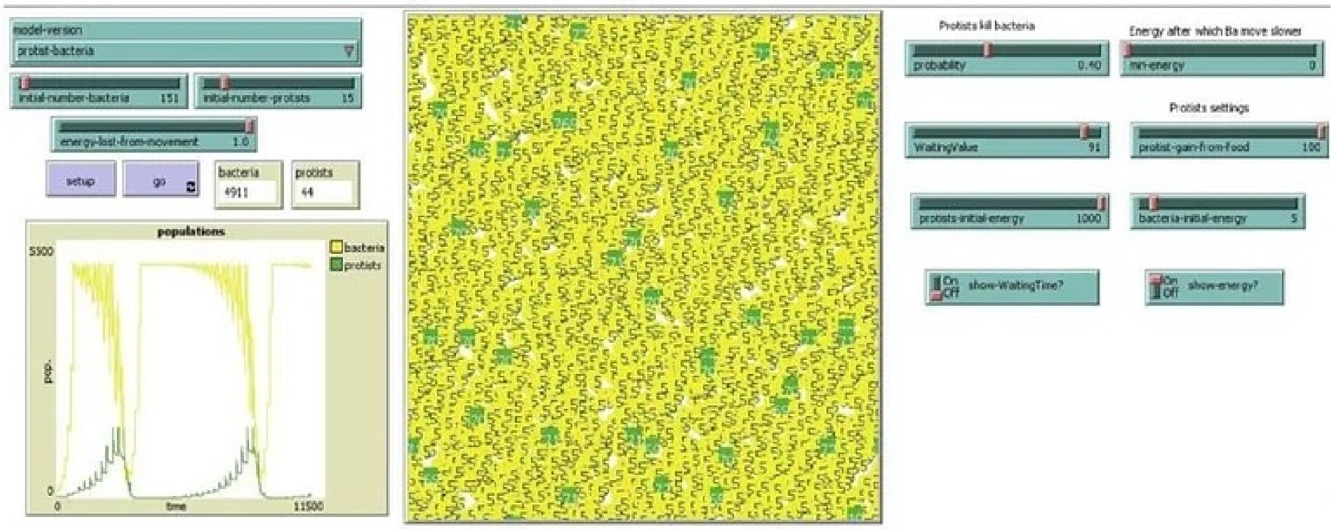
An Introduction to Agent-Based Modeling in Biological Systems Using NetLogo



model-version: protist-bacteria

initial-number-bacteria 151

initial-number-protists 15

energy-lost-from-movement 1.0

bacteria 4911

protists 44

Protists kill bacteria

probability 0.40

Energy after which Ba move slower

min-energy 0

WaitingValue 91

Protists settings

protist-gain-from-food 100

protists-initial-energy 1000

bacteria-initial-energy 5

show-WaitingTime? On [✓ Off]

show-energy? [✓ On] Off

Authors
Faichal Ayeva
Susan Bailey
Diana White

This manual was created for an Arts & Sciences Mini-Conference workshop held on October 23rd 2021, supported by the School of Arts & Sciences Advisory Council at Clarkson University especially through generous gifts from David A. Walsh '67.

1 What is Agent Based Modeling?

The underlying philosophy of agent based modeling (ABM) is that individual agents respond to a set of *individualized rules*, where rules can differ between different subgroups of agents in the population. Aside from individualized rules, agents can also interact (or not) with other agents, and so a set of *interaction rules* can be defined to describe what happens to each agent after interaction, or just prior to an interaction. A simple example of agents within a ABM are *cars* moving within a 2-D lattice, or grid, comprised of individual cells (locations where the car can move). Each agent has rules for how it should drive in the absence of other agents (either straight, turn left, or turn right), and in the presence of other agents (a car can not move into a cell with another car).

Examples of individualized rules for an agent in a population-type model can govern things like: movement in space, growth, production/ replication, and death/ removal from the population. Examples for individualized rules in a market model might govern things like: production, consumption, or the selling of goods. Interaction rules can also work to change an agent's state. For population models, this could involve: a gain in energy when an agent interacts with a food item (and so eats it), a change from alive to dead when a prey agent interacts with a predator agent (and so eats the prey), or simply a change in the direction of motion when a moving agent interacts with another agent or some other attribute of its environments that impeded its movement. It is important to point out here, that an agent may be stationary or motile.

For systems comprising of combinations of moving and stationary agents, an interesting question one can ask is what does the system of agents look like after running the system (running a simulation) for a long time? In ABMs, time is typically represented by each iteration of the model, that is, each time the individual and interactions rules are implemented. In ABMs which describe agents moving through space, *so-called* emergent phenomena can result from the interactions of those individual agents. An emergent phenomenon can have properties that are decoupled from the properties of the individual parts, creating patterns like clumps or aggregates of agents, strips or other interesting patterns.

The beauty of using ABM, as opposed to other types of computational or mathematical models to understand collective population-level behavior, lies in the fact that one does not require an extensive background in mathematics to develop and simulate such models. In particular, high-school students or undergraduates, who have basic skills in math and computer science can tackle modeling with ABM and come up with very cool simulations in no time at all! In addition, such modeling can include a very natural description of the system which is intuitive for the coder of the ABM to understand.

The overall goal of the workshop for which this manual was developed is for undergrads, graduate students, and faculty with little experience in coding to become acquainted with using ABMs, through hands-on experiential learning. In particular, we hope that you enjoy learning how to create and implement your own ABM, and how to use your ABM to gain insight into real biological phenomena (by comparing your results to real-world examples). Near the end of this manual, we will leave you with some unanswered questions, in the hopes that you can get creative and think of possible modeling solutions. Maybe your model will lead to interesting/new results!

In this workshop, you will learn the basics of ABM by writing your own ABM to simulate a real-world biological system. You will be using the open source software Netlogo, a popular software commonly used by both beginners and experts in the ABM community. Before you get started, you should go to the Netlogo website (https://ccl.northwestern.edu/netlogo/) and follow the download instructions to install and set it up on your computer. While there is also a web version of Netlogo available, we recommend using the desktop version, if possible, as not all functions are available in the web version.

Once you have finished installation, the following sections will provide a step-by-step guide to the basics of Netlogo programming, including opening and saving .nlogo files, setting up your Netlogo environment, and how to work with the graphical user interface (GUI). It is important note that this tutorial does not give as extensive an overview of the software Netlogo as some textbooks. For a more detailed description of the software, as well as other fun tutorials, see the textbook and other resources by Railsback and Grimm, or head to the Netlogo website.

Introduction to the computational aspects of ABM

ABM is a computational approach for simulating the actions of a group of *agents* (individuals), in response to both their local environment, as well as in response to their interactions with other agents (either of the same or different type). Typically, the overall goal of ABM is to explore *group* or *collective* behavior, which can emerge from simulation of many interacting agents. In general, there can be many agents defined in an ABM, where each agent has its own set of rules for how it “acts” alone, and when it encounters other agents (of a similar or different type). In addition, agents of different types can have different sets of rules for how they behave independently, and when they encounter other agents.

In Netlogo, four different categories of agents are defined: Mobile agents, Patches, Links, and the Observer.

1. *Mobile agents* (also called *turtles* in Netlogo) are the type of agents we most typically think about when using ABMs. They are individuals in the system that move around and can interact with one another, and with their environment.

2. *Patches* represent locations in space. In Netlogo, patches are square, and can be set to any size. Further, you can have as many patches in a 2D lattice, or a 3D lattice as you’d like (where the entire lattice is referred to as the “World” in Netlogo). The mobile agents will move between the patches, and can interact with other mobile agents on (or close to) the patch where they are located. You can also set rules for how a mobile agent should behave when it is on a given patch.

3. *Links* define the interactions between pairs of mobile agents.

4. *The Observer* controls everything (i.e., you). It controls the dynamics of the system over time, how you display those dynamics, and how you might record certain information.

Next we define the rules of the ABM, including what the environment looks like, how agents move, and how agents interact with each other and their local environment. In Netlogo, we do this using built-in commands called *primitives*. Common primitives you will use in this tutorial are:

`setup` → used to initialize the environment and starting conditions

`fd` → moves a defined agent (also called "turtle" in Netlogo) forward by a specified value

`hatch` → a *reproduction* command, creates a defined number of new agents

`ask` → executes the procedures that follow it on a defined "agentset" (i.e. group of agents)

`die` → a *death* command, removes an agent from the system

As with learning any new programming language, you should become familiar with the **User Manual**. Probably, the most important part of this manual is **The Netlogo Dictionary**, which houses information for all the **variables** (can be numbers, text, agents, or agentsets) you might use, as well as all the commands (called *primitives* in Netlogo, which tell agents what to do). Global variables belong to the observer, and are variables such as model parameters, and environmental characteristics (like patches) that affect all agents. Take a moment now to briefly check out the user manual and practice using the dictionary by searching for a few of the primitives listed above.

2 Modeling a growing population of bacteria

Next, we'll walk you through the steps for designing your very own ABM. We'll start by setting up the Netlogo environment, which is the domain (or space) in which your agents will move around. The end result will be the creation of an ABM that explores the behavior of a group of bacteria (the agents), as they move through a two dimensional environment (similar to how they'd move across a Petri dish in an experimental setting). Rules for these agents will include things like reproduction, death, energy gain from eating food within their environment, and rules for movement.

A little bit about growing bacteria in the lab

In the lab, bacteria are cultured in a number of different types of experimental environments, three of which are: 1) well-mixed liquid, 2) semi-solid agar, and 3) solid agar. Agar is a gel-like substance that makes a bacteria culture environment range from semi-solid to solid, depending on the concentration used. In all three environments, an experiment is initiated with some amount of food (which we will define below), and an initial number of bacteria (typically a small value). After the experiment is initiated, the bacteria eat food (depleting it over time), reproduce, and die. The important difference between these three lab environments is in how the bacteria move around. In a well-mixed liquid environment, the whole culture is shaken continuously (usually on a mechanical rotating platform) and so one can think about this as random mixing, essentially picking bacteria up and placing them back down at random across the 2-D environment at every time step. In a semi-solid agar, can bacteria often move by swimming slowly (using structures called flagella and cilia) - one could think about this as a random or so-called "diffusive" movement. In a solid agar environment, bacterial cells typically cannot move at all. Each bacterial cell is fixed at its initial position, but the population as a whole can expand outwards through reproduction. That is, as bacterial cells reproduce through cells simply dividing in two, "daughter" cells take up space right next to their "mother" cells. In this tutorial, we will take you step-by-step through building a model of the population dynamics of bacteria living in a semi-solid agar environment.

After you have completed this introductory tutorial, you are encouraged to try modifying your model to describe the dynamics of bacteria living and moving within the well-mixed liquid and the solid agar environments, as well as developing and exploring new questions of your own.

Starting Netlogo

Netlogo is a relatively user-friendly coding platform, with a graphical user interface (GUI) where you can "click" on options rather than hard coding everything yourself. In that sense, it's a nice introduction to coding!

Let's begin by opening a new Netlogo GUI/ interface and file, and saving it. Under the **file** icon, select **save** and write "Semi-solidAgar.nlogo". Now, click the **code** tab in the top left corner, and write a short introduction for what you are planning to code. Maybe write something like this:

```
;Model of the dynamics of bacteria in a 2-D environment. --- Jenny Coder Oct 2021
```

Note that there is a semi-colon at the beginning of the description. Semi-colons are used in Netlogo to write comments that won't be read during simulation (some people refer to this as *commenting* your code). In general, it's always good practice to add comments to your code as you go, so that when you come back to it you will remember what you've done. It's also good practice to check and save your code after each new addition. To check your code, you can click the **check** icon on top of the code window (see Figure 1).

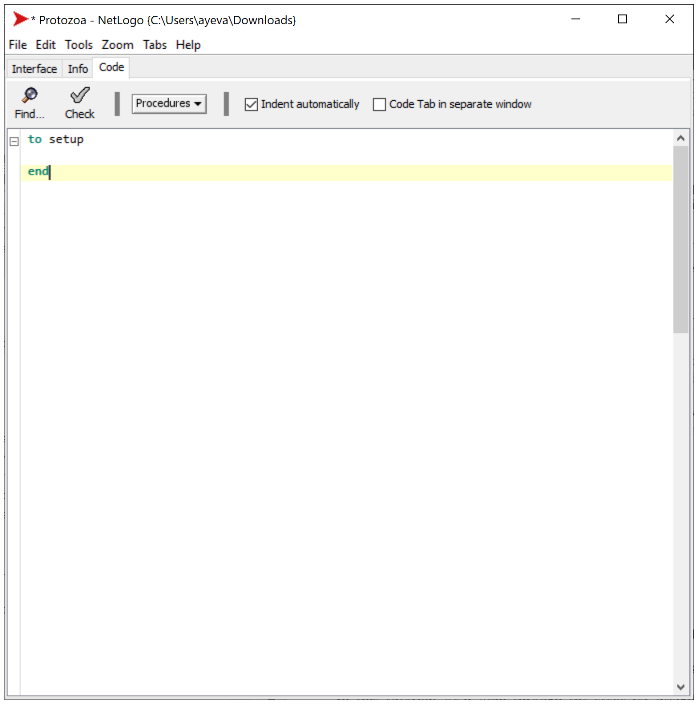
Next, let's setup the Netlogo environment. Go back to the **interface** tab and begin by clicking the **button** icon, which gives a drop down list of options. Once you select "button" from the drop down list, the "+ Add" icon will be highlighted. Now add a button by clicking anywhere in the white part of the GUI. You will see a box appear, similar to that shown in Figure 2. Within the small window that pops up, type "setup" in the "Commands" field and click OK. You should see a box (called a button) appear in the white part of the GUI where clicked. Notice that your new button called "setup" is red, and if we click the button nothing happens. That's because we haven't added any code to the setup command yet. Let's do that now.

Click on the **code** tab, and type the following in the blank white space (this space is where all your code will be housed).

```
to setup

end
```

Figure 1 provides an illustration of what your code interface should look like at this point.

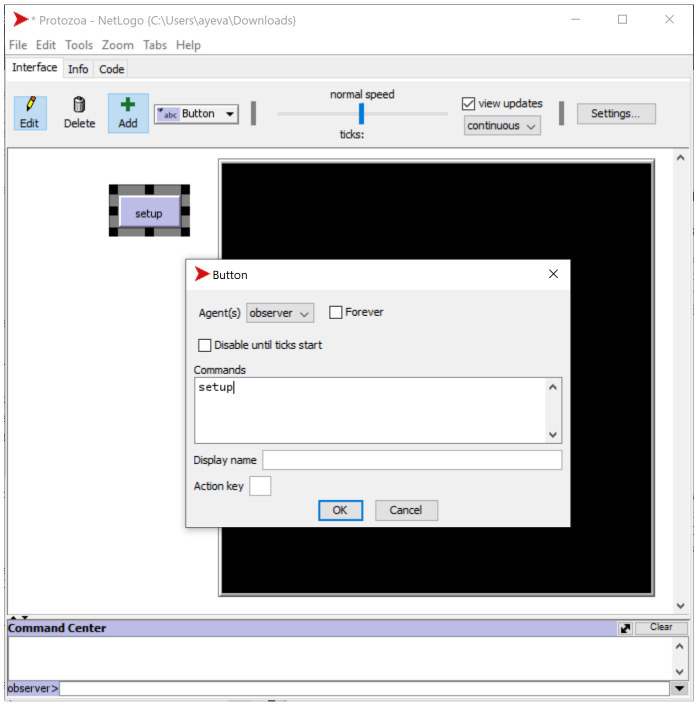

* Protozoa - NetLogo {C:\Users\ayeva\Downloads}
File Edit Tools Zoom Tabs Help
Interface Info Code
Find...
Check
Procedures
Code Tab in separate window

Indent automatically: ✓

```
to setup

end
```

Figure 1: A screen capture of the code interface, where all your code will be housed.



```
setup
```

Figure 2: Model setup

Note the word `to` at the beginning of the setup procedure. `to` is used at the beginning of each procedure to start that command. Click the **check** icon at the top left to *check* that your code has no errors (again, do this often). Clicking **check** does not run the code, it simply checks it for errors. Now that you have created the setup procedure and checked that there are no errors, you should see under the **interface** tab that the **setup** button is no longer red, which means that the setup procedure is well defined, although it still has no instructions. Let's provide those instructions now.

Setting up your Netlogo environment - the setup command

Here we will add code to the **setup** procedure, to initialize the environment. We will define the environment size, the number and placement of bacteria in the environment, and where the food is located. Note that we've included a `clear-all` command. This allows us to begin each setup from scratch (i.e., we erase everything that is left over from the last run of the model so that everything is newly initialized). It's a good habit to always begin the setup procedure with `clear-all` or `ca` (short for clear-all), since there could be old parameter values left from a previous simulation which might make things act wonky.

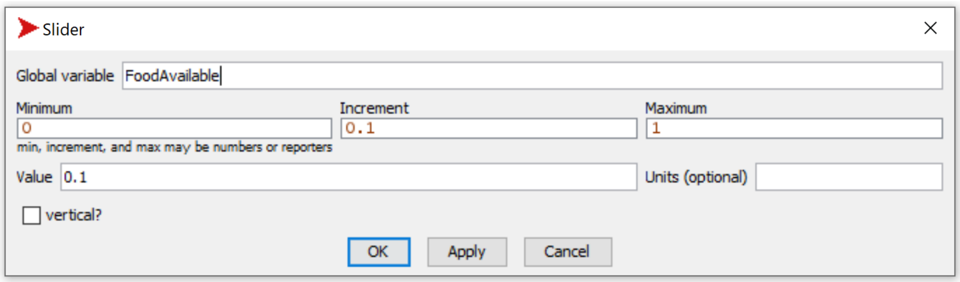
In the setup procedure, add an **ask** primitive to initialize a procedure for only the patches. Here, all primitive statements, as well as "if" and "ifelse" statements are placed just outside a pair of square brackets "[ ]". The placement/indentation of the brackets is important. In particular, the open bracket "[" and the close bracket "]" should have a line of their own. The indentation is not required, but it can help to use indentation to help make the code easier to read. For those of you new to coding, an "if" statement means that a particular command (here, the statement `set pcolor green`), will only be carried out if a specified condition is met (here, the condition `if random-float 1 < FoodAvailable`). If the condition is not met, the command will not be executed. Before moving on to a detailed discussion of this condition and command, try writing out the code as described below and guess what the command will do.

```
to setup
clear-all
ask patches
[
    if random-float 1 < FoodAvailable
    [
        set pcolor green
    ]
]
reset-ticks
end
```

Let's discuss what is going on in the condition `if random-float 1 < FoodAvailable`. First, the primitive `random-float` draws a random floating number ("floating" just means that the number is rational, that is one integer divided by a another integer) that is greater than or equal to zero but less than the number that follows it (in this case, 1). `FoodAvailable` has not been defined or assigned a value yet, but we will define it as the proportion of the environment that is filled with food for the bacteria. Since our condition is `if random-float 1 < FoodAvailable` , if we were to set `FoodAvailable` to 0.3 (30%) then food would randomly be placed on 30% of the patches in the environment (since the probability of a random number draw between 0 and 1 being less than or 0.3 is just less than 0.3 (30% of the environment)).

Now, since we have an "if" statement, our command `set pcolor green` would be executed for 30 % of the patches in the environment, if we set FoodAvailable to 0.3. In other words, 30 % of the patches would be colored green (and we will define green patches as those that have food for bacteria). Now, try checking your code. What do you notice? An error! Think about why that might be happening...

... It's because we haven't defined our variable `FoodAvailable`. There are a few ways we can do this, for example we could declare FoodAvailable and assign it a value (to do this, we can type `set FoodAvailable 0.3`) just under the `clear-all` command. But, let's get more fancy (and also more flexible!) and use a **slider**. Sliders allow you to easily assign different values to a variable at the beginning of each new simulation. To make a slider, go to the main GUI interface, right click anywhere in the white space, and select **slider**. Just like in Figure 3, assign `FoodAvailable` to Global variable and assign a minimum value, an increment, and a maximum value for the variable (in this case we've selected, 0, 0.1, and 1 respectively). This means you'll be able to change `FoodAvailable` to any of the values 0, 0.1, 0.2, ... , 1.



Global variable FoodAvailable
Minimum 0
Increment 0.1
Maximum 1
min, increment, and max may be numbers or reporters
Value 0.1
Units (optional)
vertical?

Figure 3: Window for creating a slider.

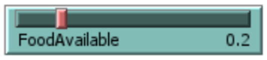


Figure 4: An example slider that should show up on the main GUI interface. In this example, the variable maximum is set to 1, and the slider is set to 0.2, meaning only 20 % of the environment would be filled with food.

Notice the new slider that looks like Figure 4 in the GUI interface. Now if you try checking your code (try it!), there should not be any errors. If you see there are no errors, execute your setup by clicking on the **setup** button. You should see a portion of the environment turn green (this proportion will depend on the value you choose on the slider you just created). As stated above, we would like every green patch to represent food that the bacteria can eat, such that when a bacteria moves to that patch, it eats the food, and returns the patch to its original color, black. In addition, when the bacteria eats food, it should gain energy (like you and I do after we eat). As a final note, you should see the line of code `reset-ticks` at the end of the setup procedure. This will be explained in full detail when defining our next "go" procedure. For now, just think of it as resetting the "time" of the simulation back to zero.

To set up the mobile agents in our model (the bacteria), we first need to give them names, defining both a plural and singular version of their name. We also want each agent in our model to have "energy" (more about this soon) and so we also need to define this attribute. To do this, in the code interface, add the following lines of code just before the setup command:

```
breed[bacteria bacterium]
bacteria-own[energy]
```

Next, inside of the `to setup` procedure, add the following lines of code (just before the reset-ticks and end commands) to initialize the bacteria:

```
create-bacteria initial-number-bacteria
```

```
[
    set shape "square"
    set color white
    set size 2
    setxy random-xcor random-ycor
]
```

The primitive `create` initiates bacteria as an agent, and the variable placed after it defines the number of agents (bacteria) to initially place in your environment. Here that number is defined by `initial-number-bacteria`. The primitives `shape`, `color` and `size` set the characteristics of what the bacteria will look like. Be creative, or use the numbers/colors chosen here. Just as we created a slider for `FoodAvailable`, go ahead and create a slider for `initial-number-bacteria` (see example in Figure 5).

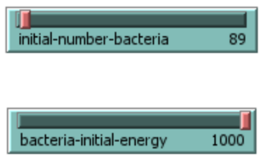


Figure 5: Sliders for the initial bacteria population size and the amount of energy bacteria gain from eating food in a single patch.

Now we want to tell each agent where to start (to initialize their positions in 2D space). The command `setxy random-xcor random-ycor` randomly places each agent (bacteria) at any patch in the environment with equal probability, by randomly choosing an x and y coordinate in the 2D space. You should see a roughly uniform distribution of bacteria through your domain (environment) once you execute the setup command again.

Finally, in the setup, we need to define a variable `energy`, to describe the initial amount of energy given to each bacteria. As before, create a slider for this energy called `bacteria-initial-energy`. You should see an example of both sliders you've created in Figure 5 (also note the example ranges we've provided. You can use your own ranges or the ones we've provided). See the new line of code below to add the energy command:

```
create-bacteria initial-number-bacteria
[
    set shape "square"
    set color white
    set size 2
    set energy bacteria-initial-energy
    setxy random-xcor random-ycor
]
```

The full code up to this point should look like the following:

```
breed[bacteria bacterium]
bacteria-own[energy]

to setup
clear-all
ask patches
[
    if random-float 1 < FoodAvailable
    [
        set pcolor green
    ]
]
reset-ticks
end

create-bacteria initial-number-bacteria
[
    set shape "square"
    set color white
    set size 2
    set energy bacteria-initial-energy
    setxy random-xcor random-ycor
]
```

Bringing the model to life - the go command

When we run our model, we'd like to see the bacteria reproduce periodically, move around randomly in the 2D environment, and eat food whenever they find it (when they land on a patch that has food). As in reality, bacteria will gain a certain amount of energy each time they consume food. In addition, they lose a certain amount of energy each time they move and when they reproduce. And, if their energy level drops to zero, they die and so are removed from the system (essentially they starve to death). In this section, we will work to create all these procedures. In particular, we will define procedures for bacterial movement, reproduction, and death.

First create a button (just as you did with the setup button), and call it "go". This button will be what you click to simulate your model. From Figure 2, you'll see that there is a checkbox option called **forever**. For "go" button, check the box **forever** and select the tab **OK**. This means that the simulation will continue to run for as long you want. Clicking "go" the first time will start your simulation, and clicking it a second time will stop it (you can click the button a third time to continue it again and so on). You'll notice that there are a pair of circular arrows on the go button. This just indicates that you've chosen the **forever** option for this button.

Just below your setup code, you will now create the "skeleton" of the **go** procedure with the words `to go` and `end`, following the same procedure used for **setup**.

```
to go

end
```

Now, declare the main procedures of the bacteria in the to go procedure (movement, death, and reproduction):

```
to go
   ask bacteria
   [
      move-bacteria
      death-bacteria
      reproduce-bacteria
   ]
   tick
end
```

Once you hit the to **go** button on the main GUI interface, all of these procedures will be launched. As of now, we haven't added instructions to the procedures, so we'll do that now. Let's bring this model to life! Note that we've added a line of code not yet introduced, that is, the word `tick`. This allows you to keep track of *time* in the simulation. Right now `tick` is arbitrary in the sense that it just means one iteration of our simulation. Later we will provide units for it.

The movement command

The "skeleton" of the move command will have the words `to move-bacteria` and `end` as follows:

```
to move-bacteria

end
```

Notice that this command is defined for all bacteria. Next, we will use an "ifelse" command, which is similar to the "if" command. However, with "ifelse", when the condition given is not met, the first command is not executed, and the code moves onto a second command, executing that second command instead. Here, we will define the condition to be met as whether the patch is colored green (has food) or black (does not have food). If it is green, the bacteria will eat the food and gain energy, the patch will turn black, and the bacteria will move to a new patch, losing energy due to that movement. If the patch is black, the bacteria will just move to a new patch, again losing energy due to that movement. First type the following lines of code:

```
to move-bacteria
    ifelse pcolor = green
    [
        set pcolor black
        set energy energy + bacteria-gain-from-food
    ]
    [
    ]
end
```

The `ifelse` primitive executes the code in first set of brackets if the condition `pcolor = green` is true. If false, the code in the second set of brackets is executed (which is blank, so nothing happens). Furthermore, if the patch is green, the command `set pcolor black` is executed and the patch turns black. Now, once a bacterium has eaten the food in the patch, the energy of that bacterium will increase, updating to the current energy plus the energy gained from eating, `bacteria-gain-from-food`. Before checking your code, you must define `bacteria-gain-from-food`. Try creating a slider which sets the range of energy gain from 0 and 825.

Next, let's include the movement rules. Finish the above movement procedure by writing out the rest of the code as follows:

```
to move-bacteria
    ifelse pcolor = green
    [
        set pcolor black
        set energy energy + bacteria-gain-from-food
        rt random 90
        lt random 90
        fd 1
        set energy energy - energy-lost-from-movement
    ]
    [
        rt random 90
        lt random 90
        fd 1
        set energy energy - energy-lost-from-movement
    ]
end
```

The commands `rt random 90` and `lt random 90` allow the bacteria to randomly move between a wedge of 90 degrees to the left or right direction (from it's current direction), and `fd 1` moves the bacteria forward 1 step (the step size is the vertical/ horizontal length of a patch) at each iteration of the simulation (which in Netlogo is called a **tick**). The command `set energy energy - energy-lost-from-movement` causes the bacteria to loose the amount of energy you set as `energy-lost-from-movement`. Again, you will need to set this energy, and you can do so using a slider. Note that this energy lost from movement should be less than the initial energy given to the bacteria, as well as the energy gain from eating, or else the bacteria will lose energy quickly and die (and the simulation won't be very interesting). The code in the second set of brackets is similar to the first, but it only runs when there is no food (the patch is black), which means the bacteria doesn't gain any energy from eating.

You will notice that right now, your code doesn't pass the "check" because we haven't defined death-bacteria and reproduce-bacteria yet, and so this means the "set up" and "go" buttons aren't working. If you'd like to try running your partially completed code at this stage, you can do so by temporarily putting a semicolon, ;, in front of those two lines of code in the to go part of the code on the top of page 12 (this will temporarily turn these lines into "comments", but don't forget to "uncomment" these lines later when you need them to work!). Now try running your code and see what happens! Press the setup button and then the go button. Ask yourself whether the results make sense.

The reproduction command

Now that we've completed the movement procedure, it's time to define reproduction. Like before make the "skeleton" of the reproduction procedure. Type the following:

```
to reproduce-bacteria

end
```

Now, this is where we need to start thinking about units for our system. Bacteria reproduce at a wide range of rates, depending on the particular species ans conditions of their environment, such as the temperature and type of food resources available. For example, under ideal lab conditions, the common bacteria *Escherichia coli* reproduces about once every 20 minutes, but likely closer to about every 900 minutes (i.e. 15 hours) when growing out in the "wild". Let's start by setting reproduction rate to an intermediate value, choosing a rate of 1/240 minutes (i.e. 1/4 hours).

Now, within the reproduction procedure, add the following lines of code:

```
to reproduce-bacteria
    if random-float 1 < 1 / 240 and energy > 0
    [
        set energy (energy / 2)
        hatch 1 [ move-bacteria ]
    ]
end
```

You'll notice that we've used a "if" command again. By now you should be comfortable using this, and understand the construction of the command. You'll also notice the `random-float` floating variable. Scroll up to the setup procedure if you forget how this works. Here, we are setting the condition that if a random number, drawn between 0 and 1, is less than the reproduction rate, a new bacteria is created. In the code above, reproduction rate is set at 1/240, but if you would like, you can try replacing `1 / 240` with `rate-of-reproduction` and setting this value with a slider as you have done previously. This slider will allow you to easily adjust reproduction rate to look at its impacts on bacteria population dynamics.

The `hatch` command is used in Netlogo for reproduction of agents, and we've specified here to only create 1 new agent. In addition to creating a new agent (which is often called the "daughter cell"), we split the energy evenly between mother and daughter. We also allow that new daughter cell to move away from its mother's location by calling on the move command `move-bacteria`. Think about why we've added `energy > 0` in the "if" condition, also. Once again, check your code and try running it. You will need to keep the `death-bacteria` line in the `to go` part of the code "commented" because we still haven't defined this procedure, but make sure that the `reproduce-bacteria` line is *not* "commented" (i.e. does not have a semi-colon at the start of the line).

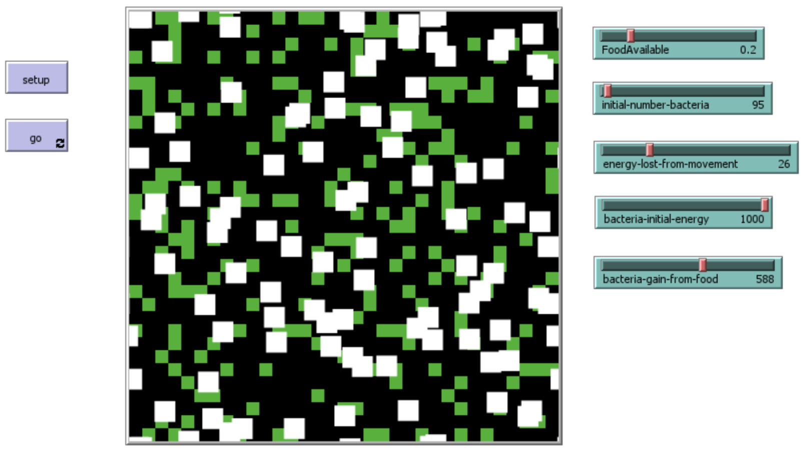
The death command

Your model is almost complete. All we have left to do is define a procedure for how bacteria die (i.e., leave the Netlogo environment). As with all other procedures, start with your "skeleton" (as you did before), and once you are done (check your code!) add the following instructions to it:

```
to death-bacteria
    if energy <= 0
    [
        die
    ]
end
```

Now that you understand "if" statements, you should follow what these lines of code do. The death procedure works such that if the condition `energy <= 0` is met, the command `die` is executed, and that bacterial cell is removed from the Netlogo environment permanently.

Congratulations on setting up and running your first ABM! To see you what your completed environment should look like in the main GUI after clicking the **setup** button, see figure 6.

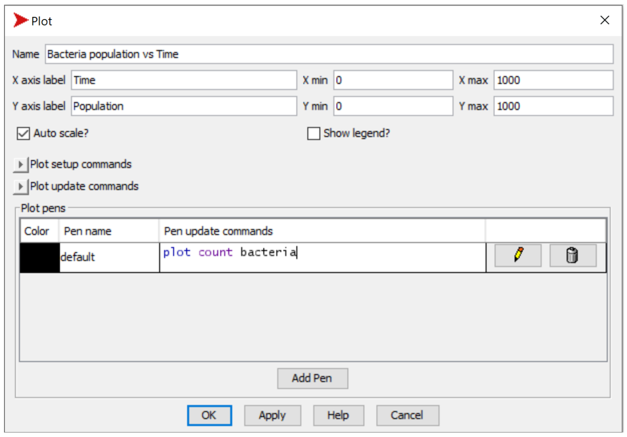


FoodAvailable 0.2
initial-number-bacteria 95
energy-lost-from-movement 26
bacteria-initial-energy 1000
bacteria-gain-from-food 588

Figure 6: Semi-Solid Agar model Netlogo World. Notice all your sliders (which you can move around), your **setup** and **go** buttons, and the environment where bacteria are colored white and the food is colored green. All empty patches are black.

Let's try running the completed code. Click the go button once, wait a little, and then click the go button again to stop it. While running your code, you should observe bacteria moving around, food disappearing, new bacteria appearing (via reproduction), and bacteria eventually disappearing (leaving the environment via death). In general, in an actual lab experiment with bacteria, you will be able to observe/record data in 2 ways: (1) spatial density over time (similar to the 2D grid we've displayed here), and (2) total population size over time. Total population size data is much easier to collect in the lab and easier to compare, so to aid in that comparison, let's also collect total population size data from the ABM. To do this, we will create a graph of total bacterial population size vs time. The code for this graph will count all the bacteria in the environment at each tick and plot that number. Time will be on the "x-axis" and total population size will be on the "y-axis".

Right click anywhere in the main GUI tab, select "plot" and name your plot - for example call it **Bacteria population vs Time** (see Figure 7). Set a range of 0 to 1220 for bacteria along the y-axis (you can always change this, and in fact you should if you have larger population sizes) and 0 to 1000 along the x-axis (again, you can always change this). Recall the time here is measured in minutes (since we have defined reproduction as 1/240 minutes at each "tick"). For both the x and y-axis and enter "plot count bacteria" for the pen update commands. You may also choose to show a legend. This is not that useful now, as you are only plotting one one population, but in a future model this may be important if you have multiple populations coexisting, and legends with plot colors will help you to distinguish between those populations.


Plot

Name: Bacteria population vs Time

X axis label: Time — X min: 0 — X max: 1000

Y axis label: Population — Y min: 0 — Y max: 1000

[✓] Auto scale? — [ ] Show legend?

Plot setup commands

Plot update commands

Plot pens

Color	Pen name	Pen update commands
■	default	`plot count bacteria`

Figure 7: Bacteria population vs Time graph settings

Now try running your model. You should see a window pop open, showing the graph of population size vs time at each tick/instant in time. Figure 8 shows a screen shot of the main GUI interface after clicking go a second time to stop the simulation. Notice the new graph **Bacteria population vs Time** which appears to have stopped been at t = 400 minutes (the 400th tick). The curve is headed downwards, and will continue to do so since there is no food left (green patches) for the bacteria to eat.

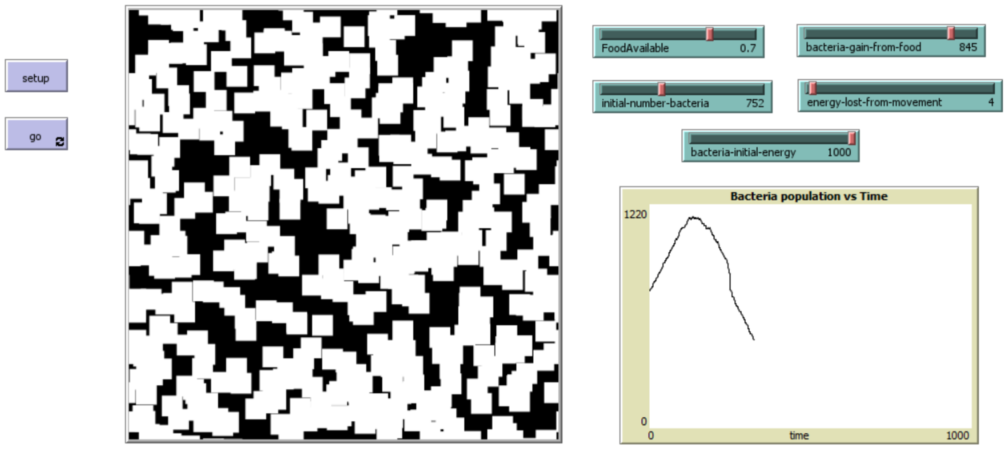

setup
go
FoodAvailable 0.7
bacteria-gain-from-food 845
initial-number-bacteria 752
energy-lost-from-movement 4
bacteria-initial-energy 1000
Bacteria population vs Time
1220
0
0
time
1000

Figure 8: Notice that all the patches in the environment are either white or black, as all the food has already been consumed by the bacteria. The graph of **Bacteria population vs Time** is moving downwards, and if you let the simulation continue, it will go to zero as there is no more food for the bacteria to eat and they loose energy each time they move.

3 Getting Creative: Questions and model extensions

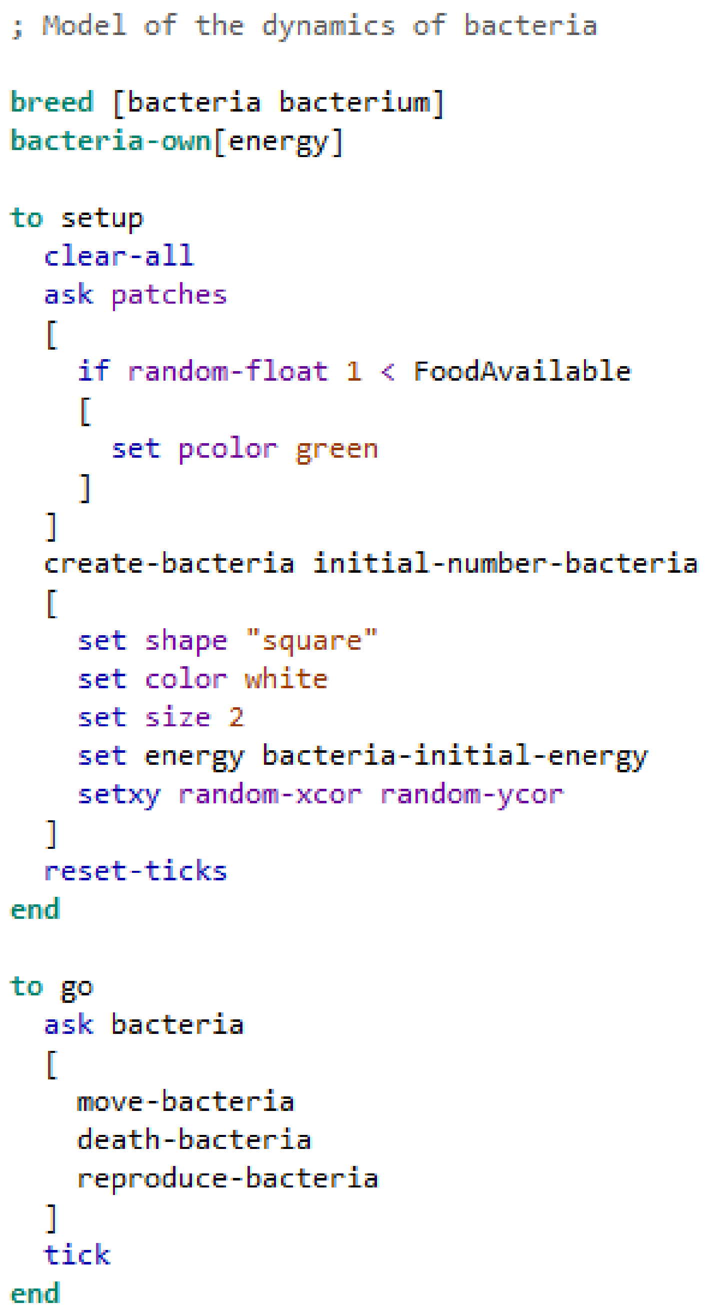
There are many questions that you can ask using the model you have created. For example, how does adjusting the initial number of bacteria change the population dynamics of the system? How does changing the proportion of the 2D environment that is filled with food change the dynamics of the system? Can you think of other interesting questions you could ask by adjusting the sliders? Choose a few questions and explore the range of outcomes by adjusting your sliders and running your model a few times.

Extending your model

You have created a simple bacterial growth model, however there are a multitude of ways that this model could be extended or adjusted to model a slightly different scenario. For example, think back to the description of different types of bacteria growth environments at the beginning of this tutorial. In your first simple model, we have aimed to capture bacterial movement that might occur in a semi-solid agar environment. How might you describe bacterial movement in a solid agar environment or a liquid environment? See if you can come up with a way to adjust/ extend your model to appropriately describe one of these alternate scenarios.

4 Appendix: Final code

Here is what the code for your completed bacterial population growth model should look like:



```
; Model of the dynamics of bacteria

breed [bacteria bacterium]
bacteria-own[energy]

to setup
  clear-all
  ask patches
  [
    if random-float 1 < FoodAvailable
    [
      set pcolor green
    ]
  ]
  create-bacteria initial-number-bacteria
  [
    set shape "square"
    set color white
    set size 2
    set energy bacteria-initial-energy
    setxy random-xcor random-ycor
  ]
  reset-ticks
end

to go
  ask bacteria
  [
    move-bacteria
    death-bacteria
    reproduce-bacteria
  ]
  tick
end
```

```
to move-bacteria
  ifelse pcolor = green
  [
    set pcolor black
    set energy energy + bacteria-gain-from-food
    rt random 90
    lt random 90
    fd 1
    set energy energy - energy-lost-from-movement
  ]

  [
    rt random 90
    lt random 90
    fd 1
    set energy energy - energy-lost-from-movement
  ]
end

to reproduce-bacteria
  if random-float 1 < 1 / 240 and energy > 0
  [
    set energy (energy / 2)
    hatch 1 [move-bacteria ]
  ]

end


to death-bacteria
  if energy <= 0
  [
    die
  ]
end
```