Agent-based models of spread of infectious diseases using Python

SIR and SEIR Model

Patrick Vincent N. Lubenia

13 May 2020

Contents

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.

Empty `class`

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

```
1 class perfomer:
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 model below.

SIR Model

Overview

The Susceptible-Infectious-Recovered (SIR) Model is a compartmental model (members of the population are classified into distinct compartments) in epidemiology. This modeling framework stems from Kermack and McKendrick [3]. During an epidemic, people in a population are classified as

1. Susceptible: Capable of contracting the disease
2. Infectious: Has the disease and is able to transmit it to susceptible people
3. Recovered: Had the disease and no longer has it

In the basic model, a person transitions linearly from being susceptible, to being infectious, and finally to being recovered. A person belonging to particular compartment cannot go back to any previous state. Furthermore, once a person recovers, he is already immune to getting the disease. Complicated models allow movements between compartments, even allowing going back to a previous state.

The SIR Model tracks the number of members in each compartment as time goes by. Starting with a given number of susceptible people, this number declines as they interact with infectious individuals, which, in turn, decrease as people recover. This simple transition between states are modeled using ordinary differential equations.

Let t be time (in days), S the number of susceptible people, I the number of infectious people, R the number of recovered people, N the total number of people in the population, β the transmission rate, and γ the recovery rate. The set of ordinary differential equations describing the change in the compartments over time is given by:

$$\begin{aligned}
\frac{dS}{dt} &= -\frac{\beta I}{N} S \\
\frac{dI}{dt} &= \frac{\beta I}{N} S - \gamma I \\
\frac{dR}{dt} &= \gamma I.
\end{aligned}$$

These equations assume random mixing of individuals in the population, regardless of their state. Hence, infectious people are still able to mingle with everyone else. Solving these equations simultaneouly gives the following result:

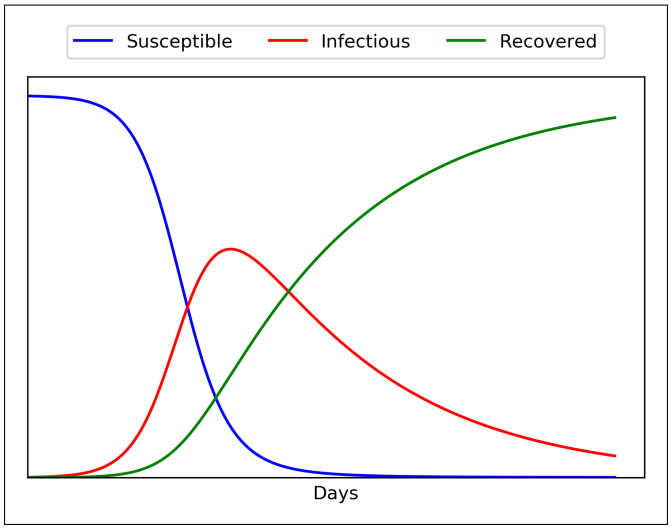

Figure 1: SIR Model. Solution to SIR Model.

Figure 1 shows that at the start of an epidemic, the number of infectious individuals exponentially increase as there are numerous susceptibles people. As infectious people recover, eventually, infectiousness peak, then start to decline. The purpose of the model that follows is to show that agent-based models can simulate the same trend in spread of infectious diseases in a "natural" way without using equations.

This model follows an agent-based modeling framework with the following tasks that must be undertaken [4]:

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 Code

Needed Libraries

To start the code, the following libraries are imported:

```
# For assigning agents to random locations and for choosing random agents
import random as rd

# For visualizing the state of the agents using plot
import matplotlib.pyplot as plt

# For checking if a filename already exists, to avoid overwriting files
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". The agents in this model are people with different states (compartments) located in different places. Thus, each agent needs the following attributes:

- Spatial location
- State: Susceptible, Infectious, or Recovered
- Number of days the person has had the disease
- Is the person immune already?

The class called `agent` is initialized:

```
class agent:
  pass
```

A function called `create_agents` is defined to create the population:

```
def create_agents():

  # Allow the following variable to be accessed outside the function
  global agents_list

  # Initialize list of agents
  agents_list = []

  # Create specified number of agents
  for i in range(n_agents):

    # Create an agent
    agent_ = agent()

    # Assign the agent to a random location
    agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)

    # Set the agent to be initially susceptible
    # 0 = susceptible, 1 = infectious, 2 = recovered
    agent_.state = 0

```

```
    # Count number of days the agent is infectious
    agent_.days_infectious = 0

    # Add the agent to the list
    agents_list.append(agent_)
```

`global` allows variables to be accessed outside the function. It is automatically created when the function is run. Note that this function needs 1 variable to be predefined: n_agents (total number of people in the population).

Note: In this model, there is nothing special about the environment where the people are moving around so Modeling Task 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.

Modeling Task 2c: Behavior of agents

The next task is to "describe the rules for how agents behave on their own". This model explores 2 ways the people behave:

1. They go about their usual ways whether they are sick or not
2. Infectious people do not move around (they sort of limit their social contact by not moving around but others are free to engage them).

Scenario 1: In the scenario where people act as if there is no epidemic, the function `move` is very simple to simulate moving around.

```
def move():

  # Agent moves to a random location
  for agent_ in agents_list:
    agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)
```

Scenario 2: In the scenario where social contact is somewhat limited, the function can be modified easily. Infectious people are excluded from being assigned random coordinates.

```
def move():

  # Infectious people do not move
  for agent_ in agents_list:
    if agent_.state != 1:
      agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)
```

Modeling Task 2d: Interaction of agents

The final task is to "describe the rules for how agents interact with each other". In this model, an infectious person infects non-immune people within a specified radius.

The function `infect` simultaneously infects people within the radius of an infectious in different locations:

```
def infect():

  # Allow the following variable to be accessed outside the function
  global infectious_list

  # Create list of infectious people
  infectious_list = [agent_ for agent_ in agents_list if agent_.state == 1]

  # Create list of neighbors of infectious
  for agent_ in infectious_list:

    # Create list of neighbors within the radius specified
    neighbors = [neighbor for neighbor in agents_list if (neighbor.x - agent_.x)**2 + (neighbor.y -
    agent_.y)**2 <= infectious_radius**2]

    # Remove person himself from list of neighbors
    neighbors.remove(agent_)

    # Susceptible neighbors become infectious (recovered individuals are immune)
    for agent_ in neighbors:
      if agent_.state == 0:
        agent_.state = 1
```

Note that this function needs 1 variable to be predefined: infectious_radius (radius of neighborhood where people inside can become infectious).

Updating States

Since infectious people eventually recover after a certain number of days, the duration of their infectiousness must be monitored. This updating occurs at the start of each day.

```
def monitor():

  # Add count for number of days an infectious person has been infectious
  for agent_ in agents_list:
    if agent_.state == 1:
      agent_.days_infectious += 1

      # Infectious people who reach the end of infectious period recover
      if agent_.days_infectious > days_to_recover:
        agent_.state = 2
```

Note that this function needs 1 variable to be predefined: days_to_recover (number of days a person stays infectious before recovering).

Counting

To determine the progression of the disease, the number of susceptible, infectious, and recovered people must be counted.

```
def count():

  # Allow the following variables to be accessed outside the function
    global S_count, I_count, R_count

    # Count susceptible
    S_count = len([agent_ for agent_ in agents_list if agent_.state == 0])

    # Count infectious
    I_count = len([agent_ for agent_ in agents_list if agent_.state == 1])

    # Count recovered
    R_count = len([agent_ for agent_ in agents_list if agent_.state == 2])
```

Visualization

2 functions are used for visualization:

1. One to visualize the states of the system
2. Another to graph the trend of the various compartments of the model.

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

```
def visualize():

  # To ensure updated count of compartments
  count()

  # Initialize subplots
  fig, ax = plt.subplots()

  # Assign color per compartment
  agent_color = {0: 'b', 1: 'r', 2: 'g'}

  # Plot each person with his corresponding color
  for agent_ in agents_list:
    ax.plot(agent_.x, agent_.y, '.', color = agent_color[agent_.state])

  # Dummy plots for legend
  ax.plot([], [], 'bo', label = 'Susceptible: '+ str(S_count))
  ax.plot([], [], 'ro', label = 'Infectious: '+ str(I_count))
  ax.plot([], [], 'go', label = 'Recovered: '+ str(R_count))

```

```
  # Place legend below the figure
  ax.legend(ncol = 3, loc = 'upper center', bbox_to_anchor = (0.5, -0.01))

  # Fix the window: choose largest range for random location choices
  ax.set_xlim([0, 1])
  ax.set_ylim([0, 1])

  # Title
  ax.set_title('Day: ' + str(day_count), loc = 'left')

  # Remove extra tick marks on the axes
  ax.set_xticks([])
  ax.set_yticks([])

  # Prepare format of file name
  filename = 'SIR'

  # Starting filename count
  i = 1

  # Check if filename already exists; add 1 if it does
  while os.path.exists('{}{:d}.png'.format(filename, i)):
    i += 1

  # Save figure
  plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)
```

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. Viewing window shoud be the as large as possible to include the largest range for random numbers generated for the coordinates. The number of iterations in the title represents day count. The tight specification in `savefig` removes extra white spaces around the figure. Note that this function needs 1 variable to be predefined: day_count (number of iterations done).

To show the trend of the various compartments, the function `trend` is created:

```
def trend():

  # To ensure updated count of compartments
  count()

  # Initialize subplots
  fig, ax = plt.subplots()

  # Plot daily count of each compartment
  ax.plot(days, S_list, color = 'blue', label = 'Susceptible')
  ax.plot(days, I_list, color = 'red', label = 'Infectious')
  ax.plot(days, R_list, color = 'green', label = 'Recovered')

  # Place legend on top of the figure
  ax.legend(ncol = 3, loc = 'upper center', bbox_to_anchor = (0.5, 1.15))

  # Set the axes to meet at (0, 0)
  ax.set_xlim(left = 0)
  ax.set_ylim(bottom = 0)

  # Horizontal axis label
  ax.set_xlabel('Days')

  # Prepare format of file name
  filename = 'SIR_Trend'

  # Starting filename count
  i = 1

  # Check if filename already exists; add 1 if it does
  while os.path.exists('{}{:d}.png'.format(filename, i)):
    i += 1

  # Save figure
  plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)
```

The idea is similar to the function `visualize`, except that the daily figures for each compartment are graphed instead, and the legend is placed on top of the figure. Note that this function needs 4 variable to be predefined: days (day counts), S_list (daily number of susceptible), I_list (daily number of infectious), and R_list (daily number of recovered).

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.

Day 0

To follow the SIR model framework, the initial number of susceptible, infectious, and recovered individuals are specified. The total gives the number of agent in the agent-based model. In this model, there are initially 1,000 susceptible, 1 infectious, and 0 recovered. The infectious radius is 0.025, and it takes 7 days for an infectious to recover:

```
# Comment all visualize() for faster simulation
# But replace with count()

## Initial day

# Start day count
day_count = 0

# Initial values
S_initial = 1000
I_initial = 1
R_initial = 0

# Initial total population
n_agents = S_initial + I_initial + R_initial

# Infectious radius
infectious_radius = 0.025

# Number of days of infectiousness
days_to_recover = 7

# Initialize list of days
days = []
days.append(0)

# Initialize daily count of susceptible
S_list = []
S_list.append(S_initial)

# Initialize daily count of infectious
I_list = []
I_list.append(I_initial)

# Initialize daily count of recovered
R_list = []
R_list.append(R_initial)

# Create the population
create_agents()

# Choose a specified number of random people and make them infectious
for agent_ in rd.sample(agents_list, I_initial):
  agent_.state = 1

# From the remaining susceptible, choose a specified number of random people and make them recovered
for agent_ in rd.sample([agent_ for agent_ in agents_list if agent_.state != 1], R_initial):
  agent_.state = 2

visualize()
# count()
```

Since created agents are initially susceptible, initial number of infectious people are incorporated. `sample` is used to ensure chosen people are not repeated. From the remaining susceptible people, random individuals are initially assigned as recovered.

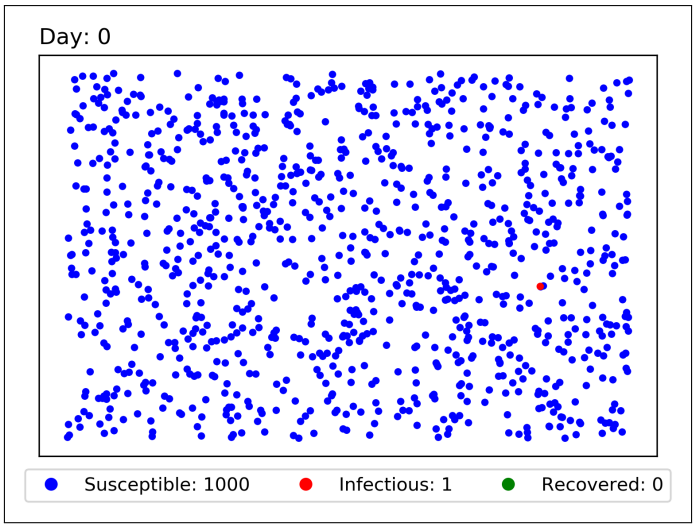

Figure 2: Initial State of the Population. 1,000 susceptible, 1 infectious, and 0 recovered.

As shown in Figure 2, at the end of Day 0, 1 infectious individual appeared. Assume that he only starts infecting other people the following day.

Scenario 1

Consider the first scenario where people move normally as if there is no epidemic. The simulation code goes:

```
## Succeeding days

# Number of runs
runs = 100

for i in range(runs):

  # Add day count
  day_count += 1

  # Monitor infectiousness and recovery at the start of the day
  monitor()

  # Let everyone move
  move()

  # Visualize new positions
  visualize()
  # count()

  # Infections start after the movement
  infect()

  # Visualize
  visualize()
  # count()

  # Update lists
  days.append(day_count)
  S_list.append(S_count)
  I_list.append(I_count)
  R_list.append(R_count)

  # Stop simulation once number of infectious becomes 0
  if I_count == 0:
    break

# Plot trend
trend()
```

The maximum number of runs (corresponding to day counts) is indicated for the simulation. As mentioned in the section Updating States, monitoring occurs at the start of the day

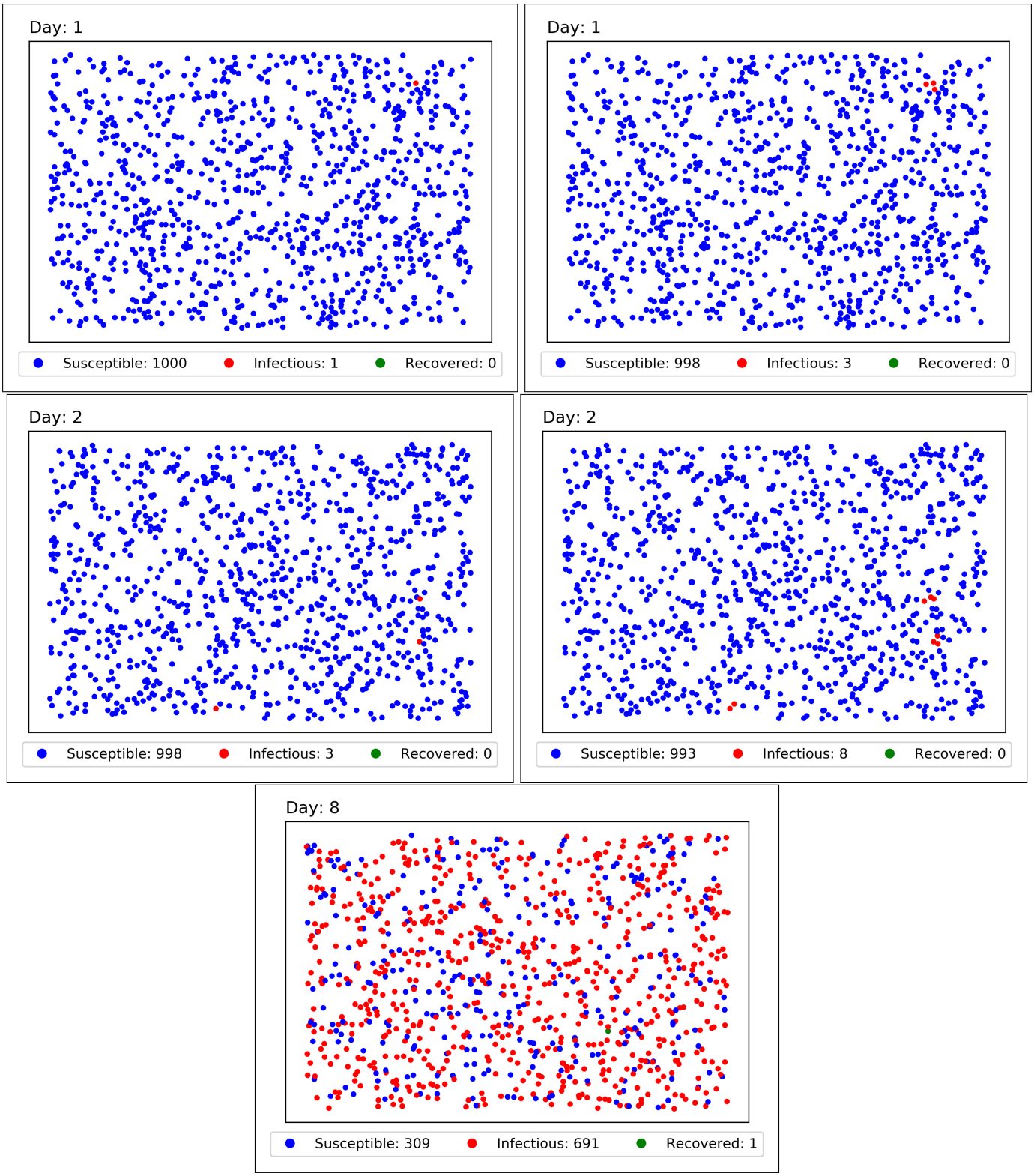

Figure 3: Scenario 1, Some States of the Population. Once infected, an infectious individual starts infecting the following day. The first recovery happens at the start of Day 8, after 7 days of infectiousness.

The top left panel of Figure 3 shows the start of Day 1 after the population moved. The upper right panel shows 2 additional infections at the end of the day. In the middle left panel, the start of Day 2, the 3 infectious individuals scattered around the population. By the end of Day 2, the middle right panel, there were already a total of 8 infectious people. The last panel illustrates the appearance of the recovery of the first infectious person, which happened after 7 days, the recovery time.

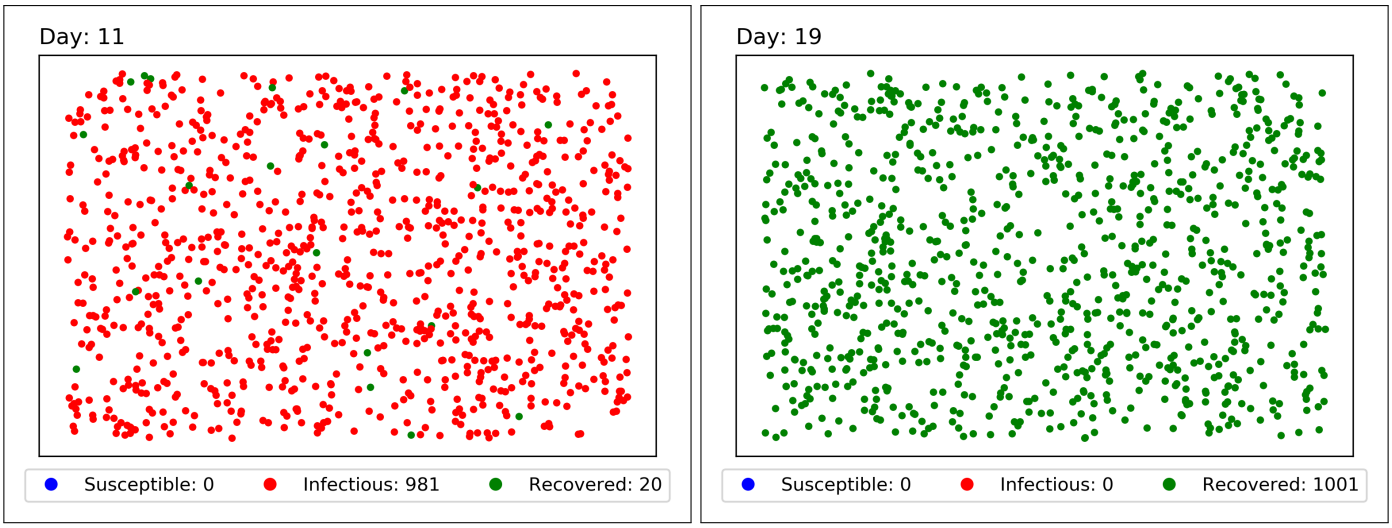

Figure 4: Scenario 1, Some States of the Population. By the end of Day 11, there are no more susceptible individuals. By Day 19, everyone has already recovered.

Figure 4 shows that by the end of Day 11, everyone was already either infectious or recovered. The epidemic has reached its peak, and the infectious are just waiting for recover. By Day 19, everyone had already recovered.

To better appreciate what happened to the population, the following graph is generated:

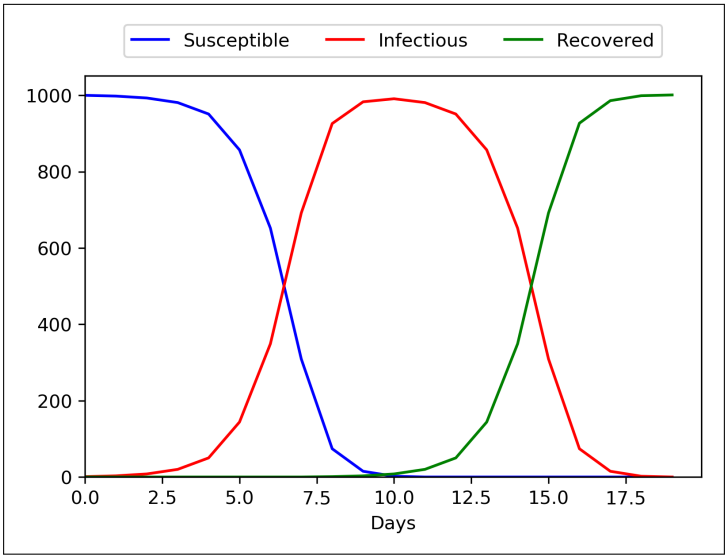

Figure 5: Scenario 1, SIR Model. Daily trend of the number of susceptible, infectious, and recovered individuals.

In Figure 5, as expected in any spread of disease, the number of infectious individuals initially increased exponentials, reached a peak, then declined. The trend of the 3 compartments are similar to the trend shown when the solution to the SIR system of ordinary differential equations in the Overview were graphed.

Scenario 2

Consider now the second scenario where infectious people do not move while they are infectious. Use the alternative `move` function defined earlier.

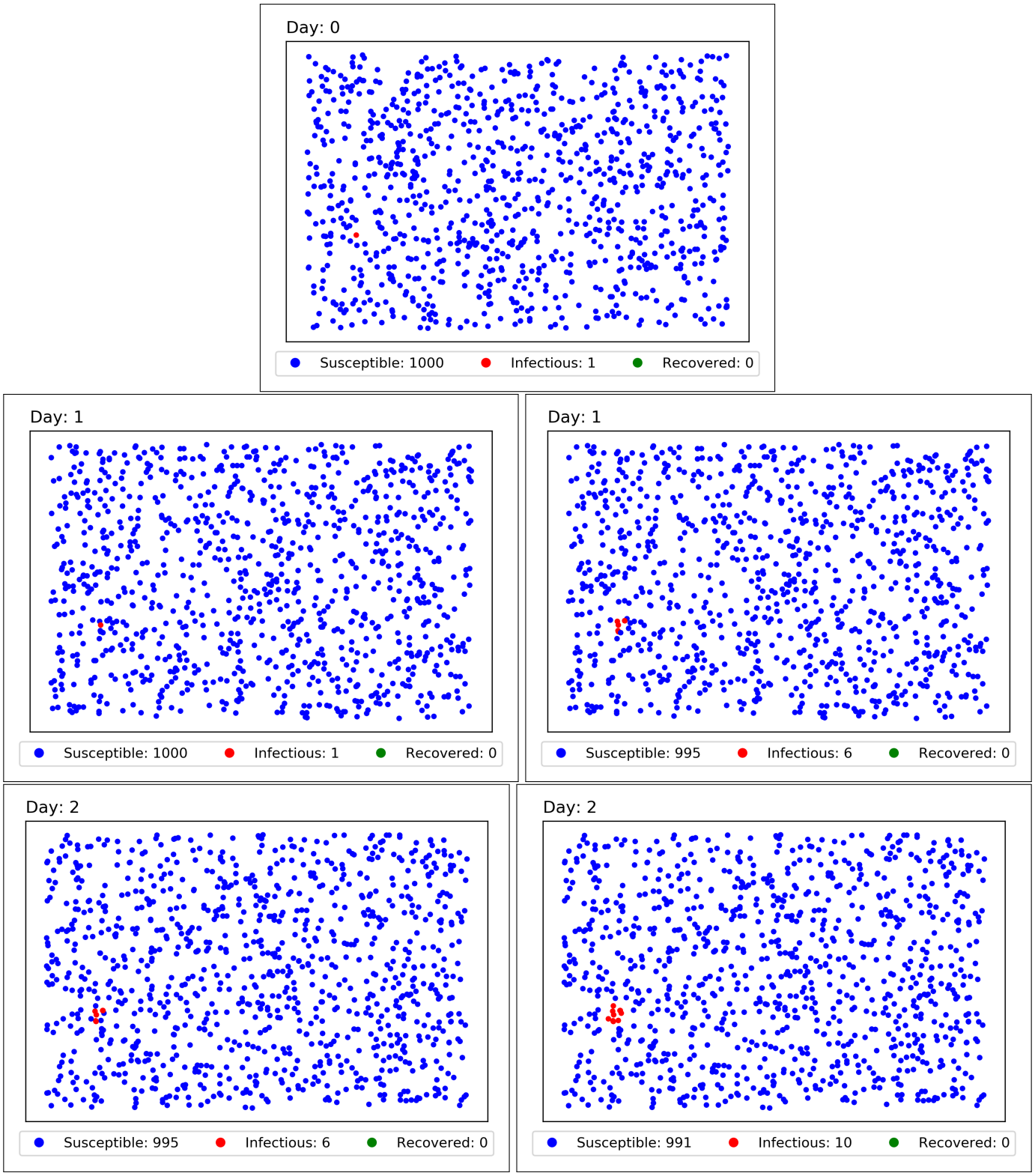

Figure 6: Scenario 2, Some States of the Population. Infectious individuals do not move.

The top panel in Figure 6 shows the initial state of the population. Notice that at the start of Day 1 (middle left panel), the infectious individual has not moved. The rest of the panels show that the infection is largely contained in his area (until other people approach him).

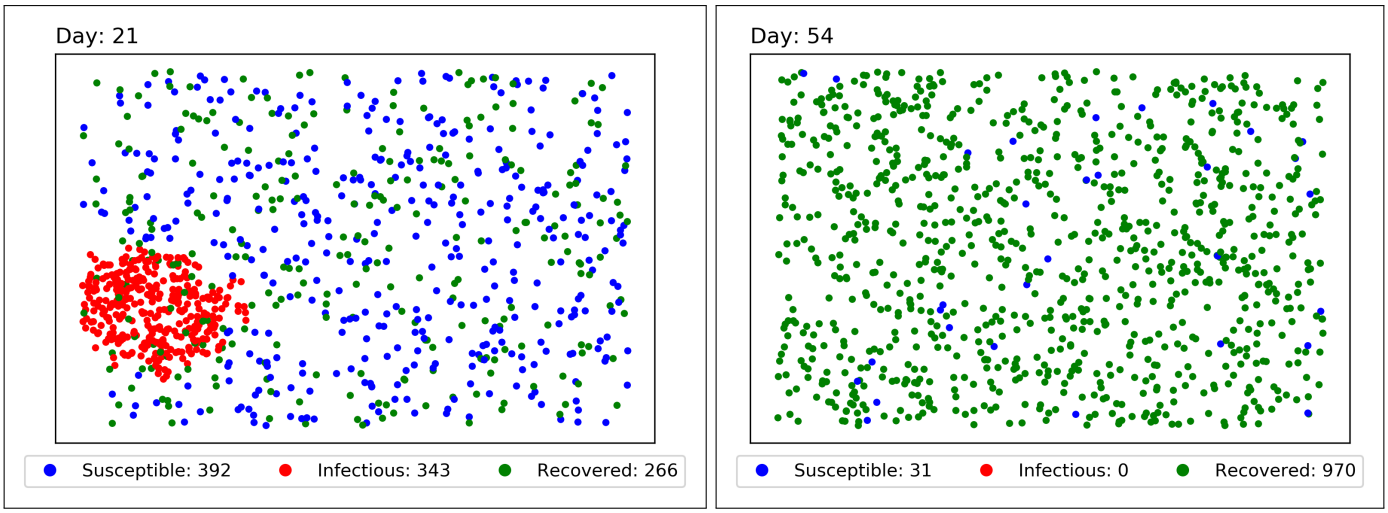

Figure 7: Scenario 2, Some States of the Population. By Day 21, the epidemic is at its peak. And by Day 54, the epidemic has stopped.

In Figure 7, the left diagram shows the peak of the epidemic at Day 21. The epidemic was contained in the neighborhood of the first infected individual. The number of infectious people increased as susceptible individuals came into contact with them: there were no limitations to the movement of non-infectious people. Compared to the first scenario, by the end of the epidemic, there were still some people who were never infected in scenario 2.

The improvement in the containment of the epidemic is evident in the trend graph:

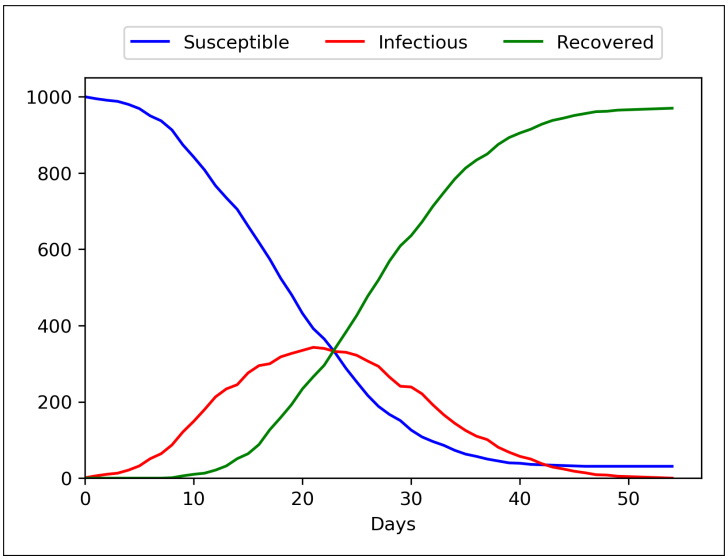

Figure 8: Scenario 2, SIR Model. Daily trend of the number of susceptible, infectious, and recovered individuals.

Figure 8 shows a clear flattening of the curve. Compared to scenario 1, even if the duration of the epidemic was longer in scenario 2, the number of individuals declined. In scenario 1, at the peak of the epidemic, almost everyone was infectious. Having a lot of infectious people at a time can overwhelm the health care capacity of a community. On the other hand, scenario 2 gave a more manageable scenario wherein about only 35% of the population was being taken care of at the peak of the epidemic. Again, the trend of the 3 compartments are similar to the trend shown when the solution to the SIR system of ordinary differential equations in the Overview was generated.

SEIR Model

A modification to the SIR model is the inclusion of an Exposed stage where a person already has the virus causing the disease, but is not yet infectious. Exposed individuals cannot transmit the disease. The time it takes before an infected individual becomes infectious is called the incubation period.

It is easy to incorporate the incubation period in the agent-based model. The following are the modifications for each function.

`create_agents`: An attribute `days_exposed` is added to count the number of days a person has been exposed. Note that the attribute `state` does not change but an additional state is allowed: 3 for exposed.

```
def create_agents():
  ...
    agent_.days_exposed = 0
```

`move`: For the second scenario, both infectious and exposed people are not allowed to move. Originally, only infectious individuals were not permitted.

```
def move():
  for agent_ in agents_list:
    if agent_.state != 1 and agent_.state != 3:
      agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)
```

`infect`: After creating a list of all neighbors, susceptible neighbors become exposed. Originally, susceptible neighbors became infectious.

```
def infect():
  ...
    for agent_ in neighbors:
      if agent_.state == 0:
        agent_.state = 3
```

`monitor`: The code below is added to update the count for the number of days an exposed person has been exposed. If the person reaches the end of the incubation period, he becomes infectious.

```
def monitor():
  ...
  for agent_ in agents_list:
    if agent_.state == 3:
      agent_.days_exposed += 1
      if agent_.days_exposed > incubation_period:
        agent_.state = 1
```

`count`: Count for the number of exposed people are added.

```
def count():
  ...
  global E_count
  ...
  E_count = len([agent_ for agent_ in agents_list if agent_.state == 3])
```

`visualize`: Color magenta is assigned for exposed individuals. The dummy plot and legend are updated to include exposed people. And the file name is changed to SEIR.

```
def visualize():
  ...
  agent_color = {0: 'b', 1: 'r', 2: 'g', 3: 'm'}
  ...
  ax.plot([], [], 'mo', label = 'Exposed: '+ str(E_count))
  ...
  ax.legend(ncol = 2, loc = 'upper center', bbox_to_anchor = (0.5, -0.01))
  ...
  filename = 'SEIR'
```

`trend`: Like in `visualize`, the plot and legend are updated to include exposed people. Note that the legend is pushed a little bit further up since the layout now contains two rows. The file name is changed to SEIR_Trend.

```
def trend():
  ...
  ax.plot(days, E_list, color = 'magenta', label = 'Exposed')
  ...
  ax.legend(ncol = 2, loc = 'upper center', bbox_to_anchor = (0.5, 1.2))
  ...
  filename = 'SEIR_Trend'
```

Simulation: The simulation involves no initial exposed individual, and an incubation period of 3 days. Movement is based on scenario 2 where infectious and exposed people do not move.

```
...
E_initial = 0
...
n_agents = S_initial + E_initial + I_initial + R_initial
...
incubation_period = 3
...
E_list = []
E_list.append(E_initial)
...
for agent_ in rd.sample([agent_ for agent_ in agents_list if agent_.state != 1 and agent_.state != 2],
     E_initial):
  agent_.state = 3
...
for i in range(runs):
  ...
  E_list.append(E_count)
  ...
  if I_count == 0 and E_count == 0:
    break
```

For the simulation, the initial value for the number of exposed people is added; thus, the variable n_agents must include this value as well. The incubation period parameter must be set. The list of daily number of exposed individuals is initialized. Excluding those assigned to be infectious and recovered, random people are chosen based the number of initial exposed people. The for loop is updated to append the count for the number of exposed people to the initialized list. Finally, the simulation stops before the end if both the number of infectious and exposed individuals have both reached 0.

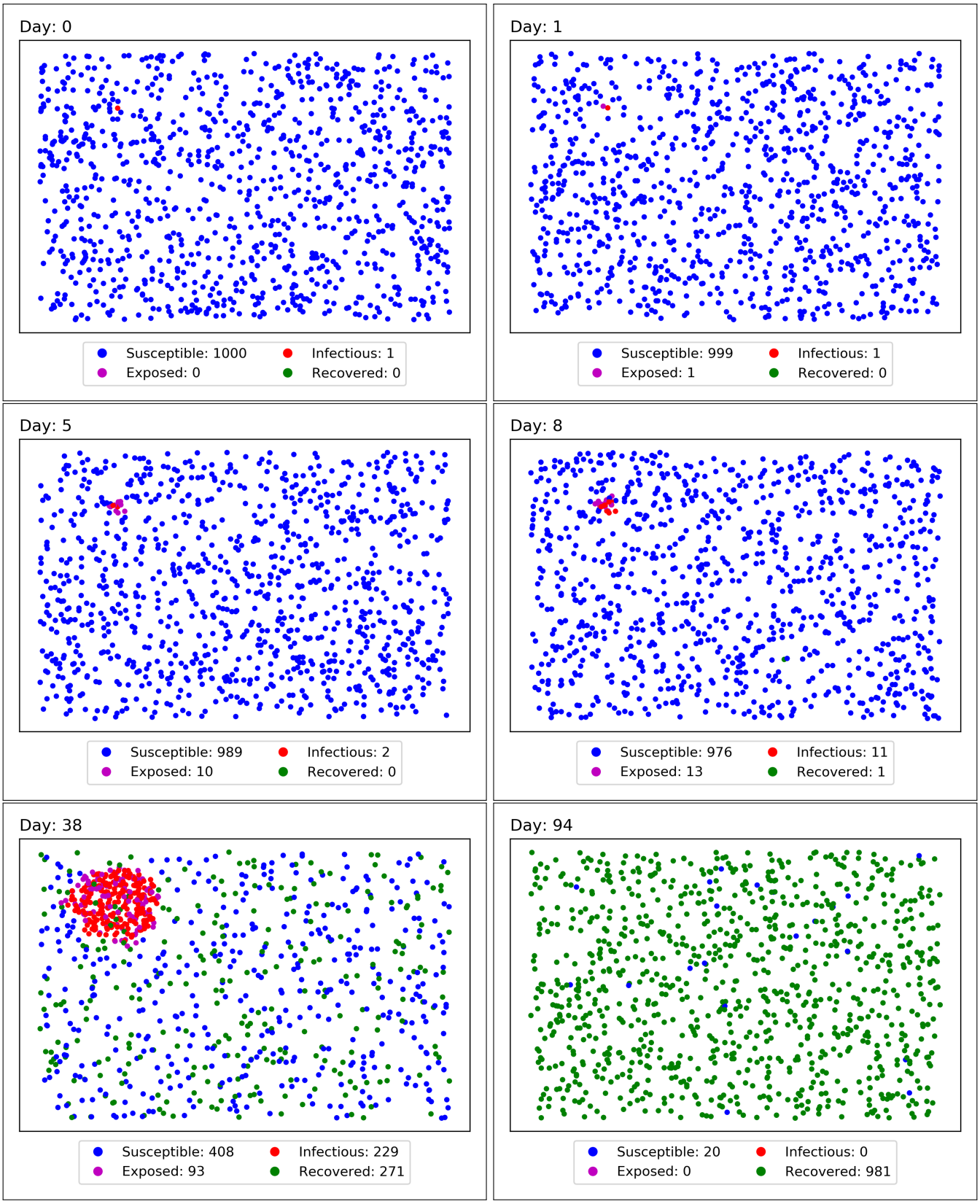

Figure 9: SEIR Model, Some States of the Population. Exposed and infectious individuals do not move.

The lower left panel of Figure 9 shows that the disease at its peak. It was localized in the vicinity of the initial infectious individual. It is noticeable, however, that it took longer for the peak to arrive (Day 38 vs Day 21 in SIR model) because of the incubation period added into the model. It took even longer for the end of the epidemic to come as susceptible people had a lot of opportunities to come into contact with infectious people, even if they were localized (lower right panel).

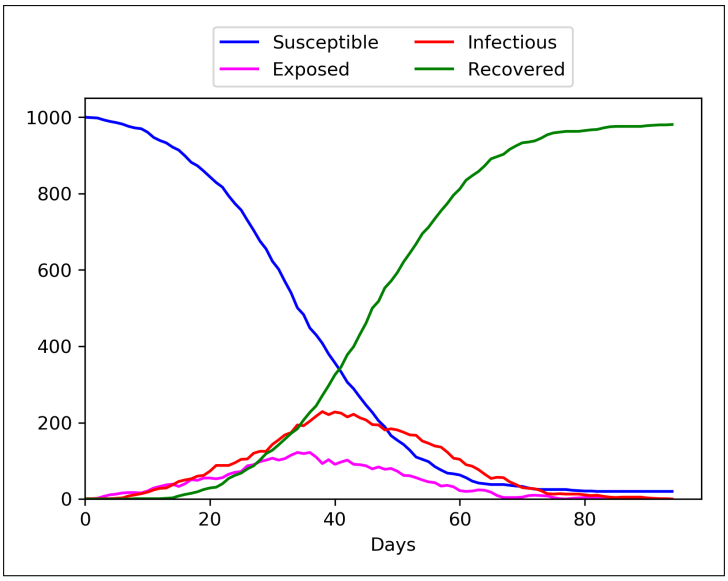

Figure 10: SEIR Model. Daily trend of the number of susceptible, exposed, infectious, and recovered individuals.

Figure 10 shows even more flattening of the curve compared to scenario 2 of the SIR model due to the early arrest of exposed individuals. The trend of the 4 compartments are comparable to the trend shown when the solution to the SIR system of ordinary differential equations in the Overview was generated.

Note: Let E the number of exposed people and σ the latency rate. The set of ordinary differential equations describing the change in the compartments over time for the SEIR Model is given by:

$$\begin{aligned}
\frac{dS}{dt} &= -\frac{\beta I}{N} S \\
\frac{dE}{dt} &= \frac{\beta I}{N} S - \sigma E \\
\frac{dI}{dt} &= \sigma E - \gamma I \\
\frac{dR}{dt} &= \gamma I.
\end{aligned}$$

Conclusion

The SIR and SEIR compartmental models are widely used in modeling the spread of infectious diseases. Typically, modelers use the equation-based models because they are easier to implement. However, as shown in this paper, agent-based models, even if more involved in programming, are easier to modify because of the intuitiveness of its components.

One of the most powerful arguments for agent-based models is the ease with which modifications can be incorporated into the model. In this paper, it was quite easy to to transition from the SIR to the SEIR framework. In equation based models, this can be done easily by simply adding terms and equations. However, the effects of the added terms are not readily interpretable. They only become apparent when the solution to the system has been generated. And even then, interpretation may be cumbersome especially if one is not skilled in interpreting how ordinary differential equations behave. In an agent-based model, the modifications are logical. The simulation just needs to be run, and the result interpreted based on logic.

Improvements to the model can be easily done. As an example, to simulate quarantine, the random coordinates assigned to agents in the `move` function may be limited to a certain area or a far location. Probabilities may also be incorporated so that the state of being exposed, infectious, or recovered depend on certain probability values. Bounded areas of movement may also be added to simulate social distancing.

Agent-based models are powerful, although they can be very tedious to make. It can be less difficult by following an organized framework so that the different components, e.g., environment, agents, interactions. And once the general model has been setup, modifications can be easily done.

Appendix A: SIR Model Full Code

```
### Needed libraries ###

# For assigning agents to random locations and for choosing random agents
import random as rd

# For visualizing the state of the agents using plot
import matplotlib.pyplot as plt

# For checking if a filename already exists, to avoid overwriting files
import os



### Initialize class to create people ###

class agent:
  pass



### Create population ###

def create_agents():

  # Allow the following variable to be accessed outside the function
  global agents_list

  # Initialize list of agents
  agents_list = []

  # Create specified number of agents
  for i in range(n_agents):

    # Create an agent
    agent_ = agent()

    # Assign the agent to a random location
    agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)

    # Set the agent to be initially susceptible
    # 0 = susceptible, 1 = infectious, 2 = recovered
    agent_.state = 0

    # Count number of days the agent is infectious
    agent_.days_infectious = 0

    # Add the agent to the list
    agents_list.append(agent_)



### Everyone moves to a random location ###

# Scenario 1: Normal day

# Comment scenario not used
# def move():

# # Agent moves to a random location
# for agent_ in agents_list:
#   agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)

## Scenario 2: Somewhat limited social contact for infectious people

def move():

  # Infectious people do not move
  for agent_ in agents_list:
    if agent_.state != 1:
      agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)



### Infection happens within a radius (for non-recovered people) ###

```

```
def infect():

    # Allow the following variable to be accessed outside the function
    global infectious_list

    # Create list of infectious people
    infectious_list = [agent_ for agent_ in agents_list if agent_.state == 1]

    # Create list of neighbors of infectious
    for agent_ in infectious_list:

        # Create list of neighbors within the radius specified
        neighbors = [neighbor for neighbor in agents_list if (neighbor.x - agent_.x)**2 + (neighbor.y
    - agent_.y)**2 <= infectious_radius**2]

        # Remove person himself from list of neighbors
        neighbors.remove(agent_)

        # Susceptible neighbors become infectious (recovered individuals are immune)
        for agent_ in neighbors:
            if agent_.state == 0:
                agent_.state = 1



### Monitor progress of infectious people at the start of the day ###

def monitor():

  # Add count for number of days an infectious person has been infectious
  for agent_ in agents_list:
    if agent_.state == 1:
      agent_.days_infectious += 1

      # Infectious people who reach the end of infectious period recover
      if agent_.days_infectious == days_to_recover + 1:
        agent_.state = 2



### Count members of each compartment ###

def count():

  # Allow the following variables to be accessed outside the function
    global S_count, I_count, R_count

    # Count susceptible
    S_count = len([agent_ for agent_ in agents_list if agent_.state == 0])

    # Count infectious
    I_count = len([agent_ for agent_ in agents_list if agent_.state == 1])

    # Count recovered
    R_count = len([agent_ for agent_ in agents_list if agent_.state == 2])



### Visualize population ###

def visualize():

  # To ensure updated count of compartments
  count()

  # Initialize subplots
  fig, ax = plt.subplots()

  # Assign color per compartment
  agent_color = {0: 'b', 1: 'r', 2: 'g'}

  # Plot each person with his corresponding color
  for agent_ in agents_list:
    ax.plot(agent_.x, agent_.y, '.', color = agent_color[agent_.state])

  # Dummy plots for legend
  ax.plot([], [], 'bo', label = 'Susceptible: '+ str(S_count))
```

```
  ax.plot([], [], 'ro', label = 'Infectious: '+ str(I_count))
  ax.plot([], [], 'go', label = 'Recovered: '+ str(R_count))

  # Place legend below the figure
  ax.legend(ncol = 3, loc = 'upper center', bbox_to_anchor = (0.5, -0.01))

  # Fix the window: choose largest range for random location choices
  ax.set_xlim([0, 1])
  ax.set_ylim([0, 1])

  # Title
  ax.set_title('Day: ' + str(day_count), loc = 'left')

  # Remove extra tick marks on the axes
  ax.set_xticks([])
  ax.set_yticks([])

  # Prepare format of file name
  filename = 'SIR'

  # Starting filename count
  i = 1

  # Check if filename already exists; add 1 if it does
  while os.path.exists('{}{:d}.png'.format(filename, i)):
    i += 1

  # Save figure
  plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)



### Graph trend of compartments ###

def trend():

  # To ensure updated count of compartments
  count()

  # Initialize subplots
  fig, ax = plt.subplots()

  # Plot daily count of each compartment
  ax.plot(days, S_list, color = 'blue', label = 'Susceptible')
  ax.plot(days, I_list, color = 'red', label = 'Infectious')
  ax.plot(days, R_list, color = 'green', label = 'Recovered')

  # Place legend on top of the figure
  ax.legend(ncol = 3, loc = 'upper center', bbox_to_anchor = (0.5, 1.15))

  # Set the axes to meet at (0, 0)
  ax.set_xlim(left = 0)
  ax.set_ylim(bottom = 0)

  # Horizontal axis label
  ax.set_xlabel('Days')

  # Prepare format of file name
  filename = 'SIR_Trend'

  # Starting filename count
  i = 1

  # Check if filename already exists; add 1 if it does
  while os.path.exists('{}{:d}.png'.format(filename, i)):
    i += 1

  # Save figure
  plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)



### Simulation ###

# Comment all visualize() for faster simulation
# But replace with count()

```

```
## Initial day

# Start day count
day_count = 0

# Initial values
S_initial = 1000
I_initial = 1
R_initial = 0

# Initial total population
n_agents = S_initial + I_initial + R_initial

# Infectious radius
infectious_radius = 0.025

# Number of days of infectiousness
days_to_recover = 7

# Initialize list of days
days = []
days.append(0)

# Initialize daily count of susceptible
S_list = []
S_list.append(S_initial)

# Initialize daily count of infectious
I_list = []
I_list.append(I_initial)

# Initialize daily count of recovered
R_list = []
R_list.append(R_initial)

# Create the population
create_agents()

# Choose a specified number of random people and make them infectious
for agent_ in rd.sample(agents_list, I_initial):
  agent_.state = 1

# From the remaining susceptible, choose a specified number of random people and make them recovered
for agent_ in rd.sample([agent_ for agent_ in agents_list if agent_.state != 1], R_initial):
  agent_.state = 2

visualize()
# count()


## Succeeding days

# Number of runs
runs = 100

for i in range(runs):

  # Add day count
  day_count += 1

  # Monitor infectiousness and recovery at the start of the day
  monitor()

  # Let everyone move
  move()

  # Visualize new positions
  visualize()
  # count()

  # Infections start after the movement
  infect()

  # Visualize
  visualize()
  # count()

```

```
  # Update lists
  days.append(day_count)
  S_list.append(S_count)
  I_list.append(I_count)
  R_list.append(R_count)

  # Stop simulation once number of infectious becomes 0
  if I_count == 0:
    break

# Plot trend
trend()

####### end of code #######
```

Appendix B: SEIR Model Full Code

```
### Needed libraries ###

# For assigning agents to random locations and for choosing random agents
import random as rd

# For visualizing the state of the agents using plot
import matplotlib.pyplot as plt

# For checking if a filename already exists, to avoid overwriting files
import os



### Initialize class to create people ###

class agent:
  pass



### Create population ###

def create_agents():

  # Allow the following variable to be accessed outside the function
  global agents_list

  # Initialize list of agents
  agents_list = []

  # Create specified number of agents
  for i in range(n_agents):

    # Create an agent
    agent_ = agent()

    # Assign the agent to a random location
    agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)

    # Set the agent to be initially susceptible
    # 0 = susceptible, 1 = exposed, 2 = infectious, 3 = recovered
    agent_.state = 0

    # Count number of days the agent is exposed
    agent_.days_exposed = 0

    # Count number of days the agent is infectious
    agent_.days_infectious = 0

    # Add the agent to the list
    agents_list.append(agent_)



### Everyone moves to a random location ###

# Scenario 1: Normal day

# Comment scenario not used
# def move():

# # Agent moves to a random location
# for agent_ in agents_list:
#   agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)

## Scenario 2: Somewhat limited social contact for infectious people

def move():

  # Exposed and infectious people do not move
  for agent_ in agents_list:
    if agent_.state != 1 and agent_.state != 2:
      agent_.x, agent_.y = rd.uniform(0, 1), rd.uniform(0, 1)


```

```
76
77 ### Infection happens within a radius (for non-recovered people) ###
78
79 def infect():
80
81     # Allow the following variable to be accessed outside the function
82     global infectious_list
83
84     # Create list of infectious people
85     infectious_list = [agent_ for agent_ in agents_list if agent_.state == 2]
86
87     # Create list of neighbors of infectious
88     for agent_ in infectious_list:
89
90         # Create list of neighbors within the radius specified
91         neighbors = [neighbor for neighbor in agents_list if (neighbor.x - agent_.x)**2 + (neighbor.y
    - agent_.y)**2 <= infectious_radius**2]
92
93         # Remove person himself from list of neighbors
94         neighbors.remove(agent_)
95
96         # Susceptible neighbors become exposed (recovered individuals are immune)
97         for agent_ in neighbors:
98             if agent_.state == 0:
99                 agent_.state = 1
100
101
102
103 ### Monitor progress of infectious people at the start of the day ###
104
105 def monitor():
106
107   # Add count for number of days an exposed person has been exposed
108   for agent_ in agents_list:
109     if agent_.state == 1:
110       agent_.days_exposed += 1
111
112       # Exposed people who reach the end of incubation period become infectious
113       if agent_.days_exposed > incubation_period:
114         agent_.state = 2
115
116   # Add count for number of days an infectious person has been infectious
117   for agent_ in agents_list:
118     if agent_.state == 2:
119       agent_.days_infectious += 1
120
121       # Infectious people who reach the end of infectious period recover
122       if agent_.days_infectious > days_to_recover:
123         agent_.state = 3
124
125
126
127 ### Count members of each compartment ###
128
129 def count():
130
131   # Allow the following variables to be accessed outside the function
132   global S_count, E_count, I_count, R_count
133
134   # Count susceptible
135   S_count = len([agent_ for agent_ in agents_list if agent_.state == 0])
136
137   # Count exposed
138   E_count = len([agent_ for agent_ in agents_list if agent_.state == 1])
139
140   # Count infectious
141   I_count = len([agent_ for agent_ in agents_list if agent_.state == 2])
142
143   # Count recovered
144   R_count = len([agent_ for agent_ in agents_list if agent_.state == 3])
145
146
147
148 ### Visualize population ###
149
150 def visualize():
151
```

```
  # To ensure updated count of compartments
  count()

  # Initialize subplots
  fig, ax = plt.subplots()

  # Assign color per compartment
  agent_color = {0: 'b', 1: 'm', 2: 'r', 3: 'g'}

  # Plot each person with his corresponding color
  for agent_ in agents_list:
    ax.plot(agent_.x, agent_.y, '.', color = agent_color[agent_.state])

  # Dummy plots for legend
  ax.plot([], [], 'bo', label = 'Susceptible: '+ str(S_count))
  ax.plot([], [], 'mo', label = 'Exposed: '+ str(E_count))
  ax.plot([], [], 'ro', label = 'Infectious: '+ str(I_count))
  ax.plot([], [], 'go', label = 'Recovered: '+ str(R_count))

  # Place legend below the figure
  ax.legend(ncol = 2, loc = 'upper center', bbox_to_anchor = (0.5, -0.01))

  # Fix the window: choose largest range for random location choices
  ax.set_xlim([0, 1])
  ax.set_ylim([0, 1])

  # Title
  ax.set_title('Day: ' + str(day_count), loc = 'left')

  # Remove extra tick marks on the axes
  ax.set_xticks([])
  ax.set_yticks([])

  # Prepare format of file name
  filename = 'SEIR'

  # Starting filename count
  i = 1

  # Check if filename already exists; add 1 if it does
  while os.path.exists('{}{:d}.png'.format(filename, i)):
    i += 1

  # Save figure
  plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)



### Graph trend of compartments ###

def trend():

  # To ensure updated count of compartments
  count()

  # Initialize subplots
  fig, ax = plt.subplots()

  # Plot daily count of each compartment
  ax.plot(days, S_list, color = 'blue', label = 'Susceptible')
  ax.plot(days, E_list, color = 'magenta', label = 'Exposed')
  ax.plot(days, I_list, color = 'red', label = 'Infectious')
  ax.plot(days, R_list, color = 'green', label = 'Recovered')

  # Place legend on top of the figure
  ax.legend(ncol = 2, loc = 'upper center', bbox_to_anchor = (0.5, 1.15))

  # Set the axes to meet at (0, 0)
  ax.set_xlim(left = 0)
  ax.set_ylim(bottom = 0)

  # Horizontal axis label
  ax.set_xlabel('Days')

  # Prepare format of file name
  filename = 'SEIR_Trend'

```

```
  # Starting filename count
  i = 1

  # Check if filename already exists; add 1 if it does
  while os.path.exists('{}{:d}.png'.format(filename, i)):
    i += 1

  # Save figure
  plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)



### Simulation ###

# Comment all visualize() for faster simulation
# But replace with count()

## Initial day

# Start day count
day_count = 0

# Initial values
S_initial = 1000
E_initial = 0
I_initial = 1
R_initial = 0

# Initial total population
n_agents = S_initial + E_initial + I_initial + R_initial

# Infectious radius
infectious_radius = 0.025

# Incubation period
incubation_period = 3

# Number of days of infectiousness
days_to_recover = 7

# Initialize list of days
days = []
days.append(0)

# Initialize daily count of susceptible
S_list = []
S_list.append(S_initial)

# Initialize daily count of exposed
E_list = []
E_list.append(E_initial)

# Initialize daily count of infectious
I_list = []
I_list.append(I_initial)

# Initialize daily count of recovered
R_list = []
R_list.append(R_initial)

# Create the population
create_agents()

# Choose a specified number of random people and make them exposed
for agent_ in rd.sample(agents_list, E_initial):
  agent_.state = 1

# From the remaining susceptible, choose a specified number of random people and make them infectious
for agent_ in rd.sample([agent_ for agent_ in agents_list if agent_.state != 1], I_initial):
  agent_.state = 2

# From the remaining susceptible, choose a specified number of random people and make them recovered
for agent_ in rd.sample([agent_ for agent_ in agents_list if agent_.state != 1 and agent_.state != 2],
    R_initial):
  agent_.state = 3

visualize()
```

```
# count()


## Succeeding days

# Number of runs
runs = 100

for i in range(runs):

  # Add day count
  day_count += 1

  # Monitor infectiousness and recovery at the start of the day
  monitor()

  # Let everyone move
  move()

  # Visualize new positions
  visualize()
  # count()

  # Infections start after the movement
  infect()

  # Visualize
  visualize()
  # count()

  # Update lists
  days.append(day_count)
  S_list.append(S_count)
  E_list.append(E_count)
  I_list.append(I_count)
  R_list.append(R_count)

  # Stop simulation once number of infectious becomes 0
  if I_count == 0:
    break

# Plot trend
trend()

####### 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.

[3] Kermack, William O, and Anderson G McKendrick. "A Contribution to the Mathematical Theory of Epidemics." *Proceedings of the Royal Society*, vol. 115, no. 772, 1927, pp. 700-721.

[4] Sayama, Hiroki. "Agent-Environment Interaction." *LibreTexts*, 10 November 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/19.03%3A_Agent-Environment_Interaction.