Langton's Ant

By: Dylan Bassi (20462393) & Yuqian Hu (20534552)
For: Prof. E. Vrscay – PMATH 370
2018/04/23

1 Introduction

1.1 Introduction to Langton's Ant

Langton's ant, invented by Chris Langton in 1986, is a discrete dynamical system that follows a virtual ant as it runs along a square lattice. At every discrete time point, the ant will move forward in time and move one square forward, changing the state of the square that it has just entered. Depending on the state of the square, the ant will either rotate left or right or not rotate at all. Then as this dynamical system continues, as the ant in accordance to the rules and moves over the same lattices that it has traveled before; continuing to change the states of the squares and the direction of the ant[1]. To help us visualize this result, we can colour the tiles of the lattice based on their state.

Since it's creation many extensions to the ant have been created with a great amount of work being done finding rules and configurations that allow for interesting behaviour to emerge of many iterations[2,3]. The extensions explored in this report will be allow for arbitrary rule length, and for allowing the ant to take on different states that respond to different coloured tiles with varying behaviour. These behaviours include chaotic, periodic and fractal-like natures over many iterations. It is also possible to create initial configurations that allow for the interaction of ants in such as a manner as to transfer logical information and pass it through gates[2]. Although not performed for this project, the implications of this feature are briefly discussed.

1.2 What to expect from Langton's Ant:

In the most basic version where there are only two colours, black and white, and ant can only turn left or right depending on the colour, there are three categories of movement that the ant can experience. We will be using that expected behaviour to showcase our algorithm is producing the same expected results as others. In the first few hundred steps, the ant walks in a random manner often creating a very simple pattern along the way. This is often called the simplicity mode of behaviour. In the following several thousand iterations, the ant follows a chaotic mode of behaviour with no particular pattern or reason. After the chaos, we reach the ordered phase where in the simplest model, the ant will proceed to create a highway towards the end of the board. This is shown in Figure X where the previous 3 behaviours are reproduced by our algorithm in Sec. 2.

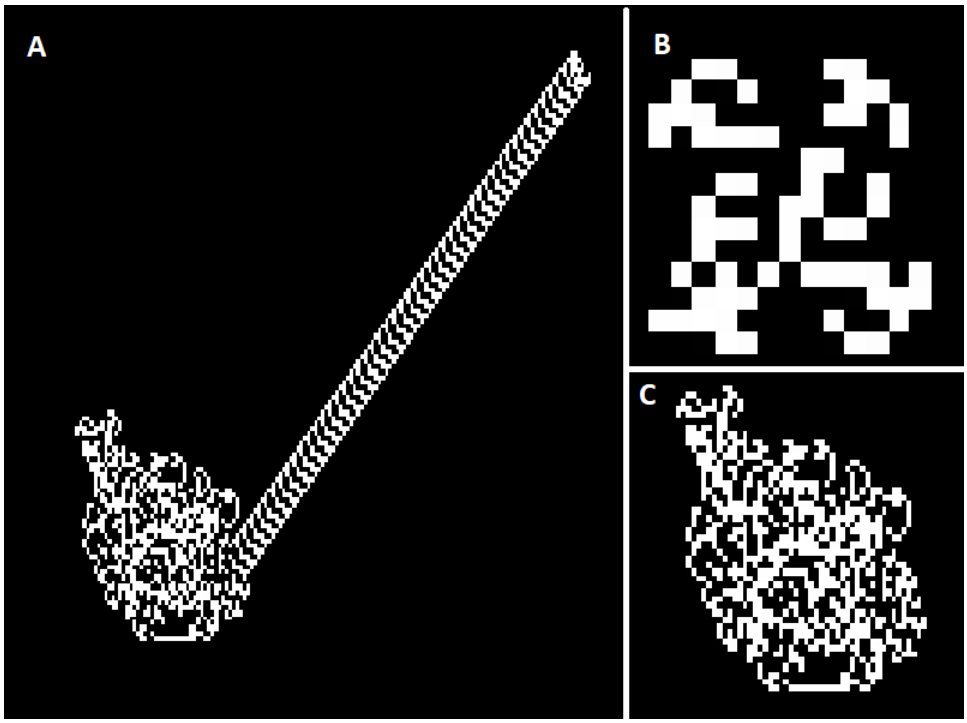

Figure 1 – Showcases our algorithm in it's most basic form; one single ant with two possible colours and left or right directions. A) showcases the ant in its final form of behaviour after 15000 iterations. B) Showcases the ant in its simplicity mode of behaviour after 500 iterations. C) Showcases the chaotic nature of the ant after 10000 iterations.

1.3 Connection to the Course:

Like many of the discrete dynamical systems studied in class, Langton's Ant often display chaotic and sensitive dependence to initial conditions behaviour. Depending on the formulation of the rules initially set up by the creator and the number iterations defined, the destination of the ant will differ greatly. The interesting thing regarding Langton's Ant comes from the fact that ant appears to be in a long preperiodic orbit. The ant cycles the board through various configurations until finally it reaches a configuration that will only result in the same pattern of changes. There has been a lot of work in the area to create rules and specific starting conditions such that the emergent order builds a specific structure, although little has been concluded regarding the theoretical understanding of this system.

2 Background

2.1 The Original Algorithm:

Langton's ant is a cellular automaton. A cellular automaton is a discrete system that evolves a two-dimensional grid over time via some sort of algorithm. The original example of Langton's ant was characterized by an "ant" tile that could move around the grid. The grid is composed of tiles that can be either in a 0 or 1 state. The ant changes its direction based on the tile state, changes the state of the tile, and then moves forward to the next square. The algorithm employed for the basic example of Langton's ant is very simple and can be broken down into the follow steps:

1. There is an "ant" tile that each iteration checks the grid for the state that the tile is in.
2. Based on the state of the tile, the ant turns left (if in a 0 state) or right (if in a 1 state).
3. The ant changes the state of the tile and moves to tile in front of it
4. Steps 1-3 are repeated until the ant hits a boundary of the grid.

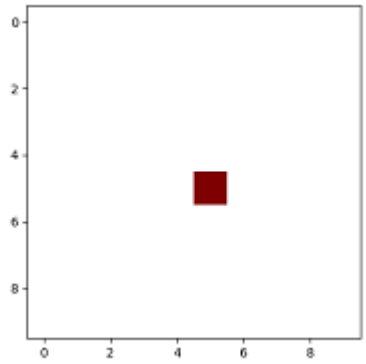

Figure 2a – The initial state of the board. The red pixel represents the ant's position.

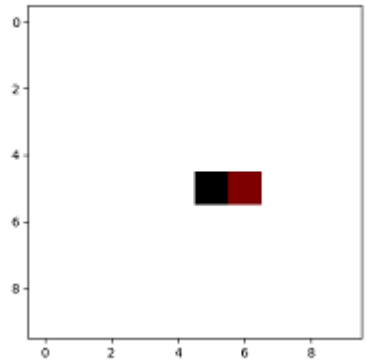

Figure 2b – The state of the board after one iteration. The ant changed the state of the pixel and moved to the right.

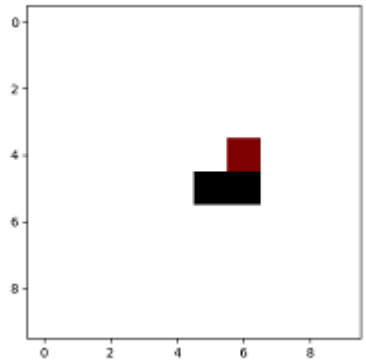

Figure 2c – The state of the board after two iterations. Once again, we see another black tile, and the ant moves right.

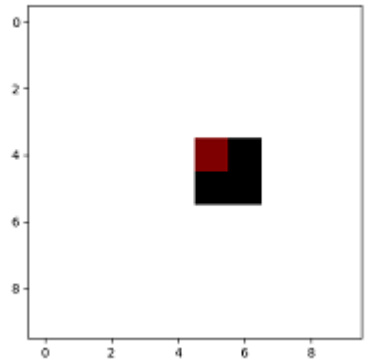

Figure 2d – The state of the board after three iterations. The ant continues circling producing another black tile.

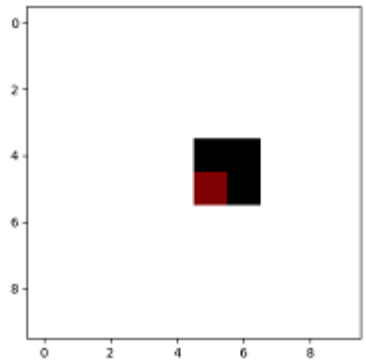

Figure 2e – The state of the board after four iterations. The ant completes a 2x2 square of black tiles.

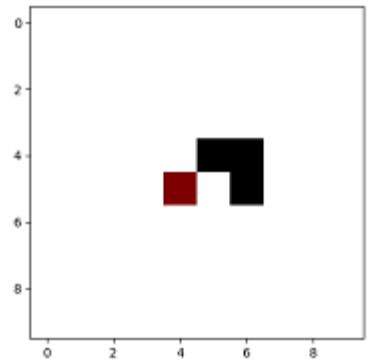

Figure 2f – The state of the board after five iterations. The ant turns left and makes the tile white.

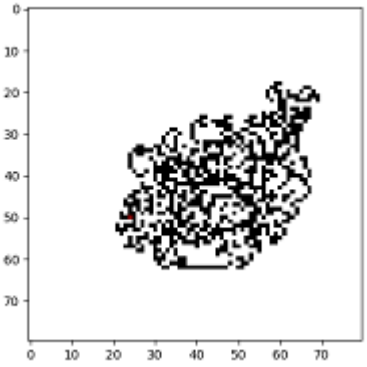

Figure 2g – The state of the board after 10000 iterations. The board is very messy, with no clear pattern emerging.

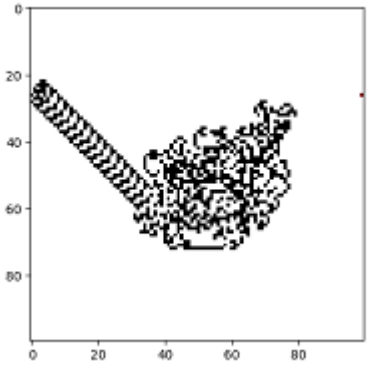

Figure 2h – The state of the board after 11668 iterations. The ant has fallen into a periodic orbit, producing a "highway" until it hits the edge of the grid.

2.3 An Algorithm for Generalized States of the Board:

A natural extension to Langton's ant would be to allow generalized states of the board, instead of just 0 and 1. This creates a whole new world of complexity to the ant, as various rules can be assigned to the ant regarding how it should change its direction. Various complex behaviours are achievable using this generalized algorithm by allowing a "rule string". The rule string is a series of 'L's and 'R's corresponding to instructions on whether to turn left or right. The state of the tile corresponds to which character in the rule string to decide. For example, in the string "RL" the ant turns right on a tile in state 0 and left on a tile in state 1. This produces the Langton's ant map previously described. The rule string allows the behaviour of the ant to be encoded as an initial user parameter by assigning a sequence of characters as different directions for the ant to move. The algorithm for the generalized states is executed in a very similar manner to the original algorithm:

1. The ant checks the tile that it is on for its state.
2. The ant uses the state of the tile to determine which character from the rule string to use. This means, the ant uses the rule[tile_state] value to determine which direction to turn.
3. The ant increments the state of the tile, if the state of the tile is equal to the length of the rule string, the value is reset to 0.
4. The ant moves forward to the next tile.
5. Steps 1-4 are repeated until the ant hits the edge of the board.

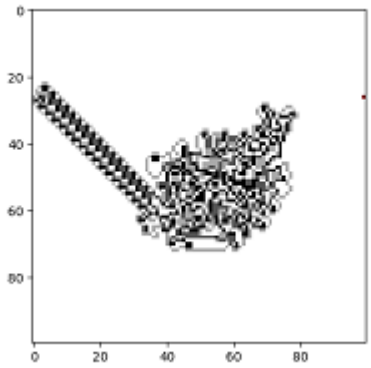

Figure 3a – A generalized board with rule 'RLRL' after 11668 iterations. This does not affect the path of the ant, merely increasing the number of states.

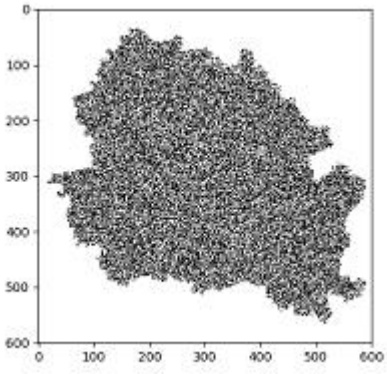

Figure 3b – A generalized board with rule 'RLR' after 1.5e7 iterations. The ant following this rule never falls into any periodic behaviour.

2.4 An Algorithm for General Ant States (Turmites):

The other way to extend the case of Langton's ant is to allow the ant to take on multiple states as well. These have been dubbed turmites, a portmanteau of Turing machines and termites. With turmites, it is not just the board that is assigned different state values, but the ants as well. This allows for the behaviour of a termite for a given square to be different based on its state. With a generalized turmite algorithm, one should be able to produce all the earlier ant maps, as they are just specific subsets of the multistate case. Once again, we can tweak our original algorithm just a bit to allow us to have multiple states of turmites:

1. The ant checks the tile it is on for its state and its own state.
2. The ant uses the rule corresponding to the state of the tile and its own state.
3. The ant changes its direction based on the rule and the state of the system.
4. The ant changes to the state of the tile to prescription of the rule
5. The ant moves forward one tile
6. Steps 1-5 are repeated until the ant hits the edge of the board.

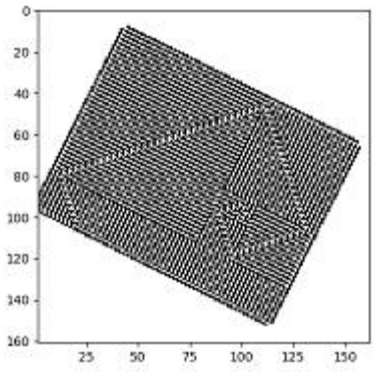

Figure 4a – A turmite map with rule '181181121010'. The turmite appears to trace out a Fibonacci spiral.

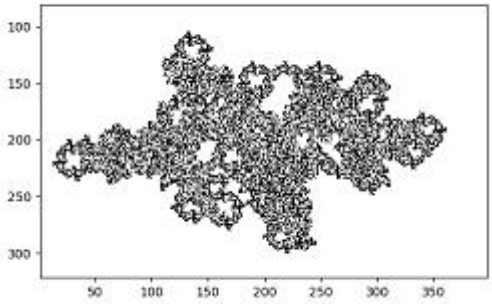

Figure 4b – A turmite map with rule '120121010011'. The turmite traces a complex pattern with recurring themes appearing throughout

3 Code

3.1 Coding Single State Ants:

The code for the single state ants is composed of three main parts:

1. A Global Part – Where the parameters are defined; the main function for the code; and a function which updates the state of our board and ants are contained
2. The Grid – A class which contains the board, and the function for plotting the system
3. The Ants – A class which we use to move across the grid's board in order to change the state of the system.

3.1.1 The Global Section:

In this section, the main parameters are defined by the user. These parameters include the size of the image, the number of iterations, the number of ants, and the initial positions and starting directions of the ants. In this pre-amble for the script, the rule is also defined as a string of 'L's and 'R's. The index of the string corresponds to the state of the tile the ant is on. Therefore, there are as many possible colours on the board as there are characters in our string. We can also change the restriction of the finite geometry in this section if desired. We could choose to allow the board to be generated with it or allow a toroidal geometry.

In our `main` function we first check the rule is actually a string. Next, we create an instance of the `Grid` class using the x and y dimensions as well as the geometry prescribed by the parameters. We also create all the ants and iterate the system until the maximum iteration value. This is executed by the `update(grid, ants, i)` function. This function uses the ant's `move` function. The function returns the updated grid and ant and instantly plots the state of the

system if any exception is encountered. Finally, once the maximum iteration value is achieved, the system is plotted, and an image is saved.

3.1.2 The Grid:

The grid is a class which not surprisingly acts as the grid for our ant to move on. When a grid is instantiated, it is assigned a board array of dimensions defined by the main parameters, with all tile values in an initial state of 0. Our grids come with a function to determine whether the ant is still on the board or not. They also come with the `final_plot(self, ants, step)`function. This function effectively places a visual marker for the ant and plots the 2-D array. Another zero-valued 2-D array is created of equal size to the board. The tile in this array that corresponds to the position of the ant is assigned a value of 1, while the other values are masked so that we can overlay the ant image over the board image.

```
def final_plot(self, ants, step):

    # plot the board state and ants
    # use a mask to make the ant array transparent and overlay only
    # the ants' positions onto the final result
    y = np.zeros((self.dimen[0], self.dimen[1]))
    for ant in ants:
        y[ant.position[0], ant.position[1]] = 1
    y = np.ma.masked_where(y == 0, y)

    # use imshow to print matrix elements using gray colormap. Ants are red.
    plt.imshow(self.board, cmap=plt.get_cmap('plasma_r'), interpolation='none')
    plt.imshow(y, cmap=mpl.cm.jet_r, interpolation='none')
    plt.savefig(rules[0]+'-'+str(num_ants)+'ants'+'-'+str(step+1)+'steps'+'.png')
```

Figure 5 – The final_plot function used to create the output image of the code. The array 'y' tracks the position of the ant and has all other values masked so we can overlay it on the image of the board.

3.1.3 The Ant:

The ant class is the workhorse of our code. Every instance of the ant is assigned four possible directions it could face, corresponding to up, down, left and right on the board. These values are kept in a list and cyclically permuted based upon the value of the tile, and the corresponding rule for the ant. The ant is also instantiated with the rules so that it does not have to be fed in every iteration. There are two functions that the ant uses: `move` and `cycle_dir`. The `cycle_dir` function takes the rule for the tile state and permutes through the possible orientations based on the rule.

The move function is what allows us algorithm to work. The function is fed in the ant, the board and the rules for the ant. First the ant must determine the state of the tile that is on. Next it uses the state of the tile to determine which character of the rule string is needed. The ant then changes it direction based on the rule string. For values of 'L' the ant permutes its orientation

forward in the list of possible directions. For values of 'R' the ant permutes its orientation backwards in the list of possible directions. After changing direction, the ant increments the state of the board. If the tile reaches the maximum value, the state of the tile is reset to 0. Finally, the ant moves forward and returns the updated state of the board.

```
    def move(self, board, rule):
        # get state of board and current direction
        state = board[self.position[0], self.position[1]]
        directive = rule[state]
        # change the ant's direction
        self.face_direction = self.cycle_dir(directive)
        # cyclically increment the state of the board
        board[self.position[0], self.position[1]] = (state + 1) % self.nstates
        # apply motion based on new direction
        self.position[0] = self.position[0] +
self.possiblefacings[self.face_direction][0]
        self.position[1] = self.position[1] +
self.possiblefacings[self.face_direction][1]
        return board
```

Figure 6 – The move function used to move the ant around the board. The ant teases out the rule for the state of the tile it is on and uses this directive to change its direction. The ant increments the state of the tile and proceeds to move onto the next tile.

3.2 Coding Turmites:

To generalize our system to allow for multiple ant states significant alterations to the format of the rule was required. The code in general follows the same structure as for the case of single-state ant. We start off with a global section that defines the parameters of the system, and the executes the main function of the code. This is followed by our two classes, the grid and the ant. This time however, the global section has an additional function, `splitrule(rule)`, which converts our rule string into an array of size M x N, where M is the number of possible turmite states and N is the number of possible tile states. Each member of this array corresponds to the behaviour of the turmite for a particular tile state and turmite state. An example of how this function acts on rules is presented in Fig. 7.

```
Rule String:
111180121010

Rule Array:
[['111' '180']
 ['121' '010']]
```

Figure 7 – An example of how the splirule function operates on strings.

3.2.1 Turmite Rules:

To truly generalize the algorithm, the structure of the rule had to be reconsidered. Previously, the rules for the algorithm forced us to proceed into the next state of the tile the ant occupied. This was not a problem, as any other structure would have resulted in the ant becoming stuck producing one type of tile. For turmites however, we do not necessarily want to always increment the state of the tile, as that may depend on the state of the turmite. Similarly, we do not

want to force our turmite to enter the next state after each iteration. Clearly, for the turmite rules we need to be able to encode three pieces of information at each iteration: how to change the tile state, how to change direction, and how to change the turmite state. Therefore, instead of each singlet member of the rule string being a rule, we must require that each triplet member of the rule string is a rule. The first number in the rule string corresponds to how to change the board state, the second number tells the turmite which direction to turn to, and the third tells us what state to change the ant to. As an example, in Fig. 7, if the ant is on a tile in state 0, and has a state of 1, its rule would be '121'. This tells the ant change the state of the tile 1, turn the direction corresponding to '2' and change its state to 1.

```
def splitrule(rule):
    rules = np.chararray((nstates,ncolours),itemsize=3)
    rules_lst = []
    k=0
    n_rules = len(rule)/3


    for i in range(n_rules):
        rules_lst.append(rule[i*3:3*(i+1)])

    for i in range(nstates):
        for j in range(ncolours):
            if (i > 0 and k <max((i,j))):
                k = k+1
            rules[i][j] = str(rule[3*(i+j+k):3*(i+j+k+1)])
    return rules
```

Figure 8 – The splitrule function, which transforms a turmite rule string into a rule MxN matrix, where M is the number of ant states and N is the number of tile states.

4 Behaviour of the System

4.1 Langton's Ant with Arbitrary Rule Length:

Based on a brief review of literature, theoretical descriptions for predicting the behaviour of ants remains a challenge[3,4,5]. In fact, the only way to determine the nature of the algorithm is to simulate it[5]. Langton's original ant showed a period of seemingly random behaviour before falling into a pattern that produces the familiar highway. This specific example of a cellular automaton was very early in the conception of the field and sparked great interest in how these systems might produce complicated behaviour from simple algorithms. The generalization of the ant further proves to us that changing the algorithms ever so slightly can produce vastly different and interesting behaviour. The only way to determine what behaviour an ant will display is by simulating it. Fig. 9 shows several examples of the behaviour that can be generated by Langton's ant. This behaviour ranges from chaotic, symmetric, space-filling, highway-producing and much more.

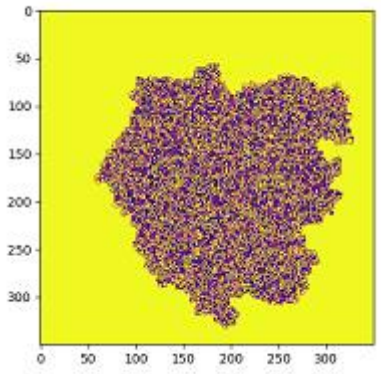

Figure 9a – A generalized board with rule 'RLR' after 2e6 iterations. This map is truly chaotic with no known periodic behaviour emerging.

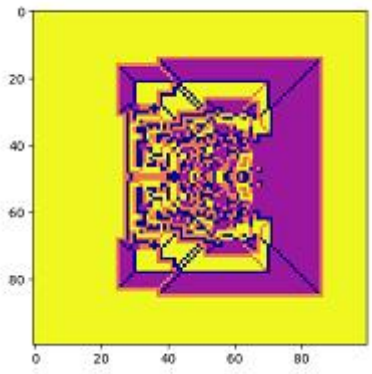

Figure 9b – A generalized board with rule 'RLLR' after 6e4 iterations. This map grows symmetric bilaterally. While preserving the general shape.

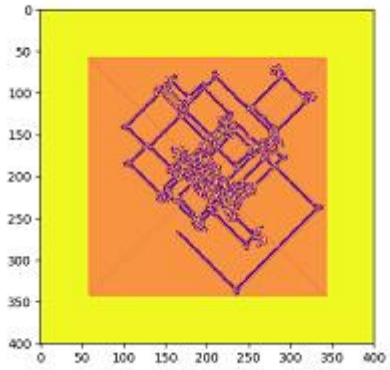

Figure 9c – A generalized board with rule 'LRRRRRLLR' after 4e5 iterations. This map expands in a square region. The ant remains within the edges of the square, creating highways.

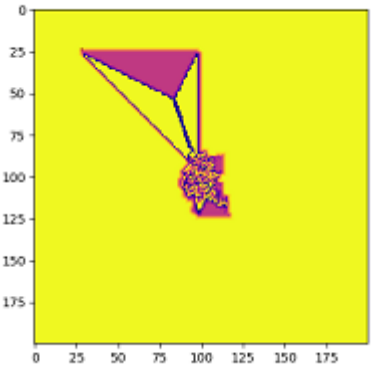

Figure 9d – A generalized board with rule 'RRLLLRLLLRRR after 4e4 iterations. This map expands a space filling triangle after an initial period of seeming chaos.

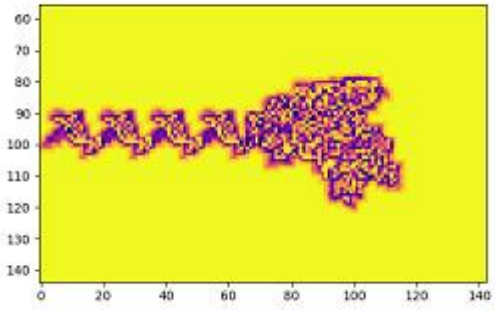

Figure 9e – A generalized board with rule 'LLRRRLRLRLLR' after 36797 iterations. This map produces a highway after a period of initial chaos more complex than Langton's ant.

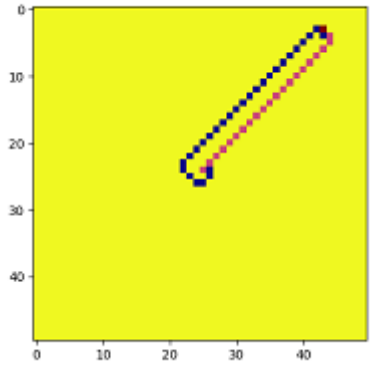

Figure 9f – A generalized board with rule 'RRL' after 4e2 iterations. This map almost instantly produces a highway with a simple asymmetry.

One of the interesting behaviours that we found was discovered by adding extra terms in the rule string that cause the ant to not change its direction at all. This was encoded in our rules as an 'N' so that whenever the ant is fed the 'N' rule, it does not change direction, and only increments the tile and move forward. This behaviour drastically changes the patterns that emerge. A few examples of how this addition to the rule changes the behaviour is present in Fig. 10, with the specific rule in Fig. 10f examined in greater detail in Fig. 12. The number of groups is not specific to producing the pattern, however increasing the value inherently increases the number of states we can have and thus anti-aliases the image for us.

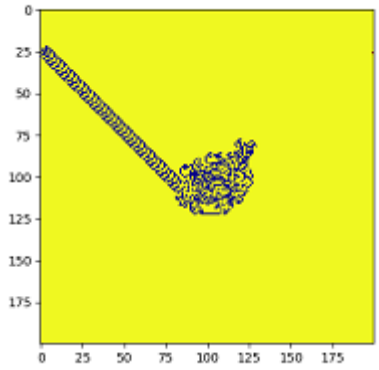

Figure 10a – The state of the board after 11668 iterations. The ant has fallen into a periodic orbit, producing a "highway" until it hits the edge of the grid. The red pixel represents the ant's position.

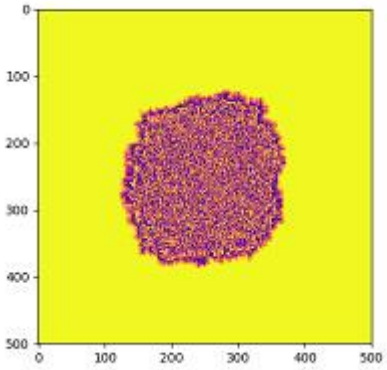

Figure 10b – Extending the rule with 8 groups of 'N' rules produces chaotic behaviour with no patterns ever emerging, regardless of the number of groups.

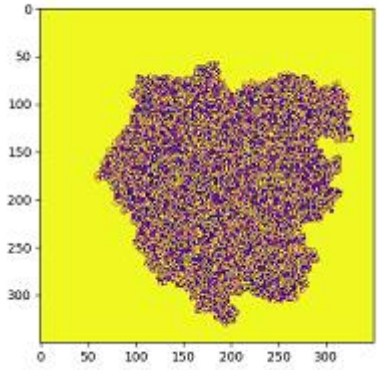

Figure 10c – A generalized board with rule 'RLR' after 2e6 iterations. Before addition of our 'N' rules, this map is purely chaotic.

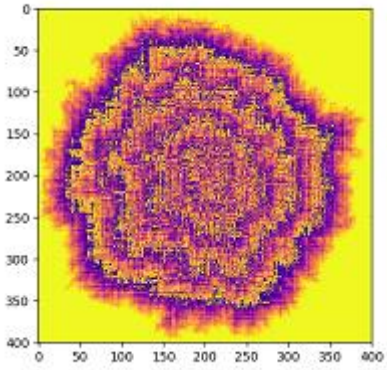

Figure 10d – Extending the rule with 16 groups of 'N' rules produces nearly similar concentric rings. Consecutive rings look alike, but minor variations lead to outer and inner rings appearing different.

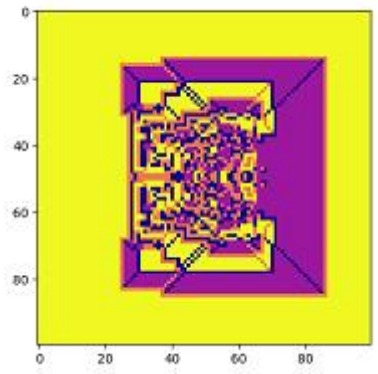

Figure 10e – A generalized board with rule 'RLLR' after 6e4 iterations. This map grows symmetrically along the bilaterally.

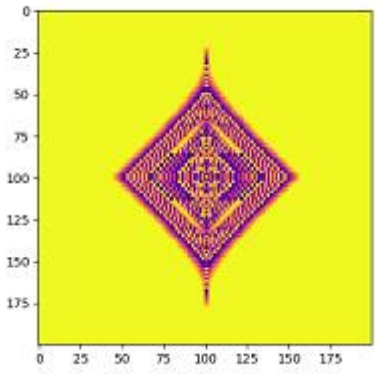

Figure 10f – Extending the rule with 8 groups of 'N' rules produces a symmetric diamond with fractal-like properties.

A closer inspection of Fig. 10f reveals some fascinating implications of these extra 'N' rules. In Fig.11 We see that adding a group of 4 'N' rules we can alter the behaviour from Fig. 10e to produce a new symmetric object. This object has the added property that it is fractal like in nature, producing self-similar sets over greater numbers of iterations. We also see that this property is not restricted to the number of groups of Ns that we add so long as they come in quadruplets. In Fig. 11b, we see how the addition of the '{N}' group increased the symmetry of the map.

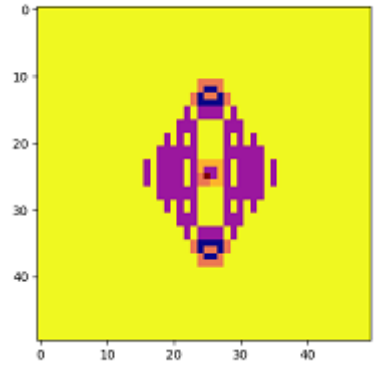

Figure 11a – A generalized board with rule 'RLLRNNNN' after 9e4 iterations. The map is clearly highly symmetric, although no indication of the fractal property has emerged.

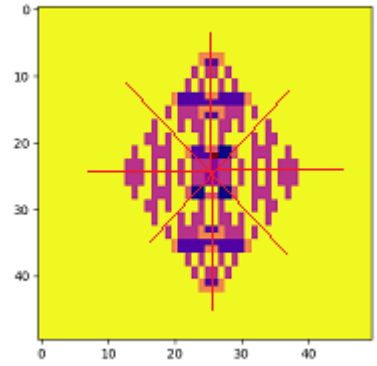

Figure 11b – A generalized board with rule 'RLLRNNNN' after 9e5 iterations. The map looks similar to higher number of iterations, with fractal nature slowly emerging. Planes of refection added to display increased symmetry.

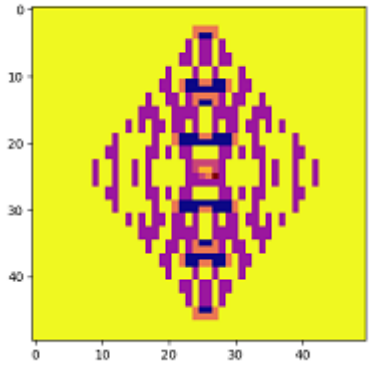

Figure 11c – A generalized board with rule 'RLLRNNNN' after 9e6 iterations. The fractal nature becomes evident, although detail is still not quite distinguishable.

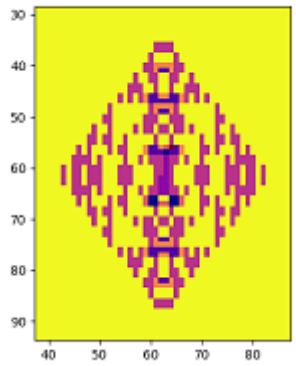

Figure 11d – A generalized board with rule 'RLLRNNNN' after 9e7 iterations. The fractal nature has become quite clear, looking like a Sierpinski Gasket.

After noticing how concatenating the {'NNNN'} rule onto the 'RLLR' rule gave rise to highly regular, self-similar shapes, it only seemed natural to examine what the addition of extra '{N}' rules would do. Not surprisingly, we observed that the deformation of the original structure was exaggerated. Contrasting Fig. 11d and 12b we can observe how the perfect fractal nature of the image is replaced with a nearly self-similar set. The addition of extra board states allows for a finer transition between consecutive states. This is observed as a 'fading out' of the purple to yellow in our image, best observed on the boundary, as the tiles smoothly transition from a low state to a high state. In Fig. 13, we see how increasing the number of '{N}' rules produces highly complex features. We slowly convert the symmetric 'RLLR' to a fractal structure with 2 and 3 groups of '{N}' and watch the behaviour convert back to another symmetric pattern, this time with several planes of reflection, picked up from the first addition of an '{N}' group.

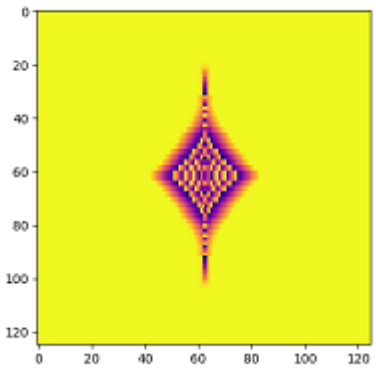

Figure 12a – A generalized board with rule 'RLLR'+8{'NNNN'} after 9e4 iterations. The map is clearly highly symmetric, already some of the fractal property can be observed.

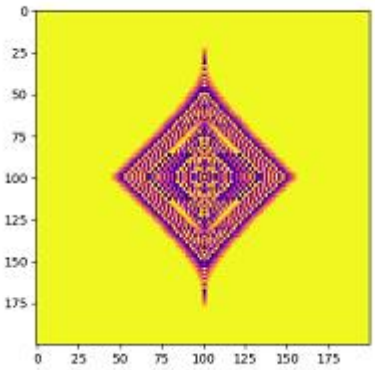

Figure 12b – A generalized board with rule 'RLLR'+8{'NNNN'} after 9e6 iterations. The map is highly fractal-like, although we still have a region of chaos in the center.

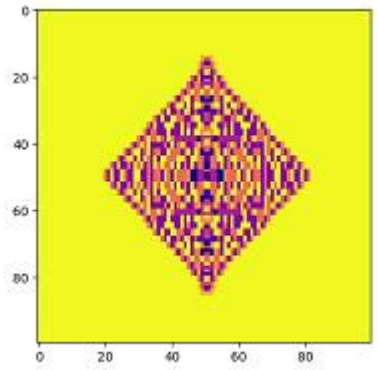

Figure 13a – A generalized board with rule 'RLLR'+2{'NNNN'} after 9e7 iterations. The map is highly fractal-like, although the fractal nature is visually lost due to shading differences; the interior fractal is a different shade than the exterior.

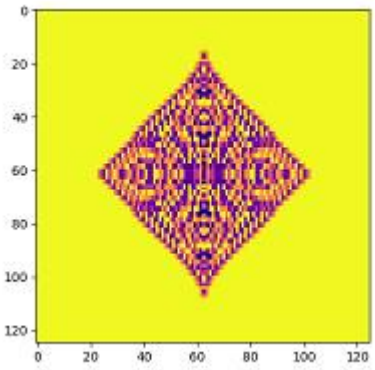

Figure 13b – A generalized board with rule 'RLLR'+3{'NNNN'} after 9e7 iterations. The map has lost regions of fractal behaviour, with a new symmetric pattern emerging.

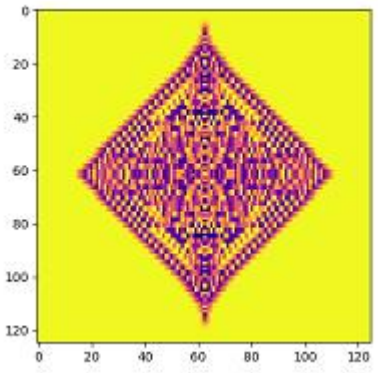

Figure 13c – A generalized board with rule 'RLLR'+4{'NNNN'} after 9e7 iterations. Self-similarity is nearly gone now as the image converts to a highly symmetric pattern with recurring features.

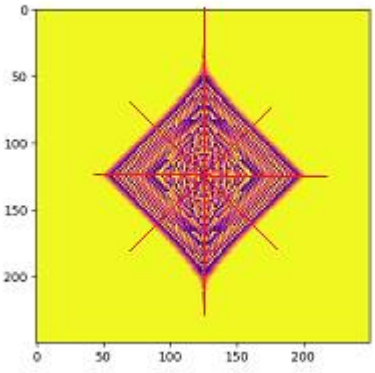

Figure 13d – A generalized board with rule 'RLLR'+8{'NNNN'} after 9e7 iterations. The map is hardly fractal like and has mostly taken on a symmetric pattern. The planes of reflection are overlaid to show the increased symmetry.

4.2 Behaviour of Turmites:

Similar to the behaviour of the ant, turmites have wildly unpredictable and remarkable structure. The original Langton's ant looks very much like a two-dimensional image, the ant traces out what appears to be a relatively thin line, not generally filling in much space. We are really only able to achieve any visual depiction of texture by incorporating extra terms in our rule string. In turmites, since the turmites themselves store information, we can achieve the depiction of texture and depth using only two colours; the behaviour of the turmites causes them to spend more time in certain regions of the map. An example of this can be observed in Fig. 14.

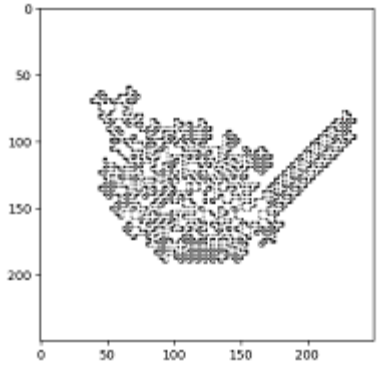

Figure 14a – A turmite with rule '1810211120121100l0'. The first two triplets encode the information for Langton's ant, while the rest cycle through our ant without changing direction, overlaying a texture.

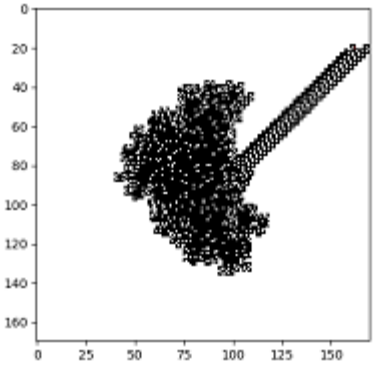

Figure 14b – A turmite with rule '181081110111'. This map displays how behaviour similar to the single-state ants can be reproduced with added texture, giving a feel for depth based on how turmites interact with the board.

Since we were able to induce fascinating behaviour by adding additional forward increments to the board, it was only natural to examine the effect of adding additional forward turmite rules. In order to move forward and increment the board while maintaining the original path, we must add additional states of and which increment the board without changing direction. This is very similar to what was done in Sec. 4.1 with the ant rules, except instead of just incrementing the state of the board, we also want to increment the state of the ant. Due to the structure of our triplets for our two-colour system we must add two additional triplets: {11x01x}, where 'x' is the next ant statel. Fig. 15 shows how the inclusion of these additional states leaves the overall structure the same, it merely reduces the density of points visited by the ant. Taking the symmetric region and

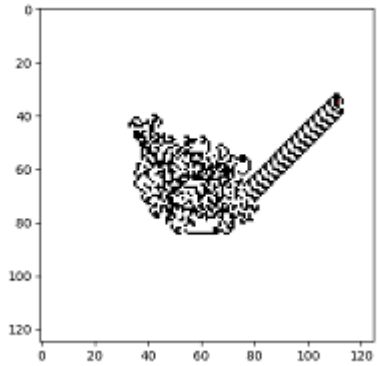

Figure 15a – A turmite with rule '180020' produces the familiar Langton's ant.

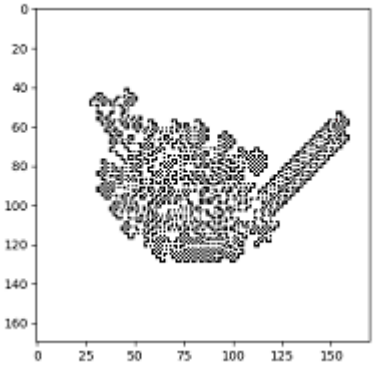

Figure 15b – A turmite with rule '181021110010' produces the same overall image, however there is a semi-inverted texture to it. The {110} and {010} triplet force our ant to cycle states moving forward.

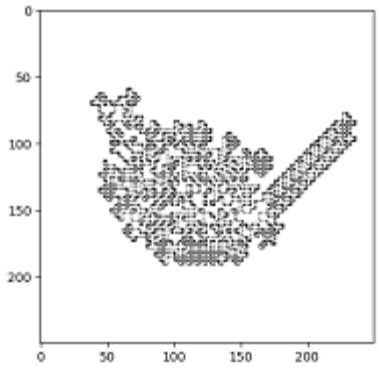

Figure 15c – A turmite with rule '181021…012110010' produces the same overall image, however the semi-inverted texture is getting less dense.

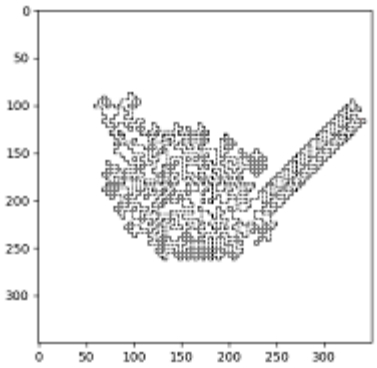

Figure 15d – A turmite with rule '181021…013110010' produces the same overall image, however the texture is beginning to get lost, there is only a vague appreciation of the density of states.

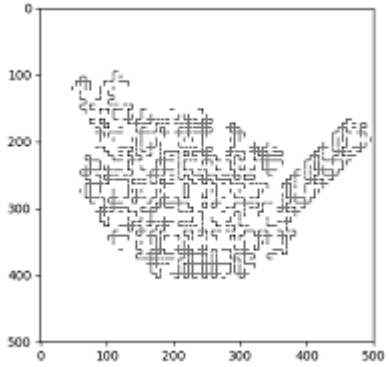

Figure 15e – A turmite with rule '181021…016110010' produces the same overall image, however the semi-inverted texture is getting less dense.

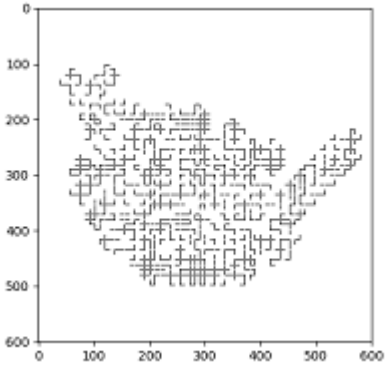

Figure 15f – A turmite with rule '181021…018110010' produces the same overall image. At this point the map looks almost like a grid instead of a filled in region.

5 Further Topics

5.1 Relation to Computing:

Langton's ant has been proven to be capable of being a 2D Turing machine; able to compute a series of logic and Boolean gates. Since the ant follows a very specific type of rules and board initially defined by the creator, each individual gate must a follow a different design. For example, AND and NOT gates may both output the same values, TRUE or FALSE, however AND requires two inputs whereas NOT only requires one[1]. Therefore, the board and trajectory of the two ants must be different. In all operators, there must be a specific ant entrance and exit and dedicated

areas for the input and output. In simplistic terms, during every activation, the ant will traverse its way towards the input area and depending on the input, either make its way toward the output area or toward the exit. After each activation, we can simply read the output area to determine if it has been altered by the ant. Figure 16 shows the boards of both AND and NOT gates, and the junctions that will determine the ant's trajectory.

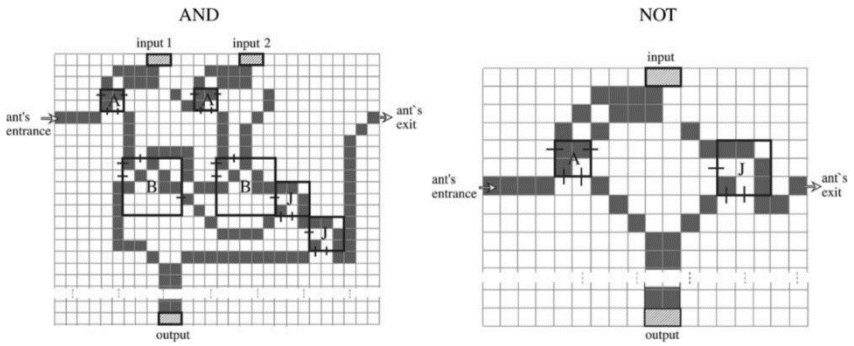

Figure 16: Showcases the boards for boolean operators AND and NOT. Each board must have its own unique design to direct to specific locations of the board (Image taken from Gajardo, A., Moreira, A., & Goles, E. (2000). Complexity of Langton's ant. *Elsevier*)

The squared bolded boxes labeled A, B or J, are junctions designed such that there are only several possible entrances and exits for the ant. After the first entrance, the ant would change the junction in such a way where the only possible exit is the one which it has not taken[2]. This allows the board to direct the ant in such a way where it is possible for the board to tell if the ant has processed the input or not.

6 Conclusion

Langton's ant is a cellular automaton that has inspired a great deal of motivation into how complex phenomenon can emerge from simple interactions with the environment[5]. The ants and turmites provide a visually striking indication of how seemingly random systems can have deterministic underpinnings. Much like chaotic systems, we see how the initial conditions can drastically alter the results of iterating our system. Despite the vast amount of work that people have put into understanding the algorithm, theoretical descriptions remain elusive. Further investigations into the prediction of long term behaviour remains a highly motivating field as it may help elucidate the emergence of physical phenomenon from a statistical mechanics point of view[5].

7 References

1. Langton, C. G. (1986). Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena*, 22(1-3), 120-149.
2. Gajardo, A., Moreira, A., & Goles, E. (2002). Complexity of Langton's ant. *Discrete Applied Mathematics*, *117*(1-3), 41-50.
3. Moreira, A., Gajardo, A., & Goles, E. (2001). Dynamical behavior and complexity of Langton's ant. *Complexity*, 6(4), 46-52.
4. Beuret, O., & Tomassini, M. (1997, July). Behaviour of multiple generalized langton's ants. *In Artificial Life V* (pp. 45-50).
5. Hamann, H., Schmickl, T., & Crailsheim, K. (2011, April). Thermodynamics of emergence: Langton's ant meets Boltzmann. In Artificial Life (ALIFE), *2011 IEEE Symposium on Artificial Life* (pp. 62-69). IEEE.

Appendix A – Generalized Langton's Ant Code:

```
import numpy as np
import re
import sys
import matplotlib.pyplot as plt
import matplotlib as mpl

dimen_x = 100
dimen_y = 100
maxiter = 2000000
num_ants = 1
face_direction = [0]
rules = ['rn']
antstate = 0
xpos = [int(dimen_x/2)]
ypos = [int(dimen_y/2)]
nstates = len(rules[0])
geometry = 'finite'
antstates = len(rules) - 1

def checkrule(string, search=re.compile(r'[^lnru]').search):

    return not bool(search(string))


def main(argv):

    for rule in rules:
        if not checkrule(rule):
            sys.exit('Invalid rule. Exiting.')

    # create grid and ants
    grid = Grid(dimen_x, dimen_y, geometry)
    ants = []
    for i in range(num_ants):
        ants.append(Ant(face_direction[i], xpos[i], ypos[i], rules, nstates))

    for i in np.arange(maxiter):
        # loop over ants (in case there's more than one)
        grid, ants = update(grid, ants, i, antstates)

    grid.final_plot(ants, maxiter-1)


def update(grid, ants, i, antstates):

    for ant in ants:
        try:
            grid.board= ant.move(grid.board, ant.state, ant.rules)
            if ant.state == antstates:
                ant.state = 0
            else:
                ant.state = ant.state + 1
        except:
            # general error handling: just quit, write debug msg
            grid.final_plot(ants, i)
```

```
            sys.exit("Something weird is going on")

        # check geometry to make sure we didn't fall off a cliff, quit if we did
        # for other topologies, apply boundary conditions
        if not grid.check_geometry(ant.position):
            # end the simulation if the ant hits the wall
            grid.final_plot(ants, i)
            sys.exit("Ant fell off the map at timestep = %d!" % i)

    return grid, ants


class Grid:

    def __init__(self, dimen_x, dimen_y, geometry):

        self.dimen = (dimen_x, dimen_y)
        self.geometry = geometry
        self.board = np.zeros((self.dimen[0], self.dimen[1]), dtype=np.int)

    def final_plot(self, ants, step):

        # plot the board state and ants
        # use a mask to make the ant array transparent and overlay only
        # the ants' positions onto the final result
        y = np.zeros((self.dimen[0], self.dimen[1]))
        for ant in ants:
            y[ant.position[0], ant.position[1]] = 1
        y = np.ma.masked_where(y == 0, y)

        # use imshow to print matrix elements using gray colormap. Ants are red.
        plt.imshow(self.board, cmap=plt.get_cmap('plasma_r'), interpolation='none')
        plt.imshow(y, cmap=mpl.cm.jet_r, interpolation='none')
        plt.savefig(rules[0]+'-'+str(num_ants)+'ants'+'-'+str(step+1)+'steps'+'.png')
        plt.show()

    def check_geometry(self, antpos):

        # return true if in valid geometry, flase if ant has fallen off the map
        # also, for non-finite, but bounded geometries, adjust ant position
        check = True
        if self.geometry == 'finite' and (antpos[0] < 0 or antpos[0] > self.dimen[0] or
                                          antpos[1] < 0 or antpos[1] > self.dimen[0]):
            check = False

        return check

class Ant:

    """
    Facing Direction:
         Up             [0,-1]  1
    Left    Right  2 [-1,0]  [1,0]  0
        Down          3  [0,1]
    dirs = [[1,0],[0,-1],[-1,0],[0,1]]
    index of dirs is the face_direction
    Right turn applies cyclic shift in negative direction
    Left turn applies cyclic shift in positive direction
```

```
    """

    def __init__(self, face_direction, xpos, ypos, rules, nstates):

        self.face_direction = face_direction
        self.position = [xpos, ypos]
        self.rules = rules
        self.nstates = nstates
        self.state = 0
        self.possiblefacings = ((1, 0), (0, -1), (-1, 0), (0, 1))
        self.geometry = geometry

    def move(self, board, antstate, rules):
        rule = self.rules[antstate]
        # get state of board and current direction
        state = board[self.position[0], self.position[1]]
        directive = rule[state]

        # change the ant's direction
        self.face_direction = self.cycle_dir(directive)

        # cyclically increment the state of the board
        board[self.position[0], self.position[1]] = (state + 1) % self.nstates

        # apply motion based on new direction
        self.position[0] = self.position[0] +
self.possiblefacings[self.face_direction][0]
        self.position[1] = self.position[1] +
self.possiblefacings[self.face_direction][1]

        return board

    def cycle_dir(self, directive):
        dir_r = (self.face_direction - 1) % len(self.possiblefacings)
        dir_l = (self.face_direction + 1) % len(self.possiblefacings)
        dir_u = (self.face_direction + 2) % len(self.possiblefacings)
        dir_n = self.face_direction

        new_face_direction = None
        # perform a cyclic permutation on the possible facing
        # directions with respect to the movement directive
        if directive == 'r':
            new_face_direction = dir_r
        elif directive == 'l':
            new_face_direction = dir_l
        elif directive == 'u':
            new_face_direction = dir_u
        elif directive == 'n':
            new_face_direction = dir_n

        return new_face_direction


#pretend there's command-line arguments for now
if __name__ == "__main__":
    main(sys.argv[1:])
```

Appendix B – Turmite Code:

```
import numpy as np
import re
import sys
import matplotlib.pyplot as plt
import matplotlib as mpl

#turmites

dimen_x = 200
dimen_y = 200
maxiter = 20000
num_ants = 1
face_direction = [0]
rule = '120081111010'
xpos = [int(dimen_x/2)]
ypos = [int(dimen_y/2)]
nstates = len(rule)/6
ncolours = 2
geometry = 'finite'


def splitrule(rule):
    rules = np.chararray((nstates,ncolours),itemsize=3)
    rules_lst = []
    k=0
    n_rules = len(rule)/3


    for i in range(n_rules):
        rules_lst.append(rule[i*3:3*(i+1)])

    for i in range(nstates):
        for j in range(ncolours):
            if (i > 0 and k <max((i,j))):
                k = k+1

            rules[i][j] = str(rule[3*(i+j+k):3*(i+j+k+1)])

    return rules

rules = splitrule(rule)

def main(argv):

    # create grid and ants
    grid = Grid(dimen_x, dimen_y, geometry)
    ants = []
    for i in range(num_ants):
        ants.append(Ant(face_direction[i], xpos[i], ypos[i], rules, nstates))

    for i in np.arange(maxiter):
        # loop over ants (in case there's more than one)
        grid, ants = update(grid, ants, i, rules)

    grid.final_plot(ants, maxiter-1)
```

```
def update(grid, ants, i, rules):
    for ant in ants:
        try:
            grid.board, ant.state= ant.move(grid.board, ant.state, rules)

        except:
            # general error handling: just quit, write debug msg
            grid.final_plot(ants, i)
            sys.exit("Something weird is going on")

        # check geometry to make sure we didn't fall off a cliff, quit if we did
        # for other topologies, apply boundary conditions
        if not grid.check_geometry(ant.position):
            # end the simulation if the ant hits the wall
            grid.final_plot(ants, i)
            sys.exit("Ant fell off the map at timestep = %d!" % i)

    return grid, ants


class Grid:

    def __init__(self, dimen_x, dimen_y, geometry):

        self.dimen = (dimen_x, dimen_y)
        self.geometry = geometry
        self.board = np.zeros((self.dimen[0], self.dimen[1]), dtype=np.int)

    def final_plot(self, ants, step):

        # plot the board state and ants
        # use a mask to make the ant array transparent and overlay only
        # the ants' positions onto the final result
        y = np.zeros((self.dimen[0], self.dimen[1]))
        for ant in ants:
            y[ant.position[0], ant.position[1]] = 1
        y = np.ma.masked_where(y == 0, y)

        # use imshow to print matrix elements using gray colormap. Ants are red.
        plt.imshow(self.board, cmap=plt.get_cmap('gray_r'), interpolation='none')
        plt.imshow(y, cmap=mpl.cm.jet_r, interpolation='none')
        plt.savefig(rule+'-'+str(num_ants)+'ants'+'-'+str(step+1)+'steps'+'.png')
        plt.show()

    def check_geometry(self, antpos):

        # return true if in valid geometry, flase if ant has fallen off the map
        # also, for non-finite, but bounded geometries, adjust ant position
        check = True
        if self.geometry == 'finite' and (antpos[0] < 0 or antpos[0] > self.dimen[0] or
                                          antpos[1] < 0 or antpos[1] > self.dimen[0]):
            check = False

        return check

class Ant:
```

```
    """
    Facing Direction:
         Up              [0,-1]  1
    Left    Right  2 [-1,0]  [1,0]  0
        Down          3  [0,1]
    dirs = [[1,0],[0,-1],[-1,0],[0,1]]
    index of dirs is the face_direction
    Right turn applies cyclic shift in negative direction
    Left turn applies cyclic shift in positive direction
    """

    def __init__(self, face_direction, xpos, ypos, rules, nstates):

        self.face_direction = face_direction
        self.position = [xpos, ypos]
        self.rules = rules
        self.nstates = nstates
        self.state = 0
        self.possiblefacings = ((1, 0), (0, -1), (-1, 0), (0, 1))
        self.geometry = geometry

    def move(self, board, ant_state, rules):
        # get state of board and current direction
        state = board[self.position[0], self.position[1]]
        rule = rules[state][int(ant_state)]
        directive = rule[1]

        # change the ant's direction
        self.face_direction = self.cycle_dir(directive)

        # cchange state of board, and ant
        board[self.position[0], self.position[1]] = rule[0]
        #print(rule[0])
        ant_state = rule[2]
        #print(rule[0], rule[2])
        # apply motion based on new direction
        self.position[0] = self.position[0] +
self.possiblefacings[self.face_direction][0]
        self.position[1] = self.position[1] +
self.possiblefacings[self.face_direction][1]

        return board, ant_state

    def cycle_dir(self, directive):
        dir_r = (self.face_direction - 1) % len(self.possiblefacings)
        dir_l = (self.face_direction + 1) % len(self.possiblefacings)
        dir_u = (self.face_direction + 2) % len(self.possiblefacings)
        dir_n = self.face_direction

        new_face_direction = None
        # perform a cyclic permutation on the possible facing
        # directions with respect to the movement directive
        if directive == '1':
            new_face_direction = dir_n
        elif directive == '2':
            new_face_direction = dir_r
        elif directive == '4':
```

```
            new_face_direction = dir_u
        elif directive == '8':
            new_face_direction = dir_l

        return new_face_direction


#pretend there's command-line arguments for now
if __name__ == "__main__":
    main(sys.argv[1:])
```