Simulated Self-Assembly of a Robot Swarm

COMP5400M Bio-Inspired Computing CW2

Tom Milner, Tyler Green, Tom Reynolds

May 2025

1 Introduction

Self-assembly is a form of sophisticated collective activity demonstrated by a number of biological systems, whereby groups of individuals create large physical structures by linking themselves together. While on their own, individuals can only have a relatively small effect on their environment, the self-assembled structure can interact and operate at a scale orders of magnitude larger. One of the most visually striking examples is the ability for ants to construct a living bridge across impassible terrain, to create a smooth path for following individuals. [1]

This is of interest to researchers in the field of robotics because of the potential for swarms of simple, inexpensive robots to collectively self-assemble into vastly more complex entities. One such system proposed by Rubenstein et al. [2] attempted to create a thousand-strong group of autonomous robots, called Kilobots, which cooperate through localised interactions with a highly robust algorithm for shape assembly. While the system was largely successful, limitations in the system speed and the shapes it can create do exist. Shapes take a long time to form, around 12 hours with the full swarm, and they must be continuous with no holes or gaps.

This project aims to first implement the base Rubenstein algorithm for a robot swarm in a Python based simulation, before addressing some of the limitations in the original system, most notably by developing the ability for the swarm to cross gaps between shapes. The simulation attempts to be as faithful to the capabilities of Kilobots as possible, particularly with regard to communication and localisation, where the simple, low-power design of Kilobots restricts complexity. Every agent receives an identical set of instructions, and through purely local interactions the swarm exhibits large scale shape formation.

2 Base Algorithm

The self-assembly algorithm used by the Kilobots is shown in Figure 1 [2]. All agents receive a bitmap representation of a shape, which is scaled to match the area the group can cover. Initially the group is in a tightly packed but arbitrarily shaped cluster, where four agents (shown in green) are designated as the seeds. Agents in the main cluster have no knowledge of their location; the seeds set the origin and location of the shape. These seeds are also the start of the gradient system, which provides the cluster with a general sense of distance to the origin, represented in agent body lengths.

Alongside gradient formation, the Kilobots have only two other simple abilities, edge-following and localisation. Agents release from the cluster from highest to lowest gradient, and orbit around successive members of the cluster to the seeds. Once at the seeds, the agents can start estimating their own position based on distances to known, localised neighbours. As long as an agent has at least three stationary, localised neighbours, it can calculate its position using a distributed form of trilateration. Once the agent knows where it is, it can check if it is inside the shape. When this happens, the agent continues to edge-follow until either it is about to exit the shape, or it is about to edge-follow around another agent with the same gradient.

This sequence of release from the cluster, edge-following, and stopping continues until the whole shape is full. Agents in the final shape are freely positioned, meaning they have no pre-determined finishing

positions, like they would in task or path based algorithms. This is one aspect that gives the algorithm its robust nature, as small errors or variance in agent packing in the final shape does not impact overall shape formation. Any agents left over after the shape is complete remain edge following around the shape until manually removed.

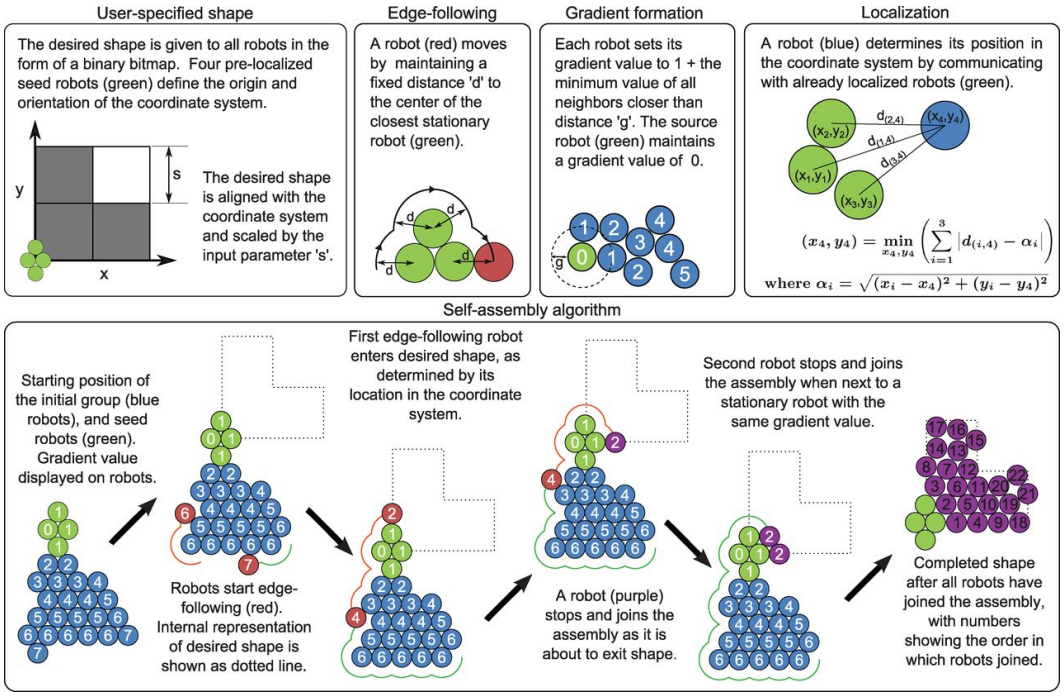


Figure 1: The collective self-assembly algorithm developed by Rubenstein et. al. [2]

3 Implementation

3.1 Simulation Engine

The simulation runs on a custom-built 2D simulation engine, developed in Python 13.3.2 [3]. The engine uses the "pygame" library [4] to handle rendering graphics to the screen, and to manage keyboard/click events. The simulation is run frame-by-frame at a user-specified frame rate measured in frames per second (fps). There are two phases to each frame:

1. `update()` The update phase of the frame is responsible for updating any internal state within the simulation. Computation that must happen on every frame runs here, for example, calculating the new positions of moving simulation objects.
2. `draw()`: During the draw phase, all simulation objects are drawn to the screen - no computation happens during this phase.

The `draw()` phase can be run at a slower frame rate than the `update()` phase. This allows processor intensive simulations to update their internal state many hundreds of times per second, whilst only drawing themselves to the screen every few seconds, allowing the processor to prioritise processing the `update` phase.

3.1.1 Simulation Objects

Every object within the simulation is a `SimulationObject`, examples of which can be seen in Figure 2. These are Python objects that are updated and rendered by the simulation engine, and obey rudimentary physics like velocity and collision detection. They are circular in shape, and have a red radial line pointing in their current direction. The body of a `SimulationObject` can be set to any RGB colour.

Each simulation object provides the engine with an update function and a draw function, which are called when the engine enters each respective phase. The object can move in one of two movement modes:

1. **Vector-based**: Objects will follow a specified velocity vector in a straight line.
2. **Orbit-based**: Objects will orbit around another object, with zero space between their perimeters.

Simulation objects also detect collisions with other simulation objects.

Figure 2: An example `SimulationObject`.

3.1.2 Neighbourhoods & Spatial Hashing

For collision detection (and later, localisation (3.3.5)) simulation objects need to query the objects in their immediate surroundings. A naive method to achieve this would be to query the positions of every object in the simulation, and compare the position to that of the original object to see if they are nearby. While this would be effective, it would be inefficient, as the original object would be computing object properties from objects too far away in the simulation to be of interest.

The simulation implements a more efficient neighbour-finding algorithm based on spatial hashing [5]. Each simulation is split into a grid of *neighbourhoods*. A neighbourhood is a square region of defined size within the simulation. The simulation keeps track of object locations and assigns each object to a corresponding neighbourhood. Then, for an object to access it's nearest neighbours, it only has to query its current neighbourhood, and the neighbourhoods surrounding it - this makes the solution robust for objects that are on the borders of neighbourhoods.

3.1.3 Simulation Tests

To test the simulation engine itself, a suite of test suites were written for: movement, collision detection, orbiting, and edge-following (orbiting an agent until collision with another agent, then orbiting that agent, and so on).

3.1.4 Analytics Engine

The simulation has a built in analytics package developed with Matplotlib [6] for use in generating plots and statistics based on the agents location, available at any point during the simulation when paused. The primary purpose is to provide current visual information on agent location errors, which are calculated as the difference between an agents local position and its actual position on the surface. The plots available are largely based on those found in the original publication [2], and are shown and discussed in Section 3.4. The statistics provided with the plots are used to quantify the localisation errors and how they develop as the agents get further away from the seeds, and can also be used to compare the ability of the agents to form different shapes.

3.2 Simulating Kilobots

A Kilobot, otherwise known as an agent, is simulated by an `Agent` class. This class inherits a Simulation Object, enabling edge-following, collision detection, and neighbourhood detection. To achieve the functionality of a Kilobot the agent implements a modified version of the state machine detailed in the original paper [2]; some extra states are added to allow for extended functionality, as will be explained in Section 4.

The overall behaviour of the Kilobots was implemented as a state machine in this simulation for simplicity and readability. The state machine can be seen in Figure 3, and works as follows:

`IDLE`
The initial state. If an agent is a seed agent, it immediately localises using it's set position and transitions to the `LOCALISED` state. If the agent is ready to start edge following (see Section 3.3.3), transition to the `MOVING_AROUND_CLUSTER` state.

`LOCALISED`
In this state, the agent is fully localised and stationary. It no longer edge-follows, and the simulation does not have to worry about agents being bumped out of shape, so regular re-localisation is not necessary.

`MOVING_AROUND_CLUSTER`
To move around the initial cluster of agents, the agent edge-follows until it encounters a seed robot, upon which the agent enters the `MOVING_OUTSIDE_SHAPE` state. If the agent enters the `MOVING_AROUND_CLUSTER` state more than twice, it must have encircled the shape twice without finding a position in which to localise. Thus, it must transition to the `BRIDGING` state to find a new shape.

`MOVING_OUTSIDE_SHAPE`
The agent is moving outside the shape when it is edge-following robots that are localised within the shape, but not within the shape itself. The agent will edge-follow until they enter the shape, where they enter the `MOVING_INSIDE_SHAPE` state. If the agent reaches a seed agent, they have orbited the enter shape without entering it, and revert back to the `MOVING_AROUND_CLUSTER` shape.

`MOVING_INSIDE_SHAPE`
Once the agent is within the shape it edge-follows until meeting the stopping criteria (Section 3.3.6), upon which enters the `LOCALISED` state.

`BRIDGING`
Bridging is an extension of the original Kilobot agent implementation, allowing agents to "smell" other shapes in the simulation and build bridges towards them (see Section 4.1). Bridging agents will build bridges towards a shape until they are within a shape, upon which they enter the `MOVING_INSIDE_SHAPE` shape.

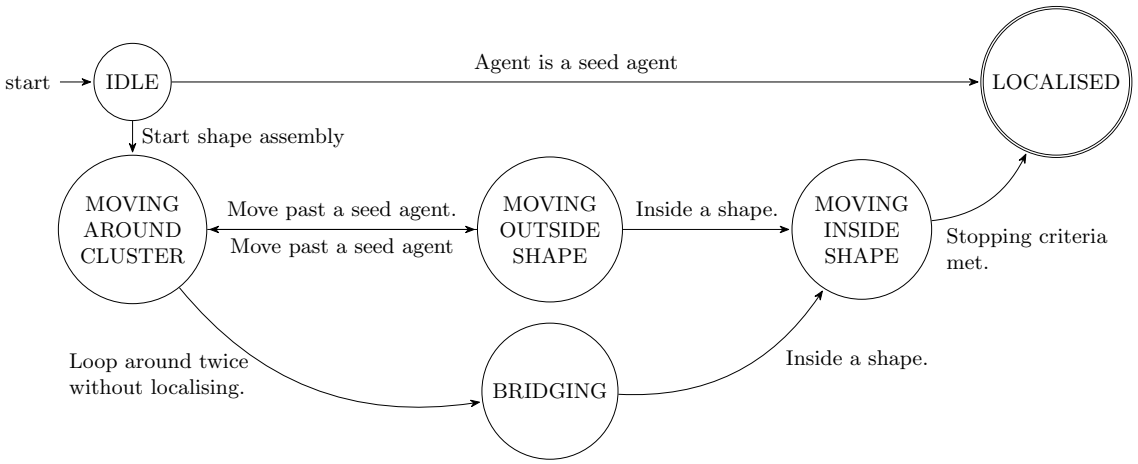


Figure 3: Agent state machine.

The simulation supports adding random noise to the localised position of the Agent, enabling the simulation to closely mimic the real-word results observed in the original paper [2].

3.3 Shape Assembly

3.3.1 Shape Creation

The desired shape is created by the user in a graphical user interface (`shape_creation.py`, Figure 4) by clicking on a canvas of pixels to toggle them between black and white (and grey for multicolour shape formation, explained in 4.2). The shape can be saved to the directory as a bitmap file, in which each pixel is represented by a value of 255 for white or 0 for black, before being loaded into the simulation.

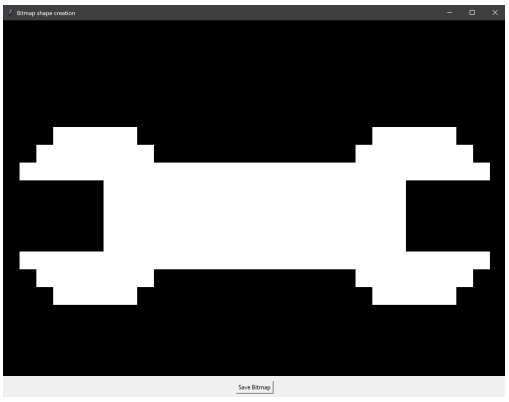


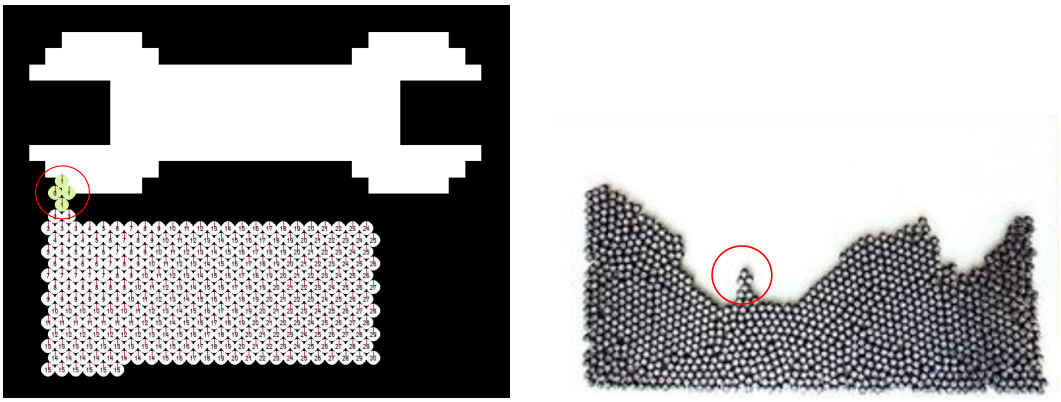
Figure 4: Shape creation GUI

3.3.2 Initial State

In its initial state, the simulation comprises of the four green seed agents, a tightly packed cluster of the rest of the agents, and an image of the desired shape to be formed. Figure 5

shows a comparison of the original Kilobot initial state with the simulated version, with the seed agent location highlighted. The simulated cluster appears more regular than the physical version, although this is purely for simplicity as the agents can begin in any cluster shape. The number of agents in the cluster is set so that the agents cover a slightly higher proportion of the window area than the shape,

The shape is imported from the directory and scaled to a set proportion of the simulation window size. It is then aligned with the coordinate system with the bottom left corner placed at the seed origin, and the bitmap representation of the shape is given to all agents, for use in shape assembly. At this point, the simulation is ready to start, and agents begin to release from the cluster to start edge-following.



(a) Simulation initial state

(b) Kilobots initial state [2]

Figure 5: Initial states of the Kilobots and the simulation. Seed robots circled in red

3.3.3 Edge Following

To begin edge following a the `start_edge_following()` is called. This checks that the agent is on the edge of the cluster and no neighbours are moving. However, to ensure that not all agents start edge following immediately, the following probability in Equation 1 must also be met.

$$P_{\text{start}} = \frac{1}{\text{average\_start\_time} \cdot (\text{fps} + 1)} \tag{1}$$

This makes the average time between agents starting equal to the average start time defined by the user. Once started the `follow_edges()` method checks and fixes collisions, before updating the current orbit object which the agent orbiting around, allowing the agent to move clockwise around the cluster.

3.3.4 Gradient Formation

The gradient formation closely follows the original publication by updating the agents gradient to one more than the lowest gradient of agents it is touching. This creates a general sense of distance throughout the swarm, with the gradient representing distance in agent body-lengths from the seed agents.

3.3.5 Localisation

The localisation algorithm also closely follows the approach in the original publication, developed specially for the limited communication capabilities of the Kilobots. It is an approximated form of distributed trilateration shown in Equation 2, which aims to find the position at which the calculated distances to each neighbour matches the measured distances between agents.

$$\min_{X_{self}, Y_{self}} \left(\sum_{N} \left| D_N - \sqrt{(X_N - X_{self})^2 + (Y_N - Y_{self})^2} \right| \right) \tag{2}$$

In each iteration, for every localised neighbour, the unit vector from the agents previous localised position and the current neighbours localised position is found. This vector is then scaled by the measured distance between the two agents, resulting in a new estimated position. Finally, The agents localised

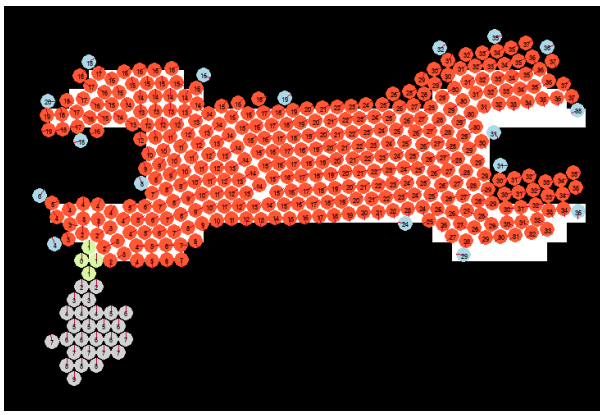
position is updated by moving one-quarter of the way towards this new position. The algorithm repeats the process, until the difference between the previous localised position and the new localised position reaches a threshold.

3.3.6 Stopping Criteria

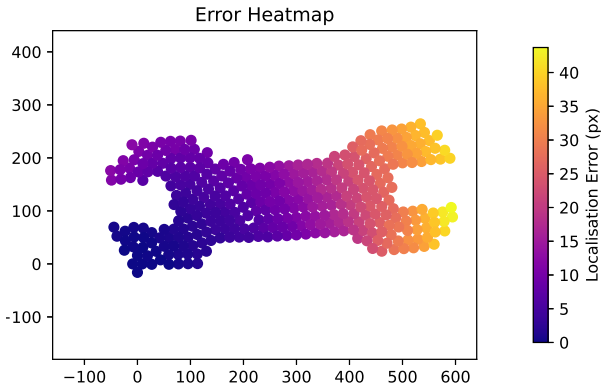
Once the agent is in the `MOVING_INSIDE_SHAPE` state there are two criteria that may cause it to stop. The first is a check to see if it is about to edge-follow to a point outside the shape, which stops the agents on the edge of the shape. The second stops the agent if it about to edge-follow past another agent that is also inside the shape and has the same or greater gradient value. This causes the agents to fill the shapes in waves of equal gradient value.

3.4 Results

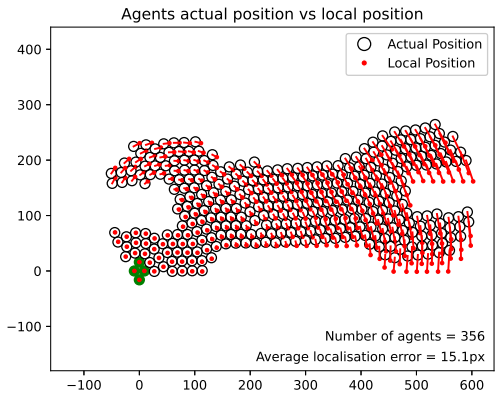
Figure 6 shows a wrench shape formed by 400 agents, with accompanying error plots generated by the analytics engine. The agents successfully completed the shape, with a small amount of skew due to localisation errors propagating through the shape as seen in Figures 6b and 6c, showing an average error of 15 pixels and a maximum error of around 40 pixels. Even with the agents with higher gradient having higher localisation errors (Figure 6d), the shape is still formed largely correctly, demonstrating the robustness of the algorithm. Agents left over after the shape was completed remained edge-following, as in the Kilobots implementation [2].



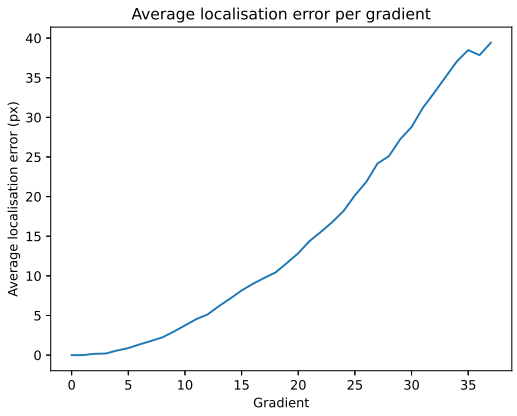
(a) 400 agent completed wrench



(b) Heatmap of localisation error



(c) Comparison of agent local and actual position



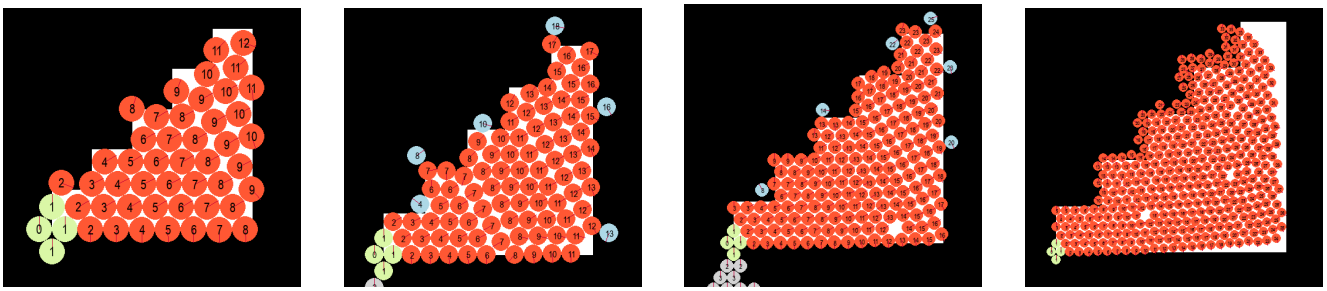
(d) Average localisation error for each gradient value

Figure 6: Completed 400 agent wrench with error plots generated by the analytics engine

In addition, the variance of agent packing and localisation error was investigated across the same shape with differing numbers of agents. To increase the difficulty for the agents, a small amount of random noise was added in their localisation. The results are shown in 7. The trials with 50, 100, and 200 agents all successfully completed the shape, however the 400 agent trial did not, likely due to the increasing errors

expanding the size of the shape the agents are attempting to fill. As expected, the average error increases as the number of agents increases, although visually the 200 agent trial appears closest to the desired shape. With fewer agents, the features of the shape are too small in comparison to the agent size, so they cannot be represented clearly by the circular agents. Therefore for more complex shapes, increasing the number of agents would likely often be beneficial even with the associated extra localisation error.

It is also clear that variance in the agent packing does not affect overall shape formation, with packing changing across all shapes and small holes present between agents in the 200 and 400 agents shapes (Figures 7c and 7d). One source of packing variability that is not present here though is agents being pushed around after stopping in the shape. This could be implemented to make the simulation more realistic, as currently stopped agents are immovable. The Kilobots however do get pushed around fairly frequently after stopping in the tests completed by Rubenstein et al. [2].



(a) 50 agents, average error 0.221 px

(b) 100 agents, average error 2.31 px

(c) 200 agents, average error 6.18 px

(d) 400 agents, average error 16.3 px

Figure 7: Variance in packing and error in the staircase shape for different numbers of agents

4 Algorithm Improvements

4.1 Bridging

One clear limitation of the base algorithm is that only continuous shapes can be assembled. More complex structures which have separate islands cannot be built. The reason for this is the constrained edge following motion of the agents prevents them from exploring empty space away from the swarm. Since this is a core aspect of the algorithm, it is vital for any additional behaviour to not deviate from these simple motion rules. Consequently, random walk motion commonly seen other bio-inspired works and in nature such ant colony optimisation [7] cannot be used as this requires agents to move independently from the swarm to explore. This highlights the first problem which needs to be solved: how do the agents know which direction to travel?

In the current implementation, agents can only detect the shape colour at their current localised position in the environment. Long-sighted vision to detect other areas would make agents too complex and remove the simplicity constraints that this self-assembly algorithm was designed for. As agents have an image of the desired shape, it was deemed acceptable that the centre of masses (COM) of each island could be calculated and their vectors from the seed. From a nature perspective this can be seen as a simplification of using cues in the external environment such as diffusion of heat or scents to travel in the general direction of the source. This was implemented using OpenCV [8] before agent creation to detect and segment islands and generate a list of the centre of masses. The locations were then transformed into the localised frame of reference relative to the seed and this was used when initialising the agents.

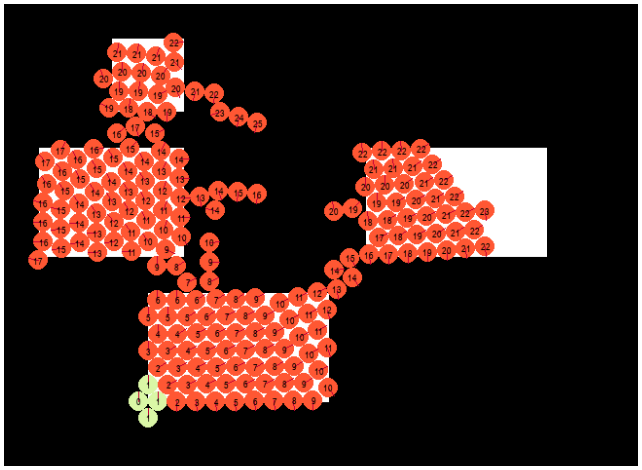
The next problem to solve is how do the agents travel in that direction? As agents can only edge follow, the edge of the swarm must move in the desired direction. This can be achieved by stopping the agent at the desired location on the edge, forming a bridgehead, eventually leading to a bridge being created between shapes. To detect when to bridge, the `check_bridging()` method returns whether the agent is on a vector between the closest COM and another COM, where the closest COM is that of the shape the agent is currently orbiting. To ensure that all COMs can be reached, but the closest one is favoured,

the probability of an agent bridging can be seen in Equation 3.

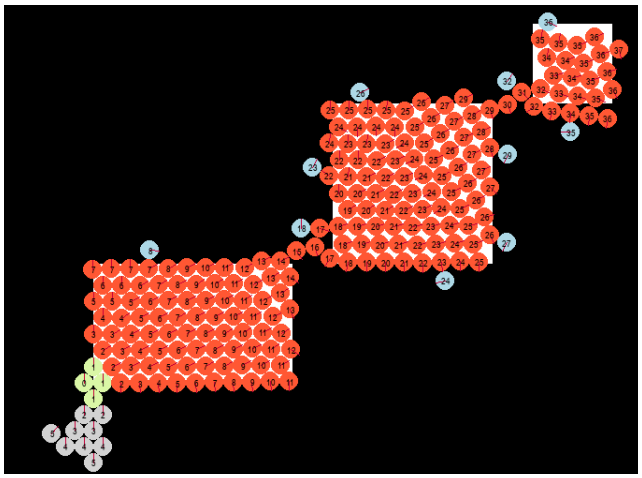
$$P_{bridging} = \frac{d_{ideal}}{\|\mathbf{d}_{COM}\|} \cdot p_{nominal} \cdot (num\_loops - 1) \tag{3}$$

To ensure this bridging behaviour is only activated once the current area the swarm is edge following is full, the `check_if_looped()` method is used. This only transitions to the `BRIDGING` state once the agent passes to the negative left axis of the seed for second time.

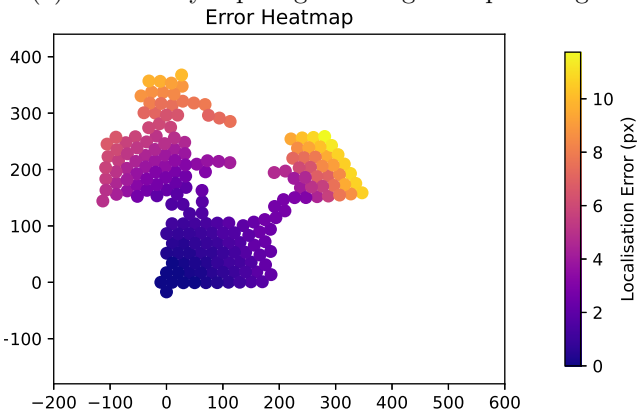
As seen in Figure 8 the implementation of the bridging was successful in enabling the other islands to be filled in. Figure 8a highlights the probable nature of the bridging. As seen, the islands are bridged at the closest points but some agents have attempted to bridge at other. This is because during the time taken for the final shortest bridge to be made, a large proportion of the agents have already travelled around and are now in the bridging state. This coupled with the possibility of the D_{ideal} being too large, results in the longer bridges seen attempted. A more complete example can be seen in Figure 8b. However, the smallest, final island looks to be skewed indicating there is substantial error between the localised positions and the actual positions, quantified in Figure 8d. This highlights that the co-linear nature of the bridge means the agents struggle to localise correctly and thus induces the large error seen towards the end of the bridging.



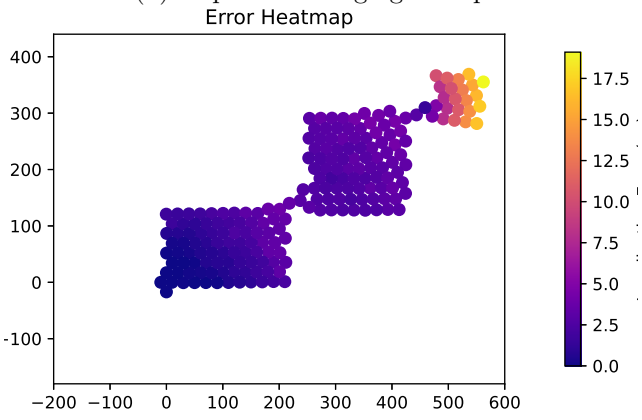
(a) Probability aspect generating multiple bridges



(b) Sequential bridging example



(c) Error map of four island simulation



(d) Error map of sequential bridging example

Figure 8: Examples of bridging and bridging induced error

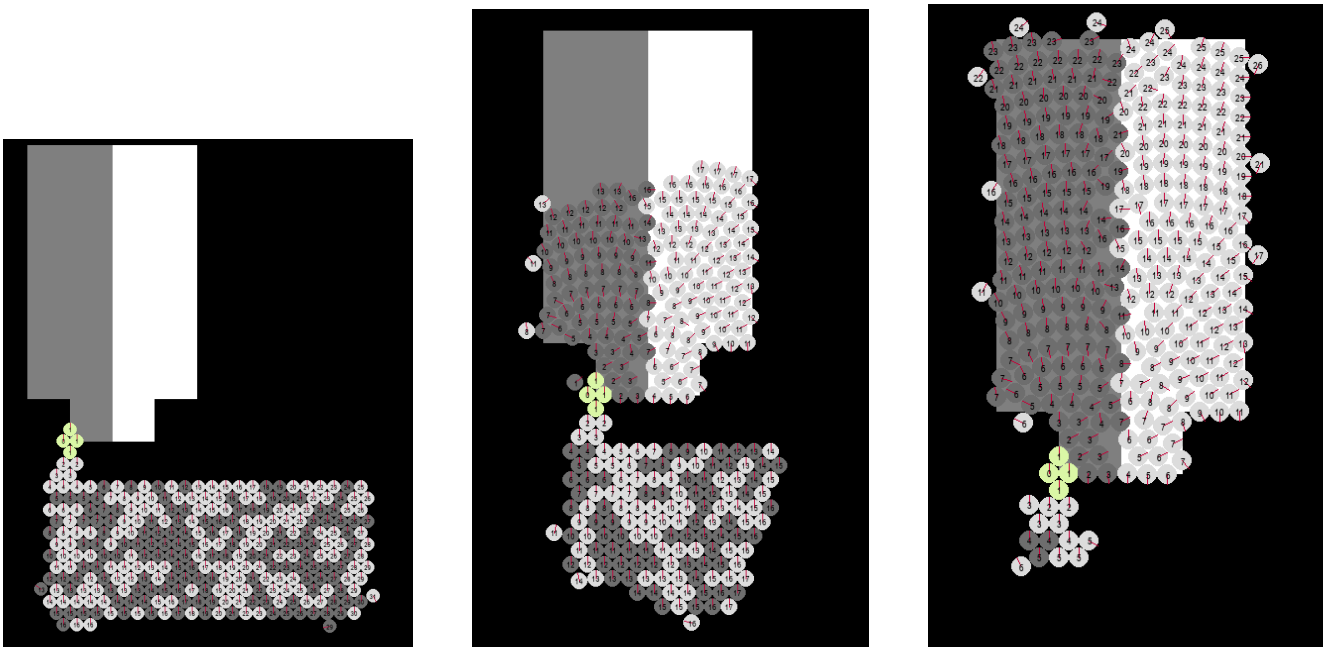
To improve this capability, modifying the bridge geometry to allow better localisation is key. This could be done by widening the bridge, making it two agents wide would make more agents to be available for the bridging agent to localise off. Another method, reducing the required agents needed would be for the bridge to zigzag. However, this would need the heading of neighbours, adding complexity to the agent. Another improvement needed to finalise the algorithm would be to develop a methodology for the agents to un-bridge. Current simulations leave bridging agents in their place, removing the accuracy

the assembled shape to the initial desired shape. In addition, this means that additional agents would not need to be added to the simulation to facilitate this feature.

4.2 Multicolour Shape Formation

Another limitation with the original algorithm is that it is only able to make shapes with one 'type' of agent inside. In real-world use for this type of robotic swarm construction, it would be more useful to have the ability to create shapes with different agents, varying for example in colour or material. This flexibility could be used to create more interesting images, or even to assign groups of agents as different 'components' in a larger structure. This is implemented in the simulation in its simplest form, with shapes able to have two colours - white and grey.

The shape creator (`shape_creator.py`) has the capability for grey pixels to be set through pixel toggling. When saved, the black, grey and white pixels are given values of 0, 127 and 255 respectively. Upon loading into the simulation, the `place_shape()` method detects that grey pixels are present and sets the simulation to `'multicoloured'` mode. At the same time, it calculates the proportion of the shape that is white, which is used to ensure that the correct number of white agents are placed in the cluster, and does the same for grey agents. Shape assembly is completed in largely the same way as the base simulation, with the exception that white agents only transition into the `MOVE_INSIDE_SHAPE()` state when they are in the **white** area of the shape, effectively meaning they cannot 'see' the grey area, and vice versa for grey agents. The assembly process is shown in Figure 9, with the agents successfully forming a two-tone shape. Excess agents are left to edge-follow as before (Figure 9c).



(a) Simulation initial state

(b) Halfway through shape formation

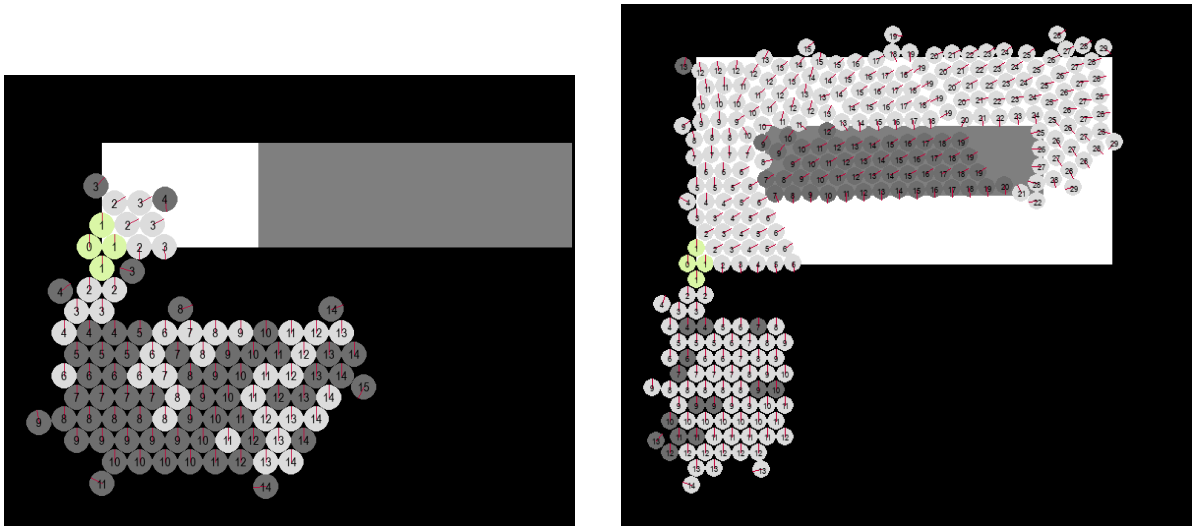
(c) Multicolour shape formation completed

Figure 9: Stages of multicolour shape formation

This assembly method leads to a limitation with this implementation, where agents of a colour that does not touch the seed rely on agents of the other colour to provide a bridge before they can start filling the shape. The range of shapes that can be formed is therefore reduced; if the bridge required is too large, agents that cannot reach their colour will keep edge-following around the cluster, blocking agents of the bridge colour from releasing and ultimately causing the bridge to never form (Figure 10a).

As all shape assembly states still use the same edge-following and stopping as the base algorithm, another limitation occurs in the case where one colour of agents blocks the other colour from reaching certain areas of the shape. If this occurs, the shape will not be filled in correctly by agents (Figure 10b). One way of mitigating both of these limitations would be to implement a more intelligent way of placing and releasing the agents in the initial cluster. At present, the locations of each colour of agent are randomly

distributed throughout the cluster. However the distribution of colours in the shape could be analysed and applied to the agents in the cluster so they release roughly in the same order in which they can fill the shape, always allowing them to complete the shape.



(a) Failure due to insufficient bridge

(b) Failure due to cut off area

Figure 10: Failure modes of multicolour shape formation

5 Conclusion

Biological self-assembly such as the collective behaviour shown by army ants uses large groups of simple individuals to create complex shapes with much greater influence on the environment than the individuals alone. This behaviour has inspired the creation of swarms of low-complexity robots that achieve significant global behaviour by assembling into different forms, such as the Kilobots. The implementation of the Kilobots in simulation shown here demonstrates the ability of these agents to autonomously self-assemble with consistent success, as well as highlighting some limitations of the original system.

A biologically-inspired bridging algorithm was introduced to allow to agents to form shapes with multiple 'islands', through a simplified version of attraction to a beneficial region of space applied to the agents. This significantly increases the flexibility of the agents, as one cluster and one set of seeds can form multiple shapes rather than requiring multiple different sets of agents, and could be used to simulate crossing impassible terrain. An implementation of multicolour shape formation was also explored, facilitating different types of agents to form together in one shape. The simulation engine also provides scope for future improvements and addition to the shape formation algorithm, with potential avenues including having agents of varying size, or simulating large scale damage and repair. Gaining further properties inspired by the natural world could push the simulation to be a base for a highly intricate, but still robust, robotic swarm for complex self-assembly.

References

[1] C. Anderson, G. Theraulaz, and J.-L. Deneubourg, "Self-assemblages in insect societies," *Insectes soc.*, vol. 49, pp. 99–110, 2002.

[2] M. Rubenstein, A. Cornejo, and R. Nagpal, "Programmable self-assembly in a thousand-robot swarm," *Science*, vol. 345, no. 6198, pp. 795–799, 2014.

[3] P. S. Foundation, "Python 3.13.2 programming language." https://www.python.org/downloads/release/python-3132/. Accessed: 04/05/2025.

[4] pygame, "Pygame UI library." https://www.pygame.org. Accessed: 04/05/2025.

[5] E. Hastings and J. Mesit, "Optimization of large-scale, real-time simulations by spatial hashing," 01 2005.

[6] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[7] M. Dorigo, M. Birattari, and T. Stützle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[8] OpenCV Team, "OpenCV library." https://opencv.org/, 2025. Accessed: 2025-05-05.