A simple implementation of Schelling's segregation model in NetLogo

Fabrizio Iozzi
Dondena Centre for Research on Social Dynamics
Università Bocconi

Abstract

We present an introduction to the NetLogo simulation environment using Schelling's Segregation Model presented by Nobel Prize Winner Thomas Schelling in 1978. While reviewing the model, its Netlogo implementation is described step by step, using visual tools and the needed programming in the Netlogo language. Two extensions to the original model are proposed and programmed. All the models are fully described in the text.

Keywords: NetLogo, Schelling, simulation, agent-based, segregation

1 Foreword

This document is an introduction to NetLogo [6] using the Segregation Model presented by Nobel Prize Winner Thomas Schelling in [4]. It assumes no previous knowledge of programming and it is aimed at graduate students in the fields of Economics and Social Sciences who want to experiment with agent based simulation in the most simple way. Those with some programming skills should take a look at one the many simulation toolkits available on the Internet; a good starting point is [1].

2 The Schelling segregation model

Thomas Schelling firstly introduced his models of segregation in a group of articles in the late sixties, early seventies. Those ideas have been collected in his book [5].

Schelling developed and analyzed two models. In the one dimensional model a population is randomly displaced along a straight line. There are two groups, "circles" and "crosses". For each person–an agent in the contemporary notation–a neighbor is defined, consisting of the other agents laying on the left and on the right of each agent. The size of the neighbor is the number of agents on the right (or on the left) that are in the neighbor. Each agent is "happy" or not depending on the number of its neighbors: if the percentage of neighbors of the same breed over the total number of neighbors is under a fixed threshold, the person is unhappy. Therefore, in the situation depicted in figure 2, assuming the neighbor is of size 2 and that the threshold for happiness is 50%, the red circle is unhappy because only one out of its four neighbors, i.e. the 25%, is of its kind, while the blue circle is happy because the majority of its neighbors is made up of circles. Each unhappy agent moves to another empty place on the line or simply inserts itself between two other agents in a place in which it would be "happy". The movements take place one after another, disregarding the fact that previously happy agents could become unhappy or viceversa. After each unhappy agent has moved, a new assessment of the happiness of all agents takes place and a new movements phase begins. This goes on indefinitely until there are no unhappy agents.

Schelling also proposes a two dimensional version of the same model. A grid hosts the agents and each agent has a neighbor made up of four cells, those immediately up, down, left and right to the agent. Rules of movement are similar to those in the one dimensional case.

Schelling shows that in both models, after a limited number of moves, a segregation pattern emerges, and that this "macro" behavior is largely unpredictable based only on the only simple rule each agent follows, namely that to move if it's unhappy.

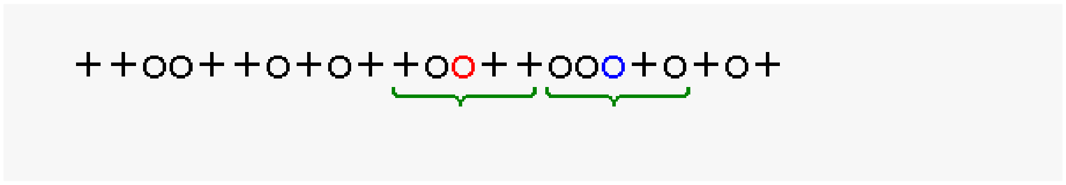


Fig. 1: Schelling's linear model

3 Setting up the simulation

NetLogo is a programmable simulation environment available for free. It includes a graphical user interface with three tabs for the design of the simulation space, its programming interface and the documentation of the model. The distribution includes the package, a user's manual and a number of ready models available under the `File-Models Library` menu item. We assume NetLogo is already installed on the user's machine.

3.1 The world

On opening NetLogo we are presented with the Interface tab. This is where the user creates the simulation space, the *World* in the NetLogo jargon. It is a rectangular checkerboard whose properties are set by clicking on the `Settings` button on the button bar. Each square is called a *patch*. The world has a coordinate system whose origin can be set to be anywhere. Departing a little from Schelling's description, we setup a square world, centered at the origin with, say, the maximum x coordinate equal to 20. The dimensions of the world will be 41×41. In the `World` section of the window you can also decide whether the world is a torus, i.e. if the right and left side of the world (and the upper and lower) are to be ideally joined together, transforming the rectangle into a doughnut. Unselect the two "World wraps

. . . ” checkboxes. Click `Apply` to see the changes to the world. If the world does not fit your screen you can modify the size of the patches.

3.2 Three parameters

Schelling's model is characterized by three parameters. They are the rate of similar wanted in a person's neighborhood, the rate of vacant space in the world and the proportion of the two populations in the world. In exploring the model we will be interested in seeing how the variation of these parameters affects the result of the simulation and NetLogo provides a simple tool to define a parameter and to modify it. This tool is a `slider` and it's available in the toolbar. Click on `slider`, move the mouse somewhere in the white space and click again to create a slider. You must provide some information. The name of the variable associated with the slider will be `perc-vacant-space` (be sure not to type spaces between the characters). Set the minimum value at 10, the increment at 5 and the maximum at 90; finally, set the start value (labeled simply as "Value") as 30. Click on `OK` and you'll see the slider on the screen. You can already play with it, moving the cursor on the right and on the left, changing the value of the parameter.

Similarly, create another two sliders, one for the `perc-similar-wanted` parameter (range 10-90, increment of 5 and start value of 50) and the other for the `perc-group1` with the same values. As there are only two groups, the proportion of one of them determines the other. Notice that the higher `perc-similar-wanted` is, the less persons of one group are willing to stay close to people of the other group. So, `perc-similar-wanted` could be thought of as a measure of intolerance.

If a mistake is made in creating a slider, right-click on it and select `Edit` to modify the parameters. If you want to move the slider on the screen, click anywhere outside an object and drag to include the object you want to move. It will become selected (dark gray color) and you can now drag it in the new place.

A final comment on what is meant by neighbor is required. Following [4], given a patch not on the border of the world, we will call the set of the 8 surrounding patches the neighbor of the patch. This definition is very common in simulations and is called "Moore's neighbor". Less frequently, a neighbor is defined as the 4 patches on the north, south, east and west of a patch. The latter neighbor is called a "von Neumann" neighbor. NetLogo allows for the usage of both type of neighbors, as you'll see later. If the world is not a torus, border patches do not possess an 8 patch neighbor: instead, they have either a 5 patch neighbor (for the patches along the borders) or a 3 patches neighbor (the patches on the four corners).

3.3 The Procedures tab. Global variables

A complete description of the model requires the definition of other variables. In this model we will add the variables in table 1

These variables are either quantities computed from the model at each time of its development or quantities we setup "behind the scenes" (like the two colors). To define these variables click on the `Procedures` tab and type the following instructions:

```
globals [
  percent-unhappy     ; perc of unhappy turtles
```

Tab. 1: Global variables

percent-unhappy	percentage of unhappy people
percent-unhappy-1	percentage of unhappy people in group 1
percent-unhappy-2	percentage of unhappy people in group 2
segregation-1	measure of segregation of people in group 1
segregation-2	measure of segregation of people in group 2
color1	color to characterize/represent people in group 1
color2	color to characterize/represent people in group 2

```
  percent-unhappy-1     ; perc of unhappy turtles of group 1
  percent-unhappy-2     ; perc of unhappy turtles of group 2
  segregation-1         ; measure of segr. for group 1
  segregation-2         ; measure of segr. for group 2
  color1                 ; color of group 1
  color2                 ; color of group 2
]
```

The declarations of these variables are included in the “globals block”. Globals means that these variables are visible (and changeable) in any part of the program. They do not depend on a particular patch or agent, as they are somewhat “collective” properties of the model.

The words following the semicolon are ignored by the engine but serve a very important purpose: they *document* the code, in the sense that they are written to explain the meaning of the previous (or following) commands or statements. These explanations are usually called *comments* and you’ll see many of them in the following. It is a good habit to comment the code while writing it because if you have to go back to the code after a while, comments will help you go back to your ideas and the way you expressed them in the syntax of the programming language.

3.4 Agents

In NetLogo, each patch can contain a variable number of *agents*. Agents are the “persons” who live and interact in the world. In Schelling’s model they are real persons of two different groups but they can be particles of whatever you want to model, e.g., particles of a fluid moving in some space, cars moving in the traffic, etc. The NetLogo Models Library contains a lot of models from a lot of fields of study.

NetLogo’s ancestor is Logo [3], a programming language invented in the 60s by a group of scientists led by mathematician, computer scientist and educator Seymour Papert. From the very beginning it was designed as a language understandable by children, and to make it more appealing for them its objects were idealized as turtles which children could move on the screen and instruct to perform some actions. For this reason, NetLogo agents are called `turtles`.

According to Schelling's model, at each step of the simulation each agent has to compute its state of happiness, dividing the number of surrounding agents of the same color by the total number of the surrounding agents. If this ratio is greater or equal to a preset threshold, the agent is happy, otherwise it is not and will move to find a better position. It is natural to bind to each agent the numbers involved in this computation, i.e. the 2 numbers of neighbors (of the same group and the total number of neighbors) and a third variable representing the state of happiness of the agent. This variable (we will call it `happy`) will have only two possible values, true and false. Variables of this kind are usually called boolean variables and in NetLogo documentation and examples their names usually end with a "?". However, We won't follow this convention.

To define this set of variables for agents, leave an empty blank line and then type the following:

```
turtles-own [
  happy            ; true if turtle has at least percent-similar-wanted
      of similar in its neighbor
  similar-nearby   ; counts neighbors of the same color
  total-nearby     ; counts the n. of neighbors
]
```

The name of the group that contains these variables is self-evident: `turtles-own` means quantities owned by each single turtle. In NetLogo, each turtle always has a set of parameters characterizing it. Some of them are built-in (e.g. the position of the turtle in the world, i.e. its coordinates), others are defined by the model and, in our case, are the previous quantities.

It's important to notice that the three variables are not independent and, in fact, could be reduced to two, the similar neighbors and the total neighbors, because the state of happiness is computed comparing the ratio of similars to total neighbors to the percentage of similars wanted. In fact, this kind of "saving" might be not so useful depending on the way the computation is actually performed (i.e. the way the simulation is programmed and the system which is run into). As a rule of thumb, a few more turtle variables add clarity without sacrificing performance.

3.5 The setup procedure

A typical NetLogo simulation consists of two main tasks: the setup of the environment with the desired features and the simulation itself, i.e. the changing of the state of agents according to some rules. In NetLogo these task are usually performed with two buttons, which we will name `Setup` and `Go`. Clicking on these buttons will trigger the execution of some commands that will be listed in the Procedures tab. We start here with the Setup button. Click on the "Button" button (!) on the toolbar, move the mouse somewhere in the white area and click: a button will appear and you will be asked to provide some parameters. All parameters but one can be left at their default value. The only thing that's to be typed is the name of the procedure you are going to call by pressing this button. As this is intended to be the setup button, we will type `setup` in the “Commands” textbox. Pressing `Ok` creates the button with its label (or caption) equal to “setup”. As there is no setup procedure for the moment, the name on the button and the name “Interface” on the tab are displayed in red. NetLogo is warning us that a button points to the execution of an unknown procedure. To fix things, go to the procedures tab and type the setup procedure.

As before, leave a blank line (just for clarity purposes) after the previous declaration and type the following text:

```
 1 to setup
 2   ca                   ; clear all (not really useful explanation)
 3   set color1 45         ; yellow
 4   set color2 85         ; cyan
 5   let number-of-people int(world-width * world-height * (1 - perc-
       vacant-space / 100))
 6   ask n-of number-of-people patches [
 7     sprout 1 [set color color2]
 8   ]
 9   ask n-of int(number-of-people * perc-group1 / 100) turtles [
10     set color color1
11   ]
12   update
13 end
```

The procedure begins in line 17 with `to setup`. The keyword `to` tells NetLogo we are about to write a procedure for doing something. `setup` is the name of the procedure and corresponds to the name we wrote in the button dialog. The sequence of commands ends in line 30 with the `end` statement.

The first command is `ca`, which is an abbreviation for `clear-all`. `ca` clears all, i.e. resets all global variables to zero, destroys all turtles, resets all patches, drawing, plots, and output. Usually `ca` is put at the beginning of the setup procedure to make sure there are no legacies from previous runs of the model. As in the variable declaration, it is possible to comment this line putting a semicolon and the comment after it. It is not useful to comment this line because `ca` is a predefined command and is very common so one expects the average user to understand its meaning without further explanation.

Then we set the color of the two groups. This is done by assigning a number to the variable `colorx` (with $x = 1, 2$). To see which number corresponds with which color, you can select `Tools-Color Swatches` from the NetLogo menu. As we are going to distinguish people by their color, we will refer to color1 and color2 very frequently in the rest of the program. This is why we choose to use variables and not explicit colors. If we wrote, say, “yellow” then everything would have been related to “yellow” and if we wanted to change the color from yellow to red we would have to modify the code in many points.

Now it’s time to create agents (i.e. turtles). First of all we must know how many agents are to be created. The dimensions of the world are already setup and there are 2 NetLogo built-in variables that tell us what they are: world-width and world-height. Their product is the total number of patches in our world. The percentage of vacant space is indicated by the slider so the world will have

$$\text{world-width} \times \text{world-height} * \frac{\text{perc-vacant-space}}{100}$$

empty patches and

$$\text{world-width} \times \text{world-height} * \left(1 - \frac{\text{perc-vacant-space}}{100}\right)$$

agents (assuming that on one patch there can be only one turtle). This number is computed in the following line of the text and is assigned to the variable `number-of-people`. Notice that this variable is declared and initialized with a `let` command: it is a *local* variable, i.e. it is visible and accessible only from within this procedure. At the end of the setup procedure its value is lost. In the code there's an additional `int` that encloses the computation: it ensures that we get an integer number with the necessary rounding.

To actually create turtles there are two ways: create turtles and then place them in the world or browse the patches of the world and ask them to create turtles. We follow the second method: we will ask some patches to create turtles on them, we set turtle color to color2 (i.e. cyan) and then we will change a percentage of the cyan turtles into yellow ones. Lines 22–27 perform this task. The command `ask n-of x patches` asks x patches in the world to do what is enclosed in the following brackets. In our case x = number-of-people and we create one turtle on each patch. The command to create turtles is `sprout`, which accepts two parameters: how many turtles to create and what to do with them as soon as they are created. So we write `sprout 1 [set color color2]` which means "create a turtle on yourself and let the turtle execute the command `set color color2`". In this way we create a world with the right number of turtles but all the turtles are of the same color. To get the right world we need to modify a number of the turtles to change their color. The global variable `perc-group1` contains the percentage and the statements in lines 26–28 perform the required task. The `color` variable is a built-in turtle variable we can set via the `set color` command.

Notice that the two blocks are analogous: both "ask" something of some agents, the first asks the patches, the second the turtles. In addition, they do not ask all of the living agents but just a subset of them made up of a fixed number. How does NetLogo decide which agent to use to perform the action? This is done behind the scenes by NetLogo. When executing statements like `ask n-of` NetLogo creates a random sampling of the agentset. The sample is different every time the procedure is called so every simulation will start from a different situation.

Now that the world is created, each agent is in a state of happiness or not, depending on where it has been created in the world. Therefore, we must compute each turtle's happiness, assigning the right values to the three turtle variable, and then do some final stuff. These operations (computing and displaying happiness and updating everything) must be done at the end of the setup but will also be done at the end of each cycle of the simulation. To avoid replicating the same commands twice, we collect all these operations under a sub–procedure which we call `update`. At the moment, `update` hasn't been defined yet and pressing the `Check` button we get an appropriate error message stating that "UPDATE has not been defined". Therefore we define update writing a few more lines

```
1 to update
2   show-happiness
3   update-globals
4   do-plot
```

```
end
```

The procedure `update` calls in turn three other procedures, whose names are `do-plot`, `update-globals` and `show-happiness`. We will complete these procedures in the next paragraphs but to check if everything is ok up to now, just insert under the `update` procedure the following three "empty procedures":

```
to show-happiness
end

to update-globals
end

to do-plot
end
```

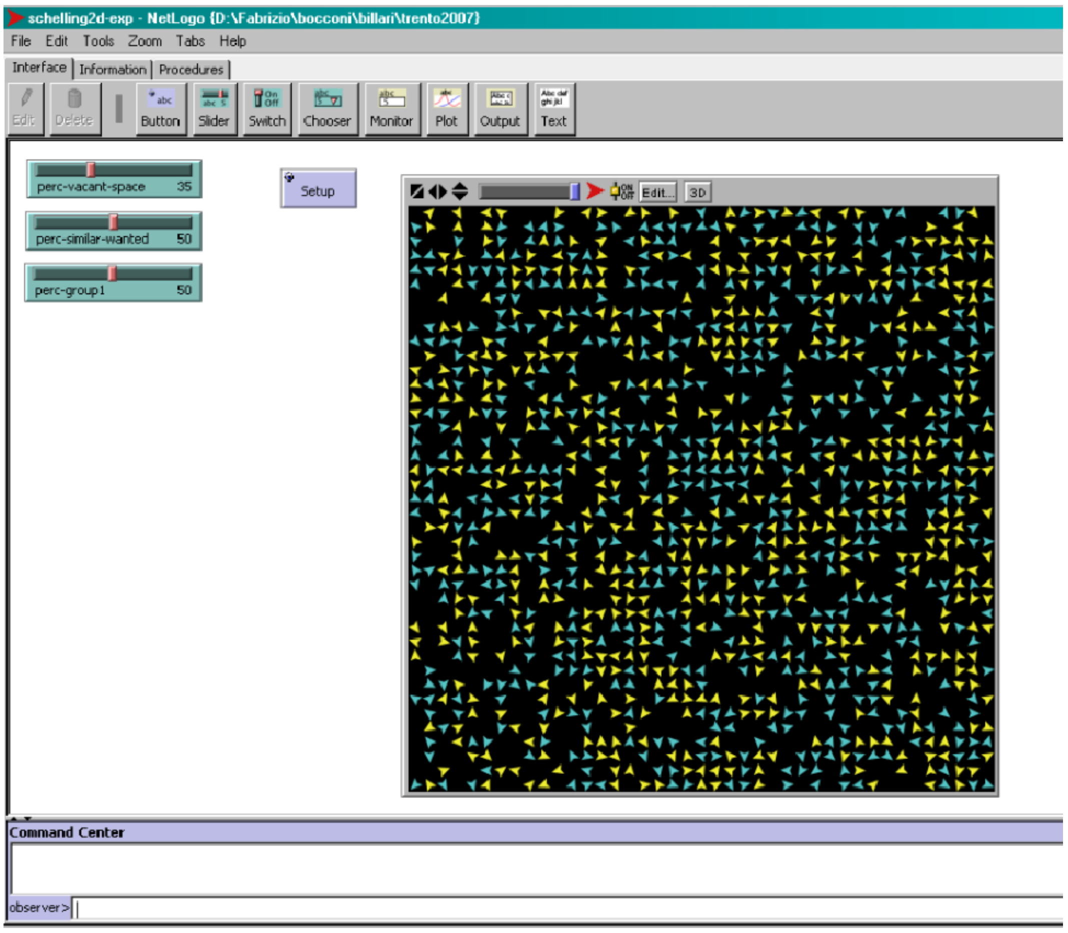
These procedures do nothing: they are just here for syntactical reasons. NetLogo has been instructed to execute these procedures and it must find them. The fact that the procedures do nothing is not important. This way of programming, i.e. creating apparently useless procedures to check for the syntax of other code, is very common and helps in prototyping a model from top to bottom: the modeler begins by creating empty procedures that, in the end, will do their tasks and then begins filling the procedures and periodically checks if everything is ok so far.

We can now check if what we wrote so far is working. Turn to the interface tab and press OK. Then press the Setup button. You should get a screen like the one in figure 2. Agents are represented by small triangles with random orientation. They are yellow and cyan according to the ratio specified by the corresponding slider. To complete the setup it remains only to assign to each agent its state of happiness or unhappiness.

3.6 The computation of happiness

To compute its happiness, each agent must perform three computations: the number of agents of its group in its neighbor, the number of agents of any group in its neighbor and the ratio of the former to the latter. If the one that performs the operation is the agent itself, the syntax of the operation is

```
to compute-happiness
  set total-nearby count turtles-on neighbors
  set similar-nearby count (turtles-on neighbors) with [color = [color]
      of myself]
  ifelse (total-nearby = 0) [
    set happy true
  ] [
    set happy (similar-nearby / total-nearby >= perc-similar-wanted /
        100)
  ]
  ifelse (happy = true) [
    set shape "face happy"
  ] [
    set shape "face sad"
  ]
end
```


schelling2d-exp - NetLogo {D:\Fabrizio\bocconi\billari\trento2007}
File Edit Tools Zoom Tabs Help
Interface Information Procedures
Edit Delete Button Slider Switch Chooser Monitor Plot Output Text
Setup
Edit... 3D
Command Center
observer>

perc-vacant-space 35
perc-similar-wanted 50
perc-group1 50

Fig. 2: A simple (and incomplete) setup of Schelling's model

This code sets the value of the turtle variables. Let's analyze it in detail. The first is a `set` statement, and its general syntax to assign a `value` to a `variable` is `set variable value`

```
set total-nearby count turtles-on neighbors
```

We assign the `total-nearby` variable of the turtle the value given by `count turtles-on neighbors`. The last expression uses the `count` reporter (think of it as a mathematical function that returns a value). The count acts on a set defined by the `turtles-on neighbors` expression whose meaning is "the turtles on the neighbors of the calling agent". In NetLogo a neighbor is the set of the eight patches surrounding a single patch (less than eight if the cell is on the border).

The second statement looks similar:

```
set similar-nearby count (turtles-on neighbors) with [color = color-of
    myself]
```

Here we assign the `similar-nearby` variable of the turtle the value given by another count expression but we count only the turtles living on neighbor patches with the same color of the calling agent. `color-of myself` is an expression that means "yellow" if the calling agent is yellow, and "cyan" if the calling agent is cyan.

At first sight, it seems that only one more statement is needed, namely that of line 49, where the happy variable is assigned the value of the comparison between the ratio computed in the neighbor of the patch and the intolerance. However, the world has many empty spaces and it could happen that an agent is alone in its neighbor. So the total neighbors would be 0 and the ratio would be impossible to compute. To take this situation into account we need a conditional statement, i.e. a statement that is executed only if a certain condition is true. Lines 46–50 show the solution and the syntax of the conditional statement. The condition must be enclosed within parentheses. If the condition is true, NetLogo executes the commands enclosed in the first couple of brackets; if the condition is false, those enclosed in the second couple are executed. As you see, we decided that a lone agent is always happy.

We use the keyword `ifelse` because we want NetLogo to do something either if the condition is met or if it is not. If you want NetLogo to do something only if the condition is true, use the keyword `if` followed by the condition in parentheses and the commands enclosed in brackets.

Lines 51–55 follow an analogous approach to visualize the state of an agent. NetLogo has a number of built-in shapes for representing turtles (and there is also a tool to create your own). Two predefined shapes are appropriate for this model, the “happy face” and the “sad face”. In lines 51-55, a choice for the shape of the turtle is made according to its state, which is kept in the variable `happy`, set in the previous lines.

The procedure just written works for a single turtle. To apply it to all the turtles in the world we modify the `show-happiness` procedure in this way

```
to show-happiness
  ask turtles [
    compute-happiness
  ]
end
```

Go back to the interface tab and click again on setup. You’ll find that each turtle now has a face symbol, happy or sad, depending on its state.

3.7 Final setup. The measure of segregation. Monitors

To complete the setup we must compute some quantities related to segregation and kept in the global variables defined at the beginning. The code for this computation is in the `update-globals` procedure which is called by setup (via the `update` function) and will be called at each step of the simulation, to show the dynamics of characteristic quantities of the model.

The syntax for the computation is the following

```
to update-globals
  set percent-unhappy (count turtles with [not happy] / count turtles)
      ; perc of unhappy turtles
  set percent-unhappy-1 (count turtles with [not happy and color =
     color1] / count turtles with [color = color1])  ; perc of unhappy
      turtles of color 1
  set percent-unhappy-2 (count turtles with [not happy and color =
     color2] / count turtles with [color = color2])   ; perc of
     unhappy turtles of color 2
  set segregation-1 mean [count (turtles-on neighbors) with [color =
      color1]] of turtles with [color = color1]
  set segregation-2 mean [count (turtles-on neighbors) with [color =
      color2]] of turtles with [color = color2]
end
```

The global parameter `percent-unhappy` is computed with a simple formula:

```
  set percent-unhappy (count turtles with [not happy] / count turtles)
        ; perc of unhappy turtles
```

i.e. the ratio of the number of unhappy turtles to the total number of turtles. Group parameters are computed in a similar way

```
  set percent-unhappy-1 (count turtles with [not happy and color =
      color1] / count turtles with [color = color1])  ; perc of unhappy
       turtles of color 1
  set percent-unhappy-2 (count turtles with [not happy and color =
      color2] / count turtles with [color = color2])   ; perc of
      unhappy turtles of color 2
```

adding the condition that the turtles should belong to one of the groups. Notice the presence of two "boolean" (logical) operators: `not` and `and`.

It remains only to compute a measure of the segregation of one group. In [4], this measure is referred to each group. For each agent, the number of similar agents in its neighbor is computed. Then we take the average over all the agents in the same group. This is taken as an average measure of the segregation of each group separately. To write this algorithm as a formula, let $i = 1, 2$ represents the two groups. Therefore:

$$\text{segregation}_i = \frac{\sum_{\text{turtles of group } i} \text{neighbor turtles of group } i}{\text{number of turtles of group } i}$$

The corresponding NetLogo code for both groups is

```
  set segregation-1 mean [count (turtles-on neighbors) with [color =
      color1]] of turtles with [color = color1]
  set segregation-2 mean [count (turtles-on neighbors) with [color =
      color2]] of turtles with [color = color2]
```

NetLogo can compute mean and standard deviation for numbers related to agentsets. It uses the syntax `mean values-from <agentset> [quantity]` which means "take the mean value of <quantity> over the specified <agentset>". In our case, the value is the number of turtles of the same color, i.e. `count (turtles-on neighbors) with [color = color1]`, while the agentset over which to compute the mean is `turtles with [color = color1]`. The same is done for group 2.

Now we have set two global variables to the segregation of the two populations. To make their values visible to us (and to the user) we add a `monitor` on the screen. A monitor is a simple window that displays the value of a specified expression (i.e. a variable or a quantity computed from variables). Go back to the interface tab, click on the monitor button and place a monitor on the screen. Releasing the mouse button opens a dialog in which you must insert the value of the quantity to be displayed, in our case `segregation-1`. Do the same for `segregation-2` and, if you wish, for percentage of happy people of both groups.

There is also a textbox to set the display name of the monitor where you can type, e.g., "Segregation of group 1", etc.

Pressing setup again will create another world and the monitor will display the computed segregation for each of the population. Segregation of the two groups can be different because in a single neighbor, the numbers of neighbors of the two colors are not related to one another, given the presence of empty spaces. If there were no empty spaces, the number of neighbors of the two colors will sum to 8 in a Moore neighbor, or 4 in a von Neumann neighbor. To check if everything is ok, one should compare the averages with their expected values. For example, if one sets the three main parameters (vacant space, intolerance and proportion of groups) to 50 %, in an 8 patch neighbor we expect (on the average) 4 of them to be empty and 4 of them to be occupied by turtles of both colors. The turtles, in turn, are expected in equal proportion, i.e. 2 of color 1 and 2 of color 2. In a border patch, there are 5 neighbors and on the average half of them (i.e. 2.5) are inhabited by turtles, 1.25 of them for each color. Finally, in a corner patch there are 3 neighbors, so 1.5 of them are inhabited, 0.75 for each color. Our world has $41 \times 41 = 1681$ patches, 1521 "internal" patches, 156 on the borders and 4 corner patches. Therefore, in our world the expected segregation for each group is

$$\frac{1521 \times 2 + 156 \times 1.25 + 4 \times 0.75}{1681} = 1.927424$$

Each setup should produce configurations whose segregation values should have 1.927424 as their mean. Whenever possible, it is best to verify the coherence of the model at setup time, as in this case. After a few steps in the simulation it would be much more difficult to distinguish setup errors from normal consequences of the dynamics of the model.

Finally, it is worth noting that the Schelling measure of segregation is not the only one generally accepted. A review of proposed measures of segregation together with a lot of data from the United States is available in [2].

4 Running the simulation

4.1 Rules of movement. Randomization

The rules defining the dynamics of the model are specified in the original article [4]. According to Schelling, the distance between two patches is the sum of the absolute values of the differences of the coordinates, as if the turtles moved from one patch to another along horizontal and vertical lines and *not* diagonal. Schelling then states that each agent who is not happy should move to the nearest position in which it would be happy. Let's call these positions "happy patches". Notice that what is a happy patch for one color is generally not a happy patch for the other color.

If we allow for enough vacant space and fix the ratio of the size of the two populations neither too high nor too low, there will always be a happy place for a turtle to move into. However we must take into account a much more serious problem. After each turtle moves, the state of the entire system changes and patches that were happy for the next turtle to move into can become not happy. This is a problem Schelling outlines in his article and it is obviously a potential source of bias in the dynamics of the model. What will happen if, e.g., all the turtles of group 1 will move before any turtle of group 2? Will their segregation dynamics change in some way?

For some problems, the perfect solution would be to evolve agents “in parallel” with each agent disregarding what others do. But for many problems this is not viable. For example, in Schelling’s model a parallel movement would possibly lead to two or more turtles ending on the same patch which is not an acceptable situation. A good compromise is to evolve agents through a sequential process (i.e. turtles moving in turn) randomizing the sequence. For example, in a world with 6 agents, the first round of movement could be $\{4, 2, 5, 6, 3, 1\}$, the second $\{2, 3, 6, 5, 1, 4\}$, and so on, each time generating a random permutation of the numbers 1,. . . ,6. Therefore, NetLogo designers implemented a built-in mechanism that does this randomization transparently for the user. Each time a command (or a set of commands) is issued to an agentset, the order in which agents execute the command is randomized.

To keep things simple, we deviate from Schelling’s prescription and define a new rule of movement: each unhappy agents moves to a random empty patch, disregarding the fact that in the new place it would be happy or not. This rule does not guarantee that a turtle will eventually find a good place to settle down but in fact this is the case, mainly because moving randomly in the world the chance that a turtle will never land on a happy place is very small. This obviously also depends on the value of the other parameters, especially on the percentage of vacant space: if this is small, there are fewer empty patches and consequently fewer happy patches to move to.

4.2 Implementing the dynamics

To run the simulation we create another button named `Go`. In the dialog box that pops up, type `go` in the commands textbox, and check the Forever checkbox. By doing this, NetLogo will repeat the called procedure continuously, which is what we want for the dynamics of the simulation. To stop the simulation when a certain condition is met, we will write a conditional statement in the go procedure, as the following listing shows.

```
to go
  if not any? turtles with [not happy] [stop]
  move-unhappy
  update
end

to move-unhappy
  ask turtles with [not happy] [ random-move ]
end

to random-move
  ; find an empty patches
  setxy random-pxcor random-pycor
  if any? other turtles-here [random-move]
end
```

In line 59 we check for the presence of unhappy turtles. The `any?` operator reports true if the given agentset is non-empty, false otherwise. In the latter case (no more unhappy turtles), the `stop` command ends the simulation. If there are still some unhappy turtles, the `move-unhappy` is called and then `update` takes care of updating the global variables (the `update` procedure was described in section 3.5).

The `move-unhappy` procedure just asks each unhappy turtle to execute the `random-move`. This is where the automatic randomization takes place. The `random-move` procedure firstly moves the agent onto a random position in the world; `random-pxcor` and `random-pycor` return the x and y coordinates of a random patch in the world. `setxy` acts on a turtle setting its coordinates to the specified values, in fact moving it to a new position (neglecting the chance that a turtle is moved to its current position, which has a very small probability). The problem is that landing on a random patch there is a chance that the patch is already inhabited by another turtle. To avoid this, in line 71 we check if there are other turtles in the same position (`other-turtles-here` because the turtle has already moved and "here" means "in the new position"). If this is the case, we call the `random-move` procedure again, iterating the process to find another place to move to. To get an idea, the process of moving is similar to that of a person who wants to move to a new flat, choosing an address on the map at random and then checking if the house is actually free or inhabited. If it is inhabited, a new random address is chosen in the hope that eventually an empty flat will be found.

It seems that this procedure is highly inefficient (nobody would search for a new flat that way!). However, as one can see running the simulation, this is not the case. In fact, the time spent in passing through inhabited patches until an empty one is found must be compared to that spent in the search for an empty patch. This requires a scan of the entire world and a random choice of one of the empty patches. To show that the results might be surprising, in the following sections we set up a similar model in which the user can control how unhappy turtles move.

The dynamics of the simulation is now complete. You can now go back to the interface tab and run the simulation a number of times. The complete code for this simulation is in appendix A.

5 Analyzing and extending the simulation

5.1 Change of parameters

A single simulation does not say too much to the researcher. To draw some conclusions, one has to run the simulation with the same starting conditions a number of times and then compute some statistics. In addition, to investigate the dependance of the results on any of the parameters, a typical experiment will consist of running the simulation many times varying the value of the parameter of interest. Doing this by hand can be extremely tedious and time consuming.

NetLogo comes with "BehaviorSpace", a basic tool to automatically run sets of simulations with varying parameters. BehavioSpace is available in the Tools menu. A collection of simulations is called an experiment. To create a new experiment, press `New` in the dialog window and you are presented with a complex window in which you design the experiment.

After giving the experiment a meaningful name (avoid names like "exp1"), you can insert the parameter you want to vary during the experiment. For our model, to vary the intolerance parameter from 30% to 70% with a step of 1, type

```
["perc-similar-wanted" [30 1 70]]
```

In the Repetitions textbox you can tell NetLogo how many simulation you want to run for a single value of intolerance. You can leave 1 in this textbox. Then you have to insert the expressions you want NetLogo to record for you. We are interested in the segregation measures so we type `segregation-1` and `segregation-2` in the next textbox. At the end of each simulation NetLogo will automatically record the values of these parameters. The next checkbox allows a more detailed output than the standard one, with the value of the segregation recorded at each tick.

Then, you have to tell BehaviorSpace what is the name of the procedure to call at setup time (usually `setup`), the procedure to call to run the simulation (usually `go`), the stop condition and a set of final commands you may want to execute. In our model, a stop condition is already coded in the go procedure but if you do not already have such a condition, you can set it here.

Finally, you can insert a time limit, i.e. a maximum number of ticks a single each simulation can run. This is very useful in situations in which an equilibrium is not reached and evolution loops. This actually happens if parameters take particular range of values. In the Schelling model, for example, choosing a low percentage of vacant space and a high intolerance obliges turtles to a long wandering through the world and the final "all happy" world is reached much later. In addition, there is no guarantee that a small set of agents enters a sort of loop in which they continuously move to the same unhappy position because the algorithm "does not see" the happy ones or because there are no more happy places.

There is no predefined rule to set these parameters and a number of hands-on experiments has to be carried out. After that, you can program BehaviorSpace to run the simulations for you. Each time an experiment is run, BehaviorSpace asks for the kind of output to produce. Outputs are text files that can be imported and further analyzed with, e.g., a spreadsheet application or a statistical software.

5.2 Change the dynamics. Adding a third group

In the model of the previous section we explicitly departed from Schelling's rules of movement for the sake of simplicity. It turns out that to follow Schelling's rules some more programming is required. Here are the steps to build a possible solution.

Turtles must move to the nearest happy patch. However, the meaning of "happy patch" depends on the turtle: what is happy for a turtle of group 1 is almost certainly not happy for a turtle of group 2 (except for patches with no neighbors and patches with an equal number of neighbors in the two groups in which case intolerance is exactly 50%). Therefore, when searching for the nearest happy patch we must take into account the color of the searching turtle. As the procedure for moving turtles is necessarily randomized (i.e. the moving procedure knows the group of the turtle only when it is called by the turtle, not in advance) there are two algorithms to solve the problem. The first algorithm looks at patches at distance 1 from the calling turtle, searches among them for a happy patch and moves the turtle to one of them or, iif none was found, looks at patches at distance 2 and searches again, each time increasing the radius of the search until a happy patch is found. The second algorithm, searches for happy patches all over the world and then moves the turtle to the nearest of them.

The algorithms could be compared on a theoretical basis but the main disadvantage of the first is that it is very difficult to follow its flow with the programming structures of NetLogo (in technical terms, unlike other programming environments, it is not straightforward to program the loop that searches for happy patches at increasing distances). The second one is more easily implemented in NetLogo, once a small trick is done. The trick consists of adding a couple of patch variables to the model, in this way:

```
patches-own [
  happy1
  happy2
]
```

Patches are now equipped with two variables that are true or false if the patch is a happy place for one of the groups. The `compute-happiness` procedure now includes a few more lines:

```
  ask patches [
    compute-potential-happiness
  ]
```

to compute also the happiness of patches according to the following procedure:

```
to compute-potential-happiness
  let nearby count turtles-on neighbors
  let nearby1 count turtles-from neighbors [turtles-here with [color =
      color1]]
  let nearby2 count turtles-from neighbors [turtles-here with [color =
      color2]]
  ifelse (nearby = 0) [
    set happy1 true
    set happy2 true
  ] [
    set happy1 (nearby1 / nearby >= perc-similar-wanted / 100)
    set happy2 (nearby2 / nearby >= perc-similar-wanted / 100)
  ]
end
```

At each tick, “global” happiness is computed, i.e. happiness of agents and happiness of patches. Then every unhappy agent moves to the nearest happy patch and the turn ends.

We now face a problem Schelling was aware of. After moving the first agent, the world has changed and what were once happy patches may now be unhappy, and viceversa. The problem lies in the sequential mechanism and Schelling’s suggestion is to choose movements taking into account the situation *before the first movement*: agents will move without considering what other agents did in the same tick (turn). This also makes things simpler in NetLogo because the configuration of happy patches must be updated only at the beginning of each tick of movements, and not before every single movement!

The movement procedure is the following:

```
to schelling-move
  if (color = color1) [
    let target min-one-of (patches with [happy1 and not any? turtles-
        here]) [abs ( pxcor - xcor-of myself ) + abs ( pycor - ycor-of
        myself )]
    if (target != nobody) [
      set xcor pxcor-of target
      set ycor pycor-of target
    ]
  ]
  if (color = color2) [
    let target min-one-of (patches with [happy2 and not any? turtles-
       here]) [abs ( pxcor - xcor-of myself ) + abs ( pycor - ycor-of
       myself )]
    if (target != nobody) [
      set xcor pxcor-of target
      set ycor pycor-of target
    ]
  ]
end
```

The complete solution to the problem is in appendix C

If the model is programmed in the right way, it should be easy to modify it to add a third group. The only problem is how to specify the relative percentages of the groups. One way to solve this problem is to have the user point two sliders to the percentages of groups 1 and 2 and then compute the size of group 3 by difference from the total. Using the same dynamics (that of random movement) an interesting fact happens when a third group is added: with a high level of intolerance, segregation takes much more time to appear, at least for the two smallest groups, than it takes for two groups. If the groups are all the same size, segregation does not appear at all, as the random movement continuously breaks the random clusters of similar that spontaneously grew during evolution. Therefore, the presence of a majority group seems to be the key factor in generating segregation, as Schelling proposed.

The extension to three groups is shown in the appendix B.

A The schelling2dv4.nlogo file

```
globals [
  percent-unhappy     ; perc of unhappy turtles
  percent-unhappy-1    ; perc of unhappy turtles of group 1
  percent-unhappy-2    ; perc of unhappy turtles of group 2
  segregation-1        ; measure of segr. for group 1
  segregation-2        ; measure of segr. for group 2
  color1                ; color of group 1
  color2                ; color of group 2
]

turtles-own [
  happy            ; true if turtle has at least percent-similar-wanted of similar in its neighbor
  similar-nearby   ; counts neighbors of the same color
  total-nearby     ; counts the n. of neighbors
]

to setup
  ca                ; clear all (not really useful explanation)
  set color1 45      ; yellow
  set color2 85      ; cyan
  let number-of-people int(world-width * world-height * (1 - perc-vacant-space / 100))
  ask n-of number-of-people patches [
    sprout 1 [set color color2]
  ]
  ask n-of int(number-of-people * perc-group1 / 100) turtles [
    set color color1
  ]
  update
end

to update
  show-happiness
  update-globals
  do-plot
end

to show-happiness
  ask turtles [
    compute-happiness
  ]
end

to compute-happiness
  set total-nearby count turtles-on neighbors
  set similar-nearby count (turtles-on neighbors) with [color = [color] of myself]
  ifelse (total-nearby = 0) [
    set happy true
  ] [
    set happy (similar-nearby / total-nearby >= perc-similar-wanted / 100)
  ]
  ifelse (happy = true) [
    set shape "face happy"
  ] [
    set shape "face sad"
  ]
end

to go
  if not any? turtles with [not happy] [stop]
  move-unhappy
  update
end

to move-unhappy
  ask turtles with [not happy] [ random-move ]
end

to random-move
  ; find an empty patches
  setxy random-pxcor random-pycor
  if any? other turtles-here [random-move]
end

to update-globals
  set percent-unhappy (count turtles with [not happy] / count turtles) ; perc of unhappy turtles
  set percent-unhappy-1 (count turtles with [not happy and color = color1] / count turtles with [color = color1])  ; perc of unhappy turtles of color 1
  set percent-unhappy-2 (count turtles with [not happy and color = color2] / count turtles with [color = color2])   ; perc of unhappy turtles of color 2
  set segregation-1 mean [count (turtles-on neighbors) with [color = color1]] of turtles with [color = color1]
  set segregation-2 mean [count (turtles-on neighbors) with [color = color2]] of turtles with [color = color2]
end

to do-plot
  set-current-plot "Unhappy"
  set-current-plot-pen "Total"
  plot percent-unhappy
  set-current-plot-pen "1"
  plot percent-unhappy-1
  set-current-plot-pen "2"
  plot percent-unhappy-2
  set-current-plot "Segregation"
  set-current-plot-pen "seg1"
  plot segregation-1
  set-current-plot-pen "seg2"
  plot segregation-2
```

```
end
```

B The schelling2d-3gv4.nlogo file

```
globals [
  percent-unhappy     ; perc of unhappy turtles
  percent-unhappy-1    ; perc of unhappy turtles of group 1
  percent-unhappy-2    ; perc of unhappy turtles of group 2
  percent-unhappy-3    ; perc of unhappy turtles of group 3
  segregation-1        ; measure of segr. for group 1
  segregation-2        ; measure of segr. for group 2
  segregation-3        ; measure of segr. for group 3
  color1                ; color of group 1
  color2                ; color of group 2
  color3                ; color of group 3
]

turtles-own [
  happy             ; true if turtle has at least percent-similar-wanted
      of similar in its neighbor
  similar-nearby    ; counts neighbors of the same color
  total-nearby      ; counts the n. of neighbors
]

to setup
  ca                  ; clear all (not really useful explanation)
  set color1 45       ; yellow
  set color2 85       ; cyan
  set color3 25       ; orange
  let number-of-people int(world-width * world-height * (1 - perc-
     vacant-space / 100))
  ask n-of number-of-people patches [
    sprout 1 [set color color1]
  ]
  ask n-of int(number-of-people * (1 - perc-group1 / 100)) turtles [
    set color color2
  ]
  ask n-of int((number-of-people * (1 - perc-group1 / 100)) * (1 - perc
     -group2 / 100)) turtles with [color = color2] [
    set color color3
  ]
  update
end

to update
  show-happiness
  update-globals
  do-plot
end

to show-happiness
  ask turtles [
    compute-happiness
  ]
end
```

```

to compute-happiness
  set total-nearby count turtles-on neighbors
  set similar-nearby count (turtles-on neighbors) with [color = color-of myself]
  ifelse (total-nearby = 0) [
    set happy true
  ] [
    set happy (similar-nearby / total-nearby >= perc-similar-wanted / 100)
  ]
  ifelse happy [
    set shape "face happy"
  ] [
    set shape "face sad"
  ]
end

to go
  if not any? turtles with [not happy] [stop]
  move-unhappy
  update
end

to move-unhappy
  ask turtles with [not happy] [ random-move ]
end

to random-move
  ; find an empty patches
  setxy random-pxcor random-pycor
  if any? other-turtles-here [random-move]
end

to update-globals
  set percent-unhappy (count turtles with [not happy] / count turtles)  ; perc of unhappy turtles
  set percent-unhappy-1 (count turtles with [not happy and color = color1] / count turtles with [color = color1])  ; perc of unhappy turtles of color 1
  set percent-unhappy-2 (count turtles with [not happy and color = color2] / count turtles with [color = color2])   ; perc of unhappy turtles of color 2
  set percent-unhappy-3 (count turtles with [not happy and color = color3] / count turtles with [color = color3])   ; perc of unhappy turtles of color 2
  set segregation-1 mean values-from turtles with [color = color1] [count (turtles-on neighbors) with [color = color1]]
  set segregation-2 mean values-from turtles with [color = color2] [count (turtles-on neighbors) with [color = color2]]
  set segregation-3 mean values-from turtles with [color = color3] [count (turtles-on neighbors) with [color = color3]]
end

```

```
to do-plot
  set-current-plot "Unhappy"
  set-current-plot-pen "Total"
  plot percent-unhappy
  set-current-plot-pen "1"
  plot percent-unhappy-1
  set-current-plot-pen "2"
  plot percent-unhappy-2
  set-current-plot-pen "3"
  plot percent-unhappy-3
  set-current-plot "Segregation"
  set-current-plot-pen "seg1"
  plot segregation-1
  set-current-plot-pen "seg2"
  plot segregation-2
  set-current-plot-pen "seg3"
  plot segregation-3
end
```

C The schelling2d-extv4.nlogo file

```
globals [
  percent-unhappy     ; perc of unhappy turtles
  percent-unhappy-1    ; perc of unhappy turtles of group 1
  percent-unhappy-2    ; perc of unhappy turtles of group 2
  segregation-1        ; measure of segr. for group 1
  segregation-2        ; measure of segr. for group 2
  color1                ; color of group 1
  color2                ; color of group 2
]

turtles-own [
  happy            ; true if turtle has at least percent-similar-wanted of similar in its neighbor
  similar-nearby   ; counts neighbors of the same color
  total-nearby     ; counts the n. of neighbors
]

patches-own [
  happy1
  happy2
]

to setup
  ca                ; clear all (not really useful explanation)
  set color1 45      ; yellow
  set color2 85      ; cyan
  let number-of-people int(world-width * world-height * (1 - perc-vacant-space / 100))
  ask n-of number-of-people patches [
    sprout 1 [set color color2]
  ]
  ask n-of int(number-of-people * perc-group1 / 100) turtles [
    set color color1
  ]
  update
end

to update
  show-happiness
  update-globals
  do-plot
end

to show-happiness
  ask turtles [
    compute-happiness
  ]
  ask patches [
    compute-potential-happiness
  ]
end
```

```

to compute-happiness
  set total-nearby count turtles-on neighbors
  set similar-nearby count (turtles-on neighbors) with [color = [color] of myself]
  ifelse (total-nearby = 0) [
    set happy true
  ] [
    set happy (similar-nearby / total-nearby >= perc-similar-wanted / 100)
  ]
  ifelse happy [
    set shape "face happy"
  ] [
    set shape "face sad"
  ]
end

to compute-potential-happiness
  let nearby count turtles-on neighbors
  ;let nearby1 count (turtle-set turtles-on neighbors) [turtles-here with [color = color1]])
  ;let nearby2 count (turtle-set turtles-on neighbors [turtles-here with [color = color2]])
  let nearby1 count (turtle-set (turtles-on neighbors) with [color = color1])
  let nearby2 count (turtle-set (turtles-on neighbors) with [color = color2])
  ifelse (nearby = 0) [
    set happy1 true
    set happy2 true
  ] [
    set happy1 (nearby1 / nearby >= perc-similar-wanted / 100)
    set happy2 (nearby2 / nearby >= perc-similar-wanted / 100)
  ]
end

to go
  if not any? turtles with [not happy] [stop]
  move-unhappy
  update
end

to move-unhappy
  if (move-type = "random1") [ask turtles with [not happy] [ random-move1 ]]
  if (move-type = "random2") [ask turtles with [not happy] [without-interruption [ random-move2 ]]]
  if (move-type = "schelling") [ask (turtles with [not happy]) [without-interruption [ schelling-move ]]]
end

to random-move1
  setxy random-pxcor random-pycor
  if any? other turtles-here [random-move1]
end

to random-move2
  ask one-of patches with [not any? turtles-here] [
    set [xcor] of myself pxcor
    set [ycor] of myself pycor
  ]
end

to schelling-move
  if (color = color1) [
    let target min-one-of (patches with [happy1 and not any? turtles-here]) [abs ( pxcor - [xcor] of myself ) + abs ( pycor - [ycor] of myself )]
    if (target != nobody) [
      set xcor [pxcor] of target
      set ycor [pycor] of target
    ]
  ]
  if (color = color2) [
    let target min-one-of (patches with [happy2 and not any? turtles-here]) [abs ( pxcor - [xcor] of myself ) + abs ( pycor - [ycor] of myself )]
    if (target != nobody) [
      set xcor [pxcor] of target
      set ycor [pycor] of target
    ]
  ]
end

to update-globals
  set percent-unhappy (count turtles with [not happy] / count turtles)     ; perc of unhappy turtles
  set percent-unhappy-1 (count turtles with [not happy and color = color1] / count turtles with [color = color1])  ; perc of unhappy turtles of color 1
  set percent-unhappy-2 (count turtles with [not happy and color = color2] / count turtles with [color = color2])   ; perc of unhappy turtles of color 2
  set segregation-1 mean [count (turtles-on neighbors) with [color = color1]] of turtles with [color = color1]
  set segregation-2 mean [count (turtles-on neighbors) with [color = color2]] of turtles with [color = color2]
end

to do-plot
  set-current-plot "Unhappy"
  set-current-plot-pen "Total"
  plot percent-unhappy
  set-current-plot-pen "1"
  plot percent-unhappy-1
  set-current-plot-pen "2"
  plot percent-unhappy-2
```

```
set-current-plot "Segregation"
set-current-plot-pen "seg1"
plot segregation-1
set-current-plot-pen "seg2"
```

References

[1] Swarm wiki. `http://www.swarm.org/`. retrieved Oct 27, 2008.

[2] U.S. Census Bureau, Housing and Household Economic Statistics Division. Racial and Ethnic Residential Segregation in the United States: 1980-2000. `http://www.census.gov/hhes/www/housing/housing_patterns/housing_patterns.html`. retrieved Oct 27, 2008.

[3] Wikipedia page for Logo. `http://en.wikipedia.org/wiki/Logo_%28programming_language%29`. retrieved Oct 27, 2008.

[4] Thomas C. Schelling. Dynamic models of segregation. *Journal of Mathematical Sociology*, 1:143–186, 1971.

[5] Thomas C. Schelling. *Micromotives and Macrobehavior*. W. W. Norton; Revised edition (October 16, 2006), USA, 2006.

[6] Uri Wilensky. Netlogo. `http://ccl.northwestern.edu/netlogo/`, 1999. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.