COVID-19 SIMULATION PROJECT

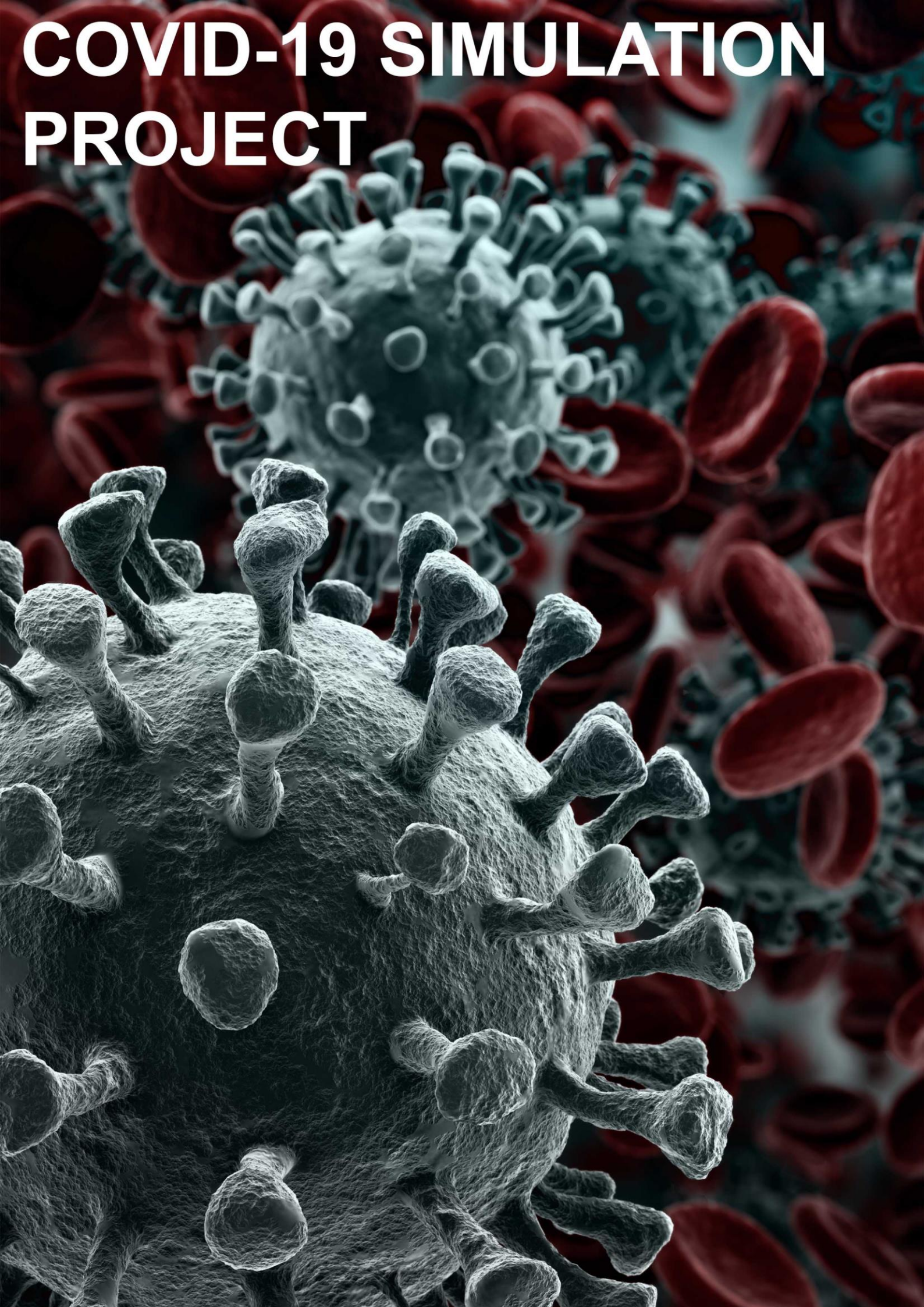


Table of Contents

COVID-19 SIR-Simulation

1 Introduction

1.1 Scope of Work

In the light of the situation in the early months of 2020, when the virus COVID-19 caused a global pandemic and led to massive uncertainty both in the health sector and the economy, and governments across the globe struggled to find fitting policies to balance the risks for public health with the risks to the local and global economy, I decided to code a SIR Model in python in order to shed some light on possible consequences of different actions during the outbreak of a pandemic.

A simple 2D world is created, where a certain population walks around in a markov chain random walk. After a patient zero is created, there are different actions that can be taken: the government can either do nothing and let the virus spread, or they can introduce travel restrictions across countries in order to curtail the spread, or they can do combinations of those two (i.e. doing nothing for a certain period of time, and then introducing travel restrictions, or the other way around).

1.2 The SIR Model

The SIR model divides the population of a country (or across countries) into 3 departments: susceptible are those who are healthy but not immune to the virus, infectious are those who currently are infected with the virus, and recovered/removed are those who either recovered from the virus and have some immunity, or who deceased due to the health consequences of the virus.

1.3 Markov Chain

A markov chain is a stochastic model that describes a sequence of events in which each event only depends on the state of the previous event. That means that a person in our example can walk in 4 directions, left, right, upwards and downwards and ends up in a position that only depends on the position it has been in before.

2 Code Explanation

2.1 Libraries

In order to gain some performance boost, the numpy library has been used to conduct computational difficult calculations. For the visualizations, matplotlib has been utilized

```
# LIBRARIES
import numpy as np
import matplotlib.pyplot as plt
```

2.2 Classes

2.2.1 Population

2.2.1.1 Initialization

For the initialization of the population, there needs to be a number of people in this 2D world (popsize), the distance within which a person can infect another person (infect_threshold) the probability with which one gets infected (infect_prob), the size of the 2D world (world_size), the number of different cities within the 2D world, since that will be important for travel restrictions (city_number), and the size of the step each person takes a day in his random walk (step_size):

```
class Population():
    def __init__(
            self,
            popsize=100,
            infect_threshold=50,
            infect_prob=1,
            world_size=100,
            city_number=16,
            step_size=40,
            seed=None):
        self.popsize = popsize
        self.infect_threshold = infect_threshold
        self.infect_prob = infect_prob
        self.world_size = world_size
        x = np.zeros(popsize)
        y = np.zeros(popsize)
        city = np.zeros(popsize)
        infection_period = np.zeros(popsize)
        state = np.empty(popsize, dtype=str)
        for i in range(popsize):
            state[i] = 'S'
            x[i] = np.random.randint(0, world_size)
            y[i] = np.random.randint(0, world_size)
        self.x, self.y, self.state = x, y, state
        self.infection_period = infection_period
        self.city_number = city_number
        self.city_grid_range = int(np.sqrt(city_number))
        self.city_size = int(world_size / np.sqrt(city_number))
        self.row = np.zeros(popsize)
        self.col = np.zeros(popsize)
        self.city = city
        self.step_size = step_size
```

2.2.1.2 Create Different Cities

It is important to define for each person in which city he is currently walking in. In order to do so, a grid has to be defined; this will be shown with 16 cities and a city size of 250x250 in a total world of 1000x1000

City Grid Example with 16 cities (4 horizontaly, 4 verticaly) and a city size of 250				
	0-250	250-500	500-750	750-1000
0-250	1	5	9	13
250-500	2	6	10	14
500-750	3	7	11	15
750-1000	4	8	12	16

depending on a person's current x and y coordinates he will get assigned to a row and a column and city number is then calculated with the following formula:

$$city = row + \sqrt{number\ of\ cities} * (column - 1)$$

The code runs generically with any number of cities that has an integer square root and looks as follows:

```
def check_city(self):
    city_size = self.city_size
    city_number = self.city_number
    popsize = self.popsize
    city_grid_range = self.city_grid_range
    for personindex in range(popsize):
        x = self.x[personindex]
        y = self.y[personindex]
        row, col = self.row[personindex], self.col[personindex]
        for i in range(city_grid_range):
            if city_size * (i) <= x and x < city_size * (i+1):
                col = i+1
                break
            else:
                col = None
        for i in range(city_grid_range):
            if city_size * (i) <= y and y < city_size * (i+1):
                row = i+1
                break
            else:
                row = None

        city = row + np.sqrt(city_number) * (col - 1)
        self.city[personindex] = city
        self.row[personindex], self.col[personindex] = row, col
```

2.2.1.3 Random Walk Without Travel Restrictions

The standard random walk without travel restrictions is defined as follows: every person has a certain probability to walk on the given day. If he walks, he can walk in one of four directions with the defined step size: left, right, upwards, downwards. If he would step out, let us say on the right side of the world, he will re-appear on the left side of the world. This is to ensure that nobody can walk outside of the defined world.

In the end of each walk, it is checked whether the person is infecting another person in its surrounding (here it is important that we know whether or not travel restrictions have been enacted, since we assume that you cannot infect across borders once they are closed down):

```
def rand_walk(self, personindex, step_size, action='nothing'):
    if np.random.rand() > 0.5:
        return

    x = self.x[personindex]
    y = self.y[personindex]
    world_size = self.world_size
    val = np.random.randint(1, 4)
    if val == 1:
        x = (x + step_size) % world_size
        y = y
    elif val == 2:
        x = (x - step_size) % world_size
        y = y
    elif val == 3:
        x = x
        y = (y + step_size) % world_size
    else:
        x = x
        y = (y - step_size) % world_size
    self.x[personindex], self.y[personindex] = x, y
    self.check_infection(personindex, action)
```

2.2.1.4 Random Walk With Travel Restrictions

In the random walk with travel restrictions, it has to be checked that a person only walks inside the city he or she has been when the travel restrictions have been enacted.

In the end of each walk, it is checked whether the person is infecting another person in its surrounding (here it is important that we know whether or not travel restrictions have been enacted, since we assume that you cannot infect across borders once they are closed down):

```
def rand_walk_inside_city(self, personindex, step_size, action='nothing'):
    if np.random.rand() > 0.5:
        return
    x = self.x[personindex]
    y = self.y[personindex]
    city_size = self.city_size
    row, col = self.row[personindex], self.col[personindex]
    val = np.random.randint(1, 4)
    if val == 1:
        if (x + step_size) < col * city_size:
            x += step_size
        else:
            x = city_size * (col - 1) + \
                ((x + step_size) - (col * city_size))
        y = y
    elif val == 2:
        if (x - step_size) > (col - 1) * city_size:
            x -= step_size
        else:
            x = (col * city_size) - \
                np.abs((((col - 1) * city_size) - (x - step_size)))
        y = y
    elif val == 3:
        x = x
        if (y + step_size) < row * city_size:
            y += step_size
        else:
            y = city_size * (row - 1) + \
                ((y + step_size) - (row * city_size))
    else:
        x = x
        if (y - step_size) > (row - 1) * city_size:
            y -= step_size
        else:
            y = (row * city_size) - \
                np.abs((((row - 1) * city_size) - (y - step_size)))
    self.x[personindex], self.y[personindex] = x, y
    self.check_infection(personindex, action)
```

2.2.1.5 Checking If the Walking Person is Infecting Anyone

After each random walk, it is checked whether the person who just walked will infect anyone else. That depends on the distance between them, the probability of infection, if the other person is immune and whether or not they are in the same city if the borders are closed down:

```
def check_infection(self, personindex, action='nothing'):
    x, y, state, city = self.x, self.y, self.state, self.city
    infect_threshold = self.infect_threshold
    infect_prob = self.infect_prob
    popsize = self.popsize
    for otherindex in range(popsize):
        if otherindex != personindex:
            if action == 'travelrestriction' and city[personindex] != city[otherindex]:
                d = np.inf
            else:
                pother = (x[otherindex], y[otherindex])
                pperson = (x[personindex], y[personindex])
                d = distance(pother, pperson)
            if d < infect_threshold and np.random.rand() < infect_prob:
                if (state[personindex] ==
                        'I' and state[otherindex] == 'S'):
                    state[otherindex] == 'I'
                    self.state[otherindex] = state[otherindex]
                elif(state[otherindex] == 'I' and state[personindex] == 'S'):
                    state[personindex] = 'I'
                    self.state[personindex] = state[personindex]
```

The distance is calculated as follows:

```
def distance(pi, pj):
    # Unpack the points into their coordinates
    xi, yi = pi
    xj, yj = pj
    # Just the usual formula for the distance
    return np.sqrt((xi - xj)**2 + (yi - yj)**2)
```

2.2.2 SIR Simulation

2.2.2.1 Initialization

The initialization is the same as for the population since the variables are handed down to that class, plus we also have to define a number of days we want the simulation to run and follow the number of healthy, infected and recovered patients::

```
class SIR_Simulation():
    def __init__(
            self,
            popsize=100,
            days=10,
            infect_prob=1,
            infect_threshold=50,
            world_size=100,
            step_size=40,
            seed=None):
        if not (isinstance(popsize, int) and popsize >= 1):
            raise Exception("n must be an int greater than 1")
        self.days = days
        self.popsize = popsize
        self.infect_prob = infect_prob
        self.infect_threshold = infect_threshold
        self.world_size = world_size
        self.population = Population(
            popsize, infect_threshold, infect_prob, world_size)
        healthy = np.zeros(days)
        infected = np.zeros(days)
        removed = np.zeros(days)
        self.infected = healthy
        self.healthy = infected
        self.removed = removed
        self.graph_pos = 0
```

2.2.2.2 Creating Patient Zero

When creating patient zero, we choose a random person out of the population and infect him or her:

```
def create_patient_zero(self):
    population = self.population
    popsize = self.popsize
    index = np.random.randint(0, popsize)
    population.state[index] = 'I'
    self.population = population
```

2.3 Visualization

2.3.1 Visualizing the current state of the 2D World

In order to visualize the world after each day, we create little dots with different colors for each person according to his or her state (green for susceptible, red for infectious and light blue for recovered):

```
def display(self):
    population = self.population
    popsize = self.popsize
    x, y = population.x, population.y
    # Clear the figure
    plt.clf()
    # Plot the person locations with colors for S,I,R
    for index in range(popsize):
        color = 'none'
        if population.state[index] == 'S':
            color = "green"
        elif population.state[index] == 'I':
            color = "red"
        else:
            color = "lightblue"
        plt.plot(x[index], y[index], "o", c=color)
        for i in range(int(np.sqrt(population.city_number))):
            plt.plot((population.city_size *
                      i, population.city_size *
                      i), (0, self.world_size), c='black')
            plt.plot((0, self.world_size), (population.city_size *
                                            i, population.city_size * i), c='black')
    #hack to force a display update
    plt.pause(0.00001)
```

2.3.2 Tracking and Plotting the Situation of the Health State of the World over the Days

In order to get a feeling for the health state of the world, a time series was established in order to track the number of susceptible, infectious and recovered over the days:

```
def plot_graph(self):
    ax = plt.subplot()
    ax.set_ylabel('Popoluation')
    ax.set_xlabel('Days')
    plt.plot(
        np.arange(
            0,
            self.days),
        self.healthy,
        color='green',
        label='Susceptible')
    plt.plot(
        np.arange(
            0,
            self.days),
        self.infected,
        color='red',
        label='Infected')
    plt.plot(
        np.arange(
            0,
            self.days),
        self.removed,
        color='lightblue',
        label='Removed')
    plt.xticks(np.arange(0, self.days, 5.0))
    plt.legend()
```

2.4 The Actual Simulation Function

The simulation is actually just an accumulation of the different functions described above: it runs for the given number of days with a given action (either doing nothing or imposing travel restrictions), and after two weeks (14 days), infected people recover and become immune (or they die in response to the virus):

```
def simulate(self, timespan, action='nothing'):
    popsize = self.popsize
    if action == 'travelrestriction':
        self.population.check_city()

    for day in range(timespan):
        steps_nothing = np.random.randint(
            (self.world_size / 5), self.world_size)
        steps_action = np.random.randint(
            (self.population.city_size / 5),
            self.population.city_size)
        for index in range(popsize):
            if action == 'nothing':
                self.population.rand_walk(index, steps_nothing, action)
            elif action == 'travelrestriction':
                self.population.rand_walk_inside_city(
                    index, steps_action, action)

            if self.population.state[index] == 'I' and self.population.infection_period[index] < 14:
                self.population.infection_period[index] += 1
            elif self.population.state[index] == 'I' and self.population.infection_period[index] >= 14:
                self.population.infection_period[index] = np.inf
                self.population.state[index] = 'R'
        self.display()
        self.healthy[self.graph_pos] = np.count_nonzero(
            self.population.state == 'S')
        self.infected[self.graph_pos] = np.count_nonzero(
            self.population.state == 'I')
        self.removed[self.graph_pos] = np.count_nonzero(
            self.population.state == 'R')
        print('healthy: {0}; infected: {1}; removed: {2}'.format(
            self.healthy[self.graph_pos], self.infected[self.graph_pos], self.removed[self.graph_pos]))
        print(
            'end of day {0} with action {1}'.format(
                self.graph_pos, action))
        self.graph_pos += 1
```

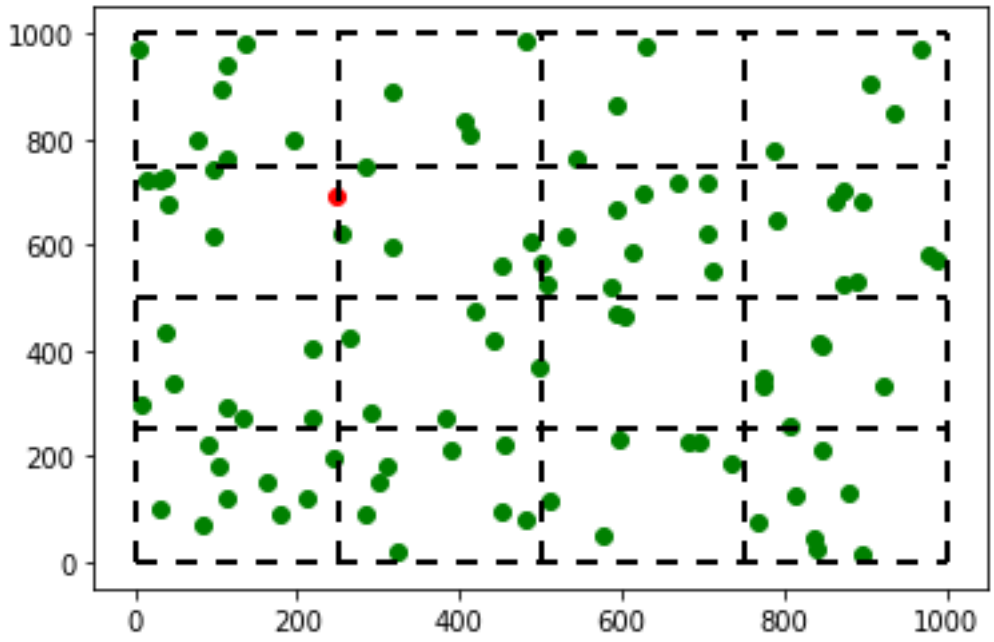
3 Different Simulations

3.1 Doing Nothing and let the Virus spread

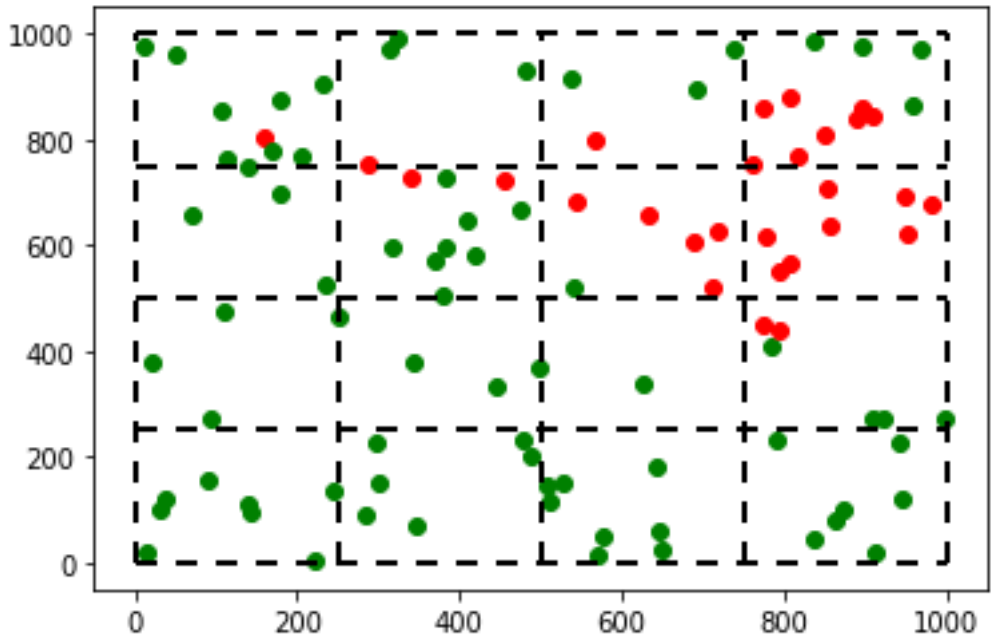
In this scenario, we let the virus spread for 100 days without any policy intervention, the cities are marked with dashed lines to show that people can travel freely:

```
###Do Nothing and let the Virus spread###
print('start')
sim = SIR_Simulation(
    popsize=100,
    days=100,
    infect_threshold=150,
    infect_prob=0.4,
    world_size=1000,
    step_size=40)
sim.create_patient_zero()
sim.simulate(sim.days, action='nothing')
sim.plot_graph()
```

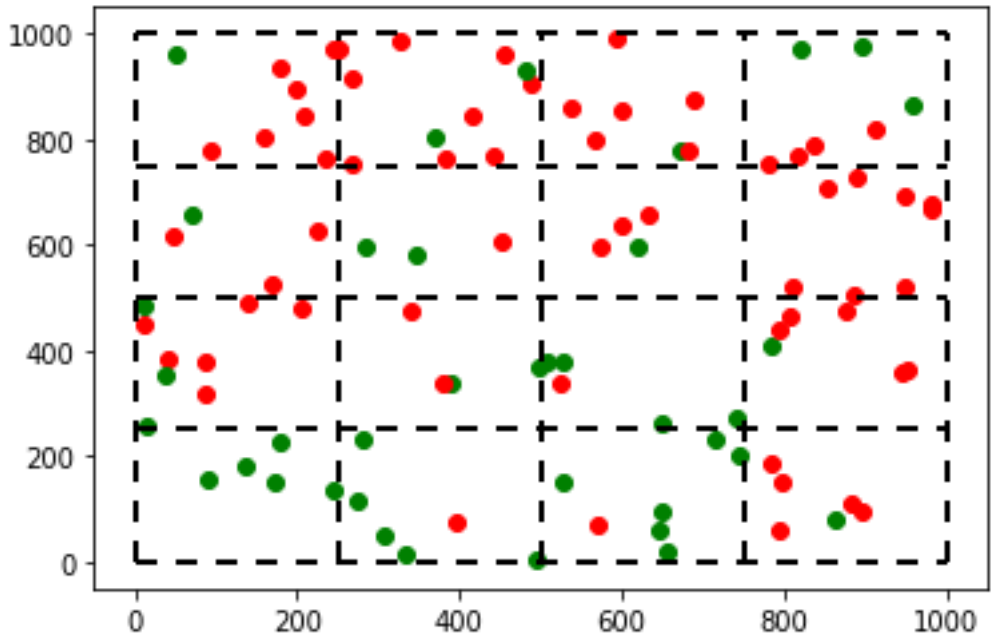
3.1.1 End of First Day



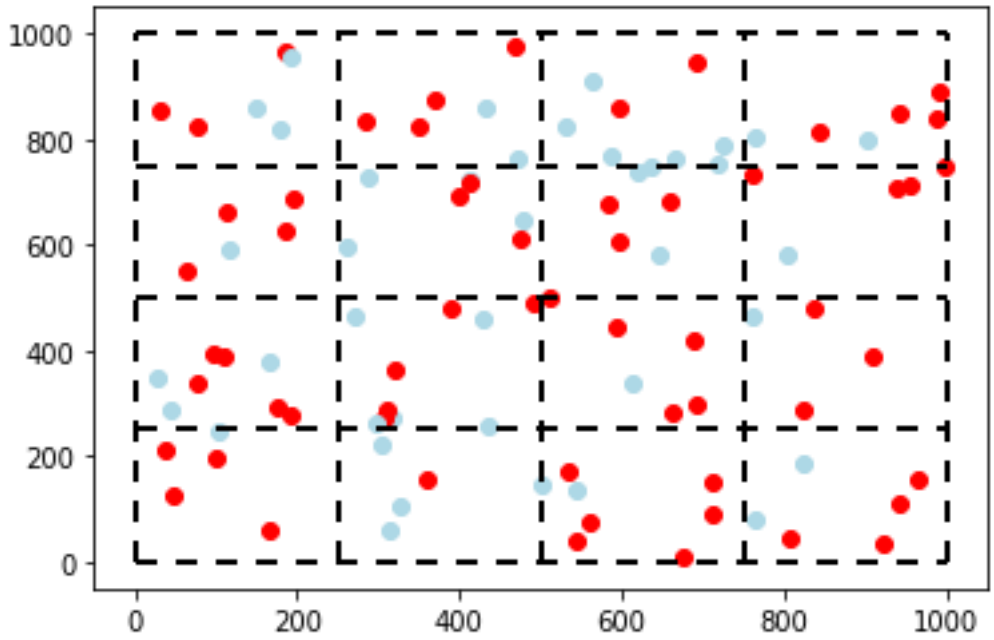
3.1.2 End of Fifth Day



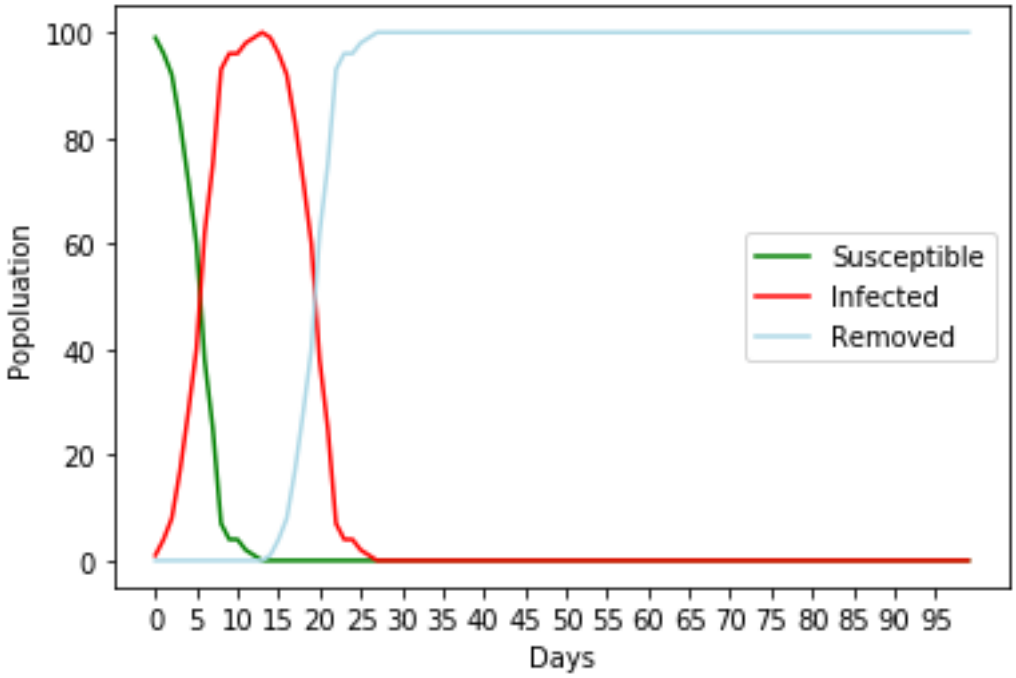
3.1.3 End of Seventh Day



3.1.4 End of 20th Day



3.1.5 Time Series



3.1.6 Comment on Results

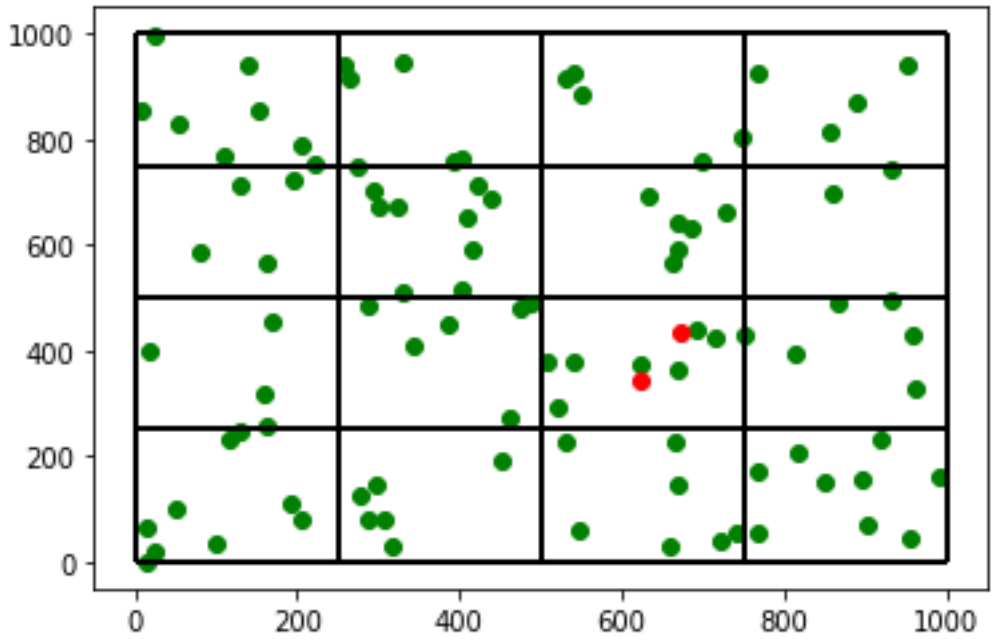
We see that without policy intervention, the virus spreads rapidly within a few days, most likely overburdening the health care system in a short period.

3.2 Impose Travel Restrictions immediately

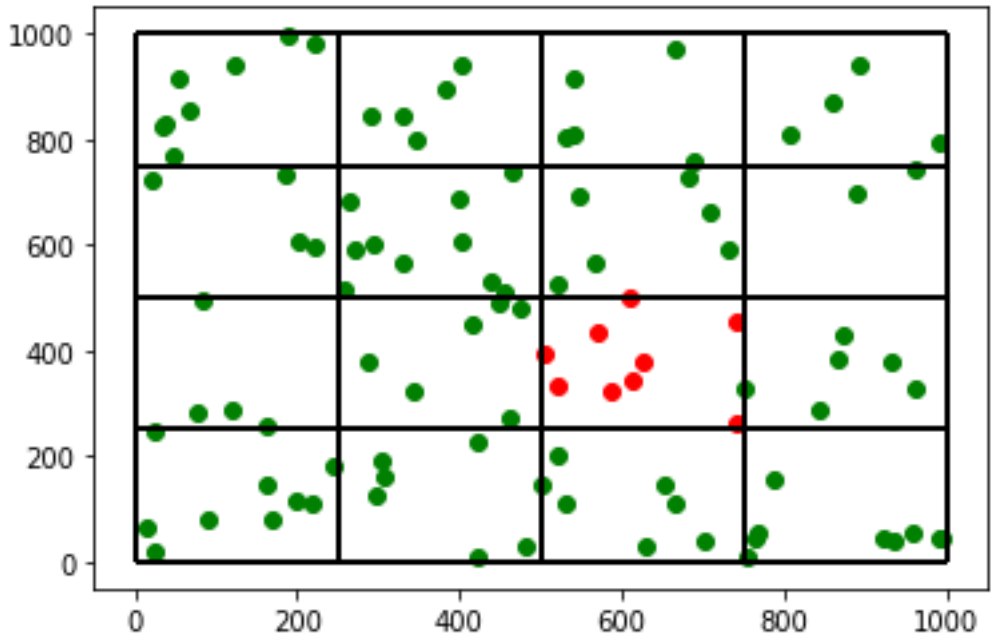
In this scenario, we impose travel restrictions starting from the first day, the cities are marked with solid lines to show that people cannot leave their city anymore:

```
###Impose Travel Restrictions immediately###
print('start')
sim = SIR_Simulation(
    popsize=100,
    days=100,
    infect_threshold=150,
    infect_prob=0.4,
    world_size=1000,
    step_size=40)
sim.create_patient_zero()
sim.simulate(sim.days, action='travelrestriction')
sim.plot_graph()
```

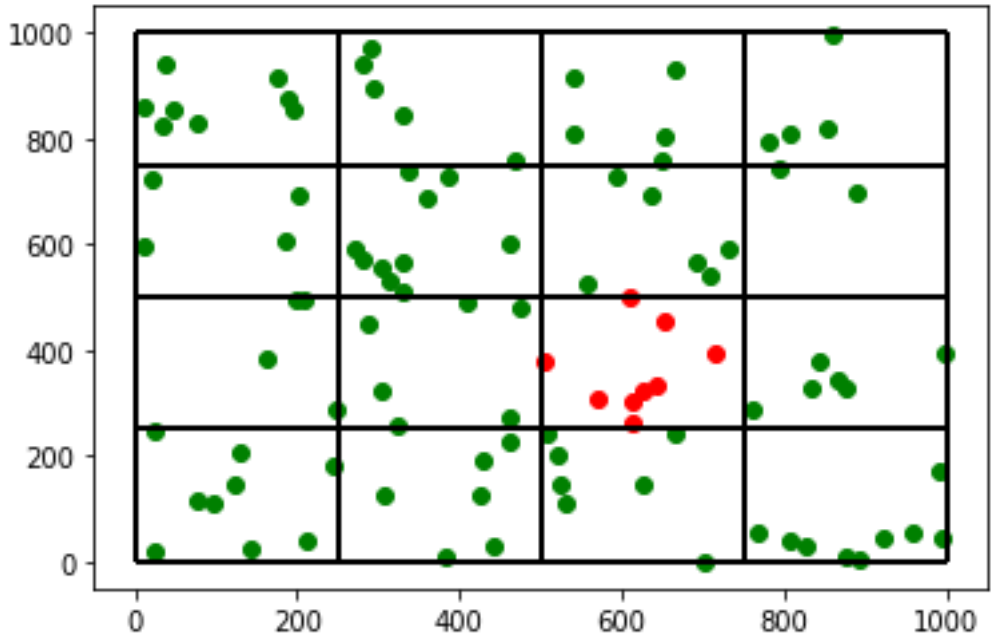
3.2.1 End of First Day



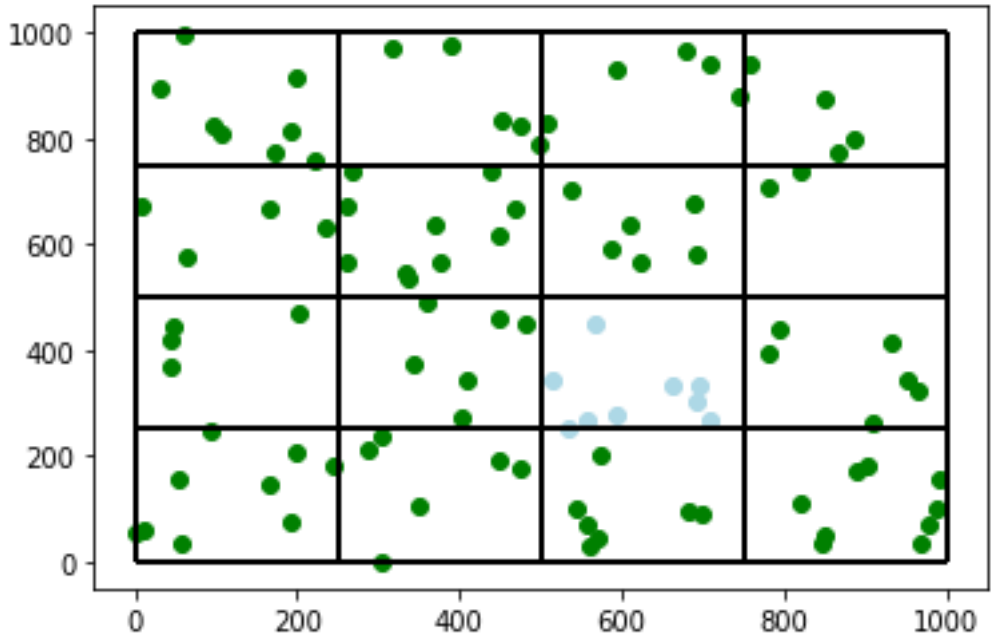
3.2.2 End of Fifth Day



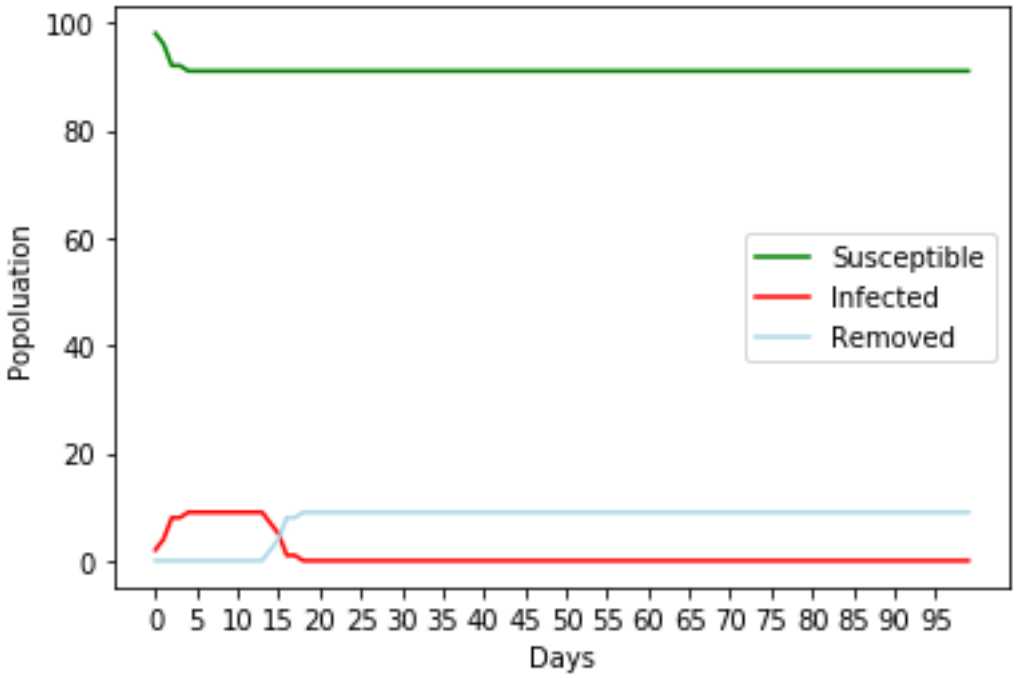
3.2.3 End of Seventh Day



3.2.4 End of 20th Day



3.2.5 Time Series



3.2.6 Comment

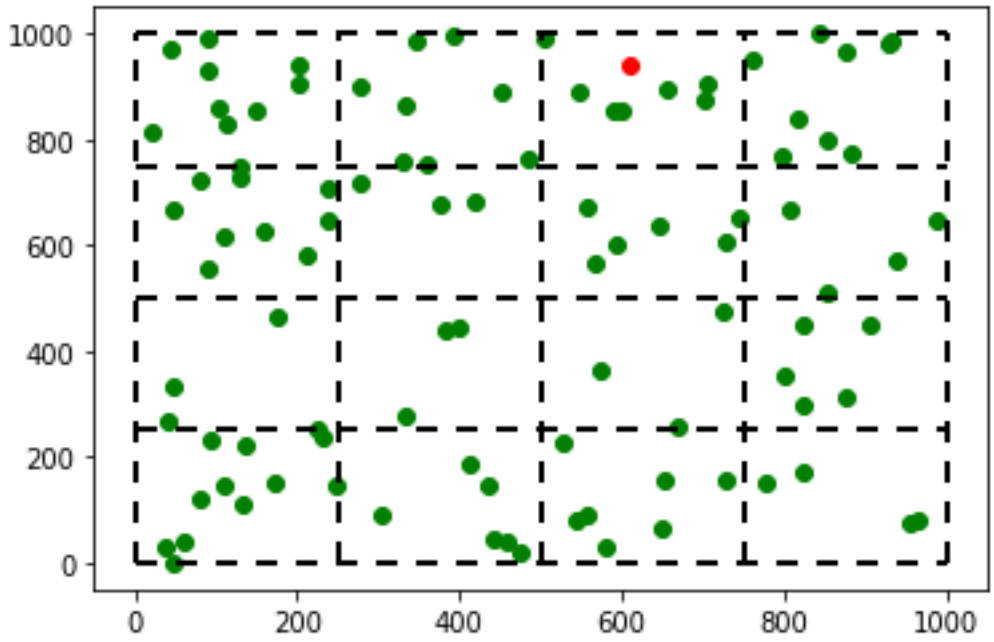
We see that with swift policy action, the virus can be contained within one city, and the health care system will not be overburdened. However, this also means not a lot of people got immune to the virus, and a second wave of infections could be likely.

3.3 First do nothing and after some Time impose Travel Restrictions

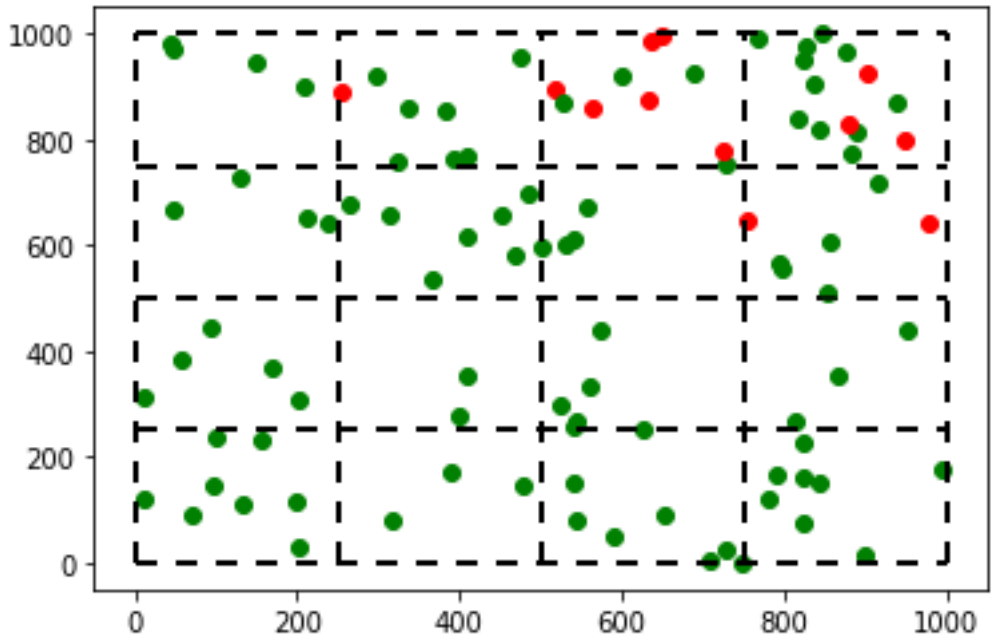
In this scenario, we let the virus spread freely for the first 8 days, afterwards, travel restrictions are introduced:

```
###First Do Nothing then Introduce Travel Restrictions###
print('start')
sim = SIR_Simulation(
    popsize=100,
    days=100,
    infect_threshold=150,
    infect_prob=0.4,
    world_size=1000,
    step_size=40)
nothing_action_ratio = 0.08 #8 days of doing nothing
days_nothing = int(sim.days * nothing_action_ratio)
days_action = sim.days - days_nothing
sim.create_patient_zero()
sim.simulate(days_nothing, action='nothing')
sim.simulate(days_action, action='travelrestriction')
sim.plot_graph()
```

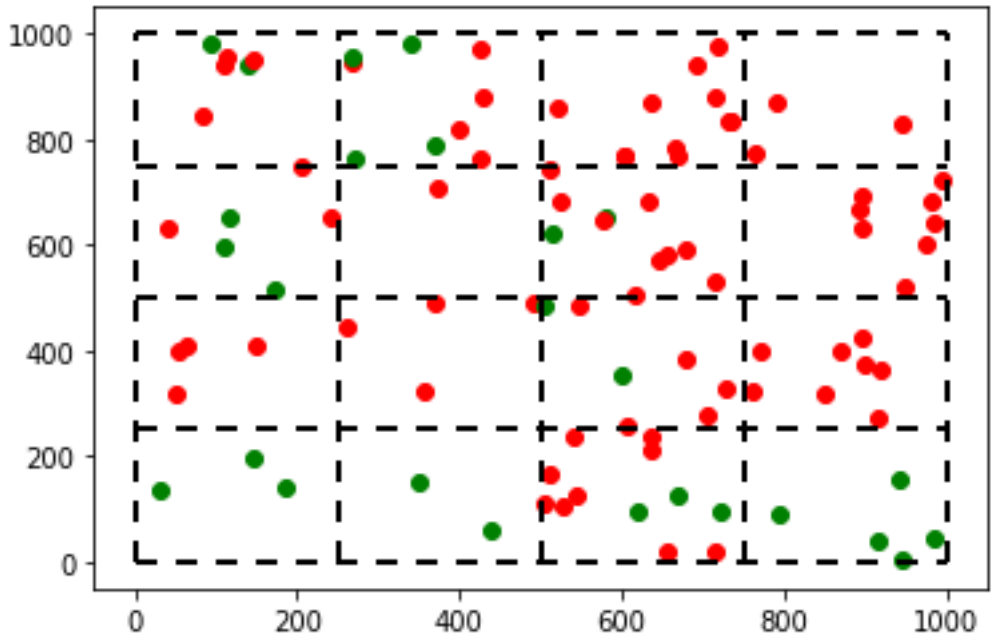
3.3.1 End of First Day



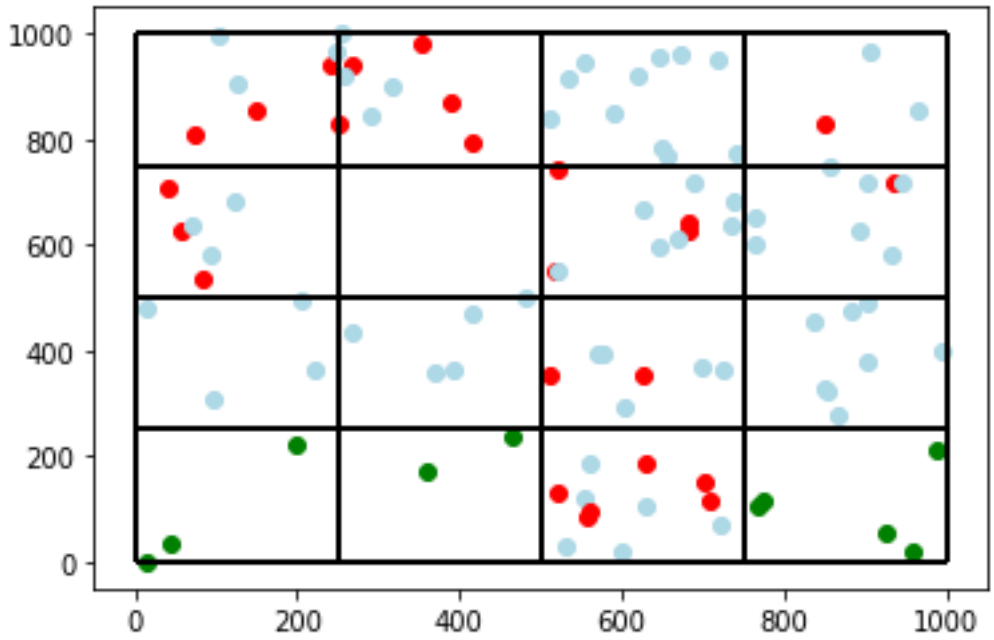
3.3.2 End of Fifth Day



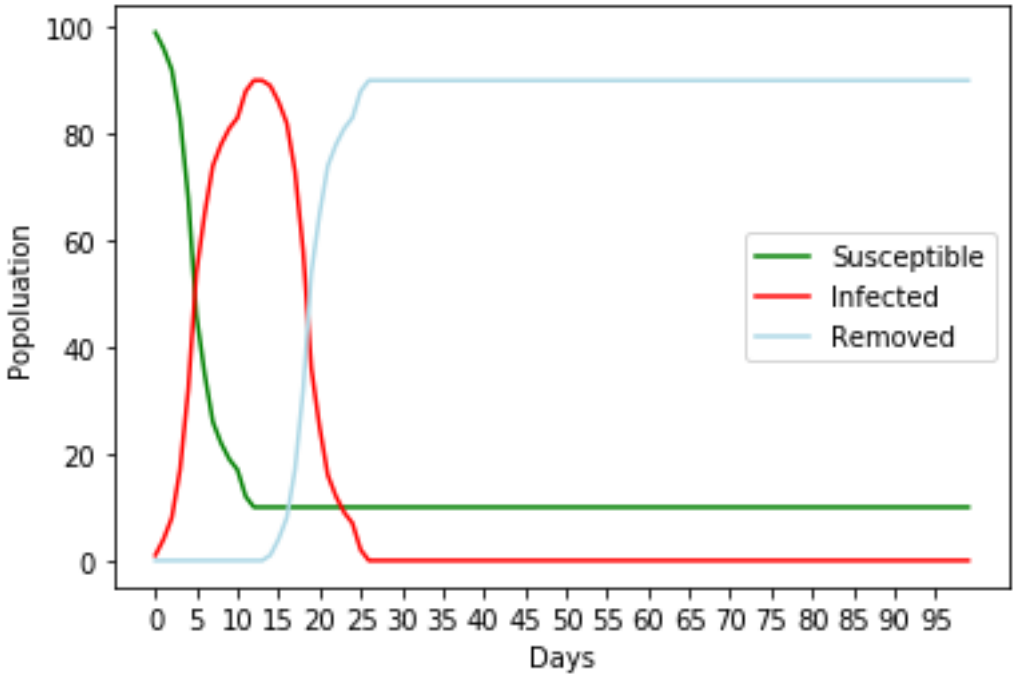
3.3.3 End of Seventh Day



3.3.4 End of 20th Day



3.3.5 Time Series



3.3.6 Comment

We see that a policy intervention that comes too late has little effect on the spread of the virus, only 3 towns are safely isolated in this simulation. A late intervention will not help in the task to reduce the burden on the health system.