First steps in NetLogo with a movement model

Iza Romanowska and Colin Wren

Introduction

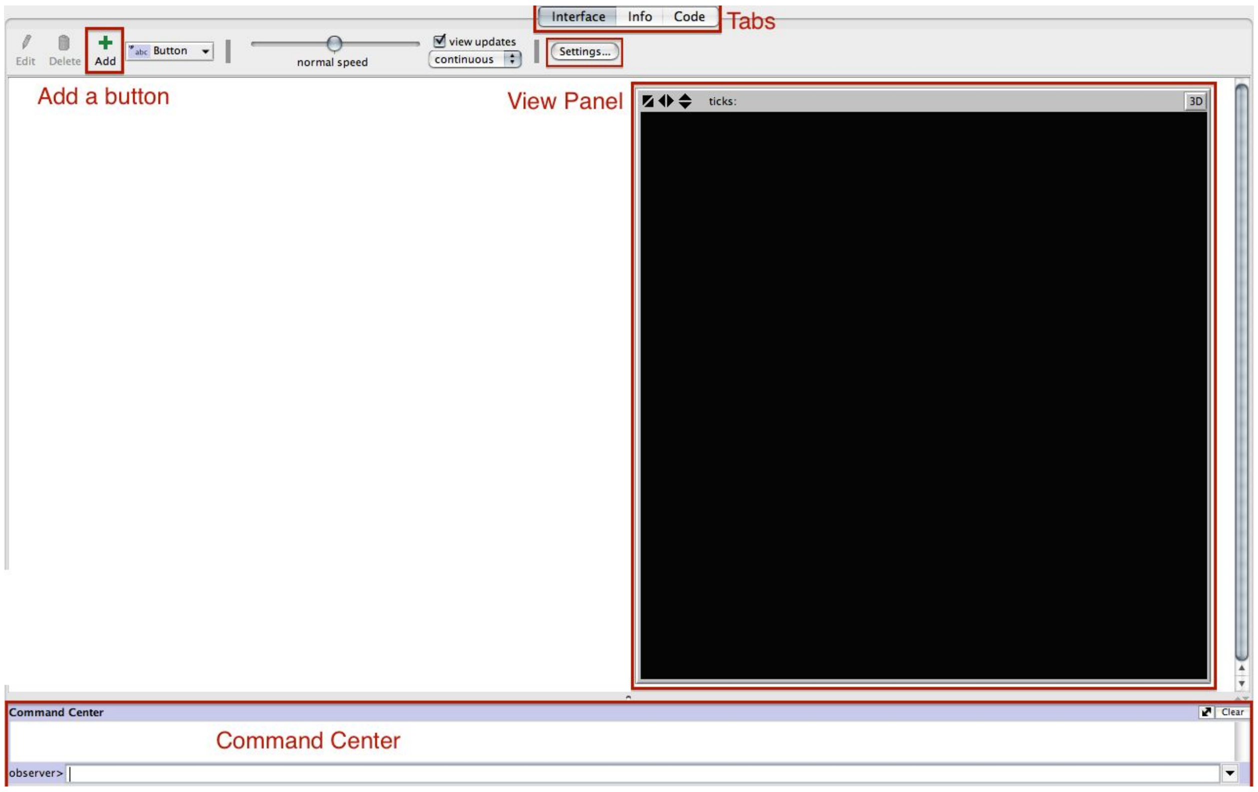
We will start with a simple model of human dispersal. Models of human movement lend themselves particularly well to simulation and ABM in particular. They concern a dynamic process that is inherently spatial - two characteristics that make ABM shine in comparison to other modelling techniques. To illustrate the process of simulation building we will use a model developed by Young and Bettinger (1995) (*Y&B model* henceforth). It was designed to investigate the first dispersal of humans out of Africa. This model is a good example of the so-called 'models from first principles' or 'toy-models', which are very popular in archaeology and related disciplines. We will use the Y&B model as a case study in a NetLogo tutorial to show the basics of the software. Finally, we will critically look back at what we have been doing and try to better define such terms as 'model' and 'simulation' and discuss how these tools fit within the scientific process and how can they aid archaeological research.

Tutorial

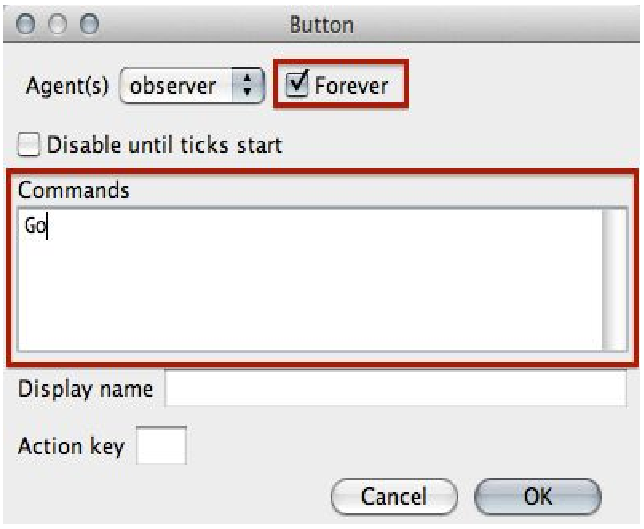
Logo - the language of NetLogo - was developed to resemble a natural language as much as possible, which means that it is easy to read and easy to write. In fact, it was originally designed to get school kids to code. In this introductory tutorial, you will earn your first coding stripes but also explore some of the most powerful capacities of simulation.

Let's start with the NetLogo interface. When you open it you will immediately notice three tabs: **Interface**, **Info** and **Code**. Let's look at them in turn. The Interface window consists of:

- the **View Panel** for watching the simulation,
- a few buttons, choosers and a slider along the top of the window,
- the **Command Center** towards the bottom of the window.



Setup procedure



Most simulations consist of two phases. In NetLogo lingo, these are usually called ‘setup’ and ‘go’. In the **setup procedure,** we create the starting population of agents and build their environment. The **go procedure** is the main simulation loop where we need to describe all processes that the agents and the environment undergo at each time step.

We will start with creating buttons that will be used to initiate ‘go’ and ‘setup’. Click on the ‘Add’ button at the top of the window and then click anywhere on the white space. A dialogue box should pop up. Write `setup` in the ‘Commands’ box and click OK. Follow the above step to create a second button and write `go` in the ‘Commands’ box. This time also tick the ‘Forever’ box. ‘Forever’ means that this action will repeat until the simulation ends.

You can see that the text on both buttons has instantly turned red indicating an error. Although you will find ‘setup’ and ‘go’ in virtually every Netlogo model the names are customary. This means that the Netlogo interpreter does not recognise these terms - we need to define them. Let’s move to the Code Tab to fix it. To define a procedure you need to mark it in a way that will ensure NetLogo knows it is a procedure and not, for example, a parameter. To do so we use the words `to` and `end`. Also, we can already specify one action that the simulation has to perform at each time step: tick. The tick command is like moving the clock forward, it marks the end of all procedures in this time step and starts a new one. Type the following in the code box:

```
to setup
end

to go
  tick
end
```

If you now click on the ‘Procedures’ button at the top of the screen, you will find that ‘setup’ and ‘go’ are listed there. Next to it is the debugger button ‘Check’ - if you click on it, it will check if the basic syntax of the code is correct. Try it!

We will start coding by creating a number of agents, in NetLogo lingo called ‘turtles’. We will also give them a few variables. There are two types of **commands** in NetLogo: procedures that the modeller defines themselves (just as we just did with ‘setup’ and ‘go’) and built-in procedures called **primitives**. Primitives save you time because you do not have to define everything yourself! You can find them listed and documented in the NetLogo Dictionary (https://ccl.northwestern.edu/netlogo/docs/dictionary.html).

We start the setup with the `clear-all` primitive which removes any remnants of the previous run of the simulation including plots and monitors. This way we always start with a clean slate. Similarly, `reset-ticks` sets our clock back to zero. If you look back at the pseudocode you will see that all we need to achieve here is to create a number of agents. We will use the `crt` primitive to do so - you can go to the NetLogo dictionary and check it out. We also need to locate them somewhere in the world. In this case, we choose the location to be at a random cell between cells (0, 0) and (5, 5). Type <u>inside</u> the setup procedure (i.e. between `to setup` and `end`**)**:

```
to setup
      clear-all
      crt 20 [
            setxy random 5 random 5
      ]
      reset-ticks
end
```

The brackets after `crt` (short for `create-turtles`) can also enclose features (variables) - of the agents other than its location, for example, `color`, `size`, `shape`. All of these are built-in variables and will have default values if none are assigned. But you can also customise them, for example, this way:

```
to setup
      clear-all
      crt 20 [
            set color random 140
            set size 1
            set shape "turtle"
            setxy random 5 random 5
      ]
      reset-ticks
end
```

If you are all done, hit the check button and move to the Interface Tab. Click on your setup button and see what happens. Can you see your agents clustered in the middle of the screen? You can change the parameters of the View Panel. Click on the Settings button in the top right corner. A window pops up in which you can change the point of origin, the size of the world and the size of each patch (cell). Change the point of origin to the bottom left corner and unclick the ‘world wraps horizontally’ and ‘world wraps vertically’ boxes. This will make our map a flat surface instead of an edge wrapped torus (a doughnut shape). Now all agents cluster in the bottom left corner.

Go procedure

All commands to agents in NetLogo are initiated by the word ask and enclosed in square brackets `[ ]`. In this piece of code, we will ask all turtles (`ask turtles`) to turn right (`rt`) by a random number of degrees up to a full circle (`random 360`) and go forward one step (`fd 1`). Write these procedures inside the ‘go’ procedure, before ‘tick’, so that it looks like this:

```
to go
    ask turtles [
         rt random 360
         fd 1
    ]
    tick
end
```

`fd` stands for `forward` and the number after indicates how far the turtle should move (the unit is the length of a patch) while `rt` stands for `right` and the number indicates the angle in degrees of the turn (if the number is negative the agent will turn left). Hit the check button and move to the interface tab to see what happens. You can use the speed slider at the top of the screen to make the simulation slower so you can see the turtles' movement pattern more clearly.

In Y&B model, there are two factors driving the wave of advance. One is random mobility, the other is population growth. Without the growth, the populations won’t be able to cover a larger region, instead they will just become butter scraped over too much bread. So next we will try out population growth through reproduction and then put the two together.

Introducing the if-statement

In the previous step, the movement happened with every tick of the model. We could do the same with reproduction, but we would rather be able to control how often reproduction occurs. This is where an if-statement come in. An if-statement consists of two elements: a *condition* and a *code block*. Code inside if-statements is only run if the *condition* is true. If the *condition* is false, the code inside is simply skipped.

Comment out the turn and forward lines by adding a semicolon at the start of the line before the commands. This tells NetLogo to ignore those lines which is usually safer than deleting them. Then add the following:

```
ask turtles
    ;rt random 360
    ;fd 1
    if random-float 1 <= pop_growth
       [ reproduce ]
  ]
```

The `random` primitive, we used before, selects an integer (whole number) up to but not including the number indicated (so `random 3` will choose at random from: [0, 1, 2]).

`random-float` instead selects a random decimal number, or floating-point number, up to the number you put after. It is commonly used to represent probability, where, for example, 0.20 stands for 20% chance. If you now click the 'Check' button, NetLogo will warn you that *pop_growth* is not defined. We will define it on the Interface Tab by making a slider. Switch to the Interface tab now, select slider from the drop-down list [fig], click Add, and then click somewhere besides your View panel to create the slider. Type `pop_growth` into the Global Variable box and define a minimum, maximum, and the increment, or amount of increase for every notch on the slider. We want `pop_growth` to be a percentage chance of reproduction, so we'll set the minimum to be 0, maximum 1, and increment to be 0.05. Set the value to whatever you like within the range, we will experiment with it later.

Now we need to switch back to the code tab to write our reproduce procedure. `reproduce` is not a NetLogo primitive, but we can use the `hatch` primitive in our definition. A hatched turtle is a clone of its parent and so will inherit all of its parent's characteristics, called variable states, like `color`, `size`, and `shape`. We'll add code within hatch to change `color` a little at birth just for visual effect. NetLogo's tools menu has a Color Swatches option so you can see how the numbers match up to the colours. To do this, add the following code below the go procedure:

```
to reproduce
     hatch 1 [
       set color color + 0.1
     ]
end
```

This line takes the old colour value, which was inherited from the parent, and adds to that value by 0.1. The first `color` refers to the variable we're setting, the second `color` is the current value of the variable. It's like saying, set my colour to my old colour of red plus a little brightness.

Check the code and test it out by clicking setup and then go. It won't look like much is happening because we commented out the movement lines before and we did not ask the hatched turtles to move. Instead, the new turtles inherit their parent's xy position making a bunch of turtle piles. We could simply uncomment those movement lines by deleting the semi-colons, but that wouldn't stop the turtles from crawling over each other on the same cells. Instead, change reproduce to match the following code:

```
to reproduce
     if any? neighbors with [count turtles-here = 0] [
          let empty-patch one-of neighbors with [count turtles-here =
      0]
          hatch 1 [
               set color color + 0.1
               move-to empty-patch
          ]
     ]
end
```

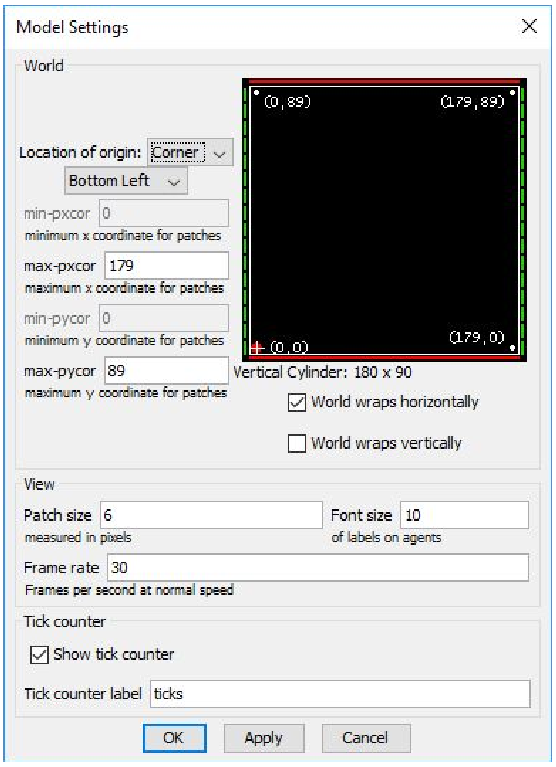
`neighbors` in a NetLogo primitive that checks the 8 cells that are touching the current cell [fig], there is also a `neighbors4` version which looks at only 4 cells (no diagonals). *`with`* followed by a condition in square brackets further reduces the number of cells being looked at by not looking at any that already have turtles on them. When it is put all together, this code checks to see if there are `any?` neighboring cells with no turtles on them assigns a random one of those cells to a temporary variable we made with `let` called *empty-patch*, and then later asks the newly hatched turtle to `move-to` that patch when it is born.

Young and Bettinger's model actually used a gradient to decide where to move new groups, but our version is a close approximation where hatched turtles random walk onto empty cells at birth. Note that here the turtles always move to cell centres, the turning (rt) and forward (fd) way we coded it earlier results in turtles moving all over the patches. We'll discuss the differences more in a later chapter.

For now, go back to the interface tab and run the model a few times. What patterns do you see? Do all turtles contribute equally to the dispersal wave? Why or why not? What happens if a turtle tries to reproduce but there are no empty patches? What happens when you change the population growth rate, do the higher values always translate into faster dispersal? (You can compare the number of ticks before a specific cell is occupied).

Extension

We got pretty far with Young & Bettinger's model before but we left out an important part, the landscape. A map is not just about giving shape to our dispersing population, there are also some significant barriers to dispersal like oceans, lakes, or ice sheets that we should account for. Dispersing populations will need to go around these areas and funnel through some geographic bottlenecks.


Model Settings

World

Location of origin: Corner

Bottom Left

min-pxcor 0
minimum x coordinate for patches

max-pxcor 179
maximum x coordinate for patches

min-pycor 0
minimum y coordinate for patches

max-pycor 89
maximum y coordinate for patches

(0,89) (179,89) (0,0) (179,0)

Vertical Cylinder: 180 x 90

☑ World wraps horizontally

☐ World wraps vertically

View

Patch size 6
measured in pixels

Font size 10
of labels on agents

Frame rate 30
Frames per second at normal speed

Tick counter

☑ Show tick counter

Tick counter label ticks

First though, we need an actual landscape. The map we're using was adapted from the Stage 3 project (ref). Make sure it is located in the same folder as your NetLogo model.

In order to bring this landscape into our NetLogo model, we need to recognize a few things:

1) We want the map image to be imported as actual data, not just a background image.
2) We then need extra agent code to stop movement into some patches.

If you have ever worked with Geographic Information Systems (GIS), you'll already have some experience with gridded raster data. If not, all you need to understand here is that the map is a grid of cells, or `patches` in NetLogo terminology, that each has a number corresponding to them that represents their colour value. All we need to do here is import our map into NetLogo so that numerical values will be assigned to each `patch`'s variable called `pcolor`. NetLogo has a command do this easily. Within the `setup` procedure, before we create any turtles, add the following line so that it looks like this:

```
to setup
      clear-all
      import-pcolors “ch1_map.bmp"
```

Click the setup button on the Interface. We can see the map now, but it does not fit our View panel very well. The map image we imported (if you check it’s file properties) is 334 x 212 pixels and we would like each pixel to be assigned to one patch. Click settings and set the max-pxcor to 333 and the max-pycor to 211. NetLogo counts the first column and first row as zero, so we need one less than the image dimensions to make it all line up. Look at the “Box: ” listing just under the black square in this Model Settings panel to verify that it’s all correct. Since this is the globe, also check the “World wraps horizontally” box and then click Ok (Fig). If your View panel is far too large now, re-open Settings and set the “Patch size” setting to a smaller value until the View fits. This setting is just for how it looks on your screen, it doesn’t change the number of patches.

Click Setup again and verify that the map fits well now. Run the model again and you’ll notice a couple of things: 1) it’ll take a lot longer to finish the run because there are a lot more agents and patches now, and 2) the agents are completely ignoring our map.

Young & Bettinger’s dispersal model keeps things simple by having patches be coded as either one for land or zero for water or ice sheets. In our case, the map we imported has water all the same colour value. As you saw earlier when we looked up NetLogo’s Color Swatches tool, colours are represented by numbers from 0 to 140 but you can also refer to them using `color` keywords like `white`, `green`, `sky`, or `violet`. All we need to know is the pcolor that is assigned to water on the imported map. If you can’t see the map on your View panel, click setup. Then right-click on any water patch and click *Inspect patch X Y*, where X and Y correspond to the coordinates of the water patch you clicked on. An inspection panel will come up that provides variable values of that patch, as well as a close-up view of the patch, note that the `pcolor` value is 9.9 which is the value for `white`.

We have two things left to do, move the turtles’ start location to eastern Africa and then tell the turtles that they can only move on land. First, in the View panel right-click on a cell in the heart of Kenya and note its x,y coordinates. In `setup`, change the x,y coordinates inside the `crt` command to `(your_X_coordinate + random 5)` like this:

```
 setxy ( 110 + random 5 ) ( 90 + random 5 )
```

The parentheses are not actually necessary, but they help clarify what you’re trying to do for any future readers of your code (like yourself six months from now). Next, to restrict the turtles' movement to land only change the following code to limit the set of patches the turtles are evaluating for their offsprings:

```
 to reproduce
   if any? neighbors with [count turtles-here = 0 AND pcolor != white] [
     let empty-patch one-of neighbors with [count turtles-here = 0 AND
 pcolor != white]
      ...
```

The ! before the != means “does not equal” which tells the agents **not** to consider any `white` cells for their offspring. The `AND`, known as a logical operator in computer programming, is a way of combining criteria within the *condition* so that both have to be true for a patch to be considered.

Check your code, then switch to the Interface tab to `setup` and run your code a few times. What do you notice about the dispersal wave as it passes through geographic bottlenecks like the Sinai Peninsula or south of the Black Sea? What pattern do you notice in the agent colours and what could this represent?

Summary

The **simulation** we built in this chapter is a **model** of human dispersal. It is a model because it is a simplified version of the actual behaviour of day-to-day mobility and reproduction decisions made by human groups. Many possible factors influencing dispersal were deliberately left out of our model because we are using it to test the hypothesis that the pattern and speed of human dispersal across the world were primarily driven by population growth and undirected movements. All that is left for us to do to test this hypothesis is to **parameterize** the model by assigning reasonable values to our population growth rate and movement distance relative to the time step and then run the model. If the arrival date of the simulated population arrives in Israel, Italy, China, Indonesia, Alaska, Chile and other locations around the date of the known first occupation in those regions then we will have supported the hypothesis.

Our simulation results may also lead to theory building if it exposes particular patterns in the spreading dispersal wave. It can also nudge the researcher to go back to the archaeological record and to explore additional mechanisms. For example, your agents will likely arrive in France before China. This could be because of differences in population growth rates or carrying capacity as Y&B suggested, or it could have been due to competition from Neanderthals.

Models are not static end products in the research process, but a part of the cumulative understanding we are building about the past at both a theoretical and empirical level (Dunnell 1982; Hesse 1978). Models are also unlikely to prove your hypothesis exactly correct. A common saying in modelling in the sciences, attributed to statistician George Box (1979), is that 'all models are wrong, but some are useful'. In other words, while you are unlikely to replicate the exact timings of dispersal across the world, hopefully, the model will provide insight into what is and is not a likely factor driving dispersal. Even better is when a model, built according to a common narrative explanation, is completely wrong and we have to learn something new.

Finally, it is worth reflecting on the technical side of things. If prior to reading this chapter you had never seen a line of code and the whole programming enterprise felt daunting please spend a moment to contemplate the fact that after only a couple hours of basic training you are able to replicate a published scientific study. It truly is not rocket science.

End of chapter exercises

1. Easy: change the agent shape.
2. Easy: Add a second population of turtles spreading from East Asia.
3. Add a slider 'initial-pop-size'. Use it to vary the size of the original population.
4. Difficult: Add more population dynamics by adding age and death after a certain age. Step two: add the probability of death (similar to reproduction-probability). Define this probability as a function of age, so that the older a turtle gets the higher their probability of dying.
5. Y&B use a variable 'carrying capacity' to indicate which areas can and which cannot be occupied. Give all patches variable called 'carrying-capacity' [check 'patches-own']. For land

patches set it to 1, and for water patches set it to 0. Modify the reproduction procedure to only allow turtles to hatch onto empty patches where carrying-capacity is equal to 1. Step two: use the following table to give different carrying capacity to different biomes.

Citation:

Romanowska, I., C. Wren & S. Crabtree, in prep. 'Agent-based modelling for Archaeologists."

Further reading

The Fisher-Skellam-KPP equation has been widely applied in archaeology to a number of the classic case studies of dispersal.

New World:

Young, D. A., & Bettinger, R. L. (1992). The Numic spread: a computer simulation. *American Antiquity*, *57*(1), 85–99.

Anderson, D. G., & Gillam, J. C. (2000). Paleoindian Colonization of the Americas: Implications from an Examination of Physiography, Demography, and Artifact Distribution. *American Antiquity*, *65*(1), 43–66.

Homo sapiens into Eurasia:

Bettinger, R. L., & Young, D. A. (2004). Hunter-Gatherer Population Expansion in North Asia and the New World. In D. B. Madsen (Ed.), *Entering America: Northeast Asia and Beringia Before the Last Glacial Maximum*. University of Utah Press.

Homo erectus into Eurasia using a slightly different cellular automata approach:

Mithen, S. J., & Reed, M. (2002). Stepping out: a computer simulation of hominid dispersal from Africa. *Journal of Human Evolution*, *43*(4), 433–462.

Nikitas, P., & Nikita, E. (2005). A study of hominin dispersal out of Africa using computer simulations. *Journal of Human Evolution*, *49*(5), 602–617.

Hughes, J. K., Haywood, A., Mithen, S. J., Sellwood, B. W., & Valdes, P. J. (2007). Investigating early hominin dispersal patterns: developing a framework for climate data integration. *Journal of Human Evolution*, *53*(5), 465–474. https://doi.org/10.1016/j.jhevol.2006.12.011

Neolithic agriculture

Ammerman, A. J., & Cavalli-Sforza, L. L. (1973). A population model for the diffusion of early farming in Europe. In C. Renfrew (Ed.), *The Explanation of Culture Change: Models in Prehistory* (pp. 343–358). London: Duckworth.

Pinhasi, R., Fort, J., & Ammerman, A. J. (2005). Tracing the Origin and Spread of Agriculture in Europe. *PLoS Biol*, *3*(12), e410. https://doi.org/10.1371/journal.pbio.0030410

Fort, J. (2018). The Neolithic Transition: Diffusion of People or Diffusion of Culture? In *Diffusive Spreading in Nature, Technology and Society* (pp. 313–331). Springer, Cham. https://doi.org/10.1007/978-3-319-67798-9_16

For more theory on the mathematics of the basic equation and its many variants:

Steele, J. (2009). Human dispersals: Mathematical models and the archaeological record. *Human Biology*, *81*(2–3), 121–140.

For more theory on modelling dispersals with ABM and other simulation techniques
Romanowska, Iza, "So You Think You Can Model? A Guide to Building and Evaluating Archaeological Simulation Models of Dispersals" (2015). Human Biology Open Access Pre-Prints. 79. http://digitalcommons.wayne.edu/humbiol_preprints/79