Building a Simple Rock-Paper-Scissors Model in NetLogo

In this activity, we'll build a NetLogo model in which the turtle agents play a simplified form of Rock-Paper-Scissors.

Background

A natural first question might be: What is it about Rock-Paper-Scissors that makes it a good subject for a NetLogo model? In fact, it's interesting in part because even a basic implementation includes a number of important intermediate 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 collection of *Escherichia coli*, normally called *E. coli*. There are three types of *E. coli*:

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

The three types have certain 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 “wins” and 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 wins against R.
- Type C wins against type S because the colicin produced by type C kills type 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

In the case of *E. coli*, the parallels in behavior between the real world and the model we will build is very close, with similar correspondence between the simulation results and the *E. coli* experimental results. There are other biological systems with cyclical dominance that are not quite as closely modeled with this approach[1], but a model like this can still help us learn something interesting about those ecosystems as well.

The basic characteristics of the model are as follows:

- The terrain will be completely populated by turtle agents – i.e. there is one turtle agent per patch.
- Turtles will not move.
- Each turtle can only play against those turtles directly adjacent to it – i.e. those located immediately to the North, East, West, or South of it.
- Each turtle will start out with a strategy of rock, paper, or scissors, selected randomly. It will maintain this strategy until it is defeated, at which point it adopts the strategy of the victor. Whenever it changes to a new strategy, it will continue with the new strategy until it is once again defeated, and once again adopts the victor's strategy.

1Another 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 they 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 do not guard as many females as the orange throat males do, but they guard them more carefully.

Orange throat males win against blue throat males, simply due to their strength. Blue throat males lose against orange throat males, but can win against 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 cannot sneak into blue throat males' territories due to the latter's vigilance, but they can successfully sneak into orange throat males' territories and steal females. In fact, this system is not a simple RPS game because there are two types of females, and 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

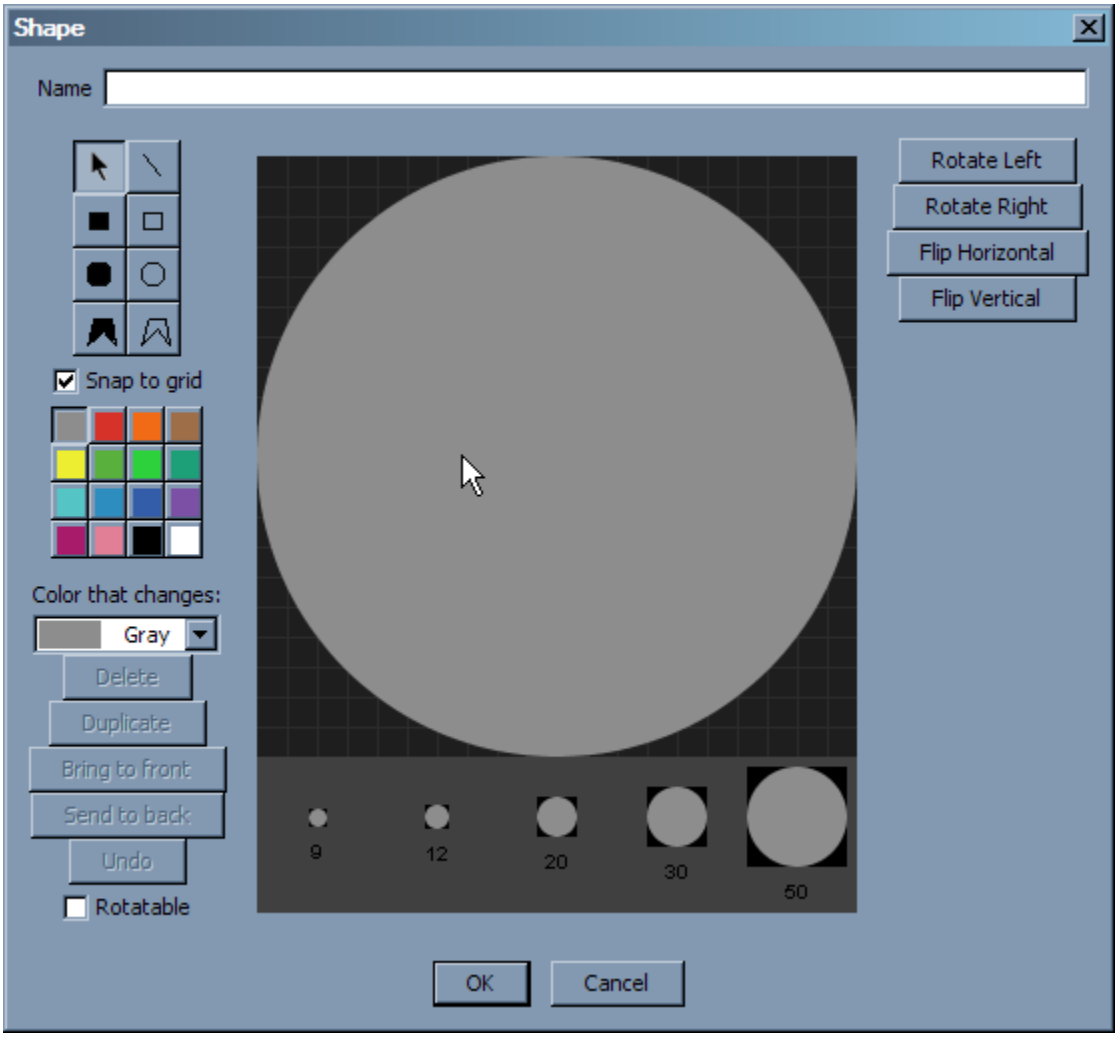
Rather than make this model specifically about the three *E. coli* strains, let's simply have three generic species: rocks, papers, and scissors. In NetLogo, we do this with three breed statements (at the top of the **Procedures** tab):

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

(In this document, any new code added at each step will be italicized and in blue; code that was created in a previous step will be in normal style and black.)

To make things very simple, let's make the turtle shapes circles, colored red, green, and blue (respectively). Select the **Tools/Turtle Shapes Editor** menu option, and find the "circle" shape. We will use this shape – but to make things easier in our code, we will create three special versions of this shape: one colored red, one colored green, and one colored blue. We do this by selecting the "circle" shape, and then pressing the **Duplicate** button.

We should now see the **Shape** editing window:



Give this shape a new name (I suggest "rocks-circle" or "rock"), then click on the big gray circle. When it is selected, click on the red tile in the color palette to the left. This will change the color of the circle to red. Click **OK**, and our new shape appears in the shapes list. (Important: Changing the color in the **Color that changes** pull-down will *not* change the color of the selected shape; instead, it will just tell NetLogo that the parts of the shape drawn in that color in the **Shape** editor will be the parts of the shape that change color with `set color`.)

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

Now our model needs a setup procedure:

```
to setup

end
```

(Note that the above shows a blank line in the procedure; this is where we'll start writing the statements that will instruct NetLogo how to set up the model.)

We'll begin the setup by clearing everything, and then telling NetLogo what shapes we will use for our breeds:

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

Note that the shape names must be in quotes, and must be written exactly as we wrote them when creating our custom shapes.

Next, we create our 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 will use a third method: since we want exactly one turtle (whether of the rock, paper, or scissors breed) in each patch, we will simply tell each patch agent to "sprout" exactly 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
```

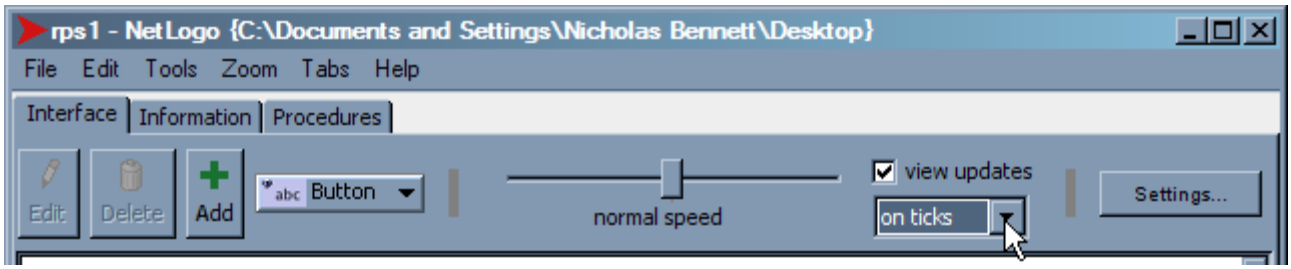
In the innermost set of brackets, we will write the code that each of the new turtles will execute as part of the setup procedure. Obviously, the most important thing that each turtle needs to do is select its strategy – which, in our model, is also its breed. We will do this randomly, using the `one-of` reporter. This reporter selects one of the items in a list at random. To use it in our model, we will create a list of the three breeds, and select from that list:

```
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
```

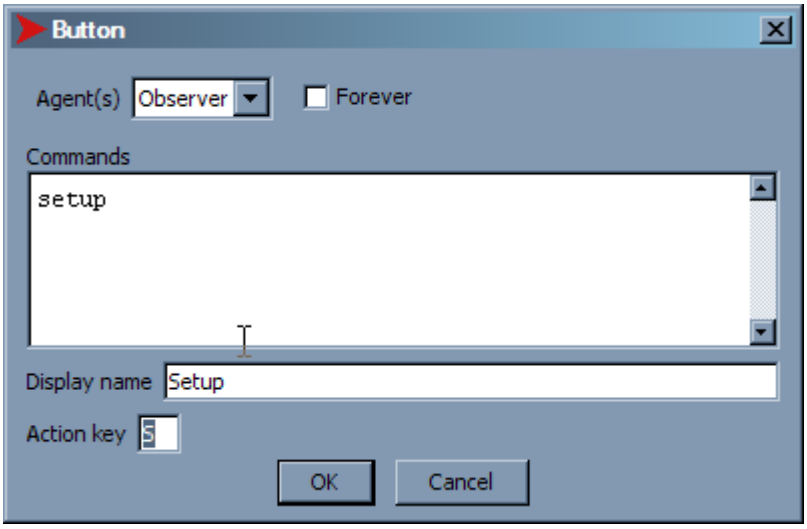
In this case, the parentheses around "`list rock papers scissors`" are essential; if they are omitted, NetLogo will report an error.

At this point, it's a good idea to check and save our code. (Remember to end the filename with ".nlogo".)

Now, let's create a button that will run the setup procedure. But first, let's make a change that will make the model display update a bit more smoothly. Switch to the **Interface** tab, and look for the **view updates** checkbox (it will be close to the top of the window, just to the right of the speed slider). Make sure that the checkbox is checked, then pull down the menu below it, and select **on ticks**:



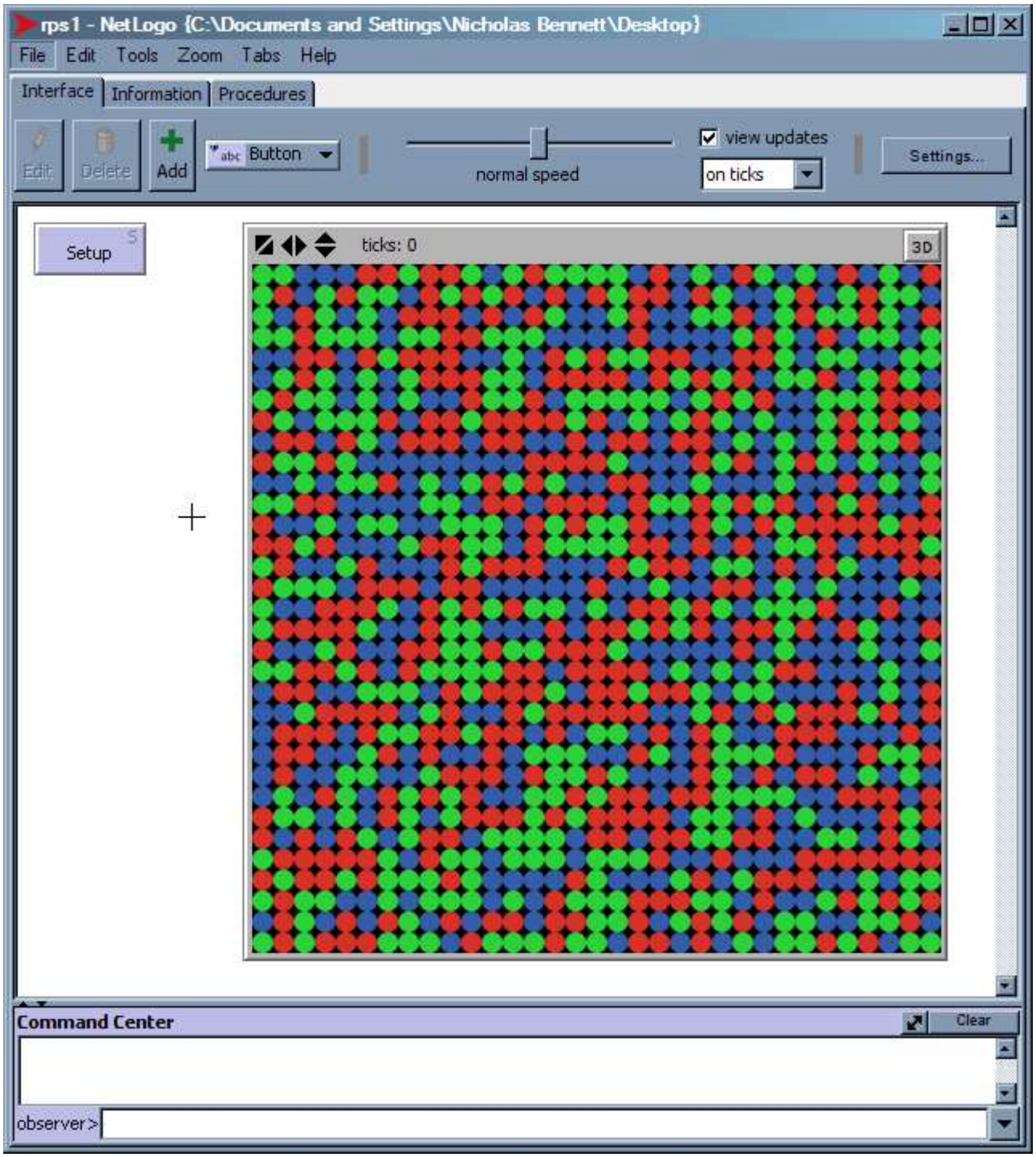
Now, for our setup button: right-click on the whitespace to the left of the NetLogo "world", and fill in the Button dialog options:



(Note that the **Display name** and **Action key** values are not essential, and may be changed or omitted; the **Commands** are what is important here.) After specifying "setup" in the **Commands**, click **OK**.

Save your model again.

Now, let's see what it does so far: click your **Setup** button, and see what happens. You should see something like this:



Once you have fixed any problems that occurred, be sure to save again.

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

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

Let's write a procedure called "play", which will eventually implement the logic described above:

```
to play

end
```

First, we select the opponent at random. You might remember that there is a NetLogo command, `neighbors`, which gives us the set of the neighboring patches (i.e. those adjacent to the current patch). In this case, we will use a variation of that, `neighbors4`, which returns only those neighboring patches to the sides or up or down, but not those located diagonally. We will then use another command, `turtles-on`, to get the set of turtle agents that are standing on those patches. Finally, we will use `one-of` (which we have 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 our opponent is, we can compare breeds. We will do this comparison a few different times, for the different possibilities; to start, let's look for ties (if there is one, the turn is over):

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

Now, let's identify the 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 have already ruled out ties, we only need to check for the above combinations; if we don't find a match, that means the current agent loses the contest.

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

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

Check (and fix, as necessary) and save your code.

Task 3: Telling the Agents to Play

The last procedure we need to write (for this basic version of the model) 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
```

Check and save the model.

Now, the contents of our **Procedures** tab should look something like this:

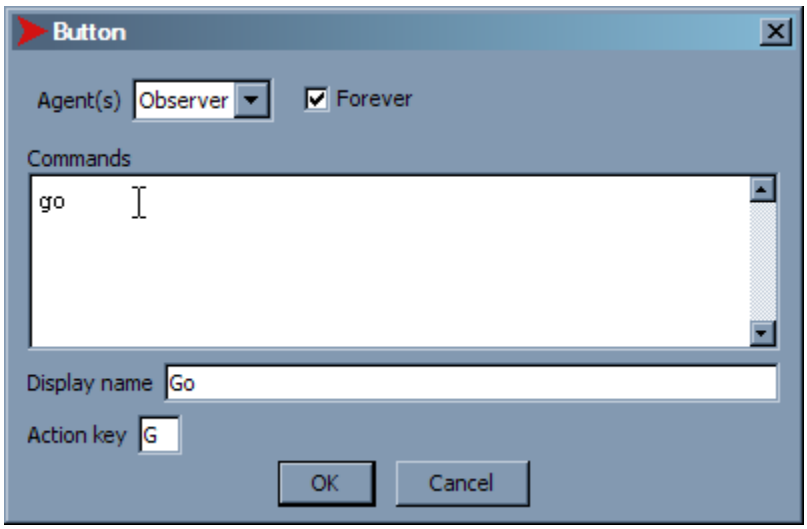
```
breed [ rocks ]
breed [ papers ]
breed [ scissors ]

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 my-breed breed
    let opponent-breed [breed] of opponent
    if (my-breed = opponent-breed) [
        stop
    ]
    ifelse (my-breed = rocks and opponent-breed = scissors)
            or (my-breed = papers and opponent-breed = rocks)
            or (my-breed = scissors and opponent-breed = papers) [
        ask opponent [
            set breed my-breed
        ]
    ] [
        set breed opponent-breed
    ]
end

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

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:



Click OK. Save your model, and then click your new **Go** button to see what happens!

Task 4: 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. Change the topology from a torus to a rectangle, by unchecking the **World wraps horizontally** and **World wraps vertically** options.
2. Return the world to a torus, but make it much smaller, by changing the **max-pxcor** and **max-pycor** values to 10 or 12.
3. Make the world much larger, by changing **max-pxcor** and **max-pycor** to 50 (you will probably want to change the **Patch size** to 5, in order to see the entire NetLogo world on the screen).

Discussion

1. If we view our model as an ecosystem, which of the following types of interactions are present?
    - Predation
    - Parasitism
    - Symbiosis
    - Competition
    - Cannibalism
2. Did you try any of the variations described on the previous page? If so, what differences (if any) in the model behavior did you notice?

Task 5: Plotting the Populations

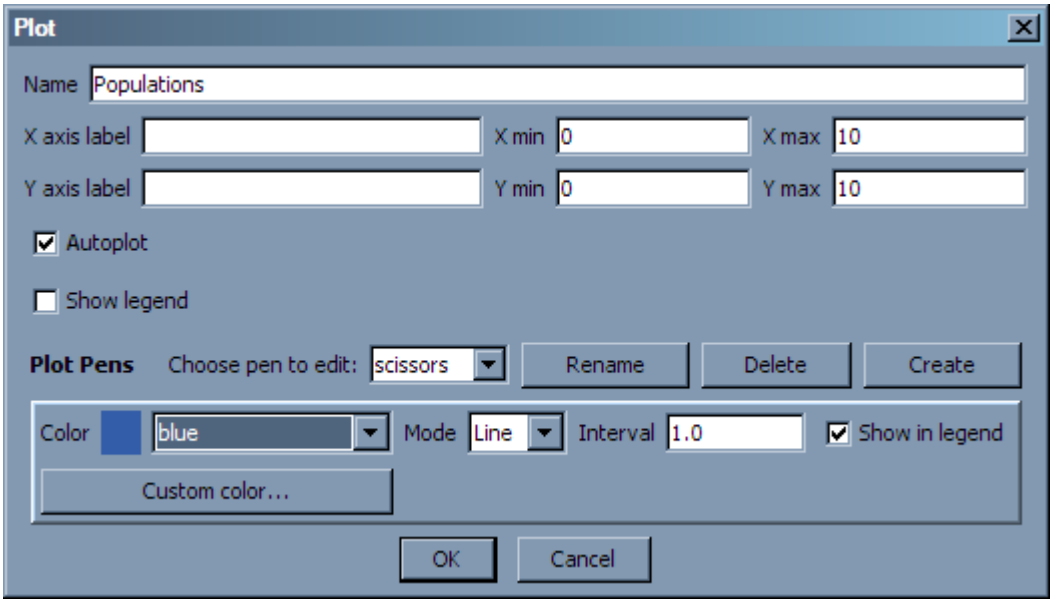
While we can get a general sense of the relative proportions of the three breeds in our model, it's difficult to get a more precise picture just by watching the model. Something we could do to improve the situation is plot the three different populations over time.

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:



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. This reporter 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, and just before `tick` is used to update the simulation time counter:

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

Check your code and save your model.

Test your model. If some of your experiments in task #4 ended with one breed taking over, try those experiment again, as well.

Discussion

1. What do you notice about the variability of the breed populations over time? Does the amount of variability stay constant? Does it change significantly?

2. What seems to happen to the variability of the breed populations shortly before one breed takes over completely?

3. There doesn't seem to be much point in allowing the simulation model to continue running when only one breed remains. How might we stop the model run when this happens?

4. After a model runs for several hundred ticks, the plot becomes little more than a jumble of color, almost impossible to read. Is there a command in the NetLogo dictionary which would let us reset the plot (possibly from a button) without restarting the simulation?

Optional Task: Shaking the Flask

In one set of *E. coli* experiments, one of the experiments described included putting the *E. Coli* in a flask, and shaking it periodically, so that the individual bacteria didn't always remain in the same neighborhoods. In our model, however, each turtle agent remains stationary, and only competes with its immediate neighbors. Can you think of any changes we might make in our model, to match the shaken flask *E. coli* experiment?

Come up with a scheme which allows for a user-specifiable amount of rearranging of agents. Any rearrangement moves (whatever those consist of) should be performed before each tick. Also, in working out your approach, try to find one which gives a fine level of control over the amount of randomness, and a wide range of values.