Contents

1 A short Introduction to NetLogo

* a summary of programming guide and interface tab guide

Agents are beings that can follow instructions in the NetLogo **world**, which is two dimensional and divided into grids of **patches** that can be seen as a piece of ground. There are four types of agents: **Turtles** are objects that move around in the world over **patches**. **Links** are agents that connect two turtles. The **observer** refers to people looking at the world of turtles and patches – like us.

In NetLogo, **Commands** and **reporters** are instructions that tell agents what to do. A command is an action for an agent to carry out, while a reporter "reports" back a value that it computes. The commands and reporters that are built into NetLogo are called **primitives**, and the ones we define are called **procedures**.

The interface items we will be using are **buttons** and **choosers**, and they also follow a section of instructions that we define. There are two types of buttons – once or forever. When clicking on a once button, it executes its instructions once. The forever button executes the instructions again and again until it is clicked again to stop. A chooser allows us to choose a value from a list of choices. In the interface toolbar, we can control the speed at which the program runs, or to change the model settings.

Patches have **coordinates**, and we can find a patch's coordinates *pxcor*, which increases as you move to the right and *pycor*, which increases as you move up. We can determine the total number of patches by setting *min-pxcor*, *max-pxcor*, *min-pycor*, and *max-pycor*. Turtles also have coordinates *xcor* and *ycor*. While a patch's coordinates are integers, a turtle's coordinates can have decimals. In addition, We can set the color for turtles by setting the *color* built-in variable, or *pcolor* for patches.

2 Our model in NetLogo

2.1 Brief introduction

In our model, we have three starting point choices representing what our fish looks like before its pattern fully forms including: two rows of yellow, two columns of black, yellow and black mixed.

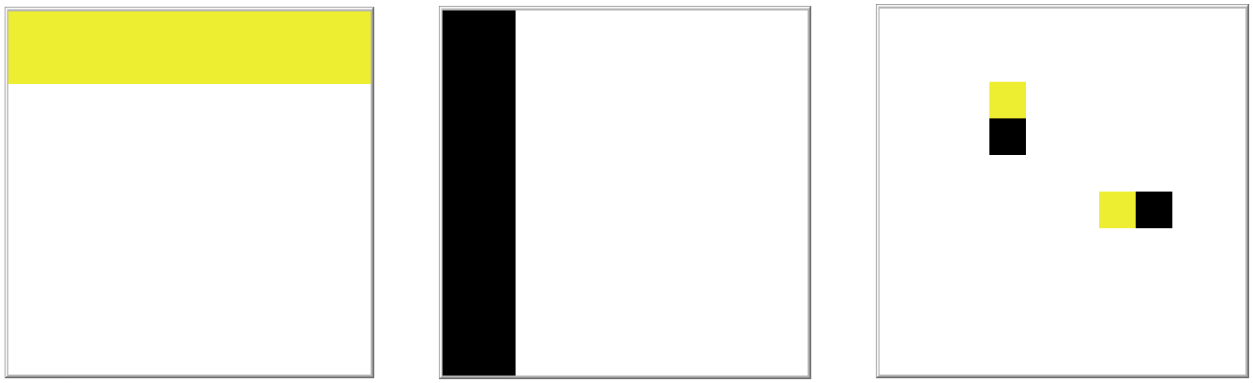


Figure 1: Starting choices: top-yellow, side-black, random

In addition, we have four different rules: one side different, the rest the same(abbreviated as ODRS); all sides the same(ASTS); all sides different(ASD); no rules(rand). Suppose we are looking at the color of one cell, then based on the rule we choose, we set the color for its neighboring four cells. We show an example of the first three rules below. For random, the colors of neighboring cells do not depend on the center cell that we are looking at.

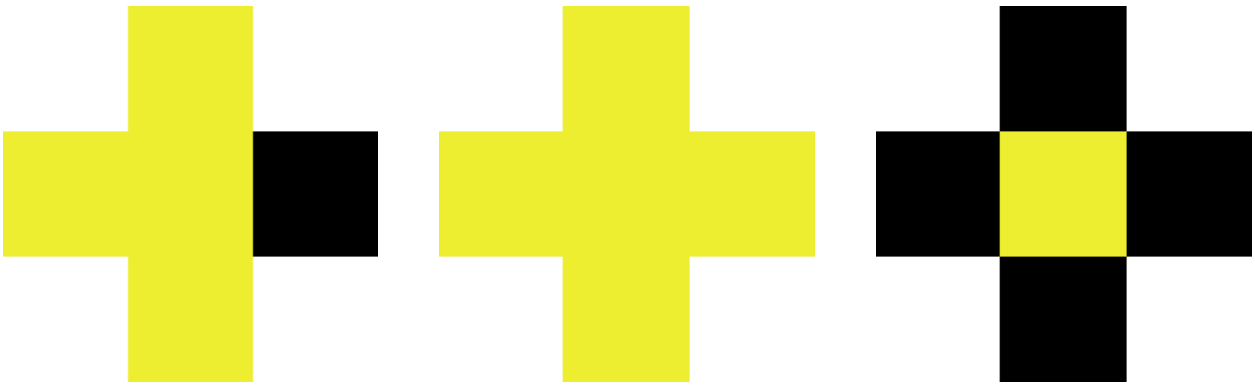


Figure 2: Rules: ODRS, ASTS, ASD

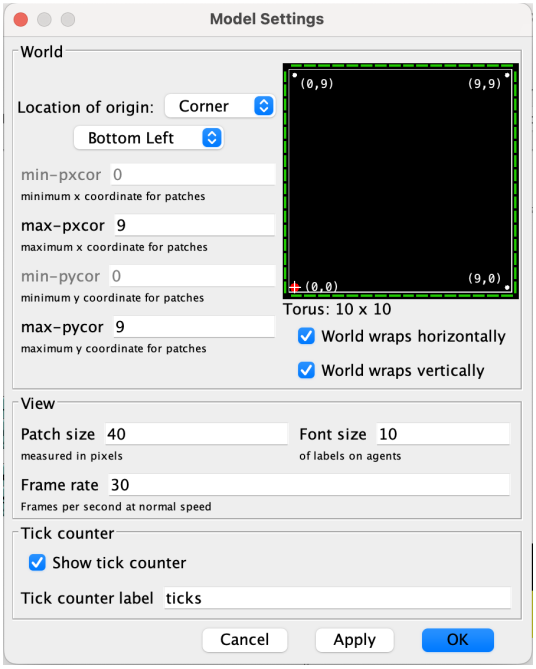
2.2 Tutorial of the model

2.2.1 Setting Up User Interface

We can create a new model by selecting "New" from the File menu. To access the model settings, click "settings" on the top right.

Here, we are setting location of origin to "Corner"-"Bottom Left" for convenience. To modify number of cells we want to see in our world represented by patches, we can choose our max-pxcor and max-pycor. Here, to set a 10x10 world, we set the max-pxcor and max-pycor to 9. The cell size can be set by adjusting "Patch size" under View.

We also want to uncheck the two boxes for "World wraps horizontally" and "World wraps vertically."



We want to add buttons to interact with the model. The most basic buttons we use to initialize settings and start the simulation are "setup" and "go." We would want to click the "setup" button once to have everything initialized and be ready to go, then once we click the "go" button, the simulation should start and the commands should run over and over. In addition, we want to be able to choose the starting points and rules, so we will also need two choosers besides buttons. First, to create a "setup" button:

1. Select "Button" in the drop-down menu;
2. Click the "Add" icon;
3. Click in the empty white area near the world window;
4. Type "setup" in Commands and click "ok".

What we have typed in Commands box is the procedure "setup" that we want to execute. We can define this procedure in the Code tab. For now, since the "setup" procedure is not defined yet, the button turns red.

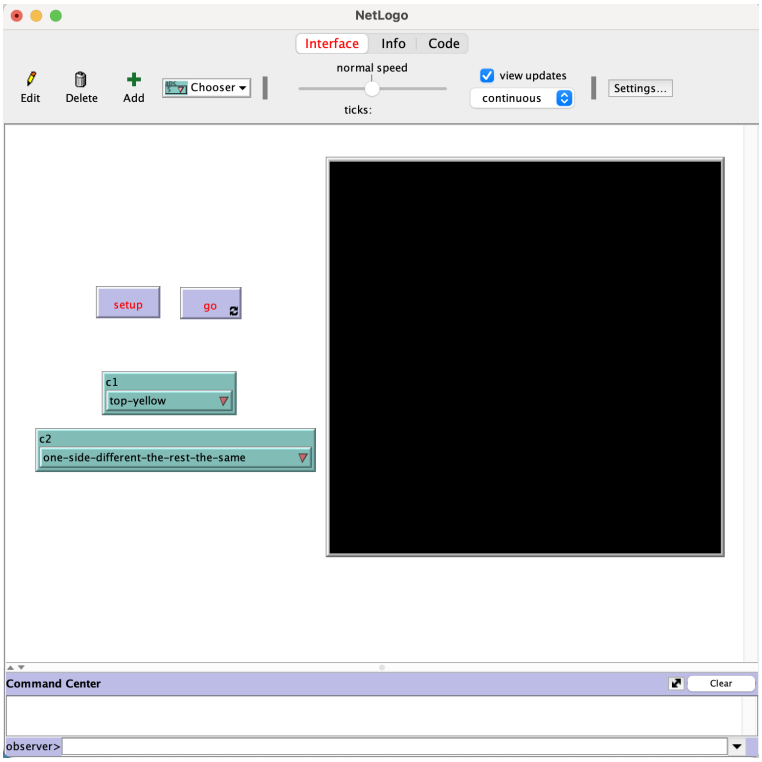
Similarly, we create a "go" button typing "go" in Commands. Since we would want the "go" procedure to run over and over, we want to check the "Forever" checkbox.

To create a chooser for starting points:

1. Select "Chooser" in the drop-down menu;
2. Click the "Add" icon;
3. Click in the empty white area near the world window;
4. Type "Starting-Choice" in Global variable;
5. Type "top-yellow", "side-black", and "random" in Commands, each in a new line, and click "ok".

We can create another chooser for rules following the same steps. Here, we type "Rule-Choice" in Global variable and "one-side-different-the-rest-the-same", "all-sides-the-same", "all-sides-different", and "no-rules" in Commands.

After creating buttons and choosers, this is what we should see:



2.2.2 Coding

We still need to define the procedures related to the buttons, which are "setup" and "go." A procedure is a set of commands that begins with to and ends with end.

What do we need for the setup procedure? We want to simulate how patches are colored following a rule of our choice, so we will need to set up patches. We would like to see the patches being colored one at a time, so we will use a turtle as a marker to tell us where we are looking at. Therefore, there are two things we need to do: set up a turtle and set up patches.

```
1  ; set up the simulation environment
2  to setup
3    clear-all ; clears everything including turtles, patches, links, etc
4    setup-turtle ; calls the procedure to set up the turtle
5    setup-patches ; calls the procedure to set up the patches
6  end
```

Here, the setup-turtle(line 4) and setup-patches(line 5) procedures are not defined yet, and we call them helper procedures. They can help us keep our code clean and readable, and we will be using them a lot in our program.

Let's first consider the overall logic for our setup-turtle procedure. We need to first create a turtle, then give it specific rules to follow based on the starting point of our choice. Thus, we will need to introduce conditionals that handle decisions. We can use a series of *if* commands, which executes a statement if a condition specified is true. We can also use another conditional command, which is the *ifelse* command. The difference between this command and the if command is that there is an additional block that executes if the condition specified is not true.

In other words, this is the structure for the if command:

if (condition) [statement]

When condition is true, the statement will be executed. Condition here is a boolean variable, meaning it can be either true or false.

This is the structure for the ifelse command:

ifelse (condition) [statement 1] [statement 2]

When condition is true, statement 1 will be executed. When condition is false, statement 2 will be executed.

The code for setup-turtle procedure is attached below:

```
14  ; set up turtle
15  to setup-turtle
16    create-turtles 1[ ; creates one turtle
17      set hidden? true ; hides the turtle, because we don't actually need to see it
18      ifelse Starting-Choice = "top-yellow" ; checks if the initial condition user chooses is "top-yellow"
19        [setxy min-pxcor max-pycor - 1 ; if so, places the turtle on the leftmost position of the second row
20         set heading 90] ; sets the turtle to face east
21        [ifelse Starting-Choice = "side-black" ; checks if the initial condition user chooses is "side-black"
22          [setxy min-pxcor + 1 max-pycor ; if so, places the turtle on the top of the second column
23           set heading 180]  ; sets the turtle to face south
24          ; the line below corresponds to the starting choice of "random" since we are using an ifelse statement
25          [move-to one-of patches with [pcolor != white]]]] ; moves the turtle to a random patch that is not white
26  end
```

Here, the turtle is like an indicator of which patch we are looking at. We know that regardless of the starting points we are choosing, we always want to create a turtle(line 16) and set it to hidden(line 17) because we don't need to actually see it. We only need to see the color change of the patch. Then, depending on the starting conditions we choose, we need to set the initial position of the turtle, then set the direction in which it moves. When the value of "Starting-Choice" chooser is "top-yellow"(line 18), it means that our turtle should start on the leftmost patch of the second row(line 19), and we choose to have the turtle move to the right by setting heading to 90(line 20). Similarly, we set the initial position of the turtle for starting choice "side-black." For the random starting point, since we do not know the locations of colored patches, we want to move the turtle to a patch that is not white so that we have something to work with in the first step(line 25).

Besides setting up turtles, we also need to set up patches. The code for setup-patches procedure is attached below:

```
28  ; set up patches
29  to setup-patches
30      ask patches[set pcolor white] ; sets all patches to white (think of this as making a white canvas!)
31      ifelse Starting-Choice = "top-yellow" ; checks if the initial condition user chooses is "top-yellow"
32        [ask patches with[pycor > max-pycor - 2][set pcolor yellow]] ; if so, set the top two rows of patches to yellow
33        [ifelse Starting-Choice = "side-black" ; checks if the initial condition user chooses is "side-black"
34          [ask patches with[pxcor < 2][set pcolor black]] ; if so, set the leftmost two columns of patches to black
35          [setup-patches-random]] ; if the starting choice is "random", we call the procedure to set up patches!
36  end
```

We would want to start with all patches set to white(line 30), then similarly, depending on the starting choice, we would set colors of the patches accordingly. Note that our world is set to be like the first quadrant of the coordinate system, so the pycor increases as we look at patches from bottom to top, and the pxcor increases as we look at patches from left to right. For example, if we have "top-yellow" as the starting choice(line 31), we want to set the top two rows of patches to yellow(line 32), and similarly for "side-black"(line 33). For "random," the situation is slightly more complicated, and we want to use a helper procedure called "setup-patches-random"(line 35).

Code is attached below:

```
38  ; set up patches - random
39  to setup-patches-random
40    ; Selects a random patch that is not on the edge
41    ask one-of patches with [pxcor != min-pxcor and pxcor != max-pxcor and
42                             pycor != min-pycor and pycor != max-pycor][
43      set pcolor yellow ; sets the selected patch's color to yellow
44      let nl neighbors4 ; get the four neighboring patches (north, south, east, west)
45      if any? nl ; checks to see if there are any neighbors
46        [let rand one-of nl ; chooses one random neighbor
47          ask rand [set pcolor black]]] ; sets the chosen neighbor's color to black
48
49    ; then the following section does the same thing! (we want in total of 4 patches colored)
50    ask one-of patches with [pcolor = white and ; we need to exclude the patches that already have a color
51                             pxcor != min-pxcor and pxcor != max-pxcor and
52                             pycor != min-pycor and pycor != max-pycor][
53      set pcolor yellow
54      let nl neighbors4
55      if any? nl
56        [let rand one-of nl
57          ask rand [set pcolor black]]]
58  end
```

We want to start with two sets of one yellow patch and one black patch combination where overlap is allowed. We want to randomly select a patch that is not on the edges of our world(line 41-42), then set its color to yellow (line 43). Then, we can access its neighbors by using the *neighbors4* primitive(line 44), which reports an agentset named "nl" that contains the four(or less if we are currently on the edge) patches surrounding the patch we are currently looking at, abbreviated as current patch, and let's call those four patches "neighbors" moving forward, using north, west, south, and east to denote their relative positions to the current patch. Then, we randomly choose one of the neighbors and set the color to black(line 46-47). Similarly, we choose the other set. The only thing different is that we do not want to choose the exact two patches that we have previously chosen, so we would pick one with white color, then again set color of one of its neighbors(line 50-57).

Now we are done with the setup procedure. We still need to define the "go" procedure:

```
8   ; what to do in each step of the simulation
9   to go
10    if (all? patches [pcolor != white]) [stop] ; stops the simulation if all patches are assigned a color(either yellow or black)
11    go-turtle ; calls the procedure to move the turtle
12  end
```

We want this procedure to execute over and over until all the patches are either yellow or black, and this is determined using an if command(line 10). We are checking if the color of all the patches are not white. From this point forward, a patch that has not been assigned yellow or black will be called an empty patch. Then, just as how we used the helper procedures for the setup procedure, we will use a helper procedure "go-turtle" (line 11).

Here's the code for the "go-turtle" procedure:

```
60  ; helper procedure to first set patch color, then move the turtle
61  to go-turtle
62    ask turtles[ ; asks the turtle to perform the following actions
63      set-patch-color ; calls the procedure to set the color of the current patch
64      ifelse Starting-Choice = "top-yellow" ; checks if the initial condition user chooses is "top-yellow"
65        ; this section corresponds to if the user chooses "top-yellow"
66        [ifelse ycor > min-pycor AND xcor < max-pxcor ; checks if the turtle is not at the right edge or the bottom edge
67          [fd 1] ; if satisfies the condition, moves the turtle forward by 1
68          [setxy 0 (ycor - 1)]] ; otherwise, moves the turtle to the leftmost position and one patch down
69        ; end of this section
70        [ifelse Starting-Choice = "side-black" ; checks if the initial condition user chooses is "side-black"
71          ; if the user chooses "side-black"
72          [ifelse ycor > min-pycor AND xcor < max-pxcor ; checks if the turtle is not at the right edge or the bottom edge
73            [fd 1] ; if satisfies the condition, moves the turtle forward by 1
74            [setxy xcor + 1 max-pycor]] ; otherwise, moves the turtle one patch to the right and to the topmost position
75          ; end of this section
76          ; if the user does not choose "top-yellow" or "side-black" -- then we are left with the last choice "random"
77          [move-to one-of patches with [pcolor != white] ; moves the turtle to a random patch that is not white
78            let test 0 ; initializes a variable called test to be 0
79            while [test < 1][ ; enters a while loop that continues until var test is at least 1
80              if (all? patches [pcolor != white]) [stop] ; stops if all patches are assigned a color that is not white
81              move-to one-of patches with [pcolor != white] ; moves to another random patch that is not white
82              set test 0 ; resets test to 0
83              ; the line below checks if there are any white neighbors and sets test to 1 if true
84              if any? neighbors4 with [pcolor = white] [set test 1]]]]]
85  end
```

Regardless of the rule we choose to use, there are two things we need to do for each step: first we will need to set the color for neighbors of our current patch, and second we need to move our turtle to a different patch so that we can look at other empty patches. The first step is dealt with by the helper procedure "set-patch-color"(line 63), which we will explain in a bit. The second part where we move our turtle is again depending on the rule we are using. If we are choosing the "top-yellow" starting condition(line 64), we can simply move the turtle one patch to the right(line 67) until we reach the end of the row(condition checked by line 66), then move the turtle to the leftmost position on the next row(line 68). Similar logic applies to the "side-black" option(line 70), except that the turtle is moving downward(line 73).

What if we are choosing the "random" option? Since now we don't know where we have colored patches, we will again need to move to one of the patches that has been assigned a color other than white. This will do the work we need, but there is one optional choice we can make to improve the performance of our code. As our simulation is almost done, the empty patches that yet need to be assigned a color becomes less and less. Therefore, there will be more and more patches that are not white, and it takes time for our program to run the set-patch-color helper method every step. We can use one additional test to see if the patch we randomly picked has empty neighbors(line 77-84), and only run the set-patch-color helper method if so. In this way, we will be able to save some time especially if we have a huge amount of patches.

Now we are back to the "set-patch-color" procedure. Before actually writing the code, let's sort out the logic. We pick colors for the neighbors by looking at the current patch, then assign either yellow or black according to our chosen rule. We will want to know the amount of empty patch among the four neighbors, whether our current patch is on one of the four corners(called corner patches) or on the four sides of the world but not one of the corners(called side patches), and if current patch is a side patch, which of the four sides it is on (left, right,

up, down). We want to know the color of our current patch and for convenience its opposite color (i.e: if current patch is yellow, its opposite color would be black). In addition, if the neighbors are colored, we want to record their colors as well. If those information become easily accessible, it will be a lot easier for us to set colors of patches. Therefore, we can use a reporter, which is a procedure that tells us something that we can reuse whenever it is needed.

The code for this reporter will be divided and given in sections.

We can start by defining a list of variables we would like to use. Here, all the variables are listed for convenience. However, normally when we write code, we may not be able to come up with a list of everything that we need, but we can always come back and add more during the process.

```
; reporter to return a list of info needed later
to-report get-info
  ; we first initialize a list of variables we want to report later
  let cemp 0 ; counts the number of neighboring patches that are white
  let typepatch 0 ; records type of current patch; 0 middle, 1 side, 2 corner
  let sidepatch 0 ; records type of side patch: 1 - left, 2 - right, 3 - up, 4 - down

  ; will be stored in dcol []; Color code: 0 white, 1 yellow, 2 black, 3 DNE(for initializing purpose!)
  let so 3 ; south
  let ea 3 ; east
  let we 3 ; west
  let no 3 ; north
  let cu 0 ; current patch color, not really necessary here, but helpful for later
  let du 0 ; opposite color of current patch color (if cu = yellow, du = black and vice versa)

  ; initialize color for neighbors; our patch colors should only be white, yellow or black; red is for initialzing only
  let north red
  let south red
  let west red
  let east red
```

Finding the color for current patch and its opposite color is the easiest step and we can easily take care of that by using an ifelse statement. Code attached below:

```
  ; this section of code is for setting the color var
  let cur [pcolor] of patch pxcor pycor ; cur is the color value of the current patch
  ifelse cur = yellow ; if current patch is yellow
    [set cu 1 ; set cu to be the color code of yellow, which is 1
     set du 2] ; if cu is 1, set du to 2
    [set cu 2 ; set cu to be the color code of black, which is 2
     set du 1] ; if cu is 2, set du to 1
```

We can also determine whether the current patch is a corner patch or a side patch by looking at the current location of our turtle, and when it is indeed a side patch, we can record which side it is on. Let's look at the code for this section:

```
179   ; determine if the current patch is a side patch and which side it's on
180   ifelse xcor = min-pxcor AND ycor != min-pycor AND ycor != max-pycor ; if on the leftmost column and not on the top/bottom row
181     [set sidepatch 1] ; in our code, leftmost side patch is given numerical value 1
182     [ifelse xcor = max-pxcor AND ycor != min-pycor AND ycor != max-pycor ; if on the rightmost collumn and not on the top/bottom row
183       [set sidepatch 2] ; sets sidepatch type to 2 - right
184       [ifelse xcor != min-pxcor AND xcor != max-pxcor AND ycor = max-pycor ; if on the top row but not leftmost/rightmost column
185         [set sidepatch 3] ; sets sidepatch type to 3 - up
186         [if xcor != min-pxcor AND xcor != max-pxcor AND ycor = min-pycor ; if on the bottom row but not leftmost/rightmost column
187           [set sidepatch 4]]]] ; sets sidepatch type to 4 - down
188
189   ; determine the type of the current patch (0 - middle, 1 - side, 2 - corner)
190   ; side patch
191   if sidepatch != 0 [set typepatch 1] ; if the value of sidepatch variable is not 0, we have initialized it in the above section
192   ; corner patch
193   ifelse xcor = min-pxcor AND ycor = min-pycor ; if on the leftmost column and on the bottom row
194     [set typepatch 2] ; lower left corner
195     [ifelse xcor = min-pxcor AND ycor = max-pycor ; if on the leftmost column and on the top row
196       [set typepatch 2] ; upper left corner
197       [ifelse xcor = max-pxcor AND ycor = min-pycor ; if on the rightmost column and on the bottom row
198         [set typepatch 2] ; lower right corner
199         [if xcor = max-pxcor AND ycor = max-pycor ; if on the rightmost column and on the top row
200           [set typepatch 2]]]] ; upper right corner
```

Here, we first want to determine whether the current patch is a side patch. For a patch to be a side patch, either its x-coordinate or y-coordinate should be at the extremes but not both, as here we are excluding corner patches(conditions checked at line 180, 182, 184, 186). If so, we further specify the type of the sidepatch determined by the edge it's on(line 181, 183, 185, 187).

Then, we have another section for recording the type of the current patch. Recall that this piece of information is stored in the variable "typepatch," where 0 means a patch that is not a side or corner patch, 1 means a side patch, and 2 means a corner patch. We have initialized variable "sidepatch" to be 0, and we would only update this variable in the above section if our current patch is indeed a sidepatch. Thus, as long as sidepatch has a non-zero value, we know the current patch is a sidepatch and we can set variable "typepatch" to be 1(line 191). For a patch to be a corner patch, its coordinates should be at the extremes(either maximum or minimum) of both x and y coordinates in the Netlogo world(line 192-200). As the initial value for "typepatch" is set to be 0, it wouldn't be necessary for us to determine whether a patch is a middle patch – they are by default middle patches unless we update their type to be side or corner.

We would want to record colors for current patch's neighbors, but before going into details, we would like to make everything as easy as possible for ourselves. Let's consider this question: do we need to look at corner patches? If we think about it, the answer is no. For example, if our current patch is the leftmost corner patch, we can assign its color by looking at its east and south neighbors. As long as we take care of all patches other than corner patches, we will be able to assign colors to all the patches. Therefore, we can ignore all the corner patches for simplicity, and that leaves us with two possible situations: 1. our current patch is a side patch; 2. our current patch is a middle patch (all patches that are not side patches or corner patches).

Let's look at the first situation:

```
202   ; this section corresponds to storing info if our current patch is a sidepatch
203   if sidepatch != 0[
204     ; get pcolor values of the three neighbors
205     if sidepatch != 1[set west [pcolor] of patch (pxcor - 1) pycor] ; get west neighbor's color
206     if sidepatch != 2[set east [pcolor] of patch (pxcor + 1) pycor] ; get east neighbor's color
207     if sidepatch != 3[set north [pcolor] of patch pxcor (pycor + 1)] ; get north neighbor's color
208     if sidepatch != 4[set south [pcolor] of patch pxcor (pycor - 1)] ; get south neighbor's color
209
210     ; convert pcolor values to our vars and count #of empty patches
211     if sidepatch != 1[ ; if current patch is not on the leftmost column
212       ifelse west = white ; if the color of west patch is white
213         [set we 0 ; sets we to numerical color code 0
214          set cemp cemp + 1] ; increment count of empty(white) patches
215         [ifelse west = yellow ; if the color of west patch is yellow
216           [set we 1] ; sets we to numerical color code 1
217           [set we 2]]] ; if not white or yellow, color should be black -- so we set we to 2
218     if sidepatch != 2[ ; if current patch is not on the rightmost column
219       ifelse east = white ; if the color of east patch is white
220         [set ea 0 ; sets ea to numerical color code 0
221          set cemp cemp + 1] ; increment count of empty(white) patches
222         [ifelse east = yellow ; if the color of east patch is yellow
223           [set ea 1] ; sets ea to numerical color code 1
224           [set ea 2]]] ; sets ea to numerical color code 2
225     if sidepatch != 3[ ; if current patch is not on the top row
226       ifelse north = white ; if the color of north patch is white
227         [set no 0 ; sets no to numerical color code 0
228          set cemp cemp + 1] ; increment count of empty(white) patches
229         [ifelse north = yellow ; if the color of north patch is yellow
230           [set no 1] ; sets no to numerical color code 1
231           [set no 2]]] ; sets no to numerical color code 2
232     if sidepatch != 4[ ; if current patch is not on the bottom row
233       ifelse south = white ; if the color of south patch is white
234         [set so 0 ; sets so to numerical color code 0
235          set cemp cemp + 1] ; increment count of empty(white) patches
236         [ifelse south = yellow ; if the color of south patch is yellow
237           [set so 1] ; sets so to numerical color code 1
238           [set so 2]]]] ; sets so to numerical color code 2
```

Using the information we stored in the sidepatch variable from earlier, we can choose to record the color information in four variables: west, east, north, and south. We want to convert color information to numerical values so we can store and use our information more conveniently, as defined earlier in our get-info variable list. Note that here we are still recording the pcolor information rather than our numerical value for colors, meaning we are actually storing "white" rather than 0 which represents "white" in our color code. We will convert those pcolor information into our numerical value in the section below. We also have to be careful when recording the colors because any side patch has only three neighbors, and patches on different sides have different neighbors. For example, a left side patch does not have a west neighbor, and we would need to take care of it in our code. For example, if variable sidepatch has value 1, indicating we are at the leftmost edge and the current patch does not have a west neighbor, then we will skip line 205 and run line 206-208, recording the color information for its east, north, and south neighbors. We can do so by using some ifelse statements. In addition, if the color of a patch is white, we can add 1 to the cemp count, which is for counting how many empty patches we have(for example, line 214).

Recording the neighbors' colors for a middle patch is even easier:

```
240   ; middle patch - record all four sides
241   if xcor != min-pxcor AND xcor != max-pxcor AND ycor != min-pycor AND ycor != max-pycor[ ; not on any of the edges
242     ; we will have all four neighbors!
243     set north [pcolor] of patch pxcor (pycor + 1)
244     set south [pcolor] of patch pxcor (pycor - 1)
245     set west [pcolor] of patch (pxcor - 1) pycor
246     set east [pcolor] of patch (pxcor + 1) pycor
247
248     ; the idea of the following section is exactly the same as the above section for sidepatches
249     ; the only difference is that now we are storing information for all four neighbors instead of 3
250     ifelse north = white
251       [set no 0
252        set cemp cemp + 1]
253       [ifelse north = yellow
254         [set no 1]
255         [set no 2]]
256     ifelse south = white
257       [set so 0
258        set cemp cemp + 1]
259       [ifelse south = yellow
260         [set so 1]
261         [set so 2]]
262     ifelse west = white
263       [set we 0
264        set cemp cemp + 1]
265       [ifelse west = yellow
266         [set we 1]
267         [set we 2]]
268     ifelse east = white
269       [set ea 0
270        set cemp cemp + 1]
271       [ifelse east = yellow
272         [set ea 1]
273         [set ea 2]]]
```

Since we know we will have all four neighboring patches by our definition of a middle patch, we can simply record all the color information and store them numerically.

Now that we have obtained all the information we want to get from looking at a patch, we want to report them back, which is the last section of our reporter code:

```
275     let dcol (list no so we ea cu du) ; creates a list dcol for storing the color information
276     let l (list cemp typepatch sidepatch dcol) ; creates a list containing the cemp, typepatch, sidepatch, dcol[]
277     report l ; returns the information list l
278   end
```

We can store our variables in a list, and though not necessary, we can store all the color information into a list to improve readability(line 275). After creating our list containing all needed information(line 276), we can report our information list back(line 277).

We can attempt our set-patch-color procedure now that we have the information available. Again, the code for this procedure will be divided and given in sections.

We can first take the variables out of the information list for us to use them more easily, although it is also fine to access them directly from the list every time:

```
87  ; set-patch-color helper procedure
88  to set-patch-color
89    let l get-info ; gets information about the current patch and its surroundings from the reporter get-info
90    ; item 0 l means getting the first element from a list called l
91    ; the following variables are all defined in the get-info reporter
92    let cemp item 0 l ; sets cemp to be the number of white neighboring patches
93    let typepatch item 1 l ; sets typepatch to be one of the types of the current patch: middle, side, corner
94    let sidepatch item 2 l ; sets sidepatch to be one of the options: not a sidepatch, left, right, up, down
95    let dcol item 3 l ; dcol is a list to store colors of patches as numerical values
96    ; color code: 0 white, 1 yellow, 2 black, 3 DNE
97    let no item 0 dcol ; color of the north neighboring patch
98    let so item 1 dcol ; color of the south neighboring patch
99    let we item 2 dcol ; color of the west neighboring patch
100   let ea item 3 dcol ; color of the east neighboring patch
101   let cu item 4 dcol ; color of the current patch
102   let du item 5 dcol ; color of the opposite color of current patch (if cu is black, du would be yellow)
103   let rand random 2 + 1 ; randomly chooses either 1 or 2, then assign to var rand
```

Here, besides getting all the information from the list, we are declaring a new variable rand(line 103). We are going to use it for the "no-rules" choice. The built in procedure rand[number] will give us a random number from 0(included) to [number](not included), for example, rand 2 will return a random number chosen from either 0 or 1. Recall that in our color system, 0 means white, 1 means yellow, 2 means black, and 3 means DNE for initialization. So if we want to set a patch to either yellow or black randomly, we would need a random number between 1 and 2, and thus rand 2 plus 1 will give us our desired result.

Let's analyze how we should set the colors when following different rules:

```
105   let col cu ; initializes the color setter with the current patch's color
106   ifelse Rule-Choice = "one-side-different-the-rest-the-same" ; checks if the rule choice is "OSRS"
107   ; the following section corresponds to this rule:
108   ; the line below calls a reporter ifdiff that takes a parameter l containing the info we get before
109   [ifelse ifdiff l ; ifdiff determines whether one of the neighbors of the current patch has a different color
110     [set col cu] ; if ifdiff reporter reports true, sets col to cu
111     [ifelse cemp = 1 ; if ifdiff reporter reports false, we check if cemp is equal to 1
112       [set col du][stop]]] ; if cemp is 1, col is set to du; otherwise stops
113   ; end of this section
114   [ifelse Rule-Choice = "all-sides-different" ; checks if the rule choice is "ASD"
115     [set col du] ; if so, sets col to du
116     ; end of this section
117     [if Rule-Choice = "no-rules"[set col rand]]] ; checks if the rule choice is "no-rules", if so, set col to rand
```

If we have the "one-side-different-the-rest-the-same" rule, we would like to see if one of the neighbors of our current patch has a different color than current color. For example, if our current patch is yellow and one of its neighbors is already set to black, we know that all other neighbors should be set to yellow. This information should be easily obtained, and we can use a reporter called "ifdiff," which returns either true or false. We will look at the detailed implementation of this reporter, but for now, we only need to know that this reporter tells us whether one of the neighbors of our current patch has a different color(true if that's the case). We first call this reporter to determine whether the condition is true(line 109); if so, we can set our color setter col to cu, since we already have one patch that is of different color(line 110). If we don't have such a neighbor, we can then check if there's only one neighbor that is an empty patch(line 111). If so, we can set this patch to be of different color(line 112). If not, we can't make a decision for now, so we stop the procedure(line 112).

When we have the "all-sides-different" rule, we can simply assign the opposite color to all the neighbors, which is stored in variable du from our list of information(line 115). If we

choose "no-rules," then we can simply set all neighbors to rand(line 117). Similarly to the "all-sides-different" rule, when we have the "all-sides-the-same" rule, all neighbors should be set to the same color as that of our current patch. As variable "col" is initialized to be cu, we do not need to rewrite it in this section.

Now, let's look at how the "ifdiff" reporter is implemented:

```
280 ; this is the reporter for checking whether one of the neighbors of the current patch has a different color
281 to-report ifdiff [l] ; takes the information list l
282   ; a list of vars needed from l
283   let sidepatch item 2 l
284   let dcol item 3 l
285   let no item 0 dcol
286   let so item 1 dcol
287   let we item 2 dcol
288   let ea item 3 dcol
289   let cu item 4 dcol
290   ; indicator for the returning value
291   let diff false
292
293   ; checks each neighbor for a difference in color from the current patch, ignoring patches that are white
294   ; if any of the neighbors has a different color than the current patch, set diff to be true
295   if sidepatch != 1[if we != cu AND we != 0[set diff true]]
296   if sidepatch != 2[if ea != cu AND ea != 0[set diff true]]
297   if sidepatch != 3[if no != cu AND no != 0[set diff true]]
298   if sidepatch != 4[if so != cu AND so != 0[set diff true]]
299
300   ; returns true if there's a difference; false otherwise
301   report diff
302 end
```

The "ifdiff" reporter requires the information we have obtained from the "get-info" procedure. Here, we need one additional boolean variable diff that we want to report(line 291). The logic is quite simple: without loss of generality, suppose our current patch is a left patch. The only neighbor that it doesn't have is the west neighbor. Therefore, we would check for all other three neighbors. If one of them does not have the same color as our current patch and is not white, we can set the variable diff to true. If none of the neighbors satisfy this criteria, diff remains false.

Note that we have determined the value of our color setter "col", which means that we can then set the numerical color values for all neighboring patches that still needs to be assigned a color. This can be done easily:

```
119     ; if sidepatch != 1 checks if current patch is on the leftmost edge; similarly for 2 - rightmost, 3 - top, 4 - bottom
120     ; note that we will only skip the section when the condition does not satisfy!
121     if sidepatch != 1[if we = 0[set we col]] ; if current patch has a west neighbor and if we is 0(white), set we to col
122     if sidepatch != 2[if ea = 0[set ea col]] ; if current patch has a east neighbor and if ea is 0(white), set ea to col
123     if sidepatch != 3[if no = 0[set no col]] ; if current patch has a north neighbor and if no is 0(white), set no to col
124     if sidepatch != 4[if so = 0[set so col]] ; if current patch has a south neighbor and if so is 0(white), set so to col
```

Up until now, we are still working with the numerical values that represent actual color values. We want to record the numerical values of the colors we want to assign to the neighbors. We can use our normal four variables we, ea, no, so to store the numerical information.

Now, with all the numerical values stored in we, ea, no, so, we are ready to convert those numerical values to actual color variables and assign them to patches:

```
  ; recall that all our colors are stored as a numerical value instead of an actual color value! 0 white, 1 yellow, 2 black, 3 DNE
  ; this section is for actually assigning the color value to the patches
  if sidepatch != 1[ ; if not on the leftmost edge
    ifelse we = 1[ ; if the numerical value of we is 1
      ask patch (pxcor - 1) pycor [set pcolor yellow] ; assign the west patch color to yellow
    ][if we = 2 ; if the numerical value of we is 2
      [ask patch (pxcor - 1) pycor [set pcolor black]]]] ; assign the west patch color to black
  if sidepatch != 2[ ; if not on the rightmost edge
    ifelse ea = 1[ ; if the numerical value of ea is 1
      ask patch (pxcor + 1) pycor [set pcolor yellow] ; assign the east patch color to yellow
    ][if ea = 2 ; if the numerical value of ea is 2
      [ask patch (pxcor + 1) pycor [set pcolor black]]]] ; assign the east patch color to black
  if sidepatch != 3[ ; if not on the top edge
    ifelse no = 1[ ; if the numerical value of no is 1
      ask patch pxcor (pycor + 1) [set pcolor yellow] ; assign the north patch color to yellow
    ][if no = 2 ; if the numerical value of no is 2
      [ask patch pxcor (pycor + 1) [set pcolor black]]]] ; assign the north patch color to black
  if sidepatch != 4[ ; if not on the bottom edge
    ifelse so = 1[ ; if the numerical value of so is 1
      ask patch pxcor (pycor - 1) [set pcolor yellow] ; assign the south patch color to yellow
    ][if so = 2 ; if the numerical value of so is 2
      [ask patch pxcor (pycor - 1) [set pcolor black]]]] ; assign the south patch color to black
end
```

Since we already know which numerical value corresponds to which color, and the names storing those numerical values represent the positions of the neighbors relative to our current patch, we can easily convert them back to color values and coordinates.

We are done with all the coding. If we go back to the interface window, choose our starting choice and rule choice, the click set up and run, we should be able to see the pattern formation simulation.

3 Netlogo Model

Github page