A STUDY OF SIMPLE NON-EQUILIBRIUM STATISTICAL PHYSICS MODELS: MOLECULAR MOTORS AND TRAFFIC JAMS

An Honors Thesis

by

JOSHUA ALEJANDRO GONZALEZ SIEGAL

Submitted to Washington and Lee University in fulfillment
of the requirements for Honors consideration for the degree of

BACHELOR OF SCIENCE

May 2010

Department of Physics and Engineering

Para mis Padres,

por todo su amor y apollo

ACKNOWLEDGEMENTS

It is with great respect and admiration that I acknowledge my advisor, Professor Irina Mazilu, for her continued guidance and support during my time at Washington and Lee University. This thesis and the opportunities I have had to learn and engage in research over the course of four years would not have been possible without her encouragement and leadership. She has been an excellent mentor and friend and I truly admire her dedication to her students and am humbled to have been able to share in her work.

It is also with great pleasure that I would like to thank my father, Oscar Gonzalez, for the many hours he spent teaching me C ++ and reviewing and helping me build and debug my computer simulations. I have very fond memories of our many phone and email conversations and I could not have possibly succeeded in my research endeavors without his help. Whatever honors or awards my work should ever receive (even a Nobel Prize) I would be happy, honored and proud to share them with him.

It is with sincere gratitude and admiration that I thank Joel Shinofield, it is an honor to be able to call him my coach. I am deeply touched by his care and concern for me as an athlete, academic and individual, which has been unwavering. I have a deep respect for his character and for his dedication to his family and team. His advice, experience and my interactions with him have been a constant illumination on the path to uncovering and understanding the complexities of human existence.

I also acknowledge Washington and Lee University for supporting my research with Robert E. Lee Research grants for undergraduate research. Special thanks are due to the Physics Department for acknowledging and funding my work with the H.T. Williams Undergraduate Research Scholarship. I am also deeply indebted to Gabriela Zamora for allowing me to work with her and Professor Irina Mazilu on their project for microtubule growth dynamics.

This thesis and my attraction to complex systems has grown out of a long term passion and interest in a variety of subjects. The study of complex systems is truly an interdisciplinary subject and I am indebted to my professors from every field. I would like to specifically thank Professors Robin LeBlanc, Tyler Dickovick, David Sukow, Rebecca Harris, Paul Bourdon, and

Florentien Verhage for their tutelage and for showing and passing on to me their passion for learning, knowledge and action.

Over the course of four years of undergraduate study I have had many exciting experiences and have forged deep friendships and created lasting memories. I will never forget the many people who have made my college experience truly unforgettable. I am especially grateful to the swim team and to the other senior swimmers, Winston Stagg, Jonathan Geisen, Kevin Corn, Nick Talluri, Brandon Barnds, Ian Childers, and Dan Austin, you are incredible friends and teammates. I equally grateful to my fraternity brothers and my roommates, Bobby Bowler, Brad Watts, Ross Draber, Ryan Castle, Dalton Harris, Jeff White, and Grant Lewandrowski, for always being there for me and for reminding me that there is also a time to relax and enjoy the company of friends in life. I would also like to thank my great friends, Andrea Hanick and Corinne Smith, I have shared much of my time at school with them. I am deeply indebted to them for their advise and cannot imagine how different my experience at W&L would have been without their friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Traditionally, the objective of science has been to understand the fundamental simplicity and elegance of the laws of nature. The advancement of the human race has for a long time been predicated on the scientific discovery of the rules by which the underlying forces of nature and society operate. Every facet of our modern day lifestyle, from our ability to drive cars, build skyscrapers, create medicines for illness and connect with people from across the globe, relies on our understanding of the basic principles of every field from physics and biology to economics and political science. Never in the history of the world has the human race had such a wide-ranging and deep understanding of society, nature, and the universe as a whole. However, as we have pushed the boundaries of scientific discovery outward it has become readily apparent that our bodies, society and nature all exhibit complex behaviors that cannot be explained without considering the interactions of their constituent elements as part of a larger system. The study of complex systems is an exciting and new interdisciplinary science which has lead to novel insights into the beautiful and complex behavior of the natural world; it is the leading edge of 21st century scientific investigation.

The accumulation of scientific knowledge, as well as technological advancements, has finally made the study of complex systems accessible. However, the field itself is as intricate and diverse as the systems which it seeks to understand. While there are general concepts and goals that form the theoretical underpinning of complex systems analysis, there also exists a great amount of variation in the types of systems under investigation and in the methods available for doing so. The study of complex systems requires a strong grounding in both mathematical and analytic techniques as well as the design and engineering of experimental simulations, which would be impossible without the aid of modern day computing systems. As more and more investigators from every academic field begin to move into the realm of complex systems our understanding of the world will be dramatically enriched and altered. New discoveries and possibilities for the advancement of science, technology and society as a whole will emerge from the efforts and achievements of those scientists who work collaboratively to uncover the mysterious dynamics of complex systems.

1.1. Organization of the Thesis

The thesis is organized as follows. Over the course of the next six chapters we will move from providing a general overview of complex systems to more specific examples and problems in the field. Given the diversity of the field and the difficulty of providing an exact definition of complex systems we believe it is important to understand the unifying concepts of complex systems before moving on to specific examples of such systems in order to highlight those concepts.

In Chapter 2 we identify the important concepts of a general system. In particular we examine the elements of a system, describe general methods of analyzing systems and classify and define important features of system analysis. In Chapter 3 we tackle the more difficult task of developing and defining the "complexity" of the systems we seek to understand. Building upon many of the concepts presented in Chapter 2 we will further describe the elements that make up a complex system as well as provide more specific methods of analysis. Chapter 3 will provide an understanding of emergence a fundamental concept in the field of complex systems. In Chapter 4 we move into the realm of thermodynamics and computer simulations in order to present relevant methods of modeling complex systems. Specifically, we will present the problem of a one-dimensional random walk. Chapters 5 and 6 will present specific research in the area of non-equilibrium statistical mechanics. In Chapter 5 we present research of a Monte Carlo study of a two-lane driven diffusive system, with an interest in explaining the coarsening phenomena for both the steady and transient states of the system. Chapter 6 is a presentation of a complete analytic and computational treatment of microtubule growth. Both Chapters 5 and 6 outline specific areas for future research with regards to those specific systems and problems.

CHAPTER 2
A SYSTEMS APPROACH

2.1 Introduction

A system can be most generally described as a set of interacting or interdependent parts which form a unified whole. Today, the system is one of the most basic forms through which we classify and understand the natural world and society. Every day we both encounter and act as parts of a multitude of systems, including but not limited to mechanical, electrical, educational, political, economic and environmental systems. In this section we will provide a general concept of systems and systems theory as well as general methods for studying systems.

2.2. Large-Scale Systems

It is necessary to establish that "large-scale" is a relative measure of which physical size is not the defining characteristic. Indeed, microscopic systems such as the movement of molecular motors within the cell can be classified as large-scale systems. While other types of systems exist, our preoccupation is with systems that can be categorized on the basis of any one or more of the reasons given by V. Vemuri (1978):

- Too many elements are needed to accurately characterize the system. In other words, "the structure or configuration of the system is rarely self-evident" (Vemuri 2). In fact, in such systems there exist a large number of possible configurations of the system, each of which is unique.
- The relationships between the elements and the overall behavior of the system "are generally statistical in nature" (Vemuri 2).
- The systems under consideration are dynamic, "they evolve in time" (Vemuri 2). The system exists within an environment which remains outside the control of any observer and whose influence on the system is "not apparent at the outset" (Vemuri 2).
- Finally, large-scale systems are often characterized by the specialized approaches needed to understand them.

2.3 Analyzing Systems

Knowing where and how to begin investigating a system is a conceptually difficult endeavor. The process of analyzing a system invariably requires a fair amount of creativity and educated guessing; the types of assumptions made are crucial to the analysis of a system. However, while scientific discovery may at times come from monumental "leaps of faith" and the creative genius of certain individuals, there are systematic methods of investigation that provide general approaches to the study of any discipline. Three of those approaches are described below [1].

2.3.1 The Leibnizian Approach

This approach is a formal mathematical procedure based on the notion that "truth is *analytic*" (Vemuri 11). This is generally the way in which most of the laws of physics are derived, with the notable exception of certain laws in electrodynamics which were determined experimentally. This type of investigation tends to focus on the structure and "associated properties of a system" (Vemuri 11). The main methods of this approach are "*decomposition* [...] and *aggregation*" (Vemuri 11). Decomposition is the processing of breaking a larger system into smaller "subsystems" (Vemuri 11) which can be solved and later reconstructed to form the solution to the original system. This approach is related to the idea of universality, which will be described later on in this work. Aggregation is the process of uniting the variables of the original system in order to "reduce the dimensionality" (Vemuri 11) of a system. The Leibnizian approach is best adapted to systems with "simple and well-defined structure" (Vemuri 11) and in which the basic assumptions are "clearly definable" (Vemuri 11).

2.3.2 The Lockean Approach

Named after the English philosopher, John Locke, this approach is empirical in nature, resting on the "assumption that truth is *experimental*" (Vemuri 13). A model of this type of approach does not rest on "prior assumptions" (Vemuri 13) and its validity lies in its ability to determine relationships and predict future behavior. The Lockean approach is generally, although not exclusively, statistical in nature and requires the extensive collection of data. Some of the better known statistical tools include "regression analysis, analysis of variance, and correlation analysis" (Vemuri 13). One of the major challenges of analyzing systems via the

Lockean approach is "the identification, definition, quantification and measurement of relevant attributes" (Vemuri 13). Because systems are composed of a number of diverse elements it is often hard to distinguish which relationships and interactions are the most relevant, not to mention the difficulty of establishing the particular behavior of the system to which they are relevant. The uncertainties inherent in the Lockean approach are highlighted in our study of a two-lane driven diffusive system and while an exact solution does not exist we suggest a number of possible approaches.

2.3.3 The Kantian Approach

The Kantian approach is a synthesis of both the Lockean and Leibnizian approaches and was developed by the philosopher Immanuel Kant. The basis of this model is that the "experimental data and theoretical base are inseparable" (Vemuri 14). Therefore, the collection of data is not possible without the guidance of theory. Likewise, it is not possible to build a theory without empirical evidence. The analysis of complex systems can be generally best pursued via this approach, as a fully analytic or empirical understanding of complex systems is difficult to arrive at.

2.4 Important System Concepts

2.4.1 Linearity

The study of systems is greatly dependent on the linearity of the system under consideration. A linear system is one in which the behavior of the system is directly proportional to the variables used to describe the behavior. Mathematically, linear systems can be described using linear equations for which powerful analytic techniques already exist. The use of determinants and matrix methods "apply only for treating linear systems," (Vemuri 58) greatly simplifying the analytical approach. We can also make use of the principle of superposition when analyzing linear systems. Complex systems however, are not linear systems thus complicating the methods in which we can mathematically treat them. The principle of superposition does not apply to nonlinear systems nor are simple matrix methods enough to adequately handle the systems of equations which describe complex systems. Nonlinear systems are also often referred to as nonequilibrium systems, especially in physics.

2.4.2 Lumped Models

In a lumped systems theory, the behavior under consideration can be described completely by the relationship between inputs and outputs at "its external terminals" (Vemuri 59). Systems of this form are composed of an array elements "interconnected in some specific manner" (Vemuri 59). For instance, these connections could be treaties in international politics, emotions in social networks or bonds between "cells" of a gas lattice. The spatial distribution of the elements of a lumped model does not serve a primary function and is therefore disregarded. Mathematically, "a set of *ordinary* differential equations" (Vemuri 59) is enough to describe a lumped system, with time serving as the independent variable when investigating the dynamic behavior of a lumped system (Vemuri 1978). A distributed system is one in which the internal behavior of the components is also of interest (Vemuri 59). Distributed systems are considered "field problems" (Vemuri 59) and are modeled by systems of partial differential equations.

References

[1] V. Vemuri, *Modeling of Complex Systems: An Introduction* (Academic Press, Inc., New York, 1978).

CHAPTER 3
COMPLEXITY AND EMERGENCE

3.1 Introduction

The discussion of complexity that follows is directly linked, and in many ways merely an extension, of many of the topics covered in the previous chapter on large-scale systems; large-scale systems are complex systems. One of the underlying and fundamental concepts of science is the notion of universality. The concept of universality holds that there exist universal laws to explain the natural phenomena and diverse systems of the world. If commonality and universality did not exist then we would be unable to understand and scientifically study even simple systems that occur in the fields of physics and biology. Take for instance the concept that all matter is composed of the same indivisible building blocks, atoms (modern discoveries have actually shown that the smallest building blocks of the atom are quarks), an idea that has existed since the time of the Greeks. The motion of matter is also described by a set of universal mechanical laws, both classical and quantum, and the study of biology is based on the common molecular and cellular systems of organisms. The study of complex systems is an attempt to try to discern the universality that occurs when the systems are highly complex.

3.1.1 Examples and Elements of Complex Systems

Conceptually, one of the easiest ways to attempt to understand complex systems is by looking at examples of complex versus simple systems. Simple systems in physics include an oscillating spring and the orbits of planets. On the opposite end of the spectrum, governments, the brain and ecosystems are all examples of complex systems. There are also certain properties which we can attribute to complex systems and use in order to develop methods of classification [1]:

- Constituents (and their number)
- Interactions (and their strength)
- Formation/Operation (and their time scales)

- Diversity/Variability
- Environment (and its demands)
- Activity(ies) (and its[their] objective[s])

3.2 Emergence

Our ability to understand and study complex behaviors can be simplified by an understanding of the concept of emergence. Emergence is the idea that the collective behavior of a system is contained in the behavior of the parts of the system. We can understand and study the collective behavior insofar as we study the parts in the context of the entire system. Emergence can be used to refer to either locally emergent or globally emergent properties of a system and is an important concept of thermodynamics and statistical mechanics. The study of thermodynamics, which investigates the properties and behaviors of collections of gas particles, provides an easily understood example of locally emergent properties.

3.2.1 Local Emergence

If we consider a single gas particle we can describe its entire behavior by its position and velocity. However, if we collect a large number of gas particles together we can now measure the pressure and temperature of the collective gas. The temperature and pressure are emergent properties; they cannot be understood simply by understanding the behavior of a single gas particle [1]. If we remove a small sample of the gas we can still "define and measure" (Bar-Yam 10) the temperature and pressure. Phase transitions (such as from water to ice) are also a display of collective, locally emergent behavior than can be observed both at the macroscopic and microscopic level [1]. Notice also that local emergence is strongly tied to systems in equilibrium; our collection of gas particles are not interacting with the outside world.

3.2.2 Global Emergence

The study of complex systems concerns itself with studying and understanding globally emergent properties and behaviors. When we speak of global emergence we are referring to properties or behaviors of a system as a whole. The individual and collective behaviors of the elements of the system are interdependent; the whole system, macroscopically and microscopically, is affected if we remove only a small part of it [1]. As the complexity of a system grows, our ability to observe and conceptually grasp that the arising behaviors are emerging from the individual and collective elements of the system is often difficulty. For instance, we can understand that the associative memory that humans posses is a function of an extremely large and complex network of neurons and synapses. However, understanding consciousness as an emergent behavior of the brain and body is somewhat more difficult. Philosophically, Descartes' Cartesian dualism suggests that consciousness is a phenomena that is distinct from the body, or even distinct from the brain (mind-brain split). This would suggest that is impossible to scientifically study or understand consciousness. On the other hand, a complex systems approach would view consciousness as a globally emergent property of the neurobiological system that is our brain and body. The difficulty arises in studying and attempting to understand how the simpler systems and elements of our brain give rise to such a complex and mysterious behavior.

References

[1] Yaneer Bar-Yam, *Dynamics of Complex Systems*(Addison-Wesley, Reading, Massachusetts, 1992).

CHAPTER 4
COMPUTER SIMULATION AND RANDOM WALKS

4.1 Introduction

Over the years, computer simulations have become established methods of conducting research in a number of scientific fields. We are especially interested in the use of computer simulations in the study of non-equilibrium statistical physics. Computer simulations provide a unique method of investigation; they are neither purely analytic or experimental in nature, operating somewhere in between these two ends of the scientific spectrum. Simulations allow us to quickly and cheaply run a great number of "experiments" while easily changing a variety of parameters of a given system. With computers we can easily move to reduce or increase the dimensionality of a particular system [1], all the while generating copious amounts of data. The problems presented in Chapters 5 and 6 make heavy use of Monte Carlo computer simulations, which are used to explore variations of "random walk" problems. Random walk models are used frequently in statistical physics and thermodynamics and familiar examples of simple random walk problems include the diffusion of a gas particle and Brownian motion [1]. Given the importance of random walks to our work and to the study of complex systems at large, below we present a brief overview of a simple one dimensional walk.

4.2 One Dimensional Random Walk

4.2.1 General Concept

Although it would seem that the subject of drunkards would be better suited for investigation by a student of chemistry, one of the most common, idealized, examples of random walk behavior given in physics is that of a drunk walker leaving a bar. We consider the drunkard to start the walk at the exit of the bar, a location we mark as $x = 0$. We limit the walker to being able to take one, and only one step in a given time interval. Furthermore, we also establish that each step is of equal length l and that the direction of each step, right or left, is independent of the previous step [1]. During each interval of time, the walker has a probability p of taking a step to the right and a probability $q = 1 - p$ of taking a step to the left. After a series of time intervals and steps, we measure the final position of the walker. Three quantities are of interest to us in this problem, the mean net displacement of the walker, the probability of a walker undergoing a certain net displacement after a given number of steps, and the dispersion of the walker [1].

4.2.2 Variables and Quantities

After N number of steps the walker finds himself at a position $x = ml$, where m is the net displacement of the walker from the origin and l is the length of each step. The total number of steps taken, N, can be expressed as [2]:

$$N = n_1 + n_2 \tag{1}$$

where n_1 represents the total number of steps taken to the right and n_2 represents the total number of steps taken to the left [2]. From the relations given, we can assert that the net displacement is given by:

$$m = n_1 - n_2 \tag{2}$$

Using equation (1), we can rewrite the net displacement as [2]:

$$m = n_1 - (N - n_1) = 2n_1 - N \tag{3}$$

In N, the probability of any given sequence of n_1 steps to the right and n_2 steps to the left is given by [2]:

$$p^{n_1}q^{n_2} \tag{4}$$

The total number of possible ways in which N steps can be taken so that n_1 are to the right and n_2 are to the left is express by [2]:

$$\frac{N!}{n_1!\,n_2!} \tag{5}$$

Because the displacement m can be determined, as in equation (3), solely by the total number of steps N and the number of steps taken to the right n_1, we can therefore equate the probability $W_N(n_1)$ of taking n_1 steps to the right (in a total of N steps), to the probability $P_N(m)$ of the walker being at position m after N steps:

$$P_N(m) = W_N(n_1) \tag{6}$$

Joining equations (4) and (5) the probability $P_N(m)$ is given by the binomial distribution [2]:

$$P_N(m) = \frac{N!}{n_1!n_2!}p^{n_1}q^{n_2} \tag{7}$$

4.2.2 Calculation of Mean Values

The mean number of steps to the right and to the left can be established simply from the total number of steps N and the probabilities p and q [2]:

$$\overline{n_1} = Np \tag{8}$$
$$\overline{n_2} = Nq \tag{9}$$

Since p and q sum to 1, we can properly establish that the mean number of steps in each direction sum to N [2]:

$$\overline{n_1} + \overline{n_2} = N(p+q) = N \tag{10}$$

Using equation (2) the mean displacement (measured to the right in units of step length l) can be written as [2]:

$$\overline{m} = \overline{n_1 - n_2} = \overline{n_1} - \overline{n_2} = N(p-q) \tag{11}$$

When $p = q = \frac{1}{2}$, we can see that the mean net displacement of the walker is zero, the random walk ends where it began.

The final quantity of interest is the dispersion, $\overline{(\Delta n_1)^2}$ which can be written as [2]:

$$\overline{(\Delta n_1)^2} = \overline{(n_1 - \overline{n_1})^2} = \overline{n_1{}^2} - \overline{n_1}^2 \tag{12}$$

The quantity $\overline{n_1{}^2}$ can be computed by [2]:

$$\overline{n_1{}^2} = \sum_{n_1=0}^{N} W(n_1) n_1{}^2 = \sum_{n_1=0}^{N} \frac{N!}{n_1!\,(N-n_1)!} p^{n_1} q^{N-n_1} n_1 \tag{13}$$

Through differentiation we can produce an extra factor of n_1 and simplify (13) to:

$$\begin{aligned} \overline{\overline{n_1{}^2}} &= p[N + pN(N-1)] \\ &= Np[1 + pN - p] \\ &= (Np)^2 + Npq \\ &= \overline{n_1}^2 + Npq \end{aligned} \tag{14}$$

4.3 Random Walk Variants

The one dimensional random walk is the foundation for a number of problems in statistical physics and thermodynamics. It can easily be built upon and expanded as problems and models grow in complexity. As will be seen in Chapters 5 and 6, the dynamics of molecular motors and microtubule growth can be modeled by variants of the one dimensional random walk. We simply add more dimensions, multiple walkers and biased directions to the movements of each of the elements. Another key feature of random walks is that they can be modeled exceptionally well and with some degree of ease by Monte Carlo computer simulations.

References

[1] Harvey Gould, Jan Tobochn ik, *An Introduction to Computer Simulation Methods* (Addison-Wesley, Reading, Massachusetts, 1988).

[2] F. Reif, *Fundamentals of Statistical and Thermal Physics* (McGraw-Hill, San Francisco, California, 1965).

CHAPTER 5
A MONTE CARLO STUDY OF A TWO-LANE DRIVEN DIFFUSIVE SYSTEM

5.1 Introduction

Bio-molecular motors are nature's nanomachines that convert chemical energy into mechanical work with performance and scale unparalleled by any man-made devices. Molecular motors are responsible for the transport of fuel and genetic information along the tiniest of tracks such as actin filaments, microtubules, RNA or DNA. There is great interest in developing a fundamental understanding of the operating principles of bio-molecular motors in order to exploit this knowledge to harvest, modify, and integrate these macromolecular assemblies into useful devices from nano to macro scale. The behavior of the cell and its structure depends on the active, directed transport of macromolecules, membranes or chromosomes at the inter and intra cellular level. Just as a disruption of traffic hurts the functioning of a city, so can defective molecular transport result in a variety of diseases. So far, most biological studies were focused on the behavior of individual motors.

Molecular motors have two important characteristics: they interact with cytoskeletal filaments, and this interaction is modulated by the ATP hydrolysis reaction. If this system were at chemical equilibrium, the ratios of the local binding and detachment rates would be fixed and there would be no net movement. But in the cell, the ATP hydrolysis reaction is maintained *out of equilibrium*. The net motion of the molecular motors along the filaments happens above a critical concentration of ATP, for which the rate of stimulated detachment is sufficiently high. This situation is analogous to phase transitions in condensed matter physics. In paramagnetic to ferromagnetic transitions, for example, cooperative behavior can align spins, even in the absence of external magnetic fields. Indeed, the general mathematical properties at the critical point are closely related in both systems. But there are significant differences as well; the team of motors is a non-equilibrium system that is controlled by chemical kinetics, as opposed to an equilibrium system that is controlled by temperature. We provide below a short description of cytoskeletal motors and their biological functions.

5.1.1 Basic Description of Cytoskeletal Motors

Our focus is on one category of bio-molecular motors, namely the cytoskeletal motors. Their biological functions are summarized in the following table [10]:

Cytoskeletal Motors				
Motor	Acts on	Energy source	Motion	Role
Kinesin	Microtubule	ATP	linear	Mitosis Organelle transport
Myosin	Actin	ATP	linear	Muscle contraction Organelle transport
Dynein	Microtubule	ATP	linear	Ciliary beating Organelle transport Mitosis

Table 5.1 Summary of the behavior of three cytoskeletal motors; Kinesin, Myosin and Dynein

Each molecular motor family has members that transport vesicles through the cell cytoplasm along linear assemblies of molecules, actin in the case of the myosins and tubulin for both of the other families. The kinesins and dyneins move or 'walk' along microtubules - tubes constructed from tubulin - carrying their cargo. The microtubules are polar structures - only one end (the plus end) is capable of rapid growth by adding more tubulin molecules - and the action of the motor molecules is polarized so that they move in one direction only. Kinesins are plus-end directed motors while dyneins move towards the minus end of the microtubule. The motive power for muscle activity is provided by myosin motors, organized as thick filaments which interact with an array of thin actin filaments to cause the shortening of elements within each myofibril. Modern experimental setups using optical tweezers, fluorescence and evanescent field microscopy, viscous drag, atomic force microscopy and micro needles have produced quite a large amount of data on single molecule properties [11,12,13]. However, very little is known about the cooperative effects arising from motor-motor interactions [14].

5.2 Driven Lattice Gas Models for Molecular Motors

In recent years, the non-equilibrium physics community has shown an increased interest in modeling molecular motors using the driven lattice gas approach. Experiments [15] showed that cytoskeletal molecular motors perform a linear, unidirectional motion. Motivated by their theoretical simplicity and their non-trivial non-equilibrium behavior, the choice of asymmetric exclusion process (ASEP) systems and their close relatives, the driven diffusive systems, in modeling some molecular motors is perfectly natural. In contrast to their equilibrium counterpart, one-dimensional driven systems exhibit a variety of non-trivial behavior, such as boundary induced phase transitions, phase separation and spontaneous symmetry breaking even when the dynamics is local [16]. In biological terms, this translates into a variety of "cell malfunctions" and "molecular traffic jams" with serious consequences such as genetic mutations and various diseases [17].

Many studies have considered models in which the dynamics are the same everywhere in the system [18,19]. However, when trying to relate these systems to many interacting molecular motors, the effect of non-conservation and disorder cannot be ignored in many cases [20]. These models were studied using both computer simulation techniques and analytical means [21]. Some studies are concerned with the non-equilibrium transport properties of the motors [22], others concentrated on the physical interactions of the models with their support (actin filaments or microtubules) [23], such as load-velocity relations or adhesion forces between motor and support [24]. The formation of localized shocks in one-dimensional driven diffusive systems with spatially homogeneous creation and annihilation of particles was also studied [25]. A special interest was shown in describing the movements of molecular motors on cytoskeletal filaments as random walks on a line [26]. At any given time, the molecular traffic involves a variety of molecular motors, so efforts were made to study systems with two species of active molecular motors moving on filaments into opposite directions [27].

5.2.1 Universality

A very important concept in the context of driven systems is the concept of universality. A vast amount of experience from equilibrium statistical mechanics has taught us that macroscopic long-distance, long-time properties are independent of numerous microscopic details, such as the precise form of inter-atomic interactions (as long as they remain short-ranged). Systems differing only in such irrelevant microscopic features are said to belong to the same universality class. If universality holds, then simple models, which are within the reach of a theoretical study, can be designed to understand and predict characteristics of much more complex systems belonging to the same universality class.

5.2.2 Introduction of Driven Diffusive Models

Katz, Lebowitz, and Spohn introduced the first-driven diffusive model in 1983, as a seemingly trivial modification of the Ising lattice gas model. The prototype displayed such surprising and counterintuitive behavior, that it sparked the interest of the condensed matter community and a plethora of studies in variations and extensions of the model followed.

5.2.3 A Two-Lane Driven Diffusive Model

Our two-lane model with two types of particles and open boundary conditions is a member of the driven diffusive class. In recent studies[33], it was found that there is a stark contrast between the one-dimensional model and the two dimensional counterpart. For the one dimensional case, an analytical solution was found [34], that showed the absence of a phase transition for the system, equivalent to the one-dimensional equilibrium Ising model gas. However, the fully two dimensional model displays a particularly rich phase diagram [35]. Exploring the concept of a "lower critical dimension", extensive Monte Carlo studies were performed in the case of a "quasi-one dimensional", or a "two-lane model". To this date, the results are inconclusive. On one hand, the computer simulations indicate the presence of clustering and phase transition. On the other hand, analytical studies [35] showed that this is just a finite-size effect, and that the clustering disappears as the system reaches the thermodynamic limit.

The purpose of this study is to explore in detail the cluster growth in a two-lane lattice gas model using both simulations and analytical techniques. We focus on both the transition regime and the steady-state, in an effort to shed some light on the behavior of the system at this lower-dimension boundary. Our system has a dual purpose: it is a wonderful avenue to explore the novel non-equilibrium behavior at this critical lower dimension and also serves as a simple model for the collective behavior of the kinesin and dynein molecular motors.

5.3 Computer Model of a Two-Lane System

At any given time, the traffic of molecular motors in cells is quite heavy, since every single type of motor performs its own specific task. Experimental data shows that the dynein and kinesin motors share the same filament tracks, but move in opposite directions. We are proposing to explore the motor to motor interaction by using a two-species two-lane driven model.

Our model consists of a lattice that contains Lx2 sites, where L is the variable length of the lattice. A site can be empty, 0, or be occupied by at most one particle. There are two types of particles, "positive" + and "negative" − , which travel in biased directions across the two lane lattice with open boundary conditions in both directions. The particles are driven in opposite directions by an external field along the x-axis; in a visual sense positive particles move to the right and negative particles move to the left. Hopping attempts are only successful if the target site is not occupied by another motor.

Without going into the details, our computer model performs the following actions in one Monte Carlo step, which is then repeated a minimum of Lx2 times:

- Randomly picks a site along the lattice
- Randomly picks a bond (can be imagined as the walls separating sites) connecting two nearest-neighbor sites on the lattice. There are four types of bonds: a right bond, a left bond and two vertical bonds, top and bottom.
- Based on the occupation of the two sites on either side of the bond, the program carries out the appropriate movements. The movements and interactions of the particles are governed by the following rules:
    - No backwards jumps are allowed.

  - If the bond is in the x-direction and connects a particle and a vacancy in either a $+0$ or $0-$ combination, then a particle-hole exchange always occurs. If the bond connects a particle pair $+ -$ then a charge-exchange occurs with a probability γ.
  - If the bond is in the vertical y-direction (cross-lane) the following hold true: particle-hole exchanges always occur and charge-exchanges never occur. Furthermore, although open boundary conditions exists, a particle in the bottom cell with a bottom bond selected cannot hop out of the lattice (the same is true for a cell in the top lane with a top bond selected)
- After performing a movement the program moves to the left end of the lattice and randomly picks the first cell in one of the two lanes. Since negative cells move right to left, if a negative cell occupies this end cell then a routine will be executed that will cause the particle to exit the lattice according to some exit probability μ. The same process is then repeated at the right end of the lattice for positive particles, where the exit probability is controlled by a variable δ.
- Once again at the ends, the program runs a routine checking for empty end sites (randomly choosing between lanes) into which particles can enter the lattice. As opposed to the scenario of exiting the lattice, positive particles will enter from the left with probability α, and negative particles will enter from the right with probability β.
- The program then randomly selects another site along the lattice. If the site is empty then the program runs an attachment routine which randomly chooses either a positive or negative particle to attach to the lattice with a given probability λ. This probability is doubled if any of the nearest neighbor sites of the chosen cell are also occupied.
- The final movement of one Monte Carlo step is a detachment routine in which once again the program selects a random cell. If the cell is occupied the particle in the cell will detach itself from the lattice according to some given probability τ.

Other Rules of the Lattice:

- Initially half the sites of the lattice are populated with particles.
- The number of initial positive particles equals the number of initial negative particles.
- A cluster is defined as any number of particles greater than 1 that are connected by nearest neighbor sites.

5.3.1 Monte Carlo Simulations

In studying non-equilibrium systems, Monte Carlo (MC) simulation techniques prove to be invaluable tools. Using computer simulations, we can easily probe how a well defined model is behaving under certain conditions. Using these simulations we can study the critical behavior, measure order parameters and fluctuations, and check the accuracy of our analytical treatment of a model.

Our intent is to use Monte Carlo simulations to explore the coarsening phenomenon for our two lane system as it evolves towards a jammed state. Also, these computer simulations provide us with some insight regarding the transitory behavior of the system where the matrix approach fails. The sample results presented below are a small fraction of the data that we have so far analyzed. Our "experimental" results from the Monte Carlo simulations have far outpaced our ability to develop our analytical model; the simulations are too complex and the amount of data recorded is simple staggering. However, the results do provide significant and interesting data as well as insights into ways to further understand this system.

For our simulations we choose a system size of L = 100 and one trial involved the execution of 500,000 of the MC steps detailed above. At the end of one trial we record the number of clusters, the size of the biggest cluster and the starting and ending positions on the lattice of each cluster. Our investigation explores the effects of the attach and detach rates across different horizontal exchange rates (γ). We therefore held the entrance and exit rates constant at .5. The attachment, detachment and gamma rates were allowed to vary from 0 to 1 in intervals of .1 in the following manner:

- Initially the attachment and detachment rates would be set to zero and γ would be allowed to vary from 0 to 1 in .1 intervals.
- 100 trials would be run at each γ value and the results would be averaged for each γ value (total number of trials 1100).
- The detachment rate would then be incremented by .1 (while keeping the attachment rate at zero) and another set of 1100 trials would be run, again 100 trials at each γ value from 0 to 1.
- The detachment rate would be incremented this way all the way to 1 (essentially attachment being held constant while detachment would vary and looking across all

gamma values), then would be reset to 0 and the attachment rate would be incremented for the first time to .1 and the whole process repeated until the attachment rate had been allowed to vary all the way from 0 to 1. (Total number of trials 133,100).

- This entire process was once again repeated, this time incrementing the detachment rate at the slowest pace (detachment held constant and attachment varying, across all gamma values).

In each subset of 1100 trials, the computer simulation averages, at each gamma value, the number of clusters, the size of the biggest cluster, the number of positive, negative and total particles that enter, exit, attach and detach from the lattice as well the number that attempt to enter, exit, attach and detach. It can also, for each individual trial, provide snapshots of the number of particles that have entered, exited, attached and detached from the lattice at intervals of 25,000 MC steps.

In all of the trials, no matter whether we were varying the detachment rate (first 133,100 trials) or the attachment rate (second 133,100 trials) the average number of clusters and the size of the biggest cluster were consistent over all γ values, therefore the results we present are those with a γ value of .5. The only notable exception to this result occurred on the two occasions in which both the attachment and detachment rates are equal to zero.

5.4 Simulation Results

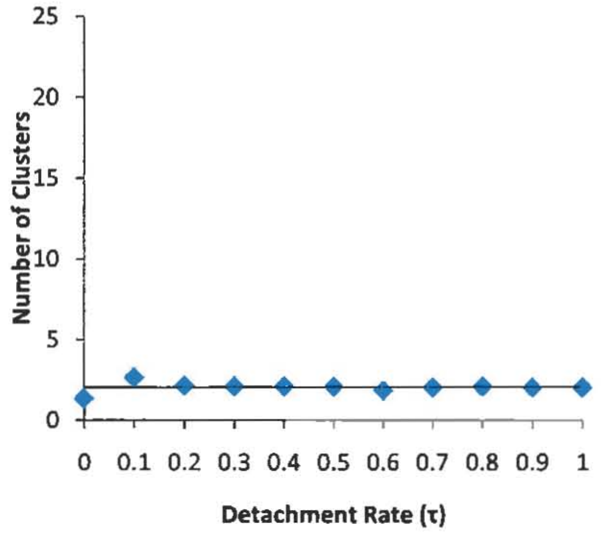


Figure 5.1. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\lambda = 0$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

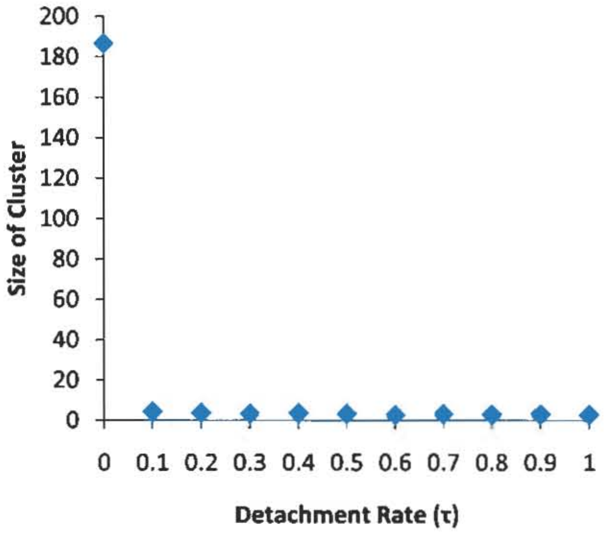


Figure 5.2. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\lambda = 0$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

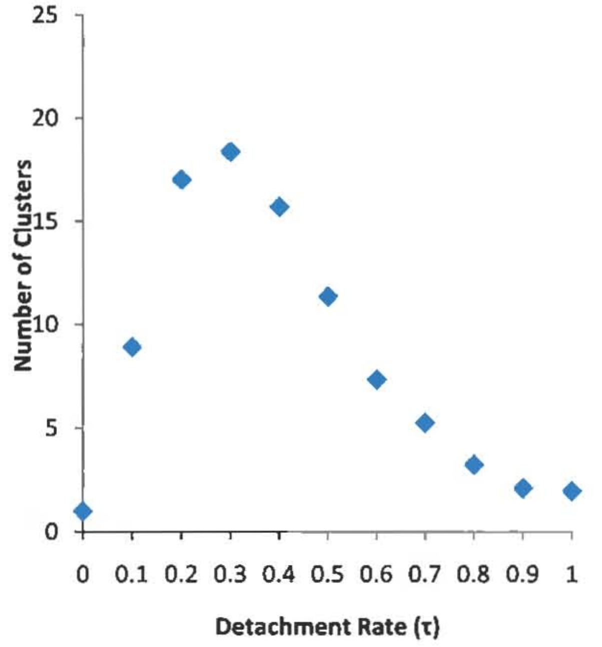


Figure 5.3. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\lambda = 0.1$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

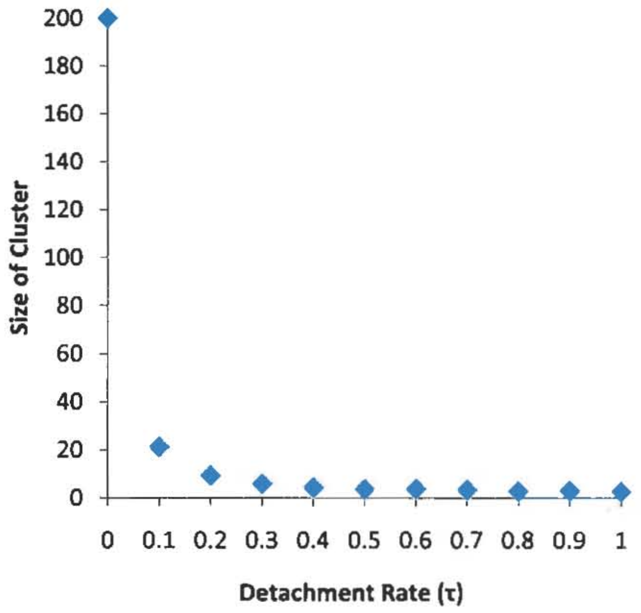


Figure 5.4. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\lambda = 0.1$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

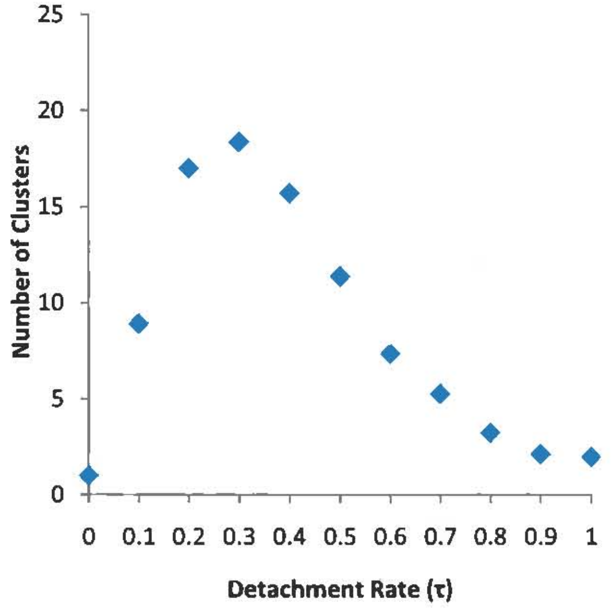


Figure 5.5. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.2 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

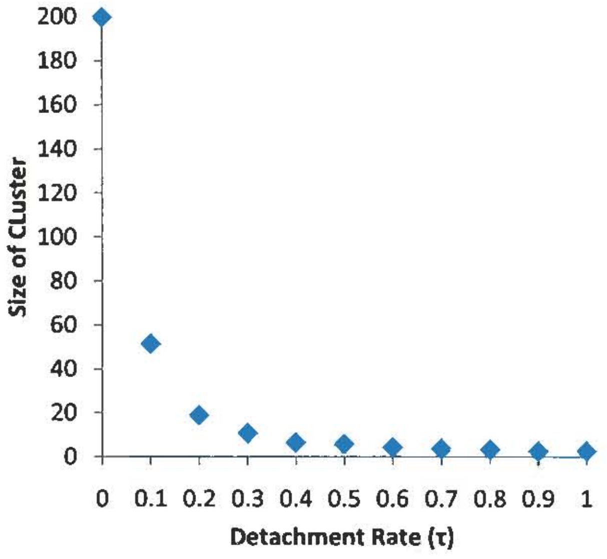


Figure 5.6. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.2 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

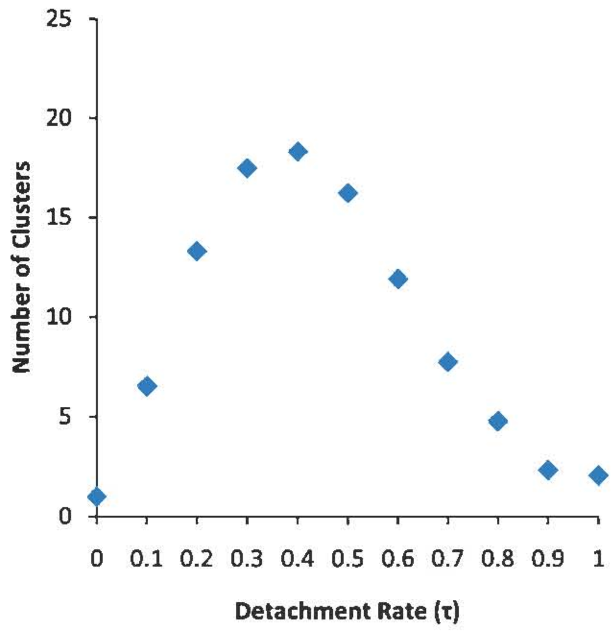


Figure 5.7. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.3 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

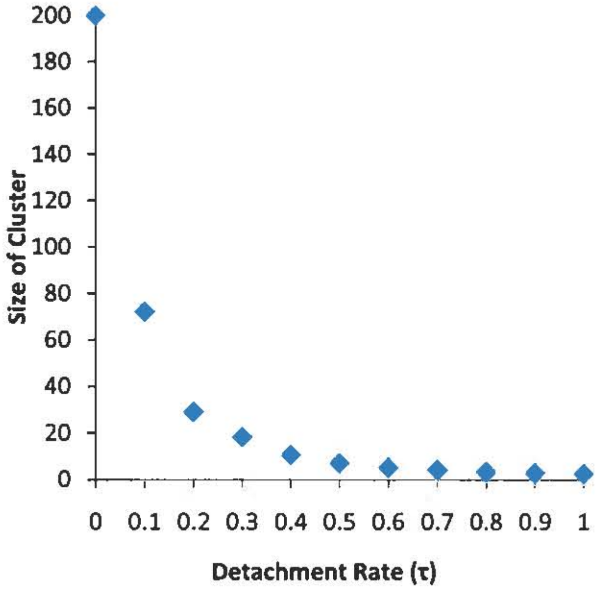


Figure 5.8. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.3 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

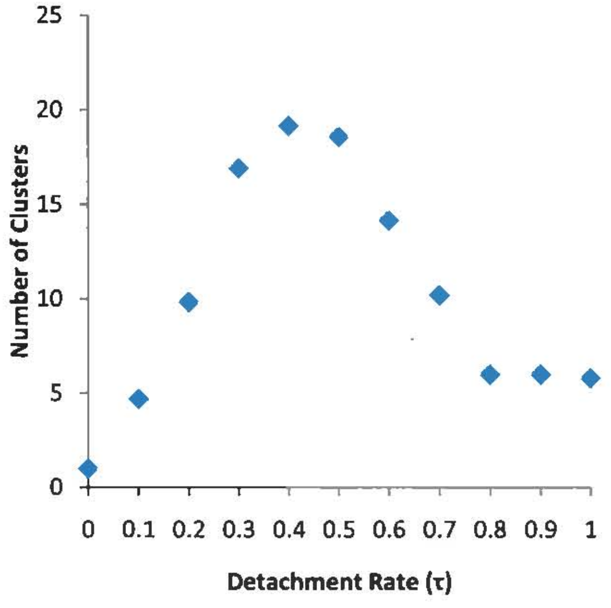


Figure 5.9. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.4 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

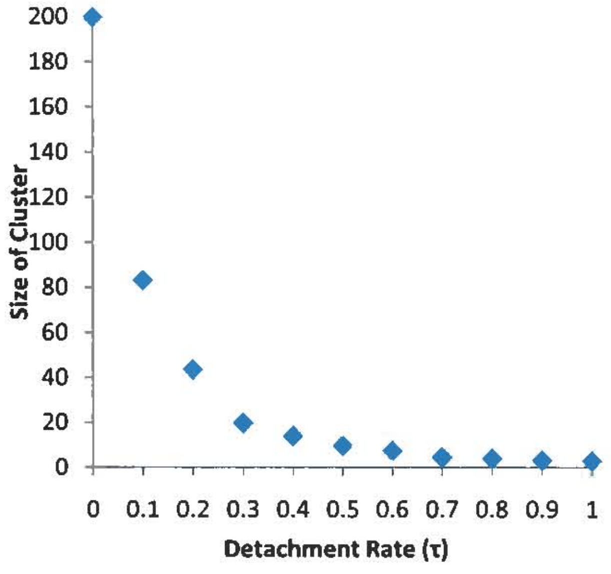


Figure 5.10. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.4 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

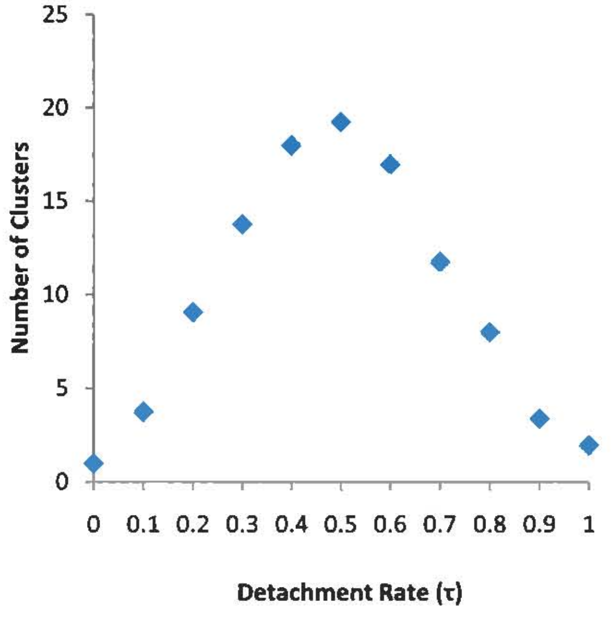


Figure 5.11. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.5 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

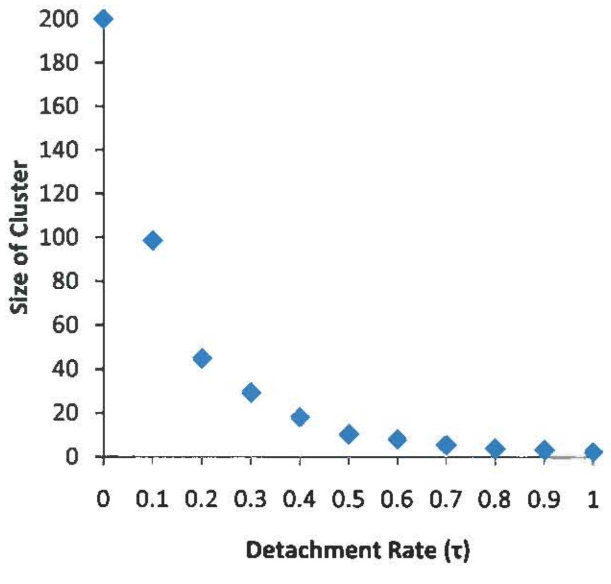
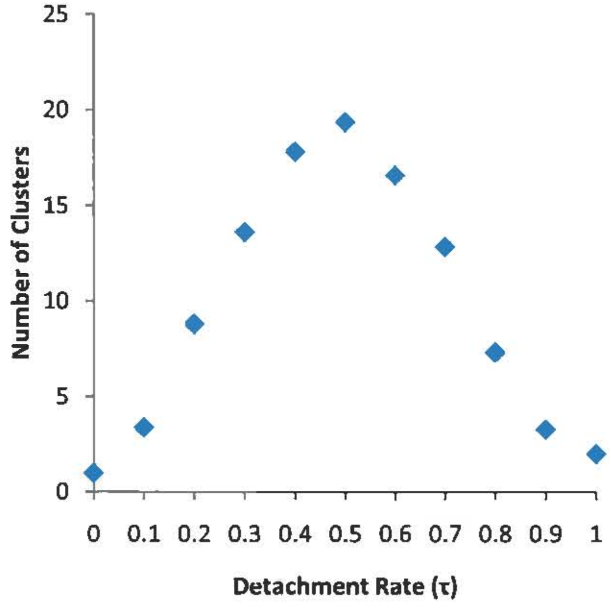


Figure 5.12. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.5 (e) γ = 0.5 (f) β = α = δ = μ = 0.5



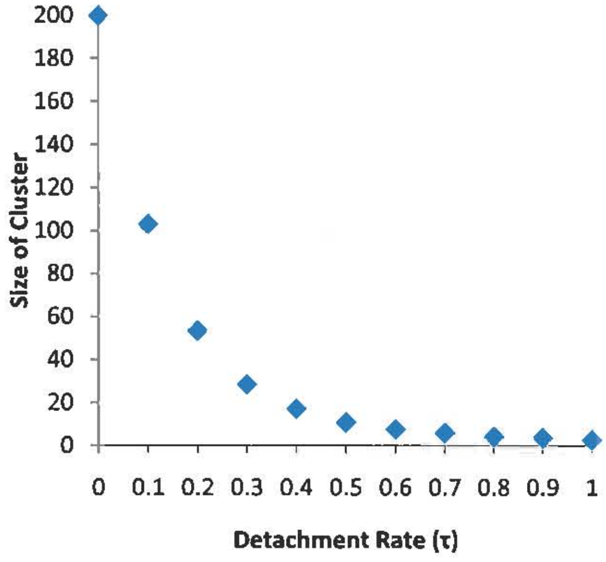
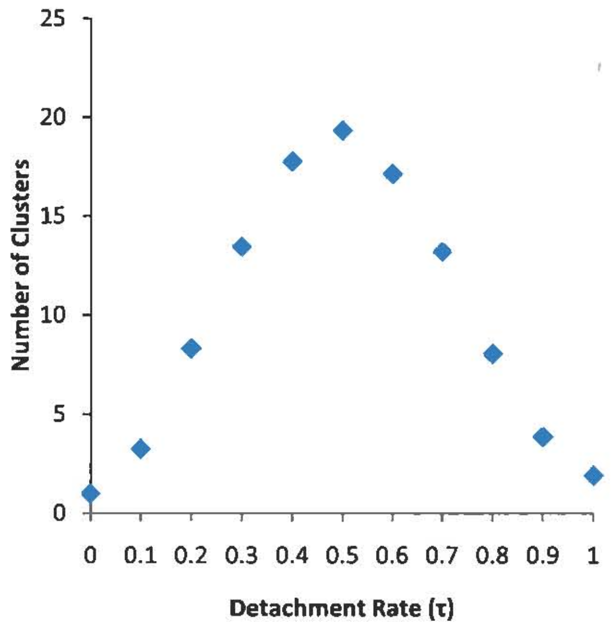


Figure 5.13. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.6 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

Figure 5.14. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.6 (e) γ = 0.5 (f) β = α = δ = μ = 0.5



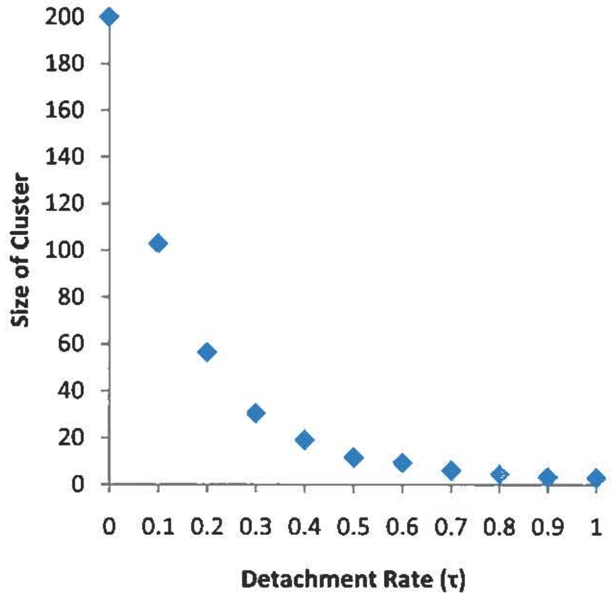


Figure 5.15. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.7 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

Figure 5.16. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.7 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

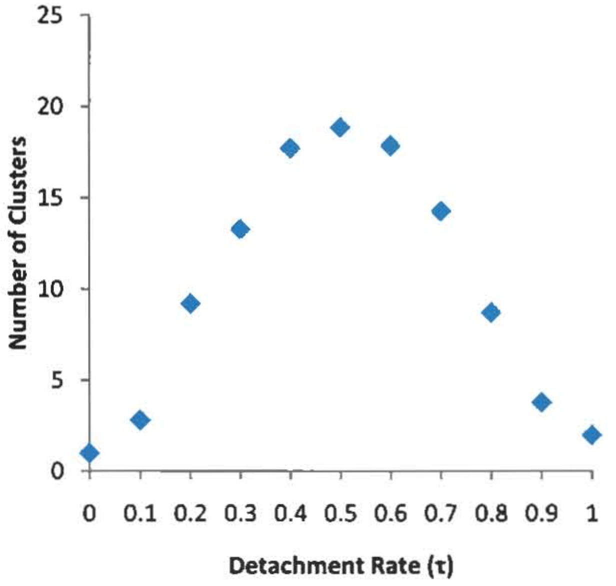


Figure 5.17. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.8 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

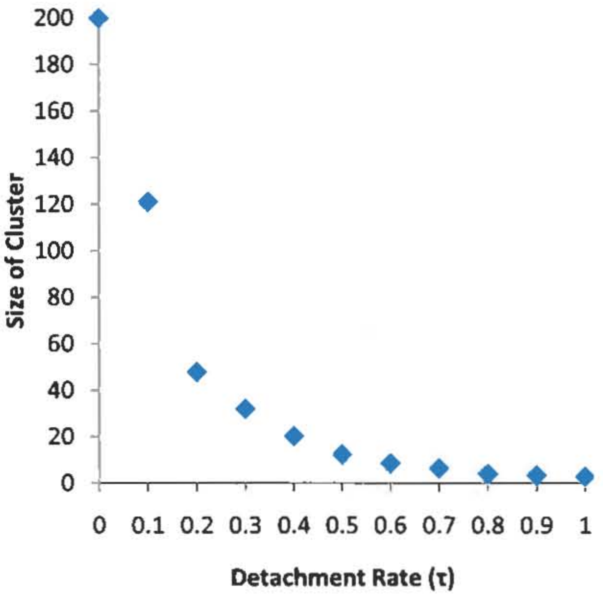


Figure 5.18. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.8 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

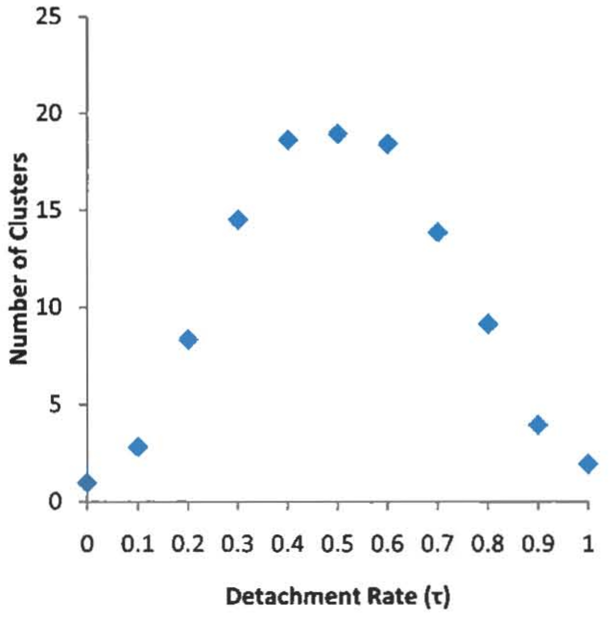


Figure 5.19. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.9 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

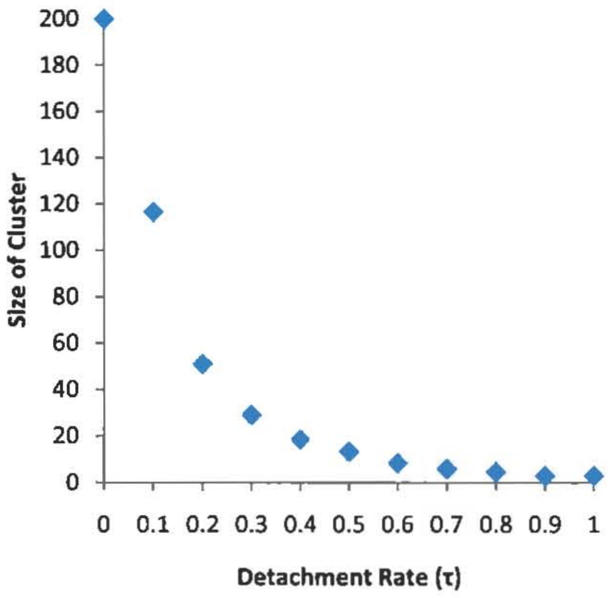
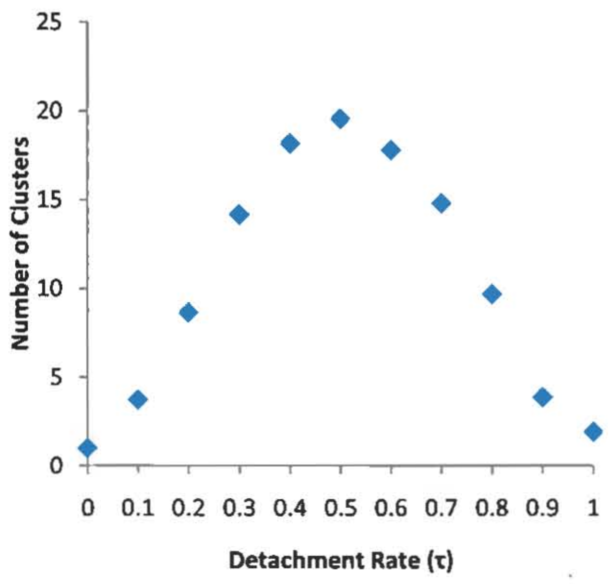


Figure 5.20. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) λ = 0.9 (e) γ = 0.5 (f) β = α = δ = μ = 0.5



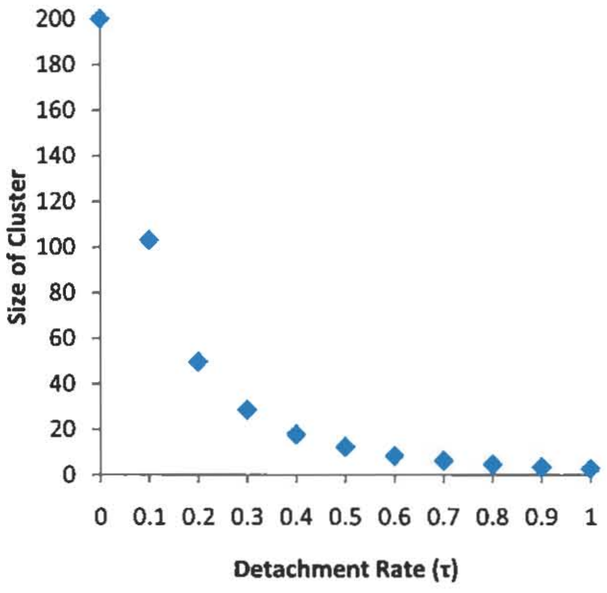
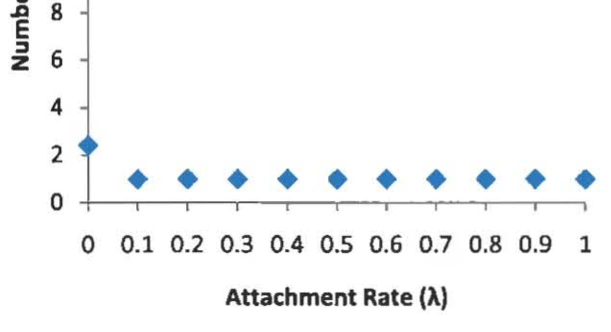


Figure 5.21. Dependence of Average Number of Clusters on (a) varying Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\lambda = 1$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

Figure 5.22. Dependency of Average Size of Biggest Cluster on (a) Detachment Rate (τ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\lambda = 0.9$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$



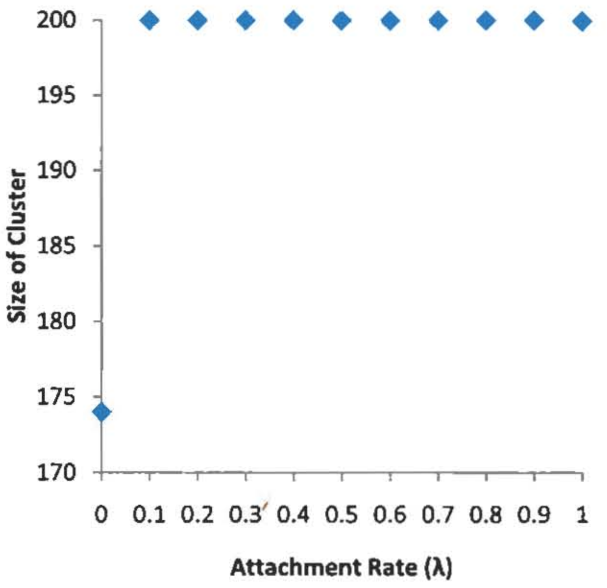


Figure 5.23. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

Figure 5.24. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

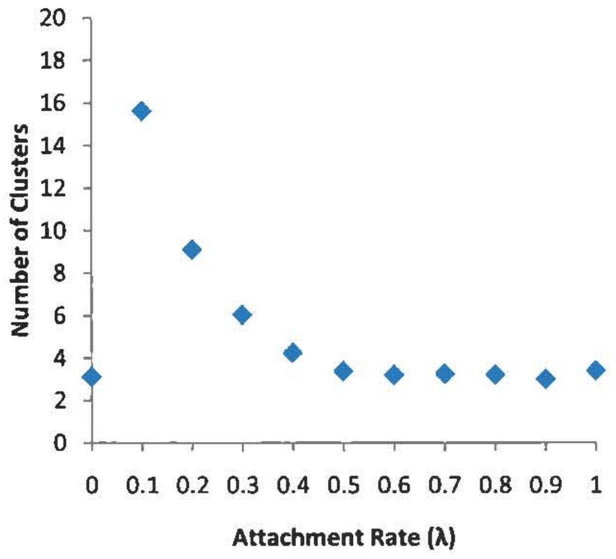


Figure 5.25. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.1$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

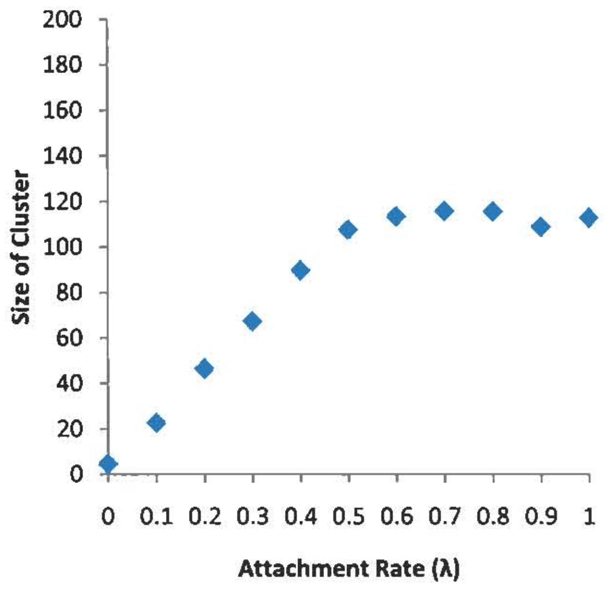


Figure 5.26. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.1$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

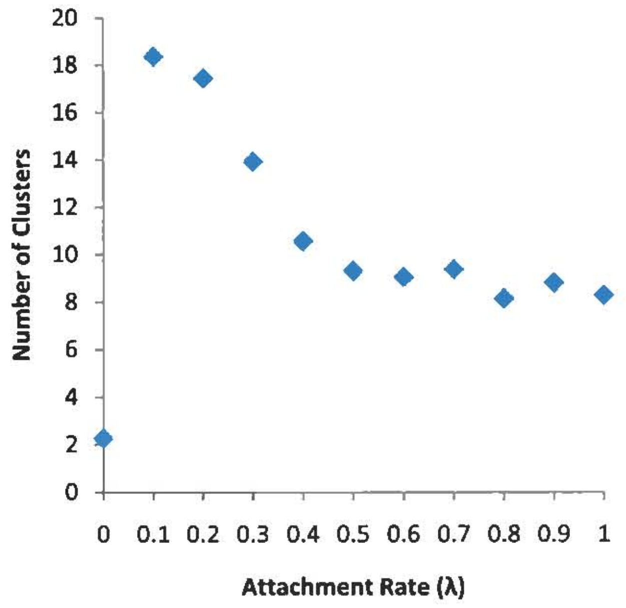


Figure 5.27. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.2$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

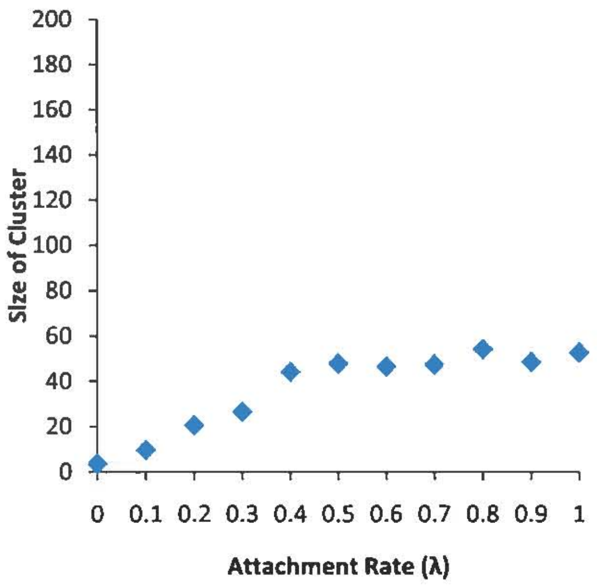


Figure 5.28. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.2$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

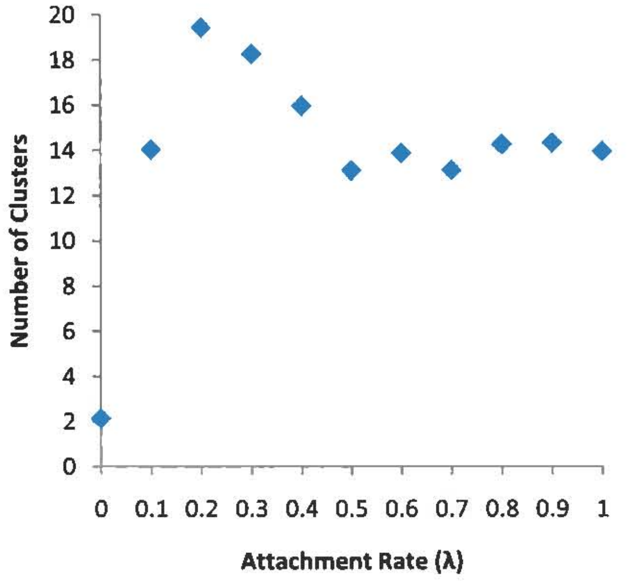


Figure 5.29. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.3 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

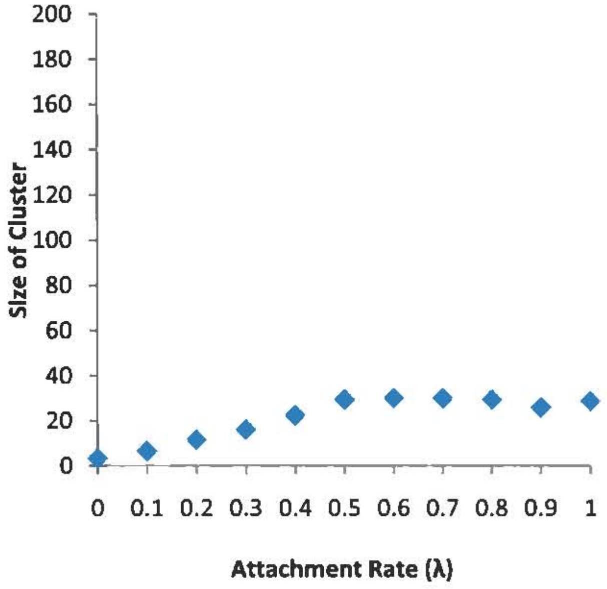


Figure 5.30. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.3 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

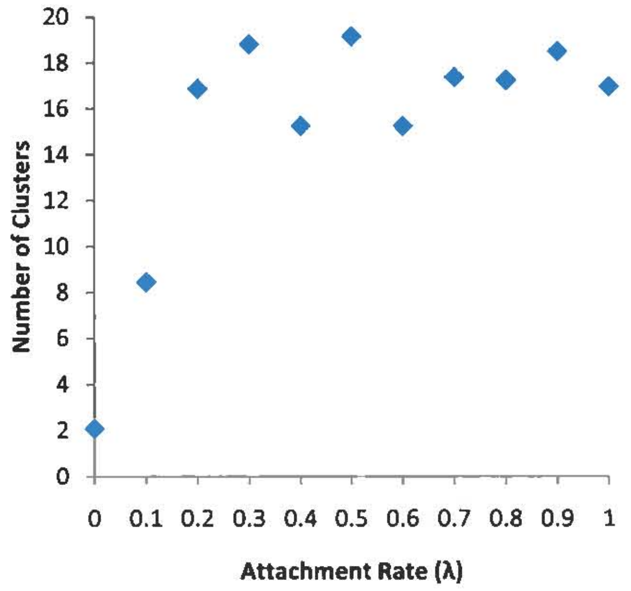


Figure 5.31. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.4 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

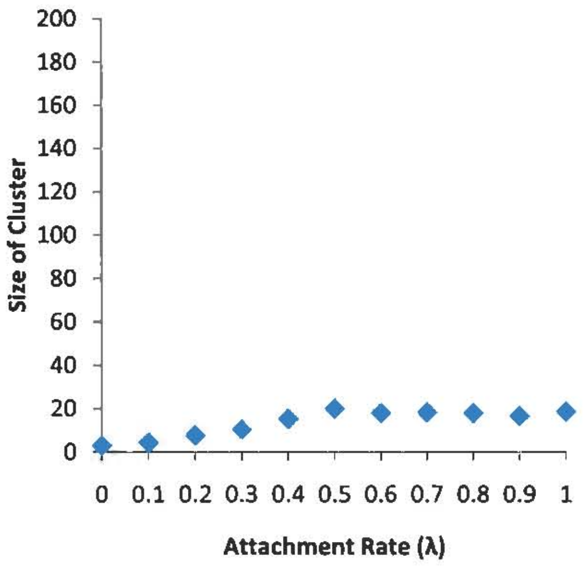


Figure 5.32. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.4 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

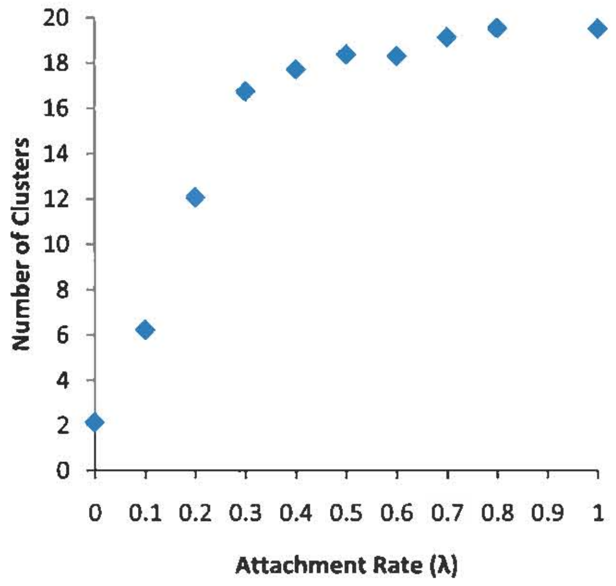


Figure 5.33. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.5$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

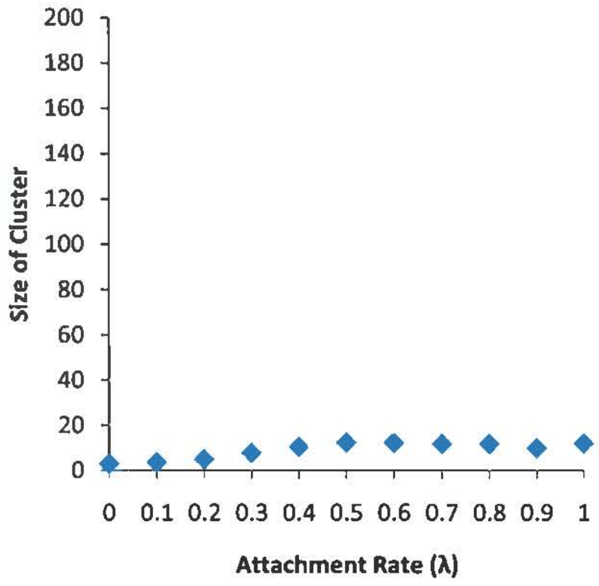


Figure 5.34. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.5$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

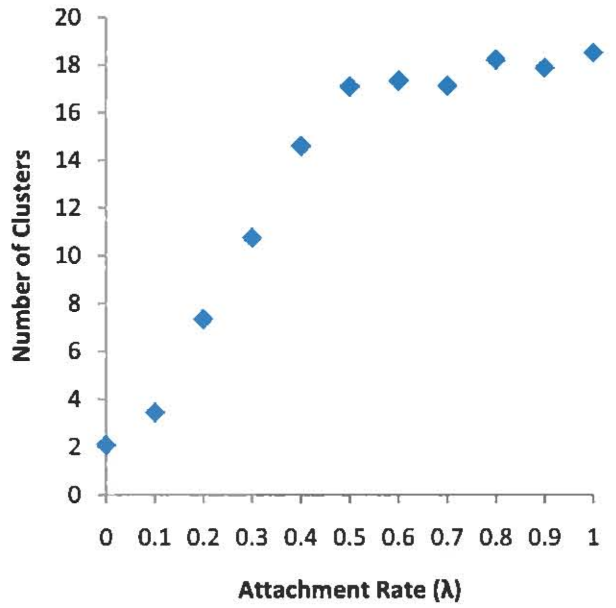


Figure 5.35. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.6$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

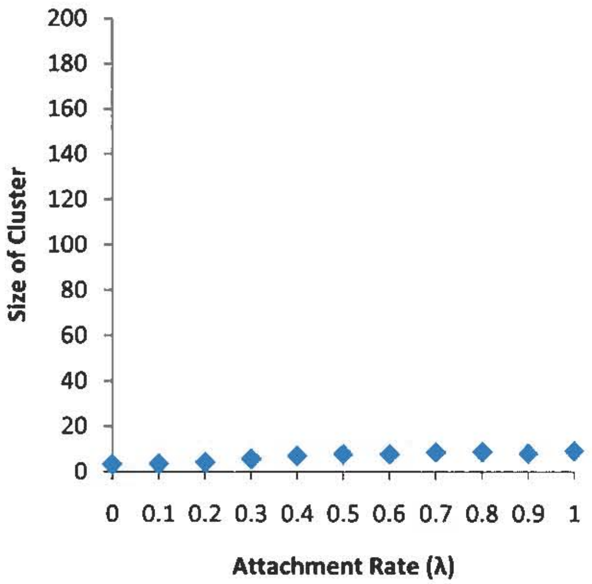


Figure 5.36. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) $\tau = 0.6$ (e) $\gamma = 0.5$ (f) $\beta = \alpha = \delta = \mu = 0.5$

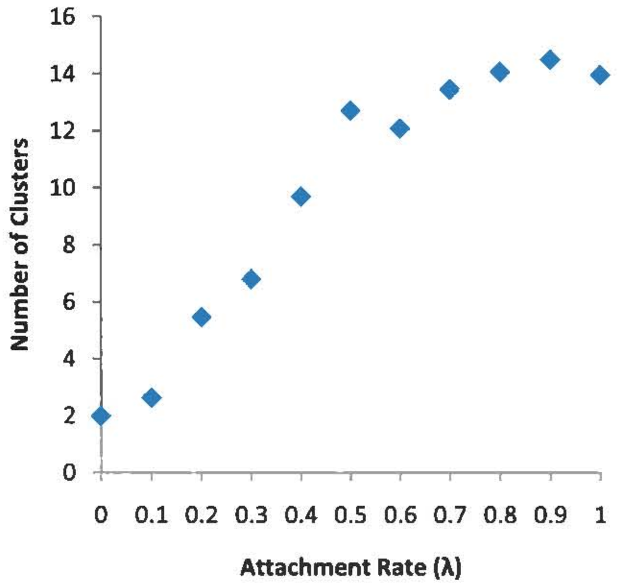


Figure 5.37. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.7 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

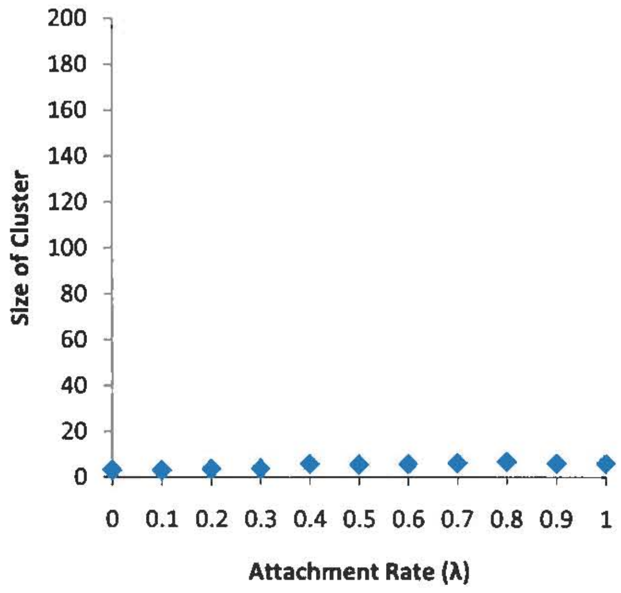


Figure 5.38. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.7 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

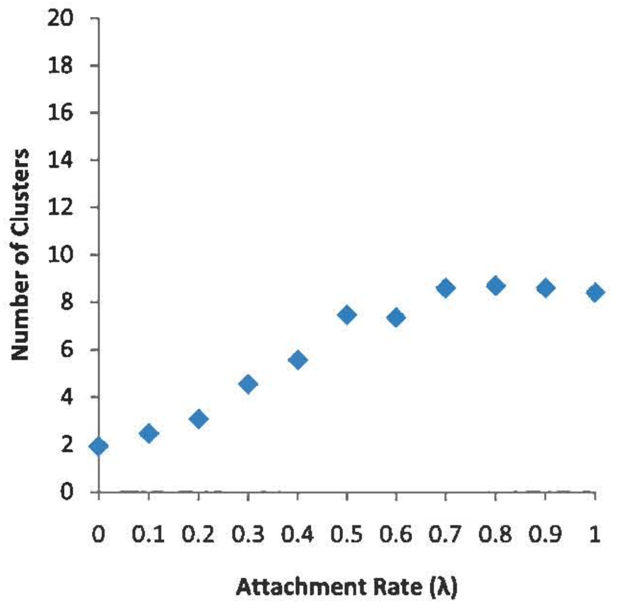


Figure 5.39. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.8 (e) γ = 0.5(f) β = α = δ = μ = 0.5

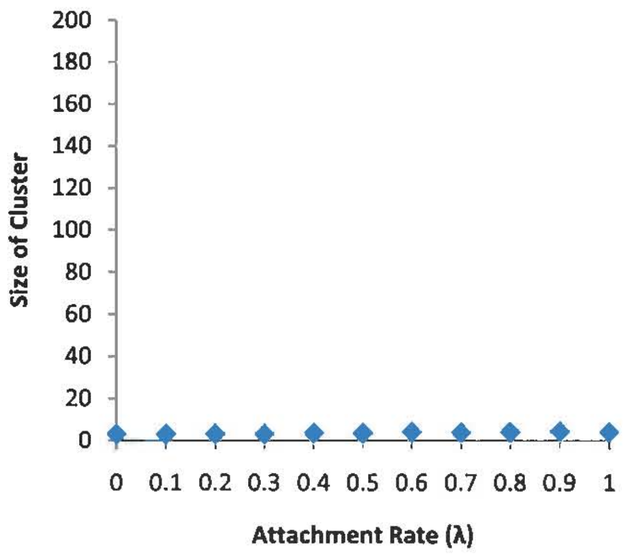


Figure 5.40. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.8 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

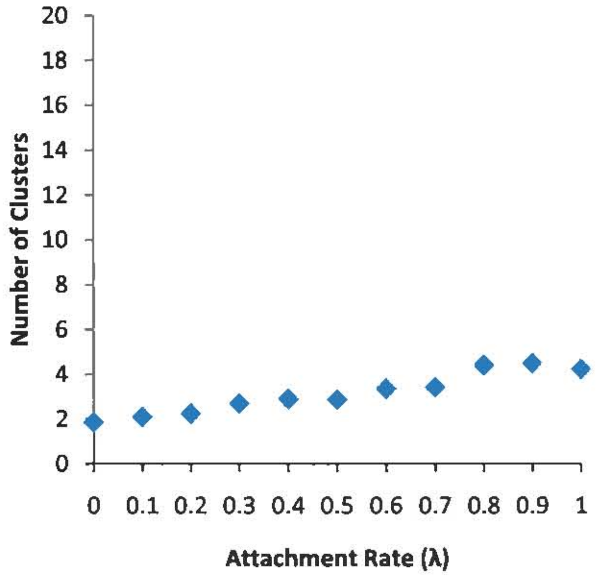


Figure 5.41. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.9 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

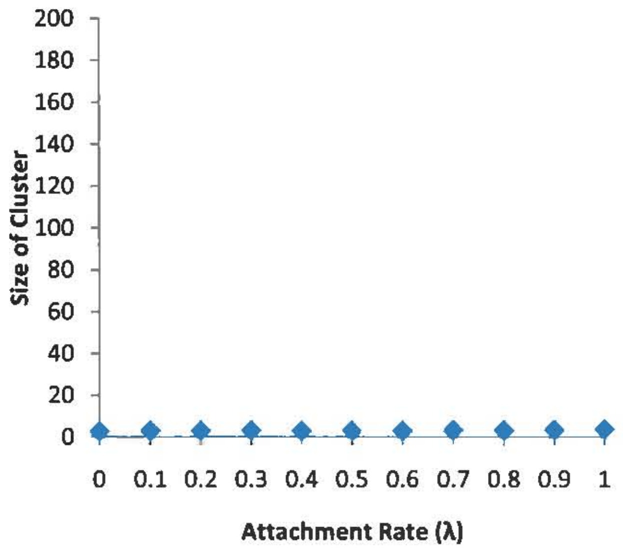


Figure 5.42. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 0.9 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

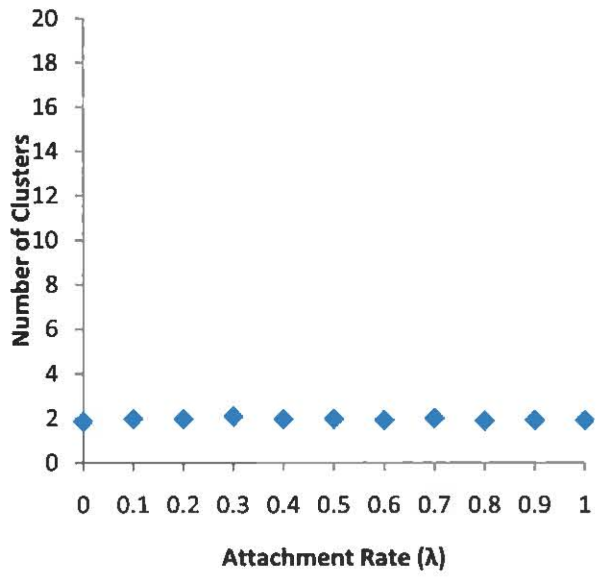


Figure 5.43. Dependence of Average Number of Clusters on (a) varying Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 1 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

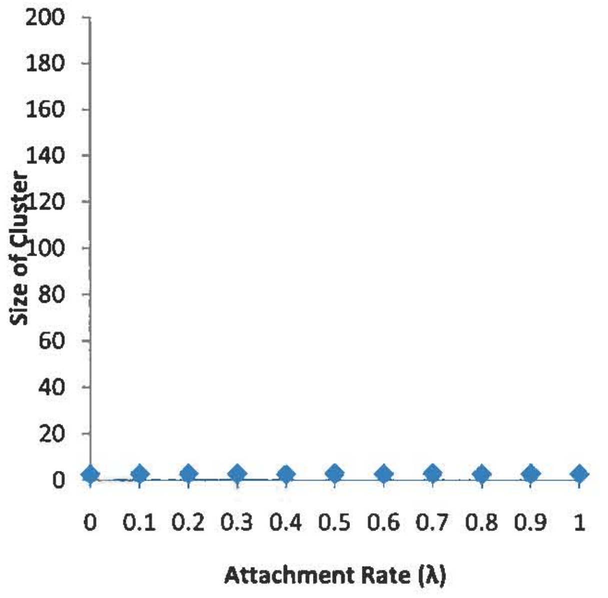


Figure 5.44. Dependency of Average Size of Biggest Cluster on (a) Attachment Rate (λ) for (b) lattice size L = 100 (c) MCS = 500,000 (d) τ = 1 (e) γ = 0.5 (f) β = α = δ = μ = 0.5

5.5 Summary

Motivated by the interesting connections between non-equilibrium statistical physics and molecular motors, we have presented a model of a two-lane driven diffusive system that retains some of the characteristics of the motion of the kinesin and dynein molecular motors on cellular tracks. Using Monte Carlo simulations we studied the formation of clusters in the system as a function of the attachment rate λ and detachment rate τ of particles to the lattice. We have concluded that the effect of the particle exchange rate γ, which is important to the coarsening phenomenon in two-lane systems with boundary conditions, is relatively weak when compared to the effects of the attachment and detachment rates. The results presented above for a γ value of .5 can be seen across all γ values. If we let the τ vary we see that for ever increasing values of λ the average number of clusters moves towards a normal distribution with a peak value of twenty clusters at $\tau = 0.5$. At the same time the size of the biggest cluster in the system exponentially decays as τ increases no matter the λ value. When we fix τ and let λ vary we notice that the average number of clusters evolves into a logarithmic growth function at $\tau = 0.5$ with a limit of twenty clusters. The number of clusters for values of τ greater than 0.5 keep their logarithmic form but the limit decreases until reaching a minimum at a value of two. The average size of the biggest cluster is also described by a logarithmic growth function with a maximum value of 200 when $\tau = 0$. As τ increases the maximum value of the logarithmic function decreases exponentially, eventually becoming linear at a minimum value of about 2.6 when $\tau = 1$. This study highlights the importance of the relationship between attachment and detachment rates to the overall behavior of two-lane driven diffusive systems and can serve as an important guide in the search for an analytical solution to such a system. Future research could investigate the effects of entrance and exit rates on the behavior of such a system. We would also like to suggest the possibility of applying concepts from factorial design to the study of driven diffusive systems. Such an approach could help pinpoint the variables and relationships between variables that matter most to the behavior of the system.

References

[1] K. Svoboda and S.M. Block, *Cell* 77:773 (1994).

[2] K. Svoboda, P.P. Mitra and S.M. Blockthor, *Proc. Natl. Acad.* USA 99:6696 (2002).

[3] H. Kojima, E. Muto, H. Higuchi and T. Yanagida, *Biophys. J.* 73:2012 (1997).

[4] Mazilu I., Schmittmann B., *Journal of Statistical Physics*, **113** (3/4): 505 (2003).

[5] I. Mazilu, *Steady state properties of some driven diffusive systems*, PhD theses, (Virginia Tech, Blacksburg, 2002) http://scholar.lib.vt.edu/theses/available/etd-09022002-101837.

[6] J. W. Gibbs *The Elementary Principles in Statistical Mechanics,* (Scribner, N.Y, 1902).

[7] L. Boltzmann *Lectures on Gas Theory*, English translation (Berkeley, California, 1964).

[8] R. K. Pathria Statistical Mechanics 2nd ed. (Butterworth-Heinemann, Oxford, 1996).

[9] B. Schmittmann and R. K. P. Zia *Phase Transitions and Critical Phenomena* Vol. 17, edited by C. Domb and J. L. Lebowitz (Academic, London, 1995).

[10] Philip Nelson *Biological Physics* (W. H. Freeman and Company, N.Y., 2004).

[11] H. Higuchi, E. Muto, Y. Inoue, and T. Yanagida, *Proc.Natl. Acad. Sci.* USA **94**:4395 (1997).

[12] C.M. Coppin, D.W. Pierce, L. Hsu and R.D.Yale *Proc.Natl. Acad. Sci.* USA **94**:8539 (1997).

[13] M.D. Wang, M.J. Schnitzer, H. Yin, R. Landick, J. Gelles and S.M. Block, *Science* **282**: 902 (1998).

[14] Angelika Krebs, Kenneth N. Goldie and Andreas Hoenger, ***J. of Molec. Bio.*** **, 335 (1): 139 (2004).**

[15] Howard J., Mechanics of Motor Proteins and the Cytoskeleton (Sinauer Associates, Sunderland, 2001).

[16] E. Levine and R. D. Willmann *J. Phys. A: Math. Gen* **37**:3333 (2004).

[17] Aridor M, Hannan LA. *Traffic Jam: A compendium of human diseases that affect intracellular transport processes.* Traffic 2000; **1**: 836-851.

[18] D. Mukamel in *Soft and Fragile Matter: Non-equilibrium Dynamics, Metastability and Flow*, Eds. M.E. Cates and M.R.Evans (Institute of Physics Publishing, Bristol, 2000).

[19] M. R. Evans and R. A. Blythe, *Physica A* **313**:110 (2002).

[20] M.R. Evans, T. Hanney and Y. Kafri *Disorder and non-conservation in a driven diffusive system* cond-mat/0405611 (2004).

[21] Hans C. Fogedby, Ralf Metzler and Axel Svane *Exact solution of a linear molecular motor model driven by two-step fluctuations and subject to protein friction* cond-mat/0312364 (2003).

[22] Rakos, M. Paessens and G.M. Schütz, *Phys. Rev. Lett.*, **91** (2003).

[23] Michael E. Fisher and Anatoly B. Kolomeisky *Proc.Natl. Acad. Sci* USA (1999).

[24] A. Parmeggiani *et al. Europhys. Lett.* **56**: 603 (2001).

[25] V. Popkov *et al. Phys. Rev. E* **67**:066117 (2003).

[26] T. M. Nieuwenhuizen, Stefan Klumpp and Reinhard Lipowsky *Random walks of molecular motors arising from diffusional encounters with immobilized filaments* cond-mat/ 0312422 (2003).

[27] Stefan Klumpp and Reinhard Lipowsky *Phase transitions in systems with two species of molecular motors* cond-mat/ 0402195 (2004).

[28] R. Thornton, R. Tinker "Constructing Student Knowledge in Science", Chapter in *New Directions in Educational Technology,* E. Scanlon and T O'Shea, eds. (Berlin-Heidelberg-New York, Springer-Verlag, NATO ASI Series F: Advanced Educational Technology **96**, pp. 153-171. Series, 1992).

[29] Howard J., *Nature*, **389**:561 (1997).

[30] Donald T. Haynie, *Biological thermodynamics* (Cambridge University Press, 2001).

[31] C.R. Bagshaw *Muscle Contraction* (Chapman & Hall, London, 1993).

[32] K. Binder and D. W. Heermann *Monte Carlo Simulation in Statistical Physics* (Springer, Berlin, 1988).

[33] B. Schmittmann, K. Hwang and R.K.P. Zia *Europhys. Lett.* **19**, pp. 19-25 (1992).

[34] I. Vilfan, R.K.P. Zia, and B. Schmittmann, *Physical Review Letters* 73, pp. 2071-2074 (1994).

[35] F H Jafarpour, F E Ghafari and S R Masharian, *J. Phys. A: Math. Gen.* **38**: 4579-4588 (2005).

CHAPTER 6
STOCHASTIC MODEL OF MICROTUBULE DYNAMICS

6.1 Introduction

Stochastic processes are omnipresent in fields as diverse as physics, biology and economics. "Birth-and-death" processes, or "generation-recombination" processes, also known as "one-step processes" [1], play an important role in modeling various systems, such as photon emission or absorption, chemical reactions, population dynamics. "Random walks" are probably one of the best known one-step processes, with far reaching applications. The random walk problem is well studied, but depending on the question posed, this topic continues to present new and interesting puzzles, and to invite new applications outside of the physics area.

In this paper, we use random walk theory to model the length dynamics of microtubules, one of the principal components of the cytoskeleton [2]. These polar, linear polymers have two major roles in the cell: they form a rigid internal skeleton for some cells and they also act as cellular tracks along which motor proteins can move structures within the cell. These tracks can either grow out from the centrosome towards the periphery of the cell or can be free. Microtubules are dynamic structures that undergo continuous assembly and disassembly within the cell. These polar structures have a fast-growing plus end and a slow-growing minus end. They remain in a non-equilibrium state driven by sudden polymerization changes, switching stochastically from growing to shrinking and vice versa. Microtubules grow by the attachment of GTP (guanosine triphosphate tubulin units) at their plus end. However, if the GTP cap hydrolyzes into GDP (guanosine diphosphate tubulin units), the microtubule is destabilized, the GDP complexes are detached and shrinkage ensues. Another interesting aspect of microtubule dynamics is "treadmilling" [3], when free microtubules display persistent growth at one end and shrinking from the other end.

From the point of view of statistical physics, microtubules represent interesting non-equilibrium systems, amenable to a stochastic treatment. In 1984, Mitchison and Kirschner [4] discovered the "dynamic instability" of microtubules (the random transition between states of assembly and disassembly). Mitchison and Kirschner conjectured that this instability is a consequence of competition between assembly and GTP hydrolysis; once the stabilizing GTP cap is gone, the microtubule dissociates rapidly (process known in the literature as "the microtubule catastrophe"). Despite the large number of theoretical and experimental studies [5-9], there still is no coherent model for the microtubule dynamics able to explain the experimental data fully. On the theoretical front, a variety of models have been proposed: very detailed ones that include all 13 protofilaments that make up the microtubule [10,11] and minimalist models that contain as few parameters as possible, such as the ones proposed by Flyvbjerg et al. [12] and Antal et al. [13].

This paper presents a two-state model of microtubule length dynamics that incorporates a variable rate of switching between a growing and shrinking microtubule. This is a novel approach—so far, the theoretical models proposed assumed constant rates of transition between states. We do not account for stochastic avalanches or catastrophes that occur as part of in vivo and in vitro microtubule experiments. Specifically, based on some simple dynamical rules first introduced in Ref. [13], in the context of our model, we calculate the mean length of the microtubule as a function of time, its variance and the diffusion coefficient of the microtubule tip. We compare our analytical and computational results with experimental results reported in Ref. [4], and find good qualitative agreement.

This paper is structured as follows: We first (Section 2) give an overview of "one-step processes", and define our model. Using the generating function technique, we solve the master equation for some particular cases to obtain the probability distribution for the lengths, the average length and the diffusion coefficient (Sections 3 and 4). In Section 5 we discuss the

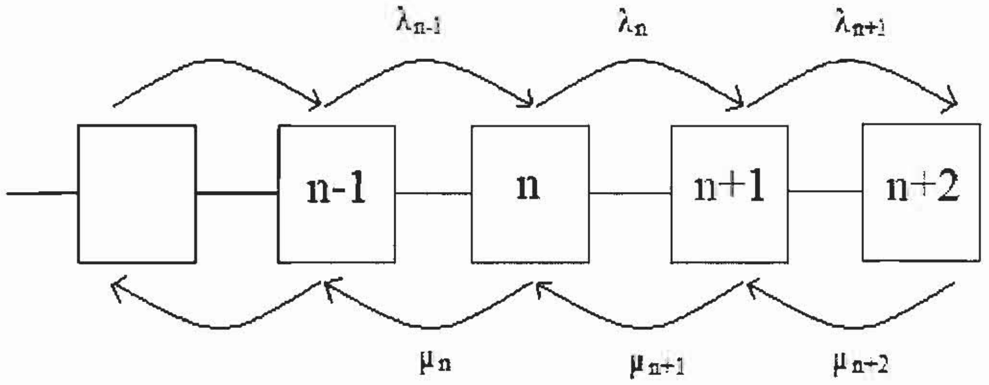


Fig. 6.1. General definition of a one-step process and its transition probabilities. The process is defined as a Markov process on an integer set $\{n\}$ with jumps allowed only between adjacent sites. Adapted from [1].

distribution of positive monomers for the case of variable rates. We compare our analytical results with the computer simulations and the experimental biological data and conclude with a summary of our work and some open questions.

6.2 Analytic Model

A starting point for the study of one-step processes is the master equation, which expresses the conservation of configurational probabilities. In general, consider a system in state "r" at time "t". P_r (t) is the probability that this system is in this particular state. The time dependence of P_r is given by the master equation, which states:

$$\frac{dP_n}{dt} = \sum_s P_s W_{sr} - \sum_s P_r W_{rs} \tag{1}$$

This is a balance (continuity) equation. The probability of state "r" increases with time due to states that evolve into state "r" and it decreases with time because of transitions from state "r" to other states. In this equation, W stands for the transition rates to and from state"r". Knowing W allows one to calculate all probabilities P_r as a function of time.

For the special case of one-step processes, transitions happen only between adjacent states, labeled as a set of integers "n" (Fig. 1). The evolution of probability $p_n(t)$ is described by the following master equation:

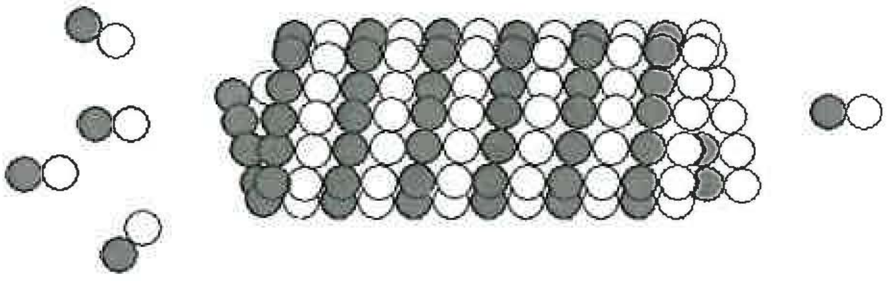
$$\frac{dp_n}{dt} = \mu_{n+1}p_{n+1} + \lambda_{n-1}p_{n-1} - (\mu_n + \lambda_n)p_n \tag{2}$$

where μ_n is the probability per unit time of a transition from state "n" to state "$n-1$",and λ_n is the probability per unit time of a transition from state "n" to state "$n+1$" (see Fig. 1). These quantities can be constants (this is the classical random walk case), or polynomials in n.

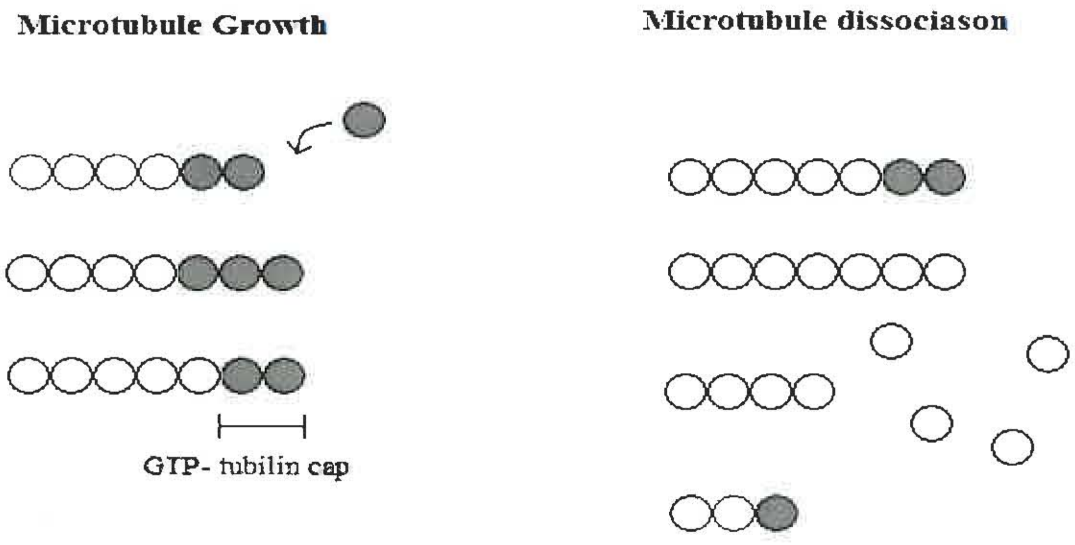
In the context of one-step processes, we define the following idealized model for the microtubule (Fig. 2), based on a theoretical model introduced by Ref. [13]. Using Antal et al. notation, treat the microtubule as an ordered set of GTP ("$+$") and GDP ("$-$") monomers. The microtubule evolves according to the following rules:

- Attachment. A microtubule grows by attachment of a guanosine triphosphate tubulin unit (GTP+ monomer) at either end. We define λ_1 to be the attachment rate at the left (negative) end of the microtubule, and λ_r the corresponding attachment rate at the right (positive) end of the microtubule.
- Detachment. A microtubule shrinks by detachment of a guanosine diphosphate tubulin unit (GDP− monomer) at either ends. We define μ_1 to be the detachment rate at the left (negative) end of the microtubule, and μ_r the corresponding detachment rate at the right (positive) end of the microtubule.
- Conversion. Each GTP+ monomer can convert via hydrolysis into a GDP− monomer. We assume this rate to be 1.

(a)



(b)



(c)

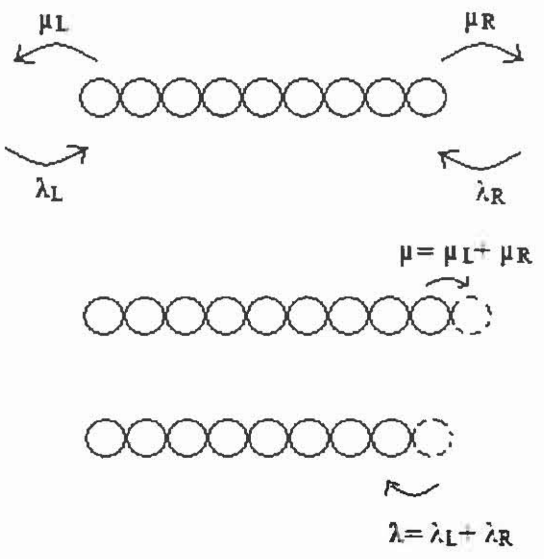


Fig.6.2. (a) Standard sketch of the complex 13 filament structure of microtubules; (b) Idealized model of microtubule assembly and disassembly; (c) Our model-cartoon of the *overall* microtubule growth and shrinkage. Rates λ and μ are variable.

Note that this is a general model that includes microtubule growth and shrinkage at both ends. In reality, the microtubule is polarized. GTP$^+$ units tend to attach to its " + " end. Since we monitor the overall length of the microtubule, the identity of the polarized ends becomes irrelevant. Therefore, our model applies to both attached and free microtubules.

Note the following connection between the length dynamics of the microtubule and one-step processes: at each time step, the length of the microtubule can increase or decrease by one unit, due to tubulin attachment or detachment. The growth or shrinking processes can happen at both ends. If L, the length of the microtubule, is our variable, it will increase or decrease by one. We want to calculate the probability distribution of microtubule lengths, as well as the average length under various scenarios. We start by writing the master equation for the length probability:

$$\frac{dP(L,t)}{dt} = -[\lambda_l + \lambda_r + \mu_l + \mu_r]P(L,t) + [\lambda_l + \lambda_r]P(L-1,t) + [\mu_l + \mu_r]P(L+1,t) \quad (3)$$

$P(L,t)$ represents the probability of the microtubule to have a certain length L at time t. In the master equation, the loss term $-[\lambda_l + \lambda_r + \mu_l + \mu_r]P(L,t)$ accounts for the change in length due to growth and shrinkage of the microtubule at both ends, with the appropriate rates. The gain terms take into consideration the configurations that evolve into the present configuration due to attachment $[\lambda_l + \lambda_r]P(L-1,t)$ and detachment $[\mu_l + \mu_r]P(L+1,t)$. The conversion of the GDP+ into GTP− . does not play a role in the change of the microtubule length.

We define $\lambda = \lambda_l + \lambda_r$ as the overall growth probability and $\mu = \mu_l + \mu_r$ as the overall shrinkage probability. The theory applies to λ and μ as general functions of the length L. In reality, this is still an approximation, since the attachment and detachment probabilities depend on other factors, such as the identity of the ends, temperature or concentration of tubulins in the surrounding solution. In our minimalist model, we do not differentiate between the two, and look only at the overall increase or decrease of microtubule length.

Given the master equation, we first find the probability distribution of microtubule length for constant attachment and detachment rates. Then, we generalize the model to include rates that are linear functions of L. Alongside the analytical work, we also present the Monte Carlo simulation results, and when appropriate, compare our study with the reported experimental data.

6.3 Probability Distribution of Microtubule Length for Constant Attachment and Detachment Rates

We review the statement of the simple random walk problem in one dimension: a drunkard, starting his journey at a lamp post on the street, can step either to the right or left, with probability p and $1-p$, respectively. Each step is of equal length (let us assume $l = 1$) and is independent of the previous one.

We first consider our model with λ and μ being constants. Just like our random walker, the microtubule can be in two states: growth with probability λ (analogous to the walker moving right with probability p), or shrinkage with probability μ (analogous to the walker moving left with probability $1-p$). Our goal is to calculate $P(L,t)$, the probability of microtubule to have length "L" at time "t", $\langle L \rangle$, mean microtubule length, and $D = \frac{\langle L^2 \rangle - \langle L \rangle^2}{2t}$ define in the literature as the diffusion coefficient of the microtubule tip.

In our calculations, we use the generating function method, sketched below [1]:

(1) Define the generating function as: $f(z,t) = \sum_{L=-\infty}^{\infty} P_L z^L$, with z an auxiliary variable. Some properties of this generating function are[1]:

- Since: $\sum_L P_L(t) = 1$ and $P_L(t) \geq 0$, this generating function exists for $|z| = 1$.
- Also, $f(1,t) = 1, f'(1,t) = \langle L(t)\rangle, f''(1,t) = \langle L(t)^2\rangle - \langle L(t)\rangle$.

(2) Write the master equation in terms of the generating function, and solve it with the appropriate initial conditions.

(3) If the generating function is known, its first and second order derivatives give us the mean length and the diffusion coefficient. Also, by expanding the generating function in a power series in z, one can recover the probability length distribution $P(L,t)$.

6.3.1 Unrestricted Growth

We first assume that the microtubule can only grow with overall rate λ:

In this case, the master equation becomes:

$$\frac{dP(L,t)}{dt} = -\lambda P(L,t) + \lambda P(L-1,t) = \lambda(P(L-1,t) - P(L,t)). \tag{4}$$

This is known as the Poisson process, with the initial condition$P(L,0) = \delta_{L,0}$. This condition is sufficient and it also covers the case $P(L,t) = \delta_{L,L_0}$ for any initial length.

In terms of the generating function, the master equation becomes:

$$\frac{df(z,t)}{dt} = -\lambda f(z,t) + \lambda z f(z,t). \tag{5}$$

For the initial condition $P(L,t) = \delta_{L,L_0}$, which translates into $f(z,t=0) = 1$, the solutions is the Poisson distribution:

$$P(L,t) = \frac{[\lambda t]^L}{L!} e^{-\lambda t}. \tag{6}$$

The average length and its variance are: $\langle L(t)\rangle = \lambda t, \langle L(t)^2\rangle - \langle L(t)\rangle^2 = \lambda t$. This gives us also the diffusion coefficient to be: $D = \frac{\lambda}{2}$ (results also reported in Ref. [13]).

6.3.2 Asymmetrical Random Walk: $\lambda = \mu$

We take the model one step further, and assume the presence of dissociation processes as well. In analogy with the random walk problem in one dimension, now the microtubule can grow or shrink with different rates (from a biological point of view, we assume that the system constantly shifts in between polymerization and de-polymerization regimes). This is the case of the asymmetrical random walk with continuous time.

The master equation becomes:

$$\frac{dP(L,t)}{dt} = -(\lambda + \mu)P(L,t) + \lambda P(L-1,t) + \mu P(L+1,t). \tag{7}$$

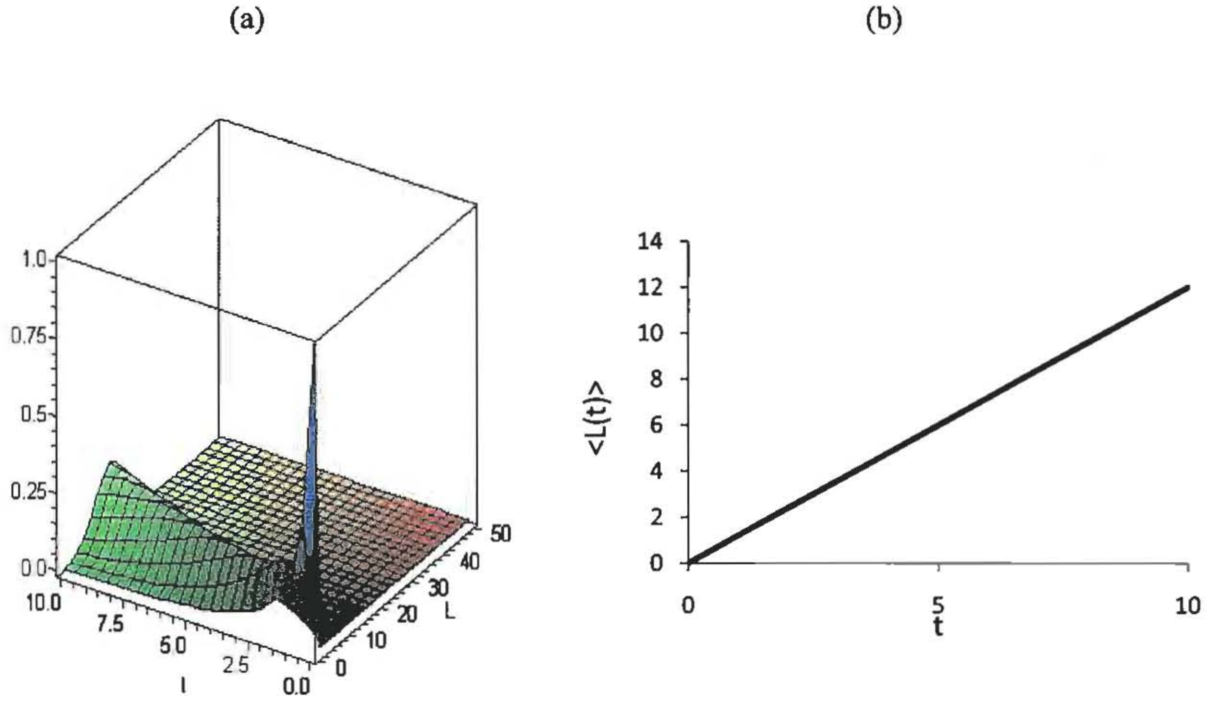


Fig. 6.3. (a) Probability length distribution $P(L,t)$ for constant rates $\lambda = 1.6$ and $\mu = 0.4$ (growth regime); (b) Mean length $\langle L(t)\rangle = (\lambda - \mu)t$, for $\lambda = 1.6$ and $\mu = 0.4$ (growth regime)

We use again the generating function technique. Define: $f(z,t) = \sum P_L\, z^L$, with z being an auxiliary variable. In terms of $f(z)$, the master equation becomes:

$$\frac{\partial f(z,t)}{\partial t} = (1-z)(\frac{\mu}{z} - \lambda) f(z,t) \tag{8}$$

with the general solution:

$$f(z,t) = \Omega(z) e^{\frac{-t(z-1)(\mu-\lambda z)}{z}} \tag{9}$$

where $\Omega(z)$ is an arbitrary function of z. Given the initial condition: $f(z,0) = 1$ ($\Omega(z) = 1$).

Our generating function is:

$$f(z,t) = e^{\frac{-t(z-1)(\mu-\lambda z)}{z}}. \tag{10}$$

From the generating function, by taking the first and second derivatives and evaluating it at $z = 1$, we obtain:

- $\langle L(t) \rangle = (\lambda - \mu)t$
- $\langle L(t)^2 \rangle = (\lambda - \mu)^2 t^2 + (\lambda + \mu)t$
- Variance: $\langle L(t)^2 \rangle - \langle L(t) \rangle^2 = (\lambda + \mu)t$
- Diffusion coefficient: $D = \frac{\lambda+\mu}{2}$.

We also recover the probability length distribution for microtubules:

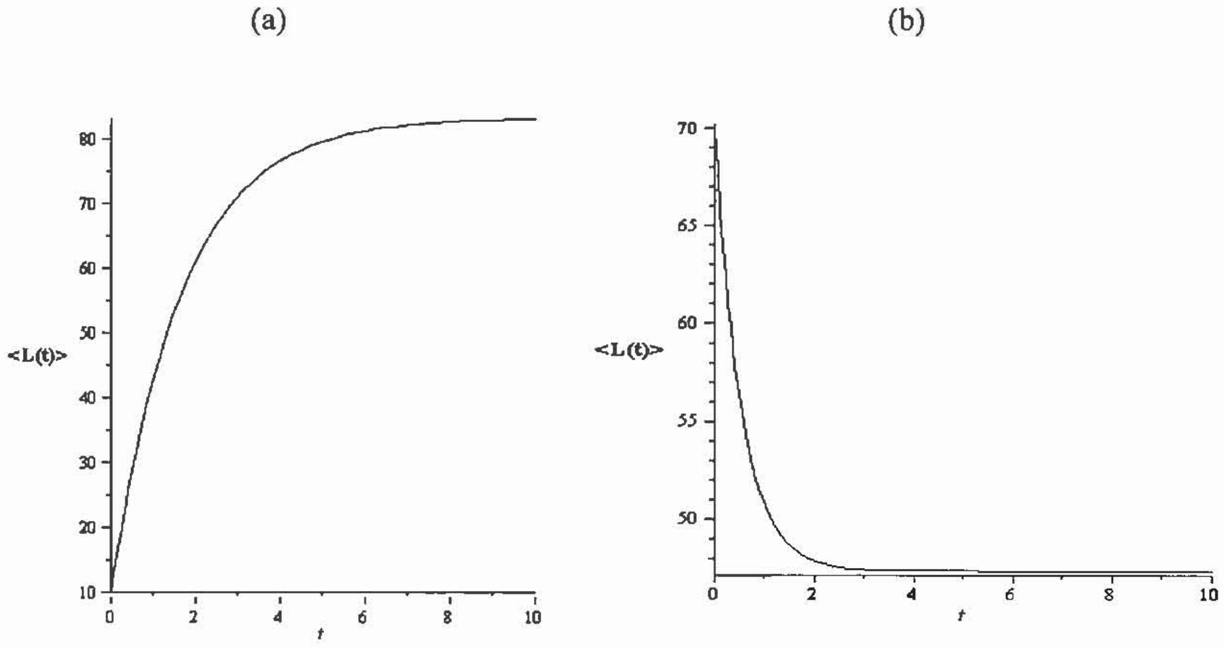
$$P_L(t) = e^{-2t} \sum_l \frac{t^{2l+L}}{(l+L)!\, l!} (\frac{\lambda}{\mu})^{\frac{L}{2}} e^{-(\mu+\lambda-2\sqrt{\mu\lambda})t} \tag{11}$$

where $\sum_l \frac{t^{2l+L}}{(l+L)!l!}$ is $I_{|L|}(2t)$, the $|L|th$ modified Bessel function.

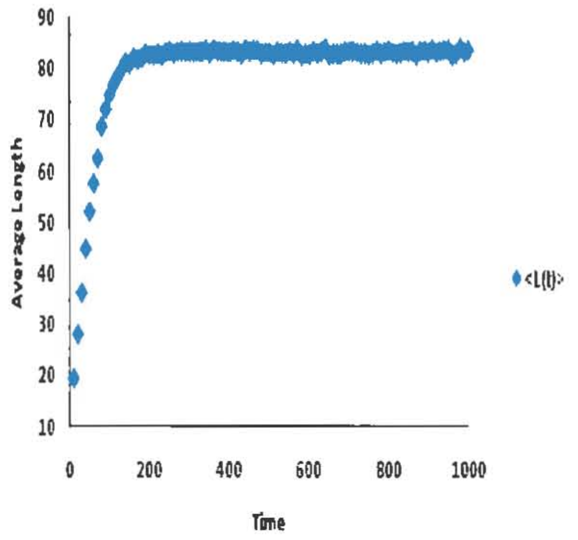
These analytical results are known from the one-dimensional asymmetric random walk theory. In Fig. 3 we can see the plotted analytical solutions for $\langle L \rangle$ and $P(L,t)$. The qualitative agreement with Mitchison and Kirschner's data (Fig. 5(a)) for higher tubulin concentrations is quite good. (Their data plots are sketched in Fig. 5, also can be found in Ref. [4]).

6.4 Mean Microtubule Length for Variable Attachment and Detachment Rates

Microtubule evolution is a complex process, sensitive to external factors such as temperature and solution concentration of tubulins. The polymerization and dissociation rates are in general not constant. We consider a case when the attachment and detachment probability rates depend linearly on the length of the microtubule. From a physical point of view, the length of the microtubule cannot become negative, and also it cannot exceed a certain physical maximum length. We will incorporate these constraints in our analysis and treat this problem as a one-step process with natural boundaries[1].



(c) (d)



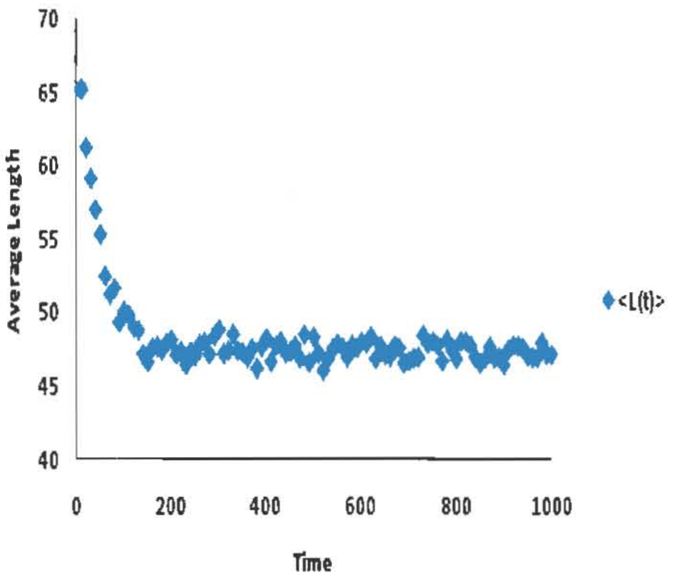


Fig. 6.4. (a) Mean length $\langle L(t)\rangle$, for variable rates with $\alpha = 0.1$ and $\beta = 0.5, L_0 = 10, L_{max} = 100$ (growth regime). (b) Mean length$\langle L(t)\rangle$, for variable rates with $\alpha = 0.9$ and $\beta = 1, L_0 = 70, L_{max} = 90$ (shrink regime); (c),(d) Associate Monte Carlo results; time unit is 100 MCS.

Specifically, we now consider the growth and shrinkage rates as linear functions of the microtubule length. The micro-tubule growth rate decreases with the size of the system, and it becomes zero when the microtubule reaches its maximum size L_{max} (the growth stops). The shrinkage rate increases with L. Based on the previous notations, we now have:

$$\lambda = \beta(L_{max} - L) \tag{12}$$

$$\mu = \alpha L \tag{13}$$

where α and β are both positive constants.

The master equation now becomes:

$$\frac{dP(L,t)}{dt} = -(\lambda(L) + \mu(L))P(L,t) + \lambda(L)P(L-1,t) + \mu(L)P(L+1,t). \tag{14}$$

Substituting the expressions for $\lambda(L)$ and $\mu(L)$,

$$\frac{dP(L,t)}{dt} = -((\alpha - \beta)L + \beta L_{max})P(L,t) + \beta(L_{max} - L)P(L-1,t) + \alpha LP(L+1,t). \quad (15)$$

Following again the method presented in [1], we use the generating function: $f(z,t) = \sum_{L=-\infty} P_L z^L$, with z being an auxiliary variable. We consider the initial conditions to be $P(L,0) = \delta_{L,L_0}$.

In terms of $f(z)$, the master equation becomes:

$$\frac{\partial f(z,t)}{\partial t} = (1-z)(\alpha + \beta z)\frac{\partial f(z,t)}{\partial z} - \beta L_{max}(1-z)f(z,t). \quad (16)$$

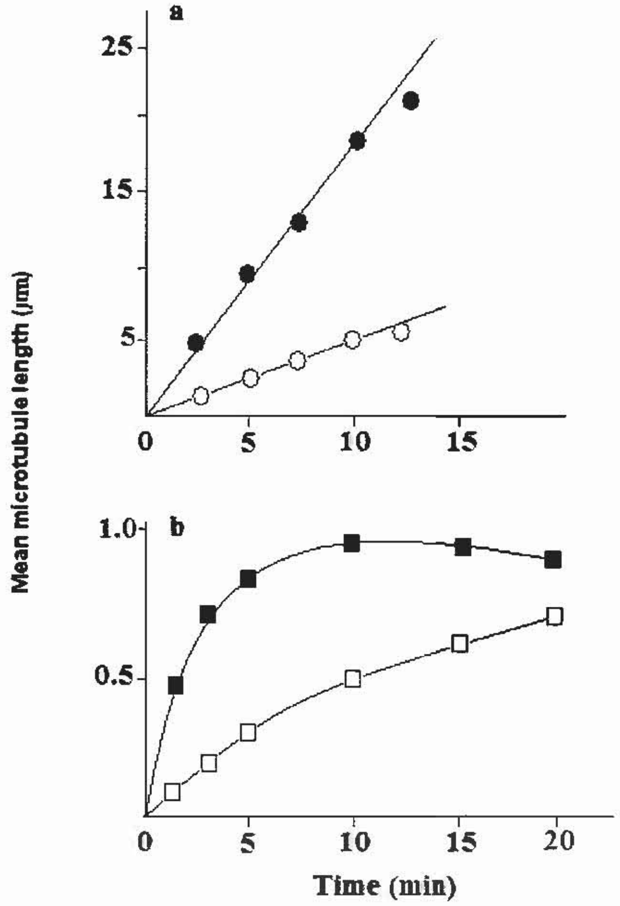


Fig. 6.5. Experimental data (sketch reproduced from Mitchison et al.[4]) (a) Polymerization at 15μM (immunofluorescence method); (b) Polymerization at 3μM (electron microscopy method). The closed symbols show data for plus end, the open symbols show data for the minus end of the microtubule.

For the finite range $0 \le L \le L_{max}$, with α and β positive, the generating function is (more detailed calculations of a related problem are presented in [1]):

$$f(z,t) = (\alpha+\beta)^{-L_{max}}[\alpha(1-\epsilon)+(\alpha\epsilon+\beta)z]^{L_0}[(\alpha+\beta\epsilon+\beta(1-\epsilon)z)]^{L_{max}-L_0} \tag{17}$$

where $\epsilon = e^{-(\alpha+\beta)t}$, and with the initial condition: $P_L(0) = \delta_{L,L_0}$.

Given this generating function, we can extract information about the average length of the microtubule as a function of time:

$$\langle L(t)\rangle = \frac{L_0\alpha\epsilon + \beta L_{max}(1-\epsilon) + \beta L_0\epsilon}{\alpha+\beta} \tag{18}$$

where $\epsilon = e^{-(\alpha+\beta)t}$.

Knowing the generating function, one can also calculate the variance and diffusion coefficient. In Fig. 4 we plot $\langle L\rangle$ for the two regimes (a) overall growth and (b) overall shrinkage. We show for comparison, the analytical results and the Monte Carlo simulation results. If we compare these results with Fig. 5(b)(lower concentrations of tubulin in surrounding solution) in Mitchison's paper we notice, again, a good qualitative agreement.

6.5 Distribution of Positive Monomers

We mentioned earlier that the length of the microtubule can change only due to the addition of GTP+ units. For the special case of the unrestricted growth with constant λ, the mean number of positive monomers evolves according to ([13]):

$$\langle N\rangle = \lambda(1-e^{-t}) \tag{19}$$

For the more general case of unrestricted growth but with variable λ, the problem is rather complex. We start, again, with the master equation:

$$\frac{dP(L,N,t)}{dt} = -(\lambda(L)+N)P(L,N,t) + \lambda(L)P(L-1,t) + (N+1)P(L,N+1). \tag{20}$$

Because of the L dependence of λ, the probability distribution of the number of positive monomers is also dependent on length. We can think of this as a joint distribution $P(L, N, t)$ of a microtubule to have length L and a number of monomers N.

The right-hand side of the master equation has "gain terms" that account for all states that can create a microtubule of length L with N positive monomers, either through a positive monomer addition to a microtubule of length $L - 1(\lambda(L)P(L-1,t))$or via a conversion of a positive monomer into a negative one $((N+1)P(L, N+1))$. Likewise, the "loss" term $(-(\lambda(L) + N)P(L, N, t))$ describes the evolution of the microtubule of length L with N positive monomers into other states via growth or conversion. Substituting λ, we arrive at the following form of the master equation:

$$\frac{dP(L,N,t)}{dt} = -(\beta(L_{max} - L) + N)P(L,N,t) + \beta(L_{max} - L)P(L-1,t) + (N+1)P(L,N+1) \quad (21)$$

In order to use, again, the generating function technique, a two-variable generating function is needed:

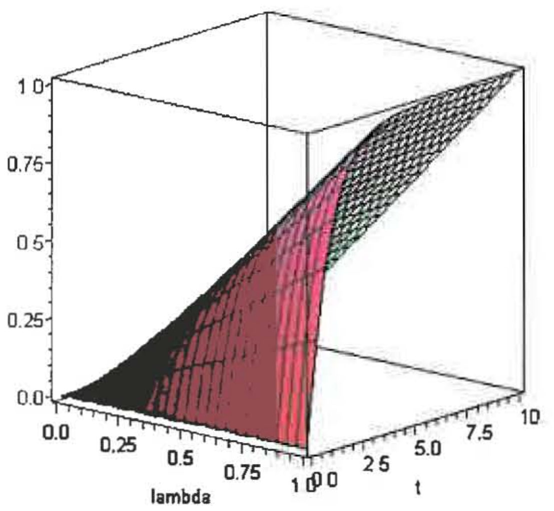
$$f(x,y,t) = \sum_{N \leq L \leq 0} x^L y^N P(L,N,t). \quad (22)$$

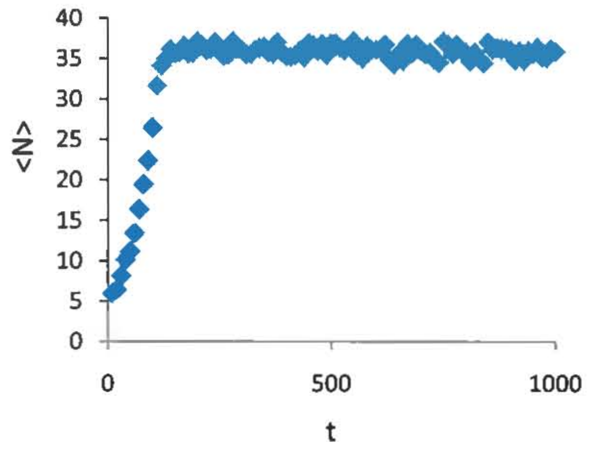
The master equation is re-written in terms of this new generating function as:

$$\frac{\partial}{\partial t} f(x,y,t) = -\frac{\partial}{\partial y} f(x,y,t) + \beta(-L_{max} + xy(L_{max} - 1))f(x,y,t) + \beta x(1 - xy)\frac{\partial}{\partial x} f(x,y,t). \quad (23)$$

(a)

(b)





(c)

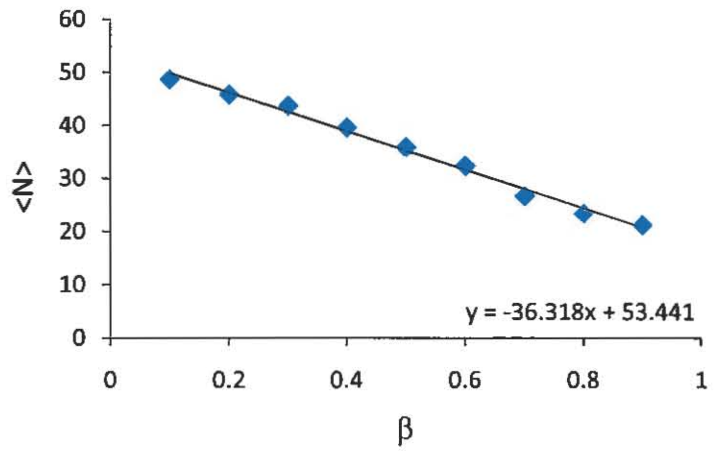


Fig. 6.6 Mean number of positive monomers; (a) For constant rates $\langle N\rangle = \lambda(1-e^{-t})$, (b) Monte Carlo simulation results for variable rate $\lambda = \beta(L_{max} - L)$ with $L_0 = 10, L_{max} = 100$ (growth only), (c) Dependence of the average maximum number of monomers N_{max} on β.

Using Maple or Mathematica, one can solve this equation with the initial conditions $P(L, N, 0) = \delta_{L,L_0}\delta_{N,N_0}$. Unfortunately, the analytical solution is quite complicated, and we were not able to extract useful information regarding the average number of monomers or their probability distribution. We relied on Monte Carlo simulations instead. For a constant growth rate, in the asymptotic case, $N \to \lambda$. For λ variable, in the limit $t \to \infty$, N plateaus as well (see Fig. 6). In

Fig. 6(b) we see the linear relationship between the maximum number of GTP$^+$ and β. For a given set of parameters ($L_{max} = 100, L_0 = 10$), the plateau value decreases as β increases.

6.6 Summary

We investigated a simple two-state model of a free microtubule that switches between growth and shrinking phases by attachment/detachment of GTP$^+$ units. Our main focus was to find the time dependence of the mean microtubule length, its variance and the diffusion coefficient, and to compare it with computer simulation results and experimental data. We first made the analogy between one-dimensional random walks and the microtubule dynamics, in the case of constant growth and shrinkage rates. We then extended our study for rates that are linear functions of L. We also analyzed the distribution of positive monomers for the growth only case with variable rates. Monte Carlo simulations seem to suggest that in the long run the average number of GTP$^+$ settles on a plateau, just like for the constant rate situation. However, the value of this plateau changes (decreases) as our parameter β increases - a novel feature compared to the constant rates' case.

Although some of our analytical results are in good qualitative agreement with the experiments, we should point out that this is a simple model that does not include the complex range of experimental parameters that influence the evolution of microtubules. A new avenue of research would be to address the issue of the GTP cap and dynamic instability in the context of this model with variable switching rates between polymerization and de-polymerization states.

References

[1] N.G. Van Kampen, Stochastic Processes in Physics and Chemistry, 3rd ed., Elsevier, 2002.

[2] Lodish, et al., Molecular Cell Biology, 5th ed., W.H. Freeman and Company, NY, 2004.

[3] D. Panda, H.P. Miller, L. Wilson, PNAS 96 (1999) 12459.

[4] T. Mitchison, M. Kirschner, Nature (London) 312 (1984) 237.

[5] M. Dogterom, S. Leibler, Phys. Rev. Lett. 70 (1993) 1347.

[6] D.J. Bicout, Phys. Rev. E 56 (1997) 6656.

[7] D.J. Bicout, R.J. Rubin, Phys. Rev. E 59 (1999) 913.

[8] T.L. Hill, Y. Chen, PNAS 81 (1984) 5772.

[9] S. Grego, V. Cantillana, E.D. Salmon, Biophys. J. 81 (2001) 66.

[10] V. VanBuren, D.J. Odde, L. Cassimeris, PNAS 99 (2002) 6035.

[11] V. VanBuren, L. Cassimeris, D.J. Odde, Biophys. J. 89 (2005) 2911.

[12] H. Flyvbjerg,T.E. Holy, S. Leibler, Phys. Rev. Lett. 73 (1994) 2372.

[13] T. Antal, P.L. Krapivsky, S. Redner, M. Mailman, B. Charkraborty, Phys. Rev. E 76 (2007) 41907.

```
/**********************************************************/
//    2D_main.cpp
//
// This file contains run instructions for instantiation
// and execution of two-lane driven diffusive lattice
// Monte Carlo simulation.
//
//
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/**********************************************************/

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cmath>
#include <time.h>
#include <fstream>
#include <string.h>

#include "2D_param.h"
#include "Lattice_2D.h"
// #include "2D_simulation.h"

using namespace std;

/**********************************************************/
//        Global Variables
/**********************************************************/

/**********************************************************/
//        Code
/**********************************************************/

int main(int argc, char *argv[])
{

        int j;
        double i;
        double interval;
        i = 0;


        SimConfInfo sim_running_conf;
  Lattice_2D* latticePtr = 0;

        ZERO(&sim_running_conf.result_file_name);

  // set default simulation configurattion
  set_default_sim_conf();
  if ( parse_param(argc,argv) < 0 )
    return -1;
  copy_param_to_running_conf(&sim_running_conf);
  print_running_conf(&sim_running_conf);
```

```
        latticePtr = new Lattice_2D(&sim_running_conf);
        if ( latticePtr == 0 ) {
                // log error in creation of myLattice
                cout << "ERROR: unable to create myLattice ... exit program" << endl;
                return -1;
        }

        if (!latticePtr->initialize())
                // initialization failed, for now exit
                exit(-1);

#ifdef _TESTMODE
        latticePtr->runTest();
        cout << "Done with tests" << endl;
#endif

        interval = (sim_running_conf.numSteps)/20;

        for (j=0; j < sim_running_conf.numSteps; j++)
        {
                if(i == interval)
                {
                        latticePtr->snapshot();
                        i = 0;
                }
                latticePtr->runMonteCarloStep();
                latticePtr->exit_ends();
                latticePtr->enter();
                latticePtr->attach();
                latticePtr->detach();
                i++;
        }

        latticePtr->findClusters();
        latticePtr->resultClusters();
        cout << "Done with simulation" << endl;

        if (latticePtr->findClusters() < 0)
        {
                // log error in creation of new cluster
                cout << "ERROR: failure while finding clusters" << endl;
                return -1;
        }
        delete latticePtr;

  return 0;
}
```

```
/*********************************************************/
//    2D_param.h
//
// This file contains... <add description>
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/*********************************************************/

#ifndef _2D_PARAM_H
#define _2D_PARAM_H

#include <iostream>
#include <cstdlib>

using namespace std;

/*********************************************************/
//
//         Macros
//
/*********************************************************/

#define ZERO(p)        memset((char *)p, 0, sizeof(*p))

/*********************************************************/
//
//         Definitions
//
/*********************************************************/

// Default values for simulation
#define SIM_ROWS_IN_LATTICE        500
#define SIM_COLUMNS_IN_LATTICE     2
#define SIM_NUMBER_TRIALS          1
#define SIM_NUMBER_MONTE_CARLO_STEPS 500000
#define SIM_ATTACH_DETACH_STEP      10
#define SIM_NUMBER_OF_PARTICLES     17
#define SIM_POPULATION_RATE        1
#define SIM_Q1                 0.6
#define SIM_Q2                 0.2
#ifdef _TESTMODE
#define SIM_ATTACH                                   1
#else
#define SIM_ATTACH             0.2
#endif
#ifdef _TESTMODE
#define SIM_DETACH                                   1
#else
#define SIM_DETACH             1
#endif
#define SIM_MAX_SIZE_FILENAME      100
#define SIM_DEFAULT_FILENAME       "2D_result.txt"
#define SIM_RESULT_FILE_DEFAULT            "summary.txt"
#define SIM_ATTEMPT_FLOW_FILENAME   "attemptflow.txt"
```

```
#define  SIM_POSITIVE_FLOW_FILENAME        "positiveflow.txt"
#define SIM_NEGATIVE_FLOW_FILENAME  "negativeflow.txt"
#define SIM_TOTAL_FLOW_FILENAME      "totalflow.txt"
#define SIM_Q3                                          0.2
#define SIM_G                                           0.2
#define SIM_P                                           0.5
#ifdef _TESTMODE
#define SIM_ALPHA_E                             1
#else
#define SIM_ALPHA_E                             0.5
#endif
#ifdef _TESTMODE
#define SIM_BETA_E                                      1
#else
#define SIM_BETA_E                                      0.5
#endif
#ifdef _TESTMODE
#define SIM_GAMMA_EX                    1
#else
#define SIM_GAMMA_EX                                    0.5
#endif
#ifdef _TESTMODE
#define SIM_DELTA_EX                        1
#else
#define SIM_DELTA_EX                        0.5
#endif


/**********************************************************/
//        Typedef
/**********************************************************/

typedef struct sim_conf_info {
   int rows;              // number of rows in the lattice
   int columns;           // number of columns in the lattice
   int numTrials;         // number of trials
   int numSteps;          // number of Monte Carlo steps
   int attach_detach_step;  // after this many steps, there is a chance of attach/detach
   int number_of_particles;  // the initial number of particles in the lattice
   double population_rate;  // initial probability that a particle will be positive
   double q1;            // probability of positive particle to move to the right or of a negative to move left
   double q2;            // probability used for stay put or move opposite direction
   double attach;         // attach rate
   double detach;         // detach rate
   double g;                               // gamma particle charge exchange probability
   double p;                               // populate percentage of lattice
   double alpha_e;                      // enter rate/probability for positivie particles
   double beta_e;                     // enter rate/probability for negative particles
   double gamma_ex;                   // exit rate/probability for positive particles
   double delta_ex;                   // exit rate/probability for negative particles
   char result_file_name[SIM_MAX_SIZE_FILENAME];
   char summary_file_name[SIM_MAX_SIZE_FILENAME];
   char attempt_file_name[SIM_MAX_SIZE_FILENAME];
   char positive_file_name[SIM_MAX_SIZE_FILENAME];
   char negative_file_name[SIM_MAX_SIZE_FILENAME];
         char total_file_name[SIM_MAX_SIZE_FILENAME];
} SimConfInfo;

/***********************************************************/
//        Functions declarations
/***********************************************************/

void set_default_sim_conf(void);
void print_running_conf(SimConfInfo* sim_running_conf);
void copy_param_to_running_conf(SimConfInfo* sim_running_conf);
int parse_param(int argc, char **argv);

/***********************************************************/
//        Do not place anything after endif
/***********************************************************/
#endif 2D_param_h
```

```
/*********************************************************/
//    2D_param.cpp
//
// This file contains execution of default parameter
// configurations for lattice
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/*********************************************************/

#include "2D_param.h"

/*********************************************************/
//        Global Variables
/*********************************************************/

SimConfInfo sim_default_conf;

/*********************************************************/
//        Functions
/*********************************************************/

void set_default_sim_conf(void)
{
   sim_default_conf.rows              = SIM_ROWS_IN_LATTICE;
   sim_default_conf.columns           = SIM_COLUMNS_IN_LATTICE;
   sim_default_conf.numTrials         = SIM_NUMBER_TRIALS;
   sim_default_conf.numSteps          = SIM_NUMBER_MONTE_CARLO_STEPS;
   sim_default_conf.attach_detach_step = SIM_ATTACH_DETACH_STEP;
   sim_default_conf.number_of_particles = SIM_NUMBER_OF_PARTICLES;
   sim_default_conf.population_rate   = SIM_POPULATION_RATE;
   sim_default_conf.q1                = SIM_Q1;
   sim_default_conf.q2                = SIM_Q2;
   sim_default_conf.attach            = SIM_ATTACH ;
   sim_default_conf.detach            = SIM_DETACH;
   sim_default_conf.g                 = SIM_G;
   sim_default_conf.p                 = SIM_P;
   sim_default_conf.alpha_e           = SIM_ALPHA_E;
   sim_default_conf.beta_e            = SIM_BETA_E;
   sim_default_conf.gamma_ex          = SIM_GAMMA_EX;
   sim_default_conf.delta_ex          = SIM_DELTA_EX;

        ZERO(&sim_default_conf.result_file_name);
   strncpy(sim_default_conf.result_file_name,
                          SIM_DEFAULT_FILENAME,
                          SIM_MAX_SIZE_FILENAME);

        ZERO(&sim_default_conf.summary_file_name);
   strncpy(sim_default_conf.summary_file_name,
                          SIM_RESULT_FILE_DEFAULT,
                          SIM_MAX_SIZE_FILENAME);

        ZERO(&sim_default_conf.attempt_file_name);
   strncpy(sim_default_conf.attempt_file_name,
                          SIM_ATTEMPT_FLOW_FILENAME,
                        SIM_MAX_SIZE_FILENAME);

        ZERO(&sim_default_conf.positive_file_name);
  strncpy(sim_default_conf.positive_file_name,
                        SIM_POSITIVE_FLOW_FILENAME,
                        SIM_MAX_SIZE_FILENAME);

        ZERO(&sim_default_conf.negative_file_name);
  strncpy(sim_default_conf.negative_file_name,
                        SIM_NEGATIVE_FLOW_FILENAME,
                        SIM_MAX_SIZE_FILENAME);

}

void print_running_conf(SimConfInfo* ptr_running_conf)
{

        cout << "rows in lattice = " << ptr_running_conf->rows << endl;
        cout << "columns in lattice = " << ptr_running_conf->columns << endl;
  cout << "#Trials = " << ptr_running_conf->numTrials << endl;
  cout << "#Monte Carlo steps = " << ptr_running_conf->numSteps << endl;
  /*cout << "Attach detach step = " << ptr_running_conf->attach_detach_step << endl;
  cout << "Number of particles = " << ptr_running_conf->number_of_particles << endl;
  cout << "Population rate = " << ptr_running_conf->population_rate << endl;
  cout << "q1 = " << ptr_running_conf->q1 << endl;
  cout << "q2 = " << ptr_running_conf->q2 << endl;*/
  cout << "attach = " << ptr_running_conf->attach << endl;
  cout << "detach = " << ptr_running_conf->detach << endl;
        cout << "Gamma exchange g = " << ptr_running_conf->g << endl;
        cout << "Population percentage p = " << ptr_running_conf->p << endl;
        cout << "Enter probability alpha_e = " << ptr_running_conf->alpha_e << endl;
        cout << "Enter probability beta_e = " << ptr_running_conf->beta_e << endl;
        cout << "Exit probability gamma_ex = " << ptr_running_conf->gamma_ex << endl;
        cout << "Exit probability delta_ex = " << ptr_running_conf->delta_ex << endl;
  cout << "Result file name = " << ptr_running_conf->result_file_name << endl;
        cout << "Summary file name = " << ptr_running_conf->summary_file_name << endl;
        cout << "Attempts file name = " << ptr_running_conf->attempt_file_name << endl;
        cout << "Positive flow file name = " << ptr_running_conf->positive_file_name << endl;
        cout << "Negative flow file name = " << ptr_running_conf->negative_file_name << endl;
        cout << "Total flow file name = " << ptr_running_conf->total_file_name << endl;
}
void copy_param_to_running_conf(SimConfInfo* ptr_running_conf)
{

  int slen = 0;

        ptr_running_conf->rows            = sim_default_conf.rows;
        ptr_running_conf->columns          = sim_default_conf.columns;
  ptr_running_conf->numTrials         = sim_default_conf.numTrials;
  ptr_running_conf->numSteps          = sim_default_conf.numSteps;
  ptr_running_conf->attach_detach_step  = sim_default_conf.attach_detach_step;
  ptr_running_conf->number_of_particles = sim_default_conf.number_of_particles;
  ptr_running_conf->population_rate     = sim_default_conf.population_rate;
  ptr_running_conf->q1              = sim_default_conf.q1;
  ptr_running_conf->q2              = sim_default_conf.q2;
  ptr_running_conf->attach           = sim_default_conf.attach;
  ptr_running_conf->detach          = sim_default_conf.detach;
        ptr_running_conf->g                                         = sim_default_conf.g;
        ptr_running_conf->p                                         = sim_default_conf.p;
        ptr_running_conf->alpha_e                           = sim_default_conf.alpha_e;
        ptr_running_conf->beta_e                    = sim_default_conf.beta_e;
        ptr_running_conf->gamma_ex          = sim_default_conf.gamma_ex;
        ptr_running_conf->delta_ex                          = sim_default_conf.delta_ex;

  slen = strlen(sim_default_conf.result_file_name);
  memcpy(ptr_running_conf->result_file_name,
                  sim_default_conf.result_file_name,
                  slen);

  slen = strlen(sim_default_conf.summary_file_name);
  memcpy(ptr_running_conf->summary_file_name,
                  sim_default_conf.summary_file_name,
                  slen);
        ptr_running_conf->summary_file_name[slen] = '\0';
        cout << ptr_running_conf->summary_file_name << endl;

        slen = strlen(sim_default_conf.attempt_file_name);
  memcpy(ptr_running_conf->attempt_file_name,
                  sim_default_conf.attempt_file_name,
                  slen);
        ptr_running_conf->attempt_file_name[slen] = '\0';
        cout << ptr_running_conf->attempt_file_name << endl;

        slen = strlen(sim_default_conf.positive_file_name);
  memcpy(ptr_running_conf->positive_file_name,
                  sim_default_conf.positive_file_name,
                  slen);
        ptr_running_conf->positive_file_name[slen] = '\0';
        cout << ptr_running_conf->positive_file_name << endl;

        slen = strlen(sim_default_conf.negative_file_name);
  memcpy(ptr_running_conf->negative_file_name,
                  sim_default_conf.negative_file_name,
                  slen);
        ptr_running_conf->negative_file_name[slen] = '\0';
        cout << ptr_running_conf->negative_file_name << endl;

        slen = strlen(sim_default_conf.total_file_name);
  memcpy(ptr_running_conf->total_file_name,
                  sim_default_conf.total_file_name,
                  slen);
        ptr_running_conf->total_file_name[slen] = '\0';
        cout << ptr_running_conf->total_file_name << endl;

}

int parse_param(int argc, char **argv)
{
  int       i = 0;
  char      *c1;
  char      *c2;
```

```
int         slen = 0;

while (i < argc) {
   cout << "argv[" << i << "]: " << argv[i] << endl;
   c1 = argv[i];
   c2 = argv[++i];

                  if (strcmp(c1, "-ro") == 0) {
                           if (isdigit(*c2)) {
                                    sim_default_conf.rows = atoi(c2);
                           }
                           else {
                                    cout << "invalid value of rows in lattice (rows = " << c2 << endl;
         return -1;
      }
   }
                  else if (strcmp(c1, "-co") == 0) {
                           if (isdigit(*c2)) {
         sim_default_conf.columns = atoi(c2);
      }
      else {
         cout << "invalid value of columns in lattice (columns = " << c2 << endl;
         return -1;
      }
   }
   else if (strcmp(c1, "-nt") == 0) {
                           if (isdigit(*c2)) {
                                              sim_default_conf.numTrials = atoi(c2);
      }
      else {
         cout << "invalid value for the number of trials (trials = " << c2 << endl;
         return -1;
      }
   }
   else if (strcmp(c1, "-st") == 0) {
      if (isdigit(*c2)) {
         sim_default_conf.numSteps = atoi(c2);
      }
      else {
         cout << "invalid value for number of Monte Carlo steps = " << c2 << endl;
         return -1;
      }
   }
   else if (strcmp(c1, "-ar") == 0) {
      if (isdigit(*c2)) {
         sim_default_conf.attach = atof(c2);
      }
      else {
         cout << "invalid value for attach rate = " << c2 << endl;
         return -1;
      }
   }
   else if (strcmp(c1, "-o") == 0) {
      slen = strlen(c2);
      if (slen >= SIM_MAX_SIZE_FILENAME) {
         cout << "Name [" << c2 << "] for output file  is too long" << endl;
      return -1;
   }
   memcpy(sim_default_conf.result_file_name, c2, slen);
   sim_default_conf.result_file_name[slen] = '\0';
}
            else if (strcmp(c1, "-np") ==0) {
                     if (isdigit(*c2)) {
                              sim_default_conf.number_of_particles = atoi(c2);
                     }
                     else {
                              cout << "invalid value for number of particles = " << c2 << endl;
                              return -1;
                     }
            }
            else if (strcmp(c1, "-dr") ==0) {
                     if (isdigit(*c2)) {
                              sim_default_conf.detach = atof(c2);
                     }
                     else {
                              cout << "invalid value for detach rate = " << c2 << endl;
                              return -1;
                     }
            }
            else if (strcmp(c1, "-pr") ==0) {
                     if (isdigit(*c2)) {
                              sim_default_conf.population_rate = atof(c2);
                     }
                     else {
                              cout << "invalid value for population rate = " << c2 << endl;
                              return -1;
                     }
            }
            else if (strcmp(c1, "-ads") ==0) {
                     if (isdigit(*c2)) {
                              sim_default_conf.attach_detach_step = atoi(c2);
                     }
                     else {
                              cout << "invalid value for attach detach step = " << c2 << endl;
                              return -1;
                     }
            }
            else if (strcmp(c1, "-pprob") ==0) {
                     if (isdigit(*c2)) {
                              sim_default_conf.q1 = atof(c2);
                     }
                     else {
                              cout << "invalid value for positive move probability = " << c2 << endl;
                              return -1;
                     }
            }
            else if (strcmp(c1, "-sprob") ==0) {
                     if (isdigit(*c2)) {
                              sim_default_conf.q2 = atof(c2);
                     }
                     else {
                              cout << "invalid value for negative move probability = " << c2 << endl;
```

```
				return -1;
			}
		}
		else if (strcmp(c1, "-g") ==0) {
			if (isdigit(*c2)) {
				sim_default_conf.g = atof(c2);
			}
			else {
				cout << "invalid value for g = " << c2 << endl;
				return -1;
			}
		}
		else if (strcmp(c1, "-p") ==0) {
			if (isdigit(*c2)) {
				sim_default_conf.p = atof(c2);
			}
			else {
				cout << "invalid value for p = " << c2 << endl;
				return -1;
			}
		}
		else if (strcmp(c1, "-ae") ==0) {
			if (isdigit(*c2)) {
				sim_default_conf.alpha_e = atof(c2);
			}
			else {
				cout << "invalid value for alpha_e = " << c2 << endl;
				return -1;
			}
		}
		else if (strcmp(c1, "-be") ==0) {
			if (isdigit(*c2)) {
				sim_default_conf.beta_e = atof(c2);
			}
			else {
				cout << "invalid value for beta_e = " << c2 << endl;
				return -1;
			}
		}
		else if (strcmp(c1, "-gex") ==0) {
			if (isdigit(*c2)) {
				sim_default_conf.gamma_ex = atof(c2);
			}
			else {
				cout << "invalid value for gamma_ex = " << c2 << endl;
				return -1;
			}
		}
		else if (strcmp(c1, "-dex") ==0) {
			if (isdigit(*c2)) {
				sim_default_conf.delta_ex = atof(c2);
			}
			else {
				cout << "invalid value for delta_ex = " << c2 << endl;
				return 01;
			}
```

```
                }
      else if (strcmp(c1, "-result") == 0) {
         slen = strlen(c2);
         if (slen >= SIM_MAX_SIZE_FILENAME) {
            cout << "Name [" << c2 << "] for summary file  is too long" << endl;
            return -1;
         }
         memcpy(sim_default_conf.summary_file_name, c2, slen);
         sim_default_conf.summary_file_name[slen] = '\0';
                        cout << sim_default_conf.summary_file_name << endl;
      }
      else if (strcmp(c1, "-attempt") == 0) {
         slen = strlen(c2);
         if (slen >= SIM_MAX_SIZE_FILENAME) {
            cout << "Name [" << c2 << "] for attempt file  is too long" << endl;
            return -1;
         }
         memcpy(sim_default_conf.attempt_file_name, c2, slen);
         sim_default_conf.attempt_file_name[slen] = '\0';
                        cout << sim_default_conf.attempt_file_name << endl;
      }
      else if (strcmp(c1, "-positive") == 0) {
         slen = strlen(c2);
         if (slen >= SIM_MAX_SIZE_FILENAME) {
            cout << "Name [" << c2 << "] for positiveflow file  is too long" << endl;
            return -1;
         }
         memcpy(sim_default_conf.positive_file_name, c2, slen);
         sim_default_conf.positive_file_name[slen] = '\0';
                        cout << sim_default_conf.positive_file_name << endl;
      }
      else if (strcmp(c1, "-negative") == 0) {
         slen = strlen(c2);
         if (slen >= SIM_MAX_SIZE_FILENAME) {
            cout << "Name [" << c2 << "] for negativeflow file  is too long" << endl;
            return -1;
         }
         memcpy(sim_default_conf.negative_file_name, c2, slen);
         sim_default_conf.negative_file_name[slen] = '\0';
                        cout << sim_default_conf.negative_file_name << endl;
      }
      else if (strcmp(c1, "-total") == 0) {
         slen = strlen(c2);
         if (slen >= SIM_MAX_SIZE_FILENAME) {
            cout << "Name [" << c2 << "] for totalflow file  is too long" << endl;
            return -1;
         }
         memcpy(sim_default_conf.total_file_name, c2, slen);
         sim_default_conf.total_file_name[slen] = '\0';
                        cout << sim_default_conf.total_file_name << endl;
      }
        }
        return 1;
}
```

```
/**********************************************************/
//    RandomNumber.h:
//
// This file contains declaration of random functions
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/**********************************************************/

#ifndef _RANDOMNUMBER_H
#define _RANDOMNUMBER_H

#include <time.h>
#include <cstdlib>

void   initialize_random_number();
double getRandomBetween0And1();
int    getRandomBetween0AndMax(int max);


/**********************************************************/
//        Do not place anything after endif
/**********************************************************/

#endif _RANDOMNUMBER_H
```

```
/**********************************************************/
//    RandomNumber.cpp
//
// This file contains definition of random number functions
//
// Author: Josh Gonzalez
// Date:  June/20/2007
//
/**********************************************************/

#include "RandomNumber.h"

//////////////////////////////////////////////////////////////////////
// initialize_random_number
//////////////////////////////////////////////////////////////////////

void initialize_random_number()
{
        time_t    seconds;

   time(&seconds);
   srand((unsigned int) seconds);
}

//////////////////////////////////////////////////////////////////////
// getRandomBetween0And1
//
// returns random number >= 0 and < 1
//
//////////////////////////////////////////////////////////////////////

double getRandomBetween0And1()
{
        return ((double)rand()/(double)RAND_MAX);
}

//////////////////////////////////////////////////////////////////////
// getRandomBetween0AndMax
//
// returns random number >=0 and < max
//
//////////////////////////////////////////////////////////////////////

int getRandomBetween0AndMax(int max)
{
        return(rand() % max);
}
```

```
/************************************************************/
//    Lattice_2D.h: interface for the Lattice_2D class.
//
// This file contains all constant, structure, typedef, and class definitions
// for Lattice_2D class.
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/************************************************************/

#ifndef _LATTICE_2D_H
#define _LATTICE_2D_H

#include <fstream>
#include <vector>
#include <time.h>
#include <list>
#include "Cluster.h"
#include "2D_param.h"
#include "RandomNumber.h"

using namespace std;

#define POSITIVE_PARTICLE  1
#define NEGATIVE_PARTICLE  -1
#define EMPTY          0
#define LEFT                       0
#define  RIGHT                     1
#define TOP                              2
#define BOTTOM                           3
class Lattice_2D
{
private:
        int                            rows;
        int                            columns;
        int                            numEmptyCells;

        int       x_coord;  //use as working x coordinate
        int       y_coord;  //use as working y coordinate
        int       x_coord_bond; //x coordinates of cell bonded to x_coord
         int      y_coord_bond; // y coordinates of cell bonded to y_coord
//      int       total_positive;
//      int       total_negative;
//      int       positive_current;
//      int       negative_current;
        int       attach_positive;
        int       attach_negative;
        int       detach_positive;
        int       detach_negative;
        int       exit_positive;
        int       exit_negative;
        int       enter_positive;
        int       enter_negative;
        int       att_ex;
        int       att_e;
        int         total_enter;
        int         total_exit;
        int         total_attach;
        int         total_detach;
        double      snap_enter;
        double      snap_exit;
        double      snap_attach;
        double      snap_detach;
        int         lastCell;    // x coordinate of last cell
        int         bondedCellPosition;
        int         enterCellPosition;
        bool        noCluster;
        SimConfInfo*   my_ptr_running_conf;

        list<Cluster*>  myClusterList;
        Cluster*        currentClusterPtr;
        list<Cluster*>::iterator biggestClusterPtr;

        vector<int>         l_columns;
        vector<vector<int> > lattice;

        ofstream        testLog;
        ofstream        summaryFile;
        ofstream        resultFile;
        ofstream        positiveFlowFile;
        ofstream        negativeFlowFile;
        ofstream        totalFlowFile;
        ofstream        attemptFlowFile;
        time_t          timeheader;

public:

                                        Lattice_2D(SimConfInfo* ptr_running_conf);
        virtual                 ~Lattice_2D();

        bool        initialize();
        void        clear();
        void        populate();
        void        runMonteCarloStep();
        bool        randomlyFindEmptyCell();
        void        selectRandomCell();
        void        selectRandomBondedCell();
        void        setBondLeftOfCell(int x, int y);
        void        leftBondMovements();
        void        setBondRightOfCell(int x, int y);
        void        rightBondMovements();
        void        setBondTop(int x, int y);
        void        setBondBottom(int x, int y);
        void        verticalBondMovements();
        bool        isCellEmpty(int x, int y);
        bool        isCellPositive(int x, int y);
        bool        isCellNegative(int x, int y);
        bool        isColumnEmpty(int x);
        bool        isVerticalBond(int x);
        bool        isHorizontalBond(int x, int y);
        void        moveRight(int x, int y);
void        moveLeft(int x, int y);
void        exchange(int x1, int y1, int x2, int y2);
void        chargeExchange();
void        attach();
void        detach();
void        snapshot();
void        enter();
void        exit_ends();
void        exit_neg();
void        exit_pos();
void        addPositiveParticle(int x, int y);
void        addNegativeParticle(int x, int y);
void        exitParticle(int x, int y);
int         findClusters();
void        findBiggestCluster();

int         findNumberOfParticles(int x);
void        printLattice();

void        printLatticeToTestFile();
bool        openTestLogFile();
bool        openResultFile();
bool        openFile(char* file_name, ofstream  f_handle);
bool        openSummaryFile();
bool        openPositiveFlowFile();
bool        openNegativeFlowFile();
bool        openTotalFlowFile();
bool        openAttemptFlowFile();
bool        openSnapshotsFile();
int         runTest();
void        testRandom();
void        testMoveRight();
void        testMoveLeft();
void        testExchange();
void        testSetBondLeftOfCell();
void        testSetBondRightOfCell();
void        testSetBondTop();
void        testSetBondBottom();
void        testExitParticle();
void        testEnter();
void        testExitEnds();
void        testAddPositiveParticle();
void        testAddNegativeParticle();
void        testLeftBondMovementsPositive();
void        testLeftBondMovementsEmpty();
void        testLeftBondMovementsNegative();
void        testRightBondMovementsNegative();
void        testRightBondMovementsEmpty();
void        testRightBondMovementsPositive();
void        testVerticalBondMovementsEmpty();
void        testVerticalBondMovementsSamePolarity();
void        testVerticalBondMovementsExchange();
void        testIsColumnEmpty();
void        testIsVerticalBond();
void        testIsHorizontalBond();
void        testFindCluster();
```

```
        void        logClusters();
        void        resultClusters();

};

/*************************************************************/
//          Do not place anything after endif
/*************************************************************/
#endif _LATTICE_2D_H
```

```
/************************************************************/
//     Lattice_2D.cpp
//
// This file contains the implementation of the Lattice_2D class as well
//  as tests of each of the Lattice_2D functions
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/************************************************************/

#include "Lattice_2D.h"

////////////////////////////////////////////////////////////////////////
// Lattice_2D::Constructor
////////////////////////////////////////////////////////////////////////

Lattice_2D::Lattice_2D(SimConfInfo* ptr_running_conf)
{
        my_ptr_running_conf = ptr_running_conf;
        rows = ptr_running_conf->rows;
        columns = ptr_running_conf->columns;
        numEmptyCells = rows*columns;
        // resize lattice to the correct number of row and columns
        l_columns.resize(columns);
        lattice.resize(rows, l_columns);
        lastCell = lattice.size() - 1;
}

////////////////////////////////////////////////////////////////////////
// Lattice_2D::Destructor
////////////////////////////////////////////////////////////////////////

Lattice_2D::~Lattice_2D()
{

}

////////////////////////////////////////////////////////////////////////
// Lattice_2D::initialize
////////////////////////////////////////////////////////////////////////
bool
Lattice_2D::initialize()
{
        // initialize random number
        initialize_random_number();
        // initialize the output stream for test/debug
        // and check if it is opened properly

        openResultFile();

        // remove all particles from the lattice
        clear();
        populate();
        attach_positive = 0;
        attach_negative = 0;
        detach_positive = 0;
        detach_negative = 0;
        exit_positive = 0;
        exit_negative = 0;
        enter_positive = 0;
        enter_negative = 0;
//      positive_current = 0;
//      negative_current = 0;
        att_e = 0;
        att_ex = 0;
        snap_enter = 0;
        snap_exit = 0;
        snap_attach = 0;
        snap_detach = 0;

        return true;
}

///////////////////////////////////////////////////////////////////
// Lattice_2D::clear
//
// Clear the lattice by removing all particles from it
//
///////////////////////////////////////////////////////////////////

void
Lattice_2D::clear()
{
        int i,j;

        // set all slots in the lattice  to 0's (no particles)
        for (j=0; j <l_columns.size(); j++)
        {
                for (i=0; i< lattice.size(); i++)
                        lattice[i][j] = EMPTY;
        }
}

///////////////////////////////////////////////////////////////////
// Lattice_2D::populate
//
// Populate lattice with particles
//
///////////////////////////////////////////////////////////////////

void
Lattice_2D::populate()
{
        int i;

//      total_positive = 0;
//      total_negative = 0;

        // for now assume lattice has 1000 cells (500x2) and half of the cells
        // should contain particles. The number of positive particles is equal
        // to the number of negative particles.
```

```
  // TODO: use configuration parameters to populate lattice

        for (i=0; i < (lattice.size()*(my_ptr_running_conf->p)); i++)
        {
                if (!randomlyFindEmptyCell()) {
                        testLog << "ERROR: in populate, no empty cells left!" << endl;
                        exit(-1);
                }
                // in empty slot insert positive particle
                lattice[x_coord][y_coord] = POSITIVE_PARTICLE;
                numEmptyCells--;
//              total_positive++;
                if (!randomlyFindEmptyCell()) {
                        testLog << "ERROR: in populate, no empty cells left!" << endl;
                        exit(-1);
                };
                // in empty slot insert positive particle
                lattice[x_coord][y_coord] = NEGATIVE_PARTICLE;
                numEmptyCells--;
//              total_negative++;
        }

}

////////////////////////////////////////////////////////////////////
// Lattice_2D::snapshot
////////////////////////////////////////////////////////////////////
void
Lattice_2D::snapshot()
{
        total_enter = enter_positive + enter_negative;
        total_exit = exit_positive + exit_negative;
        total_attach = attach_positive + attach_negative;
        total_detach = detach_positive + detach_negative;

        snap_enter = total_enter - snap_enter;
        snap_exit = total_exit - snap_exit;
        snap_attach = total_attach - snap_attach;
        snap_detach = total_detach - snap_detach;

        resultFile << snap_enter << "\t" << snap_exit << "\t" << snap_attach << "\t";
        resultFile << snap_detach << endl;

        snap_enter = total_enter;
        snap_exit = total_exit;
        snap_attach = total_attach;
        snap_detach = total_detach;

}

////////////////////////////////////////////////////////////////////
// Lattice_2D::enter
////////////////////////////////////////////////////////////////////
void
Lattice_2D::enter()
```

```
{
	double r1;
	x_coord = 0;
	r1 = getRandomBetween0And1();
	if (r1 <= my_ptr_running_conf->alpha_e)
	{
		// check for empty space at beginning of lattice
		// in order to enter positive particle
		enterCellPosition = getRandomBetween0AndMax(2);
		switch (enterCellPosition)
		{
			case 0:
				y_coord = 0;
				if (isCellEmpty(x_coord,y_coord))
				{
					addPositiveParticle(x_coord,y_coord);
					enter_positive++;
//					total_positive++;
				}
				else
					att_e++;
				break;
			case 1:
				y_coord = 1;
				if (isCellEmpty(x_coord, y_coord))
				{
					addPositiveParticle(x_coord, y_coord);
					enter_positive++;
//					total_positive++;
				}
				else
					att_e++;
				break;
		}
	}

	x_coord = lattice.size()-1;
	r1 = getRandomBetween0And1();
	if (r1 <= my_ptr_running_conf->beta_e)
	{
		// check for empty space at end of lattice
		// in order to enter negative particle
		enterCellPosition = getRandomBetween0AndMax(2);

		switch (enterCellPosition)
		{
			case 0:
				y_coord = 0;
				if (isCellEmpty(x_coord, y_coord))
				{
					addNegativeParticle(x_coord, y_coord);
					enter_negative++;
//					total_negative++;
				}
				else
					att_e++;
```

```
                                        break;
                              case 1:
                                        y_coord = 1;
                                        if (isCellEmpty(x_coord, y_coord))
                                        {
                                                  addNegativeParticle(x_coord, y_coord);
                                                  enter_negative++;
//                                                total_negative++;
                                        }
                                        else
                                                  att_e++;
                                        break;
                    }
          }
          return;
}

///////////////////////////////////////////////////////////////////////
// Lattice_2D::exit_negative
///////////////////////////////////////////////////////////////////////
void
Lattice_2D::exit_neg()
{
          double r1;

          if(isCellNegative(x_coord,y_coord))
          {
                    r1 = getRandomBetween0And1();
                    if (r1 <= my_ptr_running_conf->gamma_ex)
                    {
                              exitParticle(x_coord,y_coord);
                              exit_negative++;
//                            total_negative--;
                    }
                    else
                              att_ex++;
          }
}

///////////////////////////////////////////////////////////////////////
// Lattice_2D::exit_pos
///////////////////////////////////////////////////////////////////////
void
Lattice_2D::exit_pos()
{
          double r1;
          if (isCellPositive(x_coord,y_coord))
          {
                    r1 = getRandomBetween0And1();
                    if (r1 <= my_ptr_running_conf->delta_ex)
                    {
                              exitParticle(x_coord,y_coord);
                              exit_positive++;
//                            total_positive--;
                    }
                    else
```

```
                        att_ex++;
                }
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::exit_ends
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::exit_ends()
{
        x_coord = 0;

        enterCellPosition = getRandomBetween0AndMax(2);

        switch (enterCellPosition)
        {
                case 0:
                        y_coord = 0;
                        exit_neg();
                        break;
                case 1:
                        y_coord = 1;
                        exit_neg();
                        break;
        }

        x_coord = lattice.size()-1;

        enterCellPosition = getRandomBetween0AndMax(2);

        switch (enterCellPosition)
        {
                case 0:
                        y_coord = 0;
                        exit_pos();
                        break;
                case 1:
                        y_coord = 1;
                        exit_pos();
                        break;
        }
        return;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::addPositiveParticle
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::addPositiveParticle(int x, int y)
{
        lattice[x_coord][y_coord] = POSITIVE_PARTICLE;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::addNegativeParticle
//////////////////////////////////////////////////////////////////////
```

```
void
Lattice_2D::addNegativeParticle(int x, int y)
{
        lattice[x_coord][y_coord] = NEGATIVE_PARTICLE;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::runMonteCarloStep
//
////////////////////////////////////////////////////////////////////
void
Lattice_2D::runMonteCarloStep()
{
        selectRandomCell();
        selectRandomBondedCell();
        if (bondedCellPosition == LEFT)
        {
                leftBondMovements();
                return;
        }
        if (bondedCellPosition == RIGHT)
        {
                rightBondMovements();
                return;
        }
        if (bondedCellPosition == TOP)
        {
                verticalBondMovements();
                return;
        }
        if (bondedCellPosition == BOTTOM)
        {
                verticalBondMovements();
                return;
        }

}

////////////////////////////////////////////////////////////////////
// Lattice_2D::randomlyFindEmptyCell
//
// find empty cell and places its coordinates in x_coord and y_coord
////////////////////////////////////////////////////////////////////
bool
Lattice_2D::randomlyFindEmptyCell()
{
        bool loop = true;

        if (numEmptyCells == 0)
                return false;

        while (loop)
        {
                selectRandomCell();
                if (isCellEmpty(x_coord, y_coord))
                        loop = false;
        }
        return true;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::selectRandomCell
//
// sets x_coord and y_coord randomly
////////////////////////////////////////////////////////////////////
void
Lattice_2D::selectRandomCell()
{
        x_coord = getRandomBetween0AndMax(rows);
        y_coord = getRandomBetween0AndMax(columns);
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::selectRandomBondedCell
//
// sets x_coord_bond and y_coord_bond
////////////////////////////////////////////////////////////////////
void
Lattice_2D::selectRandomBondedCell()
{
        bondedCellPosition = getRandomBetween0AndMax(4);

        switch (bondedCellPosition)
        {
                case LEFT :     // left
                        setBondLeftOfCell(x_coord, y_coord);
                        break;
                case RIGHT :    // right
                        setBondRightOfCell(x_coord, y_coord);
                        break;
                case TOP :      // top
                  setBondTop(x_coord, y_coord);
                  break;
                case BOTTOM :   // bottom
                        setBondBottom(x_coord, y_coord);
                        break;
                default:
                        cout << "ERROR: not a valid number for bondedCellPosition " << bondedCellPosition
<< endl;

        }
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::setBondLeftOfCell
//
// sets x_coord_bond and y_coord_bond to left side of x and y
////////////////////////////////////////////////////////////////////
void
Lattice_2D::setBondLeftOfCell(int x, int y)
{
        if (x == 0)
```

```
                    // set to same cell
                    x_coord_bond = 0;
          else
                    x_coord_bond = x - 1;
          y_coord_bond = y;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::setBondRightOfCell
//
// sets x_coord_bond and y_coord_bond to right side of x and y
////////////////////////////////////////////////////////////////////
void
Lattice_2D::setBondRightOfCell(int x, int y)
{
          if (x == lattice.size()-1)
                    // set to same cell
                    x_coord_bond = lattice.size()-1;
          else
                    x_coord_bond = x + 1;
          y_coord_bond = y;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::setBondTop
//
// sets x_coord_bond and y_coord_bond above
////////////////////////////////////////////////////////////////////
void
Lattice_2D::setBondTop(int x, int y)
{
          if (y == 0)
                    // set last cell in the same row
                    y_coord_bond = 1;
          else
                    y_coord_bond = 1;
          x_coord_bond = x;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::setBondBottom
//
// sets x_coord_bond and y_coord_bond above
////////////////////////////////////////////////////////////////////
void
Lattice_2D::setBondBottom(int x, int y)
{
          if (y == 1)
                    // set last cell in the same row
                    y_coord_bond = 0;
          else
                    y_coord_bond = 0;
          x_coord_bond = x;
}

////////////////////////////////////////////////////////////////////
```

```
// Lattice_2D::isCellEmpty
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::isCellEmpty(int x, int y)
{
        if (lattice[x][y] == EMPTY)
                return true;
        return false;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::isCellPositive
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::isCellPositive(int x, int y)
{
        if (lattice[x][y] == POSITIVE_PARTICLE)
                return true;
        return false;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::isCellNegative
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::isCellNegative(int x, int y)
{
        if (lattice[x][y] == NEGATIVE_PARTICLE)
                return true;
        return false;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::isColumnEmpty
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::isColumnEmpty(int x)
{
        if (isCellEmpty(x, 0) && isCellEmpty(x, 1))
                return true;
        return false;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::isVerticalBond
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::isVerticalBond(int x)
{
        // vertical bond is present is both cell have a particle
        // not just one
        if ((!isCellEmpty(x, 0)) &&
                (!isCellEmpty(x, 1)))
                return true;
        return false;
}
```

```
///////////////////////////////////////////////////////////////////////
// Lattice_2D::isHorizontalBond
///////////////////////////////////////////////////////////////////////
bool
Lattice_2D::isHorizontalBond(int x, int y)
{
	int next_x;

	// take care of end of lattice coordinates
	if (x == lattice.size()-1)
		return false;

	next_x = x+1;
	// horizontal bond is present if both cell have a particle
	// not just one
	if ((!isCellEmpty(x, y)) &&
		(!isCellEmpty(next_x, y)))
		return true;
	return false;

}

///////////////////////////////////////////////////////////////////////
// Lattice_2D::moveRight
///////////////////////////////////////////////////////////////////////
void
Lattice_2D::moveRight(int x, int y)
{
	lattice[x+1][y] = lattice[x][y];
	// original cell is set to empty after move
   lattice[x][y] = EMPTY;
}

///////////////////////////////////////////////////////////////////////
// Lattice_2D::moveLeft
///////////////////////////////////////////////////////////////////////
void
Lattice_2D::moveLeft(int x, int y)
{
	lattice[x-1][y] = lattice[x][y];
	// original cell is set to empty after move
   lattice[x][y] = EMPTY;
}

///////////////////////////////////////////////////////////////////////
// Lattice_2D::exchange
///////////////////////////////////////////////////////////////////////
void
Lattice_2D::exchange(int x1, int y1, int x2, int y2)
{
	int temp;

	temp = lattice[x2][y2];
	lattice[x2][y2] = lattice[x1][y1];
	lattice[x1][y1] = temp;
```

```
}

///////////////////////////////////////////////////////////////////
// Lattice_2D::chargeExchange
///////////////////////////////////////////////////////////////////
void
Lattice_2D::chargeExchange()
{
	if(my_ptr_running_conf->g == 0)
		return;
	double r1;
	r1 = getRandomBetween0And1();

	if (r1 <= my_ptr_running_conf->g)
		exchange(x_coord_bond, y_coord_bond, x_coord, y_coord);
}

///////////////////////////////////////////////////////////////////
// Lattice_2D::exitParticle
///////////////////////////////////////////////////////////////////
void
Lattice_2D::exitParticle(int x, int y)
{
	lattice[x_coord][y_coord] = EMPTY;
}

///////////////////////////////////////////////////////////////////
// Lattice_2D::attach
///////////////////////////////////////////////////////////////////
void
Lattice_2D::attach()
{
	int c;
	double r1;
	double a;
	a = my_ptr_running_conf->attach;

	selectRandomCell();

	if(isCellEmpty(x_coord,y_coord))
	{
		if(my_ptr_running_conf->attach == 0)
			return;

		r1 = getRandomBetween0And1();
		if(x_coord == lattice.size()-1)
		{
			if(!(isCellEmpty(x_coord-1,y_coord)) || y_coord == 0 && (!(isCellEmpty(x_coord,1)))
				|| y_coord == 1 && (!(isCellEmpty(x_coord,0))))
				a = a*2;
		}
		if(x_coord == 0)
		{
			if(!(isCellEmpty(x_coord+1,y_coord)) || y_coord == 0 && (!(isCellEmpty(x_coord,1)))
				|| y_coord == 1 && (!(isCellEmpty(x_coord,0))))
```

```
					a = a*2;
			}
			if(x_coord != lattice.size()-1 && x_coord != 0)
			{
				if(!(isCellEmpty(x_coord+1,y_coord)) || !(isCellEmpty(x_coord-1,y_coord)) || y_coord
== 0 && (!(isCellEmpty(x_coord,1)))
					|| y_coord == 1 && (!(isCellEmpty(x_coord,0))))
					a = a*2;
			}
			if (r1 <= a)
			{
				c = getRandomBetween0AndMax(2);
				switch (c)
				{
					case 0:
						addPositiveParticle(x_coord,y_coord);
						attach_positive++;
//						total_positive++;
						return;

					case 1:
						addNegativeParticle(x_coord,y_coord);
						attach_negative++;
//						total_negative++;
						return;

				}
			}
		}
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::detach
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::detach()
{
	double r1;

	r1 = getRandomBetween0And1();

	if(r1 <= my_ptr_running_conf->detach)
	{
		if(my_ptr_running_conf->detach == 0)
			return;
		if(isCellPositive(x_coord,y_coord))
		{
			detach_positive++;
//			total_positive--;
		}
		else
		{
			detach_negative++;
//			total_negative--;
		}
		exitParticle(x_coord,y_coord);
```

```
                return;
        }
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::leftBondMovements
////////////////////////////////////////////////////////////////////
void
Lattice_2D::leftBondMovements()
{

        if ((x_coord == x_coord_bond) && isCellPositive(x_coord, y_coord))
                return;

        if ((x_coord == x_coord_bond) && isCellNegative(x_coord, y_coord))
        {
                exitParticle(x_coord, y_coord);
                return;
        }

        if (isCellPositive(x_coord, y_coord))
                return;

        if (isCellEmpty(x_coord, y_coord) &&
                isCellEmpty(x_coord_bond, y_coord_bond))
                return;

        if (isCellEmpty(x_coord, y_coord) &&
                isCellNegative(x_coord_bond, y_coord_bond))
                return;

        if (isCellNegative(x_coord, y_coord) &&
                isCellEmpty(x_coord_bond, y_coord_bond))
        {
                moveLeft(x_coord, y_coord);
                /*if(x_coord == 50)
                {
                        negative_current++;
                }*/
                return;
        }

        if (isCellEmpty(x_coord, y_coord) &&
                isCellPositive(x_coord_bond, y_coord_bond))
        {
                moveRight(x_coord_bond, y_coord_bond);
                /*if(x_coord == 50)
                {
                        positive_current++;
                }*/
                return;
        }

        if (isCellNegative(x_coord, y_coord) &&
                isCellPositive(x_coord_bond, y_coord_bond))
        {
```

```
				chargeExchange();
				/*if(x_coord == 50)
				{
					positive_current++;
					negative_current++;
				}*/
				return;
			}
			return;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::rightBondMovements
////////////////////////////////////////////////////////////////////
void
Lattice_2D::rightBondMovements()
{

		if ((x_coord == x_coord_bond) && isCellNegative(x_coord, y_coord))
			return;

		if ((x_coord == x_coord_bond) && isCellPositive(x_coord, y_coord))
		{
			exitParticle(x_coord, y_coord);
			return;
		}

		if (isCellNegative(x_coord, y_coord))
			return;

		if (isCellEmpty(x_coord, y_coord) &&
			isCellEmpty(x_coord_bond, y_coord_bond))
			return;

		if (isCellEmpty(x_coord, y_coord) &&
			isCellPositive(x_coord_bond, y_coord_bond))
			return;

		if (isCellEmpty(x_coord, y_coord) &&
			isCellNegative(x_coord_bond, y_coord_bond))
		{
			moveLeft(x_coord_bond, y_coord_bond);
			/*if(x_coord == 49)
			{
				negative_current++;
			}*/
			return;
		}

		if (isCellPositive(x_coord, y_coord) &&
			isCellEmpty(x_coord_bond, y_coord_bond))
		{
			moveRight(x_coord, y_coord);
			/*if(x_coord == 49)
			{
				positive_current++;
```

```
                }*/
                return;
        }

        if (isCellPositive(x_coord, y_coord) &&
                isCellNegative(x_coord_bond, y_coord_bond))
        {
                chargeExchange();
                /*if(x_coord == 49)
                {
                        positive_current++;
                        negative_current++;
                }*/
                return;
        }
        return;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::verticalBondMovements
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::verticalBondMovements()
{
/*      int xtemp;
        int ytemp;
*/

        // if bond is the same as cell (e.g. bottom bond on bottom cell or top bond on top cell)
        if (y_coord == y_coord_bond)
                return;

        // if both cells are empty there is nothing to do
        if ((isCellEmpty(x_coord, y_coord) &&
                isCellEmpty(x_coord_bond, y_coord_bond)))
                return;

        // if a particle-hole pair exists the particle jumps
        if ((isCellEmpty(x_coord, y_coord) &&
                (!(isCellEmpty(x_coord_bond, y_coord_bond)))))
        {
                exchange(x_coord_bond, y_coord_bond, x_coord, y_coord);
                return;
        }

        if (!(isCellEmpty(x_coord, y_coord)) &&
                isCellEmpty(x_coord_bond, y_coord_bond))
        {
                exchange(x_coord_bond, y_coord_bond, x_coord, y_coord);
                return;
        }

        // if both particle are of equal polarity there is nothing to do
        if (lattice[x_coord][y_coord] == lattice[x_coord_bond][y_coord_bond])
                return;
```

```
/*	exchange(x_coord_bond, y_coord_bond, x_coord, y_coord);
	xtemp = x_coord_bond;
	ytemp = y_coord_bond;

	// execute movements in cell [x_coord][y_coord]
	if (isCellPositive(x_coord, y_coord))
	{
		setBondRightOfCell(x_coord, y_coord);
		rightBondMovements();
	}
	else
	{
		// particle at [x_coord][y_coord] is negative
		setBondLeftOfCell(x_coord, y_coord);
		leftBondMovements();
	}

	// now execute movements in cell[xtemp][ytemp] which was the other
	// cell on the original bond
	x_coord = xtemp;
	y_coord = ytemp;

	if (isCellPositive(x_coord, y_coord))
	{
		setBondRightOfCell(x_coord, y_coord);
		rightBondMovements();
	}
	else
	{
		// we know particle is negative
		setBondLeftOfCell(x_coord, y_coord);
		leftBondMovements();
	}*/
	return;
}

///////////////////////////////////////////////////////////////////
// Lattice_2D::findClusters
///////////////////////////////////////////////////////////////////
int
Lattice_2D::findClusters()
{
	int x;

	noCluster = true;
	for (x=0; x < lattice.size(); x++)
	{
		// if column x is empty and cluster has not been found
		// continue until the start of a cluster is found
		if (isColumnEmpty(x) && noCluster == true)
			continue;

		if ((noCluster == false) &&
		   (!isHorizontalBond(x,0))&&
			(!isHorizontalBond(x,1)) &&
			(!isVerticalBond(x)))
```

```
{
        // found end of cluster
        currentClusterPtr->addParticlesToCluster(findNumberOfParticles(x));
        noCluster = true;
        // update end position of cluster with previous column (x-1)
        if (isColumnEmpty(x))
                currentClusterPtr->setEndPosition(x-1);
        else
                currentClusterPtr->setEndPosition(x);
        // insert pointer to current cluster in to my list of
        // cluster pointers
        myClusterList.push_back(currentClusterPtr);
        continue;
}

// test to see if the a bond in the current column
if (isVerticalBond(x) ||
        isHorizontalBond(x,0) ||
        isHorizontalBond(x,1))
{
        if (noCluster == true)
        {
                //found a cluster
                noCluster = false;
                // create cluster keeping track of it start x position
                //  and keep track of its number of particles
                currentClusterPtr = new Cluster(x);
                if ( currentClusterPtr == 0 )
                {
                        // log error in creation of new cluster
                        cout << "ERROR: unable to create a new cluster instance" << endl;
                        return -1;
                }
                currentClusterPtr->addParticlesToCluster(findNumberOfParticles(x));
                // test for end of lattice
                if (x == lattice.size()-1)
                {
                        currentClusterPtr->setEndPosition(x);
                        // insert pointer to current cluster in to my list of cluster pointers
                        myClusterList.push_back(currentClusterPtr);
                }

                continue;
        }
        else
        {
                currentClusterPtr->addParticlesToCluster(findNumberOfParticles(x));
                // test for end of lattice
                if (x == lattice.size()-1)
                {
                        currentClusterPtr->setEndPosition(x);
                        // insert pointer to current cluster in to my list of cluster pointers
                        myClusterList.push_back(currentClusterPtr);
                }
                continue;
        }
```

```
                }
        }

        return 1;
}

/////////////////////////////////////////////////////////////////////
// Lattice_2D::findBiggestCluster
// set biggestClusterPtr to the cluster with the largest number of
// particles
/////////////////////////////////////////////////////////////////////
void
Lattice_2D::findBiggestCluster()
{
        int currentBiggestCount=0;
        list<Cluster*>::iterator clusterPtr;

        for (clusterPtr = myClusterList.begin();
                clusterPtr != myClusterList.end(); clusterPtr++)
                {
                if ((*clusterPtr)->getNumberOfParticles() >= currentBiggestCount) {
                        biggestClusterPtr = clusterPtr;
                        currentBiggestCount = (*clusterPtr)->getNumberOfParticles();
                }
        }
}

/////////////////////////////////////////////////////////////////////
// Lattice_2D::findNumberOfParticles(int x)
/////////////////////////////////////////////////////////////////////
int
Lattice_2D::findNumberOfParticles(int x)
{
        int number;
        number = 0;
        if (!(isCellEmpty(x,0)))
                number++;

        if (!(isCellEmpty(x,1)))
                number++;

        return number;
}


/////////////////////////////////////////////////////////////////////
// Lattice_2D::printLattice
//
// prints out contents of each cell two rows by 50
/////////////////////////////////////////////////////////////////////
void
Lattice_2D::printLattice()
{
        int i,j;

        resultFile << "====== Start of printLattice ========================" << endl;
```

```
	for (j=0; j < l_columns.size(); j++)
	{
		resultFile << "===== Row " << j+1 << " ===========================" << endl;
		for (i=0; i < lattice.size(); i++)
		{
			if (i%50 == 0) {
				resultFile << endl;
				resultFile << "[" << i << "," << j << "]";
				if (i < 50)
					resultFile << "  ";
				if (i == 50)
					resultFile << " ";
				resultFile << ":";
			}
			resultFile << lattice[i][j];
		}
		resultFile << endl << endl;
	}
	resultFile << "=====  End of printLattice ========================" << endl << endl;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::printLatticeToTestFile
//
// prints out contents of each cell two rows by 50
////////////////////////////////////////////////////////////////////
void
Lattice_2D::printLatticeToTestFile()
{
	int i,j;

	testLog << "=====  Start of printLattice ========================" << endl;
	for (j=0; j < l_columns.size(); j++)
	{
		testLog << "===== Row " << j+1 << " ===========================" << endl;
		for (i=0; i < lattice.size(); i++)
		{
			if (i%50 == 0) {
				testLog << endl;
				testLog << "[" << i << "," << j << "]";
				if (i < 50)
					testLog << "  ";
				if (i == 50)
					testLog << " ";
				testLog << ":";
			}
			testLog << lattice[i][j];
		}
		testLog << endl << endl;
	}
	testLog << "=====  End of printLattice ========================" << endl << endl;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::openTestLogFile
////////////////////////////////////////////////////////////////////
```

```
bool
Lattice_2D::openTestLogFile()
{

        testLog.open("log_2D.txt");
  if (!testLog.is_open())
  {
    cout << "Cannot open testLog file: log_2D.txt" << endl;
    return false;
  }
        // output the time to the test Log file
  time (&timeheader);
  testLog << "This data is taken on: " << ctime (&timeheader) << endl << endl;
        return true;

}

////////////////////////////////////////////////////////////////////////
// Lattice_2D::openPositiveFlowFile
////////////////////////////////////////////////////////////////////////
bool
Lattice_2D::openPositiveFlowFile()
{
        positiveFlowFile.open(my_ptr_running_conf->positive_file_name, fstream::app);
        if (!positiveFlowFile.is_open())
        {
                cout << "Cannot open file: " << my_ptr_running_conf->positive_file_name << endl;
                return false;
        }
        return true;
}

////////////////////////////////////////////////////////////////////////
// Lattice_2D::openNegativeFlowFile
////////////////////////////////////////////////////////////////////////
bool
Lattice_2D::openNegativeFlowFile()
{
        negativeFlowFile.open(my_ptr_running_conf->negative_file_name, fstream::app);
        if (!positiveFlowFile.is_open())
        {
                cout << "Cannot open file: " << my_ptr_running_conf->negative_file_name << endl;
                return false;
        }
        return true;
}

////////////////////////////////////////////////////////////////////////
// Lattice_2D::openTotalFlowFile
////////////////////////////////////////////////////////////////////////
bool
Lattice_2D::openTotalFlowFile()
{
        totalFlowFile.open(my_ptr_running_conf->total_file_name, fstream::app);
        if (!totalFlowFile.is_open())
        {
                cout << "Cannot open file: " << my_ptr_running_conf->total_file_name << endl;
                return false;
        }
        return true;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::openAttemptFlowFile
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::openAttemptFlowFile()
{
        attemptFlowFile.open(my_ptr_running_conf->attempt_file_name, fstream::app);
        if (!attemptFlowFile.is_open())
        {
                cout << "Cannot open file: " << my_ptr_running_conf->attempt_file_name << endl;
                return false;
        }
        return true;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::openSummaryFile
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::openSummaryFile()
{

        summaryFile.open(my_ptr_running_conf->summary_file_name, fstream::app);
  if (!summaryFile.is_open())
  {
    cout << "Cannot open file: " << my_ptr_running_conf->summary_file_name << endl;
    return false;
  }
        return true;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::openResultFile
//////////////////////////////////////////////////////////////////////
bool
Lattice_2D::openResultFile()
{

        // TODO: need to get name for configuration parameters
        resultFile.open(my_ptr_running_conf->result_file_name);
  if (!resultFile.is_open())
  {
    cout << "Cannot open result file: ";
                cout << my_ptr_running_conf->result_file_name << endl;
    return false;
  }
        // output the time to the test Log file
  time (&timeheader);
  resultFile << "This data is taken on: " << ctime (&timeheader) << endl << endl;
        return true;
```

```
}

//////////////////////////////////////////////////////////////////////////
// Lattice_2D::openFile
//////////////////////////////////////////////////////////////////////////
bool
Lattice_2D::openFile(char* file_name, ofstream  f_handle)
{

	f_handle.open(file_name);
   if (!f_handle.is_open())
   {
      cout << "Cannot open file: " << file_name << endl;
      return false;
   }
	return true;
}

//////////////////////////////////////////////////////////////////////////
// Lattice_2D::runTest
//////////////////////////////////////////////////////////////////////////
int
Lattice_2D::runTest()
{

	openTestLogFile();
	// clear the lattice
   clear();
	testRandom();
	testMoveRight();
	clear();
	testMoveLeft();
	clear();
	testExchange();
	testSetBondLeftOfCell();
	testSetBondRightOfCell();
	testSetBondTop();
	testSetBondBottom();
	clear();
	testExitParticle();
	clear();
	testEnter();
	clear();
	testAddPositiveParticle();
	clear();
	testAddNegativeParticle();
	clear();
	testLeftBondMovementsPositive();
	clear();
	testLeftBondMovementsEmpty();
	clear();
	testLeftBondMovementsNegative();
	clear();
	testRightBondMovementsNegative();
	clear();
```

```
        testRightBondMovementsEmpty();
        clear();
        testRightBondMovementsPositive();
        clear();
        testVerticalBondMovementsEmpty();
        clear();
        testVerticalBondMovementsSamePolarity();
        clear();
        testVerticalBondMovementsExchange();
        clear();
        testIsColumnEmpty();
        clear();
        testIsVerticalBond();
        clear();
        testIsHorizontalBond();
        clear();
        testFindCluster();
        return 1;

}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testRandom
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testRandom()
{
        int i;

        testLog << "=====  Start Random Number Test ========" << endl;

        for (i=0; i < rows; i++)
        {
                testLog << "[" << i << "] = " << getRandomBetween0AndMax(rows);
                testLog << "\t" << getRandomBetween0AndMax(columns);
          testLog << "\t"  <<  getRandomBetween0And1() << endl;
        }

        testLog << "=====  End Random Number Test ========" << endl;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testMoveRight
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testMoveRight()
{
        lattice[0][0] = POSITIVE_PARTICLE;

        testLog << "Lattice before testMoveRight" << endl;
        moveRight(0,0);
        if (!(lattice[0][0] == EMPTY &&
                   lattice[1][0] == POSITIVE_PARTICLE))
                testLog << "moveRight failed at [0][0]" << endl;

        else testLog << "Test Passed" << endl;
```

```
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::testMoveLeft
////////////////////////////////////////////////////////////////////
void
Lattice_2D::testMoveLeft()
{
	lattice[lattice.size()-1][1] = NEGATIVE_PARTICLE;

	testLog << "Lattice before testMoveLeft" << endl;
	moveLeft(lattice.size()-1,1);
	if (!(lattice[lattice.size()-1][1] == EMPTY &&
		   lattice[lattice.size()-2][1] == NEGATIVE_PARTICLE))
		  testLog << "moveLeft failed at [" << lattice.size()-1 << "][1]" << endl;

	else testLog << "Test Passed" << endl;
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::testExchange
////////////////////////////////////////////////////////////////////
void
Lattice_2D::testExchange()
{
	lattice[3][0] = POSITIVE_PARTICLE;
	lattice[3][1] = NEGATIVE_PARTICLE;
	lattice[299][0] = NEGATIVE_PARTICLE;
	lattice[299][1] = POSITIVE_PARTICLE;
	lattice[125][0] = EMPTY;
	lattice[125][1] = NEGATIVE_PARTICLE;
	lattice[421][0] = POSITIVE_PARTICLE;
	lattice[422][1] = EMPTY;

	testLog << "Lattice before testExchange" << endl;
	printLatticeToTestFile();
	exchange(3,0,3,1);

	if (!(lattice[3][0] == NEGATIVE_PARTICLE &&
		   lattice[3][1] == POSITIVE_PARTICLE))
		  testLog << "exchange failed at [3][0] [3][1]" << endl;
	testLog << "Lattice after testExchange" << endl;

	exchange(299,0,299,1);

	if (!(lattice[299][0] == POSITIVE_PARTICLE &&
		   lattice[299][1] == NEGATIVE_PARTICLE))
		  testLog << "exchange failed at [299][0] [299][1]" << endl;

	exchange(125,0,125,1);

	if (!(lattice[125][0] == NEGATIVE_PARTICLE &&
		   lattice[125][1] == EMPTY))
		  testLog << "exchange failed at [125][0] [125][1]" << endl;
```

```
        exchange(421,0,421,1);

        if (!(lattice[421][0] == EMPTY &&
                 lattice[421][1] == POSITIVE_PARTICLE))
                testLog << "exchange failed at [421][0] [421][1]" << endl;

        printLatticeToTestFile();

}

/////////////////////////////////////////////////////////////////////
// Lattice_2D::testSetBondLeftOfCell
/////////////////////////////////////////////////////////////////////
void
Lattice_2D::testSetBondLeftOfCell()
{
        int x, y;

        x = 0;
        y = 1;
        setBondLeftOfCell(x,y);
        testLog << "setBondLeft Test 1" <<endl;
        testLog << "Cell left of [" << x << "][" << y << "] is ";
        testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;

        x = lattice.size()-1;
        setBondLeftOfCell(x,y);
        testLog << "setBondLeft Test 2" <<endl;
        testLog << "Cell left of [" << x << "][" << y << "] is ";
        testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;

}

/////////////////////////////////////////////////////////////////////
// Lattice_2D::testSetBondRightOfCell
/////////////////////////////////////////////////////////////////////
void
Lattice_2D::testSetBondRightOfCell()
{
        int x, y;

        x = 0;
        y = 1;
        setBondRightOfCell(x,y);
        testLog << "setBondRight Test 1" <<endl;
        testLog << "Cell right of [" << x << "][" << y << "] is ";
        testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;

        x = lattice.size()-1;
        setBondRightOfCell(x,y);
        testLog << "setBondRight Test 2" <<endl;
        testLog << "Cell right of [" << x << "}[" << y << "] is ";
        testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;
}

/////////////////////////////////////////////////////////////////////
```

```
// Lattice_2D::testSetBondTop
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testSetBondTop()
{
	int x, y;

	x = 0;
	y = 1;
	setBondTop(x,y);
	testLog << "setBondTop Test 1" << endl;
	testLog << "Cell in another row  of [" << x << "][" << y << "] is ";
	testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;

	x = 0;
	y = 0;
	setBondTop(x,y);
	testLog << "setBondTop Test 2" << endl;
	testLog << "Cell in another row of [" << x << "][" << y << "] is ";
	testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testSetBondBottom
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testSetBondBottom()
{
	int x, y;

	x = 0;
	y = 1;
	setBondBottom(x,y);
	testLog << "setBondBottom Test 1" << endl;
	testLog << "Cell in another row  of [" << x << "][" << y << "] is ";
	testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;

	x = 0;
	y = 0;
	setBondBottom(x,y);
	testLog << "setBondBottom Test 2" << endl;
	testLog << "Cell in another row of [" << x << "][" << y << "] is ";
	testLog << "[" << x_coord_bond << "][" << y_coord_bond << "]" << endl;
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testExitParticle
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testExitParticle()
{
	x_coord = 0;
	y_coord = 0;
	lattice[0][0] = NEGATIVE_PARTICLE;

	testLog << "Lattice before movement" << endl;
	printLatticeToTestFile();
	exitParticle(x_coord, y_coord);

	if (!(lattice[0][0] == EMPTY))
		testLog << "exitParticle failed" << endl;

	testLog << "Lattice after movement" << endl;
	printLatticeToTestFile();

	x_coord = lattice.size()-1;
	y_coord = 1;
	lattice[lattice.size()-1][1] = POSITIVE_PARTICLE;

	testLog << "Lattice before movement" << endl;
	printLatticeToTestFile();
	exitParticle(x_coord, y_coord);

	if (!(lattice[lattice.size()-1][1] == EMPTY))
		testLog << "exitParticle failed" << endl;

	testLog << "Lattice after movement" << endl;
	printLatticeToTestFile();
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testEnter
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testEnter()
{
	testLog << "Enter Test 1" << endl;
	enter();

	if (isColumnEmpty(0) && isColumnEmpty(lattice.size()-1))
		testLog << "Enter failed" << endl;
	printLatticeToTestFile();
	clear();

	testLog << "Enter Test 2" << endl;
	lattice[0][0] = POSITIVE_PARTICLE;
	lattice[0][1] = NEGATIVE_PARTICLE;
	lattice[lattice.size()-1][0] = POSITIVE_PARTICLE;
	lattice[lattice.size()-1][1] = POSITIVE_PARTICLE;

	enter();

	if (!(lattice[0][0] == POSITIVE_PARTICLE &&
		lattice[0][1] == NEGATIVE_PARTICLE &&
		lattice[lattice.size()-1][0] == POSITIVE_PARTICLE &&
		lattice[lattice.size()-1][1] == POSITIVE_PARTICLE))
		testLog << "Enter 2 failed" << endl;
	printLatticeToTestFile();
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testExitEnds
```

```
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testExitEnds()
{
        testLog << "Exit Ends Test" << endl;
        lattice[0][0] = NEGATIVE_PARTICLE;
        lattice[0][1] = POSITIVE_PARTICLE;
        lattice[lattice.size()-1][0] = POSITIVE_PARTICLE;
        lattice[lattice.size()-1][1] = NEGATIVE_PARTICLE;

        exit_ends();

        if (!(lattice[0][0] == EMPTY &&
                lattice[0][1] == POSITIVE_PARTICLE &&
                lattice[lattice.size()-1][0] == EMPTY &&
                lattice[lattice.size()-1][1] == NEGATIVE_PARTICLE))
                testLog << "Exit Ends failed" << endl;
        printLatticeToTestFile();
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testAddPostiveParticle
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testAddPositiveParticle()
{
        testLog << "Add Particle Test: Positive" << endl;
        x_coord = 0;
        y_coord = 0;
        addPositiveParticle(x_coord,y_coord);
        x_coord = 20;
        y_coord = 1;
        addPositiveParticle(x_coord,y_coord);
        x_coord = lattice.size()-1;
        y_coord = 1;
        addPositiveParticle(x_coord,y_coord);

        if (!(lattice[0][0] == POSITIVE_PARTICLE &&
                lattice[20][1] == POSITIVE_PARTICLE &&
                lattice[lattice.size()-1][1] == POSITIVE_PARTICLE))
                testLog << "Add Particle (pos) failed" << endl;
        printLatticeToTestFile();
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testAddNegativeParticle
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testAddNegativeParticle()
{
        testLog << "Add Particle Test: Negative" << endl;
        x_coord = 0;
        y_coord = 1;
        addNegativeParticle(x_coord,y_coord);
        x_coord = 20;
        y_coord = 0;
```

```
	addNegativeParticle(x_coord,y_coord);
	x_coord = lattice.size()-1;
	y_coord = 0;
	addNegativeParticle(x_coord,y_coord);

	if (!(lattice[0][1] == NEGATIVE_PARTICLE &&
		lattice[20][0] == NEGATIVE_PARTICLE &&
		lattice[lattice.size()-1][0] == NEGATIVE_PARTICLE))
		testLog << "Add Particle (neg) failed" << endl;
	printLatticeToTestFile();
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testLeftBondMovementsPositive
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testLeftBondMovementsPositive()
{
	testLog << "==========Start leftBondMovements Test==========" <<endl;
	testLog << "Testing Positive" <<endl;
	x_coord = 1;
	y_coord = 0;
	lattice[1][0] = POSITIVE_PARTICLE;
	setBondLeftOfCell(x_coord, y_coord);

	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	leftBondMovements();

	if (!(lattice[1][0] == POSITIVE_PARTICLE))
		testLog << "leftBondMovementsPositive failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();

	clear();

	x_coord = 0;
	y_coord = 1;
	lattice[0][1] = POSITIVE_PARTICLE;
	setBondLeftOfCell(x_coord, y_coord);

	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	leftBondMovements();

	if (!(lattice[0][1] == POSITIVE_PARTICLE))
		testLog << "leftBondMovementsPositive failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();

}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testLeftBondMovementsEmpty
//////////////////////////////////////////////////////////////////////
void
```

```
Lattice_2D::testLeftBondMovementsEmpty()
{
	testLog << "=======Testing Empty=======" <<endl;
	testLog << "Test 2 Empty" <<endl;
	x_coord = 1;
	y_coord = 0;
	lattice[1][0] = EMPTY;
	lattice[0][0] = EMPTY;
	setBondLeftOfCell(x_coord, y_coord);

	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	leftBondMovements();

	if (!(lattice[1][0] == EMPTY && lattice[0][0] == EMPTY))
		testLog << "leftBondMovementsEmpty failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();

	clear();

	testLog << "Test 2 Empty ends" <<endl;
	x_coord = 0;
	y_coord = 0;
	lattice[0][0] = EMPTY;
	lattice[499][0] = EMPTY;
	setBondLeftOfCell(x_coord, y_coord);

	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	leftBondMovements();

	if (!(lattice[1][0] == EMPTY && lattice[499][0] == EMPTY))
		testLog << "leftBondMovementsEmpty failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();

	clear();

	testLog << "Test Empty & Neg" <<endl;
	x_coord = 1;
	y_coord = 0;
	lattice[0][0] = NEGATIVE_PARTICLE;
	lattice[1][0] = EMPTY;
	setBondLeftOfCell(x_coord, y_coord);

	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	leftBondMovements();

	if (!(lattice[1][0] == EMPTY && lattice[0][0] == NEGATIVE_PARTICLE))
		testLog << "leftBondMovementsEmpty failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();

	clear();
```

```
        testLog << "Test Empty & Positive" <<endl;
        lattice[1][0] = EMPTY;
        lattice[0][0] =     POSITIVE_PARTICLE;
        setBondLeftOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        leftBondMovements();

        if (!(lattice[1][0] == POSITIVE_PARTICLE && lattice[0][0] == EMPTY))
                testLog << "leftBondMovementsEmpty failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

        clear();

        x_coord = 0;
        y_coord = 0;
        testLog << "Test Empty at end" <<endl;
        lattice[0][0] = EMPTY;
        setBondLeftOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        leftBondMovements();

        if (!(lattice[0][0] == EMPTY))
                testLog << "leftBondMovementsEmpty failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

}

////////////////////////////////////////////////////////////////////
// Lattice_2D::testLeftBondMovementsNegative
////////////////////////////////////////////////////////////////////
void
Lattice_2D::testLeftBondMovementsNegative()
{
        testLog << "==========Testing Negative=========" <<endl;
        testLog << "Test Negative & Empty End" <<endl;
        x_coord = 0;
        y_coord = 0;
        lattice[0][0] = NEGATIVE_PARTICLE;
        lattice[499][0] = EMPTY;
        setBondLeftOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        leftBondMovements();

        if (!(lattice[0][0] == EMPTY && lattice[499][0] == EMPTY))
                testLog << "leftBondMovementsNegative failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();
        clear();

        testLog << "Test Negative & Empty Regular" <<endl;
        x_coord = 1;
        lattice[1][0] = NEGATIVE_PARTICLE;
        lattice[0][0] = EMPTY;
        setBondLeftOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        leftBondMovements();

        if (!(lattice[0][0] == NEGATIVE_PARTICLE && lattice[1][0] == EMPTY))
                testLog << "leftBondMovementsNegative failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();
        testLog << "================End leftBondMovements Test================" <<endl;

}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testRightBondMovementsNegative
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testRightBondMovementsNegative()
{

        testLog << "==========Start rightBondMovements Test==========" <<endl;
        testLog << "Testing Negative" <<endl;
        x_coord = 1;
        y_coord = 0;
        lattice[1][0] = NEGATIVE_PARTICLE;
        setBondRightOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        rightBondMovements();

        if (!(lattice[1][0] == NEGATIVE_PARTICLE))
                testLog << "RightBondMovementsNegative failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

        clear();

        x_coord = 0;
        lattice[0][0] = NEGATIVE_PARTICLE;
        setBondRightOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        rightBondMovements();

        if (!(lattice[0][0] == NEGATIVE_PARTICLE))
                testLog << "RightBondMovementsNegative failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

}

////////////////////////////////////////////////////////////////
// Lattice_2D::testRightBondMovementsEmpty
////////////////////////////////////////////////////////////////
void
Lattice_2D::testRightBondMovementsEmpty()
{
        testLog << "=======Testing Empty=======" <<endl;
        testLog << "Test 2 Empty" <<endl;
        x_coord = 1;
        y_coord = 0;
        lattice[1][0] = EMPTY;
        lattice[0][0] = EMPTY;
        setBondRightOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        rightBondMovements();

        if (!(lattice[1][0] == EMPTY && lattice[0][0] == EMPTY))
                testLog << "rightBondMovementsEmpty failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

        clear();

        testLog << "Test 2 Empty ends" <<endl;
        x_coord = 0;
        y_coord = 0;
        lattice[0][0] = EMPTY;
        lattice[499][0] = EMPTY;
        setBondRightOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        rightBondMovements();

        if (!(lattice[1][0] == EMPTY && lattice[499][0] == EMPTY))
                testLog << "rightBondMovementsEmpty failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

        clear();

        testLog << "Test Empty & Neg" <<endl;
        x_coord = 0;
        y_coord = 0;
        lattice[0][0] = EMPTY;
        lattice[1][0] = NEGATIVE_PARTICLE;
        setBondRightOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        rightBondMovements();

        if (!(lattice[1][0] == EMPTY && lattice[0][0] == NEGATIVE_PARTICLE))
                testLog << "rightBondMovementsEmpty failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

        clear();

        testLog << "Test Empty & Positive" <<endl;
        lattice[1][0] = POSITIVE_PARTICLE;
        lattice[0][0] =     EMPTY;
        setBondRightOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        rightBondMovements();

        if (!(lattice[1][0] == POSITIVE_PARTICLE && lattice[0][0] == EMPTY))
                testLog << "rightBondMovementsEmpty failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

}

////////////////////////////////////////////////////////////////////
// Lattice_2D::testRightBondMovementsPositive
////////////////////////////////////////////////////////////////////
void
Lattice_2D::testRightBondMovementsPositive()
{
        testLog << "========Testing Positive=======" <<endl;
        testLog << "Test Positive & Empty End" <<endl;
        x_coord = 499;
        y_coord = 0;
        lattice[0][0] = EMPTY;
        lattice[499][0] = POSITIVE_PARTICLE;
        setBondRightOfCell(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        rightBondMovements();

        if (!(lattice[0][0] == EMPTY && lattice[499][0] == EMPTY))
                testLog << "rightBondMovementsPositive failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

        clear();

        testLog << "Test Positive & Empty Regular" <<endl;
        x_coord = 0;
        lattice[1][0] = EMPTY;
        lattice[0][0] = POSITIVE_PARTICLE;
        setBondRightOfCell(x_coord, y_coord);
```

```
	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	rightBondMovements();

	if (!(lattice[0][0] == EMPTY && lattice[1][0] == POSITIVE_PARTICLE))
		testLog << "rightBondMovementsPositive failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();
	testLog << "==============End rightBondMovements Test================"
<<endl;

}

///////////////////////////////////////////////////////////////////
// Lattice_2D::testVerticalBondMovementsEmpty
///////////////////////////////////////////////////////////////////
void
Lattice_2D::testVerticalBondMovementsEmpty()
{
	testLog << "===========Start VerticalBondMovements Test============" <<endl;
	testLog << "Test 2 Empty (1)" <<endl;
	x_coord = 0;
	y_coord = 0;
	lattice[0][0] = EMPTY;
	lattice[0][1] = EMPTY;
	setBondTop(x_coord, y_coord);

	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	verticalBondMovements();

	if (!(lattice[0][0] == EMPTY && lattice[0][1] == EMPTY))
		testLog << "verticalBondMovementsEmpty failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();

	clear();
	testLog << "Test 2 Empty (2)" <<endl;
	x_coord = 0;
	y_coord = 0;
	lattice[0][0] = EMPTY;
	lattice[0][1] = EMPTY;
	setBondBottom(x_coord, y_coord);

	testLog << "Lattice before movement" <<endl;
	printLatticeToTestFile();
	verticalBondMovements();

	if (!(lattice[0][0] == EMPTY && lattice[0][1] == EMPTY))
		testLog << "verticalBondMovementsEmpty failed" <<endl;
	testLog << "Lattice after movement" <<endl;
	printLatticeToTestFile();

	clear();
	testLog << "Test 1 Empty 1 Negative (1)" <<endl;
```

```
lattice[0][0] = EMPTY;
lattice[0][1] = NEGATIVE_PARTICLE;
setBondTop(x_coord, y_coord);

testLog << "Lattice before movement" <<endl;
printLatticeToTestFile();
verticalBondMovements();

if (!(lattice[0][0] == EMPTY && lattice[0][1] == NEGATIVE_PARTICLE))
            testLog << "verticalBondMovementsEmpty failed" <<endl;
testLog << "Lattice after movement" <<endl;
printLatticeToTestFile();

clear();
testLog << "Test 1 Empty 1 Negative (2)" <<endl;
lattice[0][0] = EMPTY;
lattice[0][1] = NEGATIVE_PARTICLE;
setBondBottom(x_coord, y_coord);

testLog << "Lattice before movement" <<endl;
printLatticeToTestFile();
verticalBondMovements();

if (!(lattice[0][0] == EMPTY && lattice[0][1] == NEGATIVE_PARTICLE))
            testLog << "verticalBondMovementsEmpty failed" <<endl;
testLog << "Lattice after movement" <<endl;
printLatticeToTestFile();

clear();
testLog << "Test 1 Empty 1 Positive (1)" <<endl;
lattice[0][0] = POSITIVE_PARTICLE;
lattice[0][1] = EMPTY;
setBondTop(x_coord, y_coord);

testLog << "Lattice before movement" <<endl;
printLatticeToTestFile();
verticalBondMovements();

if (!(lattice[0][0] == POSITIVE_PARTICLE && lattice[0][1] == EMPTY))
            testLog << "verticalBondMovementsEmpty failed" <<endl;
testLog << "Lattice after movement" <<endl;
printLatticeToTestFile();

clear();
testLog << "Test 1 Empty 1 Positive (2)" <<endl;
lattice[0][0] = POSITIVE_PARTICLE;
lattice[0][1] = EMPTY;
setBondBottom(x_coord, y_coord);

testLog << "Lattice before movement" <<endl;
printLatticeToTestFile();
verticalBondMovements();

if (!(lattice[0][0] == POSITIVE_PARTICLE && lattice[0][1] == EMPTY))
            testLog << "verticalBondMovementsEmpty failed" <<endl;
testLog << "Lattice after movement" <<endl;
```

```
        printLatticeToTestFile();
}

/////////////////////////////////////////////////////////////////
// Lattice_2D::testVerticalBondMovementsSamePolarity
/////////////////////////////////////////////////////////////////
void
Lattice_2D::testVerticalBondMovementsSamePolarity()
{
        testLog << "Test Same Polarity Positive" <<endl;
        x_coord = 0;
        y_coord = 0;
        lattice[0][0] = POSITIVE_PARTICLE;
        lattice[0][1] =     POSITIVE_PARTICLE;
        setBondTop(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        verticalBondMovements();

        if (!(lattice[0][0] == POSITIVE_PARTICLE && lattice[0][1] == POSITIVE_PARTICLE))
                testLog << "verticalBondMovementsSamePolarity failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();

        clear();
        y_coord = 1;
        testLog << "Test Same Polarity Negative" <<endl;
        lattice[0][0] = NEGATIVE_PARTICLE;
        lattice[0][1] =     NEGATIVE_PARTICLE;
        setBondBottom(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
        printLatticeToTestFile();
        verticalBondMovements();

        if (!(lattice[0][0] == NEGATIVE_PARTICLE && lattice[0][1] == NEGATIVE_PARTICLE))
                testLog << "verticalBondMovementsSamePolarity failed" <<endl;
        testLog << "Lattice after movement" <<endl;
        printLatticeToTestFile();
}

/////////////////////////////////////////////////////////////////
// Lattice_2D:testVerticalBondMovementsExchange
/////////////////////////////////////////////////////////////////
void
Lattice_2D::testVerticalBondMovementsExchange()
{
        testLog << "Test Exchange Negative over End" <<endl;
        x_coord = 0;
        y_coord = 0;
        lattice[0][0] = POSITIVE_PARTICLE;
        lattice[0][1] = NEGATIVE_PARTICLE;
        setBondTop(x_coord, y_coord);

        testLog << "Lattice before movement" <<endl;
```

```
		printLatticeToTestFile();
		verticalBondMovements();

		if (!(lattice[1][1] == POSITIVE_PARTICLE && lattice[499][0] == NEGATIVE_PARTICLE))
				testLog << "verticalBondMovementsExchange failed" <<endl;
		testLog << "Lattice after movement" <<endl;
		printLatticeToTestFile();

		clear();
		testLog << "Test Exchange No End" <<endl;
		x_coord = 1;
		lattice[1][0] = NEGATIVE_PARTICLE;
		lattice[1][1] = POSITIVE_PARTICLE;
		setBondTop(x_coord, y_coord);

		testLog << "Lattice before movement" <<endl;
		printLatticeToTestFile();
		verticalBondMovements();

		if (!(lattice[0][1] == NEGATIVE_PARTICLE && lattice[2][0] == POSITIVE_PARTICLE))
				testLog << "verticalBondMovementsExchange failed" <<endl;
		testLog << "Lattice after movement" <<endl;
		printLatticeToTestFile();

		clear();
		testLog << "Test Exchange Positive over End" <<endl;
		x_coord = 499;
		y_coord = 1;
		lattice[499][0] = NEGATIVE_PARTICLE;
		lattice[499][1] = POSITIVE_PARTICLE;
		setBondBottom(x_coord, y_coord);

		testLog << "Lattice before movement" <<endl;
		printLatticeToTestFile();
		verticalBondMovements();

		if (!(lattice[498][1] == NEGATIVE_PARTICLE && lattice[0][0] == POSITIVE_PARTICLE))
				testLog << "verticalBondMovementsExchange failed" <<endl;
		testLog << "Lattice after movement" <<endl;
		printLatticeToTestFile();
}

//////////////////////////////////////////////////////////////////////
// Lattice_2D::testIsColumnEmpty
//////////////////////////////////////////////////////////////////////
void
Lattice_2D::testIsColumnEmpty()
{
		testLog << "Test of method isColumnIsEmpty() " <<endl;
   if (isColumnEmpty(0))
		{
				testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
				testLog << " is empty" << endl;
		}

		lattice[0][0] = POSITIVE_PARTICLE;
```

```
        if (!isColumnEmpty(0))
        {
                testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
                testLog << " is not empty" << endl;
        }

        lattice[0][1] = POSITIVE_PARTICLE;
        if (!isColumnEmpty(0))
        {
                testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
                testLog << " is not empty" << endl;
        }

        lattice[0][0] = EMPTY;
        if (!isColumnEmpty(0))
        {
                testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
                testLog << " is not empty" << endl;
        }
}

/////////////////////////////////////////////////////////////////////////
// Lattice_2D::testIsVerticalBond
/////////////////////////////////////////////////////////////////////////
void
Lattice_2D::testIsVerticalBond()
{
        testLog << endl << "Test of method isVerticalBond() " <<endl;
  if (!isVerticalBond(0))
        {
                testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
                testLog << " does not have a vertical bond" << endl;
        }

        lattice[0][0] = POSITIVE_PARTICLE;
        if (!isVerticalBond(0))
        {
                testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
                testLog << " does not have a vertical bond" << endl;
        }

        lattice[0][1] = POSITIVE_PARTICLE;
        if (isVerticalBond(0))
        {
                testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
                testLog << " does have a vertical bond" << endl;
        }

        lattice[0][0] = EMPTY;
        if (!isVerticalBond(0))
        {
                testLog << "column with values " << lattice[0][0] << "," << lattice[0][1];
                testLog << " does not have a vertical bond" << endl;
        }
}
```

```
////////////////////////////////////////////////////////////////////
// Lattice_2D::testIsHorizontalBond
////////////////////////////////////////////////////////////////////
void
Lattice_2D::testIsHorizontalBond()
{
        testLog << endl << "Test of method isHorizontalBond() " <<endl;
   if (!isHorizontalBond(0,0))
        {
                testLog << "row with values " << lattice[0][0] << "," << lattice[1][0];
                testLog << " does not have a horizontal bond" << endl;
        }

        lattice[1][0] = POSITIVE_PARTICLE;
        if (!isHorizontalBond(0,0))
        {
                testLog << "row with values " << lattice[0][0] << "," << lattice[1][0];
                testLog << " does not have a horizontal bond" << endl;
        }

        lattice[0][0] = POSITIVE_PARTICLE;
        if (isHorizontalBond(0,0))
        {
                testLog << "row with values " << lattice[0][0] << "," << lattice[1][0];
                testLog << " does have a horizontal bond" << endl;
        }

        lattice[1][0] = EMPTY;
        if (!isHorizontalBond(0,0))
        {
                testLog << "row with values " << lattice[0][0] << "," << lattice[1][0];
                testLog << " does not have a horizontal bond" << endl;
        }
}

////////////////////////////////////////////////////////////////////
// Lattice_2D::testFindCluster
////////////////////////////////////////////////////////////////////
void
Lattice_2D::testFindCluster()
{

        testLog << endl << "Test of findCluster method" << endl;
        testLog <<endl;
        testLog << "Test 1: Start and Middle" <<endl;
        testLog << "Should show one cluster of 2 at 0 and one cluster of 5 at 250/252" <<endl;
        // add one cluster at the start of the lattice
        lattice[0][0] = POSITIVE_PARTICLE;
        lattice[1][0] = NEGATIVE_PARTICLE;

     //add another cluster in the middle of the lattice
        lattice[250][0] = NEGATIVE_PARTICLE;
        lattice[250][1] = POSITIVE_PARTICLE;
        lattice[251][1] = POSITIVE_PARTICLE;
        lattice[252][0] = POSITIVE_PARTICLE;
        lattice[252][1] = NEGATIVE_PARTICLE;
```

```
findClusters();
logClusters();

testLog << "Test 2: End (one Cluster)" <<endl;
testLog << "Should show one cluster of 4 at the end of lattice" <<endl;

//add one cluster at the end of the lattice
lattice[(lattice.size()-3)][0] = POSITIVE_PARTICLE;
lattice[(lattice.size()-2)][0] = POSITIVE_PARTICLE;
lattice[(lattice.size()-1)][0] = POSITIVE_PARTICLE;
lattice[(lattice.size()-1)][1] = NEGATIVE_PARTICLE;

findClusters();
logClusters();

testLog << "Test 3: End 2 (one Cluster) " <<endl;
testLog << "Should show one cluster of 2 at the end of lattice" <<endl;

//add one cluster at the end of the lattice
lattice[(lattice.size()-1)][0] = POSITIVE_PARTICLE;
lattice[(lattice.size()-1)][1] = NEGATIVE_PARTICLE;

findClusters();
logClusters();

clear();
testLog << "Test 4: Connect Clusters at both ends" <<endl;
testLog << "Should show one cluster starting lattice end and ending at lattice beginning/5 particles"
        <<endl;
//add one cluster at the beginning of the lattice
lattice[0][1] = POSITIVE_PARTICLE;
lattice[1][1] = NEGATIVE_PARTICLE;
lattice[1][0] = NEGATIVE_PARTICLE;

//add one cluster at the end of the lattice
lattice[(lattice.size()-1)][0] = POSITIVE_PARTICLE;
lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

findClusters();
logClusters();

testLog << "Test 5: Cluster at one end and single particle at other end (one Cluster)" <<endl;
testLog << "Test 1" <<endl;
testLog << "Should show one cluster of 3, starting at the end of the lattice" <<endl;

//add one cluster at the beginning of the lattice
lattice[0][0] = NEGATIVE_PARTICLE;
lattice[0][1] = POSITIVE_PARTICLE;

//add one particle at the end of the lattice
lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;


findClusters();
logClusters();
```

```
        testLog << "Test 6: Cluster at end single particle at the beginning (one Cluster)" <<endl;
        testLog << "Should show one cluster of 4, starting near the end of the lattice" <<endl;

        //add one cluster at the end of the lattice
        lattice[(lattice.size()-2)][0] = POSITIVE_PARTICLE;
        lattice[(lattice.size()-1)][0] = NEGATIVE_PARTICLE;
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

        //add one particle at the begginning of the lattice
        lattice[0][0] = NEGATIVE_PARTICLE;

        findClusters();
        logClusters();

        testLog << "Test 7: Connect single particles at both ends (no Cluster)" <<endl;
        testLog << "Should show a cluster of two, starting at end of lattice" <<endl;

        //add one particle at the beginning of the lattice
        lattice[0][0] = POSITIVE_PARTICLE;

        //add one particle at the end of the lattice
        lattice[(lattice.size()-1)][0] = NEGATIVE_PARTICLE;

        findClusters();
        logClusters();

        testLog << "Test 8: Do not connect single particles at ends (no Cluster) " <<endl;
        testLog << "Should show no cluster" <<endl;

        //add one particle at the beginning of the lattice
        lattice[0][0] = NEGATIVE_PARTICLE;

        //add one particle at the end of the lattice where no horizontal bond exists
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;


        findClusters();
        logClusters();

        testLog << "Test 9: Connect single particles at both ends (one Cluster)" <<endl;
        testLog << "Should show two clusters, one of two at the end of the lattice and one of 3 in the middle"
<<endl;

        //add one particle at the beginning of the lattice
        lattice[0][1] = NEGATIVE_PARTICLE;

        //add one particle at the end of the lattice
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

        //add cluster in middle
        lattice[244][0] = NEGATIVE_PARTICLE;
        lattice[245][0] = POSITIVE_PARTICLE;
        lattice[246][0] = NEGATIVE_PARTICLE;

        findClusters();
        logClusters();

        testLog << "Test 10: Do not connect single particles at ends (one Cluster) " <<endl;
        testLog << "Should show one cluster in middle of size 3" <<endl;

        //add one particle at the beginning of the lattice
        lattice[0][0] = NEGATIVE_PARTICLE;

        //add one particle at the end of the lattice where no horizontal bond exists
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

        //add cluster in middle
        lattice[341][1] = NEGATIVE_PARTICLE;
        lattice[341][0] = POSITIVE_PARTICLE;
        lattice[340][1] = POSITIVE_PARTICLE;


        findClusters();
        logClusters();

        testLog << "Test 11: Connect single particles at both ends (multiple Clusters)" <<endl;
        testLog << "Should show four clusters, one of two at the end of the lattice and three others of 2,3 and 5 in
the middle" <<endl;

        //add one particle at the beginning of the lattice
        lattice[0][1] = NEGATIVE_PARTICLE;

        //add one particle at the end of the lattice
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

        //add clusters in middle
        lattice[244][0] = NEGATIVE_PARTICLE;
        lattice[245][0] = POSITIVE_PARTICLE;
        lattice[246][0] = NEGATIVE_PARTICLE;
        lattice[12][1] = POSITIVE_PARTICLE;
        lattice[11][1] = NEGATIVE_PARTICLE;
        lattice[122][0] = POSITIVE_PARTICLE;
        lattice[122][1] = NEGATIVE_PARTICLE;
        lattice[123][1] = NEGATIVE_PARTICLE;
        lattice[124][1] = NEGATIVE_PARTICLE;
        lattice[124][0] = NEGATIVE_PARTICLE;

        findClusters();
        logClusters();

        testLog << "Test 12: Do not connect single particles at ends (multiple Clusters) " <<endl;
        testLog << "Should show 3 clusters in middle of sizes 2,3 and 5" <<endl;

        //add one particle at the beginning of the lattice
        lattice[0][0] = NEGATIVE_PARTICLE;

        //add one particle at the end of the lattice where no horizontal bond exists
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

        //add clusters in middle
        lattice[341][1] = NEGATIVE_PARTICLE;
        lattice[341][0] = POSITIVE_PARTICLE;
        lattice[340][1] = POSITIVE_PARTICLE;
        lattice[12][1] = POSITIVE_PARTICLE;
        lattice[11][1] = NEGATIVE_PARTICLE;
        lattice[122][0] = POSITIVE_PARTICLE;
        lattice[122][1] = NEGATIVE_PARTICLE;
        lattice[123][1] = NEGATIVE_PARTICLE;
        lattice[124][1] = NEGATIVE_PARTICLE;
        lattice[124][0] = NEGATIVE_PARTICLE;



        findClusters();
        logClusters();

        testLog << "Test 13: Cluster at end single particle at the beginning (multiple Clusters)" <<endl;
        testLog << "Should show one cluster of 4, starting near the end of the lattice and 3 other clusters in the
middle" <<endl;

        //add one cluster at the end of the lattice
        lattice[(lattice.size()-2)][0] = POSITIVE_PARTICLE;
        lattice[(lattice.size()-1)][0] = NEGATIVE_PARTICLE;
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

        //add one particle at the begginning of the lattice
        lattice[0][0] = NEGATIVE_PARTICLE;

        //add clusters in middle
        lattice[341][1] = NEGATIVE_PARTICLE;
        lattice[341][0] = POSITIVE_PARTICLE;
        lattice[340][1] = POSITIVE_PARTICLE;
        lattice[12][1] = POSITIVE_PARTICLE;
        lattice[11][1] = NEGATIVE_PARTICLE;
        lattice[122][0] = POSITIVE_PARTICLE;
        lattice[122][1] = NEGATIVE_PARTICLE;
        lattice[123][1] = NEGATIVE_PARTICLE;
        lattice[124][1] = NEGATIVE_PARTICLE;
        lattice[124][0] = NEGATIVE_PARTICLE;

        findClusters();
        logClusters();

        testLog << "Final Test 14: Cluster at one end and single particle at other end (multiple Clusters)" <<endl;
        testLog << "Test 1" <<endl;
        testLog << "Should show one cluster of 3, starting at the end of the lattice and 3 others clusters in the
middle" <<endl;

        //add one cluster at the beginning of the lattice
        lattice[0][0] = NEGATIVE_PARTICLE;
        lattice[0][1] = POSITIVE_PARTICLE;

        //add one particle at the end of the lattice
        lattice[(lattice.size()-1)][1] = POSITIVE_PARTICLE;

        //add clusters in middle
        lattice[341][1] = NEGATIVE_PARTICLE;
        lattice[341][0] = POSITIVE_PARTICLE;
        lattice[340][1] = POSITIVE_PARTICLE;
        lattice[12][1] = POSITIVE_PARTICLE;
        lattice[11][1] = NEGATIVE_PARTICLE;
        lattice[122][0] = POSITIVE_PARTICLE;
        lattice[122][1] = NEGATIVE_PARTICLE;
        lattice[123][1] = NEGATIVE_PARTICLE;
        lattice[124][1] = NEGATIVE_PARTICLE;
        lattice[124][0] = NEGATIVE_PARTICLE;


        findClusters();
        logClusters();

}

/////////////////////////////////////////////////////////////////////
// Lattice_2D::logClusters
/////////////////////////////////////////////////////////////////////
void
Lattice_2D::logClusters()
{
        int i;
        list<Cluster*>::iterator clusterPtr;

        testLog << "Number of cluster found = " << myClusterList.size() << endl;
        i=1;
        for (clusterPtr = myClusterList.begin(); clusterPtr != myClusterList.end(); clusterPtr++)
        {
         testLog << " === Cluster [" << i << "]:" << endl;
    testLog << "Number of particles: " << (*clusterPtr)->getNumberOfParticles() << endl;
         testLog << "Start position: " << (*clusterPtr)->getStartPosition() << endl;
         testLog << "End   position: " << (*clusterPtr)->getEndPosition() << endl;
         testLog << "=====" << endl;
         i++;
        }
        myClusterList.clear();
        clear();
}

/////////////////////////////////////////////////////////////////////
// Lattice_2D::resultClusters()
/////////////////////////////////////////////////////////////////////
void
Lattice_2D::resultClusters()
{
        int i;
        list<Cluster*>::iterator clusterPtr;

        total_enter = enter_positive + enter_negative;
        total_exit = exit_positive + exit_negative;
        total_attach = attach_positive + attach_negative;
        total_detach = detach_positive + detach_negative;
        findBiggestCluster();

        testLog << "\nEnter\tExit\tAtt.\tDet.\n" << endl;
        resultFile << "Number of cluster found = " << myClusterList.size() << endl;
        i=1;
        for (clusterPtr = myClusterList.begin(); clusterPtr != myClusterList.end(); clusterPtr++)
        {
         resultFile << " === Cluster [" << i << "]:" << endl;
         resultFile << "Number of particles: " << (*clusterPtr)->getNumberOfParticles() << endl;
         resultFile << "Start position: " << (*clusterPtr)->getStartPosition() << endl;
         resultFile << "End   position: " << (*clusterPtr)->getEndPosition() << endl;
         resultFile << "=====" << endl;
         i++;
        }
        if (myClusterList.size() > 0) {
                resultFile << "=== Biggest cluster ===" << endl;
                resultFile << "Number of particles: " << (*biggestClusterPtr)->getNumberOfParticles() << endl;
                resultFile << "Start position: " << (*biggestClusterPtr)->getStartPosition() << endl;
                resultFile << "End   position: " << (*biggestClusterPtr)->getEndPosition() << endl;
        }

        // print flow results to appropriate files
        if(openPositiveFlowFile())
        {
                // gamma value | size of lattice | enter | exit | attach | detach
                positiveFlowFile << my_ptr_running_conf->g << "\t" << my_ptr_running_conf->rows << "\t";
                positiveFlowFile << enter_positive << "\t" << exit_positive << "\t";
                positiveFlowFile << attach_positive << "\t" << detach_positive << endl;
//              add positive current above
        }

        if(openNegativeFlowFile())
        {
                // gamma value | size of lattice | enter | exit | attach | detach
                negativeFlowFile << my_ptr_running_conf->g << "\t" << my_ptr_running_conf->rows << "\t";
                negativeFlowFile << enter_negative << "\t" << exit_negative << "\t";
                negativeFlowFile << attach_negative << "\t" << detach_negative << endl;
//              add negative current above
        }
        if(openAttemptFlowFile())
        {
                // gamma value | size of lattice | enter | exit | attach | detach
                attemptFlowFile << my_ptr_running_conf->g << "\t" << my_ptr_running_conf->rows << "\t";
                attemptFlowFile << att_e << "\t" << att_ex << endl;
        }
        if(openTotalFlowFile())
        {
                // gamma value | size of lattice | enter | exit | attach | detach
                totalFlowFile << my_ptr_running_conf->g << "\t" << my_ptr_running_conf->rows << "\t";
                totalFlowFile << total_enter << "\t" << total_exit << "\t";
                totalFlowFile << total_attach << "\t" << total_detach << endl;
//              add total current above
        }

        // print result to to summary file
        if (openSummaryFile()) {
                // gamma value | size of lattice | number of clusters | biggest cluster # particles
                summaryFile << my_ptr_running_conf->g << "\t" << my_ptr_running_conf->rows << "\t";
                summaryFile << myClusterList.size() << "\t";
```

```
			if (myClusterList.size() == 0) {
				//summaryFile << "0 \t 0 \t 0" << endl;
			}
			else {
				//summaryFile << (*biggestClusterPtr)->getStartPosition() << "\t";
				//summaryFile << (*biggestClusterPtr)->getEndPosition() << "\t";
				summaryFile << (*biggestClusterPtr)->getNumberOfParticles() << endl;
//				change above endline to tab
//				summaryFile << total_positive << "\t" << total_negative << endl;
			}
		}
		myClusterList.clear();
		clear();
}
```

```
/***********************************************************/
//    Cluster.h:
//
// This file contains structure and class definitions of Cluster class
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/***********************************************************/

#ifndef _Cluster_H
#define _Cluster_H

class Cluster
{
private:
        int      start_position_x;
        int      end_position_x;
        int      totalNumberOfParticles;


public:
                                        Cluster(int x);
        virtual                    ~Cluster();

        void     setEndPosition(int x);
        void     setStartPosition(int x);
        int      getStartPosition();
        int      getEndPosition();
        void     addParticlesToCluster(int number);
        int      getNumberOfParticles();

};

/***********************************************************/
//       Do not place anything after endif
/***********************************************************/
#endif _Cluster_H
```

```
/**********************************************************/
//     Cluster.cpp
//
// This file contains the implementation of the Cluster class
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/**********************************************************/

#include "Cluster.h"

//////////////////////////////////////////////////////////////////////
// Cluster::Constructor
//////////////////////////////////////////////////////////////////////

Cluster::Cluster(int x)
{
        start_position_x = x;
        totalNumberOfParticles = 0;
}

//////////////////////////////////////////////////////////////////////
// Cluster::Destructor
//////////////////////////////////////////////////////////////////////

Cluster::~Cluster()
{

}

//////////////////////////////////////////////////////////////////////
// Cluster::setEndPosition
//////////////////////////////////////////////////////////////////////
void
Cluster::setEndPosition(int x)
{
        end_position_x = x;
}

//////////////////////////////////////////////////////////////////////
// Cluster::setStartPosition
//////////////////////////////////////////////////////////////////////
void
Cluster::setStartPosition(int x)
{
        start_position_x = x;
}

//////////////////////////////////////////////////////////////////////
// Cluster::getEndPosition
//////////////////////////////////////////////////////////////////////
int
Cluster::getEndPosition()
{
        return end_position_x;
}

////////////////////////////////////////////////////////////////////
// Cluster::getStartPosition
////////////////////////////////////////////////////////////////////
int
Cluster::getStartPosition()
{
	return start_position_x;
}

////////////////////////////////////////////////////////////////////
// Cluster::addParticlesToCluster
////////////////////////////////////////////////////////////////////
void
Cluster::addParticlesToCluster(int number)
{
	totalNumberOfParticles += number;
}

////////////////////////////////////////////////////////////////////
// Cluster::getNumberOfParticles
////////////////////////////////////////////////////////////////////
int
Cluster::getNumberOfParticles()
{
	return totalNumberOfParticles;
}
```

APPENDIX B: C++ CODE FOR AVERAGE RESULTS PROGRAM

```
/***********************************************************/
//    Read Summary.cpp
//
// This file contains code to average the results of multiple Monte Carlo
//  trials of Lattice_2D simulation
//
// Author: Josh Gonzalez
// Date:   June/11/2008
//
/***********************************************************/

#include <fstream>
#include <cmath>

using namespace std;

int main(int argc,char *argv[])
{
        float gamma;
        int size;
        int number_of_clusters;
        int size_of_biggest_clusters;
        int enter;
        int exit;
        int attach;
        int detach;
        int total_positive;
        int total_negative;
        int positive_current;
        int negative_current;
        int total_current;
        int sum_total_positive;
        int sum_total_negative;
        int sum_positive_current;
        int sum_negative_current;
        int sum_total_current;
        int sum_number_of_clusters;
        int sum_size_of_biggest_clusters;
        int sum_enter;
        int sum_exit;
        int sum_attach;
        int sum_detach;
        double averageenter;
        double averageexit;
        double averageattach;
        double averagedetach;
        double averagenumber;
        double averagesize;
        double averagePositive;
        double averageNegative;
        double averagepositivecurrent;
        double averagenegativecurrent;
        double averagetotalcurrent;

   double num_runs;
        char *c1;
```

```
	char *summary_location;
	char *average_location;
	char *attempt_location;
	char *positive_location;
	char *negative_location;
	char *total_location;
	char *average_attempt;
	char *average_positive;
	char *average_negative;
	char *average_total;
	int i;

	printf("argv[1]: %s\n",argv[1]);
   c1 = argv[1];
   num_runs = atoi(c1);

	summary_location = argv[2];
	//printf("summary_location %s\n",summary_location);
	average_location = argv[3];
	attempt_location = argv[4];
	positive_location = argv[5];
	negative_location = argv[6];
	total_location = argv[7];
	average_attempt = argv[8];
	average_positive = argv[9];
	average_negative = argv[10];
	average_total = argv[11];


	ofstream averagefile;
	averagefile.open(average_location);
	averagefile << "Averages File \n#Runs per Gamma value =" << num_runs << endl;
	averagefile << "Gamma\tSize\tAvg#Clst.\tAvgSizeBigClst.\ttotalpos.\ttotalneg." << endl;
//	add total positive and total negative above

	ofstream averageattempt;
	averageattempt.open(average_attempt);
	averageattempt << "Average Attempt\n#Runs per Gamma value = " << num_runs << endl;
	averageattempt << "Gamma\tSize\tEnter\tExit\tAttach\tDetach" << endl;

	ofstream averagepositive;
	averagepositive.open(average_positive);
	averagepositive << "Average Positive\n#Runs per Gamma value = " << num_runs << endl;
	averagepositive << "Gamma\tSize\tEnter\tExit\tAttach\tDetach" << endl;

	ofstream averagenegative;
	averagenegative.open(average_negative);
	averagenegative << "Average negative\n#Runs per Gamma value = " << num_runs << endl;
	averagenegative << "Gamma\tSize\tEnter\tExit\tAttach\tDetach" << endl;

	ofstream averagetotal;
	averagetotal.open(average_total);
	averagetotal << "Average total\n#Runs per Gamma value = " << num_runs << endl;
	averagetotal << "Gamma\tSize\tEnter\tExit\tAttach\tDetach" << endl;

	FILE * summaryFile;
```

```
        summaryFile = fopen(summary_location,"r");
        if (summaryFile == NULL)   //if pointer does not exist(can't open file) prints error message
        {
                printf("ERROR: unable to open summary.txt\n");

  }

        sum_number_of_clusters = 0;
        sum_size_of_biggest_clusters = 0;
        sum_total_positive = 0;
        sum_total_negative = 0;
  i=0;

        while(fscanf(summaryFile,"%f %d %d %d %d
%d",&gamma,&size,&number_of_clusters,&size_of_biggest_clusters,&total_positive,&total_negative)!=EOF)
        {
                sum_number_of_clusters += number_of_clusters;
                sum_size_of_biggest_clusters += size_of_biggest_clusters;
                sum_total_positive += total_positive;
                sum_total_negative += total_negative;
                //printf(" [%d] = %f %d %d %d\n", i, gamma, size,
number_of_clusters,size_of_biggest_clusters);
                i++;
                if ( i == num_runs ) {
                        averagenumber = sum_number_of_clusters/num_runs;
                        averagesize = sum_size_of_biggest_clusters/num_runs;
                        averagePositive = sum_total_positive/num_runs;
                        averageNegative = sum_total_negative/num_runs;
                        averagefile << gamma << "\t" << size << "\t" << averagenumber << "\t" << averagesize
<< "\t";

                        averagefile << averagePositive << "\t" << averageNegative << endl;
                        sum_number_of_clusters = 0;
                        sum_size_of_biggest_clusters = 0;
                        sum_total_positive = 0;
                        sum_total_negative = 0;
                        i = 0;
                }
        }

        printf("done reading file\n\n");

        fclose(summaryFile);

        averagefile.close();

        FILE * attemptFile;
        attemptFile = fopen(attempt_location,"r");
        if (attemptFile == NULL)   //if pointer does not exist(can't open file) prints error message
        {
                printf("ERROR: unable to open attemptflow.txt\n");

  }

        sum_enter = 0;
        sum_exit = 0;
        i=0;
        while(fscanf(attemptFile,"%f %d %d %d",&gamma,&size,&enter,&exit)!=EOF)
        {
                sum_enter += enter;
                sum_exit += exit;
                i++;
                if(i == num_runs)
                {
                        averageenter = sum_enter/num_runs;
                        averageexit = sum_exit/num_runs;
                        averageattempt << gamma << "\t" << size << "\t" << averageenter << "\t" <<
averageexit << endl;
                        i = 0;
                        sum_enter = 0;
                        sum_exit = 0;
                }
        }
        printf("done reading file\n\n");
        fclose(attemptFile);
        averageattempt.close();

        FILE * positiveFile;
        positiveFile = fopen(positive_location,"r");
        if (positiveFile == NULL)   //if pointer does not exist(can't open file) prints error message
        {
                printf("ERROR: unable to open positiveflow.txt\n");

   }

        sum_enter = 0;
        sum_exit = 0;
        sum_attach = 0;
        sum_detach = 0;
        sum_positive_current = 0;
        i = 0;

        while(fscanf(positiveFile,"%f %d %d %d %d %d
%d",&gamma,&size,&enter,&exit,&attach,&detach,&positive_current)!=EOF)
        {
                sum_enter += enter;
                sum_exit += exit;
                sum_attach += attach;
                sum_detach += detach;
                sum_positive_current += positive_current;
                i++;
                if(i == num_runs)
                {
                        averageenter = sum_enter/num_runs;
                        averageexit = sum_exit/num_runs;
                        averageattach = sum_attach/num_runs;
                        averagedetach = sum_detach/num_runs;
                        averagepositivecurrent = sum_positive_current/num_runs;
                        averagepositive << gamma << "\t" << size << "\t" << averageenter << "\t" <<
averageexit << "\t" << averageattach << "\t";
                        averagepositive << averagedetach << "\t" << averagepositivecurrent << endl;
                        i = 0;
                        sum_enter = 0;
                        sum_exit = 0;
                        sum_attach = 0;
                        sum_detach = 0;
                        sum_positive_current = 0;
                }
        }
        printf("done reading file\n\n");
        fclose(positiveFile);
        averagepositive.close();

        FILE * negativeFlowFile;
        negativeFlowFile = fopen(negative_location,"r");
        if (negativeFlowFile == NULL)   //if pointer does not exist(can't open file) prints error message
        {
                printf("ERROR: unable to open negativeflow.txt\n");

   }

        sum_enter = 0;
        sum_exit = 0;
        sum_attach = 0;
        sum_detach = 0;
        sum_negative_current = 0;
        i = 0;

        while(fscanf(negativeFlowFile,"%f %d %d %d %d %d
%d",&gamma,&size,&enter,&exit,&attach,&detach,&negative_current)!=EOF)
        {
                sum_enter += enter;
                sum_exit += exit;
                sum_attach += attach;
                sum_detach += detach;
                sum_negative_current += negative_current;
                i++;
                if(i == num_runs)
                {
                        averageenter = sum_enter/num_runs;
                        averageexit = sum_exit/num_runs;
                        averageattach = sum_attach/num_runs;
                        averagedetach = sum_detach/num_runs;
                        averagenegativecurrent = sum_negative_current/num_runs;
                        averagenegative << gamma << "\t" << size << "\t" << averageenter << "\t" <<
averageexit << "\t" << averageattach << "\t";
                        averagenegative << averagedetach << "\t" << averagenegativecurrent << endl;
                        i = 0;
                        sum_enter = 0;
                        sum_exit = 0;
                        sum_attach = 0;
                        sum_detach = 0;
                        sum_negative_current = 0;
                }
        }
        printf("done reading file\n\n");
        fclose(negativeFlowFile);
        averagenegative.close();
        FILE * totalFlowFile;
        totalFlowFile = fopen(total_location,"r");
        if (totalFlowFile == NULL)   //if pointer does not exist(can't open file) prints error message
        {
                printf("ERROR: unable to open totalflow.txt\n");

  }

        sum_enter = 0;
        sum_exit = 0;
        sum_attach = 0;
        sum_detach = 0;
        sum_total_current = 0;
        i=0;

        while(fscanf(totalFlowFile,"%f %d %d %d %d %d
%d",&gamma,&size,&enter,&exit,&attach,&detach,&total_current)!=EOF)
        {
                sum_enter += enter;
                sum_exit += exit;
                sum_attach += attach;
                sum_detach += detach;
                sum_total_current += total_current;
                i++;
                if(i == num_runs)
                {
                        averageenter = sum_enter/num_runs;
                        averageexit = sum_exit/num_runs;
                        averageattach = sum_attach/num_runs;
                        averagedetach = sum_detach/num_runs;
                        averagetotalcurrent = sum_total_current/num_runs;
                        averagetotal << gamma << "\t" << size << "\t" << averageenter << "\t" << averageexit
<< "\t" << averageattach << "\t";
                        averagetotal << averagedetach << "\t" << averagetotalcurrent << endl;
                        i = 0;
                        sum_enter = 0;
                        sum_exit = 0;
                        sum_attach = 0;
                        sum_detach = 0;
                        sum_total_current = 0;
                }
        }
        printf("done reading file\n\n");
        fclose(totalFlowFile);
        averagetotal.close();

        return 0;
}
```

APPENDIX C: C++ CODE FOR MICROTUBULE GROWTH DYNAMICS

```
/**********************************************************/
//     RandomNumber.h:
//
// This file contains declaration of random functions for
//  microtubule growth dynamics
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/**********************************************************/

#ifndef _RANDOMNUMBER_H
#define _RANDOMNUMBER_H

#include <time.h>
#include <cstdlib>

void   initialize_random_number();
double  getRandomBetween0And1();
int    getRandomBetween0AndMax(int max);


/**********************************************************/
//          Do not place anything after endif
/**********************************************************/

#endif _RANDOMNUMBER_H
```

```
/*********************************************************/
//    RandomNumber.cpp:
//
// This file contains definition of random functions for microtubule
//  growth dynamics
//
// Author: Josh Gonzalez
// Date:   June/20/2007
//
/*********************************************************/

#include "RandomNumber.h"

////////////////////////////////////////////////////////////////
// initialize_random_number
////////////////////////////////////////////////////////////////

void initialize_random_number()
{
        time_t   seconds;

   time(&seconds);
   srand((unsigned int) seconds);
}

////////////////////////////////////////////////////////////////
// getRandomBetween0And1
//
// returns random number >= 0 and < 1
//
////////////////////////////////////////////////////////////////

double getRandomBetween0And1()
{
        return ((double)rand()/(double)RAND_MAX);
}

////////////////////////////////////////////////////////////////
// getRandomBetween0AndMax
//
// returns random number >=0 and < max
//
////////////////////////////////////////////////////////////////

int getRandomBetween0AndMax(int max)
{
        return(rand() % max);
}
```

```
/*********************************************************/
//    Changing Lattice Length
//                  Functions.h:
//
// This file contains declaration of microtubule growth functions
//
// Author: Josh Gonzalez
// Date:   June/19/2009
//
/*********************************************************/

#ifndef _FUNCTIONS_H
#define _FUNCTIONS_H


#include <time.h>
#include <iostream>
#include <cmath>
#include <fstream>

using namespace std;

//////////////////////////////////////////////////////////////////////////////
// Functions
//////////////////////////////////////////////////////////////////////////////
void intro();  //displays title and prompts user to input variables
bool openResultsLogFile(); //write results
void setInitialRates();
void setNumberPositiveSites();
//void setRatioAndX();
void growORshrink();
void positive_change_negative();
void updateRates();
void runSetFor1TimeStep();
void averageFor1TimeStep();
void reportResults();
void runTrial();

/*********************************************************/
//        Do not place anything after endif
/*********************************************************/
#endif _FUNCTIONS_H
```

```
/*********************************************************/
//	Functions.cpp:
//
// This file contains constants and definitions of functions for
// microtubule growth dynamics
//
// Author: Josh Gonzalez
// Date:	June/22/2009
//
/*********************************************************/

#include "Functions.h"
#include "RandomNumber.h"

////////////////////////////////////////
// Definitions and Globals
////////////////////////////////////////
#define L_max 100
#define L_initial 10
#define Beta 0.8
#define Alpha 0.1

int L;
int L_final[100];
int t;
int positiveSites;
int positive_Sites[100];
double L_sum;
double L_average;
double positive_sum;
double positive_average;
double lambda;
double mu;
double x;
ofstream resultsLog;

////////////////////////////////////////
// intro
////////////////////////////////////////
void intro()
{
	cout << "Dynamic Lattice Length" << endl;
	cout << "Max size of lattice is " << L_max << endl;
	cout << "\nInitial size of lattice is " << L_initial << endl;
	cout << "Beta is = " << Beta << endl;
	cout << "\nAlpha is = " << Alpha << endl;
	resultsLog << "Changing Lattice Length Average Results\n" << endl;
	resultsLog << "Max size of Lattice = " << L_max << endl;
	resultsLog << "\nInitial size of lattice is " << L_initial << endl;
	resultsLog << "\nBeta is = " << Beta << endl;
	resultsLog << "\nAlpha is = " << Alpha << endl;
	resultsLog << "\nt\tAverageLength";
	resultsLog << "\tAveragePositiveSites" << endl;

	return;
}
```

```
////////////////////////////////////////
// setInitialRates
////////////////////////////////////////
void setInitialRates()
{
	L = L_initial;
	lambda = (Beta*(L_max - L))/L_max;
	mu = Alpha*L;

	return;
}

////////////////////////////////////////
// setNumberPositiveSites
////////////////////////////////////////
void setNumberPositiveSites()
{
	positiveSites = getRandomBetween0AndMax(L_initial+1);

	return;
}

/*////////////////////////////////////////
// setRatioAndX
////////////////////////////////////////
void setRatioAndX()
{
	double r;

	r = lambda;
	x = 1/(r+1);

	return;
}
*/
//////////////////////////////////////////
// growORshrink
//////////////////////////////////////////
void growORshrink()
{
	double r;

	r = getRandomBetween0And1();

	if(r <= lambda)
	{
		return;
	}
	if(r > lambda)
	{
		L++;
		positiveSites++;
	}

	return;
```

```
}

/////////////////////////////////////////////////
// positive_change_negative
/////////////////////////////////////////////////
void positive_change_negative()
{
        double r;

        if(positiveSites > 0)
        {
                r = getRandomBetween0And1();
                if(r <= .5)
                {
                        positiveSites--;
                }
        }

        return;
}

/////////////////////////////////////////////////
// updateRates
/////////////////////////////////////////////////
void updateRates()
{
        lambda = (Beta*(L_max - L))/L_max;
        mu = Alpha*L;

        return;
}

/////////////////////////////////////////////////
// runSetFor1TimeStep
/////////////////////////////////////////////////
void runSetFor1TimeStep()
{
        int n;
        int i;

        for(n=0;n<100;n++)
        {
                setInitialRates();
                setNumberPositiveSites();
                for(i=0;i<t;i++)
                {
//                      setRatioAndX();
                        growORshrink();
                        if(L==0)
                        {
                                break;
                        }
                        if(L==L_max)
                        {
                                break;
                        }
```

```
                    positive_change_negative();
                    updateRates();
                }

                L_final[n] = L;
                positive_Sites[n] = positiveSites;
        }
        return;
}

////////////////////////////////////////////
// averageFor1TimeStep
////////////////////////////////////////////
void averageFor1TimeStep()
{
        int n;

        L_sum = 0;
        positive_sum = 0;

        for(n=0;n<100;n++)
        {
                L_sum += L_final[n];
                positive_sum += positive_Sites[n];
        }

        L_average = L_sum/100;
        positive_average = positive_sum/100;

        return;
}

////////////////////////////////////////////
// reportResults
////////////////////////////////////////////
void reportResults()
{
        resultsLog << t << "\t" << L_average << "\t" << positive_average << endl;

        return;
}

////////////////////////////////////////////
// runTrial
////////////////////////////////////////////
void runTrial()
{
        for(t=10;t<=1000;t=t+10)
        {
                runSetFor1TimeStep();
//              cout << t << "\t" << x << endl;
                averageFor1TimeStep();
                reportResults();
        }

        resultsLog.close();
        return;
}

/////////////////////////////////////////////////////////////////////////////
// openResultsLogFile
/////////////////////////////////////////////////////////////////////////////
bool
openResultsLogFile()
{

        resultsLog.open("summary.txt");
   if (!resultsLog.is_open())
   {
      cout << "Cannot open resultsLog file: summary.txt" << endl;
      return false;
   }

        return true;
}
```

```
/**********************************************************/
//    Changing Lattice Length
//            Main.cpp:
// Execution of code for Microtubule growth dynamics
//
// Author: Josh Gonzalez
// Date:   June/19/2009
//
/**********************************************************/

#include <iostream>
#include <iomanip>
#include <cmath>
#include "Functions.h"
#include "RandomNumber.h"

using namespace std;


int main()
{
        initialize_random_number();
        openResultsLogFile();
        intro();
        runTrial();

        return 0;

}
```