# NetLogo Tutorial Series: Rock-Paper-Scissors Ecosystem

Nicholas Bennett
Grass Roots Consulting
nickbenn@g-r-c.com

November 2010

## *Copyright*

## *Acknowledgments*

Development of this curricular material and the accompanying NetLogo models was funded in part by:

- Santa Fe Institute Summer Internship/Mentorship (SIM) program for high school students;
- New Mexico Supercomputing Challenge;
- Project GUTS.

Participants in the above programs have also provided invaluable feedback on earlier versions of this material.

## *Introduction*

In this activity, we'll build a NetLogo model in which the turtle agents, grouped into different breeds or strains in an ecosystem, play a simplified form of Rock-Paper-Scissors.

A natural first question might be: What is there about Rock-Paper-Scissors that makes it a good subject for a NetLogo model? For one thing, even a basic implementation includes a number of important intermediate NetLogo concepts and techniques. But arguably more interesting is the fact that there are some biological systems which behave, in certain respects, like Rock-Paper-Scissors.

One example is a particular ecosystem, made up of *Escherichia coli*, or *E. coli* bacteria [1]. In this ecosystem, there are three types of *E. coli*:

- Type C has the ability to create a toxin called colicin, but is itself resistant to the toxin.
- Type R is resistant to colicin, but it doesn't have the ability to make it.
- Type S gets killed when exposed to colicin.

Partly due to the differences with respect to the production of and resistance to colicin, the three types have advantages and disadvantages, with respect to each other:

- Type R bacteria grow more rapidly than type C bacteria, because not having a portion of its metabolism dedicated to producing colicin allows type R to reproduce more efficiently. If those two types of bacteria are put in a same container, then type R dominates and eventually displaces type C.
- Type S bacteria grow even more rapidly than type R bacteria, because type S absorbs nutrients more efficiently (partly because none of the type S metabolism is dedicated to resisting the colicin toxin). Hence S dominates and displaces R.
- The colicin produced by type C kills type S, so C dominates and displaces S.

Hence, these three types of bacteria have the cyclical dominance feature of Rock-Paper-Scissors: R beats C, C beats S, but S beats R.

## *Model Behavior*

The agent behaviors we'll define in our model, while simplified greatly from the real world, end up producing an aggregate behavior that parallels the real world behavior of the *E. coli* ecosystem quite well. There are other biological systems with cyclical dominance that aren't quite as well suited to this approach[1], but a model like this can still help us learn something interesting about some of those ecosystems as well.

The basic characteristics of the model are as follows:

- The terrain is completely populated by turtle agents – i.e. there is one turtle agent per patch.
- Turtles do not move (though this can change).
- Each turtle competes only against the turtles directly (not diagonally) adjacent to it.
- Each turtle starts out with a strategy of rock, paper, or scissors, selected at random. It maintains this strategy until it is defeated, at which point it adopts the strategy of the victor. Whenever it changes to a new strategy, it thens continue with the new strategy until it's defeated again, and once again adopts the victor's strategy. In this admittedly simplistic manner, we model the death of one organism, and its replacement by the offspring (mitotic, in the case of *E. coli*) of another.

---

1 Another parallel to Rock-Paper-Scissors in biological systems is found in the Common Side-blotched Lizard (*Uta stansburiana*). This species has three different types of males that can be distinguished by the color of their throats. Each color variation corresponds to a specific evolutionary strategy for mating:

- Orange throat males have the largest territories and posses a large number of females, and are physically the strongest among the three types.
- Yellow throat males are the weakest of all the types. However, they look similar to females, so they try to sneak into territories of other males and breed with the females.
- Blue throat males don't guard as many females as the orange throat males do, but they guard them more carefully.

Orange throat males dominate blue throat males, due to their strength advantage. Blue throat males dominate yellow throat males because of strength, and because they guard females closely, which makes it difficult for yellow throat males to sneak in. Yellow throat males are able to sneak into orange throat males' territories and steal females, and can thus dominate them, in terms of breeding opportunities. In fact, this system is not a simple RPS game because there are also two types of females, and because females control the system as well by choosing their mate. Nevertheless, there is cyclical dominance, and a strong resemblance to the behavior exhibited by our model.

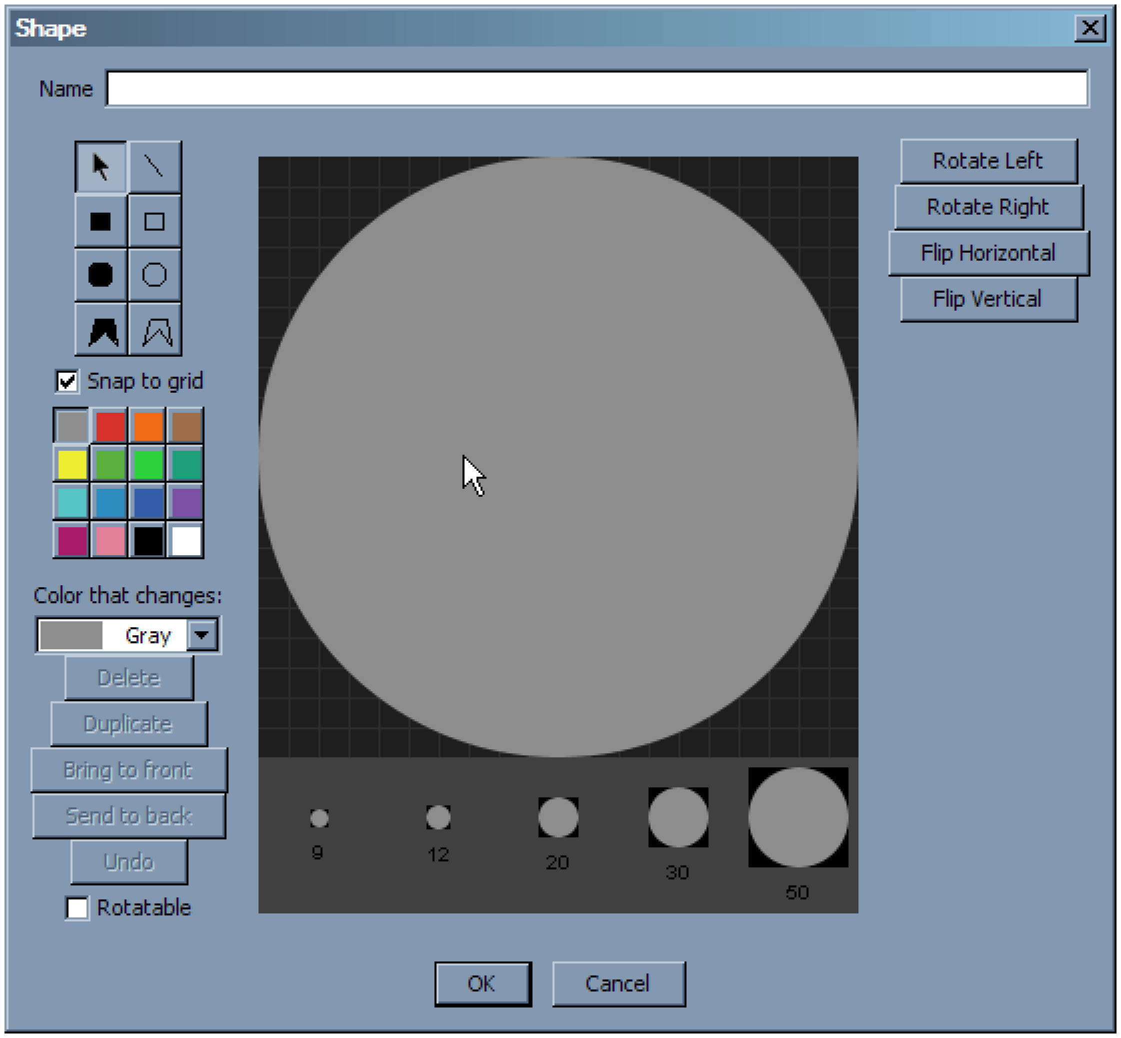
## *Task 1: Setting Up the Model*

Rather than make this model specifically about the three *E. coli* strains, let's create three generic species: rocks, papers, and scissors. In NetLogo, we do this with breed statements at the top of the **Procedures** tab:

```
breed [ rocks ]
breed [ papers ]
breed [ scissors ]
```

(In this document, new code added at each step will be in bold blue type; code that was created in a previous step will be in gray type, and italicized is shown with new code.)

To make things very simple, let's use turtle shapes that are red, green, and blue circles. Select the **Tools/Turtle Shapes Editor** menu option, and find the "circle" shape. We'll use this shape, but we'll create three custom versions of it: one colored red, one colored green, and one colored blue. We do this by selecting the "circle" shape, and then pressing the **Duplicate** button; you should now be looking at the **Shape** editing window.



Give this shape a new name (such as "rocks-circle"), then click on the big gray circle. When it's selected, click on the red tile in the color palette to the left. This changes the color of the circle to red. Click **OK**, and the new shape appears in the shapes list.

Repeat the above process – duplicating the original circle, giving a new name to the new shape, and changing the color – to create green and blue circle shapes for papers and scissors (respectively). When you're done, close the **Turtle Shapes Editor** window.

Now our model needs a `setup` procedure:

```
to setup

end
```

Notice that we've included the `end` statement that ends the procedure; this helps us maintain correct syntax as we write our code.

Let's begin `setup` by clearing everything, and telling NetLogo what shapes our breeds use:

```
to setup
  clear-all
  set-default-shape rocks "rocks-circle"
  set-default-shape papers "papers-circle"
  set-default-shape scissors "scissors-circle"
end
```

The shape names must be in double quotes, and written exactly as you wrote them when creating the custom shapes.

Next, we need to create the turtle agents. In our previous work with NetLogo, we saw that the observer can create turtle agents with the `create-turtles` (or `create-<breeds>`) statement, and turtle agents can create other turtle agents with the `hatch` statement. In this case, we'll use a third method: since we want exactly one turtle agent in each patch, we simply ask each patch agent to `sprout` one turtle on that patch:

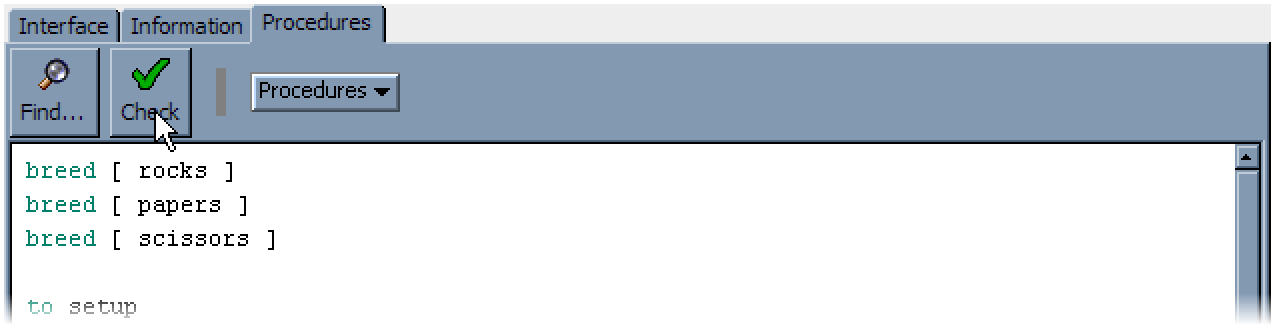
```
to setup
  clear-all
  set-default-shape rocks "rocks-circle"
  set-default-shape papers "papers-circle"
  set-default-shape scissors "scissors-circle"
  ask patches [
    sprout 1 [

    ]
  ]
end
```

In the innermost set of brackets, we need to write the code that each of the turtle agents will execute as part of the **setup** procedure. The most important thing that each turtle must do is select its strategy – which, in this model, is also its breed. This will be a random selection, using the **one-of** reporter. This reporter selects an item at random from a list (or an agentset). In this case, the list will include the three breeds we've defined.

```
to setup
  clear-all
  set-default-shape rocks "rocks-circle"
  set-default-shape papers "papers-circle"
  set-default-shape scissors "scissors-circle"
  ask patches [
    sprout 1 [
      set breed (one-of (list rocks papers scissors))
    ]
  ]
end
```

Here, the inner parentheses around **list rock papers scissors** are required; if they're omitted, NetLogo will report an error.[2] The outer parentheses, however, are optional; they're simply used to make it clearer to the reader that the code is evaluating a non-trivial expression, and assigning the result to **breed**.

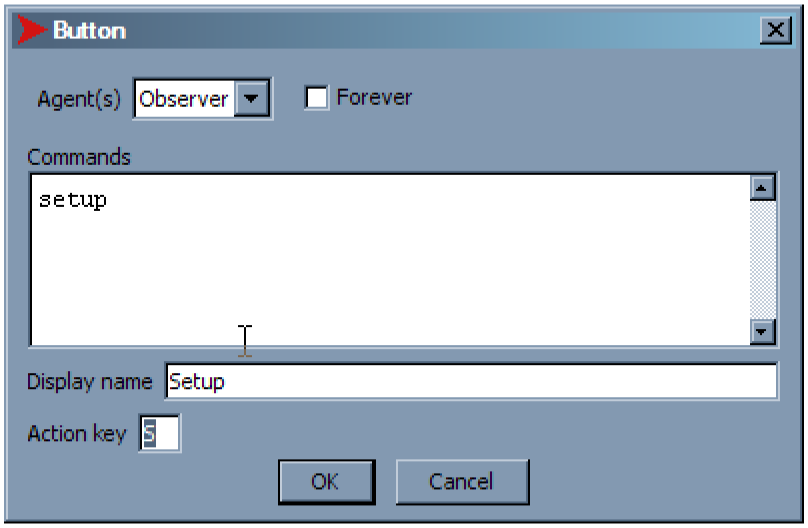
When writing code in the **Procedures** tab, don't forget to use the **Check** button to validate the syntax of your code. When there are errors, the messages displayed, though useful, won't always tell you exactly what's wrong; however, the cursor is usually positioned at the first point in your code that NetLogo is unable to understand.



```
breed [ rocks ]
breed [ papers ]
breed [ scissors ]

to setup
```

2 By default, the **list** statement creates a list containing the two values that follow it. If we need a list with some other number of items, **list** and the values to be included must be enclosed in parentheses.

Now, let's make a button to call the **`setup`** procedure. In the **Interface** tab, create a new button[3], and fill in the dialog options:



Agent(s): Observer
Forever: ☐

Commands:

```
setup
```

Display name: Setup

Action key: S

(Note that the **Display name** and **Action key** values aren't essential, and may be changed or omitted; the important part in this case is what we type in **Commands**.) After specifying **`setup`** in the **Commands**, click **OK**.

At this point, it would be a good idea to check and save your model. Remember to end the filename with ".nlogo".

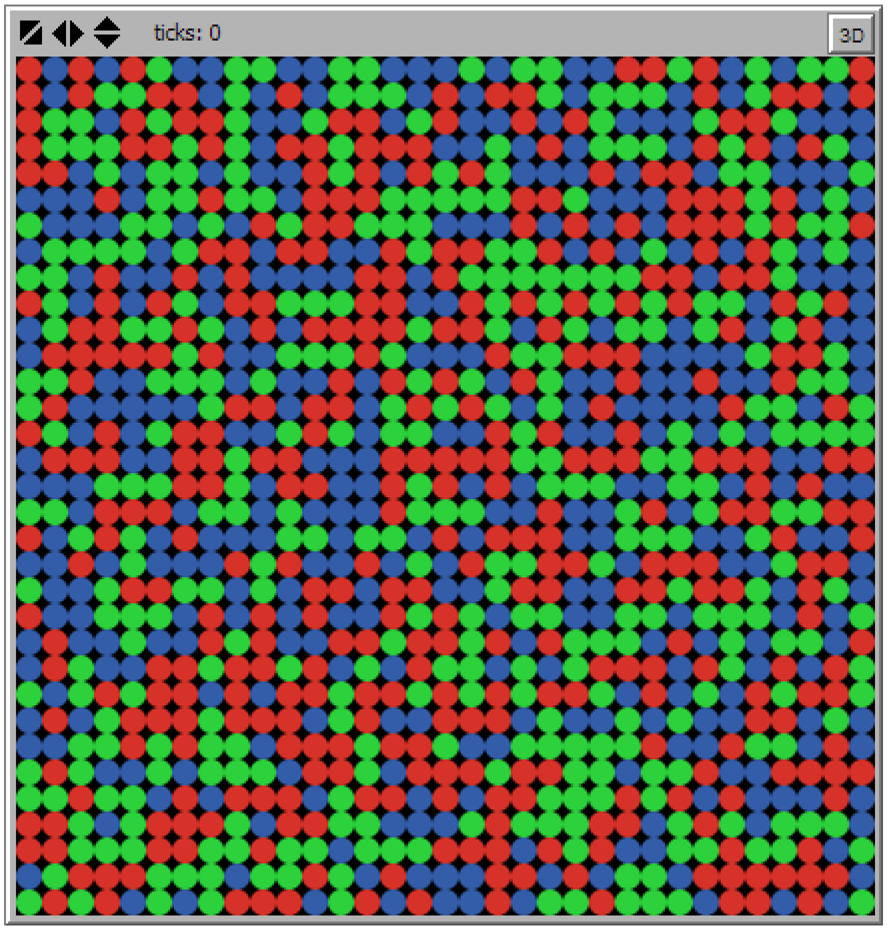
3 You can create user interface elements in any of three ways:

a) Select the element you want to add from the pull-down menu to the right of the Add button at the top of the Interface tab; then click in the whitespace below that, where you to place the new element.

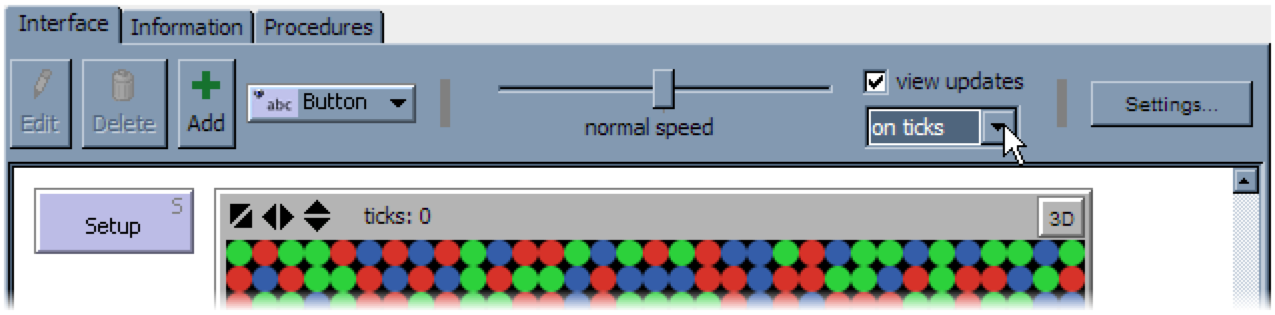
b) If the element you want to add is already the currently selected item in the pull-down menu, you can simply click the Add button, and then click in the whitespace.

c) Right-click (or Ctrl-click) in the whitespace, where you want to place the new element. From the pop-up menu that appears, select the desired element.

Test your work so far by clicking the **Setup** button;  you should see something like this:



When you click the **Setup** button a few times, you'll notice that the screen sometimes gets updated before the setup process is complete, making the display appear uneven. There are a few ways to fix this; the simplest is to tell NetLogo that it should only update the display when the **tick** command is executed (more on that later), or when a button finishes processing. To do this, make sure the **view updates** checkbox at the top of the **Interface** tab is checked, and select **on ticks** from the pull-down menu below the checkbox:



## *Task 2: Teaching the Agents to Play Rock-Paper-Scissors*

In each iteration, each turtle agent selects one of its neighbors at random, and compares that neighbor's breed to its own breed. If the breeds are the same, the competition ends in a tie. If the breeds are different, then the winner is determined by the normal rules of Rock-Paper-Scissors. The breed of the loser will then change to that of the winner.

Create a new procedure called **`play`**, which will eventually implement the above behavior:

```
to play

end
```

Now we need to add the code to select an opponent at random. You might remember that there's a NetLogo reporter, **`neighbors`**, which gives us the set of the neighboring patches (i.e. those directly or diagonally adjacent to the current patch). In this case, we'll use a variation of that, **`neighbors4`**, which returns only those neighboring patches directly adjacent, but not those diagonally adjacent. We'll then use another command, **`turtles-on`**, to get the set of turtle agents that are standing on those patches. Finally, we'll use **`one-of`** (which we've used before) to select one of those turtles at random.

```
to play
  let opponent (one-of turtles-on neighbors4)
end
```

Now that we know who the opponent is, we can use **`[breed] of opponent`** to get its breed, and compare it to that of the current agent. Then, we can check for a tie, and end the procedure if there is one.

```
to play
  let opponent (one-of turtles-on neighbors4)
  let opponent-breed ([breed] of opponent)
  if (breed = opponent-breed) [
    stop
  ]
end
```

Note that the parentheses around **`breed = opponent-breed`** are optional.[4]

4 In many programming languages (in particular, virtually all that have evolved or borrowed from the B programming language – e.g. C, C++, C#, Java, JavaScript, ActionScript, Perl, PHP) parentheses are required for the conditional expressions used with **`if`**. However, even when they aren't required, using parentheses around conditional expressions can make code easier to read and understand.

These are the three conditions under which the current agent wins the contest:

- The current agent's breed is **rocks**, and the opponent's is **scissors**.
- The current agent's breed is **papers**, and the opponent's is **rocks**.
- The current agent's breed is **scissors**, and the opponent's is **papers**.

Since we've already ruled out ties, we only need to check for combinations of breeds given in the conditions above; any other combination means the current agent loses the contest.

Pay close attention to the next bit of code; the brackets and parentheses are very important.

```
to play
  let opponent (one-of turtles-on neighbors4)
  let opponent-breed ([breed] of opponent)
  if (breed = opponent-breed) [
    stop
  ]
  ifelse ((breed = rocks and opponent-breed = scissors)
      or (breed = papers and opponent-breed = rocks)
      or (breed = scissors and opponent-breed = papers)) [
    ask opponent [
      set breed ([breed] of myself)
    ]
  ] [
    set breed opponent-breed
  ]
end
```

Be sure to check your code, as usual. (Note that inside the **ask opponent […]** block, **myself** refers to the agent doing the asking, not to the agent being asked.)

## *Task 3: Telling the Agents to Play*

The last procedure we need to write (for now) is one in which the observer tells all of the turtle agents to play. As we often do, we will call this procedure **go**:

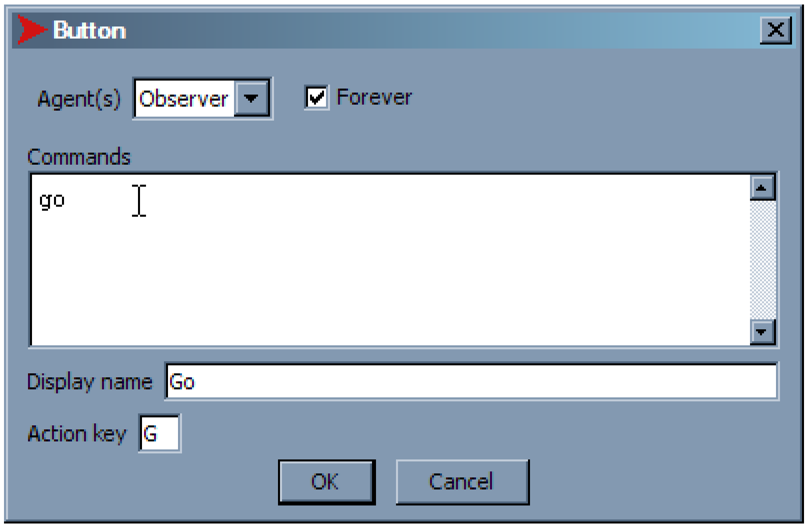
```
to go
  ask turtles [
    play
  ]
  tick
end
```

The contents of our **Procedures** tab should now look something like this:

```
breed [ rocks ]
breed [ papers ]
breed [ scissors ]

to setup
  clear-all
  set-default-shape rocks "rocks-circle"
  set-default-shape papers "papers-circle"
  set-default-shape scissors "scissors-circle"
  ask patches [
    sprout 1 [
      set breed (one-of (list rocks papers scissors))
    ]
  ]
end

to play
  let opponent (one-of turtles-on neighbors4)
  let opponent-breed ([breed] of opponent)
  if (breed = opponent-breed) [
    stop
  ]
  ifelse ((breed = rocks and opponent-breed = scissors)
      or (breed = papers and opponent-breed = rocks)
      or (breed = scissors and opponent-breed = papers)) [
    ask opponent [
      set breed ([breed] of myself)
    ]
  ] [
    set breed opponent-breed
  ]
end

to go
  ask turtles [
    play
  ]
  tick
end
```

Check (and fix, as necessary) and save the model.

Finally, let's make a **Go** button. Switch back to the **Interface** tab, and create a new button to call the **go** procedure. This time, make sure to check the **Forever** option:



Agent(s): Observer
[✓] Forever
Commands:

```
go
```

Display name: Go
Action key: G

Click **OK**. Save your model, and then click the **Go** button.

Run the model a few times. What do you notice? How would you characterize the results?

## *Task 4: Space Experiments*

After getting a feel for how the model runs, use the **Settings...** button in the **Interface** tab to try the following changes. After each change, run a few trials, and see if the results are significantly different from what you observed before.

1. Make the world much smaller, by changing **max-pxcor** and **max-pycor** to 12 (you may want to increase the **Patch size** to 20).
2. Change the topology from a torus to a rectangle, by unchecking the **World wraps horizontally** and **World wraps vertically** options.
3. Make the world a torus again, but much larger: change **max-pxcor** and **max-pycor** to 25 (you'll probably need to change the **Patch size** to 10, in order to see the entire NetLogo world on the screen), and check the **World wraps horizontally** and **World wraps vertically** options.

What differences (if any) in the aggregate behavior did you notice after each of the change to the model?

## *Task 5: Plotting the Populations*

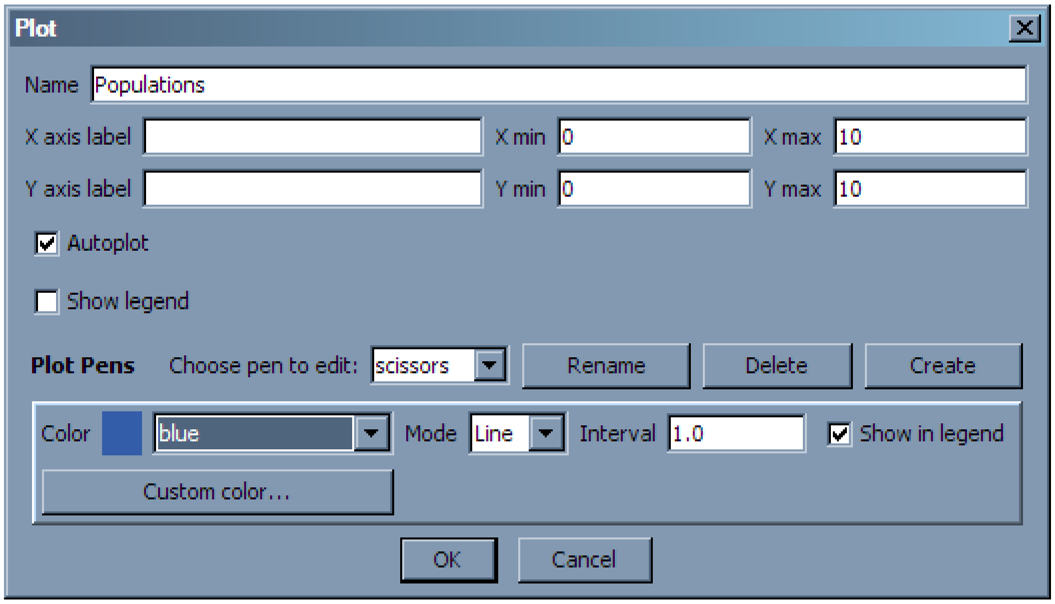
While we can get a general sense of the relative proportions of the three breeds in our model, it's difficult to get a more precise picture just by watching the model. Something we could do to improve the situation is plot the three different populations over time.

Switch back to the **Interface** tab and create a new plot. (You create a plot the same way you create a button, except that you select **Plot** from the menu of user interface devices.)

In the **Plot** properties, make the following changes:

1. Change the **Name** value to "Populations".
2. Use the **Rename** button to change the name of the default pen to "rocks".
3. Use the **Color** pull-down menu to select the color red for the "rocks" pen.
4. With the **Create** button and **Color** pull-down menu, create a "papers" pen, using the color green.
5. Repeat step #4, creating a blue "scissors" pen.

The **Plot** properties window should now look like this:


Plot

Name: Populations

X axis label: (empty)  X min: 0  X max: 10

Y axis label: (empty)  Y min: 0  Y max: 10

[✓] Autoplot

[ ] Show legend

Plot Pens  Choose pen to edit: scissors

Color: blue  Mode: Line  Interval: 1.0  [✓] Show in legend

Click **OK** and save your model.

Now we need to write the code that will do the plotting. The basic operations used in plotting most graphs are these:

1. Use **`set-current-plot`** to specify the name of the plot in which subsequent plotting operations will be performed. (When there is only one plot, this command is not strictly necessary; however, it's a good idea to get in the habit of using it.)
2. Use **`set-current-plot-pen`** to select one of the previously created pens for plotting.
3. Use **`plot`** (or **`plotxy`**) to plot a data point; this point will usually be connected by a line from the previous point plotted with the pen selected in step #2.
4. Repeat steps #3 and #4 as necessary, to plot the current data values for all pens.

We'll follow the same steps in our model. In our case, the data we want to plot is simply the population of each breed; we can get this by using the **`count`** ***`agentset`*** reporter expression, which returns the number of agents in the specified agentset (which can be a breed).

Switch to the **Procedures** tab and put the plotting code in a new **`update-plot`** procedure (note that the names of the plot and the plot pens are all given in double quotes; these are required):

```
to update-plot
    set-current-plot "Populations"
    set-current-plot-pen "rocks"
    plot count rocks
    set-current-plot-pen "papers"
    plot count papers
    set-current-plot-pen "scissors"
    plot count scissors
end
```

Of course, writing a procedure for plotting isn't enough: we also need to call that procedure, so that our plot is automatically updated as the model runs. A logical place to do this is in the **`go`** procedure, just after all of the agents have completed a round of play:

```
to go
    ask turtles [
        play
    ]
    tick
    update-plot
end
```

So far, your code will plot the population at the end of each iteration. However, it's probably a good idea to plot the initial values as well, before the model runs. To do that, simply invoke the **`update-plot`** procedure from the **`setup`** procedure:

```
to setup
    clear-all
    set-default-shape rocks "rocks-circle"
    set-default-shape papers "papers-circle"
    set-default-shape scissors "scissors-circle"
    ask patches [
        sprout 1 [
            set breed (one-of (list rocks papers scissors))
        ]
    ]
    update-plot
end
```

Check your code, save your model, and test it. What do you notice about the variability of the breed populations over time? Does the amount of variability stay mostly constant?

If some of your experiments in task #4 ended with one breed taking over, try those experiment again. What do you notice about the variability of the populations shortly before one breed (and then another, necessarily) goes extinct?

## *Next Steps*

The model we've built so far implements the agent behaviors described earlier, and includes a plotting feature that helps us observe the aggregate behavior over time. We can conduct experiments to examine the sensitivity of our model to world size and topology, and use the results of those experiments to help understand analogous properties of real world ecosystems with cyclical dominance.

Of course, there's almost always something else that can be added to any model. In this case, there are some very powerful and interesting features that can be added without too much effort. However, for the next three tasks, the code won't be shown – though there are some hints to help you get started. This will give you a chance to practice your NetLogo programming skills, and add to the NetLogo commands and reporters you've used.

In completing these tasks, don't forget to keep the NetLogo Dictionary (available under the **Help** menu) open in another window! Even for experienced programmers, having the documentation handy is always a good idea.

### *Task 6: Stopping Automatically*

There isn't much point in allowing the simulation model to continue running when turtles of only one breed remain. Try to add code to the **`go`** procedure that checks for this condition, and executes the **`stop`** command accordingly.

Here are some hints that should help you get started:

- **`count turtles`** returns the total number of turtles, while **`count`** ***`agentset`*** returns the number of agents in an agentset (which can be a turtle breed). For example, **`count rocks`** returns the number of turtles of the rocks breed.
- To test this change, you might need to make the world smaller, at least temporarily, so that it's more likely that one breed will take over.

### *Task 7: Resetting the Plot*

After this model runs for several hundred ticks, the plot becomes little more than a jumble of colored lines, almost impossible to read. Add a button to your model (along with any necessary additions or changes to the code) to clear the plot area, so that new plot updates will be easier to read. Further, do this in such a way that the plotted X values still match the overall tick count of the model (as seen at the top of the NetLogo world), without resetting the tick count when the plot is reset.

There are a few ways to accomplish this task, but these hints should help in any case:

- **`clear-plot`** resets the current plot to its default values (those specified in the **Plot** dialog boxes); **`clear-all-plots`** does this for all plots in a model. **`clear-all`** also resets all plots (along with the built-in and custom agents and globals).
- **`plotxy`** plots a point specified by explicit X and Y values; the **`plot`** command doesn't require specification of an X value. The first time **`plot`** is executed for a given plot pen, it automatically uses an X value of 0; each time it's executed for the same plot pen in that plot, the X value is increased by the value specified in the **Plot** dialog, in **Interval**. When the plot is reset (by **`clear-plot`**, **`clear-all-plots`**, or **`clear-all`**), the X value is reset to 0 as well.
- **`set-plot-x-range`** can be used to control the minimum and maximum X values in a plot. (If **Autoplot** is checked in the **Plot** dialog, the plot will automatically expand when a point outside the current X and Y axis ranges is plotted.)

## *Task 8: Shaking the Flask*

Some of the experiments that inspired this model (and many others) involved shaking a flask of *E. Coli* periodically, so that the bacteria wouldn't always remain in the same neighborhoods. Multiple experiments were performed, with different amounts of mixing, to see what effect, if any, the mixing would have on the aggregate behavior of the ecosystem. In our model, however, there isn't any mixing: each turtle agent remains stationary, always competing with the same set of neighbors.

For this task, change the model so that mixing is implemented via rearrangement of some number of agents at the start of each iteration, while preserving the one-turtle-per-patch scheme. The amount of mixing should be user-specifiable (including the possibility of no mixing at all), with a fine level of control.

There are many approaches to this problem, and no single best approach. These tips should help with at least some of the alternative methods:

- In NetLogo, a slider is generally the best user interface device for allowing the user to specify a single value within a wide range of values.
- The **`repeat`** command can be used to execute a command block a specified number of times.
- A turtle agent can be moved so that it is centered on a patch or another turtle with the **`move-to`** command.
- The **`n-of`** reporter is used to select a given number of agents (or list items) at random from an agentset (or list), and return the result as an agentset (or list).
- A randomly ordered list of agents can be constructed from an agentset with the expression **`[self] of`** ***`agentset`***.
- The **`foreach`** command and the **`map`** and **`reduce`** reporters are very useful constructs for iterating over and processing the items in a list.

Another approach altogether involves treating the physical locations as nodes on a lattice network, and randomly rewiring some fraction of the network – either initially (quenched randomness), or as the model proceeds (annealed randomness) [1],[2]. The methods described above are most analogous to the annealed randomness of the lattice-based approach.

## *References*

[1] K. I. Kircher, “Emergent Behavior of Rock-Paper-Scissors Game”, Term essay, Physics 569: Emergent States of Matter, University of Illinois at Urbana-Champaign, IL, USA, May 2006.

[2] G. Szabó, A. Szolnoki, R. Izsák, “Rock-scissors-paper game on regular small-world networks”, *Journal of Physics A: Mathematical and General*, vol. 37, no. 7, pp. 2599-2609, Feb. 2004.