based on the benchmark simulations, the swarm robot frameworks are capable of imitating the behavior of real-world organisms and materials. The KBS and multiagent frameworks are equally capable of this. However, the multiagent framework, by its design, more easily allows swarm simulations that do not follow the specific sensing and movement methods of a real-world Kilobot. Additionally, instead of this behavior being encoded manually, it has been shown that these simulations can be run using a neural network to determine their behavior.

Chapter 6

Future Work

With a working framework for swarm robot simulations, and proof that neural networks can be used to implement swarm behavior, we can begin imitating more complex swarm behaviors. Of particular interest are slime molds, specifically their abilities to effectively path find and store memory despite a lack of any kind of central nervous system. Additionally, the use of Kilobots in previous experiments have led to difficulties in properly implementing and studying swarm algorithms. To circumvent this issue, a custom-designed swarm robot with different methods of sensing, communicating, moving, and programming is currently in development.. The multiagent framework will be instrumental in testing and studying the ability of this new swarm robot model in its implementation of swarm algorithms.

Bibliography

[1] Toshiyuki Nakagaki, Hiroyasu Yamada, and Ágota Tóth. Maze-solving by an amoeboid organism. *Nature*, 407:470–470, 2000.

[2] Atsushi Tero, Seiji Takagi, Tetsu Saigusa, Kentaro Ito, Dan P. Bebber, Mark D. Fricker, Kenji Yumiki, Ryo Kobayashi, and Toshiyuki Nakagaki. Rules for biologically inspired adaptive network design. *Science*, 327:439–442, 1 2010.

[3] Reza Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Transactions on Automatic Control*, 51:401–420, 3 2006.

[4] Optimisation of boids swarm model based on genetic algorithm and particle swarm optimisation algorithm (comparative study). In *Proceedings - 28th European Conference on Modelling and Simulation, ECMS 2014*, pages 643–650. European Council for Modelling and Simulation, 2014.

[5] Brett M. Martin, Ryan D. Winz, Luke J. McFadden, and Tor J. Langehaug. Distributed boids simulation: Performance analysis and implementation challenges. In *Proceedings - 2023 Congress in Computer Science, Computer Engineering, and Applied Computing, CSCE 2023*, pages 780–785. Institute of Electrical and Electronics Engineers Inc., 2023.

[6] I. Lebar Bajec, N. Zimic, and M. Mraz. The computational beauty of flocking: Boids revisited. *Mathematical and Computer Modelling of Dynamical Systems*, 13:331–347, 8 2007.

[7] Ralf Mayet, Jonathan Roberz, Thomas Schmickl, and Karl Crailsheim. Antbots: A feasible visual emulation of pheromone trails for swarm robots. In Marco Dorigo, Mauro Birattari, Gianni A. Di Caro, Ren'e Doursat, Andries P. Engelbrecht, Dario Floreano, Luca Maria Gambardella, Roderich Gross, Erol Sahin, Hiroki Sayama, and Thomas Stutzle, editors, *Swarm Intelligence*, pages 84–94, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Mohamed S. Talamali, Thomas Bose, Matthew Haire, Xu Xu, James A.R. Marshall, and Andreagiovanni Reina. Sophisticated collective foraging with minimalist agents: a swarm robotics test. *Swarm Intelligence*, 14:25–56, 3 2020.

[9] Enzo Grossi and Massimo Buscema. Introduction to artificial neural networks, 12 2007.

[10] A novel foraging algorithm for swarm robotics based on virtual pheromones and neural network. *Applied Soft Computing Journal*, 90, 5 2020.

[11] Craig W Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21:25–34, 1987.

[12] Cristian Jimenez-Romero, David Sousa-Rodrigues, Jeffrey H. Johnson, and Vitorino Ramos. A model for foraging ants, controlled by spiking neural networks and double pheromones. 7 2015.

[13] Joshua Hecker, Kenneth Letendre, Karl Stolleis, Daniel Washington, and Melanie Moses. Formica ex machina: Ant swarm foraging from physical to virtual and back again. volume 7461, pages 252–259, 09 2012.

[14] A. Boussard, J. Delescluse, A. Pérez-Escudero, and A. Dussutour. Memory inception and preservation in slime moulds: The quest for a common mechanism. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 374, 2019.

[15] Tetsu Saigusa, Atsushi Tero, Toshiyuki Nakagaki, and Yoshiki Kuramoto. Amoebae anticipate periodic events. *Physical Review Letters*, 100, 1 2008.

[16] Masashi Tachikawa. A mathematical model for period-memorizing behavior in physarum plasmodium. *Journal of Theoretical Biology*, 263:449–454, 4 2010.

[17] Farhad Soleimanian Gharehchopogh, Alaettin Ucan, Turgay Ibrikci, Bahman Arasteh, and Gultekin Isik. Slime mould algorithm: A comprehensive survey of its variants and applications. *Archives of Computational Methods in Engineering*, 30:2683–2723, 5 2023.

[18] F. Ghanbari, F. Costanzo, D. P. Hughes, and C. Peco. Phase-field modeling of constrained interactive fungal networks. *Journal of the Mechanics and Physics of Solids*, 145, 12 2020.

[19] Farshad Ghanbari, Joe Sgarrella, and Christian Peco. Emergent dynamics in slime mold networks. *Journal of the Mechanics and Physics of Solids*, 179:105387, 2023.

[20] Y Xiao, N Fani, F Tavangarian, and C Peco. Nested structure role in the mechanical response of spicule inspired fibers. *Bioinspiration Biomimetics*, 19(4):46008, may 2024.

[21] Olivia Lowe, Christian Peco, and Fariborz Tavangarian. Modeling of the Bending Behavior to Study Nested-Cylinder Structure in Spicules. In *TMS 2023 152nd Annual Meeting & Exhibition Supplemental Proceedings*, pages 1215–1221, Cham, 2023. Springer Nature Switzerland.

[22] Farshad Ghanbari and Christian Peco. Eulerian Finite-strain Elasticity with Phase-field and the Reference Map Technique. *Transactions of the Japan Society for Computational Engineering and Science*, 27:8–13, 2023.

[23] Joe Sgarrella, Farshad Ghanbari, and Christian Peco. I-STL2MOOSE: From STL data to integrated volumetrical meshes for MOOSE. *SoftwareX*, 21:101273, 2023.

[24] Derek Gaston, Chris Newman, Glen Hansen, and Damien Lebrun-Grandié. MOOSE: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design*, 239(10):1768–1778, 2009.

[25] Manik Kumar, Joe Sgarrella, and Christian Peco. Neural networks for emergent behavior in biological microstructures. *Engineering Computations*, ahead-of-p(ahead-of-print), jan 2024.

[26] Manik Kumar, Nilay Upadhyay, Shishir Barai, Wesley F Reinhart, and Christian Peco. A bio-lattice deep learning framework for modeling discrete biological materials. *Journal of the Mechanical Behavior of Biomedical Materials*, 164:106900, 2025.

[27] Shishir Barai, Feihong Liu, Manik Kumar, and Christian Peco. Neural network-driven framework for efficient microstructural modeling of particle-enriched composites. *Materials Today Communications*, 42(October 2024):111278, 2025.

[28] Feihong Liu, Andrea P Argüelles, and Christian Peco. Numerical dispersion and dissipation in 3D wave propagation for polycrystalline homogenization. *Finite Elements in Analysis and Design*, 240:104212, 2024.

[29] Andrew Adamatzky. *Thirty Seven Things to Do with Live Slime Mould*, volume 23, pages 709–738. 2017.

[30] Madeleine Beekman and Tanya Latty. Brainless but multi-headed: Decision making by the acellular slime mould physarum polycephalum, 11 2015.

[31] Physarum polycephalum - a new take on a classic model system, 9 2017.

[32] Heiko Hamann. *Swarm Robotics: A Formal Approach*. Springer International Publishing, 2018.

[33] Michael Rubenstein, Christian Ahler, Nick Hoff, Adrian Cabrera, and Radhika Nagpal. Kilobot: A low cost robot with scalable operations designed for collective behaviors. *Robotics and Autonomous Systems*, 62:966–975, 2014.

[34] Eric Rohmer, Surya P. N. Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326, 2013.

[35] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm robotic behaviors and current applications, 4 2020.

[36] Álvaro Gutiérrez. Recent advances in swarm robotics coordination: Communication and memory challenges. *Applied Sciences (Switzerland)*, 12, 11 2022.

[37] Michael Rubenstein. Self-assembly and self-healing for robotic collectives. 12 2009.

[38] Alan Oliveira de Sá, Nadia Nedjah, and Luiza de Macedo Mourelle. Distributed and resilient localization algorithm for swarm robotic systems. *Applied Soft Computing*, 57:738–750, 2017.

[39] Luneque Silva Junior and Nadia Nedjah. Wave algorithm applied to collective navigation of robotic swarms. *Applied Soft Computing Journal*, 57:698–707, 8 2017.

[40] Seth Tisue. Netlogo: Design and implementation of a multi-agent modeling environment. 01 2004.

[41] Xin-She Yang. *Nature-Inspired Metaheuristic Algorithms*. 07 2010.

[42] Marco Dorigo and Mauro Birattari. *Ant Colony Optimization*, pages 36–39. Springer US, Boston, MA, 2010.

[43] Literature review on travelling salesman problem international journal of research literature review on travelling salesman problem. *Article in International Journal of Research*, 2018.

[44] Yunxiang Zhang, Yanling Zhou, and Jing Cao. Research and simulation of ant colony foraging behavior based on netlogo. In *Proceedings of the International Conference on Computation, Big-Data and Engineering 2022, ICCBE 2022*, pages 177–180. Institute of Electrical and Electronics Engineers Inc., 2022.

[45] Wei Xiang, Jiaping Ren, Kuan Wang, Zhigang Deng, and Xiaogang Jin. Biologically inspired ant colony simulation. *Computer Animation and Virtual Worlds*, 30, 9 2019.

[46] Eduardo G. Rodriguez, Christian Peco, and Daniel Millán. The influence of material properties distribution of waves in 1d: Application to cryoultrasonics. *Mecánica Computacional*, 39(7):217–217, 2022.

[47] Eduardo G. Rodriguez, Daniel Millán, and Christian Peco. Representative Volume Element (RVE) Analysis for mechanical characterization of ice with metallic's inclusion of micro/nano particles. *XXXIX Congreso Argentino de Mecánica Computacional y I Congreso Argentino Uruguayo de Mecánica Computacional*, 40:506, 2023.

[48] Farshad Ghanbari, Joe Sgarrella, and Christian Peco. Coding soft matter with bionetworks-inspired emergent principles. *Bioinspiration, Biomimetics, and Bioreplication XIII*, 12481:5–14, 2023.

[49] Alexandre Guével, Yue Meng, Christian Peco, Ruben Juanes, and John E Dolbow. A Darcy–Cahn–Hilliard model of multiphase fluid-driven fracture. *Journal of the Mechanics and Physics of Solids*, 181:105427, 2023.

[50] Benjamin W. Spencer, Wen Jiang, John E. Dolbow, and Christian Peco. Pellet cladding mechanical interaction modeling using the extended finite element method. *Top Fuel 2016: LWR Fuels with Enhanced Safety and Performance*, pages 929–938, 2016.

[51] T. Shimada, K. Nishiguchi, C. Peco, S. Okazawa, and M. Tsubokura. Eulerian formulation using lagrangian marker particles with reference map technique for fluid-structure interaction problem. In *9th edition of the International Conference on Computational Methods for Coupled Problems in Science and Engineering*, pages 1–7, 2021.

[52] T Shimada, K Nishiguchi, C Peco, S Okazawa, and M Tsubokura. Eulerian unified formulation for fluid-structure interaction problems using marker particles with Reference map, 2022.

[53] Koji Nishiguchi, Tokimasa Shimada, Christian Peco, Keito Kondo, Shigenobu Okazawa, and Makoto Tsubokura. Eulerian finite volume method using Lagrangian markers with reference map for incompressible fluid–structure interaction problems. *Computers Fluids*, 274:106210, 2024.

[54] Rene Y. Choi, Aaron S. Coyner, Jayashree Kalpathy-Cramer, Michael F. Chiang, and J. Peter Campbell. Introduction to machine learning, neural networks, and deep learning. *Translational Vision Science and Technology*, 9, 2020.

[55] Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8, 12 2021.

[56] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alex Wiltschko. A gentle introduction to graph neural networks. *Distill*, 6, 8 2021.

[57] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.