A DISCRETE, STOCHASTIC MODEL AND CORRECTION METHOD FOR BACTERIAL SOURCE TRACKING

By

MARK DANIEL LEACH

A thesis submitted in partial fulfillment of
the requirement for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

May 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of MARK DANIEL LEACH find it satisfactory and recommend that it be accepted.

_____

Chair

_____

_____

ACKNOWLEDGMENT

This project was funded by USDA NRI contract 2002-35102-12374 and by the Agricultural Animal Health Program at the College of Veterinary Medicine, Washington State University, Pullman, WA.

A DISCRETE, STOCHASTIC MODEL AND CORRECTION METHOD

FOR BACTERIAL SOURCE TRACKING

Abstract

by Mark Daniel Leach, M.S.
Washington State University
May 2007

Chair: Shira L. Broschat

We have developed a model to test several underlying assumptions of bacterial source tracking (BST) when the BST method is based on detection of discrete genetic markers from source-specific bacteria. The model consists of an environment and discrete-time input signals that represent sources of contamination partitioned into marker-bearing and non-marker-bearing units "shed" into the environment. Simulations run for different types of environmental contamination patterns indicate that if hosts shed different percentages of BST markers, the environment cannot be accurately characterized unless a correction method is used. The correction method, which requires the solution to a linear system, reduces the mean error in estimating the proportions of host contamination to below 3%. The effectiveness of the method depends on accurate knowledge of the occurrence and prevalence of markers in the various hosts; this may be a challenging task, especially if these values vary across populations in space and time. In addition, the correction method does not compensate for environments with low-density or unmixed contamination. In conclusion, our simulations highlight several fundamental challenges that may prevent absolute quantification of fecal input using discrete marker BST.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

Fecal contamination is the chief contributor of pathogens in surface water and the identification and mitigation of sources of contamination is an on-going problem. Bacterial source tracking (BST) refers to techniques for identifying sources of fecal contamination, distinguishing between host origin, and assigning proportions to individual sources. BST techniques utilize bacterial indicators that are not themselves pathogenic, but which imply the presence of pathogenic bacteria or viruses. Recent efforts have focused on library-independent BST methods that identify genetic markers for indicator bacteria specific to particular hosts [15]. In theory, the occurrence of these discrete markers should be constant through time and over a wide geographic area, but questions regarding the distribution of the markers among hosts and the validity of field applications remain unanswered.

DNA markers specific to human and non-human sources have been identified [3, 4, 8, 9, 11, 12, 14]. A number of techniques have been used to identify these markers including the screening of fecal samples for specific 16S rRNA sequences [1, 2, 10], using mixed-genome microarrays to identify bacterial genomic DNA markers [14], and employing suppression subtraction hybridization libraries to identify unique markers [8]. Polymerase chain reaction (PCR) is then employed to detect markers in fecal and environmental samples. It is unclear, however, how many isolates need to be screened, as well as from how many hosts and from what geographic regions, in order to adequately characterize (validate) the markers. Our experience in screening markers suggests that they may show adequate host-specificity in PCR screening at the fecal sample testing stage, but they may be unable to distinguish between hosts in field studies because of ubiquitous distribution in water samples (D.R. Call, unpublished data). While this may be due to differences in the survival of marker strains in aqueous environments, the point is that the distribution of the marker is not predictable based on screening isolates in the laboratory. Even if accurate quantitative methods are available, another problem is that each marker may represent different masses of fecal pollution; without *a priori* knowledge of this representational bias, it will be difficult to attribute volume of waste according to marker abundance. Further complications include our lack of knowledge about the relative health risks posed by fecal contamination from different hosts and recent evidence that

BST markers correlate poorly with fecal indicators [10].

The measurement of markers in the field is closely related to the problem of marker characterization above. We need to establish criteria for interpreting results based on marker prevalence in samples from known sources. That is, assuming our marker characterization is correct, we need to know how to interpret measurements when there is non-specific host occurrence or differences in the prevalence between hosts. For instance, a small non-specific percentage of marker may lead to ambiguous results if the non-specific host is a large contributor to the waste stream. Nevertheless, if we know the relative prevalence of the different markers among hosts and we can make a quantitative measurement of the sample (e.g., with quantitative PCR), we should be able to correct the sampled data to estimate the actual host contributions in the environment.

In this paper we develop a model for addressing some of the issues that arise with the use of discrete genetic markers from source-specific bacteria. A primary motivation for this effort is that there are no gold standards by which to assess the interpretation of BST marker data from field samples. Models provide us with a means to compare estimates with known distributions. In particular, we are interested in a model that will represent significant statistical variation with regard to the distribution of bacteria in an environment, the difference in prevalence among markers, the non-specific occurrence of markers, and the effects of differences in survival between marker-bearing bacteria. Our goal was not to develop a model of the dynamics of bacterial growth and survival in aqueous environments, but to derive simplified scenarios that can be used to locate primary problems in the BST methodology that need further attention. If problems arise in the simplified case, then we assume they will also be present in real-world applications.

2 Model Development

We modeled the BST system as a multi-input, multi-output system where discrete-time input signals represent fecal contamination coming into an environment, and the output signals represent statistics derived from the measurement of markers in a sample space. Fecal contamination is par-

titioned into abstract units that act as cellular automata. For each time step the internal state of the environment evolves according to simple rules that govern the automata. Some of the rules define random processes and this introduces variation that simulates the uncertainty in actual environments.

Environment. The environment is an abstract representation of a waterway. Our approach is to create an environmental model that is as simple as possible but still able to simulate significant issues of variation in the process of BST. In the sampling process, we test for the presence of markers in a waterway such as a stream, river, or lake (see below). The essential character of the environment is a one-dimensional surface (such as a riverbank or stream segment) where points along the surface correspond to variations in the concentration of fecal contamination. Accordingly, our model consists of a one-dimensional array of bins. The sources of contamination are distributed along the array and each source sheds contamination into proximal bins. A sample space for measurement of the environment consists of a subset of the total bins.

Hosts. The model has five different host species arbitrarily designated cow, dog, elk, goose, and human. We used these names from previous studies [8, 14] instead of generic host names, but the names do not reflect any biological parameters in the simulation. Each host type has the same range of possible parameter values.

In actual environments sources of fecal contamination can be classified as point and non-point. Point sources include agriculture and human, such as runoff from pastures or sewer overflows. Non-point sources can include wildlife, such as elk or waterfowl, where the entry point of contamination is not readily apparent. Each source in the model sheds contamination of a single host type and at a rate assigned to that host. Each source also has a location along the environment bin array. Non-point sources are modeled by distributing contamination in a uniform distribution over the entire environment, whereas point sources are modeled by distributing the contamination in a normal distribution with the mean at the source location. Each source is a constant, discrete signal $S_i(k) = K_i$ where the constant K_i is determined by the host type.

Cells. In an aqueous environment the bacterial cells are small and numerous enough to consider

the level of contamination a continuous variable. In the simulation, however, a continuous variable presents a problem in terms of mixing and sampling. We need a method of sampling a "chunk" of the environment that contains random levels and types of contamination from the various hosts. By dividing the contamination into discrete units and randomly distributing them into the environment, we can simulate the mixing of bacterial contamination without specifying the environment as a whole. Thus, our approach is to specify only the behavior and traits of the discrete units (cells). We do not define the distribution of cells in the environment directly, but allow it to evolve over time through the action of a large number of these cellular automata.

Based on factors such as sunlight, temperature, and species, bacteria can show considerable variation in their survival and persistence in aqueous environments [5, 6, 7, 13]. In the model, we assume that the bacteria do not persist or replicate in the environment, but begin to die off as soon as shed from a source. We model cell decay according to the formula $Ne^{-\alpha k}$ where N is the number of cells at time 0, and k is the time step. Each time step represents approximately one hour. A random lifespan is assigned to each cell from an exponential distribution and at each time step the cell ages until its age equals its lifespan; then the cell dies and is removed from the environment. From the perspective of the environment as a whole this constitutes exponential decay of cells over time. Each cell is also marked according to what type of host it came from and what type, if any, of marker it carries. This allows the measurement of cells in the environment including counts of the whole environment and counts within a particular sample space.

Marker Matrix. In a best-case scenario BST markers would be 100% specific to their assigned hosts, but in the real world there may be non-specific occurrence because of transient carriage or incomplete host-specificity. In our model, samples from known hosts are hypothetically tested against a group of markers (e.g., cow, dog, elk, goose, human) and the proportion of markers is determined for the host. For instance, analysis of cow feces may indicate this host produces indicator bacteria where 50% harbor a cow marker, 10% harbor a goose marker and the remaining 40% harbor no recognized marker. This data can be represented in a column vector as (0.5, 0.0, 0.0, 0.1, 0.0)^T where the elements represent the prevalence of a particular marker. When contamination

from each host is characterized, the column vectors can be combined into a matrix representing the prevalence data. In the best case of perfect specificity, the marker matrix $\mathbf{M}$ can be indicated as the product of the identity matrix and some constant. For instance, if each marker occurs in 70% of host contamination, then $\mathbf{M} = 0.7 \cdot \mathbf{I}$.

This method of describing the distribution of markers may not be practical. We may be able to determine the relative percentages of marker in isolates (e.g., through quantitative PCR), but not know the true proportion of bacteria that harbors no marker. In addition, we may not know the amount of marker produced per unit weight of fecal contamination. Our model does not address these ambiguities, but assumes an ideal marker matrix where the proportion of "no marker" is known and the proportions represent the total contamination entering the environment.

Sampling. Two methods are used to estimate the contribution of hosts. In the first method, we randomly select a group of bins and count all of the marker-bearing cells in those bins. This method provides an exact quantification of the markers in the sample space. We can then convert the total counts for each host type to a percentage. After applying this same procedure to the entire environment, counting both marker-bearing and non-marker-bearing cells together, we can compare the percentages of marker-bearing cells in the sample to the total cells in the environment. For example, if each host is equally represented in the environment and each source has the same shedding and decay rates, the percentage of cells in the environment are similar for each host with some variation (Table 1). In practice, however, we can only measure the marker-bearing cells in the sample space and this can create significant error in the estimates of proportional contributions of each host (Table 1). The second method of estimating host contributions is to perform a presence/absence measurement of the bins in the sample space. In this case, we register a true or false response for each host per bin and then calculate the percentage of positive bins in the sample space for each host.

TABLE 1. Environment statistics. Statistics derive from quantitative measurement of an environment sampled after 300 time steps and show cell counts, proportional volume of cells, and percent error between environment as a whole and the sample space.

	Cells in the environment				
	Cow	Dog	Elk	Goose	Human
Total cell count:	3502	3812	3618	3722	3666
Percentage total cells:	19.1%	20.8%	19.7%	20.3%	20.0%
	Cells in sampled bins				
	Cow	Dog	Elk	Goose	Human
Marker-bearing cell count:	125	195	76	188	163
Percentage marker-bearing cells:	16.7%	26.1%	10.2%	25.2%	21.8%
Percent error:	-12.6%	25.5%	-48.2%	24.1%	9.0%

We also use two types of sample spaces, contiguous and uniform. In the contiguous case the sample space consists of a block of contiguous bins selected at a uniform random location within the environment with the restriction that the position of the block fits within the environment. This simulates a sampling scheme whereby water samples are collected near each other. In the uniform case each water sample (bin) is selected from a uniform random location within the model environment.

Environment Characterization. For different sets of model parameters the environment evolves in a distinct manner. For instance, if the environment is small and there are a large number of cells shed per time step, the environment will "fill up" and reach an equilibrium with a large number of cells in each bin. If the cells are shed in a uniform distribution, then this type of environment will be dense and well-mixed. Each bin is likely to contain cells of each host type and the standard deviation of cells per bin will be relatively small compared to the average. Alternatively, if the environment is large or there are few cells shed per time step, the system will evolve to a sparse equilibrium state. There may be empty bins and bins that lack cells or markers for a particular host. When the cells are shed in a Gaussian distribution about the sources, the variation is increased, particularly for sparse environments.

In order to characterize the density and mixing of the environment in one parameter, the model counts the cells in all bins and calculates the mean and standard deviation for each host. The percent coefficient of variation (COV) for each host ($\sigma/\mu \times 100$) serves as a measure of the state

of the total environment. For instance, there may be an average of 80 cow cells (both marker- and non-marker-bearing) per bin with a standard deviation of 10 cells, yielding a COV of 12.5%. The COV indicates the density of pollution in the environment because when bins contain large numbers of cells there will be less "spread" in the distribution.

Correction Method. The columns of the marker matrix determine the number of marker-bearing cells in the environment. The relationship between marker prevalence profiles, the proportion of markers in the sample, and the proportional volume of total indicator bacteria can be represented as a system of linear equations. The system of equations can be represented as a matrix equation $\mathbf{M} \cdot \mathbf{x} = \mathbf{b}$, where the goal is to measure $\mathbf{b}$ from an environmental sample and solve for $\mathbf{x}$. We would expect the diagonal elements of the marker matrix to be non-zero (i.e., cows producing cow marker, dog producing dog marker, and so on). This is a sufficient condition for $\det \mathbf{M} \neq 0$ and the existence of a unique solution for the linear system. Once $\mathbf{x}$ is found, we can normalize its elements so that the proportional volumes p_i can be compared to the proportional volumes of total cells in the environment.

$$p_i = \frac{x_i}{\sum_{i=1}^{5} x_i}$$

This normalized solution represents the relative volumes of cells for each host in the sample space. The model compares this solution to the cell volumes in the environment as a whole.

3 Materials and Methods

We implemented the BST model as a simulation designed on a Linux platform using the KDevelop integrated development environment. The simulator is written in ANSI C with the addition of functions in the random number generation, statistical, and linear algebra modules of the Gnu Scientific Library (GSL). The code is platform-independent if GSL is available on the system (source code is available from the authors).

The input to the simulator includes two files. One file contains true random integers (www.random.org) that act as seeds for the GSL random number generator. The simulator loads the seed file, uses some of the seeds during the simulation, and stores the remainder back into the source file. The simulator uses the seeds periodically during the simulation to avoid repetition of random number sequences and to ensure unique outputs from each simulation. The other input file is a configuration file that holds the simulation parameters.

The output of a simulation includes a statistical summary (see results section) and a file that gives an ASCII representation of the environment. The output file shows the contents of each bin, one per line, where the letters c, d, e, g, and h represent cells from the five hosts. This file allows us to assess the shape of the environment through visual inspection. The simulation also includes output functions to measure the COV as a function of time. We used a curve-fitting tool (Logger Pro 3, Vernier Software & Technology, Beaverton, OR) to plot the COV data and determine parameters that describe the curve.

4 Results

We divided simulations into categories based on environment type and sampling method. We first established a "best-case scenario" and then modified the simulation parameters to examine model sensitivity given eight additional scenarios. Our goal was to determine which factors contribute the most uncertainty to interpretation of BST data and, as a result, to enable us to better define where future research should be focused. Each simulation included three sampling methods: presence/absence, quantitative, and corrected (see below).

Environment Equilibrium. To derive a reasonable value for the sampling time, we measured the percent coefficient of variation (see above) of selected environments as they evolved over time. We determined that the typical environment reaches an equilibrium and converges as k^B where B is derived from an empirical plot of the COV vs. time (Figure 1). For example, for the "best-case scenario" (see the next subsection) the COV converges to a small value. The equation $Ak^B + C$

provides the closest fit to this curve (Figure 1) where $A = 221.5$, $B = -0.5232$, $C = 4.398$ and the root mean square error (RMSE) is 1.075. The RMSE for Ak^B was 1.473, indicating a better fit with a non-zero steady-state. The fitted equation also indicates that doubling the simulation time from 300 to 600 will only decrease the COV from 15.6% to 12.2%, and to reduce it to 5.96% requires 13,000 time steps. We did not analyze the COV for every simulation, but assumed 300 time steps would yield a low percent change (0.12% in this example) between time steps.

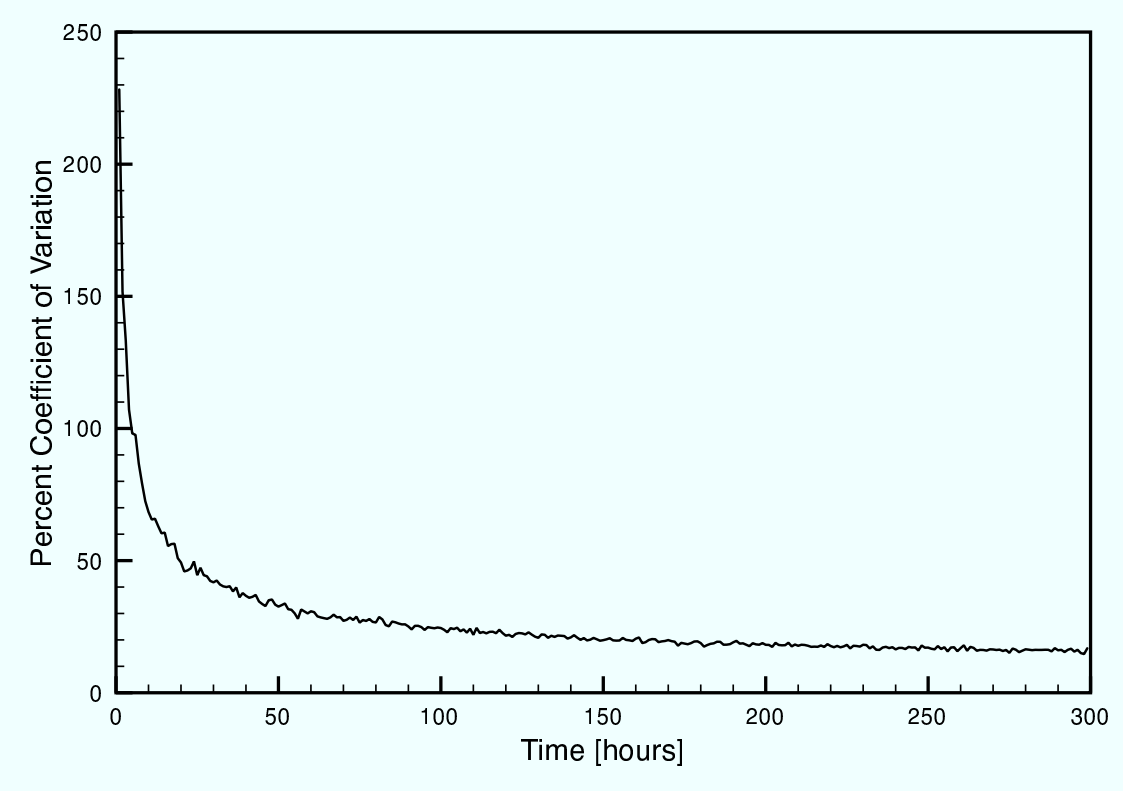

Figure 1: Environmental variance associated with stochastic simulations. The percent coefficient of variation (COV = $\sigma/\mu \times 100$) approaches an equilibrium as a simulation proceeds. This is a typical result where system equilibrium can be approximated with 300 time steps.

Best-case scenario (#1). In the best-case environment each host has the same number of sources and each source sheds the same number of cells per time step. The cells are also distributed uniformly in the environment. These conditions produce well-mixed environments containing approximately equal numbers of cells from each host. In addition, for a best-case scenario, the markers are specific to a single host and occur in the same percentages of isolates from each host

($\mathbf{M} = 0.7 \cdot \mathbf{I}$). Simulation with these parameters (Table 2) produced dense environments with an average of 60 cells per bin. We calculated the mean and standard deviation of the 30 runs and report the average error between the actual volumes of cells in the environment and the volumes measured from the sample space (Tables 3 and A1). Negative values indicate an underestimation and positive values an overestimation of the actual proportional volumes. Both sampling schemes produced $< 1\%$ errors in proportional estimates with standard deviation ranging from 0.5 (presence/absence) to 6.02 (quantitative). The percent coefficient of variation is low for a best-case scenario (COV < 30.0), which is indicative of a dense, well-mixed environment.

TABLE 2. Best-case scenario parameters.

Number of sources per host:	5
Source signal strength:	10 cells per time step
Environment size:	800 bins
Simulation time:	300 steps
Cell distribution:	uniform
Sample space:	30 contiguous bins
Simulation repetitions:	30

Low density scenario (#2). To create a sparse environment, we used the parameters from the best-case scenario but decreased the signal strengths to 1 and increased the environment size to 2000 bins. This is equivalent to less pollution dispersed over a broader area. As expected, a sparse environment generated significantly greater variance and resulted in underestimates for contributions from each host. From a sampling perspective the variance was higher for both methods, but average error was considerably greater (17%) for quantitative sampling compared to presence/absence (1.7%) (Tables 3 and A2).

Unequal cell volume scenario (#3). In the previous example the measure for presence/absence of specific markers performed well because each marker is equally distributed among host bacteria. Because this measurement registers presence for any marker-bearing cells within a bin, it tends to estimate equal representation of hosts in uniform environments. When proportional cell volume is changed (signal strength), however, estimates of presence/absence for different host-specific markers is overestimated for hosts with a lower volume and underestimated for those cases with

a higher volume of cells with the net result being considerable error (100%) (Tables 3 and A3). Unequal cell volume is equivalent to having unequal contribution of fecal contamination between hosts. The quantitative measure performs very well in this scenario (mean error 1.31%) although the standard deviation is elevated compared to the best-case scenario (Table 3).

Unequal marker prevalence (#4). For this scenario we included best-case parameters with 100% host-specificity and an equal volume of cells shed by each host, but the prevalence of markers in the indicator bacteria population was variable (Table 4). This is equivalent to the case when a marker for one host is found frequently in the GI flora whereas a marker for another host is found rarely. This scenario produced considerable error in the quantitative measure (19.8%) while the presence/absence estimates remained quite robust (0.13%) (Tables 3 and A4). Nevertheless, when one or more of the prevalence parameters fall below 10%, the presence/absence estimates begin to show high error rates due to low marker-bearing cell density (data not shown). When quantitative estimates are corrected for the underlying prevalence matrix, the estimated proportions are corrected accordingly (Table 3). That is, when marker prevalence is known, quantitative estimates can be corrected.

TABLE 3. Average percent error and associated standard deviation for nine simulation scenarios. Each scenario was executed 30 times and the values shown here represent the average error between estimates for samples and environments.

	Environmental scenario[b]								
Sampling scheme[a]	1	2	3	4	5	6	7	8	9
Presence/absence average:	0.08	1.71	103.91	0.13	68.22	21.43	5.93	0.25	9.05
Standard deviation:	0.54	25.96	17.12	0.65	6.13	95.66	20.99	0.82	30.85
Quantitative average:	0.50	17.25	1.31	19.81	1.05	19.12	6.28	23.62	27.01
Standard deviation:	6.02	27.36	9.09	6.86	6.50	125.02	31.14	6.12	27.92
Corrected[c] average:	—	—	—	0.66	—	—	—	0.51	—
Standard deviation:	—	—	—	6.92	—	—	—	8.35	—
Average COV:	28.35	141.79	43.66	28.43	35.62	147.87	147.74	28.43	28.45

[a]Three estimation procedures are reported. The presence/absence procedure measures the number of bins that contain at least one marker-bearing cell; the quantitative procedure measures the exact number of marker-bearing cells in each bin; the corrected measurement is derived from applying the correction method to the quantitative data. The COV is the percent coefficient of variation ($\sigma/\mu \times 100$) in the number of cells per bin.
[b]The nine environment scenarios include: 1. Best-case; 2. Low-density environment; 3. Unequal cell volumes; 4. Unequal marker prevalence; 5. Unequal decay rates; 6. Unmixed environment, contiguous sample space; 7. Unmixed environment, uniform sample space; 8. Non-specific markers; 9. Low marker prevalence.
[c]The corrected data are given only for those scenarios that include variation in marker prevalence.

Unequal decay rates (#5). The cells decay (die off) exponentially according to the parameter α in e^{α} where $\alpha \leq 0$. Altering the values of α (via the exponential distribution parameters) had an effect similar to altering the signal strength. The error increased dramatically for presence/absence sampling, causing an overestimation of lower host volumes and an underestimation of higher host volumes, whereas quantitative estimates were very robust (Tables 3 and A5). This scenario might arise if two different species of bacteria with differential survivorship were chosen as fecal markers for two different host populations.

Unmixed environment (#6). To simulate an unmixed environment, we ran the simulation with the cell distribution following a Gaussian pattern with a standard deviation of 20 bins and a sample space of 30 contiguous bins. Sampling in an unmixed environment such as this results in large errors regardless of sampling method because the sample space is likely to occur in an area dominated by a single host (Tables 3 and A6). The correction factor has no effect in this case because the variation is not dependent on the marker matrix. This scenario, while producing unsatisfactory estimates, probably represents a closer approximation to reality than a uniformly mixed model.

We can compensate for the large mean error rate in this scenario by changing from a contiguous to uniform random sampling strategy (scenario #7). By sampling bins from the entire environment, we are compensating for the effect of sampling in a localized area of the environment (contiguous bins), which is dominated by cells from a nearby contamination source. In this case, both presence/absence and quantitative errors were reasonable ($< 6.3\%$), but standard deviations remained high (Tables 3 and A7).

Non-specific markers (#8). Markers are likely to have some level of non-specific occurrence, either due to transient carriage or incomplete specificity (Table 4). In this scenario we introduce non-specific markers at the 10% level; that is, while a marker may be present in 50% of host-specific isolates, it can also be found in up to 10% of isolates from non-specific hosts. For example, the marker matrix can be set so that elk and human markers are found in other hosts (Table 4) and this leads to significant error in the quantitative measure (Tables 3 and A8). The presence/absence estimation remains close to best-case, but this is not surprising because at the density employed in

this particular simulation there is likely to be at least one cell from each host in each bin (Table 3). Importantly, this scenario demonstrates that if the underlying distribution matrix is known, then the correction algorithm generally recovers accurate quantitative estimates.

Low marker prevalence (#9). Our simulations also show low marker shedding rates can yield significant problems. For example, Soule et al. [14] reported that as few as 2% of *Enterococcus* tested from a given host were marker-bearing isolates. To simulate this shedding level, we used a marker matrix $\mathbf{M} = 0.02 \cdot \mathbf{I}$. The results show significant error in the quantitative measure and unsatisfactory performance for presence/abscense sampling (Tables 3 and A9). In effect, this error is due to sparse marker occurence within an otherwise dense environment.

TABLE 4. Marker prevalence matrices. Matrix columns indicate the relative percentages of markers found in five hosts (cow, dog, elk, goose, human) and rows correspond to five distinct markers. These matrices correspond to the marker prevalence in simulation scenarios 4 and 8.

Non-uniform prevalence	Non-specific occurrence
$\begin{pmatrix} 0.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.8 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.4 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.6 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.7 \end{pmatrix}$	$\begin{pmatrix} 0.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 & 0.0 & 0.0 \\ 0.1 & 0.1 & 0.5 & 0.1 & 0.1 \\ 0.0 & 0.0 & 0.0 & 0.5 & 0.0 \\ 0.1 & 0.0 & 0.1 & 0.0 & 0.5 \end{pmatrix}$

5 Discussion

5.1 Sources of Error

Our simulations highlight two significant sources of error when applying genetic BST markers: first, errors that arise from variation in the distribution and density of fecal contamination in the environment; second, errors that arise from variation in the levels of marker occurrence and specificity among hosts. In the first case we derived significant error from environments that are sparse

(low-density) in terms of total amount of contamination or in marker-bearing contamination and from environments that are not well-mixed. Given an accurate method for quantitative measurement and a sufficiently large sample space, environments that vary in terms of total amount of contamination per host did not show significant error. We also found a more pronounced error in environments that were both sparse and unmixed. In practice, we might hypothesize that environments are likely to have one or both of these features. Furthermore, our coefficient of variation measurement (which indicates the degree of mixing and density of contamination) would likely be a time-varying parameter; if samples are obtained over a period of days, the environment may be well-mixed and dense on one occasion and not on another. Such variation is also likely at different sampling locations. Further experiments are needed to determine how actual environments correspond to models that include sparseness and mixing as variables. Without experimental data regarding the validity of these parameters, it is difficult to estimate the significance of the second type of error. At the very least our simulations suggest that more studies are needed before we can have confidence in measurements that rely on quantification.

In terms of variation in the levels of marker occurrence, we found that significant error arises when markers do not represent the actual amount of fecal contamination due to each host. Another obvious source of error arises when there is non-specific marker occurrence. The former variation is perhaps the most significant error suggested by our simulations in that, in real-world applications, we would be highly unlikely to find markers that are shed at exactly the same percentage of total contamination for all hosts. For this reason, our results indicate that without accurate knowledge of the levels of marker occurrence, confidence for any quantitative measurements will be low. According to our study, in order to apply discrete marker BST one would need to first determine the amount of marker produced by each host and then determine quantitative measurements of environmental samples. For instance, for a given amount of contamination from cow hosts one would need to determine what percentage contains no marker, cow marker, dog marker, goose marker, etc. Soule et al. [14] suggested quantitative PCR as a possible method, but highlighted some complications related to its use. For instance, because of the presence of PCR inhibiting factors

in environmental samples, a standard curve needs to be determined for each sample. Determining the percentage of "no marker" in a sample may be difficult for quantitative PCR (this assumes availability of a genus, species, or lineage marker with a high degree of specificity). However, our model and correction method will work even when relative percentages, rather than absolute quantification, of markers are estimated. For either a relative or absolute quantification approach, it is clear that an accurate measure of marker occurrence will be critical for generating accurate estimates.

5.2 Error Correction

Assuming a valid method of quantification can be identified, there are still hurdles to overcome with regard to determining levels of marker occurrence. One question that needs to be answered is whether marker prevalence is constant in space and time. If the prevalence is not constant, a new set of marker occurrence data may be needed for each geographical area, which would make the BST method time consuming and cost ineffective [16]. Yet if these problems can be overcome, our simulations imply that we can make up for errors in measurement by applying the correction method outlined in this study. Furthermore, this method can compensate for any level of non-specific occurrence or difference in marker prevalence. Thus, less effort could be spent finding markers with high specificity as long as the prevalence between host populations is accurately measured. In addition, the correction method described here only applies to error arising from marker prevalence and occurrence, but not from variation due to mixing and density of the environment.

The distribution of sampling points (localized vs. random over a wide area) can also affect estimates. Scenarios producing high estimation variance can be mitigated to some extent by distributing sample collections over a wide spatial domain as was demonstrated by scenarios 6 and 7, given unmixed environments. Quantification via presence/absence measurement performed well in the low-density environments, but only when we assume equal volumes of cells by host which may be unreasonable in practice.

The preceding discussion applies to the quantitative method of measurement, which was the main focus of our study. But we also examined the presence/absence method of measurement, which is likely to be a frequently used format for discrete genetic marker BST. This method showed significant error in two cases: first, when there were unequal amounts of contamination among hosts but at least some contamination from each host in nearly every sample (thus estimating equal contribution between hosts); second, when there were equal volumes of contamination among hosts but significant number of samples that lacked contamination from one or more hosts (e.g., due to low density in marker-bearing contamination). Thus, presence/absence only performed better than the quantitative measure in dense environments that have equal proportions of contamination among hosts, which is probably an unlikely scenario.

5.3 Unmodeled Sources of Variation

In addition to our model parameters, there are other aspects of variation we did not examine in this study. For instance, our model does not include time-varying rates of contamination input to the environment, which we would expect from real-world sources. Transient events such as a rain storm or a disturbance in a riverbed may produce environments that are dense and rich in a particular marker and which then disperse over time. We also assumed simplified survival dynamics for bacteria in an aqueous environment. We assumed that all cells of a particular host followed a constant decay rate once they entered the environment, but the decay rate is likely to be time-varying and the survival characteristics of marker-bearing bacteria may differ from other bacteria from the same host. Sampling methods that require bacterial culture may also produce variance due to differential recovery rates. Another source of variation we did not model concerns the amount of marker produced per unit weight of fecal material. In the marker prevalence matrix we model the amount of non-marker producing cells, but this refers only to the indicator bacteria (e.g., *Enterococcus*). The marker-per-unit-weight variation is similar to the variation we simulated with unequal marker prevalence and, thus, could have a similar effect on measurement error.

Our assumption is that the unmodeled sources of variation mentioned above would lower our

confidence in quantitative measurements beyond that implied by modeled variation. This combined with the lack of a method for correcting for sparse, unmixed environments, and the uncertainty in deriving accurate levels of marker occurrence per total amount of contamination leads us to conclude that absolute quantification of fecal contamination may not be possible using discrete marker BST. In order to confirm or modify our conclusions we suggest that future research be directed in two directions: first, developing methods to accurately quantify a group of markers in a given sample; second, clearly defining marker prevalence across hosts, space, and time to assess the feasibility of "correcting" estimates from field samples. If the variance in measurement of marker prevalence is found to be low, research could then focus on determining whether real-world environments are suitable (of sufficient density and uniformity) for use of the correction method introduced in this study.

References

[1] Bernhard, A. E.; Field, K. G. A PCR assay to discriminate human and ruminant feces on the basis of host differences in *Bacteroides-Prevotella* genes encoding 16S rRNA. Appl. Environ. Microbiol. 2000, 66, 4571-4574.

[2] Bernhard, A. E.; Field, K. G. Identification of nonpoint sources of fecal pollution in coastal waters by using host-specific 16S ribosomal DNA genetic markers from fecal anaerobes. Appl. Environ. Microbiol. 2000, 66, 1587-1594.

[3] Dick, L. K.; Bernhard, A. E.; Brodeur, T. J.; Santo Domingo, J. W.; Simpson, J. M.; Walters, S. P.; Field, K. G.; Host distributions of uncultivated fecal *Bacteroidales* bacteria reveal genetic markers for fecal source identification. Appl. Environ. Microbiol. 2005, 71, 3184-3191.

[4] Dick, L. K.; Field, K. G.; Rapid estimation of numbers of fecal *Bacteroidetes* by use of a quantitative PCR assay for 16S rRNA genes. Appl. Environ. Microbiol. 2004. 70, 5695-5697.

[5] Durán, A. E.; Muniesa, M.; Méndez, X.; Valero, F.; Lucena, F.; Jofre, J. Removal and inactivation of indicator bacteriophages in fresh waters. J. Appl. Microbiol. 2002, 92, 338-347.

[6] Ferguson, C. M.; Coote, B. G.; Ashbolt, N. J.; Stevenson, I. M. Relationships between indicators, pathogens and water quality in an esturine system. Wat. Res. 1996, 30, 2045-2054.

[7] Gabutti, G.; De Donno, A.; Bagordo, F.; Montangna, M. T. Comparative survival of faecal and human contaminants and use of *Staphylococcus aureus* as an effective indicator of human pollution. Mar. Pollut. Bull. 2000, 40, 697-700.

[8] Hamilton, M. J.; Yan, T.; Sadowsky, M. J. Development of goose- and duck-specific DNA markers to determine sources of *Escherichia coli* in waterways. Appl. Environ. Microbiol. 2006, 72, 4012-4019.

[9] Scott, T. M.; Jenkins, T. M.; Lukasik, J.; Rose, J. B.; Potential use of a host associated molecular marker in *Enterococcus faecium* as an index of human fecal pollution. Environ. Sci. Technol. 2005, 39, 283-287.

[10] Shanks, O. C.; Nietch, C.; Simonich, M.; Younger, M.; Reynolds, D.; Field, K. G. Basin-wide analysis of the dynamics of fecal contamination and fecal source identification in Tillamook Bay, Oregon. Appl. Environ. Microbiol. 2006, 72, 5537-5546.

[11] Shanks, O. C.; Santo Domingo, J. W.; Lamendella, R.; Kelty, C. A.; Graham, J. E.; Competitive metagenomic DNA hybridization identifies host-specific microbial genetic markers in cow fecal samples. Appl. Environ. Microbiol. 2006, 72, 4054-4060.

[12] Shanks, O. C.; Santo Domingo, J. W.; Lu, J.; Kelty, C. A.; Graham, J. E.; Identification of bacterial DNA markers for the detection of human fecal pollution in water. Appl. Environ. Microbiol. 2006 (in press).

[13] Sinton, L. W.; Hall, C. H.; Lynch, P. A.; Davies-Colley, R. J. Sunlight inactivation of fecal indicator bacteria and bacteriophages from waste stabilization pond effluent in fresh and saline waters. Appl. Environ. Microbiol. 2002, 68, 1122-1131.

[14] Soule, M.; Kuhn, E.; Loge, F.; Gay, J.; Call, D. R. Using DNA microarrays to identify library-independent markers for bacterial source tracking. Appl. Environ. Microbiol. 2006, 72, 1843-1851.

[15] Stewart, J. R.; Santo Domingo, J. W.; Wade, T. J. *In* Santo Domingo, J. W.; Sadowsky, M. J. (ed.) 2007, Microbial Source Tracking. ASM Press, Washington, D.C.

[16] Stoeckel, D. M.; Mathes, M. V.; Hyer, K. E.; Hagedorn, C.; Kator, H.; Lukasik, J.; OBrien, T. L.; Fenger, T. W.; Samadpour, M.; Strickler, K. M.; Wiggins, B. A. Comparison of seven protocols to identify fecal contamination sources using *Escherichia coli.* Environ. Sci. Technol. 2004, 15, 6109-6117.

Appendix A

Detailed Simulation Results

TABLE A1. Best-case scenario.

	Cow	Dog	Elk	Goose	Human
Presence/absence average:	-0.16	-0.03	-0.00	0.12	0.08
Standard deviation:	0.44	0.69	0.53	0.54	0.49
Quantitative average:	-0.94	0.31	-0.14	0.94	-0.18
Standard deviation:	6.36	5.64	6.02	5.61	6.45
Corrected average:	-0.94	0.32	-0.14	0.94	-0.18
Standard deviation:	6.36	5.64	6.02	5.61	6.45
Average COV:	28.00	28.42	28.33	28.43	28.58

TABLE A2. Low density environment scenario.

	Cow	Dog	Elk	Goose	Human
Presence/absence average:	1.79	-0.96	-0.08	-3.33	2.40
Standard deviation:	24.58	31.10	22.38	25.70	26.02
Quantitative average:	-15.50	-18.68	-17.09	-22.22	-12.79
Standard deviation:	25.48	33.69	24.52	24.03	29.12
Average COV:	141.26	141.99	141.84	141.82	142.03

TABLE A3. Unequal cell volume scenario.

	Cow	Dog	Elk	Goose	Human
Signal strength:	1	6	10	5	15
Presence/absence average:	360.72	34.64	-18.49	60.09	-45.62
Standard deviation:	74.05	3.41	1.99	4.80	1.35
Quantitative average:	3.16	-1.72	0.79	0.62	-0.26
Standard deviation:	22.55	8.08	3.76	7.50	3.56
Average COV:	90.07	36.72	28.45	39.96	23.10

TABLE A4. Unequal marker prevalence scenario.

	Cow	Dog	Elk	Goose	Human
Presence/absence average:	0.12	-0.00	-0.32	0.21	0.00
Standard deviation:	0.60	0.54	1.05	0.55	0.51
Quantitative average:	-16.21	32.35	-33.36	1.26	15.87
Standard deviation:	6.23	9.31	5.23	6.64	6.88
Corrected average:	0.46	-0.76	-0.14	1.18	-0.74
Standard deviation:	7.20	7.53	7.38	6.43	6.03
Average COV:	28.54	28.21	28.38	28.46	28.54

TABLE A5. Unequal decay rates scenario simulation results.

	Cow	Dog	Elk	Goose	Human
Decay rate (α):	-0.0100	-0.0030	-0.0020	-0.0200	-0.0015
Presence/absence average:	72.86	-16.06	-26.47	194.39	-31.33
Standard deviation:	3.91	1.58	1.24	22.63	1.27
Quantitative average:	0.61	-1.70	-0.33	1.47	1.16
Standard deviation:	6.78	5.75	4.41	11.09	4.46
Average COV:	41.05	28.29	26.64	56.54	25.61

TABLE A6. Unmixed environment (with contiguous sampling) scenario.

	Cow	Dog	Elk	Goose	Human
Presence/absence average:	17.67	-13.99	3.59	32.30	-39.60
Standard deviation:	115.73	83.97	85.83	111.91	80.89
Quantitative average:	33.55	-13.95	-2.46	-11.91	-33.74
Standard deviation:	165.13	118.84	133.06	96.08	111.96
Average COV:	150.30	153.52	139.05	145.23	151.26

TABLE A7. Unmixed environment (with uniform sampling) scenario.

	Cow	Dog	Elk	Goose	Human
Presence/absence average:	4.69	-1.05	-6.59	-7.26	10.06
Standard deviation:	20.45	19.53	19.87	24.10	21.01
Quantitative average:	1.60	-2.29	-5.49	-11.43	10.60
Standard deviation:	33.81	25.31	31.82	35.39	29.36
Average COV:	142.34	151.69	148.39	152.13	144.13

TABLE A8. Non-specific marker occurrence scenario.

	Cow	Dog	Elk	Goose	Human
Presence/absence average:	0.07	-0.61	0.25	0.06	0.24
Standard deviation:	0.75	1.13	0.72	0.98	0.53
Quantitative average:	-19.89	-19.78	45.90	-19.36	13.19
Standard deviation:	4.55	5.02	7.84	5.84	7.32
Corrected average:	-0.68	-0.55	1.02	-0.03	0.24
Standard deviation:	5.83	6.16	12.19	7.20	10.38
Average COV:	28.46	28.38	28.49	28.64	28.19

TABLE A9. Low marker prevalence scenario.

	Cow	Dog	Elk	Goose	Human
Presence/absence average:	-1.69	-2.97	-18.01	12.70	9.86
Standard deviation:	24.84	33.08	29.64	31.07	35.60
Quantitative average:	-28.52	-29.80	-42.62	-14.43	-19.69
Standard deviation:	25.12	30.83	23.54	30.28	29.86
Average COV:	28.33	28.47	28.38	28.51	28.54

Model Parameters

TABLE A10. Summary of model parameters.

Number of sources in the environment per host
Signal strength for each host (cells shed per time step)
Decay rate of cells from each host
Marker prevalence for each host (marker matrix data)
Environment size (number of bins)
Bin size (max number of cells per bin)
Simulation time (number of time steps before sampling)
Number of bins sampled
Distribution (Gaussian or uniform)
Standard deviation for Gaussian distribution
Number of simulation repetitions
Sampling method (contiguous or random throughout environment)

Appendix B

Main Simulation Module

```
/*************************************************************************
 *   Copyright (C) May 2007 by Mark Leach                                *
 *   mleach@eecs.wsu.edu                                                 *
 *   simulation.c                                                        *
 *   Main module for the microbial source tracking (MST) simulation.     *
 *************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include <gsl_rng.h>
#include <gsl_randist.h>
#include <gsl_math.h>
#include <gsl_statistics_double.h>
#include "mst_sim.h"

void run_sim(sim_type * sim_ptr, stats_type * stats_ptr, gsl_rng * r_ptr);

int main(int argc, char *argv[])
{
  sim_type *sim_ptr = NULL;
  stats_type *stats_ptr = NULL;
  gsl_rng *r_ptr;

  if (!(stats_ptr = (stats_type *) malloc(sizeof(stats_type)))) {
    fprintf(stderr, "Error allocating stats object.\n");
    return EXIT_FAILURE;
  }

  mst_clear_stats(stats_ptr);

  if (!(sim_ptr = mst_build_sim())) {
    return EXIT_FAILURE;
  }

  mst_load_seeds(sim_ptr, "rands.dat");

  if (!sim_ptr->run_sim)
    return EXIT_FAILURE;

  /* set up the random number generator */
  gsl_rng_env_setup();
  r_ptr = gsl_rng_alloc(gsl_rng_mt19937);

  /* run the simulation */
  gsl_rng_set(r_ptr, sim_ptr->seeds[0]);
  run_sim(sim_ptr, stats_ptr, r_ptr);
```

```
  mst_store_seeds(sim_ptr, "rands.dat");
  gsl_rng_free(r_ptr);
  mst_free_sim(sim_ptr);
  free(stats_ptr);
  return EXIT_SUCCESS;
}


/*********************************************************************
* run_sim:    Run the simulation.
* In:         Simulation pointer, stats pointer, random number generator.
* Out:        None.
* Modifies:   Simulation and stats objects.
* Allocation: None.
*********************************************************************/
void run_sim(sim_type * sim_ptr, stats_type * stats_ptr, gsl_rng * r_ptr)
{
  int i, j, k;
  double sdu_stats[HOSTS][MAX_STATS]; /* sample difference uncorrected */
  double sdc_stats[HOSTS][MAX_STATS]; /* sample difference corrected */
  double sp_stats[HOSTS][MAX_STATS];  /* presence stats */
  double ep_stats[HOSTS][MAX_STATS];  /* environment parameter */

  /* clear the stats arrays */
  for (i = 0; i < HOSTS; i++) {
    for (j = 0; j < MAX_STATS; j++) {
      sdu_stats[i][j] = 0.0;
      sdc_stats[i][j] = 0.0;
      sp_stats[i][j] = 0.0;
      ep_stats[i][j] = 0.0;
    }
  }

  /* outer loop for repetitions */
  for (i = 0; i < sim_ptr->parameters[REPS]; i++) {
    /* create the environment */
    mst_build_pop(sim_ptr, r_ptr);

    /* generate and die off loop */
    for (j = 0; j < sim_ptr->parameters[SIM_TIME]; j++) {
      mst_generate(sim_ptr, r_ptr);
      mst_die_off(sim_ptr);
    }

    mst_count_env(sim_ptr, stats_ptr);
    mst_count_markers(sim_ptr, stats_ptr);
    mst_sample_env(sim_ptr, r_ptr, stats_ptr);
    mst_get_cov(sim_ptr, stats_ptr);
    mst_solve_system(sim_ptr, stats_ptr);

    /* Accumulate the stats for each repetition. */
    for (k = COW; k < HOSTS; k++) {
      /* % error marker vs. total cells (uncorrected) */
      sdu_stats[k][i] =
```

```
        100 * (stats_ptr->sample_marker[k] -
              stats_ptr->total_cells[k]) / stats_ptr->total_cells[k];

    /* % error marker vs. total cells (corrected) */
    sdc_stats[k][i] =
        100 * (stats_ptr->sample_corrected[k] -
              stats_ptr->total_cells[k]) / stats_ptr->total_cells[k];

    /* % error presence quantification (uncorrected) */
    sp_stats[k][i] =
        100 * (stats_ptr->sample_presence[k] -
              stats_ptr->total_cells[k]) / stats_ptr->total_cells[k];

    /* environment parameter */
    ep_stats[k][i] = stats_ptr->env_density[k];
  }

  mst_clear_env(sim_ptr);
  mst_clear_stats(stats_ptr);
}

/* Calculate averages and print the simulation statistics. */
/* uncorrected mean +/- */
printf("\n & %.4f & %.4f & %.4f & %.4f & %.4f\n",
      gsl_stats_mean(&sp_stats[COW][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sp_stats[DOG][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sp_stats[ELK][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sp_stats[GOOSE][0], 1,
                     sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sp_stats[HUMAN][0], 1,
                     sim_ptr->parameters[REPS]));
/* uncorrected sd +/- */
printf(" & %.4f & %.4f & %.4f & %.4f & %.4f\n",
      gsl_stats_sd(&sp_stats[COW][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_sd(&sp_stats[DOG][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_sd(&sp_stats[ELK][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_sd(&sp_stats[GOOSE][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_sd(&sp_stats[HUMAN][0], 1, sim_ptr->parameters[REPS]));
/* uncorrected mean quantitative */
printf("\n & %.4f & %.4f & %.4f & %.4f & %.4f\n",
      gsl_stats_mean(&sdu_stats[COW][0], 1,
                     sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sdu_stats[DOG][0], 1,
                     sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sdu_stats[ELK][0], 1,
                     sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sdu_stats[GOOSE][0], 1,
                     sim_ptr->parameters[REPS]),
      gsl_stats_mean(&sdu_stats[HUMAN][0], 1,
                     sim_ptr->parameters[REPS]));
/* uncorrected sd quantitative */
printf(" & %.4f & %.4f & %.4f & %.4f & %.4f\n",
      gsl_stats_sd(&sdu_stats[COW][0], 1, sim_ptr->parameters[REPS]),
      gsl_stats_sd(&sdu_stats[DOG][0], 1, sim_ptr->parameters[REPS]),
```

```
        gsl_stats_sd(&sdu_stats[ELK][0], 1, sim_ptr->parameters[REPS]),
        gsl_stats_sd(&sdu_stats[GOOSE][0], 1,
                     sim_ptr->parameters[REPS]),
        gsl_stats_sd(&sdu_stats[HUMAN][0], 1, sim_ptr->parameters[REPS]));
  /* corrected mean quantitative */
  printf("\n & %.4f & %.4f & %.4f & %.4f & %.4f\n",
        gsl_stats_mean(&sdc_stats[COW][0], 1,
                       sim_ptr->parameters[REPS]),
        gsl_stats_mean(&sdc_stats[DOG][0], 1,
                       sim_ptr->parameters[REPS]),
        gsl_stats_mean(&sdc_stats[ELK][0], 1,
                       sim_ptr->parameters[REPS]),
        gsl_stats_mean(&sdc_stats[GOOSE][0], 1,
                       sim_ptr->parameters[REPS]),
        gsl_stats_mean(&sdc_stats[HUMAN][0], 1,
                       sim_ptr->parameters[REPS]));
  /* corrected sd quantitative */
  printf(" & %.4f & %.4f & %.4f & %.4f & %.4f\n",
        gsl_stats_sd(&sdc_stats[COW][0], 1, sim_ptr->parameters[REPS]),
        gsl_stats_sd(&sdc_stats[DOG][0], 1, sim_ptr->parameters[REPS]),
        gsl_stats_sd(&sdc_stats[ELK][0], 1, sim_ptr->parameters[REPS]),
        gsl_stats_sd(&sdc_stats[GOOSE][0], 1,
                     sim_ptr->parameters[REPS]),
        gsl_stats_sd(&sdc_stats[HUMAN][0], 1, sim_ptr->parameters[REPS]));
  /* environment parameter */
  printf("\n & %.4f & %.4f & %.4f & %.4f & %.4f\n",
        gsl_stats_mean(&ep_stats[COW][0], 1, sim_ptr->parameters[REPS]),
        gsl_stats_mean(&ep_stats[DOG][0], 1, sim_ptr->parameters[REPS]),
        gsl_stats_mean(&ep_stats[ELK][0], 1, sim_ptr->parameters[REPS]),
        gsl_stats_mean(&ep_stats[GOOSE][0], 1,
                       sim_ptr->parameters[REPS]),
        gsl_stats_mean(&ep_stats[HUMAN][0], 1,
                       sim_ptr->parameters[REPS]));
}
```

Simulation Functions

```
/*****************************************************************************
 *   Copyright (C) May 2007 by Mark Leach                                    *
 *   mleach@eecs.wsu.edu                                                     *
 *   mst_sim.c                                                               *
 *   Functions for the mst module.                                           *
 *****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <gsl_rng.h>
#include <gsl_randist.h>
#include <gsl_math.h>
#include <gsl_statistics_double.h>
#include <gsl_linalg.h>
#include "mst_sim.h"
```

```
/*************************************************************************
* mst_load_conf: Load the simulation parameters from a config file.
* In:            Simulation.
* Out:           False if there is no config file or there is an error
*                reading the file, otherwise true.
* Modifies:      Simulation object.
* Allocation:    None.
*************************************************************************/
BOOL mst_load_conf(sim_type * sim_ptr)
{
  int i, j;
  char buffer[BUF_LENGTH];
  FILE *confp;

  if (!(confp = fopen("mst_sim.conf", "r"))) {
    fprintf(stderr, "File mst_sim.conf not found.\n");
    return FALSE;
  }

  /* load the number of hosts */
  for (i = 0; i < HOSTS; i++) {
    if (fgets(buffer, BUF_LENGTH, confp) != NULL) {
      sim_ptr->hosts[i] = atoi(strtok(buffer, " "));
    } else {
      return FALSE;
    }
  }

  /* load the signal strengths */
  for (i = 0; i < HOSTS; i++) {
    if (fgets(buffer, BUF_LENGTH, confp) != NULL) {
      sim_ptr->rates[i] = atoi(strtok(buffer, " "));
    } else {
      return FALSE;
    }
  }

  /* load the decay rates */
  for (i = 0; i < HOSTS; i++) {
    if (fgets(buffer, BUF_LENGTH, confp) != NULL) {
      sim_ptr->decay[i] = atoi(strtok(buffer, " "));
    } else {
      return FALSE;
    }
  }

  /* load the marker matrix */
  for (i = 0; i < HOSTS; i++) {
    for (j = 0; j < HOSTS; j++) {
      if (fgets(buffer, BUF_LENGTH, confp) != NULL) {
        sim_ptr->markers[j][i] = atoi(strtok(buffer, " "));
      } else {
        return FALSE;
```

```
      }
    }
  }

  /* load the simulation parameters */
  for (i = 0; i < PARAMS; i++) {
    if (fgets(buffer, BUF_LENGTH, confp) != NULL) {
      sim_ptr->parameters[i] = atoi(strtok(buffer, " "));
    } else {
      return FALSE;
    }
  }

  return TRUE;
}

/*****************************************************************
* mst_build_host: Allocate one instance of a host.
* In:             Simulation pointer, kind of host to build, location of
*                 the host.
* Out:            NULL if there is an error allocating host, otherwise
*                 the pointer to new host.
* Modifies:       None.
* Allocation:     One host object.
*****************************************************************/
host_type *mst_build_host(sim_type * sim_ptr, species_type kind,
                          int location)
{
  host_type *host;

  if (!(host = (host_type *) malloc(sizeof(host_type)))) {
    fprintf(stderr, "Error allocating host.\n");
    return host;
  }

  host->species = kind;
  host->location = location;
  host->shed_rate = sim_ptr->rates[kind];
  return host;
}

/*****************************************************************
* mst_build_pop: Place hosts into the population array. If there is an
*                attempt to place a host in an already-occupied spot,
*                keep trying with a new random number up to max tries of
*                SEARCHLIMIT.
* In:            Simulation pointer, random number generator
* Out:           None.
* Modifies:      The population array.
* Allocation:    None.
*****************************************************************/
void mst_build_pop(sim_type * sim_ptr, const gsl_rng * r_ptr)
{
  int i;
```

```
  int count = 0;
  unsigned long rand;
  species_type host;

  for (host = COW; host <= HUMAN; host++) {
    for (i = 0; i < sim_ptr->hosts[host]; i++) {
      rand = gsl_rng_uniform_int(r_ptr, sim_ptr->parameters[ENV_MAX]);

      /* keep searching for a spot in the environment up to SEARCHLIMIT */
      while (sim_ptr->pop[rand] != NULL) {

        if (count++ > SEARCHLIMIT) {
          fprintf(stderr, "Too many hosts in environment.\n");
          return;
        }

        rand = gsl_rng_uniform_int(r_ptr, sim_ptr->parameters[ENV_MAX]);
      }

      sim_ptr->pop[rand] = mst_build_host(sim_ptr, host, rand);
    }
  }
}

/*********************************************************************
* mst_build_sim: Build the simulation object.
* In:            None.
* Out:           NULL if simulation cannot be allocated or there is an
*                error loading the config file, otherwise pointer to
*                the simulation.
* Modifies:      None.
* Allocation:    Environment pointer.
*********************************************************************/
sim_type *mst_build_sim(void)
{
  int i, j;
  sim_type *sim_ptr;

  if (!(sim_ptr = (sim_type *) malloc(sizeof(sim_type)))) {
    fprintf(stderr, "Error allocating simulation.\n");
    return sim_ptr;
  }

  if (!mst_load_conf(sim_ptr)) {
    return NULL;
  }

  /* initialize seeds */
  for (i = 0; i < SEED_MAX; i++) {
    sim_ptr->seeds[i] = 0;
  }

  /* initialize population to empty */
  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
```

```
    sim_ptr->pop[i] = NULL;
  }

  /* initialize environment to empty */
  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {
      sim_ptr->ground[i][j] = NULL;
    }
  }

  sim_ptr->run_sim = TRUE;

  return sim_ptr;
}

/*************************************************************************
* mst_add_eu:   Add an excretory unit (cell) to the environment.
* In:           Simulation pointer, index of cell, cell.
* Out:          None.
* Modifies:     One of the environment bins.
* Allocation:   None.
*************************************************************************/
void mst_add_eu(sim_type * sim_ptr, int index, eu_type * eu_ptr)
{
  int i = 0;

  /* traverse array to empty spot */
  while (sim_ptr->ground[index][i++] != NULL) {

    if (i == sim_ptr->parameters[EU_MAX])
      return;
  }

  sim_ptr->ground[index][--i] = eu_ptr;
  return;
}

/*************************************************************************
* mst_get_marker: Get a random marker type according to the given host.
* In:             Simulation pointer, random number generator, host type.
* Out:            Marker type.
* Modifies:       None.
* Allocation:     None.
*************************************************************************/
species_type
mst_get_marker(sim_type * sim_ptr, const gsl_rng * r_ptr,
               species_type host)
{
  unsigned long int rand =
      gsl_rng_uniform_int(r_ptr, sim_ptr->parameters[M_BOUND]);
  species_type marker;
  int i1, i2, i3, i4, i5;

  /* Construct an interval. */
```

```
  i1 = sim_ptr->markers[COW][host];
  i2 = i1 + sim_ptr->markers[DOG][host];
  i3 = i2 + sim_ptr->markers[ELK][host];
  i4 = i3 + sim_ptr->markers[HUMAN][host];
  i5 = i4 + sim_ptr->markers[GOOSE][host];

  /* Check if random number is in the interval. */
  if (rand < i1)
    marker = COW;
  else if (rand >= i1 && rand < i2)
    marker = DOG;
  else if (rand >= i2 && rand < i3)
    marker = ELK;
  else if (rand >= i3 && rand < i4)
    marker = HUMAN;
  else if (rand >= i4 && rand < i5)
    marker = GOOSE;
  else
    marker = NONE;

  return marker;
}

/*************************************************************************
* mst_get_life_span: Get the life span of a cell using an exponential
*                    distribution.
* In:                Simulation pointer, random number generator,
*                    location of the shedding host in the population
* Out:               Life span of a cell.
* Modifies:          None.
* Allocation:        None.
*************************************************************************/
int mst_get_life_span(sim_type * sim_ptr, const gsl_rng * r_ptr, int index)
{
  double mu;
  double rand;
  int span;
  mu = sim_ptr->decay[sim_ptr->pop[index]->species];
  rand = gsl_ran_exponential(r_ptr, mu);
  span = ceil(rand);
  return span;
}

/*************************************************************************
* mst_get_index: Get the location (bin) of a new cell.
* In:            Simulation pointer, random number generator, location
*                of the host in the population
* Out:           Life span of a cell.
* Modifies:      None.
* Allocation:    None.
*************************************************************************/
int mst_get_index(sim_type * sim_ptr, const gsl_rng * r_ptr, int index)
{
  double g_rand;
```

```
  switch (sim_ptr->parameters[DIST]) {
  case GAUSS:
    g_rand = gsl_ran_gaussian(r_ptr, (double) sim_ptr->parameters[SD]);
    return (g_rand + sim_ptr->pop[index]->location);
    break;
  case UNIFORM:
    return gsl_rng_uniform_int(r_ptr, sim_ptr->parameters[ENV_MAX]);
    break;
  default:
    return 0;
  }
}

/*****************************************************************
* mst_generate: Generate signals from all hosts for one time step.
* In:           Simulation pointer, random number generator.
* Out:          None.
* Modifies:     None.
* Allocation:   Total number of cells for one time step.
*****************************************************************/
void mst_generate(sim_type * sim_ptr, const gsl_rng * r_ptr)
{
  int i, j, index;
  eu_type *eu_ptr;

  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    if (sim_ptr->pop[i] != NULL) {
      for (j = 0; j < sim_ptr->rates[sim_ptr->pop[i]->species]; j++) {

        if (!(eu_ptr = (eu_type *) malloc(sizeof(eu_type)))) {
          fprintf(stderr, "EU memory allocation error.\n");
          return;
        }

        eu_ptr->age = 0;
        eu_ptr->life_span = mst_get_life_span(sim_ptr, r_ptr, i);
        eu_ptr->species = sim_ptr->pop[i]->species;
        eu_ptr->marker =
            mst_get_marker(sim_ptr, r_ptr, sim_ptr->pop[i]->species);
        index = mst_get_index(sim_ptr, r_ptr, i);

        if (index >= 0 && index < sim_ptr->parameters[ENV_MAX]) {
          mst_add_eu(sim_ptr, index, eu_ptr);
        }
      }
    }
  }
}

/*****************************************************************
* mst_die_off: Age each cell in environment and check if it dies.
* In:          Simulation pointer.
* Out:         None.
```

```
* Modifies:    The age of every cell in the environment.
* Allocation:  None.
*************************************************************************/
void mst_die_off(sim_type * sim_ptr)
{
  int i, j;
  eu_type *eu_ptr;

  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {
      if (sim_ptr->ground[i][j] != NULL) {
        eu_ptr = sim_ptr->ground[i][j];

        if (eu_ptr->age == eu_ptr->life_span) {
          free(sim_ptr->ground[i][j]);
          sim_ptr->ground[i][j] = NULL;
        } else {
          eu_ptr->age++;
        }
      }
    }
  }
}

/*************************************************************************
* mst_free_sim: Free the allocated objects in the simulation.
* In:           Simulation pointer.
* Out:          None.
* Modifies:     Simulation.
* Allocation:   None.
*************************************************************************/
void mst_free_sim(sim_type * sim_ptr)
{
  int i, j;

  /* free cells in the bins */
  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {
      if (sim_ptr->ground[i][j] != NULL) {
        free(sim_ptr->ground[i][j]);
      }
    }
  }

  /* free the hosts in the population */
  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    if (sim_ptr->pop[i] != NULL) {
      free(sim_ptr->pop[i]);
    }
  }

  /* free the simulation */
  free(sim_ptr);
  return;
```

```
}

/************************************************************************
* mst_clear_env: Set the pointers in the simulation to NULL.
* In:            Simulation pointer.
* Out:           None.
* Modifies:      Environment and population arrays.
* Allocation:    None.
************************************************************************/
void mst_clear_env(sim_type * sim_ptr)
{
  int i, j;

  /* free eu's in the bins */
  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {
      if (sim_ptr->ground[i][j] != NULL) {
        free(sim_ptr->ground[i][j]);
        sim_ptr->ground[i][j] = NULL;
      }
    }
  }

  /* free the hosts in the population */
  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    if (sim_ptr->pop[i] != NULL) {
      free(sim_ptr->pop[i]);
      sim_ptr->pop[i] = NULL;
    }
  }

  return;
}

/************************************************************************
* mst_count_env: Count all of the cells in the environment and store
*                the statistics.
* In:            Simulation pointer, statistics pointer.
* Out:           None.
* Modifies:      Stats object.
* Allocation:    None.
************************************************************************/
void mst_count_env(sim_type * sim_ptr, stats_type * stats_ptr)
{
  int i, j;
  int count[HOSTS];
  int sum = 0;

  for (i = COW; i < HOSTS; i++) {
    count[i] = 0;
  }

  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {
```

```
      if (sim_ptr->ground[i][j] != NULL) {
        if (sim_ptr->ground[i][j]->species == COW) {
          count[COW]++;
        } else if (sim_ptr->ground[i][j]->species == DOG) {
          count[DOG]++;
        } else if (sim_ptr->ground[i][j]->species == ELK) {
          count[ELK]++;
        } else if (sim_ptr->ground[i][j]->species == GOOSE) {
          count[GOOSE]++;
        } else if (sim_ptr->ground[i][j]->species == HUMAN) {
          count[HUMAN]++;
        }
      }
    }
  }

  for (i = COW; i < HOSTS; i++) {
    sum += count[i];
  }

  for (i = COW; i < HOSTS; i++) {
    stats_ptr->total_cells[i] = (double) count[i] / sum;
  }
}

/*************************************************************************
* mst_print_env: Print the environment to file.
* In:            Simulation pointer.
* Out:           Text version of bins in the environment.
* Modifies:      None.
* Allocation:    None.
*************************************************************************/
void mst_print_env(sim_type * sim_ptr)
{
  int i, j;
  FILE *fp;

  if (!(fp = fopen("env_map.dat", "w"))) {
    fprintf(stderr, "Error opening environment output file.\n");
    return;
  }

  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    fprintf(fp, "%d ", i);
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {
      if (sim_ptr->ground[i][j] != NULL) {
        if (sim_ptr->ground[i][j]->species == COW) {
          fprintf(fp, "r");
        } else if (sim_ptr->ground[i][j]->species == DOG) {
          fprintf(fp, "b");
        } else if (sim_ptr->ground[i][j]->species == ELK) {
          fprintf(fp, "y");
        } else if (sim_ptr->ground[i][j]->species == GOOSE) {
          fprintf(fp, "g");
```

```
        } else if (sim_ptr->ground[i][j]->species == HUMAN) {
          fprintf(fp, "k");
        }
      }
    }

    fprintf(fp, "\n");
  }

  fclose(fp);
}


/*************************************************************************
* mst_count_markers: Count all of the marker-bearing cells in the
*                    environment.
* In:                Simulation pointer, stats pointer.
* Out:               None.
* Modifies:          None.
* Allocation:        None.
*************************************************************************/
void mst_count_markers(sim_type * sim_ptr, stats_type * stats_ptr)
{
  int i, j;
  int count[HOSTS];
  int sum = 0;

  for (i = COW; i < HOSTS; i++) {
    count[i] = 0;
  }

  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {
      if (sim_ptr->ground[i][j] != NULL) {
        if (sim_ptr->ground[i][j]->marker == COW) {
          count[COW]++;
        } else if (sim_ptr->ground[i][j]->marker == DOG) {
          count[DOG]++;
        } else if (sim_ptr->ground[i][j]->marker == ELK) {
          count[ELK]++;
        } else if (sim_ptr->ground[i][j]->marker == GOOSE) {
          count[GOOSE]++;
        } else if (sim_ptr->ground[i][j]->marker == HUMAN) {
          count[HUMAN]++;
        }
      }
    }
  }

  for (i = COW; i < HOSTS; i++) {
    sum += count[i];
  }

  for (i = COW; i < HOSTS; i++) {
```

```
    stats_ptr->total_marker[i] = (double) count[i] / sum;
  }
}

/*************************************************************************
* mst_solve_system: Solve the linear system (bst correction method) and
*                   store the corrected statistics.
* In:               Simulation pointer, stats pointer.
* Out:              None.
* Modifies:         Stats object.
* Allocation:       None.
*************************************************************************/
void mst_solve_system(sim_type * sim_ptr, stats_type * stats_ptr)
{
  double A_data[HOSTS * HOSTS];
  double b_data[HOSTS];
  int s, i, j;
  gsl_matrix_view m;
  gsl_vector_view b;
  gsl_vector *x;
  gsl_permutation *p;
  double sum = 0;

  /* load the marker matrix */
  for (i = 0; i < HOSTS; i++) {
    for (j = 0; j < HOSTS; j++) {
      A_data[i + j * HOSTS] =
          (double) (sim_ptr->markers[j][i]) /
          (sim_ptr->parameters[M_BOUND]);
    }
  }

  for (i = COW; i < HOSTS; i++) {
    b_data[i] = (double) stats_ptr->sample_marker[i];
  }

  /* put vectors and matrix in proper form */
  m = gsl_matrix_view_array(A_data, HOSTS, HOSTS);
  b = gsl_vector_view_array(b_data, HOSTS);
  x = gsl_vector_alloc(HOSTS);
  p = gsl_permutation_alloc(HOSTS);
  gsl_linalg_LU_decomp(&m.matrix, p, &s);
  gsl_linalg_LU_solve(&m.matrix, p, &b.vector, x);

  for (i = COW; i < HOSTS; i++) {
    sum += gsl_vector_get(x, i);
  }

  for (i = COW; i < HOSTS; i++) {
    stats_ptr->sample_corrected[i] = gsl_vector_get(x, i) / sum;
  }

  gsl_permutation_free(p);
}
```

```
/*****************************************************************
* mst_sample_bin: Sample a bin and store stats for the total and
*                 marker-bearing cells in the environment.
* In:             Simulation pointer, stats pointer, location of
*                 bin to sample.
* Out:            None.
* Modifies:       Stats object.
* Allocation:     None.
*****************************************************************/
void
mst_sample_bin(sim_type * sim_ptr, stats_type * s_ptr, int loc)
{
  int i; int sum_t = 0; int sum_m = 0;
  int marker_count[HOSTS] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
  int total_count[HOSTS] = { 0.0, 0.0, 0.0, 0.0, 0.0 };

  for (i = 0; i < sim_ptr->parameters[EU_MAX]; i++) {
    if (sim_ptr->ground[loc][i] != NULL) {

      /* count sample (all cells) */
      if (sim_ptr->ground[loc][i]->species == COW) {
        total_count[COW]++;
      } else if (sim_ptr->ground[loc][i]->species == DOG) {
        total_count[DOG]++;
      } else if (sim_ptr->ground[loc][i]->species == ELK) {
        total_count[ELK]++;
      } else if (sim_ptr->ground[loc][i]->species == GOOSE) {
        total_count[GOOSE]++;
      } else if (sim_ptr->ground[loc][i]->species == HUMAN) {
        total_count[HUMAN]++;
      }

      /* count sample (marker-bearing cells) */
      if (sim_ptr->ground[loc][i]->marker == COW) {
        marker_count[COW]++;
      } else if (sim_ptr->ground[loc][i]->marker == DOG) {
        marker_count[DOG]++;
      } else if (sim_ptr->ground[loc][i]->marker == ELK) {
        marker_count[ELK]++;
      } else if (sim_ptr->ground[loc][i]->marker == GOOSE) {
        marker_count[GOOSE]++;
      } else if (sim_ptr->ground[loc][i]->marker == HUMAN) {
        marker_count[HUMAN]++;
      }
    }
  }

  /* create sums for normalization */
  for (i = COW; i < HOSTS; i++) {
    sum_t += total_count[i];
    sum_m += marker_count[i];
  }
```

```
  /* tally stats for presence */
  for (i = COW; i < HOSTS; i++) {
    if (marker_count[i] != 0) {
      s_ptr->sample_presence[i] = 1;
    }
  }

  /* store normalized stats */
  for (i = COW; i < HOSTS; i++) {
    if (sum_t != 0) {
      s_ptr->sample_cells[i] = (double) total_count[i] / sum_t;
    } else {
      s_ptr->sample_cells[i] = 0;
    }

    if (sum_m != 0) {
      s_ptr->sample_marker[i] = (double) marker_count[i] / sum_m;
    } else {
      s_ptr->sample_marker[i] = 0;
    }
  }
}

/*************************************************************************
* mst_get_sspace: Get the sample space for an environment. Fill the
*                 simulation sample space with bin locations from a
*                 continguous or uniform distribution.
* In:             Simulation pointer, random number generator.
* Out:            None.
* Modifies:       Simulation object (sample space).
* Allocation:     None.
*************************************************************************/
void mst_get_sspace(sim_type * sim_ptr, const gsl_rng * r_ptr)
{
  BOOL test = TRUE;
  int rand;
  int i;

  if (sim_ptr->parameters[S_TYPE]) {

    /* find the sample space */
    while (test) {
      rand = gsl_rng_uniform_int(r_ptr, sim_ptr->parameters[ENV_MAX]);

      if ((rand + sim_ptr->parameters[NUM_SAMPLES]) <
          sim_ptr->parameters[ENV_MAX]) {
        test = FALSE;
      }
    }

    /* fill the (contiguous) sample space */
    for (i = 0; i < sim_ptr->parameters[NUM_SAMPLES]; i++) {
      sim_ptr->sample_space[i] = rand + i;
    }
```

```
  } else {

    /* fill sample space with bin chosen at random uniform locations */
    for (i = 0; i < sim_ptr->parameters[NUM_SAMPLES]; i++) {
      sim_ptr->sample_space[i] =
          gsl_rng_uniform_int(r_ptr, sim_ptr->parameters[ENV_MAX]);
    }
  }
}

/*****************************************************************
* mst_sample_env: Sample an environment.
* In:             Simulation pointer, random number generator, stats
*                 pointer.
* Out:            None.
* Modifies:       Overall simulation stats object.
* Allocation:     Temporary stats object.
*****************************************************************/
void
mst_sample_env(sim_type * sim_ptr, const gsl_rng * r_ptr,
               stats_type * stats_ptr)
{
  int i, j, k;
  stats_type *s_ptr = NULL;
  int s_sum = 0;
  double sp_count[HOSTS] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
  double cell_count[HOSTS] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
  double marker_count[HOSTS] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
  double s_cells[HOSTS][MAX_STATS]; /* normalized sampled cells   */
  double s_marker[HOSTS][MAX_STATS]; /* normalized sampled markers */
  double s_presence[HOSTS][MAX_STATS]; /* positive bins             */

  /* clear the stats arrays */
  for (i = 0; i < HOSTS; i++) {
    for (j = 0; j < MAX_STATS; j++) {
      s_cells[i][j] = 0.0;
      s_marker[i][j] = 0.0;
    }
  }

  /* Allocate the stats object for one bin stats. */
  if (!(s_ptr = (stats_type *) malloc(sizeof(stats_type)))) {
    fprintf(stderr, "Error allocating stats object.\n");
    return;
  }

  /* Clear the temp stats object. */
  for (k = COW; k < HOSTS; k++) {
    s_ptr->sample_cells[k] = 0.0;
    s_ptr->sample_marker[k] = 0.0;
    s_ptr->sample_presence[k] = 0.0;
  }

  mst_get_sspace(sim_ptr, r_ptr);
```

```
  /* sample each of the bins in the sample space  */
  for (i = 0; i < sim_ptr->parameters[NUM_SAMPLES]; i++) {

    mst_sample_bin(sim_ptr, s_ptr, sim_ptr->sample_space[i]);

    /* save the results of the bin sample */
    for (k = COW; k < HOSTS; k++) {
      s_cells[k][i] = s_ptr->sample_cells[k];
      s_marker[k][i] = s_ptr->sample_marker[k];
      s_presence[k][i] = s_ptr->sample_presence[k];
    }

    /* clear the temp stats object */
    for (k = COW; k < HOSTS; k++) {
      s_ptr->sample_cells[k] = 0.0;
      s_ptr->sample_marker[k] = 0.0;
      s_ptr->sample_presence[k] = 0.0;
    }
  }

  /* count up the total counts and presence tally */
  for (i = COW; i < HOSTS; i++) {
    for (j = 0; j < sim_ptr->parameters[NUM_SAMPLES]; j++) {
      sp_count[i] += s_presence[i][j];
      cell_count[i] += s_cells[i][j];
      marker_count[i] += s_marker[i][j];
    }
  }

  for (k = COW; k < HOSTS; k++) {
    s_sum += sp_count[k];
  }

  /* Store the averages of the sample space and presence tally. */
  for (k = COW; k < HOSTS; k++) {
    stats_ptr->sample_cells[k] =
        gsl_stats_mean(&s_cells[k][0], 1,
                       sim_ptr->parameters[NUM_SAMPLES]);
    stats_ptr->sample_marker[k] =
        gsl_stats_mean(&s_marker[k][0], 1,
                       sim_ptr->parameters[NUM_SAMPLES]);

    if (s_sum != 0) {
      stats_ptr->sample_presence[k] = sp_count[k] / s_sum;
    } else {
      stats_ptr->sample_presence[k] = 0;
    }
  }

  /* free the temporary stats object */
  free(s_ptr);
}
```

```
/*************************************************************************
* mst_get_cov:  Get the statistic that measure the environmental
*               homogeneity (coefficient of variation)
* In:           Simulation pointer, stats pointer.
* Out:          None.
* Modifies:     Stats object.
* Allocation:   None.
*************************************************************************/
void mst_get_cov(sim_type * sim_ptr, stats_type * stats_ptr)
{
  int i, j, k;
  double bin[HOSTS];
  double sd[HOSTS];
  double mean[HOSTS];
  double stats[HOSTS][ARRAY_MAX];

  /* initialize the stats array */
  for (i = 0; i < sim_ptr->parameters[HOSTS]; i++) {
    for (j = 0; j < sim_ptr->parameters[ENV_MAX]; j++) {
      stats[i][j] = 0.0;
    }
  }

  /* initialize the bin array */
  for (k = COW; k < HOSTS; k++) {
    bin[k] = 0.0;
    sd[k] = 0.0;
    mean[k] = 0.0;
  }

  /* go through each bin in the environment */
  for (i = 0; i < sim_ptr->parameters[ENV_MAX]; i++) {

    /* clear the bin array from previous count */
    for (k = COW; k < HOSTS; k++) {
      bin[k] = 0.0;
    }

    /* go through each cell in a bin */
    for (j = 0; j < sim_ptr->parameters[EU_MAX]; j++) {

      if (sim_ptr->ground[i][j] != NULL) {
        if (sim_ptr->ground[i][j]->species == COW) {
          bin[COW] += 1;
        } else if (sim_ptr->ground[i][j]->species == DOG) {
          bin[DOG] += 1;
        } else if (sim_ptr->ground[i][j]->species == ELK) {
          bin[ELK] += 1;
        } else if (sim_ptr->ground[i][j]->species == GOOSE) {
          bin[GOOSE] += 1;
        } else if (sim_ptr->ground[i][j]->species == HUMAN) {
          bin[HUMAN] += 1;
        }
```

```
      }
    }

    /* store the stats for bin i */
    for (k = COW; k < HOSTS; k++) {
      stats[k][i] = bin[k];
    }

  }

  /* store the mean of each host */
  for (k = COW; k < HOSTS; k++) {
    mean[k] =
        gsl_stats_mean(&stats[k][0], 1, sim_ptr->parameters[ENV_MAX]);
  }

  /* store the standard deviation of each host */
  for (k = COW; k < HOSTS; k++) {
    sd[k] = gsl_stats_sd(&stats[k][0], 1, sim_ptr->parameters[ENV_MAX]);
  }

  /* store the stats in the stats object */
  for (k = COW; k < HOSTS; k++) {
    stats_ptr->env_density[k] = (100 * sd[k]) / mean[k];
  }
}

/*************************************************************************
* mst_clear_stats: Set all of the stats to 0.
* In:              Stats pointer.
* Out:             None.
* Modifies:        Stats object.
* Allocation:      None.
*************************************************************************/
void mst_clear_stats(stats_type * stats_ptr)
{
  int i = 0;

  for (i = COW; i < HOSTS; i++) {
    stats_ptr->total_cells[i] = 0.0;
    stats_ptr->total_marker[i] = 0.0;
    stats_ptr->sample_cells[i] = 0.0;
    stats_ptr->sample_marker[i] = 0.0;
    stats_ptr->sample_corrected[i] = 0.0;
    stats_ptr->sample_presence[i] = 0.0;
    stats_ptr->env_density[i] = 0.0;
  }
}

/*************************************************************************
* mst_print_mat: Print the marker matrix.
* In:            Simulation pointer.
* Out:           None.
* Modifies:      None.
```

```
* Allocation:    None.
*************************************************************************/
void mst_print_mat(sim_type * sim_ptr)
{
  int i, j;

  for (i = 0; i < HOSTS; i++) {
    for (j = 0; j < HOSTS; j++) {
      printf("%.4f ",
            (double) (sim_ptr->markers[i][j]) /
            (sim_ptr->parameters[M_BOUND]));
    }
    putchar('\n');
  }
}

/*************************************************************************
* mst_load_seeds: Load the random integer seeds from a file into
*                 the seed array of the simulation object.
* In:             Simulation pointer, name of the seed file.
* Out:            None.
* Modifies:       Simulation object (seeds array).
* Allocation:     None.
*************************************************************************/
void mst_load_seeds(sim_type * sim_ptr, char *s)
{
  FILE *fp;
  char buffer[BUF_LENGTH];
  int i = 0;

  if (!(fp = fopen(s, "r"))) {
    fprintf(stderr, "Error opening seeds file.\n");
    return;
  }

  while (fgets(buffer, BUF_LENGTH, fp) != NULL) {
    if (i < SEED_MAX) {
      sim_ptr->seeds[i++] = atoi(buffer);
    } else
      break;
  }

  if ((i - 1) < sim_ptr->parameters[REPS]) {
    fprintf(stderr, "Insufficient seeds to run simulation.\n");
    sim_ptr->run_sim = FALSE;
  }

  fclose(fp);
}

/*************************************************************************
* mst_store_seeds:  Store unused seeds back into the seeds file.
* In:               Simulation pointer, name of the file.
* Out:              None.
```

```
* Modifies:         None.
* Allocation:       None.
*************************************************************************/
void mst_store_seeds(sim_type * sim_ptr, char *s)
{
  FILE *fp;
  int i = sim_ptr->parameters[REPS];

  if (!(fp = fopen(s, "w"))) {
    fprintf(stderr, "Error opening seeds file.\n");
    return;
  }

  while (sim_ptr->seeds[i]) {
    if (i < SEED_MAX) {
      fprintf(fp, "%d\n", sim_ptr->seeds[i++]);
    } else
      break;
  }

  fclose(fp);
}

/*************************************************************************
* mst_print_stats: Print the simulation stats object.
* In:              Stats pointer.
* Out:             None.
* Modifies:        None.
* Allocation:      None.
*************************************************************************/
void mst_print_stats(stats_type * stats_ptr)
{
  printf("%.5f,%.5f,%.5f,%.5f,%.5f\n",
         stats_ptr->total_cells[COW],
         stats_ptr->total_cells[DOG],
         stats_ptr->total_cells[ELK],
         stats_ptr->total_cells[GOOSE],
         stats_ptr->total_cells[HUMAN]);
  printf("%.5f,%.5f,%.5f,%.5f,%.5f\n",
         stats_ptr->total_marker[COW],
         stats_ptr->total_marker[DOG],
         stats_ptr->total_marker[ELK],
         stats_ptr->total_marker[GOOSE],
         stats_ptr->total_marker[HUMAN]);
  printf("%.5f,%.5f,%.5f,%.5f,%.5f\n",
         stats_ptr->sample_cells[COW],
         stats_ptr->sample_cells[DOG],
         stats_ptr->sample_cells[ELK],
         stats_ptr->sample_cells[GOOSE],
         stats_ptr->sample_cells[HUMAN]);
  printf("%.5f,%.5f,%.5f,%.5f,%.5f\n",
         stats_ptr->sample_marker[COW],
         stats_ptr->sample_marker[DOG],
         stats_ptr->sample_marker[ELK],
```

```
          stats_ptr->sample_marker[GOOSE],
          stats_ptr->sample_marker[HUMAN]);
  printf("%.5f,%.5f,%.5f,%.5f,%.5f\n",
          stats_ptr->sample_corrected[COW],
          stats_ptr->sample_corrected[DOG],
          stats_ptr->sample_corrected[ELK],
          stats_ptr->sample_corrected[GOOSE],
          stats_ptr->sample_corrected[HUMAN]);
  printf("%.5f,%.5f,%.5f,%.5f,%.5f\n",
         stats_ptr->env_density[COW],
         stats_ptr->env_density[DOG],
         stats_ptr->env_density[ELK],
         stats_ptr->env_density[GOOSE],
         stats_ptr->env_density[HUMAN]);
}
```

Simulation Header File

```
/*************************************************************************
 *   Copyright (C) May 2007 by Mark Leach                                *
 *   mleach@eecs.wsu.edu                                                 *
 *   mst_sim.h                                                           *
 *   Header file for the mst module.                                     *
 *************************************************************************/

#include <gsl_rng.h>
#define HOSTS          5   /* number of kinds of hosts */
#define BUF_LENGTH  1250   /* file input buffer */
#define ARRAY_MAX   5000   /* maximum size of environment and eu arrays */
#define SEED_MAX    10000  /* maximum no. of seeds read from file */
#define SEARCHLIMIT 10000  /* maximum times to search for an empty spot */
#define STATS           6  /* # of rows of stats to store */
#define MAX_STATS     400  /* maximum number of stored stats */

enum { ENV_MAX, EU_MAX, SIM_TIME, M_BOUND, NUM_SAMPLES, DIST, REPS, S_TYPE,
  SD, PARAMS
};
enum { GAUSS, UNIFORM };
typedef enum { FALSE, TRUE } BOOL;
typedef enum { COW, DOG, ELK, GOOSE, HUMAN, NONE } species_type;

typedef struct {
  double total_cells[HOSTS]; /* % total in environment */
  double total_marker[HOSTS]; /* % total marker-bearing in environment */
  double sample_cells[HOSTS]; /* % total in sample space */
  double sample_marker[HOSTS]; /* % marker-bearing in sample space */
  double sample_corrected[HOSTS]; /* % corrected */
  double sample_presence[HOSTS]; /* % presence/absence measurement */
  double env_density[HOSTS]; /* % COV parameter */
} stats_type;

typedef struct {
  species_type species;
```

```
  int location;
  int shed_rate;
} host_type;

typedef struct {
  int age;
  int life_span;
  species_type species;
  species_type marker;
} eu_type;

typedef struct {
  BOOL run_sim;
  int seeds[SEED_MAX];
  int hosts[HOSTS];            /* number of hosts in the environment */
  int rates[HOSTS];            /* signal strengh of each type of host */
  int decay[HOSTS];            /* decay rates for each type of host cell */
  int parameters[PARAMS];      /* simulation parameters */
  int markers[HOSTS][HOSTS];   /* marker prevalence matrix */
  host_type *pop[ARRAY_MAX];   /* array of hosts */
  int sample_space[ARRAY_MAX]; /* sample space */
  eu_type *ground[ARRAY_MAX][ARRAY_MAX]; /* 2-D array of cells */
} sim_type;

BOOL mst_load_conf(sim_type * sim_ptr);
host_type *mst_build_host(sim_type * sim_ptr, species_type kind,
                          int location);
void mst_build_pop(sim_type * sim_ptr, const gsl_rng * r_ptr);
sim_type *mst_build_sim(void);
void mst_add_eu(sim_type * sim_ptr, int index, eu_type * eu_ptr);
species_type mst_get_marker(sim_type * sim_ptr, const gsl_rng * r_ptr,
                            species_type host);
int mst_get_life_span(sim_type * sim_ptr, const gsl_rng * r, int index);
void mst_generate(sim_type * sim_ptr, const gsl_rng * r_ptr);
void mst_die_off(sim_type * sim_ptr);
void mst_free_sim(sim_type * sim_ptr);
void mst_count_env(sim_type * sim_ptr, stats_type * stats_ptr);
void mst_count_markers(sim_type * sim_ptr, stats_type * stats_ptr);
int mst_get_index(sim_type * sim_ptr, const gsl_rng * r_ptr, int inx);
void mst_load_seeds(sim_type * sim_ptr, char *s);
void mst_store_seeds(sim_type * sim_ptr, char *s);
void mst_clear_env(sim_type * sim_ptr);
void mst_print_mat(sim_type * sim_ptr);
void mst_print_env(sim_type * sim_ptr);
void mst_get_cov(sim_type * sim_ptr, stats_type * stats_ptr);
void mst_solve_system(sim_type * sim_ptr, stats_type * stats_ptr);
void mst_clear_stats(stats_type * stats_ptr);
void mst_print_stats(stats_type * stats_ptr);
void mst_sample_env(sim_type * sim_ptr, const gsl_rng * r_ptr,
                    stats_type * stats_ptr);
void mst_sample_bin(sim_type * sim_ptr, stats_type * s_ptr, int loc);
void mst_get_sspace(sim_type * sim_ptr, const gsl_rng * r_ptr);
```