

# Agent-based models of segregation using Python

Patrick Vincent N. Lubenia

6 May 2020

## Contents

<b>Agent-Based Models</b>	<b>2</b>
<b>Python Basics: class</b>	<b>2</b>
class with Methods . . . . .	2
Empty class . . . . .	4
<b>Model 1</b>	<b>4</b>
Overview . . . . .	4
The Code . . . . .	4
Needed Libraries . . . . .	4
Environment: creating the class . . . . .	5
Agents: creating a method . . . . .	5
Metrics: creating a method . . . . .	6
Behavior: creating a method . . . . .	7
Quantifying Similarity: Similarity Ratio . . . . .	7
Visualization . . . . .	9
Implementation . . . . .	9
<b>Model 2</b>	<b>11</b>
Overview . . . . .	11
The Code . . . . .	11
Needed Libraries . . . . .	11
Modeling Task 1a: Attributes of the agents . . . . .	11
Modeling Task 2d: How agents interact with each other . . . . .	12
Quantifying Similarity: Similarity Ratio . . . . .	13
Visualization . . . . .	14
Implementation . . . . .	14
<b>Appendix A: Model 1 Full Code</b>	<b>17</b>
<b>Appendix B: Model 2 Full Code</b>	<b>22</b>
<b>References</b>	<b>25</b>

# Agent-Based Models

Agent-based modeling is a framework for creating and simulating models of complex systems. It is a mindset wherein a system is described from the point of view of the units constituting it [1]. Agent-based models are computational simulation models that involve numerous discrete agents. They show a system's emergent collective behavior resulting from the interactions of the agents. In contrast to equation-based models, each agent's behaviors in an agent-based model are described in an algorithmic fashion by rules rather than equations. Agents in the model do not typically perform actions together at constant time-steps [2]. Their decisions follow discrete-event cues or a series of interactions.

Depending on one's objectives, agents: are discrete entities, may have internal states, may be spatially localized, may perceive and interact with the environment, may behave based on predefined rules, may be able to learn and adapt, and may interact with other agents. Generally, agent-based models: often lack central supervisors/controllers and may produce nontrivial "collective behavior" as a whole.

The following scientific method-based approach must be kept in mind when designing an agent-based model:

1. Specific problem to be solved by the model
2. Availability of data
3. Method of model validation.

In order to be scientifically meaningful, an agent-based model must be:

1. Built using empirically-derived assumptions, then simulate to produce emergent behavior: for predictions; or
2. Built using hypothetical assumptions, then simulate to reproduce observed behavior: for explanations.

Once a code has been programmed, its basic implementation structure has 3 parts:

1. Initialization
2. Updating
3. Visualization.

Agents are initially placed in the model's environment. The system is then updated according to rules that govern the behavior of the environment and/or agents. Finally, states are visualized in order to appreciate the changes in the system.

The agent-based modeling framework is open-ended and flexible. It may be tempting to be detailed to make a model more realistic. But it must be remembered that increased complexity leads to increased difficulty in analysis. Moreover, the open-endedness of the framework makes it code-intense as lots of details of the simulation must be manually taken care of. Thus, codes must be kept simple and organized.

## Python Basics: class

### class with Methods

The main tool used in agent-based modeling in Python is a **class**. A **class** is created when objects with their own built-in functions (called *methods*) are desired. The following example creates a **rectangle** class with built-in functions for computing the rectangle's perimeter and area:

```
1 # Initialize the class
2 class rectangle:
3
4     # Initialize object when rectangle is called
5     # length, width: needed inputs
6     def __init__(self, length, width):
7
8         # Assign variable length to attribute called length
9         self.length = length
10
11        # Assign variable width to attribute called width
12        self.width = width
13
14        # Define method called perimeter
15        # No input needed: Uses variables in initialized object
```

```

16 def perimeter(self):
17
18     # Return perimeter of the rectangle
19     return 2*self.length + 2*self.width
20
21 # Define method called area
22 # No input needed: Uses variables in initialized object
23 def area(self):
24
25     # Return area of the rectangle
26     return self.length*self.width

```

`self` is a placeholder for the variable name given to the rectangle. To create a rectangle `r` with length 3 and width 4, type

```

1 r = rectangle(3, 4)

```

To retrieve the length and width of rectangle `r`, type

```

1 r.length
2 r.width

```

To get the perimeter and area of rectangle `r`, type

```

1 r.perimeter()
2 r.area()

```

Note that additional variables, which are not taken from the inputs to the class object, can be defined. For example, a variable called `diagonal`, which does not initially have a value:

```

1 class rectangle:
2     def __init__(self, length, width):
3         self.length = length
4         self.width = width
5
6         # Not in the list of needed inputs
7         # Initialized for methods that use the variable diagonal
8         self.diagonal = []

```

After creating rectangle `r` as above, typing

```

1 r.diagonal

```

returns an empty list. The following adds a function that computes the diagonal of the rectangle and places that value inside the variable called `diagonal`. The full code is as follows:

```

1 # Needed for computing square root
2 import numpy as np
3
4 class rectangle:
5     def __init__(self, length, width):
6         self.length = length
7         self.width = width
8         self.diagonal = []
9     def perimeter(self):
10         return 2*self.length + 2*self.width
11     def area(self):
12         return self.length*self.width
13     def diag(self):
14         self.diagonal = np.sqrt(self.length**2 + self.width**2)

```

The following computes the value of the diagonal places it in `r.diagonal`:

```

1 r.diag()

```

There is no output, however, because the defined function does not specify a `return` value. This use of `class` is utilized in the first model.

## Empty class

Sometimes, a `class` is created when objects are needed to be flexible enough to be given desired attributes. In

```
1 class performer:
2     pass
```

the `pass` on line 2 allows one to create an empty class. This class can now be used to create a performer:

```
1 performer_ = performer()
```

It is simple to add attributes to the object called performer, say its location in terms of coordinates, name, and age:

```
1 # Assign attribute x to performer_
2 performer_.x = 3
3
4 # Assign attribute y to performer_
5 performer_.y = 4
6
7 # Assign attribute name to performer_
8 performer_.name = 'Luca'
9
10 # Assign attribute age to performer_
11 performer_.age = 16
```

This use of `class` is utilized in the second model.

## Model 1

### Overview

The following model is based on a tutorial by Moujahid [3]. It follows an agent-based modeling framework with the following structure:

1. Environment: Where the individual components of the system move around
2. Agents: Units interacting with each other and making decisions
3. Metrics: Rules that the agents follow
4. Behavior: How agents interact.

The case study to be used is the Schelling segregation model which was named after Thomas Schelling, an American economist and a Nobel laureate in economics. His segregation model studies the emergent segregation that occurs in a community with 2 groups of people [5]. The question that the model wishes to answer is: *How high should the residents' threshold be in order for segregation to occur?*

To start the modeling process, the elements of the framework are defined as follows:

1. Environment: A community with houses
2. Agents: Residents of the community
3. Metrics: A resident is happy if a certain number of his neighbors have the same group as his
4. Behavior: A resident stays put if he is happy; otherwise, he moves to a different location

## The Code

### Needed Libraries

To start the code, the following libraries are imported:

```
1 # For pairing x- and y-values to create Cartesian products; an ordered pair represents the address a
   resident
2 import itertools
3
4 # For shuffling all houses in the community to randomly assign them to residents; and for choosing a
   random empty house to move into (for unhappy residents)
5 import random as rd
6
```

```

7 # For visualizing the distribution of the residents using scatterplot
8 import matplotlib.pyplot as plt
9
10 # For checking if a filename already exists, to avoid overwriting files
11 import os

```

## Environment: creating the class

The community is represented by a grid, and each grid cell represents a house which can be occupied by at most one resident. The following need to be defined:

- Dimensions of the community
- Percentage of houses that are empty (so residents have places to move into if they are unhappy)
- The happiness threshold that determines if a resident moves or not
- Number of groups in the community.

Define a class called Schelling:

```

1 class Schelling:
2     def __init__(self, width, height, empty_ratio, happiness_threshold, groups, n_iterations):
3
4         # Width of community grid
5         self.width = width
6
7         # Height of community grid
8         self.height = height
9
10        # Percentage of empty houses
11        self.empty_ratio = empty_ratio
12
13        # Minimum similarity ratio, to be considered happy
14        self.happiness_threshold = happiness_threshold
15
16        # Number of groups of residents
17        self.groups = groups
18
19        # Maximum iterations
20        self.n_iterations = n_iterations
21
22        # List of empty houses
23        self.empty_houses = []
24
25        # Dictionary of residents
26        self.agents = {}

```

Calling Schelling needs 6 inputs. There are 2 extra variables that are used later: empty\_houses and agents. In the dictionary of residents, each resident is represented by a tuple whose first element is an ordered pair (its address) and second element is a number (its group number).

## Agents: creating a method

A method called populate is added. It randomly scatters the residents throughout the community:

```

1 def populate(self):
2
3     # Create address (coordinates) of residents
4     all_houses = list(itertools.product(range(self.width), range(self.height)))
5
6     # Shuffle the order of houses
7     rd.shuffle(all_houses)
8
9     # Determine number of empty houses
10    n_empty = int(self.empty_ratio*len(all_houses))
11
12    # Assign first few houses as empty
13    self.empty_houses = all_houses[:n_empty]
14
15    # Assign the rest to be occupied
16    remaining_houses = all_houses[n_empty:]

```

```

17 # Assign houses by group to residents
18 houses_by_group = [remaining_houses[i::self.groups] for i in range(self.groups)]
19
20 # Create dictionary of residents
21 # Each resident is defined by his address and group number
22 for i in range(self.groups):
23     agent = dict(zip(houses_by_group[i], [i+1]*len(houses_by_group[i])))
24     self.agents.update(agent)
25

```

## Metrics: creating a method

In the `is_unhappy` method, for each resident, it checks each neighbor to see if it is of the same group as the resident, computes the happiness of the resident, then sees if he is happy. This method is of a negative nature (“is UNhappy” instead of “is happy”) so that if it is true, i.e., the resident is unhappy, then the the resident transfers to another house. The method is used inside the `move` method.

```

1 def is_unhappy(self, x, y):
2
3     # Get group number of resident
4     group = self.agents[(x, y)]
5
6     # Initialize variables
7     count_similar = 0
8     count_different = 0
9
10    # Check similarity with bottom left neighbor
11    if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
12        if self.agents[(x-1, y-1)] == group:
13            count_similar += 1
14        else:
15            count_different += 1
16
17    # Check similarity with bottom neighbor
18    if y > 0 and (x, y-1) not in self.empty_houses:
19        if self.agents[(x, y-1)] == group:
20            count_similar += 1
21        else:
22            count_different += 1
23
24    # Check similarity with bottom right neighbor
25    if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
26        if self.agents[(x+1, y-1)] == group:
27            count_similar += 1
28        else:
29            count_different += 1
30
31    # Check similarity with left neighbor
32    if x > 0 and (x-1, y) not in self.empty_houses:
33        if self.agents[(x-1,y)] == group:
34            count_similar += 1
35        else:
36            count_different += 1
37
38    # Check similarity with right neighbor
39    if x < (self.width-1) and (x+1, y) not in self.empty_houses:
40        if self.agents[(x+1,y)] == group:
41            count_similar += 1
42        else:
43            count_different += 1
44
45    # Check similarity with upper left neighbor
46    if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
47        if self.agents[(x-1,y+1)] == group:
48            count_similar += 1
49        else:
50            count_different += 1
51
52    # Check similarity with upper neighbor
53    if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
54        if self.agents[(x,y+1)] == group:
55            count_similar += 1
56        else:
57            count_different += 1
58
59    # Check similarity with upper right neighbor

```

```

60     if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
61         if self.agents[(x+1,y+1)] == group:
62             count_similar += 1
63         else:
64             count_different += 1
65
66     # Resident is NOT unhappy, i.e., happy if he has no neighbors
67     if (count_similar + count_different) == 0:
68         return False
69
70     # Check if similarity ratio is below happiness threshold
71     else:
72         return float(count_similar/(count_similar + count_different)) < self.happiness_threshold

```

In each similarity check, if a neighbor is of the same group as the resident, then a count is added to count\_similar. Otherwise, a count is added to count\_different. If the similarity ratio is less than the happiness threshold, then `is_unhappy` returns True, i.e., the resident is indeed unhappy. If the similarity ratio is NOT less than (i.e., more than) the happiness threshold, then `is_unhappy` returns False, i.e., the resident is actually happy. Note that The method is only triggered when a resident is unhappy. Once triggered, the resident is moved to action.

### Behavior: creating a method

`move` allows residents to transfer to another house if they are unhappy. it uses the `is_unhappy` method.

```

1  def move(self):
2
3      # Maximum iterations allowed
4      for i in range(self.n_iterations):
5
6          # Initialize count
7          n_changes = 0
8
9          # Check each resident
10         for agent in self.agents:
11
12             # Activated if resident is unhappy
13             if self.is_unhappy(agent[0], agent[1]):
14
15                 # Get a random empty house
16                 empty_house = rd.choice(self.empty_houses)
17
18                 # Get group number of the resident
19                 agent_group = self.agents[agent]
20
21                 # Assign the empty house to the resident
22                 self.agents[empty_house] = agent_group
23
24                 # Remove the original residence from the dictionary
25                 del self.agents[agent]
26
27                 # Remove the now-occupied house from the list of empty houses
28                 self.empty_houses.remove(empty_house)
29
30                 # Add the house the resident just left to the list of empty houses
31                 self.empty_houses.append(agent)
32
33                 # Count as a move
34                 n_changes += 1
35
36         # Iteration stops if everyone is already happy
37         if n_changes == 0:
38             break

```

### Quantifying Similarity: Similarity Ratio

The model is already complete at this point. But to go the extra mile, the overall similarity ratio of the entire community can be computed using the `similarity` method. The similarity ratio in each house is computed (as in the `is_unhappy` method) and the average over all households is taken.

```

1  def similarity(self):
2
3      # Initialize list
4      similarity = []

```

```

5
6 # Do for each resident
7 for agent in self.agents:
8
9     # Initialize variables
10    count_similar = 0
11    count_different = 0
12
13    # Get address and group number of the resident
14    x = agent[0]
15    y = agent[1]
16    group = self.agents[(x,y)]
17
18    # Check similarity with bottom left neighbor
19    if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
20        if self.agents[(x-1, y-1)] == group:
21            count_similar += 1
22        else:
23            count_different += 1
24
25    # Check similarity with bottom neighbor
26    if y > 0 and (x, y-1) not in self.empty_houses:
27        if self.agents[(x, y-1)] == group:
28            count_similar += 1
29        else:
30            count_different += 1
31
32    # Check similarity with bottom right neighbor
33    if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
34        if self.agents[(x+1, y-1)] == group:
35            count_similar += 1
36        else:
37            count_different += 1
38
39    # Check similarity with left neighbor
40    if x > 0 and (x-1, y) not in self.empty_houses:
41        if self.agents[(x-1, y)] == group:
42            count_similar += 1
43        else:
44            count_different += 1
45
46    # Check similarity with right neighbor
47    if x < (self.width-1) and (x+1, y) not in self.empty_houses:
48        if self.agents[(x+1, y)] == group:
49            count_similar += 1
50        else:
51            count_different += 1
52
53    # Check similarity with upper left neighbor
54    if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
55        if self.agents[(x-1, y+1)] == group:
56            count_similar += 1
57        else:
58            count_different += 1
59
60    # Check similarity with upper neighbor
61    if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
62        if self.agents[(x, y+1)] == group:
63            count_similar += 1
64        else:
65            count_different += 1
66
67    # Check similarity with upper right neighbor
68    if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
69        if self.agents[(x+1,y+1)] == group:
70            count_similar += 1
71        else:
72            count_different += 1
73
74    # Place similarity ratio in the list
75    try:
76        similarity.append(float(count_similar/(count_similar + count_different)))
77
78    # If there are no neighbors, similarity ratio is 1
79    except:
80        similarity.append(1)
81

```



```

82 # Compute average similarity ratio over all residents
83 return sum(similarity)/len(similarity)

```

Error in computation of similarity ratio only occurs when we 0/0 happens. In this case, a similarity ratio of 1 is used.

## Visualization

A scatterplot is produced to visualize what happens to the residents. The method `visualize` does exactly this:

```

1  def visualize(self, state):
2
3      # Initialize subplots
4      fig, ax = plt.subplots()
5
6      # Assign color to each group (define more as needed)
7      agent_colors = {1:'b', 2:'r', 3:'g', 4:'c', 5:'m', 6:'y', 7:'k'}
8
9      # Create a scatterplot for each resident, colored based on his group
10     for agent in self.agents:
11         ax.scatter(agent[0]+0.5, agent[1]+0.5, color = agent_colors[self.agents[agent]], edgecolors = '
            white')
12
13     # Title
14     ax.set_title('Schelling Model: ' + state + ' State' + '\n' + 'Similarity: ' + str("{:.1f}".format(
            schelling.similarity()*100)) + '%')
15
16     # Horizontal axis label
17     ax.set_xlabel(str(self.empty_ratio*100) + '% Empty Houses' + '\n' + str(self.happiness_threshold*1
            00) + '% Happiness Threshold' + '\n' + str(self.groups) + ' groups')
18
19     # Ensure all residents are visible
20     ax.set_xlim([0, self.width])
21     ax.set_ylim([0, self.height])
22
23     # Remove extra tick marks on the axes
24     ax.set_xticks([])
25     ax.set_yticks([])
26
27     # Prepare format of file name
28     filename = 'Model1_' + state
29
30     # Starting filename count
31     i = 1
32
33     # Check if filename already exists; add 1 if it does
34     while os.path.exists('{f}{:d}.png'.format(filename, i)):
35         i += 1
36
37     # Save figure
38     plt.savefig('{f}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)

```

The method takes in a string which indicates the state of the system (ideally either “Initial” or “Final”). Subplots are used instead of the simpler `plt.scatter` so that when the file is run, successive runs of plots do not overlap into one figure. Up to 7 groups are assigned colors. New colors must be assigned if more than 7 groups are created. The additional 0.5 in the coordinates ensures that the plots on the edges are not squished to the sides, and the white edgecolor ensures that when plots are too close to each other, they can still be identified from each other. The tight specification in `savefig` removes extra white spaces around the figure.

## Implementation

The simulation follows the basic code implementation structure mentioned in the introduction: agents are created (Initialization), then they are allowed to move according to their defined behavior (Updating). Plots are used to visualize (Visualization) what happens to the system before and after the agents move. The model simulates a community 25×25 big with 25% empty houses populated by two groups. Residents are already happy if 30% of their neighbors are from the same group as they do. Residents are allowed to move up to 500 times if they are unhappy.

```

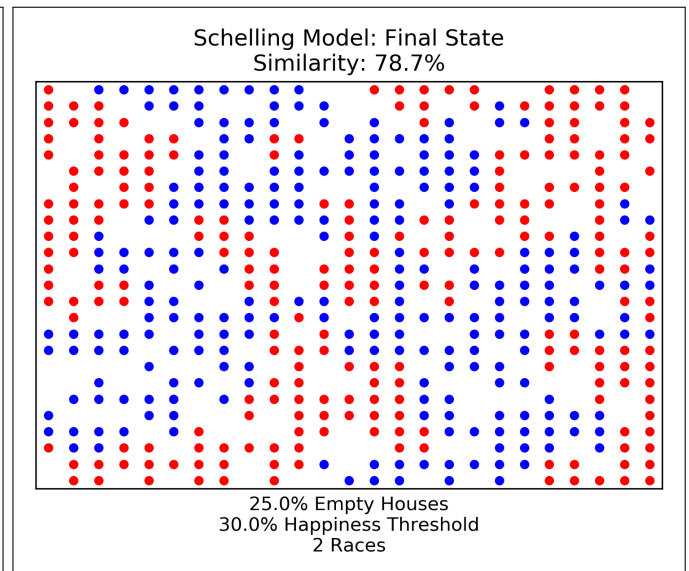
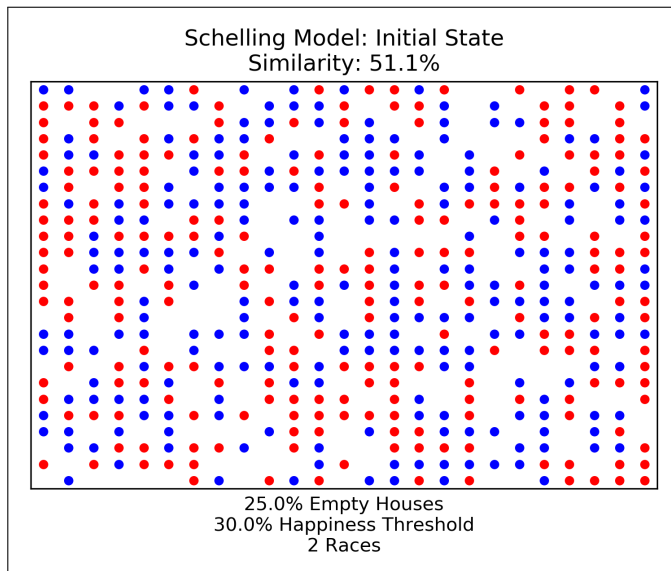
1 # Initialize parameters
2 schelling = Schelling(width = 25, height = 25, empty_ratio = 0.25, happiness_threshold = 0.30, groups
    = 2, n_iterations = 500)
3
4 # Place residents on the community
5 schelling.populate()

```

```

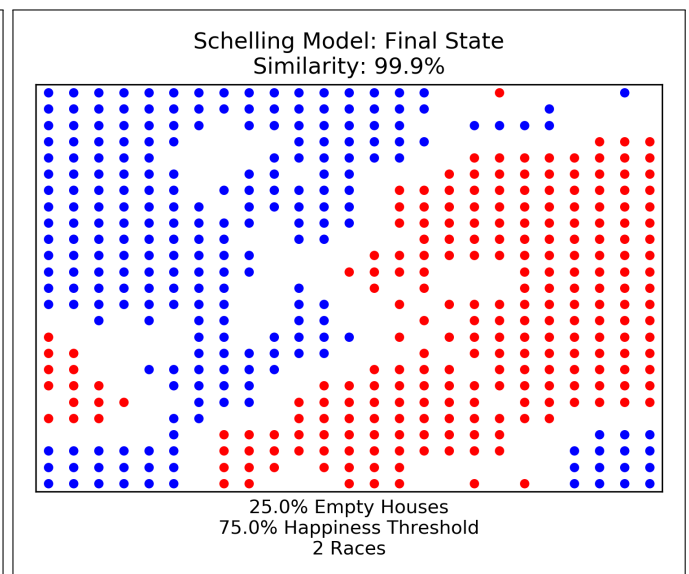
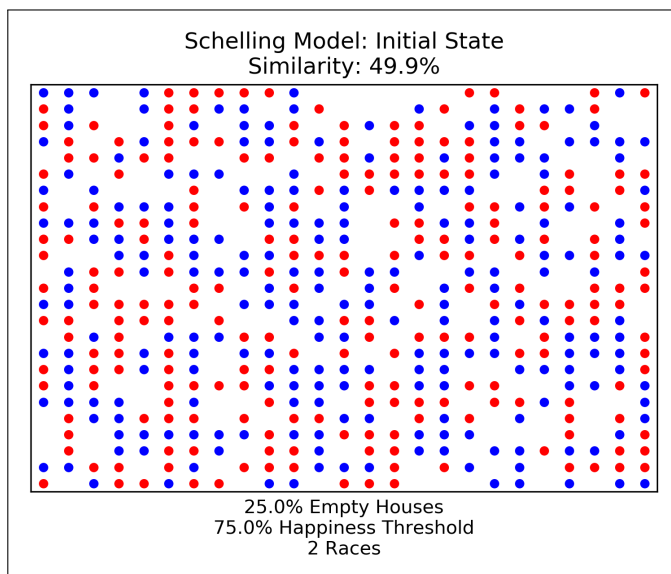
6
7 # Visualize initial state
8 schelling.visualize('Initial')
9
10 # Allow unhappy residents to move
11 schelling.move()
12
13 # Visualize final state
14 schelling.visualize('Final')

```



Initial overall similarity ratio among neighbors was 51.1%. After allowing unhappy residents to move, the overall similarity ratio increased to 78.7%, much higher than the happiness threshold of 30%. It can be seen visually that some form of segregation resulted in the final stage.

Using a higher happiness threshold of 75%:



Overall similarity ratio was initially 49.9%. The high happiness threshold resulted in an almost completely segregated community with overall similarity ratio of 99.9%. The diagram shows this clearly.

Randomly assigning residents to 2 groups distributes the population evenly, leading to an initial overall similarity ratio around 50%. However, even with a low happiness threshold of 30%, segregation naturally occurs, with overall similarity ratio much higher than what residents are willing to tolerate. Thus, on a macro level, observing segregation may not be indicative of what people feel at the micro level. Hence, the answer to the question *How high should the agents' threshold be in order for segregation to occur?* is: **not so high**.

# Model 2

## Overview

The following model is shorter than the previous one, and is based on the model by Sayama [4]. It follows an agent-based modeling framework with the following tasks that must be undertaken:

1. Design the data structure to store the:
  - (a) Attributes of the agents
  - (b) States of the environment
2. Describe the rules for how:
  - (a) The environment behaves on its own
  - (b) Agents interact with the environment
  - (c) Agents behave on their own
  - (d) Agents interact with each other.

Not all these tasks are needed in every agent-based model.

The succeeding model is slightly different from the previous one. In this model, there are still 2 groups of residents, and each resident observes his neighborhood. If he is not satisfied with the residents surrounding him, he moves to a different location. The differences:

- The environment is not a defined grid: the number of residents is defined, and they are located randomly
- The number of neighbors is not limited to 8: a neighborhood is defined using a radius.

The question still remains: *How high should the residents' threshold be in order for segregation to occur?*

## The Code

### Needed Libraries

To start the code, the following libraries are imported:

```
1 # For assigning a random location to agents
2 import random as rd
3
4 # For visualizing the distribution of agents using plot
5 import matplotlib.pyplot as plt
6
7 # For checking if a filename already exists, to avoid overwriting files
8 import os
```

### Modeling Task 1a: Attributes of the agents

The first task is to “design the data structure to store the attributes of the agents”. Each resident has the following attributes:

1. Spatial location: random coordinates
2. Group number: 0 or 1.

Note that this modeling task incorporates both the Environment and Agents parameters in the previous model.

The class called `agent` is initialized:

```
1 class agent:
2     pass
```

A function called `create_agents` is defined to create the residents of the community:

```

1 def create_agents():
2
3     # Allow list of residents to be accessed outside the function
4     global agents_list
5
6     # Initialize list
7     agents_list = []
8
9     # Create specified number of residents
10    for each_agent in range(n_agents):
11
12        # Create an agent
13        agent_ = agent()
14
15        # Assign to a random address
16        agent_.x = rd.random()
17        agent_.y = rd.random()
18
19        # Assign to a random group
20        agent_.group = rd.randint(0, groups-1)
21
22        # Place resident on the list
23        agents_list.append(agent_)

```

Using `global` automatically creates the variable outside the function. Note that this function needs 2 variables to be predefined: `n_agents` (total number of residents) and `groups` (total number of groups).

The following function groups the residents by their number:

```

1 def group_by_number():
2
3     # Allow list of groups to be accessed outside the function
4     global group
5
6     # Initialize list
7     group = []
8
9     # Group according to group number
10    for group_number in range(groups):
11
12        # A resident is grouped with other residents with the same group number
13        group.append([agent_ for agent_ in agents_list if agent_.group == group_number])

```

Note that once grouped, the residents remain in their respective groups. If they decide to move, they are still part of the same group, but their coordinates change.

### Additional Notes:

1. There are no separate environments that interact with the residents so Modeling Tasks 1b (“design the data structure to store the states of the environment”), 2a (“describe the rules for how the environment behaves on its own”), and 2b (“describe the rules for how agents interact with the environment”) are skipped.
2. Residents do not do anything by themselves so Modeling Task 2c (“describe the rules for how agents behave on their own”) is also skipped.

### Modeling Task 2d: How agents interact with each other

The other task that needs to be done is to “describe the rules for how agents interact with each other”. A resident checks everyone within its neighborhood (defined by a radius). If he is satisfied with the number of people belonging to the same group as he does, he stays put. Otherwise, he moves to a different location. Note that this modeling task incorporates both the Metrics and Behavior parameters in the previous model. A function named `move` implements this:

```

1 def move():
2
3     # Allow number of iterations done to be accessed outside the function
4     global iteration
5
6     # Maximum iterations allowed
7     for iteration in range(n_iterations):
8
9         # Initialize count

```

```

10     n_changes = 0
11
12     # Check each resident
13     for agent_ in agents_list:
14
15         # Create list of neighbors within the radius specified
16         neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y -
neighbor.y)**2 < radius**2]
17
18         # Remove resident himself from list of neighbors
19         neighbors.remove(agent_)
20
21         # Check if there are neighbors within the radius
22         if len(neighbors) > 0:
23
24             # Compute similarity ratio
25             satisfaction = len([neighbor for neighbor in neighbors if neighbor.group == agent_.group])/len
(neighbors)
26
27             # Move resident to a random location if similarity ratio is below threshold
28             if satisfaction < threshold:
29                 agent_.x, agent_.y = rd.random(), rd.random()
30
31             # Count as a move
32             n_changes += 1
33
34     # Iteration stops if everyone is already happy
35     if n_changes == 0:
36         break

```

A neighbor is someone who falls within the defined radius, with the resident as the center. This function needs 3 additional variables to be predefined: `n_interations` (maximum number of iterations allowed), `radius` (how small the neighborhood of the resident that needs to be checked), and `threshold` (minimum similarity ratio in order to be considered satisfied). Note also that the change in coordinates does not affect the grouping done by the function `group_by_number` so there is no need to rerun the grouping function.

### Quantifying Similarity: Similarity Ratio

The model is already complete at this point. But to go the extra mile, the overall similarity ratio of the entire community can be computed using the `similarity_ratio` function. The similarity ratio of each person is computed (as in the `move` function) and the average over all residents is taken.

```

1 def similarity_ratio():
2
3     # Allow overall similarity ratio to be accessed outside the function
4     global overall_similarity_ratio
5
6     # Initialize list
7     similarity_ratios = []
8
9     # Check each resident
10    for agent_ in agents_list:
11
12        # Create list of neighbors within the radius specified
13        neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y -
neighbor.y)**2 < radius**2]
14
15        # Remove resident himself from list of neighbors
16        neighbors.remove(agent_)
17
18        # Check if there are neighbors within the radius
19        if len(neighbors) > 0:
20
21            # Place similarity ratio in the list
22            try:
23                similarity_ratios.append(len([neighbor for neighbor in neighbors if neighbor.type == agent_.
type])/len(neighbors))
24
25            # If there are no neighbors, similarity ratio is 1
26            except:
27                similarity_ratios.append(1)
28
29    # Compute average similarity ratio over all residents
30    overall_similarity_ratio = sum(similarity_ratios)/len(similarity_ratios)

```

```

31
32 # Outputs overall similarity ratio
33 return overall_similarity_ratio

```

Error in computation of similarity ratio occurs only when we 0/0 happens. In this case, a similarity ratio of 1 is used.

## Visualization

To visualize the states of the system, the function `visualize` is created:

```

1 def visualize(state):
2
3     # Initialize subplots
4     fig, ax = plt.subplots()
5
6     # Create a scatterplot by group, plotting each resident
7     for Group in range(groups):
8         ax.plot([agent_.x for agent_ in group[Group]], [agent_.y for agent_ in group[Group]], 'o')
9
10    # Title
11    ax.set_title(state + ' State' + ' || ' + 'Segregation: ' + str("{:.1f}".format(similarity_ratio()*100)) + '%')
12
13    # Horizontal axis label
14    ax.set_xlabel(str(n_agents) + ' Residents' + ' || ' + str(groups) + ' Groups' + ' || ' + str(radius*100) + '% Neighborhood || ' + str(threshold*100) + '% Threshold' + '\n' + 'moves: ' + str(iteration))
15
16    # Remove extra tick marks on the axes
17    ax.set_xticks([])
18    ax.set_yticks([])
19
20    # Prepare format of file name
21    filename = 'Model2_' + state
22
23    # Starting filename count
24    i = 1
25
26    # Check if filename already exists; add 1 if it does
27    while os.path.exists('{:d}.png'.format(filename, i)):
28        i += 1
29
30    # Save figure
31    plt.savefig('{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)

```

The function takes in a string which indicates the state of the system (ideally either “Initial” or “Final”). Subplots are used instead of the simpler `plt.scatter` so that when the file is run, successive runs of plots do not overlap into one figure. The tight specification in `savefig` removes extra white spaces around the figure.

## Implementation

The simulation follows the basic code implementation structure mentioned in the introduction: agents are created (Initialization), then they are allowed to move according to their defined behavior (Updating). Plots are used to visualize (Visualization) what happens to the system before and after the agents move. To compare with the first model, the model simulates a community with 1,000 residents belonging to 2 groups. Each resident cares only about a neighborhood radius of 10%, and are already happy if 31% of their neighbors are from the same group as they do. Residents are allowed to move up to 500 times if they are unhappy.

```

1 # Number of residents
2 n_agents = 1000
3
4 # Number of groups
5 groups = 2
6
7 # Maximum iterations
8 n_iterations = 100
9
10 # Neighborhood radius
11 radius = 0.1
12
13 # Satisfaction threshold
14 threshold = 0.31

```

```

15
16 # Create residents
17 create_agents()
18
19 # Group residents according to group number
20 group_by_number()
21
22 # Needed for label of initial state
23 iteration = 0
24
25 # Visualize initial state
26 visualize('Initial')
27
28 # Allow unsatisfied residents to move
29 move()
30
31 # Visualize initial state
32 visualize('Final')

```

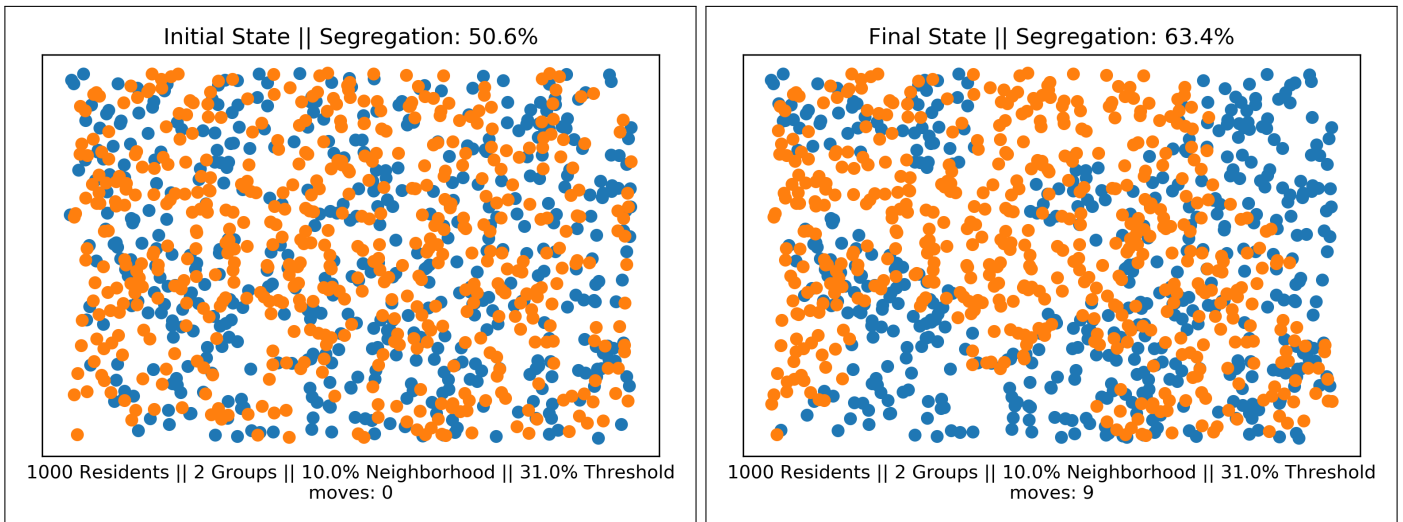


Figure 1: Simulation of Group Segregation. A high segregation level much higher than the happiness threshold naturally occurs.

Figure 1 shows that initial overall similarity ratio among neighbors was 50.6%. After allowing unhappy residents to move, the overall similarity ratio increased to 63.4%, much higher than the happiness threshold of 31%. It can be seen visually that some form of segregation resulted in the final stage.

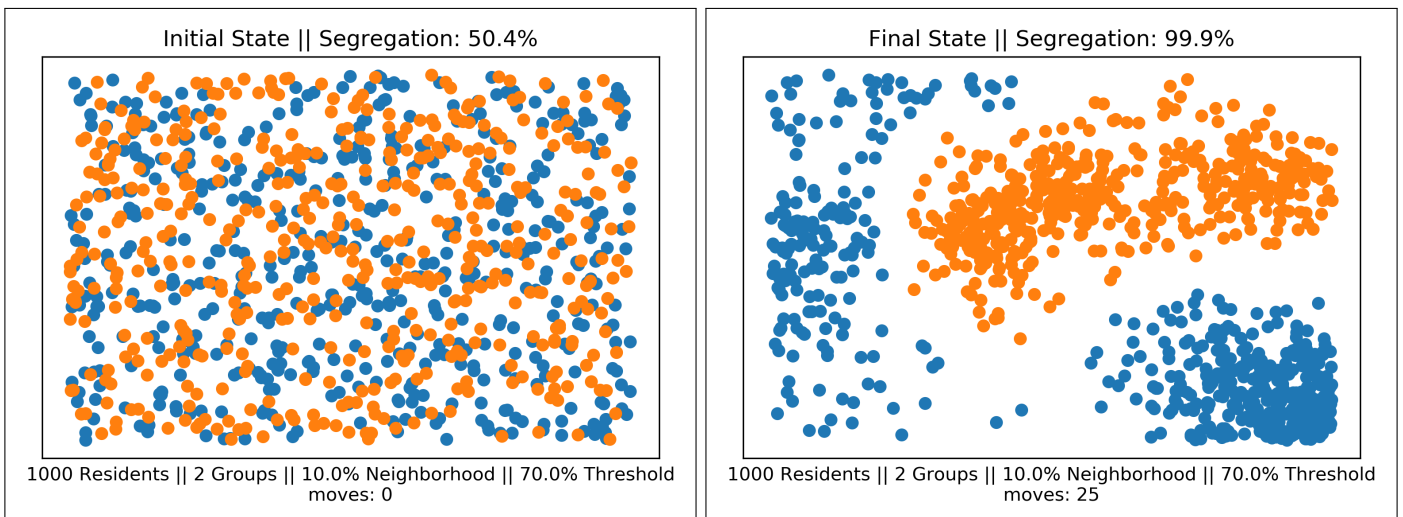


Figure 2: Simulation of Group Segregation. Higher happiness threshold leads to almost complete segregation.

Figure 2 shows that overall similarity ratio was initially 50.4%. The high happiness threshold resulted in an almost completely segregated community with overall similarity ratio of 99.9%. The diagram shows this clearly.

Just like in the first model, Randomly assigning residents to 2 groups distributes the population evenly, leading to an initial overall similarity ratio around 50%. However, even with a low happiness threshold of 31%, segregation naturally occurs, with overall similarity ratio much higher than what residents are willing to tolerate. Thus, on a macro level, observing segregation may not be indicative of what people feel at the micro level. Hence, the answer to the question *How high should the agents' threshold be in order for segregation to occur?* is also: **not so high**.

The code for the second model is much shorter than the first. One of the major drawbacks, however, is the implementation time (depending on the chosen parameters). The more agents involved, and the bigger the neighborhood radius of each agent (thus, the more neighbors to be checked for similarity), the longer it takes to run the model. This is an improvement, however, as the interaction between agents are more natural compared to the first model.



## Appendix A: Model 1 Full Code

```
1 ### Needed libraries ###
2
3 # For pairing x- and y-values to create Cartesian products; an ordered pair represents the address a
  resident
4 import itertools
5
6 # For shuffling all houses in the community to randomly assign them to residents; and for choosing a
  random empty house to move into (for unhappy residents)
7 import random as rd
8
9 # For visualizing the distribution of the residents using scatterplot
10 import matplotlib.pyplot as plt
11
12 # For checking if a filename already exists, to avoid overwriting files
13 import os
14
15
16
17 ### Agent-based model ###
18
19 class Schelling:
20     def __init__(self, width, height, empty_ratio, happiness_threshold, groups, n_iterations):
21
22         # Width of community grid
23         self.width = width
24
25         # Height of community grid
26         self.height = height
27
28         # Percentage of empty houses
29         self.empty_ratio = empty_ratio
30
31         # Minimum similarity ratio, to be considered happy
32         self.happiness_threshold = happiness_threshold
33
34         # Number of groups of residents
35         self.groups = groups
36
37         # Maximum iterations
38         self.n_iterations = n_iterations
39
40         # List of empty houses
41         self.empty_houses = []
42
43         # Dictionary of residents
44         self.agents = {}
45
46
47     ## Place residents on the community
48
49     def populate(self):
50
51         # Create address (coordinates) of residents
52         all_houses = list(itertools.product(range(self.width), range(self.height)))
53
54         # Shuffle the order of houses
55         rd.shuffle(all_houses)
56
57         # Determine number of empty houses
58         n_empty = int(self.empty_ratio*len(all_houses))
59
60         # Assign first few houses as empty
61         self.empty_houses = all_houses[ : n_empty]
62
63         # Assign the rest to be occupied
64         remaining_houses = all_houses[n_empty : ]
65
66         # Assign houses by group to residents
67         houses_by_group = [remaining_houses[i::self.groups] for i in range(self.groups)]
68
69         # Create dictionary of residents
70         # Each resident is defined by his address and group number
71         for i in range(self.groups):
72             agent = dict(zip(houses_by_group[i], [i+1]*len(houses_by_group[i])))
73             self.agents.update(agent)
```

```

74
75
76 ## Checker if resident is unhappy
77
78 def is_unhappy(self, x, y):
79
80     # Get group number of resident
81     group = self.agents[(x, y)]
82
83     # Initialize variables
84     count_similar = 0
85     count_different = 0
86
87     # Check similarity with bottom left neighbor
88     if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
89         if self.agents[(x-1, y-1)] == group:
90             count_similar += 1
91         else:
92             count_different += 1
93
94     # Check similarity with bottom neighbor
95     if y > 0 and (x, y-1) not in self.empty_houses:
96         if self.agents[(x, y-1)] == group:
97             count_similar += 1
98         else:
99             count_different += 1
100
101     # Check similarity with bottom right neighbor
102     if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
103         if self.agents[(x+1, y-1)] == group:
104             count_similar += 1
105         else:
106             count_different += 1
107
108     # Check similarity with left neighbor
109     if x > 0 and (x-1, y) not in self.empty_houses:
110         if self.agents[(x-1,y)] == group:
111             count_similar += 1
112         else:
113             count_different += 1
114
115     # Check similarity with right neighbor
116     if x < (self.width-1) and (x+1, y) not in self.empty_houses:
117         if self.agents[(x+1,y)] == group:
118             count_similar += 1
119         else:
120             count_different += 1
121
122     # Check similarity with upper left neighbor
123     if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
124         if self.agents[(x-1,y+1)] == group:
125             count_similar += 1
126         else:
127             count_different += 1
128
129     # Check similarity with upper neighbor
130     if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
131         if self.agents[(x,y+1)] == group:
132             count_similar += 1
133         else:
134             count_different += 1
135
136     # Check similarity with upper right neighbor
137     if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
138         if self.agents[(x+1,y+1)] == group:
139             count_similar += 1
140         else:
141             count_different += 1
142
143     # Resident is NOT unhappy, i.e., happy if he has no neighbors
144     if (count_similar + count_different) == 0:
145         return False
146
147     # Check if similarity ratio is below happiness threshold
148     else:
149         return float(count_similar/(count_similar + count_different)) < self.happiness_threshold
150

```

```

151
152 ## Resident moves if he is unhappy
153
154 def move(self):
155
156     # Maximum iterations allowed
157     for i in range(self.n_iterations):
158
159         # Initialize count
160         n_changes = 0
161
162         # Check each resident
163         for agent in self.agents:
164
165             # Activated if resident is unhappy
166             if self.is_unhappy(agent[0], agent[1]):
167
168                 # Get a random empty house
169                 empty_house = rd.choice(self.empty_houses)
170
171                 # Get group number of the resident
172                 agent_group = self.agents[agent]
173
174                 # Assign the empty house to the resident
175                 self.agents[empty_house] = agent_group
176
177                 # Remove the original residence from the dictionary
178                 del self.agents[agent]
179
180                 # Remove the now-occupied house from the list of empty houses
181                 self.empty_houses.remove(empty_house)
182
183                 # Add the house the resident just left to the list of empty houses
184                 self.empty_houses.append(agent)
185
186                 # Count as a move
187                 n_changes += 1
188
189         # Iteration stops if everyone is already happy
190         if n_changes == 0:
191             break
192
193
194 ## Compute similarity ratio
195
196 def similarity(self):
197
198     # Initialize list
199     similarity = []
200
201     # Do for each resident
202     for agent in self.agents:
203
204         # Initialize variables
205         count_similar = 0
206         count_different = 0
207
208         # Get address and group number of the resident
209         x = agent[0]
210         y = agent[1]
211         group = self.agents[(x,y)]
212
213         # Check similarity with bottom left neighbor
214         if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
215             if self.agents[(x-1, y-1)] == group:
216                 count_similar += 1
217             else:
218                 count_different += 1
219
220         # Check similarity with bottom neighbor
221         if y > 0 and (x, y-1) not in self.empty_houses:
222             if self.agents[(x, y-1)] == group:
223                 count_similar += 1
224             else:
225                 count_different += 1
226
227         # Check similarity with bottom right neighbor

```

```

228     if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
229         if self.agents[(x+1, y-1)] == group:
230             count_similar += 1
231         else:
232             count_different += 1
233
234     # Check similarity with left neighbor
235     if x > 0 and (x-1, y) not in self.empty_houses:
236         if self.agents[(x-1, y)] == group:
237             count_similar += 1
238         else:
239             count_different += 1
240
241     # Check similarity with right neighbor
242     if x < (self.width-1) and (x+1, y) not in self.empty_houses:
243         if self.agents[(x+1, y)] == group:
244             count_similar += 1
245         else:
246             count_different += 1
247
248     # Check similarity with upper left neighbor
249     if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
250         if self.agents[(x-1, y+1)] == group:
251             count_similar += 1
252         else:
253             count_different += 1
254
255     # Check similarity with upper neighbor
256     if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
257         if self.agents[(x, y+1)] == group:
258             count_similar += 1
259         else:
260             count_different += 1
261
262     # Check similarity with upper right neighbor
263     if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
264         if self.agents[(x+1,y+1)] == group:
265             count_similar += 1
266         else:
267             count_different += 1
268
269     # Place similarity ratio in the list
270     try:
271         similarity.append(float(count_similar/(count_similar + count_different)))
272
273     # If there are no neighbors, similarity ratio is 1
274     except:
275         similarity.append(1)
276
277     # Compute average similarity ratio over all residents
278     return sum(similarity)/len(similarity)
279
280
281 ## Visualize the state
282
283 def visualize(self, state):
284
285     # Initialize subplots
286     fig, ax = plt.subplots()
287
288     # Assign color to each group (define more as needed)
289     agent_colors = {1: 'b', 2: 'r', 3: 'g', 4: 'c', 5: 'm', 6: 'y', 7: 'k'}
290
291     # Create a scatterplot for each resident, colored based on his group
292     for agent in self.agents:
293         ax.scatter(agent[0]+0.5, agent[1]+0.5, color = agent_colors[self.agents[agent]], edgecolors = 'white')
294
295     # Title
296     ax.set_title('Schelling Model: ' + state + ' State' + '\n' + 'Similarity: ' + str("{:.1f}".format(schelling.similarity()*100)) + '%')
297
298     # Horizontal axis label
299     ax.set_xlabel(str(self.empty_ratio*100) + '% Empty Houses' + '\n' + str(self.happiness_threshold*100) + '% Happiness Threshold' + '\n' + str(self.groups) + ' groups')
300
301     # Ensure all residents are visible

```

```

302     ax.set_xlim([0, self.width])
303     ax.set_ylim([0, self.height])
304
305     # Remove extra tick marks on the axes
306     ax.set_xticks([])
307     ax.set_yticks([])
308
309     # Prepare format of file name
310     filename = 'Model1_' + state
311
312     # Starting filename count
313     i = 1
314
315     # Check if filename already exists; add 1 if it does
316     while os.path.exists('{:d}.png'.format(filename, i)):
317         i += 1
318
319     # Save figure
320     plt.savefig('{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)
321
322
323
324 ### Simulation ###
325
326 # Initialize parameters
327 schelling = Schelling(width = 25, height = 25, empty_ratio = 0.25, happiness_threshold = 0.30, groups
328                      = 2, n_iterations = 500)
329
330 # Place residents on the community
331 schelling.populate()
332
333 # Visualize initial state
334 schelling.visualize('Initial')
335
336 # Allow unhappy residents to move
337 schelling.move()
338
339 # Visualize final state
340 schelling.visualize('Final')
341
342 ##### End of Code #####

```

## Appendix B: Model 2 Full Code

```
1 ### Needed libraries ###
2
3 # For assigning a random location to agents
4 import random as rd
5
6 # For visualizing the distribution of agents using plot
7 import matplotlib.pyplot as plt
8
9 # For checking if a filename already exists, to avoid overwriting files
10 import os
11
12
13
14 ### Initialize class to create residents ###
15
16 class agent:
17     pass
18
19
20
21 ### Create residents ###
22
23 def create_agents():
24
25     # Allow list of residents to be accessed outside the function
26     global agents_list
27
28     # Initialize list
29     agents_list = []
30
31     # Create specified number of residents
32     for each_agent in range(n_agents):
33
34         # Create an agent
35         agent_ = agent()
36
37         # Assign to a random address
38         agent_.x = rd.random()
39         agent_.y = rd.random()
40
41         # Assign to a random group
42         agent_.group = rd.randint(0, groups-1)
43
44         # Place resident on the list
45         agents_list.append(agent_)
46
47
48
49 ### Group residents according to group number ###
50
51 def group_by_number():
52
53     # Allow list of groups to be accessed outside the function
54     global group
55
56     # Initialize list
57     group = []
58
59     # Group according to group number
60     for group_number in range(groups):
61
62         # A resident is grouped with other residents with the same group number
63         group.append([agent_ for agent_ in agents_list if agent_.group == group_number])
64
65
66
67 ### Let resident move if he is not satisfied with his neighbors ###
68
69 def move():
70
71     # Allow number of iterations done to be accessed outside the function
72     global iteration
73
74     # Maximum iterations allowed
75     for iteration in range(n_iterations):
```

```

76 # Initialize count
77 n_changes = 0
78
79 # Check each resident
80 for agent_ in agents_list:
81
82     # Create list of neighbors within the radius specified
83     neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y -
84 neighbor.y)**2 < radius**2]
85
86     # Removes the resident himself from list of neighbors
87     neighbors.remove(agent_)
88
89     # Check if there are neighbors within the radius
90     if len(neighbors) > 0:
91
92         # Compute similarity ratio
93         satisfaction = len([neighbor for neighbor in neighbors if neighbor.group == agent_.group])/len
94 (neighbors)
95
96         # Move resident to a random location if similarity ratio is below threshold
97         if satisfaction < threshold:
98             agent_.x, agent_.y = rd.random(), rd.random()
99
100         # Count as a move
101         n_changes += 1
102
103 # Iteration stops if everyone is already happy
104 if n_changes == 0:
105     break
106
107 ### Compute similarity ratio ###
108
109 def similarity_ratio():
110
111     # Allow overall similarity ratio to be accessed outside the function
112     global overall_similarity_ratio
113
114     # Initialize list
115     similarity_ratios = []
116
117     # Check each resident
118     for agent_ in agents_list:
119
120         # Create list of neighbors within the radius specified
121         neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y -
122 neighbor.y)**2 < radius**2]
123
124         # Remove resident himself from list of neighbors
125         neighbors.remove(agent_)
126
127         # Check if there are neighbors within the radius
128         if len(neighbors) > 0:
129
130             # Place similarity ratio in the list
131             try:
132                 similarity_ratios.append(len([neighbor for neighbor in neighbors if neighbor.type == agent_.
133 type])/len(neighbors))
134
135             # If there are no neighbors, similarity ratio is 1
136             except:
137                 similarity_ratios.append(1)
138
139     # Compute average similarity ratio over all residents
140     overall_similarity_ratio = sum(similarity_ratios)/len(similarity_ratios)
141
142     # Outputs overall similarity ratio
143     return overall_similarity_ratio
144
145
146 ### Visualize the state ###
147
148 def visualize(state):

```

```

149 # Initialize subplots
150 fig, ax = plt.subplots()
151
152 # Create a scatterplot by group, plotting each resident
153 for Group in range(groups):
154     ax.plot([agent_.x for agent_ in group[Group]], [agent_.y for agent_ in group[Group]], 'o')
155
156 # Title
157 ax.set_title(state + ' State' + ' || ' + 'Segregation: ' + str("{:.1f}".format(similarity_ratio()*10
158 0)) + '%')
159
160 # Horizontal axis label
161 ax.set_xlabel(str(n_agents) + ' Residents' + ' || ' + str(groups) + ' Groups' + ' || ' + str(radius*
162 100) + '% Neighborhood || ' + str(threshold*100) + '% Threshold' + '\n' + 'moves: ' + str(iteration
163 ))
164
165 # Remove extra tick marks on the axes
166 ax.set_xticks([])
167 ax.set_yticks([])
168
169 # Prepare format of file name
170 filename = 'Model2_' + state
171
172 # Starting filename count
173 i = 1
174
175 # Check if filename already exists; add 1 if it does
176 while os.path.exists('{f:d}.png'.format(filename, i)):
177     i += 1
178
179 # Save figure
180 plt.savefig('{f:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)
181
182 ### Simulation ###
183
184 # Number of residents
185 n_agents = 1000
186
187 # Number of groups
188 groups = 2
189
190 # Maximum iterations
191 n_iterations = 100
192
193 # Neighborhood radius
194 radius = 0.1
195
196 # Satisfaction threshold
197 threshold = 0.31
198
199 # Create residents
200 create_agents()
201
202 # Group residents according to group number
203 group_by_number()
204
205 # Needed for label of initial state
206 iteration = 0
207
208 # Visualize initial state
209 visualize('Initial')
210
211 # Allow unsatisfied residents to move
212 move()
213
214 # Visualize initial state
215 visualize('Final')
216
217 ##### End of Code #####

```



## References

- [1] Bonabeau, Eric. “Agent-Based Modeling: Methods and Techniques for Simulating Human Systems.” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, 14 Mar. 2002, pp. 7280-7287.
- [2] Castiglione, Filippo. “Agent Based Modeling.” *Scholarpedia*, Brain Corporation, 29 Sep. 2006, [www.scholarpedia.org/article/Agent\\_based\\_modeling](http://www.scholarpedia.org/article/Agent_based_modeling).
- [3] Moujahid, Adil. “An Introduction to Agent-Based Models: Simulating Segregation with Python.” *binPress*, [www.binpress.com/simulating-segregation-with-python/](http://www.binpress.com/simulating-segregation-with-python/).
- [4] Sayama, Hiroki. “Agent-Based Models.” *LibreTexts*, 23 June 2019, [https://math.libretexts.org/Bookshelves/Applied\\_Mathematics/Book%3A\\_Introduction\\_to\\_the\\_Modeling\\_and\\_Analysis\\_of\\_Complex\\_Systems\\_\(Sayama\)/19%3A\\_Agent-Based\\_Models/19.02%3A\\_Building\\_an\\_Agent-Based\\_Model](https://math.libretexts.org/Bookshelves/Applied_Mathematics/Book%3A_Introduction_to_the_Modeling_and_Analysis_of_Complex_Systems_(Sayama)/19%3A_Agent-Based_Models/19.02%3A_Building_an_Agent-Based_Model).
- [5] Schelling, Thomas C. “Dynamic Models of Segregation.” *Journal of Mathematical Sociology*, vol. 1, 1971, pp. 143-186.