A Primer on Mathematical Modeling with NetLogo

Glenn Ledder

January 2026

Contents

1 Introduction: Agent-Based Modeling and the NetLogo Programming Environment

This chapter-length introduction to mathematical modeling with NetLogo is what I would want my students to know before starting to do a research project with me, and I am writing it because other tutorials inevitably cover considerable material that my students would not need while not including topics that they do need. Chief among these is how to do data analysis in NetLogo. NetLogo is not well designed for data analysis, instead being designed with the assumption that NetLogo programs will write their output into a spreadsheet file, which the user will analyze using some other software. In my view, it seems wasteful to do anything manually that could be done automatically by a computer program.

Any background in mathematical modeling, programming in NetLogo, or other programming would be helpful in understanding this material; however, I have attempted to make it as self-contained as possible.

1.1 Deterministic and Stochastic Mathematical Models

The focus of these notes is on using NetLogo for mathematical modeling rather than on teaching mathematical modeling per se. I have written extensively on mathematical modeling, so the reader can find much more on this topic elsewhere [**?**]. Here, I feel it necessary merely to summarize the fundamental idea of mathematical modeling, which is that mathematical models have a structure similar to that of a function. There are one or more input quantities along with a rule or set of rules that determine one or more output quantities. The goal of studying a mathematical model is to characterize how the output quantities depend on the input quantities. The difference between a mathematical model and a function is that functions normally have clearly defined formulas that make it easy to see the connections between output and input quantities, whereas mathematical models require another mathematics problem to create the connections. Figure 1 illustrates this structure, with the underlying math problem described as the "narrow" view of a model and the mapping from model inputs to model outputs described as the "broad" view.

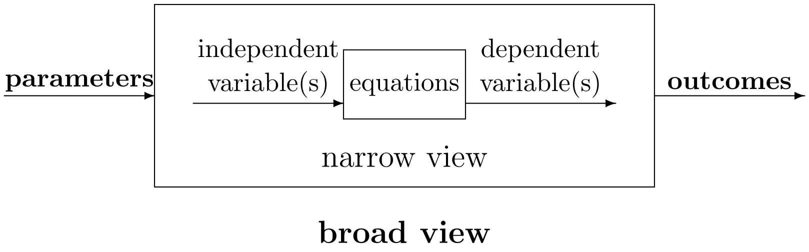

Figure 1: Narrow and broad views of mathematical models.

Consider the SIR epidemic model as a simple example. The underlying model is a set of two differential equations for variables S and I, which represent the population fractions of susceptible and infectious individuals,

$$\begin{aligned} \frac{dS}{dt} &= -\beta SI \\ \frac{dI}{dt} &= \beta SI - \gamma I, \end{aligned} \tag{1}$$

along with initial conditions $I(0) = I_0$ and $S(0) = S_0$, with $I_0 + S_0 \leq 1$. The variables in the narrow view are t, $S(t)$ and $I(t)$. The parameters β, γ, I_0, and S_0 function as constants in the underlying math problem, but they are the independent variables in the mathematical model. The output quantities depend on the goal of the modeling project. For the SIR epidemic model, we might want to know the population fraction S_∞ that remains susceptible as $I \to 0$ or the maximum value I_m of $I(t)$.

Deterministic differential equation models are abstractions of real-world settings, in which broad features might be deterministic, but fine-grained features are random. The parameter γ in (1) is the reciprocal of the mean duration of the infectious period. In reality, the duration of the infectious period is a random parameter, at best governed by a known probability distribution. Using the average rate γ in place of the actual probability distribution may well allow the model to adequately approximate output quantities such as S_∞, but these output quantities are also governed by probability distributions, and the best the deterministic model can do is provide an expected value for output quantities.

The use of the differential equation system as an SIR model tacitly assumes that the underlying probability distribution for the recovery process is exponential. Without providing details, suffice it to say that the exponential distribution is a very poor model for recovery times. This calls into question any conclusions about real SIR settings that can be drawn from the flawed deterministic model.

There are alternatives to the deterministic model. One of these is an *agent-based model* (ABM)[1], in which the recovery times for different individuals in a database are drawn randomly from a probability distribution with mean $1/\gamma$ and whatever distribution shape is deemed to fit real recovery time data best.

1.2 Why Read These Notes?

When I was first looking for a programming environment to use for agent-based modeling, I set this list of requirements:

1. The programming environment should have built-in commands that simplify database manipulation.
2. The programming environment should have simple commands to incorporate spatial structure, both for the database and for visual display.
3. The programming environment should allow for straightforward data analysis, including statistical calculations and curve fitting.

This did not seem to be too restrictive a set of requirements, but I was unable to find any programming environments that met all three. As of early 2026, I still don't know of any. Conventional programming environments, such as Matlab and R, can support database manipulation, but not easily. The code required to select all user records that have a certain attribute requires loops and other sophisticated programming. It is even more difficult to use a conventional programming environment to support spatial structuring. I presume that a Matlab expert could write code that identifies the near neighbors of any one agent and that has a visual display of an animation. Not many mathematical modelers have these skills or want to learn them.

The obvious candidate for my requirements in NetLogo. NetLogo was designed for database manipulation and visual display. The language structure makes it very easy to identify all agents with a particular property, such as having a Boolean variable be true or being within a specific distance of a specified agent. It is the third requirement in which NetLogo falls short. Programming an experiment that directs multiple runs of a simulation, calculates some property of that simulation, records and displays the aggregate results, and then fits the results to an empirical model is much harder in NetLogo than in conventional programming environments. There is an add-on product, called BehaviorSpace, that is designed to do some of these functions, but not the data analysis.

If what you want to do is perform multiple runs with a sequence of parameter values, save the data into a file, and then use some other software to analyze the data, then NetLogo plus BehaviorSpace solves all your problems and you could discard some of the sections in these notes. But if you want to be able to run all the simulations and then click a button to implement a curve-fitting algorithm and plot the best-fit curve, then you will likely be disappointed, as was I, if you expected that the curators of NetLogo would have included this sort of programming in their otherwise excellent models library.

In my journey, I could find no examples of automated data analysis within a NetLogo program. This was not too surprising once I delved into the NetLogo dictionary and discovered how difficult it was (and still is in early 2026) to do regular scientific computation with NetLogo. The syntax for calculations may be convenient for people whose computational needs are limited, but there are no solvers for differential equations or linear algebraic systems, and the idea of writing such code for NetLogo is daunting.

---

[1] also called an individual-based model (IBM)

In spite of the difficulties of using NetLogo for what I wanted, there was no reasonable alternative because it is so easy to do agent-based modeling with spatial structure in NetLogo. I felt that what I wanted to do must be possible, if obscure. I had sufficient background in scientific computation and sufficient motivation to tackle the problem of how to get NetLogo to do these things that it was not well designed to do. Perhaps there are solutions other than mine out there or they could yet be written by someone who is more expert at NetLogo than I am. For those whose goals are similar to mine, who are not afraid to dig into program code, and who mistrust canned software solutions, I offer my solution for their adoption.

It will be some time before we get far enough along to add data analysis to a NetLogo program. The reader can find other tutorials on NetLogo. These will not be focused on the specific goal of doing parameter studies on agent-based mathematical models, so they will include instruction in a lot of topics that are peripheral to that goal. This is why my notes start with the very basics of NetLogo coding, suitable for NetLogo novices.

The best way to learn any programming language is the "Rosetta stone" method, where one first studies model programs and then adapts their features to the programs one wants to write. Help files packaged with the software are not always useful, as they often illustrate a code element with the aim of showing the most sophisticated things it can do rather than showing an example that presumes the reader did not already know more advanced programming than the level of the command being explained. While it is (barely) possible to use the "Rosetta Stone" method with a complete program, it is far simpler to do it with a graded sequence of semi-finished programs that break the complete program into bite-size lessons. Programs with built-in flaws are better than ones that do what the programmer intended. The reader will find several flaws deliberately written into the semi-finished programs, although hopefully there aren't any in the two finished programs.

In these notes, we are going to study foraging behavior. This works well for a first example because there is a clear spatial structure, but only a few simple rules are required to describe the behavior of the agents. We will need only two types of agents: foragers and resource patches. In NetLogo, there is one set of language structures for the spatially defined patches and another for the agents that occupy patches, referred to in NetLogo as "turtles". Aside from the built-in attributes for location and color, our patches will require only two items, the resource level and the status of being visited or not visited. Turtle attributes required, in addition to the built-in attributes of location and direction, are the net level of resources-collected-minus-resources-spent and an attribute that indicates the status as "moving" or "not moving". We'll start with a program that merely creates the world for the simulation, and then we'll gradually add the rules that govern the changes to the database at each time step and an animation to modify the visual state of the system.

2 forager.1: Building the Virtual World

The NetLogo programming environment consists of a visual interface and a block of code, which contains most of the programming. The interface for forager.1 contains a visual display of the virtual world and a single button, labeled **setup**.

Start by clicking the setup button on the interface a few times. The setup code creates a virtual world in which some random patches (those with resources) appear green and the other patches are light brown. The turtles[2] could appear anywhere in the grid. While at the interface, right-click the setup button and choose **Edit** to bring up a dialog box with the programming for the button widget. Note that the command box contains the word **setup**. This tells us that the code will contain instructions in a procedure called **to setup**; these instructions are executed when the setup button is clicked. The Display name box is empty, so the button label is by default the same name as the command. We'll return to the interface after looking at the code.

The first thing to notice about the code is that there are sections delineated by long strings of semicolons. Everything in a line after a semicolon is a comment, so these long strings are unnecessary. I have used them to provide a visual structure. Shorter strings delineate blocks of code within the larger structure. The middle section contains all procedures that are linked to interface buttons. The bottom section contains procedures that are called by other procedures. The top section contains some definitions in three groups,

---

[2] The foraging programs use a stylized drawing of an insect larva to represent the agents, but the agents are still referred to as "turtles".

labeled **globals**, **patches-own**, and **turtles-own**. These three groups are required at the beginning of any NetLogo program.

Globals are any quantities that are not associated with particular patches or turtles or defined in the interface. Their values need to be assigned in a line of code in one or more procedures. Patches and turtles have many pre-defined attributes; for example, the turtle attribute **color** and the patch attribute **pcolor** both specify a property of a given turtle or patch. These need not be the same for all turtles or patches; we've already seen that patches can have different colors. Other attributes needed for the simulation rules need to be defined here. The patch attribute **visit?** and the turtle attribute **move?** are Boolean (true-false) attributes that indicate whether a patch contains a turtle and whether a turtle is moving or not. Note that NetLogo is very flexible about variable names, allowing some punctuation marks not allowed by most programming languages. It is common practice to end the name of a Boolean variable with a question mark. The patches and turtles also have a real number attribute that indicates the energy level. These could be named anything, but the use of **penergy** for patches is modeled after the use of **pcolor** for patch color.

The procedure called **to setup** executes the instructions needed to create the virtual world of the model. One could put all the setup details in this procedure, but it is better programming practice to instead put the details in subsidiary procedures and use **to setup** simply to call those procedures. Two of the subsidiary procedures, color-coded in blue, are pre-defined NetLogo code; one of these clears all old variable values and the other restarts the simulation clock at 0 ticks. The user-defined subsidiary procedures are used to set the parameters, build a skeleton virtual world, and populate that world with resource patches and turtles.

The user-defined procedure **to set-parameters** has three commands, each of which assigns a value to one of the globals. Note the syntax of the **set** command. Unlike most programming languages, NetLogo does not use an operator character, such as an equal sign or an arrow, to assign values to names. Instead, blank spaces are used. Blank spaces serve other purposes as well, and this can be a source of confusion because the meaning of a blank space is determined by the context. For the **set** command, the item after the first space is the name of the variable whose value is being set and the item after the second space is the value to be used. Such values can be given as numbers or names of previously-defined quantities. For example, replacing **set n-t 10** with **set n-t n-p** would in this case set **n-t** to whatever value had previously been assigned to **n-p**.

The procedure **to reset-experiment** builds a blank virtual world in which all patches are empty and there are no turtles. To do this, the user-defined attributes **visit?** and **penergy** are given default values of false and 0, respectively, and the built-in attribute **pcolor** is set to a light brown. The code in this procedure shows one of the features of the NetLogo language that is very convenient. We want to apply the three **set** commands to all the patches. In many languages, we would have to loop over sets of indices. In NetLogo, **ask patches** means that all the commands in the subsequent block will be applied to all patches.

The procedure **to-populate** adds resources and turtles to the blank virtual world of the **reset-experiment** procedure. New color and energy values are to be added to n-p of the patches, marking these as resource patches. To select the patches for this change, we have the command **ask n-of n-p patches**. Unlike **ask** blocks, which are applied to all patches, **ask n-of** blocks apply only to a randomly selected set of patches.

Whereas patches are pre-defined, turtles have to be created using the built-in **create-turtles** command. The command has one blank space, followed by a constant or variable indicating the number of turtles to create. The following block assigns values to the attributes of the turtles so created. The default shape of turtles is an arrowhead and the default color is random. Here, for better visual sense, the "turtles" appear as pictures of insect larvae, chosen because the graphic is sharply delineated and because insect larvae are extensive foragers. Individual patches might represent leaves on a plant. The spatial coordinates of the turtles are assigned randomly using **setxy**. It remains only to set the user defined attributes. All turtles are initially assigned an energy level of 0 and a status of moving. However, this is not appropriate if the turtle happens to start in a resource patch. Another nice feature of NetLogo is that turtle parameters include the parameters of the patches they reside in. So **if penergy > 0** selects those turtles that have been placed randomly into patches that were previously selected as resource patches. For these turtles, the movement status must be changed to false and the visit status of the associated patch set to true.

Back at the interface, you can right-click on any turtle and choose to inspect it. This will show you the attributes of that turtle, including many that we did not include in the code. Click the setup button until a turtle is randomly assigned to a resource patch. Inspecting several turtles will allow you to observe the attribute differences between the turtle in a resource patch and turtles in empty patches. You can also inspect patches to see the differences between empty patches, unoccupied resource patches, and occupied resource patches. It is always good programming practice to stop frequently to confirm that the new code does what you intended it to do.

One last thing to observe is the Settings window, which you can see by clicking the appropriate button at the upper-right corner of the interface. One can choose different patch sizes and different numbers of patches. Here, I have chosen a patch size of 16, which is larger than the default size, so that the animation is easier to visualize. To compensate for this, I have chosen a grid that is 21 by 21, which is smaller than the default. One can also choose between the default settings of having the world wrap horizontally and/or vertically or have actual boundaries. Wrapping simplifies things by making all patches interior patches.[3] Boundaries may be important for some simulations, but they often require special movement schemes for turtles that bump into a boundary.

3 forager.2: Animating the Virtual World

The interface for forager.2 includes two new buttons, labeled **go once** and **go**. The **go** button has an additional icon on the right side that indicates repetition. If you right-click on these buttons in turn, you will see that both are linked to a **to go** code block. You will also see that they are disabled until ticks start, which is why they are initially grayed out. The difference between them is that the **forever** box is checked for the button labeled **go**.

Click the **setup** button and then click the **go once** button several times. Then click **go**. After a number of ticks pass, it becomes clear that the turtles never leave the first patch they enter. (Once the turtles are all trapped in patches, you can stop the animation by clicking **go** again.) Of course this is a program design error. Flawed programs are often more instructive than correctly designed programs.

In the code, we have added several new globals that are needed for the animation. The **to setup** procedure is unchanged, as are its subsidiary procedures, except that **to set-parameters** now includes commands to set the values of the new globals. The new **to go** procedure begins with the **tick** command to advance the simulation clock and then contains two user-defined subsidiary procedures.

The feed procedure increases the energy of turtles that are not moving. These are selected from the full set of turtles by the code **with [not move?]**. Note that the energy change command uses **energy + dt** to calculate the new energy. The spaces here are a required part of any calculation. The parentheses are not required, but have been included to decrease confusion. The code **set energy energy + dt** is acceptable NetLogo syntax, but you have to remind yourself that the set command requires two arguments separated only by spaces, one for the name of the quantity being set and one for the value assigned to it. Without the parentheses, it looks like the set command has four arguments rather than one because the spaces in the calculation are visually indistinguishable from the spaces in the set command.

The turtle movement procedure is similar in structure to **to feed** in that it selects a subset of the turtles and changes some of the attributes of those turtles. Each turtle moves a distance $v\,dt$ in its current direction and has its energy decremented by a maintenance cost. The final command checks whether the new location is a resource patch, in which case the turtle is marked as not moving and the patch is marked as being visited.

You can now see why the turtles are stuck in the first patch they enter. There is no code to check that the patch still has enough resources. There isn't even any code that changes the resource level of a patch, so in forager.2 all of the resource patches have their full amount of resource for the entire animation.

It is necessary to understand how time works in NetLogo models. Simulation time is measured in ticks, but there is no reason why one tick should correspond to one unit of time as defined by the underlying model. In this program, I have divided each unit of model time into 20 ticks; therefore, each tick corresponds to 0.05 units of model time. With a velocity of 10 distance units per model time unit and just 0.05 units of model time for each tick, the turtles move only 0.5 distance units in a tick. Because this number is less than 1, there is no chance that a turtle will move through a patch without stopping. Each tick will either move the turtle to the next patch or move the turtle to a different point within the same patch.

---

[3] Technically, wrapping gives the world the shape of a torus rather than a flat plate. If you keep going in either coordinate direction, you come back to where you started rather than reaching an edge.

4 forager.3: Monitoring Progress

The changes from forager.2 to forager.3 are about monitoring the progress of the simulation, both for the turtles and for the patches. The interface has two new elements, a plot window and a monitor window. Both of these track the mean turtle energy per model time. Recall that turtle energy increases while visiting a patch because of feeding and decreases while moving because of maintenance costs. The number that appears in the monitor window will change accordingly, and the graph of mean energy per time will keep a historical record. Click **setup** and then **go** to see these new features in action. Run for about 1000 ticks this way. The mean should still be changing noticeably.

If you keep watching for several thousands of ticks, you will see that eventually all or nearly all of the patches are depleted. The turtles will keep moving, but eventually their mean energy per time becomes negative. Something is still missing from the model. The patches now decrease in resource level as a result of feeding, but they do not make any recovery afterward. The total amount of energy available through food is fixed, while the amount of energy lost while moving is unlimited. The model is realistic for some settings, but in our setting we want the model to be sustainable. We'll fix that in forager.4.

Before turning to the code, right-click on the monitor element of the interface and choose **Edit** so that you can inspect the element. The quantity being reported represents the mean energy per unit time, but for brevity, the associated variable is called **mean-energy**. In the code, you will find this new variable listed among the globals.

There are two other new globals. One of these is **X**, described as the "giving up patch energy". This is the energy level at which a patch will no longer be considered usable for feeding. Notice that the **set-parameters** procedure assigns values to both new quantities. **mean-energy** is actually a variable, so the assignment merely gives it an initial value. Eventually, we will want our program to treat **X** as an independent variable that is "chosen" in some sense by the turtles, but for now **X** is a fixed parameter, like all globals carried over from forager.2.

The go procedure now includes three subsidiary procedures that are new to forager.3; these are **to check-food**, **to set-color**, and **to update-result**. Before looking at each of these, note that **to feed** now includes an instruction that decrements the patch energy as the turtle feeds.

The change in the energy of patches being visited eventually triggers code that causes the turtles to move away. This is in the new **check-food** procedure that is run at the beginning of each animation step. The built-in **ask with** construction allows the program to make changes in the attributes of all turtles that are not moving and occupying patches whose energy level has fallen below the giving up level of **X**. These turtles are sent on their way with a change to their **move?** attribute and their departure from the patch is recorded by setting the **visit?** attribute of the patch to false. There is one additional command that randomly changes the turtle's movement direction. The command is easy to interpret. A random number between 0 and 89 is added to -45 to create a random number between -45 and 44. The turtle then turns to the right or left by up to 45 degrees.

The remaining new procedures provide output for the observer. **to set-color** assigns a new color value to each of the resource patches. Many of these have the same level of resource as in the previous step, but it is easiest to set the color of all the resource patches. The formula **53 + color-slope * (1 - penergy)** calculates a numerical value that is then interpreted by **pcolor** as a particular hue. The quantity **color-slope** is a global set to the value $5/(1-X)$. Since **penergy** can range from **X** to 1, the quantity being added to 53 ranges from 5 when **penergy** is **X** to 0 when **penergy** is 1. Thus, the color will be a linear scale with 53 for a full patch and 58 for a depleted patch. A look at the NetLogo manual shows that color 55 is a neutral green, with higher/lower numbers in the 50s representing progressively lighter/darker green hues. This scheme makes it easy for the observer to see patch energy in the interface grid.

The procedure **to update-result** calculates the mean energy per unit model time in three instructions. The whole formula could be combined into a single instruction, but that would increase the likelihood of coding errors.[4] The first two instructions calculate variables that appear only in this one procedure. It is not good practice to make such variables global. They need not be, provided they are assigned using **let** rather than **set**. The locally-defined quantities are the average of the energies of all the turtles and the total amount of model time represented by the simulation time. **mean-energy** is then set as the ratio of the two. As in most NetLogo instructions that involve calculations, the parentheses in each of these lines are optional;

---

[4] No prizes are given for the code that consists of the fewest lines; the only prize is for code that contains no errors.

I include them so that it is more clear what the spaces mean; for example, the first space after **time** is the space that separates the name of the variable and the value, while the spaces inside the parentheses of that instruction are the spaces required to separate quantities in a formula.

5 forager.4: Adding Sustainability

Many realistic foraging systems have a natural end time, such as when adults have collected enough energy to reproduce. Here, we want a simplified world in which foraging is the only activity for the animals that live there; hence, we need to make the world sustainable by adding a model component that allows depleted patches to recover. This change appears in forager.4 as the procedure **to grow-patch**. All patches that contain some resource and are not being visited grow at a fixed rate, but with total energy limited to 1. The code required to achieve this involves two new NetLogo primitives. The **min** primitive selects the smallest value out of a list. The code in the parentheses after **min** defines the desired list to have two items, one that is the calculated new energy value and the other the allowable maximum of 1. So once **penergy + r * dt** reaches 1, the new patch energy will be set at 1 and will remain at that level until the next visit.

An additional change in forager.4 is a feature that will eventually allow the model to be used to address a question. There are many versions of the optimal foraging problem, in which a foraging model contains a parameter that represents the time or resource level at which the forager abandons a patch. The question to be addressed is what value of that parameter yields an optimal result. In mathematical terms, the foraging model is cast as a function with an independent variable representing the giving up time or resource level and a dependent variable representing the mean energy gain per unit time over a long time period. Here, we have defined the giving up resource level to be a quantity **X**. This quantity is chosen in the interface rather than in the code, allowing the experimenter to compare simulation results with different values of giving up resource level.

The desired output quantity is not quite the same thing as **mean-energy** because that variable changes on every tick. As a practical solution to this problem, we can run each simulation long enough for the mean energy per unit time to level off. The value given in the monitor window then approximates the long-term average. We should not feel confident about the third digit in this quantity, as it would have changed in one direction or the other with a few more hundreds of ticks, depending on where the turtle(s) are when the procedure is stopped. It seems to take about 20000 ticks to obtain ongoing values of the mean energy per unit time that stay in a range of perhaps 0.01. For the default **X** of 0.3, this value is approximately 0.37. You are seeing a good illustration of the importance of stochasticity (randomness) in biology. In chemistry, stochasticity is not generally noticed because the numbers of molecules in a small quantity of a substance is astronomical. In biology, where populations are seldom higher than millions, that little bit of randomness makes a noticeable difference. Real biology experiments require multiple runs and averaging of results; even then, it is important to recognize that the exact value of the average could be a little different in a repeat of the experiment.

The model of forager.4 still has flaws. While each run looks satisfactory, see what happens when you vary the value of **X**. Lowering **X** always improves the mean energy per unit time. A thought experiment can help you appreciate that this is not realistic. Suppose you are in an apple orchard to pick apples. You start at one tree and stay there so that you don't lose time by moving. First you pick all the low-hanging fruit. Eventually you have to jump to reach the closest apple. Then you would need to climb a ladder. This is not what you would actually do. Once it gets difficult to reach an apple, you would just move on to another tree. The point is that the more depleted a resource patch gets, the harder it is to harvest the next unit of resource. In forager.4, the harvest rate is always 1 resource unit per unit of model time, so there is no penalty for staying until a patch is empty. In addition, completely emptying a patch would in reality mean that the patch could not repopulate with resource. This is another error in the model of forager.4: the patch growth rate does not depend on the current resource level. A better model would have the growth rate be low when a patch is nearly empty because there is little resource available for reproduction and also low when a patch is nearly full because the patch is nearly saturated. The final version of the forager program will fix these modeling errors. It is all too easy to focus exclusively on simulation coding, but if you want your simulation to have any scientific value, you need to use the best model you can get.

6 forager: The Corrected Model

As noted previously, the model used in forager.4 is flawed because all relationships were assumed to have fixed rates. This is wrong both for patch growth and for feeding. A better growth model is the logistic growth model, given as

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right). \tag{2}$$

The constant K in the model represents the size of the patch when it is fully populated. This is **x0-p** in the NetLogo code. If starting from a small initial value, the population growth will be roughly proportional to population size x initially; as the population approaches its maximum of K, its growth will be roughly proportional to population deficit $1 - x/K$.

The feeding model is flawed because in reality it gets more difficult to harvest a resource as the resource level decreases. With E as the turtle energy, our new model is

$$\frac{dE}{dt} = \frac{qx}{a + x}, \tag{3}$$

where x is the patch resource level, q is a scale parameter, and a is the semi-saturation constant, so named because when $x = a$ the harvesting rate is half of what it would be as $x \to \infty$. This is the Holling type 2 predation rate model. Detailed information about this and the logistic growth model can be found in the Ledder mathematical biology book [2]. While the logistic growth and Holling functional response formulas could be written directly into the commands in which they are used, it is better programming practice to encode these formulas as functions that could then be reused in other procedures or programs. NetLogo doesn't use the term "function", instead using the term "reporter" for a procedure whose job is to use data passed in from a calling procedure to calculate results that are then passed back to that procedure. These are defined using **to report** and must end with "**report** result", where "result" is either a single quantity or a list of quantities.

One additional change has been made to the code, which is to include the resource growth in the **feed** procedure, since growth is assumed to occur at all times. This will not change the results very much because visits are short and infrequent.

The behavior of the model now looks appropriate for a sustainable optimal foraging model. Patches are depleted quickly when visited and then recover, slowly at first, then faster, and then slowly again. The mean energy per unit time stabilizes at around 10000 to 20000 ticks. If you experiment with different values for the giving up level, you will find that turtles are less successful if this value is too high or too low. In the former case, too much time is spent traveling; in the latter case, the harvesting rate becomes too low to justify staying longer. With the default parameter values, the optimal value of X is about 0.2.

We could learn more about the forager model by varying some of the parameters and seeing what effect this has on the optimal giving up level. To do this with forager is very slow, as you would have to modify the code for each change of parameters and then run the model with various giving up levels in order to see what level is optimal with the current parameter set. Given the power of computers to execute logic and computation, this is way too much manual labor. With forager completed, our next task is to automate some of the computation required to do these experiments. This will be done in stages, starting with a program version that automates the computation of mean energy results for a sequence of giving up levels, and concluding with a more sophisticated version that also automates the process of computing optimal giving up levels over a range of values for one parameter. NetLogo has built-in features that can accomplish a limited version of what we want; however, to achieve the full power of automation will require careful use of the NetLogo language to do things that the language is not designed to do.

7 optimal_foraging.1: Automating Determination of an Infinite-Time Approximation and Saving Data for Multiple Runs

Eventually, we will want an optimal foraging program that automates the calculation of the long-term mean energy gain rate, the selection of **X** values, variation in the value of another parameter, and the calculation of the optimal giving up level. As a first step, optimal_foraging.1 automates the calculation of the long term mean energy gain rate. The user still has to specify a value of the giving up level **X** for each simulation run. Additional automation will gradually be added to subsequent versions. To see the result in action, click **setup** and then **go** in the interface. Each simulation run takes 20000 ticks, so it is necessary to move the **Model Speed** slider all the way to the right.

To automate the result calculation, we simply add an instruction to the **go** procedure to stop at a given tick count, named **runtime** in the program. We choose a tick count large enough to obtain fairly reproducible results, but not so large that each simulation run takes too long. This program also tricks NetLogo into saving results from each simulation run while redoing the simulation for other values of **X**. The problem is that resetting the tick counter deletes the values of all global variables. We want global variables associated with a given simulation run, like the energy of a patch, to be reset for the next run, but we also need global variables that carry simulation results (the independent variable **X** and dependent variable **mean-energy**) through all runs. These are **xlist** and **ylist**. They cannot be set in **to reset-experiment**, which is run for every new simulation. Instead, they are set just once, directly in the setup procedure, and then updated at the end of each simulation run.

The procedures that reset the experiment and populate the world appear in the code for both the setup and go procedures so that they will run at the beginning of each simulation as well as before the first simulation. **to reset-experiment** requires one additional instruction to create a blank world. The code that created the patches is fine as is because it overwrites any old values, but turtles from an earlier simulation will still be present without an additional instruction. This is provided by an instruction that asks all current turtles to **die**, removing them from the database.

In addition to the procedures that recreate the experiment world, the go procedure needs a subsidiary output-and-save procedure to save the results from each run and report those results both as text and as a point on a plot. The procedures **to reset-experiment**, **to populate**, and **to output-and-save** need to be executed only once for each simulation run, while the subsidiary procedures that run the simulation, such as **to feed**, need to be executed on each tick. This is accomplished by the **foreach** block in the go procedure. The **foreach** command requires two inputs: a list and a block of instructions to be run for each element of the list. In this case, the list is a list of integers from 0 to **runtime**-1, achieved with the code **range runtime**. The block of instructions needs to include all of the subsidiary procedures that are run for each tick in the basic forager program. These procedures could be included individually in the **foreach** block; instead, for greater transparency they have been incorporated into a single subsidiary procedure called **run-one-step**. An additional global variable, **start-ticks**, records the tick count at the beginning of each run. This is necessary because the calculation of mean energy per unit time requires the simulation time, which is **ticks-start-ticks**, given that we could not reset the tick counter at the beginning of the run.

We come now to the output-and-save procedure, executed at the end of each simulation. This procedure must do three things: record the result in **xlist** and **ylist**, report the results to the monitor window on the interface, and plot a point on the results graph. Recording the results utilizes the familiar **set** command along with the new built-in command **lput**. As usual, the **set** command requires two arguments set off by blank spaces. The first argument is the name to which the assignment is made, namely **xlist** or **ylist**, and the second is the value to be assigned to these names. The construction **lput X xlist** appends the new value of **X** to the end of **xlist**. Note that **lput** does not append a value to the variable name **xlist**; rather it creates a new list that consists of the old value of **xlist** plus the new value.[5] To reuse the name **xlist** for this new list, we have to use **set**.

Four commands create a line of text in the monitor window to give the results. The command **output-type** types a line of text without a carriage return, while **output-print** adds a carriage return after typing. We can enter several pieces of information using **output-type** and then enter the final piece using **output-print**. Each of the x and y results are typed in two pieces, a literal fragment in quotation marks that types a variable name and some punctuation, and the value of the specified variable. In the case of **mean energy**, the default precision produced by the calculation is misleading. It is not good scientific practice to report more decimal digits than is warranted for some purpose. Here precision is specifically chosen to be 3 decimal digits, which is more than is justified by the reproducibility of the results but needed to locate the point accurately on a graph.

Plots constructed through commands in the plot window of the interface all have time in ticks as the

---

[5] NetLogo commands never change the content of a list; they always create a new list from the old one.

x coordinate. Any other plot has to be constructed through a combination of interface elements and code elements. If you right-click on the experiment results plot window, you will see that there is one pen, called "results". There are boxes for plot setup commands, plot update commands, and pen update commands. All of these have been left blank, as the necessary commands appear in the code instead. Coding the plot starts with the built-in command **set-current-plot**, included in the setup procedure, which associates subsequent plot commands in the code with the plot identified as "Experiment Outcomes". A program can have multiple plots controlled by the code, so a **set-current-plot** command is required any time the program needs to plot points or curves on a different plot from before. In the output-and-save procedure for optimal_foraging.1, the built-in **plotxy** command adds one point to the plot.

8 optimal_foraging.2: Automating a Set of Simulation Runs

optimal_foraging.1 allows the user to obtain a visual display of how the net mean energy gain rate changes as the giving up resource level changes. But to accomplish this, the user must manually order each run. The next step in automation of experiments is to get a program that will do a sequence of simulation runs with the click of a single button. That is what has been added to make optimal_foraging.2.

For a first look at optimal_foraging.2, ignore the column of sliders on the left side of the interface window. Set the **Model Speed** at the maximum and then click **go**. The program automatically performs multiple runs with different values of the giving up level **X**; for each run, it displays the result and plots a point. Note that the **go** button no longer has the "forever" box checked. This is because the **setup** procedure has been incorporated into **to go**. The multiple simulation runs have to be directed within the code rather than with a setting on a button.

The **setup** procedure is run once at the beginning of **to go**. As part of the setup, the value of **X** is set to an initial value of 0.02. The simulation runs are executed by a block of code inside a loop. The **while** command requires two arguments, both in square brackets; the first is a Boolean condition, and the second is a block of code. The block of code repeats as long as the Boolean condition remains true. At the end of the repeating block there is a command that increases the value of **X**. Barring some other termination code, the value of **X** will eventually reach 1.0, which is too large to be a possible giving up level.

It is unlikely that the optimal giving up level will be close to 1.0, so a better stopping criterion is coded into the loop using an **if** command. Like the **while** command, **if** requires two arguments, a Boolean condition and a block of code. The difference is that **if** is a one-time command rather than a loop. The block of code is executed once if the Boolean condition is "true". The **stop** command in the block of code ends the execution of the go procedure when the latest *y* coordinate for the point on the graph is less than 1/3 of that for the highest point on the graph. We are looking for the value of **X** that gives the largest mean energy rate; the analysis will require enough points beyond this value to give good results, but any **X** values so large that the output variable is reduced by a factor of 3 will not be needed. The syntax difference between **while** and **if** is confusing. For **while**, the Boolean condition must be placed inside square brackets. For **if**, the condition is not bracketed (the parenthesis used here is optional). If there is a good reason for these commands to have two different structures, it escapes me.

Now back to the interface, we see that the basic model parameters are now specified by sliders. Eventually we will want to know how these parameters impact the optimal giving up resource level. First we will need code to calculate that optimum; for now, we can see how the parameter values affect the result by visually identifying an approximate peak in the **mean-energy** vs **X** graph and recording the results by hand.

9 optimal.foraging: Calculating the Optimal Giving Up Level

And now we move on to the finished optimal foraging program. Click **go** as before. This time, the optimal giving up level and corresponding mean energy rate are written in the monitor window and an orange curve appears in the plot window. Before turning to the code, inspect the plot window. You will see that a second pen has been added, colored orange and labeled "model". The instructions that produced the optimal **X** and orange curve are in the new subsidiary procedure **to analyze-and-report**, which appears in the code for the go procedure, just before the procedure stops. Note that this command appears twice because the procedure could stop either because the early stop criterion is achieved or because the value of **X** has reached its maximum.

There is no reason to expect that the graph of mean energy gain vs giving up level should be a parabola, but experience shows that the empirical fit to a parabola is quite good, so the vertex of the parabola serves as an excellent estimate of the optimal giving up level.

The principal quantities needed for the reporting are the optimal **X** value and the corresponding mean energy gain, called **optimal-X** and **optimal-Y**, along with two lists of points used to plot the best curve, called **model-X** and **model-Y**.

Each of the four desired output quantities is the target of a **set** command in **to analyze-and-report**. Several steps are needed to create a list of X values for the plot. The interval delineated by the minimum and maximum X values of the data is divided by 50 to create equally spaced nodes. The range command is then used to set the list of model X values. The details are a bit obscure. When **range** has three arguments, they represent the starting value, the stop value, and the spacing. The final value is actually the value just prior to the stop value, so we need a bit of extra code to to identify the correct stop value for the desired final value.

The other three output quantities are determined by a function **parabola**, which contains the code for the mathematical computations. **parabola** requires three arguments: the data from the simulation runs, and the list of X values for the plot. It returns a list that contains the desired information, but that list needs to be unpacked. The command that calls **parabola** assigns the results to the local variable **fit**. **parabola** packs the results in the order **optimal-X**, **optimal-Y**, **model-Y**; hence these are items 0, 1, and 2 in the result variable **fit**.

The remainder of **to analyze-and-report** consists of the text reporting and plot commands. The text reporting commands are similar to what we've seen before, but the plot commands are somewhat different. Prior to plotting the points, it is necessary to select the "model" plot pen. The 51 points are plotted using a **foreach** command. The code inside the square brackets plots each pair of items in **model-X** and **model-Y**.

It remains to understand how the **parabola** function works. Details of the mathematical algorithm appear in an appendix; here, we concentrate on the NetLogo code element that we have not seen before. The code is divided into 4 sections: preparation, sums, parabola coefficients, and desired results. The new code element is the built-in function **map**, which creates a new list by applying a formula to elements of another list. The first use we make of this command is to create lists of shifted variables from the original data. The structure of the command is "**let** newlist **(map** [formula] oldlist)". The structure of the formula inside the brackets is fairly transparent. First there is a name to be used for the individual entries in the original list, then an arrow, and then the formula that uses the original list entry to create the corresponding entry in the new list. This block of **map** commands includes a command that makes a list of x^2 values from the list of shifted X data. The reason for this list is clear in the next block, where the needed sums are calculated. We need the sums of x^3, x^4, xy, and x^2y. Each of these can be obtained as a dot product of two of the three shifted lists we just created rather than as a sum of elements in an additional list created in the same way as the x^2 list. The dot product is an example of an elementary mathematical construction that we might have expected to be built in. Not being intended for scientific computation, NetLogo has few such elements, but it is not hard to write one and include it in any programs for which it is needed. The code for the dot product is short, but includes a more complicated use of **map**, required because the new list is to be constructed from elements of multiple lists. This means that we can't use either of the precursor lists as the "oldlist" in the command structure. The way to manage this is to use a list of integers as the "oldlist". The first two commands in **to-report dot** create a list of integers that represent the item numbers in the precursor lists. The formula used in **map** picks out items from the precursor lists corresponding to a generic item number (the question mark, as I have written it) and then performs the calculation. Note what has been gained by writing a dot product reporter and then using it to calculate the sums in **to-report parabola**. Were the dot product reporter not available, we would have had to use the same **map** structure with "oldlist" of integers twice in the parabola reporter, once each for **SXY** and **SX2Y**. Not only is that unnecessary because of the dot product reporter, but it wasn't even necessary for the programmer to understand the code for the dot product reporter, which could have been copied from another program in the same way I expect the reader to write programs by copying elements of my programs.

10 Next Steps

With optimal_foraging, we can systematically vary one of the parameters, run the program, and record the optimal **X**. Then we can plot a graph of the optimal **X** vs the input parameter. However, given what we've learned about automating NetLogo scenarios, we should think about how to automate the data collection for this experiment. We have two options.

optimal_foraging.1 provides a template for how to proceed using NetLogo coding. In that program, the user chooses different values of **X**. The program calculates mean energy and plots the result on a graph of mean energy vs giving up level. In the same way, a first version of a new program could have the user choose different values of one of the parameters and then the program would calculate optimal **X** and plot that result against the parameter value. Some of the code in optimal_foraging is unnecessary. We could remove the code that prints the results for each **X**, that which plots the same, that which prepares data for the parabola plot and that which plots the parabola. Instead, we would have each run just print the optimal **X** and plot that quantity. Some of this work has been done in a new NetLogo program called optimal_foraging_function. The reader would have to add code to automate selection of parameter values and plotting of results in a graph of **optimal-X** vs the parameter value.

A simpler way to accomplish the goal of studying the effect of parameters on the optimal giving-up level would be to use BehaviorSpace as the NetLogo developers intended it to be used. The program optimal_foraging_function uses the optimal foraging model to create a function in which the independent variables are the parameters defined with sliders in optimal_foraging and the dependent variables are **optimal-X** and **optimal-Y**.

If you run optimal_foraging_function several times with the default parameter values, you will see that there is a lot of variation in the results. This is a consequence of the significant level of stochasticity in the underlying model. As a consequence, it would be a good idea to run the function 3 or 4 times for each set of input parameters, regardless of whether use you BehaviorSpace or your own code that automates both experiment design and data analysis.

11 Appendix: The Algorithm of the Parabola Function

Ignoring the actual variables used in the optimal foraging program, the task of the **Parabola** reporter is to use a set of data points x and y and a list of values x_p for a plot of the model to identify the parabola vertex (x^*, y^*) and the values y_p for the model plot, by fitting the data to a parabola. The straightforward way to do this is to construct an error function of the coefficients as

$$E(a,b,c) = \frac{1}{2}\sum(ax^2+bx+c-y)^2. \tag{4}$$

The optimal coefficients can then be found as the unique point where all the partial derivatives of E vanish. The three equations so obtained form a linear algebraic system, which can be solved numerically using a standard solver—except that NetLogo does not have such a solver. Writing a multi-use solver is outside the scope of what most mathematical modelers are able to do, since it requires extensive knowledge of numerical linear algebra. Instead, I employed a modified method designed for the specific problem at hand. The method is based on the observation that the problem is simplified if the data happens to be ordered so that the mean x and y values are 0. Thus, the first step is to replace the original xy data with XY data, using

$$X = x - \bar{x}, \qquad Y = y - \bar{y}. \tag{5}$$

Because the sums of the x values and the y values are both 0, the partial derivatives of the function

$$E(A,B,C) = \frac{1}{2}\sum(AX^2+BX+C-Y)^2 \tag{6}$$

with respect to C and B simplify, so as to yield the equations

$$\left(\sum X^2\right)A + nC = 0, \qquad \left(\sum X^3\right)A + \left(\sum X^2\right)B = \left(\sum XY\right), \tag{7}$$

$$\left(\sum X^4\right)A+\left(\sum X^3\right)B+\left(\sum X^2\right)C=\left(\sum X^2Y\right), \tag{8}$$

where all sums are understood to be over the full set of data points. These equations are partly decoupled, allowing the first two to be used to substitute into the third, yielding a scalar equation for A. From here, it is a simple matter to compute the vertex in XY coordinates, calculate Y_p from X_p, and then convert everything back to the original coordinate system.

References

[1] Ledder, G.: *Mathematical Model FAQ*, https://gledder1.github.io/MathBioEd/Mathematical_Models.pdf (2021)

[2] Ledder, G.: *Mathematical Modeling for Epidemiology and Ecology*, 2nd ed. Springer Nature, New York (2022)

Index