Introduction

Mathematical epidemiological models have been a topic of much interest in epidemiology, mathematics, and even history as of late.[1] These models can be used to approximate the complex interpersonal interactions that result in infectious diseases spreading, and their results can be used to plan responses to both current and future epidemics. Alternatively, they can be used in planning health policies, such as new vaccination programs.[2] One of the more popular epidemiological model types is the deterministic one, wherein a population is defined by a system of differential equations that are characterized by certain parameters, such as the rate of infection and recovery. These models are deterministic due to their solutions, numerical or otherwise, being fixed for a given set of initial conditions and parameters, which has made them popular due to their simplicity. In fact, the first such epidemiological models were of this type, stemming from the original one published by Kermack and McKendrick in 1927.[3] In that model, with vital statistics like birth and death rates omitted, a population in the midst of an epidemic is modeled by the following system of differential equations:[3]

$$\left.\begin{aligned}\frac{dS}{dt} &= -\frac{\beta SI}{N}\\ \frac{dI}{dt} &= \frac{\beta SI}{N} - \gamma I = -\left(\frac{dS}{dt} + \gamma I\right)\\ \frac{dR}{dt} &= \gamma I = -\left(\frac{dI}{dt} + \frac{dS}{dt}\right)\end{aligned}\right\} \tag{1}$$

In this system, S, I, and R correspond to the number of susceptible, infected, and recovered people in the population, respectively. N, β, and γ are the system parameters corresponding to the total population, average number of people infected per infected person, and rate of recovery, respectively.[3,4] This model nowadays is better known as the Susceptible-Infected-Recovered, or SIR, model. In the model, susceptible, or vulnerable, people are infected at a constant rate for each infected person in the population. They progress to being infected until they recover, at which point they are immune to all further infections. The SIR model is the simplest of all epidemiological models, but more accurate extensions of it are popular nowadays.

Variations of the SIR model exist, and they can account for different disease properties, such as the SIS model, where recovered people are still susceptible to further infection, like in the case of meningitis or the flu.[5] Alternatively, newer models may take into account the additional vulnerability of young and old people to disease, birth and death rates, or diseases being confined geographically within a population. With the advances in computational resource in recent years, these models can be used to create simulations with non-deterministic solutions. These simulations have become popular for modeling epidemics due to their improved ability to consider the geographical locations of infected and susceptible people. Moreover, these simulations, much like reality, are discrete. Thus, there is no such thing as a non-integer number of people, and outside of taking average results, half a person cannot be infected over the course of any given timestep, unlike the continuous deterministic models. In fact, with modern object-oriented programming, these systems are oftentimes easier to simulate than derive mathematically.[6]

In this project, I sought to use C++ 2017 to program a simulation of a population suffering from an epidemic by using the SIR model as the basis. Then, I planned to use the model to account for the presence of vaccinated people in a community, counting as none of the three types of people defined in Equation Set 1. I then tested three different models for disease propagation following the principles of the SIR model to find which one achieved the most accurate and appropriate output. Using the most appropriate disease propagation model, I then performed a sensitivity analysis on the system to determine first order and second order effects of infection probability and vaccination rate. Lastly, I performed simulations to determine the minimum vaccination rate required for herd immunity, defined as 95% of the unvaccinated population being safe from infection, to take effect.

Methodology

Classes

To begin the project, I developed a set of classes contained within the *infection_classes.h* header file. I first programmed a *person* class to represent each member of the population that had the private member integer *disease_status* that represented whether a person was vaccinated, vulnerable, sick, or recovered. A key of what each value of *disease_status* represented is presented in Table 1.

Table 1. Key of *disease_status* meanings.

***disease_status* Value**	**Meaning**
-2	Person is vaccinated.
-1	Person is recovered.
0	Person is vulnerable.
>0	Person is sick for this many more days.

This key was chosen in order to make writing the *update* method easier and to easily collect information on a person's status for the simulation and debugging. In addition, it allowed me to make my constructor for a person as simple as this:

```
person(int disease_status=0):
       disease_status(disease_status){};
```

Then, for the *update* method, which I used to update the status of sick people to reflect the remainder of their illness, I wrote the following code:

```
void update(){
     if (disease_status > 1)
         disease_status--;
     else if (disease_status == 1)
         disease_status -= 2;
}
```

For this method, it first checks to see if a person will be sick for an additional day. If so, it decrements the number of days remaining in their sickness by one. However, if they are on their last day of illness, it will instead set them as being recovered. I used a similar setup for my *infect* method, which took the number of days to infect someone as the argument, and due to the meaning of *disease_status*, simply set it to that as follows:

```
void infect(int infection_length){
    disease_status = infection_length;
}
```

Lastly, I created three critical methods for the class that played a larger role in the simulation: *is_vulnerable*, *is_infected*, *get_status*. They are as follows:

```
bool is_vulnerable(){
    return (disease_status == 0);
}

bool is_infected(){
    return (disease_status > 0);
}

int get_status(){
    return disease_status;
}
```

I used the *is_vulnerable* and *is_infected* methods to determine whether or not someone could be infected or infect others. These were important for running the checks to allow the disease to propagate. I used the method *get_status*, on the other hand, to obtain statistics of how many people were in each category, which I used to create methods in the *population* class that I used to hold every person in the simulation.

I programmed the *population* class to contain every member of the population as well as statistics on the disease, population information, and simulation constraints. For the sake of readability, I left comments in the following code excerpt defining how each parameter is represented by each private member of the *population* class with the exception of *spread_type* and *symbol_key*, which are better explained in detail alongside their usage in the *population* class's methods.

```
private:
    vector<person> people; //vector of the people in the population of the sim
    int population_count; // number of people in the population
    int infection_length; //how long each infected person will be sick for
    int spread_type;
    int spread_count; //how many people each person has a chance to infect daily
    float infection_chance;  //chance of an interaction causing an infection
    float percent_vaccinated; //fraction of population that's vaccinated
    string symbol_key = "#-?+";
```

With the exception of *symbol_key*, all private members are taken as arguments to the *population* constructor, with default arguments specified in the case of simpler simulations being needed for demonstration purposes. However, the most critical part of the constructor is the initialization of the *people* vector:

```
people.reserve(population_count);
for(int index = 0; index < population_count; index++){
    bool is_vaccinated = (1.0*rand()/RAND_MAX < percent_vaccinated);
    if (is_vaccinated)
        people[index] = person(-2);
    else
        people[index] = person(0);
```

To randomly assign the vaccinated members of the population, I generated a random number between 0 and 1 for each iteration of constructing a *person* object in the *people* vector. If the number was below *percent_vaccinated*, it would construct the person to be vaccinated. Otherwise, they would be constructed to be vulnerable. I set this up so that if 100% of the population was vaccinated, all generated *person* objects would be vaccinated, and if 0% were vaccinated, none of them would be vaccinated. However, due to the random assignment, especially for smaller population sizes, it is possible that additional or fewer members of a given population are vaccinated as intended. While I considered using another implementation involving a for loop and randomly generated indices for the *people* vector after initializing it, I decided not to in the interest of maintaining code readability. Moreover, all simulations I used the *people* class for were run over multiple trials and for a large enough population size that this error would be small and likely be averaged between trials. Lastly, real-world vaccination statistics inherently have error in them that is greater than any error stemming from my choice of method to randomly assign vaccinated members of the population.

The first important class method I programmed was *count_infected*, which as the name suggests, returns an integer representing how many members of the population are currently infected.

```
int count_infected(){
    int infected_count = 0;
    for(int index = 0; index < population_count; index++){
        if(people[index].is_infected())
            infected_count++;
    }
    return infected_count;
```

While I considered keeping a private member to fulfil the same function that would be updated with each infection, I realized it would have greatly increased the difficulty of maintaining the code in the future. I programmed this method so the simulation main programs would have a way of detecting if the disease had died out. I programmed a similar method for counting the number of vulnerable people, *count_vulnerable*, which follows the exact same algorithm with the exception of calling the *is_vulnerable* method of each *person* object in the *people* vector. In contrast to the *count_infected* method, I used the *count_vulnerable* method for herd immunity statistics and to perform my sensitivity analysis later on in order to determine how many vulnerable people were present at the start and end of each simulation.

The next class method I programmed was *population_breakdown*, which used the *symbol_key* class member to return and optionally print a set of symbols representing the status of each member of the population:

```
string population_breakdown(bool print_status=false){
    string breakdown = "";
    for(int index = 0; index < population_count; index++){
        switch (people[index].get_status()){
            case -2:
                breakdown += symbol_key[0];
                break;
            case -1:
                breakdown += symbol_key[1];
                break;
            case 0:
                breakdown += symbol_key[2];
                break;
            default:
                breakdown += symbol_key[3];
                break;
        }
    }
    if (print_status)
        cout << breakdown;
    return breakdown;
}
```

Each symbol in *symbol_key* was set to represent a certain kind of status, with #, -, ?, and + corresponding to the vaccinated, recovered, vulnerable, and sick statuses, respectively. While I could have defined *symbol_key* within the scope of this method, I wanted to set it as a private class member to make it slightly more obvious in case someone wanted to change the key later or expand it for a more robust disease model. For this method, I iterated over the *people* vector and used the *get_status* method of the *person* class to determine which symbol from *symbol_key* to append to the *breakdown* string. Optionally, I allowed the user to specify whether they wanted to print *breakdown* to the console when calling the method. I allowed printing both for debugging purposes as well as to allow users to observe the system changing in real-time if desired. For reference, a sample output for a small population towards the start of an epidemic simulation is given:

```
+???++???????+?+??+?+??#?+????++???+???+?+??#?+??++?+++?#??+?#?
```

This method was particularly important for analyzing the behavior of a *population* object as the simulation progressed, since the strings could be written to a file and processed in Python or another programming language that handles strings well in order to generate numerical data on the population of the system. Alternatively, this method could be altered to return a vector of integers representing the number of each member of the population to reduce the amount of processing required.

The infection methods are the next essential component of the *population* class. The first and only untargeted method, *infect_random*, is straightforward and thus has its code omitted. A random index in the *people* vector is generated, and the corresponding *person* is checked to see if they are vulnerable. If they are vulnerable, the method infects them by using the *infect* method of the *person* class with *infection_length* as the argument. The other infection methods, which I refer to as targeted methods, take an index of an infected population member as an argument. The first of these methods, *infect_spreading*, generates a random index corresponding to a *person* in the *people* vector other than the *person* whose index was supplied to the function as an argument. Then, the following code checks to see if that other person is vulnerable, and if they are, it rolls a check to infect them:

```
if(people[random_index].is_vulnerable()){
    float infection_roll = 1.0 * rand() / RAND_MAX;
    if (infection_roll < infection_chance)
        people[random_index].infect(infection_length);
```

If the check passes, it infects that person using the *infect* method as shown. This process is repeated for a number of times equal to the *spread_count* member of the *population* class. This method best reflects the random sampling assumption given in the basic SIR model.

For the second targeted infection method, *infect_geographical*, a different approach is taken. Instead, the nearest neighbors of the *person* corresponding to the supplied index are checked in a similar process to see if they are infected. To maintain parity between the two methods, the same number of people are checked for a given *spread_count* value. For *infect_geographical*, I implemented this by checking a number of neighbors on each side of the provided index. For example, given an index of 1200 and a *spread_count* of 4, the *person* objects at indices 1198, 1199, 1201, and 1202 in the *people* vector will be checked. As a result, the supplied *spread_count* to the constructor of the *population* class must be divisible by 2 if this method is chosen. To account for an infected *person* at either the start or end of the *people* vector, I programmed a quick check on the index of a neighbor, *neighbor_index*, that effectively makes the *people* vector cyclical for the purpose of determining neighbors:

```
if (neighbor_index < 0)
    neighbor_index += population_count;
else if (neighbor_index >= population_count)
    neighbor_index -= population_count;
```

I did this to prevent undefined behavior from occurring at runtime and to prevent edge effects from affecting simulations using this method.

The last and most complicated of the three targeted infection methods is *infect_hybrid*. This hybrid method performs half of its infection checks on the nearest neighbors of the *person* corresponding to the supplied index and the other half on random people. As a result, *spread_count* for simulations using this method must be divisible by 4. I programmed this method to both account for geographical effects and random interactions. On a day-to-day basis, the average person will interact both with people they see every day, such as family or coworkers, and random people, such as those on public transportation. While I believed this method would establish a more accurate picture of the propagation of diseases, it is still incomplete due to not accounting for repeated interactions or the higher likelihood of infection one may have from closer physical contact with family members relative to strangers.

Lastly, the *population* class has its own *update* method that can be considered to progress the number of days in the simulation. It iterates over the *people* vector and checks each person to see if they are infected. If they are infected, it checks the *spread_type* private member of the *population* class, which determines which type of targeted infection method is called, as shown in Table 2.

Table 2. Key of which targeted infection method is called for a *spread_count* value.

***spread_count* Value**	**Targeted Infection Method Called**
1	*infect_spreading*
2	*Infect_geographical*
3	*Infect_hybrid*

After calling the relevant targeted infection method, the *update* method of the current *person* is called, and the next person is put through the same process until the system ends. A major factor to note is that if a *person* object is infected before they are iterated over, they will be treated as if they are fully contagious and have been such for the whole day. However, this issue is offset by all infected *person* objects having their *update* method called if they are infected, which means that the day will be counted towards the illness time of only people who are infected and allowed to spread the disease during the day. While this factor could have affected the temporal variation of the system, I did not believe that this affected the qualitative data I obtained from my simulations. Overall, setting up these classes was critical to my use of them in creating epidemiological simulations.

Simulations

Simulation Commonalities

I performed four different types of simulations during this experiment with three separate main programs. However, all simulations shared the same parameters for constructing their *population* objects, as shown in Table 3.

Table 3. Common Simulation Parameters.

Parameter	**Value**
Total Population	50,000 people
Initial Number Infected	5 people
Length of Infection	6 days
Number of Daily Interactions of Infected People	4 Interactions

I chose the total population to be 50,000 to create a large enough system to create useful data and to mitigate any poor luck during the random parts of each simulation. I decided to have 5 people be initially infected in each simulation in order to prevent the disease from dying out too quickly due to random factors, and I determined that an infection period of 6 days was close enough to many common real-world diseases. Lastly, I opted to have infected people interact with 4 other people per day in order to examine all three targeted infection methods under equal footing. I decided not to choose 8 interactions per day because it seemed somewhat excessive for the simulations and would have increased simulation runtime, which became a limiting factor for one of the simulations later on.

All simulations began by seeding the random number generator by calling *srand(time(0))* and initializing the common parameters listed in Table 3. Besides that, all simulations also shared the same two code snippets. The first was to generate the first few infected people in the *population* object *ut_campus*:

```
for(int patient_zeros = 0; patient_zeros < initial_infected; patient_zeros++){
    ut_campus.infect_random();
}
```

I did this in order to properly setup the system's initial conditions before I began monitoring the simulation. The other major shared piece of code was a white loop for each simulation:

```
while (ut_campus.count_infected() > 0){
    ut_campus.update();
}
```

This loop was made to run until the disease for a given simulation died out. Depending on the simulation, I added additional code to this loop to update variables and write the data as necessary, but this base structure was carried over for all of them.

Exporting and Analyzing Data

In general, all simulations stored their data in the form of a stringstream. I wrote functions for each simulation tailored to each's desired output data in order to write the data to the stringstream and to save that data to a .csv file once the simulation was complete. For the simplest type of data, the type I used in my sensitivity analysis, I wrote the following *write_data* and *save_data* functions:

```
void write_data(stringstream &my_data, float value){
    my_data << value << '\n';
}

void save_data(stringstream const &my_data, string filename){
    ofstream output_file;
    output_file.open (filename);
    output_file << my_data.str();
    output_file.close();
}
```

I chose to write my data to a stringstream during runtime in order to maintain program performance by minimizing the number of times I had to write to the disk, but it came with the risk that if the program was unable to be run to completion, I would be left with no data because I only wrote the data to a file at the very end of the program. In addition, the *save_data* function can only save files to the folder in which it was run, which made data management inconvenient. While I wanted to create a folder for each simulation, neither of the compilers I used during this project, neither the GNU Compiler Collection (GCC) C++ compiler nor the Intel compiler had the C++ 2017 standard namespace, filesystem, implemented. As a result, I had no way to ensure crossplatform compatibility for any standard I used to create folders, so I opted to just save the output files in the directory each program was run in.

All data was processed and analyzed using Python and a set of assorted libraries including NumPy, Matplotlib, and pandas, among others. For the sake of brevity, the code used to analyze is omitted from this report and is briefly summarized when needed. All Python code has been included alongside my project submission in the *Python* directory. Note that much of the IO operations in the code is hard-coded for directories on my computer and must be edited accordingly to account for other runtime environments.

Individual Simulations

The first set of simulations I performed used the program compiled from the *simple_sim.cpp* file. I tested combinations of infection probability that ranged from 0% to 90% against vaccination rates that ranged from 0% to 99%. For each combination, I ran three trials for each of the three targeted infection methods and monitored the progression of each system over time. With each iteration, I called *ut_campus.population_breakdown()* and wrote the output string to a *data* stringstream alongside an integer *days* that represented the number of times the while loop in the program had been iterated. At the end of each trial, I saved the data to a .csv file that was named appropriately to reflect the independent variables, trial number, and targeted infection method used. I analyzed this data using the *normal_data_analysis.py* file.

The second set of simulations I performed were used to determine the impact of infection probability and vaccination rate on the end state of each simulation, represented by the fraction of vulnerable people who were left uninfected at the end of the simulation and the disease lifetime, or the number of days the disease progressed until there were no infected people remaining. For these simulations, I performed 50 trials on each data point as I incremented the vaccination rate from 0% to 99.5% and the infection probability from 0% to 100% in increments of 0.5%. I wrote the data for each trial to a single .csv at the conclusion of the simulation. I performed this simulation using the program compiled from the *parameter_variation.cpp* file, and I analyzed the data using the *parameter_variation_analysis.py* file.

I performed the third set of simulations using the program compiled from the *herd_immunity_sim.cpp* file. I used these simulations to determine the required vaccination rate in population to achieve herd immunity for a given infection probability, which I defined as meaning that 95% of vulnerable people on average remained uninfected. In the file, I performed three trials per combination of infection probability and vaccination rate, iterating across each from 0% to 100% and 0% to 99%, respectively, in increments of 1%. I stored the results of each trial in the vector *all_vuln_safe*, which I took the average of at the conclusion of the three trials by using the *average* function I wrote:

```cpp
float average(vector<float> data){
    float mean = accumulate(data.begin(), data.end(), 0.0) / (1.0 * data.size());
    return mean;
}
```

In this function, I used the *accumulate* function from the C++ standard library to sum the vector. Note that the final argument to the *accumulate* function is 0.0 instead of 0 due to the function returning an integer for the latter case. I then tested this average against the threshold I set for herd immunity:

```
if (average(all_vuln_safe) > 0.95){
    all_req_vaccines[infection_trial] = (vacc_rate);
    break;
}
```

Here, I used the break statement to break the for loop containing the different vaccination rates. The current vaccination rate, *vacc_rate*, is then written to the current index of *all_req_vaccines*, a vector containing the required vaccination rate for the current infection probability. This process is repeated three more times for the current infection probability before the average of *all_req_vaccines* is taken, and that average value is written to the data stringstream. I analyzed this data using *herd_immunity_analysis.py*.

Lastly, I performed the final simulation using the program compiled from *sensitivity_analysis.cpp* to perform a sensitivity analysis using the Sobol process on the infection probability and vaccination rate with respect to the end state of the simulation. I first had to program a function to read data from .txt files containing the sample input points generated from the Saltelli extension of the Sobol sequence in Python as follows:

```
vector<float> read_input(string filename){
    ifstream file(filename);
    string line;
    std::string::size_type size_thing;
    vector<float> input_data;
    while(getline(file, line)){
        input_data.push_back(stof(line, &size_thing));
    }
    return input_data;
}
```

Unfortunately, I did not have enough time to program properly parsing a .csv file for reading. Instead, this code takes an input of a single filename and reads each line of a .txt-like file with that name and stores the floating-point value of that line in a vector that the function returns. I used this function to read the inputs I generated in Python back into C++. From there, I ran 10 trials per datapoint and averaged them for the output fraction of vulnerable people present at the end of each trial and the disease lifetime of each trial. I saved these files separately to an output .csv that I then analyzed using the *sensitivity_analysis.py* script.

Testing Environment

All tests were performed under an Ubuntu environment on a desktop computer with 16GB DDR3 RAM and an Intel Xeon E3-1240 v3 processor. All code was verified to compile and run correctly using both GCC and Intel C++ compilers. Code was also verified to compile and run correctly using the GCC C++ compiler on Windows.

Data and Results

Baseline Simulations

I first ran a variety of simple simulations using the program compiled from the *simple_sim.cpp* file. While I tested a variety of simulations, the observations I found were largely generalizable to three types of systems: ones with high infection probabilities relative to their vaccination rates, ones with similar infection probabilities to their vaccination rates, and ones with low infection probabilities relative to their vaccination rates. For each case, I plotted the population breakdown of each class as a function of time for the deterministic ODE provided in Equation Set 1 alongside the data from simulations using each of the three targeted infection methods, as shown in Figure 1 for the first case, corresponding to an infection probability of 40% and a vaccination rate of 0%.

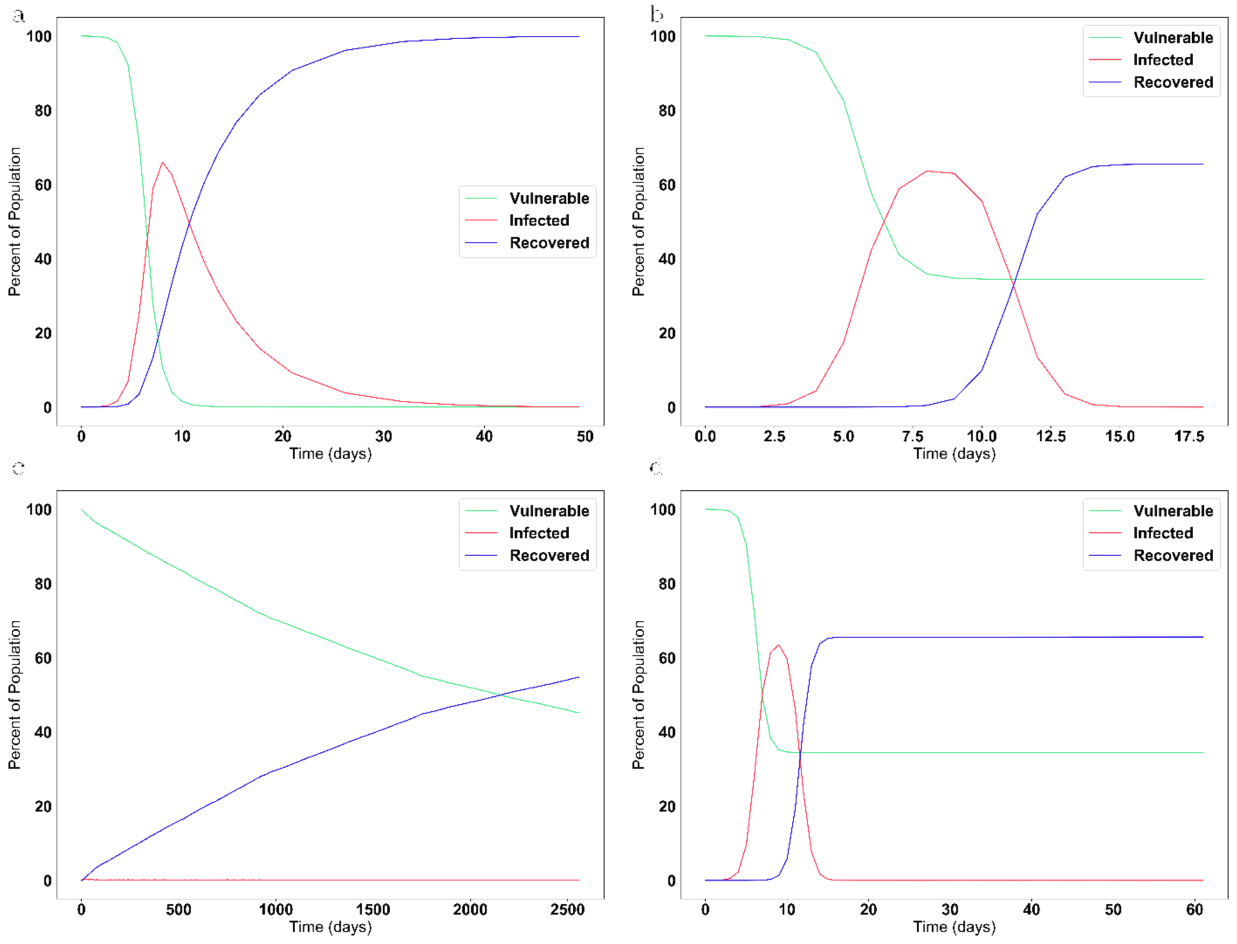

Figure 1: Population breakdown plots for systems with 40% infection probability and 0% vaccination rate in **(a)** a deterministic system, **(b)** a system with the random infection method, **(c)** a system with the geographical infection method, and **(d)** a system with the hybrid infection method.

I was initially surprised by the results in Figure 1(a). No model matched the deterministic one well. I believed that this was due to the temporal issues due to my implementation of the different targeted infection methods. Because I iterated and infected members of the original *people* vector instead of a copy, the model did not accurately represent the simultaneous interactions assumed by the SIR model. Instead, my implementation of the methods effectively split each time step into thousands of smaller ones, resulting in the variance I observed. However, for all systems but the one with the geographical infection method, I observed the number of vulnerable and recovered members of the population behaving as a step function, while the number of infected people behaved somewhat similarly to a Gaussian bell curve. I thought this was due to the number of infected people being low at first and increasing rapidly as infections lead to future infections. Then, I believed that the number of infected people reached a maximum once the number of vulnerable people decreased to the point where it was difficult to find further people to infect. Then, the number of infected people decays as the number of recovered people immune to the disease increase and present further transmission of the disease. I found these observations to be consistent with the mathematical relationships presented in Equation Set 1 as well due to the derivative of the number of people infected increasing alongside the number of vulnerable and infected people until enough infected people become recovered, at which point it changes signs and the decay I observed occurred. The shapes of the three curves also matched those from official epidemiological computational surveys, which I deemed to indicate that my methods were somewhat correct.[6]

Moreover, the deterministic system concluded with every vulnerable person being infected, which I did not expect for a disease with an infection probability of only 40%. However, this result can be explained by using the basic reproductive number of the system, R_0, which is defined as:

$$R_0 = \frac{\beta}{\gamma} \tag{2}$$

For epidemiological models, R_0 is the ratio of the growth of a disease among a population to its decay. For $R_0 > 1$, a disease can be considered to be self-sustaining in deterministic epidemiological models.[7] In the case of the SIR model, I thought this was reflected in the deterministic model resulting in no one being safe from infection. For the other models, I thought this factor due to the discrete nature of the models. In the cases of the models shown in Figure 1(a), 1(b), and 1(d), the infected members of the population peak at roughly 60% of the total population. However, for the non-deterministic models, each person is represented discretely, so their illnesses all end simultaneously after that maximum is attained. For the deterministic model, which is continuous, the duration of a person's illness is based on a mean value, and thus the probability distribution allows the disease to continue infecting people until it completely infects the whole population.

For the non-deterministic models, I found that both the random and hybrid methods behaved similarly. However, the number of infected people in the random method decayed more quickly, which I thought was because the geographical aspect of the hybrid infection method artificially slowed down the spread of the disease. Lastly, the behavior of the purely geographical model was odd initially because it did not behave similarly to the step function I observed all the other models behave like. I thought this was due to the rate of the disease's spread being limited severely by the geographical constraints. This is reflected by the number of infected never growing significantly for that model despite the number of recovered people increasing. I thought this represented the disease spreading at roughly the same pace people recovered due to the geographical constraints, which would have resulted in the number of infected people being roughly constant.

I continued my analysis for a system with 40% infection probability and 20% vaccination rate to account for cases where both factors are of similar magnitude, as presented in Figure 2.

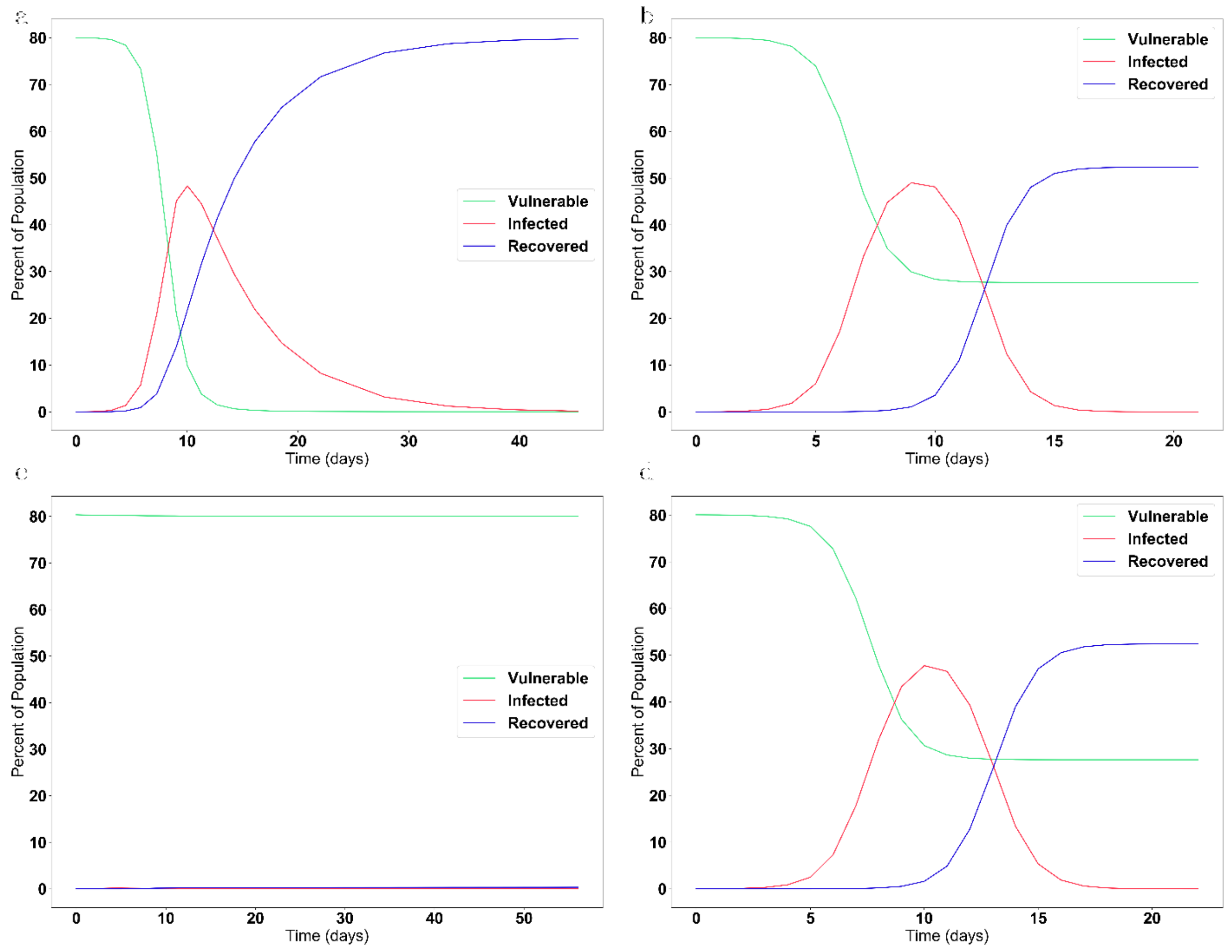

Figure 2: Population breakdown plots for systems with 40% infection probability and 20% vaccination rate in **(a)** a deterministic system, **(b)** a system with the random infection method, **(c)** a system with the geographical infection method, and **(d)** a system with the hybrid infection method. Note that the y-axes of all plots have been adjusted due to the presence of vaccinated individuals.

As shown in Figure 2, I found similar results to the data in Figure 1 with the exception of the geographical infection model. In the case of the geographical infection model, I found that the growth of the disease was almost non-existent, and almost no members of the population were infected. I believed that the presence of vaccinated individuals exacerbated the problems with the geographical model I noted during my analysis of the first dataset. I attributed this exacerbation to the vaccinated individuals acting as an effective wall to the spread of disease, preventing it from moving past them. As a result, once all infected individuals were confined between at least four vaccinated people, the disease died out. This also caused smaller growth to occur in the number of infected for the same reason, blocking regions of infected individuals from infecting other people.

Lastly, I performed an analysis on a system with 90% infection probability and 95% vaccination rate to observe the behavior of each system when the number of vaccinated people was high enough to deter the spread of disease, as shown in Figure 3.

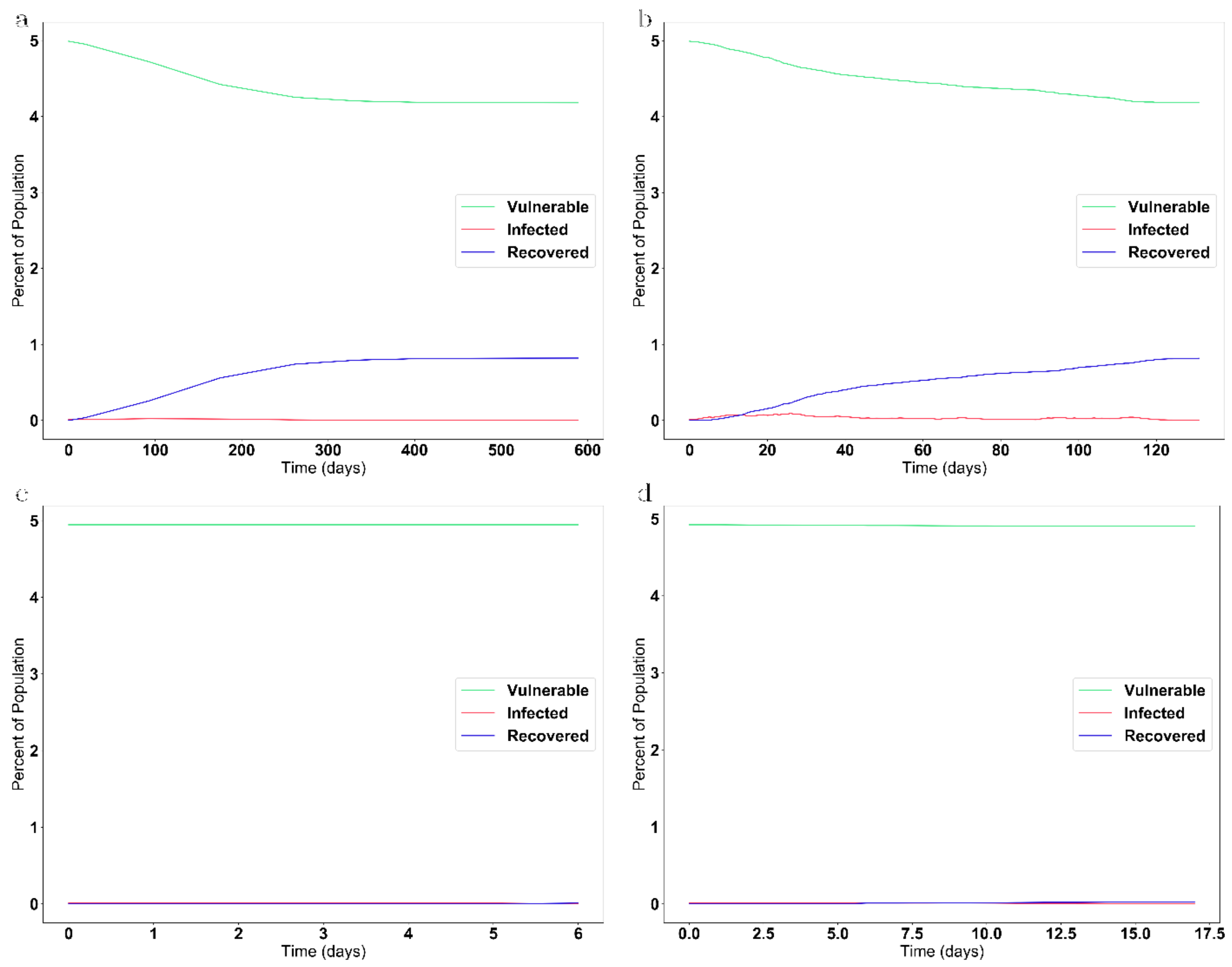

Figure 3: Population breakdown plots for systems with 90% infection probability and 95% vaccination rate in **(a)** a deterministic system, **(b)** a system with the random infection method, **(c)** a system with the geographical infection method, and **(d)** a system with the hybrid infection method. Note that the y-axes of all plots have been adjusted due to the presence of vaccinated individuals.

Due to the jamming effect I mentioned earlier for the geographical model in the presence of vaccinated individuals, I thought that the hybrid model suffered from a similar effect. In contrast, both the deterministic and random infections models performed similarly, instead having the growth and decay of their step functions be reduced significantly as a result of the low infected population reducing the rate at which individuals leave the pool of vulnerable individuals and enter the pool of recovered ones. Qualitatively, I took the agreement between both models as evidence that the random infection method was the most accurate in reflecting the behavior of the deterministic model for most system conditions, and I decided to use it for the remainder of my experiments as a result.

Herd Immunity Analysis

In the next part of the experiment, I modeled herd immunity using the random infection method. I evaluated various infection probabilities and plotted the minimum average vaccination rate required to keep 95% of vulnerable people safe, as shown in Figure 4.

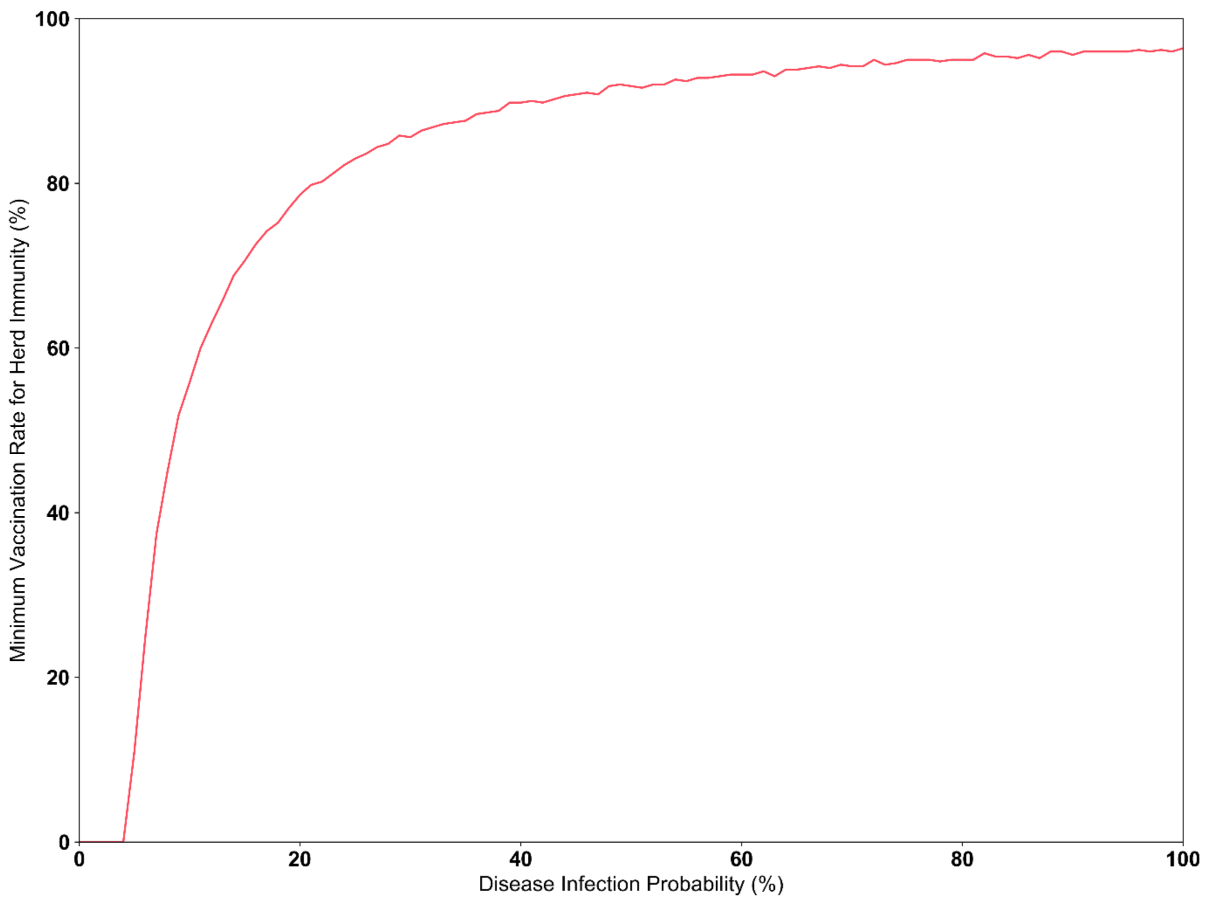

Figure 4: Plot of required vaccination rate to achieve herd immunity as a function of infection probability.

From the plot, I noticed that for low probabilities of infection, diseases will generally exhaust themselves without infecting most vulnerable people. Then, I believed that the rapid increase in the minimum vaccination rate was due to the disease being able to survive for significantly longer once a certain probability of infection was achieved for a given vaccination rate. However, eventually the required number of vaccinations tapers off, which I attributed to herd immunity impeding the disease from infecting people once a critical number of vaccinated people in the population was achieved, with that critical number growing slowly for increasing infection probability. This is reflected by the following relationship for the threshold for herd immunity:[7]

$$\text{Threshold} = 1 - \frac{1}{R_0} \tag{3}$$

Here, the threshold is defined as the fraction of the population that must be vaccinated for herd immunity to come into effect. To verify this, I plotted my data as a function of R_0 alongside the relationship in Equation 3 on the same axis, shown in Figure 5.

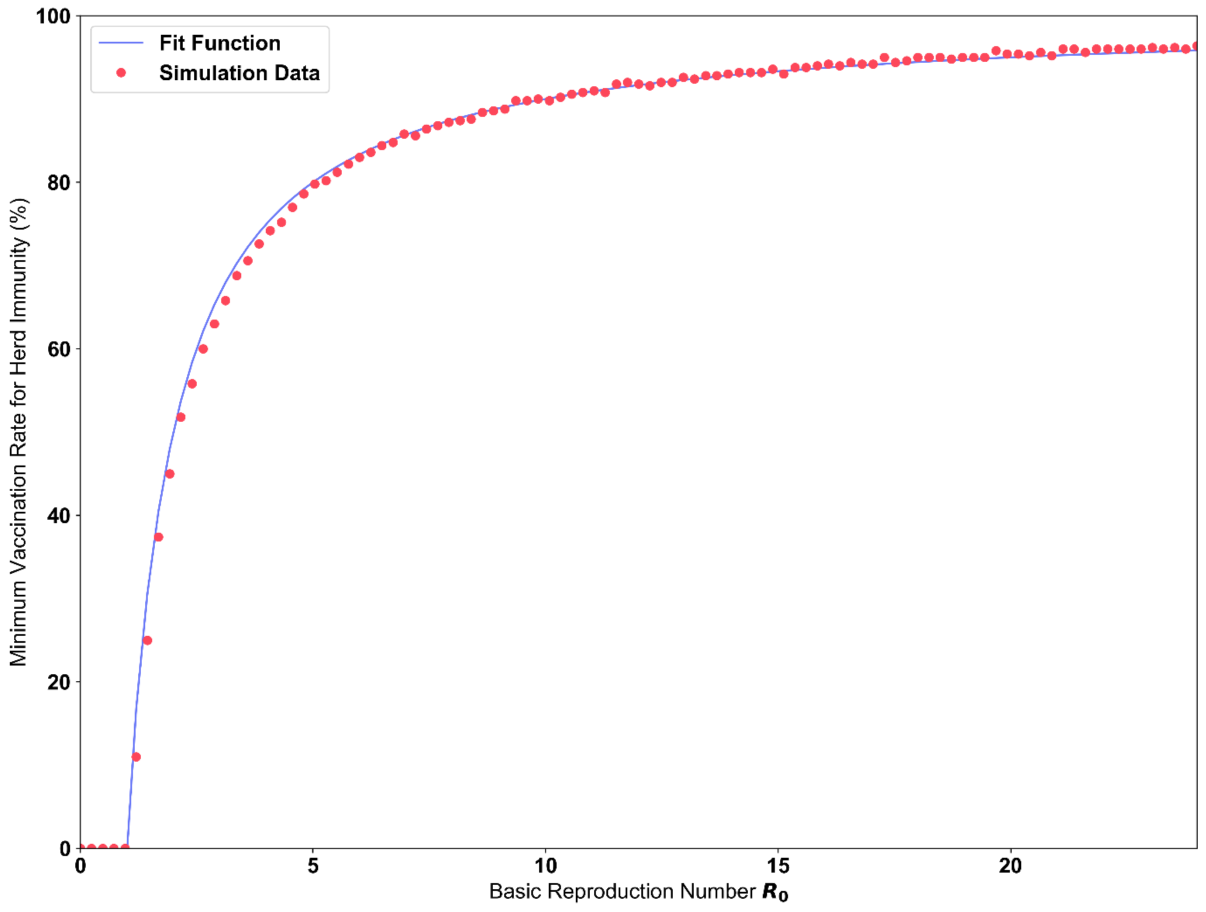

Figure 4: Plot of required vaccination rate to achieve herd immunity against the basic reproduction number of a given disease.

Due to the excellent agreement between the data and the threshold function, I was able to verify that my simulation accurately modeled herd immunity. Moreover, the reason behind the decaying increase in the minimum required vaccination rate became more apparent. With increasing R_0, the disease becomes limited by the number of vulnerable individuals present in the population. As a result, only slight changes to that number are required in order to halt the progression of a disease's spread. More importantly, this indicates that herd immunity is critical in halting the spread of infectious diseases and that vaccinated members of a population are critical in halting the spread of disease.

Parameter and Sensitivity Analysis

To continue my project, I examined the effects of varying the input parameters to the system, the infection probability and vaccination rate, on the end state variables of disease lifetime and fraction of vulnerable people who are uninfected. To begin, I produced a surface plot of disease lifetime of these parameters, as shown in Figure 5.

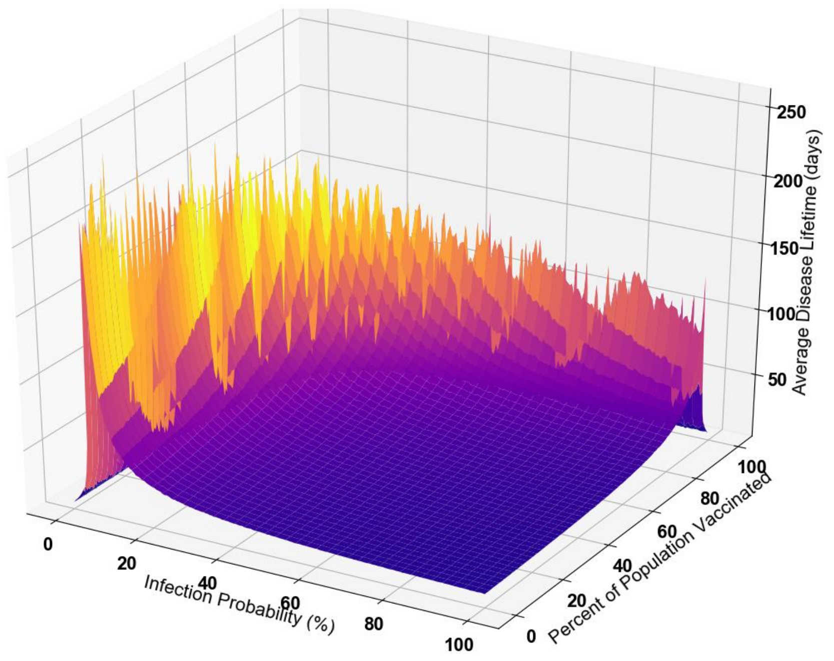

Figure 5: Surface plot of disease lifetime as a function of system input parameters.

To better interpret this data, I plotted disease lifetime for fixed infection probabilities as a function of vaccination rate, shown in Figure 6.

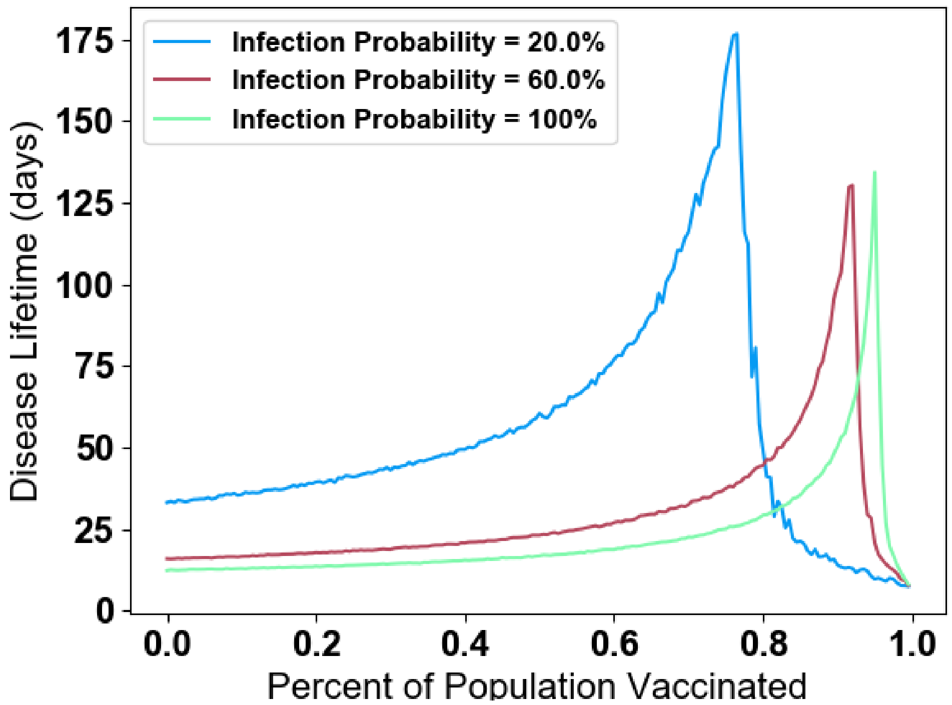

Figure 6: Plot of disease lifetime as a function of vaccination rate for selected infection probabilities.

From Figure 5, I observed the formation of a hump in all three cases. I took this to indicate that the lifetime of a disease was a tug-of-war between both system input parameters. For low infection probabilities or highly vaccinated populations, the disease spreads too slowly and dies quickly. For populations with high probabilities of infection and greater numbers of vulnerable people, the disease instead spreads too quickly and infects almost everyone immediately, causing it to die out quickly as well. In order for a disease to survive in the long-term, it must have its spread limited by a population that is not too vulnerable, but it must be allowed to spread to at least some new people each day in order to survive. This is where the maxima from the surface plot in Figure 5 originated from, corresponding to the maxima shown in Figure 6.

I performed a similar analysis on the fraction of vulnerable population members left uninfected at the end of the simulation to further evaluate the impact of herd immunity. I generated a surface plot of the data, shown in Figure 7.

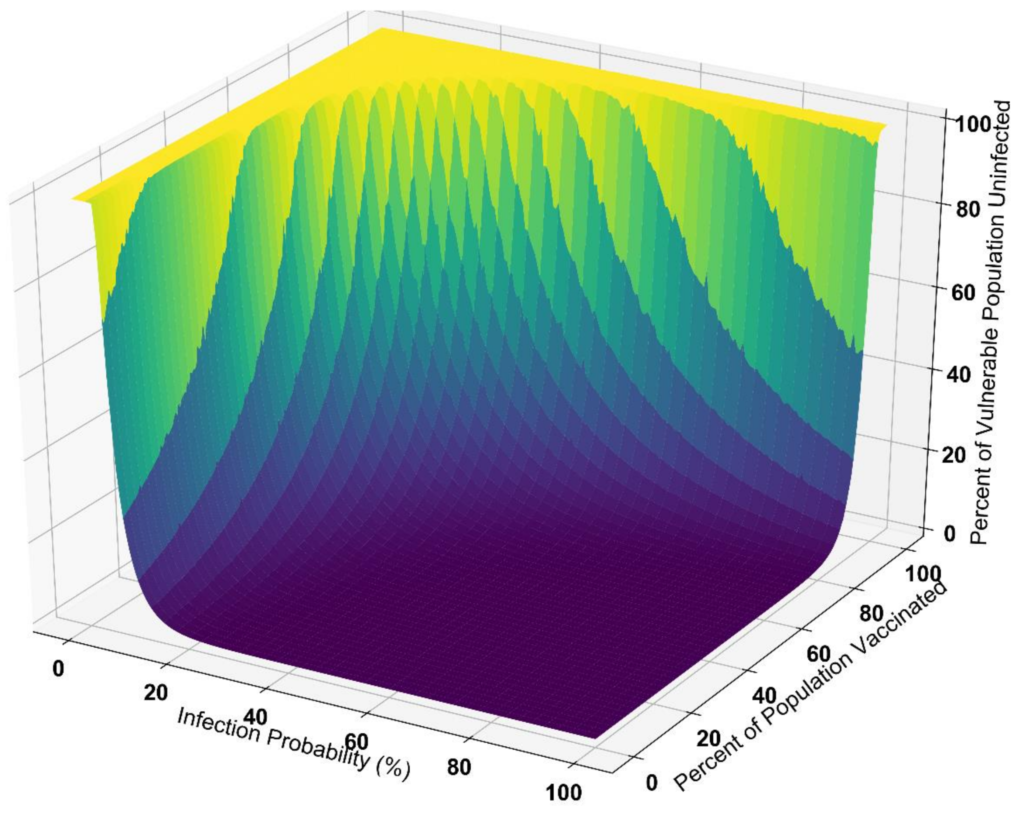

Figure 7: Surface plot of vulnerable population members left uninfected as a function of system input parameters.

As expected, I found that for low infection probabilities, almost no members of a population must be vaccinated in order to keep the vulnerable ones safe. However, as a disease becomes more infectious, the importance of vaccinations increases before tapering off as the infectiousness reaches a limit on how much it increases the speed of diseases spreading. These findings were consistent with my herd immunity analysis, further indicating the importance of herd immunity in protecting vulnerable members of a population from disease.

Lastly, I performed a simple sensitivity analysis on these parameters to determine their impact on the end state of the system. Due to the nature of the Sobol process used the end state variable monitored does not matter–the same result will be produced regardless. I found the first order and second order coefficients describing the impact of each variable on the system. Here, the first order coefficients describe the impact of an individual variable while holding all others constant, while the second order coefficient describes the impact of the independent variables as they relate to each other and their interactions with one another in determining the end state of the system, with all of these coefficients ranging from 0 to 1, with higher values indicating greater system sensitivity. I present these variables in Table 4.

Table 4. Sensitivity Analysis Coefficients

First Order Infection Probability	First Order Vaccination Rate	Second Order
0.183	0.085	0.732

The low first order values for both independent variables indicates a weak reliance on them for the system parameters. Furthermore, the vaccination rate has a lower impact on the system than the other parameters, indicative of how the required vaccination rate to achieve herd immunity tapers off. Lastly, the high second order interactions were consistent with the behavior of disease lifetime changing drastically as a function of both input parameters, indicating that the interactions between both were significant in determining the final state of the population.

References

(1) Lang, J. C.; De Sterck, H.; Kaiser, J. L.; Miller, J. C. Analytic Models for SIR Disease Spread on Random Spatial Networks. *J. Complex Networks* **2018**, *6* (6), 948–970.

(2) Huppert, A.; Katriel, G. Mathematical Modelling and Prediction in Infectious Disease Epidemiology. *Clin. Microbiol. Infect.* **2013**, *19* (11), 999–1005.

(3) Kermack, W. O.; McKendrick, A. G. A Contribution to the Mathematical Theory of Epidemics. *Proc. R. Soc. A Math. Phys. Eng. Sci.* **1927**, *115* (772), 700–721.

(4) Weisstein, E. W. Kermack-McKendrick Model http://mathworld.wolfram.com/Kermack-McKendrickModel.html (accessed Dec 8, 2019).

(5) Hethcote, H. W. Three Basic Epidemiological Models. In *Applied Mathematical Ecology*; Levin, S. A., Hallam, T. G., Gross, L. J., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 1989; pp 119–144.

(6) Holme, P. Model Versions and Fast Algorithms for Network Epidemiology. *J. Logist. Eng. Univ.* **2014**, *30* (3), 1–7.

(7) Fine, P.; Eames, K.; Heymann, D. L. “Herd Immunity”: A Rough Guide. *Clin. Infect. Dis.* **2011**, *52* (7), 911–916.