Analysis - Simulating disease spread through Cellular Automata and the SIR model

Zebedee Marsh

November 27, 2020

Contents

1 Introduction

Since 1999, there have been 11 major disease outbreaks (Daily Sundial, 2020) across the globe, eight of which involved thousands of cases. To effectively control an outbreak and ultimately stop it in its tracks, it's current and past states must be analysed so it's spread can be predicted, which is essential in influencing how governments should deal with such a threat.

This is currently being seen in how the UK government is dealing with COVID-19, where a balance between a total lockdown and total freedom needs to be struck to control how the virus spreads. This balance is essential in allowing businesses to continue operating, keeping the economy afloat and allowing people their freedoms to do what they want.

My project aims to provide a starting point for where a disease outbreak can be modelled either by cellular automata or the SIR model, which can be used in an educational environment to teach people about the spread of diseases. A user will be able to input custom parameters according to the disease they are trying to simulate and be able to introduce special events such as a partial or permanent lockdown, to be able to see how the disease spread could change. A user should also be able to compare the two different ways of simulating a disease.

Source: https://sundial.csun.edu/156361/news/a-timeline-of-outbreaks-from-2000-to-present/

2 Potential user

Mr Bliss - Biology teacher at Abingdon School As this model provides a starting point for modelling disease outbreak, it's most effective use would be in an education environment, where it is used to teach people about how disease can spread and how changes in parameters such as the infection rate, recovery rate, time of infection and immunity can affect how effectively a disease is transmitted throughout a cohort.

3 How the problem was researched

To first understand how I could make this project, I had to research the two main methods of modelling disease spread, the SIR model and Cellular Automata. Once I got the basic idea of how each model worked, I further researched how I could implement these in Python, as well as looking at how to use some Python libraries to help me.

- Find out how SIR and Cellular Automata work
- Research how to implement SIR and Cellular Automata in code
- Learn how to use the Matplotlib library to draw graphs for my disease models
- Learn how to use Pandas to export data to an external file
- Learn how to use Tkinter to create a GUI interface and multiple pages
- Learn how to create, use and modify an SQL database using Sqlite3

4 Research of existing solutions

TBD

5 Interview with target user

Q1. How could such a tool like this be used in the classroom?
A1. As a comparator of diseases; To be able to decide the best method to restrict a disease; To possibly predict the start of a second spike
E1. I will try to implement a feature where the parameters of two diseases could be entered, and a graph could be calculated and shown for both, which would allow a user to compare the spread of the two diseases. I will also add features that a government may try and implement to restrict a disease. This could include a lockdown feature or a vaccine introduction which would allow a user to see how the spread of a disease could change.

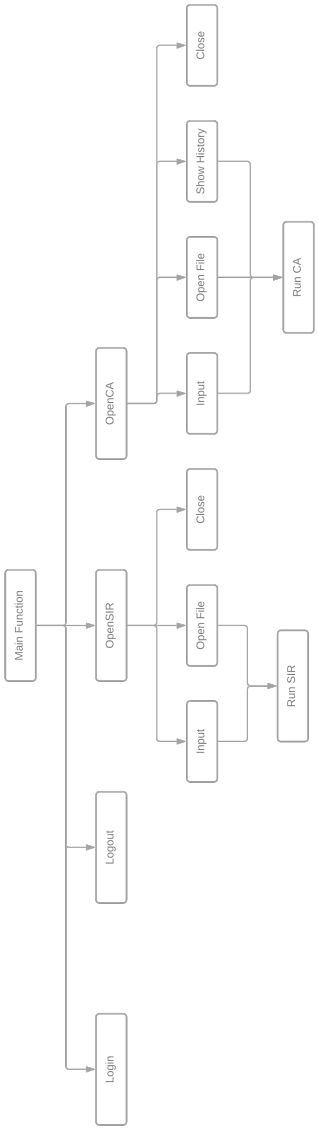
Q1. How could such a tool like this be used in the classroom?
A1. As a comparator of diseases; To be able to decide the best method to restrict a disease; To possibly predict the start of a second spike
E1. I will try to implement a feature where the parameters of two diseases could be entered, and a graph could be calculated and shown for both, which would allow a user to compare the spread of the two diseases. I will also add features that a government may try and implement to restrict a disease. This could include a lockdown feature or a vaccine introduction which would allow a user to see how the spread of a disease could change.

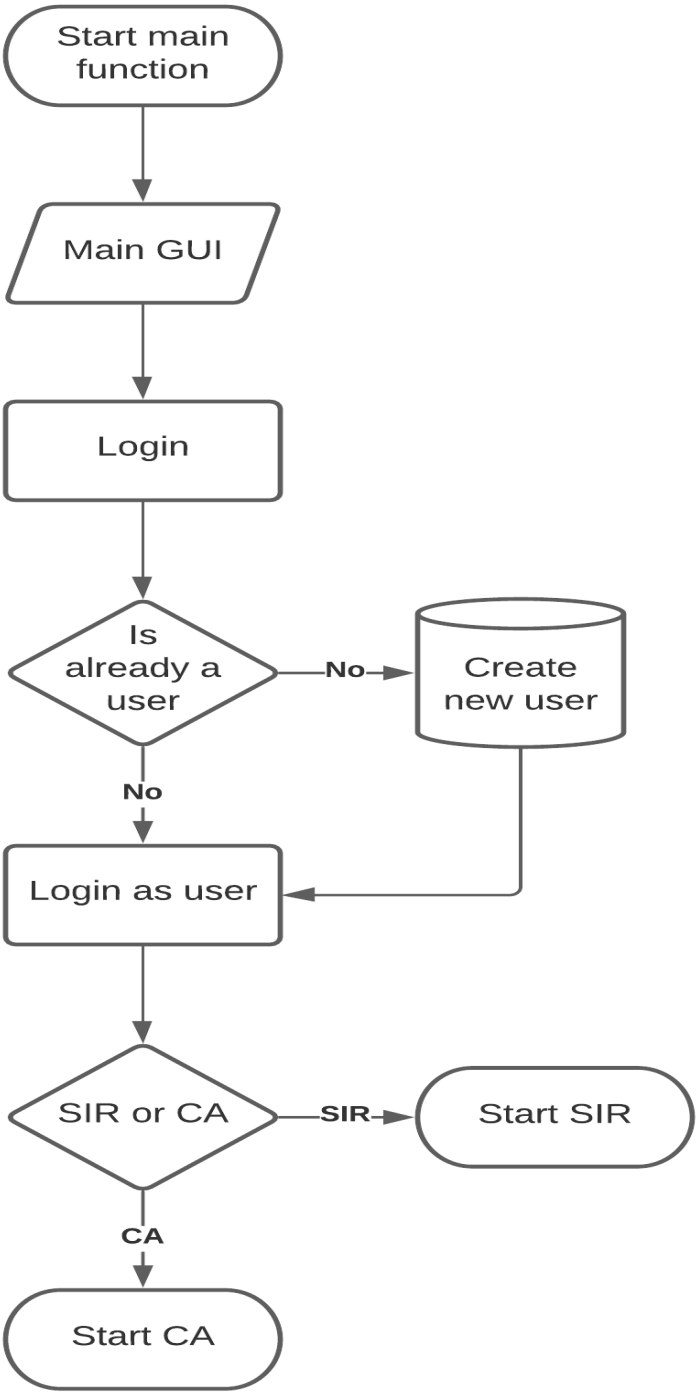
6 Prototyping of code

TBD

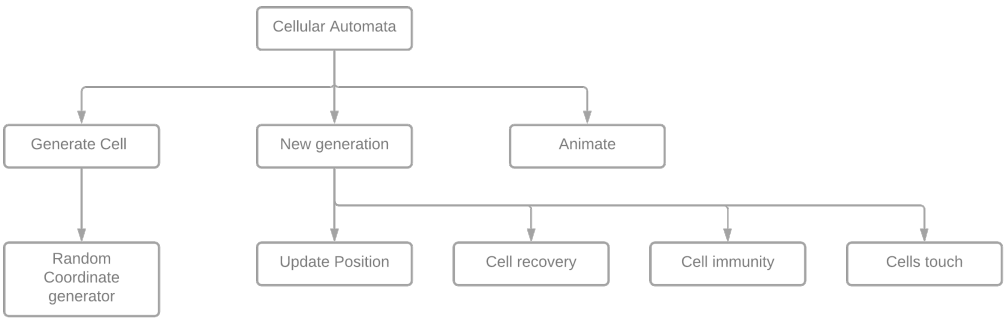
7 Documented design

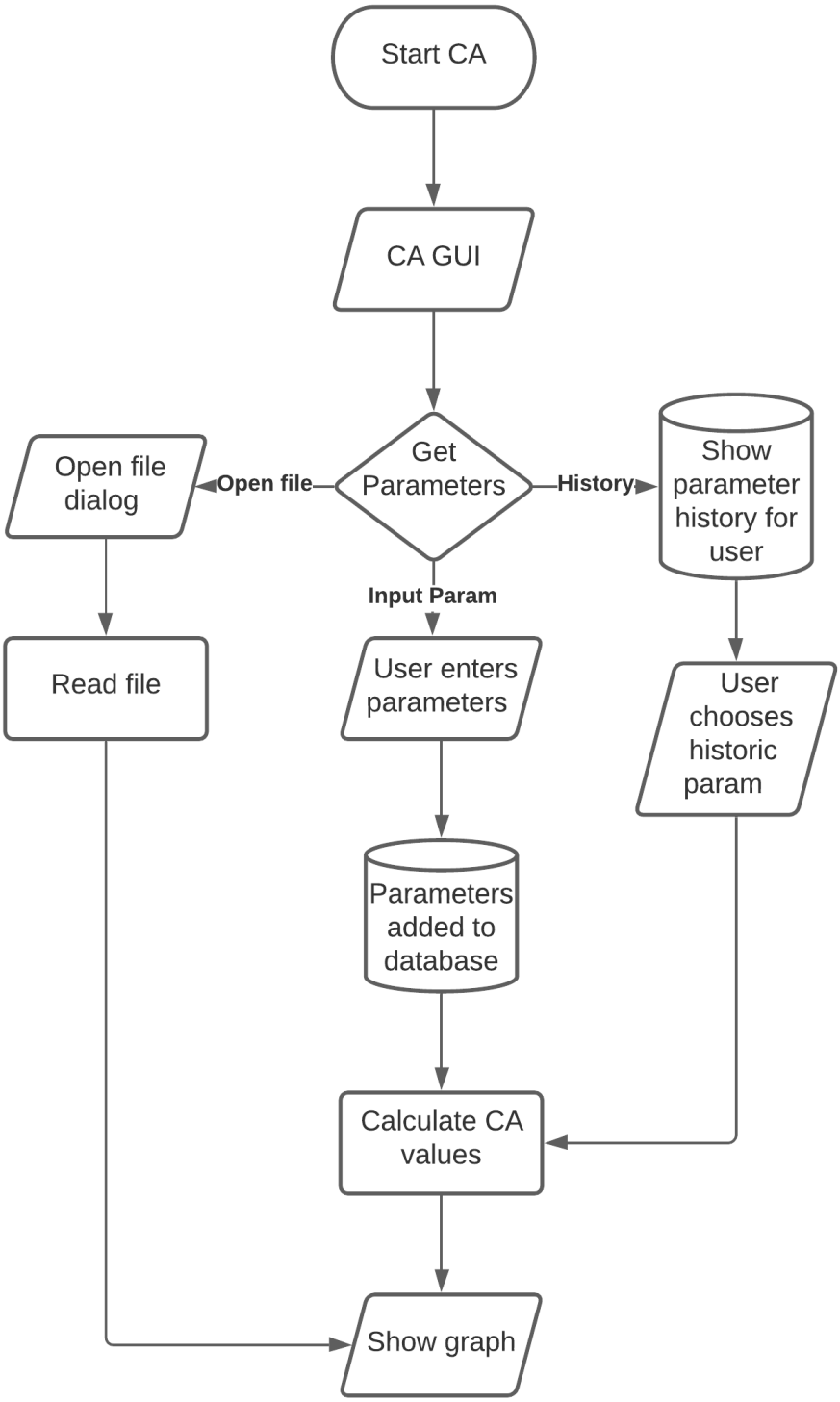
7.1 Main function



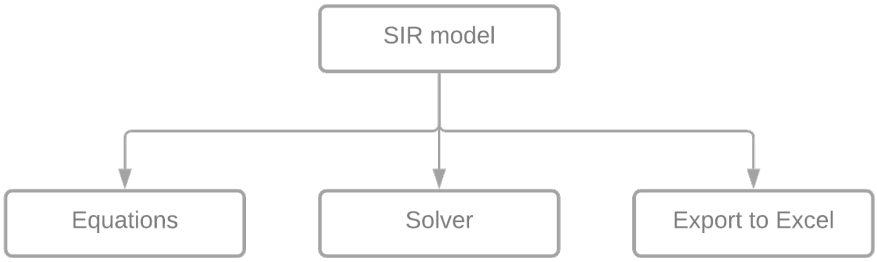

Start main function
Main GUI
Login
Is already a user
No
Create new user
No
Login as user
SIR or CA
SIR
Start SIR
CA
Start CA

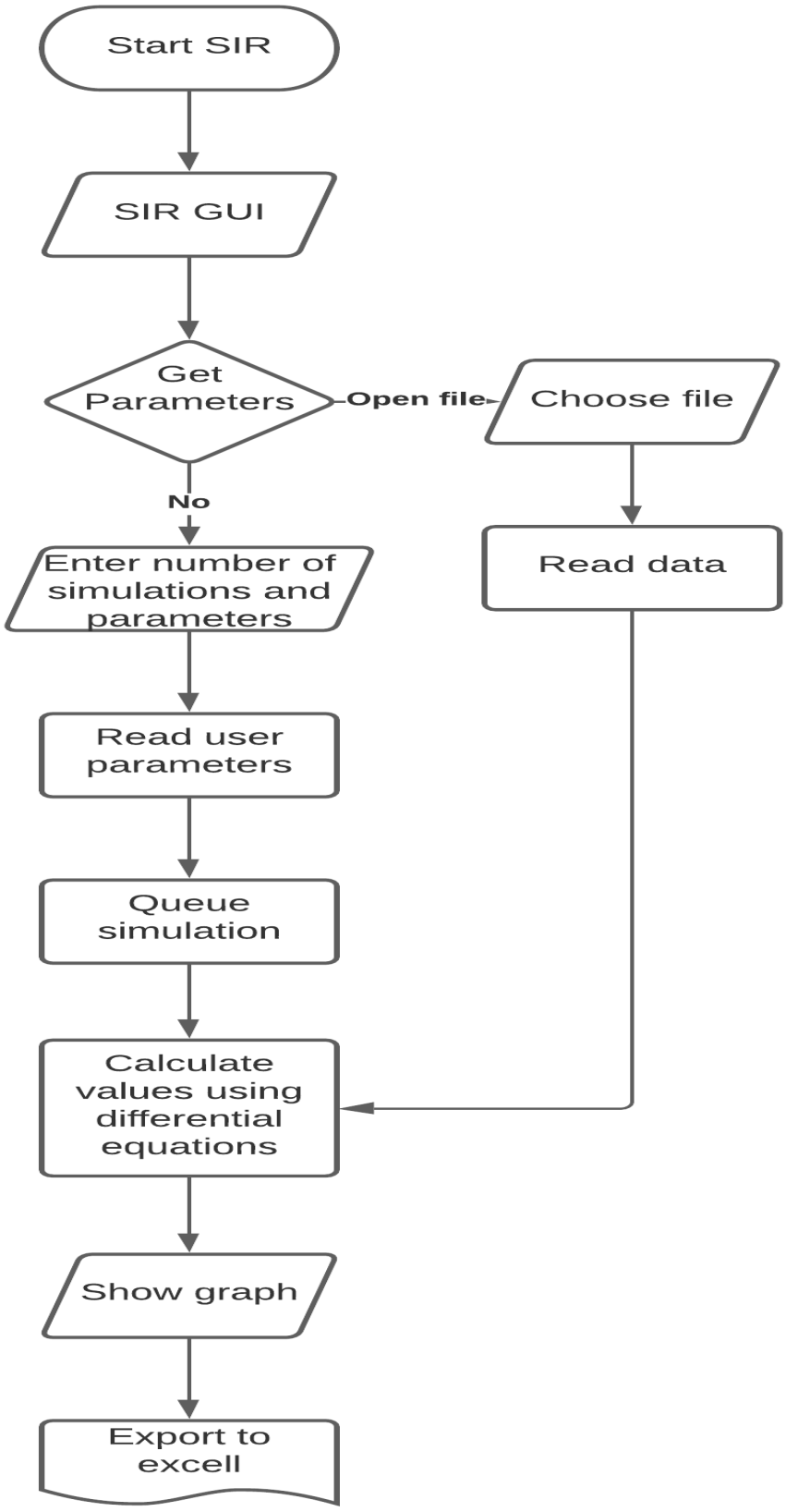
7.2 Cellular automata




Start CA
CA GUI
Get Parameters
Open file
History
Input Param
Open file dialog
Read file
Show parameter history for user
User chooses historic param
User enters parameters
Parameters added to database
Calculate CA values
Show graph

7.3 SIR Model




Start SIR
SIR GUI
Get Parameters
Open file
Choose file
No
Enter number of simulations and parameters
Read data
Read user parameters
Queue simulation
Calculate values using differential equations
Show graph
Export to excell

8 The code

8.1 Main Function

```
# pylint: disable=unused-variable
import tkinter as tk
from tkinter import ttk
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import time
from matplotlib.animation import FuncAnimation
import matplotlib
import csv
import json
from tkinter import filedialog

import sub_SIR_model as my_sir
import sub_CA_model as my_ca
import sub_sql_functions as my_sql

current_user = "None"


class gui_Main_Window:

    def __init__(self, master):
        self.master = master
        self.frame = ttk.Frame(master, padding=5)

        # self.lbl_name = ttk.Label(self.frame, text='This is the main page')
        self.e_user = ttk.Entry(self.frame)
        self.btn_login = ttk.Button(self.frame, text='Login', command=self.login)
        self.btn_SIR = ttk.Button(self.frame, text='SIR model', command=self.openSIR,
            state=tk.DISABLED)
        self.btn_CA = ttk.Button(self.frame, text='Cellular Automata',
            command=self.openCA, state=tk.DISABLED)

        self.frame.grid(row=0, column=0, sticky='nsew')

        # self.lbl_name.grid(column=1, row=0, columnspan=3, sticky='n')
        self.e_user.grid(column=1, row=0, sticky='n')
        self.btn_login.grid(column=2, row=0, sticky='n')
        self.btn_SIR.grid(column=1, row=1, columnspan=2, sticky='s')
        self.btn_CA.grid(column=1, row=2, columnspan=2, sticky='s')

        self.master.columnconfigure(0, weight=1)
        self.master.rowconfigure(0, weight=1)

        self.frame.columnconfigure(1, weight=1)
        self.frame.rowconfigure(1, weight=1)

    def openSIR(self):
        # self.master.destroy()
```

```
        root2 = tk.Tk()
        root2.title('SIR Model')
        # root2.geometry('1400x900')
        new_window = gui_First_SIR_Window(root2)
        root2.mainloop()

    def openCA(self):
        # self.master.destroy()
        root2 = tk.Tk()
        root2.title('CA Model')
        new_window = gui_First_CA_Window(root2)
        root2.mainloop()

    def login(self):
        username = str(self.e_user.get())
        my_sql.enter_username(username)
        global current_user
        current_user = username

        self.btn_SIR['state'] = tk.NORMAL
        self.btn_CA['state'] = tk.NORMAL
        self.e_user.destroy()
        self.btn_login.destroy()
        self.lbl_name = ttk.Label(self.frame, text=f'Hello {current_user}')
        self.lbl_name.grid(column=1, row=0)
        self.btn_logout = ttk.Button(self.frame, text='logout', command=self.logout)
        self.btn_logout.grid(column=2, row=0)

    def logout(self):
        print('logout')
        self.e_user = ttk.Entry(self.frame)
        self.btn_login = ttk.Button(self.frame, text='Login', command=self.login)
        self.e_user.grid(column=1, row=0, sticky='n')
        self.btn_login.grid(column=2, row=0, sticky='n')
        self.btn_SIR['state'] = tk.DISABLED
        self.btn_CA['state'] = tk.DISABLED


# ----------------------------------------

class gui_First_SIR_Window:
    """Class asking user to enter how many graphs they want"""

    def __init__(self, master):
        self.master = master
        self.frame = ttk.Frame(master, padding=5)

        self.lbl_name = ttk.Label(self.frame, text='This is the SIR model')

        self.lbl_num_sim = ttk.Label(self.frame, text='Number of simulations')
        self.e_num_sim = ttk.Entry(self.frame)
        # self.num_sim.insert(0, 1)
        self.btn_open_file = ttk.Button(self.frame, text='Load File',
            command=self.open_file)
```

```
        self.btn_input_param = ttk.Button(self.frame, text='Enter Parameters',
            command=self.input)
        self.btn_close = ttk.Button(self.frame, text='Close', command=self.close)

        self.frame.grid(row=0, column=0, sticky='nsew')

        self.lbl_name.grid(column=0, row=0, columnspan=2, sticky='n')
        self.lbl_num_sim.grid(column=0, row=1)
        self.e_num_sim.grid(column=1, row=1)
        # self.num_sim.grid(column=0, row=1)
        self.btn_open_file.grid(column=0, row=2)
        self.btn_input_param.grid(column=1, row=2)
        self.btn_close.grid(column=0, row=3, sticky='s')

        self.master.columnconfigure(0, weight=1)
        self.master.rowconfigure(0, weight=1)

        self.frame.columnconfigure(1, weight=1)
        self.frame.rowconfigure(1, weight=1)

    def close(self):
        self.master.destroy()

    def open_file(self):
        filename = filedialog.askopenfilename()
        df = pd.read_excel(filename)
        # print(df)

    def input(self):
        self.number_of_simulations = int(self.e_num_sim.get())
        print(self.number_of_simulations)

        # self.number_of_simulations = 1
        root3 = tk.Tk()
        root3.title('Input Parameters')
        input_window = gui_SIR_Param(root3, self.number_of_simulations)
        root3.mainloop()


class gui_SIR_Param:
    """Class for entering SIR parameters"""

    def __init__(self, master, num_sim):
        self.param_list = [[], [], [], [], []]
        self.counter = 0

        self.number_of_simulations = num_sim
        self.master = master
        self.frame = ttk.Frame(master, padding=5)
        self.frame.grid(row=0, column=0, sticky='nsew')

        # label variables
        self.lbl_name = ttk.Label(self.frame, text='Enter parameters for SIR model')
        self.s_name = ttk.Label(self.frame, text='Susceptible')
        self.i_name = ttk.Label(self.frame, text='Infected')
```

```
        self.r_name = ttk.Label(self.frame, text='Recovered')
        self.tr_name = ttk.Label(self.frame, text='Transmission rate')
        self.re_name = ttk.Label(self.frame, text='Recovery rate')
        self.btn_enter = ttk.Button(self.frame, text='Enter', command=self.enter_param)

        # button variables
        self.e_s = ttk.Entry(self.frame)
        self.e_i = ttk.Entry(self.frame)
        self.e_r = ttk.Entry(self.frame)
        self.e_tr = ttk.Entry(self.frame)
        self.e_re = ttk.Entry(self.frame)

        # gridding label variables
        self.lbl_name.grid(column=0, row=0, columnspan=2, sticky='n')
        self.s_name.grid(column=0, row=1, sticky='w')
        self.i_name.grid(column=0, row=2, sticky='w')
        self.r_name.grid(column=0, row=3, sticky='w')
        self.tr_name.grid(column=0, row=4, sticky='w')
        self.re_name.grid(column=0, row=5, sticky='w')

        self.e_s.grid(column=1, row=1, sticky='w')
        self.e_i.grid(column=1, row=2, sticky='w')
        self.e_r.grid(column=1, row=3, sticky='w')
        self.e_tr.grid(column=1, row=4, sticky='w')
        self.e_re.grid(column=1, row=5, sticky='w')

        self.btn_enter.grid(column=0, row=6, columnspan=2, sticky='s')

        self.master.columnconfigure(0, weight=1)
        self.master.rowconfigure(0, weight=1)

        # weight elements for resizing
        self.frame.columnconfigure(0, weight=1)
        self.frame.rowconfigure(0, weight=1)
        self.frame.rowconfigure(1, weight=1)
        self.frame.rowconfigure(2, weight=1)
        self.frame.rowconfigure(3, weight=1)
        self.frame.rowconfigure(4, weight=1)
        self.frame.rowconfigure(5, weight=1)
        self.frame.rowconfigure(6, weight=1)

    def enter_param(self):
        self.counter += 1

        self.param_list[0].append(float(self.e_s.get()))
        self.param_list[1].append(float(self.e_i.get()))
        self.param_list[2].append(float(self.e_r.get()))
        self.param_list[3].append(float(self.e_tr.get()))
        self.param_list[4].append(float(self.e_re.get()))

        self.e_s.delete(0, 'end')
        self.e_i.delete(0, 'end')
        self.e_r.delete(0, 'end')
        self.e_tr.delete(0, 'end')
        self.e_re.delete(0, 'end')
```

```
        if self.counter == self.number_of_simulations:
            self.submit_param()
            self.counter = 0

    def submit_param(self):
        # print(self.param_list)
        self.master.destroy()
        queue = my_sir.QueueSimulation(self.number_of_simulations, self.param_list[0],
            self.param_list[1],
                                       self.param_list[2],
                                       self.param_list[3], self.param_list[4],
                                       100, current_user)

        queue.run_simulation()


# ----------------------------------------

class gui_First_CA_Window:

    def __init__(self, master):
        self.master = master
        self.frame = ttk.Frame(master, padding=5)

        self.lbl_name = ttk.Label(self.frame, text='This is the Cellular Automata
            model')
        # self.num_sim = ttk.Entry(self.frame)
        # self.num_sim.insert(0, 1)
        self.btn_open_file = ttk.Button(self.frame, text='Load File',
            command=self.open_file)
        self.btn_input_param = ttk.Button(self.frame, text='Enter Parameters',
            command=self.input)
        self.btn_show_history = ttk.Button(self.frame, text='History',
            command=self.show_history)
        self.btn_close = ttk.Button(self.frame, text='Close', command=self.close)

        self.frame.grid(row=0, column=0, sticky='nsew')

        self.lbl_name.grid(column=0, row=0, columnspan=2, sticky='n')
        # self.num_sim.grid(column=0, row=1)
        self.btn_open_file.grid(column=0, row=1)
        self.btn_input_param.grid(column=1, row=1)
        self.btn_show_history.grid(column=0, row=2)
        self.btn_close.grid(column=1, row=2, sticky='s')

        self.master.columnconfigure(0, weight=1)
        self.master.rowconfigure(0, weight=1)

        self.frame.columnconfigure(1, weight=1)
        self.frame.rowconfigure(1, weight=1)

    def open_file(self):
        # enter file stuff
        print("Opening file dialog")
```

```
        file = filedialog.askopenfile(mode="r")
        lines = [line.rstrip('\n') for line in file]
        file.close()

        if lines[-1] == "":
            del lines[-1]

        # result = [json.loads(item) for item in lines]
        self.master.destroy()

        ca = my_ca.cellular_automata(0, 0, 0, 0, 0, 0, False, 0, False, 0, True, lines)
        ca.new_generation()
        self.master.destroy()

    def input(self):
        # manual user input
        root3 = tk.Tk()
        root3.title('Input Parameters')
        input_window = gui_CA_Param(root3)

    def show_history(self):
        root3 = tk.Tk()
        root3.title('History')
        history_window = gui_CA_history(root3)

    def close(self):
        self.master.destroy()


class gui_CA_Param:
    """Class for entering CA parameters
    Need to make some entry boxes dependent on checkboxes
    """

    def __init__(self, master):
        self.master = master
        self.frame = ttk.Frame(master, padding=5)
        self.frame.grid(row=0, column=0, sticky='nsew')

        # label variables
        self.l_lbl_name = ttk.Label(self.frame, text='Enter parameters for CA model')
        self.l_no_cells = ttk.Label(self.frame, text='Number of cells')
        self.l_gen = ttk.Label(self.frame, text='Generations')
        self.l_size_x = ttk.Label(self.frame, text='Size x')
        self.l_size_y = ttk.Label(self.frame, text='Size y')
        self.l_inf_rad = ttk.Label(self.frame, text='Infection radius')
        self.l_no_inf = ttk.Label(self.frame, text='Number of infected')
        self.l_r_i = ttk.Label(self.frame, text='Recovered can be infected?')
        self.l_d_r = ttk.Label(self.frame, text='Days until recovery')
        self.l_use_imm = ttk.Label(self.frame, text='Use immunity')
        self.l_d_i = ttk.Label(self.frame, text='Days of immunity')

        # bool values for checkbuttons
        self.b_r_i = tk.BooleanVar()
        self.b_u_i = tk.BooleanVar()
```

```
# entry and checkbutton variables
self.e_no_cells = ttk.Entry(self.frame)
self.e_gen = ttk.Entry(self.frame)
self.e_size_x = ttk.Entry(self.frame)
self.e_size_y = ttk.Entry(self.frame)
self.e_inf_rad = ttk.Entry(self.frame)
self.e_no_inf = ttk.Entry(self.frame)
self.cb_r_i = ttk.Checkbutton(self.frame, variable=self.b_r_i)
self.e_d_r = ttk.Entry(self.frame)
self.cb_use_imm = ttk.Checkbutton(self.frame, variable=self.b_u_i)
self.e_d_i = ttk.Entry(self.frame)

self.btn_enter = ttk.Button(self.frame, text='Enter', command=self.enter_param)

# grid label variables
self.l_lbl_name.grid(column=0, row=0, columnspan=2, sticky='n')
self.l_no_cells.grid(column=0, row=1, sticky='w')
self.l_gen.grid(column=0, row=2, sticky='w')
self.l_size_x.grid(column=0, row=3, sticky='w')
self.l_size_y.grid(column=0, row=4, sticky='w')
self.l_inf_rad.grid(column=0, row=5, sticky='w')
self.l_no_inf.grid(column=0, row=6, sticky='w')
self.l_r_i.grid(column=0, row=7, sticky='w')
self.l_d_r.grid(column=2, row=7, sticky='w')
self.l_use_imm.grid(column=0, row=8, sticky='w')
self.l_d_i.grid(column=2, row=8, sticky='w')

# grid entry variables
self.e_no_cells.grid(column=2, row=1, sticky='w')
self.e_gen.grid(column=2, row=2, sticky='w')
self.e_size_x.grid(column=2, row=3, sticky='w')
self.e_size_y.grid(column=2, row=4, sticky='w')
self.e_inf_rad.grid(column=2, row=5, sticky='w')
self.e_no_inf.grid(column=2, row=6, sticky='w')
self.cb_r_i.grid(column=1, row=7, sticky='w')
self.e_d_r.grid(column=3, row=7, sticky='w')
self.cb_use_imm.grid(column=1, row=8, sticky='w')
self.e_d_i.grid(column=3, row=8, sticky='w')

self.btn_enter.grid(column=0, row=9, columnspan=4, sticky='s')

# grid main column and tow
self.master.columnconfigure(0, weight=1)
self.master.rowconfigure(0, weight=1)

# weight elements for resizing
self.frame.columnconfigure(0, weight=1)
self.frame.rowconfigure(0, weight=1)
self.frame.rowconfigure(1, weight=1)
self.frame.rowconfigure(2, weight=1)
self.frame.rowconfigure(3, weight=1)
self.frame.rowconfigure(4, weight=1)
self.frame.rowconfigure(5, weight=1)
self.frame.rowconfigure(6, weight=1)
```

```
    def enter_param(self):
        # should call CA function in this
        no_cells = int(self.e_no_cells.get())
        generations = int(self.e_gen.get())
        size_x = int(self.e_size_x.get())
        size_y = int(self.e_size_y.get())
        inf_rad = int(self.e_inf_rad.get())
        no_inf = int(self.e_no_inf.get())
        rec_inf = self.b_r_i.get()
        days_rec = int(self.e_d_r.get())
        use_imm = self.b_u_i.get()
        days_imm = int(self.e_d_i.get())

        self.master.destroy()
        # print(rec_inf)
        # print(use_imm)

        arguments = [no_cells, generations, size_x, size_y, inf_rad, no_inf, rec_inf,
            days_rec, use_imm,
                    days_imm, False]

        my_sql.ca_enter_param(current_user, arguments)

        # ca = my_ca.cellular_automata(no_cells, generations, size_x, size_y, inf_rad,
            no_inf, rec_inf, days_rec, use_imm,
        #                            days_imm, False)
        ca = my_ca.cellular_automata(*arguments) # arguments sent as separate
            parameters

        ca.new_generation()


class gui_CA_history:

    def __init__(self, master):
        self.master = master
        self.frame = ttk.Frame(master, padding=5)
        self.frame.grid(row=0, column=0, sticky='nsew')
        self.user_history = my_sql.ca_return_history(current_user) # user_history is a
            list of tuples
        # print(self.user_history)
        # print(self.user_history[0][1:])


        self.lb_history = tk.Listbox(self.frame, width=50)
        for i in range(len(self.user_history)):
            to_insert = str(i) + " " + str(self.user_history[i][1:])
            self.lb_history.insert(i, to_insert)

        self.lbl_title = ttk.Label(self.frame, text=f"History of {current_user}")
        self.btn_exit = ttk.Button(self.frame, text="Exit", command=self.exit)
        self.lbl_text = ttk.Label(self.frame, text="Enter number to simulate with same
            parameters")
        self.e_sim_num = ttk.Entry(self.frame)
```

```
        self.btn_use = ttk.Button(self.frame, text="Use values", command=self.use)

        self.lbl_title.grid(column=1, row=1, columnspan=3)
        self.lbl_text.grid(column=1, row=2, columnspan=3)
        self.lb_history.grid(column=1, row=3)
        self.e_sim_num.grid(column=2, row=3)
        self.btn_use.grid(column=3, row=3)
        self.btn_exit.grid(column=2, row=4, columnspan=2)

    def exit(self):
        self.master.destroy()

    def use(self):
        """User enter number and set of parameters are retrieved from the database"""
        sim_number = int(self.e_sim_num.get())
        sim_param = list(self.user_history[sim_number][1:])
        print(sim_param)
        sim_param.append(False)
        ca = my_ca.cellular_automata(*sim_param)
        ca.new_generation()



# ----------------------------------------


root = tk.Tk()
root.title('Main Window')

window = gui_Main_Window(root)

root.mainloop()
```

8.2 SIR Model Function

```
import matplotlib.pyplot as plt
import pandas as pd
import sub_sql_functions as my_sql

class QueueSimulation:

    def __init__(self, n, s_list, i_list, r_list, b_list, g_list, t, current_user):
        self.n = n # number of simulations to be run
        self.parameters = []
        for i in range(n):
            self.parameters.append([s_list[i], i_list[i], r_list[i], b_list[i],
                g_list[i], t, i + 1])
            my_sql.sir_enter_param(current_user, [s_list[i], i_list[i], r_list[i],
                b_list[i], g_list[i], t])
        print(self.parameters)

    def run_simulation(self):
        sir_model = SIR_model()
        for i in range(self.n):
```

```
            sir_model.SIR_model(*self.parameters[i])
        plt.show()


class SIR_model:

    def __init__(self):
        pass

    def SIR_model(self, s0, i0, r0, beta, gamma, t, f, *args, **kwargs):
        """
        :param s0: number of susceptible people
        :param i0: number of infected people
        :param r0: number of recovered people
        :param beta: transmission rate of an individual
        :param gamma: recovery rate of an individual
        :param t: how long the simulation should run for
        :param f: how many graphs should be calculated simultaneously
        """

        plt.figure(f) # names the graph
        N = s0 + i0 + r0 # total population

        timearray = list(range(1, int(t))) # creates time array

        def eqns(param):
            # e = random.uniform(0, 1)
            e = 0
            S, I, R = param
            dsdt = (-(beta * S * I) / N) + (e * R) # rate of change of susceptible
                individuals
            didt = ((beta * S * I) / N) - gamma * I # rate of change of infected
                individuals
            drdt = (gamma * I) - (e * R) # rate of change of recovered individuals
            return dsdt, didt, drdt

        def solver(): # solves differential equations in eqns
            param = (s0, i0, r0)
            solver_result = [[], [], []]
            for time in timearray:
                eqns_results = eqns(param)
                x, y, z = (param[0] + eqns_results[0]), (param[1] + eqns_results[1]),
                    (param[2] + eqns_results[2])
                solver_result[0].append(x)
                solver_result[1].append(y)
                solver_result[2].append(z)
                param = (x, y, z)
            return solver_result

        solver_result = solver()

        # print(solver_result)

        def export_to_excel(): # exports calculated results to excel
            data = {'Time': timearray,
```

```
                'S': solver_result[0],
                'I': solver_result[1],
                'R': solver_result[2]}

        df = pd.DataFrame(data, columns=['Time', 'S', 'I', 'R'])
        name = "output" + str(f) + ".xlsx"
        df.to_excel(name, sheet_name='output')

    export_to_excel()
    plt.plot(timearray, solver_result[0], label="S(t)")
    plt.plot(timearray, solver_result[1], label="I(t)")
    plt.plot(timearray, solver_result[2], label="R(t)")
```

8.3 CA Function

```
# pylint: disable=unused-variable
# pylint: enable=too-many-lines

import matplotlib.pyplot as plt
import random
from matplotlib.animation import FuncAnimation
import json


class cell:
    """Each cell will be a class instance of this class"""

    def __init__(self, x, y, infected, d_r, d_i, u_i):
        self.x = x
        self.y = y
        self.infected = infected
        self.recovered = False
        self.recover_count = d_r # default - can be changed - time until infected cell
            recovers
        self.immune = False
        self.immune_count = d_i
        self.original_immune = d_i
        self.use_immunity = u_i
        # self.infection_rate = i

    def cell_test_function(self):
        return self.x, self.y, self.infected

    def location(self):
        # print(self.x, self.y)
        return self.x, self.y

    def recover_generation(self):
        self.recover_count -= 1
        if self.recover_count <= 0:
            self.recovered = True
            self.infected = False
            self.recover_count = 10
            if self.use_immunity:
```

```
            self.immune = True

    def immunity_generation(self):
        if self.immune:
            self.immune_count -= 1
            if self.immune_count <= 0:
                self.immune = False
                self.immune_count = self.original_immune

    def movement(self):
        def nothing():
            pass

        # mvmt = 5
        mvmt = random.randint(0, 50)

        def north():
            self.y -= mvmt

        def northeast():
            self.x += mvmt
            self.y -= mvmt

        def east():
            self.x += mvmt

        def southeast():
            self.x += mvmt
            self.y += mvmt

        def south():
            self.y += mvmt

        def southwest():
            self.x -= mvmt
            self.y += mvmt

        def west():
            self.x -= mvmt

        def northwest():
            self.x -= mvmt
            self.y -= mvmt

        instructions = {
            0: nothing,
            1: north,
            2: northeast,
            3: east,
            4: southeast,
            5: south,
            6: southwest,
            7: west,
            8: northwest
        }
```

```
        rand = random.randint(0, 8)
        instructions[rand]() # like a switch case condition - for constant time
            complexity
        # print(self.x, self.y)


class cellular_automata:
    """Main class to run function"""

    def __init__(self, no_cells, generations, size_x, size_y, infection_radius,
        infected, r_i, d_r, immunity, d_i,
                  user_file, *uf_param):
        """Creates list of how every many cells the user inputs so a list will look
            like ['cell1','cell2','cell3'...]. Each
        element in that list will then be used as a dictionary key where the
            definition will be a class instance of cell. Each
        class instance will be created with a random x and y coordinate, and an
            infected status.
        r_i : recovered can be infected
        d_r : days until recovery
        """
        self.user_file = user_file
        self.use_immunity = immunity
        self.number_of_cells = no_cells
        self.number_of_infected = infected
        self.infection_radius = infection_radius
        self.generations = generations
        self.cell_list = []
        self.cell_object_dict = {}
        self.size_x = size_x
        self.size_y = size_y
        self.full_list = []
        self.r_i = r_i # recovered can be infected

        if len(uf_param) != 0:
            self.uf_param = uf_param
            self.lines = list(uf_param)[0]

        for i in range(no_cells):
            self.cell_list.append(("cell" + str(i)))

        infected_counter = 0
        for element in self.cell_list: # creates user inputted number of infected
            cells first, then creates normal cells
            infected_counter += 1
            infected_status = False
            if infected_counter < self.number_of_infected:
                infected_status = True
            rand_x, rand_y = self.rand_coordinate_generator()
            self.cell_object_dict[element] = cell(rand_x, rand_y,
                                                  infected_status, d_r, d_i,
                                                  self.use_immunity) # creates a
                                                      dictionary of cell objects
```

```
def rand_coordinate_generator(self):
    """Generates random coordinates for cells being generated"""
    rand_x = random.randint(0, self.size_x)
    rand_y = random.randint(0, self.size_y)
    return rand_x, rand_y

def export_data(self, sus_full, inf_full, rec_full, imm_full, lg_values,
    time_array):
    """Should export data in a format that it can be analysed and reused
    gen_imm empty when not using immunity, but shouldn't be a problem"""

    filename = "ca_output.txt"
    with open(filename, 'w') as file:
        file.write(str(sus_full) + "\n")
        file.write(str(inf_full) + "\n")
        file.write(str(rec_full) + "\n")
        file.write(str(imm_full) + "\n")
        file.write(str(lg_values) + "\n")
        file.write(str(time_array) + "\n")

def import_data(self):

    if self.lines[-1] == "":
        del self.lines[-1]

    result = [json.loads(item) for item in self.lines]
    # print(result)
    return result

def update_position(self):
    """For each cell object, movement function will be called to see where the
        cell will move, and the x coordinate list and y coordinate list will be
        returned of the new generation"""
    loc_x = []
    loc_y = []
    infected = []
    recovered = []
    immune = []

    for cell_obj_name in self.cell_list: # for each cell object name ie. 'cell1',
        'cell2' etc, use the name as a dictionary key and run movement function
        and get location
        self.cell_object_dict[cell_obj_name].movement()
        loc_x.append(self.cell_object_dict[cell_obj_name].location()[0])
        loc_y.append(self.cell_object_dict[cell_obj_name].location()[1])
        infected.append(self.cell_object_dict[cell_obj_name].infected)
        recovered.append(self.cell_object_dict[cell_obj_name].recovered)
        immune.append(self.cell_object_dict[cell_obj_name].immune)

    # print(infected)
    # print(recovered)
    # print(immune)
    # print("------------")
```

```
    # self.collect_data() # this function could be moved into the new generation
        class function

    return loc_x, loc_y, infected, recovered, immune

def cells_touch(
        self): # need to change to put all infected in a list, THEN compare
            susceptible otherwise not proper
    """If another cell in in a certain radius of an infected cell, it will become
        infected. To be changed"""
    infected_locations = [] # list of tuples of infected cells
    recovered_obj = []
    susceptible_obj = []
    for cell_name in self.cell_list:
        if self.cell_object_dict[cell_name].infected:
            infected_locations.append(self.cell_object_dict[cell_name].location())
        elif self.cell_object_dict[cell_name].recovered:
            recovered_obj.append(self.cell_object_dict[cell_name]) # holds
                recovered both with and without immunity
        else:
            susceptible_obj.append(self.cell_object_dict[cell_name])

    def touch(cell_obj):
        sus_x, sus_y = cell_obj.location()
        for infected_tuple in infected_locations:
            if (sus_x - infected_tuple[0]) ** 2 + (
                    sus_y - infected_tuple[1]) <= self.infection_radius ** 2: #
                        equation of a circle
                # self.cell_object_dict[cell_name].infected = True # need to chance
                    for chance
                # PROBLEM
                cell_obj.infected = True
                # print(f'{cell_obj} has been infected')
                # cell is infected, LISTS ARE NOT UPDATED

    if self.r_i: # if recovered can be infected - doesn't change immunity status
        but is currently dependent on it - CHANGE
        for recovered in recovered_obj:
            if not recovered.immune:
                touch(recovered)
    for susceptible in susceptible_obj:
        if not susceptible.immune:
            touch(susceptible)

def cell_recovery(self):
    """Cells automatically recover after a certain period of time"""
    for cell_name in self.cell_list:
        cell_obj = self.cell_object_dict[cell_name]
        if cell_obj.recovered == False and cell_obj.infected == True:
            cell_obj.recover_generation()
        elif cell_obj.recovered == True and cell_obj.infected:
            cell_obj.recover_generation()

def cell_immunity(self):
    for cell_name in self.cell_list:
```

```
            cell_obj = self.cell_object_dict[cell_name]
            if cell_obj.recovered == True and cell_obj.immune == True:
                cell_obj.immunity_generation()

    def new_generation(self):
        """Main definition for running program. For the number of generations to
            simulate, call self.update_position() to get new coordinate lists"""

        # coordinates = []

        sus_full = []
        inf_full = []
        rec_full = []
        imm_full = []

        lg_values = [[], [], []]

        time_array = list(range(0, self.generations))

        if not self.user_file:
            # if user chosing own file will not need - put in function later
            for i in range(
                    self.generations):

                if i % 50 == 0:
                    print("Generating generation " + str(i))

                x_list, y_list, infected, recovered, immune = self.update_position()

                # check if cells touch here, then can adjust objects if need
                self.cells_touch() # adjusts the objects, not any list

                # recovery function
                self.cell_recovery()

                # immunity function
                if self.use_immunity:
                    self.cell_immunity()

                gen_sus = []
                gen_inf = []
                gen_rec = []
                gen_imm = []

                # puts cells in list depending on their status and whether immunity is
                    used
                if self.use_immunity:
                    for inf in range(len(self.cell_list)):
                        if infected[inf]: # if True
                            gen_inf.append([x_list[inf], y_list[inf]])
                        elif immune[inf]:
                            gen_imm.append([x_list[inf], y_list[inf]])
                        elif recovered[inf]:
                            gen_rec.append([x_list[inf], y_list[inf]])
                        else:
```

```
                    gen_sus.append([x_list[inf], y_list[inf]])
            else:
                for inf in range(len(self.cell_list)):
                    if infected[inf]: # if True
                        gen_inf.append([x_list[inf], y_list[inf]])
                    elif recovered[inf]:
                        gen_rec.append([x_list[inf], y_list[inf]])
                    else:
                        gen_sus.append([x_list[inf], y_list[inf]])

            # print(gen_inf)
            # print(gen_rec)
            # print(gen_sus)
            # print("--------------")

            sus_full.append(gen_sus)
            inf_full.append(gen_inf)
            rec_full.append(gen_rec)
            if self.use_immunity:
                imm_full.append(gen_imm)

            lg_values[0].append(len(gen_sus))
            lg_values[1].append(len(gen_inf))
            lg_values[2].append((len(gen_rec) + len(gen_imm)))

            # coordinates.append([x_list, y_list])

            # print(sus_full)

    # self.export_to_excel() # will soon be redundant
    if not self.user_file:
        self.export_data(sus_full, inf_full, rec_full, imm_full, lg_values,
            time_array) # WORKING NOW

    # if using own values, assign them here
    if self.user_file:
        sus_full, inf_full, rec_full, imm_full, lg_values, time_array =
            self.import_data()
        self.generations = len(time_array)
        print("Imported data!")
        # print(sus_full)
        # print(inf_full)
        # print(rec_full)
        # print(imm_full)
        # print(lg_values)
        # print(time_array)

    fig, axs = plt.subplots(2)
    fig.suptitle('Cellular Automata')

    # animate_graph = Anim(self.generations, sus_full, inf_full, rec_full,
        imm_full, self.use_immunity, time_array, lg_values)

    def animate(i): # need to adjust to work with two graphs
```

```
            #
                https://stackoverflow.com/questions/42621036/how-to-use-funcanimation-to-update-and-ar

            axs[0].cla()

            x_sus_full = [cell[0] for cell in sus_full[i]]
            y_sus_full = [cell[1] for cell in sus_full[i]]
            x_inf_full = [cell[0] for cell in inf_full[i]]
            y_inf_full = [cell[1] for cell in inf_full[i]]
            x_rec_full = [cell[0] for cell in rec_full[i]]
            y_rec_full = [cell[1] for cell in rec_full[i]]

            axs[0].scatter(x_sus_full, y_sus_full, color='blue')
            axs[0].scatter(x_inf_full, y_inf_full, color='red')
            axs[0].scatter(x_rec_full, y_rec_full, color='purple')
            if self.use_immunity:
                axs[0].scatter([cell[0] for cell in imm_full[i]], [cell[1] for cell in
                    imm_full[i]], color='gray')

            # plots line graph
            axs[1].plot(time_array[0:i], lg_values[0][0:i], label="Susceptible",
                color="blue")
            axs[1].plot(time_array[0:i], lg_values[1][0:i], label="Infected",
                color="red")
            axs[1].plot(time_array[0:i], lg_values[2][0:i], label="recovered",
                color="purple")

        # plt.xlim(0, self.size_x)
        # plt.ylim(0, self.size_y)

        ani = FuncAnimation(plt.gcf(), animate, frames=self.generations, interval=100,
            repeat=False)

        # ani.save('video.mp4', writer='ffmpeg', fps=30, dpi=250)

        plt.tight_layout()
        plt.show()
```