Epidemic Modeling Techniques for Smallpox

by

Cory Y. McLean

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

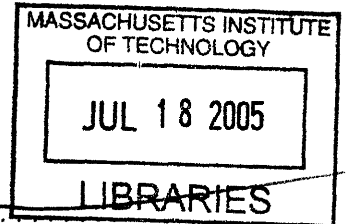
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 17, 2004 [September 2004]

©2004 Cory Y. McLean. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.



Author ....

Department of Electrical Engineering and Computer Science

August 17, 2004

Certified by ....

Adam Szpiro

MIT Lincoln Laboratory

Thesis Supervisor

Certified by ....

Lucila Ohno-Machado

HST Affiliated Faculty

Thesis Supervisor

Accepted by ....

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

This work was sponsored by the United States Air Force under Air Force Contract No. F19628-00-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

ARCHIVES

Epidemic Modeling Techniques for Smallpox

by

Cory Y. McLean

Submitted to the Department of Electrical Engineering and Computer Science
on August 17, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Infectious disease models predict the impact of outbreaks. Discrepancies between model predictions stem from both the disease parameters used and the underlying mathematics of the models. Smallpox has been modeled extensively in recent years to determine successful response guidelines for a future outbreak. Five models, which range in fidelity, were created for this thesis in an attempt to reveal the differences inherent in the mathematical techniques used in the models. The disease parameters were standardized across all models. Predictions for various outbreak scenarios are given, and the strengths and weaknesses of each modeling technique are discussed. The mixing strategy used greatly affects the predictions of the models. The results gathered indicate that mass vaccination should be considered as a primary response technique in the event of a future smallpox outbreak.

Thesis Supervisor: Adam Szpiro
Title: MIT Lincoln Laboratory

Thesis Supervisor: Lucila Ohno-Machado
Title: HST Affiliated Faculty

Acknowledgments

I would like to thank Dr. Adam Szpiro for being my thesis supervisor. In addition to posing the original research idea, he provided comments along the way that helped guide my research.

I would also like to thank Professor Lucila Ohno-Machado for being my faculty thesis supervisor. Her questions and suggestions helped me improve the quality of my thesis writing dramatically.

Professor Markus Zahn, my academic advisor, provided me with guidance and help during my entire four years at MIT on all topics, ranging from classes to recommendations to thesis struggles. He played an active role in helping me finish my thesis and I owe him a big thanks for being an excellent advisor and friend.

Laura Petrillo gave me amazing support on the thesis, and was a source of encouragement, motivation, empathy, and numerous writing suggestions. She helped me more than I could have ever hoped for.

My parents have constantly supported me in everything I do, and for that I cannot thank them enough.

Contents

1 Introduction **15**

2 Background **19**

2.1 Smallpox disease stages 19

2.2 Intervention policies and methods 20

2.3 Modeling approaches 21

2.3.1 Transition modeling strategies 22

2.3.2 Interactions between people 28

2.3.3 The vaccination queue 29

3 Models implemented **31**

3.1 Description of attack scenario 31

3.1.1 Untraced states 34

3.1.2 Queue states 35

3.1.3 Quarantine state 35

3.1.4 Traced but unsuccessfully vaccinated states 35

3.1.5 Immune/recovered and dead states 36

3.1.6 Disease parameters of the models 36

3.2 The agent-based model 36

3.2.1 Mixing strategy 38

3.2.2 Computer implementation of the agent-based model 39

3.3 The individual-based model with heterogeneous mixing 41

3.3.1 Mixing strategy 42

3.3.2 Computer implementation of the individual-based model with heterogeneous mixing 43
3.4 The individual-based model with homogeneous mixing 44
3.5 The general transitions model 45
3.5.1 Approximations necessary 46
3.5.2 Computer implementation of the general transitions model . . 48
3.6 The ordinary differential equation population-based model 49
3.6.1 Computer implementation of the ordinary differential equation population-based model 52

4 Results **53**
4.1 Test results 53
4.2 Validation techniques and results 60

5 Analysis and Discussion **63**
5.1 Model differences 63
5.1.1 Transition probability generation 63
5.1.2 Mixing strategy 67
5.1.3 Vaccination queue removal 68
5.1.4 Contact tracing 68
5.2 Altering disease parameters 68
5.2.1 The number of initial infections 70
5.2.2 The basic reproductive rate 71
5.2.3 The contact naming accuracy 72
5.2.4 The length of time before intervention commences 73
5.3 The computational cost of each model 73

6 Future Work **75**

7 Conclusion **77**

A Simulator source code **81**

A.1 The agent-based simulator 81

A.2 The individual-based simulator with heterogeneous mixing 90

A.3 The general transitions simulator 97

A.4 The ODE simulator 102

B Helper classes **105**

B.1 Person 105

B.2 Building 110

B.3 Random number generator 112

B.4 Transition probabilities 113

B.5 Output averager 115

B.6 Log file printer 117

List of Figures

2-1 Exponential transitions 25

2-2 Gaussian transitions 27

3-1 The queue transition strategy of the general transitions model 50

3-2 The mathematical representation of the ODE model 51

4-1 The susceptible populations with no intervention 54

4-2 The recovered population with no intervention 54

4-3 The dead population with no intervention 54

4-4 The susceptible populations with traced vaccination after five days . . 56

4-5 The recovered population with traced vaccination after five days . . . 56

4-6 The dead population with traced vaccination after five days 56

4-7 The susceptible populations with traced vaccination after 25 days . . 57

4-8 The recovered population with traced vaccination after 25 days 57

4-9 The dead population with traced vaccination after 25 days 57

4-10 The susceptible populations with mass vaccination after five days . . 58

4-11 The recovered population with mass vaccination after five days 58

4-12 The dead population with mass vaccination after five days 58

4-13 The susceptible populations with mass vaccination after 25 days . . . 59

4-14 The recovered population with mass vaccination after 25 days 59

4-15 The dead population with mass vaccination after 25 days 59

4-16 The susceptible populations in a 300 day simulation 61

4-17 The recovered population in a 300 day simulation 61

4-18 The dead population in a 300 day simulation 61

5-1 The disease generations of recovered individuals 64
5-2 Uniform transitions 65
5-3 The susceptible populations with uniform and Gaussian transitions . 66
5-4 The recovered population with uniform and Gaussian transitions . . . 66
5-5 The dead population with uniform and Gaussian transitions 66
5-6 The susceptible populations of the ODE model and the corresponding individual-based model 69
5-7 The recovered populations of the ODE model and the corresponding individual-based model 69
5-8 The dead populations of the ODE model and the corresponding individual-based model 69
5-9 The effects of altering model parameters 71
5-10 The death predictions of the model with $R_0 = 3.0$ and $R_0 = 6.0$ . . . 72
5-11 The effects on infectious people of waiting 5 or 25 days before traced vaccination 74

List of Tables

2.1 Transition model properties 22

2.2 Model interaction properties 22

3.1 Disease states 34

3.2 Smallpox outbreak parameters 37

3.3 Disease rates for the ODE model 50

3.4 Notational abbreviations 52

5.1 Changing parameters and their effects on outbreak impact 70

Chapter 1

Introduction

Infectious disease models help policymakers to decide what type of response is most likely to be successful in stopping an outbreak. Because of its capabilities as a bioterrorist agent [26], smallpox has been modeled extensively in recent years. The models, which range in fidelity, use historical data gathered from various smallpox outbreaks to predict the magnitude of a future attack and determine a successful response. Predictions of outbreak impact vary between models. Consequently, different types of responses are recommended by each model. The lack of agreement between models stems from two sources: different model parameters and different modeling approaches. This thesis analyzes the predictions of three standard modeling approaches to determine their implications in terms of recommended response strategies. Two intermediate models are introduced to both aid analysis of the standard modeling approaches and present possible modeling alternatives. Disease parameters are standardized across all models so different predictions can be attributed entirely to the modeling approach used.

Disease modeling is based on the idea that diseases have several different states. One of the simplest models has three states: susceptible, infectious, and removed. In the model, susceptible people become infectious through contact with infectious people. Infectious individuals are eventually removed from the model, which indicates that these individuals have either developed immunity or died. Most models use the principles of this simple model, but increase the realism by extending it to have more

states.

Models vary in fidelity and computational complexity. A frequently used, low-fidelity model is the ordinary differential equation population-based model. This model treats everyone in each particular disease state equally, and fractions of each population transition to different states in accordance with the equations that describe the model. In contrast, the stochastic individual-based model with homogeneous mixing analyzes each person individually, and individual transitions between states are governed by the probabilistic chance that an individual will transition given the state of the entire population at that time and the amount of time the individual has spent in the current state. This model is more complex than the population-based model, but each person still has an equal chance of interacting with any other person in the entire population. The highest-fidelity model implemented for this thesis is a stochastic agent-based model. Agent-based models simulate actual interactions between subjects as realistically as possible—adults go to work, children go to school or daycare centers, people interact more frequently with their neighbors than with strangers, etc. Heterogeneous mixing occurs as a result of the social network modeling. A generalized transition population-based model (a population-based intermediate model) bridges the gap between the population-based and individual-based models by grouping each state population by the time spent in the state and altering transition rules accordingly. A stochastic individual-based model with heterogeneous mixing (an individual-based intermediate model) determines whether the mixing strategy is the key difference between individual-based and agent-based models by employing a reasonable heterogeneous mixing strategy. The intermediate models attempt to replicate the higher-fidelity results of the agent-based and individual-based models without incurring the increase in computational complexity associated with the higher-fidelity models. A summary of the models' characteristics is given in Tables 2.1 and 2.2.

The practice of modeling transitions between states becomes more complex when intervention methods are introduced. Intervention methods included in the models are quarantine, isolation, and vaccination. Both isolation of symptomatic individuals and vaccination of at-risk individuals can greatly reduce the impact of an outbreak if

administered effectively. Quarantine is largely a preventative measure implemented to ensure that possibly infected individuals do not spread the disease further. There are two main vaccination strategies, traced vaccination and mass vaccination. Traced vaccination involves the identification and vaccination of contacts of infected individuals. Vaccination of the entire community at the first sign of an epidemic is known as mass vaccination.

Chapter 2

Background

The administration of routine smallpox vaccinations ceased in the United States in 1972, and the World Health Organization declared smallpox eradicated in 1980 [22]. Only two known stores of smallpox remain, one in the United States and one in Russia [26]. Clandestine stores of the live virus are assumed to exist, however, and the limited herd immunity of people in the United States makes smallpox an effective bioterrorist weapon [12]. In a contrast to anthrax, which is not contagious, smallpox has a basic reproductive rate R_0 between three and six [5, 23]. This means that each primary infected individual (or index case) will infect between three and six other people on average in a fully susceptible population if no efforts are taken to reduce infection spread. The actual reproductive rate, R, is slightly lower because of residual immunity in the population and decreases further as the disease spreads. Still, to minimize the effects of a smallpox outbreak, infection control measures must be swift and effective.

2.1 Smallpox disease stages

According to the Centers for Disease Control and Prevention (CDC), individuals infected with smallpox transition through a number of disease stages while the disease runs its course. Initially, exposure to smallpox is followed by a seven to seventeen day incubation period in which subjects feel normal and are not contagious. Vaccination

of infected individuals provides immunity if administered during the first three to seven days of the incubation period. Subjects then transition into the prodromal state, and can exhibit symptoms such as fever, malaise, head and body aches, and sometimes vomiting. The prodromal state lasts two to four days, in which subjects are mildly contagious. The final state is characterized by a pustular rash which develops first in the mouth and then spreads to the face, arms, and legs within 24 hours. The rash gradually forms into scabs which fall off 15–25 days after the onset of the rash. Subjects are contagious throughout the rash state. When all scabs have resolved, the disease has run its course and subjects are no longer contagious. The mortality rate of smallpox is roughly 30%, and survivors are often left with pitted scars on the face and extremities [6].

2.2 Intervention policies and methods

There are three main ways to reduce the spread of smallpox: quarantine, isolation and vaccination. Quarantine involves restricting the movement of those presumed to be exposed to a disease [8] either until the people express symptoms or for a generation of the disease, so a lack of symptoms is reasonable proof that a quarantined individual is disease-free. The isolation of infected individuals, in designated hospitals or other facilities, minimizes the likelihood that the individuals will spread the disease to the susceptible population. Vaccination builds immunity to the disease by inoculating individuals with a similar but less harmful pathogen. Vaccination of infected individuals can reduce the effects of smallpox if administered early in the incubation state. For smallpox, vaccination of susceptible individuals results in immunity in nearly 98% of cases [29, p. 10936]. As mentioned in Chapter 1, there are two main vaccination strategies, traced vaccination and mass vaccination. Both vaccination policies are examined in this thesis.

In the traced vaccination policy, symptomatic individuals are immediately isolated and questioned to determine the people with whom they have been in recent contact. Those contacts are then traced and vaccinated as quickly as possible. Edward Kaplan

et. al. performed a study in which he determined that traced vaccination is effective only if what he calls the "race to trace" can be won consistently [29]. As Kaplan et. al. put it, "even if a contact is identified immediately on detection of the infecting index case, the time from infection of the contact to detection of the index (and tracing of the contact) can exceed the time during which the infected contact remains sensitive to the vaccine" [29, p. 10935]. Rapid determination of infected contacts is critical to bring the outbreak under control, since the vaccine is only effective in infected individuals for three to seven days [6].

Mass vaccination, on the other hand, is a strategy in which every person in the population is vaccinated immediately after the first symptomatic people are isolated. No contact tracing is necessary in this policy since the entire population is vaccinated, regardless of each person's contacts. In mass vaccination, resource constraints such as the number of vaccinators and vaccinator efficiency limit the number of people treated each day.

A study has shown that traced vaccination "would prevent more smallpox cases per dose of vaccine than would mass vaccination" [30, p. 1342], but since the United States has enough of the smallpox vaccine to administer a shot to everyone in the country, the focus of this study is to discover ways to minimize the effects of an outbreak without regard to the wastefulness of the policy.

The main objection to a mass vaccination policy stems from the danger of the vaccine itself. The smallpox vaccine is comprised of a live virus of the pox family, and causes death in approximately 1 out of every 1 million people vaccinated. A national call for vaccination would thus cause roughly 300 deaths due to vaccine complications.

2.3 Modeling approaches

The two main differences between modeling approaches are the disease transition model used and the way people are assumed to interact with each other. Table 2.1 indicates the transition modeling properties of each of the five models. Table 2.2 gives the interaction strategy of each model. People mix in either a homogeneous or

Table 2.1: Transition model properties

Model	Granularity	Prediction	Transition probabilities
Agent-based	Individual-based	Stochastic	Arbitrary
Individual-based with heterogeneous mixing	Individual-based	Stochastic	Arbitrary
Individual-based with homogeneous mixing	Individual-based	Stochastic	Arbitrary
General transitions	Population-based	Deterministic	Arbitrary
ODE	Population-based	Deterministic	Exponential

Table 2.2: Model interaction properties

Model	Mixing strategy	Vaccination
Agent-based	Heterogeneous	Queue
Individual-based with heterogeneous mixing	Heterogeneous	Queue
Individual-based with homogeneous mixing	Homogeneous	Queue
General transitions	Homogeneous	Approximate
ODE	Homogeneous	Approximate

heterogeneous manner. The vaccination queue can be treated either as a first-come, first-served structure or as a structure that requires approximations.

A simplified model, introduced here for the purpose of illuminating the key features of disease modeling, is the SIR model [18]. The SIR model has only three states: susceptible, infectious, and removed. Susceptible individuals can only transition to the infectious state, and do so if the disease is transmitted to them through an infectious person. Infectious people only transition to the removed state, and do so after the disease has run its course. Removed people do not transition out of the removed state, which represents either immunity or death.

2.3.1 Transition modeling strategies

Accurate determination of the transition modeling strategies between states is essential for effective disease modeling. In general, transitions out of a state are idiosyncratic to both the disease being modeled and to the particular state. With this

flexibility, models are able to produce any type of behavior. The arbitrary transition behavior can be captured by individual-based models and the general transitions population-based model—the models in which there is knowledge of how much time a person has spent in his or her current state. The low-fidelity ordinary differential equation population-based model (the ODE model) is incapable of implementing arbitrary transition behavior because the population in each state holds no information on how long it has been in the state.

The ODE SIR model represents transitions by a set of ordinary differential equations. The rate of change of each state population is represented by an equation similar to those in Equations 2.1–2.3 [14].

$$\frac{dS}{dt} = -\beta SI \tag{2.1}$$

$$\frac{dI}{dt} = \beta SI - rI \tag{2.2}$$

$$\frac{dR}{dt} = rI \tag{2.3}$$

In this model, β represents the infection rate of the disease, and r represents the disease rate (r^{-1} is the average time an individual remains infectious before transitioning into the removed state). The population transitions follow the mean field of exponential distributions.

As model fidelity increases to individual- and agent-based models, each individual is treated separately and calculates his or her own probability of transitioning states during each timestep, which can be dependent upon not only the state of the entire population but also the amount of time the individual has spent in his or her current state. The general transitions population-based model divides the population in each state into subpopulations that represent the people who have spent a specific amount of time in the state. This allows the general transitions model to use arbitrary transition probabilities as well, even though it is a population-based model. Typically, a disease has a specified range of time in which a person remains in each state, and the exact parameters used in the model determine its behavior. The transition strategy

for state I is given in Equation 2.4.

$$P_t(I \rightarrow R) = \Pr\{\text{Transition from } I \text{ to } R \,|\, \text{time spent in state, } \Delta t\} \tag{2.4}$$

An interesting property of the ODE model is exposed when individual-based models are created that replicate the output of the ODE model. To calculate the transition probability of leaving the state, the solutions to the ordinary differential equations of the ODE model must be determined. The analysis of the difference between the models is best highlighted through an example.

Exponential transition probabilities

Assume the time that people remain infectious is normally distributed with a mean of four days and a standard deviation of one day for the disease being modeled. The ODE SIR model thus has the following differential equation represent the transition out of state I:

$$\frac{dI}{dt} = -\frac{1}{4}I \tag{2.5}$$

The solution to this equation is, for some constant A:

$$I(t) = Ae^{-\frac{1}{4}t} \tag{2.6}$$

Thus, if the initial population in state I is I_0, after one timestep dt the population that remains in state I will be $I_0 \cdot e^{-\frac{1}{4}dt}$. Thus the fraction of the population that transitions out of state I during the timestep is $1 - e^{-\frac{1}{4}dt}$. After the second timestep the population in state I will be $(I_0 e^{-\frac{1}{4}dt})e^{-\frac{1}{4}dt}$. The fraction of the population at time dt that transitions in the second timestep again equals $(1 - e^{-\frac{1}{4}})$. Using induction we see that this fraction remains constant regardless of the timestep in which the population is evaluated. Thus, in the individual-based model, the probability of a person transitioning out of state I on any given day is independent of how long the

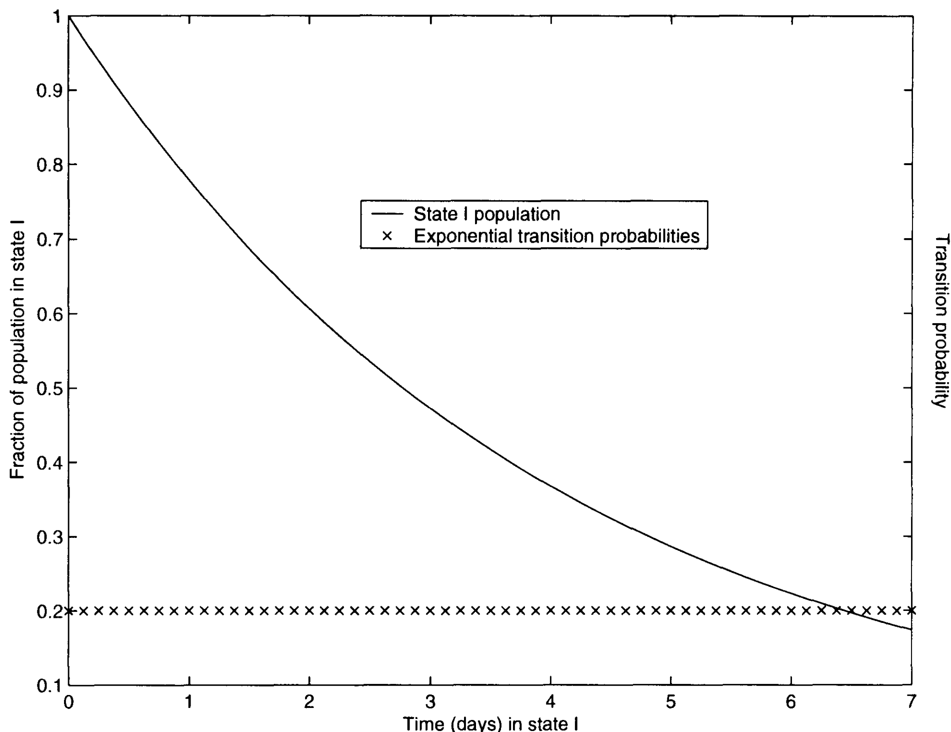


Figure 2-1: Exponential transitions

person has been in state I, and instead is simply constant and equal to $1 - e^{-\frac{1}{4}dt}$, where dt is the timestep of the model. A graph of the transitions produced by such probabilities is given in Figure 2-1 for a timestep $dt = 0.8924$.

Gaussian transition probabilities

The individual-based models and the general transitions population-based model, unlike ODE models, can implement transmission probability generators that give other results. The CDC website [6] gives bounds for time spent in each state in a form like "Prodrome Duration: 2 to 4 days". Thus, the probability of transition out of the prodrome state should depend on the time spent in the state. There are many probability functions that can produce such behavior. Gaussian distributions are used in this thesis. It should be noted that the Gaussian distribution assumption may not exactly represent the true distribution of transition time out of each smallpox disease state. The effects of using other distributions are given in Section 5.1.1.

A Gaussian or normal distribution on a variate x with mean μ and standard deviation σ has the following probability function [3]:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.7}$$

The transition probabilities that produce normally-distributed transitioning populations are dependent upon the amount of time the person has spent within the state. Calculation of the transition probabilities requires a rudimentary understanding of probabilistic analysis. The probability of transitioning during a given timestep t is equal to the area under the curve $P(x)$ between t and $t + dt$ divided by the area under the curve $P(x)$ from t to ∞. In other words,

$$P_t(I \rightarrow R) = \frac{\int_t^{t+dt} P(x)\,dx}{\int_t^{\infty} P(x)\,dx} \tag{2.8}$$

For the first step, we integrate all the way from $-\infty$ to dt, since the area under the tail is negligible for most disease distributions. Since there is no closed-form solution to these integrals, the erf function [1] is used to determine the transition probability. $erf(x)$ is defined below, and comes from integrating the standard normal distribution (normal distribution with mean 0 and standard deviation 1):

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\,dt \tag{2.9}$$

The transition probabilities are thus governed by the following equations (2.10–2.12), where $n \times dt = t$.

$$\frac{erf(\mu - (n-1)\times dt) - erf(\mu - n\times dt)}{\frac{1}{2} + erf(\mu - (n-1)\times dt)} \qquad \text{if } n\times dt \leq \mu \tag{2.10}$$

$$\frac{erf(n\times dt - \mu) - erf((n-1)\times dt - \mu)}{\frac{1}{2} - erf((n-1)\times dt - \mu)} \qquad \text{if } (n-1)\times dt \geq \mu \tag{2.11}$$

$$\frac{erf(\mu - (n-1)\times dt) + erf(n\times dt - \mu)}{\frac{1}{2} + erf(\mu - (n-1)\times dt)} \qquad \text{otherwise} \tag{2.12}$$

The transition probabilities and the population that remains in the state after

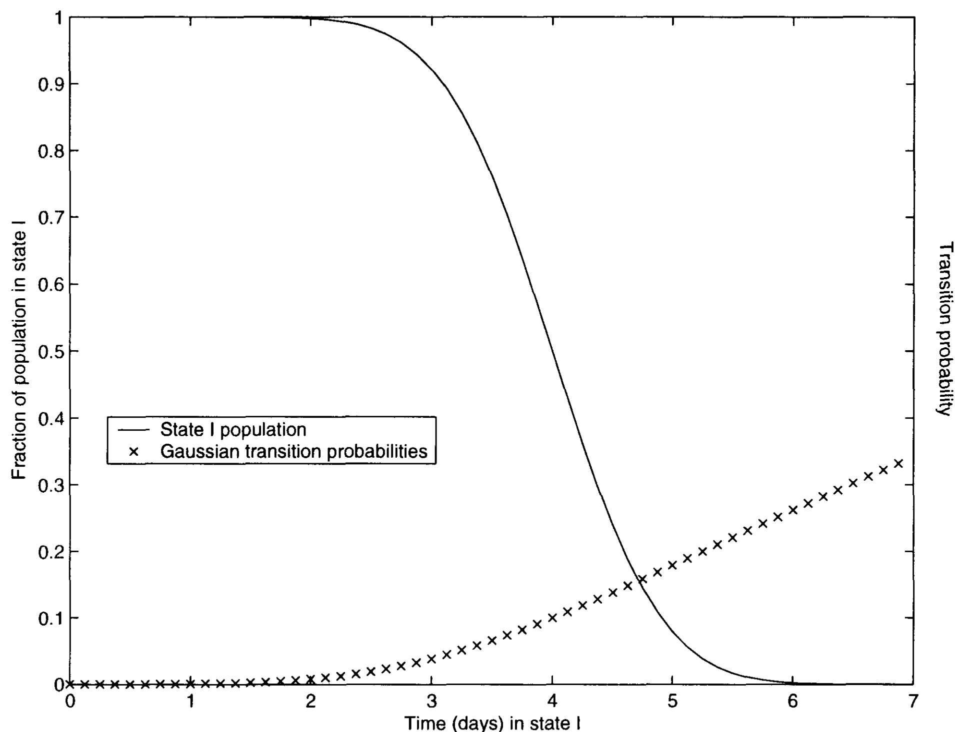


Figure 2-2: Gaussian transitions

each timestep under those probabilities are given in Figure 2-2 for a timestep $dt = 0.0208$.

As Figure 2-2 shows, the transition times of the population are different than in the ODE model. In fact, the use of arbitrary transition probabilities allows any transition time curve to be generated.

It is important to keep in mind that because the individual-based models use (pseudo)random numbers to generate transitions, the actual transitions (and consequently the actual outputs) of the individuals in the models may vary between runs. The graphs displayed here represent the expected behavior of the model, not the output of any actual simulation. The averaged behavior of a number of simulation runs will produce transition curves that become closer and closer to the ideal output graphed above as the number of runs is increased.

In summary, the strategies of transition between states of a model play a large part in determining the model's output. ODE models are limited to exponential distribu-

tions. These exponential distributions, while sometimes able to capture the general spread of an epidemic, are largely unrealistic when applied to single individuals within the population. Individual-based models can implement arbitrary transition probabilities which cause individuals in the model to transition between states in accordance with empirical evidence, but are more computationally intensive.

2.3.2 Interactions between people

Smallpox is spread through personal contact. Consequently, the way that people interact in a model affects its prediction. There are two types of mixing strategies, homogeneous and heterogeneous. In the homogeneous mixing strategy, each person is equally likely to encounter any other person during each timestep. Typically, population-based models and some individual-based models use homogeneous mixing. Under this mixing assumption, disease transmission may be overestimated since in reality people tend to interact with nearly the same set of acquaintances each day. Agent-based and higher-fidelity individual-based models address this concern by implementing heterogeneous mixing strategies, possibly causing the disease to become localized in certain subpopulations rather than spread throughout the entire community.

The differences between individual-based and population-based models are made further apparent in the traced vaccination policy. In traced vaccination, symptomatic people list their recent contacts, who are immediately vaccinated. If infected contacts are found quickly enough, the vaccine is effective and the outbreak is stopped. Since traced vaccination involves tracking individual people, it is treated very differently in the two types of models.

Individual-based contact tracing

Individual-based models have the ability to explicitly model interactions between people. Thus, when an infectious person **A** infects a susceptible person **B**, **B** can be explicitly recorded as being infected by **A**. Similarly, all the individual's contacts

(people with whom the individual interacts but does not necessarily infect) can be recorded. This ability allows an individual to determine exactly which people should be queued for vaccination when he becomes symptomatic—his contact list is scanned and those people are queued for vaccination. The individuals singled out for vaccination can be examined to determine their respective disease stages at the time of tracing. These calculations give insight into the effectiveness of traced vaccination, as they allow an exact determination of the percentage of contacts who were still sensitive to the vaccine upon insertion into the vaccination queue.

Population-based contact tracing

In contrast to the individual-based models, the population-based models implemented in this thesis do not explicitly model interactions between people. Instead, the population-based models rely on approximations based on disease parameters to predict what fractions of populations in each disease stage should enter the vaccination queue during each timestep. These approximations are central to the predictions given by the population-based models under a traced vaccination scheme. The approximations will be analyzed in Section 5.1.4.

2.3.3 The vaccination queue

The vaccination queue is treated quite differently in the individual-based models and the population-based models. The individual-based models create a single large queue and transition people accordingly. If two or more people enter the queue during the same timestep, they are placed in the queue randomly with respect to each other but behind every person who has already spent time in the queue.

The population-based models do not have the ability to differentiate between individual people in the queue. The queue is broken into four separate states that represent queued people in different stages of infection—(0) susceptible, (1) vaccine-sensitive, (2) vaccine-insensitive, or (3) infectious. People in all other disease stages are not eligible for vaccination. To imitate the individual-based queue transitions,

the population-based model assumes that the fraction of the entire queue population to be vaccinated during a single timestep is proportional to the relative numbers of people in the queue. In other words, if N people transition out of the queue during a given timestep, $N \times \frac{Q_j}{Q}$ transition from state Q_j, where Q_j represents one of the four queue states and $Q = \sum_{j=0}^{3} Q_j$.

Chapter 3

Models implemented

This chapter introduces the population and attack scenario considered in this thesis (Section 3.1) and describes in detail the five models implemented (Sections 3.2–3.6).

3.1 Description of attack scenario

The attack scenario modeled for this thesis consists of 101,355 people in Cambridge, Massachusetts, who undergo their daily routines while a subpopulation of ten individuals infected with smallpox travels in their midst. Once the outbreak is identified (the delay time is a parameter of the model that varies the predicted attack impact), intervention methods including quarantine, isolation, and either traced or mass vaccination are introduced into the model. Vaccination occurs in hospitals, and isolation in hospitals and other designated buildings. Quarantine is implemented in homes and designated areas.

Once an outbreak is identified, the entire population becomes immediately aware of it through news sources (television, radio, internet, etc.). The Centers for Disease Control and Prevention has laid out a set of guidelines for response to a smallpox outbreak which calls for "surveillance and containment" (traced vaccination) of symptomatic individuals and their contacts [7]. Although the possibility of moving to large-scale (mass) vaccination is not ruled out, public health resources are used most efficiently in the "surveillance and containment" strategy [29]. The official CDC

policy calls for traced vaccination that switches to mass vaccination after two generations of the disease spread if the outbreak has not been contained [7]. Thus, the model provides implementations of both traced and mass vaccination. The vaccine is administered only by highly trained nurses since it requires a bifurcated needle. Vaccinations are performed in the eight Cambridge hospitals.

The disease is spread from infectious to susceptible individuals through close personal contact. The CDC website indicates that "direct and fairly prolonged face-to-face contact is required to spread smallpox from one person to another" in most cases, but can also be spread "through direct contact with infected bodily fluids or contaminated objects such as bedding or clothing" [6]. Aerosol deployment can spread the virus in rare cases. Hence, a primary objective of smallpox modeling is to determine a strategy to simulate interactions between people realistically.

The stages of the disease were described in Section 2.1. In the model, untraced infected individuals transition through the different disease stages in order, and ultimately either recover and become immune to further smallpox infection or die. When an infected individual becomes overtly symptomatic, he or she is isolated from the rest of the population to limit the spread of the disease. The symptomatic individual is questioned to determine his or her recent contacts to whom the disease might have spread. Those contacts may or may not have been infected by the individual, but all of them are tracked down and ordered to receive the smallpox vaccine. The response to the recent severe acute respiratory syndrome (SARS) epidemic demonstrated that complete isolation of symptomatic individuals is a reasonable assumption.

The heightened awareness of the public to the threat of a bioterrorist attack and the distinctive symptoms of smallpox make it highly unlikely that an overtly symptomatic person would remain within the mixing population. Thus, the model assumes that infected people will only spread the disease while in the prodromal stage. The prodromal stage is thought to be less contagious than the rash stage, however. Consequently, the published estimates of the smallpox R_0 being between 3.5–6 [23] and even spiking up to 20 in certain outbreaks [22] could be overestimates of the contagiousness of smallpox in a future outbreak. An estimated R_0 of 3.0 was used in

the base case of each model as a result. It is important to note that the assumption that individuals only spread the disease while in the prodromal state biases the results in favor of mass vaccination, since the effective percentage of contacts traced would increase dramatically if individuals were only infectious during the rash stage. The effects of an increase in contact naming accuracy is given in Section 5.2.3.

Vaccination occurs at the hospital in a first-come, first-served manner, since all contacts are presumed to be equally at-risk. Since checking into the hospital and filling out the requisite paperwork takes time, each individual is forced to remain in the hospital for an hour before receiving a vaccination shot. If an individual begins to exhibit overt smallpox symptoms, he or she is immediately isolated regardless of whether he or she had already received the vaccine. If an individual who received the vaccine exhibits a fever, he or she is remanded to quarantine for a maximum of sixteen days (the average time it takes a person who becomes infected to exhibit symptoms of smallpox). If the person becomes symptomatic during the quarantine period, he or she is isolated immediately.

If an individual survives vaccination, he or she is assumed to be immune to the disease. The person is thus allowed to move freely within the population again. However, the vaccine has a rate of effectiveness of roughly 97.5% [29], so a small fraction of vaccinated individuals remains susceptible to infection. Also, individuals infected before receiving the vaccine may be insensitive to the vaccine, in which case they will continue to progress through the disease stages and be able to infect others. Once infected individuals become symptomatic, they are isolated until the disease has run its course.

The seventeen states of the model are taken from an ordinary differential equation population-based model by Kaplan et. al. [29]. The states correspond to the disease stages described in Section 2.1, but are further subdivided to indicate whether a person has been traced and vaccinated or not. The states are shown in Table 3.1 and are described in detail below.

Table 3.1: Disease states

State name in Kaplan et. al. paper	**Potential Treatment Stage**	**Name used in this thesis**
Unexposed	Untraced, queued for vaccination, vaccinated	Susceptible
Asymptomatic, non-infectious, vaccine-sensitive	Untraced, queued for vaccination, vaccinated	Disease stage 1
Asymptomatic, non-infectious, vaccine-insensitive	Untraced, queued for vaccination, vaccinated	Disease stage 2
Prodromal and infectious	Untraced, queued for vaccination, vaccinated, quarantined	Disease stage 3
Symptomatic	Untraced, isolated	Disease stage 4
Immune		Recovered
Dead		

3.1.1 Untraced states

Untraced susceptible

Individuals in the untraced susceptible state S^0 transition in two ways: by becoming infected or by being identified as a contact of a symptomatic individual. The number of people that transitions from susceptible to infected during each timestep depends upon the disease transmission rate, the number of infectious people in the population, and the number of susceptible people in the population. The number of untraced susceptibles that transitions into the vaccination queue during each timestep depends upon the fraction of the entire population that the untraced susceptibles represent, the number of contacts generated by each symptomatic individual, and the number of people becoming symptomatic. The people that are queued correspond to susceptible people who are identified as contacts of an infectious individual—people the infectious individual thought he might have infected but actually did not.

Other untraced states

The other untraced states represent infected people in the various disease stages who have not been found and vaccinated yet. These untraced states gain and lose people

through natural disease progression. The number of people transitioning depends upon the average time spent in the state as well as the number of people in the state. The asymptomatic states also transition subpopulations that become queued for vaccination. Some of the subpopulations leave due to incorrect identification by symptomatic individuals and others are correctly identified by their true indexes of infection.

3.1.2 Queue states

Vaccination and quarantine commence after a specified amount of time, τ. People in the vaccination queue transition through disease stages naturally. People in the prodromal disease state who exhibit a fever are immediately quarantined. Vaccination accounts for the rest of the transitioning population. Only susceptible people and those in stage 1 of the disease can develop immunity from smallpox in the model. Vaccinated people in other disease stages become traced but remain infected.

3.1.3 Quarantine state

Quarantine is reserved for people waiting to receive a vaccination who exhibit prodromal symptoms when evaluated. Quarantined individuals remain so for an average of sixteen days, until they become symptomatic and are isolated. In this manner, they are unable to infect others but continue their progress through the disease stages.

3.1.4 Traced but unsuccessfully vaccinated states

Traced susceptible

The traced susceptible population is comprised of susceptible individuals who underwent unsuccessful (but not fatal) vaccinations. Since the smallpox vaccine is not 100% effective, a small fraction of the vaccinated population remains susceptible after vaccination. Those vaccinated individuals mix freely through the entire population and are thus susceptible to infection.

Other traced states

People progress naturally through the traced and infected states in an identical fashion to the other infected states. Individuals enter each state through natural transitions or from ineffective vaccination and leave each state through natural transitions. Since the traced populations believe the vaccination was successful, the populations mix freely and consequently the traced infectious individuals may transmit the disease to susceptible people.

3.1.5 Immune/recovered and dead states

The immune/recovered population consists of people who underwent successful vaccination and people who survived the disease. The dead population is comprised of people who suffered from vaccine-related deaths and people that died from the disease. Individuals thus transition into these states from only the vaccination queue states and the symptomatic and isolated states.

3.1.6 Disease parameters of the models

Because of the eradication of smallpox in 1980, there is no recent data regarding the parameters of the disease (transmission rate, morbidity and mortality, disease stage rates, vaccine efficacy, etc.). Historical data from outbreaks in the 1970s and earlier provide estimates of these critical parameters but may not be entirely accurate. In particular, the transmission rate of smallpox may be different now due to technological advances in travel and public transportation methods and the high concentrations of people in cities. A summary of the disease parameters is given in Table 3.2. References for all parameters values are cited in the table.

3.2 The agent-based model

The agent-based model is the highest-fidelity model implemented for this thesis. The model attempts to replicate the actions of Cambridge inhabitants as realistically as

Table 3.2: Smallpox outbreak parameters

Parameter	Description	Value	Source
R_0	Basic reproductive rate	3.0	[6]
c	Names generated per index	50	[29]
p	Fraction of infectees named	0.5	[29]
N	Population size	101,355	[9]
n	Number of vaccinators	50	[9]
μ	Service rate	50 vacc's/(vaccinators ·days)	[29]
h	Fraction febrile in stage 3	0.9	[29]
v_0	Vaccine efficacy, stage 0	0.975	[6]
v_1	Vaccine efficacy, stage 1	0.975	[6]
δ	Smallpox death rate	0.3	[6]
f	Vaccination fatality rate	10^{-6} people/vaccination	[6]
$I_1^0(0)$	Initial number infected	10 people	In text
τ	Time before intervention	5 days, 25 days, never	[29]

possible. To do so, data from the 2000 census [9] was used along with an agent-based model of human interactions and movement created by Ronald Hoffeld at the Massachusetts Institute of Technology Lincoln Laboratory [11] to determine accurate building and population requirements. The model generates between 55,000 and 60,000 workers, roughly 11,000 children attending school, and 25,000 to 30,000 college students attending MIT, Harvard, and Lesley. Each day, every person probabilistically decides his or her actions depending upon the day of the week and the time of day. On a typical weekday, a worker goes to work from 9 to 5, decides whether to eat at home or at a restaurant, decides whether to go to the movies or return home, and finally returns home to bed. Students attend school during the day and then may go out as well, returning to their homes or dorms in the evening. On weekends people may go to recreational parks, and each person's propensity for dining out and attending movies increases.

In all three individual-based models, the disease transitions within an infected person attempt to follow the description of smallpox given in Section 2.1. All disease transitions between states are Gaussian in nature, with two standard deviations encompassing the published transition times. For example, the CDC website [6] in-

dicates that an infected person is vaccine-sensitive for three to seven days. In the agent-based model, this is represented by Gaussian transitions with a mean of five days and a standard deviation of one day.

Transitions out of the vaccination queue and the susceptible states are not Gaussian. The transitions out of the vaccination queue depend upon the queue workers' effectiveness and the length of the vaccination line. Transitions out of the susceptible states are based on whether the susceptible person came in contact with an infectious person during the timestep. The probability of an infectious person transmitting the disease to a susceptible person is calculated to be consistent with the value of R_0 used in the model, and assumes that infectious contacts are spread uniformly over the infectious period.

3.2.1 Mixing strategy

In the agent-based model, there is no need to explicitly determine a mixing strategy. People interact with each other during their everyday routines. Interactions occur as a result of the routine modeling rather than the other way around. Since people go to the same workplace and home every day, they see mostly the same people each day. Random connections are made at restaurants, theaters, and parks. The goal of the mixing strategy was to create a reasonable "small-world network" [32] representation of the Cambridge inhabitants. The resulting connectivity graph (represented by people as nodes and contacts as edges) should be a "randomized network", which is a regular network with a limited number of random connections. "Scale-free" networks are also of interest in smallpox modeling, but were not implemented for this thesis. Both network structures are discussed below.

Small world networks

"Small-world" networks have been the subject of much recent study. The term refers to large, sparse graphs of short characteristic length and a large degree of clustering [32]. Randomized and scale-free networks are two varieties of small-world net-

works.

Randomized networks can be created in a variety of ways. Work by Watts and Strogatz showed that a regular network could maintain virtually the same clustering coefficient but drastically reduce its characteristic path length by introducing a small number of random connections between vertices [32]. The agent-based model creates a small-world network in this manner. Clusters are formed in each family and workplace, and random connections are formed at restaurants, theaters, and recreational parks.

Scale-free networks are "characterized by an uneven distribution of connectedness. Instead of the nodes of these networks having a random pattern of connections, some nodes act as "very connected" hubs [13]." The scale-free networks exhibit a short characteristic path length but the implications of these networks to disease modeling are great. Diseases spread more widely when the hubs are targeted [15].

3.2.2 Computer implementation of the agent-based model

In the agent-based model, each person is a separate object. Each person has a unique identification number, a list of contacts, a current state indicator, the time spent in the current state, a home indicator, a work indicator, and a list of family members.

Similarly, each building in Cambridge is a separate object. There are 44,858 buildings and they are split between houses, dorms, restaurants, theaters, schools, colleges, offices, and hospitals in accordance with the census. Each building has a maximum occupancy, a unique identification number, and a list of the people who are currently inside it. The identification number is used to enable people to return to the same homes and work buildings every day. The building occupancy limits the number of people who can be inside the building at any time—restaurants that become filled cause patrons to go elsewhere, much like in real life. Transportation modeling is omitted.

Each day, every person moves between his or her assigned home to work and back, and also may stop at restaurants and movie theaters. While in each building, a person may interact with other people inside the same building. Interactions are

recorded by each person and an infected individual's history is examined when he or she becomes symptomatic. The individual's contacts are then notified and ordered to enter the hospital for vaccination. An individual who receives a vaccination returns home immediately afterward and remains there for the rest of the day. Since the vaccine is a live virus, special care must be taken to avoid spread of the virus to others. People may choose to remain at home longer than a single day. However, returning to work or school the following day is a reasonable assumption of the average behavior of the population and is thus implemented in the agent-based model.

Disease propagation occurs through contact between infectious and susceptible people. To represent the disease having an $R_0 = 3.0$, the average behavior of an agent must be noted, to determine the probability that contact between an infectious and susceptible person leads to infection. Contact is made when individuals are in the same building at the same time. In the model, agents made an average of 183 contacts in a three-day period without intervention. The model parameters indicate that each symptomatic person names 50 contacts, and that contacts are named with a 50% probability. Thus, each agent should only make 100 contacts in a three-day period to satisfy the model assumptions. Consequently, when individuals were in the same building at the same time, the probability that they became contacts was reduced to $\frac{100}{183} \approx 0.55$. After the reduction, 57% of contacts were co-workers, 37% of contacts were random, and 6% of contacts were family. Co-worker contacts lasted an average of eight hours a day, random contacts lasted an average of 2.5 hours a day, and family contacts lasted an average of 13.5 hours a day. The total average number of contact timesteps in a three-day period is roughly 3,800. Thus, the probability of infecting one contact in one timestep is equal to $\frac{R_0}{3{,}800} \approx 0.00079$. The fact that family members and co-workers are more likely to become infected than random individuals is taken into account since the individual remains in the buildings with those contacts a larger portion of the time.

In mass vaccination, while at the hospital each agent should come into contact with only a subset of the entire population in the vaccination queue. If that were not the case, the agent-based model would predict an inflated outbreak impact since the

average number of people contacted in the hospital would be much greater than usual. To overcome this effect, the number of contacts a person has in the hospital is limited to 27, since 27 contacts $\times$ 72 hours $\times$ 2 timesteps/hr $= 3,888$ contact timesteps. Thus the $R_0 = 3.0$ constraint is maintained. This assumption is justified by the fact that all 101,355 people in the vaccination queue would not mix equally in real life. The hospital contacts are generated uniformly over the entire population. The homogeneous mixing assumption causes most infections to occur in the hospital. Further study of actual hospital mass vaccination strategies is needed to justify the assumption.

3.3 The individual-based model with heterogeneous mixing

The individual-based model with heterogeneous mixing is a lower-fidelity attempt at agent-based modeling. In this model, each person is a distinct entity in the community and people interact in a limited manner, much like in the agent-based model. Estimations, rather than explicit models of the day-to-day behavior of each person, are used to determine how people interact with one another in the individual-based model with heterogeneous mixing. Buildings and homes are not created in the individual-based models since precise movements and interactions between people are not simulated. Since entire population movements are not explicitly modeled, the individual-based model with heterogeneous mixing is less complex and requires less computation than the agent-based model. These benefits motivate the use of the individual-based model with heterogeneous mixing.

Disease progression within each individual is modeled identically in this model and the agent-based model. The vaccination strategy used is identical as well since neither disease progression nor vaccination depends upon interpersonal interactions. Thus, since the only difference between the models is the way in which people interact, individual-based models with heterogeneous mixing require carefully chosen mixing

strategies to produce accurate predictions. Since both the movement of people and the disease spread possesses inherent elements of randomness, the agent-based model and the individual-based model with heterogeneous mixing may produce different predictions of outbreak impact.

3.3.1 Mixing strategy

The determination of a good mixing strategy was the most critical and difficult design decision of the individual-based model with heterogeneous mixing. The results of the agent-based model provide a good basis to determine a mixing strategy consistent with the model assumptions.

In the agent-based model, interactions between people are limited by the geographical structure of the model. To emulate this structure, the individual-based model with heterogeneous mixing can arbitrarily limit the people with whom an individual may interact to a subset of the population. People in the individual-based model with heterogeneous mixing do not actually travel to different buildings and come in contact with one another. Instead, contacts are generated probabilistically and are limited to a subset of the entire population. An accurate modeling structure can be determined by careful examination of the agent-based model. The individuals in the agent-based model interact heavily with their family members and co-workers, and infrequently with strangers at restaurants, movie theaters, and parks. The average family size in the agent-based model is 2.59 and increases to 10.21 when college dorm occupants are considered "family." The average company size is eighty people, and each person interacts with only a subset of the entire company. To emulate the agent-based model, "clusters" of people that represent companies are created by restricting the co-worker identification numbers to be similar. Random contacts are chosen uniformly over the entire population and families are not implemented.

3.3.2 Computer implementation of the individual-based model with heterogeneous mixing

As in the agent-based model, each person in the individual-based model with heterogeneous mixing is a separate object. Each person has a unique identification number, a list of contacts, a current state indicator, the time spent in the current disease state, and a record of who infected him or her (if he or she is no longer susceptible). Disease propagation occurs through contact between infectious and susceptible people.

The mixing strategy described above was implemented wholly through the use of each person's unique identification number. The initially infected individuals were picked in a uniformly random manner and were each given contacts that encompassed most of the population. This decision emulates the probable attack in which the initially infected individuals attempt to infect as many different populations as possible. All other individuals gather their co-worker contacts uniformly from the "office-representing" cluster of people with similar identification numbers. Persons with ids between 0-79 are a cluster, 80-159 are a cluster, etc. A co-worker contact of person P is generated using the formula below.

$$New\ Contact = floor(\frac{P.getId()}{80}) \times 80 + floor(80 \times (uniform(0,1))) \tag{3.1}$$

A check is made to ensure that people do not generate themselves as contacts. Random contacts are chosen uniformly from the entire population and represent contacts made at restaurants, theaters, or parks.

The percentage of contacts that were co-workers was found in the following manner. In the agent-based model, family and co-workers comprised 63% of the agent's contact list, and the other 37% of contacts were random. However, an agent interacted with his or her family and co-workers much more frequently than with strangers, and consequently was more likely to infect family or co-workers than random contacts. There are a number of ways to approximate this behavior in the individual-based model with heterogeneous mixing. One way makes the probability of infect-

ing a co-worker be higher than the probability of infecting a random contact, and making each individual's contact list have the same percentage of each type of contact as in the agent-based model. A different solution is to simply ignore the fact that co-workers and family are more likely to become infected and treat everyone equally. The method implemented for this thesis approximates the behavior of the agent-based model without requiring different infection probabilities. The fraction of contact timesteps, rather than just contacts, was calculated for each type of contact. Family contacts comprised 13.5 hours/day $\times$ 2 timesteps/hour $\times$ 3 days $\times$ 6 contacts = 486 contact timesteps. Co-workers made 2,736 contact timesteps, and random contacts made 555 contact timesteps. Thus, the percentage of total contact timesteps in a three-day period spent with random contacts was $\frac{555}{3777} \approx 15\%$. Consequently, random people make up 15% of the contacts for each person in the individual-based model with heterogeneous mixing, and co-workers make up the other 85% of each person's contact list. Since each person has 100 contacts, an infectious person has a $\frac{3.0 \; infections/3 \; days}{100 \; contacts \times 144 \; timesteps/3 \; days} \approx 0.000208$ probability of infecting one of his contacts in a single timestep.

The vaccination line is treated as a queue at a single hospital and is implemented using the vector class. The queue holds references to the actual person objects, so that people can transition through disease states normally and have their internal changes updated in the queue as well. This design choice prohibits the possibility that a person could transition into the symptomatic state but still receive a vaccination, since in an actual outbreak the person would be immediately isolated if symptoms appeared.

3.4 The individual-based model with homogeneous mixing

The person objects in the individual-based model with homogeneous mixing are identical to the person objects in the individual-based model with heterogeneous mixing.

Intrapersonal disease progression is modeled identically in the two models as well. The vaccination queue is maintained in the same manner also. The only difference between the two models lies in the mixing strategy employed.

In the individual-based model with homogeneous mixing, the complex mixing scheme based upon the unique identification number of each person is ignored. Instead, each person in the simulation is assumed to have an equal chance of coming in contact with any other person during each timestep. This "free mixing" assumption lessens the computational load of the model but fails to reproduce the localized behavior that most real people exhibit. Thus, the disease will not be localized in small subpopulations as it can be in the agent-based model and the individual-based model with heterogeneous mixing. Free mixing is assumed to be a "worst-case" assumption [28] since the at-risk population is as large as possible. Therefore the traced vaccination policy should perform more weakly in the individual-based model with homogeneous mixing, but the mass vaccination policy should perform in a similar manner.

3.5 The general transitions model

The general transitions model attempts to imitate the major strengths of the individual-based models while significantly reducing the computational cost of performing a simulation. To reduce computational cost while trying to maintain the benefits of modeling arbitrary transition probabilities, the generalized transition model splits each disease state into the probable fractions of the population in each state that have been in the state for a certain amount of time. As mentioned earlier, people tend to remain in each state for a predictable amount of time. The generalized transition model provides the ability to represent arbitrary transition probabilities in an effort to mimic the population dynamics expressed in stochastic models. The model is similar to the integro-differential equation anthrax model of Wein et. al. [20].

The general transitions model is a population-based model. Transitions between states are enacted upon fractions of the population in each state rather than upon

individuals within the community. However, the time during which each fraction of the population remains within its current state is recorded. This information allows each fraction of a population to transition at a rate dependent upon the amount of time the fraction has been in the state, thus capturing the essence of the natural transition method of the individual-based model with homogeneous mixing. The fractions of each population represent the number of individuals within the state that have remained in the state for that amount of time. When employed in a model that does not contain intervention techniques, the general transitions model can produce results that are identical to the averaged results of a large number of individual-based simulations.

Unfortunately, the general transitions model cannot explicitly capture the complex population dynamics inherent in traced vaccination. Since fractions of populations are being transitioned, there are no individuals maintaining contact lists that are queued for vaccination when the individuals become symptomatic. Instead, estimations based on the disease parameters and mixing assumptions must be used to determine the size of the population that should enter the vaccination queue during each timestep.

3.5.1 Approximations necessary

There are four estimations necessary when intervention is introduced in population-based models:

- The number of people who should be infected each timestep
- The number of people in each queue state who should be vaccinated
- The number of people each newly symptomatic case infected during the time of his or her infectiousness
- The expected number of untraced contacts previously infected by each newly symptomatic case who are in each disease state when the symptomatic case is detected

The number of people infected during each timestep depends upon the disease transmission rate, the number of infectious people in the population, and the number of susceptible people in the population. The mass action law [31] governs the population-based models. Consequently, the number of susceptible people infected in a timestep equals $\beta ISdt$, where dt is the timestep of the model and β is the infection rate of smallpox.

The second parameter was described in depth in Section 2.3.3. An analysis of the validity of the assumption is given in Section 5.1.3.

The average number of people a newly symptomatic case would have infected by time t during his or her infectiousness in a population comprised entirely of susceptible people, $R_0(t)$, depends upon the length of time the individual was infectious, the infection rate of the disease, and the number of susceptible people in the population while the individual was infectious. When arbitrary transition probabilities are used, integrals must be evaluated to determine the probability $P(t)$ that an individual was infectious for the length of time t. An equation for $R_0(t)$ is given in Equation 3.2. τ represents the amount of time before intervention commences and $p(t)$ is the probability that a person transitions out of the state in his or her t^{th} step.

$$R_0(t) = \int_0^{t+\tau} P(t-x)\,\beta\,[S^0(t-x) + Q_0(t-x) + S^1(t-x)]\,dx \tag{3.2}$$

$$P(x) = \prod_{i=0}^{x/dt} (1 - p(i)) \tag{3.3}$$

Although the $R_0(t)$ found by Equation 3.2 is not exact, it should produce the same value as an averaged calculation of $R_0(t)$ taken from many runs of the individual-based model.

$\lambda_j(t)$, described as "the expected number of untraced contacts previously infected by an index detected at time t who are in disease stage j when the index is detected" [29], is a key parameter when determining the effectiveness of traced vaccination. To quell the outbreak, infected people must be vaccinated while the vaccine is

still effective. In other words, an increase in $\lambda_1(t)$ corresponds to a decrease in outbreak impact, since infected people are only sensitive to the vaccine when in disease stage 1. $\kappa(t)$ represents the "rate with which anyone in the population is randomly traced at time $u = 0$" [29]. Equations for each of these parameters are given in Equations 3.4–3.6, with $R_0(t)$ and $P(t)$ as described above. Each newly symptomatic case lists c contacts, but only a fraction p of the $R_0(t)$ real contacts are identified. Thus, a total of $[c - pR_0(t)]$ contacts are incorrectly identified, and are mapped proportionately over the entire population.

$$\lambda_j(t) = \int_0^{t+\tau} P(t-x)\,\beta S^0(t-x)\,e^{-\int_{t-x}^{t}\kappa(u)\,du} \times \gamma_j(t)\,dx \tag{3.4}$$

$$\gamma_j(t) = \Pr\{\text{Contact in stage } j \text{ at } t \mid \text{infected at } (t-x)\} \tag{3.5}$$

$$\kappa(u) = \frac{[c - pR_0(u)]\,r_3(u)\,I_3(u)}{N} \tag{3.6}$$

Exact calculations of the probabilities that a contact is in stage j at time t given that the contact was infected at time $t - x$ can be derived from sums of products, but the computational cost of determining them reduces the benefits of using a population-based model.

3.5.2 Computer implementation of the general transitions model

The general transitions model is optimized to perform the fewest computations necessary during each simulation. Like the individual-based models, the general transitions model allows the use of arbitrary transition probabilities to determine transition times between states. Since fractions of each population are transferred each timestep, the transitions that have equal probabilities of transferring each day are represented by single values. These populations are untraced susceptible, traced susceptible, recovered/immune, and dead. Each person in the susceptible populations has an equal chance of becoming infected during each timestep, regardless of the time spent in the

state. The recovered/immune and dead populations never transfer states again.

The populations whose transition probabilities are dependent upon the time spent in the state are represented by vectors of double values. In each vector v, the entry $v.at(i)$ corresponds to the subpopulation that has been in the state for i timesteps. Transitions out of the state happen in two different ways: natural transitions and transitions due to vaccination.

Natural transitions cause fractions of each subpopulation to be transferred. Since each state has transition probabilities $p(t)$, where $p(t)$ is the probability that a person will transition out of the state in his or her t^{th} step, the total size of the population that naturally transitions each timestep (ΔP) is given by the equation in Equation 3.7.

$$\Delta P = \sum_{i=0}^{v.size()} v.at(i) \cdot p(i) \tag{3.7}$$

The queue transitions are treated differently, since the queue is a first-come, first-served structure. As mentioned in Section 2.3.3, the populations transition in proportions relative to the number of people in the queue. However, under that assumption, the populations that have waited the longest for treatment are treated first. A pseudocode representation of this transitioning strategy is given in Figure 3-1.

3.6 The ordinary differential equation population-based model

The ordinary differential equation population-based model uses seventeen ordinary differential equations to govern the transitions between various stages of the disease. The equations are shown in Equations 3.8–3.22, with the parameters of the model given in Table 3.2. As in the paper by Kaplan et. al., in each differential equation "the subscript j denotes the stage of infection, whereas the superscript l denotes whether a person has yet to be traced ($l = 0$) or has already been traced and vaccinated ($l =$

```
void transitionOutOfState(vector<double> & v, double numToTransition) {
   double populationToTransfer = numToTransition;
   int subState = v.size() - 1;  // People in queue longest leave first
   while(populationToTransfer > 0) {
      if(v.at(subState) > populationToTransfer) {
         v.at(subState) -= populationToTransfer;
         populationToTransfer = 0;
      }
      else {
         populationToTransfer -= v.at(subState);
         v.pop_back();
         subState--;
      }
   }
}
```

Figure 3-1: The queue transition strategy of the general transitions model

1). The state variables are: S^l = number of type l susceptibles; I_j^l = number of type l infected persons in disease stage j ($j = 1, 2, 3, 4$); Q_j = number in tracing/vaccination queue at disease stage j ($j = 0$ means susceptibles); H = number in febrile quarantine; Z = number immune (from vaccination) or recovered from smallpox; D = number dead" [29].

The states are grouped by the tracing variable (untraced, queued, or traced) and intervention does not commence until after a pre-specified time, τ. A major difference between the ODE model and the other models studied stems from the property that the population transitions follow the mean field of exponential distributions. Since arbitrary transition probabilities cannot be used in the ODE model, specific disease rates must be given. The disease rates employed are given in Table 3.3.

Table 3.3: Disease rates for the ODE model

r_1	Disease stage 1 rate	$\mu = (5\ days)^{-1}$	[6]
r_2	Disease stage 2 rate	$\mu = (7\ days)^{-1}$	[6]
r_3	Disease stage 3 rate	$\mu = (3\ days)^{-1}$	[6]
r_4	Disease stage 4 rate	$\mu = (16\ days)^{-1}$	[6]

The meaning of each differential equation is readily discernible, and additional information can be found in the Appendix of the Kaplan et. al. paper [29]. Notational

$$\frac{dS^0}{dt} = -\beta I_3 S^0 - [c - pR_0(t)]\,\frac{S^0}{N}\,r_3 I_3 \tag{3.8}$$

$$\frac{dI_1^0}{dt} = \beta I_3 S^0 - \{[c - pR_0(t)]\,\frac{I_1^0}{N} + p\lambda_1(t)\}\,r_3 I_3 - r_1 I_1^0 \tag{3.9}$$

$$\frac{dI_j^0}{dt} = r_{j-1} I_{j-1} - \{[c - pR_0(t)]\,\frac{I_j^0}{N} + p\lambda_j(t)\}\,r_3 I_3 - r_j I_j^0 \tag{3.10}$$

$$\frac{dI_4^0}{dt} = r_3 I_3^0 - r_4 I_4^0 \tag{3.11}$$

$$\frac{dQ_0}{dt} = [c - pR_0(t)]\,\frac{S^0}{N} r_3\, I_3 - \beta I_3 Q_0 - \mu\, Q_0 \min(1, \frac{n}{Q}) \tag{3.12}$$

$$\frac{dQ_1}{dt} = \beta I_3 Q_0 + \{[c - pR_0(t)]\,\frac{I_1^0}{N} + p\lambda_1(t)\} r_3\, I_3 - \mu\, Q_1 \min(1, \frac{n}{Q}) - r_1 Q_1 \tag{3.13}$$

$$\frac{dQ_j}{dt} = r_{j-1} Q_{j-1} + \{[c - pR_0(t)]\frac{I_j^0}{N} + p\lambda_j(t)\} r_3 I_3 - \mu Q_j \min(1, \frac{n}{Q}) - r_j Q_j \tag{3.14}$$

$$\frac{dH}{dt} = (1-f) h \mu Q_3 \min(1, \frac{n}{Q}) - r_3 H \tag{3.15}$$

$$\frac{dS^1}{dt} = (1-f)(1-v_0)\mu Q_0 \min(1, \frac{n}{Q}) - \beta S^1 I_3 \tag{3.16}$$

$$\frac{dI_1^1}{dt} = \beta S^1 I_3 + (1-f)(1-v_1)\mu Q_1 \min(1, \frac{n}{Q}) - r_1 I_1^1 \tag{3.17}$$

$$\frac{dI_2^1}{dt} = r_1 I_1^1 + (1-f)\mu Q_2 \min(1, \frac{n}{Q}) - r_2 I_2^1 \tag{3.18}$$

$$\frac{dI_3^1}{dt} = r_2 I_2^1 + (1-f)(1-h)\mu Q_3 \min(1, \frac{n}{Q}) - r_3 I_3^1 \tag{3.19}$$

$$\frac{dI_4^1}{dt} = r_3 (I_3^1 + Q_3 + H) - r_4 I_4^1 \tag{3.20}$$

$$\frac{dZ}{dt} = (1-f)(v_0 Q_0 + v_1 Q_1)\,\mu \min(1, \frac{n}{Q}) + (1-\delta)\, r_4\,(I_4^0 + I_4^1) \tag{3.21}$$

$$\frac{dD}{dt} = f \mu\, Q \min(1, \frac{n}{Q}) + \delta\, r_4\,(I_4^0 + I_4^1) \tag{3.22}$$

Figure 3-2: The mathematical representation of the ODE model

Table 3.4: Notational abbreviations

Notation	Definition
j	Stage of infection
k	Boolean indicating tracing ($k = 0$ if person is not traced)
S^k	Number of type k susceptibles
I_j^k	Number of type k infected persons in disease stage j
Q_j	Number in vaccination queue in disease stage j
H	Number in febrile quarantine
Z	Number immune or recovered
D	Number dead from smallpox
I_3	Total number of freely mixing infectious individuals
Q	Total number of people in vaccination queue
τ	Time interval between attack and initiation of intervention
$R_0(t)$	The average number of people infected by a newly symptomatic case
$\lambda_j(t)$	The expected number of untraced contacts previously infected by an index case detected at time t who are in disease stage j when the index is detected

abbreviations can be found in Table 3.4.

3.6.1 Computer implementation of the ordinary differential equation population-based model

Every state transition is governed by a differential equation in the ODE model. Consequently, Euler's method [2] can be used as a first-order approximation of the transitions. The essence of Euler's method lies in the assumption that given a step dt that is "small enough", for any t where the functions $y(t)$ and $y'(t) = f(y,t)$ are defined, $y(t+dt) \approx y(t) + y'(t) \cdot dt$. Although the Runge-Kutta method [4] produces a higher-precision prediction by using a weighted average to determine a more accurate slope between t and $t + dt$ than the slope $y'(t)$ used in Euler's method, the relatively small changes in slope of each curve do not warrant the increase in complexity of the algorithm. Furthermore, the Runge-Kutta method is not as easily extensible to the generalized transition model described in Section 3.5 as is Euler's method.

Chapter 4

Results

The five models were tested and validated under a variety of different initial conditions and parameters to determine the sources of discrepancies between the models. Both traced and mass vaccination schemes were studied.

4.1 Test results

The base case scenario consisted of ten initially infected individuals in the population and no intervention methods. Three graphs are shown below: the susceptible, recovered, and dead populations.

As the graphs show, the individual-based model with heterogeneous mixing and the agent-based model predict similar outbreaks, and are the most optimistic of the five models. The individual-based model with homogeneous mixing and the general transitions model produce virtually identical outputs, as expected. The ODE model produces the most pessimistic prediction. Figure 4-1 shows that the ODE model predicts the largest number of people to be infected (absent intervention, the only transition out of the susceptible state is into disease stage 1). Those infected individuals eventually either recover or die, which is why both the recovered and dead populations are largest in the ODE model.

Intervention methods are introduced after five and 25 days. Intervention after five days simulates national intelligence finding out about the outbreak before any-

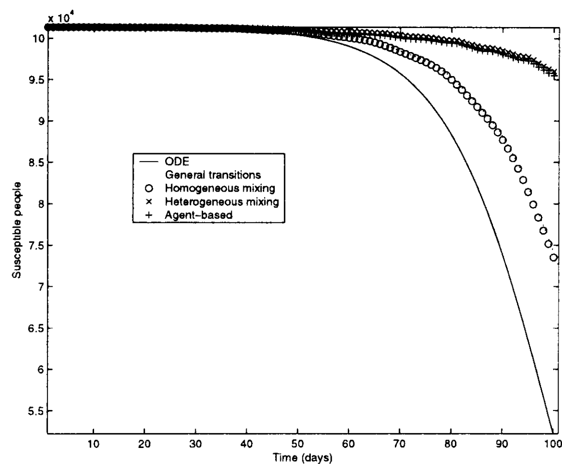


Figure 4-1: The susceptible populations with no intervention

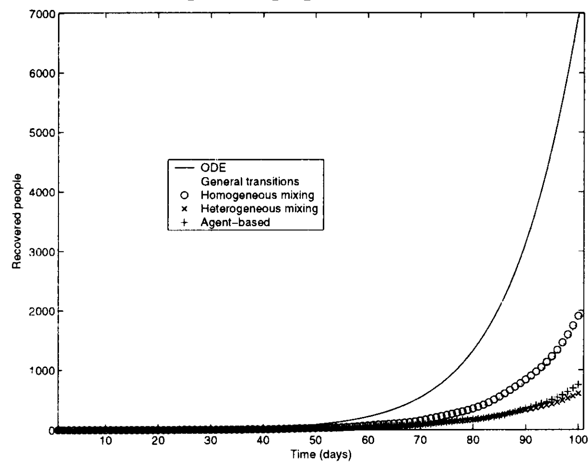


Figure 4-2: The recovered population with no intervention

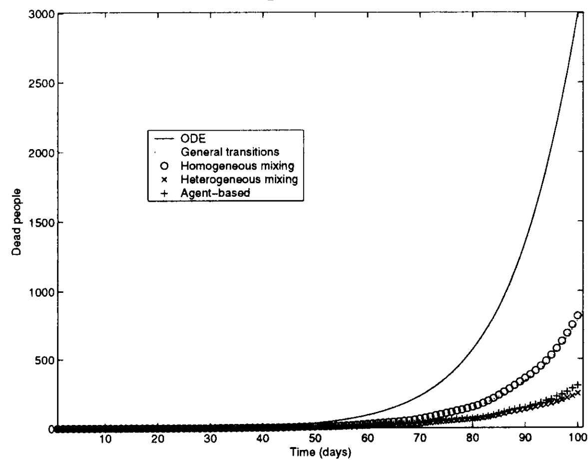


Figure 4-3: The dead population with no intervention

one becomes symptomatic. Intervention after 25 days simulates recognition of the disease after two generations have become symptomatic (it is assumed that the first generation would not enter hospitals since they are attackers and want to delay public recognition of the disease as long as possible). Although in an actual attack the initially infected individuals would try to infect as many people as possible, in the models they are treated in the same manner as every other individual. The graphs of the susceptible, recovered, and dead populations under both scenarios are shown in Figures 4-4–4-9. Traced vaccination is implemented in each of the models presented.

The agent-based model and the individual-based model with heterogeneous mixing predict the smallest outbreaks and are similar in all cases. The individual-based model with homogeneous mixing and the general transitions model produce results that are similar at first and diverge as the outbreaks continue, and the ODE model predicts the largest outbreaks in all cases.

Mass vaccination is implemented after five and 25 days for the reasons described above. Since mass vaccination eliminates the complex population dynamics involved when contacts must be traced and queued individually, the graphs should be more similar than those under traced vaccination. Similarly, the earlier mass vaccination starts, the more similar the predictions should be. The graphs of the susceptible, recovered, and dead populations under both intervention time scenarios are shown in Figures 4-10–4-15. The predictions given by the models are distinct, but the scale of the predictions must be noted. The difference between models in number of deaths predicted in mass vaccination is much smaller than in traced vaccination. Furthermore, the individual-based model with heterogeneous mixing and the agent-based model give predictions that are similar to the individual-based model with homogeneous mixing. This result stems from the assumption that visiting the hospital creates a more homogeneous mixing scenario. The individual-based model with homogeneous mixing and the general transitions model give different results for reasons not fully understood.

The simulations were terminated after 100 days due to the prohibitively large computational cost of running the agent-based model. The current disease and pop-

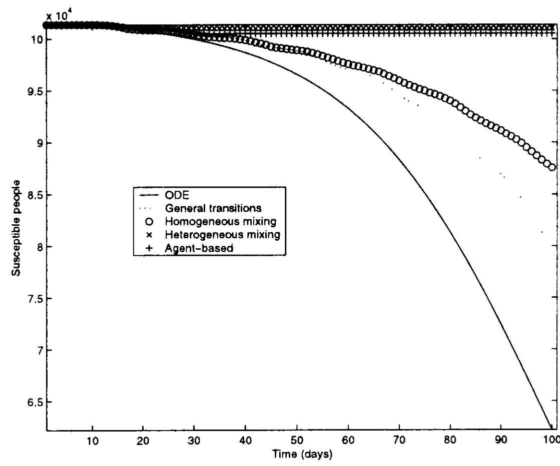


Figure 4-4: The susceptible populations with traced vaccination after five days

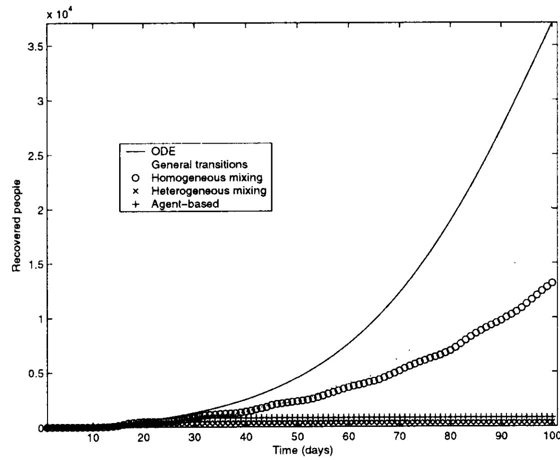


Figure 4-5: The recovered population with traced vaccination after five days

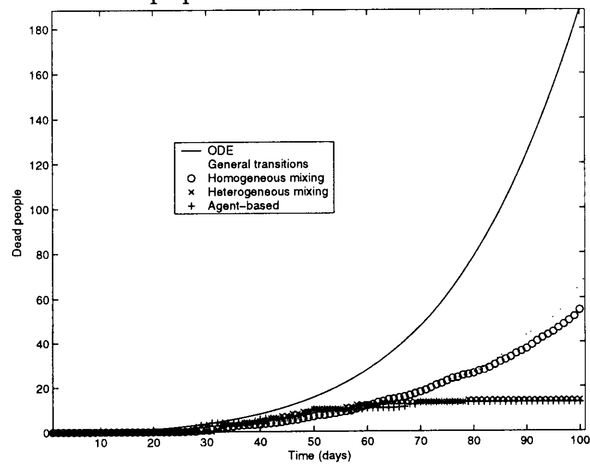


Figure 4-6: The dead population with traced vaccination after five days

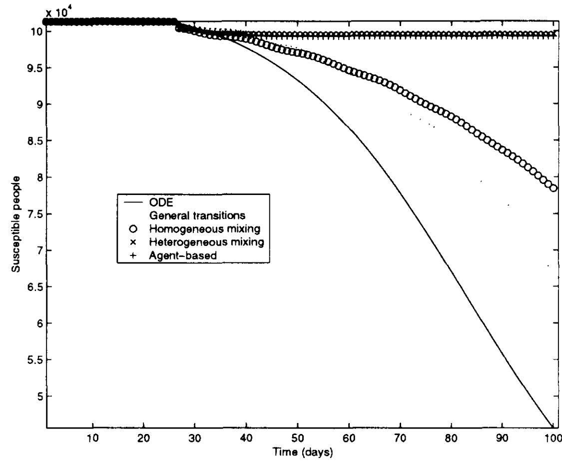


Figure 4-7: The susceptible populations with traced vaccination after 25 days

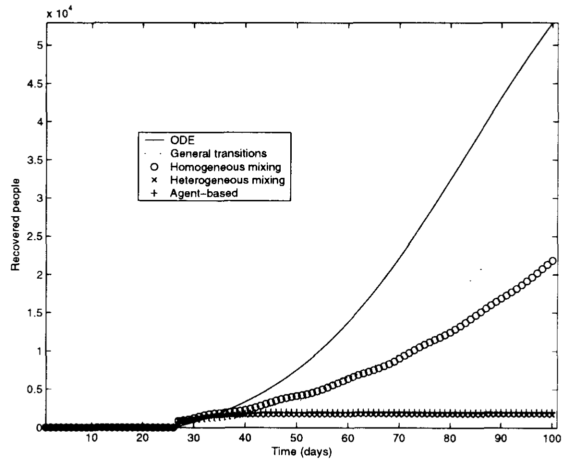


Figure 4-8: The recovered population with traced vaccination after 25 days

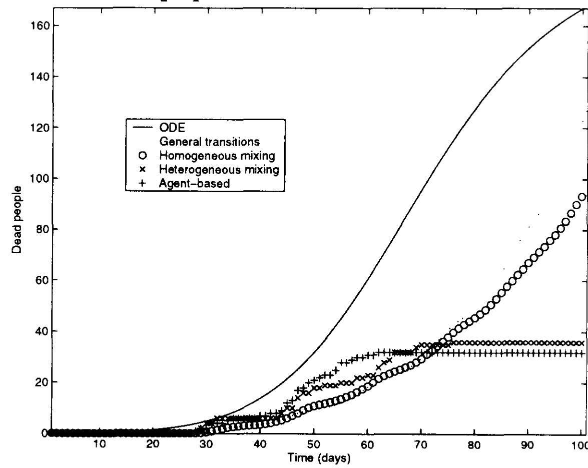


Figure 4-9: The dead population with traced vaccination after 25 days

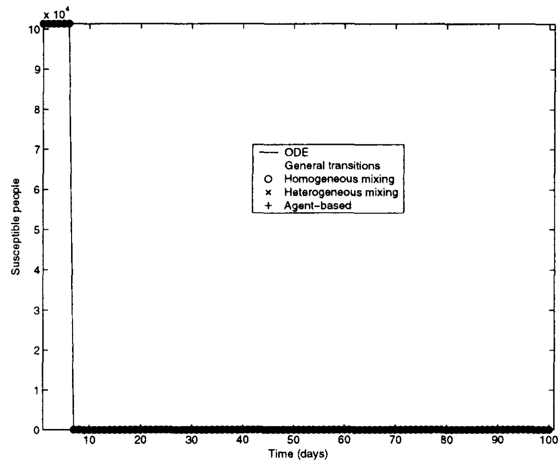


Figure 4-10: The susceptible populations with mass vaccination after five days

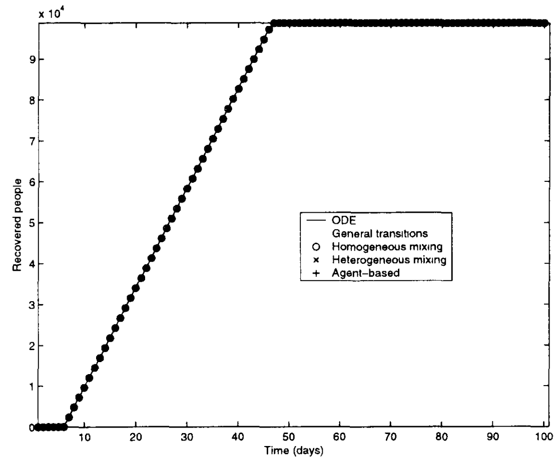


Figure 4-11: The recovered population with mass vaccination after five days

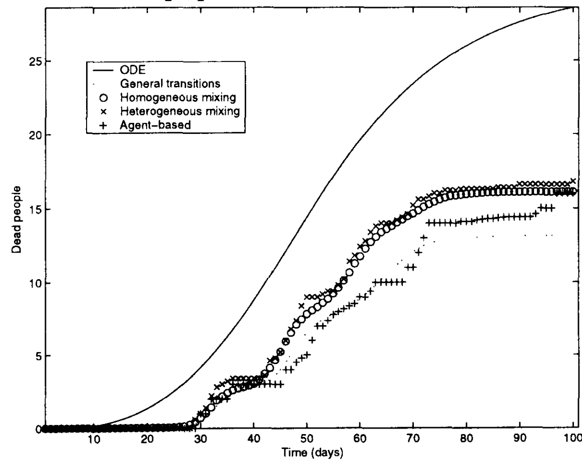


Figure 4-12: The dead population with mass vaccination after five days

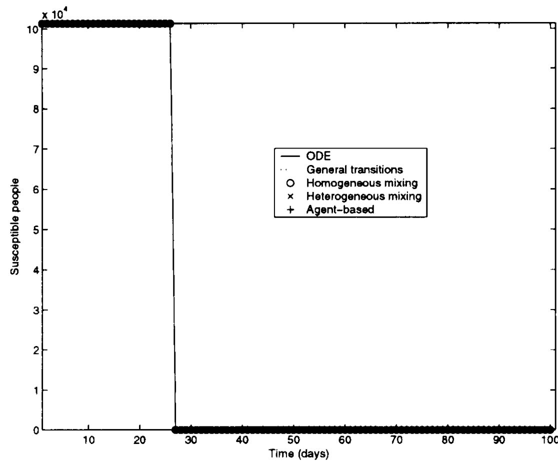


Figure 4-13: The susceptible populations with mass vaccination after 25 days

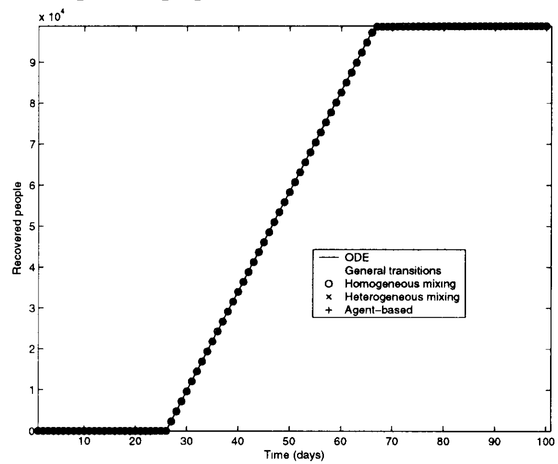


Figure 4-14: The recovered population with mass vaccination after 25 days

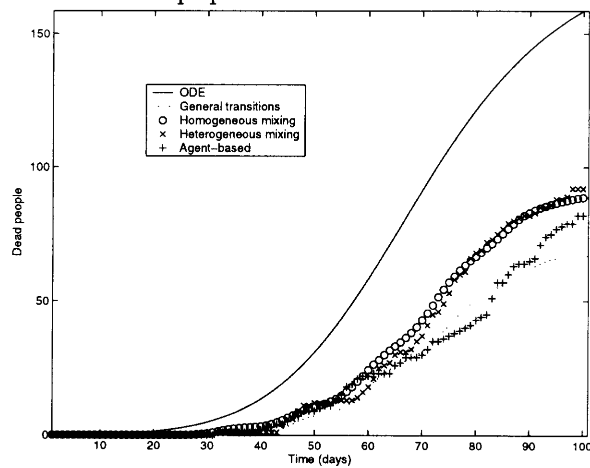


Figure 4-15: The dead population with mass vaccination after 25 days

ulation parameters indicate that an outbreak will not run its course after 100 days, however. This is evidenced in the models by a failure to reach constant susceptible, recovered, and dead populations—in other words, there are still infectious individuals in the population after 100 days. To provide a benchmark "worst-case" estimate of outbreak impact, the individual-based model with homogeneous mixing was run for 300 days. The model was run with no intervention, with traced vaccination after 25 days, and with mass vaccination after 25 days. The results of the trials are given in Figures 4-16–4-18. As the graphs show, the epidemic appears to stop after approximately 125 days in mass vaccination, after 150 days with no vaccination, and after approximately 200 days in traced vaccination. Mass vaccination predicts the smallest outbreak and also stops the outbreak in the shortest amount of time. The outbreak with no intervention terminates before the outbreak with traced vaccination because the entire population becomes infected very rapidly.

4.2 Validation techniques and results

The models were validated through testing, but some significant issues require additional study. The validation attempted to address the four main possible sources of error mentioned by Gottfried: the data, the models, the model implementations, and interpretation of the results [24]. The data used and the types of models created were justified in earlier sections of this thesis and are based largely upon the work of Edward Kaplan et. al. The ODE model was run using the equations and parameters from [29] to ensure proper implementation of the model. Higher-fidelity models can usually be simplified to mimic lower-fidelity models, so those tests were completed first in an effort to validate the implementations of the higher-fidelity models. The use of exponential transition probabilities in the general transitions model caused its predictions to be identical to that of the ODE model. The average prediction from the individual-based model with homogeneous mixing, with no intervention, is identical to the prediction of the general transitions model with no intervention. The predictions from the individual-based model with heterogeneous mixing are identical to the

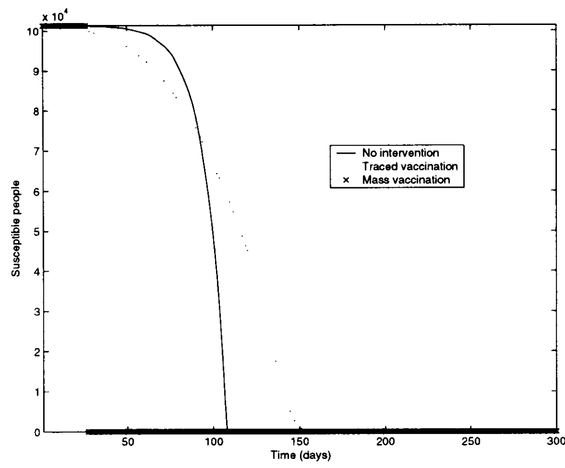


Figure 4-16: The susceptible populations in a 300 day simulation

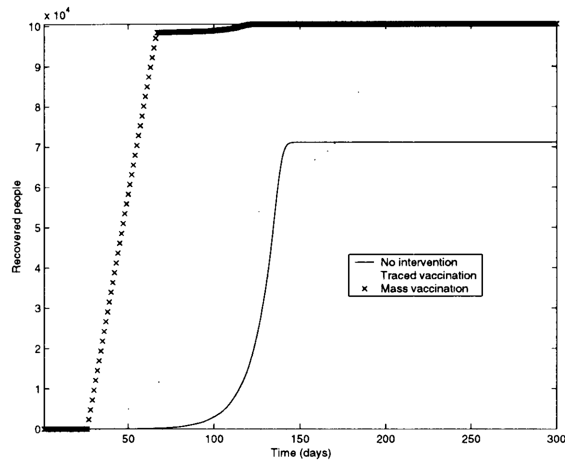


Figure 4-17: The recovered population in a 300 day simulation

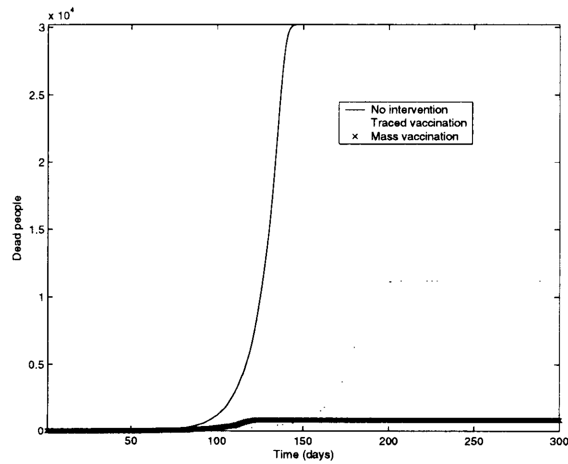


Figure 4-18: The dead population in a 300 day simulation

predictions from the individual-based model with homogeneous mixing when the heterogeneous mixing constraints are removed. Similarly, the agent-based model gives identical results to the homogeneous mixing model when all mixing and movement constraints are removed.

A number of results are not explained by the model differences. In particular, it is not clear why the general transitions model and the individual-based model with homogeneous mixing produce different results when intervention techniques are implemented. Also, the Gaussian transition probability assumption creates distinct generational patterns in the populations for many generations. This pattern may be an exaggeration of true outbreak characteristics. Further validation and study of the existing models and parameters is necessary to fully understand the results.

Chapter 5

Analysis and Discussion

The data gathered gives much insight into the differences between the models. Not only do the models differ under each intervention scheme, but also the models produce noticeably different results when disease parameters are altered. This discussion examines the causes of change in an effort to understand both the strengths of each modeling technique and the implications of incorrect disease parameter assumptions.

5.1 Model differences

The five models vary in a number of ways. The critical differences lie in two main areas: transition probability generation and individual-based vs. population-based modeling. The latter difference is separable into the mixing, queue removal, and contact tracing strategies. Each of these differences and its effects on model predictions is analyzed below.

5.1.1 Transition probability generation

The curves produced by the ODE model are the source of the most glaring difference between the five models. As mentioned in Section 2.3.1, the other four models are able to employ arbitrary transition probabilities. Due to its relatively long incubation time, smallpox outbreaks frequently exhibit distinct generational patterns. For example,

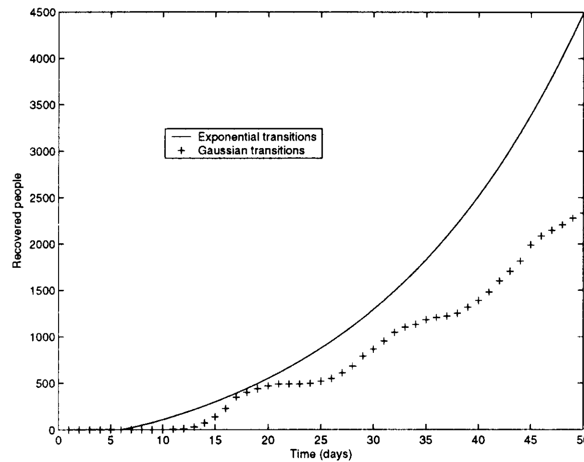


Figure 5-1: The disease generations of recovered individuals

the data from the 1972 Kosovo outbreak shows clear generations in the removed population [23].

In contrast, the ODE model is governed by the exponential solutions to its differential equations. The curves produced thus follow the mean field of exponential distributions. The smooth exponential curves produced by the ODE model thus lose any generational patterns exhibited by actual smallpox outbreaks. The difference between the ODE model and the other models is shown in Figure 5-1 for recovered individuals with the standard outbreak parameters and traced vaccination after 25 days.

As the graph shows, the model that implements Gaussian transitions exhibits distinct generations of recovered individuals in the first 50 days of an outbreak. As the number of people in the recovered state increases, the early generations become less noticeable. However, the Gaussian transitions may exaggerate the generational patterns of the outbreak. Further investigation of smallpox parameters and historical data would be helpful in determining the desirability of this behavior. The model predicts an incubation time of twelve days (seven while vaccine-sensitive and five when vaccine-insensitive) and an infectious period of three days, leading to generations every twelve to fifteen days. In addition, it is useful to recall that the exponential transition probabilities correspond to a constant probability of transitioning out of

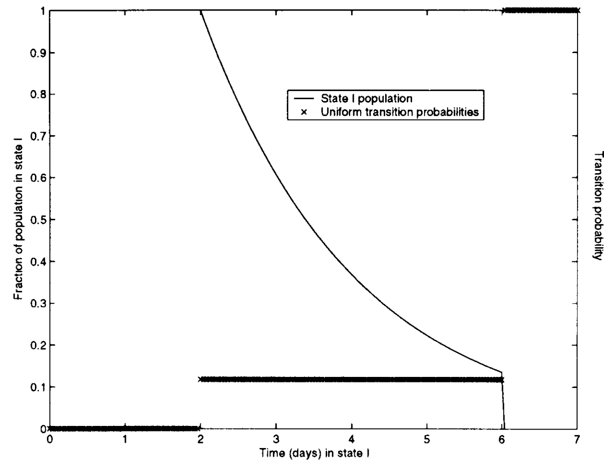


Figure 5-2: Uniform transitions

the current state, regardless of the time spent in the state. This assumption undermines the effectiveness of early intervention, since infected people may become vaccine-insensitive in an unrealistically short amount of time. In fact, the exponential transitions of the ODE model produce the most pessimistic predictions in all intervention cases.

As mentioned in Section 2.3.1, the choice to use probabilities that would yield Gaussian distributions on transitioning times was not validated by any study of smallpox state transitions. To analyze the validity of the assumption, other probability generators were used. A uniform probability generator and the expected population remaining in the state is shown in Figure 5-2 for the same state as in Section 2.3.1 (average transition time of 4 days, transitions after 2–6 days) and a timestep of 0.25. The results of using these probability generators for a homogeneous mixing model with no intervention are shown in Figures 5-3–5-5.

As the graphs show, the predictions given by both models are similar. The Gaussian transition model predicts slightly fewer infections but the difference in death predictions is small.

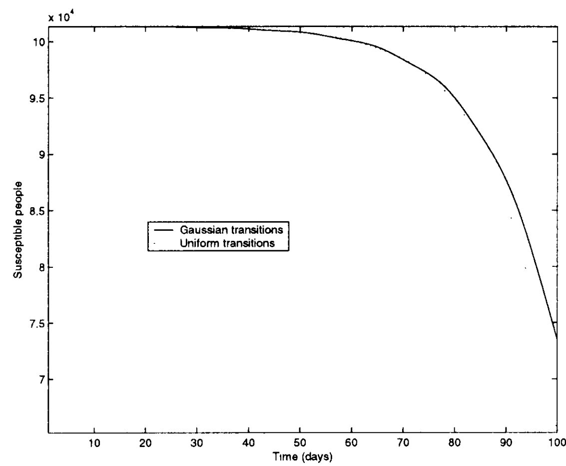


Figure 5-3: The susceptible populations with uniform and Gaussian transitions

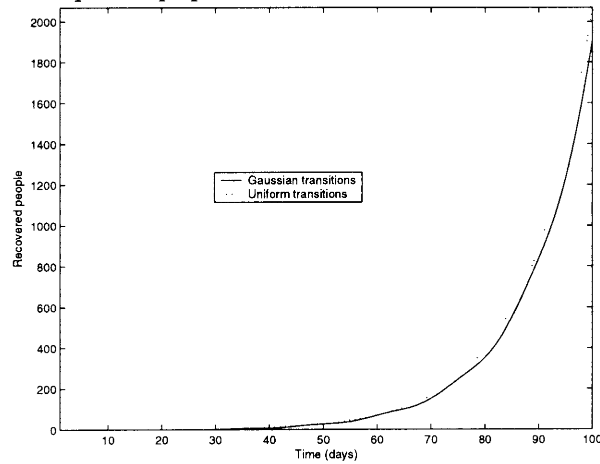


Figure 5-4: The recovered population with uniform and Gaussian transitions

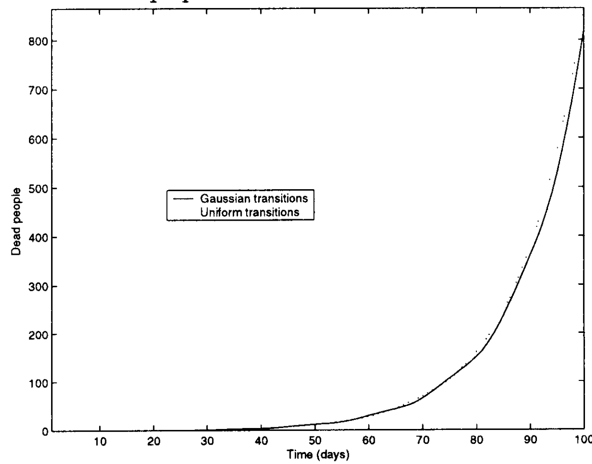


Figure 5-5: The dead population with uniform and Gaussian transitions

5.1.2 Mixing strategy

The mixing strategy employed can play a large part in the prediction of a model. Kaplan et. al. based their suggestion that mass vaccination should become the standard CDC policy (as opposed to the current policy of traced vaccination that switches to mass vaccination after two disease generations if the outbreak has not been contained [7]) on the predictions of his ODE model that employed homogeneous mixing [29]. Halloran, on the other hand, used a stochastic agent-based simulator to show that traced vaccination had a higher ratio of cases prevented to doses used than mass vaccination [25]. Furthermore, when some limited herd immunity was assumed, Halloran's model showed that traced vaccination became competitive with mass vaccination as a means of controlling a smallpox outbreak.

The results of Figures 4-1–4-15 show that the mixing strategy plays a large role in outbreak impact predictions. As expected, the models that use Gaussian transitions between states predict a smaller attack impact than the model that uses exponential transitions. Unlike the three lowest-fidelity models, however, the outbreak impact predicted by the agent-based model and the individual-based model with heterogeneous mixing is larger in mass vaccination than in traced vaccination. In mass vaccination, the predictions given by the two highest-fidelity models are very similar to the prediction given by the individual-based model with homogeneous mixing. The similarity is explained by the treatment of individuals in the hospital. As described in Sections 3.2 and 3.3, when individuals are in the vaccination queue, homogeneous mixing is assumed. Consequently, the disease spreads more freely than when it was restricted to subpopulations by the heterogeneous mixing assumption. The results are different than the results of Kaplan et. al. [29] which favor mass vaccination. The assumption of homogeneous mixing in the vaccination queue may be overly pessimistic, however. When the agent-based model implemented mass vaccination in a manner that eliminated contact between individuals, the prediction of outbreak impact was smaller than in traced vaccination. Refining the model assumptions to more realistically reflect the disease transmission characteristics in hospital queues would

produce more accurate results.

5.1.3 Vaccination queue removal

The treatment of the vaccination queues in each model is described in Section 2.3.3. One would expect the behavior of the two vaccination strategies to be very similar in the average case. This assumption was tested by monitoring each person in the individual-based models as he or she was treated. The percentages of vaccinated people in each disease state was then compared to the percentages of people in each disease state in the entire queue. A weighted average taken over many timesteps showed that the actual percentages of each type of person that transitioned was nearly identical to the assumption of the population-based models. Thus, the vaccination queue does not seem to be a cause of the different predictions between models.

5.1.4 Contact tracing

As mentioned in Section 3.5, a newly symptomatic person's contacts are not explicitly known in the population-based models. Approximations to the disease stages of these contacts (given in Equations 3.2–3.6) must be used in the population-based models. In the paper by Kaplan et. al. [29], the equations are further simplified to reduce the computational cost incurred by each simulation.

The further approximations did not yield particularly accurate results when run against an individual-based model that used exponential transition probabilities and homogeneous mixing. However, when the integrals of Equations 3.2–3.6 were formally evaluated, the results were much more accurate. The graphs of the susceptible, recovered, and dead populations for the two models are given in Figures 5-6–5-8.

5.2 Altering disease parameters

Accurate model parameters must be used in order to be confident in the prediction given by any model. Smallpox models are no exception—in fact, an underprediction of

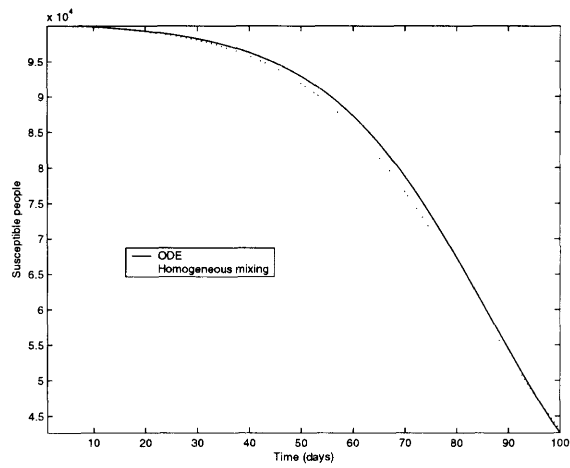


Figure 5-6: The susceptible populations of the ODE model and the corresponding individual-based model

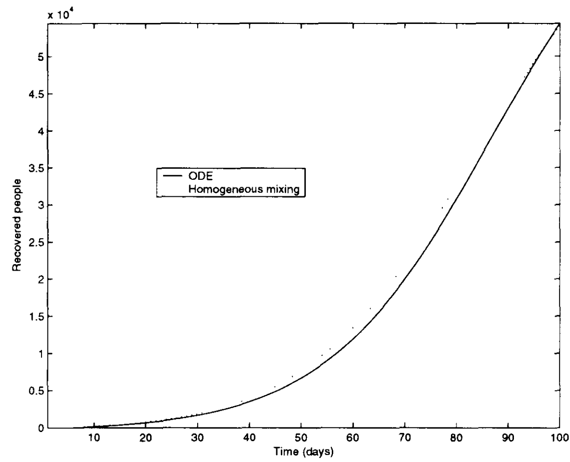


Figure 5-7: The recovered populations of the ODE model and the corresponding individual-based model

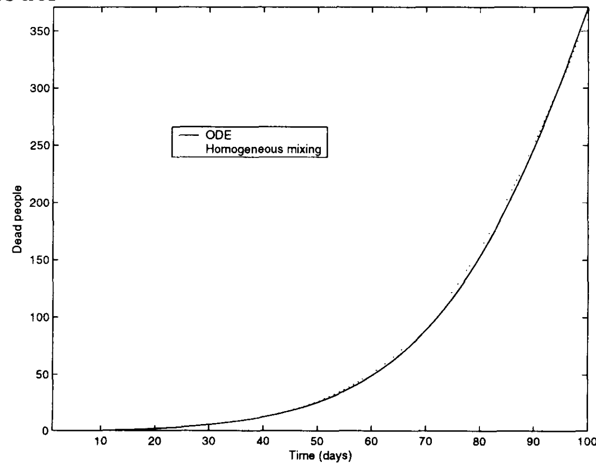


Figure 5-8: The dead populations of the ODE model and the corresponding individual-based model

Table 5.1: Changing parameters and their effects on outbreak impact

Parameter changed	**Multiplication factor**
Initial infections doubled	2.03
Infectiousness doubled	43.51
Contact percentage increased to 85%	0.18
Intervention commenced 20 days earlier	0.67

the magnitude of an outbreak could be disastrous. The effects of changing a number of parameters is examined below. Each parameter is analyzed with respect to the predictions given by the individual-based model with homogeneous mixing. The base case for comparison has ten initial infections, $R_0 = 3.0$, traced vaccination beginning on day 25, and a 50% accuracy in naming contacts. The individual-based model with homogeneous mixing was consistently the most pessimistic model (ie. predicted the most deaths) of the individual-based models. The changed parameters were:

- The number of initial infections
- The basic reproductive rate
- The day intervention techniques commence
- The percentage of contacts correctly identified

A graph of the different death predictions is given in Figure 5-9 with the reproductive rate case omitted to preserve graph scale. A summary of the parameter changes and their effects on the dead populations is given in Table 5.1. The multiplication factors in the table correspond to the size of the estimated outbreak when compared to the base case. For instance, the base case predicts 94 deaths and the model with twice the number of initial infections predicts 191 deaths. The multiplication factor for that case is thus 2.03, since $2.03 \times 94 \approx 191$.

5.2.1 The number of initial infections

Intuitively, doubling the number of initial infections should double the number of predicted deaths given by the model. The model predicts a multiplication factor of

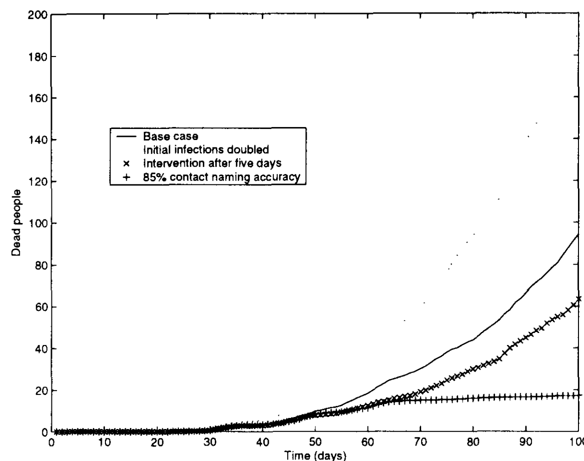


Figure 5-9: The effects of altering model parameters

2.03, which is very close to the expected 2.0. A smaller multiplication factor can be consistent with the model in certain scenarios, however. An explanation for a smaller multiplication factor is necessary if more than half of the initially susceptible population becomes infected in the base case. In that scenario, when the initial number of infected people is doubled, the susceptible population will not be large enough to account for all the infections that are predicted to take place.

5.2.2 The basic reproductive rate

The basic reproductive rate of smallpox, R_0, is the number of people an infectious person would infect in an entirely susceptible population. The models presented here assumed that $R_0 = 3.0$. However, an examination of the effects of an increased R_0 is useful. Figure 5-10 shows the death predictions of the standard model and one in which people are twice as infectious for an 80 day simulation. As the figure shows, the increase in deaths with the new assumption is dramatic. After 100 days the model with $R_0 = 6.0$ predicts a 44-fold increase in deaths. This number, while alarming, is consistent with expectations. Since every infected individual infects twice as many people as in the base case, the death toll should raise exponentially in each disease generation. Generations are separated by roughly 14 days, and the first generation of infected individuals either recover or die at approximately day 30.

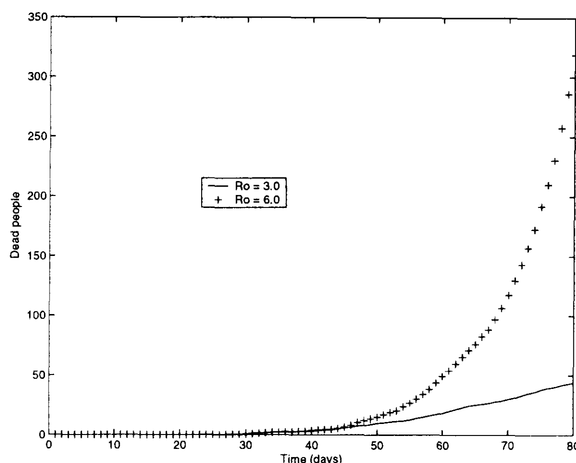


Figure 5-10: The death predictions of the model with $R_0 = 3.0$ and $R_0 = 6.0$

Thus, the increase in deaths could be approximately $2^{\frac{100-30+14}{14}} = 64$-fold. However, when the reproductive rate is doubled, the susceptible population is exhausted—the case mentioned in the previous section. This limitation may account for the smaller multiplication factor.

As the basic reproductive rate rises, the cost of an individual not naming contacts who were infected by him increases dramatically. Consequently, the traced vaccination strategy becomes weaker and weaker when compared to mass vaccination. This effect is examined in Section 5.2.3.

5.2.3 The contact naming accuracy

Correct identification of contacts is critical when using the traced vaccination strategy. When an infected contact is not traced, he or she is free to spread the disease to susceptible people as long as he or she is not randomly contacted by someone else. A larger basic reproductive rate amplifies this effect, since each unidentified infected contact infects more people on average. The reduction in deaths following an improvement in contact naming accuracy to 85% gives a multiplication factor of 0.18.

The basic reproductive rate of the disease and the contact naming accuracy play key roles in the prediction of outbreak impact. A population that names contacts

accurately may benefit from a traced vaccination policy, especially if the disease strain is not very infectious. On the other hand, mass vaccination could more effectively reduce the impact of an outbreak if the disease is highly infectious and the infected population unknowingly transmits the disease to many other people. A more precise knowledge of the reproductive rate of smallpox and the contact naming accuracy of a population must be known to make a truly informed decision regarding the most effective vaccination policy.

5.2.4 The length of time before intervention commences

Intervention techniques are most effective when started as soon as possible. The model indicates that outbreak impact is reduced by a multiplication factor of 0.68 when intervention commences on day five. The intervention day is slightly misleading, however, as nothing happens in traced vaccination until individuals become symptomatic. Rather than at day five, the earlier intervention actually commences around day 15—the day the initially infected individuals become symptomatic.

The graph presented in Figure 5-11 shows the different numbers of untraced infectious people when intervention techniques are delayed five and 25 days. The first 15–20 days are very similar in both scenarios. However, the scenario with intervention starting on day 25 shows a dramatic decrease in untraced infected people on day 25. That jagged drop indicates that earlier intervention would have reduced the effects of the outbreak.

5.3 The computational cost of each model

The computational cost of the individual-based models is much greater than that of the population-based models. All models run in linear time with respect to the number of days to simulate. The population-based models run in constant time with respect to the population size, and a 100 day simulation takes under ten seconds to complete. In contrast, the individual-based models run in time that is linear in the population size. In addition, the agent-based model runs in $O(NM)$ time, where

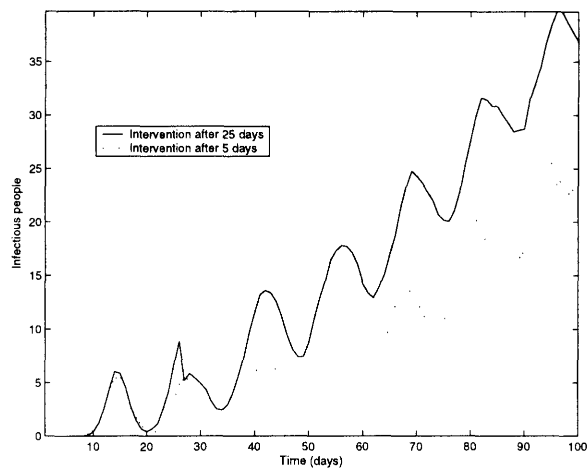


Figure 5-11: The effects on infectious people of waiting 5 or 25 days before traced vaccination

N is the size of the population and M is the number of buildings in the model. Furthermore, each individual-based models must average a number of runs to ensure its prediction is not an outlier. In the end, each individual-based simulation takes between ten and fifteen minutes, and the agent-based model takes multiple hours to simulate mass vaccination.

Chapter 6

Future Work

There are a number of directions in which future work could bring insight to smallpox modeling. First, additional validation and interpretation of the results of the existing models and parameters would be useful to fully understand the sources of differences between the models. In particular, the cause of the differences in predictions of the general transitions model and the individual-based model with homogeneous mixing could be better understood.

In addition, the network structure of the city could be determined more rigorously. The personal relationships in the agent-based model were treated as a randomized regular network. Analysis of the true network structure of people in the city may prove that assumption incorrect. A paper by Chowell et. al. indicates that the city of Portland, Oregon, possesses a scale-free network structure [16]. Scale-free networks cause the mixing strategy and choice of initial infectives to play a much larger role in the prediction of outbreak impact [15], and that relationship could be examined in detail.

The granularity of the agent-based model could also be diminished. Halloran et. al. created an agent-based model of 2000 people with a high degree of precision. A drawback of higher-fidelity models is that "as model realism is increased the transparency associated with simple frameworks is often lost and the validation of model conclusions becomes harder [21, p. 683]." However, a higher-fidelity model would only add to the information on smallpox modeling and could prove quite useful.

In particular, public transportation modeling and more accurate vaccination queue modeling could be introduced into the models.

Finally, the smallpox model predictions can only be as accurate as the parameters used in the models. Although the modeling strategies implemented here and elsewhere give insight into the spread of disease, the lack of knowledge of the disease characteristics limits the ability of the models to make a recommendation for future actions and intervention policies. Further study of the smallpox transmission rate, herd immunity of the vaccinated public, and infectious stage would provide a stronger base on which to predict the impact of a smallpox outbreak.

Chapter 7

Conclusion

The models implemented for this thesis vary widely in fidelity and produce different results. The ODE model gives the most pessimistic predictions of outbreak impact but is the least computationally expensive. The three individual-based models produce similar results in mass vaccination, but differ dramatically in traced vaccination.

Mass vaccination outperforms traced vaccination for the homogeneous mixing models in all scenarios examined in this thesis. This finding is consistent with the Kaplan et. al. paper [29] on which these models are based. The Kaplan et. al. model is biased toward mass vaccination, however, since it assumes that people are only infectious during the prodromal stage of the disease. The heterogeneous mixing models created for this thesis predict smaller outbreaks in traced vaccination than in mass vaccination. The discrepancy stems from the treatment of the vaccination queue. The queue is treated as a homogeneous mixing structure in all models. When the vaccination queue is treated as a structure that allows no mixing, mass vaccination predicts smaller outbreaks than traced vaccination.

The individual-based models are more intuitive than the population-based models. However, the intuition of working with individual objects comes at the price of computational complexity. The agent-based model is significantly more computationally intensive than the other models, particularly in mass vaccination simulations. The predictions given by the individual-based model with heterogeneous mixing are similar enough to justify not using the agent-based model. An agent-based model

that produces a scale-free network could produce different results, but the randomized networks implemented for this thesis appear to be accurately modeled by an appropriate heterogeneous mixing strategy.

The individual-based model with heterogeneous mixing attempts to replicate the mixing strategy of the agent-based model and succeeds for all the scenarios presented in this thesis. However, a more detailed agent-based model may not be replicable by a heterogeneous mixing model.

The individual-based model with homogeneous mixing explicitly models all interactions between individuals, but the mixing strategy is unrealistic. Like the other individual-based models, the homogeneous mixing model is not scalable to large populations. The individual-based models run in linear time with respect to the population size.

The general transitions model can use an arbitrary transition probability generator and is scalable to large populations. However, the general transitions model uses approximations to represent complex population dynamics and is not intuitively easy to code.

The ODE model is the fastest and is scalable to large populations. The wealth of published mathematical representations of disease models makes an ODE model easy to implement. However, the ODE model must also use approximations to represent complex population dynamics. In addition, the outputs of the ODE model follow unrealistic exponential curves.

In conclusion, each model possesses benefits and drawbacks that greatly affect its predictions. The predictions given by each model rely heavily on the mixing and intervention strategies used. Depending on the network structure of a city and the city's implementation of mass vaccination, traced vaccination may be more effective than previously presumed at containing a smallpox outbreak. However, the penalty for underestimating an outbreak can be large, and mass vaccination predicts a low (under 170 deaths in a population of 101,355 people) number of deaths in all models. Thus, it appears that mass vaccination is the "safer" strategy—its worst case prediction is much smaller than the worst case prediction of traced vaccination. Similar

to the paper by Kaplan et. al., this thesis indicates that mass vaccination should be considered further as a standard response technique when preparing for a smallpox outbreak.

Appendix A

Simulator source code

A.1 The agent-based simulator

```
/////////////////////////////////////////////////////////////////
// Agent.h: interface for the Agent class.
//
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
//                         CLASS DESCRIPTION
//
// The Agent class is the highest-fidelity model created for this
// thesis.  People within the Agent model are assigned homes,
// workplaces, and families, and move around in a realistic manner.
// Smallpox is spread through the community under the disease
// parameters predicted by the CDC website.
//
/////////////////////////////////////////////////////////////////

#if !defined(AFX_AGENT_H__C27CC67E__INCLUDED_)
#define AFX_AGENT_H__C27CC67E__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include <iostream>
#include <vector>
#include <cmath>
#include "time.h"
#include "Building.h"
#include "Person.h"
#include "Printer.h"
#include "Averager.h"
#include "Probability.h"
#include "RandNum.h"
using namespace std;

class Agent
{
public:
  Agent();
  virtual ~Agent();

  void RunSimulation();
  void reset();

private:

  // MEMBER FUNCTIONS
  void RunSimulationStep(double step);
  void naturalTransition(Person & per, int queueSize, double step,
                         bool transtime, int numUSusc,
                         int numQ0, int numTSusc);
  void naturalTransitionHelper(Person & per);
  void queueContacts(Person & per);

  bool hasIntervention(double step);

  double comboChoose(double a, int b);
  double miniFact(double a, int b);
  double Fact(int b);
  double min(double a, double b);
  double max(double a, double b);
  void RepCheck();

  // MEMBER VARIABLES
  const double timestep;
  const double DAYSTORUN;
  const double INITIALINFECTIONS;
  const double INTERVENTIONDAY;
  const int BUILDINGNUM;
  double QUEUETIMEBEFORETREATMENT;
  const bool GAUSSIAN;
  const bool MASSVACCINATION;
  bool massvacChanged;
  const double beta;    // Infection rate
  const double c;          // Names generated per index
  const double p;    // Fraction of infectees named by index
  const double N;    // Population size
  const double n;          // Number of vaccinators
  const double mu;         // Service rate (traced vaccination)
  const double h;          // Fraction febrile in stage 3
  const double v_0;    // Vaccine efficacy, stage 0
  const double v_1;    // Vaccine efficacy, stage 1
  const double delta;      // Smallpox death rate
  const double f;    // Vaccination fatality rate
  double inter;    // 1 if intervention has commenced,
                         // otherwise 0

  Person * simul;
  Building * bldgs;
  double * statePop;
  double * nar;

  Printer pr;
  Probability prob;
  long randSeed;
  RandNum r;
  Averager av;
  short flag;
};

#endif // !defined(AFX_AGENT_H__C27CC67E__INCLUDED_)


/////////////////////////////////////////////////////////////////
// Agent.cpp: implementation of the Agent class.
//
/////////////////////////////////////////////////////////////////

#include "Agent.h"
using namespace std;


//
// GLOBAL VARIABLES
//

// number at which this type of building ends
const int ONEPER = 17649;
const int TWOPER = 22649;
const int THREEPER = 40244;
const int FOURPER = 42428;
```

```
const int COLLEGEDORMS = 42615;
const int WORKBLGS = 44427;
const int SCHOOLS = 44448;
const int COLLEGES = 44451;
const int THEATERS = 44454;
const int RESTAURANTS = 44854;
const int PARKS = 44857;
const int HOSPITAL = 44857;

// building occupancies
const int DORMOCC = 175;
const int WORKOCC = 80;
const int SCHOOLOCC = 650;
const int COLLEGEOCC = 11000;
const int THEATEROCC = 2250;
const int RESTOCC = 30;
const int PARKOCC = 10000;
const int HOSPOCC = 1000000;

// number of person at which this living type person ends
const int SINGLE = 17649;
const int DOUBLE = 27649;
const int TRIPLE = 80434;
const int QUAD = 89170;
const int COLLEGEKID = 101355;

// Averaging arrays
double aus[100];
double auinf1[100];
double auinf2[100];
double auinf3[100];
double auinf4[100];
double aq0[100];
double aq1[100];
double aq2[100];
double aq3[100];
double aquar[100];
double atsusc[100];
double atinf1[100];
double atinf2[100];
double atinf3[100];
double atinf4[100];
double ad[100];
double az[100];
double qprob[6];

/////////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////////

Agent::Agent()
: timestep(1.0/48),
  DAYSTORUN(100),
  INITIALINFECTIONS(10),
  INTERVENTIONDAY(25),
  BUILDINGNUM(44858),
  QUEUETIMEBEFORETREATMENT(0.0),
  GAUSSIAN(true),
  MASSVACCINATION(false),
  massvacChanged(false),
  beta(0.00001),
  c(50),
  p(0.5),
  N(101355),
  n(50),
  mu(50),
  h(0.9),
  v_0(0.975),
  v_1(0.975),
  delta(0.3),
  f(0.000001),
  inter(0),
  pr(),
  prob(),
  randSeed(-time(0)),
  r(),
  av((int)DAYSTORUN),
  flag(1)
{
  simul = new Person[(int)N];
  bldgs = new Building[BUILDINGNUM];
  statePop = new double[17];
  nar = new double[17];
  for(int i = 0; i < 17; i++) {
    statePop[i] = 0;
    nar[i] = 0;
  }
}

Agent::~Agent()
{
  delete [] simul;
```

```
  delete [] bldgs;
  delete [] statePop;
  delete [] nar;
}

/******************************************************************
**
**  Runs a single stochastic simulation.
**
**  Requires: nothing
**  Modifies: all people in simulation
**  Returns:  nothing
*/
void Agent::RunSimulation() {
  // PREPROCESSING
  cout << "Beginning preprocessing." << endl;
  for(int i = 0; i < HOSPITAL+1; i++)
    bldgs[i].setID(i);
  for(int i = 0; i < N; i++)
    simul[i].setId(i);

  // DETERMINING OCCUPANCY OF EACH TYPE OF BUILDING
  // 42615 total homes
  // 17649 people living alone
  for(int i = 0; i < ONEPER; i++) {
    bldgs[i].setOccupancy(1);
  }
  // 2-person families
  for(int i = ONEPER; i < TWOPER; i++) {
    bldgs[i].setOccupancy(2);
  }
  // families with child
  for(int i = TWOPER; i < THREEPER; i++) {
    bldgs[i].setOccupancy(3);
  }
  // families with 4 people
  for(int i = THREEPER; i < FOURPER; i++) {
    bldgs[i].setOccupancy(4);
  }
  // college dorms
  for(int i = FOURPER; i < COLLEGEDORMS; i++) {
    bldgs[i].setOccupancy(DORMOCC);
  }

  // work buildings
  for(int i = COLLEGEDORMS; i < WORKBLGS; i++) {
    bldgs[i].setOccupancy(WORKOCC);
  }

  // schools
  for(int i = WORKBLGS; i < SCHOOLS; i++) {
    bldgs[i].setOccupancy(SCHOOLOCC);
  }

  // colleges
  for(int i = SCHOOLS; i < COLLEGES; i++) {
    bldgs[i].setOccupancy(COLLEGEOCC);
  }

  // theaters
  for(int i = COLLEGES; i < THEATERS; i++) {
    bldgs[i].setOccupancy(THEATEROCC);
  }

  // restaurants
  for(int i = THEATERS; i < RESTAURANTS; i++) {
    bldgs[i].setOccupancy(RESTOCC);
  }

  // parks
  for(int i = RESTAURANTS; i < PARKS; i++) {
    bldgs[i].setOccupancy(PARKOCC);
  }

  // hospital
  bldgs[HOSPITAL].setOccupancy(HOSPOCC);


  // DETERMINING HOMES OF PEOPLE AND FAMILY STRUCTURE
  // single people
  int peopleSpot = 0;
  // people 0-17648
  for(int i = 0; i < ONEPER; i++) {
    simul[peopleSpot++].setHome(i);
    bldgs[i].incNumAssigned();
  }
  // people 17649-27648
  for(int i = ONEPER; i < TWOPER; i++) {
    simul[peopleSpot++].setHome(i);
    simul[peopleSpot++].setHome(i);
    bldgs[i].incNumAssigned();
```

```
    bldgs[i].incNumAssigned();
  }
  // people 27649-80433
  for(int i = TWOPER; i < THREEPER; i++) {
    simul[peopleSpot++].setHome(i);
    simul[peopleSpot++].setHome(i);
    simul[peopleSpot++].setHome(i);
    bldgs[i].incNumAssigned();
    bldgs[i].incNumAssigned();
    bldgs[i].incNumAssigned();
  }
  // people 80434-89169
  for(int i = THREEPER; i < FOURPER; i++) {
    simul[peopleSpot++].setHome(i);
    simul[peopleSpot++].setHome(i);
    simul[peopleSpot++].setHome(i);
    simul[peopleSpot++].setHome(i);
    bldgs[i].incNumAssigned();
    bldgs[i].incNumAssigned();
    bldgs[i].incNumAssigned();
    bldgs[i].incNumAssigned();
  }
  // people 89170-101354
  for(int i = FOURPER; i < COLLEGEDORMS; i++) {
    for(int j = 0; j < 65; j++) {
      simul[peopleSpot++].setHome(i);
      bldgs[i].incNumAssigned();
    }
  }
  while(peopleSpot < N) {
    simul[peopleSpot++].setHome(FOURPER +
                                 (int)(r.getNext(&randSeed) *
                                 (COLLEGEDORMS-FOURPER)));
  }

  // DETERMINING WORK OF EACH PERSON
  // people living in single homes or two person families work
  int bldg;
  double rand;
  for(int i = 0; i < DOUBLE; i++) {
    bldg = 0;
    while(!(bldgs[bldg].canAssign())) {
      rand = r.getNext(&randSeed);
      bldg = COLLEGEDORMS + (int)(rand*(WORKBLGS-COLLEGEDORMS));
    }
    simul[i].setWork(bldg);
    bldgs[bldg].incNumAssigned();
  }

  // some people in three person families go to college, others go
  // to work, others go to school

  for(int i = DOUBLE; i < TRIPLE; i++) {
    bldg = 0;
    while(!(bldgs[bldg].canAssign())) {
      rand = r.getNext(&randSeed);
      if(rand < .54)
bldg = COLLEGEDORMS + (int)((r.getNext(&randSeed)) *
                             (WORKBLGS-COLLEGEDORMS));
      else if(rand < .67) {
bldg = WORKBLGS + (int)((r.getNext(&randSeed)) *
                         (SCHOOLS-WORKBLGS));
      }
      else {
bldg = SCHOOLS + r.nextThree(&randSeed);
      }
    }
    simul[i].setWork(bldg);
    bldgs[bldg].incNumAssigned();
  }

  // four person families have two go to work, and the others to
  // school and college

  for(int i = TRIPLE; i < QUAD; i++) {
    bldg = 0;
    while(!(bldgs[bldg].canAssign())) {
      rand = r.getNext(&randSeed);
      if(rand < .5)
bldg = COLLEGEDORMS + (int)((r.getNext(&randSeed)) *
                             (WORKBLGS-COLLEGEDORMS));
      else if(rand < .8)
bldg = WORKBLGS + (int)((r.getNext(&randSeed)) *
                         (SCHOOLS-WORKBLGS));
      else {
bldg = SCHOOLS + r.nextThree(&randSeed);
      }
    }
    simul[i].setWork(bldg);
    bldgs[bldg].incNumAssigned();
  }

  // College students are assigned to one of the three Cambridge
  // universities
  for(int i = QUAD; i < COLLEGEKID; i++) {
    bldg = 0;
    while(!(bldgs[bldg].canAssign())) {
      bldg = SCHOOLS + r.nextThree(&randSeed);
    }
    simul[i].setWork(bldg);
    bldgs[bldg].incNumAssigned();
  }

  // Determine family members, since they are your closest contacts
  for(int i = 0; i < N; i++) {
    bldgs[simul[i].getHome()].addOccupant(simul[i].getId());
    simul[i].updateCurrentBldg(simul[i].getHome());
  }

  for(int i = SINGLE; i < COLLEGEDORMS; i++) {
    vector<int> fam = bldgs[i].getOccupantVector();
    for(int j = 0; j < fam.size(); j++) {
      for(int k = 0; k < fam.size(); k++) {
simul[j].addToFamily(fam.at(k));
      }
    }
  }

  for(int i = 0; i < BUILDINGNUM; i++) {
    bldgs[i].removeAllOccupants();
  }

  // Simulation starts at the beginning of the work/school/college
  // day, ending with sleeping and waking up the next day.
  for(int i = 0; i < N; i++) {
    bldgs[simul[i].getWork()].addOccupant(simul[i].getId());
    simul[i].updateCurrentBldg(simul[i].getWork());
  }

  // Random infections
  for(int i = 0; i < INITIALINFECTIONS; i++) {
    rand = r.getNext(&randSeed);
    if(rand < 12000.0/65000) {
      simul[QUAD+(int)(r.getNext(&randSeed)*12185)].transitionStates(1);
    }
    else {
      simul[(int)(r.getNext(&randSeed)*80000)].transitionStates(1);
    }
  }

  cout << "Preprocessing complete.  Beginning simulation." << endl;

  // Simulation starts at 9:00am on a Monday morning.
  for(double d = 0; d < DAYSTORUN; d += timestep) {
    RunSimulationStep(d);
  }

  // Prints average R of the simulation.
  int avgNumInfected = 0;
  int totalNumInfected = 0;
  for(int j = 0; j < N; j++) {
    if(simul[j].getEverInfected()){
      totalNumInfected++;
      avgNumInfected += simul[j].getNumInfected();
    }
  }

  // Stores output of simulation to global arrays
  for(int j = 0; j < DAYSTORUN; j++) {
    av.changeDayOfStuff(j);
    nar = av.returnDOS();
    aus[j] = nar[0];
    auinf1[j] = nar[1];
    auinf2[j] = nar[2];
    auinf3[j] = nar[3];
    auinf4[j] = nar[4];
    aq0[j] = nar[5];
    aq1[j] = nar[6];
    aq2[j] = nar[7];
    aq3[j] = nar[8];
    aquar[j] = nar[9];
    atsusc[j] = nar[10];
    atinf1[j] = nar[11];
    atinf2[j] = nar[12];
    atinf3[j] = nar[13];
    atinf4[j] = nar[14];
    az[j] = nar[15];
    ad[j] = nar[16];
  }

  nar = NULL;

  // Prints probabilities of finding contacts in each disease stage
  cout << (qprob[1]+0.0)/qprob[5] << endl;
```

```
  cout << (qprob[2]+0.0)/qprob[5] << endl;
  cout << (qprob[3]+0.0)/qprob[5] << endl;

}

/*****************************************************************
**
**  Resets all values of people in simulation to initial values.
**
**  Requires: nothing
**  Modifies: all people in simulation
**  Returns:  nothing
*/
void Agent::reset() {
  massvacChanged = false;

  for(int i = 0; i < N; i++)
    simul[i].reset();

  for(int i = 0; i < BUILDINGNUM; i++)
    bldgs[i].removeAllOccupants();

  for(int i = 0; i < 17; i++)
    statePop[i] = 0;

}


//
// PRIVATE FUNCTIONS
//
void Agent::RunSimulationStep(double step) {
  RepCheck();
  for(int a = 0; a < 17; a++) {
    statePop[a] = 0;
  }
  for(int a = 0; a < N; a++) {
    statePop[simul[a].getState()]++;
  }

  // Prints output to log files at each integer day
  if (step - int(step+.001) < timestep - .00001) {
    //cout << "Day " << (int)(step+.001) << endl;
    pr.PrintDay(statePop);
    av.addToAvg(int(step+.001), statePop);
  }


  // Workday starts at 9:00, goes until 5:00.  Then people may go
  //to a restaurant for two hours or may go home.  Then at 7:00
  // people may go to the movies until 10:00.  Then they go home
  // until 9:00 the next morning.

  // On weekends, people may go to the park from 9:00 to 5:00 and
  // then people may go to a restaurant for two hours or may go
  // home.  Then at 7:00 people may go to the  movies until 10:00.
  // Then they go home until 9:00 the next morning.

  bool weekday;
  if(((int)(step*48) % 7) < 5)
    weekday = true;
  else
    weekday = false;

  bool transtime;
  if((int)((step-(int)step)*48) == 0) {
    flag = 1;
    transtime = true;
  }
  else if((int)((step-(int)step)*48) == 16) {
    flag = 2;
    transtime = true;
  }
  else if((int)((step-(int)step)*48) == 20) {
    flag = 3;
    transtime = true;
  }
  else if((int)((step-(int)step)*48) == 26) {
    flag = 0;
    transtime = true;
  }
  else {
    transtime = false;
  }

  // flag == 1 means during work
  // flag == 2 means eating at a restaurant or at home
  // flag == 3 means at the movies or at home
  // flag == 0 means everyone is at home

  if(transtime) {
    for(int i = 0; i < HOSPITAL; i++) {
      bldgs[i].removeAllOccupants();
    }


    if(flag == 1) {
      for(int i = 0; i < N; i++) {
if(!simul[i].isInQueue()) {
  if(weekday) {
    // everyone goes to work
    bldgs[simul[i].getWork()].addOccupant(simul[i].getId());
    simul[i].updateCurrentBldg(simul[i].getWork());
  }
  else {
    if(r.getNext(&randSeed) < .2) {
      // Person goes to the park
      int randm = RESTAURANTS + r.nextThree(&randSeed);
      bldgs[randm].addOccupant(simul[i].getId());
      simul[i].updateCurrentBldg(randm);
    }
    else {
      // Person stays home
      bldgs[simul[i].getHome()].addOccupant(simul[i].getId());
      simul[i].updateCurrentBldg(simul[i].getHome());
    }
  }
}

      }
    }

    else if(flag == 2) {
      for(int i = 0; i < N; i++) {
// Person goes either home or to a restaurant.  Assume
// people eat out less frequently on weekdays than on
// weekends.
if(!simul[i].isInQueue()) {
  if(weekday) {
    if(r.getNext(&randSeed) < .04) {
      int ran = 0;
      while(!(bldgs[ran].addOccupant(simul[i].getId()))) {
ran = THEATERS + (int)(r.getNext(&randSeed) *
                              (RESTAURANTS-THEATERS));
      }
      simul[i].updateCurrentBldg(ran);
    }
    else {
      // If not at a restaurant, goes home
      bldgs[simul[i].getHome()].addOccupant(simul[i].getId());
      simul[i].updateCurrentBldg(simul[i].getHome());
    }
  }
  else {
    if(r.getNext(&randSeed) < .08) {
      int ran = 0;
      while(!(bldgs[ran].addOccupant(simul[i].getId()))) {
ran = THEATERS + (int)(r.getNext(&randSeed) *
                              (RESTAURANTS-THEATERS));
      }
      simul[i].updateCurrentBldg(ran);
    }
    else {
      // If not at a restaurant, goes home
      bldgs[simul[i].getHome()].addOccupant(simul[i].getId());
      simul[i].updateCurrentBldg(simul[i].getHome());
    }
  }
}

      }
    }

    else if(flag == 3) {
      // People leave restaurants for the movies or home.
      // People leave home for the movies.
      for(int i = 0; i < N; i++) {
if(!simul[i].isInQueue()) {
  unsigned short curbldg = simul[i].getCurrentBldg();
  double prob;
  if(weekday)
    prob = .01;
  else
    prob = .04;
  if(r.getNext(&randSeed) < prob) {
    int ran = COLLEGES + r.nextThree(&randSeed);
    bldgs[ran].addOccupant(simul[i].getId());
    simul[i].updateCurrentBldg(ran);
  }
  else {
    // If doesn't go to movies, goes home
    bldgs[simul[i].getHome()].addOccupant(simul[i].getId());
    simul[i].updateCurrentBldg(simul[i].getHome());
  }
}
      }
```

```
    }

    else {
      // People all go home
      for(int i = 0; i < N; i++) {
if(!simul[i].isInQueue()) {
  bldgs[simul[i].getHome()].addOccupant(simul[i].getId());
  simul[i].updateCurrentBldg(simul[i].getHome());
}
      }
    }

  }

  // Now that people are all in their correct places, we can do
  // disease transitions given their state and the state of people
  // around them.

  // First are the transitions out of the hospital queue.  People
  // who have just been vaccinated go home immediately, but may
  // start working the next day if they feel up to it.
  Person * trans = NULL;
  int queueNum = 0;
  int qFinished = 0;
  int queueSize = bldgs[HOSPITAL].getOccupantNum();
  int totalToQueueChange = (int)min((double)queueSize, n*mu*timestep);

  while(bldgs[HOSPITAL].getOccupantNum() > 0 &&
        queueNum < totalToQueueChange &&
        hasIntervention(step) && qFinished < queueSize) {
    qFinished++;
    queueNum++;

    trans = &simul[bldgs[HOSPITAL].getLastOccupant()];
    bldgs[HOSPITAL].removeLastOccupant();
    int tState = trans->getState();


    if(tState > 8) {
      // Person is not in queue anymore, so the leaving should not
      // affect how many people can be treated in that timestep
      // (the person was not treated)
      queueNum--;
      trans->updateCurrentBldg(trans->getHome());
      bldgs[trans->getHome()].addOccupant(trans->getId());
    }


    // Person has not been in queue long enough to have received
    // treatment
    else if(trans->getTimeInQueue() <
            QUEUETIMEBEFORETREATMENT/timestep ||
            trans->getQueueAltered()) {
      trans->unsetQueueAltered();

      // Inserts person back into queue at random place
      double ran = r.getNext(&randSeed);
      int pTA = (int)ran*bldgs[HOSPITAL].getOccupantNum();
      bldgs[HOSPITAL].insertHospitalOccupant(trans->getId(), pTA);
      queueNum--;
    }

    // Standard queue transition depending upon values of person
    // transitioning
    else {
      trans->queueTransition(r.getNext(&randSeed),
                             r.getNext(&randSeed), f, v_0, v_1, h);

      // Person goes home for the rest of the day
      bldgs[trans->getHome()].addOccupant(trans->getId());
      trans->updateCurrentBldg(trans->getHome());
    }
  }


  // Mass vaccination changes
  if(MASSVACCINATION && !massvacChanged && hasIntervention(step)) {
    massvacChanged = true;
    for(int as = 0; as < N; as++) {
      if(simul[as].getState() < 4) {
simul[as].setAltered();

// People who are untraced enter the vaccination queue
if(simul[as].getState() == 0)
  simul[as].transitionStates(5);
else if(simul[as].getState() == 1)
  simul[as].setState(6);
else if(simul[as].getState() == 2)
  simul[as].setState(7);
else if(simul[as].getState() == 3)
  simul[as].setState(8);
else
  cout << "non existent state" << endl;
      }
    }
  }

  for(int i = 0; i < 17; i++) {
    statePop[i] = 0;
  }

  for(int i = 0; i < N; i++) {
    statePop[simul[i].getState()]++;
  }


  // Standard disease transitions for everyone
  for(int i = 0; i < N; i++) {
    naturalTransition(simul[i], bldgs[HOSPITAL].getOccupantNum(),
                      step, transtime, (int)statePop[0],
                      (int)statePop[5], (int)statePop[10]);
  }

  RepCheck();
}

/*****************************************************************
**
**  Performs natural transition of Person per depending upon
**  community state.
**
**  Requires: nothing
**  Modifies: per
**  Returns:  nothing
*/
void Agent::naturalTransition(Person & per, int queueSize,
                              double step, bool transtime,
                              int numUSusc, int numQ0,
                              int numTSusc) {
  if(per.getAltered()) {
    per.unsetAltered();
    per.incTotalSteps();

    if(per.getState() == 4 ||
       per.getState() == 9 ||
       per.getState() == 14) {
      // Just became symptomatic and isolated, and his contacts
      // should be placed in queue if intervention has begun

      if(hasIntervention(step)) {
queueContacts(per);
      }
      else {
// If intervention has not begun, we allow person to
// transition through states as normal, but set the
// contactAltered flag to true, indicating that the person
// should queue his/her contacts as soon as intervention
// commences

per.setContactAltered();

      }
    }

    else if(per.getState() == 5 || per.getState() == 6 ||
            per.getState() == 7 || per.getState() == 8) {
      // Just got placed in hospital, so the hospital should now
      // hold them

      double randNum = r.getNext(&randSeed);

      // pToAdd puts person in queue randomly compared to
      // everyone else entering queue on this day, but after
      // everyone who has been in the queue already
      bldgs[per.getCurrentBldg()].removeOccupantID(per.getId());
      double dd = (double)bldgs[HOSPITAL].getOccupantNum() -
                  queueSize;
      int pToAdd = (int)(randNum*max(0.0, dd));
      bldgs[HOSPITAL].insertHospitalOccupant(per.getId(), pToAdd);
      per.updateCurrentBldg(HOSPITAL);
    }

    else {
      // Do nothing
    }

    return;
  }

  // Not altered, so time to transition naturally
  per.incTotalSteps();

  // Random chance that person gains contact during this timestep
```

```
  if(per.getCurrentBldg() != per.getHome()) {
    vector<int> v = bldgs[per.getCurrentBldg()].getOccupantVector();
    for(int ve = 0; ve < v.size(); ve++) {
      if(r.getNext(&randSeed) < 100.0/183 &&
          v.at(ve) != per.getId()) {
per.forceAddContact(v.at(ve), (int)(c/p));
      }
    }
  }

  if(per.getContactAltered() && hasIntervention(step)) {
    // Person identified as symptomatic before intervention
    // commenced, now that intervention commenced they should
    // queue their contacts for vaccination
    queueContacts(per);
    per.unsetContactAltered();
  }

  if(per.getState() == 3 || per.getState() == 8 ||
     per.getState() == 13) {
    // Uses gathered information to determine number of people
    // infectious person should infect today

    per.setEverInfected(true);
    int numToInfect = 0;
    int numTests;
    if(per.getState() == 8) {
      numTests = min(bldgs[HOSPITAL].getOccupantNum(), 27);
    }
    else {
      numTests = bldgs[per.getCurrentBldg()].getOccupantNum();
    }
    for(int i = 0; i < numTests; i++) {
      if(r.getNext(&randSeed) < 0.00079) {
numToInfect++;
      }
    }

    per.addNumInfected(numToInfect);

    int numSuscU = 0;
    int numSuscT = 0;
    int numQueue0 = 0;

    // Infects susceptibles proportionately to the population sizes
    while(numToInfect > 0) {
      if(r.getNext(&randSeed) < numUSusc/(numUSusc+numTSusc+numQ0))
numSuscU++;
      else if(r.getNext(&randSeed) <
            (numUSusc+numQ0)/(numUSusc + numTSusc + numQ0))
numQueue0++;
      else
numTSusc++;
      numToInfect--;
    }


    if(per.getState() != 8) {
      // Infects people in the current building with them
      vector<int> vec =
         bldgs[per.getCurrentBldg()].getOccupantVector();

      for(int i = 0; i < vec.size(); i++) {
double randNum;
if(numSuscU == 0 && numQueue0 == 0 && numSuscT == 0)
break;

// Infects non-altered susceptible people
else if(numSuscU > 0 && simul[vec[i]].getState() == 0 &&
                !(simul[vec[i]].getAltered())) {
  simul[vec[i]].transitionStates(1);
  simul[vec[i]].setAltered();
  simul[vec[i]].setIndex(per.getId());
  randNum = r.getNext(&randSeed);
  if(per.getCurrentBldg() != per.getHome()) {
    per.addInfected(vec[i]);
  }
  numSuscU--;
}
else if(numQueue0 > 0 && simul[vec[i]].getState() == 5 &&
                !(simul[vec[i]].getAltered())) {
  simul[vec[i]].transitionStates(6);
  simul[vec[i]].setAltered();
  simul[vec[i]].setIndex(per.getId());
  randNum = r.getNext(&randSeed);
  if(per.getCurrentBldg() != per.getHome()) {
    per.addInfected(vec[i]);
  }
  numQueue0--;
}
else if(numSuscT > 0 && simul[vec[i]].getState() == 10 &&
                !(simul[vec[i]].getAltered())) {
  simul[vec[i]].transitionStates(11);
  simul[vec[i]].setAltered();
  simul[vec[i]].setIndex(per.getId());
  randNum = r.getNext(&randSeed);
  if(per.getCurrentBldg() != per.getHome()) {
    per.addInfected(vec[i]);
  }
  numSuscT--;
}
else {}
      }
      int numLeft = numSuscU + numQueue0 + numSuscT;
      if(numLeft > 0) {
per.addNumInfected(-numLeft);
//cout << "Not enough people became infected" << endl;
      }
    }
    else {
      // Homogeneous mixing in the vaccination queue
      int numAttempts = 100;
      while(numToInfect > 0 && numAttempts > 0) {
numAttempts--;
int aPerson = (int)(r.getNext(&randSeed)*N);
if(simul[aPerson].getState() == 5 &&
        !(simul[aPerson].getAltered())) {
  simul[aPerson].transitionStates(6);
  simul[aPerson].setAltered();
  simul[aPerson].setIndex(per.getId());
  if(randNum < p) {
    per.addInfected(aPerson);
  }
  numToInfect--;
}
      }
    }
  }

  // Determines whether person should stay in current state or
  // transition
  naturalTransitionHelper(per);
}

/*****************************************************************
**
**  Determines and changes state of per strictly through disease
**  transitions.
**
**  Requires: nothing
**  Modifies: per
**  Returns:  nothing
*/
void Agent::naturalTransitionHelper(Person & per) {
  // Transitions do not occur on someone who was just altered to
  // be in the state
  if(per.getAltered()) return;

  // All transitions are based upon transition probabilities of
  // the disease
  if(per.getState() == 0) {
    per.incStepsInState();
    return;
  }
  else if(per.getState() == 1) {
    if(r.getNext(&randSeed) <
       prob.getProb(1, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(2);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 2) {
    if(r.getNext(&randSeed) <
       prob.getProb(2, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(3);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 3) {
    if(r.getNext(&randSeed) <
       prob.getProb(3, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(4);
      // Here altered indicates that contacts should be queued
      per.setAltered();
      return;
    }
```

```
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 4) {
    if(r.getNext(&randSeed) <
       prob.getProb(4, per.getStepsInState(), GAUSSIAN)) {
      if(r.getNext(&randSeed) < delta) {
per.transitionStates(16);
return;
      }
      else {
per.transitionStates(15);
return;
      }
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 5) {
    per.incStepsInState();
    per.incTimeInQueue();
    return;
  }
  else if(per.getState() == 6) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(6, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(7);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 7) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(7, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(8);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 8) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(8, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(14);
      // Here altered indicates that contacts should be queued
      per.setAltered();
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 9) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(9, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(14);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 10) {
    per.incStepsInState();
    return;
  }
  else if(per.getState() == 11) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(11, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(12);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 12) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(12, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(13);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 13) {
    if(r.getNext(&randSeed) <
       prob.getProb(13, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(14);
      // Here altered indicates that contacts should be queued
      per.setAltered();
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 14) {
    if(r.getNext(&randSeed) <
       prob.getProb(14, per.getStepsInState(), GAUSSIAN)) {
      if(r.getNext(&randSeed) < delta) {
per.transitionStates(16);
return;
      }
      else {
per.transitionStates(15);
return;
      }
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 15 || per.getState() == 16) {
    per.incStepsInState();
    return;
  }
  else {
    cout << "not a valid state when updating." << endl;
    return;
  }
}

/*****************************************************************
**
**  Transitions states of contacts of recently symptomatic person
**  and readies them for placement into hospital queue on next
**  timestep.
**
**  Requires: nothing
**  Modifies: per, all types of contacts of per
**  Returns:  nothing
*/
void Agent::queueContacts(Person &per) {
  vector<int> cont = per.getInfectedsVector();
  vector<int> more = per.getFamilyVector();
  for(int i = 0; i < more.size(); i++) {
    cont.push_back(more[i]);
  }

  more = per.getContactVector();

  while((cont.size() < c/p) && (more.size() > 0)) {
    cont.push_back(more.back());
    more.pop_back();
  }

  // cont now holds all contacts to be queued
  int qcont = cont.size();
  for(int i = 0; i < cont.size(); i++) {
    int id = cont.at(i);
    if(r.getNext(&randSeed) < p) {

      if(simul[id].getState() < 4) {
simul[id].setAltered();
      }

      if(id < per.getId() && !simul[id].getAltered())
        simul[id].decTotalSteps();
      else
        simul[id].setQueueAltered();
```

```
      // Transitions untraced people into the vaccination queue
      if(simul[id].getState() == 0) {
simul[id].transitionStates(5);
qcont--;
      }
      else if(simul[id].getState() == 1) {
simul[id].setState(6);
qprob[1]++;
      }
      else if(simul[id].getState() == 2) {
simul[id].setState(7);
qprob[2]++;
      }
      else if(simul[id].getState() == 3) {
simul[id].setState(8);
qprob[3]++;
      }

      else {
qcont--;
      }
    }
    else {
    }
  }
  if(qcont > 0) {
    qprob[5] += qcont;
  }
}

/*****************************************************************
**
**  Returns true if time is past time for intervention to start.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  True iff step > INTERVENTIONDAY
*/
bool Agent::hasIntervention(double step) {
  if(step < INTERVENTIONDAY)
    return false;
  else
    return true;
}


/*****************************************************************
**
**  Returns mathematical choose operation of aCb, defined as:
**  aCb = a!/(b!*(a-b)!)
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  a Choose b
*/
double Agent::comboChoose(double a, int b) {
  double retval;
  if(b > a)
    retval = 0;
  else {
    retval = miniFact(a, b)/Fact(b);
  }
  return retval;
}

/*****************************************************************
**
**
** Performs the multiplication of integers between range a and b.
**
** Requires: a is an integer
** Modifies: nothing
** Returns:  Performs operation b*(b+1)*(b+2)*...*(a-1)*a, a helper
**           function to calculate the choose operation
*/
double Agent::miniFact(double a, int b) {
  double retval = 1;
  for(int i = 0; i < b; i++)
    retval *= (a - i);
  return retval;
}

/*****************************************************************
**
**  Performs the factorial operation.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Performs operation b*(b-1)*(b-2)*...*3*2*1, as a
**            helper function for the choose operation
**/
double Agent::Fact(int b) {
  double retval = 1;
  if(b < 1)
    retval = 1.0;
  else if (b < 3)
    retval = (double)b;
  else {
    for(int i = 2; i <= b; i++)
      retval *= i;
  }
  return retval;
}

/*****************************************************************
**
**  Returns the minimum of a and b.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The smaller of a and b.
**/
double Agent::min(double a, double b) {
  if(a < b)
    return a;
  else
    return b;
}

/*****************************************************************
**
**  Returns the maximum of a and b.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The larger of a and b.
**/
double Agent::max(double a, double b) {
  if(a > b)
    return a;
  else
    return b;
}

/*****************************************************************
**
**  Test function to maintain representation invariant.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  nothing
**/
void Agent::RepCheck() {
  /*
  for(int i = 0; i < BUILDINGNUM; i++) {
    if(bldgs[i].getOccupantNum() > bldgs[i].getOccupancy())
      cout << "building " << i << " holds too many people." << endl;
  }
  for(int i = 0; i < N; i++) {
    Building b = bldgs[simul[i].getCurrentBldg()];
    if(!b.containsID(simul[i].getId())) {
      cout <<  i << " not in current building.  Current bldg = ";
      cout << simul[i].getCurrentBldg() << endl;
      cout << "Actual building containing them is: ";
      for(int j = 0; j < BUILDINGNUM; j++) {
if(bldgs[j].containsID(simul[i].getId()))
  cout << j << endl;
      }
    }
  }


  if(flag == 1) {
    for(int i = COLLEGES; i < RESTAURANTS; i++) {
      if(bldgs[i].getOccupantVector().size() != 0) {
cout << "People should be at work or parks." << endl;
      }
    }
  }
  else if(flag == 2) {
    for(int i = COLLEGEDORMS; i < THEATERS; i++) {
      if(bldgs[i].getOccupantVector().size() != 0) {
cout << "No one should be at work anymore." << endl;
      }
    }
    for(int i = RESTAURANTS; i < PARKS; i++) {
      if(bldgs[i].getOccupantVector().size() != 0) {
cout << "People should be at restaurants or home." << endl;
      }
    }
  }
  else if(flag == 3) {
    for(int i = COLLEGEDORMS; i < COLLEGES; i++) {
      if(bldgs[i].getOccupantVector().size() != 0) {
```

```
cout << "No one should be at work anymore." << endl;
      }
    }
    for(int i = THEATERS; i < PARKS; i++) {
      if(bldgs[i].getOccupantVector().size() != 0) {
cout << "People should be at the movies or home." << endl;
      }
    }
  }
  else if(flag == 0) {
    for(int i = COLLEGEDORMS; i < PARKS; i++) {
      if(bldgs[i].getOccupantVector().size() != 0) {
cout << "Error.  Everyone should be at home now." << endl;
      }
    }
  }

  for(int i = 0; i < N; i++) {
    if(simul[i].isInQueue() &&
       !(simul[i].getCurrentBldg() == HOSPITAL)) {
      cout << "Error.  Should be in hospital." << endl;
    }
  }
  */
}


//
// GLOBAL HELPER FUNCTIONS
//

/*******************************************************************
**
**  Prints dnum to log file output.
**
**  Requires: nothing
**  Modifies: output
**  Returns:  nothing
*/
void Print(double dnum, ostream& output) {
  output << dnum << endl;
}

/*******************************************************************
**
**  Prints day and dnum to log file output.
**
**  Requires: nothing
**  Modifies: output
**  Returns:  nothing
*/
void Print(int day, double dnum, ostream& output) {
  output << day << " \t" << dnum << endl;
}

/*******************************************************************
**
**  Runs the number of simulations desired and produces outputs.
**
**  Requires: nothing
**  Modifies: output
**  Returns:  nothing
*/
int main() {
  // The number of runs to average over
  int numRuns = 10;

  // Number of days in each simulation
  int days = 100;

  // Initializing averaged values to zeroes
  for(int as = 0; as < days; as++) {
    aus[as] = 0;
    auinf1[as] = 0;
    auinf2[as] = 0;
    auinf3[as] = 0;
    auinf4[as] = 0;
    aq0[as] = 0;
    aq1[as] = 0;
    aq2[as] = 0;
    aq3[as] = 0;
    aquar[as] = 0;
    atsusc[as] = 0;
    atinf1[as] = 0;
    atinf2[as] = 0;
    atinf3[as] = 0;
    atinf4[as] = 0;
    az[as] = 0;
    ad[as] = 0;
  }

  for(int as = 0; as < 8; as++)
    qprob[as] = 0;

  // Log file names and streams
  string f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13;
  string f14, f15, f16, f17;
  f1 = "S_0.txt";
  f2 = "I_10.txt";
  f3 = "I_20.txt";
  f4 = "I_30.txt";
  f5 = "I_40.txt";
  f6 = "Q_0.txt";
  f7 = "Q_1.txt";
  f8 = "Q_2.txt";
  f9 = "Q_3.txt";
  f10 = "H.txt";
  f11 = "S_1.txt";
  f12 = "I_11.txt";
  f13 = "I_21.txt";
  f14 = "I_31.txt";
  f15 = "I_41.txt";
  f16 = "Z.txt";
  f17 = "D.txt";

  ofstream S_0out(f1.c_str());
  ofstream I_10out(f2.c_str());
  ofstream I_20out(f3.c_str());
  ofstream I_30out(f4.c_str());
  ofstream I_40out(f5.c_str());
  ofstream Q_0out(f6.c_str());
  ofstream Q_1out(f7.c_str());
  ofstream Q_2out(f8.c_str());
  ofstream Q_3out(f9.c_str());
  ofstream Hout(f10.c_str());
  ofstream S_1out(f11.c_str());
  ofstream I_11out(f12.c_str());
  ofstream I_21out(f13.c_str());
  ofstream I_31out(f14.c_str());
  ofstream I_41out(f15.c_str());
  ofstream Zout(f16.c_str());
  ofstream Dout(f17.c_str());

  Agent a;
  for(int i = 0; i < numRuns; i++) {
    a.RunSimulation();
    a.reset();
    cout << "sim " << i+1 << " of " << numRuns << " is completed.";
    cout << endl;
  }

  // Printing averaged values to log files
  for(int i = 0; i < days; i++) {
    Print(aus[i]/numRuns, S_0out);
    Print(auinf1[i]/numRuns, I_10out);
    Print(auinf2[i]/numRuns, I_20out);
    Print(auinf3[i]/numRuns, I_30out);
    Print(auinf4[i]/numRuns, I_40out);
    Print(aq0[i]/numRuns, Q_0out);
    Print(aq1[i]/numRuns, Q_1out);
    Print(aq2[i]/numRuns, Q_2out);
    Print(aq3[i]/numRuns, Q_3out);
    Print(aquar[i]/numRuns, Hout);
    Print(atsusc[i]/numRuns, S_1out);
    Print(atinf1[i]/numRuns, I_11out);
    Print(atinf2[i]/numRuns, I_21out);
    Print(atinf3[i]/numRuns, I_31out);
    Print(atinf4[i]/numRuns, I_41out);
    Print(az[i]/numRuns, Zout);
    Print(ad[i]/numRuns, Dout);
  }

  // Closing all output streams and terminating program
  S_0out.close();
  I_10out.close();
  I_20out.close();
  I_30out.close();
  I_40out.close();
  Q_0out.close();
  Q_1out.close();
  Q_2out.close();
  Q_3out.close();
  Hout.close();
  S_1out.close();
  I_11out.close();
  I_21out.close();
  I_31out.close();
  I_41out.close();
  Zout.close();
  Dout.close();

  return 0;
}
```

A.2 The individual-based simulator with heterogeneous mixing

```
//////////////////////////////////////////////////////////////////////
// STOHet.h: interface for the STOHet class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
//                          CLASS DESCRIPTION
//
// The STOHet class creates a stochastic simulator using
// heterogeneous mixing of people to determine contacts.
// It can employ either Gaussian or exponential transition
// probabilities.
//
//////////////////////////////////////////////////////////////////////
#if !defined(AFX_STOHET_H__INCLUDED_)
#define AFX_STOHET_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <cmath>
#include "RandNum.h"
#include "time.h"
#include "Probability.h"
#include "Person.h"
#include "Printer.h"
#include "Averager.h"
using namespace std;

class STOHet
{
public:
  STOHet();
  virtual ~STOHet();

  void RunSimulation();
  void reset();

private:
  // MEMBER FUNCTIONS
  void RunSimulationStep(double step);
  void naturalTransition(Person & per, int queueSize, double step,
                        int numUSusc, int numQSusc, int numTSusc);
  void naturalTransitionHelper(Person & per);
  void queueContacts(Person & per);
  int getNumToInfect(int numSusceptible);

  bool hasIntervention(double step);

  double comboChoose(double a, int b);
  double miniFact(double a, int b);
  double Fact(int b);
  double min(double a, double b);
  double max(double a, double b);

  // MEMBER VARIABLES
  const double timestep;
  const double DAYSTORUN;
  const double INITIALINFECTIONS;
  const double INTERVENTIONDAY;
  double QUEUETIMEBEFORETREATMENT;
  const bool GAUSSIAN;
  const bool MASSVACCINATION;
  bool massvacChanged;
  const double beta;      // Infection rate
  const double c;            // Names generated per index
  const double p;      // Fraction of infectees named by index
  const double N;      // Population size
  const double n;            // Number of vaccinators
  const double mu;           // Service rate (traced vaccination)
  const double h;      // Fraction febrile in stage 3
  const double v_0;      // Vaccine efficacy, stage 0
  const double v_1;      // Vaccine efficacy, stage 1
  const double delta;        // Smallpox death rate
  const double f;      // Vaccination fatality rate
  double inter;      // 1 if intervention has commenced,
                           // otherwise 0

  Printer pr;
  Probability prob;
  long randSeed;
  RandNum r;
  Averager av;

  Person * simul;
  vector<Person*> totalQueue;
  double * statePop;
  double * nar;

};

#endif // !defined(AFX_STOHET_H__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// STOHet.cpp: implementation of the STOHet class.
//
//////////////////////////////////////////////////////////////////////

#include "STOHet.h"
#include <fstream>
#include <string>
using namespace std;


//
// GLOBAL AVERAGING ARRAYS
//
double aus[100];
double auinf1[100];
double auinf2[100];
double auinf3[100];
double auinf4[100];
double aq0[100];
double aq1[100];
double aq2[100];
double aq3[100];
double aquar[100];
double atsusc[100];
double atinf1[100];
double atinf2[100];
double atinf3[100];
double atinf4[100];
double ad[100];
double az[100];
double qprob[6];

void Print(double dnum, ostream& output);
void Print(int day, double dnum, ostream& output);

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

STOHet::STOHet()
: timestep(1.0/48),
  DAYSTORUN(100),
  INTERVENTIONDAY(25),
  beta(0.00001),
  c(50),
  p(0.5),
  N(101355),
  r_1(1.0/3),
  r_2(1.0/8),
  r_3(1.0/3),
  r_4(1.0/12),
  n(50),
  mu(50),
  h(0.9),
  v_0(0.975),
  v_1(0.975),
  delta(0.3),
  f(0.000001),
  INITIALINFECTIONS(10),
  inter(0),
  QUEUETIMEBEFORETREATMENT(0.0),
  GAUSSIAN(true),
  MASSVACCINATION(false),
  massvacChanged(false),
  pr(),
  prob(),
  randSeed(-time(0)),
  r(),
  av((int)DAYSTORUN)
```

```
{
  simul = new Person[(int)N];
  for(int i = 0; i < N; i++)
    simul[i].setId(i);

  totalQueue.clear();
  for(int i = 0; i < INITIALINFECTIONS; i++)
    simul[(int)(r.getNext(&randSeed)*N)].transitionStates(1);

  statePop = new double[17];
  nar = new double[17];
  for(int i = 0; i < 17; i++) {
    statePop[i] = 0;
    nar[i] = 0;
  }

}

STOHet::~STOHet()
{
  delete [] simul;
  delete [] statePop;
  delete [] nar;
}

/*****************************************************************
**
**  Runs a single stochastic simulation.
**
**  Requires: nothing
**  Modifies: all people in simulation
**  Returns:  nothing
*/
void STOHet::RunSimulation() {

  // Finding the ten randomly infected people and adding contacts
  // evenly to them for the largest infected population
  int * te = new int[10];
  int asd = 0;
  for(int i = 0; i < 10; i++)
    te[i] = 0;
  for(int a = 0; a < N; a++) {
    if(simul[a].getState() == 1) {
      te[asd] = a;
      asd++;
    }
  }
  for(int a = 0; a < c; a++) {
    simul[te[0]].addContact(99+100*a,(int)c);
    simul[te[1]].addContact(1100+100*a,(int)c);
    simul[te[2]].addContact(12200+100*a,(int)c);
    simul[te[3]].addContact(3300+100*a,(int)c);
    simul[te[4]].addContact(14400+100*a,(int)c);
    simul[te[5]].addContact(5500+100*a,(int)c);
    simul[te[6]].addContact(16600+100*a,(int)c);
    simul[te[7]].addContact(7700+100*a,(int)c);
    simul[te[8]].addContact(18800+100*a,(int)c);
    simul[te[9]].addContact(6000+100*a,(int)c);
  }
  delete [] te;

  // Run simulation
  for (double i = 0; i < DAYSTORUN; i += timestep) {
    RunSimulationStep(i);
  }

  for(int j = 0; j < DAYSTORUN; j++) {
    av.changeDayOfStuff(j);
    nar = av.returnDOS();
    aus[j] = nar[0];
    auinf1[j] = nar[1];
    auinf2[j] = nar[2];
    auinf3[j] = nar[3];
    auinf4[j] = nar[4];
    aq0[j] = nar[5];
    aq1[j] = nar[6];
    aq2[j] = nar[7];
    aq3[j] = nar[8];
    aquar[j] = nar[9];
    atsusc[j] = nar[10];
    atinf1[j] = nar[11];
    atinf2[j] = nar[12];
    atinf3[j] = nar[13];
    atinf4[j] = nar[14];
    az[j] = nar[15];
    ad[j] = nar[16];
  }
  nar = NULL;

  // Determining q probabilities as discussed in Kaplan paper
  cout << (qprob[1]+0.0)/qprob[5] << endl;
  cout << (qprob[2]+0.0)/qprob[5] << endl;
  cout << (qprob[3]+0.0)/qprob[5] << endl;

  // Determining effective R of simulation
  int avgNumInfected = 0;
  int totalNumInfected = 0;
  for(int j = 0; j < N; j++) {
    if(simul[j].getEverInfected()){
      totalNumInfected++;
      avgNumInfected += simul[j].getNumInfected();
    }
  }
  cout << endl << "Average number infected = ";
  cout << (1.0*avgNumInfected)/totalNumInfected << endl;
}

/*****************************************************************
**
**  Resets all values of people in simulation to initial values.
**
**  Requires: nothing
**  Modifies: all people in simulation
**  Returns:  nothing
*/
void STOHet::reset() {

  for(int i = 0; i < N; i++)
    simul[i].reset();

  massvacChanged = false;

  totalQueue.clear();

  for(int i = 0; i < INITIALINFECTIONS; i++)
    simul[(int)(r.getNext(&randSeed)*N)].transitionStates(1);

  for(int i = 0; i < 17; i++)
    statePop[i] = 0;
}


//
// PRIVATE METHODS
//

/*****************************************************************
**
**  Performs a single step of the simulation, transitioning all
**  people correctly.
**
**  Requires: nothing
**  Modifies: all people in simulation
**  Returns:  nothing
*/
void STOHet::RunSimulationStep(double step) {
  // Resets statePop array for accurate counting of numbers in each
  // state
  for(int i = 0; i < 17; i++) {
    statePop[i] = 0;
  }

  // Finds number of people in each state during this timestep, also
  // checks for invalid states
  for(int i = 0; i < N; i++) {
    if(simul[i].getState() < 0 || simul[i].getState() > 16)
      cout << "incorrect state" << endl;
    statePop[simul[i].getState()]++;
  }

  // Prints output to log files at each integer day
  if (step - int(step+.00001) < timestep - .00001) {
    pr.PrintDay(statePop);
    av.addToAvg(int(step+.00001), statePop);
  }

  // Determines total number of people to transition out of the
  // vaccination queue during this day
  int totalChangedThisTimeStep = (int)min((double)totalQueue.size(),
                                          n*mu*timestep);
  int queueNum = 0;
  Person* transitioner = NULL;
  // Tests whether everyone left in the queue is has not been there
  // long enough to transition out--if qFinished >= queueSize, we
  // need to go to the next timestep
  int qFinished = 0;
  int queueSize = totalQueue.size();

  while(totalQueue.size() > 0 &&
        queueNum < totalChangedThisTimeStep &&
        hasIntervention(step) && qFinished < queueSize) {
    qFinished++;
    queueNum++;
```

```
    transitioner = totalQueue[totalQueue.size() - 1];
    int tState = transitioner->getState();
    totalQueue.pop_back();

    if(tState > 8) {
      // Person is not in queue anymore, so the leaving should not
      // affect how many people can be treated in that timestep
      // (the person was not treated)
      queueNum--;
    }

    // Person has not been in queue long enough to have received
    // treatment
    else if(transitioner->getTimeInQueue() <
            QUEUETIMEBEFORETREATMENT/timestep ||
            transitioner->getQueueAltered()) {
      transitioner->unsetQueueAltered();

      // Inserts person back into queue at random place
      double ran = r.getNext(&randSeed);
      int pTA = (int)(ran*totalQueue.size());
      vector<Person*>::iterator ite = totalQueue.begin();
      for(int m = 0; m < pTA; m++)
ite++;
      totalQueue.insert(ite, transitioner);
      queueNum--;
    }

    // Standard queue transition depending upon values of person
    // transitioning
    else {
      transitioner->queueTransition(r.getNext(&randSeed),
                                    r.getNext(&randSeed), f, v_0,
                                    v_1, h);
    }
  }


  // Checking for mass vaccination and if so, queuing all untraced
  // people
  if(MASSVACCINATION && !massvacChanged && hasIntervention(step)) {
    massvacChanged = true;
    for(int i = 0; i < N; i++) {
      if(simul[i].getState() < 4) {
if(simul[i].getState() == 0)
  simul[i].transitionStates(5);
else if(simul[i].getState() == 1)
  simul[i].setState(6);
else if(simul[i].getState() == 2)
  simul[i].setState(7),
else if(simul[i].getState() == 3)
  simul[i].setState(8);
simul[i].setAltered();
      }
    }
  }

  //
  // NON-QUEUE TRANSITIONS BETWEEN STATES
  //

  // Each run through the loop is a person being updated on a
  // particular day

  // Re-evaluating number of people in each state after queue
  // transitions
  for(int i = 0; i < 17; i++) {
    statePop[i] = 0;
  }
  for(int i = 0; i < N; i++) {
    if(simul[i].getState() < 0 || simul[i].getState() > 16)
      cout << "incorrect state" << endl;
    statePop[simul[i].getState()]++;
  }

  // Performs natural transition of people during given timestep
  int numInQueueAtStartOfDay = totalQueue.size();

  for(int i = 0; i < N; i++) {
    naturalTransition(simul[i], numInQueueAtStartOfDay, step,
                      (int)statePop[0], (int)statePop[5],
                      (int)statePop[10]);
  }
}

/*********************************************************************
**
** Performs natural transition of Person per depending upon
** community state.
**
** Requires: nothing
** Modifies: per
** Returns: nothing
*/
void STOHet::naturalTransition(Person & per, int queueSize,
                               double step, int numUSusc,
                               int numQSusc, int numTSusc) {
  if(per.getAltered()) {
    per.unsetAltered();
    per.incTotalSteps();

    if(per.getState() == 4 || per.getState() == 9 ||
       per.getState() == 14) {
      // Just became symptomatic and isolated, and his contacts
      // should be placed in queue if intervention has begun

      if(hasIntervention(step) && !MASSVACCINATION)
queueContacts(per);

      else {
// If intervention has not begun, we allow person to
// transition through states as normal, but set the contact
// Altered flag to true, indicating that the person should
// queue his/her contacts as soon as intervention commences

per.setContactAltered();
      }
    }

    else if(per.getState() == 5 || per.getState() == 6 ||
            per.getState() == 7 || per.getState() == 8) {
      // Just got placed in queues, so the queue should now hold
      // them

      double randNum = r.getNext(&randSeed);

      // pToAdd puts person in queue randomly compared to
      // everyone else entering queue on this day, but after
      // everyone who has been in the queue already
      double dd = (double)totalQueue.size() - queueSize;
      int pToAdd = (int)(randNum*max(0.0, dd));
      vector<Person*>::iterator it = totalQueue.begin();
      for(int m = 0; m < pToAdd; m++)
it++;
      totalQueue.insert(it, &per);
    }

    else {
      // Do nothing
    }

    return;
  }

  // Not altered, so time to transition naturally
  per.incTotalSteps();

  // Random chance that person gains contact during this timestep
  if(r.getNext(&randSeed) < timestep) {
    vector<int> ve = per.getContactVector();
    int newCont;
    if(r.getNext(&randSeed) < 0.85) {
      // Co-worker contact
      newCont = (int)(r.getNext(&randSeed)*80) +
                (int)(per.getId()/80)*80;
    }
    else {
      // Random contact
      newCont = (int)(r.getNext(&randSeed)*80);
    }
    if(0 <= newCont && newCont <= N && newCont != per.getId() {
      per.addContact(newCont,(int)(c/p));
    }
  }

  if(per.getContactAltered() && hasIntervention(step)) {
    // Person identified as symptomatic before intervention
    // commenced, now that intervention commenced they should
    // queue their contacts for vaccination
    queueContacts(per);
    per.unsetContactAltered();
  }

  double numSusceptible = numUSusc + numQSusc + numTSusc;
  if(per.getState() == 3 || per.getState() == 8 ||
     per.getState() == 13) {
    per.alterEverInfected(true);
    // Uses binomial distribution to determine number of people
    // infectious person should infect today
    int numToInfect = getNumToInfect(per.getContactVector().size());
    int numSuscU = 0;
    int numQueue0 = 0;
    int numSuscT = 0;
```

```
    per.addNumInfected(numToInfect);
    while(numToInfect > 0) {
      double randNum = r.getNext(&randSeed);

      // Infects proportionate numbers of each susceptible state
      if(randNum < numUSusc/numSusceptible)
numSuscU++;
      else if(randNum < (numUSusc + numQSusc)/numSusceptible)
numQueue0++;
      else
numSuscT++;
      numToInfect--;
    }

    if(!per.getState() == 8) {
      // Heterogeneous mixing
      vector<int> vec = per.getContactVector();
      for(int i = vec.size()-1; i >= 0; i--) {
double randNum;

int pos = vec.at(i);
if(numSuscU == 0 && numQueue0 == 0 && numSuscT == 0)
  break;

// Infects non-altered susceptible people
else if(numSuscU > 0 && simul[pos].getState() == 0 &&
                !(simul[pos].getAltered())) {
  simul[pos].transitionStates(1);
  simul[pos].setAltered();
  simul[pos].setIndex(per.getId());
  per.addInfected(pos);
  numSuscU--;
}
else if(numQueue0 > 0 && simul[pos].getState() == 5 &&
                !(simul[pos].getAltered())) {
  simul[pos].transitionStates(6);
  simul[pos].setAltered();
  simul[pos].setIndex(per.getId());
  per.addInfected(pos);
  numQueue0--;
}
else if(numSuscT > 0 && simul[pos].getState() == 10 &&
                !(simul[pos].getAltered())) {
  simul[pos].transitionStates(11);
  simul[pos].setAltered();
  simul[pos].setIndex(per.getId());
  per.addInfected(pos);
  numSuscT--;
}
else {}
      }
      int numLeft = numSuscU + numQueue0 + numSuscT;
      if(numLeft > 0) {
per.addNumInfected(-numLeft);
      }
    }

    else {
      numToInfect = numSuscU + numQueue0 + numSuscT;
      int numAttempts = 100;
     while(numToInfect > 0 && numAttempts > 0) {
       numAttempts--;
int aPerson = (int)(r.getNext(&randSeed)*N);
if(simul[aPerson].getState() == 5 &&
           !simul[aPerson].getAltered()) {
  simul[aPerson].transitionStates(6);
          simul[aPerson].setAltered();
          simul[aPerson].setIndex(per.getId());
          per.addInfected(aPerson);
  numToInfect--;
}
      }
    }

  // Determines whether person should stay in current state or
  // transition
  naturalTransitionHelper(per);
}

/******************************************************************
**
**  Transitions Person per naturally as model dictates.
**
**  Requires: nothing
**  Modifies: all people in simulation
**  Returns:  nothing
*/
void STOHet::naturalTransitionHelper(Person & per) {
  // Altered people just got into the state, should not transition
  if(per.getAltered()) return;

  // All transitions dictated by disease parameters
```

```
  if(per.getState() == 0) {
    per.incStepsInState();
    return;
  }
  else if(per.getState() == 1) {
    if(r.getNext(&randSeed) <
       prob.getProb(1, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(2);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 2) {
    if(r.getNext(&randSeed) <
       prob.getProb(2, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(3);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 3) {
    if(r.getNext(&randSeed) <
       prob.getProb(3, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(4);
      // Here altered indicates that contacts should be queued
      per.setAltered();
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 4) {
    if(r.getNext(&randSeed) <
       prob.getProb(4, per.getStepsInState(), GAUSSIAN)) {
      if(r.getNext(&randSeed) < delta) {
per.transitionStates(16);
return;
      }
      else {
per.transitionStates(15);
return;
      }
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 5) {
    per.incStepsInState();
    per.incTimeInQueue();
    return;
  }
  else if(per.getState() == 6) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(6, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(7);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 7) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(7, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(8);
      return;
    }
    else {
      per.incStepsInState();
      return;
    }
  }
  else if(per.getState() == 8) {
    per.incTimeInQueue();
    if(r.getNext(&randSeed) <
       prob.getProb(8, per.getStepsInState(), GAUSSIAN)) {
      per.transitionStates(14);
      // Here altered indicates that contacts should be queued
      per.setAltered();
```

```
        return;
      }
      else {
        per.incStepsInState();
        return;
      }
    }
    else if(per.getState() == 9) {
      per.incTimeInQueue();
      if(r.getNext(&randSeed) <
         prob.getProb(9, per.getStepsInState(), GAUSSIAN)) {
        per.transitionStates(14);
        return;
      }
      else {
        per.incStepsInState();
        return;
      }
    }
    else if(per.getState() == 10) {
      per.incStepsInState();
      return;
    }
    else if(per.getState() == 11) {
      per.incTimeInQueue();
      if(r.getNext(&randSeed) <
         prob.getProb(11, per.getStepsInState(), GAUSSIAN)) {
        per.transitionStates(12);
        return;
      }
      else {
        per.incStepsInState();
        return;
      }
    }
    else if(per.getState() == 12) {
      per.incTimeInQueue();
      if(r.getNext(&randSeed) <
         prob.getProb(12, per.getStepsInState(), GAUSSIAN)) {
        per.transitionStates(13);
        return;
      }
      else {
        per.incStepsInState();
        return;
      }
    }
    else if(per.getState() == 13) {
      if(r.getNext(&randSeed) <
         prob.getProb(13, per.getStepsInState(), GAUSSIAN)) {
        per.transitionStates(14);
        // Here altered indicates that contacts should be queued
        per.setAltered();
        return;
      }
      else {
        per.incStepsInState();
        return;
      }
    }
    else if(per.getState() == 14) {
      if(r.getNext(&randSeed) <
         prob.getProb(14, per.getStepsInState(), GAUSSIAN)) {
        if(r.getNext(&randSeed) < delta) {
per.transitionStates(16);
return;
        }
        else {
per.transitionStates(15);
return;
        }
      }
      else {
        per.incStepsInState();
        return;
      }
    }
    else if(per.getState() == 15 || per.getState() == 16) {
      per.incStepsInState();
      return;
    }
    else {
      cout << "not a valid state when updating." << endl;
      return;
    }
}

/*****************************************************************
**
**  Queues all recorded contacts of Person per.
**
**  Requires: nothing
**  Modifies: all contacts of per
**  Returns:  nothing
*/
void STOHet::queueContacts(Person & per) {
  vector<int> cont = per.getInfectedsVector();
  vector<int> more = per.getContactVector();
  int numToAdd = (int)min(((c/p)-cont.size()), (double)more.size());
  for(int i = 0; i < numToAdd; i++) {
    cont.push_back(more.at(i));
  }

  int contactsNeeded = (int)((c/p) - cont.size());
  int qcont = cont.size();
  if(!MASSVACCINATION) {
    for(int i = 0; i < cont.size(); i++) {
      if(r.getNext(&randSeed) < p) {
int contactId = cont[i];
if(!(contactId >= 0 && contactId < N)) {
  contactId = (int)(r.getNext(&randSeed)*N);
}
int stateOfContact = simul[contactId].getState();

if(stateOfContact < 4)
  simul[contactId].setAltered();

if(contactId < per.getId())
  simul[contactId].decTotalSteps();
else
  simul[contactId].setQueueAltered();

if(stateOfContact == 0) {
  simul[contactId].transitionStates(5);
  qcont--;
}
else if(stateOfContact == 1) {
  simul[contactId].setState(6);
  qprob[1]++;
}
else if(stateOfContact == 2) {
  simul[contactId].setState(7);
  qprob[2]++;
}
else if(stateOfContact == 3) {
  simul[contactId].setState(8);
  qprob[3]++;
}

else {
  qcont--;
}
      }
      else {
      }
    }
    if(qcont > 0) {
      qprob[5] += qcont;
    }
  }
}

/*****************************************************************
**
**  Determines number of people an infectious person should infect
**  using a standard binomial distribution.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  number of people an infectious person should infect
*/
int STOHet::getNumToInfect(int numSusceptible) {
  int retval = 0;
  for(int i = 0; i < numSusceptible; i++) {
    if(r.getNext(&randSeed) < 0.000208) {
      retval++;
    }
  }
  return retval;
}


/*****************************************************************
**
**  Determines whether intervention has commenced yet.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  True iff intervention techniques have begun.
*/
bool STOHet::hasIntervention(double step) {
  if(step < INTERVENTIONDAY)
    return false;
```

```
  else
    return true;
}

/******************************************************************
**
**  Performs the choose operation aCb, defined as: a!/(b!(a-b)!).
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  a Choose b as defined above
*/
double STOHet::comboChoose(double a, int b) {
  double retval;
  if(b > a)
    retval = 0;
  else {
    retval = miniFact(a, b)/Fact(b);
  }
  return retval;
}

/******************************************************************
**
**  Performs the multiplication of integers between range a and b.
**
**  Requires: a is an integer
**  Modifies: nothing
**  Returns:  Performs operation b*(b+1)*(b+2)*...*(a-1)*a, a
**            helper function to calculate the choose operation
**/
double STOHet::miniFact(double a, int b) {
  double retval = 1;
  for(int i = 0; i < b; i++)
    retval *= (a - i);
  return retval;
}

/******************************************************************
**
**  Performs the factorial operation.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Performs operation b*(b-1)*(b-2)*...*3*2*1, as a
**            helper function for the choose operation
*/
double STOHet::Fact(int b) {
  double retval = 1;
  if(b < 1)
    retval = 1.0;
  else if (b < 3)
    retval = (double)b;
  else {
    for(int i = 1; i <=b; i++)
      retval *= i;
  }
  return retval;
}

/******************************************************************
**
**  Returns the smaller of a and b.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The smaller number of a and b
*/
double STOHet::min(double a, double b) {
  if (a < b)
    return a;
  else
    return b;
}

/******************************************************************
**
**  Returns the larger of a and b.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The larger number of a and b
*/
double STOHet::max(double a, double b) {
  if (a > b)
    return a;
  else
    return b;
}

//
```

```
// GLOBAL FUNCTIONS
//

int main() {
  // Number of simulations to average
  int numRuns = 5;

  // Length of each simulation to record
  int days = 100;

  // Initializing averaging arrays
  for(int as = 0; as < days; as++) {
    aus[as] = 0;
    auinf1[as] = 0;
    auinf2[as] = 0;
    auinf3[as] = 0;
    auinf4[as] = 0;
    aq0[as] = 0;
    aq1[as] = 0;
    aq2[as] = 0;
    aq3[as] = 0;
    aquar[as] = 0;
    atsusc[as] = 0;
    atinf1[as] = 0;
    atinf2[as] = 0;
    atinf3[as] = 0;
    atinf4[as] = 0;
    az[as] = 0;
    ad[as] = 0;
  }

  for(int as = 0; as < 8; as++)
    qprob[as] = 0;

  // Opening streams and log files
  string f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13;
  string f14, f15, f16, f17;
  f1 = "S_0.txt";
  f2 = "I_10.txt";
  f3 = "I_20.txt";
  f4 = "I_30.txt";
  f5 = "I_40.txt";
  f6 = "Q_0.txt";
  f7 = "Q_1.txt";
  f8 = "Q_2.txt";
  f9 = "Q_3.txt";
  f10 = "H.txt";
  f11 = "S_1.txt";
  f12 = "I_11.txt";
  f13 = "I_21.txt";
  f14 = "I_31.txt";
  f15 = "I_41.txt";
  f16 = "Z.txt";
  f17 = "D.txt";

  ofstream S_0out(f1.c_str());
  ofstream I_10out(f2.c_str());
  ofstream I_20out(f3.c_str());
  ofstream I_30out(f4.c_str());
  ofstream I_40out(f5.c_str());
  ofstream Q_0out(f6.c_str());
  ofstream Q_1out(f7.c_str());
  ofstream Q_2out(f8.c_str());
  ofstream Q_3out(f9.c_str());
  ofstream Hout(f10.c_str());
  ofstream S_1out(f11.c_str());
  ofstream I_11out(f12.c_str());
  ofstream I_21out(f13.c_str());
  ofstream I_31out(f14.c_str());
  ofstream I_41out(f15.c_str());
  ofstream Zout(f16.c_str());
  ofstream Dout(f17.c_str());

  // Running simulations
  STOHet s;
  for(int i = 0; i < numRuns; i++) {
    s.RunSimulation();
    s.reset();
    cout << "sim " << i+1 << " of " << numRuns << " is finished.";
    cout << endl;
  }

  // Printing outputs
  for(int i = 0; i < days; i++) {
    Print(aus[i]/numRuns, S_0out);
    Print(auinf1[i]/numRuns, I_10out);
    Print(auinf2[i]/numRuns, I_20out);
    Print(auinf3[i]/numRuns, I_30out);
    Print(auinf4[i]/numRuns, I_40out);
    Print(aq0[i]/numRuns, Q_0out);
    Print(aq1[i]/numRuns, Q_1out);
    Print(aq2[i]/numRuns, Q_2out);
```

```
  Print(aq3[i]/numRuns, Q_3out);
  Print(aquar[i]/numRuns, Hout);
  Print(atsusc[i]/numRuns, S_1out);
  Print(atinf1[i]/numRuns, I_11out);
  Print(atinf2[i]/numRuns, I_21out);
  Print(atinf3[i]/numRuns, I_31out);
  Print(atinf4[i]/numRuns, I_41out);
  Print(az[i]/numRuns, Zout);
  Print(ad[i]/numRuns, Dout);
}

// Close all output streams and end program
S_0out.close();
I_10out.close();
I_20out.close();
I_30out.close();
I_40out.close();
Q_0out.close();
Q_1out.close();
Q_2out.close();
Q_3out.close();
Hout.close();
S_1out.close();
I_11out.close();
I_21out.close();
I_31out.close();
I_41out.close();
Zout.close();
Dout.close();

return 0;
}

//
// GLOBAL HELPER FUNCTIONS
//

/******************************************************************
**
**  Prints dnum to log file output.
**
**  Requires: nothing
**  Modifies: output
**  Returns:  nothing
*/
void Print(double dnum, ostream& output) {
  output << dnum << endl;
}

/******************************************************************
**
**  Prints day and dnum to log file output.
**
**  Requires: nothing
**  Modifies: output
**  Returns:  nothing
*/
void Print(int day, double dnum, ostream& output) {
  output << day << " \t" << dnum << endl;
}
```

A.3 The general transitions simulator

```
//////////////////////////////////////////////////////////////////////
// GeneralTransitions.h: interface for the GeneralTransitions class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// CLASS DESCRIPTION
//
// The general transitions class is the first smallpox simulator
// to be able to use arbitrary transition probabilities.  It is a
// deterministic, population-based model, so it can reproduce the
// results of the ODE model.  However, if the GAUSSIAN flag is set
// to true, the probabilities of transitioning states are changed
// to Gaussian probabilities described on the CDC website about
// smallpox, thus creating a more realistic model.  Approximations
// about the people to put in the vaccination queue are still
// described by the Kaplan model, but the queue is treated more
// realistically than in the ODE model.
//
//////////////////////////////////////////////////////////////////////
#if !defined(AFX_GENERALTRANSITIONS_H__INCLUDED_)
#define AFX_GENERALTRANSITIONS_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include "Printer.h"
#include "Probability.h"
using namespace std;

class GeneralTransitions
{
public:
  GeneralTransitions();
  virtual ~GeneralTransitions();

  void RunSimulation();

private:
  // MEMBER FUNCTIONS
  void RunSimulationStep(double step);
  double removeProportions(int state);
  double remPor(int state, vector<double> & v);
  void updateStatePop();

  void addToVec(vector<double> & v, double val);
  bool removeFromEnd(vector<double> & v, double val);
  void insertInVec(vector<double> & source,
                   vector<double> & target, double prob);
  double hasIntervention(double step);

  double min(double a, double b);
  double q(int j, double k);
  double abs(double d);

// MEMBER VARIABLES

  const double timestep;
  const double INITIALINFECTIONS;
  const double DAYSTORUN;
  const double INTERVENTIONDAY;
  bool MASSVACCINATION;
  const bool GAUSSIAN;
  const double beta; // Infection rate
  const double c;       // Names generated per index
  const double p; // Fraction of infectees named by index
  const double N; // Population size
  const double n;       // Number of vaccinators
  const double mu;      // Service rate (traced vaccination)
  const double r_1;     // Disease rate 1
  const double r_2;     // Disease rate 2
  const double r_3;     // Disease rate 3
  const double r_4;     // Disease rate 4
  const double h; // Fraction febrile in stage 3
  const double v_0; // Vaccine efficacy, stage 0
  const double v_1; // Vaccine efficacy, stage 1
  const double delta;  // Smallpox death rate
  const double f; // Vaccination fatality rate
  double inter;         // 1 if intervention has commenced,
                        // otherwise 0

  Printer pr;
  Probability prob;


  double u_susc;
  vector<double> u_inf1;
  vector<double> u_inf2;
  vector<double> u_inf3;
  vector<double> u_inf4;
  vector<double> q_0;
  vector<double> q_1;
  vector<double> q_2;
  vector<double> q_3;
  vector<double> quar;
  double t_susc;
  vector<double> t_inf1;
  vector<double> t_inf2;
  vector<double> t_inf3;
  vector<double> t_inf4;
  double z;
  double d;

  double statePop[17];
};

#endif // !defined(AFX_GENERALTRANSITIONS_H__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// GeneralTransitions.cpp: implementation of the GeneralTransitions
// class.
//
//////////////////////////////////////////////////////////////////////

#include "GeneralTransitions.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

GeneralTransitions::GeneralTransitions()
: timestep(1.0/48),
  INITIALINFECTIONS(10),
  DAYSTORUN(100),
  INTERVENTIONDAY(25),
  MASSVACCINATION(true),
  GAUSSIAN(true),
  beta(0.00001),
  c(50),
  p(0.5),
  N(101355),
  n(50),
  mu(50),
  r_1(1.0/5),
  r_2(1.0/7),
  r_3(1.0/3),
  r_4(1.0/16),
  h(0.9),
  v_0(0.975),
  v_1(0.975),
  delta(0.3),
  f(0.000001),
  inter(0),
  pr(),
  prob(),
  u_susc(0),
  u_inf1(),
  u_inf2(),
  u_inf3(),
  u_inf4(),
  q_0(),
  q_1(),
  q_2(),
  q_3(),
  quar(),
  t_susc(0),
  t_inf1(),
  t_inf2(),
  t_inf3(),
  t_inf4(),
  z(0),
  d(0)

{
  for(int i = 0; i < 17; i++)
    statePop[i] = 0;

  u_inf1.clear();
  u_inf2.clear();
```

```
  u_inf3.clear();
  u_inf4.clear();
  q_0.clear();
  q_1.clear();
  q_2.clear();
  q_3.clear();
  quar.clear();
  t_inf1.clear();
  t_inf2.clear();
  t_inf3.clear();
  t_inf4.clear();

  u_inf1.push_back(INITIALINFECTIONS);
  u_susc = N-INITIALINFECTIONS;
}

GeneralTransitions::~GeneralTransitions()
{
}


void GeneralTransitions::RunSimulation() {
  double i;
  for(i = 0; i < DAYSTORUN; i += timestep) {
    RunSimulationStep(i);
  }
}



////////////////////////////////////////////////////////////////////
// PRIVATE METHODS
////////////////////////////////////////////////////////////////////

/******************************************************************
**
**  Runs a step of the simulation, and outputs numbers in each
**  state to log files
**
**  Requires: nothing
**  Modifies: Log files associated with each ostream object.
**  Returns:  nothing
*/
void GeneralTransitions::RunSimulationStep(double step) {
  updateStatePop();
  if (step - int(step+.00001) < timestep - .00001) {
    pr.PrintDay(statePop);
  }


  inter = hasIntervention(step);

  double I_3 = statePop[3] + statePop[8] + statePop[13];
  double Q = statePop[5] + statePop[6] + statePop[7] + statePop[8];
  double R_0 = beta*(statePop[0] + statePop[5] + statePop[10])/r_3;

  if(!inter) { // NO QUEUES OR QUARANTINE YET
    // NO ONE IN ANY QUEUE OR TRACED STATES
    // TRANSITIONS ARE ONLY NATURAL DISEASE PROGRESSIONS
    // DO THEM IN REVERSE ORDER TO AVOID TRANSITIONING SOME PEOPLE
    // FASTER THAN THEY SHOULD BE TRANSFERRED

    // TRANSITIONING OUT OF SYMPTOMATIC INFECTED INTO RECOVERED
    // AND DEAD
    double numToTransfer = 0;
    double totalNumToTransfer = 0;
    totalNumToTransfer = removeProportions(4);
    int i =0;

    z += (1-delta)*totalNumToTransfer;
    d += delta*totalNumToTransfer;


    // TRANSITIONING OUT OF ASYMPTOMATIC INTO SYMPTOMATIC INFECTED
    totalNumToTransfer = removeProportions(3);

    vector<double>::iterator it = u_inf4.begin();
    if(totalNumToTransfer > 0)
      u_inf4.insert(it, totalNumToTransfer);


    // TRANSITIONING OUT OF VACCINE-INSENSITIVE INTO ASYMPTOMATIC
    totalNumToTransfer = removeProportions(2);

    it = u_inf3.begin();
    if(totalNumToTransfer > 0)
      u_inf3.insert(it, totalNumToTransfer);


    // TRANSITIONING OUT OF VACCINE-SENSITIVE INTO VACC. INSENSITIVE
    totalNumToTransfer = removeProportions(1);
    it = u_inf2.begin();
    if(totalNumToTransfer > 0)
      u_inf2.insert(it, totalNumToTransfer);


    // TRANSITIONING OUT OF SUSCEPTIBLE INTO VACCINE-SENSITIVE
    totalNumToTransfer = min(statePop[0],
                             timestep*beta*I_3*statePop[0]);
    it = u_inf1.begin();
    if(totalNumToTransfer > 0) {
      u_inf1.insert(it, totalNumToTransfer);
      u_susc -= totalNumToTransfer;
    }
  }
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////

  else {  // QUEUES AND QUARANTINE ARE WORKING
    double numToTransfer = 0;
    double totalNumToTransfer = 0;
    double queueTreatable;
    if (Q < 0) {
      cout << "error in Q" << endl;
    }
    else if(Q == 0) {
      queueTreatable = 0;
    }
    else {
      queueTreatable = min(timestep*mu*n, Q)/Q;
    }

    // TRANSITIONING OUT OF TRACED SYMPTOMATIC INTO RECOVERED/DEAD
    int i = 0;
    totalNumToTransfer = removeProportions(14);

    z += (1-delta)*totalNumToTransfer;
    d += delta*totalNumToTransfer;

    // TRANSITIONING OUT OF ASYMPTOMATIC INTO SYMPTOMATIC INFECTED
    totalNumToTransfer = removeProportions(13);

    vector<double>::iterator it;
    if(totalNumToTransfer > 0) {
      it = t_inf4.begin();
      t_inf4.insert(it, totalNumToTransfer);
    }

    // TRANSITIONING OUT OF VACCINE-INSENSITIVE INTO ASYMPTOMATIC
    totalNumToTransfer = removeProportions(12);

    if(totalNumToTransfer > 0) {
      it = t_inf3.begin();
      t_inf3.insert(it, totalNumToTransfer);
    }

    // TRANSITIONING OUT OF VACCINE-SENSITIVE INTO VACC. INSENSITIVE
    totalNumToTransfer = removeProportions(11);

    if(totalNumToTransfer > 0) {
      it = t_inf2.begin();
      t_inf2.insert(it, totalNumToTransfer);
    }

    // TRANSITIONING OUT OF TRACED SUSCEPTIBLE INTO VACCINE-SENSITIVE
    totalNumToTransfer = min(statePop[10],
                             timestep*beta*I_3*statePop[10]);
    it = t_inf1.begin();
    if(totalNumToTransfer > 0) {
      t_inf1.insert(it, totalNumToTransfer);
      t_susc -= totalNumToTransfer;
    }

    // TRANSITIONING OUT OF QUARANTINE INTO SYMPTOMATIC
    totalNumToTransfer = removeProportions(9);

    addToVec(t_inf4, totalNumToTransfer);
    // new state was already created by transition from
    // traced asymptomatic


    // TRANSITIONING OUT OF 3RD QUEUE STATE INTO QUARANTINE,
    // T_INF3, T_INF4, DEAD
    // ASSUME THAT QUEUE TRANSITIONS HAPPEN BEFORE NATURAL
    // TRANSITIONS FOR ALL QUEUING STATES
    totalNumToTransfer = statePop[8]*queueTreatable;
    it = quar.begin();
    if(totalNumToTransfer > 0) {
      quar.insert(it, h*(1-f)*totalNumToTransfer);
      addToVec(t_inf3, (1-h)*(1-f)*totalNumToTransfer);
      d += f*totalNumToTransfer;
      removeFromEnd(q_3, totalNumToTransfer);
    }
```

```
totalNumToTransfer = removeProportions(8);
addToVec(t_inf4, totalNumToTransfer);
// new state was already created by transition from
// traced asymptomatic


// TRANSITIONING OUT OF 2ND QUEUE STATE INTO 3RD QUEUE STATE,
// T_INF2, DEAD
totalNumToTransfer = statePop[7]*queueTreatable;
if(totalNumToTransfer > 0) {
  addToVec(t_inf2, totalNumToTransfer);
  d += f*totalNumToTransfer;
  removeFromEnd(q_2, totalNumToTransfer);
}

totalNumToTransfer = removeProportions(7);

it = q_3.begin();
if(totalNumToTransfer > 0)
  q_3.insert(it, totalNumToTransfer); // Natural transition


// TRANSITIONING OUT OF 1ST QUEUE STATE INTO 2ND QUEUE STATE,
// T_INF1, RECOVERED, DEAD
totalNumToTransfer = statePop[6]*queueTreatable;
if(totalNumToTransfer > 0) {
  addToVec(t_inf1, (1-f)*(1-v_1)*totalNumToTransfer);
  z += (1-f)*v_1*totalNumToTransfer;
  d += f*totalNumToTransfer;
}
removeFromEnd(q_1, totalNumToTransfer);

totalNumToTransfer = removeProportions(6);

it = q_2.begin();
if(totalNumToTransfer > 0)
  q_2.insert(it, totalNumToTransfer); // Natural transition


// TRANSITIONING OUT OF SUSCEPTIBLE QUEUE STATE INTO 1ST QUEUE
// STATE, T_SUSC, RECOVERED, DEAD
totalNumToTransfer = statePop[5]*queueTreatable;
t_susc += (1-f)*(1-v_0)*totalNumToTransfer;
z += (1-f)*v_0*totalNumToTransfer;
d += f*totalNumToTransfer;

removeFromEnd(q_0, totalNumToTransfer);


double updatedSizeOfQ_0 = 0;
for(i = 0; i < q_0.size(); i++) {
  updatedSizeOfQ_0 += q_0.at(i);
}
totalNumToTransfer = timestep*beta*I_3*updatedSizeOfQ_0;
it = q_1.begin();
if(totalNumToTransfer > 0) {
  q_1.insert(it, totalNumToTransfer);
  removeFromEnd(q_0, totalNumToTransfer);
}

// TRANSITIONING OUT OF UNTRACED SYMPTOMATIC INTO RECOVERED
// AND DEAD
totalNumToTransfer = removeProportions(4);

z += (1-delta)*totalNumToTransfer;
d += delta*totalNumToTransfer;


// TRANSITIONING OUT OF UNTRACED INFECTIOUS INTO QUEUED,
// UNTRACED SYMPTOMATIC
double kappa = (c - p*R_0)*r_3*I_3/N;
double lambda1 = q(1, kappa)*beta*statePop[0]/(r_3 + kappa);
double lambda2 = q(2, kappa)*beta*statePop[0]/(r_3 + kappa);
double lambda3 = q(3, kappa)*beta*statePop[0]/(r_3 + kappa);

totalNumToTransfer = min(statePop[3],
                         timestep*((c-p*R_0)*statePop[3]/N +
                         p*lambda3)*r_3*I_3);

addToVec(q_3, totalNumToTransfer);
removeFromEnd(u_inf3, totalNumToTransfer);


totalNumToTransfer = removeProportions(3);

it = u_inf4.begin();
if(totalNumToTransfer > 0)
  u_inf4.insert(it, totalNumToTransfer);


// TRANSITIONING OUT OF VACCINE INEFFECTIVE INTO QUEUE,
// INFECTIOUS
totalNumToTransfer = min(statePop[2],
                         timestep*((c-p*R_0)*statePop[2]/N +
                         p*lambda2)*r_3*I_3);

addToVec(q_2, totalNumToTransfer);
removeFromEnd(u_inf2, totalNumToTransfer);

totalNumToTransfer = removeProportions(2);

it = u_inf3.begin();
if(totalNumToTransfer > 0)
  u_inf3.insert(it, totalNumToTransfer);


// TRANSITIONING OUT OF VACCINE EFFECTIVE INTO QUEUE, VACCINE
// INEFFECTIVE
totalNumToTransfer = min(statePop[1],
                         timestep*((c-p*R_0)*statePop[1]/N +
                         p*lambda1)*r_3*I_3);

addToVec(q_1, totalNumToTransfer);
removeFromEnd(u_inf1, totalNumToTransfer);

totalNumToTransfer = removeProportions(1);

it = u_inf2.begin();
if(totalNumToTransfer > 0)
  u_inf2.insert(it, totalNumToTransfer);


// TRANSITIONING OUT OF UNTRACED SUSCEPTIBLE INTO Q_0,
// UNTRACED VACCINE EFFECTIVE
totalNumToTransfer = min(statePop[0],
                    timestep*((c-p*R_0)*statePop[0]/N)*r_3*I_3);
it = q_0.begin();
q_0.insert(it, totalNumToTransfer);
u_susc -= totalNumToTransfer;

totalNumToTransfer = timestep*beta*I_3*u_susc;
u_susc -= totalNumToTransfer;
it = u_inf1.begin();
if(totalNumToTransfer > 0)
  u_inf1.insert(it, totalNumToTransfer);


// MASS VACCINATION
if(MASSVACCINATION && inter) {
  int j;
  double stateSize = u_susc;
  it = q_0.begin();
  q_0.insert(it, stateSize);
  stateSize = u_susc = 0;

  int queuelen = q_1.size();
  int inflen = u_inf1.size();
  if(inflen <= queuelen) {
for(j = 0; j < inflen; j++) {
  q_1.at(j) += u_inf1.at(j);
}
  }
  else {
for(j = 0; j < queuelen; j++) {
  q_1.at(j) += u_inf1.at(j);
}
for(j = queuelen; j < inflen; j++) {
  q_1.push_back(u_inf1.at(j));
}
  }
  u_inf1.clear();

  queuelen = q_2.size();
  inflen = u_inf2.size();
  if(inflen <= queuelen) {
for(j = 0; j < inflen; j++) {
  q_2.at(j) += u_inf2.at(j);
}
  }
  else {
for(j = 0; j < queuelen; j++) {
  q_2.at(j) += u_inf2.at(j);
}
for(j = queuelen; j < inflen; j++) {
  q_2.push_back(u_inf2.at(j));
}
  }
  u_inf2.clear();

  queuelen = q_3.size();
  inflen = u_inf3.size();
  if(inflen <= queuelen) {
for(j = 0; j < inflen; j++) {
```

```
    q_3.at(j) += u_inf3.at(j);
  }
        }
        else {
  for(j = 0; j < queuelen; j++) {
    q_3.at(j) += u_inf3.at(j);
  }
  for(j = queuelen; j < inflen; j++) {
    q_3.push_back(u_inf3.at(j));
  }
        }
        u_inf3.clear();

      }

    }
  }


/*****************************************************************
**
**  Returns the total number of people transitioned out of state
**  during the timestep, depending upon how long they have been in
**  the state.
**
**  Requires: nothing
**  Modifies: It calls another function which modifies the vector
**            holding the people in the given state
**  Effects:  Returns the total number of people removed from the
**            state due to natural transitions
**/
double GeneralTransitions::removeProportions(int state) {
  if(state == 0 || state == 10 || state == 15 || state == 16) {
    cout << "can't do vector operations on non-vectors" << endl;
    return -1;
  }
  else if(state == 1)
    return remPor(state, u_inf1);
  else if(state == 2)
    return remPor(state, u_inf2);
  else if(state == 3)
    return remPor(state, u_inf3);
  else if(state == 4)
    return remPor(state, u_inf4);
  else if(state == 5)
    return remPor(state, q_0);
  else if(state == 6)
    return remPor(state, q_1);
  else if(state == 7)
    return remPor(state, q_2);
  else if(state == 8)
    return remPor(state, q_3);
  else if(state == 9)
    return remPor(state, quar);
  else if(state == 11)
    return remPor(state, t_inf1);
  else if(state == 12)
    return remPor(state, t_inf2);
  else if(state == 13)
    return remPor(state, t_inf3);
  else if(state == 14)
    return remPor(state, t_inf4);
  else {
    cout << "not a valid state" << endl;
    return -1;
  }
}

/*****************************************************************
**
**  Returns the total number of people transitioned out of state
**  during the timestep, depending upon how long they have been in
**  the state.
**
**  Requires: nothing
**  Modifies: v
**  Effects:  Returns the total number of people removed from the
**            state due to natural transitions
**/
double GeneralTransitions::remPor(int state, vector<double> & v) {
  double totalNumToTransfer = 0;
  double numToTransfer;
  int i;
  for(i = 0; i < v.size(); i++) {
    // Probability that people in the state for i steps transition
    // now
    double probab = prob.getProb(state, i, GAUSSIAN);
    numToTransfer = 0;

    // Transferring that number of people out of the vector
    numToTransfer = probab*v.at(i);
    if(numToTransfer > 0 && abs(numToTransfer) < 99990) {
      v.at(i) -= numToTransfer;
      if(probab == 1 || v.at(i) <= 0)
v.pop_back();
      totalNumToTransfer += numToTransfer;
    }

    // Tests for correctness
    else if (numToTransfer == 0) {}
    else {
      if(probab < 0)
cout << "Probab less than 0: probab = " << probab << endl;
      else if (v.at(i) < 0) {
cout << "vector holds negative people. v.at(i) = ";
cout << v.at(i) << endl;
      }
    }
  }
  return totalNumToTransfer;
}

/*****************************************************************
**
**  Calculates number of people in each state during the timestep
**
**  Requires: nothing
**  Modifies: statePop array
**  Returns:  nothing
*/
void GeneralTransitions::updateStatePop() {
  int a;
  for(int i = 0; i < 17; i++)
    statePop[i] = 0;

  statePop[0] += u_susc;

  for(a = 0; a < u_inf1.size(); a++)
    statePop[1] += u_inf1.at(a);
  for(a = 0; a < u_inf2.size(); a++)
    statePop[2] += u_inf2.at(a);
  for(a = 0; a < u_inf3.size(); a++)
    statePop[3] += u_inf3.at(a);
  for(a = 0; a < u_inf4.size(); a++)
    statePop[4] += u_inf4.at(a);
  for(a = 0; a < q_0.size(); a++)
    statePop[5] += q_0.at(a);
  for(a = 0; a < q_1.size(); a++)
    statePop[6] += q_1.at(a);
  for(a = 0; a < q_2.size(); a++)
    statePop[7] += q_2.at(a);
  for(a = 0, a < q_3.size(); a++)
    statePop[8] += q_3.at(a);
  for(a = 0; a < quar.size(); a++)
    statePop[9] += quar.at(a);

  statePop[10] = t_susc;

  for(a = 0; a < t_inf1.size(); a++)
    statePop[11] += t_inf1.at(a);
  for(a = 0; a < t_inf2.size(); a++)
    statePop[12] += t_inf2.at(a);
  for(a = 0; a < t_inf3.size(); a++)
    statePop[13] += t_inf3.at(a);
  for(a = 0; a < t_inf4.size(); a++)
    statePop[14] += t_inf4.at(a);

  statePop[15] = z;
  statePop[16] = d;
}


/*****************************************************************
**
**  Adds val people to the front of the state vector v.
**
**  Requires: val > 0
**  Modifies: v
**  Effects:  Adds people to new state vector
**/
void GeneralTransitions::addToVec(vector<double> & v, double val) {
  if (!(v.empty())) {
    v.front() += val;
  }
  else {
    vector<double>::iterator it = v.begin();
    v.insert(it, val);
  }
}

/*****************************************************************
**
**  Removes val people from the end of the vector, corresponding to
**  queued people receiving vaccination (first come, first served).
```

```
**
**  Requires: val > 0
**  Modifies: v
**  Effects:  Returns boolean indicating whether everyone supposed
**            to be removed is, should always return true
**/
bool GeneralTransitions::removeFromEnd(vector<double> & v,
                                       double val) {
  int begin = v.size() -1;
  for(int i = begin; i >= 0; i--) {
    if(val - v.at(i) < 0) {
      // Only some of the people in this state will be removed
      v.at(i) -= val;
      val = 0;
      break;
    }
    else {
      val -= v.at(i);
      // all people in this state move, so state is empty
      v.at(i) = 0;
      v.pop_back();
    }
  }
  bool retval = (!((int)val));
  // returns 1 if all people supposed to be removed are
  if(retval)
    return retval;
  else {
    cout << "not everyone removed from vector" << endl;
    return retval;
  }
}

void GeneralTransitions::insertInVec(vector<double> & source,
                                     vector<double> & target,
                                     double prob) {
  for(int i = 0; i < source.size(); i++) {
    if(i < target.size()) {
      target.at(i) += source.at(i)*prob;
    }
    else {
      target.push_back(source.at(i)*prob);
    }
    source.at(i) -= source.at(i)*prob;
  }
}


/*******************************************************************
**
**  Calculates whether intervention has commenced.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  1 if intervention has commenced, 0 otherwise
*/
double GeneralTransitions::hasIntervention(double step) {
  if (step < INTERVENTIONDAY)
    return 0;
  else return 1;
}

/*******************************************************************
**
**  Calculates minimum of two numbers
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Minimum of a and b
*/
double GeneralTransitions::min(double a, double b) {
  if (a < b)
    return a;
  else
    return b;
}

/*******************************************************************
**
**  Returns the conditional probability that a contact of an
**  index detected at time t is in stage j of disease given that
**  the contact has not been traced by time t (see Kaplan).
**
**  Requires: nothing
**  Modifies: nothing
**  Effects:  Returns a double value between 0 and 1 indicating
**            conditional probability
**/
double GeneralTransitions::q(int j, double k) {
  double retval = 1;    // Probability q1 (Eq. 19 in Kaplan paper)
  double rk = 0;
  double rj = 0;
  if(j == 1)
    rj = r_1;
  else if(j == 2)
    rj = r_2;
  else if (j == 3)
    rj = r_3;
  else if (j == 4)
    rj = r_4;
  else
    cout << "Error in calculating q (rj)" << endl;

  for (int i = 1; i <= j - 1; i++) {
    if(i == 1)
      rk = r_1;
    else if (i == 2)
      rk = r_2;
    else if (i == 3)
      rk = r_3;
    else if (i == 4)
      rk = r_4;
    else
      cout << "Error in calculating q (rk)" << endl;

    retval *= rk/(rk + r_3 + k);
  }

  retval *= (r_3 + k)/(rj + r_3 + k);

  return retval;
}

/*******************************************************************
**
**  Returns the absolute value of d.
**
**  Requires: nothing
**  Modifies: nothing
**  Effects:  Returns absolute value of d
**/
double GeneralTransitions::abs(double d) {
  if (d < 0)
    return -1*d;
  else
    return d;
}
```

A.4 The ODE simulator

```
//////////////////////////////////////////////////////////////////////
// ODEpopbased.h: interface for the ODEpopbased class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
//    CLASS DESCRIPTION
//
// The ODE model is the most rudimentary smallpox simulator.  It
// consists of seventeen different states, and the transitions
// between states are governed by equations given in Edward
// Kaplan's paper "Emergency response to a smallpox attack: The
// case for mass vaccination," except for the transition out of the
// quarantined state, in which everyone goes to the symptomatic and
// isolated state rather than some going into the asymptomatic and
// infectious state.
//
//////////////////////////////////////////////////////////////////////
#if !defined(AFX_ODEPOPBASED_H__INCLUDED_)
#define AFX_ODEPOPBASED_H__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <iostream>
#include <string>
#include <cmath>
#include "Printer.h"
using namespace std;



class ODEpopbased
{
public:
  ODEpopbased();
  virtual ~ODEpopbased();

  void runSimulation();

private:
  // MEMBER FUNCTIONS
  void runSimulationStep(double step);
  double updateS_0(double I_3, double R_0);
  double updateI_10(double I_3, double R_0, double lambda1);
  double updateI_20(double I_3, double R_0, double lambda2);
  double updateI_30(double I_3, double R_0, double lambda3);
  double updateI_40();
  double updateQ_0(double I_3, double R_0, double Q);
  double updateQ_1(double I_3, double R_0, double lambda1, double Q);
  double updateQ_2(double I_3, double R_0, double lambda2, double Q);
  double updateQ_3(double I_3, double R_0, double lambda3, double Q);
  double updateH(double Q);
  double updateS_1(double I_3, double Q);
  double updateI_11(double I_3, double Q);
  double updateI_21(double Q);
  double updateI_31(double Q);
  double updateI_41();
  double updateZ(double Q);
  double updateD(double Q);
  double min(double a, double b);
  double hasIntervention();
  double q(int j, double k);

  // MEMBER VARIABLES
  const double beta;      // Infection rate
  const double c;         // Names generated per index
  const double p;   // Fraction of infectees named by index
  const double N;   // Population size
  const double r_1;       // Disease stage 1 rate
  const double r_2;       // Disease stage 2 rate
  const double r_3;       // Disease stage 3 rate
  const double r_4;       // Disease stage 4 rate
  const double n;         // Number of vaccinators
  const double mu;        // Service rate (traced vaccination)
  const double h;   // Fraction febrile in stage 3
  const double v_0;   // Vaccine efficacy, stage 0
  const double v_1;   // Vaccine efficacy, stage 1
  const double delta;     // Smallpox death rate
  const double f;   // Vaccination fatality rate



  const double timestep;
  const double DAYSTORUN;
  const double INITIALINFECTIONS;
  bool MASSVACCINATION;
  const double INTERVENTIONDAY;

  double statePop[17];
  double statePopTemp[17];

  double * tempArray;

  double inter;  // 1 if intervention has commenced, otherwise 0

  double day;

  Printer pr;


  int totstep;

  double * s_0;
  double * q_0;
  double * s_1;
  double * kap;
  double * r_0;
  double * lam1;
  double * lam2;
  double * lam3;

};

#endif // !defined(AFX_ODEPOPBASED_H__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// ODEpopbased.cpp: implementation of the ODEpopbased class.
//
//////////////////////////////////////////////////////////////////////

#include "ODEpopbased.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
bool newway = false;
ODEpopbased::ODEpopbased()
  : timestep(0.001),
    DAYSTORUN(100),
    INITIALINFECTIONS(10),
    MASSVACCINATION(false),
    INTERVENTIONDAY(25),
    beta(0.00001),
    c(50),
    p(0.5),
    N(101355),
    r_1(1.0/5),
    r_2(1.0/7),
    r_3(1.0/3),
    r_4(1.0/16),
    n(50),
    mu(50),
    h(0.9),
    v_0(0.975),
    v_1(0.975),
    delta(0.3),
    f(0.000001),
    day(0),
    inter(0),
    pr(),
    totstep(0)
{
  for(int i = 0; i < 17; i++) {
    statePop[i] = 0;
    statePopTemp[i] = 0;
  }

  statePop[1] = INITIALINFECTIONS;// Initial infections
  statePop[0] = N - statePop[1]; // All others susceptible

  tempArray = new double[(int)DAYSTORUN];
  s_0 = new double[(int)(DAYSTORUN/timestep)];
  q_0 = new double[(int)(DAYSTORUN/timestep)];
  s_1 = new double[(int)(DAYSTORUN/timestep)];
  kap = new double[(int)(DAYSTORUN/timestep)];
  r_0 = new double[(int)(DAYSTORUN/timestep)];
  lam1 = new double[(int)(DAYSTORUN/timestep)];
  lam2 = new double[(int)(DAYSTORUN/timestep)];
  lam3 = new double[(int)(DAYSTORUN/timestep)];

}

ODEpopbased::~ODEpopbased()
{
```

```
  delete [] tempArray;
  delete [] s_0;
  delete [] q_0;
  delete [] s_1;
  delete [] kap;
  delete [] r_0;
  delete [] lam1;
  delete [] lam2;
  delete [] lam3;
}


////////////////////////////////////////////////////////////////////
// PUBLIC METHODS
////////////////////////////////////////////////////////////////////
void ODEpopbased::runSimulation() {
  for(int j = 0; j < (int)(DAYSTORUN/timestep)-1; j++) {
    s_0[j] = 0;
    q_0[j] = 0;
    s_1[j] = 0;
    kap[j] = 0;
    r_0[j] = 0;
    lam1[j] = 0;
    lam2[j] = 0;
    lam3[j] = 0;
  }
  for(double i = 0; i < DAYSTORUN; i+= timestep) {
    runSimulationStep(i);
    if(i - (int)i < timestep + .00000001)
      cout << "day "<< (int)i << endl;
  }

}



////////////////////////////////////////////////////////////////////
// PRIVATE METHODS
////////////////////////////////////////////////////////////////////

/*******************************************************************
**
**  Runs a step of the simulation, and outputs numbers in each
**  state to log files
**
**  Requires: nothing
**  Modifies: Log files associated with each ostream object.
**  Returns:  nothing
*/
void ODEpopbased::runSimulationStep(double step) {
  day += timestep;

  // Indicates whether intervention has commenced
  inter = hasIntervention();

  // Definitions of these variables are given in Kaplan's paper
  double I_3 = statePop[3] + statePop[8] + statePop[13];
  double Q = statePop[5] + statePop[6] + statePop[7] + statePop[8];
  double R_0 = beta*(statePop[0] + statePop[5] + statePop[10])/r_3;

  double lambda1;
  double lambda2;
  double lambda3;
  if(newway) {

    // Actual calculations of lambdas rather than integral
    // approximations given in Kaplan's paper.
    totstep++;
    s_0[totstep-1] = statePop[0];
    q_0[totstep-1] = statePop[5];
    s_1[totstep-1] = statePop[10];

    double rsum = 0;


    for(int j = 0; j < totstep-1; j++) {
      rsum += r_0[j];
    }
    r_0[totstep-1] = exp(-r_3*timestep)*rsum +
                     beta*(s_0[totstep-1] + q_0[totstep-1] +
                     s_1[totstep-1]);

    rsum = 0;
    for(int j = 0; j < totstep-1; j++) {
      rsum += lam1[j];
    }
    lam1[totstep-1] = exp(-r_3*timestep)*exp(kap[totstep-1])*rsum;
    lam2[totstep-1] = q(2,kap[totstep-1])*lam1[totstep-1];
    lam3[totstep-1] = q(3,kap[totstep-1])*lam1[totstep-1];

    lambda1 = q(1,kap[totstep-1])*lam1[totstep -1];
    lambda2 = lam2[totstep -1];
    lambda3 = lam3[totstep -1];
  }
  else {
    // Integral approximations

    double kappa;
    if (inter)
      kappa = (c - p*R_0)*r_3*I_3/N;
    else
      kappa = 0;

    // Probabilities determined by stochastic model (cheating)
    double q1, q2, q3;
    if(day < 5 || day >= 50) {
      q1 = q2 = q3 = 0;
    }

    else {
      q1 = .570;
      q2 = .364;
      q3 = .063;
    }


    lambda1 = q1*beta*statePop[0]/(r_3 + kappa);
    lambda2 = q2*beta*statePop[0]/(r_3 + kappa);
    lambda3 = q3*beta*statePop[0]/(r_3 + kappa);

/*
  lambda1 = q(1, kappa)*beta*statePop[0]/(r_3 + kappa);
  lambda2 = q(2, kappa)*beta*statePop[0]/(r_3 + kappa);
  lambda3 = q(3, kappa)*beta*statePop[0]/(r_3 + kappa);
*/
  }

  // Finding number to transition out of each state
  statePopTemp[0]  = updateS_0(I_3, R_0);
  statePopTemp[1]  = updateI_10(I_3, R_0, lambda1);
  statePopTemp[2]  = updateI_20(I_3, R_0, lambda2);
  statePopTemp[3]  = updateI_30(I_3, R_0, lambda3);
  statePopTemp[4]  = updateI_40();
  statePopTemp[5]  = updateQ_0(I_3, R_0, Q);
  statePopTemp[6]  = updateQ_1(I_3, R_0, lambda1, Q);
  statePopTemp[7]  = updateQ_2(I_3, R_0, lambda2, Q);
  statePopTemp[8]  = updateQ_3(I_3, R_0, lambda3, Q);
  statePopTemp[9]  = updateH(Q);
  statePopTemp[10] = updateS_1(I_3, Q);
  statePopTemp[11] = updateI_11(I_3, Q);
  statePopTemp[12] = updateI_21(Q);
  statePopTemp[13] = updateI_31(Q);
  statePopTemp[14] = updateI_41();
  statePopTemp[15] = updateZ(Q);
  statePopTemp[16] = updateD(Q);

  // Transitioning correct number of people from each state
  for(int i = 0; i < 17; i++) {
    statePop[i] += statePopTemp[i];
  }

  if(MASSVACCINATION && inter) {
    statePop[5] += statePop[0];
    statePop[6] += statePop[1];
    statePop[7] += statePop[2];
    statePop[8] += statePop[3];
    statePop[0] = 0;
    statePop[1] = 0;
    statePop[2] = 0;
    statePop[3] = 0;
    MASSVACCINATION = false;
  }
  // Printing number of people in each state, if start of new day
  if (step - int(step) < timestep) {
    pr.PrintDay(statePop);
  }
}

/*******************************************************************
**
**  Calculate correct number of people to transition each day
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Number of people to transition from each state each
**            timestep
*/
double ODEpopbased::updateS_0(double I_3, double R_0) {
  return (timestep * (-beta*I_3*statePop[0] -
                      (c-p*R_0)*statePop[0]*r_3*I_3*inter/N));
}
double ODEpopbased::updateI_10(double I_3, double R_0,
```

```
                             double lambda1) {
  return timestep * (beta*I_3*statePop[0] -
                    ((c-p*R_0)*statePop[1]/N + p*lambda1) *
                    r_3*I_3*inter - r_1*statePop[1]);
}
double ODEpopbased::updateI_20(double I_3, double R_0,
                             double lambda2) {
  return timestep * (r_1*statePop[1] - ((c-p*R_0)*statePop[2]/N +
                                     p*lambda2)*r_3*I_3*inter -
                                     r_2*statePop[2]);
}
double ODEpopbased::updateI_30(double I_3, double R_0,
                             double lambda3) {
  return timestep * (r_2*statePop[2] - ((c-p*R_0)*statePop[3]/N +
                                     p*lambda3)*r_3*I_3*inter -
                                     r_3*statePop[3]);
}
double ODEpopbased::updateI_40() {
  return timestep * (r_3 * statePop[3] - r_4 * statePop[4]);
}
double ODEpopbased::updateQ_0(double I_3, double R_0, double Q) {
  return timestep * ((c-p*R_0)*statePop[0]*r_3*I_3*inter/N -
                     beta*I_3*statePop[5] -
                     mu*statePop[5]*min(1.0, n/Q));
}
double ODEpopbased::updateQ_1(double I_3, double R_0,
                             double lambda1, double Q) {
  return timestep * (beta*I_3*statePop[5] +
                    ((c-p*R_0)*statePop[1]/N + p*lambda1) *
                    r_3*I_3*inter -
                    mu*statePop[6]*min(1.0, n/Q) -
                    r_1*statePop[6]);
}
double ODEpopbased::updateQ_2(double I_3, double R_0,
                             double lambda2, double Q) {
  return timestep * (r_1*statePop[6] + ((c-p*R_0)*statePop[2]/N +
                    p*lambda2)*r_3*I_3*inter -
                    mu*statePop[7]*min(1.0, n/Q) -
                    r_2*statePop[7]);
}
double ODEpopbased::updateQ_3(double I_3, double R_0,
                             double lambda3, double Q) {
  return timestep * (r_2*statePop[7] + ((c-p*R_0)*statePop[3]/N +
                    p*lambda3)*r_3*I_3*inter -
                    mu*statePop[8]*min(1.0, n/Q) -
                    r_3*statePop[8]);
}
double ODEpopbased::updateH(double Q) {
  return timestep * ((1-f)*h*mu*statePop[8]*min(1.0, n/Q) -
                    r_4*statePop[9]);
}
double ODEpopbased::updateS_1(double I_3, double Q) {
  return timestep * ((1-f)*(1-v_0)*mu*statePop[5]*min(1.0, n/Q) -
                    beta*statePop[10]*I_3);
}
double ODEpopbased::updateI_11(double I_3, double Q) {
  return timestep * (beta*statePop[10]*I_3 +
                    (1-f)*(1-v_1)*mu*statePop[6]*min(1.0, n/Q) -
                    r_1*statePop[11]);
}
double ODEpopbased::updateI_21(double Q) {
  return timestep * (r_1*statePop[11] +
                    (1-f)*mu*statePop[7]*min(1.0, n/Q) -
                    r_2*statePop[12]);
}
double ODEpopbased::updateI_31(double Q) {
  return timestep * (r_2*statePop[12] +
                    (1-f)*(1-h)*mu*statePop[8]*min(1.0, n/Q) -
                    r_3*statePop[13]);
}
double ODEpopbased::updateI_41() {
  return timestep*(r_3*(statePop[13] + statePop[8] + statePop[9]) -
                  r_4*statePop[14]);
}
double ODEpopbased::updateZ(double Q) {
  return timestep * ((1-f)*(v_0*statePop[5] + v_1*statePop[6]) *
                    mu*min(1.0, n/Q) +
                    (1-delta)*r_4*(statePop[4]+statePop[14]));
}
double ODEpopbased::updateD(double Q) {
  return timestep * (f*mu*Q*min(1.0, n/Q) +
                    delta*r_4*(statePop[4] + statePop[14]));
}

/*****************************************************************
**
**  Calculates minimum of two numbers
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Minimum of a and b
*/
double ODEpopbased::min(double a, double b) {
  if (a <= b)
    return a;
  else
    return b;
}

/*****************************************************************
**
**  Calculates whether intervention has commenced.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  1 if intervention has commenced, 0 otherwise
*/
double ODEpopbased::hasIntervention() {
  if (day < INTERVENTIONDAY)
    return 0;
  else
    return 1;
}


/*****************************************************************
**
**  Returns the conditional probability that a contact of an
**  index detected at time t is in stage j of disease given that
**  the contact has not been traced by time t (see Kaplan).
**
**  Requires: nothing
**  Modifies: nothing
**  Effects:  Returns a double value between 0 and 1 indicating
**            conditional probability
**/
double ODEpopbased::q(int j, double k) {
  double retval = 1;    // Probability q1 (Eq. 19 in Kaplan paper)
  double rk = 0;
  double rj = 0;
  if(j == 1)
    rj = r_1;
  else if(j == 2)
    rj = r_2;
  else if (j == 3)
    rj = r_3;
  else if (j == 4)
    rj = r_4;
  else
    cout << "Error in calculating q (rj)" << endl;

  for (int i = 1; i <= j - 1; i++) {
    if(i == 1)
      rk = r_1;
    else if (i == 2)
      rk = r_2;
    else if (i == 3)
      rk = r_3;
    else if (i == 4)
      rk = r_4;
    else
      cout << "Error in calculating q (rk)" << endl;

    retval *= rk/(rk + r_3 + k);
  }

  retval *= (r_3 + k)/(rj + r_3 + k);

  return retval;
}
```

Appendix B

Helper classes

B.1 Person

```
//////////////////////////////////////////////////////////////////////
// Person.h: interface for the Person class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
//     CLASS DESCRIPTION
//
// The person class holds the state for a single person, which
// includes an identification number, current state, number of
// steps (total, in current state, in queue), a vector of contacts,
// and various flags. The stochastic model simulates many people
// interacting and spreading smallpox.
//
//////////////////////////////////////////////////////////////////////
#if !defined(AFX_PERSON_H__INCLUDED_)
#define AFX_PERSON_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <iostream>
#include <vector>
using namespace std;

class Person
{
public:
  Person();
  virtual ~Person();

  // ACCESSOR FUNCTIONS
  int getId() const;
  int getState() const;
  int getStepsInState() const;
  int getIndex() const;
  int getTimeInQueue() const;
  int getTotalSteps() const;

  bool getAltered() const;
  bool getContactAltered() const;
  bool getQueueAltered() const;
  bool isInQueue() const;

  vector<int> getContactVector();
  vector<int> getFamilyVector();
  vector<int> getInfectedsVector();

  unsigned short getHome() const;
  unsigned short getWork() const;
  unsigned short getCurrentBldg() const;

  bool getEverInfected() const;
  int getNumInfected() const;


  // MUTATOR FUNCTIONS
  void transitionStates(int newState);
  void queueTransition(double rand1, double rand2, double f,
                       double v_0, double v_1, double h);
  void reset();

  void setId(int newId);
  void setState(int state);
  void incStepsInState();
  void setIndex(int index);
  void incTimeInQueue();
  void incTotalSteps();
  void decTotalSteps();

  void setAltered();
  void unsetAltered();
  void setContactAltered();
  void unsetContactAltered();
  void setQueueAltered();
  void unsetQueueAltered();

  bool addContact(int id, int listSize);
  void forceAddContact(int id, int listSize);
  void addToFamily(int famId);
  void addInfected(int id);

  void setHome(int i);
  void setWork(int i);
  void updateCurrentBldg(int bldg);

  void setEverInfected(bool b);
  void addNumInfected(int i);

private:
  int id;
  int currentState;
  int stepsInCurrentState;
  int myIndex;
  int timeInQueue;
  int totalSteps;

  bool isAltered;
  bool isContactAltered;
  bool isQueueAltered;

  vector<int> contacts;
  vector<int> family;
  vector<int> infecteds;

  unsigned short home;
  unsigned short work;
  unsigned short currentbldg;

  bool everInfected;
  int numInfected;
};
#endif // !defined(AFX_PERSON_H__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// Person.cpp: implementation of the Person class.
//
//////////////////////////////////////////////////////////////////////

#include "Person.h"
```

```
////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////

Person::Person()
  :  id(0),
     currentState(0),
     stepsInCurrentState(0),
     myIndex(-1),
     timeInQueue(0),
     totalSteps(0),
     isAltered(false),
     isContactAltered(false),
     isQueueAltered(false),
     home(0),
     work(0),
     currentbldg(0),
     everInfected(false),
     numInfected(0)
{
  contacts.clear();
  family.clear();
  infecteds.clear();
}

Person::~Person()
{
}

//
// ACCESSOR FUNCTIONS
//

/*******************************************************************
**
**  Returns identification number of person.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  id of person
*/
int Person::getId() const {
  return id;
}

/*******************************************************************
**
**  Returns current state of person.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  current state of person
*/
int Person::getState() const {
  return currentState;
}

/*******************************************************************
**
**  Returns number of steps spent in current state of person.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  number of steps spent in current state of person
*/
int Person::getStepsInState() const {
  return stepsInCurrentState;
}

/*******************************************************************
**
**  Returns id of person who infected this person
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  id of person who infected this, -1 if this not
**            infected
*/
int Person::getIndex() const {
  return myIndex;
}

/*******************************************************************
**
**  Returns number of steps spent in queue.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  time spent in queue
*/
int Person::getTimeInQueue() const {
  return timeInQueue;
}

/*******************************************************************
**
**  Returns total number of steps taken in simulation.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  total number of steps taken
*/
int Person::getTotalSteps() const {
  return totalSteps;
}


/*******************************************************************
**
**  Returns flag indicating whether person altered in transition.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  altered flag
*/
bool Person::getAltered() const {
  return isAltered;
}

/*******************************************************************
**
**  Returns flag indicating whether person needs to queue their
**  contacts.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  contact altered flag
*/
bool Person::getContactAltered() const {
  return isContactAltered;
}

/*******************************************************************
**
**  Returns flag indicating whether person altered into queue.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  queue altered flag
*/
bool Person::getQueueAltered() const {
  return isQueueAltered;
}

bool Person::isInQueue() const {
  return ((currentState > 4) && (currentState < 9));
}

/*******************************************************************
**
**  Returns contact vector (holding other ids) of person.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  contact vector of person
*/
vector<int> Person::getContactVector() {
  return contacts;
}

/*******************************************************************
**
**  Returns family vector (holding other ids) of person.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  vector of family member ids of person
*/
vector<int> Person::getFamilyVector() {
  return family;
}

/*******************************************************************
**
**  Returns infected vector (holding other ids) of person.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  vector of people this person infected
*/
vector<int> Person::getInfectedsVector() {
  return infecteds;
}
```

```
/******************************************************************
**
** Returns the number of the building this lives in.
**
** Requires: nothing
** Modifies: nothing
** Returns:  Id of home of this person.
*/
unsigned short Person::getHome() const {
  return home;
}

/******************************************************************
**
** Returns the number of the building this works in.
**
** Requires: nothing
** Modifies: nothing
** Returns:  Id of workplace of this person.  Id of school/college
**           for students.
*/
unsigned short Person::getWork() const {
  return work;
}

/******************************************************************
**
** Returns the number of the building this person is currently in.
**
** Requires: nothing
** Modifies: nothing
** Returns:  Id of current building of this person.
*/
unsigned short Person::getCurrentBldg() const {
  return currentbldg;
}


/******************************************************************
**
** Returns true iff this was ever infectious.
**
** Requires: nothing
** Modifies: nothing
** Returns:  True iff this person was ever capable of spreading
**           disease.
*/
bool Person::getEverInfected() const {
  return everInfected;
}

/******************************************************************
**
** Returns the number of people this person infected during a
** simulation.
**
** Requires: nothing
** Modifies: nothing
** Returns:  Total number of people this person infected.
*/
int Person::getNumInfected() const {
  return numInfected;
}


//
// MUTATOR FUNCTIONS
//

/******************************************************************
**
**  Transitions person to new state.
**
**  Requires: 0 <= newState <= 16
**  Modifies: currentState, stepsInCurrentState
**  Returns:  nothing
*/
void Person::transitionStates(int newState) {
  currentState = newState;
  stepsInCurrentState = 0;
}


/******************************************************************
**
**  Performs a transition of someone in the queue who has been
**  popped out after receiving a vaccination shot.
**
**  Requires: nothing
**  Modifies: this
**  Returns:  nothing
*/
void Person::queueTransition(double rand1, double rand2, double f,
                             double v_0, double v_1, double h) {
  if(currentState == 5) {
    // Person was queued and susceptible, can either be traced and
    // susceptible, immune, or dead from vaccine problems.

    if(rand1 <= f) {
      // Person died from vaccine-related complications
      transitionStates(16);
      setAltered();
    }
    else {
      if(rand2 < v_0) {
// Person is immune
transitionStates(15);
setAltered();
      }
      else {
// Person is traced and susceptible
transitionStates(10);
setAltered();
      }
    }
  }

  else if (currentState == 6) {
    // Person was queued and stage 1 infectious, can either be
    // traced and infectious, immune, or dead from vaccine problems.

    if(rand1 <= f) {
      // Person died from vaccine-related complications
      transitionStates(16);
      setAltered();
    }
    else {
      if(rand2 < v_1) {
// Person is immune
transitionStates(15);
setAltered();
      }
      else {
// Person is traced and stage 1 infectious
setState(11);
setAltered();
      }
    }
  }

  else if (currentState == 7) {
    // Person was queued and stage 2 infectious, can either be
    // traced and infectious or dead from vaccine problems, as
    // vaccine will no longer provide immunity.
    if(rand1 <= f) {
      // Person died from vaccine-related complications
      transitionStates(16);
      setAltered();
    }
    else {
      // Person is traced and stage 2 infectious
      setState(12);
      setAltered();
    }
  }


  else if (currentState == 8) {
    // Person was queued and stage 3 infectious, can either be
    // traced and infectious, quarantined, or dead from vaccine
    // problems.
    if(rand1 <= f) {
      // Person died from vaccine-related complications
      transitionStates(16);
      setAltered();
    }
    else {
      if(rand2 < h) {
// Person has been quarantined
transitionStates(9);
setAltered();
      }
      else {
// Person is traced and stage 3 infectious
setState(13);
setAltered();
      }
    }
  }

  else {
```

```
    cout << currentState << endl;
  }
}

/*****************************************************************
**
**  Resets all values except the id of this to default values, to
**  run multiple simulations.
**
**  Requires: nothing
**  Modifies: this
**  Returns:  nothing
*/
void Person::reset() {
  currentState = 0;
  stepsInCurrentState = 0;
  myIndex = -1;
  timeInQueue = 0;
  totalSteps = 0;

  isAltered = false;
  isContactAltered = false;
  isQueueAltered = false;

  contacts.clear();
  family.clear();
  infecteds.clear();

  home = 0;
  work = 0;
  currentbldg = 0;

  everInfected = false;
  numInfected = 0;
}


/*****************************************************************
**
**  Sets identification number of this
**
**  Requires: newId unique in simulation, newId != -1
**  Modifies: id
**  Returns:  nothing
*/
void Person::setId(int newId) {
  id = newId;
}

/*****************************************************************
**
**  Sets current state of person to state.
**
**  Requires: 0 <= state <= 16
**  Modifies: currentState
**  Returns:  nothing
*/
void Person::setState(int state) {
  currentState = state;
}

/*****************************************************************
**
**  Increments steps taken in current state.
**
**  Requires: nothing
**  Modifies: stepsInCurrentState
**  Returns:  nothing
*/
void Person::incStepsInState() {
  stepsInCurrentState++;
}

/*****************************************************************
**
**  Sets index of this (id of person who infected this).
**
**  Requires: index is the id of a person in the simulation
**  Modifies: myIndex
**  Returns:  nothing
*/
void Person::setIndex(int index) {
  myIndex = index;
}

/*****************************************************************
**
**  Increments number of steps spent in the vaccination queue.
**
**  Requires: nothing
**  Modifies: timeInQueue
**  Returns:  nothing
*/
void Person::incTimeInQueue() {
  timeInQueue++;
}

/*****************************************************************
**
**  Increments total number of steps taken in simulation.
**
**  Requires: nothing
**  Modifies: totalSteps
**  Returns:  nothing
*/
void Person::incTotalSteps() {
  totalSteps++;
}

/*****************************************************************
**
**  Decrements total number of steps taken in simulation.  Needed
**  when person transitions naturally, but in the same timestep
**  another person places them into the queue for vaccination.
**
**  Requires: nothing
**  Modifies: totalSteps
**  Returns:  nothing
*/
void Person::decTotalSteps() {
  totalSteps--;
}


/*****************************************************************
**
**  Sets the altered flag to true.
**
**  Requires: nothing
**  Modifies: isAltered
**  Returns:  nothing
*/
void Person::setAltered() {
  isAltered = true;
}

/*****************************************************************
**
**  Sets the altered flag to false.
**
**  Requires: nothing
**  Modifies: isAltered
**  Returns:  nothing
*/
void Person::unsetAltered() {
  isAltered = false;
}

/*****************************************************************
**
**  Sets the contact altered flag to true.
**
**  Requires: nothing
**  Modifies: isContactAltered
**  Returns:  nothing
*/
void Person::setContactAltered() {
  isContactAltered = true;
}

/*****************************************************************
**
**  Sets the contact altered flag to false.
**
**  Requires: nothing
**  Modifies: isContactAltered
**  Returns:  nothing
*/
void Person::unsetContactAltered() {
  isContactAltered = false;
}

/*****************************************************************
**
**  Sets the queue altered flag to true.
**
**  Requires: nothing
**  Modifies: isQueueAltered
**  Returns:  nothing
*/
void Person::setQueueAltered() {
  isQueueAltered = true;
}
```

```
/*****************************************************************
**
**  Sets the queue altered flag to false.
**
**  Requires: nothing
**  Modifies: isQueueAltered
**  Returns:  nothing
*/
void Person::unsetQueueAltered() {
  isQueueAltered = false;
}


/*****************************************************************
**
**  If contact vector is not full, adds contact to contact vector.
**
**  Requires: id is the id of a person in the simulation
**  Modifies: contacts
**  Returns:  true if contact is added, false if vector is full
*/
bool Person::addContact(int id, int listSize) {
  if(contacts.size() < listSize) {
    for(int i = 0; i < contacts.size(); i++) {
      if(id == contacts.at(i))
return false;
    }
    contacts.push_back(id);
    return true;
  }
  else {
    return false;
  }
}

/*****************************************************************
**
**  Adds contact by removing oldest contact of person if contact
**  list is full, otherwise simply adds contact.
**
**  Requires: id is the id of a person in the simulation
**  Modifies: contacts
**  Returns:  nothing
*/
void Person::forceAddContact(int id, int listSize) {
  if(contacts.size() < listSize) {
    contacts.push_back(id);
  }
  else {
    vector<int>::iterator it = contacts.begin();
    contacts.erase(it, it+1);
    contacts.push_back(id);
  }
}

/*****************************************************************
**
**  Adds contact to family of this person.
**
**  Requires: id is the id of a person in the simulation
**  Modifies: family
**  Returns:  nothing
*/
void Person::addToFamily(int famId) {
  if(famId != id)
    family.push_back(id);
}

/*****************************************************************
**
**  Adds contact that this person infected.
**
**  Requires: id is the id of a person in the simulation
**  Modifies: infecteds
**  Returns:  nothing
*/
void Person::addInfected(int id) {
  infecteds.push_back(id);
}

/*****************************************************************
**
**  Sets the home of this person to i.
**
**  Requires: i is the number of a residence in the simulation
**  Modifies: home
**  Returns:  nothing
*/
void Person::setHome(int i) {
  home = i;
}

/*****************************************************************
**
**  Sets the workplace of this person to i.
**
**  Requires: i is the number of a work building in the simulation
**  Modifies: work
**  Returns:  nothing
*/
void Person::setWork(int i) {
  work = i;
}

/*****************************************************************
**
**  Sets the current building of this person to i.
**
**  Requires: i is the number of a building in the simulation
**  Modifies: currentbldg
**  Returns:  nothing
*/
void Person::updateCurrentBldg(int bldg) {
  currentbldg = bldg;
}


/*****************************************************************
**
**  Sets the everInfected flag to b.
**
**  Requires: nothing
**  Modifies: everInfected
**  Returns:  nothing
*/
void Person::setEverInfected(bool b) {
  everInfected = b;
}

/*****************************************************************
**
**  Adds i to the number of people this person has infected.
**
**  Requires: nothing
**  Modifies: numInfected
**  Returns:  nothing
*/
void Person::addNumInfected(int i) {
  numInfected += i;
}
```

B.2 Building

```
//////////////////////////////////////////////////////////////////////
// Building.h: interface for the Building class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
//                           CLASS DESCRIPTION
//
// The Building class is a helper class for the agent-based model.
// Buildings have various characteristics, such as the maximum
// occupancy and the people currently in the building.  People can
// enter the building if the maximum occupancy is not exceeded and
// may leave the building at any time.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_BUILDING_H__INCLUDED_)
#define AFX_BUILDING_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <iostream>
#include <string>
#include <vector>
#include "Person.h"
using namespace std;

class Building
{
public:
  Building();
  virtual ~Building();

  // ACCESSOR FUNCTIONS
  int getID();
  int getOccupancy();
  bool canAssign();
  unsigned short getNumAssigned();
  int getOccupantNum();
  int getOccupant(int place);
  int getLastOccupant();
  vector<int> getOccupantVector();
  bool containsID(int id);

  // MUTATOR FUNCTIONS
  void setID(int id);
  void setOccupancy(int o);
  void incNumAssigned();
  bool addOccupant(int id);
  void addHospitalOccupant(int id);
  bool insertHospitalOccupant(int id, int place);
  bool removeOccupantID(int id);
  void removeLastOccupant();
  void removeAllOccupants();

private:
  int ID;
  int OCCUPANCY;
  unsigned short numAssigned;
  vector<int> occupants;

};

#endif // !defined(AFX_BUILDING_H__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// Building.cpp: implementation of the Building class.
//
//////////////////////////////////////////////////////////////////////

#include "Building.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

Building::Building()
  :   ID(-1),
      OCCUPANCY(0),
      numAssigned(0)
{
  occupants.clear();
}

Building::~Building()
{
}

/*************************************************************
**
**  Returns the Id number of this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The id of this building.
*/
int Building::getID() {
  return ID;
}

/*************************************************************
**
**  Returns the occupancy of this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The occupancy of this building.
*/
int Building::getOccupancy() {
  return OCCUPANCY;
}

/*************************************************************
**
**  Returns whether more people can be assigned to this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns.  True iff the number assigned is smaller than the
**            building occupancy.
*/
bool Building::canAssign() {
  return (numAssigned < OCCUPANCY);
}

/*************************************************************
**
**  Returns the number of people assigned to this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The number of people assigned to this building.
*/
unsigned short Building::getNumAssigned() {
  return numAssigned;
}

/*************************************************************
**
**  Returns the number of people currently inside this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The number of people currently inside this building.
*/
int Building::getOccupantNum() {
  return occupants.size();
}

/*************************************************************
**
**  Returns the id of the person in place of this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The id of a person inside this building.
*/
int Building::getOccupant(int place) {
  return occupants[place];
}

/*************************************************************
**
**  Returns the id of the last person in this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The id of a person inside this building.
*/
int Building::getLastOccupant() {
  return occupants.back();
}
```

```
/*************************************************************
**
**  Returns the entire vector of ids of the people in this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  The vector of ids of the people in this building.
*/
vector<int> Building::getOccupantVector() {
  return occupants;
}


/*************************************************************
**
**  Returns true if person with ID# id is inside this building.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  True iff occupants contains id.
*/
bool Building::containsID(int id) {
for(int i = 0; i < occupants.size(); i++) {
if(occupants.at(i) == id)
return true;
}
return false;
}


//
// MUTATOR FUNCTIONS
//

/*************************************************************
**
**  Sets the identification number of the building to id.
**
**  Requires: nothing
**  Modifies: ID
**  Returns:  nothing
*/
void Building::setID(int id) {
  ID = id;
}

/*************************************************************
**
**  Sets the maximum occupancy of the building to o.
**
**  Requires: nothing
**  Modifies: OCCUPANCY
**  Returns:  nothing
*/
void Building::setOccupancy(int o) {
  OCCUPANCY = o;
}

/*************************************************************
**
**  Increases the number of people assigned to this building by one.
**
**  Requires: nothing
**  Modifies: numAssigned
**  Returns:  nothing
*/
void Building::incNumAssigned() {
  numAssigned++;
}

/*************************************************************
**
**  Adds person with ID# id to this building if maximum occupancy
**  will not be exceeded.
**
**  Requires: nothing
**  Modifies: occupants
**  Returns:  True iff there is room to add another person to the
**            building.
*/
bool Building::addOccupant(int id) {
  if(occupants.size() < OCCUPANCY) {
    occupants.push_back(id);
    return true;
  }
  else return false;
}

/*************************************************************
**
**  Adds person to hospital.  Does not check for size constraints.
**
**  Requires: nothing
**  Modifies: occupants
**  Returns:  nothing
*/
void Building::addHospitalOccupant(int id) {
  vector<int>::iterator it = occupants.begin();
  occupants.insert(it, id);
}

/*************************************************************
**
**  Adds person to hospital in particular spot place.
**
**  Requires: nothing
**  Modifies: occupants
**  Returns:  nothing
*/
bool Building::insertHospitalOccupant(int id, int place) {
  vector<int>::iterator it = occupants.begin();
  if(place > occupants.size())
    return false;
  for(int i = 0; i < place; i++) {
    it++;
  }
  occupants.insert(it, id);
  return true;
}

/*************************************************************
**
**  Removes id from this building if id is present.
**
**  Requires: nothing
**  Modifies: occupants
**  Returns:  True iff id is a member of occupants.
*/
bool Building::removeOccupantID(int id) {
  for(int i = 0; i < occupants.size(); i++) {
    if((occupants.at(i)) == id) {
      occupants.erase(occupants.begin()+i, occupants.begin()+i+1);
      return true;
    }
  }
  return false;
}

/*************************************************************
**
**  Removes last person in building.
**
**  Requires: nothing
**  Modifies: occupants
**  Returns:  nothing
*/
void Building::removeLastOccupant() {
  occupants.pop_back();
}

/*************************************************************
**
**  Removes all people from building.
**
**  Requires: nothing
**  Modifies: occupants
**  Returns:  nothing
*/
void Building::removeAllOccupants() {
  occupants.clear();
  numAssigned = 0;
}
```

B.3 Random number generator

```
//////////////////////////////////////////////////////////////////////
// RandNum.h: interface for the RandNum class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// CLASS DESCRIPTION
//
// This class provides pseudorandom numbers distributed uniformly
// between 0 and 1.  It is generally seeded with the time the
// computer has been running, as found in the time.h class.
//
//////////////////////////////////////////////////////////////////////
#if !defined(AFX_RANDNUM_H__INCLUDED_)
#define AFX_RANDNUM_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class RandNum
{
public:
RandNum();
virtual ~RandNum();
double getNext(long *idum);
int nextThree(long *idum);
};

#endif // !defined(AFX_RANDNUM_H__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// RandNum.cpp: implementation of the RandNum class.
//
//////////////////////////////////////////////////////////////////////

#include "RandNum.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

RandNum::RandNum()
{
}

RandNum::~RandNum()
{
}

/*******************************************************************
**
**  Creates pseudo-random numbers for use in calculating transition
**  probabilities.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Pseudo-random number for use in modeling random
**            processes.  Taken from numerical recipes.
*/
double RandNum::getNext(long *idum) {
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

  int j;
  long k;
  static long iy=0;
  static long iv[NTAB];
  double temp;
  double retval;

  if (*idum <= 0 || !iy) {
    if (-(*idum) < 1) *idum=1;
    else *idum = -(*idum);
    for (j=NTAB+7;j>=0;j--) {
      k=(*idum)/IQ;
      *idum=IA*(*idum-k*IQ)-IR*k;
      if (*idum < 0) *idum += IM;
      if (j < NTAB) iv[j] = *idum;
    }
    iy=iv[0];
  }
  k=(*idum)/IQ;
  *idum=IA*(*idum-k*IQ)-IR*k;
  if (*idum < 0) *idum += IM;
  j=iy/NDIV;
  iy=iv[j];
  iv[j] = *idum;
  if ((temp=AM*iy) > RNMX) retval = RNMX;
  else retval = temp;
#undef IA
#undef IM
#undef AM
#undef IQ
#undef IR
#undef NTAB
#undef NDIV
#undef EPS
#undef RNMX

  return retval;
}

/*******************************************************************
**
**  Creates pseudo-random number between 0-2 inclusive.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Pseudo-random number for use in modeling random
**            processes.  Taken from numerical recipes.
*/
int RandNum::nextThree(long * idum) {
  double rand = getNext(idum);
  if(rand < .33333)
    return 0;
  else if (rand < .66666)
    return 1;
  else return 2;
}
```

B.4 Transition probabilities

```
////////////////////////////////////////////////////////////////////
// Probability.h: interface for the Probability class.
//
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
//    CLASS DESCRIPTION
//
// The Probability class allows Gaussian transitions to occur
// between states.  For Gaussian transitions, both the number of
// steps in the current state and the value of the current state
// matter in determining whether a person should transition
// naturally during a given timestep.  Thus the Probability class
// only has one function, getProb, which does just that.  There is
// also an exponential transition probability built in.
//
////////////////////////////////////////////////////////////////////
#if !defined(AFX_PROBABILITY_H__INCLUDED_)
#define AFX_PROBABILITY_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <iostream>
#include <string>
#include <fstream>
using namespace std;

class Probability
{
public:
  Probability();
  virtual ~Probability();

  double getProb(int state, int stepsInState, bool Gaussian);

private:
  string erftable;

  // The different transition probabilities.  An array titled:
  // meanXsdYpZ corresponds to transition probabilities
  // for a Gaussian random variable with mean X, and standard
  // deviation Y.Z.
  double mean5sd1[385];
  double mean7sd1p5[553];
  double mean3sd0p5[217];
  double mean16sd2[577];

};

#endif // !defined(AFX_PROBABILITY_H__INCLUDED_)


////////////////////////////////////////////////////////////////////
// Probability.cpp: implementation of the Probability class.
//
////////////////////////////////////////////////////////////////////

#include "Probability.h"
#include <cmath>
using namespace std;

////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////

Probability::Probability()
  : erftable()
{
  ifstream erf1, erf2, erf3, erf4;
  string et1, et2, et3, et4;
  et1 = "/home/cory/popbased/probs/mean5sd1.txt";
  et2 = "/home/cory/popbased/probs/mean7sd1p5.txt";
  et3 = "/home/cory/popbased/probs/mean3sd0p5.txt";
  et4 = "/home/cory/popbased/probs/mean16sd2.txt";

  erf1.open(et1.c_str());
  erf2.open(et2.c_str());
  erf3.open(et3.c_str());
  erf4.open(et4.c_str());
  if (erf1.fail()) {
    cout << "open1 failed" << endl;
  }
  if (erf2.fail()) {
    cout << "open2 failed" << endl;
  }
  if (erf3.fail()) {
    cout << "open3 failed" << endl;
  }
  if (erf4.fail()) {
    cout << "open4 failed" << endl;
  }

  // Garbage variable for mean 16, standard dev 2, since
  // probability of transitioning before day 10 is extremely tiny.
  double retval = 0;

  // Each transition probability array stores the value 1 at the end
  // (after 3 standard deviations from the mean) since people should
  // not be stuck in the same stage forever.  This ensures
  // transition if a person has not done so already.
  for(int i = 0; i < 384; i++)
    erf1>>mean5sd1[i];
  mean5sd1[384] = 1;

  for(int i = 0; i < 552; i++)
    erf2>>mean7sd1p5[i];
  mean7sd1p5[552] = 1;

  for(int i = 0; i < 216; i++)
    erf3>>mean3sd0p5[i];
  mean3sd0p5[216] = 1;

  for(int i = 0; i < 479; i++)
    erf4>>retval;
  for(int i = 0; i < 577; i++)
    erf4>>mean16sd2[i];
  mean16sd2[577] = 1;

  erf1.close();
  erf2.close();
  erf3.close();
  erf4.close();

}

Probability::~Probability()
{
}


/*****************************************************************
**
**  Returns probability of transitioning state given current state
**  and time spent there.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  Gaussian or exponential probability
*/
double Probability::getProb(int state, int stepsInState,
                            bool Gaussian) {

  // TESTING PROBABILITY FOR EMULATION WITH DIFFERENTIAL EQUATION
  // MODEL
  if(!Gaussian) {
    if(state == 0 || state == 5 || state == 10 || state == 15 ||
       state == 16) {
      return -1;
    }
    else if(state == 1 || state == 6 || state == 11) {
      return 1-exp(-1.0/(5*48));
    }
    else if(state == 2 || state == 7 || state == 12) {
      return 1-exp(-1.0/(7*48));
    }
    else if(state == 3 || state == 8 || state == 13) {
      return 1-exp(-1.0/(3*48));
    }
    else if(state == 4 || state == 14 || state == 9) {
      return 1-exp(-1.0/(16*48));
    }
    else return -1;
  }

  else {
    // GAUSSIAN PROBABILITIES, gathered using erf function
    if(state == 0 || state == 5 || state == 10 || state == 15 ||
       state == 16) {
      cout << "Not a valid state for Gaussian transition" << endl;
      return -1;
    }
    else if(state == 1 || state == 6 || state == 11) {
      if(stepsInState > 384) {
return 1;
```

```
      }
      return mean5sd1[stepsInState];
    }

    else if(state == 2 || state == 7 || state == 12) {
      if(stepsInState > 552)
return 1;
      return mean7sd1p5[stepsInState];
    }
    else if(state == 3 || state == 8 || state == 13) {
      if(stepsInState > 216)
return 1;
      return mean3sd0p5[stepsInState];
    }
    else if(state == 4 || state == 14 || state == 9) {
      if(stepsInState < 480)
return 0;
      else if(stepsInState >= 1057)
return 1;
      else
return mean16sd2[stepsInState-480];
    }
    else {
      cout << "State is invalid." << endl;
      return -1;
    }
  }
}
```

B.5 Output averager

```
//////////////////////////////////////////////////////////////////////
// Averager.h: interface for the Averager class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
//      CLASS DESCRIPTION
//
// This class allows the stochastic models to be run multiple
// times, in order to average their outputs to determine the mean
// behavior of the models.  Since the stochastic model only keeps
// track of its populations in the current timestep of the current
// run, this class holds all timesteps of each run.
//
//////////////////////////////////////////////////////////////////////
#if !defined(AFX_AVERAGER_H__INCLUDED_)
#define AFX_AVERAGER_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <iostream>
using namespace std;

class Averager
{
public:
  Averager(int numDays);
  virtual ~Averager();

  // Adds each state of timestep i to its corresponding array
  void addToAvg(int i, double * ar);

  // Selects particular timestep i and puts state values into
  // dayOfStuff array
  void changeDayOfStuff(int i);

  // Returns the dayOfStuff array
  double *returnDOS();

private:
  double * ususc;
  double * uinf1;
  double * uinf2;
  double * uinf3;
  double * uinf4;
  double * q0;
  double * q1;
  double * q2;
  double * q3;
  double * quar;
  double * tsusc;
  double * tinf1;
  double * tinf2;
  double * tinf3;
  double * tinf4;
  double * z;
  double * d;
  double * dayOfStuff;
};

#endif // !defined(AFX_AVERAGER_H__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// Averager.cpp: implementation of the Averager class.
//
//////////////////////////////////////////////////////////////////////

#include "Averager.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

Averager::Averager(int numDays)
{
  ususc = new double[numDays];
  uinf1 = new double[numDays];
  uinf2 = new double[numDays];
  uinf3 = new double[numDays];
  uinf4 = new double[numDays];
  q0 = new double[numDays];
  q1 = new double[numDays];
  q2 = new double[numDays];
  q3 = new double[numDays];
  quar = new double[numDays];
  tsusc = new double[numDays];
  tinf1 = new double[numDays];
  tinf2 = new double[numDays];
  tinf3 = new double[numDays];
  tinf4 = new double[numDays];
  z = new double[numDays];
  d = new double[numDays];
  dayOfStuff = new double[17];

  for(int i = 0; i < 17; i++)
    dayOfStuff[i] = 0;

  for(int i = 0; i < numDays; i++) {
    ususc[i] = 0;
    uinf1[i] = 0;
    uinf2[i] = 0;
    uinf3[i] = 0;
    uinf4[i] = 0;
    q0[i] = 0;
    q1[i] = 0;
    q2[i] = 0;
    q3[i] = 0;
    quar[i] = 0;
    tsusc[i] = 0;
    tinf1[i] = 0;
    tinf2[i] = 0;
    tinf3[i] = 0;
    tinf4[i] = 0;
    z[i] = 0;
    d[i] = 0;
  }
}

Averager::~Averager()
{
  delete [] ususc;
  delete [] uinf1;
  delete [] uinf2;
  delete [] uinf3;
  delete [] uinf4;
  delete [] q0;
  delete [] q1;
  delete [] q2;
  delete [] q3;
  delete [] quar;
  delete [] tsusc;
  delete [] tinf1;
  delete [] tinf2;
  delete [] tinf3;
  delete [] tinf4;
  delete [] z;
  delete [] d;
  delete [] dayOfStuff;
}

/*****************************************************************
**
**  Adds values from array ar to arrays holding each state in each
**  timestep.
**
**  Requires: 0 <= i <= numDays, length of ar = 17
**  Modifies: All arrays holding each state
**  Returns:  nothing
*/
void Averager::addToAvg(int i, double * ar) {
  ususc[i] += ar[0];
  uinf1[i] += ar[1];
  uinf2[i] += ar[2];
  uinf3[i] += ar[3];
  uinf4[i] += ar[4];
  q0[i] += ar[5];
  q1[i] += ar[6];
  q2[i] += ar[7];
  q3[i] += ar[8];
  quar[i] += ar[9];
  tsusc[i] += ar[10];
  tinf1[i] += ar[11];
  tinf2[i] += ar[12];
  tinf3[i] += ar[13];
  tinf4[i] += ar[14];
  z[i] += ar[15];
  d[i] += ar[16];
}

/*****************************************************************
**
**  Changes dayOfStuff array to hold the number of people in each
**  state to day i
```

```
**
**  Requires: 0 <= i <= numDays
**  Modifies: dayOfStuff
**  Returns:  nothing
*/
void Averager::changeDayOfStuff(int i) {
  dayOfStuff[0] = ususc[i];
  dayOfStuff[1] = uinf1[i];
  dayOfStuff[2] = uinf2[i];
  dayOfStuff[3] = uinf3[i];
  dayOfStuff[4] = uinf4[i];
  dayOfStuff[5] = q0[i];
  dayOfStuff[6] = q1[i];
  dayOfStuff[7] = q2[i];
  dayOfStuff[8] = q3[i];
  dayOfStuff[9] = quar[i];
  dayOfStuff[10] = tsusc[i];
  dayOfStuff[11] = tinf1[i];
  dayOfStuff[12] = tinf2[i];
  dayOfStuff[13] = tinf3[i];
  dayOfStuff[14] = tinf4[i];
  dayOfStuff[15] = z[i];
  dayOfStuff[16] = d[i];
}

/******************************************************************
**
**  Returns array of numbers of people in each state for the day
**  that dayOfStuff array currently holds.
**
**  Requires: nothing
**  Modifies: nothing
**  Returns:  dayOfStuff [array of length 17]
*/
double * Averager::returnDOS() {
  return dayOfStuff;
}
```

B.6 Log file printer

```
///////////////////////////////////////////////////////////////////
// Printer.h: interface for the Printer class.
//
///////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////
//        CLASS DESCRIPTION
//
// This class is used to print output to log files.  The call
// PrintDay takes in an array of the day's values for each state,
// and prints them out to the appropriate states.
//
///////////////////////////////////////////////////////////////////


#if !defined(AFX_PRINTER_H__INCLUDED_)
#define AFX_PRINTER_H__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <iostream>
#include <string>
#include <fstream>
using namespace std;

class Printer
{
public:
  Printer();
  virtual ~Printer();
  void PrintDay(double * statePop);

private:

  void Print(double dnum, ostream& output);
  string f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13;
  string f14, f15, f16, f17;
  ofstream S_0out, I_10out, I_20out, I_30out, I_40out;
  ofstream Q_0out, Q_1out, Q_2out, Q_3out, Hout;
  ofstream S_1out, I_11out, I_21out, I_31out, I_41out, Zout, Dout;
};

#endif // !defined(AFX_PRINTER_H__INCLUDED_)


///////////////////////////////////////////////////////////////////
// Printer.cpp: implementation of the Printer class.
//
///////////////////////////////////////////////////////////////////

#include "Printer.h"

///////////////////////////////////////////////////////////////////
// Construction/Destruction
///////////////////////////////////////////////////////////////////

Printer::Printer()
  :
  f1("S_0d.txt"),
  f2("I_10d.txt"),
  f3("I_20d.txt"),
  f4("I_30d.txt"),
  f5("I_40d.txt"),
  f6("Q_0d.txt"),
  f7("Q_1d.txt"),
  f8("Q_2d.txt"),
  f9("Q_3d.txt"),
  f10("Hd.txt"),
  f11("S_1d.txt"),
  f12("I_11d.txt"),
  f13("I_21d.txt"),
  f14("I_31d.txt"),
  f15("I_41d.txt"),
  f16("Zd.txt"),
  f17("Dd.txt"),
  S_0out(f1.c_str()),
  I_10out(f2.c_str()),
  I_20out(f3.c_str()),
  I_30out(f4.c_str()),
  I_40out(f5.c_str()),
  Q_0out(f6.c_str()),
  Q_1out(f7.c_str()),
  Q_2out(f8.c_str()),
  Q_3out(f9.c_str()),
  Hout(f10.c_str()),
  S_1out(f11.c_str()),
  I_11out(f12.c_str()),
  I_21out(f13.c_str()),
  I_31out(f14.c_str()),
  I_41out(f15.c_str()),
  Zout(f16.c_str()),
  Dout(f17.c_str())
{
}

Printer::~Printer()
{
  S_0out.close();
  I_10out.close();
  I_20out.close();
  I_30out.close();
  I_40out.close();
  Q_0out.close();
  Q_1out.close();
  Q_2out.close();
  Q_3out.close();
  Hout.close();
  S_1out.close();
  I_11out.close();
  I_21out.close();
  I_31out.close();
  I_41out.close();
  Zout.close();
  Dout.close();
}

/*****************************************************************
**
**  Prints output of statePop to proper log file.
**
**  Requires: statePop be array with length 17
**  Modifies: Log files associated with each ostream object.
**  Returns:  nothing
*/
void Printer::PrintDay(double * statePop) {
  Print(statePop[0], S_0out);
  Print(statePop[1], I_10out);
  Print(statePop[2], I_20out);
  Print(statePop[3], I_30out);
  Print(statePop[4], I_40out);
  Print(statePop[5], Q_0out);
  Print(statePop[6], Q_1out);
  Print(statePop[7], Q_2out);
  Print(statePop[8], Q_3out);
  Print(statePop[9], Hout);
  Print(statePop[10], S_1out);
  Print(statePop[11], I_11out);
  Print(statePop[12], I_21out);
  Print(statePop[13], I_31out);
  Print(statePop[14], I_41out);
  Print(statePop[15], Zout);
  Print(statePop[16], Dout);
}

/*****************************************************************
**
**  Prints dnum to log file output.
**
**  Requires: nothing
**  Modifies: output
**  Returns:  nothing
*/
void Printer::Print(double dnum, ostream& output) {
  output << dnum << endl;
}
```

Bibliography

[1] *http://mathworld.wolfram.com/Erf.html.*

[2] *http://mathworld.wolfram.com/EulerForwardMethod.html.*

[3] *http://mathworld.wolfram.com/NormalDistribution.html.*

[4] *http://mathworld.wolfram.com/Runge-KuttaMethod.html.*

[5] *http://www.bt.cdc.gov/agent/anthrax/faq/index.asp.*

[6] *http://www.bt.cdc.gov/agent/smallpox/overview/disease-facts.asp.*

[7] *http://www.bt.cdc.gov/agent/smallpox/response-plan/files/guide-c-part-2.pdf.*

[8] *http://www.bt.cdc.gov/agent/smallpox/training/overview/ppt/isolationquarantine.ppt.*

[9] *http://www.ci.cambridge.ma.us/˜CDD/data/demo/2000_sf3profile.pdf.*

[10] *http://www.science.uwaterloo.ca/course_notes/science/sci255/sci255lecture7.ppt.*

[11] *Personal meeting with Ronald Hoffeld.*

[12] US sounds alarm over smallpox weapon threat. *Nature*, 399:628, 1999.

[13] A. L. Barabási and E. Bonabeau. Scale-free networks. *Scientific American*, May 2003.

[14] C. Castillo-Chavez. *Mathematical Approaches for Emerging and Reemerging Infectious Diseases.* Springer-Verlag New York, Inc., 2002.

[15] G. Chowell and C. Castillo-Chavez. *Bioterrorism: Mathematical Modeling Applications in Homeland Security.* Society for Industrial and Applied Mathematics, 2003.

[16] G. Chowell, J. M. Hyman, S. Eubank, and C. Castillo-Chavez. Analysis of a Real-World Network: The City of Portland. *Los Alamos Unclassified Report LA-UR-02-6658*, 2002.

[17] J. R. Clymer. *Systems Analysis Using Simulation and Markov Models.* Prentice-Hall, Inc., 1990.

[18] D. J. Daley and J. M. Gani. *Epidemic Modelling: An Introduction.* Cambridge University Press: New York, NY, 1999.

[19] J. L. Devore. *Probability and Statistics for Engineering and the Sciences.* Duxbury, 2000.

[20] L. M. Wein et. al. Emergency response to an anthrax attack [supporting text]. In *Proc. Natl. Acad. Sci.*, 2003.

[21] N. M. Ferguson et. al. Planning for smallpox outbreaks. *Nature*, 425:681–685, october 2003.

[22] F. Fenner, D. A. Henderson, I. Arita, Z. Jezek, and I. D. Ladnyi. Smallpox and its eradication. *World Health Organization*, 1988.

[23] R. Gani and S. Leach. Transmission potential of smallpox in contemporary populations. *Nature*, 414:748–751, December 2001.

[24] B. S. Gottfried. *Elements of Stochastic Process Simulation.* Prentice-Hall, Inc., 1984.

[25] M. E. Halloran, I. M. Longini Jr., A. Nizam, and Y. Yang. Containing bioterrorist smallpox. *Science*, 298:1428–1432, November 2002.

[26] D. A. Henderson. The looming threat of bioterrorism. *Science*, 283:1279–1282, February 1999.

[27] W. W. Hines, D. C. Montgomery, D. M. Goldsman, and C. M. Borror. *Probability and Statistics in Engineering.* John Wiley & Sons, Inc., 2003.

[28] E. H. Kaplan. *Math. Biosci.*, 105:97–109, 1991.

[29] E. H. Kaplan, D. L. Craft, and L. M. Wein. Emergency response to a smallpox attack: The case for mass vaccination. In *Proc. Natl. Acad. Sci.*, pages 10935–10940, 2002.

[30] J. Koopman. Controlling smallpox. *Science*, 298:1342–1344, November 2002.

[31] M. J. Sienko and R. A. Plane. *Chemistry.* McGraw-Hill Book Company, 1976.

[32] D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, June 1998.