STOCHASTIC MODELS OF COLLECTIVE MOTION OF MYXOCOCCUS XANTHUS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Matthew L. Rissler

Mark Alber, Director

Graduate Program in Mathematics
Notre Dame, Indiana
August 2008
STOCHASTIC MODELS OF COLLECTIVE MOTION OF MYXOCOCCUS XANTHUS

Abstract

by

Matthew L. Rissler

Myxobacteria are a group of Gram-negative bacteria that act in a “social” manner. This is most noticeable in the production of the fruiting bodies while cells are under starvation. The social aspect is also evident during swarming, or coordinated expansion of a colony with sufficient nutrient. In this thesis, both fruiting body formation and swarming of Myxobacteria are modeled. It is demonstrated that a modified Lattice Gas Cellular Automata model of cells with reversals is capable of aggregation and mounding in fruiting body formation. In the model, the amount of spores produced reaches a maximum at a reversal frequency similar to that found in experiments. Swarming of cells with A and S motility is also modeled as a modified lattice gas cellular automata, and S-motility in an off-lattice manner.
To my parents and grandparents who have done so much for me.
CONTENTS

FIGURES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vi

TABLES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . viii

ACKNOWLEDGMENTS . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix

CHAPTER 1: INTRODUCTION TO THE STOCHASTIC MODELING OF
CELLS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.1 Continuous vs. Discrete . . . . . . . . . . . . . . . . . . . . . . . . 1
1.1.1 Continuous Models . . . . . . . . . . . . . . . . . . . . . . . . 2
1.1.2 Discrete Models . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.1.3 Hybrid Models . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.1.4 Cell-Based Models . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.1.5 The models of this paper . . . . . . . . . . . . . . . . . . . . . 4
1.2 Monte Carlo Methods . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.2.1 Stochastic Processes . . . . . . . . . . . . . . . . . . . . . . . . 5
1.2.2 Statistical Sampling . . . . . . . . . . . . . . . . . . . . . . . . 9
1.3 Lattice Gas Cellular Automata . . . . . . . . . . . . . . . . . . . . . 11
1.3.1 Cellular Automata . . . . . . . . . . . . . . . . . . . . . . . . . 11
1.3.2 Lattice Gas CA . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
1.3.3 Other models . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
1.4 Self-Propelled Rods . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
1.5 Other Models Applied to Myxobacteria . . . . . . . . . . . . . . . . 21
1.5.1 Aggregation Models . . . . . . . . . . . . . . . . . . . . . . . . 21
1.5.2 Rippling Models . . . . . . . . . . . . . . . . . . . . . . . . . . 22
1.5.3 Swarming Models . . . . . . . . . . . . . . . . . . . . . . . . . 25

CHAPTER 2: MYXOBACTERIA: AN OVERVIEW . . . . . . . . . . . . . . 28
2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.2 Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.3 Motility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.3.1 Motility systems . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.3.2 Reversals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.4 Swarming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
FIGURES

1.1 A schematic of the Buffon’s needle experiment. ........................... 10
1.2 Examples of neighborhoods for cellular automata. ......................... 12
1.3 Example of the Game of Life. .................................................. 13
1.4 An example of the HPP model. .................................................. 15
1.5 Collision rules for the FHP model. .......................................... 16
1.6 An example of the CPM. ......................................................... 18

2.1 Illustration of the possible motility engines of myxobacteria. .......... 31

3.1 The cell in the LGCA swarming model. ....................................... 38
3.2 Two areas for the alignment process. ....................................... 39
3.3 Initial states for the LGCA swarming model. ............................... 48
3.4 A representative radial distribution is on the left and a representative
distance versus time plot is on the right. ..................................... 49
3.5 Results for simulations at the values in Table 3.1. ....................... 51
3.6 The fits for data from simulations for mutants. .......................... 52
3.7 An example of merging peninsulas in simulation. ....................... 52
3.8 An example of a peninsula emerging from a loop in experiment. .... 53

4.1 Representative snapshots from simulations. ............................... 58
4.2 Results from varying the initial density of cells. ........................ 59
4.3 Results from varying the rate of adding cells to the simulation domain. 60

5.1 A representative cell with coordinates \((x, y)\), direction \(\theta\), and length
\(length\). .................................................................................. 62
5.2 Schematic of how the pulling algorithm works .............................. 63
5.3 Areas checked during movement. ................................. 66
5.4 The four possible situations of a cell that could cause a collision. . . 67
5.5 Experimental pictures of $A^-S^+$ cells. ............................. 68
5.6 Initial state for an arrowhead simulation is on the left. The eventual rounded profile is on the right. ................................. 69
5.7 Initial state for the two phase simulation is on the left. The eventual flattened profile is on the right. ................................. 69
3.1 VALUES REACHED BY THE GRADIENT PARAMETER SWEEP  50
ACKNOWLEDGMENTS

I would like to acknowledge my parents and family who have done so much to get me to this point. I would also like to thank Angela for all her support and being such an inspiration to me.

Thank you to my advisor, Prof. Mark Alber for his support and help getting me through my Doctorate. Also thanks to Yi Jiang from LANL and Dale Kaiser from Stanford for time during summers, many helpful conversations, and good advice. Thanks to Olga Sozinova, and Audi Kiskowski for the existing modeling work on fruiting body formation, and Yilin Wu for being a helpful compatriot in modeling. Thanks to my committee: Holly Goodson, Bei Hu, Zhiliang Xu, and Yongtao Zhang.

Finally, I would like to thank the NIH for funding with the NIH Grant No. 1R01-GM076692-01: Interagency Opportunities in Multiscale Modeling in Biomedical, Biological and Behavioral Systems NSF 04.6071, and the Center for Applied Mathematics for summer funding. Some simulations were performed on the Notre Dame Biocomplexity Cluster supported in part by NSF MRI Grant No. DBI-0420980.
CHAPTER 1

INTRODUCTION TO THE STOCHASTIC MODELING OF CELLS

Modeling of cell-cell interactions is one of the important areas of mathematical biology. Mathematical biology is an area of applied mathematics which deals with investigating biological systems through mathematical modeling. These systems can vary in size from the population of elephants down to the molecular interactions of proteins. They can vary in length of time investigated from generations of an organism down to the several nanosecond vibrations of an atom[48, 67]. In between these extreme scales lies the realm of cell-cell interactions. Because these interactions are not at the extremes, cells can be viewed in several ways which highlight several different approaches to mathematical biology. These different approaches will be discussed in the following two sections.

1.1 Continuous vs. Discrete

The first method of looking at cells is that they are the small building blocks of large structured groups of cells, such as tissues and organisms. From this view, cells are mostly interchangeable and the position of individual cells is basically irrelevant. Because of this the density of cells can be viewed as a continuous variable that varies over space and time in a smooth fashion.

The second view of cells is that they are compartmentalized volumes of interacting proteins, DNA, RNA, and other molecules. Based on the concentrations and
interactions of these substances with in a cell, it will react to its environment in various fashions. This is modeled to help determine what is important to the cell making decisions based on its environment and the internal state of the cell. Often even mixing within the cell is assumed so concentrations of the substances of interest may be modeled with ordinary differential equations (ODEs). Otherwise, the subcompartments and space within a cell would be modeled. Also for some of the reactants the concentration is so low that small random perturbations may have large effects[58].

In between these scales, cells are both decision-making compartments and the building blocks of tissues and organized structures. As such, the small scale reactions of proteins cannot be fully modeled, and the differences between individual cells are too important to be averaged out. This level is the area of cell-based modeling.

1.1.1 Continuous Models

In continuous models, the length scales of the structures of interest are large in comparison to smallest length scales of the problem. The smallest length scales are, for example, the size of particles in fluids or the size of the cells in many biological problems. These models may be deterministic or stochastic. The dynamics of the deterministic models involve an assumption of averaging over space and time and are represented by partial differential equations (PDEs) or integro-partial differential equations. The dynamics of stochastic continuous models involve high levels of randomness over several time scales in some variables and are represented by stochastic differential equations (SDEs). Both types of continuous models deal with densities of cells and concentrations of substances in biological applications.

An advantage of deterministic continuous models is that they have been studied for over 300 years. The techniques for studying such models are mature and
well-established. This often means that once the model is formulated, one can immediately analyze what the dynamics of the model are in special cases. Also, the long-time scale dynamics often are tractable. Stochastic differential equation models have been studied for just over a century. A disadvantage of continuous models is that the effect of small-scale features must be introduced in an averaged sense. This often obscures the ability to describe an individual cell. Another disadvantage is that, with the exception of simple cases, the model must be discretized to study it via computational methods.

1.1.2 Discrete Models

In discrete models, the length scales of structures of interest are comparable to the smallest length scales. These models may also be deterministic or stochastic. The deterministic discrete systems often are used for simplified models of processes, such as the Game of Life or the HPP LGCA (see below). However, stochasticity is quickly necessary when modeling. In order to achieve this stochastic nature a Monte Carlo algorithm is used. A brief introduction to discrete stochastic processes and history and description of the Monte Carlo method follows in the next section.

An advantage of the discrete models is that the small-scale interactions can be quantified. Also they are implementable on a computer. However, several disadvantages exist. The first is that we cannot know the long-time scale dynamics \textit{a priori} for many models, as demonstrated by the Turing completeness of the Game of Life. The second is that for the stochastic models many simulations are necessary to have significant understanding of the dynamics of the system. This difficulty, included with the computational complexity of the models leads to a large computational load for these models. This load is normally higher than that for the discretization of continuous models. Another method for analyzing the dynamics is to use
a limiting procedure to obtain a related PDE that can be analyzed using existing techniques.

1.1.3 Hybrid Models

One method of taking advantage of the strength of both continuous and discrete models is to combine them into one model. This is often done when the problem of interest has multiple scales that matter. For the small space scales or fast time scales one uses a continuous approach and for the larger space scales or slower time scales a discrete approach is used. This type of model is quite prevalent. Some of the difficulties from this approach come from the need to match scales appropriately and the need for computational resources due to the discrete part of the model.

1.1.4 Cell-Based Models

Relatively recently agent-based models have come to the forefront as a method of modeling systems of objects. In an agent-based model, a set of objects interact in an algorithmic fashion. More specifically cell-based models are a set of agent-based models that take cells as the agents. In these models, cells have a spatial representation. This spatial representation can be defined continuously or discretely on a lattice. Time is discrete, either due to the algorithmic nature of cell-cell interactions, or because of the discretization of the differential equations governing cells. The major distinction of cell-based modeling is that the cell is an integral part of the model.

1.1.5 The models of this paper

For the models in this dissertation, we use discrete cell-based models because both in swarming and fruiting body formation the dynamics operate on the scale of thousands of cells. The peninsulas formed during swarming usually are tens of cells
across, and fruiting bodies are only tens to hundreds of cell lengths across. Because of this a discrete model is appropriate. Due the length scale and variability in the reactions of cells, stochastic interactions are also appropriate.

1.2 Monte Carlo Methods

Stochastic modeling has its roots in the development of probability theory under the name of statistical sampling. However, it was not used much prior to the advent of the electronic computer. The expansion of the usage of stochastic methods came after the second World War. During this time, the methods of using an algorithm that involves random numbers to gain understanding of a problem were dubbed Monte Carlo methods.

1.2.1 Stochastic Processes

Monte Carlo methods were designed carry out trials of stochastic models. Stochastic models are based on representing a system as a stochastic process or a collection of random variables that evolve over time.

**Definition 1.** A stochastic process is a collection of random variables \( X_t(s) : t \in T, s \in S \), where \( T \) is an index set (usually time) and \( S \) is the sample space of the random variables.

The sample space is the domain of possible outcomes of the random variables. The random variables are variables that take on values with certain probabilities. An example of a sample space is \( S = \{H, T\} \) in the case of flipping a coin, where \( H \) represents the coin coming up heads and \( T \) represents tails. An example of a stochastic process is \( X_t(s) = \{\text{the number of heads observed after } t \text{ coin flips}\} \). The state space in this case is \( S_t = \{0, 1, \ldots, t\} \). While these examples are discrete in nature, continuous stochastic processes exist.
The best understood subset of stochastic processes is the processes that satisfy

\[ Pr\{X_{n+1} = j|X_0 = i_0, X_1 = i_1, \ldots, X_{n-1} = i_{n-1}, X_n = i\} = Pr\{X_{n+1} = j|X_n = i\} \]

(1.1)

for all \( n \) and \( i_0, \ldots, i_{n-1}, i, j \). That is to say that the system depends only on the current state, and none of the preceding ones or that it is memoryless. A stochastic process that satisfies this requirement is called Markovian, or simply a Markov process.

For a discrete Markov process, or Markov Chain, we can define the transition probabilities, or the chance of moving from one state to the next as \( p_{i,j}(n) = Pr\{X_{n+1} = j|X_n = I\} \). If these probabilities do not depend on \( n \) then the Markov Chain has stationary transition probabilities, and can be represented as a matrix. In the case of counting the number of heads observed for a fair coin this matrix is

\[
P = \begin{pmatrix}
\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & \cdots \\
0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & \cdots \\
0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & \cdots \\
\vdots & \ddots & 0 & \ddots & \ddots & 0 & \ddots
\end{pmatrix}
\]

(1.2)

This matrix representation leads to the understanding that probability of traveling from state \( i \) to state \( j \) over \( n \) steps, \( p_{i,j}^{(n)} \), is simply \( P_{i,j}^n \) or the \( i,j \) entry of the \( n \)th power of \( P \).

States can be classified by whether they can reach one another. More explicitly this means that \( p_{i,j}^{(n)} > 0 \) and \( p_{j,i}^{(m)} > 0 \) for some positive integers \( m \) and \( n \). States that satisfy this are said to communicate. The state space can be divided into classes of those states that communicate with each other. If all states communicate the Markov chain is irreducible, otherwise it is reducible. Notice that communicating requires that the system can go from \( i \) to \( j \) and back to \( i \). So in a reducible chain, the system may move from one class to another, but not return. For the counting
heads example the classes are \{\{0\}, \{1\}, \ldots\} because the number of heads observed can only increase.

This possibility of not returning to the original state leads to the following definition:

**Definition 2.** Let $f_{i,i}^{(n)} = \text{Prob}\{X_n = i, X_m \neq i, m = 1, 2, \ldots, n - 1|X_0 = i\}$. State $i$ is transient if $\sum_{n=1}^{\infty} f_{i,i}^{(n)} \leq 1$. It is recurrent if $\sum_{n=1}^{\infty} f_{i,i}^{(n)} = 1$.

For recurrent states it makes sense to talk about the equilibrium solution to the system, or $\pi$ such that $P\pi = \pi$. For transient states, it should be clear that the corresponding entries in $\pi = 0$. With the matrix understanding of $P$ this is simply an eigenvalue problem. However, the solution is not always unique. For the counting heads example all states are transient. In a finite Markov chain this could not happen.

As another example consider a particle undergoing a random walk on the integers. Each time step, the particle can move to the left one space or to the right one space. It chooses each choice with equal probability. For this example, the matrix associated is

$$\mathbf{P} = \begin{pmatrix}
\vdots & 0 & \cdots & \cdots & \cdots & 0 & 0 & 0 & 0 \\
\cdots & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & \cdots \\
\cdots & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & \cdots & \cdots \\
\vdots & 0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & \vdots
\end{pmatrix} \quad (1.3)$$

This is clearly irreducible, and with a bit of work and Stirling's formula it can be shown that every state is recurrent. Interestingly, if you extend this to two spatial dimensions each state is still recurrent. However, in the extension to three dimensions every state is transient. Be thankful for this the next time you are near a smokey fire or a bad smell.
Now imagine that we have a particle undergoing a symmetric random walk on 
\{-5, -4, \ldots, 4, 5\}. Assume that at the boundaries, moving outward with probability 
\(\frac{1}{2}\) is changed to not moving. The matrix for this system is

\[
P = \begin{pmatrix}
\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2}
\end{pmatrix}
\]  

(1.4)

Since the system is irreducible and finite, every state is recurrent. The stationary 
solution for this is each state being occupied with probability \(\frac{1}{11}\).

For simple models, the enumerating and analysis of the matrices associated with 
the model is relatively easy. However, for more involved models and ease of changing 
the model, collecting statistics for multiple trials of the system is easier and more 
intuitive. Each of these trials is a realization of a stochastic process according to 
the transition probabilities. A Monte Carlo simulation is simply a computational 
method for gaining a trial.
1.2.2 Statistical Sampling

The classic example of statistical sampling, the name of the field before Monte Carlo methods, is Buffon’s Needle. The name comes from Georges Louis Leclerc, Comte de Buffon, who thought of this experiment as a method for estimating \( \pi \).

The experiment is simple, take a plane that is ruled by evenly spaced parallel lines with spacing \( w \) and a pin with length \( l < w \) as in Figure 1.1. Drop the needle so that it lands with a uniform spatial distribution and at all angles with uniform probability.

Due to symmetry, one needs only to consider \( 0 \leq x \leq \frac{w}{2} \) and \( 0 \leq \theta \leq \frac{\pi}{2} \), where \( x \) is the distance of the center of mass of the needle from a line, and \( \theta \) is the angle the needle makes with a line parallel to the evenly spaced lines. The needle will cross a line if \( x \leq \frac{l}{2} \sin \theta \). So \( P(\text{crossing}) = \int_{0}^{\frac{\pi}{2}} \int_{0}^{\frac{l}{2} \sin \theta} \frac{1}{w \pi} dxd\theta = \frac{2l}{w \pi} \). Laplace observed that this could be used to experimentally approximate \( \pi = \frac{2lN}{wH} \) where \( N \) is the total number of drops of the needle and \( H \) is the number that intersect a line by repeating the experiment enough times.

Several mathematicians did this experiment, though none has received the scrutiny that the results of Mario Lazzarini have. In 1901, Lazzarini published a result for the above experiment of \( H = 1808 \), and \( N = 3408 \) with a needle of length \( l = 2.5 \text{cm} \) and lines separated by width \( w = 3 \text{cm} \). This leads to an estimate of \( \pi = \frac{355}{113} \). This result is specious because \( \frac{355}{113} \) is the best rational approximation of \( \pi \) short of a 5-digit numerator and denominator. The other reported values of the experiment also lack appropriate variation[7].

Mechanical calculators allowed a step forward in using statistical sampling. This is shown by the use of such methods by Lord Kelvin’s secretary in 1901[61] and Enrico Fermi in the 1930’s while working on neutron diffusion[59]. When ENIAC was built in 1945 it was quickly percieved as being useful for more than simply
calculating firing tables. Across the country, while at Los Alamos during World War II, Stanislaw Ulam had the idea for the Monte Carlo method while trying to calculate how often he would win at Canfield, a version of solitaire. ENIAC could play the games far faster than he could, and then he made the mental leap to using it for calculations for the hydrogen bomb. John von Neumann and Nicolas Metropolis worked with Ulam on the initial basics of the Monte Carlo Method. When ENIAC was moved to the Navy Ballistic Research Laboratory in 1947, it was programmed for Monte Carlo calculations[61].

Bombs were not the only problems that were addressed by these pioneers. Simulations of hard-sphere gases in two and three dimensions were among the first papers using this method. These papers included the name Monte Carlo method, which was coined by Metropolis because Ulam had an uncle who “just had to go to Monte Carlo” from time to time[59]. The building of MANIAC at Los Alamos
in 1948 led to greater ability to run simulations and development\[59\]. Since then
development of computational power, as well as improvements in algorithms, has led to an explosion of applications and techniques for Monte Carlo simulations.

The basic premise of Monte Carlo simulations is that a stochastic model for a process is developed. This model is then simulated for one path numerous times. Statistics for the results of these paths are collected and compared with statistics for the process being modeled. Several examples of algorithms for implementing the Monte Carlo simulations of stochastic follow. The discussion will now be restricted to those that relate to the models in this work.

1.3 Lattice Gas Cellular Automata

Lattice Gas Cellular Automata (LGCA) are a subset of Cellular Automata (CA) that were originally designed to gain understanding of gases by simplifying the state space to being discrete in time, space, and velocity. This section will start by explaining what CA are and how they developed.

1.3.1 Cellular Automata

Cellular automata are dynamical systems that are discrete in space and time. Their dynamics are typified by being local in nature with a parallel method of evolution. The system is made up of many cells that can take on a finite number of states. Let $\phi_{\vec{x}}(t)$ denote the state $\phi$ of the cell at position $\vec{x}$ at step $t$ with $\phi$ taking on values in $\{0, \ldots, n - 1\}$, where $n$ is the number of states that each cell can take. These cells are arranged in a lattice. The update of the system is synchronous with each cell changing under a rule, $\phi_{\vec{x}}(t + 1) = f(\phi_{\vec{x} + \vec{y}}(t))$, where $\vec{y}$ varies over the values from a set $\Omega$. $\Omega$ often is simply those vectors, $\vec{y}$, with integer entries such that $|\vec{y}| < C$, where $C$ is a real number. Given a $\vec{x}$, the cells that are at $\vec{x} + \vec{y}$ for some $\vec{y} \in \Omega$ are the neighborhood of $\vec{x}$. Figure 1.2 shows several common neighborhoods
CA were introduced by John von Neumann as a model of biological replication. His CA was two dimensional and each cell could take one of 29 states. With this system von Neumann showed that there was a self-replicating pattern\cite{33}. John Conway invented the Game of Life in 1970 as a simple model for plant propagation. Cells can have only two states, “alive” or “dead”. Using the Moore neighborhood, if a “dead” cell has 3 “alive” neighbors it becomes “alive”. If an “alive” cell has 2 or 3 “alive” neighbors, it stays “alive”. Any other state leads to the cell either staying “dead” or transitioning from “alive” to “dead”\cite{16}. Figure 1.3 shows several transitions under this rule.

Due to the simplicity of the system and the expansion of the use of computers, Conway’s Game of Life has received a great deal of attention. Much attention was spent on finding stable structures, periodic structures (a “blinker” is in the upper right hand corner of Figure 1.3), and structures that are self-propagating. “Gliders” are the best known example of a self-propagating structure. One is in the upper
left-hand corner of Figure 1.3. Notice that it appears in the same shape in the final frame but having moved down a space and to the right a space. Eventually it was shown that the Game of Life is Turing complete\cite{8}.

In one dimension, the simplest neighborhood is the sites to the right and left of $\vec{x}$. The simplest set of values that $\phi$ can take on is the binary set, $\{0, 1\}$. Wolfram, in a series of papers, investigated what occurred for each of the possible $2^{2^3} = 256$ rules\cite{86, 87, 55}. These rules fall into one of 4 categories:

1. The system settles into a homogenous state.
2. The system settles in a simple pattern that may be periodic in time.
3. The system generates chaotic patterns.
4. The system develops locally complex patterns that may be long-lived.

Of the classes, class 4 is the most interesting due to the lack of correlation between it and continuous dynamical systems. These categories have been refined since\cite{50}. Matthew Cook proved that one of the rules in this simple system, rule 110 under Wolfram’s naming, is also Turing complete\cite{17}.

1.3.2 Lattice Gas CA

While general Cellular Automata can make good models, in the interest of developing a discrete microscale model for fluid flow, Hardy, de Pazzis, and Pomeau developed the HPP model\cite{27}. This brought about a new class of systems called
Lattice Gas Cellular Automata (LGCA). In these systems, one is looking at particles of a gas that move on a lattice. Time, space and velocity are all discrete quantities. LGCA differ from CA in several ways:

1. The cells have binary states, but represent the presence or absence of a particle. The particles occupy the directed links between the lattice sites, or nodes.

2. The update rule is split into two steps, propagation and collision. This splitting allows for simple update rules.

3. The update rules are defined to conserve mass and momentum.

In the HPP model, particles travel on the directed links, or channels, between the nearest neighbors of nodes on a square lattice. Each particle has unit mass and velocity in the direction of the channel it occupies. So at each node there can be at most 4 particles. A diagram for this model is shown in Figure 1.4. The propagation rule, \( P \), simply moves each particle one node in the direction of the channel it occupies. The collision rule, \( C \), is simple, if only two particles are at the same node and they have opposite direction, the two particles switch to the two unoccupied channels at the node. While these rules conserve mass and momentum, this system is not a good model for fluid flow. The problem lies in the fact that the lattice tensor of rank 4, or \( \sum c_{i\alpha_1} c_{i\alpha_2} c_{i\alpha_3} c_{i\alpha_4} = 2\delta_{\alpha_1\alpha_2\alpha_3\alpha_4} \), where \( c_{i,j} \) is the \( j \)th component of the direction of the \( i \)th link at a node, is non-isotropic, or not invariant under arbitrary rotations or reflections\[85\]. This anisotropy causes the momentum flux to have preferred directions of motion. In this case, the diagonals of length \( \sqrt{2} \).

In 1986, Frisch, Hasslacher, and Pomeau (FHP) generalized the HPP model to a lattice with hexagonal symmetry. This has a larger symmetry group which allows for the destruction of the invariants that prevent the HPP model from having appropriate macroscale dynamics. The FHP model is similar to the HPP except that particles now travel on the links between nodes on a hexagonal lattice. The major difference is that there are more collision rules that must exist. The collisions
Figure 1.4. An example of the HPP model. Propagation is followed by the collision operator. Notice the oppositely oriented two-particle collision on the bottom row is the only collision rule that leads to a change.

that lead to changes in state are shown in Figure 1.5. Unlike the HPP model where all collision rules are deterministic, rule (a) in Figure 1.5 should be stochastic. This serves to destroy a chirality invariance that would be introduced if this rule always led to rotating in the same direction. Further generalizations allow for particles at rest to reside at nodes. This model does display better macroscale dynamics. An asymptotic limit that separates length scales around the equilibrium state leads to a form of the Navier-Stokes equations for fluid flow[23].

To move beyond these simple neighborhoods, multispeed models have been developed. These models work on the links between a larger neighborhood of nodes. However, as more links are added to each node, the collision dynamics become more intricate. Some multispeed models can also lead to an appropriate continuous limit.

The models in chapters 3 and 4 are modifications of the FHP model. The collision rules for these models are not defined solely at one point, but for an extended domain.
Figure 1.5. Collision rules for the FHP model. The two options for collisions (a) and (b) are chosen at random with equal probability.

1.3.3 Other models

Other CA models of interest also exist. A major branch of CA are those that have developed from the Ising Model[34]. Developed originally as a model for magnetism, this model has an array of sites, each of which can take on two states, \( S_i \in \{-1, 1\} \).

As a model for magnetism there are two effects acting on the atoms represented by the sites. The first is the influence of the magnetic field generated by nearby atoms. The second is the bulk magnetic field in which the material is placed. Each of these are represented in the total energy function for the system,\

\[
E = \sum_{i,j} J_{ij} S_i S_j + \sum_i h_i S_i,
\]

where \( J_{ij} \) describes the strength of interaction between the spins at sites \( i \) and \( j \), and \( h_i \) describes the strength of the external field at site \( i \).

The spins on the lattice are asynchronously updated, normally via the Metropolis algorithm:

1. One site is selected with uniform probability.
2. The change in energy, \( \Delta E \), is calculated for switching the state of that site.
If $\Delta E \leq 0$, the state of the site is switched.

- If $\Delta E > 0$, the state of the site is switched with probability $P = e^{-\beta \Delta E}$, the Boltzmann probability.
- Otherwise, the site is left alone.

In two and three dimensions this simple model with simple stochastic dynamics undergoes a phase transition with respect to $\beta$, which is proportional to the inverse of the temperature of the system. To model more complex dynamics, this model was extended by Potts, who expanded the number of states that a site can take on from 2 to some integer, $n[64]$. Extensions of the Potts model have been used to model crystal formation in metals and the coarsening of bubbles in a foam[5, 76].

Another extension of the Potts model has been used for biological modeling. The Cellular Potts Model (CPM) was developed by Glazier and Graner[26]. It has been used for many problems, including cell sorting, cancer modeling, vasculargenesis, and limb formation[35, 14, 39]. One major difference between CPM and the original Potts model is that each site has two indices. The first index, $C_i$, labels what type of the cell inhabits the site and the second, $T_i$, labels which cell of that type it is. Here cell actually means a biological cell. Figure 1.6 gives a snapshot of a CPM state.

Another major difference from the Potts model lies in the energy function. Instead of representing the energy associated with the spin of atoms, it represents the energy associated with each cell. The basic energy involves two terms. The first represents the energy associated with the surface area of the cell. The second represents an energy associated with the volume of the cell. In two dimensions, these are simply the perimeter of a cell and its area, respectively. Their expressions are $E_{SA} = \sum_{C,T} \alpha_{SA}(\gamma_{SA} - L_{C,T})^2$, and $E_V = \sum_{C,T} \alpha_{V}(\gamma_V - V_{C,T})^2$, where the $\gamma_i$ represent the expected value of either surface area or volume and the $\alpha_i$ model the variability in the actual value. Another energy term represents the adhesive interaction between
Figure 1.6. An example of the CPM. Types of cells are represented by color (or hatching), while separate cells of the same type are represented by a difference of index.

adjacent cells. Its expression is $E_A = \sum_{i, j \text{ near } i} \sum_{i, T_i \neq T_j} f(C_i, C_j)$, where $f$ describes the relative adhesivity between each of the cell types. Other energy-like functions also have been introduced in various applications, such as a gradient-following function for chemotaxis and haptotaxis, functions to maintain an elongated shape and a function to prevent a site labeled as a certain cell from becoming disconnected the rest of that cell.

Another important difference from the Ising model is which indices sites can transition to when they are selected. In the Ising model, development of clusters of site with the same index is an emergent property, but has no more meaning than that. In the CPM, however, the clusters of sites represent cells, and having a
different cell pop up in the middle of an existing cell does not make biological sense. Because of this, sites are only allowed to select from the states of the sites within their neighborhood. For example, the circled site in the lower left hand corner in Figure 1.6 can change to north-east hatching 1 or 2, or remain as south-east hatching 1 if one is using the von Neumann neighborhood. Using the Moore neighborhood would also allow it to transition to south-east hatching 2 or a blank site.

The CPM gives rises to an example of another type of analysis for stochastic systems, trying to find an appropriate PDE to relate to the system. Take a one-dimensional CPM, or a lattice of sites on the integers. Each cell can be defined by two variables, the position of the cell’s center of mass (on the lattice of $\mathbb{Z} + \frac{\mathbb{Z}}{2}$) and the length of the cell. From this formulation, you can write down an ODE for the change of length and position. This ODE can be transformed into a PDE once you assume that the length of a cell does not vary much from the average length. If the probability of moving left or right for a cell is the same you arrive at a simple diffusion process. This makes sense since a cell is then simply undergoing a symmetric random walk which the limit of resembles a simple diffusion model that depends on the parameters of the CPM[79]. However, if these probabilities are not the same then the equation becomes a more complicated equation. In the case of a chemotactic gradient the related PDE is the Keller-Segel equation[1]. This limit method has also been extended to two dimensions[2].

1.4 Self-Propelled Rods

In 1995, Vicsek et al. developed a model for oriented particles on the plane. An important aspect of this model was that particles aligned with other particles nearby[81]. The basics of the model are that there are $N$ particles labeled $1, 2, \cdots, N$ that move with unit velocity in a direction $\theta$. The system update is then defined by
the differential equations

\[
\begin{align*}
\vec{x}_i(t + \Delta t) &= \vec{v}(\theta_i) \Delta t + \vec{x}_i(t) \\
\theta_i(t + \Delta t) &= \frac{\sum_j f(\vec{x}_i, \vec{x}_j, \theta_j, t)}{N(\vec{x}_i, t)} + \eta(t),
\end{align*}
\]

where $\vec{x}_i \in \mathbb{R}^2$ and $\theta_i \in [-\pi, \pi)$ are the position and orientation of particle $i$, $\vec{v}(\theta)$ is the vector, $\cos(\theta_i)\hat{i} + \sin(\theta_i)\hat{j}$, and $\eta(t)$ is uncorrelated noise uniformly distributed in $[\eta_0/2, \eta_0/2]$. In the original model, $\vec{f}$ is simply $\theta_j$ if $|x_i - x_j| < d^*$ and zero otherwise, and $N(x_i)$ simply counts how many particles are within $d^*$ of $x_i$. Basically, at each time step each particle moves in the direction that is the average of the direction of nearby particles from the prior timestep, plus a little noise.

This simple model demonstrates some interesting behaviors. The first behavior is flocking, or a cluster of particles moving with the same direction. As may be apparent from its name, this behavior is meant to bring to mind the movement of flocks of birds or schools of fish. Milling, or organisms moving in a stationary circular pattern, is another behavior that occurs in nature. However, these patterns are at best transient in the original model. When a short-range hard-core repulsion is added to prevent particles from coming to close to each other, a stable milling pattern develops[15].

The model has been expanded to more dimensions, and more complex interactions. However, most of the extensions still deal with either point particles or symmetric hard-core interactions. These assumptions work passably for birds, fish and other organisms that have some visual ability. However, for bacteria, and especially myxobacteria, contact interactions are important. Chapter 5 describes preliminary results for a rod-based model in the same vein.
1.5 Other Models Applied to Myxobacteria

The prior sections were intended to give an introduction to the mathematical roots of the models in the following chapters. This section is intended to provide background on the models that have been applied to myxobacteria. Models have been developed to address three different stages of the life cycle of a colony. The first stage is the rippling stage after the onset of starvation. The second stage modeled is the aggregation of cells for fruiting body formation, and formation of the bodies. The third stage is the swarming of vegetative cells under growth conditions. Description of these stages and much of the biological background on myxobacteria can be found in Chapter 2.

1.5.1 Aggregation Models

Aggregation was first modeled as a combination of slime interaction and chemotaxis by Stevens[78]. The chemotactic aspect of the model was based on the fact that similar interactions had been shown to agree with the aggregation of the slime mold *Dictystelium discoideum*[49, 46, 38], and there was no positive or negative evidence for or against it in Myxobacteria. Simple slime interactions were not able to form stable aggregates in the model.

The model is a square-latticed cellular automaton where a cell is made up of eight sites. The leading end of the cell is named the “head”. Each time step the head moves to one of the sites in the head’s von Neumann neighborhood that the cell does not occupy. The probability of moving into a site is increased by the presence of slime or a diffusing chemoattractant. With these rules aggregates form. However, a diffusing chemoattractant has little correspondence to the real biology of the system[19].

Papers by Alber et al.[3, 4, 47] describe a two-dimensional model that leads
to aggregates. This model is a LGCA on a hexagonal lattice where cells have an extended body with a $3 \times 21$ outline. The collision step is the important part of this model. At the ends of the extended body are two C-signaling regions. The collision step turns cells in a manner that prefers to overlap these regions. This model showed that the simple rule of increasing overlap of the ends creates stable aggregates. Also this model discovered that early aggregates transfer cells to each other via streams.

Two papers by Sozinova et al.[73, 74] describe the extension of the previous model to three dimensions. Cells now have an ellipsoidal extended body and move on a face-centered cubic lattice. The model is the predecessor of the model in Chapter 4. This model produces a hemispherical aggregate with the development of spores.

1.5.2 Rippling Models

Stevens and Lutscher[52] modeled rippling as a one-dimensional system with left and right moving cells. The equations for this model are:

\[ u_t^+ + \gamma u^+_x = -(\mu + \lambda^+)u^+ + (\mu + \lambda^-)u^- \]  
\[ u_t^- + \gamma u^-_x = (\mu + \lambda^+)u^+ - (\mu + \lambda^-)u^- . \]  

(1.6)  
(1.7)

Testing several forms that the function $\lambda$ can take, they show that standing ripples can develop. They also show that adding mutants that turn differently can prevent ripples from occurring. By varying some parameters, they also show that this model can lead to aggregation.

A two-dimensional continuous model of rippling was published in 2001[32]. This model used the Fokker-Plank equation to derive the continuous equations from a simple model for one cell. The model for a cell involved a stochastic self-propelled particle that had an internal reversal clock. When the clock was halfway through
its cycle, the cell reversed direction. Using the observation that the cells of interest are aligned parallel to each other, the equation for the change in density, \( n(x, y, \phi) \), is

\[
\frac{\partial n}{\partial t} = -\frac{\partial}{\partial x} \left( -D_x \frac{\partial n}{\partial x} \pm v_x n \right) - \frac{\partial}{\partial y} \left( -D_y \frac{\partial n}{\partial y} \right) - \frac{\partial}{\partial \phi} \left( -D_\phi \frac{\partial n}{\partial \phi} \pm \omega_\pm n \right),
\]

(1.8)

where \( D_i \) is the effective diffusion in \( i = x, y, \phi \); \( v_x \) is the average velocity of a cell; and \( \omega_\pm = \omega_0 + \omega_n F(\phi) \), where \( \omega_0 \) is a constant, \( \omega_n \) is a hill function with respect to oncoming cells, and \( F(\phi) \) represents the switching behavior of a refractory period. This model can match experimental measurements of wavelength, speed and period of ripples. By placing an individual model cell in the continuous model, it was demonstrated that the waves that seemed to interpenetrate were actually caused by almost complete reflection of the individual cells.

This group also modeled rippling in two dimensions with slime orientation dynamics\[31\]. The equations for this model are:

\[
\frac{\partial n(x, y, t)}{\partial t} = \nabla \cdot \left( J_{\text{Diffusion}} + J_{\text{Gliding}} \right) + K \cdot n
\]

(1.9)

\[
\frac{\partial T(x, y, t)}{\partial t} = \alpha \cdot \nabla^2 T(x, y, t) + \tau
\]

(1.10)

Here, \( n = (n_s^+, n_r^+, n_s^-, n_r^-, n_0) \) is a vector that represents the local density of cells that are positively or negatively moving (subscript) and sensitive or refractory (superscript) or stationary\( (n_0) \). The positive or negative direction is relative to the slime field \( T \) which is being restructured. \( J_{\text{Diffusion}} = \begin{pmatrix} \nabla n_s^+ \\ \nabla n_r^+ \\ \nabla n_s^- \\ \nabla n_r^- \\ D_n \nabla n_0 \end{pmatrix} \) represents the random fluctuations of the direction of motion of the cells. \( J_{\text{Gliding}} = \begin{pmatrix} v_0 T n_s^+ \\ v_0 T n_r^+ \\ -v_0 T n_s^- \\ -v_0 T n_r^- \\ 0 \end{pmatrix} \) represents the gliding movement of the cells. Therefore, the direction of gliding is
determined by $T$. $K$ is a matrix that determines the rate of switching between
directions, as well as the stationary, refractory, and sensitive states. Rippling and
aggregation are produced by this model.

The actual reversal control network has been modeled[29]. The network involves
the Frz proteins. This model is three sets of ODEs that display several responses
like those found in experiment. The first response is that the output parameter
oscillates. The second is that the system has a refractory period. And the third is
that once C-signal reaches a threshold the output stops oscillating. This model was
then used in an agent-based model to investigate the development of rippling[71, 70].

The rippling phase was also modeled by a discrete 3-D model in space and time.
The discretization of space has two scales: in the $x$-direction, the length of a lattice
site is the length of a cell, and the $y$- and $z$-directions spacing is the width of a cell.
The discretization of space is 1 minute. In the model, cells move in the $x$-direction
taking into account collisions in an asynchronous manner, with each cell moving
individually and climbing around other cells by moving one space in either the $y$- or
$z$-direction. The dynamics for reversal are simple. A cell has a refractory period, $\tau$,
in which it will not reverse after a reversal. Once this period has expired, the cell
reverse once it meets a cell moving in the opposite direction in one of the sites of the
von Neumann neighborhood in the $y − z$ plane of the site in front of the cell. Rippling
can be modeled with this system. This model was modified to include an internal
clock and different collision dynamics to advance this clock besides the refractory
period. The phase space of possible collision rules was explored and several possible
regimes were found[11].

Alber et al. [3, 47, 4] modeled reversal with a two dimensional LGCA. Similar to
the above model, each cell had an internal clock. However, this model incorporates
an internal timer that has a refractory period after each reversal when the timer
increases at a constant rate. After that period, the timer increases at a rate proportional to the number of head-to-head collisions the cell is experiencing. Once the timer reaches a maximum value, the cell reverses. This model also develops ripples with and without mutant cells.

1.5.3 Swarming Models

A continuous model of swarming was published in 2006[25]. This model matches two experimental papers that describe swarming on two separate time scales. The first paper is by Burchard, and describes the expansion rate of a colony on a long time scale of 50-250 hours[12]. The second, by Kaiser and Crosby looks at the spreading over the first 4 hours after inoculation[43]. For both cases the problem is reduced to a one-dimensional radial problem. The equation for the short time scale is

\[
\frac{\partial C}{\partial t} = \frac{\partial}{\partial r} \left[ D_c \frac{\partial C}{\partial r} \right] - \left[ D_c \frac{\partial (\ln A)}{\partial r} \right] - \left[ \frac{\partial (\ln A)}{\partial t} \right] C. \tag{1.11}
\]

\( C \) is the cell density, \( A \) is the area occupied by cells, and \( D_c \) is the effective diffusion rate of cells. This equation is arrived at from a conservation equation for the product of the area covered by cells and the density of cells. The authors make an assumption about the movement of the tips of peninsulas. They assume that no new peninsulas are formed during the spreading and that peninsulas merge at a certain rate. While the merging is observed, it is not reasonable to assume that no more are created during swarming. Also, the initial area occupied by the bacteria is much lower than the actual experiment.

The equations for the long time scale take nutrient into account and are

\[
\frac{\partial C}{\partial t} = \frac{\partial}{\partial r} \left[ D_c(C) \frac{\partial C}{\partial r} \right] + \frac{1}{r} D_c(C) \frac{\partial C}{\partial r} + pCN \tag{1.12}
\]

\[
\frac{\partial N}{\partial t} = D_n \left[ \frac{\partial^2 N}{\partial r^2} + \frac{1}{r} \frac{\partial N}{\partial r} \right] - g p C N. \tag{1.13}
\]
The first equation deals with the spread and production of cells, \( C \). The second deals with spread and consumption of nutrients, \( N \). \( p \) represents the growth rate of the cells, and \( g \) represents the ratio between nutrient uptake and cell production per nutrient uptake. This set of equations was solved numerically, and it was found that the expansion rate agreed with experiment in that it depended on the square root of the initial concentration of nutrient.

The next model for swarming does not actually look at swarming. Instead, it looks at the forces acting on a myxobacteria that has been tethered via its pili to the substrate. A cell tethered in this fashion will whip back and forth due to its A-motility engines[75]. Wolgemuth modeled this situation as a linearly elastic filament that is fixed at one end and forced at the other[89]. From this he was able to estimate the force of the A-motility engines.

This view of myxobacteria as an elastic filament was incorporated into a swarming model[77]. The model is a CPM, except that each cell is made up of subcells. The centers of mass of these subcells are used to determine two energy functions. The first is 

\[
E_{\text{length}}(\mu) = \zeta \sum_{\nu=1}^{s-1} (|S_{\mu,\nu} - S_{\mu,\nu+1}| - D)^2,
\]

where \( s \) is the number of subcells, \( \mu, \nu \) are the indices for the cell and subcell, respectively, and \( D \) and \( \zeta \) are parameters that determine target length and influence of this energy. This energy basically restricts the length of the cell. The second energy depends on an approximation of the curvature of the cell body, 

\[
E_{\text{curve}}(\mu) = \xi \sum_{\nu=1}^{s-2} \left( \frac{2\sin(\angle(S_{\mu,\nu}, S_{\mu,\nu+1}, S_{\mu,\nu+2}))}{|S_{\mu,\nu+2} - S_{\mu,\nu}|} \right)^2.
\]

Movement is induced by adding force in the direction from the prior node to the next node. Collective motion is induced by this model, though important aspects such as reversals or individual motility systems are not taken into account.

The elastic filament paradigm was also used by Wu, et al.[91] Instead of a CPM, cells are modeled as a chain of nodes that may have arbitrary position. A Hamiltonian for each cell is given by 

\[
H = \sum_{i=1}^{N-1} K_b(r_i - r_0)^2 + \sum_{i=1}^{N-2} K_\theta \theta_i^2,
\]

where \( r_i \) is the length
of the $i$th link between two nodes in a cell, and $\theta_i$ is the $i$th angle formed by three nodes. Each motility is modeled via a slime field for A-motility and local alignment for S-motility. Cells collide with each other and update by moving the head of a cell and then using the Metropolis algorithm to update where the other nodes are. An internal reversal clock is also included.

Other models have simply looked at the rod-like aspect of Myxobacteria. One model represents cells as self-propelled rectangular particles that have an orientation, $\theta^i$ that interacted via a “soft” volume exclusion. The equations governing this system are:

\begin{align}
  v^{(i)}_\parallel &= \frac{1}{\zeta_\parallel} \left( F - \frac{\partial U^{(i)}_\parallel}{\partial x_\parallel} \right) \quad \text{(1.14)} \\
  v^{(i)}_\perp &= -\frac{1}{\zeta_\perp} \left( \frac{\partial U^{(i)}_\perp}{\partial x_\perp} \right) \quad \text{(1.15)} \\
  \dot{\theta}^{(i)} &= -\frac{1}{\zeta_\theta} \left( \frac{\partial U^{(i)}_\theta}{\partial x_\theta} \right) \quad \text{(1.16)} \\
  v^{(i)}_x &= v^{(i)}_\parallel \cos \theta^{(i)} + v^{(i)}_\perp \sin \theta^{(i)} \quad \text{(1.17)} \\
  v^{(i)}_y &= v^{(i)}_\parallel \sin \theta^{(i)} - v^{(i)}_\perp \cos \theta^{(i)} \quad \text{(1.18)} \\
  U^{(i)}(x^{(i)}, \theta^{(i)}, x^{(j)}, \theta^{(j)}) &= \phi \sum_{j=1,j\neq i}^N \{ [\gamma - a_0(x^{(i)}, \theta^{(i)}, x^{(j)}, \theta^{(j)})]^{-\beta} - \gamma^{-\beta} \}. \quad \text{(1.19)}
\end{align}

The $v_\alpha$ are the velocity in the direction parallel and perpendicular the direction of a cell. The $\zeta_\alpha$ are the drag coefficients for each direction and rotations. $F$ is the self-propelled motive force. The $\gamma$ is related the compressibility of a cell, the $a_0$ represents the overlapping of two cells $i$ and $j$, $\beta$ determines the stiffness against overlapping, and $\phi$ represents the interaction strength. This model leads to clustering of cells, but only loosely applies to myxobacteria or other biological systems[63].

27
2.1 Introduction

Myxobacteria are an interesting group of bacteria. Predominantly soil bacteria, they live in a communal manner that aids in many aspects of a colony’s life cycle, from vegetative expansion to sporulation in fruiting bodies. As a bacteria, the system is simpler to investigate than more complex organisms, but the resulting behaviors can still be compared to those in more complex organisms.

2.2 Structure

The cell of a myxobacterium can be found in two states. The first is a motile vegetative state and the second is an immotile myxospore. In the vegetative state, cells are rod-shaped and metabolically active. Myxospores are round, quiescent, environmentally resistant cells that can survive up to 16 years without nutrient, are resistant to ultraviolet light, and can survive temperatures up to 90° C for brief periods of time. Vegetative cells are much less hardy[65].

*Myxococcus xanthus*, one of the better studied and modeled myxobacteria, have organization according their direction of motion. The rod-shaped cell has a length of 6-7\( \mu m \) and a diameter of 1\( \mu m \) and travels along its long axis. At the leading end, Type IV pili (Tfp) are extended from the cell body. These pili are also found in *Pseudomonas aeruginosa* and *Neisseria gonorrhoeae*, among other bacteria[56].
Some proteins associated with the pili occur at both ends of the cell, but pili are localized at only the leading end[62]. During movement myxobacteria reverse direction, stopping the production of pili at one end and starting production at the other[44]. This is controlled by a network of proteins. The mechanics and effects of pili and the control of reversals will be discussed in the following section on motility.

There are also other surface-bound proteins that are localized to the poles of the cell. Junctional pores similar to those found in cyanobacteria are found mostly at the ends of a cell[88]. These pores have been observed to excrete slime that is thought to be used to propel the cell forward. In wild-type cells only those pores at the rear of the cell excrete slime[44]. These pores and the related motility will also be discussed in the following section.

As a Gram-negative bacteria, the boundary of a myxobacterium is made up of three layers. The inner layer is a cell membrane, the middle layer is mostly peptidoglycan, and the outer layer is another cell membrane. Unlike other Gram-negative bacteria the peptidoglycan layer of the cell is a not a sacculus, or a continuous envelope, that can be extracted from the cell. It is thought that the peptidoglycan of myxobacteria is arranged in a patch-like structure that is held together by other polypeptides[65]. This explanation accounts for the combination of the strength of the cell wall and the flexibility of the cell. This peptidoglycan layer also exists in myxospores in another state, but exactly how the rearrangement between these states occurs is still not understood[20].

On the outside of the entire cell, fibrils extend outward. These fibrils are 15 or 30nm in diameter and up to 50µm in length and are composed of a combination of protein and polysaccharides[21]. Removal of fibrils prevents S-motility, and heavily reduces the cohesiveness of cells[6]. Research also seems to show that fibrils are involved in sensing of and reaction to certain macromolecules[18].
2.3 Motility

Myxobacteria move by gliding motility. This method of movement occurs as a smooth translation over a surface. This movement occurs without flagella and myxobacteria are incapable of swimming in aqueous solution[88]. In *Myxococcus xanthus* the direction of motion is reversed once every approximately 8 minutes under optimal growth conditions. When undergoing a reversal, a cell stops, pauses, and then begins moving in the opposite direction.

2.3.1 Motility systems

It is known that there exist two motility systems in *Myxococcus xanthus*, A-motility and S-motility. A-motility (The A stands for adventurous.) has the ability to move cells in an individual manner. S-motility (The S stands for social.) moves cells only in the presence of other cells. Both of these motility systems are active in wild-type cells, but they can be deactivated independently by mutating numerous genes[28].

S-motility is mediated by Tfp[66, 42, 82, 62]. Tfp are helical structures made from a polymer of PilA subunits. The diameter of a pilus is 5–7 nm and the length can be up to 4µm with an average length of 1–2µm. The force that is exerted during the retraction of a single pilus is 110±30 pN[54]. In *Myxococcus xanthus*, the tips of the Tfp are thought to bind the polysacharride to fibrils that extend from neighboring cells[62]. Motility is caused by a pili originating from the front of a cell being extended and binding to the network of fibrils from other cells. The pili then retracts, pulling the cell that the pili originates from forward. The cells that donate fibrils to the fibril network are pulled back slightly but not to the extent that the pulling cell is moved forward[41].

The engine for A-motility is far less understood. There exist two proposed
propulsive forces for A-motility. The first proposal involves the excretion of a polysaccharide gel from pores at rear of the cell. Work by Wolgemuth showed that this could provide enough force to propel the cell forward[90]. The second proposed force involves transmembrane focal adhesion sites along the length of the cell that act as anchors to the substrate. These adhesion sites then interact with the cytoskeleton inside the cell as a molecular ratchet to provide the motive force for the cell[60]. A schematic of the motility systems is shown in Figure 2.1.

Figure 2.1. Illustration of the possible motility engines of myxobacteria. The cell is moving left with the Tfp at the leading edge and slime extrusion at the rear of the cell. The possible focal adhesion sites are underneath the cell.

The motility systems of *Myxococcus xanthus* have advantages on different types of surfaces. On more aqueous surfaces S-motility is capable of faster movement, whereas on drier surfaces A-motility is the more capable motility system[69]. However, there is evidence that the A-motility system can mutate to a mechanism that is capable of movement in wetter environments like S-motile cells[80].

### 2.3.2 Reversals

Since myxobacteria move on the surface of materials and do not turn easily, reversals provide their method of altering their path to avoid jamming with other cells or other obstacles. The reversal process has been studied quite extensively[68, 30, 29, 44]. It has been shown that the Frz proteins are essential for regulating the reversal period of *Myxococcus xanthus*[10]. As mentioned in Chapter 1, there has been some mathematical modeling of a possible network of oscillating proteins
involving the Frz proteins[29]. The actual mechanism by which the reversals are synchronized in the motility systems is still not completely understood, but there are indications that FrzS and MglA are involved in activating or deactivating proteins at the head of the cell[57].

When a cell reverses, the pili at the front of the cell are deactivated, and pili extend from what was the rear of the cell. Slime extrusion from the rear of the cell stops and begins at what was the front of the cell. The net effect is that the cell pauses for approximately a minute and then moves in the opposite direction along the same path it had been following. The reversal process behaves differently during some stages of development. This will be addressed later in Section 2.5.

2.4 Swarming

During vegetative growth myxobacteria swarm outward. Swarming is the organized movement of a group of agents. In myxobacteria, swarming occurs in the presence of nutrients or prey bacteria as the colony spreads outward. The rate at which swarming occurs depends on many variables, including the nutrient concentration, the initial density of cells in the colony, the concentration of agar, and the motility of the cells[43, 80]. The maximum swarming rates of different motility mutants are that cells with only A-motility, or $A^+S^-$, swarm at the rate of 0.62 $\mu m/s$ and only S-motile, or $A^-S^+$, swarm at the rate of 0.47 $\mu m/s$. However, wild-type cells, $A^+S^+$, swarm at the rate of 1.58 $\mu m/s$, which is more than the sum for the individual swarming rates[43].

One mechanism of organization during swarming is the alignment of nearby cells with each other. One reason for this alignment is simply the shape of myxobacteria[45, 63]. However, the flexibility of the cell body, the constant motion of cells, and reversals also contribute to alignment. Another contributing factor to the alignment
is the fact that A-motility causes A-motile cells to follow slime trails left behind by other cells[13]. This is attributed to the elasticotaxis, or taxis in relation to the stress on the substrate, of A-motility[22].

When approaching prey cells, a flare of a myxobacteria colony will approach near the prey and then turn toward it. It was originally thought that this was a chemotactic response, but the same response was observed when latex and glass beads were placed on the agar[19]. Elasticotaxis is believed to be the cause of this response. Elasticotaxis has been demonstrated by observing colonies of $A^+S^-$ mutants of *Myxococcus xanthus* that are grown on agar that has been compressed in one direction. The colonies expand fastest in the direction perpendicular to the direction of compression[22].

There is some debate over whether chemotaxis plays a role in myxobacteria. It has been shown that gradients in nutrient level have no noticeable effect on the reversal of individual cells[19]. It was also argued that based on the movement speed and the diffusion speed of most chemicals, chemotaxis would not be viable in myxobacteria[19]. Recently, it has been shown that *Myxococcus xanthus* will adjust its reversal frequency in the presence of a gradient of phosphatidylethanolamine (PE). This lipid is hydrophobic and so does not diffuse quickly. However, PE is found in the outer membrane of *Myxococcus xanthus*, and it seems that fibrils are the sensory mechanism. One possible explanation for this arrangement is to allow cells to sense the local density of other cells[18].

2.5 Morphogenetic Development

Under starvation conditions, myxobacteria go through a developmental process that ends in the formation of a fruiting body that is mostly made up of myxospores. In *Myxococcus xanthus*, this process has several stages: rippling, aggregation, and
finally sporulation. The end product is a hemispherical mound of myxospores.

2.5.1 Rippling

Rippling is a behavior that occurs during morphogenetic development in *Myxococcus xanthus*, but in other myxobacteria it occurs not only at the beginning of swarming but also when the colony encounters food[9]. The large-scale structure of ripples has the appearance of standing waves. These waves appear to pass through each other. However, when individual cells are tracked by making a small subset of the population luminesce, individual cells do not move much more than a wavelength or two[83]. Instead of the waves passing through each other like a pair of solitons, the cells in the waves are completely reflected[83].

Rippling is accomplished by the interaction of a surface-bound protein, C-signal. In *Myxococcus xanthus*, starvation causes cells to release a mixture of short polypeptides called A-signal. The detection of A-signal leads to the production of C-signal. C-signal is made up of a signaling protein and receptor protein, both of which are localized to the poles of the cell[51]. C-signal interacts with other proteins in two important ways. The first is that it is self-promoting: activation of C-signal leads to more production of C-signal. The second is that it interacts with the Frz proteins to alter the reversal frequency of cells. The rippling phase is not required for fruiting body formation, so the reason it occurs is still unknown. However, it is thought to aid in aligning cells toward aggregation sites[84].

2.5.2 Aggregation

Aggregates are formed as cells accumulate in various locations on a surface. This is accomplished by the streaming of cells towards the aggregates[4]. An aggregate begins as a jam of cells. This jam is formed by cells that are immobilized due to being entwined with each other[32]. As a cell encounters this jam, it has three
possible responses: it can become jammed itself until it reverses, it can bend and turn to move around the jam, or it can climb over the jam. As enough cells turn to go around the jam, the aggregate becomes circular. Eventually, the jam dissolves and the aggregate becomes toroidal. As more cells join the aggregate, the inside fills and the aggregate mounds up[32].

2.5.3 Sporulation

As aggregation continues, C-signal accrues on the surface of each cell, and once it reaches a threshold, the cell undergoes a physical change and becomes a rounded spore. The exact process by which this rounding occurs is still unknown, but it is thought that increased cross-linking in the peptidoglycan layers may play a role[65]. Sporulation can also be induced by exposing cells to glycerol, lysozyme, and other cell-wall-damaging agents[65]. However, these induced spores have a different outer structure than those that are created by starvation-induced development[21]. One reason for this is that approximately 90% of cells autolyse during the developmental program, and this releases one of the proteins from the lysed cells that makes up the outer layer of starvation-induced spores[21].

2.6 Fruiting Body Structure

The resulting spores have arranged themselves into a structure that varies based on the species of myxobacteria. In some species the spores are protected by a tough sac called a sporangiole. In M. xanthus, the spores are covered only by slime and form into a hemispherical mound that eventually lifts itself and forms into a spherical structure. In other species the morphology of the fruiting body takes on various treelike structures[65, 21].

The make-up of the fruiting body for M. xanthus is a mound of myxospores protected by a layer of slime. The body as a whole measures about 200 µm in
diameter and is made up of approximately 100,000 cells[53, 65].

When conditions around the fruiting body improve, due to either changes locally or transport, the body germinates and the colony already contains thousands of cells. The germination of spores involves the breaking of the myxospore coat, and the emergence of a motile cell. The coat remains behind as the cell moves out to explore its surroundings[65].
CHAPTER 3

LATTICE GAS SWARMING MODEL

3.1 Overview

The lattice based swarming model is a two-dimensional Hybrid Lattice Gas Cellular Automaton/Reaction-Diffusion Model. The underlying lattice has hexagonal symmetry. Cells are represented as lines, and are tracked by their center of mass. Growth and nutrient are included as well. First, the structure of the simulation is discussed, followed by a discussion of its implementation. Finally, the results of the simulations are discussed, followed by possible future directions.

3.2 Description of Structures

There are two structures that make up the model: the mediums which are the sites of the lattice, and the cells that move on these sites.

3.2.1 Medium

Each lattice site defines a place where cells may be centered and overlap. These represent the agar on which the cells move. Each site is defined by its neighbors, what cells are centered there, how many cells overlap in each of the three orientations, nutrient, and how much slime has been deposited in each of the three orientations. An orientation describes one of the three lines that cross through a
lattice site. Direction will denote one of the six rays that originate from a lattice site.

3.2.2 Cell

Each cell is defined by the position of its center of mass, direction of movement, the amount it is bent, and its length. From these four state variables, the extended domain of the cell is determined. Whether the cell has turned or moved this time step is also tracked. A representative cell is displayed in Figure 3.1.

![Figure 3.1](image)

Figure 3.1. This cell has a length of 8 sites and is moving to the right. The sites of the cell are indexed from the back of the cell to the front of the cell. The center of mass of the cell is always at \( \lfloor \frac{\text{length}}{2} \rfloor \). The cross hatched spot is the center of mass, and the light green slanted hatched spots are the rest of the body of the cell. The black circles are included to show the hexagonal lattice.

3.3 Description of the motility algorithm

The important part of the motility algorithm is the process for determining which direction to turn. Due to biological considerations, cells are only allowed to turn 60° to the left or right, or not turn at all. Data related to turning in each of these 3 possibilities is collected. From this data, a weight, \( w_i \), is assigned to each outcome. Then a Monte Carlo decision step is done to decide which direction to turn. The probability for turning in the \( i \)th direction is \( \frac{w_i}{\sum w_i} \). A random number between 0 and 1 is generated and used to determine which direction to turn.
3.3.1 Physical Alignment

Motility alignment is modeled as having three factors: neighboring cells, pili attachment sites, and slime. The neighboring cells have several effects that are modeled for each cell. The first effect is alignment of the elongated cells side to side. The area checked for neighboring cells to align with is the black lattice sites surrounding the area of blue north east hatched, red north west, and green horizontal sites in Figure 3.2(a).

![Figure 3.2](image)

Figure 3.2. (a) A cell checks the sites neighboring the possible postions, or those that have a curve passing through them, for cells that this cell would coalign with. (b) A cell checks the three domains to discover how many pili attachment sites exist in each domain.

The second factor caused by neighboring cells is deflection, or the change in direction caused by running into other cells. The site directly in front of the cell is checked to see if cells are present there.

The third factor taken into account is physical resistance to the presence of other cells. For each of the possible turns, how many occupied lattice sites the cell would cross is calculated. This region corresponds to the solid circles in Figure 3.2(a).
3.3.2 Alignment due to motility systems

Alignment with slime is checked at the center of the cell, in order for the cell to stay on a slime track after it turns. Slime is not produced by $A^-S^+$ cells, and so slime is not modeled when modeling these mutants.

Alignment due to pili attachment is modeled as cells wanting to turn towards the area with the most sites to attach to. To accomplish this the regions in Figure 3.2(b) are scanned to count the number of cells overlapping at each of these sites. This is not calculated in $A^+S^-$ cells.

A direction is picked according to the Monte Carlo step, and the cell bends at the middle to point in that direction. If the cell is S-motile, it checks the horizontally-lined cyan area in Figure 3.2(b). If no cell is in that area, the cell stalls with some probability that represents the ratio of A-motility and S-motility. This ratio is taken from the speeds from the 1983 paper[43] and the 1999 paper[75]. When modeling $A^-S^+$ cells this probability is 1.

3.3.3 Movement

After all cells have aligned, each previously unstalled cell attempts to move one lattice space forward. If the number of cells overlapping in the space in front of the cell, $N_f$, has less than a threshold number of cells overlapping at it, or if $N_0r \geq N_f$, where $N_0$ is the number of cells overlapping at the center of the cell, and $r$ is a uniform random number in $(0,1)$, then the cell moves forward. This is to represent pseudo-three-dimensionality of the model. If neither of these conditions are fulfilled, the cell stalls. If the cell moves, it straightens out as it moves.

3.3.4 Slime Production

When modeling A-motile cells (wild-type or $A^+S^-$), after all cells have moved or stalled, each cell deposits one unit of slime in the orientation of the back half of
the cell at the last site of the extended cell. This would be site 0 in our example cell in Figure 3.1.

3.4 Description of each time-step

Each time-step has two parts: movement, and growth and diffusion. The movement step consists of using the alignment, movement, and slime production as described above. The number of times this movement step is repeated depends on which motility mutant is being modeled. With a lattice spacing of \(0.6 \mu m\) and a time-step of \(0.63166\) min/time-step, which results in a speed for an unobstructed cell with enough cells for S-motility of \(3.8 \mu m/\)min, and cell length of \(4.2 - 7.8 \mu m\).

3.4.1 Growth and division

The growth algorithm involves two steps: nutrient absorption and elongation. At each site where a cell overlaps, the nutrient level is checked to see if it is above minimum threshold for growth. If it is above the threshold, then the cell absorbs one unit of nutrient from that lattice site. If a cell absorbs nutrient at more than half of its sites, then it has a chance to elongate. The probability to elongate is such that under sufficient nutrient conditions, the cell reaches twice its minimum length in 210 simulation minutes on average.

Once a cell grows to twice its minimum length, it tries to divide. In order to divide, the cell must have enough space at the sites at \(\frac{1}{4}\) and \(\frac{3}{4}\) of its length. Each of these sites is checked to see if they have more than an overcrowding threshold of cells centered at them. If not, the cell divides. The two new cells prefer to be aligned in the direction that the original half of the cell was. If it cannot do that because another cell occupies that path, then it tries to antialign from its old direction. If it cannot do that, then it randomly chooses a direction at its center.
After all cells have undergone the growth algorithm, nutrient diffusion is solved via a finite-difference method.

3.4.2 Reversal

Each cell has an internal clock. Every time a cell reverses, the time until the next reversal is sampled from a binomial distribution that has a mean of 9.79 minutes and a standard deviation of 1.7584 minutes. The initial reversal time is sampled from a uniform distribution from 0 to 10 minutes.

3.5 Implementation

This model is stochastic in nature and could be formulated as a hybrid discrete Markov Chain continuous differential equation model. However, enumerating the requisite vector and system is an unwielding process. This formulation is also not conducive to making simple changes to the assumptions of the model. As such, the model is implemented in a cell-based algorithm.

The algorithm has been implemented in the C programming language as shown in Appendix A. While C is not an object-oriented programming language, the program is organized with structures and algorithms devoted to cells, and ones for the underlying agar. The GNU Scientific Library Random Number Generators[24] were used. The default mt19937 generator was used for all simulations.

3.5.1 Structures

Medium Structure

The underlying growth medium is implemented as a C struct, medium. This struct has several fields. The first field, neighbor, is an array of six pointers that catalogs which other mediums are connected to this one. The second field, resident, is an array of twelve pointers that catalogs the cells that are centered
at this location. The first six pointers are for moving cells, the next six are for cells that have moved or reversed recently, or are jammed. The next field, next, is a pointer that tells us which medium is next when iterating through the simulation domain. The next two fields, numcells and oldnumcells, are arrays of length three that keep track of how many cells overlap at this location in each of the possible orientations. Another array of length three, slime, tracks the amount of slime deposited in each orientation. The next field, numhere, tracks how many cells are centered at this lattice site. The final three fields, old_nutrient, nutrient, and new_nutrient, track nutrient values for the $t - 1, t, t + 1$ timesteps.

Cell Structure

Cells are also implemented as a C struct. The first field for a cell, body, is an array of pointers to those mediums that make up the body of the cell. The second field, center, is a pointer that leads to the medium that is the center of mass for this cell. The rest of the fields are integers that describe the state of the cell. The first, turn_time, keeps track of the countdown clock for reversals. The next, length, tells us what the length of the cell is. The next two, direction and bend, describe which direction the cell is aligned towards, and how much it is bent. The last three, moving, jammed, and turned, are binary switches of whether the cell is moving, jammed, or has turned yet.

3.5.2 Functions

Medium Functions

The following functions are associated with the medium because they are associated with a spatial aspect of the model. They can be divided into three groups: those dealing with stepping through the lattice to update the state of cells, those dealing with collecting information about the state of the system, and those dealing
solely with updating medium structures. Those in the first group are move_cells, align_cells, moved, reversal, and growth. The second group consists of get_cells_nbhd, get_cell_align, get_obstacles, and get_deflections. Those in the third group are medium_init, medium_addneighbor, set_numcells, slimeit, diffuse_nutrient, and set_nutrient.

- **move_cells** - This function simply tells each cell located at this medium to move if possible. It also clears the turned field and resets the jammed field for each cell. When a cell moves it sets the moved state, and moves into the 6 + i cell field in neighbor i, where i is the direction of motion of this cell, if it is unoccupied.

- **align_cells** - This function determines in which direction each cell will turn during the alignment phase. To start, an index, i, is selected with uniform probability for resident to eliminate any bias in alignment. i is then incremented through all 12 indices in order {i mod 12, (i + 1) mod 12, ...(i + 11) mod 12}. For each index, resident[i mod 12] is checked for a cell. Then turned is checked to see if it is true for that cell, i.e. that cell has already been aligned. If not, data is collected from the surrounding area using the functions from the second group:

  - get_cells_nbhd(\(e_i\) below) - This function calculates how many cells are in the regions in Figure 3.2b. It simply moves around the lattice in a branched fashion to explore each region. The function returns the sum of the multiplicity of cells overlapping at all of the mediums in each domain.

  - get_cell_align(\(a_i\)) - This function determines how many cells would be co-aligned with this cell in the black sites in Figure 3.2a if it turns into the direction adjacent to those sites.

  - get_obstacles(\(c_i\)) - This function calculates the number of occupied sites that the cell would cross if it turned to that direction.

  - get_deflections(\(b_i\)) - This function calculates the number of cells that cross the path of this cell directly in front of this cell. It returns the number of cells that would be parallel to it if it turned left or right, respectively.

Then this information is used to calculate the probability of turning left, right, or staying straight. The weight for each direction is then

\[
w(i) = e^{w(i)} = e^{\text{ALIGN} \times a_i + \text{DEFLECT} \times b_i - \text{OBSTACLES} \times c_i + \text{SLIME} \times d_i + \text{NBHD} \times e_i},
\]

where SLIME, ALIGN, OBSTACLES, NBHD, DEFLECT, and \(\beta\) are parameters and \(i \in \{-1, 0, 1\}\) represents turning left, straight and right, respectively. The variables for these values come from the respective functions. The \(d_i\) are collected from the slime field at this medium. If the cell is already bent in direction \(j\), \(w(j) = 0\) is set to prevent a 120° bend. The weights
are normalized into a probability by \( P(i) = \frac{w(i)}{\sum_{i=1}^{N} w(i)} \). A uniformly distributed random number, \( x \), is generated in \((0, 1)\). If \( x < P(-1) \), the cell turns left. If \( P(-1) \leq x < P(-1) + P(0) \) it stays straight. And if \( P(-1) + P(0) \leq x \), it turns right. If the chosen direction already has a cell in it, the cell doesn’t turn.

- **moved** - This function simply moves cells that have space into a situation where they can move. Cells in \( \text{cells}[i+6] \) are moved to \( \text{cells}[i] \) and then \text{moving} is reset to one.

- **reversal** - This function checks if cells at this \text{medium} can reverse. If \( \text{turn.time} \leq \text{time} \), where \( \text{time} \) is the global time, the cell reverses, as long as there is not a cell already in that direction. If there is a cell blocking reversal, the cell stalls until it can reverse, i.e. \text{moving} = 0 and \text{jammed}=1.

- **growth** - This function has cells consume one unit of \text{nutrient} from each site in \text{body} for each cell in \text{cells}. If \text{nutrient} < \text{NUTRIENT.CUTOFF}, that site does not absorb nutrient. If over half of the sites in a cell absorb nutrient, then the cell has a chance to grow with a probability that is set so the average growth time matches that of experiment. This is done by calling \text{grow.cell}. \text{grow.cell} is also called if the cell is already twice its minimum length and can divide.

- **medium_init** - This function is used to initialize a medium. All values are set to zero or \text{NULL}, except for \text{nutrient} and \text{old_nutrient}, which are set to an initial nutrient value.

- **medium_addneighbor** - This function sets the neighbor of a \text{medium} given another \text{medium} and the direction it is in.

- **set_numcells** - This function simply copies the values in \text{numcells} to \text{old_numcells}.

- **slimeit** - This function has each \text{cell} in \text{cells} increment the appropriate \text{slime} field in its tail.

- **diffuse_nutrient** - This function carries out a finite difference diffusion method. A Du Fort-Frankel scheme was modified to solve on the hexagonal lattice. The no-flux boundary conditions maintain the second-order accuracy of the method.

- **set_nutrient** - This function moves the value in \text{nutrient} to \text{old_nutrient} and \text{new_nutrient} to \text{nutrient}.

**Cell functions**

The following functions are associated with the cells, because they actually carry out the processes of cells. They are:
• **extend cell** - This function sets the medium in body of a cell. To do this it starts at a pointer to a medium, medium_ptr, and sets it to body[length/2]. Then for those sites in front of the center of mass, the function associates medium_ptr->neighbor[dir] with body[length/2+1] and so forth until it has associated body[length] with the neighbor of the neighbor of ... the neighbor of body[length/2]. For the rear of the cell it associates medium_ptr->neighbor[(dir-bend+3)%6] with body[length/2-1] and so forth until it gets to body[0].

• **cell_init** - Given a medium, a direction, and a length, this function creates a cell at medium with dir of direction, and a length of the length given. If it is initializing cells at the beginning of a simulation, it sets a random length and a random turn time.

• **getnumcells** - This function returns the number of cells that are at a medium. It sums the values in oldnumcells and returns them.

• **cell_move** - This function moves a cell in the direction it is pointing. If the number of cells at the medium in front of the cell is less than CELL_THRESHOLD, it moves. Otherwise, it checks how many cells overlap this one. The function then divides this number by length and multiplies by a uniformly distributed random number in (0,1). If this is greater than the number of cells in front of the cell, it moves.

  If the cell moves, it moves to the cells[dir+6] of medium_ptr->neighbor[dir], decreases the numcells at all of the mediums that it is leaving and all that it is moving to, updates the numhere of the old medium_ptr and the new, straightens if it was bent, and sets moving=0. If it does not move, it sets moving=0 and jammed=1.

• **cell_turn** - This function is basically for bookkeeping. It decrements all the appropriate numcells in the body of this cell. It then sets the resident that it is in to NULL and points the correct resident to itself. It then uses extend cell to set the body pointers appropriately.

• **cell_reverse** - Similar to cell_turn, this function takes care of the bookkeeping. The only difference is that moving is set to 0, and the resident that points to the cell in the ending is in the upper six.

• **grow cell** - This function is in charge of increasing cell_length by one, and if that makes the cell long enough to divide into two cells, it does. First, it checks how many cells overlap the medium that the cell will extend into by summing the numcells at that medium. If length is even, simply add the appropriate medium to the front of the cell. If it is odd, add a medium at the rear of the cell. If this medium has less than a cutoff of cells overlapping at it, the cell grows. In the case of even length, it simply uses extend cell to add a medium to the front and increment length. If length is odd it moves all the pointers in body to the next highest spot and adds a pointer to the appropriate medium. This ensures that center points to the same medium as body[length/2], in integer arithmetic.
Once through the growth process, a division process is done if length equals MIN_LENGTH. First, check if the number of cells centered, numhere, at what will be the center of the two daughter cells is less than a threshold. If it is, go ahead with division. The daughter cells prefer to be in the same direction as the mother cell, and if that is not possible, it prefers to be in the same orientation. If that fails, the cell is deflected to a random direction. If no free direction can be found, the daughter cell dies.

3.6 Simulations

3.6.1 Experimental Results for Rate/Density Relationship

Before discussing the results of the Lattice Gas Swarming model, a discussion of the pertinent experimental results is necessary. The experimental expansion rates of a swarm were found to be fit by the function

\[ r = A + B \left(1 - e^{-\frac{x}{C}}\right) \]  

(3.2)

where \( x \) is the initial density of cells, \( A, B, C \) are fit parameters, and \( r \) is the linear expansion rate of the swarm. For wild-type cells the best fit was \( r = 0.1 + 1.48(1 - e^{-\frac{x}{48}}) \). For \( A^+ S^- \) mutants the best fit was \( r = 0.1 + 0.52(1 - e^{-\frac{x}{20}}) \), and for \( A^- S^+ \) the fit was \( r = 0.47(1 - e^{-\frac{x}{190}}) \). An interesting note is that the sum of the limit rates of expansion for mutants does not equal the rate for wild-type cells. [43]

3.6.2 Description

Each simulation is run using one of two methods. The first starts by placing a circular region of cells in the center of a square simulation domain. The boundary conditions of this system are irrelevant for cells, because the run time is shorter than the time for cells to reach the edge. For the diffusion solver, a periodic boundary condition is used. The second method involves a rectangular domain with cells placed on the far left of the domain. The top and bottom boundaries are periodic and the left and right boundaries are no-flux boundaries. This method is to simplify the first method by approximating a sector of the circular domain. In the first method,
cells spread outward in all directions and in the second, cells spread rightward. Representative initial states are shown in Figure 3.3.

In Kaiser and Crosby[43], cells were allowed to swarm, and the distance swarmed from the original boundary was averaged over 6 points. The same is done for simulations by radially averaging the distribution. A representative distribution is shown in Figure 3.4. To determine where the edge of the colony is, a cutoff value is determined by comparing the distribution with snapshots of the the simulation. The point at which the colony drops below this threshold is the boundary. Tracking this point, one notices that the colony grows linearly as seen in Figure 3.4. The slope of a least-squares fit to this data is the speed of expansion of the colony.

3.6.3 Parameter Space Exploration

Due to the large parameter space, a gradient method was used to attempt to maximize the expansion rate. After picking a biologically plausible set of parameter values, the initial density was changed from a low value to a high value, as in experiment, and the expansion rate was traced. Then a density was chosen that is slightly higher than the linear region in the initial density versus expansion rate graph and the following algorithm was applied at this density:
Figure 3.4. A representative radial distribution is on the left and a representative distance versus time plot is on the right.

1. Choose a subspace of the parameter space.
2. Explore the area near the current point.
3. Choose the next point in the direction of maximal increase on that subspace.
4. Go to Step 1.

The values that were reached are listed in Table 3.1.

3.6.4 Simulation Results

Simulations were run at the values in Table 3.1. A combination of the two initial conditions and simulation domains were used. The results are plotted in Figure 3.5. Fitting the data to Equation 3.2 results in the parameters $A = 0.26 \pm 0.13$, $B = 0.38 \pm 0.13$, and $C = 80 \pm 56$. If only considering those values from the circular initial conditions, the best fit equation is given by $A = -0.39 \pm 0.09$, $B = 0.90 \pm 0.09$, and $C = 91 \pm 15$. For those values from looking at the rectangular domain, the best fit parameters are $A = 0.27 \pm 0.20$, $B = 0.62 \pm 0.20$, and $C = 58 \pm 37$. 
### TABLE 3.1

VALUES REACHED BY THE GRADIENT PARAMETER SWEEP

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELL_THRESHOLD</td>
<td>0</td>
</tr>
<tr>
<td>MIN_LENGTH</td>
<td>7</td>
</tr>
<tr>
<td>GEN_TIME</td>
<td>210.0</td>
</tr>
<tr>
<td>PRESSURE</td>
<td>3</td>
</tr>
<tr>
<td>S_THRESHOLD</td>
<td>3</td>
</tr>
<tr>
<td>MAX_S</td>
<td>100</td>
</tr>
<tr>
<td>ALIGN_THRESH</td>
<td>10</td>
</tr>
<tr>
<td>SYN_DEFLECT</td>
<td>6.0</td>
</tr>
<tr>
<td>SYN_ALIGN</td>
<td>6.0</td>
</tr>
<tr>
<td>SYN_SLIME</td>
<td>8.0</td>
</tr>
<tr>
<td>SYN_DENS</td>
<td>4.0</td>
</tr>
<tr>
<td>SYN_OBS</td>
<td>20.0</td>
</tr>
<tr>
<td>BETA</td>
<td>2.0</td>
</tr>
<tr>
<td>STRAIGHT</td>
<td>1.0</td>
</tr>
<tr>
<td>SLIME_DRY_COEF</td>
<td>0.01</td>
</tr>
<tr>
<td>S_TWITCHING</td>
<td>0.99</td>
</tr>
<tr>
<td>REVERSE_PAUSE</td>
<td>0.5</td>
</tr>
<tr>
<td>S_A_RATIO</td>
<td>0.4</td>
</tr>
</tbody>
</table>
Figure 3.5. Results for simulations at the values in Table 3.1. The upper curve is the curve from experimental results. The lower dashed curve is fitted to the results from simulations with a circular initial domain, the triangular points. The upper dashed curve is fitted to the results from simulations with a rectangular domain, the circular points. The lower solid curve is fitted to all simulations.

The variation in the rectangular domain simulations arises from the fact that due to the width of the simulation domain, only one peninsula is being formed. The peninsula may extend quickly, or it may curve back into the initial domain. When and if this happens is the determining factor for the expansion rate for the system.

Simulations were also run for mutants. The results are plotted in Figure 3.6. For $A^+ S^-$ the parameters for Equation 3.2 are $A = 0.088 \pm 0.004$, $B = 0.175 \pm 0.004$, and $C = 199 \pm 13$. For $A^- S^+$ the parameters for Equation 3.2 are $A = -0.025 \pm 0.007$, $B = 0.125 \pm 0.005$, and $C = 279 \pm 24$. 

51
3.6.5 Other Observations from Simulations

There are several other effects that are noticeable from watching the simulations. The first is that there appears to be a spacing between peninsulas. A minimum distance is required by the fact that S-motility will pull two peninsulas together. An example of this is shown in Figure 3.7. However, there is no obvious rationale that a maximum distance should exist. There may be a standing wave instability at the edge of the colony.

Figure 3.7. An example of merging peninsulas in simulation.
Another aspect also noticeable in Figure 3.7 is the fact that peninsulas maintain themselves and bend similar to experimental peninsulas. However, unlike in experiment, if a loop of cells happens in a simulation, it contracts upon itself. In experiments, loops maintain their position and new peninsulas are generated from them. An example is shown in Figure 3.8. Relatedly, it is apparent that at low initial densities in the model the colony collapses in on itself. This is different from the experimental results. However, this may be due to the increased density of cells at the edge of the drop as the liquid dries.

![Figure 3.8. An example of a peninsula emerging from a loop in experiment.](image)

3.7 Discussion

The model is successful in reproducing the shape of the curves for changing density found in experiment. For wild-type cells, the $C$ values for experiment and simulation overlap. The “zero density” spreading rate, $B$, for mutant cells is also close to the experimental levels. The ratio of the limiting values of $A^+S^-$ spreading and wild-type cells is 0.44 which is close to the experimental value of 0.39. However, the ratio between the limiting rates of $A^-S^+$ and wild-type cells is only 0.16 as opposed to the experimental value of 0.30. For neither mutant are the $C$ values similar to experimental values.

It should be noted that the observed expansion rate for the experimental results
can be achieved by the model, as is shown by the values above the experimental curve.

3.8 Future Work

The area that begs the most work is looking into how to model S-motility. Chapter 5 discusses an attempt to look at specifically this problem. Continuing to explore parameter space may increase the rate of expansion. It may also be helpful to attempt to derive a continuous limit of this model. The factors that seem most appropriate for taking a limit with are those represented by parameters SYN_DEFLECT, SYN_ALIGN, and SYN_DENS.

This model could also be modified to model other organisms. Ant foraging or human fishing effects are a couple of examples that would be able to take advantage of trail laying and following aspects of the model for ants and the ability to model underlying nutrient dynamics would help in examples for fishing.
CHAPTER 4

LATTICE BASED FRUITING BODY FORMATION MODEL

4.1 Introduction

Prior versions of the Lattice Based Fruiting Body Formation model were successful in modeling aggregation and the beginning of fruiting body formation[73, 74]. These versions of the model made the assumption that cells did not reverse. This assumption was made based on the then current thought in biological research[37]. However, it has since been shown that during aggregation cells do reverse[72]. In response, reversals were added to the model.

4.1.1 Description of the Model

This model is a three-dimensional (3D) biological Lattice Gas Cellular Automata based on rules suggested by observations of myxobacteria. The model incorporates movement, reversals, C-signaling, slime trail following, and sporulation. The cell body is represented as a collection of pixels in a 3D face-centered cubic lattice, with the (111) planes acting as the horizontal planes of the simulation. This gives hexagonal symmetry in the x-y plane, and each site has 12 nearest neighbors: 6 in the same plane, 3 above the site and 3 below.

Once the geometric shape, orientation, and size of a cell are provided, the pixels that comprise the cell are automatically assigned. Rod-shaped cells are approximated as ellipsoids, and spores as spheres of the same volume. Each cell has a head
area and a tail area at the leading and trailing poles, respectively. The sizes of these areas are equal, but can be varied, along with the size of the cell. For this work, cells have a length of 11.0 and a diameter of 2.4 lattice units.

The long axis of the cell can be oriented along one of the 12 lattice directions (or channels). Only one cell can occupy each channel at the node at the cell’s center of mass. This exclusion rule allows the overlapping of the extended bodies of cells. Cells are allowed to bend 60 degrees to account for the flexibility of actual myxobacteria. To simulate surface gliding, cells must maintain contact with the bottom plane, unless they are climbing on top of other cells. Cells only climb if all neighboring nodes in the current layer are occupied. Once on the upper layer, the cell moves in the direction that maximizes the overlap of its pole areas with the local slime density.

The head area is also used for determining alignment of the cells. Cells turn in a direction according to the slime trails left in the area by prior cells. Each time step a cell is allowed to turn 60 degrees in either direction or remain in its current channel. To determine which direction the cell will choose, one calculates the probability of turning in each direction. This probability is determined by the overlap of the head area with the slime density field, which stores the number of slime trails that crossed a node the previous time step. The probability of choosing orientation \( (i) \) is

\[
P_i = \frac{\exp(\beta C(i))}{Z},
\]

where \( \beta \) is an alignment parameter, \( Z \) is a normalization factor, and \( C(i) \) is a function describing the overlap of the head area with slime trails when the cell is oriented in the \( i \)-th direction.

The head and tail areas also act as C-signaling sensitive regions. When a polar region from one cell overlaps with the polar region of another cell, both cells increase their C-signal by a unit. C-signal acts as a developmental clock. Once it has crossed a threshold, the rod-like, motile cell differentiates into a non-motile round
spore. Spores have a radius of 2.5 lattice units. Spores are then pushed by motile cells using simple mechanical collisions to determine the direction of motion of the spores.

Each cell also contains a reversal clock. The clock is randomly assigned a value with a uniform distribution centered on an average reversal period. Each time step the clock is decreased by one, until it reaches zero. Then the cell is reversed by simply reorienting it in the opposite direction, and the clock is reset using the same distribution. To represent the pausing seen in observations, the cell is held stationary for one time step after reversal.

4.2 Results

The initial conditions for the simulations are the same as in Sozinova, et al.[74] There is an initial jam in the center of the simulation domain. The lowest level of the simulation domain is populated with fixed density. All cells start with no C-signal. The simulation is then allowed to run with a constant rate of cells entering the domain. After 4500 time-steps, the time when a definite rounded aggregate has formed, the jam dissolves and cells begin to sporulate. The simulation then runs for another 4000 time-steps. The time of these events corresponds to 4500 time-steps≈24 hours or about 20 seconds per time-step. This differs from the previous model in which the time-step was approximately 30 seconds.

A representative plot of cells just after the unjamming and then another at the end of the simulation are shown in Figure 4.1. One can see that rounding of the fruiting body occurs during aggregation and continues as cells sporulate. The number of spores is sampled at time-steps throughout the simulation after the jam dissolves.

To see what the effect of reversals on sporulation was during different situations,
two conditions were changed. First, the initial density of cells was varied. The results are plotted in Figure 4.2. Varying the initial density did not change the response, however there is a mild maximum in the range of 30-40 time steps.

Second, the rate at which cells were introduced to the simulation domain was varied. The results appear in Figure 4.3. One can see that as the rate at which cells are added to the domain approaches the value used in the density variation, 30, the resulting sporulation percentage approaches the values from the density variation results. However, if one averages over the variations of the rate at which cells are added, the plotted curve results. This curve has a maximum around 30 time-steps which corresponds to about 10 minutes, the natural reversal frequency of cells in experiment.

4.3 Conclusions

A reversal clock was added to cells in the model. The resulting model retains the ability to model aggregation and fruiting body formation. Varying the initial density of the simulation has little effect on the final number of spores. However, varying the rate at which cells are recruited to the aggregate does have an effect. If
you look at the rate of sporulation with different rates of influx, there is a maximum for sporulation near the reversal period that corresponds to the natural frequency. This implies that the reversal frequency that will produce the maximal amount of spores is not necessarily for a single density, but for a range of densities.

4.4 Future Work

The stability of the maximum with respect to the other parameters of the model is unknown. The factors that effect the maximum and how strong the maximum is may help the discussion of why reversals exist in myxobacteria and why they happen with the observed period. While experiments that look at what the effect
Figure 4.3. Results from varying the rate of adding cells to the simulation domain.

of changing the density of cells on sporulation exist, they are from an evolutionary perspective[40].
CHAPTER 5

OFF-LATTICE SWARMING MODEL

5.1 Introduction

In this chapter, the off-lattice model of myxobacteria swarming will be discussed. First the model will be described, followed by a description of its implementation. Then some preliminary results will be discussed, and finally possible extensions and improvements on the model will be stated.

5.2 Description of the model

The model is a variant of a self-propelled particle (SPP) model. SPP models were originally designed for simulation of flocking of birds or schooling of fish. Each element is represented as a point with a velocity vector. The elements have three areas of interaction. In the closest area, elements repel each other, and in the furthest, they attract each other. In between, elements align with each other. These rules lead to the development of flocking or schooling patterns.

Several modifications were made to this model. The basic elements, or cells, are rays, having a length and direction, instead of being only point particles. Each cell is represented by the x and y coordinates of its rear end, the angle it makes with the x-axis, and its length. (See Figure 5.1.)

S-motility is modeled as the process of extending and retracting pili. To model this, a cell is randomly selected to move. Then an attachment point is randomly
selected in a sector of the circle centered at the head of the cell with a radius equal to the maximum length of a pili. Then the amount of fibril at this point is calculated. To do this, cells that are less than a fibril length away and can contribute a fibril that is anchored in the sector of a circle that is away from the cell are counted. If the number of such cells is above a threshold, the pili attaches. (See Figure 5.2.)

Once the pili attaches, the cell is moved towards the attachment site. However, the need to account for collisions with other cells is important. To do this, there is a check for other cells in the triangle formed by the cell’s ends and the attachment site. Also, cells that cross the moving cell are ignored to allow for unjamming from random initial conditions and mistakes. (See Figure 5.2)

The cell that the moving cell will encounter first, or the obstacle is tracked. If the point of collision is near one of the ends of the obstacle, then the moving cell is moved in its direction a distance proportional to the dot product between the vector from the point of collision to the attachment site and the vector for the obstacle.
If the collision is not near an end, the moving cell has a chance of overrunning the obstacle, albeit at a slower rate than if there was no collision.

![Diagram](image_url)

Figure 5.2. The large cell has selected the large dot as its attachment site from the sector in front of the cell. There are three cells contributing to the pili at that site because they are in the sector that is evenly divided by the line from the head to the attachment site. The triangle formed by the cell and the attachment site is searched to determine if cells are in the way of motion. One cell is in this area.

5.3 Implementation

The algorithm has been implemented in the C programming language. The source code is in Appendix B. The GSL random number generator, mt19937, was used for any random numbers used in the algorithm.

5.3.1 Structures
Cell Structures

The cell structure contains several pieces of information. This includes the position of the cell in an array of length two, position, its direction in a field
angle, its size in an array length, its next reversal time in a field clock, what agar structure it is at in a pointer here, and which cell was initiated after it in a pointer next. The minimum value for length is 0.25 and its maximum is 0.5, and angle is in the range \((-\pi, \pi)\).

Agar Structures

The agar structure describes the spatial domain of the model. Each agar is a member of the square lattice that makes up the simulation domain. agar are squares that have sides of length 1. agar contain two fields: the first is an array of pointers towards the cell structures that are in this agar, cells, and the second is simply an counter of the number of cell structures that are at this agar, or num here.

5.3.2 Functions

The functions for this simulation can be divided into two categories: those dealing with individual cells, and those dealing with the spatial interaction of many cells. Those that deal with the movement of cells are S_move, move_cell, check_move, and reverse. The functions that account for the spatial aspect of cell interaction are fibril_dens, check_triangle, and add_cell and remove_cell.

Cell Movement Functions

- S_move - This function carries out the algorithm for simulation of S-motility. A pointer to a cell, this is passed to the function. Three points are stored in an array, checkpoints: the position of the head of this, the position of the tail, and the point that is chosen to be used as a pili attachment site. The pili attachment site is uniformly selected at random within a sector. The position is generated by uniformly selecting an angle, \(\theta\), within the sector, and then selecting a distance, \(r\), which is the maximum distance times the square root of a uniformly distributed random number on \((0, 1)\).

  If the density at the pili attachment site, as determined by fibril_dens, is high enough, this will pull towards that site. Then the values of an array, barrier, are set to track the distance that this will move in the x-direction, the y-direction, and what angle the cell will need to turn. Then this information is passed to check_triangle, which checks what cell this will
encounter first while moving, obstacle, and how far that collision is away from this->position in the x- and y- directions and how much this needs to turn to go directly to that collision point.

If this collides with another cell, it is moved to the point of collision. Then the collision site is checked to see if it is in either of the end fifths of obstacle. If it is, the dot product between the vector from the collision site to the pili attachment and the unit vector in the direction of obstacle is taken. This value is multiplied by PUSH_RATIO, which represents the proportion of the motive force that gets transferred to obstacle, to get dist. obstacle and dist are then passed to check_move. obstacle is then moved the amount returned. Then this is moved if the collision occurred at the head of this such that the head of this is at the same place on obstacle as before it moved obstacle. Collisions with cells other than obstacle are taken into account by using check_triangle again.

- **move_cell** - This function implements the movement of a cell a certain distance. The function has inputs of a distance, dist, and a pointer to the cell that needs to be moved, this. The function simply adds dist*(cos(this->angle)/sin(this->angle)) to this->position and calls add_cell and remove_cell if the cell has left the domain of the agar structure it was in.

- **check_move** - This function implements a collision-checking algorithm to prevent cells from overrunning each other. The inputs for the algorithm are a pointer to the cell to be moved and the distance it should be moved. First, which agar contain cells that could interact with the path of the cell are determined. These are the agar that fall within the maximum cell length, 0.5, of the line between the end of the cell (x1, y1) and where it will end up if it moves the input distance (x2, y2). For simplicity the agar that intersect with the rectangle (min(xi) − 0.5, max(xi) + 0.5) × (min(yi) − 0.5, max(yi) + 0.5) are searched as shown in Figure 5.3. Then the line segments associated with the cell structures in those agar are checked to see if they intersect with the movement path. If they do, the movement distance is reduced to the distance at which the intersection occurs. Once all the agar structures in those agar have been checked, the function returns the distance to the first collision that will be encountered by the moving cell.

- **reverse** - This function simply implements the reversal of a cell. The x- and y-coordinates for the head are calculated, and position is set to these values. Then the function sets angle = angle + π/2, and increments clock by a binomially distributed random number.

Spatial Interaction Functions

- **fibril dens** - This function determines if the number of cells that can contribute useful fibril to a point in space is above a threshold. Given the point that the pili attached to in S.move, attachpt, and the point at the front of the moving cell, frontpt, it looks in all the agar within 2*INIT_LENGTH + FIBRIL_LENGTH, or the sum of the maximum length of a cell and the length of a fibril, of attachpt for other cells.
Figure 5.3. The agar are represented by the lattice of squares and the cell structures are represented by arrows. The yellow squares and circles illustrate which agar are searched during the check_move function.

For each cell in those agar, the function checks whether either of the endpoints of the cell, pt, meet two conditions. The first is if the endpoint is within FIBRIL_LENGTH of attachpt, and the second is if the angle formed by frontpt, attachpt, pt is less a cutoff, FIB ANG CUT. If so, it increments strength, which counts how many appropriate contributions have been found. After searching all of those agar, it checks if strength is greater than a threshold, FIB CUT. If so it returns 1; if not, 0.

- check_triangle - Given the moving cell, this, and an attachment point, pt, this function searches for where the first collision occurs as this moves towards pt. It searches for cells in the smallest rectangle that encloses the area that is within 2*INIT LENGTH of the triangle formed by the endpoints of this and pt. The cells in those agar are iterated through and assigned to that. The function first checks whether this and that intersect. To do this, the function checks if the rectangles that they are diagonals of overlap, and if so, whether they intersect. If the rectangles intersect, then that is not taken into account for collisions.

If they do not intersect, the function then classifies which of the four cases of cells that intersect the triangle that falls into. The four interesting cases are illustrated in Figure 5.4. The easiest case, labeled a in the Figure, is when a cell crosses only the segment from the tail of this to pt. Then the only point it needs to check is the end of that that is inside the triangle. If the angle formed by the ends of this and the interior endpoint of that is less than the current barrier, it replaces the value of the barrier. The case labeled
is simple as well, because the function merely has to check what angles correspond to the two endpoints of that.  is an interesting case, because none of the endpoints lie within the triangle. For this case the function looks at the angle that corresponds to the point where that intersects the segment from the head of this to pt. The case illustrated by c necessitates comparing the angles corresponding to the intersection point as for case b, and the interior endpoint of that with the current angle measurement. Once all possible that are checked, it returns what angle this can turn, and the distance it will move to place its head on the segment between the head and pt after it has turned.

Figure 5.4. The four possible situations of a cell that could cause a collision. In each case the function finds the minimum angle formed between the end points of the moving cell and the end point in the triangle or the intersection of the other cell with the segment between the head of the moving cell and the attachment, . In this situation, a causes the determining collision shown by the dashed line.

- **add_cell** - This function is called when initializing the simulation and by move_cell. Its only input is a cell, this. It simply adds a pointer to this in cells for the agar that this overlaps, increments num_here for that agar, and sets this->here as a pointer to that agar.

- **remove_cell** - This function is also called by move_cell. Using the input of a cell, this, it removes the pointer to this from cells and decrements num_here.

5.4 Preliminary Results

The simulation domain is a square domain with periodic boundary conditions. Simulations usually consist of placing an initial condition on the left-hand side of
the domain and allowing the system to propagate. This is often aided by defining a no-flux boundary that moves to the right as the simulation progresses.

In experiments with $A^-S^+$ cells, there is a distinct pattern of movement as can be seen in Figure 5.5. Cells form into an arrowhead-shaped group which then moves forward toward the pointed end. It was attempted to recreate this formation in our model. To do this, the model was initiated with a peninsula of cells that were pointed at a single point. This shape was not a stable one in the model. Eventually, the arrowhead rounds up into a circular body that remains stationary. An example of this initial condition and result is in Figure 5.6.

![Figure 5.5. Experimental pictures of $A^-S^+$ cells. Note the arrowhead shape that is maintained on the edge of the swarm.](image)

Observing experimental movies, it was noticeable that when an organized group from the interior of a colony encountered those cells on the edge of a colony, those cells on the edge were pushed outward. This state was simulated by beginning with two domains of cells. The first domain is a rectangular domain that extends the vertical from the top to the bottom of the domain. The cells in this domain are aligned vertically. The second domain is aligned horizontally and intermixes with the other domain. An example of this is in Figure 5.7. The edge of the colony moves...
Figure 5.6. Initial state for an arrowhead simulation is on the left. The eventual rounded profile is on the right.

some when the coaligned group collides with it. However, it does not maintain this status as shown in Figure 5.7.

Figure 5.7. Initial state for the two phase simulation is on the left. The eventual flattened profile is on the right.

5.5 Future Work

This model approximates the motion of $A^{-}S^{+}$ mutants of *M. xanthus*. However, due to the lack of area of an individual cell, too many cells can be fit into a space. This also prevents pushing events that should occur. Something similar to what
has been done in Staruss, et al.[77] may be appropriate, though a priori collision finding would be more exciting algorithmically. Also, the processes of division and growth of cells may play an important role in providing the pressure for movement.
In this paper, three models of myxobacteria were discussed. The first model was a modified Lattice Gas Cellular Automata that models swarming of a colony of wild-type cells as well as colonies of cells lacking a motility system. The second model was another modified LGCA that models the aggregation, sporulation and fruiting body formation of *Myxococcus xanthus*. The third model was an off-lattice stick model of S-motility in myxobacteria.

6.1 Lattice Gas Swarming Model

A modified Lattice Gas model for swarming in myxobacteria was developed. The alignment of cells on the lattice was determined by physical interactions of the rod shaped cells as well as motility specific alignment with slime trail following for A-motility and simplified pili dynamics for S-motility. The parameter space for this model was investigated to maximize the expansion rate. With the resulting parameters, the relationship between density and expansion rate was investigated as in experiment.

For both wild-type cells and $A^+S^-$ mutants, the rates of expansion predicted by the model are approximately 41% of the experimental values. For $A^-S^+$ mutants, the predicted expansion rate was only 21% of the experimental value. The relationship between initial density and the expansion rate was closest for wild-type cells.
For $A^+S^-$ mutants the dependence was an order of magnitude off, and for $A^-S^+$ mutants the dependence was of similar magnitude. For both mutants, the “zero density” expansion rate was comparable.

6.2 Lattice Gas Sporulation Model

An existing modified Lattice Gas model for aggregation and fruiting body formation was altered to include a reversal clock for each cell. The addition of reversals does not prevent aggregation or rounding up of the fruiting body, but the number of simulation steps necessary to reach these states increases. The reversal period was varied to explore the dependence of sporulation. Changing the initial density cells at the beginning of the simulation has little effect on the amount of sporulation. However, varying the rate at which cells enters the simulation domain results in maximum in the spore production near the natural reversal rate for myxobacteria.

6.3 Off-Lattice Swarming Model

An off-lattice stochastic model with extended cell representation was developed. This model models S-motility in $M. xanthus$. A couple initial conditions that come from experimental pictures were simulated, arrowhead shaped peninsulas and an organized body pushing against the edge. The results of these simulations were comparable to experimental pictures in the short term, but not for longer runs.

6.4 Contributions

The models in this dissertation continue to demonstrate that LGCA are effective models for cell-cell interactions. Several new interactions for cells were implemented as well as internal variables for the cells moving on the lattice. It was shown that reversals near the natural reversal period produce fruiting bodies with the maximum
of spores in the model. The first model showed that our understanding of S-motility was lacking, which the third model is attempting to address.
APPENDIX A

LGCA SWARMING MODEL

A.1 cell.h

#define CELL_THRESHOLD 0
#define MIN_LENGTH 7
#define GEN_TIME 210.0
#define MAX_REVERSAL ((int)(11.76*60/TIMESTEP))
#define MAX_REVERSALI (MAX_REVERSAL+1)
#define BINOM_P 0.5
#define PRESSURE 3
#define S_THRESHOLD 3
#define MAX_S 100
#define ALIGN_THRESH 10
#define SYN_DEFLECT 6.0
#define SYN_ALIGN 6.0
#define SYN_SLIME 8.0
#define SYN_DENS 4.0
#define SYN_OBS 20.0
#define BETA 2.0
#define STRAIGHT 100.0
#define NUTRIENT_CUT 100.0
#define SLIME_DRY_COEF 0.01
#define S_TWITCHING 0.99
#define REVERSE_PAUSE 0.5
#define S_A_RATIO 0.4
#include <gsl/gsl_rng.h>

/**************************************************************************
 * Definitions for the cells of the system.                              *
 *                                                                      *
 *-----------------------------------------------------------------------*
 *                                                                      *
 * struct cell
Holds the information for the cell:

- Position
- Which nodes are the cell
- Whether it is moving
- How much it is bent
- What direction the cell is pointed in

The length of the cell

Whether the cell is stalled or not

Whether the cell has turned yet or not

Routines

- `cell_init` -- Initializes the cell
- `cell_move` -- moves the cell
- `cell_turn` -- Turn the cell
- `cell_reverse` -- Reverse the direction of the cell
- `grow_cell` -- increase cell length by one
  and divide if necessary

```
struct cell
{
    struct medium *body[2*MIN_LENGTH]; //tail = site 0
    struct medium *center; //pointer to the node at the center
    unsigned short int turn_time;
    unsigned short int length:4;
    signed int bend:2; //the actual_bend = bend*60
    unsigned int direction:3; //The direction the cell is moving in.
    unsigned int moving:1; //Are we moving?
    unsigned int jammed:1;
    unsigned int turned:1;
}
```

```c
cell_init -- Initializes the cell

Parameters
    this -- which cell we want to initialize
    medium_ptr -- where to put the cell
    dir -- direction it’s pointed in.
    length -- length of the new cell.
    (0 for random length between 7-14)
    r -- random number generator
```
extern void cell_init(struct cell *this, struct medium *medium_ptr, int dir, int lngth, gsl_rng * r);

cell_move -- Moves the cell in the direction it’s pointed.
Parameters
    this -- which cell we want to move
    ctr -- total number of cells
    moves -- number of successful moves
    r -- random number generator

extern void cell_move(struct cell *this, int *ctr, int *moves, gsl_rng * r);

cell_turn -- Turn the cell
Parameters
    this -- which cell we want to turn.
    dir -- direction to turn to.

extern void cell_turn(struct cell *this, int dir);
cell_reverse -- reverse the direction of cell motion

Parameters

this -- the cell to reverse
dir -- the channel that the cell is in

extern void cell_reverse(struct cell *this, int dir);

grow_cell -- increase cell length by one and divide if necessary

Parameters

this -- which cell is growing
    spot -- the channel that the cell occupies
cntr -- how many cells are in the simulation
r -- random number generator

extern void grow_cell(struct cell *this, int spot, int *cntr,
                        gsl_rng * r);

A.2 cell.c

/*****************************************************************************/
 * Definitions for the cells of the system.                                 *
 *                                                                          *
 *---------------------------------------------------------------------------*
 *---------------------------------------------------------------------------*
 * Routines                                                                *
 *---------------------------------------------------------------------------*
 *                                                                          *
 *      cell_init -- Initializes the cell                                  *
 *      cell_move -- moves the cell                                         *
 *      cell_turn -- Turn the cell                                         *
 *      cell_reverse -- Reverse the direction of the cell                  *
 *      grow_cell -- increase cell length by one                           *
 *                  and divide if necessary                                *
 *---------------------------------------------------------------------------*/
#ifndef _CELL_INCLUDED_
#include "cell.h"
#define _CELL_INCLUDED_
#endif

#ifndef _MED_INCLUDED_
#include "medium.h"
#define _MED_INCLUDED_
#endif

#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_rng.h>
define min(a,b) ((a) <= (b) ? (a) : (b))

/****************************************************************************
extend_cell -- does the extension from one part of the cell body
to the next taking into account edges.
Parameters
	this -- what cell we are using
	here -- index of the body site we are connecting from
	there -- index of the body site we are connecting to
	dir -- what direction we are connecting in
****************************************************************************/
static void
extend_cell (struct cell *this, int here, int there, int dir)
{
    if (this->body[here]->neighbor[dir] == NULL)
    {   /*check if we can extend in that direction*/
        this->body[there] = this->body[here];
    } else
    {
        this->body[there] =
        this->body[here]->neighbor[dir];
    }
}

/****************************************************************************
cell_init -- Initializes the cell
Parameters
	this -- which cell we want to initialize
	medium_ptr -- where to put the cell
	dir -- direction it’s pointed in.
	r -- random number generator

78
/*********************************************/
void
cell_init (struct cell *this, struct medium *medium_ptr, int dir,
         int lgth, gsl_rng * r)
{
    // create the cell
    int i;
    this->center = medium_ptr;
    this->direction = dir % 6;
    this->moving = 1;
    this->jammed = 0;
    this->bend = 0;
    if (lgth == 0) // want a random length
    {
        this->length =
gsl_rng_uniform_int (r, MIN_LENGTH)
            + MIN_LENGTH;
        this->turn_time =
gsl_rng_uniform_int (r,
            (int) (MAX_REVERSAL * BINOM_P));
    }
    else
    {
        this->length = lgth;
    }
    this->body[this->length / 2] = this->center;
    (this->center->numhere)++;
    for (i = this->length / 2; i > 0; i--)
    {
        extend_cell (this, i, i - 1,
            (this->direction + 3) % 6);
    }
    for (i = this->length / 2; i < this->length - 1; i++)
    {
        extend_cell (this, i, i + 1, this->direction);
    }
    for (i = 0; i < this->length; i++)
    {
        (this->body[i]->numcells[this->direction % 3])++;
    }
}

/*********************************************/
getnumcells - the number of cells overlapping in the site in
direction from here
Parameters
getnumcells (struct medium *here, int dir)
{
    int ret, i;
    if (here->neighbor[dir] != NULL)
    {
        ret = 0;
        for (i = 0; i < 3; i++)
        {
            ret += here->neighbor[dir]->oldnumcells[i];
        }
        return (ret);
    }
    else
    {
        return (-1);
    }
}

cell_move -- Moves the cell in the direction it’s pointed.
Parameters
    this -- which cell we want to move
    ctr -- total number of cells
    moves -- total number of moves
    r -- random number generator
cell_move (struct cell *this, int *ctr, int *moves, gsl_rng * r)
{  //move the cell
    int i;
    struct medium *here;
    here = this->center;
    if (here->neighbor[this->direction] != NULL)
    {
        if (here->neighbor[this->direction]->
            resident[(this->direction + 6)]
            == NULL && (getnumcells(this->body[this->length - 1],this->direction)
                <= CELL_THRESHOLD
            || (here->oldnumcells[0] + here->oldnumcells[1]

80
+ here->oldnumcells[2]) * gsl_rng_uniform (r)
  >= getnumcells(this->body[this->length - 1],
               this->direction))
{
  (*moves)++;
  for (i = 0; i < this->length / 2; i++)
  {
    this->body[i]->numcells[
                      (this->direction - this->bend + 3) % 3]--;
  }
  for (i = this->length / 2;
       i < this->length; i++)
  {
    this->body[i]->
       numcells[this->direction % 3]--;  
  }
  (here->numhere)--;
  here->resident[this->direction] = NULL;
  this->center =
      here->neighbor[this->direction];
  (this->center->numhere)++;
  this->center->resident[this->direction + 6]
      = this;
  this->body[this->length / 2] =this->center;
  for (i = this->length / 2; i > 0; i--)
  {
    extend_cell (this, i, i - 1, 
             (this->direction + 3) % 6);
  }
  for (i = this->length / 2;
     i < this->length - 1; i++)
  {
    extend_cell (this, i, i + 1, 
                  this->direction);
  }
  this->bend = 0;
  this->moving = 0;
  for (i = 0; i < this->length; i++)
  {
    (this->body[i]->
       numcells[this->direction % 3])++;
  }
}
else
{
    this->moving = 0;
    this->jammed = 1;
    if (gsl_rng_uniform(r) < REVERSE_PAUSE)
    {
        (this->turn_time)++;
    }
    if (here->resident[this->direction + 6] ==
        NULL)
    {
        here->resident[this->direction + 6] = this;
        here->resident[this->direction] = NULL;
    }

}
else
{
    this->jammed = 1;
    this->moving = 0;
}
}

/***********************************************************/
cell_turn -- Turn the cell
Parameters
    this -- which cell we want to turn.
    dir -- direction to turn to.
/***********************************************************/
void
cell_turn (struct cell *this, int dir)
{ //reorient the cell
    int i;
    struct medium *here;
    here = this->center;
    for (i = this->length / 2; i < this->length; i++)
    {
        (this->body[i]->numcells[this->direction % 3])--;
    }
    if (dir < 6)
    {
        here->resident[this->direction] = NULL;
else
{
here->resident[this->direction + 6] = NULL;
}
this->direction = dir % 6;
here->resident[dir] = this;
for (i = this->length / 2; i < this->length - 1; i++)
{
extend_cell (this, i, i + 1, this->direction);
}
for (i = this->length / 2; i < this->length; i++)
{
(this->body[i]->numcells[this->direction % 3])++;
}

/***********************************************************/
cell_reverse -- Turn the cell
Parameters
   this -- which cell we want to turn.
/**************************************************************/
void
cell_reverse (struct cell *this, int dir)
{ //reverse the cell
int i;
for (i = 0; i < this->length / 2; i++)
{
(this->body[i]->numcells[(this->direction -
    this->bend + 3) % 3])--;
}
for (i = this->length / 2; i < this->length; i++)
{
(this->body[i]->numcells[(this->direction) % 3])--;
}
this->center->resident[dir] = NULL;
this->direction = (this->direction + 3) % 6;
this->center->resident[this->direction + 6] = this;
this->moving = 0;
this->bend = 0;
this->body[this->length / 2] = this->center;
for (i = this->length / 2; i < this->length - 1; i++)
{
extend_cell (this, i, i + 1, this->direction);
}
for (i = this->length / 2; i > 0; i--)
{
extend_cell (this, i, i - 1,
(this->direction + 3) % 6);
}
for (i = 0; i < this->length; i++)
{
(this->body[i]->numcells[this->direction % 3])++;
}

/***************************************************************/
grow_cell -- increase cell length by one and divide if necessary
Parameters
this -- which cell is growing
spot -- channel that the cell resides in
cntr -- how many cells are in the simulation
r -- random number generator
***************************************************************************/
void
grow_cell (struct cell *this, int spot, int *cntr, gsl_rng * r)
{
int i;
int tries;
if (this->length < 2 * MIN_LENGTH)
{
if (this->length % 2 == 0)
{
extend_cell (this, this->length - 1,
    this->length, this->direction);
this->body[this->length]->
    numcells[this->direction % 3]++;
this->length++;
}
else
{
for (i = this->length; i > 0; i--)
{
this->body[i] = this->body[i - 1];
}
extend_cell (this, 1, 0,
    (this->direction - this->bend + 3) % 6);
this->body[0]->numcells[(this->direction
    - this->bend + 3) % 3]++;
this->length++;
if (this->length == 2 * MIN_LENGTH)
{
    if (this->body[MIN_LENGTH / 2]->numhere < PRESSURE
        && this->body[MIN_LENGTH + MIN_LENGTH / 2]->
        numhere < PRESSURE)
    {
        for (i = 0; i < MIN_LENGTH; i++)
        {
            this->body[i]->
            numcells[(this->direction - this->bend + 3) % 3]--;
        }
        if (this->body[MIN_LENGTH / 2]->
            resident[(this->direction -
                this->bend + 6) % 6] == NULL)
        {
            this->body[MIN_LENGTH / 2]->
            resident[(this->direction -
                this->bend + 6) % 6] =
            malloc (sizeof (struct cell));
            cell_init (this->
                body[MIN_LENGTH / 2]->
                resident[(this->direction -
                    this->bend + 6) % 6],
            this->body[MIN_LENGTH / 2],
            (this->direction - this->bend + 6) % 6, MIN_LENGTH, r);
            this->body[MIN_LENGTH / 2]->
            resident[(this->direction - this->bend + 6) % 6]->turn_time
            = this->turn_time;
            (*cntr)++;
        }
        else if (this->body[MIN_LENGTH / 2]->
            resident[(this->direction - this->bend + 6) % 12] == NULL)
        {
            this->body[MIN_LENGTH / 2]->
            resident[(this->direction - this->bend + 6) % 12]
            = malloc (sizeof (struct cell));
            cell_init (this->
                body[MIN_LENGTH / 2]->
                resident[(this->direction - this->bend + 6) % 12],
            this->body[MIN_LENGTH / 2], (this->direction - this->bend + 6) % 12, MIN_LENGTH, r);
            this->body[MIN_LENGTH / 2]->
resident[(this->direction - this->bend + 6) % 12]->turn_time = this->turn_time;
(*cntr)++;
}
else if (this->body[MIN_LENGTH / 2]->
        resident[(this->direction - this->bend + 3) % 6] == NULL)
{
    this->body[MIN_LENGTH / 2]->
    resident[(this->direction - this->bend + 3) % 6] = malloc (sizeof (struct cell));
cell_init (this->body[MIN_LENGTH / 2]->resident[(this->direction - this->bend + 3) % 6],
            this->body[MIN_LENGTH / 2], (this->direction - this->bend + 3) % 6,
            MIN_LENGTH, r);
    this->body[MIN_LENGTH / 2]->
    resident[(this->direction - this->bend + 3) % 6]->turn_time = this->turn_time;
    (*cntr)++;
}
else if (this->body[MIN_LENGTH / 2]->
         resident[(this->direction - this->bend + 9) % 12] == NULL)
{
    this->body[MIN_LENGTH / 2]->
    resident[(this->direction - this->bend + 9) % 12] = malloc (sizeof (struct cell));
cell_init (this->body[MIN_LENGTH / 2]->resident[(this->direction - this->bend + 9) % 12],
            this->body[MIN_LENGTH / 2], (this->direction - this->bend + 9) % 12,
            MIN_LENGTH, r);
    this->body[MIN_LENGTH / 2]->
    resident[(this->direction - this->bend + 9) % 12]->turn_time = this->turn_time;
    (*cntr)++;
}
else
{
    tries = 0;
    while (tries < 20)
    {
        i =
gsl_rng_uniform_int (r, 12);
        if (this->body[MIN_LENGTH / 2]->resident[i] == NULL)
this->body[MIN_LENGTH / 2]->resident[i] = malloc (sizeof(struct cell));
cell_init (this->body[MIN_LENGTH / 2]->resident[i],
          this->body[MIN_LENGTH / 2], i, MIN_LENGTH, r);
this->body[MIN_LENGTH / 2]->resident[i]->turn_time = this->turn_time;
(*cntr)++;
break;
} else {
    tries++;
}
if (tries == 20) {
    printf ("New cell died due to overcrowding\n");
}
for (i = 0; i < MIN_LENGTH; i++) {
    this->body[i] = this->body[i + MIN_LENGTH];
    this->body[i + MIN_LENGTH] = NULL;
}
this->length = MIN_LENGTH;
this->center->resident[spot] = NULL;
this->center->numhere--;
this->center = this->body[MIN_LENGTH / 2];
this->bend = 0;
if (this->center->resident[this->direction % 6] == NULL) {
    this->center->resident[(this->direction) % 6] = this;
    this->center->numhere++;
} else if (this->center->
    resident[(this->direction + 6) % 12] == NULL) {
    this->center->resident[(this->direction + 6) % 12] = this;
    this->center->numhere++;
} else if (this->center->
    resident[(this->direction + 3) % 6] == NULL) {
    this->center->resident[(this->direction + 3) % 6] = this;
    this->direction = (this->direction + 3) % 6;
    this->center->numhere++;
} else if (this->center->
    resident[(this->direction + 9) % 12] == NULL)
{
    this->center->resident[(this->direction + 9) % 12] = this;
    this->direction = (this->direction + 9) % 6;
    this->center->numhere++;
}
else
{
    for (i = 0; i < MIN_LENGTH; i++)
    {
        this->body[i]->numcells[this->direction % 3]--;
    }
    tries = 0;
    while (tries < 20)
    {
        i = gsl_rng_uniform_int (r, 12);
        if (this->center->resident[i] == NULL)
        {
            this->center->resident[i] = this;
            this->direction = i % 6;
            for (i = MIN_LENGTH / 2; i > 0; i--)
            {
                extend_cell (this, i, i - 1, (this->direction + 3) % 6);
                extend_cell (this, MIN_LENGTH - 1 - i, MIN_LENGTH - i,
                this->direction);
            }
            this->center->numhere++;
            for (i = 0; i < MIN_LENGTH; i++)
            {
                this->body[i]->numcells[this->direction % 3]++;
            }
            break;
        }
    }
else
{
    tries++;
}
}
if (tries == 20)
{
    printf ("Old cell died due to overcrowding\n");
    free (this);
    (*cntr)--;
}
A.3 medium.h

#define TURN_LIMIT 100
#define SLIME_COEF_PROD 1
#define TIMESTEP (343.0/(2*MIN_LENGTH-(MIN_LENGTH/2+1)))
#define SPACING (6.0/(2*MIN_LENGTH-(MIN_LENGTH/2+1)))
#define DIFFUSION_RATE 500.0
#define NUTRIENT_INIT 2000000.0
#include <gsl/gsl_rng.h>

This describes the lattice that the cells are on.
Can be used to create any connectivity you desire,
but you can have at most 6 neighbors.

struct medium
  what the cells sit on. Holds information of what cells
  are centered here, how many cells overlap at this site,
  the amount of slime present in the 3 orientations and
  where the cells can go from here.

Routines
  medium_init -- initialize the site.
  medium_addneighbor -- connect site to a neighbor.
  turn_stalled_cells -- turns the stalled cells at the site.
  set_numcells -- transfers newcells to oldcells.
  move_cells -- move all cells at site.
  A_align_cells -- Monte Carlo alignment step to model
  A-motility.
  S_align_cells -- Monte Carlo alignment step to model
  S-motility.
  Syn_align_cells -- Monte Carlo alignment step to model
  A- and S-motility.
  moved -- check stationary sites.
  reversal -- reverse cells in need of reversing.
  slime_it -- deposits slime at the tail of the cells at this
  site.
  diffuse_nutrient -- diffuses nutrient.
  set_nutrient -- moves nutrient up a timestep.
  growth -- cells absorb nutrient, grow and divide.
struct medium {
    double nutrient; //The amount of nutrient at the site.
    double new_nutrient; // current nutrient amount
    double old_nutrient; //nutrient amount from the last timestep
    struct medium *neighbor[6]; //nearest neighbors to this site.
    struct cell *resident[12]; //Cells that are at this site.
    //More than 6 because we may have stationary cells.
    struct medium *next;
    //The next structure. Used for looping through sites.
    unsigned short int numcells[3];
    //How many cells overlap on this site.
    unsigned short int oldnumcells[3];
    unsigned short int slime[3]; //How much slime is present.
    unsigned short int numhere;
    //How many cells are centered at this site.
};

medium_init -- Initialize the site
Parameters
    here -- pointer to the site to initialize
    x,y,z -- indices of this site
**********************************************************************
extern void medium_init(struct medium *here, int x, int y);

medium_addneighbor -- Connect site to neighbor
Parameters
    here -- pointer to the site we’re at
    neighbor -- pointer to the site to connect
    dir -- direction from here to neighbor
**********************************************************************
extern void addneighbor(struct medium *here, int dir,
                        struct medium *neighbor);

set_numcells -- moves oldnumcells to numcells
Parameters
    here -- the site we are at
**********************************************************************
extern void set_numcells(struct medium *here);
move_cells -- move cells at the site taking into account climbing

Parameters
    here -- the site we are at.
    ctr -- total number of cells.
    tries -- number of attempted moves.
    moves -- number of successful moves.
    r -- random number generator.

extern void move_cells(struct medium *here, int *ctr, int *tries,
                      int *moves, gsl_rng * r);

align_cells --
change direction of motion according to surrounding area

Parameters
    here -- the site we are at
    r -- random number generator
    mutant_switch -- what type of mutant we are simulating

extern void align_cells(struct medium *here, gsl_rng * r,
                        int mutant_switch);

moved -- change stationary cells to moving cells if they have space

Parameters
    here -- site we are at.

extern void moved(struct medium *here);

reversal -- do updating of reversing direction in cells

Parameters
    this -- site we are at.
    time -- the time step we are on.
    distrib -- reversal distribution.
    r -- random number generator

extern void reversal(struct medium *this, int time,
                     double *distrib, gsl_rng * r);

slimeit -- update the slime level at a site

Parameters
    this -- the site we are at.
    prod -- the amount of slime to produce
extern void slimeit(struct medium *here, int prod);

/**
 * diffuse_nutrient -- do diffusion of the nutrient
 * Parameters
 * here -- the site we are at.
 */
extern void diffuse_nutrient(struct medium *here);

/**
 * set_nutrient -- moves new_nutrient to nutrient
 * Parameters
 * here -- the site we are at.
 */
extern void set_nutrient(struct medium *here, gsl_rng * r);

/**
 * growth -- cells absorb nutrient, grow and divide.
 * Parameters
 * here -- the site we are at.
 * ctr -- total number of cells.
 * r -- random number generator
 */
extern void growth(struct medium *here, int *ctr, gsl_rng * r);

A.4 medium.c

/**
 * This describes the lattice that the cells are on. Can be used to
 * create any connectivity you desire, but you can have at most 18
 * neighbors.
 */

struct medium

what the cells sit on. Holds information of what cells are
centered here and how many cells overlap at this site and
where the cells can go from here.

Routines
medium_init -- initialize the site.
medium_addneighbor -- connect site to a neighbor.
set_numcells -- transfers newcells to oldcells.
get_obstacles -- what the cell will run into if it turns.
move_cells -- move all cells at site.
align_cells -- Monte Carlo alignment step to model all motility.
moved -- check stationary sites.
reversal -- reverse cells in need of reversing.
slime it -- deposits slime at the tail of the cells at this site.
diffuse nutrient -- diffuses nutrient.
set nutrient -- moves nutrient up a timestep.
growth -- cells absorb nutrient, grow and divide.

#define min(a,b) ((a) <= (b) ? (a) : (b))

medium_init -- Initialize the site
Parameters
  here -- pointer to the site to initialize
  x,y,z -- indices of this site

void
medium_init (struct medium *here, int x, int y)
{
  //Create the site
  int i;
  for (i = 0; i < 6; i++)
  {
    here->neighbor[i] = NULL;
    here->resident[2 * i] = NULL;
    here->resident[2 * i + 1] = NULL;
    here->slime[i % 3] = 0;
    here->numcells[i % 3] = 0;
    here->oldnumcells[i % 3] = 0;
  } 
  here->numhere = 0;
  here->nutrient = NUTRIENT_INIT;
  here->new_nutrient = 0.0;
  here->old_nutrient = NUTRIENT_INIT;
}
/**************************************************************
medium_addneighbor -- Connect site to neighbor
Parameters
  here -- pointer to the site we’re at
  neighbor -- pointer to the site to connect
  dir -- direction from here to neighbor
***************************************************************/
void
addneighbor (struct medium *here, int dir, struct medium *nbr)
{
  //Do connectivity
  here->neighbor[dir] = nbr;
  nbr->neighbor[(dir + 3) % 6] = here;
}

/**************************************************************
get_cells_nbhd -- Collect how many cells are within 8 away from
the end of the cell.
Parameters
  here -- which site we are at.
  dir -- which direction to go
***************************************************************/
int
get_cells_nbhd (struct medium *here, int dir)
{
  int i, j, k;
  int ret;
  struct medium *axis, *temp;
  axis = here;
  ret = 0;
  for (i = 0; i < (int) ((float)MIN_LENGTH * 1.5); i++)
    {
      temp = axis;
      for (j = 0; j < i; j++)
        {
          if (temp->neighbor[(dir + 5) % 6] != NULL)
            {
              temp = temp->neighbor[(dir + 5) % 6];
              for (k = 0; k < 3; k++)
                {
                  ret += temp->oldnumcells[k];
                }
            }
          temp = axis;
    }
for (j = 0; j < i; j++)
{
    if (temp->neighbor[(dir + 1) % 6] != NULL)
    {
        temp = temp->neighbor[(dir + 1) % 6];
        for (k = 0; k < 3; k++)
        {
            ret += temp->oldnumcells[k];
        }
    }
}
if (axis->neighbor[dir] != NULL)
{
    axis = axis->neighbor[dir];
    for (k = 0; k < 3; k++)
    {
        ret += axis->oldnumcells[k];
    }
} else
{
    return (ret);
}
for (i = (int) ((float)MIN_LENGTH * 1.5); i > 0; i--)
{
    temp = axis;
    for (j = 0; j < i; j++)
    {
        if (temp->neighbor[(dir + 5) % 6] != NULL)
        {
            temp = temp->neighbor[(dir + 5) % 6];
            for (k = 0; k < 3; k++)
            {
                ret += temp->oldnumcells[k];
            }
        }
    }
    temp = axis;
    for (j = 0; j < i; j++)
    {
        if (temp->neighbor[(dir + 1) % 6] != NULL)
        {
            temp = temp->neighbor[(dir + 1) % 6];
            for (k = 0; k < 3; k++)
            {
                ret += temp->oldnumcells[k];
            }
        }
    }
}
{ ret += temp->oldnumcells[k]; }
}
}

if (axis->neighbor[dir] != NULL)
{
 axis = axis->neighbor[dir];
 for (k = 0; k < 3; k++)
 { ret += axis->oldnumcells[k]; }
}
else
{
 return (ret);
}
}
return (ret);

*******************************************************************************
get_cell_align --
 Find how many cells the cell will align with if it turns in the
direction
Parameters
here -- the center of the cell
dir -- the direction it might want to turn
dist -- how far to go out
*******************************************************************************

int get_cell_align (struct medium *here, int dir, int dist)
{
 int i, j;
 int ret;
 struct medium *axis, *temp;
 axis = here;
 ret = 0;
 for (i = 0; i < dist; i++)
 { temp = axis;
 for (j = 0; j < 1; j++)
 { if (temp->neighbor[(dir + 5) % 6] != NULL)
 {
temp = temp->neighbor[(dir + 5) % 6];
ret += temp->oldnumcells[dir % 3];
}
}
temp = axis;
for (j = 0; j < 1; j++)
{
if (temp->neighbor[(dir + 1) % 6] != NULL)
{
temp = temp->neighbor[(dir + 1) % 6];
ret += temp->oldnumcells[dir % 3];
}
}
if (axis->neighbor[dir] != NULL)
{
axis = axis->neighbor[dir];
}
else
{
return (ret);
}
}
for (i = 0; i < 1; i++)
{
if (axis->neighbor[dir] != NULL)
{
temp = axis->neighbor[dir];
ret += temp->oldnumcells[dir % 3];
}
return (ret);
}

/***************************************************************************/
get_deflections --
    Find how many cells are pushing the cell to turn at the tip.
Parameters
here -- the center of the cell
dir -- the direction it might want to turn
***************************************************************************/
int
get_deflections (struct cell *this, int dir)
{
if (this->body[this->length - 1]->neighbor[this->direction]
   != NULL)
return (this->body[this->length - 1]->
neighbor[this->direction]->oldnumcells[dir]);
} else {
return (0);
} }

/***************************************************************************/
gen_obstacles -- Find how many cells the cell would cross to turn.
Parameters
here -- the center of the cell
array -- array to put weights into
*****************************************************************************/
void gen_obstacles(struct cell *this, int *array) {
struct medium *here,*there;
int i,j,l;
for(i=0;i<3;i++) {
array[i]=0;
}
for(i=this->length/2+1;i<this->length;i++) {
here=this->body[i];
there=this->body[i];
for(j=this->length-1-i;j>0;j--) {
if(here->neighbor[(this->direction+5)%6]!=NULL) {
here=here->neighbor[(this->direction+5)%6];
for(l=0;l<3;l++) {
array[2]+=here->oldnumcells[l];
}
}
if(there->neighbor[(this->direction+1)%6]!=NULL) {
there=there->neighbor[(this->direction+1)%6];
for(l=0;l<3;l++) {
array[1]+=there->oldnumcells[l];
}
move_cells -- move cells at the site taking into account edges
Parameters
  here -- the site we are at.
*************************************************************************/
void
move_cells (struct medium *here, int *ctr, int *tries, int *moves,
         gsl_rng * r)
{
  int dir;
  struct cell *this;
  for (dir = 0; dir < 6; dir++)
  {
    if (here->resident[dir] != NULL)
    {
      this = here->resident[dir];
      this->turned = 0;
      if (this->moving != 0)
      {
        (*tries)++;
        this->jammed = 0;
        cell_move (this, ctr, moves, r);
      }
    }
  }
}

align_cells -- change direction of motion according to nearby cells
Parameters
  here -- the site we are at
  r -- random number generator
  mutant_switch -- what type of cells we have
*************************************************************************/
void
align_cells (struct medium *here, gsl_rng * r, int mutant_switch)
{
  int i, j;
  double Z;
  double temp;
int total_slime;
double cells[3];
double align[3];
double deflections[3];
int obstacles[3];
double cell_weight[3];
int startdir;
startdir = gsl_rng_uniform_int (r, 6);
struct cell *this;
total_slime = 0;
for (i = -1; i < 2; i++)
{
    total_slime += here->slime[(i + 3) % 3];
}
for (i = 0; i < 6; i++)
{
    if (here->resident[((startdir + i) % 6) != NULL])
    {
        this = here->resident[[(startdir + i) % 6];
        if (this->turned == 0)
        {
            get_obstacles(this, obstacles);
            for (j = -1; j < 2; j++)
            {
                if (mutant_switch == 0 || mutant_switch == 1)
                {
                    cells[(j + 3) % 3] = SYN_DENS *
                    get_cells_nbhd (this->body[this->length - 1],
                                    (startdir + i + j + 6) % 6);
                }
                else
                {
                    cells[(j + 3) % 3] = 0;
                }
            }
            deflections[(j + 3) % 3] = SYN_DEFLECT *
            get_deflections (this, (startdir + i + j + 6) % 3);
            align[(j + 3) % 3] = SYN_ALIGN *
            get_cell_align (here, (startdir + i + j + 6) % 6, this->length / 2);
            cell_weight[(j + 3) % 3] =
            exp ((align[(j + 3) % 3] + deflections[(j + 3) % 3]
                + cells[(j + 3) % 3]
                + SYN_SLIME * here->slime[(startdir + i + j + 6) % 3]
                - SYN_OBS*obstacles[(j + 3) % 3]) / BETA);
        }
    }
}
Z = cell_weight[0] + cell_weight[1] + cell_weight[2];
    || cells[2] >= S_THRESHOLD || deflections[1]!=0
    || deflections[2]!=0
    || total_slime - here->slime[(startdir+i)%3] > 0)
{
    this->turned = 1;
    temp = gsl_rng_uniform (r);
    if (temp < cell_weight[2] / Z)
    {
        if (here-> resident[(startdir + i + 5) % 6] == NULL
            && this->bend != -1)
        {
            cell_turn (this, (startdir + i + 5) % 6);
            (this->bend)--;
        }
    }
    else if (temp < (cell_weight[2] + cell_weight[0]) / Z)
    {
        continue;
    }
    else
    {
        if (here->resident[(startdir + i + 1) % 6] == NULL
            && this->bend != 1)
        {
            cell_turn (this, (startdir + i + 1) % 6);
            (this->bend)++;
        }
    }
}

if (mutant_switch == 0)
{
    temp = S_A_RATIO*S_TWITCHING;
}
else if (mutant_switch == 1)
{
    temp = S_TWITCHING;
}
else
{
    temp = 0.0;
}
if (gsl_rng_uniform (r) < temp)
{  
if (get_cells_nbhd(this->body[this->length - 1], this->direction)  
< S_THRESHOLD)
{
this->moving = 0;
}
}

void
set_numcells (struct medium *here)
{
  int j;
  for (j = 0; j < 3; j++)
  {
    here->oldnumcells[j] = here->numcells[j];
  }
}

/*******************************
moved -- change stationary cells to moving cells if they have space
Parameters
    here -- site we are at.
*******************************
void
moved (struct medium *here)
{
  int i;
  for (i = 6; i < 12; i++)
  {
    if (here->resident[i] != NULL)
    {
      if (here->resident[i - 6] == NULL)
      {
        here->resident[i - 6] = here->resident[i];
        here->resident[i] = NULL;
      }
    }
  }
  for (i = 0; i < 6; i++)
  {

if (here->resident[i] != NULL)
{
    here->resident[i]->moving = 1;
}
}
}

/******************************************************************************
timejump - randomly generate a time jump based on a distribution
Parameters
distrib - array of PMF
*******************************************************************************/
int
timejump (double *distrib, gsl_rng * r)
{
    int i = 0;
    double rv, cprob = 0.0;
    rv = gsl_rng_uniform (r);
    while (cprob < 1.0)
    {
        cprob = cprob + *(distrib + i);
        if (rv < cprob)
        {
            return (i);
        }
        else
        {
            i += 1;
        }
    }
    printf ("Time jump didn’t work right");
    return (0);
}

/******************************************************************************
reversal - reverse those cells that need to be reversed
Parameters
      here -- site we are at.
      time -- the time step we are on.
*******************************************************************************/
void
reversal (struct medium *here, int time, double *distrib,
          gsl_rng * r)
{
    int i;
struct cell *this;
for (i = 6; i < 12; i++)
{
    if (here->resident[i] != NULL)
    {
this = here->resident[i];
if (this->turn_time <= time)
    {
if (here->resident[(i + 3) % 6 + 6] == NULL)
    {
this->turn_time = time + timejump (distrib, r);
cell_reverse (this, i);
}
else
    {
this->moving = 0;
this->jammed = 0;
}
}
}

for (i = 0; i < 6; i++)
{
    if (here->resident[i] != NULL)
    {
this = here->resident[i];
if (this->turn_time <= time)
    {
if (here->resident[(i + 3) % 6 + 6] == NULL)
    {
this->turn_time = time + timejump (distrib, r);
cell_reverse (this, i);
}
else if (here->resident[i + 6] == NULL)
    {
here->resident[i + 6] = this;
here->resident[i] = NULL;
this->moving = 0;
this->jammed = 0;
}
else
    {
this->moving = 0;
}
}
}
/*********************************************
slimeit -- update the slime level at a site
Parameters
this -- the site we are at.
prod -- how much to produce.
**********************************************/
void
slimeit (struct medium *here, int prod)
{
  int i;
  for (i = 0; i < 12; i++)
  {
    if (here->resident[i] != NULL)
    {
      here->resident[i]->body[0]->
      slime[(i - here->resident[i]->bend + 3) % 3] += prod;
    }
  }
}

/******************************************************/

diffuse_nutrient -- do diffusion of the nutrient
Parameters
here -- the site we are at.
***************************************************/
void
diffuse_nutrient (struct medium *here)
{
  int i;
  int method;
  float nbr_nutrient;
  method = 0;
  for (i = 0; i < 6; i++)
  {
    if (here->neighbor[i] == NULL)
    {
      method += pow (2, i);
    }
  }
  switch (method)
  {
  }
case 0:
nbr_nutrient = 0;
for (i = 0; i < 6; i++)
{
    nbr_nutrient += here->neighbor[i]->nutrient;
}
here->new_nutrient =
((DIFFUSION_RATE * 4.0 / 3.0 * TIMESTEP / SPACING / SPACING * (nbr_nutrient - 3 * here->old_nutrient)) +
  here->old_nutrient)
  / (1 + (DIFFUSION_RATE * 4.0 * TIMESTEP / SPACING / SPACING));
break;

case 55:
    //Same as 1
case 35:
    //Same as 1
case 1:
here->new_nutrient = here->neighbor[3]->nutrient;
break;

case 47:
    //same as 2
case 7:
    //Same as 2
case 2:
here->new_nutrient = here->neighbor[4]->nutrient;
break;

case 31:
    //Same as 4
case 14:
    //Same as 4
case 4:
here->new_nutrient = here->neighbor[5]->nutrient;
break;

case 62:
    //same as 8
case 28:
    //Same as 8
case 8:
here->new_nutrient = here->neighbor[0]->nutrient;
break;
case 61:
    //same as 16
    case 56:
        //Same as 16
        case 16:
            here->new_nutrient = here->neighbor[1]->nutrient;
            break;

    case 59:
        //Same as 32
        case 49:
            //same as 32
            case 32:
                here->new_nutrient = here->neighbor[2]->nutrient;
                break;

    case 39:
        //Same as 3
        case 3:
            here->new_nutrient =
                (here->neighbor[3]->nutrient +
                here->neighbor[4]->nutrient) / 2.0;
            break;

    case 15:
        //Same as 6
        case 6:
            here->new_nutrient =
                (here->neighbor[4]->nutrient +
                here->neighbor[5]->nutrient) / 2.0;
            break;

    case 30:
        //Same as 12
        case 12:
            here->new_nutrient =
                (here->neighbor[5]->nutrient +
                here->neighbor[0]->nutrient) / 2.0;
            break;

    case 60:
        //Same as 24
        case 24:
            here->new_nutrient =
                (here->neighbor[0]->nutrient +
                here->neighbor[1]->nutrient +
                here->neighbor[2]->nutrient +
                here->neighbor[3]->nutrient +
                here->neighbor[4]->nutrient +
                here->neighbor[5]->nutrient) / 6.0;
            break;
here->neighbor[1]->nutrient) / 2.0;
break;

case 57:
    //Same as 48
    case 48:
        here->new_nutrient =
        (here->neighbor[1]->nutrient +
        here->neighbor[2]->nutrient) / 2.0;
        break;

case 51:
    //Same as 33
    case 33:
        here->new_nutrient =
        (here->neighbor[2]->nutrient +
        here->neighbor[3]->nutrient) / 2.0;
        break;

default:
    printf("Diffusion broke. Are you sure the lattice is right?\n");
    break;
}
}

/****************************************************
set_nutrient -- moves new_nutrient to nutrient and dries slime
Parameters
here -- the site we are at.
****************************************************/
void
set_nutrient (struct medium *here, gsl_rng * r)
{
    int i;
    here->old_nutrient = here->nutrient;
    here->nutrient = here->new_nutrient;
    for (i = 0; i < 3; i++)
    {
        here->slime[i] -=
        gsl_ran_binomial(r, SLIME_DRY_COEF, here->slime[i]);
    }
}

/****************************************************
growth -- divides cells if their clock is up.

Parameters
here -- the site we are at.
*******************************************************************************/

void
growth (struct medium *here, int *ctr, gsl_rng * r)
{
  int i, j;
  int eat;
  struct cell *this;
  for (i = 0; i < 12; i++)
  {
    if (here->resident[i] != NULL)
    {
      this = here->resident[i];
      eat = 0;
      for (j = 0; j < this->length; j++)
      {
        if (this->body[j]->nutrient > NUTRIENT_CUT)
        {
          this->body[j]->nutrient = this->body[j]->nutrient - 1.0;
          eat++;
        }
      }
    }
  }
  if (eat > .5 * this->length || this->length == 2 * MIN_LENGTH)
  {
    if (gsl_rng_uniform (r) <
        1.0 / (GEN_TIME * 60.0 / TIMESTEP / MIN_LENGTH) 
          || this->length == 2 * MIN_LENGTH)
    {
      grow_cell (here->resident[i], i, ctr, r);
    }
  }
}

A.5 write_gif.h
#include <stdio.h>
extern void write_gif(FILE *of, int matrix[], int dimx, int dimy,
                      int image_width, int image_height);

A.6 write_gif.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MY_BITS (int) 12
#define HSIZE (int) 5003    /* 80% occupancy */
#define MAXCODE(n_bits) (((code_int) 1 << (n_bits)) - 1)

typedef int code_int;
typedef long int count_int;
typedef unsigned char char_type;

FILE *gfp;

static int have_c_map = 0;
static int colormap_size,bits_pixel;
static char *Red,*Green,*Blue;
char *red,*green,*blue;

int GetPixel (void);
long count_down;
int width,height,mwidth,mheight,x,y;
int *image;

int hash[256],shah[256];

void GIFEncode(int Background,int BitsPerPixel,
                int (* getpixel)(void));

void Putword (int word,FILE *fp);
void compress (int init_bits,int (*getpixel)(void));

void
write_gif(FILE *of, int matrix[],int dimx,int dimy,
          int image_width, int image_height)
{
    int nobjects;
    long index;
    int i,dummy;
   
    gfp=of;

    width = image_width; height= image_height;


mwidth = dimx; mheight = dimy;
x = 0; y = 0;

count_down = (long) width * (long) height;

image = matrix;

for (i = 0; i < 256; i++) {
    shah[i] = 0;
    hash[i] = -1;
}

for (y = 0; y < height; y++) {
    index = (long) y * mwidth;
    for (x = 0; x < width; x++) {
        dummy = image[index++] % 256;
        hash[dummy] = 1;
    }
}
x = 0; y = 0;

nobjects = 0;
for (i = 0; i < 256; i++)
    if (hash[i] != -1) {
        hash[i] = nobjects;
        shah[nobjects] = i;
        nobjects++;
    }

if (nobjects <= 256)
    bits_pixel = 8;
if (nobjects <= 128)
    bits_pixel = 7;
if (nobjects <= 64)
    bits_pixel = 6;
if (nobjects <= 32)
    bits_pixel = 5;
if (nobjects <= 16)
    bits_pixel = 4;
if (nobjects <= 8)
    bits_pixel = 3;
if (nobjects <= 4)
    bits_pixel = 2;
if ( nobjects <= 2 )
    bits_pixel = 1;
colormap_size = 1 << bits_pixel;

if ( !have_c_map ) {
    FILE *cfp=fopen("map.ppm","r");
    int i,vred,vgreen,vblue;
    char buffer[121];

    Red =(char *) malloc((size_t) 256*sizeof(char));
    Green=(char *) malloc((size_t) 256*sizeof(char));
    Blue=(char *) malloc((size_t) 256*sizeof(char));

    fscanf(cfp,"%*s%*i%*i%*i");
    fgets(buffer,120,cfp);
    for ( i=0; i < 256; i++) {
        fgets(buffer,120,cfp);
        sscanf(buffer,"%i%i%i",&vred,&vgreen,&vblue);
        Red[i]=(char) vred;
        Green[i]=(char) vgreen;
        Blue[i] = (char) vblue;
    }
    fclose(cfp);
    have_c_map=1;
}

red =(char *) malloc((size_t) colormap_size*sizeof(char));
green=(char *) malloc((size_t) colormap_size*sizeof(char));
blue =(char *) malloc((size_t) colormap_size*sizeof(char));

for ( i =0; i < nobjects; i++) {
    red[i] = Red[ shah[i] ];
    green[i]= Green[ shah[i] ];
    blue[i] = Blue[ shah[i] ];
}

GIFEncode(0,bits_pixel,GetPixel);
free(red); free( green); free(blue);
return;
}

int
GetPixel(void)
{

112
long int index;
if ( y >= height ) return EOF;
index = (long) y * (long) mwidth + x;
x++;
if ( x >= width ) {
    y++;
    x=0;
}
return hash[ image[index] % 256 ];
}

void
GIFEncode(int Background,int BitsPerPixel,int (* getpixel)(void))
{
    int B;
    int InitCodeSize;
    int i;

    if( BitsPerPixel <= 1 )
        InitCodeSize = 2;
    else
        InitCodeSize = BitsPerPixel;

    fwrite("GIF87a", 1, 6, gfp );
    Putword(width, gfp ); Putword(height, gfp );
    B = 0x80;
    B |= (BitsPerPixel - 1) << 5;
    B |= (BitsPerPixel - 1);
    fputc( B, gfp );
    fputc( Background, gfp );
    fputc( 0, gfp );
    for( i=0; i< colormap_size; i++ ) {
        fputc( red[i], gfp );
        fputc( green[i], gfp );
        fputc( blue[i], gfp );
    }
    fputc( ',', gfp );
    Putword((int) 0, gfp ); Putword((int) 0, gfp );
    Putword(width, gfp ); Putword( height, gfp );
    /* Image is not interlaced */
    fputc( 0x00, gfp );
/* Initial code size */
fputc( InitCodeSize, gfp );

compress( InitCodeSize+1, getpid );

fputc( 0, gfp );
fputc( ';', gfp );
return;
}

void
Putword(int word, FILE *fp)
{
    fputc( word & 0xff, fp );
    fputc( (word / 256) & 0xff, fp );
    return;
}

/* encode encode encode encode */
void  output ();
void  cl_block ();
void  cl_hash ();
void writeerr ();
void  char_out ();
void  flush_char ();

int  n_bits;       /* number of bits/code */
int  maxbits =MY_BITS;  /* user settable max # bits/code */
code_int  maxcode=0;  /* maximum code, given n_bits */

/* should NEVER generate this code */
code_int  maxmaxcode = (code_int)1 << MY_BITS;
count_int  *HashTab0f;
unsigned short  *CodeTab0f;
code_int  hsize ;  /* for dynamic table sizing */
code_int  free_ent;  /* first unused entry */

int  clear_flg;
long int  in_count ;  /* length of input */
long int  out_count;  /* # of codes output (for debugging) */

int  g_init_bits;
int  ClearCode, EOFCode;
unsigned long  cur_accum;
int cur_bits;
unsigned long masks[] = { 0x0000, 0x0001, 0x0003, 0x0007, 0x000F, 
0x001F, 0x003F, 0x007F, 0x00FF, 
0x01FF, 0x03FF, 0x07FF, 0x0FFF, 
0x1FFF, 0x3FFF, 0x7FFF, 0xFFFF });
int a_count;
char_type *accum;

void
compress(int init_bits,int (*getpixel)(void))
{
    register long fcode;
    register code_int i;
    register int c;
    register code_int ent;
    register code_int disp;
    register code_int hsize_reg;
    register int hshift=0;

    a_count = 0;
cur_bits = 0;
cur_accum = 0;
in_count = 1;
out_count = 0;
clear_flg = 0;

g_init_bits = init_bits;
n_bits = init_bits;

maxcode = MAXCODE(n_bits);
maxbits = MY_BITS;
hsize = HSIZE;
ClearCode = (1 << (init_bits - 1));
EOFCode = ClearCode + 1;
free_ent = ClearCode + 2;

HashTableOf = ( count_int *) malloc((size_t)HSIZE*sizeof(count_int));
for ( i=0; i < HSIZE; i++)
    HashTableOf[i] = -1L;

CodeTabOf = ( unsigned short *)
    malloc((size_t)HSIZE*sizeof(unsigned short));
accum =(char_type *) malloc((size_t) 256 *sizeof(char_type));

ent = (*getpixel)();
for ( fcode = (long) hsize; fcode < 65536L; fcode *= 2L )
    ++hshift;

hshift = 8 - hshift; /* set hash code range bound */

hsize_reg = hsize;
cl_hash( (count_int) hsize_reg); /* clear hash table */

output( (code_int)ClearCode );

while ( (c = (*getpixel)()) != EOF ){
    ++in_count;

    fcode = (long) (((long) c << maxbits) + ent);
    i = (((code_int)c << hshift) ^ ent); /* xor hashing */

    if ( HashTabOf[i] == fcode ) {
        ent = CodeTabOf[i];
        continue;
    } else if ( (long)HashTabOf[i] < 0 ) /* empty slot */
        goto nomatch;

    disp = hsize_reg - i; /* secondary hash (after G. Knott) */
    if ( i == 0 )
        disp = 1;

    probe:
        if ( (i -= disp) < 0 )
            i += hsize_reg;
    
    if ( HashTabOf[i] == fcode ) {
        ent = CodeTabOf[i];
        continue;
    }

    if ( (long)HashTabOf[i] > 0 )
        goto probe;

nomatch:
    output ( (code_int) ent );
    out_count++;

    ent = c;

    if ( free_ent < maxmaxcode ) {
        CodeTabOf[i] = free_ent++; /* code -> hashtable */
        HashTabOf[i] = fcode;
    } else
        cl_block();
void output(code)
    code_int code;
{

    cur_accum &= masks[cur_bits];

    if( cur_bits > 0 )
        cur_accum |= ((long)code << cur_bits);
    else
        cur_accum = code;

    cur_bits += n_bits;

    while( cur_bits >= 8 ) {
        char_out( (unsigned int)(cur_accum & 0xff) );
        cur_accum >>= 8;
        cur_bits -= 8;
    }

    if ( free_ent > maxcode || clear_flg ) {
        if( clear_flg ) {
            maxcode = MAXCODE( n_bits = g_init_bits);
            clear_flg = 0;
        } else {
            ++n_bits;
            if ( n_bits == maxbits )
                maxcode = maxmaxcode;
            else
                maxcode = MAXCODE(n_bits);
        }
    }
}
if ( code == EOFCode ) {
    while ( cur_bits > 0 ) {
        char_out( (unsigned int)(cur_accum & Oxff) );
        cur_accum >>= 8;
        cur_bits -= 8;
    }
    flush_char();
    fflush( gfp);
    if( ferror( gfp ) )
        writeerr();
    return;
}

void cl_block () {
    cl_hash ( (count_int) hsize );
    free_ent = ClearCode + 2;
    clear_flg = 1;

    output( (code_int)ClearCode );
    return;
}

void cl_hash(hsize)
    count_int hsize;
{
    count_int *htab_p = HashTabOf+hsize;

    long i;
    long m1 = -1;

    i = hsize - 16;
    do {
        *(htab_p-16) = m1;
        *(htab_p-15) = m1;
        *(htab_p-14) = m1;
        *(htab_p-13) = m1;
        *(htab_p-12) = m1;
        *(htab_p-11) = m1;
        *(htab_p-10) = m1;
        *(htab_p-9) = m1;
    }
*(htab_p-8) = m1;
*(htab_p-7) = m1;
*(htab_p-6) = m1;
*(htab_p-5) = m1;
*(htab_p-4) = m1;
*(htab_p-3) = m1;
*(htab_p-2) = m1;
*(htab_p-1) = m1;
htab_p -= 16;
} while ((i -= 16) >= 0);

for ( i += 16; i > 0; --i )
**--htab_p = m1;
return;
}

void
writeerr()
{
    printf("error writing output file\n");
    exit(-1);
}

void
char_out(c)
    int c;
{
    accum[ a_count++ ] = c;
    if( a_count >= 254 )
        flush_char();
}

void
flush_char()
{
    if( a_count > 0 ) {
        fputc( a_count, gfp);
        fwrite( accum, 1, a_count, gfp );
        a_count = 0;
    }
    return;
}
A.7 main.c

// This is a simulation of
// Myxobacteria formation of fruiting bodies.
#include "medium.h"
#include "cell.h"
#include "write.gif.h"
#ifndef _STD_IO_
#include <stdio.h>
#define _STD_IO_
#endif
#include <string.h>
#ifndef _STD_LIB_
#include <stdlib.h>
#define _STD_LIB_
#endif
#include <memory.h>
#include <math.h>
#include <time.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_rng.h>

#define PRINT_INTERVAL 5
#define min(a,b) ((a) <= (b) ? (a) : (b))
#define max(a,b) ((a) >= (b) ? (a) : (b))

// holder -- linked list used to hold density values
struct holder{
    struct holder *next;
    int value[3];
};

// dist -- provides an array of probability cutoff values according to some distribution
Parameters
cutoffs- array to set

void
dist (double *cutoffs)
{
    int i;
}
for (i = 0; i < MAX_REVERSALI; i++)
{
    *(cutoffs + i) = gsl_ran_binomial_pdf (i, BINOM_P, MAX_REVERSAL);
}

/********************************************
get_numcells -- retrieve the number of cells reported by the
input file
Parameters
    infile -- pointer to the input file
**********************************************/
int get_numcells (FILE * infile)
{
    char temp[16];
    int num, done;
    done = 0;
    while (done == 0)
    {
        fgets (temp, sizeof (temp), infile);
        if (temp[0] == 35)
        {
            continue;
        }
        else
        {
            sscanf (temp, "%d", &num);
            done = 1;
        }
    }
    return (num);
}

/**********************************************
put_numcells -- print the number of cells in the system to the
output file
Parameters
    outfile -- pointer to output file
    numcells -- number of cells to write
***********************************************/
void put_numcells (FILE * outfile, int numcells)
{
    fprintf (outfile, "%d \n", numcells);
void get_size (FILE * infile, int *dims)
{
    int done = 0;
    char temp[20];
    while (done == 0)
    {
        fgets (temp, sizeof (temp), infile);
        if (temp[0] == 35)
        {
            continue;
        }
        else
        {
            sscanf (temp, "%d %d ", dims, (dims + 1));
            done = 1;
        }
    }
}

void put_size (FILE * outfile, int *dims)
{
    fprintf (outfile, "%d %d \n", *dims, *(dims + 1));
}

void get_cells -- get the cells from the input file and place on the lattice
Parameters
    infile -- pointer to the input file
here -- pointer to the site we are putting the cells
ctr -- pointer for the cell counter
r -- random number generator
*****************************************************************/
void
get_cells (FILE * infile, struct medium *here, int *ctr,
gsl_rng * r)
{
int j;
unsigned short int file_cell, file_slime, twopower;
fread (&file_cell, 1, sizeof (unsigned short int), infile);
for (j = 11; j >= 0; j--)
{
twopower = (unsigned int) pow (2, j);
if (file_cell >= twopower)
{
here->resident[j] = malloc (sizeof (struct cell));
cell_init (here->resident[j], here, j, 0, r);
(*ctr)++;
file_cell -= twopower;
}
}
for (j = 0; j < 3; j++)
{
fread (&file_slime, 1, sizeof (unsigned short int), infile);
here->slime[j] = file_slime;
}
fread (&file_cell, 1, sizeof (unsigned short int), infile);
here->new_nutrient = file_cell;
here->nutrient = file_cell;
here->old_nutrient = file_cell;
}

/*******************************************************************************/
put_cells -- write cells to the output file
Parameters
    outfile -- output file
    here -- site we are at
    ctr -- counter
*******************************************************************************/
void
put_cells (FILE * outfile, struct medium *here, int *ctr)
{
int j;
unsigned short int file_cell;
file_cell = 0;
for (j = 0; j < 12; j++)
{
    if (here->resident[j] != NULL)
    {
        file_cell += (int) pow (2, j);
    }
}
fwrite (&file_cell, 1, sizeof (unsigned short int), outfile);
for (j = 0; j < 3; j++)
{
    file_cell = (unsigned short int) here->slime[j];
    fwrite (&file_cell, 1, sizeof (unsigned short int), outfile);
}
file_cell = (unsigned short int) here->new_nutrient;
fwrite (&file_cell, 1, sizeof (unsigned short int), outfile);

/*******************************************************************************/
init_row -- create a row of sites and connect them
Parameters
    dim2 -- row length
    startnode -- first site of the row
    dim0 -- height of row
    dim1 -- depth of row
    downnode -- node below startnode
*******************************************************************************/
struct medium *
init_row (int dim0, struct medium *startnode, int dim1)
{
    struct medium *media_ptr;
    int i;
    media_ptr = startnode;
    if (dim1 == 0)
    { //The front row
        for (i = 1; i < dim0; i++)
        {
            media_ptr->next = malloc (sizeof (struct medium));
            if (media_ptr->next == NULL)
            {
                printf ("Out of memory.");
                exit (8);
            }
            medium_init (media_ptr->next, i, dim1);
            addneighbor (media_ptr, 3, media_ptr->next);
        }
    }
}
media_ptr = media_ptr->next;
} 
} 
else 
{ // not the front row
for (i = 1; i < dim0 - 1; i++)
{
media_ptr->next = malloc (sizeof (struct medium));
if (media_ptr->next == NULL)
{
printf ("Out of memory." acompanhado de memória);
exit (8);
} 
medium_init (media_ptr->next, i, dim1);
addneighbor (media_ptr, 3, media_ptr->next);
addneighbor (media_ptr->next, 5, media_ptr->neighbor[4]);
addneighbor (media_ptr->next, 4,
media_ptr->neighbor[4]->neighbor[3]);
media_ptr = media_ptr->next;
}
media_ptr->next = malloc (sizeof (struct medium));
if (media_ptr->next == NULL)
{
printf ("Out of memory." acompanhado de memória);
exit (8);
} 
medium_init (media_ptr->next, i, dim1);
addneighbor (media_ptr, 3, media_ptr->next);
addneighbor (media_ptr->next, 5, media_ptr->neighbor[4]);
if (media_ptr->neighbor[4]->neighbor[3] != NULL)
{
    addneighbor (media_ptr->next, 4,
media_ptr->neighbor[4]->neighbor[3]);
}
media_ptr = media_ptr->next;
}
return (media_ptr);
}

/**************************************************
finish_layer -- zip back to front
Parameters
        bottomnode -- front node
        topnode -- back node

125
void finish_layer(struct medium *bottomnode,
struct medium *topnode, int dim)
{
    struct medium *bottom_ptr, *top_ptr;
    int i;
    bottom_ptr=bottomnode;
    top_ptr=topnode;
    for(i=0;i<dim;i++)
    {
        addneighbor(top_ptr, 2, bottom_ptr);
        if(bottom_ptr->neighbor[0] != NULL)
        {
            addneighbor(top_ptr, 1, bottom_ptr->neighbor[0]);
        }
        bottom_ptr=bottom_ptr->next;
        top_ptr=top_ptr->next;
    }
}

put_outdens -- write out density image file

Parameters
    of -- output file
    firstnode -- first node in the system
    dim -- dimension array

void put_outdens (FILE * of, struct medium *firstnode,
    struct holder *first, int *hash_val, int *dim)
{
    struct medium *node;
    struct holder *now;
    int i, max[2];
    node = firstnode;
    now = first;
    while (now != NULL)
    {

now->value[0] = 0;
now->value[1] = 0;
now->value[2] = 0;
now = now->next;
}
now = first;
max[0] = 0;
max[1] = 0;
while ((now != NULL) && (node != NULL))
{
    for (i = 0; i < 3; i++)
    {
        now->value[0] += node->numcells[i];
        now->value[2] += node->slime[i];
    }
    if (max[0] < now->value[0])
    {
        max[0] = now->value[0];
    }
    if (max[1] < now->value[2])
    {
        max[1] = now->value[2];
    }
    now->value[1] +=
min ((int) (max(node->nutrient,0.0) / NUTRIENT_CUT), 1);

now = now->next;
node = node->next;
}
if ( (!((now == NULL) && (node == NULL)))
{
    printf ("Mismatch of holders to nodes");
}
now = first;
i = 0;
while (now != NULL)
{
    if (max[1] == 0)
    {
        hash_val[i] =
255 -
((int) min (now->value[0] , 7.0)) -
8 * now->value[1];
    }
else

{  
    hash_val[i] = 255 -
    ((int) min (now->value[0], 7.0))
    - 8 * now->value[1] -
    16 * min ((int)
            ((16.0 *
                ((float) now->value[2])) /
                (float) max[1]), 15);
}
now = now->next;
i++;
}
write_gif (of, hash_val, dim[0], dim[1], dim[0], dim[1]);
}

void
printconstants (int x, int y, int swtch)
{
    printf("Dimensions: x=%d, y=%d\n", x, y);
    printf("Jamming Threshold: %d\n", CELL_THRESHOLD);
    printf("Beta: %f\n", BETA);
    printf("Straight weight: %f\n", STRAIGHT / (STRAIGHT + 2.0));
    printf("Medium size: %d\n", sizeof (struct medium));
    printf("Cell size: %d\n", sizeof (struct cell));
    printf("Average Reversal Time: %f steps\n", MAX_REVERSAL * BINOM_P);
    printf("Diffusion Coefficient: %f\n", DIFFUSION_RATE);
    printf("Time step: %f seconds\n", TIMESTEP);
    printf("Lattice Spacing: %f micrometers\n", SPACING);
    printf("Average division time: %f hours\n", GEN_TIME / 60.0);
    printf("Mutant Type: %d\n", swtch);
    printf("SYN_SLIME: %f \n", SYN_SLIME);
    printf("SYN_ALIGN: %f \n", SYN_ALIGN);
    printf("SYN_DENS: %f \n", SYN_DENS);
    printf("SYN_DEFLECT: %f \n", SYN_DEFLECT);
    printf("SYN_OBS: %f \n", SYN_OBS);
}

int
main (int argc, char *argv[])
{
char *ifn;
FILE *infile;
char *ofp;
char temp[64];
FILE *outdens, *outfile;
int dimension[2];
int num_cells;
int i, j, counter, padding, num HOLDERS, mutant_switch;
int move_tries, move_successes;
char step[32], end[16];
struct medium *firstnode, *startnode, *endnode;
struct holder *first, *now;
int *hash_val;
int sub_steps;
int time_steps;
double distribution[MAX_REVERSALI];
const gsl_rng_type *T;
gsl_rng *r;
gsl_rng_env_setup ();
T = gsl_rng_default;
r = gsl_rng_alloc (T);
if (argc != 5)
{
    fprintf (stderr, 
"Usage: is gomyxo inputfilename outputfileprefix 
    number_of_steps mutant_switch\n"
);
    exit (8);
}
ifn = argv[1];
//Open Files----------------------------------------------
infile = fopen (ifn, "r"); //Open input file
if (infile == NULL)
{
    fprintf (stderr, "Check input file name.\n");
    exit (8);
}
num_cells = get_numcells (infile);
ge t_size (infile, dimension);
counter = 0;
firstnode = malloc (sizeof (struct medium));
medium_init (firstnode, 0, 0);
startnode = firstnode;
for (j = 0; j < (dimension[1] - 1); j++)
{
    // do all but the back row
    endnode = init_row (dimension[0], startnode, j);
endnode->next = malloc(sizeof(struct medium));
medium_init(endnode->next, 0, j + 1);
if (j % 2 == 0)
{
  addneighbor(startnode, 1, endnode->next);
  startnode = endnode->next;
}
else
{
  addneighbor(startnode, 2, endnode->next);
  addneighbor(startnode->neighbor[3], 1, endnode->next);
  startnode = endnode->next;
}
endnode = init_row(dimension[0], startnode, j);
finish_layer(firstnode, startnode, dimension[0]);
sscanf(argv[3], "%d", &time_steps);
sscanf(argv[4], "%d", &mutant_switch);
printconstants(dimension[0], dimension[1], mutant_switch);
startnode = firstnode;
while (startnode != NULL)
{
  get_cells(infile, startnode, &counter, r);
  startnode = startnode->next;
}
printf("We placed %d cells.\n", counter);
fclose(infile); //close infile
padding = (int) log10((double) time_steps);
hash_val = malloc((size_t)(dimension[0] *
dimension[1] * sizeof(int)));
first = malloc(sizeof(struct holder));
first->value[0] = 0;
first->value[1] = 0;
hash_val[0] = 0;
now = first;
num HOLDERS = 1;
for (i = 1; i < (dimension[0] * dimension[1]); i++)
{
  now->next = malloc(sizeof(struct holder));
  now = now->next;
  now->value[0] = 0;
  now->value[1] = 0;
  hash_val[i] = 0;
  num HOLDERS++;
}
dist (distribution);
strcpy (step, "00000.gif");
ofp = argv[2];
strcpy (temp, ofp);
outd = fopen (strcat (temp, step), "w");
    /** density from top down. **/
put_outd (outd, firstnode, first, hash_val, dimension);
fclose (outd);
move_tries = 0;
move_successes = 0;
if (mutant_switch == 0)
{
    sub_steps = 4;
}
else if (mutant_switch == 1)
{
    sub_steps = 1;
}
else
{
    sub_steps = 1;
}
startnode = firstnode;
while (startnode != NULL)
{
    if (startnode->numcells[0] +
        startnode->numcells[1] +
        startnode->numcells[2] > 0
    || startnode->oldnumcells[0] +
        startnode->oldnumcells[1] +
        startnode->oldnumcells[2] > 0)
    {
        set_numcells (startnode);
    }
    startnode = startnode->next;
}
for (i = 0; i < time_steps; i++)
{
    for (j = 0; j < sub_steps; j++)
    {
        startnode = firstnode;
        while (startnode != NULL)
        {
            if (startnode->numhere > 0)
            {
align_cells (startnode, r, mutant_switch);
}
startnode = startnode->next;
}
startnode = firstnode;
while (startnode != NULL)
{
    {
        set_numcells (startnode);
    }
    startnode = startnode->next;
}
startnode = firstnode;
while (startnode != NULL)
{
    if (startnode->numhere > 0)
    {
        move_cells (startnode, &counter, &move_tries, &move_successes, r);
    }
    startnode = startnode->next;
}
startnode = firstnode;
while (startnode != NULL)
{
    {
        set_numcells (startnode);
    }
    if (startnode->numhere > 0)
    {
        if (mutant_switch != 1)
        {
            slimeit (startnode, SLIME_COEF_PROD);
        }
        moved (startnode);
    }
    startnode = startnode->next;
}
startnode = firstnode;
while (startnode != NULL)
{
    if (startnode->numhere > 0)
    {
        growth (startnode, &counter, r);
        reversal (startnode, i, distribution, r);
        moved (startnode);
    }
    startnode = startnode->next;
}
startnode = firstnode;
while (startnode != NULL)
{
    if (startnode->numcells[0] + startnode->numcells[1] +
        startnode->numcells[2] > 0 || startnode->oldnumcells[0] +
    {
        set_numcells (startnode);
    }
    diffuse_nutrient (startnode);
    startnode = startnode->next;
}
startnode = firstnode;
while (startnode != NULL)
{
    set_nutrient (startnode, r);
    startnode = startnode->next;
}
if (i % PRINT_INTERVAL == 0)
{
    switch (padding - (int) log10 ((double) i + 1))
    {
        case 5:
            strcpy (step, "00000");
            break;
        case 4:
            strcpy (step, "0000");
            break;
        case 3:
            strcpy (step, "000");
            break;
        case 2:
            strcpy (step, "00");
            break;
        case 1:
            133
    }
strcpy (step, "0");
break;
default:
strcpy (step, "");
}
sprintf (end, "%d.gif", i + 1);
strcat (step, end);
ofp = argv[2];
strcpy (temp, ofp);
outdens = fopen (strcat (temp, step), "w"); //open picture file
    /** density from top down. **/
put_outdens (outdens, firstnode, first, hash_val, dimension);
fclose (outdens);
}
}
printf ("Tries: %d, Successes:%d, Percentage: %f\n", move_tries,
                move_successes, (float) move_successes / (float) move_tries);
strcpy (temp, ofp);
outfile = fopen (strcat (temp, ".out"), "w"); //open data file
if (outdens == NULL || outfile == NULL)
{
printf ("Check output file.\n");
}
put_numcells (outfile, counter);
counter = 0;
put_size (outfile, dimension);
startnode = firstnode;
do
{
    { 
put_cells (outfile, startnode, &counter);
startnode = startnode->next;
    }
while (startnode->next != NULL);
fclose (outfile);
return (0);
}
B.1 agar.c

/**************************************************************************
 *                 agar.c
 *
 * Tue Mar 28 13:15:58 2006
 * Copyright 2006 User
 * Email
**************************************************************************/
#include "cell.h"
#include "defs.h"
#include "functions.h"
#include "agar.h"
#include <gsl/gsl_math.h>
#ifndef GSLRNG
# define GSLRNG
# include <gsl/gsl_rng.h>
#endif
# include <stdio.h>

struct agar *surface[DIMENSION][DIMENSION];

void
remove_cell (struct cell *thisun)
{
  int i;
  int nowcell;
  struct agar *here;
  here = thisun->here;
  for (i = 0; i < here->num_here; i++)
  {
    if (thisun == here->cells[i])
    {
      nowcell = i;
    }
    else
    {
      here->num_here--;
      for (j = i; j < here->num_here; j++)
      {  
        (here->cells[j] = here->cells[j+1]);  
     }
    }
  }
  here->num_here = nowcell;
}


nowcell = i;
break;
}
}
for (i = nowcell; i < here->num_here - 1; i++)
{
    here->cells[i] = here->cells[i + 1];
}
here->cells[here->num_here - 1] = NULL;
here->num_here--;

void
add_cell (struct cell *thisun)
{
    int x, y;
    x = (int) (thisun->position[0]);
    y = (int) (thisun->position[1]);
    surface[x][y]->cells[surface[x][y]->num_here] = thisun;
    thisun->here = surface[x][y];
    thisun->here->num_here++;
}

int
fibril_dens (double attachpt[2], struct cell *here,
    double frontpt[2], int t)
{
    int i, j, k, imod, jmod;
    double debug;
    double dx, dy, dist, strength;
    struct cell *thisun;
    double pts[3][2];
    int max_x, max_y, min_x, min_y;

    pts[1][0] = attachpt[0];
    pts[1][1] = attachpt[1];
    pts[0][0] = frontpt[0];
    pts[0][1] = frontpt[1];
    strength = 0;
    min_x = (int) (fmod (attachpt[0] -
        (2 * INIT_LENGTH + FIBRIL_LENGTH),
        DIMENSION));
    min_y =
        (int) (fmod
        (attachpt[1] - (2 * INIT_LENGTH + FIBRIL_LENGTH),
        }
max_x =
    (int) (fmod
    (attachpt[0] + (2 * INIT_LENGTH + FIBRIL_LENGTH) + 1,
    DIMENSION));
max_y =
    (int) (fmod
    (attachpt[1] + (2 * INIT_LENGTH + FIBRIL_LENGTH) + 1,
    DIMENSION));
for (i = min_x; mod (i, DIMENSION) != max_x; i++)
{
    imod = mod (i, DIMENSION);
    for (j = min_y; mod (j, DIMENSION) != max_y; j++)
    {
        jmod = mod (j, DIMENSION);
        for (k = 0; k < surface[imod][jmod]->num_here; k++)
        {
            thisun = surface[imod][jmod]->cells[k];
            if (thisun != here)
            {
                pts[2][0] = thisun->position[0];
                pts[2][1] = thisun->position[1];
                dx = wrap_subtract (pts[1][0], pts[2][0]);
                dy = wrap_subtract (pts[1][1], pts[2][1]);
                dist = sqrt (dx * dx + dy * dy);
                if (dist < 2 * INIT_LENGTH + FIBRIL_LENGTH)
                {
                    if (fabs (internal_angle (pts)) > FIB_ANG_CUT)
                    {
                        strength += 1;
                        // printf("t=%d, Attached to cell at (%f,%f)
                        // from (%f,%f) thru (%f,%f)\n", t,
                        // pts[2][0],pts[2][1], pts[0][0],pts[0][1],
                        // pts[1][0],pts[1][1]);
                    }
                    else
                    {
                        pts[2][0] =
                        thisun->position[0] +
                        thisun->length * cos (thisun->angle);
                        pts[2][1] =
                        thisun->position[1] +
                        thisun->length * sin (thisun->angle);
                        dist = sqrt (dx * dx + dy * dy);
                        if (dist < 2 * INIT_LENGTH + FIBRIL_LENGTH)
thisun->position[1] +
    thisun->length * sin (thisun->angle);
    dx = wrap_subtract (pts[1][0], pts[2][0]);
    dy = wrap_subtract (pts[1][1], pts[2][1]);
    dist = sqrt (dx * dx + dy * dy);
    if (dist < FIBRIL_LENGTH)
    {
        if (fabs (internal_angle (pts)) >
            FIB_ANG_CUT)
        {
            strength += 1;
            // printf("t=%d, Attached to cell at
            // (%f,%f)
            // from (%f, %f) thru (%f,%f)\n",
            // t, pts[2][0],pts[2][1], pts[0][0],
            // pts[0][1], pts[1][0],pts[1][1]);
        }
    }
}
else
{
    pts[2][0] =
        thisun->position[0] +
        thisun->length * cos (thisun->angle);
    pts[2][1] =
        thisun->position[1] +
        thisun->length * sin (thisun->angle);
    dx = wrap_subtract (pts[1][0], pts[2][0]);
    dy = wrap_subtract (pts[1][1], pts[2][1]);
    dist = sqrt (dx * dx + dy * dy);
    if (dist < FIBRIL_LENGTH)
    {
        if (fabs (internal_angle (pts)) >
            FIB_ANG_CUT)
        {
            strength += 1;
            // printf("t=%d, Attached to cell at
            // (%f,%f)
            // from (%f, %f) thru (%f,%f)\n",
            // t, pts[2][0],pts[2][1], pts[0][0],
            // pts[0][1], pts[1][0],pts[1][1]);
        }
    }
}
if (strength > FIB_CUT)
{
    return (1);
}
else
{
    return (0);
}

void
check_triangle (double pt1[2], double pt2[2], double pt3[2],
double barrier[3], struct cell **obstacle, int t)
{
    int min_x, min_y, max_x, max_y;
    int i, imod, j, jmod, k;
    double v0[2], v1[2], v2[2];
    double checkpoints[3][2], dx, dy, m_32, m_13, m_21;
    double ang[2], temp, m;
    double x[2], y[2];
    struct cell *that;
    unsigned int test;

    v2[0] = wrap_subtract (pt1[0], pt2[0]); // x component
    // from head to
    // tail
    v0[0] = wrap_subtract (pt3[0], pt1[0]); // x component
    // from tail to
    // attachment

    min_x =
    mod ((int)
         (min (pt1[0], min (pt1[0] - v0[0], pt1[0] + v2[0])) -
          (2 * INIT_LENGTH)), DIMENSION);
    max_x =
    mod ((int)
         (max (pt1[0], max (pt1[0] - v0[0], pt1[0] + v2[0])) +
          (2 * INIT_LENGTH)) + 1, DIMENSION);
    // +1 is to check highest possible spot in for loop
v2[1] = wrap_subtract(pt1[1], pt2[1]); // y """
v0[1] = wrap_subtract(pt3[1], pt1[1]); // y " " " "

min_y =
    mod((int)
        (min(pt1[1], min(pt1[1] - v0[2], pt1[1] + v2[1])) -
        (2 * INIT_LENGTH)), DIMENSION);
max_y =
    mod((int)
        (max(pt1[1], max(pt1[1] - v0[2], pt1[1] + v2[1])) +
        (2 * INIT_LENGTH)) + 1, DIMENSION);

// check if the cell crosses the triangle
// use barycentric coordinates
// see http://www.blackpawn.com/texts/pointinpoly/default.html
v1[0] = wrap_subtract(pt2[0], pt3[0]); // x component
// from attachment
// to head
v1[1] = wrap_subtract(pt2[1], pt3[1]); // y """

m_13 = (v0[1]) / (v0[0]);
m_32 = (v1[1]) / (v1[0]);
m_21 = (v2[1]) / (v2[0]);

checkpoints[0][0] = pt2[0];
checkpoints[0][1] = pt2[1];
checkpoints[1][0] = pt1[0];
checkpoints[1][1] = pt1[1];
checkpoints[2][0] = pt3[0];
checkpoints[2][1] = pt3[1];
temp = internal_angle(checkpoints);
// printf("t=%d,This cell: tail=(%f,%f)
// head=(%f,%f)\tAttachment=(%f,%f)\n",
// t,pt1[0],pt1[1],pt2[0],pt2[1],pt3[0],pt3[1]);

for (i = min_x; mod(i, DIMENSION) != max_x; i++)
{
    imod = mod(i, DIMENSION);
    for (j = min_y; mod(j, DIMENSION) != max_y; j++)
    {
        jmod = mod(j, DIMENSION);
        for (k = 0; k < (surface[imod][jmod])->num_here; k++)
        {

140
that = surface[imod][jmod]->cells[k];
if ((pt1[0] != that->position[0]
    || pt1[1] != that->position[1])
    && that != *obstacle)
{
    x[0] = that->position[0];
    x[1] =
        that->position[0] + cos (that->angle) * that->length;
    y[0] = that->position[1];
    y[1] =
        that->position[1] + sin (that->angle) * that->length;
    test = 0;
    // printf("That cell: tail=(%f,%f) head=(%f,%f)\n",
             x[0],y[0],x[1],y[1]);
    dx = wrap_subtract (x[0], pt2[0]);
    dy = wrap_subtract (y[0], pt2[1]);
    if (((((pt1[0] <= x[0]) && (pt1[0] >= x[1])
          || ((pt1[0] >= x[0]) && (pt1[0] <= x[1]))) ||
          (((pt2[0] <= x[0]) && (pt2[0] >= x[1]))) ||
          (((pt1[0] <= x[0]) && (pt2[0] <= x[0]))) ||
          (((pt1[0] <= x[1]) && (pt2[0] >= x[1]))))
        &&
        (((pt1[1] <= y[0]) && (pt1[1] >= y[1])
          || ((pt1[1] >= y[0]) && (pt1[1] <= y[1]))) ||
          (((pt1[1] <= y[0]) && (pt2[1] <= y[0]))) ||
    { // they can cross
        /* (y-pt2[1])=m_21*(x-pt2[0])
(y-that->position[1])=tan(that->angle)*(x-that->position[0])*/
        (y-that->position[1])=tan(that->angle)*(x-that->position[0])
        //m_21*(x-pt2[0])+pt2[1] =
        tan(that->angle)*(x-that->position[0])+that->position[1]
        //m_21*(x-pt2[0])+pt2[1] =
        tan(that->angle)*(x-(that->position[0]+pt2[0]-pt2[0]))
        +(that->position[1]+pt2[1]-pt2[1]) //m_21*(x-pt2[0])+pt2[1] =
        tan(that->angle)*(x-(dx+pt2[0]))+(dy+pt2[1])
        //(m_21-tan(that->angle))*x=
        m_21*pt2[0]-pt2[1]-tan(that->angle)*(dx+pt2[0]))+(dy+pt2[1])
        //(m_21-tan(that->angle))*x=
        (m_21-tan(that->angle))*pt2[0]-tan(that->angle)*dx)+dy

141
// x = pt2[0] + (dy - dx * tan(that->angle)) / (m_21 - tan(that->angle)) */

m =
    wrap_subtract (y[1], y[0]) / wrap_subtract (x[1], x[0]);
if (isinf (m_21))
    temp = pt2[0];
else if (isinf (m))
    temp = x[0];
else
{
    temp = pt2[0] + (dy - dx * m) / (m_21 - m);
    // x value of where that's line crosses this
    // cell's line
}
if (!(((temp < x[0] || temp > x[1])
    && (temp < x[1] || temp > x[0]))
    || ((temp < pt2[0] || temp > pt1[0])
    && (temp > pt2[0] || temp < pt1[0])))
{ // does cross this cell
    test = 1;
}
else
{
    test = 2;
}
}
if (test % 2 == 0)
{
    test = 0;
if (((v0[0] * wrap_subtract (y[0], pt3[1])
    - v0[1] * wrap_subtract (x[0], pt3[0])) *
    (v1[0] * wrap_subtract (y[0], pt2[1])
    - v1[1] * wrap_subtract (x[0], pt2[0])) > 0)
&&
    ((v1[0] * wrap_subtract (y[0], pt2[1]) -
    v1[1] * wrap_subtract (x[0],
    pt2[0]))) * (v2[0] * wrap_subtract (y[0], pt1[1]) -
    v2[1] * wrap_subtract (x[0], pt1[0]))
> 0))
{
    test = 1; // tail inside triangle
}
if (((v0[0] * wrap_subtract (y[1], pt3[1])
    - v0[1] * wrap_subtract (x[1], pt3[0])) *
    (v1[0] * wrap_subtract (y[1], pt2[1]))
- v1[1] * wrap_subtract (x[1], pt2[0])) > 0)

&
(v1[0] * wrap_subtract (y[1], pt2[1]) -
 v1[1] * wrap_subtract (x[1],
 pt2[0])) * (v2[0] *
warp_subtract
(y[1],
 pt1[1]) -
v2[1] *
warp_subtract
(x[1],
 pt1[0]))
 > 0))
{
 test += 2; // head inside triangle
}
if (test)
{
/* (y-pt2[1])=m_32*(x-pt2[0])
 (y-that->position[1])=tan(that->angle)*(x-that->position[0])
 //m_32*(x-pt2[0])+pt2[1]=
 tan(that->angle)*(x-that->position[0])+that->position[1]
 //m_32*(x-pt2[0])+pt2[1]=
 tan(that->angle)*(x-(that->position[0]+pt2[0]-pt2[0]))
 +(that->position[1]+pt2[1]-pt2[1]) //m_32*(x-pt2[0])+pt2[1]=
 tan(that->angle)*(x-((dx+pt2[0]))+(dy+pt2[1])
 //m_32=(x-((dx+pt2[0]))+(dy+pt2[1])
 +(m_32-tan(thar->angle))*x=
 m_32*pt2[0]-pt2[1]-tan(thar->angle)*(dx+pt2[0])+(dy+pt2[1])
 //m_32=(x-((dx+pt2[0]))+(dy+pt2[1])
 +(m_32-tan(thar->angle))*x=
 (m_32-tan(thar->angle))*pt2[0]-tan(thar->angle)*dx+dy
 //x=pt2[0]-(dy-dx*tan(thar->angle))/(m_32-tan(thar->angle) */
if (test == 1)
{
 // tail in
 m =
 wrap_subtract (y[1],
 y[0]) / wrap_subtract (x[1],
 x[0]);
 if (isinf (m_32))
 temp = pt2[0];
 else if (isinf (m))
 temp = x[0];
 else
{
 temp =
 pt2[0] + (dy - dx * m) / (m_32 - m);
checkpoints[2][0] = x[0];
checkpoints[2][1] = y[0];
ang[1] = internal_angle (checkpoints);
if (ang[1] * barrier[2] > 0) {
    if (fabs (ang[1]) < fabs (barrier[2])) {
        barrier[2] = ang[1];
        m =
            wrap_subtract (y[0], pt1[1]) /
            wrap_subtract (x[0], pt1[0]);
        dx = wrap_subtract (pt2[0], pt1[0]);
        dy = wrap_subtract (pt2[1], pt1[1]);
        if (isinf (m_32)) {
            checkpoints[2][0] = pt1[0];
            checkpoints[2][1] = m * dx + pt1[1];
        } else if (isinf (m)) {
            checkpoints[2][0] = pt1[0];
            checkpoints[2][1] =
                m_32 * wrap_subtract (checkpoints[2][0],
                pt2[0]) + pt2[1];
        } else {
            checkpoints[2][0] =
                (dy - m_32 * dx) / (m - m_32) +
                pt1[0];
            checkpoints[2][1] =
                m_32 * wrap_subtract (checkpoints[2][0],
                pt2[0]) + pt2[1];
        }
        barrier[0] =
            wrap_subtract (checkpoints[2][0],
            pt1[0]);
        barrier[1] =
            144
wrap_subtract (checkpoints[2][1], pt1[1]);
*obstacle = that;
// printf("Jammed tail t=%d\t Contact
// at (%f,%f)\t,
// Should move (%f,%f), Angle= %f\n",
// t,
// checkpoints[2][0],checkpoints[2][1]
// ,barrier[0],barrier[1],
// barrier[2]);

if (((temp >= x[0] && temp <= x[1])
 || (temp >= x[1] && temp <= x[0])))
 || ((temp <= pt2[0] && temp >= pt3[0])
 || (temp >= pt2[0] && temp <= pt3[0])))
{
    m =
    wrap_subtract (y[1], y[0]) /
    wrap_subtract (x[1], x[0]);
    checkpoints[2][0] = temp;
    if (isinf (m_32))
    {
        checkpoints[2][1] =
        m * wrap_subtract (temp, x[0]) + y[0];
    }
    else
    {
        checkpoints[2][1] =
        m_32 * wrap_subtract (temp, pt2[0]) +
        pt2[1];
    }
    ang[0] = internal_angle (checkpoints);
    if (ang[0] * barrier[2] > 0)
    {
        if (fabs (ang[0]) < fabs (barrier[2]))
        {
            barrier[0] =
            wrap_subtract (temp, pt1[0]);
            barrier[1] =
            wrap_subtract (checkpoints[2][1],
            pt1[1]);
            barrier[2] = ang[0];
    }
*obstacle = that;

// printf("Jammed body 1 t=%d\n"
// Contact at (%f,%f)\n,
// Should move (%f,%f), Angle=
// %f\n",
// t,
// checkpoints[2][0],checkpoints[2][1],
// barrier[0],barrier[1],
// barrier[2]);

else if (test == 2)
{
    // head in
    m =
    wrap_subtract (y[1],
    y[0]) / wrap_subtract (x[1],
    x[0]);
    if (isinf (m_32))
        temp = pt2[0];
    else if (isinf (m))
        temp = x[0];
    else
    {
        temp =
        pt2[0] + (dy - dx * m) / (m_32 - m);
        // x coordinate of where this cell’s line
        // intersects
        // the head-attachment line
    }
    checkpoints[2][0] = x[1];
    checkpoints[2][1] = y[1];
    ang[1] = internal_angle (checkpoints);    
    {
        if (fabs (ang[1]) < fabs (barrier[2]))
        {
            barrier[2] = ang[1];
            m =
            wrap_subtract (y[1],
            pt1[1]) /
            wrap_subtract (x[1], pt1[0]);
            dx = wrap_subtract (pt2[0], pt1[0]);
            dy = wrap_subtract (pt2[1], pt1[1]);
if (isinf (m_32))
{
  checkpoints[2][0] = pt1[0];
  checkpoints[2][1] = m * dx + pt1[1];
}
else if (isinf (m))
{
  checkpoints[2][0] = pt1[0];
  checkpoints[2][1] =
  m_32 *
  wrap_subtract (checkpoints[2][0],
  pt2[0]) + pt2[1];
}
else
{
  checkpoints[2][0] =
  (dy - m_32 * dx) / (m - m_32) +
  pt1[0];
  checkpoints[2][1] =
  m_32 *
  wrap_subtract (checkpoints[2][0],
  pt2[0]) + pt2[1];
}
barrier[0] =
  wrap_subtract (checkpoints[2][0],
  pt1[0]);
barrier[1] =
  wrap_subtract (checkpoints[2][1],
  pt1[1]);
*obstacle = that;
// printf("Jammed head t=%d\t Contact
// at (%f,%f)\t,
// Should move (%f,%f), Angle= %f\n",
// t,
// checkpoints[2][0],checkpoints[2][1],
// barrier[0],barrier[1], barrier[2]);
}
}

  if (((temp >= x[0] && temp <= x[1])
  || (temp >= x[1] && temp <= x[0]))
  && ((temp <= pt2[0] && temp >= pt3[0])
  || (temp >= pt2[0] && temp <= pt3[0])))
  {
    m =
wrap_subtract (y[1], y[0]) /
wrap_subtract (x[1], x[0]);
checkpoints[2][0] = temp;
if (isinf (m_32))
{
    checkpoints[2][1] =
    m * wrap_subtract (temp, x[0]) + y[0];
}
else
{
    checkpoints[2][1] =
    m_32 * wrap_subtract (temp, pt2[0]) +
    pt2[1];
}
ang[0] = internal_angle (checkpoints);
if (ang[0] * barrier[2] > 0)
{
    if (fabs (ang[0]) < fabs (barrier[2]))
    {
        barrier[0] =
        wrap_subtract (temp, pt1[0]);
        barrier[1] =
        wrap_subtract (checkpoints[2][1],
                       pt1[1]);
        barrier[2] = ang[0];
        *obstacle = that;
        // printf("Jammed body 2 t=%d\t"  
        // Contact at (%f,%f)\t, 
        // Should move (%f,%f), Angle= 
        // %f\n", 
        // t, 
        // checkpoints[2][0],checkpoints[2][1], 
        // barrier[0],barrier[1], 
        // barrier[2]);
    }
}
}
else
{
    // both in
    checkpoints[2][0] = x[1];
    checkpoints[2][1] = y[1];
    ang[1] = internal_angle (checkpoints);
checkpoints[2][0] = x[0];
checkpoints[2][1] = y[0];
ang[0] = internal_angle(checkpoints);
if (ang[0] * barrier[2] > 0)
{
  if (fabs(ang[0]) < fabs(barrier[2]))
  {
    barrier[2] = ang[0];
    m =
      wrap_subtract(y[0],
      pt1[1]) /
      wrap_subtract(x[0], pt1[0]);
    dx = wrap_subtract(pt2[0], pt1[0]);
    dy = wrap_subtract(pt2[1], pt1[1]);
    if (isinf(m_32))
    {
      checkpoints[2][0] = pt1[0];
      checkpoints[2][1] = m * dx + pt1[1];
    }
    else if (isinf(m))
    {
      checkpoints[2][0] = pt1[0];
      checkpoints[2][1] = m_32 *
      wrap_subtract(checkpoints[2][0],
      pt2[0]) + pt2[1];
    }
    else
    {
      checkpoints[2][0] =
      (dy - m_32 * dx) / (m - m_32) +
      pt1[0];
      checkpoints[2][1] =
      m_32 *
      wrap_subtract(checkpoints[2][0],
      pt2[0]) + pt2[1];
    }
    barrier[0] =
      wrap_subtract(checkpoints[2][0],
      pt1[0]);
    barrier[1] =
      wrap_subtract(checkpoints[2][1],
      pt1[1]);
    *obstacle = that;
    // printf("Jammed tail 1 t=%d\t

// Contact at (%f,%f)
// Should move (%f,%f), Angle= %f
// t,
// checkpoints[2][0],checkpoints[2][1],
// barrier[0],barrier[1], barrier[2]);
}
}
{
    if (fabs (ang[1]) < fabs (barrier[2]))
    {
        barrier[2] = ang[1];
        m =
            wrap_subtract (y[1],
                          pt1[1]) /
            wrap_subtract (x[1], pt1[0]);
        dx = wrap_subtract (pt2[0], pt1[0]);
        dy = wrap_subtract (pt2[1], pt1[1]);
        if (isinf (m_32))
        {
            checkpoints[2][0] = pt1[0];
            checkpoints[2][1] = m * dx + pt1[1];
        }
        else if (isinf (m))
        {
            checkpoints[2][0] = pt1[0];
            checkpoints[2][1] =
                m_32 *
                wrap_subtract (checkpoints[2][0],
                               pt2[0]) + pt2[1];
        }
        else
        {
            checkpoints[2][0] =
                (dy - m_32 * dx) / (m - m_32) +
                pt1[0];
            checkpoints[2][1] =
                m_32 *
                wrap_subtract (checkpoints[2][0],
                               pt2[0]) + pt2[1];
        }
        barrier[0] =
            wrap_subtract (checkpoints[2][0],
                           pt1[0]);
        barrier[1] =
            150
wrap_subtract (checkpoints[2][1],
pt1[1]);
*obstacle = that;
// printf("Jammed head 1 t=%d\t
// Contact at (%f,%f)\t
// Should move (%f,%f), Angle= %f\n",
// t,
// checkpoints[2][0],checkpoints[2][1],
// barrier[0],barrier[1], barrier[2]);
}

}
}

else
{

test = 0;

if ((((((pt3[0] <= x[0]) && (pt3[0] >= x[1]))
|| ((pt3[0] >= x[0])
   && (pt3[0] <= x[1])))
|| (((pt2[0] <= x[0]) && (pt2[0] >= x[1]))
|| ((pt2[0] >= x[0])
   && (pt2[0] <= x[1])))
|| (((pt3[0] <= x[0]) && (pt2[0] >= x[0]))
|| ((pt3[0] >= x[0])
   && (pt2[0] <= x[0])))
|| (((pt3[1] <= y[0]) && (pt3[1] >= y[1]))
|| ((pt3[1] >= y[0])
   && (pt3[1] <= y[1])))
|| ((pt2[1] >= y[0])
   && (pt2[1] <= y[1])))
   && ((pt3[1] <= y[0]) && (pt3[1] >= y[1]))
   && ((pt3[1] >= y[0])
   && (pt3[1] <= y[1]))
   && (pt2[1] <= y[0]))
   && (pt2[1] >= y[0]))
   && (pt2[1] <= y[1]))
   && (pt2[1] >= y[1]))
   && (pt2[1] <= y[0]))
   && (pt2[1] >= y[0])))
   { test = 1; // that cell can overlap the head
// attachment line. 
}
if (test)
{
    m =
    wrap_subtract (y[1],
y[0]) / wrap_subtract (x[1],
x[0]);
    if (isinf (m_32))
        temp = pt2[0];
    else if (isinf (m))
        temp = x[0];
    else
        {
        temp =
        pt2[0] + (dy - dx * m) / (m_32 - m);
        // x coordinate of where this cell’s line
        // intersects the head-attachment line
        // checkpoints[2][0]=pt3[0];
        // checkpoints[2][1]=pt3[1];
        if (((temp >= x[0] && temp <= x[1])
            || (temp >= x[1] && temp <= x[0])) &&
            ((temp <= pt2[0] && temp >= pt3[0])
            || (temp >= pt2[0] && temp <= pt3[0])))
            {
            checkpoints[2][0] = temp;
            if (isinf (m_32))
            {
                checkpoints[2][1] =
                m * wrap_subtract (temp, x[0]) + y[0];
            }
            else
            {
                checkpoints[2][1] =
                m_32 * wrap_subtract (temp,
                pt2[0]) +
                pt2[1];
            }
        }
        ang[0] = internal_angle (checkpoints);
        if (ang[0] * barrier[2] > 0)
            {
            if (fabs (ang[0]) < fabs (barrier[2]))
            {
                barrier[2] = ang[0];
                barrier[0] =
            }
wrap_subtract (checkpoints[2][0],
    pt1[0]);
barrier[1] =
wrap_subtract (checkpoints[2][1],
    pt1[1]);
*obstacle = that;
// printf("Jammed body 3 t=%d\t
// Contact at (%f,%f)\t,
// Should move (%f,%f), Angle=
// %f\n",,
// t,
// checkpoints[2][0],checkpoints[2][1],
// barrier[0],barrier[1],
// barrier[2]);
}
}
}
}

else
{
// printf("Overrunning t=%d\n",t);
}

}
}
}

B.2 agar.h

/*****************************************************************
* agar.h
* Mon Mar 27 17:25:43 2006
* Copyright 2006 User
* Email
*****************************************************************/
#ifndef GSLRNG
#define GSLRNG
#include <gsl/gsl_rng.h>
#endif
struct agar
{


// have a width= 2(the maximum length of the cell).
//      double center[2];       //x=0 y=1
struct cell *cells[1024];
//      double nutrient;
//      double slime[100][100];
//      double slime_angle[100][100];
      int num_here;
}

extern void remove_cell (struct cell *thisun);
extern void add_cell (struct cell *thisun);
extern void add_slime (struct cell *thisun);
extern double slime_dir (struct cell *thisun);
extern int fibril_dens (double attachpt[2], struct cell *thisun,
      double frontpt[2], int t);
extern void check_triangle (double pt1[2], double pt2[2],
      double pt3[2], double barrier[3],
      struct cell **obstacle, int t);

B.3  cell.c

#include "cell.h"
#include "defs.h"
#include "functions.h"
#include "agar.h"
#include <gsl/gsl_math.h>
#include <stdio.h>
#ifndef GSLRNG
#define GSLRNG
#include <gsl/gsl_rng.h>
#endif
#include <math.h>
#include <gsl/gsl_randist.h>

extern struct agar *surface[DIMENSION][DIMENSION];

double
check_move (struct cell *thisun, double mv_dist)
{
      int min_x, max_x, min_y, max_y;
      double dx, dy, x1, x2;
      double pt1[2], pt2[2];
      double sina, cosa, tana, tanb;
      int i, j, k, imod, jmod;
      double temp, x, y;
      struct cell *that;
      if (mv_dist == 0)
{  
    return (0);
  }

sina = sin (thisun->angle);
cosa = cos (thisun->angle);
if (mv_dist > 0)
  {
    pt1[0] = thisun->position[0] + thisun->length * cosa;
    pt2[0] = pt1[0] + mv_dist * cosa;
  }
else if (mv_dist < 0)
  {
    pt1[0] = thisun->position[0];
    pt1[1] = thisun->position[1];
    pt2[0] = pt1[0] + mv_dist * cosa;
  }
dx = wrap_subtract (pt2[0], pt1[0]);

min_x =
  mod ((int) (min (pt1[0], pt1[0] + dx) - (2 * INIT_LENGTH)),
  DIMENSION);
max_x =
  mod ((int) (max (pt1[0], pt1[0] + dx) + (2 * INIT_LENGTH)) + 1,
  DIMENSION);
// +1 is to check highest possible spot in for loop

dy = wrap_subtract (pt2[1], pt1[1]);

min_y =
  mod ((int) (min (pt1[1], pt1[1] + dx) - (2 * INIT_LENGTH)),
  DIMENSION);
max_y =
  mod ((int) (max (pt1[1], pt1[1] + dx) + (2 * INIT_LENGTH)) + 1,
  DIMENSION);

tana = sina / cosa;

for (i = min_x; mod (i, DIMENSION) != max_x; i++)
  {
    imod = mod (i, DIMENSION);
    for (j = min_y; mod (j, DIMENSION) != max_y; j++)
{    jmod = mod (j, DIMENSION);
    for (k = 0; k < (surface[imod][jmod])->num_here; k++)
    {
        that = surface[imod][jmod]->cells[k];
        if (that != thisun)
        {
            x1 = that->position[0];
            x2 = x1 + that->length * cos (that->angle);
            if (sqrt (pow (wrap_subtract (x1, pt1[0]), 2) +
                   pow (wrap_subtract (that->position[1], pt1[1]),
                        2)) <
                thisun->length + that->length + fabs (mv_dist))
            {
                tanb = tan (that->angle);
                x = pt1[0] + (-tanb * dx + dy) / (tana - tanb);
                if (((x >= pt1[0] && x <= pt2[0]) ||
                     (x >= pt1[0] && x <= pt2[0])) &&
                     ((x <= x1 && x >= x2) || (x >= x1 && x <= x2)))
                {
                    y = tana * (x - pt1[0]) + pt1[1];
                    temp =
                        sqrt (pow (wrap_subtract (x, pt1[0]), 2) +
                               pow (wrap_subtract (y, pt1[1]), 2));
                    if (temp < fabs (mv_dist))
                    {
                        if (mv_dist > 0)
                        {
                            mv_dist = temp;
                        }
                        else
                        {
                            mv_dist = -temp;
                        }
                    }
                }
            }
        }
    }
}

return (mv_dist);
void move_cell (struct cell *thisun, double mv_dist, int t)
{
    double oldx, oldy, cosa;
    if (mv_dist != 0)
    {
        oldx = thisun->position[0];
        oldy = thisun->position[1];
        cosa = cos (thisun->angle);
        if ((oldx + cosa * mv_dist > WALL) || cosa > 0)
        {
            // thisun->position[0] += cos (thisun->angle) * mv_dist;
            thisun->position[0] =
            fmod (oldx + cosa * mv_dist, DIMENSION);
            // thisun->position[1] += sin (thisun->angle) * mv_dist;
            thisun->position[1] =
            fmod (oldy + sin (thisun->angle) * mv_dist, DIMENSION);
            // printf("t=%d Moving a cell from (%f,%f) %f units to
            // (%f,%f) \n",
            // t,oldx,oldy,mv_dist, thisun->position[0],
            // thisun->position[1]);
            if (isnan (thisun->position[0]))
            {
                printf ("Something Broke");
            }
        }
        if (((int) oldx) -
            ((int) thisun->position[0]) +
            ((int) oldy) - ((int) thisun->position[1]) != 0)
        {
            remove_cell (thisun);
            add_cell (thisun);
        }
    }
}

void cell_init (struct cell
    *thisun, double x, double y, double angle,
    double length, int t)
{
    thisun->angle = angle;
    thisun->position[0] = x;
    thisun->position[1] = y;
    thisun->length = length;
}
// thisun->celldist = CELLWIDTH * length;
// thisun->sensdist = SENSDIST * length;
thisun->next = NULL;
thisun->clock = t;
add_cell (thisun);
}

void reverse (struct cell *thisun, int t, gsl_rng * r)
{
    double x, y;
    if (thisun->clock <= t)
    {
        x = thisun->position[0] + cos (thisun->angle) * thisun->length;
        y = thisun->position[1] + sin (thisun->angle) * thisun->length;
        thisun->position[0] = x;
        thisun->position[1] = y;
        if (isnan (thisun->position[0]))
        {
            printf ("Something Broke");
        }
        thisun->angle =
        fmod (thisun->angle + 2.0 * M_PI, 2.0 * M_PI) - M_PI;
        thisun->clock +=
        gsl_ran_binomial (r, REVERSE_PROB,
            MAX_REVERSE_DEV) + AVE_REVERSE -
            MAX_REVERSE_DEV * REVERSE_PROB;
    }
}

void S_move (struct cell *thisun, int t, gsl_rng * r)
{
    // int i;
    double theta, radius, dist, dists[3];
    double barrier[3];
    double checkpoints[3][2];
    struct cell *obstacle;
    double v[2], u[2];

    checkpoints[0][0] = fmod (thisun->position[0] +
        (thisun->length) * cos (thisun->angle), DIMENSION); // front
    // of
    // cell
    checkpoints[0][1] = fmod (thisun->position[1] +
    // of
    // cell
}
(thisun->length) * sin (thisun->angle),
DIMENSION);
checkpoints[1][0] = thisun->position[0]; // tail of cell
checkpoints[1][1] = thisun->position[1];
theta = PILI_ANG * gsl_rng_uniform (r) - PILI_ANG / 2.0 + thisun->angle;
radius = PILI_LENGTH * sqrt (gsl_rng_uniform (r));
// want to move the front of the cell towards
// (front[0]+cos(theta)*radius, front[1]+sin(theta)*radius)
checkpoints[2][0] =
  fmod (checkpoints[0][0] + cos (theta) * radius, DIMENSION);
// pili attachment site
checkpoints[2][1] =
  fmod (checkpoints[0][1] + sin (theta) * radius, DIMENSION);

if (fibril_dens (checkpoints[2], thisun, checkpoints[0], t))
{
  barrier[0] =
    wrap_subtract (checkpoints[2][0], checkpoints[1][0]);
  barrier[1] =
    wrap_subtract (checkpoints[2][1], checkpoints[1][1]);
  barrier[2] = internal_angle (checkpoints);
  obstacle = thisun;
  check_triangle (checkpoints[1], checkpoints[0], checkpoints[2],
                 barrier, &obstacle, t);
  thisun->angle = thisun->angle + barrier[2];
  thisun->angle = fmod (thisun->angle + M_PI, 2 * M_PI) - M_PI;
  dist = sqrt (dot (barrier, barrier)) - thisun->length;

  if (obstacle != thisun)
  {
    move_cell (thisun, COLLISION_FACTOR * dist, t);
    dists[0] = dist_pt_line (thisun->position[0] +
      cos (thisun->angle) * 
    thisun->length,
    thisun->position[1] +
      sin (thisun->angle) * 
    thisun->length, obstacle);
    dists[1] =
      dist_pt_line (obstacle->position[0] +
        cos (obstacle->angle) * obstacle->length, 
        obstacle->position[1] +
        sin (obstacle->angle) * obstacle->length,

thisun);
dists[2] = dist_pt_line (obstacle->position[0], obstacle->position[1],
    thisun);
    if (dists[1] < END_LENGTH * obstacle->length
    || dists[2] < END_LENGTH * obstacle->length)
    {
    // move obstacle along axis
    v[0] =
        wrap_subtract (checkpoints[2][0],
            thisun->position[0] +
            cos (thisun->angle) * thisun->length);
    // vector from point of contact to attachment site
    v[1] =
        wrap_subtract (checkpoints[2][1],
            thisun->position[1] +
            sin (thisun->angle) * thisun->length);
    u[0] = cos (obstacle->angle); // vector for the cell
    u[1] = sin (obstacle->angle);
    dist = PUSH_RATIO * dot (v, u);
    }
    dist = check_move (obstacle, dist);
    move_cell (obstacle, dist, t);
    // move thisun
    if (dist > 0 && dists[0] < TOLER)
    {
    checkpoints[0][0] = thisun->position[0] +
        cos (thisun->angle) * thisun->length;
    checkpoints[0][1] = thisun->position[1] +
        sin (thisun->angle) * thisun->length;
    checkpoints[1][0] = thisun->position[0];
    checkpoints[1][1] = thisun->position[1];
    checkpoints[2][0] = checkpoints[0][0] + dist * u[0];
    checkpoints[2][1] = checkpoints[0][1] + dist * u[1];
    barrier[2] = internal_angle (checkpoints);
    barrier[0] =
        wrap_subtract (checkpoints[2][0], checkpoints[1][0]);
    barrier[1] =
        wrap_subtract (checkpoints[2][1], checkpoints[1][1]);
    check_triangle (checkpoints[1], checkpoints[0],
        checkpoints[2], barrier, &obstacle, t);
    thisun->angle = thisun->angle + barrier[2];
    thisun->angle =
        fmod (thisun->angle + M_PI, 2 * M_PI) - M_PI;
    dist = sqrt (dot (barrier, barrier)) - thisun->length;
    }
move_cell (thisun, dist, t);
}
}
else
{
    move_cell (thisun, dist, t);
}
}

B.4  cell.h

/**************************************************************************
 * cell.h
 *
 * Mon Mar 27 17:23:46 2006
 * Copyright 2006 User
 * Email
 **************************************************************************/
 ifndef GSLRNG
 define GSLRNG
 include <gsl/gsl_rng.h>
 endif
 struct cell
 {
 //the cells, they have direction, position, and length.
 // The position is at the back of the cell.
 double position[2];
 double angle;
 double length;
 struct cell *next;
 struct agar *here;
 int clock;
);
 extern double check_move (struct cell *thisun, double mv_dist);
 extern void move_cell (struct cell *thisun, double mv_dist, int t);
 extern void cell_init (struct cell *thisun, double x, double y,
 double angle, double length, int t);
 extern void reverse (struct cell *thisun, int t, gsl_rng * r);
 extern void S_move (struct cell *thisun, int t, gsl_rng * r);

B.5  defs.h

**************************************************************************
 *  defs.h
**************************************************************************
#ifndef GSLRNG
#define GSLRNG
#include <gsl/gsl_rng.h>
#endif

#define NUM_CELLS 100
//number of cells to start with
#define TIME_STEPS 100
//number of times to run
#define PRINT_STEP 1
#define INNER_STEP 1 // number of times to on average
// update each cell per TS
#define DIMENSION 20
//width of feeding domain
#define RADIUS 3
#define PEN_WIDTH 3
#define PEN_LENGTH 4
#define PEN_FRAC 0
//radius to initialize cells in
#define TOLER 0.00001
#define INIT_LENGTH 0.25
// ~ 4 um
#define REVERSE_PROB 0.5
#define MAX_REVERSE_DEV 10
#define AVE_REVERSE 21
#define TRIG_CUTOFF 0.2
#define PILI_LENGTH 0.5
// ~ 8 um max length
#define PILI_ANG M_PI/4.0
#define FIBRIL_LENGTH 1.0
// ~ 24 um
#define FIB_CUT 5.0
#define FIB_ANG_CUT 8.0*M_PI/16.0
#define END_LENGTH 0.2
#define WALL (1)
#define COLLISION_FACTOR 0.99
#define OVERRUN_FACTOR 0.5
#define OVERRUN_PROB 0.0
#define PUSH_RATIO 0.01
#define absmint(a,b) (fabs(a) <= fabs(b) ? (a) : (b))
//returns value closest to zero

162
```c
#define min(a,b) ((a) <= (b) ? (a) : (b))
#define max(a,b) ((a) >= (b) ? (a) : (b))

#include <gsl/gsl_math.h>
#include <gsl/gsl_rng.h>
#include <math.h>

int mod (int i, int md) {
    if (i >= md) {
        while (i >= md) {
            i -= md;
        }
    }
    if (i < 0) {
        while (i < 0) {
            i += md;
        }
    }
    return (i);
}

double fmod (double val, double md) {
    if (val >= md) {
        while (val >= md)
```
{ val -= md; }

} if (val < 0) {
  while (val < 0) {
    val += md;
  }
  if (val == md) {
    val = 0;
  }
}
return (val);

double wrap_subtract (double x, double y) {
  double ret;
  ret = x - y;
  if (fabs (ret) > DIMENSION - 2) {
    if (ret < 0) {
      ret += DIMENSION;
    } else {
      ret -= DIMENSION;
    }
  }
  if (isnan (ret)) {
    printf ("Something Broke");
  }
  return (ret);
}

double dist_pt_line (double x, double y, struct cell *thisun) {
  double endx, otherendx, endy, angle;
double x_star, cosa, sina, tana, d;
angle = thisun->angle;
endx = thisun->position[0];
endy = thisun->position[1];
if (angle != M_PI_2 || angle != 3 * M_PI_2)
{
  cosa = cos (angle);
sina = sin (angle);
otherendx = thisun->position[0] + thisun->length * cosa;
tana = tan (angle);
x_star =
  sina * sina * endx + sina * cosa * (y - endy) +
  cosa * cosa * x;
if (endx < otherendx)
  {
    if (x_star < endx)
    {
      return (sqrt (pow (endx - x, 2) + pow (endy - y, 2)));
    }
    else if (x_star > otherendx)
    {
      return (sqrt (pow (x - (otherendx), 2) +
                    pow (y - (endy + sina * thisun->length),
                         2)));
    }
    else
    {
      d = sqrt
      (pow (x - x_star, 2) +
       pow (y - (endy + (x_star - endx) * tana), 2));
      return (d);
    }
  }
else
  {
    if (x_star > endx)
    {
      return (sqrt (pow (endx - x, 2) + pow (endy - y, 2)));
    }
    else if (x_star < otherendx)
    {
      return (sqrt (pow (x - (otherendx), 2) +
                    pow (y - (endy + sina * thisun->length),
                         2)));
    }
  }
else
{
d = sqrt(pow(x - x_star, 2) +
    pow(y - (endy + (x_star - endx) * tana), 2));
return (d);
}
}
else
{
    if (angle == M_PI_2)
    {
        if (y > endy)
{
            if (y < endy + thisun->length)
            {
                return (abs(x - endx));
            }
        else
        {
            return (sqrt(pow(x - endx, 2) +
                pow(y - endy - thisun->length, 2)));
        }
    }
    else
{
        return (sqrt(pow(x - endx, 2) + pow(y - endy, 2)));
    }
}
else
{
    if (y < endy)
{
        if (y > endy - thisun->length)
        {
            return (abs(x - endx));
        }
    else
    {
        return (sqrt(pow(x - endx, 2) +
            pow(y - endy - thisun->length, 2)));
    }
}
else
{
    return (sqrt (pow (x - endx, 2) + pow (y - endy, 2)));
}
}

double internal_angle (double pt[3][2])
{
    // Find the angle from segment pt[1]->pt[0] to segment
    // pt[1]->pt[2]
    double angle[2];
    double ret;

    angle[0] = atan2 (wrap_subtract (pt[2][1], pt[1][1]),
                     wrap_subtract (pt[2][0], pt[1][0]));
    angle[1] = atan2 (wrap_subtract (pt[0][1], pt[1][1]),
                     wrap_subtract (pt[0][0], pt[1][0]));
    ret = angle[0] - angle[1];
    ret = fmod (ret + M_PI, 2.0 * M_PI);
    ret = ret - M_PI;
    return (ret);
}

double dot (double v1[2], double v2[2])
{
    return (v1[0] * v2[0] + v1[1] * v2[1]);
}

B.7 functions.h

/**********************************************
 * functions.h
 *
 * Mon Mar 27 17:26:38 2006
 * Copyright 2006 User
 * Email
 ***********************************************/
extern int mod (int i, int md);
extern double fmod (double val, double md);
extern double wrap_subtract (double x, double y);
extern double dist_pt_line (double x, double y,
                            struct cell *thisun);
extern double internal_angle (double pt[3][2]);
extern double dot (double v1[2], double v2[2]);

B.8 main.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <memory.h>
#include <gsl/gsl_math.h>
#ifndef GSLRNG
#define GSLRNG
#include <gsl/gsl_rng.h>
#endif
#include <gsl/gsl_randist.h>
#include <bzlib.h>
#include "cell.h"
#include "agar.h"
#include "functions.h"
#include "defs.h"

gsl_rng *r;
extern struct agar *surface[DIMENSION][DIMENSION];
struct cell *first, *last;
int numcells;

void
output_agar (BZFILE * of)
{
  int i, j;
  int bzerror;

  for (i = 0; i < DIMENSION; i++)
    {
      for (j = 0; j < DIMENSION; j++)
        {
          BZ2_bzWrite (&bzerror, of, surface[i][j],
                        sizeof (struct agar));
        }
    }
}
void
output_cells (struct cell *begin, int t, BZFILE * of)
{
    struct cell *thisun;
    int i, bzerror;

    thisun = begin;

    for (i = 0; i < numcells; i++)
    {
        BZ2_bzWrite (&bzerror, of, thisun, sizeof (struct cell));
        thisun = thisun->next;
    }
}

int
main (int argc, char *argv[])
{
    int t;
    int i, j, now, test;
    double x, y, dir;
    struct cell *thisun;
    char ofn[32];
    FILE *outfile;
    BZFILE *bzfile;
    int bzerror;
    unsigned int bzin, bzout;
    const gsl_rng_type *T;
    gsl_rng_env_setup ();
    T = gsl_rng_default;
    r = gsl_rng_alloc (T);
    gsl_rng_set (r, INT_MAX * pow (cos (time (NULL)), 2));

    for (i = 0; i < DIMENSION; i++)
    {
        for (j = 0; j < DIMENSION; j++)
        {
            surface[i][j] = malloc (sizeof (struct agar));
            surface[i][j]->num_here = 0;
        }
    }
    first = malloc (sizeof (struct cell));
    thisun = first;
    i = 1;
while (i)
{
    x = 2 * gsl_rng_uniform (r);
    y = 2 * gsl_rng_uniform (r);
    if (x * x + y * y <= 1)
        i--;
}

x = RADIUS * x;
y = RADIUS * y;
t = 0;

cell_init (thisun, /* DIMENSION / 2.0 + */ x,
    /* DIMENSION / 2.0 + */ y,
    2 * M_PI * gsl_rng_uniform (r) - M_PI,
    INIT_LENGTH + INIT_LENGTH * gsl_rng_uniform (r),
    gsl_rng_uniform_int (r, MAX_REVERSE_DEV) + 1);

for (i = 1; i < NUM_CELLS; i++)
{
    if (gsl_rng_uniform (r) < PEN_FRAC) // not in peninsula
    {
        x = fmod (WALL + RADIUS * gsl_rng_uniform (r), DIMENSION);
        y = fmod (/* RADIUS */ DIMENSION * gsl_rng_uniform (r)
    /* +(DIMENSION-RADIUS)/2.0 */ , DIMENSION);
        dir = (gsl_rng_uniform (r)) * 2 * M_PI;
    }
    else
    {
        test = 1;
        while (test)
        {
            x = fmod ((PEN_LENGTH - INIT_LENGTH) *
gsl_rng_uniform (r) + WALL + RADIUS,
            DIMENSION);
            y =
            fmod (PEN_WIDTH * (gsl_rng_uniform (r) - 1.0 / 2.0) +
            DIMENSION / 2.0, DIMENSION);
            if (fabs ((y - DIMENSION / 2.0)) /
                (((double) (PEN_LENGTH + WALL + RADIUS)) - x) <=
                ((double) PEN_WIDTH) / (2.0 * PEN_LENGTH)
                && (((double) (PEN_LENGTH + WALL + RADIUS)) -
                x) / fabs ((y - DIMENSION / 2.0)) >=
                (2.0 * PEN_LENGTH) / ((double) PEN_WIDTH))
            {
                test--;
            }
        }
    }
dir =
atan (-y - DIMENSION / 2.0) /
((PEN_LENGTH + WALL + RADIUS) - x)) +
gsl_rng_uniform_int (r, 2) * M_PI;
}
thisun->next = malloc (sizeof (struct cell));
thisun = thisun->next;
cell_init (thisun, x, //DIMENSION / 2.0 + x * cos (y),
y, //DIMENSION / 2.0 + x * sin (y),
dir, //2 * M_PI * gsl_rng_uniform
// (r)-M_PI, //0
INIT_LENGTH + INIT_LENGTH * gsl_rng_uniform (r),
gsl_rng_uniform_int (r, (AVE_REVERSE)));
}
last = thisun;
numcells = NUM CELLS;
sprintf (ofn, "SPPcells%d.dat.bz2", -1);
outfile = fopen (ofn, "w");
bzfile = BZ2_bzWriteOpen (&bzerror, outfile, 9, 0, 0);
output_cells (first, -1, bzfile);
BZ2_bzWriteClose (&bzerror, bzfile, 0, &bzin, &bzout);
fclose (outfile);
sprintf (ofn, "SPPagar%d.dat.bz2", -1);
outfile = fopen (ofn, "w");
bzfile = BZ2_bzWriteOpen (&bzerror, outfile, 9, 0, 0);
output_agar (bzfile);
BZ2_bzWriteClose (&bzerror, bzfile, 0, &bzin, &bzout);
fclose (outfile);
for (t = 0; t < TIME_STEPS; t++)
{
    for (i = 0; i < INNER_STEP; i++)
    {
        for (j = 0; j < NUM CELLS; j++)
        {
            now = gsl_rng_uniform_int (r, numcells + 1);
            thisun = first;
            while (now != 0 && thisun->next != NULL)
            {
                thisun = thisun->next;
                now--;
            }
            //align (thisun);
            S_move (thisun, t * INNER_STEP + i, r);
            reverse (thisun, t * INNER_STEP + i, r);
        }
    }
}
//

grow (thisun);
}

if (t \% PRINT_STEP == 0)
{
    sprintf (ofn, "SPPcells%d.dat.bz2", t);
    outfile = fopen (ofn, "w");
    bzfile = BZ2_bzWriteOpen (&bzerror, outfile, 9, 0, 0);
    output_cells (first, t, bzfile);
    BZ2_bzWriteClose (&bzerror, bzfile, 0, &bzin, &bzout);
    fclose (outfile);
    sprintf (ofn, "SPPagar%d.dat.bz2", t);
    outfile = fopen (ofn, "w");
    bzfile = BZ2_bzWriteOpen (&bzerror, outfile, 9, 0, 0);
    output_agar (bzfile);
    BZ2_bzWriteClose (&bzerror, bzfile, 0, &bzin, &bzout);
    fclose (outfile);
}

return (0);


