

AI Disclosure

BMI 500, HW11, Fall 2025

Juyoung Lee

Prompt examples & Response summary

I'm building an agent-based model of pandemic spread for my assignment.
Each agent can be Susceptible (S), Infected (I), or Recovered (R).
The environment should be a 75×75 grid with 100 agents that move randomly.
I want to simulate infection (p) and recovery (q) over 200 steps.
Could you help me design the logical flow and modular code structure, including which functions and variables to define for:
- Initialization of agents
- Movement rules
- Infection and recovery updates
- Recording results at each time step
Give detailed pseudocode or partial implementation guidance.

Here's a **structured, modular plan and pseudocode** to help you implement your **agent-based pandemic model** cleanly and extend it later (e.g., for social distancing).

The logic is organized into four main modules: initialization, movement, infection/recovery updates, and recording.

3. Detailed Pseudocode (Modular)

agent.py

python

Copy code

```
import random

class Agent:
    def __init__(self, agent_id, x, y, state="S"):
        self.id = agent_id
        self.x = x
        self.y = y
        self.state = state # "S", "I", or "R"

    def move(self, grid_size):
        """Randomly move one step in 8 directions or stay put."""
        dx, dy = random.choice([(1,0), (-1,0), (0,1), (0,-1),
                               (1,1), (-1,-1), (1,-1), (-1,1), (0,0)])
        self.x = (self.x + dx) % grid_size
        self.y = (self.y + dy) % grid_size
```

`environment.py`

python

[Copy code](#)

```
import math

class Environment:
    def __init__(self, grid_size, agents):
        self.grid_size = grid_size
        self.agents = agents

    def get_neighbors(self, agent, radius=1):
        """Return agents within infection radius."""
        neighbors = []
        for other in self.agents:
            if other.id != agent.id:
                dist = math.sqrt((agent.x - other.x)**2 + (agent.y - other.y)**2)
                if dist <= radius:
                    neighbors.append(other)
        return neighbors
```

simulation.py

python

 Copy code

```
import random
from environment import Environment

class Simulation:
    def __init__(self, grid_size=75, n_agents=100, p=0.2, q=0.05, steps=200):
        self.grid_size = grid_size
        self.n_agents = n_agents
        self.p = p # infection probability
        self.q = q # recovery probability
        self.steps = steps
        self.agents = []
        self.results = {"S": [], "I": [], "R": []}

    def initialize_agents(self):
        """Randomly place agents and infect one of them."""
        for i in range(self.n_agents):
            x = random.randint(0, self.grid_size-1)
            y = random.randint(0, self.grid_size-1)
            state = "I" if i == 0 else "S" # infect one
            self.agents.append(Agent(i, x, y, state))
        self.env = Environment(self.grid_size, self.agents)

    def update(self):
        """Update all agents: movement, infection, recovery."""
        # Move
        for agent in self.agents:
            agent.move(self.grid_size)
```

```
# Infection and recovery
for agent in self.agents:
    if agent.state == "I":
        # Try infecting neighbors
        for neighbor in self.env.get_neighbors(agent):
            if neighbor.state == "S" and random.random() < self.p:
                neighbor.state = "I"
        # Try recovery
        if random.random() < self.q:
            agent.state = "R"

def record(self):
    """Record number of agents in each state."""
    states = [a.state for a in self.agents]
    self.results["S"].append(states.count("S"))
    self.results["I"].append(states.count("I"))
    self.results["R"].append(states.count("R"))

def run(self):
    """Run full simulation."""
    self.initialize_agents()
    for t in range(self.steps):
        self.update()
        self.record()
```

analysis.py

python

Copy code

```
import matplotlib.pyplot as plt

def plot_results(results):
    plt.plot(results["S"], label="Susceptible")
    plt.plot(results["I"], label="Infected")
    plt.plot(results["R"], label="Recovered")
    plt.xlabel("Time step")
    plt.ylabel("Number of agents")
    plt.legend()
    plt.title("SIR Model (Agent-Based)")
    plt.show()
```

main.py

python

Copy code

```
from simulation import Simulation
from analysis import plot_results

if __name__ == "__main__":
    sim = Simulation(grid_size=75, n_agents=100, p=0.2, q=0.05, steps=200)
    sim.run()
    plot_results(sim.results)
```



Now I need to implement:

- 1) a movement probability (move_prob) to simulate reduced mobility, and
- 2) avoidance behavior (avoid_strength) so that agents move away from infected neighbors.

How can I modify the existing movement logic efficiently without rewriting everything? suggest function-level logic or partial code examples that integrate cleanly with my current ABM.

Excellent — you already have a clean, modular ABM structure, so you can layer **mobility probability** (`move_prob`) and **avoidance behavior** (`avoid_strength`) into your `Agent.move()` and `Environment.get_neighbors()` logic with minimal changes.

Step 1: Add Parameters to `Agent`

Extend your `Agent` class to accept `move_prob` and `avoid_strength` during initialization.

python

 Copy code

```
class Agent:  
    def __init__(self, agent_id, x, y, state="S", move_prob=1.0, avoid_strength=0.0):  
        self.id = agent_id  
        self.x = x  
        self.y = y  
        self.state = state  
        self.move_prob = move_prob  
        self.avoid_strength = avoid_strength
```

These parameters can be globally controlled by the `Simulation` class (so you can easily experiment).

Step 2: Modify `move()` Function

Currently, `move()` randomly selects a direction.

We'll modify it to:

- Skip movement based on `move_prob`.
- Compute a **repulsion vector** away from infected neighbors.

python

 Copy code

```
import random  
  
class Agent:  
    ...  
    def move(self, grid_size, env=None):  
        # 1. Skip movement based on probability  
        if random.random() > self.move_prob:  
            return # stay still  
  
        # 2. Compute avoidance direction (if environment and infected neighbors exist)  
        dx, dy = 0, 0  
        if env and self.avoid_strength > 0:  
            infected_neighbors = [a for a in env.get_neighbors(self, radius=2) if a.state == "I"]  
            for inf in infected_neighbors:  
                # Move away: repulsion = (self - inf)  
                dx += (self.x - inf.x)  
                dy += (self.y - inf.y)  
            # Normalize if nonzero  
            if dx != 0 or dy != 0:  
                dx = int(dx / abs(dx)) if dx != 0 else 0  
                dy = int(dy / abs(dy)) if dy != 0 else 0  
                # Weighted combination: random ↴ avoidance bias
```

Step 3: Pass Environment Reference in Simulation

In your `Simulation.update()` loop, modify how you call `move()`:

```
python Copy code
for agent in self.agents:
    agent.move(self.grid_size, env=self.env)
```

This gives each agent access to the current environment (to sense infected neighbors).

Step 4: Manage Parameters in Simulation Initialization

You can set these parameters globally when initializing agents:

```
python Copy code
def initialize_agents(self):
    for i in range(self.n_agents):
        x = random.randint(0, self.grid_size-1)
        y = random.randint(0, self.grid_size-1)
        state = "I" if i == 0 else "S"
        # Example: 80% movement, moderate avoidance
        self.agents.append(
            Agent(i, x, y, state,
                  move_prob=0.8,
                  avoid_strength=0.5)
        )
    self.env = Environment(self.grid_size, self.agents)
```