This report is on the Cellular Automaton SLIR Model project programmed in Python by Corbin Matamoros. First it will provide a brief summary of cellular automata, followed by a description of cell parameters of this project model. This project models the spread of COVID-19 in a closed, finite population.

**What is a Cellular Automaton (CA)?**

Taking the definition from the class, “A CA is an array of identically programmed automata, or cells, which interact with one another in a neighborhood and have definite state.” With respect to Python, a CA can be represented as a 1D or 2D list of elements/cells – each cell represented as a custom data type – which can interact with each other based on a set of rules. Each cell must have the same rules applied to them at all times, regardless of their state, and the states, rules, interactions, and what cells are considered another cell’s neighbors are all user-defined.

**Project description**

Use a two-dimensional array as a CA to model disease spread. A full day has passed in the simulation if the entire array has been updated once through based on the rules defined by the programmer. Implementation is decided by the programmer based on the following considerations:

1. What does each cell represent? An individual or a location an individual might occupy (implying the individuals can move around in the array)?
2. How should the CA be initialized in terms of number of infected individuals, placement of individuals, etc.?
3. What are the disease parameters? Will the disease be fictitious or based in real life?
4. How will the neighborhood be defined? Moore, von Neumann, or some other?
5. How does the disease spread?
6. What ends the simulation?

Each of these questions will be addressed in the following section.

**Design and Implementation**

Using the Python library NumPy, we create two two-dimensional lists (dubbed lists A and B) of tuples where the tuple is a grouping of four integers. The first tuple value represents the cell’s state (‘0’ for unoccupied, ‘1’ for occupied by a susceptible individual, ‘2’ for occupied by an infected individual in the disease’s latent stage, ‘3’ for occupied by an infectious individual, and ‘4’ for occupied by a recovered individual); the second value is the number of days the individual has been in the latent stage; the third value is the number of days the individual has been in the infectious stage; and the last value is the number of “exposure points” a susceptible individual has. (Note: “exposure points” are awarded to only susceptible individuals based on the number of infectious neighbors. Once the user-defined limit of “exposure points” is met, the susceptible individual transitions to the latent stage.) Each cell represents a location an individual may or may not occupy.

Each 2D list is initialized to zeroed-out tuples. Next, the initial infectious individuals are randomly dispersed throughout list A. Lastly, the rest of the population – assumed susceptible – are randomly dispersed, keeping in mind two or more individuals cannot occupy the same location. The population size and number of initially infectious individuals are user defined in a .json file. (Note: the user cannot set a population size that would not fit in the boundaries of the grid.) The disease parameters, such as latent and infectious period, are defined in the .json file as well.

Users are able to alternate between the von Neumann and Moore neighborhood contact methods by modifying the .json file. The rules for contacts remain the same for both, however. If a spot in list A is unoccupied or occupied by a recovered individual, the program ignores it and moves to the next spot. In the same way, if the spot is occupied by a susceptible individual, we count the number of infectious neighbors and add the result to the fourth tuple value as that individual’s number of “susceptible points.” If the spot is occupied by an individual in the latent stage, its second tuple value is incremented by one. If the spot is occupied by an infectious individual, its third tuple value is incremented by one. For susceptible individuals, if the number in their fourth tuple value is equal to the user-defined “susceptible points” limit, their first tuple value is incremented by one, sending them into the latent stage. For the individuals in the latent or infectious stage, if the number in their second and third tuple values, respectively, are equal to the disease’s latent or infectious periods, their first tuple value is incremented by one, putting the latent-staged individual in the infectious stage and infectious individual in the recovered stage. All changes to the individuals in list A are not made in list A. Instead they are made in list B. This prevents earlier modifications to list A from interfering with later ones.

After all interactions are assessed and recorded to list B, list B is copied back to list A, and list B is zeroed out. The next day begins. The simulation ends when there are no individuals in the latent nor infectious stages.

**Project Assessment**

This program is very expandable. One could add parameters that could represent an individual wearing a mask if they’re infectious, quarantining if they are infectious and displaying symptoms, or even returning to the susceptible stage (to simulate an SLIS). With respect to COVID-19, the CDC estimates 40% of cases are asymptomatic1, so one could modify this program to have 60% of all infectious individuals wearing masks/quarantining.

In more realistic simulations, individuals would be allowed to move around the environment. This was not implemented into this program, however. Moving individuals proved to be a difficult task to complete efficiently by the assignment due date, so that functionality was not included.

In the future, we will add better simulation visuals so people can see the spread of the disease in real time, maybe using Pygame to represent each location in list A as a colorable tile, changing their shades depending on the state of the individual occupying it.