Agent-based models of segregation using Python

Patrick Vincent N. Lubenia

6 May 2020

Contents

Agent-Based Models

Agent-based modeling is a framework for modeling and simulating 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 a lot of 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 usually 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. Furthermore, generally, agent-based models: often lack central supervisors/controllers and may produce nontrivial "collective behavior" as a whole.

One must keep in mind the following scientific method-based approach when designing an agent-based model:

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

And in order to be scientifically meaningful, an agent-based model must be built and used as follows:

1. Built using empirically-derived assumptions, then simulate to produce emergent behavior: for predictions
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. Visualization
3. Updating.

Agents are placed/activated in the model. The system is then visualized to grasp the initial state of the model. Finally, environment is updated and the agents are allowed to move accordingly. Intermediate states may be visualized once more or just the final state in order to determine how much the system has changed.

The agent-based modeling framework is open-ended and flexible. It may be tempting to add lots of details 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: 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 that is used in agent-based modeling in Python is a `class`. A `class` is created objects with their own built-in functions are desired. The following example creates a `rectangle` class with built-in functions for computing the rectangle's perimeter and area:

```
1 class rectangle:
2   def __init__(self, length, width):
3     self.length = length
4     self.width = width
5   def perimeter(self):
6     return 2*self.length + 2*self.width
7   def area(self):
8     return self.length*self.width
```

Line 1 creates the class. In line 2, *self* is a placeholder for the variable name that is given to the rectangle. The same line also shows that calling `rectangle` needs two inputs: length and width. For example, to create a rectangle r with length 3 and width 4:

```
1 r = rectangle(3, 4)
```

Lines 3-4 allow the length and width of rectangle r to be called:

```
r.length
r.width
```

Inside the `rectangle` class, the built-in functions `perimeter` (lines 5-6) and `area` (lines 7-8) are defined. They call on the dimensions inputted when the rectangle is created to compute the rectangle's perimeter and area, respectively. These built-in functions are called *methods*. To get the perimeter and area of rectangle r:

```
r.perimeter()
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:

```
class rectangle:
  def __init__(self, length, width):
    self.length = length
    self.width = width
    self.diagonal = []
```

After creating rectangle r as above, typing

```
r.diagonal
```

simply returns an empty list. The extra variable is for methods that utilize the said variable. Add now a function that computes the diagonal of the rectangle and places that value inside the variable called diagonal. `numpy` needs to be imported for computing the square root. The full code is as follows:

```
import numpy as np

class rectangle:
  def __init__(self, length, width):
    self.length = length
    self.width = width
    self.diagonal = []
  def perimeter(self):
    return 2*self.length + 2*self.width
  def area(self):
    return self.length*self.width
  def diag(self):
    self.diagonal = np.sqrt(self.length**2 + self.width**2)
```

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

```
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

```
class perfomer:
  pass
```

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

```
p = performer()
```

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

```
p.x = 3
p.y = 4
p.name = 'Luca'
p.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]. This model follows the framework that an agent-based model has the following basic 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. 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 four parameters of an agent-based model 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:

```
import itertools
import random as rd
import matplotlib.pyplot as plt
import os
```

1. itertools: For pairing x- and y-values to create Cartesian products; an ordered pair represents the address a resident
2. random: 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)
3. pyplot: For visualizing the distribution of the residents using scatterplot
4. os: For checking if a filename already exists, to avoid overwriting files

Environment: creating the class

The community is represented by a grid, and each 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.

```
class Schelling:
  def __init__(self, width, height, empty_ratio, happiness_threshold, groups, n_iterations):
    self.width = width
    self.height = height
    self.empty_ratio = empty_ratio
    self.happiness_threshold = happiness_threshold
    self.groups = groups
    self.n_iterations = n_iterations
    self.empty_houses = []
    self.agents = {}
```

A class called `Schelling` is created. This needs six inputs:

- width: Width of the community
- height: Height of the community
- empty_ratio: Ratio of empty houses to total houses
- happiness_threshold: Minimum ratio of similar neighbors to different neighbors desired by a resident
- groups: Number of groups in the community
- n_iterations: Maximum number of times to run the simulation while there are still unhappy residents

There are also two extra variables that are used later:

- empty_houses: A list of the addresses of empty houses
- agents: A dictionary defining the residents; each resident is defined by his
  - Address (an ordered pair)
  - Group (a number)

Agents: creating a method

A method called `populate` is added: it randomly scatters the residents throughout the community:

```
1  def populate(self):
2    all_houses = list(itertools.product(range(self.width), range(self.height)))
3    rd.shuffle(all_houses)
4    n_empty = int(self.empty_ratio*len(all_houses))
5    self.empty_houses = all_houses[ : n_empty]
6    remaining_houses = all_houses[n_empty : ]
7    houses_by_group = [remaining_houses[i::self.groups] for i in range(self.groups)]
8    for i in range(self.groups):
9      agent = dict(zip(houses_by_group[i], [i+1]*len(houses_by_group[i])))
10     self.agents.update(agent)
```

Line 2 creates a list of all the houses: each house is represented by its coordinates in the grid. The `product` function inside `itertools` creates the Cartesian products using the whole numbers of the width and the height of the community. In line 3, the `shuffle` function from `random` shuffles all the houses so that the list of houses is in random order. Line 4 counts how many empty houses there should be based on the empty_ratio indicated. Line 5 assigns this number of empty houses to the first houses in the list. The rest of the houses are assigned in another variable in line 6. The remaining houses are then assigned (by skip counting) to the different groups in line 7, e.g., if there are 3 groups, houses 0, 0+3, 0+3+3, and so on (according to order in the list) are assigned to the first group; houses 1, 1+3, 1+3+3, and so on to the second group; and houses 2, 2+3, 2+3+3, and so on to the third group. Each group of houses within a group is a list themselves (which makes the variable remaining_houses is a list of lists). Lastly, lines 8-10 create a dictionary of all residents: line 9 gets each ordered pair (the Cartesian product) in each group of houses by group and pairs them with their corresponding group number. This creates a dictionary whose keys are ordered pairs, and values are group numbers. Line 10 combines all these dictionaries in one. 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).

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    group = self.agents[(x, y)]
3    count_similar = 0
4    count_different = 0
5    if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
6      if self.agents[(x-1, y-1)] == group:
7        count_similar += 1
8      else:
9        count_different += 1
10   if y > 0 and (x, y-1) not in self.empty_houses:
11     if self.agents[(x, y-1)] == group:
12       count_similar += 1
13     else:
14       count_different += 1
15   if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
16     if self.agents[(x+1, y-1)] == group:
17       count_similar += 1
18     else:
19       count_different += 1
20   if x > 0 and (x-1, y) not in self.empty_houses:
21     if self.agents[(x-1,y)] == group:
22       count_similar += 1
23     else:
24       count_different += 1
25   if x < (self.width-1) and (x+1, y) not in self.empty_houses:
26     if self.agents[(x+1,y)] == group:
27       count_similar += 1
28     else:
29       count_different += 1
30   if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
31     if self.agents[(x-1,y+1)] == group:
32       count_similar += 1
33     else:
34       count_different += 1
35   if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
36     if self.agents[(x,y+1)] == group:
37       count_similar += 1
38     else:
39       count_different += 1
40   if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
41     if self.agents[(x+1,y+1)] == group:
42       count_similar += 1
43     else:
44       count_different += 1
45   if (count_similar + count_different) == 0:
46     return False
47   else:
48     return float(count_similar/(count_similar + count_different)) < self.happiness_threshold
```

The method calls on the x- and y-coordinates of a resident's address (line 1) since the method is used inside another method, and is not used independently, unlike the rest of the other methods. Line 2 gets the group number of the resident being analyzed. Lines 3-4 initialize the count for the number of people similar or different, respectively, in group to the resident. Line 5 checks if there is a neighbor residing to the lower left of the resident (consequently, the resident is not at the leftmost edge of the community nor at the bottom left corner). If that neighbor is of the same group as the resident, then a count is added to count_similar (lines 6-7). Otherwise, a count is added to count_different (lines 8-9). Lines 10-14 check the neighbor below the resident (consequently, the resident is not at the bottom edge of the community). Lines 15-19 check the lower right neighbor (the resident is not at the bottom edge nor at the bottom right corner). Lines 20-24 check the left side of the resident (the resident is not at the leftmost edge). Lines 25-29 check the resident's right side (the resident is not at the rightmost edge). Lines 30-34 check the resident's upper left neighbor (the resident is not at the top edge nor at the top left corner). Lines 35-39 check the resident's neighbor on top (resident is not at the top edge). Lastly, lines 40-44 check the resident's upper right neighbor (the resident is not at the top edge nor at the top right corner). In lines 45-46, if the resident has no neighbor (hence, no similar or different count), then `is_unhappy` returns False, i.e., the resident is happy. Otherwise, 48 checks the similarity ratio against the happiness threshold: 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    for i in range(self.n_iterations):
3      n_changes = 0
4      for agent in self.agents:
5        if self.is_unhappy(agent[0], agent[1]):
6          empty_house = rd.choice(self.empty_houses)
7          agent_group = self.agents[agent]
8          self.agents[empty_house] = agent_group
9          del self.agents[agent]
10         self.empty_houses.remove(empty_house)
11         self.empty_houses.append(agent)
12         n_changes += 1
13     if n_changes == 0:
14       break
```

Line 2 allows the `move` method to run based on the number of iterations indicated. Line 3 initializes the count for the number of relocations. If everyone is already happy and no one moves, lines 13-14 stop the iteration process. Line 4 goes through each resident. If a resident is unhappy (using the `is_unhappy` in line 5), a random empty house is chosen from the list of empty houses (line 6), and the current resident's group is assigned to it (line 7): effectively, the resident is reassigned to a different house and the dictionary of residents is updated (line 8). Line 9 removes the old record of the reassigned resident. Line 10 removes the now-occupied house from the list of empty houses. And line 11 puts the house left by the resident into the list of empty houses. Finally, line 12 add a count if a resident moves.

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    similarity = []
3    for agent in self.agents:
4      count_similar = 0
5      count_different = 0
6      x = agent[0]
7      y = agent[1]
8      group = self.agents[(x,y)]
9      if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
10       if self.agents[(x-1, y-1)] == group:
11         count_similar += 1
12       else:
13         count_different += 1
14     if y > 0 and (x, y-1) not in self.empty_houses:
15       if self.agents[(x, y-1)] == group:
16         count_similar += 1
17       else:
18         count_different += 1
19     if x < (self.width-1) 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     if x > 0 and (x-1, y) not in self.empty_houses:
25       if self.agents[(x-1, y)] == group:
26         count_similar += 1
27       else:
28         count_different += 1
29     if x < (self.width-1) and (x+1, y) not in self.empty_houses:
30       if self.agents[(x+1, y)] == group:
31         count_similar += 1
32       else:
33         count_different += 1
34     if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
35       if self.agents[(x-1, y+1)] == group:
```

```
36            count_similar += 1
37          else:
38            count_different += 1
39        if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
40          if self.agents[(x, y+1)] == group:
41            count_similar += 1
42          else:
43            count_different += 1
44        if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
45          if self.agents[(x+1,y+1)] == group:
46            count_similar += 1
47          else:
48            count_different += 1
49        try:
50          similarity.append(float(count_similar/(count_similar + count_different)))
51        except:
52          similarity.append(1)
53      return sum(similarity)/len(similarity)
```

Line 2 initializes the list which contains the similarity ratios of all residents. For each resident, the x- (line 6) and y-coordinates (line 7) of their address, and their group number (8) are extracted. Lines 9-48 do exactly the same steps as in the `is_unhappy` method. Lines 49-50 append the similarity ratio to the list started in line 2. If there is an error in computation, which only occurs when we 0/0 happens, a similarity ratio of 1 is appended (lines 51-52). Based on the final list of similarity ratios, line 53 computes the overall average.

Visualization

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

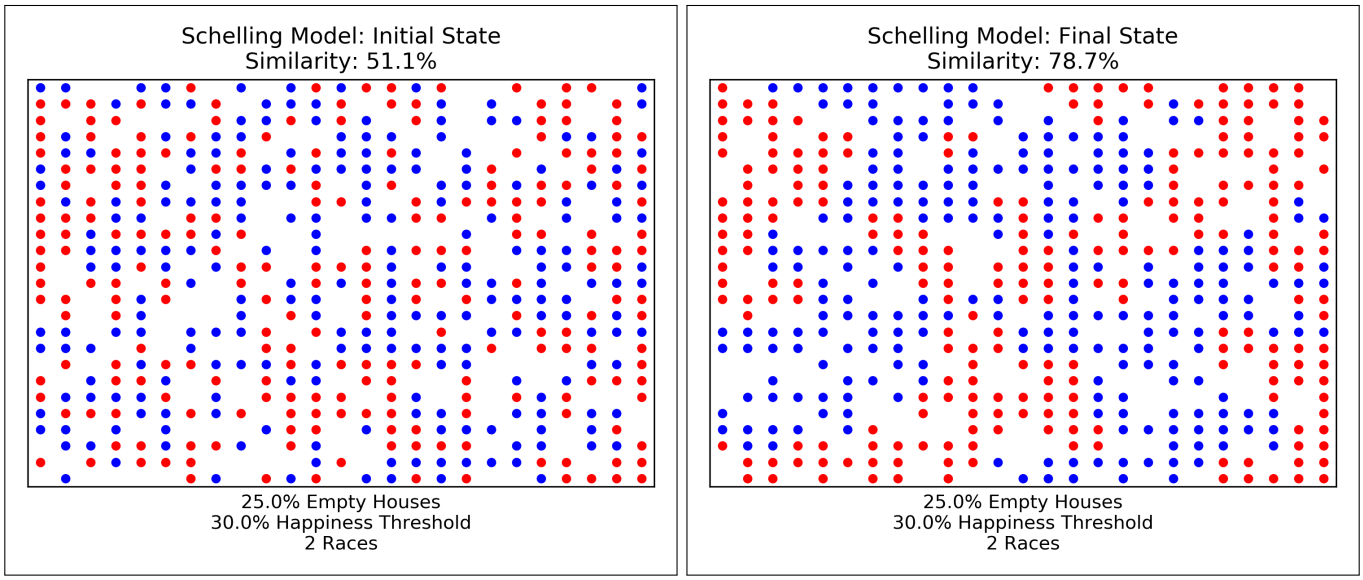
```
1   def plot(self, state):
2     fig, ax = plt.subplots()
3     agent_colors = {1:'b', 2:'r', 3:'g', 4:'c', 5:'m', 6:'y', 7:'k'}
4     for agent in self.agents:
5       ax.scatter(agent[0]+0.5, agent[1]+0.5, color = agent_colors[self.agents[agent]],
6                  edgecolors = 'white')
7     ax.set_title('Schelling Model: ' + state + ' State' + '\n' + 'Similarity: '
8                  + str("{:.1f}".format(schelling.similarity()*100)) + '%')
9     ax.set_xlabel(str(self.empty_ratio*100) + '% Empty Houses' + '\n'
10                 + str(self.happiness_threshold*100) + '% Happiness Threshold' + '\n'
11                 + str(self.groups) + ' groups')
12    ax.set_xlim([0, self.width])
13    ax.set_ylim([0, self.height])
14    ax.set_xticks([])
15    ax.set_yticks([])
16    filename = 'Model1_' + state
17    i = 1
18    while os.path.exists('{}{:d}.png'.format(filename, i)):
19      i += 1
20    plt.savefig('{}{: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"; line 1). 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 (line 2 creates subplots). Line 3 assigns colors to each group (up to 7 groups). Lines 4-6 create a scatterplot: each resident is placed in its address in the grid with its corresponding group color. The additional 0.5 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. Lines 7-8 place a title on the figure, indicating the indicated state. The label on the horizontal axis includes the empty ratio, happiness threshold, and number of groups in the community (lines 9-11). Lines 12-13 ensures that all residents are visible in the figure. Lines 14-15 remove the tickmarks on the axes. Line 16 starts the filename of the figure. Line 17 starts the figure count at 1. Line 18 checks if the current file number exists. If it does, 19 adds 1 to the counter. Finally, line 20 saves the figure with high resolution (300 dots per inch). The tight specification removes extra white spaces around the figure. Lines 16-20 prevent files from being overwritten by successive runs of the model.

Implementation

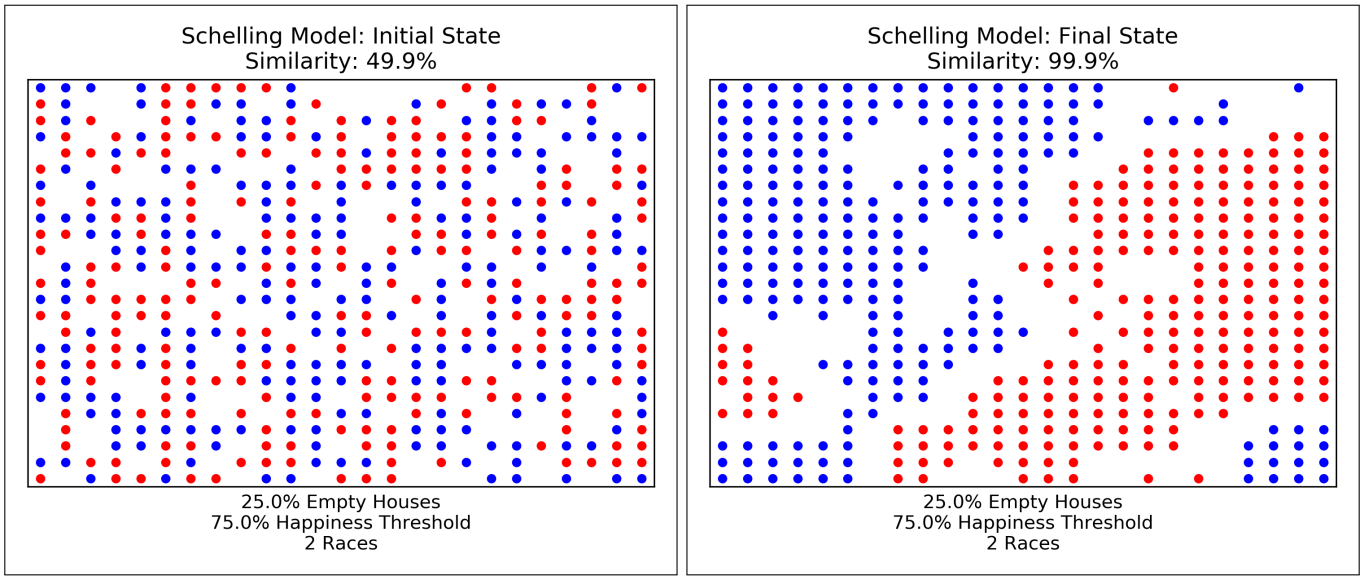
The simulation follows the basic code implementation structure mentioned in the introduction: agents are created (Initialization), then they are allowed to move according to their defined behavior (Updating). Plots are used to visualize (Visualization) what happens to the system before and after the 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.

```
schelling = Schelling(width = 25, height = 25, empty_ratio = 0.25, happiness_threshold = 0.30,
                      groups = 2, n_iterations = 500)
schelling.populate()
schelling.plot('Initial')
schelling.move()
schelling.plot('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]. This model follows the framework that an agent-based model has a more granular structure. The following tasks 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:

```
import random as rd
import matplotlib.pyplot as plt
import os
```

1. random: For assigning a random location to agents
2. pyplot: For visualizing the distribution of agents using plot
3. os: For checking if a filename already exists, to avoid overwriting files

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:

```
class agent:
  pass
```

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

```
1  def create_agents():
2    global agents_list
3    agents_list = []
4    for each_agent in range(n_agents):
5      agent_ = agent()
6      agent_.x = rd.random()
7      agent_.y = rd.random()
8      agent_.group = rd.randint(0, groups-1)
9      agents_list.append(agent_)
```

`global` in line 2 allows the variable called agents_list to be accessible outside the function. It is automatically created when the function is run. Line 3 initializes the list of residents. Lines 4-8 create the indicated number of residents (n_agents) using the class `agent`, and puts the following attributes in each resident: random x- and y-coordinates, and group number (based on the variable groups). Finally, line 9 places each resident in the list on line 3.

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    global group
3    group = []
4    for group_number in range(groups):
5      group.append([agent_ for agent_ in agents_list if agent_.group == group_number])
```

Line 2 allows the variable group to be automatically created when the function is run. Line 3 initializes the list of groups. Lines 4-5 start with the first group, get each resident in the list that belong to that group, and create a list that is placed in the list on line 3 (thus, the variable group is a list of lists). The process is repeated for all group numbers.

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") will be skipped.
2. Residents do not do anything by themselves so Modeling Task 2c ("describe the rules for how agents behave on their own") will also be skipped.

Modeling Task 2d: 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    global iteration
3    for iteration in range(n_iterations):
4      n_changes = 0
5      for agent_ in agents_list:
6        neighbors = [neighbor for neighbor in agents_list
7                     if (agent_.x - neighbor.x)**2 + (agent_.y - neighbor.y)**2 < radius**2 and
8                         neighbor != agent_]
9        if len(neighbors) > 0:
10         satisfaction = len([neighbor for neighbor in neighbors
11                             if neighbor.group == agent_.group])/len(neighbors)
12         if satisfaction < threshold:
13           agent_.x, agent_.y = rd.random(), rd.random()
14           n_changes += 1
15     if n_changes == 0:
16       break
```

To keep track of the number of iterations between the initial and final state, line 2 automatically releases the vairable iteration outside the function ones the for loop in line 3 terminates (for a maximum of n_iterations). Line 4 initializes the count for the number of relocations in the succeeding for loop. Lines 5-8 consider each resident, create a list of neighbors who fall within the defined radius with the resident as the center. The last condition prevents the resident from counting itself as a neighbor. In lines 9-11, if the resident has neighbors, its satisfaction (or similarity ratio as defined in the previous model) is computed: in lines 12-14, if the satisfaction is below the defined threshold, the resident moves to a random location, and the counter for the number of relocations increases by 1. Lines 15-16 allows the iteration to terminate if no resident moves, i.e., everyone is already satisfied.

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    global overall_similarity_ratio
3    similarity_ratios = []
4    for agent_ in agents_list:
5      neighbors = [neighbor for neighbor in agents_list
6                   if (agent_.x - neighbor.x)**2 + (agent_.y - neighbor.y)**2 < radius**2 and
7                       neighbor != agent_]
8      if len(neighbors) > 0:
9        try:
10         similarity_ratios.append(len([neighbor for neighbor in neighbors
11                                       if neighbor.type == agent_.type])/len(neighbors))
12       except:
13         similarity_ratios.append(1)
14   overall_similarity_ratio = sum(similarity_ratios)/len(similarity_ratios)
15   return overall_similarity_ratio
```

Line 2 allows the variable overall_similarity_ratio to be automatically created when the function is run. Line 3 initializes the list of similarity ratios computed for each resident (done in lines 7-11). If there is an error in computation, which only occurs when we 0/0 happens, a similarity ratio of 1 is appended (lines 12-13). The overall similarity ratio is computed as the average among all similarity ratios (line 14). Line 15 allows the overall similarity ratio value to be called explicitly outside the function.

Visualization

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

```
1  def visualize(state):
2    fig, ax = plt.subplots()
3    for Group in range(groups):
4      ax.plot([agent_.x for agent_ in group[Group]], [agent_.y for agent_ in group[Group]], 'o')
5    ax.set_title(state + ' State' + ' || '
6                 + 'Segregation: ' + str("{:.1f}".format(similarity_ratio()*100)) + '%')
7    ax.set_xlabel(str(n_agents) + ' Residents' + ' || '
8                  + str(groups) + ' Groups' + ' || '
9                  + str(radius*100) + '% Neighborhood || '
10                 + str(threshold*100) + '% Threshold' + '\n'
11                 + 'moves: ' + str(iteration))
12   ax.set_xticks([])
13   ax.set_yticks([])
14   filename = 'Model2_' + state
15   i = 1
16   while os.path.exists('{}{:d}.png'.format(filename, i)):
17     i += 1
18   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"; line 1). Subplots are used instead of the simpler `plt.plot` so that when the file is run, successive runs of plots do not overlap into one figure (line 2 creates subplots). Lines 3-4 get the x- and y-coordinates of each resident in each group and plot them. Lines 5-6 place the state and overall similarity ratio in the title. Lines 7-11 place along the horizontal axis the

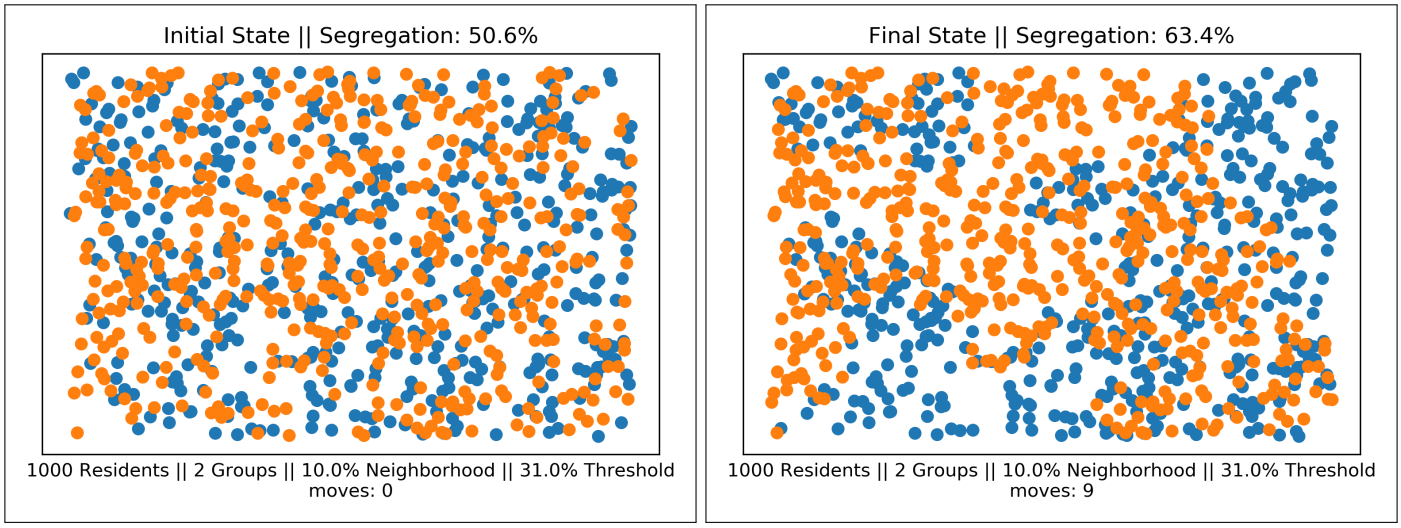
number of residents, number of groups, neighborhood radius of each resident, and the number of iterations to reach the state. Lines 12-13 remove the tickmarks on the axes. Line 14 starts the filename of the figure. Line 15 starts the figure count at 1. Line 16 checks if the current file number exists. If it does, 17 adds 1 to the counter. Finally, line 18 saves the figure with high resolution (300 dots per inch). The tight specification removes extra white spaces around the figure. Lines 14-18 prevent files from being overwritten by successive runs of the model.

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.

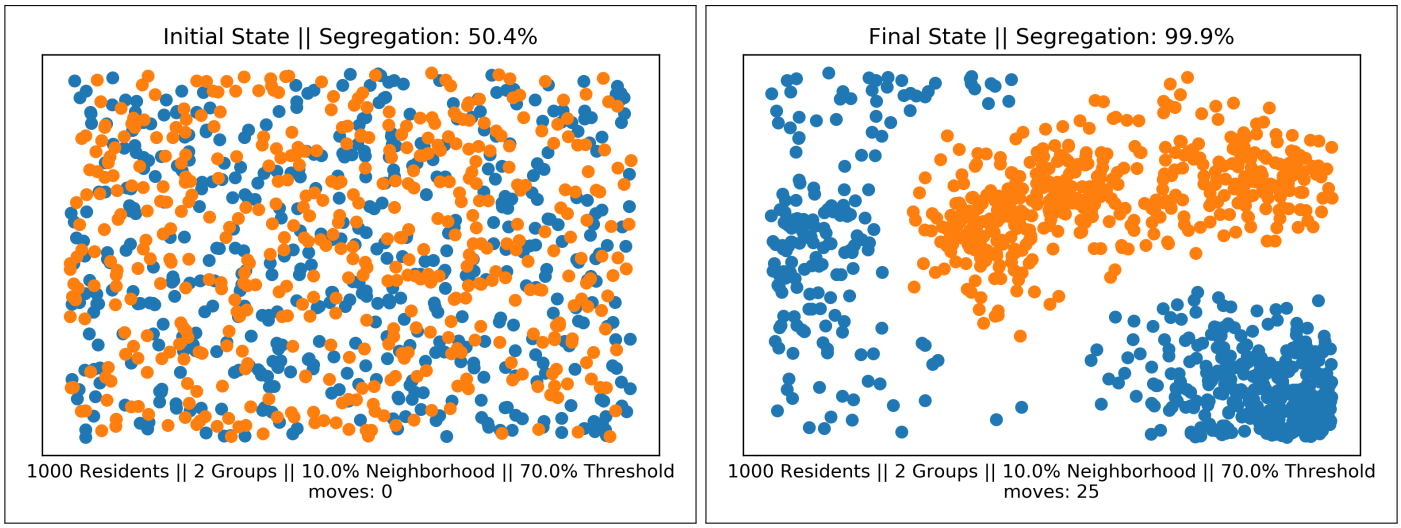
```
1  n_agents = 1000
2  groups = 2
3  n_iterations = 100
4  radius = 0.1
5  threshold = 0.31
6  create_agents()
7  group_by_number()
8  iteration = 0
9  visualize('Initial')
10 move()
11 visualize('Final')
```

The iteration number is defined in line 8 in order for the count to appear in the graph of the initial state.



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.

Using a higher happiness threshold of 70%:



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 firs 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 (74 lines vs 165). 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.

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.

[3] Moujahid, Adil. "An Introduction to Agent-Based Models: Simulating Segregation with Python." *binPress*, 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.