Uppsala University

Modelling Complex Systems

1MA256

Lab3 report

Author:
Kristensen.Samuel
Place of publication: Uppsala

May 2, 2024

1 Problem description

For this lab we are experimenting with the lattice gas cellular automation (LGCA) for susceptible-infected-removed (SIR) type of an epidemic.

The SIR model is one of the simplest compartmental models, the model consists of three compartments:

- S: The number of susceptible individuals.
- I: The number of infectious individual.
- R: The number of removed, recovered or deceased individuals

We will use a constant population meaning that for R the number counted are recovered individuals.

The LGCA or BIO-LGCA model is a discrete model for moving and interacting biological agents, a type of cellular automation. The BIO-LGCA model describes cells and other motile biological agents as point particles moving on a discrete lattice, thereby interacting with nearby particles. For this lab we will be working with a hexagonal grid using offset coordinates, where each node in the grid is a cell containing a population of a random number between 0 and 6 persons.

The BIO-LGCA model is defined by a lattice, a state space, a neighborhood and a rule. The lattice is as we discussed before a grid of hexagonials, the state space describes all possible states of particles or persons in this case within every lattice site and we have that there can be 6 different persons in each lattice site. The neighbourhood is the adjacent or the subset of lattice sites which determines the dynamics of a given site in the lattice. Particles only interact with other particles within their neighborhood. And finally the rule dictates how particles move, proliferate, or die with time. In BIO-LGCA, the rule is divided into two steps, a probabilistic interaction step followed by a deterministic transport step.

The rules for LGCA in this laboration is as follows:

- As a result of an application of the contact operation C, individuals can change their type, meaning that susceptible individuals can become infected, and infected individuals can recover. More precisely, each susceptible individual at a node r, independently of other individuals, can become infected with probability $1-(1-r)^{n_I}$, where n_I is a number of infected individuals at the node r, and r $\in$ [0,1]. Similarly, each infected individual at the node r, independently of other individuals, can recover with probability a, where a $\in$ [0,1].
- As a result of using the randomization operation R, applied at each node independently of the other nodes, a population of individuals residing at a node r is randomly redistributed among edges/channels originating from the node r.
- In the propagation step, governed by the operator P individuals simultaneously move from their nodes to the neighbouring ones through the channels assigned to them in the randomization step. The movement of individuals is purely deterministic in the propagation step.

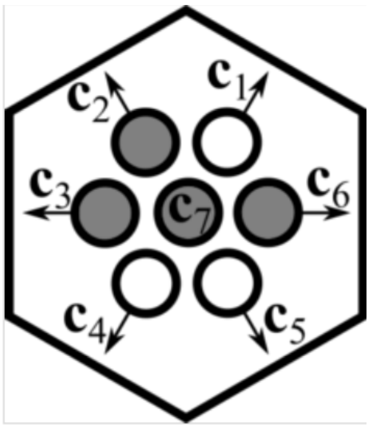


Figure 1: Hexagon with people in it

We can see from figure 1 an example of how the hexagon lattice can look like with people in them and directions for the deterministic transport step.

2 The core model

To start we have to build a model of the basic code to simulate the SIR BIO-LGCA. We do this by building three classes, the Hexagon, Cell and Person classes.

The Hexagon class represents the entire hexagonal grid simulation. It consists of multiple Cell objects arranged in a grid formation. This class manages the propagation of simulation steps, including interaction, movement, and demographic analysis.

The Cell class represents a single hexagonal cell in the simulation grid. Each cell can hold a population of individuals, which are distributed across a velocity_channel. Individuals interact within the cell, potentially infecting each other based on infection probabilities. The population can also undergo random movement within the cell.

The Person class represents individuals within the simulation. Each person has a status indicating whether they are susceptible, infected, or recovered from the disease. These statuses can change based on random probabilities of infection and recovery.

To see whether it follows the SIR model we simulate it with 10^4 nodes for 100 steps using r = 0.3, a = 0.2. Since the starting population for each cell is randomised between 0 and 6 the starting population is a bit different each time (did not really know how to make it constant without screwing up the code) however the mean for each cell is around 3 thus the population is around 30000. The starting infected number of people is also randomized but with a percentage change based on the starting population, here we set that rate to 5%

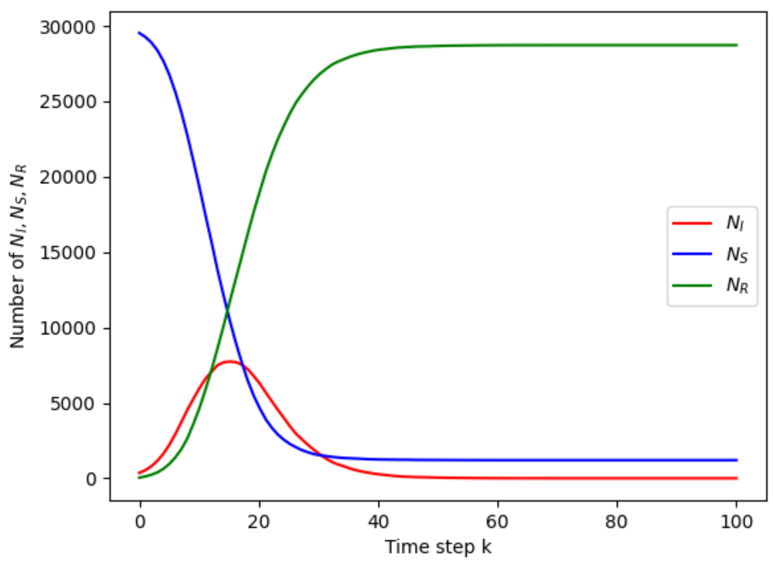


Figure 2: The simulation for the SIR BIO-LGCA

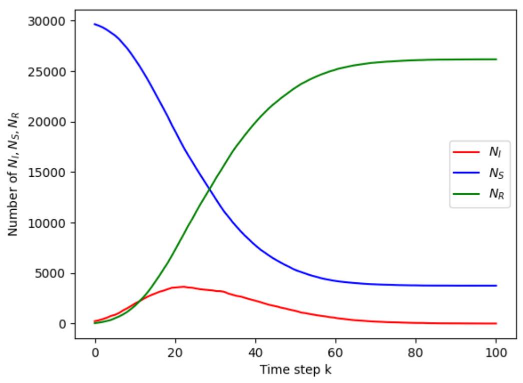
The figure 2 shows the population of infected persons (N_I), susceptible persons (N_S) and recovered persons (N_R) for the simulation. We can see that it follows the SIR model for the case when people cannot be infected again.

2.1 Task 1 a) Rest channels

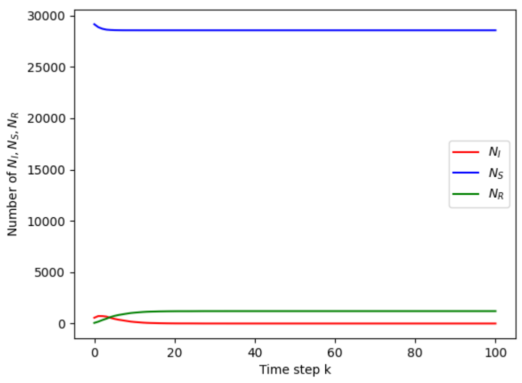
We are now going to add one to several rest channels by implementing it as a parameter in the code. Adding rest channels means that people may stay put and don't necessarily need to move around in the transport phase.

The solution for this was implemented by adding rest channels for each cell with certain indices. If after the random shuffle in the cell a person was assigned a resting index the person was placed in a rest channel. This was done in the populate function where the persons were randomly assigned places in each cell as well in the random movement function which determined the direction and now if the person was moving at all.

The implementation is simulated with 4 and 6 resting channels for the same initial data as before (10^4 nodes, 100 steps, r = 0.3 and a = 0.2). And also initial data for the starting population done in the same way as the first simulation.



(a) Simulation with 4 rest channels



(b) Simulation with 6 rest channels

Figure 3: The simulation for 4 and 6 rest channels

The figure 3 depicts the simulation with 4 resting channels (the left) and 6 resting channels (the right) and is showing the population of infected persons (N_I), susceptible persons (N_S) and recovered persons (N_R) for the simulations.

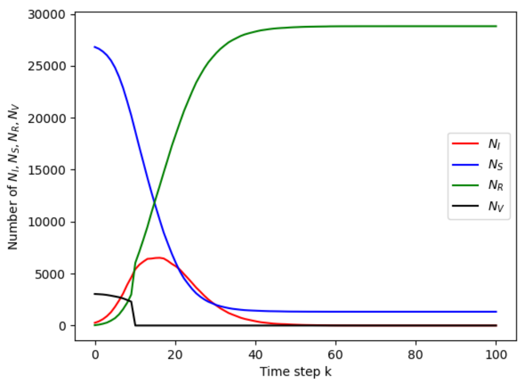
The figure with 4 resting channels gives a longer infection than the regular simulation. The reason why that is is because since fewer are moving around the decease is not reaching everybody as fast as before. We also see that more people are susceptible in the end meaning that not as many became infected as before. The figure with 6 resting channels shows that the decease is spreading a bit in the beginning before dying out very fast. This makes sense, when no one is moving around the infection cannot spread and only the people in the same cell can catch it but they will recover quite fast. This is total isolation and the majority of the people does not get affected at all.

2.2 Task 1 b) vaccinated

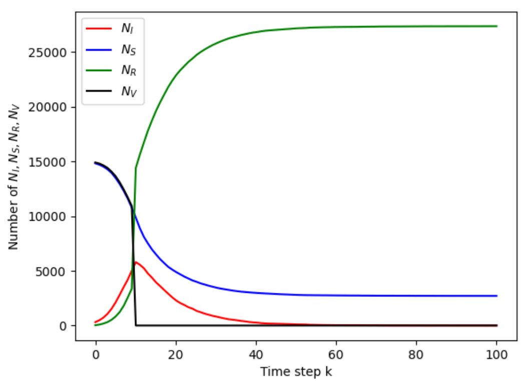
We will now add the fourth population - vaccinated, N_v. A vaccinated person stays susceptible, and behaves susceptible for k time steps (k written in as a parameter). If during these time steps the vaccinated individuals have not becomes infected, than they become recovered after k steps.

The implementation for this is fairly straight forward. Simply add another status for each person, randomly populate the cells with a percentage of vaccinated people and run the simulation with the condition that if the vaccinated person is not infected during k steps they then become recovered.

We simulate this with the same initial conditions as in the core model2 with the addition that k = 10 and for two scenarios, that 10% of the population are vaccinated and 50% of the population are vaccinated.



(a) Simulation with 10% vaccinated



(b) Simulation with 50% vaccinated

Figure 4: The simulation for 10% and 50% of the population vaccinated

Figure 4 shows the two simulations with the left image being that 10% of the population vaccinated at the start and the right image is with 50% of the population vaccinated at the start.

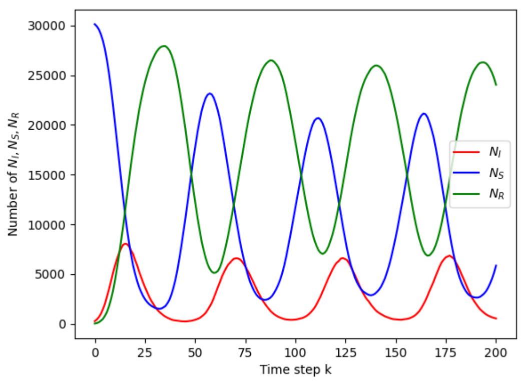
The case when about 50% of the population being vaccinated is quite interesting, we see at the start that the susceptible and vaccinated people follow the exact same curve. They are both about the same size and both act the same way since they have the same percentage chance of getting infected. Then after 10 days the vaccinated people that did not get infected all become recovered, the vaccinated drops down to 0 and the recovered jumps up a lot. If you look closely at the case with 10% we see the same thing but much smaller.

The amount of vaccinated is bound to go down to 0 since it is just initially vaccinated people, would be interesting to see a continuation of vaccinating a percentage of people during the time.

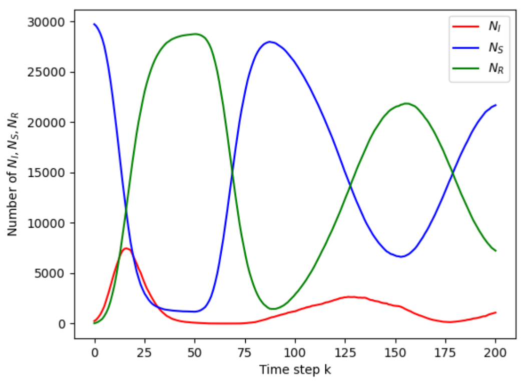
2.3 Task 1 c) Reinfection

We will now allow the recovered individuals to be reinfected: they stay recovered for some l time steps, and then become susceptible. We will take l > k since usually the time that it takes the immunity to kick in after vaccination is much shorter than the time the vaccine/acquired immunity keeps you immune.

We add this in the update function for the class Person, once a person becomes recovered they stay recovered for l steps, after those steps are done they become susceptible again. We again test this with the initial configuration from section 2 and with the added conditions that they get susceptible again after l=30 and l= 50 steps. We also run for 200 steps to see the longer affects.



(a) Simulation with 30 days immune



(b) Simulation with 50 days immune

Figure 5: The simulation for 30 and 50 days immune

The left figure in 5 shows how the population behaves after becoming susceptible again after 30 days, and the right figure shows the same thing but fr 50 days. We observe that the infection never dies down (at least for these parameters), once the initial "wave" of infections hit a lot of people become recovered but then after the l days have gone they again become susceptible and if there are still people that are infected a second wave comes and so on.

If either enough time goes from being recovered to susceptible or maybe the r-value for the infection is low enough it would die but at least not for this case. We can see a tendency to this for the case with 50 days but some people are still infected and the infection is alive.

Note: We have a lot of variables that we can alter; r, a, rest channels, k, l, percentage infected, percentage vaccinated meaning that to see how they all effect each other is really difficult. I focused on seeing how one thing affected the core model but it would be interesting changing all parameters to see the full effect.

3 Task 2

Did not really have time and could not really figure out stuff for the circle barrier.

4 Code

```
from hexagon import *
import random
import matplotlib.pyplot as plt
import numpy as np

class Person:
    def __init__(self, status, k, l):
        """The status is if the person is S,I,R (or vaccinated), k for time
            untill vaccination
          gets recovered and l for time from recovered to susceptible"""
        self.status = status
        self.k = k
        self.l = l

    def update(self, infection_prob, recover_prob,l):
        """Uppdates the status based on infection and recovery probability, also
            the rules for k and l"""
        if self.status == "susceptible":
            if random.random() < infection_prob:
                self.status = "infected"
        elif self.status == "infected":
            if random.random() < recover_prob:
                self.status = "recovered"
                self.l = l
        elif self.status == "vaccinated":
            if random.random() < infection_prob:
                self.status = "infected"
            elif self.k == 0:
                self.status = "recovered"
                self.l = l
            else:
                self.k -=1
        elif self.status == "recovered":
            if self.l == 0:
                self.status = "susceptible"
            else:
                self.l -= 1

    def is_infected(self):
        """Returns True if person is infected"""
        return True if self.status == "infected" else False

class Cell:
    def __init__(self, population:int, infection_prob, recover_prob,
        rest_channels):
        """Get number of population in cell, sets up velocity channels and rest
            channels,
        also the infection probability and recover probability"""
        self.population = population
        self.rest_channels = rest_channels
        self.velocity_channel = [None] * 6
        self.temp_velocity_channel = [None] * 6
        self.rest_channel = [None] * self.rest_channels
        self.rest_indices = random.sample(range(6),self.rest_channels)

        self.infection_prob = infection_prob
        self.recover_prob = recover_prob


    def populate(self, percent_infected,k, percent_vaccinated, l):
        """Populates the cell with people randomly infected, vaccinated and
            susceptible based on percentage"""
        list1 = [x for x in range(6)]
        rest_index = 0
        for person in range(0,self.population):
            placed = False
            rand = random.random()
            if rand < percent_infected:
                status = "infected"
            elif rand < percent_infected + percent_vaccinated:
                status = "vaccinated"
            else:
                status = "susceptible"

            while not placed:
                location = random.sample(list1, 1)[0]
                list1.remove(location)
                if location not in self.rest_indices:
                    self.velocity_channel[location] = Person(status,k, l)
                    placed = True

                elif location in self.rest_indices:
                    location = rest_index
                    self.rest_channel[location] = Person(status,k, l)
                    rest_index += 1
                    placed = True

    def contact_operation(self,l):
        """The contact operation C, calculates the probability of getting
            infected at each cell"""
        total_population = self.velocity_channel+self.rest_channel
        num_infected = sum(1 for person in total_population if person and
            person.is_infected())
        probability = 1 - (1 - self.infection_prob) ** num_infected
        for person in total_population:
            if person:
                person.update(probability, self.recover_prob,l)

    def randomization_operator(self):
        """The randomization operator R, randomly moves people in the cells"""
        total_population = self.velocity_channel + self.rest_channel
        #random.shuffle(total_population)
        total_population = [person for person in total_population if person is
            not None]

        self.rest_channel = [None]*self.rest_channels
        self.velocity_channel = [None]* 6

        list1 = [x for x in range(len(total_population))]
        rest_index = 0
        for person in total_population:
            placed = False
            while not placed:
                location = random.sample(list1, 1)[0]
                list1.remove(location)

                if location in self.rest_indices:
                    location = rest_index
                    self.rest_channel[rest_index] = person
                    rest_index +=1
                    placed = True

                else:
                    empty_slots = [index for index, val in
                        enumerate(self.velocity_channel) if val is None]
                    if empty_slots:
                        velocity_index = random.choice(empty_slots)
                        self.velocity_channel[velocity_index] = person
                        placed = True


    def receive_new_person(self, person:Person, position):
        """Helper function to recieve a new person"""
        self.temp_velocity_channel[position] = person

    def svc(self):
        """Switches temp velocity with velocity channel"""
        self.velocity_channel = self.temp_velocity_channel
        self.temp_velocity_channel = [None] * 6

    def remove_person(self, person):
        """Removes a person"""
        for i in range(len(self.velocity_channel)):
            if self.velocity_channel[i] == person:
                self.velocity_channel[i] = None
        for i in range(len(self.rest_channel)):
            if self.rest_channel[i] == person:
                self.rest_channel[i] = None

    def calculate_population(self):
        """Calculates the whole population"""
        self.population = sum(1 for person in self.velocity_channel if person) +
            sum(1 for person in self.rest_channel if person)

    def demographic(self):
        """Calculates the demographics for each state"""
        susceptible = sum(1 for person in self.velocity_channel if person and
            person.status == "susceptible") + sum(1 for person in
            self.rest_channel if person and person.status == "susceptible")
        infected = sum(1 for person in self.velocity_channel if person and
            person.status == "infected") + sum(1 for person in self.rest_channel
            if person and person.status == "infected")
        recovered = sum(1 for person in self.velocity_channel if person and
            person.status == "recovered") + sum(1 for person in
            self.rest_channel if person and person.status == "recovered")
        vaccinated = sum(1 for person in self.velocity_channel if person and
            person.status == "vaccinated") + sum(1 for person in
            self.rest_channel if person and person.status == "vaccinated")
        return susceptible, infected, recovered, vaccinated

class Hexagon:
    def __init__(self, right, bottom, r_val, a, rest_channels,k,l):
        """Creates the hexagon grid, finds max column, row in offset
            coordinates"""
        self.grid = {}
        for q in range(0, right):
            q_offset = math.floor(q/2.0)
            for r in range(0-q_offset, bottom - q_offset):
                population = random.randint(0, 6)
                self.grid[Hex(q,r,-q-r)] = Cell(population, r_val, a,
                    rest_channels)

        self.col = max(qoffset_from_cube(ODD, hex_key).col for hex_key in
            self.grid) + 1
        self.row = max(qoffset_from_cube(ODD, hex_key).row for hex_key in
            self.grid) + 1
        self.rest_channels = rest_channels
        self.k = k
        self.l = l

    def run_step(self):
        """Next step in the simulation"""
        for hex_key, cell_value in self.grid.items():
            cell_value.contact_operation(self.l)
            cell_value.randomization_operator()
            for i, person in enumerate(cell_value.velocity_channel):
                if person:
                    self.switch_person(hex_key, person, i)

        for cell_value in self.grid.values():
            cell_value.svc()
    def switch_person(self, curr_hex, person, direction):
        """Switches the person with the neighbouring cell"""
        neighbouring_hexagon = hex_neighbor(curr_hex, direction)
        neighbour_offset_cord = qoffset_from_cube(ODD, neighbouring_hexagon)
        neighbouring_hexagon = qoffset_wraparound_to_hex(neighbour_offset_cord,
            self.col, self.row)
        neighbour_cell = self.grid[neighbouring_hexagon]
        neighbour_cell.receive_new_person(person, direction)

    def total_deomgraphic(self):
        """Calculates the whole demographic for the hexagon grid"""
        susceptible, infected, recovered, vaccinated = 0, 0, 0, 0
        for cell_value in self.grid.values():
            sus, inf, rec, vac = cell_value.demographic()
            susceptible += sus
            infected += inf
            recovered += rec
            vaccinated += vac
        return susceptible, infected, recovered, vaccinated

    def calculate_population(self):
        """Calculates the total population"""
        return sum(cell_value.population for cell_value in self.grid.values())

    def populate_grid(self, percent_infected,percent_vaccinated):
        """Populates the whole grid"""
        for cell in self.grid.values():
            cell.populate(percent_infected,self.k,percent_vaccinated, self.l)

def run_simulation(steps, r, a, rest_channels,k,l, percent_infected,
    percent_vaccinated):
    """Runs the simulation with the parameters given"""
    grid = Hexagon(100,100,r,a, rest_channels,k,l)
    grid.populate_grid(percent_infected, percent_vaccinated)
    print("Initial population:", grid.calculate_population())

    susceptible = np.empty(steps)
    infected = np.empty(steps)
    recovered = np.empty(steps)
    vaccinated = np.empty(steps)

    susceptible[0], infected[0], recovered[0], vaccinated[0] =
        grid.total_deomgraphic()
    for i in range(steps):
        grid.run_step()
        susceptible[i], infected[i], recovered[i], vaccinated[i] =
            grid.total_deomgraphic()

    return susceptible, infected, recovered, vaccinated

def plot_fig(steps,susceptible,infected,recovered, vaccinated):
    """Plots the figure"""
    x = np.linspace(0, steps, steps)
    plt.plot(x, infected, linewidth=1.5, color='red')
    plt.plot(x, susceptible, linewidth=1.5, color='blue')
    plt.plot(x, recovered, linewidth=1.5, color='green')
    plt.plot(x, vaccinated, linewidth=1.5, color='black')
    plt.legend(["$N_{I}$", "$N_{S}$", "$N_{R}$", "$N_{V}$"])
    plt.xlabel("Time step k")
    plt.ylabel("Number of $N_{I},N_{S},N_{R}, N_{V}$")
    plt.show()

if __name__ == "__main__":
    """Set parameters here"""
    steps = 100
    r = 0.3
    a = 0.2
    rest_channels = 0
    k = 10
    l = 50
    percent_infected = 0.005
    percent_vaccinated = 0.1

    susceptible, infected, recovered, vaccinated =
        run_simulation(steps,r,a,rest_channels,k,l,percent_infected,
        percent_vaccinated)

    plot_fig(steps,susceptible,infected,recovered, vaccinated)
```