NetLogo Tutorial Series:
Introduction to Diffusion-limited Aggregation

Nicholas Bennett
nickbenn@g-r-c.com

October 2015

Credits and licenses

"Sierpinski triangle.svg" by Beojan Stanislaus is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License [1].

"Koch curve.svg" by Fibonacci is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License [2].

"DLA Cluster.jpg" by Kevin R. Johnson is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License [3].

"Fractal Broccoli.jpg" by Jon Sullivan has been released into the public domain [4].

"Thorax Lung 3d (2).jpg" by AndreasHeinemann is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License [5].

All Mandelbrot and Julia set images rendered in XaoS, © 2008, Jan Hubicka and Thomas Marsh [6].

All programming screen captures are from NetLogo, © 1999-2015, Uri Wilensky [7].

NetLogo version and code

The code in this tutorial was written with NetLogo v5.2 and tested with NetLogo v5.2 and v5.2.1; all screen shots are taken with NetLogo v5.2. However, the code should run correctly as written with all NetLogo v5.x.

NetLogo code listings and fragments are displayed in **`DejaVu Sans Mono Bold 10.5pt`** type. Type colors match the colors used in the NetLogo code editor:

- **`Command primitives`**
- **`Reporter primitives and predefined agent variables`**
- **`Keywords`**
- **`Literal values and predefined symbolic constants`**
- **`Breeds, procedures, variables`**
- ***`Placeholders`***

Fractals

Self-similar patterns

A fractal is a structure or a set of points that shows non-trivial repeated patterns at all scales of interest. If the patterns are repeated identically at all scales, we say the fractal is *exactly self-similar*; example of such fractals include the Sierpinski triangle (Figure 1) and Koch curve (Figure 2): no matter how far we zoom in on either of these two, (in a mathematical sense, at least), we see the same pattern repeated identically.

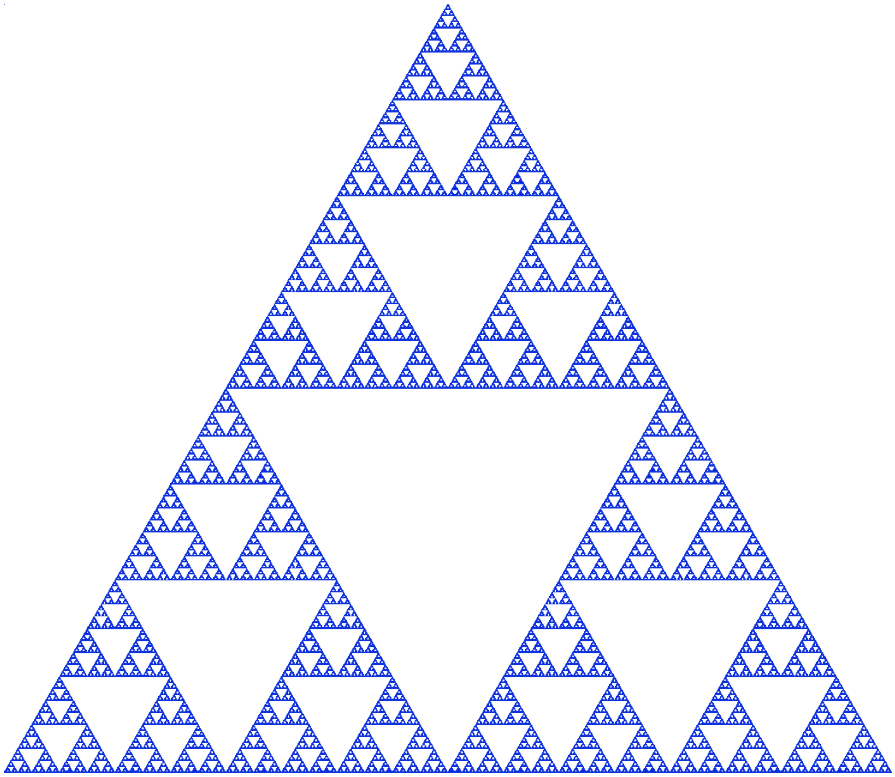

Figure 1 – Sierpinski triangle Error: Reference source not found.

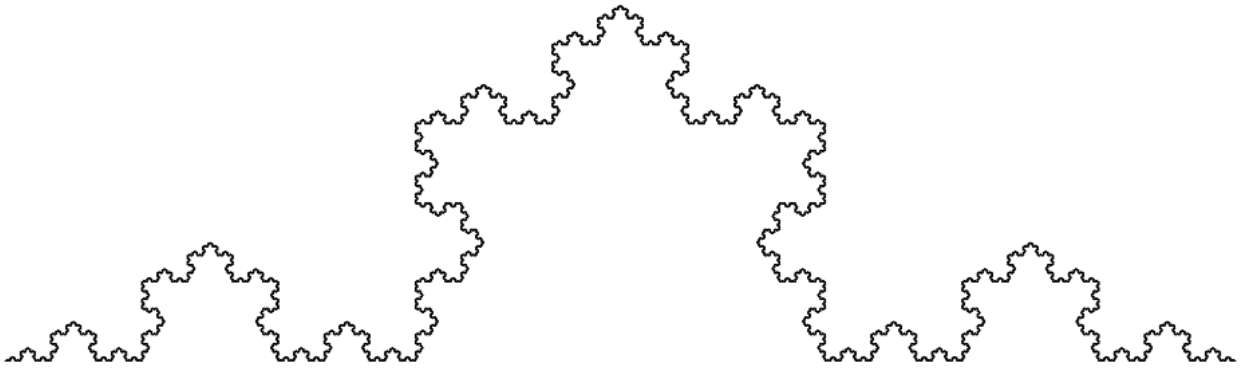

Figure 2 – Koch curve Error: Reference source not found.

Another class of fractals with identical self-similarity are some instances of Julia sets (closely related to the Mandelbrot set, below). For example, in Figure 3, we see what appears to be a rotated and recolored version of Figure 4. In fact, the former is a portion of the Julia set defined by the iterative function $z \rightarrow z^2+i$; the latter shows a very small portion of the former, magnified 100 times. In both, we see the same branching structures, with the same relative sizes and branching angles.

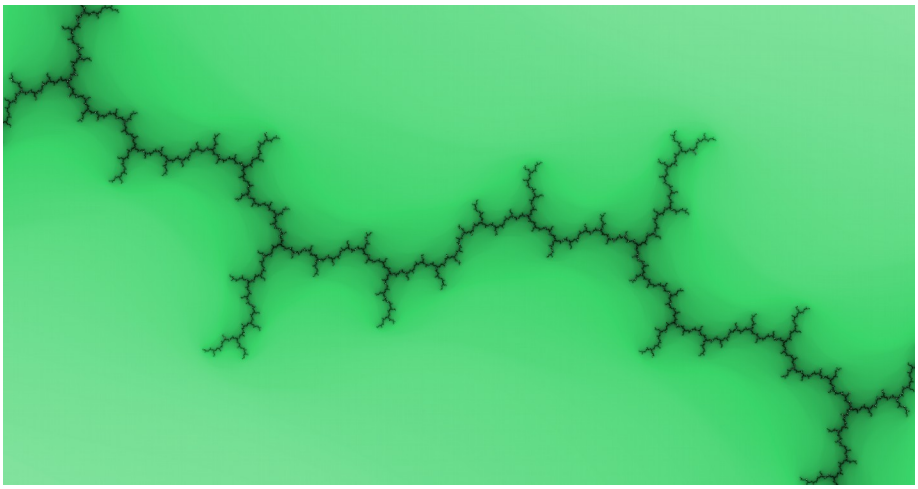

Figure 3 – Portion of Julia set with c = i [6].

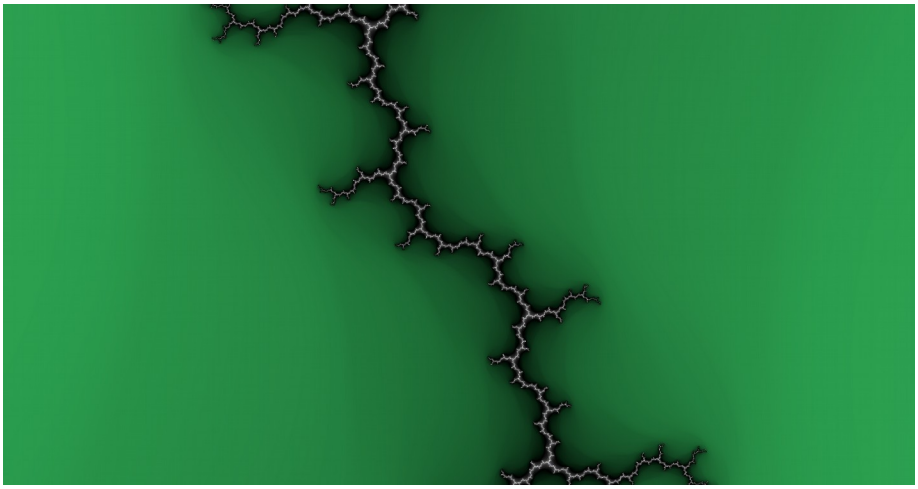

Figure 4 – Portion of Julia set with c = i, magnified 100X [6].

Some other fractals exhibit *quasi-self-similarity*, where the pattern is repeated at different scales, but with some distortion or degeneration. Many neighborhoods in the Mandelbrot set are self-similar in this fashion: in Figure 5 and Figure 6, we see similar – but not identical – bulb and branch structures when we magnify one portion of the Mandelbrot set by a factor of 100.

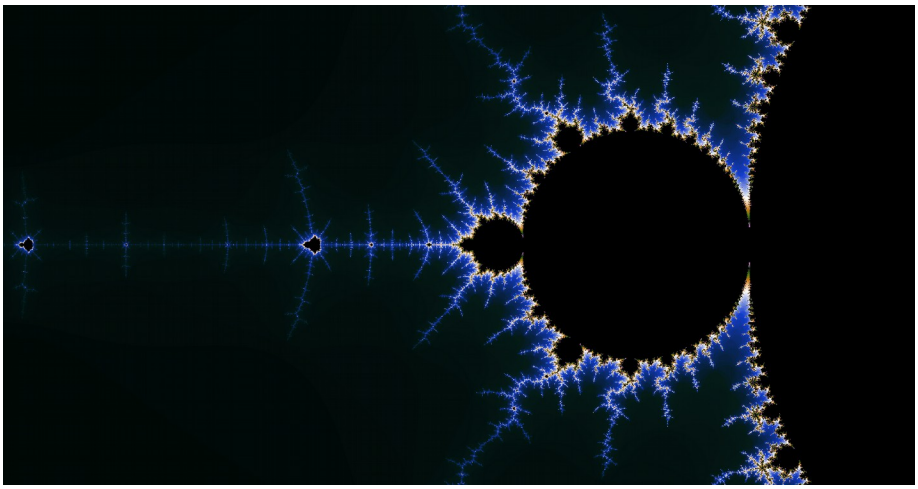

Figure 5 – Mandelbrot set centered at -1.401 – 0.001i., magnified 10X [6].

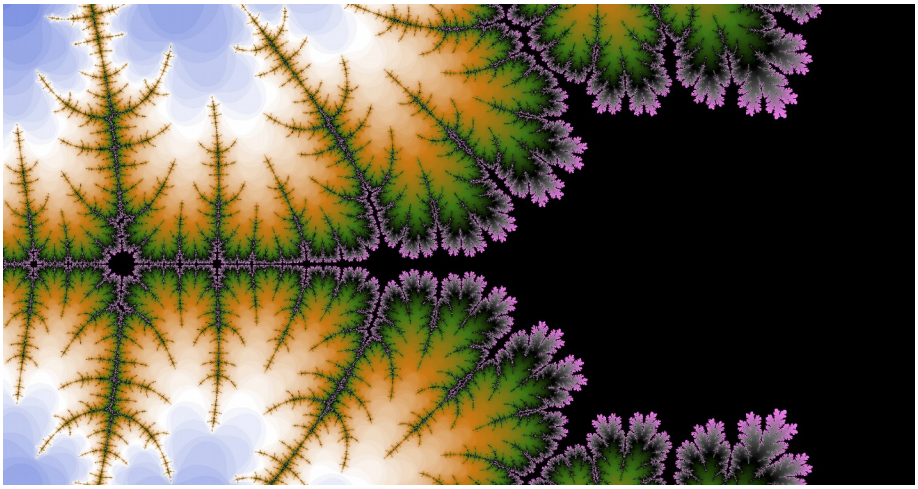

Figure 6 – Mandelbrot set with same center as Figure 5, magnified 1,000X [6].

Statistical self-similarity and random fractals

Theoretical fractals are usually expressed via iterative or recursive mathematical operations. Based on these, the identical or quasi self-similarity can be shown mathematically. However, fractals found in nature are generally *statistically self-similar*: while the patterns in these fractals are not repeated exactly, key statistical measures of those patterns are repeated (at least approximately) at all the scales of interest.

Many plants exhibit statistically self-similar patterns in their branches, leaves, and blossoms. A striking example of this is found in the Romanesco broccoli (Figure 7).

The human body, like that of most animals, has fractal structures as well – e.g. in the blood vessels, nervous system, and lungs (Figure 8). We'll come back (briefly) to the topic of lungs when we explore the concept of fractal dimension.

Figure 7 – Romanesco broccoli [4].

Figure 8 – Human lung from 3D CT scan [5].

When simulating natural fractal processes on a computer, we often include random numbers in the calculations that generate these fractals. This technique is quite often seen in movies and video games, where terrain, vegetation, and other features can be generated via random fractal algorithms. By design, these fractals will generally have the statistical self-similarity of the natural fractals they simulate.

Fractal dimension

Mathematically, fractals can be defined in virtually any number of dimensions. However, our common sense understanding of dimension and measurement is challenged by fractals. In particular, while common geometric measures like area and perimeter allow us to talk about non-fractal shapes in analytical and comparative terms, they are of much less use to us when it comes to fractals.

For example, consider the process of constructing the Sierpinski triangle: start with an equilateral triangle; dividing that triangle into 4 smaller, identical equilateral triangles; remove the center triangle of the 4; repeat the same process of subdivision and removal with the 3 smaller triangles that remain; repeat the same process indefinitely with the progressively smaller triangles that remain. We can see that each time we do this subdivision and removal, we decrease the total area enclosed by a factor of ¼; at the same time, we increase the perimeter (remember to include the perimeter of the edge exposed by removing the inner triangle) by a factor of ½. If we do this an infinite number of times, we end up with an area of zero, and an infinite perimeter! In fact, any theoretical fractal curve has an infinite length, even though it may be entirely contained within a finite area of the plane.

In 1967, several years before he coined the term "fractal", Benoit Mandelbrot published a paper describing some natural and theoretical curves (e.g. the coastline of Britain, the Koch curve) in terms of self-similarity and "fractional dimension" [8]. In the paper, he suggests the latter as a useful measure when talking about fractals.

It had long been observed that natural coastlines have statistical self-similarity, and that that when measuring coastlines, the measured length increases as the increment of measurement decreases. The more irregular the coastline, the more pronounced this effect is – and the effect doesn't disappear, even when our increment of measurement becomes very small.

Contrast the task of measuring a coastline – or measuring a theoretical fractal curve – with the task of measuring the circumference of a large circle (or measuring the perimeter of any non-fractal figure). If we have a rigid measuring instrument, and if we always place the ends of the instrument on the perimeter of the circle, then each measurement underestimates the circumference. But if we first use a yardstick, then a 12" ruler, then lengths of wood cut to 6" lengths, 3" lengths, etc., then our estimates get more accurate as we move to smaller measuring sticks; more importantly, we can see that our estimate is approaching some fixed limit. This is exactly what we expect for a non-fractal curve.

On the other hand, if the measured length doesn't seem to approach a limit, but continues to increase by a constant factor (or at least roughly constant) as the increment of measurement gets smaller by a constant factor (as is the case for theoretical fractals and – within practical limits – for coastlines), Mandelbrot asserted that we can use these two factors to compute a measure of the complexity of the curve. He called this property of the curve its *fractional dimension*; it's now more commonly called *fractal dimension*, and it can be calculated or estimated not only for curves but also for constructs of higher topological dimensions (surfaces, volumes, etc.).

At the risk of over-simplifying, we can think of this fractal dimension as a measure of how much an *n*-dimensional figure (e.g. a 1-dimensional curve) "fills" a higher-dimensional space (e.g. a 2-dimensional plane).

Applying this to the examples we've seen so far, we can say that another way of distinguishing a fractal curve from a non-fractal curve is that while a non-fractal curve (i.e. the curve of a circle) has a fractal dimension of 1, a fractal curve has a fractal dimension higher than 1, but less than 2. This is also true of branching fractals on a 2-dimensional plane, such as the Julia set example shown previously (p. 4).

Fractals defined in higher dimensions have similar relationships between embedding dimension, topological dimension, and fractal dimension. For example, if a theoretical 3-dimensional fractal has a surface, the area of that surface is infinite; the surface itself is 2-dimensional (topologically speaking), but it will generally have a fractal dimension greater than 2, and less than 3.

In the case of the human lung (Figure 8), the fractal dimension of the alveoli surface is approximately 2.97. In other words, we have a surface of topological dimension 2, nearly filling a 3-dimensional space. Since the ability of lungs to absorb oxygen into the bloodstream is largely a function of the alveoli surface area, it makes sense that this branching structure would evolve to have very close to the maximum possible surface area.

Diffusion-limited aggregation

Definition

In 1981, physicists Thomas Witten and Leonard Sander proposed an approach for explaining and modeling the way that certain materials formed low density, high surface area, branching structures (e.g. Figure 9) [9]. These statistically self-similar fractal structures typically form when the component materials are distributed through diffusion in some liquid or gas. Reflecting this, the modeling approach (and the explanatory mechanism for the real-world processes) is called diffusion-limited aggregation (DLA). In a model, it operates as follows:

1. Start with a stationary particle at the center of a large lattice – a regularly spaced arrangement of points.[1] This is the "seed" around which the aggregate will form.
2. Add a particle to a random point on the lattice, some distance away from the seed.
3. Let this new particle take a random step along the lattice – i.e. move to one of the adjacent lattice points, selected at random with equal likelihood. If this step causes the particle to leave the boundaries of the lattice, reposition it as in step 2.
4. Repeat step 3 until the random walk brings the moving particle to a point adjacent to a stationary particle. When that happens, the moving particle joins the aggregate – that is, it becomes stationary itself.
5. Repeat steps 2-4 to add more particles to the aggregate, until it reaches the desired mass or size.

An example of an aggregate formed in this fashion (using a model similar to the one we'll be building) is shown in Figure 10.

1 Typically, either a square or equilateral triangular lattice is used. The approach can also work in more than 2 dimensions; for example, in 3 dimensions, we could use a cubic or regular tetrahedral lattice.

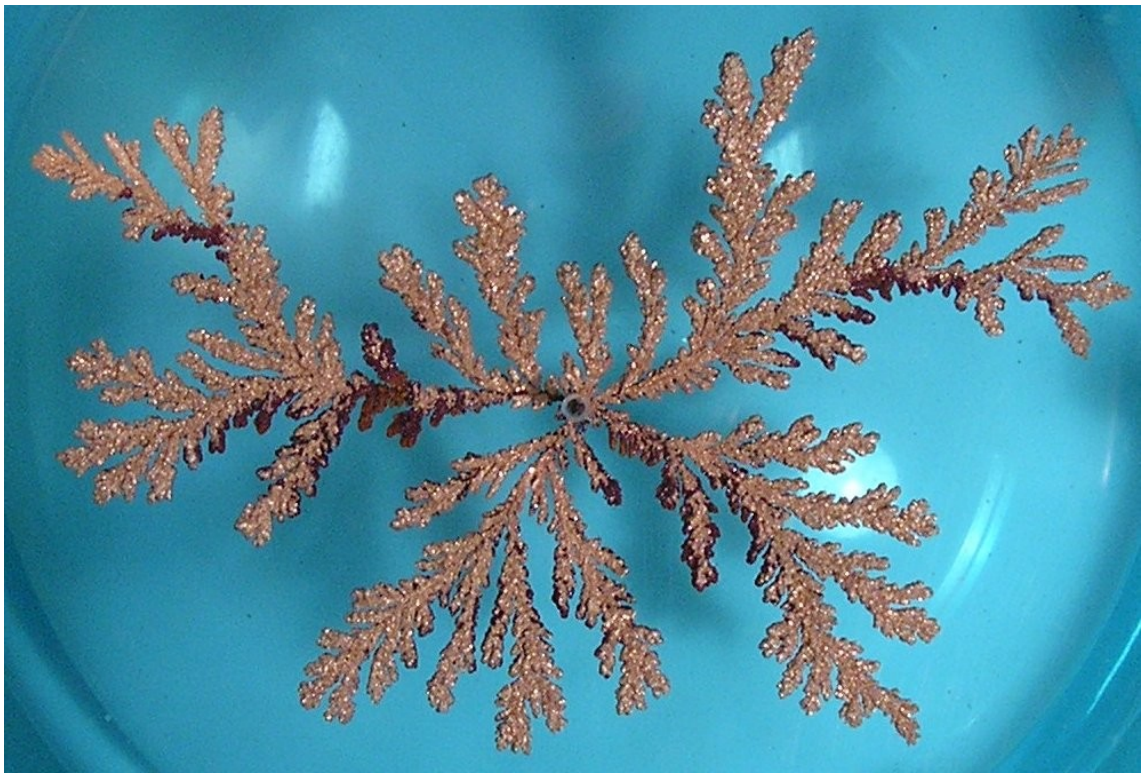

Figure 9 – Copper aggregate formed in a copper sulfate solution [3].

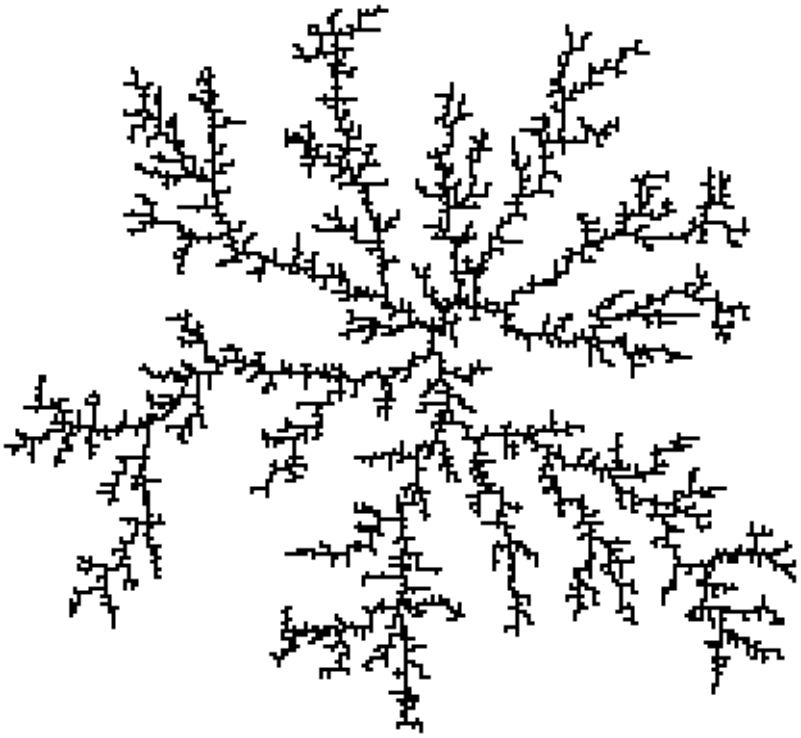

Figure 10 – DLA simulation with 5,000 particles on a square lattice

Development and use

In the years since the first publication of the approach, DLA and its variants have been used (and extended) to model a much wider variety of systems than the authors originally anticipated. Applications include not only chemical and physical systems characterized by diffusion, but biological, ecological, and social systems as well; in such applications, diffusion and aggregation are used as analogues or proxies for the actual transport or grouping mechanisms in the systems.

One such application is found in two 1989 papers on urban growth, by S. Alexander Fotheringham, Michael Batty, and Paul Longley [10],[11]. In the second paper ("Diffusion-Limited Aggregation and the Fractal Nature of Urban Growth"), the authors conclude:

> The present research should be viewed as providing a means of understanding elements such as the contiguous nature of development, the tentacular nature of urban growth, and the presence of urban density gradients, that are common to many cities. In a broader context, DLA provides an example that order can be abstracted from structures such as cities that are seemingy chaotic.

Another surprising class of examples is found in the work of mathematician and artist Gary Greenfield, who has explored the artistic application of DLAs in combination with evolutionary computing and image rendering via mosaic tiles [12],[13]. An example of Greenfield's work combining DLA and tiling is seen in Error: Reference source not found.

In 2000, Leonard Sander wrote a follow-up to the original 1981 DLA paper (he has also written several other papers on the subject), partly with the aim of surveying the breadth of DLA applications [14]. In this paper, he states:

> What makes this subject so interesting and, in fact, rather peculiar are three facts.
>
> (i) The extremely simple process seems to seize the essential ingredients of a great many natural phenomena with very little physical input.
>
> (ii) It produces clusters of intriguing complexity which look very much like real objects which are random, tenuous and approximately self-similar. The mathematical fact that the simple algorithm makes self-similar (fractal) clusters is remarkable. The fact that things very like this occur rather commonly in nature is still more remarkable.
>
> (iii) The simple process in the algorithm has resisted analysis despite the fact that the model is very widely known. This is a devilishly difficult model to solve,

even approximately.

It is precisely this set of characteristics that make DLA an interesting topic for NetLogo implementation – and a good fit with NetLogo's capabilities.

Figure 11 – Algorithmic enlargement of 194 X 259 photograph (by J. Ward) using DLA dendrite mosaic tiles [13].

Basic NetLogo implementation

General specifications

For our first DLA model, we'll stick fairly closely to Witten & Sander's description (using a turtle breed as the particles and patches as the lattice) with a few exceptions:

- Instead of adding a single particle at a time to the aggregate, we'll create several moving particles; all of these particles will take a step in a random walk on each tick. However, these particles will not interact with each other in any way.
- Rather than relocate a moving particle to a random location when a random walk step would cause it to leave the lattice, we'll enable horizontal and vertical wrapping (these options are enabled by default anyway).
- When a particle becomes stationary and is added to the aggregate, we'll indicate it by changing the patch color where the particle is located; then, the particle will be moved to a random location in the NetLogo world, as if a new moving particle were being added at that time. Thus, the aggregate will appear as colored patches.
- For this version of the model, we won't worry about initially placing the particles (or moving them after aggregation) some distance away from the aggregate; instead, a random patch location will be chosen.

NetLogo concepts and vocabulary

The following assumes some familiarity with basic concepts of agent-based modeling and NetLogo. Even with this background, it might be useful to review the companion document, "NetLogo Tutorial Series: Introduction and Core Concepts" [15].

The following tables summarize the NetLogo keywords, predefined agent variables, command primitives, reporter primitives, and user interface elements this model will use. Although the sheer number shown here may be intimidating, most are quite simple to understand and use. Even better, while the full list of command & reporter primitives, predefined agent variables, predefined constants, and keywords numbers in the hundreds, those used in the model make up a sizable fraction of the essential subset of the NetLogo language elements, sufficient to build most models.

Keep in mind that when the ***`agentset`*** placeholder appears in these summaries, the specified agentset may be a predefined agentset (**`turtles`**, **`patches`**, **`links`**), a breed agentset (declared with the **`breed`** keyword), or any reporter expression that evaluates to an agentset.

A more extensive summary of agentset-related primitives can be found in the companion document, "NetLogo Tutorial Series: Set Theory Concepts and Applications" [16].

These table entries should not be read as normative or authoritative; the NetLogo Dictionary should be used for that purpose [17].

Keyword	Declaration or definition purpose
`breed [plural-name singular name]`	Declares a custom turtle breed. ***plural-name*** is declared as a global agentset (all turtles of this breed are added to the agentset automatically), and can be used in a **create-*breeds*** command. ***singular-name*** is optional (but highly recommended); it is declared as a reporter which can be invoked as **singular-name *who-number*** to identify a single turtle of this breed. All **breed** declarations must appear before any procedure definitions in the code.
to ***procedure-name*** [***input-parameters***] ***commands*** **end**	Declares and defines the procedure ***procedure-name***. When invoked, the current agent executes ***commands***. The square brackets and ***input-parameters*** are optional; if specified, the symbol names in ***input-parameters*** may be accessed as local variables in the scope of the procedure, and actual parameter values must be provided (as literal values or reporter expressions) when ***procedure-name*** is invoked.

Table 1 – Keywords used in basic implementation of DLA.

Predefined agent variable	Contains
color	Color of turtle
heading	Direction turtle is facing, expressed in degrees (0 to 360)
pcolor	Color of patch
shape	Display shape of turtle. This value must be the name of one of the shapes in the **Turtle Shapes Editor** (available via the **Tools/Turtle Shapes Editor** menu command).

Table 2 – Predefined agent variables used in basic implementation of DLA.

Command primitive	Action	Applicable agents
`ask agent [commands]` `ask agentset [commands]`	Instructs ***agent*** or ***agentset*** to execute ***commands***.	observer, turtles, patches, links
`clear-all`	Kills all turtle and link agents; resets patch agents to default state; resets global variables to default values; clears all plots; clears all output elements; disables simulation clock.	observer
`create-breeds number [commands]`	Creates ***number*** of turtles of breed ***breeds*** (which may be a custom breed or the generic `turtles` breed). The square brackets and ***commands*** are optional; if specified, each turtle created will execute ***commands***, after all ***number*** turtles have been created.	observer
`forward number`	Instructs the current turtle to move ***number*** steps forward.	turtles
`if condition [commands]`	The current agent evaluates the Boolean reporter ***condition***, and – if the result is `true` – executes ***commands***.	observer, turtles, patches, links
`let variable value`	Declares a local ***variable*** in the scope of the current procedure or command block, and assigns it an initial ***value***.	observer, turtles, patches, links
`reset-ticks`	Enables and sets the simulation clock to an initial value of 0.	observer
`set variable value`	Assigns ***value*** to a global variable, agent ***variable*** of the current agent (a turtle may treat the variables of the patch where it is standing as its own variables), or local ***variable*** already declared in the current scope.	observer, turtles, patches, links
`setxy x y`	Changes the (X, Y) coordinates of the current turtle to (***x*** , ***y***).	turtles
`tick`	Advances the simulation clock by 1.	observer

Table 3 – Command primitives used in basic implementation of DLA.

Reporter primitive	Reports …
`any?` ***agentset***	… `true` if ***agentset*** has any elements; `false` if it's empty.
`count` ***agentset***	… the number of elements in ***agentset***.
`neighbors4`	… the subset of `patches` that are directly – but not diagonally – adjacent to the current patch.
`[`***reporter***`] of` ***agent*** `[`***reporter***`] of` ***agentset***	… the value of ***reporter***, when evaluated by ***agent***, or a list of the values of ***reporter*** evaluated by all members of ***agentset*** (in random order).
`one-of` ***list*** `one-of` ***agentset***	… one element of ***list*** or ***agentset***, selected at random with equal probability.
`patch` ***x y***	… the patch agent located at coordinates (***x***, ***y***) (automatically rounded to integers, as necessary).
`random-pxcor` `random-pycor`	… a random integer in the range of the currently defined X or Y axis (respectively), with uniform probability.
agentset `with` `[`***condition***`]`	… the subset of ***agentset*** containing all elements for which ***condition*** reports a value of `true`.

Table 4 – Reporter primitives used in basic implementation of DLA.

User interface element	Purpose
button	Instructs all agents of a single type (observer, turtles, patches, links) to execute a sequence of commands. The button can be configured for execution once per button click, or as a toggle – clicked once to activate and again to deactivate; in the latter mode, the specified commands are executed repeatedly while the button is activated (pressed).
monitor	Evaluates and displays the result of a specified reporter expression. This evaluation is performed by the observer, asynchronously with any model code – and regardless of whether any button is currently pressed.
slider	Declares a global variable and presents a user control for setting its value within a specified numeric range.

Table 5 – User interface elements used in basic implementation of DLA.

Configuring the NetLogo world

1. Start by opening NetLogo 5.x, or – if it's already running – selecting the **File/New** menu command, to create a new model. (Do not use the 3D version of NetLogo for this model.)
2. Before we write any code, we need to set the logical size and topology of the world, and make sure that the entire world can be displayed on the screen. (The last part isn't strictly necessary, but it does make it a lot easier, when we're building a model, if we can see the whole thing.)
3. Maximize the NetLogo window, so it takes up the entire screen (or as much of it as you prefer). Then click the **Settings…** button in the toolbar area near the top of the **Interface** tab. In the **Model Settings** dialog window, change or verify the following settings:
   - **Location of origin**: Center
   - **max-pxcor**: 87
   - **max-pycor**: 87
   - **World wraps horizontally**: checked
   - **World wraps vertically**: checked
   - **Patch size**: 3
   - **Font size**: (not used in this model; leave set to default value)
   - **Frame rate**: 30
   - **Show tick counter**: checked
   - **Tick counter label**: ticks

   Model settings should now resemble Figure 12.
4. Click the **OK** button to apply the specified settings.
5. If the NetLogo world occupies too much or too little space, repeat steps 3-4, as necessary, adusting the **Patch size** setting.

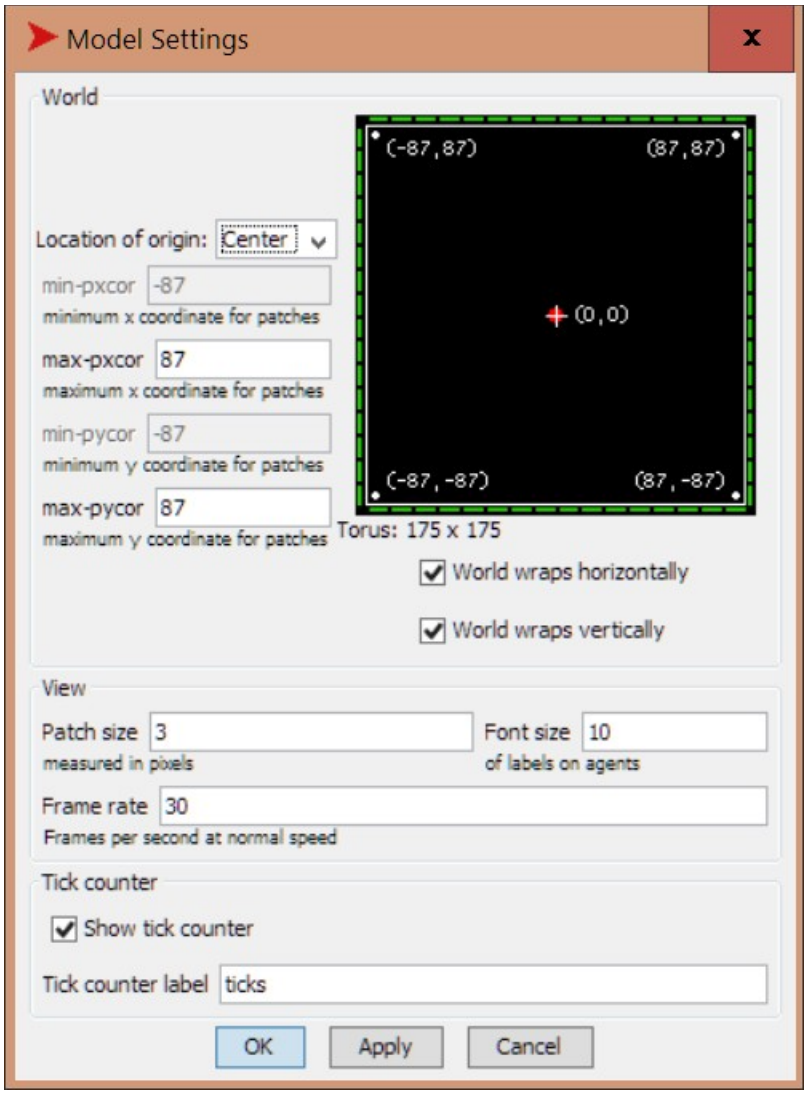

Figure 12 – Recommended model settings for basic implementation.

Creating a slider for the number of particles

1. Create a slider by right-clicking (Ctrl-click on Mac) somewhere in the white space to the left or right of the NetLogo world and selecting **Slider** from the pop-up menu, or by selecting **Slider** from the pull-down menu to the right of the **Add** button and clicking in the white space to the left or right of the NetLogo world.
2. In the **Slider** dialog window that appears, set or verify the following:
   - **Global variable**: num-particles

     (Note: Make sure there are *no* spaces in this field – not even after the end of the text!)

- **Minimum**: 0
- **Increment**: 10
- **Maximum**: 500
- **Value**: 500
- **Units**: (blank)
- **vertical?** not checked

3. After checking your settings against Figure 13, click the **OK** button to create the slider with the specified settings, and to declare a global variable named **`num-particles`**. (Note that this declaration will not appear in the **Code** tab of your model, but is implicit in the existence of a slider by the same name.)

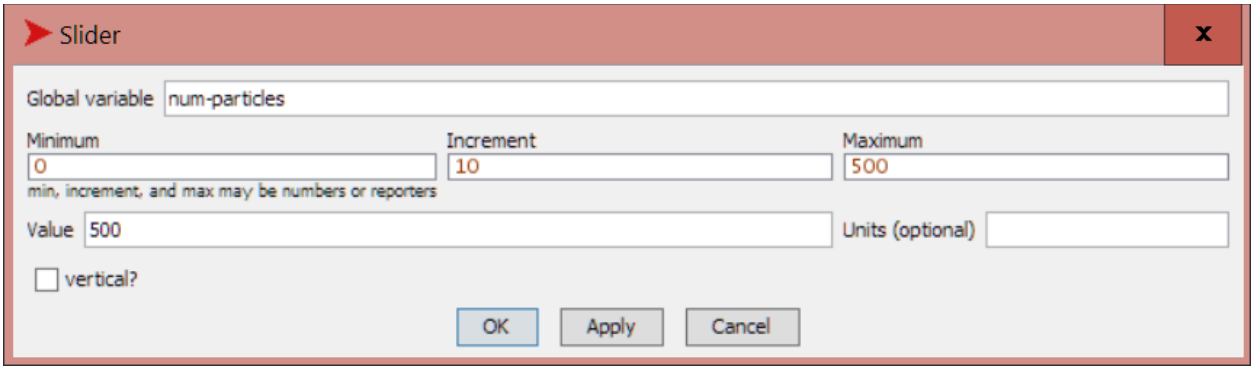

*Figure 13 – Settings for the **num-particles** slider and global variable.*

Writing code to initialize the model

1. Switch to the **Code** tab.
2. Write the code shown in Listing 1 to declare a **`particles`** breed, and to initialize the model with a single white patch in the center of the world, and **`num-particles`** turtles of the **`particles`** breed distributed randomly around the world. Note that by convention, the main command procedure for initializing the model is called **setup**; it's not required to use this name, but since the convention is widely used, following it will make it easier for others to understand your code.

   As you write and the review the code, you might find it useful to read the summary of the keywords, primitives, and variables in the "NetLogo concepts and vocabulary" section (p. 13).

   Note that in this code (and in some instances that follow), parentheses have been used for visual clarity; however, they're not required in this case, or in any of the code in this tutorial. Be sure that if you use parentheses, they are correctly balanced.

3. Use the **Check** button (in the toolbar near the top of the **Code** tab) to verify that the syntax of your code is correct. Keep in mind that the great majority of errors caught by this feature have the following causes:
   - Incorrect spelling of NetLogo primitives, keywords, predefined variables, and predefined constants (accidentally replacing hyphens with spaces or underscores is a particularly common culprit);
   - Inconsistent spelling of your own variable, breed, and procedure names;
   - Missing white space before or after mathematical or logical operators.
   - Mismatched or misplaced square brackets or parentheses.
4. If an error messages appears when you check your code, read the message carefully, note the highlighted location in the code, and use this information to correct the error.

```
breed [particles particle]

to setup
  clear-all
  ask (patch 0 0) [
    set pcolor white
  ]
  create-particles num-particles [
    set color green
    set shape "circle"
    setxy random-pxcor random-pycor
  ]
  reset-ticks
end
```

*Listing 1 – **particles** breed and **setup** procedure.*

Creating a button to invoke the `setup` procedure

1. In the **Interface** tab, create a button in the same way that you previously created a slider, but selecting **Button** from the pop-up or pull-down menu.
2. In the Button dialog window, set the following values:
   - **Agent(s)**: observer
   - **Forever**: not checked
   - **Disable until ticks start**: not checked
   - **Commands**

     ```
     setup
     ```

- **Display name**: (optional; if not specified, the text of the **Commands** field will be displayed as the button label)
- **Action key**: (optional)

3. Verify your settings against Figure 14 (keeping in mind that **Display name** and **Action key** are optional), and click **OK** to create your S**etup** button.

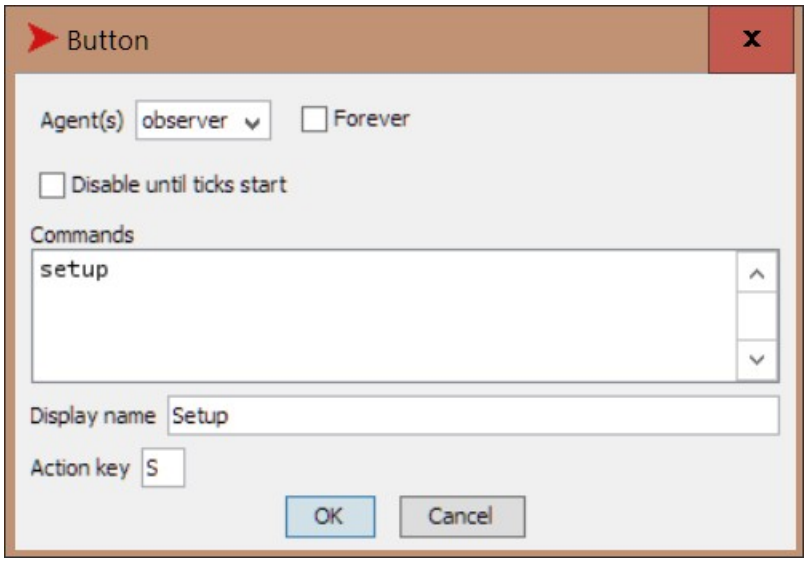

*Figure 14 – **Setup** button settings.*

Saving and testing your model so far

1. From the pull-down control below the **view updates** checkbox (in the toolbar of the **Interface** tab), select **on ticks**. For this model, this setting will make the display of the model smoother – in the initialization as well as (eventually) in the main simulation execution.
2. Use the **File/Save** command to name and save your model (make sure it's saved to an appropriate data directory, and not the NetLogo program directory).
3. Click the **Setup** button to invoke your `setup` procedure, and initialize your model. If you have the **num-particles** slider set to 500, your NetLogo world should resemble Figure 15. The small white square at the center of the world is the patch at the center of the lattice, and each green dot is a turtle agent of the `particles` breed.
4. Repeat step 2 with different values of **num-particles**, verifying that the number of particles created changes with different slider settings.

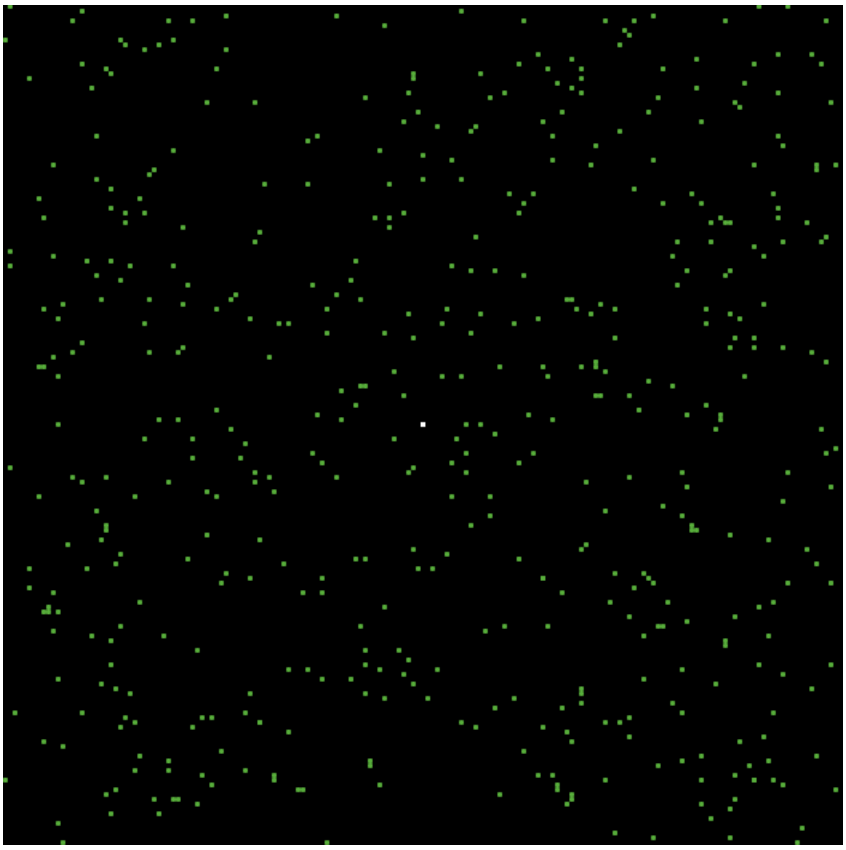

*Figure 15 – NetLogo world after setup, with **num-particles** = 500.*

Adding code for particle movement

1. In the **Code** tab, add the code shown in Listing 2 after your **`setup`** procedure. (Make sure not to start typing a new procedure before the **`end`** keyword of the previous procedure; that will cause a syntax error.)

   There are a few important points to note in this code:

   - The new code is split into two procedures, **`go`** and **`move`**. Just as we followed the convention of naming the initialization procedure **`setup`**, we will follow the convention of naming our main model execution procedure **`go`**.
   - Since our particles will eventually be performing two fundamentally distinct operations – moving, and then joining the aggregate when they get close enough to it – we'll implement those operations in separate procedures, and invoke both of them from the **`go`** procedure. For now, we've only implemented the movement, but we'll be adding the aggregation soon enough.
   - In the **`move`** procedure, the current particle selects a compass direction at random from a list of 4 directions, corresponding to north (up), east (right), south (down),

and west (left). It assigns the selected to direction to its **heading** variable; setting this variable is one of a few different methods for changing the direction that the turtle is facing. Finally, the turtle moves forward one step. This type of movement is known as a "2-dimensional random walk"; when repeated a large number of times, a random walk approximates *Brownian motion*, the random motion of particles in a fluid.

2. Use the **Check** button to verify that your code is correctly typed; correct any flagged errors.

```
to go
  ask particles [
    move
  ]
  tick
end

to move
  set heading (one-of [0 90 180 270])
  forward 1
end
```

*Listing 2 – **go** and **move** procedures for particle movement.*

Creating a button to test particle movement

1. Create another button in the **Interface** tab.
2. In the Button dialog window, set the following values:
   - **Agent(s)**: observer
   - **Forever**: checked
   - **Disable until ticks start**: checked
   - **Commands**

     ```
     go
     ```

   - **Display name**: (optional)
   - **Action key**: (optional)
3. Verify your settings against Figure 16 (again, remember that **Display name** and **Action key** are optional), and click **OK** to create your **Go** button.
4. Save your model.
5. Test particle movement in your model by clicking the **Setup** button to create particles, and the **Go** button to set them in motion. Verify that all particles remain in

motion until you click the **Go** button again. (If every particle only moves once, and the **Go** button doesn't stay down, edit the settings of the **Go** button by right-clicking on it, and selecting Edit… from the context menu that appears. In the **Button** settings dialog window, make sure that the **Forever** checkbox is checked.)

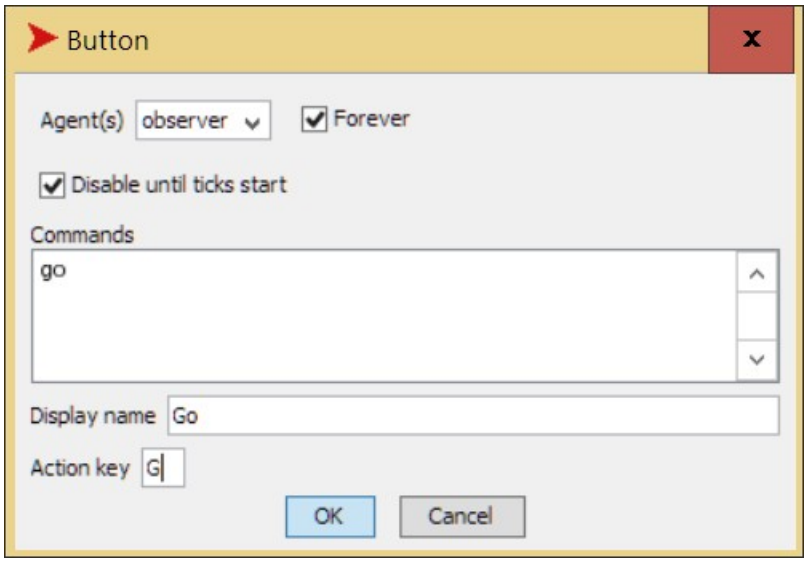

*Figure 16 – **Go** button settings.*

Adding code for particle aggregation

Now we'll add the code for particle aggregation. This will consist of a new procedure, and an additional line of code in an existing procedure (**`go`**). Because of this, the code in Error: Reference source not found includes both the existing and the new procedure. However, the code written previously is shown in gray italic type. Do not write the code displayed in italics! Instead, use it to locate the appropriate place to add the new code.

1. In the **Code** tab, add the non-italic code shown in Listing 3 to the code of your model. Pay close attention to the line of code added to the **`go`** procedure: it *must* be added after the **`move`** command, but before the close of the square brackets in which the **`move`** command is contained. The **`aggregate`** procedure may be written immediately after the **`go`** procedure, and before the **`move`** procedure, or added at the bottom of the code.

   Again, there are some important points to note in the new code:

   - **`neighbors4`** reports the set of patches directly (but not diagonally) adjacent to the current patch. If it's used by a turtle (as it is in this case), the current patch is the patch on which the turtle is standing.

- **with** is a very powerful general purpose set-oriented reporter, used to construct and report a subset of a specified set, where every member of the subset satisfies some condition of interest. In this case, we're using it to get the subset of patches in **neighbors4** whose **pcolor** (patch color) variable has a value that's not equal to **black**. (Alternatively, we could check for a **pcolor** equal to **white**, since that's the color we're using for the aggregate. However, if we later decided to change the color of the aggregate – or even have multiple aggregates of different colors forming at the same time – we'd have to remember to change any other references to **white** in our code. With the approach we're using now, as long as we keep **black** as the default patch color, we won't have to remember to change details like that in our code as our model evolves.)
- Because NetLogo allows procedure and variable names to include many non-alphanumeric characters, arithmetic operators such as **+**, **-**, **/**, *****, **^**, etc. and logical operators such as **=**, **!=**, **>**, **<**. **>=**, **<=**, etc. must have white space before and after them, to be recognized as operators. The only exceptions are for **-** (the minus sign): When it precedes a numeric literal with no intervening white space, it's treated as part of the literal value (making the literal value negative); when it follows a left parenthesis with no intervening white space, but is then followed by white space, it is treated as a unary negation operator for the following term (whether literal or symbolic).
- Because we'll be referring to this subset of patches more than once, we're using **let** to declare the local variable **aggregate-neighbors**, and assigning the subset to that variable.
- We're using the **any?** reporter to check **aggregate-neighbors** has any elements. If so, we can safely conclude that the current particle is adjacent to the aggregate, and should be added to it.
- To add the particle to the aggregate, we select **one-of aggregate-neighbors** at random, and use **of** to have the selected patch report to the particle its **pcolor**; the particle then changes the **pcolor** of the patch it's standing on to the reported color. Finally, the particle moves to a random location, as if it were a new particle in the lattice.

2. Check your code syntax; correct any flagged errors.

```
to go
  ask particles [
    move
    aggregate
  ]
  tick
end

to aggregate
  let aggregate-neighbors (neighbors4 with [pcolor != black])
  if (any? aggregate-neighbors) [
    set pcolor ([pcolor] of one-of aggregate-neighbors)
    setxy random-pxcor random-pycor
  ]
end
```

*Listing 3 – **aggregate** procedure definition and invocation.*

Test your basic DLA model

You're now in condition to test the not only the movement of particles, but also the addition of particles to the aggregate structure, when their random walks lead them to it. Since you've already created a button that invokes the go procedure, and since you've modified the go procedure to invoke the new aggregate procedure, you won't need to create any new user interface elements to test your model.

1. Save the model.

   In general, it's important to save before testing any major changes to the main simulation logic of a model. It's easy to write code – unintentionally – that causes NetLogo to become unresponsive. We can often use the **Tools/Halt** menu command to break out of such a condition, but this doesn't always work: occasionally it's necessary to kill the NetLogo process altogether. If we haven't saved our latest changes when we do that, we'll lose some of our work.

2. Click **Setup**, then **Go**, to create the particles and set them in motion. Before too long, you should see the white spot in the center of the world – i.e. the aggregate – expand from a single square to an irregular clump of patches, as randomly walking particles collide with and join the aggregate.

   Notice the speed slider in the toolbar of the Interface tab – set by default to **normal speed**. Sliding it to the right causes NetLogo to wait a shorter time – and update the screen less frequently – when executing the **`tick`** command. With your DLA model, you'll generally want to set the speed slider close to its maximum value. (At the maximum speed, the display will only be updated every few seconds, and NetLogo will try to process as many iterations of **`go`** as possible between screen updates.)

You can also slide the speed slider far to the left, to slow your model down. This will help if it appears that your model isn't doing what you expect: by observing the model run very slowly, you'll be able to see if the particles are moving in an unexpected fashion, or if particles aren't correctly joining the aggregate when they are directly adjacent to some portion of it.

Monitoring the size of the aggregate

We haven't (yet) included any code in the model for halting execution automatically when the aggregate grows to some critical size. But even without that, we can easily monitor the size of the aggregate as it grows, by adding a display of the number of patches that are part of the aggregate.

1. In the **Interface** tab, use the menu of interface elements to add a monitor.
2. In the **Monitor** dialog window, set the following values:
   - **Reporter**:
     ```
     count patches with [pcolor != black]
     ```
   - **Display name**: Aggregate Size
   - **Decimal places**: 0
   - **Font size**: 11
3. After checking your settings against Figure 17, click **OK**.
4. Save your model.
5. Run the model as usual, watching the value displayed by **Aggregate Size** change as the number of patches in the aggregate increases.

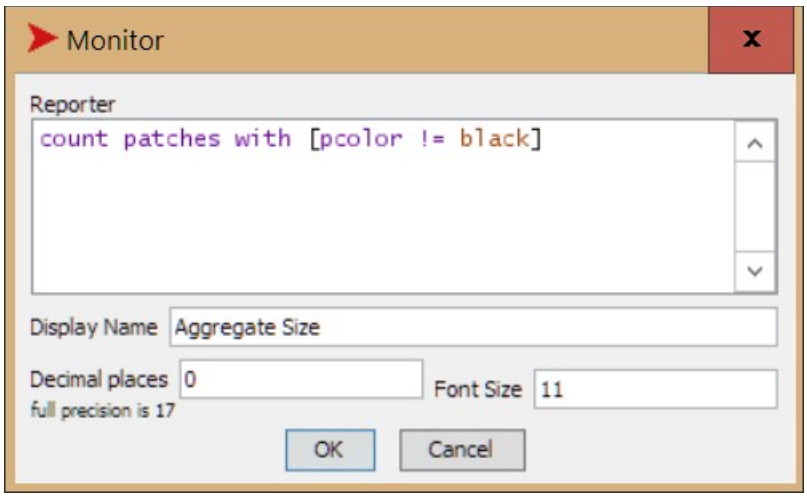

*Figure 17 – **Aggregate Size** monitor settings.*

This page intentionally left blank.

Extending the basic implementation

What's missing?

As noted in “General specifications” (p. 13), we've left a few elements out of our model that were part of Witten & Sander's original DLA definition.

As it turns out, it's fairly common for DLA implementations to include multiple particles moving simultaneously (within reason, of course – the fundamental theory assumes this number remains fairly low), rather than waiting for each particle to join the aggregate before adding the next particle to the lattice. So we won't change that aspect of our model.

On the other hand, we will address the likelihood of particles entering the lattice very close to the existing aggregate; we'll modify our code to have particles enter the lattice (and re-enter, after adding a patch to the aggregate) far from the center. Additionally, we'll add a stopping condition – code that halts execution automatically when a critical threshold is reached. In this case, we'll stop the model when any part of the aggregate is more than a certain distance away from the center of the world (the location of the seed).

NetLogo concepts and vocabulary

There's not much we need to add to our vocabulary for these changes – but the new keywords and primitives are useful in a wide variety of models – and essentially in quite a few.

Keyword	Declaration or definition purpose
`globals [variables]`	Declares global variables. There can be only one `globals` declaration in the code of a model file, but we can include multiple variables inside the brackets, as long as they're separated by white space. `globals` must appear in the code before any procedure definitions; it can appear before or after breed declarations.

Table 6 – Additional keywords used in extended implementation of DLA.

Command primitive	Action	Applicable agents
stop	Halts execution of the current procedure, returning immediately to the point of invocation. If the halted procedure was invoked directly by a button with the **forever** option checked, and **stop** was executed by the same agent that started execution of the procedure (i.e. not within an **ask** command block in the procedure), execution of that button's commands stops immediately.	observer, turtles, patches, links
move-to ***agent***	The current turtle agent moves to the identical (X, Y) location as ***agent***. If ***agent*** is a patch, then the turtle moves to the exact center of that patch.	turtles

Table 7 – Additional command primitives used in extended implementation of DLA.

Reporter primitive	Reports …
distancexy ***x y***	… the Euclidean distance between the current agent and the point by (***x***, ***y***). This reporter may not be used by the observer or a link agent, since neither of these has a location.
min-pxcor **max-pxcor** **min-pycor** **max-pycor**	… the minimum and maximum integral values along the X and Y coordinate axes, respectively, of the NetLogo world. The values reported are the same values specified in the **Model Settings** dialog window; these reporters are very useful for building without hard-coded world dimensions in the code.
patches	… the agentset containing all patches in the NetLogo world.

Table 8 – Additional reporter primitives used in extended implementation of DLA.

Changing the code for particle placement and adding the stop condition

To reduce the likelihood that a particle enters or re-enters the lattice very close to some part of the aggregate, we'll place each particle a distance of just under the half the width of the world (or more) from the center of the lattice. To make the code for this as simple as possible, we'll make some assumption about the shape (but not the size) of the world, and the location of the origin.

Specifically, we'll assume the following:

- The world is square – that is, its height and width are equal.
- The origin of the coordinate system (which is also the location of the seed of the aggregate) is at the center of the world.

While it's a good idea to avoid making too many assumptions about the dimensions of the world – and especially to avoid embedding many such assumptions in our code – these are reasonably safe assumptions to make in this case.

Because the origin is assumed to be at the center of the world, we can conclude that the integral distance from the origin to the patches located on the *X*-axis, at its minimum and maximum values, is **`max-pxcor`**. Further, because the world is assumed to be square, the distance from the origin to each the patches at the extremes of the *Y*-axis is also **`max-pxcor`**. If, when placing the particles randomly, we make sure that each is at a distance from the origin of at least **`max-pxcor`**, they will thus be located (immediately after placement) somewhere along the edges and/or close to the corners. This will be sufficient for our purposes.

1. Use the **File/Save As…** menu command to save your model under a new name.

   Saving under a new name is highly recommended before making significant changes to a working (even if incomplete) model. Doing this ensures that even if you make an irrecoverable mistake with the changes to the new version, you haven't lost the work done previously.

2. Make the changes shown in Listing 4 to the **`setup`**, **`go`**, and **`aggregate`** procedures, and add the **`globals`** declaration block.

   Note that this code listing once again shows previously written code (not including the **`breed`** declaration or the **`move`** procedure, which are unchanged) in gray italic type. This time, however, a couple of those previous lines must be deleted; these are displayed in strike-through type. The new code will not function correctly without removing these struck-through lines from the code.

Even though we're only adding a few lines, there are important features to notice:

- Notice the global variable named **`limit-reached?`**, and that when this variable is referenced in the code, the question mark is included as part of the variable name. This follows a widely used Logo convention of using a question mark as the final character of a variable name to indicate that the variable is intended to hold a Boolean value. Keep in mind that NetLogo itself doesn't care how we name our variables: we can assign any kind of value to any variable we declare, and NetLogo will go along without complaint. Conventions like these are simply for the purpose of making code more self-documenting – and thus, easier to understand and maintain.
- Note that in the **`setup`** procedure, values are being assigned to the global variables *after* the **`clear-all`** command is executed. That order is important: If we reverse it, any values we assign to those variables will be lost when **`clear-all`** is executed.
- In **`setup`**, we're assign a subset of **`patches`** to the **`entry-patches`** variable – namely the subset that is at a distance of at least **`max-pxcor`** from the origin. We compute this subset once, and then use it from then on – selecting individual patches from it at random (using **`one-of`**), and using those patches as destinations for the **`move-to`** command when placing particles on the lattice.
- In the **`aggregate`** procedure, we've added code to check the distance of the patch being added to the aggregate; if that distance is greater than 90% of **`max-pxcor`**, then we conclude that the aggregate has grown close enough to the boundaries of the lattice that we should stop execution. Since a particle can't stop a button that's being executed by the observer, our approach is to have the particle set **`limit-reached?`** to **`true`**. The next time the observer executes **`go`**, it checks the value of **`limit-reached?`**. Since that value is now **`true`**, the observer executes the **`stop`** command, exiting the **`go`** procedure. Finally, since **`go`** was invoked directly from a button with the **forever** option checked, the **`stop`** command also terminates execution of that button.

3. As usual, check and fix (if needed) your code changes, and save your model.
4. Now, when you run the model, you should see all particles starting out on the edges or near the corners of the world. If we slow down the model execution, and watch for a specific particle to reach the aggregate, we'll also see that particle being relocated in the same fashion (reducing the number of agents will also help in verifying this).

   To verify that the stopping condition is in force, test the model at maximum speed,

with the maximum number of particles. Execution should stop automatically by the time the value shown by the **Aggregate Size** monitor reaches 2,000 or so; at that time, you should see something like the aggregate in Figure 18.

```
globals [
  limit-reached?
  entry-patches
]

to setup
  clear-all
  set limit-reached? false
  set entry-patches (patches with [distancexy 0 0 >= max-pxcor])
  ask (patch 0 0) [
    set pcolor white
  ]
  create-particles num-particles [
    set color green
    set shape "circle"
    setxy random-pxcor random-pycor
    move-to (one-of entry-patches)
  ]
  reset-ticks
end

to go
  if limit-reached? [
    stop
  ]
  ask particles [
    move
    aggregate
  ]
  tick
end

to aggregate
  let aggregate-neighbors (neighbors4 with [pcolor != black])
  if (any? aggregate-neighbors) [
    set pcolor ([pcolor] of one-of aggregate-neighbors)
    setxy random-pxcor random-pycor
    if (distancexy 0 0 > 0.9 * max-pxcor) [
      set limit-reached? true
    ]
    move-to (one-of entry-patches)
  ]
end
```

*Listing 4 – **globals** block, **setup**, **go** & **aggregate** procedure changes for extended implementation.*

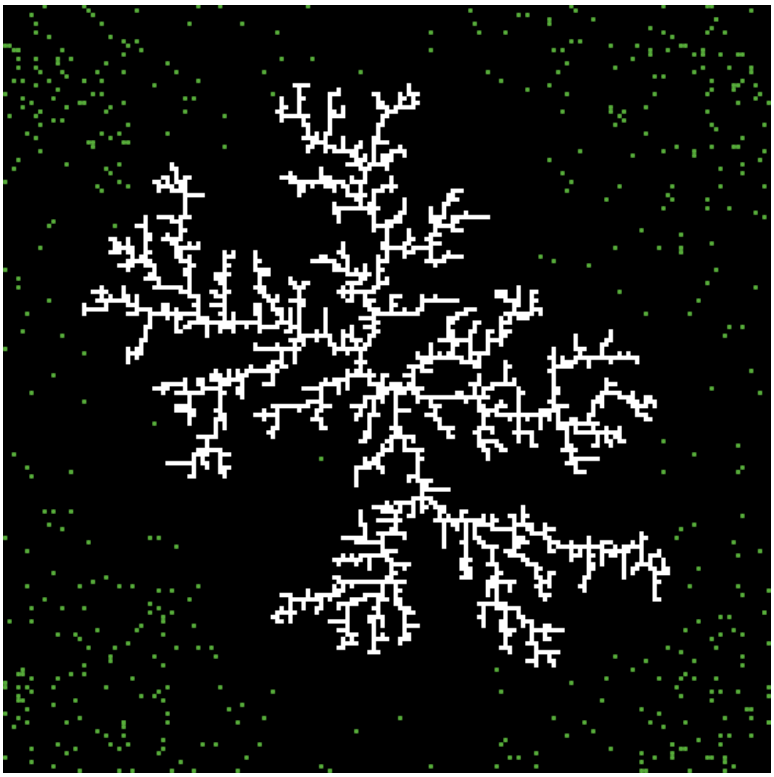

Figure 18 – Extended implementation, after reaching stopping condition.

Questions

1. Can you think of any systems – chemical, biological, social, etc. - that produce structures resembling those produced by your DLA so far?
2. Without worrying about the specific NetLogo code required, are there any basic functional changes that could be made to your DLA, to make it model more closely the processes you thought of for #1?

Aggregation along a line

Description

So far, we've been looking at an aggregate that starts with a single seed particle. Among other variants, we can also use DLA to describe and model processes in which aggregation occurs on a surface, or – in 2 dimensions – along a line.

In a simple model of such a process, we could start with seed particles occupying an entire row or column of the lattice. In fact, the only things we would absolutely need to change in our current model to accommodate this are the code for creating the initial seed(s), and the code for placing the particles on the lattice. In addition, we might want to treat the line along which aggregation begins as one-sided, instead of 2-sided; to do this, we would turn off wrapping in one direction, and create the line of seeds along a non-wrapping edge of the world.

In fact, we're going to do all of those things now. But this time, there will be fewer step-by-step instructions; you've already been introduced to almost all of the keywords, primitives, and predefined variables that you'll need for this version of the model.

NetLogo concepts and vocabulary

The only new elements needed for this model are predefined, read-only variables of every patch agent, holding that patch's (*X*, *Y*) coordination location, and a couple of reporter primitives giving the overall world dimensions. (Actually, we won't be using all of these, but some are included for completeness.)

Predefined agent variable	Contains
`pxcor` `pycor`	Integer *X*- and Y-coordinates (respectively) of the current patch.

Table 9 – Additional predefined agent variables used in aggregation along a line.

Reporter primitive	Reports …
`world-width` `world-height`	… the width and height (respectively) of the NetLogo world. These are always integer values.

Table 10 – Additional reporter primitives used in aggregation along a line.

Changing the world size and topology

1. As we did before the last set of changes, start by saving your model under a new name.
2. Open the **Model Settings** dialog (with the **Settings…** button), and change just the following parameters (all others should remain unchanged):
   - **max-pxcor**: 137
   - **World wraps vertically**: not checked (Note how the colors and line patterns along the edges of the small square representing the world change – indicating that the horizontal edges along the top and bottom of the world are now impassable – when you uncheck this option.)
3. Review the settings, to make sure that all of them are as intended; then click **OK**.

   You should now have a NetLogo world that is wider than it is tall. The height (in terms of patches, and in terms of the space occupied on the screen) should be unchanged from the previous model.

Changing the initial placement of seeds and particles

Up to this point, in our setup procedure, we ask a single patch to set it's **`pcolor`** to **`white`**; this patch is the seed of the aggregate. Now we need to change this. Specifically, we're going to ask all of the patches along the bottom row of the world to change their **`pcolor`** values to **`white`**.

We've seen how we can **`ask`** all of the agents in a turtle breed (e.g. **`particles`**) to execute a block of commands. We've also seen how we can use the **`with`** reporter to construct a subset of a given agentset, based on a predicate condition: this is how we identified the subset of a particle's neighboring patches that were already included in the aggregate; it's also how we identified the set of patches that were sufficiently far from the origin to serve as entry points into the lattice for particle placement.

Also, in our last set of changes, we changed the way that members of the **`particles`** breed were initially placed on the lattice, in the **`setup`** procedure. We won't change the code that places the particles, but we will change the set of patches that the placement locations are randomly selected from.

1. Review the assignment of a value to **`entry-patches`** in the **`setup`** procedure. Instead of selecting patches that are at least a certain distance from the origin, can you think of a way to select the patches that are in the top row of patches in the world? (Hint: Compare the **`pycor`** patch variable to **`max-pycor`**.)

2. Review the **ask** command currently used in the **setup** procedure. Can you think of a suitable condition that you could use with the **with** reporter, to get the subset of patches making up the entire bottom row of patches in the world? (Hint: Compare the **pycor** patch variable to **min-pycor**.)

3. Try to make the necessary changes to the **setup** procedure on your own. If you get stuck, or want to check your solution code before testing it, compare your **setup** procedure to that shown in Listing 5. Keep in mind that there are multiple approaches possible. If your approach is implemented with different code, it doesn't mean it's wrong: Test it out and see!

   This listing (which includes only the **setup** procedure) again shows code previously written in gray italics, as well as strike-through type for lines to be removed.

4. Check, save, and test your changes – but just using the **Setup** button; we need to change a few more lines in the **aggregate** procedure before we can use the **Go** button again.

   When you click the **Setup** button, you should see a broken green line along the top of the world; these are particles randomly placed in the top row of patches.

   Much harder to see is the row of white patches along the bottom of the world; this is the row of seed patches.

```
to setup
  clear-all
  set limit-reached? false
  set entry-patches (patches with [distancexy 0 0 >= max-pxcor]
  set entry-patches (patches with [pycor = max-pycor])
  ask (patch 0 0) [
  ask (patches with [pycor = min-pycor]) [
    set pcolor white
  ]
  create-particles num-particles [
    set color green
    set shape "circle"
    move-to (one-of entry-patches)
  ]
  reset-ticks
end
```

*Listing 5 – **setup** procedure changes for aggregation along a line.*

Changing the stopping condition

At the moment, our stopping condition code looks at the distance of the new aggregate patch from the origin, setting the variable **`limit-reached?`** to **`true`** if the distance is greater than a critical value. However, now that we'll be building our aggregates up from a line at the bottom of the world, rather than out from a seed at the origin, that condition no longer makes sense. Instead, we should base our test on the height of the aggregate above the bottom of the world.

1. Modify the condition of the **`if`** command in **`aggregate`**, to compare the current patch's height above the bottom row of patches to the overall height of the world. If the former is greater than 90% of the latter, set the value of **`limit-reached?`** to **`true`**. (Hint: **`pycor - min-pycor`** gives the height of the current patch above the bottom row of patches.)
2. When you're ready – or if you get stuck and need some concrete clues – review your changes to the aggregate procedure against those shown in Listing 6. Again, this is just one approach; there are definitely alternatives that work just as well.
3. Check, save, and test your code. Verifying that the stopping condition works correctly will take patience: Because all of the particles are starting so far from the aggregate, it takes a long time for the particles to reach and expand the aggregate.

   Eventually, you should see branching tree structures climbing from the bottom of the world, similar to those in Figure 19.

```
to setup
  clear-all
  set limit-reached? false
  set entry-patches (patches with [distancexy 0 0 >= max-pxcor]
  set entry-patches (patches with [pycor = max-pycor])
  ask (patch 0 0) [
  ask (patches with [pycor = min-pycor]) [
    set pcolor white
  ]
  create-particles num-particles [
    set color green
    set shape "circle"
    move-to (one-of entry-patches)
  ]
  reset-ticks
end
```

*Listing 6 – **aggregate** procedure changes for aggregation along a line.*

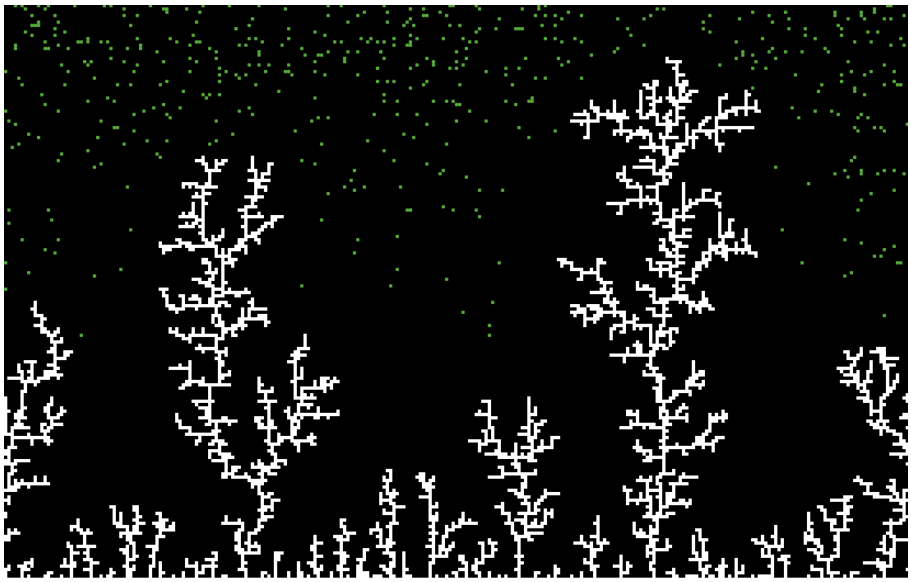

Figure 19 – Aggregation along a line.

Questions

1. What additional systems (beyond those you came up with previously) does this model resemble?
2. Can you think of any real-world processes that don't operate by simple aggregation, but in which aggregation (and specifically, DLA) might serve as a reasonable proxy for the actual mechanisms at work?

This page intentionally left blank.

Possible next steps

You've seen how the basic DLA model works, and experimented with a variation for producing aggregation along a line. There are a few directions for simple change or enhancement that are worth exploring. When you have time, pick one or more of the following, and try to implement it. (Don't forget: When using a previous model as a starting point, don't forget to save that model under a new name before making changes!)

1. Instead of looking only at the directly adjacent neighboring patches for aggregation, include the diagonally adjacent patches as well. (Hint: While the patch set reported by **`neighbor4`** includes only the patches directly adjacent to the current patch, **`neighbors`** also includes diagonally adjacent patches.)
2. Experiment with multiple distinct seeds of different colors. In doing so, see if you can find a way to place these seeds randomly, but within a certain distance from the origin. Also, note that currently, your code instructs a particle to pick one of its aggregate neighbors at random when selecting a color for the patch; for an advanced exercise, when a particle has multiple aggregates (of different colors) in its neighborhood, select the majority color among the aggregate patches in the neighborhood. (Hint: Assuming we have a variable called **`aggregate-neighbors`**, as currently defined in the code, **`[pcolor] of aggregate-neighbors`** reports a list of colors of the members of **`aggregate-neighbors`**; the **`mode`** reporter can be used to find the most common item in a list.)
3. Currently, particles that reach a patch adjacent to the aggregate add to the aggregate (i.e. change the color of the current patch to that of the aggregate) unconditionally. Will we get different results if the particle adds to the aggregate according to a configurable probability? (Hint: **`random-float 1`** reports a random value between 0 and 1.) For an advanced exercise, consider using BehaviorSpace to explore the effect of this parameter on the density of the aggregate.
4. Our particles currently move in a random walk fashion. This is a stationary process: over time, there's no specific direction in which they're most likely to drift. Can you modify the aggregation-on-a-line model, so that the particles are more likely to drift down over time? In fact, there's a simply way to modify the code so that the probability of stepping up (i.e. in the 0° direction) is zero, but the probability of selecting any of the other three directions remains equal; can you make that change?
5. Currently, particle movement and aggregation takes place on a lattice. Modify the **`setup`**, **`move`**, and **`aggregate`** procedures to remove the le attice constraint. Without the lattice, how should a particle detect that it is close to a portion of the lattice?

(Hints: **`right`** ***`angle`***, **`left`** ***`angle`***, **`set heading`** ***`angle`***, and **`face`** ***`agent`*** can all be used to turn a turtle; **`random-float 360`** can be used to compute a random floating-point value between 0 and 360; ***`agentset`*** **`in-radius`** ***`range`*** reports the subset of ***`agentset`*** that is located within ***`range`*** of the current agent.)

References

[1] B. Stanislaus, "Sierpinski triangle.svg", 21 Oct. 2007. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/4/45/Sierpinski_triangle.svg. [Accessed: 5 Oct. 2015].

[2] Fibonacci, "Koch curve.svg", 19 May 2007. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/5/5d/Koch_curve.svg. [Accessed: 5 Oct. 2015].

[3] K. R.. Johnson, "DLA Cluster.jpg", 19 May 2006. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/b/b8/DLA_Cluster.JPG. [Accessed: 5 Oct. 2015].

[4] J. Sullivan, "Fractal Broccoli.jpg", 21 Aug. 2004. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/4/4f/Fractal_Broccoli.jpg. [Accessed: 8 Oct. 2015].

[5] A. Heinemann, "Thorax Lung 3d (2).jpg", 5 Oct. 2005. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/7/77/Thorax_Lung_3d_%282%29.jpg. [Accessed: 8 Oct. 2015]

[6] J. Hubicka, T. Nash, *XaoS*, 2008. [Online]. Available: http://matek.hu/xaos/doku.php. [Accessed: 5 Oct. 2015].

[7] U. Wilensky, *NetLogo*, 2015. [Online]. Available: http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL. [Accessed: 5 Oct. 2015].

[8] B. Mandelbrot, "How long is the coast of Britain? Statistical self-similarity and fractional dimension", *Science*, vol. 156, no. 3775, pp. 636-638, May 1967.

[9] T. L. Witten, Jr., L. M. Sander, "Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon", *Physical Review Letters*, vol. 47, no. 9, pp. 1400-1403, Nov. 1981.

[10] M. Batty, P.Longley, S. Fotheringham, "Urban growth and form: scaling, fractal geometry, and diffusion-limited aggregation", *Environment and Planning A*, vol. 21, no. 11, pp. 1447-1472, Nov. 1989.

[11] A. S. Fotheringham, M. Batty, P.Longley, "Diffusion-Limited Aggregation and the Fractal Nature of Urban Growth", *Papers of the Regional Science Association*, vol. 67, no. 1, pp. 55-69, Dec. 1989.

[12] G. Greenfield, “Evolved Diffusion Limited Aggregation Compositions”, *Applications of Evolutionary Computing*, vol. 4974, pp. 402-411, Jan. 2008.

[13] G. Greenfield, “Connectivity and a Diffusion Limited Aggregation Digital Image Magnification Technique”, Journal for Geometry and Graphics, vol. 13, no. 2, pp. 187-194, Jan. 2009.

[14] L. M. Sander, “Diffusion-Limited Aggregation: A Kinetic Critical Phenomenon?”, *Contemporary Physics*, vol. 41, no. 4, pp. 203-218, Jul. 2000.

[15] N. Bennett, “NetLogo Tutorial Series: Introduction and Core Concepts“, Oct. 2015. [Online]. Available: http://www.g-r-c.com/tutorials/netlogo/Core %20Concepts%20(English).pdf. [Accessed: 5 Oct. 2015].

[16] N. Bennett, “NetLogo Tutorial Series: Set Theory Concepts and Applications“, Oct. 2015. [Online]. Available: http://www.g-r-c.com/tutorials/netlogo/Set %20Theory%20Concepts%20and%20Applications.pdf. [Accessed: 5 Oct. 2015].

[17] U. Wilensky, “NetLogo Dictionary”, *NetLogo User Manual*, 1 Oct. 2015. [Online]. Available: http://ccl.northwestern.edu/netlogo/docs/dictionary.html. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. [Accessed: 5 Oct. 2015].

Appendix: Source code listings

Basic implementation

```
breed [particles particle]

to setup
  clear-all
  ask patch 0 0 [
    set pcolor white
  ]
  create-particles num-particles [
    set color green
    set shape "circle"
    setxy random-pxcor random-pycor
  ]
  reset-ticks
end

to go
  ask particles [
    move
    aggregate
  ]
  tick
end

to move
  set heading (one-of [0 90 180 270])
  forward 1
end

to aggregate
  let aggregate-neighbors (neighbors4 with [pcolor != black])
  if (any? aggregate-neighbors) [
    set pcolor ([pcolor] of one-of aggregate-neighbors)
    setxy random-pxcor random-pycor
  ]
end
```

Extended implementation

```
breed [particles particle]

globals [
  limit-reached?
  entry-patches
]

to setup
  clear-all
  set limit-reached? false
  set entry-patches (patches with [distancexy 0 0 >= max-pxcor])
  ask patch 0 0 [
    set pcolor white
  ]
  create-particles num-particles [
    set color green
    set shape "circle"
    move-to (one-of entry-patches)
  ]
  reset-ticks
end

to go
  if limit-reached? [
    stop
  ]
  ask particles [
    move
    aggregate
  ]
  tick
end

to move
  set heading (one-of [0 90 180 270])
  forward 1
end

to aggregate
  let aggregate-neighbors (neighbors4 with [pcolor != black])
  if (any? aggregate-neighbors) [
    set pcolor ([pcolor] of one-of aggregate-neighbors)
    if (distancexy 0 0 > 0.9 * max-pxcor) [
      set limit-reached? true
    ]
    move-to (one-of entry-patches)
  ]
end
```

Aggregation along a line

```
breed [particles particle]

globals [
  limit-reached?
  entry-patches
]

to setup
  clear-all
  set limit-reached? false
  set entry-patches (patches with [pycor = max-pycor])
  ask (patches with [pycor = min-pycor]) [
    set pcolor white
  ]
  create-particles num-particles [
    set color green
    set shape "circle"
    move-to (one-of entry-patches)
  ]
  reset-ticks
end

to go
  if limit-reached? [
    stop
  ]
  ask particles [
    move
    aggregate
  ]
  tick
end

to move
  set heading (one-of [0 90 180 270])
  forward 1
end

to aggregate
  let aggregate-neighbors (neighbors4 with [pcolor != black])
  if (any? aggregate-neighbors) [
    set pcolor ([pcolor] of one-of aggregate-neighbors)
    if (pycor - min-pycor > 0.9 * world-height) [
      set limit-reached? true
    ]
    move-to (one-of entry-patches)
  ]
end
```