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

Zebedee Marsh

February 21, 2021

Contents

1 Introduction

Since 1999, there have been 11 major disease outbreaks 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 Initial objectives

These objects represent my initial aims of what this project should be able to do. These may change later on throughout this project.

- Simulate the spread of a disease using the SIR model
- Simulate the spread of a disease using the Cellular Automata model
- Input custom parameters or use default values (from a previous disease) including infection rate, incubation period and death rate for the SIR model
- Input custom parameters or use default values (from a previous disease) relating to the number and infectivity of cells for the CA model
- Set a quarantine period or introduce a vaccine (therefore limiting disease transmission)
- Click on a graph and show statistics from that point in time
- Export graphs produced as png
- Upload past simulation results into the program to generate a graph from that previous simulation
- Be able to compare graphs of two different diseases

3 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.

4 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

4.1 The SIR model

The SIR model is a simple disease spread model which uses differential equations to determine the number of people in three categories, the susceptible, the infected and the recovered. The susceptible population represent those who can become infected a disease, the infected population represent those currently infected with the disease who are able to spread it to other people and the recovered population represent those who are no longer infected. In some models, people in the recovered population can be infected again.

4.1.1 Equations for the SIR model

People in each group

$S = S(t)$ - number of susceptible individuals

$I = I(t)$ - number of infected individuals

$R = R(t)$ - number of recovered individuals

The Susceptible equation

$$\frac{dS}{dt} = \frac{-\beta S(t)I(t)}{N}$$

where β is the rate at which infections occur (a positive constant)

The Infected equation

$$\frac{dI}{dt} = \frac{\beta S(t)I(t)}{N} - \gamma I(t)$$

where γ is the rate at which people recover

The Recovered equation

$$\frac{dR}{dt} = \gamma I(t)$$

Model without vital dynamics

$$\frac{dS}{dt} + \frac{dI}{dt} + \frac{dR}{dt} = 0$$

As this model doesn't take into account vital dynamics ie. people aren't added through birth or removed through death, the sum of all the changes is constant and equal to 0

4.2 The Cellular Automata model

The basic idea of cellular automata that there there is a collection of cells with a finite set of states. Every new generation, the state of a cell can be changed depending on the state of it's neighbouring cells.

4.2.1 My implementation of CA

- Cells move randomly within a grid. Most cells will be susceptible but some will be infected
- If a susceptible cell comes in close range to an infected cell, there is a chance that that susceptible cell will be come infected
- Infected cells are only infected for a certain period of time, so after a certain number of generations the cell will no longer be infected and will be in the recovered category

5 Prototyping of code

5.1 The scientific models

5.1.1 Elementary cellular automata

My initial cellular automata model was very basic and only worked on a 1D line. The shape of each new line would be decided by looking at the previous line and applying a ruleset to it.

```python
class CA:
    def __init__(self, width, generations):
        self.ruleset = [0, 1, 1, 1, 1, 0, 1, 1]
        self.width = width
        self.cells = [0] * width
        self.new_cells = [None] * width
        self.cells[int((width / 2) - 1)] = 1
        self.generations = generations
        self.cell_timeline = []

    def rules(self, a, b, c, i):
        index = int((a + b + c), 2)
        ruleset = [0, 1, 0, 1, 1, 0, 1, 0]
        self.new_cells[i] = ruleset[index]

    def draw(self):
        output = ""
        for cell_state in self.cell_timeline:
            for element in cell_state:
                if element == 0:
                    output += " " + " "
                else:
                    output += "X" + " "
            print(output)
            time.sleep(0.1)
            output = ""

    def ca(self):
        for gen in range(self.generations):
            for i in range(self.width):
                a = str(self.cells[((i - 1) % self.width)])
                b = str(self.cells[i])
                c = str(self.cells[((i + 1) % self.width)])
                # print(a,b,c)
                self.rules(a, b, c, i)
            self.cells = self.new_cells
            self.cell_timeline.append(self.cells)
            self.new_cells = [None] * self.width
        self.draw()

ca = CA(100, 100)
ca.ca()
```

The result of elementary cellular automata

```
                              x x
                             x   x
                            x x x x
                           x       x
                          x x     x x
                         x   x   x   x
                        x x x x x x x x
                       x               x
                      x x             x x
                     x   x           x   x
                    x x x x         x x x x
                   x       x       x       x
                  x x     x x     x x     x x
                 x   x   x   x   x   x   x   x
                x x x x x x x x x x x x x x x x
               x                               x
              x x                             x x
             x   x                           x   x
            x x x x                         x x x x
           x       x                       x       x
          x x     x x                     x x     x x
         x   x   x   x                   x   x   x   x
        x x x x x x x x                 x x x x x x x x
       x               x               x               x
      x x             x x             x x             x x
     x   x           x   x           x   x           x   x
    x x x x         x x x x         x x x x         x x x x
   x       x       x       x       x       x       x       x
  x x     x x     x x     x x     x x     x x     x x     x x
 x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
```

5.1.2 Basic SIR model

My starting block for the SIR model was a function which calculated the rate of change of the susceptible, infected and recovered population.

```
def eqns(y0, t, beta, gamma):
S, I, R = y0
dsdt = -(beta * S * I) / N # rate of change of susceptible individuals
didt = ((beta * S * I) / N) - gamma * I # rate of change of infected individuals
drdt = gamma * I # rate of change of recovered individuals
return dsdt, didt, drdt
```

5.2 External library testing TBD

5.2.1 Tkinter

My intial prototying for tkinter was done with the help of a YouTube video which focused on having multiple tkinter pages in a single file. Link: https://youtu.be/jBUpjijYtCk

This prototype allowed there to be multiple Tkinter windows which could be switched between by a button click. The way this method worked was by removeing and adding widgets depending on which page should be shown. Although this method may have worked for me, in the final project I went with destroying a Tkinter window and starting a completely new one when a user wanted to switch, as the code for this was easier to understand and write.

```
import tkinter as tk


class Gui(tk.Tk):
   def __init__(self, *args,
                  **kwargs): # *args - pass through any number of variables; **kwargs
                      - pass through dictionaries
       tk.Tk.__init__(self, *args, **kwargs) # initialising tkinter
       container = tk.Frame(self) # the frame of the window
       container.pack(side="top", fill="both", expand=True)
       container.grid_rowconfigure(0, weight=1) # 0 sets minimum size, weight shows
           priority
       container.grid_columnconfigure(0, weight=1)
       self.frames = {} # allows application to open different types of pages easily

       for F in (StartPage, PageOne): # all pages need to be listed here
           frame = F(container, self)
           self.frames[F] = frame # saving classes to dictionary, "loading it in"
           frame.grid(row=0, column=0, sticky="nsew")

       self.show_frame(StartPage) # start page is shown first

   def show_frame(self, cont):
       frame = self.frames[cont] # looks for cont in dict
       frame.tkraise() # which is then raised


class StartPage(tk.Frame):

   def __init__(self, parent,
                  controller): # parent class is Gui, controller class is main class,
                      allowing show_frame to be called
       tk.Frame.__init__(self, parent)

       label = tk.Label(self, text="Start Page")
       label.pack(pady=10, padx=10)
       button1 = tk.Button(self, text="Page 1", command=lambda:
           controller.show_frame(PageOne))
       button1.pack()


class PageOne(tk.Frame):

   def __init__(self, parent, controller):
       tk.Frame.__init__(self, parent)

       label = tk.Label(self, text="This is page 1")
       label.pack(pady=10, padx=10)
       button2 = tk.Button(self, text="Start Page", command=lambda:
           controller.show_frame(StartPage))
       button2.pack()


app = Gui()
app.mainloop()
```

The result

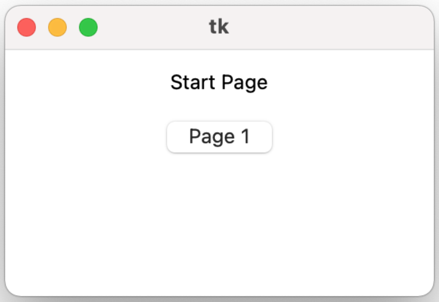


Figure 1: Start Tkinter window

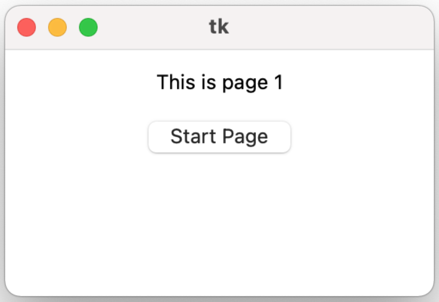


Figure 2: Page 1 Tkinter window

5.2.2 Matplotlib - FuncAnimation

The initial way I tried to update a graph with new values was by deleting the graph and plotting a new one. Funcanimation provided an easier way to do this, with the option of saving the animation as a video which was one of the objectives for my project. This prototype shows how a line graph works with funcanimate.

It works for both line graphs and scatter graphs

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from itertools import count
import random

x_full = [[229, 29, 52, 90, 78], [228, 29, 52, 90, 79], [229, 28, 53, 89, 79],
    [228, 27, 52, 90, 78], [227, 28, 51, 89, 79], [227, 27, 50, 88, 78], [226,
    28, 51, 88, 79], [227, 29, 51, 89, 79], [227, 30, 51, 90, 79], [228, 30, 51,
    90, 79], [229, 30, 50, 89, 78], [228, 30, 49, 90, 79], [228, 30, 50, 89, 78],
    [228, 31, 50, 88, 78], [229, 32, 49, 87, 78], [230, 33, 50, 87, 77], [230,
    34, 50, 88, 76], [231, 35, 51, 87, 75], [232, 36, 50, 87, 74], [232, 37, 51,
    86, 74]]
y_full = [[37, 23, 94, 195, 90], [37, 22, 93, 194, 89], [36, 22, 94, 195, 88],
    [36, 22, 95, 194, 88], [35, 22, 94, 195, 88], [36, 22, 95, 196, 89], [35, 21,
    94, 197, 90], [34, 20, 94, 196, 91], [34, 20, 94, 197, 90], [35, 19, 94, 196,
    90], [35, 18, 95, 197, 89], [35, 18, 94, 198, 88], [35, 19, 93, 197, 89],
    [35, 20, 94, 196, 90], [34, 20, 95, 196, 90], [33, 19, 95, 195, 90], [33, 19,
    96, 196, 91], [33, 20, 95, 196, 90], [32, 20, 94, 195, 90], [31, 19, 93, 194,
    91]]

x_vals = []
y_vals = []

index = count()

def animate(i):
    x_vals.append(next(index))
    y_vals.append(random.randint(0,5))
    plt.plot(x_vals, y_vals)
```

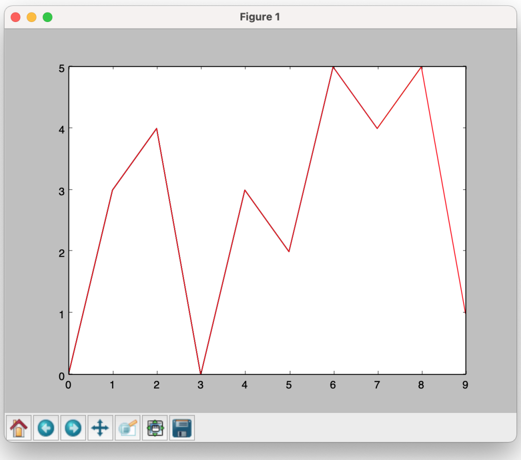


Figure 3: Graph 1

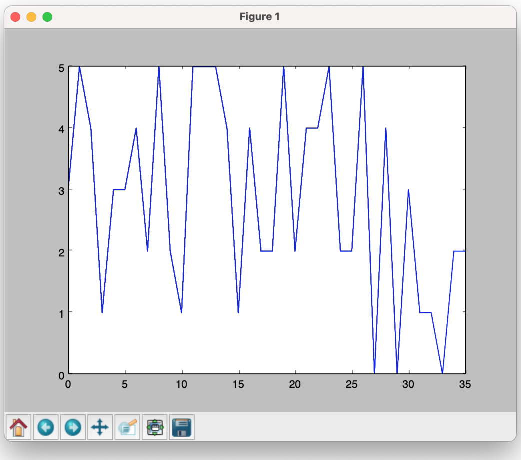


Figure 4: Graph 2

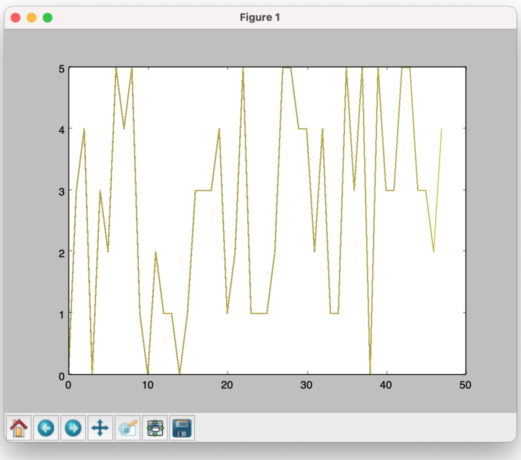


Figure 5: Graph 3

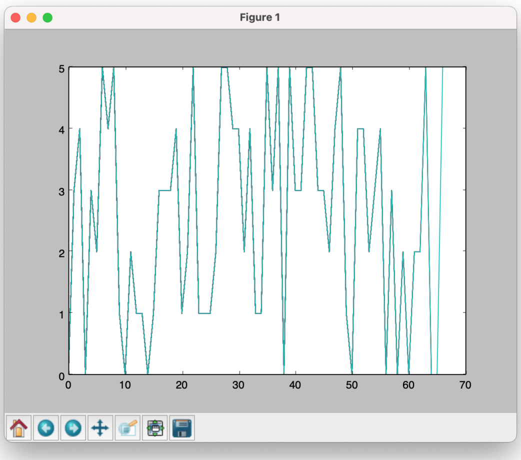


Figure 6: Graph 4

```
    ani = FuncAnimation(plt.gcf(), animate, interval=10) # get current figure,
        function, interval
    plt.show()
```

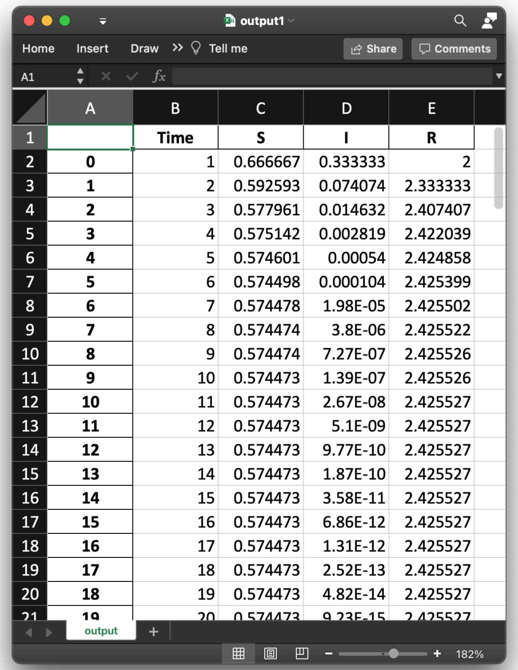
The result

5.2.3 Pandas

I used Pandas to help export the data generated using the SIR model. This allowed me to generate an excel file with appropriate column names, which could possibly be used later for data analysis

```
def export_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')
```



	A	B	C	D	E
1		Time	S	I	R
2	0	1	0.666667	0.333333	2
3	1	2	0.592593	0.074074	2.333333
4	2	3	0.577961	0.014632	2.407407
5	3	4	0.575142	0.002819	2.422039
6	4	5	0.574601	0.00054	2.424858
7	5	6	0.574498	0.000104	2.425399
8	6	7	0.574478	1.98E-05	2.425502
9	7	8	0.574474	3.8E-06	2.425522
10	8	9	0.574474	7.27E-07	2.425526
11	9	10	0.574473	1.39E-07	2.425526
12	10	11	0.574473	2.67E-08	2.425527
13	11	12	0.574473	5.1E-09	2.425527
14	12	13	0.574473	9.77E-10	2.425527
15	13	14	0.574473	1.87E-10	2.425527
16	14	15	0.574473	3.58E-11	2.425527
17	15	16	0.574473	6.86E-12	2.425527
18	16	17	0.574473	1.31E-12	2.425527
19	17	18	0.574473	2.52E-13	2.425527
20	18	19	0.574473	4.82E-14	2.425527
21	19	20	0.574473	9.23E-15	2.425527

5.2.4 Pygame

One of the options I considered to simulate and show my Cellular Automata simulation was Pygame. With Pygame I would be able to easily create cells which could move randomly in a set space, and using Pygame's collision method I could detect if cells touched. In the end I decided against this method as although it may have been easier, using Matplotlib graphs would grant me extra flexibility, for example if I wanted to export position data of each cell I would already know how to do that with Matplotlib. The prototype I tested was made with the help of a YouTube video covering Pygame: https://youtu.be/NjvIooRpuH4

```
import pygame
import time

pygame.init()

black = (0,0,0)
white = (255,255,255)
red = (255,0,0)

gameDisplay = pygame.display.set_mode((800,600))
pygame.display.set_caption('Test')
```

```
clock = pygame.time.Clock()

carImg = pygame.image.load('unnamed.jpg')

def text_objects(text, font):
    textSurface = font.render(text, True, black) # what to render, anti-aliasing,
        colour
    return textSurface, textSurface

def message_display(text):
    largeText = pygame.font.Font('freesansbold.ttf', 115)
    TextSurf, TextRect = text_objects(text, largeText)
    TextRect.center = (800/2, 600/2)
    gameDisplay.blit(TextSurf, TextRect)

    time.sleep(2)

    game_loop()

def crash():
    message_display('You crashed')

def car(x,y): # car object
    gameDisplay.blit(carImg, (x,y)) # drawing carImg to x,y

def game_loop():

    x = (100)
    y = (100)
    x_change = 0

    gameExit = False

    while not gameExit: # user controls
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                gameExit = True

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_LEFT:
                    x_change = -10
                elif event.key == pygame.K_RIGHT:
                    x_change = 10

            if event.type == pygame.KEYUP:
                if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
                    x_change = 0

            print(event)

        x += x_change

        gameDisplay.fill(white)
        car(x,y)
        if x > 800 or x < 0: # boundaries
            crash()

        pygame.display.update()
        clock.tick(60)

game_loop()
pygame.quit()
```

5.3 SQL database TDB

6 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.

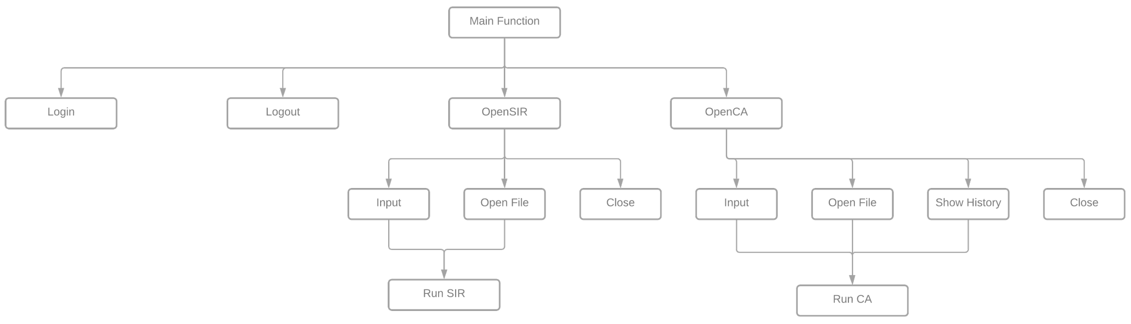
Q1. How could such a tool like this be used in the classroom?

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

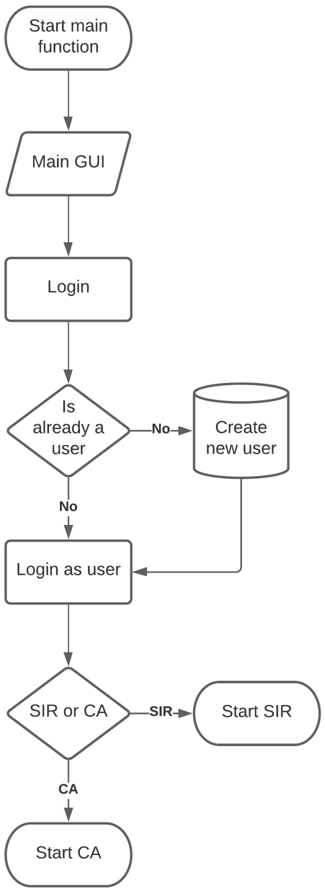
E1. I will try to implement a feature where the parameters of two diseases could be entered, and a graph could be calculated and shown for both, which would allow a user to compare the spread of the two diseases. I will also add features that a government may try and implement to restrict a disease. 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.

7 Documented design

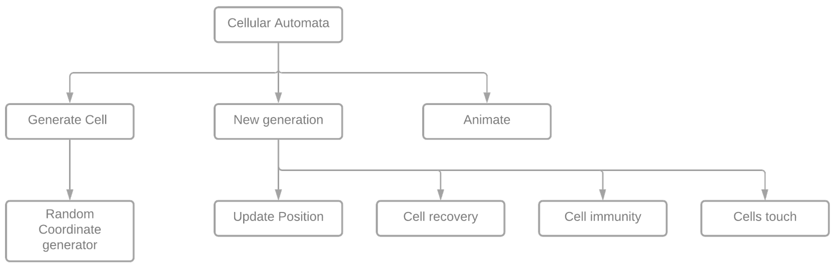
7.1 Main function structure diagram



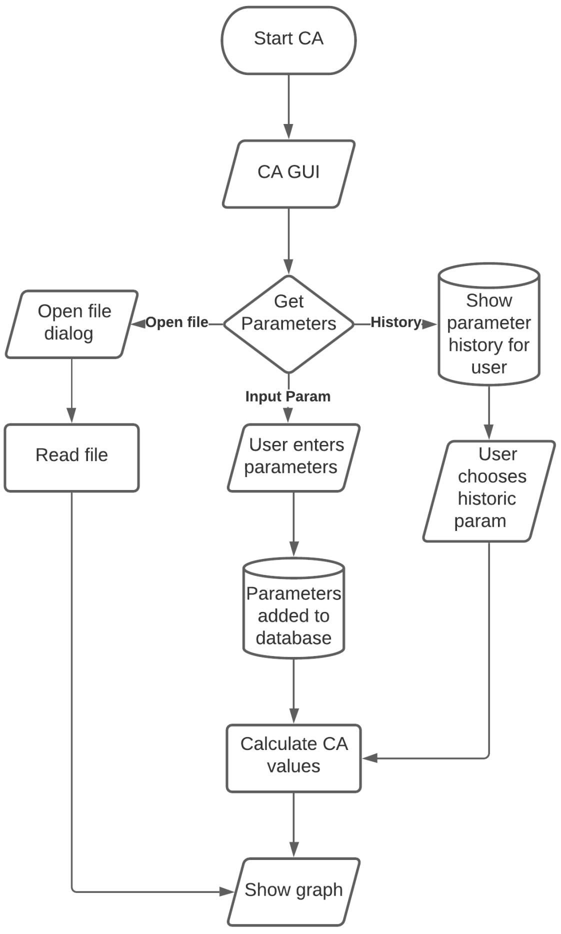
7.2 Main function flowchart



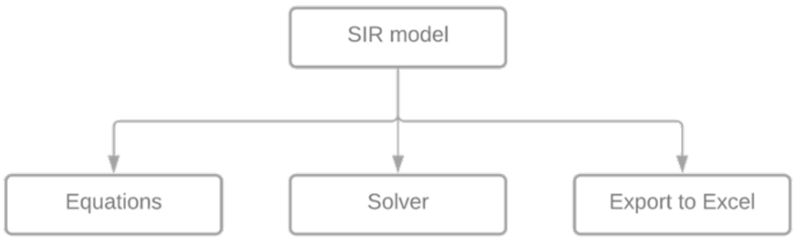
7.3 Cellular automata structure diagram



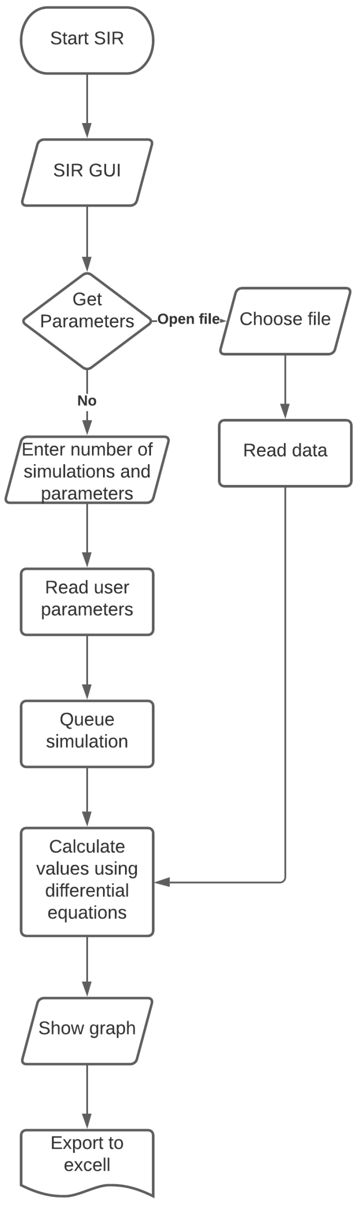
7.4 Cellular automata flowchart



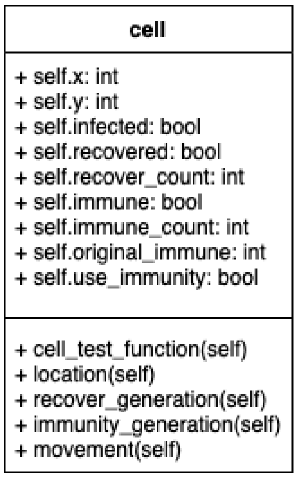
7.5 SIR Model structure diagram

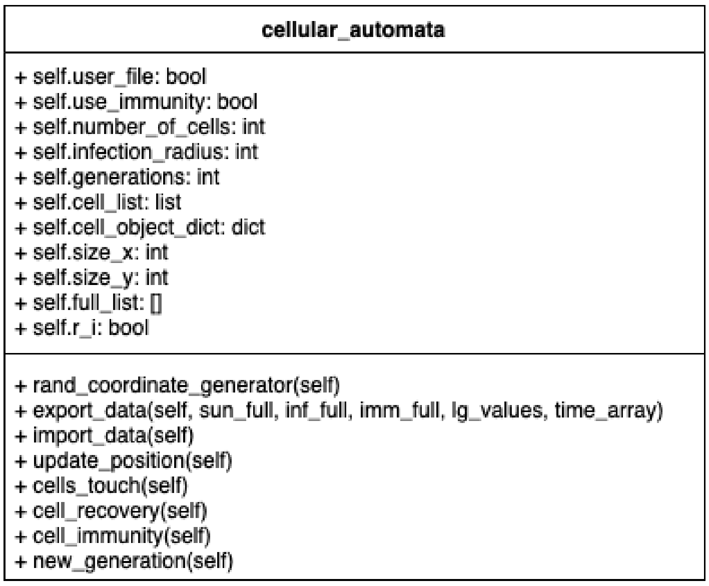


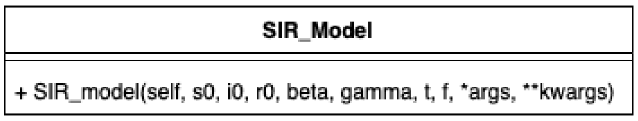
7.6 SIR model flowchart

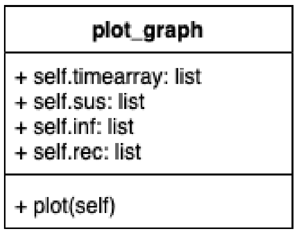


7.7 Cellular automata and SIR model UML charts

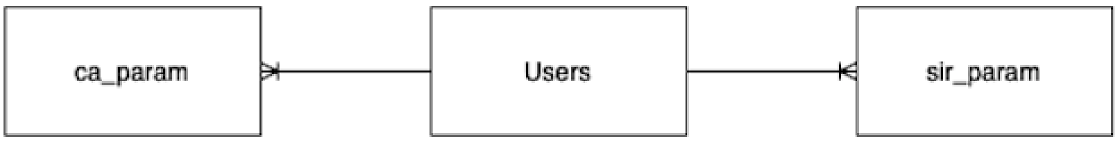

cell
+ self.x: int
+ self.y: int
+ self.infected: bool
+ self.recovered: bool
+ self.recover_count: int
+ self.immune: bool
+ self.immune_count: int
+ self.original_immune: int
+ self.use_immunity: bool
+ cell_test_function(self)
+ location(self)
+ recover_generation(self)
+ immunity_generation(self)
+ movement(self)




SIR_Model
+ SIR_model(self, s0, i0, r0, beta, gamma, t, f, *args, **kwargs)



7.8 SQL Entity relationship diagram



7.9 SQL queries used

Creating a new table

```
CREATE TABLE users (username text PRIMARY KEY, see_all integer)

CREATE TABLE ca_param (
    user string,
    no_cells integer,
    generations integer,
    size_x integer,
    size_y integer,
    infection_radius integer,
    no_infected integer,
    recovered_can_be_infected integer,
    days_until_recovered integer,
    use_immunity integer,
    days_of_immunity integer
    )

CREATE TABLE sir_param (
    user string,
    sus0 integer,
    inf0 integer,
    rec0 integer,
    beta integer,
    gamma integer,
    time integer
    )

INSERT INTO users VALUES (:username, :see_all), {'username': 'admin', 'see_all': 1}
```

This code sets up a clean database containing 3 tables, users, ca_param and sir_param. In the users table there are two columns, one containing the username and another containing a value to decide whether that user can see the history of all calculations performed and not just the calculations performed by that user. THIS WILL BE UPDATED. The ca_param and sir_param table contain columns to record the user doing the calculation and the parameters the user entered.

Inserting a new user into the user table

```
INSERT OR IGNORE INTO users VALUES (:username, :see_all),{'username': in_user,
    'see_all': 0}
```

Entering parameters from the Cellular Automata model

```
INSERT INTO ca_param VALUES (
    :user, :no_cells, :generations, :size_x, :size_y,
    :infection_radius, :no_infected, :recovered_can_be_infected,
    :days_until_recovered, :use_immunity, :days_of_immunity),
    {'user': in_user, 'no_cells': up[0], 'generations': up[1], 'size_x': up[2],
        'size_y': up[3],
        'infection_radius': up[4], 'no_infected': up[5], 'recovered_can_be_infected':
            up[6],
        'days_until_recovered': up[7], 'use_immunity': up[8], 'days_of_immunity':
           up[9]}
```

This query will be passed the user, in_user, and a list of the parameters entered, up. This will be added into the ca_param table.

Entering parameters from the SIR model

```
INSERT INTO sir_param VALUES (:user, :sus0, :inf0, :rec0, :beta, :gamma,
    :time),{'user': in_user, 'sus0': up[0], 'inf0': up[1], 'rec0': up[2], 'beta':
    up[3], 'gamma': up[4], 'time': up[5]
    }
```

Similar to function above except for the SIR model

Return history from Cellular Automata model

```
SELECT * FROM ca_param WHERE user=:curr_user, {'curr_user': in_user}
```

Return history from SIR model

```
SELECT * FROM sir_param WHERE user=:curr_user, {'curr_user':in_user}
```

8 Technical Solution

8.1 External Libraries

```
    import tkinter as tk
    from tkinter import ttk
    import pandas as pd
    from tkinter import filedialog
    from tkinter import messagebox

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

Tkinter - For all my GUI code, I use Tkinter to structure my windows. Tkinter provides a good set of widgets including Labels, Entry boxes and Buttons which are easy to structure using Tkinter's Grid geometry manager. Ttk provides newer looking widgets. Tkinter also comes with Filedialog which is used when the user choses a file from a previous simulation, and MEssagebox is used for error nessages. (Pandas) - This is used to create a dataframe to allow simulation results to be exported into an Excel file.

8.2 SIR Model

8.2.1 SIR Submit Parameters

```
self.param_list = [[], [], [], [], [], []]

def submit_param(self):
"""Creates queue object from my_sir function and calls the run simulation function
"""
# print(self.param_list)
self.master.destroy()

pass1 = True
for sub_list in self.param_list:
    for value in sub_list:
        if value < 0:
            pass1 = False
            err = error("Value", "Negative numbers cannot be used for any of these
                inputs")

if not pass1:
    root3 = tk.Tk()
    root3.geometry("+{}+{}".format(200, 200))
    root3.title('Input Parameters')
    input_window = gui_SIR_Param(root3, self.number_of_simulations)
    root3.mainloop()
else:
    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],
                                 self.param_list[5], current_id)
    queue.run_simulation()
```

When a user enters the number of SIR graphs they want to produce, if the number is greater than 1 and less than 10, a queue will be created. If they enter 1 there is no queue and if they enter more then 10 they will be asked to enter a lower number. If a user enters a number greater than 2 for number of graphs they want to produce, the Enter Parameters window will show up once where the user enters the first set of parameters, and when they click submit another Enter Parameters window will show up. This continues until they've entered all the sets of parameters for the number of graphs they want. Each time submit is pressed, these parameters are placed in a 2D List.

eg. List = [[sus1, sus2], [inf1, inf2], [rec1, rec2], [beta1, beta2], [gamma1, gamma2], [time1, time2]] Where sus1, inf1, rec1, beta1, gamma1 and time1 are the first set of parameters, and sus2, inf2, rec2, beta2, gamma2 and time2 are the second set ofparameters

8.2.2 SIR Queue Simulation

```
class QueueSimulation:

def __init__(self, n, s_list, i_list, r_list, b_list, g_list, t, current_id):
    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], i + 1])
        my_sql.sir_enter_param(current_id, [s_list[i], i_list[i], r_list[i],
            b_list[i], g_list[i], t[i]])
    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()
```

This class takes in the 2D list mentioned in SIR Submit Parameters and groups them into their sets.

eg. List = [[sus1, sus2], [inf1, inf2], [rec1, rec2], [beta1, beta2], [gamma1, gamma2], [time1, time2]] turns into Parameters = [[sus1, inf1, rec1, beta1, gamma1, time1],[sus2, inf2, rec2, beta2, gamma2, time2]]

This essential creates a queue of parameters which will then be used to generate values for the SIR Graph.

8.2.3 SIR Solver

```
def eqns(param):
    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
```

Eqns - This function contains the established SIR equations which calculate rate of change of the different states. S is susceptible individuals, I is infected individuals and R is recovered individuals. Beta represents the infection rate and Gamma represents the rate of recovery. The Eqns function takes in the current values of S, I and R, and using the beta and gamma constants calculates the rate of change of each class and returns them. **Solver** - This function solves the differential equations in the eqns function. A time array contains a list of numbers, 1,2,3,4... which represent the time period, or number of generations, to calculate S, I and R values for. For each number in this time period list, the current values of S, I and R are input into the eqns function which calculates the rate of change of each state. This is then returned and added to the S, I and R values to calculate the current S, I and R values for that time period. These new values are appended to a list. This is then repeated, where the current S, I and R values are input into the eqns function, the change calculated and added to the current S, I and R and the values entered into a list.

The list of values is structured as so: List = [[s1, i1, r1], [s2, i2, r2], [s3, i3, r3]...] Where the letter represents the state, whether susceptible, infected or recovered, and the number represents the generation it belongs to. Once all the S, I, R values have been calculated for each generation, it is plotted using MatPlotLib

8.2.4 SIR Plot

```
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)")
```

Each generation is plotted in a TKinter line graph.

8.3 CA Model

8.3.1 CA Cell Class

```
class cell:
"""Each cell will be a class instance of this class. Gives each cell its own
    attributes"""

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.original_recover = d_r
        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 = self.original_recover
            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)
```

Each cell will be an instance of this cell class. Each cell will therefore be independent and have it's own parameters, including

- It's own X and Y coordinates within the specified range by the user
- Whether it's infected
- Whether it's a recovered cell (ie has been infected) and if not how many generations until it recovers
- Whether it's immune (ie has been infected but is immune) and if so for how long

To calculate whether or not the cell is recovered or whether it's immune, there are two functions.

Recovery If a cell is infected, it will have a value stating the number of days it will be infected for. This value is entered by the user when entering parameters for the simulation. For each generation, if a cell is infected, it's recover_count variable will count down until it reaches 0. Once it reaches 0, the cell will no longer be infected but now recovered, and may have immunity if the user as set it. The recover_count variable will then reset in case the user has enabled recovered cells to be infected again.

Immunity Much like the recovery function, if the user has set it, cells can have an immunity after they've recovered. This immunity will last for a specified number of generations in the immunity_count variable. For each generation, this variable will count down until it reaches 0, where a cell no longer is immune. The immunity_count variable is reset to it's original value.

Movement function This function changes the position of a cell randomly. A random mvmt number between 0 and 50 is generated which determinues how far a cell moves. Another random number between 0 and 8 is generated to determine the direction in which this cell moves. The different directions are stored in a dictionary to enable it to act like a switch case condtion which results in constant time complexity, saving a bit of time and processing.

8.3.2 CA Initial Cell Creation

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

This code from the main cellular automata class creates the initial cells. The total number of cells to create and the number of infected cells are input into the function. Initially, a list is created containing cell names "cell1", "cell2", "cell3" etc. Then, the function starts creating cell object instances of the cell class. For each cell in the initial cell list:

- While the current number of infected cells is less than the number of infected cells wanted, the cell will be classed as infected
- A set of random coordinates will be generated within the specified x and y coordinatse by the user
- A cell object with these parameters and the specified days until recovery and days immune will be created and stored in a dictionary

8.3.3 CA New 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]])


        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)))
```

This function is the main function of the CA class and calculates the state of each cell object for each generation. The way this happens is as follows:

- Positions of cells are updated via the self.update_position() function
- Susceptible cells within a specified distance of infected cells have a chance at getting infected via the self.cells_touch() function
- Infected cells have their days until recovery variable counted down via the self.cell_recovery() function. If this countdown reaches 0, the cell will no longer be infected
- If the user has opted to give cells immunity when they have recovered, this immunity count variable is counted down and if it reaches 0 the cell becomes susceptible
- The coordinates of cells in the different states (susceptible, recovered, infected) are placed in lists for Matplotlib to plot.

How cell coordinates are entered into the list The coordinates of each cell is placed in it's respective list for each generation, and each generation list is appended to a master list which is plotted by Matplotlib. For example:

generation_1_infected_list = [[x1, y1], [x2, y2]]

where x1 and y1 are x and y coordinates of cell 1 and x2 and y2 are x and y coordinates of cell 2

generation_2_infected_list = [[x1, y1], [x6, y6]]

infected_full_list = [[[x1, y1], [x2, y2]], [[x1, y1], [x6, y6]]]

8.4 The code

8.4.1 Main Function

```
INSERT CODE HERE
```

8.4.2 CA Model

```
INSERT CODE HERE
```

8.4.3 SIR Model

```
INSERT CODE HERE
```

8.4.4 SQL Functions

```
INSERT CODE HERE
```

9 Evaluation

9.1 Evaluating against initial objectives

Initial objectives

- Simulate the spread of a disease using the SIR model

I have achieved this as shown in the testing video where an SIR graph can be produced through a variety of methods. This methods result in a matplotlib window showing a line graph with 3 lines representing susceptible, infected, and recovered. Unfortunaty the graph produced is quite simple due to way I have implemented this model. This model only takes in account the number of susceptible, infected and recovered individuals, as well as the transmission rate and recovery rate, which means it ignores factors such as natural birth and death rate, as well as different susceptibilities of individuals. To improve this simulation, I would add the ability to enter more custom parameters such as infectibility, natural birth and death rate, as well as death rate from the virus. A variable transmission rate and recovery rate may also make this simulation more realistic as in the real world, both rates are variable and can depend on things such as whether a vaccine has been rolled out or if a disease has mutated into a more contagious varient.

- Simulate the spread of a disease using the Cellular Automata model

This has been achived as shown in the testing video, where two animated graphs are shown as the user submits the parameters for it. I feel like this objective has been comfortably met as my algorithm manages to simulate disease spread in the way cellular automata intends, with individual cells acting the way they should if they come into contact with an infected cell. I am very pleased with the way this is represented to the user, as two matplotlib graphs, one a scatter graph and the other a line graph. The scatter graph shows which state a cell is in and allows the user to visualise clearly how close one cell is from another, which I believe is the best way to show this simulation. The line graph below it shows how the total number of cells in a group, susceptible, infected or recovered, changes with time, which I think is a nice addition and shows the similarities between the CA model and SIR model.

If i had more time on this project, I don't think I'd need to improve the core functionality of this model but I would add a feature that would allow the cellular automata model to run indefinitely, and there'd be options such as introduce a new virus or introduce a vaccine. This way, the code would be able to generate values such as a cells coordinates and whether it's infected as it's being animated on a matplotlib graph, which would allow the simulation to go on for how ever long a person would want. Another improvement I'd like to make is similar to one on my SIR model whether there could be natural birth and death rates, as currently there are only a set number of cells which doesn't change throughout the simulation.

- Input custom parameters or use default values (from a previous disease) including infection rate, incubation period and death rate for the SIR model
- Input custom parameters or use default values (from a previous disease) relating to the number and infectivity of cells for the CA model

The ability to input custom parameters is there as shown in the testing where a user can enter a number of parameters relating to the simulation, but I have not been able to add default values from a previous disease. This was due to a time constraint, as I

THIS NEEDS TO BE COMPELTED

- Set a quarantine period or introduce a vaccine (therefore limiting disease transmission)
- Click on a graph and show statistics from that point in time
- Export graphs produced as png
- Upload past simulation results into the program to generate a graph from that previous simulation
- Be able to compare graphs of two different diseases

10 Testing

Testing video available at https://youtu.be/JWIhQDoZN2M